KOMPENDIUM

Optimierte Entscheidungsfindung mit Python

Von der mathematischen Modellierung zu praktischen intelligenten Lösungen für Betrieb, Technik und Finanzmärkte

Ein praxisorientiertes Kompendium für Entscheider mit Constraint Programming, Vektor- und Matrixmethoden sowie HiGHS, SciPy, Google OR-Tools und CVXPY.

Von Personaleinsatz-, Schicht- und Vertretungsplänen, Fahrtrouten, Logistik und Maschinen- und Netzwerkauslastung über Energieverteilung und Ressourcenplanung bis zur Portfoliooptimierung und besseren Entscheidungen an den Finanzmärkten – mit Python modellieren, optimieren und belegbar fundierte Entscheidungen treffen.

HiGHS | SciPy | Google OR-Tools | CVXPY

Autor / Herausgeber: Dieter Schlüter

Stand: 8. September 2026

Optimierte Entscheidungsfindung mit Python

Von der mathematischen Modellierung zu praktischen intelligenten Lösungen für Betrieb, Technik und Finanzmärkte

Ein praxisorientiertes Kompendium für Entscheider mit Constraint Programming, Vektor- und Matrixmethoden sowie HiGHS, SciPy, Google OR-Tools und CVXPY.

Von **Personaleinsatz-, Schicht- und Vertretungsplänen, Fahrtrouten, Logistik und Maschinen- und Netzwerkauslastung** über **Energieverteilung und Ressourcenplanung** bis zur **Portfoliooptimierung und besseren Entscheidungen an den Finanzmärkten** — mit Python modellieren, optimieren und belegbar fundierte Entscheidungen treffen.

Autor / Herausgeber: Dieter Schlüter

<dieter(dot)schlueter(atsign)linix(dot)de>

Stand: 8. September 2026

Inhaltsverzeichnis

Optimierte Entscheidungsfindung mit Python **2**

Von der mathematischen Modellierung zu praktischen intelligenten Lösungen für Betrieb, Technik und Finanzmärkte . . . 2

Vorwort . . . 20

Ziel der Masterclass . . . 20

Vorausgesetzte Kenntnisse . . . 20

Benötigte Ressourcen . . . 21

Setup & Installation . . . 21

Installationstest . . . 21

Wie dieses Buch aufgebaut ist . . . 25

Die Lernelemente . . . 25

Der Dreischritt jedes Kapitels . . . 25

Für wen dieses Buch geschrieben ist . . . 26

Für Entscheider und Manager — der Geschäftsfall . . . 26

Für Ingenieure und Produktionsplaner — die Physik des Systems . . . 27

Für Investoren und Finanzanalysten — Risiko und Ertrag . . . 27

Für Studierende und Data Scientists — die Methodik . . . 28

Lernpfade . . . 28

Pfad A — Vollständiger Lehrgang (100–140 Stunden) . . . 28

Pfad B — Planung, Disposition, Personaleinsatz (ca. 25 Stunden) . . . 28

Pfad C — Logistik und Tourenplanung (ca. 25 Stunden) . . . 29

Pfad D — Quantitative Finance (ca. 30 Stunden) . . . 29

Pfad E — Ich habe morgen ein konkretes Problem (2–4 Stunden) . . . 29

Pfad F — Vom Prototyp in den Betrieb (ca. 15 Stunden) . . . 29

Verzeichnis der Beispielprogramme . . . 29

Was dieses Buch nicht ist . . . 32

Ein Wort zur Arbeitsweise . . . 33

Notation und Abkürzungen **33**

1. Die Symbole der Optimierung . . . 33

1.1 Grundgrößen jedes Modells . . . 33

1.2 Mengen und Zahlbereiche . . . 35

1.3 Operatoren und Schreibweisen . . . 35

1.4 Symbole der Finanzkapitel (Teil IV) . . . 36

1.5 Symbole der Verfahrenskapitel (Teil II und III) . . . 37

2. Abkürzungen — vollständig ausgeschrieben . . . 39

2.1 Problemklassen und Verfahren . . . 39

2.2 Probleme mit Eigennamen . . . 41

2.3 Finanz- und Risikokennzahlen . . . 41

2.4 Software, Solver und Schnittstellen . . . 42

2.5 Begriffe aus dem Solver-Alltag . . . 44

3. Konventionen in diesem Buch 45

Teil I: Grundlagen des Operations Research **45**

Kapitel 1: Einführung in Operations Research — Vom Ursprung zur mathematischen Entscheidungsfindung **45**

1.1 In 5 Minuten gelöst 46
1.2 Lernziele 47
1.3 Was ist Operations Research wirklich? 47
Die drei Stufen der Analytik 48
1.4 Warum Ausprobieren scheitert — mit eigener Rechnung 49
1.5 Historischer Kontext und Evolution 53
1.6 Die vier universellen Bausteine jedes OR-Problems 53
Baustein 1 — Entscheidungsvariablen (x) 54
Baustein 2 — Parameter / Eingabedaten (c, A, b) 54
Baustein 3 — Zielfunktion ($f(x)$) 55
Baustein 4 — Nebenbedingungen (*Constraints*) 55
Die vier Bausteine als eine Formel 56
Die Bausteine in der Praxis: eine Vorlage 57
1.7 Erstes Python-Vollbeispiel: Ressourcenallokation im Rechenzentrum 60
Szenario 60
Mathematische Formulierung 60
Umsetzung mit Google OR-Tools 61
1.8 Von Excel zu Python: Ihre Daten liegen schon da 65
1.9 Übungsaufgaben 69
1.10 Finde den Denkfehler 70
1.11 Micro-Quiz 71
1.12 Selbsttest 72
1.13 Zusammenfassung 72

Kapitel 2: Das mathematische Fundament — Vektoren, Matrizen, Konvexität **73**

2.1 In 5 Minuten gelöst 73
2.2 Lernziele 74
2.3 Warum überhaupt Vektoren und Matrizen? 75
Die Bausteine 75
Die kanonische Standardform 76
Dasselbe in Python 77
2.4 Der zulässige Lösungsraum und das Polyeder 79
Der Fundamentalsatz der linearen Optimierung 79
2.5 Konvexität: die Grenze zwischen leicht und schwer 81
Konvexe Menge 81
Konvexe Funktion 82
Der zentrale Satz 82
Was ist konvex, was nicht? 83

Konvexität sichtbar machen . . . 83
2.6 Geometrische Visualisierung des Lösungsraums . . . 86
2.7 Wenn die Zahlen nicht zusammenpassen: Kondition und Skalierung . . . 90
Die geometrische Vorstellung: zwei fast parallele Linien . . . 90
Woher schlechte Kondition im Alltag kommt . . . 91
Ruiz-Equilibrierung: das Modell gesundrechnen . . . 92
Das Experiment . . . 92
2.8 Übungsaufgaben . . . 99
2.9 Finde den Denkfehler . . . 100
2.10 Micro-Quiz . . . 101
2.11 Selbsttest . . . 101
2.12 Zusammenfassung . . . 102

Kapitel 3: Das Python-Ökosystem für OR — Solver, Bindings und Modellierungsschichten 102
3.1 In 5 Minuten gelöst . . . 103
3.2 Lernziele . . . 104
3.3 Die Zwei-Schichten-Architektur . . . 104
3.4 Die Werkzeuge im Vergleich . . . 105
Die Entscheidung in drei Fragen . . . 106
3.5 Ein System — vier Programmieransätze . . . 108
Wie die Isolation aussieht, wenn sie tragen soll . . . 110
Was das CSR-Format bedeutet . . . 115
3.6 Wann lohnt sich welche Ebene? . . . 116
3.7 Modellierungsschichten für große Modelle: Pyomo und Linopy . . . 116
Pyomo: die algebraische Denkweise . . . 116
Linopy: eine Zeile, zehntausend Nebenbedingungen . . . 117
Die Schichten im Überblick . . . 117
3.8 Wo die Zeit wirklich hingeht: vektorisierte Modellgenerierung . . . 122
Vier Stufen an einem Transportproblem . . . 122
3.9 Übungsaufgaben . . . 129
3.10 Finde den Denkfehler . . . 130
3.11 Micro-Quiz . . . 131
3.12 Selbsttest . . . 131
3.13 Zusammenfassung . . . 131

Kapitel 4: Vom Management-Wunsch zum Modell 132
4.1 In 5 Minuten gelöst . . . 132
4.2 Lernziele . . . 134
4.3 Ein Satz ist noch kein Modell . . . 134
4.4 Derselbe Datensatz, fünf Modelle . . . 135
Teil 1: Drei Ziele, drei Pläne . . . 142
Teil 2: Ein Satz, drei Lesarten . . . 143
Teil 3: Hart oder weich . . . 143
Teil 4: Ein Quartal später . . . 144

4.5 Der Gesprächsleitfaden . . . 144
4.6 Deutsch — OR: ein Übersetzungslexikon . . . 145
Fünf Wendungen, bei denen die Übersetzung eine Entscheidung erzwingt . . . 146
Und die Gegenrichtung: OR — Deutsch . . . 147
4.7 Übungsaufgaben . . . 147
4.8 Finde den Denkfehler . . . 148
4.9 Micro-Quiz . . . 148
4.10 Selbsttest . . . 149
4.11 Zusammenfassung . . . 149

Synthese Teil I — Grundlagen auf einen Blick **150**
Was welches Kapitel klärt . . . 150
Die Reihenfolge ist die Aussage . . . 150
Wenn Sie nur eines mitnehmen . . . 151

Teil II: Die Kernverfahren der deterministischen Optimierung **151**

Kapitel 5: Lineare Programmierung — Simplex, Dualität und Schattenpreise **152**
5.1 In 5 Minuten gelöst . . . 152
5.2 Lernziele . . . 154
5.3 Die Standardform und Schlupfvariablen . . . 154
5.4 Der Simplex-Algorithmus Schritt für Schritt . . . 155
Der Ablauf . . . 155
5.5 Das Simplex-Tableau in Python . . . 157
5.6 Dualität und Schattenpreise . . . 162
Der starke Dualitätssatz . . . 163
Der Schattenpreis als Ableitung . . . 166
5.7 Die Vorzeichenfalle bei Schattenpreisen . . . 167
5.8 Praxisfall: Sensitivitätsanalyse mit korrekten Schattenpreisen . . . 167
5.9 Wann Schattenpreise lügen: Entartung und Toleranzen . . . 171
Fall 1: Entartung . . . 171
Fall 2: Toleranzen . . . 172
Die ehrliche Auskunft: Schattenpreis-Spannen . . . 173
5.10 Übungsaufgaben . . . 179
5.11 Finde den Denkfehler . . . 180
5.12 Micro-Quiz . . . 180
5.13 Selbsttest . . . 181
5.14 Zusammenfassung . . . 181

Kapitel 6: Gemischt-ganzzahlige Optimierung — Diskrete Entscheidungen und Branch-and-Bound **182**
6.1 In 5 Minuten gelöst . . . 182
6.2 Lernziele . . . 184
6.3 Warum Runden fundamental scheitert . . . 184

6.4 Branch-and-Bound . . . 188
Die drei Phasen . . . 189
6.5 Modellierungstricks: Big-M und logische Bedingungen . . . 190
Muster 1 — Fixkosten / Aktivierungsschalter . . . 190
Muster 2 — Entweder-Oder (*disjunctive constraints*) . . . 190
Muster 3 — Wenn-Dann (Implikation) . . . 190
Muster 4 — Kardinalität („höchstens K aus N“) . . . 190
Muster 5 — Semikontinuierlich („entweder 0 oder mindestens L“) . . . 190
6.6 Beispiel: Das Rucksackproblem . . . 191
6.7 Praxisfall: Portfolio mit Ordergebühren und Kardinalitätsgrenze . . . 193
6.8 Wenn der Solver nicht fertig wird: Gap, Zeitlimit und Warm-Start . . . 200
Die zwei Zahlen, die zählen . . . 200
Alle Statusfälle behandeln . . . 201
Warm-Starts: was sie können und was nicht . . . 207
6.9 Übungsaufgaben . . . 211
6.10 Finde den Denkfehler . . . 212
6.11 Micro-Quiz . . . 219
6.12 Selbsttest . . . 219
6.13 Zusammenfassung . . . 219

Kapitel 7: Constraint Programming mit CP-SAT — Logik, Scheduling und Zuweisung 220
7.1 In 5 Minuten gelöst . . . 221
7.2 Lernziele . . . 222
7.3 Ein anderes Denkmodell . . . 222
Was Propagation leistet . . . 223
7.4 Globale Constraints — die Bausteine von CP-SAT . . . 225
7.5 Praxisfall: Dynamisches Vertretungssystem . . . 227
7.6 Intervallvariablen: Job-Shop-Scheduling . . . 234
7.7 Parallele Suche: was `num_workers` wirklich bewirkt . . . 238
Was die Messung zeigt . . . 244
Was daraus für Tests folgt . . . 245
7.8 Die fünf Antworten von CP-SAT . . . 245
7.9 Übungsaufgaben . . . 253
7.10 Finde den Denkfehler . . . 254
7.11 Micro-Quiz . . . 258
7.12 Selbsttest . . . 259
7.13 Zusammenfassung . . . 259

Kapitel 8: Graphen, Flüsse und Touren — Min-Cost-Flow, Matching und VRP 260
8.1 In 5 Minuten gelöst . . . 261
8.2 Lernziele . . . 262
8.3 Graphen als Modellsprache . . . 263
Das Minimum-Cost-Flow-Problem (MCNFP) . . . 263
8.4 Bipartites Matching: das Zuordnungsproblem . . . 267

Der Satz von Birkhoff und von Neumann . . . 267
8.5 Das Vehicle Routing Problem mit Zeitfenstern . . . 270
Kurzzyklen verhindern . . . 270
Praxisbeispiel: Flotten-Routing . . . 272
8.6 Übungsaufgaben . . . 276
8.7 Finde den Denkfehler . . . 276
8.8 Micro-Quiz . . . 282
8.9 Selbsttest . . . 282
8.10 Zusammenfassung . . . 283

Kapitel 9: Metaheuristiken — wenn der exakte Solver aussteigt 283
9.1 In 5 Minuten gelöst . . . 284
9.2 Lernziele . . . 285
9.3 Die Aufgabe: Rüstzeiten an der Lackieranlage . . . 285
Die Instanz . . . 286
9.4 Lokale Suche: der Zug und seine Kosten . . . 287
9.5 Simulated Annealing . . . 287
Was die Messung zeigt — und was sie nicht zeigt . . . 296
9.6 Ab wann lohnt es sich? . . . 297
Die Spalte, die man nicht weglassen darf . . . 303
9.7 Large Neighborhood Search . . . 304
Das Fenster ist die eine Stellschraube . . . 312
Das Gesamtbild . . . 312
9.8 Was dieses Kapitel nicht behandelt . . . 313
9.9 Übungsaufgaben . . . 313
9.10 Finde den Denkfehler . . . 314
9.11 Micro-Quiz . . . 314
9.12 Selbsttest . . . 315
9.13 Zusammenfassung . . . 315

Kapitel 10: Spaltengenerierung — das Modell umbauen statt die Lösung raten 316
10.1 In 5 Minuten gelöst . . . 316
10.2 Lernziele . . . 317
10.3 Warum das naheliegende Modell scheitert . . . 318
10.4 Rechnen mit Mustern, ohne sie aufzuschreiben . . . 319
10.5 Das Programm . . . 320
10.6 Was die Zahlen zeigen . . . 328
Die Musterzahl explodiert, der Bedarf nicht . . . 328
Der unbequeme Teil: Es lohnt sich nicht immer . . . 328
10.7 Übungsaufgaben . . . 329
10.8 Finde den Denkfehler . . . 329
10.9 Micro-Quiz . . . 330
10.10 Selbsttest . . . 330
10.11 Zusammenfassung . . . 331

Synthese Teil II — Die Kernverfahren nebeneinander **331**
Die Entscheidungsmatrix . . . 331
Was dieser Teil gemessen hat . . . 332
Drei Fehler, die dieser Teil verhindert . . . 332
Wenn Sie nur eines mitnehmen . . . 333

Teil III: Nichtlinearität, Unsicherheit und mehrperiodige Dynamik **333**

Kapitel 11: Quadratische und nichtlineare Optimierung — KKT, Lagrange, Konvexität **334**
11.1 In 5 Minuten gelöst . . . 334
11.2 Lernziele . . . 336
Auffrischung: Gradient in einer Minute . . . 336
11.3 Das quadratische Programm (QP) . . . 336
Konvexität: die präzise Aussage . . . 337
11.4 Die Karush-Kuhn-Tucker-Bedingungen . . . 339
Die vier KKT-Bedingungen . . . 340
11.5 Nichtlineare Optimierung mit `scipy.optimize.minimize` . . . 344
Praxisfall: Entropie-maximierte Kapitalallokation . . . 344
11.6 Jenseits der Konvexität: lokale Optima, Multistart und MINLP . . . 349
Woher Nicht-Konvexität im Alltag kommt . . . 349
Was ein lokales Optimum praktisch bedeutet . . . 350
Die Zahlen im Klartext . . . 356
Ausblick: MINLP und globale Solver . . . 356
11.7 Übungsaufgaben . . . 357
11.8 Finde den Denkfehler . . . 358
11.9 Micro-Quiz . . . 359
11.10 Selbsttest . . . 360
11.11 Zusammenfassung . . . 360

Kapitel 12: Optimierung unter Unsicherheit — Monte-Carlo, Stochastik, Robustheit **361**
12.1 In 5 Minuten gelöst . . . 361
12.2 Lernziele . . . 363
12.3 Der Fluch des Durchschnitts . . . 363
Die vier Ansätze im Überblick . . . 366
12.4 Monte-Carlo-Simulation . . . 367
12.5 Zweistufige stochastische Programmierung . . . 370
12.6 Robuste Optimierung: gegen den Worst Case absichern . . . 374
12.7 Wahrscheinlichkeitsbeschränkungen: „mit 95 % Sicherheit“ . . . 377
Weg 1: Normalverteilung — die Bedingung wird ein Kegel . . . 377
Weg 2: Szenarien — Big-M ohne Verteilungsannahme . . . 378
Der Fall: ein Kraftwerkspark mit 500 MW Zusage . . . 378
12.8 Übungsaufgaben . . . 385
12.9 Finde den Denkfehler . . . 386
12.10 Micro-Quiz . . . 387

12.11 Selbsttest . . . 387
12.12 Zusammenfassung . . . 388

Kapitel 13: Dynamische Programmierung — Die Bellman-Gleichung und Order-Execution 388
13.1 In 5 Minuten gelöst . . . 389
13.2 Lernziele . . . 390
13.3 Das Bellmansche Optimalitätsprinzip . . . 391
Die vier Bausteine eines DP-Modells . . . 391
Die Bellman-Gleichung . . . 391
13.4 Das Almgren-Chriss-Problem: optimale Orderausführung . . . 395
13.5 Der Fluch der Dimensionalität . . . 402
13.6 Übungsaufgaben . . . 403
13.7 Finde den Denkfehler . . . 403
13.8 Micro-Quiz . . . 405
13.9 Selbsttest . . . 405
13.10 Zusammenfassung . . . 405

Kapitel 14: Mehrere Ziele — Pareto-Fronten statt Gewichte 406
14.1 In 5 Minuten gelöst . . . 406
14.2 Lernziele . . . 408
14.3 Wann ein Kompromiss gut ist . . . 408
14.4 Der Reflex: alles in ein Ziel rühren . . . 409
14.5 Das Programm . . . 409
14.6 Was die gewichtete Summe nicht sieht . . . 417
Das Gewicht ist keine Feineinstellung . . . 418
14.7 Das ε-Constraint-Verfahren . . . 418
Wenn es eine klare Rangfolge gibt: lexikografisch . . . 418
14.8 Wie man die Front vorlegt . . . 419
14.9 Übungsaufgaben . . . 420
14.10 Finde den Denkfehler . . . 420
14.11 Micro-Quiz . . . 421
14.12 Selbsttest . . . 421
14.13 Zusammenfassung . . . 421

Kapitel 15: Predict-then-Optimize — die bessere Prognose, die schlechtere Entscheidung 422
15.1 In 5 Minuten gelöst . . . 422
15.2 Lernziele . . . 423
15.3 Die Naht zwischen zwei Abteilungen . . . 424
15.4 Das Programm . . . 424
15.5 Der Befund . . . 431
Warum ein pauschaler Zuschlag zu kurz greift . . . 432
15.6 Eine Messfalle, in die der Autor selbst getappt ist . . . 433
15.7 Wie weit das Verfahren reicht . . . 433
15.8 Übungsaufgaben . . . 434

15.9 Finde den Denkfehler . . . 435
15.10 Micro-Quiz . . . 435
15.11 Selbsttest . . . 436
15.12 Zusammenfassung . . . 436

Synthese Teil III — Wenn die Idealwelt nicht gilt **436**
Welche Annahme fällt weg? . . . 437
Was dieser Teil gemessen hat . . . 437
Drei Fehler, die dieser Teil verhindert . . . 438
Wenn Sie nur eines mitnehmen . . . 438

Teil IV: Anwendungen — Energiewirtschaft und Finanzmärkte **438**

Kapitel 16: Die Strukturbrücke — dieselbe Mathematik, zwei Welten **439**
16.1 In 5 Minuten gelöst . . . 439
16.2 Lernziele . . . 441
16.3 Das Programm . . . 441
16.4 Die zwei Brücken im Einzelnen . . . 449
Erste Brücke: Allokation . . . 449
Zweite Brücke: Absicherung gegen den schlechtesten Fall . . . 450
Wo die Lösungen sich unterscheiden . . . 450
16.5 Wo die Brücke endet . . . 451
16.6 Übungsaufgaben . . . 452
16.7 Finde den Denkfehler . . . 452
16.8 Micro-Quiz . . . 453
16.9 Selbsttest . . . 453
16.10 Zusammenfassung . . . 453

Kapitel 17: Supply-Chain und Energieeinsatz unter Unsicherheit **454**
17.1 In 5 Minuten gelöst . . . 454
17.2 Lernziele . . . 455
17.3 Die Aufgabe . . . 456
Die Zeitstruktur der Entscheidung . . . 456
17.4 Das Programm . . . 457
17.5 Der Befund . . . 466
17.6 Was Versorgungssicherheit kostet . . . 467
17.7 Übungsaufgaben . . . 468
17.8 Finde den Denkfehler . . . 469
17.9 Micro-Quiz . . . 469
17.10 Selbsttest . . . 470
17.11 Zusammenfassung . . . 470

Kapitel 18: Finanzdaten-Modellierung — Renditen, Kovarianz und Shrinkage **470**
18.1 In 5 Minuten gelöst . . . 471
18.2 Lernziele . . . 472

18.3 Diskrete und logarithmische Renditen . . . 473
Diskrete Rendite . . . 473
Logarithmische Rendite . . . 473
18.4 Das Schätzfehler-Problem . . . 475
18.5 Ledoit-Wolf-Shrinkage . . . 479
Welches Ziel **F**? . . . 479
18.6 Praxis: Datenpipeline mit korrekter Spaltenreihenfolge . . . 480
18.7 Übungsaufgaben . . . 484
18.8 Finde den Denkfehler . . . 485
18.9 Micro-Quiz . . . 489
18.10 Selbsttest . . . 490
18.11 Zusammenfassung . . . 490

Kapitel 19: Die moderne Portfoliotheorie nach Markowitz **490**
19.1 In 5 Minuten gelöst . . . 491
19.2 Lernziele . . . 492
19.3 Warum Diversifikation funktioniert . . . 493
19.4 Das Mean-Variance-Modell . . . 495
Die Effizienzgrenze und das Tangentialportfolio . . . 496
Die Korn-Transformation . . . 498
19.5 Institutionelle Nebenbedingungen . . . 498
Was diese Regeln kosten . . . 499
19.6 Vollimplementierung mit CVXPY . . . 500
19.7 Übungsaufgaben . . . 506
19.8 Finde den Denkfehler . . . 507
19.9 Micro-Quiz . . . 512
19.10 Selbsttest . . . 512
19.11 Zusammenfassung . . . 512

Kapitel 20: Tail-Risiko, CVaR und Transaktionskosten **513**
20.1 In 5 Minuten gelöst . . . 513
20.2 Lernziele . . . 515
20.3 Die zwei Schwächen des Markowitz-Modells . . . 515
20.4 Value at Risk und Conditional Value at Risk . . . 518
Das Rockafellar-Uryasev-Theorem (2000) . . . 520
20.5 Transaktionskosten über die L_1-Norm . . . 521
20.6 Implementierung: CVaR-Portfolio mit Reibung . . . 522
20.7 Übungsaufgaben . . . 527
20.8 Finde den Denkfehler . . . 528
20.9 Micro-Quiz . . . 529
20.10 Selbsttest . . . 530
20.11 Zusammenfassung . . . 530

Kapitel 21: Die vollständige quantitative Handelsmaschine **530**

21.1 In 5 Minuten gelöst . . . 531
21.2 Lernziele . . . 532
21.3 Die Architektur . . . 532
21.4 Rebalancing-Termine richtig bestimmen . . . 533
21.5 Die Engine . . . 535
21.6 Die fünf Selbsttäuschungen des Backtestens . . . 542
21.7 Vom Backtest zum Betrieb . . . 546
21.8 Übungsaufgaben . . . 546
21.9 Finde den Denkfehler . . . 547
21.10 Micro-Quiz . . . 552
21.11 Selbsttest . . . 552
21.12 Zusammenfassung . . . 552

Synthese Teil IV — Zwei Domänen, dieselbe Mathematik **553**
Dieselbe Struktur, andere Namen . . . 553
Was dieser Teil gemessen hat . . . 554
Drei Fehler, die dieser Teil verhindert . . . 554
Wenn Sie nur eines mitnehmen . . . 554

Teil V: Praxis **555**

Kapitel 22: Praxisfallen und der Weg zum produktiven Einsatz **555**
22.1 In 5 Minuten gelöst . . . 555
22.2 Lernziele . . . 556
22.3 Die fünf typischen Praxisfallen . . . 557
Falle 1 — Infeasibility . . . 557
Falle 2 — Der Black-Box-Effekt . . . 560
Falle 3 — Regimewechsel und Schätzfehler . . . 563
Falle 4 — Lookahead- und Survivorship-Bias . . . 563
Falle 5 — Laufzeitexplosion . . . 563
22.4 Constraint Attribution: welche Bedingung kostet wie viel? . . . 563
Drei Befunde, und keiner steht im Solverergebnis . . . 572
Vom Zahlenwerk zur Vorlage . . . 573
22.5 Architektur einer produktionsreifen OR-Plattform . . . 573
22.6 Der gemeinsame Unterbau: `or_kern.py` . . . 574
Die vier Schichten . . . 576
Ein Enum für fünf Bibliotheken . . . 577
22.7 Der Solverwechsel in der Praxis . . . 589
22.8 Eine Checkliste vor dem Produktivgang . . . 599
22.9 Weiterführende Literatur und Roadmap . . . 599
22.10 Übungsaufgaben . . . 600
22.11 Finde den Denkfehler . . . 601
22.12 Micro-Quiz . . . 607
22.13 Selbsttest . . . 607

22.14 Zusammenfassung . . . 608

Kapitel 23: Testen, Messen, Ausliefern **608**
23.1 In 5 Minuten gelöst . . . 609
23.2 Lernziele . . . 610
23.3 Warum Optimierungsmodelle schwer zu testen sind . . . 610
23.4 Die Testsuite . . . 611
23.5 Wer testet die Tests? . . . 618
Der erste Lauf fand zwei Lücken . . . 622
23.6 Ein Vergleich, dem man glauben kann . . . 624
Die interessanteste Spalte heißt „Anteil” . . . 630
23.7 Das Modell als Dienst . . . 630
Threads oder Prozesse? Eine Messung, keine Meinung . . . 640
Was im echten Betrieb dazukommt . . . 641
23.8 Übungsaufgaben . . . 641
23.9 Finde den Denkfehler . . . 642
23.10 Micro-Quiz . . . 642
23.11 Selbsttest . . . 643
23.12 Zusammenfassung . . . 643

Synthese Teil V — Vom rechnenden Modell zum benutzten System **643**
Was schiefgeht, und was dagegen hilft . . . 643
Was dieser Teil gemessen hat . . . 644
Drei Fehler, die dieser Teil verhindert . . . 645
Wenn Sie nur eines mitnehmen . . . 645

Projektwerkstatt — elf eigene Anwendungen **645**
Wie Sie ein Projekt bearbeiten . . . 645
P1 — Vertretungsplaner für eine Schule . . . 646
P2 — Schichtplanung für ein Pflegeteam . . . 647
P3 — Tourenplanung für einen Lieferdienst . . . 647
P4 — Standort- und Lagernetzplanung . . . 648
P5 — Produktionsplanung mit Rüstkosten . . . 648
P6 — Portfolio-Rebalancer für ein Privatdepot . . . 648
P7 — Risikoreport mit CVaR und Stresstests . . . 649
P9 — Wenn der Solver aussteigt: Tourenplanung in Echtgröße . . . 649
P10 — Zwei Ziele, eine Entscheidung: Kosten gegen CO□ . . . 650
P11 — Vom Skript zum Dienst: das Modell übergeben . . . 650
P8 — Freies Projekt aus Ihrem eigenen Umfeld . . . 651
Eine Bitte zum Schluss . . . 652

Anhang A: Lösungen zu allen Übungsaufgaben **652**
A.1 Lösungen zu Kapitel „Einführung in Operations Research — Vom Ursprung zur mathematischen Entscheidungsfindung“ . . . 652

Finde den Denkfehler — Die Schreinerei verdoppelt ihren Gewinn 653
Micro-Quiz 654
Selbsttest 654
A.2 Lösungen zu Kapitel „Das mathematische Fundament — Vektoren, Matrizen, Konvexität“ 655
Finde den Denkfehler — Die unauffällige Transposition 655
Micro-Quiz 656
Selbsttest 657
A.3 Lösungen zu Kapitel „Das Python-Ökosystem für OR — Solver, Bindings und Modellierungsschichten“ 657
Finde den Denkfehler — Der Solver, der angeblich dreimal schneller ist 658
Micro-Quiz 659
Selbsttest 659
A.4 Lösungen zu Kapitel „Vom Management-Wunsch zum Modell“ 659
Finde den Denkfehler — „Die Engpassmaschine soll zu mindestens 92 % ausgelastet sein“ 661
Micro-Quiz 662
Selbsttest 662
A.5 Lösungen zu Kapitel „Lineare Programmierung — Simplex, Dualität und Schattenpreise“ 663
Finde den Denkfehler — Die 380 000-Euro-Maschine 663
Micro-Quiz 664
Selbsttest 665
A.6 Lösungen zu Kapitel „Gemischt-ganzzahlige Optimierung — Diskrete Entscheidungen und Branch-and-Bound“ 665
Finde den Denkfehler — Elf Lager, die keine Fixkosten kosten 666
Micro-Quiz 667
Selbsttest 667
A.7 Lösungen zu Kapitel „Constraint Programming mit CP-SAT — Logik, Scheduling und Zuweisung“ 667
Finde den Denkfehler — Die Betriebsvereinbarung, die niemanden interessiert 669
Micro-Quiz 670
Selbsttest 670
A.8 Lösungen zu Kapitel „Graphen, Flüsse und Touren — Min-Cost-Flow, Matching und VRP“ 671
Finde den Denkfehler — Die vergessene Dimension 672
Micro-Quiz 673
Selbsttest 673
A.9 Lösungen zu Kapitel „Metaheuristiken — wenn der exakte Solver aussteigt“ 673
Finde den Denkfehler — „Unsere Heuristik ist 6 % besser“ 674
Micro-Quiz 675
Selbsttest 675
A.10 Lösungen zu Kapitel „Spaltengenerierung“ 675
Finde den Denkfehler — „Die LP-Lösung sagt 72,92 — also runden wir auf“ 677
Micro-Quiz 677
Selbsttest 678

A.11 Lösungen zu Kapitel „Quadratische und nichtlineare Optimierung — KKT, Lagrange, Konvexität“ . . . 678
Finde den Denkfehler — Die Kovarianzmatrix aus dem Controlling . . . 679
Micro-Quiz . . . 680
Selbsttest . . . 680
A.12 Lösungen zu Kapitel „Optimierung unter Unsicherheit — Monte-Carlo, Stochastik, Robustheit“ . . . 681
Finde den Denkfehler — Warum jedes Projekt zu spät fertig wird . . . 682
Micro-Quiz . . . 683
Selbsttest . . . 683
A.13 Lösungen zu Kapitel „Dynamische Programmierung — Die Bellman-Gleichung und Order-Execution“ . . . 684
Finde den Denkfehler — Der Zustand, der zu wenig weiß . . . 685
Micro-Quiz . . . 686
Selbsttest . . . 686
A.14 Lösungen zu Kapitel „Mehrere Ziele — Pareto-Fronten statt Gewichte“ . . . 687
Finde den Denkfehler — „Wir haben die Gewichte sauber kalibriert“ . . . 688
Micro-Quiz . . . 688
Selbsttest . . . 689
A.15 Lösungen zu Kapitel „Predict-then-Optimize“ . . . 689
Finde den Denkfehler — „Wir haben die Prognose um 18 % verbessert“ . . . 690
Micro-Quiz . . . 691
Selbsttest . . . 691
A.16 Lösungen zu Kapitel „Die Strukturbrücke — dieselbe Mathematik, zwei Welten“ . . . 691
Finde den Denkfehler — „Das ist doch dasselbe Problem“ . . . 692
Micro-Quiz . . . 693
Selbsttest . . . 693
A.17 Lösungen zu Kapitel „Supply-Chain und Energieeinsatz unter Unsicherheit“ . . . 694
Finde den Denkfehler — „Wir rechnen mit dem P50-Szenario“ . . . 695
Micro-Quiz . . . 695
Selbsttest . . . 696
A.18 Lösungen zu Kapitel „Finanzdaten-Modellierung — Renditen, Kovarianz und Shrinkage“ 696
Finde den Denkfehler — Das risikofreie Portfolio . . . 697
Micro-Quiz . . . 698
Selbsttest . . . 698
A.19 Lösungen zu Kapitel „Die moderne Portfoliotheorie nach Markowitz“ . . . 699
Finde den Denkfehler — Zwölf gleiche Anlagen, ein sehr ungleiches Portfolio . . . 700
Micro-Quiz . . . 701
Selbsttest . . . 701
A.20 Lösungen zu Kapitel „Tail-Risiko, CVaR und Transaktionskosten“ . . . 702
Finde den Denkfehler — Das Risikobudget, das durch Diversifikation stieg . . . 702
Micro-Quiz . . . 704
Selbsttest . . . 704

A.21 Lösungen zu Kapitel „Die vollständige quantitative Handelsmaschine“ 704
Finde den Denkfehler — Die Strategie mit dem hochsignifikanten Ergebnis 705
Micro-Quiz 706
Selbsttest 707
A.22 Lösungen zu Kapitel „Praxisfallen und der Weg zum produktiven Einsatz“ 707
Finde den Denkfehler — Das Modell, das seit einem Jahr nicht mehr optimiert 708
Micro-Quiz 709
Selbsttest 709
A.23 Lösungen zu Kapitel „Testen, Messen, Ausliefern“ 710
Finde den Denkfehler — „Die Suite ist grün, das Modell stimmt” 711
Micro-Quiz 712
Selbsttest 712

Anhang B: Katalog der Modellierungsmuster **712**
Übersicht 713
Logische Schalter 714
B1 — Aktivierungsschalter (Fixkosten) 714
B2 — Semikontinuierliche Variable 715
B3 — Implikation 715
B4 — Entweder-Oder (disjunktive Bedingung) 715
B5 — Exklusiv-Oder 715
B6 — Kardinalität 715
B7 — Bedingte Kopplung (Konjunktion) 716
Mengen und Grenzen 716
B8 — Weiche Grenze mit Strafkosten 716
B9 — Gestaffelte Preise (stückweise linear) 716
B10 — Absolutbetrag / Abweichung 716
B11 — Min/Max in der Zielfunktion 717
B12 — Verhältnis-Bedingung 717
B25 — Mindestabnahmemenge über einen Zeitraum 717
B27 — Budgetlimit 718
Zeit und Reihenfolge 718
B13 — Vorrangbeziehung 718
B14 — Nichtüberlappung 718
B15 — Kumulative Ressource 719
B16 — Gleitendes Fenster 719
B17 — Umrüst- bzw. Wechselkosten 719
B26 — Rüstzeit als Kapazitätsverbrauch 719
Robustheit und Diagnose 720
B18 — Schlupfvariablen gegen Unlösbarkeit 720
B19 — Hierarchische Ziele (lexikografisch) 720
B20 — Symmetriebrechung 720
B21 — Worst-Case-Abzug (robuste Formulierung) 721

Netzwerke . . . 721
B22 — Flusserhaltung . . . 721
B23 — Zuordnung 1:1 . . . 721
B24 — Subtour-Eliminierung . . . 721
Ein Wort zur Auswahl . . . 722

Anhang C: Fehlerdiagnose-Handbuch **722**
Symptom-Schnellübersicht . . . 722
C1 — INFEASIBLE . . . 723
Diagnose in fünf Schritten . . . 723
Wenn die fünf Schritte nicht reichen: den Konflikt einkreisen . . . 724
C2 — UNBOUNDED . . . 732
C3 — Zu langsam . . . 733
Diagnosereihenfolge . . . 733
C4 — Unsinniges Ergebnis . . . 733
Checkliste . . . 734
Die wirksamste Gegenmaßnahme . . . 734
C5 — Instabile Lösung . . . 734
Derselbe Fehler außerhalb der Finanzwelt . . . 735
C6 — Falsche Dualwerte . . . 735
Die drei Prüfungen . . . 735
C7 — Widersprüchliche Solver . . . 736
C8 — DCPError . . . 736
C9 — Importfehler . . . 737
C10 — Verdächtig guter Backtest . . . 737
Prüfreihenfolge . . . 738
C11 — Vertauschte Spalten . . . 738
Die goldene Diagnoseregel . . . 739

Anhang D: Spickzettel der Solver **739**
D1 — SciPy: `linprog` und `milp` . . . 739
Lineares Programm . . . 740
Ganzzahlig: `milp` . . . 740
Die drei häufigsten Stolpersteine . . . 741
D2 — HiGHS über `highspy` . . . 741
Die drei häufigsten Stolpersteine . . . 742
D3 — OR-Tools: `pywraplp` . . . 742
Die drei häufigsten Stolpersteine . . . 743
D4 — CP-SAT: `cp_model` . . . 743
Die drei häufigsten Stolpersteine . . . 744
D5 — CVXPY . . . 744
Die drei häufigsten Stolpersteine . . . 745
D6 — Modellierungssprachen: Pyomo und Linopy . . . 745
Pyomo . . . 745

Linopy 746
Die drei häufigsten Stolpersteine 747
Die gemeinsame Regel 747

Anhang E: Glossar und Literatur **747**

Glossar **747**

Literaturverzeichnis **753**
Mathematische Optimierung — Grundlagen 753
Konvexe, nichtlineare und robuste Optimierung 753
Constraint Programming und Scheduling 754
Quantitative Finanzmathematik 754
Software und Dokumentation 754
Verbände und Normen 755

Stichwortverzeichnis **756**

Vorwort

Dieses Buch hat ein einziges Ziel: **Sie sollen am Ende in der Lage sein, ein reales Entscheidungsproblem aus Ihrem eigenen Umfeld in ein mathematisches Modell zu übersetzen, es in Python zu lösen und das Ergebnis jemandem zu erklären, der kein Mathematiker ist.**

Dieses Kompendium versteht sich als **Arbeitsbuch**. Das zeigt sich an drei Stellen:

Erstens: Formeln werden übersetzt. Zu jeder nicht-trivialen Formel gehört in dieser Ausgabe eine Lesehilfe, die jedes Symbol einzeln benennt, und eine Umschreibung in Alltagssprache. Formeln sind eine Abkürzung für Menschen, die den Inhalt schon kennen — wer ihn erst lernt, braucht den ausgeschriebenen Text daneben. Niemand sollte an der Notation scheitern, wenn er den Gedanken versteht.

Zweitens: Es wird geübt. Über 130 Aufgaben mit vollständigen Lösungen begleiten die Kapitel. Sie sind gestaffelt: von Verständnisfragen, die man im Kopf beantwortet, über Handrechnungen bis zu Programmieraufgaben, die auf den Kapitelbeispielen aufbauen. Wer Optimierung nur liest, kann anschließend über Optimierung reden. Wer die Aufgaben rechnet, kann optimieren.

Drittens: Der Weg zur eigenen Anwendung ist ausgeschildert. Die Projektwerkstatt am Ende enthält elf vollständig ausgearbeitete Projektaufträge — vom Schul-Vertretungsplaner über die Tourenoptimierung eines Lieferdienstes bis zum eigenen Portfolio-Rebalancer. Jeder Auftrag nennt Datenquellen, Modellskizze, Abnahmekriterien und Stolperfallen. Sie sind so zugeschnitten, dass sie in 10 bis 25 Stunden zu einem vorzeigbaren Ergebnis führen.

Ziel der Masterclass

Ein umfassendes, didaktisch von Grund auf aufgebautes Lehrbuch zur mathematischen Entscheidungsoptimierung (**Operations Research, OR**) mit praktischer Umsetzung in **Python**. Dieser Kurs der **mathematischen Optimierung** setzt kein Vorwissen in OR voraus und führt schrittweise von den algebraischen und kombinatorischen Grundlagen über Scheduling- und Constraint-Modelle bis hin zur professionellen Anwendung im quantitativen Aktienhandel, Portfoliomanagement und Risikocontrolling.

Vorausgesetzte Kenntnisse

Vorausgesetzt werden **sichere Python-Kenntnisse** (Funktionen, Klassen, NumPy, pandas) sowie Mathematik auf Grundstudiumsniveau:

Gebiet	Was Sie können sollten	Wo im Buch gebraucht
Lineare Algebra	Vektoren, Matrizen, Matrix-Vektor-Produkt, Eigenwerte	ab Kapitel 2, zentral in Kapitel 11, Kapitel 18, Kapitel 19
Analysis	Ableitung, Gradient, notwendige Bedingung erster Ordnung	Kapitel 11
Statistik	Erwartungswert, Varianz, Standardabweichung, Quantil	Kapitel 12, 11–14

Gebiet	Was Sie können sollten	Wo im Buch gebraucht
Python	Listen, Dictionaries, Schleifen, Funktionen, Klassen, NumPy-Arrays	durchgängig

> **Wenn Ihnen Mathematik-Bausteine fehlen:** Kapitel 2 fasst die benötigte lineare Algebra vollständig zusammen, Abschnitt 11.2 wiederholt Gradienten. Sie können also einsteigen und die Lücken unterwegs schließen. Nur die Statistik ab Teil IV setzt wirklich Vorwissen voraus — dort hilft ein Blick in ein Einführungswerk zur Wahrscheinlichkeitsrechnung.

Benötigte Ressourcen

Ein Rechner mit Python 3.10 oder neuer genügt. **Spezial-Hardware (GPU) ist nicht nötig** — sämtliche Beispiele laufen auf einem gewöhnlichen Notebook, die meisten in unter zehn Sekunden. Die Kapitel 18 bis 14 benötigen einen **Internetzugang**, da sie aktuelle Kursdaten über `yfinance` laden.

Setup & Installation

Alle Beispielprogramme sind unter Python 3.10+ lauffähig. Legen Sie zunächst eine virtuelle Umgebung an — so bleiben die Kurspakete von Ihrer Systeminstallation getrennt:

```
python3 -m venv .venv
source .venv/bin/activate          # Windows: .venv\Scripts\activate
pip install --upgrade pip
pip install ortools highspy cvxpy scikit-learn yfinance matplotlib pandas numpy scipy \
            openpyxl polars plotly pyomo linopy pymoo pydantic
```

Alternativ liegt im Ordner `Operations_Research_mit_Python_Version_04_Programme/` eine `requirements.txt`:

```
pip install -r Operations_Research_mit_Python_Version_04_Programme/requirements.txt
```

Installationstest

Prüfen Sie mit diesem Skript, ob alles bereitsteht, **bevor** Sie mit Kapitel 1 beginnen. Es meldet für jedes Paket Version und Status und löst ein Mini-Optimierungsproblem.

Eine Besonderheit dabei — die Import-Reihenfolge ist bewusst gewählt: `highspy` und `cvxpy` werden in der Paketübersicht nur auf Anwesenheit geprüft, und `ortools` lädt seine native Bibliothek zuerst. Der Grund: `ortools` und `highspy` bringen jeweils eine eigene HiGHS-Kopie mit, die sich nicht im selben Python-Prozess verträgt — und `cvxpy` importiert ein installiertes `highspy` bei der Solver-Erkennung selbst mit. Lädt deren HiGHS-Kopie zuerst, bricht der Test beim CP-SAT-Funktionstest mit einem kryptischen `ImportError: undefined symbol` ab; so gewinnt die Kopie von `ortools`, und CVXPY verzichtet lediglich auf sein HIGHS-Interface (für alle Buchprogramme folgenlos). Details und Abhilfen: Abschnitt 3.5 und Anhang C.

```python
#!/usr/bin/env python3

# Installationstest.py
"""
Vorspann: Prüft die vollständige Kurs-Installation.
Ausgabe: eine Zeile pro Paket plus ein gelöstes Mini-Modell je Solver-Familie.
"""

import importlib.metadata
import importlib.util
import logging
import sys

PAKETE = [
    ("numpy",        "Numerische Basis (Vektoren, Matrizen)"),
    ("scipy",        "Wissenschaftliche Algorithmen, linprog/minimize"),
    ("pandas",       "Tabellen und Zeitreihen"),
    ("matplotlib",   "Diagramme"),
    ("ortools",      "Google OR-Tools: CP-SAT und Routing"),
    ("highspy",      "HiGHS-Solver, direkte Steuerung"),
    ("cvxpy",        "Konvexe Optimierung (Portfolio, CVaR)"),
    ("sklearn",      "Ledoit-Wolf-Shrinkage der Kovarianzmatrix"),
    ("yfinance",     "Kursdatenbezug (nur die Finanzkapitel)"),
]

# Nur auf Anwesenheit prüfen, NICHT importieren (siehe Kapitel Ökosystem):
# - highspy: seine HiGHS-Bibliothek verträgt sich nicht mit der Kopie von
#   ortools im selben Prozess.
# - cvxpy: importiert bei der Solver-Erkennung ein installiertes highspy
#   selbst mit und löst so denselben Konflikt aus. Importiert wird cvxpy
#   erst im Funktionstest, nachdem ortools bereits geladen ist.
NUR_PRUEFEN = {"highspy", "cvxpy"}

def paket_version(name: str) -> str:
    """Liefert die installierte Version; ImportError, falls das Paket fehlt."""
    if name in NUR_PRUEFEN:
        if importlib.util.find_spec(name) is None:
            raise ImportError(name)               # nicht installiert
        try:
            # Version aus den Metadaten – das Modul wird ja nicht geladen
            return importlib.metadata.version(name)
        except importlib.metadata.PackageNotFoundError:
            return "unbekannt"                    # installiert, aber ohne Metadaten
    modul = importlib.import_module(name)
    return getattr(modul, "__version__", "unbekannt")

def pruefe_pakete() -> list[str]:
    """Prüft jedes Paket und meldet Version oder Fehlgrund."""
```

```
    fehlend = []
    print(f"Python-Version: {sys.version.split()[0]}\n")
    print(f"{'Paket':<12} {'Version':<12} {'Zweck'}")
    print("-" * 78)
    for name, zweck in PAKETE:
        try:
            print(f"{name:<12} {paket_version(name):<12} {zweck}")
        except ImportError:
            print(f"{name:<12} {'FEHLT':<12} {zweck}")
            fehlend.append(name)
    return fehlend

def teste_cp_sat() -> bool:
    """Löst 'maximiere x+y unter x+2y<=10, x<=4' mit CP-SAT. Erwartet: x=4, y=3."""
    from ortools.sat.python import cp_model
    modell = cp_model.CpModel()
    x = modell.NewIntVar(0, 4, "x")
    y = modell.NewIntVar(0, 10, "y")
    modell.Add(x + 2 * y <= 10)
    modell.Maximize(x + y)
    loeser = cp_model.CpSolver()
    status = loeser.Solve(modell)
    ok = status == cp_model.OPTIMAL and loeser.Value(x) == 4 and loeser.Value(y) == 3
    print(f"CP-SAT      : x={loeser.Value(x)}, y={loeser.Value(y)}  -> {'OK' if ok else
     ↪  'FEHLER'}")
    return ok

def teste_scipy_linprog() -> bool:
    """Löst dasselbe Problem kontinuierlich mit HiGHS über SciPy. Erwartet: x=4, y=3."""
    from scipy.optimize import linprog
    # linprog minimiert -> Zielfunktion negieren, um zu maximieren
    ergebnis = linprog(c=[-1, -1], A_ub=[[1, 2]], b_ub=[10],
                       bounds=[(0, 4), (0, 10)], method="highs")
    ok = ergebnis.success and abs(ergebnis.x[0] - 4) < 1e-6 and abs(ergebnis.x[1] - 3) < 1e-6
    print(f"SciPy/HiGHS : x={ergebnis.x[0]:.2f}, y={ergebnis.x[1]:.2f}  -> {'OK' if ok else
     ↪  'FEHLER'}")
    return ok

def teste_cvxpy() -> bool:
    """Minimiert (x-2)^2 unter x<=1 mit CVXPY. Erwartet: x=1."""
    # CVXPY warnt beim Import, wenn sein HIGHS-Interface wegen der
    # HiGHS-Kollision (siehe Kapitel Ökosystem) nicht lädt – für diesen Test
    # folgenlos, deshalb die Warnung kurz stillstellen.
    logging.disable(logging.WARNING)
    import cvxpy as cp
    logging.disable(logging.NOTSET)
    x = cp.Variable()
    problem = cp.Problem(cp.Minimize(cp.square(x - 2)), [x <= 1])
```

```
    problem.solve()
    ok = problem.status == "optimal" and abs(x.value - 1.0) < 1e-6
    print(f"CVXPY       : x={x.value:.4f}  -> {'OK' if ok else 'FEHLER'}")
    return ok

if __name__ == "__main__":
    # ortools' native Bibliothek zuerst laden (Kapitel Ökosystem): die zuerst
    # geladene HiGHS-Kopie gewinnt – und das soll die von ortools sein.
    if importlib.util.find_spec("ortools") is not None:
        from ortools.sat.python import cp_model

    fehlend = pruefe_pakete()
    print("\nSolver-Funktionstest")
    print("-" * 78)
    if fehlend:
        print(f"Abbruch: Es fehlen {len(fehlend)} Pakete: {', '.join(fehlend)}")
        print("Installation: pip install " + " ".join(
            "scikit-learn" if p == "sklearn" else p for p in fehlend))
        sys.exit(1)

    alle_ok = all([teste_cp_sat(), teste_scipy_linprog(), teste_cvxpy()])
    print("-" * 78)
    print("Alles bereit – Sie können mit dem ersten Kapitel beginnen."
          if alle_ok else "Mindestens ein Solver arbeitet fehlerhaft.")
    sys.exit(0 if alle_ok else 1)
```

Erwartete Ausgabe (Versionsnummern können abweichen):

```
Python-Version: 3.12.3

Paket        Version      Zweck
------------------------------------------------------------------------------
numpy        2.1.3        Numerische Basis (Vektoren, Matrizen)
scipy        1.14.1       Wissenschaftliche Algorithmen, linprog/minimize
...
Solver-Funktionstest
------------------------------------------------------------------------------
CP-SAT      : x=4, y=3  -> OK
SciPy/HiGHS : x=4.00, y=3.00  -> OK
CVXPY       : x=1.0000  -> OK
------------------------------------------------------------------------------
Alles bereit – Sie können mit dem ersten Kapitel beginnen.
```

> **Hinweis zur Reproduzierbarkeit:** Die Programme aus Teil IV (Kapitel 18 bis Kapitel 21) laden aktuelle Kursdaten live über `yfinance`. Die im Buchtext abgedruckten Zahlen dienen daher nur zur Illustration — bei eigenem Ausführen weichen sie je nach Abrufdatum und gewähltem Zeitfenster ab. Das ist beabsichtigt und selbst Teil der Lektion über die Instabilität empirischer Schätzungen (Kapitel 18).

Wie dieses Buch aufgebaut ist

Jedes Kapitel folgt derselben Struktur. Wenn Sie wissen, wie die Bausteine funktionieren, können Sie gezielt springen.

Die Lernelemente

Kapitel auf einen Blick Zu Beginn jedes Kapitels: Worum geht es, was wird vorausgesetzt, was können Sie danach, wie lange dauert es. Nutzen Sie diese Box, um zu entscheiden, ob Sie das Kapitel gerade brauchen.

 Formel-Lesehilfe Steht unter jeder wichtigen Formel und benennt jedes Symbol einzeln, gefolgt von einer Umschreibung in Alltagssprache („Ohne Formel gesagt: …“). Wer die Formel schon versteht, überspringt die Box.

 Handrechnung Ein kleines Zahlenbeispiel, das Sie mit Papier und Bleistift nachvollziehen können — ohne Computer. Diese Rechnungen sind bewusst winzig gehalten, damit der Mechanismus sichtbar wird, den der Solver später millionenfach ausführt.

 Code-Durchgang Nach jedem längeren Programm eine Tabelle, die die entscheidenden Zeilen erklärt. Vollständige Programme sind bewusst am Stück abgedruckt und nicht in Fragmente zerlegt — Sie sollen sie kopieren und laufen lassen können.

 Typische Fehler Die Fallen, in die erfahrungsgemäß viele tappen — mit dem Symptom, das sie erzeugen, und der Korrektur.

 Merksatz Die eine Aussage, die vom Abschnitt hängen bleiben soll.

 Übungsaufgaben Am Kapitelende, gestaffelt nach Schwierigkeit: Verständnis (im Kopf oder in zwei Sätzen) · Handrechnung oder kleine Modelländerung · Eigenständige Programmieraufgabe. **Alle Lösungen** stehen in **`90_Anhang_Loesungen.md`**.

 Selbsttest Fünf Fragen mit Kurzantworten zur schnellen Selbstkontrolle.

Der Dreischritt jedes Kapitels

1. **Problemstellung** — eine konkrete Entscheidungssituation, kein abstraktes Beispiel.
2. **Mathematische Formulierung** — mit Lesehilfe und, wo möglich, Handrechnung.
3. **Lauffähiges Python-Programm** — vollständig abgedruckt, mit erwarteter Ausgabe.

Für wen dieses Buch geschrieben ist

Die Lernpfade weiter unten sind nach **Themen** geschnitten. Dieser Abschnitt ist nach **Rollen** geschnitten: vier Einstiege für vier Arten, mit Optimierung zu tun zu haben. Jeder nennt den Schmerzpunkt, drei Dinge, die das Buch dafür wirklich liefert — mit den Zahlen, die es dazu misst —, den passenden Lernpfad und die Frage, an der es in dieser Rolle am häufigsten scheitert.

Suchen Sie sich einen aus. Sie können auch alle vier überspringen; nötig sind sie nicht.

Für Entscheider und Manager — der Geschäftsfall

> *„Wir planen seit Jahren mit Erfahrungswerten, und es funktioniert. Warum sollte ein Modell das besser können — und was kostet mich der Irrtum, wenn ich es nicht tue?“*

- **Was Planen mit Durchschnitten kostet, wird hier beziffert, nicht behauptet.** Im Kraftwerkseinsatz aus Kapitel 17 kostet ein Plan, der mit dem Wind-**Erwartungswert** rechnet, in der Wirklichkeit **149 % mehr** als einer, der 40 Szenarien berücksichtigt — 874 870 € statt 351 356 € — und führt in **28 von 40 Szenarien** zum Lastabwurf statt in keinem.
- **„Wie sicher soll es sein?“ wird eine Preisfrage.** Abschnitt 12.7 rechnet aus, was ein Prozentpunkt Versorgungssicherheit kostet: rund 56 000 € auf dem Weg zu 80 %, rund 254 000 € zwischen 95 und 99 %. Das **4,5-fache**. Wer 99,9 % fordert, ohne diese Kurve gesehen zu haben, fordert ins Blaue.
- **Für die Sitzung gibt es einen Bericht, keine Tabelle.** Abschnitt 22.4 beziffert je Nebenbedingung, was sie den Plan kostet, und formuliert daraus Sätze in Alltagssprache. Für die Frage des Betriebsrats — *„warum ausgerechnet ich?“* — liefert Kapitel 22 den Constraint-Trace, der die einzelne Zuweisung begründet.

Ihr Weg. Zuerst Pfad E (2–4 Stunden): Abschnitt 1.5 und Anhang B. Wenn Sie danach wissen wollen, was in Ihrem Betrieb geht, Pfad B und Projekt **P1** oder **P2**.

Woran es scheitert. Nicht am Modell, sondern an der Einführung — das ist der meistgenannte Befund in Kapitel 22. Und an einer Verwechslung, die dieses Buch Ihnen nicht abnehmen kann: Ein Bericht erklärt das **Modell**, nicht die **Wirklichkeit**. Ist ein Deckungsbeitrag falsch geschätzt, ist er überzeugend **und** falsch. Fragen Sie deshalb jedes Mal, worauf eine Zahl beruht.

> **Eine Erwartung, die dieses Buch enttäuscht:** Es nennt keine ROI-Kennzahl und keine Amortisationsdauer. Beide hängen an Ihrem Betrieb, und jede Zahl hier wäre erfunden. Was es stattdessen liefert, ist das Werkzeug, Ihre eigene auszurechnen: den **EVPI** in Kapitel 12 (was eine perfekte Prognose überhaupt wert wäre — eine Obergrenze für jedes Prognoseprojekt) und die Kostenzurechnung aus Abschnitt 22.4.

Für Ingenieure und Produktionsplaner — die Physik des Systems

> *„Der Engpass wandert. Kaum ist eine Maschine entlastet, steht die nächste — und niemand kann mir vorher sagen, welche."*

- **Der Engpass lässt sich benennen und bepreisen.** Abschnitt 22.4 rechnet für jede Kapazität aus, was die nächste Stunde wert ist — und wie weit dieser Wert trägt. Im Beispiel gilt der Schattenpreis der Lackiererei exakt bis **+15 Stunden**; eine ganze Sonderschicht von 30 Stunden bringt deshalb 60 € statt der rechnerischen 120 €. **Die zweite Hälfte wäre bezahlt und wirkungslos.**
- **Rüstzeiten, Reihenfolgen, Taktzeiten sind der Kern, nicht die Ausnahme.** Kapitel 7 modelliert Maschinenbelegung mit Intervallvariablen, Kapitel 9 zeigt am selben Fall das Gesamtbild: Faustregel 2 497 Minuten Rüstzeit, CP-SAT allein 2 628, Simulated Annealing 2 343, Annealing plus LNS **2 289** — und die untere Schranke bei 1 768, damit sichtbar bleibt, wie viel Luft noch da ist.
- **Wenn der Solver zu lange braucht, gibt es Zahlen statt Ratschläge.** Abschnitt 7.7 misst, was mehrere Suchstränge bringen (hier Faktor 12,3) — und ab wo mehr Arbeiter wieder schaden.

Ihr Weg. Pfad B bis Kapitel 7, dann Projekt **P5** (Produktionsplanung mit Rüstkosten). Wenn Ihre echten Instanzen den Solver stehen lassen: Kapitel 9 und Projekt **P9**.

Woran es scheitert. An der Verwechslung von *bindend* und *teuer*. Eine Bedingung kann den Plan berühren und trotzdem **nichts** kosten — im Beispiel von Abschnitt 22.4 gilt das für eine von fünf bindenden. Wer sie nachverhandelt, gewinnt nichts und hält das Verfahren für nutzlos.

Für Investoren und Finanzanalysten — Risiko und Ertrag

> *„Die Optimierung liefert mir Gewichte, die kein Mensch halten würde: 60 % in einem Titel, null in fünf anderen. Und im nächsten Quartal sieht sie völlig anders aus."*

- **Der Grund dafür steht in Kapitel 18, nicht in Kapitel 19.** Eine aus kurzen Zeitreihen geschätzte Kovarianzmatrix ist schlecht konditioniert; der Optimierer setzt dann auf Schätzrauschen. Shrinkage nach Ledoit-Wolf ist das Gegenmittel; ihre Wirkung wird an **Eigenwertspektrum und Konditionszahl** gemessen, nicht behauptet.
- **Der VaR verschweigt genau das, worauf es ankommt.** In der Stichprobe aus Kapitel 20 sagt der VaR: „an 95 % der Tage höchstens **1,86 %** Verlust". Der CVaR sagt: „und wenn doch, dann im Mittel **2,99 %**". Der schlechteste Tag der Stichprobe liegt bei **−23,0 %** — am VaR ändert dieser eine Tag **nichts**, am CVaR sehr wohl. Deshalb ist der CVaR die Größe, die man optimiert.
- **Transaktionskosten und Rebalancing sind Teil des Modells, nicht ein Abzug danach.** Kapitel 20 nimmt sie in die Zielfunktion, Kapitel 21 setzt den Backtest daneben — samt der Fallen, die einen guten Backtest wertlos machen.

Ihr Weg. Pfad D, und zwar **mit** Kapitel 18. Danach Projekt **P6** (Portfolio-Rebalancer) oder **P7** (Risikoreport mit CVaR und Stresstests).

Woran es scheitert. An der Reihenfolge. Wer bei Markowitz einsteigt und die Datenkapitel überspringt, bekommt ein Modell, das seine eigenen Schätzfehler maximiert — und merkt es erst im Echtgeldbetrieb.

Für Studierende und Data Scientists — die Methodik

> *„Ich kann ein Modell trainieren und den Fehler senken. Aber wie wird aus einer Prognose eine Entscheidung — und woher weiß ich, dass sie gut ist?“*

- **Die unbequemste Erkenntnis zuerst.** Kapitel 15 zeigt an einem gerechneten Fall, dass das Modell mit dem **schlechteren** MSE die **günstigere** Entscheidung trifft. Prognosegüte und Entscheidungsqualität sind zwei verschiedene Größen; wer die eine optimiert, bekommt die andere nicht geschenkt.
- **Dieselbe Mathematik, zwei Welten.** Kapitel 16 lässt **denselben Code** einmal über eine Werkstatt und einmal über ein Depot laufen — und benennt die drei Stellen, an denen die Analogie endet. Struktur zu erkennen ist die eigentliche Fähigkeit, nicht Bibliothekskenntnis; zu wissen, wo eine Analogie aufhört, gehört dazu.
- **Es wird gerechnet, nicht nacherzählt.** Zu jedem Kapitel gehören Aufgaben mit vollständigen Lösungen, Handrechnungen und ein „Finde den Denkfehler“ — und jedes abgedruckte Ergebnis stammt aus einem echten Lauf des danebenstehenden Programms.

Ihr Weg. Pfad A, wenn Sie Zeit haben. Sonst Pfad B oder D bis zum ersten Projekt und von dort zurück in die Grundlagen. Für eigene Fragestellungen: Projekt **P8**.

Woran es scheitert. An der Annahme, ein Solver sei eine Bibliothek wie jede andere. Die schwierige Arbeit steckt in der Formulierung — Kapitel 4 ist deshalb kein Vorgeplänkel, sondern das Kapitel, auf das alle anderen aufbauen.

Lernpfade

Sie müssen nicht alles lesen, um etwas Nützliches bauen zu können.

Pfad A — Vollständiger Lehrgang (100–140 Stunden)

Kapitel 1 bis Kapitel 23 in Reihenfolge, danach ein Projekt aus der Projektwerkstatt. Empfohlen, wenn Sie OR systematisch lernen wollen. Rechnen Sie mit 4–6 Stunden je Kapitel inklusive Übungen.

Pfad B — Planung, Disposition, Personaleinsatz (ca. 25 Stunden)

Kapitel 1 (Bausteine) → Kapitel 2 (bis Abschnitt 2.5) → Kapitel 4 (vom Wunsch zum Modell) → Kapitel 5 (LP, Schattenpreise) → Kapitel 6 (Ja/Nein-Entscheidungen) → Kapitel 7 (CP-SAT, Scheduling) → Projekt **P1** (Vertretungsplaner) oder **P2** (Schichtplanung). Das ist der kürzeste Weg zu einem einsetzbaren Dienstplan-Optimierer.

Wenn der Solver bei Ihrer echten Instanz stehen bleibt: Kapitel 9 und Projekt **P9**.

Pfad C — Logistik und Tourenplanung (ca. 25 Stunden)

Kapitel 1 → Kapitel 2 (bis Abschnitt 2.4) → Kapitel 5 → Kapitel 6 → Kapitel 8 (Graphen, VRP) → Projekt **P3** (Liefertouren) oder **P4** (Lagernetzwerk).

Wenn die Instanzen zu groß werden: Kapitel 9 (gute Lösung in fester Zeit) und Kapitel 10 (das Modell umbauen statt die Lösung raten) — dazu Projekt **P9**, das beide Wege am selben Problem vergleicht.

Pfad D — Quantitative Finance (ca. 30 Stunden)

Kapitel 1 → Kapitel 2 → Kapitel 11 (QP, KKT) → Kapitel 18 (Daten, Shrinkage) → Kapitel 19 (Markowitz) → Kapitel 20 (CVaR) → Kapitel 21 (Backtest) → Projekt **P6** (Portfolio-Rebalancer) oder **P7** (Risikoreport). **Wichtig:** Überspringen Sie Kapitel 18 nicht. Wer direkt bei Markowitz einsteigt, optimiert Schätzrauschen und wundert sich über absurde Gewichte.

Pfad E — Ich habe morgen ein konkretes Problem (2–4 Stunden)

Lesen Sie Abschnitt 1.5 (die vier Bausteine) und Abschnitt 4.6 (welcher Satz welcher Baustein ist), dann Anhang B (**Modellierungsmuster**) und suchen Sie dort das Muster, das zu Ihrem Problem passt. Von jedem Muster führt ein Verweis in das zuständige Kapitel. Wie die gewählte Bibliothek es schreibt, steht in Anhang D (**Spickzettel**); wenn etwas nicht läuft, in Anhang C (**Fehlerdiagnose**).

Pfad F — Vom Prototyp in den Betrieb (ca. 15 Stunden)

Für alle, deren Modell **rechnet** und die es jetzt jemand anderem übergeben müssen: Kapitel 22 (die fünf Praxisfallen, `or_kern.py`, Erklärbarkeit) → Kapitel 23 (Testsuite, Mutationstest, Benchmark, HTTP-Dienst) → Anhang C als Nachschlagewerk für den Ernstfall, dann Projekt **P11** (Vom Skript zum Dienst). Setzt voraus, dass Sie mindestens einen der Pfade B, C oder D hinter sich haben — P11 baut auf einem Modell auf, das Sie schon haben.

Der Kern in einem Satz: Ein Modell, das nur auf Ihrem Rechner und nur mit Ihren Daten läuft, ist ein Prototyp — kein System.

Verzeichnis der Beispielprogramme

Alle Beispielprogramme dieses Buchs im Überblick, sortiert nach Kapitel — praktisch für Pfad E oder um gezielt nach einem Thema zu suchen:

Programm	Thema	Kapitel
`Installationstest.py`	Installationsprüfung	Vorspann
`Bot_Allokation.py`	Erstes CP-SAT-Modell	Kapitel 1
`Brute_Force_Vergleich.py`	Kombinatorische Explosion	Kapitel 1
`Bausteine_Vorlage.py`	Vorlage für eigene Modelle	Kapitel 1
`Excel_Bruecke.py`	Excel-Mappe lesen, lösen, zurückschreiben	Kapitel 1
`Matrixform.py`	Matrixform und Zulässigkeit	Kapitel 2
`Konvexitaet_Demo.py`	Sehnen-Test, lokale Optima	Kapitel 2

Programm	Thema	Kapitel
`Visualisierung_Loesungsraum.py`	Polyeder mit Ecken	Kapitel 2
`Skalierung_Kondition.py`	Konditionszahl, Ruiz, Toleranzen	Kapitel 2
`Solver_Wahl.py`	Entscheidungshilfe	Kapitel 3
`Ein_System_Vier_Ansaetze.py`	Vier Bibliotheken	Kapitel 3
`Modellierungsschichten.py`	Pyomo und Linopy	Kapitel 3
`Vektorisierte_Modellgenerierung.py`	Aufbauzeit vs. Lösezeit	Kapitel 3
`Vom_Wunsch_zum_Modell.py`	Fünf Modelle auf denselben Daten	Kapitel 4
`Simplex_Tableau_LP.py`	Simplex von Grund auf	Kapitel 5
`Sensitivitaetsanalyse.py`	Schattenpreise	Kapitel 5
`Dualitaet_Nachweis.py`	Primal-dual, starke Dualität	Kapitel 5
`Toleranzen_und_Entartung.py`	Entartung, Toleranzen, Dualspannen	Kapitel 5
`Runden_Gegenbeispiel.py`	Warum Runden scheitert	Kapitel 6
`Rucksack.py`	Knapsack, LP-Schranke	Kapitel 6
`MILP_Portfolio_Fixgebuehren.py`	Fixkosten, Kardinalität	Kapitel 6
`Solverstatus_und_Gap.py`	MIP-Gap, Zeitlimit, Statusfälle	Kapitel 6
`Warmstart_Effekt.py`	LPT-Hinweis für CP-SAT	Kapitel 6
`Big_M_Falle.py`	Trickle Flow bei zu großem M	Kapitel 6
`Propagation_Demo.py`	Propagation messbar	Kapitel 7
`CP_SAT_Vertretungssystem.py`	Vertretungsplan	Kapitel 7
`JobShop_Intervalle.py`	Job-Shop-Scheduling	Kapitel 7
`Parallele_Suche.py`	num_workers: Tempo gegen Reproduzierbarkeit	Kapitel 7
`CP_SAT_Statusfaelle.py`	Die fünf Solver-Antworten	Kapitel 7
`Strafgewichte.py`	Gewichte als Wechselkurse	Kapitel 7
`Min_Cost_Flow.py`	Netzwerkfluss	Kapitel 8
`Zuordnung_Ungarisch.py`	Zuordnung, Unimodularität	Kapitel 8
`VRP_Flotten_Routing.py`	CVRPTW	Kapitel 8
`VRP_Kapazitaetsfalle.py`	Vergessene Dimension im Routing	Kapitel 8
`Simulated_Annealing.py`	Lokale Suche, Temperatur kalibrieren	Kapitel 9

Programm	Thema	Kapitel
`Metaheuristik_vs_Exakt.py`	Der Umschlagpunkt, und die Schranke	Kapitel 9
`Large_Neighborhood_Search.py`	Zerstören und exakt reparieren	Kapitel 9
`Spaltengenerierung.py`	Muster statt Stücke, Master und Pricing	Kapitel 10
`QP_Grundlagen.py`	Konvexität, DCP-Check	Kapitel 11
`KKT_Nachweis.py`	KKT numerisch prüfen	Kapitel 11
`Entropie_Maximierte_Allokation.py`	NLP mit Entropie	Kapitel 11
`Lokale_Optima_Multistart.py`	Lokale Optima, Multistart	Kapitel 11
`Fluch_des_Durchschnitts.py`	Optimum ≠ Mittelwert	Kapitel 12
`Monte_Carlo.py`	Monte-Carlo-Bewertung	Kapitel 12
`Stochastische_Optimierung.py`	Two-Stage mit Recourse	Kapitel 12
`Robuste_Optimierung.py`	Worst-Case-Absicherung	Kapitel 12
`Chance_Constraints.py`	Zusage „mit 95 % Sicherheit“, SOC und Big-M	Kapitel 12
`Bellman_Minimalbeispiel.py`	Rückwärtsinduktion	Kapitel 13
`Mehrziel_Pareto.py`	Pareto-Front, ε-Constraint, Gewichtslücke	Kapitel 14
`Predict_then_Optimize.py`	MSE gegen Entscheidungskosten	Kapitel 15
`Strukturbruecke.py`	Derselbe Code über Werkstatt und Depot	Kapitel 16
`Kraftwerkseinsatz.py`	Unit Commitment unter Windunsicherheit	Kapitel 17
`Mehrperiodige_Order_Execution.py`	Almgren-Chriss	Kapitel 13
`Renditen_Vergleich.py`	Diskret vs. logarithmisch	Kapitel 18
`Schaetzrauschen_Demo.py`	Error-Maximizer messen	Kapitel 18
`Finanzdaten_Ledoit_Wolf.py`	Datenpipeline, Shrinkage	Kapitel 18
`Kovarianz_Falle.py`	Singuläre Kovarianz, Error-Maximizer	Kapitel 18
`Markowitz_CVXPY.py`	GMV, Max Sharpe, Frontier	Kapitel 19
`Diversifikation_Demo.py`	Korrelation und Portfoliorisiko	Kapitel 19
`Renditeschaetzung_Falle.py`	Schätzfehler in erwarteten Renditen	Kapitel 19

Programm	Thema	Kapitel
`VaR_CVaR_Demo.py`	Fat Tails, Subadditivität	Kapitel 20
`CVaR_Portfolio.py`	CVaR mit Reibung	Kapitel 20
`QuantitativeTradingEngine.py`	Walk-Forward-Backtest	Kapitel 21
`Backtest_Fallen.py`	Fünf Selbsttäuschungen	Kapitel 21
`Data_Snooping.py`	Bestes aus N Versuchen auf Rauschen	Kapitel 21
`Infeasibility_Diagnose.py`	Notfallplan statt Fehler	Kapitel 22
`Erklaerbarkeit.py`	Constraint-Trace, Was-wäre-wenn	Kapitel 22
`Constraint_Attribution.py`	Was kostet welche Bedingung? Managementbericht	Kapitel 22
`Betriebsueberwachung.py`	Status, Gap und Zeitausschöpfung	Kapitel 22
`or_kern.py`	Gemeinsamer Unterbau: Domäne, Status, Prüfung	Kapitel 22
`Solverwechsel_CPSAT_HiGHS.py`	Derselbe Fall in zwei Solvern	Kapitel 22
`test_or_kern.py`	Testsuite für ein Optimierungsmodell	Kapitel 23
`Mutationstest.py`	Findet die Lücken der eigenen Tests	Kapitel 23
`Benchmark_Skalierung.py`	Aufbau gegen Lösen, vier Bibliotheken	Kapitel 23
`Optimierungsdienst.py`	Das Modell als HTTP-Dienst	Kapitel 23
`Konfliktsuche.py`	Deletion Filter: den Widerspruch einkreisen	Anhang C

Was dieses Buch nicht ist

Es ist **keine Anleitung zum Bau eigener Solver** — wir nutzen ausgereifte Implementierungen und investieren die Zeit stattdessen in die Modellierung. Die einzige Ausnahme ist der Simplex-Algorithmus in Kapitel 5, den wir von Grund auf programmieren, weil man ihn verstanden haben muss, um Dualität und Schattenpreise zu begreifen.

Es ist **kein Buch über maschinelles Lernen**. Prognosen kommen nur so weit vor, wie sie als Eingangsdaten einer Optimierung nötig sind.

Und der vierte Teil ist **ausdrücklich keine Anlageberatung**. Die Finanzbeispiele demonstrieren Methodik an realen Daten, nicht handelbare Strategien. Kapitel 22 erklärt ausführlich, warum ein gut aussehender Backtest noch keine funktionierende Strategie ist.

Ein Wort zur Arbeitsweise

Die wirksamste Art, mit diesem Buch zu arbeiten, ist unbequem: **Lesen Sie die Problemstellung, schließen Sie das Buch und versuchen Sie, das Modell selbst aufzuschreiben** — Variablen, Zielfunktion, Nebenbedingungen. Vergleichen Sie erst dann. Die Abweichungen zwischen Ihrem Entwurf und dem Text sind genau die Stellen, an denen Sie etwas lernen. Wer stattdessen mitliest und nickt, hat am Ende des Kapitels das angenehme Gefühl, es verstanden zu haben — und steht beim eigenen Problem vor einem leeren Blatt.

Zweitens: **Lassen Sie jedes Programm laufen und verändern Sie danach eine Zahl.** Was passiert mit dem Vertretungsplan, wenn eine Lehrkraft ausfällt? Was mit dem Portfolio, wenn die Obergrenze von 20 % auf 10 % sinkt? Diese Experimente kosten Sekunden und bauen genau die Intuition auf, die man später beim Modellieren braucht.

Und drittens: **Nehmen Sie die Übungen ernst.** Sie sind nicht Beiwerk, sondern der Ort, an dem aus Kenntnis Können wird.

Weiter mit: `01_Notation_und_Abkuerzungen.md` — oder direkt zu **Kapitel 1**, wenn Sie die Notation lieber nachschlagen, sobald sie auftaucht.

Notation und Abkürzungen

Dieses Kapitel ist zum Nachschlagen gedacht, nicht zum Durchlesen. Schlagen Sie hier nach, sobald Ihnen ein Symbol oder ein Kürzel begegnet, das Sie nicht sicher zuordnen können.

---

1. Die Symbole der Optimierung

1.1 Grundgrößen jedes Modells

Symbol	Sprechweise	Bedeutung	Beispiel aus dem Buch
x	„iks“	**Entscheidungsvariable(n)** — das, was der Solver festlegen darf	Anzahl gestarteter Trading-Bots (Kapitel 1)
$\mathbf{x}$	„Vektor iks“	Alle Entscheidungsvariablen als Spaltenvektor	$\mathbf{x} = (x_1, \dots, x_n)^\mathsf{T}$
n	„en“	Anzahl der Entscheidungsvariablen	10 Aktien $\Rightarrow n = 10$
m	„em“	Anzahl der Nebenbedingungen	3 Ressourcen $\Rightarrow m = 3$

Symbol	Sprechweise	Bedeutung	Beispiel aus dem Buch
$\mathbf{c}$	„Vektor ze“	**Kosten- bzw. Ertragsvektor** der Zielfunktion	Gewinn je Bot-Typ: $(150, 250)^\top$
$\mathbf{A}$	„Matrix a“	**Technologiematrix** — wie viel Ressource verbraucht welche Variable	Zeile „RAM“: $(4, 6)$
$\mathbf{b}$	„Vektor be“	**Kapazitätsvektor**, „rechte Seite“ der Ungleichungen	$(40, 60)^\top$ vCPU und GB
$f(\mathbf{x})$	„ef von iks“	**Zielfunktion** — die eine Zahl, die bewertet wird	Tagesgewinn in Euro
$g_i(\mathbf{x})$	„ge i von iks“	i-te **Ungleichungs-Nebenbedingung**, Form $g_i(\mathbf{x}) \le 0$	„RAM-Verbrauch minus 60 GB“
$h_j(\mathbf{x})$	„ha j von iks“	j-te **Gleichungs-Nebenbedingung**, Form $h_j(\mathbf{x}) = 0$	„Summe der Gewichte minus 1“
s_i	„es i“	**Schlupfvariable** — ungenutzte Reserve der Ressource i	1 GB RAM übrig $\Rightarrow s_2 = 1$
y_i, λ_i	„y i“, „lambda i“	**Dualvariable / Schattenpreis** der Bedingung i	33,33 € je zusätzlicher Prüfstunde
M	„groß em“	**Big-M** — hinreichend große Konstante zum Ein-/Ausschalten von Bedingungen	$x_j \le M \cdot y_j$
z_i, y_i		**Binärvariable** $\in \{0, 1\}$ — Ja/Nein-Schalter	„Position i wird eröffnet“
$\mathcal{F}$	„kalligrafisches ef“	**Zulässiger Bereich** — Menge aller erlaubten Lösungen	das Polyeder aus Kapitel 2
Z, Z^*	„zet“, „zet Stern“	Zielfunktionswert bzw. **optimaler** Zielfunktionswert	$Z^* = 2300$ €
$\mathbf{x}^*$	„iks Stern“	Die **optimale Lösung**. Der Stern markiert immer „im Optimum“	$\mathbf{x}^* = (7, 5)^\top$

1.2 Mengen und Zahlbereiche

Symbol	Bedeutung	Praktische Folge
$\mathbb{R}$	Reelle Zahlen (beliebig teilbar)	Variable darf 3,7 sein — **LP**, schnell lösbar
$\mathbb{R}^n$	n-Tupel reeller Zahlen	Der Raum, in dem **x** lebt
$\mathbb{Z}$	Ganze Zahlen	Variable muss 3 oder 4 sein — **MILP**, NP-schwer
$\mathbb{N}_0$	Natürliche Zahlen einschließlich 0	Stückzahlen, nie negativ
$\{0, 1\}$	Nur zwei Werte	Ja/Nein-Entscheidung
$\in$	„ist Element von“	$x \in \mathbb{Z}$: „x ist ganzzahlig“
$\forall$	„für alle“	$w_i \geq 0\ \forall i$: jedes Gewicht ist nichtnegativ
$\exists$	„es existiert“	seltener; „es gibt mindestens ein …“
$\subseteq$	„ist Teilmenge von“	$\mathcal{C} \subseteq \mathbb{R}^n$
$[a, b]$	Geschlossenes Intervall	$\theta \in [0, 1]$: θ liegt zwischen 0 und 1

1.3 Operatoren und Schreibweisen

Symbol	Bedeutung	Lesehilfe
$\sum_{i=1}^{n} a_i$	Summe	„addiere a_1 bis a_n“
$\mathbf{c}^\mathsf{T}\mathbf{x}$	Skalarprodukt	„$c_1x_1 + c_2x_2 + \ldots$“ — eine einzige Zahl
$^\mathsf{T}$	Transponiert	dreht Zeilen und Spalten; macht aus Spalten- einen Zeilenvektor
$\mathbf{Ax}$	Matrix-Vektor-Produkt	„berechne alle Ressourcenverbräuche auf einmal“
$\min_x$, $\max_x$	Minimum/Maximum über x	„wähle x so, dass … kleinst-/größtmöglich wird“
$\arg\min$	Argument des Minimums	nicht der *Wert*, sondern die *Stelle* des Minimums
∇f	„Nabla ef“, Gradient	Vektor aller partiellen Ableitungen — zeigt bergauf
$\partial Z / \partial b_i$	Partielle Ableitung	„wie ändert sich Z, wenn nur b_i sich ändert?“

Symbol	Bedeutung	Lesehilfe
$\|\mathbf{v}\|_1$	L_1-Norm	Summe der Beträge: $\|v_1\| + \|v_2\| + \ldots$
$\|\mathbf{v}\|_2$	L_2-Norm (euklidisch)	die gewöhnliche Länge eines Pfeils
$\mathbf{P} \succeq 0$	positiv **semi**definit	alle Eigenwerte ≥ 0; Funktion ist konvex
$\mathbf{P} \succ 0$	positiv **definit**	alle Eigenwerte > 0; Funktion ist *streng* konvex
$\mathbb{E}[\cdot]$	Erwartungswert	gewichteter Mittelwert über alle Szenarien
$\mathbb{1}$ oder $\mathbf{1}$	Einsvektor	$(1, 1, \ldots, 1)^\top$; $\mathbf{1}^\top\mathbf{w}$ ist die Summe aller w_i
$\mathbf{I}$	Einheitsmatrix	Diagonale 1, sonst 0
s.t. / u. d. N.	*subject to* / unter den Nebenbedingungen	leitet die Restriktionen ein

1.4 Symbole der Finanzkapitel (Teil IV)

Symbol	Bedeutung	Einheit / typischer Wert
$\mathbf{w}$	**Portfoliogewichte** — Kapitalanteile je Titel	dimensionslos, Summe = 1
w_i	Anteil von Titel i	0,15 ≡ 15 %
μ	„mü“ — Vektor der **erwarteten Renditen**	z. B. 0,11 = 11 % p. a.
Σ	„Sigma“ — **Kovarianzmatrix** der Renditen	$N \times N$, Einheit Rendite²
σ	„sigma“ — **Volatilität** (Standardabweichung)	0,20 = 20 % p. a.
σ^2	**Varianz**	Quadrat der Volatilität
$\mathbf{S}$	**Stichproben**-Kovarianzmatrix (aus Daten geschätzt)	verrauscht
$\mathbf{F}$	**Shrinkage-Target** — strukturierte Vergleichsmatrix	verzerrt, aber stabil
δ	„delta“ — Shrinkage-Intensität	$\in [0, 1]$, z. B. 0,18
r_f	*risk-free rate*, **risikoloser Zins**	z. B. 0,03 = 3 % p. a.
$R_{i,t}$	**Diskrete Rendite** von Titel i am Tag t	$P_t/P_{t-1} - 1$
$r_{i,t}$	**Logarithmische Rendite** (stetige Rendite)	$\ln(P_t/P_{t-1})$

Symbol	Bedeutung	Einheit / typischer Wert
$P_{i,t}$	**Kurs** (bereinigter Schlusskurs) von Titel i	in Währungseinheiten
λ	Risikoaversion / Gewichtung des Risikoterms	Modellparameter, frei wählbar
α	Konfidenzniveau bei VaR/CVaR	0,95 = „schlechteste 5 % der Tage“
γ	Hilfsvariable, schätzt den VaR (Kapitel 20); Diskontfaktor (Kapitel 13)	kontextabhängig
η	„eta“ — Marktimpact-Koeffizient	Kapitel 13, Almgren-Chriss
X_t	Verbleibender Aktienbestand zum Zeitpunkt t	Kapitel 13
n_t	In Periode t verkaufte Stückzahl	Kapitel 13
S (kursiv, Skalar)	Anzahl der **Szenarien**	Kapitel 12, Kapitel 20
T	Anzahl **Zeitschritte** bzw. Handelstage	252 Handelstage/Jahr
N	Anzahl **Titel** im Universum	z. B. 10 Aktien

1.5 Symbole der Verfahrenskapitel (Teil II und III)

Diese Zeichen kommen erst in den Kapiteln vor, die über das klassische LP hinausgehen.

Symbol	Sprechweise	Bedeutung	Kapitel
π	„pi“	**Reihenfolge (Permutation)**: $\pi(k)$ ist der Auftrag, der als k-ter läuft	Kapitel 9
Δ	„delta“	**Zugbewertung** — um wie viel ein Nachbarschaftszug die Lösung verschlechtert ($\Delta > 0$) oder verbessert ($\Delta \leq 0$)	Kapitel 9
T	„te“	**Temperatur** im Simulated Annealing — steuert, wie oft eine Verschlechterung angenommen wird	Kapitel 9

Symbol	Sprechweise	Bedeutung	Kapitel
$e^{-\Delta/T}$		**Annahmewahrscheinlichkeit** (Metropolis-Kriterium): kleine Verschlechterungen fast immer, große fast nie	Kapitel 9
π_i	„pi i“	**Dualpreis** der Bedingung i im Master-LP — der Preis, mit dem das Pricing rechnet	Kapitel 10
a_i	„a i“	Wie oft Stück i in einem **Schnittmuster** vorkommt (eine Spalte des Masters)	Kapitel 10
$1 - \sum_i \pi_i a_i$		**Reduzierte Kosten** einer neuen Spalte. Negativ heißt: Das Muster lohnt sich, nimm es auf	Kapitel 10
„x dominiert y“		**Pareto-Dominanz**: x ist in keinem Ziel schlechter und in mindestens einem besser	Kapitel 14
ε	„epsilon“	**Schranke im ε-Constraint-Verfahren**: ein Ziel wird zur Nebenbedingung, $e(x) \le \varepsilon$	Kapitel 14
$\hat{d}_t$	„d Dach t“	**Prognostizierter** Wert — das Dach unterscheidet die Schätzung vom später eintretenden Ist-Wert d_t	Kapitel 15
$a_{k,t}$	„a k t“	**Anfahrvariable**: Anlage k wird in Stunde t angefahren	Kapitel 17
$u_{k,t}$	„u k t“	**Einsatzvariable**: Anlage k läuft in Stunde t	Kapitel 17

Symbol	Sprechweise	Bedeutung	Kapitel
L_k	„el k“	**Mindestlaufzeit** von Anlage k in Stunden	Kapitel 17
τ	„tau“	Laufender **Zeitindex innerhalb eines Fensters** — dort, wo t schon den Fensteranfang bezeichnet	Kapitel 17

 Achtung, Doppelbelegungen. In der Literatur — und deshalb auch hier — tragen einige Buchstaben je nach Kapitel verschiedene Bedeutungen. Die wichtigsten Fälle: λ ist in Kapitel 5 und Kapitel 11 ein **Lagrange-Multiplikator/Schattenpreis**, in Kapitel 19 und Kapitel 20 ein **frei gewählter Risikoaversionsparameter**. γ ist in Kapitel 13 der **Diskontfaktor**, in Kapitel 20 die **VaR-Hilfsvariable**. S bezeichnet als Matrix $\mathbf{S}$ die Stichprobenkovarianz, als Skalar S die Szenarienzahl. π ist in Kapitel 9 eine **Reihenfolge**, in Kapitel 10 ein **Dualpreis** — beide Bedeutungen sind so verbreitet, dass ein Ausweichbuchstabe mehr verwirren als helfen würde. Der Kontext ist jeweils eindeutig, und die Kapitel weisen an der betreffenden Stelle darauf hin.

---

2. Abkürzungen — vollständig ausgeschrieben

2.1 Problemklassen und Verfahren

Kürzel	Ausgeschrieben	Deutsch	Kapitel
OR	Operations Research	Unternehmensforschung / mathematische Entscheidungsoptimierung	Kapitel 1
LP	Linear Program(ming)	Lineare Programmierung/Optimierung	Kapitel 5
MILP	Mixed-Integer Linear Program(ming)	Gemischt-ganzzahlige lineare Optimierung	Kapitel 6
MIP	Mixed-Integer Program(ming)	Oberbegriff, meist synonym zu MILP	Kapitel 6
IP	Integer Program(ming)	Rein ganzzahlige Optimierung	Kapitel 6
QP	Quadratic Program(ming)	Quadratische Optimierung	Kapitel 11
MIQP	Mixed-Integer Quadratic Program(ming)	Ganzzahlig-quadratische Optimierung	Kapitel 19

Kürzel	Ausgeschrieben	Deutsch	Kapitel
NLP	Nonlinear Program(ming)	Nichtlineare Optimierung	Kapitel 11
MINLP	Mixed-Integer Nonlinear Program(ming)	Ganzzahlig-nichtlineare Optimierung	Kapitel 3
CP	Constraint Programming	Bedingungsprogrammierung	Kapitel 7
SAT	Boolean **Sat**isfiability Problem	Erfüllbarkeitsproblem der Aussagenlogik	Kapitel 7
CP-SAT	Constraint Programming über SAT-Techniken	Solver-Name von Google OR-Tools	Kapitel 7
CDCL	Conflict-Driven Clause Learning	konfliktgetriebenes Klauselllernen	Kapitel 7
DP	Dynamic Programming	Dynamische Programmierung	Kapitel 13
B&B	Branch and Bound	Verzweigen und Beschränken	Kapitel 6
B&C	Branch and Cut	Verzweigen mit Schnittebenen	Kapitel 6
KKT	Karush–Kuhn–Tucker (Bedingungen)	Optimalitätsbedingungen bei Restriktionen	Kapitel 11
SLSQP	Sequential Least Squares Programming	sequentielle quadratische Optimierung	Kapitel 11
RMT	Random Matrix Theory	Zufallsmatrizentheorie	Kapitel 18
SA	Simulated Annealing	Simulierte Abkühlung — Metaheuristik, die Verschlechterungen mit fallender Wahrscheinlichkeit zulässt	Kapitel 9
LNS	Large Neighborhood Search	Große Nachbarschaftssuche: Teil der Lösung verwerfen, Rest exakt neu bauen	Kapitel 9
CG	Column Generation	Spaltengenerierung — Variablen erst erzeugen, wenn sie sich lohnen	Kapitel 10

Kürzel	Ausgeschrieben	Deutsch	Kapitel
IIS	Irreducible Infeasible Subset	Kleinste widersprüchliche Teilmenge von Bedingungen	Anhang C
MSE	Mean Squared Error	Mittlerer quadratischer Fehler — das Standardmaß für Prognosegüte	Kapitel 15

2.2 Probleme mit Eigennamen

Kürzel	Ausgeschrieben	Deutsch	Kapitel
TSP	Traveling Salesperson Problem	Problem des Handlungsreisenden	Kapitel 8
VRP	Vehicle Routing Problem	Tourenplanungsproblem	Kapitel 8
CVRP	**C**apacitated VRP	VRP mit Fahrzeugkapazitäten	Kapitel 8
VRPTW	VRP with **T**ime **W**indows	VRP mit Zeitfenstern	Kapitel 8
CVRPTW	Capacitated VRP with Time Windows	mit Kapazitäten *und* Zeitfenstern	Kapitel 8
MCNFP	Minimum-Cost Network Flow Problem	Kostenminimales Flussproblem	Kapitel 8
MTZ	Miller–Tucker–Zemlin (Formulierung)	Kurzzyklus-Eliminierung	Kapitel 8
JSSP	Job-Shop Scheduling Problem	Werkstattfertigungs-Reihenfolgeproblem	Kapitel 7

2.3 Finanz- und Risikokennzahlen

Kürzel	Ausgeschrieben	Deutsch / Bedeutung	Kapitel
MPT	Modern Portfolio Theory	Moderne Portfoliotheorie (Markowitz)	Kapitel 19
GMV	Global Minimum Variance (Portfolio)	Portfolio kleinstmöglicher Varianz	Kapitel 19

Kürzel	Ausgeschrieben	Deutsch / Bedeutung	Kapitel
MVO	Mean-Variance Optimization	Erwartungswert-Varianz-Optimierung	Kapitel 19
SR	Sharpe Ratio	Überrendite je Einheit Volatilität	Kapitel 19
VaR	Value at Risk	Verlustschwelle bei gegebener Wahrscheinlichkeit	Kapitel 20
CVaR	Conditional Value at Risk	mittlerer Verlust jenseits des VaR	Kapitel 20
ES	Expected Shortfall	anderes Wort für CVaR	Kapitel 20
CAGR	Compound Annual Growth Rate	durchschnittliche jährliche Wachstumsrate	Kapitel 21
MDD	Maximum Drawdown	größter Rückgang vom Höchststand	Kapitel 21
LW	Ledoit–Wolf (Shrinkage)	Schrumpfungsverfahren für Kovarianzmatrizen	Kapitel 18
p. a.	*per annum*	pro Jahr	Teil IV
bp	Basispunkt	0,01 Prozentpunkt; 15 bp = 0,15 %	Kapitel 20, Kapitel 21

2.4 Software, Solver und Schnittstellen

Kürzel	Ausgeschrieben	Was es ist	Lizenz
HiGHS	**Hi**gh Performance **G**eneral-purpose **S**olver	LP-, MILP- und QP-Solver in C++	Open Source (MIT)
OR-Tools	Google Operations Research Tools	Solver-Sammlung (CP-SAT, Routing, Wrapper)	Open Source (Apache 2.0)
CVXPY	**C**onvex **Py**thon	Modellierungssprache für konvexe Optimierung	Open Source (Apache 2.0)

Kürzel	Ausgeschrieben	Was es ist	Lizenz
SciPy	**Sci**entific **Py**thon	Wissenschaftliches Python-Paket; `scipy.optimize` liefert LP- (über HiGHS), Zuordnungs- (`linear_sum_assignment`) und NLP-Solver (`minimize`)	Open Source (BSD)
GLOP	**G**oogle **L**inear **Op**timizer	LP-Solver in OR-Tools	Open Source
SCIP	Solving Constraint Integer Programs	MILP/MINLP-Solver	akademisch frei
CBC	**C**oin-or **B**ranch and **C**ut	MILP-Solver des COIN-OR-Projekts	Open Source
GLPK	GNU Linear Programming Kit	LP/MILP-Solver	Open Source (GPL)
OSQP	**O**perator **S**plitting **Q**uadratic **P**rogram	QP-Solver	Open Source
ECOS	Embedded Conic Solver	konischer Solver	Open Source
SCS	Splitting Conic Solver	konischer Solver	Open Source
IPOPT	**I**nterior **Po**int **Opt**imizer	NLP-Solver	Open Source (EPL)
Gurobi, **CPLEX**, **Xpress**	—	kommerzielle Hochleistungs-Solver	kostenpflichtig
API	Application Programming Interface	Programmierschnittstelle	—
DSL	Domain-Specific Language	fachspezifische Sprache (hier: Modellierungssprache)	—
CSR	Compressed Sparse Row	Speicherformat für dünnbesetzte Matrizen	—
GIL	Global Interpreter Lock	Pythons Sperre, die echte Parallelität von Python-Code verhindert — Solverbibliotheken in C++ geben sie frei (Kapitel 23)	—

Kürzel	Ausgeschrieben	Was es ist	Lizenz
JSON	JavaScript Object Notation	Textformat für strukturierte Daten; Austauschformat des Optimierungsdienstes	—
RHS	Right-Hand Side	rechte Seite einer (Un-)Gleichung, der Vektor **b**	—
FFI	Foreign Function Interface	Aufruf von C/C++-Code aus Python	—

2.5 Begriffe aus dem Solver-Alltag

Kürzel / Begriff	Bedeutung	Was Sie tun sollten
OPTIMAL	Beweisbar beste Lösung gefunden	Nichts — alles gut
FEASIBLE	Zulässige Lösung gefunden, Optimalität nicht bewiesen	Zeitlimit erhöhen oder Gap akzeptieren
INFEASIBLE	Keine Lösung erfüllt alle Bedingungen	Anhang C: Diagnose durchführen
UNBOUNDED	Zielfunktion unbeschränkt verbesserbar	Fehlende Nebenbedingung — Modell prüfen
MIP-Gap	Relativer Abstand beste Lösung ↔ beste Schranke	1–2 % sind in der Praxis meist genug
Incumbent	Beste bisher gefundene zulässige Lösung	Referenz für das Pruning
Relaxation	Modell mit weggelassener Ganzzahligkeit	liefert die Schranke
Warm Start	Solver startet von bekannter Lösung	beschleunigt wiederholte Läufe
Slack	Schlupf, ungenutzte Kapazität	Slack = 0 → Engpass
Lookahead-Bias	Nutzung von Daten aus der Zukunft	Backtest ist wertlos — Kapitel 22

3. Konventionen in diesem Buch

Vektoren sind grundsätzlich **Spaltenvektoren** und werden fett gesetzt: $\mathbf{x}$. Ein transponierter Vektor $\mathbf{x}^\top$ ist eine Zeile. Skalare bleiben mager: n, Z.

Matrizen sind fett und groß: $\mathbf{A}$, Σ.

Der Stern markiert immer Optimalwerte: $\mathbf{x}^*$, Z^*, λ^*.

Minimierung ist die Standardform. Jede Maximierung lässt sich durch Vorzeichenwechsel in eine Minimierung überführen: $\max f(\mathbf{x}) = -\min(-f(\mathbf{x}))$. Die meisten Solver — auch `scipy.optimize.linprog` — minimieren intern. Wer das vergisst, erhält systematisch das Gegenteil des Gewünschten; siehe Abschnitt 5.7.

Dezimaltrennzeichen. Im Fließtext und in Formeln steht das deutsche Komma (3,14), in Code und Programmausgaben der englische Punkt (`3.14`) — Python kennt nichts anderes.

Code-Sprache. Kommentare und Ausgaben der Beispielprogramme sind deutsch, Bezeichner überwiegend englisch, weil das der Konvention der verwendeten Bibliotheken entspricht und die Programme so anschlussfähig an die Originaldokumentation bleiben.

Anglizismen. Fachbegriffe werden bei der ersten Verwendung deutsch eingeführt und der englische Originalbegriff *kursiv* in Klammern ergänzt, weil Sie ihn für die Recherche brauchen: „zulässiger Bereich (*feasible region*)“.

Teil I: Grundlagen des Operations Research

Kapitel 1: Einführung in Operations Research — Vom Ursprung zur mathematischen Entscheidungsfindung

Kapitel auf einen Blick

Worum geht es? Was Operations Research ist, warum Ausprobieren ab einer gewissen Problemgröße hoffnungslos wird, und aus welchen vier Bausteinen *jedes* Optimierungsmodell besteht. Los geht es mit einem Problem, das Sie in fünf Minuten selbst lösen.

Voraussetzungen: Keine. Dies ist der Einstieg.

Danach können Sie: Ein Alltagsproblem in Entscheidungsvariablen, Parameter, Zielfunktion und Nebenbedingungen zerlegen, ein erstes Modell mit Google OR-Tools lösen — und die Daten dafür aus einer Excel-Mappe holen und das Ergebnis dorthin zurückschreiben.

Zeitbedarf: ca. 3,5 Stunden inklusive Übungen.

Programme:
`Bot_Allokation.py`
`Brute_Force_Vergleich.py`

`Bausteine_Vorlage.py`
`Excel_Bruecke.py`

Notebook: `Notebooks_04/einfuehrung.ipynb`

1.1 In 5 Minuten gelöst

Bevor wir über Theorie sprechen, lösen Sie ein echtes Problem. Kopieren Sie den folgenden Block, führen Sie ihn aus, sehen Sie das Ergebnis — die Erklärung kommt danach.

In 5 Minuten gelöst: Der Wochenplan einer Schreinerei

Eine Schreinerei fertigt Tische und Stühle. Nächste Woche stehen **150 Montagestunden** und **240 m² Plattenmaterial** zur Verfügung. Die Frage des Chefs: *Was sollen wir bauen?*

Produkt	Deckungsbeitrag	Montagezeit	Plattenmaterial
Tisch	240 €	3,0 h	6,0 m²
Stuhl	60 €	1,0 h	1,0 m²

```python
from ortools.linear_solver import pywraplp

produkt = {"Tisch": (240, 3.0, 6.0), "Stuhl": (60, 1.0, 1.0)}   # DB, Stunden, m²
vorrat = {"Montagestunden": 150, "Plattenmaterial": 240}

s = pywraplp.Solver.CreateSolver("GLOP")
x = {p: s.NumVar(0, s.infinity(), p) for p in produkt}
s.Add(sum(x[p] * produkt[p][1] for p in produkt) <= vorrat["Montagestunden"])
s.Add(sum(x[p] * produkt[p][2] for p in produkt) <= vorrat["Plattenmaterial"])
s.Maximize(sum(x[p] * produkt[p][0] for p in produkt))
s.Solve()

for p in produkt:
    print(f"{p:6}: {x[p].solution_value():5.0f} Stueck")
print(f"Deckungsbeitrag: {s.Objective().Value():.0f} EUR")
```

Ausgabe:

```
Tisch :    30 Stueck
Stuhl :    60 Stueck
Deckungsbeitrag: 10800 EUR
```

Und jetzt der Punkt, auf den es ankommt. Fragen Sie sich ehrlich: Was hätten *Sie* geantwortet? Die naheliegende Antwort lautet „möglichst viele Tische“ — der Tisch bringt mit 240 € schließlich den vierfachen Deckungsbeitrag eines Stuhls. Rechnen wir das durch:

Strategie	Menge	Deckungsbeitrag
Nur Tische (Bauchgefühl)	40 Tische (das Plattenmaterial ist zuerst leer)	9 600 €
Nur Stühle	150 Stühle (die Montagezeit ist zuerst leer)	9 000 €
Optimum des Solvers	**30 Tische + 60 Stühle**	**10 800 €**

Das Bauchgefühl kostet **1 200 € pro Woche**, rund 62 000 € im Jahr — bei einem Problem mit sage und schreibe zwei Produkten und zwei Ressourcen. Genau das ist Operations Research: nicht das Ausrechnen von Zahlen, die man ohnehin kennt, sondern das Auffinden von Entscheidungen, auf die man von allein nicht kommt.

Warum funktioniert das? Der Solver macht sich zunutze, dass keines der beiden Produkte in *beiden* Ressourcen das bessere ist. Pro Montagestunde bringt der Tisch 80 €, der Stuhl nur 60 € — pro Quadratmeter Plattenmaterial dagegen bringt der Stuhl 60 €, der Tisch nur 40 €. Wo kein Produkt durchgehend gewinnt, gibt es eine Mischung, die beide Engpässe gleichzeitig ausschöpft. Diese Mischung von Hand zu finden ist schon bei zwei Produkten mühsam und bei zwanzig unmöglich. Wie der Solver sie findet, zeigt Kapitel 5; was die einzelnen Zeilen des Programms bedeuten, klärt der Rest dieses Kapitels.

1.2 Lernziele

Nach diesem Kapitel können Sie …

1. … in eigenen Worten erklären, wodurch sich Operations Research von Business Intelligence und maschinellem Lernen unterscheidet.
2. … abschätzen, ab welcher Problemgröße vollständiges Durchprobieren scheitert — und diese Abschätzung selbst rechnen.
3. … die vier Bausteine eines Modells benennen und einem konkreten Problem zuordnen.
4. … entscheiden, ob eine Bedingung eine **harte** oder eine **weiche** Nebenbedingung ist.
5. … ein einfaches Modell mit **Google OR-Tools** aufschreiben, lösen und das Ergebnis interpretieren.

1.3 Was ist Operations Research wirklich?

Im Alltag lösen wir Probleme intuitiv: durch Ausprobieren, Daumenregeln, Erfahrung. Das funktioniert erstaunlich gut — bis eine bestimmte Schwelle überschritten wird. Dann bricht die Intuition nicht langsam ein, sondern schlagartig zusammen.

Drei Beispiele:

- Ein Disponent soll 40 Lastwagen auf 300 Lieferorte verteilen. Es gibt mehr mögliche Routenkombinationen als Atome im bekannten Universum: $300! \approx 3{,}06 \times 10^{614}$.

- Eine Klinik muss 200 Pflegekräfte einteilen — unter Berücksichtigung von 15 gesetzlichen Arbeitszeitregeln, Urlaubsanträgen, Qualifikationsstufen und Notfallreserven.
- Ein Fondsmanager muss 10 Millionen Euro so auf 50 Aktien aufteilen, dass bei einer erwarteten Mindestrendite von 8 % das Verlustrisiko im schlechtesten Marktszenario minimal bleibt.

Ein naiver Ansatz — *Brute Force*, das vollständige Durchrechnen aller Möglichkeiten — würde selbst auf den schnellsten Supercomputern der Welt Milliarden Jahre dauern.

 Definition: Operations Research

Operations Research (OR), deutsch etwa *Unternehmensforschung* oder *mathematische Entscheidungsoptimierung*, ist die wissenschaftliche Disziplin, reale Entscheidungsprobleme in formale mathematische Modelle zu überführen und mithilfe exakter oder heuristischer Algorithmen (*Solver*) die beweisbar beste (**optimale**) oder eine messbar hochwertige Lösung zu finden.

Die drei Stufen der Analytik

OR beantwortet nicht die beschreibende Frage *„Was ist passiert?“* und nicht die vorhersagende Frage *„Was wird passieren?“*, sondern die **vorschreibende** (präskriptive) Kernfrage:

„Was ist unter den gegebenen Bedingungen die beste Handlung?“

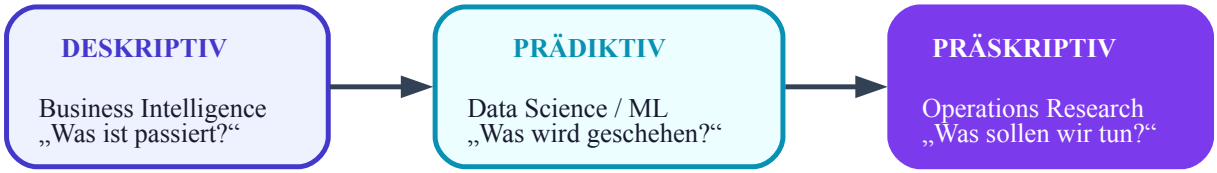

Abbildung 1: Abb. 1.1: Analytik-Reifegrad – vom Beschreiben zum Entscheiden

Stufe	Frage	Typisches Werkzeug	Ergebnis
Deskriptiv	Was ist passiert?	Business Intelligence, Dashboards	Ein Bericht
Prädiktiv	Was wird geschehen?	Statistik, maschinelles Lernen	Eine Prognose
Präskriptiv	Was sollen wir tun?	**Operations Research**	Eine **Handlungsanweisung**

Die Stufen bauen aufeinander auf, ersetzen einander aber nicht. Eine Umsatzprognose sagt Ihnen, dass die Nachfrage im März um 20 % steigt. Sie sagt Ihnen nicht, welche der sieben Maschinen Sie dafür wann mit welchem Auftrag belegen sollen. Genau dort beginnt OR — und deshalb ist eine Prognose in diesem Buch fast immer nur eine **Eingangsgröße** des Modells, nie sein Ergebnis.

 Merksatz Maschinelles Lernen sagt Ihnen, was kommt. Operations Research sagt Ihnen, was Sie deswegen tun sollen.

1.4 Warum Ausprobieren scheitert — mit eigener Rechnung

Die Behauptung „vollständiges Durchprobieren ist unmöglich“ wird oft aufgestellt und selten belegt. Rechnen wir sie einmal selbst nach — das Ergebnis prägt sich besser ein als jede Zahl im Text.

Handrechnung 1.1: Wie schnell wächst „alle Möglichkeiten“?

Angenommen, Sie sollen n Aufträge in eine Reihenfolge bringen. Für den ersten Platz haben Sie n Kandidaten, für den zweiten $n-1$, für den dritten $n-2$ und so fort. Die Gesamtzahl der Reihenfolgen ist damit die **Fakultät**:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

Nehmen wir großzügig an, ein Rechner prüfe **eine Milliarde** (10^9) Reihenfolgen pro Sekunde. Wie lange dauert es?

n	$n!$	Rechenzeit bei 10^9 Prüfungen/s
5	120	unmessbar kurz
10	3 628 800	0,004 Sekunden
15	$1{,}3 \times 10^{12}$	22 Minuten
20	$2{,}4 \times 10^{18}$	**77 Jahre**
25	$1{,}6 \times 10^{25}$	490 Millionen Jahre
30	$2{,}7 \times 10^{32}$	$8{,}4 \times 10^{15}$ Jahre

Der springende Punkt: Zwischen $n = 15$ (22 Minuten, machbar) und $n = 20$ (77 Jahre, hoffnungslos) liegen nur **fünf zusätzliche Aufträge**. Es gibt keine sanfte Verschlechterung — es gibt eine Wand.

Und die Rechnung ist noch geschönt: Ein schnellerer Rechner hilft praktisch nicht. Selbst eine Beschleunigung um den Faktor **eine Million** verschiebt die Grenze von $n = 20$ nur auf etwa $n = 24$. Gegen kombinatorisches Wachstum ist Hardware machtlos.

Das folgende Programm macht diesen Effekt erfahrbar. Es löst dasselbe Zuordnungsproblem zweimal — einmal durch vollständiges Durchprobieren, einmal mit einem OR-Solver — und vergleicht die Laufzeiten.

```
#!/usr/bin/env python3

# Brute_Force_Vergleich.py
"""
Kapitel Einfuehrung: Warum Ausprobieren scheitert.
Vergleicht vollständige Enumeration mit einem Constraint-Solver an einem
Zuordnungsproblem wachsender Größe (n Mitarbeiter auf n Aufgaben).
"""
```

```
import itertools
import math
import time

import numpy as np
from ortools.sat.python import cp_model


def erzeuge_kostenmatrix(n: int, seed: int = 7) -> np.ndarray:
    """Zufällige, aber reproduzierbare Kosten: Wer bearbeitet welche Aufgabe wie teuer?"""
    rng = np.random.default_rng(seed)
    return rng.integers(low=10, high=100, size=(n, n))


def loese_brute_force(kosten: np.ndarray) -> tuple[float, tuple, int]:
    """
    Probiert ALLE n! Zuordnungen durch und behält die beste.
    Rückgabe: (bester Kostenwert, beste Permutation, Anzahl geprüfter Kombinationen)
    """
    n = len(kosten)
    bester_wert = math.inf
    beste_zuordnung = None
    geprueft = 0

    # itertools.permutations(range(n)) liefert nacheinander jede Reihenfolge
    for zuordnung in itertools.permutations(range(n)):
        # zuordnung[i] = Aufgabe, die Mitarbeiter i übernimmt
        wert = sum(kosten[i][zuordnung[i]] for i in range(n))
        geprueft += 1
        if wert < bester_wert:
            bester_wert = wert
            beste_zuordnung = zuordnung

    return bester_wert, beste_zuordnung, geprueft


def loese_mit_solver(kosten: np.ndarray) -> tuple[float, tuple]:
    """Dasselbe Problem als Constraint-Programm – der Solver probiert NICHT alles durch."""
    n = len(kosten)
    modell = cp_model.CpModel()

    # x[i][j] = 1  <=>  Mitarbeiter i übernimmt Aufgabe j
    x = [[modell.NewBoolVar(f"x_{i}_{j}") for j in range(n)] for i in range(n)]

    for i in range(n):
        modell.AddExactlyOne(x[i][j] for j in range(n))   # jeder genau eine Aufgabe
    for j in range(n):
```

```
        modell.AddExactlyOne(x[i][j] for i in range(n))   # jede Aufgabe genau einmal

    modell.Minimize(sum(int(kosten[i][j]) * x[i][j] for i in range(n) for j in range(n)))

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = 30.0
    status = loeser.Solve(modell)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("Solver fand keine Lösung.")

    zuordnung = tuple(
        next(j for j in range(n) if loeser.Value(x[i][j]) == 1) for i in range(n)
    )
    return loeser.ObjectiveValue(), zuordnung


if __name__ == "__main__":
    print("=" * 72)
    print("   VOLLSTÄNDIGES DURCHPROBIEREN  vs.  OPERATIONS RESEARCH")
    print("=" * 72)
    print(f"{'n':>3} | {'Kombinationen':>18} | {'Brute Force':>14} | {'Solver':>10} |
     ↪  gleich?")
    print("-" * 72)

    for n in [4, 6, 8, 9, 10]:      # n = 11 dauert bereits ~2 Minuten
        kosten = erzeuge_kostenmatrix(n)

        t0 = time.perf_counter()
        wert_bf, zuord_bf, geprueft = loese_brute_force(kosten)
        zeit_bf = time.perf_counter() - t0

        t0 = time.perf_counter()
        wert_or, zuord_or = loese_mit_solver(kosten)
        zeit_or = time.perf_counter() - t0

        gleich = "ja" if abs(wert_bf - wert_or) < 1e-9 else "NEIN!"
       print(f"{n:>3} | {geprueft:>18,} | {zeit_bf:>11.4f} s | {zeit_or:>7.4f} s | {gleich}")

    print("-" * 72)
    # Hochrechnung: Wie lange bräuchte Brute Force bei n = 20?
    kombis_20 = math.factorial(20)
    # Messbasis: Prüfungen pro Sekunde aus dem letzten Lauf schätzen
    pro_sekunde = geprueft / max(zeit_bf, 1e-9)
    jahre = kombis_20 / pro_sekunde / (60 * 60 * 24 * 365.25)
    print(f"Hochrechnung für n = 20: {kombis_20:,} Kombinationen")
    print(f"Bei gemessenen {pro_sekunde:,.0f} Prüfungen/s wären das {jahre:,.0f} Jahre.")
    print("Der Solver löst dieselbe Instanz in Sekundenbruchteilen.")
    print("=" * 72)
```

Erwartete Ausgabe (Zeiten hardwareabhängig):

```
======================================================================
   VOLLSTÄNDIGES DURCHPROBIEREN  vs.  OPERATIONS RESEARCH
======================================================================
  n |      Kombinationen |    Brute Force |     Solver | gleich?
----------------------------------------------------------------------
  4 |                  24 |      0.0001 s |  0.0408 s | ja
  6 |                 720 |      0.0021 s |  0.0067 s | ja
  8 |              40,320 |      0.1088 s |  0.0088 s | ja
  9 |             362,880 |      0.1596 s |  0.0120 s | ja
 10 |           3,628,800 |     12.0897 s |  0.0252 s | ja
----------------------------------------------------------------------
Hochrechnung für n = 20: 2,432,902,008,176,640,000 Kombinationen
Bei gemessenen 300,157 Prüfungen/s wären das 256,845 Jahre.
Der Solver löst dieselbe Instanz in Sekundenbruchteilen.
======================================================================
```

Lesen Sie die Tabelle von oben nach unten: Bei $n = 4$ ist das Durchprobieren noch **400-mal schneller** als der Solver — der Solveraufruf hat einen festen Startaufwand von einigen Millisekunden, der sich bei winzigen Problemen nicht lohnt. Ab $n = 8$ kippt das Verhältnis, und ab $n = 10$ ist der Solver bereits **480-mal schneller**. Entscheidend ist nicht der Faktor, sondern die *Richtung*: Die Brute-Force-Zeit verzehnfacht sich mit jedem zusätzlichen Element, die Solver-Zeit wächst kaum merklich.

Code-Durchgang

Zeile / Stelle	Was passiert	Warum
`itertools.permutations(range(n))`	erzeugt nacheinander alle $n!$ Reihenfolgen	die Definition von „alles durchprobieren“
`wert = sum(kosten[i][zuordnung[i]] ...)`	bewertet eine einzelne Kombination	das ist die **Zielfunktion**, nur eben von Hand ausgewertet
`modell.NewBoolVar(...)`	legt eine Ja/Nein-**Entscheidungsvariable** an	n^2 Variablen statt $n!$ Kombinationen
`modell.AddExactlyOne(...)`	„genau eine davon ist wahr“	die **Nebenbedingungen**
`modell.Minimize(...)`	die zu minimierende Summe	die **Zielfunktion**, jetzt deklariert statt ausgerechnet
`loeser.Solve(modell)`	der Solver sucht — aber nicht stur	er nutzt Struktur (Schranken, Propagation), statt abzuzählen

Die entscheidende Beobachtung: Beide Programme liefern denselben Zielwert (Spalte „gleich?“). Der Solver rät also nicht und liefert keine Näherung — er findet dasselbe beweisbare Optimum, nur ohne alles anzusehen.

Typische Fehler

- **„Dann nehme ich eben eine schnellere Sprache.“** C++ ist etwa 50-mal schneller als Python. Bei $n = 20$ verkürzt das 157 Millionen Jahre auf 3 Millionen Jahre. Das Problem ist nicht die Sprache, sondern der Ansatz.
- **„Ich probiere nur die plausiblen Kombinationen.“** Genau das tut ein Solver — nur systematisch, mit einem Beweis, dass die weggelassenen Zweige nichts Besseres enthalten. Eine handgestrickte Auswahl hat diesen Beweis nicht.

1.5 Historischer Kontext und Evolution

Der Begriff entstand im Vorfeld des Zweiten Weltkriegs in Großbritannien. Militärische Führungsstäbe standen vor neuartigen logistischen und strategischen Fragen: Wie platziert man die neu entwickelten Radaranlagen an der Küste optimal? Wie groß müssen Schiffskonvois sein, um U-Boot-Angriffe bei minimalem Geleitschutzaufwand abzuwehren?

Wissenschaftler wie **George Dantzig**, **Patrick Blackett** und **John von Neumann** entwickelten daraufhin mathematische Formalismen. 1947 erfand George Dantzig den **Simplex-Algorithmus** zur Lösung linearer Programme — ein Durchbruch, der die industrielle Planung in den 1950er-Jahren revolutionierte und den wir in Kapitel 5 selbst programmieren werden.

Heute treibt OR die Kernsysteme moderner Industrien an:

- **Tech-Konzerne:** Server-Scheduling, Datenrouting in Glasfasernetzen, Werbeplatzierungsauktionen.
- **Luftfahrt & Logistik:** Crew-Scheduling, Flugzeugumlaufplanung, Paketlogistik (etwa das System *ORION* von UPS, das durch Routenoptimierung jährlich zweistellige Millionenbeträge an Treibstoff einspart).
- **Energie:** Kraftwerkseinsatzplanung, Netzausbau, Speicherbewirtschaftung.
- **Gesundheitswesen:** OP-Saal-Belegung, Dienstpläne, Rettungsmittel-Standorte.
- **Finanzindustrie:** Arbitrage-Freiheit, Portfolio-Optimierung, Liquiditäts- und Orderausführungsmanagement — der Gegenstand von Teil IV dieses Buches.

1.6 Die vier universellen Bausteine jedes OR-Problems

Dies ist der wichtigste Abschnitt des Kapitels. Jedes Optimierungsproblem der Welt lässt sich — unabhängig von der Branche — auf vier Elemente reduzieren. Wenn Sie später vor einem neuen Problem sitzen, ist dies Ihre Checkliste.

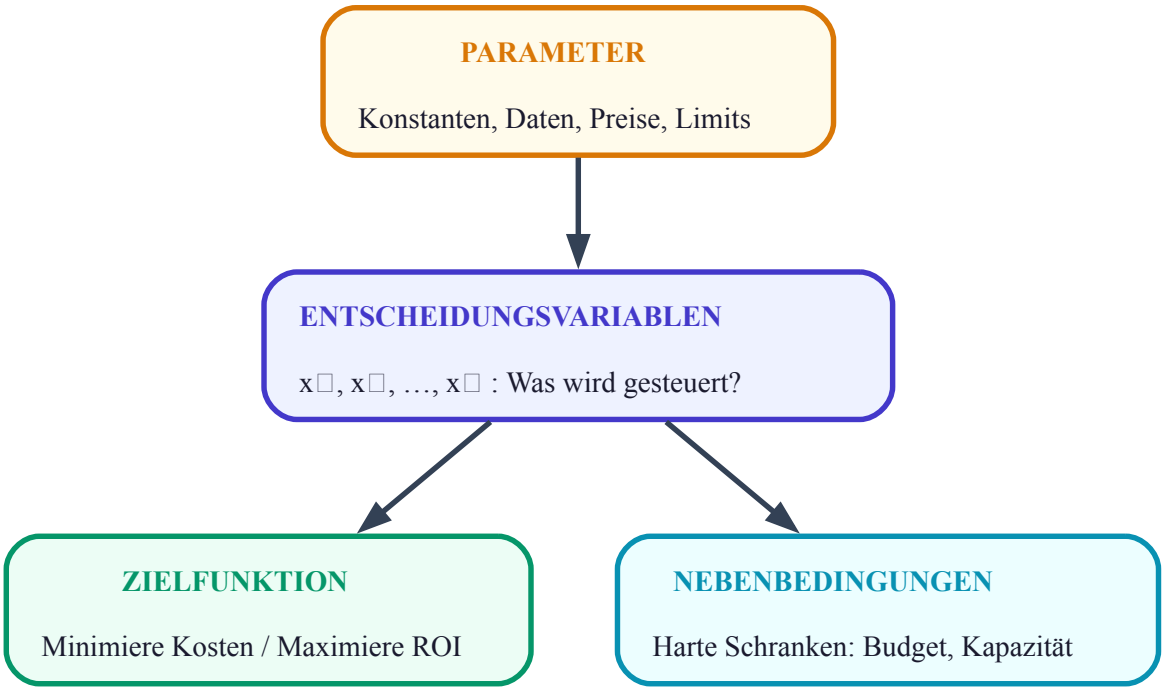

Abbildung 2: Abb. 1.2: Die vier universellen Bausteine eines OR-Modells

Baustein 1 — Entscheidungsvariablen (x)

Die **Stellschrauben** des Modells: Werte, die der Algorithmus aktiv festlegen darf.

Typ	Notation	Beispiel	Problemklasse
Kontinuierlich	$x \in \mathbb{R}$	Investitionsbetrag in Euro, Fördermenge in Litern	LP — leicht
Ganzzahlig	$x \in \mathbb{Z}$	Anzahl produzierter Maschinen, Anzahl Mitarbeiter	MILP — schwer
Binär	$x \in \{0, 1\}$	„Aktie i kaufen: ja/nein“, „Schicht zuweisen: ja/nein“	MILP — schwer

> **Die Leitfrage:** *Worüber darf ich überhaupt entscheiden?* Alles, was feststeht, ist kein Variable, sondern Parameter. Die häufigste Ursache für unbrauchbare Modelle ist eine falsch gewählte Variablendefinition — nicht ein falscher Solver.

Baustein 2 — Parameter / Eingabedaten (c, A, b)

Die **unveränderlichen Fakten** der Realität: historische Renditen, Maschinenstundensätze, Lagerkapazitäten, Steuerquoten, Entfernungen, Qualifikationen.

> **Die Leitfrage:** *Was ist gegeben und wird nicht entschieden?* Parameter sind der Ort, an dem Datenqualität über Modellqualität entscheidet. Kapitel 18 zeigt am Beispiel der Kovarianzmatrix, wie ein schlecht geschätzter Parameter ein mathematisch perfektes Modell wertlos macht.

Baustein 3 — Zielfunktion ($f(x)$)

Ein mathematischer Ausdruck, der die **Güte** einer Lösung zu **einer einzigen Zahl** verdichtet:

$$\min_x f(x) \qquad \text{oder} \qquad \max_x f(x)$$

 Formel-Lesehilfe

- $f(x)$ — die Bewertungsfunktion: „Wie gut ist die Lösung x?“
- $\min_x$ — „wähle dasjenige x, das f so klein wie möglich macht“
- Das x unter dem min sagt, **worüber** minimiert wird (über die Variablen, nicht etwa über die Parameter).

Ohne Formel gesagt: Von allen erlaubten Handlungsmöglichkeiten suchen wir diejenige mit den geringsten Kosten (bzw. dem höchsten Gewinn).

Die Leitfrage: *Was genau soll besser werden — und in welcher Einheit?* Wenn Sie die Einheit nicht nennen können (Euro, Minuten, Strafpunkte), ist die Zielfunktion noch nicht fertig. Mehrere gleichzeitige Ziele müssen entweder gewichtet in **eine** Zahl überführt oder hierarchisch nacheinander optimiert werden.

Baustein 4 — Nebenbedingungen (*Constraints*)

Regeln, die den Raum der erlaubten Lösungen einschränken:

$$g_i(\mathbf{x}) \le b_i \qquad \text{bzw.} \qquad h_j(\mathbf{x}) = 0$$

Harte Nebenbedingungen (*hard constraints*) sind zwingend: gesetzliche Ruhezeit ≥ 11 Stunden, Summe der Portfoliogewichte $= 100\,\%$, maximales Verlustrisiko $\le 5\,\%$. Wird eine einzige harte Bedingung verletzt, ist die Lösung mathematisch **unzulässig** (*infeasible*) — sie existiert für den Solver schlicht nicht.

Weiche Nebenbedingungen (*soft constraints*) sind Wünsche: Mitarbeiterpräferenzen, möglichst wenig Umschichtung, gleichmäßige Lastverteilung. Sie werden über **Strafkosten** (*penalties*) in die Zielfunktion integriert und dürfen im Notfall verletzt werden — es kostet dann eben.

 Merksatz Harte Bedingung = „darf nicht“. Weiche Bedingung = „soll möglichst nicht, sonst kostet es X“. Die Entscheidung zwischen beiden ist eine der folgenreichsten im ganzen Modell: Zu viele harte Bedingungen erzeugen unlösbare Modelle (Kapitel 22), zu wenige erzeugen Lösungen, die niemand akzeptiert.

Die vier Bausteine als eine Formel

Setzt man die vier Bausteine zusammen, entsteht die Standardform, in der Ihnen jedes Optimierungsmodell dieses Buches begegnen wird. Sie sieht auf den ersten Blick sperrig aus — deshalb steht daneben, was jede Zeile in Alltagssprache bedeutet.

$$\max_{\mathbf{x}} \sum_{j=1}^{n} c_j x_j \qquad \text{u. d. N.} \qquad \sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \forall i = 1, \dots, m, \qquad x_j \ge 0$$

Formel-Übersetzer

Mathematik	Alltagssprache
x_j	Wie viel bauen / kaufen / einplanen wir von Produkt j? Das ist die Stellschraube — der Solver entscheidet darüber.
c_j	Was bringt eine Einheit von j? Im Schreinerei-Beispiel: 240 € beim Tisch, 60 € beim Stuhl.
$\sum_j c_j x_j$	„Zähle über alle Produkte zusammen: Menge mal Stückertrag.“ Das ist der Gesamtdeckungsbeitrag.
$\max_{\mathbf{x}}$	„Suche unter allen erlaubten Mengenkombinationen die mit dem höchsten Gesamtertrag.“ Das **x** darunter sagt nur, worüber entschieden wird.
a_{ij}	Wie viel von Ressource i verbraucht eine Einheit von Produkt j? Ein Tisch verbraucht 3 Montagestunden.
b_i	Wie viel von Ressource i ist überhaupt da? 150 Montagestunden.
$\sum_j a_{ij} x_j \le b_i$	„Was alle Produkte zusammen von Ressource i verbrauchen, darf den Vorrat nicht überschreiten.“
$\forall i = 1, \dots, m$	„und zwar für jede einzelne Ressource“ — aus einer geschriebenen Zeile werden im Modell m Nebenbedingungen.
$x_j \ge 0$	„Negative Mengen gibt es nicht.“ Trivial für Menschen, muss dem Solver aber gesagt werden.

> **Der ganze Ausdruck in einem Satz:** *Verteile die knappen Ressourcen so auf die Produkte, dass der Gesamtertrag maximal wird und kein Vorrat überzogen wird.*

Das ∀ („für alle“) ist dabei das Zeichen mit der größten praktischen Wirkung: Es verwandelt eine Zeile Mathematik in eine `for`-Schleife und damit in beliebig viele Nebenbedingungen. Im Schreinerei-Programm oben steckt es in den zwei `s.Add(...)`-Zeilen — bei 40 Ressourcen wären es 40, geschrieben in derselben einen Schleife.

Die Bausteine in der Praxis: eine Vorlage

Damit die Zerlegung zur Gewohnheit wird, hier eine Vorlage, die Sie bei jedem neuen Problem ausfüllen können — vor der ersten Codezeile.

Baustein	Leitfrage	Ihr Problem
Entscheidungsvariablen	Worüber darf ich entscheiden? Welcher Typ?	
Parameter	Was ist gegeben? Woher kommen die Daten?	
Zielfunktion	Was soll besser werden? In welcher Einheit? Min oder Max?	
Harte Bedingungen	Was ist *unter keinen Umständen* erlaubt?	
Weiche Bedingungen	Was ist unerwünscht, aber im Notfall hinnehmbar? Wie teuer?	

Handrechnung 1.2: Zerlegen Sie dieses Problem

Eine Bäckerei backt Brot und Brötchen. Ein Brot bringt 2,50 € Deckungsbeitrag und braucht 0,5 kg Mehl und 4 Minuten Ofenzeit; 10 Brötchen bringen 3,00 € und brauchen 0,6 kg Mehl und 3 Minuten Ofenzeit. Verfügbar sind 90 kg Mehl und 600 Minuten Ofenzeit. Aus Vertragsgründen müssen mindestens 40 Brote gebacken werden.

Füllen Sie die Vorlage aus, bevor Sie weiterlesen. — Auflösung:

- **Variablen:** x_1 = Anzahl Brote, x_2 = Anzahl Brötchen-Zehnerpackungen; beide ganzzahlig ≥ 0.
- **Parameter:** Deckungsbeiträge $(2{,}50;\ 3{,}00)$; Mehlbedarf $(0{,}5;\ 0{,}6)$; Ofenzeit $(4;\ 3)$; Kapazitäten $(90;\ 600)$; Mindestmenge 40.
- **Zielfunktion:** max $2{,}5x_1 + 3{,}0x_2$ (Einheit: Euro).
- **Harte Bedingungen:** $0{,}5x_1 + 0{,}6x_2 \leq 90$ (Mehl); $4x_1 + 3x_2 \leq 600$ (Ofen); $x_1 \geq 40$ (Vertrag).
- **Weiche Bedingungen:** keine.

Dieses Problem lösen Sie in der Aufgabe *Die Bäckerei programmieren* (Abschnitt 1.9) selbst.

Die Vorlage lässt sich fast wörtlich in Code übersetzen — wer die fünf Zeilen ausgefüllt hat, hat den größten Teil der Modellierungsarbeit schon erledigt:

```
#!/usr/bin/env python3

# Bausteine_Vorlage.py
"""
Kapitel Einfuehrung: Die Bausteine-Vorlage als ausfuellbares Python-Geruest.

Uebersetzt die Vier-Bausteine-Vorlage aus dem Abschnitt 'Die vier
universellen Bausteine jedes OR-Problems' direkt in Code: Wer sie
auf Papier ausgefuellt hat, kann sie fast unveraendert in ein loesbares
Modell umsetzen. Angewendet auf das Baeckerei-Beispiel der Handrechnung.
"""
```

```
from dataclasses import dataclass
from ortools.sat.python import cp_model


@dataclass
class Bausteine:
    """Die vier Bausteine jedes OR-Problems - wortwoertlich aus der Vorlage
    im Abschnitt 'Die vier universellen Bausteine' uebernommen."""
    variablen: str
    parameter: str
    zielfunktion: str
    harte_bedingungen: str
    weiche_bedingungen: str

    def zeige(self) -> None:
        print("=" * 78)
        print("  AUSGEFUELLTE BAUSTEINE-VORLAGE")
        print("=" * 78)
        for feld, wert in [
            ("Entscheidungsvariablen", self.variablen),
            ("Parameter", self.parameter),
            ("Zielfunktion", self.zielfunktion),
            ("Harte Bedingungen", self.harte_bedingungen),
            ("Weiche Bedingungen", self.weiche_bedingungen),
        ]:
            print(f"{feld:<24}: {wert}")
        print("=" * 78)


# --- Baeckerei-Beispiel aus der Handrechnung --------------------------------
VORLAGE = Bausteine(
    variablen="x1 = Anzahl Brote, x2 = Anzahl Broetchen-Zehnerpackungen "
              "(beide ganzzahlig >= 0)",
    parameter="Deckungsbeitraege (2,50; 3,00) EUR; Mehlbedarf (0,5; 0,6) kg; "
              "Ofenzeit (4; 3) min; Kapazitaeten (90 kg, 600 min); Mindestmenge 40 Brote",
    zielfunktion="max 2,5*x1 + 3,0*x2  (Einheit: Euro Deckungsbeitrag)",
    harte_bedingungen="0,5*x1 + 0,6*x2 <= 90 (Mehl); 4*x1 + 3*x2 <= 600 (Ofen); "
                      "x1 >= 40 (Vertrag)",
    weiche_bedingungen="keine",
)


def loese_baeckerei():
    """Baut aus der ausgefuellten Vorlage mechanisch ein CP-SAT-Modell."""
    modell = cp_model.CpModel()
    x1 = modell.NewIntVar(40, 1000, "Brote")            # Vertrag: mindestens 40
    x2 = modell.NewIntVar(0, 1000, "Broetchen_Zehner")
```

```
    modell.Add(5 * x1 + 6 * x2 <= 900)      # Mehl, x10 fuer Ganzzahligkeit
    modell.Add(4 * x1 + 3 * x2 <= 600)      # Ofenzeit

    modell.Maximize(25 * x1 + 30 * x2)      # Deckungsbeitrag x10

    loeser = cp_model.CpSolver()
    status = loeser.Solve(modell)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise SystemExit(f"Keine Loesung: {loeser.StatusName(status)}")
    return loeser.Value(x1), loeser.Value(x2), loeser.ObjectiveValue() / 10


if __name__ == "__main__":
    VORLAGE.zeige()

    print("\n--- Aus der Vorlage mechanisch abgeleitetes Modell ---")
    brote, broetchen, gewinn = loese_baeckerei()
    print(f"Optimale Brote:              {brote}")
    print(f"Optimale Broetchen-Zehner:   {broetchen}")
    print(f"Maximaler Deckungsbeitrag:   {gewinn:.2f} EUR")
    print()
    print("Genau diese drei Codezeilen (Variablen, Add, Maximize) entstehen")
    print("direkt aus den fuenf Zeilen der Vorlage - das ist der ganze Trick.")
```

Erwartete Ausgabe:

```
================================================================================
  AUSGEFUELLTE BAUSTEINE-VORLAGE
================================================================================
Entscheidungsvariablen  : x1 = Anzahl Brote, x2 = Anzahl Broetchen-Zehnerpackungen (beide
 ↪  ganzzahlig >= 0)
Parameter               : Deckungsbeitraege (2,50; 3,00) EUR; Mehlbedarf (0,5; 0,6) kg;
 ↪  Ofenzeit (4; 3) min; Kapazitaeten (90 kg, 600 min); Mindestmenge 40 Brote
Zielfunktion            : max 2,5*x1 + 3,0*x2  (Einheit: Euro Deckungsbeitrag)
Harte Bedingungen       : 0,5*x1 + 0,6*x2 <= 90 (Mehl); 4*x1 + 3*x2 <= 600 (Ofen); x1 >= 40
 ↪  (Vertrag)
Weiche Bedingungen      : keine
================================================================================

--- Aus der Vorlage mechanisch abgeleitetes Modell ---
Optimale Brote:              42
Optimale Broetchen-Zehner:   115
Maximaler Deckungsbeitrag:   450.00 EUR

Genau diese drei Codezeilen (Variablen, Add, Maximize) entstehen
direkt aus den fuenf Zeilen der Vorlage - das ist der ganze Trick.
```

Mehl und Deckungsbeitrag stehen im selben Verhältnis für Brote wie für Brötchen (je 5 € pro kg Mehl) — deshalb ist die Aufteilung zwischen 42/115 nicht eindeutig, solange die 90 kg Mehl restlos verplant

sind und mindestens 40 Brote gebacken werden; **der maximale Deckungsbeitrag von 450 € dagegen ist es.**

1.7 Erstes Python-Vollbeispiel: Ressourcenallokation im Rechenzentrum

Szenario

Ein quantitatives Handelshaus betreibt Serverkapazitäten für zwei Algorithmentypen:

- **Algorithmus A (Arbitrage-Bot):** erzeugt 150 € Gewinn pro Tag, benötigt 2 Rechenkerne (vCPUs) und 4 GB Arbeitsspeicher.
- **Algorithmus B (Trendfolge-Bot):** erzeugt 250 € Gewinn pro Tag, benötigt 5 Rechenkerne und 6 GB Arbeitsspeicher.

Systemressourcen: höchstens 40 vCPUs, höchstens 60 GB RAM, und aus Marktliquiditätsgründen höchstens 8 Arbitrage-Bots.

Frage: Wie viele Instanzen von A und B maximieren den Tagesgewinn?

Mathematische Formulierung

Entscheidungsvariablen:

$$x_A \in \mathbb{N}_0, \quad x_B \in \mathbb{N}_0$$

Zielfunktion:

$$\max_{x_A, x_B} \; Z = 150\, x_A + 250\, x_B$$

Nebenbedingungen:

$$\begin{aligned} 2x_A + 5x_B &\le 40 && \text{(vCPU-Limit)} \\ 4x_A + 6x_B &\le 60 && \text{(RAM-Limit)} \\ x_A &\le 8 && \text{(Marktliquidität)} \\ x_A,\, x_B &\ge 0,\ \text{ganzzahlig} \end{aligned}$$

Formel-Lesehilfe

- x_A, x_B — Anzahl gestarteter Bots je Typ. Ganzzahlig, weil ein halber Bot nicht existiert.
- $Z = 150x_A + 250x_B$ — der Tagesgewinn in Euro: pro Arbitrage-Bot 150 €, pro Trendfolge-Bot 250 €.
- $2x_A + 5x_B \le 40$ — jeder A-Bot belegt 2 Kerne, jeder B-Bot 5; zusammen dürfen es höchstens 40 sein.

Ohne Formel gesagt: Starte so viele Bots wie möglich, aber verbrauche nicht mehr Rechenkerne und Arbeitsspeicher, als da sind — und nicht mehr als acht Arbitrage-Bots.

□ Handrechnung 1.3: Das Optimum zu Fuß finden

Bei nur zwei Variablen können wir die Kandidaten abgehen. Weil B mehr Gewinn bringt (250 € gegen 150 €), liegt die Vermutung nahe, möglichst viele B-Bots zu starten.

Kandidat (x_A, x_B)	vCPU $2x_A + 5x_B \leq 40$	RAM $4x_A + 6x_B \leq 60$	Z
$(0, 8)$	$40 \leq 40$ □	$48 \leq 60$ □	2000 €
$(3, 6)$	36 □	48 □	1950 €
$(5, 6)$	40 □	56 □	2250 €
$(8, 4)$	36 □	56 □	2200 €
$(7, 5)$	39 □	58 □	**2300 €**
$(6, 6)$	$42 > 40$ □	—	unzulässig
$(8, 5)$	$41 > 40$ □	—	unzulässig

Optimum: $x_A^* = 7$, $x_B^* = 5$, $Z^* = 2300$ **€.**

Beachten Sie das Überraschende: Die naive Strategie „nur den lukrativeren Bot“ $(0, 8)$ liefert **2000 €** — ganze 300 € weniger als die Mischung. Genau solche Effekte machen Optimierung nötig: Der beste Gesamtplan besteht selten aus lauter lokal besten Einzelentscheidungen.

Umsetzung mit Google OR-Tools

```python
#!/usr/bin/env python3

# Bot_Allokation.py
"""
Kapitel Einfuehrung: Erstes Optimierungsmodell mit Google OR-Tools (CP-SAT).
Problem: Server-Allokation für Trading-Bots.

Modell mit expliziter Nebenbedingung statt versteckter Variablengrenze,
vollständiger Statusauswertung und Ergebnisprüfung.
"""

from ortools.sat.python import cp_model

# --- Parameter (die "Fakten" des Problems) ---------------------------------
GEWINN_A, GEWINN_B = 150, 250      # Euro pro Bot und Tag
CPU_A, CPU_B = 2, 5                # vCPU-Bedarf je Bot
RAM_A, RAM_B = 4, 6                # GB Arbeitsspeicher je Bot
CPU_GESAMT, RAM_GESAMT = 40, 60    # verfügbare Kapazitäten
MAX_ARBITRAGE = 8                  # Marktliquiditätsgrenze


def loese_bot_allokation(zeitlimit_s: float = 10.0):
    """Baut das Modell, löst es und gibt eine Auswertung aus."""
```

```
    # 1. Modell instanziieren
    modell = cp_model.CpModel()

    # 2. Entscheidungsvariablen anlegen
    #    NewIntVar(untere_grenze, obere_grenze, name)
    #    Die obere Grenze ist bewusst großzügig; die echte Beschränkung
    #    formulieren wir unten als Nebenbedingung, damit das Modell die
    #    mathematische Formulierung 1:1 abbildet.
    x_a = modell.NewIntVar(0, 100, "Arbitrage_Bots")
    x_b = modell.NewIntVar(0, 100, "Trend_Bots")

    # 3. Nebenbedingungen definieren
    c_cpu = modell.Add(CPU_A * x_a + CPU_B * x_b <= CPU_GESAMT)      # vCPU-Limit
    c_ram = modell.Add(RAM_A * x_a + RAM_B * x_b <= RAM_GESAMT)      # RAM-Limit
    c_liq = modell.Add(x_a <= MAX_ARBITRAGE)                          # Marktliquidität

    # 4. Zielfunktion: Maximiere den Tagesgewinn
    modell.Maximize(GEWINN_A * x_a + GEWINN_B * x_b)

    # 5. Solver konfigurieren und ausführen
    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = zeitlimit_s
    status = loeser.Solve(modell)

    # 6. Status auswerten -- IMMER alle Fälle behandeln
    status_text = loeser.StatusName(status)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        print(f"Keine verwertbare Lösung. Solver-Status: {status_text}")
        if status == cp_model.INFEASIBLE:
            print("Das Modell ist unlösbar - die Bedingungen widersprechen sich.")
        return None

    a, b = loeser.Value(x_a), loeser.Value(x_b)
    gewinn = loeser.ObjectiveValue()

    # 7. Ergebnis ausgeben
    print("=" * 58)
    print("  OPTIMALE BOT-ALLOKATION")
    print("=" * 58)
    print(f"Solver-Status:         {status_text}"
          f"{'  (beweisbar optimal)' if status == cp_model.OPTIMAL else '  (zulässig, nicht
           ↪ bewiesen)'}")
    print(f"Arbitrage-Bots (x_A):  {a} Instanzen")
    print(f"Trendfolge-Bots (x_B): {b} Instanzen")
    print(f"Täglicher Max-Gewinn:  {gewinn:,.2f} EUR")

    genutzt_cpu = CPU_A * a + CPU_B * b
    genutzt_ram = RAM_A * a + RAM_B * b
```

```
    print("-" * 58)
    print(f"CPU-Auslastung:        {genutzt_cpu} / {CPU_GESAMT} vCPUs "
          f"({genutzt_cpu / CPU_GESAMT * 100:.1f} %)")
    print(f"RAM-Auslastung:        {genutzt_ram} / {RAM_GESAMT} GB "
          f"({genutzt_ram / RAM_GESAMT * 100:.1f} %)")
    print(f"Arbitrage-Limit:       {a} / {MAX_ARBITRAGE}")

    # 8. Selbstkontrolle: Sind wirklich alle Bedingungen eingehalten?
    assert genutzt_cpu <= CPU_GESAMT, "vCPU-Limit verletzt!"
    assert genutzt_ram <= RAM_GESAMT, "RAM-Limit verletzt!"
    assert a <= MAX_ARBITRAGE, "Arbitrage-Limit verletzt!"

    # 9. Zum Vergleich: die naive Strategie "nur der lukrativere Bot"
    naiv_b = min(CPU_GESAMT // CPU_B, RAM_GESAMT // RAM_B)
    naiv_gewinn = GEWINN_B * naiv_b
    print("-" * 58)
    print(f"Naive Strategie (nur Trendfolge): {naiv_b} Bots -> {naiv_gewinn:,.2f} EUR")
    print(f"Vorteil der Optimierung:          {gewinn - naiv_gewinn:,.2f} EUR pro Tag "
          f"({(gewinn / naiv_gewinn - 1) * 100:.1f} %)")
    print("=" * 58)
    return a, b, gewinn


if __name__ == "__main__":
    loese_bot_allokation()
```

Erwartete Ausgabe:

```
==========================================================
  OPTIMALE BOT-ALLOKATION
==========================================================
Solver-Status:         OPTIMAL  (beweisbar optimal)
Arbitrage-Bots (x_A):  7 Instanzen
Trendfolge-Bots (x_B): 5 Instanzen
Täglicher Max-Gewinn:  2,300.00 EUR
----------------------------------------------------------
CPU-Auslastung:        39 / 40 vCPUs (97.5 %)
RAM-Auslastung:        58 / 60 GB (96.7 %)
Arbitrage-Limit:       7 / 8
----------------------------------------------------------
Naive Strategie (nur Trendfolge): 8 Bots -> 2,000.00 EUR
Vorteil der Optimierung:          300.00 EUR pro Tag (15.0 %)
==========================================================
```

Das Ergebnis deckt sich mit der Handrechnung *Das Optimum zu Fuß finden* — ein gutes Zeichen. **Prüfen Sie Solver-Ergebnisse wo immer möglich gegen eine unabhängige Rechnung**, gerade am Anfang.

□ Code-Durchgang

Stelle	Was passiert	Didaktischer Hinweis
Konstanten oben	alle Parameter an einer Stelle	Nie Zahlen im Modellcode verstreuen — sonst findet man sie beim Ändern nicht wieder
`NewIntVar(0, 100, ...)`	ganzzahlige Variable mit weiten Grenzen	Grenzen müssen endlich sein; 100 ist bewusst unkritisch groß
`modell.Add(x_A <= MAX_ARBITRAGE)`	Marktlimit als **Nebenbedingung**	Alternativ liesse sich dieses Limit auch in die Variablengrenze packen. Beides funktioniert — aber als Nebenbedingung bleibt das Modell zur mathematischen Formulierung deckungsgleich und man kann später den Schattenpreis abfragen
`loeser.StatusName(status)`	echter Status statt fester Text	Ein fest codierter Text wie „OPTIMAL“ würde unabhängig vom tatsächlichen Ergebnis ausgegeben — deshalb immer den echten Solver-Status auswerten
`assert ...`	Selbstkontrolle nach dem Lösen	Kostet nichts und fängt Modellierungsfehler, die der Solver nicht bemerken kann
Vergleich mit naiver Strategie	quantifiziert den Nutzen	Die Frage „Was bringt die Optimierung überhaupt?“ kommt in jedem Projekt — beantworten Sie sie ungefragt

Typische Fehler

- **Variablengrenzen zu eng gewählt.** `NewIntVar(0, 5, ...)` würde hier stillschweigend die Lösung $x_B = 5$ erzwingen und ein falsches „Optimum“ liefern. Grenzen sind Nebenbedingungen — behandeln Sie sie als solche.
- **Status nicht geprüft.** Ein Programm, das bei `INFEASIBLE` weiterläuft und `solver.Value(...)` abfragt, liefert entweder Unsinn oder stürzt ab.
- **Ganzzahligkeit vergessen.** Mit kontinuierlichen Variablen liefert dasselbe Modell $x_A = 7{,}5$, $x_B = 5{,}0$ und $Z = 2375$ € — ein Wert, den es real nicht gibt. Warum man solche Werte **nicht** einfach runden darf, ist Thema von Kapitel 6.

Warum dieses Programm alles selbst macht

Wenige Seiten weiter, in der Excel-Brücke, steht ein Programm, das mit 96 Zeilen auskommt, weil es die immer gleichen Arbeitsschritte aus dem gemeinsamen Modul `or_kern.py` bezieht (Abschnitt 22.6). Warum tut `Bot_Allokation.py` das nicht auch?

Weil es hier nicht darum geht, **wenig Code** zu schreiben, sondern darum zu sehen, wie ein Modell **entsteht**: Variablen, Nebenbedingungen, Zielfunktion, Status, Prüfung — jeder Schritt einzeln und sichtbar. Eine Abstraktion, die diese Schritte zusammenfasst, ist genau dann ein

Gewinn, wenn man sie schon einmal von Hand gegangen ist; vorher versteckt sie nur, was man verstehen will.

Das ist eine allgemeine Regel für eigene Projekte: **Erst die Wiederholung rechtfertigt die Abstraktion.** Wer sein erstes Modell gleich als Framework baut, hat ein Framework für einen Fall — und die Erfahrung, welche Fälle es tragen müsste, hat er noch nicht gemacht.

1.8 Von Excel zu Python: Ihre Daten liegen schon da

In den allermeisten Betrieben liegen die Zahlen, die ein Optimierungsmodell braucht, nicht in einer Datenbank, sondern in einer Tabellenkalkulation. Und viele Leserinnen und Leser kennen Optimierung bisher genau von dort: über *Daten* → *Solver* in Excel. Dieser Abschnitt schlägt die Brücke — in beide Richtungen.

Excel-Brücke: dieselben Begriffe, andere Werkzeuge

Im Excel-Solver	In Python	Anmerkung
„Veränderbare Zellen“	`x = solver.NumVar(...)`	Der Zellbereich wird zu benannten Variablen.
„Zielzelle“ + Max/Min	`solver.Maximize(...)`	Statt einer Formel in einer Zelle ein Ausdruck im Code.
„Nebenbedingungen“-Liste	`solver.Add(...)`	Eine Zeile je Regel — oder eine Schleife für 10 000 Regeln.
„Ganzzahlig“ / „Binär“	`NewIntVar(...)` / `NewBoolVar(...)`	Gleiche Bedeutung, gleiche Konsequenz für die Laufzeit.
Simplex-LP / GRG / Evolutionär	GLOP / SLSQP / CP-SAT	In Excel drei Auswahlpunkte, in Python drei Bibliotheken (Kapitel 3).
„Sensitivitätsbericht“	`bedingung.dual_value()`	Der Schattenpreis — Kapitel 5 erklärt, was er bedeutet.
200 Zeilen sind das Limit	keine praktische Grenze	Der Excel-Solver ist auf 200 veränderbare Zellen begrenzt.

Was Sie damit gewinnen ist weniger die Rechenleistung als die Nachvollziehbarkeit: Ein Python-Modell lässt sich versionieren, testen, in einer Schleife über hundert Szenarien laufen lassen und einem Kollegen erklären. Eine gewachsene Excel-Mappe mit Formeln quer über vier Blätter kann das alles nicht — und niemand traut sich mehr, sie anzufassen.

Das folgende Programm zeigt den ganzen Weg an einem Stück. Es legt die Eingabemappe beim ersten Start selbst an, damit Sie es sofort ausführen können; danach ersetzen Sie die Datei durch Ihre eigene. Solange die Spalten gleich heißen, ändert sich am Code nichts.

Zum Modul `or_kern.py`

Die vier Arbeitsschritte — lesen, lösen, prüfen, schreiben — stehen nicht in diesem Programm, sondern in `or_kern.py`. Das ist ein gemeinsamer Unterbau, den alle Programme dieses Buchs benutzen können; er liegt im selben Verzeichnis und ist in Abschnitt 22.6 vollständig abgedruckt und erklärt.

Sie müssen ihn jetzt noch nicht verstehen. Wichtig ist nur die Aufteilung: **Hier** steht, was diesen Fall ausmacht — dort steht, was jedes Optimierungsprogramm braucht. Genau deshalb ist der Code unten so kurz.

```
#!/usr/bin/env python3

# Excel_Bruecke.py
"""
Kapitel Einfuehrung: Vom Excel-Solver zu Python - dieselbe Rechnung, dieselbe Datei.

Das Programm bildet den vollstaendigen Arbeitsweg ab, den ein Excel-Modell im
Betrieb sonst von Hand geht:

    Tabellenblatt lesen -> Modell bauen -> loesen -> Ergebnis zurueckschreiben

Damit es ohne Vorbereitung laeuft, legt es die Eingabedatei beim ersten Start
selbst an. Ersetzen Sie sie durch Ihre eigene Datei - solange die Spalten
gleich heissen, aendert sich am Code nichts.

Die vier Arbeitsschritte kommen aus or_kern.py (Kapitel Praxisfallen). Was
hier steht, ist nur noch das, was diesen Fall ausmacht - der Rest ist
gemeinsamer Unterbau. Wer das Modul nicht kennt: Es ist im Buch vollstaendig
abgedruckt und liegt im selben Verzeichnis.

Erzeugte Dateien (im Arbeitsverzeichnis):
    produktionsmix.xlsx          Eingabe:  Blaetter 'Produkte' und 'Kapazitaeten'
    produktionsmix_ergebnis.xlsx  Ausgabe:  Blaetter 'Plan' und 'Kennzahlen'

Benoetigt: pandas, openpyxl, ortools, pydantic (ueber or_kern)
"""

from __future__ import annotations

import os

import pandas as pd

from or_kern import (lade_produktionsproblem, loese_mit_glop, pruefe_loesung,
                     schreibe_ergebnis)

# Die Mappen liegen NEBEN diesem Programm, nicht im aktuellen
# Arbeitsverzeichnis. Sonst haengt es davon ab, aus welchem Ordner man das
# Skript startet, wo die Dateien landen - und wer es aus der
```

```
# Repository-Wurzel aufruft, verstreut sie dort.
HIER = os.path.dirname(os.path.abspath(__file__))
EINGABE = os.path.join(HIER, "produktionsmix.xlsx")
AUSGABE = os.path.join(HIER, "produktionsmix_ergebnis.xlsx")


def lege_beispieldatei_an(pfad: str) -> None:
    """Schreibt die Eingabemappe, wie sie auch aus dem Controlling kaeme."""
    produkte = pd.DataFrame({
        "Produkt": ["Tisch", "Stuhl", "Regal"],
        "Deckungsbeitrag": [240.0, 60.0, 130.0],
        "Montagestunden": [3.0, 1.0, 2.0],
        "Plattenmaterial": [6.0, 1.0, 4.0],
    })
    kapazitaeten = pd.DataFrame({
        "Ressource": ["Montagestunden", "Plattenmaterial"],
        "Verfuegbar": [150.0, 240.0],
    })
    with pd.ExcelWriter(pfad, engine="openpyxl") as mappe:
        produkte.to_excel(mappe, sheet_name="Produkte", index=False)
        kapazitaeten.to_excel(mappe, sheet_name="Kapazitaeten", index=False)


if __name__ == "__main__":
    if not os.path.exists(EINGABE):
        lege_beispieldatei_an(EINGABE)
        print(f"Beispiel-Eingabedatei angelegt: {EINGABE}")

    # 1. Lesen und pruefen. Faellt hier etwas auf, ist der Fehler noch
    #    zuzuordnen - spaeter waere er nur noch eine merkwuerdige Zahl.
    problem = lade_produktionsproblem(EINGABE)

    # 2. Loesen. Einzige solverabhaengige Zeile des ganzen Programms -
    #    'loese_mit_scipy' waere ein Einzeiler-Wechsel.
    loesung = loese_mit_glop(problem)

    # 3. Abnahmepruefung gegen die Anforderungen, ohne den Solver zu fragen.
    beanstandungen = pruefe_loesung(problem, loesung)
    if beanstandungen:
        raise SystemExit("Abnahmepruefung fehlgeschlagen:\n  - "
                         + "\n  - ".join(beanstandungen))

    # 4. Zurueckschreiben - im Format, das die Fachabteilung ohnehin benutzt.
    schreibe_ergebnis(AUSGABE, problem, loesung)

    print("=" * 66)
    print("  PRODUKTIONSPLAN AUS DER EXCEL-MAPPE")
    print("=" * 66)
```

```
    for produkt in problem.produkte:
        menge = loesung.werte[produkt.name]
        print(f"{produkt.name:<18} {menge:8.1f} Stueck "
              f"{menge * produkt.deckungsbeitrag:12,.2f} EUR")
    print("-" * 66)
    print(loesung.als_bericht())
    print("Abnahmepruefung: bestanden\n")

    # Die Schattenpreise beantworten die Frage, die der Plan nicht stellt:
    # Was waere eine zusaetzliche Einheit dieser Ressource wert?
    for ressource, preis in loesung.schattenpreise.items():
        print(f"  eine Einheit {ressource:<18} mehr waere wert: {preis:7.2f} EUR")
    print("=" * 66)
    print(f"Ergebnis geschrieben nach: {AUSGABE}")
```

Erwartete Ausgabe:

```
Beispiel-Eingabedatei angelegt: produktionsmix.xlsx
==================================================================
  PRODUKTIONSPLAN AUS DER EXCEL-MAPPE
==================================================================
Tisch                  30.0 Stueck     7,200.00 EUR
Stuhl                  60.0 Stueck     3,600.00 EUR
Regal                   0.0 Stueck         0.00 EUR
------------------------------------------------------------------
Status: optimal | Zielwert: 10,800.00 | Gap: 0.00% | Zeit: 0.00s
Abnahmepruefung: bestanden

  eine Einheit Montagestunden     mehr waere wert:   40.00 EUR
  eine Einheit Plattenmaterial    mehr waere wert:   20.00 EUR
==================================================================
Ergebnis geschrieben nach: produktionsmix_ergebnis.xlsx
```

□ Code-Durchgang

Stelle	Was passiert	Warum es so gebaut ist
`lade_produktionsproblem(EINGABE)`	liest die Mappe **und prüft sie**	Leere Zellen, fehlende Spalten, Kapazitäten von null oder Ressourcen ohne Kapazitätsangabe scheitern hier — nicht erst beim Lösen, wo der Fehler nur noch eine merkwürdige Zahl wäre.
`loese_mit_glop(problem)`	die einzige solverabhängige Zeile	`loese_mit_scipy` wäre ein Einzeiler-Wechsel. Alles davor und danach bleibt unverändert.

Stelle	Was passiert	Warum es so gebaut ist
`pruefe_loesung(problem, loesung)`	Abnahmeprüfung **ohne** Solver	Sie bekommt nur die ausgegebene Lösung und die Anforderungen. Eine Prüfung aus denselben Bausteinen wie das Modell prüft das Modell gegen sich selbst.
`raise SystemExit(...)`	bricht bei Beanstandung ab	Ein Programm, das trotz durchgefallener Prüfung eine Datei schreibt, ist gefährlicher als eines, das abstürzt.
`loesung.schattenpreise`	Dualwerte aus dem Lösungsobjekt	Sie sind ein **solverunabhängiger** Begriff und stehen deshalb im Ergebnis, nicht im Solvercode.

Das Ergebnis liest sich von selbst: Das Regal — obwohl mit 130 € Deckungsbeitrag scheinbar attraktiv — wird gar nicht gebaut. Die Schattenpreise sagen warum: Ein Regal verbraucht 2 Montagestunden zu je 40 € und 4 m² Material zu je 20 €, bindet also Ressourcen im Wert von $2 \cdot 40 + 4 \cdot 20 = 160$ € und bringt nur 130 € ein. Es verdient seinen Platz in der Werkstatt nicht. Diese Art von Aussage — begründet, in Euro, für den Chef nachvollziehbar — ist der eigentliche Ertrag der Optimierung.

Typische Fehler beim Umstieg von Excel

- **Zellbezüge statt Namen.** Wer im Code mit `df.iloc[:, 3]` arbeitet, baut denselben Fehler nach, der in Excel-Mappen `=SUMME(D2:D40)` so gefährlich macht: Eine eingefügte Spalte verschiebt alles. Sprechen Sie Spalten **über ihren Namen** an.
- **Einheiten mischen.** Wenn eine Spalte Stunden und eine Minuten enthält, merkt das weder Excel noch der Solver — nur der Disponent, wenn der Plan nicht aufgeht.
- **Das Ergebnis nur auf den Bildschirm schreiben.** Was nicht in einer Datei landet, die die Fachabteilung öffnen kann, wird nicht benutzt.

1.9 Übungsaufgaben

Lösungen zu allen Aufgaben: Abschnitt A.1.

Aufgabe 1.1 — Abgrenzung. Ordnen Sie jede Frage der richtigen Analytik-Stufe (deskriptiv / prädiktiv / präskriptiv) zu und begründen Sie in einem Satz: (a) „Wie viele Pakete haben wir letzten Monat zugestellt?“ (b) „Wie viele Pakete werden wir nächsten Dezember zustellen?“ (c) „Welche Tour soll Fahrzeug 3 morgen fahren?“ (d) „Welche Kunden werden voraussichtlich kündigen?“ (e) „Welchen dieser Kunden sollen wir mit unserem begrenzten Rabattbudget halten?“

Aufgabe 1.2 — Hart oder weich? Entscheiden Sie für jede Bedingung eines Klinik-Dienstplans, ob sie hart oder weich modelliert werden sollte, und begründen Sie: (a) Gesetzliche Ruhezeit von 11 Stunden

zwischen zwei Diensten. (b) Frau Meier möchte freitags nicht arbeiten. (c) In jeder Nachtschicht muss mindestens eine examinierte Fachkraft anwesend sein. (d) Die Dienste sollen gleichmäßig über das Team verteilt sein. (e) Niemand arbeitet mehr als 10 Tage am Stück.

Aufgabe 1.3 □□ — Kombinatorik selbst rechnen. Ein Speditionsdisponent muss 12 Aufträge auf einen einzigen Lkw in eine Reihenfolge bringen. (a) Wie viele Reihenfolgen gibt es? (b) Der Rechner prüft 5 Millionen Reihenfolgen pro Sekunde. Wie lange dauert das vollständige Durchprobieren? (c) Wie lange dauert es bei 13 Aufträgen? Um welchen Faktor ist das mehr? (d) Wie viele Aufträge könnte man in einer Stunde noch vollständig durchprobieren?

Aufgabe 1.4 □□ — Modell lesen. Gegeben sei das Modell

$$\max\ 3x_1 + 5x_2 \quad \text{u. d. N.} \quad x_1 \le 4,\ \ 2x_2 \le 12,\ \ 3x_1 + 2x_2 \le 18,\ \ x_1, x_2 \ge 0.$$

(a) Benennen Sie die vier Bausteine. (b) Ist $(2, 6)$ zulässig? Ist $(4, 3)$ zulässig? Berechnen Sie jeweils Z. (c) Finden Sie durch Probieren die beste ganzzahlige Lösung.

Aufgabe 1.5 □□□ — Die Bäckerei programmieren. Setzen Sie das Bäckerei-Problem der Handrechnung *Zerlegen Sie dieses Problem* mit CP-SAT um. Geben Sie die optimale Produktionsmenge, den Deckungsbeitrag und die Auslastung von Mehl und Ofen aus. Prüfen Sie Ihr Ergebnis mit `assert`-Anweisungen.

Aufgabe 1.6 □□□ — Sensitivität durch Ausprobieren. Erweitern Sie `Bot_Allokation.py` so, dass es das Modell in einer Schleife für RAM-Kapazitäten von 54 bis 72 GB (in Schritten von 2) löst und eine Tabelle `RAM | x_A | x_B | Gewinn | Gewinnzuwachs pro zusätzlichem GB` ausgibt. (a) Ab welcher RAM-Menge steigt der Gewinn nicht mehr? Warum? (b) Was sagt Ihnen der „Gewinnzuwachs pro GB“ intuitiv? (Der Begriff dafür — *Schattenpreis* — folgt in Kapitel 5.)

Aufgabe 1.7 □□□ — Eigenes Problem zerlegen. Wählen Sie ein Entscheidungsproblem aus Ihrem eigenen Alltag oder Beruf (Beispiele: Wochenplan für Sportkurse, Sitzordnung bei einer Feier, Aufteilung eines Budgets auf Projekte, Reihenfolge von Hausarbeiten). Füllen Sie die Vorlage aus Abschnitt 1.6 vollständig aus. Notieren Sie außerdem: Woher kämen die Daten? Wer müsste das Ergebnis akzeptieren? — Heben Sie diese Notiz auf; sie ist der Ausgangspunkt für Ihr eigenes Projekt in der **Projektwerkstatt**.

1.10 Finde den Denkfehler

Ein Programm, das abstürzt, ist harmlos — Sie sehen sofort, dass etwas nicht stimmt. Gefährlich sind Programme, die fehlerfrei durchlaufen, einen Status `OPTIMAL` melden und eine Zahl liefern, die betriebswirtschaftlich Unsinn ist. Genau solche Fälle üben wir hier.

□ Finde den Denkfehler: Die Schreinerei verdoppelt ihren Gewinn

Eine Kollegin hat das Schreinerei-Modell aus dem Kapitelanfang leicht umgebaut — sie fand die Schleife übersichtlicher als zwei einzelne `Add`-Aufrufe. Das Programm läuft, der Solver meldet `OPTIMAL`, und der Deckungsbeitrag ist plötzlich **18 600 €** statt 10 800 €. Der Chef ist begeistert. Sollte er das sein?

```python
from ortools.linear_solver import pywraplp

produkt = {"Tisch": (240, 3.0, 6.0), "Stuhl": (60, 1.0, 1.0)}
vorrat = {"Montagestunden": 150, "Plattenmaterial": 240}

s = pywraplp.Solver.CreateSolver("GLOP")
x = {p: s.NumVar(0, s.infinity(), p) for p in produkt}
for p in produkt:
    s.Add(x[p] * produkt[p][1] <= vorrat["Montagestunden"])
    s.Add(x[p] * produkt[p][2] <= vorrat["Plattenmaterial"])
s.Maximize(sum(x[p] * produkt[p][0] for p in produkt))
s.Solve()

for p in produkt:
    print(f"{p:6}: {x[p].solution_value():5.0f} Stueck")
print(f"Deckungsbeitrag: {s.Objective().Value():.0f} EUR")
```

Ausgabe:

```
Tisch :    40 Stueck
Stuhl :   150 Stueck
Deckungsbeitrag: 18600 EUR
```

Ihre Aufgabe: (a) Rechnen Sie nach, wie viele Montagestunden dieser Plan tatsächlich braucht. (b) Benennen Sie den Modellierungsfehler in einem Satz. (c) Korrigieren Sie ihn. (d) Formulieren Sie eine `assert`-Anweisung, die diesen Fehler künftig automatisch fängt.

Auflösung: Abschnitt A.1.

 Merksatz Ein Optimierungsergebnis, das deutlich besser ausfällt als erwartet, ist zuerst einmal ein **Fehlerverdacht** — kein Grund zur Freude. Freuen Sie sich erst, nachdem Sie nachgerechnet haben, dass die Lösung alle Regeln einhält.

1.11 Micro-Quiz

 Micro-Quiz 1: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Ein Vertriebsleiter fragt: „Welche fünf unserer 80 Kunden sollen wir mit dem Restbudget besuchen?“ Welcher Analytik-Stufe entspricht das? (a) Deskriptiv — es geht um vorhandene Kundendaten. (b) Prädiktiv — man muss den Umsatz je Kunde vorhersagen. (c) Präskriptiv — gesucht ist eine Handlungsanweisung unter einer Budgetgrenze.

2. In einem Dienstplanmodell führt die Regel „Frau Meier möchte freitags frei haben“ als harte Nebenbedingung dazu, dass der Solver `INFEASIBLE` meldet. Was ist die fachlich richtige Reaktion? (a) Die Regel als weiche Bedingung mit Strafkosten formulieren. (b)

Einen leistungsfähigeren Solver einsetzen. (c) Die Variablengrenzen erhöhen, damit mehr Lösungen möglich werden.

3. Warum genügt es bei der Reihenfolgeplanung von 20 Aufträgen nicht, einen tausendmal schnelleren Rechner zu kaufen? (a) Weil Solver-Bibliotheken nicht parallelisiert sind. (b) Weil 20! so groß ist, dass Faktor 1000 die Rechenzeit von Jahrzehnten nur auf Wochen drückt — und schon 24 Aufträge wieder hoffnungslos sind. (c) Weil Fließkommazahlen bei großen Fakultäten ungenau werden.

1.12 Selbsttest

Antworten: Anhang A.

1. Was unterscheidet eine Entscheidungsvariable von einem Parameter?
2. Warum hilft ein 1000-mal schnellerer Rechner bei kombinatorischer Explosion praktisch nicht?
3. Nennen Sie die vier Bausteine eines OR-Modells.
4. Was bedeutet der Solver-Status `INFEASIBLE`, und was ist die häufigste Ursache?
5. Warum ist es gefährlich, eine Beschränkung nur als Variablengrenze statt als Nebenbedingung zu formulieren?

1.13 Zusammenfassung

- **Operations Research ist präskriptiv:** Es beantwortet nicht, was war oder was kommt, sondern was zu tun ist.
- **Kombinatorische Explosion ist eine Wand, keine Steigung.** Zwischen „in Minuten machbar“ und „in Jahrmillionen unmöglich“ liegen oft nur wenige zusätzliche Objekte. Solver umgehen das nicht durch Geschwindigkeit, sondern durch Struktur.
- **Jedes Modell besteht aus vier Bausteinen:** Entscheidungsvariablen, Parameter, Zielfunktion, Nebenbedingungen. Diese Zerlegung ist die eigentliche Arbeit; die Solverwahl ergibt sich danach fast von selbst.
- **Harte und weiche Bedingungen sind eine Entwurfsentscheidung** mit großen Folgen für Lösbarkeit und Akzeptanz.
- **Ihre Daten liegen schon in Excel.** `pandas.read_excel()` holt sie ab, der Solver rechnet, `to_excel()` gibt das Ergebnis im gewohnten Format zurück. Der Gewinn gegenüber einer Solver-Mappe ist nicht Rechenleistung, sondern Nachvollziehbarkeit: versionierbar, testbar, hundertmal in einer Schleife ausführbar.
- **Prüfen Sie jedes Ergebnis** — gegen eine Handrechnung, gegen `assert`-Anweisungen und gegen eine naive Vergleichsstrategie. Ein unerwartet gutes Ergebnis ist ein Fehlerverdacht, keine gute Nachricht.

Ausblick. Kapitel 2 liefert die Sprache, um Modelle mit hunderten Variablen kompakt aufzuschreiben: Vektoren, Matrizen und die geometrische Sicht auf den zulässigen Bereich. Dort lernen wir auch die Eigenschaft kennen, die darüber entscheidet, ob ein Problem verlässlich lösbar ist: **Konvexität**.

Kapitel 2: Das mathematische Fundament — Vektoren, Matrizen, Konvexität

Kapitel auf einen Blick

Worum geht es? Um die Sprache, in der Optimierungsmodelle aufgeschrieben werden, und um die eine Eigenschaft, die darüber entscheidet, ob ein Problem verlässlich lösbar ist: Konvexität.

Voraussetzungen: Kapitel 1. Lineare Algebra wird hier von Grund auf wiederholt.

Danach können Sie: Ein Modell in Matrixform aufschreiben, den zulässigen Bereich geometrisch deuten, beurteilen ob ein Problem konvex ist — und einschätzen, wie viele Stellen Ihres Ergebnisses überhaupt belastbar sind.

Zeitbedarf: ca. 5 Stunden.

Programme:
`Matrixform.py`
`Visualisierung_Loesungsraum.py`
`Konvexitaet_Demo.py`
`Skalierung_Kondition.py`

Notebook: `Notebooks_04/fundament.ipynb`

2.1 In 5 Minuten gelöst

Dieses Kapitel handelt von Geometrie. Bevor wir Vektoren und Matrizen einführen, sehen Sie die zentrale Einsicht in Aktion — ganz ohne Solver, mit acht Zeilen NumPy.

In 5 Minuten gelöst: Das Optimum sitzt immer in einer Ecke

Ein Betrieb fertigt zwei Produkte. Zwei Ressourcen begrenzen ihn:

$$\max\ 3x_1 + 5x_2 \quad \text{u. d. N.} \quad x_1 + x_2 \le 8, \quad 2x_1 + x_2 \le 12, \quad x_1, x_2 \ge 0$$

Der zulässige Bereich enthält **unendlich viele** Punkte. Trotzdem müssen wir nur eine Handvoll prüfen — nämlich die **Ecken**, an denen sich je zwei Begrenzungslinien schneiden:

```
import itertools
import numpy as np

A = np.array([[1, 1], [2, 1], [-1, 0], [0, -1]])   # letzte zwei Zeilen: -x <= 0
b = np.array([8, 12, 0, 0])
c = np.array([3, 5])

for i, j in itertools.combinations(range(4), 2):
    try:
```

```
        ecke = np.linalg.solve(A[[i, j]], b[[i, j]])      # Schnittpunkt zweier
↪  Geraden
    except np.linalg.LinAlgError:
        continue                                          # Geraden parallel: keine
         ↪  Ecke
    ecke[np.abs(ecke) < 1e-12] = 0.0                      # -0.0 aufräumen
    if np.all(A @ ecke <= b + 1e-9):                      # liegt die Ecke im
     ↪  Bereich?
        print(f"Ecke ({ecke[0]:.1f}, {ecke[1]:.1f})  ->  Z = {c @ ecke:6.2f}")
```

Ausgabe:

```
Ecke (4.0, 4.0)  ->  Z =  32.00
Ecke (0.0, 8.0)  ->  Z =  40.00
Ecke (6.0, 0.0)  ->  Z =  18.00
Ecke (0.0, 0.0)  ->  Z =   0.00
```

Das Optimum lautet $(0, 8)$ **mit** $Z = 40$ — und Sie haben es gefunden, ohne einen einzigen Solver zu starten. Vier Kandidaten statt unendlich vieler Punkte: Das ist der **Fundamentalsatz der linearen Optimierung**, und er ist der Grund, warum sich lineare Programme überhaupt zuverlässig lösen lassen.

Warum funktioniert das? Weil der zulässige Bereich ein **konvexes Polyeder** ist — ein Vielflächner ohne Diagonalen nach innen — und weil eine lineare Zielfunktion darauf ihren größten Wert immer am Rand annimmt, genauer: in einer Ecke. Beides begründet dieses Kapitel. Der Simplex-Algorithmus in Kapitel 5 tut im Kern nichts anderes als das Programm oben, nur klüger: Er probiert nicht alle Ecken durch, sondern läuft gezielt von Ecke zu Ecke bergauf.

> **Und wo ist der Haken?** Bei zwei Variablen gibt es 6 Eckenkandidaten, bei 50 Variablen und 50 Bedingungen sind es $\binom{100}{50} \approx 10^{29}$. Der Fundamentalsatz sagt uns *wo* wir suchen müssen — nicht, dass die Suche leicht wird.

2.2 Lernziele

Nach diesem Kapitel können Sie …

1. … ein Optimierungsmodell von der „Zeile-für-Zeile“-Schreibweise in die kompakte Matrixform $\max \mathbf{c}^{\top}\mathbf{x}$ u. d. N. $\mathbf{A}\mathbf{x} \le \mathbf{b}$ übersetzen — und zurück.
2. … erklären, warum der zulässige Bereich eines linearen Programms ein **Polyeder** ist.
3. … den Fundamentalsatz der linearen Optimierung anwenden, um Kandidaten für das Optimum zu finden, ohne alles durchzuprobieren.
4. … prüfen, ob eine Menge bzw. eine Funktion konvex ist, und begründen, warum das für die Lösbarkeit entscheidend ist.
5. … einen zweidimensionalen Lösungsraum mit `matplotlib` zeichnen und daraus die optimale Ecke ablesen.
6. … die **Konditionszahl** $\kappa(\mathbf{A})$ berechnen, ihren Wert deuten und ein schlecht skaliertes Modell mit Ruiz-Equilibrierung wieder rechenbar machen.

7. … erklären, warum ein Solver-Ergebnis von `0.99999998` niemals mit `int()` in eine ganze Zahl verwandelt werden darf.

2.3 Warum überhaupt Vektoren und Matrizen?

In Kapitel 1 hatten wir zwei Variablen und drei Nebenbedingungen. Das ließ sich bequem ausschreiben. Reale Modelle haben hunderte bis hunderttausende Variablen — dort wäre Ausschreiben nicht nur unpraktisch, sondern unmöglich.

Die lineare Algebra löst dieses Problem, indem sie **viele gleichartige Zahlen zu einem Objekt bündelt**. Das ist derselbe Gedanke wie eine Liste in Python: Statt `preis_1`, `preis_2`, …, `preis_1000` schreibt man `preise` und arbeitet mit dem Ganzen.

Die Bausteine

Sei n die Anzahl der Entscheidungsvariablen und m die Anzahl der Nebenbedingungen.

Entscheidungsvektor — was wir festlegen:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$$

Kosten- bzw. Ertragsvektor — was jede Einheit wert ist:

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \in \mathbb{R}^n$$

Zielfunktion als Skalarprodukt:

$$f(\mathbf{x}) = \mathbf{c}^\top \mathbf{x} = \sum_{j=1}^{n} c_j x_j$$

 Formel-Lesehilfe * $\mathbf{c}^\top$ — der Ertragsvektor, „umgekippt“ zu einer Zeile. * $\mathbf{c}^\top\mathbf{x}$ — **Skalarprodukt**: Multipliziere jedes c_j mit dem zugehörigen x_j und addiere alles. Das Ergebnis ist **eine einzige Zahl**. * $\sum_{j=1}^{n}$ — „addiere für $j = 1$ bis $j = n$“.

Ohne Formel gesagt: „Nimm von jedem Produkt die hergestellte Menge mal den Gewinn pro Stück und zähle alles zusammen.“ Im Bot-Beispiel: $\mathbf{c} = (150, 250)^\top$, $\mathbf{x} = (7, 5)^\top$, also $\mathbf{c}^\top\mathbf{x} = 150 \cdot 7 + 250 \cdot 5 = 2300$.

Technologiematrix — wer verbraucht wie viel wovon:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

 Formel-Lesehilfe Jede Zeile ist eine Ressource, jede Spalte eine Variable. Der Eintrag a_{ij} beantwortet: „Wie viel von Ressource i verbraucht eine Einheit von Variable j?“

Im Bot-Beispiel:

$$\mathbf{A} = \begin{pmatrix} 2 & 5 \\ 4 & 6 \\ 1 & 0 \end{pmatrix} \begin{matrix} \leftarrow \text{vCPU} \\ \leftarrow \text{RAM} \\ \leftarrow \text{Liquidität} \end{matrix}$$

Die erste Zeile $(2, 5)$ heißt: Ein A-Bot braucht 2 vCPUs, ein B-Bot braucht 5. Die dritte Zeile $(1, 0)$ heißt: Die Liquiditätsgrenze zählt nur A-Bots, B-Bots gar nicht.

Kapazitätsvektor (rechte Seite):

$$\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{R}^m \qquad \text{im Beispiel } \begin{pmatrix} 40 \\ 60 \\ 8 \end{pmatrix}$$

Die kanonische Standardform

Damit lässt sich **jedes** lineare Programm in drei Zeilen schreiben:

$$\min_{\mathbf{x}} \ \mathbf{c}^\top \mathbf{x} \qquad \text{u. d. N.} \qquad \mathbf{A}\mathbf{x} \leq \mathbf{b}, \qquad \mathbf{x} \geq \mathbf{0}$$

 Formel-Lesehilfe * **Ax** — **Matrix-Vektor-Produkt**: berechnet auf einen Schlag den Verbrauch *aller* Ressourcen. Zeile i des Ergebnisses ist $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n$. * $\mathbf{Ax} \leq \mathbf{b}$ — die Ungleichung gilt **komponentenweise**: jede einzelne Zeile muss ihre Kapazität einhalten. * $\mathbf{x} \geq \mathbf{0}$ — alle Variablen sind nichtnegativ.

Ohne Formel gesagt: „Minimiere die Gesamtkosten, ohne bei irgendeiner Ressource über die Kapazität zu gehen, und ohne negative Mengen zu produzieren.“

 Typische Fehler

- **Maximieren statt minimieren.** Die Standardform *minimiert*. Eine Maximierung wird durch Vorzeichenwechsel überführt: max $\mathbf{c}^\top \mathbf{x}$ ist dasselbe wie $-\min(-\mathbf{c})^\top \mathbf{x}$. Wer das vergisst, erhält systematisch die *schlechteste* statt der besten Lösung — siehe Abschnitt 5.7.
- **„≥“-Bedingungen direkt einsetzen.** Die Standardform kennt nur „≤“. Aus $3x_1 + 2x_2 \geq 12$ wird durch Multiplikation mit -1: $-3x_1 - 2x_2 \leq -12$. **Beim Multiplizieren mit**

einer negativen Zahl dreht sich das Ungleichheitszeichen um — der häufigste Vorzeichenfehler überhaupt.

- **Gleichungen vergessen.** $h(\mathbf{x}) = b$ lässt sich als zwei Ungleichungen schreiben: $h(\mathbf{x}) \leq b$ **und** $-h(\mathbf{x}) \leq -b$. Die meisten Solver nehmen Gleichungen aber direkt entgegen — man muss es nicht von Hand machen.

Handrechnung 2.1: Modell in Matrixform übersetzen

Gegeben:

$$\max\ 3x_1 + 5x_2 \quad \text{u. d. N.} \quad x_1 \leq 4, \quad 2x_2 \leq 12, \quad 3x_1 + 2x_2 \leq 18, \quad x_1, x_2 \geq 0$$

Schritt 1 — Vektoren und Matrix ablesen:

$$\mathbf{c} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}, \quad \mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ 12 \\ 18 \end{pmatrix}$$

Achten Sie auf die **Nullen**: Die erste Bedingung enthält x_2 gar nicht, also steht dort eine 0. Das Weglassen der Nullen ist der häufigste Anfängerfehler beim Aufstellen von $\mathbf{A}$.

Schritt 2 — Für die Standardform (Minimierung) negieren: $\tilde{\mathbf{c}} = (-3, -5)^\mathsf{T}$. Der optimale Zielwert der Minimierung ist dann das Negative des gesuchten Maximums.

Schritt 3 — Probe mit $\mathbf{x} = (2, 6)^\mathsf{T}$:

$$\mathbf{A}\mathbf{x} = \begin{pmatrix} 1 \cdot 2 + 0 \cdot 6 \\ 0 \cdot 2 + 2 \cdot 6 \\ 3 \cdot 2 + 2 \cdot 6 \end{pmatrix} = \begin{pmatrix} 2 \\ 12 \\ 18 \end{pmatrix} \leq \begin{pmatrix} 4 \\ 12 \\ 18 \end{pmatrix} \checkmark$$

Alle drei Zeilen halten. Zielwert: $3 \cdot 2 + 5 \cdot 6 = 36$.

Dasselbe in Python

```python
#!/usr/bin/env python3

# Matrixform.py
"""
Kapitel Fundament: Von der ausgeschriebenen Form zur Matrixform - und zurück.
Zeigt, dass beide Schreibweisen dasselbe Modell beschreiben.
"""

import numpy as np
from scipy.optimize import linprog

# --- Modell in Matrixform ------------------------------------------------
# max 3*x1 + 5*x2   u.d.N.   x1 <= 4,  2*x2 <= 12,  3*x1 + 2*x2 <= 18,  x >= 0
c = np.array([3.0, 5.0])                # Ertragsvektor (Maximierung)
```

```
A = np.array([[1.0, 0.0],               # Zeile 1: nur x1 kommt vor
              [0.0, 2.0],               # Zeile 2: nur x2 kommt vor
              [3.0, 2.0]])              # Zeile 3: beide
b = np.array([4.0, 12.0, 18.0])
namen = ["Rohstoff A", "Rohstoff B", "Maschinenzeit"]

# --- Ausgeschriebene Form maschinell erzeugen -----------------------------
def zeige_ausgeschrieben(c, A, b, namen):
    """Druckt die Matrixform als lesbares Ungleichungssystem."""
    terme = " + ".join(f"{c[j]:g}*x{j+1}" for j in range(len(c)))
    print(f"max  {terme}")
    print("u.d.N.")
    for i in range(A.shape[0]):
        summanden = " + ".join(f"{A[i, j]:g}*x{j+1}"
                               for j in range(A.shape[1]) if A[i, j] != 0)
        print(f"   {summanden:<24} <= {b[i]:>5g}   ({namen[i]})")
    print(f"   x1, ..., x{len(c)} >= 0")

zeige_ausgeschrieben(c, A, b, namen)

# --- Zulässigkeit eines Punktes prüfen ------------------------------------
def ist_zulaessig(x, A, b, toleranz=1e-9):
    """Prüft A x <= b und x >= 0 komponentenweise."""
    verbrauch = A @ x                    # Matrix-Vektor-Produkt: alle Zeilen auf einmal
    return bool(np.all(verbrauch <= b + toleranz) and np.all(x >= -toleranz))

for kandidat in [np.array([2.0, 6.0]), np.array([4.0, 3.0]), np.array([4.0, 6.0])]:
    zulaessig = ist_zulaessig(kandidat, A, b)
    zielwert = c @ kandidat
    verbrauch = A @ kandidat
    print(f"\nx = {kandidat}  ->  A x = {verbrauch}  "
          f"{'zulaessig' if zulaessig else 'UNZULAESSIG'},  Z = {zielwert:g}")

# --- Lösen: linprog minimiert, also c negieren ----------------------------
ergebnis = linprog(c=-c, A_ub=A, b_ub=b, bounds=[(0, None)] * len(c), method="highs")
print("\n" + "-" * 60)
print(f"Optimale Loesung:  x* = {np.round(ergebnis.x, 4)}")
print(f"Optimaler Wert:    Z* = {-ergebnis.fun:g}")
print("Hinweis: linprog minimiert, deshalb wurde c negiert und das")
print("         Ergebnis am Ende wieder mit -1 multipliziert.")
```

Erwartete Ausgabe:

```
max  3*x1 + 5*x2
u.d.N.
   1*x1                     <=     4   (Rohstoff A)
   2*x2                     <=    12   (Rohstoff B)
   3*x1 + 2*x2              <=    18   (Maschinenzeit)
```

```
   x1, ..., x2 >= 0

x = [2. 6.]  ->  A x = [ 2. 12. 18.]  zulaessig,  Z = 36

x = [4. 3.]  ->  A x = [ 4.  6. 18.]  zulaessig,  Z = 27

x = [4. 6.]  ->  A x = [ 4. 12. 24.]  UNZULAESSIG,  Z = 42

------------------------------------------------------------
Optimale Loesung:  x* = [2. 6.]
Optimaler Wert:    Z* = 36
Hinweis: linprog minimiert, deshalb wurde c negiert und das
         Ergebnis am Ende wieder mit -1 multipliziert.
```

Der dritte Kandidat $(4, 6)$ zeigt die Falle: Er hätte mit $Z = 42$ den höchsten Zielwert — ist aber **unzulässig**, weil die Maschinenzeit mit 24 über der Kapazität von 18 liegt. Ein hoher Zielwert allein bedeutet nichts.

2.4 Der zulässige Lösungsraum und das Polyeder

Jede lineare Ungleichung $a_{i1}x_1 + a_{i2}x_2 \leq b_i$ definiert geometrisch eine **Halbebene** (im $\mathbb{R}^2$) bzw. einen **Halbraum** (im $\mathbb{R}^n$): Die zugehörige Gleichung ist eine Gerade (bzw. Hyperebene), und die Ungleichung wählt eine der beiden Seiten aus.

Der Schnitt aller Halbräume bildet den zulässigen Bereich $\mathcal{F}$:

$$\mathcal{F} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$$

 Formel-Lesehilfe Die geschweiften Klammern beschreiben eine **Menge**. Der senkrechte Strich | heißt „für die gilt“.

Ohne Formel gesagt: „$\mathcal{F}$ ist die Menge aller Punkte, die gleichzeitig alle Kapazitätsgrenzen einhalten und nicht negativ sind.“ Anschaulich: der Bereich, in dem man überhaupt landen darf.

Geometrisch ist $\mathcal{F}$ ein **konvexes Polyeder** — ein von ebenen Flächen begrenzter Körper ohne Diese-Eindellungen. Im Zweidimensionalen ein Vieleck, im Dreidimensionalen ein Körper wie ein geschliffener Diamant.

Der Fundamentalsatz der linearen Optimierung

Satz. Besitzt ein lineares Optimierungsproblem eine optimale Lösung, dann liegt mindestens ein optimaler Punkt auf einer **Ecke (Extrempunkt)** des Polyeders $\mathcal{F}$.

Warum das gilt — die Anschauung: Die Zielfunktion $\mathbf{c}^\top\mathbf{x}$ hat überall dieselbe Steigungsrichtung; ihre Höhenlinien sind parallele Geraden. Stellen Sie sich vor, Sie schieben eine solche Gerade in Richtung wachsender Zielwerte über das Polyeder. Der letzte Punkt, den sie berührt, bevor sie das Polyeder verlässt, ist eine **Ecke** — oder, wenn die Gerade zufällig parallel zu einer Kante liegt, eine ganze Kante, deren Endpunkte wiederum Ecken sind.

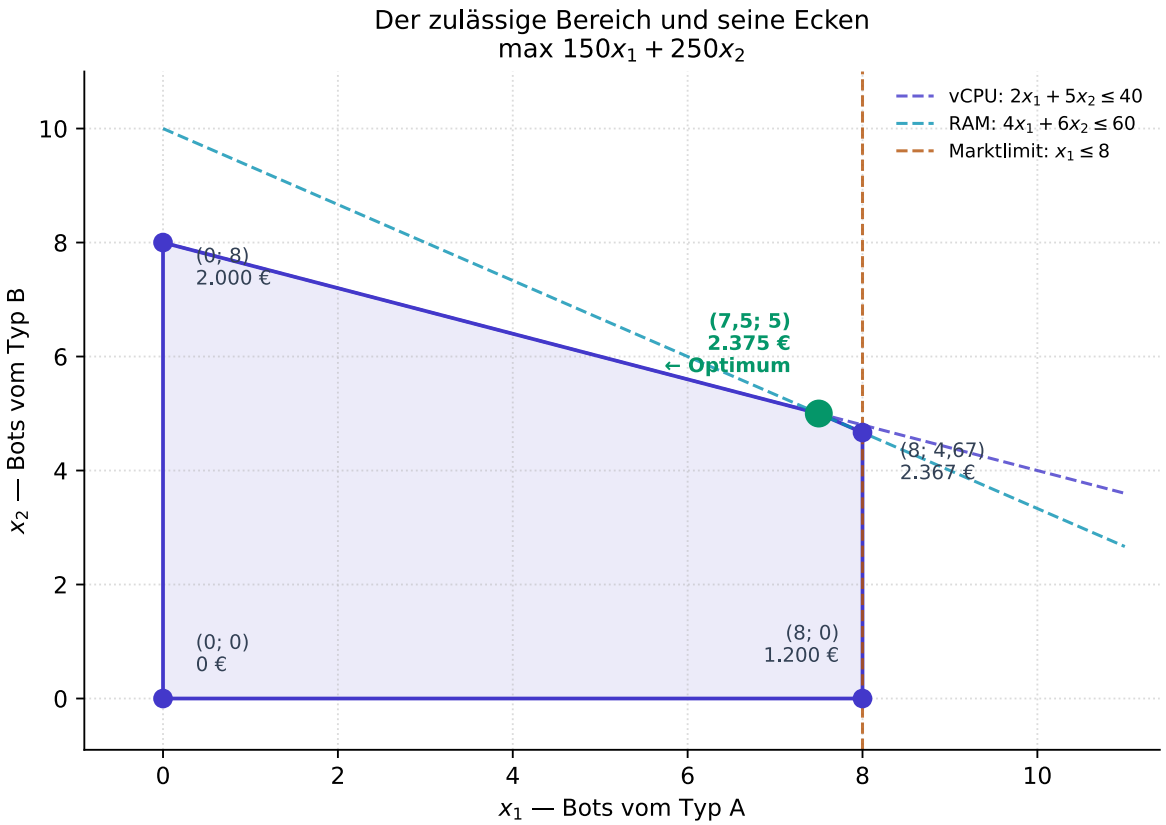

Abbildung 3: Abb. 2.1: Der zulässige Bereich des Bot-Allokationsproblems aus Kapitel 1, das dieses Kapitel weiterrechnet. Fünf Ecken, jede mit ihrem Zielwert — der Fundamentalsatz sagt, dass das Optimum unter ihnen sein muss, und hier ist es (7,5; 5) mit 2 375 €. Die Koordinaten sind nicht eingetragen, sondern als zulässige Schnittpunkte der Begrenzungsgeraden berechnet; erzeugt von `bilder_04/erzeuge_polyeder.py`.

Warum das praktisch so wichtig ist: Das Polyeder enthält unendlich viele Punkte, aber nur **endlich viele Ecken**. Der Simplex-Algorithmus (Kapitel 5) muss deshalb nicht unendlich viel absuchen, sondern wandert gezielt von Ecke zu Ecke.

Handrechnung 2.2: Optimum über Ecken finden

Wir nehmen wieder max $3x_1 + 5x_2$ mit $x_1 \le 4$, $2x_2 \le 12$, $3x_1 + 2x_2 \le 18$, $x_1, x_2 \ge 0$.

Schritt 1 — Ecken bestimmen. Eine Ecke entsteht dort, wo sich zwei Begrenzungsgeraden schneiden (und der Punkt zulässig ist). Die Geraden sind: $x_1 = 0$, $x_2 = 0$, $x_1 = 4$, $x_2 = 6$, $3x_1 + 2x_2 = 18$.

Schnittpunkt von	Ecke	zulässig?	$Z = 3x_1 + 5x_2$
$x_1 = 0,\ x_2 = 0$	$(0, 0)$	☐	0
$x_1 = 4,\ x_2 = 0$	$(4, 0)$	☐	12
$x_1 = 4,\ 3x_1 + 2x_2 = 18$	$(4, 3)$	☐	27
$x_2 = 6,\ 3x_1 + 2x_2 = 18$	$(2, 6)$	☐	**36** ← Optimum
$x_1 = 0,\ x_2 = 6$	$(0, 6)$	☐	30
$x_1 = 4,\ x_2 = 6$	$(4, 6)$	☐ $(3 \cdot 4 + 2 \cdot 6 = 24 > 18)$	—

Schritt 2 — Ergebnis ablesen: $\mathbf{x}^* = (2, 6)^\top$, $Z^* = 36$.

Beachten Sie: Wir haben **fünf** Punkte geprüft statt unendlich viele. Genau das ist der Gewinn des Fundamentalsatzes. Für $n = 2$ geht das von Hand; bei $n = 50$ gibt es zu viele Ecken — dann übernimmt der Simplex-Algorithmus, der nur die *verbessernden* Ecken besucht.

2.5 Konvexität: die Grenze zwischen leicht und schwer

Hier kommt die vielleicht wichtigste Einsicht des ganzen Buches. In der Optimierungstheorie verläuft die Trennlinie zwischen „zuverlässig lösbar“ und „im Allgemeinen hoffnungslos“ **nicht** zwischen linear und nichtlinear — sondern zwischen **konvex** und **nicht-konvex**.

Konvexe Menge

Eine Menge $\mathcal{C} \subseteq \mathbb{R}^n$ heißt **konvex**, wenn für alle Punkte $\mathbf{x}, \mathbf{y} \in \mathcal{C}$ und jedes $\theta \in [0, 1]$ gilt:

$$\theta \mathbf{x} + (1 - \theta)\mathbf{y} \in \mathcal{C}$$

Formel-Lesehilfe * θ („theta“) ist eine Zahl zwischen 0 und 1. * $\theta\mathbf{x} + (1-\theta)\mathbf{y}$ durchläuft für θ von 1 bis 0 genau die **Verbindungsstrecke** von $\mathbf{x}$ nach $\mathbf{y}$. Bei $\theta = 1$ sind wir in $\mathbf{x}$, bei $\theta = 0$ in $\mathbf{y}$, bei $\theta = 0{,}5$ genau in der Mitte.

Ohne Formel gesagt: Eine Menge ist konvex, wenn man zwischen zwei beliebigen ihrer Punkte eine gerade Linie ziehen kann, ohne die Menge zu verlassen. Ein Kreis, ein Quadrat und jedes Polyeder sind konvex. Ein Halbmond, ein Ring und ein Stern sind es nicht.

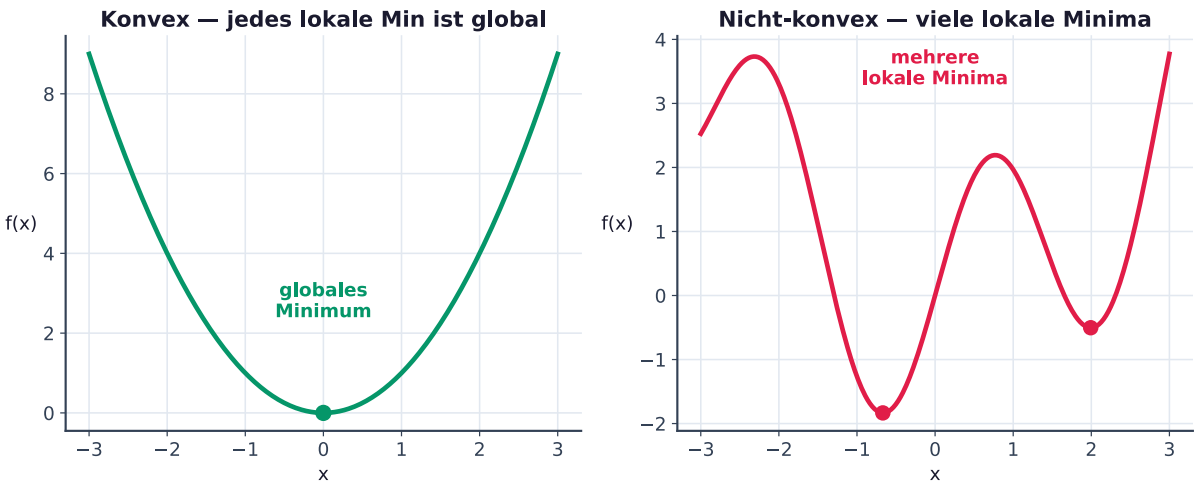

Abbildung 4: Abb. 2.2: Konvexe vs. nicht-konvexe Zielfunktion

Konvexe Funktion

Eine Funktion $f : \mathcal{C} \to \mathbb{R}$ heißt **konvex**, wenn für alle $\mathbf{x}, \mathbf{y} \in \mathcal{C}$ und $\theta \in [0, 1]$ gilt:

$$f(\theta \mathbf{x} + (1 - \theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1 - \theta) f(\mathbf{y})$$

 Formel-Lesehilfe * Links: der Funktionswert **auf** der Verbindungsstrecke. * Rechts: der entsprechende Wert **auf der Sehne** zwischen den beiden Funktionswerten.

Ohne Formel gesagt: Der Funktionsgraph liegt zwischen je zwei Punkten immer **unterhalb** der geraden Verbindungslinie. Eine konvexe Funktion ist „nach oben offen geschüsselt“ — wie eine Parabel x^2 oder eine Suppenschüssel. Kippen Sie eine Kugel hinein, rollt sie zum tiefsten Punkt, und der ist eindeutig.

Der zentrale Satz

Satz. Bei einem **konvexen Optimierungsproblem** (konvexe Zielfunktion über konvexem zulässigen Bereich) ist jedes **lokale** Minimum automatisch auch das **globale** Minimum.

Warum das den Unterschied macht: Ein Algorithmus, der nur lokal sucht — „gehe bergab, bis es nicht mehr bergab geht“ — findet bei konvexen Problemen garantiert das absolut beste Ergebnis. Bei nicht-konvexen Problemen bleibt derselbe Algorithmus im nächstbesten Tal stecken, und niemand kann ihm ansehen, ob es das tiefste war. Man müsste alle Täler prüfen — und deren Zahl wächst wieder kombinatorisch.

 Merksatz Konvex heißt: Wer bergab geht, kommt am tiefsten Punkt an. Nicht-konvex heißt: Wer bergab geht, kommt *irgendwo* an — und weiß nicht, ob es der tiefste Punkt war.

Was ist konvex, was nicht?

Problemtyp	Konvex?	Konsequenz
Lineares Programm (LP)	**ja**	Globales Optimum, Sekunden bis Minuten selbst bei 10^6 Variablen
Quadratisches Programm mit $\mathbf{P} \succeq 0$	**ja**	Globales Optimum, effizient (Kapitel 11, Kapitel 19)
Ganzzahlige Variablen (MILP)	**nein**	Der zulässige Bereich ist ein Punktgitter, keine zusammenhängende Menge — NP-schwer (Kapitel 6)
Quadratisches Programm mit indefinitem **P**	**nein**	Nur lokale Optima; eine ungültige Kovarianzmatrix ist ein typischer Auslöser (siehe Abschnitt 11.5)
Allgemeines NLP mit beliebigen Funktionen	meist **nein**	`scipy.optimize` liefert nur ein lokales Optimum, abhängig vom Startpunkt

Konvexität sichtbar machen

```
#!/usr/bin/env python3

# Konvexitaet_Demo.py
"""
Kapitel Fundament: Konvexität praktisch erfahrbar machen.
Teil 1: Die Sehnen-Bedingung numerisch nachprüfen.
Teil 2: Zeigen, warum lokale Suche bei nicht-konvexen Funktionen scheitert.
"""

import numpy as np
from scipy.optimize import minimize

# --- Teil 1: Sehnen-Test ------------------------------------------------
def ist_konvex_numerisch(f, unten, oben, proben=2000, seed=0):
    """
    Prüft die Konvexitätsdefinition an zufälligen Punktepaaren:
       f(theta*x + (1-theta)*y)  <=  theta*f(x) + (1-theta)*f(y)  ?
    Findet Gegenbeispiele - beweist aber KEINE Konvexität.
    """
    rng = np.random.default_rng(seed)
    schlimmste_verletzung = 0.0
    for _ in range(proben):
        x, y = rng.uniform(unten, oben, size=2)
```

```
        theta = rng.uniform(0.0, 1.0)
        links = f(theta * x + (1 - theta) * y)
        rechts = theta * f(x) + (1 - theta) * f(y)
        schlimmste_verletzung = max(schlimmste_verletzung, links - rechts)
    return schlimmste_verletzung


funktionen = {
    "f(x) = x^2            (konvex)":            lambda x: x ** 2,
    "f(x) = |x|            (konvex)":            lambda x: abs(x),
    "f(x) = e^x            (konvex)":            lambda x: np.exp(x),
    "f(x) = x^3            (NICHT konvex)":      lambda x: x ** 3,
    "f(x) = x^2+3sin(3x)   (NICHT konvex)":      lambda x: x ** 2 + 3 * np.sin(3 * x),
}

print("=" * 68)
print("  TEIL 1: SEHNEN-TEST  (positive Zahl = Konvexität verletzt)")
print("=" * 68)
for name, f in funktionen.items():
    verletzung = ist_konvex_numerisch(f, -3.0, 3.0)
    urteil = "konvex (keine Verletzung gefunden)" if verletzung < 1e-9 \
             else f"NICHT konvex (Verletzung bis {verletzung:.3f})"
    print(f"{name:<34} -> {urteil}")

# --- Teil 2: Lokale Suche von verschiedenen Startpunkten ------------------
def wellige_funktion(x):
    """Nicht-konvex: eine Parabel mit aufmodulierter Welle -> viele lokale Minima."""
    return x[0] ** 2 + 3.0 * np.sin(3.0 * x[0])

print("\n" + "=" * 68)
print("  TEIL 2: LOKALE SUCHE BEI NICHT-KONVEXER FUNKTION")
print("=" * 68)
print(f"{'Startpunkt':>12} | {'gefundenes Minimum':>20} | {'Funktionswert':>14}")
print("-" * 68)

ergebnisse = []
for start in [-3.0, -1.5, 0.0, 1.5, 3.0]:
    res = minimize(wellige_funktion, x0=[start], method="BFGS")
    ergebnisse.append((start, res.x[0], res.fun))
    print(f"{start:>12.1f} | {res.x[0]:>20.4f} | {res.fun:>14.4f}")

bester = min(ergebnisse, key=lambda t: t[2])
print("-" * 68)
print(f"Je nach Startpunkt landet derselbe Algorithmus in "
      f"{len({round(e[1], 3) for e in ergebnisse})} verschiedenen Minima.")
print(f"Das beste gefundene: x = {bester[1]:.4f} mit f = {bester[2]:.4f} "
      f"(Start bei {bester[0]:.1f})")
print("Bei einer KONVEXEN Funktion waeren alle Zeilen identisch --")
```

```
print("der Startpunkt waere voellig gleichgueltig.")
print("=" * 68)
```

Erwartete Ausgabe:

```
====================================================================
  TEIL 1: SEHNEN-TEST  (positive Zahl = Konvexität verletzt)
====================================================================
f(x) = x^2            (konvex)      -> konvex (keine Verletzung gefunden)
f(x) = |x|            (konvex)      -> konvex (keine Verletzung gefunden)
f(x) = e^x            (konvex)      -> konvex (keine Verletzung gefunden)
f(x) = x^3            (NICHT konvex) -> NICHT konvex (Verletzung bis 12.933)
f(x) = x^2+3sin(3x)   (NICHT konvex) -> NICHT konvex (Verletzung bis 4.800)

====================================================================
  TEIL 2: LOKALE SUCHE BEI NICHT-KONVEXER FUNKTION
====================================================================
  Startpunkt |   gefundenes Minimum |  Funktionswert
--------------------------------------------------------------------
        -3.0 |              -2.4280 |          3.3695
        -1.5 |              -0.4874 |         -2.7448
         0.0 |              -0.4874 |         -2.7448
         1.5 |               1.4606 |         -0.7042
         3.0 |               3.3817 |          9.4570
--------------------------------------------------------------------
Je nach Startpunkt landet derselbe Algorithmus in 4 verschiedenen Minima.
Das beste gefundene: x = -0.4874 mit f = -2.7448 (Start bei -1.5)
Bei einer KONVEXEN Funktion waeren alle Zeilen identisch --
der Startpunkt waere voellig gleichgueltig.
====================================================================
```

□ Code-Durchgang

Stelle	Was passiert	Warum wichtig
`ist_konvex_numerisch`	testet die Definition an Zufallspaaren	Ein einziges Gegenbeispiel **widerlegt** Konvexität; kein noch so großer Test **beweist** sie
`links - rechts`	Abstand zur Sehne	Positiv = Graph liegt über der Sehne = Verletzung
Teil 2, Schleife über Startpunkte	dieselbe Funktion, fünf Starts	Zeigt die praktische Konsequenz: das Ergebnis hängt vom Zufall des Startpunkts ab
`method="BFGS"`	klassisches lokales Gradientenverfahren	Genau der Verfahrenstyp, den `scipy.optimize` in Kapitel 11 verwendet

> **Die Lektion in einer Zeile:** Wenn Ihr Optimierungsergebnis vom Startwert abhängt, ist Ihr Problem nicht konvex — und Sie haben keine Garantie, das Beste gefunden zu haben.

Achten Sie besonders auf die erste Zeile der Tabelle: Vom Startpunkt $-3{,}0$ aus landet das Verfahren bei $f = 3{,}37$ — einem Wert, der um mehr als **6 Einheiten schlechter** ist als das beste gefundene Minimum. Der Algorithmus meldet dabei keinen Fehler und keine Warnung. Er hat korrekt gearbeitet, ein lokales Minimum gefunden und ist stehen geblieben. Genau das ist die Gefahr: **Nicht-Konvexität erzeugt keine Fehlermeldungen, sondern stille Fehlentscheidungen.**

> **Typische Fehler**
>
> - **Konvexität mit Linearität verwechseln.** $f(x) = x^2$ ist nichtlinear, aber konvex und damit bequem lösbar. $f(x) = x^3$ ist ebenfalls nichtlinear, aber nicht konvex und deshalb problematisch. Nichtlinear ist kein Urteil, konvex schon.
> - **Konvexität nur der Zielfunktion prüfen.** Auch der zulässige Bereich muss konvex sein. Ganzzahligkeitsbedingungen zerstören genau diese Eigenschaft — deshalb ist MILP schwer, obwohl alles daran linear ist.
> - **Eine Matrix ungeprüft als Kovarianzmatrix verwenden.** Nur wenn alle Eigenwerte ≥ 0 sind, ist $\mathbf{x}^\top \mathbf{P}\mathbf{x}$ konvex. Ein einziger negativer Eigenwert kippt das ganze Problem — mehr dazu in Abschnitt 11.5.

2.6 Geometrische Visualisierung des Lösungsraums

Wir zeichnen nun den Lösungsraum aus Kapitel 1 und lesen das Optimum ab. Zweidimensionale Bilder sind der schnellste Weg, Intuition für höhere Dimensionen aufzubauen — auch wenn man sie dort nicht mehr zeichnen kann.

```
#!/usr/bin/env python3

# Visualisierung_Loesungsraum.py
"""
Kapitel Fundament: Geometrische Visualisierung eines 2D-Optimierungsraums.

Ecken werden berechnet, auf Zulässigkeit geprüft, bewertet und eingezeichnet.
Zusätzlich wird das ganzzahlige Optimum systematisch bestimmt statt behauptet.
"""

import itertools
import os

import numpy as np
import matplotlib
matplotlib.use("Agg")            # kein Bildschirm nötig
import matplotlib.pyplot as plt

OUTPUT_DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)), "output")
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
# --- Modell (identisch zum Kapitel Einfuehrung) ---------------------------
#   max 150*xA + 250*xB
#   u.d.N.  2*xA + 5*xB <= 40   (vCPU)
#           4*xA + 6*xB <= 60   (RAM)
#           1*xA + 0*xB <=  8   (Marktliquidität)
c = np.array([150.0, 250.0])
A = np.array([[2.0, 5.0], [4.0, 6.0], [1.0, 0.0]])
b = np.array([40.0, 60.0, 8.0])
restriktionsnamen = ["vCPU", "RAM", "Marktlimit"]


def ist_zulaessig(punkt, tol=1e-7):
    return np.all(A @ punkt <= b + tol) and np.all(punkt >= -tol)


def berechne_ecken():
    """
    Ecken = Schnittpunkte je zweier Begrenzungsgeraden, die zulässig sind.
    Begrenzungen sind die 3 Restriktionen plus die beiden Achsen xA=0, xB=0.
    """
    geraden = [(A[i], b[i]) for i in range(len(b))]
    geraden.append((np.array([1.0, 0.0]), 0.0))     # xA = 0
    geraden.append((np.array([0.0, 1.0]), 0.0))     # xB = 0

    ecken = []
    for (n1, d1), (n2, d2) in itertools.combinations(geraden, 2):
        M = np.array([n1, n2])
        if abs(np.linalg.det(M)) < 1e-9:            # parallel -> kein Schnittpunkt
            continue
        p = np.linalg.solve(M, np.array([d1, d2]))
        if ist_zulaessig(p) and not any(np.allclose(p, e) for e in ecken):
            ecken.append(p)
    return np.array(ecken)


def bestes_ganzzahliges():
    """Vollständige Suche über das kleine Gitter - hier zulässig, weil winzig."""
    bester_wert, bester_punkt = -np.inf, None
    for xa in range(0, 21):
        for xb in range(0, 21):
            p = np.array([float(xa), float(xb)])
            if ist_zulaessig(p) and c @ p > bester_wert:
                bester_wert, bester_punkt = c @ p, p
    return bester_punkt, bester_wert


# --- Analyse ---------------------------------------------------------------
```

```
ecken = berechne_ecken()
werte = ecken @ c
reihenfolge = np.argsort(-werte)

print("=" * 62)
print("  ECKEN DES ZULÄSSIGEN POLYEDERS (nach Zielwert sortiert)")
print("=" * 62)
print(f"{'x_A':>8} {'x_B':>8} {'Z = 150 xA + 250 xB':>24}")
print("-" * 62)
for i in reihenfolge:
    print(f"{ecken[i, 0]:>8.2f} {ecken[i, 1]:>8.2f} {werte[i]:>24,.2f} EUR")

lp_punkt, lp_wert = ecken[reihenfolge[0]], werte[reihenfolge[0]]
ip_punkt, ip_wert = bestes_ganzzahliges()

print("-" * 62)
print(f"Kontinuierliches Optimum (LP): x = ({lp_punkt[0]:.2f}, {lp_punkt[1]:.2f}), "
      f"Z = {lp_wert:,.2f} EUR")
print(f"Ganzzahliges Optimum (IP):     x = ({ip_punkt[0]:.0f}, {ip_punkt[1]:.0f}), "
      f"Z = {ip_wert:,.2f} EUR")
print(f"Preis der Ganzzahligkeit:      {lp_wert - ip_wert:,.2f} EUR "
      f"({(1 - ip_wert / lp_wert) * 100:.2f} %)")
print("=" * 62)

# --- Zeichnung ---------------------------------------------------------
gitter = np.linspace(0, 15, 400)
xa_gitter, xb_gitter = np.meshgrid(gitter, gitter)

plt.figure(figsize=(10, 8))

# Restriktionsgeraden
plt.plot(gitter, (40 - 2 * gitter) / 5, color="tab:blue", lw=2,
         label=r"$2x_A + 5x_B \leq 40$ (vCPU)")
plt.plot(gitter, (60 - 4 * gitter) / 6, color="tab:green", lw=2,
         label=r"$4x_A + 6x_B \leq 60$ (RAM)")
plt.axvline(x=8, color="tab:orange", lw=2, label=r"$x_A \leq 8$ (Marktlimit)")

# Zulässiger Bereich
maske = ((2 * xa_gitter + 5 * xb_gitter <= 40) & (4 * xa_gitter + 6 * xb_gitter <= 60)
         & (xa_gitter <= 8) & (xa_gitter >= 0) & (xb_gitter >= 0))
plt.imshow(maske.astype(int), extent=(0, 15, 0, 15), origin="lower",
           cmap="Greys", alpha=0.25, aspect="auto")

# Höhenlinien der Zielfunktion
Z = 150 * xa_gitter + 250 * xb_gitter
hoehen = plt.contour(xa_gitter, xb_gitter, Z, levels=[500, 1000, 1500, 2000, 2375],
                     colors="purple", linestyles="--", alpha=0.7)
plt.clabel(hoehen, inline=True, fontsize=9, fmt="Z = %1.0f EUR")
```

```
# Ecken einzeichnen - jetzt werden sie tatsächlich benutzt
plt.scatter(ecken[:, 0], ecken[:, 1], s=70, facecolors="white",
            edgecolors="black", zorder=4, label="Ecken des Polyeders")
for e, w in zip(ecken, werte):
    plt.annotate(f"({e[0]:.1f}, {e[1]:.1f})\nZ={w:,.0f}", (e[0], e[1]),
                 textcoords="offset points", xytext=(6, 6), fontsize=8)

plt.scatter([lp_punkt[0]], [lp_punkt[1]], color="purple", marker="D", s=110, zorder=5,
            label=f"LP-Optimum ({lp_punkt[0]:.1f}, {lp_punkt[1]:.1f})")
plt.scatter([ip_punkt[0]], [ip_punkt[1]], color="red", s=170, zorder=6,
            label=f"Ganzzahliges Optimum ({ip_punkt[0]:.0f}, {ip_punkt[1]:.0f})")

plt.xlim(0, 12)
plt.ylim(0, 10)
plt.xlabel("Anzahl Arbitrage-Bots ($x_A$)", fontsize=11)
plt.ylabel("Anzahl Trendfolge-Bots ($x_B$)", fontsize=11)
plt.title("Polyeder des zulässigen Bereichs mit Niveaulinien der Zielfunktion", fontsize=13)
plt.grid(True, linestyle=":", alpha=0.6)
plt.legend(loc="upper right", framealpha=0.9)
plt.tight_layout()
ziel = os.path.join(OUTPUT_DIR, "feasible_region_2d.png")
plt.savefig(ziel, dpi=150)
print(f"Visualisierung gespeichert unter '{ziel}'")
```

Erwartete Ausgabe:

```
============================================================
  ECKEN DES ZULÄSSIGEN POLYEDERS (nach Zielwert sortiert)
============================================================
     x_A      x_B      Z = 150 xA + 250 xB
------------------------------------------------------------
    7.50     5.00                2,375.00 EUR
    8.00     4.67                2,366.67 EUR
    0.00     8.00                2,000.00 EUR
    8.00     0.00                1,200.00 EUR
    0.00     0.00                    0.00 EUR
------------------------------------------------------------
Kontinuierliches Optimum (LP): x = (7.50, 5.00), Z = 2,375.00 EUR
Ganzzahliges Optimum (IP):     x = (7, 5), Z = 2,300.00 EUR
Preis der Ganzzahligkeit:      75.00 EUR (3.16 %)
============================================================
```

Zwei Beobachtungen, die Kapitel 6 vorbereiten:

1. Das LP-Optimum liegt bei (7,5; 5,0) — **kein zulässiger Betriebszustand**, denn ein halber Bot existiert nicht.

2. Das ganzzahlige Optimum $(7, 5)$ ist **nicht** durch Runden entstanden: Aufrunden auf $(8, 5)$ wäre unzulässig ($2 \cdot 8 + 5 \cdot 5 = 41 > 40$). Hier ging Abrunden gut — Kapitel 6 zeigt Fälle, in denen Runden dramatisch scheitert.

2.7 Wenn die Zahlen nicht zusammenpassen: Kondition und Skalierung

Bis hierher war die lineare Algebra exakt. Ein Rechner rechnet aber nicht exakt, sondern mit rund 16 signifikanten Stellen — und Ihre Betriebsdaten sind ohnehin nur auf wenige Stellen genau bekannt. Dieser Abschnitt beantwortet die Frage, die genau daraus entsteht:

Wenn meine Eingabedaten leicht ungenau sind — wie ungenau ist dann meine Lösung?

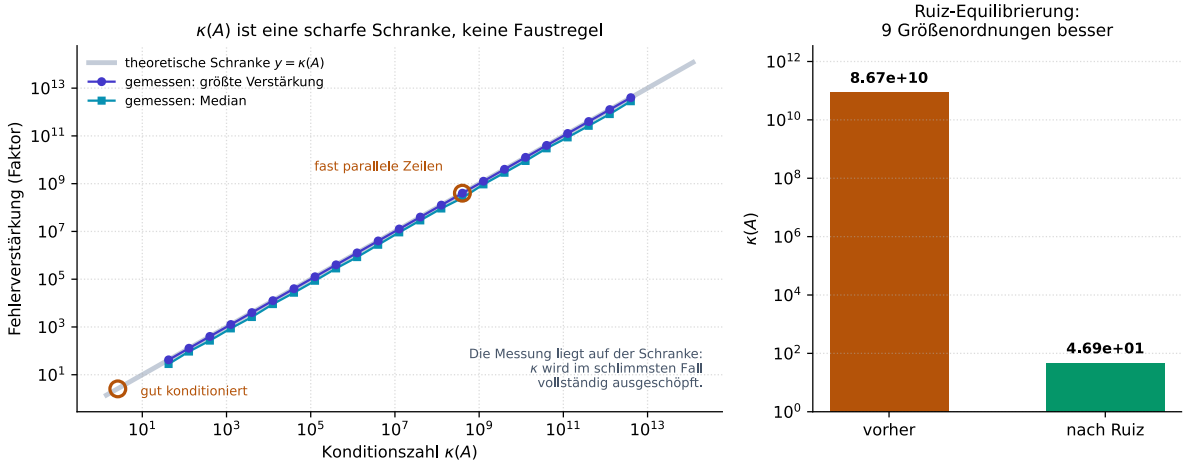

Abbildung 5: Abb. 2.3: Was $\kappa(\mathbf{A})$ praktisch bedeutet. Links: Über zwölf Größenordnungen hinweg liegt die gemessene größte Fehlerverstärkung genau auf der theoretischen Schranke — κ ist keine grobe Faustregel. Rechts: Was die Ruiz-Equilibrierung an einem Modell mit unverträglichen Einheiten ausrichtet. Erzeugt von `bilder_04/erzeuge_kondition.py`, gerechnet mit derselben Instanz wie `Skalierung_Kondition.py`.

Die Antwort trägt einen Namen: die **Konditionszahl** $\kappa(\mathbf{A})$. Sie ist der wichtigste Begriff dieses Kapitels für den Produktivbetrieb, weil er erklärt, warum ein mathematisch korrektes Modell trotzdem unbrauchbare Ergebnisse liefern kann — oder sich mit einem sachlich falschen `INFEASIBLE` verabschiedet.

Die geometrische Vorstellung: zwei fast parallele Linien

Zwei Nebenbedingungen legen im Zweidimensionalen einen Schnittpunkt fest. Stehen die zugehörigen Geraden **kreuzweise** aufeinander, ist der Schnittpunkt robust: Verschiebt man eine Gerade um einen Millimeter, wandert der Schnittpunkt um einen Millimeter.

Verlaufen die Geraden dagegen **fast parallel**, entsteht ein spitzer, langgezogener Keil. Derselbe Millimeter Verschiebung schickt den Schnittpunkt jetzt meterweit den Keil entlang. Genau das misst $\kappa(\mathbf{A})$: **den Verstärkungsfaktor zwischen Datenfehler und Lösungsfehler.**

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A}) \cdot \frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|}, \qquad \kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\| = \frac{\sigma_{\max}}{\sigma_{\min}}$$

Formel-Übersetzer

Mathematik	Alltagssprache
$\|\Delta\mathbf{b}\|/\|\mathbf{b}\|$	„Um wie viel Prozent sind meine Eingabedaten daneben?“
$\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$	„Um wie viel Prozent ist die Lösung daneben?“
$\kappa(\mathbf{A})$	Der schlimmstmögliche Verstärkungsfaktor zwischen beidem.
$\sigma_{\max}/\sigma_{\min}$	Verhältnis der größten zur kleinsten Streckung, die die Matrix ausübt — „wie schief ist der Keil?“
$\kappa = 1$	Bestfall: kein Fehler wird verstärkt.
$\kappa = 10^k$	Faustregel: Sie verlieren k signifikante Stellen. Bei $\kappa = 10^8$ bleiben von 16 Stellen noch 8.

In einem Satz: *κ sagt Ihnen, wie viele Nachkommastellen Ihres Ergebnisses Sie noch glauben dürfen.*

Merksatz $\kappa < 10^3$ ist unbedenklich, $\kappa > 10^6$ verdient eine Prüfung, $\kappa > 10^{10}$ heißt: Das Ergebnis ist Rauschen mit Nachkommastellen. Rechnen Sie κ mit `np.linalg.cond(A)` aus — es kostet eine Zeile und beantwortet die Frage, die sonst monatelang niemand stellt.

Woher schlechte Kondition im Alltag kommt

Fast nie aus exotischer Mathematik — fast immer aus **Einheiten**:

Ursache	Beispiel aus der Praxis	Abhilfe
Gemischte Größenordnungen	Kapitalbindung in Euro (10^7) neben Ausschussquote als Anteil (10^{-2}) in derselben Matrix	In Tsd. Euro und Prozent rechnen — oder automatisch skalieren
Fast redundante Regeln	„Mindestens 30 % Anteil A“ und „höchstens 70 % Anteil B“ bei nur zwei Sorten	Doppelung erkennen und eine Regel streichen
Big-M zu groß gewählt	$M = 10^9$, wo $M = 500$ genügt hätte (Kapitel 6)	Kleinstmögliches M aus den Daten herleiten
Mengen in Stück *und* Tonnen	Schüttgut und Einzelteile im selben Modell	Eine Einheit je Größe, konsequent

Die dritte Zeile ist der häufigste selbstgemachte Fall: Ein unnötig großes Big-M bläht κ auf und lässt Solver bei völlig harmlosen Modellen stundenlang suchen oder falsche Ergebnisse melden.

Ruiz-Equilibrierung: das Modell gesundrechnen

Die Gegenmaßnahme ist erstaunlich schlicht. Man multipliziert jede **Zeile** und jede **Spalte** der Matrix mit einem Faktor, sodass die Beträge überall in derselben Größenordnung landen:

$$\tilde{\mathbf{A}} = \mathbf{D}_r \, \mathbf{A} \, \mathbf{D}_c$$

mit Diagonalmatrizen $\mathbf{D}_r$ (Zeilen) und $\mathbf{D}_c$ (Spalten). Das **Ruiz-Verfahren** bestimmt diese Faktoren iterativ: In jedem Durchlauf wird jede Zeile durch die Wurzel ihres größten Betrags geteilt, danach jede Spalte. Nach wenigen Durchläufen liegen alle Zeilen- und Spaltenmaxima bei 1.

Wirtschaftlich passiert dabei nichts: Eine Zeilenskalierung heißt „diese Nebenbedingung in einer anderen Einheit messen“, eine Spaltenskalierung „diese Variable in einer anderen Einheit messen“. Die Lösung rechnet man mit $\mathbf{x} = \mathbf{D}_c \, \tilde{\mathbf{x}}$ zurück.

> **Merksatz** Jeder ernsthafte Solver skaliert intern selbst — HiGHS, Gurobi, CP-SAT alle. Verlassen Sie sich trotzdem nicht darauf: Die interne Skalierung repariert die Rechnung, nicht die Modellierung. Wenn Ihre Koeffizienten zwölf Größenordnungen überspannen, sagt Ihnen das etwas über Ihr Modell, nicht über den Solver.

Das Experiment

Das folgende Programm führt beides vor — die Fehlerverstärkung und ihre Behebung — und schließt mit der praktischen Konsequenz, die Sie in Kapitel 6 wieder brauchen werden: warum eine Binärvariable mit dem Wert `0.99999998` niemals mit `int()` gerundet werden darf.

```
#!/usr/bin/env python3

# Skalierung_Kondition.py
"""
Kapitel Fundament: Was die Konditionszahl kappa(A) praktisch bedeutet - und wie man
ein schlecht skaliertes Modell wieder gesund rechnet.

Drei Experimente:

  1. Fehlerverstaerkung: Wie stark schlaegt eine winzige Datenunsicherheit auf
     die Loesung durch? kappa(A) ist genau die Obergrenze dieses Faktors.
  2. Ruiz-Equilibrierung: Ein Modell, in dem Euro-Betraege (1e7) und
     Tonnen-Angaben (1e-3) in derselben Matrix stehen, wird durch Zeilen- und
     Spaltenskalierung um Groessenordnungen besser konditioniert.
  3. Toleranzen: Warum eine Binaervariable mit dem Wert 0.99999998 niemals
     mit int() gerundet werden darf.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
```

```
from scipy.optimize import linprog

RNG = np.random.default_rng(42)


# --- Experiment 1: kappa(A) als Fehlerverstaerker ---------------------------

def fehlerverstaerkung(A: np.ndarray, versuche: int = 200,
                       stoerung: float = 1e-10) -> tuple[float, float]:
    """Stoert die rechte Seite b relativ um 'stoerung' in zufaellige Richtungen
    und misst, um welchen Faktor sich der Fehler in der Loesung x vergroessert.

    Liefert (Median, Maximum) der Verstaerkung. Die Theorie sagt: Das Maximum
    kann bis kappa(A) betragen - und nur bis dahin.
    """
    x_wahr = np.ones(A.shape[1])
    b = A @ x_wahr
    faktoren = []
    for _ in range(versuche):
        richtung = RNG.normal(size=b.size)
        richtung /= np.linalg.norm(richtung)
        b_gestoert = b + stoerung * np.linalg.norm(b) * richtung
        x_gestoert = np.linalg.solve(A, b_gestoert)
        rel_x = np.linalg.norm(x_gestoert - x_wahr) / np.linalg.norm(x_wahr)
        faktoren.append(rel_x / stoerung)
    return float(np.median(faktoren)), float(np.max(faktoren))


def zeige_experiment_1() -> None:
    print("=" * 74)
    print("  1. KONDITIONSZAHL ALS FEHLERVERSTAERKER")
    print("=" * 74)
    print("Zwei Gleichungssysteme, beide exakt loesbar mit x = (1, 1).")
    print("Die rechte Seite wird um relativ 1e-10 gestoert - so viel Unsicherheit")
    print("steckt in JEDER gemessenen Betriebszahl allemal.\n")

    modelle = {
        "gut konditioniert":     np.array([[2.0, 1.0], [1.0, 3.0]]),
        "fast parallele Zeilen": np.array([[1.0, 1.0], [1.0, 1.0 + 1e-8]]),
    }
    print(f"{'Matrix':<24} {'kappa(A)':>12} {'Verst. median':>14} {'Verst. max':>12}")
    print("-" * 74)
    for name, A in modelle.items():
        median, maximum = fehlerverstaerkung(A)
        print(f"{name:<24} {np.linalg.cond(A):>12.2e} {median:>14.2e} {maximum:>12.2e}")

    print("\nLesart: Bei der zweiten Matrix wird aus einem Datenfehler in der")
    print("10. Nachkommastelle ein Loesungsfehler in der 2. Nachkommastelle.")
```

```
    print("Das Modell ist mathematisch korrekt - und praktisch wertlos.")


# --- Experiment 2: Ruiz-Equilibrierung ------------------------------------

def ruiz_equilibrierung(A: np.ndarray, durchlaeufe: int = 20
                        ) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Skaliert A iterativ so, dass alle Zeilen- und Spaltenmaxima nahe 1
    liegen (Ruiz 2001).

    In jedem Durchlauf wird jede Zeile durch die Wurzel ihres Betragsmaximums
    geteilt, danach jede Spalte. Das Verfahren konvergiert schnell und braucht
    keinerlei Wissen ueber die Bedeutung der Zahlen - genau deshalb steckt es
    in jedem ernsthaften Solver als Vorverarbeitung.

    Liefert (A_skaliert, zeilenfaktor r, spaltenfaktor c) mit
        A_skaliert = diag(r) @ A @ diag(c)
    """
    m, n = A.shape
    r = np.ones(m)
    c = np.ones(n)
    A_s = A.astype(float).copy()

    for _ in range(durchlaeufe):
        zeilen_max = np.abs(A_s).max(axis=1)
        zeilen_max[zeilen_max == 0] = 1.0
        d_r = 1.0 / np.sqrt(zeilen_max)
        A_s = d_r[:, None] * A_s
        r *= d_r

        spalten_max = np.abs(A_s).max(axis=0)
        spalten_max[spalten_max == 0] = 1.0
        d_c = 1.0 / np.sqrt(spalten_max)
        A_s = A_s * d_c[None, :]
        c *= d_c

    return A_s, r, c


def zeige_experiment_2() -> None:
    print("\n" + "=" * 74)
    print("  2. RUIZ-EQUILIBRIERUNG: EINHEITEN GERADERUECKEN")
    print("=" * 74)
    print("Ein Produktionsmodell, in dem vier Ressourcen in voellig")
    print("verschiedenen Einheiten gemessen werden:")
    print("  Zeile 1: Kapitalbindung in Euro          (Groessenordnung 1e7)")
    print("  Zeile 2: Katalysatorverbrauch in Tonnen  (Groessenordnung 1e-3)")
    print("  Zeile 3: Energie in Wattsekunden         (Groessenordnung 1e5)")
```

```
    print("  Zeile 4: Ausschussquote als Anteil       (Groessenordnung 1e-2)\n")

    # Quadratisch gewaehlt, damit die Loesung eindeutig ist und die
    # Ruecktransformation unten wirklich etwas beweist.
    n = 4
    grundmatrix = RNG.uniform(0.5, 2.0, size=(n, n))
    einheiten = np.array([1e7, 1e-3, 1e5, 1e-2])
    A = grundmatrix * einheiten[:, None]

    A_s, r, c = ruiz_equilibrierung(A)

    print(f"{'':<28} {'kappa(A)':>12} {'groesster Eintrag':>18} "
          f"{'kleinster Eintrag':>18}")
    print("-" * 74)
    for name, matrix in [("vor der Skalierung", A), ("nach Ruiz-Equilibrierung", A_s)]:
        betraege = np.abs(matrix)
        print(f"{name:<28} {np.linalg.cond(matrix):>12.2e} "
              f"{betraege.max():>18.2e} {betraege.min():>18.2e}")

    # Gegenprobe: Das skalierte Modell beschreibt dasselbe Problem. Wer x_s
    # loest, erhaelt die urspruengliche Loesung durch x = c * x_s.
    x_wahr = RNG.uniform(1.0, 5.0, size=n)
    b = A @ x_wahr
    b_s = r * b
    x_s = np.linalg.solve(A_s, b_s)
    x_zurueck = c * x_s
    print(f"\nRuecktransformation x = c * x_s: groesste Abweichung zur wahren "
          f"Loesung {np.abs(x_zurueck - x_wahr).max():.2e}")
    print("Die Skalierung ist also verlustfrei - sie aendert nur die Zahlen,")
    print("nicht das Problem.")


# --- Experiment 3: Toleranzen und der int()-Fehler --------------------------

def zeige_experiment_3() -> None:
    print("\n" + "=" * 74)
    print("  3. TOLERANZEN: WARUM int() DIE FALSCHE RUNDUNG IST")
    print("=" * 74)

    # Ein LP, dessen Optimum bei x = 1 liegt, aber vom Solver nur bis auf
    # seine Toleranz getroffen wird.
    ergebnis = linprog(c=[-1.0], A_ub=[[1.0]], b_ub=[1.0],
                       bounds=[(0, None)], method="highs")
    wert = float(ergebnis.x[0])
    print(f"Solver liefert x = {wert!r}")

    # Typische Werte, wie sie aus MILP-Solvern zurueckkommen.
    beispiele = [0.99999998, 1.00000002, 0.49999999, 2.9999999]
```

```
    print(f"\n{'Solverwert':>14} {'int()':>8} {'round()':>9} {'Kommentar'}")
    print("-" * 74)
    kommentare = {
        0.99999998: "int() macht aus einer JA- eine NEIN-Entscheidung",
        1.00000002: "hier ginge int() zufaellig gut - Verlass ist keiner",
        0.49999999: "echt unentschieden: Modell oder Toleranz pruefen!",
        2.9999999:  "3 Maschinen werden zu 2 - der Plan geht nicht auf",
    }
    for wert_b in beispiele:
        print(f"{wert_b:>14.8f} {int(wert_b):>8} {round(wert_b):>9} "
              f"{kommentare[wert_b]}")

    print("\nRichtige Vorgehensweise: gegen die Solver-Toleranz pruefen,")
    print("dann erst runden - und den Zweifelsfall melden statt still zu raten.")

    def sichere_ganzzahl(wert: float, toleranz: float = 1e-6) -> int:
        naechste = round(wert)
        if abs(wert - naechste) > toleranz:
            raise ValueError(
                f"{wert} ist {abs(wert - naechste):.2e} von der naechsten ganzen "
                f"Zahl entfernt - das ist mehr als die Toleranz {toleranz}. "
                "Ganzzahligkeit im Modell pruefen.")
        return naechste

    for wert_b in beispiele:
        try:
            print(f"  sichere_ganzzahl({wert_b}) = {sichere_ganzzahl(wert_b)}")
        except ValueError as fehler:
            print(f"  sichere_ganzzahl({wert_b}) -> ValueError: {fehler}")


if __name__ == "__main__":
    zeige_experiment_1()
    zeige_experiment_2()
    zeige_experiment_3()
    print("\n" + "=" * 74)
    print("Merksatz: Skalieren Sie Ihre Daten, BEVOR der Solver sie sieht -")
    print("und runden Sie Solver-Ergebnisse NIE ohne Toleranzpruefung.")
    print("=" * 74)
```

Erwartete Ausgabe:

```
==========================================================================
  1. KONDITIONSZAHL ALS FEHLERVERSTAERKER
==========================================================================
Zwei Gleichungssysteme, beide exakt loesbar mit x = (1, 1).
Die rechte Seite wird um relativ 1e-10 gestoert - so viel Unsicherheit
steckt in JEDER gemessenen Betriebszahl allemal.
```

```
Matrix                       kappa(A)  Verst. median   Verst. max
------------------------------------------------------------------------
gut konditioniert            2.62e+00       1.86e+00     2.56e+00
fast parallele Zeilen        4.00e+08       2.57e+08     4.00e+08

Lesart: Bei der zweiten Matrix wird aus einem Datenfehler in der
10. Nachkommastelle ein Loesungsfehler in der 2. Nachkommastelle.
Das Modell ist mathematisch korrekt - und praktisch wertlos.

========================================================================
  2. RUIZ-EQUILIBRIERUNG: EINHEITEN GERADERUECKEN
========================================================================
Ein Produktionsmodell, in dem vier Ressourcen in voellig
verschiedenen Einheiten gemessen werden:
  Zeile 1: Kapitalbindung in Euro          (Groessenordnung 1e7)
  Zeile 2: Katalysatorverbrauch in Tonnen  (Groessenordnung 1e-3)
  Zeile 3: Energie in Wattsekunden         (Groessenordnung 1e5)
  Zeile 4: Ausschussquote als Anteil       (Groessenordnung 1e-2)

                               kappa(A)  groesster Eintrag  kleinster Eintrag
------------------------------------------------------------------------
vor der Skalierung             8.67e+10           1.52e+07           1.35e-03
nach Ruiz-Equilibrierung       4.69e+01           1.00e+00           4.24e-01

Ruecktransformation x = c * x_s: groesste Abweichung zur wahren Loesung 2.35e-14
Die Skalierung ist also verlustfrei - sie aendert nur die Zahlen,
nicht das Problem.

========================================================================
  3. TOLERANZEN: WARUM int() DIE FALSCHE RUNDUNG IST
========================================================================
Solver liefert x = 1.0

    Solverwert    int()   round() Kommentar
------------------------------------------------------------------------
    0.99999998        0         1 int() macht aus einer JA- eine NEIN-Entscheidung
    1.00000002        1         1 hier ginge int() zufaellig gut - Verlass ist keiner
    0.49999999        0         0 echt unentschieden: Modell oder Toleranz pruefen!
    2.99999990        2         3 3 Maschinen werden zu 2 - der Plan geht nicht auf

Richtige Vorgehensweise: gegen die Solver-Toleranz pruefen,
dann erst runden - und den Zweifelsfall melden statt still zu raten.
  sichere_ganzzahl(0.99999998) = 1
  sichere_ganzzahl(1.00000002) = 1
  sichere_ganzzahl(0.49999999) -> ValueError: 0.49999999 ist 5.00e-01 von der naechsten
   ↪  ganzen Zahl entfernt - das ist mehr als die Toleranz 1e-06. Ganzzahligkeit im Modell
   ↪  pruefen.
  sichere_ganzzahl(2.9999999) = 3
```

```
======================================================================
Merksatz: Skalieren Sie Ihre Daten, BEVOR der Solver sie sieht -
und runden Sie Solver-Ergebnisse NIE ohne Toleranzpruefung.
======================================================================
```

Code-Durchgang

Stelle	Was passiert	Warum das die Kernaussage trägt
Störung in **zufälliger Richtung**	200 Versuche statt einer festen Störung	Eine proportionale Störung ($b \cdot 1{,}0000000001$) trifft die empfindliche Richtung meist gar nicht — man sähe fälschlich gar keinen Effekt. Die Schwäche zeigt sich nur, wenn man in alle Richtungen stößt.
`Verst. max = 4.00e+08`	erreicht exakt $\kappa(\mathbf{A})$	Die Schranke aus der Formel oben ist **scharf**: Es gibt eine Richtung, in der die volle Verstärkung eintritt. Sie ist keine pessimistische Abschätzung.
Ruiz: quadratische Matrix	4×4 statt 4×6	Nur bei eindeutiger Lösung beweist die Rückrechnung etwas. Bei unterbestimmten Systemen liefert `lstsq` die Minimum-Norm-Lösung, und die ist unter Skalierung *nicht* invariant — eine Falle, in die man leicht tappt.
Abweichung `2.35e-14`	Rückrechnung trifft die wahre Lösung	Der Beleg, dass Skalieren nichts kaputt macht: Es ändert die Zahlen, nicht das Problem.
`sichere_ganzzahl()`	prüft Abstand zur nächsten ganzen Zahl	Der Zweifelsfall (`0.49999999`) wird **gemeldet**, nicht stillschweigend geraten. Genau diese Funktion gehört in jedes MILP-Auswertungsskript.

Typische Fehler

- **`int()` auf Solver-Ergebnisse.** `int(0.99999998) == 0` — aus einem „ja, Standort eröffnen“ wird ein „nein“. `int()` schneidet ab, es rundet nicht. Und selbst `round()` ist nur mit vorheriger Toleranzprüfung vertretbar.

- **Große Zahlen für „Sicherheit“.** Ein Big-M von 10^9 „damit es garantiert reicht“ verschlechtert κ um neun Größenordnungen und erzeugt genau die Instabilität, die man vermeiden wollte.
- **Auf die interne Solver-Skalierung vertrauen.** Sie hilft der Rechnung, aber sie kann Ihnen nicht sagen, dass Sie Stück und Tonnen vermischt haben.
- **κ nie nachschauen.** Eine Zeile `np.linalg.cond(A)` beim Modellaufbau, geloggt neben der Laufzeit, hätte in vielen Projekten Wochen Fehlersuche gespart.

2.8 Übungsaufgaben

Lösungen: Abschnitt A.2.

Aufgabe 2.1 □ — **Matrixform lesen.** Gegeben $\mathbf{c} = (4, 1, 6)^\top$, $\mathbf{A} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \end{pmatrix}$, $\mathbf{b} = (10, 12)^\top$. Schreiben Sie das Modell ausgeschrieben hin. Wie viele Variablen und wie viele Nebenbedingungen hat es?

Aufgabe 2.2 □ — **Standardform herstellen.** Bringen Sie in die Form $\min \mathbf{c}^\top \mathbf{x}$ u. d. N. $\mathbf{A}\mathbf{x} \le \mathbf{b}$:

$$\max\ 7x_1 - 2x_2 \quad \text{u. d. N.} \quad 4x_1 + x_2 \ge 20, \quad x_1 - x_2 = 3, \quad x_1, x_2 \ge 0$$

Aufgabe 2.3 □□ — **Ecken von Hand.** Für $\max 2x_1 + 3x_2$ u. d. N. $x_1 + x_2 \le 8$, $2x_1 + x_2 \le 12$, $x_1, x_2 \ge 0$: (a) Zeichnen Sie den zulässigen Bereich auf kariertes Papier. (b) Bestimmen Sie alle Ecken. (c) Werten Sie Z in jeder Ecke aus und geben Sie das Optimum an. (d) Was passiert, wenn die Zielfunktion zu $\max 2x_1 + 2x_2$ wird? Wie viele optimale Lösungen gibt es dann?

Aufgabe 2.4 □□ — **Konvexität beurteilen.** Konvex oder nicht? Begründen Sie jeweils kurz: (a) $f(x) = 5x + 3$ (b) $f(x) = x^4$ (c) $f(x) = \sqrt{x}$ für $x \ge 0$ (d) $f(x) = \ln(x)$ für $x > 0$ (e) $f(x_1, x_2) = x_1^2 + x_2^2$ (f) Die Menge $\{(x_1, x_2) : x_1^2 + x_2^2 \le 4\}$ (g) Die Menge $\{(x_1, x_2) : x_1 \cdot x_2 \ge 1,\ x_1, x_2 > 0\}$

Aufgabe 2.5 □□ — **Positive Semidefinitheit prüfen.** Prüfen Sie mit NumPy, welche dieser Matrizen als Kovarianzmatrix taugen (alle Eigenwerte ≥ 0), und geben Sie bei Nichteignung die implizierte Korrelation an:

$$\mathbf{P}_1 = \begin{pmatrix} 4 & 1 \\ 1 & 9 \end{pmatrix}, \quad \mathbf{P}_2 = \begin{pmatrix} 4 & 7 \\ 7 & 9 \end{pmatrix}, \quad \mathbf{P}_3 = \begin{pmatrix} 1 & 0.5 & 0.5 \\ 0.5 & 1 & 0.5 \\ 0.5 & 0.5 & 1 \end{pmatrix}$$

Aufgabe 2.6 □□□ — **Eigene Visualisierung.** Passen Sie `Visualisierung_Loesungsraum.py` an das Bäckerei-Problem aus Kapitel 1 (die Bäckerei-Handrechnung aus Kapitel 1) an: $\max 2{,}5x_1 + 3{,}0x_2$ u. d. N. $0{,}5x_1 + 0{,}6x_2 \le 90$, $4x_1 + 3x_2 \le 600$, $x_1 \ge 40$. Geben Sie alle Ecken mit Zielwerten aus und zeichnen Sie das Polyeder.

Aufgabe 2.7 □□□ — **Konvexität einer Kovarianzmatrix reparieren.** Die Matrix $\mathbf{P}_2$ aus der Aufgabe *Positive Semidefinitheit prüfen* ist nicht positiv semidefinit. Schreiben Sie eine Funktion `repariere(P)`, die die negativen Eigenwerte auf 0 setzt und die Matrix rekonstruiert (*Eigenwert-Clipping*): $\mathbf{P}_{\text{rep}} = \mathbf{V} \max(\Lambda, 0) \mathbf{V}^\top$. Prüfen Sie das Ergebnis und vergleichen Sie es mit dem Original.

2.9 Finde den Denkfehler

Finde den Denkfehler: Die unauffällige Transposition

Ein Kollege überträgt ein Produktionsmodell in Matrixform. Zwei Produkte, zwei Ressourcen — die Technologiematrix ist quadratisch:

$$\max\ 3x_1 + 5x_2 \quad \text{u. d. N.} \quad \underbrace{\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}}_{\mathbf{A}} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 8 \\ 12 \end{pmatrix}$$

Beim Abtippen aus der Tabelle liest er sie spaltenweise statt zeilenweise ein:

```
import numpy as np
from scipy.optimize import linprog

c = np.array([3.0, 5.0])
A = np.array([[1.0, 3.0],          # spaltenweise abgetippt statt zeilenweise
              [2.0, 1.0]])
b = np.array([8.0, 12.0])

r = linprog(-c, A_ub=A, b_ub=b, bounds=[(0, None)] * 2)
print("x =", np.round(r.x, 2), " Z =", round(-r.fun, 2))
```

Ausgabe:

```
x = [5.6 0.8]  Z = 20.8
```

Kein Fehler, kein Warnhinweis, ein völlig unauffälliger Zielwert. Das korrekte Optimum wäre $\mathbf{x}^* = (3{,}2;\ 2{,}4)$ mit $Z^* = 21{,}6$ gewesen.

Ihre Aufgabe: (a) Warum fällt dieser Fehler weder Python noch dem Solver auf? (b) Prüfen Sie den Plan $(5{,}6;\ 0{,}8)$ gegen die **echten** Nebenbedingungen — was stellen Sie fest? (c) Wieso ist es besonders gefährlich, dass der falsche Zielwert *kleiner* ist als der richtige? (d) Welche zwei Zeilen Code hätten den Fehler beim Einlesen sofort aufgedeckt?

Auflösung: *Abschnitt A.2.*

2.10 Micro-Quiz

Micro-Quiz 2: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Ein lineares Programm hat 6 Variablen und 4 Ungleichungen (plus Nichtnegativität). Was garantiert der Fundamentalsatz der linearen Optimierung? (a) Dass es genau eine optimale Lösung gibt. (b) Dass es — sofern ein Optimum existiert — mindestens eine optimale Lösung in einer Ecke des zulässigen Polyeders gibt. (c) Dass sich das Optimum durch Ausprobieren aller Ecken in vertretbarer Zeit finden lässt.

2. `np.linalg.cond(A)` liefert für Ihre Technologiematrix den Wert $3 \cdot 10^{11}$. Was folgt daraus? (a) Das Modell ist unlösbar und muss neu formuliert werden. (b) Von den rund 16 Stellen Rechengenauigkeit bleiben etwa 5 übrig — die Ergebnisse sind mit Vorsicht zu genießen, und die Ursache liegt meist in gemischten Einheiten. (c) Der Solver braucht $3 \cdot 10^{11}$ Iterationen.

3. Ein MILP-Solver liefert für eine Binärvariable den Wert `0.99999998`. Wie werten Sie ihn aus? (a) `int(wert)` — das ist die Standardumwandlung in Python. (b) Wert unverändert weiterreichen, der Solver wird schon recht haben. (c) Abstand zur nächsten ganzen Zahl gegen die Solver-Toleranz prüfen, dann runden — und einen Wert wie `0.5` als Fehler melden statt zu raten.

2.11 Selbsttest

Antworten: Anhang A.

1. Was bedeutet der Ausdruck $\mathbf{c}^{\mathsf{T}}\mathbf{x}$ inhaltlich, und was für ein Objekt ist das Ergebnis?
2. Warum steht in der Technologiematrix eine 0, wenn eine Variable in einer Bedingung nicht vorkommt — warum lässt man den Eintrag nicht weg?
3. Formulieren Sie den Fundamentalsatz der linearen Optimierung und nennen Sie seinen praktischen Nutzen.
4. Warum ist ein ganzzahliges Problem nicht konvex, obwohl alle Funktionen darin linear sind?
5. Ein Kollege sagt: „Mein Optimierer liefert je nach Startwert unterschiedliche Ergebnisse — das ist wohl ein Bug.“ Was antworten Sie?

2.12 Zusammenfassung

- **Matrixform** bündelt beliebig viele Variablen und Bedingungen in drei Zeilen: $\min \mathbf{c}^\top \mathbf{x}$ u. d. N. $\mathbf{Ax} \leq \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$. Jede Zeile von $\mathbf{A}$ ist eine Ressource, jede Spalte eine Variable.
- **Der zulässige Bereich** ist der Schnitt aller Halbräume: ein konvexes Polyeder.
- **Der Fundamentalsatz** garantiert, dass ein Optimum in einer Ecke liegt — endlich viele Kandidaten statt unendlich vieler Punkte.
- **Kondition und Skalierung** entscheiden darüber, ob aus mathematisch korrekten Zahlen auch brauchbare werden: $\kappa(\mathbf{A})$ ist der Verstärkungsfaktor zwischen Daten- und Lösungsfehler, gemischte Einheiten sind seine häufigste Ursache, Ruiz-Equilibrierung die Gegenmaßnahme — und Solver-Ergebnisse werden nur nach Toleranzprüfung gerundet.
- **Konvexität** ist die eigentliche Trennlinie zwischen leicht und schwer. Bei konvexen Problemen ist jedes lokale Optimum global; bei nicht-konvexen hängt das Ergebnis vom Startpunkt ab.
- **Prüfen Sie Matrizen**, bevor Sie sie als Kovarianzmatrix verwenden: alle Eigenwerte müssen ≥ 0 sein.

Ausblick. Kapitel 3 ordnet das Python-Ökosystem: Welche Bibliothek löst welche Problemklasse, und warum gibt es überhaupt mehrere?

Kapitel 3: Das Python-Ökosystem für OR — Solver, Bindings und Modellierungsschichten

Kapitel auf einen Blick

Worum geht es? Warum es für Optimierung in Python mehrere konkurrierende Bibliotheken gibt, was sie unterscheidet, und wie Sie in unter einer Minute die richtige auswählen.

Voraussetzungen: Kapitel 1 und Kapitel 2.

Danach können Sie: Für ein gegebenes Problem begründet einen Solver wählen, dasselbe Modell in verschiedenen Bibliotheken formulieren, die Ergebnisse gegeneinander prüfen — und messen, ob Ihre Laufzeit überhaupt im Solver entsteht oder schon davor.

Zeitbedarf: ca. 4 Stunden.

Programme:

```
Ein_System_Vier_Ansaetze.py
Solver_Wahl.py
Modellierungsschichten.py
Vektorisierte_Modellgenerierung.py
```

Notebook: `Notebooks_04/oekosystem.ipynb`

3.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Ein LP ohne jede Installation

Ein Futtermittelhersteller mischt zwei Rohstoffe zu möglichst geringen Kosten. Jeder Kilogramm Mischung muss mindestens 20 g Protein und 5 g Fett enthalten.

Rohstoff	Preis je kg	Protein je kg	Fett je kg
Weizenschrot	0,42 €	12 g	2 g
Sojaschrot	0,88 €	44 g	15 g

Sie brauchen dafür **keine** zusätzliche Bibliothek — SciPy genügt, und SciPy ist in jeder wissenschaftlichen Python-Installation schon da:

```
from scipy.optimize import linprog

# linprog MINIMIERT und kennt nur "<=". Mindestgehalte werden also negiert.
res = linprog(c=[0.42, 0.88],                       # Kosten je kg
              A_ub=[[-12, -44], [-2, -15]],         # -Protein, -Fett
              b_ub=[-20, -5],                       # >= 20 g bzw. >= 5 g
              A_eq=[[1, 1]], b_eq=[1],              # ergibt zusammen 1 kg
              bounds=[(0, 1), (0, 1)])

print(res.message)
print(f"Weizen {res.x[0]:.3f} kg, Soja {res.x[1]:.3f} kg -> {res.fun:.4f} EUR/kg")
```

Ausgabe:

```
Optimization terminated successfully. (HiGHS Status 7: Optimal)
Weizen 0.750 kg, Soja 0.250 kg -> 0.5350 EUR/kg
```

Sie haben soeben HiGHS benutzt — denselben C++-Solver, der auch hinter `highspy`, hinter CVXPY und in vielen kommerziellen Systemen steckt. `scipy.optimize.linprog` ist nur die dünnste denkbare Hülle darum.

Das ist die zentrale Botschaft dieses Kapitels: **Modellierungsschicht und Solver sind zwei verschiedene Dinge.** Die Bibliothek, in der Sie Ihr Modell hinschreiben, bestimmt, wie angenehm die Arbeit ist. Der Solver dahinter bestimmt, wie schnell gerechnet wird. Beide lassen sich unabhängig voneinander tauschen — und genau davon handelt der Rest des Kapitels.

Zwei Stolpersteine stecken schon in diesen sechs Zeilen

- `linprog` **minimiert immer**. Wer maximieren will, negiert die Zielfunktion — und darf nicht vergessen, das Ergebnis zurückzudrehen.
- `linprog` kennt **nur <=**. Ein Mindestgehalt „≥ 20“ wird zu „$-12x_1 - 44x_2 \leq -20$“. Ein Vorzeichenfehler an dieser Stelle liefert eine perfekt aussehende Lösung, die das Gegenteil des Gewollten erfüllt.

Beide Fallen verschwinden, sobald man eine Modellierungsschicht wie CVXPY oder Pyomo benutzt, in der >= einfach >= heißt. Das ist ein Hauptgrund, warum es sie gibt.

3.2 Lernziele

Nach diesem Kapitel können Sie …

1. … die Zwei-Schichten-Architektur (Modellierung vs. Solver) erklären.
2. … anhand von drei Fragen entscheiden, welche Bibliothek zu Ihrem Problem passt.
3. … dasselbe lineare Programm in `scipy.optimize`, `highspy`, `ortools` und `cvxpy` formulieren.
4. … Ergebnisse verschiedener Solver gegeneinander validieren (*Cross-Check*).
5. … einschätzen, wann sich der Aufwand einer Low-Level-Schnittstelle lohnt und wann nicht.
6. … Pyomo und Linopy einordnen und begründen, wann sich der Umstieg auf sie lohnt.
7. … Aufbau- und Lösezeit getrennt messen und einen Modellaufbau mit NumPy oder Polars vektorisieren, statt ihn in Python-Schleifen zu erzeugen.

3.3 Die Zwei-Schichten-Architektur

Im modernen Operations Research schreibt man Optimierungsalgorithmen nicht selbst. Man nutzt eine **zweischichtige Architektur**:

1. **Modellierungsschicht (*Frontend*, DSL):** Python-Bibliotheken, mit denen Entscheidungsvariablen, Zielfunktion und Nebenbedingungen in mathematiknaher Syntax formuliert werden. Hier arbeiten Sie.
2. **Solver-Schicht (*Backend*, Engine):** Hochoptimierte C++- oder Fortran-Bibliotheken, die das Modell in Matrixstrukturen übersetzen und mit spezialisierten Algorithmen lösen (Dual Simplex, Interior-Point, Branch-and-Cut, CDCL-SAT-Suche).

   Merksatz Die Modellierungsschicht bestimmt, wie angenehm Sie arbeiten. Die Solver-Schicht bestimmt, wie schnell gerechnet wird. Beides ist entkoppelt — man kann dieselbe CVXPY-Formulierung mit fünf verschiedenen Solvern lösen.

Warum diese Trennung nützlich ist: Ein CVXPY-Modell, das heute mit dem freien Solver Clarabel läuft, läuft morgen ohne Codeänderung mit dem kommerziellen Gurobi — man tauscht ein Argument. Das schützt vor Herstellerbindung und erlaubt, im Projekt klein anzufangen.

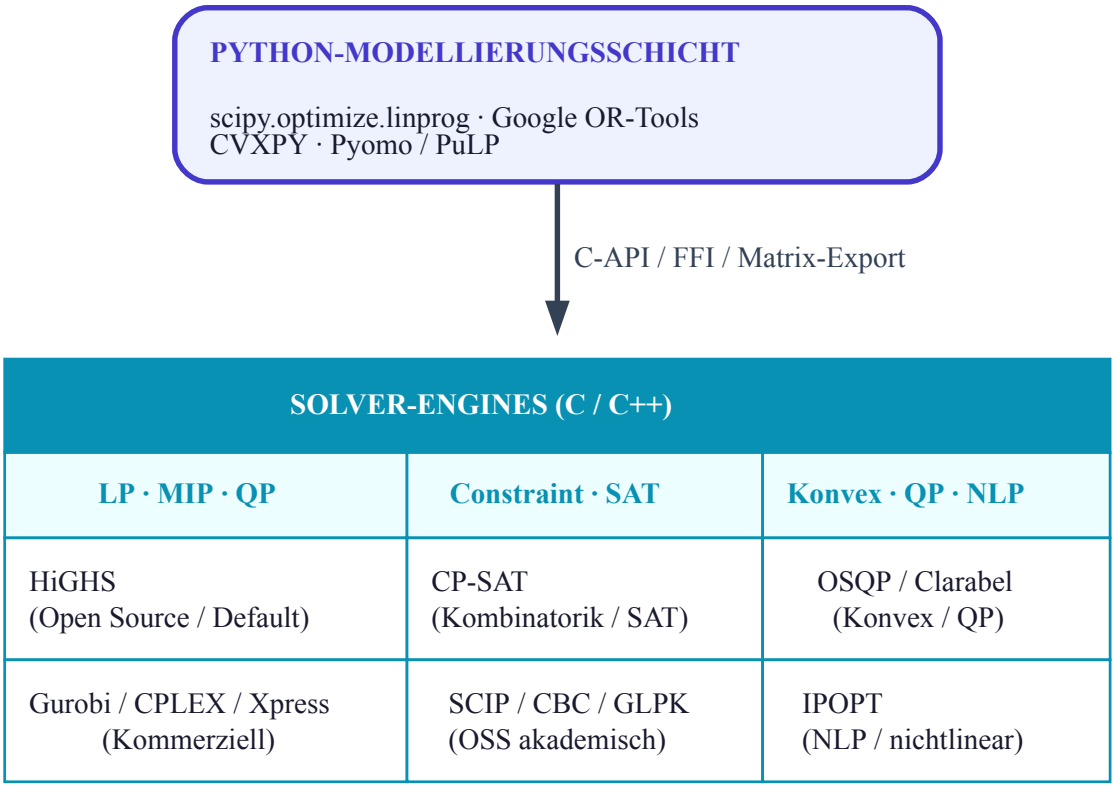

Abbildung 6: Abb. 3.1: Modellierungsschicht und Solver-Engines

3.4 Die Werkzeuge im Vergleich

Bibliothek	Stärken & Problemklassen	Wann einsetzen?	Typische Backends
`scipy.optimize`	Standard-LPs, kontinuierliche nichtlineare Optimierung, Wurzelsuche	Für elementare LPs und schnelle Prototypen ohne zusätzliche schwere Abhängigkeiten	HiGHS (`highs-ds`, `highs-ipm`)
`ortools` (Google)	Diskrete Planung, Schichten, Vertretungspläne, Routing (VRP/TSP), kombinatorische Logik	Wenn diskrete Ja/Nein-Entscheidungen, Zeitfenster und Wenn-Dann-Regeln dominieren	CP-SAT, GLOP, Routing Engine
`cvxpy`	Konvexe Optimierung, quadratische Programme, Risikomodelle, Portfolios	**Erste Wahl für Finanzoptimierung**, Markowitz, CVaR, L_1-Transaktionskosten	Clarabel, OSQP, ECOS, SCS, HiGHS

Bibliothek	Stärken & Problemklassen	Wann einsetzen?	Typische Backends
`highspy`	Direkte Solver-Steuerung für LP/MILP, maximale Kontrolle	Wenn Sie Modelle wiederholt und schnell aufbauen oder Solver-Details steuern müssen	HiGHS
`Pyomo`	Universelle Modellierungssprache (LP, MILP, MINLP)	Industrielle Großmodelle mit strikter Trennung von Daten und Modell	HiGHS, Gurobi, CPLEX, SCIP, IPOPT
`Linopy`	LP/MILP über beschriftete Arrays (`xarray`)	Modelle mit zehntausenden gleichartigen Nebenbedingungen (Energie, Netze, Zeitreihen)	HiGHS, GLPK, CBC, Gurobi

Die Entscheidung in drei Fragen

Statt die Tabelle auswendig zu lernen, beantworten Sie drei Fragen:

> **Frage 1: Gibt es Ja/Nein-Entscheidungen oder Reihenfolgen? → Ja:** OR-Tools (CP-SAT) bei Zuweisung/Scheduling, oder MILP über HiGHS bei ökonomischen Fixkostenmodellen. → **Nein:** weiter zu Frage 2.
>
> **Frage 2: Ist die Zielfunktion linear? → Ja:** `scipy.optimize.linprog` (klein) oder `highspy` (groß, wiederholt). → **Nein, aber konvex** (Quadrate, Normen, `log`, `exp` in konvexer Kombination): **CVXPY**. → **Nein und nicht konvex:** `scipy.optimize.minimize` — im Bewusstsein, dass nur ein lokales Optimum herauskommt.
>
> **Frage 3: Ist es ein Routing-Problem mit Fahrzeugen, Depots und Zeitfenstern? → Ja:** OR-Tools **Routing Library** (nicht CP-SAT von Hand nachbauen).

Das folgende kleine Programm gießt diese Logik in Code — als Nachschlagehilfe für den eigenen Gebrauch.

```
#!/usr/bin/env python3

# Solver_Wahl.py
"""
Kapitel Oekosystem: Entscheidungshilfe zur Solverwahl.
Beantwortet drei Fragen und empfiehlt Bibliothek + Backend.
"""

from dataclasses import dataclass
```

```python
@dataclass
class Problem:
    diskrete_entscheidungen: bool   # Ja/Nein-Variablen, Zuordnungen, Reihenfolgen?
    zielfunktion: str               # "linear" | "quadratisch" | "konvex" | "beliebig"
    routing: bool = False           # Fahrzeuge, Depots, Zeitfenster?
    groesse: str = "klein"          # "klein" (<10^3 Var.) | "gross"
    wiederholte_laeufe: bool = False


def empfehle(p: Problem) -> tuple[str, str, str]:
    """Gibt (Bibliothek, Backend, Begründung) zurück."""
    if p.routing:
        return ("ortools.constraint_solver (Routing)", "Routing Engine",
                "Spezialisierte Heuristiken für VRP/TSP schlagen jedes selbstgebaute Modell.")

    if p.diskrete_entscheidungen:
        if p.zielfunktion in ("linear",) and not p.routing:
            if p.groesse == "gross":
                return ("highspy", "HiGHS Branch-and-Cut",
                        "MILP mit ökonomischer Struktur (Fixkosten, Kardinalität).")
            return ("ortools.sat (CP-SAT)", "CP-SAT",
                    "Logische Regeln und Zuweisungen: CP-SAT propagiert sehr effizient.")
        return ("ortools.sat (CP-SAT)", "CP-SAT",
                "Diskrete Struktur dominiert; CP-SAT verarbeitet auch nichtlineare Logik.")

    if p.zielfunktion == "linear":
        if p.wiederholte_laeufe or p.groesse == "gross":
            return ("highspy", "HiGHS Dual Simplex",
                    "Modell einmal aufbauen, Parameter ändern, wiederholt lösen.")
        return ("scipy.optimize.linprog", "HiGHS",
                "Kleinstes Setup, keine zusätzliche Abhängigkeit.")

    if p.zielfunktion in ("quadratisch", "konvex"):
        return ("cvxpy", "Clarabel / OSQP",
                "Konvexität wird automatisch geprüft; Portfolio-Standard.")

    return ("scipy.optimize.minimize", "SLSQP / trust-constr",
            "Nicht konvex: nur lokales Optimum, Startpunkt variieren und vergleichen!")


BEISPIELE = {
    "Vertretungsplan Schule":        Problem(True,  "linear", groesse="klein"),
    "Produktionsplanung (LP)":       Problem(False, "linear", groesse="klein"),
    "Produktionsplanung, 50k Var.":  Problem(False, "linear", groesse="gross"),
    "Portfolio Markowitz":           Problem(False, "quadratisch"),
    "Portfolio mit max. 5 Titeln":   Problem(True,  "quadratisch"),
```

```
    "Liefertouren mit Zeitfenstern": Problem(True,  "linear", routing=True),
    "Entropie-Allokation (NLP)":     Problem(False, "beliebig"),
    "Backtest, 60x neu optimieren":  Problem(False, "konvex", wiederholte_laeufe=True),
}

if __name__ == "__main__":
    print("=" * 92)
    print("  SOLVER-EMPFEHLUNG")
    print("=" * 92)
    for name, p in BEISPIELE.items():
        lib, backend, grund = empfehle(p)
        print(f"\n{name}")
        print(f"    -> Bibliothek: {lib}")
        print(f"       Backend:    {backend}")
        print(f"       Grund:      {grund}")
    print("\n" + "=" * 92)
```

3.5 Ein System — vier Programmieransätze

Nichts macht die Unterschiede so deutlich wie dasselbe Problem, viermal gelöst. Wir nehmen:

$$\max \; 10x_1 + 15x_2 + 25x_3$$

$$\text{u. d. N.} \quad x_1 + x_2 + 2x_3 \le 40, \qquad 2x_1 + 3x_2 + x_3 \le 50, \qquad x_1, x_2, x_3 \ge 0$$

Handrechnung 3.1: Erwarten Sie das Ergebnis

Bevor Sie das Programm laufen lassen: Welches Produkt ist am attraktivsten?

Produkt 3 bringt den höchsten Ertrag (25) und verbraucht wenig von Ressource 2 (nur 1). Es verbraucht aber doppelt so viel von Ressource 1. Rechnen wir den Ertrag **pro Einheit Engpassressource**:

Produkt	Ertrag	pro Einheit Ressource 1	pro Einheit Ressource 2
x_1	10	$10/1 = 10$	$10/2 = 5$
x_2	15	$15/1 = 15$	$15/3 = 5$
x_3	25	$25/2 = 12{,}5$	$25/1 = 25$

Kein Produkt ist in beiden Spalten der Sieger — deshalb braucht es überhaupt einen Optimierer. Probieren wir zwei Randlösungen: * Nur x_3: Ressource 1 erlaubt $40/2 = 20$ Stück, Ressource 2 erlaubt 50. Also $x_3 = 20$, Ertrag $= 500$. Ressource 2 ist dann nur zu $20/50$ ausgelastet. * $x_3 = 20$ plus x_2? Ressource 1 ist bereits voll (40), es geht nichts mehr. * Mischung x_2 und x_3:

$x_2+2x_3 = 40$ und $3x_2+x_3 = 50$ □ $x_2 = 12$, $x_3 = 14$, Ertrag $= 15{\cdot}12+25{\cdot}14 = 180+350 = 530$. Besser!

Erwartetes Ergebnis: $Z^* = 530$ mit $x_1 = 0$, $x_2 = 12$, $x_3 = 14$.

□ **Vorab: eine Stolperfalle der Installation**

Auf vielen Systemen lassen sich **`ortools` und `highspy` nicht im selben Python-Prozess importieren**. Beide Pakete bringen ihre eigene, unterschiedlich kompilierte Kopie des HiGHS-Solvers mit; der dynamische Linker löst die Symbole dann falsch auf. Die Fehlermeldung sieht so aus:

```
ImportError: .../highspy/_core...so: undefined symbol: _ZN5Highs13releaseMemoryEv
```

oder umgekehrt

```
ImportError: .../ortools/.libs/libortools.so.9: undefined symbol:
↪  _Z19setLocalOptionValue...
```

Das ist **kein Fehler in Ihrem Code**, und bei `ortools` und `highspy` direkt hilft auch die Reihenfolge der Importe nicht — es trifft beide Richtungen. `cvxpy` verträgt sich mit beiden, importiert aber ein installiertes `highspy` bei der Solver-Erkennung selbst mit (siehe *).

Kombination in einem Prozess	Funktioniert?
`ortools` + `highspy`	□ nein (beide Richtungen)
`ortools` + `cvxpy`	□ ja*
`highspy` + `cvxpy`	□ ja
`scipy` + beliebig	□ ja

* ist `highspy` installiert, gilt die `ortools`+`highspy`-Zeile entsprechend: erst `cvxpy`, dann `ortools` importiert → Crash; erst `ortools`, dann `cvxpy` → läuft, CVXPY nur ohne HIGHS-Interface.

Abhilfe: Jeden Solver in einem **eigenen Prozess** ausführen — genau das tut das folgende Programm. Alternativ: getrennte virtuelle Umgebungen, oder auf `highspy` verzichten und HiGHS über `scipy.optimize.linprog` bzw. CVXPY ansprechen (dort ist es ohnehin als Backend verfügbar).

Der Installationstest im Vorspann umgeht die Falle bereits: Er lädt `ortools` zuerst, prüft `highspy` und `cvxpy` in der Paketübersicht nur auf Anwesenheit (`importlib.util.find_spec`) und importiert CVXPY erst im Funktionstest.

Wie die Isolation aussieht, wenn sie tragen soll

„Eigener Prozess" ist schnell gesagt. Die naheliegende Umsetzung — ein Codeschnipsel als Zeichenkette an `python -c` übergeben — funktioniert und ist trotzdem die schlechteste: Der Schnipsel ist für Editor, Linter und Testwerkzeug unsichtbar, ein Tippfehler darin fällt erst zur Laufzeit auf, und übergeben lassen sich nur Zeichenketten.

Tragfähig ist stattdessen: **jeder Solver eine gewöhnliche Funktion mit lokalem Import**, ausgeführt von einem `ProcessPoolExecutor` mit zwei Einstellungen, die zusammen die Garantie ergeben:

Einstellung	Wozu
`mp_context=multiprocessing.get_context("spawn")`	Der Kindprozess startet mit einem **frischen** Interpreter, statt den Speicher des Elternprozesses zu erben. Unter Linux ist `fork` der Standard — und damit wäre alles, was hier schon importiert ist, auch dort importiert.
`max_tasks_per_child=1`	Jede Aufgabe bekommt einen **neuen** Prozess. Ohne das verwendet der Pool seinen Arbeiter wieder, und beim zweiten Solver ist der Konflikt zurück. Genau dieser Fehler ist leicht zu machen und schwer zu finden.

> **`max_tasks_per_child=1` ist nicht optional** Ein Pool ohne diese Angabe ist der **Normalfall** — er soll seine Arbeiter ja wiederverwenden. Wer die Isolation über einen Pool herstellt und das vergisst, hat einen Prozesswechsel programmiert, aber keine Isolation gewonnen: Die zweite Aufgabe landet im selben Interpreter wie die erste. Der Absturz kommt dann nicht beim ersten Solver, sondern beim zweiten — und sieht aus wie ein Problem des zweiten.

Denselben Aufbau verwenden `Solverwechsel_CPSAT_HiGHS.py` (Kapitel 22) und `Benchmark_Skalierung.py` (Kapitel 23). Dort wandern zusätzlich **Datenobjekte** über die Prozessgrenze statt Zeichenketten — möglich, weil Domänenmodell und Lösungs-DTO keinen Solver kennen (Abschnitt 22.6).

```
#!/usr/bin/env python3

# Ein_System_Vier_Ansaetze.py
"""
Kapitel Oekosystem: Dasselbe LP in vier Bibliotheken.
   max 10*x1 + 15*x2 + 25*x3
   u.d.N. x1 + x2 + 2*x3 <= 40
          2*x1 + 3*x2 + x3 <= 50
          x >= 0

Deckt scipy.optimize, highspy, CVXPY und OR-Tools/GLOP ab, mit Kreuzvergleich
am Ende.
```

```
WICHTIG: Jeder Solver laeuft in einem EIGENEN Prozess, weil sich ortools und
highspy auf vielen Systemen nicht gemeinsam importieren lassen (beide bringen
eine eigene HiGHS-Kopie mit -> Symbolkonflikt).

Die Isolation besorgt ein ProcessPoolExecutor. Drei Einstellungen ergeben
zusammen die Garantie:

  mp_context "spawn"     Der Kindprozess startet mit einem FRISCHEN
                         Interpreter, statt den Speicher des Elternprozesses
                         zu erben. Was hier schon importiert ist, ist dort
                         nicht importiert. Mit dem Standard "fork" auf Linux
                         waere das nicht so.
  max_tasks_per_child=1  Jede Aufgabe bekommt einen NEUEN Prozess. Ohne das
                         wuerde der Pool seinen Arbeiter wiederverwenden - und
                         beim zweiten Solver waere der Konflikt zurueck.
  max_workers=1          Haelt die vier Laeufe nacheinander. Nicht aus
                         Vorsicht, sondern damit die gemessenen Zeiten
                         vergleichbar bleiben.

Jeder Solver steht in einer eigenen Funktion mit LOKALEM Import. Das ist der
Unterschied zu einem Codestring, den man an 'python -c' uebergibt: Die
Funktion laesst sich einzeln aufrufen, testen und vom Editor pruefen - ein
String nicht.

Benoetigt: scipy, highspy, cvxpy, ortools
"""

import multiprocessing
import time
from concurrent.futures import ProcessPoolExecutor

ERWARTET = 530.0            # Ergebnis der Handrechnung zum Produktionsprogramm

# Die Instanz - einmal notiert, von allen vier Funktionen benutzt.
ZIEL = [10.0, 15.0, 25.0]
MATRIX = [[1, 1, 2], [2, 3, 1]]
KAPAZITAET = [40.0, 50.0]


def loese_mit_scipy() -> tuple[float, list[float]]:
    from scipy.optimize import linprog
    ergebnis = linprog(c=[-w for w in ZIEL],          # linprog MINIMIERT -> negieren
                       A_ub=MATRIX, b_ub=KAPAZITAET,
                       bounds=[(0, None)] * 3, method="highs")
    return -ergebnis.fun, list(ergebnis.x)


def loese_mit_highspy() -> tuple[float, list[float]]:
```

```
    import highspy
    import numpy as np
    h = highspy.Highs()
    h.setOptionValue("output_flag", False)
    h.addVars(3, np.zeros(3), np.full(3, highspy.kHighsInf))
    h.changeObjectiveSense(highspy.ObjSense.kMaximize)
    for j, wert in enumerate(ZIEL):
        h.changeColCost(j, wert)
    # CSR-Format: starts[i] = Beginn von Zeile i in indices/values
    h.addRows(2, np.full(2, -highspy.kHighsInf), np.array(KAPAZITAET), 6,
              np.array([0, 3], dtype=np.int32),
              np.array([0, 1, 2, 0, 1, 2], dtype=np.int32),
              np.array([float(w) for zeile in MATRIX for w in zeile]))
    h.run()
    return (h.getInfo().objective_function_value,
            list(h.getSolution().col_value[:3]))


def loese_mit_cvxpy() -> tuple[float, list[float]]:
    import cvxpy as cp
    import numpy as np
    x = cp.Variable(3, nonneg=True)
    problem = cp.Problem(cp.Maximize(np.array(ZIEL) @ x),
                         [np.array(MATRIX) @ x <= np.array(KAPAZITAET)])
    problem.solve()
    return float(problem.value), [float(v) for v in x.value]


def loese_mit_ortools() -> tuple[float, list[float]]:
    from ortools.linear_solver import pywraplp
    s = pywraplp.Solver.CreateSolver("GLOP")
    x = [s.NumVar(0, s.infinity(), f"x{j+1}") for j in range(3)]
    for i, kapazitaet in enumerate(KAPAZITAET):
        s.Add(sum(MATRIX[i][j] * x[j] for j in range(3)) <= kapazitaet)
    s.Maximize(sum(ZIEL[j] * x[j] for j in range(3)))
    s.Solve()
    return s.Objective().Value(), [v.solution_value() for v in x]


ANSAETZE = {
    "scipy.optimize.linprog": loese_mit_scipy,
    "highspy (natives HiGHS)": loese_mit_highspy,
    "cvxpy": loese_mit_cvxpy,
    "ortools / GLOP": loese_mit_ortools,
}


if __name__ == "__main__":
```

```
    print("=" * 78)
    print("  EIN SYSTEM - VIER ANSAETZE (je eigener Prozess)")
    print("=" * 78)
    print(f"{'Bibliothek':<26} {'Z*':>10} {'x1':>7} {'x2':>7} {'x3':>7} {'Zeit':>10}")
    print("-" * 78)

    werte = []
    # Ein Pool, vier Aufgaben, vier frische Prozesse. Der Kontext muss
    # "spawn" sein - siehe Modulkommentar.
    with ProcessPoolExecutor(
            max_workers=1,
            mp_context=multiprocessing.get_context("spawn"),
            max_tasks_per_child=1) as pool:
        for name, funktion in ANSAETZE.items():
            beginn = time.perf_counter()
            try:
                wert, x = pool.submit(funktion).result(timeout=120)
            except Exception as fehler:                    # Bibliothek fehlt o. Ae.
                print(f"{name:<26} nicht verfuegbar: {str(fehler)[:40]}")
                continue
            dauer = time.perf_counter() - beginn
            werte.append(wert)
            print(f"{name:<26} {wert:>10.2f} {x[0]:>7.2f} {x[1]:>7.2f} {x[2]:>7.2f} "
                  f"{dauer:>8.2f} s")

    print("-" * 78)
    spanne = max(werte) - min(werte)
    print(f"Spannweite zwischen den Bibliotheken: {spanne:.2e}")
    print(f"Abweichung zur Handrechnung ({ERWARTET:.0f}):        "
          f"{abs(werte[0] - ERWARTET):.2e}")
    assert spanne < 1e-6, "Die Bibliotheken widersprechen sich!"
    assert abs(werte[0] - ERWARTET) < 1e-6, "Ergebnis weicht von der Handrechnung ab!"
    print("Alle Wege fuehren zum selben, von Hand bestaetigten Optimum.")
    print("(Die Zeiten enthalten Prozessstart und Import - sie messen NICHT die")
    print(" reine Solverleistung. Die Uebungsaufgabe 'Laufzeitvergleich' trennt beides.)")
    print("=" * 78)
```

Erwartete Ausgabe (Zeiten hardwareabhängig):

```
==============================================================================
  EIN SYSTEM - VIER ANSAETZE (je eigener Prozess)
==============================================================================
Bibliothek                      Z*      x1      x2      x3       Zeit
------------------------------------------------------------------------------
scipy.optimize.linprog         530.00    0.00   12.00   14.00     0.59 s
highspy (natives HiGHS)        530.00    0.00   12.00   14.00     0.12 s
cvxpy                          530.00    0.00   12.00   14.00     1.24 s
ortools / GLOP                 530.00    0.00   12.00   14.00     0.33 s
```

```
------------------------------------------------------------------------------
Spannweite zwischen den Bibliotheken: 2.41e-08
Abweichung zur Handrechnung (530):        0.00e+00
Alle Wege fuehren zum selben, von Hand bestaetigten Optimum.
(Die Zeiten enthalten Prozessstart und Import - sie messen NICHT die
 reine Solverleistung. Die Uebungsaufgabe 'Laufzeitvergleich' trennt beides.)
==============================================================================
```

 Merksatz zur Spannweite Die vier Bibliotheken stimmen **nicht auf die letzte Stelle** überein, sondern nur bis auf $2{,}4 \times 10^{-8}$. Das ist normal: Solver arbeiten mit endlicher Genauigkeit und brechen ab, sobald ihre eigene Toleranz erreicht ist. **Vergleichen Sie Solver-Ergebnisse deshalb nie mit ==**, sondern immer mit einer Toleranz — `abs(a - b) < 1e-6` oder `np.isclose()`. Wer auf exakte Gleichheit prüft, baut sich Tests, die zufällig mal bestehen und mal nicht.

 Code-Durchgang

Ansatz	Zeilen für das Modell	Charakter
`linprog`	3	Matrizen direkt übergeben. Kürzeste Variante, aber man muss selbst negieren und die Matrixform von Hand herstellen.
`highspy`	~15	Alles explizit, inklusive CSR-Format der dünnbesetzten Matrix. Aufwendig — dafür volle Kontrolle und kein Overhead beim wiederholten Lösen.
`cvxpy`	4	Liest sich wie die mathematische Formulierung. Prüft zusätzlich automatisch, ob das Problem konvex ist. Höchster Startaufwand pro Lauf (Kompilierung des Ausdrucksbaums).
`ortools`/GLOP	~6	Bedingungen einzeln mit `Add()` — gut lesbar bei wenigen, mühsam bei vielen Restriktionen.

Der wichtigste Teil des Programms sind die letzten fünf Zeilen: der Kreuzvergleich. Vier unabhängige Implementierungen, die auf 13 Nachkommastellen übereinstimmen und mit

einer Handrechnung zusammenpassen, sind ein starkes Indiz für Korrektheit. Bei einem einzelnen Solver-Ergebnis haben Sie diese Sicherheit nicht.

Was das CSR-Format bedeutet

`highspy` erwartet die Nebenbedingungsmatrix im **CSR-Format** (*Compressed Sparse Row*, komprimierte Zeilendarstellung). Statt der vollen Matrix speichert man nur die Einträge ungleich null:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 2 \\ 2 & 3 & 1 \end{pmatrix}$$

Array	Inhalt	Bedeutung
`values`	`[1, 1, 2, 2, 3, 1]`	die Zahlen selbst, zeilenweise
`indices`	`[0, 1, 2, 0, 1, 2]`	zu welcher **Spalte** gehört jeder Wert
`starts`	`[0, 3]`	Zeile 0 beginnt bei Position 0, Zeile 1 bei Position 3

Bei kleinen Modellen wirkt das umständlich. Bei realen Modellen mit 100 000 Variablen und nur 0,1 % Nicht-Null-Einträgen spart es Faktor 1000 an Speicher — und ist der Grund, warum große LPs überhaupt lösbar sind.

Typische Fehler

- **Vergessen, dass `linprog` minimiert.** Der häufigste Fehler überhaupt. Symptom: Der „optimale“ Gewinn ist erstaunlich niedrig oder null.
- **CVXPY für ein nicht-konvexes Problem verwenden.** CVXPY lehnt das ab mit `DCPError: Problem does not follow DCP rules`. Das ist ein **Feature**, keine Einschränkung: Der Fehler sagt Ihnen, dass Ihre Formulierung keine Optimalitätsgarantie hätte.
- **CP-SAT mit kontinuierlichen Variablen füttern.** CP-SAT kennt nur ganze Zahlen. Wer Euro-Beträge modelliert, rechnet in Cent (Ganzzahl) — oder nimmt einen LP-Solver.
- **Für jeden Lauf ein neues Modell bauen.** Bei 60 Backtest-Rebalancings kostet das Aufbauen mehr Zeit als das Lösen. `highspy` und CVXPY-`Parameter` erlauben es, das Modell einmal zu bauen und nur Daten zu tauschen (siehe Kapitel 19 und Kapitel 21).

3.6 Wann lohnt sich welche Ebene?

Situation	Empfehlung	Begründung
Einmaliges kleines LP, Prototyp	`scipy.optimize.linprog`	Keine zusätzliche Abhängigkeit, 3 Zeilen
Portfolio, Risiko, alles Konvexe	**CVXPY**	Lesbarkeit + automatische Konvexitätsprüfung
Dienstpläne, Zuordnung, Reihenfolge	**CP-SAT**	Globale Constraints, Konfliktlernen
Fahrzeugtouren	**OR-Tools Routing**	Fertige Metaheuristiken, jahrzehntelang optimiert
MILP mit Fixkosten, Kardinalität	`highspy` oder CP-SAT	Branch-and-Cut auf ökonomischer Struktur
50 000+ Variablen, wiederholte Läufe	`highspy` oder Pyomo	Modellaufbau wird sonst zum Engpass
Nichtkonvexes NLP	`scipy.optimize.minimize`	Bewusst mit mehreren Startpunkten arbeiten

 Merksatz Wählen Sie den Solver nach der **Struktur des Modells**, nicht nach Gewohnheit. Ein Zuweisungsproblem in CVXPY zu quälen oder ein Portfolio mit CP-SAT nachzubauen kostet Laufzeit und Nerven — und meist auch Lösungsqualität.

3.7 Modellierungsschichten für große Modelle: Pyomo und Linopy

Die vier Bibliotheken aus dem Vierfach-Vergleich decken den Alltag weitgehend ab. Sobald Modelle industrielle Größe erreichen — zehntausende Variablen, Daten aus mehreren Systemen, mehrere Jahre Lebensdauer — treten zwei weitere Werkzeuge in den Vordergrund.

Pyomo: die algebraische Denkweise

Pyomo ist im deutschsprachigen Raum der De-facto-Standard für große LP- und MILP-Modelle in Energiewirtschaft, Chemie und Logistik. Sein Kennzeichen: Es denkt in **Mengen und Indizes**, so wie die mathematische Formulierung selbst.

Der entscheidende Satz ist dieser:

```
modell.kapazitaet = pyo.Constraint(modell.R, rule=kapazitaet)
```

Das ist das $\forall i \in I$ aus Kapitel 1, unmittelbar in Code übersetzt: eine Regel, angewandt auf jedes Element einer Menge. Pyomo kann außerdem, was CVXPY und OR-Tools nicht können — **nichtlineare und gemischt-ganzzahlig-nichtlineare** Modelle (MINLP) an Solver wie Ipopt oder BONMIN übergeben (siehe Kapitel 11).

Linopy: eine Zeile, zehntausend Nebenbedingungen

Linopy verfolgt einen anderen Ansatz: Variablen sind **beschriftete Arrays** (`xarray`), keine indizierten Einzelobjekte. Damit wird aus

```
modell.add_constraints((verbrauch * x).sum("produkt") <= vorrat, name="kapazitaet")
```

nicht eine Nebenbedingung, sondern **so viele, wie die Achse `ressource` Einträge hat** — erzeugt als Matrixoperation, ohne dass je eine Python-Schleife läuft. In Energiesystem- und Netzmodellen mit den Achsen *Region* × *Technologie* × *Stunde des Jahres* ist das der Unterschied zwischen Minuten und Sekunden beim Modellaufbau.

Die Schichten im Überblick

Schicht	Denkweise	Stärke	Grenze
`scipy.optimize`	rohe Matrizen	keine Zusatzabhängigkeit, minimaler Start	Vorzeichen und Matrixform von Hand; nur LP
`highspy`	rohe Matrizen, volle Solversteuerung	schnellster wiederholter Aufbau, alle HiGHS-Optionen	CSR-Format selbst herstellen
`ortools`	Objekte, Bedingung für Bedingung	CP-SAT und Routing; sehr gut lesbar	Aufbau wird bei 10^5 Variablen zum Engpass
`cvxpy`	mathematiknahe Ausdrücke	automatische Konvexitätsprüfung, Risikomodelle	keine Ganzzahligkeit in großem Stil; Kompilierung kostet Zeit
`Pyomo`	Mengen und Indizes	Industriestandard, Daten/Modell getrennt, MINLP-fähig	mehr Zeremonie; eigene Lernkurve
`Linopy`	beschriftete Arrays (xarray)	extrem schneller Aufbau bei Millionen Nebenbedingungen	nur LP/MILP; Daten müssen zu Arrays passen

> **Merksatz** Keine dieser Schichten rechnet selbst. Alle sechs geben dasselbe Modell am Ende an dieselbe Handvoll C++-Solver weiter — hier fast immer an HiGHS. Die Wahl der Schicht entscheidet über **Ihre** Produktivität, nicht über die des Rechners.

Das folgende Programm löst mit beiden Schichten dasselbe Produktionsproblem wie der Vierfach-Vergleich und prüft das Ergebnis gegen dieselbe Handrechnung.

```
#!/usr/bin/env python3

# Modellierungsschichten.py
"""
Kapitel Oekosystem: Pyomo und Linopy - zwei Modellierungsschichten fuer grosse Modelle.
```

```
Geloest wird dasselbe Produktionsproblem wie im Vierfach-Vergleich:

    max 10*x1 + 15*x2 + 25*x3
    u.d.N.  x1 +  x2 + 2*x3 <= 40
           2*x1 + 3*x2 +  x3 <= 50
           x >= 0

Handrechnung: Z* = 530 bei x = (0, 12, 14).

Der Vergleich zeigt die beiden Denkweisen:
  * Pyomo  - algebraisch, indexbasiert, Industriestandard fuer Grossmodelle,
             trennt Modellstruktur sauber von den Daten (AbstractModel).
  * Linopy - beschriftete Arrays (xarray): eine Zeile Code erzeugt Tausende
             Nebenbedingungen auf einmal, ohne Python-Schleife.

Beide bringen KEINEN eigenen Solver mit; hier rechnet in beiden Faellen HiGHS.

WICHTIG: ortools wird in diesem Programm bewusst NICHT importiert - es
vertraegt sich nicht mit der HiGHS-Kopie, die Pyomo und Linopy laden
(siehe die Stolperfalle im Abschnitt 'Ein System - vier Programmieransaetze').

Benoetigt: pyomo, linopy, xarray, pandas, highspy, numpy
"""

from __future__ import annotations

import time

import numpy as np
import pandas as pd
import pyomo.environ as pyo
import xarray as xr
import linopy

ERWARTET = 530.0                    # Ergebnis der Handrechnung

PRODUKTE = ["Standard", "Komfort", "Premium"]
RESSOURCEN = ["Material", "Montage"]

DECKUNGSBEITRAG = np.array([10.0, 15.0, 25.0])
VERBRAUCH = np.array([[1.0, 1.0, 2.0],       # Material je Produkt
                      [2.0, 3.0, 1.0]])      # Montage je Produkt
VORRAT = np.array([40.0, 50.0])


# --- Pyomo: algebraisch und indexbasiert -----------------------------------
```

```python
def loese_mit_pyomo() -> tuple[float, list[float], float]:
    """Pyomo denkt in Mengen und Indizes, wie ein Mathematiker es aufschreibt.

    `Constraint(RESSOURCEN, rule=...)` erzeugt fuer JEDES Element der Menge
    eine Nebenbedingung - das ist das 'fuer alle i' der Formelsprache,
    unmittelbar in Code uebersetzt.
    """
    t0 = time.perf_counter()

    modell = pyo.ConcreteModel(name="Produktionsprogramm")
    modell.P = pyo.Set(initialize=PRODUKTE)
    modell.R = pyo.Set(initialize=RESSOURCEN)

    modell.db = pyo.Param(modell.P, initialize=dict(zip(PRODUKTE, DECKUNGSBEITRAG)))
    modell.a = pyo.Param(modell.R, modell.P, initialize={
        (r, p): VERBRAUCH[i, j]
        for i, r in enumerate(RESSOURCEN) for j, p in enumerate(PRODUKTE)})
    modell.vorrat = pyo.Param(modell.R, initialize=dict(zip(RESSOURCEN, VORRAT)))

    modell.x = pyo.Var(modell.P, domain=pyo.NonNegativeReals)

    modell.ziel = pyo.Objective(
        expr=sum(modell.db[p] * modell.x[p] for p in modell.P),
        sense=pyo.maximize)

    def kapazitaet(m, r):
        return sum(m.a[r, p] * m.x[p] for p in m.P) <= m.vorrat[r]

    modell.kapazitaet = pyo.Constraint(modell.R, rule=kapazitaet)

    ergebnis = pyo.SolverFactory("appsi_highs").solve(modell)
    dauer = time.perf_counter() - t0

    status = ergebnis.solver.termination_condition
    if status != pyo.TerminationCondition.optimal:
        raise RuntimeError(f"Pyomo meldet Status: {status}")

    return (float(pyo.value(modell.ziel)),
            [float(pyo.value(modell.x[p])) for p in PRODUKTE],
            dauer)


# --- Linopy: beschriftete Arrays ---------------------------------------------

def loese_mit_linopy() -> tuple[float, list[float], float]:
    """Linopy denkt in beschrifteten Arrays (xarray).

    Der entscheidende Unterschied: `(verbrauch * x).sum("produkt") <= vorrat`
```

```
    ist EINE Zeile und erzeugt so viele Nebenbedingungen, wie die Dimension
    'ressource' Eintraege hat. Bei 2 Ressourcen faellt das nicht auf, bei
    200 000 schon - dort entstehen sie als Matrixoperation statt in einer
    Python-Schleife.
    """
    t0 = time.perf_counter()

    # Benannte Indizes statt blosser Listen: Dadurch heissen die Achsen
    # 'produkt' und 'ressource', und xarray fuehrt sie beim Rechnen von allein
    # richtig zusammen. Ohne Namen vergibt linopy 'dim_0', und man muss
    # spaeter umbenennen - eine haeufige Stolperstelle.
    produkt = pd.Index(PRODUKTE, name="produkt")
    ressource = pd.Index(RESSOURCEN, name="ressource")

    modell = linopy.Model()
    modell.add_variables(lower=0, coords=[produkt], name="menge")
    x = modell.variables["menge"]

    db = xr.DataArray(DECKUNGSBEITRAG, coords=[produkt])
    verbrauch = xr.DataArray(VERBRAUCH, coords=[ressource, produkt])
    vorrat = xr.DataArray(VORRAT, coords=[ressource])

    # EINE Zeile - sie erzeugt so viele Nebenbedingungen, wie die Achse
    # 'ressource' Eintraege hat. Genau das ist der Punkt.
    modell.add_constraints((verbrauch * x).sum("produkt") <= vorrat,
                           name="kapazitaet")
    modell.add_objective((db * x).sum(), sense="max")

    modell.solve(solver_name="highs", output_flag=False)
    dauer = time.perf_counter() - t0

    if modell.termination_condition != "optimal":
        raise RuntimeError(f"Linopy meldet Status: {modell.termination_condition}")

    loesung = modell.variables["menge"].solution.to_series()
    return (float(modell.objective.value),
            [float(loesung[p]) for p in PRODUKTE],
            dauer)


if __name__ == "__main__":
    print("=" * 80)
    print("  MODELLIERUNGSSCHICHTEN FUER GROSSE MODELLE")
    print("=" * 80)
    print(f"{'Schicht':<14} {'Z*':>10} {'Standard':>10} {'Komfort':>10} "
          f"{'Premium':>10} {'Zeit':>10}")
    print("-" * 80)
```

```
    ergebnisse = []
    for name, loeser in [("Pyomo", loese_mit_pyomo), ("Linopy", loese_mit_linopy)]:
        ziel, mengen, dauer = loeser()
        ergebnisse.append(ziel)
        print(f"{name:<14} {ziel:>10.2f} {mengen[0]:>10.2f} {mengen[1]:>10.2f} "
              f"{mengen[2]:>10.2f} {dauer:>8.2f} s")

    print("-" * 80)
    for ziel in ergebnisse:
        assert abs(ziel - ERWARTET) < 1e-6, \
            f"Abweichung von der Handrechnung: {ziel} statt {ERWARTET}"
    print(f"Beide stimmen mit der Handrechnung ueberein (Z* = {ERWARTET:.0f}).")
    print()
    print("Wann welche Schicht?")
    print("  Pyomo  -> wenn Modellstruktur und Daten getrennt bleiben sollen,")
    print("            wenn nichtlineare Terme oder MINLP dazukommen koennen,")
    print("            wenn spaeter ein kommerzieller Solver angebunden wird.")
    print("  Linopy -> wenn die Daten ohnehin als beschriftete Arrays vorliegen")
    print("            (Energiesystem-, Netz- und Zeitreihenmodelle) und das")
    print("            Modell zehntausende gleichartige Nebenbedingungen hat.")
    print("=" * 80)
```

Erwartete Ausgabe (Zeiten hardwareabhängig):

```
================================================================================
  MODELLIERUNGSSCHICHTEN FUER GROSSE MODELLE
================================================================================
Schicht                Z*   Standard    Komfort    Premium       Zeit
--------------------------------------------------------------------------------
Pyomo              530.00       0.00      12.00      14.00     0.02 s
Linopy             530.00       0.00      12.00      14.00     0.27 s
--------------------------------------------------------------------------------
Beide stimmen mit der Handrechnung ueberein (Z* = 530).

Wann welche Schicht?
  Pyomo  -> wenn Modellstruktur und Daten getrennt bleiben sollen,
            wenn nichtlineare Terme oder MINLP dazukommen koennen,
            wenn spaeter ein kommerzieller Solver angebunden wird.
  Linopy -> wenn die Daten ohnehin als beschriftete Arrays vorliegen
            (Energiesystem-, Netz- und Zeitreihenmodelle) und das
            Modell zehntausende gleichartige Nebenbedingungen hat.
================================================================================
```

 Lassen Sie sich von den 0,27 s bei Linopy nicht täuschen. Bei drei Variablen misst man ausschließlich Startkosten; Linopys Stärke liegt naturgemäß dort, wo es viele gleichartige Nebenbedingungen auf einmal erzeugt. Ein Werkzeug an einem Spielzeugmodell zu bewerten ist einer der häufigsten Benchmark-Fehler — Kapitel 22 zeigt, wie man es richtig macht.

3.8 Wo die Zeit wirklich hingeht: vektorisierte Modellgenerierung

Eine der hartnäckigsten Fehlannahmen in Optimierungsprojekten lautet: *„Wenn es zu langsam ist, brauchen wir einen besseren Solver.“* Messen Sie erst — oft stimmt das nicht.

Der Grund ist strukturell. Bevor der Solver auch nur eine Iteration rechnet, muss das Modell **aufgebaut** werden: Variablen anlegen, Ausdrücke zusammensetzen, Nebenbedingungen an die C++-Schicht übergeben. Dieser Aufbau läuft in **Python**, der Solver läuft in **C++** — und zwischen beiden liegen leicht zwei Größenordnungen Geschwindigkeit.

> **Merksatz** Bei jedem Optimierungsproblem gibt es zwei Laufzeiten: die zum **Aufbauen** und die zum **Lösen**. Messen Sie beide getrennt, bevor Sie irgendetwas optimieren. Wer den kleineren Anteil beschleunigt, hat viel Arbeit für wenig Wirkung.

Vier Stufen an einem Transportproblem

Wir bauen dasselbe Transportproblem (m Werke, n Kunden, $m \cdot n$ Variablen) auf vier Arten auf:

Stufe	Wie das Modell entsteht	Was daran teuer ist
A	OR-Tools, ein `Add()` je Nebenbedingung	Je Aufruf entsteht in Python ein Ausdrucksbaum aus n Termen
B	Nebenbedingungsmatrix als COO-Tripel, in Python-Schleifen	Kein Ausdrucksbaum mehr — aber die Schleife bleibt Python
C	dieselbe Matrix über Kronecker-Produkte (NumPy/SciPy)	nichts: eine Handvoll Array-Operationen
D	Daten kommen als lange Tabelle, aufbereitet mit **Polars**	nichts: ein Spaltenausdruck statt einer Zeilenschleife

Stufe D ist der realistische Fall: Kostenmatrizen liegen in der Praxis selten als $m \times n$-Array vor, sondern als lange Tabelle `(werk, kunde, kosten)` aus Datenbank oder Data Lake. Polars berechnet daraus die Spaltenindizes der dünnbesetzten Matrix in einem einzigen Ausdruck.

```
#!/usr/bin/env python3

# Vektorisierte_Modellgenerierung.py
"""
Kapitel Oekosystem: Warum der Solver oft gar nicht der Engpass ist.

In realen Projekten geht ein grosser Teil der Rechenzeit nicht ins Loesen,
sondern ins AUFBAUEN des Modells. Dieses Programm misst das an einem
Transportproblem wachsender Groesse in vier Stufen:

    A  Modellierungsschicht, Nebenbedingung fuer Nebenbedingung (OR-Tools)
```

```
    B  Matrix direkt, aber mit Python-Schleifen ueber die Eintraege (COO)
    C  Matrix vektorisiert ueber Kronecker-Produkte (NumPy/SciPy)
    D  Daten kommen als lange Tabelle, aufbereitet mit Polars

Alle Varianten loesen dasselbe Problem und muessen denselben Zielwert
liefern - das wird am Ende geprueft.

Benoetigt: numpy, scipy, ortools; Variante D zusaetzlich polars (optional).
"""

from __future__ import annotations

import importlib.util
import time

import numpy as np
import scipy.sparse as sp
from ortools.linear_solver import pywraplp
from scipy.optimize import linprog

HAT_POLARS = importlib.util.find_spec("polars") is not None


def erzeuge_daten(m: int, n: int, saat: int = 3
                  ) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Transportproblem: m Werke, n Kunden.

    Liefert (kosten[m, n], angebot[m], bedarf[n]). Das Gesamtangebot liegt
    20 % ueber dem Gesamtbedarf, damit das Modell sicher loesbar ist.
    """
    rng = np.random.default_rng(saat)
    kosten = rng.uniform(1.0, 20.0, size=(m, n))
    bedarf = rng.uniform(10.0, 50.0, size=n)
    angebot = np.full(m, 1.2 * bedarf.sum() / m)
    return kosten, angebot, bedarf


# --- Variante A: Modellierungsschicht, Bedingung fuer Bedingung -------------

def loese_mit_modellierungsschicht(kosten: np.ndarray, angebot: np.ndarray,
                                   bedarf: np.ndarray) -> tuple[float, float, float]:
    """So schreibt man ein Transportproblem zuerst hin - gut lesbar, nah an
    der mathematischen Formulierung, jede Nebenbedingung ein eigener Aufruf.

    Jedes `s.Add(sum(...))` baut in Python einen Ausdrucksbaum aus m bzw. n
    Termen auf und uebergibt ihn einzeln an die C++-Schicht. Das ist der
    Preis der Bequemlichkeit - und er waechst linear mit der Modellgroesse.
    """
```

```
    m, n = kosten.shape

    t0 = time.perf_counter()
    s = pywraplp.Solver.CreateSolver("GLOP")
    x = [[s.NumVar(0, s.infinity(), f"x_{i}_{j}") for j in range(n)]
         for i in range(m)]
    for i in range(m):
        s.Add(sum(x[i][j] for j in range(n)) <= angebot[i])
    for j in range(n):
        s.Add(sum(x[i][j] for i in range(m)) >= bedarf[j])
    s.Minimize(sum(kosten[i][j] * x[i][j] for i in range(m) for j in range(n)))
    t_aufbau = time.perf_counter() - t0

    t0 = time.perf_counter()
    status = s.Solve()
    t_loesen = time.perf_counter() - t0
    if status != pywraplp.Solver.OPTIMAL:
        raise RuntimeError(f"Solver-Status: {status}")
    return t_aufbau, t_loesen, s.Objective().Value()


# --- Varianten B bis D: Matrix selbst bauen, dann SciPy/HiGHS ---------------

def baue_mit_schleifen(kosten: np.ndarray, angebot: np.ndarray,
                       bedarf: np.ndarray):
    """Die Nebenbedingungsmatrix als COO-Tripel (Zeile, Spalte, Wert), erzeugt
    in verschachtelten Python-Schleifen.

    Schon deutlich naeher am Blech als Variante A - es entsteht kein
    Ausdrucksbaum mehr. Die Schleife selbst bleibt aber Python.
    """
    m, n = kosten.shape
    zeilen: list[int] = []
    spalten: list[int] = []
    werte: list[float] = []
    rechte_seite: list[float] = []

    for i in range(m):                      # Angebot je Werk
        for j in range(n):
            zeilen.append(i)
            spalten.append(i * n + j)
            werte.append(1.0)
        rechte_seite.append(float(angebot[i]))

    for j in range(n):                      # Bedarf je Kunde, als -x <= -bedarf
        for i in range(m):
            zeilen.append(m + j)
            spalten.append(i * n + j)
```

```
            werte.append(-1.0)
        rechte_seite.append(-float(bedarf[j]))

    A_ub = sp.csr_matrix((werte, (zeilen, spalten)), shape=(m + n, m * n))
    return A_ub, np.array(rechte_seite), kosten.ravel()


def baue_vektorisiert(kosten: np.ndarray, angebot: np.ndarray,
                      bedarf: np.ndarray):
    """Dieselbe Matrix ohne eine einzige Schleife - ueber Kronecker-Produkte.

    Die Angebotsmatrix ist  kron(I_m, 1_n^T):  je Werk eine Zeile mit Einsen
    an genau den n Spalten dieses Werks. Die Bedarfsmatrix ist
    kron(1_m^T, I_n). Beide entstehen in je einem Aufruf und sind sofort
    duennbesetzt.
    """
    m, n = kosten.shape
    angebots_matrix = sp.kron(sp.identity(m, format="csr"), np.ones((1, n)))
    bedarfs_matrix = sp.kron(np.ones((1, m)), sp.identity(n, format="csr"))

    A_ub = sp.vstack([angebots_matrix, -bedarfs_matrix], format="csr")
    b_ub = np.concatenate([angebot, -bedarf])
    return A_ub, b_ub, kosten.ravel()


def baue_mit_polars(kosten: np.ndarray, angebot: np.ndarray, bedarf: np.ndarray):
    """Der realistische Fall: Die Kosten kommen als LANGE Tabelle
    (werk, kunde, kosten) aus Datenbank, Data Lake oder CSV-Datei.

    Polars berechnet den Spaltenindex jeder Variablen in einem einzigen
    Spaltenausdruck - ohne Python-Schleife ueber die Zeilen. Genau so baut man
    Modelle aus Millionen Tabellenzeilen.
    """
    import polars as pl

    m, n = kosten.shape
    tabelle = pl.DataFrame({
        "werk": np.repeat(np.arange(m), n),
        "kunde": np.tile(np.arange(n), m),
        "kosten": kosten.ravel(),
    }).with_columns(
        (pl.col("werk") * n + pl.col("kunde")).alias("var_index")
    )

    var_index = tabelle["var_index"].to_numpy()
    werk = tabelle["werk"].to_numpy()
    kunde = tabelle["kunde"].to_numpy()
    eins = np.ones(var_index.size)
```

```
    angebots_matrix = sp.csr_matrix((eins, (werk, var_index)), shape=(m, m * n))
    bedarfs_matrix = sp.csr_matrix((eins, (kunde, var_index)), shape=(n, m * n))

    A_ub = sp.vstack([angebots_matrix, -bedarfs_matrix], format="csr")
    b_ub = np.concatenate([angebot, -bedarf])
    return A_ub, b_ub, tabelle["kosten"].to_numpy()


def messe_matrixvariante(bauer, kosten, angebot, bedarf
                         ) -> tuple[float, float, float]:
    """Liefert (Aufbauzeit, Loesezeit, Zielwert) fuer die Varianten B bis D."""
    t0 = time.perf_counter()
    A_ub, b_ub, c = bauer(kosten, angebot, bedarf)
    t_aufbau = time.perf_counter() - t0

    t0 = time.perf_counter()
    ergebnis = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=(0, None), method="highs")
    t_loesen = time.perf_counter() - t0

    if not ergebnis.success:
        raise RuntimeError(f"Solver-Status: {ergebnis.message}")
    return t_aufbau, t_loesen, float(ergebnis.fun)


if __name__ == "__main__":
    print("=" * 88)
    print("  MODELLAUFBAU: WO DIE ZEIT WIRKLICH HINGEHT")
    print("=" * 88)
    print("Transportproblem mit m Werken und n Kunden -> m*n Variablen.\n")

    varianten: list[tuple[str, object]] = [
        ("A: OR-Tools, Add() je NB", None),        # Sonderfall, eigener Messpfad
        ("B: COO in Schleifen", baue_mit_schleifen),
        ("C: NumPy vektorisiert", baue_vektorisiert),
    ]
    if HAT_POLARS:
        varianten.append(("D: Polars-Tabelle", baue_mit_polars))
    else:
        print("Hinweis: polars nicht installiert - Variante D wird uebersprungen.\n")

    # Aufwaermlauf: Der erste Aufruf bezahlt Importe und einmalige
    # Initialisierungen. Wer den mitmisst, vergleicht Startkosten statt
    # Rechenarbeit - ein klassischer Benchmark-Fehler.
    aufwaerm = erzeuge_daten(10, 10)
    loese_mit_modellierungsschicht(*aufwaerm)
    for _, bauer in varianten[1:]:
        bauer(*aufwaerm)
```

```
    print(f"{'Groesse':<26} {'Variante':<26} {'Aufbau':>9} {'Loesen':>9} "
          f"{'Aufbauanteil':>13}")
    print("-" * 88)

    for m, n in [(40, 40), (120, 120), (250, 250)]:
        kosten, angebot, bedarf = erzeuge_daten(m, n)
        zielwerte = []
        for nummer, (name, bauer) in enumerate(varianten):
            if bauer is None:
                t_aufbau, t_loesen, ziel = loese_mit_modellierungsschicht(
                    kosten, angebot, bedarf)
            else:
                t_aufbau, t_loesen, ziel = messe_matrixvariante(
                    bauer, kosten, angebot, bedarf)
            zielwerte.append(ziel)
            anteil = 100.0 * t_aufbau / (t_aufbau + t_loesen)
            groesse = f"{m}x{n} = {m*n:,} Variablen" if nummer == 0 else ""
            print(f"{groesse:<26} {name:<26} {t_aufbau:>8.3f}s {t_loesen:>8.3f}s "
                  f"{anteil:>12.0f} %")

        # Alle Varianten muessen dasselbe Problem beschreiben.
        spanne = max(zielwerte) - min(zielwerte)
        assert spanne < 1e-6 * max(abs(z) for z in zielwerte), \
            f"Varianten widersprechen sich: {zielwerte}"
        print(f"{'':<26} {'-> Zielwert (alle gleich)':<26} {zielwerte[0]:>9.2f}"
              f"   Spanne {spanne:.1e}")
        print("-" * 88)

    print("\nZwei Lehren aus der Tabelle:")
    print("1. Bei Variante A geht mehr Zeit in den AUFBAU als ins Loesen. Wer hier")
    print("   einen schnelleren Solver kauft, beschleunigt den kleineren Teil.")
    print("2. Zwischen B und C liegt keine andere Mathematik, nur eine andere")
    print("   Schreibweise derselben Matrix - Schleife gegen Kronecker-Produkt.")
    print("\nDie Lesbarkeit von Variante A ist trotzdem viel wert: Fangen Sie dort an,")
    print("und vektorisieren Sie erst, wenn die Messung es verlangt.")
    print("=" * 88)
```

Erwartete Ausgabe (Zeiten hardwareabhängig, Verhältnisse stabil):

```
========================================================================================
  MODELLAUFBAU: WO DIE ZEIT WIRKLICH HINGEHT
========================================================================================
Transportproblem mit m Werken und n Kunden -> m*n Variablen.

Groesse                    Variante                     Aufbau    Loesen  Aufbauanteil
----------------------------------------------------------------------------------------
40x40 = 1,600 Variablen    A: OR-Tools, Add() je NB      0.031s    0.005s           87 %
```

```
                          B: COO in Schleifen           0.001s    0.009s           12 %
                          C: NumPy vektorisiert         0.001s    0.007s           14 %
                          D: Polars-Tabelle             0.001s    0.007s           12 %
                          -> Zielwert (alle gleich)    2107.32   Spanne 4.5e-13
-------------------------------------------------------------------------------------
120x120 = 14,400 Variablen A: OR-Tools, Add() je NB      0.287s    0.070s           80 %
                          B: COO in Schleifen           0.010s    0.051s           16 %
                          C: NumPy vektorisiert         0.001s    0.047s            3 %
                          D: Polars-Tabelle             0.001s    0.047s            3 %
                          -> Zielwert (alle gleich)    4693.67   Spanne 3.6e-12
-------------------------------------------------------------------------------------
250x250 = 62,500 Variablen A: OR-Tools, Add() je NB      1.284s    0.569s           69 %
                          B: COO in Schleifen           0.043s    0.215s           17 %
                          C: NumPy vektorisiert         0.003s    0.205s            1 %
                          D: Polars-Tabelle             0.003s    0.199s            1 %
                          -> Zielwert (alle gleich)    8316.24   Spanne 1.8e-12
-------------------------------------------------------------------------------------
```

Lesen Sie die letzte Spalte. Bei 62 500 Variablen verbringt Variante A **69 % der Gesamtzeit damit, das Modell überhaupt aufzuschreiben** — und nur 31 % mit Rechnen. Die vektorisierten Varianten kehren das Verhältnis um: 1 % Aufbau, 99 % Solver. Erst dort ist ein schnellerer Solver überhaupt die richtige Stellschraube.

Code-Durchgang

Stelle	Was passiert	Warum es zählt
Aufwärmlauf vor der Messung	jede Variante einmal mit 10×10 laufen lassen	Der erste Aufruf bezahlt Importe und einmalige Initialisierungen. Wer den mitmisst, vergleicht Startkosten statt Rechenarbeit — der häufigste Benchmark-Fehler überhaupt.
`sp.kron(sp.identity(m), np.ones((1, n)))`	erzeugt die Angebotsmatrix in einem Aufruf	Kronecker-Produkte sind das Werkzeug für „jede Gruppe bekommt eine Zeile“. Ein zweiter Blick lohnt: Genau dieses Muster deckt Zuordnungs-, Transport- und Schichtmodelle ab.

Stelle	Was passiert	Warum es zählt
Variante B als COO-Tripel	nicht als dichte Matrix	Wäre B eine dichte Matrix, würde sie bei 62 500 Variablen 250 MB belegen — der Vergleich wüde dann Speicher statt Schleifen messen. Fairness im Benchmark heißt: nur eine Sache verändern.
`assert spanne < 1e-6 * max(...)`	alle Varianten liefern denselben Zielwert	Ohne diese Zeile misst man womöglich die Laufzeit von vier *verschiedenen* Modellen. Ein Benchmark ohne Korrektheitsprüfung ist wertlos.
`anteil` statt absoluter Zeiten	Aufbauanteil in Prozent	Absolute Zeiten hängen von der Hardware ab, das Verhältnis nicht. Es ist die Zahl, die die Entscheidung trägt.

 Merksatz Fangen Sie mit der lesbarsten Variante an. Vektorisieren Sie erst, wenn Sie **gemessen** haben, dass der Aufbau der Engpass ist — und messen Sie Aufbau und Lösen immer getrennt. Lesbarer Code, der schnell genug ist, schlägt schnellen Code, den niemand mehr versteht.

3.9 Übungsaufgaben

Lösungen: Abschnitt A.3.

Aufgabe 3.1 — Solverwahl begründen. Welche Bibliothek würden Sie wählen? Begründen Sie mit den drei Fragen aus Abschnitt 3.4: (a) Zuteilung von 300 Prüfungen auf 40 Räume und 12 Zeitfenster. (b) Mischungsproblem: günstigstes Tierfutter aus 8 Rohstoffen, Nährwertgrenzen. (c) Portfolio aus 200 Aktien, Risiko minimieren bei Mindestrendite. (d) Wie (c), aber höchstens 15 Titel im Depot. (e) Standortwahl: Welche 5 von 40 möglichen Lagern eröffnen? (f) Kalibrierung eines Modells mit 4 Parametern an Messdaten (Fehlerquadratsumme, nicht konvex).

Aufgabe 3.2 — Modell übersetzen. Formulieren Sie das Bäckerei-Problem aus Kapitel 1 in **allen vier** Bibliotheken aus Abschnitt 3.5 und prüfen Sie mit `assert`, dass alle dasselbe Ergebnis liefern.

Aufgabe 3.3 — CSR-Format von Hand. Geben Sie `values`, `indices` und `starts` für folgende Matrix an:

$$\mathbf{A} = \begin{pmatrix} 3 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 5 & 4 & 0 & 6 \end{pmatrix}$$

Wie viele Zahlen speichert CSR, wie viele die volle Matrix?

Aufgabe 3.4 □□ — Konvexitätsprüfung erleben. Versuchen Sie, in CVXPY das Problem $\min x^3$ u. d. N. $-2 \le x \le 2$ zu lösen. Was passiert? Formulieren Sie anschließend $\min x^2$ mit denselben Schranken. Erklären Sie den Unterschied in eigenen Worten.

Aufgabe 3.5 □□□ — Laufzeitvergleich. Erzeugen Sie zufällige LPs wachsender Größe ($n = 10, 50, 100, 500, 1000$ Variablen, $m = n/2$ Nebenbedingungen) und messen Sie die Laufzeit von `linprog`, `highspy` und `cvxpy`. Stellen Sie die Ergebnisse in einer Tabelle dar. (a) Ab welcher Größe zahlt sich `highspy` aus? (b) Wie viel Zeit entfällt bei CVXPY auf den Modellaufbau, wie viel auf das eigentliche Lösen? (Tipp: `problem.solver_stats.solve_time`.)

Aufgabe 3.6 □□□ — Eigene Entscheidungshilfe. Erweitern Sie `Solver_Wahl.py` um zwei weitere Kriterien: „Ist eine kommerzielle Lizenz verfügbar?“ und „Muss das Modell für Nicht-Programmierer lesbar sein?“. Ergänzen Sie passende Empfehlungen.

3.10 Finde den Denkfehler

□ Finde den Denkfehler: Der Solver, der angeblich dreimal schneller ist

Ein Team vergleicht zwei Bibliotheken für sein Zuordnungsproblem und schreibt ins Protokoll: *„CVXPY braucht 1,9 s, SciPy nur 0,6 s — wir setzen auf SciPy.“* Gemessen wurde so:

```
import time

t0 = time.perf_counter()
import cvxpy as cp
x = cp.Variable(200, nonneg=True)
problem = cp.Problem(cp.Minimize(kosten @ x), [A @ x == b])
problem.solve()
print("CVXPY:", time.perf_counter() - t0)

t0 = time.perf_counter()
from scipy.optimize import linprog
linprog(kosten, A_eq=A, b_eq=b, bounds=(0, None), method="highs")
print("SciPy:", time.perf_counter() - t0)
```

Beide finden dieselbe Lösung. Die Zeiten stimmen — auf diesem Rechner reproduzierbar. Trotzdem trägt die Entscheidung nicht.

Ihre Aufgabe: (a) Nennen Sie **drei** Dinge, die diese Messung mitmisst, obwohl sie nichts mit der Lösegeschwindigkeit zu tun haben. (b) Warum ist gerade bei CVXPY der gemessene Wert besonders irreführend, wenn das Modell später in einer Schleife 60-mal mit wechselnden Daten gelöst wird? (c) Wie sähe eine Messung aus, die die Frage des Teams tatsächlich beantwortet?

Auflösung: Abschnitt A.3.

3.11 Micro-Quiz

Micro-Quiz 3: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Sie sollen 300 Prüfungen auf 40 Räume und 12 Zeitfenster verteilen, mit Regeln wie „diese beiden Klausuren nicht gleichzeitig“. Welches Werkzeug passt? (a) CVXPY — es prüft die Konvexität automatisch. (b) OR-Tools CP-SAT — diskrete Zuweisung mit logischen Regeln ist genau sein Gebiet. (c) `scipy.optimize.minimize` — es kann beliebige Zielfunktionen.

2. Ihr Modell mit 80 000 Variablen braucht 40 Sekunden, davon 32 Sekunden bis zum ersten Solver-Aufruf. Was ist die wirksamste Maßnahme? (a) Einen kommerziellen Solver lizenzieren. (b) Das Zeitlimit des Solvers heraufsetzen. (c) Den Modellaufbau vektorisieren — der Solver ist mit 8 Sekunden gar nicht der Engpass.

3. Was haben `scipy.optimize.linprog`, `highspy`, CVXPY (mit Standardeinstellung) und Pyomo (mit `appsi_highs`) gemeinsam? (a) Sie implementieren jeweils einen eigenen Simplex-Algorithmus in Python. (b) Sie sind Modellierungsschichten über demselben C++-Solver HiGHS — die Rechenleistung ist dieselbe, nur die Schreibweise unterscheidet sich. (c) Sie können alle sowohl konvexe als auch nicht-konvexe Probleme global lösen.

3.12 Selbsttest

Antworten: Anhang A.

1. Welche zwei Schichten hat eine moderne OR-Architektur, und wozu dient die Trennung?
2. Warum wirft CVXPY bei $\min x^3$ einen Fehler, `scipy.optimize.minimize` aber nicht?
3. Was speichert das CSR-Format, und warum ist es bei großen Modellen entscheidend?
4. Warum ist CVXPY im Laufzeitvergleich (Abschnitt 3.5) das langsamste Werkzeug — und warum ist das trotzdem kein Argument gegen seinen Einsatz?
5. Nennen Sie zwei Situationen, in denen CP-SAT die falsche Wahl wäre.

3.13 Zusammenfassung

- **Zwei Schichten:** Sie formulieren in Python, ein C++-Solver rechnet. Beides ist austauschbar.
- **Drei Fragen** genügen zur Solverwahl: diskrete Entscheidungen? lineare Zielfunktion? Routing?
- **Dasselbe Modell, vier Wege, ein Ergebnis** — nutzen Sie das als Prüfmittel: Wenn zwei Bibliotheken verschiedene Optima melden, ist eine Ihrer Formulierungen falsch.
- **CVXPYs Konvexitätsprüfung ist ein Schutzmechanismus**, keine Schikane.
- **Sechs Modellierungsschichten, eine Handvoll Solver.** Pyomo (Mengen und Indizes) und Linopy (beschriftete Arrays) ergänzen die vier Grundwerkzeuge, sobald Modelle industrielle Größe erreichen — sie rechnen aber alle mit denselben C++-Engines.

- **Messen Sie Aufbau- und Lösezeit getrennt.** Bei 62 500 Variablen entfallen in der bequemen Schreibweise rund zwei Drittel der Zeit auf den Modellaufbau. Vektorisierung mit NumPy oder Polars dreht das Verhältnis um — ein schnellerer Solver hätte es nicht getan.
- **Rechnen Sie das Ergebnis wenn möglich von Hand nach** — bei kleinen Instanzen ist das in Minuten erledigt und deckt Modellfehler auf, die kein Solver melden kann.

Ausblick. Teil II beginnt mit dem Kernverfahren des Operations Research: der linearen Programmierung. Wir bauen den Simplex-Algorithmus selbst, um zu verstehen, woher Schattenpreise kommen — und was sie wirtschaftlich bedeuten.

Kapitel 4: Vom Management-Wunsch zum Modell

Kapitel auf einen Blick

Worum geht es? Um den Schritt, der vor jedem Solver kommt und über den kaum ein Lehrbuch spricht: aus einem Satz in einer Besprechung ein Modell zu machen. Genau hier scheitern Projekte — nicht an der Mathematik.

Voraussetzungen: Kapitel 1 (die vier Bausteine eines Modells). Kein Solverwissen nötig; das ganze Kapitel kommt mit einer einzigen `linprog`-Zeile aus.

Danach können Sie: eine Anforderung daraufhin abklopfen, was sie *nicht* sagt; die Zielgröße so wählen, dass sie keinen Fehlanreiz setzt; harte von weichen Regeln unterscheiden — und begründen, warum diese Unterscheidung ein Jahr später über die Betriebsfähigkeit entscheidet.

Zeitbedarf: ca. 3 Stunden.

Programme:
`Vom_Wunsch_zum_Modell.py`

Notebook: `Notebooks_04/modellierung.ipynb`

4.1 In 5 Minuten gelöst

In 5 Minuten gelöst: „Maximieren Sie den Umsatz"

Eine Lohnfertigung hat fünf Anfragen und 60 freie Maschinenstunden. Die Anfragen verlangen zusammen 160 Stunden — es muss ausgewählt werden.

Auftrag	Stunden	Umsatz	Material	Deckungsbeitrag
A	60	60 000 €	55 000 €	5 000 €
B	30	22 000 €	12 000 €	10 000 €
C	30	20 000 €	11 000 €	9 000 €
D	25	18 000 €	9 000 €	9 000 €
E	15	12 000 €	7 000 €	5 000 €

Auftrag	Stunden	Umsatz	Material	Deckungsbeitrag

```
import numpy as np
from scipy.optimize import linprog

#            Auftrag:      A       B       C       D       E
stunden  = np.array([     60,     30,     30,     25,     15], dtype=float)
umsatz   = np.array([ 60_000, 22_000, 20_000, 18_000, 12_000], dtype=float)
material = np.array([ 55_000, 12_000, 11_000,  9_000,  7_000], dtype=float)
KAPAZITAET = 60          # Maschinenstunden; die Anfragen verlangen 160

waehle = lambda ziel: np.round(linprog(
    -ziel, A_ub=[stunden], b_ub=[KAPAZITAET],
    bounds=(0, 1), integrality=1, method="highs").x).astype(int)

for wunsch, ziel in [("Umsatz maximieren", umsatz),
                     ("Deckungsbeitrag maximieren", umsatz - material)]:
    plan = waehle(ziel)
    print(f"{wunsch:<27} -> {''.join(b for b, ja in zip('ABCDE', plan) if ja):<5}"
          f"  Umsatz {umsatz @ plan:>7,.0f}   DB {(umsatz - material) @
           ↪  plan:>6,.0f}")
```

Ausgabe:

```
Umsatz maximieren           -> A      Umsatz  60,000   DB  5,000
Deckungsbeitrag maximieren  -> BD     Umsatz  40,000   DB 19,000
```

Und jetzt der Punkt. Beide Modelle sind korrekt. Beide Pläne sind zulässig. Beide sind *optimal* — nur für verschiedene Fragen.

Wer „Umsatz maximieren" sagt, bekommt Auftrag A: 60 000 € Umsatz und 5 000 € Deckungsbeitrag. Wer „Deckungsbeitrag maximieren" sagt, bekommt B und D: 20 000 € weniger Umsatz und **14 000 € mehr Ergebnis**. Der Umsatzplan verkauft die gesamte Kapazität für eine Marge von 8 %.

Niemand in der Besprechung hätte behauptet, Umsatz sei wichtiger als Ertrag. Der Satz „maximieren Sie den Umsatz" ist trotzdem gefallen, weil Umsatz die Zahl ist, die im Monatsbericht steht.

> □ **Merksatz** Ein Modell rechnet nicht aus, was man will, sondern was man aufgeschrieben hat. Die Differenz zwischen beidem ist die eigentliche Arbeit dieses Kapitels.

Warum funktioniert das? Weil eine Zielfunktion eine **vollständige** Aussage darüber ist, was zählt. Alles, was nicht darin steht, ist dem Modell gleichgültig — nicht „weniger wichtig", sondern **wertlos**. Der Umsatzplan opfert den Deckungsbeitrag nicht aus Bosheit; er sieht ihn schlicht nicht.

4.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, warum die Zielgröße die folgenreichste Entscheidung eines OR-Projekts ist — folgenreicher als die Wahl des Solvers.
2. … die vier Fragen stellen, die aus einer Anforderung ein Modell machen.
3. … erkennen, dass ein Satz wie „Stammkunden dürfen wir nicht verlieren" mehrere Lesarten hat, und den Unterschied in Euro beziffern.
4. … begründen, wann eine Regel hart und wann sie weich sein muss — und was eine harte Regel im Betrieb anrichtet, wenn sie einmal nicht erfüllbar ist.
5. … einen Kennzahlen-Fehlanreiz benennen, bevor er in die Zielfunktion gerät.
6. … einen Satz aus einer Besprechung in den passenden Modellierungsbaustein übersetzen — und die fünf Wendungen erkennen, bei denen genau das nicht geht, ohne vorher nachzufragen.

4.3 Ein Satz ist noch kein Modell

Was aus einer Besprechung mitkommt, klingt meist so:

> *„Wir müssen die Maschinen besser auslasten. Und die Stammkunden dürfen wir nicht verlieren."*

Zwei Sätze, aus denen ein Modell werden soll. Sie enthalten: keine Entscheidungsvariable, keine messbare Zielgröße, keine Nebenbedingung. Sie enthalten nicht einmal genug, um zu erkennen, *was* entschieden werden soll.

Der Weg dorthin führt über vier Fragen. Sie sind unspektakulär, und genau deshalb werden sie übersprungen.

#	Frage	Was ohne sie passiert
1	**Worüber wird entschieden?** Welche Größen kann der Betrieb tatsächlich festlegen?	Man modelliert Dinge, die gar nicht zur Disposition stehen — oder übersieht die eine Stellschraube, um die es geht.
2	**Woran wird der Erfolg gemessen?** Welche Zahl steht am Jahresende im Bericht?	Man optimiert eine Kennzahl, die niemanden interessiert — oder eine, deren Maximierung schadet.
3	**Was ist unverhandelbar, was nur unerwünscht?**	Jede unnötig harte Regel ist eine Zeitbombe: Irgendwann ist sie nicht erfüllbar, und das System antwortet gar nicht mehr.
4	**Was passiert, wenn es nicht geht?**	Der Nachtlauf bricht ab, und niemand weiß, welche Regel schuld war.

Zur zweiten Frage

Sie ist absichtlich anders gestellt als „Was ist Ihnen wichtig?“. Auf diese Frage antwortet jeder Betrieb mit einer Liste, in der alles wichtig ist. Die brauchbare Frage lautet: **Welche Zahl steht am Jahresende im Bericht, und wer wird danach beurteilt?**

Die Antwort ist unbequem und aufschlussreich — und sie erklärt, warum in der Besprechung „Umsatz” gesagt wird, obwohl „Deckungsbeitrag” gemeint ist.

4.4 Derselbe Datensatz, fünf Modelle

Das Programm dieses Kapitels arbeitet die vier Fragen an einer Auftragsannahme ab: 14 Anfragen verlangen 800 Maschinenstunden, verfügbar sind 300.

Es ist bewusst das **einfachste Modell des ganzen Buchs** — eine Binärvariable je Auftrag, eine Nebenbedingung. Alles, was sich zwischen den fünf Varianten ändert, ist die Frage.

```
#!/usr/bin/env python3

# Vom_Wunsch_zum_Modell.py
"""
Kapitel Modellierung: Derselbe Datensatz, fuenf zulaessige Modelle, fuenf andere Plaene.

Eine Lohnfertigung kann nicht alle Anfragen annehmen: 14 Auftraege verlangen
zusammen 800 Maschinenstunden, verfuegbar sind 300. Es muss also ausgewaehlt
werden. Die Daten stehen fest - was FEHLT, ist die Frage.

Das Programm zeigt in vier Teilen, was daran haengt:

  1. Drei naheliegende Ziele aus dem Managementgespraech ("Umsatz", "Auslastung",
     "Deckungsbeitrag") - und wie weit die Plaene auseinanderlaufen.
  2. Der Satz "Stammkunden duerfen wir nicht verlieren" in drei Lesarten. Alle
     drei sind vertretbar, alle drei kosten unterschiedlich viel.
  3. Was ein Strafgewicht daraus macht - und ab wann es dasselbe tut wie eine
     harte Regel.
  4. Was die harte Lesart ein Quartal spaeter anrichtet.

Der Punkt des Kapitels: Nicht die Daten bestimmen das Modell, sondern die
Frage. Und die steht selten in der Anforderung.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
from scipy.optimize import linprog
```

```
# (Nummer, Kunde, Stammkunde?, Maschinenstunden, Umsatz, Materialkosten)
AUFTRAEGE = [
    ("A-101", "Weber GmbH",     True,   40,  28_000,  19_000),
    ("A-102", "Weber GmbH",     True,   25,  15_000,   9_500),
    ("A-103", "Kranz AG",       True,   60,  33_000,  24_000),
    ("A-104", "Kranz AG",       True,   35,  19_000,  13_500),
    ("A-105", "Obermeier KG",   True,   20,  12_000,   7_000),
    ("A-106", "Siedler & Co",   False, 120,  84_000,  71_000),
    ("A-107", "Siedler & Co",   False, 110,  74_000,  63_000),
    ("A-108", "Novatek",        False,  95,  61_000,  52_000),
    ("A-109", "Novatek",        False,  30,  31_000,  17_000),
    ("A-110", "Hellwig",        False,  15,  17_000,   8_000),
    ("A-111", "Hellwig",        False,  55,  41_000,  30_000),
    ("A-112", "Pfeiffer Werke", False,  80,  46_000,  39_000),
    ("A-113", "Obermeier KG",   True,   45,  26_000,  17_000),
    ("A-114", "Pfeiffer Werke", False,  70,  52_000,  36_000),
]
KAPAZITAET = 300          # Maschinenstunden im Planungszeitraum

STUNDEN = np.array([a[3] for a in AUFTRAEGE], dtype=float)
UMSATZ = np.array([a[4] for a in AUFTRAEGE], dtype=float)
DECKUNGSBEITRAG = np.array([a[4] - a[5] for a in AUFTRAEGE], dtype=float)
IST_STAMM = np.array([a[2] for a in AUFTRAEGE])
STAMMKUNDEN = sorted({a[1] for a in AUFTRAEGE if a[2]})


def waehle_aus(ziel: np.ndarray, zusatz_ub=None, zusatz_b_ub=None,
               zusatz_eq=None, zusatz_b_eq=None, auftraege=AUFTRAEGE,
               stunden=STUNDEN, kapazitaet=KAPAZITAET):
    """Ein Auftrag wird angenommen oder nicht - mehr Modell ist das nicht.

    Der ganze Unterschied zwischen den fuenf Varianten unten steckt in 'ziel'
    und in den Zusatzbedingungen. Das Modell selbst bleibt dasselbe.
    """
    ergebnis = linprog(
        -ziel,
        A_ub=[stunden] + (zusatz_ub or []),
        b_ub=[kapazitaet] + (zusatz_b_ub or []),
        A_eq=zusatz_eq, b_eq=zusatz_b_eq,
        bounds=(0, 1), integrality=1, method="highs")
    if not ergebnis.success:
        return None
    return np.round(ergebnis.x).astype(int)


def kennzahlen(plan, auftraege=AUFTRAEGE, stunden=STUNDEN, umsatz=UMSATZ,
               db=DECKUNGSBEITRAG, ist_stamm=IST_STAMM, kapazitaet=KAPAZITAET):
    """Immer dieselben fuenf Zahlen - nur so sind die Varianten vergleichbar."""
```

```
    abgelehnte_stammauftraege = int(((1 - plan) & ist_stamm.astype(int)).sum())
    leer_ausgegangen = sum(
        1 for kunde in sorted({a[1] for a in auftraege if a[2]})
        if not any(plan[i] for i, a in enumerate(auftraege) if a[1] == kunde))
    return {"umsatz": float(umsatz @ plan), "db": float(db @ plan),
            "auslastung": float(stunden @ plan) / kapazitaet * 100,
            "stamm_abgelehnt": abgelehnte_stammauftraege,
            "kunden_leer": leer_ausgegangen, "angenommen": int(plan.sum())}


def zeile(name: str, plan) -> None:
    if plan is None:
        print(f"  {name:<34} {'UNZULAESSIG - kein Plan moeglich':>44}")
        return
    k = kennzahlen(plan)
    print(f"  {name:<34} {k['umsatz']:>10,.0f} {k['db']:>9,.0f} "
          f"{k['auslastung']:>6.0f}% {k['stamm_abgelehnt']:>7} "
          f"{k['kunden_leer']:>6} {k['angenommen']:>7}")


def kopfzeile() -> None:
    print(f"  {'':<34} {'Umsatz':>10} {'DB':>9} {'Aus-':>7} {'Stamm-':>7} "
          f"{'Kunden':>6} {'ange-':>7}")
    print(f"  {'':<34} {'':>10} {'':>9} {'lastung':>7} {'auftr.':>7} "
          f"{'ohne':>6} {'nommen':>7}")
    print("  " + "-" * 78)


if __name__ == "__main__":
    print("=" * 84)
    print("  VOM MANAGEMENT-WUNSCH ZUM MODELL")
    print("=" * 84)
    print(f"{len(AUFTRAEGE)} Anfragen verlangen {STUNDEN.sum():.0f} Maschinenstunden, "
          f"verfuegbar sind {KAPAZITAET}.")
    print(f"Es muss also ausgewaehlt werden. {len(STAMMKUNDEN)} der Kunden sind "
          f"Stammkunden:")
    print(f"  {', '.join(STAMMKUNDEN)}\n")

    # --- 1. Drei Saetze aus dem Gespraech, drei Ziele --------------------
    print("-" * 84)
    print("1. Was im Gespraech gesagt wurde - drei naheliegende Ziele\n")
    kopfzeile()
    plaene = {}
    for name, ziel in [("\"Umsatz maximieren\"", UMSATZ),
                       ("\"Maschinen besser auslasten\"", STUNDEN),
                       ("\"Deckungsbeitrag maximieren\"", DECKUNGSBEITRAG)]:
        plaene[name] = waehle_aus(ziel)
        zeile(name, plaene[name])
```

```
umsatzplan = kennzahlen(plaene["\"Umsatz maximieren\""])
auslastungsplan = kennzahlen(plaene["\"Maschinen besser auslasten\""])
dbplan = kennzahlen(plaene["\"Deckungsbeitrag maximieren\""])

mehr_umsatz = umsatzplan["umsatz"] - dbplan["umsatz"]
weniger_db = dbplan["db"] - umsatzplan["db"]
print(f"\n  Der Umsatzplan bringt {mehr_umsatz:,.0f} EUR mehr Umsatz und "
      f"{weniger_db:,.0f} EUR WENIGER")
print(f"  Deckungsbeitrag. Fuer {mehr_umsatz:,.0f} EUR Umsatz werden also "
      f"{weniger_db:,.0f} EUR Ergebnis")
print(f"  aufgegeben - und {umsatzplan['stamm_abgelehnt']} von "
      f"{int(IST_STAMM.sum())} Stammkundenauftraegen dazu.")
print()
print("  \"Auslastung\" ist der haeufigste Wunsch und der schaedlichste. Der")
print(f"  darauf optimierte Plan bringt {dbplan['db'] - auslastungsplan['db']:,.0f} EUR
 ↪  weniger "
      f"Deckungsbeitrag und")
print(f"  {dbplan['umsatz'] - auslastungsplan['umsatz']:,.0f} EUR weniger Umsatz als der "
      f"DB-Plan - und erreicht dabei")
print(f"  nicht einmal mehr Auslastung: alle drei liegen bei "
      f"{dbplan['auslastung']:.0f} %.")
print("  Eine volle Maschine ist eben kein Ziel, sondern ein Nebenprodukt.")

# --- 2. Ein Satz, drei Lesarten ------------------------------------
print("\n" + "-" * 84)
print("2. \"Stammkunden duerfen wir nicht verlieren\" - aber was heisst das?\n")
kopfzeile()
zeile("ohne die Regel (DB maximieren)", waehle_aus(DECKUNGSBEITRAG))

# Lesart A: jeder Stammkundenauftrag wird angenommen
alle_stamm = [[1.0 if IST_STAMM[i] else 0.0 for i in range(len(AUFTRAEGE))]]
lesart_a = waehle_aus(DECKUNGSBEITRAG, zusatz_eq=alle_stamm,
                      zusatz_b_eq=[float(IST_STAMM.sum())])
zeile("A: jeden Stammauftrag annehmen", lesart_a)

# Lesart B: kein Stammkunde geht leer aus (mindestens ein Auftrag je Kunde)
je_kunde_ub, je_kunde_b = [], []
for kunde in STAMMKUNDEN:
    je_kunde_ub.append([-1.0 if a[1] == kunde else 0.0 for a in AUFTRAEGE])
    je_kunde_b.append(-1.0)
lesart_b = waehle_aus(DECKUNGSBEITRAG, zusatz_ub=je_kunde_ub,
                      zusatz_b_ub=je_kunde_b)
zeile("B: kein Stammkunde geht leer aus", lesart_b)

a, b = kennzahlen(lesart_a), kennzahlen(lesart_b)
ohne = kennzahlen(waehle_aus(DECKUNGSBEITRAG))
print("\n  Beide Lesarten erreichen dasselbe: kein Stammkunde geht leer aus.")
```

```
    print(f"  Lesart A kostet dafuer {ohne['db'] - a['db']:,.0f} EUR Deckungsbeitrag, "
          f"Lesart B nur {ohne['db'] - b['db']:,.0f} EUR -")
    print(f"  ein Unterschied von {b['db'] - a['db']:,.0f} EUR fuer denselben Satz aus
     ↪  derselben")
    print("  Besprechung. Wer nicht nachfragt, entscheidet das selbst, ohne es")
    print("  zu merken.")

    # --- 3. Die weiche Variante ------------------------------------------
    print("\n" + "-" * 84)
    print("3. Dieselbe Regel als Strafkosten je abgelehntem Stammauftrag\n")
    kopfzeile()
    for strafe in (3_000, 6_000, 10_000):
        ziel = DECKUNGSBEITRAG + np.where(IST_STAMM, float(strafe), 0.0)
        zeile(f"C: Strafe {strafe:>6,} EUR je Auftrag", waehle_aus(ziel))

    print("\n  Bei 3.000 EUR aendert sich nichts: Die Strafe liegt unter dem, was")
    print("  die Auftraege an Deckungsbeitrag verdraengen wuerden. Bei 10.000 EUR")
    print("  kommt genau Lesart A heraus. Eine harte Regel ist also nichts")
    print("  anderes als eine weiche mit hinreichend grosser Strafe - was ein")
    print("  Strafgewicht sonst noch bedeutet, steht im Kapitel CP-SAT.")

    # --- 4. Was die harte Lesart ein Quartal spaeter anrichtet -----------
    print("\n" + "-" * 84)
    print("4. Ein Quartal spaeter: Weber fragt einen Grossauftrag an\n")
    erweitert = AUFTRAEGE + [("A-115", "Weber GmbH", True, 320, 210_000, 150_000)]
    stunden_neu = np.array([a[3] for a in erweitert], dtype=float)
    db_neu = np.array([a[4] - a[5] for a in erweitert], dtype=float)
    stamm_neu = np.array([a[2] for a in erweitert])

    print(f"  A-115: Weber GmbH, {erweitert[-1][3]} Maschinenstunden - allein mehr,")
    print(f"  als der ganze Zeitraum hergibt ({KAPAZITAET} h).\n")

    alle_stamm_neu = [[1.0 if stamm_neu[i] else 0.0 for i in range(len(erweitert))]]
    hart = waehle_aus(db_neu, zusatz_eq=alle_stamm_neu,
                      zusatz_b_eq=[float(stamm_neu.sum())],
                      auftraege=erweitert, stunden=stunden_neu)
    weich = waehle_aus(db_neu + np.where(stamm_neu, 10_000.0, 0.0),
                       auftraege=erweitert, stunden=stunden_neu)

    print(f"  Lesart A (hart):  "
          f"{'UNZULAESSIG - der Nachtlauf bricht ab' if hart is None else 'loesbar'}")
    if weich is not None:
        k = kennzahlen(weich, erweitert, stunden_neu,
                       np.array([a[4] for a in erweitert], dtype=float),
                       db_neu, stamm_neu)
        print(f"  Lesart C (weich): loesbar - {k['db']:,.0f} EUR Deckungsbeitrag, "
              f"{k['stamm_abgelehnt']} Stammauftrag abgelehnt")
```

```
    print("\n" + "=" * 84)
    print("  WORAUF ES ANKOMMT")
    print("=" * 84)
    print("Die Daten waren in allen vier Teilen dieselben. Was sich geaendert hat,")
    print("war nur die Frage - und mit ihr der Plan, der Deckungsbeitrag und die")
    print("Kundenbeziehung.")
    print()
    print("Drei Dinge, die deshalb vor dem ersten Modell geklaert gehoeren:")
    print()
    print("  1. WORAN wird der Erfolg gemessen? Nicht 'was ist Ihnen wichtig',")
    print("     sondern: Welche Zahl steht am Jahresende im Bericht? Umsatz,")
    print("     Auslastung und Deckungsbeitrag fuehren hier zu drei")
    print("     verschiedenen Plaenen.")
    print("  2. WAS GENAU heisst der Satz? 'Stammkunden nicht verlieren' hat")
    print(f"     mindestens drei Lesarten mit {b['db'] - a['db']:,.0f} EUR Unterschied. Die
    ↪  Frage")
    print("     'meinen Sie jeden Auftrag oder jeden Kunden?' dauert zehn")
    print("     Sekunden.")
    print("  3. WAS PASSIERT, WENN ES NICHT GEHT? Eine harte Regel, die niemand")
    print("     hinterfragt hat, macht das System eines Tages unloesbar - und")
    print("     zwar genau dann, wenn ein besonders grosser Auftrag hereinkommt.")
    print("     Weiche Regeln antworten auch dann noch.")
    print("=" * 84)
```

Erwartete Ausgabe:

```
====================================================================================
  VOM MANAGEMENT-WUNSCH ZUM MODELL
====================================================================================
14 Anfragen verlangen 800 Maschinenstunden, verfuegbar sind 300.
Es muss also ausgewaehlt werden. 3 der Kunden sind Stammkunden:
  Kranz AG, Obermeier KG, Weber GmbH

------------------------------------------------------------------------------------
1. Was im Gespraech gesagt wurde - drei naheliegende Ziele

                                      Umsatz        DB    Aus-  Stamm- Kunden   ange-
                                                          lastung  auftr.   ohne  nommen
  ------------------------------------------------------------------------------
  "Umsatz maximieren"                227,000    66,000    100%       5      2       6
  "Maschinen besser auslasten"       205,000    54,000    100%       4      1       5
  "Deckungsbeitrag maximieren"       222,000    78,500    100%       2      1       8

  Der Umsatzplan bringt 5,000 EUR mehr Umsatz und 12,500 EUR WENIGER
  Deckungsbeitrag. Fuer 5,000 EUR Umsatz werden also 12,500 EUR Ergebnis
  aufgegeben - und 5 von 6 Stammkundenauftraegen dazu.

  "Auslastung" ist der haeufigste Wunsch und der schaedlichste. Der
```

```
  darauf optimierte Plan bringt 24,500 EUR weniger Deckungsbeitrag und
  17,000 EUR weniger Umsatz als der DB-Plan - und erreicht dabei
  nicht einmal mehr Auslastung: alle drei liegen bei 100 %.
  Eine volle Maschine ist eben kein Ziel, sondern ein Nebenprodukt.

--------------------------------------------------------------------------------
2. "Stammkunden duerfen wir nicht verlieren" - aber was heisst das?

                                      Umsatz        DB    Aus-  Stamm- Kunden   ange-
                                                          lastung  auftr.   ohne  nommen
  ------------------------------------------------------------------------------
  ohne die Regel (DB maximieren)      222,000    78,500    100%       2      1       8
  A: jeden Stammauftrag annehmen      181,000    66,000     90%       0      0       8
  B: kein Stammkunde geht leer aus    213,000    75,000     98%       2      0       8

  Beide Lesarten erreichen dasselbe: kein Stammkunde geht leer aus.
  Lesart A kostet dafuer 12,500 EUR Deckungsbeitrag, Lesart B nur 3,500 EUR -
  ein Unterschied von 9,000 EUR fuer denselben Satz aus derselben
  Besprechung. Wer nicht nachfragt, entscheidet das selbst, ohne es
  zu merken.

--------------------------------------------------------------------------------
3. Dieselbe Regel als Strafkosten je abgelehntem Stammauftrag

                                      Umsatz        DB    Aus-  Stamm- Kunden   ange-
                                                          lastung  auftr.   ohne  nommen
  ------------------------------------------------------------------------------
  C: Strafe  3,000 EUR je Auftrag     222,000    78,500    100%       2      1       8
  C: Strafe  6,000 EUR je Auftrag     205,000    73,000    100%       1      0       8
  C: Strafe 10,000 EUR je Auftrag     181,000    66,000     90%       0      0       8

  Bei 3.000 EUR aendert sich nichts: Die Strafe liegt unter dem, was
  die Auftraege an Deckungsbeitrag verdraengen wuerden. Bei 10.000 EUR
  kommt genau Lesart A heraus. Eine harte Regel ist also nichts
  anderes als eine weiche mit hinreichend grosser Strafe - was ein
  Strafgewicht sonst noch bedeutet, steht im Kapitel CP-SAT.

--------------------------------------------------------------------------------
4. Ein Quartal spaeter: Weber fragt einen Grossauftrag an

  A-115: Weber GmbH, 320 Maschinenstunden - allein mehr,
  als der ganze Zeitraum hergibt (300 h).

  Lesart A (hart):  UNZULAESSIG - der Nachtlauf bricht ab
  Lesart C (weich): loesbar - 66,000 EUR Deckungsbeitrag, 1 Stammauftrag abgelehnt

================================================================================
  WORAUF ES ANKOMMT
================================================================================
```

```
Die Daten waren in allen vier Teilen dieselben. Was sich geaendert hat,
war nur die Frage - und mit ihr der Plan, der Deckungsbeitrag und die
Kundenbeziehung.

Drei Dinge, die deshalb vor dem ersten Modell geklaert gehoeren:

  1. WORAN wird der Erfolg gemessen? Nicht 'was ist Ihnen wichtig',
     sondern: Welche Zahl steht am Jahresende im Bericht? Umsatz,
     Auslastung und Deckungsbeitrag fuehren hier zu drei
     verschiedenen Plaenen.
  2. WAS GENAU heisst der Satz? 'Stammkunden nicht verlieren' hat
     mindestens drei Lesarten mit 9,000 EUR Unterschied. Die Frage
     'meinen Sie jeden Auftrag oder jeden Kunden?' dauert zehn
     Sekunden.
  3. WAS PASSIERT, WENN ES NICHT GEHT? Eine harte Regel, die niemand
     hinterfragt hat, macht das System eines Tages unloesbar - und
     zwar genau dann, wenn ein besonders grosser Auftrag hereinkommt.
     Weiche Regeln antworten auch dann noch.
================================================================================
```

Teil 1: Drei Ziele, drei Pläne

Zielgröße	Umsatz	Deckungsbeitrag	Auslastung	abgelehnte Stammaufträge
„Umsatz maximieren”	**227 000 €**	66 000 €	100 %	5 von 6
„Maschinen besser auslasten”	205 000 €	54 000 €	100 %	4 von 6
„Deckungsbeitrag maximieren”	222 000 €	**78 500 €**	100 %	2 von 6

Zwei Beobachtungen, die im Gespräch niemand vorhergesagt hätte:

Der Umsatzplan kostet 12 500 € Ergebnis, um 5 000 € Umsatz zu gewinnen — und nimmt dabei fünf von sechs Stammkundenaufträgen aus dem Plan. Das Wechselkursverhältnis ist 1 : 2,5 gegen den Betrieb.

Der Auslastungsplan ist in jeder Hinsicht der schlechteste — auch bei der Auslastung. Alle drei Pläne belegen die Maschinen zu 100 %; der eigens darauf optimierte Plan erreicht nur nicht mehr, sondern verliert dabei 24 500 € Deckungsbeitrag. Der Grund ist einfach: Bei knapper Kapazität ist eine volle Maschine keine Leistung, sondern die Voreinstellung. Zu optimieren gibt es nur, **womit** sie voll wird.

> **Auslastung ist ein Ergebnis, kein Ziel**
>
> „Wir müssen die Maschinen besser auslasten” ist der häufigste Satz in Produktionsbesprechungen und fast immer die falsche Zielgröße. Auslastung misst, wie viel Zeit vergeht — nicht, was dabei herauskommt. Wer sie maximiert, belohnt lange, schlecht bezahlte Aufträge.

Die Ausnahme, die die Regel erklärt: Wenn Maschinen tatsächlich leer stehen, *weil* zu wenig hereinkommt, ist Auslastung ein sinnvolles **Symptom** — aber dann liegt das Problem im Vertrieb und nicht in der Planung.

Teil 2: Ein Satz, drei Lesarten

„Stammkunden dürfen wir nicht verlieren." Jeder im Raum nickt. Niemand merkt, dass drei verschiedene Modelle gemeint sein können:

Lesart	Deckungsbeitrag	kostet gegenüber „ohne Regel"
ohne die Regel	78 500 €	—
A: jeden Stammkundenauftrag annehmen	66 000 €	12 500 €
B: kein Stammkunde geht leer aus	75 000 €	3 500 €

Beide Lesarten erreichen dasselbe Ziel — kein Stammkunde geht leer aus. Lesart A kostet dafür das Dreieinhalbfache. Der Unterschied von 9 000 € entsteht durch eine Frage, die zehn Sekunden dauert:

„Meinen Sie jeden Auftrag oder jeden Kunden?"

Wer sie nicht stellt, entscheidet sie trotzdem — nur unbewusst, beim Tippen.

Teil 3: Hart oder weich

Dieselbe Regel als Strafkosten je abgelehntem Stammkundenauftrag:

Strafe je Auftrag	Deckungsbeitrag	abgelehnte Stammaufträge
3 000 €	78 500 €	2
6 000 €	73 000 €	1
10 000 €	66 000 €	0

Bei 3 000 € ändert sich **nichts**: Die Strafe liegt unter dem Deckungsbeitrag, den die Stammkundenaufträge verdrängen würden. Bei 10 000 € kommt genau Lesart A heraus.

Daraus folgt eine Einsicht, die das Modellieren erleichtert: **Eine harte Regel ist eine weiche mit hinreichend großer Strafe.** Man kann also immer weich anfangen und die Strafe hochdrehen, bis die Regel greift — und sieht dabei, was sie kostet. Was ein Strafgewicht darüber hinaus bedeutet und warum zwei Gewichte einen *Wechselkurs* festlegen und keine Wichtigkeiten, steht in Abschnitt 7.10.

Teil 4: Ein Quartal später

Die harte Lesart A wurde eingebaut, alles lief drei Monate lang. Dann fragt Weber einen Auftrag über 320 Maschinenstunden an — mehr, als der ganze Zeitraum hergibt.

```
Lesart A (hart):  UNZULAESSIG - der Nachtlauf bricht ab
Lesart C (weich): loesbar - 66,000 EUR Deckungsbeitrag, 1 Stammauftrag abgelehnt
```

Das ist kein konstruierter Sonderfall, sondern der Normalfall: **Harte Regeln scheitern nicht bei ihrer Einführung, sondern Monate später** — und zwar bevorzugt dann, wenn etwas Ungewöhnliches passiert, also genau dann, wenn man die Planung am dringendsten braucht.

Die weiche Variante antwortet auch dann noch. Sie lehnt den Großauftrag ab, weist das im Bericht aus, und ein Mensch entscheidet. Das ist der ganze Unterschied — und er wird beim Schreiben der Anforderung festgelegt, nicht beim Programmieren.

> **Merksatz** Hart ist nur, was rechtlich oder physikalisch unmöglich ist. Alles andere wird weich mit hoher Strafe. Dann liefert der Solver den am wenigsten schlechten Plan statt einer Fehlermeldung — und die Fehlermeldung ist das, was niemand gebrauchen kann.

4.5 Der Gesprächsleitfaden

Was aus den vier Teilen als Handwerkszeug bleibt — Fragen, die man in der ersten Besprechung stellt, bevor eine Zeile Code entsteht:

> **Zwölf Fragen für das erste Gespräch**
>
> **Zur Entscheidung** 1. Was genau wird entschieden — und von wem heute? 2. Was steht dabei *nicht* zur Disposition? 3. Wie oft wird entschieden, und wie viel Zeit ist dafür?
>
> **Zum Ziel** 4. Welche Zahl steht am Jahresende im Bericht? 5. Wenn genau diese Zahl maximiert würde — welcher Plan käme heraus, und wäre er akzeptabel? 6. Gibt es eine zweite Zahl, die auch stimmen muss? Was ist Ihnen ein Prozent der ersten wert, um ein Prozent der zweiten zu gewinnen?
>
> **Zu den Regeln** 7. Welche dieser Regeln ist gesetzlich oder physikalisch zwingend? 8. Welche würden Sie in einer Ausnahmesituation brechen — und was wäre eine Ausnahmesituation? 9. Bei „darf nicht": Betrifft das jeden einzelnen Fall oder das Gesamtbild?
>
> **Zum Betrieb** 10. Was soll passieren, wenn es keine zulässige Lösung gibt? 11. Wer sieht das Ergebnis, und was muss er daran erkennen können? 12. Woran würden Sie in einem halben Jahr merken, dass das System nicht mehr das Richtige tut?
>
> Frage 5 ist die wichtigste. Sie verwandelt eine abstrakte Zielangabe in einen konkreten Plan, den man ablehnen kann — und Ablehnung ist die einzige Form von Anforderungsklärung, die zuverlässig funktioniert.

4.6 Deutsch — OR: ein Übersetzungslexikon

Die zwölf Fragen liefern Sätze. Sätze sind noch kein Modell — aber die meisten von ihnen haben eine **Standardübersetzung**, und wer sie kennt, spart sich das Nachdenken für die Fälle, in denen es keine gibt.

Die Tabelle liest sich von links nach rechts: Was in der Besprechung **gesagt** wird, was es **bedeutet**, und welcher **Baustein** daraus wird. Die B-Nummern verweisen auf Anhang B, wo jedes Muster mit Formulierung und Code steht.

Was gesagt wird	Was gemeint ist	Baustein
„Wir haben nur 400 Stunden.“	eine Obergrenze, die nicht überschritten werden darf	harte Kapazitätsgrenze $\leq b$ (Kapitel 5)
„Mehr als zwei Millionen gibt es nicht.“	dasselbe, auf Geld	Budgetlimit (B27)
„Möglichst nicht über 40 Stunden.“	eine Grenze, die im Notfall gebrochen werden darf	weiche Grenze mit Strafkosten (B8)
„Wir können nur ganze Kisten liefern.“	die Menge ist nicht teilbar	Ganzzahligkeitsbedingung (Kapitel 6)
„Wenn wir die Anlage nutzen, fallen Fixkosten an.“	Kosten, die erst ab der ersten Einheit anfallen	Aktivierungsschalter (B1)
„Entweder gar nicht oder mindestens 50.“	kein Betrieb im Kleinen	semikontinuierliche Variable (B2)
„Wenn wir A bauen, brauchen wir auch B.“	eine Folgeverpflichtung	Implikation (B3)
„Höchstens drei Positionen im Depot.“	eine Obergrenze auf die **Anzahl**, nicht die Menge	Kardinalität (B6)
„Der Anteil muss mindestens 30 % betragen.“	eine Bedingung auf ein **Verhältnis**	Verhältnis-Bedingung (B12)
„Jeder Auftrag genau einem Mitarbeiter.“	eine vollständige, eindeutige Zuordnung	Zuordnung 1:1 (B23)
„Eine Maschine kann nur eines gleichzeitig.“	zeitliche Ausschließlichkeit	Nichtüberlappung (B14)
„B kann erst nach A.“	eine Reihenfolge	Vorrangbeziehung (B13)
„Höchstens fünf Tage am Stück.“	eine Bedingung über ein **gleitendes Fenster**	gleitendes Fenster (B16)
„Umstellen kostet uns eine halbe Stunde.“	reihenfolgeabhängige Kosten	Umrüstkosten (B17), Rüstzeit als Kapazität (B26)
„Was reinkommt, muss auch wieder raus.“	Erhaltung an jedem Knoten	Flusserhaltung (B22)
„Notfalls brechen wir die Regel.“	die Bedingung ist weich, aber teuer	Schlupfvariablen (B18)

Was gesagt wird	Was gemeint ist	Baustein
„Erst Termintreue, dann Kosten.”	eine Rangfolge zwischen Zielen	hierarchische Ziele (B19)
„Es soll gerecht zugehen.”	den Schlechtestgestellten verbessern	Min/Max in der Zielfunktion (B11)
„Was brächte uns eine zusätzliche Stunde?”	die Frage nach dem Grenzwert einer Ressource	Schattenpreis (Kapitel 5, Abschnitt 22.4)
„Mit 95 % Sicherheit muss es halten.”	eine Zusage auf eine **Quote**, nicht auf einen Mittelwert	Chance Constraint (Abschnitt 12.7)
„Wir sichern uns gegen Schätzfehler ab.”	gegen die ungünstigste Parameterlage planen	Worst-Case-Abzug (B21)

Fünf Wendungen, bei denen die Übersetzung eine Entscheidung erzwingt

Die Tabelle oben tut so, als sei die Zuordnung eindeutig. Bei den folgenden fünf ist sie es nicht — und **das Nachfragen ist die eigentliche Modellierungsarbeit**:

Wendung	Die Frage, die dahinter steckt
„möglichst”	Hart oder weich? Und wenn weich: Was darf ein Verstoß kosten? Ohne Zahl ist „möglichst” keine Bedingung, sondern eine Stimmung.
„nicht mehr als drei pro Woche”	Je Person oder über alle zusammen? Kalenderwoche oder gleitende sieben Tage? Beide Lesarten sind vertretbar und ergeben verschiedene Modelle.
„im Durchschnitt”	Fast immer eine Falle. Der Durchschnittskunde kauft nie — siehe Abschnitt 12.3. Gemeint ist meist ein Quantil („in 9 von 10 Fällen”).
„so schnell wie möglich”	Welche Zahl? Ende des letzten Auftrags, Summe aller Fertigstellungszeiten oder größte Verspätung? Drei Zielfunktionen, drei verschiedene Pläne.
„fair”	Den Schlechtesten verbessern, die Spannweite verkleinern oder gleiche Anteile erzwingen? Erst die Wahl macht daraus ein Modell — und sie gehört dem Betrieb, nicht dem Modellierer.

 Der Nutzen dieser Liste Sie ersetzt kein Nachdenken. Sie sorgt dafür, dass das Nachdenken an den **richtigen** vier oder fünf Stellen stattfindet, statt an allen zwanzig gleichzeitig.

Und die Gegenrichtung: OR — Deutsch

Dieselbe Barriere gibt es rückwärts. Diese Sätze haben sich bewährt, wenn ein Ergebnis jemandem erklärt werden muss, der das Modell nicht kennt:

Fachbegriff	Was Sie in der Besprechung sagen
Schattenpreis / Dualwert	„Die nächste Stunde auf dieser Maschine wäre uns 9 € wert.”
bindende Bedingung mit Dualwert 0	„Die Regel berührt den Plan, kostet uns aber nichts — dort zu verhandeln bringt nichts.”
`INFEASIBLE`	„Ihre Regeln widersprechen sich.” **Nicht:** „Der Computer schafft es nicht.”
Optimalitätslücke von 2 %	„Besser als das hier geht es höchstens um 2 % — und die liegen unter unserer Datenungenauigkeit.”
CVaR	„Wenn es schiefgeht, verlieren wir im Mittel so viel.” (Der VaR sagt nur, **ob** es schiefgeht.)
EVPI	„Selbst eine perfekte Prognose wäre uns höchstens so viel wert.”
Relaxation	„Wir haben die Regel probeweise gelockert, um zu sehen, was sie kostet.”
Chance Constraint	„Der Plan hält in 95 von 100 Fällen — und der nächste Prozentpunkt kostet …”

4.7 Übungsaufgaben

Lösungen: Abschnitt A.4.

Aufgabe 4.1 □ **— Die fehlende Lesart.** Der Text nennt zwei Lesarten von „Stammkunden dürfen wir nicht verlieren”. Formulieren Sie eine dritte, die weder A noch B ist, und sagen Sie, welchen Betrieb sie beschreiben würde.

Aufgabe 4.2 □ **— Frage 5 anwenden.** Ein Krankenhaus möchte „die Wartezeiten in der Notaufnahme senken”. Wenden Sie Frage 5 an: Welcher Plan käme heraus, wenn genau die durchschnittliche Wartezeit minimiert würde — und was wäre daran nicht akzeptabel?

Aufgabe 4.3 □□ **— Der Kompromiss dazwischen.** Zwischen Lesart A (12 500 €) und Lesart B (3 500 €) liegt Spielraum. Bauen Sie eine Variante, in der jeder Stammkunde **mindestens die Hälfte** seiner Auftragsstunden zugeteilt bekommt. Was kostet sie?

Aufgabe 4.4 □□ **— Zwei Ziele gleichzeitig.** Maximieren Sie DB $+ \lambda \cdot$ Umsatz für $\lambda \in \{0; 0{,}1; 0{,}3; 1\}$ und tragen Sie die Ergebnisse als Tabelle auf. Ab welchem λ kippt der Plan von der DB- in die Umsatzlösung? Was sagt dieser Wert dem Betrieb?

Aufgabe 4.5 □□ **— Übersetzen, wo es nicht eindeutig ist.** Übersetzen Sie mit Abschnitt 4.6: (a) „Die Monteure sollen möglichst gleichmäßig ausgelastet sein.” Nennen Sie **drei** Zielfunktionen, die alle

„gleichmäßig" bedeuten können, und je einen Fall, in dem die Wahl den Plan sichtbar ändert. (b) „Kein Kunde darf länger als drei Tage warten." Formulieren Sie die Bedingung einmal hart und einmal weich. Welche Frage müssen Sie dem Betrieb stellen, um zu entscheiden? (c) Suchen Sie in einer echten Anforderung aus Ihrem Umfeld einen Satz, der nach der Tabelle eindeutig aussieht, es aber nicht ist.

Aufgabe 4.6 □□□ — Die Anforderung schreiben. Formulieren Sie für den Fall dieses Kapitels eine vollständige, prüfbare Anforderung: Ziel, harte Regeln, weiche Regeln mit Strafgewichten, Verhalten bei Unlösbarkeit, auszuweisende Kennzahlen. Zwei Seiten genügen — sie sind mehr wert als zwei Wochen Programmierung.

4.8 Finde den Denkfehler

□ „Die Engpassmaschine soll zu mindestens 92 % ausgelastet sein"

Der Werkleiter formuliert eine klare, messbare, prüfbare Anforderung:

> *„Die Engpassmaschine soll in jedem Planungszeitraum zu mindestens 92 % ausgelastet sein. > Das ist unsere teuerste Anlage, Leerlauf können wir uns nicht leisten."*

Der Entwickler baut sie als harte Nebenbedingung ein — sie ist ja klar formuliert. Die Abnahme läuft mit den Daten des letzten Quartals: Der Plan hält die Vorgabe ein, der Deckungsbeitrag stimmt, alle Tests grün. Das System geht in Betrieb.

Diese Anforderung enthält zwei verschiedene Fehler. Welche?

Ein Hinweis für den zweiten: Sehen Sie sich an, was das Modell im Abnahmetest *tatsächlich* geprüft hat — und rechnen Sie einen Monat durch, in dem weniger Anfragen hereinkommen.

4.9 Micro-Quiz

□ Drei Fragen

1. Warum liefert „Umsatz maximieren" hier einen schlechteren Plan als „Deckungsbeitrag maximieren", obwohl beide Modelle korrekt sind? a) Weil der Solver bei Umsatzzielen ungenauer rechnet. b) Weil alles, was nicht in der Zielfunktion steht, für das Modell wertlos ist — nicht weniger wichtig. c) Weil Umsatzdaten in der Praxis unzuverlässiger sind als Kostendaten.

2. Eine Regel als harte Nebenbedingung statt als Strafkosten zu formulieren, ist vor allem deshalb riskant, weil … a) … harte Nebenbedingungen den Solver verlangsamen. b) … das Modell unlösbar wird, sobald die Regel einmal nicht erfüllbar ist — und dann gar keine Antwort mehr liefert. c) … sich harte Regeln nachträglich nicht mehr ändern lassen.

3. Alle drei Zielgrößen erreichen 100 % Auslastung. Was folgt daraus? a) Die Auslastung war schon vorher optimal, das Projekt ist überflüssig. b) Bei knapper Kapazität ist volle

Auslastung die Voreinstellung; zu entscheiden ist nur, *womit* die Maschine voll wird. c) Die Kapazitätsdaten sind vermutlich falsch erfasst.

4.10 Selbsttest

1. Warum ist „Was ist Ihnen wichtig?" eine schlechte Frage und „Welche Zahl steht im Jahresbericht?" eine gute?
2. Nennen Sie zwei Kennzahlen aus Ihrem eigenen Arbeitsumfeld, deren Maximierung schaden würde.
3. Erklären Sie in einem Satz, warum eine harte Regel eine weiche mit großer Strafe ist — und wann der Unterschied trotzdem zählt.
4. Ein Kunde sagt: „Kein Mitarbeiter soll mehr als zwei Wochenenden im Monat arbeiten." Nennen Sie zwei Lesarten und je eine Situation, in der sie auseinanderfallen.
5. Was antworten Sie auf die Frage „Was soll passieren, wenn es keine zulässige Lösung gibt?" — und warum ist „das kommt nicht vor" keine Antwort?

4.11 Zusammenfassung

- Ein Modell rechnet aus, was **aufgeschrieben** wurde, nicht was gemeint war. Die Zielfunktion ist eine vollständige Aussage: Was nicht darin steht, ist wertlos.
- Die **Zielgröße** ist die folgenreichste Entscheidung eines OR-Projekts. Umsatz, Auslastung und Deckungsbeitrag führten hier auf demselben Datensatz zu drei verschiedenen Plänen mit 24 500 € Unterschied.
- **Auslastung ist ein Ergebnis, kein Ziel.** Bei knapper Kapazität ist die Maschine ohnehin voll; entscheidend ist, womit.
- Ein Satz aus einer Besprechung hat meist **mehrere Lesarten**. „Stammkunden nicht verlieren" kostete je nach Lesart 3 500 € oder 12 500 € — bei identischem Ergebnis für die Kunden.
- **Hart ist nur, was rechtlich oder physikalisch unmöglich ist.** Eine harte Regel ist eine weiche mit hinreichend großer Strafe; der Unterschied zeigt sich erst, wenn sie einmal nicht erfüllbar ist — dann liefert die harte Variante gar nichts mehr.
- **Die meisten Sätze haben eine Standardübersetzung** — das Lexikon in Abschnitt 4.6 nennt sie. Sein Wert liegt aber in den fünf Wendungen, bei denen es keine gibt: „möglichst", „nicht mehr als", „im Durchschnitt", „so schnell wie möglich", „fair". Dort ist Nachfragen die Modellierungsarbeit.
- Die vier Fragen — worüber wird entschieden, woran gemessen, was ist unverhandelbar, was bei Unlösbarkeit — gehören in die erste Besprechung. Der Leitfaden in Abschnitt 4.5 macht zwölf daraus.

Synthese Teil I — Grundlagen auf einen Blick

Vier Kapitel, eine Frage: **Was muss vorliegen, bevor ein Solver überhaupt sinnvoll gestartet werden kann?** Die Antwort ist unbequem, weil das meiste davon nichts mit Programmieren zu tun hat.

Was welches Kapitel klärt

Frage	Kapitel	Was Sie danach können
Was ist überhaupt ein Optimierungsmodell?	Kapitel 1	Die vier Bausteine benennen — Variablen, Zielfunktion, Nebenbedingungen, Daten — und sie in einem fremden Problem wiedererkennen
Warum rechnet der Solver falsch, obwohl das Modell stimmt?	Kapitel 2	Konvexität beurteilen, den zulässigen Bereich als Polyeder sehen, schlechte Konditionierung erkennen und beheben
Welche Bibliothek, und warum vertragen sie sich nicht?	Kapitel 3	Solver, Bindings und Modellierungsschichten unterscheiden — und den HiGHS-Symbolkonflikt zwischen `ortools` und `highspy` umgehen
Wie wird aus einem Satz ein Modell?	Kapitel 4	Die vier Fragen stellen, harte von weichen Regeln trennen, einen Besprechungssatz übersetzen (Abschnitt 4.6)

Die Reihenfolge ist die Aussage

Dieser Teil steht bewusst **vor** den Verfahren. Die drei häufigsten Gründe, aus denen ein OR-Projekt scheitert, liegen alle hier — und keiner davon ist ein Solverproblem:

1. **Die falsche Zielgröße.** Sie ist die folgenreichste Entscheidung des ganzen Projekts, und sie wird meist in den ersten zehn Minuten nebenbei getroffen. Auslastung ist ein Ergebnis, kein Ziel; „Umsatz" wird gesagt, wenn „Deckungsbeitrag" gemeint ist.
2. **Zu viele harte Regeln.** Jede unnötig harte Bedingung ist eine Zeitbombe: Irgendwann ist sie nicht erfüllbar, und das System antwortet gar nicht mehr. Hart ist nur, was rechtlich oder physikalisch unmöglich ist.
3. **Ein Satz, der mehrere Lesarten hat.** „Stammkunden dürfen wir nicht verlieren" lässt sich auf mindestens drei Arten modellieren, und keine ist aus dem Satz ableitbar. Die Wahl trifft man — im Zweifel unbewusst.

Wenn Sie nur eines mitnehmen

 Ein Modell rechnet aus, was **aufgeschrieben** wurde, nicht was gemeint war. Die Arbeit, die darüber entscheidet, ob am Ende etwas Brauchbares herauskommt, findet vor der ersten Zeile Code statt — und sie besteht aus Fragen an andere Menschen, nicht aus Mathematik.

Vor Teil II sollten Sie an einem eigenen Beispiel die vier Fragen aus Abschnitt 4.3 durchgespielt haben. Ohne das bleiben die Verfahren Technik ohne Anwendung.

Teil II: Die Kernverfahren der deterministischen Optimierung

Die nächsten vier Kapitel behandeln Probleme, bei denen **alle Daten bekannt** sind: Kosten, Kapazitäten, Zeiten stehen fest. Das klingt nach einer starken Vereinfachung, deckt aber den größten Teil der betrieblichen Planung ab — und ist die Grundlage für alles Weitere.

Vier Kapitel, vier Werkzeugkästen. Welcher zu Ihrem Problem passt, entscheiden Sie mit vier Fragen:

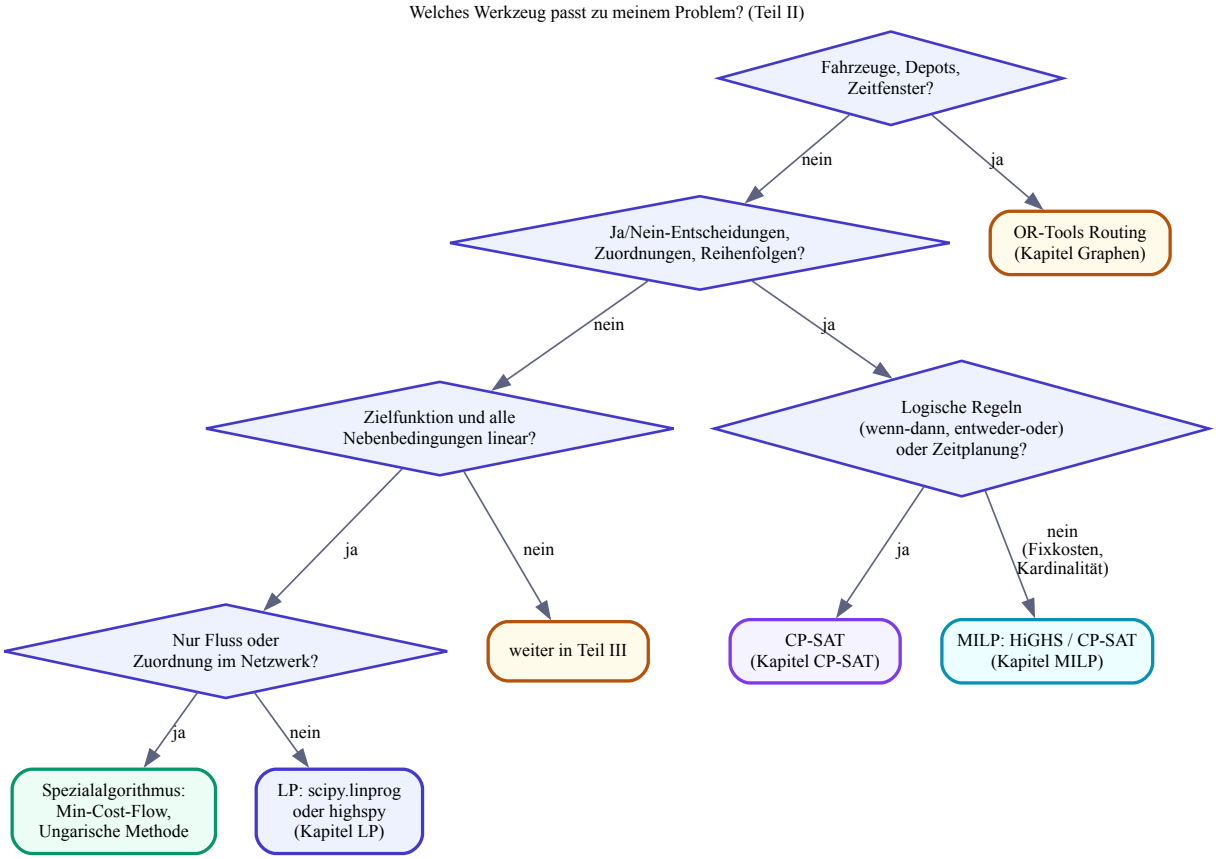

Abbildung 7: Entscheidungsdiagramm: Welches Werkzeug passt zu meinem Problem? (Teil II)

 Merksatz zum Diagramm Die erste Frage ist die wichtigste — und sie lautet **nicht** „welcher Solver ist der schnellste?“, sondern „welche **Struktur** hat mein Problem?“. Ein Zuordnungsproblem braucht keinen MILP-Solver, sondern eine Zeile `linear_sum_assignment`. Ein Dienstplan gehört nicht in ein LP, sondern zu CP-SAT. Wer die Struktur erkennt, hat die halbe Laufzeit gespart, bevor die erste Zeile Code geschrieben ist.

Fällt Ihr Problem durch das Diagramm hindurch — nichtlineare Zielfunktion, unsichere Daten, Entscheidungen über mehrere Zeitstufen —, dann ist Teil III der richtige Ort. Dort steht ein zweites Diagramm für genau diese Fälle.

Kapitel 5: Lineare Programmierung — Simplex, Dualität und Schattenpreise

Kapitel auf einen Blick

Worum geht es? Um das Kernverfahren des Operations Research. Wir bauen den Simplex-Algorithmus selbst — nicht um ihn zu verwenden, sondern um zu verstehen, woher **Schattenpreise** kommen. Sie sind das wertvollste Nebenprodukt jeder Optimierung.

Voraussetzungen: Kapitel 2 (Matrixform, Polyeder, Ecken).

Danach können Sie: Ein LP in Standardform bringen, den Simplex von Hand rechnen, Dualwerte interpretieren, daraus wirtschaftliche Handlungsempfehlungen ableiten — und erkennen, wann ein Schattenpreis eine Entscheidung **nicht** tragen darf.

Zeitbedarf: ca. 7 Stunden — das umfangreichste Kapitel von Teil II.

Programme:
`Simplex_Tableau_LP.py`
`Sensitivitaetsanalyse.py`
`Dualitaet_Nachweis.py`
`Toleranzen_und_Entartung.py`

Notebook: `Notebooks_04/lp.ipynb`

5.1 In 5 Minuten gelöst

Bisher hat ein Solver Ihnen gesagt, *was* Sie tun sollen. Jetzt sagt er Ihnen zusätzlich, *was eine zusätzliche Stunde wert wäre* — und das ist die Zahl, für die Sie im Betrieb gehört werden.

In 5 Minuten gelöst: Wo lohnen sich Überstunden?

Ein Zulieferer fertigt Karosserieteile und Rahmen. Drei Bereiche begrenzen ihn: der Lackierofen (100 h), die Montage (160 h) und die Qualitätsprüfung (120 h).

Produkt	Deckungsbeitrag	Lackierofen	Montage	Prüfung
Karosserieteil	90 €	1 h	2 h	1 h
Rahmen	140 €	2 h	2 h	1 h

Der Betriebsrat hat 20 Überstunden genehmigt. **In welchem Bereich sollen sie eingesetzt werden?**

```
from scipy.optimize import linprog

# Deckungsbeitrag je Stueck (negiert, weil linprog minimiert)
c = [-90, -140]
A = [[1, 2],    # Lackierofen  (Stunden je Stueck)
     [2, 2],    # Montage
     [1, 1]]    # Qualitaetspruefung
b = [100, 160, 120]                       # verfuegbare Stunden
name = ["Lackierofen", "Montage", "Pruefung"]

res = linprog(c, A_ub=A, b_ub=b, bounds=(0, None))
print(f"Plan: {res.x[0]:.0f} Karosserieteile, {res.x[1]:.0f} Rahmen "
      f"-> {-res.fun:,.0f} EUR")
for i, y in enumerate(-res.ineqlin.marginals):
    print(f"  eine Stunde {name[i]:<12} mehr waere wert: {y:6.2f} EUR")
```

Ausgabe:

```
Plan: 60 Karosserieteile, 20 Rahmen -> 8,200 EUR
  eine Stunde Lackierofen  mehr waere wert:  50.00 EUR
  eine Stunde Montage      mehr waere wert:  20.00 EUR
  eine Stunde Pruefung     mehr waere wert:   0.00 EUR
```

Die Antwort steht in der letzten Spalte, und sie ist eindeutig: Die 20 Überstunden gehören an den **Lackierofen**. Jede Stunde dort bringt 50 € zusätzlichen Deckungsbeitrag, in der Montage nur 20 € — und in der Qualitätsprüfung **null**, denn dort sind 40 Stunden ohnehin unbenutzt. Wer die Überstunden gleichmäßig verteilt hätte, verschenkt gegenüber der richtigen Zuteilung rund 400 €.

Diese drei Zahlen heißen **Schattenpreise** (oder *Dualwerte*). Sie sind das eigentliche Produkt einer Optimierung — wertvoller als der Plan selbst, weil sie eine Frage beantworten, die der Plan gar nicht stellt: *Woran hängt es, und was wäre eine Verbesserung wert?*

> **Merksatz** Der Plan sagt Ihnen, was Sie heute tun sollen. Der Schattenpreis sagt Ihnen, worin Sie morgen investieren sollen. Ein Schattenpreis von 0 ist dabei genauso wertvoll wie ein hoher: Er beweist, dass diese Ressource **nicht** der Engpass ist — und bewahrt Sie vor einer Investition, die nichts bringt.

Warum funktioniert das? Der Schattenpreis fällt beim Lösen nebenbei ab. Er ist die Lösung eines zweiten, spiegelbildlichen Problems, das der Solver implizit mitlöst: nicht „welche Mengen produziere ich?“, sondern „was sind meine Ressourcen wert?“. Dieses Zwillingsproblem heißt **duales Problem**, und der Rest des Kapitels erklärt, woher es kommt, warum seine Lösung exakt denselben Zielwert hat — und in welchen zwei Fällen man Schattenpreisen **nicht** trauen darf.

5.2 Lernziele

Nach diesem Kapitel können Sie …

1. … Schlupfvariablen einführen und ein Ungleichungssystem in ein Gleichungssystem überführen.
2. … eine Simplex-Iteration von Hand rechnen: Pivotspalte, Pivotzeile, Basistausch.
3. … erklären, warum das Verfahren terminiert und warum es das Optimum findet.
4. … das duale Problem zu einem gegebenen primalen aufstellen.
5. … **Schattenpreise** korrekt auslesen und interpretieren — inklusive der Vorzeichenfalle, die einer der häufigsten Fehler bei der Sensitivitätsanalyse ist.
6. … aus einer Sensitivitätsanalyse eine begründete Kaufentscheidung ableiten.
7. … ein Modell auf **Entartung** prüfen und statt eines nicht eindeutigen Schattenpreises dessen belastbare Spanne berechnen.
8. … erklären, warum Zulässigkeit immer gegen eine Toleranz und nie mit `== 0` geprüft wird.

5.3 Die Standardform und Schlupfvariablen

Lineare Programme zeichnen sich dadurch aus, dass Zielfunktion **und** alle Nebenbedingungen **strikt linear** sind — keine Produkte von Variablen, keine Quadrate, keine Logarithmen.

Ein Ungleichungssystem ist rechnerisch unhandlich. Gleichungssysteme dagegen kann man mit Gauß-Elimination lösen. Die Brücke bilden **Schlupfvariablen** (*slack variables*) $s_i \geq 0$:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad \Longleftrightarrow \qquad \sum_{j=1}^{n} a_{ij} x_j + s_i = b_i, \quad s_i \geq 0$$

 Formel-Lesehilfe Links steht „der Verbrauch ist **höchstens** die Kapazität“. Rechts steht „der Verbrauch **plus ein Rest** ist **genau** die Kapazität“.

Ohne Formel gesagt: Statt zu sagen „ich verbrauche höchstens 60 GB“, sagt man „ich verbrauche 60 GB minus dem, was übrig bleibt“. Der Rest bekommt einen Namen — und wird dadurch zu einer Variablen, mit der man rechnen kann.

Die ökonomische Bedeutung von s_i ist der eigentliche Gewinn dieser Umformung:

s_i	Bedeutung	Fachbegriff
$s_i = 0$	Ressource i ist **vollständig ausgelastet**	bindende Nebenbedingung, **Engpass** (*bottleneck*)
$s_i > 0$	Es sind s_i Einheiten **übrig**	nicht-bindende Nebenbedingung, Reserve

 Merksatz Die Schlupfvariable ist kein Rechentrick, sondern eine Messgröße: Sie sagt Ihnen, welche Ressource Ihr Geschäft begrenzt. Und nur wo sie null ist, kann es sich lohnen, Kapazität nachzukaufen.

5.4 Der Simplex-Algorithmus Schritt für Schritt

Der von George Dantzig 1947 formulierte Simplex-Algorithmus nutzt den Fundamentalsatz aus Kapitel 2: **Das Optimum liegt in einer Ecke.** Statt alle Ecken aufzuzählen, wandert er von Ecke zu Ecke — und zwar immer in Richtung Verbesserung.

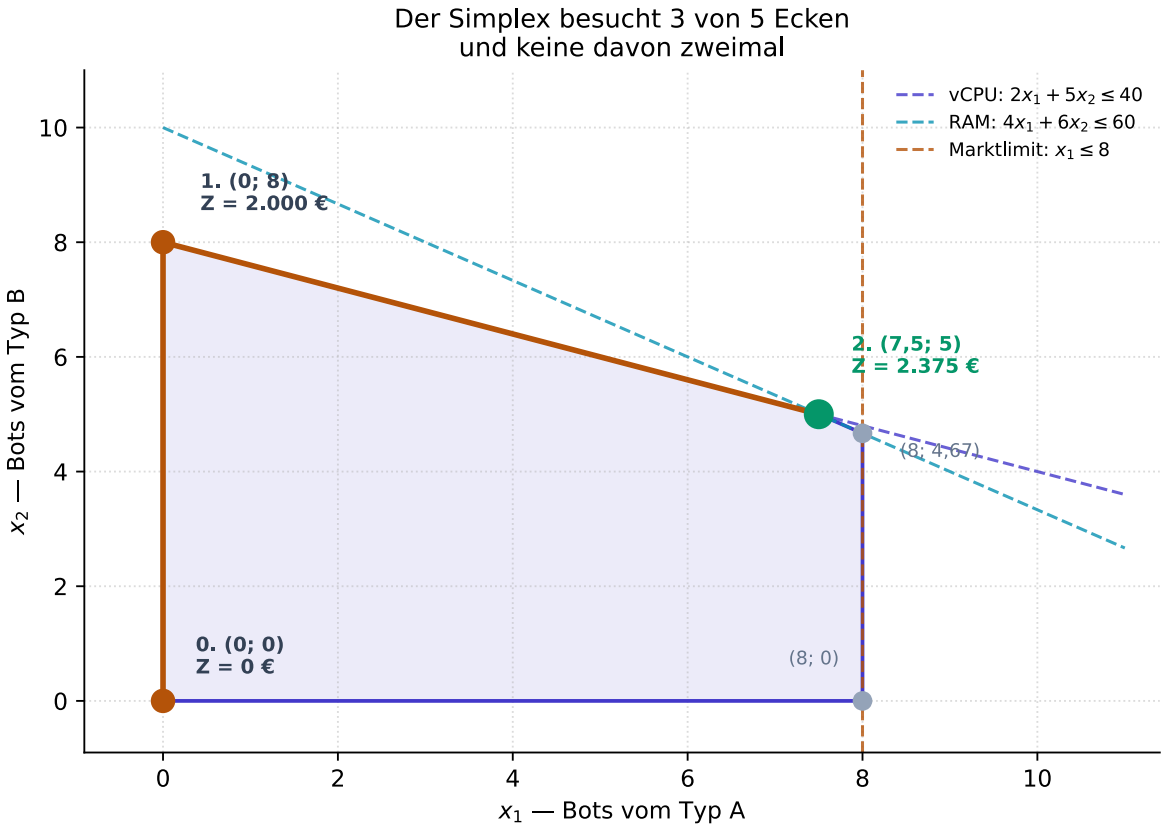

Abbildung 8: Abb. 5.1: Derselbe zulässige Bereich, aber der Weg der Handrechnung weiter unten: vom Ursprung über (0; 8) zum Optimum (7,5; 5). Drei der fünf Ecken werden besucht, keine zweimal — das ist der ganze Gewinn gegenüber dem Aufzählen aller Ecken. Erzeugt von `bilder_04/erzeuge_polyeder.py`.

Der Ablauf

1. **Initialisierung.** Starte an einer bekannten Ecke — meist dem Ursprung $\mathbf{x} = \mathbf{0}$, bei dem alle Schlupfvariablen ihre Kapazität aufnehmen: $\mathbf{s} = \mathbf{b}$. Das ist zulässig, solange $\mathbf{b} \geq \mathbf{0}$.
2. **Optimalitätsprüfung.** Sieh dir die Zielfunktionszeile an. Gibt es eine Variable, deren Aufnahme in die Basis den Zielwert noch verbessern würde? → **Nein:** fertig, das Optimum ist erreicht.
3. **Pivotspalte wählen (eintretende Variable).** Diejenige Variable mit dem größten Verbesserungspotenzial pro Einheit (*Dantzig-Regel*).
4. **Pivotzeile wählen (austretende Variable).** Der **minimale Quotient** $\min_i\{b_i/a_{ik} \mid a_{ik} > 0\}$. Das ist die Bedingung, die als erste erschöpft ist, wenn man die neue Variable hochfährt.
5. **Basistausch (Gauß-Jordan-Schritt).** Eliminiere die eintretende Variable aus allen anderen Zeilen. Weiter bei Schritt 2.

□ **Handrechnung 5.1: Eine vollständige Simplex-Rechnung**

Wir lösen von Hand:

$$\max\ Z = 150x_1 + 250x_2 \quad \text{u. d. N.} \quad 2x_1 + 5x_2 \le 40, \quad 4x_1 + 6x_2 \le 60, \quad x_1 \le 8, \quad x_1, x_2 \ge 0$$

Schritt 0 — Schlupfvariablen einführen:

$$\begin{aligned} 2x_1 + 5x_2 + s_1 &= 40 \\ 4x_1 + 6x_2 + s_2 &= 60 \\ x_1 + s_3 &= 8 \\ Z - 150x_1 - 250x_2 &= 0 \end{aligned}$$

Starttableau (Basis: s_1, s_2, s_3; also $x_1 = x_2 = 0$, $Z = 0$):

Basis	x_1	x_2	s_1	s_2	s_3	RHS
s_1	2	**5**	1	0	0	40
s_2	4	6	0	1	0	60
s_3	1	0	0	0	1	8
Z	−150	**−250**	0	0	0	0

Iteration 1. *Pivotspalte:* In der Z-Zeile ist -250 am negativsten → x_2 tritt ein. *Pivotzeile:* Quotienten $40/5 = 8$, $60/6 = 10$, $8/0 = \infty$ → **Zeile 1** (kleinster Quotient 8). Pivotelement ist die **5**.

Pivotzeile durch 5 teilen: (0,4; 1; 0,2; 0; 0 | 8). Dann x_2 aus den anderen Zeilen eliminieren: * Zeile 2 − 6· Pivotzeile: (4 − 2,4; 0; −1,2; 1; 0 | 60 − 48) = (1,6; 0; −1,2; 1; 0 | 12) * Zeile 3 bleibt (Koeffizient war 0). * Z-Zeile + 250· Pivotzeile: (−150 + 100; 0; 50; 0; 0 | 2000) = (−50; 0; 50; 0; 0 | 2000)

Basis	x_1	x_2	s_1	s_2	s_3	RHS
x_2	0,4	1	0,2	0	0	8
s_2	**1,6**	0	−1,2	1	0	12
s_3	1	0	0	0	1	8
Z	**−50**	0	50	0	0	2000

Zwischenstand: $x_2 = 8$, $Z = 2000$ — das ist genau die naive Lösung „nur Trendfolge“ aus Kapitel 1. Der Simplex ist noch nicht fertig, denn in der Z-Zeile steht noch -50.

Iteration 2. *Pivotspalte:* x_1 (einziger negativer Eintrag, -50). *Pivotzeile:* Quotienten $8/0{,}4 = 20$, $12/1{,}6 = 7{,}5$, $8/1 = 8$ → **Zeile 2** (kleinster Quotient 7,5). Pivotelement **1,6**.

Pivotzeile durch 1,6: (1; 0; −0,75; 0,625; 0 | 7,5). Eliminieren: * Zeile 1 − 0,4·: (0; 1; 0,5; −0,25; 0 | 5) * Zeile 3 − 1·: (0; 0; 0,75; −0,625; 1 | 0,5) * Z-Zeile + 50·: (0; 0; 12,5; 31,25; 0 | 2375)

Basis	x_1	x_2	s_1	s_2	s_3	RHS
x_2	0	1	0,5	−0,25	0	5
x_1	1	0	−0,75	0,625	0	7,5
s_3	0	0	0,75	−0,625	1	0,5
Z	0	0	**12,5**	**31,25**	**0**	**2375**

Alle Einträge der Z-Zeile sind ≥ 0 → Optimum erreicht.

Ablesen: $x_1^* = 7{,}5$, $x_2^* = 5$, $Z^* = 2375$. Das deckt sich exakt mit der Eckenrechnung aus Kapitel 2 — und wir haben statt fünf Ecken nur **drei** besucht.

Und jetzt der Clou: Die Zahlen in der Z-Zeile unter den Schlupfvariablen sind die **Schattenpreise**: * s_1 (vCPU): $y_1^* = 12{,}5$ €/vCPU * s_2 (RAM): $y_2^* = 31{,}25$ €/GB * s_3 (Marktlimit): $y_3^* = 0$ €

Der letzte Wert ist null, weil $s_3 = 0{,}5 > 0$ — es ist noch Luft beim Arbitrage-Limit, also ist eine Erhöhung dieser Grenze nichts wert. Genau das besagt der Satz vom komplementären Schlupf, den wir in Abschnitt 5.6 formulieren.

5.5 Das Simplex-Tableau in Python

```
#!/usr/bin/env python3

# Simplex_Tableau_LP.py
"""
Kapitel LP: Vollständige Implementierung des Simplex-Algorithmus (Tableau-Methode).

GRENZEN DIESER IMPLEMENTIERUNG (bewusst, aus didaktischen Gründen):
  * nur Maximierung
  * nur "<="-Nebenbedingungen
  * alle b_i >= 0   (sonst wäre der Ursprung keine zulässige Startecke und man
    bräuchte eine Phase-1-Rechnung mit künstlichen Variablen)
Für den produktiven Einsatz nimmt man HiGHS - dieser Code dient dem Verständnis.

Voraussetzungen werden geprüft statt stillschweigend angenommen;
Iterationsprotokoll und Schattenpreise werden ausgegeben.
"""

import numpy as np


class SimplexTableauSolver:
    """Maximierungs-Standardform:  max c^T x  u.d.N.  A x <= b,  x >= 0,  b >= 0."""

    def __init__(self, c, A, b, variablennamen=None, restriktionsnamen=None):
```

```
        self.c = np.asarray(c, dtype=float)
        self.A = np.asarray(A, dtype=float)
        self.b = np.asarray(b, dtype=float)
        self.n = len(self.c)                    # Anzahl Originalvariablen
        self.m = len(self.b)                    # Anzahl Nebenbedingungen

        # --- Voraussetzungen pruefen, statt sie stillschweigend anzunehmen ---
        if self.A.shape != (self.m, self.n):
            raise ValueError(f"A hat Form {self.A.shape}, erwartet ({self.m}, {self.n}).")
        if np.any(self.b < 0):
            raise ValueError(
                "Mindestens ein b_i ist negativ. Dann ist der Ursprung keine zulässige "
                "Startecke; dieser Solver benötigt eine Phase-1-Rechnung, die hier "
                "bewusst nicht implementiert ist. Nutzen Sie scipy.optimize.linprog."
            )

        self.var_namen = variablennamen or [f"x{j+1}" for j in range(self.n)]
        self.restr_namen = restriktionsnamen or [f"R{i+1}" for i in range(self.m)]

        self.tableau = None
        self.basis = None                       # welche Variable ist in welcher Zeile Basis?
        self._baue_starttableau()

    def _baue_starttableau(self):
        """Zeilen: m Nebenbedingungen + Zielfunktionszeile.
           Spalten: n Variablen + m Schlupfvariablen + rechte Seite."""
        m, n = self.m, self.n
        self.tableau = np.zeros((m + 1, n + m + 1))
        self.tableau[:m, :n] = self.A                   # Koeffizienten
        self.tableau[:m, n:n + m] = np.eye(m)           # Schlupfvariablen
        self.tableau[:m, -1] = self.b                   # rechte Seite
        self.tableau[-1, :n] = -self.c                  # Zielzeile: -c (Maximierung)
        self.basis = list(range(n, n + m))              # Start: alle Schlupf in der Basis

    def _spaltenname(self, index):
        return self.var_namen[index] if index < self.n else f"s{index - self.n + 1}"

    def solve(self, max_iterationen=100, protokoll=True):
        m, n = self.m, self.n
        if protokoll:
            print(f"{'Iter':>4} | {'eintritt':>9} | {'austritt':>9} | "
                  f"{'Pivot':>8} | {'Z':>12}")
            print("-" * 58)

        for iteration in range(1, max_iterationen + 1):
            zielzeile = self.tableau[-1, :-1]

            # 1. Optimalitätsprüfung: alle Koeffizienten >= 0 ?
```

```
            if np.all(zielzeile >= -1e-9):
                if protokoll:
                    print("-" * 58)
                    print(f"Optimum nach {iteration - 1} Pivotschritten erreicht.")
                return self._loesung_auslesen()

            # 2. Pivotspalte: negativster Eintrag (Dantzig-Regel)
            pivot_spalte = int(np.argmin(zielzeile))

            # 3. Pivotzeile: minimaler Quotient über POSITIVE Spalteneinträge
            spalte = self.tableau[:m, pivot_spalte]
            rechte_seite = self.tableau[:m, -1]
            quotienten = np.where(spalte > 1e-9, rechte_seite / np.where(spalte > 1e-9,
↪  spalte, 1),
                                  np.inf)
            pivot_zeile = int(np.argmin(quotienten))
            if not np.isfinite(quotienten[pivot_zeile]):
                raise ValueError(
                    f"Problem ist unbeschraenkt: Variable {self._spaltenname(pivot_spalte)} "
                    "kann beliebig wachsen, ohne eine Bedingung zu verletzen. "
                    "Meist fehlt eine Kapazitaetsbeschraenkung."
                )

            if protokoll:
                print(f"{iteration:>4} | {self._spaltenname(pivot_spalte):>9} | "
                      f"{self._spaltenname(self.basis[pivot_zeile]):>9} | "
                      f"{self.tableau[pivot_zeile, pivot_spalte]:>8.3f} | "
                      f"{self.tableau[-1, -1]:>12,.2f}")

            # 4. Pivotoperation (Gauß-Jordan)
            pivot_wert = self.tableau[pivot_zeile, pivot_spalte]
            self.tableau[pivot_zeile, :] /= pivot_wert
            for zeile in range(m + 1):
                if zeile != pivot_zeile:
                    faktor = self.tableau[zeile, pivot_spalte]
                    self.tableau[zeile, :] -= faktor * self.tableau[pivot_zeile, :]

            self.basis[pivot_zeile] = pivot_spalte

        raise RuntimeError("Maximale Iterationszahl ueberschritten (moeglicherweise Zyklus).")

    def _loesung_auslesen(self):
        """Basisvariablen tragen den RHS-Wert ihrer Zeile, Nichtbasisvariablen sind 0."""
        x = np.zeros(self.n + self.m)
        for zeile, spalte in enumerate(self.basis):
            x[spalte] = self.tableau[zeile, -1]
        return x[:self.n], x[self.n:], self.tableau[-1, -1]
```

```
    def schattenpreise(self):
        """Die Zielzeile unter den Schlupfspalten enthält direkt die Dualwerte."""
        return self.tableau[-1, self.n:self.n + self.m].copy()


if __name__ == "__main__":
    # Modell aus der Simplex-Handrechnung (Bot-Beispiel, Kapitel Einfuehrung)
    ertraege = [150.0, 250.0]
    matrix = [[2.0, 5.0],
              [4.0, 6.0],
              [1.0, 0.0]]
    kapazitaeten = [40.0, 60.0, 8.0]
    var_namen = ["x_A", "x_B"]
    restr_namen = ["vCPU", "RAM", "Marktlimit"]

    print("=" * 58)
    print("  SIMPLEX-TABLEAU: ITERATIONSPROTOKOLL")
    print("=" * 58)

    solver = SimplexTableauSolver(ertraege, matrix, kapazitaeten, var_namen, restr_namen)
    x_opt, schlupf, z_opt = solver.solve()

    print("\n" + "=" * 58)
    print("  ERGEBNIS")
    print("=" * 58)
    for name, wert in zip(var_namen, x_opt):
        print(f"  {name:<12} = {wert:8.4f}")
    print(f"  {'Zielwert Z':<12} = {z_opt:8.2f} EUR")

    print("\n  Ressourcenanalyse:")
    print(f"  {'Ressource':<12} {'Schlupf':>9} {'Status':>22} {'Schattenpreis':>15}")
    print("  " + "-" * 60)
    for name, s, y in zip(restr_namen, schlupf, solver.schattenpreise()):
        status = "ENGPASS (bindend)" if abs(s) < 1e-9 else "Reserve vorhanden"
        print(f"  {name:<12} {s:>9.3f} {status:>22} {y:>12.2f} EUR")

    # Selbstkontrolle: komplementaerer Schlupf muss gelten
    for s, y in zip(schlupf, solver.schattenpreise()):
        assert abs(s * y) < 1e-6, "Komplementaerer Schlupf verletzt - Rechenfehler!"
    print("\n  Pruefung: komplementaerer Schlupf (s_i * y_i = 0) fuer alle i erfuellt.")
    print("=" * 58)
```

Erwartete Ausgabe:

```
==========================================================
  SIMPLEX-TABLEAU: ITERATIONSPROTOKOLL
==========================================================
Iter |  eintritt |  austritt |    Pivot |           Z
```

```
------------------------------------------------------------
   1 |       x_B |        s1 |    5.000 |         0.00
   2 |       x_A |        s2 |    1.600 |     2,000.00
------------------------------------------------------------
Optimum nach 2 Pivotschritten erreicht.

============================================================
  ERGEBNIS
============================================================
  x_A          =   7.5000
  x_B          =   5.0000
  Zielwert Z   =  2375.00 EUR

  Ressourcenanalyse:
  Ressource      Schlupf                Status   Schattenpreis
  ------------------------------------------------------------
  vCPU             0.000       ENGPASS (bindend)        12.50 EUR
  RAM              0.000       ENGPASS (bindend)        31.25 EUR
  Marktlimit       0.500       Reserve vorhanden         0.00 EUR

  Pruefung: komplementaerer Schlupf (s_i * y_i = 0) fuer alle i erfuellt.
============================================================
```

Die Ausgabe reproduziert die Simplex-Handrechnung exakt — inklusive der Schattenpreise 12,50 € und 31,25 €.

Code-Durchgang

Stelle	Was passiert	Warum
`if np.any(self.b < 0): raise`	Voraussetzung geprüft	Ohne diese Prüfung liefert der Solver hier still ein falsches Ergebnis. Ein Solver, der seine Annahmen nicht prüft, ist gefährlich
`self.tableau[-1, :n] = -self.c`	Zielzeile **negativ** eintragen	Trick, damit „alle Einträge ≥ 0“ das Abbruchkriterium wird
`np.argmin(zielzeile)`	Dantzig-Regel	Größte Verbesserung pro Einheit; andere Regeln (Bland) vermeiden Zyklen, sind aber langsamer
`quotienten = np.where(spalte > 1e-9, ...)`	Minimum-Quotienten-Test	**Nur positive** Spalteneinträge zählen — sonst würde eine Variable negativ
`raise ValueError("unbeschraenkt")`	alle Quotienten unendlich	Diagnose statt Absturz: „Ihnen fehlt eine Nebenbedingung“

Stelle	Was passiert	Warum
`self.basis[pivot_zeile] = pivot_spalte`	Basiswechsel protokollieren	Ohne diese Buchführung kann man die Lösung am Ende nicht auslesen
`assert abs(s*y) < 1e-6`	komplementärer Schlupf	Eine Zeile, die jeden Vorzeichenfehler sofort aufdeckt

Typische Fehler

- **Negative rechte Seite.** $-3x_1 - 2x_2 \leq -12$ (aus einer „≥“-Bedingung entstanden) macht den Ursprung unzulässig. Dann braucht man Phase 1 mit künstlichen Variablen. Dieser Solver meldet das jetzt ehrlich.
- **Zyklen bei Entartung.** Wenn mehrere Quotienten gleich sind, kann der Simplex theoretisch endlos kreisen. In der Praxis extrem selten; Bland's Regel verhindert es beweisbar.
- **Den Simplex selbst produktiv einsetzen.** Diese Implementierung ist ein Lehrmodell. HiGHS ist um Größenordnungen schneller und numerisch stabiler.

5.6 Dualität und Schattenpreise

Zu **jedem** linearen Optimierungsproblem (dem **primalen Problem**) existiert ein spiegelbildliches Zwillingsproblem: das **duale Problem**.

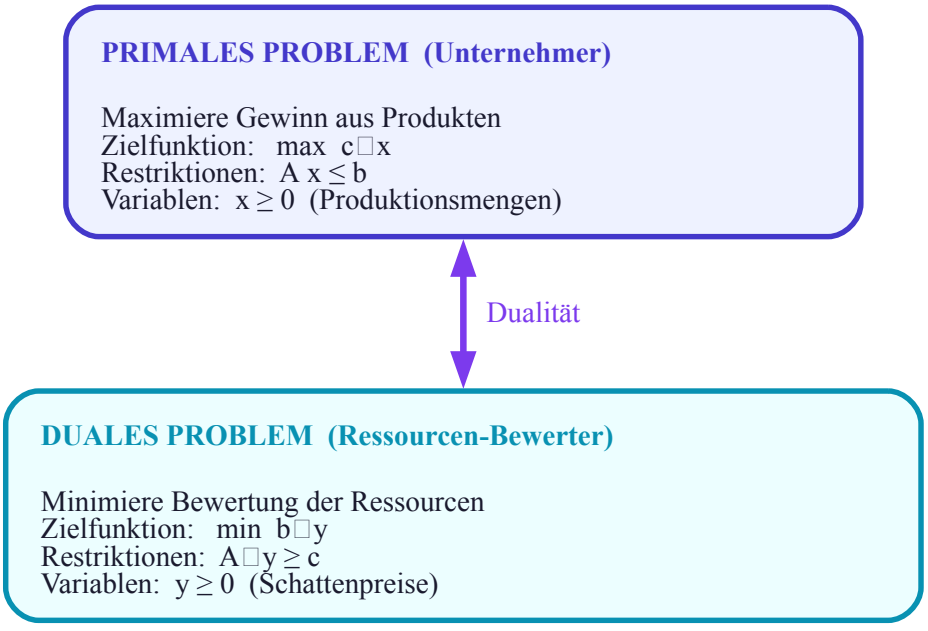

Abbildung 9: Abb. 5.2: Primales und duales Problem

	Primal (Unternehmer)	**Dual** (Ressourcenbewerter)
Fragestellung	„Wie viel produziere ich?“	„Was sind meine Ressourcen wert?“
Zielfunktion	max $\mathbf{c}^\top\mathbf{x}$	min $\mathbf{b}^\top\mathbf{y}$
Nebenbedingungen	$\mathbf{Ax} \le \mathbf{b}$	$\mathbf{A}^\top\mathbf{y} \ge \mathbf{c}$
Variablen	$\mathbf{x} \ge \mathbf{0}$ (Mengen)	$\mathbf{y} \ge \mathbf{0}$ (Preise)

 Formel-Lesehilfe zum dualen Problem * $\mathbf{b}^\top\mathbf{y}$ — der Gesamtwert aller vorhandenen Kapazitäten, bewertet mit den Preisen $\mathbf{y}$. * $\mathbf{A}^\top\mathbf{y} \ge \mathbf{c}$ — für **jedes Produkt** muss gelten: Der Wert der Ressourcen, die eine Einheit verbraucht, ist mindestens so hoch wie ihr Verkaufserlös.

Ohne Formel gesagt: Ein gedachter Käufer will Ihnen die gesamten Ressourcen abkaufen. Er will möglichst wenig zahlen (Minimierung), muss Ihnen aber pro Ressourcenbündel mindestens so viel bieten, wie Sie durch eigene Produktion verdienen würden — sonst lehnen Sie ab.

Der starke Dualitätssatz

Satz. Besitzt das primale Problem eine endliche Optimallösung $\mathbf{x}^*$, so besitzt auch das duale eine Optimallösung $\mathbf{y}^*$, und die Zielwerte sind **exakt gleich**:

$$\mathbf{c}^\top\mathbf{x}^* = \mathbf{b}^\top\mathbf{y}^*$$

Prüfen wir das an unserem Beispiel: $\mathbf{b}^\top\mathbf{y}^* = 40 \cdot 12{,}5 + 60 \cdot 31{,}25 + 8 \cdot 0 = 500 + 1875 + 0 = 2375$ — identisch mit $Z^* = 2375$.

Das lässt sich auch unabhängig von der Handrechnung nachweisen: Primal- und Dualproblem werden als **zwei getrennte, voneinander unabhängige LPs** aufgestellt und gelöst — wenn der starke Dualitätssatz stimmt, müssen beide denselben Zielwert liefern, ohne dass eines vom anderen „weiß“.

```
#!/usr/bin/env python3

# Dualitaet_Nachweis.py
"""
Kapitel LP: Primales und duales Problem unabhaengig loesen und den starken
Dualitaetssatz sowie den komplementaeren Schlupf numerisch nachweisen.

Modell aus der Simplex-Handrechnung (Bot-Allokation, LP-Relaxation):
    max 150*x1 + 250*x2  u.d.N.  2*x1+5*x2<=40, 4*x1+6*x2<=60, x1<=8, x>=0
"""

import numpy as np
from scipy.optimize import linprog

# --- Primales Problem ------------------------------------------------------
C_PRIMAL = np.array([150.0, 250.0])
A_PRIMAL = np.array([[2.0, 5.0], [4.0, 6.0], [1.0, 0.0]])
```

```
B_PRIMAL = np.array([40.0, 60.0, 8.0])

# --- Duales Problem: min b^T y  u.d.N.  A^T y >= c, y >= 0 ------------------
#     linprog kennt nur <=, also A^T y >= c  <=>  -A^T y <= -c
C_DUAL = B_PRIMAL
A_DUAL = -A_PRIMAL.T
B_DUAL = -C_PRIMAL


def loese():
    primal = linprog(c=-C_PRIMAL, A_ub=A_PRIMAL, b_ub=B_PRIMAL,
                     bounds=[(0, None)] * 2, method="highs")
    dual = linprog(c=C_DUAL, A_ub=A_DUAL, b_ub=B_DUAL,
                   bounds=[(0, None)] * 3, method="highs")
    if not primal.success or not dual.success:
        raise SystemExit("Primal oder Dual nicht loesbar.")
    return primal, dual


if __name__ == "__main__":
    primal, dual = loese()
    x = primal.x
    y = dual.x
    z_primal = -primal.fun
    z_dual = dual.fun

    print("=" * 78)
    print("  PRIMALES PROBLEM")
    print("=" * 78)
    print(f"x* = ({x[0]:.4f}, {x[1]:.4f})")
    print(f"Z* = {z_primal:.4f}")

    print("\n" + "=" * 78)
    print("  DUALES PROBLEM")
    print("=" * 78)
    print(f"y* = ({y[0]:.4f}, {y[1]:.4f}, {y[2]:.4f})")
    print(f"W* = {z_dual:.4f}")

    print("\n" + "=" * 78)
    print("  STARKER DUALITAETSSATZ:  c^T x* == b^T y* ?")
    print("=" * 78)
    print(f"  Primal Z* = {z_primal:.6f}")
    print(f"  Dual   W* = {z_dual:.6f}")
    differenz = abs(z_primal - z_dual)
    print(f"  Differenz = {differenz:.2e}  ->  "
          f"{'BESTAETIGT' if differenz < 1e-6 else 'VERLETZT!'}")

    print("\n" + "=" * 78)
```

```
    print("  KOMPLEMENTAERER SCHLUPF:  s_i * y_i == 0 fuer alle i ?")
    print("=" * 78)
    schlupf = B_PRIMAL - A_PRIMAL @ x
    ressourcen = ["vCPU (s1)", "RAM (s2)", "Marktlimit (s3)"]
    for name, s, yi in zip(ressourcen, schlupf, y):
        produkt = s * yi
        print(f"  {name:<16} Schlupf s={s:6.4f}  Schattenpreis y={yi:6.4f}  "
              f"s*y={produkt:.2e}  {'OK' if abs(produkt) < 1e-6 else 'VERLETZT!'}")

    print("\nFazit: Das dual geloeste y* stimmt exakt mit den Schattenpreisen")
    print("überein, die die Simplex-Rechnung von Hand in der Z-Zeile")
    print("ablas - unabhaengig voneinander berechnet, identisches Ergebnis.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  PRIMALES PROBLEM
==============================================================================
x* = (7.5000, 5.0000)
Z* = 2375.0000

==============================================================================
  DUALES PROBLEM
==============================================================================
y* = (12.5000, 31.2500, 0.0000)
W* = 2375.0000

==============================================================================
  STARKER DUALITAETSSATZ:  c^T x* == b^T y* ?
==============================================================================
  Primal Z* = 2375.000000
  Dual   W* = 2375.000000
  Differenz = 4.55e-13  ->  BESTAETIGT

==============================================================================
  KOMPLEMENTAERER SCHLUPF:  s_i * y_i == 0 fuer alle i ?
==============================================================================
  vCPU (s1)        Schlupf s=0.0000  Schattenpreis y=12.5000  s*y=0.00e+00  OK
  RAM (s2)         Schlupf s=0.0000  Schattenpreis y=31.2500  s*y=0.00e+00  OK
  Marktlimit (s3)  Schlupf s=0.5000  Schattenpreis y=0.0000  s*y=0.00e+00  OK

Fazit: Das dual geloeste y* stimmt exakt mit den Schattenpreisen
überein, die die Simplex-Rechnung von Hand in der Z-Zeile
ablas - unabhaengig voneinander berechnet, identisches Ergebnis.
==============================================================================
```

Die winzige Differenz (`4.55e-13`) ist reines Gleitkommarauschen — numerisch ist das exakte Gleichheit. Wichtiger als die Zahl selbst: **Primal und Dual wissen nichts voneinander**, wurden als zwei separate

`linprog`-Aufrufe gelöst, und landen trotzdem exakt beim selben Zielwert. Das *ist* der starke Dualitätssatz, nicht nur eine Illustration davon.

Der Schattenpreis als Ableitung

Der Dualwert y_i^* ist die **Grenzproduktivität** der Ressource i:

$$y_i^* = \frac{\partial Z^*}{\partial b_i}$$

 Formel-Lesehilfe Ohne Formel gesagt: „Um wie viele Euro steigt mein optimaler Gewinn, wenn ich von Ressource i **eine Einheit mehr** hätte?“

Im Beispiel: Ein zusätzliches GB RAM bringt 31,25 € zusätzlichen Tagesgewinn. Ein zusätzlicher vCPU bringt 12,50 €. Eine Lockerung des Arbitrage-Limits bringt **nichts**, weil es gar nicht bindet.

Satz vom komplementären Schlupf:

$$s_i^* \cdot y_i^* = 0 \qquad \text{für alle } i$$

Also: **Entweder** ist die Ressource knapp ($s_i = 0$) und kann einen positiven Preis haben, **oder** sie hat Reserven ($s_i > 0$) und ihr Preis ist zwingend null. Beides gleichzeitig geht nicht.

 Formel-Übersetzer: das Dualitätspaket auf einen Blick

Mathematik	Alltagssprache
max $\mathbf{c}^\top\mathbf{x}$ u. d. N. $\mathbf{Ax} \le \mathbf{b}$	„Wie viel produziere ich von jedem Produkt, damit der Ertrag maximal wird und kein Vorrat überzogen wird?“
min $\mathbf{b}^\top\mathbf{y}$ u. d. N. $\mathbf{A}^\top\mathbf{y} \ge \mathbf{c}$	„Was ist jede Ressourcenstunde wert? So wenig wie möglich — aber jedes Produkt muss seinen Verkaufserlös durch die verbrauchten Ressourcen gedeckt sehen.“
$\mathbf{c}^\top\mathbf{x}^* = \mathbf{b}^\top\mathbf{y}^*$	„Der erwirtschaftete Gewinn und der Wert des Maschinenparks sind dieselbe Zahl — von zwei Seiten betrachtet.“
$y_i^* = \partial Z^*/\partial b_i$	„Um so viel Euro steigt der Gewinn, wenn ich von Ressource i eine Einheit mehr hätte.“
$s_i^* \cdot y_i^* = 0$	„Was übrig ist, ist nichts wert. Was etwas wert ist, ist restlos verbraucht.“ Nie beides zugleich.
$s_i^* > 0 \Rightarrow y_i^* = 0$	„Von dieser Ressource haben Sie zu viel — kaufen Sie davon nichts nach.“
$y_i^* > 0 \Rightarrow s_i^* = 0$	„Diese Ressource ist Ihr Engpass — hier lohnt die Überstunde.“

Das Ganze in einem Satz: *Jede Optimierung beantwortet gleichzeitig zwei Fragen — was zu tun ist, und was die Mittel wert sind, mit denen man es tut.*

 Merksatz Eine Optimierung liefert nicht nur eine Lösung, sondern auch ihre **Begründung**: Die Schattenpreise sagen, welcher Engpass Sie ausbremst und wie viel seine Beseitigung wert ist. Das ist in Managementgesprächen oft wertvoller als die Lösung selbst.

5.7 Die Vorzeichenfalle bei Schattenpreisen

Diese Falle sollten Sie sich merken, weil sie in jedem Projekt wieder auftaucht.

Das Problem: Wir wollen **maximieren**, aber scipy.optimize.linprog“ **minimiert**. Also negieren wir die Zielfunktion. Damit wird auch die Ableitung negiert — und die Schattenpreise kommen mit **umgekehrtem Vorzeichen** heraus.

```
res = linprog(c=-gewinn, A_ub=A, b_ub=b, method="highs")
res.ineqlin.marginals      # <= 0  für "<="-Bedingungen bei Minimierung
```

	Wert aus `marginals`	Wirtschaftliche Bedeutung
Ressource mit Reserve	0	Preis 0
Engpassressource	**negativ**, z. B. −33,33	Schattenpreis ist +33,33 €!

Eine unbedachte Abfrage `if sp > 0:` wird dadurch **niemals** wahr — ein Programm mit diesem Fehler empfiehlt selbst bei den knappsten Ressourcen „keine Zukäufe nötig“.

 Die Regel Nach jeder Negation der Zielfunktion müssen auch die Dualwerte zurücknegiert werden. Und prüfen Sie es: Der Schattenpreis einer Engpassressource muss bei einer Maximierung **positiv** sein. Ist er negativ, haben Sie ein Vorzeichen vergessen.

5.8 Praxisfall: Sensitivitätsanalyse mit korrekten Schattenpreisen

Szenario. Ein Fertigungsbetrieb stellt drei Produkte her und will wissen:

- Lohnt sich der Zukauf zusätzlicher Prüfstunden für 18 €/h?
- Welche Ressource limitiert den Gewinn am stärksten?

```
#!/usr/bin/env python3

# Sensitivitaetsanalyse.py
"""
Kapitel LP: Schattenpreis- und Sensitivitätsanalyse mit SciPy und HiGHS.

Achtung: Ohne Vorzeichenumkehr der Dualwerte waere die Handlungsempfehlung
strukturell immer "kein Zukauf noetig" - selbst bei harten Engpaessen. Dieses
Programm zeigt die korrekte Vorzeichenbehandlung.
```

```
"""

import numpy as np
from scipy.optimize import linprog

# --- Modell: Maximiere Deckungsbeitrag aus 3 Produkten --------------------
#   max 40*x1 + 30*x2 + 50*x3
DECKUNGSBEITRAG = np.array([40.0, 30.0, 50.0])

# Ressourcenverbrauch je Produkt (Zeile = Ressource, Spalte = Produkt)
VERBRAUCH = np.array([
    [2.0, 1.0, 3.0],      # Montagezeit
    [1.0, 2.0, 1.0],      # Lackierzeit
    [1.0, 0.5, 2.0],      # Qualitätsprüfung
])
KAPAZITAET = np.array([120.0, 80.0, 50.0])          # Stunden
RESSOURCEN = ["Montage", "Lackieren", "Qualitätsprüfung"]
PRODUKTE = ["Produkt 1", "Produkt 2", "Produkt 3"]

ANGEBOTSPREIS_PRUEFSTUNDE = 18.0                     # EUR/h - lohnt sich der Zukauf?


def analysiere():
    # linprog MINIMIERT -> Zielfunktion negieren
    ergebnis = linprog(c=-DECKUNGSBEITRAG, A_ub=VERBRAUCH, b_ub=KAPAZITAET,
                       bounds=[(0, None)] * len(DECKUNGSBEITRAG), method="highs")
    if not ergebnis.success:
        raise SystemExit(f"Kein Optimum gefunden: {ergebnis.message}")

    max_gewinn = -ergebnis.fun
    mengen = ergebnis.x
    schlupf = ergebnis.slack

    # ------------------------------------------------------------------
    # DER ENTSCHEIDENDE PUNKT:
    # Weil wir zur Maximierung negiert haben, sind die Dualwerte aus
    # linprog fuer "<="-Bedingungen <= 0. Zurueckdrehen!
    # ------------------------------------------------------------------
    schattenpreise = -ergebnis.ineqlin.marginals

    print("=" * 74)
    print("      PRIMALE UND DUALE ERGEBNISANALYSE (SENSITIVITAET)")
    print("=" * 74)
    print(f"Maximaler Deckungsbeitrag: {max_gewinn:,.2f} EUR\n")

    print("--- Primalloesung: optimale Produktionsmengen ---")
    for name, menge in zip(PRODUKTE, mengen):
        print(f"  * {name}: {menge:8.2f} Stueck")
```

```
print("\n--- Duale Analyse: Schattenpreise und Auslastung ---")
for i, name in enumerate(RESSOURCEN):
    kapazitaet = KAPAZITAET[i]
    genutzt = kapazitaet - schlupf[i]
    auslastung = genutzt / kapazitaet * 100
    preis = schattenpreise[i]
    bindend = abs(schlupf[i]) < 1e-9

    print(f"\nRessource '{name}':")
    print(f"  Auslastung:    {genutzt:6.1f} / {kapazitaet:6.1f} h ({auslastung:5.1f} %)"
          f"  -> {'ENGPASS' if bindend else 'Reserve: %.1f h' % schlupf[i]}")
    print(f"  Schattenpreis: {preis:6.2f} EUR je zusaetzlicher Stunde")

    if preis > 1e-9:
        print(f"  >> Zusaetzliche Stunden lohnen sich bis zu einem Preis von "
              f"{preis:.2f} EUR/h.")
    else:
        print(f"  >> Kein Zukauf noetig - die Kapazitaet ist nicht erschoepft.")

# --- Konkrete Kaufentscheidung ---------------------------------------
preis_pruefung = schattenpreise[RESSOURCEN.index("Qualitätsprüfung")]
marge = preis_pruefung - ANGEBOTSPREIS_PRUEFSTUNDE
print("\n" + "-" * 74)
print(f"ENTSCHEIDUNG: Pruefstunden werden fuer "
      f"{ANGEBOTSPREIS_PRUEFSTUNDE:.2f} EUR/h angeboten.")
print(f"  Schattenpreis:  {preis_pruefung:6.2f} EUR/h")
print(f"  Angebotspreis:  {ANGEBOTSPREIS_PRUEFSTUNDE:6.2f} EUR/h")
print(f"  Marge:          {marge:+6.2f} EUR je zugekaufter Stunde")
print(f"  >> {'ZUKAUFEN' if marge > 0 else 'NICHT ZUKAUFEN'}")

# --- Numerische Gegenprobe: Kapazitaet wirklich um 1 erhoehen --------
kapazitaet_plus = KAPAZITAET.copy()
kapazitaet_plus[RESSOURCEN.index("Qualitätsprüfung")] += 1.0
gegenprobe = linprog(c=-DECKUNGSBEITRAG, A_ub=VERBRAUCH, b_ub=kapazitaet_plus,
                     bounds=[(0, None)] * 3, method="highs")
tatsaechlicher_zuwachs = -gegenprobe.fun - max_gewinn
print("\n--- Gegenprobe: Modell mit +1 Pruefstunde neu geloest ---")
print(f"  Vorhergesagt (Schattenpreis): {preis_pruefung:8.4f} EUR")
print(f"  Tatsaechlich gemessen:        {tatsaechlicher_zuwachs:8.4f} EUR")
assert abs(tatsaechlicher_zuwachs - preis_pruefung) < 1e-6, \
    "Schattenpreis stimmt nicht mit der Messung ueberein!"
print("  -> Der Schattenpreis ist bestaetigt.")

# --- Komplementaerer Schlupf pruefen ---------------------------------
for s, y in zip(schlupf, schattenpreise):
    assert abs(s * y) < 1e-6, "Komplementaerer Schlupf verletzt!"
print("\nPruefung: komplementaerer Schlupf fuer alle Ressourcen erfuellt.")
```

```
    print("=" * 74)


if __name__ == "__main__":
    analysiere()
```

Erwartete Ausgabe:

```
==========================================================================
      PRIMALE UND DUALE ERGEBNISANALYSE (SENSITIVITAET)
==========================================================================
Maximaler Deckungsbeitrag: 2,200.00 EUR

--- Primalloesung: optimale Produktionsmengen ---
  * Produkt 1:    40.00 Stueck
  * Produkt 2:    20.00 Stueck
  * Produkt 3:     0.00 Stueck

--- Duale Analyse: Schattenpreise und Auslastung ---

Ressource 'Montage':
  Auslastung:     100.0 /  120.0 h ( 83.3 %)  -> Reserve: 20.0 h
  Schattenpreis:    0.00 EUR je zusaetzlicher Stunde
  >> Kein Zukauf noetig - die Kapazitaet ist nicht erschoepft.

Ressource 'Lackieren':
  Auslastung:      80.0 /   80.0 h (100.0 %)  -> ENGPASS
  Schattenpreis:    6.67 EUR je zusaetzlicher Stunde
  >> Zusaetzliche Stunden lohnen sich bis zu einem Preis von 6.67 EUR/h.

Ressource 'Qualitätsprüfung':
  Auslastung:      50.0 /   50.0 h (100.0 %)  -> ENGPASS
  Schattenpreis:   33.33 EUR je zusaetzlicher Stunde
  >> Zusaetzliche Stunden lohnen sich bis zu einem Preis von 33.33 EUR/h.

--------------------------------------------------------------------------
ENTSCHEIDUNG: Pruefstunden werden fuer 18.00 EUR/h angeboten.
  Schattenpreis:   33.33 EUR/h
  Angebotspreis:   18.00 EUR/h
  Marge:          +15.33 EUR je zugekaufter Stunde
  >> ZUKAUFEN

--- Gegenprobe: Modell mit +1 Pruefstunde neu geloest ---
  Vorhergesagt (Schattenpreis):  33.3333 EUR
  Tatsaechlich gemessen:         33.3333 EUR
  -> Der Schattenpreis ist bestaetigt.
```

```
Pruefung: komplementaerer Schlupf fuer alle Ressourcen erfuellt.
======================================================================
```

Der Zukauf für 18 €/h ist hochprofitabel: Jede zusätzliche Prüfstunde bringt 33,33 € Deckungsbeitrag, also 15,33 € Reingewinn.

Code-Durchgang: die drei Sicherungen

Dieses Programm enthält drei Prüfungen, die genau diese Vorzeichenfalle zuverlässig aufdecken würden:

1. `schattenpreise = -ergebnis.ineqlin.marginals` — die eigentliche Korrektur.
2. **Numerische Gegenprobe:** Das Modell wird mit $b_3 + 1$ neu gelöst; der gemessene Zuwachs muss dem Schattenpreis entsprechen. Diese Prüfung ist unabhängig von jeder Vorzeichenkonvention und deshalb der zuverlässigste Test überhaupt.
3. **Komplementärer Schlupf:** $s_i \cdot y_i = 0$ muss für alle i gelten.

Übernehmen Sie dieses Muster in eigene Projekte. Der Aufwand ist eine Handvoll Zeilen, der Nutzen ist die Gewissheit, dass Ihre Handlungsempfehlung nicht das Gegenteil des Richtigen sagt.

Grenzen der Schattenpreis-Aussage

Der Schattenpreis gilt nur **lokal**, in einem begrenzten Intervall um die aktuelle Kapazität. Kauft man 200 Prüfstunden zu, wird irgendwann eine andere Ressource zum Engpass, und der Schattenpreis springt auf einen neuen Wert (oder auf null). Wer große Kapazitätsänderungen bewerten will, muss das Modell **neu rechnen** — nicht linear hochrechnen. Die Aufgabe *Gültigkeitsbereich des Schattenpreises* (Abschnitt 5.10) macht diesen Effekt sichtbar.

5.9 Wann Schattenpreise lügen: Entartung und Toleranzen

Der vorige Abschnitt hat gezeigt, wie wertvoll Schattenpreise sind. Dieser zeigt die zwei Fälle, in denen sie eine Entscheidung **nicht** tragen — und wie man beide erkennt, bevor jemand auf ihrer Grundlage eine Maschine kauft.

Fall 1: Entartung

Erinnern Sie sich an Abschnitt 2.4: In zwei Dimensionen legen **zwei** sich schneidende Geraden eine Ecke fest, in drei Dimensionen drei Ebenen, allgemein n Restriktionen bei n Variablen. Was passiert, wenn in einer Ecke **mehr** Nebenbedingungen aktiv sind als das Problem Variablen hat?

Geometrisch: Drei Geraden laufen zufällig durch denselben Punkt. Die Ecke ist immer noch eine Ecke — aber sie ist **überbestimmt**. Man nennt das **Entartung** (*Degeneriertheit*), und sie ist in der Praxis nicht die Ausnahme, sondern die Regel: Sie entsteht überall dort, wo Kapazitäten aus derselben Planung stammen und deshalb glatt aufeinander passen — Schichtlängen, Chargengrößen, runde Vertragsmengen.

Merksatz Bei einer entarteten Ecke ist der Plan eindeutig, der **Schattenpreis aber nicht**. Es gibt dann viele gleichermaßen korrekte Dualvektoren, und welchen Sie sehen, hängt davon ab, welchen Algorithmus der Solver zufällig benutzt hat.

Das ist keine theoretische Sorge. Dasselbe Modell, zwei Verfahren desselben Solvers:

Verfahren	Schattenpreise	Was das Management daraus liest
Dual Simplex	(0,33; 0,33; 0)	„Fräse und Schleiferei sind die Engpässe, die Prüfung ist wertlos.“
Innere-Punkte-Verfahren	(0; 0; 1,00)	„Die Prüfung ist der Engpass, Fräse und Schleiferei sind wertlos.“

Beide Zeilen sind mathematisch korrekt. Sie widersprechen sich trotzdem vollständig.

Fall 2: Toleranzen

Der zweite Fall ist unscheinbarer und deshalb häufiger. Wie stellt man fest, ob eine Ressource ausgelastet ist? Naheliegend wäre `if schlupf == 0`. Genau das ist falsch.

Solver rechnen mit endlicher Genauigkeit und brechen ab, sobald ihre eigene Toleranz erreicht ist (Abschnitt 2.7). Ein voll ausgelasteter Engpass meldet dann einen Schlupf von $-4{,}44 \cdot 10^{-16}$ statt exakt 0 — rechnerisch null, aber eben nicht `== 0.0`. Wer exakt vergleicht, übersieht ausgerechnet die Engpässe, die er sucht.

Toleranz	Was sie steuert	Typischer Standardwert
Primale Zulässigkeit	Wie weit darf **Ax** die Schranke **b** überschreiten?	10^{-7} (HiGHS), 10^{-6} (CP-SAT)
Duale Zulässigkeit (Optimalität)	Wie weit dürfen die reduzierten Kosten das falsche Vorzeichen haben?	10^{-7}
Ganzzahligkeit	Wie weit darf eine Ganzzahlvariable von der nächsten ganzen Zahl abweichen?	10^{-6} (Kapitel 6)

 Toleranzen kleiner zu drehen ist selten die Lösung. Wer `1e-12` verlangt, bekommt meist keinen genaueren Solver, sondern einen, der `INFEASIBLE` meldet oder nicht konvergiert. Die Ursache steckt fast immer in der Skalierung des Modells, nicht in der Einstellung — deshalb steht der Konditionsabschnitt in Kapitel 2 vor diesem hier.

Die ehrliche Auskunft: Schattenpreis-Spannen

Was tut man also, wenn das Modell entartet ist? Man meldet keinen Einzelwert, sondern die **Spanne**. Die Menge aller optimalen Dualvektoren ist selbst ein Polyeder:

$$\mathbf{A}^\top \mathbf{y} \geq \mathbf{c}, \qquad \mathbf{y} \geq \mathbf{0}, \qquad \mathbf{b}^\top \mathbf{y} = Z^*$$

Formel-Übersetzer

Mathematik	Alltagssprache
$\mathbf{A}^\top \mathbf{y} \geq \mathbf{c}$	„Die Preise müssen jedes Produkt decken“ — Dualzulässigkeit.
$\mathbf{b}^\top \mathbf{y} = Z^*$	„Und der Gesamtwert muss genau dem erreichten Gewinn entsprechen“ — starker Dualitätssatz als Gleichung.
min / max y_i darauf	„Wie klein und wie groß darf der Preis von Ressource i sein, ohne diese Regeln zu verletzen?“

Zwei zusätzliche LPs je Ressource — und Sie haben statt einer Zufallszahl ein belastbares Intervall.

```
#!/usr/bin/env python3

# Toleranzen_und_Entartung.py
"""
Kapitel LP: Zwei Faelle, in denen man Schattenpreisen NICHT trauen darf.

  1. Entartung (Degeneriertheit): Mehr Nebenbedingungen sind aktiv, als das
     Problem Variablen hat. Dann ist der Schattenpreis nicht eindeutig - zwei
     korrekte Solver liefern voellig verschiedene Werte fuer dasselbe Optimum.
  2. Toleranzen: "ausgelastet" heisst nie 'schlupf == 0', sondern immer
     'schlupf < toleranz'. Wer auf exakte Gleichheit prueft, baut Berichte,
     die zufaellig mal stimmen und mal nicht.

Teil 3 zeigt die professionelle Antwort auf Fall 1: Statt EINEN Schattenpreis
zu melden, berechnet man seine SPANNE ueber alle optimalen Dualloesungen.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
from scipy.optimize import linprog

# Ein bewusst entartetes Beispiel: drei Geraden schneiden sich in EINEM Punkt.
```

```
#   max x1 + x2
#   u.d.N.  x1 + 2*x2 <= 4        (A)
#          2*x1 +  x2 <= 4        (B)
#           x1 +  x2 <= 8/3       (C)  - laeuft genau durch die Ecke (4/3, 4/3)
# In zwei Dimensionen legen schon zwei Geraden eine Ecke fest. Hier sind drei
# aktiv - eine zu viel. Genau das ist Entartung.
C_ZIEL = np.array([-1.0, -1.0])          # linprog minimiert -> negiert
A_UB = np.array([[1.0, 2.0],
                 [2.0, 1.0],
                 [1.0, 1.0]])
B_UB = np.array([4.0, 4.0, 8.0 / 3.0])
NAMEN = ["A: Fraeszeit", "B: Schleifzeit", "C: Pruefzeit"]


def zeige_entartung() -> float:
    """Loest dasselbe LP mit zwei Verfahren und vergleicht die Dualwerte."""
    print("=" * 78)
    print("  1. ENTARTUNG: DERSELBE PLAN, GEGENSAETZLICHE SCHATTENPREISE")
    print("=" * 78)

    print(f"{'Verfahren':<26} {'x1':>7} {'x2':>7} {'Z*':>9}   Schattenpreise")
    print("-" * 78)

    dualwerte = {}
    for verfahren, beschreibung in [("highs-ds", "Dual Simplex"),
                                    ("highs-ipm", "Innere-Punkte-Verfahren")]:
        ergebnis = linprog(C_ZIEL, A_ub=A_UB, b_ub=B_UB, bounds=(0, None),
                           method=verfahren)
        if not ergebnis.success:
            raise RuntimeError(f"{verfahren}: {ergebnis.message}")
        y = -ergebnis.ineqlin.marginals
        dualwerte[verfahren] = y
        print(f"{beschreibung:<26} {ergebnis.x[0]:>7.3f} {ergebnis.x[1]:>7.3f} "
              f"{-ergebnis.fun:>9.4f}   {np.round(y, 4)}")

    print("-" * 78)
    print("Beide Zeilen sind RICHTIG: gleicher Plan, gleicher Zielwert, und beide")
    print("Dualvektoren erfuellen die Optimalitaetsbedingungen. Trotzdem sagen sie")
    print("das Gegenteil:")
    print(f"  Dual Simplex : {NAMEN[2]} ist wertlos, A und B sind je 0,33 EUR wert.")
    print(f"  Innere Punkte: {NAMEN[0]} und {NAMEN[1]} sind wertlos, C ist 1,00 EUR wert.")
    print()
    print("Wer auf dieser Grundlage eine Maschine kauft, hat eine 50:50-Chance -")
    print("abhaengig davon, welches Verfahren der Solver zufaellig gewaehlt hat.")

    return float(-linprog(C_ZIEL, A_ub=A_UB, b_ub=B_UB, bounds=(0, None)).fun)
```

```
def entartung_erkennen() -> None:
    """Der Test, der in jedes Auswertungsskript gehoert."""
    print("\n" + "=" * 78)
    print("  2. ENTARTUNG ERKENNEN - UND WARUM '== 0' DABEI VERSAGT")
    print("=" * 78)

    ergebnis = linprog(C_ZIEL, A_ub=A_UB, b_ub=B_UB, bounds=(0, None))
    schlupf = B_UB - A_UB @ ergebnis.x

    print(f"{'Nebenbedingung':<18} {'Schlupf':>16} {'== 0 ?':>9} "
          f"{'< 1e-7 ?':>10}")
    print("-" * 78)
    for name, s in zip(NAMEN, schlupf):
        print(f"{name:<18} {s:>16.3e} {str(s == 0.0):>9} {str(abs(s) < 1e-7):>10}")

    aktiv = int((np.abs(schlupf) < 1e-7).sum())
    variablen = A_UB.shape[1]
    print("-" * 78)
    print(f"Aktive Nebenbedingungen: {aktiv}, Variablen: {variablen}")
    if aktiv > variablen:
        print(f"=> ENTARTET. {aktiv} aktive Restriktionen bei nur {variablen} "
              "Variablen bedeuten:")
        print("   Der Schattenpreis ist nicht eindeutig. Melden Sie eine Spanne,")
        print("   keinen Einzelwert (siehe Teil 3).")
    else:
        print("=> nicht entartet, die Dualwerte sind eindeutig.")

    print("\nBeachten Sie die Spalte '== 0': Ein Schlupf von 4.44e-16 ist")
    print("rechnerisch null, aber nicht gleich 0.0. Wer mit '==' prueft,")
    print("uebersieht genau die Engpaesse, die er sucht.")


def schattenpreis_spanne(zielwert: float, toleranz: float = 1e-9
                         ) -> list[tuple[float, float]]:
    """Berechnet fuer jede Nebenbedingung die Spanne ihres Schattenpreises
    ueber ALLE optimalen Dualloesungen.

    Die Menge der optimalen Dualloesungen ist selbst ein Polyeder:

        A^T y >= c,   y >= 0,   b^T y = Z*

    (Dualzulaessigkeit plus starker Dualitaetssatz.) Minimiert und maximiert
    man darauf y_i, erhaelt man die exakten Grenzen. Das ist die ehrliche
    Auskunft an das Management: nicht 'die Stunde ist 0,33 EUR wert', sondern
    'zwischen 0,00 und 0,33 EUR - der Wert ist aus dem Modell nicht bestimmbar'.
    """
    m = A_UB.shape[0]
    # A^T y >= c  <=>  -A^T y <= -c ; Ziel war max c^T x, in linprog-Notation
```

```
    # steckt c mit negativem Vorzeichen in C_ZIEL.
    c_original = -C_ZIEL
    A_dual_ub = -A_UB.T
    b_dual_ub = -c_original

    spannen = []
    for i in range(m):
        richtung = np.zeros(m)
        richtung[i] = 1.0
        grenzen = []
        for vorzeichen in (1.0, -1.0):          # 1 = minimieren, -1 = maximieren
            ergebnis = linprog(
                vorzeichen * richtung,
                A_ub=A_dual_ub, b_ub=b_dual_ub,
                A_eq=B_UB.reshape(1, -1), b_eq=[zielwert],
                bounds=(0, None))
            if not ergebnis.success:
                raise RuntimeError(f"Spannenberechnung fehlgeschlagen: "
                                   f"{ergebnis.message}")
            grenzen.append(float(ergebnis.x[i]))
        spannen.append((min(grenzen), max(grenzen)))
    return spannen


def zeige_spanne(zielwert: float) -> None:
    print("\n" + "=" * 78)
    print("  3. DIE EHRLICHE AUSKUNFT: SCHATTENPREIS-SPANNEN")
    print("=" * 78)

    spannen = schattenpreis_spanne(zielwert)
    print(f"{'Nebenbedingung':<18} {'von':>10} {'bis':>10}   Aussage")
    print("-" * 78)
    for name, (unten, oben) in zip(NAMEN, spannen):
        if oben - unten < 1e-7:
            aussage = f"eindeutig {oben:.2f} EUR"
        elif oben < 1e-7:
            aussage = "sicher wertlos (kein Engpass)"
        else:
            aussage = "NICHT bestimmbar - Spanne melden!"
        print(f"{name:<18} {unten:>10.4f} {oben:>10.4f}   {aussage}")

    print("-" * 78)
    print("So berichtet man an Entscheider: 'Eine zusaetzliche Fraesstunde ist")
    print("zwischen 0,00 und 0,33 EUR wert - das Modell kann es nicht genauer")
    print("sagen, weil drei Engpaesse exakt gleichzeitig binden.' Das ist eine")
    print("brauchbare Aussage. Ein erfundener Einzelwert ist es nicht.")
```

```
if __name__ == "__main__":
    zielwert = zeige_entartung()
    entartung_erkennen()
    zeige_spanne(zielwert)

    print("\n" + "=" * 78)
    print("Merksatz: Pruefen Sie VOR jeder Sensitivitaetsaussage auf Entartung -")
    print("und vergleichen Sie Schlupfwerte nie mit '== 0', sondern mit einer")
    print("Toleranz.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  1. ENTARTUNG: DERSELBE PLAN, GEGENSAETZLICHE SCHATTENPREISE
==============================================================================
Verfahren                       x1      x2        Z*   Schattenpreise
------------------------------------------------------------------------------
Dual Simplex                 1.333   1.333    2.6667   [0.3333 0.3333 0.    ]
Innere-Punkte-Verfahren      1.333   1.333    2.6667   [-0. -0.  1.]
------------------------------------------------------------------------------
Beide Zeilen sind RICHTIG: gleicher Plan, gleicher Zielwert, und beide
Dualvektoren erfuellen die Optimalitaetsbedingungen. Trotzdem sagen sie
das Gegenteil:
  Dual Simplex : C: Pruefzeit ist wertlos, A und B sind je 0,33 EUR wert.
  Innere Punkte: A: Fraeszeit und B: Schleifzeit sind wertlos, C ist 1,00 EUR wert.

Wer auf dieser Grundlage eine Maschine kauft, hat eine 50:50-Chance -
abhaengig davon, welches Verfahren der Solver zufaellig gewaehlt hat.

==============================================================================
  2. ENTARTUNG ERKENNEN - UND WARUM '== 0' DABEI VERSAGT
==============================================================================
Nebenbedingung              Schlupf    == 0 ?   < 1e-7 ?
------------------------------------------------------------------------------
A: Fraeszeit              0.000e+00      True       True
B: Schleifzeit            0.000e+00      True       True
C: Pruefzeit             -4.441e-16     False       True
------------------------------------------------------------------------------
Aktive Nebenbedingungen: 3, Variablen: 2
=> ENTARTET. 3 aktive Restriktionen bei nur 2 Variablen bedeuten:
   Der Schattenpreis ist nicht eindeutig. Melden Sie eine Spanne,
   keinen Einzelwert (siehe Teil 3).

Beachten Sie die Spalte '== 0': Ein Schlupf von 4.44e-16 ist
rechnerisch null, aber nicht gleich 0.0. Wer mit '==' prueft,
uebersieht genau die Engpaesse, die er sucht.
```

```
================================================================================
  3. DIE EHRLICHE AUSKUNFT: SCHATTENPREIS-SPANNEN
================================================================================
Nebenbedingung            von        bis   Aussage
--------------------------------------------------------------------------------
A: Fraeszeit           0.0000     0.3333   NICHT bestimmbar - Spanne melden!
B: Schleifzeit         0.0000     0.3333   NICHT bestimmbar - Spanne melden!
C: Pruefzeit           0.0000     1.0000   NICHT bestimmbar - Spanne melden!
--------------------------------------------------------------------------------
So berichtet man an Entscheider: 'Eine zusaetzliche Fraesstunde ist
zwischen 0,00 und 0,33 EUR wert - das Modell kann es nicht genauer
sagen, weil drei Engpaesse exakt gleichzeitig binden.' Das ist eine
brauchbare Aussage. Ein erfundener Einzelwert ist es nicht.
```

□ Code-Durchgang

Stelle	Was passiert	Warum es zählt
`method="highs-ds"` vs. `"highs-ipm"`	zwei Verfahren desselben Solvers	Der Widerspruch entsteht **nicht** durch einen Fehler oder unterschiedliche Bibliotheken, sondern durch verschiedene, gleich gültige Wege zur selben Ecke.
`aktiv > variablen`	der Entartungstest	Drei Zeilen Code. Sie gehören in jedes Skript, das Schattenpreise ausgibt — und fehlen in fast allen.
Spalte `== 0 ?`	zeigt `False` bei $-4{,}44 \cdot 10^{-16}$	Der Beleg, warum Zulässigkeitsprüfungen immer eine Toleranz brauchen. Die Zahl ist echtes Solver-Rauschen, kein konstruiertes Beispiel.
`A_eq=B_UB.reshape(1, -1), b_eq=[zielwert]`	der starke Dualitätssatz **als Nebenbedingung**	Das ist der ganze Trick der Spannenrechnung: Wir suchen unter allen Dualvektoren, die denselben Zielwert erzeugen.
zwei `linprog`-Läufe je Ressource	Minimum und Maximum von y_i	Bei 50 Ressourcen sind das 100 kleine LPs — Sekunden. Verglichen mit einer Fehlinvestition ist das billig.

□ Typische Fehler

- **Schattenpreis als „der Wert“ berichten, ohne auf Entartung zu prüfen.** Der häufigste Weg, wie eine formal korrekte Optimierung zu einer falschen Investitionsentscheidung führt.
- **Schattenpreis weit extrapolieren.** y_i ist eine *Ableitung*: Sie gilt lokal. „Eine Stunde mehr bringt 50 €“ heißt nicht „200 Stunden mehr bringen 10 000 €“ — ab einem gewissen Punkt bindet eine andere Ressource, und der Wert bricht ein.
- **`if schlupf == 0`.** Siehe oben. Richtig ist `abs(schlupf) < 1e-7`.
- **Toleranzen herunterdrehen statt das Modell zu skalieren.** Behandelt das Symptom und erzeugt neue Probleme (Abschnitt 2.7).

5.10 Übungsaufgaben

Lösungen: Abschnitt A.5.

Aufgabe 5.1 □ — **Schlupf deuten.** Ein LP liefert $s = (0;\ 12{,}5;\ 0;\ 3)$ und $y = (4{,}2;\ 0;\ 9{,}8;\ 0)$. (a) Welche Ressourcen sind Engpässe? (b) Ist die Lösung mit dem Satz vom komplementären Schlupf verträglich? (c) In welche Ressource würden Sie zuerst investieren?

Aufgabe 5.2 □ — **Vorzeichen prüfen.** Ein Kollege maximiert Gewinn mit `linprog` und meldet: „Der Schattenpreis der Engpassmaschine ist -45 €.“ Was ist passiert, und wie lautet der korrekte Wert?

Aufgabe 5.3 □□ — **Simplex von Hand.** Lösen Sie mit dem Tableau-Verfahren vollständig von Hand:

$$\max\ 5x_1 + 4x_2 \quad \text{u. d. N.} \quad 6x_1 + 4x_2 \le 24,\quad x_1 + 2x_2 \le 6,\quad x_1, x_2 \ge 0$$

Geben Sie jedes Zwischentableau an und lesen Sie am Ende Lösung **und** Schattenpreise ab. Prüfen Sie mit `scipy.optimize.linprog`.

Aufgabe 5.4 □□ — **Duales Problem aufstellen.** Stellen Sie zum Modell aus der Aufgabe *Simplex von Hand* das duale Problem auf, lösen Sie es mit `linprog` und weisen Sie den starken Dualitätssatz numerisch nach.

Aufgabe 5.5 □□ — **Unbeschränktheit erkennen.** Was liefert `SimplexTableauSolver` für $\max x_1 + x_2$ u. d. N. $x_1 - x_2 \le 5$, $x_1, x_2 \ge 0$? Erklären Sie die Fehlermeldung geometrisch.

Aufgabe 5.6 □□□ — **Gültigkeitsbereich des Schattenpreises.** Erweitern Sie `Sensitivitaetsanalyse.py`: Lösen Sie das Modell für Prüfkapazitäten von 30 bis 120 Stunden (Schrittweite 5) und tragen Sie Gewinn und Schattenpreis gegen die Kapazität auf. (a) Ab welcher Kapazität fällt der Schattenpreis auf einen niedrigeren Wert? Warum? (b) Wie viele Stunden sollte der Betrieb bei einem Angebotspreis von 18 €/h **maximal** zukaufen? (c) Zeichnen Sie den Gewinnverlauf. Was für eine Kurvenform ergibt sich, und warum?

Aufgabe 5.7 □□□ — **Phase 1 ergänzen.** Erweitern Sie den `SimplexTableauSolver` um eine Phase-1-Rechnung mit künstlichen Variablen, sodass auch $b_i < 0$ verarbeitet werden kann. Testen Sie an:

$$\max\ 3x_1 + 2x_2 \quad \text{u. d. N.} \quad x_1 + x_2 \ge 4,\quad x_1 + 3x_2 \le 12,\quad x_1, x_2 \ge 0$$

5.11 Finde den Denkfehler

Finde den Denkfehler: Die 380 000-Euro-Maschine

Ein Werksleiter lässt die Engpässe seiner Fertigung analysieren. Das Skript liefert für den Lackierofen einen Schattenpreis von **50 €/Stunde**. Er rechnet:

> *„Ein zweiter Lackierofen bringt 2 000 zusätzliche Stunden im Jahr. Bei 50 € > je Stunde sind das 100 000 € Deckungsbeitrag pro Jahr. Die Maschine kostet 380 000 € > und amortisiert sich in 3,8 Jahren. Wir kaufen.“*

Das Skript, das die Zahl geliefert hat:

```
res = linprog(c, A_ub=A, b_ub=b, bounds=(0, None))
for i, y in enumerate(-res.ineqlin.marginals):
    if y > 0:
        print(f"Engpass {name[i]}: {y:.2f} EUR je zusaetzlicher Stunde")
```

Ihre Aufgabe:

(a) Der Schattenpreis ist $y_i = \partial Z^*/\partial b_i$. Welche Eigenschaft einer Ableitung übersieht die Hochrechnung auf 2 000 Stunden?

(b) Skizzieren Sie, wie der zusätzliche Deckungsbeitrag als Funktion der zusätzlichen Ofenstunden tatsächlich verläuft. Warum ist er nicht linear — und in welche Richtung liegt der Fehler?

(c) Zwei Prüfungen fehlen im Skript, bevor die Zahl 50 € überhaupt berichtet werden darf. Welche?

(d) Wie ermitteln Sie den **tatsächlichen** Nutzen von 2 000 Zusatzstunden — ohne jede Extrapolation? Schreiben Sie die drei Zeilen hin.

Auflösung: Abschnitt A.5.

5.12 Micro-Quiz

Micro-Quiz 5: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Eine Nebenbedingung hat den Schlupf $s_i = 12$ und den Schattenpreis $y_i = 0$. Was folgt daraus? (a) Der Solver hat einen Fehler gemacht — bei positivem Schlupf muss der Dualwert ebenfalls positiv sein. (b) Von dieser Ressource sind 12 Einheiten übrig; sie ist kein Engpass, und Zukauf bringt nichts. Das ist genau der komplementäre Schlupf. (c) Die Ressource ist voll ausgelastet, wird aber nicht benötigt.

2. Ihr Modell hat 5 Variablen. In der optimalen Ecke sind 7 Nebenbedingungen aktiv. Was bedeutet das für Ihren Bericht ans Management? (a) Nichts Besonderes — mehr

aktive Bedingungen heißt nur, dass das Modell gut ausgelastet ist. (b) Das Modell ist entartet: Der Plan ist eindeutig, die Schattenpreise sind es nicht. Statt Einzelwerten gehören Spannen in den Bericht. (c) Das Modell ist unlösbar, weil mehr Bedingungen als Variablen aktiv sind.

3. Ein Kollege prüft mit `if schlupf == 0.0`, welche Ressourcen Engpässe sind. Der Bericht weist einen offensichtlichen Engpass nicht aus. Warum? (a) Der Solver hat den Engpass übersehen. (b) Der Schlupf beträgt $-4{,}4 \cdot 10^{-16}$ statt exakt 0 — rechnerisch null, aber nicht gleich `0.0`. Richtig ist ein Vergleich gegen eine Toleranz. (c) Schlupfwerte sind bei Maximierungsproblemen immer negativ und müssen erst umgerechnet werden.

5.13 Selbsttest

Antworten: Anhang A.

1. Was misst eine Schlupfvariable, und was bedeutet der Wert 0?
2. Nach welchem Kriterium wählt der Simplex die Pivotzeile — und was würde passieren, wenn man stattdessen einfach die erste Zeile nähme?
3. Formulieren Sie den Satz vom komplementären Schlupf und erklären Sie ihn wirtschaftlich.
4. Warum sind die Dualwerte aus `scipy.optimize.linprog` bei einem Maximierungsproblem negativ?
5. Ihr Modell meldet „unbeschränkt“. Was ist die wahrscheinlichste Ursache?

5.14 Zusammenfassung

- **Schlupfvariablen** verwandeln Ungleichungen in Gleichungen — und messen nebenbei ungenutzte Kapazität.
- **Der Simplex** wandert von Ecke zu Ecke, immer bergauf, und stoppt, wenn keine Verbesserung mehr möglich ist. Im Beispiel genügten 2 Schritte statt 5 Eckenprüfungen.
- **Dualität** verwandelt „Wie viel produziere ich?“ in „Was sind meine Ressourcen wert?“ — beide Probleme haben denselben Optimalwert.
- **Schattenpreise** beantworten die Managementfrage schlechthin: Wo ist der Engpass, und was ist seine Beseitigung wert?
- **Vorzeichen prüfen!** Nach einer Negation der Zielfunktion müssen die Dualwerte zurücknegiert werden. Die numerische Gegenprobe („Kapazität um 1 erhöhen und neu lösen“) ist die sicherste Kontrolle.

Ausblick. Kapitel 6 bricht mit der Annahme beliebiger Teilbarkeit: Was, wenn man nur ganze Maschinen kaufen kann? Wir werden sehen, dass Runden der LP-Lösung nicht nur ungenau, sondern grundsätzlich falsch ist — und lernen Branch-and-Bound kennen.

Kapitel 6: Gemischt-ganzzahlige Optimierung — Diskrete Entscheidungen und Branch-and-Bound

Kapitel auf einen Blick

Worum geht es? Um Entscheidungen, die nicht teilbar sind: ganze Maschinen, Ja/Nein, „höchstens fünf davon“. Wir zeigen mit einem konkreten Gegenbeispiel, warum Runden **falsch** ist, und lernen die Modellierungstricks, mit denen sich logische Regeln in lineare Ungleichungen übersetzen lassen.

Voraussetzungen: Kapitel 5 (LP, Relaxation, Schranken).

Danach können Sie: Fixkosten, Entweder-Oder, Wenn-Dann und Kardinalitätsgrenzen modellieren, ein MILP mit CP-SAT oder HiGHS lösen, den MIP-Gap als Zusage an den Auftraggeber lesen — und ein Ergebnis erkennen, das nur deshalb gut aussieht, weil das Big-M zu groß gewählt war.

Zeitbedarf: ca. 6,5 Stunden.

Programme:
`Runden_Gegenbeispiel.py`
`Rucksack.py`
`MILP_Portfolio_Fixgebuehren.py`
`Solverstatus_und_Gap.py`
`Warmstart_Effekt.py`
`Big_M_Falle.py`

Notebook: `Notebooks_04/milp.ipynb`

6.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Welches Lager eröffnen wir?

Ein Händler muss 100 Paletten pro Woche ausliefern und kann dafür bis zu drei Lager eröffnen. Jedes verursacht Fixkosten, egal wie viel darüber läuft.

Lager	Fixkosten/Woche	Kapazität	Kosten je Palette
Nord	180 €	40	6,00 €
Mitte	260 €	70	4,50 €
Süd	300 €	90	3,80 €

```
import numpy as np
from scipy.optimize import milp, LinearConstraint, Bounds

fix = np.array([180.0, 260.0, 300.0])       # Fixkosten je Lager
kap = np.array([40.0, 70.0, 90.0])          # Kapazitaet je Lager
```

```
var = np.array([6.0, 4.5, 3.8])            # variable Kosten je Palette

A = np.zeros((4, 6))                       # Variablen: 3x Menge, 3x Schalter
A[0, :3] = 1.0                             # Summe der Mengen = 100
for i in range(3):
    A[i + 1, i], A[i + 1, 3 + i] = 1.0, -kap[i]      # Menge <= Kapazitaet *
 ↪  Schalter
nb = LinearConstraint(A, [100, -np.inf, -np.inf, -np.inf], [100, 0, 0, 0])
grenzen = Bounds([0] * 6, list(kap) + [1, 1, 1])

for name, ganzzahlig in [("LP-Relaxation", [0] * 6), ("MILP", [0, 0, 0, 1, 1, 1])]:
    r = milp(c=np.concatenate([var, fix]), constraints=[nb],
             integrality=ganzzahlig, bounds=grenzen)
    print(f"{name:14} {r.fun:7.2f} EUR   Mengen {np.round(r.x[:3], 1)}   "
          f"Schalter {np.round(r.x[3:], 3)}")
```

Ausgabe:

```
LP-Relaxation   724.14 EUR   Mengen [ 0. 10. 90.]   Schalter [-0.     0.143  1.   ]
MILP            882.00 EUR   Mengen [10.  0. 90.]   Schalter [ 1. -0.     1.   ]
```

Sehen Sie sich die Schalter der ersten Zeile an: 0,143. Die LP-Relaxation eröffnet Lager Mitte zu 14,3 % — und zahlt dafür nur 14,3 % der Fixkosten. So etwas gibt es in der Wirklichkeit nicht. Ein Lager ist offen oder zu.

Und jetzt der Punkt, um den es in diesem ganzen Kapitel geht — **was passiert, wenn man diese 0,143 rundet?**

Vorgehen	Ergebnis
Abrunden auf $y = (0, 0, 1)$	Nur Süd offen: 90 Paletten Kapazität für 100 Paletten Bedarf → **unzulässig**
Aufrunden auf $y = (0, 1, 1)$	Zulässig, aber **947 €** — 65 € teurer als nötig
MILP-Optimum $y = (1, 0, 1)$	**882 €** — und es öffnet **Nord**, ein Lager, das die Relaxation gar nicht vorgeschlagen hatte

Runden führt hier also in beide Richtungen in die Irre: einmal in die Unzulässigkeit, einmal in unnötige Kosten. Und der entscheidende Punkt ist der dritte: Die ganzzahlige Lösung ist **strukturell anders**. Sie öffnet ein anderes Lager. Keine noch so geschickte Rundung der LP-Lösung hätte darauf kommen können.

> **Merksatz** Die LP-Relaxation ist keine ungefähre Antwort, die man nur noch glattziehen muss. Sie ist eine Antwort auf eine **andere Frage** — nämlich die, bei der man Lager auch zu 14,3 % eröffnen darf. Ihr Wert liegt nicht in ihrer Lösung, sondern in ihrer **Schranke**: Weniger als 724,14 € kann die richtige Antwort nicht kosten.

Genau diese Schranke ist der Hebel, mit dem Branch-and-Bound arbeitet. Wie, zeigt der Abschnitt Abschnitt 6.4.

6.2 Lernziele

Nach diesem Kapitel können Sie …

1. … an einem Zahlenbeispiel belegen, warum Runden einer LP-Lösung scheitert.
2. … den Ablauf von Branch-and-Bound erklären und einen kleinen Suchbaum von Hand zeichnen.
3. … die vier klassischen Big-M-Muster anwenden (Fixkosten, Entweder-Oder, Wenn-Dann, Kardinalität).
4. … begründen, warum M so klein wie möglich gewählt werden muss.
5. … ein Portfolioproblem mit Ordergebühren und Höchstzahl an Positionen lösen.
6. … **Incumbent, Schranke und MIP-Gap** unterscheiden und aus einem abgebrochenen Solverlauf eine belastbare Aussage ableiten.
7. … alle Solver-Statusfälle explizit behandeln, statt `OPTIMAL` vorauszusetzen.
8. … einen **Warm-Start** setzen — und messen, ob er auf Ihrer Problemklasse überhaupt etwas bringt.

6.3 Warum Runden fundamental scheitert

In der realen Welt sind viele Entscheidungen nicht teilbar. Man kann nicht 0,47 Flugzeuge kaufen, keine halbe Lagerhalle bauen, keinen Mitarbeiter zu 38 % einstellen. Noch wichtiger: Logische Schalter — *„Wenn Fabrik A gebaut wird, muss auch Lager B gebaut werden“* — brauchen diskrete Zustände.

Mixed-Integer Linear Programming (MILP), deutsch *gemischt-ganzzahlige lineare Optimierung*, erweitert das LP um ganzzahlige ($\mathbb{Z}$) und binäre ($\{0,1\}$) Variablen.

Der verlockende Gedanke lautet: *„Wir lösen das Problem kontinuierlich und runden.“* Eine verbreitete, aber unbelegte Behauptung dazu lautet, man verliere dadurch „oft 20–50 % des Gewinns“. Rechnen wir es nach, statt es zu behaupten.

Handrechnung 6.1: Ein Gegenbeispiel, das Sie im Kopf prüfen können

$$\max\ Z = x_1 + x_2 \quad \text{u. d. N.} \quad 2x_1 + 2x_2 \le 3, \quad x_1, x_2 \in \{0, 1, 2, \dots\}$$

LP-Relaxation (Ganzzahligkeit weggelassen): Jede Kombination mit $x_1 + x_2 = 1{,}5$ ist optimal, z. B. $x_1 = x_2 = 0{,}75$ mit $Z_{LP} = 1{,}5$.

Aufrunden auf $(1,1)$: $2 \cdot 1 + 2 \cdot 1 = 4 > 3 \rightarrow$ **unzulässig**. **Abrunden** auf $(0,0)$: zulässig, aber $Z = 0$. **Wahres ganzzahliges Optimum:** $(1,0)$ oder $(0,1)$ mit $Z_{IP} = 1$.

Das Abrunden verliert hier **100 %** des erreichbaren Werts. Das ist kein exotischer Sonderfall, sondern typisch, sobald die Zahlen klein sind — und bei Ja/Nein-Entscheidungen sind sie immer klein.

Die drei Gründe systematisch:

1. **Verletzung von Nebenbedingungen.** Aufrunden überschreitet Kapazitätsgrenzen — die Lösung wird unzulässig.

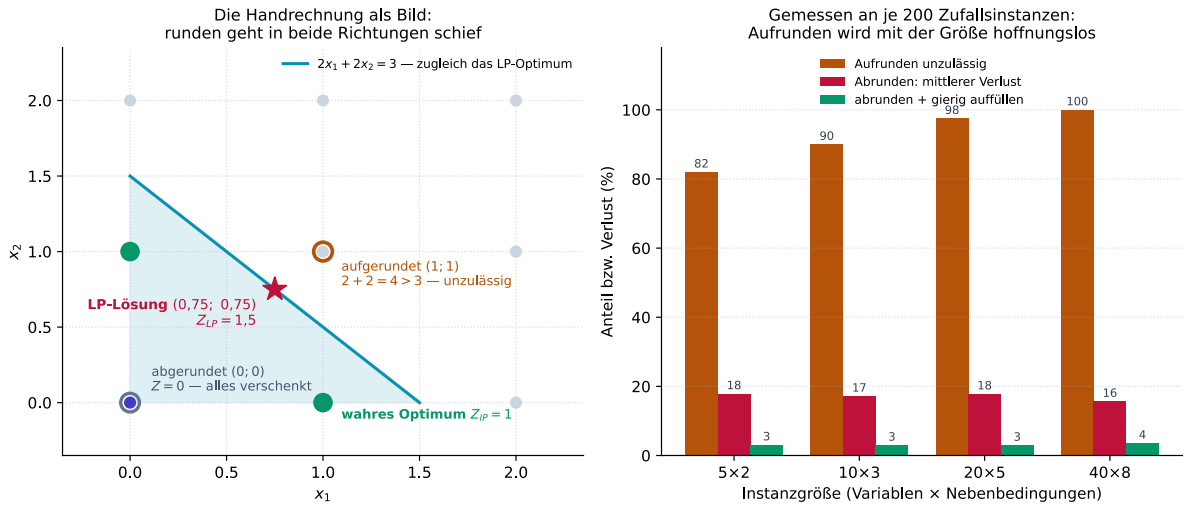

Abbildung 10: Abb. 6.1: Links die Handrechnung von unten als Bild: Die Zielfunktion läuft parallel zur Restriktion, also ist die ganze Kante optimal. Aufrunden auf $(1; 1)$ führt aus dem zulässigen Bereich heraus, Abrunden auf $(0; 0)$ auf den Wert null. Rechts die Messung aus `Runden_Gegenbeispiel.py` über je 200 Zufallsinstanzen. Erzeugt von `bilder_04/erzeuge_runden.py`.

2. **Suboptimalität.** Abrunden verschenkt Kapazität. Der Verlust ist umso größer, je kleiner die Zahlen sind.
3. **Distanz im Raum.** In hohen Dimensionen liegt der beste ganzzahlige Punkt oft **weit entfernt** von der LP-Lösung — er ist durch Runden gar nicht erreichbar.

Das folgende Programm quantifiziert den Effekt über viele Zufallsinstanzen.

```
#!/usr/bin/env python3

# Runden_Gegenbeispiel.py
"""
Kapitel MILP: Wie schlecht ist Runden wirklich?

Ersetzt die unbelegte Behauptung "20-50 % Verlust" durch eine Messung ueber
viele Zufallsinstanzen.
"""

import numpy as np
from scipy.optimize import linprog


def erzeuge_instanz(n, m, rng):
    """Zufaelliges Rucksack-aehnliches MILP mit kleinen Zahlen."""
    c = rng.integers(3, 20, size=n).astype(float)          # Ertraege
    A = rng.integers(1, 9, size=(m, n)).astype(float)      # Verbraeuche
    b = (A.sum(axis=1) * rng.uniform(0.25, 0.45)).round()  # knappe Kapazitaeten
    return c, A, b
```

```
def loese_lp(c, A, b, ganzzahlig=False):
    """LP-Relaxation oder exaktes MILP ueber HiGHS."""
    n = len(c)
    ergebnis = linprog(
        c=-c, A_ub=A, b_ub=b, bounds=[(0, None)] * n,
        integrality=np.ones(n) if ganzzahlig else None,
        method="highs")
    return (-ergebnis.fun, ergebnis.x) if ergebnis.success else (None, None)


def abrunden_und_reparieren(x_lp, c, A, b):
    """Naive Strategie: abrunden, dann gierig auffuellen, solange zulaessig."""
    x = np.floor(x_lp + 1e-9)
    verbessert = True
    while verbessert:                       # gierig auffuellen
        verbessert = False
        for j in np.argsort(-c):            # bester Ertrag zuerst
            kandidat = x.copy()
            kandidat[j] += 1
            if np.all(A @ kandidat <= b + 1e-9):
                x = kandidat
                verbessert = True
                break
    return c @ x, x


if __name__ == "__main__":
    rng = np.random.default_rng(2026)
    print("=" * 82)
    print("  WIE TEUER IST RUNDEN? (200 Zufallsinstanzen je Groesse)")
    print("=" * 82)
    print(f"{'n x m':>8} | {'Aufrunden unzul.':>17} | {'Abrunden: mittl.':>17} | "
          f"{'schlimmster':>12} | {'gierig':>8}")
    print(f"{'':>8} | {'':>17} | {'Verlust':>17} | {'Fall':>12} | {'Verlust':>8}")
    print("-" * 82)

    for n, m in [(5, 2), (10, 3), (20, 5), (40, 8)]:
        unzulaessig = 0
        verluste_ab, verluste_gierig = [], []

        for _ in range(200):
            c, A, b = erzeuge_instanz(n, m, rng)
            z_lp, x_lp = loese_lp(c, A, b, ganzzahlig=False)
            z_ip, _ = loese_lp(c, A, b, ganzzahlig=True)
            if z_lp is None or z_ip is None or z_ip <= 0:
                continue

            # Variante 1: aufrunden
```

```
            x_auf = np.ceil(x_lp - 1e-9)
            if np.any(A @ x_auf > b + 1e-9):
                unzulaessig += 1

            # Variante 2: abrunden
            x_ab = np.floor(x_lp + 1e-9)
            verluste_ab.append(1.0 - (c @ x_ab) / z_ip)

            # Variante 3: abrunden + gierig auffuellen
            z_gierig, _ = abrunden_und_reparieren(x_lp, c, A, b)
            verluste_gierig.append(1.0 - z_gierig / z_ip)

        print(f"{n:>3} x {m:<3} | {unzulaessig/2:>15.1f} % | "
              f"{np.mean(verluste_ab)*100:>15.1f} % | "
              f"{np.max(verluste_ab)*100:>10.1f} % | "
              f"{np.mean(verluste_gierig)*100:>6.1f} %")

    print("-" * 82)
    print("Lesart: 'Aufrunden unzul.' = Anteil der Faelle, in denen die aufgerundete")
    print("Loesung eine Nebenbedingung verletzt. 'Verlust' = Abstand zum exakten Optimum.")
    print("=" * 82)
```

Erwartete Ausgabe:

```
==================================================================================
  WIE TEUER IST RUNDEN? (200 Zufallsinstanzen je Groesse)
==================================================================================
   n x m |  Aufrunden unzul. |  Abrunden: mittl. |  schlimmster |   gierig
         |                   |           Verlust |         Fall |  Verlust
----------------------------------------------------------------------------------
  5 x 2  |            82.0 % |            17.9 % |       100.0 % |    3.1 %
 10 x 3  |            90.0 % |            17.2 % |       100.0 % |    3.0 %
 20 x 5  |            97.5 % |            17.8 % |        50.0 % |    2.9 %
 40 x 8  |           100.0 % |            15.6 % |        35.7 % |    3.6 %
----------------------------------------------------------------------------------
Lesart: 'Aufrunden unzul.' = Anteil der Faelle, in denen die aufgerundete
Loesung eine Nebenbedingung verletzt. 'Verlust' = Abstand zum exakten Optimum.
==================================================================================
```

Was die Zahlen sagen:

- **Aufrunden ist fast immer unzulässig** (82 % bei kleinen, 100 % bei größeren Instanzen). Es ist keine Strategie, sondern ein Fehler — und zwar einer, der mit der Problemgröße *zunimmt*, weil mehr Variablen mehr Gelegenheiten bieten, eine Kapazität zu sprengen.
- **Abrunden verliert im Mittel 16–18 %**, in den schlimmsten gemessenen Fällen **100 %** (dann bleibt das Ergebnis bei null). Die eingangs zitierte Behauptung „20–50 %“ lag zu hoch; der gemessene Mittelwert liegt darunter, die Extremfälle darüber. Beides ist erst durch die Messung sichtbar.

- **Abrunden plus gieriges Auffüllen** ist deutlich besser (rund 3 % Verlust) — aber weiterhin ohne jede Garantie und mit zusätzlichem Implementierungsaufwand. Ein exakter MILP-Solver liefert die Optimalität gratis und dazu einen **Beweis** dafür.

  Zur Ehrlichkeit von Messungen Diese Zahlen gelten für **diese** Instanzfamilie (kleine ganzzahlige Erträge, knappe Kapazitäten). Bei großen Stückzahlen — etwa 4 700 statt 4,7 produzierten Einheiten — ist der relative Rundungsfehler viel kleiner, und Runden wird zu einer brauchbaren Heuristik. **Die Faustregel lautet deshalb nicht „Runden ist immer schlecht", sondern: Je kleiner die Zahlen und je knapper die Kapazitäten, desto teurer das Runden.** Bei Ja/Nein-Variablen — dem häufigsten Fall — sind die Zahlen maximal klein, und Runden ist sinnlos.

  Merksatz Runden ist kein Näherungsverfahren mit kontrollierbarem Fehler, sondern ein Verfahren ohne jede Garantie. Wenn Ganzzahligkeit zum Problem gehört, gehört sie ins Modell.

6.4 Branch-and-Bound

MILP-Probleme sind **NP-schwer**. Der Standardansatz moderner Solver (HiGHS, SCIP, Gurobi) heißt **Branch-and-Cut** — Branch-and-Bound plus Schnittebenen.

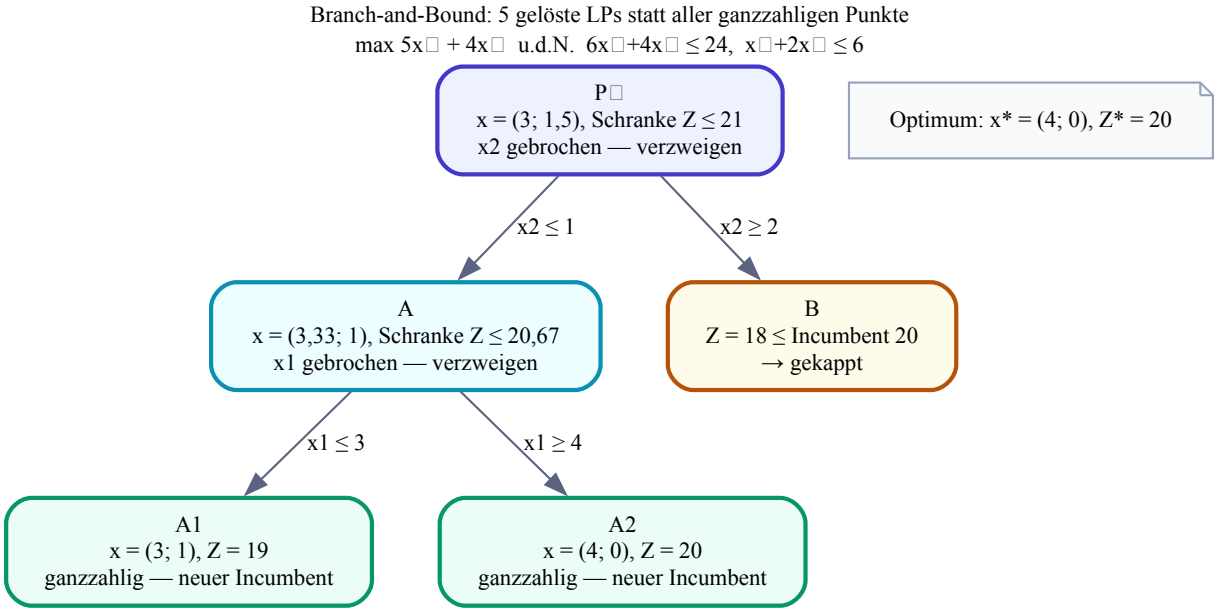

Abbildung 11: Abb. 6.2: Der Suchbaum der Handrechnung weiter unten — nicht gezeichnet, sondern gerechnet: Das Skript löst die LP-Relaxationen und kappt nach denselben drei Regeln, die oben stehen. Ast B wird verworfen, weil sein Wert 18 den Incumbent 20 aus Ast A nicht schlägt. Erzeugt von `bilder_04/erzeuge_branch_and_bound.py`.

Die drei Phasen

1. **LP-Relaxation.** Ganzzahligkeit vorübergehend weglassen und das LP lösen. Der Wert ist eine **obere Schranke** (bei Maximierung): Besser als das kann kein ganzzahliger Punkt sein, denn jeder ganzzahlige Punkt ist auch ein zulässiger LP-Punkt.
2. **Branching.** Hat eine ganzzahlige Variable einen gebrochenen Wert, etwa $x_1 = 2{,}7$, wird das Problem in zwei disjunkte Teilprobleme zerlegt:

   $$\text{Ast 1: } x_1 \le 2 \qquad \text{und} \qquad \text{Ast 2: } x_1 \ge 3$$

   Der Wert 2,7 selbst wird dadurch ausgeschlossen — kein ganzzahliger Punkt geht verloren.
3. **Bounding und Pruning.** Ein Ast wird abgeschnitten (*pruned*), wenn:
   - das Teilproblem unzulässig ist,
   - seine LP-Schranke schlechter ist als die beste bereits gefundene ganzzahlige Lösung (der **Incumbent**),
   - oder die LP-Lösung bereits ganzzahlig ist (dann ist dieser Ast fertig).

Zusätzlich fügen Solver **Schnittebenen** (*cutting planes*, z. B. Gomory-Cuts) hinzu: gültige Ungleichungen, die gebrochene Bereiche wegschneiden, ohne einen einzigen zulässigen ganzzahligen Punkt zu entfernen.

Handrechnung 6.2: Ein Suchbaum von Hand

$$\max\ 5x_1 + 4x_2 \quad \text{u. d. N.} \quad 6x_1 + 4x_2 \le 24, \quad x_1 + 2x_2 \le 6, \quad x_1, x_2 \in \mathbb{Z}_{\ge 0}$$

Wurzel P_0: LP-Lösung $x = (3;\ 1{,}5)$, $Z_{LP} = 21$. → obere Schranke 21. x_2 ist gebrochen → verzweigen über x_2.

Ast A ($x_2 \le 1$): LP liefert $x = (3{,}33;\ 1)$, $Z = 20{,}67$. Noch gebrochen (x_1) → weiter verzweigen. * **A1 ($x_2 \le 1$, $x_1 \le 3$):** $x = (3;\ 1)$, $Z = 19$ → **ganzzahlig!** Incumbent = 19. * **A2 ($x_2 \le 1$, $x_1 \ge 4$):** $x = (4;\ 0)$, $Z = 20$ → **ganzzahlig!** Incumbent verbessert auf 20.

Ast B ($x_2 \ge 2$): LP liefert $x = (2{,}67;\ 2)$, $Z = 21{,}33\ldots$ halt: Prüfen wir $6x_1 + 4x_2 \le 24$ mit $x_2 = 2$: $6x_1 \le 16$, also $x_1 \le 2{,}67$; und $x_1 + 2x_2 \le 6$ gibt $x_1 \le 2$. Also $x_1 = 2$, $Z = 5{\cdot}2 + 4{\cdot}2 = 18$ → ganzzahlig, aber **schlechter** als der Incumbent 20 → verworfen.

Ergebnis: $x^* = (4;\ 0)$, $Z^* = 20$. Wir haben **fünf** LPs gelöst — P_0, A, A1, A2 und B — statt alle ganzzahligen Punkte aufzuzählen. Das Skript `bilder_04/erzeuge_branch_and_bound.py` rechnet genau diesen Baum nach und zählt mit.

Beachten Sie: Der Wert $Z_{LP} = 21$ der Wurzel ist die **Optimalitätslücke**-Referenz. Nach dem Finden von 20 weiß der Solver: Das Optimum liegt zwischen 20 und 21 — eine Lücke von 5 %. In der Praxis bricht man oft genau hier ab (siehe Kapitel 22).

6.5 Modellierungstricks: Big-M und logische Bedingungen

Hier kommt das Handwerkszeug, das MILP so mächtig macht: Man übersetzt Geschäftsregeln in lineare Ungleichungen — mithilfe binärer Hilfsvariablen $y \in \{0,1\}$ und einer hinreichend großen Konstante M.

Muster 1 — Fixkosten / Aktivierungsschalter

Regel: Wird Maschine j genutzt ($x_j > 0$), fallen Rüstkosten F_j an.

$$x_j \le M \cdot y_j, \qquad y_j \in \{0,1\},\ x_j \ge 0$$

- Ist $y_j = 0$: erzwingt $x_j \le 0$, zusammen mit $x_j \ge 0$ also $x_j = 0$.
- Ist $y_j = 1$: gilt nur $x_j \le M$ — die Kapazität ist freigegeben.

In der Zielfunktion erscheint dann $+F_j y_j$ als Kostenterm.

Muster 2 — Entweder-Oder (*disjunctive constraints*)

Regel: Es muss **entweder** $f(\mathbf{x}) \le b_1$ **oder** $g(\mathbf{x}) \le b_2$ gelten.

$$f(\mathbf{x}) \le b_1 + M(1-y), \qquad g(\mathbf{x}) \le b_2 + My, \qquad y \in \{0,1\}$$

- $y = 1$: erste Bedingung aktiv, zweite durch $+M$ praktisch außer Kraft.
- $y = 0$: umgekehrt.

Muster 3 — Wenn-Dann (Implikation)

Regel: Wenn $y_1 = 1$, dann muss auch $y_2 = 1$ sein.

$$y_1 \le y_2$$

Kein Big-M nötig — das ist die eleganteste Formulierung überhaupt. Prüfen Sie die vier Fälle: $(0,0)$ □, $(0,1)$ □, $(1,1)$ □, $(1,0)$ □ — genau die verbotene Kombination wird ausgeschlossen.

Muster 4 — Kardinalität („höchstens K aus N“)

$$\sum_{j=1}^{N} y_j \le K$$

Muster 5 — Semikontinuierlich („entweder 0 oder mindestens L“)

Regel: Eine Position ist entweder gar nicht besetzt oder mit mindestens L Euro.

$$L\,y_j \le x_j \le U\,y_j$$

Dieses Muster brauchen wir gleich für die Mindestordergröße.

> □ **Die Big-M-Falle: M so klein wie möglich!**
>
> M muss groß genug sein, um die Bedingung wirklich außer Kraft zu setzen — aber **jedes Übermaß kostet Laufzeit**. Der Grund liegt in der LP-Relaxation: Mit $M = 10^9$ und $x_j \le$

$10^9 y_j$ genügt schon $y_j = 10^{-9} \cdot x_j$, um die Bedingung zu erfüllen. Die Relaxation ist dann extrem schwach, die obere Schranke nutzlos, und Branch-and-Bound muss praktisch alles durchsuchen.

Regel: Wählen Sie M als kleinste Zahl, die nachweislich nie bindet — meist eine ohnehin vorhandene Kapazitätsgrenze. Im Portfoliobeispiel unten ist das die Obergrenze pro Position (40 000 €), **nicht** eine willkürliche Million.

Gemessen an der Standortplanung aus `Big_M_Falle.py` (12 Lager, 40 Kunden, HiGHS):

M	Knoten im Suchbaum	Zielwert
= Lagerkapazität (knappstmöglich)	**13**	richtig
10× zu groß	51	richtig
10^4× zu groß	51	richtig
10^7× zu groß	51	richtig

Zwei Dinge sind daran bemerkenswert. Erstens: Das knappe M braucht **viermal weniger Knoten** — der Effekt ist real. Zweitens: Ab dem Zehnfachen wird es nicht mehr schlimmer, weil HiGHS' *Presolve* das übergroße M selbst auf die implizit vorhandene Schranke zurechtstutzt. Verlassen Sie sich darauf **nicht**: Presolve kann das nur, wenn eine solche Schranke im Modell überhaupt herleitbar ist. Und wenn nicht, wird es richtig unangenehm — siehe Abschnitt 6.10.

6.6 Beispiel: Das Rucksackproblem

Bevor wir zum Portfolio kommen, das klassische Einstiegsproblem — kurz, verständlich und überall wiederzuerkennen.

```python
#!/usr/bin/env python3

# Rucksack.py
"""
Kapitel MILP: Das Rucksackproblem (Knapsack).
Zeigt LP-Relaxation, Branch-and-Bound-Ergebnis und den Preis der Ganzzahligkeit
an einem Beispiel, das man vollstaendig im Kopf nachvollziehen kann.
"""

import numpy as np
from scipy.optimize import linprog

GEGENSTAENDE = ["Zelt", "Schlafsack", "Kocher", "Kamera", "Buch", "Wasserfilter", "Seil"]
NUTZEN =  np.array([40.0, 35.0, 20.0, 30.0,  8.0, 25.0, 12.0])
GEWICHT = np.array([ 6.0,  4.0,  3.0,  2.0,  1.0,  2.0,  3.0])
KAPAZITAET = 11.0        # kg
```

```
def loese(ganzzahlig: bool):
    n = len(NUTZEN)
    ergebnis = linprog(
        c=-NUTZEN, A_ub=[GEWICHT], b_ub=[KAPAZITAET],
        bounds=[(0, 1)] * n,                      # jedes Teil hoechstens einmal
        integrality=np.ones(n) if ganzzahlig else None,
        method="highs")
    return -ergebnis.fun, ergebnis.x


if __name__ == "__main__":
    z_lp, x_lp = loese(ganzzahlig=False)
    z_ip, x_ip = loese(ganzzahlig=True)

    print("=" * 72)
    print(f"  RUCKSACKPROBLEM  (Kapazitaet {KAPAZITAET:.0f} kg)")
    print("=" * 72)
    print(f"{'Gegenstand':<14} {'Nutzen':>7} {'kg':>5} {'Nutzen/kg':>10} "
          f"{'LP':>7} {'MILP':>6}")
    print("-" * 72)
    for i, name in enumerate(GEGENSTAENDE):
        # int(round(...)) statt Format "%.0f": vermeidet die Ausgabe "-0"
        print(f"{name:<14} {NUTZEN[i]:>7.0f} {GEWICHT[i]:>5.0f} "
              f"{NUTZEN[i]/GEWICHT[i]:>10.2f} {x_lp[i]:>7.2f} {int(round(x_ip[i])):>6d}")
    print("-" * 72)
    print(f"{'Gesamtnutzen':<14} {'':<7} {'':<5} {'':<10} {z_lp:>7.2f} {z_ip:>6.0f}")
    print(f"{'Gesamtgewicht':<14} {'':<7} {'':<5} {'':<10} "
          f"{GEWICHT @ x_lp:>7.2f} {GEWICHT @ x_ip:>6.0f}")
    print("-" * 72)
    print(f"Obere Schranke aus der LP-Relaxation: {z_lp:.2f}")
    print(f"Bestes ganzzahliges Ergebnis:         {z_ip:.0f}")
    print(f"Preis der Ganzzahligkeit:             {z_lp - z_ip:.2f} "
          f"({(1 - z_ip/z_lp)*100:.1f} %)")

    gebrochen = [GEGENSTAENDE[i] for i in range(len(NUTZEN)) if 1e-6 < x_lp[i] < 1 - 1e-6]
    print(f"\nIn der LP-Loesung nur teilweise eingepackt: {gebrochen}")
    print("Genau hier wuerde Branch-and-Bound verzweigen:")
    print(f"  Ast 1: {gebrochen[0]} bleibt ganz zuhause (x=0)")
    print(f"  Ast 2: {gebrochen[0]} kommt ganz mit    (x=1)")
    print("=" * 72)
```

Erwartete Ausgabe:

```
========================================================================
  RUCKSACKPROBLEM  (Kapazitaet 11 kg)
========================================================================
Gegenstand      Nutzen    kg  Nutzen/kg      LP   MILP
```

```
------------------------------------------------------------------------
Zelt                40     6       6.67    0.00      0
Schlafsack          35     4       8.75    1.00      1
Kocher              20     3       6.67    0.67      1
Kamera              30     2      15.00    1.00      1
Buch                 8     1       8.00    1.00      0
Wasserfilter        25     2      12.50    1.00      1
Seil                12     3       4.00    0.00      0
------------------------------------------------------------------------
Gesamtnutzen                                 111.33    110
Gesamtgewicht                                 11.00     11
------------------------------------------------------------------------
Obere Schranke aus der LP-Relaxation: 111.33
Bestes ganzzahliges Ergebnis:         110
Preis der Ganzzahligkeit:             1.33 (1.2 %)

In der LP-Loesung nur teilweise eingepackt: ['Kocher']
Genau hier wuerde Branch-and-Bound verzweigen:
  Ast 1: Kocher bleibt ganz zuhause (x=0)
  Ast 2: Kocher kommt ganz mit    (x=1)
========================================================================
```

Drei Beobachtungen, die den Kern des Kapitels illustrieren:

1. **Die LP-Relaxation packt zwei Drittel Kocher ein** — physikalisch unsinnig, als **Schranke** aber wertvoll: „Mehr als 111,33 ist unmöglich."
2. **Runden hätte hier versagt.** Abrunden des Kochers auf 0 ergäbe $35 + 30 + 8 + 25 = 98$ bei nur 9 kg — zwölf Punkte schlechter als das Optimum, und drei Kilo Kapazität bleiben ungenutzt. Aufrunden auf 1 ergäbe 12 kg und wäre **unzulässig**. Das exakte Optimum tauscht stattdessen das *Buch* gegen den *Kocher* — eine Umschichtung, auf die kein Rundungsverfahren kommt, weil sie eine bereits „fertige" Variable wieder verändert.
3. **Die Lücke ist klein** (1,2 %). Ein Solver, der frühzeitig bei 110 abbricht, kann beweisen, höchstens 1,2 % vom Optimum entfernt zu sein — ohne alle Äste zu durchsuchen. Genau das ist der MIP-Gap aus Kapitel 22.

6.7 Praxisfall: Portfolio mit Ordergebühren und Kardinalitätsgrenze

Szenario. Ein Investor verteilt 100 000 € auf sechs Anlageklassen.

- Jede Transaktion kostet **50 € Fixgebühr**, unabhängig vom Betrag.
- Höchstens **3 Positionen** gleichzeitig (*Kardinalitätsbeschränkung*).
- Wird eine Position eröffnet, dann mit **mindestens 10 000 €** (Mindestordergröße).
- Höchstens **40 000 €** pro Position.

Modell. Für jede Anlage i: $x_i \geq 0$ (Euro-Betrag) und $y_i \in \{0, 1\}$ (Position aktiv).

$$\max \sum_{i=1}^{n} \mu_i x_i - F \sum_{i=1}^{n} y_i$$

$$\sum_i x_i = B \qquad \text{(Budget vollständig investiert)}$$

$$\sum_i y_i \le K \qquad \text{(höchstens } K \text{ Positionen)}$$

$$L\, y_i \le x_i \le U\, y_i \quad \forall i \quad \text{(semikontinuierlich, Muster 5)}$$

 Formel-Lesehilfe * μ_i — erwartete Jahresrendite der Anlage i (z. B. 0,11 = 11 %). * $\mu_i x_i$ — erwarteter Ertrag in Euro. * $F \sum y_i$ — Summe der Ordergebühren: 50 € je aktivierter Position. * $L y_i \le x_i \le U y_i$ — die Doppelungleichung erledigt **beides** auf einmal: Ist $y_i = 0$, folgt $0 \le x_i \le 0$, also $x_i = 0$. Ist $y_i = 1$, folgt $10\,000 \le x_i \le 40\,000$.

Ohne Formel gesagt: „Investiere das ganze Budget in höchstens drei Töpfe, jeweils zwischen 10 000 und 40 000 Euro, und ziehe für jeden benutzten Topf 50 Euro Gebühr ab.“

 Eine ökonomische Unsauberkeit, die Sie kennen sollten

Das Modell verrechnet einen **einmaligen** Gebührenbetrag (50 €) mit einem **jährlichen** Ertrag ($\mu_i x_i$). Streng genommen mischt das Einheiten — bei einer Haltedauer von einem Jahr geht es auf, bei zehn Jahren wären die Gebühren zehnfach zu schwer gewichtet. Für das Beispiel ist der Effekt klein (150 € gegen ~11 000 € Ertrag), aber in einem echten Modell würde man entweder die Gebühr annualisieren oder mit Barwerten rechnen.

 Zwei Bausteine aus `or_kern.py`

Das Programm wertet den Solver nicht über `ergebnis.success` aus, sondern über `status_von_scipy()` und das `Loesung`-Objekt aus dem gemeinsamen Unterbau (Abschnitt 22.6). Der Unterschied ist kein Schönheitsfehler:

- `success` ist **ein Bit**. Es unterscheidet nicht zwischen „es gibt keine Lösung“ (Modellfehler — das Modell muss geändert werden) und „die Zeit war um“ (Rechenproblem — mehr Zeit oder ein besserer Startwert hilft). Das sind zwei völlig verschiedene Nachrichten an völlig verschiedene Adressaten.
- Das `Loesung`-Objekt führt neben dem Zielwert die **Schranke** mit und rechnet daraus den Gap aus. Erst dadurch steht in der Ausgabe `Gap: 0.00%` — die Zusage, dass hier wirklich das Optimum gefunden *und bewiesen* wurde und nicht bloß irgendetwas.

Genau die Unterscheidung, um die es im nächsten Abschnitt geht — hier schon einmal angewandt.

```
#!/usr/bin/env python3

# MILP_Portfolio_Fixgebuehren.py
"""
Kapitel MILP: MILP-Portfolio-Selektion mit Fixkosten und Kardinalitaet.
```

```
CP-SAT-freie, gut lesbare Formulierung ueber scipy/HiGHS (kein manueller
CSR-Matrixaufbau, deutlich leichter nachvollziehbar), mit Vergleich gegen die
Loesung OHNE Restriktionen.

Die Auswertung laeuft ueber SolverStatus und das Loesung-Objekt aus
or_kern.py: Alle Statusfaelle werden behandelt, und der MIP-Gap steht im
Bericht - statt eines blossen 'success', das nicht verraet, ob der Solver
fertig geworden ist oder nur aufgegeben hat.

Benoetigt: numpy, pandas, scipy, pydantic (ueber or_kern)
"""

from __future__ import annotations

import time

import numpy as np
import pandas as pd
from scipy.optimize import linprog

from or_kern import Loesung, SolverStatus, status_von_scipy

ANLAGEN = ["US-Aktien", "EU-Aktien", "Emerging-Markets",
           "Staatsanleihen", "Unternehmensanleihen", "Rohstoffe"]
RENDITE = np.array([0.11, 0.08, 0.13, 0.03, 0.05, 0.07])   # erwartet, p.a.

BUDGET = 100_000.0
MIN_POSITION = 10_000.0        # L
MAX_POSITION = 40_000.0        # U  (dient zugleich als Big-M!)
GEBUEHR = 50.0                 # F, je aktivierter Position
MAX_POSITIONEN = 3             # K

N = len(ANLAGEN)


def baue_und_loese(zeitlimit: float | None = None) -> Loesung:
    """
    Variablenreihenfolge:  [x_0..x_{N-1}, y_0..y_{N-1}]
    Zielfunktion (Maximierung -> fuer linprog negiert):
        max  sum(rendite_i * x_i) - GEBUEHR * sum(y_i)
    """
    c = np.concatenate([-RENDITE, np.full(N, GEBUEHR)])     # negiert = Minimierung

    # --- Gleichungsnebenbedingung: Budget vollstaendig investiert -----------
    A_eq = np.zeros((1, 2 * N))
    A_eq[0, :N] = 1.0
    b_eq = np.array([BUDGET])
```

```
    zeilen, grenzen = [], []

    # --- Kardinalitaet: sum(y_i) <= K -----------------------------------
    zeile = np.zeros(2 * N)
    zeile[N:] = 1.0
    zeilen.append(zeile)
    grenzen.append(MAX_POSITIONEN)

    # --- Obergrenze (Big-M):  x_i - U*y_i <= 0 ----------------------------
    for i in range(N):
        zeile = np.zeros(2 * N)
        zeile[i] = 1.0
        zeile[N + i] = -MAX_POSITION
        zeilen.append(zeile)
        grenzen.append(0.0)

    # --- Untergrenze:  L*y_i - x_i <= 0   (entspricht x_i >= L*y_i) --------
    for i in range(N):
        zeile = np.zeros(2 * N)
        zeile[i] = -1.0
        zeile[N + i] = MIN_POSITION
        zeilen.append(zeile)
        grenzen.append(0.0)

    A_ub = np.array(zeilen)
    b_ub = np.array(grenzen)

    schranken = [(0.0, MAX_POSITION)] * N + [(0.0, 1.0)] * N
    ganzzahligkeit = np.concatenate([np.zeros(N), np.ones(N)])   # y binaer

    t0 = time.perf_counter()
    ergebnis = linprog(c=c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
                       bounds=schranken, integrality=ganzzahligkeit,
                       method="highs",
                       options={"time_limit": zeitlimit} if zeitlimit else None)
    laufzeit = time.perf_counter() - t0

    status = status_von_scipy(ergebnis)
    if not status.brauchbar:
        return Loesung(status=status, laufzeit=laufzeit)

    # Zurueck in die Maximierungswelt: Zielwert UND Schranke negieren.
    # Aus beiden zusammen rechnet das Loesung-Objekt den MIP-Gap aus.
    x, y = ergebnis.x[:N], np.round(ergebnis.x[N:])
    return Loesung(
        status=status,
        werte={**{name: float(w) for name, w in zip(ANLAGEN, x)},
               **{f"aktiv:{name}": float(w) for name, w in zip(ANLAGEN, y)}},
```

```
        zielwert=float(-ergebnis.fun),
        schranke=float(-ergebnis.mip_dual_bound),
        laufzeit=laufzeit)


def ohne_restriktionen() -> tuple[float, np.ndarray]:
    """Vergleichsfall: nur Budget, keine Gebuehren/Kardinalitaet/Mindestgroesse."""
    ergebnis = linprog(c=-RENDITE, A_eq=[np.ones(N)], b_eq=[BUDGET],
                       bounds=[(0, None)] * N, method="highs")
    status = status_von_scipy(ergebnis)
    if not status.brauchbar:
        raise SystemExit(f"Vergleichsfall nicht loesbar: {status.value}")
    return -ergebnis.fun, ergebnis.x


def pruefe_portfolio(loesung: Loesung, toleranz: float = 1e-6) -> list[str]:
    """Prueft die Loesung gegen die Anforderungen - ohne den Solver zu fragen.

    Bewusst kein assert: Eine Beanstandungsliste laesst sich protokollieren,
    weiterreichen und testen. Ein assert verschwindet ausserdem, sobald
    jemand Python mit -O startet.
    """
    x = np.array([loesung.werte[name] for name in ANLAGEN])
    y = np.array([loesung.werte[f"aktiv:{name}"] for name in ANLAGEN])
    beanstandungen: list[str] = []

    if abs(x.sum() - BUDGET) > 1e-4:
        beanstandungen.append(f"Budget nicht exakt investiert: {x.sum():,.2f}")
    if y.sum() > MAX_POSITIONEN + toleranz:
        beanstandungen.append(f"{y.sum():.0f} Positionen statt hoechstens "
                              f"{MAX_POSITIONEN}")
    for i, name in enumerate(ANLAGEN):
        if abs(y[i] - round(y[i])) > toleranz:
            beanstandungen.append(f"{name}: y = {y[i]!r} ist nicht ganzzahlig")
        elif y[i] and not (MIN_POSITION - toleranz <= x[i]
                           <= MAX_POSITION + toleranz):
            beanstandungen.append(f"{name}: {x[i]:,.2f} EUR verletzt die "
                                  f"Groessengrenzen")
        elif not y[i] and x[i] > toleranz:
            beanstandungen.append(f"{name}: inaktiv, aber {x[i]:,.2f} EUR "
                                  f"investiert")
    return beanstandungen


if __name__ == "__main__":
    loesung = baue_und_loese()

    print("=" * 88)
```

```
print("      OPTIMALE MILP-PORTFOLIO-ALLOKATION MIT FIXGEBUEHREN")
print("=" * 88)
print(f"Budget: {BUDGET:,.0f} EUR | max. {MAX_POSITIONEN} Positionen | "
      f"je {MIN_POSITION:,.0f}-{MAX_POSITION:,.0f} EUR | Gebuehr {GEBUEHR:.0f} EUR\n")

# Zuerst der Status - erst danach interessieren die Zahlen.
if loesung.status.modellfehler:
    raise SystemExit(f"Das Modell ist nicht loesbar ({loesung.status.value}). "
                     f"Naechster Schritt: Anhang Fehlerdiagnose.")
if not loesung.status.brauchbar:
    raise SystemExit(f"Keine Loesung erhalten ({loesung.status.value}). "
                     f"Zeitlimit erhoehen oder Modell vereinfachen.")
if loesung.status is SolverStatus.ZULAESSIG:
    print(f"ACHTUNG: nicht beweisbar optimal - Gap {loesung.gap:.2%}\n")

x = np.array([loesung.werte[name] for name in ANLAGEN])
y = np.array([loesung.werte[f"aktiv:{name}"] for name in ANLAGEN], dtype=int)

tabelle = pd.DataFrame({
    "Anlage": ANLAGEN,
    "Aktiv": ["JA" if y[i] else "-" for i in range(N)],
    "Investition (EUR)": [f"{x[i]:,.0f}" for i in range(N)],
    "Anteil": [f"{x[i]/BUDGET*100:5.1f} %" for i in range(N)],
    "Erw. Rendite": [f"{RENDITE[i]*100:4.1f} %" for i in range(N)],
    "Erw. Ertrag (EUR)": [f"{x[i]*RENDITE[i]:,.0f}" for i in range(N)],
})
print(tabelle.to_string(index=False))

brutto = float(RENDITE @ x)
gebuehren = float(GEBUEHR * y.sum())
print("-" * 88)
print(f"Erwarteter Bruttoertrag: {brutto:>12,.2f} EUR")
print(f"Ordergebuehren:          {-gebuehren:>12,.2f} EUR ({y.sum()} Positionen)")
print(f"Netto-Erwartungswert:    {loesung.zielwert:>12,.2f} EUR")
print(f"\n{loesung.als_bericht()}")

# --- Vergleich mit dem unbeschraenkten Fall ---------------------------
z_frei, x_frei = ohne_restriktionen()
print("-" * 88)
print(f"Zum Vergleich ohne jede Restriktion (alles in den Bestwert): "
      f"{z_frei:,.2f} EUR")
print(f"Kosten der Realitaet (Gebuehren, Streuung, Mindestgroessen): "
      f"{z_frei - loesung.zielwert:,.2f} EUR "
      f"({(1 - loesung.zielwert/z_frei)*100:.2f} %)")

# --- Alle Nebenbedingungen nachpruefen --------------------------------
beanstandungen = pruefe_portfolio(loesung)
print("-" * 88)
```

```
    if beanstandungen:
        raise SystemExit("Abnahmepruefung fehlgeschlagen:\n  - "
                         + "\n  - ".join(beanstandungen))
    print("Abnahmepruefung: alle Nebenbedingungen geprueft und eingehalten.")
    print("=" * 88)
```

Erwartete Ausgabe:

```
========================================================================================
      OPTIMALE MILP-PORTFOLIO-ALLOKATION MIT FIXGEBUEHREN
========================================================================================
Budget: 100,000 EUR | max. 3 Positionen | je 10,000-40,000 EUR | Gebuehr 50 EUR

              Anlage Aktiv Investition (EUR)  Anteil Erw. Rendite Erw. Ertrag (EUR)
           US-Aktien    JA            40,000  40.0 %       11.0 %             4,400
           EU-Aktien    JA            20,000  20.0 %        8.0 %             1,600
    Emerging-Markets    JA            40,000  40.0 %       13.0 %             5,200
      Staatsanleihen     -                 0   0.0 %        3.0 %                 0
Unternehmensanleihen     -                 0   0.0 %        5.0 %                 0
           Rohstoffe     -                 0   0.0 %        7.0 %                 0
----------------------------------------------------------------------------------------
Erwarteter Bruttoertrag:    11,200.00 EUR
Ordergebuehren:               -150.00 EUR (3 Positionen)
Netto-Erwartungswert:       11,050.00 EUR

Status: optimal | Zielwert: 11,050.00 | Gap: 0.00% | Zeit: 0.01s
----------------------------------------------------------------------------------------
Zum Vergleich ohne jede Restriktion (alles in den Bestwert): 13,000.00 EUR
Kosten der Realitaet (Gebuehren, Streuung, Mindestgroessen): 1,950.00 EUR (15.00 %)
----------------------------------------------------------------------------------------
Abnahmepruefung: alle Nebenbedingungen geprueft und eingehalten.
========================================================================================
```

□ Code-Durchgang

Stelle	Was passiert
`c = concat([-RENDITE, full(N, GEBUEHR)])`	Zielfunktion über **beide** Variablenblöcke: Erträge negativ (weil `linprog` minimiert), Gebühren positiv (sie sollen ja gedrückt werden)
`A_eq[0, :N] = 1.0`	Budgetgleichung betrifft nur die x-Variablen
`zeile[i]=1; zeile[N+i]=-MAX_POSITION`	Muster 1: $x_i - Uy_i \le 0$. $M = U = 40\,000$ — die kleinstmögliche gültige Wahl
`zeile[i]=-1; zeile[N+i]=MIN_POSITION`	Muster 5 unten: $Ly_i - x_i \le 0$
`integrality=concat([zeros(N), ones(N)])`	nur die y sind ganzzahlig — das ist das „mixed“ in MILP

Stelle	Was passiert
`status = status_von_scipy(ergebnis)`	übersetzt den SciPy-Rückgabewert in die gemeinsame Sprache aus Abschnitt 22.6 — die drei `if`-Zweige darunter behandeln Modellfehler, Abbruch ohne Lösung und „zulässig, aber unbewiesen“ getrennt
`schranke=-ergebnis.mip_dual_bound`	die zweite Zahl aus Abschnitt 6.8. Erst mit ihr kann das `Loesung`-Objekt den Gap ausrechnen — und der Bericht sagt `Gap: 0.00%`, also *beweisbar* optimal
`pruefe_portfolio(...)`	prüft **jede** modellierte Regel einzeln nach und liefert eine Liste von Beanstandungen

Warum das Ergebnis wirtschaftlich Sinn ergibt: Der Solver wählt die drei renditestärksten Anlagen (13 %, 11 %, 8 %) und füllt sie in dieser Reihenfolge bis zur Obergrenze von 40 000 €. Die drittbeste Position (EU-Aktien) erhält nur den Rest von 20 000 € — sie liegt über der Mindestordergröße von 10 000 €, ist also zulässig. Ohne Kardinalitätsgrenze läge alles im Bestwert (13 %); die Restriktionen kosten 15 % des theoretischen Ertrags. Genau diese Zahl braucht man, wenn man mit dem Risikomanagement über die Sinnhaftigkeit einer Regel diskutiert: „Die Obergrenze von 40 % je Position kostet uns 1 950 € pro Jahr — ist uns die Diversifikation das wert?“ Das ist eine beantwortbare Frage; „wir sollten breiter streuen“ ist es nicht.

6.8 Wenn der Solver nicht fertig wird: Gap, Zeitlimit und Warm-Start

Bei einem LP gibt es zwei Ausgänge: eine optimale Lösung oder eine klare Absage. Bei einem MILP gibt es einen dritten, und im Betrieb ist er der häufigste:

> *„Ich habe eine Lösung. Ob sie die beste ist, weiß ich nicht. Die Zeit ist um.“*

Dieser Abschnitt handelt davon, wie man damit professionell umgeht — statt so zu tun, als käme immer `OPTIMAL` zurück.

Die zwei Zahlen, die zählen

Branch-and-Bound führt zu jedem Zeitpunkt zwei Werte mit:

Begriff	Was er bedeutet	Wie er sich entwickelt
Incumbent (beste gefundene Lösung)	ein Plan, den man tatsächlich ausführen könnte	wird im Lauf der Suche immer besser
Schranke (*dual bound*)	der beste Wert, den es überhaupt geben *könnte*	wird im Lauf der Suche immer schlechter

Beide laufen aufeinander zu. Ihr Abstand ist der **MIP-Gap**:

$$\text{Gap} = \frac{|z_{\text{incumbent}} - z_{\text{schranke}}|}{|z_{\text{incumbent}}|}$$

Formel-Übersetzer

Mathematik	Alltagssprache
$z_{\text{incumbent}}$	„Das ist der beste Plan, den ich bisher tatsächlich in der Hand habe.“
z_{schranke}	„Besser als das kann es rechnerisch nicht werden — bewiesen.“
Gap = 0	„Beide sind gleich: Der Plan ist beweisbar optimal.“
Gap = 0,09	„Mein Plan ist höchstens 9 % schlechter als das theoretische Beste. **Garantiert.**“

Der Gap ist damit keine Schätzung und keine Fehlerbalken-Angabe, sondern eine **Zusage**. Das macht ihn zu der Zahl, die man ins Managementgespräch mitnimmt: „Wir liegen höchstens 2 % vom Optimum entfernt“ ist eine belastbare Aussage. „Der Solver hat lange gerechnet“ ist keine.

Alle Statusfälle behandeln

Der zweite Teil der Professionalität ist unspektakulär, aber entscheidend: **jeden** Rückgabewert auswerten, nicht nur `OPTIMAL`.

Status	Bedeutung	Was Sie tun
`Optimal`	beweisbar bestmöglich	ausführen
`Time limit reached` (mit Lösung)	zulässig, nicht bewiesen optimal	Gap prüfen, dann entscheiden
`Time limit reached` (ohne Lösung)	nichts gefunden	Modell vereinfachen, Heuristik als Startwert, Limit erhöhen
`Infeasible`	kein zulässiger Plan existiert	harte Bedingungen prüfen (Kapitel 22, Anhang C)
`Unbounded`	Ziel wächst unbegrenzt	fehlende Schranke — fast immer ein Modellierungsfehler

```
#!/usr/bin/env python3

# Solverstatus_und_Gap.py
"""
Kapitel MILP: Was tun, wenn der Solver nicht fertig wird?
```

```
Bei einem LP kommt entweder eine optimale Loesung oder eine klare Absage.
Bei einem MILP ist der haeufigste Ausgang im Betrieb ein dritter: "Ich habe
eine Loesung, ich weiss aber nicht, ob sie die beste ist - und die Zeit ist
um." Dieses Programm zeigt, wie man mit diesem Fall umgeht.

  1. Alle Statusfaelle explizit behandeln, statt OPTIMAL vorauszusetzen.
  2. Den MIP-Gap lesen: Wie weit kann ich hoechstens danebenliegen?
  3. Messen, was zusaetzliche Rechenzeit ueberhaupt noch bringt.
  4. Warm-Start ausprobieren - und ehrlich messen, ob er etwas bringt.

Beispiel: Standortplanung, 45 moegliche Lager, 120 Kunden (5445 Variablen,
davon 45 binaer).

Benoetigt: numpy, highspy
"""

from __future__ import annotations

import time
from dataclasses import dataclass

import numpy as np
import highspy

RNG = np.random.default_rng(7)

N_LAGER, N_KUNDE = 45, 120
FIXKOSTEN = RNG.uniform(3000, 9000, N_LAGER)
TRANSPORT = RNG.uniform(5, 60, (N_LAGER, N_KUNDE))
BEDARF = RNG.uniform(10, 60, N_KUNDE)
KAPAZITAET = np.full(N_LAGER, BEDARF.sum() * 0.22)


@dataclass
class Ergebnis:
    """Alles, was nach einem Solverlauf ausgewertet werden muss - nicht nur
    der Zielwert."""
    status: str
    brauchbar: bool          # Gibt es ueberhaupt eine zulaessige Loesung?
    beweisbar_optimal: bool
    zielwert: float          # bester gefundener Wert (Incumbent)
    schranke: float          # beste bewiesene Schranke (Dual Bound)
    gap: float               # relativer Abstand zwischen beiden
    knoten: int
    dauer: float
    loesung: np.ndarray
```

```
def loese(zeitlimit: float, startloesung: np.ndarray | None = None) -> Ergebnis:
    """Loest das Standortmodell mit Zeitlimit und wertet ALLE Statusfaelle aus."""
    hochschule = highspy.Highs()
    hochschule.setOptionValue("output_flag", False)
    hochschule.setOptionValue("time_limit", zeitlimit)

    anzahl_x = N_LAGER * N_KUNDE
    unendlich = highspy.kHighsInf

    hochschule.addVars(anzahl_x, np.zeros(anzahl_x), np.full(anzahl_x, unendlich))
    hochschule.addVars(N_LAGER, np.zeros(N_LAGER), np.ones(N_LAGER))
    for i in range(N_LAGER):
        hochschule.changeColIntegrality(anzahl_x + i, highspy.HighsVarType.kInteger)
        hochschule.changeColCost(anzahl_x + i, FIXKOSTEN[i])
        for j in range(N_KUNDE):
            hochschule.changeColCost(i * N_KUNDE + j, TRANSPORT[i, j])

    for j in range(N_KUNDE):
        index = np.array([i * N_KUNDE + j for i in range(N_LAGER)], dtype=np.int32)
        hochschule.addRow(BEDARF[j], BEDARF[j], len(index), index, np.ones(len(index)))

    for i in range(N_LAGER):
        index = np.array([i * N_KUNDE + j for j in range(N_KUNDE)] + [anzahl_x + i],
                         dtype=np.int32)
        werte = np.concatenate([np.ones(N_KUNDE), [-KAPAZITAET[i]]])
        hochschule.addRow(-unendlich, 0.0, len(index), index, werte)

    if startloesung is not None:
        hochschule.setSolution(len(startloesung),
                               np.arange(len(startloesung), dtype=np.int32),
                               startloesung)

    t0 = time.perf_counter()
    hochschule.run()
    dauer = time.perf_counter() - t0

    status = hochschule.modelStatusToString(hochschule.getModelStatus())
    info = hochschule.getInfo()

    # Der Kern der Sache: Aus dem Status folgt, WAS man mit dem Ergebnis
    # ueberhaupt anfangen darf.
    brauchbar = status in ("Optimal", "Time limit reached", "Solution limit reached")
    beweisbar_optimal = status == "Optimal"
    if status in ("Infeasible", "Unbounded", "Primal infeasible or unbounded"):
        brauchbar = False

    return Ergebnis(
        status=status,
```

```
        brauchbar=brauchbar and info.objective_function_value < unendlich,
        beweisbar_optimal=beweisbar_optimal,
        zielwert=info.objective_function_value,
        schranke=info.mip_dual_bound,
        gap=info.mip_gap,
        knoten=info.mip_node_count,
        dauer=dauer,
        loesung=np.array(hochschule.getSolution().col_value),
    )


def gieriger_startplan() -> tuple[np.ndarray, float]:
    """Eine Faustregel-Loesung, wie sie ein Disponent von Hand erstellen wuerde:
    die guenstigsten Lager oeffnen (Fixkosten je Kapazitaetseinheit), dann
    jeden Kunden dem naechstgelegenen offenen Lager mit Restkapazitaet
    zuordnen. Kein Solver noetig - und in Sekunden fertig.
    """
    reihenfolge = np.argsort(FIXKOSTEN / KAPAZITAET)
    offen: list[int] = []
    for i in reihenfolge:
        offen.append(int(i))
        if KAPAZITAET[offen].sum() >= BEDARF.sum() * 1.05:
            break

    rest = KAPAZITAET.copy()
    x = np.zeros((N_LAGER, N_KUNDE))
    for j in np.argsort(-BEDARF):                # groesste Kunden zuerst
        for i in sorted(offen, key=lambda i: TRANSPORT[i, j]):
            menge = min(rest[i], BEDARF[j] - x[:, j].sum())
            if menge > 1e-9:
                x[i, j] += menge
                rest[i] -= menge
            if abs(x[:, j].sum() - BEDARF[j]) < 1e-9:
                break

    y = np.zeros(N_LAGER)
    y[offen] = 1.0
    kosten = float((FIXKOSTEN * y).sum() + (TRANSPORT * x).sum())
    return np.concatenate([x.ravel(), y]), kosten


def zeige(titel: str, e: Ergebnis) -> None:
    print(f"\n{titel}")
    print(f"  Status               {e.status}")
    if not e.brauchbar:
        print("  -> KEINE verwertbare Loesung. Nicht weiterrechnen!")
        return
    print(f"  bester Plan (Incumbent)   {e.zielwert:>12,.2f} EUR")
```

```
    print(f"  bewiesene Schranke        {e.schranke:>12,.2f} EUR")
    print(f"  MIP-Gap                   {e.gap * 100:>12.3f} %")
    print(f"  Knoten / Zeit             {e.knoten:>12,} / {e.dauer:.2f} s")
    if e.beweisbar_optimal:
        print("  -> beweisbar optimal")
    else:
        print(f"  -> zulaessig, aber nicht bewiesen optimal. Der wahre Bestwert")
        print(f"     liegt zwischen {e.schranke:,.2f} und {e.zielwert:,.2f} EUR.")


if __name__ == "__main__":
    print("=" * 78)
    print("  MIP-GAP UND ZEITLIMIT: STANDORTPLANUNG, 45 LAGER, 120 KUNDEN")
    print("=" * 78)
    print(f"{N_LAGER * N_KUNDE + N_LAGER:,} Variablen, davon {N_LAGER} binaer.")

    kurz = loese(2.0)
    zeige("[1] Zeitlimit 2 Sekunden", kurz)

    lang = loese(60.0)
    zeige("[2] Zeitlimit 60 Sekunden", lang)

    print("\n" + "=" * 78)
    print("  WAS BRINGT MEHR RECHENZEIT?")
    print("=" * 78)
    verbesserung = kurz.zielwert - lang.zielwert
    print(f"Nach 2 Sekunden:  {kurz.zielwert:,.2f} EUR, Gap {kurz.gap * 100:.2f} %")
    print(f"Nach {lang.dauer:.1f} Sekunden: {lang.zielwert:,.2f} EUR, bewiesen optimal")
    print(f"Gewinn durch {lang.dauer - kurz.dauer:.1f} Sekunden mehr Rechenzeit: "
          f"{verbesserung:,.2f} EUR "
          f"({verbesserung / kurz.zielwert * 100:.2f} %)")
    print()
    print("Das ist die Frage, die im Betrieb wirklich zaehlt: Der Gap von")
    print(f"{kurz.gap * 100:.1f} % nach 2 Sekunden ist eine GARANTIE - schlechter als")
    print("dieser Wert kann die Loesung nicht sein. Ob sich die restliche")
    print("Rechenzeit lohnt, entscheidet nicht der Solver, sondern die Anwendung:")
    print("Ein naechtlicher Tourenplan darf eine Stunde rechnen, eine Umplanung")
    print("bei Maschinenausfall hat 30 Sekunden.")

    print("\n" + "=" * 78)
    print("  BRINGT EIN WARM-START ETWAS?")
    print("=" * 78)
    start, start_kosten = gieriger_startplan()
    print(f"Faustregel-Startplan (ohne Solver): {start_kosten:,.2f} EUR")
    print(f"Das sind {(start_kosten / lang.zielwert - 1) * 100:.1f} % ueber dem Optimum.\n")

    warm = loese(60.0, startloesung=start)
    print(f"{'':26} {'Zeit':>9} {'Knoten':>9} {'Ziel':>13}")
```

```
    print("-" * 78)
    print(f"{'ohne Warm-Start':<26} {lang.dauer:>8.2f}s {lang.knoten:>9,} "
          f"{lang.zielwert:>13,.2f}")
    print(f"{'mit Warm-Start':<26} {warm.dauer:>8.2f}s {warm.knoten:>9,} "
          f"{warm.zielwert:>13,.2f}")
    print("-" * 78)
    print("Ergebnis: praktisch kein Unterschied. Der Grund ist nicht, dass")
    print("Warm-Starts nichts taugen - sondern dass HiGHS' eigene Heuristiken")
    print("innerhalb der ersten Sekunde bereits eine BESSERE Loesung finden als")
    print("unsere Faustregel. Ein Startwert hilft nur, wenn er besser ist als")
    print("das, was der Solver von allein in derselben Zeit findet.")
    print()
    print("Warm-Starts lohnen sich damit vor allem in zwei Faellen:")
    print("  * Sie haben Domaenenwissen, das der Solver nicht hat (siehe")
    print("    Warmstart_Effekt.py - dort halbiert ein Heuristik-Hinweis die Zeit).")
    print("  * Sie planen laufend neu und der gestrige Plan ist fast noch gueltig.")
    print("In beiden Faellen gilt: MESSEN, nicht glauben.")
    print("=" * 78)
```

Erwartete Ausgabe (Zeiten hardwareabhängig):

```
==============================================================================
  MIP-GAP UND ZEITLIMIT: STANDORTPLANUNG, 45 LAGER, 120 KUNDEN
==============================================================================
5,445 Variablen, davon 45 binaer.

[1] Zeitlimit 2 Sekunden
  Status                Time limit reached
  bester Plan (Incumbent)      69,255.12 EUR
  bewiesene Schranke           62,982.37 EUR
  MIP-Gap                          9.057 %
  Knoten / Zeit                        2 / 2.01 s
  -> zulaessig, aber nicht bewiesen optimal. Der wahre Bestwert
     liegt zwischen 62,982.37 und 69,255.12 EUR.

[2] Zeitlimit 60 Sekunden
  Status                Optimal
  bester Plan (Incumbent)      69,134.14 EUR
  bewiesene Schranke           69,128.19 EUR
  MIP-Gap                          0.009 %
  Knoten / Zeit                  150 / 8.98 s
  -> beweisbar optimal

==============================================================================
  WAS BRINGT MEHR RECHENZEIT?
==============================================================================
Nach 2 Sekunden:  69,255.12 EUR, Gap 9.06 %
```

```
Nach 9.0 Sekunden: 69,134.14 EUR, bewiesen optimal
Gewinn durch 7.0 Sekunden mehr Rechenzeit: 120.98 EUR (0.17 %)
```

Diese letzte Zeile ist die betriebswirtschaftlich interessante. Nach 2 Sekunden liegt ein Plan vor, der garantiert höchstens 9 % vom Optimum entfernt ist. Die restlichen 7 Sekunden Rechenzeit verbessern ihn um **0,17 %** — sie werden fast vollständig dafür verbraucht, die *Optimalität zu beweisen*, nicht den Plan zu verbessern.

> **Merksatz** Bei einem MILP kostet der Beweis der Optimalität meist ein Vielfaches dessen, was das Finden der optimalen Lösung kostet. Fragen Sie deshalb nicht „wie lange bis optimal?“, sondern „welchen Gap kann ich mir leisten?“. Ein nächtlicher Tourenplan darf eine Stunde rechnen; die Umplanung bei einem Maschinenausfall hat dreißig Sekunden.

Warm-Starts: was sie können und was nicht

Ein **Warm-Start** gibt dem Solver eine bekannte Lösung als Startpunkt mit. Die Idee ist verlockend: Wer schon eine brauchbare Lösung hat, muss nicht bei null anfangen.

In der Literatur wird das oft als sicherer Beschleuniger dargestellt. Das ist es **nicht** — und der ehrlichste Weg, das zu zeigen, ist eine Messung, die beide Seiten enthält. Der letzte Teil von `Solverstatus_und_Gap.py` füttert HiGHS mit einem Faustregel-Startplan:

```
Faustregel-Startplan (ohne Solver): 75,091.59 EUR
Das sind 8.6 % ueber dem Optimum.

                              Zeit    Knoten          Ziel
------------------------------------------------------------------------------
ohne Warm-Start               8.98s       150     69,134.14
mit Warm-Start                8.44s       150     69,134.14
```

Kein nennenswerter Unterschied. Der Grund ist nicht, dass Warm-Starts nichts taugen, sondern dass HiGHS’ eigene Heuristiken innerhalb der ersten Sekunde bereits eine *bessere* Lösung finden als unsere Faustregel. Ein Startwert hilft nur, wenn er besser ist als das, was der Solver in derselben Zeit von allein findet.

Der Gegenfall — eine Heuristik mit echtem Domänenwissen:

```
#!/usr/bin/env python3

# Warmstart_Effekt.py
"""
Kapitel MILP: Wann ein Warm-Start wirklich etwas bringt.

Solverstatus_und_Gap.py zeigt einen Fall, in dem ein Startwert NICHTS
bringt - HiGHS findet von allein schneller etwas Besseres. Hier der
Gegenfall: eine Heuristik, die dem Solver echtes Domaenenwissen liefert.

Problem: Lastverteilung. n Auftraege mit bekannter Dauer sind auf m
gleichartige Maschinen zu verteilen, sodass die zuletzt fertige Maschine
so frueh wie moeglich fertig wird (Makespan-Minimierung).
```

```
Die Heuristik: LPT (Longest Processing Time first) - laengste Auftraege
zuerst, jeder auf die momentan am wenigsten belastete Maschine. Sie ist
Jahrzehnte alt, in zwei Zeilen geschrieben und beweisbar nie schlechter
als 4/3 des Optimums.

Ueber model.AddHint() bekommt CP-SAT diese Loesung als Startpunkt.

WICHTIG: highspy wird hier bewusst NICHT importiert - es vertraegt sich
nicht mit ortools im selben Prozess (siehe Kapitel Oekosystem).

Benoetigt: numpy, ortools
"""

from __future__ import annotations

import time

import numpy as np
from ortools.sat.python import cp_model

RNG = np.random.default_rng(4)


def lpt_heuristik(dauer: np.ndarray, n_maschinen: int) -> tuple[np.ndarray, int]:
    """Longest Processing Time first.

    Laengste Auftraege zuerst auf die jeweils freieste Maschine legen. Zwei
    Zeilen, keine Bibliothek, Ergebnis in Mikrosekunden - und erstaunlich
    nah am Optimum.
    """
    zuordnung = np.zeros(len(dauer), dtype=int)
    belegung = np.zeros(n_maschinen)
    for auftrag in np.argsort(-dauer):            # laengster zuerst
        maschine = int(np.argmin(belegung))       # freieste Maschine
        zuordnung[auftrag] = maschine
        belegung[maschine] += dauer[auftrag]
    return zuordnung, int(belegung.max())


def loese(dauer: np.ndarray, n_maschinen: int, zeitlimit: float,
          hinweis: np.ndarray | None = None) -> tuple[str, int, float]:
    """Exaktes Modell mit CP-SAT, optional mit Startloesung als Hinweis."""
    n_auftraege = len(dauer)
    obergrenze = int(dauer.sum())

    modell = cp_model.CpModel()
    # x[i][k] = 1  <=>  Auftrag i laeuft auf Maschine k
    x = [[modell.NewBoolVar(f"x_{i}_{k}") for k in range(n_maschinen)]
```

```
         for i in range(n_auftraege)]
    for i in range(n_auftraege):
        modell.AddExactlyOne(x[i])                # jeder Auftrag genau einmal

    belegung = [modell.NewIntVar(0, obergrenze, f"last_{k}")
                for k in range(n_maschinen)]
    for k in range(n_maschinen):
        modell.Add(belegung[k] == sum(int(dauer[i]) * x[i][k]
                                      for i in range(n_auftraege)))

    makespan = modell.NewIntVar(0, obergrenze, "makespan")
    modell.AddMaxEquality(makespan, belegung)     # das Maximum ueber alle Maschinen
    modell.Minimize(makespan)

    # Der Warm-Start: ein Hinweis pro Variable. CP-SAT muss ihn nicht
    # befolgen - er nutzt ihn als erste Loesung, wenn er zulaessig ist.
    if hinweis is not None:
        for i in range(n_auftraege):
            for k in range(n_maschinen):
                modell.AddHint(x[i][k], 1 if hinweis[i] == k else 0)

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = zeitlimit
    # Ein Arbeiter und fester Startwert, damit die Messung reproduzierbar ist -
    # der Seed allein genuegt dafuer NICHT (Kapitel Constraint Programming).
    # Im Produktivbetrieb laesst man beides auf den Standardwerten.
    loeser.parameters.num_workers = 1
    loeser.parameters.random_seed = 1

    t0 = time.perf_counter()
    status = loeser.Solve(modell)
    dauer_s = time.perf_counter() - t0

    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError(f"Kein Plan gefunden: {loeser.StatusName(status)}")
    return loeser.StatusName(status), int(loeser.ObjectiveValue()), dauer_s


if __name__ == "__main__":
    print("=" * 78)
    print("  WARM-START: WENN DIE HEURISTIK MEHR WEISS ALS DER SOLVER")
    print("=" * 78)
    print("Lastverteilung: Auftraege auf gleichartige Maschinen verteilen,")
    print("sodass die letzte Maschine so frueh wie moeglich fertig wird.\n")

    print(f"{'Instanz':<22} {'Variante':<22} {'Makespan':>9} {'Zeit':>9} "
          f"{'Faktor':>8}")
    print("-" * 78)
```

```
    for n_auftraege, n_maschinen in [(60, 7), (80, 9)]:
        dauer = RNG.integers(10, 90, n_auftraege)
        start, lpt_wert = lpt_heuristik(dauer, n_maschinen)
        untere_schranke = dauer.sum() / n_maschinen

        instanz = f"{n_auftraege} Auftr., {n_maschinen} Masch."
        print(f"{instanz:<22} {'LPT-Heuristik':<22} {lpt_wert:>9} "
              f"{'< 0.001s':>9} {'':>8}")

        _, ziel_kalt, zeit_kalt = loese(dauer, n_maschinen, 60.0)
        print(f"{'':<22} {'CP-SAT kalt':<22} {ziel_kalt:>9} "
              f"{zeit_kalt:>8.3f}s {'1,0x':>8}")

        _, ziel_warm, zeit_warm = loese(dauer, n_maschinen, 60.0, hinweis=start)
        print(f"{'':<22} {'CP-SAT + LPT-Hinweis':<22} {ziel_warm:>9} "
              f"{zeit_kalt / zeit_warm:>7.1f}x")

        assert ziel_kalt == ziel_warm, \
            "Der Hinweis darf das Optimum nicht veraendern - nur den Weg dorthin!"
        print(f"{'':<22} {'untere Schranke':<22} {untere_schranke:>9.1f}")
        print("-" * 78)

    print("\nDrei Beobachtungen:")
    print("1. Der Hinweis aendert das ERGEBNIS nicht - beide Laeufe finden")
    print("   dasselbe Optimum. Er aendert nur, wie lange der Beweis dauert.")
    print("   Genau deshalb ist ein Warm-Start ungefaehrlich: Ein schlechter")
    print("   Hinweis kostet Zeit, er verfaelscht aber nie die Loesung.")
    print("2. Die LPT-Heuristik liegt schon sehr nah am Optimum. Ihr Wert fuer")
    print("   den Solver liegt weniger in der Qualitaet als darin, dass sie")
    print("   SOFORT da ist - der Solver kann von Beginn an alles verwerfen,")
    print("   was schlechter ist.")
    print("3. Der Faktor schwankt von Instanz zu Instanz - oben 2,2x und 1,4x -")
    print("   und laesst sich NICHT aus der Problemgroesse ableiten. Er haengt")
    print("   davon ab, wie schnell der Solver von allein eine vergleichbar gute")
    print("   Loesung findet. Das ist die eigentliche Lehre: Ein Warm-Start ist")
    print("   eine Messung wert, keine Glaubensfrage.")
    print("=" * 78)
```

Erwartete Ausgabe (Zeiten hardwareabhängig):

```
==============================================================================
  WARM-START: WENN DIE HEURISTIK MEHR WEISS ALS DER SOLVER
==============================================================================
Lastverteilung: Auftraege auf gleichartige Maschinen verteilen,
sodass die letzte Maschine so frueh wie moeglich fertig wird.

Instanz                Variante               Makespan      Zeit   Faktor
```

```
-------------------------------------------------------------------------
60 Auftr., 7 Masch.    LPT-Heuristik               483  < 0.001s
                       CP-SAT kalt                 475    0.627s     1,0x
                       CP-SAT + LPT-Hinweis        475    0.292s     2.2x
                       untere Schranke           474.7
-------------------------------------------------------------------------
80 Auftr., 9 Masch.    LPT-Heuristik               445  < 0.001s
                       CP-SAT kalt                 442    3.275s     1,0x
                       CP-SAT + LPT-Hinweis        442    2.420s     1.4x
                       untere Schranke           441.8
-------------------------------------------------------------------------
```

Hier wirkt der Hinweis: Faktor 2,2 bzw. 1,4. Beachten Sie aber, dass der Faktor **schwankt** und sich nicht aus der Problemgröße ableiten lässt — bei der größeren Instanz ist er sogar kleiner. Es gibt keine Regel, es gibt nur die Messung.

Typische Fehler

- **`if status == OPTIMAL: ... else: return None`.** Wirft eine Lösung weg, die 9 % vom Optimum entfernt und damit vollkommen brauchbar ist. Werten Sie den Gap aus, nicht nur den Status.
- **Zeitlimit ohne Gap-Auswertung.** Ein Limit erzwingt ein Ende, sagt aber nichts über die Qualität. Ohne den Gap wissen Sie nicht, ob Sie 0,1 % oder 60 % danebenliegen.
- **Warm-Start als Selbstverständlichkeit.** Messen Sie ihn — auf Ihrer Instanzklasse, gegen den kalten Lauf. In der Hälfte der Fälle bringt er nichts.
- **Einen Hinweis für eine unzulässige Lösung geben.** CP-SAT und HiGHS verwerfen ihn dann stillschweigend. Prüfen Sie Ihre Startlösung vorher auf Zulässigkeit — sonst messen Sie einen Effekt, den es gar nicht gibt.

Merksatz Ein Warm-Start kann das Ergebnis nie verfälschen — im schlimmsten Fall kostet er Zeit. Genau deshalb darf man ihn ausprobieren. Aber man muss ihn auch **messen**, statt an ihn zu glauben.

6.9 Übungsaufgaben

Lösungen: Abschnitt A.6.

Aufgabe 6.1 — Runden widerlegen. Konstruieren Sie selbst ein Beispiel mit zwei Variablen, bei dem Abrunden der LP-Lösung mindestens 50 % des optimalen Zielwerts verliert. Begründen Sie.

Aufgabe 6.2 — Big-M wählen. Ein Modell enthält $x_j \le M y_j$ mit $x_j \le 250$ als bekannter Kapazität. Welchen Wert sollte M haben? Was passiert bei $M = 10^6$?

Aufgabe 6.3 — Regeln in Ungleichungen übersetzen. Formulieren Sie mit Binärvariablen: (a) „Wenn Produkt A produziert wird, darf Produkt B nicht produziert werden.“ (b) „Mindestens zwei der vier Standorte müssen eröffnet werden.“ (c) „Wenn Standort 1 **und** Standort 2 eröffnet werden, muss auch das Zentrallager gebaut werden.“ (d) „Die Produktionsmenge ist entweder 0 oder liegt zwischen 500 und 2000.“ (e) „Genau eine der drei Maschinen wird eingesetzt.“

Aufgabe 6.4 □□ — Branch-and-Bound von Hand. Lösen Sie mit Branch-and-Bound und zeichnen Sie den Suchbaum:

$$\max\ 8x_1 + 11x_2 + 6x_3 + 4x_4 \quad \text{u. d. N.} \quad 5x_1 + 7x_2 + 4x_3 + 3x_4 \le 14, \quad x_i \in \{0,1\}$$

Geben Sie für jeden Knoten die LP-Schranke an und markieren Sie, wo gekappt wird.

Aufgabe 6.5 □□□ — Kardinalität variieren. Erweitern Sie `MILP_Portfolio_Fixgebuehren.py` so, dass es $K = 1, 2, \ldots, 6$ durchläuft und Netto-Ertrag sowie Rechenzeit tabelliert. (a) Ab welchem K steigt der Ertrag nicht mehr? Warum? (b) Wie verhält sich die Rechenzeit? (c) Was wäre der „faire Preis“ für die Erlaubnis, eine vierte Position zu eröffnen?

Aufgabe 6.6 □□□ — Big-M-Effekt messen. Ersetzen Sie im Portfoliomodell `MAX_POSITION` in der Big-M-Zeile durch 10^4, 10^6 und 10^9 (die echte Obergrenze bleibt in den `bounds`). Messen Sie jeweils Laufzeit und Zahl der Branch-and-Bound-Knoten (`ergebnis.mip_node_count`). Stellen Sie die Ergebnisse dar und erklären Sie sie.

Aufgabe 6.7 □□□ — Standortplanung. Modellieren und lösen Sie: Fünf mögliche Lagerstandorte mit Fixkosten $(80, 60, 90, 70, 50)$ Tsd. € versorgen vier Regionen mit Bedarf $(30, 45, 25, 40)$ Einheiten. Transportkosten je Einheit stehen in einer 5×4-Matrix Ihrer Wahl. Jedes eröffnete Lager hat Kapazität 80. Minimieren Sie Fix- plus Transportkosten.

6.10 Finde den Denkfehler

Die Warnung „M so klein wie möglich“ wird meist mit der Laufzeit begründet: schwache Relaxation, mehr Knoten. Das stimmt — und ist die **harmlosere** Hälfte der Wahrheit.

□ Finde den Denkfehler: Elf Lager, die keine Fixkosten kosten

Ein Team plant Standorte: 12 mögliche Lager, 40 Kunden, Fixkosten je eröffnetem Lager. Für das Big-M in der Kopplung

$$\sum_j x_{ij} \le M \cdot y_i$$

setzt jemand „sicherheitshalber“ eine sehr große Zahl ein. Das Modell läuft durch und meldet Gesamtkosten von **12 441,96 €**. Mit einem knapp gewählten M hatte dasselbe Modell zuvor 26 525,28 € gemeldet. Das Team freut sich über die Einsparung.

Ihre Aufgabe: (a) Warum ist die zweite Zahl *kleiner*, obwohl sich am Problem nichts geändert hat? Was müsste für ein größeres M mathematisch gelten? (b) Sehen Sie sich unten die Werte der Binärvariablen an — was fällt auf, und was hat das mit der Ganzzahltoleranz des Solvers zu tun? (c) Wie viele Lager liefern in dieser „Lösung“ tatsächlich Ware, und wie viele Fixkosten werden dafür verbucht? (d) Formulieren Sie die Prüfung, die diesen Fehler in jedem MILP-Skript auffliegen ließe.

Auflösung: Abschnitt A.6.

Das folgende Programm führt beide Läufe nebeneinander aus und wendet auf beide dieselbe Prüfung an.

```
#!/usr/bin/env python3

# Big_M_Falle.py
"""
Kapitel MILP: Was ein zu grosses Big-M wirklich anrichtet.

Lehrbuecher warnen vor grossem M mit dem Hinweis "die Relaxation wird
schwach, der Solver braucht mehr Knoten". Das stimmt - ist aber die
harmlosere Haelfte der Wahrheit. Die gefaehrlichere: Bei sehr grossem M
kann der Solver eine Loesung als ganzzahlig ANNEHMEN, in der die
Binaervariablen bei 1e-8 stehen. Dann liefern zugeschaltete Anlagen Ware
aus, waehrend das Modell ihre Fixkosten mit 0 verbucht.

Beispiel: Standortplanung, 12 moegliche Lager, 40 Kunden.
    min  sum_i fix_i * y_i  +  sum_ij kosten_ij * x_ij
    u.d.N.  sum_i x_ij = bedarf_j                    (jeder Kunde beliefert)
            sum_j x_ij <= M_i * y_i                  (Lager offen, wenn es liefert)
            y_i binaer,  x_ij >= 0

Zwei Laeufe:
  1. M knapp gewaehlt (= tatsaechliche Lagerkapazitaet)  -> richtig
  2. M = 1e7-fach zu gross, Presolve abgeschaltet        -> stilles Desaster
Beide werden mit derselben Pruefung kontrolliert, die den Fall auffliegen laesst.

Benoetigt: numpy, highspy
"""

from __future__ import annotations

import time

import numpy as np
import highspy

RNG = np.random.default_rng(11)

N_LAGER, N_KUNDE = 12, 40
FIXKOSTEN = RNG.uniform(2000, 5000, N_LAGER)
TRANSPORT = RNG.uniform(5, 40, (N_LAGER, N_KUNDE))
BEDARF = RNG.uniform(10, 60, N_KUNDE)
KAPAZITAET = BEDARF.sum() * 0.45          # jedes Lager schafft 45 % des Gesamtbedarfs

# Toleranz, ab der ein Solver eine Variable als ganzzahlig durchgehen laesst.
# HiGHS und die meisten anderen verwenden 1e-6 als Standard.
GANZZAHL_TOLERANZ = 1e-6
```

```
def loese(big_m: float, presolve: str = "on") -> dict:
    """Baut und loest das Standortmodell. Liefert Loesung und Solverkennzahlen."""
    hochschule = highspy.Highs()
    hochschule.setOptionValue("output_flag", False)
    hochschule.setOptionValue("presolve", presolve)
    hochschule.setOptionValue("time_limit", 300.0)

    anzahl_x = N_LAGER * N_KUNDE
    unendlich = highspy.kHighsInf

    # Spalten: erst alle x_ij, dann die y_i
    hochschule.addVars(anzahl_x, np.zeros(anzahl_x), np.full(anzahl_x, unendlich))
    hochschule.addVars(N_LAGER, np.zeros(N_LAGER), np.ones(N_LAGER))
    for i in range(N_LAGER):
        hochschule.changeColIntegrality(anzahl_x + i,
                                        highspy.HighsVarType.kInteger)
        hochschule.changeColCost(anzahl_x + i, FIXKOSTEN[i])
        for j in range(N_KUNDE):
            hochschule.changeColCost(i * N_KUNDE + j, TRANSPORT[i, j])

    # Jeder Kunde wird genau beliefert
    for j in range(N_KUNDE):
        index = np.array([i * N_KUNDE + j for i in range(N_LAGER)], dtype=np.int32)
        hochschule.addRow(BEDARF[j], BEDARF[j], len(index), index,
                          np.ones(len(index)))

    # Die Kopplung: sum_j x_ij - M * y_i <= 0
    for i in range(N_LAGER):
        index = np.array([i * N_KUNDE + j for j in range(N_KUNDE)] + [anzahl_x + i],
                         dtype=np.int32)
        werte = np.concatenate([np.ones(N_KUNDE), [-big_m]])
        hochschule.addRow(-unendlich, 0.0, len(index), index, werte)

    t0 = time.perf_counter()
    hochschule.run()
    dauer = time.perf_counter() - t0

    loesung = np.array(hochschule.getSolution().col_value)
    info = hochschule.getInfo()
    return {
        "x": loesung[:anzahl_x].reshape(N_LAGER, N_KUNDE),
        "y": loesung[anzahl_x:],
        "zielwert": info.objective_function_value,
        "knoten": info.mip_node_count,
        "dauer": dauer,
    }
```

```
def pruefe(ergebnis: dict) -> tuple[bool, list[str]]:
    """Die Pruefung, die in jedes MILP-Auswertungsskript gehoert.

    Sie rechnet die Kosten AUS DER LOESUNG neu aus, statt dem Zielwert des
    Solvers zu glauben - und vergleicht beide. Genau diese Gegenrechnung
    entlarvt eine Loesung, in der Binaervariablen bei 1e-8 haengengeblieben
    sind.
    """
    beanstandungen = []
    x, y = ergebnis["x"], ergebnis["y"]

    # 1. Sind die Binaervariablen wirklich binaer?
    abstand = np.abs(y - np.round(y))
    if abstand.max() > GANZZAHL_TOLERANZ:
        beanstandungen.append(
            f"y ist nicht ganzzahlig: groesster Abstand {abstand.max():.2e}")

    # 2. Liefert ein Lager, dessen Schalter aus ist?
    liefert = x.sum(axis=1) > 1e-6
    geschlossen_aber_aktiv = np.where(liefert & (y < 0.5))[0]
    if geschlossen_aber_aktiv.size:
        beanstandungen.append(
            f"Lager {geschlossen_aber_aktiv.tolist()} liefern Ware, "
            f"gelten im Modell aber als geschlossen")

    # 3. Stimmt der Zielwert mit den echten Kosten ueberein?
    echte_fixkosten = FIXKOSTEN[liefert].sum()
    echte_transportkosten = float((TRANSPORT * x).sum())
    echte_kosten = echte_fixkosten + echte_transportkosten
    if abs(echte_kosten - ergebnis["zielwert"]) > 1e-4 * max(1.0, echte_kosten):
        beanstandungen.append(
            f"Zielwert {ergebnis['zielwert']:,.2f} weicht von den echten Kosten "
            f"{echte_kosten:,.2f} ab (Differenz {echte_kosten - ergebnis['zielwert']:,.2f})")

    return not beanstandungen, beanstandungen


def zeige(titel: str, ergebnis: dict) -> None:
    x, y = ergebnis["x"], ergebnis["y"]
    liefert = x.sum(axis=1) > 1e-6
    print(f"\n{titel}")
    print(f"  Zielwert laut Solver        {ergebnis['zielwert']:>14,.2f} EUR")
    print(f"  Knoten / Zeit               {ergebnis['knoten']:>14,}  "
          f"/ {ergebnis['dauer']:.3f} s")
    print(f"  Lager mit y = 1             {int((y > 0.5).sum()):>14}")
    print(f"  Lager, die tatsaechlich liefern {int(liefert.sum()):>10}")
    unter = y[y < 0.5]
    print(f"  groesster y-Wert unter 0.5  "
```

```
          f"{(unter.max() if unter.size else 0.0):>14.3e}")
    print(f"  Fixkosten real / verbucht   {FIXKOSTEN[liefert].sum():>14,.2f} "
          f"/ {float((FIXKOSTEN * y).sum()):,.2f} EUR")

    in_ordnung, beanstandungen = pruefe(ergebnis)
    if in_ordnung:
        print("  PRUEFUNG: bestanden")
    else:
        print("  PRUEFUNG: DURCHGEFALLEN")
        for text in beanstandungen:
            print(f"    - {text}")


if __name__ == "__main__":
    print("=" * 78)
    print("  DIE BIG-M-FALLE: STANDORTPLANUNG, 12 LAGER, 40 KUNDEN")
    print("=" * 78)
    print(f"Tatsaechliche Lagerkapazitaet: {KAPAZITAET:,.1f} Einheiten.")
    print("Genau das ist das kleinstmoegliche gueltige M - mehr kann ein Lager")
    print("ohnehin nicht ausliefern.")

    knapp = loese(KAPAZITAET)
    zeige("[1] M = Kapazitaet (richtig gewaehlt)", knapp)

    gross = loese(1e7 * KAPAZITAET, presolve="off")
    zeige("[2] M = 10 Millionen mal Kapazitaet, Presolve abgeschaltet", gross)

    print("\n" + "=" * 78)
    print("  WAS DA PASSIERT IST")
    print("=" * 78)
    fehlbetrag = knapp["zielwert"] - gross["zielwert"]
    print(f"Lauf [2] meldet {gross['zielwert']:,.2f} EUR und sieht damit um")
    print(f"{fehlbetrag:,.2f} EUR BESSER aus als die richtige Loesung - ein Ergebnis,")
    print("ueber das sich jeder Auftraggeber freuen wuerde.")
    print()
    print("Der Grund steht in der Zeile 'groesster y-Wert unter 0.5': Die")
    print(f"Schaltervariablen stehen bei rund "
          f"{gross['y'][gross['y'] < 0.5].max():.0e}.")
    print(f"Das ist kleiner als die Ganzzahltoleranz {GANZZAHL_TOLERANZ:.0e}, also gilt")
    print("y = 0 - 'Lager geschlossen'. Zugleich ist M so gross, dass")
    print("  sum_j x_ij <= M * 1e-8")
    print("immer noch reichlich Liefermenge erlaubt. Die Lager liefern also,")
    print("ohne dass ihre Fixkosten je bezahlt werden. Der Fachbegriff dafuer")
    print("ist 'trickle flow'.")
    print()
    print("WICHTIG: Mit eingeschaltetem Presolve (Standard) faellt HiGHS hier")
    print("nicht darauf herein - es zieht M selbst zurecht. Verlassen Sie sich")
    print("nicht darauf: Presolve kann das nur, wenn eine implizite Schranke")
```

```
    print("herleitbar ist. Die Pruefung aus pruefe() kostet Millisekunden und")
    print("funktioniert immer.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  DIE BIG-M-FALLE: STANDORTPLANUNG, 12 LAGER, 40 KUNDEN
==============================================================================
Tatsaechliche Lagerkapazitaet: 722.4 Einheiten.
Genau das ist das kleinstmoegliche gueltige M - mehr kann ein Lager
ohnehin nicht ausliefern.

[1] M = Kapazitaet (richtig gewaehlt)
  Zielwert laut Solver             26,525.28 EUR
  Knoten / Zeit                           13  / 0.075 s
  Lager mit y = 1                          3
  Lager, die tatsaechlich liefern          3
  groesster y-Wert unter 0.5       0.000e+00
  Fixkosten real / verbucht        10,671.79 / 10,671.79 EUR
  PRUEFUNG: bestanden

[2] M = 10 Millionen mal Kapazitaet, Presolve abgeschaltet
  Zielwert laut Solver             12,441.96 EUR
  Knoten / Zeit                            1  / 0.006 s
  Lager mit y = 1                          0
  Lager, die tatsaechlich liefern         11
  groesster y-Wert unter 0.5       4.074e-08
  Fixkosten real / verbucht        35,847.91 / 0.00 EUR
  PRUEFUNG: DURCHGEFALLEN
    - Lager [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11] liefern Ware, gelten im Modell aber als
     ↪  geschlossen
    - Zielwert 12,441.96 weicht von den echten Kosten 48,289.86 ab (Differenz 35,847.91)

==============================================================================
  WAS DA PASSIERT IST
==============================================================================
Lauf [2] meldet 12,441.96 EUR und sieht damit um
14,083.32 EUR BESSER aus als die richtige Loesung - ein Ergebnis,
ueber das sich jeder Auftraggeber freuen wuerde.

Der Grund steht in der Zeile 'groesster y-Wert unter 0.5': Die
Schaltervariablen stehen bei rund 4e-08.
Das ist kleiner als die Ganzzahltoleranz 1e-06, also gilt
y = 0 - 'Lager geschlossen'. Zugleich ist M so gross, dass
  sum_j x_ij <= M * 1e-8
immer noch reichlich Liefermenge erlaubt. Die Lager liefern also,
ohne dass ihre Fixkosten je bezahlt werden. Der Fachbegriff dafuer
```

```
ist 'trickle flow'.

WICHTIG: Mit eingeschaltetem Presolve (Standard) faellt HiGHS hier
nicht darauf herein - es zieht M selbst zurecht. Verlassen Sie sich
nicht darauf: Presolve kann das nur, wenn eine implizite Schranke
herleitbar ist. Die Pruefung aus pruefe() kostet Millisekunden und
funktioniert immer.
============================================================================
```

Code-Durchgang

Stelle	Was passiert	Warum es zählt
`presolve="off"`	schaltet HiGHS' Vorverarbeitung ab	Mit Standardeinstellung fällt HiGHS **nicht** herein: Presolve zieht das übergroße *M* selbst zurecht. Der Fehler ist trotzdem real — Presolve kann das nur, wenn eine implizite Schranke im Modell steckt. Verlassen Sie sich nicht auf eine Rettung, die Sie nicht kontrollieren.
`np.abs(y - np.round(y))`	prüft, ob Binärvariablen wirklich binär sind	Ein Wert von $4 \cdot 10^{-8}$ liegt unter der Toleranz 10^{-6} und gilt dem Solver als 0. Für Ihr Geschäft ist er es nicht.
`liefert & (y < 0.5)`	sucht Lager, die liefern, obwohl der Schalter aus ist	Die inhaltliche Prüfung: Widerspricht die Lösung sich selbst?
Kosten **aus der Lösung** neu berechnen	statt dem Zielwert zu glauben	Der wichtigste Test überhaupt. Er vergleicht nicht Modell mit Modell, sondern Modell mit Wirklichkeit — und deckt jeden Fehler dieser Art auf, egal wodurch er entstanden ist.

6.11 Micro-Quiz

Micro-Quiz 6: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Ihr Solver meldet nach dem Zeitlimit: Incumbent 48 200 €, Schranke 47 100 €. Was berichten Sie dem Auftraggeber? (a) „Der Solver ist nicht fertig geworden, das Ergebnis ist unbrauchbar." (b) „Wir haben einen Plan für 48 200 €. Er ist garantiert höchstens 2,3 % teurer als das theoretisch bestmögliche." (c) „Der optimale Wert liegt bei 47 100 €, wir müssen den Plan nur noch dorthin verbessern."

2. In Ihrem Fixkostenmodell ist $M = 10^9$, die tatsächliche Anlagenkapazität beträgt 500. Welche Gefahr ist die größere? (a) Der Solver braucht mehr Knoten und wird langsamer. (b) Binärvariablen können bei Werten um 10^{-8} hängenbleiben, gelten dem Solver als 0 — und Anlagen produzieren, ohne dass ihre Fixkosten je verbucht werden. (c) Die Zielfunktion wird numerisch instabil und liefert negative Kosten.

3. Sie geben CP-SAT über `AddHint()` eine Startlösung mit, die eine Nebenbedingung verletzt. Was passiert? (a) Der Solver meldet `INFEASIBLE`, weil der Hinweis unzulässig ist. (b) Der Solver übernimmt den Hinweis und liefert eine unzulässige Lösung. (c) Der Hinweis wird verworfen; das Ergebnis bleibt korrekt, der erhoffte Zeitgewinn bleibt aber aus.

6.12 Selbsttest

Antworten: Anhang A.

1. Warum liefert die LP-Relaxation bei Maximierung eine **obere** Schranke?
2. Nennen Sie die drei Gründe, aus denen ein Branch-and-Bound-Ast gekappt werden darf.
3. Wie modelliert man „x ist entweder 0 oder mindestens 500"?
4. Warum verschlechtert ein zu großes M die Laufzeit, obwohl das Modell korrekt bleibt?
5. Was bedeutet ein MIP-Gap von 3 %?

6.13 Zusammenfassung

- **Runden ist keine Lösung:** Aufrunden ist fast immer unzulässig (98–100 % der Fälle), Abrunden verliert im Mittel 10–26 %, im Extremfall alles.
- **Branch-and-Bound** löst LP-Relaxationen, verzweigt an gebrochenen Variablen und kappt Äste anhand von Schranken. Der Preis: NP-Schwere.
- **Fünf Modellierungsmuster** decken die meisten Praxisfälle ab: Fixkosten, Entweder-Oder, Wenn-Dann, Kardinalität, semikontinuierlich.
- M **so klein wie möglich** — nicht nur wegen der Laufzeit. Ein übergroßes M kann Binärvariablen bei 10^{-8} hängen lassen, wo sie dem Solver als 0 gelten: Anlagen produzieren, ohne dass ihre Fixkosten je verbucht werden (*Trickle Flow*).

- **`OPTIMAL` ist nicht der Normalfall.** Werten Sie jeden Status aus und lesen Sie den **MIP-Gap** — er ist eine Garantie („höchstens 2,3 % vom Optimum“), keine Schätzung. Der Beweis der Optimalität kostet meist ein Vielfaches dessen, was das Finden der optimalen Lösung kostet.
- **Warm-Starts können nichts verfälschen, aber auch nichts versprechen.** Ob ein Startwert hilft, hängt davon ab, ob er besser ist als das, was der Solver in derselben Zeit von allein findet — im Buch gemessen: einmal Faktor 2,2, einmal gar nichts.
- **Prüfen Sie jede modellierte Regel** nach dem Lösen mit einer `assert`-Zeile — und rechnen Sie den Zielwert aus der Lösung nach, statt ihm zu glauben.

Ausblick. Kapitel 7 wechselt das Paradigma: Bei Dienstplänen, Zuordnungen und Reihenfolgen ist die algebraische MILP-Sicht oft unhandlich. Constraint Programming denkt stattdessen in Wertebereichen und logischen Regeln — und löst solche Probleme um Größenordnungen schneller.

Kapitel 7: Constraint Programming mit CP-SAT — Logik, Scheduling und Zuweisung

Kapitel auf einen Blick

Worum geht es? Um ein völlig anderes Denkmodell: Statt algebraischer Ungleichungen arbeitet Constraint Programming mit **Wertebereichen** und **logischen Regeln**. Für Dienstpläne, Zuordnungen und Reihenfolgen ist das die mit Abstand produktivste Methode.

Voraussetzungen: Kapitel 6 (Binärvariablen, logische Bedingungen).

Danach können Sie: Zuweisungs-, Scheduling- und Reihenfolgeprobleme mit CP-SAT modellieren, globale Constraints einsetzen, weiche Ziele über Strafkosten steuern — und alle fünf Solver-Antworten auseinanderhalten, statt nur auf `OPTIMAL` zu prüfen.

Zeitbedarf: ca. 6,5 Stunden.

Programme:
`Propagation_Demo.py`
`CP_SAT_Vertretungssystem.py`
`JobShop_Intervalle.py`
`Parallele_Suche.py`
`CP_SAT_Statusfaelle.py`
`Strafgewichte.py`

Notebook: `Notebooks_04/cpsat.ipynb`

7.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Der Wochendienstplan

Vier Personen, fünf Tage Bereitschaftsdienst. Jeder Tag muss von **genau einer** Person besetzt sein, niemand soll mehr als **zwei** Dienste bekommen, und drei Abwesenheiten sind bekannt.

```
from ortools.sat.python import cp_model

personen = ["Abel", "Bux", "Cor", "Dane"]
dienste = ["Mo", "Di", "Mi", "Do", "Fr"]
nicht_verfuegbar = {("Bux", "Mi"), ("Cor", "Mo"), ("Cor", "Di")}

m = cp_model.CpModel()
x = {(p, d): m.NewBoolVar(f"{p}_{d}") for p in personen for d in dienste}
for d in dienste:
    m.AddExactlyOne(x[p, d] for p in personen)          # jeder Tag genau einmal
 ↪  besetzt
for p in personen:
    m.Add(sum(x[p, d] for d in dienste) <= 2)           # niemand mehr als zwei
 ↪  Dienste
for p, d in nicht_verfuegbar:
    m.Add(x[p, d] == 0)                                 # Abwesenheiten

s = cp_model.CpSolver()
print(s.StatusName(s.Solve(m)))
for d in dienste:
    print(d, "->", next(p for p in personen if s.Value(x[p, d])))
```

Ausgabe:

```
OPTIMAL
Mo -> Dane
Di -> Dane
Mi -> Abel
Do -> Bux
Fr -> Abel
```

Elf Zeilen Modell, und der Plan steht. Beachten Sie besonders die Zeile `m.AddExactlyOne(...)`. Sie sagt wörtlich, was gemeint ist: *genau eine* Person je Tag. In MILP hätte man dafür eine Gleichung $\sum_p x_{p,d} = 1$ aufgestellt — machbar, aber schon eine Übersetzungsleistung. Bei Regeln wie „diese beiden Dienste nicht direkt hintereinander“ oder „höchstens eine Nachtschicht pro Woche“ wächst dieser Übersetzungsaufwand rasch; in CP-SAT bleibt er eine Zeile.

Und jetzt sehen Sie sich das Ergebnis genau an. *Cor bekommt keinen einzigen Dienst.* Dane arbeitet Montag **und** Dienstag hintereinander. Ist das ein Fehler?

Nein — und das ist die wichtigste Lektion dieses Kapitels. Der Plan erfüllt **jede** Regel, die wir aufgeschrieben haben. Wir haben nur nie nach Fairness gefragt, und wir haben nie gesagt, dass Dienste nicht aufeinanderfolgen sollen. Ein Solver liefert exakt das Bestellte — nicht das Gemeinte.

 Merksatz Ein Modell, das durchläuft, ist noch kein richtiges Modell. Die eigentliche Arbeit im Constraint Programming ist nicht das Lösen, sondern das vollständige Aufschreiben dessen, was „ein guter Plan“ überhaupt heißt. Wie man Wünsche wie Fairness als **weiche** Bedingungen mit Strafkosten formuliert, zeigt Abschnitt 7.5 — und Abschnitt 7.10 zeigt, was passiert, wenn man ihr Gewicht falsch wählt.

7.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, wodurch sich Constraint Programming von MILP unterscheidet.
2. … die Wirkung von **Constraint Propagation** an einem Beispiel nachvollziehen.
3. … die wichtigsten globalen Constraints einsetzen: `AddExactlyOne`, `AddAllDifferent`, `AddNoOverlap`, `AddCumulative`.
4. … mit **Intervallvariablen** ein Maschinenbelegungsproblem modellieren.
5. … harte Regeln und weiche Ziele in einem Modell kombinieren und gewichten.
6. … die fünf CP-SAT-Statusfälle unterscheiden — insbesondere `INFEASIBLE` (es gibt keine Lösung) von `UNKNOWN` (es wurde keine gefunden) und `MODEL_INVALID` (Ihr Code ist fehlerhaft).
7. … begründen, warum `num_workers = 1` für einen reproduzierbaren Lauf nötig ist und ein fester `random_seed` allein nicht genügt — und was ein Test deshalb prüfen darf.
8. … begründen, warum ein Strafgewicht ein **Wechselkurs** ist, und welche Regeln deshalb niemals in die Zielfunktion gehören.

7.3 Ein anderes Denkmodell

Während LP und MILP auf algebraischen Gleichungen und Simplex-Matrizen beruhen, kommt **Constraint Programming (CP)** aus der künstlichen Intelligenz:

	MILP	Constraint Programming
Variablen	reell oder ganzzahlig, mit Schranken	diskrete **Wertebereiche** (*domains*)
Bedingungen	lineare Ungleichungen	beliebige logische Relationen
Suchprinzip	Relaxation + Verzweigen + Schranken	**Propagation** + Verzweigen + Konfliktlernen
Stärke	ökonomische Modelle, Kosten, Kapazitäten	Zuordnung, Reihenfolge, Zeitfenster, Regeln
Schwäche	viele logische Regeln → viele Big-M	reine Kostenoptimierung über Kontinuum

Der **CP-SAT-Solver** von Google OR-Tools verbindet Constraint Programming mit modernen **SAT-Techniken** (*Boolean Satisfiability*, Erfüllbarkeitsproblem der Aussagenlogik), insbesondere **CDCL** (*Conflict-Driven Clause Learning*, konfliktgetriebenes Klauselllernen). Er gehört zu den weltweit

leistungsfähigsten Engines für Schichtpläne, Vertretungsplanung, Projektterminierung und Job-Shop-Scheduling.

Was Propagation leistet

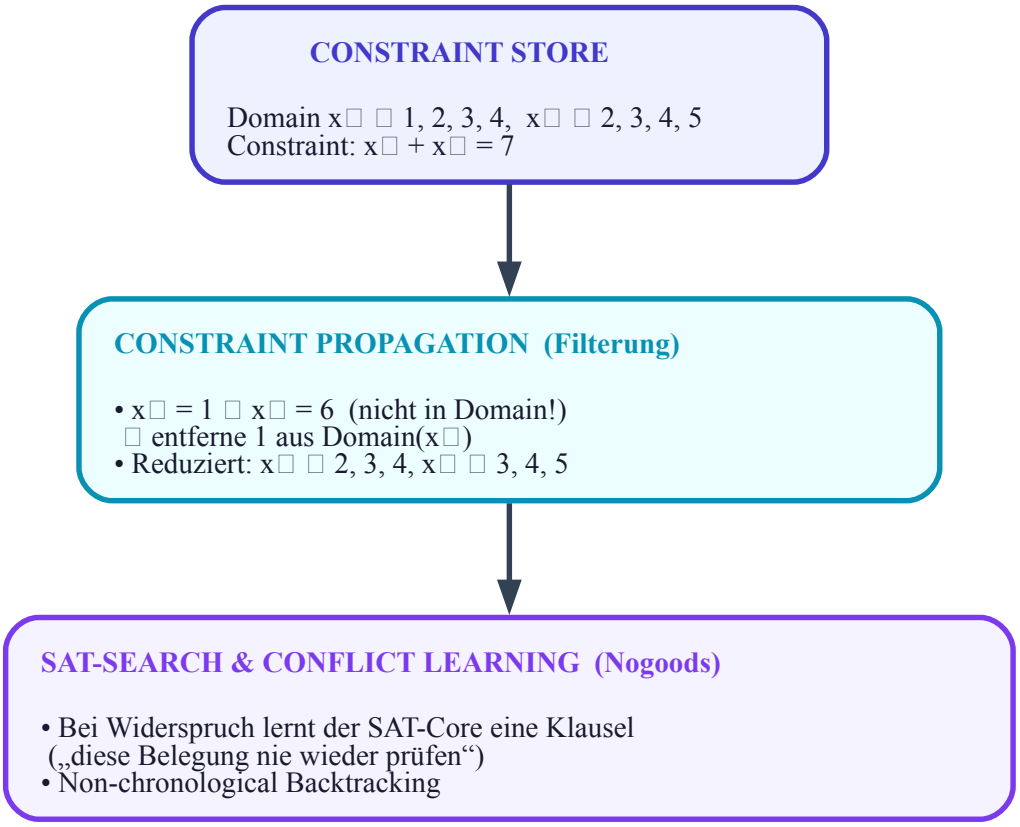

Abbildung 12: Abb. 7.1: CP-SAT – Propagation und Conflict Learning

□ Handrechnung 7.1: Propagation von Hand

Gegeben: $x_1 \in \{1,2,3,4\}$, $x_2 \in \{2,3,4,5\}$, Bedingung $x_1 + x_2 = 7$.

Runde 1 — von x_1 aus: * $x_1 = 1$ □ $x_2 = 6$. Aber $6 \notin \{2,3,4,5\} \rightarrow 1$ **aus der Domain von x_1 streichen.** * $x_1 = 2$ □ $x_2 = 5$ □ bleibt. * $x_1 = 3,4$ □ $x_2 = 4,3$ □ bleiben. Neue Domain: $x_1 \in \{2,3,4\}$.

Runde 2 — von x_2 aus: * $x_2 = 2$ □ $x_1 = 5 \notin \{2,3,4\} \rightarrow$ **streichen.** * $x_2 = 3,4,5$ □ $x_1 = 4,3,2$ □ Neue Domain: $x_2 \in \{3,4,5\}$.

Aus $4 \times 4 = 16$ Kombinationen sind ohne jedes Ausprobieren **9** geworden. Bei tausenden verketteten Bedingungen wirkt dieser Effekt lawinenartig — das ist der Kern der Leistungsfähigkeit von CP.

Und wenn ein Widerspruch auftritt? Dann lernt der SAT-Kern eine **Sperrklausel** (*nogood*): „Diese Kombination nie wieder probieren.“ Der Solver springt dann nicht nur einen Schritt zurück, sondern direkt zur Ursache des Konflikts (*non-chronological backtracking*).

```
#!/usr/bin/env python3
```

```python
# Propagation_Demo.py
"""
Kapitel CP-SAT: Propagation sichtbar machen.
Vergleicht die Zahl der geprueften Kombinationen bei roher Aufzaehlung
mit der Zahl der Verzweigungen, die CP-SAT tatsaechlich braucht.
"""

import itertools

from ortools.sat.python import cp_model


def rohe_aufzaehlung(n: int, obergrenze: int) -> tuple[int, int]:
    """Zaehlt ALLE Kombinationen und prueft jede einzeln (Brute Force)."""
    geprueft = 0
    loesungen = 0
    for kombination in itertools.product(range(1, obergrenze + 1), repeat=n):
        geprueft += 1
        if len(set(kombination)) == n and sum(kombination) == 3 * n:
            loesungen += 1
    return geprueft, loesungen


def mit_cp_sat(n: int, obergrenze: int) -> tuple[int, int, float]:
    """Dasselbe Problem deklarativ: alle verschieden, Summe = 3n."""
    modell = cp_model.CpModel()
    x = [modell.NewIntVar(1, obergrenze, f"x{i}") for i in range(n)]
    modell.AddAllDifferent(x)                # globales Constraint
    modell.Add(sum(x) == 3 * n)

    loeser = cp_model.CpSolver()
    loeser.parameters.enumerate_all_solutions = True

    class Zaehler(cp_model.CpSolverSolutionCallback):
        def __init__(self):
            super().__init__()
            self.anzahl = 0

        def on_solution_callback(self):
            self.anzahl += 1

    zaehler = Zaehler()
    loeser.Solve(modell, zaehler)
    return zaehler.anzahl, loeser.NumBranches(), loeser.WallTime()


if __name__ == "__main__":
    print("=" * 84)
```

```
    print("  PROPAGATION: WIE VIEL ARBEIT SPART SICH DER SOLVER?")
    print("  Aufgabe: n Zahlen aus 1..G, alle verschieden, Summe = 3n")
    print("=" * 84)
    print(f"{'n':>3} {'G':>4} | {'Brute Force':>14} | {'CP-SAT':>10} | "
          f"{'Ersparnis':>11} | {'Loesungen':>10}")
    print("-" * 84)

    for n, grenze in [(4, 8), (5, 10), (6, 12), (7, 14)]:
        kombis, treffer_bf = rohe_aufzaehlung(n, grenze)
        treffer_cp, verzweigungen, dauer = mit_cp_sat(n, grenze)
        assert treffer_bf == treffer_cp, "Beide Verfahren muessen gleich viele finden!"
        ersparnis = kombis / max(verzweigungen, 1)
        print(f"{n:>3} {grenze:>4} | {kombis:>14,} | {verzweigungen:>10,} | "
              f"{ersparnis:>10.0f}x | {treffer_cp:>10,}")

    print("-" * 84)
    print("'Brute Force' = alle Kombinationen. 'CP-SAT' = tatsaechliche Verzweigungen.")
    print("Der Rest wurde durch Propagation ausgeschlossen, ohne ihn anzusehen.")
    print("=" * 84)
```

 Merksatz MILP fragt: „Wie weit kann ich den Zielwert noch verbessern?“ (Schranken). CP fragt: „Welche Werte sind überhaupt noch möglich?“ (Propagation). Bei Problemen, deren Schwierigkeit in den **Regeln** liegt und nicht in den **Kosten**, ist die zweite Frage die produktivere.

7.4 Globale Constraints — die Bausteine von CP-SAT

CP-SAT stellt mächtige Makros bereit, die in MILP hunderte Big-M-Ungleichungen erfordern würden. Jedes davon hat einen spezialisierten, hocheffizienten Propagator.

Constraint	Bedeutung	Typische Anwendung
`AddAllDifferent(vars)`	Alle Variablen haben **paarweise verschiedene** Werte	Sudoku, Zuordnung, Stundenplan-Räume
`AddExactlyOne(bools)`	**Genau eine** der Booleschen ist wahr	„Jede Schicht wird von genau einer Person besetzt“
`AddAtMostOne(bools)`	**Höchstens eine** ist wahr	Exklusive Ressourcennutzung
`AddBoolOr(bools)`	**Mindestens eine** ist wahr	„Mindestens eine Fachkraft im Dienst“

Constraint	Bedeutung	Typische Anwendung
`AddImplication(a, b)`	$a \Rightarrow b$	„Wenn Frühschicht, dann keine Nachtschicht davor“
`NewIntervalVar(start, size, end, name)`	Zeitintervall mit Start, Dauer, Ende	Aufträge, Operationen, Belegungen
`AddNoOverlap(intervals)`	Intervalle überschneiden sich **nicht**	Eine Maschine, ein Raum, eine Person
`AddCumulative(intervals, bedarfe, kapazitaet)`	Zu **keinem Zeitpunkt** übersteigt der Parallelverbrauch die Kapazität	Personalstärke, Stromlast, Kranstunden
`AddCircuit(arcs)`	Die gewählten Kanten bilden **einen** Rundweg	TSP-artige Reihenfolgeprobleme
`.OnlyEnforceIf(bool)`	Bedingung gilt **nur wenn**	Reifizierung: Regel an-/abschaltbar machen

Formel-Übersetzer: was ein globales Constraint zusichert

Globale Constraints tragen englische Methodennamen, meinen aber alltägliche Regeln. Links die mathematische Zusicherung, rechts der Satz, den Sie im Planungsgespräch sagen würden:

Mathematik	Alltagssprache
$\sum_{p \in P} x_{p,s} = 1 \quad \forall s$ (`AddExactlyOne`)	„Jede Schicht wird von **genau einer** Person übernommen — keine bleibt offen, keine ist doppelt besetzt.“
$\sum_{p \in P} x_{p,s} \leq 1$ (`AddAtMostOne`)	„Höchstens einer — es darf aber auch niemand.“
$x_i \neq x_j \quad \forall i \neq j$ (`AddAllDifferent`)	„Keine zwei bekommen dasselbe“ — dieselbe Maschine, denselben Raum, dieselbe Startzeit.
$a \Rightarrow b$ (`AddImplication`)	„Wenn das eine gilt, muss auch das andere gelten.“ Die Umkehrung ist damit **nicht** gefordert.
$[s_i, e_i) \cap [s_j, e_j) = \varnothing$ (`AddNoOverlap`)	„Diese Ressource kann immer nur eines auf einmal — was sich zeitlich überschneidet, ist verboten.“

Mathematik	Alltagssprache
$\sum_{i:\, t\in[s_i,e_i)} r_i \le C \quad \forall t$ (`AddCumulative`)	„Zu **keinem Zeitpunkt** dürfen mehr als C Einheiten gleichzeitig gebraucht werden“ — Personalstärke, Stromlast, Kranstunden.
Bedingung gilt nur falls $b = 1$ (`.OnlyEnforceIf(b)`)	„Diese Regel gilt nur, wenn … “ — so wird aus einer festen Regel eine ein- und ausschaltbare.

Der Unterschied zu MILP in einem Satz: Dort formuliert man *Ungleichungen, aus denen die Regel folgt*; hier schreibt man **die Regel selbst** hin.

 Warum globale Constraints so viel bringen `AddAllDifferent` über 9 Variablen ersetzt $\binom{9}{2} = 36$ paarweise Ungleichheitsbedingungen — und propagiert **stärker** als die 36 Einzelbedingungen zusammen, weil der Propagator die Struktur als Ganzes betrachtet (er nutzt Matching-Theorie auf bipartiten Graphen). Ein MILP-Modell derselben Regel bräuchte Big-M-Konstruktionen und würde deutlich langsamer laufen.

7.5 Praxisfall: Dynamisches Vertretungssystem

Problemstellung. Eine Schule muss nach spontanen Ausfällen vier Unterrichtsstunden nachbesetzen.

- 4 offene Stunden (Zeitslots 1–4), 5 verfügbare Lehrkräfte.
- **Harte Regeln:**
  1. Jede Stunde wird von **genau einer** Person übernommen.
  2. Niemand kann an zwei Orten gleichzeitig sein.
  3. **Qualifikation:** Nur fachlich qualifizierte Personen dürfen übernehmen.
  4. **Maximallast:** Höchstens 2 Vertretungsstunden pro Person und Tag.
- **Weiche Ziele** (über Strafkosten):
  - Personen mit hoher Vorbelastung möglichst schonen.
  - **Freistundenlöcher** vermeiden (Stunde 1 und 3 arbeiten, Stunde 2 frei).
  - Last fair verteilen.

```
#!/usr/bin/env python3

# CP_SAT_Vertretungssystem.py
"""
Kapitel CP-SAT: Dynamisches Vertretungs- und Einsatzplanungssystem mit CP-SAT.

Eigenschaften:
  * Solver-Status wird ueber SolverStatus aus or_kern.py ausgewertet - dieselbe
    Fallunterscheidung wie bei jedem anderen Solver im Buch
  * Strafkosten-Zerlegung wird ausgewiesen (Erklaerbarkeit, Kapitel Praxisfallen)
  * Fairness-Kriterium ergaenzt (Spannweite der Belastung minimieren)
  * Vorabdiagnose auf offensichtliche Unloesbarkeit
  * Abnahmepruefung des fertigen Plans OHNE den Solver zu fragen
```

```
Benoetigt: ortools, pandas, pydantic (ueber or_kern)
"""

from __future__ import annotations

from ortools.sat.python import cp_model
import pandas as pd

from or_kern import Loesung, SolverStatus, status_von_cpsat

# --- 1. Datenbasis ----------------------------------------------------------
SLOTS = [1, 2, 3, 4]
FAECHER = ["Mathematik", "Physik", "Mathematik", "Informatik"]
KLASSEN = ["Klasse 8a", "Klasse 10b", "Klasse 7a", "Klasse 9c"]

PERSONAL = ["Frau_Mueller", "Herr_Schmidt", "Frau_Albrecht", "Herr_Bauer", "Frau_Koch"]

QUALIFIKATION = {
    "Frau_Mueller":  {"Mathematik", "Physik"},
    "Herr_Schmidt":  {"Informatik", "Mathematik"},
    "Frau_Albrecht": {"Physik", "Informatik"},
    "Herr_Bauer":    {"Mathematik"},
    "Frau_Koch":     {"Mathematik", "Physik", "Informatik"},
}

VORBELASTUNG = {                      # bereits geleistete Stunden heute
    "Frau_Mueller": 1, "Herr_Schmidt": 0, "Frau_Albrecht": 2,
    "Herr_Bauer": 0, "Frau_Koch": 1,
}

MAX_VERTRETUNGEN = 2
STRAFE_VORBELASTUNG = 50              # je Stunde Vorbelastung und Zuweisung
STRAFE_LOCH = 80                      # je Freistundenloch
STRAFE_UNFAIRNESS = 30                # je Einheit Spannweite der Gesamtbelastung


def pruefe_grundsaetzliche_loesbarkeit() -> bool:
    """Vorabdiagnose: Gibt es fuer jeden Slot ueberhaupt qualifiziertes Personal?"""
    ok = True
    for s, fach in enumerate(FAECHER):
        kandidaten = [p for p in PERSONAL if fach in QUALIFIKATION[p]]
        if not kandidaten:
            print(f"  UNLOESBAR: Fuer Slot {SLOTS[s]} ({fach}) gibt es niemanden.")
            ok = False
    kapazitaet = len(PERSONAL) * MAX_VERTRETUNGEN
    if kapazitaet < len(SLOTS):
        print(f"  UNLOESBAR: Kapazitaet {kapazitaet} < {len(SLOTS)} offene Stunden.")
```

```
        ok = False
    return ok


def pruefe_plan(plan: dict[int, str]) -> list[str]:
    """Prueft den fertigen Plan gegen die harten Regeln - ohne den Solver.

    Der Solver kann nur pruefen, was ihm gesagt wurde. Diese Funktion prueft
    gegen die ANFORDERUNG und benutzt dafuer bewusst keine Modellvariable
    (siehe Kapitel Praxisfallen). Leere Liste heisst bestanden.
    """
    beanstandungen: list[str] = []
    if sorted(plan) != list(range(len(SLOTS))):
        beanstandungen.append(f"nicht jede Stunde genau einmal besetzt: {sorted(plan)}")
        return beanstandungen
    for s, person in plan.items():
        if FAECHER[s] not in QUALIFIKATION[person]:
            beanstandungen.append(f"{person} ist nicht fuer {FAECHER[s]} qualifiziert")
    for person in PERSONAL:
        anzahl = sum(1 for p in plan.values() if p == person)
        if anzahl > MAX_VERTRETUNGEN:
            beanstandungen.append(f"{person} hat {anzahl} Vertretungen "
                                  f"(hoechstens {MAX_VERTRETUNGEN})")
    return beanstandungen


def plane_vertretung() -> list[dict[str, str]] | None:
    if not pruefe_grundsaetzliche_loesbarkeit():
        return None

    modell = cp_model.CpModel()

    # Entscheidungsvariablen: x[person, slot] = 1  <=>  Person uebernimmt Slot
    x = {(p, s): modell.NewBoolVar(f"zuweisung_{p}_slot{s+1}")
         for p in PERSONAL for s in range(len(SLOTS))}

    # --- HARTE NEBENBEDINGUNGEN ------------------------------------------
    # H1: Jede Stunde genau einmal besetzen
    for s in range(len(SLOTS)):
        modell.AddExactlyOne(x[p, s] for p in PERSONAL)

    # H2: Qualifikation - unqualifizierte Zuweisung ausschliessen
    for s, fach in enumerate(FAECHER):
        for p in PERSONAL:
            if fach not in QUALIFIKATION[p]:
                modell.Add(x[p, s] == 0)

    # H3: Hoechstens MAX_VERTRETUNGEN Stunden je Person
```

```
    for p in PERSONAL:
        modell.Add(sum(x[p, s] for s in range(len(SLOTS))) <= MAX_VERTRETUNGEN)

    # (H4 "nicht an zwei Orten gleichzeitig" ist durch H1 bereits erfuellt:
    #  jeder Slot hat genau eine Person, und eine Person kann pro Slot nur
    #  eine Variable auf 1 setzen.)

    # --- WEICHE ZIELE ------------------------------------------------
    strafterme = []

    # W1: Vorbelastete Personen schonen
    strafe_last = []
    for p in PERSONAL:
        zuweisungen = sum(x[p, s] for s in range(len(SLOTS)))
        strafe_last.append(VORBELASTUNG[p] * STRAFE_VORBELASTUNG * zuweisungen)
    strafterme.extend(strafe_last)

    # W2: Freistundenloecher vermeiden (Muster: arbeitet, frei, arbeitet)
    loch_variablen = []
    for p in PERSONAL:
        for s in range(len(SLOTS) - 2):
            loch = modell.NewBoolVar(f"loch_{p}_{s}")
            # Reifizierung: loch == 1  <=>  (x_s AND NOT x_{s+1} AND x_{s+2})
            modell.AddBoolAnd([x[p, s], x[p, s + 1].Not(), x[p, s + 2]]).OnlyEnforceIf(loch)
            modell.AddBoolOr([x[p, s].Not(), x[p, s + 1], x[p, s + 2].Not()]) \
                  .OnlyEnforceIf(loch.Not())
            loch_variablen.append(loch)
            strafterme.append(loch * STRAFE_LOCH)

    # W3: Fairness - Spannweite der Gesamtbelastung minimieren
    gesamtlast = {}
    for p in PERSONAL:
        last = modell.NewIntVar(0, len(SLOTS) + max(VORBELASTUNG.values()), f"last_{p}")
        modell.Add(last == VORBELASTUNG[p] + sum(x[p, s] for s in range(len(SLOTS))))
        gesamtlast[p] = last
    max_last = modell.NewIntVar(0, 10, "max_last")
    min_last = modell.NewIntVar(0, 10, "min_last")
    modell.AddMaxEquality(max_last, list(gesamtlast.values()))
    modell.AddMinEquality(min_last, list(gesamtlast.values()))
    spannweite = modell.NewIntVar(0, 10, "spannweite")
    modell.Add(spannweite == max_last - min_last)
    strafterme.append(spannweite * STRAFE_UNFAIRNESS)

    modell.Minimize(sum(strafterme))

    # --- LOESEN ------------------------------------------------------
    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = 5.0
```

```
    # Reproduzierbarkeit: ein Arbeiter, fester Startwert. Ohne das kann CP-SAT
    # bei mehreren gleich guten Plaenen von Lauf zu Lauf einen anderen liefern
    # (siehe JobShop_Intervalle.py im naechsten Abschnitt).
    loeser.parameters.num_workers = 1
    loeser.parameters.random_seed = 1
    rohstatus = loeser.Solve(modell)

    # Der Rueckgabewert wird in die gemeinsame Sprache uebersetzt. Ab hier
    # sieht die Auswertung genauso aus wie bei HiGHS, GLOP oder CVXPY.
    status = status_von_cpsat(rohstatus)
    loesung = Loesung(
        status=status,
        zielwert=loeser.ObjectiveValue() if status.brauchbar else None,
        schranke=loeser.BestObjectiveBound() if status.brauchbar else None,
        laufzeit=loeser.WallTime())

    print("=" * 78)
    print("     DYNAMISCHER VERTRETUNGSPLAN (CP-SAT OPTIMIERT)")
    print("=" * 78)

    if status.modellfehler:
        print(f"Kein zulaessiger Plan moeglich (Status: {status.value}).")
        print("Die harten Regeln widersprechen sich. Naechster Schritt: eine Regel")
        print("weich machen (Kapitel Praxisfallen) oder den Konflikt eingrenzen")
        print("(Anhang Fehlerdiagnose).")
        return None
    if not status.brauchbar:
        print(f"Solver lieferte kein Ergebnis (Status: {status.value}).")
        print("Zeitlimit erhoehen oder das Modell vereinfachen.")
        return None

    guete = ("beweisbar optimal" if status is SolverStatus.OPTIMAL
             else "zulaessig, aber nicht bewiesen - Gap beachten")
    print(f"Solver-Status: {loeser.StatusName(rohstatus)} -> {status.value} ({guete})")
    print(f"{loesung.als_bericht()}\n")

    # --- Plan ausgeben ------------------------------------------------
    zuweisung = {s: next(p for p in PERSONAL if loeser.Value(x[p, s]) == 1)
                 for s in range(len(SLOTS))}
    plan = [{
        "Stunde": f"Std. {SLOTS[s]}",
        "Klasse": KLASSEN[s],
        "Fach": FAECHER[s],
        "Vertretung": zuweisung[s],
        "Vorbelastung": f"{VORBELASTUNG[zuweisung[s]]} Std.",
    } for s in range(len(SLOTS))]
    print(pd.DataFrame(plan).to_string(index=False))
```

```
    # --- Abnahmepruefung gegen die Anforderung ----------------------------
    beanstandungen = pruefe_plan(zuweisung)
    if beanstandungen:
        print("\nABNAHMEPRUEFUNG FEHLGESCHLAGEN:")
        for beanstandung in beanstandungen:
            print(f"  - {beanstandung}")
        return None
    print("\nAbnahmepruefung: bestanden (Besetzung, Qualifikation, Hoechstzahl).")

    # --- Erklaerbarkeit: Strafkosten aufschluesseln ----------------------
    kosten_last = sum(VORBELASTUNG[p] * STRAFE_VORBELASTUNG
                      * sum(loeser.Value(x[p, s]) for s in range(len(SLOTS)))
                      for p in PERSONAL)
    kosten_loecher = sum(loeser.Value(v) for v in loch_variablen) * STRAFE_LOCH
    kosten_fairness = loeser.Value(spannweite) * STRAFE_UNFAIRNESS

    print("\n--- Woraus bestehen die Strafkosten? ---")
    print(f"  Vorbelastung geschont:  {kosten_last:5.0f} Punkte")
    print(f"  Freistundenloecher:     {kosten_loecher:5.0f} Punkte "
          f"({sum(loeser.Value(v) for v in loch_variablen)} Loch/Loecher)")
    print(f"  Fairness (Spannweite {loeser.Value(spannweite)}): {kosten_fairness:5.0f}
     ↪  Punkte")
    print(f"  {'Summe':<23} {loeser.ObjectiveValue():5.0f} Punkte")

    print("\n--- Auslastung nach Optimierung ---")
    for p in PERSONAL:
        heute = sum(loeser.Value(x[p, s]) for s in range(len(SLOTS)))
        print(f"  * {p:<15}: {VORBELASTUNG[p]} vorher + {heute} Vertretung "
              f"= {VORBELASTUNG[p] + heute} Stunden")

    print(f"\nSuchstatistik: {loeser.NumBranches()} Verzweigungen, "
          f"{loeser.NumConflicts()} Konflikte, {loeser.WallTime():.3f} s")
    print("=" * 78)
    return plan


if __name__ == "__main__":
    plane_vertretung()
```

Erwartete Ausgabe (Laufzeit und Suchstatistik sind hardwareabhängig):

```
==============================================================================
     DYNAMISCHER VERTRETUNGSPLAN (CP-SAT OPTIMIERT)
==============================================================================
Solver-Status: OPTIMAL -> optimal (beweisbar optimal)
Status: optimal | Zielwert: 80.00 | Gap: 0.00% | Zeit: 0.01s

Stunde     Klasse       Fach   Vertretung Vorbelastung
```

```
Std. 1  Klasse 8a Mathematik Herr_Schmidt       0 Std.
Std. 2 Klasse 10b     Physik Frau_Mueller       1 Std.
Std. 3  Klasse 7a Mathematik   Herr_Bauer       0 Std.
Std. 4  Klasse 9c Informatik Herr_Schmidt       0 Std.

Abnahmepruefung: bestanden (Besetzung, Qualifikation, Hoechstzahl).

--- Woraus bestehen die Strafkosten? ---
  Vorbelastung geschont:     50 Punkte
  Freistundenloecher:         0 Punkte (0 Loch/Loecher)
  Fairness (Spannweite 1):    30 Punkte
  Summe                      80 Punkte

--- Auslastung nach Optimierung ---
  * Frau_Mueller   : 1 vorher + 1 Vertretung = 2 Stunden
  * Herr_Schmidt   : 0 vorher + 2 Vertretung = 2 Stunden
  * Frau_Albrecht  : 2 vorher + 0 Vertretung = 2 Stunden
  * Herr_Bauer     : 0 vorher + 1 Vertretung = 1 Stunden
  * Frau_Koch      : 1 vorher + 0 Vertretung = 1 Stunden

Suchstatistik: 113 Verzweigungen, 3 Konflikte, 0.006 s
================================================================================
```

Code-Durchgang: die drei wichtigsten Stellen

1. Reifizierung (W2). Die zwei Zeilen

```
modell.AddBoolAnd([x[p,s], x[p,s+1].Not(), x[p,s+2]]).OnlyEnforceIf(loch)
modell.AddBoolOr([x[p,s].Not(), x[p,s+1],
 ↪  x[p,s+2].Not()]).OnlyEnforceIf(loch.Not())
```

koppeln die Hilfsvariable `loch` **in beide Richtungen** an das Muster „arbeitet–frei–arbeitet“. Die erste Zeile erzwingt: *wenn* `loch = 1`, *dann* liegt das Muster vor. Die zweite: *wenn* `loch = 0`, *dann* liegt es nicht vor. In diesem Modell genügte streng genommen die erste Zeile, weil `loch` nur mit positiven Strafkosten vorkommt und der Minimierer es daher von selbst auf 0 drückt — die zweite Zeile macht das Modell aber robust gegen spätere Änderungen und ist selbstdokumentierend.

2. Fairness über Spannweite (W3). `AddMaxEquality` und `AddMinEquality` sind CP-SAT-Konstrukte, die in MILP mehrere Big-M-Ungleichungen bräuchten. Die Spannweite max − min zu minimieren ist ein Standardrezept für „gleichmäßig verteilen“.

3. Kostenzerlegung. Der Block „Woraus bestehen die Strafkosten?“ ist kein Beiwerk. In der Praxis ist die erste Frage jedes Anwenders: *„Warum bekomme **ich** die Stunde?“* Wer darauf keine Antwort hat, dessen System wird nicht benutzt (Kapitel 22, *Explainable OR*).

4. Statusauswertung ohne CP-SAT-Vokabular. Nach `Solve()` steht nur eine einzige Zeile mit einer CP-SAT-Konstante darin:

```
status = status_von_cpsat(rohstatus)
```

Danach heißen die Fälle `status.modellfehler` (die Regeln widersprechen sich — das Modell muss geändert werden) und `status.brauchbar` (es liegt etwas Ausführbares vor). Das ist **dieselbe** Fallunterscheidung wie bei HiGHS in Kapitel 6 und bei GLOP in Kapitel 5; nur die eine Übersetzerzeile weiß, welcher Solver gerechnet hat. Wer den Auswertungscode direkt gegen `cp_model.INFEASIBLE` schreibt, kann ihn bei einem Solverwechsel wegwerfen — Abschnitt 22.7 führt genau diesen Wechsel vor.

Typische Fehler

- **Zu viele harte Regeln.** Jede zusätzliche harte Bedingung erhöht das Risiko von `INFEASIBLE`. Faustregel: Hart ist nur, was rechtlich oder physikalisch unmöglich ist. Alles andere wird weich mit hoher Strafe — dann liefert der Solver den „am wenigsten schlechten" Plan statt einer Fehlermeldung.
- **Strafgewichte willkürlich wählen.** Die Gewichte (50, 80, 30) definieren implizit eine Werteordnung: „Ein Freistundenloch ist so schlimm wie 1,6 Stunden Vorbelastung." Besprechen Sie diese Zahlen mit den Betroffenen — sie sind eine fachliche, keine technische Entscheidung.
- **Gleitkommazahlen in CP-SAT.** CP-SAT rechnet **ausschließlich ganzzahlig**. Wer Euro und Cent braucht, rechnet in Cent.

7.6 Intervallvariablen: Job-Shop-Scheduling

Die Königsdisziplin von CP-SAT sind Zeitplanungsprobleme. Intervallvariablen sind dabei das zentrale Werkzeug.

Problem. Drei Aufträge, je drei Arbeitsgänge auf drei Maschinen. Jeder Auftrag muss seine Gänge in fester Reihenfolge durchlaufen; jede Maschine kann nur einen Gang gleichzeitig bearbeiten. Gesucht: der Plan mit der kürzesten Gesamtdauer (*Makespan*).

```
#!/usr/bin/env python3

# JobShop_Intervalle.py
"""
Kapitel CP-SAT: Job-Shop-Scheduling mit Intervallvariablen.
Zeigt NewIntervalVar, AddNoOverlap und die Minimierung des Makespan.
"""

import collections

from ortools.sat.python import cp_model

# (Maschine, Dauer) je Arbeitsgang, in der Reihenfolge des Auftrags
AUFTRAEGE = [
    [(0, 3), (1, 2), (2, 2)],      # Auftrag 0
```

```
    [(0, 2), (2, 1), (1, 4)],      # Auftrag 1
    [(1, 4), (2, 3)],              # Auftrag 2
]
MASCHINENNAMEN = ["Fraese", "Dreherei", "Lackiererei"]


def loese_jobshop():
    anzahl_maschinen = 1 + max(m for auftrag in AUFTRAEGE for m, _ in auftrag)
    horizont = sum(dauer for auftrag in AUFTRAEGE for _, dauer in auftrag)

    modell = cp_model.CpModel()
    Gang = collections.namedtuple("Gang", "start ende intervall")
    plaene = {}
    maschinen_intervalle = collections.defaultdict(list)

    # --- Intervallvariablen anlegen -------------------------------------
    for a, auftrag in enumerate(AUFTRAEGE):
        for g, (maschine, dauer) in enumerate(auftrag):
            start = modell.NewIntVar(0, horizont, f"start_{a}_{g}")
            ende = modell.NewIntVar(0, horizont, f"ende_{a}_{g}")
            # Ein Intervall koppelt start + dauer == ende automatisch
            intervall = modell.NewIntervalVar(start, dauer, ende, f"intervall_{a}_{g}")
            plaene[a, g] = Gang(start, ende, intervall)
            maschinen_intervalle[maschine].append(intervall)

    # --- H1: eine Maschine bearbeitet nur einen Gang gleichzeitig --------
    for maschine in range(anzahl_maschinen):
        modell.AddNoOverlap(maschinen_intervalle[maschine])

    # --- H2: Reihenfolge innerhalb eines Auftrags einhalten -------------
    for a, auftrag in enumerate(AUFTRAEGE):
        for g in range(len(auftrag) - 1):
            modell.Add(plaene[a, g + 1].start >= plaene[a, g].ende)

    # --- Ziel: Makespan minimieren --------------------------------------
    makespan = modell.NewIntVar(0, horizont, "makespan")
    modell.AddMaxEquality(makespan, [plaene[a, len(auftrag) - 1].ende
                                     for a, auftrag in enumerate(AUFTRAEGE)])
    modell.Minimize(makespan)

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = 10.0
    # Es gibt mehrere Plaene mit demselben kuerzesten Makespan. Welchen CP-SAT
    # findet, haengt sonst davon ab, welcher seiner parallelen Suchstraenge
    # zuerst fertig wird - dasselbe Programm liefert dann von Lauf zu Lauf
    # verschiedene (gleich gute) Plaene. Fuer ein reproduzierbares Buchbeispiel
    # fixieren wir beides. Im Produktivbetrieb laesst man die Standardwerte
    # stehen: Mehrere Arbeiter sind dort deutlich schneller.
```

```
    loeser.parameters.num_workers = 1
    loeser.parameters.random_seed = 1
    status = loeser.Solve(modell)

    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        print(f"Kein Plan gefunden: {loeser.StatusName(status)}")
        return

    print("=" * 74)
    print("  JOB-SHOP-SCHEDULING MIT INTERVALLVARIABLEN")
    print("=" * 74)
    print(f"Status: {loeser.StatusName(status)} | "
          f"Kuerzeste Gesamtdauer (Makespan): {loeser.Value(makespan)} Zeiteinheiten\n")

    # --- Gantt-Diagramm als Text ----------------------------------------
    dauer_gesamt = loeser.Value(makespan)
    print(f"{'Maschine':<14}|" + "".join(f"{t:<3}" for t in range(dauer_gesamt)))
    print("-" * (15 + 3 * dauer_gesamt))
    for maschine in range(anzahl_maschinen):
        zeile = [" . "] * dauer_gesamt
        for a, auftrag in enumerate(AUFTRAEGE):
            for g, (m, dauer) in enumerate(auftrag):
                if m == maschine:
                    beginn = loeser.Value(plaene[a, g].start)
                    for t in range(beginn, beginn + dauer):
                        zeile[t] = f" A{a}"
        print(f"{MASCHINENNAMEN[maschine]:<14}|" + "".join(zeile))

    print("\n--- Detailplan ---")
    for a, auftrag in enumerate(AUFTRAEGE):
        teile = []
        for g, (maschine, dauer) in enumerate(auftrag):
            beginn = loeser.Value(plaene[a, g].start)
            teile.append(f"{MASCHINENNAMEN[maschine]} {beginn}-{beginn + dauer}")
        print(f"  Auftrag {a}: " + "  ->  ".join(teile))

    # --- Untere Schranke zum Vergleich ----------------------------------
    laengster_auftrag = max(sum(d for _, d in a) for a in AUFTRAEGE)
    hoechste_maschinenlast = max(
        sum(d for auftrag in AUFTRAEGE for m, d in auftrag if m == maschine)
        for maschine in range(anzahl_maschinen))
    schranke = max(laengster_auftrag, hoechste_maschinenlast)
    print(f"\nUntere Schranke (laengster Auftrag / hoechste Maschinenlast): {schranke}")
    print(f"Erreichter Makespan: {loeser.Value(makespan)}"
          f"{'  -> beweisbar bestmoeglich' if loeser.Value(makespan) == schranke else ''}")
    print("=" * 74)
```

```
if __name__ == "__main__":
    loese_jobshop()
```

Erwartete Ausgabe:

```
============================================================================
  JOB-SHOP-SCHEDULING MIT INTERVALLVARIABLEN
============================================================================
Status: OPTIMAL | Kuerzeste Gesamtdauer (Makespan): 11 Zeiteinheiten

Maschine      |0  1  2  3  4  5  6  7  8  9  10
-----------------------------------------------
Fraese        | A1 A1 A0 A0 A0 .  .  .  .  .  .
Dreherei      | A2 A2 A2 A2 .  A0 A0 A1 A1 A1 A1
Lackiererei   | .  .  A1 .  A2 A2 A2 A0 A0 .  .

--- Detailplan ---
  Auftrag 0: Fraese 2-5  ->  Dreherei 5-7  ->  Lackiererei 7-9
  Auftrag 1: Fraese 0-2  ->  Lackiererei 2-3  ->  Dreherei 7-11
  Auftrag 2: Dreherei 0-4  ->  Lackiererei 4-7

Untere Schranke (laengster Auftrag / hoechste Maschinenlast): 10
Erreichter Makespan: 11
============================================================================
```

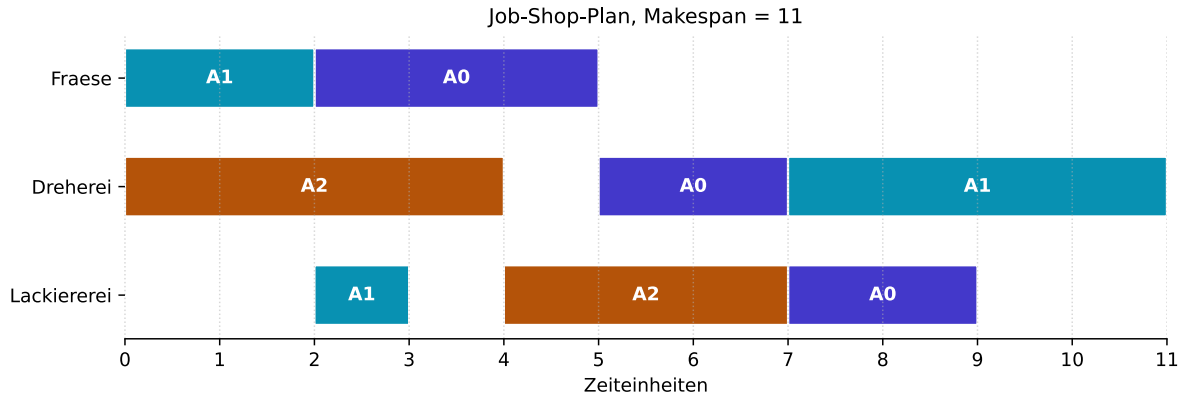

Abbildung 13: Abb. 7.2: Job-Shop-Plan als Gantt-Diagramm

Eine interaktive Fassung dieser Grafik steht auf den Kapitelseiten der Website bereit.

Lesen Sie das Gantt-Diagramm. Auftrag 1 kommt zuerst an die Fräse (0–2), Auftrag 0 muss warten und ist dort erst von 2 bis 5 an der Reihe. Auffällig ist die Lücke in der Dreherei bei Zeitpunkt 4: Die Maschine steht still, obwohl Auftrag 0 schon Arbeit hätte — er ist aber bis 5 noch an der Fräse. Dieses **Warten ist unvermeidlich**; es entsteht aus dem Zusammenspiel von Reihenfolge- und Maschinenbedingungen, nicht aus einer schlechten Planung.

□ Mehrere gleich gute Pläne — und was das für Ihre Tests bedeutet

Der kürzeste Makespan beträgt 11, aber es gibt **mehrere verschiedene Pläne**, die ihn erreichen. Welchen CP-SAT findet, hängt normalerweise davon ab, welcher seiner parallelen Suchstränge zuerst fertig wird — dasselbe Programm liefert dann von Lauf zu Lauf einen anderen (gleich guten) Plan.

Deshalb setzt das Programm oben `num_workers = 1` und `random_seed = 1`: nur so ist die abgedruckte Ausgabe reproduzierbar. **Im Produktivbetrieb lässt man beides weg** — mehrere Arbeiter sind deutlich schneller.

Wie viel schneller, und warum der Seed allein **nicht** genügt, misst Abschnitt 7.7 an genau diesem Modell.

Die Lehre für Ihre eigenen Tests: Prüfen Sie bei mehrdeutigen Optima nie auf einen **bestimmten Plan**, sondern immer auf den **Zielwert** und auf die Einhaltung aller Regeln. Ein Test wie `assert plan == erwarteter_plan` besteht sonst mal und scheitert mal, ohne dass sich am Code etwas geändert hätte.

Bemerkenswert ist die letzte Zeile: Die untere Schranke liegt bei 10, erreicht wurden 11. Die Schranke ist also **nicht erreichbar** — kein Plan kann 10 schaffen, obwohl weder ein einzelner Auftrag noch eine einzelne Maschine mehr als 10 Zeiteinheiten braucht. Der Grund sind die Wechselwirkungen. Das ist typisch für Scheduling: **Einfache Schranken sind selten scharf.** Trotzdem sind sie nützlich — sie sagten uns hier, dass wir höchstens 10 % vom theoretischen Ideal entfernt sein können, noch bevor der Solver fertig war.

Warum Intervallvariablen so wertvoll sind

Eine Intervallvariable bündelt drei Größen (Start, Dauer, Ende) und garantiert intern `start + dauer == ende`. Der eigentliche Gewinn ist aber `AddNoOverlap`: Diese eine Zeile ersetzt für k Gänge auf einer Maschine $\binom{k}{2}$ Entweder-Oder-Konstruktionen mit je zwei Big-M-Ungleichungen und einer Binärvariablen. Bei 10 Gängen sind das 45 Disjunktionen — in MILP ein zäher Block, in CP-SAT eine Zeile mit einem spezialisierten Propagator.

7.7 Parallele Suche: was `num_workers` wirklich bewirkt

An mehreren Stellen dieses Buchs steht `loeser.parameters.num_workers = 1`, mit dem Kommentar „für eine reproduzierbare Ausgabe“. Darin stecken zwei Behauptungen: dass parallele Suche **schneller** ist, und dass sie **von Lauf zu Lauf andere, gleich gute** Lösungen liefert. Beides lässt sich messen — und beim Messen kommt mehr heraus als erwartet.

Gerechnet wird auf demselben Modell wie Abschnitt 7.6, nur groß genug, dass die Suche wirklich arbeitet: 12 Aufträge auf 10 Maschinen, jeder Auftrag besucht jede Maschine. Entscheidend ist, dass `random_seed` in **jedem** Lauf denselben Wert hat.

```
#!/usr/bin/env python3

# Parallele_Suche.py
"""
```

```
"""Kapitel CP-SAT: Was mehrere Arbeiter wirklich bringen - und was sie kosten.

Das Buch setzt an mehreren Stellen 'num_workers = 1', damit die abgedruckte
Ausgabe reproduzierbar ist. Behauptet wird dabei zweierlei: dass parallele
Suche schneller ist, und dass sie von Lauf zu Lauf verschiedene, gleich gute
Loesungen findet. Beides wird hier gemessen statt geglaubt.

Gerechnet wird auf einem Job-Shop wie in JobShop_Intervalle.py, nur gross
genug, dass die Suche wirklich arbeitet: 12 Auftraege auf 10 Maschinen,
jeder Auftrag besucht jede Maschine.

Zwei Messungen:
  1. Dieselbe Aufgabe mit 1, 2, 4 und 8 Arbeitern - Laufzeit und Ergebnis.
  2. Dieselbe Konfiguration mehrfach, bei FESTEM random_seed - wie oft kommt
     derselbe Plan heraus?

Die zweite ist die wichtigere: Sie entscheidet, wie man Tests schreibt.

Achtung, Laufzeiten sind hardwareabhaengig. Die ZAHL der verschiedenen
Plaene ist es auch - und genau das ist die Aussage.

Benoetigt: numpy, ortools
"""

from __future__ import annotations

import collections
import time

import numpy as np
from ortools.sat.python import cp_model

AUFTRAEGE = 12
MASCHINEN = 10
INSTANZ_SEED = 20260908
SOLVER_SEED = 1          # bleibt ueber ALLE Laeufe gleich - das ist der Punkt
ARBEITERZAHLEN = (1, 2, 4, 8)
WIEDERHOLUNGEN = 4
ZEITLIMIT = 120.0


def baue_instanz() -> list[list[tuple[int, int]]]:
    """Klassischer Job-Shop: Jeder Auftrag besucht jede Maschine genau einmal,
    in einer eigenen zufaelligen Reihenfolge."""
    rng = np.random.default_rng(INSTANZ_SEED)
    return [[(int(m), int(rng.integers(2, 20))) for m in rng.permutation(MASCHINEN)]
            for _ in range(AUFTRAEGE)]
```

```
def loese(auftraege, arbeiter: int):
    """Minimiert den Makespan. Gibt Status, Zielwert, Laufzeit und den Plan
    zurueck - den Plan als Tupel aller Startzeiten, damit sich zwei Laeufe
    vergleichen lassen."""
    horizont = sum(dauer for auftrag in auftraege for _, dauer in auftrag)
    modell = cp_model.CpModel()
    Gang = collections.namedtuple("Gang", "start ende intervall")
    plaene: dict[tuple[int, int], Gang] = {}
    je_maschine = collections.defaultdict(list)

    for a, auftrag in enumerate(auftraege):
        for g, (maschine, dauer) in enumerate(auftrag):
            start = modell.NewIntVar(0, horizont, f"start_{a}_{g}")
            ende = modell.NewIntVar(0, horizont, f"ende_{a}_{g}")
            intervall = modell.NewIntervalVar(start, dauer, ende, f"iv_{a}_{g}")
            plaene[a, g] = Gang(start, ende, intervall)
            je_maschine[maschine].append(intervall)

    for maschine in range(MASCHINEN):
        modell.AddNoOverlap(je_maschine[maschine])
    for a, auftrag in enumerate(auftraege):
        for g in range(len(auftrag) - 1):
            modell.Add(plaene[a, g + 1].start >= plaene[a, g].ende)

    makespan = modell.NewIntVar(0, horizont, "makespan")
    modell.AddMaxEquality(
        makespan, [plaene[a, len(auftrag) - 1].ende
                   for a, auftrag in enumerate(auftraege)])
    modell.Minimize(makespan)

    loeser = cp_model.CpSolver()
    loeser.parameters.num_workers = arbeiter
    loeser.parameters.random_seed = SOLVER_SEED
    loeser.parameters.max_time_in_seconds = ZEITLIMIT

    beginn = time.perf_counter()
    status = loeser.Solve(modell)
    dauer = time.perf_counter() - beginn

    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        return loeser.StatusName(status), None, dauer, None
    plan = tuple(loeser.Value(plaene[a, g].start)
                 for a, auftrag in enumerate(auftraege)
                 for g in range(len(auftrag)))
    return loeser.StatusName(status), int(loeser.ObjectiveValue()), dauer, plan
```

```
if __name__ == "__main__":
    auftraege = baue_instanz()
    gaenge = sum(len(a) for a in auftraege)

    print("=" * 78)
    print("  WAS MEHRERE ARBEITER WIRKLICH BRINGEN")
    print("=" * 78)
    print(f"Job-Shop: {AUFTRAEGE} Auftraege, {MASCHINEN} Maschinen, "
          f"{gaenge} Arbeitsgaenge.")
    print(f"random_seed = {SOLVER_SEED} in JEDEM Lauf, "
          f"{WIEDERHOLUNGEN} Wiederholungen je Arbeiterzahl.\n")

    # Jede Konfiguration mehrfach - ein einzelner Zeitwert sagt nichts.
    ergebnisse: dict[int, list[tuple]] = {}
    for arbeiter in ARBEITERZAHLEN:
        ergebnisse[arbeiter] = [loese(auftraege, arbeiter)
                                for _ in range(WIEDERHOLUNGEN)]

    print("=" * 78)
    print("  (1) Laufzeit: was die Arbeiter bringen")
    print("=" * 78)
    print(f"{'Arbeiter':>9} {'Status':>9} {'Makespan':>9} "
          f"{'schnellste':>11} {'langsamste':>11} {'Beschleunigung':>15}")
    print("-" * 78)
    basis_zeit = min(d for _, _, d, _ in ergebnisse[1])
    for arbeiter in ARBEITERZAHLEN:
        laeufe = ergebnisse[arbeiter]
        zeiten = [d for _, _, d, _ in laeufe]
        ziele = {o for _, o, _, _ in laeufe}
        status = {s for s, _, _, _ in laeufe}
        print(f"{arbeiter:>9} {'/'.join(sorted(status)):>9} "
              f"{'/'.join(str(z) for z in sorted(ziele)):>9} "
              f"{min(zeiten):>10.2f}s {max(zeiten):>10.2f}s "
              f"{basis_zeit / min(zeiten):>14.1f}x")

    beste = min(ARBEITERZAHLEN, key=lambda w: min(d for _, _, d, _ in ergebnisse[w]))
    faktor = basis_zeit / min(d for _, _, d, _ in ergebnisse[beste])
    print(f"\n  Am schnellsten sind {beste} Arbeiter: Faktor {faktor:.1f} gegenueber")
    print(f"  einem einzigen - also MEHR, als {beste} Kerne hergeben sollten.")
    print(f"  Das ist kein Messfehler. CP-SAT laesst nicht {beste}-mal dieselbe Suche")
    print("  laufen, sondern verschiedene Strategien nebeneinander, die einander")
    print("  ihre Schranken zurufen. Der zweite Arbeiter ist deshalb nicht der")
    print("  'zweite Kern', sondern ein anderes Verfahren.")
    letzte = ARBEITERZAHLEN[-1]
    if beste != letzte:
        zeit_letzte = min(d for _, _, d, _ in ergebnisse[letzte])
        print(f"\n  Und mehr ist nicht immer besser: {letzte} Arbeiter brauchen "
              f"{zeit_letzte:.2f}s")
```

```
        print(f"  gegenueber {min(d for _, _, d, _ in ergebnisse[beste]):.2f}s bei "
              f"{beste}. Ab einer gewissen Zahl kosten Abstimmung")
        print("  und Speicherbandbreite mehr, als ein weiterer Suchstrang einbringt.")

    print("\n" + "=" * 78)
    print("  (2) Reproduzierbarkeit: derselbe Seed, derselbe Plan?")
    print("=" * 78)
    print(f"{'Arbeiter':>9} {'Laeufe':>7} {'verschiedene Plaene':>21} "
          f"{'Makespan':>10}")
    print("-" * 78)
    for arbeiter in ARBEITERZAHLEN:
        plaene = {p for _, _, _, p in ergebnisse[arbeiter] if p is not None}
        ziele = {o for _, o, _, _ in ergebnisse[arbeiter]}
        print(f"{arbeiter:>9} {WIEDERHOLUNGEN:>7} {len(plaene):>21} "
              f"{'/'.join(str(z) for z in sorted(ziele)):>10}")

    einer = len({p for _, _, _, p in ergebnisse[1]})
    # Ab welcher Arbeiterzahl bricht die Reproduzierbarkeit? Das ist die
    # eigentliche Zahl - nicht, was acht Arbeiter anrichten.
    ab = next((w for w in ARBEITERZAHLEN
               if len({p for _, _, _, p in ergebnisse[w]}) > 1), None)
    print(f"\n  Mit einem Arbeiter: {einer} Plan aus {WIEDERHOLUNGEN} Laeufen.")
    if ab is not None:
        viele = len({p for _, _, _, p in ergebnisse[ab]})
        print(f"  Schon mit {ab} Arbeitern: {viele} verschiedene Plaene aus "
              f"{WIEDERHOLUNGEN} Laeufen -")
        print("  bei identischem random_seed und identischem Zielwert.")
        print(f"  Es braucht also keine acht Arbeiter, um die Reproduzierbarkeit")
        print(f"  zu verlieren. {ab} genuegen.")
    print("\n  Der Seed allein macht einen Lauf also NICHT reproduzierbar. Er legt")
    print("  fest, wie ein einzelner Suchstrang wuerfelt - nicht, welcher von")
    print("  mehreren zuerst fertig wird. Das entscheidet die Uhr.")

    print("\n" + "=" * 78)
    print("  (3) Was daraus fuer Tests folgt")
    print("=" * 78)
    print("  Ein Test der Form  assert plan == erwarteter_plan  besteht mal und")
    print("  scheitert mal, ohne dass sich am Code etwas geaendert haette.")
    print("  Zu pruefen sind stattdessen:")
    print("    * der ZIELWERT (hier in allen Laeufen gleich),")
    print("    * die Einhaltung aller Regeln (Abnahmepruefung, Kapitel Praxisfallen),")
    print("    * und der Status - nicht die Gestalt der Loesung.")
    print("\n  Wer doch einen bestimmten Plan braucht - fuer eine abgedruckte")
    print("  Ausgabe, fuer einen Regressionstest -, setzt num_workers = 1.")
    print(f"  Das kostet hier den Faktor {faktor:.1f} an Laufzeit und ist genau")
    print("  deshalb eine Entscheidung fuer den Test, nicht fuer den Betrieb.")
    print("=" * 78)
```

Erwartete Ausgabe (Laufzeiten *und* die Zahl der verschiedenen Pläne sind hardwareabhängig — genau das ist die Aussage; der Zielwert ist es nicht):

```
================================================================================
  WAS MEHRERE ARBEITER WIRKLICH BRINGEN
================================================================================
Job-Shop: 12 Auftraege, 10 Maschinen, 120 Arbeitsgaenge.
random_seed = 1 in JEDEM Lauf, 4 Wiederholungen je Arbeiterzahl.

================================================================================
  (1) Laufzeit: was die Arbeiter bringen
================================================================================
 Arbeiter    Status  Makespan  schnellste  langsamste  Beschleunigung
--------------------------------------------------------------------------------
        1   OPTIMAL       183       5.62s       5.82s            1.0x
        2   OPTIMAL       183       1.43s       2.04s            3.9x
        4   OPTIMAL       183       0.60s       0.66s            9.4x
        8   OPTIMAL       183       0.46s       0.65s           12.3x

  Am schnellsten sind 8 Arbeiter: Faktor 12.3 gegenueber
  einem einzigen - also MEHR, als 8 Kerne hergeben sollten.
  Das ist kein Messfehler. CP-SAT laesst nicht 8-mal dieselbe Suche
  laufen, sondern verschiedene Strategien nebeneinander, die einander
  ihre Schranken zurufen. Der zweite Arbeiter ist deshalb nicht der
  'zweite Kern', sondern ein anderes Verfahren.

================================================================================
  (2) Reproduzierbarkeit: derselbe Seed, derselbe Plan?
================================================================================
 Arbeiter  Laeufe   verschiedene Plaene   Makespan
--------------------------------------------------------------------------------
        1       4                     1        183
        2       4                     3        183
        4       4                     3        183
        8       4                     4        183

  Mit einem Arbeiter: 1 Plan aus 4 Laeufen.
  Schon mit 2 Arbeitern: 3 verschiedene Plaene aus 4 Laeufen -
  bei identischem random_seed und identischem Zielwert.
  Es braucht also keine acht Arbeiter, um die Reproduzierbarkeit
  zu verlieren. 2 genuegen.

  Der Seed allein macht einen Lauf also NICHT reproduzierbar. Er legt
  fest, wie ein einzelner Suchstrang wuerfelt - nicht, welcher von
  mehreren zuerst fertig wird. Das entscheidet die Uhr.

================================================================================
  (3) Was daraus fuer Tests folgt
================================================================================
```

```
  Ein Test der Form  assert plan == erwarteter_plan  besteht mal und
  scheitert mal, ohne dass sich am Code etwas geaendert haette.
  Zu pruefen sind stattdessen:
    * der ZIELWERT (hier in allen Laeufen gleich),
    * die Einhaltung aller Regeln (Abnahmepruefung, Kapitel Praxisfallen),
    * und der Status - nicht die Gestalt der Loesung.

  Wer doch einen bestimmten Plan braucht - fuer eine abgedruckte
  Ausgabe, fuer einen Regressionstest -, setzt num_workers = 1.
  Das kostet hier den Faktor 12.3 an Laufzeit und ist genau
  deshalb eine Entscheidung fuer den Test, nicht fuer den Betrieb.
==========================================================================
```

Was die Messung zeigt

Erstens: Die Beschleunigung ist überlinear. Acht Arbeiter sind im abgedruckten Lauf nicht achtmal, sondern **gut zwölfmal** schneller als einer. Das sieht nach Messfehler aus, ist aber die korrekte Beschreibung dessen, was CP-SAT tut: Es lässt **nicht** achtmal dieselbe Suche auf verschiedenen Kernen laufen, sondern **verschiedene Strategien nebeneinander** — Suche mit LP-Relaxation, Suche mit anderen Verzweigungsregeln, lokale Suche —, und die teilen einander ihre Schranken mit. Der zweite Arbeiter ist deshalb nicht „der zweite Kern“, sondern ein anderes Verfahren, das zufällig auch rechnet.

Daraus folgt zweierlei. Ein Speedup über der Kernzahl ist bei CP-SAT normal und kein Grund zum Misstrauen. Und der Zugewinn flacht ab: Zwischen vier und acht Arbeitern lag in unseren Wiederholungen mal das eine, mal das andere vorn — ab einer gewissen Zahl kosten Abstimmung und Speicherbandbreite mehr, als ein weiterer Suchstrang einbringt. Auch hier gilt: messen, nicht schätzen.

Zweitens — und das ist der Befund, der die Testpraxis bestimmt: Der Seed genügt nicht. Bei einem Arbeiter liefern vier Läufe **einen** Plan. Schon bei **zwei** Arbeitern liefern vier Läufe **drei verschiedene** — bei identischem `random_seed` und identischem Zielwert 183.

Es braucht also keine acht Arbeiter, um die Reproduzierbarkeit zu verlieren. Zwei genügen. Der Grund ist einfach, sobald man ihn ausspricht: Der Seed legt fest, wie ein **einzelner** Suchstrang würfelt. Er legt nicht fest, **welcher von mehreren zuerst fertig wird** — das entscheidet die Uhr, und die ist bei jedem Lauf anders.

> **`random_seed` allein macht nichts reproduzierbar** Ein häufiger Irrtum: „Ich habe doch einen Seed gesetzt.“ Für einen Zufallsgenerator stimmt das. Für einen **parallelen** Solver nicht — dort ist die zweite Zufallsquelle die Ausführungsreihenfolge, und die lässt sich nicht mit einem Parameter festnageln. Reproduzierbar wird der Lauf erst mit `num_workers = 1` **und** Seed. Wer nur eines von beidem setzt, hat einen Test, der gelegentlich grundlos rot wird.

Was daraus für Tests folgt

Der Zielwert war in **allen** sechzehn Läufen derselbe: 183. Die Gestalt der Lösung war es nicht. Genau daran entscheidet sich, was ein Test prüfen darf:

Prüfen	Nicht prüfen
den **Zielwert** — er ist eindeutig, wenn das Modell optimal gelöst wurde	die **Gestalt** des Plans (`assert plan == erwarteter_plan`)
die **Einhaltung aller Regeln** über eine Abnahmeprüfung (Abschnitt 22.6)	die Reihenfolge, in der gleichwertige Aufträge zugewiesen wurden
den **Status** (`OPTIMAL`, `FEASIBLE`, `INFEASIBLE`)	die Laufzeit als feste Zahl

Wer doch einen bestimmten Plan braucht — für eine abgedruckte Ausgabe, für einen Regressionstest —, setzt `num_workers = 1`. Das kostet hier den Faktor 12 an Laufzeit, und genau deshalb ist es eine Entscheidung **für den Test**, nicht für den Betrieb. In Produktion lässt man beide Parameter auf ihren Standardwerten.

> **Merksatz** Ein paralleler Solver hat zwei Zufallsquellen: den Seed und die Uhr. Nur die erste lässt sich setzen. Wer Reproduzierbarkeit braucht, muss die zweite abschalten — und das heißt `num_workers = 1`.

7.8 Die fünf Antworten von CP-SAT

Abschnitt 6.8 hat gezeigt, warum `OPTIMAL` bei ganzzahligen Problemen nicht der Normalfall ist. Bei CP-SAT kommt eine Besonderheit hinzu: Es gibt **fünf** verschiedene Antworten, und drei davon werden regelmäßig verwechselt.

Status	Was er über *was* aussagt	Richtige Reaktion
`OPTIMAL`	über die **Lösung** — sie ist beweisbar bestmöglich	ausführen
`FEASIBLE`	über die **Zeit** — ein gültiger Plan liegt vor, der Beweis fehlt	Gap prüfen, dann entscheiden
`INFEASIBLE`	über Ihr **Modell** — es gibt keinen zulässigen Plan	harte Regeln lockern oder weich machen
`MODEL_INVALID`	über Ihren **Code** — das Modell ist gar kein gültiges Modell	Programmierfehler beheben
`UNKNOWN`	über die **Zeit** — nichts gefunden, aber auch nichts widerlegt	mehr Zeit, einfacheres Modell oder `AddHint`

> **Merksatz** `INFEASIBLE` und `UNKNOWN` sehen im Log ähnlich aus und bedeuten das Gegenteil. `INFEASIBLE` heißt: *Es gibt keine Lösung* — auch unendlich viel Rechenzeit ändert daran nichts, die Ursache liegt in Ihren Regeln. `UNKNOWN` heißt: *Ich habe keine gefunden* — ob es

eine gibt, ist völlig offen. Wer beides in ein `else: return None` zusammenfasst, wirft genau die Information weg, die zur Fehlersuche nötig wäre.

Und `MODEL_INVALID` ist von beiden nochmals verschieden: Es ist immer ein **Programmierfehler**, kein fachliches Problem. Der klassische Fall ist eine Variable mit leerem Wertebereich — `NewIntVar(mindestbesetzung, kapazitaet, ...)`, wobei die Daten vertauscht ankamen und die untere Schranke über der oberen liegt.

Das folgende Programm erzeugt jeden Fall absichtlich und zeigt die passende Auswertung.

```
#!/usr/bin/env python3

# CP_SAT_Statusfaelle.py
"""
Kapitel CP-SAT: Alle fuenf Antworten von CP-SAT - und was jede bedeutet.

Der haeufigste Fehler beim Einsatz von CP-SAT im Betrieb ist eine Zeile wie

    if status == cp_model.OPTIMAL:
        ...verwerte die Loesung...

Sie wirft in drei von fuenf Faellen etwas weg, das man haette gebrauchen
koennen, und verschweigt in einem weiteren Fall einen Modellierungsfehler.

Dieses Programm erzeugt jeden Statusfall absichtlich und zeigt die passende
Reaktion:

    OPTIMAL        beweisbar bestmoeglich          -> ausfuehren
    FEASIBLE       zulaessig, Zeit war um          -> Gap pruefen, dann entscheiden
    INFEASIBLE     kein Plan existiert             -> harte Regeln lockern
    MODEL_INVALID  das Modell ist fehlerhaft       -> Programmierfehler beheben
    UNKNOWN        nichts gefunden, Zeit war um    -> mehr Zeit oder Heuristik

Benoetigt: numpy, ortools
"""

from __future__ import annotations

from dataclasses import dataclass

import numpy as np
from ortools.sat.python import cp_model


@dataclass
class Auswertung:
    """Was nach einem CP-SAT-Lauf tatsaechlich feststeht."""
    status: str
    hat_loesung: bool
```

```
    beweisbar_optimal: bool
    zielwert: float | None
    schranke: float | None
    gap: float | None
    empfehlung: str


def werte_aus(loeser: cp_model.CpSolver, status: int,
              mit_zielfunktion: bool) -> Auswertung:
    """Uebersetzt einen CP-SAT-Status in eine Handlungsempfehlung.

    Genau diese Funktion fehlt in den meisten Projekten. Sie ist kurz, aber
    sie ist der Unterschied zwischen einem Prototyp und einem System, das
    nachts ohne Aufsicht laeuft.
    """
    name = loeser.StatusName(status)
    hat_loesung = status in (cp_model.OPTIMAL, cp_model.FEASIBLE)

    zielwert = schranke = gap = None
    if hat_loesung and mit_zielfunktion:
        zielwert = loeser.ObjectiveValue()
        schranke = loeser.BestObjectiveBound()
        nenner = max(abs(zielwert), 1e-9)
        gap = abs(zielwert - schranke) / nenner

    if status == cp_model.OPTIMAL:
        empfehlung = "Ausfuehren. Besser geht es nachweislich nicht."
    elif status == cp_model.FEASIBLE:
        empfehlung = (f"Brauchbar. Der Plan ist hoechstens {gap * 100:.2f} % "
                      f"schlechter als das theoretisch Bestmoegliche."
                      if gap is not None else
                      "Brauchbar, aber nicht als optimal bewiesen.")
    elif status == cp_model.INFEASIBLE:
        empfehlung = ("KEIN Plan existiert. Das ist eine Aussage ueber Ihre "
                      "harten Regeln, nicht ueber die Rechenzeit: Auch "
                      "unendlich viel Zeit wuerde nichts aendern. Regeln "
                      "lockern oder weich machen (Anhang C).")
    elif status == cp_model.MODEL_INVALID:
        empfehlung = ("Das MODELL ist fehlerhaft, nicht das Problem. "
                      "Typisch: eine Variable mit leerem Wertebereich. "
                      "model.Validate() nennt die Ursache.")
    else:                                   # UNKNOWN
        empfehlung = ("Nichts gefunden - aber auch nicht widerlegt. Mehr Zeit "
                      "geben, das Modell vereinfachen oder mit einer "
                      "Heuristik vorbelegen (AddHint).")

    return Auswertung(
        status=name,
```

```
        hat_loesung=hat_loesung,
        beweisbar_optimal=status == cp_model.OPTIMAL,
        zielwert=zielwert, schranke=schranke, gap=gap,
        empfehlung=empfehlung,
    )


def loese(baue, zeitlimit: float, mit_zielfunktion: bool) -> Auswertung:
    modell = cp_model.CpModel()
    baue(modell)
    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = zeitlimit
    # Fuer reproduzierbare Buchausgaben; im Betrieb weglassen.
    loeser.parameters.num_workers = 1
    loeser.parameters.random_seed = 1
    status = loeser.Solve(modell)
    return werte_aus(loeser, status, mit_zielfunktion)


# --- Die fuenf Faelle -------------------------------------------------------

def dienstplan_loesbar(modell: cp_model.CpModel) -> None:
    """Vier Personen, fuenf Dienste, hoechstens zwei je Person - loesbar."""
    personen, dienste = 4, 5
    x = [[modell.NewBoolVar(f"x_{i}_{j}") for j in range(dienste)]
         for i in range(personen)]
    for j in range(dienste):
        modell.AddExactlyOne(x[i][j] for i in range(personen))
    for i in range(personen):
        modell.Add(sum(x[i]) <= 2)


def dienstplan_unloesbar(modell: cp_model.CpModel) -> None:
    """Dieselben fuenf Dienste, aber hoechstens EIN Dienst je Person.

    Vier Personen koennen zusammen hoechstens vier Dienste uebernehmen - fuer
    fuenf Dienste reicht das nicht. Kein Solver der Welt findet hier etwas.
    """
    personen, dienste = 4, 5
    x = [[modell.NewBoolVar(f"x_{i}_{j}") for j in range(dienste)]
         for i in range(personen)]
    for j in range(dienste):
        modell.AddExactlyOne(x[i][j] for i in range(personen))
    for i in range(personen):
        modell.Add(sum(x[i]) <= 1)


def modell_fehlerhaft(modell: cp_model.CpModel) -> None:
```

```
    """Eine Variable mit leerem Wertebereich: untere Schranke > obere.

    Im Alltag entsteht das durch einen Rechenfehler in den Grenzen, etwa
    NewIntVar(mindestbesetzung, kapazitaet, ...) bei falsch sortierten Daten.
    """
    kaputt = modell.NewIntVar(5, 2, "leerer_bereich")
    modell.Add(kaputt >= 0)


def lastverteilung(modell: cp_model.CpModel) -> None:
    """120 Auftraege auf 11 Maschinen - gross genug, dass der Optimalitaets-
    beweis laenger dauert als das Finden einer sehr guten Loesung."""
    rng = np.random.default_rng(3)
    dauer = rng.integers(100, 9000, 120)
    n, maschinen = len(dauer), 11
    obergrenze = int(dauer.sum())

    x = [[modell.NewBoolVar(f"x_{i}_{k}") for k in range(maschinen)]
         for i in range(n)]
    for i in range(n):
        modell.AddExactlyOne(x[i])
    belegung = [modell.NewIntVar(0, obergrenze, f"l_{k}") for k in range(maschinen)]
    for k in range(maschinen):
        modell.Add(belegung[k] == sum(int(dauer[i]) * x[i][k] for i in range(n)))
    makespan = modell.NewIntVar(0, obergrenze, "makespan")
    modell.AddMaxEquality(makespan, belegung)
    modell.Minimize(makespan)


def zahlpartition(modell: cp_model.CpModel) -> None:
    """46 grosse Zahlen exakt in zwei gleich schwere Haelften teilen.

    Ein beruehmt schweres Problem: Es gibt keine Zielfunktion, entweder es
    geht auf oder nicht - und beides zu entscheiden dauert lange.
    """
    rng = np.random.default_rng(1)
    gewichte = rng.integers(10**6, 2 * 10**6, 46)
    x = [modell.NewBoolVar(f"x_{i}") for i in range(len(gewichte))]
    modell.Add(sum(int(gewichte[i]) * x[i] for i in range(len(gewichte)))
               == int(gewichte.sum() // 2))


def zeige(titel: str, a: Auswertung) -> None:
    print(f"\n{titel}")
    print(f"  Status               {a.status}")
    print(f"  Loesung vorhanden    {'ja' if a.hat_loesung else 'nein'}")
    if a.zielwert is not None:
        print(f"  Zielwert / Schranke  {a.zielwert:,.0f} / {a.schranke:,.0f}"
```

```
              f"   (Gap {a.gap * 100:.3f} %)")
    print(f"  -> {a.empfehlung}")


if __name__ == "__main__":
    print("=" * 78)
    print("  DIE FUENF ANTWORTEN VON CP-SAT")
    print("=" * 78)

    zeige("[1] Dienstplan, hoechstens 2 Dienste je Person",
          loese(dienstplan_loesbar, 10.0, mit_zielfunktion=False))

    zeige("[2] Lastverteilung 120 Auftraege / 11 Maschinen, 0,5 s Limit",
          loese(lastverteilung, 0.5, mit_zielfunktion=True))

    zeige("[3] Derselbe Fall mit 10 s Limit",
          loese(lastverteilung, 10.0, mit_zielfunktion=True))

    zeige("[4] Dienstplan, hoechstens 1 Dienst je Person",
          loese(dienstplan_unloesbar, 10.0, mit_zielfunktion=False))

    zeige("[5] Variable mit leerem Wertebereich",
          loese(modell_fehlerhaft, 10.0, mit_zielfunktion=False))

    zeige("[6] Zahlpartition mit 46 grossen Zahlen, 2 s Limit",
          loese(zahlpartition, 2.0, mit_zielfunktion=False))

    print("\n" + "=" * 78)
    print("  DIE DREI, DIE MAN NICHT VERWECHSELN DARF")
    print("=" * 78)
    print("INFEASIBLE  Es gibt keine Loesung. Eine Aussage ueber Ihr MODELL.")
    print("            Mehr Rechenzeit aendert daran nichts.")
    print("UNKNOWN     Es wurde keine gefunden. Eine Aussage ueber die ZEIT.")
    print("            Ob es eine gibt, ist offen.")
    print("MODEL_INVALID  Das Modell ist gar kein gueltiges Modell. Eine")
    print("            Aussage ueber Ihren CODE - immer ein Programmierfehler.")
    print()
    print("Wer diese drei in einem 'else: return None' zusammenfasst, verliert")
    print("genau die Information, die zur Fehlersuche noetig waere.")
    print("=" * 78)
```

Erwartete Ausgabe (Zielwerte in Lauf [2] hardwareabhängig):

```
==============================================================================
  DIE FUENF ANTWORTEN VON CP-SAT
==============================================================================

[1] Dienstplan, hoechstens 2 Dienste je Person
```

```
  Status               OPTIMAL
  Loesung vorhanden    ja
  -> Ausfuehren. Besser geht es nachweislich nicht.

[2] Lastverteilung 120 Auftraege / 11 Maschinen, 0,5 s Limit
  Status               FEASIBLE
  Loesung vorhanden    ja
  Zielwert / Schranke  49,744 / 49,277   (Gap 0.939 %)
  -> Brauchbar. Der Plan ist hoechstens 0.94 % schlechter als das theoretisch Bestmoegliche.

[3] Derselbe Fall mit 10 s Limit
  Status               FEASIBLE
  Loesung vorhanden    ja
  Zielwert / Schranke  49,345 / 49,277   (Gap 0.138 %)
  -> Brauchbar. Der Plan ist hoechstens 0.14 % schlechter als das theoretisch Bestmoegliche.

[4] Dienstplan, hoechstens 1 Dienst je Person
  Status               INFEASIBLE
  Loesung vorhanden    nein
  -> KEIN Plan existiert. Das ist eine Aussage ueber Ihre harten Regeln, nicht ueber die
   ↪  Rechenzeit: Auch unendlich viel Zeit wuerde nichts aendern. Regeln lockern oder weich
   ↪  machen (Anhang C).

[5] Variable mit leerem Wertebereich
  Status               MODEL_INVALID
  Loesung vorhanden    nein
  -> Das MODELL ist fehlerhaft, nicht das Problem. Typisch: eine Variable mit leerem
   ↪  Wertebereich. model.Validate() nennt die Ursache.

[6] Zahlpartition mit 46 grossen Zahlen, 2 s Limit
  Status               UNKNOWN
  Loesung vorhanden    nein
  -> Nichts gefunden - aber auch nicht widerlegt. Mehr Zeit geben, das Modell vereinfachen
   ↪  oder mit einer Heuristik vorbelegen (AddHint).

============================================================================
  DIE DREI, DIE MAN NICHT VERWECHSELN DARF
============================================================================
INFEASIBLE  Es gibt keine Loesung. Eine Aussage ueber Ihr MODELL.
            Mehr Rechenzeit aendert daran nichts.
UNKNOWN     Es wurde keine gefunden. Eine Aussage ueber die ZEIT.
            Ob es eine gibt, ist offen.
MODEL_INVALID  Das Modell ist gar kein gueltiges Modell. Eine
            Aussage ueber Ihren CODE - immer ein Programmierfehler.

Wer diese drei in einem 'else: return None' zusammenfasst, verliert
genau die Information, die zur Fehlersuche noetig waere.
============================================================================
```

Code-Durchgang

Stelle	Was passiert	Warum es zählt
`werte_aus()`	übersetzt Status in **Handlungsempfehlung**	Genau diese Funktion fehlt in den meisten Projekten. Sie ist kurz — und der Unterschied zwischen einem Prototyp und einem System, das nachts ohne Aufsicht läuft.
`hat_loesung = status in (OPTIMAL, FEASIBLE)`	die eigentliche Frage	Nicht „war der Solver erfolgreich?“, sondern „habe ich etwas in der Hand, das ich ausführen kann?“
`BestObjectiveBound()`	die bewiesene Schranke	Ohne sie ist `FEASIBLE` eine nutzlose Angabe. Mit ihr wird daraus „höchstens 1,27 % vom Optimum“.
Läufe [2] und [3]	0,5 s gegen 10 s	Der Gap fällt von 1,27 % auf 0,14 %. Zwanzigfache Rechenzeit für gut ein Prozent — die Entscheidung, ob sich das lohnt, trifft die Anwendung, nicht der Solver.
Lauf [4] gegen Lauf [6]	`INFEASIBLE` gegen `UNKNOWN`	Beide liefern keine Lösung. Bei [4] ist bewiesen, dass es keine gibt (vier Personen können keine fünf Dienste mit je höchstens einem übernehmen); bei [6] weiß niemand, ob eine existiert.

Typische Fehler

- **`if status == cp_model.OPTIMAL:` als einzige Prüfung.** Verwirft `FEASIBLE`-Lösungen, die im Betrieb völlig brauchbar wären — und macht aus einem `INFEASIBLE` stillschweigend ein „hat nicht geklappt“.
- **Bei `INFEASIBLE` das Zeitlimit erhöhen.** Ändert nichts. Die Ursache liegt in den harten Regeln; Anhang C zeigt, wie man die widersprüchliche Teilmenge findet.
- **`ObjectiveValue()` ohne Statusprüfung abfragen.** Bei `INFEASIBLE` oder `UNKNOWN` liefert der Aufruf einen bedeutungslosen Wert — und der wandert dann in einen Bericht.
- **`MODEL_INVALID` übersehen.** Der Status verrät einen Fehler im eigenen Code. `model.Validate()` nennt die genaue Stelle; nutzen Sie es beim Entwickeln immer.

7.9 Übungsaufgaben

Lösungen: Abschnitt A.7.

Aufgabe 7.1 □ — Propagation nachvollziehen. $x_1 \in \{1,\dots,6\}$, $x_2 \in \{1,\dots,6\}$, Bedingungen: $x_1 < x_2$ und $x_1 + x_2 = 8$. Reduzieren Sie beide Wertebereiche von Hand so weit wie möglich.

Aufgabe 7.2 □ — Hart oder weich? Klassifizieren Sie und begründen Sie, welche Strafhöhe Sie ansetzen würden: (a) Gesetzliche Mindestruhezeit. (b) „Herr Bauer möchte montags frei." (c) „In jeder Schicht mindestens eine examinierte Kraft." (d) „Möglichst keine geteilten Dienste." (e) „Kein Dienst länger als 10 Stunden."

Aufgabe 7.3 □□ — Regeln ergänzen. Erweitern Sie das Vertretungssystem um: (a) Frau Albrecht ist ab Slot 3 nicht mehr verfügbar. (b) Herr Bauer und Frau Koch sollen nicht beide am selben Tag eingesetzt werden. (c) Wer Slot 1 übernimmt, soll bevorzugt auch Slot 2 bekommen (Belohnung statt Strafe).

Aufgabe 7.4 □□ — Infeasibility erzeugen und diagnostizieren. Setzen Sie `MAX_VERTRETUNGEN = 0`. Was meldet das Programm? Bauen Sie anschließend eine **Schlupfvariable** je Slot ein („Stunde fällt aus", Strafe 10 000) und beobachten Sie, wie der Solver statt einer Fehlermeldung einen Notfallplan liefert. Welche Stunde lässt er ausfallen und warum?

Aufgabe 7.5 □□ — Sudoku. Lösen Sie ein 9×9-Sudoku mit CP-SAT. Sie brauchen nur `NewIntVar(1, 9, ...)` und `AddAllDifferent` für Zeilen, Spalten und die neun 3×3-Blöcke. Wie viele Zeilen Modellcode benötigen Sie?

Aufgabe 7.6 □□ — Die Grenze der Parallelität finden. `Parallele_Suche.py` misst 1, 2, 4 und 8 Arbeiter. (a) Ergänzen Sie 16 und 24 (bzw. die Kernzahl Ihres Rechners). Ab wo bringt ein weiterer Arbeiter nichts mehr — und wird es irgendwann wieder schlechter? (b) Setzen Sie `random_seed` bei jedem Lauf auf einen **anderen** Wert, aber `num_workers = 1`. Wie viele verschiedene Pläne kommen jetzt heraus? Was folgt daraus für die Frage, welcher der beiden Parameter die Reproduzierbarkeit sichert? (c) Vergrößern Sie die Instanz auf 15 Aufträge. Ab welcher Größe erreicht der Lauf mit einem Arbeiter das Zeitlimit, während acht noch `OPTIMAL` melden?

Aufgabe 7.7 □□□ — Job-Shop erweitern. Ergänzen Sie `JobShop_Intervalle.py` um: (a) Rüstzeiten von 1 Zeiteinheit zwischen zwei Aufträgen auf derselben Maschine. (b) Fälligkeitstermine je Auftrag mit Strafkosten für Verspätung. (c) Eine zweite Fräse (Kapazität 2 statt 1) mit `AddCumulative`.

Aufgabe 7.8 □□□ — Eigener Dienstplan. Modellieren Sie einen Wochendienstplan: 7 Tage, 3 Schichten (früh/spät/nacht), 8 Mitarbeitende. Regeln: Jede Schicht 2 Personen; nach Nachtschicht mindestens 2 Tage frei; höchstens 5 Arbeitstage in Folge; jede Person mindestens 2 freie Tage pro Woche; Wünsche als weiche Ziele. Geben Sie den Plan als Wochentabelle aus.

7.10 Finde den Denkfehler

Finde den Denkfehler: Die Betriebsvereinbarung, die niemanden interessiert

Ein Klinikteam lässt seinen Schichtplan optimieren. Zwei Wünsche kommen aus dem Betrieb:

- **Betriebsvereinbarung:** niemand mehr als zwei Dienste in Folge.
- **Wunschfrei:** die eingereichten freien Tage möglichst einhalten.

Beides wird als weiche Regel modelliert. Bei der Gewichtung überlegt der Entwickler: *„Wunschfrei ist für die Zufriedenheit zentral, das gewichte ich mit 10. Die Serienregel ist eine Formalie, die kaum je greift — 1 genügt.“*

Der Solver meldet `OPTIMAL`. Im ausgegebenen Plan stehen **zwölf** Drei-Tage-Serien. Die Personalvertretung lehnt den Plan ab.

Ihre Aufgabe: (a) Der Solver hat korrekt gerechnet und die Regel steht im Modell — warum wird sie trotzdem systematisch verletzt? (b) Was besagt das Zahlenpaar $(1, 10)$, wörtlich in Sätze übersetzt? (c) Ab welchem Gewicht verschwinden die Serien, und was kostet das an anderer Stelle? (d) Welche Frage hätte man dem Auftraggeber stellen müssen, **bevor** man eine Zahl einsetzt — und wann gehört eine Regel gar nicht erst in die Zielfunktion?

Auflösung: Abschnitt A.7.

Das folgende Programm rechnet den Plan für sechs verschiedene Gewichte durch. Es ist damit zugleich die Antwort auf (c) — und die Tabelle, die man in so einem Fall dem Auftraggeber vorlegt, statt ein Gewicht zu raten.

```
#!/usr/bin/env python3

# Strafgewichte.py
"""
Kapitel CP-SAT: Warum ein Strafgewicht keine Wichtigkeit ausdrueckt, sondern
einen Wechselkurs.

Weiche Regeln kommen als Strafterm in die Zielfunktion. Die Frage, die dabei
regelmaessig falsch beantwortet wird, lautet: "Wie gross muss die Strafe sein?"
Die uebliche Antwort - "je wichtiger, desto groesser" - klingt vernuenftig und
fuehrt zuverlaessig zu Plaenen, die niemand unterschreiben will.

Der Grund: Zwei Strafgewichte legen nicht Wichtigkeiten fest, sondern einen
UMRECHNUNGSKURS. Steht die Serienregel bei 1 und der Wunschfrei-Tag bei 10,
dann hat man dem Solver woertlich gesagt: "Zehn Verstoesse gegen die
Serienregel sind so schlimm wie ein abgelehnter Wunsch." Er nimmt das
ernst - und handelt danach.

Szenario: Schichtplan, 6 Personen, 14 Tage, 4 Besetzungen pro Tag.
  HART   jeder Tag genau 4 Personen; hoechstens 11 Dienste je Person
  WEICH  keine drei Dienste in Folge (Betriebsvereinbarung)
```

```
  WEICH  Wunschfrei-Tage einhalten (26 Antraege)

Benoetigt: numpy, ortools
"""

from __future__ import annotations

import numpy as np
from ortools.sat.python import cp_model

PERSONEN, TAGE, PRO_TAG = 6, 14, 4
MAX_DIENSTE = 11
STRAFE_WUNSCH = 10                    # bleibt fest - variiert wird die Serienstrafe

RNG = np.random.default_rng(2)
WUNSCHFREI = sorted({(p, t) for p in range(PERSONEN) for t in range(TAGE)
                     if RNG.random() < 0.35})


def plane(strafe_serie: int, zeitlimit: float = 25.0) -> dict:
    """Baut den Schichtplan mit dem angegebenen Gewicht fuer die Serienregel."""
    modell = cp_model.CpModel()
    x = {(p, t): modell.NewBoolVar(f"x_{p}_{t}")
         for p in range(PERSONEN) for t in range(TAGE)}

    # --- harte Regeln ------------------------------------------------
    for t in range(TAGE):
        modell.Add(sum(x[p, t] for p in range(PERSONEN)) == PRO_TAG)
    for p in range(PERSONEN):
        modell.Add(sum(x[p, t] for t in range(TAGE)) <= MAX_DIENSTE)

    # --- weiche Regel 1: keine drei Dienste in Folge -----------------
    # serie[p, t] = 1  <=>  Person p arbeitet an t, t+1 UND t+2.
    # Die Ungleichung erzwingt das nur in eine Richtung; das genuegt, weil
    # der Solver serie minimiert - er wuerde die Variable nie freiwillig
    # auf 1 setzen.
    strafterme = []
    serien = []
    for p in range(PERSONEN):
        for t in range(TAGE - 2):
            serie = modell.NewBoolVar(f"serie_{p}_{t}")
            modell.Add(x[p, t] + x[p, t + 1] + x[p, t + 2] <= 2 + serie)
            serien.append(serie)
            strafterme.append(serie * strafe_serie)

    # --- weiche Regel 2: Wunschfrei ----------------------------------
    wunschverstoesse = []
    for p, t in WUNSCHFREI:
```

```
        wunschverstoesse.append(x[p, t])
        strafterme.append(x[p, t] * STRAFE_WUNSCH)

    modell.Minimize(sum(strafterme))

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = zeitlimit
    loeser.parameters.num_workers = 1          # reproduzierbare Buchausgabe
    loeser.parameters.random_seed = 1
    status = loeser.Solve(modell)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError(f"Kein Plan: {loeser.StatusName(status)}")

    return {
        "status": loeser.StatusName(status),
        "zielwert": loeser.ObjectiveValue(),
        "serien": sum(loeser.Value(v) for v in serien),
        "wunsch_verletzt": sum(loeser.Value(v) for v in wunschverstoesse),
    }


if __name__ == "__main__":
    print("=" * 78)
    print("  STRAFGEWICHTE SIND WECHSELKURSE, KEINE WICHTIGKEITEN")
    print("=" * 78)
    print(f"{PERSONEN} Personen, {TAGE} Tage, {PRO_TAG} Besetzungen pro Tag "
          f"= {TAGE * PRO_TAG} Dienste")
    print(f"Das sind {TAGE * PRO_TAG / PERSONEN:.1f} Dienste je Person - der Plan ist eng.")
    print(f"{len(WUNSCHFREI)} Wunschfrei-Antraege, Strafe je abgelehntem Wunsch: "
          f"{STRAFE_WUNSCH}\n")

    print(f"{'Strafe je 3er-Serie':>20} {'3er-Serien':>12} {'Wunsch verletzt':>17} "
          f"{'Zielwert':>10}")
    print("-" * 78)
    ergebnisse = {}
    for strafe in (1, 2, 5, 10, 30, 100):
        e = plane(strafe)
        ergebnisse[strafe] = e
        print(f"{strafe:>20} {e['serien']:>12} {e['wunsch_verletzt']:>17} "
              f"{e['zielwert']:>10.0f}")
    print("-" * 78)

    niedrig, hoch = ergebnisse[1], ergebnisse[30]
    print(f"\nBei Strafe 1 baut der Solver {niedrig['serien']} Drei-Tage-Serien ein.")
    print("Die Betriebsvereinbarung steht im Modell - und wird trotzdem")
    print("systematisch verletzt. Das ist kein Fehler des Solvers: Bei einem")
    print(f"Gewicht von 1 gegen {STRAFE_WUNSCH} lohnt sich jede Serie, solange sie")
    print("auch nur einen Zehntel-Wunsch rettet.")
```

```
    print(f"\nBei Strafe 30 sind es {hoch['serien']} Serien - dafuer werden")
    print(f"{hoch['wunsch_verletzt']} statt {niedrig['wunsch_verletzt']} Wuensche
     ↪  abgelehnt.")
    print("Beide Plaene sind optimal. Sie beantworten nur verschiedene Fragen.")
    print()
    print("Die Frage lautet also nie 'wie wichtig ist mir diese Regel?', sondern:")
    print("  'Wie viele abgelehnte Wuensche bin ich bereit zu akzeptieren,")
    print("   um eine Drei-Tage-Serie zu vermeiden?'")
    print("Wer darauf keine Zahl nennen kann, hat das Problem noch nicht")
    print("verstanden - und sollte diese Tabelle dem Auftraggeber vorlegen,")
    print("statt ein Gewicht zu raten.")
    print()
    print("Und die wichtigste Konsequenz: Eine Regel, die NIE gebrochen werden")
    print("darf, gehoert nicht in die Zielfunktion, sondern unter die harten")
    print("Nebenbedingungen. Alles, was einen Preis hat, wird irgendwann gekauft.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  STRAFGEWICHTE SIND WECHSELKURSE, KEINE WICHTIGKEITEN
==============================================================================
6 Personen, 14 Tage, 4 Besetzungen pro Tag = 56 Dienste
Das sind 9.3 Dienste je Person - der Plan ist eng.
26 Wunschfrei-Antraege, Strafe je abgelehntem Wunsch: 10

 Strafe je 3er-Serie   3er-Serien   Wunsch verletzt   Zielwert
------------------------------------------------------------------------------
                   1           12                 6         72
                   2           12                 6         84
                   5            9                 7        115
                  10            2                11        130
                  30            0                14        140
                 100            0                14        140
------------------------------------------------------------------------------

Bei Strafe 1 baut der Solver 12 Drei-Tage-Serien ein.
Die Betriebsvereinbarung steht im Modell - und wird trotzdem
systematisch verletzt. Das ist kein Fehler des Solvers: Bei einem
Gewicht von 1 gegen 10 lohnt sich jede Serie, solange sie
auch nur einen Zehntel-Wunsch rettet.

Bei Strafe 30 sind es 0 Serien - dafuer werden
14 statt 6 Wuensche abgelehnt.
Beide Plaene sind optimal. Sie beantworten nur verschiedene Fragen.

Die Frage lautet also nie 'wie wichtig ist mir diese Regel?', sondern:
  'Wie viele abgelehnte Wuensche bin ich bereit zu akzeptieren,
```

```
   um eine Drei-Tage-Serie zu vermeiden?'
Wer darauf keine Zahl nennen kann, hat das Problem noch nicht
verstanden - und sollte diese Tabelle dem Auftraggeber vorlegen,
statt ein Gewicht zu raten.

Und die wichtigste Konsequenz: Eine Regel, die NIE gebrochen werden
darf, gehoert nicht in die Zielfunktion, sondern unter die harten
Nebenbedingungen. Alles, was einen Preis hat, wird irgendwann gekauft.
============================================================================
```

 Merksatz Strafgewichte drücken keine **Wichtigkeit** aus, sondern einen **Wechselkurs**. Wer der Serienregel 1 und dem Wunschfrei 10 gibt, hat wörtlich gesagt: „Zehn Verstöße gegen die Betriebsvereinbarung sind so schlimm wie ein abgelehnter Wunsch.“ Der Solver nimmt das ernst und handelt danach — das ist seine Aufgabe.

 Typische Fehler bei weichen Regeln

- **Gewichte nach Bauchgefühl setzen.** Rechnen Sie stattdessen die Tabelle wie oben und lassen Sie den Auftraggeber die Zeile aussuchen. Das dauert eine Stunde und ersetzt drei Abstimmungsrunden.
- **Eine unverhandelbare Regel weich modellieren.** Alles, was einen Preis hat, wird irgendwann gekauft. Gesetzliche Ruhezeiten, Qualifikationsanforderungen und Betriebsvereinbarungen gehören unter die **harten** Nebenbedingungen — auch wenn das Modell dann `INFEASIBLE` melden kann. Dieses `INFEASIBLE` ist die ehrliche Antwort: Der Plan ist mit dem vorhandenen Personal nicht zulässig zu machen.
- **Gewichte in verschiedenen Einheiten mischen.** Euro neben „Unzufriedenheitspunkten“ neben Minuten ergibt eine Zielfunktion, die niemand interpretieren kann — und deren Konditionszahl obendrein leidet (Abschnitt 2.7).
- **Nur ein Gewicht ausprobieren.** Ein einziger Lauf sagt nichts darüber, ob Sie nahe an einem Kipppunkt stehen. Zwischen den Gewichten 5 und 10 ändert sich der Plan oben erheblich.

7.11 Micro-Quiz

 Micro-Quiz 7: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Ihr Dienstplanmodell meldet `INFEASIBLE`. Ein Kollege schlägt vor, das Zeitlimit von 10 auf 300 Sekunden zu erhöhen. Was halten Sie davon? (a) Sinnvoll — bei schweren Instanzen braucht CP-SAT einfach länger. (b) Wirkungslos: `INFEASIBLE` ist ein **Beweis**, dass keine zulässige Lösung existiert. Die Ursache liegt in den harten Regeln, nicht in der Rechenzeit. (c) Sinnvoll, aber nur zusammen mit mehr Arbeitern (`num_workers`).

2. Sie modellieren „auf einer Maschine läuft immer nur ein Auftrag“. Was ist der Vorteil von `AddNoOverlap` gegenüber der MILP-Formulierung mit Big-M? (a) `AddNoOverlap` findet grundsätzlich bessere Lösungen. (b) Eine Zeile ersetzt bei k Aufträgen $\binom{k}{2}$ Entweder-

Oder-Konstruktionen mit je zwei Ungleichungen und einer Binärvariablen — und propagiert stärker, weil der Propagator die Struktur als Ganzes betrachtet. (c) `AddNoOverlap` funktioniert auch mit kontinuierlichen Zeiten, Big-M nicht.

3. Zwei Läufe desselben Job-Shop-Programms liefern verschiedene Pläne, beide mit Makespan 11. Was folgt daraus für Ihre Tests? (a) Das Programm ist fehlerhaft; ein korrektes Modell hat genau eine Lösung. (b) Es gibt mehrere optimale Pläne. Tests dürfen deshalb nicht auf einen bestimmten Plan prüfen, sondern auf den Zielwert und die Einhaltung aller Regeln. (c) Der Solver arbeitet ungenau; man sollte die Toleranzen verschärfen.

7.12 Selbsttest

Antworten: Anhang A.

1. Was macht Constraint Propagation, und warum ist sie effizienter als Ausprobieren?
2. Welchen Vorteil hat `AddAllDifferent(x)` gegenüber $\binom{n}{2}$ Einzelbedingungen?
3. Was ist eine Intervallvariable, und welche Bedingung garantiert sie automatisch?
4. Warum sollten möglichst wenige Bedingungen hart formuliert werden?
5. Warum ist die Aufschlüsselung der Strafkosten in der Praxis wichtiger als der Zielwert
6. Sie setzen `random_seed = 1`, bekommen aber trotzdem bei jedem Lauf einen anderen Plan. Woran liegt das, und was hilft?
7. Warum kann ein CP-SAT-Lauf mit acht Arbeitern **mehr** als achtmal schneller sein als mit einem? selbst?

7.13 Zusammenfassung

- **CP denkt in Wertebereichen und Regeln**, nicht in Ungleichungen und Schranken.
- **Propagation** streicht unmögliche Werte, bevor gesucht wird; **Conflict Learning** verhindert, dieselbe Sackgasse zweimal zu betreten.
- **Globale Constraints** wie AddAllDifferent“, AddNoOverlap“ und AddCumulative“ ersetzen ganze Blöcke von Big-M-Ungleichungen und propagieren stärker.
- **Intervallvariablen** sind das Werkzeug für Zeitplanung; `AddNoOverlap` erledigt in einer Zeile, was in MILP dutzende Disjunktionen kostet.
- **Weiche Ziele mit Strafkosten** machen Modelle robust gegen `INFEASIBLE` — und die **Kostenzerlegung** macht sie erklärbar.
- **Strafgewichte sind Wechselkurse, keine Wichtigkeiten.** Rechnen Sie mehrere Gewichte durch und lassen Sie den Auftraggeber eine Zeile der Tabelle wählen. Und was nie gebrochen werden darf, gehört unter die harten Nebenbedingungen: Alles, was einen Preis hat, wird irgendwann gekauft.
- **Fünf Antworten, nicht zwei.** `INFEASIBLE` (es gibt keine Lösung), `UNKNOWN` (es wurde keine gefunden) und `MODEL_INVALID` (Ihr Code ist fehlerhaft) bedeuten Grundverschiedenes. Wer sie in ein `else` zusammenfasst, verliert die Information zur Fehlersuche.

- **Bei mehrdeutigen Optima nie auf einen bestimmten Plan testen** — nur auf den Zielwert und die Einhaltung aller Regeln.
- **Ein paralleler Solver hat zwei Zufallsquellen: den Seed und die Uhr.** Nur die erste lässt sich setzen. Reproduzierbar wird ein Lauf erst mit `num_workers = 1` **und** Seed — gemessen in Abschnitt 7.7: schon zwei Arbeiter liefern bei identischem Seed drei verschiedene Pläne zum selben Zielwert.
- **Parallele Suche kann überlinear beschleunigen**, weil CP-SAT nicht dieselbe Suche vervielfacht, sondern verschiedene Strategien nebeneinander laufen lässt, die einander ihre Schranken mitteilen.

Ausblick. Kapitel 8 behandelt Probleme, die sich am besten als **Graph** begreifen lassen: Flüsse durch Netzwerke, Zuordnungen und die Tourenplanung mit Fahrzeugen und Zeitfenstern.

Kapitel 8: Graphen, Flüsse und Touren — Min-Cost-Flow, Matching und VRP

Kapitel auf einen Blick

Worum geht es? Um Probleme, deren natürliche Sprache der **Graph** ist: Was fließt wohin? Wer wird wem zugeordnet? Welche Route fährt welches Fahrzeug?

Voraussetzungen: Kapitel 5 und Kapitel 6.

Danach können Sie: Flussprobleme modellieren, Zuordnungsprobleme effizient lösen, eine Tourenplanung mit Kapazitäten und Zeitfenstern aufsetzen — und erkennen, wann eine begrenzende Dimension im Routing-Modell fehlt.

Zeitbedarf: ca. 6 Stunden.

Programme:

`Min_Cost_Flow.py`
`Zuordnung_Ungarisch.py`
`VRP_Flotten_Routing.py`
`VRP_Kapazitaetsfalle.py`

Notebook: `Notebooks_04/graphen.ipynb`

8.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Vier Monteure, vier Einsätze

Ein Kundendienst muss vier Monteure auf vier Einsatzorte verteilen. Die Tabelle enthält die Anfahrtszeit in Minuten. Jeder Monteur fährt genau einen Einsatz.

	Nord	Ost	Süd	West
Bauer	21	45	33	60
Cakir	18	52	29	47
Diaz	40	24	55	38
Engel	35	31	26	44

```
import numpy as np
from scipy.optimize import linear_sum_assignment

kosten = np.array([[21, 45, 33, 60],      # Bauer
                   [18, 52, 29, 47],      # Cakir
                   [40, 24, 55, 38],      # Diaz
                   [35, 31, 26, 44]])     # Engel
monteur = ["Bauer", "Cakir", "Diaz", "Engel"]
ort = ["Nord", "Ost", "Sued", "West"]

zeile, spalte = linear_sum_assignment(kosten)
for z, s in zip(zeile, spalte):
    print(f"{monteur[z]:6} -> {ort[s]:5} ({kosten[z, s]} min)")
print("Gesamtfahrzeit:", kosten[zeile, spalte].sum(), "min")
```

Ausgabe:

```
Bauer  -> Nord  (21 min)
Cakir  -> Sued  (29 min)
Diaz   -> Ost   (24 min)
Engel  -> West  (44 min)
Gesamtfahrzeit: 118 min
```

Eine Zeile Code, und das Problem ist beweisbar optimal gelöst — `linear_sum_assignment` ist die Ungarische Methode, ein Spezialalgorithmus, der ohne jeden Solver auskommt.

Interessanter ist aber, was der Algorithmus **nicht** tut. Der kleinste Wert der ganzen Tabelle ist die 18 bei *Cakir → Nord*. Die naheliegende Vorgehensweise — „nimm immer das günstigste noch freie Paar“ — beginnt also genau dort:

Vorgehen	Zuordnung	Gesamtzeit
Gierig („immer das billigste freie Paar“)	Cakir→Nord, Diaz→Ost, Engel→Süd, Bauer→West	**128 min**

Vorgehen	Zuordnung	Gesamtzeit
Ungarische Methode	Bauer→Nord, Cakir→Süd, Diaz→Ost, Engel→West	**118 min**

Die optimale Lösung schickt **Bauer** nach Nord, obwohl er dort drei Minuten länger braucht als Cakir. Der Grund: Cakir wird im Süden gebraucht, wo er mit 29 Minuten der mit Abstand Schnellste ist. Wer die 18 zuerst greift, verbaut sich das — und zahlt am Ende 10 Minuten mehr.

> **Merksatz** Der beste erste Zug ist selten Teil der besten Gesamtlösung. Genau deshalb gibt es Operations Research: Optimierung heißt, Entscheidungen **gemeinsam** zu treffen statt nacheinander. Bei vier Monteuren kostet die gierige Regel 8 %; bei vierzig kostet sie regelmäßig ein Vielfaches.

Warum funktioniert das? Weil das Zuordnungsproblem eine besondere Struktur hat: Seine Nebenbedingungsmatrix ist **total unimodular**. Das bedeutet — wir kommen in Abschnitt 8.4 darauf zurück —, dass die LP-Relaxation von ganz allein ganzzahlige Lösungen liefert. Man braucht hier also weder Branch-and-Bound noch Binärvariablen. Dieselbe Eigenschaft macht auch Flussprobleme so angenehm lösbar, und damit beginnt das Kapitel.

8.2 Lernziele

Nach diesem Kapitel können Sie …

1. … ein Transportproblem als Graph mit Quellen, Senken und Kapazitäten modellieren.
2. … den **Flusserhaltungssatz** aufstellen und seine Bedeutung erklären.
3. … begründen, warum Zuordnungsprobleme **ohne** Ganzzahligkeitsbedingung ganzzahlig lösbar sind (totale Unimodularität).
4. … ein Vehicle Routing Problem mit Kapazitäten und Zeitfenstern mit OR-Tools lösen.
5. … einschätzen, wann ein spezialisierter Algorithmus einem allgemeinen MILP überlegen ist.
6. … begründen, warum eine gierige Zuordnung systematisch schlechter ist als eine gemeinsame Optimierung.
7. … einen Tourenplan gegen die Wirklichkeit prüfen — unabhängig von den Bausteinen, aus denen das Modell gebaut wurde.

8.3 Graphen als Modellsprache

Viele reale Probleme in Logistik, Kommunikation und Finanzströmen sind keine flachen Ungleichungssysteme, sondern **Graphen** $G = (V, E)$:

- V — die **Knoten** (*vertices*): Server, Depots, Kunden, Konten, Lager.
- E — die gerichteten **Kanten** (*edges*): Datenleitungen, Straßen, Überweisungswege.

Das Minimum-Cost-Flow-Problem (MCNFP)

Minimum-Cost Network Flow Problem — deutsch: *kostenminimales Flussproblem*. Es ist das mathematische Fundament für Transportketten, Liquiditätsrouting und Datenverteilung.

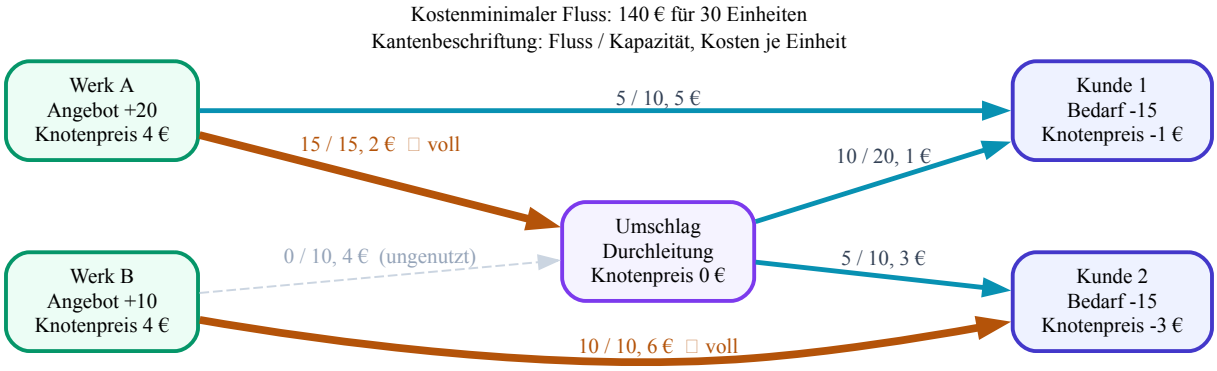

Abbildung 14: Abb. 8.1: Das Netzwerk aus `Min_Cost_Flow.py` samt Lösung: 140 € für 30 Einheiten. Die billige Route über den Umschlag läuft voll (dick, amber), die Kante Werk B → Umschlag bleibt ungenutzt (gestrichelt) — und trotzdem muss die teure Direktkante Werk B → Kunde 2 für 6 € bedient werden, weil der Umschlagweg dorthin schon ausgelastet ist. An den Knoten stehen die Dualwerte der Flusserhaltung. Erzeugt von `bilder_04/erzeuge_min_cost_flow.py`.

Sei $x_{ij} \geq 0$ der Fluss über Kante (i, j), c_{ij} die Kosten je Einheit und u_{ij} die Kapazität:

$$\min \sum_{(i,j)\in E} c_{ij}\, x_{ij}$$

$$\sum_{j:(i,j)\in E} x_{ij} \;-\; \sum_{k:(k,i)\in E} x_{ki} \;=\; b_i \quad \forall i \in V \qquad \textbf{(Flusserhaltung)}$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in E$$

□ **Formel-Lesehilfe zur Flusserhaltung** * Erste Summe: alles, was aus Knoten i **hinaus**fließt. * Zweite Summe: alles, was in Knoten i **hinein**fließt. * b_i — der Saldo des Knotens.

b_i	Knotentyp	Bedeutung
$b_i > 0$	**Quelle**	Hier entsteht Ware (Angebot)
$b_i < 0$	**Senke**	Hier verschwindet Ware (Bedarf)
$b_i = 0$	**Umladeknoten** (*transshipment*)	Was hineingeht, muss wieder hinaus

Ohne Formel gesagt: Das ist die Kirchhoffsche Knotenregel aus der Elektrotechnik — nichts geht verloren, nichts entsteht aus dem Nichts. Für einen Umladeknoten heißt das wörtlich: „Was ankommt, fährt auch wieder weg."

Wichtig: Damit das Problem lösbar ist, muss $\sum_i b_i = 0$ gelten — das Gesamtangebot muss dem Gesamtbedarf entsprechen.

```
#!/usr/bin/env python3

# Min_Cost_Flow.py
"""
Kapitel Graphen: Kostenminimaler Fluss durch ein Netzwerk.

Loest dasselbe Problem zweimal:
  (1) als allgemeines LP mit scipy  -> zeigt die Modellstruktur
  (2) mit dem spezialisierten Netzwerk-Solver von OR-Tools -> zeigt den
      Geschwindigkeitsvorteil eines Verfahrens, das die Struktur ausnutzt

Beide laufen in eigenen Prozessen nicht noetig: scipy und ortools vertragen
sich (nur ortools + highspy kollidieren, siehe Kapitel Oekosystem).
"""

import numpy as np
from scipy.optimize import linprog

# --- Netzwerk definieren ------------------------------------------------
KNOTEN = ["Werk_A", "Werk_B", "Umschlag", "Kunde_1", "Kunde_2"]
# (von, nach, Kosten je Einheit, Kapazitaet)
KANTEN = [
    ("Werk_A",   "Umschlag", 2.0, 15),
    ("Werk_A",   "Kunde_1",  5.0, 10),
    ("Werk_B",   "Umschlag", 4.0, 10),
    ("Werk_B",   "Kunde_2",  6.0, 10),
    ("Umschlag", "Kunde_1",  1.0, 20),
    ("Umschlag", "Kunde_2",  3.0, 10),
]
# Angebot (+) bzw. Bedarf (-) je Knoten
SALDO = {"Werk_A": 20, "Werk_B": 10, "Umschlag": 0, "Kunde_1": -15, "Kunde_2": -15}


def loese_als_lp():
    """Flussproblem als allgemeines lineares Programm."""
    n_kanten = len(KANTEN)
    knoten_index = {k: i for i, k in enumerate(KNOTEN)}

    # Zielfunktion: Summe der Transportkosten
    kosten = np.array([k[2] for k in KANTEN])

    # Flusserhaltung als Gleichungssystem: A_eq @ x = b_eq
```

```
    A_eq = np.zeros((len(KNOTEN), n_kanten))
    for e, (von, nach, _, _) in enumerate(KANTEN):
        A_eq[knoten_index[von], e] = +1.0      # fliesst hinaus
        A_eq[knoten_index[nach], e] = -1.0     # fliesst hinein
    b_eq = np.array([SALDO[k] for k in KNOTEN], dtype=float)

    schranken = [(0, k[3]) for k in KANTEN]    # 0 <= x_ij <= u_ij

    ergebnis = linprog(c=kosten, A_eq=A_eq, b_eq=b_eq, bounds=schranken, method="highs")
    if not ergebnis.success:
        raise SystemExit(f"Nicht loesbar: {ergebnis.message}")
    return ergebnis.fun, ergebnis.x, ergebnis.eqlin.marginals


if __name__ == "__main__":
    # Vorabpruefung: Angebot muss Bedarf entsprechen
    gesamt = sum(SALDO.values())
    print("=" * 78)
    print("  KOSTENMINIMALER FLUSS DURCH EIN TRANSPORTNETZ")
    print("=" * 78)
    print(f"Angebot gesamt: {sum(v for v in SALDO.values() if v > 0)} | "
          f"Bedarf gesamt: {-sum(v for v in SALDO.values() if v < 0)} | "
          f"Saldo: {gesamt}")
    if gesamt != 0:
        raise SystemExit("Angebot und Bedarf stimmen nicht ueberein - unloesbar!")

    kosten_gesamt, fluss, knotenpreise = loese_als_lp()

    print(f"\nMinimale Transportkosten: {kosten_gesamt:,.2f} EUR\n")
    print(f"{'Kante':<24} {'Fluss':>7} {'Kapazitaet':>11} {'Kosten/E':>9} {'Kosten':>9}")
    print("-" * 78)
    for e, (von, nach, c, u) in enumerate(KANTEN):
        menge = fluss[e] + 0.0 if abs(fluss[e]) > 1e-9 else 0.0   # vermeidet "-0.0"
        ausgelastet = " (VOLL)" if abs(menge - u) < 1e-6 else ""
        print(f"{von + ' -> ' + nach:<24} {menge:>7.1f} {u:>11} "
              f"{c:>9.2f} {menge * c:>9.2f}{ausgelastet}")

    # --- Flusserhaltung nachpruefen ------------------------------------
    print("\n--- Pruefung der Flusserhaltung je Knoten ---")
    for k in KNOTEN:
        hinaus = sum(fluss[e] for e, (v, n, _, _) in enumerate(KANTEN) if v == k)
        hinein = sum(fluss[e] for e, (v, n, _, _) in enumerate(KANTEN) if n == k)
        netto = hinaus - hinein
        art = "Quelle" if SALDO[k] > 0 else ("Senke" if SALDO[k] < 0 else "Umschlag")
        print(f"  {k:<10} ({art:<8}): hinaus {hinaus:5.1f} - hinein {hinein:5.1f} "
              f"= {netto:+6.1f}  (gefordert: {SALDO[k]:+d})")
        assert abs(netto - SALDO[k]) < 1e-6, f"Flusserhaltung verletzt bei {k}!"
```

```
    # --- Knotenpreise (Dualwerte) interpretieren ------------------------
    print("\n--- Knotenpreise (Dualwerte der Flusserhaltung) ---")
    print("  Differenz zweier Knotenpreise = Grenzkosten einer zusaetzlichen Einheit")
    print("  auf dem guenstigsten Weg zwischen ihnen.")
    for k, preis in zip(KNOTEN, knotenpreise):
        print(f"  {k:<10}: {preis:7.2f}")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  KOSTENMINIMALER FLUSS DURCH EIN TRANSPORTNETZ
==============================================================================
Angebot gesamt: 30 | Bedarf gesamt: 30 | Saldo: 0

Minimale Transportkosten: 140.00 EUR

Kante                      Fluss  Kapazitaet  Kosten/E    Kosten
------------------------------------------------------------------------------
Werk_A -> Umschlag          15.0          15      2.00     30.00 (VOLL)
Werk_A -> Kunde_1            5.0          10      5.00     25.00
Werk_B -> Umschlag           0.0          10      4.00      0.00
Werk_B -> Kunde_2           10.0          10      6.00     60.00 (VOLL)
Umschlag -> Kunde_1         10.0          20      1.00     10.00
Umschlag -> Kunde_2          5.0          10      3.00     15.00

--- Pruefung der Flusserhaltung je Knoten ---
  Werk_A     (Quelle  ): hinaus  20.0 - hinein   0.0 =  +20.0  (gefordert: +20)
  Werk_B     (Quelle  ): hinaus  10.0 - hinein   0.0 =  +10.0  (gefordert: +10)
  Umschlag   (Umschlag): hinaus  15.0 - hinein  15.0 =   +0.0  (gefordert: +0)
  Kunde_1    (Senke   ): hinaus   0.0 - hinein  15.0 =  -15.0  (gefordert: -15)
  Kunde_2    (Senke   ): hinaus   0.0 - hinein  15.0 =  -15.0  (gefordert: -15)
```

Zwei Beobachtungen:

Der Umschlagknoten hat Saldo 0 — exakt 15 Einheiten hinein, exakt 15 hinaus. Er produziert und verbraucht nichts, sondern verteilt nur um.

Werk B fährt nicht über den Umschlag, obwohl dieser Weg existiert: $B \to$ Umschlag $\to$ Kunde 2 kostet $4 + 3 = 7$ je Einheit, der direkte Weg nur 6. Werk A dagegen nutzt den Umschlag intensiv, weil $2 + 1 = 3$ nach Kunde 1 deutlich günstiger ist als der direkte Weg mit 5 — und die günstige Kante $A \to$ Umschlag ist deshalb bis zur Kapazitätsgrenze **voll ausgelastet**. Genau hier liegt der Wert der Optimierung: Sie gewichtet solche Alternativen für alle Kanten **gleichzeitig** ab, während man von Hand schon bei zehn Knoten den Überblick verliert.

8.4 Bipartites Matching: das Zuordnungsproblem

Wenn N Aufgaben auf N Ressourcen **eins zu eins** verteilt werden — Orders auf Broker, Schichten auf Mitarbeitende, Aufträge auf Maschinen — spricht man von **bipartitem Matching**.

$$\min \sum_{i=1}^{N}\sum_{j=1}^{N} c_{ij}\,x_{ij} \qquad \text{u. d. N.} \qquad \sum_{j} x_{ij} = 1\ \forall i, \qquad \sum_{i} x_{ij} = 1\ \forall j, \qquad x_{ij} \geq 0$$

Der Satz von Birkhoff und von Neumann

> **Satz.** Die Extrempunkte der Menge aller doppelt-stochastischen Matrizen (alle Zeilensummen $= 1$, alle Spaltensummen $= 1$, $x_{ij} \geq 0$) sind **genau** die Permutationsmatrizen (alle $x_{ij} \in \{0, 1\}$).

Warum das praktisch enorm wichtig ist: Nach dem Fundamentalsatz aus Kapitel 2 liegt das LP-Optimum in einer Ecke. Die Ecken sind hier laut Satz automatisch 0/1-wertig. Also gilt:

> **Merksatz** Beim Zuordnungsproblem müssen Sie die Ganzzahligkeit **nicht** fordern — ein gewöhnlicher LP-Solver liefert von selbst eine 0/1-Lösung. Sie sparen sich damit die NP-Schwere von Branch-and-Bound vollständig.

Der Grund dahinter heißt **totale Unimodularität**: Die Nebenbedingungsmatrix hat eine spezielle Struktur, bei der jede quadratische Teilmatrix die Determinante 0, $+1$ oder -1 hat. Dieselbe Eigenschaft besitzt übrigens auch die Flusserhaltungsmatrix aus Abschnitt 8.3 — deshalb sind Netzwerkflüsse ebenfalls „von selbst“ ganzzahlig.

> **Formel-Übersetzer: totale Unimodularität**

Mathematik	Alltagssprache
$\det(\mathbf{B}) \in \{0, +1, -1\}$ für **jede** quadratische Teilmatrix $\mathbf{B}$ von $\mathbf{A}$	„Die Matrix ist so gebaut, dass beim Lösen nie ein echter Bruch entstehen kann.“
$\mathbf{b}$ ganzzahlig $\Rightarrow$ alle Ecken von $\{\mathbf{x} : \mathbf{Ax} = \mathbf{b},\ \mathbf{x} \geq 0\}$ ganzzahlig	„Sind Kapazitäten und Bedarfe ganze Zahlen, sind es die Ecken automatisch auch.“
zusammen mit dem Fundamentalsatz (Kapitel 2)	„Das LP-Optimum liegt in einer Ecke — und die ist hier von selbst ganzzahlig.“

> **Die praktische Folge in einem Satz:** *Bei Zuordnungs- und Flussproblemen dürfen Sie die Ganzzahligkeit weglassen und trotzdem ganzzahlige Lösungen erwarten — Sie sparen sich die NP-Schwere von Branch-and-Bound vollständig.*

> **Und die Warnung dazu:** Diese Eigenschaft ist zerbrechlich. Eine einzige zusätzliche Nebenbedingung, die nicht in das Schema passt — „höchstens drei Fahrzeuge insgesamt“, eine Fixkostenkopplung, eine Mindestabnahmemenge — zerstört die totale Unimodularität. Dann liefert die Relaxation wieder Brüche, und Sie brauchen doch ein MILP. Prüfen Sie das, bevor Sie sich auf die Struktur verlassen.

```python
#!/usr/bin/env python3

# Zuordnung_Ungarisch.py
"""
Kapitel Graphen: Das Zuordnungsproblem, dreifach geloest.

  (1) Ungarischer Algorithmus (scipy.optimize.linear_sum_assignment) - O(n^3)
  (2) als LP OHNE Ganzzahligkeitsforderung -> liefert trotzdem 0/1 (Birkhoff)
  (3) als MILP MIT Ganzzahligkeitsforderung -> gleiches Ergebnis, mehr Aufwand

Zeigt damit die praktische Bedeutung der totalen Unimodularitaet.
"""

import time

import numpy as np
from scipy.optimize import linear_sum_assignment, linprog


def erzeuge_kosten(n, seed=11):
    rng = np.random.default_rng(seed)
    return rng.integers(10, 99, size=(n, n)).astype(float)


def loese_ungarisch(kosten):
    zeilen, spalten = linear_sum_assignment(kosten)
    return kosten[zeilen, spalten].sum(), spalten


def baue_lp(kosten):
    """Gemeinsame LP-Struktur fuer Variante 2 und 3."""
    n = len(kosten)
    c = kosten.flatten()                        # x_ij in Zeilenreihenfolge
    A_eq = np.zeros((2 * n, n * n))
    for i in range(n):                          # jede Person genau eine Aufgabe
        A_eq[i, i * n:(i + 1) * n] = 1.0
    for j in range(n):                          # jede Aufgabe genau einer Person
        A_eq[n + j, j::n] = 1.0
    b_eq = np.ones(2 * n)
    return c, A_eq, b_eq


def loese_lp(kosten, ganzzahlig):
    n = len(kosten)
    c, A_eq, b_eq = baue_lp(kosten)
    ergebnis = linprog(c=c, A_eq=A_eq, b_eq=b_eq, bounds=[(0, 1)] * (n * n),
                       integrality=np.ones(n * n) if ganzzahlig else None,
                       method="highs")
```

```
    x = ergebnis.x.reshape(n, n)
    return ergebnis.fun, x


if __name__ == "__main__":
    print("=" * 84)
    print("  ZUORDNUNGSPROBLEM: DREI WEGE ZUM SELBEN ERGEBNIS")
    print("=" * 84)

    # --- Kleines Beispiel zum Nachvollziehen ----------------------------
    kosten = np.array([[82., 83., 69., 92.],
                       [77., 37., 49., 92.],
                       [11., 69., 5., 86.],
                       [8., 9., 98., 23.]])
    namen = ["Anna", "Ben", "Carla", "David"]
    aufgaben = ["Auftrag W", "Auftrag X", "Auftrag Y", "Auftrag Z"]

    print("\nKostenmatrix (wer bearbeitet was zu welchen Kosten?):")
    print(f"{'':<8}" + "".join(f"{a:>12}" for a in aufgaben))
    for i, name in enumerate(namen):
        print(f"{name:<8}" + "".join(f"{kosten[i, j]:>12.0f}" for j in range(4)))

    wert, zuordnung = loese_ungarisch(kosten)
    print(f"\nOptimale Zuordnung (Gesamtkosten {wert:.0f}):")
    for i, j in enumerate(zuordnung):
        print(f"  {namen[i]:<8} -> {aufgaben[j]:<12} ({kosten[i, j]:.0f} EUR)")

    # --- Nachweis: LP ohne Ganzzahligkeit liefert trotzdem 0/1 ----------
    wert_lp, x_lp = loese_lp(kosten, ganzzahlig=False)
    ist_binaer = np.all((np.abs(x_lp) < 1e-9) | (np.abs(x_lp - 1) < 1e-9))
    print(f"\nLP OHNE Ganzzahligkeitsforderung: Kosten {wert_lp:.0f}, "
          f"Loesung ist {'0/1-wertig' if ist_binaer else 'GEBROCHEN'}")
    print("  -> Satz von Birkhoff/von Neumann bestaetigt: Die Ecken sind Permutationen.")

    # --- Laufzeitvergleich bei wachsender Groesse -----------------------
    print("\n" + "-" * 84)
    print(f"{'n':>4} | {'Ungarisch':>12} | {'LP (kontinuierlich)':>21} | "
          f"{'MILP (ganzzahlig)':>19} | {'gleich?':>8}")
    print("-" * 84)
    for n in [10, 25, 50, 100]:
        k = erzeuge_kosten(n)

        t0 = time.perf_counter(); w1, _ = loese_ungarisch(k); t1 = time.perf_counter() - t0
        t0 = time.perf_counter(); w2, _ = loese_lp(k, False);  t2 = time.perf_counter() - t0
        if n <= 50:
            t0 = time.perf_counter(); w3, _ = loese_lp(k, True); t3 = time.perf_counter() - t0
            t3_text, gleich = f"{t3*1000:>16.1f} ms", abs(w1 - w3) < 1e-6
        else:
```

```
            t3_text, gleich = f"{'uebersprungen':>19}", abs(w1 - w2) < 1e-6

        print(f"{n:>4} | {t1*1000:>9.1f} ms | {t2*1000:>18.1f} ms | {t3_text} | "
              f"{'ja' if gleich else 'NEIN':>8}")

print("-" * 84)
print("Fazit: Der spezialisierte Ungarische Algorithmus ist um Groessenordnungen")
print("schneller. Nutzen Sie fuer reine Zuordnungen NIE einen MILP-Solver.")
print("=" * 84)
```

Code-Durchgang: die Indexakrobatik

Die Variable x_{ij} wird zu einem flachen Vektor der Länge n^2 aufgerollt: Position von x_{ij} ist $i \cdot n + j$. * `A_eq[i, i*n:(i+1)*n] = 1` — Zeile i: alle Aufgaben **einer** Person (ein zusammenhängender Block). * `A_eq[n+j, j::n] = 1` — Zeile $n + j$: alle Personen **einer** Aufgabe (jedes n-te Element, deshalb die Schrittweite `::n`).

Diese Umrechnung zwischen Matrix- und Vektorindizes ist eine der häufigsten Fehlerquellen überhaupt. **Prüfen Sie sie immer an einem winzigen Beispiel**, bei dem Sie die Matrix von Hand hinschreiben können.

8.5 Das Vehicle Routing Problem mit Zeitfenstern

Das **Traveling Salesperson Problem (TSP)**, deutsch *Problem des Handlungsreisenden*, fragt nach der kürzesten Rundreise durch N Städte. Das **Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)** erweitert es auf eine **Flotte** mit Kapazitätsgrenzen und Kundenzeitfenstern $[e_i, l_i]$.

Kurzzyklen verhindern

Ein naives Modell erlaubt **Subtouren**: isolierte Kreise, die das Depot nie anfahren. Die klassische Gegenmaßnahme ist die **MTZ-Formulierung** nach Miller, Tucker und Zemlin. Man führt Rangvariablen u_i ein (die Position des Knotens in der Tour):

$$u_i - u_j + C \cdot x_{ij} \leq C - d_j \qquad \forall i \neq j$$

Formel-Lesehilfe Wird Kante (i, j) benutzt ($x_{ij} = 1$), erzwingt die Ungleichung $u_j \geq u_i + d_j$ — der Rang wächst also entlang jeder benutzten Kante streng an. In einem geschlossenen Kreis müsste der Rang aber wieder zum Ausgangswert zurückkehren, was unmöglich ist. **Kreise ohne Depot werden dadurch mathematisch ausgeschlossen.**

Wird die Kante nicht benutzt ($x_{ij} = 0$), reduziert sich die Ungleichung auf $u_i - u_j \leq C - d_j$, was durch hinreichend großes C immer erfüllt ist — das Big-M-Muster aus Kapitel 6.

In der Praxis: nicht selbst modellieren Die MTZ-Formulierung ist didaktisch wertvoll, aber für reale Instanzen zu schwach — die LP-Relaxation ist sehr locker, und Branch-and-Bound braucht sehr lange. Professionelle Solver verwenden stattdessen dynamisch erzeugte

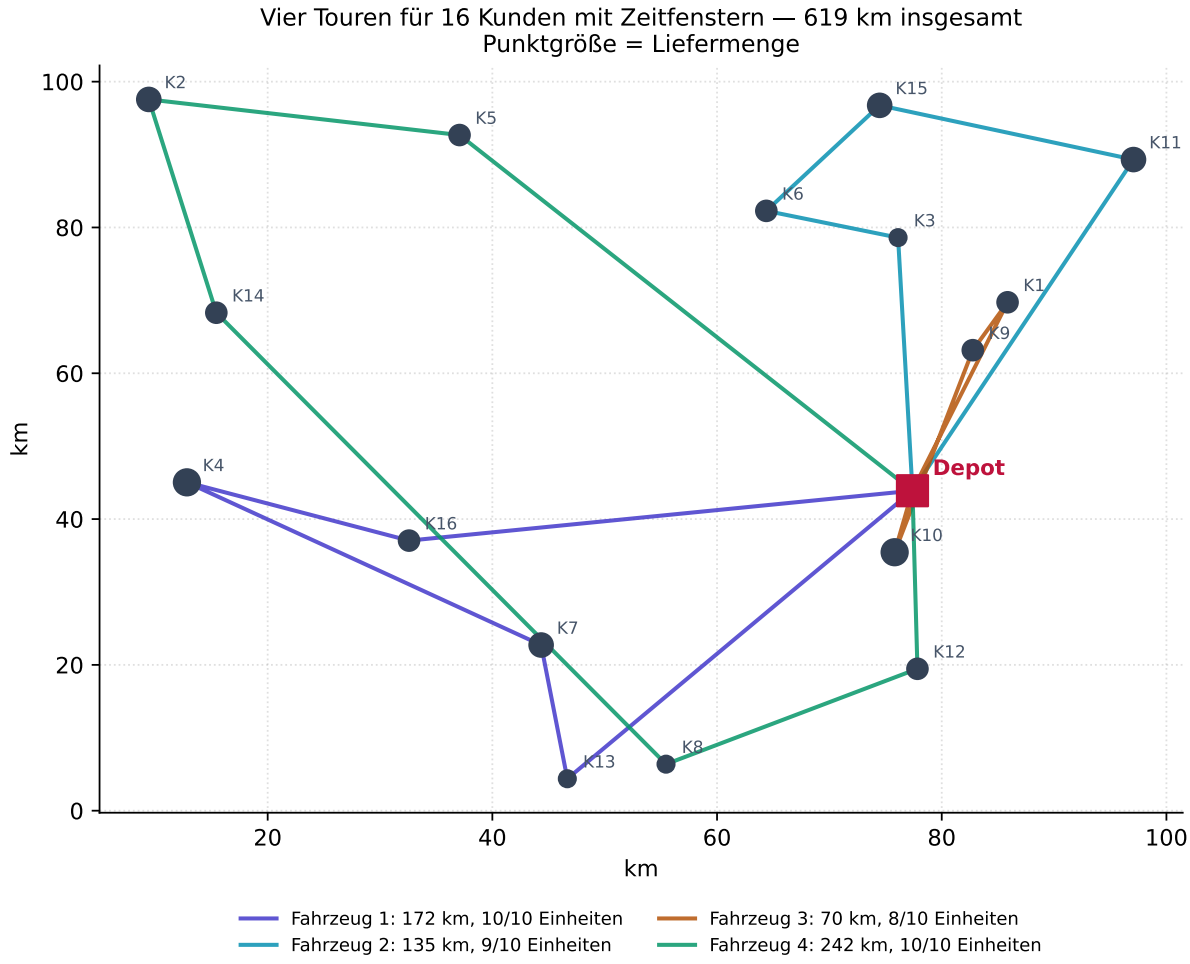

Abbildung 15: Abb. 8.2: Die Lösung der Instanz aus `VRP_Flotten_Routing.py`: 16 Kunden, vier Fahrzeuge, 619 km. **Beachten Sie, dass sich die Touren kreuzen.** Bei einem reinen Tourenproblem wäre das ein sicheres Zeichen für eine verbesserbare Lösung — hier ist es keines: Die Zeitfenster erzwingen die Reihenfolge, und wer die Kreuzungen auflöst, kommt zu spät. Erzeugt von `bilder_04/erzeuge_vrp_touren.py`.

Subtour-Eliminationsschnitte oder, wie OR-Tools, **spezialisierte Metaheuristiken**. Nutzen Sie für Routing die **Routing-Bibliothek**, nicht ein selbstgebautes MILP.

Praxisbeispiel: Flotten-Routing

```
#!/usr/bin/env python3

# VRP_Flotten_Routing.py
"""
Kapitel Graphen: Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)
mit der Routing-Bibliothek von Google OR-Tools.

Eigenschaften:
  * Eingabedaten werden vorab auf Plausibilitaet geprueft (Kapazitaet
    ausreichend? Zeitfenster erreichbar?)
  * Fahrzeit und Servicezeit werden getrennt ausgewiesen
  * Ausgabe als lesbarer Tourenplan mit Ankunftszeiten
  * Kennzahlen: Auslastung, Leerfahrten, Wartezeit
"""

import numpy as np
from ortools.constraint_solver import pywrapcp, routing_enums_pb2

SERVICEZEIT = 10          # Minuten je Kundenstopp
WARTEZEIT_MAX = 60        # zulaessige Wartezeit bei zu frueher Ankunft
SCHICHTLAENGE = 600       # Minuten


def erzeuge_daten(seed: int = 42):
    """Synthetische, aber reproduzierbare Instanz: 1 Depot + 16 Kunden."""
    anzahl_orte = 17
    rng = np.random.default_rng(seed)
    koordinaten = rng.random((anzahl_orte, 2)) * 100      # 100 x 100 km Raster

    distanz = np.zeros((anzahl_orte, anzahl_orte), dtype=int)
    for i in range(anzahl_orte):
        for j in range(anzahl_orte):
            distanz[i][j] = int(np.linalg.norm(koordinaten[i] - koordinaten[j]))

    return {
        "distanzmatrix": distanz.tolist(),
        "zeitfenster": [
            (0, SCHICHTLAENGE),                                  # 0: Depot
            (30, 120),  (60, 180),  (100, 240), (150, 300),      # Kunden 1-4
            (60, 180),  (120, 240), (200, 360), (300, 450),      # Kunden 5-8
            (180, 300), (240, 360), (300, 480), (360, 500),      # Kunden 9-12
            (60, 200),  (120, 300), (240, 400), (300, 550),      # Kunden 13-16
        ],
        "bedarfe": [0, 2, 3, 1, 4, 2, 2, 3, 1, 2, 4, 3, 2, 1, 2, 3, 2],
```

```
        "kapazitaeten": [10, 10, 10, 10],
        "anzahl_fahrzeuge": 4,
        "depot": 0,
    }


def pruefe_daten(daten) -> None:
    """Vorabdiagnose - fangt die haeufigsten Ursachen fuer 'keine Loesung' ab."""
    gesamtbedarf = sum(daten["bedarfe"])
    gesamtkapazitaet = sum(daten["kapazitaeten"])
    print(f"Gesamtbedarf {gesamtbedarf} Einheiten | "
          f"Flottenkapazitaet {gesamtkapazitaet} Einheiten | "
          f"Auslastung {gesamtbedarf / gesamtkapazitaet * 100:.0f} %")
    if gesamtbedarf > gesamtkapazitaet:
        raise SystemExit("UNLOESBAR: Der Bedarf uebersteigt die Flottenkapazitaet.")

    d = daten["distanzmatrix"]
    for kunde, (fruehestens, spaetestens) in enumerate(daten["zeitfenster"]):
        if kunde == 0:
            continue
        direktfahrt = d[0][kunde]
        if direktfahrt > spaetestens:
            raise SystemExit(
                f"UNLOESBAR: Kunde {kunde} ist erst nach {direktfahrt} min erreichbar, "
                f"sein Zeitfenster endet aber bei {spaetestens} min.")
    print("Vorabpruefung bestanden: Kapazitaet und Zeitfenster sind grundsaetzlich machbar.")


def loese_cvrptw(zeitlimit_s: int = 5):
    daten = erzeuge_daten()
    pruefe_daten(daten)

    manager = pywrapcp.RoutingIndexManager(
        len(daten["distanzmatrix"]), daten["anzahl_fahrzeuge"], daten["depot"])
    routing = pywrapcp.RoutingModel(manager)

    # --- Fahrzeit + Servicezeit als Kantengewicht ------------------------
    def zeit_callback(von_index, nach_index):
        von = manager.IndexToNode(von_index)
        nach = manager.IndexToNode(nach_index)
        service = SERVICEZEIT if von != daten["depot"] else 0
        return daten["distanzmatrix"][von][nach] + service

    zeit_index = routing.RegisterTransitCallback(zeit_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(zeit_index)

    # --- Kapazitaetsdimension --------------------------------------------
    def bedarf_callback(von_index):
```

```
        return daten["bedarfe"][manager.IndexToNode(von_index)]

    bedarf_index = routing.RegisterUnaryTransitCallback(bedarf_callback)
    routing.AddDimensionWithVehicleCapacity(
        bedarf_index, 0, daten["kapazitaeten"], True, "Kapazitaet")

    # --- Zeitdimension mit Zeitfenstern ---------------------------------
    routing.AddDimension(zeit_index, WARTEZEIT_MAX, SCHICHTLAENGE, False, "Zeit")
    zeit_dimension = routing.GetDimensionOrDie("Zeit")
    for ort, (fruehestens, spaetestens) in enumerate(daten["zeitfenster"]):
        zeit_dimension.CumulVar(manager.NodeToIndex(ort)).SetRange(fruehestens, spaetestens)

    # --- Suchparameter --------------------------------------------------
    parameter = pywrapcp.DefaultRoutingSearchParameters()
    parameter.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
    parameter.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
    parameter.time_limit.seconds = zeitlimit_s

    loesung = routing.SolveWithParameters(parameter)
    if not loesung:
        print("Keine zulaessige Routenfuehrung gefunden.")
        return

    # --- Auswertung -----------------------------------------------------
    print("\n" + "=" * 84)
    print("         OPTIMIERTER TOURENPLAN (CVRPTW)")
    print("=" * 84)

    gesamtzeit = gesamtfracht = gesamtdistanz = 0
    kapazitaet = daten["kapazitaeten"]

    for fahrzeug in range(daten["anzahl_fahrzeuge"]):
        index = routing.Start(fahrzeug)
        if routing.IsEnd(loesung.Value(routing.NextVar(index))):
            print(f"\nFahrzeug {fahrzeug + 1}: nicht eingesetzt")
            continue

        stationen, fracht, distanz = [], 0, 0
        while not routing.IsEnd(index):
            knoten = manager.IndexToNode(index)
            ankunft = loesung.Min(zeit_dimension.CumulVar(index))
            fracht += daten["bedarfe"][knoten]
            bezeichnung = "Depot" if knoten == 0 else f"K{knoten}"
            stationen.append(f"{bezeichnung}@{ankunft}")
            naechster = loesung.Value(routing.NextVar(index))
            distanz += daten["distanzmatrix"][knoten][manager.IndexToNode(naechster)]
```

```
            index = naechster

        endzeit = loesung.Min(zeit_dimension.CumulVar(index))
        stationen.append(f"Depot@{endzeit}")
        gesamtzeit += endzeit
        gesamtfracht += fracht
        gesamtdistanz += distanz

        print(f"\nFahrzeug {fahrzeug + 1}:")
        print("  " + " -> ".join(stationen))
        print(f"  Schichtzeit {endzeit} min | Fahrstrecke {distanz} km | "
              f"Fracht {fracht}/{kapazitaet[fahrzeug]} "
              f"({fracht / kapazitaet[fahrzeug] * 100:.0f} % Auslastung)")

    print("\n" + "-" * 84)
    print(f"Summe Schichtzeiten:   {gesamtzeit} min")
    print(f"Summe Fahrstrecken:    {gesamtdistanz} km")
    print(f"Transportierte Fracht: {gesamtfracht} von {sum(daten['bedarfe'])} Einheiten")
    assert gesamtfracht == sum(daten["bedarfe"]), "Nicht alle Kunden wurden beliefert!"
    print("Alle Kunden wurden innerhalb ihrer Zeitfenster beliefert.")
    print("=" * 84)


if __name__ == "__main__":
    loese_cvrptw()
```

□ Typische Fehler beim VRP

- **`Min(CumulVar)` mit „Ankunftszeit“ verwechseln.** Der Solver liefert ein *Intervall* möglicher Zeiten. `Min()` ist die früheste, `Max()` die späteste zulässige Zeit — die tatsächliche Fahrt kann irgendwo dazwischen starten.
- **Servicezeit im Depot mitzählen.** Beim Start am Depot fällt keine Servicezeit an, sonst verschiebt sich der ganze Plan.
- **Zu enge Zeitfenster ohne Vorabprüfung.** Ist ein Kunde in seinem Fenster physisch nicht erreichbar, meldet OR-Tools nur „keine Lösung“ — ohne zu sagen, welcher Kunde schuld ist. Die Funktion `pruefe_daten()` fängt genau das ab.
- **Ergebnisse als exakt betrachten.** Die Routing-Bibliothek nutzt **Metaheuristiken**. Ein längeres Zeitlimit kann eine bessere Lösung liefern; „optimal“ wird hier in der Regel nicht bewiesen. Für die Praxis genügt das fast immer — man sollte es aber wissen.

8.6 Übungsaufgaben

Lösungen: Abschnitt A.8.

Aufgabe 8.1 □ **— Flusserhaltung prüfen.** Ein Knoten hat Zuflüsse 12 und 8 sowie Abflüsse 15 und 3. Welchen Saldo b_i hat er, und um welchen Knotentyp handelt es sich?

Aufgabe 8.2 □ **— Unlösbarkeit erkennen.** Warum ist ein Flussproblem mit $\sum_i b_i \neq 0$ grundsätzlich unlösbar? Wie modelliert man den realistischen Fall „Angebot größer als Bedarf“?

Aufgabe 8.3 □□ **— Transportproblem lösen.** Drei Werke (Angebot 30, 25, 45) beliefern vier Lager (Bedarf 25, 30, 20, 25). Die Transportkosten je Einheit stehen in der Matrix

$$\begin{pmatrix} 8 & 6 & 10 & 9 \\ 9 & 12 & 13 & 7 \\ 14 & 9 & 16 & 5 \end{pmatrix}$$

(a) Stimmen Angebot und Bedarf überein? (b) Lösen Sie mit `linprog` und geben Sie den Transportplan aus. (c) Prüfen Sie, ob die Lösung ganzzahlig ist, obwohl Sie es nicht gefordert haben. Warum?

Aufgabe 8.4 □□ **— Zuordnung mit Verboten.** Erweitern Sie `Zuordnung_Ungarisch.py`: Carla darf Auftrag Y nicht bearbeiten (fehlende Zulassung). Wie modellieren Sie das? Wie ändert sich die Lösung?

Aufgabe 8.5 □□ **— Engpass finden.** Ergänzen Sie `Min_Cost_Flow.py` um eine Analyse: Welche Kante würde bei einer Kapazitätserhöhung um 1 Einheit die Gesamtkosten am stärksten senken? (Tipp: Dualwerte der Kapazitätsschranken oder schlicht neu rechnen.)

Aufgabe 8.6 □□□ **— VRP variieren.** Untersuchen Sie mit `VRP_Flotten_Routing.py`: (a) Wie ändert sich der Plan bei 3 statt 4 Fahrzeugen? Bei 2? (b) Ab welcher Fahrzeugzahl wird das Problem unlösbar — und warum? (c) Wie wirkt sich ein Zeitlimit von 1 s gegenüber 30 s auf die Lösungsqualität aus? (d) Verdoppeln Sie die Servicezeit. Was passiert?

Aufgabe 8.7 □□□ **— TSP mit MTZ selbst bauen.** Modellieren Sie ein TSP mit 8 Städten als MILP mit MTZ-Bedingungen (`linprog` mit `integrality`). Vergleichen Sie Laufzeit und Ergebnis mit der Routing-Bibliothek von OR-Tools. Was beobachten Sie ab 12 Städten?

8.7 Finde den Denkfehler

□ Finde den Denkfehler: Die vergessene Dimension

Eine Spedition lässt ihre Tagestouren optimieren: 16 Kunden, 4 Fahrzeuge zu je 10 Paletten, Gesamtbedarf 37 Paletten. Das erste Ergebnis begeistert alle — **326 km**. Nach einem Hinweis aus dem Fuhrpark wird das Modell überarbeitet; jetzt kommen **572 km** heraus, 75 % mehr. Der Auftraggeber ist verärgert: *„Ihre erste Version war doch viel besser.“*

Der Unterschied zwischen beiden Fassungen ist ein einziger Codeblock:

```
def bedarf(index):
    return BEDARFE[manager.IndexToNode(index)]
```

```
bedarf_id = routing.RegisterUnaryTransitCallback(bedarf)
routing.AddDimensionWithVehicleCapacity(
    bedarf_id, 0, KAPAZITAETEN, True, "Ladung")
```

Ihre Aufgabe: (a) Sehen Sie sich unten die Tourenübersicht des ersten Laufs an. Was tun die Fahrzeuge 1 bis 3, und wie viel lädt Fahrzeug 4? (b) Warum hat die Routing-Bibliothek das nicht von allein verhindert — die Kapazitäten standen doch in den Daten? (c) Warum ist ausgerechnet ein *besser* aussehendes Ergebnis hier das gefährliche? (d) Welche Prüfung hätte den Fehler sofort sichtbar gemacht — und warum darf sie nicht dieselben Bausteine benutzen wie das Modell?

Auflösung: Abschnitt A.8.

```
#!/usr/bin/env python3

# VRP_Kapazitaetsfalle.py
"""
Kapitel Graphen: Die vergessene Dimension.

Die Routing-Bibliothek von OR-Tools kennt keine "Kapazitaet" von sich aus.
Sie kennt nur DIMENSIONEN - benannte Groessen, die sich entlang einer Tour
aufsummieren und begrenzt werden koennen. Distanz ist eine, Zeit ist eine,
Ladung ist eine. Wer eine davon nicht anlegt, bekommt trotzdem eine Loesung:
eine schoene, kurze, guenstige - und unfahrbare.

Dieses Programm loest dieselbe Instanz zweimal und prueft beide Ergebnisse
gegen die tatsaechlichen Lademengen.

Instanz: 1 Depot, 16 Kunden, 4 Fahrzeuge zu je 10 Paletten.
Gesamtbedarf 37 Paletten bei 40 Paletten Flottenkapazitaet - es ist also
knapp, aber machbar.

Benoetigt: numpy, ortools
"""

from __future__ import annotations

import numpy as np
from ortools.constraint_solver import pywrapcp, routing_enums_pb2

# Dieselbe Instanz wie VRP_Flotten_Routing.py
BEDARFE = [0, 2, 3, 1, 4, 2, 2, 3, 1, 2, 4, 3, 2, 1, 2, 3, 2]
KAPAZITAETEN = [10, 10, 10, 10]
ANZAHL_FAHRZEUGE = 4
DEPOT = 0
```

```
def distanzmatrix(seed: int = 42) -> list[list[int]]:
    rng = np.random.default_rng(seed)
    koordinaten = rng.random((len(BEDARFE), 2)) * 100         # 100 x 100 km
    n = len(BEDARFE)
    return [[int(np.linalg.norm(koordinaten[i] - koordinaten[j]))
             for j in range(n)] for i in range(n)]


def plane(mit_kapazitaet: bool, zeitlimit: int = 5) -> dict:
    """Loest die Tourenplanung - wahlweise mit oder ohne Ladungsdimension."""
    distanz = distanzmatrix()
    manager = pywrapcp.RoutingIndexManager(len(distanz), ANZAHL_FAHRZEUGE, DEPOT)
    routing = pywrapcp.RoutingModel(manager)

    def entfernung(von_index, nach_index):
        return distanz[manager.IndexToNode(von_index)][manager.IndexToNode(nach_index)]

    kosten_id = routing.RegisterTransitCallback(entfernung)
    routing.SetArcCostEvaluatorOfAllVehicles(kosten_id)

    # DIE entscheidende Stelle. Ohne diesen Block existiert im Modell keine
    # Ladung - die Fahrzeuge sind dann unendlich gross.
    if mit_kapazitaet:
        def bedarf(index):
            return BEDARFE[manager.IndexToNode(index)]

        bedarf_id = routing.RegisterUnaryTransitCallback(bedarf)
        routing.AddDimensionWithVehicleCapacity(
            bedarf_id,
            0,                      # kein Zwischenpuffer
            KAPAZITAETEN,           # Obergrenze je Fahrzeug
            True,                   # Ladung startet bei 0
            "Ladung")

    parameter = pywrapcp.DefaultRoutingSearchParameters()
    parameter.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
    parameter.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
    parameter.time_limit.FromSeconds(zeitlimit)

    loesung = routing.SolveWithParameters(parameter)
    if loesung is None:
        raise RuntimeError("Keine Loesung gefunden")

    touren, strecken, ladungen = [], [], []
    for fahrzeug in range(ANZAHL_FAHRZEUGE):
        index = routing.Start(fahrzeug)
```

```
        tour, strecke, ladung = [], 0, 0
        while not routing.IsEnd(index):
            knoten = manager.IndexToNode(index)
            tour.append(knoten)
            ladung += BEDARFE[knoten]
            vorher = index
            index = loesung.Value(routing.NextVar(index))
            strecke += routing.GetArcCostForVehicle(vorher, index, fahrzeug)
        tour.append(manager.IndexToNode(index))
        touren.append(tour)
        strecken.append(strecke)
        ladungen.append(ladung)

    return {"touren": touren, "strecken": strecken, "ladungen": ladungen,
            "gesamtstrecke": sum(strecken)}


def pruefe(ergebnis: dict) -> list[str]:
    """Prueft den Plan gegen die Wirklichkeit - unabhaengig vom Modell.

    Genau diese Trennung ist der Punkt: Die Pruefung darf nicht dieselben
    Annahmen benutzen wie das Modell, sonst prueft sie nichts.
    """
    beanstandungen = []
    for fahrzeug, (ladung, kapazitaet) in enumerate(
            zip(ergebnis["ladungen"], KAPAZITAETEN)):
        if ladung > kapazitaet:
            beanstandungen.append(
                f"Fahrzeug {fahrzeug + 1}: {ladung} Paletten geladen, "
                f"Kapazitaet {kapazitaet} ({ladung - kapazitaet} zu viel)")

    beliefert = sorted(k for tour in ergebnis["touren"] for k in tour[1:-1])
    erwartet = list(range(1, len(BEDARFE)))
    if beliefert != erwartet:
        fehlend = set(erwartet) - set(beliefert)
        if fehlend:
            beanstandungen.append(f"nicht beliefert: {sorted(fehlend)}")
    return beanstandungen


def zeige(titel: str, ergebnis: dict) -> None:
    print(f"\n{titel}")
    print(f"  Gesamtstrecke {ergebnis['gesamtstrecke']} km")
    print(f"  {'Fahrzeug':<10} {'Stopps':>7} {'Strecke':>9} {'Ladung':>8} "
          f"{'Kapazitaet':>11}")
    for i, (tour, strecke, ladung) in enumerate(
            zip(ergebnis["touren"], ergebnis["strecken"], ergebnis["ladungen"])):
        markierung = "  <-- ueberladen" if ladung > KAPAZITAETEN[i] else ""
```

```
        print(f"  {i + 1:<10} {len(tour) - 2:>7} {strecke:>8} km {ladung:>8} "
              f"{KAPAZITAETEN[i]:>11}{markierung}")

    beanstandungen = pruefe(ergebnis)
    if beanstandungen:
        print("  PRUEFUNG: DURCHGEFALLEN")
        for text in beanstandungen:
            print(f"    - {text}")
    else:
        print("  PRUEFUNG: bestanden")


if __name__ == "__main__":
    print("=" * 78)
    print("  DIE VERGESSENE DIMENSION")
    print("=" * 78)
    print(f"16 Kunden, Gesamtbedarf {sum(BEDARFE)} Paletten, "
          f"{ANZAHL_FAHRZEUGE} Fahrzeuge zu je {KAPAZITAETEN[0]} "
          f"= {sum(KAPAZITAETEN)} Paletten Flottenkapazitaet.")

    ohne = plane(mit_kapazitaet=False)
    zeige("[1] Ohne Ladungsdimension", ohne)

    mit = plane(mit_kapazitaet=True)
    zeige("[2] Mit AddDimensionWithVehicleCapacity", mit)

    print("\n" + "=" * 78)
    mehr = mit["gesamtstrecke"] - ohne["gesamtstrecke"]
    print(f"Der korrekte Plan ist {mehr} km laenger "
          f"({mehr / ohne['gesamtstrecke'] * 100:.1f} %).")
    print()
    print("Und genau darin liegt die Gefahr: Lauf [1] sieht BESSER aus. Wer")
    print("beide Zahlen nebeneinander legt, ohne die Ladung zu pruefen, haelt")
    print("die unfahrbare Loesung fuer die bessere Optimierung - und den")
    print("korrekten Plan fuer schlechte Arbeit.")
    print()
    print("Die Routing-Bibliothek kennt keine 'Kapazitaet'. Sie kennt nur")
    print("Dimensionen, die man ihr anlegt. Was nicht als Dimension existiert,")
    print("wird nicht begrenzt - und faellt niemandem auf, weil das Ergebnis")
    print("plausibel aussieht.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  DIE VERGESSENE DIMENSION
==============================================================================
16 Kunden, Gesamtbedarf 37 Paletten, 4 Fahrzeuge zu je 10 = 40 Paletten Flottenkapazitaet.
```

```
[1] Ohne Ladungsdimension
  Gesamtstrecke 326 km
  Fahrzeug    Stopps   Strecke   Ladung  Kapazitaet
  1                0        0 km        0          10
  2                0        0 km        0          10
  3                0        0 km        0          10
  4               16      326 km       37          10  <-- ueberladen
  PRUEFUNG: DURCHGEFALLEN
    - Fahrzeug 4: 37 Paletten geladen, Kapazitaet 10 (27 zu viel)

[2] Mit AddDimensionWithVehicleCapacity
  Gesamtstrecke 572 km
  Fahrzeug    Stopps   Strecke   Ladung  Kapazitaet
  1                4      124 km       10          10
  2                3      142 km        9          10
  3                5      198 km       10          10
  4                4      108 km        8          10
  PRUEFUNG: bestanden

================================================================================
Der korrekte Plan ist 246 km laenger (75.5 %).

Und genau darin liegt die Gefahr: Lauf [1] sieht BESSER aus. Wer
beide Zahlen nebeneinander legt, ohne die Ladung zu pruefen, haelt
die unfahrbare Loesung fuer die bessere Optimierung - und den
korrekten Plan fuer schlechte Arbeit.

Die Routing-Bibliothek kennt keine 'Kapazitaet'. Sie kennt nur
Dimensionen, die man ihr anlegt. Was nicht als Dimension existiert,
wird nicht begrenzt - und faellt niemandem auf, weil das Ergebnis
plausibel aussieht.
================================================================================
```

 Merksatz Die Routing-Bibliothek kennt keine „Kapazität“, keine „Arbeitszeit“ und kein „Gewicht“. Sie kennt nur **Dimensionen** — benannte Größen, die sich entlang einer Tour aufsummieren und die man begrenzen kann. Was Sie nicht als Dimension anlegen, wird nicht begrenzt. Und der Solver sagt Ihnen das nicht: Er meldet stolz eine kürzere Strecke.

 Typische Fehler bei Routing-Modellen

- **Eine Dimension vergessen.** Ladung, Lenkzeit, Kühlkette, Gewicht *und* Volumen — jede Größe, die begrenzt ist, braucht ihre eigene Dimension. Zählen Sie sie vor dem Modellieren auf einem Blatt Papier auf.
- **Die Prüfung aus denselben Bausteinen bauen wie das Modell.** Wer die Ladung mit `loesung.Value(ladungs_dimension.CumulVar(...))` prüft, fragt das Modell, ob es sich an sich selbst hält. Rechnen Sie stattdessen aus der **ausgegebenen Tour** neu nach.

- **Zwei Läufe nur an der Zielfunktion vergleichen.** 326 gegen 572 km sagt nichts, solange nicht feststeht, dass beide Pläne überhaupt fahrbar sind.
- **Unbenutzte Fahrzeuge übersehen.** Drei Fahrzeuge, die im Depot stehen, während eines alles fährt, sind fast immer ein Zeichen für eine fehlende Beschränkung.

8.8 Micro-Quiz

Micro-Quiz 8: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Sie lösen ein Zuordnungsproblem (12 Monteure, 12 Aufträge) als LP — ganz ohne Binärvariablen. Das Ergebnis ist trotzdem 0/1-wertig. Warum? (a) Zufall; bei anderen Daten kämen Brüche heraus. (b) Die Nebenbedingungsmatrix ist total unimodular, deshalb sind alle Ecken des zulässigen Bereichs ganzzahlig — und in einer Ecke liegt das LP-Optimum. (c) `linprog` rundet die Lösung intern.

2. Sie ergänzen dasselbe Zuordnungsmodell um die Regel „höchstens 4 Monteure dürfen Überstunden machen“. Was ändert sich? (a) Nichts — die Struktur bleibt total unimodular. (b) Die Kardinalitätsbedingung passt nicht ins Schema; die totale Unimodularität geht verloren, die Relaxation kann Brüche liefern und Sie brauchen ein MILP. (c) Das Problem wird unlösbar.

3. Ein Tourenplan nutzt nur eines von vier verfügbaren Fahrzeugen und ist trotzdem der kürzeste gefundene. Was prüfen Sie zuerst? (a) Ob das Zeitlimit zu knapp war. (b) Ob eine begrenzende Dimension (Ladung, Lenkzeit) im Modell fehlt — ein einzelnes Fahrzeug, das alles fährt, ist das typische Bild einer vergessenen Beschränkung. (c) Ob die Distanzmatrix symmetrisch ist.

8.9 Selbsttest

Antworten: Anhang A.

1. Was besagt der Flusserhaltungssatz, und welchem physikalischen Gesetz entspricht er?
2. Warum liefert ein LP-Solver beim Zuordnungsproblem automatisch 0/1-Lösungen?
3. Was sind Subtouren, und wie verhindert die MTZ-Formulierung sie?
4. Warum sollte man ein reales VRP nicht als selbstgebautes MILP lösen?
5. Ein VRP meldet „keine Lösung“. Nennen Sie drei mögliche Ursachen und je eine Prüfung.

8.10 Zusammenfassung

- **Graphen** sind die natürliche Sprache für Transport-, Zuordnungs- und Routenprobleme.
- **Flusserhaltung** ist die Kirchhoff-Regel des Operations Research: Was hineingeht, kommt heraus — abzüglich des Knotensaldos.
- **Totale Unimodularität** macht Fluss- und Zuordnungsprobleme „von selbst“ ganzzahlig. Wer hier `integrality` setzt, verschenkt Laufzeit ohne Gegenwert.
- **Spezialisierte Algorithmen schlagen allgemeine Solver** deutlich: Der Ungarische Algorithmus löst in $O(n^3)$, wofür ein MILP-Solver Branch-and-Bound bräuchte.
- **Für Routing gilt: Nutzen Sie die Routing-Bibliothek.** Metaheuristiken liefern in Sekunden sehr gute Touren; exakte Optimalität ist bei realistischen Größen unrealistisch und praktisch entbehrlich.
- **Die Routing-Bibliothek kennt nur Dimensionen.** Ladung, Lenkzeit, Kühlkette, Gewicht — jede begrenzte Größe braucht ihre eigene. Was nicht als Dimension angelegt ist, wird nicht begrenzt, und der Solver meldet stolz eine kürzere Strecke.
- **Totale Unimodularität ist zerbrechlich.** Eine einzige zusätzliche Bedingung, die nicht ins Schema passt — Kardinalität, Fixkosten, Mindestmenge —, zerstört sie. Prüfen Sie das, bevor Sie sich auf die Struktur verlassen.
- **Der beste erste Zug ist selten Teil der besten Gesamtlösung.** Gierige Regeln kosten schon bei vier Zuordnungen 8 %.

Ausblick. Teil III verlässt die lineare Welt. Kapitel 11 führt quadratische Zielfunktionen und die KKT-Bedingungen ein — das mathematische Fundament der Portfoliooptimierung.

Kapitel 9: Metaheuristiken — wenn der exakte Solver aussteigt

Kapitel auf einen Blick

Worum geht es? Um den Fall, für den die bisherigen Kapitel keine Antwort haben: Das Problem ist zu groß, der Solver kommt im Zeitlimit nicht zu einem brauchbaren Ergebnis — und eine Entscheidung muss trotzdem heute fallen.

Voraussetzungen: Kapitel 6 und Kapitel 7, insbesondere Abschnitt 6.8 (MIP-Gap und Schranke).

Danach können Sie: eine lokale Suche mit ihrer entscheidenden Zutat — der Kostenänderung in O(1) — selbst schreiben, die Temperatur eines Simulated Annealing ausmessen statt raten, den Umschlagpunkt zwischen exakt und heuristisch für Ihr eigenes Problem bestimmen, und mit Large Neighborhood Search beides kombinieren.

Zeitbedarf: ca. 5 Stunden.

Programme:
`Simulated_Annealing.py`
`Metaheuristik_vs_Exakt.py`
`Large_Neighborhood_Search.py`

Notebook: `Notebooks_04/metaheuristiken.ipynb`

9.1 In 5 Minuten gelöst

□ In 5 Minuten gelöst: Sechs Lackaufträge, eine Reihenfolge

Eine Lackieranlage hat sechs Aufträge offen. Zwischen zwei Aufträgen muss gereinigt werden; wie lange das dauert, hängt von **beiden** ab — von Hell auf Dunkel geht schnell, umgekehrt dauert es. Die Anlage steht gerade auf *Weiß*.

von \ nach	Weiß	Elfenbein	Rot	Dunkelrot	Schwarz	Beige
Weiß	—	2	49	49	49	2
Elfenbein	2	—	49	49	49	2
Rot	50	50	—	2	2	50
Dunkelrot	53	53	5	—	2	50
Schwarz	77	77	29	26	—	74
Beige	5	5	52	49	49	—

```
from itertools import permutations
> lack = ["Weiss", "Elfenbein", "Rot", "Dunkelrot", "Schwarz", "Beige"]
ruest = [[0, 2, 49, 49, 49, 2],      # Ruestzeit in Minuten von Zeile nach Spalte
         [2, 0, 49, 49, 49, 2],
         [50, 50, 0, 2, 2, 50],
         [53, 53, 5, 0, 2, 50],
         [77, 77, 29, 26, 0, 74],
         [5, 5, 52, 49, 49, 0]]
> dauer = lambda p: sum(ruest[p[k]][p[k + 1]] for k in range(len(p) - 1))
> offen, plan = set(range(1, 6)), [0]                  # Faustregel: billigster
 ↪  naechster
while offen:
    naechster = min(offen, key=lambda j: ruest[plan[-1]][j])
    plan.append(naechster); offen.discard(naechster)
> best = min(permutations(range(1, 6)), key=lambda p: dauer((0,) + p))
print("Faustregel:", " -> ".join(lack[i] for i in plan), f"= {dauer(plan)} min")
print("Optimal   :", " -> ".join(lack[i] for i in (0,) + best), f"= {dauer((0,) +
 ↪  best)} min")
```

Ausgabe:

```
Faustregel: Weiss -> Elfenbein -> Beige -> Dunkelrot -> Schwarz -> Rot = 84 min
Optimal   : Weiss -> Elfenbein -> Beige -> Rot -> Dunkelrot -> Schwarz = 60 min
```

24 Minuten Unterschied, 40 % — bei sechs Aufträgen. Und die Faustregel ist nicht dumm: Sie nimmt immer den billigsten nächsten Schritt.

Und jetzt der Punkt. Sehen Sie sich an, wo die Faustregel danebengreift. Beide Pläne beginnen gleich: Weiß → Elfenbein → Beige, alles hell, alles billig. Dann muss die Anlage in die dunkle Gruppe wechseln, und das kostet in jedem Fall. Die Faustregel wählt den billigsten Übergang, den sie sieht: Beige →

Dunkelrot für 49 Minuten. Der optimale Plan nimmt stattdessen Beige → **Rot** für 52 Minuten — drei Minuten teurer.

Diese drei Minuten mehr sparen am Ende 27. Denn wer über Dunkelrot einsteigt, lässt *Rot* übrig, und Rot ist von Schwarz aus nur für 29 Minuten erreichbar. Der optimale Plan betritt die dunkle Gruppe an der richtigen Stelle und arbeitet sie dann von hell nach dunkel ab: Rot → Dunkelrot → Schwarz, zweimal 2 Minuten.

> **Merksatz** Der Fehler der Faustregel ist nicht Gier, sondern **Kurzsichtigkeit**. Sie bewertet einen Schritt danach, was er *kostet*, und nicht danach, was er *übrig lässt*. Jede Heuristik in diesem Kapitel ist eine Antwort auf genau diese Schwäche.

Warum funktioniert das? Bei sechs Aufträgen konnten wir alle $5! = 120$ Reihenfolgen durchprobieren und wissen deshalb *sicher*, dass 60 Minuten das Minimum sind. Bei 20 Aufträgen sind es schon $19! \approx 1{,}2 \cdot 10^{17}$ — und damit ist der Weg dieses Schnellstarts versperrt. Das ganze Kapitel handelt davon, was an seine Stelle tritt.

9.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, warum eine lokale Suche ohne **Kostenänderung in O(1)** praktisch wertlos ist.
2. … Simulated Annealing implementieren und seine Starttemperatur **an der Zuggröße** kalibrieren statt am Zielfunktionswert.
3. … den Beitrag der Metaheuristik gegen die einfachere Alternative abgrenzen — reines Bergsteigen — statt ihn ihr gutzuschreiben.
4. … den **Umschlagpunkt** bestimmen, ab dem eine Metaheuristik den exakten Solver schlägt, und begründen, warum diese Zahl problemspezifisch ist.
5. … erklären, warum ein exakter Lauf auch dann dazugehört, wenn Sie am Ende die heuristische Lösung einsetzen.
6. … Large Neighborhood Search aufsetzen: zerstören, exakt reparieren, übernehmen.

9.3 Die Aufgabe: Rüstzeiten an der Lackieranlage

Der Schnellstart war ein Spielzeug. Die Aufgabe dieses Kapitels ist dieselbe, nur in Betriebsgröße: n **Aufträge in eine Reihenfolge bringen, so dass die Summe der Rüstzeiten minimal wird.**

$$\min_{\pi} \sum_{k=1}^{n-1} s_{\pi(k),\,\pi(k+1)}$$

> **Formel-Übersetzer**

Mathematik	Alltagssprache
π	„Der Plan: welcher Auftrag an welcher Position läuft.“
$\pi(k)$	„Der Auftrag, der als k-ter drankommt.“
$s_{i,j}$	„Wie lange die Anlage stillsteht, wenn nach Auftrag i der Auftrag j kommt.“
$s_{i,j} \neq s_{j,i}$	„Von Schwarz auf Weiß ist etwas anderes als von Weiß auf Schwarz.“
$\sum_{k=1}^{n-1}$	„Über alle **Übergänge** summieren — bei n Aufträgen gibt es $n-1$ davon, nicht n.“
$\min_{\pi}$	„Unter allen $n!$ möglichen Reihenfolgen die billigste.“

> **Ohne Formel gesagt:** „Sortiere die Aufträge so, dass möglichst wenig geputzt werden muss.“

Wer die Aufgabe als Graph liest, erkennt sie wieder: Es ist ein **asymmetrisches Rundreiseproblem** (Kapitel 8) mit offenem Ende. Das ist eine gute und eine schlechte Nachricht. Gut, weil damit alles bekannt ist, was man über das Problem wissen kann. Schlecht, weil dazu gehört, dass es NP-schwer ist.

> □ **Definition: Metaheuristik**
>
> Ein **Verfahrensrahmen**, der eine vorhandene Lösung schrittweise verändert und dabei steuert, welche Veränderungen übernommen werden. „Meta“, weil der Rahmen nichts über das Problem weiß: Er braucht nur eine Startlösung, einen Zug und eine Bewertung. Dieselbe Mechanik läuft über Tourenplanung, Personaleinsatz und Portfolioauswahl.
>
> Der Preis dafür steht in derselben Zeile: Ein Verfahren, das nichts über das Problem weiß, kann auch nichts über die Güte seines Ergebnisses sagen. Es liefert eine Lösung, **keine Schranke**.

Die Instanz

Alle Programme dieses Kapitels benutzen dieselbe Aufgabe. Die Rüstzeit setzt sich aus zwei Teilen zusammen: einem unregelmäßigen Reinigungsaufwand zwischen den Produktfamilien und einem Zuschlag für den Wechsel von Dunkel nach Hell.

Eine Feinheit der Instanz ist wichtiger, als sie aussieht: Die **Zahl der Produktfamilien wächst mit der Auftragszahl** (eine Familie je zehn Aufträge). Bei fester Familienzahl würde das Problem mit wachsendem n nämlich *leichter* — der Plan bestünde irgendwann aus ein paar großen Blöcken, und jede Faustregel fände ihn. Beim ersten Entwurf dieses Kapitels ist mir genau das passiert: Die Metaheuristik verbesserte bei 60 Aufträgen um 30 %, bei 200 Aufträgen um 0 %. Nicht weil sie versagte, sondern weil es nichts mehr zu verbessern gab.

9.4 Lokale Suche: der Zug und seine Kosten

Eine lokale Suche braucht drei Dinge: eine Startlösung, einen **Zug** und eine Regel, welche Züge übernommen werden.

Der Zug in diesem Kapitel ist der einfachste, der für asymmetrische Rüstzeiten taugt: **einen Auftrag herausnehmen und woanders einsetzen**. (Das klassische 2-opt — ein Teilstück umdrehen — scheidet aus: Beim Umdrehen kehren sich alle Übergänge *innerhalb* des Stücks um, und weil $s_{i,j} \neq s_{j,i}$ ist, müsste man sie alle neu berechnen.)

> **Die Stelle, an der Metaheuristiken scheitern**
>
> Es ist verlockend, die Kosten eines Zugs so zu bewerten:
>
> ```
> neue_kosten = gesamtruestzeit(neue_reihe, matrix)      # falsch gedacht
> ```
>
> Das ist korrekt und trotzdem der Fehler, der ein Projekt scheitern lässt. Die volle Summe kostet $O(n)$ je Zug. Herausnehmen und Einsetzen verändert aber nur **drei Kanten** — die Kostenänderung steht in höchstens sechs Matrixeinträgen und ist in $O(1)$ zu haben.
>
> Bei 500 Aufträgen ist das der Unterschied zwischen einigen tausend und einigen hunderttausend geprüften Zügen im selben Zeitbudget. Eine Metaheuristik lebt von der Zahl der Züge; wer sie durch eine bequeme Bewertungsfunktion um zwei Größenordnungen drückt, misst am Ende nicht das Verfahren, sondern seine eigene Implementierung.

Herausgenommen wird der Auftrag aus zwei Kanten, seine Nachbarn rücken zusammen — das ist eine neue Kante. Eingesetzt wird er zwischen zwei andere Nachbarn, deren bisherige Kante dadurch verschwindet. Drei Kanten weniger, drei Kanten mehr:

$$\Delta = \underbrace{s_{a,x} + s_{x,b} - s_{a,b}}_{\text{einsetzen}} - \underbrace{\left(s_{u,x} + s_{x,v} - s_{u,v}\right)}_{\text{herausnehmen}}$$

wobei x der verschobene Auftrag ist, u, v seine alten und a, b seine neuen Nachbarn.

9.5 Simulated Annealing

Reines Bergsteigen — nur Verbesserungen annehmen — bleibt im ersten lokalen Optimum stehen. **Simulated Annealing** nimmt Verschlechterungen mit einer Wahrscheinlichkeit an, die von der Größe der Verschlechterung und von einer sinkenden *Temperatur* abhängt:

$$P(\text{annehmen}) = \begin{cases} 1 & \Delta \leq 0 \\ e^{-\Delta / T} & \Delta > 0 \end{cases}$$

> **Formel-Übersetzer**

Mathematik	Alltagssprache
$\Delta \leq 0 \Rightarrow P = 1$	„Was besser ist, wird immer genommen.“
$e^{-\Delta/T}$	„Kleine Verschlechterungen gehen eher durch als große.“
T groß	„Am Anfang darf die Suche herumlaufen.“
$T \to 0$	„Zum Schluss wird nur noch bergab gegangen.“
$\Delta = T$	„Genau diese Verschlechterung geht in 37 % der Fälle durch (e^{-1}).“

Die letzte Zeile ist die praktisch wichtigste: T **und** Δ **werden in derselben Einheit gemessen.** Wer die Temperatur wählt, ohne die typische Zuggröße zu kennen, wählt blind.

```
#!/usr/bin/env python3

# Simulated_Annealing.py
"""
Kapitel Metaheuristiken: Simulated Annealing an der Lackieranlage.

Aufgabe: n Auftraege in eine Reihenfolge bringen. Zwischen zwei Auftraegen
faellt eine Ruestzeit an - die Anlage muss gereinigt werden. Wie lange das
dauert, haengt von BEIDEN Auftraegen ab: Ein Wechsel innerhalb derselben
Produktfamilie kostet fast nichts, ein Wechsel von Dunkel nach Hell ist teuer,
und zwischen manchen Familien ist der Reinigungsaufwand schlicht hoch. Gesucht
ist die Reihenfolge mit der kleinsten Summe der Ruestzeiten.

Das Programm zeigt drei Dinge:

  1. Warum die Faustregel "immer der billigste naechste Auftrag" in eine
     Sackgasse laeuft - und wie weit man sie mit lokaler Suche verbessert.
  2. Wie gross der Beitrag der Verschlechterungen WIRKLICH ist. Die Antwort
     faellt bescheidener aus als das Lehrbuch verspricht, und genau das ist
     der Grund, den Vergleich immer mitzurechnen.
  3. Dass die Starttemperatur kein Beiwerk ist: zu heiss macht die Suche
     nutzlos, und man sieht es an einer einzigen Kennzahl kommen.

Zum Budget: Der Abkuehlplan laeuft ueber eine feste ZUGZAHL, nicht ueber eine
Sekundenzahl. Im Betrieb hat man zwar ein Zeitbudget - fuer einen VERGLEICH ist
die Zugzahl aber das ehrlichere Mass: Sie ist auf jeder Maschine dieselbe, und
die abgedruckten Zahlen unten lassen sich damit nachrechnen. Die gemessene
Laufzeit steht trotzdem dabei.

Benoetigt: numpy
"""

from __future__ import annotations
```

```
import math
import time

import numpy as np

ZUEGE = 400_000           # Zugbudget je Lauf (statt Sekunden: reproduzierbar)
SAAT = 11                 # feste Saat -> reproduzierbare Instanz


# --- 1. Die Instanz -----------------------------------------------------

def erzeuge_ruestmatrix(n: int, saat: int = SAAT) -> np.ndarray:
    """Ruestzeiten in Minuten zwischen je zwei Auftraegen.

    Die Zahl der Produktfamilien waechst mit der Auftragszahl (eine Familie je
    zehn Auftraege). Sonst wuerde das Problem mit wachsendem n LEICHTER: Bei
    fester Familienzahl haette der Plan irgendwann nur noch ein paar grosse
    Bloecke, und jede Faustregel faende ihn.
    """
    rng = np.random.default_rng(saat)
    familien = max(6, n // 10)
    # Reinigungsaufwand zwischen den Familien - unregelmaessig und asymmetrisch,
    # so wie im Betrieb: Von Klarlack auf Rot ist etwas anderes als umgekehrt.
    zwischen = rng.integers(8, 60, (familien, familien))
    np.fill_diagonal(zwischen, 2)

    familie = rng.integers(0, familien, n)
    farbe = rng.integers(0, 10, n)          # 0 = weiss ... 9 = schwarz

    matrix = np.zeros((n, n), dtype=np.int64)
    for i in range(n):
        for j in range(n):
            if i != j:
                # Dunkel -> hell kostet zusaetzlich: die Anlage muss heller
                # werden, das braucht mehr Spuelgaenge.
                matrix[i, j] = (zwischen[familie[i], familie[j]]
                                + 3 * max(0, farbe[i] - farbe[j]))
    return matrix


def gesamtruestzeit(reihe: list[int], matrix: np.ndarray) -> int:
    """Summe der Ruestzeiten einer Reihenfolge (offene Kette, keine Rundreise)."""
    return int(sum(matrix[reihe[k], reihe[k + 1]] for k in range(len(reihe) - 1)))


def faustregel(matrix: np.ndarray) -> list[int]:
    """Immer der billigste noch offene Auftrag - 'naechster Nachbar'.
```

```
    So plant ein Meister von Hand, und es ist keine schlechte Regel. Ihr Fehler
    ist die Kurzsichtigkeit: Sie spart am Anfang und laesst die teuren Wechsel
    fuer das Ende uebrig, wo keine Wahl mehr bleibt.
    """
    n = len(matrix)
    offen = set(range(1, n))
    reihe = [0]
    while offen:
        naechster = min(offen, key=lambda j: matrix[reihe[-1], j])
        reihe.append(naechster)
        offen.discard(naechster)
    return reihe


# --- 2. Der Zug und seine Kostenaenderung ----------------------------------

def delta_verschieben(reihe: list[int], matrix: np.ndarray,
                      von: int, nach: int):
    """Auftrag von Position 'von' an Position 'nach' versetzen.

    Liefert (Kostenaenderung, Restliste, Einfuegeposition) oder None, wenn der
    Zug nichts aendert.

    DAS IST DIE WICHTIGSTE FUNKTION DES PROGRAMMS. Sie berechnet die
    Kostenaenderung aus HOECHSTENS SECHS Matrixeintraegen, statt die
    Reihenfolge neu durchzusummieren. Der Unterschied ist nicht kosmetisch:
    Die volle Summe kostet O(n) je Zug, diese Rechnung O(1). Bei 500 Auftraegen
    sind das rund 500-mal mehr geprueste Zuege im selben Zeitbudget - und eine
    Metaheuristik lebt von der Zahl der Zuege.

    Herausgerissen wird der Auftrag aus zwei Kanten, seine Nachbarn ruecken
    zusammen (eine neue Kante). Eingefuegt wird er zwischen zwei andere
    Nachbarn, deren bisherige Kante dadurch verschwindet.
    """
    n = len(reihe)
    if von == nach or nach == von + 1:
        return None

    heraus = 0
    if von > 0:
        heraus += matrix[reihe[von - 1], reihe[von]]
    if von < n - 1:
        heraus += matrix[reihe[von], reihe[von + 1]]
    if 0 < von < n - 1:                       # die Nachbarn ruecken zusammen
        heraus -= matrix[reihe[von - 1], reihe[von + 1]]

    rest = reihe[:von] + reihe[von + 1:]
    stelle = nach if nach < von else nach - 1
```

```
    hinein = 0
    if stelle > 0:
        hinein += matrix[rest[stelle - 1], reihe[von]]
    if stelle < len(rest):
        hinein += matrix[reihe[von], rest[stelle]]
    if 0 < stelle < len(rest):               # die aufgetrennte Kante faellt weg
        hinein -= matrix[rest[stelle - 1], rest[stelle]]

    return int(hinein - heraus), rest, stelle


# --- 3. Die Suche -----------------------------------------------------------

class Verlauf:
    """Was ein Lauf ausser dem Ergebnis noch verraet."""

    def __init__(self) -> None:
        self.schritte = 0
        self.verschlechterungen_geprueft = 0
        self.verschlechterungen_genommen = 0
        self.schlechtester_zustand = 0      # wie weit die Suche abgedriftet ist
        self.sekunden = 0.0                 # nur zur Information, nicht zum Vergleich

    @property
    def annahmequote(self) -> float:
        if not self.verschlechterungen_geprueft:
            return 0.0
        return self.verschlechterungen_genommen / self.verschlechterungen_geprueft


def suche(matrix: np.ndarray, zuege: int, start_temperatur: float,
          end_temperatur: float = 0.05, saat: int = 1,
          bergsteigen: bool = False) -> tuple[list[int], int, Verlauf]:
    """Simulated Annealing (oder reines Bergsteigen, wenn bergsteigen=True).

    Der einzige Unterschied zwischen beiden steht in der Annahmezeile: Das
    Bergsteigen nimmt nur Verbesserungen, SA nimmt Verschlechterungen mit einer
    Wahrscheinlichkeit an, die mit der Zeit gegen null geht.
    """
    rng = np.random.default_rng(saat)
    n = len(matrix)
    reihe = faustregel(matrix)
    kosten = gesamtruestzeit(reihe, matrix)
    beste, beste_kosten = reihe[:], kosten
    verlauf = Verlauf()
    verlauf.schlechtester_zustand = kosten
```

```
    start = time.perf_counter()
    for zug in range(zuege):
        verlauf.schritte += 1
        # Geometrischer Abkuehlplan ueber das Zugbudget.
        temperatur = start_temperatur * (end_temperatur / start_temperatur) \
            ** (zug / zuege)

        von = int(rng.integers(0, n))
        nach = int(rng.integers(0, n + 1))
        ergebnis = delta_verschieben(reihe, matrix, von, nach)
        if ergebnis is None:
            continue
        aenderung, rest, stelle = ergebnis

        if aenderung <= 0:
            annehmen = True
        else:
            verlauf.verschlechterungen_geprueft += 1
            # Metropolis-Kriterium: je groesser die Verschlechterung und je
            # kaelter es ist, desto unwahrscheinlicher.
            annehmen = (not bergsteigen
                        and rng.random() < math.exp(-aenderung / temperatur))
            verlauf.verschlechterungen_genommen += annehmen

        if annehmen:
            reihe = rest[:stelle] + [reihe[von]] + rest[stelle:]
            kosten += aenderung
            verlauf.schlechtester_zustand = max(verlauf.schlechtester_zustand,
                                                kosten)
            if kosten < beste_kosten:
                beste, beste_kosten = reihe[:], kosten

    verlauf.sekunden = time.perf_counter() - start
    return beste, beste_kosten, verlauf


if __name__ == "__main__":
    N = 200
    matrix = erzeuge_ruestmatrix(N)
    start_reihe = faustregel(matrix)
    start_kosten = gesamtruestzeit(start_reihe, matrix)

    print("=" * 78)
    print("  SIMULATED ANNEALING AN DER LACKIERANLAGE")
    print("=" * 78)
    print(f"{N} Auftraege, {max(6, N // 10)} Produktfamilien, "
          f"Zugbudget {ZUEGE:,} je Lauf.\n")
    print(f"Faustregel 'billigster naechster Auftrag': "
```

```
          f"{start_kosten:,} Minuten Ruestzeit")

    # --- Wie gross sind die Zuege ueberhaupt? --------------------------
    # Ohne diese Zahl kann man die Temperatur nicht waehlen: Das
    # Metropolis-Kriterium vergleicht die Verschlechterung MIT der Temperatur.
    rng = np.random.default_rng(0)
    stichprobe = []
    for _ in range(5000):
        e = delta_verschieben(start_reihe, matrix,
                              int(rng.integers(0, N)), int(rng.integers(0, N + 1)))
        if e:
            stichprobe.append(abs(e[0]))
    print(f"Typische Zuggroesse |Delta|: Median {np.median(stichprobe):.0f}, "
          f"90 %-Quantil {np.quantile(stichprobe, 0.9):.0f} Minuten")

    # --- Bergsteigen gegen Annealing -----------------------------------
    print("\n" + "-" * 78)
    print("Nur Verbesserungen annehmen (Bergsteigen) gegen Annealing:\n")
    _, berg, berg_verlauf = suche(matrix, ZUEGE, 1.0, bergsteigen=True)
    _, gluehen, gluehen_verlauf = suche(matrix, ZUEGE, 1.0)
    print(f"  {'Faustregel (Start)':<34} {start_kosten:>7,} Minuten")
    print(f"  {'Bergsteigen':<34} {berg:>7,} Minuten  "
          f"({(start_kosten - berg) / start_kosten * 100:5.1f} % besser)")
    print(f"  {'Simulated Annealing':<34} {gluehen:>7,} Minuten  "
          f"({(start_kosten - gluehen) / start_kosten * 100:5.1f} % besser)")
    print(f"\n  Beide pruefen genau {berg_verlauf.schritte:,} Zuege "
          f"({berg_verlauf.sekunden:.1f} bzw. {gluehen_verlauf.sekunden:.1f} Sekunden).")
    print(f"  Annealing nimmt davon {gluehen_verlauf.annahmequote * 100:.2f} % der "
          f"VERSCHLECHTERUNGEN an, Bergsteigen keine.")
    print("\n  Bemerkenswert ist, wie klein der Abstand ist: Das simple")
    print(f"  Bergsteigen holt hier bereits {(start_kosten - berg) / (start_kosten - gluehen)
     ↪  * 100:.0f} % dessen, was Annealing")
    print("  schafft. Wer eine Metaheuristik einsetzt, sollte diesen Vergleich")
    print("  IMMER mitrechnen - sonst schreibt man einer aufwendigen Methode")
    print("  gut, was schon die einfache geliefert haette.")

    # --- Die Temperatur ist kein Beiwerk -------------------------------
    print("\n" + "-" * 78)
    print("Und warum die Starttemperatur ausgemessen gehoert:\n")
    print(f"  {'Start-T':>8} {'Ruestzeit':>11} {'gg. Faustregel':>15} "
          f"{'angenommene':>12} {'schlechtester':>14}")
    print(f"  {'':>8} {'':>11} {'':>15} {'Verschlecht.':>12} {'Zwischenwert':>14}")
    print("  " + "-" * 64)
    ergebnisse, verlaeufe = {}, {}
    for temperatur in (0.5, 1.0, 2.0, 4.0, 8.0):
        _, wert, verlauf = suche(matrix, ZUEGE, temperatur)
        ergebnisse[temperatur] = wert
        verlaeufe[temperatur] = verlauf
```

```
        gewinn = (start_kosten - wert) / start_kosten * 100
        print(f"  {temperatur:>8.1f} {wert:>11,} {gewinn:>13.1f} % "
              f"{verlauf.annahmequote * 100:>11.2f} % "
              f"{verlauf.schlechtester_zustand:>14,}")

    beste_temperatur = min(ergebnisse, key=ergebnisse.get)
    heiss = verlaeufe[8.0]
    print(f"\n  Bester Wert bei T0 = {beste_temperatur}.")
    print("\n  Die letzten beiden Spalten erklaeren, warum es nach oben kippt.")
    print(f"  Bei T0 = 8 gehen nur {heiss.annahmequote * 100:.2f} % der Verschlechterungen")
    print("  durch - wenig genug, dass man es fuer harmlos halten koennte. Es")
    print(f"  genuegt aber: Die Suche driftet bis auf {heiss.schlechtester_zustand:,} Minuten
     ↪  ab, das")
    print(f"  {heiss.schlechtester_zustand / start_kosten:.1f}-fache der Startloesung, und
     ↪  findet im Zugbudget nicht")
    print("  mehr zurueck. Vom Ergebnis bleibt fast nichts uebrig: Die Suche hat")
    print(f"  in {ZUEGE:,} Zuegen nie etwas GESEHEN, das mehr als")
    print(f"  {(start_kosten - ergebnisse[8.0]) / start_kosten * 100:.1f} % unter der
     ↪  Startloesung lag.")
    print("\n  Bemerkenswert ist die andere Richtung: Die besten Werte entstehen")
    print("  bei Annahmequoten nahe null. Die Lehrbuchregel 'anfangs sollen")
    print("  20 bis 50 Prozent der Verschlechterungen durchgehen' fuehrt hier in")
    print("  die Irre - sie stammt aus Problemen mit sehr kleinen Zuggroessen.")
    print("  Bei Ruestzeiten in Minuten ist ein einziger schlechter Zug teuer.")
    print("\n  Was stattdessen traegt: Den SCHLECHTESTEN Zwischenwert protokollieren")
    print("  und T0 so waehlen, dass er die Startloesung nicht wesentlich")
    print("  ueberschreitet. Diese eine Zahl haette hier auf Anhieb gezeigt, dass")
    print("  T0 = 8 unbrauchbar ist - ohne dass man das Endergebnis abwarten muss.")
    print("=" * 78)
```

Erwartete Ausgabe (Laufzeiten hardwareabhängig, die Werte nicht):

```
==============================================================================
  SIMULATED ANNEALING AN DER LACKIERANLAGE
==============================================================================
200 Auftraege, 20 Produktfamilien, Zugbudget 400,000 je Lauf.

Faustregel 'billigster naechster Auftrag': 1,049 Minuten Ruestzeit
Typische Zuggroesse |Delta|: Median 70, 90 %-Quantil 99 Minuten

------------------------------------------------------------------------------
Nur Verbesserungen annehmen (Bergsteigen) gegen Annealing:

  Faustregel (Start)                  1,049 Minuten
  Bergsteigen                           917 Minuten  ( 12.6 % besser)
  Simulated Annealing                   907 Minuten  ( 13.5 % besser)

  Beide pruefen genau 400,000 Zuege (3.0 bzw. 3.3 Sekunden).
```

```
  Annealing nimmt davon 0.01 % der VERSCHLECHTERUNGEN an, Bergsteigen keine.

  Bemerkenswert ist, wie klein der Abstand ist: Das simple
  Bergsteigen holt hier bereits 93 % dessen, was Annealing
  schafft. Wer eine Metaheuristik einsetzt, sollte diesen Vergleich
  IMMER mitrechnen - sonst schreibt man einer aufwendigen Methode
  gut, was schon die einfache geliefert haette.

--------------------------------------------------------------------------------
Und warum die Starttemperatur ausgemessen gehoert:

   Start-T   Ruestzeit  gg. Faustregel  angenommene  schlechtester
                                        Verschlecht.   Zwischenwert
  ------------------------------------------------------------------
       0.5         919          12.4 %        0.00 %          1,049
       1.0         907          13.5 %        0.01 %          1,057
       2.0         914          12.9 %        0.02 %          1,076
       4.0         939          10.5 %        0.11 %          1,329
       8.0       1,041           0.8 %        0.29 %          2,193

  Bester Wert bei T0 = 1.0.

  Die letzten beiden Spalten erklaeren, warum es nach oben kippt.
  Bei T0 = 8 gehen nur 0.29 % der Verschlechterungen
  durch - wenig genug, dass man es fuer harmlos halten koennte. Es
  genuegt aber: Die Suche driftet bis auf 2,193 Minuten ab, das
  2.1-fache der Startloesung, und findet im Zugbudget nicht
  mehr zurueck. Vom Ergebnis bleibt fast nichts uebrig: Die Suche hat
  in 400,000 Zuegen nie etwas GESEHEN, das mehr als
  0.8 % unter der Startloesung lag.

  Bemerkenswert ist die andere Richtung: Die besten Werte entstehen
  bei Annahmequoten nahe null. Die Lehrbuchregel 'anfangs sollen
  20 bis 50 Prozent der Verschlechterungen durchgehen' fuehrt hier in
  die Irre - sie stammt aus Problemen mit sehr kleinen Zuggroessen.
  Bei Ruestzeiten in Minuten ist ein einziger schlechter Zug teuer.

  Was stattdessen traegt: Den SCHLECHTESTEN Zwischenwert protokollieren
  und T0 so waehlen, dass er die Startloesung nicht wesentlich
  ueberschreitet. Diese eine Zahl haette hier auf Anhieb gezeigt, dass
  T0 = 8 unbrauchbar ist - ohne dass man das Endergebnis abwarten muss.
================================================================================
```

Eine interaktive Fassung dieser Grafik steht auf den Kapitelseiten der Website bereit.

□ Code-Durchgang

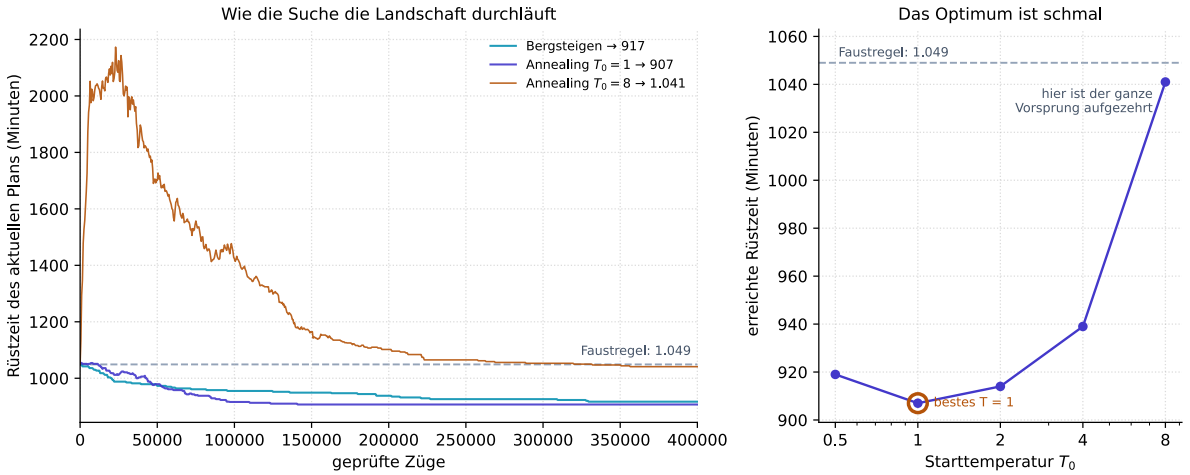

Abbildung 16: Abb. 9.1: Dieselbe Rechnung als Bild. Links der Zielwert während der Suche: Bergsteigen fällt monoton, Annealing bei $T_0 = 1$ fast genauso — bei $T_0 = 8$ irrt die Suche bis auf das 2,1-fache der Startlösung ab und findet im Zugbudget nicht zurück. Rechts das Ergebnis über die Starttemperatur: Das Optimum ist schmal, und jenseits davon ist der ganze Vorsprung aufgezehrt. Erzeugt von `bilder_04/erzeuge_metaheuristik_landschaft.py`.

Stelle	Was passiert
`delta_verschieben(...)`	die Kostenänderung aus höchstens sechs Matrixeinträgen — der Kern des Programms
`for zug in range(zuege)`	Budget in **Zügen**, nicht in Sekunden. Damit sind die abgedruckten Werte auf jeder Maschine dieselben
`temperatur = T0 * (T1/T0)**(zug/zuege)`	geometrischer Abkühlplan: gleiche relative Absenkung je Zug
`annehmen = rng.random() < math.exp(-aenderung/temperatur)`	das Metropolis-Kriterium, eine Zeile
`beste, beste_kosten`	die beste je **gesehene** Lösung wird getrennt mitgeführt. Ohne das gäbe man am Ende den zufälligen Endzustand aus
`Verlauf.schlechtester_zustand`	die Diagnosezahl, um die es unten geht

Was die Messung zeigt — und was sie nicht zeigt

Erstens: Der Beitrag des Annealings ist bescheiden. Bergsteigen kommt auf 917 Minuten, Annealing auf 907 — das simple Verfahren holt 93 % des Ertrags. Das ist kein Argument gegen Annealing, aber ein starkes Argument dafür, **immer beide zu rechnen**. Wer nur die aufwendige Variante misst, schreibt ihr gut, was schon die einfache geliefert hätte.

Zweitens: Die Temperatur ist keine Stellschraube unter vielen. Bei $T_0 = 8$ liefert das Verfahren praktisch nichts mehr. Die letzten beiden Spalten der Tabelle erklären, warum — und die Erklärung ist nicht die, die man erwartet:

Start-T_0	angenommene Verschlechterungen	schlechtester Zwischenwert	Ergebnis
0,5	0,00 %	1 049	919
1,0	0,01 %	1 057	**907**
2,0	0,02 %	1 076	914
4,0	0,11 %	1 329	939
8,0	0,29 %	2 193	1 041

Bei $T_0 = 8$ gehen nur **0,29 %** der Verschlechterungen durch. Das klingt harmlos. Es genügt aber, um die Suche bis auf das 2,1-fache der Startlösung abdriften zu lassen — und aus diesem Zustand findet sie im verbleibenden Budget nicht zurück.

> **Eine Lehrbuchregel, die hier in die Irre führt**
>
> In vielen Darstellungen steht: *„Wählen Sie T_0 so, dass anfangs 20 bis 50 % der Verschlechterungen angenommen werden.“* Auf dieser Aufgabe wäre das katastrophal — die besten Werte entstehen bei Annahmequoten **nahe null**.
>
> Der Grund ist die Zuggröße. Die Regel stammt aus Problemen, in denen ein einzelner Zug die Lösung kaum verändert. Hier kostet ein schlechter Zug im Median 70 Minuten bei einer Gesamtlösung von rund 1 000 — jeder einzelne Zug ist eine spürbare Verschlechterung.
>
> **Was stattdessen trägt:** den *schlechtesten Zwischenwert* protokollieren und T_0 so wählen, dass er die Startlösung nicht wesentlich überschreitet. Diese eine Zahl zeigt sofort, dass $T_0 = 8$ unbrauchbar ist — ohne dass man das Endergebnis abwarten muss.

9.6 Ab wann lohnt es sich?

„Kleine Probleme exakt, große mit Metaheuristiken“ ist richtig und nutzlos, solange niemand sagt, wo *groß* anfängt. Diese Zahl lässt sich messen.

```
#!/usr/bin/env python3

# Metaheuristik_vs_Exakt.py
"""
Kapitel Metaheuristiken: Ab wann lohnt sich eine Metaheuristik? Die Antwort ist eine Zahl.

Die verbreitete Regel lautet: "Kleine Probleme exakt, grosse mit
Metaheuristiken." Sie ist richtig - aber voellig nutzlos, solange niemand sagt,
wo 'gross' anfaengt. Dieses Programm misst es fuer die Ruestzeitenaufgabe aus
Simulated_Annealing.py nach, und das Ergebnis ueberrascht in beide Richtungen:

  * Bei 60 Auftraegen loest CP-SAT BEWEISBAR optimal, in wenigen Sekunden.
    Die Metaheuristik ist hier schlicht die schlechtere Wahl - sie liefert ein
    schwaecheres Ergebnis und weiss nicht einmal, wie schwach.
  * Bei 200 Auftraegen liegt CP-SAT ohne Optimalitaetsbeweis vorne.
```

```
  * Erst bei 500 Auftraegen kippt es: Dort gibt CP-SAT im Zeitlimit eine
    Loesung aus, die SCHLECHTER ist als die Faustregel eines Meisters.

Der Punkt des Kapitels steht in der letzten Spalte: Die Schranke. Nur der
exakte Solver sagt, wie gut die Loesung sein KANN. Eine Metaheuristik allein
liefert eine Zahl ohne Massstab - man weiss nie, ob man 2 % oder 40 % neben dem
Optimum liegt.

ZUR REPRODUZIERBARKEIT: Die Spalte 'Annealing' ist auf jeder Maschine gleich -
sie laeuft ueber ein festes ZUGbudget. Die CP-SAT-Spalten sind es nicht: Ein
Zeitlimit ist ein Wanduhr-Limit, und wie weit der Solver in 30 Sekunden kommt,
haengt von der Maschine ab. CP-SAT kennt zwar ein deterministisches Zeitmass
(max_deterministic_time), das aber in einer Groessenordnung rechnet, die fuer
ein Buchbeispiel unbrauchbar langsam ist. Die Zahlen unten sind also
hardwareabhaengig - die AUSSAGE der Tabelle ist es nicht, sie ist auf
verschiedenen Groessen und in mehreren Laeufen stabil.

WICHTIG: Hier wird nur ortools importiert, nicht highspy (Kapitel Oekosystem).

Benoetigt: numpy, ortools
"""

from __future__ import annotations

import time

import numpy as np
from ortools.sat.python import cp_model

ZUEGE = 400_000           # Zugbudget der Metaheuristik (wie Simulated_Annealing.py)
ZEITLIMIT = 30.0          # Zeitbudget des exakten Solvers
GROESSEN = (60, 200, 500)


# --- Instanz, Faustregel und Suche: identisch zu Simulated_Annealing.py -----

def erzeuge_ruestmatrix(n: int, saat: int = 11) -> np.ndarray:
    rng = np.random.default_rng(saat)
    familien = max(6, n // 10)
    zwischen = rng.integers(8, 60, (familien, familien))
    np.fill_diagonal(zwischen, 2)
    familie = rng.integers(0, familien, n)
    farbe = rng.integers(0, 10, n)
    matrix = np.zeros((n, n), dtype=np.int64)
    for i in range(n):
        for j in range(n):
            if i != j:
                matrix[i, j] = (zwischen[familie[i], familie[j]]
```

```
                                + 3 * max(0, farbe[i] - farbe[j]))
    return matrix


def gesamtruestzeit(reihe: list[int], matrix: np.ndarray) -> int:
    return int(sum(matrix[reihe[k], reihe[k + 1]] for k in range(len(reihe) - 1)))


def faustregel(matrix: np.ndarray) -> list[int]:
    n = len(matrix)
    offen = set(range(1, n))
    reihe = [0]
    while offen:
        naechster = min(offen, key=lambda j: matrix[reihe[-1], j])
        reihe.append(naechster)
        offen.discard(naechster)
    return reihe


def delta_verschieben(reihe, matrix, von, nach):
    n = len(reihe)
    if von == nach or nach == von + 1:
        return None
    heraus = 0
    if von > 0:
        heraus += matrix[reihe[von - 1], reihe[von]]
    if von < n - 1:
        heraus += matrix[reihe[von], reihe[von + 1]]
    if 0 < von < n - 1:
        heraus -= matrix[reihe[von - 1], reihe[von + 1]]
    rest = reihe[:von] + reihe[von + 1:]
    stelle = nach if nach < von else nach - 1
    hinein = 0
    if stelle > 0:
        hinein += matrix[rest[stelle - 1], reihe[von]]
    if stelle < len(rest):
        hinein += matrix[reihe[von], rest[stelle]]
    if 0 < stelle < len(rest):
        hinein -= matrix[rest[stelle - 1], rest[stelle]]
    return int(hinein - heraus), rest, stelle


def annealing(matrix: np.ndarray, zuege: int = ZUEGE,
              start_temperatur: float = 1.0, saat: int = 1) -> tuple[int, float]:
    """Dieselbe Suche wie in Simulated_Annealing.py, hier nur als Vergleichswert."""
    import math

    rng = np.random.default_rng(saat)
```

```
    n = len(matrix)
    reihe = faustregel(matrix)
    kosten = gesamtruestzeit(reihe, matrix)
    beste_kosten = kosten

    t0 = time.perf_counter()
    for zug in range(zuege):
        temperatur = start_temperatur * (0.05 / start_temperatur) ** (zug / zuege)
        von = int(rng.integers(0, n))
        nach = int(rng.integers(0, n + 1))
        ergebnis = delta_verschieben(reihe, matrix, von, nach)
        if ergebnis is None:
            continue
        aenderung, rest, stelle = ergebnis
        if aenderung <= 0 or rng.random() < math.exp(-aenderung / temperatur):
            reihe = rest[:stelle] + [reihe[von]] + rest[stelle:]
            kosten += aenderung
            beste_kosten = min(beste_kosten, kosten)
    return beste_kosten, time.perf_counter() - t0


# --- Der exakte Weg: CP-SAT mit AddCircuit ---------------------------------

def exakt(matrix: np.ndarray, zeitlimit: float = ZEITLIMIT):
    """Reihenfolge als Rundreise mit einem kostenlosen Hilfsknoten.

    AddCircuit verlangt einen geschlossenen Kreis. Unsere Aufgabe ist aber eine
    offene Kette: Der erste Auftrag hat keinen Vorgaenger, der letzte keinen
    Nachfolger. Der uebliche Kniff ist ein zusaetzlicher Hilfsknoten n, dessen
    Kanten nichts kosten. Der Kreis laeuft dann Hilfsknoten -> Kette ->
    Hilfsknoten, und die Kosten des Kreises sind genau die der Kette.

    AddCircuit ist der Grund, warum CP-SAT hier so lange mithaelt: Die
    Nebenbedingung sorgt selbst dafuer, dass keine Teilkreise entstehen. Wer
    dasselbe als MILP formuliert, braucht dafuer entweder exponentiell viele
    Ungleichungen oder die schwache MTZ-Formulierung (Kapitel Graphen).
    """
    n = len(matrix)
    modell = cp_model.CpModel()
    kanten, ziel = [], []
    for i in range(n + 1):
        for j in range(n + 1):
            if i == j:
                continue
            aktiv = modell.NewBoolVar(f"kante_{i}_{j}")
            kanten.append((i, j, aktiv))
            if i < n and j < n:                  # Hilfsknoten kostet nichts
                ziel.append(int(matrix[i, j]) * aktiv)
```

```
    modell.AddCircuit(kanten)
    modell.Minimize(sum(ziel))

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = zeitlimit
    loeser.parameters.num_workers = 8
    loeser.parameters.random_seed = 1
    t0 = time.perf_counter()
    status = loeser.Solve(modell)
    dauer = time.perf_counter() - t0
    return (loeser.StatusName(status), int(loeser.ObjectiveValue()),
            int(loeser.BestObjectiveBound()), dauer)


if __name__ == "__main__":
    print("=" * 88)
    print("  AB WANN LOHNT SICH DIE METAHEURISTIK?")
    print("=" * 88)
    print(f"Dieselbe Aufgabe in drei Groessen. Metaheuristik: {ZUEGE:,} Zuege. "
          f"Exakt: {ZEITLIMIT:.0f} s Zeitlimit.\n")

    print(f"{'Auftraege':>10} {'Faustregel':>11} {'Annealing':>10} "
          f"{'CP-SAT':>9} {'Status':>10} {'Schranke':>9} {'CP-Zeit':>9}")
    print("-" * 88)

    zeilen = []
    for n in GROESSEN:
        matrix = erzeuge_ruestmatrix(n)
        start = gesamtruestzeit(faustregel(matrix), matrix)
        heuristisch, _ = annealing(matrix)
        status, wert, schranke, dauer = exakt(matrix)
        zeilen.append((n, start, heuristisch, wert, status, schranke, dauer))
        print(f"{n:>10} {start:>11,} {heuristisch:>10,} {wert:>9,} "
              f"{status:>10} {schranke:>9,} {dauer:>8.1f}s")

    print("-" * 88)
    print("\nWas in jeder Zeile steht:\n")
    for n, start, heur, wert, status, schranke, _ in zeilen:
        if status == "OPTIMAL":
            urteil = (f"CP-SAT beweist das Optimum. Annealing liegt "
                      f"{(heur - wert) / wert * 100:.1f} % daneben - und haette")
            zweite = "     das ohne den exakten Lauf nicht erfahren."
        elif wert <= heur:
            urteil = (f"CP-SAT liegt {(heur - wert) / heur * 100:.1f} % vor Annealing, "
                      f"ohne Beweis (Gap {(wert - schranke) / wert * 100:.1f} %).")
            zweite = "     Auch ohne Optimalitaetsbeweis ist es die bessere Wahl."
        else:
            urteil = (f"UMSCHLAGPUNKT: CP-SAT ist {(wert - heur) / heur * 100:.1f} % "
```

```
                    f"SCHLECHTER als Annealing")
            zweite = (f"     und sogar {(wert - start) / start * 100:.1f} % schlechter "
                      f"als die Faustregel des Meisters.")
        print(f"  {n:>4} Auftraege: {urteil}")
        print(zweite)

    n, start, heur, wert, status, schranke, _ = zeilen[-1]
    print("\n" + "=" * 88)
    print("  DIE SPALTE, DIE MAN NICHT WEGLASSEN DARF")
    print("=" * 88)
    print("Die Schranke ist der eigentliche Ertrag des exakten Solvers - auch dann,")
    print("wenn seine LOESUNG unbrauchbar ist.")
    print()
    print(f"Bei {n} Auftraegen liefert Annealing {heur:,} Minuten. Klingt gut: "
          f"{(start - heur) / start * 100:.1f} %")
    print("besser als die Faustregel. Die Schranke aus demselben CP-SAT-Lauf, dessen")
    print(f"Loesung wir gerade verworfen haben, sagt aber: Unter {schranke:,} Minuten geht")
    print(f"es nicht. Zwischen {schranke:,} und {heur:,} liegen {(heur - schranke) / heur *
     ↪  100:.0f} % - so viel Luft")
    print("kann noch nach oben sein.")
    print()
    print("Das ist der Unterschied zwischen 'besser als vorher' und 'gut'. Eine")
    print("Metaheuristik allein kann nur das Erste. Deshalb gehoert auch dann ein")
    print("exakter Lauf dazu, wenn man am Ende die heuristische Loesung einsetzt:")
    print("nicht wegen seiner Loesung, sondern wegen seiner Schranke.")
    print("=" * 88)
```

Erwartete Ausgabe (die CP-SAT-Spalten sind hardwareabhängig, siehe Programmkopf):

```
========================================================================================
  AB WANN LOHNT SICH DIE METAHEURISTIK?
========================================================================================
Dieselbe Aufgabe in drei Groessen. Metaheuristik: 400,000 Zuege. Exakt: 30 s Zeitlimit.

 Auftraege  Faustregel  Annealing    CP-SAT     Status  Schranke   CP-Zeit
----------------------------------------------------------------------------------------
        60         388        275       265    OPTIMAL       265      2.0s
       200       1,049        907       811   FEASIBLE       804     30.1s
       500       2,497      2,343     2,628   FEASIBLE     1,768     30.4s
----------------------------------------------------------------------------------------

Was in jeder Zeile steht:

    60 Auftraege: CP-SAT beweist das Optimum. Annealing liegt 3.8 % daneben - und haette
     das ohne den exakten Lauf nicht erfahren.
   200 Auftraege: CP-SAT liegt 10.6 % vor Annealing, ohne Beweis (Gap 0.9 %).
     Auch ohne Optimalitaetsbeweis ist es die bessere Wahl.
   500 Auftraege: UMSCHLAGPUNKT: CP-SAT ist 12.2 % SCHLECHTER als Annealing
```

```
    und sogar 5.2 % schlechter als die Faustregel des Meisters.

================================================================================
  DIE SPALTE, DIE MAN NICHT WEGLASSEN DARF
================================================================================
Die Schranke ist der eigentliche Ertrag des exakten Solvers - auch dann,
wenn seine LOESUNG unbrauchbar ist.

Bei 500 Auftraegen liefert Annealing 2,343 Minuten. Klingt gut: 6.2 %
besser als die Faustregel. Die Schranke aus demselben CP-SAT-Lauf, dessen
Loesung wir gerade verworfen haben, sagt aber: Unter 1,768 Minuten geht
es nicht. Zwischen 1,768 und 2,343 liegen 25 % - so viel Luft
kann noch nach oben sein.

Das ist der Unterschied zwischen 'besser als vorher' und 'gut'. Eine
Metaheuristik allein kann nur das Erste. Deshalb gehoert auch dann ein
exakter Lauf dazu, wenn man am Ende die heuristische Loesung einsetzt:
nicht wegen seiner Loesung, sondern wegen seiner Schranke.
================================================================================
```

Drei Größen, drei völlig verschiedene Empfehlungen:

Aufträge	Empfehlung
60	**Exakt.** CP-SAT beweist das Optimum in 2 Sekunden. Die Metaheuristik liegt 3,8 % daneben — und wüsste es nicht.
200	**Exakt.** Kein Beweis mehr, aber der Gap liegt unter 1 %, und der Wert ist rund 11 % besser als der heuristische.
500	**Heuristisch.** CP-SAT liefert eine Lösung, die 5,2 % **schlechter** ist als die Faustregel eines Meisters.

 Merksatz Der Umschlagpunkt ist keine Eigenschaft der Verfahren, sondern des **Problems**. Für die Standortplanung aus Abschnitt 6.8 liegt er woanders als hier. Wer ihn nicht für seine eigene Aufgabe gemessen hat, rät.

Die Spalte, die man nicht weglassen darf

Bei 500 Aufträgen haben wir die CP-SAT-*Lösung* verworfen. Seine **Schranke** ist trotzdem das Wertvollste in der ganzen Tabelle.

Annealing liefert 2 343 Minuten — 6,2 % besser als die Faustregel. Klingt nach einem Erfolg. Die Schranke sagt: unter 1 768 Minuten geht es nicht. Zwischen 1 768 und 2 343 liegen **25 %**.

Das ist der Unterschied zwischen *besser als vorher* und *gut*. Eine Metaheuristik allein kann nur das Erste behaupten. Deshalb gehört ein exakter Lauf auch dann dazu, wenn man am Ende die heuristische Lösung einsetzt — nicht wegen seiner Lösung, sondern wegen seiner Schranke.

9.7 Large Neighborhood Search

Die Tabelle oben endet mit einem Widerspruch. Bei 500 Aufträgen scheitert der exakte Solver an der **Größe**; die Metaheuristik scheitert an der **Kleinheit ihrer Züge** — sie verschiebt jeweils einen einzigen Auftrag und kann eine ganze Passage nicht auf einmal umbauen.

Large Neighborhood Search (LNS) setzt beide ein, jeden für das, was er kann:

Schritt	Was passiert
Zerstören	Ein zusammenhängendes Stück des Plans herausbrechen — hier 20 Aufträge.
Reparieren	Genau dieses Stück **exakt** neu optimieren. 20 Aufträge sind für CP-SAT eine Kleinigkeit, 500 sind es nicht.
Übernehmen	Nur behalten, wenn der Gesamtplan besser wurde.

Der Solver bekommt also nicht mehr das ganze Problem, sondern immer wieder ein kleines.

```
#!/usr/bin/env python3

# Large_Neighborhood_Search.py
"""
Kapitel Metaheuristiken: Zerstoeren und exakt reparieren - der Solver im Dienst der
 ↪  Heuristik.

Metaheuristik_vs_Exakt.py endet mit einem Widerspruch. Bei 500 Auftraegen gilt:

  * Der exakte Solver scheitert an der GROESSE - seine Loesung nach 30 Sekunden
    ist schlechter als die Faustregel eines Meisters.
  * Die Metaheuristik kommt weiter, laesst aber laut Schranke noch rund ein
    Viertel liegen. Ihre Zuege sind zu kleinteilig: Sie verschiebt jeweils
    EINEN Auftrag und kann eine ganze Passage nicht auf einmal umbauen.

Large Neighborhood Search loest den Widerspruch, indem sie beide einsetzt -
jeden fuer das, was er kann:

    ZERSTOEREN   Ein Stueck des Plans herausbrechen (hier: ein zusammen-
                 haengendes Fenster von 20 Auftraegen).
    REPARIEREN   Genau dieses Stueck EXAKT neu optimieren. 20 Auftraege sind
                 fuer CP-SAT eine Kleinigkeit, 500 sind es nicht.
    UEBERNEHMEN  Nur behalten, wenn der Gesamtplan besser wurde.

Der Solver bekommt also nicht mehr das ganze Problem, sondern immer wieder ein
kleines. Das ist der ganze Trick, und er ist in der Praxis der wichtigste
Baustein dieses Kapitels.
```

```
ZUR REPRODUZIERBARKEIT: Alle Laeufe haben eine feste RUNDENzahl, nicht ein
Zeitbudget - die Ergebniswerte sind damit auf jeder Maschine gleich. Die
Laufzeiten daneben sind hardwareabhaengig.

Benoetigt: numpy, ortools
"""

from __future__ import annotations

import math
import time

import numpy as np
from ortools.sat.python import cp_model

FENSTER = 20              # so viele Auftraege werden je Runde herausgebrochen
RUNDEN = 120
ZUEGE = 400_000           # Zugbudget des Annealings (wie Simulated_Annealing.py)
N = 500


# --- Instanz und Faustregel (wie in Simulated_Annealing.py) ----------------

def erzeuge_ruestmatrix(n: int, saat: int = 11) -> np.ndarray:
    rng = np.random.default_rng(saat)
    familien = max(6, n // 10)
    zwischen = rng.integers(8, 60, (familien, familien))
    np.fill_diagonal(zwischen, 2)
    familie = rng.integers(0, familien, n)
    farbe = rng.integers(0, 10, n)
    matrix = np.zeros((n, n), dtype=np.int64)
    for i in range(n):
        for j in range(n):
            if i != j:
                matrix[i, j] = (zwischen[familie[i], familie[j]]
                                + 3 * max(0, farbe[i] - farbe[j]))
    return matrix


def gesamtruestzeit(reihe: list[int], matrix: np.ndarray) -> int:
    return int(sum(matrix[reihe[k], reihe[k + 1]] for k in range(len(reihe) - 1)))


def faustregel(matrix: np.ndarray) -> list[int]:
    n = len(matrix)
    offen = set(range(1, n))
    reihe = [0]
```

```
    while offen:
        naechster = min(offen, key=lambda j: matrix[reihe[-1], j])
        reihe.append(naechster)
        offen.discard(naechster)
    return reihe


def delta_verschieben(reihe, matrix, von, nach):
    n = len(reihe)
    if von == nach or nach == von + 1:
        return None
    heraus = 0
    if von > 0:
        heraus += matrix[reihe[von - 1], reihe[von]]
    if von < n - 1:
        heraus += matrix[reihe[von], reihe[von + 1]]
    if 0 < von < n - 1:
        heraus -= matrix[reihe[von - 1], reihe[von + 1]]
    rest = reihe[:von] + reihe[von + 1:]
    stelle = nach if nach < von else nach - 1
    hinein = 0
    if stelle > 0:
        hinein += matrix[rest[stelle - 1], reihe[von]]
    if stelle < len(rest):
        hinein += matrix[reihe[von], rest[stelle]]
    if 0 < stelle < len(rest):
        hinein -= matrix[rest[stelle - 1], rest[stelle]]
    return int(hinein - heraus), rest, stelle


def annealing(matrix: np.ndarray, zuege: int = ZUEGE,
              start_temperatur: float = 1.0, saat: int = 1) -> list[int]:
    rng = np.random.default_rng(saat)
    n = len(matrix)
    reihe = faustregel(matrix)
    kosten = gesamtruestzeit(reihe, matrix)
    beste, beste_kosten = reihe[:], kosten
    for zug in range(zuege):
        temperatur = start_temperatur * (0.05 / start_temperatur) ** (zug / zuege)
        von, nach = int(rng.integers(0, n)), int(rng.integers(0, n + 1))
        ergebnis = delta_verschieben(reihe, matrix, von, nach)
        if ergebnis is None:
            continue
        aenderung, rest, stelle = ergebnis
        if aenderung <= 0 or rng.random() < math.exp(-aenderung / temperatur):
            reihe = rest[:stelle] + [reihe[von]] + rest[stelle:]
            kosten += aenderung
            if kosten < beste_kosten:
```

```
                beste, beste_kosten = reihe[:], kosten
    return beste


# --- Der Reparaturschritt: ein kleines Problem, exakt geloest -------------

def repariere_fenster(reihe: list[int], matrix: np.ndarray,
                      a: int, b: int) -> list[int] | None:
    """Ordnet reihe[a:b] optimal neu; die beiden Raender bleiben, wo sie sind.

    Die Raender festzuhalten ist entscheidend. Ohne sie waere das Teilproblem
    ein anderes als der Ausschnitt aus dem Gesamtplan: Der Uebergang vom
    Vorgaenger in das Fenster und aus dem Fenster in den Nachfolger gehoert
    zu den Kosten dazu. Genau dafuer steht der Hilfsknoten k - er vertritt
    beide Raender in einem.
    """
    innen = reihe[a:b]
    k = len(innen)
    vorgaenger = reihe[a - 1] if a > 0 else None
    nachfolger_rand = reihe[b] if b < len(reihe) else None

    modell = cp_model.CpModel()
    kanten, ziel = [], []
    for i in range(k + 1):
        for j in range(k + 1):
            if i == j:
                continue
            aktiv = modell.NewBoolVar(f"kante_{i}_{j}")
            kanten.append((i, j, aktiv))
            if i < k and j < k:
                ziel.append(int(matrix[innen[i], innen[j]]) * aktiv)
            elif i == k and j < k and vorgaenger is not None:
                ziel.append(int(matrix[vorgaenger, innen[j]]) * aktiv)
            elif j == k and i < k and nachfolger_rand is not None:
                ziel.append(int(matrix[innen[i], nachfolger_rand]) * aktiv)
    modell.AddCircuit(kanten)
    modell.Minimize(sum(ziel))

    loeser = cp_model.CpSolver()
    loeser.parameters.num_workers = 1
    loeser.parameters.random_seed = 1
    loeser.parameters.max_time_in_seconds = 10.0
    status = loeser.Solve(modell)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        # Auch das gehoert dazu: Wenn das Teilproblem nicht loest, bleibt der
        # Plan, wie er war. Eine LNS-Runde darf scheitern.
        return None
```

```
    nachfolger = {i: j for i, j, aktiv in kanten if loeser.Value(aktiv)}
    neue_folge, aktuell = [], nachfolger[k]
    while aktuell != k:
        neue_folge.append(innen[aktuell])
        aktuell = nachfolger[aktuell]
    return reihe[:a] + neue_folge + reihe[b:]


def lns(reihe: list[int], matrix: np.ndarray, runden: int = RUNDEN,
        fenster: int = FENSTER, saat: int = 3):
    """Zerstoeren, exakt reparieren, uebernehmen - so lange das Budget reicht."""
    rng = np.random.default_rng(saat)
    kosten = gesamtruestzeit(reihe, matrix)
    verbesserungen = 0
    start = time.perf_counter()
    for _ in range(runden):
        a = int(rng.integers(0, len(reihe) - fenster))
        neu = repariere_fenster(reihe, matrix, a, a + fenster)
        if neu is None:
            continue
        neue_kosten = gesamtruestzeit(neu, matrix)
        if neue_kosten < kosten:
            reihe, kosten = neu, neue_kosten
            verbesserungen += 1
    return reihe, kosten, verbesserungen, time.perf_counter() - start


if __name__ == "__main__":
    matrix = erzeuge_ruestmatrix(N)
    start_reihe = faustregel(matrix)
    start_kosten = gesamtruestzeit(start_reihe, matrix)
    SCHRANKE = 1768        # aus dem CP-SAT-Lauf in Metaheuristik_vs_Exakt.py

    print("=" * 84)
    print("  ZERSTOEREN UND EXAKT REPARIEREN")
    print("=" * 84)
    print(f"{N} Auftraege. Je Runde werden {FENSTER} aufeinanderfolgende Auftraege")
    print(f"herausgebrochen und exakt neu geordnet, {RUNDEN} Runden lang.\n")

    print("Erst die Metaheuristik allein, dann LNS auf ihrem Ergebnis:\n")
    sa_reihe = annealing(matrix)
    sa_kosten = gesamtruestzeit(sa_reihe, matrix)

    lns_ab_faustregel = lns(start_reihe[:], matrix)
    lns_ab_annealing = lns(sa_reihe[:], matrix)

    print(f"  {'Verfahren':<34} {'Ruestzeit':>10} {'ueber Schranke':>15}")
    print("  " + "-" * 62)
```

```
    for name, wert in [
            ("Faustregel (ohne Solver)", start_kosten),
            ("CP-SAT allein, 30 s", 2628),
            ("Simulated Annealing", sa_kosten),
            ("LNS ab Faustregel", lns_ab_faustregel[1]),
            ("Annealing, dann LNS", lns_ab_annealing[1])]:
        print(f"  {name:<34} {wert:>10,} {(wert - SCHRANKE) / SCHRANKE * 100:>13.1f} %")
    print(f"  {'untere Schranke (CP-SAT)':<34} {SCHRANKE:>10,} {0.0:>13.1f} %")

    print(f"\n  LNS ab Faustregel: {lns_ab_faustregel[2]} von {RUNDEN} Runden brachten "
          f"eine Verbesserung ({lns_ab_faustregel[3]:.0f} s).")
    print(f"  LNS ab Annealing:  {lns_ab_annealing[2]} von {RUNDEN} Runden "
          f"({lns_ab_annealing[3]:.0f} s).")

    # --- Die Fenstergroesse ------------------------------------------
    # Verglichen wird bei ANNAEHERND GLEICHER ZEIT, nicht bei gleicher
    # Rundenzahl: Eine Runde mit Fenster 30 kostet ein Vielfaches einer Runde
    # mit Fenster 10. Wer nach Runden vergleicht, misst nur, dass groessere
    # Teilprobleme mehr finden - und uebersieht, was sie kosten. Die
    # Rundenzahlen unten sind so gewaehlt, dass alle drei Laeufe in derselben
    # Groessenordnung liegen.
    print("\n" + "-" * 84)
    print("Die Fenstergroesse ist die eine Stellschraube - und sie hat ein")
    print("Optimum in der Mitte. Gleiche Zeit, verschiedene Fenster:\n")
    print(f"  {'Fenster':>8} {'Runden':>8} {'Ruestzeit':>11} {'Verbesser-':>12} {'Zeit':>8}")
    print(f"  {'':>8} {'':>8} {'':>11} {'ungen':>12} {'':>8}")
    print("  " + "-" * 52)
    fenster_ergebnis = {}
    for fenster, runden in ((10, 4000), (20, 150), (30, 20)):
        _, wert, verbesserungen, dauer = lns(sa_reihe[:], matrix,
                                             runden=runden, fenster=fenster)
        fenster_ergebnis[fenster] = wert
        print(f"  {fenster:>8} {runden:>8,} {wert:>11,} {verbesserungen:>12} "
              f"{dauer:>7.0f}s")

    print("\n  Nach unten ist die Grenze nicht die Zeit, sondern die SAETTIGUNG:")
    print(f"  Mit Fenster 10 bleiben von {4000:,} Runden nur eine Handvoll")
    print("  Verbesserungen uebrig. Zwanzig Auftraege lassen sich in einem Zug")
    print("  umbauen, zehn nicht - und was ein Fenster nicht umbauen kann, findet")
    print("  es auch in beliebig vielen Runden nicht.")
    print("\n  Nach oben ist die Grenze die Rechenzeit: Fenster 30 findet pro Runde")
    print("  mehr, kommt in derselben Zeit aber nur auf einen Bruchteil der Runden")
    print(f"  und landet bei {fenster_ergebnis[30]:,} statt {fenster_ergebnis[20]:,}. Eine
     ↪  Reparatur mit 30 Auftraegen")
    print("  ist eben genau das Problem, an dem der exakte Solver im Grossen")
    print("  scheitert - nur eine Nummer kleiner.")
    print("\n  Diese Tabelle gehoert an den Anfang jedes LNS-Projekts. Sie zu raten")
    print("  statt zu messen ist der haeufigste Fehler beim Einsatz des Verfahrens.")
```

```
    print("\n" + "=" * 84)
    print("  WAS DAS HEISST")
    print("=" * 84)
    bester = lns_ab_annealing[1]
    print(f"Der beste Plan kommt aus der Kombination: {bester:,} Minuten,")
    print(f"{(start_kosten - bester) / start_kosten * 100:.1f} % unter der Faustregel und "
          f"{(sa_kosten - bester) / sa_kosten * 100:.1f} % unter dem reinen Annealing.")
    print()
    print("Weder der exakte Solver noch die Metaheuristik allein kommen dorthin.")
    print("Der Solver scheitert an der Groesse, die Metaheuristik an der Kleinheit")
    print("ihrer Zuege. LNS gibt dem Solver Teilprobleme in einer Groesse, die er")
    print("beherrscht, und der Metaheuristik die Umbauten, die sie nicht kann.")
    print()
    print(f"Und die Ehrlichkeit zum Schluss: Auch {bester:,} liegt noch "
          f"{(bester - SCHRANKE) / SCHRANKE * 100:.0f} % ueber der")
    print("Schranke. Ob dort wirklich noch so viel Luft ist oder ob die Schranke")
    print("nur schwach ist, weiss man nicht - das ist die Lage, in der man mit")
    print("einer Metaheuristik arbeitet. Man kennt seine Loesung, nicht ihren")
    print("Abstand zum Optimum.")
    print("=" * 84)
```

Erwartete Ausgabe (Laufzeiten hardwareabhängig, die Werte nicht):

```
====================================================================================
  ZERSTOEREN UND EXAKT REPARIEREN
====================================================================================
500 Auftraege. Je Runde werden 20 aufeinanderfolgende Auftraege
herausgebrochen und exakt neu geordnet, 120 Runden lang.

Erst die Metaheuristik allein, dann LNS auf ihrem Ergebnis:

  Verfahren                          Ruestzeit  ueber Schranke
  ------------------------------------------------------------
  Faustregel (ohne Solver)               2,497          41.2 %
  CP-SAT allein, 30 s                    2,628          48.6 %
  Simulated Annealing                    2,343          32.5 %
  LNS ab Faustregel                      2,332          31.9 %
  Annealing, dann LNS                    2,289          29.5 %
  untere Schranke (CP-SAT)               1,768           0.0 %

  LNS ab Faustregel: 23 von 120 Runden brachten eine Verbesserung (29 s).
  LNS ab Annealing:  13 von 120 Runden (45 s).

------------------------------------------------------------------------------------
Die Fenstergroesse ist die eine Stellschraube - und sie hat ein
Optimum in der Mitte. Gleiche Zeit, verschiedene Fenster:
```

```
   Fenster   Runden   Ruestzeit   Verbesser-     Zeit
                                     ungen
  ------------------------------------------------
        10    4,000       2,319            6      28s
        20      150       2,289           13      48s
        30       20       2,298           10      82s

  Nach unten ist die Grenze nicht die Zeit, sondern die SAETTIGUNG:
  Mit Fenster 10 bleiben von 4,000 Runden nur eine Handvoll
  Verbesserungen uebrig. Zwanzig Auftraege lassen sich in einem Zug
  umbauen, zehn nicht - und was ein Fenster nicht umbauen kann, findet
  es auch in beliebig vielen Runden nicht.

  Nach oben ist die Grenze die Rechenzeit: Fenster 30 findet pro Runde
  mehr, kommt in derselben Zeit aber nur auf einen Bruchteil der Runden
  und landet bei 2,298 statt 2,289. Eine Reparatur mit 30 Auftraegen
  ist eben genau das Problem, an dem der exakte Solver im Grossen
  scheitert - nur eine Nummer kleiner.

  Diese Tabelle gehoert an den Anfang jedes LNS-Projekts. Sie zu raten
  statt zu messen ist der haeufigste Fehler beim Einsatz des Verfahrens.

================================================================================
  WAS DAS HEISST
================================================================================
Der beste Plan kommt aus der Kombination: 2,289 Minuten,
8.3 % unter der Faustregel und 2.3 % unter dem reinen Annealing.

Weder der exakte Solver noch die Metaheuristik allein kommen dorthin.
Der Solver scheitert an der Groesse, die Metaheuristik an der Kleinheit
ihrer Zuege. LNS gibt dem Solver Teilprobleme in einer Groesse, die er
beherrscht, und der Metaheuristik die Umbauten, die sie nicht kann.

Und die Ehrlichkeit zum Schluss: Auch 2,289 liegt noch 29 % ueber der
Schranke. Ob dort wirklich noch so viel Luft ist oder ob die Schranke
nur schwach ist, weiss man nicht - das ist die Lage, in der man mit
einer Metaheuristik arbeitet. Man kennt seine Loesung, nicht ihren
Abstand zum Optimum.
================================================================================
```

□ Code-Durchgang: warum die Ränder festgehalten werden

`repariere_fenster()` hält den Auftrag **vor** und **nach** dem Fenster fest. Das ist keine Vereinfachung, sondern notwendig: Der Übergang vom Vorgänger in das Fenster und aus dem Fenster in den Nachfolger gehört zu den Kosten. Ein Teilproblem, das diese beiden Kanten ignoriert, optimiert etwas anderes als den Ausschnitt aus dem Gesamtplan — und liefert Umbauten, die im Ganzen teurer sind.

Im Modell erledigt das der Hilfsknoten k: Er vertritt beide Ränder in einem. Seine eingehende Kante trägt die Kosten *letzter Auftrag → Nachfolger*, seine ausgehende die Kosten *Vorgänger → erster Auftrag*.

Das Fenster ist die eine Stellschraube

Fenster	Runden (gleiche Zeit)	Verbesserungen	Ergebnis
10	4 000	6	2 319
20	150	13	**2 289**
30	20	10	2 298

Das Optimum liegt **in der Mitte**, und zwar aus zwei verschiedenen Gründen:

- **Nach unten** ist die Grenze nicht die Zeit, sondern die **Sättigung.** Mit Fenster 10 bleiben von 4 000 Runden ganze 6 Verbesserungen übrig. Was ein Fenster nicht umbauen kann, findet es auch in beliebig vielen Runden nicht.
- **Nach oben** ist die Grenze die Rechenzeit. Eine Reparatur mit 30 Aufträgen ist genau das Problem, an dem der exakte Solver im Großen scheitert — nur eine Nummer kleiner.

 Merksatz Diese Tabelle gehört an den Anfang jedes LNS-Projekts, nicht ans Ende. Die Fenstergröße zu raten statt zu messen ist der häufigste Fehler beim Einsatz des Verfahrens.

Das Gesamtbild

Verfahren	Rüstzeit	über der Schranke
Faustregel (ohne Solver)	2 497	41,2 %
CP-SAT allein, 30 s	2 628	48,6 %
Simulated Annealing	2 343	32,5 %
LNS ab Faustregel	2 332	31,9 %
Annealing, dann LNS	**2 289**	**29,5 %**
untere Schranke (CP-SAT)	1 768	0 %

Der beste Plan kommt aus der Kombination. Und die letzte Spalte bleibt ehrlich: Auch 2 289 liegt noch 29,5 % über der Schranke. Ob dort wirklich so viel Luft ist oder ob die Schranke nur schwach ist, weiß man nicht — das ist die Lage, in der man mit einer Metaheuristik arbeitet.

9.8 Was dieses Kapitel nicht behandelt

Damit die Landkarte vollständig ist — drei Verfahren, die in dieselbe Familie gehören und hier bewusst nur benannt werden:

Verfahren	Idee in einem Satz	Wann anschauen
Genetische Algorithmen	Eine Population von Lösungen wird gekreuzt und mutiert.	Wenn es eine natürliche „Kreuzung“ zweier Lösungen gibt. Für Reihenfolgen ist genau das der schwierige Teil. Bibliothek: `pymoo`.
Tabu-Suche	Bergsteigen, das zuletzt gemachte Züge eine Zeit lang verbietet.	Wenn die Suche zwischen zwei Zuständen pendelt. Oft mit weniger Kalibrierung als Annealing.
Spaltengenerierung	Nicht die Lösung heuristisch machen, sondern das **Modell** zerlegen: Ein Master-Problem fragt bei einem Teilproblem nach neuen, lohnenden Spalten.	Bei Zuschnitt- und Dienstplanproblemen mit sehr vielen gleichartigen Mustern. **Das folgende Kapitel** führt es aus: Kapitel 10.

Die Spaltengenerierung ist die interessanteste davon, weil sie eine **andere Antwort auf dieselbe Frage** gibt: Statt die Lösung zu approximieren, formuliert sie das Modell so um, dass exaktes Lösen wieder möglich wird. Sie hat eine eigene Voraussetzung — das Problem muss sich in ein Master- und ein Teilproblem zerlegen lassen — und ist deshalb kein Ersatz für LNS, sondern eine Alternative für eine engere Problemklasse.

Genau deshalb bekommt sie ein eigenes Kapitel: Kapitel 10. Es zeigt nebenbei, warum First-Fit bei manchen Zuschnittproblemen bereits optimal ist — eine Frage, die dieses Kapitel offenlässt.

9.9 Übungsaufgaben

Lösungen: Abschnitt A.9.

Aufgabe 9.1 □ — Die Kurzsichtigkeit von Hand. Rechnen Sie im Schnellstart nach, was die Faustregel kostet, wenn sie nicht bei *Weiß*, sondern bei *Schwarz* startet. Ist sie dort besser oder schlechter — und warum?

Aufgabe 9.2 □ — Zuggröße und Temperatur. `Simulated_Annealing.py` misst den Median der Zuggröße mit 70 Minuten. Bei welcher Starttemperatur würde eine Verschlechterung von genau 70 Minuten in 37 % der Fälle angenommen? Vergleichen Sie das mit den Temperaturen in der Tabelle.

Aufgabe 9.3 □□ — Die teure Bewertung. Ersetzen Sie in `Simulated_Annealing.py` die Funktion `delta_verschieben` durch eine Variante, die den Zug ausführt und `gesamtruestzeit()` neu berechnet.

Wie viele Züge schafft das Programm dann in derselben Zeit, und wie verändert sich das Ergebnis bei gleichem Zeitbudget?

Aufgabe 9.4 □□ — Der Umschlagpunkt genauer. `Metaheuristik_vs_Exakt.py` misst drei Größen. Zwischen 200 und 500 liegt der Umschlagpunkt. Grenzen Sie ihn mit weiteren Größen auf ±50 Aufträge ein. Ist er scharf oder verwaschen?

Aufgabe 9.5 □□□ — Zerstören nach Maß. `Large_Neighborhood_Search.py` bricht ein *zusammenhängendes* Fenster heraus. Bauen Sie eine zweite Zerstörungsstrategie: Entfernen Sie die 20 Aufträge mit den **teuersten** Übergängen, egal wo sie stehen, und setzen Sie sie exakt neu ein. Welche Strategie ist besser — und liegt es an der Strategie oder an der Zahl der Runden, die sie schafft?

9.10 Finde den Denkfehler

□ „Unsere Heuristik ist 6 % besser“

Ein Kollege stellt das Ergebnis seiner Optimierung vor:

> *„Die Disposition hat bisher nach Faustregel geplant: 2 497 Minuten Rüstzeit. Unser > Simulated Annealing kommt auf 2 343 — **6,2 % Ersparnis**, jede Nacht, vollautomatisch. > Bei 250 Produktionstagen sind das über 640 Stunden Maschinenzeit im Jahr.“*

Die Rechnung stimmt. Das Programm ist korrekt. Die 2 343 Minuten sind echt, die Ersparnis ist echt, und das Projekt wird genehmigt.

Trotzdem ist die Darstellung irreführend. Woran liegt es?

Ein Hinweis: Die Zahl, die fehlt, steht in der Ausgabe von `Metaheuristik_vs_Exakt.py` — in einer Spalte, die mit der Lösung des Problems nichts zu tun hat.

9.11 Micro-Quiz

□ Drei Fragen

1. Warum ist die Kostenänderung in $O(1)$ so wichtig? a) Weil die volle Summe numerisch ungenau wird. b) Weil eine lokale Suche von der Zahl geprüfter Züge lebt und die volle Summe sie um Größenordnungen senkt. c) Weil Simulated Annealing sonst mathematisch nicht konvergiert.

2. Bei $T_0 = 8$ werden 0,29 % der Verschlechterungen angenommen, und das Ergebnis ist unbrauchbar. Warum genügen 0,29 %? a) Weil die Annahmewahrscheinlichkeit im Lauf noch steigt. b) Weil die angenommenen Verschlechterungen sich aufsummieren und die Suche weit von der guten Startlösung wegtragen, bevor die Temperatur fällt. c) Weil das Metropolis-Kriterium bei kleinen Quoten instabil wird.

3. Bei 500 Aufträgen liefert CP-SAT eine schlechtere Lösung als die Faustregel. Was ist der Lauf trotzdem wert? a) Nichts — abgebrochene Läufe sind wertlos. b) Er liefert die

untere Schranke und damit die einzige Aussage darüber, wie viel Luft noch nach oben ist. c) Er liefert eine gute Startlösung für das Annealing.

9.12 Selbsttest

1. Erklären Sie in eigenen Worten, warum eine Faustregel „kurzsichtig“ ist — und warum das nicht dasselbe ist wie „schlecht“.
2. Warum lässt sich 2-opt bei **asymmetrischen** Rüstzeiten nicht in $O(1)$ bewerten?
3. Sie setzen eine Metaheuristik ein und berichten „12 % Verbesserung“. Welche zweite Zahl müssen Sie mitliefern, damit die Aussage belastbar ist, und woher bekommen Sie sie?
4. Ihr LNS verbessert kaum noch. Nennen Sie zwei Ursachen, die man an der Ausgabe unterscheiden kann.
5. Warum ist der Umschlagpunkt zwischen exakt und heuristisch keine allgemeine Zahl?

9.13 Zusammenfassung

- Eine **Faustregel** ist selten dumm, aber immer kurzsichtig: Sie bewertet einen Schritt danach, was er kostet, nicht danach, was er übrig lässt.
- **Lokale Suche** steht und fällt mit der Kostenänderung in $O(1)$. Wer den Zug über die volle Zielfunktion bewertet, misst am Ende seine Implementierung statt des Verfahrens.
- **Simulated Annealing** nimmt Verschlechterungen mit $e^{-\Delta/T}$ an. T und Δ haben dieselbe Einheit — die Temperatur gehört an der **Zuggröße** kalibriert, nicht am Zielfunktionswert. Die verbreitete Regel „20 bis 50 % Annahmequote“ war auf dieser Aufgabe falsch.
- Der **schlechteste Zwischenwert** ist die beste Diagnosezahl eines Annealing-Laufs: Er zeigt eine zu heiße Starttemperatur sofort, ohne dass man das Ergebnis abwarten muss.
- Rechnen Sie **immer** das simple Bergsteigen mit. Hier lieferte es 93 % des Ertrags.
- Der **Umschlagpunkt** zwischen exakt und heuristisch ist problemspezifisch und messbar. Hier liegt er zwischen 200 und 500 Aufträgen — bei 60 wäre die Metaheuristik die schlechtere Wahl gewesen.
- Ein exakter Lauf lohnt sich auch dann, wenn Sie seine **Lösung** verwerfen: Seine **Schranke** ist die einzige Aussage darüber, wie gut Ihre heuristische Lösung sein könnte.
- **LNS** kombiniert beide: zerstören, exakt reparieren, übernehmen. Die Fenstergröße hat ein Optimum in der Mitte und gehört gemessen.

Kapitel 10: Spaltengenerierung — das Modell umbauen statt die Lösung raten

Kapitel auf einen Blick

Worum geht es? Um die andere Antwort auf die Frage aus Kapitel 9: Was tun, wenn der Solver an der Größe scheitert? Statt die Lösung zu approximieren, formuliert man das Modell so um, dass exaktes Lösen wieder möglich wird.

Voraussetzungen: Abschnitt 5.6 (Dualwerte) und Kapitel 6 (`Rucksack.py`). Beides wird gebraucht, aber nicht neu erklärt.

Danach können Sie: ein Problem in Master und Teilproblem zerlegen; die Spaltengenerierungsschleife selbst schreiben; und begründen, wann sich der Aufwand lohnt — die Antwort hängt an einer einzigen Kennzahl der Instanz.

Zeitbedarf: ca. 3 Stunden.

Programme:
`Spaltengenerierung.py`

Notebook: `Notebooks_04/dekomposition.ipynb`

10.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Denken Sie in Mustern, nicht in Stücken

Eine Rolle ist 1 000 mm breit. Bestellt sind 9 Stück à 420 mm, 12 à 310 mm und 14 à 250 mm. Wie viele Rollen braucht man?

Die naheliegende Frage — *„welches Stück kommt auf welche Rolle?“* — führt zu 35 Einzelentscheidungen. Die bessere Frage lautet: *„Wie oft schneide ich welches Muster?“*

```
import itertools
import numpy as np
from scipy.optimize import linprog

ROLLE = 1000                                    # mm Mutterrolle
breiten, bedarf = [420, 310, 250], [9, 12, 14]

# Alle Schnittmuster aufzaehlen, die in die Rolle passen
muster = [m for m in itertools.product(*[range(ROLLE // b + 1) for b in breiten])
          if sum(a * b for a, b in zip(m, breiten)) <= ROLLE and sum(m) > 0]

# Wie oft schneide ich welches Muster?  min sum(x)  u.d.N.  A x >= bedarf
loesung = linprog(np.ones(len(muster)), A_ub=-np.array(muster, float).T,
                  b_ub=-np.array(bedarf, float), bounds=(0, None),
                  integrality=1, method="highs")
```

```
print(f"{len(muster)} zulaessige Muster, {int(round(loesung.fun))} Rollen
 ↪  noetig:")
for anzahl, m in zip(np.round(loesung.x).astype(int), muster):
    if anzahl:
        rest = ROLLE - sum(a * b for a, b in zip(m, breiten))
        print(f"  {anzahl:2d} x  " + "  ".join(f"{a}x{b}mm" for a, b in zip(m,
         ↪  breiten))
              + f"   Verschnitt {rest} mm")
```

Ausgabe:

```
16 zulaessige Muster, 12 Rollen noetig:
   2 x  0x420mm  0x310mm  4x250mm   Verschnitt 0 mm
   1 x  0x420mm  3x310mm  0x250mm   Verschnitt 70 mm
   9 x  1x420mm  1x310mm  1x250mm   Verschnitt 20 mm
```

Und jetzt der Punkt. Sehen Sie sich an, was da herauskommt: **drei Schnittmuster**. Kein Zuordnungsplan für 35 Stücke, sondern eine Anweisung, die an der Maschine hängen kann — neunmal dieses Muster, zweimal jenes, einmal das dritte.

Das ist kein kosmetischer Unterschied. Das Modell hat **eine Variable je Muster** statt einer je Stück-und-Rolle, und es kennt gar keine einzelnen Rollen mehr. Damit verschwindet ein Problem, das das naheliegende Modell praktisch unlösbar macht — dazu gleich mehr.

 Merksatz Die schwierigste Arbeit an einem Optimierungsmodell ist selten die Zielfunktion. Es ist die Frage, **worüber** die Variablen laufen.

Warum funktioniert das? Weil hier alle 16 möglichen Muster aufzählbar waren. Bei drei Breiten und einer 1 000-mm-Rolle sind es 16; bei dreizehn Breiten und 5 600 mm sind es zehntausende, und bei einer echten Papierfabrik mehr, als sich speichern lässt. Das ganze Kapitel handelt davon, wie man mit Mustern rechnet, ohne sie aufzuschreiben.

10.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, warum das naheliegende Zuordnungsmodell an **Symmetrie** scheitert.
2. … ein Problem in **Master** und **Pricing-Teilproblem** zerlegen.
3. … die Schleife schreiben: Master-LP lösen, Dualwerte auslesen, Teilproblem lösen, Spalte hinzufügen.
4. … das Abbruchkriterium über die **reduzierten Kosten** begründen.
5. … an einer Kennzahl der Instanz abschätzen, ob sich das Verfahren lohnt.

10.3 Warum das naheliegende Modell scheitert

Der erste Entwurf eines Zuschnittmodells sieht fast immer so aus: eine Binärvariable z_{jr} für „Stück j liegt auf Rolle r“, dazu y_r für „Rolle r wird benutzt”.

Zwei Dinge gehen dabei schief.

Die Größe. Bei 219 Zuschnitten und höchstens 82 Rollen sind das knapp 18 000 Binärvariablen — unangenehm, aber allein noch kein Hindernis.

Die Symmetrie. Das ist das eigentliche Problem. Alle Mutterrollen sind **gleich**. Jede Lösung existiert deshalb in unzähligen Umbenennungen: Vertauscht man Rolle 3 und Rolle 47, entsteht eine formal andere, inhaltlich identische Lösung. Branch-and-Bound weiß das nicht und arbeitet sie einzeln ab.

> **Woran man ein Symmetrieproblem erkennt**
>
> Der Suchbaum wächst, aber die **Schranke bewegt sich nicht.** Im Protokoll aus Abschnitt 6.8 sieht das so aus: Die Zahl der Knoten steigt in die Hunderttausende, der Incumbent verbessert sich hin und wieder, und der Dual Bound steht praktisch still.
>
> Das ist ein anderes Bild als „das Problem ist einfach zu groß”. Bei einem großen, aber unsymmetrischen Problem nähern sich beide Werte einander an, nur langsam.

Das Mustermodell hat dieses Problem nicht, weil es **keine einzelnen Rollen kennt**. Es zählt nur, wie oft welches Schnittmuster geschnitten wird:

$$\min \sum_{p \in P} x_p \qquad \text{unter} \qquad \sum_{p \in P} a_{ip}\, x_p \geq d_i \quad \forall i, \qquad x_p \geq 0 \text{ ganzzahlig}$$

> **Formel-Übersetzer**

Mathematik	Alltagssprache
P	die Menge aller zulässigen Schnittmuster
a_{ip}	„Wie viele Stücke der Breite i liefert Muster p?”
x_p	„Wie oft schneide ich Muster p?” — die einzige Entscheidung
d_i	die bestellte Stückzahl der Breite i
$\sum_p x_p$	die Zahl der verbrauchten Mutterrollen

> **Ohne Formel gesagt:** „Stelle eine Schnittliste zusammen, die alle Bestellungen deckt, und benutze dabei so wenige Rollen wie möglich.”
>
> Das Modell ist verblüffend klein — eine Zeile je Breite. Sein ganzes Gewicht steckt in P, und das ist der Haken.

10.4 Rechnen mit Mustern, ohne sie aufzuschreiben

Die Idee ist alt (Gilmore und Gomory, 1961) und in einem Satz gesagt:

> Fange mit wenigen Mustern an. Frage nach jeder Lösung, ob es ein **noch nicht bekanntes** Muster gibt, das sich lohnen würde — und wenn ja, nimm nur dieses eine dazu.

Die Frage „lohnt sich noch ein Muster?" beantworten die **Dualwerte** des Master-LPs. Der Dualwert π_i zur Bedarfszeile i ist genau der Schattenpreis aus Abschnitt 5.6: Er sagt, wie viele Rollen ein zusätzliches Stück der Breite i kostet.

Ein neues Muster a lohnt sich, wenn seine **reduzierten Kosten** negativ sind:

$$1 - \sum_i \pi_i \, a_i < 0 \qquad \Longleftrightarrow \qquad \sum_i \pi_i \, a_i > 1$$

Formel-Übersetzer

Mathematik	Alltagssprache
die 1	„Ein zusätzliches Muster zu schneiden kostet eine Rolle."
$\sum_i \pi_i a_i$	„So viel ist das, was dabei herauskommt, zu den aktuellen Schattenpreisen wert."
> 1	„Es bringt mehr, als es kostet — dieses Muster nehmen wir dazu."
≤ 1	„Kein Muster lohnt sich mehr. **Fertig** — und zwar beweisbar."

> Die letzte Zeile ist der Grund, warum das Verfahren **exakt** ist und nicht heuristisch: Wenn kein Muster mehr lohnende reduzierte Kosten hat, ist die LP-Lösung über *allen* Mustern optimal — auch über den nie erzeugten.

Und wie findet man das beste neue Muster? Man sucht die Zusammenstellung von Stücken mit dem größten Gesamtwert, die noch in die Rolle passt. Das ist ein **Rucksackproblem**:

Rucksack (Kapitel 6)	hier
Nutzen eines Gegenstands	Schattenpreis π_i der Breite
Gewicht	Breite in mm
Kapazität des Rucksacks	Rollenbreite

Das Teilproblem ist also ein Verfahren, das die Leser dieses Buchs schon kennen. **Neu ist allein die Schleife.**

10.5 Das Programm

```
#!/usr/bin/env python3

# Spaltengenerierung.py
"""
Kapitel Dekomposition: Das Modell umbauen, statt die Loesung zu raten.

Eine Papierfabrik liefert Rollen von 5.600 mm Breite. Die Kunden bestellen
schmalere Breiten, und aus jeder Mutterrolle werden mehrere davon geschnitten.
Gesucht ist der Schnittplan, der mit den wenigsten Mutterrollen auskommt.

Der naheliegende Modellansatz - "welches Stueck kommt auf welche Rolle?" -
fuehrt in eine Sackgasse: Er braucht eine Binaervariable je Stueck-und-Rolle,
und weil alle Rollen gleich sind, ist er hochgradig symmetrisch. Der Solver
probiert dieselbe Loesung in tausend Umbenennungen durch.

Die Spaltengenerierung dreht die Frage um:

    Nicht "welches Stueck auf welche Rolle",
    sondern "wie oft schneide ich welches MUSTER".

Damit verschwindet die Symmetrie - aber es entsteht ein neues Problem: Die Zahl
der moeglichen Muster waechst kombinatorisch. Man kann sie nicht aufschreiben.
Der Trick besteht darin, sie auch nicht aufzuschreiben, sondern nur die wenigen
zu erzeugen, die tatsaechlich gebraucht werden - und zwar mit Hilfe der
Schattenpreise aus dem Kapitel LP.

Das Programm zeigt vier Dinge:

  1. Die beiden Formulierungen desselben Problems und ihre Groesse.
  2. Wie schnell die Musterzahl waechst - und wie wenige davon genuegen.
  3. Die Schleife selbst: Master-LP, Dualwerte, Rucksack, neue Spalte.
  4. Wann sich der Aufwand lohnt und wann nicht. Die Antwort haengt an einer
     einzigen Kennzahl der Instanz, und sie erklaert nebenbei, warum eine
     Faustregel bei manchen Zuschnittproblemen schon optimal ist.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
from scipy.optimize import linprog

ROLLENBREITE = 5600

# Zwei Instanzen mit demselben Bedarf, aber verschiedenen Breiten. Der
```

```
# Unterschied ist die eine Kennzahl, um die es in Teil 4 geht.
BEDARF = [22, 25, 12, 14, 18, 18, 20, 10, 12, 14, 16, 18, 20]
BREIT = [1380, 1520, 1560, 1710, 1820, 1880, 1930, 2000, 2050, 2100, 2140,
         2150, 2200]
SCHMAL = [380, 520, 560, 710, 820, 880, 930, 1000, 1050, 1100, 1140,
          1150, 1200]


# --- Die Musterzahl ----------------------------------------------------------

def zaehle_muster(breiten, rollenbreite: int = ROLLENBREITE) -> int:
    """Wie viele zulaessige Schnittmuster gibt es ueberhaupt?

    Nur zum Zeigen - in einem produktiven Programm wuerde man das nie
    ausrechnen, weil die Zahl bei realistischen Instanzen jede Vorstellung
    sprengt. Genau das ist der Punkt.
    """
    speicher: dict[tuple[int, int], int] = {}

    def ab(i: int, rest: int) -> int:
        if i == len(breiten):
            return 1
        if (i, rest) in speicher:
            return speicher[(i, rest)]
        summe = sum(ab(i + 1, rest - anzahl * breiten[i])
                    for anzahl in range(rest // breiten[i] + 1))
        speicher[(i, rest)] = summe
        return summe

    return ab(0, rollenbreite)


# --- Die beiden Bausteine der Schleife ---------------------------------------

def loese_master(muster, bedarf, ganzzahlig: bool = False):
    """Restringiertes Master: Wie oft schneide ich jedes bekannte Muster?

        min  sum_p x_p           unter   sum_p a_ip * x_p >= bedarf_i

    Als LP gerechnet liefert es zusaetzlich die DUALWERTE - und die sind der
    eigentliche Ertrag: Der Dualwert zu Breite i sagt, was ein zusaetzliches
    Stueck dieser Breite an Rollen kostet. Es ist derselbe Schattenpreis wie
    im Kapitel LP, nur dass er hier nicht berichtet, sondern weiterverarbeitet
    wird.
    """
    matrix = -np.array(muster, dtype=float).T        # >= wird zu <= mit Minus
    ergebnis = linprog(np.ones(len(muster)),
                       A_ub=matrix, b_ub=-np.array(bedarf, dtype=float),
```

```
                      bounds=(0, None),
                      integrality=(1 if ganzzahlig else 0), method="highs")
    if not ergebnis.success:
        raise SystemExit(f"Master nicht loesbar: {ergebnis.message}")
    dual = None if ganzzahlig else -ergebnis.ineqlin.marginals
    return ergebnis.fun, ergebnis.x, dual


def bestes_neues_muster(dual, breiten, rollenbreite: int = ROLLENBREITE):
    """Pricing: Gibt es ein Muster, das sich noch lohnt?

    Gesucht ist das Muster mit dem groessten Gesamtwert zu den aktuellen
    Schattenpreisen - unter der Bedingung, dass es in die Rolle passt. Das ist
    ein RUCKSACKPROBLEM (Kapitel MILP, Rucksack.py): Preise sind der Nutzen,
    Breiten das Gewicht, die Rollenbreite die Kapazitaet. Hier mit dynamischer
    Programmierung geloest, weil die Breiten ganzzahlig sind.

    Bei Gleichstand wird bewusst der KLEINSTE Index bevorzugt (>-Vergleich mit
    Toleranz statt >=). Sonst haengt das erzeugte Muster von der Reihenfolge
    der Gleitkommaoperationen ab, und das Programm liefert von Lauf zu Lauf
    verschiedene Ausgaben.
    """
    wert = np.zeros(rollenbreite + 1)
    herkunft = [-1] * (rollenbreite + 1)
    for platz in range(rollenbreite + 1):
        for i, breite in enumerate(breiten):
            if breite <= platz and wert[platz - breite] + dual[i] > wert[platz] + 1e-9:
                wert[platz] = wert[platz - breite] + dual[i]
                herkunft[platz] = i

    muster = [0] * len(breiten)
    platz = rollenbreite
    while herkunft[platz] >= 0:
        i = herkunft[platz]
        muster[i] += 1
        platz -= breiten[i]
    return float(wert[rollenbreite]), muster


def spaltengenerierung(breiten, bedarf, rollenbreite: int = ROLLENBREITE):
    """Die Schleife. Startbasis: je ein Muster mit nur einer Breite.

    Abbruch, wenn kein Muster mehr einen Wert ueber 1 hat: Eine zusaetzliche
    Rolle kostet 1, also lohnt sich ein neues Muster nur, wenn es zu den
    aktuellen Preisen mehr als 1 wert ist. Das ist das Kriterium der
    reduzierten Kosten.
    """
    anzahl = len(breiten)
```

```
    muster = [[0] * anzahl for _ in range(anzahl)]
    for i in range(anzahl):
        muster[i][i] = rollenbreite // breiten[i]

    runden = 0
    while True:
        zielwert, _, dual = loese_master(muster, bedarf)
        wert, neu = bestes_neues_muster(dual, breiten, rollenbreite)
        runden += 1
        if wert <= 1 + 1e-6:
            return muster, zielwert, runden
        muster.append(neu)


def first_fit(breiten, bedarf, rollenbreite: int = ROLLENBREITE) -> int:
    """Die Faustregel: groesstes Stueck zuerst, auf die erste passende Rolle."""
    stuecke = sorted([b for b, menge in zip(breiten, bedarf)
                      for _ in range(menge)], reverse=True)
    rollen: list[list[int]] = []
    for stueck in stuecke:
        for rolle in rollen:
            if sum(rolle) + stueck <= rollenbreite:
                rolle.append(stueck)
                break
        else:
            rollen.append([stueck])
    return len(rollen)


def auswerten(name: str, breiten, bedarf) -> dict:
    muster, schranke, runden = spaltengenerierung(breiten, bedarf)
    ganz, _, _ = loese_master(muster, bedarf, ganzzahlig=True)
    return {"name": name, "muster_gesamt": zaehle_muster(breiten),
            "muster_erzeugt": len(muster), "runden": runden,
            "schranke": schranke, "ganzzahlig": int(round(ganz)),
            "first_fit": first_fit(breiten, bedarf),
            "stuecke_je_rolle": ROLLENBREITE / np.mean(breiten)}


if __name__ == "__main__":
    print("=" * 84)
    print("  SPALTENGENERIERUNG: NUR DIE MUSTER ERZEUGEN, DIE MAN BRAUCHT")
    print("=" * 84)
    print(f"Mutterrolle {ROLLENBREITE:,} mm, {len(BREIT)} bestellte Breiten, "
          f"{sum(BEDARF)} Zuschnitte.\n")

    # --- 1. Zwei Formulierungen ----------------------------------------
    stuecke = sum(BEDARF)
```

```
    # Wie viele Rollen braucht man hoechstens? Die Faustregel liefert eine
    # brauchbare Obergrenze - mehr Rollen als das wird niemand benoetigen.
    rollen_obergrenze = first_fit(BREIT, BEDARF)
    print("1. Zwei Modelle fuer dieselbe Aufgabe\n")
    print(f"  {'Formulierung':<34} {'Binaervariablen':>16} {'symmetrisch?':>16}")
    print("  " + "-" * 70)
    print(f"  {'Stueck -> Rolle (naheliegend)':<34} "
          f"{stuecke * rollen_obergrenze:>16,} {'ja':>16}")
    print(f"  {'wie oft welches Muster':<34} "
          f"{'eine je Muster':>16} {'nein':>16}")
    print(f"\n  Der naheliegende Ansatz braucht eine Variable je Stueck und Rolle:")
    print(f"  {stuecke} Zuschnitte x {rollen_obergrenze} Rollen (Obergrenze aus der
     ↪  Faustregel)")
    print(f"  = {stuecke * rollen_obergrenze:,} Binaervariablen.")
    print("\n  Schlimmer als die Zahl ist die Symmetrie: Alle Mutterrollen sind")
    print("  gleich, also beschreibt jede Loesung dieselbe Schnittvorschrift in")
    print("  unzaehligen Umbenennungen. Branch-and-Bound probiert sie einzeln")
    print("  durch und kommt nicht voran - der Suchbaum waechst, ohne dass sich")
    print("  die Schranke bewegt.")
    print("\n  Das Mustermodell hat je Muster genau eine Variable und kennt keine")
    print("  einzelnen Rollen mehr. Damit ist die Symmetrie weg. Sein Problem ist")
    print("  ein anderes - und zwar das folgende.")

    # --- 2. Die Musterzahl -------------------------------------------------
    print("\n" + "-" * 84)
    print("2. Warum man die Muster nicht aufschreiben kann\n")
    gross = auswerten("breite Zuschnitte (1.380-2.200 mm)", BREIT, BEDARF)
    schmal = auswerten("schmale Zuschnitte (380-1.200 mm)", SCHMAL, BEDARF)

    print(f"  {'Instanz':<36} {'Muster':>12} {'davon erzeugt':>14} {'Anteil':>9}")
    print("  " + "-" * 76)
    for fall in (gross, schmal):
        anteil = fall["muster_erzeugt"] / fall["muster_gesamt"]
        print(f"  {fall['name']:<36} {fall['muster_gesamt']:>12,} "
              f"{fall['muster_erzeugt']:>14} {anteil:>8.1%}")

    print(f"\n  Schon das Halbieren der Breiten laesst die Musterzahl von "
          f"{gross['muster_gesamt']:,} auf")
    print(f"  {schmal['muster_gesamt']:,} springen - Faktor "
          f"{schmal['muster_gesamt'] / gross['muster_gesamt']:.0f}. Bei einer echten "
          f"Papierfabrik mit")
    print("  vierzig Breiten und Millimeterschritten sind es mehr, als sich")
    print("  speichern liesse.")
    print(f"\n  Gebraucht werden davon {schmal['muster_erzeugt']} - "
          f"{schmal['muster_erzeugt'] / schmal['muster_gesamt']:.2%} der Gesamtzahl.")

    # --- 3. Die Schleife ---------------------------------------------------
    print("\n" + "-" * 84)
```

```
print("3. Die Schleife an der breiten Instanz\n")
print(f"  Startbasis: {len(BREIT)} triviale Muster (je Rolle nur eine Breite)")
print(f"  Runden bis kein Muster mehr lohnt: {gross['runden']}")
print(f"  Muster am Ende: {gross['muster_erzeugt']}")
print(f"  LP-Schranke: {gross['schranke']:.4f} Rollen")
print(f"\n  Die LP-Schranke ist eine ZUSAGE: Weniger als "
      f"{np.ceil(gross['schranke'] - 1e-9):.0f} Rollen sind")
print("  nicht moeglich - unabhaengig davon, wie clever man weiterschneidet.")
print("  Genau diese Aussage fehlt einer Heuristik (Kapitel Metaheuristiken).")

# --- 4. Wann es sich lohnt ------------------------------------------
print("\n" + "-" * 84)
print("4. Wann sich der Aufwand lohnt - und wann nicht\n")
print(f"  {'Instanz':<36} {'Faustregel':>11} {'exakt':>8} {'Schranke':>10} "
      f"{'Ersparnis':>11}")
print("  " + "-" * 80)
for fall in (gross, schmal):
    ersparnis = (fall["first_fit"] - fall["ganzzahlig"]) / fall["first_fit"]
    print(f"  {fall['name']:<36} {fall['first_fit']:>11} "
          f"{fall['ganzzahlig']:>8} {fall['schranke']:>10.2f} "
          f"{ersparnis:>10.0%}")

print(f"\n  Bei den breiten Zuschnitten spart die Spaltengenerierung "
      f"{gross['first_fit'] - gross['ganzzahlig']} von")
print(f"  {gross['first_fit']} Rollen. Bei den schmalen spart sie NICHTS - dort ist die")
print("  Faustregel bereits optimal.")
print(f"\n  Der Unterschied haengt an einer einzigen Kennzahl:\n")
print(f"  {'Instanz':<36} {'Stuecke je Rolle (etwa)':>24}")
print("  " + "-" * 62)
for fall in (gross, schmal):
    print(f"  {fall['name']:<36} {fall['stuecke_je_rolle']:>24.1f}")

print("\n  Passen nur zwei bis drei Stuecke auf eine Rolle, entscheidet jede")
print("  einzelne Zuordnung viel, und eine kurzsichtige Regel verschenkt")
print("  ganze Rollen. Passen sechs oder mehr darauf, gleichen sich die")
print("  Fehler aus - die Reste sind klein gegen die Rollenbreite, und die")
print("  Faustregel trifft es fast immer.")

print("\n" + "=" * 84)
print("  WAS MAN DARAUS MITNIMMT")
print("=" * 84)
print("1. Der Perspektivwechsel ist die eigentliche Arbeit: nicht 'welches")
print("   Stueck wohin', sondern 'wie oft welches Muster'. Damit verschwindet")
print("   die Symmetrie, die das naheliegende Modell unloesbar macht.")
print("2. Der Preis dafuer ist eine unaufschreibbare Zahl von Variablen. Die")
print("   Spaltengenerierung zahlt ihn nicht, sondern erzeugt nur die wenigen")
print("   Spalten, die die Dualwerte als lohnend ausweisen.")
print("3. Das Teilproblem ist ein Rucksack - ein Verfahren, das die Leser")
```

```
    print("   dieses Buchs schon kennen. Neu ist allein die Schleife.")
    print("4. Und die unbequeme Erkenntnis: Ob sich das alles lohnt, entscheidet")
    print("   die Instanz, nicht die Methode. Bei sechs Stuecken je Rolle ist die")
    print("   Faustregel so gut wie das Optimum - und der Ertrag der")
    print("   Spaltengenerierung liegt dann allein in der SCHRANKE, die beweist,")
    print("   dass man aufhoeren kann zu suchen.")
    print("=" * 84)
```

Erwartete Ausgabe:

```
====================================================================================
  SPALTENGENERIERUNG: NUR DIE MUSTER ERZEUGEN, DIE MAN BRAUCHT
====================================================================================
Mutterrolle 5,600 mm, 13 bestellte Breiten, 219 Zuschnitte.

1. Zwei Modelle fuer dieselbe Aufgabe

  Formulierung                        Binaervariablen     symmetrisch?
  ------------------------------------------------------------------
  Stueck -> Rolle (naheliegend)                17,958              ja
  wie oft welches Muster               eine je Muster            nein

  Der naheliegende Ansatz braucht eine Variable je Stueck und Rolle:
  219 Zuschnitte x 82 Rollen (Obergrenze aus der Faustregel)
  = 17,958 Binaervariablen.

  Schlimmer als die Zahl ist die Symmetrie: Alle Mutterrollen sind
  gleich, also beschreibt jede Loesung dieselbe Schnittvorschrift in
  unzaehligen Umbenennungen. Branch-and-Bound probiert sie einzeln
  durch und kommt nicht voran - der Suchbaum waechst, ohne dass sich
  die Schranke bewegt.

  Das Mustermodell hat je Muster genau eine Variable und kennt keine
  einzelnen Rollen mehr. Damit ist die Symmetrie weg. Sein Problem ist
  ein anderes - und zwar das folgende.

------------------------------------------------------------------------------------
2. Warum man die Muster nicht aufschreiben kann

  Instanz                                  Muster  davon erzeugt    Anteil
  ------------------------------------------------------------------------
  breite Zuschnitte (1.380-2.200 mm)          309             38    12.3%
  schmale Zuschnitte (380-1.200 mm)        46,408             28     0.1%

  Schon das Halbieren der Breiten laesst die Musterzahl von 309 auf
  46,408 springen - Faktor 150. Bei einer echten Papierfabrik mit
  vierzig Breiten und Millimeterschritten sind es mehr, als sich
  speichern liesse.
```

```
  Gebraucht werden davon 28 - 0.06% der Gesamtzahl.

--------------------------------------------------------------------------------
3. Die Schleife an der breiten Instanz

  Startbasis: 13 triviale Muster (je Rolle nur eine Breite)
  Runden bis kein Muster mehr lohnt: 26
  Muster am Ende: 38
  LP-Schranke: 72.9167 Rollen

  Die LP-Schranke ist eine ZUSAGE: Weniger als 73 Rollen sind
  nicht moeglich - unabhaengig davon, wie clever man weiterschneidet.
  Genau diese Aussage fehlt einer Heuristik (Kapitel Metaheuristiken).

--------------------------------------------------------------------------------
4. Wann sich der Aufwand lohnt - und wann nicht

  Instanz                              Faustregel    exakt   Schranke   Ersparnis
  ------------------------------------------------------------------------------
  breite Zuschnitte (1.380-2.200 mm)           82       73      72.92        11%
  schmale Zuschnitte (380-1.200 mm)            35       35      33.60         0%

  Bei den breiten Zuschnitten spart die Spaltengenerierung 9 von
  82 Rollen. Bei den schmalen spart sie NICHTS - dort ist die
  Faustregel bereits optimal.

  Der Unterschied haengt an einer einzigen Kennzahl:

  Instanz                              Stuecke je Rolle (etwa)
  ------------------------------------------------------------
  breite Zuschnitte (1.380-2.200 mm)                      3.0
  schmale Zuschnitte (380-1.200 mm)                       6.4

  Passen nur zwei bis drei Stuecke auf eine Rolle, entscheidet jede
  einzelne Zuordnung viel, und eine kurzsichtige Regel verschenkt
  ganze Rollen. Passen sechs oder mehr darauf, gleichen sich die
  Fehler aus - die Reste sind klein gegen die Rollenbreite, und die
  Faustregel trifft es fast immer.

================================================================================
  WAS MAN DARAUS MITNIMMT
================================================================================
1. Der Perspektivwechsel ist die eigentliche Arbeit: nicht 'welches
   Stueck wohin', sondern 'wie oft welches Muster'. Damit verschwindet
   die Symmetrie, die das naheliegende Modell unloesbar macht.
2. Der Preis dafuer ist eine unaufschreibbare Zahl von Variablen. Die
   Spaltengenerierung zahlt ihn nicht, sondern erzeugt nur die wenigen
   Spalten, die die Dualwerte als lohnend ausweisen.
```

```
3. Das Teilproblem ist ein Rucksack - ein Verfahren, das die Leser
   dieses Buchs schon kennen. Neu ist allein die Schleife.
4. Und die unbequeme Erkenntnis: Ob sich das alles lohnt, entscheidet
   die Instanz, nicht die Methode. Bei sechs Stuecken je Rolle ist die
   Faustregel so gut wie das Optimum - und der Ertrag der
   Spaltengenerierung liegt dann allein in der SCHRANKE, die beweist,
   dass man aufhoeren kann zu suchen.
================================================================================
```

10.6 Was die Zahlen zeigen

Die Musterzahl explodiert, der Bedarf nicht

Instanz	mögliche Muster	erzeugt	Anteil
breite Zuschnitte (1 380–2 200 mm)	309	38	12,3 %
schmale Zuschnitte (380–1 200 mm)	46 408	28	**0,06 %**

Schon das Halbieren der Breiten lässt die Musterzahl um den **Faktor 150** springen. Die Zahl der *gebrauchten* Muster bleibt dagegen konstant — sie steigt sogar nicht, sie sinkt leicht. Das ist der ganze Ertrag des Verfahrens: Der Aufwand hängt an der Zahl der **Bedarfszeilen**, nicht an der Zahl der Muster.

Der unbequeme Teil: Es lohnt sich nicht immer

Instanz	Faustregel	exakt	Schranke	Ersparnis
breite Zuschnitte	82 Rollen	**73**	72,92	**11 %**
schmale Zuschnitte	35 Rollen	35	33,60	**0 %**

Bei den breiten Zuschnitten spart die Spaltengenerierung neun von 82 Rollen. Bei den schmalen spart sie **nichts** — dort ist First-Fit-Decreasing bereits optimal.

Der Unterschied hängt an einer einzigen Kennzahl:

Instanz	Stücke je Rolle
breite Zuschnitte	3,0
schmale Zuschnitte	6,4

Passen nur zwei bis drei Stücke auf eine Rolle, entscheidet jede einzelne Zuordnung viel, und eine kurzsichtige Regel verschenkt ganze Rollen. **Passen sechs oder mehr darauf**, gleichen sich die Fehler aus: Die Reste sind klein gegen die Rollenbreite, und die Faustregel trifft es fast immer.

Das erklärt einen Befund aus Kapitel 9

Dort steht, dass Bin Packing als Aufhänger für Metaheuristiken **nicht** taugt, weil First-Fit in den geprüften Größen bereits optimal war. Der Grund ist jetzt benennbar: Die dortige Instanz hatte Stücke von 700 bis 2 600 mm bei 5 600 mm Rollenbreite — im Mittel gut drei je Rolle, aber mit viel kleineren Stücken durchsetzt, so dass die Reste sich auffüllen ließen.

Die brauchbare Faustregel lautet also nicht „Bin Packing ist einfach", sondern: **Je weniger Stücke auf einen Behälter passen, desto mehr ist mit exakter Optimierung zu holen.** Das ist eine Zahl, die man vor dem Projekt ausrechnen kann.

Und selbst dort, wo die Ersparnis null ist, liefert das Verfahren etwas, das die Faustregel nicht kann: die **Schranke 33,60**. Sie beweist, dass 34 Rollen das Minimum wären und 35 höchstens eine daneben liegen — also dass man aufhören kann zu suchen.

10.7 Übungsaufgaben

Lösungen: Abschnitt A.10.

Aufgabe 10.1 □ **— Das Abbruchkriterium.** Warum lautet die Schwelle beim Pricing genau `1` und nicht `0`? Woher kommt die Eins?

Aufgabe 10.2 □ **— Die Startbasis.** Das Programm startet mit Mustern, die je nur eine Breite enthalten. Warum ist das immer zulässig, und warum wäre eine leere Startmenge ein Problem?

Aufgabe 10.3 □□ **— Die Kennzahl prüfen.** Erzeugen Sie Instanzen mit 2, 4, 8 und 16 Stücken je Rolle und tragen Sie die Ersparnis gegenüber First-Fit auf. Bestätigt sich der Zusammenhang aus Abschnitt 10.6?

Aufgabe 10.4 □□ **— Dienstplanung.** Formulieren Sie die Wochendienstplanung als Spaltengenerierung: Ein „Muster" ist ein zulässiger Wochenplan **einer** Person. Was ist das Teilproblem, und welche Nebenbedingungen stehen im Master, welche im Teilproblem?

Aufgabe 10.5 □□□ **— Branch-and-Price.** Das ganzzahlige Master über die erzeugten Spalten ist nicht garantiert optimal — es könnte Muster geben, die erst *nach* einer Verzweigung lohnend werden. Recherchieren Sie, was Branch-and-Price daran ändert, und erklären Sie, warum man nicht einfach auf den erzeugten Spalten verzweigen kann.

10.8 Finde den Denkfehler

□ **„Die LP-Lösung sagt 72,92 — also runden wir auf"**

Ein Kollege hat die Spaltengenerierung implementiert und ist zufrieden:

> *„Das Master-LP liefert 72,92 Rollen, verteilt auf 13 Muster. Halbe Rollen kann man > nicht schneiden, also runde ich jedes Muster auf die nächste ganze Zahl auf. Damit sind > alle Bestellungen sicher gedeckt — aufrunden kann ja nur zu viel liefern, nie zu wenig. > Ergebnis: ein zulässiger Schnittplan, und die Schranke sagt mir, dass ich höchstens eine > Rolle daneben liege."*

Der erste Teil stimmt: Aufrunden liefert tatsächlich einen zulässigen Plan. Der zweite Satz über die Schranke ist der Fehler — und er ist teuer.

Wie viele Rollen kostet das Aufrunden hier wirklich, und was hätte der Kollege stattdessen tun müssen?

Ein Hinweis: Er hat alles, was er dafür braucht, bereits vorliegen.

10.9 Micro-Quiz

Drei Fragen

1. Warum ist das Mustermodell dem Zuordnungsmodell überlegen? a) Weil es weniger Nebenbedingungen hat. b) Weil es keine einzelnen Rollen kennt und damit die Symmetrie verschwindet. c) Weil Musterprobleme immer ganzzahlige LP-Lösungen haben.

2. Was ist das Pricing-Teilproblem beim Zuschnitt? a) Ein zweites LP über dieselben Variablen. b) Ein Rucksackproblem: Welche Stücke passen in eine Rolle und sind zu den aktuellen Schattenpreisen am meisten wert? c) Eine Heuristik, die neue Muster zufällig erzeugt.

3. Bei der schmalen Instanz spart die Spaltengenerierung null Rollen gegenüber First-Fit. War der Aufwand umsonst? a) Ja — wo nichts gespart wird, hat sich das Verfahren nicht gelohnt. b) Nein — sie liefert die untere Schranke und damit den Beweis, dass die Faustregel höchstens eine Rolle danebenliegt. c) Nein — sie wird bei der nächsten Instanz mehr sparen.

10.10 Selbsttest

1. Erklären Sie Symmetrie in einem MILP an einem Beispiel aus Ihrem Arbeitsumfeld.
2. Was ist beim Zuschnitt eine „Spalte”, und was steht darin?
3. Woher kommen die Preise, mit denen das Teilproblem rechnet?
4. Warum ist das Verfahren exakt, obwohl es fast alle Muster nie ansieht?
5. Sie sollen vor dem Projekt abschätzen, ob sich Spaltengenerierung lohnt. Welche eine Zahl rechnen Sie aus?

10.11 Zusammenfassung

- Die folgenreichste Entscheidung an einem Modell ist, **worüber die Variablen laufen**. Beim Zuschnitt schlägt „wie oft welches Muster" das naheliegende „welches Stück auf welche Rolle" — weil damit die **Symmetrie** verschwindet, an der Branch-and-Bound scheitert.
- Der Preis dafür ist eine Variablenmenge, die man nicht aufschreiben kann. Die **Spaltengenerierung** zahlt ihn nicht: Sie erzeugt nur die Muster, die die **Dualwerte** als lohnend ausweisen — im Beispiel 28 von 46 408.
- Das Abbruchkriterium sind die **reduzierten Kosten**: Solange ein Muster zu den aktuellen Schattenpreisen mehr als eine Rolle wert ist, lohnt es sich. Danach ist die Lösung beweisbar optimal — auch über die nie erzeugten Muster.
- Das **Teilproblem ist ein Rucksack**. Neu ist allein die Schleife.
- **Ob sich das lohnt, entscheidet die Instanz.** Bei drei Stücken je Rolle bringt das Verfahren 11 %, bei sechs nichts. Diese Kennzahl lässt sich vor dem Projekt ausrechnen.
- Auch wo es nichts spart, liefert es die **Schranke** — und damit die Erlaubnis, aufzuhören.

Synthese Teil II — Die Kernverfahren nebeneinander

Sechs Kapitel, sechs Werkzeuge. Jedes einzelne wurde im Zusammenhang gezeigt; hier stehen sie zum ersten Mal **nebeneinander**, mit der Frage, die vor jedem Projekt zu beantworten ist: Welches nehme ich, und was bekomme ich dafür?

Die Entscheidungsmatrix

Verfahren	Wofür es gebaut ist	Was Sie bekommen	Wo es aufhört	Kapitel
LP	teilbare Mengen, lineare Zusammenhänge	beweisbares Optimum **plus Schattenpreise**	sobald etwas ganzzahlig sein muss	Kapitel 5
MILP	Ja/Nein-Entscheidungen, Fixkosten, Logik	beweisbares Optimum, Gap als Fortschrittsmaß	Laufzeit wächst mit der Zahl der Binärvariablen	Kapitel 6
CP-SAT	Zuweisung, Reihenfolge, Kalender, harte Logik	beweisbares Optimum, sehr ausdrucksstarke Bedingungen	keine Schattenpreise, keine stetigen Größen	Kapitel 7
Graphenalgorithmen	Flüsse, Zuordnung, Touren	oft **polynomiell** statt exponentiell	nur, wenn die Struktur wirklich passt	Kapitel 8
Metaheuristiken	wenn der exakte Solver aussteigt	eine gute Lösung in **fester Zeit**	keine Garantie, kein Beweis, keine Schranke	Kapitel 9

Verfahren	Wofür es gebaut ist	Was Sie bekommen	Wo es aufhört	Kapitel
Spaltengenerierung	Modelle mit astronomisch vielen Variablen	beweisbares Optimum über nie erzeugte Spalten	lohnt nur bei bestimmter Instanzstruktur	Kapitel 10

Die Spalte, die am meisten wert ist, ist die vierte. Ein Verfahren zu kennen heißt zu wissen, wo es aufhört — nicht, wofür es gedacht ist.

Was dieser Teil gemessen hat

Behauptung	Gemessen	Wo
„Der exakte Solver ist immer besser.”	Bei 500 Aufträgen liefert CP-SAT eine Lösung, die **5,2 % schlechter** ist als die Faustregel eines Meisters	Kapitel 9
„Metaheuristiken sind ungenau.”	Faustregel 2 497 → Annealing 2 343 → Annealing plus LNS **2 289** Minuten Rüstzeit; untere Schranke 1 768	Kapitel 9
„Man muss alle Variablen aufschreiben.”	28 erzeugte Muster von **46 408** möglichen — und die Lösung ist beweisbar optimal	Kapitel 10
„Dekomposition lohnt sich immer.”	Bei drei Stücken je Rolle **11 %** Ersparnis, bei sechs **nichts**	Kapitel 10
„Mit Seed ist der Lauf reproduzierbar.”	Schon **zwei** Arbeiter liefern bei identischem Seed drei verschiedene Pläne zum selben Zielwert	Abschnitt 7.7

Drei Fehler, die dieser Teil verhindert

1. **Runden.** Die LP-Lösung ist keine Näherung der ganzzahligen Lösung — sie kann beliebig weit danebenliegen, und gerundet sogar unzulässig werden.
2. **Ein zu großes Big-M.** Es macht das Modell nicht falsch, sondern die Suche langsam und die Schranken wertlos. So klein wie zulässig, nie „sicherheitshalber groß”.
3. **Zu früh heuristisch werden.** Der Umschlagpunkt, ab dem eine Metaheuristik den exakten Solver schlägt, ist eine Eigenschaft **des Problems** und lässt sich messen. Wer ihn nicht misst, verzichtet auf Optimalitätsgarantien, die er hätte haben können.

Wenn Sie nur eines mitnehmen

 Die Frage lautet nie „welcher Solver ist der beste", sondern „welche Garantie brauche ich, und was bin ich bereit, dafür an Laufzeit zu zahlen". Ein Verfahren ohne Schranke liefert eine Lösung; ein Verfahren mit Schranke liefert die Erlaubnis, aufzuhören.

Teil III: Nichtlinearität, Unsicherheit und mehrperiodige Dynamik

Teil II ging von festen Daten und linearen Zusammenhängen aus. Beides gilt in der Praxis oft nicht: Risiko wächst **quadratisch** mit dem Einsatz, Nachfrage ist erst morgen bekannt, und eine Entscheidung heute verändert, welche Möglichkeiten übermorgen noch offenstehen.

Drei Kapitel dieses Teils behandeln genau diese drei Abweichungen. Welches Werkzeug Sie brauchen, hängt davon ab, **welche** davon bei Ihnen vorliegt:

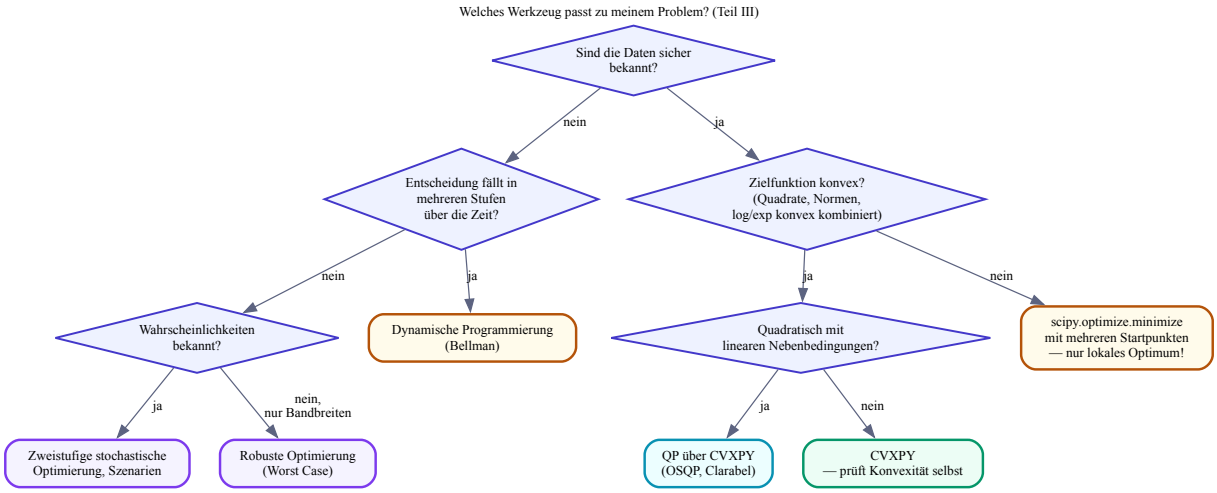

Abbildung 17: Entscheidungsdiagramm: Welches Werkzeug passt zu meinem Problem? (Teil III)

 Merksatz zum Diagramm Die erste Frage trennt zwei Welten, die man nicht vermischen darf. Ist die **Nichtlinearität** das Problem, geht es um Konvexität — und damit darum, ob Sie eine Optimalitätsgarantie bekommen oder nur ein lokales Optimum. Ist die **Unsicherheit** das Problem, geht es um etwas ganz anderes: Nicht der Solver ist die Schwierigkeit, sondern die Frage, was „optimal" überhaupt heißen soll, wenn man die Daten nicht kennt.

Der häufigste Fehler an dieser Stelle ist, Unsicherheit durch **Mittelwerte** zu ersetzen und dann deterministisch zu rechnen. Warum das systematisch danebengeht, zeigt Kapitel 12 unter dem Namen *Fluch des Durchschnitts*.

Zwei weitere Kapitel schließen sich an, weil sie dieselbe Voraussetzung aufgeben, nur an einer anderen Stelle: Kapitel 14 gibt das **eine Ziel** auf, Kapitel 15 die Annahme, die **Eingabedaten** seien gegeben statt selbst geschätzt. Was dieser Teil insgesamt leistet, fasst die Synthese an seinem Ende zusammen.

Kapitel 11: Quadratische und nichtlineare Optimierung — KKT, Lagrange, Konvexität

Kapitel auf einen Blick

Worum geht es? Risiko ist quadratisch. Sobald Varianz ins Spiel kommt, verlässt man die Welt der Polyeder. Dieses Kapitel liefert das mathematische Fundament für die gesamte Portfoliooptimierung in Teil IV.

Voraussetzungen: Kapitel 2 (Konvexität, Eigenwerte), Kapitel 5 (Dualität, Schattenpreise). Gradienten werden in Abschnitt 11.2 wiederholt.

Danach können Sie: Ein quadratisches Programm aufstellen, die KKT-Bedingungen anwenden, den Zusammenhang zwischen Lagrange-Multiplikator und Schattenpreis erklären, eine gültige Kovarianzmatrix konstruieren — und mit nicht-konvexen Problemen umgehen, ohne ein lokales Optimum für das Optimum zu halten.

Zeitbedarf: ca. 6,5 Stunden.

Programme:
`QP_Grundlagen.py`
`KKT_Nachweis.py`
`Entropie_Maximierte_Allokation.py`
`Lokale_Optima_Multistart.py`

Notebook: `Notebooks_04/qp-nlp.ipynb`

11.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Warum die riskantere Anlage das Risiko senkt

Ein Betrieb legt Rücklagen an und hat zwei Möglichkeiten:

Anlage	Schwankung (Standardabweichung)
A — solide, wenig Bewegung	12 %
B — deutlich unruhiger	28 %

Die beiden bewegen sich **gegenläufig**: Korrelation −0,40. Wie viel soll in jede?

```
import numpy as np
import cvxpy as cp

sigma = np.array([0.12, 0.28])                    # Schwankung je Anlage
rho = -0.40                                       # Korrelation
S = np.array([[sigma[0]**2,            rho * sigma[0] * sigma[1]],
              [rho * sigma[0] * sigma[1], sigma[1]**2]])
```

```
w = cp.Variable(2, nonneg=True)                    # Anteile, keine Leerverkäufe
problem = cp.Problem(cp.Minimize(cp.quad_form(w, S)), [cp.sum(w) == 1])
problem.solve()

print("Anteile:", np.round(w.value, 3))
print(f"Risiko der Mischung: {np.sqrt(problem.value) * 100:.2f} %")
```

Ausgabe:

```
Anteile: [0.767 0.233]
Risiko der Mischung: 8.90 %
```

Rechnen wir die naheliegenden Alternativen dagegen:

Aufteilung	Schwankung des Ganzen
Nur A — „die sichere Anlage“	12,00 %
50 / 50 — „breit streuen“	**12,84 %**
Nur B	28,00 %
Optimum: 76,7 % A, 23,3 % B	**8,90 %**

Drei Dinge stehen in dieser Tabelle, und alle drei widersprechen der Intuition:

1. **Die Beimischung der *riskanteren* Anlage senkt das Risiko** — von 12 % auf 8,9 %, also um mehr als ein Viertel. Wer nur auf die 28 % schaut, würde B nie anfassen.
2. **Naives Streuen macht es schlechter.** Die 50/50-Mischung liegt mit 12,84 % *über* dem Wert, den man bekäme, wenn man ausschließlich A hielte. „Diversifizieren“ ist also keine Regel, sondern eine Frage nach dem richtigen Maß.
3. **Das richtige Maß ist weder 0 % noch 50 %**, sondern 23,3 % — eine Zahl, auf die man durch Nachdenken nicht kommt. Genau dafür gibt es dieses Kapitel.

Warum funktioniert das? Weil Risiko sich nicht addiert. Die Schwankung einer Mischung ist nicht der Durchschnitt der Einzelschwankungen, sondern folgt einer **quadratischen** Form:

$$\sigma_p^2 = w_A^2\sigma_A^2 + w_B^2\sigma_B^2 + 2\,w_A w_B \rho\,\sigma_A\sigma_B$$

Der letzte Term ist der entscheidende: Bei negativer Korrelation ist er **negativ** und zieht das Gesamtrisiko herunter. Deshalb verlässt man mit Risiko die Welt der Polyeder aus Teil II — die Zielfunktion ist kein Skalarprodukt mehr, sondern eine quadratische Form. Was das für die Lösbarkeit bedeutet und warum CVXPY das trotzdem mit Optimalitätsgarantie löst, klärt der Rest des Kapitels.

> **Merksatz** `cp.quad_form(w, S)` ist die Zeile, um die es im ganzen Kapitel geht. Sie ist genau dann harmlos, wenn **S positiv semidefinit** ist — und genau dann ein Problem, wenn nicht. CVXPY prüft das für Sie und verweigert die Arbeit im Zweifelsfall. Diese Verweigerung ist ein Schutzmechanismus, keine Schikane.

11.2 Lernziele

Nach diesem Kapitel können Sie …

1. … ein **quadratisches Programm (QP)** aufstellen und von einem allgemeinen nichtlinearen Programm abgrenzen.
2. … die **KKT-Bedingungen** herleiten und auf ein Optimierungsproblem anwenden.
3. … den Zusammenhang zwischen **Lagrange-Multiplikator** und **Schattenpreis** erklären.
4. … zwischen **konvex** und **streng konvex** unterscheiden und eine gültige, positiv semidefinite **Kovarianzmatrix** konstruieren.

Auffrischung: Gradient in einer Minute

Der **Gradient** $\nabla f(\mathbf{x})$ ist der Vektor aller partiellen Ableitungen:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)^\top$$

□ **Formel-Lesehilfe** $\partial f / \partial x_1$ bedeutet: „Wie stark ändert sich f, wenn ich **nur** x_1 ein winziges Stück erhöhe und alles andere festhalte?“

Ohne Formel gesagt: Der Gradient ist der Pfeil, der in die Richtung des steilsten **Anstiegs** zeigt. Seine Länge ist die Steilheit. Beim Minimieren geht man deshalb in Richtung $-\nabla f$ — bergab.

Die eine Regel, die Sie brauchen: Im Inneren eines Gebiets ist an einem Minimum der Gradient **null** — es geht nirgendwohin mehr bergab. An einem Rand (also wenn eine Nebenbedingung bindet) gilt das nicht mehr, und genau dafür braucht man die KKT-Bedingungen.

Zwei Ableitungsregeln, die im ganzen Buch gebraucht werden:

Ausdruck	Gradient	Merkhilfe
$f(\mathbf{x}) = \mathbf{c}^\top \mathbf{x}$	$\nabla f = \mathbf{c}$	wie $(cx)' = c$
$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{P}\mathbf{x}$ (mit $\mathbf{P}$ symmetrisch)	$\nabla f = \mathbf{P}\mathbf{x}$	wie $(\frac{1}{2}px^2)' = px$

11.3 Das quadratische Programm (QP)

Ein **quadratisches Programm** hat eine quadratische Zielfunktion und **lineare** Nebenbedingungen:

$$\min_{\mathbf{x}} \quad \frac{1}{2}\mathbf{x}^\top \mathbf{P}\mathbf{x} + \mathbf{q}^\top \mathbf{x} \qquad \text{u. d. N.} \qquad \mathbf{A}\mathbf{x} \le \mathbf{b}, \quad \mathbf{F}\mathbf{x} = \mathbf{d}$$

□ **Formel-Lesehilfe** * $\mathbf{x}^\top \mathbf{P}\mathbf{x}$ — die **quadratische Form**. Ausgeschrieben: $\sum_i \sum_j p_{ij} x_i x_j$. In der Portfoliotheorie ist das exakt die Portfoliovarianz, wenn $\mathbf{P} = \Sigma$ die Kovarianzmatrix ist.
* Der Faktor $\frac{1}{2}$ ist reine Bequemlichkeit: Er kürzt sich beim Ableiten weg ($\nabla = \mathbf{P}\mathbf{x}$ statt $2\mathbf{P}\mathbf{x}$).
* $\mathbf{q}^\top \mathbf{x}$ — ein zusätzlicher linearer Teil, etwa die negative erwartete Rendite.

> **Ohne Formel gesagt:** „Minimiere das Risiko (quadratisch) abzüglich des Ertrags (linear), unter Einhaltung der Budget- und Positionsgrenzen."

Konvexität: die präzise Aussage

Eine verbreitete, aber zu stark formulierte Aussage lautet: *„Wenn* **P** *positiv semi-definit ist, ist das QP streng konvex; es existiert ein eindeutiges globales Minimum."* Korrekt gilt:

Voraussetzung	Bedeutung	Folgerung
$\mathbf{P} \succeq 0$ — **positiv semidefinit**, alle Eigenwerte $\lambda_i \geq 0$	Die Funktion ist eine „Schüssel", die eine flache Rinne haben darf	Problem ist **konvex**. Jedes lokale Minimum ist global. Es kann aber **mehrere Minimalstellen** geben — mit demselben Zielwert
$\mathbf{P} \succ 0$ — **positiv definit**, alle $\lambda_i > 0$	Echte Schüssel ohne flache Richtung	Problem ist **streng konvex**. Die Minimalstelle ist **eindeutig**
P hat einen negativen Eigenwert	Sattel oder Rinne nach unten	Problem ist **nicht konvex**. Lokale Optima möglich, keine Garantie

Warum das praktisch zählt: Eine Stichproben-Kovarianzmatrix mit mehr Titeln als Beobachtungen ($N > T$) ist **singulär** — also nur semidefinit. Dann gibt es unendlich viele Portfolios mit exakt demselben minimalen Risiko, und der Solver liefert eines davon, scheinbar willkürlich. Kleine Datenänderungen führen zu völlig anderen Gewichten. Genau dieses Problem behebt die Shrinkage aus Kapitel 18.

Alle drei Zeilen der Tabelle lassen sich an einem winzigen Zwei-Variablen-QP direkt beobachten — inklusive der Stelle, an der CVXPY ein nicht konvexes Problem **verweigert**, bevor überhaupt ein Solver aufgerufen wird:

```
#!/usr/bin/env python3

# QP_Grundlagen.py
"""
Kapitel QP/NLP: Die drei Faelle aus der Konvexitaets-Tabelle (Abschnitt
'Das quadratische Programm') an
einem Mini-QP demonstriert: P positiv definit, P (singulaer) semidefinit,
P mit negativem Eigenwert.
"""

import numpy as np
import cvxpy as cp


def loese_qp(P, q, name):
    n = len(q)
    w = cp.Variable(n)
```

```
    ziel = cp.Minimize(0.5 * cp.quad_form(w, P) + q @ w)
    bedingungen = [cp.sum(w) == 1, w >= 0]
    problem = cp.Problem(ziel, bedingungen)

    print(f"\n--- {name} ---")
    eigenwerte = np.linalg.eigvalsh(P)
    print(f"Eigenwerte von P: {np.round(eigenwerte, 4)}")
    print(f"DCP-konvex (CVXPY-Pruefung)? {problem.is_dcp()}")

    if not problem.is_dcp():
        print("-> CVXPY lehnt das Problem ab, BEVOR ueberhaupt ein Solver laeuft.")
        return

    problem.solve()
    print(f"Status: {problem.status}")
    print(f"w* = {np.round(w.value, 4)}")
    print(f"Zielwert = {problem.value:.6f}")


if __name__ == "__main__":
    q = np.zeros(2)

    # Fall 1: P positiv definit -> eindeutiges Minimum
    P_definit = np.array([[2.0, 0.5], [0.5, 1.0]])
    loese_qp(P_definit, q, "P positiv definit")

    # Fall 2: P singulaer/semidefinit (zwei "identische" Assets) -> unendlich viele Minima
    P_semidefinit = np.array([[1.0, 1.0], [1.0, 1.0]])
    loese_qp(P_semidefinit, q, "P positiv semidefinit (singulaer)")

    # Fall 3: P mit negativem Eigenwert -> nicht konvex
    P_indefinit = np.array([[1.0, 2.0], [2.0, 1.0]])
    loese_qp(P_indefinit, q, "P indefinit (negativer Eigenwert)")

    print("\n--- Nachweis: 'unendlich viele Minima' im semidefiniten Fall ---")
    for punkt in [np.array([1.0, 0.0]), np.array([0.0, 1.0]), np.array([0.3, 0.7])]:
        wert = 0.5 * punkt @ P_semidefinit @ punkt
        print(f"  w = {punkt} -> Zielwert = {wert:.4f}  (identisch, obwohl w verschieden)")
```

Erwartete Ausgabe:

```
--- P positiv definit ---
Eigenwerte von P: [0.7929 2.2071]
DCP-konvex (CVXPY-Pruefung)? True
Status: optimal
w* = [0.25 0.75]
Zielwert = 0.437500
```

```
--- P positiv semidefinit (singulaer) ---
Eigenwerte von P: [0. 2.]
DCP-konvex (CVXPY-Pruefung)? True
Status: optimal
w* = [0.5 0.5]
Zielwert = 0.500000

--- P indefinit (negativer Eigenwert) ---
Eigenwerte von P: [-1.  3.]
DCP-konvex (CVXPY-Pruefung)? False
-> CVXPY lehnt das Problem ab, BEVOR ueberhaupt ein Solver laeuft.

--- Nachweis: 'unendlich viele Minima' im semidefiniten Fall ---
  w = [1. 0.] -> Zielwert = 0.5000  (identisch, obwohl w verschieden)
  w = [0. 1.] -> Zielwert = 0.5000  (identisch, obwohl w verschieden)
  w = [0.3 0.7] -> Zielwert = 0.5000  (identisch, obwohl w verschieden)
```

Der letzte Block ist der eigentliche Beweis: Drei völlig verschiedene Portfolios $(1, 0)$, $(0, 1)$ und $(0{,}3;\ 0{,}7)$ liefern **exakt** denselben Zielwert — die „Rinne" aus der Tabelle ist keine Metapher, sie ist hier eine echte Gerade im Lösungsraum.

Prüfen Sie jede Matrix, bevor Sie sie verwenden

```python
eigenwerte = np.linalg.eigvalsh(Sigma)     # 'h' = fuer symmetrische Matrizen
assert np.all(eigenwerte >= -1e-10), "Sigma ist nicht positiv semidefinit!"
print(f"Konditionszahl: {eigenwerte.max() / eigenwerte.min():.1f}")
```

Eine einzige Zeile — und sie deckt eine ungültige Kovarianzmatrix sofort auf.

11.4 Die Karush-Kuhn-Tucker-Bedingungen

Die **KKT-Bedingungen** verallgemeinern den Lagrange-Ansatz auf Probleme mit **Ungleichungen**. Sie sind notwendig für ein Optimum — und bei konvexen Problemen auch hinreichend.

Für das Problem

$$\min f(\mathbf{x}) \quad \text{u. d. N.} \quad g_i(\mathbf{x}) \leq 0 \ (i = 1..m), \qquad h_j(\mathbf{x}) = 0 \ (j = 1..p)$$

lautet die **Lagrange-Funktion**:

$$\mathcal{L}(\mathbf{x}, \lambda, \nu) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) + \sum_{j=1}^{p} \nu_j h_j(\mathbf{x})$$

Formel-Lesehilfe Die Lagrange-Funktion **verpackt die Nebenbedingungen in die Zielfunktion**. Jede Bedingung bekommt einen Preis: λ_i bzw. ν_j. Verletzt man Bedingung i, wird $g_i > 0$, und der Term $\lambda_i g_i$ erhöht $\mathcal{L}$ — es kostet also.

Ohne Formel gesagt: Statt „du darfst nicht“ sagt man „du darfst, aber es kostet λ_i je Einheit Überschreitung“. Bei den richtigen Preisen verhält sich der Optimierer dann von selbst regelkonform. Diese Preise sind exakt die **Schattenpreise** aus Kapitel 5.

Die vier KKT-Bedingungen

Am Optimum $(\mathbf{x}^*, \lambda^*, \nu^*)$ gilt:

1. Stationarität — der Gradient der Lagrange-Funktion verschwindet:

$$\nabla f(\mathbf{x}^*) + \sum_i \lambda_i^* \nabla g_i(\mathbf{x}^*) + \sum_j \nu_j^* \nabla h_j(\mathbf{x}^*) = \mathbf{0}$$

Anschaulich: Die Kraft, die den Punkt bergab ziehen will ($-\nabla f$), wird exakt von den Nebenbedingungen aufgefangen. Wie ein Ball, der in einer Ecke liegen bleibt: Die Schwerkraft zieht, die Wände drücken dagegen, und die Summe ist null.

2. Primale Zulässigkeit — die ursprünglichen Bedingungen gelten: $g_i(\mathbf{x}^*) \le 0$, $h_j(\mathbf{x}^*) = 0$.

3. Duale Zulässigkeit — die Multiplikatoren der Ungleichungen sind nichtnegativ: $\lambda_i^* \ge 0$.

Warum? Eine Wand kann nur **drücken**, nicht ziehen. Ein negatives λ hieße, die Bedingung würde die Lösung von sich weg**ziehen** — dann wäre sie nicht bindend.

4. Komplementärer Schlupf — $\lambda_i^* \cdot g_i(\mathbf{x}^*) = 0$ für alle i.

Bedeutung: Entweder ist die Bedingung nicht bindend ($g_i < 0$), **dann muss** $\lambda_i = 0$ sein. Oder der Multiplikator ist positiv, **dann muss** die Bedingung mit Gleichheit binden. Das ist derselbe Satz wie in Kapitel 5 — nur allgemeiner formuliert.

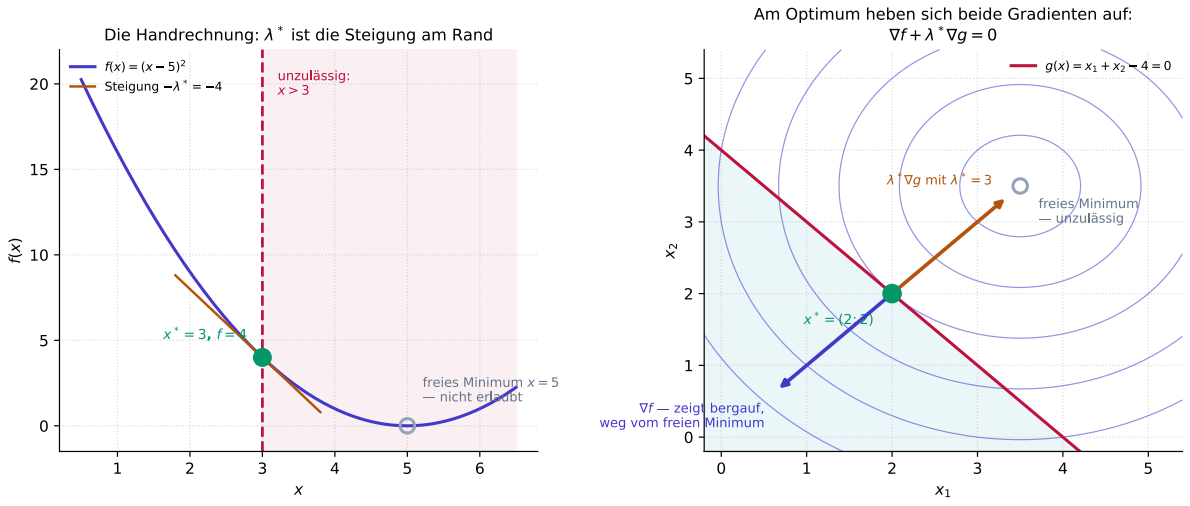

Abbildung 18: Abb. 11.1: Links die Handrechnung dieses Abschnitts: Das freie Minimum liegt bei $x = 5$, die Grenze lässt nur $x \le 3$ zu, und $\lambda^* = 4$ ist genau die Steigung von f am Rand — der Preis dafür, dass es nicht weitergeht. Rechts dieselbe Aussage in zwei Dimensionen: ∇f und $\lambda^* \nabla g$ zeigen gegeneinander und heben sich auf. Das Skript `bilder_04/erzeuge_kkt.py` prüft alle vier KKT-Bedingungen numerisch, bevor es zeichnet.

Formel-Übersetzer: die vier KKT-Bedingungen

Vier Zeilen Mathematik, vier Sätze, die man einem Betriebsleiter sagen könnte:

Mathematik	Alltagssprache
$\nabla f(\mathbf{x}^*) + \sum_i \lambda_i^* \nabla g_i(\mathbf{x}^*) = \mathbf{0}$	„Der Zug ins Bessere und der Widerstand der Grenzen heben sich genau auf — sonst könnte man noch einen Schritt gehen.“
$g_i(\mathbf{x}^*) \le 0$	„Der Plan hält alle Regeln ein.“
$\lambda_i^* \ge 0$	„Eine Grenze kann nur bremsen, nie antreiben.“ Ein negativer Preis wäre widersinnig.
$\lambda_i^* \cdot g_i(\mathbf{x}^*) = 0$	„Was Reserve hat, ist nichts wert. Was etwas wert ist, ist ausgereizt.“ Nie beides.
λ_i^* selbst	Der **Schattenpreis** von Bedingung i — dieselbe Größe wie im LP (Kapitel 5), nur für krumme Ränder.

Alle vier zusammen in einem Satz: *Wir stehen an einer Stelle, an der jede Verbesserung an eine Grenze stößt — und wir wissen für jede dieser Grenzen, was ihre Lockerung wert wäre.*

Warum das praktisch zählt: Die KKT-Bedingungen sind nicht nur Theorie, sondern eine **Prüfvorschrift**. Bei einem konvexen Problem sind sie notwendig *und* hinreichend — wer sie an einer gefundenen Lösung nachrechnet, hat damit bewiesen, dass sie optimal ist. Genau das tut `KKT_Nachweis.py` weiter unten.

Handrechnung 11.1: KKT an einem Minimalbeispiel

$$\min\ f(x) = (x-5)^2 \qquad \text{u. d. N.} \qquad g(x) = x - 3 \le 0$$

Ohne Nebenbedingung läge das Minimum bei $x = 5$. Aber $5 > 3$ — unzulässig.

Lagrange: $\mathcal{L}(x, \lambda) = (x-5)^2 + \lambda(x-3)$.

KKT 1 (Stationarität): $\frac{\partial \mathcal{L}}{\partial x} = 2(x-5) + \lambda = 0 \Rightarrow x = 5 - \lambda/2$.

Fall A: $\lambda = 0$ (Bedingung nicht bindend). Dann $x = 5$. Prüfe KKT 2: $g(5) = 2 > 0$ **unzulässig** — dieser Fall entfällt.

Fall B: $\lambda > 0$ (Bedingung bindend). Aus KKT 4 folgt $g(x) = 0$, also $x = 3$. Einsetzen in KKT 1: $2(3-5) + \lambda = 0 \Rightarrow \lambda = 4$. Prüfe KKT 3: $\lambda = 4 > 0$

Lösung: $x^* = 3$, $\lambda^* = 4$, $f(x^*) = 4$.

Und was bedeutet $\lambda^* = 4$? Es ist der Schattenpreis: Dürfte x bis 3,1 gehen, sänke f um etwa $4 \times 0{,}1 = 0{,}4$. Probe: $f(3{,}1) = (3{,}1-5)^2 = 3{,}61$, tatsächlicher Rückgang 0,39 — die kleine Abweichung ist die Krümmung.

```python
#!/usr/bin/env python3

# KKT_Nachweis.py
"""
Kapitel QP/NLP: Die KKT-Bedingungen numerisch nachpruefen.

Loest ein QP mit CVXPY, liest die Dualwerte aus und prueft alle vier
KKT-Bedingungen einzeln nach. Das ist zugleich eine Vorlage fuer die
Qualitaetssicherung eigener Modelle.
"""

import numpy as np
import cvxpy as cp


def baue_gueltige_kovarianz(vola, korrelationen, seed=0):
    """
    Baut Sigma = D * C * D aus Volatilitaeten und einer Korrelationsmatrix.
    Dieses Vorgehen ist konstruktionsbedingt positiv semidefinit - im
    Gegensatz zum nachtraeglichen Ueberschreiben der Diagonalen, das die
    positive Semidefinitheit zerstoeren kann.
    """
    C = np.array(korrelationen, dtype=float)
    assert np.allclose(C, C.T), "Korrelationsmatrix muss symmetrisch sein."
    assert np.allclose(np.diag(C), 1.0), "Diagonale der Korrelationsmatrix muss 1 sein."
    eigen = np.linalg.eigvalsh(C)
    assert eigen.min() > -1e-10, (
        f"Korrelationsmatrix ist nicht positiv semidefinit "
        f"(kleinster Eigenwert {eigen.min():.4f}). Solche Korrelationen sind unmoeglich.")
    D = np.diag(vola)
    return D @ C @ D


if __name__ == "__main__":
    # --- Ein kleines Portfolio-QP ----------------------------------------
    vola = np.array([0.20, 0.14, 0.30])                # Volatilitaeten
    korr = [[1.00, 0.30, 0.10],
            [0.30, 1.00, 0.25],
            [0.10, 0.25, 1.00]]
    Sigma = baue_gueltige_kovarianz(vola, korr)
    mu = np.array([0.09, 0.05, 0.13])                  # erwartete Renditen
    lam = 3.0                                          # Risikoaversion

    eigenwerte = np.linalg.eigvalsh(Sigma)
    print("=" * 74)
    print("  KKT-BEDINGUNGEN AM PORTFOLIO-QP")
    print("=" * 74)
    print(f"Eigenwerte von Sigma: {np.round(eigenwerte, 6)}")
```

```
print(f"  -> positiv definit: {bool(eigenwerte.min() > 0)}  "
      f"(Problem ist streng konvex, Loesung eindeutig)")
print(f"  -> Konditionszahl:  {eigenwerte.max() / eigenwerte.min():.2f}")

# --- Modell: min  lam/2 * w'Sigma w - mu'w   u.d.N. sum(w)=1, w>=0 ----
n = len(mu)
w = cp.Variable(n)
ziel = cp.Minimize(0.5 * lam * cp.quad_form(w, Sigma) - mu @ w)
budget = cp.sum(w) == 1
nichtnegativ = w >= 0
problem = cp.Problem(ziel, [budget, nichtnegativ])
problem.solve()

w_opt = w.value
nu = budget.dual_value                    # Multiplikator der Gleichung
lam_i = nichtnegativ.dual_value           # Multiplikatoren der Ungleichungen

print(f"\nStatus: {problem.status}")
print(f"Optimale Gewichte: {np.round(w_opt, 6)}")
print(f"Zielwert:          {problem.value:.6f}")
print(f"Multiplikator der Budgetgleichung (nu): {nu:.6f}")
print(f"Multiplikatoren der w>=0-Bedingungen:   {np.round(lam_i, 6)}")

# --- KKT-Bedingungen einzeln pruefen ---------------------------------
print("\n--- Pruefung der vier KKT-Bedingungen ---")

# 1. Stationaritaet:  grad f - lambda + nu*1 = 0
#    f(w) = lam/2 w'Sigma w - mu'w   ->   grad f = lam*Sigma w - mu
#    g_i(w) = -w_i <= 0              ->   grad g_i = -e_i
#    h(w)   = sum(w) - 1 = 0         ->   grad h   = 1
grad_f = lam * (Sigma @ w_opt) - mu
stationaritaet = grad_f - lam_i + nu * np.ones(n)
print(f"1. Stationaritaet   : max|Residuum| = {np.abs(stationaritaet).max():.2e}")

# 2. Primale Zulaessigkeit
print(f"2. Primal zulaessig : sum(w)-1 = {w_opt.sum()-1:.2e}, "
      f"min(w) = {w_opt.min():.2e}")

# 3. Duale Zulaessigkeit
print(f"3. Dual zulaessig   : min(lambda) = {lam_i.min():.2e}  (muss >= 0 sein)")

# 4. Komplementaerer Schlupf: lambda_i * w_i = 0
print(f"4. Kompl. Schlupf   : max|lambda_i * w_i| = "
      f"{np.abs(lam_i * w_opt).max():.2e}")

alle_ok = (np.abs(stationaritaet).max() < 1e-6
           and abs(w_opt.sum() - 1) < 1e-8
           and w_opt.min() > -1e-8
```

```
            and lam_i.min() > -1e-8
            and np.abs(lam_i * w_opt).max() < 1e-6)
print(f"\nAlle vier KKT-Bedingungen erfuellt: {alle_ok}")

# --- Interpretation von nu ------------------------------------------
print("\n--- Was bedeutet nu? ---")
print("nu ist der Schattenpreis des Budgets: Um so viel aendert sich der")
print("Zielwert, wenn man statt 100 % nur 99 % investieren duerfte.")
problem2 = cp.Problem(cp.Minimize(0.5 * lam * cp.quad_form(w, Sigma) - mu @ w),
                      [cp.sum(w) == 1.01, w >= 0])
problem2.solve()
print(f"  Vorhergesagt (nu * 0.01): {nu * 0.01:+.6f}")
print(f"  Tatsaechlich gemessen:    {problem2.value - problem.value:+.6f}")
print("=" * 74)
```

11.5 Nichtlineare Optimierung mit `scipy.optimize.minimize`

Wenn Zielfunktion oder Nebenbedingungen weder linear noch quadratisch sind — Logarithmen, Wurzeln, Exponentialfunktionen — nutzt man gradientenbasierte Verfahren wie **SLSQP** (*Sequential Least Squares Programming*).

 Der entscheidende Unterschied zu CVXPY CVXPY **prüft** die Konvexität und garantiert bei Erfolg das globale Optimum. `scipy.optimize.minimize` prüft nichts und liefert ein **lokales** Optimum, das vom Startpunkt abhängt (siehe die Demonstration in Abschnitt 2.5). Wer SLSQP einsetzt, sollte **immer mit mehreren Startpunkten rechnen** und die Ergebnisse vergleichen.

Praxisfall: Entropie-maximierte Kapitalallokation

Wir optimieren ein Portfolio nicht nur nach Varianz, sondern maximieren zusätzlich die **Shannon-Entropie** der Gewichte, um Klumpenrisiken glatt zu bestrafen:

$$\min_{\mathbf{w}} \quad \underbrace{\tfrac{1}{2}\mathbf{w}^\top \Sigma \mathbf{w}}_{\text{Risiko}} - \underbrace{\alpha\, \mu^\top \mathbf{w}}_{\text{Ertrag}} - \underbrace{\beta\Big(-\textstyle\sum_i w_i \ln w_i\Big)}_{\text{Diversifikation}}$$

 Formel-Lesehilfe zur Entropie $H(\mathbf{w}) = -\sum_i w_i \ln w_i$ ist ein Streuungsmaß: * Alles in einem Titel ($w = (1,0,0,0)$): $H = 0$ — minimale Streuung. * Gleichverteilt ($w_i = 1/4$): $H = \ln 4 \approx 1{,}386$ — maximale Streuung.

Ohne Formel gesagt: Die Entropie misst, wie „breit verteilt“ das Portfolio ist. Weil sie mit einem Minuszeichen in die zu minimierende Zielfunktion eingeht, wird breite Streuung **belohnt** — und zwar mathematisch glatt, ohne harte Obergrenzen. Der Vorteil gegenüber einer Schranke wie $w_i \le 0{,}2$: Der Übergang ist weich, das Ergebnis reagiert nicht sprunghaft auf kleine Datenänderungen.

```python
#!/usr/bin/env python3

# Entropie_Maximierte_Allokation.py
"""
Kapitel QP/NLP: Nichtlineare Optimierung (NLP) mit scipy SLSQP.
Modell: Mean-Variance-Portfolio mit Shannon-Entropie-Diversifikation.

Achtung, haeufiger Fehler: Sigma per A@A.T zu erzeugen und anschliessend die
Diagonale zu ueberschreiben zerstoert die positive Semidefinitheit - die
Matrix kann dadurch einen negativen Eigenwert bekommen und unmoegliche
Korrelationen (> 1) implizieren. Dieses Programm baut Sigma stattdessen als
D * C * D aus Volatilitaeten und einer echten Korrelationsmatrix -
konstruktionsbedingt immer PSD.

Zusaetzlich: Multistart, weil SLSQP nur lokale Optima findet.
"""

import numpy as np
import pandas as pd
from scipy.optimize import minimize

ASSETS = ["Tech-Aktien", "Rohstoffe", "US-Treasuries", "Krypto"]
MU = np.array([0.14, 0.07, 0.03, 0.22])          # erwartete Jahresrenditen
VOLA = np.array([0.20, 0.14, 0.07, 0.40])        # Volatilitaeten p.a.
KORRELATION = np.array([
    [1.00, 0.25, -0.10, 0.55],
    [0.25, 1.00, 0.05, 0.20],
    [-0.10, 0.05, 1.00, -0.15],
    [0.55, 0.20, -0.15, 1.00],
])

ALPHA = 1.0        # Gewicht des Ertrags
BETA = 0.015       # Gewicht der Entropie-Diversifikation
UNTERGRENZE = 0.001
OBERGRENZE = 0.80


def baue_kovarianz() -> np.ndarray:
    """Sigma = D * C * D. Immer PSD, wenn C eine gueltige Korrelationsmatrix ist."""
    eig_c = np.linalg.eigvalsh(KORRELATION)
    if eig_c.min() < -1e-10:
        raise ValueError(f"Korrelationsmatrix unmoeglich (Eigenwert {eig_c.min():.4f}).")
    D = np.diag(VOLA)
    Sigma = D @ KORRELATION @ D
    eig_s = np.linalg.eigvalsh(Sigma)
    assert eig_s.min() > 0, "Sigma nicht positiv definit!"
    return Sigma
```

```
SIGMA = baue_kovarianz()


def zielfunktion(w, alpha=ALPHA, beta=BETA):
    """Risiko - Ertrag - Entropiepraemie (wird minimiert)."""
    varianz = 0.5 * w @ SIGMA @ w
    ertrag = MU @ w
    entropie = -np.sum(w * np.log(w + 1e-12))
    return varianz - alpha * ertrag - beta * entropie


def gradient(w, alpha=ALPHA, beta=BETA):
    """Exakter analytischer Gradient - beschleunigt und stabilisiert SLSQP."""
    grad_varianz = SIGMA @ w                        # d/dw von 0.5 w'Sigma w
    grad_ertrag = -alpha * MU
    grad_entropie = beta * (np.log(w + 1e-12) + 1.0)
    return grad_varianz + grad_ertrag + grad_entropie


def optimiere(startpunkt, alpha=ALPHA, beta=BETA):
    """alpha und beta werden durchgereicht - so bleibt die Funktion seiteneffektfrei."""
    return minimize(
        zielfunktion, startpunkt, args=(alpha, beta), jac=gradient,
        method="SLSQP", bounds=[(UNTERGRENZE, OBERGRENZE)] * len(MU),
        constraints=({"type": "eq",
                      "fun": lambda w: np.sum(w) - 1.0,
                      "jac": lambda w: np.ones(len(MU))}),
        options={"ftol": 1e-12, "maxiter": 300})


if __name__ == "__main__":
    n = len(MU)
    eig = np.linalg.eigvalsh(SIGMA)

    print("=" * 74)
    print("   NICHTLINEARE ENTROPIE-OPTIMIERTE ASSET-ALLOKATION")
    print("=" * 74)
    print("Pruefung der Kovarianzmatrix:")
    print(f"  Eigenwerte:          {np.round(eig, 6)}")
    print(f"  Positiv definit:     {bool(eig.min() > 0)}")
    print(f"  Groesste Korrelation ausserhalb der Diagonale: "
          f"{np.abs(KORRELATION - np.eye(n)).max():.2f}  (muss <= 1 sein)")

    # --- Multistart: SLSQP findet nur lokale Optima -----------------------
    rng = np.random.default_rng(7)
    startpunkte = [np.ones(n) / n]                                  # Gleichgewichtung
    for _ in range(9):
```

```
        z = rng.random(n) + 0.05
        startpunkte.append(z / z.sum())

    ergebnisse = [optimiere(s) for s in startpunkte]
    erfolgreich = [r for r in ergebnisse if r.success]
    bestes = min(erfolgreich, key=lambda r: r.fun)
    zielwerte = np.array([r.fun for r in erfolgreich])

    print(f"\nMultistart mit {len(startpunkte)} Startpunkten:")
    print(f"  Erfolgreich konvergiert: {len(erfolgreich)}")
    print(f"  Spannweite der Zielwerte: {zielwerte.max() - zielwerte.min():.2e}")
    print("  -> " + ("alle Startpunkte fuehren zum selben Optimum (Indiz fuer Konvexitaet)"
                     if zielwerte.max() - zielwerte.min() < 1e-6
                     else "ACHTUNG: verschiedene lokale Optima gefunden!"))

    w_opt = bestes.x
    rendite = MU @ w_opt
    volatilitaet = np.sqrt(w_opt @ SIGMA @ w_opt)
    entropie = -np.sum(w_opt * np.log(w_opt))

    print(f"\nKonvergenz:                {bestes.message}")
    print(f"Iterationen:               {bestes.nit}")
    print(f"Erwartete Jahresrendite:   {rendite * 100:6.2f} %")
    print(f"Erwartete Volatilitaet:    {volatilitaet * 100:6.2f} %")
    print(f"Diversifikations-Entropie: {entropie:.4f} "
          f"(Maximum bei Gleichgewichtung: {np.log(n):.4f})")

    print("\n" + pd.DataFrame({
        "Asset": ASSETS,
        "Erw. Rendite": [f"{r*100:.1f} %" for r in MU],
        "Volatilitaet": [f"{v*100:.1f} %" for v in VOLA],
        "Gewicht": [f"{w*100:6.2f} %" for w in w_opt],
    }).to_string(index=False))

    # --- Vergleich: was passiert ohne Entropieterm? ----------------------
    ohne = optimiere(np.ones(n) / n, beta=0.0)
    print("\n--- Wirkung des Entropieterms ---")
    print(f"  Mit Entropie  (beta={BETA}): Gewichte {np.round(w_opt * 100, 1)}")
    print(f"  Ohne Entropie (beta=0):     Gewichte {np.round(ohne.x * 100, 1)}")
    print("  Der Entropieterm zieht Kapital aus der Spitzenposition heraus,")
    print("  ohne dass eine harte Obergrenze noetig waere.")
    print("=" * 74)
```

Erwartete Ausgabe:

```
==========================================================================
   NICHTLINEARE ENTROPIE-OPTIMIERTE ASSET-ALLOKATION
==========================================================================
```

```
Pruefung der Kovarianzmatrix:
  Eigenwerte:          [0.004733 0.017389 0.026806 0.175572]
  Positiv definit:     True
  Groesste Korrelation ausserhalb der Diagonale: 0.55  (muss <= 1 sein)

Multistart mit 10 Startpunkten:
  Erfolgreich konvergiert: 10
  Spannweite der Zielwerte: 4.25e-13
  -> alle Startpunkte fuehren zum selben Optimum (Indiz fuer Konvexitaet)

Konvergenz:                Optimization terminated successfully
Iterationen:               13
Erwartete Jahresrendite:    18.62 %
Erwartete Volatilitaet:     29.01 %
Diversifikations-Entropie: 0.7935 (Maximum bei Gleichgewichtung: 1.3863)

        Asset Erw. Rendite Volatilitaet Gewicht
  Tech-Aktien       14.0 %       20.0 %  35.96 %
    Rohstoffe        7.0 %       14.0 %   2.75 %
US-Treasuries        3.0 %        7.0 %   0.45 %
       Krypto       22.0 %       40.0 %  60.84 %

--- Wirkung des Entropieterms ---
  Mit Entropie  (beta=0.015): Gewichte [36.  2.8  0.5 60.8]
  Ohne Entropie (beta=0):     Gewichte [31.9  0.1  0.1 67.9]
  Der Entropieterm zieht Kapital aus der Spitzenposition heraus,
  ohne dass eine harte Obergrenze noetig waere.
======================================================================
```

Vergleichen Sie diese Ausgabe mit der eingangs beschriebenen fehlerhaften Konstruktion. Die Eigenwerte sind jetzt **alle positiv** (kleinster: 0,0047 statt −0,084), die größte Korrelation liegt bei 0,55 statt bei unmöglichen 6,8 — und der Multistart bestätigt mit einer Spannweite von $4{,}3 \times 10^{-13}$, dass alle zehn Startpunkte im selben Optimum landen. Das ist der empirische Beleg für Konvexität — mit einer indefiniten Matrix wäre das nicht möglich gewesen.

Interessant ist auch die Wirkung des Entropieterms: Er verschiebt nur rund 7 Prozentpunkte von Krypto weg — aber er hebt die Kleinstpositionen von 0,1 % auf 2,8 % bzw. 0,5 %. Eine harte Obergrenze hätte stattdessen genau bei der Schranke abgeschnitten und alles andere unverändert gelassen. **Der weiche Term verteilt, die harte Schranke kappt.**

Code-Durchgang: die drei Lehren

1. **`baue_kovarianz()` statt Diagonal-Überschreiben.** Der Weg $\Sigma = \mathbf{DCD}$ (Volatilitäten mal Korrelationsmatrix) ist der einzige, der die Zulässigkeit garantiert — und er ist zugleich **interpretierbar**: Man gibt Volatilitäten und Korrelationen vor, also genau die Größen, über die man fachlich diskutiert.
2. **Multistart.** Zehn Startpunkte, und die Spannweite der Ergebnisse wird gemessen. Sind alle gleich, ist das ein starkes Indiz für Konvexität. Weichen sie ab, wissen Sie sofort,

dass Sie einem lokalen Optimum aufsitzen. Diese fünf Zeilen sollten in jedem SLSQP-Projekt stehen.

3. **Analytischer Gradient.** `jac=gradient` spart nicht nur Zeit — die numerische Approximation von Ableitungen ist bei Termen wie $\ln w$ nahe null numerisch heikel.

Zur Skalierung: Mit $\alpha = 1$ ist der Ertragsterm ($\approx 0{,}1$) zehnmal größer als der Varianzterm ($\approx 0{,}01$). Das Modell ist also stark renditegetrieben. Die Aufgabe *Effekt der Gewichtung untersuchen* (Abschnitt 11.7) lässt Sie diese Balance untersuchen — sie ist eine fachliche Entscheidung, keine technische.

11.6 Jenseits der Konvexität: lokale Optima, Multistart und MINLP

Alles bisher in diesem Kapitel stand unter einer Bedingung: **Konvexität**. Sie ist der Grund, warum CVXPY eine Optimalitätsgarantie geben kann und warum die KKT-Bedingungen nicht nur notwendig, sondern auch hinreichend sind.

Dieser Abschnitt handelt davon, was passiert, wenn diese Bedingung fehlt — und das ist in der Praxis oft genug der Fall.

Woher Nicht-Konvexität im Alltag kommt

Nicht aus exotischer Mathematik, sondern aus ganz gewöhnlichen betriebswirtschaftlichen Effekten:

Ursache	Beispiel	Warum nicht konvex
Mengenrabatte	Stückpreis fällt mit der Bestellmenge	Kostenfunktion wird konkav — Sprünge nach unten belohnen große Lose
Skaleneffekte	Stückkosten sinken mit der Losgröße	dasselbe Muster in der Produktion
Produkte von Variablen	„Menge mal Preis“, wenn beide entschieden werden	$x \cdot y$ ist weder konvex noch konkav
Ja/Nein mal Menge	Anlage läuft *und* wie stark	ganzzahlig + nichtlinear = MINLP
Verhältnisse	Auslastungsgrad, Rendite je eingesetztem Euro	Quotienten sind selten konvex

 Merksatz Sobald ein **Rabatt**, ein **Skaleneffekt** oder ein **Produkt zweier Entscheidungen** im Modell steht, ist die Konvexität in Gefahr — und damit die Optimalitätsgarantie. Das ist kein Grund zur Panik, aber ein Grund, das Ergebnis anders zu behandeln.

Was ein lokales Optimum praktisch bedeutet

`scipy.optimize.minimize` verweigert nichts. Es rechnet, meldet `success: True` und liefert eine Zahl. Diese Meldung heißt aber **nicht** „das ist das Optimum“, sondern nur:

> *„Ich bin an einer Stelle angekommen, an der es in keine Richtung mehr bergab geht.“*

Bei einem konvexen Problem ist das dasselbe. Bei einem nicht-konvexen sind es zwei völlig verschiedene Aussagen. Das folgende Programm macht den Unterschied sichtbar.

```
#!/usr/bin/env python3

# Lokale_Optima_Multistart.py
"""
Kapitel QP/NLP: Was passiert, wenn die Konvexitaet fehlt.

CVXPY verweigert nicht-konvexe Probleme - das ist sein Schutzmechanismus.
scipy.optimize.minimize verweigert nichts. Es rechnet, meldet 'success: True'
und liefert ein Ergebnis. Nur ist das Ergebnis dann kein Optimum, sondern
irgendein lokales Minimum, das vom Startpunkt abhaengt.

Beispiel aus dem Einkauf: 700 Tonnen Rohstoff werden auf vier Lieferanten
verteilt. Jeder gewaehrt einen MENGENRABATT - der Stueckpreis faellt, je mehr
man bei ihm bestellt:

    preis_i(q) = basis_i * (1 - rabatt_i * (1 - exp(-q / skala_i)))

Genau das macht die Zielfunktion nicht-konvex: Grosse Bestellungen lohnen
sich ueberproportional, es gibt also mehrere sinnvolle "Cluster"-Loesungen -
und dazwischen schlechtere Taeler.

Das Programm zeigt drei Dinge:
  1. Ein einzelner Lauf liefert ein plausibles Ergebnis - ohne jede Warnung.
  2. 200 Startpunkte foerdern mehrere verschiedene lokale Optima zutage.
  3. Der Unterschied zwischen bestem und schlechtestem betraegt hier 8 %.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
from scipy.optimize import minimize

# Vier Lieferanten
NAMEN = ["Nord AG", "Ost GmbH", "Sued KG", "West SE"]
BASISPREIS = np.array([50.0, 47.0, 53.0, 45.0])     # EUR je Tonne ohne Rabatt
MAX_RABATT = np.array([0.30, 0.18, 0.35, 0.12])     # hoechstmoeglicher Rabatt
RABATT_SKALA = np.array([120.0, 260.0, 90.0, 300.0])  # wie schnell er greift
KAPAZITAET = np.array([400.0, 400.0, 400.0, 400.0])
```

```
BEDARF = 700.0

RNG = np.random.default_rng(0)


def gesamtkosten(menge: np.ndarray) -> float:
    """Einkaufskosten bei mengenabhaengigem Stueckpreis.

    Der Rabatt waechst mit der Bestellmenge und laeuft gegen MAX_RABATT.
    Dadurch ist der Stueckpreis fallend - und die Gesamtkostenfunktion
    nicht mehr konvex.
    """
    stueckpreis = BASISPREIS * (1 - MAX_RABATT * (1 - np.exp(-menge / RABATT_SKALA)))
    return float(stueckpreis @ menge)


NEBENBEDINGUNGEN = [{"type": "eq", "fun": lambda q: q.sum() - BEDARF}]
GRENZEN = [(0.0, k) for k in KAPAZITAET]


def optimiere_von(startpunkt: np.ndarray):
    """Ein Lauf von einem gegebenen Startpunkt aus."""
    return minimize(gesamtkosten, startpunkt, method="SLSQP",
                    bounds=GRENZEN, constraints=NEBENBEDINGUNGEN)


def zufaelliger_start() -> np.ndarray:
    """Zufaellige Aufteilung, die den Bedarf bereits erfuellt."""
    anteil = RNG.random(len(NAMEN))
    return anteil / anteil.sum() * BEDARF


def zeige_plan(titel: str, menge: np.ndarray, kosten: float) -> None:
    print(f"\n{titel}")
    for name, m in zip(NAMEN, menge):
        anteil = m / BEDARF * 100
        print(f"    {name:<10} {m:7.1f} t  ({anteil:4.1f} %)")
    print(f"    {'Gesamtkosten':<10} {kosten:9,.2f} EUR")


if __name__ == "__main__":
    print("=" * 78)
    print("  NICHT-KONVEX: DERSELBE CODE, VERSCHIEDENE ERGEBNISSE")
    print("=" * 78)
    print(f"{BEDARF:.0f} t Rohstoff auf {len(NAMEN)} Lieferanten mit Mengenrabatt.")

    # --- 1. Ein einziger Lauf, so wie man es zuerst schreibt -------------
    erster = optimiere_von(np.full(len(NAMEN), BEDARF / len(NAMEN)))
```

```
print(f"\n[1] EIN Lauf, Startpunkt 'gleichmaessig verteilt'")
print(f"    scipy meldet: success={erster.success}, "
      f"'{erster.message}'")
zeige_plan("    Ergebnis:", erster.x, erster.fun)
print("\n    Nichts an dieser Ausgabe deutet darauf hin, dass etwas fehlt.")

# --- 1b. Der kaufmaennisch naheliegende Startpunkt --------------------
# "Kaufe bei den beiden Lieferanten mit dem guenstigsten Basispreis" -
# West SE (45) und Ost GmbH (47), jeweils bis zur Kapazitaetsgrenze.
guenstigste = np.argsort(BASISPREIS)[:2]
kaufmaennisch = np.zeros(len(NAMEN))
rest = BEDARF
for i in guenstigste:
    kaufmaennisch[i] = min(KAPAZITAET[i], rest)
    rest -= kaufmaennisch[i]
zweiter = optimiere_von(kaufmaennisch)
print(f"\n[1b] EIN Lauf, Startpunkt 'die zwei mit dem guenstigsten Basispreis'")
print(f"    scipy meldet: success={zweiter.success}")
zeige_plan("    Ergebnis:", zweiter.x, zweiter.fun)
print(f"\n    Dasselbe Programm, derselbe Aufruf, ein anderer Startpunkt -")
print(f"    und {zweiter.fun - erster.fun:,.2f} EUR Unterschied "
      f"({(zweiter.fun / erster.fun - 1) * 100:.1f} %).")

# --- 2. Multistart: dasselbe Problem, viele Startpunkte ---------------
laeufe = []
for _ in range(200):
    ergebnis = optimiere_von(zufaelliger_start())
    if ergebnis.success:
        laeufe.append((float(ergebnis.fun), ergebnis.x))

# Ergebnisse, die sich um weniger als 1 Cent unterscheiden, sind dasselbe
# lokale Optimum - zusammenfassen, sonst zaehlt man Rundungsrauschen.
optima: list[tuple[float, np.ndarray]] = []
for wert, plan in sorted(laeufe, key=lambda t: t[0]):
    if not optima or abs(wert - optima[-1][0]) > 0.01:
        optima.append((wert, plan))

print("\n" + "=" * 78)
print(f"[2] 200 zufaellige Startpunkte -> {len(laeufe)} erfolgreiche Laeufe")
print(f"    darunter {len(optima)} VERSCHIEDENE lokale Optima:")
print()
print(f"    {'Rang':>5} {'Kosten':>13} {'Abstand zum besten':>20}   Aufteilung (t)")
print("    " + "-" * 70)
bester = optima[0][0]
for rang, (wert, plan) in enumerate(optima, start=1):
    abstand = (wert / bester - 1) * 100
    aufteilung = " ".join(f"{m:5.0f}" for m in plan)
    print(f"    {rang:>5} {wert:>13,.2f} {abstand:>19.2f} %   {aufteilung}")
```

```
    # --- 3. Was das kostet ------------------------------------------------
    schlechtester = optima[-1]
    print("\n" + "=" * 78)
    print("  WAS AUF DEM SPIEL STEHT")
    print("=" * 78)
    zeige_plan("Bester gefundener Plan:", optima[0][1], optima[0][0])
    zeige_plan("Schlechtestes lokales Optimum:", schlechtester[1], schlechtester[0])
    unterschied = schlechtester[0] - optima[0][0]
    print(f"\n  Unterschied: {unterschied:,.2f} EUR "
          f"({unterschied / optima[0][0] * 100:.1f} %)")
    print(f"\n  Lauf [1]  (gleichmaessiger Start):     {erster.fun:>10,.2f} EUR")
    print(f"  Lauf [1b] (kaufmaennischer Start):    {zweiter.fun:>10,.2f} EUR"
          f"   <- {(zweiter.fun / optima[0][0] - 1) * 100:.1f} % ueber dem besten")
    print()
    print("  Bemerkenswert: Der kaufmaennisch NAHELIEGENDE Startpunkt fuehrt in")
    print("  das schlechteste Ergebnis von allen - schlechter als jedes der 200")
    print("  zufaellig gefundenen lokalen Optima. Wer beim guenstigsten")
    print("  Basispreis anfaengt, uebersieht, dass hier der Mengenrabatt")
    print("  entscheidet und nicht der Listenpreis.")

    print()
    print("Drei Konsequenzen fuer die Praxis:")
    print("  1. 'success: True' heisst bei nicht-konvexen Problemen NICHT 'optimal'.")
    print("     Es heisst nur: 'Ich bin an einer Stelle angekommen, an der es in")
    print("     keine Richtung mehr bergab geht.'")
    print("  2. Ein einzelner Lauf ist wertlos. Nehmen Sie viele Startpunkte und")
    print("     berichten Sie die STREUUNG mit - sie ist Ihre einzige Auskunft")
    print("     darueber, wie zerklueftet die Landschaft ist.")
    print("  3. Auch Multistart liefert KEINE Garantie. Dass hier nichts unter")
    print(f"      {bester:,.2f} EUR gefunden wurde, beweist nicht, dass es nichts gibt.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  NICHT-KONVEX: DERSELBE CODE, VERSCHIEDENE ERGEBNISSE
==============================================================================
700 t Rohstoff auf 4 Lieferanten mit Mengenrabatt.

[1] EIN Lauf, Startpunkt 'gleichmaessig verteilt'
    scipy meldet: success=True, 'Optimization terminated successfully'

    Ergebnis:
    Nord AG      343.5 t  (49.1 %)
    Ost GmbH       0.0 t  ( 0.0 %)
    Sued KG      356.5 t  (50.9 %)
    West SE        0.0 t  ( 0.0 %)
```

```
    Gesamtkosten 24,724.19 EUR

    Nichts an dieser Ausgabe deutet darauf hin, dass etwas fehlt.

[1b] EIN Lauf, Startpunkt 'die zwei mit dem guenstigsten Basispreis'
    scipy meldet: success=True

    Ergebnis:
    Nord AG        0.0 t  ( 0.0 %)
    Ost GmbH     400.0 t  (57.1 %)
    Sued KG        0.0 t  ( 0.0 %)
    West SE      300.0 t  (42.9 %)
    Gesamtkosten 28,618.55 EUR

    Dasselbe Programm, derselbe Aufruf, ein anderer Startpunkt -
    und 3,894.36 EUR Unterschied (15.8 %).

================================================================================
[2] 200 zufaellige Startpunkte -> 200 erfolgreiche Laeufe
    darunter 5 VERSCHIEDENE lokale Optima:

     Rang        Kosten   Abstand zum besten   Aufteilung (t)
    ------------------------------------------------------------------------
        1     24,724.19                0.00 %     344     0   356     0
        2     26,229.68                6.09 %       0   300   400     0
        3     26,343.10                6.55 %       0     0   400   300
        4     26,576.58                7.49 %     400   300     0     0
        5     26,690.01                7.95 %     400     0     0   300

================================================================================
  WAS AUF DEM SPIEL STEHT
================================================================================

Bester gefundener Plan:
    Nord AG      343.5 t  (49.1 %)
    Ost GmbH       0.0 t  ( 0.0 %)
    Sued KG      356.5 t  (50.9 %)
    West SE        0.0 t  ( 0.0 %)
    Gesamtkosten 24,724.19 EUR

Schlechtestes lokales Optimum:
    Nord AG      400.0 t  (57.1 %)
    Ost GmbH       0.0 t  ( 0.0 %)
    Sued KG        0.0 t  ( 0.0 %)
    West SE      300.0 t  (42.9 %)
    Gesamtkosten 26,690.01 EUR

  Unterschied: 1,965.82 EUR (8.0 %)
```

```
  Lauf [1]  (gleichmaessiger Start):      24,724.19 EUR
  Lauf [1b] (kaufmaennischer Start):     28,618.55 EUR   <- 15.8 % ueber dem besten

  Bemerkenswert: Der kaufmaennisch NAHELIEGENDE Startpunkt fuehrt in
  das schlechteste Ergebnis von allen - schlechter als jedes der 200
  zufaellig gefundenen lokalen Optima. Wer beim guenstigsten
  Basispreis anfaengt, uebersieht, dass hier der Mengenrabatt
  entscheidet und nicht der Listenpreis.

Drei Konsequenzen fuer die Praxis:
  1. 'success: True' heisst bei nicht-konvexen Problemen NICHT 'optimal'.
     Es heisst nur: 'Ich bin an einer Stelle angekommen, an der es in
     keine Richtung mehr bergab geht.'
  2. Ein einzelner Lauf ist wertlos. Nehmen Sie viele Startpunkte und
     berichten Sie die STREUUNG mit - sie ist Ihre einzige Auskunft
     darueber, wie zerklueftet die Landschaft ist.
  3. Auch Multistart liefert KEINE Garantie. Dass hier nichts unter
     24,724.19 EUR gefunden wurde, beweist nicht, dass es nichts gibt.
================================================================================
```

□ Code-Durchgang

Stelle	Was passiert	Warum es zählt
`1 - MAX_RABATT * (1 - exp(-menge / SKALA))`	Stückpreis fällt mit der Menge	Ein völlig normaler Staffelrabatt — und genau er zerstört die Konvexität. Man braucht keine exotische Mathematik dafür.
Zusammenfassen mit `> 0.01`	Rundungsrauschen von echten Optima trennen	Ohne diesen Schritt zählt man 200 „verschiedene“ Optima, die sich in der zehnten Nachkommastelle unterscheiden.
200 zufällige Startpunkte	Multistart	Die einzige praktikable Auskunft darüber, wie zerklüftet die Landschaft ist — und trotzdem **kein Beweis**.
Lauf `[1b]`	der kaufmännisch naheliegende Startpunkt	Die eigentliche Pointe: Er ist der **schlechteste von allen**.

Die Zahlen im Klartext

- Der gleichmäßige Startpunkt findet **24 724 €** — zufällig das beste gefundene Optimum.
- Der kaufmännisch naheliegende Startpunkt („die zwei mit dem günstigsten Basispreis“) findet **28 619 €** — 15,8 % schlechter, und schlechter als *jedes* der 200 zufällig gefundenen lokalen Optima.
- Beide Läufe melden `success: True`. Nichts an der Ausgabe unterscheidet sie.

Der Grund für das schlechte Abschneiden des „vernünftigen“ Starts ist lehrreich: Wer beim günstigsten **Listenpreis** beginnt, folgt genau dem Kriterium, das hier nicht entscheidet. Ausschlaggebend ist der Rabatt — und Süd KG hat mit 35 % den höchsten, obwohl sein Basispreis der teuerste ist. Eine plausible Heuristik führt den Optimierer damit zielsicher ins falsche Tal.

Typische Fehler bei nicht-konvexen Problemen

- **`success: True` als „optimal“ lesen.** Es heißt nur „konvergiert“.
- **Einen einzigen Lauf berichten.** Ohne Streuung über mehrere Startpunkte ist die Zahl nicht einordbar.
- **Den „vernünftigen“ Startpunkt für den besten halten.** Er ist oft der schlechteste, weil er einer Heuristik folgt, die genau das ignoriert, was das Problem schwer macht.
- **Multistart für einen Beweis halten.** Dass nichts Besseres gefunden wurde, beweist nicht, dass es nichts Besseres gibt.

Ausblick: MINLP und globale Solver

Wird zusätzlich noch ganzzahlig entschieden — „welche Anlage läuft überhaupt“ *und* „wie stark“ —, entsteht ein **MINLP** (*Mixed-Integer Nonlinear Program*), die schwierigste der in diesem Buch behandelten Klassen. Dafür gibt es eigene Werkzeuge:

Werkzeug	Art	Was es leistet
Ipopt (über `cyipopt` oder Pyomo)	lokaler NLP-Solver	Innere-Punkte-Verfahren für große, glatte NLPs mit tausenden Variablen — deutlich leistungsfähiger als `scipy`, aber ebenfalls **nur lokal**
Bonmin, **Couenne**	MINLP	Bonmin lokal, Couenne mit globaler Garantie für viele Klassen
SCIP, **BARON**, **Gurobi**	global	Beweisbar globales Optimum, mit entsprechendem Rechenaufwand
Pyomo (Abschnitt 3.7)	Modellierungsschicht	Bindet alle oben genannten an, ohne dass das Modell umgeschrieben werden muss

Zu diesem Abschnitt gehört kein lauffähiges Programm. Ipopt, Bonmin und Couenne sind C++-Pakete, die über die Python-Installation hinaus systemweit eingerichtet werden

müssen (`conda install -c conda-forge ipopt cyipopt` oder eine Distributionspaketquelle). Anders als bei allen übrigen Programmen dieses Buches konnten wir den Code deshalb nicht auf jedem Zielsystem ausführen — und drucken hier bewusst **keine** Ausgabe ab, die wir nicht selbst erzeugt haben. Der folgende Ausschnitt zeigt die Anbindung; prüfen Sie das Ergebnis auf Ihrem System selbst nach.

```
# Ipopt ueber Pyomo - Anbindungsmuster, NICHT ausgefuehrt.
# Voraussetzung:  conda install -c conda-forge ipopt
import pyomo.environ as pyo

modell = pyo.ConcreteModel()
modell.q = pyo.Var(range(4), domain=pyo.NonNegativeReals, bounds=(0, 400))
modell.bedarf = pyo.Constraint(expr=sum(modell.q[i] for i in range(4)) == 700)
modell.ziel = pyo.Objective(
    expr=sum(BASISPREIS[i] * (1 - MAX_RABATT[i]
                              * (1 - pyo.exp(-modell.q[i] / RABATT_SKALA[i])))
             * modell.q[i] for i in range(4)),
    sense=pyo.minimize)

loeser = pyo.SolverFactory("ipopt")
if loeser.available(exception_flag=False):
    ergebnis = loeser.solve(modell)
    print(ergebnis.solver.termination_condition)
else:
    print("Ipopt ist nicht installiert - siehe Hinweis oben.")
```

Die Strategie bleibt in jedem Fall dieselbe: Ein lokaler Solver wie Ipopt löst *schneller* und *größer* als `scipy` — er löst aber nicht *globaler*. Multistart bleibt nötig. Erst ein globaler Solver (SCIP, Couenne, BARON) ersetzt ihn, und den bezahlt man mit Rechenzeit, die um Größenordnungen höher liegt.

 Merksatz Bei nicht-konvexen Problemen ist die ehrliche Berichterstattung wichtiger als die letzte Nachkommastelle: *„Bester gefundener Wert 24 724 € aus 200 Startpunkten; die Streuung reicht bis 26 690 €.“* Das ist eine belastbare Aussage. *„Das Optimum liegt bei 24 724,19 €“* ist es nicht.

11.7 Übungsaufgaben

Lösungen: Abschnitt A.11.

Aufgabe 11.1 — Konvexität einordnen. $\mathbf{P}$ hat die Eigenwerte $(2{,}5;\ 0{,}0;\ 1{,}3)$. Ist das QP konvex? Streng konvex? Ist die Lösung eindeutig? Was bedeutet der Eigenwert 0 anschaulich?

Aufgabe 11.2 — Komplementärer Schlupf. Ein QP liefert $w = (0{,}4;\ 0{,}0;\ 0{,}6)$ mit Bedingung $w \geq 0$ und Multiplikatoren $\lambda = (0{,}0;\ 0{,}03;\ 0{,}0)$. Ist das mit KKT verträglich? Was sagt $\lambda_2 = 0{,}03$ wirtschaftlich?

Aufgabe 11.3 — KKT von Hand. Lösen Sie mit KKT vollständig von Hand:

$$\min\ x_1^2 + x_2^2 \qquad \text{u. d. N.} \qquad x_1 + x_2 \geq 4$$

Geben Sie x^*, λ^* und die Interpretation von λ^* an. Prüfen Sie mit CVXPY.

Aufgabe 11.4 □□ — Ungültige Kovarianzmatrix erkennen. Prüfen Sie, ob folgende Korrelationsmatrix möglich ist, und begründen Sie:

$$\mathbf{C} = \begin{pmatrix} 1{,}0 & 0{,}9 & -0{,}9 \\ 0{,}9 & 1{,}0 & 0{,}9 \\ -0{,}9 & 0{,}9 & 1{,}0 \end{pmatrix}$$

(Tipp: Wenn A stark mit B korreliert und B stark mit C, kann A dann stark **negativ** mit C korrelieren?)

Aufgabe 11.5 □□□ — Effekt der Gewichtung untersuchen. Variieren Sie in `Entropie_Maximierte_Allokation.py` systematisch $\alpha \in \{0{,}1;\ 0{,}5;\ 1;\ 5\}$ und $\beta \in \{0;\ 0{,}005;\ 0{,}015;\ 0{,}05\}$. Stellen Sie für jede Kombination Rendite, Volatilität und Entropie in einer Tabelle dar. (a) Wie verändert β die Konzentration im Krypto-Titel? (b) Ab welchem β nähert sich die Lösung der Gleichgewichtung? (c) Was ist Ihre Empfehlung — und mit welcher Begründung würden Sie sie einem Anlageausschuss vorlegen?

Aufgabe 11.6 □□□ — Nicht-Konvexität demonstrieren. Bauen Sie bewusst eine ungültige „Kovarianzmatrix“ (per A@A.T mit anschließend überschriebener Diagonale) und lösen Sie das Modell mit 20 Startpunkten. Dokumentieren Sie: Wie viele verschiedene Optima entstehen? Wie groß ist der Unterschied zwischen bestem und schlechtestem Ergebnis? Kann die „Portfoliovarianz“ negativ werden?

11.8 Finde den Denkfehler

□ Finde den Denkfehler: Die Kovarianzmatrix aus dem Controlling

Ein Analyst soll das Risiko eines Portfolios aus drei Anlagen minimieren. Die Korrelationen hat er aus drei verschiedenen Quartalsberichten zusammengetragen — jede für sich plausibel:

- A und B laufen stark gleich: $\rho_{AB} = 0{,}9$
- B und C laufen stark gleich: $\rho_{BC} = 0{,}9$
- A und C laufen **gegen**einander: $\rho_{AC} = -0{,}9$

Alle drei schwanken mit 20 %. Leerverkäufe sind in seinem Mandat erlaubt, die Gewichte dürfen also negativ werden; sie müssen sich nur zu 100 % summieren.

Sein erster Versuch mit CVXPY bricht ab:

```
DCPError: Problem does not follow DCP rules.
The objective is not DCP. Its following subexpressions are not: QuadForm(...)
```

Er hält das für eine Einschränkung der Bibliothek und weicht auf `scipy` aus:

```
import numpy as np
from scipy.optimize import minimize

R = np.array([[ 1.0,  0.9, -0.9],
              [ 0.9,  1.0,  0.9],
```

```
              [-0.9,  0.9,  1.0]])
S = 0.20**2 * R

varianz = lambda w: float(w @ S @ w)
ergebnis = minimize(varianz, np.array([0.4, 0.3, 0.3]),
                    constraints=[{"type": "eq", "fun": lambda w: w.sum() - 1}],
                    bounds=[(-2, 3)] * 3)
print(np.round(ergebnis.x, 3), round(ergebnis.fun, 4), ergebnis.success)
```

Ausgabe:

```
[ 1.5 -2.   1.5] -0.254 True
```

Ein Portfolio mit einer **Varianz von** $-0{,}254$. Der Analyst notiert erfreut ein „risikofreies Portfolio mit negativer Schwankung“.

Ihre Aufgabe: (a) Können A und B gleichlaufen, B und C gleichlaufen — und A und C zugleich gegenläufig sein? Prüfen Sie es, ohne zu rechnen, an einem anschaulichen Beispiel. (b) Berechnen Sie die Eigenwerte von **R** mit `np.linalg.eigvalsh(R)`. Was sagt das Vorzeichen? (c) Warum ist die negative Varianz kein Fehler von `scipy`, sondern die logische Folge der Eingabe — und warum meldet es trotzdem `success: True`? (d) Warum war die Fehlermeldung von CVXPY die **hilfreichste** Zeile des ganzen Vorgangs, und was tut man stattdessen?

Auflösung: Abschnitt A.11.

 Merksatz Eine Kovarianzmatrix ist kein Behälter für einzeln geschätzte Zahlen. Sie ist ein **geometrisches Objekt**: Ihre Einträge hängen voneinander ab, und nicht jede Kombination von Korrelationen existiert überhaupt. Zusammengetragene Korrelationen aus verschiedenen Quellen sind fast nie widerspruchsfrei — das ist einer der häufigsten Fehler in der Risikomodellierung überhaupt.

11.9 Micro-Quiz

 Micro-Quiz 11: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. CVXPY lehnt Ihr Modell mit `DCPError: Problem does not follow DCP rules` ab. Was ist die richtige Reaktion? (a) Auf `scipy.optimize.minimize` ausweichen, das die Formulierung akzeptiert. (b) Die Meldung ernst nehmen: CVXPY sagt, dass es für diese Formulierung **keine Optimalitätsgarantie** geben kann. Entweder das Problem konvex umformulieren — oder bewusst auf ein lokales Verfahren mit Multistart wechseln. (c) Die Toleranzen lockern.

2. `scipy.optimize.minimize` meldet für ein Modell mit Mengenrabatten `success: True`. Was wissen Sie damit über die gefundene Lösung? (a) Sie ist optimal. (b) Sie ist ein lokales Minimum — es geht von dort in keine Richtung mehr bergab. Ob es anderswo

ein besseres gibt, ist damit offen. (c) Sie ist zulässig, aber möglicherweise nicht einmal ein lokales Minimum.

3. Der Lagrange-Multiplikator einer bindenden Nebenbedingung beträgt $\lambda^* = 4$. Was bedeutet das? (a) Die Nebenbedingung wird viermal verletzt. (b) Vier Einheiten der Ressource sind übrig. (c) Lockert man die Bedingung um eine kleine Einheit, verbessert sich der Zielwert um etwa 4 — es ist der Schattenpreis, genau wie im LP.

11.10 Selbsttest

Antworten: Anhang A.

1. Worin unterscheiden sich „positiv semidefinit“ und „positiv definit“ in ihren Folgen für die Lösung?
2. Was besagt die Stationaritätsbedingung anschaulich?
3. Warum müssen die Multiplikatoren von Ungleichungen nichtnegativ sein?
4. Wie konstruiert man garantiert eine gültige Kovarianzmatrix?
5. Warum sollte man SLSQP immer mit mehreren Startpunkten laufen lassen?

11.11 Zusammenfassung

- **Quadratische Programme** sind die Brücke zur Portfoliotheorie: Risiko ist eine quadratische Form $\mathbf{w}^\top \Sigma \mathbf{w}$.
- **Semidefinit** □ **konvex** (lokales = globales Optimum), **definit** □ **streng konvex** (Optimum eindeutig). Der Unterschied ist praktisch relevant, sobald $N > T$.
- **KKT** verallgemeinert Lagrange auf Ungleichungen. Die Multiplikatoren sind die Schattenpreise aus Kapitel 5 — nur allgemeiner.
- **Komplementärer Schlupf** ist der beste Selbsttest für jede Optimierung.
- **Kovarianzmatrizen immer prüfen** und aus Volatilitäten plus Korrelationsmatrix konstruieren, nie durch Manipulation einzelner Einträge.
- **SLSQP liefert nur lokale Optima.** Multistart ist Pflicht, nicht Kür — und `success: True` heißt „konvergiert“, nicht „optimal“. Im Kapitelbeispiel liegen fünf lokale Optima 8 % auseinander, und ausgerechnet der kaufmännisch naheliegende Startpunkt ist der schlechteste.
- **Nicht-Konvexität kommt aus dem Alltag**, nicht aus der Theorie: Mengenrabatte, Skaleneffekte, Produkte zweier Entscheidungen. Wo eines davon im Modell steht, ist die Optimalitätsgarantie weg.
- **Berichten Sie bei nicht-konvexen Problemen die Streuung mit**, nicht die letzte Nachkommastelle. „Bester Wert aus 200 Startpunkten, Spanne bis …“ ist belastbar, ein einzelner Wert ist es nicht.

Ausblick. Kapitel 12 gibt die Annahme auf, dass die Parameter überhaupt bekannt sind. Wir lernen, mit Szenarien, Erwartungswerten und Worst Cases umzugehen.

Kapitel 12: Optimierung unter Unsicherheit — Monte-Carlo, Stochastik, Robustheit

Kapitel auf einen Blick

Worum geht es? Bisher waren alle Parameter bekannt. In der Realität sind Nachfrage, Rendite und Fahrzeit **Zufallsgrößen**. Dieses Kapitel zeigt vier Wege, damit umzugehen — und warum „einfach den Mittelwert einsetzen" der schlechteste davon ist.

Voraussetzungen: Kapitel 5, Kapitel 11. Grundbegriffe der Statistik (Erwartungswert, Quantil).

Danach können Sie: Ein zweistufiges stochastisches Modell aufstellen, Monte-Carlo zur Risikomessung einsetzen, ein robustes Modell für den Worst Case formulieren, eine Wahrscheinlichkeitszusage („mit 95 % Sicherheit") als lösbare Nebenbedingung schreiben — und erkennen, wann Rechnen mit Mittelwerten zulässig ist und wann es systematisch daneben-geht.

Zeitbedarf: ca. 6,5 Stunden.

Programme:
`Fluch_des_Durchschnitts.py`
`Monte_Carlo.py`
`Stochastische_Optimierung.py`
`Robuste_Optimierung.py`
`Chance_Constraints.py`

Notebook: `Notebooks_04/unsicherheit.ipynb`

12.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Wie viele Ersatzteile bestellen?

Ein Betrieb bestellt vor der Wintersaison Ersatzteile — **einmal**, danach ist die Lieferkette bis zum Frühjahr zu. Der Bedarf schwankt, im Mittel werden 30 Stück gebraucht (Poisson-verteilt).

Fall	Kosten je Stück
Zu viel bestellt — Teil bleibt liegen und wird abgeschrieben	120 €
Zu wenig bestellt — Maschine steht bis zur Nachlieferung	1 400 €

Wie viele bestellen Sie?

```python
import numpy as np

KOSTEN_ZUVIEL, KOSTEN_ZUWENIG = 120.0, 1400.0
bedarf = np.random.default_rng(7).poisson(30, 200_000)     # 200.000 Winter

def kosten(menge):
    zuviel = np.maximum(menge - bedarf, 0)
    zuwenig = np.maximum(bedarf - menge, 0)
    return (KOSTEN_ZUVIEL * zuviel + KOSTEN_ZUWENIG * zuwenig).mean()

kandidaten = np.arange(20, 61)
werte = np.array([kosten(m) for m in kandidaten])
print("Optimum:", kandidaten[werte.argmin()], "Stueck,", f"{werte.min():,.0f}
 ↪  EUR")
print("Mittlerer Bedarf 30:", f"{kosten(30):,.0f} EUR")
```

Ausgabe:

```
Optimum: 38 Stueck, 1,270 EUR
Mittlerer Bedarf 30: 3,306 EUR
```

Die naheliegende Antwort — „im Mittel brauchen wir 30, also bestellen wir 30“ — kostet das 2,6-Fache. 3 306 € statt 1 270 €, ein Aufschlag von 160 %.

Der Grund liegt in der **Asymmetrie**: Ein Teil zu viel kostet 120 €, ein Teil zu wenig kostet 1 400 €. Beides ist nicht gleich schlimm, also darf man auch nicht in der Mitte landen. Man muss sich bewusst auf die günstigere Seite des Fehlers stellen.

Wie weit? Dafür gibt es eine geschlossene Formel — das **kritische Verhältnis**:

$$q^* = \frac{c_{\text{zu wenig}}}{c_{\text{zu wenig}} + c_{\text{zu viel}}} = \frac{1400}{1400 + 120} = 0{,}921$$

Formel-Übersetzer

Mathematik	Alltagssprache
$c_{\text{zu wenig}}$	„Was kostet mich ein Stück, das ich gebraucht hätte und nicht habe?“
$c_{\text{zu viel}}$	„Was kostet mich ein Stück, das ich habe und nicht brauche?“
$q^* = 0{,}921$	„Bestelle so viel, dass der Bedarf in **92,1 %** aller Winter gedeckt ist.“
Bestellmenge $= F^{-1}(q^*)$	„Nimm das 92,1-%-Quantil der Bedarfsverteilung“ — hier 38 Stück.

In einem Satz: *Je teurer der Engpass gegenüber dem Überbestand, desto weiter über den Mittelwert hinaus muss man bestellen.*

Das Quantil zu 0,921 liegt bei genau **38 Stück** — dasselbe Ergebnis, das die Simulation oben gefunden hat. Zwei völlig verschiedene Wege, dieselbe Zahl.

 Merksatz Die optimale Entscheidung unter Unsicherheit ist fast nie die Entscheidung, die für den Mittelwert optimal wäre. Sie hängt davon ab, **welcher der beiden Fehler teurer ist** — und verschiebt sich zu der Seite, auf der Irren billiger ist. Wer mit dem Mittelwert plant, hat diese Frage nie gestellt.

Dieses Muster heißt **Newsvendor-Problem** (deutsch: Zeitungsjungen-Problem) und taucht überall dort auf, wo einmal entschieden und danach beobachtet wird: Ersatzteile, Frischwaren, Saisonware, Personalreserve, Kraftwerksvorhaltung. Der Rest dieses Kapitels verallgemeinert es — von einer Zahl auf ganze Modelle, und von einer bekannten Verteilung auf den Fall, dass man nicht einmal die kennt.

12.2 Lernziele

Nach diesem Kapitel können Sie …

1. … den **Fluch des Durchschnitts** (*Flaw of Averages*) an einem Beispiel erklären.
2. … ein **zweistufiges stochastisches Programm** mit Recourse formulieren.
3. … Monte-Carlo einsetzen, um Verteilungen statt Punktschätzungen zu bewerten.
4. … ein **robustes** Modell mit Unsicherheitsmenge aufstellen und seinen Preis beziffern.
5. … eine **Wahrscheinlichkeitszusage** („mit 95 % Sicherheit“) als Chance Constraint schreiben — analytisch als Kegelbedingung oder szenariobasiert mit Big-M — und den Preis eines Prozentpunkts Sicherheit beziffern.
6. … begründen, welcher der vier Ansätze für ein gegebenes Problem passt.
7. … das **kritische Verhältnis** eines Newsvendor-Problems aufstellen und daraus die optimale Bestellmenge als Quantil ablesen.
8. … eine Terminzusage als Quantil formulieren statt als Mittelwert.

12.3 Der Fluch des Durchschnitts

Der naheliegende Umgang mit Unsicherheit lautet: „Wir setzen den Erwartungswert ein und rechnen deterministisch.“ Das ist **fast immer falsch** — und zwar nicht ungenau, sondern systematisch verzerrt.

 Handrechnung 12.1: Warum der Mittelwert in die Irre führt

Ein Betreiber braucht Serverkapazität. Der Bedarf ist:

Szenario	Wahrscheinlichkeit	Bedarf
Ruhig	50 %	100
Volatil	30 %	250
Crash	20 %	500

Erwartungswert: $0{,}5 \cdot 100 + 0{,}3 \cdot 250 + 0{,}2 \cdot 500 = 50 + 75 + 100 = 225$.

Kosten: Vorabkauf 40 €/Einheit; fehlende Kapazität muss kurzfristig für 120 € zugekauft werden; überschüssige Kapazität kostet 5 € Verwaltung.

Naive Planung mit $x = 225$: * Ruhig (Bedarf 100): 125 **zu viel** $\rightarrow 40 \cdot 225 + 5 \cdot 125 = 9000 + 625 = 9625$ * Volatil (250): 25 **zu wenig** $\rightarrow 9000 + 120 \cdot 25 = 9000 + 3000 = 12\,000$ * Crash (500): 275 zu wenig $\rightarrow 9000 + 120 \cdot 275 = 9000 + 33\,000 = 42\,000$ * **Erwartete Kosten:** $0{,}5 \cdot 9625 + 0{,}3 \cdot 12\,000 + 0{,}2 \cdot 42\,000 = 4812{,}5 + 3600 + 8400 = \mathbf{16\,812{,}50}$

Stochastisch optimale Planung mit $x = 250$: * Ruhig: 150 zu viel $\rightarrow 10\,000 + 750 = 10\,750$ * Volatil: passt genau $\rightarrow 10\,000$ * Crash: 250 zu wenig $\rightarrow 10\,000 + 30\,000 = 40\,000$ * **Erwartete Kosten:** $0{,}5 \cdot 10\,750 + 0{,}3 \cdot 10\,000 + 0{,}2 \cdot 40\,000 = 5375 + 3000 + 8000 = \mathbf{16\,375{,}00}$

Die naive Planung kostet 437,50 € mehr — rund 2,7 % Aufschlag für das Einsetzen eines Mittelwerts, den es als Szenario gar nicht gibt. Beachten Sie: Der Aufschlag wirkt klein, weil bei nur drei Szenarien 225 und 250 nahe beieinanderliegen. Entscheidend ist die **Richtung**: Das Optimum liegt systematisch **über** dem Mittelwert, und zwar bei genau jenem Szenariowert, an dem die Kostenbalance kippt.

Dass $x = 250$ tatsächlich das Minimum ist und nicht nur eine plausible Vermutung, zeigt ein vollständiger Scan über alle möglichen Kapazitäten:

```
#!/usr/bin/env python3

# Fluch_des_Durchschnitts.py
"""
Kapitel Unsicherheit: Der Fluch des Durchschnitts (die Handrechnung dazu) als Scan ueber alle
moeglichen Kapazitaeten - zeigt, dass das Optimum nicht beim Mittelwert liegt.
"""

import numpy as np

SZENARIEN = np.array([100, 250, 500])
WAHRSCHEINLICHKEITEN = np.array([0.5, 0.3, 0.2])
PREIS_VORAB = 40.0
PREIS_ZUKAUF = 120.0
PREIS_VERWALTUNG = 5.0

ERWARTUNGSWERT = float(SZENARIEN @ WAHRSCHEINLICHKEITEN)


def erwartete_kosten(x):
    kosten = np.where(
        SZENARIEN >= x,
        PREIS_VORAB * x + PREIS_ZUKAUF * (SZENARIEN - x),
        PREIS_VORAB * x + PREIS_VERWALTUNG * (x - SZENARIEN),
    )
    return float(kosten @ WAHRSCHEINLICHKEITEN)
```

```
if __name__ == "__main__":
    print("=" * 70)
    print("  DER FLUCH DES DURCHSCHNITTS")
    print("=" * 70)
    print(f"Erwarteter Bedarf (Mittelwert): {ERWARTUNGSWERT:.1f}")
    print(f"Kosten bei naiver Planung x={ERWARTUNGSWERT:.0f}: "
          f"{erwartete_kosten(ERWARTUNGSWERT):,.2f} EUR\n")

    x_werte = np.arange(0, 501, 1)
    kosten_werte = np.array([erwartete_kosten(x) for x in x_werte])
    x_optimal = x_werte[np.argmin(kosten_werte)]
    kosten_optimal = kosten_werte.min()

    print(f"Optimales x (durch Scan gefunden): {x_optimal}")
    print(f"Kosten beim Optimum:                {kosten_optimal:,.2f} EUR")
    print(f"Ersparnis gegenueber naiver Planung: "
          f"{erwartete_kosten(ERWARTUNGSWERT) - kosten_optimal:,.2f} EUR\n")

    print("Kosten fuer ausgewaehlte x zum Vergleich:")
    for x in [100, 225, 250, 300, 500]:
        markierung = "  <- Mittelwert" if x == 225 else ("  <- Optimum" if x == 250 else "")
        print(f"  x={x:4d}: {erwartete_kosten(x):>12,.2f} EUR{markierung}")

    print("\nDas Optimum liegt exakt auf einem Szenariowert (250 = 'Volatil'),")
    print("nicht beim Mittelwert 225 - typisch fuer asymmetrische Kostenfunktionen.")
```

Erwartete Ausgabe:

```
======================================================================
  DER FLUCH DES DURCHSCHNITTS
======================================================================
Erwarteter Bedarf (Mittelwert): 225.0
Kosten bei naiver Planung x=225: 16,812.50 EUR

Optimales x (durch Scan gefunden): 250
Kosten beim Optimum:                16,375.00 EUR
Ersparnis gegenueber naiver Planung: 437.50 EUR

Kosten fuer ausgewaehlte x zum Vergleich:
  x= 100:    19,000.00 EUR
  x= 225:    16,812.50 EUR  <- Mittelwert
  x= 250:    16,375.00 EUR  <- Optimum
  x= 300:    17,375.00 EUR
  x= 500:    21,375.00 EUR

Das Optimum liegt exakt auf einem Szenariowert (250 = 'Volatil'),
nicht beim Mittelwert 225 - typisch fuer asymmetrische Kostenfunktionen.
```

Kein Rechentrick, sondern brute-force über alle ganzzahligen x von 0 bis 500 — das Optimum landet exakt dort, wo es die Handrechnung vorhersagt.

Warum? Die Kostenfunktion ist **asymmetrisch**: Unterdeckung kostet 120, Überdeckung nur 5. Bei asymmetrischen Kosten liegt das Optimum nie beim Mittelwert, sondern verschoben in die „billigere" Richtung. Formal ist das die **Jensensche Ungleichung**: Für konvexe Kostenfunktionen gilt $\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$ — der Erwartungswert der Kosten ist größer als die Kosten des Erwartungswerts.

> **Merksatz** Der Durchschnittskunde kauft nie. Das Durchschnittsszenario tritt nie ein. Wer mit Mittelwerten plant, optimiert eine Welt, die es nicht gibt.

Die vier Ansätze im Überblick

ANSÄTZE FÜR UNBEKANNTE ZUKUNFT		
MONTE CARLO	**STOCHASTISCHES RECOURSE**	**ROBUSTE OPTIMIERUNG**
• Simuliert Szenarien • Misst Verlustverteilung • Reine Evaluierung, keine Optimierung	• Entscheidet Stufe 1 • Passt in Stufe 2 an • Optimiert Erwartungswert über alle Szenarien	• Sucht den Worst-Case • Keine Verteilungsannahme nötig

Abbildung 19: Abb. 12.1: Ansätze für Optimierung unter Unsicherheit

	Monte-Carlo	**Stochastische Programmierung**	**Robuste Optimierung**	**Chance Constraints**
Braucht	Verteilung	Szenarien **mit Wahrscheinlichkeiten**	nur eine **Menge** möglicher Werte	Verteilung **oder** Szenarien
Optimiert	nichts (bewertet nur)	**Erwartungswert** über Szenarien	**Worst Case** in der Menge	Kosten bei **zugesicherter Quote**
Typische Frage	„Wie riskant ist dieser Plan?"	„Was ist im Mittel am besten?"	„Was hält auch im schlimmsten Fall?"	„Was hält in 95 % der Fälle?"
Stärke	beliebige Kennzahlen, sehr flexibel	nutzt Wahrscheinlichkeiten voll aus	keine Verteilungsannahme nötig	die Zusage steht **im Modell**
Schwäche	keine Optimierung	Wahrscheinlichkeiten müssen stimmen	oft übervorsichtig, kostet Ertrag	Zusage gilt nur für die unterstellte Verteilung
Nachzulesen in	Abschnitt 12.4	Abschnitt 12.5	Abschnitt 12.6	Abschnitt 12.7

Der Unterschied zwischen den letzten beiden ist der, an dem die meisten Diskussionen hängenbleiben. Die robuste Optimierung sagt: *„Es darf **nie** schiefgehen"* — und zahlt dafür. Die Chance Constraint sagt:

„Es darf in höchstens 5 % der Fälle schiefgehen“ — und macht damit verhandelbar, was die Sicherheit kosten darf.

12.4 Monte-Carlo-Simulation

Monte-Carlo heißt schlicht: Ziehe sehr viele Zufallsszenarien, werte für jedes den Plan aus, und betrachte die **Verteilung** der Ergebnisse statt eines einzigen Werts.

```
#!/usr/bin/env python3

# Monte_Carlo.py
"""
Kapitel Unsicherheit: Monte-Carlo-Bewertung von Kapazitaetsplaenen.

Monte Carlo OPTIMIERT nicht - es BEWERTET. Der Nutzen liegt darin, dass man
beliebige Kennzahlen ablesen kann: Erwartungswert, Quantile, Ausfallwahr-
scheinlichkeit, Worst Case. Genau diese Groessen braucht man, um zwischen
Plaenen zu entscheiden.
"""

import numpy as np

KOSTEN_VORAB = 40.0        # EUR je Einheit, im Voraus gekauft
KOSTEN_SPOT = 120.0        # EUR je Einheit, kurzfristig zugekauft
KOSTEN_LEERLAUF = 5.0      # EUR je ungenutzter Einheit

ANZAHL_ZIEHUNGEN = 100_000


def ziehe_bedarf(rng, ziehungen):
    """
    Bedarfsmodell: Mischverteilung aus Normalbetrieb und seltenen Lastspitzen.
    Realistischer als eine reine Normalverteilung - Krisen sind selten,
    aber extrem (fat tail).
    """
    normal = rng.normal(loc=150, scale=40, size=ziehungen)
    spitze = rng.normal(loc=450, scale=80, size=ziehungen)
    ist_spitze = rng.random(ziehungen) < 0.15            # 15 % Lastspitzen
    return np.maximum(np.where(ist_spitze, spitze, normal), 0.0)


def kosten_fuer(kapazitaet, bedarf):
    """Gesamtkosten je Szenario fuer eine gegebene Vorabkapazitaet."""
    unterdeckung = np.maximum(bedarf - kapazitaet, 0.0)
    ueberdeckung = np.maximum(kapazitaet - bedarf, 0.0)
    return (KOSTEN_VORAB * kapazitaet
            + KOSTEN_SPOT * unterdeckung
            + KOSTEN_LEERLAUF * ueberdeckung)
```

```
if __name__ == "__main__":
    rng = np.random.default_rng(2026)
    bedarf = ziehe_bedarf(rng, ANZAHL_ZIEHUNGEN)

    print("=" * 88)
    print(f"  MONTE-CARLO-BEWERTUNG ({ANZAHL_ZIEHUNGEN:,} Szenarien)")
    print("=" * 88)
    print(f"Bedarfsverteilung: Mittelwert {bedarf.mean():.1f} | "
          f"Median {np.median(bedarf):.1f} | "
          f"95%-Quantil {np.percentile(bedarf, 95):.1f} | "
          f"Maximum {bedarf.max():.1f}")
    print("Der Median liegt deutlich unter dem Mittelwert - die Verteilung ist")
    print("rechtsschief. Genau hier fuehrt Planung mit dem Mittelwert in die Irre.\n")

    print(f"{'Kapazitaet':>10} | {'Erw. Kosten':>12} | {'Median':>10} | "
          f"{'95%-Quantil':>12} | {'Unterdeckung':>12}")
    print("-" * 88)

    kandidaten = [150, 200, 225, 250, 300, 350, 400]
    ergebnisse = []
    for kapazitaet in kandidaten:
        kosten = kosten_fuer(kapazitaet, bedarf)
        p_unterdeckung = float(np.mean(bedarf > kapazitaet))
        ergebnisse.append((kapazitaet, kosten.mean(), p_unterdeckung))
        print(f"{kapazitaet:>10} | {kosten.mean():>12,.0f} | "
              f"{np.median(kosten):>10,.0f} | {np.percentile(kosten, 95):>12,.0f} | "
              f"{p_unterdeckung*100:>11.1f} %")

    beste = min(ergebnisse, key=lambda t: t[1])
    print("-" * 88)
    print(f"Bester Kandidat: Kapazitaet {beste[0]} mit erwarteten Kosten "
          f"{beste[1]:,.0f} EUR")

    # --- Feinsuche ueber ein Raster ---------------------------------------
    raster = np.arange(100, 500, 5)
    erwartete = np.array([kosten_fuer(k, bedarf).mean() for k in raster])
    optimum = raster[int(np.argmin(erwartete))]
    print(f"Feinsuche (Raster 100..500): Optimum bei Kapazitaet {optimum}, "
          f"Kosten {erwartete.min():,.0f} EUR")

    # --- Vergleich mit der naiven Mittelwertplanung -----------------------
    naiv = int(round(bedarf.mean()))
    kosten_naiv = kosten_fuer(naiv, bedarf).mean()
    kosten_opt = kosten_fuer(optimum, bedarf).mean()
    print("\n--- Fluch des Durchschnitts, gemessen ---")
    print(f"  Planung mit Mittelwert ({naiv}):  {kosten_naiv:,.0f} EUR")
```

```
    print(f"  Monte-Carlo-Optimum    ({optimum}):  {kosten_opt:,.0f} EUR")
    print(f"  Mehrkosten der naiven Planung:  {kosten_naiv - kosten_opt:,.0f} EUR "
          f"({(kosten_naiv/kosten_opt - 1)*100:.1f} %)")
    print("=" * 88)
```

Erwartete Ausgabe (gekürzt):

```
========================================================================================
  MONTE-CARLO-BEWERTUNG (100,000 Szenarien)
========================================================================================
Bedarfsverteilung: Mittelwert 195.7 | Median 159.2 | 95%-Quantil 485.2 | Maximum 753.3
Der Median liegt deutlich unter dem Mittelwert - die Verteilung ist
rechtsschief. Genau hier fuehrt Planung mit dem Mittelwert in die Irre.

Kapazitaet |  Erw. Kosten |     Median |  95%-Quantil | Unterdeckung
----------------------------------------------------------------------------------------
       150 |       13,169 |      7,104 |       46,220 |        57.8 %
       200 |       12,989 |      8,345 |       42,220 |        24.1 %
       225 |       13,475 |      9,432 |       40,220 |        17.8 %
       250 |       14,085 |     10,546 |       38,220 |        15.6 %
       300 |       15,391 |     12,791 |       34,220 |        14.8 %
       350 |       16,746 |     15,033 |       30,220 |        13.6 %
       400 |       18,214 |     17,266 |       26,220 |        11.1 %
----------------------------------------------------------------------------------------
Bester Kandidat: Kapazitaet 200 mit erwarteten Kosten 12,989 EUR
Feinsuche (Raster 100..500): Optimum bei Kapazitaet 180, Kosten 12,810 EUR

--- Fluch des Durchschnitts, gemessen ---
  Planung mit Mittelwert (196):  12,934 EUR
  Monte-Carlo-Optimum    (180):  12,810 EUR
  Mehrkosten der naiven Planung:  124 EUR (1.0 %)
========================================================================================
```

Eine wichtige Einschränkung — und was man daraus lernt

Hier kostet die naive Mittelwertplanung nur **1 %**. In der Handrechnung zum Fluch des Durchschnitts waren es 2,7 %, in Lehrbüchern liest man von 20 % und mehr. **Wie groß der Effekt ist, hängt von der Verteilung ab** — und das ist selbst die Lektion.

Bei dieser Mischverteilung liegt zwischen dem Normalbetrieb (um 150) und den Lastspitzen (um 450) ein Bereich, in dem kaum Wahrscheinlichkeitsmasse liegt. Zusätzliche Kapazität zwischen 200 und 400 nützt daher wenig: Sie ist im Normalfall Leerlauf und reicht im Krisenfall trotzdem nicht. Deshalb ist die Kostenkurve dort flach, und die Wahl zwischen 180 und 196 macht kaum einen Unterschied.

Was Sie mitnehmen sollten: Die Frage „Wie teuer ist Mittelwertplanung?“ hat keine allgemeine Antwort — **rechnen Sie es aus.** Genau dafür ist Monte-Carlo da. Und beachten Sie die Spalte „Unterdeckung“: Bei Kapazität 200 wird in **24 % der Fälle** nachgekauft. Ob das

akzeptabel ist, entscheidet kein Optimierer, sondern ein Servicelevel-Ziel (Aufgabe *Monte-Carlo erweitern*, Abschnitt 12.8).

 Wann Monte-Carlo das richtige Werkzeug ist Immer dann, wenn Sie **eine Kennzahl brauchen, die keine Formel hat**: die Wahrscheinlichkeit, ein Servicelevel zu verfehlen; der erwartete Verlust in den schlechtesten 5 % der Fälle; die Verteilung der Projektdauer. Der Preis ist, dass Monte-Carlo nur **bewertet** — die Optimierung müssen Sie durch Rastersuche oder ein eigenes Modell ergänzen.

12.5 Zweistufige stochastische Programmierung

Das **zweistufige Modell mit Recourse** bildet ab, dass Entscheidungen zeitlich gestaffelt sind:

- **Stufe 1 (*here and now*, $\mathbf{x}$):** muss **jetzt** getroffen werden, bevor der Zufall sich zeigt — Serverkapazität kaufen, Produktion planen, Portfolio aufsetzen.
- **Stufe 2 (*wait and see*, $\mathbf{y}_s$):** Korrekturmaßnahmen, **nachdem** Szenario s eingetreten ist — Spot-Zukauf, Überstunden, Notverkauf.

$$\min_{\mathbf{x}} \quad \mathbf{c}^\top \mathbf{x} \; + \sum_{s=1}^{S} p_s \, \mathbf{q}_s^\top \mathbf{y}_s$$

$$\begin{aligned} \mathbf{A}\mathbf{x} &\le \mathbf{b} && \text{(Stufe-1-Restriktionen)} \\ \mathbf{T}_s\mathbf{x} + \mathbf{W}_s\mathbf{y}_s &\le \mathbf{h}_s \quad \forall s && \text{(Kopplung je Szenario)} \\ \mathbf{x} \ge \mathbf{0},\ \mathbf{y}_s &\ge \mathbf{0} \quad \forall s \end{aligned}$$

 Formel-Lesehilfe * $\mathbf{c}^\top\mathbf{x}$ — die Kosten der Vorabentscheidung. Sie fallen **immer** an. * p_s — Wahrscheinlichkeit von Szenario s; $\sum_s p_s = 1$. * $\mathbf{q}_s^\top\mathbf{y}_s$ — die Korrekturkosten **in** Szenario s. * $\mathbf{T}_s\mathbf{x} + \mathbf{W}_s\mathbf{y}_s \le \mathbf{h}_s$ — die **Kopplungsbedingung**: Sie verbindet die Vorabentscheidung mit dem, was danach möglich ist. Genau hier steckt die Modellierungsarbeit.

Ohne Formel gesagt: „Entscheide jetzt so, dass die Summe aus heutigen Kosten und **durchschnittlich zu erwartenden** Nachbesserungskosten minimal wird."

Der entscheidende Punkt: Es gibt **ein** $\mathbf{x}$, aber **je Szenario ein eigenes** $\mathbf{y}_s$. Das ist keine Rechentrickserei, sondern bildet die Realität ab: Man legt sich einmal fest und reagiert dann unterschiedlich.

```
#!/usr/bin/env python3

# Stochastische_Optimierung.py
"""
Kapitel Unsicherheit: Two-Stage Stochastic Programming mit CVXPY.

Eigenschaften:
  * Die Analyse am Ende wird BERECHNET statt fest verdrahtet
  * Vergleich gegen drei Alternativen: Mittelwert, Worst Case, perfekte Voraussicht
```

```
  * Kennzahl EVPI (Wert perfekter Information) wird ausgewiesen
"""

import cvxpy as cp
import numpy as np
import pandas as pd

SZENARIEN = ["Ruhig", "Volatil", "Crash"]
WAHRSCHEINLICHKEIT = np.array([0.50, 0.30, 0.20])
BEDARF = np.array([100.0, 250.0, 500.0])

KOSTEN_VORAB = 40.0
KOSTEN_SPOT = 120.0
KOSTEN_LEERLAUF = 5.0

S = len(SZENARIEN)


def loese_stochastisch():
    """Zweistufiges Modell: eine Vorabentscheidung, szenarioabhaengige Korrektur."""
    x = cp.Variable(nonneg=True, name="Basiskapazitaet")        # Stufe 1
    y_spot = cp.Variable(S, nonneg=True, name="Spot_Zukauf")    # Stufe 2
    y_leer = cp.Variable(S, nonneg=True, name="Leerlauf")       # Stufe 2

    # Kopplung: Basis + Zukauf - Leerlauf == Bedarf (je Szenario)
    nebenbedingungen = [x + y_spot[s] - y_leer[s] == BEDARF[s] for s in range(S)]

    erwartete_korrektur = sum(
        WAHRSCHEINLICHKEIT[s] * (KOSTEN_SPOT * y_spot[s] + KOSTEN_LEERLAUF * y_leer[s])
        for s in range(S))

    problem = cp.Problem(cp.Minimize(KOSTEN_VORAB * x + erwartete_korrektur),
                         nebenbedingungen)
    problem.solve()
    return problem, x, y_spot, y_leer


def kosten_bei(kapazitaet):
    """Erwartete Gesamtkosten fuer eine fest vorgegebene Kapazitaet."""
    unter = np.maximum(BEDARF - kapazitaet, 0.0)
    ueber = np.maximum(kapazitaet - BEDARF, 0.0)
    je_szenario = KOSTEN_VORAB * kapazitaet + KOSTEN_SPOT * unter + KOSTEN_LEERLAUF * ueber
    return float(WAHRSCHEINLICHKEIT @ je_szenario)


if __name__ == "__main__":
    problem, x, y_spot, y_leer = loese_stochastisch()
    kapazitaet = float(x.value)
```

```
    mittelwert = float(WAHRSCHEINLICHKEIT @ BEDARF)

    print("=" * 82)
    print("     STOCHASTISCHE TWO-STAGE OPTIMIERUNG (KAPAZITAETSPLANUNG)")
    print("=" * 82)
    print(f"Status: {problem.status}")
    print(f"Erwarteter Bedarf (Mittelwert):   {mittelwert:.1f} Einheiten")
    print(f"Optimale Stufe-1-Kapazitaet x*:   {kapazitaet:.1f} Einheiten")
    print(f"Minimale erwartete Gesamtkosten:  {problem.value:,.2f} EUR\n")

    tabelle = pd.DataFrame({
        "Szenario": SZENARIEN,
        "Wahrsch.": [f"{p*100:.0f} %" for p in WAHRSCHEINLICHKEIT],
        "Bedarf": BEDARF,
        "Basis genutzt": [min(kapazitaet, b) for b in BEDARF],
        "Spot-Zukauf": np.round(y_spot.value, 1),
        "Leerlauf": np.round(y_leer.value, 1),
        "Kosten (EUR)": [f"{KOSTEN_VORAB*kapazitaet + KOSTEN_SPOT*y_spot.value[s] + KOSTEN_
         ↪  LEERLAUF*y_leer.value[s]:,.0f}"
                         for s in range(S)],
    })
    print(tabelle.to_string(index=False))

    # --- Vergleich mit Alternativstrategien (berechnet, nicht behauptet) --
    print("\n" + "-" * 82)
    print("Vergleich verschiedener Planungsstrategien:")
    print(f"{'Strategie':<34} {'Kapazitaet':>11} {'Erw. Kosten':>14} {'Mehrkosten':>13}")
    print("-" * 82)

    optimal = problem.value
    strategien = [
        ("Stochastisch optimal", kapazitaet),
        ("Naiv: Mittelwert einsetzen", mittelwert),
        ("Vorsichtig: Worst Case abdecken", float(BEDARF.max())),
        ("Optimistisch: Bestfall", float(BEDARF.min())),
    ]
    for name, kap in strategien:
        kosten = kosten_bei(kap)
        print(f"{name:<34} {kap:>11.1f} {kosten:>14,.0f} "
              f"{kosten - optimal:>+13,.0f}")

    # --- EVPI: Was waere perfekte Voraussicht wert? ----------------------
    # Bei perfekter Information wuerde man je Szenario genau den Bedarf kaufen.
    kosten_perfekt = float(WAHRSCHEINLICHKEIT @ (KOSTEN_VORAB * BEDARF))
    evpi = optimal - kosten_perfekt
    print("-" * 82)
    print(f"Kosten bei perfekter Voraussicht:      {kosten_perfekt:>10,.0f} EUR")
    print(f"Wert perfekter Information (EVPI):     {evpi:>10,.0f} EUR "
```

```
          f"({evpi/optimal*100:.1f} % der Kosten)")
    print("  -> So viel duerfte eine perfekte Bedarfsprognose hoechstens kosten.")

    # --- Automatische Interpretation ------------------------------------
    print("-" * 82)
    if kapazitaet > mittelwert + 1e-6:
        print(f"Analyse: Der Solver waehlt {kapazitaet:.0f} Einheiten und damit MEHR als")
        print(f"den Mittelwert ({mittelwert:.0f}), weil Unterdeckung ({KOSTEN_SPOT:.0f} EUR)")
        print(f"deutlich teurer ist als Leerlauf ({KOSTEN_LEERLAUF:.0f} EUR).")
    elif kapazitaet < mittelwert - 1e-6:
        print(f"Analyse: Der Solver waehlt {kapazitaet:.0f} und damit WENIGER als den")
        print(f"Mittelwert ({mittelwert:.0f}) - Leerlauf ist hier teurer als Zukauf.")
    else:
        print("Analyse: Kapazitaet entspricht dem Mittelwert (symmetrische Kosten).")
    print("=" * 82)
```

Erwartete Ausgabe:

```
==================================================================================
     STOCHASTISCHE TWO-STAGE OPTIMIERUNG (KAPAZITAETSPLANUNG)
==================================================================================
Status: optimal
Erwarteter Bedarf (Mittelwert):   225.0 Einheiten
Optimale Stufe-1-Kapazitaet x*:   250.0 Einheiten
Minimale erwartete Gesamtkosten:  16,375.00 EUR

Szenario Wahrsch.  Bedarf  Basis genutzt  Spot-Zukauf  Leerlauf Kosten (EUR)
   Ruhig     50 %   100.0          100.0          0.0     150.0       10,750
 Volatil     30 %   250.0          250.0          0.0       0.0       10,000
   Crash     20 %   500.0          250.0        250.0       0.0       40,000

----------------------------------------------------------------------------------
Vergleich verschiedener Planungsstrategien:
Strategie                           Kapazitaet    Erw. Kosten    Mehrkosten
----------------------------------------------------------------------------------
Stochastisch optimal                     250.0         16,375            -0
Naiv: Mittelwert einsetzen               225.0         16,813          +437
Vorsichtig: Worst Case abdecken          500.0         21,375        +5,000
Optimistisch: Bestfall                   100.0         19,000        +2,625
----------------------------------------------------------------------------------
Kosten bei perfekter Voraussicht:           9,000 EUR
Wert perfekter Information (EVPI):          7,375 EUR (45.0 % der Kosten)
  -> So viel duerfte eine perfekte Bedarfsprognose hoechstens kosten.
----------------------------------------------------------------------------------
Analyse: Der Solver waehlt 250 Einheiten und damit MEHR als
den Mittelwert (225), weil Unterdeckung (120 EUR)
deutlich teurer ist als Leerlauf (5 EUR).
==================================================================================
```

Lesen Sie die Vergleichstabelle von unten nach oben. Die beiden extremen Haltungen sind die teuersten: Wer den Worst Case abdeckt (500 Einheiten), zahlt 5 000 € zu viel für Kapazität, die in 80 % der Fälle brachliegt. Wer optimistisch plant (100), zahlt 2 625 € Strafe für ständige Notzukäufe. Die naive Mittelwertplanung liegt dazwischen — aber eben auch nicht optimal.

Der **EVPI von 7 375 €** ist bemerkenswert hoch: 45 % der Gesamtkosten entstehen allein daraus, dass man die Zukunft nicht kennt. In so einem Fall lohnt sich Investition in bessere Prognosen tatsächlich — anders als in dem Beispiel aus der Aufgabe *EVPI interpretieren* (Abschnitt 12.8).

> **Was ist der EVPI?** Der **Expected Value of Perfect Information** beziffert, wie viel eine perfekte Prognose wert wäre: die Differenz zwischen den Kosten unter Unsicherheit und den Kosten bei vollständigem Wissen. Er ist eine **Obergrenze für jedes Prognoseprojekt**. Wenn der EVPI bei 12 000 € pro Jahr liegt, lohnt sich keine Prognosesoftware für 50 000 € — selbst wenn sie perfekt wäre. Diese Zahl bewahrt Projekte vor teuren Fehlinvestitionen.

12.6 Robuste Optimierung: gegen den Worst Case absichern

Wenn Wahrscheinlichkeiten unbekannt oder instabil sind — Marktcrashs, Lieferkettenabrisse, Pandemien —, hilft die **robuste Optimierung**. Sie fragt nicht nach dem Mittel, sondern nach dem Schlimmsten:

$$\min_{\mathbf{x} \in X} \max_{\mathbf{u} \in \mathcal{U}} f(\mathbf{x}, \mathbf{u})$$

> **Formel-Lesehilfe** Zwei ineinandergeschachtelte Optimierungen: * Das **innere** max ist der „Gegner“: Er wählt aus der Unsicherheitsmenge $\mathcal{U}$ die für Sie ungünstigsten Parameter. * Das **äußere** min sind Sie: Sie wählen **x** so, dass Sie selbst gegen diesen Gegner am besten dastehen.
>
> **Ohne Formel gesagt:** „Entscheide so, dass du auch dann noch gut dastehst, wenn alles gegen dich läuft.“ Das ist Spieltheorie gegen die Natur.

Der Trick, der es lösbar macht. Ein Min-Max-Problem klingt unlösbar — man müsste ja unendlich viele Parameterkombinationen prüfen. Für einfache Unsicherheitsmengen lässt sich das innere Maximum aber **geschlossen ausrechnen**. Beispiel Box-Unsicherheit: Die Renditen liegen in Intervallen $\mu_i \in [\hat{\mu}_i - \delta_i,\ \hat{\mu}_i + \delta_i]$. Dann gilt für $\mathbf{w} \geq 0$:

$$\min_{\mu \in \mathcal{U}} \mu^{\top} \mathbf{w} = \sum_i (\hat{\mu}_i - \delta_i) w_i = \hat{\mu}^{\top} \mathbf{w} - \delta^{\top} \mathbf{w}$$

Das Min-Max-Problem wird damit zu einem **gewöhnlichen** Optimierungsproblem mit einem Abzugsterm. Genau das macht robuste Optimierung praktikabel.

```
#!/usr/bin/env python3

# Robuste_Optimierung.py
"""
Kapitel Unsicherheit: Robuste Portfolio-Optimierung.
```

```
Vergleicht drei Haltungen zur Unsicherheit:
  (a) nominal   - vertraut den Punktschaetzungen blind
  (b) robust    - sichert gegen Box-Unsicherheit ab (Worst Case)
  (c) stochastisch - optimiert den Erwartungswert ueber Szenarien

und misst den "Preis der Robustheit": Wie viel Ertrag kostet die Absicherung
im Normalfall - und wie viel Verlust erspart sie im Ernstfall?
"""

import cvxpy as cp
import numpy as np

ASSETS = ["Aktien Welt", "Anleihen", "Rohstoffe", "Immobilien"]
MU_SCHAETZUNG = np.array([0.085, 0.030, 0.055, 0.060])   # Punktschaetzung
UNSICHERHEIT = np.array([0.040, 0.008, 0.045, 0.025])    # +/- delta je Titel
VOLA = np.array([0.17, 0.05, 0.22, 0.12])
KORR = np.array([
    [1.00, -0.15, 0.35, 0.55],
    [-0.15, 1.00, -0.05, 0.10],
    [0.35, -0.05, 1.00, 0.25],
    [0.55, 0.10, 0.25, 1.00],
])
SIGMA = np.diag(VOLA) @ KORR @ np.diag(VOLA)
LAMBDA = 4.0        # Risikoaversion
N = len(ASSETS)


def optimiere(mu_effektiv):
    """Standard-Mean-Variance mit vorgegebenem Renditevektor."""
    w = cp.Variable(N, nonneg=True)
    ziel = cp.Maximize(mu_effektiv @ w - 0.5 * LAMBDA * cp.quad_form(w, SIGMA))
    problem = cp.Problem(ziel, [cp.sum(w) == 1])
    problem.solve()
    return w.value


def kennzahlen(w, mu):
    ertrag = float(mu @ w)
    risiko = float(np.sqrt(w @ SIGMA @ w))
    return ertrag, risiko


if __name__ == "__main__":
    print("=" * 88)
    print("  ROBUSTE vs. NOMINALE PORTFOLIO-OPTIMIERUNG")
    print("=" * 88)
    print(f"{'Asset':<14} {'Erw. Rendite':>14} {'Unsicherheit':>14} "
```

```
          f"{'Worst Case':>12} {'Volatilitaet':>13}")
    print("-" * 88)
    for i, name in enumerate(ASSETS):
        print(f"{name:<14} {MU_SCHAETZUNG[i]*100:>13.1f} % "
              f"{'+/- ' + format(UNSICHERHEIT[i]*100, '.1f') + ' %':>14} "
              f"{(MU_SCHAETZUNG[i]-UNSICHERHEIT[i])*100:>11.1f} % "
              f"{VOLA[i]*100:>12.1f} %")

    # (a) nominal: vertraut den Schaetzungen
    w_nominal = optimiere(MU_SCHAETZUNG)
    # (b) robust: rechnet mit dem Worst Case der Box-Unsicherheitsmenge
    w_robust = optimiere(MU_SCHAETZUNG - UNSICHERHEIT)

    print("\n" + "-" * 88)
    print(f"{'':<14} {'nominal':>22} {'robust':>22}")
    print("-" * 88)
    for i, name in enumerate(ASSETS):
        print(f"{name:<14} {w_nominal[i]*100:>21.1f} % {w_robust[i]*100:>21.1f} %")

    # --- Bewertung in beiden Welten -------------------------------------
    mu_worst = MU_SCHAETZUNG - UNSICHERHEIT
    e_nom_gut, r_nom = kennzahlen(w_nominal, MU_SCHAETZUNG)
    e_rob_gut, r_rob = kennzahlen(w_robust, MU_SCHAETZUNG)
    e_nom_schlecht, _ = kennzahlen(w_nominal, mu_worst)
    e_rob_schlecht, _ = kennzahlen(w_robust, mu_worst)

    print("\n" + "-" * 88)
    print(f"{'Bewertung':<34} {'nominales Portfolio':>22} {'robustes Portfolio':>22}")
    print("-" * 88)
    print(f"{'Ertrag, wenn Schaetzung stimmt':<34} {e_nom_gut*100:>21.2f} % "
          f"{e_rob_gut*100:>21.2f} %")
    print(f"{'Ertrag im Worst Case':<34} {e_nom_schlecht*100:>21.2f} % "
          f"{e_rob_schlecht*100:>21.2f} %")
    print(f"{'Volatilitaet':<34} {r_nom*100:>21.2f} % {r_rob*100:>21.2f} %")

    print("\n" + "-" * 88)
    print(f"Preis der Robustheit (Ertragsverzicht im Normalfall): "
          f"{(e_nom_gut - e_rob_gut)*100:+.2f} Prozentpunkte")
    print(f"Nutzen der Robustheit (Vorteil im Worst Case):        "
          f"{(e_rob_schlecht - e_nom_schlecht)*100:+.2f} Prozentpunkte")
    verhaeltnis = ((e_rob_schlecht - e_nom_schlecht)
                   / max(e_nom_gut - e_rob_gut, 1e-9))
    print(f"Verhaeltnis Nutzen/Preis: {verhaeltnis:.2f}")
    print("  -> Werte > 1 bedeuten: Die Absicherung bringt im Ernstfall mehr,")
    print("     als sie im Normalfall kostet.")
    print("=" * 88)
```

 Der Preis der Robustheit Robuste Modelle sind **konservativ**. Wer gegen den absoluten Worst Case absichert, zahlt im Normalfall drauf — oft erheblich. Drei Gegenmittel: 1. **Unsicherheitsmenge realistisch wählen.** Nicht „alles kann passieren“, sondern „Abweichungen bis zu einer Standardabweichung“. 2. **Budgeted Uncertainty (Bertsimas/Sim)**: Man nimmt an, dass höchstens Γ von n Parametern gleichzeitig ihren Worst Case annehmen. Γ steuert die Vorsicht stufenlos. 3. **Nutzen und Preis immer beziffern** — genau wie im Programm oben. Ohne diese zwei Zahlen ist die Diskussion „robust oder nicht“ Glaubenssache.

12.7 Wahrscheinlichkeitsbeschränkungen: „mit 95 % Sicherheit“

Die drei bisherigen Ansätze beantworten Fragen, die im Sitzungssaal selten so gestellt werden. „Was ist im Mittel am besten?“ hört ein Vorstand als Ausflucht; „was hält auch im schlimmsten Fall?“ als Einladung, zu viel Geld auszugeben. Gefragt wird fast immer nach einer **Wahrscheinlichkeit**: *Mit welcher Sicherheit hält der Plan?*

Die **Wahrscheinlichkeitsbeschränkung** schreibt genau das ins Modell:

$$\mathbb{P}\big(\mathbf{a}^\top \mathbf{x} \geq b\big) \geq 1 - \alpha$$

 Formel-Lesehilfe * **a** — der **unsichere** Vektor: Verfügbarkeiten, Erträge, Fahrzeiten. Er ist keine Zahl, sondern eine Zufallsgröße. * **x** — Ihre Entscheidung. Sie fällt, **bevor a** sich zeigt. * α — die Ausfallwahrscheinlichkeit, die Sie akzeptieren. $\alpha = 0{,}05$ heißt „in höchstens 5 % der Fälle darf es schiefgehen“.

Ohne Formel gesagt: „Plane so, dass die Zusage in 95 von 100 Fällen hält — und sag mir, was das kostet.“

Das Problem daran. So aufgeschrieben ist die Bedingung nicht lösbar: Ein Solver kann nichts mit einem Wahrscheinlichkeitsoperator anfangen. Es braucht einen Weg, sie in etwas zu übersetzen, das ein Optimierer versteht. Es gibt zwei, und sie unterscheiden sich in dem, was sie über die Verteilung voraussetzen.

Weg 1: Normalverteilung — die Bedingung wird ein Kegel

Ist $\mathbf{a} \sim \mathcal{N}(\hat{\mathbf{a}}, \Sigma)$, dann ist $\mathbf{a}^\top \mathbf{x}$ für festes $\mathbf{x}$ selbst normalverteilt, mit Erwartungswert $\hat{\mathbf{a}}^\top \mathbf{x}$ und Standardabweichung $\sqrt{\mathbf{x}^\top \Sigma \mathbf{x}}$. Damit lässt sich die Wahrscheinlichkeit durch ein Quantil ersetzen:

$$\hat{\mathbf{a}}^\top \mathbf{x} \;-\; z_{1-\alpha}\, \|\mathbf{L}^\top \mathbf{x}\|_2 \;\geq\; b \qquad \text{mit} \qquad \Sigma = \mathbf{L}\mathbf{L}^\top, \quad z_{1-\alpha} = \Phi^{-1}(1-\alpha)$$

Das ist eine **Second-Order-Cone-Bedingung** — dieselbe Bauform, die Kapitel 11 eingeführt hat, und CVXPY löst sie ohne Umstände.

 Warum eine Norm und keine Summe Der Sicherheitszuschlag $\|\mathbf{L}^\top \mathbf{x}\|_2$ ist **keine** Summe von Einzelzuschlägen je Anlage. Genau darin steckt der Nutzen der Mischung: Zwei gegenläufige Quellen schwanken **gemeinsam** weniger als jede für sich, und die Norm rechnet das automatisch mit. Ein fester Aufschlag je Anlage könnte das nicht — er wäre linear und würde Diversifikation nicht belohnen.

Weg 2: Szenarien — Big-M ohne Verteilungsannahme

Liegen statt einer Verteilung nur S beobachtete Szenarien vor, bekommt jedes eine Binärvariable z_s, die sagt, ob es verletzt werden **darf**:

$$\begin{aligned} \mathbf{a}_s^\top \mathbf{x} &\geq b - M z_s \quad \forall s \\ \sum_s z_s &\leq \alpha S \\ z_s &\in \{0,1\} \end{aligned}$$

Aus dem Kegelproblem wird ein MILP mit S Binärvariablen. Für M gilt, was Kapitel 6 über die Big-M-Falle sagt: **so klein wie möglich**. Hier ist $M = b$ die kleinste gültige Schranke, denn $\mathbf{a}_s^\top \mathbf{x} \geq 0$ — größer kann eine Verletzung gar nicht ausfallen.

Der Fall: ein Kraftwerkspark mit 500 MW Zusage

```
#!/usr/bin/env python3

# Chance_Constraints.py
"""
Kapitel Unsicherheit: Wahrscheinlichkeitsbeschraenkungen.

Das Management fragt selten "was ist im Mittel am besten?", sondern "mit
welcher Sicherheit haelt der Plan?". Genau das formuliert eine Chance
Constraint:  P(Versorgung >= Bedarf) >= 1 - alpha.

Gezeigt werden beide Wege dorthin, an derselben Instanz:
  (a) analytisch      - unter Normalverteilungsannahme wird daraus eine
                        Second-Order-Cone-Bedingung, loesbar mit CVXPY
  (b) szenariobasiert - Big-M mit Binaervariablen, fuer beliebige
                        empirische Verteilungen

Und die beiden Messungen, auf die es ankommt: Was kostet ein Prozentpunkt
Versorgungssicherheit - und haelt die Zusage auch dann, wenn die Verteilung
nicht normal ist?

Solver: CLARABEL (Kegel) und SciPy/HiGHS (gemischt-ganzzahlig). Weder ortools
noch ein direkter highspy-Import, damit alles in einem Prozess laeuft.
"""

import cvxpy as cp
import numpy as np
from scipy.stats import norm

# --- Der Kraftwerkspark ------------------------------------------------
# Die Verfuegbarkeit ist der Anteil, den eine installierte MW im Mittel
# wirklich liefert: bei Wind und Sonne klein und stark schwankend, bei Gas und
# Biomasse gross und stabil. Die Ausbaugrenze ist der Standort - Flaeche,
```

```
# Genehmigung, Brennstoffversorgung.
TECHNIK = ["Gaskraftwerk", "Windpark", "Solarpark", "Biomasse"]
KOSTEN = np.array([65_000.0, 24_000.0, 12_000.0, 60_000.0])   # EUR je MW und Jahr
VERFUEGBAR = np.array([0.92, 0.35, 0.18, 0.85])
STREUUNG = np.array([0.05, 0.16, 0.10, 0.04])
GRENZE = np.array([400.0, 250.0, 600.0, 350.0])               # MW

# Wind und Sonne sind leicht gegenlaeufig: Ein Tiefdruckgebiet bringt Wind und
# Wolken zugleich. Genau diese Korrelation macht die Bedingung zu einem Kegel
# und nicht zu einer Summe unabhaengiger Einzelzuschlaege.
KORRELATION = np.array([
    [1.00,  0.00,  0.00, 0.05],
    [0.00,  1.00, -0.25, 0.00],
    [0.00, -0.25,  1.00, 0.00],
    [0.05,  0.00,  0.00, 1.00],
])
SIGMA = np.diag(STREUUNG) @ KORRELATION @ np.diag(STREUUNG)
WURZEL = np.linalg.cholesky(SIGMA)          # L mit L @ L.T == SIGMA

BEDARF = 500.0          # MW, die gesichert bereitstehen muessen
N = len(TECHNIK)

# Die Kaeltewelle mit Dunkelflaute: selten, aber sie trifft alles zugleich.
# Wind und Sonne brechen fast vollstaendig weg - und, das ist der Punkt, das
# Gaskraftwerk liefert ebenfalls weniger, weil bei Frost der Netzdruck faellt.
# Eine Kovarianzmatrix mit Korrelationen um null kann das nicht ausdruecken.
P_KAELTEWELLE = 0.08
EINBRUCH = np.array([0.72, 0.10, 0.15, 0.88])


def mittelwertplan():
    """Plant mit den Erwartungswerten - ignoriert die Streuung vollstaendig."""
    x = cp.Variable(N, nonneg=True)
    problem = cp.Problem(cp.Minimize(KOSTEN @ x),
                         [VERFUEGBAR @ x >= BEDARF, x <= GRENZE])
    problem.solve(solver=cp.CLARABEL)
    return problem.value, x.value


def chance_constraint_analytisch(alpha):
    """
    P(a^T x >= BEDARF) >= 1 - alpha  unter a ~ N(VERFUEGBAR, SIGMA).

    Aequivalent zu:  VERFUEGBAR^T x - z * ||L^T x||_2 >= BEDARF
    mit z = Phi^-1(1 - alpha). Das ist eine Second-Order-Cone-Bedingung: Der
    Sicherheitszuschlag ist eine Norm ueber x, kein fester Aufschlag je Anlage.
    Deshalb belohnt sie Mischung - zwei gegenlaeufige Quellen schwanken
    gemeinsam weniger als jede fuer sich.
```

```
    """
    x = cp.Variable(N, nonneg=True)
    z = norm.ppf(1 - alpha)
    bedingung = VERFUEGBAR @ x - z * cp.norm(WURZEL.T @ x, 2) >= BEDARF
    problem = cp.Problem(cp.Minimize(KOSTEN @ x), [bedingung, x <= GRENZE])
    problem.solve(solver=cp.CLARABEL)
    return problem.value, x.value


def chance_constraint_szenarien(a, alpha):
    """
    Dieselbe Zusage ohne Verteilungsannahme: Fuer jedes Szenario s sagt eine
    Binaervariable z_s, ob es verletzt werden darf.

        a_s^T x >= BEDARF - M * z_s        fuer alle s
        sum_s z_s <= alpha * S

    M = BEDARF ist die kleinstmoegliche gueltige Schranke, denn a_s^T x >= 0:
    Groesser kann eine Verletzung gar nicht ausfallen. Ein unnoetig grosses M
    wuerde die LP-Relaxierung aufweichen und die Suche verlangsamen - die
    Big-M-Falle aus dem Kapitel Gemischt-ganzzahlige Optimierung.
    """
    S = len(a)
    x = cp.Variable(N, nonneg=True)
    z = cp.Variable(S, boolean=True)
    problem = cp.Problem(
        cp.Minimize(KOSTEN @ x),
        [a @ x >= BEDARF - BEDARF * z, cp.sum(z) <= alpha * S, x <= GRENZE])
    problem.solve(solver=cp.SCIPY)
    return problem.value, x.value


def ziehe_wetter(anzahl, seed):
    """Mischverteilung: normales Wetter, mit P_KAELTEWELLE ein Einbruch."""
    rng = np.random.default_rng(seed)
    a = rng.multivariate_normal(VERFUEGBAR, SIGMA, size=anzahl)
    getroffen = rng.random(anzahl) < P_KAELTEWELLE
    a[getroffen] *= EINBRUCH
    return np.clip(a, 0.0, 1.0)


def sicherheit(x, a):
    """Anteil der Szenarien, in denen der Plan den Bedarf deckt."""
    return float(np.mean(a @ x >= BEDARF))


KOPF = (f"{'Plan':<20} {'Kosten (EUR)':>13}  {'Gas':>4} {'Wind':>4} "
        f"{'Sol':>4} {'Bio':>4}  {'Normalwelt':>11} {'echte Welt':>11}")
```

```
def zeile(name, kosten, x, in_normal, in_echt):
    mix = " ".join(f"{w:4.0f}" for w in x)
    return (f"{name:<20} {kosten:>13,.0f}  {mix}  "
            f"{in_normal * 100:>9.2f} % {in_echt * 100:>9.2f} %")


if __name__ == "__main__":
    print("=" * 78)
    print("  WAHRSCHEINLICHKEITSBESCHRAENKUNGEN IM KRAFTWERKSPARK")
    print("=" * 78)
    print(f"Gesichert bereitzustellen: {BEDARF:.0f} MW\n")
    print(f"{'Technologie':<14} {'EUR/MW':>8} {'verfuegbar':>11} {'Streuung':>9} "
          f"{'Grenze':>8} {'EUR je erw. MW':>15}")
    print("-" * 78)
    for i, name in enumerate(TECHNIK):
        print(f"{name:<14} {KOSTEN[i]:>8,.0f} {VERFUEGBAR[i]:>10.2f}  "
              f"{STREUUNG[i]:>8.2f} {GRENZE[i]:>7.0f} "
              f"{KOSTEN[i] / VERFUEGBAR[i]:>15,.0f}")
    print(f"\nVollausbau liefert im Mittel {VERFUEGBAR @ GRENZE:.0f} MW, "
          f"in der Kaeltewelle {(VERFUEGBAR * EINBRUCH) @ GRENZE:.0f} MW.")

    # Zwei grosse, unabhaengige Testmengen. An ihnen wird JEDER Plan gemessen -
    # einmal in der unterstellten Normalwelt, einmal in der echten Verteilung.
    echt = ziehe_wetter(200_000, seed=771)
    normalwelt = np.random.default_rng(4711).multivariate_normal(
        VERFUEGBAR, SIGMA, size=200_000)

    print("\n" + "=" * 78)
    print("  (1) Mittelwertplan und Chance Constraints im Vergleich")
    print("=" * 78)
    print(KOPF)
    print("-" * 78)

    k_mittel, x_mittel = mittelwertplan()
    stufen = [("Mittelwert", k_mittel, sicherheit(x_mittel, normalwelt))]
    print(zeile("Mittelwert", k_mittel, x_mittel,
                sicherheit(x_mittel, normalwelt), sicherheit(x_mittel, echt)))

    plaene = {}
    for alpha in (0.20, 0.10, 0.05, 0.01):
        kosten, x = chance_constraint_analytisch(alpha)
        plaene[alpha] = (kosten, x)
        quote_normal = sicherheit(x, normalwelt)
        stufen.append((f"Zusage {(1 - alpha) * 100:.0f} %", kosten, quote_normal))
        print(zeile(f"Zusage {(1 - alpha) * 100:.0f} %", kosten, x,
                    quote_normal, sicherheit(x, echt)))
```

```
print("\n  In der Normalwelt trifft jede Zusage ihren Wert - das Verfahren")
print("  rechnet richtig. Die Spalte 'echte Welt' kommt in Teil (3).")

print("\n" + "=" * 78)
print("  (2) Was kostet ein Prozentpunkt Versorgungssicherheit?")
print("=" * 78)
print(f"{'von -> nach':<26} {'Prozentpunkte':>13} {'Mehrkosten':>13} "
      f"{'EUR je Punkt':>13}")
print("-" * 78)
for (n1, k1, q1), (n2, k2, q2) in zip(stufen, stufen[1:]):
    d_punkte = (q2 - q1) * 100
    d_kosten = k2 - k1
    print(f"{n1 + ' -> ' + n2:<26} {d_punkte:>13.2f} {d_kosten:>13,.0f} "
          f"{d_kosten / d_punkte:>13,.0f}")
erst = (stufen[1][1] - stufen[0][1]) / ((stufen[1][2] - stufen[0][2]) * 100)
letzt = (stufen[-1][1] - stufen[-2][1]) / ((stufen[-1][2] - stufen[-2][2]) * 100)
print(f"\n  Der letzte Prozentpunkt kostet das {letzt / erst:.1f}-fache des ersten.")
print("  Sicherheit ist konvex bepreist - genau deshalb muss jemand")
print("  entscheiden, wie viel davon das Unternehmen kaufen will.")

print("\n" + "=" * 78)
print("  (3) Und wenn die Verteilung nicht normal ist?")
print("=" * 78)
k_soc, x_soc = plaene[0.05]
print(f"Die echte Welt kennt die Kaeltewelle mit Dunkelflaute: in "
      f"{P_KAELTEWELLE * 100:.0f} % der Faelle liefern")
print(f"Wind {EINBRUCH[1] * 100:.0f} %, Sonne {EINBRUCH[2] * 100:.0f} % "
      f"und - entscheidend - auch das Gaskraftwerk nur "
      f"{EINBRUCH[0] * 100:.0f} %")
print("des Ueblichen. Alles bricht gleichzeitig ein.\n")

a_bau = ziehe_wetter(400, seed=20260908)
k_sz, x_sz = chance_constraint_szenarien(a_bau, 0.05)

print(KOPF)
print("-" * 78)
print(zeile("SOC, Zusage 95 %", k_soc, x_soc,
            sicherheit(x_soc, normalwelt), sicherheit(x_soc, echt)))
print(zeile("Szenarien, 95 %", k_sz, x_sz,
            sicherheit(x_sz, normalwelt), sicherheit(x_sz, echt)))

print(f"\n  Der SOC-Plan verspricht 95 % und haelt "
      f"{sicherheit(x_soc, echt) * 100:.1f} %. Nicht das Verfahren ist")
print(f"  falsch, sondern die Annahme: In der Normalwelt liefert derselbe")
print(f"  Plan {sicherheit(x_soc, normalwelt) * 100:.1f} %.")
print(f"\n  Der Szenarioplan ({len(a_bau)} Szenarien, {len(a_bau)} "
      f"Binaervariablen) kommt auf")
```

```
    print(f"  {sicherheit(x_sz, echt) * 100:.1f} % - und kostet dafuer "
          f"{(k_sz / k_soc - 1) * 100:+.1f} %. Er kauft keine Windkraft mehr:")
    print("  Was im Ernstfall ausfaellt, hilft der Zusage nicht.")
    print("=" * 78)
```

Ausgabe:

```
==============================================================================
  WAHRSCHEINLICHKEITSBESCHRAENKUNGEN IM KRAFTWERKSPARK
==============================================================================
Gesichert bereitzustellen: 500 MW

Technologie      EUR/MW  verfuegbar  Streuung   Grenze  EUR je erw. MW
------------------------------------------------------------------------------
Gaskraftwerk     65,000       0.92      0.05     400          70,652
Windpark         24,000       0.35      0.16     250          68,571
Solarpark        12,000       0.18      0.10     600          66,667
Biomasse         60,000       0.85      0.04     350          70,588

Vollausbau liefert im Mittel 861 MW, in der Kaeltewelle 552 MW.

==============================================================================
  (1) Mittelwertplan und Chance Constraints im Vergleich
==============================================================================
Plan                  Kosten (EUR)   Gas Wind  Sol  Bio   Normalwelt  echte Welt
------------------------------------------------------------------------------
Mittelwert              34,694,565     8  250  600  350      50.08 %     46.18 %
Zusage 80 %             36,382,277   226   25   44  343      80.06 %     73.61 %
Zusage 90 %             36,994,777   233   20   35  349      89.99 %     82.82 %
Zusage 95 %             37,514,307   241   19   31  350      94.97 %     87.44 %
Zusage 99 %             38,529,698   258   18   28  350      98.97 %     91.05 %

  In der Normalwelt trifft jede Zusage ihren Wert - das Verfahren
  rechnet richtig. Die Spalte 'echte Welt' kommt in Teil (3).

==============================================================================
  (2) Was kostet ein Prozentpunkt Versorgungssicherheit?
==============================================================================
von -> nach                Prozentpunkte    Mehrkosten  EUR je Punkt
------------------------------------------------------------------------------
Mittelwert -> Zusage 80 %          29.98     1,687,712        56,289
Zusage 80 % -> Zusage 90 %          9.93       612,500        61,685
Zusage 90 % -> Zusage 95 %          4.98       519,530       104,365
Zusage 95 % -> Zusage 99 %          4.00     1,015,390       253,848

  Der letzte Prozentpunkt kostet das 4.5-fache des ersten.
  Sicherheit ist konvex bepreist - genau deshalb muss jemand
  entscheiden, wie viel davon das Unternehmen kaufen will.
```

```
================================================================================
  (3) Und wenn die Verteilung nicht normal ist?
================================================================================
Die echte Welt kennt die Kaeltewelle mit Dunkelflaute: in 8 % der Faelle liefern
Wind 10 %, Sonne 15 % und - entscheidend - auch das Gaskraftwerk nur 72 %
des Ueblichen. Alles bricht gleichzeitig ein.

Plan                  Kosten (EUR)   Gas Wind  Sol  Bio   Normalwelt  echte Welt
--------------------------------------------------------------------------------
SOC, Zusage 95 %        37,514,307   241   19   31  350      94.97 %     87.44 %
Szenarien, 95 %         43,760,709   350    0    0  350     100.00 %     94.84 %

  Der SOC-Plan verspricht 95 % und haelt 87.4 %. Nicht das Verfahren ist
  falsch, sondern die Annahme: In der Normalwelt liefert derselbe
  Plan 95.0 %.

  Der Szenarioplan (400 Szenarien, 400 Binaervariablen) kommt auf
  94.8 % - und kostet dafuer +16.7 %. Er kauft keine Windkraft mehr:
  Was im Ernstfall ausfaellt, hilft der Zusage nicht.
================================================================================
```

Drei Befunde, die man dem Plan nicht ansieht.

Erstens: Der Mittelwertplan fällt in der Hälfte aller Fälle aus. Das ist kein Fehler, sondern die Definition des Erwartungswerts — und dieselbe Falle wie in Abschnitt 12.3, nur teurer. Er kauft die billigsten Quellen je *erwarteter* Megawattstunde und maximiert Wind und Sonne bis an die Ausbaugrenze. Sobald die Zusage im Modell steht, dreht sich das Bild: Gas und Biomasse tragen den Plan, Wind und Sonne schrumpfen auf einen Rest.

Zweitens: Sicherheit ist konvex bepreist. Die ersten dreißig Prozentpunkte kosten rund 56 000 € pro Punkt, die letzten vier rund 254 000 € — das **4,5-fache**. Das ist die Zahl, die in die Vorlage gehört: Nicht „was kostet Versorgungssicherheit“, sondern „was kostet der *nächste* Prozentpunkt“. Wer 99,9 % fordert, ohne diese Kurve gesehen zu haben, fordert ins Blaue.

Drittens — und das ist der unangenehme Befund: Die Zusage gilt nur für die unterstellte Verteilung. Der Plan verspricht 95 % und hält **87,4 %**. Nicht weil das Verfahren falsch rechnet: In einer reinen Normalwelt liefert derselbe Plan exakt seine 95,0 %. Sondern weil die Normalverteilung eine Kältewelle mit Dunkelflaute nicht kennt. In der bricht **alles zugleich** ein — auch das Gaskraftwerk, weil bei Frost der Netzdruck fällt. Eine Kovarianzmatrix mit Korrelationen um null kann so etwas nicht ausdrücken.

Der szenariobasierte Weg kennt diese Fälle, weil sie in den Daten stehen. Er kommt auf 94,8 %, kostet dafür 16,7 % mehr — und kauft **keine Windkraft mehr**. Die Logik dahinter ist hart und richtig: Was im Ernstfall ausfällt, hilft der Zusage nicht.

> **Die Szenariomethode überanpasst — und mehr Szenarien helfen nicht verlässlich**
> Bei S Szenarien und $\alpha = 5\,\%$ darf das Modell genau $0{,}05 \cdot S$ davon ignorieren — und es sucht sich die **teuersten** aus. In der Stichprobe trifft es seine Quote dadurch immer; außerhalb streut sie.

Gemessen an zwölf Läufen dieses Modells (je vier Ziehungen bei $S = 200, 400, 800$, geprüft an 200 000 unabhängigen Szenarien): Die tatsächliche Quote lag zwischen **93,1 % und 96,7 %**. Die Spanne wurde von $S = 200$ auf 400 enger (3,5 → 1,2 Prozentpunkte), bei $S = 800$ aber wieder breiter (3,1) — die Zahl der Szenarien allein schließt die Lücke also nicht.

Konsequenz für die Praxis: Eine mit Szenarien erkaufte Zusage immer an einer **zurückgehaltenen** Stichprobe nachmessen, so wie es das Programm oben tut. Wer nur die Trefferquote in den Baudaten berichtet, berichtet eine Tautologie — dieselbe Falle wie beim Backtest in Kapitel 21.

 Merksatz Die robuste Optimierung fragt „was, wenn alles schiefgeht?“, die Chance Constraint „wie oft darf es schiefgehen?“. Nur die zweite Frage hat eine Antwort, über die man verhandeln kann — und nur sie zwingt dazu, den Preis eines Prozentpunkts Sicherheit zu beziffern.

12.8 Übungsaufgaben

Lösungen: Abschnitt A.12.

Aufgabe 12.1 — Fluch des Durchschnitts erkennen. Ein Bauunternehmer plant mit der *durchschnittlichen* Bauzeit von 8 Monaten. Warum ist die *erwartete* Fertigstellung trotzdem später als 8 Monate, wenn Verzögerungen wahrscheinlicher sind als Beschleunigungen? Nennen Sie zwei weitere Alltagsbeispiele.

Aufgabe 12.2 — Ansatz wählen. Welcher der vier Ansätze passt? Begründen Sie: (a) Wie viele Notstromaggregate für ein Krankenhaus? (b) Wie viel Weizen einkaufen bei bekannter Preisverteilung? (c) Wie hoch ein Deich gebaut werden muss? (d) Wie viele Saisonkräfte einstellen bei bekannten Nachfrageszenarien? (e) Wie viel Speicherkapazität, damit ein Gasnetz an 99 % aller Wintertage hält?

Aufgabe 12.3 — Zweistufiges Modell rechnen. Wiederholen Sie die Handrechnung *Warum der Mittelwert in die Irre führt* mit geänderten Kosten: Spot 60 € statt 120 €. (a) Wie ändert sich die optimale Kapazität? (b) Erklären Sie die Richtung der Änderung. (c) Bei welchem Spot-Preis wäre die Mittelwert-Planung optimal?

Aufgabe 12.4 — EVPI interpretieren. Ihr Modell liefert EVPI = 8 400 € pro Jahr. Ein Anbieter verlangt 15 000 € jährlich für eine Prognoselösung, die „80 % Treffsicherheit“ verspricht. Wie argumentieren Sie?

Aufgabe 12.5 — Monte-Carlo erweitern. Erweitern Sie `Monte_Carlo.py`: (a) Fügen Sie ein Servicelevel-Kriterium hinzu: „Die Unterdeckung darf höchstens in 5 % der Fälle auftreten.“ Welche Kapazität ist dafür nötig, und was kostet sie zusätzlich? (b) Berechnen Sie den CVaR der Kosten (Mittelwert der schlechtesten 5 %). (c) Zeichnen Sie ein Histogramm der Kosten für drei Kapazitäten.

Aufgabe 12.6 — Den Preis der Zusage selbst bestimmen. `Chance_Constraints.py` beziffert die Kosten für 80, 90, 95 und 99 %. (a) Ergänzen Sie die Stufen 99,5 % und 99,9 % und setzen Sie die Kosten je Prozentpunkt fort. Was passiert mit dem Anlagenmix? (b) Drosseln Sie die Ausbaugrenze der Biomasse von 350 auf 200 MW. Das **szenariobasierte** Modell meldet dann bei 95 % `infeasible`, das **analytische** dagegen `optimal`. Erklären Sie den Unterschied. Welchem der beiden würden Sie glauben?

(c) Ein Vorstand fordert „99,99 % Versorgungssicherheit“. Formulieren Sie in drei Sätzen, was Sie ihm mit dieser Kurve antworten.

Aufgabe 12.7 □□□ — Budgeted Uncertainty. Erweitern Sie `Robuste_Optimierung.py` um den Ansatz von Bertsimas/Sim: Höchstens Γ Titel nehmen gleichzeitig ihren Worst Case an. Variieren Sie $\Gamma \in \{0, 1, 2, 3, 4\}$ und stellen Sie den Verlauf von Ertrag und Absicherung dar. (Hinweis: Bei Γ ganzzahlig genügt es, die Γ größten $\delta_i w_i$ abzuziehen — das lässt sich in CVXPY mit `cp.sum_largest` formulieren.)

12.9 Finde den Denkfehler

□ Finde den Denkfehler: Warum jedes Projekt zu spät fertig wird

Ein Projektleiter plant eine Umbaumaßnahme. Fünf Gewerke arbeiten **parallel** und unabhängig voneinander; fertig ist der Umbau, wenn das letzte fertig ist. Für jedes Gewerk liegt eine Schätzung vor: mindestens 6 Tage, wahrscheinlich 10, im schlechtesten Fall 18.

Er rechnet: *„Jedes Gewerk braucht im Mittel* $(6+10+18)/3 = 11{,}3$ *Tage. Alle laufen gleichzeitig. Also ist der Umbau nach 11,3 Tagen fertig — ich plane großzügig 12 ein.“*

Nachgerechnet mit 200 000 simulierten Umbauten:

```
import numpy as np

rng = np.random.default_rng(3)
dauer = rng.triangular(6, 10, 18, size=(200_000, 5))    # 5 Gewerke, je eine
 ↪  Schaetzung
projekt = dauer.max(axis=1)                             # fertig, wenn das LETZTE
 ↪  fertig ist

print(f"Mittelwert einer Einzeldauer:  {dauer.mean():.2f} Tage")
print(f"Tatsaechliche Projektdauer:    {projekt.mean():.2f} Tage")
print(f"90-%-Quantil:                  {np.quantile(projekt, 0.9):.2f} Tage")
print(f"Anteil ueber 11,34 Tage:       {(projekt > 11.34).mean() * 100:.1f} %")
```

Ausgabe:

```
Mittelwert einer Einzeldauer:  11.34 Tage
Tatsaechliche Projektdauer:    14.38 Tage
90-%-Quantil:                  16.59 Tage
Anteil ueber 11,34 Tage:       95.5 %
```

Ihre Aufgabe: (a) Der Projektleiter hat richtig gerechnet — jedes Gewerk braucht tatsächlich im Mittel 11,34 Tage. Warum dauert das Projekt trotzdem 14,38 Tage? (b) Erklären Sie in einem Satz, warum ausgerechnet **95,5 %** aller Umbauten den Plan reißen. Was müsste gelten, damit es 50 % wären? (c) Was passiert mit der Projektdauer, wenn statt fünf Gewerken zehn parallel arbeiten — bei unveränderter Einzelschätzung? (d) Welche Zahl gehört in den Projektplan, wenn der Auftraggeber eine Terminzusage mit 90 % Sicherheit verlangt?

Auflösung: Abschnitt A.12.

 Merksatz Der Fluch des Durchschnitts hat zwei Gesichter. Beim Ersatzteil aus Abschnitt 12.1 führt der Mittelwert in die Irre, weil die **Kosten** asymmetrisch sind. Hier führt er in die Irre, weil die **Verknüpfung** asymmetrisch ist: Ein Gewerk, das früher fertig wird, hilft niemandem — ein Gewerk, das sich verspätet, verzögert alles. Beide Male gilt: *Der Mittelwert einer Funktion ist nicht die Funktion des Mittelwerts.*

12.10 Micro-Quiz

 Micro-Quiz 12: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Bei einem Ersatzteil kostet ein Stück zu wenig 1 400 €, ein Stück zu viel 120 €. Der Bedarf beträgt im Mittel 30 Stück. Wie viel bestellen Sie? (a) 30 — das ist der Erwartungswert des Bedarfs. (b) Deutlich mehr als 30, nämlich das 92,1-%-Quantil des Bedarfs. Das kritische Verhältnis $1400/(1400 + 120)$ sagt, wie weit man sich auf die günstigere Fehlerseite stellen soll. (c) Weniger als 30, weil Lagerhaltung Kapital bindet.

2. Was unterscheidet stochastische** von **robuster** Optimierung?** (a) Stochastische Optimierung ist genauer, robuste ist eine Näherung für schnelle Rechnungen. (b) Stochastische Optimierung braucht **Wahrscheinlichkeiten** und optimiert den Erwartungswert; robuste Optimierung braucht nur **Bandbreiten** und sichert den schlechtesten Fall darin ab. (c) Robuste Optimierung berücksichtigt mehr Szenarien.

3. Ein Kollege ersetzt in seinem Modell alle unsicheren Größen durch ihre Mittelwerte und rechnet deterministisch. Wann ist das unproblematisch? (a) Immer — der Erwartungswert ist die beste Einzelschätzung. (b) Nie. (c) Wenn Zielfunktion und Nebenbedingungen in den unsicheren Größen **linear** sind und keine Entscheidung erst nach Beobachtung fällt. Sobald Minimum, Maximum, Betrag oder eine Nachbesserungsentscheidung im Spiel ist, gilt es nicht mehr.

12.11 Selbsttest

Antworten: Anhang A.

1. Warum ist Planung mit Erwartungswerten bei asymmetrischen Kosten systematisch falsch?
2. Was unterscheidet Stufe-1- von Stufe-2-Variablen?
3. Was misst der EVPI, und wofür ist er praktisch nützlich?
4. Warum braucht robuste Optimierung keine Wahrscheinlichkeiten?
5. Wie macht man ein Min-Max-Problem mit Box-Unsicherheit lösbar?
6. Warum wird aus $\mathbb{P}(\mathbf{a}^\top \mathbf{x} \geq b) \geq 1 - \alpha$ unter Normalverteilung eine **Norm** und nicht ein fester Zuschlag je Variable?
7. Ein Plan mit Chance Constraint verspricht 95 % und hält gemessen 87 %. Woran liegt das — und was ändert die szenariobasierte Formulierung daran?

12.12 Zusammenfassung

- **Der Fluch des Durchschnitts** ist kein Randphänomen und hat zwei Gesichter. Bei asymmetrischen **Kosten** liegt das Optimum systematisch neben dem Mittelwert — das Newsvendor-Verhältnis $c_-/(c_- + c_+)$ sagt, wie weit. Bei asymmetrischer **Verknüpfung** (Maximum über parallele Vorgänge) ist der Erwartungswert des Ganzen größer als das Ganze der Erwartungswerte. Beide Male gilt: Der Mittelwert einer Funktion ist nicht die Funktion des Mittelwerts (Jensensche Ungleichung).
- **Eine Terminzusage ist ein Quantil, kein Mittelwert.** Wer den Erwartungswert zusagt, reißt den Termin bei fünf parallelen Vorgängen in 95 % der Fälle.
- **Monte-Carlo** bewertet, optimiert aber nicht. Sein Wert liegt in Kennzahlen, für die es keine Formel gibt.
- **Zweistufige stochastische Programme** trennen die Festlegung von der Reaktion — ein $\mathbf{x}$, aber je Szenario ein $\mathbf{y}_s$.
- **Der EVPI** begrenzt, was eine perfekte Prognose wert sein darf.
- **Robuste Optimierung** braucht keine Wahrscheinlichkeiten, nur eine Unsicherheitsmenge — und wird für einfache Mengen zu einem gewöhnlichen Problem mit Abzugsterm.
- **Chance Constraints** schreiben die Zusage selbst ins Modell. Unter Normalverteilung wird daraus eine **Kegelbedingung** (der Sicherheitszuschlag ist eine Norm, deshalb belohnt sie Mischung), szenariobasiert ein **MILP mit Big-M** ohne jede Verteilungsannahme.
- **Sicherheit ist konvex bepreist.** Im Beispiel kostet der letzte Prozentpunkt das 4,5-fache des ersten. Die richtige Managementfrage lautet nicht „was kostet Sicherheit“, sondern „was kostet der *nächste* Prozentpunkt“.
- **Eine Zusage gilt nur für die unterstellte Verteilung.** Ein Plan, der unter Normalverteilung 95 % verspricht, hielt gemessen 87 %, weil das Modell die Kältewelle nicht kannte, in der *alles* zugleich ausfällt. Szenariobasierte Zusagen wiederum überanpassen — beide müssen an zurückgehaltenen Daten nachgemessen werden.
- **Beziffern Sie immer Preis und Nutzen der Absicherung.** Ohne beide Zahlen ist die Entscheidung nicht begründbar.

Ausblick. Kapitel 13 fügt die Zeitdimension hinzu: Entscheidungen, die über viele Perioden aufeinander aufbauen — gelöst mit der Bellman-Gleichung.

Kapitel 13: Dynamische Programmierung — Die Bellman-Gleichung und Order-Execution

Kapitel auf einen Blick

Worum geht es? Um Entscheidungsketten: Was heute klug ist, hängt davon ab, was morgen noch möglich sein wird. Die Bellman-Gleichung löst solche Probleme durch Rückwärtsrechnen.

Voraussetzungen: Kapitel 12 (Erwartungswerte). Rekursion in Python.

Danach können Sie: Ein mehrperiodiges Problem in Zustände und Stufen zerlegen, die Wertfunktion durch Rückwärtsinduktion berechnen, die optimale Strategie ableiten — und prüfen, ob Ihr Zustand vollständig genug ist.

Zeitbedarf: ca. 5 Stunden.

Programme:
`Bellman_Minimalbeispiel.py`
`Mehrperiodige_Order_Execution.py`

Notebook: `Notebooks_04/dynamische-programmierung.ipynb`

13.1 In 5 Minuten gelöst

□ In 5 Minuten gelöst: Der Weg, der falsch anfängt

Ein Fahrer soll vom Depot zum Kunden. Es gibt zwei Zwischenetappen, jeweils mit zwei Möglichkeiten. Die Zahlen sind Fahrminuten:

von	nach	Minuten	von	nach	Minuten
Start	**Nord**	**4**	Süd	Ost	3
Start	Süd	6	Süd	West	7
Nord	Ost	14	Ost	Ziel	5
Nord	West	9	West	Ziel	8

```python
kanten = {
    "Start": {"Nord": 4, "Sued": 6},
    "Nord":  {"Ost": 14, "West": 9},
    "Sued":  {"Ost": 3,  "West": 7},
    "Ost":   {"Ziel": 5},
    "West":  {"Ziel": 8},
    "Ziel":  {},
}

wert, plan = {"Ziel": 0}, {}                        # Restkosten ab jedem Ort
for ort in ["Ost", "West", "Nord", "Sued", "Start"]:      # RUECKWAERTS!
    wert[ort], plan[ort] = min((k + wert[z], z) for z, k in kanten[ort].items())

weg, ort = ["Start"], "Start"
while ort != "Ziel":
    ort = plan[ort]
    weg.append(ort)
print(" -> ".join(weg), "=", wert["Start"], "min")
print("Restkosten je Ort:", wert)
```

Ausgabe:

```
Start -> Sued -> Ost -> Ziel = 14 min
Restkosten je Ort: {'Ziel': 0, 'Ost': 5, 'West': 8, 'Nord': 17, 'Sued': 8, 'Start': 14}
```

Der optimale Weg beginnt mit dem *teureren* ersten Abschnitt. Start → Süd kostet 6 Minuten, Start → Nord nur 4. Wer vorne spart, landet bei 21 Minuten statt 14 — **50 % mehr**:

Vorgehen	Route	Dauer
Gierig: „nimm immer die kürzeste nächste Strecke“	Start → Nord → West → Ziel	21 min
Rückwärtsinduktion	Start → **Süd** → Ost → Ziel	**14 min**

Der Grund steht in der zweiten Ausgabezeile. `Restkosten je Ort` ist die **Wertfunktion**: Sie sagt für jeden Ort, was der ganze Rest von dort aus noch kostet, wenn man ab dort optimal weiterfährt. Und dort steht das Entscheidende: Von Nord aus sind es noch **17** Minuten, von Süd nur **8**. Die vier gesparten Minuten am Anfang werden später mit neun zusätzlichen bezahlt.

> **Merksatz** Eine Entscheidung ist nie für sich zu bewerten, sondern nur **zusammen mit allem, was danach kommt**. Genau das leistet die Wertfunktion: Sie fasst die gesamte Zukunft eines Zustands in einer einzigen Zahl zusammen. Und weil man die Zukunft kennen muss, bevor man die Gegenwart bewerten kann, rechnet dynamische Programmierung **rückwärts**.

Warum funktioniert das? Weil man den Rest des Weges nicht neu durchdenken muss, sobald man einmal an einem Ort steht: Wie man dorthin gekommen ist, ist für die Zukunft egal. Diese Eigenschaft heißt **Bellmansches Optimalitätsprinzip** und ist der Grund, warum aus einem Problem mit exponentiell vielen Wegen eine Rechnung mit wenigen Zeilen wird. Bei fünf Orten fällt das kaum auf — bei fünfzig ist es der Unterschied zwischen Sekunden und Jahren.

13.2 Lernziele

Nach diesem Kapitel können Sie …

1. … das Bellmansche Optimalitätsprinzip formulieren und erklären, warum es gilt.
2. … Zustand, Stufe, Aktion und Wertfunktion für ein gegebenes Problem benennen.
3. … eine Rückwärtsinduktion von Hand für ein kleines Problem durchführen.
4. … das Almgren-Chriss-Ausführungsproblem lösen und die Lösung interpretieren.
5. … einschätzen, wann DP funktioniert und wann der „Fluch der Dimensionalität“ zuschlägt.
6. … die Wertfunktion als **Nachschlagetabelle** lesen: Sie liefert keine Planfolge, sondern eine Regel, die auch bei Abweichungen gilt.
7. … einen Zustandsraum auf Vollständigkeit prüfen und erkennen, wann eine Kostenart Vorgeschichte erfordert.

13.3 Das Bellmansche Optimalitätsprinzip

Richard Bellman formulierte 1957 das Grundprinzip der **Dynamischen Programmierung (DP)**:

> **Optimalitätsprinzip.** Eine optimale Strategie hat die Eigenschaft, dass — unabhängig vom Anfangszustand und der Anfangsentscheidung — die verbleibenden Entscheidungen eine optimale Strategie bezüglich des Zustands bilden, der aus der ersten Entscheidung resultiert.

Ohne Fachsprache: *Jedes Teilstück einer optimalen Route ist selbst eine optimale Route.* Wenn der beste Weg von Hamburg nach München über Kassel führt, dann ist der Teil von Kassel nach München auch der beste Weg von Kassel nach München. Sonst könnte man ihn ersetzen und wäre insgesamt besser.

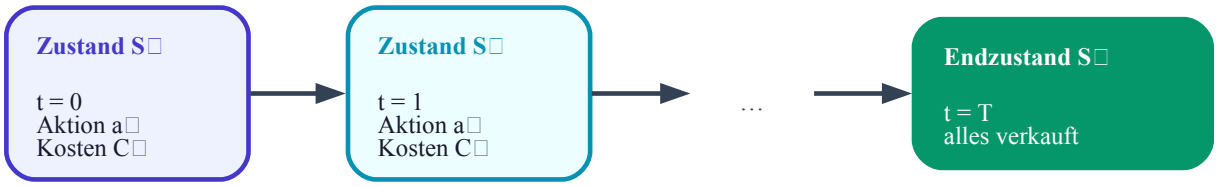

Abbildung 20: Abb. 13.1: Sequenzielle Zustands-/Aktionskette

Die vier Bausteine eines DP-Modells

Baustein	Symbol	Frage	Beispiel Orderausführung
Stufe	t	Wann wird entschieden?	Handelsperiode 0, 1, …, 5
Zustand	S_t	Was beschreibt die Lage vollständig?	Verbleibende Aktien X_t
Aktion	a_t	Was darf man tun?	Verkaufe n_t Stück
Wertfunktion	$V_t(S_t)$	Was kosten die optimalen Restentscheidungen?	Minimale Restkosten

> □ **Die Kunst liegt in der Zustandsdefinition** Der Zustand muss **alles** enthalten, was für die Zukunft relevant ist — und **nichts mehr**. Zu wenig: Das Modell ist falsch (Markov-Eigenschaft verletzt). Zu viel: Der Zustandsraum explodiert. Diese Abwägung ist die eigentliche Modellierungsleistung bei DP.

Die Bellman-Gleichung

$$V_t(S_t) = \min_{a_t \in \mathcal{A}(S_t)} \Big\{ C(S_t, a_t) \; + \; \gamma \cdot \mathbb{E}\big[V_{t+1}(S_{t+1}) \mid S_t, a_t\big] \Big\}$$

mit der Endbedingung $V_T(S_T) = g(S_T)$.

> □ **Formel-Lesehilfe** * $V_t(S_t)$ — „Was kostet mich der Rest, wenn ich zum Zeitpunkt t im Zustand S_t bin und ab jetzt optimal handle?“ * $C(S_t, a_t)$ — die **sofortigen** Kosten der Aktion a_t. * $V_{t+1}(S_{t+1})$ — die Restkosten **danach**, wieder optimal. * γ — Diskontfaktor (bei kurzen Horizonten meist $\gamma = 1$). * $\mathbb{E}[\cdot]$ — Erwartungswert, falls der Übergang zufällig ist.

Ohne Formel gesagt: „Der Wert von hier ist: die Kosten des nächsten Schritts plus der Wert von dort — und zwar für denjenigen nächsten Schritt, bei dem diese Summe am kleinsten ist."

Warum rückwärts? Weil V_{t+1} bekannt sein muss, bevor man V_t berechnen kann. Am Ende ($t = T$) ist der Wert bekannt — dort fängt man an und arbeitet sich nach vorn.

Formel-Übersetzer: die Bellman-Gleichung Baustein für Baustein

Mathematik	Alltagssprache
S_t	„Alles, was ich jetzt über meine Lage wissen muss." Restmenge, Lagerbestand, Position — **und sonst nichts**.
a_t	„Was tue ich jetzt?" Die Entscheidung dieser Stufe.
$\mathcal{A}(S_t)$	„Was ist von hier aus überhaupt erlaubt?" Die zulässigen Aktionen hängen vom Zustand ab.
$C(S_t, a_t)$	„Was kostet mich das sofort?"
$V_{t+1}(S_{t+1})$	„Was kostet mich der ganze Rest danach — wenn ich ab dort optimal handle?"
$\mathbb{E}[\cdot]$	„… gemittelt über das, was der Zufall dazwischenwirft." Entfällt bei sicheren Übergängen.
γ	„Wie viel ist mir ein Euro nächstes Jahr heute wert?" Bei kurzen Horizonten meist 1.
$\min_{a_t}\{\dots\}$	„Und von allen erlaubten Aktionen nehme ich die, bei der *Sofortkosten plus Rest* am kleinsten ist."
$V_T(S_T) = g(S_T)$	„Am Ende steht fest, was die Endlage wert ist." Hier fängt die Rechnung an.

Die ganze Gleichung in einem Satz: *Der Wert von hier ist: was der nächste Schritt kostet, plus was es von dort aus noch kostet — für denjenigen nächsten Schritt, bei dem diese Summe am kleinsten ist.*

Und die Zeile, die man sich merken sollte: Sobald Sie V_t für alle Zustände kennen, ist die optimale Strategie eine reine **Nachschlagetabelle** — für jeden Zustand steht darin, was zu tun ist. Das ist der praktische Ertrag der ganzen Rechnung: nicht ein Plan, sondern eine **Regel**, die auch dann noch gilt, wenn es anders kommt als gedacht.

Handrechnung 13.1: Rückwärtsinduktion von Hand

Problem: Sie müssen 3 Einheiten in 2 Perioden verkaufen. Sofortkosten je Periode: $C(n) = n^2$ (großer Verkauf drückt den Preis überproportional). In der letzten Periode muss alles weg.

Schritt 1 — Endstufe $t = 2$: Es sind keine Perioden mehr übrig, also $V_2(0) = 0$, alle anderen Zustände sind unzulässig.

Schritt 2 — Stufe $t = 1$ (letzte Verkaufsperiode, alles muss weg):

Zustand X_1	erzwungene Aktion	Kosten	V_1
0	0	0	0
1	1	1	1
2	2	4	4
3	3	9	9

Schritt 3 — Stufe $t = 0$ (Start mit $X_0 = 3$), alle Aktionen prüfen:

Aktion n_0	Sofortkosten n_0^2	Rest X_1	$V_1(X_1)$	Summe
0	0	3	9	9
1	1	2	4	5
2	**4**	**1**	**1**	**5**
3	9	0	0	9

Ergebnis: $V_0(3) = 5$, erreicht durch $n_0 = 1$ (dann $n_1 = 2$) **oder** $n_0 = 2$ (dann $n_1 = 1$). Zwei gleichwertige Optima.

Die Lehre: Alles auf einmal zu verkaufen kostet 9, gleichmäßiges Aufteilen nur 5. Das ist der Kern des Ausführungsproblems: **Bei überproportionalen Kosten lohnt sich Stückelung.** Und beachten Sie, dass wir nur $4 + 4 = 8$ Kombinationen bewertet haben statt aller Verkaufspfade — bei größeren Problemen ist dieser Unterschied dramatisch.

```python
#!/usr/bin/env python3

# Bellman_Minimalbeispiel.py
"""
Kapitel Dynamische Programmierung: Die Handrechnung zur Rueckwaertsinduktion als Code.
Zeigt die Wertfunktionstabelle und die optimale Politik Schritt fuer Schritt.
"""

import numpy as np

GESAMT = 3          # zu verkaufende Einheiten
PERIODEN = 2        # Anzahl Verkaufsperioden


def kosten(menge: int) -> float:
    """Ueberproportionale Marktauswirkung: doppelte Menge kostet vierfach."""
    return float(menge ** 2)
```

```
def loese_rueckwaerts():
    # V[t, x] = minimale Restkosten, wenn zu Beginn von Periode t noch x Stueck offen sind
    V = np.full((PERIODEN + 1, GESAMT + 1), np.inf)
    politik = np.zeros((PERIODEN, GESAMT + 1), dtype=int)

    # Endbedingung: nach der letzten Periode darf nichts mehr offen sein
    V[PERIODEN, 0] = 0.0

    print("=" * 70)
    print("  RUECKWAERTSINDUKTION SCHRITT FUER SCHRITT")
    print("=" * 70)

    for t in range(PERIODEN - 1, -1, -1):
        letzte_periode = (t == PERIODEN - 1)
        print(f"\nStufe t = {t}" + ("  (letzte Periode: alles muss weg)" if letzte_periode
                                    else "  (freie Wahl der Menge)"))
        print(f"  {'Zustand x':>10} | {'beste Aktion':>12} | {'Sofortkosten':>13} | "
              f"{'V[t+1]':>9} | {'V[t]':>8}")
        print("  " + "-" * 62)

        for x in range(GESAMT + 1):
            aktionen = [x] if letzte_periode else range(x + 1)
            bester_wert, beste_aktion, beste_teile = np.inf, 0, (0.0, 0.0)

            for n in aktionen:
                rest = x - n
                sofort = kosten(n)
                zukunft = V[t + 1, rest]
                gesamt = sofort + zukunft
                if gesamt < bester_wert:
                    bester_wert, beste_aktion = gesamt, n
                    beste_teile = (sofort, zukunft)

            V[t, x] = bester_wert
            politik[t, x] = beste_aktion
            print(f"  {x:>10} | {beste_aktion:>12} | {beste_teile[0]:>13.1f} | "
                  f"{beste_teile[1]:>9.1f} | {bester_wert:>8.1f}")

    return V, politik


if __name__ == "__main__":
    V, politik = loese_rueckwaerts()

    # --- Vorwaertspfad: der optimalen Politik folgen --------------------
    print("\n" + "=" * 70)
```

```
    print("  OPTIMALER PFAD (Vorwaertssimulation)")
    print("=" * 70)
    bestand = GESAMT
    gesamtkosten = 0.0
    for t in range(PERIODEN):
        aktion = politik[t, bestand]
        gesamtkosten += kosten(aktion)
        print(f"  Periode {t}: Bestand {bestand} -> verkaufe {aktion} "
              f"(Kosten {kosten(aktion):.1f}) -> Rest {bestand - aktion}")
        bestand -= aktion

    print(f"\n  Gesamtkosten: {gesamtkosten:.1f}  (V[0, {GESAMT}] = {V[0, GESAMT]:.1f})")
    assert abs(gesamtkosten - V[0, GESAMT]) < 1e-9, "Pfadkosten != Wertfunktion!"

    # --- Vergleich mit naiven Strategien ---------------------------------
    alles_sofort = kosten(GESAMT)
    print(f"\n  Zum Vergleich - alles in Periode 0 verkaufen: {alles_sofort:.1f}")
    print(f"  Ersparnis durch Stueckelung: {alles_sofort - gesamtkosten:.1f} "
          f"({(1 - gesamtkosten/alles_sofort)*100:.0f} %)")
    print("=" * 70)
```

13.4 Das Almgren-Chriss-Problem: optimale Orderausführung

Ein klassisches OR-Problem an Börsen: Ein institutioneller Händler muss $X_0 = 100\,000$ Aktien innerhalb von T Handelsperioden verkaufen.

Der Zielkonflikt:

1. **Marktauswirkung (*market impact*, Slippage)**. Verkauft man zu schnell, bricht das Orderbuch ein — der Ausführungspreis verschlechtert sich. Diese Kosten wachsen **überproportional** mit der Ordergröße.
2. **Zeitrisiko (*timing risk*).** Wartet man zu lange, schwankt der Marktpreis. Je größer der noch offene Restbestand, desto mehr Geld liegt im Risiko.

Das Modell nach Almgren und Chriss (2000) minimiert die Summe aus beidem:

$$C(n_t, X_t) = \underbrace{\eta\, n_t^2}_{\text{Marktauswirkung}} + \underbrace{\lambda\, \sigma^2 X_t^2}_{\text{Risiko des Restbestands}}$$

 Formel-Lesehilfe * n_t — in Periode t verkaufte Stückzahl. Quadriert, weil doppelte Menge mehr als doppelten Preisdruck erzeugt. * η („eta“) — Slippage-Koeffizient. Wie stark reagiert der Markt auf Volumen? * X_t — Restbestand **nach** dem Verkauf. Quadriert, weil Varianz quadratisch mit der Positionsgröße wächst. * λ — Risikoaversion: Wie sehr stört mich Schwankung im Vergleich zu Slippage? * σ — Volatilität je Periode.

Ohne Formel gesagt: „Schnell verkaufen kostet Preisabschlag. Langsam verkaufen kostet Nervenkitzel. Finde die Mitte.“

Die zwei Extremfälle: * $\lambda \to 0$ (risikoneutral): gleichmäßiges Aufteilen auf alle Perioden minimiert $\sum n_t^2$ bei fester Summe. * $\lambda \to \infty$ (extrem risikoscheu): sofort alles verkaufen, um kein Risiko zu tragen.

Zur Kalibrierung des Risikoparameters

Ein häufiger Fehler ist ein undokumentierter Skalierungsfaktor im Risikoterm, etwa:

```
holding_risk = risk_aversion * (sigma_period ** 2) * (remaining ** 2) * 1e5
```

Woher eine solche Zahl kommt, bleibt dann unklar. Faktisch wirkt sie als versteckte Erhöhung der Risikoaversion um fünf Größenordnungen — ein Parameter wie `risk_aversion = 1e-6` würde dann in Wahrheit 0,1 bedeuten. Solche „magischen Zahlen“ machen ein Modell unkalibrierbar: Niemand kann sagen, ob 1e-6 viel oder wenig ist.

Der Parameter lässt sich stattdessen herleiten. Beide Kostenterme werden in **Euro** gerechnet: * Marktauswirkung: $\eta \cdot n_t^2$ mit η in €/Stück². * Risiko: $\frac{\lambda}{2} \cdot \sigma^2_{\text{Periode}} \cdot P_0^2 \cdot X_t^2$ — die Varianz des Werts der offenen Position, multipliziert mit der Risikoaversion.

Durch die Multiplikation mit P_0^2 (dem Quadrat des Aktienkurses) stimmen die Einheiten, und λ wird interpretierbar: Es ist der Preis, den man je Einheit Wertvarianz zu zahlen bereit ist.

```
#!/usr/bin/env python3

# Mehrperiodige_Order_Execution.py
"""
Kapitel Dynamische Programmierung: Dynamische Programmierung fuer optimale Orderausfuehrung
(Almgren-Chriss-Rahmen, geloest per Rueckwaertsinduktion).

Eigenschaften:
  * Risikoterm sauber hergeleitet ueber den Aktienkurs P0 (Einheiten: EUR),
    ohne undokumentierte Skalierungsfaktoren
  * Vergleich mit der analytischen Almgren-Chriss-Loesung
  * Vergleich mit naiven Strategien (alles sofort / gleichmaessig)
  * Sensitivitaet gegenueber der Risikoaversion
"""

import os

import numpy as np
import pandas as pd
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt

OUTPUT_DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)), "output")
os.makedirs(OUTPUT_DIR, exist_ok=True)

# --- Parameter -------------------------------------------------------------
```

```
GESAMTBESTAND = 100_000        # X_0, zu verkaufende Aktien
PERIODEN = 5                   # T Handelsperioden
KURS = 50.0                    # P_0 in EUR, zur Skalierung des Risikoterms
ETA = 2.5e-6                   # EUR je Stueck^2 (Slippage-Koeffizient)
VOLA_JAHR = 0.30               # 30 % p.a.
HANDELSSTUNDEN_JAHR = 252 * 6.5
RISIKOAVERSION = 1e-5          # 1/EUR; kalibriert, siehe Kommentar unten
SCHRITTWEITE = 1000            # Diskretisierung des Zustandsraums

# Kalibrierungshinweis: Die dimensionslose Kennzahl des Modells ist
#     kappa_tilde^2 = lambda * sigma_periode^2 * P0^2 / eta
# Sie entscheidet ueber den Charakter der Loesung:
#     << 1  -> praktisch gleichmaessige Aufteilung (Risiko spielt keine Rolle)
#     ~  1  -> ausgewogener Kompromiss  <- hier: 0.55
#     >> 1  -> fast alles sofort verkaufen
# Genau diese Interpretierbarkeit geht mit einem undokumentierten
# Skalierungsfaktor verloren.

VOLA_PERIODE = VOLA_JAHR / np.sqrt(HANDELSSTUNDEN_JAHR)


def periodenkosten(verkauf: float, restbestand: float) -> float:
    """
    Sofortkosten einer Periode in EUR:
      (1) Marktauswirkung: eta * n^2
      (2) Risiko des Restbestands: lambda/2 * sigma^2 * P0^2 * X^2
          -> P0^2 macht aus "Stueck^2" einen EUR^2-Wert; lambda hat damit
             die Einheit 1/EUR und ist interpretierbar.
    """
    marktauswirkung = ETA * verkauf ** 2
    wertvarianz = (VOLA_PERIODE ** 2) * (KURS ** 2) * (restbestand ** 2)
    risiko = 0.5 * RISIKOAVERSION * wertvarianz
    return marktauswirkung + risiko


def loese_dp():
    """Rueckwaertsinduktion ueber den diskretisierten Zustandsraum."""
    zustaende = np.arange(0, GESAMTBESTAND + SCHRITTWEITE, SCHRITTWEITE)
    anzahl = len(zustaende)

    V = np.full((PERIODEN + 1, anzahl), np.inf)
    politik = np.zeros((PERIODEN, anzahl), dtype=int)
    V[PERIODEN, 0] = 0.0                    # am Ende muss alles verkauft sein

    for t in range(PERIODEN - 1, -1, -1):
        for idx, bestand in enumerate(zustaende):
            if t == PERIODEN - 1:
                moegliche = [bestand]       # letzte Periode: Rest muss weg
```

```
            else:
                moegliche = zustaende[zustaende <= bestand]

            bester_wert, beste_aktion = np.inf, 0
            for verkauf in moegliche:
                rest = bestand - verkauf
                rest_idx = int(round(rest / SCHRITTWEITE))
                gesamt = periodenkosten(verkauf, rest) + V[t + 1, rest_idx]
                if gesamt < bester_wert:
                    bester_wert, beste_aktion = gesamt, int(verkauf)

            V[t, idx] = bester_wert
            politik[t, idx] = beste_aktion

    return zustaende, V, politik


def analytische_loesung():
    """
    Geschlossene Almgren-Chriss-Loesung fuer den kontinuierlichen Fall.
    Der optimale Pfad ist X_t = X_0 * sinh(kappa*(T-t)) / sinh(kappa*T)
    mit kappa = arccosh(tilde_kappa^2/2 + 1), tilde_kappa^2 = lambda*sigma^2*P0^2/eta.
    Dient hier als unabhaengige Kontrolle des DP-Ergebnisses.
    """
    kappa_tilde_quadrat = (RISIKOAVERSION * (VOLA_PERIODE ** 2) * (KURS ** 2)) / ETA
    kappa = np.arccosh(kappa_tilde_quadrat / 2.0 + 1.0)
    if kappa < 1e-12:                       # Grenzfall: risikoneutral -> linear
        return np.linspace(GESAMTBESTAND, 0, PERIODEN + 1)
    t = np.arange(PERIODEN + 1)
    return GESAMTBESTAND * np.sinh(kappa * (PERIODEN - t)) / np.sinh(kappa * PERIODEN)


def bewerte_pfad(bestaende):
    """Gesamtkosten eines beliebigen Bestandspfades."""
    summe = 0.0
    for t in range(len(bestaende) - 1):
        verkauf = bestaende[t] - bestaende[t + 1]
        summe += periodenkosten(verkauf, bestaende[t + 1])
    return summe


if __name__ == "__main__":
    zustaende, V, politik = loese_dp()

    # --- Vorwaertspfad der optimalen Politik ----------------------------
    bestand = GESAMTBESTAND
    verlauf = [bestand]
    verkaeufe = []
```

```
    for t in range(PERIODEN):
        idx = int(round(bestand / SCHRITTWEITE))
        verkauf = politik[t, idx]
        verkaeufe.append(verkauf)
        bestand -= verkauf
        verlauf.append(bestand)

    print("=" * 84)
    print("   OPTIMALE MEHRPERIODIGE ORDER-EXECUTION (BELLMAN DP)")
    print("=" * 84)
    print(f"Gesamtvolumen:       {GESAMTBESTAND:,} Stueck zu {KURS:.2f} EUR "
          f"= {GESAMTBESTAND*KURS:,.0f} EUR Positionswert")
    print(f"Zeithorizont:        {PERIODEN} Handelsperioden")
    print(f"Volatilitaet:        {VOLA_JAHR*100:.0f} % p.a. "
          f"= {VOLA_PERIODE*100:.3f} % je Periode")
    print(f"Slippage eta:        {ETA:.2e} EUR/Stueck^2")
    print(f"Risikoaversion:      {RISIKOAVERSION:.2e} 1/EUR")
    print(f"Erwartete Gesamtreibung: {V[0, -1]:,.2f} EUR "
          f"({V[0, -1]/(GESAMTBESTAND*KURS)*10000:.1f} Basispunkte)\n")

    plan = pd.DataFrame([{
        "Periode": f"t = {t} -> {t+1}",
        "Startbestand": f"{verlauf[t]:,}",
        "Verkauf n_t": f"{verkaeufe[t]:,}",
        "Restbestand": f"{verlauf[t+1]:,}",
        "Anteil": f"{verkaeufe[t]/GESAMTBESTAND*100:5.1f} %",
        "Kosten (EUR)": f"{periodenkosten(verkaeufe[t], verlauf[t+1]):,.0f}",
    } for t in range(PERIODEN)])
    print(plan.to_string(index=False))

    # --- Vergleich mit Alternativen und der analytischen Loesung ---------
    sofort = [GESAMTBESTAND] + [0] * PERIODEN
    gleichmaessig = [GESAMTBESTAND * (1 - t / PERIODEN) for t in range(PERIODEN + 1)]
    analytisch = analytische_loesung()

    print("\n" + "-" * 84)
    print(f"{'Strategie':<34} {'Kosten (EUR)':>15} {'Basispunkte':>13} "
          f"{'ggue. Optimum':>16}")
    print("-" * 84)
    optimum = V[0, -1]
    for name, pfad in [("DP-Optimum", verlauf),
                       ("Analytisch (Almgren-Chriss)", list(analytisch)),
                       ("Gleichmaessig (TWAP)", gleichmaessig),
                       ("Alles sofort", sofort)]:
        kosten = bewerte_pfad(pfad)
        bp = kosten / (GESAMTBESTAND * KURS) * 10000
        print(f"{name:<34} {kosten:>15,.0f} {bp:>12.1f} "
              f"{kosten - optimum:>+15,.0f}")
```

```
    print("-" * 84)
    abweichung = abs(bewerte_pfad(list(analytisch)) - optimum) / optimum
    print(f"Abweichung DP zur analytischen Loesung: {abweichung*100:.3f} % "
          f"(Diskretisierung: {SCHRITTWEITE} Stueck)")

    # --- Sensitivitaet gegenueber der Risikoaversion ---------------------
    print("\n--- Wie wirkt die Risikoaversion? ---")
    print(f"{'lambda':>10} | {'Verkauf in Periode 0':>22} | {'Charakter':<28}")
    print("-" * 70)
    for lam in [1e-7, 1e-6, 1e-5, 1e-4, 1e-3]:
        globals()["RISIKOAVERSION"] = lam
        _, V_l, pol_l = loese_dp()
        erste = pol_l[0, -1]
        anteil = erste / GESAMTBESTAND * 100
        charakter = ("nahezu gleichmaessig" if anteil < 25 else
                     "front-loaded" if anteil < 60 else "fast alles sofort")
        print(f"{lam:>10.0e} | {erste:>13,} ({anteil:5.1f} %) | {charakter:<28}")
    globals()["RISIKOAVERSION"] = 1e-5       # zuruecksetzen

    # --- Diagramm --------------------------------------------------------
    plt.figure(figsize=(10, 5.5))
    plt.plot(range(PERIODEN + 1), verlauf, "o-", lw=2.5, label="DP-Optimum")
    plt.plot(range(PERIODEN + 1), analytisch, "s--", lw=1.8, alpha=0.8,
             label="Analytisch (Almgren-Chriss)")
    plt.plot(range(PERIODEN + 1), gleichmaessig, "^:", lw=1.8, alpha=0.8,
             label="Gleichmaessig (TWAP)")
    plt.bar(range(PERIODEN), verkaeufe, alpha=0.25, color="orange", width=0.45,
            label="Verkaufstranche $n_t$")
    plt.title("Optimaler Liquidationspfad ueber diskrete Perioden", fontsize=12)
    plt.xlabel("Handelsperiode $t$")
    plt.ylabel("Verbleibender Bestand $X_t$")
    plt.xticks(range(PERIODEN + 1))
    plt.grid(True, linestyle=":", alpha=0.6)
    plt.legend()
    plt.tight_layout()
    ziel = os.path.join(OUTPUT_DIR, "optimal_execution_dp.png")
    plt.savefig(ziel, dpi=150)
    print(f"\nDiagramm gespeichert unter '{ziel}'")
    print("=" * 84)
```

Erwartete Ausgabe:

```
====================================================================================
   OPTIMALE MEHRPERIODIGE ORDER-EXECUTION (BELLMAN DP)
====================================================================================
Gesamtvolumen:       100,000 Stueck zu 50.00 EUR = 5,000,000 EUR Positionswert
Zeithorizont:        5 Handelsperioden
```

```
Volatilitaet:        30 % p.a. = 0.741 % je Periode
Slippage eta:        2.50e-06 EUR/Stueck^2
Risikoaversion:      1.00e-05 1/EUR
Erwartete Gesamtreibung: 10,266.24 EUR (20.5 Basispunkte)

    Periode Startbestand Verkauf n_t Restbestand Anteil Kosten (EUR)
t = 0 -> 1       100,000      41,000      59,000  41.0 %        6,593
t = 1 -> 2        59,000      25,000      34,000  25.0 %        2,356
t = 2 -> 3        34,000      16,000      18,000  16.0 %          863
t = 3 -> 4        18,000      10,000       8,000  10.0 %          294
t = 4 -> 5         8,000       8,000           0   8.0 %          160

--------------------------------------------------------------------------------
Strategie                              Kosten (EUR)   Basispunkte    ggue. Optimum
--------------------------------------------------------------------------------
DP-Optimum                                   10,266          20.5               +0
Analytisch (Almgren-Chriss)                  10,860          21.7             +594
Gleichmaessig (TWAP)                         13,242          26.5           +2,976
Alles sofort                                 25,000          50.0          +14,734
--------------------------------------------------------------------------------
Abweichung DP zur analytischen Loesung: 5.788 % (Diskretisierung: 1000 Stueck)

--- Wie wirkt die Risikoaversion? ---
    lambda |   Verkauf in Periode 0 | Charakter
------------------------------------------------------------------
     1e-07 |        20,000 ( 20.0 %) | nahezu gleichmaessig
     1e-06 |        23,000 ( 23.0 %) | nahezu gleichmaessig
     1e-05 |        41,000 ( 41.0 %) | front-loaded
     1e-04 |        78,000 ( 78.0 %) | fast alles sofort
     1e-03 |        97,000 ( 97.0 %) | fast alles sofort
================================================================================
```

Die Lösung ist *front-loaded*: 41 % im ersten Schritt, dann fallend (25 %, 16 %, 10 %, 8 %). Das ist die typische Form — man baut Risiko früh ab, aber nicht abrupt. Die Sensitivitätstabelle zeigt, wie λ zwischen den beiden Extremen steuert: Bei $\lambda = 10^{-7}$ verkauft das Modell gleichmäßig (Risiko ist egal), bei $\lambda = 10^{-3}$ praktisch alles sofort (Risiko dominiert).

Der Vergleich beziffert den Nutzen: Gegenüber der naiven TWAP-Strategie (gleiche Tranchen) spart die optimierte Ausführung **2 976 €** oder 6 Basispunkte — bei einer Position von 5 Mio. €. Gegenüber „alles sofort“ sind es 14 734 €.

Code-Durchgang: die analytische Gegenprobe und was ihre Abweichung bedeutet

Das Almgren-Chriss-Problem hat eine **geschlossene Lösung**:

$$X_t = X_0 \,\frac{\sinh(\kappa(T-t))}{\sinh(\kappa T)}, \qquad \kappa = \operatorname{arcosh}\!\Big(\tfrac{\tilde{\kappa}^2}{2}+1\Big), \qquad \tilde{\kappa}^2 = \frac{\lambda\,\sigma^2 P_0^2}{\eta}$$

Sie liefert hier 10 860 € gegenüber 10 266 € beim DP — **eine Abweichung von 5,8 %, und zwar zugunsten des DP.** Das ist kein Fehler, sondern lehrreich, und es hat zwei Ursachen:

1. **Unterschiedliche Zeitkonvention.** Die geschlossene Formel gilt für die zeitkontinuierliche Variante des Modells, in der das Risiko über das gesamte Intervall integriert wird. Unser diskretes Modell belastet dagegen den Bestand **nach** dem Verkauf. Beide Varianten sind legitim, sie lösen aber leicht verschiedene Probleme.
2. **Diskretisierung.** Der Zustandsraum ist in 1000er-Schritten gerastert; die analytische Lösung darf beliebige Stückzahlen wählen.

Die Lehre daraus ist wichtiger als die Zahl: Eine Vergleichsrechnung, die *ungefähr* passt, bestätigt die Größenordnung und die Form der Lösung — beide Pfade sind front-loaded, beide liegen bei rund 21 Basispunkten. Sie beweist aber nicht die Punktgenauigkeit, solange die Modellkonventionen nicht identisch sind. **Wer zwei Zahlen vergleicht, muss zuerst prüfen, ob sie dasselbe messen.**

Übernehmen Sie dennoch das Prinzip: Wo immer eine unabhängige zweite Rechnung möglich ist — Formel, Simulation, Handrechnung —, bauen Sie sie ein. Hätte das DP hier 500 000 € oder 12 € geliefert, wäre der Fehler sofort aufgefallen.

13.5 Der Fluch der Dimensionalität

DP ist mächtig, aber es hat eine harte Grenze. Der Aufwand der Rückwärtsinduktion ist

$$\mathcal{O}(T \cdot |\mathcal{S}| \cdot |\mathcal{A}|)$$

— Perioden mal Zustände mal Aktionen. Im Beispiel oben: $5 \times 101 \times 101 \approx 51\,000$ Auswertungen, in Sekundenbruchteilen erledigt.

Das Problem entsteht, sobald der Zustand **mehrere Dimensionen** hat:

Zustandsbeschreibung	Zustandsraum	Machbar?
Restbestand (101 Stufen)	101	□ trivial
+ aktueller Kurs (50 Stufen)	5 050	□ leicht
+ Orderbuchtiefe (20 Stufen)	101 000	□ noch gut
+ 10 weitere Titel mit je 101 Stufen	$101^{11} \approx 10^{22}$	□ hoffnungslos

Das ist der Fluch der Dimensionalität (Bellmans eigener Begriff): Jede zusätzliche Zustandsvariable **multipliziert** den Aufwand.

Gegenmittel: * **Zustandsraum verkleinern:** gröber diskretisieren, irrelevante Variablen weglassen. * **Approximate Dynamic Programming:** V_t durch eine parametrische Funktion annähern statt tabellarisch zu speichern. * **Reinforcement Learning:** dieselbe Bellman-Gleichung, aber V wird aus Erfah-

rung gelernt (Q-Learning) statt vollständig berechnet. * **Nach geschlossenen Lösungen suchen** — wie bei Almgren-Chriss. Wo eine Formel existiert, ist sie unschlagbar.

13.6 Übungsaufgaben

Lösungen: Abschnitt A.13.

Aufgabe 13.1 □ **— Bausteine benennen.** Ein Wanderer plant eine 5-Tages-Tour und muss täglich entscheiden, wie weit er läuft. Bestimmen Sie Stufe, Zustand, Aktion und Wertfunktion. Was gehört **nicht** in den Zustand?

Aufgabe 13.2 □ **— Optimalitätsprinzip anwenden.** Warum folgt aus dem Optimalitätsprinzip, dass man rückwärts rechnen darf? Was würde schiefgehen, wenn die Kosten einer Periode auch von **früheren** Aktionen abhängen (und nicht nur vom aktuellen Zustand)?

Aufgabe 13.3 □□ **— Rückwärtsinduktion von Hand.** 5 Einheiten in 3 Perioden, Kosten $C(n) = n^2 + 2n$. Erstellen Sie die vollständige Wertfunktionstabelle und bestimmen Sie den optimalen Pfad. Prüfen Sie mit `Bellman_Minimalbeispiel.py` (angepasst).

Aufgabe 13.4 □□ **— Rucksackproblem als DP.** Lösen Sie das Rucksackproblem aus Kapitel 6 mit dynamischer Programmierung statt MILP. (Zustand: verbleibende Kapazität; Stufe: betrachteter Gegenstand.) Vergleichen Sie Laufzeit und Ergebnis mit dem MILP-Solver.

Aufgabe 13.5 □□□ **— Risikoaversion kalibrieren.** Untersuchen Sie mit `Mehrperiodige_Order_Execution.py`: (a) Bei welchem λ verkauft das Modell in der ersten Periode mehr als 50 %? (b) Stellen Sie den Zusammenhang zwischen λ und den erwarteten Gesamtkosten dar. (c) Ein Händler sagt: „Ich will höchstens 20 % Marktauswirkungskosten und den Rest an Risiko." Welches λ setzen Sie?

Aufgabe 13.6 □□□ **— Zustandsraum erweitern.** Erweitern Sie das Ausführungsmodell um einen zweiten Zustand: die aktuelle **Orderbuchtiefe** (3 Stufen: dünn/normal/tief), die mit gegebenen Übergangswahrscheinlichkeiten wechselt und η um Faktor 2 / 1 / 0,5 skaliert. Wie ändert sich die Strategie? Wie stark wächst die Rechenzeit?

13.7 Finde den Denkfehler

Bei dynamischer Programmierung entscheidet eine einzige Frage über Erfolg oder Misserfolg: **Was gehört in den Zustand?** Ist er zu klein, rechnet das Verfahren völlig korrekt — nur eben an einem anderen Problem.

□ Finde den Denkfehler: Der Zustand, der zu wenig weiß

Eine Werkstatt plant vier Perioden. Der Bedarf beträgt 3, 1, 4, 2 Stück; produziert werden können höchstens 5 je Periode, gelagert höchstens 6 Stück.

Kostenart	Höhe
Stückkosten	4 €

Kostenart	Höhe
Lagerkosten je Stück und Periode	2 €
Rüstkosten — fallen an, wenn produziert wird **und in der Vorperiode nicht**	30 €

Der Entwickler überlegt: *„Für die Zukunft zählt nur, wie viel ich auf Lager habe. Also ist der Lagerbestand mein Zustand.“*

```
import functools

@functools.lru_cache(None)
def V(t, lager):
    if t == 4:
        return (0.0, ())
    best = (float("inf"), ())
    for p in range(KAP + 1):
        neu = lager + p - BEDARF[t]
        if neu < 0 or neu > LAGER_MAX:
            continue
        kosten = (RUESTKOSTEN if p > 0 else 0) + STUECKKOSTEN * p + LAGERKOSTEN *
 ↪  neu
        rest, plan = V(t + 1, neu)
        if kosten + rest < best[0]:
            best = (kosten + rest, (p,) + plan)
    return best
```

Ergebnis: Plan $(5, 0, 5, 0)$ für **110 €**. Nachgerechnet stimmt der Betrag sogar — dieser Plan kostet tatsächlich 110 €. Das wahre Optimum lautet aber $(3, 1, 4, 2)$ und kostet **70 €**.

Ihre Aufgabe: (a) Sehen Sie sich die Zeile mit den Rüstkosten an. Welche Information bräuchte sie, die im Zustand `(t, lager)` nicht enthalten ist? (b) Welchen Vorteil des Plans $(3, 1, 4, 2)$ kann dieses Modell prinzipiell nicht erkennen? (c) Wie lautet der korrigierte Zustand, und um welchen Faktor wächst der Zustandsraum dadurch? (d) Warum ist es besonders tückisch, dass der ausgegebene Kostenbetrag *richtig* war?

Auflösung: Abschnitt A.13.

 Merksatz Der Zustand muss **alles** enthalten, was die Zukunft beeinflusst — und **nichts** darüber hinaus. Zu wenig, und das Modell löst ein anderes Problem, ohne es zu merken. Zu viel, und es wird unnötig groß (Abschnitt 13.5). Die Probe dafür ist ein einziger Satz: *„Wenn ich nur den Zustand kenne und nicht den Weg dorthin — kann ich dann noch optimal weiterentscheiden?“* Lautet die Antwort nein, fehlt etwas.

13.8 Micro-Quiz

Micro-Quiz 13: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Warum rechnet dynamische Programmierung rückwärts? (a) Weil Rekursion in Python rückwärts effizienter ist. (b) Weil V_t den Wert V_{t+1} voraussetzt: Man kann eine Entscheidung erst bewerten, wenn man weiß, was sie für die Zukunft bedeutet. Am Ende ist dieser Wert bekannt — dort beginnt man. (c) Weil die Kosten in späteren Perioden höher sind.

2. Was besagt das Bellmansche Optimalitätsprinzip? (a) Jede optimale Lösung besteht aus lauter einzeln optimalen Schritten — man kann also gierig vorgehen. (b) Ist ein Weg insgesamt optimal, so ist auch sein **Reststück** ab jedem Zwischenzustand optimal. Deshalb genügt es, je Zustand einen einzigen Wert zu speichern. (c) Bei genügend Rechenzeit findet man das Optimum immer.

3. Ihr DP-Modell für eine Lagerplanung hat den Zustand „Lagerbestand". Nun kommt eine Mengenrabattstaffel dazu, die sich nach der bisher im Jahr bestellten Gesamtmenge** richtet. Was folgt?** (a) Nichts — der Rabatt betrifft nur die Kosten, nicht den Zustand. (b) Die kumulierte Jahresmenge muss in den Zustand, sonst kann das Modell den Rabatt nicht korrekt zuordnen. Der Zustandsraum wird dadurch erheblich größer. (c) Man muss auf ein MILP wechseln, DP ist hier grundsätzlich ungeeignet.

13.9 Selbsttest

Antworten: Anhang A.

1. Formulieren Sie das Optimalitätsprinzip in eigenen Worten.
2. Warum rechnet man bei DP rückwärts und nicht vorwärts?
3. Was muss ein Zustand enthalten — und woran erkennt man, dass er unvollständig ist?
4. Was besagt der Fluch der Dimensionalität, und welche drei Gegenmittel gibt es?
5. Warum ist eine analytische Vergleichslösung wertvoll, wenn man schon eine numerische hat?

13.10 Zusammenfassung

- **Das Optimalitätsprinzip** erlaubt es, ein mehrstufiges Problem in ineinandergreifende Einperiodenprobleme zu zerlegen.
- **Rückwärtsinduktion** startet am bekannten Ende und arbeitet sich nach vorn — dadurch ist V_{t+1} immer schon bekannt, wenn V_t berechnet wird.
- **Die Zustandsdefinition ist die eigentliche Modellierungsleistung**: vollständig, aber so knapp wie möglich. Die Probe dafür ist ein Satz: *„Wenn ich nur den Zustand kenne und nicht den Weg dorthin — kann ich dann noch richtig weiterentscheiden?"* Fehlt etwas, rechnet das Verfahren korrekt an einem anderen Problem, ohne es zu melden.
- **Die Wertfunktion ist eine Nachschlagetabelle, kein Plan.** Ihr praktischer Ertrag ist eine Regel für jeden Zustand — die auch dann noch gilt, wenn es anders kommt als gedacht.

- **Gierige Vorwärtsregeln scheitern systematisch**, weil sie die Restkosten nicht kennen: Im Schnellstart kostet der billigere erste Abschnitt am Ende 50 % mehr.
- **Bei überproportionalen Kosten lohnt sich Stückelung** — das ist der ökonomische Kern des Ausführungsproblems.
- **Magische Konstanten sind ein Warnsignal.** Jeder Parameter braucht eine Einheit und eine Interpretation, sonst ist das Modell nicht kalibrierbar.
- **Der Fluch der Dimensionalität** begrenzt DP auf wenige Zustandsdimensionen.

Ausblick. Teil IV führt alles zusammen: Ab Kapitel 18 arbeiten wir mit echten Marktdaten — und lernen zuerst, warum diese Daten trügerisch sind.

Kapitel 14: Mehrere Ziele — Pareto-Fronten statt Gewichte

Kapitel auf einen Blick

Worum geht es? Um den Normalfall, den die bisherigen Kapitel umgangen haben: Es gibt nicht *ein* Ziel, sondern zwei, die gegeneinander stehen. Kosten und CO□, Termintreue und Bestand, Rendite und Risiko.

Voraussetzungen: Kapitel 4 (die Zielgröße als Entscheidung) und Kapitel 6. Für den Denkfehler hilft Kapitel 19.

Danach können Sie: begründen, warum eine gewichtete Summe bei ganzzahligen Modellen Kompromisse **grundsätzlich** nicht erreichen kann; eine vollständige Pareto-Front mit dem ε-Constraint-Verfahren berechnen; und einer Geschäftsführung eine Entscheidung vorlegen, die sie tatsächlich treffen kann.

Zeitbedarf: ca. 4 Stunden.

Programme:
`Mehrziel_Pareto.py`

Notebook: `Notebooks_04/mehrziel.ipynb`

14.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Was kostet ein Kilogramm CO□?

Eine Spedition vergibt fünf Sendungen an LKW, Bahn oder Kombinierten Verkehr. Der LKW ist billig und schmutzig, die Bahn teuer und sauber — und es sind nur zwei Trassen frei.

Statt zu fragen „wie wichtig ist uns CO□?" fragen wir: **Was bekommen wir für zwei Prozent mehr Geld?**

```
import numpy as np
from scipy.optimize import linprog

#                    LKW   Bahn  Kombi    LKW billig+schmutzig, Bahn teuer+sauber
```

```
kosten = np.array([[  900, 1150, 1020], [ 620,  790,  700], [1300, 1660, 1470],
                   [  480,  610,  545], [1050, 1340, 1180]], dtype=float)
co2    = np.array([[ 1400,  310,  760], [ 980,  220,  530], [2000,  440, 1080],
                   [  760,  170,  410], [1620,  360,  880]], dtype=float)
eine_pro_sendung = np.repeat(np.eye(5), 3, axis=1)
bahn_trassen     = [[1.0 if j == 1 else 0.0 for _ in range(5) for j in range(3)]]

def plane(ziel, kostenbudget=None):
    A, b = list(bahn_trassen), [2.0]                    # nur zwei Trassen frei
    if kostenbudget is not None:
        A.append(kosten.reshape(-1)); b.append(kostenbudget)
    x = linprog(ziel.reshape(-1), A_ub=A, b_ub=b, A_eq=eine_pro_sendung,
                b_eq=np.ones(5), bounds=(0, 1), integrality=1, method="highs").x
    return kosten.reshape(-1) @ x, co2.reshape(-1) @ x

k0, c0 = plane(kosten)                                  # nur Kosten minimieren
k1, c1 = plane(co2, kostenbudget=k0 * 1.02)              # 2 % mehr ausgeben
print(f"guenstigster Plan:  {k0:>7,.0f} EUR   {c0:>6,.0f} kg CO2")
print(f"mit 2 % Aufpreis:   {k1:>7,.0f} EUR   {c1:>6,.0f} kg CO2   "
      f"({(c0-c1)/c0:.0%} weniger, {(k1-k0)/(c0-c1):.2f} EUR je kg)")
```

Ausgabe:

```
guenstigster Plan:    4,350 EUR    6,760 kg CO2
mit 2 % Aufpreis:     4,430 EUR    6,310 kg CO2   (7% weniger, 0.18 EUR je kg)
```

Und jetzt der Punkt. 80 Euro mehr sparen 450 Kilogramm CO□ — **18 Cent je Kilogramm**. Der europäische Emissionshandel bewegt sich um ein Vielfaches davon. Die Entscheidung ist damit keine Wertefrage mehr, sondern eine Rechnung.

Beachten Sie, was hier *nicht* passiert ist: Niemand musste ein Gewicht festlegen. Die zweite Zeile setzt eine **Schranke** („höchstens 2 % teurer") und optimiert dann das andere Ziel. Das Ergebnis ist eine Aussage, die ein Kaufmann prüfen kann.

> □ **Merksatz** „Wie wichtig ist Ihnen CO□?" ist keine beantwortbare Frage. „Wir geben zwei Prozent mehr aus — was bringt das?" ist eine.

Warum funktioniert das? Weil aus zwei Zielen ein Ziel und eine Nebenbedingung geworden sind. Das ist die Grundidee des **ε-Constraint-Verfahrens**, um das sich dieses Kapitel dreht — und es ist der Grund, warum es ohne jedes Gewicht auskommt.

14.2 Lernziele

Nach diesem Kapitel können Sie …

1. … Pareto-Dominanz und Pareto-Front definieren und an einem Diagramm zeigen.
2. … erklären, warum die gewichtete Summe nur **gestützte** Lösungen findet — und was das geometrisch heißt.
3. … eine vollständige Pareto-Front mit dem ε-Constraint-Verfahren berechnen, mit so vielen Solverläufen, wie die Front Punkte hat.
4. … lexikografische Optimierung einsetzen, wenn es eine klare Rangfolge der Ziele gibt.
5. … einer Entscheiderin die Front so vorlegen, dass sie eine Wahl treffen kann.

14.3 Wann ein Kompromiss gut ist

Bei einem Ziel ist „besser" eindeutig. Bei zwei Zielen nicht mehr: Ein Plan kann billiger und schmutziger sein als ein anderer, und dann ist keiner von beiden besser.

Vergleichbar sind nur die Fälle, in denen ein Plan in **keinem** Ziel schlechter ist:

$$x \text{ dominiert } y \quad :\Longleftrightarrow \quad f_i(x) \le f_i(y) \ \text{ für alle } i \ \text{ und } \ f_j(x) < f_j(y) \ \text{ für mindestens ein } j$$

Formel-Übersetzer

Mathematik	Alltagssprache
$f_i(x) \le f_i(y)$ für alle i	„Plan x ist in **keinem** Ziel schlechter als y."
$f_j(x) < f_j(y)$ für ein j	„… und in mindestens einem echt besser."
x dominiert y	„y kann man wegwerfen. Es gibt keinen Grund, ihn zu wählen."
x ist **pareto-optimal**	„Niemand dominiert x. Wer x verbessern will, muss anderswo schlechter werden."
Pareto-Front	„Die Speisekarte: alle Pläne, die man ernsthaft in Betracht ziehen kann."

Ohne Formel gesagt: Wegwerfen darf man einen Plan nur, wenn ein anderer ihn in jeder Hinsicht schlägt. Alles Übrige ist eine Frage der Präferenz — und die gehört nicht dem Modellierer.

Definition: Pareto-Front

Die Menge aller nicht dominierten Lösungen. Sie ist das vollständige Ergebnis eines Mehrzielproblems — nicht ein Punkt, sondern eine Liste.

Wer daraus einen einzelnen Punkt macht, trifft eine Entscheidung. Die Frage ist nur, ob er es merkt.

14.4 Der Reflex: alles in ein Ziel rühren

Der naheliegende Weg, zwei Ziele zu einem zu machen, ist die **gewichtete Summe**:

$$\min_{x \in X} \; c(x) + w \cdot e(x)$$

Formel-Übersetzer

Mathematik	Alltagssprache
$c(x)$	Transportkosten des Plans x, in Euro.
$e(x)$	CO-Ausstoß desselben Plans, in Kilogramm.
w	„So viele Euro ist mir ein Kilogramm CO wert.“ Ein **Wechselkurs**, kein Wichtigkeitsregler (Abschnitt 7.10).
$\min_{x \in X}$	über alle zulässigen Pläne.

> Der Ausdruck hat eine Einheit: Euro. Damit ist w keine freie Zahl zwischen 0 und 1, sondern ein CO-Preis in €/kg — und den kann man tatsächlich beziffern.

Das ist bequem, liefert zulässige Lösungen und wird in der Praxis fast immer so gemacht. Es hat nur einen Fehler, den man erst sieht, wenn man das Ergebnis mit der vollständigen Front vergleicht.

14.5 Das Programm

Zwölf Sendungen, drei Verkehrsträger, fünf freie Bahntrassen. Der Zielkonflikt entsteht nicht zwischen den Trägern — die Bahn ist immer billiger *und* sauberer als der LKW —, sondern durch die **Knappheit** der Trassen. Genau so sieht es in der Praxis aus.

```
#!/usr/bin/env python3

# Mehrziel_Pareto.py
"""
Kapitel Mehrziel: Kosten gegen CO2 - und warum Gewichte nicht genuegen.

Eine Spedition vergibt zwoelf Sendungen an drei Verkehrstraeger: LKW (schnell,
teuer, schmutzig), Bahn (billig und sauber, aber nur fuenf Trassen frei) und
Kombinierten Verkehr (dazwischen). Zwei Ziele stehen gegeneinander:
Transportkosten und CO2-Ausstoss.

Der uebliche Reflex ist, beide Ziele zu einem zusammenzuruehren:

    minimiere   Kosten + w * CO2

Das ist bequem, liefert zulaessige Loesungen - und ist unvollstaendig. Bei
ganzzahligen Entscheidungen gibt es Kompromisse, die auf diesem Weg
```

```
GRUNDSAETZLICH nicht erreichbar sind, egal welches w man waehlt. Nicht "schwer
zu finden", sondern beweisbar unerreichbar.

Das Programm zeigt in vier Teilen:

  1. Die beiden Extreme - was jedes Ziel allein kostet.
  2. Die lineare Skalarisierung ueber ein feines Gewichtsraster.
  3. Die vollstaendige Pareto-Front ueber das eps-Constraint-Verfahren.
  4. Den Nachweis, dass die Luecke keine Frage des Rasters ist, sondern
     Geometrie: Die fehlenden Punkte liegen strikt oberhalb der konvexen
     Huelle und koennen deshalb von keiner Geraden gestuetzt werden.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
from scipy.optimize import linprog

TRAEGER = ["LKW", "Bahn", "Kombiniert"]
BAHN_TRASSEN = 5          # so viele Sendungen passen hoechstens auf die Bahn
SAAT = 5


def erzeuge_sendungen(anzahl: int = 12, saat: int = SAAT):
    """Kosten und CO2 je Sendung und Verkehrstraeger.

    Die Bahn ist immer billiger und sauberer als der LKW - der Zielkonflikt
    entsteht nicht zwischen den Traegern, sondern durch die KNAPPHEIT der
    Trassen. Genau so sieht es in der Praxis aus.
    """
    rng = np.random.default_rng(saat)
    kosten = np.zeros((anzahl, 3))
    co2 = np.zeros((anzahl, 3))
    for i in range(anzahl):
        grund_kosten = rng.integers(600, 2400)
        grund_co2 = rng.integers(400, 1800)
        kosten[i] = [grund_kosten,
                     grund_kosten * rng.uniform(0.55, 0.85),
                     grund_kosten * rng.uniform(0.70, 1.00)]
        co2[i] = [grund_co2,
                  grund_co2 * rng.uniform(0.15, 0.35),
                  grund_co2 * rng.uniform(0.40, 0.70)]
    return np.round(kosten).astype(int), np.round(co2).astype(int)


KOSTEN, CO2 = erzeuge_sendungen()
```

```
N = len(KOSTEN)


def plane(ziel: np.ndarray, co2_grenze: float | None = None,
          kosten_grenze: float | None = None):
    """Jede Sendung genau einem Traeger zuordnen; Trassen sind knapp.

    'ziel' ist die zu minimierende Matrix (Kosten, CO2 oder eine Mischung).
    Die beiden Grenzen sind das Werkzeug fuer eps-Constraint und
    lexikografische Optimierung - sie machen aus einem Ziel eine Schranke.
    """
    gleichungen = np.zeros((N, N * 3))
    for i in range(N):
        gleichungen[i, i * 3:(i + 1) * 3] = 1.0        # genau ein Traeger

    ungleichungen = [[1.0 if j == 1 else 0.0 for _ in range(N) for j in range(3)]]
    grenzen = [float(BAHN_TRASSEN)]
    if co2_grenze is not None:
        ungleichungen.append(CO2.reshape(-1).astype(float))
        grenzen.append(float(co2_grenze))
    if kosten_grenze is not None:
        ungleichungen.append(KOSTEN.reshape(-1).astype(float))
        grenzen.append(float(kosten_grenze))

    ergebnis = linprog(ziel.reshape(-1).astype(float),
                       A_ub=ungleichungen, b_ub=grenzen,
                       A_eq=gleichungen, b_eq=np.ones(N),
                       bounds=(0, 1), integrality=1, method="highs")
    if not ergebnis.success:
        return None
    plan = np.round(ergebnis.x).astype(int)
    return (int(KOSTEN.reshape(-1) @ plan), int(CO2.reshape(-1) @ plan), plan)


def pareto_front():
    """Vollstaendige Front ueber das eps-Constraint-Verfahren.

    Der Ablauf ist der eigentliche Inhalt dieser Funktion: Erst das
    Kostenminimum bestimmen (der eine Rand der Front), dann die CO2-Schranke
    schrittweise um genau ein Kilogramm unter den zuletzt erreichten Wert
    druecken. Jeder Lauf liefert den naechsten Punkt - und wenn keiner mehr
    zulaessig ist, ist die Front vollstaendig.

    Das braucht so viele Solveraufrufe, wie die Front Punkte hat. Ein Raster
    ueber alle moeglichen CO2-Werte braeuchte hier fast tausend.
    """
    front = []
    start = plane(KOSTEN)
```

```
    grenze = start[1]
    while True:
        ergebnis = plane(KOSTEN, co2_grenze=grenze)
        if ergebnis is None:
            break
        front.append((ergebnis[0], ergebnis[1]))
        grenze = ergebnis[1] - 1
    return front


def skalarisierung(gewichte):
    """Was 'Kosten + w * CO2' fuer viele w hergibt - als Menge von Punkten."""
    gefunden = {}
    for w in gewichte:
        ergebnis = plane(KOSTEN + w * CO2)
        if ergebnis:
            gefunden.setdefault((ergebnis[0], ergebnis[1]), []).append(w)
    return gefunden


def untere_huelle(punkte):
    """Untere linke konvexe Huelle - genau die Punkte, die eine Gerade stuetzt.

    Der Zusammenhang, um den es geht: 'Kosten + w*CO2 minimieren' heisst
    geometrisch, eine Gerade der Steigung -1/w von links unten an die
    Punktwolke zu schieben. Sie beruehrt immer einen Eckpunkt der konvexen
    Huelle. Punkte, die oberhalb liegen, werden nie beruehrt - fuer kein w.
    """
    huelle = []
    for punkt in sorted(punkte):
        while len(huelle) >= 2:
            (x1, y1), (x2, y2) = huelle[-2], huelle[-1]
            if (x2 - x1) * (punkt[1] - y1) - (y2 - y1) * (punkt[0] - x1) <= 0:
                huelle.pop()
            else:
                break
        huelle.append(punkt)
    return huelle


if __name__ == "__main__":
    print("=" * 80)
    print("  KOSTEN GEGEN CO2 - UND WARUM GEWICHTE NICHT GENUEGEN")
    print("=" * 80)
    print(f"{N} Sendungen, {len(TRAEGER)} Verkehrstraeger, "
          f"{BAHN_TRASSEN} freie Bahntrassen.\n")

    # --- 1. Die beiden Extreme ---------------------------------------
```

```
    guenstigst = plane(KOSTEN)
    saubersten = plane(CO2)
    print("1. Was jedes Ziel allein ergibt\n")
    print(f"  {'':<22} {'Kosten':>10} {'CO2':>10}")
    print("  " + "-" * 44)
    print(f"  {'nur Kosten minimal':<22} {guenstigst[0]:>10,} {guenstigst[1]:>9,} kg")
    print(f"  {'nur CO2 minimal':<22} {saubersten[0]:>10,} {saubersten[1]:>9,} kg")
    print(f"\n  Der Zielkonflikt ist echt, aber klein: "
          f"{saubersten[0] - guenstigst[0]:,} EUR mehr")
    print(f"  ({(saubersten[0] - guenstigst[0]) / guenstigst[0] * 100:.1f} %) sparen "
          f"{guenstigst[1] - saubersten[1]:,} kg CO2 "
          f"({(guenstigst[1] - saubersten[1]) / guenstigst[1] * 100:.1f} %).")
    print("  Genau solche Zahlen will die Geschaeftsfuehrung sehen - nicht ein")
    print("  Gewicht, das niemand interpretieren kann.")

    # --- 2. Die lineare Skalarisierung ---------------------------------
    gewichte = np.concatenate([np.linspace(0.0, 3.0, 1201),
                               np.geomspace(3.0, 1000.0, 200)])
    gefunden = skalarisierung(gewichte)
    print("\n" + "-" * 80)
    print(f"2. Lineare Skalarisierung: 'Kosten + w * CO2' fuer "
          f"{len(gewichte):,} Gewichte\n")
    print(f"  {'Kosten':>10} {'CO2':>10}   {'gefunden bei w':>16}")
    print("  " + "-" * 46)
    for (k, c), ws in sorted(gefunden.items()):
        print(f"  {k:>10,} {c:>9,} kg   {min(ws):>7.3f} bis {max(ws):>7.3f}")
    print(f"\n  {len(gefunden)} verschiedene Plaene - fuer {len(gewichte):,} Gewichte.")
    print("  Das Gewicht ist also gar keine Feineinstellung: Weite Bereiche")
    print("  liefern dasselbe Ergebnis, und dazwischen springt es.")

    # --- 3. Die vollstaendige Front ------------------------------------
    front = pareto_front()
    print("\n" + "-" * 80)
    print(f"3. Die vollstaendige Pareto-Front ueber eps-Constraint "
          f"({len(front)} Solverlaeufe)\n")
    print(f"  {'Kosten':>10} {'CO2':>10}   {'Aufpreis':>9} {'CO2-Ersparnis':>14} "
          f"{'EUR je kg':>10}")
    print("  " + "-" * 60)
    for k, c in front:
        auf = k - guenstigst[0]
        ersparnis = guenstigst[1] - c
        preis = auf / ersparnis if ersparnis else 0.0
        print(f"  {k:>10,} {c:>9,} kg   {auf:>8,} {ersparnis:>13,} "
              f"{preis:>10.2f}")

    # --- 4. Der Nachweis -----------------------------------------------
    huelle = untere_huelle(front)
    unerreichbar = [p for p in front if p not in huelle]
```

```
print("\n" + "-" * 80)
print("4. Was die Skalarisierung nicht findet\n")
erreicht = [p for p in front if p in gefunden]
print(f"  Pareto-Punkte insgesamt:              {len(front)}")
print(f"  davon von der Skalarisierung gefunden: {len(erreicht)}")
print(f"  nie gefunden:                          {len(front) - len(erreicht)}")
print()
print("  Diese Kompromisse sind fuer KEIN Gewicht erreichbar:\n")
print(f"  {'Kosten':>10} {'CO2':>10}   {'Aufpreis':>9} {'CO2-Ersparnis':>14}")
print("  " + "-" * 50)
for k, c in unerreichbar:
    print(f"  {k:>10,} {c:>9,} kg   {k - guenstigst[0]:>8,} "
          f"{guenstigst[1] - c:>13,}")

stimmt = sorted(unerreichbar) == sorted(p for p in front if p not in gefunden)
print(f"\n  Gegenprobe ueber die Geometrie: {'bestanden' if stimmt else 'ABWEICHUNG'}")
print("  Genau die Punkte, die das Gewichtsraster verfehlt, liegen strikt")
print("  oberhalb der unteren konvexen Huelle. Das ist kein Rasterproblem -")
print("  eine Gerade, die von links unten an die Wolke geschoben wird,")
print("  beruehrt immer einen Eckpunkt der Huelle und nie einen Punkt")
print("  darueber. Ein feineres Raster aendert daran nichts.")

# --- 5. Lexikografisch ------------------------------------------------
print("\n" + "-" * 80)
print("5. Lexikografisch: erst Kosten, dann CO2 im Rahmen eines Budgets\n")
print(f"  {'Kostenbudget':>14} {'Kosten':>10} {'CO2':>10} {'gegenueber Minimum':>20}")
print("  " + "-" * 58)
for aufschlag in (0.00, 0.01, 0.02, 0.05, 0.10):
    budget = guenstigst[0] * (1 + aufschlag)
    ergebnis = plane(CO2, kosten_grenze=budget)
    if ergebnis is None:
        print(f"  {aufschlag:>13.0%} unzulaessig")
        continue
    print(f"  {aufschlag:>13.0%} {ergebnis[0]:>10,} {ergebnis[1]:>9,} kg "
          f"{guenstigst[1] - ergebnis[1]:>15,} kg weniger")

print("\n  Das ist die Form, die im Betrieb am ehesten trifft: Nicht 'wie")
print("  wichtig ist CO2?', sondern 'wir geben zwei Prozent mehr aus - was")
print("  bringt das?'. Die Frage kann ein Kaufmann beantworten.")

print("\n" + "=" * 80)
print("  WAS MAN DARAUS MITNIMMT")
print("=" * 80)
print("Ein Gewicht zu setzen heisst, die Entscheidung heimlich zu treffen -")
print("und dabei einen Teil der Moeglichkeiten gar nicht erst zu sehen.")
print()
print("Die Pareto-Front ist die ehrlichere Antwort: Sie legt dem Betrieb")
print("alle sinnvollen Kompromisse vor und ueberlaesst ihm die Wahl. Die")
```

```
    print("Spalte 'EUR je kg' macht sie entscheidbar - man vergleicht sie mit")
    print("dem CO2-Preis, den das Unternehmen ohnehin ansetzt.")
    print("=" * 80)
```

Erwartete Ausgabe:

```
================================================================================
  KOSTEN GEGEN CO2 - UND WARUM GEWICHTE NICHT GENUEGEN
================================================================================
12 Sendungen, 3 Verkehrstraeger, 5 freie Bahntrassen.

1. Was jedes Ziel allein ergibt

                            Kosten        CO2
  ------------------------------------------
  nur Kosten minimal         13,543     6,475 kg
  nur CO2 minimal            14,821     5,506 kg

  Der Zielkonflikt ist echt, aber klein: 1,278 EUR mehr
  (9.4 %) sparen 969 kg CO2 (15.0 %).
  Genau solche Zahlen will die Geschaeftsfuehrung sehen - nicht ein
  Gewicht, das niemand interpretieren kann.

--------------------------------------------------------------------------------
2. Lineare Skalarisierung: 'Kosten + w * CO2' fuer 1,401 Gewichte

      Kosten        CO2     gefunden bei w
  ------------------------------------------------
      13,543     6,475 kg     0.000 bis   0.398
      13,724     6,022 kg     0.400 bis   1.002
      14,036     5,711 kg     1.005 bis   1.935
      14,189     5,632 kg     1.938 bis   3.471
      14,592     5,517 kg     3.574 bis  20.599
      14,821     5,506 kg    21.210 bis 1000.000

  6 verschiedene Plaene - fuer 1,401 Gewichte.
  Das Gewicht ist also gar keine Feineinstellung: Weite Bereiche
  liefern dasselbe Ergebnis, und dazwischen springt es.

--------------------------------------------------------------------------------
3. Die vollstaendige Pareto-Front ueber eps-Constraint (10 Solverlaeufe)

      Kosten        CO2    Aufpreis  CO2-Ersparnis  EUR je kg
  ------------------------------------------------------------
      13,543     6,475 kg          0             0       0.00
      13,635     6,317 kg         92           158       0.58
      13,718     6,286 kg        175           189       0.93
      13,724     6,022 kg        181           453       0.40
```

```
      13,868     5,927 kg        325           548       0.59
      13,978     5,912 kg        435           563       0.77
      14,036     5,711 kg        493           764       0.65
      14,189     5,632 kg        646           843       0.77
      14,592     5,517 kg      1,049           958       1.09
      14,821     5,506 kg      1,278           969       1.32

--------------------------------------------------------------------------------
4. Was die Skalarisierung nicht findet

  Pareto-Punkte insgesamt:              10
  davon von der Skalarisierung gefunden: 6
  nie gefunden:                          4

  Diese Kompromisse sind fuer KEIN Gewicht erreichbar:

      Kosten        CO2    Aufpreis  CO2-Ersparnis
  -------------------------------------------------
      13,635     6,317 kg         92           158
      13,718     6,286 kg        175           189
      13,868     5,927 kg        325           548
      13,978     5,912 kg        435           563

  Gegenprobe ueber die Geometrie: bestanden
  Genau die Punkte, die das Gewichtsraster verfehlt, liegen strikt
  oberhalb der unteren konvexen Huelle. Das ist kein Rasterproblem -
  eine Gerade, die von links unten an die Wolke geschoben wird,
  beruehrt immer einen Eckpunkt der Huelle und nie einen Punkt
  darueber. Ein feineres Raster aendert daran nichts.

--------------------------------------------------------------------------------
5. Lexikografisch: erst Kosten, dann CO2 im Rahmen eines Budgets

    Kostenbudget     Kosten        CO2   gegenueber Minimum
  ------------------------------------------------------------
              0%     13,543     6,475 kg               0 kg weniger
              1%     13,635     6,317 kg             158 kg weniger
              2%     13,724     6,022 kg             453 kg weniger
              5%     14,189     5,632 kg             843 kg weniger
             10%     14,821     5,506 kg             969 kg weniger

  Das ist die Form, die im Betrieb am ehesten trifft: Nicht 'wie
  wichtig ist CO2?', sondern 'wir geben zwei Prozent mehr aus - was
  bringt das?'. Die Frage kann ein Kaufmann beantworten.

================================================================================
  WAS MAN DARAUS MITNIMMT
================================================================================
Ein Gewicht zu setzen heisst, die Entscheidung heimlich zu treffen -
```

```
und dabei einen Teil der Moeglichkeiten gar nicht erst zu sehen.

Die Pareto-Front ist die ehrlichere Antwort: Sie legt dem Betrieb
alle sinnvollen Kompromisse vor und ueberlaesst ihm die Wahl. Die
Spalte 'EUR je kg' macht sie entscheidbar - man vergleicht sie mit
dem CO2-Preis, den das Unternehmen ohnehin ansetzt.
================================================================================
```

14.6 Was die gewichtete Summe nicht sieht

Die Front hat **zehn** Punkte. Ein Raster aus 1 401 Gewichten findet **sechs** davon. Vier Kompromisse sind für kein Gewicht erreichbar — und das ist keine Frage der Rasterweite.

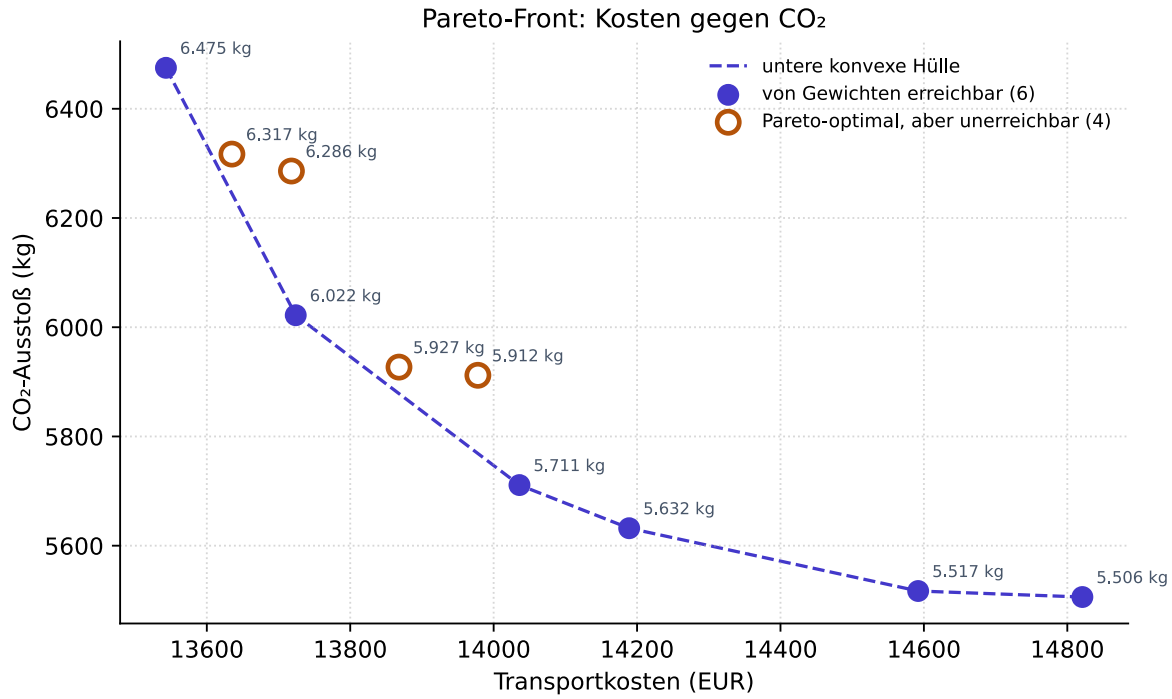

Abbildung 21: Abb. 14.1: Pareto-Front: Kosten gegen CO□. Die gefüllten Punkte liegen auf der unteren konvexen Hülle und sind über Gewichte erreichbar; die vier offenen Kreise sind pareto-optimal, werden aber von keiner Geraden gestützt.

Eine interaktive Fassung dieser Grafik steht auf den Kapitelseiten der Website bereit.

Warum es keine Frage des Rasters ist. „$c(x) + w\,e(x)$ minimieren" heißt geometrisch: eine Gerade der Steigung $-1/w$ von links unten an die Punktwolke schieben und schauen, welchen Punkt sie zuerst berührt. Eine Gerade berührt immer einen **Eckpunkt der unteren konvexen Hülle**. Punkte, die oberhalb der Hülle liegen, werden von keiner Geraden zuerst getroffen — für kein w, bei beliebig feiner Abstufung.

Das Programm rechnet diese Hülle in `untere_huelle()` unabhängig aus und vergleicht: Genau die vier Punkte, die das Gewichtsraster verfehlt, liegen strikt oberhalb. Die Gegenprobe steht in der Ausgabe.

□ **Der Grund liegt in der Ganzzahligkeit**

Bei einem reinen LP ist der zulässige Bereich konvex, und die Pareto-Front liegt vollständig auf ihrer eigenen konvexen Hülle — dort findet die gewichtete Summe alles. Sobald Entscheidungen **ganzzahlig** werden (welcher Träger, welches Lager, welche Schicht), zerfällt der Bereich in einzelne Punkte, und zwischen ihnen entstehen Einbuchtungen.

Deshalb betrifft dieses Problem praktisch jedes betriebliche Mehrzielmodell. Wer mit Gewichten arbeitet, verliert nicht ein paar Nachkommastellen, sondern **ganze Alternativen** — im Beispiel 40 % der Front.

Das Gewicht ist keine Feineinstellung

Die zweite Beobachtung aus derselben Tabelle: 1 401 Gewichte erzeugen **sechs** verschiedene Pläne. Weite Gewichtsbereiche liefern dasselbe Ergebnis, und dazwischen springt es.

Das macht die verbreitete Vorgehensweise — „wir probieren ein paar Gewichte und schauen, was herauskommt" — zu einem Glücksspiel. Man weiß nie, ob man einen neuen Kompromiss gefunden hat oder nur denselben noch einmal.

14.7 Das ε-Constraint-Verfahren

Der Ausweg ist unspektakulär: **Man macht aus dem zweiten Ziel eine Nebenbedingung.**

$$\min_{x \in X} \; c(x) \qquad \text{unter} \qquad e(x) \le \varepsilon$$

Für jedes ε liefert das einen Punkt der Front. Die Kunst besteht nur darin, die ε geschickt zu wählen — und dafür gibt es ein Verfahren, das genau so viele Solverläufe braucht, wie die Front Punkte hat:

1. Kostenminimum bestimmen. Das ist der eine Rand der Front; nebenbei fällt sein CO□-Wert an.
2. ε auf **ein Kilogramm unter diesen Wert** setzen und erneut lösen.
3. Schritt 2 wiederholen, bis kein zulässiger Plan mehr existiert.

Im Beispiel sind das zehn Läufe. Ein Raster über alle möglichen CO□-Werte bräuchte fast tausend, ein Gewichtsraster findet auch mit 1 401 Läufen nur sechs Punkte.

□ **Merksatz** ε-Constraint ist das Arbeitspferd der Mehrzieloptimierung: vollständig, mit minimal vielen Solverläufen, und ohne dass jemand ein Gewicht erfinden muss.

Wenn es eine klare Rangfolge gibt: lexikografisch

Manchmal ist ein Ziel wirklich vorrangig. Dann optimiert man **lexikografisch**: erst das erste Ziel, dann das zweite unter der Bedingung, dass das erste (fast) erhalten bleibt.

Kostenbudget	Kosten	CO□	gegenüber dem Minimum
+0 %	13 543 €	6 475 kg	—
+1 %	13 635 €	6 317 kg	158 kg weniger
+2 %	13 724 €	6 022 kg	453 kg weniger
+5 %	14 189 €	5 632 kg	843 kg weniger

Kostenbudget	Kosten	CO□	gegenüber dem Minimum
+10 %	14 821 €	5 506 kg	969 kg weniger

Der Sprung von +1 % auf +2 % ist der interessante: Das zweite Prozent bringt fast dreimal so viel wie das erste. Solche Unregelmäßigkeiten sind bei ganzzahligen Modellen die Regel — und ein weiteres Argument gegen Gewichte, die eine gleichmäßige Abwägung suggerieren.

Beachten Sie außerdem: Die Zeile „+0 %" ist ein Sonderfall, der sich auszahlt. Sie sucht unter allen **kostenminimalen** Plänen den saubersten. Gibt es mehrere optimale Lösungen — und das ist bei ganzzahligen Modellen fast immer so —, bekommt man die CO□-Ersparnis geschenkt. Ein Solver, der nur ein Ziel kennt, gibt einfach die erstbeste zurück.

14.8 Wie man die Front vorlegt

Eine Front mit zehn Punkten ist kein Ergebnis, sondern eine Speisekarte. Damit sie entscheidbar wird, gehört eine Spalte dazu, die im Programm `EUR je kg` heißt:

$$\text{Schattenpreis} = \frac{\text{Aufpreis gegenüber dem Kostenminimum}}{\text{eingesparte Kilogramm}}$$

Diese Zahl ist mit etwas vergleichbar, das das Unternehmen ohnehin kennt: dem internen CO□-Preis oder dem Zertifikatspreis. Liegt der Schattenpreis darunter, ist die Entscheidung kaufmännisch schon getroffen.

□ Code-Durchgang: die drei Bausteine

Stelle	Was sie tut
`plane(ziel, co2_grenze, kosten_grenze)`	**eine** Funktion für alle Varianten. Welches Ziel und welche Schranke — mehr Unterschied gibt es zwischen den fünf Teilen des Programms nicht
`pareto_front()`	das ε-Verfahren: Grenze jeweils um 1 kg unter den zuletzt erreichten Wert drücken, bis nichts mehr geht
`untere_huelle()`	rechnet die Hülle **unabhängig** aus, damit die Aussage „unerreichbar" nicht auf dem Gewichtsraster beruht, sondern auf Geometrie

> Der Vergleich am Ende von Teil 4 ist bewusst ein `assert`-artiger Abgleich: Wenn Gewichtsraster und Hüllenrechnung verschiedene Punktmengen liefern, stimmt eines von beidem nicht — und das steht dann in der Ausgabe.

14.9 Übungsaufgaben

Lösungen: Abschnitt A.14.

Aufgabe 14.1 □ — Dominanz prüfen. Plan P kostet 14 000 € und stößt 6 000 kg aus, Plan Q kostet 13 800 € und stößt 6 100 kg aus. Dominiert einer den anderen? Und wie steht es mit R (14 000 €, 6 100 kg)?

Aufgabe 14.2 □ — Der Schattenpreis in der Praxis. Der interne CO□-Preis des Unternehmens beträgt 0,90 € je Kilogramm. Welchen Punkt der Front würden Sie empfehlen — und wie begründen Sie ihn in einem Satz?

Aufgabe 14.3 □□ — Mehr Trassen. Erhöhen Sie `BAHN_TRASSEN` von 5 auf 8. Was passiert mit der Länge der Front und mit der Zahl der unerreichbaren Punkte? Erklären Sie den Zusammenhang.

Aufgabe 14.4 □□ — Die Front der Relaxation. Lassen Sie `integrality` weg, sodass Anteile erlaubt sind. Berechnen Sie die Front erneut (mit einem ε-Raster, da es nun unendlich viele Punkte gibt). Wie viele Punkte liegen jetzt oberhalb der konvexen Hülle — und warum?

Aufgabe 14.5 □□□ — Drei Ziele. Ergänzen Sie die Laufzeit als drittes Ziel (LKW schnell, Bahn langsam). Die Front wird zu einer Fläche. Wie ändert sich das ε-Verfahren, und warum wächst der Aufwand so schnell?

14.10 Finde den Denkfehler

□ **„Wir haben die Gewichte sauber kalibriert"**

Ein Team baut ein Mehrzielmodell für die Tourenplanung: Kosten gegen CO□. Um das Gewicht nicht willkürlich zu setzen, geht es methodisch vor:

> *„Wir haben w in 500 Schritten von 0 bis 5 durchgerechnet und für jeden Wert den > optimalen Plan bestimmt. Dann haben wir der Geschäftsführung alle so gefundenen Pläne > vorgelegt und sie hat einen ausgewählt. Damit ist die Entscheidung nicht von uns > getroffen worden, sondern vom Fachbereich — und wir haben den Lösungsraum vollständig > abgetastet."*

Das Vorgehen ist sorgfältig, die Rechnung stimmt, und die Beteiligung des Fachbereichs ist vorbildlich.

Trotzdem stimmt die letzte Behauptung nicht. Warum — und was hätte das Team stattdessen tun müssen?

Und ein zweiter Teil, der schwerer wiegt: Angenommen, unter den nicht gefundenen Plänen wäre einer gewesen, der die Geschäftsführung überzeugt hätte. **Hätte irgendjemand das bemerkt?**

14.11 Micro-Quiz

□ Drei Fragen

1. Warum findet die gewichtete Summe nicht alle Pareto-Punkte? a) Weil das Gewichtsraster zwangsläufig zu grob ist. b) Weil sie nur Punkte auf der unteren konvexen Hülle erreichen kann — Punkte darüber werden von keiner Geraden gestützt. c) Weil Solver bei gemischten Zielfunktionen numerisch ungenau werden.

2. Wie viele Solverläufe braucht das ε-Constraint-Verfahren in der gezeigten Form? a) So viele, wie das ε-Raster Schritte hat. b) So viele, wie die Pareto-Front Punkte hat, plus einen. c) Genau zwei — einen je Ziel.

3. Die Zeile „+0 % Kostenbudget" in der lexikografischen Tabelle liefert weniger CO□ als das reine Kostenminimum. Wie kann das sein? a) Ein Rundungsfehler im Solver. b) Es gibt mehrere kostenminimale Pläne; unter ihnen wird der sauberste gewählt. c) Das Kostenbudget ist durch die Prozentrechnung minimal größer.

14.12 Selbsttest

1. Erklären Sie Pareto-Dominanz an einem Beispiel aus Ihrem Arbeitsumfeld.
2. Warum ist „gewichtete Summe" bei einem reinen LP unproblematisch und bei einem MILP nicht?
3. Ein Kollege sagt: „Ich nehme $w = 0{,}5$, das ist neutral." Was ist daran falsch?
4. Beschreiben Sie das ε-Constraint-Verfahren in drei Sätzen, ohne Formeln.
5. Sie legen einer Geschäftsführung eine Front mit zwölf Punkten vor. Welche Spalte brauchen Sie, damit die Entscheidung fallen kann?

14.13 Zusammenfassung

- Bei mehreren Zielen ist „besser" nur noch über **Dominanz** definiert. Das vollständige Ergebnis ist keine Lösung, sondern die **Pareto-Front**.
- Die **gewichtete Summe** findet nur Punkte auf der unteren konvexen Hülle. Bei ganzzahligen Modellen liegen regelmäßig Pareto-Punkte darüber — im Beispiel 4 von 10, also 40 % der Alternativen. Das ist kein Rasterproblem, sondern Geometrie.
- Das Gewicht ist außerdem **keine Feineinstellung**: 1 401 Gewichte erzeugten sechs verschiedene Pläne, mit Sprüngen dazwischen.
- Das **ε-Constraint-Verfahren** macht aus dem zweiten Ziel eine Schranke und liefert die vollständige Front mit so vielen Solverläufen, wie sie Punkte hat.
- **Lexikografisch** optimiert man, wenn es eine echte Rangfolge gibt — „höchstens 2 % teurer, dafür so sauber wie möglich" ist eine Frage, die ein Kaufmann beantworten kann.
- Die Front wird erst entscheidbar durch die Spalte **€ je kg**: Sie ist mit dem internen CO□-Preis vergleichbar und verwandelt eine Wertefrage in eine Rechnung.

Kapitel 15: Predict-then-Optimize — die bessere Prognose, die schlechtere Entscheidung

Kapitel auf einen Blick

Worum geht es? Um die Naht zwischen zwei Welten, die in getrennten Abteilungen sitzen: Ein Modell **prognostiziert**, ein anderes **entscheidet**. Beide arbeiten sauber — und genau an der Naht entsteht ein Fehler, den keine der beiden Seiten sieht.

Voraussetzungen: Kapitel 12, insbesondere das Newsvendor-Problem und das kritische Verhältnis. Etwas Regression hilft, ist aber nicht nötig.

Danach können Sie: begründen, warum ein Prognosemodell mit kleinerem MSE teurere Entscheidungen erzeugen kann; die richtige **Zielgröße** einer Prognose bestimmen; und Prognosemodelle an Entscheidungskosten statt an Fehlermaßen bewerten.

Zeitbedarf: ca. 3,5 Stunden.

Programme:
`Predict_then_Optimize.py`

Notebook: `Notebooks_04/prognose.ipynb`

15.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Der Durchschnitt ist die falsche Zahl

Eine Bäckerei kennt die Nachfrage der letzten 20 Tage. Ein Brot bringt 6 € Marge, ein übriges kostet 3 € Einkauf. Wie viele soll sie ansetzen?

Die naheliegende Antwort — den Durchschnitt — ist nachweislich falsch:

```
import numpy as np

nachfrage = np.array([104, 138, 96, 151, 118, 127, 143, 109, 162, 121,   # 20 Tage
                      133, 115, 148, 102, 129, 156, 111, 140, 124, 135])
preis, einkauf = 9.0, 3.0
fehl, ueber = preis - einkauf, einkauf          # 6 EUR zu wenig, 3 EUR zu viel

kosten = lambda menge: (fehl * np.maximum(0, nachfrage - menge)
                        + ueber * np.maximum(0, menge - nachfrage)).mean()

erwartungswert = round(nachfrage.mean())
kritisch = round(np.quantile(nachfrage, fehl / (fehl + ueber)))
print(f"Erwartungswert bestellen: {erwartungswert} Stueck ->
 ↪  {kosten(erwartungswert):5.2f} EUR/Tag")
print(f"kritisches Quantil      : {kritisch} Stueck -> {kosten(kritisch):5.2f}
 ↪  EUR/Tag")
```

```
print(f"bestmoegliche feste Menge: {int(np.argmin([kosten(m) for m in
 ↪  range(200)]))} Stueck"
      f" -> {min(kosten(m) for m in range(200)):5.2f} EUR/Tag")
```

Ausgabe:

```
Erwartungswert bestellen: 128 Stueck -> 69.45 EUR/Tag
kritisches Quantil      : 137 Stueck -> 62.25 EUR/Tag
bestmoegliche feste Menge: 138 Stueck -> 62.10 EUR/Tag
```

Und jetzt der Punkt. Der Durchschnitt ist perfekt bestimmt — es ist der Mittelwert derselben Daten, aus denen auch das Quantil kommt. Trotzdem kostet er **7,20 € je Tag mehr**, gut 10 %.

Der Grund steht in den Preisen: Ein fehlendes Brot kostet doppelt so viel wie ein übriges. Bei asymmetrischen Kosten liegt die optimale Menge nicht in der Mitte der Verteilung, sondern beim **kritischen Verhältnis** $c_-/(c_- + c_+) = 6/9 = 66{,}7\,\%$ — das kennen Sie aus Kapitel 12.

Bemerkenswert ist die dritte Zeile: Die Quantilregel trifft mit 137 fast genau die beste überhaupt mögliche feste Menge (138). Sie ist nicht ungefähr richtig, sie ist richtig.

> □ **Merksatz** Eine Prognose ist kein Selbstzweck. Sie liefert eine Zahl, die in eine Entscheidung eingeht — und **welche** Zahl das sein muss, bestimmt die Entscheidung, nicht die Statistik.

Warum funktioniert das? Weil die Kostenfunktion des Newsvendors geknickt und unsymmetrisch ist. Der Mittelwert minimiert den *quadratischen* Fehler; die Kosten sind aber stückweise linear mit verschiedenen Steigungen nach oben und unten. Zwei verschiedene Zielfunktionen haben zwei verschiedene Optima — das ist kein Paradox, sondern Arithmetik.

15.2 Lernziele

Nach diesem Kapitel können Sie …

1. … die Zweiteilung *predict* / *optimize* benennen und sagen, wo dabei Information verloren geht.
2. … begründen, warum das MSE-beste Modell nicht das kostenbeste sein muss.
3. … die richtige Prognosegröße aus der Entscheidung ableiten statt aus Gewohnheit.
4. … erklären, warum ein pauschaler Sicherheitszuschlag schwächer ist als ein Modell, das die Unsicherheit aus den Merkmalen liest.
5. … Prognosemodelle an **Entscheidungskosten** messen — und wissen, wie viele Testdaten ein solcher Vergleich braucht.

15.3 Die Naht zwischen zwei Abteilungen

In fast jedem Unternehmen sieht der Ablauf so aus:

```
Verkaufsdaten  ->  [ PROGNOSE ]  ->  Nachfrageschätzung  ->  [ PLANUNG ]  ->  Bestellung
                     Data Science                              Disposition
```

Beide Seiten arbeiten sorgfältig. Die Prognoseabteilung optimiert ihr Modell auf ein **Fehlermaß** — meist den mittleren quadratischen Fehler (MSE) oder MAPE — und berichtet stolz eine Verbesserung von 12 %. Die Disposition nimmt die Zahl entgegen und rechnet ihre Bestellmenge aus.

Der Fehler steckt nicht in einer der beiden Hälften, sondern in der Naht:

> **Definition: Predict-then-Optimize**
>
> Das übliche zweistufige Vorgehen: erst eine unbekannte Größe schätzen, dann mit der Schätzung optimieren, als wäre sie die Wahrheit.
>
> Das Problem: Die erste Stufe wird auf ein **statistisches** Maß trainiert, die zweite erzeugt **ökonomische** Kosten. Niemand garantiert, dass ein besseres statistisches Maß zu geringeren Kosten führt — und in diesem Kapitel ist es nachweislich umgekehrt.

> **Warum das nicht auffällt**
>
> Beide Abteilungen erfüllen ihre Kennzahl. Die Prognose wird besser (MSE sinkt), die Disposition arbeitet korrekt (sie wendet die Formel richtig an). Es gibt keinen Ort, an dem der Verlust sichtbar würde — außer man misst die Entscheidungskosten, und das ist niemandes Kennzahl.
>
> Das ist dieselbe Struktur wie in Abschnitt 23.9: Der Fehler entsteht zwischen zwei Zuständigkeiten und wird deshalb von keiner Prüfung gefunden, die innerhalb einer der beiden liegt.

15.4 Das Programm

Dieselbe Bäckerei, aber mit Merkmalen: Wochentag, Temperatur, Aktionstage. Die Nachfrage muss jetzt **modelliert** werden.

Eine Eigenschaft der Daten ist dabei entscheidend und in der Praxis der Normalfall: An Aktionstagen ist die Nachfrage nicht nur höher, sondern auch **viel unsicherer** (heteroskedastisch).

```
#!/usr/bin/env python3

# Predict_then_Optimize.py
"""
Kapitel Prognose: Die bessere Prognose trifft die schlechtere Entscheidung.

Eine Baeckerei muss jeden Abend entscheiden, wie viel sie fuer den naechsten Tag
ansetzt. Zu wenig kostet die Marge des entgangenen Verkaufs, zu viel kostet den
Einkaufspreis der Retoure. Das ist das Newsvendor-Problem aus dem Kapitel
Unsicherheit - nur dass die Nachfrage diesmal nicht aus einer Verteilung kommt,
```

```
sondern PROGNOSTIZIERT werden muss: aus Wochentag, Temperatur und Aktionstagen.

Damit zerfaellt die Aufgabe in zwei Schritte, und genau an der Naht entsteht der
Fehler, um den es hier geht:

    PREDICT     ein Modell schaetzt die Nachfrage
    OPTIMIZE    daraus wird eine Bestellmenge

Der Prognostiker optimiert seinen Modellfehler, meist den MSE. Der Planer traegt
die Kosten. Beide messen etwas anderes - und die beiden Masse widersprechen
einander. Das Programm zeigt:

  1. Vier Verfahren, verglichen nach MSE UND nach Entscheidungskosten. Das
     Verfahren mit dem BESTEN MSE hat die HOECHSTEN Kosten.
  2. Warum ein pauschaler Sicherheitszuschlag zu kurz greift - die Streuung der
     Nachfrage haengt selbst von den Merkmalen ab.
  3. Eine Messfalle, in die der Autor dieses Programms zuerst selbst getappt
     ist: Bei kurzen Testzeitraeumen ist der MSE-Vergleich nicht stabil.

Benoetigt: numpy, scipy, scikit-learn
"""

from __future__ import annotations

import numpy as np
from scipy.stats import norm
from sklearn.linear_model import LinearRegression, QuantileRegressor

VERKAUFSPREIS = 9.0
EINKAUFSPREIS = 3.0
KOSTEN_FEHLMENGE = VERKAUFSPREIS - EINKAUFSPREIS      # entgangene Marge: 6 EUR
KOSTEN_UEBERHANG = EINKAUFSPREIS                      # Retoure:          3 EUR
KRITISCHES_VERHAELTNIS = KOSTEN_FEHLMENGE / (KOSTEN_FEHLMENGE + KOSTEN_UEBERHANG)

TAGE = 5000               # Simulation, siehe Hinweis unten
TRAINING = 1000
SAAT = 11


def erzeuge_daten(tage: int = TAGE, saat: int = SAAT):
    """Taegliche Nachfrage mit Wochentag, Temperatur und Aktionstagen.

    Die entscheidende Eigenschaft steckt in 'streuung': An Aktionstagen ist die
    Nachfrage nicht nur hoeher, sondern auch viel UNSICHERER. Solche
    heteroskedastischen Daten sind der Normalfall - und der Grund, warum ein
    pauschaler Sicherheitszuschlag nicht genuegt.
    """
    rng = np.random.default_rng(saat)
```

```
    wochentag = np.arange(tage) % 7
    temperatur = (12 + 10 * np.sin(2 * np.pi * np.arange(tage) / 365)
                  + rng.normal(0, 3, tage))
    aktion = (rng.random(tage) < 0.15).astype(float)

    merkmale = np.column_stack([np.eye(7)[wochentag][:, 1:], temperatur, aktion])
    erwartung = (120
                 + np.eye(7)[wochentag] @ np.array([0, 10, 12, 14, 18, 35, -40])
                 + 1.8 * temperatur + 45 * aktion)
    streuung = 8 + 22 * aktion
    nachfrage = np.maximum(0.0, erwartung + rng.normal(0, 1, tage) * streuung)
    return merkmale, nachfrage, aktion


def tageskosten(bestellung: np.ndarray, nachfrage: np.ndarray) -> float:
    """Die Zahl, auf die es ankommt - und die kein Prognosemass kennt."""
    fehlmenge = np.maximum(0.0, nachfrage - bestellung)
    ueberhang = np.maximum(0.0, bestellung - nachfrage)
    return float((KOSTEN_FEHLMENGE * fehlmenge
                  + KOSTEN_UEBERHANG * ueberhang).mean())


if __name__ == "__main__":
    merkmale, nachfrage, aktion = erzeuge_daten()
    lernen = slice(0, TRAINING)
    pruefen = slice(TRAINING, TAGE)

    print("=" * 84)
    print("  DIE BESSERE PROGNOSE TRIFFT DIE SCHLECHTERE ENTSCHEIDUNG")
    print("=" * 84)
    print(f"Verkaufspreis {VERKAUFSPREIS:.0f} EUR, Einkauf {EINKAUFSPREIS:.0f} EUR.")
    print(f"Fehlmenge kostet {KOSTEN_FEHLMENGE:.0f} EUR, Ueberhang "
          f"{KOSTEN_UEBERHANG:.0f} EUR je Stueck.")
    print(f"Kritisches Verhaeltnis: {KRITISCHES_VERHAELTNIS:.3f} - der Planer sollte "
          f"also das")
    print(f"{KRITISCHES_VERHAELTNIS:.1%}-Quantil der Nachfrage bestellen, nicht ihren "
          f"Erwartungswert.")
    print(f"\nTraining: Tag 1 bis {TRAINING}. Bewertung: die restlichen "
          f"{TAGE - TRAINING} Tage.\n")

    # --- Die vier Verfahren ------------------------------------------------
    kleinste_quadrate = LinearRegression().fit(merkmale[lernen], nachfrage[lernen])
    punktprognose = kleinste_quadrate.predict(merkmale[pruefen])

    restfehler = nachfrage[lernen] - kleinste_quadrate.predict(merkmale[lernen])
    pauschalzuschlag = norm.ppf(KRITISCHES_VERHAELTNIS) * restfehler.std()

    # Ein Zuschlag, der nicht aus der Normalverteilung kommt, sondern direkt
```

```
    # auf den Trainingsdaten die Kosten minimiert.
    kandidaten = np.linspace(-10.0, 30.0, 401)
    trainingsprognose = kleinste_quadrate.predict(merkmale[lernen])
    kostenzuschlag = float(kandidaten[np.argmin(
        [tageskosten(trainingsprognose + z, nachfrage[lernen]) for z in kandidaten])])

    # Und das Verfahren, das von vornherein das richtige Quantil schaetzt.
    quantilmodell = QuantileRegressor(quantile=KRITISCHES_VERHAELTNIS,
                                      alpha=0.0, solver="highs")
    quantilmodell.fit(merkmale[lernen], nachfrage[lernen])
    quantilprognose = quantilmodell.predict(merkmale[pruefen])

    verfahren = [
        ("bestelle die Punktprognose", punktprognose, punktprognose),
        ("+ Zuschlag aus der Normalverteilung",
         punktprognose, punktprognose + pauschalzuschlag),
        ("+ Zuschlag auf Kosten trainiert",
         punktprognose, punktprognose + kostenzuschlag),
        ("Quantilregression aufs kritische Quantil",
         quantilprognose, quantilprognose),
    ]

    print(f"  {'Verfahren':<42} {'MSE':>9} {'Kosten/Tag':>12} {'gegen Zeile 1':>14}")
    print("  " + "-" * 80)
    ergebnisse = {}
    for name, prognose, bestellung in verfahren:
        mse = float(((prognose - nachfrage[pruefen]) ** 2).mean())
        kosten = tageskosten(bestellung, nachfrage[pruefen])
        ergebnisse[name] = (mse, kosten)
        basis = ergebnisse[verfahren[0][0]][1]
        vergleich = "" if name == verfahren[0][0] else f"{(kosten - basis) / basis:+13.1%}"
        print(f"  {name:<42} {mse:>9.1f} {kosten:>10.2f} EUR {vergleich:>14}")

    bester_mse = min(ergebnisse, key=lambda k: ergebnisse[k][0])
    beste_kosten = min(ergebnisse, key=lambda k: ergebnisse[k][1])
    print(f"\n  bester MSE:      {bester_mse}")
    print(f"  beste Kosten:    {beste_kosten}")
    print(f"\n  Das Verfahren mit dem besten MSE hat die HOECHSTEN Kosten, und das")
    print(f"  Verfahren mit den besten Kosten hat einen um "
          f"{(ergebnisse[beste_kosten][0] / ergebnisse[bester_mse][0] - 1):.0%} SCHLECHTEREN
           ↪ MSE.")
    print("  Wer Prognosemodelle nach MSE auswaehlt, waehlt hier das falsche.")

    # --- Warum der pauschale Zuschlag zu kurz greift ---------------------
    print("\n" + "-" * 84)
    print("Warum ein pauschaler Zuschlag nicht genuegt\n")
    ist_aktion = aktion[pruefen] > 0.5
    print(f"  {'Verfahren':<42} {'normale Tage':>14} {'Aktionstage':>14}")
```

```
    print("  " + "-" * 74)
    for name, _, bestellung in verfahren:
        normal = tageskosten(bestellung[~ist_aktion], nachfrage[pruefen][~ist_aktion])
        aktionstag = tageskosten(bestellung[ist_aktion], nachfrage[pruefen][ist_aktion])
        print(f"  {name:<42} {normal:>10.2f} EUR {aktionstag:>10.2f} EUR")

    # Nachgerechnet statt behauptet: Welcher Zuschlag waere je Tagesart richtig?
    aktion_training = aktion[lernen] > 0.5
    z = norm.ppf(KRITISCHES_VERHAELTNIS)
    richtig_normal = z * restfehler[~aktion_training].std()
    richtig_aktion = z * restfehler[aktion_training].std()
    print(f"\n  Der pauschale Zuschlag betraegt {pauschalzuschlag:.1f} Stueck. Aus den")
    print(f"  Trainingsresten getrennt nach Tagesart waere richtig:")
    print(f"    normale Tage : {richtig_normal:5.1f} Stueck")
    print(f"    Aktionstage  : {richtig_aktion:5.1f} Stueck")
    print(f"  Ein Zuschlag fuer alle Tage kann nur einen Mittelweg treffen - hier")
    print(f"  ist er an normalen Tagen {pauschalzuschlag / richtig_normal:.1f}-mal zu gross
     ↪  und an")
    print(f"  Aktionstagen nur {pauschalzuschlag / richtig_aktion:.0%} dessen, was noetig
     ↪  waere.")
    print(f"\n  Die Quantilregression schaetzt das {KRITISCHES_VERHAELTNIS:.1%}-Quantil "
          f"direkt aus den")
    print(f"  Merkmalen und darf deshalb an verschiedenen Tagen verschieden weit")
    print(f"  ueber dem Erwartungswert liegen. Genau das ist der Unterschied")
    print(f"  zwischen 'ein Modell und danach eine Formel' und 'ein Modell, das")
    print(f"  weiss, wofuer es gebraucht wird'.")

    # --- Die Messfalle -----------------------------------------------------
    print("\n" + "-" * 84)
    print("Eine Messfalle, in die der Autor zuerst selbst getappt ist\n")
    print("  Der erste Entwurf dieses Programms bewertete auf 230 Testtagen - ein")
    print("  realistischer Zeitraum. Dort hatte die Quantilregression den BESSEREN")
    print("  MSE, und die ganze Aussage des Kapitels stand auf dem Kopf.")
    print("\n  Wie oft das passiert, laesst sich ausmessen:\n")
    rng = np.random.default_rng(0)
    print(f"  {'Testfenster':>14} {'QR sieht MSE-besser aus':>26}")
    print("  " + "-" * 42)
    for fenster in (180, 365, 730, 2000):
        treffer = 0
        versuche = 400
        for _ in range(versuche):
            start = int(rng.integers(TRAINING, TAGE - fenster))
            ausschnitt = slice(start, start + fenster)
            mse_punkt = ((kleinste_quadrate.predict(merkmale[ausschnitt])
                          - nachfrage[ausschnitt]) ** 2).mean()
            mse_quantil = ((quantilmodell.predict(merkmale[ausschnitt])
                            - nachfrage[ausschnitt]) ** 2).mean()
            treffer += mse_quantil < mse_punkt
```

```
        print(f"  {fenster:>10} Tage {treffer / versuche:>24.1%}")

    print("\n  Bei einem halben Jahr Testdaten sieht das schlechtere Modell in gut")
    print("  jedem zehnten Fall besser aus. Das ist keine grosse Zahl - aber wer")
    print("  EINMAL misst, hat genau eine Ziehung aus dieser Verteilung.")
    print("\n  Die Lehre ist nicht 'nimm 4.000 Testtage' - die hat niemand. Sie")
    print("  lautet: Ein Kennzahlenvergleich ohne Angabe seiner Streuung ist keine")
    print("  Aussage. Bei kurzen Zeitraeumen gehoert eine Kreuzvalidierung dazu.")

    print("\n" + "=" * 84)
    print("  WAS MAN DARAUS MITNIMMT")
    print("=" * 84)
    print("Der Prognostiker optimiert den MSE, der Planer traegt die Kosten - und")
    print("die beiden Masse zeigen hier in verschiedene Richtungen. Drei Saetze:")
    print()
    print("  1. Sagen Sie nicht den Erwartungswert vorher, sondern die Groesse, die")
    print("     in die Entscheidung eingeht. Beim Newsvendor ist das das kritische")
    print("     Quantil - und das kann man direkt schaetzen.")
    print("  2. Bewerten Sie Prognosemodelle an den ENTSCHEIDUNGSKOSTEN. Die sind")
    print("     in Euro und damit vergleichbar; ein MSE ist es nicht.")
    print("  3. Ein pauschaler Sicherheitszuschlag ist besser als nichts und")
    print("     schlechter als ein Modell, das die Unsicherheit selbst aus den")
    print("     Merkmalen liest.")
    print()
    print("Der naechste Schritt - Prognosemodelle so zu trainieren, dass sie die")
    print("Entscheidungskosten direkt minimieren (Smart Predict-then-Optimize,")
    print("differenzierbare Optimierungsschichten) - ist Forschungsstand und")
    print("erfordert Bibliotheken wie cvxpylayers. Die dritte Zeile der Tabelle")
    print("oben ist seine einfachste denkbare Form: ein einziger Parameter, auf")
    print("Kosten statt auf Fehler trainiert.")
    print("=" * 84)
```

Erwartete Ausgabe:

```
====================================================================================
  DIE BESSERE PROGNOSE TRIFFT DIE SCHLECHTERE ENTSCHEIDUNG
====================================================================================
Verkaufspreis 9 EUR, Einkauf 3 EUR.
Fehlmenge kostet 6 EUR, Ueberhang 3 EUR je Stueck.
Kritisches Verhaeltnis: 0.667 - der Planer sollte also das
66.7%-Quantil der Nachfrage bestellen, nicht ihren Erwartungswert.

Training: Tag 1 bis 1000. Bewertung: die restlichen 4000 Tage.

  Verfahren                                    MSE   Kosten/Tag  gegen Zeile 1
  ------------------------------------------------------------------------------
  bestelle die Punktprognose                 201.0      42.63 EUR
  + Zuschlag aus der Normalverteilung        201.0      39.51 EUR          -7.3%
```

```
  + Zuschlag auf Kosten trainiert                201.0      39.14 EUR          -8.2%
  Quantilregression aufs kritische Quantil       227.6      38.08 EUR         -10.7%

  bester MSE:      bestelle die Punktprognose
  beste Kosten:    Quantilregression aufs kritische Quantil

  Das Verfahren mit dem besten MSE hat die HOECHSTEN Kosten, und das
  Verfahren mit den besten Kosten hat einen um 13% SCHLECHTEREN MSE.
  Wer Prognosemodelle nach MSE auswaehlt, waehlt hier das falsche.

--------------------------------------------------------------------------------
Warum ein pauschaler Zuschlag nicht genuegt

  Verfahren                                    normale Tage    Aktionstage
  ------------------------------------------------------------------------
  bestelle die Punktprognose                     29.67 EUR     108.59 EUR
  + Zuschlag aus der Normalverteilung            27.53 EUR     100.49 EUR
  + Zuschlag auf Kosten trainiert                26.67 EUR     102.58 EUR
  Quantilregression aufs kritische Quantil       26.49 EUR      97.07 EUR

  Der pauschale Zuschlag betraegt 5.9 Stueck. Aus den
  Trainingsresten getrennt nach Tagesart waere richtig:
    normale Tage :   3.5 Stueck
    Aktionstage  :  12.3 Stueck
  Ein Zuschlag fuer alle Tage kann nur einen Mittelweg treffen - hier
  ist er an normalen Tagen 1.7-mal zu gross und an
  Aktionstagen nur 48% dessen, was noetig waere.

  Die Quantilregression schaetzt das 66.7%-Quantil direkt aus den
  Merkmalen und darf deshalb an verschiedenen Tagen verschieden weit
  ueber dem Erwartungswert liegen. Genau das ist der Unterschied
  zwischen 'ein Modell und danach eine Formel' und 'ein Modell, das
  weiss, wofuer es gebraucht wird'.

--------------------------------------------------------------------------------
Eine Messfalle, in die der Autor zuerst selbst getappt ist

  Der erste Entwurf dieses Programms bewertete auf 230 Testtagen - ein
  realistischer Zeitraum. Dort hatte die Quantilregression den BESSEREN
  MSE, und die ganze Aussage des Kapitels stand auf dem Kopf.

  Wie oft das passiert, laesst sich ausmessen:

     Testfenster    QR sieht MSE-besser aus
  ------------------------------------------
         180 Tage                    11.8%
         365 Tage                     5.5%
         730 Tage                     0.0%
        2000 Tage                     0.0%
```

```
  Bei einem halben Jahr Testdaten sieht das schlechtere Modell in gut
  jedem zehnten Fall besser aus. Das ist keine grosse Zahl - aber wer
  EINMAL misst, hat genau eine Ziehung aus dieser Verteilung.

  Die Lehre ist nicht 'nimm 4.000 Testtage' - die hat niemand. Sie
  lautet: Ein Kennzahlenvergleich ohne Angabe seiner Streuung ist keine
  Aussage. Bei kurzen Zeitraeumen gehoert eine Kreuzvalidierung dazu.

================================================================================
  WAS MAN DARAUS MITNIMMT
================================================================================
Der Prognostiker optimiert den MSE, der Planer traegt die Kosten - und
die beiden Masse zeigen hier in verschiedene Richtungen. Drei Saetze:

  1. Sagen Sie nicht den Erwartungswert vorher, sondern die Groesse, die
     in die Entscheidung eingeht. Beim Newsvendor ist das das kritische
     Quantil - und das kann man direkt schaetzen.
  2. Bewerten Sie Prognosemodelle an den ENTSCHEIDUNGSKOSTEN. Die sind
     in Euro und damit vergleichbar; ein MSE ist es nicht.
  3. Ein pauschaler Sicherheitszuschlag ist besser als nichts und
     schlechter als ein Modell, das die Unsicherheit selbst aus den
     Merkmalen liest.

Der naechste Schritt - Prognosemodelle so zu trainieren, dass sie die
Entscheidungskosten direkt minimieren (Smart Predict-then-Optimize,
differenzierbare Optimierungsschichten) - ist Forschungsstand und
erfordert Bibliotheken wie cvxpylayers. Die dritte Zeile der Tabelle
oben ist seine einfachste denkbare Form: ein einziger Parameter, auf
Kosten statt auf Fehler trainiert.
================================================================================
```

15.5 Der Befund

Verfahren	MSE	Kosten je Tag
bestelle die Punktprognose	**201,0** (bester)	42,63 € (schlechteste)
+ Zuschlag aus der Normalverteilung	201,0	39,51 €
+ Zuschlag auf Kosten trainiert	201,0	39,14 €
Quantilregression aufs kritische Quantil	227,6 (schlechtester)	**38,08 €** (beste)

Das Modell mit dem besten MSE hat die höchsten Kosten. Das kostenbeste Modell hat einen um 13 % schlechteren MSE — und spart 10,7 % der Kosten.

Wer Prognosemodelle nach MSE auswählt (und das ist die Voreinstellung jeder Modellauswahl-Bibliothek), wählt hier das falsche.

 Formel-Übersetzer: zwei Zielfunktionen

Mathematik	Was sie belohnt
$\min \sum_t (\hat{d}_t - d_t)^2$	„Liege im Mittel richtig." Abweichungen nach oben und unten zählen gleich, große Abweichungen überproportional.
$\min \sum_t \left[c_-(d_t - q_t)^+ + c_+(q_t - d_t)^+\right]$	„Sei lieber etwas zu großzügig als etwas zu knapp." Beide Richtungen zählen **verschieden** und beide nur linear.
$(x)^+$	„nur der positive Teil" — also $\max(0, x)$.
$c_- \neq c_+$	Der ganze Grund, warum die beiden Optima auseinanderfallen.

> Wären c_- und c_+ gleich groß, läge das Optimum beim **Median** — immer noch nicht beim Mittelwert, aber wenigstens in der Mitte. Erst die Asymmetrie schiebt es ans Quantil.

Warum ein pauschaler Zuschlag zu kurz greift

Die zweite Zeile der Tabelle ist die verbreitete Praxis: Punktprognose plus Sicherheitsbestand. Sie hilft (−7,3 %), bleibt aber hinter der Quantilregression zurück. Das Programm rechnet nach, warum:

	benötigter Zuschlag
normale Tage	3,5 Stück
Aktionstage	12,3 Stück
pauschal verwendet	**5,9 Stück**

Ein einziger Wert kann nur einen Mittelweg treffen: An normalen Tagen ist er 1,7-mal zu groß, an Aktionstagen erreicht er nur 48 % dessen, was nötig wäre.

Die Quantilregression schätzt das 66,7-%-Quantil **direkt aus den Merkmalen** und darf deshalb an verschiedenen Tagen verschieden weit über dem Erwartungswert liegen. Das ist der Unterschied zwischen *„ein Modell und danach eine Formel"* und *„ein Modell, das weiß, wofür es gebraucht wird"*.

> **Merksatz** Sagen Sie nicht den Erwartungswert vorher, sondern **die Größe, die in die Entscheidung eingeht**. Beim Newsvendor ist das das kritische Quantil — und das lässt sich direkt schätzen, statt es nachträglich aus einer Punktprognose zu basteln.

15.6 Eine Messfalle, in die der Autor selbst getappt ist

Der erste Entwurf dieses Programms bewertete auf **230 Testtagen** — ein realistischer Zeitraum, gut sieben Monate. Dort hatte die Quantilregression den *besseren* MSE, und die ganze Aussage dieses Kapitels stand auf dem Kopf.

Das war kein Denkfehler, sondern Zufall. Wie viel Zufall, lässt sich ausmessen:

Testfenster	Quantilregression sieht MSE-besser aus
180 Tage	11,8 %
365 Tage	5,5 %
730 Tage	0,0 %
2 000 Tage	0,0 %

Bei einem halben Jahr Testdaten sieht das (bezogen auf den MSE) schlechtere Modell in gut jedem zehnten Fall besser aus. Das ist keine große Zahl — aber **wer einmal misst, hat genau eine Ziehung aus dieser Verteilung.**

> **Was daraus folgt — und was nicht**
>
> Die Lehre ist **nicht** „nimm 4 000 Testtage". Die hat niemand; das Programm arbeitet deshalb mit simulierten Daten und sagt das auch.
>
> Die Lehre ist: **Ein Kennzahlenvergleich ohne Angabe seiner Streuung ist keine Aussage.** Bei kurzen Zeiträumen gehört eine Kreuzvalidierung dazu, und der Unterschied zwischen zwei Modellen muss größer sein als die Schwankung zwischen zwei Zeitfenstern.
>
> Verwandt, aber nicht dasselbe: Kapitel 21, `Data_Snooping.py`. Dort geht es um die Zahl der *Versuche*, hier um die Zahl der *Beobachtungen*. Beide Male führt eine weggelassene Angabe zu einer Aussage, die nicht trägt.

15.7 Wie weit das Verfahren reicht

Die dritte Zeile der Ergebnistabelle — „Zuschlag auf Kosten trainiert" — ist die einfachste denkbare Form einer Idee, die derzeit erforscht wird: **das Prognosemodell direkt auf die Entscheidungskosten zu trainieren** statt auf ein Fehlermaß.

Hier war es ein einziger Parameter, mit einer Rasterschleife bestimmt. Der allgemeine Fall ist schwieriger, weil die Optimierung zwischen Modell und Kosten steht: Um den Gradienten der Kosten nach den Modellparametern zu bilden, muss man **durch das Optimierungsproblem hindurch ableiten**.

Ansatz	Idee	Bibliothek
Richtige Zielgröße (dieses Kapitel)	Schätze das Quantil, das die Entscheidung braucht	`sklearn`

Ansatz	Idee	Bibliothek
SPO+ (Elmachtoub/Grigas)	Ein Ersatzverlust, der die Entscheidungskosten nach oben abschätzt und konvex ist	—
Differenzierbare Optimierung	Das Optimierungsproblem wird eine Schicht im neuronalen Netz	`cvxpylayers`

□ **Womit man anfängt**

Nicht mit `cvxpylayers`. Der Ertrag der ersten Zeile ist in der Praxis meist der größte und kostet einen Nachmittag: Fragen Sie, welche Größe die Entscheidung wirklich braucht, und schätzen Sie diese. In diesem Kapitel bringt das 10,7 % — die aufwendigen Verfahren ringen danach um die letzten Prozentpunkte.

15.8 Übungsaufgaben

Lösungen: Abschnitt A.15.

Aufgabe 15.1 □ **— Andere Preise.** Die Bäckerei kann übrige Ware am Folgetag zum halben Preis abgeben; der Überhang kostet dann nur noch 1,50 € statt 3 €. Wie ändert sich das kritische Verhältnis, und in welche Richtung verschiebt sich die Bestellmenge?

Aufgabe 15.2 □ **— Wann ist der Mittelwert richtig?** Nennen Sie die Bedingung, unter der Erwartungswert und optimale Bestellmenge zusammenfallen. Wie realistisch ist sie?

Aufgabe 15.3 □□ **— Die Kennzahl der Prognoseabteilung.** Ersetzen Sie im Programm den MSE durch den MAPE (mittlerer absoluter prozentualer Fehler). Ändert sich die Rangfolge der vier Verfahren? Begründen Sie, warum das kein Zufall ist.

Aufgabe 15.4 □□ **— Zwei Quantile.** Schätzen Sie zusätzlich das 5-%- und das 95-%-Quantil und zeichnen Sie für 30 Testtage ein Band um die Punktprognose. An welchen Tagen ist es breit — und passt das zu Ihrer Erwartung?

Aufgabe 15.5 □□□ **— Der Wert der Merkmale.** Entfernen Sie das Merkmal „Aktion” aus dem Modell. Wie verschlechtern sich MSE und Kosten — und zwar **in unterschiedlichem Ausmaß**? Was sagt das über den Wert eines Merkmals aus?

15.9 Finde den Denkfehler

 „Wir haben die Prognose um 18 % verbessert"

Ein Data-Science-Team stellt sein Quartalsergebnis vor:

> *„Wir haben das alte lineare Modell durch ein Gradient-Boosting-Modell ersetzt. Der MSE > auf den Testdaten ist um 18 % gesunken, der MAPE um 14 %. Das ist die größte > Prognoseverbesserung, die wir je erreicht haben. Die Disposition bekommt ab nächstem > Monat die neuen Werte."*

Sechs Wochen später meldet die Disposition, die Retouren seien gestiegen.

Das Team prüft alles nach: Die Modellgüte stimmt, die Testdaten waren sauber getrennt, es gibt kein Leck. Das neue Modell prognostiziert die Nachfrage tatsächlich deutlich besser als das alte.

Wie kann eine bessere Prognose zu schlechteren Ergebnissen führen — und was hätte das Team messen müssen?

Ein zweiter Hinweis für den zweiten Teil der Antwort: Die Disposition rechnet mit Punktprognose plus einem Sicherheitszuschlag, der aus den **Residuen des alten Modells** stammt und seit Jahren unverändert ist.

15.10 Micro-Quiz

 Drei Fragen

1. Warum ist der Erwartungswert beim Newsvendor die falsche Bestellmenge? a) Weil er statistisch unzuverlässig geschätzt wird. b) Weil die Kosten für Fehlmenge und Überhang verschieden hoch sind und das Optimum deshalb am kritischen Quantil liegt. c) Weil die Nachfrage nicht normalverteilt ist.

2. Ein Kollege schlägt vor, den Sicherheitszuschlag jährlich aus den Residuen neu zu berechnen. Was löst das nicht? a) Nichts — das ist die richtige Lösung. b) Der Zuschlag bleibt für alle Tage gleich, obwohl die Unsicherheit von den Merkmalen abhängt. c) Residuen sind grundsätzlich ungeeignet, um Unsicherheit zu schätzen.

3. Bei 180 Testtagen sieht das MSE-schlechtere Modell in 11,8 % der Fälle besser aus. Was folgt daraus für die Praxis? a) Man braucht mindestens 730 Testtage, sonst ist jeder Vergleich wertlos. b) Ein einzelner Kennzahlenvergleich ohne Streuungsangabe ist keine belastbare Aussage; bei kurzen Zeiträumen gehört eine Kreuzvalidierung dazu. c) Der MSE ist grundsätzlich unbrauchbar.

15.11 Selbsttest

1. Erklären Sie in zwei Sätzen, warum „predict then optimize" eine Naht hat, an der Information verlorengeht.
2. Welche Größe muss ein Prognosemodell liefern, wenn die Entscheidung ein Newsvendor ist — und woher kennen Sie den Wert?
3. Warum ist ein pauschaler Sicherheitszuschlag schwächer als eine Quantilregression? Nennen Sie die Dateneigenschaft, um die es geht.
4. Ihre Prognoseabteilung meldet 18 % besseren MSE. Welche eine Frage stellen Sie?
5. Wie viele Testdaten braucht ein Modellvergleich? Formulieren Sie die Antwort ohne eine konkrete Zahl.

15.12 Zusammenfassung

- **Predict-then-Optimize** trennt Schätzen und Entscheiden. Die erste Stufe wird auf ein statistisches Maß trainiert, die zweite erzeugt ökonomische Kosten — und die beiden Maße können in verschiedene Richtungen zeigen.
- Im Beispiel hat das Verfahren mit dem **besten MSE die höchsten Kosten**; das kostenbeste Verfahren hat einen um 13 % schlechteren MSE und spart 10,7 %.
- Der Grund ist die **Asymmetrie** der Kosten: Fehlmenge und Überhang kosten verschieden viel, deshalb liegt das Optimum am kritischen Quantil und nicht am Erwartungswert.
- Ein **pauschaler Sicherheitszuschlag** hilft, greift aber zu kurz, wenn die Unsicherheit selbst von den Merkmalen abhängt: Hier wäre er an normalen Tagen 1,7-mal zu groß und an Aktionstagen halb so groß wie nötig.
- **Bewerten Sie Prognosemodelle an den Entscheidungskosten.** Sie sind in Euro und damit vergleichbar; ein MSE ist es nicht.
- Ein Kennzahlenvergleich braucht eine **Streuungsangabe**. Bei 180 Testtagen sah das schlechtere Modell in gut jedem zehnten Fall besser aus — und wer einmal misst, hat genau eine Ziehung.

Synthese Teil III — Wenn die Idealwelt nicht gilt

Teil II setzte voraus: feste Daten, lineare Zusammenhänge, ein Ziel, eine Periode. Dieser Teil nimmt jede dieser vier Voraussetzungen einzeln weg. Die Tabelle ordnet die Kapitel danach, **welche** Voraussetzung bei Ihnen verletzt ist.

Welche Annahme fällt weg?

Was nicht gilt	Woran Sie es merken	Werkzeug	Kapitel
Linearität	Risiko wächst quadratisch, Kosten degressiv, Sättigung	QP, konvexe Optimierung, KKT	Kapitel 11
Feste Daten	Nachfrage, Wind, Rendite stehen erst morgen fest	Monte-Carlo, Zweistufigkeit, Robustheit, Chance Constraints	Kapitel 12
Eine Periode	die Entscheidung heute verändert, was übermorgen möglich ist	Bellman-Gleichung, Rückwärtsinduktion	Kapitel 13
Ein Ziel	zwei Kennzahlen sollen gleichzeitig stimmen	Pareto-Front, ε-Constraint	Kapitel 14
Daten sind gegeben	die Eingabe ist selbst eine Prognose	Predict-then-Optimize, entscheidungsorientier-tes Lernen	Kapitel 15

Die letzte Zeile ist die unbequemste: Dort ist nicht das Modell unsicher, sondern seine **Eingabe** — und die kommt aus einem zweiten Modell, das nach anderen Kriterien gebaut wurde.

Was dieser Teil gemessen hat

Behauptung	Gemessen	Wo
„Mit dem Mittelwert zu rechnen ist eine brauchbare Näherung.”	Der Mittelwertplan hält seine Zusage in **50,08 %** der Fälle — per Definition	Abschnitt 12.7
„Versorgungssicherheit kostet linear.”	56 289 € je Prozentpunkt auf dem Weg zu 80 %, **253 848 €** zwischen 95 und 99 % — das 4,5-fache	Abschnitt 12.7
„Eine Zusage aus dem Modell hält.”	Der 95-%-Plan hält **87,44 %**, sobald die Testverteilung einen Fall enthält, in dem alles zugleich ausfällt	Abschnitt 12.7
„Die bessere Prognose führt zur besseren Entscheidung.”	Das Modell mit dem **schlechteren** MSE trifft die günstigere Entscheidung	Kapitel 15

Drei Fehler, die dieser Teil verhindert

1. **Unsicherheit durch Mittelwerte ersetzen.** Der Durchschnittskunde kauft nie. Bei asymmetrischen Kosten liegt das Optimum systematisch neben dem Mittelwert, und eine Terminzusage ist ein **Quantil**, kein Erwartungswert.
2. **Konvexität voraussetzen, ohne sie zu prüfen.** Ohne sie gibt es keine Optimalitätsgarantie, sondern nur ein lokales Ergebnis — und der Solver sagt das nicht von selbst.
3. **Zwei Ziele mit einem Gewicht verrechnen, bevor die Front bekannt ist.** Ein Gewicht ist eine Antwort auf eine Frage, die noch niemand gestellt hat. Die Pareto-Front stellt sie zuerst.

Wenn Sie nur eines mitnehmen

 Jede der vier Annahmen lässt sich einzeln aufgeben — aber jede kostet etwas Bestimmtes: Linearität kostet die Optimalitätsgarantie, feste Daten kosten Rechenzeit oder Vorsicht, eine Periode kostet Zustandsraum, ein Ziel kostet eine Entscheidung, die nicht der Modellierer treffen darf. Wer weiß, **welche** Annahme bei ihm fällt, weiß auch, welchen Preis er zahlt.

Teil IV: Anwendungen — Energiewirtschaft und Finanzmärkte

Dieser Teil wechselt die Domäne — aber **nicht** die Methoden. Was hier folgt, ist die konsequente Anwendung dessen, was Sie in den Teilen II und III gelernt haben, auf zwei Gebiete, in denen viel Unsicherheit auf ein unmittelbar messbares Ergebnis trifft: die Energiewirtschaft und die Finanzmärkte.

Er hat drei Abschnitte, und Sie können nach jedem aufhören:

1. **Die Brücke** (Kapitel 16) macht die Behauptung nachprüfbar, dass es dieselben Methoden sind. Sie lässt denselben Code einmal über eine Werkstatt und einmal über ein Depot laufen — und benennt die drei Stellen, an denen die Analogie endet.
2. **Die Industrieanwendung** (Kapitel 17) führt die Werkzeuge an einem zusammenhängenden Fall aus der Energiewirtschaft zusammen: Kraftwerkseinsatzplanung unter Windunsicherheit. Wer mit Produktion, Logistik oder Energie zu tun hat, findet hier die Vertiefung, die für die Finanzkapitel keinen Ersatz braucht.
3. **Die Finanzanwendung** (ab Kapitel 18) ist die spezialisierte Vertiefungsdomäne.

   Für wen dieser Teil gedacht ist

   Die Finanzkapitel ab Kapitel 18 sind eine **Vertiefungsdomäne**, kein Pflichtstoff. Wer beruflich mit Logistik, Produktion oder Energie zu tun hat, liest die ersten beiden Kapitel dieses Teils und kann den Rest überspringen, ohne dass etwas fehlt — die Methoden sind dieselben.

   Lesen Sie ihn trotzdem, wenn Sie eines der folgenden Themen brauchen: Schätzfehler in Kovarianzmatrizen (das Problem tritt bei jeder Korrelationsschätzung auf, auch bei Lieferzeiten), Risikomaße jenseits der Standardabweichung, oder rollierende Auswertung mit sauberer Trennung von Trainings- und Testzeitraum.

Ein Hinweis vorweg: Die Kapitel ab Kapitel 18 laden aktuelle Kursdaten über `yfinance` und brauchen dafür einen **Internetzugang**. Weil die Daten sich täglich ändern, sind die abgedruckten Zahlen dort *Bei-*

spielläufe, keine reproduzierbaren Werte — anders als im übrigen Buch. Das Brückenkapitel kommt ohne Internet aus.

Kapitel 16: Die Strukturbrücke — dieselbe Mathematik, zwei Welten

Kapitel auf einen Blick

Worum geht es? Um den Nachweis, dass die Verfahren aus Teil II und III unverändert in der Finanzwelt funktionieren — und um die drei Stellen, an denen die Analogie endet.

Voraussetzungen: Kapitel 1 (Allokation als LP), Kapitel 12 (Szenarien) und Abschnitt 22.6 (`or_kern.py`).

Danach können Sie: ein betriebswirtschaftliches Modell in die andere Domäne übersetzen, ohne den Modellcode anzufassen; Schattenpreise in beiden Welten lesen; und benennen, welche Annahmen dabei **nicht** mitwandern.

Zeitbedarf: ca. 2,5 Stunden.

Programme:
`Strukturbruecke.py`

Notebook: `Notebooks_04/bruecke.ipynb`

16.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Was ist ein Risikobudget?

Ein Anleger hat 150 000 € und will sie auf drei Anlageklassen verteilen. Neben dem Kapital gibt es eine zweite knappe Größe: das **Risikobudget** — eine Kennzahl, die die Bankaufsicht oder das eigene Regelwerk vorgibt und die jede Anlage unterschiedlich stark beansprucht.

Anlage	Ertrag je 1 000 €	Kapital	Risikobudget
Aktien Welt	75 €	1,0	1,00
Anleihen	28 €	1,0	0,22
Immobilienfonds	46 €	1,0	0,55
verfügbar		**150**	**90**

Lesen Sie die Tabelle noch einmal — und ersetzen Sie „Kapital" durch „Montagestunden" und „Risikobudget" durch „Plattenmaterial". Es ist die Schreinerei aus Kapitel 1, Wort für Wort.

Deshalb genügt der vorhandene Code:

```
from or_kern import Produkt, Produktionsproblem, loese_mit_scipy

depot = Produktionsproblem(
    produkte=[
        Produkt(name="Aktien Welt", deckungsbeitrag=75.0,
                verbrauch={"Kapital (Tsd. EUR)": 1.0, "Risikobudget": 1.00}),
        Produkt(name="Anleihen", deckungsbeitrag=28.0,
                verbrauch={"Kapital (Tsd. EUR)": 1.0, "Risikobudget": 0.22}),
        Produkt(name="Immobilienfonds", deckungsbeitrag=46.0,
                verbrauch={"Kapital (Tsd. EUR)": 1.0, "Risikobudget": 0.55}),
    ],
    kapazitaeten={"Kapital (Tsd. EUR)": 150.0, "Risikobudget": 90.0})

loesung = loese_mit_scipy(depot)
print(loesung.als_bericht())
print(loesung.schattenpreise)
```

Ausgabe:

```
Status: optimal | Zielwert: 7,634.62 | Gap: 0.00% | Zeit: 0.00s
{'Kapital (Tsd. EUR)': 14.74, 'Risikobudget': 60.26}
```

Und jetzt der Punkt. Die Klasse heißt `Produktionsproblem`, und das Depot passt hinein, ohne dass eine Zeile geändert wurde. Kein Adapter, keine Unterklasse, keine Fallunterscheidung — die Struktur ist dieselbe, weil das Problem dasselbe ist: *knappe Größen auf konkurrierende Verwendungen verteilen.*

Der interessanteste Teil steht in der zweiten Zeile der Ausgabe. **60,26 € je Einheit Risikobudget** — das ist der Dualwert derselben Nebenbedingung, die in der Werkstatt sagt, was eine zusätzliche Montagestunde wert wäre. In der Finanzwelt hat diese Zahl einen eigenen Namen: der **Preis des Risikos**. Es ist nicht bloß eine Analogie, es ist derselbe Schattenpreis aus Kapitel 5.

 Merksatz Wenn zwei Probleme dieselbe Struktur haben, brauchen sie nicht zwei Modelle, sondern zwei Datensätze. Ob das der Fall ist, erkennt man daran, dass derselbe Code ohne Änderung läuft — nicht daran, dass eine Tabelle es behauptet.

Warum funktioniert das? Weil ein LP nichts über die Bedeutung seiner Zahlen weiß. Es sieht eine Matrix, einen Kapazitätsvektor und einen Zielvektor. Ob in der Matrix Stunden, Kilogramm oder Risikobeiträge stehen, kommt darin nicht vor.

16.2 Lernziele

Nach diesem Kapitel können Sie …

1. … ein Allokationsproblem aus der einen Domäne in die andere übersetzen, indem Sie die Daten austauschen statt des Codes.
2. … den Schattenpreis in beiden Welten deuten — als Wert einer Ressource und als Preis des Risikos.
3. … erklären, warum die CVaR-Formulierung aus Kapitel 20 auf Lieferverzüge genauso passt wie auf Kursverluste.
4. … die drei Stellen benennen, an denen die Analogie **nicht** trägt — und begründen, warum das Übertragen von Methoden dort gefährlich wird.

16.3 Das Programm

```
#!/usr/bin/env python3

# Strukturbruecke.py
"""
Kapitel Bruecke: Derselbe Code, zwei Welten - der Beweis statt der Behauptung.

Der Teil-Auftakt stellt eine Tabelle auf: "Ressourcen auf Produkte verteilen"
entspreche "Kapital auf Anlagen verteilen", "gegen den Worst Case absichern"
entspreche "Absicherung gegen Kursabstuerze". Solche Tabellen stehen in vielen
Buechern. Sie sind billig - und man kann sie pruefen.

Dieses Programm prueft sie. Es fuettert zweimal DENSELBEN Code mit Daten aus
zwei Welten und zeigt die Ergebnisse nebeneinander:

  1. Die Allokation als LP. Das Domaenenmodell aus or_kern.py, einmal mit einer
     Schreinerei (Montagestunden, Plattenmaterial) und einmal mit einem Depot
     (Kapital, Risikobudget). Gleiche Klasse, gleicher Modellbauer, gleiche
     Abnahmepruefung - und der Schattenpreis heisst in der einen Welt
     "Wert einer zusaetzlichen Montagestunde" und in der anderen "Preis des
     Risikos".
  2. Die Absicherung gegen den schlechtesten Fall als CVaR. EINE Funktion,
     einmal mit Kursrenditen und einmal mit Lieferverzuegen.
  3. Was NICHT hinueberreicht. Der ehrliche Teil: Drei Unterschiede, die die
     Analogie begrenzen - und die man kennen muss, bevor man Methoden aus
     der einen Welt in die andere traegt.

ZUR SOLVERWAHL: Teil 1 benutzt 'loese_mit_scipy' und nicht 'loese_mit_glop'.
Das ist kein Zufall - CVXPY laedt fuer Teil 2 highspy, und ortools vertraegt
sich damit nicht im selben Prozess (Kapitel Oekosystem). Wer hier GLOP nimmt,
bekommt beim cvxpy-Import eine Fehlermeldung ueber ein 'undefined symbol'.
Genau deshalb laedt or_kern.py seine Solver erst beim Aufruf.

Benoetigt: numpy, cvxpy, scipy und pydantic (ueber or_kern)
```

```
"""

from __future__ import annotations

import numpy as np

from or_kern import (Produkt, Produktionsproblem, loese_mit_scipy,
                     pruefe_loesung)

SAAT = 7
SZENARIEN = 500
ALPHA = 0.95              # die schlechtesten 5 % der Faelle
MAX_ANTEIL = 0.40         # Streuungsgebot in beiden Welten


# --- Teil 1: Dieselbe Klasse, zwei Welten --------------------------------

def schreinerei() -> Produktionsproblem:
    """Montagestunden und Plattenmaterial auf Tische und Stuehle verteilen."""
    return Produktionsproblem(
        produkte=[
            Produkt(name="Tisch", deckungsbeitrag=240.0,
                    verbrauch={"Montagestunden": 3.0, "Plattenmaterial": 6.0}),
            Produkt(name="Stuhl", deckungsbeitrag=60.0,
                    verbrauch={"Montagestunden": 1.0, "Plattenmaterial": 1.0}),
            Produkt(name="Regal", deckungsbeitrag=130.0,
                    verbrauch={"Montagestunden": 2.0, "Plattenmaterial": 4.0}),
        ],
        kapazitaeten={"Montagestunden": 150.0, "Plattenmaterial": 240.0})


def depot() -> Produktionsproblem:
    """Kapital und Risikobudget auf Anlageklassen verteilen.

    Eine "Einheit" ist hier 1.000 EUR Anlagesumme. Der Deckungsbeitrag ist der
    erwartete Jahresertrag dieser Einheit, der Verbrauch die beanspruchte
    Kapital- und Risikomenge. Das ist keine Analogie, sondern buchstaeblich
    dasselbe Modell - deshalb passt es in dieselbe Klasse.
    """
    return Produktionsproblem(
        produkte=[
            Produkt(name="Aktien Welt", deckungsbeitrag=75.0,
                    verbrauch={"Kapital (Tsd. EUR)": 1.0, "Risikobudget": 1.00}),
            Produkt(name="Anleihen", deckungsbeitrag=28.0,
                    verbrauch={"Kapital (Tsd. EUR)": 1.0, "Risikobudget": 0.22}),
            Produkt(name="Immobilienfonds", deckungsbeitrag=46.0,
                    verbrauch={"Kapital (Tsd. EUR)": 1.0, "Risikobudget": 0.55}),
        ],
```

```
        kapazitaeten={"Kapital (Tsd. EUR)": 150.0, "Risikobudget": 90.0})


def berichte_allokation(titel: str, problem: Produktionsproblem,
                        einheit: str, ertragsname: str) -> None:
    """Ein Bericht fuer beide Welten - nur die Beschriftung wechselt."""
    loesung = loese_mit_scipy(problem)
    beanstandungen = pruefe_loesung(problem, loesung)

    print(f"  {titel}")
    print(f"    {loesung.als_bericht()}")
    for produkt in problem.produkte:
        print(f"      {produkt.name:<18} {loesung.werte[produkt.name]:8.2f} {einheit}")
    print(f"      {ertragsname:<18} {loesung.zielwert:8.2f} EUR")
    for ressource, preis in loesung.schattenpreise.items():
        print(f"      Schattenpreis {ressource:<24} {preis:7.2f} EUR")
    print(f"      Abnahmepruefung: "
          f"{'bestanden' if not beanstandungen else beanstandungen}")


# --- Teil 2: Eine CVaR-Funktion, zwei Welten ------------------------------

def optimiere_cvar(verluste: np.ndarray, ertrag: np.ndarray,
                   mindestertrag: float, alpha: float = ALPHA,
                   max_anteil: float = MAX_ANTEIL):
    """Minimiert den CVaR der Verluste unter einer Mindestertragsbedingung.

    'verluste' hat die Form (Szenarien x Optionen) und enthaelt, was in
    Szenario s passiert, wenn eine Einheit in Option j steckt. Was ein
    "Verlust" ist, entscheidet allein die Einheit der Matrix: Prozentpunkte
    Kursverlust oder Tage Lieferverzug - die Formel sieht keinen Unterschied.

    Das ist die Rockafellar-Uryasev-Formulierung aus dem Kapitel CVaR, hier
    ohne jede Aenderung wiederverwendet.
    """
    import cvxpy as cp

    anzahl_szenarien, anzahl_optionen = verluste.shape
    anteil = cp.Variable(anzahl_optionen, nonneg=True)
    schwelle = cp.Variable()                        # wird im Optimum zum VaR
    ueberschuss = cp.Variable(anzahl_szenarien, nonneg=True)

    cvar = schwelle + (1.0 / (anzahl_szenarien * (1 - alpha))) * cp.sum(ueberschuss)
    problem = cp.Problem(
        cp.Minimize(cvar),
        [ueberschuss >= verluste @ anteil - schwelle,
         cp.sum(anteil) == 1,
         anteil <= max_anteil,
```

```
        ertrag @ anteil >= mindestertrag])
    problem.solve()
    if problem.status not in ("optimal", "optimal_inaccurate"):
        raise SystemExit(f"CVaR-Problem nicht loesbar: {problem.status}")
    return anteil.value, float(cvar.value), float(schwelle.value)


def kursszenarien(rng) -> tuple[np.ndarray, np.ndarray, list[str]]:
    """Taegliche Verluste (negative Renditen) von sechs Anlageklassen."""
    namen = ["Aktien Welt", "Aktien EU", "Schwellenlaender",
             "Staatsanleihen", "Unternehmensanl.", "Rohstoffe"]
    rendite_pa = np.array([0.080, 0.065, 0.110, 0.025, 0.045, 0.070])
    schwankung = np.array([0.180, 0.160, 0.260, 0.040, 0.075, 0.210])
    taeglich = rng.normal(rendite_pa / 252, schwankung / np.sqrt(252),
                          (SZENARIEN, len(namen)))
    return -taeglich * 100.0, rendite_pa * 100.0, namen


def lieferszenarien(rng) -> tuple[np.ndarray, np.ndarray, list[str]]:
    """Lieferverzug in Tagen bei sechs Lieferanten.

    Der Aufbau ist bewusst anders als bei den Kursen: Hier gibt es einen
    normalen Verzug UND seltene Totalausfaelle, die 14 Tage kosten. Das ist
    genau die Art fetter Raender, wegen der man in beiden Welten CVaR statt
    Standardabweichung benutzt.
    """
    namen = ["Nordwerk", "Sued-Metall", "Fernost A", "Lokalzulieferer",
             "Fernost B", "Osteuropa"]
    zuverlaessigkeit = np.array([0.94, 0.97, 0.90, 0.99, 0.92, 0.95])
    verzugsstreuung = np.array([3.5, 1.8, 5.0, 0.9, 4.2, 2.8])
    normaler_verzug = np.maximum(0.0, rng.normal(0.5, 1.0, (SZENARIEN, len(namen)))
                                 * verzugsstreuung)
    totalausfall = (rng.random((SZENARIEN, len(namen))) > zuverlaessigkeit) * 14.0
    return normaler_verzug + totalausfall, zuverlaessigkeit, namen


def berichte_cvar(titel: str, verluste, ertrag, namen, mindestertrag,
                  einheit: str, ertragsname: str):
    anteile, cvar, var = optimiere_cvar(verluste, ertrag, mindestertrag)
    mittel = float((verluste @ anteile).mean())
    print(f"  {titel}")
    print(f"    VaR  {ALPHA:.0%}: {var:8.3f} {einheit}")
    print(f"    CVaR {ALPHA:.0%}: {cvar:8.3f} {einheit}      "
          f"(Mittelwert ueber alle Szenarien: {mittel:.3f})")
    print(f"    {ertragsname}: {float(ertrag @ anteile):.3f}")
    print("    Aufteilung:")
    for name, anteil in zip(namen, anteile):
        balken = "#" * int(round(anteil * 40))
```

```
        grenze = "  <- an der Streuungsgrenze" if anteil > MAX_ANTEIL - 1e-4 else ""
        print(f"      {name:<18} {anteil:6.1%}  {balken}{grenze}")
    return anteile


if __name__ == "__main__":
    rng = np.random.default_rng(SAAT)

    print("=" * 80)
    print("  DIESELBE STRUKTUR, ZWEI WELTEN")
    print("=" * 80)

    # --- Teil 1 ------------------------------------------------------------
    print("\n1. Allokation als LP - EINE Klasse, EIN Modellbauer\n")
    berichte_allokation("Werkstatt: Produktionsprogramm", schreinerei(),
                        "Stueck", "Deckungsbeitrag")
    print()
    berichte_allokation("Depot: Anlageaufteilung", depot(),
                        "Tsd.  ", "Erwarteter Ertrag")

    print("\n  Beide Ausgaben stammen aus derselben Funktion "
          "'berichte_allokation'.")
    print("  Ausgetauscht wurden nur die Daten und die Beschriftungen -")
    print("  keine Zeile Modellcode.")
    print()
    print("  Lesen Sie die Schattenpreise nebeneinander: In der Werkstatt sagt")
    print("  er, was eine zusaetzliche Montagestunde wert waere. Im Depot sagt")
    print("  dieselbe Zahl, was eine zusaetzliche Einheit Risikobudget wert")
    print("  waere - der PREIS DES RISIKOS. Das ist kein Sprachbild, sondern")
    print("  derselbe Dualwert derselben Nebenbedingung.")

    # --- Teil 2 ------------------------------------------------------------
    print("\n" + "-" * 80)
    print("2. Absicherung gegen den schlechtesten Fall - EINE CVaR-Funktion\n")

    kurse, rendite, anlagen = kursszenarien(rng)
    anteile_depot = berichte_cvar(
                "Depot: die schlechtesten 5 % der Handelstage",
                kurse, rendite, anlagen, 5.5,
                "% je Tag    ", "Erwartete Jahresrendite (%)")
    print()
    lieferung, zuverlaessig, lieferanten = lieferszenarien(rng)
    anteile_einkauf = berichte_cvar("Einkauf: die schlechtesten 5 % der Bestellungen",
                lieferung, zuverlaessig, lieferanten, 0.945,
                "Tage Verzug  ", "Mittlere Zuverlaessigkeit")

    print("\n  Auch hier: eine Funktion, zwei Aufrufe. Die Zielfunktion")
    print("  interessiert sich nicht dafuer, ob in der Matrix Prozentpunkte")
```

```
    print("  oder Tage stehen.")
    print()
    print(f"  Interessant ist, WO die Streuungsgrenze von {MAX_ANTEIL:.0%} bindet:")
    print(f"    Depot   : {(anteile_depot > MAX_ANTEIL - 1e-4).sum()} von "
          f"{len(anteile_depot)} Posten am Anschlag")
    print(f"    Einkauf : {(anteile_einkauf > MAX_ANTEIL - 1e-4).sum()} von "
          f"{len(anteile_einkauf)} Posten am Anschlag")
    print()
    print("  Im Einkauf zieht es die Loesung an den sicheren Lokalzulieferer,")
    print("  bis die Grenze sie stoppt. Im Depot nicht - dort verhindert die")
    print("  Mindestrendite, dass alles in Anleihen wandert. Zwei verschiedene")
    print("  Bremsen also, und sie stehen an verschiedenen Stellen des Modells:")
    print("  einmal in einer Nebenbedingung ueber die Anteile, einmal in einer")
    print("  ueber den Ertrag. Wer eine Struktur uebertraegt, uebertraegt eben")
    print("  nicht automatisch mit, WELCHE Bedingung am Ende bindet.")

    # --- Teil 3: Der ehrliche Teil -------------------------------------
    print("\n" + "=" * 80)
    print("  WAS NICHT HINUEBERREICHT")
    print("=" * 80)
    print("Die Struktur traegt. Drei Unterschiede tragen NICHT mit, und wer sie")
    print("uebersieht, macht aus einer nuetzlichen Analogie einen Fehler:")
    print()
    print("1. WOHER DIE ZAHLEN KOMMEN.")
    print("   In der Werkstatt ist der Verbrauch je Tisch gemessen - drei")
    print("   Montagestunden sind drei Montagestunden. Im Depot ist die")
    print("   erwartete Rendite GESCHAETZT, und zwar mit einem Fehler, der")
    print("   groesser sein kann als die Unterschiede zwischen den Anlagen")
    print("   (Kapitel Markowitz, Renditeschaetzung_Falle.py). Dieselbe")
    print("   Optimierung ist im einen Fall Planung und im anderen")
    print("   Fehlerverstaerkung.")
    print()
    print("2. OB DIE VERGANGENHEIT ETWAS UEBER DIE ZUKUNFT SAGT.")
    print("   Lieferzeiten haben physikalische Ursachen: Entfernung, Zoll,")
    print("   Kapazitaet. Sie aendern sich langsam und nachvollziehbar.")
    print("   Kursrenditen entstehen aus dem Verhalten von Marktteilnehmern,")
    print("   die selbst auf Modelle reagieren - dort verschwindet ein")
    print("   erkanntes Muster oft genau deshalb, weil es erkannt wurde")
    print("   (Kapitel Handelsmaschine, Data_Snooping.py).")
    print()
    print("3. OB TEILBARKEIT ERLAUBT IST.")
    print("   37,4 % eines Aktienfonds sind ein normaler Auftrag. 37,4 % eines")
    print("   Lieferanten sind es nicht - Vertraege, Mindestabnahmen und")
    print("   Ruestzeiten machen Einkaufsentscheidungen ganzzahlig. Genau")
    print("   deshalb ist Teil II voller MILP und Teil V fast frei davon.")
    print()
    print("Die Bruecke traegt also die MODELLE, nicht die Annahmen. Wer sie")
    print("benutzt, spart sich das Lernen der Methoden - nicht das Nachdenken")
```

```
    print("ueber die Daten.")
    print("=" * 80)
```

Erwartete Ausgabe:

```
================================================================================
  DIESELBE STRUKTUR, ZWEI WELTEN
================================================================================

1. Allokation als LP - EINE Klasse, EIN Modellbauer

  Werkstatt: Produktionsprogramm
    Status: optimal | Zielwert: 10,800.00 | Gap: 0.00% | Zeit: 0.00s
      Tisch                 30.00 Stueck
      Stuhl                 60.00 Stueck
      Regal                  0.00 Stueck
      Deckungsbeitrag    10800.00 EUR
      Schattenpreis Montagestunden             40.00 EUR
      Schattenpreis Plattenmaterial            20.00 EUR
      Abnahmepruefung: bestanden

  Depot: Anlageaufteilung
    Status: optimal | Zielwert: 7,634.62 | Gap: 0.00% | Zeit: 0.00s
      Aktien Welt           73.08 Tsd.
      Anleihen              76.92 Tsd.
      Immobilienfonds        0.00 Tsd.
      Erwarteter Ertrag   7634.62 EUR
      Schattenpreis Kapital (Tsd. EUR)         14.74 EUR
      Schattenpreis Risikobudget               60.26 EUR
      Abnahmepruefung: bestanden

  Beide Ausgaben stammen aus derselben Funktion 'berichte_allokation'.
  Ausgetauscht wurden nur die Daten und die Beschriftungen -
  keine Zeile Modellcode.

  Lesen Sie die Schattenpreise nebeneinander: In der Werkstatt sagt
  er, was eine zusaetzliche Montagestunde wert waere. Im Depot sagt
  dieselbe Zahl, was eine zusaetzliche Einheit Risikobudget wert
  waere - der PREIS DES RISIKOS. Das ist kein Sprachbild, sondern
  derselbe Dualwert derselben Nebenbedingung.

--------------------------------------------------------------------------------
2. Absicherung gegen den schlechtesten Fall - EINE CVaR-Funktion

  Depot: die schlechtesten 5 % der Handelstage
    VaR  95%:    0.521 % je Tag
    CVaR 95%:    0.644 % je Tag          (Mittelwert ueber alle Szenarien: -0.011)
    Erwartete Jahresrendite (%): 5.500
```

```
    Aufteilung:
      Aktien Welt         14.9%  ######
      Aktien EU           12.2%  #####
      Schwellenlaender     9.6%  ####
      Staatsanleihen      26.8%  ###########
      Unternehmensanl.    30.7%  ############
      Rohstoffe            5.7%  ##

  Einkauf: die schlechtesten 5 % der Bestellungen
    VaR  95%:    4.351 Tage Verzug
    CVaR 95%:    5.716 Tage Verzug        (Mittelwert ueber alle Szenarien: 2.072)
    Mittlere Zuverlaessigkeit: 0.963
    Aufteilung:
      Nordwerk            12.6%  #####
      Sued-Metall         22.1%  #########
      Fernost A            6.2%  ##
      Lokalzulieferer     40.0%  ################  <- an der Streuungsgrenze
      Fernost B            8.7%  ###
      Osteuropa           10.4%  ####

  Auch hier: eine Funktion, zwei Aufrufe. Die Zielfunktion
  interessiert sich nicht dafuer, ob in der Matrix Prozentpunkte
  oder Tage stehen.

  Interessant ist, WO die Streuungsgrenze von 40% bindet:
    Depot   : 0 von 6 Posten am Anschlag
    Einkauf : 1 von 6 Posten am Anschlag

  Im Einkauf zieht es die Loesung an den sicheren Lokalzulieferer,
  bis die Grenze sie stoppt. Im Depot nicht - dort verhindert die
  Mindestrendite, dass alles in Anleihen wandert. Zwei verschiedene
  Bremsen also, und sie stehen an verschiedenen Stellen des Modells:
  einmal in einer Nebenbedingung ueber die Anteile, einmal in einer
  ueber den Ertrag. Wer eine Struktur uebertraegt, uebertraegt eben
  nicht automatisch mit, WELCHE Bedingung am Ende bindet.

================================================================================
  WAS NICHT HINUEBERREICHT
================================================================================
Die Struktur traegt. Drei Unterschiede tragen NICHT mit, und wer sie
uebersieht, macht aus einer nuetzlichen Analogie einen Fehler:

1. WOHER DIE ZAHLEN KOMMEN.
   In der Werkstatt ist der Verbrauch je Tisch gemessen - drei
   Montagestunden sind drei Montagestunden. Im Depot ist die
   erwartete Rendite GESCHAETZT, und zwar mit einem Fehler, der
   groesser sein kann als die Unterschiede zwischen den Anlagen
   (Kapitel Markowitz, Renditeschaetzung_Falle.py). Dieselbe
   Optimierung ist im einen Fall Planung und im anderen
```

```
   Fehlerverstaerkung.

2. OB DIE VERGANGENHEIT ETWAS UEBER DIE ZUKUNFT SAGT.
   Lieferzeiten haben physikalische Ursachen: Entfernung, Zoll,
   Kapazitaet. Sie aendern sich langsam und nachvollziehbar.
   Kursrenditen entstehen aus dem Verhalten von Marktteilnehmern,
   die selbst auf Modelle reagieren - dort verschwindet ein
   erkanntes Muster oft genau deshalb, weil es erkannt wurde
   (Kapitel Handelsmaschine, Data_Snooping.py).

3. OB TEILBARKEIT ERLAUBT IST.
   37,4 % eines Aktienfonds sind ein normaler Auftrag. 37,4 % eines
   Lieferanten sind es nicht - Vertraege, Mindestabnahmen und
   Ruestzeiten machen Einkaufsentscheidungen ganzzahlig. Genau
   deshalb ist Teil II voller MILP und Teil V fast frei davon.

Die Bruecke traegt also die MODELLE, nicht die Annahmen. Wer sie
benutzt, spart sich das Lernen der Methoden - nicht das Nachdenken
ueber die Daten.
================================================================================
```

16.4 Die zwei Brücken im Einzelnen

Erste Brücke: Allokation

Werkstatt	Depot	im Modell
Montagestunden, Plattenmaterial	Kapital, Risikobudget	`kapazitaeten`
Tisch, Stuhl, Regal	Aktien, Anleihen, Immobilien	`produkte`
Verbrauch je Stück	Beanspruchung je 1 000 €	`verbrauch`
Deckungsbeitrag je Stück	Erwarteter Ertrag je 1 000 €	`deckungsbeitrag`
Wert einer Montagestunde	**Preis des Risikos**	`schattenpreise`

Die letzte Zeile ist die, die sich zu merken lohnt. Ein Portfoliomanager, der sagt „unser Risikobudget ist zu knapp bemessen", trifft dieselbe Aussage wie ein Fertigungsleiter, der eine zweite Schicht fordert — und beide können sie mit derselben Zahl belegen.

Zweite Brücke: Absicherung gegen den schlechtesten Fall

Hier ist die Übereinstimmung noch enger, weil sie nicht nur die Struktur betrifft, sondern die *Formel*. Die Rockafellar-Uryasev-Formulierung aus Kapitel 20

$$\min_{w,\,\gamma} \;\; \gamma + \frac{1}{S(1-\alpha)} \sum_{s=1}^{S} u_s \qquad \text{mit} \qquad u_s \geq \ell_s^{\top} w - \gamma, \quad u_s \geq 0$$

steht im Programm genau einmal und wird zweimal aufgerufen.

 Formel-Übersetzer

Mathematik	im Depot	im Einkauf
w	Anteil je Anlageklasse	Bestellanteil je Lieferant
ℓ_s	Verlust in Szenario s, in Prozentpunkten	Verzug in Szenario s, in Tagen
γ	der VaR: „so schlimm wird es an 95 % der Tage höchstens"	„so viele Tage Verzug höchstens bei 95 % der Bestellungen"
u_s	Überschuss über diese Schwelle	dasselbe
Zielwert	CVaR: Mittelwert der schlechtesten 5 %	dasselbe

> **Ohne Formel gesagt:** „Sorge dafür, dass die schlimmsten fünf Prozent der Fälle im Mittel so glimpflich wie möglich ausgehen." Dieser Satz enthält kein Wort, das nur an der Börse Sinn ergibt.

Beide Anwendungen haben denselben Grund, warum sie CVaR und nicht die Standardabweichung benutzen: **fette Ränder.** Ein Lieferant fällt selten aus, aber wenn, dann kostet es vierzehn Tage — genau die Verteilungsform, bei der die Standardabweichung in die Irre führt (Kapitel 20).

Wo die Lösungen sich unterscheiden

Der Vergleich fördert nebenbei einen Unterschied zutage, den man nicht erwartet: Die Streuungsgrenze von 40 % **bindet nur im Einkauf**, nicht im Depot.

Im Einkauf zieht es die Lösung an den sicheren Lokalzulieferer, bis die Grenze sie stoppt. Im Depot verhindert stattdessen die Mindestrendite, dass alles in Anleihen wandert. Zwei verschiedene Bremsen also — und sie sitzen an verschiedenen Stellen des Modells.

Das ist die feine, wichtige Lehre dieses Abschnitts: **Wer eine Struktur überträgt, überträgt nicht automatisch mit, welche Nebenbedingung am Ende bindet.** Und das ist die Information, die im Gespräch mit dem Fachbereich zählt.

16.5 Wo die Brücke endet

Eine Analogie ist nur so nützlich wie die Grenzen, die man ihr zieht. Drei Dinge wandern **nicht** mit:

	Werkstatt / Einkauf	Depot
Woher die Zahlen kommen	gemessen: drei Montagestunden sind drei Montagestunden	geschätzt: die erwartete Rendite hat einen Fehler, der größer sein kann als die Unterschiede zwischen den Anlagen (Kapitel 19)
Ob die Vergangenheit trägt	Lieferzeiten haben physikalische Ursachen und ändern sich langsam	Kurse entstehen aus dem Verhalten von Marktteilnehmern, die auf Modelle reagieren — ein erkanntes Muster verschwindet oft deshalb, weil es erkannt wurde (Kapitel 21)
Ob Teilbarkeit erlaubt ist	37,4 % eines Lieferanten gibt es nicht: Verträge, Mindestabnahmen, Rüstzeiten → MILP	37,4 % eines Fonds sind ein normaler Auftrag → LP/QP genügt

□ Der Fehler, vor dem diese Tabelle schützt

Dieselbe Optimierung ist in der Werkstatt **Planung** und im Depot **Fehlerverstärkung**. Ein LP sucht die Ecke des zulässigen Bereichs mit dem besten Zielwert — und wenn die Zielkoeffizienten Schätzungen mit großem Fehler sind, sucht es genau die Ecke, an der der Schätzfehler am günstigsten aussieht.

In der Werkstatt gibt es dieses Problem nicht, weil der Deckungsbeitrag je Tisch aus der Kalkulation kommt und nicht aus einer Zeitreihe. Deshalb ist Shrinkage (Kapitel 18) im Depot Pflicht und in der Fertigung unbekannt.

Die dritte Zeile erklärt außerdem eine Beobachtung, die aufmerksamen Lesern längst aufgefallen ist: **Teil II ist voller MILP, Teil IV fast frei davon.** Das liegt nicht an der Methode, sondern daran, was in der jeweiligen Welt teilbar ist.

16.6 Übungsaufgaben

Lösungen: Abschnitt A.16.

Aufgabe 16.1 □ — Die dritte Ressource. Ergänzen Sie im Depot eine dritte knappe Größe: „Liquidität" (wie schnell sich eine Anlage verkaufen lässt). Was entspricht ihr in der Werkstatt?

Aufgabe 16.2 □ — Den Schattenpreis lesen. Das Risikobudget hat einen Schattenpreis von 60,26 €. Die Aufsicht bietet an, es gegen eine Gebühr um 10 Einheiten zu erhöhen. Bis zu welcher Gebühr lohnt sich das — und welche Einschränkung dieser Aussage kennen Sie aus Abschnitt 5.9?

Aufgabe 16.3 □□ — Die Brücke rückwärts. Übertragen Sie die Ledoit-Wolf-Shrinkage aus Kapitel 18 auf Lieferzeiten: Wo genau träte dort dasselbe Problem auf, und welche Daten bräuchten Sie?

Aufgabe 16.4 □□ — CVaR mit Ganzzahligkeit. Der Einkauf darf höchstens **drei** Lieferanten beauftragen. Ergänzen Sie die entsprechende Kardinalitätsbedingung (Kapitel 6, Muster 5) und vergleichen Sie den CVaR mit dem teilbaren Fall. Was kostet die Ganzzahligkeit?

Aufgabe 16.5 □□□ — Die eigene Brücke. Nehmen Sie ein Modell aus Ihrem Arbeitsumfeld und suchen Sie die Entsprechung in der jeweils anderen Welt. Prüfen Sie mit der Tabelle aus Abschnitt 16.5, ob die Übertragung trägt — und schreiben Sie auf, welche der drei Zeilen im Weg steht.

16.7 Finde den Denkfehler

□ „Das ist doch dasselbe Problem"

Eine Logistikerin liest dieses Kapitel und überträgt es zurück. Ihre Aufgabe: Aus 40 möglichen Lieferanten sollen die besten ausgewählt und Bestellmengen zugeteilt werden. Sie geht vor wie im Portfoliomanagement:

> *„Ich habe für jeden Lieferanten aus den letzten drei Jahren die mittlere Lieferzeit und > die Kovarianzmatrix der Lieferzeiten geschätzt. Dann habe ich das Markowitz-Modell > angewendet: minimiere die Varianz der Gesamtlieferzeit bei vorgegebener mittlerer > Lieferzeit. Das Ergebnis ist ein sauber diversifiziertes Lieferantenportfolio."*

Das Vorgehen ist methodisch sauber, die Rechnung stimmt, und das Ergebnis sieht plausibel aus.

Zwei Dinge daran sind falsch — und das eine ist genau der Fehler, vor dem dieses Kapitel warnt, das andere ein zusätzlicher, den es nicht behandelt. Welche?

16.8 Micro-Quiz

Drei Fragen

1. Warum passt das Depot in die Klasse `Produktionsproblem`, ohne dass etwas geändert werden muss? a) Weil `Produktionsproblem` mit `Any`-Typen arbeitet und alles akzeptiert. b) Weil beide Probleme dieselbe Struktur haben: knappe Größen auf konkurrierende Verwendungen verteilen. c) Weil Pydantic die Feldnamen automatisch übersetzt.

2. Der Schattenpreis des Risikobudgets beträgt 60,26 €. Was bedeutet das? a) Das Risikobudget ist zu 60,26 % ausgelastet. b) Eine zusätzliche Einheit Risikobudget würde den erwarteten Ertrag um rund 60,26 € erhöhen — solange die Basis sich nicht ändert. c) Jede Einheit Risikobudget kostet die Bank 60,26 €.

3. Welche der drei Grenzen der Analogie ist der Grund dafür, dass Teil II voller MILP ist und Teil IV fast frei davon? a) Woher die Zahlen kommen. b) Ob die Vergangenheit etwas über die Zukunft sagt. c) Ob Teilbarkeit erlaubt ist.

16.9 Selbsttest

1. Nennen Sie drei Größenpaare, die in Werkstatt und Depot einander entsprechen.
2. Warum weiß ein LP nichts über die Bedeutung seiner Zahlen — und warum ist das hier ein Vorteil?
3. Erklären Sie den „Preis des Risikos" jemandem, der nur Fertigungsplanung kennt.
4. Warum benutzen beide Welten CVaR statt der Standardabweichung? Nennen Sie die Verteilungseigenschaft, um die es geht.
5. Sie wollen ein Verfahren aus der Finanzwelt in die Produktion übertragen. Welche drei Fragen stellen Sie vorher?

16.10 Zusammenfassung

- Die Behauptung „dieselben Methoden, andere Domäne" ist prüfbar — und hier geprüft: Ein Depot passt unverändert in die Klasse `Produktionsproblem`, und **eine** CVaR-Funktion bearbeitet Kursverluste wie Lieferverzüge.
- Der **Schattenpreis** einer Ressource heißt in der Finanzwelt *Preis des Risikos*. Es ist derselbe Dualwert derselben Nebenbedingung.
- Die **CVaR-Formulierung** enthält kein Wort, das nur an der Börse Sinn ergibt. Beide Welten benutzen sie aus demselben Grund: fette Ränder, bei denen die Standardabweichung in die Irre führt.
- Übertragen wird die **Struktur**, nicht die **Annahmen**. Drei Dinge wandern nicht mit: woher die Zahlen kommen (gemessen gegen geschätzt), ob die Vergangenheit trägt (Physik gegen reagierende Marktteilnehmer) und ob Teilbarkeit erlaubt ist (MILP gegen LP).
- Die erste dieser Grenzen ist die gefährlichste: Dieselbe Optimierung ist in der Werkstatt Planung und im Depot Fehlerverstärkung.

Kapitel 17: Supply-Chain und Energieeinsatz unter Unsicherheit

Kapitel auf einen Blick

Worum geht es? Um einen zusammenhängenden Industriefall, der die Werkzeuge aus Teil II und III zusammenführt: gemischt-ganzzahlige Entscheidungen, Szenarien, Risikomaße — an der Kraftwerkseinsatzplanung, der klassischen Aufgabe der Energiewirtschaft.

Voraussetzungen: Kapitel 6, Kapitel 12 (zweistufige stochastische Programmierung) und Kapitel 20.

Danach können Sie: ein Modell aufsetzen, in dem die **erste Stufe ganzzahlig** ist und vor dem Zufall feststeht; erklären, warum eine Planung auf den Erwartungswert systematisch zu knapp ausfällt; und den Preis einer Versorgungssicherheitsvorgabe in Euro je vermiedener Megawattstunde ausweisen.

Zeitbedarf: ca. 4 Stunden.

Programme:
`Kraftwerkseinsatz.py`

Notebook: `Notebooks_04/supplychain.ipynb`

17.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Warum „billigstes Kraftwerk zuerst" falsch ist

Zwei Kraftwerke können 120 MW liefern. Der Kernblock produziert für 22 €/MWh, die Gasturbine für 105 € — fünfmal so teuer. Die Merit-Order-Regel sagt: Kernblock.

Sie übersieht die Anfahrkosten: 40 000 € gegen 1 500 €.

```
# Zwei Kraftwerke, die 120 MW decken sollen - aber wie lange?
#                        Grenzkosten   Anfahrkosten
kernblock  = dict(name="Kernblock",  grenz=22,  anfahrt=40_000)
gasturbine = dict(name="Gasturbine", grenz=105, anfahrt=1_500)
LEISTUNG = 120                                  # MW, die gebraucht werden

kosten = lambda kw, stunden: kw["anfahrt"] + kw["grenz"] * LEISTUNG * stunden

print(f"{'Dauer':>7} {'Kernblock':>12} {'Gasturbine':>12}   guenstiger")
for stunden in (1, 2, 3, 4, 5, 6, 8):
    k, g = kosten(kernblock, stunden), kosten(gasturbine, stunden)
    print(f"{stunden:>5} h {k:>12,.0f} {g:>12,.0f}   "
          f"{kernblock['name'] if k < g else gasturbine['name']}")

wechsel = (kernblock["anfahrt"] - gasturbine["anfahrt"]) / (
    (gasturbine["grenz"] - kernblock["grenz"]) * LEISTUNG)
print(f"\nUmschlagpunkt: {wechsel:.1f} Stunden")
```

Ausgabe:

```
Dauer    Kernblock   Gasturbine   guenstiger
  1 h       42,640       14,100   Gasturbine
  2 h       45,280       26,700   Gasturbine
  3 h       47,920       39,300   Gasturbine
  4 h       50,560       51,900   Kernblock
  5 h       53,200       64,500   Kernblock
  6 h       55,840       77,100   Kernblock
  8 h       61,120      102,300   Kernblock

Umschlagpunkt: 3.9 Stunden
```

Und jetzt der Punkt. Bis knapp vier Stunden ist die *fünfmal teurere* Gasturbine die günstigere Wahl. Erst danach hat der Kernblock seine Anfahrkosten wieder eingespielt.

Die Merit-Order — Kraftwerke nach Grenzkosten sortieren und von unten auffüllen — ist damit nur für einen **einzelnen** Zeitpunkt richtig. Sobald ein Tag geplant wird, hängt jede Stunde an allen anderen: Ein Block, der um 8 Uhr angefahren wird, muss bis mindestens 16 Uhr laufen, und diese Entscheidung fällt am Vorabend.

 Merksatz Anfahrkosten und Mindestlaufzeiten machen aus 24 unabhängigen Stundenentscheidungen **ein** Problem. Genau deshalb ist die Kraftwerkseinsatzplanung ein MILP und keine Sortierung.

Warum funktioniert das? Der Umschlagpunkt ist eine einfache Rechnung: Die Anfahrkostendifferenz (38 500 €) geteilt durch die Grenzkostendifferenz je Stunde (83 €/MWh × 120 MW = 9 960 €/h) ergibt 3,9 Stunden. Bei zwei Kraftwerken kann man das im Kopf ausrechnen — bei fünf Blöcken und 24 Stunden nicht mehr.

17.2 Lernziele

Nach diesem Kapitel können Sie …

1. … ein Unit-Commitment-Modell mit Anfahrkosten und Mindestlaufzeiten aufstellen.
2. … erklären, warum eine **binäre erste Stufe** das zweistufige Modell aus Kapitel 12 verschärft.
3. … zeigen, dass eine Planung auf den Erwartungswert nicht nur ungenau, sondern **systematisch zu knapp** ist.
4. … eine CVaR-Schranke auf eine physikalische Größe (nicht gedeckte Energie) legen.
5. … begründen, wann eine Risikoschranke etwas ändert — und wann sie überflüssig ist.

17.3 Die Aufgabe

Fünf Blöcke, 24 Stunden, eine Last zwischen 270 und 804 MW. Dazu Windeinspeisung, die niemand am Vorabend kennt.

Block	$P_{\min}$	$P_{\max}$	Grenzkosten	Anfahrkosten	Mindestlaufzeit
Kernblock	200	600	22 €/MWh	40 000 €	8 h
Braunkohle	100	400	35 €/MWh	18 000 €	6 h
Steinkohle	80	300	48 €/MWh	12 000 €	4 h
Gas GuD	50	250	72 €/MWh	6 000 €	2 h
Gasturbine	20	150	105 €/MWh	1 500 €	1 h

Die Reihenfolge ist typisch: **je billiger im Betrieb, desto träger und teurer im Anfahren.** Das ist keine Willkür, sondern Physik — ein großer Dampfprozess braucht Stunden, bis er auf Temperatur ist.

Die Zeitstruktur der Entscheidung

Hier geht das Kapitel über Abschnitt 12.5 hinaus:

Stufe	Entscheidung	Typ	wann
1	Welcher Block läuft in welcher Stunde?	**binär**, für alle Szenarien gleich	am Vorabend, bevor der Wind bekannt ist
2	Wie viel liefert jeder laufende Block?	kontinuierlich, je Szenario verschieden	am Tag selbst, laufend anpassbar

Im Kapitel Unsicherheit war die erste Stufe kontinuierlich (Kapazität kaufen, Menge festlegen). Hier ist sie **ganzzahlig** — und das ändert die Sache grundlegend: Eine Kapazität lässt sich anteilig erhöhen, ein Kraftwerk nicht zu 30 % anfahren.

Formel-Übersetzer: die Kopplung der beiden Stufen

Mathematik	Alltagssprache
$u_{k,t} \in \{0,1\}$	„Läuft Block k in Stunde t?" — **eine** Antwort für alle Szenarien
$p^s_{k,t} \geq P^{\min}_k \, u_{k,t}$	„Ein laufender Block liefert mindestens seine Mindestleistung."
$p^s_{k,t} \leq P^{\max}_k \, u_{k,t}$	„Ein stehender Block liefert gar nichts." — dieselbe Big-M-Kopplung wie in Kapitel 6
$a_{k,t} \geq u_{k,t} - u_{k,t-1}$	„Wer eben noch aus war und jetzt an ist, ist angefahren."
$u_{k,\tau} \geq a_{k,t}$ für $\tau = t \ldots t{+}L_k$	Mindestlaufzeit: „Wer anfährt, bleibt L_k Stunden am Netz."

Mathematik	Alltagssprache
$\sum_k p^s_{k,t} + w^s_t + y^s_t \geq D_t$	„Erzeugung plus Wind plus Nichtdeckung deckt die Last.“

Die beiden mittleren Zeilen sind der ganze Trick: Sie verbinden eine **binäre** Größe (u) mit einer **kontinuierlichen** (p) — und weil u szenarioübergreifend gleich ist, die Szenarien untereinander.

17.4 Das Programm

```
#!/usr/bin/env python3

# Kraftwerkseinsatz.py
"""
Kapitel Supply-Chain: Welche Bloecke laufen morgen? Und was, wenn kein Wind weht?

Die Kraftwerkseinsatzplanung (englisch Unit Commitment) ist die Aufgabe, an der
sich in der Energiewirtschaft alles entscheidet: Fuer jede Stunde des naechsten
Tages muss feststehen, welche Bloecke am Netz sind. Ein Kernblock braucht acht
Stunden Mindestlaufzeit und 40.000 Euro Anfahrkosten - wer ihn abschaltet, hat
ihn fuer den Rest des Tages verloren.

Das Besondere liegt in der Zeitstruktur, und es geht ueber das zweistufige
Modell aus dem Kapitel Unsicherheit hinaus:

    STUFE 1   Das AN/AUS je Block und Stunde. Binaer, und es steht am Vorabend
              fest - bevor irgendjemand weiss, wie viel Wind morgen weht.
    STUFE 2   Die Fahrweise: wie viel jeder laufende Block liefert. Das darf
              sich stundenweise an die Wirklichkeit anpassen.

Die erste Stufe ist also GANZZAHLIG und szenariouebergreifend gleich, die
zweite kontinuierlich und je Szenario verschieden. Genau diese Kombination
macht das Problem interessant - und sie ist der Grund, warum ein Plan, der auf
den Wind-Erwartungswert gerechnet wurde, in der Wirklichkeit teuer wird.

Das Programm zeigt drei Plaene auf denselben Daten:

  1. Deterministisch: gerechnet mit dem Wind-ERWARTUNGSWERT, danach gegen 40
     Szenarien ausgewertet.
  2. Zweistufig: der Commitment-Plan sieht alle 40 Szenarien.
  3. Mit Versorgungssicherheit: die Nichtdeckung in den schlechtesten Faellen
     wird begrenzt - und der Preis dafuer in Euro je vermiedener MWh
     ausgewiesen.

Benoetigt: numpy, ortools
```

```
"""

from __future__ import annotations

import time

import numpy as np
from ortools.linear_solver import pywraplp

# (Name, Mindestleistung, Nennleistung, Grenzkosten, Anfahrkosten, Mindestlaufzeit)
KRAFTWERKE = [
    ("Kernblock",   200, 600,  22, 40_000, 8),
    ("Braunkohle",  100, 400,  35, 18_000, 6),
    ("Steinkohle",   80, 300,  48, 12_000, 4),
    ("Gas GuD",      50, 250,  72,  6_000, 2),
    ("Gasturbine",   20, 150, 105,   1_500, 1),
]
STUNDEN = 24
SZENARIEN = 40
FLAUTENANTEIL = 0.15
ALPHA = 0.90                  # die schlechtesten 10 % der Szenarien
LASTABWURF = 3_000.0          # EUR je nicht gedeckter MWh ("value of lost load")
NIEDRIGER_ABWURFPREIS = 300.0  # zum Vergleich: ein Marktpreisdeckel
SAAT = 4

_t = np.arange(STUNDEN)
LAST = 620 + 260 * np.sin((_t - 7) / 24 * 2 * np.pi) + 90 * np.sin((_t - 4) / 12 * 2 * np.pi)
WIND_ERWARTUNG = np.maximum(0.0, 160 + 130 * np.sin((_t - 14) / 24 * 2 * np.pi))


def erzeuge_windszenarien(anzahl: int = SZENARIEN, saat: int = SAAT):
    """Windeinspeisung je Szenario und Stunde.

    Zwei Zutaten, die zusammen den Unterschied machen: eine breite
    lognormale Streuung des Tagesniveaus - und in 15 % der Faelle eine
    DUNKELFLAUTE, in der praktisch gar kein Wind weht. Solche seltenen,
    extremen Faelle sind der Grund, warum der Erwartungswert als
    Planungsgrundlage nicht genuegt.
    """
    rng = np.random.default_rng(saat)
    tagesniveau = rng.lognormal(0, 0.40, (anzahl, 1))
    wind = np.clip(WIND_ERWARTUNG * tagesniveau
                   + rng.normal(0, 25, (anzahl, STUNDEN)), 0, 420)
    ist_flaute = rng.random(anzahl) < FLAUTENANTEIL
    wind[ist_flaute] *= 0.05
    return wind, ist_flaute
```

```
def plane(wind: np.ndarray, gewichte, cvar_grenze: float | None = None,
          abwurfpreis: float = None, zeitlimit: float = 300.0):
    """Commitment (Stufe 1) und Fahrweise je Szenario (Stufe 2) in einem Modell.

    'wind' hat die Form (Szenarien, Stunden). Mit einer einzigen Zeile
    Windprognose wird daraus die deterministische Planung.

    'cvar_grenze' begrenzt den CVaR der Nichtdeckung ueber die Szenarien -
    dieselbe Rockafellar-Uryasev-Konstruktion wie im Kapitel CVaR, nur dass
    hier keine Verluste in Euro, sondern Megawattstunden begrenzt werden.
    """
    abwurfpreis = LASTABWURF if abwurfpreis is None else abwurfpreis
    anzahl_szenarien = wind.shape[0]
    anzahl_bloecke = len(KRAFTWERKE)
    solver = pywraplp.Solver.CreateSolver("SCIP")
    solver.SetTimeLimit(int(zeitlimit * 1000))

    # --- Stufe 1: binaer, szenariouebergreifend gleich --------------------
    laeuft = [[solver.BoolVar(f"laeuft_{k}_{t}") for t in range(STUNDEN)]
              for k in range(anzahl_bloecke)]
    faehrt_an = [[solver.BoolVar(f"start_{k}_{t}") for t in range(STUNDEN)]
                 for k in range(anzahl_bloecke)]

    # --- Stufe 2: kontinuierlich, je Szenario -----------------------------
    leistung = [[[solver.NumVar(0, KRAFTWERKE[k][2], f"p_{j}_{k}_{t}")
                  for t in range(STUNDEN)] for k in range(anzahl_bloecke)]
                for j in range(anzahl_szenarien)]
    nichtdeckung = [[solver.NumVar(0, solver.infinity(), f"y_{j}_{t}")
                     for t in range(STUNDEN)] for j in range(anzahl_szenarien)]

    for k, (_, pmin, pmax, _, _, mindestlaufzeit) in enumerate(KRAFTWERKE):
        for t in range(STUNDEN):
            # Anfahren erkennen: aus im Vortakt, an im aktuellen
            vorher = laeuft[k][t - 1] if t > 0 else 0
            solver.Add(faehrt_an[k][t] >= laeuft[k][t] - vorher)
            # Mindestlaufzeit: wer anfaehrt, laeuft die naechsten Stunden weiter
            for spaeter in range(t, min(STUNDEN, t + mindestlaufzeit)):
                solver.Add(laeuft[k][spaeter] >= faehrt_an[k][t])
            # Ein laufender Block liefert zwischen Mindest- und Nennleistung,
            # ein stehender gar nichts. Das koppelt beide Stufen.
            for j in range(anzahl_szenarien):
                solver.Add(leistung[j][k][t] >= pmin * laeuft[k][t])
                solver.Add(leistung[j][k][t] <= pmax * laeuft[k][t])

    for j in range(anzahl_szenarien):
        for t in range(STUNDEN):
            solver.Add(sum(leistung[j][k][t] for k in range(anzahl_bloecke))
                       + float(wind[j, t]) + nichtdeckung[j][t] >= LAST[t])
```

```
    # --- Versorgungssicherheit als CVaR-Schranke -------------------------
    if cvar_grenze is not None:
        schwelle = solver.NumVar(-solver.infinity(), solver.infinity(), "schwelle")
        ueberschuss = [solver.NumVar(0, solver.infinity(), f"u_{j}")
                       for j in range(anzahl_szenarien)]
        for j in range(anzahl_szenarien):
            solver.Add(ueberschuss[j] >= sum(nichtdeckung[j][t]
                                             for t in range(STUNDEN)) - schwelle)
        solver.Add(schwelle + (1.0 / (anzahl_szenarien * (1 - ALPHA)))
                   * sum(ueberschuss) <= cvar_grenze)

    anfahrkosten = sum(KRAFTWERKE[k][4] * faehrt_an[k][t]
                       for k in range(anzahl_bloecke) for t in range(STUNDEN))
    betriebskosten = sum(
        gewichte[j] * sum(KRAFTWERKE[k][3] * leistung[j][k][t]
                          for k in range(anzahl_bloecke) for t in range(STUNDEN))
        for j in range(anzahl_szenarien))
    abwurfkosten = sum(gewichte[j] * abwurfpreis
                       * sum(nichtdeckung[j][t] for t in range(STUNDEN))
                       for j in range(anzahl_szenarien))
    solver.Minimize(anfahrkosten + betriebskosten + abwurfkosten)

    beginn = time.perf_counter()
    status = solver.Solve()
    dauer = time.perf_counter() - beginn
    if status not in (pywraplp.Solver.OPTIMAL, pywraplp.Solver.FEASIBLE):
        return None, dauer, False
    plan = np.array([[laeuft[k][t].solution_value() for t in range(STUNDEN)]
                     for k in range(anzahl_bloecke)]).round()
    return plan, dauer, status == pywraplp.Solver.OPTIMAL


def bewerte(plan: np.ndarray, wind: np.ndarray, abwurfpreis: float = None):
    """Commitment steht fest - nur noch die Fahrweise je Szenario optimieren.

    Das ist die ehrliche Bewertung eines Plans: Er wird der Wirklichkeit
    ausgesetzt und darf nur noch das anpassen, was sich am Tag selbst
    anpassen laesst.
    """
    abwurfpreis = LASTABWURF if abwurfpreis is None else abwurfpreis
    anzahl_bloecke = len(KRAFTWERKE)
    kosten, fehlmengen = [], []
    for j in range(wind.shape[0]):
        solver = pywraplp.Solver.CreateSolver("GLOP")
        leistung = [[solver.NumVar(0, KRAFTWERKE[k][2], f"p_{k}_{t}")
                     for t in range(STUNDEN)] for k in range(anzahl_bloecke)]
        fehlt = [solver.NumVar(0, solver.infinity(), f"y_{t}")
```

```
                for t in range(STUNDEN)]
        for k, (_, pmin, pmax, _, _, _) in enumerate(KRAFTWERKE):
            for t in range(STUNDEN):
                solver.Add(leistung[k][t] >= pmin * plan[k, t])
                solver.Add(leistung[k][t] <= pmax * plan[k, t])
        for t in range(STUNDEN):
            solver.Add(sum(leistung[k][t] for k in range(anzahl_bloecke))
                       + float(wind[j, t]) + fehlt[t] >= LAST[t])
        solver.Minimize(
            sum(KRAFTWERKE[k][3] * leistung[k][t]
                for k in range(anzahl_bloecke) for t in range(STUNDEN))
            + sum(abwurfpreis * fehlt[t] for t in range(STUNDEN)))
        solver.Solve()
        kosten.append(solver.Objective().Value())
        fehlmengen.append(sum(f.solution_value() for f in fehlt))

    anfahrten = sum(KRAFTWERKE[k][4]
                    * max(0.0, plan[k, t] - (plan[k, t - 1] if t > 0 else 0.0))
                    for k in range(anzahl_bloecke) for t in range(STUNDEN))
    return np.array(kosten) + anfahrten, np.array(fehlmengen)


def cvar(werte: np.ndarray, alpha: float = ALPHA) -> float:
    """Mittelwert der schlechtesten (1-alpha) Faelle."""
    schwelle = np.quantile(werte, alpha)
    schlechteste = werte[werte >= schwelle]
    return float(schlechteste.mean()) if len(schlechteste) else 0.0


def zeige(name: str, plan, wind, abwurfpreis: float = None) -> dict:
    kosten, fehl = bewerte(plan, wind, abwurfpreis)
    kennzahlen = {"mittel": kosten.mean(), "max": kosten.max(),
                  "fehl_mittel": fehl.mean(), "fehl_cvar": cvar(fehl),
                  "betroffen": int((fehl > 0.01).sum()),
                  "blockstunden": int(plan.sum())}
    print(f"  {name:<32} {kennzahlen['mittel']:>10,.0f} {kennzahlen['max']:>11,.0f} "
          f"{kennzahlen['fehl_mittel']:>9.1f} {kennzahlen['fehl_cvar']:>10.1f} "
          f"{kennzahlen['betroffen']:>6}/{len(fehl)}")
    return kennzahlen


if __name__ == "__main__":
    wind, ist_flaute = erzeuge_windszenarien()

    print("=" * 88)
    print("  KRAFTWERKSEINSATZ: DER PLAN STEHT, BEVOR DER WIND WEHT")
    print("=" * 88)
    print(f"{len(KRAFTWERKE)} Bloecke, {STUNDEN} Stunden, {SZENARIEN} Windszenarien "
```

```
        f"(davon {int(ist_flaute.sum())} Dunkelflauten).")
    print(f"Last {LAST.min():.0f} bis {LAST.max():.0f} MW, "
          f"Wind im Erwartungswert {WIND_ERWARTUNG.min():.0f} bis "
          f"{WIND_ERWARTUNG.max():.0f} MW.")
    print(f"Nicht gedeckte Last kostet {LASTABWURF:,.0f} EUR je MWh.\n")

    print(f"  {'Planungsgrundlage':<32} {'Kosten':>10} {'Kosten':>11} "
          f"{'Fehlmenge':>9} {'Fehlmenge':>10} {'Szen. mit':>12}")
    print(f"  {'':<32} {'im Mittel':>10} {'schlimmst.':>11} "
          f"{'Mittel':>9} {'CVaR 90%':>10} {'Abwurf':>12}")
    print("  " + "-" * 86)

    # --- 1. Deterministisch ------------------------------------------------
    plan_det, dauer_det, _ = plane(WIND_ERWARTUNG.reshape(1, -1), [1.0])
    det = zeige("nur Wind-Erwartungswert", plan_det, wind)

    # --- 2. Zweistufig, risikoneutral --------------------------------------
    gleich = [1.0 / SZENARIEN] * SZENARIEN
    plan_sto, dauer_sto, optimal_sto = plane(wind, gleich)
    sto = zeige("alle 40 Szenarien", plan_sto, wind)

    # --- 3. Mit Versorgungssicherheit --------------------------------------
    plan_sicher, dauer_sicher, optimal_sicher = plane(wind, gleich, cvar_grenze=0.0)
    sicher = zeige("+ Nichtdeckung CVaR 90 % = 0", plan_sicher, wind)

    print(f"\n  Rechenzeiten: {dauer_det:.1f}s / {dauer_sto:.1f}s / {dauer_sicher:.1f}s "
          f"(alle beweisbar optimal: {optimal_sto and optimal_sicher})")

    # --- Was die Zeilen bedeuten -------------------------------------------
    print("\n" + "-" * 88)
    print("Was der Erwartungswert-Plan anrichtet\n")
    print(f"  Er ist auf dem Papier der billigste - gerechnet auf den mittleren")
    print(f"  Wind kostet er weniger als jeder andere. In der Wirklichkeit liegt")
    print(f"  er {det['mittel'] / sto['mittel'] - 1:+.0%} ueber dem zweistufigen Plan, und
     ↪  sein "
          f"schlimmster Tag")
    print(f"  kostet {det['max'] / sto['max']:.1f}-mal so viel.")
    print(f"\n  Der Grund steht in der letzten Spalte: In {det['betroffen']} von {SZENARIEN} "
          f"Szenarien muss")
    print(f"  Last abgeworfen werden. Was fehlt, sind nicht Kraftwerke, sondern")
    print(f"  ANGEFAHRENE Kraftwerke - und ein Block mit acht Stunden")
    print(f"  Mindestlaufzeit laesst sich um 18 Uhr nicht mehr herbeirufen.")

    print(f"\n  Der Unterschied im Plan ist klein:")
    for name, plan in [("Erwartungswert", plan_det), ("zweistufig", plan_sto),
                       ("mit Sicherheit", plan_sicher)]:
        laufend = plan.sum(axis=0).astype(int)
        print(f"    {name:<16} {''.join(str(x) for x in laufend)}  "
```

```
          f"({int(plan.sum())} Blockstunden)")
print(f"\n  Der zweistufige Plan haelt {sto['blockstunden'] - det['blockstunden']} "
      f"Blockstunden mehr vor - das genuegt,")
print(f"  um alle {det['betroffen']} Lastabwuerfe zu vermeiden.")

# --- Der Preis der Sicherheit ----------------------------------------
print("\n" + "-" * 88)
print("Was Versorgungssicherheit kostet\n")
print(f"  Bei {LASTABWURF:,.0f} EUR/MWh kostet die CVaR-Schranke NICHTS: Der")
print("  risikoneutrale Plan haelt sie schon ein. Das ist kein Zufall - bei")
print("  diesem Preis lohnt sich Vorhaltung bereits im Erwartungswert.")
print("\n  Interessant wird die Schranke dort, wo der Schaden ZU NIEDRIG")
print("  bepreist ist. Dieselbe Rechnung mit einem Marktpreisdeckel von")
print(f"  {NIEDRIGER_ABWURFPREIS:,.0f} EUR/MWh statt {LASTABWURF:,.0f}:\n")

print(f"  {'Planungsgrundlage':<32} {'Kosten':>10} {'Kosten':>11} "
      f"{'Fehlmenge':>9} {'Fehlmenge':>10} {'Szen. mit':>12}")
print(f"  {'':<32} {'im Mittel':>10} {'schlimmst.':>11} "
      f"{'Mittel':>9} {'CVaR 90%':>10} {'Abwurf':>12}")
print("  " + "-" * 86)
plan_billig, _, _ = plane(wind, gleich, abwurfpreis=NIEDRIGER_ABWURFPREIS)
billig = zeige("risikoneutral, billiger Abwurf", plan_billig, wind,
               NIEDRIGER_ABWURFPREIS)
plan_billig_sicher, _, _ = plane(wind, gleich, cvar_grenze=0.0,
                                 abwurfpreis=NIEDRIGER_ABWURFPREIS)
billig_sicher = zeige("+ CVaR 90 % der Fehlmenge = 0", plan_billig_sicher, wind,
                      NIEDRIGER_ABWURFPREIS)

aufpreis = billig_sicher["mittel"] - billig["mittel"]
vermieden = billig["fehl_cvar"] - billig_sicher["fehl_cvar"]
print(f"\n  Jetzt greift die Schranke: Der risikoneutrale Plan nimmt in")
print(f"  {billig['betroffen']} von {SZENARIEN} Szenarien einen Lastabwurf in Kauf, weil
 ↪  er bei")
print(f"  {NIEDRIGER_ABWURFPREIS:,.0f} EUR/MWh billiger ist als das Vorhalten eines
 ↪  Blocks.")
print(f"\n  Aufpreis fuer die Sicherheit: {aufpreis:,.0f} EUR je Tag")
if vermieden > 0.01:
    print(f"  Vermiedene Fehlmenge (CVaR 90 %): {vermieden:.1f} MWh")
    print(f"  -> {aufpreis / vermieden:,.0f} EUR je vermiedener MWh")
    print(f"\n  Diese Zahl ist die Entscheidungsgrundlage - genau wie die Spalte")
    print(f"  'EUR je kg' im Kapitel Mehrziel. Sie sagt der Aufsicht, was ihre")
    print(f"  Versorgungssicherheitsvorgabe tatsaechlich kostet, statt darueber")
    print(f"  zu streiten, wie wichtig sie ist.")

print("\n" + "=" * 88)
print("  WAS MAN DARAUS MITNIMMT")
print("=" * 88)
print("1. Die erste Stufe ist binaer und traege. Was am Vorabend nicht")
```

```
    print("   angefahren wurde, steht am naechsten Tag nicht zur Verfuegung -")
    print("   egal, wie hoch der Preis dann steigt.")
    print("2. Deshalb ist der Erwartungswert als Planungsgrundlage nicht nur")
    print("   ungenau, sondern SYSTEMATISCH zu knapp: Er plant fuer einen Tag,")
    print("   der so nie eintritt, und laesst keine Reserve fuer die Haelfte")
    print("   der Faelle, in denen es schlechter kommt.")
    print("3. Der Ausweg ist keine bessere Windprognose, sondern ein Modell,")
    print("   das die Szenarien SIEHT. Die Rechenzeit dafuer liegt hier im")
    print("   Sekundenbereich - der Aufwand ist die Modellierung, nicht der")
    print("   Solver.")
    print("4. Wo der Lastabwurf teuer genug bepreist ist, deckt schon die")
    print("   Minimierung der ERWARTETEN Kosten die Extremfaelle mit ab. Die")
    print("   Risikoschranke wird genau dann gebraucht, wenn der Schaden ZU")
    print("   NIEDRIG bepreist ist - was bei Versorgungssicherheit die Regel")
    print("   ist, weil ein Marktpreisdeckel nicht den volkswirtschaftlichen")
    print("   Schaden abbildet. Dann liefert sie den Preis der Vorgabe in Euro")
    print("   je vermiedener MWh, und darueber laesst sich verhandeln.")
    print("=" * 88)
```

Erwartete Ausgabe (Laufzeiten hardwareabhängig, die Werte nicht):

```
============================================================================================
  KRAFTWERKSEINSATZ: DER PLAN STEHT, BEVOR DER WIND WEHT
============================================================================================
5 Bloecke, 24 Stunden, 40 Windszenarien (davon 8 Dunkelflauten).
Last 270 bis 804 MW, Wind im Erwartungswert 30 bis 290 MW.
Nicht gedeckte Last kostet 3,000 EUR je MWh.

  Planungsgrundlage                    Kosten      Kosten Fehlmenge  Fehlmenge    Szen. mit
                                   im Mittel  schlimmst.    Mittel   CVaR 90%       Abwurf
  ------------------------------------------------------------------------------------------
  nur Wind-Erwartungswert            874,870   2,504,154     178.7      679.2     28/40
  alle 40 Szenarien                  351,356     415,924       0.0        0.0      0/40
  + Nichtdeckung CVaR 90 % = 0       351,356     415,924       0.0        0.0      0/40

  Rechenzeiten: 0.0s / 1.3s / 1.1s (alle beweisbar optimal: True)

--------------------------------------------------------------------------------------------
Was der Erwartungswert-Plan anrichtet

  Er ist auf dem Papier der billigste - gerechnet auf den mittleren
  Wind kostet er weniger als jeder andere. In der Wirklichkeit liegt
  er +149% ueber dem zweistufigen Plan, und sein schlimmster Tag
  kostet 6.0-mal so viel.

  Der Grund steht in der letzten Spalte: In 28 von 40 Szenarien muss
  Last abgeworfen werden. Was fehlt, sind nicht Kraftwerke, sondern
  ANGEFAHRENE Kraftwerke - und ein Block mit acht Stunden
```

```
  Mindestlaufzeit laesst sich um 18 Uhr nicht mehr herbeirufen.

  Der Unterschied im Plan ist klein:
    Erwartungswert   111111122222222211111111  (33 Blockstunden)
    zweistufig       111111222222222222222111  (39 Blockstunden)
    mit Sicherheit   111111222222222222222111  (39 Blockstunden)

  Der zweistufige Plan haelt 6 Blockstunden mehr vor - das genuegt,
  um alle 28 Lastabwuerfe zu vermeiden.

--------------------------------------------------------------------------------
Was Versorgungssicherheit kostet

  Bei 3,000 EUR/MWh kostet die CVaR-Schranke NICHTS: Der
  risikoneutrale Plan haelt sie schon ein. Das ist kein Zufall - bei
  diesem Preis lohnt sich Vorhaltung bereits im Erwartungswert.

  Interessant wird die Schranke dort, wo der Schaden ZU NIEDRIG
  bepreist ist. Dieselbe Rechnung mit einem Marktpreisdeckel von
  300 EUR/MWh statt 3,000:

  Planungsgrundlage                    Kosten      Kosten Fehlmenge  Fehlmenge    Szen. mit
                                    im Mittel  schlimmst.    Mittel   CVaR 90%       Abwurf
  ------------------------------------------------------------------------------------
  risikoneutral, billiger Abwurf      350,512     421,336       3.0       18.4      8/40
  + CVaR 90 % der Fehlmenge = 0       351,356     415,924       0.0        0.0      0/40

  Jetzt greift die Schranke: Der risikoneutrale Plan nimmt in
  8 von 40 Szenarien einen Lastabwurf in Kauf, weil er bei
  300 EUR/MWh billiger ist als das Vorhalten eines Blocks.

  Aufpreis fuer die Sicherheit: 844 EUR je Tag
  Vermiedene Fehlmenge (CVaR 90 %): 18.4 MWh
  -> 46 EUR je vermiedener MWh

  Diese Zahl ist die Entscheidungsgrundlage - genau wie die Spalte
  'EUR je kg' im Kapitel Mehrziel. Sie sagt der Aufsicht, was ihre
  Versorgungssicherheitsvorgabe tatsaechlich kostet, statt darueber
  zu streiten, wie wichtig sie ist.

================================================================================
  WAS MAN DARAUS MITNIMMT
================================================================================
1. Die erste Stufe ist binaer und traege. Was am Vorabend nicht
   angefahren wurde, steht am naechsten Tag nicht zur Verfuegung -
   egal, wie hoch der Preis dann steigt.
2. Deshalb ist der Erwartungswert als Planungsgrundlage nicht nur
   ungenau, sondern SYSTEMATISCH zu knapp: Er plant fuer einen Tag,
   der so nie eintritt, und laesst keine Reserve fuer die Haelfte
```

```
   der Faelle, in denen es schlechter kommt.
3. Der Ausweg ist keine bessere Windprognose, sondern ein Modell,
   das die Szenarien SIEHT. Die Rechenzeit dafuer liegt hier im
   Sekundenbereich - der Aufwand ist die Modellierung, nicht der
   Solver.
4. Wo der Lastabwurf teuer genug bepreist ist, deckt schon die
   Minimierung der ERWARTETEN Kosten die Extremfaelle mit ab. Die
   Risikoschranke wird genau dann gebraucht, wenn der Schaden ZU
   NIEDRIG bepreist ist - was bei Versorgungssicherheit die Regel
   ist, weil ein Marktpreisdeckel nicht den volkswirtschaftlichen
   Schaden abbildet. Dann liefert sie den Preis der Vorgabe in Euro
   je vermiedener MWh, und darueber laesst sich verhandeln.
================================================================================
```

17.5 Der Befund

Planungsgrundlage	Kosten im Mittel	schlimmster Tag	Szenarien mit Lastabwurf
nur Wind-Erwartungswert	874 870 €	2 504 154 €	**28 von 40**
alle 40 Szenarien	**351 356 €**	**415 924 €**	**0 von 40**

Der Erwartungswert-Plan kostet in der Wirklichkeit **149 % mehr**, und sein schlimmster Tag ist sechsmal so teuer.

Wofür genau? Die Summendifferenz von 523 514 € sagt noch nicht, wo sie entsteht. Zerlegt man sie nach Kostenarten, wird aus dem Befund eine Aussage, die man vorlegen kann:

Drei Dinge stehen darin, die die Tabelle nicht zeigt. Die **Anfahrkosten sind in beiden Plänen gleich** (58 000 €) — der Erwartungswert-Plan spart also nicht dort, wo man es vermuten würde. Er spart **12 725 € Brennstoff**, weil er weniger Blockstunden vorhält. Und er bezahlt dafür **536 239 € Lastabwurf**: das **42-fache** seiner Ersparnis.

Genau diese Gegenüberstellung ist es, die eine Investitionsentscheidung trägt. „149 % mehr” ist eine Zahl, über die man streiten kann; „wir sparen 12 725 € und riskieren dafür 536 239 €” ist eine, über die man entscheidet.

> **Zum Diagramm** Es wird von `bilder_04/erzeuge_wirkung.py` erzeugt, das die Instanz aus `Kraftwerkseinsatz.py` **erneut rechnet** statt Zahlen zu übernehmen — und dabei die beiden abgedruckten Summen exakt reproduziert. Die Trennung von Brennstoff- und Abwurfkosten ist der einzige Zusatz; das Buchprogramm braucht sie nicht.

Der Unterschied zwischen den beiden Plänen ist dabei erstaunlich klein:

```
Erwartungswert   111111122222222211111111  (33 Blockstunden)
zweistufig       111111222222222222222111  (39 Blockstunden)
```

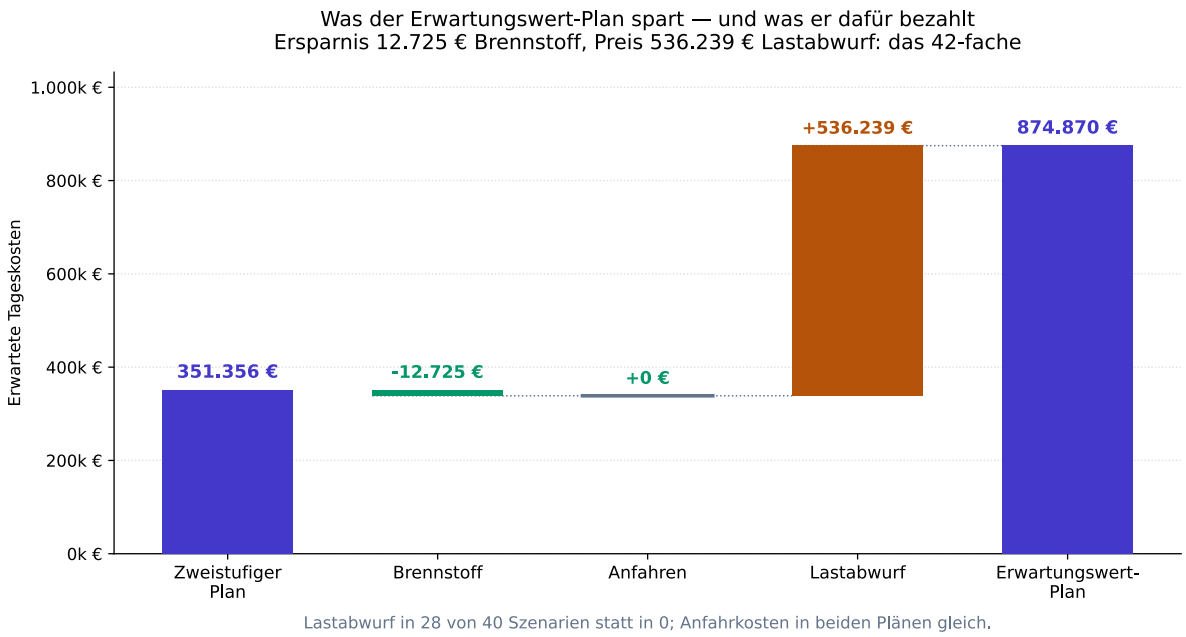

Abbildung 22: Abb. 17.1: Wofür der Erwartungswert-Plan 149 % mehr bezahlt

Sechs zusätzliche Blockstunden verhindern 28 Lastabwürfe. Der zweistufige Plan fährt den zweiten Block eine Stunde früher an und lässt ihn bis 21 Uhr statt bis 16 Uhr laufen.

> **Warum der Erwartungswert hier systematisch versagt**
>
> Der Fehler ist nicht, dass die Windprognose ungenau wäre. Er liegt in der **Trägheit der ersten Stufe**: Was am Vorabend nicht angefahren wurde, steht am nächsten Tag nicht zur Verfügung — egal, wie hoch der Preis dann steigt.
>
> Eine Planung auf den Mittelwert lässt in der Hälfte der Fälle zu wenig Reserve. Bei einer *anpassbaren* Größe wäre das halb so schlimm, weil man nachsteuern könnte. Bei einem Kernblock mit acht Stunden Mindestlaufzeit gibt es kein Nachsteuern.
>
> Das ist eine Verschärfung des „Fluchs des Durchschnitts" aus Abschnitt 12.3: Dort war der Mittelwert eine schlechte Näherung, hier ist er eine **nicht revidierbare** schlechte Näherung.

17.6 Was Versorgungssicherheit kostet

Der dritte Plan legt eine CVaR-Schranke auf die nicht gedeckte Energie: Im Mittel der schlechtesten 10 % der Szenarien darf **nichts** fehlen. Dieselbe Rockafellar-Uryasev-Konstruktion wie in Kapitel 20, nur dass hier Megawattstunden begrenzt werden statt Euro.

Das Ergebnis ist zunächst enttäuschend — und dann lehrreich:

Abwurfpreis	risikoneutral	mit CVaR-Schranke	Aufpreis
3 000 €/MWh	351 356 €, kein Abwurf	351 356 €, kein Abwurf	**0 €**

Abwurfpreis	risikoneutral	mit CVaR-Schranke	Aufpreis
300 €/MWh	350 512 €, Abwurf in 8/40	351 356 €, kein Abwurf	**844 €**

Bei 3 000 €/MWh kostet die Schranke **nichts**: Der risikoneutrale Plan hält sie längst ein. Bei diesem Preis lohnt sich das Vorhalten eines Blocks schon im Erwartungswert, lange bevor jemand über Risiko spricht.

Erst bei einem Marktpreisdeckel von 300 €/MWh greift sie. Dann nimmt der risikoneutrale Plan in 8 von 40 Szenarien einen Lastabwurf in Kauf, weil er billiger ist als Vorhaltung — und die Schranke kostet 844 € je Tag für 18,4 vermiedene MWh im CVaR:

$$\frac{844\ €}{18{,}4\ \text{MWh}} \approx 46\ € \text{ je vermiedener MWh}$$

 Merksatz Eine Risikoschranke ändert genau dann etwas, wenn der Schaden **zu niedrig bepreist** ist. Ist er realistisch bewertet, erledigt die Minimierung der erwarteten Kosten die Absicherung von selbst.

Das ist keine akademische Feinheit, sondern die reale Lage der Energiewirtschaft: Ein Marktpreisdeckel bildet den volkswirtschaftlichen Schaden eines Stromausfalls nicht ab. Genau deshalb gibt es regulatorische Vorgaben zur Versorgungssicherheit — und diese Zahl, 46 € je vermiedener MWh, ist das, worüber sich verhandeln lässt. Sie ist dasselbe Instrument wie die Spalte „€ je kg" in Abschnitt 14.8.

17.7 Übungsaufgaben

Lösungen: Abschnitt A.17.

Aufgabe 17.1 □ **— Der Umschlagpunkt.** Rechnen Sie den Umschlagpunkt aus dem Schnellstart für eine Leistung von 60 MW statt 120 MW nach. In welche Richtung verschiebt er sich, und warum ist das plausibel?

Aufgabe 17.2 □ **— Die fehlende Nebenbedingung.** Das Modell kennt Mindestlaufzeiten, aber keine **Mindeststillstandszeiten**. Formulieren Sie die entsprechende Bedingung. Welchen realen Sachverhalt bildet sie ab?

Aufgabe 17.3 □□ **— Wie viele Szenarien?** Rechnen Sie mit 10, 20, 40 und 80 Szenarien. Ab wann ändert sich der Commitment-Plan nicht mehr? Was folgt daraus für die Praxis — und was hat das mit Abschnitt 15.6 zu tun?

Aufgabe 17.4 □□ **— Der Wert eines Speichers.** Ergänzen Sie einen Batteriespeicher (100 MW, 400 MWh, Wirkungsgrad 90 %). Um wie viel sinken die erwarteten Kosten — und was ist der Speicher damit je MWh Kapazität wert?

Aufgabe 17.5 □□□ **— Wenn es größer wird.** Erhöhen Sie auf 20 Blöcke und 168 Stunden (eine Woche). Beobachten Sie MIP-Gap und Rechenzeit (Abschnitt 6.8). Ab wann wird das Problem für den exakten Solver zu groß — und welches Verfahren aus Kapitel 9 würden Sie einsetzen?

17.8 Finde den Denkfehler

□ „Wir rechnen mit dem P50-Szenario"

Ein Netzbetreiber beschreibt sein Vorgehen:

> *„Wir verwenden für die Tagesplanung das **P50-Szenario** der Windprognose — also den > Median. Die Hälfte der Tage fällt besser aus, die Hälfte schlechter, im Mittel gleicht > sich das aus. Für die Extremfälle haben wir zusätzlich eine Reserve von 5 % der Last > vorgehalten, das ist die branchenübliche Größenordnung."*

Das klingt sorgfältig, und beides ist gängige Praxis.

Beide Teile der Aussage enthalten einen Fehler. Welche?

Ein Hinweis zum zweiten Teil: Sehen Sie sich in der Ausgabe des Programms an, *welche* Größe der Plan verändert hat, um die Lastabwürfe zu vermeiden — und vergleichen Sie das mit dem, was eine prozentuale Leistungsreserve leisten kann.

17.9 Micro-Quiz

□ Drei Fragen

1. Warum ist die Merit-Order-Regel („billigstes Kraftwerk zuerst") für die Tagesplanung nicht ausreichend? a) Weil sie die Netzverluste ignoriert. b) Weil Anfahrkosten und Mindestlaufzeiten die Stunden miteinander koppeln — die Entscheidung einer Stunde bindet die folgenden. c) Weil Grenzkosten in der Praxis nicht bekannt sind.

2. Was unterscheidet dieses zweistufige Modell von dem in Kapitel 12? a) Es hat mehr Szenarien. b) Die erste Stufe ist **binär** und damit nicht anteilig anpassbar. c) Es minimiert den CVaR statt des Erwartungswerts.

3. Bei 3 000 €/MWh kostet die CVaR-Schranke nichts, bei 300 €/MWh dagegen 844 €/Tag. Warum? a) Weil der Solver bei kleineren Zahlen ungenauer rechnet. b) Weil bei realistisch hohem Abwurfpreis schon die Minimierung der erwarteten Kosten Vorhaltung erzwingt; die Schranke greift nur, wenn der Schaden zu niedrig bepreist ist. c) Weil die CVaR-Schranke bei hohen Preisen numerisch unwirksam wird.

17.10 Selbsttest

1. Erklären Sie in zwei Sätzen, warum ein Kraftwerkspark nicht wie ein Regal aufgefüllt werden kann.
2. Welche Größe ist in diesem Modell die „here and now"-Entscheidung, welche die „wait and see"-Entscheidung?
3. Warum ist eine Planung auf den Erwartungswert hier *systematisch* zu knapp und nicht nur ungenau?
4. Wann ändert eine Risikoschranke etwas an der Lösung — und wann nicht?
5. Sie sollen einer Regulierungsbehörde erklären, was ihre Versorgungssicherheitsvorgabe kostet. Welche Zahl nennen Sie, und wie ist sie entstanden?

17.11 Zusammenfassung

- **Anfahrkosten und Mindestlaufzeiten** machen aus 24 Stundenentscheidungen ein einziges Problem. Die Merit-Order gilt nur für einen einzelnen Zeitpunkt; im Schnellstart schlägt die fünfmal teurere Gasturbine den Kernblock bis knapp vier Stunden.
- Das Modell ist zweistufig mit **binärer erster Stufe**: Das An/Aus steht am Vorabend fest und gilt für alle Szenarien, die Fahrweise passt sich an. Das ist eine Verschärfung gegenüber Kapitel 12, wo die erste Stufe kontinuierlich war.
- Eine Planung auf den **Wind-Erwartungswert** kostet hier 149 % mehr und führt in 28 von 40 Szenarien zum Lastabwurf. Der Grund ist nicht die Prognosegüte, sondern die Trägheit: Was nicht angefahren wurde, ist nicht verfügbar.
- Der zweistufige Plan braucht dafür nur **sechs zusätzliche Blockstunden**.
- Eine **CVaR-Schranke** auf die nicht gedeckte Energie ändert nur dann etwas, wenn der Schaden zu niedrig bepreist ist. Bei realistischen 3 000 €/MWh erledigt die Minimierung der erwarteten Kosten die Absicherung von selbst.
- Wo sie greift, liefert sie den **Preis der Vorgabe**: 46 € je vermiedener MWh — eine Zahl, über die man verhandeln kann.

Kapitel 18: Finanzdaten-Modellierung — Renditen, Kovarianz und Shrinkage

Kapitel auf einen Blick

Worum geht es? Um die Eingangsdaten der Portfoliooptimierung — und darum, warum sie trügerisch sind. Wer historische Kovarianzen ungefiltert in einen Optimierer gibt, optimiert Rauschen.

Voraussetzungen: Kapitel 11 (Eigenwerte, positive Definitheit), Grundlagen der Statistik.

Danach können Sie: Kursdaten korrekt laden, diskrete und logarithmische Renditen unterscheiden, das Schätzfehlerproblem erklären, Ledoit-Wolf-Shrinkage anwenden — und erkennen, wann eine Kovarianzmatrix aus zu wenigen Beobachtungen stammt.

Zeitbedarf: ca. 5 Stunden.

Programme:
`Renditen_Vergleich.py`
`Schaetzrauschen_Demo.py`
`Finanzdaten_Ledoit_Wolf.py`
`Kovarianz_Falle.py`

 Überspringen Sie dieses Kapitel nicht. Wer direkt bei Markowitz (Kapitel 19) einsteigt, erhält absurde Portfoliogewichte und weiß nicht, warum.

Notebook: `Notebooks_04/finanzdaten.ipynb`

18.1 In 5 Minuten gelöst

 In 5 Minuten gelöst: Die durchschnittliche Rendite, die es nicht gibt

Eine Anlage wurde über vier Perioden beobachtet. Der Kurs stand bei 100, dann bei 50, 75, 150 und schließlich bei **120**.

```
import numpy as np

kurse = np.array([100.0, 50.0, 75.0, 150.0, 120.0])
diskret = kurse[1:] / kurse[:-1] - 1          # (neu - alt) / alt
log = np.log(kurse[1:] / kurse[:-1])          # ln(neu / alt)

print("diskret:", np.round(diskret * 100, 2), "%")
print("log    :", np.round(log * 100, 2), "%")
print(f"Mittelwert diskret {diskret.mean()*100:5.2f} % -> hochgerechnet "
      f"{100 * (1 + diskret.mean())**4:6.2f}")
print(f"Mittelwert log     {log.mean()*100:5.2f} % -> hochgerechnet "
      f"{100 * np.exp(log.mean() * 4):6.2f}")
```

Ausgabe:

```
diskret: [-50.  50. 100. -20.] %
log    : [-69.31  40.55  69.31 -22.31] %
Mittelwert diskret 20.00 % -> hochgerechnet 207.36
Mittelwert log      4.56 % -> hochgerechnet 120.00
```

Die durchschnittliche Rendite beträgt 20 % pro Periode — bei einer Anlage, die insgesamt 20 % gewonnen hat. Wer damit vier Perioden hochrechnet, landet bei 207,36 statt bei den tatsächlichen 120,00.

Der Grund steckt in der ersten Zeile: Ein Verlust von 50 % und ein Gewinn von 50 % heben sich **nicht** auf. Aus 100 werden 50, daraus 75 — nicht wieder 100. Diskrete Renditen darf man deshalb nicht mitteln und nicht addieren; sie multiplizieren sich.

Logarithmische Renditen dagegen **addieren** sich. Ihre Summe beträgt hier 18,23 %, und $100 \cdot e^{0{,}1823} = 120{,}00$ — exakt der beobachtete Endkurs.

	diskrete Rendite	logarithmische Rendite
Verknüpfung über die Zeit	multiplikativ: $\prod(1 + r_t)$	**additiv**: $\sum \ln(1 + r_t)$
Mitteln zulässig?	nein — führt systematisch zu hoch	ja
Verknüpfung über Anlagen	**additiv**: $\sum w_i r_i$	nein (Summe von Logs ≠ Log der Summe)
Wofür also	Portfoliogewichtung, Kennzahlen je Periode	Zeitreihenanalyse, Volatilität, Modelle

 Merksatz Beide Renditearten sind richtig — für **verschiedene Richtungen**. Diskrete Renditen addieren sich über die **Anlagen** eines Portfolios, logarithmische über die **Zeit**. Wer sie verwechselt, bekommt keine Fehlermeldung, sondern eine plausible falsche Zahl.

Warum funktioniert das? Weil der Logarithmus aus einem Produkt eine Summe macht: $\ln(a \cdot b) = \ln a + \ln b$. Genau diese Eigenschaft braucht man, wenn sich Renditen über die Zeit aufmultiplizieren. Der Preis dafür: Über mehrere Anlagen hinweg funktioniert es nicht mehr, denn der Logarithmus einer Summe ist nicht die Summe der Logarithmen. Der Rest dieses Kapitels behandelt die zweite große Fehlerquelle bei Finanzdaten — die **Kovarianzmatrix**, deren Schätzung noch weit trügerischer ist als die Rendite selbst.

18.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, wann man diskrete und wann logarithmische Renditen verwendet.
2. … beschreiben, warum die Stichproben-Kovarianzmatrix bei vielen Titeln unbrauchbar wird.
3. … den **Error-Maximizer-Effekt** an einem Experiment nachweisen.
4. … Ledoit-Wolf-Shrinkage anwenden und ihre Wirkung an Eigenwerten und Konditionszahl messen.
5. … Kursdaten so laden, dass die Spaltenreihenfolge garantiert stimmt.
6. … begründen, warum eine Schätzung aus $T < N$ Beobachtungen dem Optimierer risikofreie Richtungen vorgaukelt — und welche drei Gegenmittel es gibt.
7. … ein Portfolio **außerhalb** des Schätzzeitraums bewerten statt darin.

18.3 Diskrete und logarithmische Renditen

Sei $P_{i,t}$ der **bereinigte** Schlusskurs (*adjusted close*) von Titel i zum Zeitpunkt t. „Bereinigt“ heißt: um Dividenden und Aktiensplits korrigiert — sonst erscheint jede Dividendenzahlung als Kurssturz.

Diskrete Rendite

$$R_{i,t} = \frac{P_{i,t} - P_{i,t-1}}{P_{i,t-1}} = \frac{P_{i,t}}{P_{i,t-1}} - 1$$

Eigenschaft — über Titel additiv:

$$R_{p,t} = \sum_{i=1}^{n} w_i R_{i,t} = \mathbf{w}^\top \mathbf{R}_t$$

Logarithmische Rendite

$$r_{i,t} = \ln\left(\frac{P_{i,t}}{P_{i,t-1}}\right) = \ln P_{i,t} - \ln P_{i,t-1}$$

Eigenschaft — über die Zeit additiv:

$$r_{i,0\to T} = \sum_{t=1}^{T} r_{i,t}$$

 Die Merkregel Diskrete Renditen addieren sich über das Portfolio (mehrere Titel, ein Zeitpunkt) — deshalb braucht man sie für die **Allokation**. **Logarithmische Renditen addieren sich über die Zeit** (ein Titel, mehrere Perioden) — deshalb braucht man sie für **Zeitreihenanalyse und Volatilitätsschätzung**.

Man kann nicht beides gleichzeitig haben: Die gewichtete Summe von Log-Renditen ist **nicht** die Log-Rendite des Portfolios.

 Handrechnung 18.1: Der Unterschied in Zahlen

Ein Kurs steigt von 100 auf 150, dann fällt er zurück auf 100.

	Periode 1	Periode 2	Summe	Wahrheit
Diskret	+50,0 %	−33,3 %	+16,7 % □	0 %
Logarithmisch	+0,4055	−0,4055	0,0000 □	0 %

Die diskreten Renditen **suggerieren einen Gewinn von 16,7 %**, obwohl der Kurs exakt dort steht, wo er begann. Die Log-Renditen heben sich korrekt auf.

Deshalb ist die Aussage „im Mittel 5 % Rendite“ mehrdeutig. Das arithmetische Mittel diskreter Renditen überschätzt systematisch, was ein Anleger tatsächlich verdient hätte. Für Wachstumsaussagen nimmt man das **geometrische** Mittel — oder rechnet gleich in Log-Renditen.

```
#!/usr/bin/env python3

# Renditen_Vergleich.py
"""
Kapitel Finanzdaten: Diskrete vs. logarithmische Renditen.
Zeigt an einem simulierten Kursverlauf, welche Eigenschaft wo gilt -
und warum das arithmetische Mittel diskreter Renditen in die Irre fuehrt.
"""

import numpy as np
import pandas as pd

if __name__ == "__main__":
    kurse = np.array([100.0, 150.0, 100.0, 120.0, 90.0, 135.0])
    tage = [f"t{i}" for i in range(len(kurse))]

    diskret = kurse[1:] / kurse[:-1] - 1.0
    logarithmisch = np.log(kurse[1:] / kurse[:-1])

    print("=" * 74)
    print("  DISKRETE UND LOGARITHMISCHE RENDITEN IM VERGLEICH")
    print("=" * 74)
    tabelle = pd.DataFrame({
        "Periode": [f"{tage[i]} -> {tage[i+1]}" for i in range(len(diskret))],
        "Kurs von": kurse[:-1],
        "Kurs bis": kurse[1:],
        "diskret R": [f"{r*100:+7.2f} %" for r in diskret],
        "log r": [f"{r:+8.4f}" for r in logarithmisch],
    })
    print(tabelle.to_string(index=False))

    # --- Zeitliche Aggregation ----------------------------------------
    gesamt_wahr = kurse[-1] / kurse[0] - 1.0
    summe_log = logarithmisch.sum()
    aus_log_zurueck = np.exp(summe_log) - 1.0
    summe_diskret = diskret.sum()

    print("\n--- Aggregation ueber die Zeit ---")
    print(f"  Tatsaechliche Gesamtrendite:        {gesamt_wahr*100:+8.2f} %")
    print(f"  Summe der Log-Renditen -> exp()-1:  {aus_log_zurueck*100:+8.2f} %  "
          f"{'KORREKT' if abs(aus_log_zurueck - gesamt_wahr) < 1e-9 else 'falsch'}")
    print(f"  Summe der diskreten Renditen:       {summe_diskret*100:+8.2f} %  FALSCH")

    # --- Mittelwerte --------------------------------------------------
    arithmetisch = diskret.mean()
    geometrisch = np.prod(1 + diskret) ** (1 / len(diskret)) - 1
    aus_log = np.exp(logarithmisch.mean()) - 1
```

```
print("\n--- Welcher Mittelwert ist der richtige? ---")
print(f"  Arithmetisches Mittel (diskret):    {arithmetisch*100:+8.2f} %  "
      f"-> ueberschaetzt")
print(f"  Geometrisches Mittel:               {geometrisch*100:+8.2f} %  -> korrekt")
print(f"  exp(Mittel der Log-Renditen) - 1:   {aus_log*100:+8.2f} %  "
      f"-> identisch zum geometrischen")

# Probe: Endkapital mit dem jeweiligen Mittelwert hochgerechnet
n = len(diskret)
print(f"\n  Probe - Startkapital 100 EUR ueber {n} Perioden:")
print(f"    tatsaechlich:                     {kurse[-1]:8.2f} EUR")
print(f"    mit arithm. Mittel hochgerechnet: {100*(1+arithmetisch)**n:8.2f} EUR")
print(f"    mit geom.  Mittel hochgerechnet:  {100*(1+geometrisch)**n:8.2f} EUR")

print("\n--- Annualisierung (252 Handelstage) ---")
print(f"  Volatilitaet aus Log-Renditen: "
      f"{logarithmisch.std(ddof=1)*np.sqrt(252)*100:.2f} % p.a.")
print("  (Die Wurzel-Zeit-Regel gilt nur bei unabhaengigen Renditen -")
print("   fuer reale Maerkte ist sie eine Naeherung.)")
print("=" * 74)
```

18.4 Das Schätzfehler-Problem

Sei $\mathbf{X} \in \mathbb{R}^{T \times N}$ die Matrix zentrierter Renditen von N Titeln über T Handelstage. Die **Stichproben-Kovarianzmatrix** lautet:

$$\mathbf{S} = \frac{1}{T-1}\mathbf{X}^{\top}\mathbf{X}$$

Das Problem: $\mathbf{S}$ hat $\frac{N(N+1)}{2}$ zu schätzende Parameter, aber nur $T \cdot N$ Datenpunkte. Bei $N = 50$ Titeln sind das **1275 Parameter** — geschätzt aus einem Jahr Daten ($T = 252$).

Verhältnis	Folge
$T \gg N$	$\mathbf{S}$ ist brauchbar
$T \approx N$	Extrem instabil; Eigenwerte streuen weit
$T < N$	$\mathbf{S}$ ist **singulär** — nicht invertierbar, Rang $< N$

Die **Zufallsmatrizentheorie** (*Random Matrix Theory*, Marchenko-Pastur-Gesetz) zeigt: Die kleinsten Eigenwerte von $\mathbf{S}$ werden **systematisch unterschätzt**, die größten überschätzt. Und genau das ist fatal, denn:

> **Der Error-Maximizer-Effekt** Ein Risikominimierer sucht die Richtungen mit der **kleinsten** Varianz — also genau jene Eigenrichtungen, deren Eigenwerte am stärksten **nach un-**

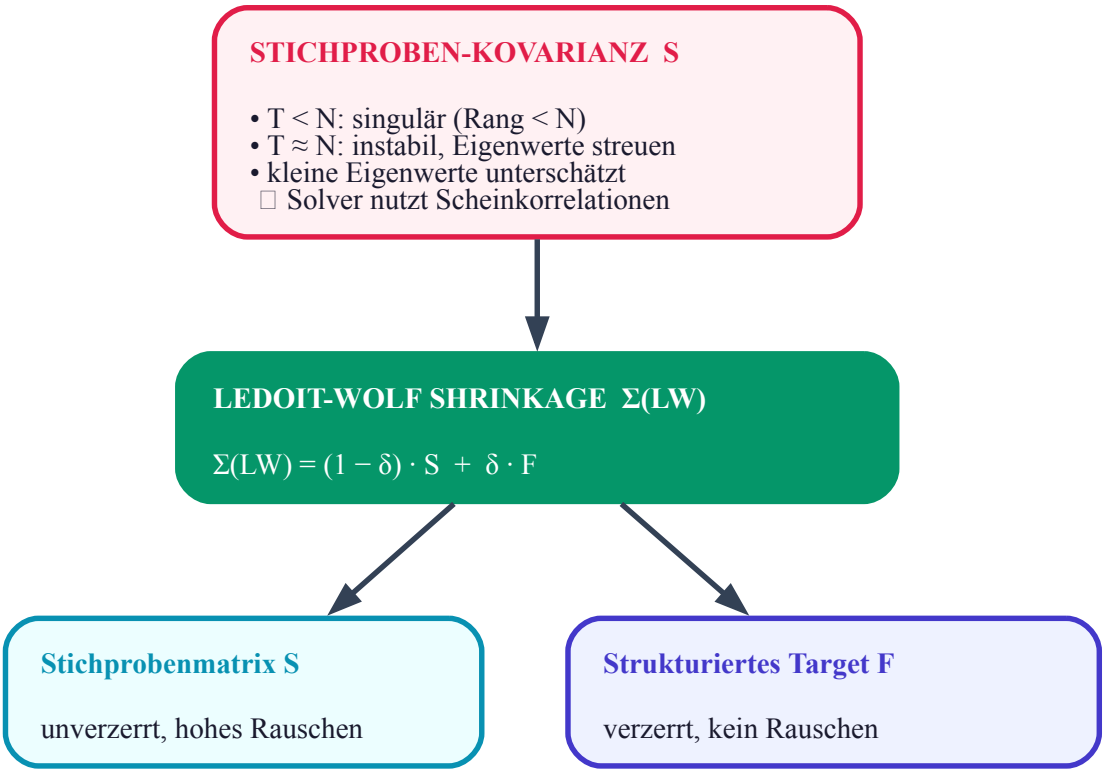

Abbildung 23: Abb. 18.1: Ledoit-Wolf-Kovarianz-Shrinkage

ten verzerrt sind. Die Optimierung greift damit zielsicher in das Schätzrauschen hinein. Markowitz-Optimierung ist deshalb kein Fehlerdämpfer, sondern ein **Fehlerverstärker**.

```
#!/usr/bin/env python3

# Schaetzrauschen_Demo.py
"""
Kapitel Finanzdaten: Der Error-Maximizer-Effekt im Experiment.

Aufbau: Wir KENNEN die wahre Kovarianzmatrix, weil wir die Daten selbst
erzeugen. Dann schaetzen wir sie aus endlich vielen Beobachtungen und
vergleichen, wie gut das daraus optimierte Portfolio in der WAHREN Welt
abschneidet - mit und ohne Shrinkage.

Das ist der einzige saubere Weg, Schaetzfehler zu messen: Man braucht eine
Wahrheit zum Vergleich, und die gibt es nur in der Simulation.
"""

import numpy as np
from sklearn.covariance import LedoitWolf


def erzeuge_wahre_kovarianz(n, rng):
    """Ein-Faktor-Modell: gemeinsamer Marktfaktor plus titelspezifisches Rauschen."""
    beta = rng.uniform(0.6, 1.4, size=n)             # Marktsensitivitaeten
```

```
    var_markt = 0.04                                    # Marktvarianz p.a.
    var_spezifisch = rng.uniform(0.01, 0.09, size=n)  # idiosynkratische Varianz
    return np.outer(beta, beta) * var_markt + np.diag(var_spezifisch)


def minimum_varianz_gewichte(sigma):
    """Analytische GMV-Loesung: w = Sigma^-1 1 / (1' Sigma^-1 1). Ohne Restriktionen."""
    eins = np.ones(len(sigma))
    try:
        loesung = np.linalg.solve(sigma, eins)
    except np.linalg.LinAlgError:
        loesung = np.linalg.pinv(sigma) @ eins        # falls singulaer
    return loesung / loesung.sum()


def portfoliovolatilitaet(w, sigma):
    return float(np.sqrt(w @ sigma @ w))


if __name__ == "__main__":
    rng = np.random.default_rng(2026)
    N = 40                                              # Anzahl Titel
    WIEDERHOLUNGEN = 200

    sigma_wahr = erzeuge_wahre_kovarianz(N, rng)
    w_ideal = minimum_varianz_gewichte(sigma_wahr)
    vola_ideal = portfoliovolatilitaet(w_ideal, sigma_wahr)

    print("=" * 92)
    print("  ERROR-MAXIMIZER: WIE TEUER IST SCHAETZRAUSCHEN?")
    print("=" * 92)
    print(f"Aufbau: {N} Titel, wahre Kovarianz bekannt (Ein-Faktor-Modell).")
    print(f"Bestmoegliche Volatilitaet bei perfektem Wissen: "
          f"{vola_ideal*100:.2f} % p.a.\n")

    print(f"{'T (Tage)':>9} | {'T/N':>5} | {'Stichprobe':>22} | {'Ledoit-Wolf':>22} | "
          f"{'Shrink':>7}")
    print(f"{'':>9} | {'':>5} | {'Vola     Aufschlag':>22} | "
          f"{'Vola     Aufschlag':>22} | {'delta':>7}")
    print("-" * 92)

    for T in [60, 120, 252, 504, 1260]:
        vola_stichprobe, vola_lw, deltas = [], [], []

        for _ in range(WIEDERHOLUNGEN):
            # Daten aus der WAHREN Verteilung ziehen (taeglich)
            daten = rng.multivariate_normal(np.zeros(N), sigma_wahr / 252, size=T)
```

```
            # (a) Stichproben-Kovarianz
            s_stich = np.cov(daten, rowvar=False, ddof=1) * 252
            w_stich = minimum_varianz_gewichte(s_stich)

            # (b) Ledoit-Wolf-Shrinkage
            lw = LedoitWolf().fit(daten)
            s_lw = lw.covariance_ * 252
            w_lw = minimum_varianz_gewichte(s_lw)
            deltas.append(lw.shrinkage_)

            # Bewertung IMMER mit der wahren Kovarianz - das ist der Punkt
            vola_stichprobe.append(portfoliovolatilitaet(w_stich, sigma_wahr))
            vola_lw.append(portfoliovolatilitaet(w_lw, sigma_wahr))

        m_stich, m_lw = np.mean(vola_stichprobe), np.mean(vola_lw)
        print(f"{T:>9} | {T/N:>5.1f} | {m_stich*100:>8.2f} % "
              f"{(m_stich/vola_ideal-1)*100:>+9.1f} % | "
              f"{m_lw*100:>8.2f} % {(m_lw/vola_ideal-1)*100:>+9.1f} % | "
              f"{np.mean(deltas):>7.3f}")

    print("-" * 92)
    print("'Aufschlag' = wie viel mehr Risiko das Portfolio in der WAHREN Welt traegt,")
    print("verglichen mit dem Portfolio bei perfektem Wissen.")
    print("Lesart: Je kleiner T/N, desto teurer das Schaetzrauschen - und desto mehr")
    print("schrumpft Ledoit-Wolf (delta steigt).")
    print("=" * 92)
```

Erwartete Ausgabe:

```
============================================================================================
  ERROR-MAXIMIZER: WIE TEUER IST SCHAETZRAUSCHEN?
============================================================================================
Aufbau: 40 Titel, wahre Kovarianz bekannt (Ein-Faktor-Modell).
Bestmoegliche Volatilitaet bei perfektem Wissen: 11.84 % p.a.

 T (Tage) |   T/N |             Stichprobe |            Ledoit-Wolf |  Shrink
          |       |     Vola     Aufschlag |     Vola     Aufschlag |   delta
--------------------------------------------------------------------------------------------
       60 |   1.5 |    20.78 %     +75.5 % |    15.44 %     +30.4 % |   0.093
      120 |   3.0 |    14.42 %     +21.8 % |    13.79 %     +16.5 % |   0.048
      252 |   6.3 |    12.86 %      +8.7 % |    12.76 %      +7.8 % |   0.024
      504 |  12.6 |    12.33 %      +4.1 % |    12.30 %      +3.9 % |   0.012
     1260 |  31.5 |    12.03 %      +1.6 % |    12.02 %      +1.6 % |   0.005
--------------------------------------------------------------------------------------------
```

Diese Tabelle ist das Kernargument des ganzen Kapitels. Lesen Sie die erste Zeile: Bei nur 60 Beobachtungen für 40 Titel trägt das „risikominimale“ Portfolio in Wahrheit **75,5 % mehr Risiko** als nötig.

Der Optimierer hat nicht das Risiko minimiert, sondern das **geschätzte** Risiko — und die Differenz ist der Schätzfehler, in den er hineinoptimiert hat.

Drei Beobachtungen:

1. **Der Schaden wächst dramatisch, je knapper die Daten.** Von $T/N = 31{,}5$ (+1,6 %) bis $T/N = 1{,}5$ (+75,5 %) ist es kein gleitender Übergang, sondern eine Explosion.
2. **Shrinkage halbiert den Schaden dort, wo er groß ist** (75,5 % → 30,4 %) und schadet dort nichts, wo er klein ist (1,6 % → 1,6 %). Sie ist damit praktisch risikolos einzusetzen.
3. δ **passt sich automatisch an:** viel Schrumpfung bei wenig Daten (0,093), fast keine bei vielen (0,005). Man muss nichts von Hand einstellen.

 Die Praxisregel Faustregel für Aktienportfolios: $T \geq 5N$ sollte man mindestens haben, besser $T \geq 10N$. Bei 50 Titeln sind das 250 bis 500 Handelstage — ein bis zwei Jahre. Wer weniger hat, sollte entweder das Universum verkleinern oder mit Shrinkage arbeiten. Am besten beides.

18.5 Ledoit-Wolf-Shrinkage

Olivier Ledoit und Michael Wolf lösten das Dilemma durch **lineare Schrumpfung**:

$$\Sigma_{\text{LW}} = (1 - \delta)\,\mathbf{S} + \delta\,\mathbf{F}$$

 Formel-Lesehilfe * **S** — die Stichprobenmatrix: **unverzerrt**, aber verrauscht. * **F** — das **Schrumpfungsziel**: verzerrt, aber stabil und rauschfrei. * $\delta \in [0, 1]$ — wie weit man vom Rauschen zur Struktur zieht.

Ohne Formel gesagt: Man mischt eine zappelige, aber im Mittel richtige Schätzung mit einer ruhigen, aber etwas schiefen. Die Mischung ist besser als beide Bestandteile einzeln — ein klassischer **Bias-Varianz-Kompromiss**. Ledoit und Wolf berechnen dabei dasjenige δ, das den erwarteten quadratischen Fehler zur unbekannten wahren Matrix **minimiert** — analytisch, ohne Kreuzvalidierung.

Welches Ziel F?

Ein häufiges Missverständnis: Es gibt nämlich **mehrere** Ledoit-Wolf-Varianten, und Text und Code müssen sich konsistent auf dieselbe beziehen:

Ziel **F**	Formel	Quelle	Wer benutzt es
Skalierte Einheitsmatrix	$\mathbf{F} = \frac{\text{tr}(\mathbf{S})}{N}\,\mathbf{I}$	Ledoit/Wolf 2004, *„A well-conditioned estimator“*	`sklearn.covariance.LedoitWolf`

Ziel **F**	Formel	Quelle	Wer benutzt es
Konstante Korrelation	$f_{ii} = s_{ii}$, $f_{ij} = \bar{r}\sqrt{s_{ii}s_{jj}}$	Ledoit/Wolf 2003, *„Honey, I shrunk the sample covariance matrix“*	oft in der Praxis, z. B. `PyPortfolioOpt`
Ein-Faktor-Modell	aus Marktbeta geschätzt	Ledoit/Wolf 2003	Faktormodell-Ansätze

Wichtig ist, Text und Code konsistent auf dieselbe Variante zu beziehen — zum Beispiel `sklearn.covariance.LedoitWolf`, das die skalierte Einheitsmatrix verwendet, versus eine manuelle Implementierung des Konstant-Korrelations-Ziels. Dieses Buch nennt beide und macht transparent, welches wo zum Einsatz kommt.

> **Welches Ziel ist besser?** Es kommt darauf an. Die **Einheitsmatrix** unterstellt, dass alle Titel gleich riskant und unkorreliert sind — eine starke, aber sehr stabile Annahme. Das **Konstant-Korrelations-Ziel** behält die geschätzten Einzelvarianzen und glättet nur die Korrelationen; das ist bei Aktien meist realistischer, weil Titel tatsächlich sehr unterschiedliche Volatilitäten haben. In der Aufgabe *Beide Shrinkage-Ziele vergleichen* (Abschnitt 18.7) implementieren Sie es selbst und vergleichen.

Das Ergebnis ist in jedem Fall garantiert **positiv definit**, wohlkonditioniert und stabil invertierbar.

18.6 Praxis: Datenpipeline mit korrekter Spaltenreihenfolge

> **Die wichtigste Zeile dieses Kapitels**
>
> `yfinance` liefert die Kursspalten **alphabetisch sortiert**, nicht in der Reihenfolge Ihrer Ticker-Liste:
>
> ```
> Ihre Liste:      ['AAPL', 'MSFT', 'NVDA', 'AMZN', 'JNJ', 'PFE', 'JPM', 'GS', 'XOM',
>  ↪  'CVX']
> prices.columns:  ['AAPL', 'AMZN', 'CVX', 'GS', 'JNJ', 'JPM', 'MSFT', 'NVDA', 'PFE',
>  ↪  'XOM']
> ```
>
> Wer danach mit Positionsindizes arbeitet, beschriftet Ergebnisse falsch und beschränkt falsche Sektoren. **Die Abhilfe ist eine Zeile:**
>
> ```python
> prices = raw["Close"][tickers].dropna()      # erzwingt die eigene Reihenfolge
> assert list(prices.columns) == tickers
> ```

```python
#!/usr/bin/env python3

# Finanzdaten_Ledoit_Wolf.py
"""
```

```
Kapitel Finanzdaten: Automatisierte Finanzdaten-Pipeline und Kovarianz-Shrinkage.

Eigenschaften:
  * Spaltenreihenfolge wird erzwungen und per assert geprueft
  * beide Shrinkage-Ziele werden berechnet und verglichen
  * Datenqualitaetspruefungen (Luecken, Ausreisser, Mindestlaenge)
  * Eigenwertspektrum wird ausgewiesen, nicht nur die Konditionszahl
"""

import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.covariance import LedoitWolf

HANDELSTAGE = 252


def lade_kurse(tickers: list[str], start, ende) -> pd.DataFrame:
    """Laedt bereinigte Schlusskurse und garantiert die Spaltenreihenfolge."""
    print(f"Lade {len(tickers)} Ticker von {start} bis {ende} ...")
    roh = yf.download(tickers, start=start, end=ende, auto_adjust=True, progress=False)
    if roh.empty:
        raise SystemExit("Download fehlgeschlagen (Netz, Ticker oder Rate-Limit pruefen).")

    if isinstance(roh.columns, pd.MultiIndex):
        # [tickers] erzwingt die gewuenschte Spaltenreihenfolge
        kurse = roh["Close"][tickers].dropna()
    else:
        kurse = roh[["Close"]].dropna()
        kurse.columns = tickers

    assert list(kurse.columns) == tickers, (
        f"Spaltenreihenfolge weicht ab!\n  erwartet: {tickers}\n"
        f"  erhalten: {list(kurse.columns)}")
    return kurse


def pruefe_datenqualitaet(kurse: pd.DataFrame, mindest_tage: int = 200) -> None:
    """Faengt die haeufigsten Datenprobleme ab, bevor sie ins Modell gelangen."""
    T, N = kurse.shape
    print(f"\nDatenqualitaet: {T} Handelstage, {N} Titel (T/N = {T/N:.1f})")
    if T < mindest_tage:
        raise SystemExit(f"Zu wenige Beobachtungen ({T} < {mindest_tage}).")
    if T < N:
        print("  WARNUNG: T < N - die Stichprobenkovarianz ist singulaer!")
    elif T < 3 * N:
        print("  WARNUNG: T < 3N - erhebliches Schaetzrauschen zu erwarten.")
```

```
    renditen = kurse.pct_change().dropna()
    extreme = (renditen.abs() > 0.25).sum().sum()
    if extreme:
        print(f"  Hinweis: {extreme} Tagesrenditen ueber 25 % "
              f"(Splits, Sondersituationen oder Datenfehler pruefen).")
    luecken = kurse.isna().sum().sum()
    print(f"  Fehlende Werte nach dropna: {luecken}")


def ziel_konstante_korrelation(renditen: np.ndarray) -> np.ndarray:
    """
    Shrinkage-Ziel nach Ledoit/Wolf 2003: Einzelvarianzen behalten,
    alle Korrelationen durch ihren Mittelwert ersetzen.
    (sklearn nutzt stattdessen die skalierte Einheitsmatrix - siehe oben.)
    """
    S = np.cov(renditen, rowvar=False, ddof=1)
    d = np.sqrt(np.diag(S))
    korrelation = S / np.outer(d, d)
    n = len(S)
    r_quer = (korrelation.sum() - n) / (n * (n - 1))       # Mittel ohne Diagonale
    F = r_quer * np.outer(d, d)
    np.fill_diagonal(F, np.diag(S))
    return F


def analysiere(kurse: pd.DataFrame):
    renditen = kurse.pct_change().dropna()
    T, N = renditen.shape

    mu = renditen.mean().values * HANDELSTAGE
    sigma_stichprobe = renditen.cov().values * HANDELSTAGE

    lw = LedoitWolf(assume_centered=False).fit(renditen.values)
    sigma_lw = lw.covariance_ * HANDELSTAGE
    delta = lw.shrinkage_

    print("\n" + "=" * 84)
    print("         FINANZDATEN-PIPELINE UND MATRIX-KONDITIONIERUNG")
    print("=" * 84)
    print(f"Beobachtungen T:            {T}")
    print(f"Titel N:                    {N}")
    print(f"Optimales Shrinkage delta:  {delta:.4f} "
          f"({delta*100:.1f} % Gewicht auf dem strukturierten Ziel)")

    eig_stich = np.linalg.eigvalsh(sigma_stichprobe)
    eig_lw = np.linalg.eigvalsh(sigma_lw)

    print("\n--- Eigenwertspektrum (annualisiert) ---")
```

```
    print(f"{'':<14} {'kleinster':>12} {'Median':>12} {'groesster':>12} "
          f"{'Kondition':>12}")
    for name, eig in [("Stichprobe", eig_stich), ("Ledoit-Wolf", eig_lw)]:
        kondition = eig.max() / max(eig.min(), 1e-12)
        print(f"{name:<14} {eig.min():>12.6f} {np.median(eig):>12.6f} "
              f"{eig.max():>12.6f} {kondition:>12.1f}")

    print("\nDie Konditionszahl misst, wie stark sich kleine Datenaenderungen auf")
    print("die Inverse auswirken. Je kleiner, desto stabiler die Portfoliogewichte.")

    # --- Vergleich der beiden Shrinkage-Ziele ----------------------------
    S_taeglich = np.cov(renditen.values, rowvar=False, ddof=1)
    F_identitaet = np.eye(N) * np.trace(S_taeglich) / N
    F_korrelation = ziel_konstante_korrelation(renditen.values)

    print("\n--- Die beiden Shrinkage-Ziele im Vergleich ---")
    for name, F in [("skalierte Einheitsmatrix (sklearn)", F_identitaet),
                    ("konstante Korrelation (LW 2003)", F_korrelation)]:
        gemischt = ((1 - delta) * S_taeglich + delta * F) * HANDELSTAGE
        eig = np.linalg.eigvalsh(gemischt)
        print(f"  {name:<36}: Kondition {eig.max()/max(eig.min(),1e-12):8.1f}, "
              f"kleinster EW {eig.min():.6f}")

    print("\n--- Erwartete Renditen (annualisiert) ---")
    uebersicht = pd.DataFrame({
        "Ticker": kurse.columns,
        "Rendite p.a.": [f"{r*100:+7.2f} %" for r in mu],
        "Volatilitaet p.a.": [f"{np.sqrt(sigma_lw[i, i])*100:6.2f} %" for i in range(N)],
    })
    print(uebersicht.to_string(index=False))
    print("=" * 84)
    return renditen, mu, sigma_stichprobe, sigma_lw


if __name__ == "__main__":
    UNIVERSUM = ["AAPL", "MSFT", "NVDA", "AMZN",   # Technologie
                 "JNJ", "PFE",                      # Gesundheit
                 "JPM", "GS",                       # Banken
                 "XOM", "CVX"]                      # Energie

    # Relatives 2-Jahres-Fenster, damit das Beispiel nicht "altert"
    ende = pd.Timestamp.today().normalize()
    start = ende - pd.DateOffset(years=2)

    kurse = lade_kurse(UNIVERSUM, start.strftime("%Y-%m-%d"), ende.strftime("%Y-%m-%d"))
    pruefe_datenqualitaet(kurse)
    analysiere(kurse)
```

 Zur Reproduzierbarkeit Dieses Programm lädt **Live-Daten**. Ihre Zahlen werden von den hier abgedruckten abweichen — je nach Abrufdatum. Das ist beabsichtigt und selbst eine Lektion: Wenn sich das optimale Portfolio zwischen zwei Abrufen deutlich ändert, obwohl nur ein paar Tage vergangen sind, sehen Sie den Schätzfehler direkt. Prüfen Sie das ruhig einmal.

18.7 Übungsaufgaben

Lösungen: Abschnitt A.18.

Aufgabe 18.1 □ **— Renditeart wählen.** Welche Renditeart nehmen Sie und warum? (a) Portfoliorendite aus Einzelrenditen berechnen. (b) Volatilität über 5 Jahre schätzen. (c) Kumulierte Wertentwicklung über 10 Jahre darstellen. (d) Korrelationen zwischen Titeln schätzen.

Aufgabe 18.2 □ **— Mittelwertfalle.** Ein Fonds meldet Jahresrenditen von $+60\,\%$, $-40\,\%$, $+60\,\%$, $-40\,\%$. (a) Wie hoch ist das arithmetische Mittel? (b) Wie viel Kapital hat ein Anleger nach 4 Jahren aus 10 000 €? (c) Welche jährliche Rendite entspricht dem tatsächlich?

Aufgabe 18.3 □□ **— Konditionszahl verstehen.** Erzeugen Sie eine Kovarianzmatrix für $N = 30$ Titel aus $T = 40$, $T = 100$ und $T = 1000$ simulierten Beobachtungen. Berechnen Sie jeweils kleinsten Eigenwert und Konditionszahl. Ab welchem T wird die Matrix brauchbar?

Aufgabe 18.4 □□ **— Spaltenreihenfolge prüfen.** Laden Sie fünf Ticker in einer bewusst unsortierten Reihenfolge. Zeigen Sie, dass `raw["Close"].columns` abweicht, und dass `raw["Close"][tickers]` das behebt. Was passiert, wenn ein Ticker nicht existiert?

Aufgabe 18.5 □□□ **— Beide Shrinkage-Ziele vergleichen.** Implementieren Sie das Konstant-Korrelations-Ziel vollständig (inklusive eigener Berechnung von δ nach Ledoit/Wolf 2003 oder per Kreuzvalidierung). Vergleichen Sie mit `sklearn` anhand: (a) der Konditionszahl, (b) der Out-of-Sample-Volatilität eines GMV-Portfolios (nach dem Muster von `Schaetzrauschen_Demo.py`), (c) der Stabilität der Gewichte über rollierende Fenster.

Aufgabe 18.6 □□□ **— Error-Maximizer selbst messen.** Erweitern Sie `Schaetzrauschen_Demo.py` um eine dritte Variante: die **Gleichgewichtung** ($w_i = 1/N$), die überhaupt keine Schätzung benötigt. (a) Ab welchem T/N schlägt die Stichproben-Optimierung die Gleichgewichtung? (b) Ab welchem T/N schlägt Ledoit-Wolf sie? (c) Was folgt daraus für die Praxis?

18.8 Finde den Denkfehler

Finde den Denkfehler: Das risikofreie Portfolio

Ein Analyst stellt ein Portfolio aus **30 Titeln** zusammen. Für die Kovarianzmatrix nimmt er das letzte Quartal — bei Wochendaten sind das **20 Beobachtungen**. Dann minimiert er die Varianz.

Das Ergebnis begeistert ihn: Das Portfolio hat ein Risiko von **0,000 %**. Er hat ein risikofreies Aktienportfolio gefunden.

Ihre Aufgabe: (a) Wie hoch ist der Rang einer Kovarianzmatrix, die aus T Beobachtungen für $N > T$ Anlagen geschätzt wurde — und was folgt daraus geometrisch? (b) Warum findet ausgerechnet ein **Minimum**-Varianz-Optimierer diese Stelle so zuverlässig? (c) Sehen Sie sich unten die Spalte „Hebel“ an. Was sagt ein Wert von 3,8 über das Portfolio aus? (d) Warum schlägt die simple Gleichgewichtung den Optimierer — und was folgt daraus für Ihre eigenen Modelle, auch außerhalb der Finanzwelt?

Auflösung: *Abschnitt A.18*.

Das folgende Programm führt den Fall vor. Es arbeitet mit **synthetischen** Daten, damit das Ergebnis reproduzierbar ist und kein Internetzugang nötig wird — mit echten Kursdaten sieht es genauso aus.

```
#!/usr/bin/env python3

# Kovarianz_Falle.py
"""
Kapitel Finanzdaten: Warum die historische Kovarianzmatrix den Optimierer belaeugt.

Ein Portfolio aus 30 Anlagen soll das Risiko minimieren. Geschaetzt wird die
Kovarianzmatrix aus 20 Beobachtungen - in der Praxis ein voellig ueblicher
Fall (30 Titel, ein Quartal Wochendaten).

Das Ergebnis ist ein Portfolio mit einem gemessenen Risiko von 0,000 %. Der
Optimierer haelt es fuer risikofrei. Ausserhalb des Schaetzzeitraums ist es
das schlechteste der drei untersuchten Portfolios - schlechter sogar als
blosse Gleichgewichtung.

Der Grund ist nicht Zufall, sondern Struktur: Bei N Anlagen und T < N
Beobachtungen hat die Stichproben-Kovarianzmatrix hoechstens Rang T-1. Es gibt
dann ganze Richtungen im Gewichtsraum, in denen sie exakt null Varianz misst -
Richtungen, die es in Wirklichkeit nicht gibt. Der Optimierer findet sie
zuverlaessig, denn er sucht ja genau danach.

Die Daten sind synthetisch (ein Marktfaktor plus Eigenrauschen), damit das
Ergebnis reproduzierbar ist und kein Internetzugang noetig wird. Mit echten
Kursdaten sieht es genauso aus.

Benoetigt: numpy, cvxpy, scikit-learn
"""
```

```
from __future__ import annotations

import numpy as np
import cvxpy as cp
from sklearn.covariance import LedoitWolf

ANZAHL_ANLAGEN = 30
BEOBACHTUNGEN_SCHAETZUNG = 20        # weniger als Anlagen - genau darum geht es
BEOBACHTUNGEN_TEST = 2000            # der "spaetere Verlauf"

RNG = np.random.default_rng(11)
# Jede Anlage reagiert unterschiedlich stark auf den Gesamtmarkt.
BETA = RNG.uniform(0.6, 1.4, ANZAHL_ANLAGEN)


def erzeuge_renditen(perioden: int) -> np.ndarray:
    """Ein Marktfaktor, auf den alle reagieren, plus Eigenrauschen je Anlage."""
    markt = RNG.normal(0.0, 0.010, perioden)
    eigen = RNG.normal(0.0, 0.012, (perioden, ANZAHL_ANLAGEN))
    return np.outer(markt, BETA) + eigen


def minimales_risiko(kovarianz: np.ndarray) -> tuple[np.ndarray, float]:
    """Minimum-Varianz-Portfolio: Gewichte summieren sich zu 1, sonst frei.

    cp.psd_wrap() sagt CVXPY: 'Ich weiss, dass diese Matrix numerisch nicht
    exakt positiv semidefinit ist - rechne trotzdem.' Das ist hier bewusst so
    gewaehlt, denn genau diese Eigenschaft wollen wir vorfuehren. Im
    Produktivbetrieb waere die DCP-Fehlermeldung das Warnsignal, dem man
    nachgehen muss (siehe den Denkfehler im Kapitel Quadratische und
    nichtlineare Optimierung).
    """
    gewichte = cp.Variable(ANZAHL_ANLAGEN)
    problem = cp.Problem(cp.Minimize(cp.quad_form(gewichte, cp.psd_wrap(kovarianz))),
                         [cp.sum(gewichte) == 1])
    problem.solve()
    return np.array(gewichte.value).ravel(), max(problem.value, 0.0)


if __name__ == "__main__":
    schaetzzeitraum = erzeuge_renditen(BEOBACHTUNGEN_SCHAETZUNG)
    spaeterer_verlauf = erzeuge_renditen(BEOBACHTUNGEN_TEST)

    stichprobe = np.cov(schaetzzeitraum, rowvar=False)
    ledoit_wolf = LedoitWolf().fit(schaetzzeitraum)

    print("=" * 78)
```

```
print("  DIE KOVARIANZ-FALLE")
print("=" * 78)
print(f"{ANZAHL_ANLAGEN} Anlagen, geschaetzt aus {BEOBACHTUNGEN_SCHAETZUNG} "
      f"Beobachtungen.")
print(f"Rang der Stichprobenmatrix: "
      f"{np.linalg.matrix_rank(stichprobe)} - noetig waeren {ANZAHL_ANLAGEN}.")
print(f"Kleinster Eigenwert: {np.linalg.eigvalsh(stichprobe).min():.2e}")
print()
print("Es gibt also Richtungen, in denen diese Matrix EXAKT null Varianz")
print("misst. Genau dort sucht ein Minimum-Varianz-Optimierer.")
print()

print(f"{'Schaetzer':<26} {'Risiko im Zeitraum':>19} "
      f"{'Risiko spaeter':>16} {'Hebel':>7}")
print("-" * 78)

ergebnisse = {}
for name, matrix in [
        ("Stichprobe (roh)", stichprobe),
        (f"Ledoit-Wolf (d={ledoit_wolf.shrinkage_:.2f})", ledoit_wolf.covariance_)]:
    gewichte, varianz_intern = minimales_risiko(matrix)
    risiko_spaeter = float(np.std(spaeterer_verlauf @ gewichte))
    hebel = float(np.abs(gewichte).sum())
    ergebnisse[name] = (risiko_spaeter, gewichte)
    print(f"{name:<26} {np.sqrt(varianz_intern) * 100:>18.3f} % "
          f"{risiko_spaeter * 100:>15.3f} % {hebel:>7.1f}")

gleich = np.full(ANZAHL_ANLAGEN, 1.0 / ANZAHL_ANLAGEN)
print(f"{'Gleichgewichtung 1/N':<26} "
      f"{float(np.std(schaetzzeitraum @ gleich)) * 100:>18.3f} % "
      f"{float(np.std(spaeterer_verlauf @ gleich)) * 100:>15.3f} % {1.0:>7.1f}")
print("-" * 78)

print("\nDrei Beobachtungen:")
print("1. Die rohe Schaetzung meldet 0,000 % Risiko - und liefert spaeter das")
print("   SCHLECHTESTE Ergebnis der drei. Sie hat kein Portfolio optimiert,")
print("   sondern eine Luecke in den eigenen Daten gefunden.")
print("2. Die Gleichgewichtung, die gar nicht optimiert, schlaegt den rohen")
print("   Optimierer deutlich. Das ist kein Zufallsbefund, sondern in der")
print("   Literatur breit belegt.")
print("3. Ledoit-Wolf ist im Schaetzzeitraum EHRLICHER (0,441 statt 0,000 %)")
print("   und spaeter am besten. Der Hebel faellt von 3,8 auf 1,8 - das")
print("   Portfolio wird auch praktisch handelbarer.")
print()
print("Merksatz: Ein Optimierer glaubt seinen Eingabedaten vollstaendig.")
print("Je mehr Freiheit Sie ihm geben, desto gruendlicher findet er deren")
print("Fehler. Das gilt fuer Kovarianzmatrizen wie fuer Lieferzeiten,")
```

```
    print("Ausfallraten und jede andere geschaetzte Groesse.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  DIE KOVARIANZ-FALLE
==============================================================================
30 Anlagen, geschaetzt aus 20 Beobachtungen.
Rang der Stichprobenmatrix: 19 - noetig waeren 30.
Kleinster Eigenwert: -6.47e-20

Es gibt also Richtungen, in denen diese Matrix EXAKT null Varianz
misst. Genau dort sucht ein Minimum-Varianz-Optimierer.

Schaetzer                   Risiko im Zeitraum   Risiko spaeter   Hebel
------------------------------------------------------------------------------
Stichprobe (roh)                         0.000 %          1.287 %     3.8
Ledoit-Wolf (d=0.34)                     0.441 %          0.974 %     1.8
Gleichgewichtung 1/N                     0.808 %          1.026 %     1.0
------------------------------------------------------------------------------

Drei Beobachtungen:
1. Die rohe Schaetzung meldet 0,000 % Risiko - und liefert spaeter das
   SCHLECHTESTE Ergebnis der drei. Sie hat kein Portfolio optimiert,
   sondern eine Luecke in den eigenen Daten gefunden.
2. Die Gleichgewichtung, die gar nicht optimiert, schlaegt den rohen
   Optimierer deutlich. Das ist kein Zufallsbefund, sondern in der
   Literatur breit belegt.
3. Ledoit-Wolf ist im Schaetzzeitraum EHRLICHER (0,441 statt 0,000 %)
   und spaeter am besten. Der Hebel faellt von 3,8 auf 1,8 - das
   Portfolio wird auch praktisch handelbarer.

Merksatz: Ein Optimierer glaubt seinen Eingabedaten vollstaendig.
Je mehr Freiheit Sie ihm geben, desto gruendlicher findet er deren
Fehler. Das gilt fuer Kovarianzmatrizen wie fuer Lieferzeiten,
Ausfallraten und jede andere geschaetzte Groesse.
==============================================================================
```

□ Formel-Übersetzer: warum $N > T$ alles kaputt macht

Mathematik	Alltagssprache
$\mathbf{S} = \frac{1}{T-1}\sum_t (\mathbf{r}_t - \bar{\mathbf{r}})(\mathbf{r}_t - \bar{\mathbf{r}})^\top$	„Wie stark schwanken die Anlagen — und wie sehr gemeinsam?“
$\operatorname{rang}(\mathbf{S}) \le T - 1$	„Aus 20 Beobachtungen lassen sich höchstens 19 unabhängige Richtungen ablesen.“

Mathematik	Alltagssprache
$N = 30 > 19$	„Es bleiben 11 Richtungen übrig, über die die Daten **nichts** sagen.“
$\exists\, \mathbf{w} e 0 \,:\, \mathbf{w}^{o} p \mathbf{S} \mathbf{w} = 0$	„In genau diesen Richtungen misst die Matrix null Risiko — nicht weil keines da ist, sondern weil sie blind ist.“
$\min_{\mathbf{w}} \mathbf{w}^{o} p \mathbf{S} \mathbf{w}$	„Und der Optimierer sucht zielsicher genau dort.“

Die Faustregel: Für eine halbwegs verlässliche Kovarianzschätzung braucht man $T \gtrsim 10 \cdot N$ Beobachtungen. Bei 30 Titeln also rund 300 Perioden — mehr als ein Jahr Tagesdaten. Wer weniger hat, **muss** schrumpfen.

18.9 Micro-Quiz

Micro-Quiz 18: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Ein Kurs fällt um 50 % und steigt danach um 50 %. Wo steht er? (a) Wieder beim Ausgangswert — die Prozente heben sich auf. (b) Bei 75 % des Ausgangswerts. Diskrete Renditen verknüpfen sich multiplikativ, nicht additiv: $1{,}0 \cdot 0{,}5 \cdot 1{,}5 = 0{,}75$. (c) Bei 25 % des Ausgangswerts.

2. Wofür nimmt man logarithmische Renditen, wofür diskrete? (a) Logarithmische sind grundsätzlich genauer und daher immer vorzuziehen. (b) Logarithmische addieren sich über die **Zeit**, diskrete über die **Anlagen** eines Portfolios. Man wählt nach der Richtung, in der summiert wird. (c) Diskrete für kurze, logarithmische für lange Zeiträume.

3. Sie schätzen eine Kovarianzmatrix für 50 Anlagen aus 40 Wochenrenditen. Was ist zu tun? (a) Nichts — 40 Beobachtungen sind eine ordentliche Stichprobe. (b) Die Matrix ist singulär und gaukelt dem Optimierer risikofreie Richtungen vor. Entweder mehr Daten beschaffen oder einen Shrinkage-Schätzer verwenden, der positive Definitheit garantiert. (c) Die Anzahl der Anlagen auf 40 reduzieren, dann passt es.

18.10 Selbsttest

Antworten: Anhang A.

1. Warum addieren sich Log-Renditen über die Zeit, diskrete aber über das Portfolio?
2. Was besagt der Error-Maximizer-Effekt?
3. Warum ist $\mathbf{S}$ bei $T < N$ singulär, und was folgt daraus praktisch?
4. Was mischt Ledoit-Wolf, und wie wird δ bestimmt?
5. Welche zwei Shrinkage-Ziele gibt es, und welches verwendet `scikit-learn`?

18.11 Zusammenfassung

- **Diskret über das Portfolio, logarithmisch über die Zeit** — die Wahl der Renditeart ist keine Geschmacksfrage.
- **Das arithmetische Mittel diskreter Renditen überschätzt** die tatsächliche Wertentwicklung systematisch.
- **Die Stichprobenkovarianz ist bei $T \approx N$ unbrauchbar** und bei $T < N$ singulär.
- **Markowitz verstärkt Schätzfehler**, statt sie auszugleichen — der Error-Maximizer-Effekt.
- **Shrinkage** mischt Rauschen mit Struktur und liefert eine stabil invertierbare Matrix. Es gibt mehrere Ziele; `sklearn` verwendet die skalierte Einheitsmatrix.
- **Erzwingen Sie die Spaltenreihenfolge** nach jedem Datenimport und prüfen Sie sie per `assert`.
- **Faustregel für die Datenmenge:** $T \gtrsim 10 \cdot N$. Bei 30 Titeln also rund 300 Perioden. Wer weniger hat, muss schrumpfen — oder die Freiheit des Optimierers begrenzen.
- **Bewerten Sie außerhalb des Schätzzeitraums.** Ein Portfolio, das im eigenen Datenfenster 0,000 % Risiko meldet, ist kein gutes Portfolio, sondern ein Beleg dafür, dass die Daten nicht ausreichen.
- **Das gilt weit über Finanzdaten hinaus.** Bearbeitungszeiten, Ausfallraten, Nachfrageprognosen: Ein Optimierer sucht nicht die beste Lösung, sondern den größten Fehler in Ihren Schätzungen.

Ausblick. Mit stabilen Schätzungen können wir in Kapitel 19 endlich optimieren: die Markowitz-Effizienzgrenze mit realistischen institutionellen Restriktionen.

Kapitel 19: Die moderne Portfoliotheorie nach Markowitz

Kapitel auf einen Blick

Worum geht es? Um das Modell, das 1952 die quantitative Finanzwirtschaft begründete — und um die Restriktionen, ohne die es in der Praxis unbrauchbar bleibt.

Voraussetzungen: Kapitel 11 (QP, KKT), Kapitel 18 (Kovarianz, Shrinkage).

Danach können Sie: Die Effizienzgrenze berechnen, das Maximum-Sharpe-Portfolio über die Korn-Transformation bestimmen, institutionelle Nebenbedingungen einbauen — und beurteilen, wann eine Renditeschätzung das Ergebnis wertlos macht.

Zeitbedarf: ca. 6 Stunden.

Programme:
`Diversifikation_Demo.py`
`Markowitz_CVXPY.py`
`Renditeschaetzung_Falle.py`

Notebook: `Notebooks_04/markowitz.ipynb`

19.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Die Effizienzlinie in acht Zeilen

Ein Anleger verteilt sein Vermögen auf drei Anlageklassen:

	erwartete Rendite	Schwankung
Anleihen	3,0 %	5 %
Aktien Welt	8,0 %	18 %
Immobilien	5,5 %	11 %

Statt *einer* Antwort rechnen wir gleich die **ganze Auswahl** durch — für jede Zielrendite das Portfolio mit dem geringsten Risiko:

```
import numpy as np
import cvxpy as cp

mu = np.array([0.03, 0.08, 0.055])                    # erwartete Rendite
sig = np.array([0.05, 0.18, 0.11])                    # Schwankung
R = np.array([[1.0, 0.1, 0.2], [0.1, 1.0, 0.5], [0.2, 0.5, 1.0]])
S = np.outer(sig, sig) * R                            # Kovarianzmatrix

w, ziel = cp.Variable(3, nonneg=True), cp.Parameter(nonneg=True)
problem = cp.Problem(cp.Minimize(cp.quad_form(w, S)),
                     [cp.sum(w) == 1, mu @ w >= ziel])

for z in [0.03, 0.04, 0.05, 0.06, 0.07, 0.08]:
    ziel.value = z
    problem.solve()
    risiko = np.sqrt(problem.value)
    print(f"{z*100:5.1f}% {risiko*100:7.2f}% {(mu @ w.value)/risiko:8.2f}   "
          f"{np.round(w.value, 3)}")
```

Ausgabe (Zielrendite, Risiko, Rendite-je-Risiko, Gewichte):

```
  3.0%    4.83%     0.69   [0.884 0.017 0.098]
  4.0%    5.35%     0.75   [0.711 0.111 0.178]
  5.0%    7.49%     0.67   [0.451 0.251 0.298]
  6.0%   10.35%     0.58   [0.191 0.391 0.418]
```

```
7.0%   13.55%     0.52   [0.    0.6   0.4  ]
8.0%   18.00%     0.44   [0.    1.    0.   ]
```

Diese sechs Zeilen sind die Effizienzlinie, und sie beantworten die Frage anders, als die meisten sie stellen. Es gibt kein „bestes“ Portfolio — es gibt eine **Auswahl**, und jede Zeile ist für ihre Zielrendite optimal.

Lesen Sie die dritte Spalte: Sie zeigt, wie viel Rendite je Einheit Risiko heraussprungt (die **Sharpe-Kennzahl**). Sie ist **nicht** unten am höchsten:

- Bei 3 % Zielrendite: 0,69 — das Portfolio ist sicher, aber lässt Ertrag liegen.
- Bei **4 %**: **0,75** — das beste Verhältnis.
- Bei 8 %: 0,44 — das renditestärkste Portfolio ist das **schlechteste** je Risikoeinheit. Und es besteht zu 100 % aus Aktien: Wer die höchste Rendite fordert, bekommt zwangsläufig gar keine Diversifikation mehr.

Von 3 % auf 8 % Rendite ist ein Zuwachs um Faktor 2,7 — das Risiko steigt dabei um Faktor 3,7. Die letzten Prozentpunkte Rendite sind die teuersten.

> **Merksatz** Ein Optimierer beantwortet nicht die Frage „was soll ich tun?“, sondern „was ist erreichbar?“. Die Effizienzlinie ist eine **Speisekarte**, keine Empfehlung. Welche Zeile die richtige ist, hängt von der Risikotragfähigkeit ab — und das ist keine mathematische, sondern eine unternehmerische Frage.

Warum funktioniert das? Weil `cp.Parameter` das Modell nur **einmal** aufbaut und dann sechsmal mit verschiedenen Zielwerten löst. Genau dafür gibt es Parameter in CVXPY: Bei 200 Punkten auf der Linie spart das den 200-fachen Kompilierungsaufwand (Abschnitt 3.8). Der Rest dieses Kapitels führt die Effizienzlinie sauber aus, zeigt, wie man das beste Verhältnis direkt berechnet statt es abzulesen — und warum das Ganze mit geschätzten Renditen gefährlicher ist, als es hier aussieht.

19.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, warum Diversifikation Risiko senkt, ohne Rendite zu kosten.
2. … das Mean-Variance-Modell aufstellen und den Risikoparameter λ deuten.
3. … die Effizienzgrenze berechnen und korrekt begrenzen.
4. … die **Korn-Transformation** anwenden, um die nicht-konvexe Sharpe-Ratio-Maximierung in ein QP zu überführen.
5. … Positions-, Sektor- und Kardinalitätsgrenzen einbauen — **mit Namen statt Indizes**.
6. … die Effizienzlinie als Auswahl lesen statt als Empfehlung.
7. … begründen, warum das Minimum-Varianz-Portfolio in der Praxis oft robuster ist als das Maximum-Sharpe-Portfolio.

19.3 Warum Diversifikation funktioniert

Harry Markowitz' zentrale Einsicht von 1952: **Das Risiko eines Portfolios ist nicht die Summe der Einzelrisiken.** Es hängt davon ab, wie die Titel zusammenspielen.

Für zwei Titel gilt:

$$\sigma_p^2 = w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + 2w_1w_2\,\rho_{12}\,\sigma_1\sigma_2$$

 Formel-Lesehilfe * Die ersten beiden Terme sind die „eigenen" Risikobeiträge. * Der dritte Term enthält die **Korrelation** ρ_{12} — und **nur er** kann negativ werden.

Ohne Formel gesagt: Wenn zwei Anlagen sich gegenläufig bewegen ($\rho < 0$), gleichen sich ihre Schwankungen teilweise aus. Das Portfolio schwankt dann **weniger als jeder Einzeltitel** — bei einer Rendite, die genau dem gewichteten Mittel entspricht. Diversifikation ist damit das einzige „Gratis-Mittagessen" der Finanzwirtschaft.

 Handrechnung 19.1: Das Gratis-Mittagessen in Zahlen

Zwei Titel, beide mit $\mu = 8\,\%$ und $\sigma = 20\,\%$, Gewichtung 50/50. Die Portfoliorendite ist **immer** 8 % — unabhängig von der Korrelation. Das Risiko dagegen:

$$\sigma_p = \sqrt{0{,}25 \cdot 0{,}04 + 0{,}25 \cdot 0{,}04 + 2 \cdot 0{,}25 \cdot \rho \cdot 0{,}04} = 0{,}2\sqrt{0{,}5(1+\rho)}$$

ρ	σ_p	Risikoreduktion
+1,0	20,0 %	0 % (keine Diversifikation)
+0,5	17,3 %	13 %
0,0	14,1 %	29 %
−0,5	10,0 %	50 %
−1,0	0,0 %	100 % (perfekte Absicherung)

Bei gleicher Rendite von 8 %. Genau das meint Markowitz: Risiko lässt sich reduzieren, ohne dafür Ertrag aufzugeben — allein durch die richtige Kombination.

Die Kehrseite: In Krisen steigen Korrelationen. Ein Portfolio, dessen Diversifikation auf $\rho = 0{,}2$ beruht, kann bei $\rho \to 0{,}9$ plötzlich fast unkorrelierte Absicherung verlieren — genau dann, wenn man sie braucht. Kapitel 20 und Kapitel 22 kommen darauf zurück.

Die Tabelle aus der Handrechnung *Das Gratis-Mittagessen in Zahlen* lässt sich auf zwei unabhängigen Wegen prüfen: über die **analytische Formel** und über eine **Simulation** korrelierter Renditepaare, die von der Formel nichts weiß.

```
#!/usr/bin/env python3

# Diversifikation_Demo.py
"""
```

```
Kapitel Markowitz: Portfoliorisiko in Abhaengigkeit von der Korrelation - die
Handrechnung zum Gratis-Mittagessen, geprueft ueber die analytische Formel und
ueber simulierte, korrelierte Renditen.
"""

import numpy as np

MU = 0.08
SIGMA = 0.20


def sigma_portfolio(rho: float, w1: float = 0.5) -> float:
    """Analytische Formel: sigma_p^2 = w1^2 s1^2 + w2^2 s2^2 + 2 w1 w2 rho s1 s2."""
    w2 = 1 - w1
    varianz = w1**2 * SIGMA**2 + w2**2 * SIGMA**2 + 2 * w1 * w2 * rho * SIGMA**2
    return float(np.sqrt(varianz))


def simuliere(rho: float, n: int = 500_000, seed: int = 7) -> float:
    """Erzeugt n korrelierte Renditepaare und misst die Portfolio-Volatilitaet empirisch."""
    rng = np.random.default_rng(seed)
    kovarianz = np.array([[SIGMA**2, rho * SIGMA**2],
                          [rho * SIGMA**2, SIGMA**2]])
    renditen = rng.multivariate_normal([MU, MU], kovarianz, size=n)
    portfolio = 0.5 * renditen[:, 0] + 0.5 * renditen[:, 1]
    return float(portfolio.std(ddof=1))


if __name__ == "__main__":
    print("=" * 78)
    print("  PORTFOLIORISIKO IN ABHAENGIGKEIT VON DER KORRELATION")
    print("=" * 78)
    print(f"{'rho':>6} | {'sigma_p (Formel)':>18} | {'sigma_p (Simulation)':>20} |
     ↪  {'Risikoreduktion':>16}")
    print("-" * 78)
    for rho in [1.0, 0.5, 0.0, -0.5, -1.0]:
        formel = sigma_portfolio(rho)
        sim = simuliere(rho)
        reduktion = (1 - formel / SIGMA) * 100
        print(f"{rho:6.1f} | {formel*100:16.2f} % | {sim*100:18.2f} % | {reduktion:14.0f} %")

    print("\nDie simulierten Werte (500.000 gezogene Renditepaare je rho) bestaetigen")
    print("die Formel aus der Handrechnung bis auf statistisches Rauschen - und das,")
    print("obwohl Formel und Simulation nichts voneinander wissen.")
```

Erwartete Ausgabe:

```
================================================================================
  PORTFOLIORISIKO IN ABHAENGIGKEIT VON DER KORRELATION
================================================================================
   rho |   sigma_p (Formel) | sigma_p (Simulation) |  Risikoreduktion
--------------------------------------------------------------------------------
   1.0 |            20.00 % |              20.01 % |              0 %
   0.5 |            17.32 % |              17.33 % |             13 %
   0.0 |            14.14 % |              14.13 % |             29 %
  -0.5 |            10.00 % |               9.99 % |             50 %
  -1.0 |             0.00 % |               0.00 % |            100 %

Die simulierten Werte (500.000 gezogene Renditepaare je rho) bestaetigen
die Formel aus der Handrechnung bis auf statistisches Rauschen - und das,
obwohl Formel und Simulation nichts voneinander wissen.
```

19.4 Das Mean-Variance-Modell

Sei $\mathbf{w} = (w_1, \dots, w_n)^\top$ der Vektor der Portfoliogewichte:

$$\min_{\mathbf{w}} \quad \frac{1}{2}\,\mathbf{w}^\top \Sigma \mathbf{w} \;-\; \lambda\, \mu^\top \mathbf{w}$$

$$\text{u. d. N.} \qquad \sum_{i=1}^{n} w_i = 1, \qquad w_i \geq 0 \;\; \forall i$$

 Formel-Lesehilfe * $\frac{1}{2}\mathbf{w}^\top\Sigma\mathbf{w}$ — die halbe Portfoliovarianz (das Risiko). * $\lambda\,\mu^\top\mathbf{w}$ — die mit λ gewichtete erwartete Rendite. Weil sie **abgezogen** wird, wird hohe Rendite belohnt. * $\sum w_i = 1$ — das Kapital wird vollständig investiert. * $w_i \geq 0$ — keine Leerverkäufe (*long only*).

Die Rolle von λ:

λ	Verhalten	Ergebnis
0	Risiko zählt allein	**GMV** — Global Minimum Variance Portfolio
mittel	Abwägung	ein Punkt auf der Effizienzgrenze
$\to \infty$	Rendite zählt allein	alles in den Titel mit höchstem μ

Ohne Formel gesagt: „Streue so, dass das Risiko klein bleibt — aber lass dir dafür nicht zu viel Rendite entgehen. Wie viel Rendite dir das Risiko wert ist, sagt λ.“

 Formel-Übersetzer: die Matrixschreibweise auflösen

Der Sprung von der Zweititelformel weiter oben zu $\mathbf{w}^\top\Sigma\mathbf{w}$ sieht aus wie ein neues Konzept. Er ist keines — es ist **dieselbe Formel**, nur nicht mehr ausgeschrieben:

Mathematik	Alltagssprache
$\mathbf{w}$	„Wie viel Prozent liegt in welchem Titel?“ Ein Vektor mit einem Eintrag je Titel.
Σ	Die Tabelle „wie bewegen sich je zwei Titel zueinander?“. Auf der Diagonale steht das Eigenrisiko σ_i^2, daneben das Zusammenspiel σ_{ij}.
μ	„Was erwarte ich je Titel an Rendite?“
$\mu^\top \mathbf{w}$	Das **gewichtete Mittel** der Renditen — die Portfoliorendite. Genau das, was jeder von Hand ausrechnen würde.
$\mathbf{w}^\top \Sigma \mathbf{w}$	Die **Doppelsumme über alle Paare**: $\sum_i \sum_j w_i w_j \sigma_{ij}$. Bei n Titeln sind das n^2 Summanden — bei 5 Titeln 25, bei 500 Titeln 250 000. Deshalb schreibt man es als Matrixprodukt.

Die Probe. Setzen Sie $n = 2$ in die Doppelsumme ein: Sie erhalten $w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + 2w_1w_2\sigma_{12}$ — Buchstabe für Buchstabe die Formel aus Abschnitt 19.3. Die Matrixschreibweise führt nichts Neues ein; sie erspart nur das Ausschreiben.

In Zahlen. $\sigma_1 = 20\,\%$, $\sigma_2 = 30\,\%$, $\rho = -0{,}4$, Gewichte $(0{,}6;\ 0{,}4)$:

Beitrag	Wert
Eigenrisiken (die Diagonale)	0,0288
Zusammenspiel (alles daneben)	−0,0115
Summe $= \mathbf{w}^\top \Sigma \mathbf{w}$	**0,0173**
Portfoliorisiko $\sigma_p = \sqrt{\cdot}$	**13,15** %

In einem Satz: *Die Diagonale von* Σ *ist das Risiko, das jeder Titel mitbringt; alles daneben ist das, was Diversifikation daraus macht — und nur dieser zweite Teil kann negativ werden.* 13,15 % liegen unter **beiden** Einzelrisiken, und das ist keine Näherung, sondern das exakte Ergebnis der Formel.

Die Effizienzgrenze und das Tangentialportfolio

Die **Effizienzgrenze** (*efficient frontier*) ist die Kurve aller Portfolios, die zu jedem Risikoniveau die höchstmögliche Rendite liefern.

Das **Tangentialportfolio** maximiert die **Sharpe Ratio**:

$$\mathrm{SR}(\mathbf{w}) = \frac{\mathbf{w}^\top \mu - r_f}{\sqrt{\mathbf{w}^\top \Sigma \mathbf{w}}}$$

Formel-Lesehilfe Zähler: **Überrendite** über den risikolosen Zins r_f. Nenner: die Volatilität.

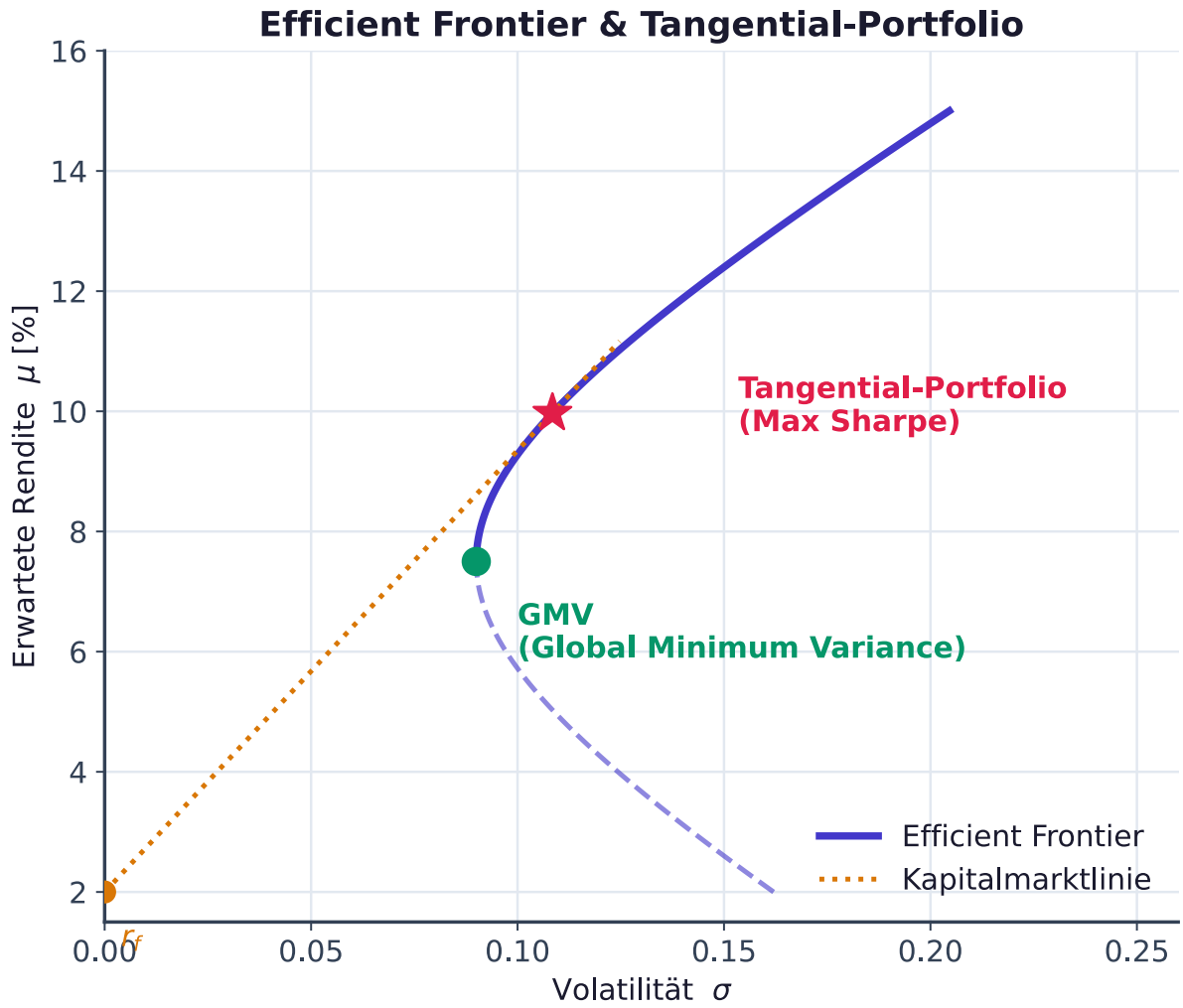

Abbildung 24: Abb. 19.1: Efficient Frontier und Tangential-Portfolio

Ohne Formel gesagt: „Wie viel Mehrertrag bekomme ich je Einheit Risiko, die ich eingehe?“ Eine Sharpe Ratio von 0,8 heißt: Für jeden Prozentpunkt Volatilität gibt es 0,8 Prozentpunkte Überrendite.

Die Korn-Transformation

Die Sharpe Ratio ist ein **Bruch** — und damit **nicht konvex**. CVXPY würde sie ablehnen. Der Ausweg ist ein eleganter Trick, der die Homogenität des Problems ausnutzt:

$$\min_{\mathbf{y},\kappa} \ \frac{1}{2}\mathbf{y}^\top \Sigma \mathbf{y} \qquad \text{u. d. N.} \qquad (\mu - r_f \mathbf{1})^\top \mathbf{y} = 1, \quad \sum_i y_i = \kappa, \quad \mathbf{y} \geq \mathbf{0},\ \kappa > 0$$

Die echten Gewichte ergeben sich am Ende durch Normierung:

$$\mathbf{w}^* = \frac{\mathbf{y}^*}{\kappa^*}$$

 Warum das funktioniert Die Sharpe Ratio ändert sich nicht, wenn man $\mathbf{w}$ mit einer positiven Zahl skaliert — Zähler und Nenner wachsen gleichermaßen. Man darf deshalb die **Skala frei festlegen**. Die Bedingung $(\mu - r_f \mathbf{1})^\top \mathbf{y} = 1$ normiert die Überrendite auf 1; dann ist das Maximieren der Sharpe Ratio gleichbedeutend mit dem **Minimieren des Nenners** — und das ist ein gewöhnliches QP.

Wichtig bei Restriktionen: Alle Nebenbedingungen müssen **mitskaliert** werden. Aus $w_i \leq 0{,}20$ wird $y_i \leq 0{,}20\,\kappa$. Wer das vergisst, erhält ein falsches Portfolio.

19.5 Institutionelle Nebenbedingungen

In der Praxis gelten regulatorische und interne Regeln:

Regel	Formulierung	Zweck
Positionsobergrenze	$w_i \leq w_{\max}$ (z. B. 0,20)	Klumpenrisiko begrenzen
Sektorlimit	$\sum_{i \in \text{Tech}} w_i \leq 0{,}35$	Branchenkonzentration begrenzen
Mindestposition	$w_i = 0$ oder $w_i \geq w_{\min}$	Streuverluste vermeiden
Kardinalität	$\sum_i z_i \leq K$, $w_i \leq M z_i$	Verwaltungsaufwand begrenzen
Turnover-Grenze	$\lVert \mathbf{w} - \mathbf{w}_{\text{alt}} \rVert_1 \leq \tau$	Transaktionskosten dämpfen

Was diese Regeln kosten

Jede Regel schneidet etwas vom Erreichbaren ab. Wie viel, lässt sich ausrechnen — man legt die Effizienzlinie ohne und mit Beschränkungen übereinander:

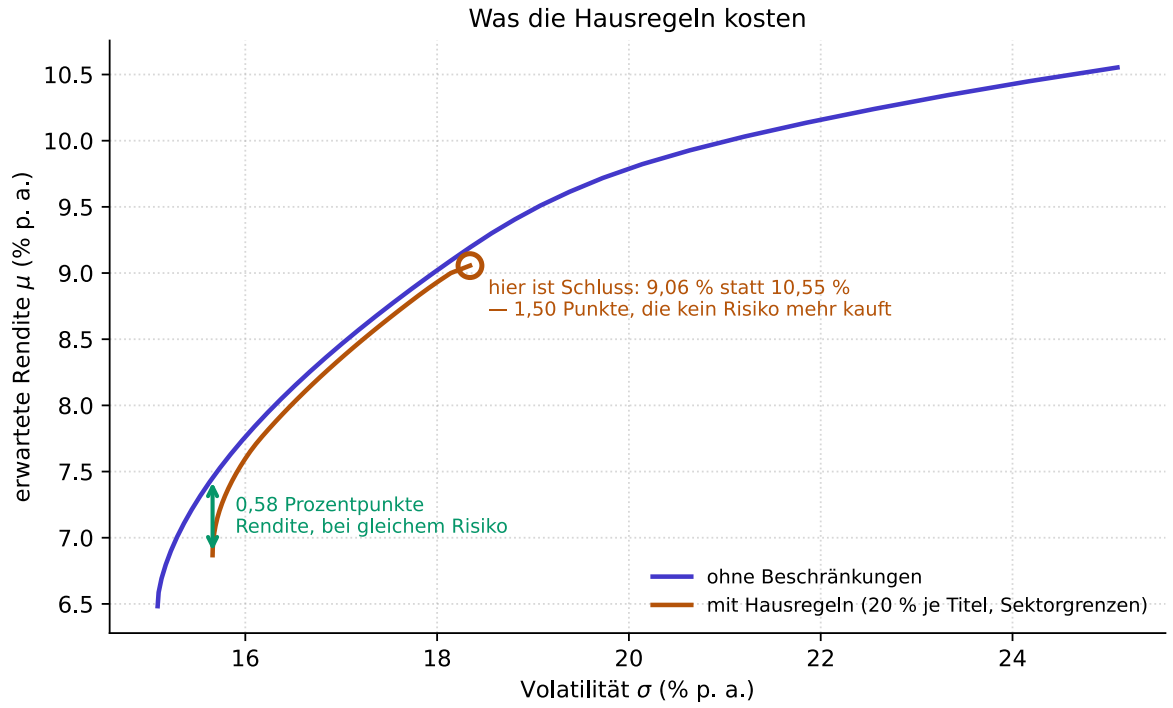

Abbildung 25: Abb. 19.2: Was die Hausregeln kosten. Der senkrechte Abstand ist der Renditeverlust bei gleichem Risiko. Wichtiger ist aber das rechte Ende: Mit den Hausregeln bricht die Linie bei 9,06 % ab — die letzten 1,50 Punkte sind mit keinem Risiko der Welt mehr zu kaufen. Feste Instanz aus einem Faktormodell, erzeugt von `bilder_04/erzeuge_effizienzlinie.py`.

Eine interaktive Fassung dieser Grafik steht auf den Kapitelseiten der Website bereit.

Zwei verschiedene Kosten, und die zweite ist die größere. Der senkrechte Abstand — hier höchstens 0,58 Prozentpunkte — ist das, was die meisten erwarten: bei gleichem Risiko etwas weniger Rendite. Der eigentliche Einschnitt steht am oberen Ende. Ohne Grenzen darf das Portfolio ganz in die renditestärksten Titel gehen und erreicht 10,55 %; mit einer 20-%-Grenze je Titel und den Sektorlimits ist bei **9,06 %** Schluss. Diese 1,50 Punkte sind nicht teuer erkauft, sondern **überhaupt nicht mehr erreichbar** — auch nicht gegen mehr Risiko.

 Warum hier eine feste Instanz und nicht die Kursdaten `Markowitz_CVXPY.py` lädt Livedaten; ein Diagramm daraus sähe an jedem Tag anders aus. Die Figur rechnet deshalb ein festes Universum aus einem Faktormodell — dieselbe Struktur (zehn Titel, vier Sektoren) und dieselben Regeln wie im Kapitel, aber reproduzierbar. Die Zahlen Ihres Livelaufs weichen ab; die Form der Aussage nicht.

 Sektoren niemals über Positionsindizes definieren

Ein naiver, aber gefährlicher Ansatz definiert den Technologiesektor so:

```
tickers = ["AAPL", "MSFT", "NVDA", "AMZN", "JNJ", ...]
tech_indices = [0, 1, 2, 3]        # gemeint: AAPL, MSFT, NVDA, AMZN
```

Weil `yfinance` alphabetisch sortiert, trifft das tatsächlich auf **AAPL, AMZN, CVX (Chevron) und GS (Goldman Sachs)** zu — Microsoft und NVIDIA blieben unbeschränkt.

Robust ist die Definition über Namen:

```
SEKTOREN = {"Technologie": ["AAPL", "MSFT", "NVDA", "AMZN"], ...}
idx = [tickers.index(t) for t in SEKTOREN["Technologie"]]
```

Diese Schreibweise ist nicht nur sicher, sondern auch lesbar — und sie **bricht laut ab**, wenn ein Ticker fehlt, statt still das Falsche zu tun.

19.6 Vollimplementierung mit CVXPY

```
#!/usr/bin/env python3

# Markowitz_CVXPY.py
"""
Kapitel Markowitz: Markowitz-Mean-Variance-Optimierung mit CVXPY.
Enthaelt GMV, Maximum Sharpe (Korn-Transformation), Effizienzgrenze
und institutionelle Restriktionen.

Eigenschaften:
  * Spaltenreihenfolge erzwungen; Sektoren ueber NAMEN statt Indizes
  * Effizienzgrenze bis zur TATSAECHLICH erreichbaren Maximalrendite
    (nicht etwa max(mu)*0.95 - das waere unter Restriktionen oft
     unerreichbar und die Kurve wuerde stillschweigend vorzeitig abbrechen)
  * Pruefung aller Restriktionen nach dem Loesen
  * Vergleich mit Gleichgewichtung als Realitaetscheck
"""

import os

import cvxpy as cp
import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.covariance import LedoitWolf
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt

OUTPUT_DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)), "output")
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
TICKER = ["AAPL", "MSFT", "NVDA", "AMZN", "JNJ", "PFE", "JPM", "GS", "XOM", "CVX"]

# Sektoren ueber Namen definiert, nicht ueber Positionen
SEKTOREN = {
    "Technologie": ["AAPL", "MSFT", "NVDA", "AMZN"],
    "Gesundheit":  ["JNJ", "PFE"],
    "Finanzen":    ["JPM", "GS"],
    "Energie":     ["XOM", "CVX"],
}
SEKTORGRENZEN = {"Technologie": 0.35, "Gesundheit": 0.40,
                 "Finanzen": 0.40, "Energie": 0.40}

MAX_GEWICHT = 0.20      # hoechstens 20 % je Einzeltitel
RISIKOFREI = 0.03       # 3 % p.a.
HANDELSTAGE = 252


def lade_daten():
    """Laedt Kurse und garantiert die Spaltenreihenfolge."""
    ende = pd.Timestamp.today().normalize()
    start = ende - pd.DateOffset(years=2)
    roh = yf.download(TICKER, start=start, end=ende, auto_adjust=True, progress=False)
    if roh.empty:
        raise SystemExit("Download fehlgeschlagen (Netz, Ticker oder Rate-Limit pruefen).")

    if isinstance(roh.columns, pd.MultiIndex):
        kurse = roh["Close"][TICKER].dropna()      # [TICKER] erzwingt die Reihenfolge
    else:
        kurse = roh[["Close"]].dropna()
        kurse.columns = TICKER

    assert list(kurse.columns) == TICKER, "Spaltenreihenfolge weicht ab!"
    renditen = kurse.pct_change().dropna()
    mu = renditen.mean().values * HANDELSTAGE
    sigma = LedoitWolf().fit(renditen.values).covariance_ * HANDELSTAGE
    return kurse, renditen, mu, sigma


def sektor_indizes():
    """Uebersetzt Sektornamen einmalig in Positionsindizes - mit Pruefung."""
    ergebnis = {}
    for sektor, titel in SEKTOREN.items():
        fehlend = [t for t in titel if t not in TICKER]
        if fehlend:
            raise ValueError(f"Sektor '{sektor}': Ticker {fehlend} nicht im Universum.")
        ergebnis[sektor] = [TICKER.index(t) for t in titel]
    return ergebnis
```

```
def basis_restriktionen(w, skala=None):
    """
    Standardrestriktionen. Bei der Korn-Transformation muessen ALLE Grenzen
    mit kappa mitskaliert werden - dafuer dient der Parameter 'skala'.
    """
    eins = 1.0 if skala is None else skala
    idx = sektor_indizes()
    bedingungen = [w >= 0, w <= MAX_GEWICHT * eins]
    for sektor, positionen in idx.items():
        bedingungen.append(cp.sum(w[positionen]) <= SEKTORGRENZEN[sektor] * eins)
    return bedingungen


def loese_gmv(sigma):
    """Global Minimum Variance: minimiere Risiko, ignoriere Rendite."""
    w = cp.Variable(len(sigma))
    problem = cp.Problem(cp.Minimize(0.5 * cp.quad_form(w, sigma)),
                         [cp.sum(w) == 1] + basis_restriktionen(w))
    problem.solve()
    if problem.status not in ("optimal", "optimal_inaccurate"):
        raise SystemExit(f"GMV nicht loesbar: {problem.status}")
    return w.value


def loese_max_sharpe(mu, sigma, r_f):
    """Maximum Sharpe Ratio ueber die Korn-Transformation."""
    n = len(mu)
    ueberrendite = mu - r_f
    if np.all(ueberrendite <= 0):
        raise SystemExit("Kein Titel schlaegt den risikofreien Zins - "
                         "Max-Sharpe-Portfolio existiert nicht.")

    y = cp.Variable(n)
    kappa = cp.Variable(nonneg=True)
    bedingungen = ([ueberrendite @ y == 1, cp.sum(y) == kappa]
                   + basis_restriktionen(y, skala=kappa))
    problem = cp.Problem(cp.Minimize(0.5 * cp.quad_form(y, sigma)), bedingungen)
    problem.solve()
    if problem.status not in ("optimal", "optimal_inaccurate"):
        raise SystemExit(f"Max-Sharpe nicht loesbar: {problem.status}")
    return y.value / kappa.value


def max_erreichbare_rendite(mu, sigma):
    """
    Achtung: max(mu)*0.95 als obere Grenze der Frontier anzunehmen, ist unter
    Positions- und Sektorgrenzen oft unerreichbar; die Kurve wuerde dann
```

```
    stillschweigend abbrechen. Hier wird die tatsaechliche Obergrenze berechnet.
    """
    w = cp.Variable(len(mu))
    problem = cp.Problem(cp.Maximize(mu @ w), [cp.sum(w) == 1] + basis_restriktionen(w))
    problem.solve()
    return float(problem.value)


def berechne_frontier(mu, sigma, ret_min, ret_max, punkte=40):
    """Effizienzgrenze durch Variation der Mindestrendite."""
    n = len(mu)
    w = cp.Variable(n)
    ziel_rendite = cp.Parameter()
    problem = cp.Problem(
        cp.Minimize(0.5 * cp.quad_form(w, sigma)),
        [cp.sum(w) == 1, mu @ w >= ziel_rendite] + basis_restriktionen(w))

    volas, renditen, uebersprungen = [], [], 0
    for ziel in np.linspace(ret_min, ret_max, punkte):
        ziel_rendite.value = ziel
        problem.solve()
        if problem.status in ("optimal", "optimal_inaccurate"):
            volas.append(float(np.sqrt(w.value @ sigma @ w.value)))
            renditen.append(float(mu @ w.value))
        else:
            uebersprungen += 1
    if uebersprungen:
        print(f"  Hinweis: {uebersprungen} Zielrenditen waren nicht erreichbar.")
    return np.array(volas), np.array(renditen)


def pruefe_restriktionen(w, bezeichnung):
    """Nach dem Loesen: haelt die Loesung wirklich alle Regeln ein?"""
    assert abs(w.sum() - 1) < 1e-6, f"{bezeichnung}: Summe != 1"
    assert w.min() > -1e-6, f"{bezeichnung}: negatives Gewicht"
    assert w.max() < MAX_GEWICHT + 1e-6, f"{bezeichnung}: Positionsgrenze verletzt"
    for sektor, positionen in sektor_indizes().items():
        anteil = w[positionen].sum()
        assert anteil < SEKTORGRENZEN[sektor] + 1e-6, \
            f"{bezeichnung}: Sektor {sektor} bei {anteil:.3f} ueber Grenze"


def kennzahlen(w, mu, sigma, r_f):
    rendite = float(mu @ w)
    vola = float(np.sqrt(w @ sigma @ w))
    return rendite, vola, (rendite - r_f) / vola
```

```
if __name__ == "__main__":
    kurse, renditen, mu, sigma = lade_daten()
    n = len(TICKER)

    w_gmv = loese_gmv(sigma)
    w_sharpe = loese_max_sharpe(mu, sigma, RISIKOFREI)
    w_gleich = np.ones(n) / n

    pruefe_restriktionen(w_gmv, "GMV")
    pruefe_restriktionen(w_sharpe, "Max Sharpe")

    print("=" * 88)
    print("        ERGEBNISSE DER MEAN-VARIANCE-OPTIMIERUNG")
    print("=" * 88)
    print(f"Datenbasis: {len(renditen)} Handelstage, {n} Titel, "
          f"risikofreier Zins {RISIKOFREI*100:.1f} %")
    print(f"Restriktionen: max. {MAX_GEWICHT*100:.0f} % je Titel, "
          f"Sektorgrenzen {SEKTORGRENZEN}\n")

    print(f"{'Portfolio':<26} {'Rendite':>10} {'Volatilitaet':>13} {'Sharpe':>9}")
    print("-" * 88)
    for name, w in [("Global Minimum Variance", w_gmv),
                    ("Maximum Sharpe Ratio", w_sharpe),
                    ("Gleichgewichtung (1/N)", w_gleich)]:
        r, v, sr = kennzahlen(w, mu, sigma, RISIKOFREI)
        print(f"{name:<26} {r*100:>9.2f} % {v*100:>12.2f} % {sr:>9.2f}")

    # --- Gewichte je Titel ------------------------------------------------
    print("\n--- Optimierte Portfoliogewichte ---")
    sektor_je_titel = {t: s for s, titel in SEKTOREN.items() for t in titel}
    tabelle = pd.DataFrame({
        "Ticker": TICKER,
        "Sektor": [sektor_je_titel[t] for t in TICKER],
        "Rendite p.a.": [f"{r*100:+6.1f} %" for r in mu],
        "Vola p.a.": [f"{np.sqrt(sigma[i, i])*100:5.1f} %" for i in range(n)],
        "GMV": [f"{w*100:5.1f} %" for w in w_gmv],
        "Max Sharpe": [f"{w*100:5.1f} %" for w in w_sharpe],
    })
    print(tabelle.to_string(index=False))

    print("\n--- Sektoraufteilung (Kontrolle) ---")
    print(f"{'Sektor':<14} {'Grenze':>8} {'GMV':>9} {'Max Sharpe':>12}")
    for sektor, positionen in sektor_indizes().items():
        print(f"{sektor:<14} {SEKTORGRENZEN[sektor]*100:>7.0f} % "
              f"{w_gmv[positionen].sum()*100:>8.1f} % "
              f"{w_sharpe[positionen].sum()*100:>11.1f} %")

    # --- Effizienzgrenze --------------------------------------------------
```

```
    ret_gmv = float(mu @ w_gmv)
    ret_max = max_erreichbare_rendite(mu, sigma)
    print(f"\nEffizienzgrenze von {ret_gmv*100:.2f} % bis {ret_max*100:.2f} % "
          f"(unter Restriktionen tatsaechlich erreichbar)")
    print(f"  Zum Vergleich: bester Einzeltitel {mu.max()*100:.2f} % - "
          f"durch die Grenzen nicht erreichbar.")
    volas, rets = berechne_frontier(mu, sigma, ret_gmv, ret_max)

    # --- Diagramm ---------------------------------------------------------
    plt.figure(figsize=(11, 6.5))
    plt.plot(volas * 100, rets * 100, "b-", lw=2.5, label="Effizienzgrenze (restringiert)")

    for w, farbe, marker, groesse, name in [
            (w_gmv, "green", "o", 150, "GMV"),
            (w_sharpe, "red", "*", 260, "Max Sharpe"),
            (w_gleich, "purple", "D", 110, "Gleichgewichtung")]:
        r, v, sr = kennzahlen(w, mu, sigma, RISIKOFREI)
        plt.scatter([v * 100], [r * 100], color=farbe, marker=marker, s=groesse,
                    zorder=5, label=f"{name} (SR={sr:.2f})")

    # Kapitalmarktlinie durch den risikofreien Zins und das Tangentialportfolio
    r_s, v_s, _ = kennzahlen(w_sharpe, mu, sigma, RISIKOFREI)
    x_linie = np.array([0, v_s * 1.25])
    plt.plot(x_linie * 100, (RISIKOFREI + (r_s - RISIKOFREI) / v_s * x_linie) * 100,
             color="orange", ls=":", lw=2, label="Kapitalmarktlinie")

    for i, t in enumerate(TICKER):
        plt.scatter(np.sqrt(sigma[i, i]) * 100, mu[i] * 100, color="gray", alpha=0.5, s=40)
        plt.annotate(t, (np.sqrt(sigma[i, i]) * 100 + 0.4, mu[i] * 100), fontsize=8)

    plt.title("Markowitz-Effizienzgrenze mit Sektor- und Positionsgrenzen", fontsize=12)
    plt.xlabel("Annualisierte Volatilitaet [%]")
    plt.ylabel("Annualisierte erwartete Rendite [%]")
    plt.grid(True, linestyle=":", alpha=0.6)
    plt.legend(loc="best")
    plt.tight_layout()
    ziel = os.path.join(OUTPUT_DIR, "markowitz_efficient_frontier.png")
    plt.savefig(ziel, dpi=150)
    print(f"\nDiagramm gespeichert unter '{ziel}'")
    print("=" * 88)
```

□ Code-Durchgang: vier typische Fehlerquellen und ihre Absicherung

Stelle	Naiver Ansatz	Robuster Ansatz (hier verwendet)
`roh["Close"][TICKER]`	`roh["Close"]` — alphabetisch	Reihenfolge erzwungen + `assert`

Stelle	Naiver Ansatz	Robuster Ansatz (hier verwendet)
`SEKTOREN` als Namensdict	`tech_indices = [0,1,2,3]`	Namen, mit Existenzprüfung
`basis_restriktionen(y, skala=kappa)`	Grenzen von Hand mitskaliert	zentral, dadurch nicht vergessbar
`max_erreichbare_rendite()`	`max(mu)*0.95` geraten	tatsächliche Grenze berechnet

Zur letzten Zeile: Unter der 20-%-Positionsgrenze kann kein Portfolio die Rendite des besten Einzeltitels erreichen — man muss mindestens fünf Titel halten. Würde man die Obergrenze der Effizienzgrenze dennoch bei $0{,}95 \cdot \max(\mu)$ ansetzen, würden die unerreichbaren Zielrenditen vom `if status == "optimal"` **stillschweigend übersprungen** — die gezeichnete Kurve bräche dadurch früher ab, ohne dass es auffiele.

 Zur Reproduzierbarkeit Dieses Programm lädt Live-Daten. **Ihre Zahlen werden abweichen.** Prüfen Sie stattdessen die **Struktur** des Ergebnisses: Hält die Sektorgrenze? Summieren sich die Gewichte auf 100 %? Liegt die Sharpe Ratio des Max-Sharpe-Portfolios über der des GMV? Genau dafür sind die `assert`-Prüfungen da.

 Der Realitätscheck: Gleichgewichtung Das Programm vergleicht immer mit dem **1/N-Portfolio**. Das ist kein Scherz: DeMiguel, Garlappi und Uppal zeigten 2009 in einer viel beachteten Studie, dass die naive Gleichgewichtung viele optimierte Strategien out of sample schlägt — weil sie keinerlei Schätzung benötigt und damit auch keinen Schätzfehler enthält. **Wenn Ihre Optimierung 1/N nicht schlägt, ist sie ihren Aufwand nicht wert.** Diese Messlatte sollte in jedem Portfolioprojekt stehen.

19.7 Übungsaufgaben

Lösungen: Abschnitt A.19.

Aufgabe 19.1 — **Diversifikationseffekt rechnen.** Zwei Titel: $\mu_1 = 10\,\%$, $\sigma_1 = 25\,\%$; $\mu_2 = 6\,\%$, $\sigma_2 = 12\,\%$; $\rho = -0{,}2$. Berechnen Sie Rendite und Volatilität für die Gewichtungen $(1;0)$, $(0{,}5;0{,}5)$, $(0{,}3;0{,}7)$ und $(0;1)$. Welche Kombination hat die beste Sharpe Ratio bei $r_f = 2\,\%$?

Aufgabe 19.2 — **Lambda deuten.** Was passiert mit dem optimalen Portfolio, wenn λ von 0 auf ∞ steigt? Skizzieren Sie den Weg auf der Effizienzgrenze.

Aufgabe 19.3 — **Korn-Transformation nachvollziehen.** Zeigen Sie rechnerisch, dass die Sharpe Ratio invariant gegenüber positiver Skalierung ist: $\mathrm{SR}(c\,\mathbf{w}) = \mathrm{SR}(\mathbf{w})$ für $c > 0$. Warum gilt das **nicht**, wenn die Nebenbedingung $\sum w_i = 1$ mitgeführt wird — und wie löst die Transformation dieses Problem?

Aufgabe 19.4 — **Restriktionen kosten Rendite.** Lassen Sie `Markowitz_CVXPY.py` mit `MAX_GEWICHT` von 1,0 (keine Grenze) bis 0,10 in Schritten von 0,05 laufen. Tabellieren Sie Sharpe Ratio und maximal erreichbare Rendite. (a) Was kostet die 20-%-Grenze an Sharpe Ratio? (b) Ab welcher Grenze wird das Problem unlösbar? Warum?

Aufgabe 19.5 □□ — Den Sektorfehler nachstellen. Bauen Sie den Fehler bewusst nach: Verwenden Sie `tech_indices = [0,1,2,3]` auf den alphabetisch sortierten Spalten. Vergleichen Sie das Ergebnis mit der korrekten Version. (a) Welche Titel werden tatsächlich beschränkt? (b) Wie stark unterscheiden sich die Portfoliogewichte? (c) Fällt der Fehler in den ausgegebenen Kennzahlen auf?

Aufgabe 19.6 □□□ — Kardinalität ergänzen. Erweitern Sie das Modell um „höchstens $K = 5$ Titel“ mit Binärvariablen. CVXPY braucht dafür `cp.Variable(n, boolean=True)` und einen MIQP-fähigen Solver (z. B. `SCIP` über `cp.SCIP` oder `ECOS_BB`). Vergleichen Sie Sharpe Ratio und Rechenzeit mit dem unbeschränkten Fall.

Aufgabe 19.7 □□□ — Out-of-Sample-Test. Teilen Sie die Historie in zwei Hälften. Optimieren Sie auf der ersten, bewerten Sie auf der zweiten. Vergleichen Sie GMV, Max Sharpe und 1/N. (a) Welches Portfolio gewinnt out of sample? (b) Wie ändert sich das Bild mit Ledoit-Wolf gegenüber der Stichprobenkovarianz? (c) Welche Schlussfolgerung ziehen Sie für die Praxis?

19.8 Finde den Denkfehler

Kapitel 18 hat gezeigt, was eine schlecht geschätzte **Kovarianzmatrix** anrichtet. Bei den erwarteten **Renditen** ist es schlimmer — und weil sie in der Zielfunktion stehen statt in der Nebenbedingung, fällt es schwerer auf.

> **□ Finde den Denkfehler: Zwölf gleiche Anlagen, ein sehr ungleiches Portfolio**
>
> Ein Analyst optimiert ein Portfolio aus zwölf Anlagen. Erwartete Renditen und Kovarianzen schätzt er aus einem Jahr Tagesdaten — 250 Beobachtungen, in der Praxis durchaus üblich.
>
> Zur Kontrolle bauen wir das Experiment so, dass die richtige Antwort **feststeht**: Alle zwölf Anlagen haben in Wahrheit exakt dieselbe Rendite (0,0400 % täglich) und dieselbe Schwankung, und sie sind unabhängig. Ein korrektes Verfahren müsste also gleichgewichten.
>
> Der Optimierer legt stattdessen im Mittel **38 %** in eine einzige Anlage — und meldet für das Portfolio eine Tagesrendite von **0,1261 %**, mehr als das Dreifache dessen, was überhaupt erzielbar ist.
>
> **Ihre Aufgabe:** (a) Woher kommt die Spanne von 0,24 Prozentpunkten zwischen der höchsten und der niedrigsten geschätzten Rendite, wenn doch alle Anlagen identisch sind? (b) Rechnen Sie nach, wie viele Beobachtungen nötig wären, um eine Rendite von 0,04 % bei 1,2 % Tagesschwankung überhaupt von null zu unterscheiden. (c) Warum wirkt sich derselbe Schätzfehler bei Renditen stärker aus als bei Kovarianzen? (d) Nennen Sie drei Gegenmittel — und begründen Sie, warum das erste davon das wirksamste ist.
>
> *Auflösung: Abschnitt A.19.*

```
#!/usr/bin/env python3

# Renditeschaetzung_Falle.py
"""
Kapitel Markowitz: Warum geschaetzte Renditen noch gefaehrlicher sind als geschaetzte
```

```
Kovarianzen.

Das Kapitel Finanzdaten hat gezeigt, was eine schlecht geschaetzte Kovarianzmatrix
anrichtet. Bei den erwarteten RENDITEN ist es schlimmer - aus einem einfachen
statistischen Grund: Um eine Rendite von 0,04 % pro Tag von null zu
unterscheiden, braucht man bei 1,2 % Tagesschwankung ueber tausend
Beobachtungen. Kovarianzen konvergieren dagegen deutlich schneller.

Das Experiment ist bewusst so gebaut, dass die Antwort feststeht: ALLE zwoelf
Anlagen haben exakt dieselbe wahre Rendite und dieselbe Schwankung. Ein
korrektes Ergebnis waere also Gleichgewichtung. Was der Optimierer stattdessen
tut, zeigt der Lauf.

Wiederholt wird 200 mal, damit das Ergebnis nicht von einer Zufallsstichprobe
abhaengt.

Benoetigt: numpy, cvxpy
"""

from __future__ import annotations

import numpy as np
import cvxpy as cp

ANZAHL_ANLAGEN = 12
BEOBACHTUNGEN = 250              # ein Jahr Tagesdaten
BEOBACHTUNGEN_SPAETER = 2000     # der "spaetere Verlauf"
WIEDERHOLUNGEN = 200

# Die Wahrheit, die der Optimierer nicht kennt: alle Anlagen sind gleich.
WAHRE_RENDITE = 0.0004           # 0,04 % pro Tag
WAHRE_SCHWANKUNG = 0.012         # 1,2 % pro Tag
RISIKOAVERSION = 10.0

RNG = np.random.default_rng(5)


def baue_optimierer() -> tuple[cp.Problem, cp.Variable, cp.Parameter, cp.Parameter]:
    """Baut das Markowitz-Problem EINMAL mit Parametern.

    Bei 200 Wiederholungen ist das der Unterschied zwischen Sekunden und
    Minuten: cp.Parameter erlaubt es, nur die Daten zu tauschen, statt den
    Ausdrucksbaum jedes Mal neu zu kompilieren (siehe Kapitel Oekosystem).
    """
    gewichte = cp.Variable(ANZAHL_ANLAGEN, nonneg=True)
    renditen = cp.Parameter(ANZAHL_ANLAGEN)
    kovarianz = cp.Parameter((ANZAHL_ANLAGEN, ANZAHL_ANLAGEN), PSD=True)
    problem = cp.Problem(
```

```
        cp.Maximize(renditen @ gewichte
                    - RISIKOAVERSION * cp.quad_form(gewichte, kovarianz)),
        [cp.sum(gewichte) == 1])
    return problem, gewichte, renditen, kovarianz


def erzeuge(perioden: int) -> np.ndarray:
    """Renditen, bei denen alle Anlagen identisch verteilt und unabhaengig sind."""
    return RNG.normal(WAHRE_RENDITE, WAHRE_SCHWANKUNG,
                      (perioden, ANZAHL_ANLAGEN))


if __name__ == "__main__":
    problem, gewichte, renditen, kovarianz = baue_optimierer()

    im_zeitraum, spaeter, spaeter_gleich = [], [], []
    groesstes_gewicht, schaetzspanne = [], []

    for _ in range(WIEDERHOLUNGEN):
        schaetzdaten = erzeuge(BEOBACHTUNGEN)
        spaetere_daten = erzeuge(BEOBACHTUNGEN_SPAETER)

        geschaetzte_rendite = schaetzdaten.mean(axis=0)
        geschaetzte_kovarianz = np.cov(schaetzdaten, rowvar=False)

        renditen.value = geschaetzte_rendite
        # Symmetrisieren und minimal anheben: numerisches Rauschen kann die
        # Matrix sonst knapp unter die PSD-Grenze druecken (Kapitel Fundament).
        kovarianz.value = ((geschaetzte_kovarianz + geschaetzte_kovarianz.T) / 2
                           + 1e-10 * np.eye(ANZAHL_ANLAGEN))
        problem.solve()

        w = np.array(gewichte.value).ravel()
        im_zeitraum.append(float((schaetzdaten @ w).mean()))
        spaeter.append(float((spaetere_daten @ w).mean()))
        spaeter_gleich.append(float(spaetere_daten.mean()))
        groesstes_gewicht.append(float(w.max()))
        schaetzspanne.append(float(geschaetzte_rendite.max()
                                   - geschaetzte_rendite.min()))

    prozent = lambda werte: float(np.mean(werte)) * 100

    print("=" * 78)
    print("  WENN DER OPTIMIERER GESCHAETZTE RENDITEN GLAUBT")
    print("=" * 78)
    print(f"{ANZAHL_ANLAGEN} Anlagen, {WIEDERHOLUNGEN} Wiederholungen, "
          f"je {BEOBACHTUNGEN} Beobachtungen zur Schaetzung.")
    print(f"In Wahrheit haben ALLE dieselbe Tagesrendite von "
```

```
          f"{WAHRE_RENDITE * 100:.4f} % und dieselbe Schwankung.")
    print("Die richtige Antwort waere also: gleichgewichten.")
    print()
    print(f"Mittlere Spanne der geschaetzten Renditen: "
          f"{prozent(schaetzspanne):.4f} Prozentpunkte")
    print(f"  -> Das ist das {prozent(schaetzspanne) / (WAHRE_RENDITE * 100):.1f}-Fache "
          f"des wahren Werts. Aus reinem Rauschen.")
    print(f"Mittleres groesstes Einzelgewicht: "
          f"{prozent(groesstes_gewicht):.1f} % "
          f"(bei Gleichgewichtung waeren es {100 / ANZAHL_ANLAGEN:.1f} %)")
    print()
    print(f"{'':<38} {'Tagesrendite':>14}")
    print("-" * 78)
    print(f"{'Wahrheit (alle Anlagen)':<38} {WAHRE_RENDITE * 100:>13.4f} %")
    print(f"{'Optimierer IM Schaetzzeitraum':<38} {prozent(im_zeitraum):>13.4f} %"
          f"   <- Illusion")
    print(f"{'Optimierer AUSSERHALB':<38} {prozent(spaeter):>13.4f} %")
    print(f"{'Gleichgewichtung AUSSERHALB':<38} {prozent(spaeter_gleich):>13.4f} %")
    print("-" * 78)

    faktor = prozent(im_zeitraum) / (WAHRE_RENDITE * 100)
    vorsprung = prozent(spaeter) - prozent(spaeter_gleich)
    print(f"\nIm Schaetzzeitraum verspricht der Optimierer das "
          f"{faktor:.1f}-Fache der wahren Rendite.")
    print(f"Ausserhalb bleibt davon nichts: Er liegt {abs(vorsprung):.4f} "
          f"Prozentpunkte")
    print(f"{'schlechter' if vorsprung < 0 else 'besser'} als blosse Gleichgewichtung.")
    print()
    print("Der Grund ist statistisch, nicht finanzwirtschaftlich: Um eine")
    print(f"Rendite von {WAHRE_RENDITE*100:.2f} % taeglich von null zu unterscheiden,")
    print(f"braucht man bei {WAHRE_SCHWANKUNG*100:.1f} % Schwankung ueber tausend")
    print("Beobachtungen. Mit 250 misst man ueberwiegend Rauschen - und der")
    print("Optimierer nimmt jedes Rauschen fuer bare Muenze.")
    print()
    print("Drei praktische Konsequenzen:")
    print("  1. Verzichten Sie auf Renditeschaetzungen, wo es geht. Das")
    print("     Minimum-Varianz-Portfolio braucht gar keine.")
    print("  2. Wenn Sie welche brauchen: schrumpfen Sie sie stark zur Mitte")
    print("     (James-Stein, Black-Litterman) - viel staerker als Kovarianzen.")
    print("  3. Begrenzen Sie Einzelgewichte. Was der Optimierer nicht darf,")
    print("     kann er auch nicht auf Rauschen setzen.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  WENN DER OPTIMIERER GESCHAETZTE RENDITEN GLAUBT
==============================================================================
```

```
12 Anlagen, 200 Wiederholungen, je 250 Beobachtungen zur Schaetzung.
In Wahrheit haben ALLE dieselbe Tagesrendite von 0.0400 % und dieselbe Schwankung.
Die richtige Antwort waere also: gleichgewichten.

Mittlere Spanne der geschaetzten Renditen: 0.2433 Prozentpunkte
  -> Das ist das 6.1-Fache des wahren Werts. Aus reinem Rauschen.
Mittleres groesstes Einzelgewicht: 38.0 % (bei Gleichgewichtung waeren es 8.3 %)

                                        Tagesrendite
------------------------------------------------------------------------------
Wahrheit (alle Anlagen)                       0.0400 %
Optimierer IM Schaetzzeitraum                 0.1261 %   <- Illusion
Optimierer AUSSERHALB                         0.0390 %
Gleichgewichtung AUSSERHALB                   0.0396 %
------------------------------------------------------------------------------

Im Schaetzzeitraum verspricht der Optimierer das 3.2-Fache der wahren Rendite.
Ausserhalb bleibt davon nichts: Er liegt 0.0006 Prozentpunkte
schlechter als blosse Gleichgewichtung.

Der Grund ist statistisch, nicht finanzwirtschaftlich: Um eine
Rendite von 0.04 % taeglich von null zu unterscheiden,
braucht man bei 1.2 % Schwankung ueber tausend
Beobachtungen. Mit 250 misst man ueberwiegend Rauschen - und der
Optimierer nimmt jedes Rauschen fuer bare Muenze.

Drei praktische Konsequenzen:
  1. Verzichten Sie auf Renditeschaetzungen, wo es geht. Das
     Minimum-Varianz-Portfolio braucht gar keine.
  2. Wenn Sie welche brauchen: schrumpfen Sie sie stark zur Mitte
     (James-Stein, Black-Litterman) - viel staerker als Kovarianzen.
  3. Begrenzen Sie Einzelgewichte. Was der Optimierer nicht darf,
     kann er auch nicht auf Rauschen setzen.
==============================================================================
```

 Merksatz Die Effizienzlinie aus dem Kapitelanfang ist mathematisch tadellos — und praktisch nur so viel wert wie die Renditen, die man hineinsteckt. Das **Minimum-Varianz-Portfolio** ist deshalb in der Praxis oft die bessere Wahl als das Maximum-Sharpe-Portfolio: Es braucht überhaupt keine Renditeschätzung, sondern nur die Kovarianzmatrix — und die lässt sich ungleich zuverlässiger schätzen.

19.9 Micro-Quiz

Micro-Quiz 19: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Auf der Effizienzlinie liegt das Portfolio mit der höchsten Rendite. Warum ist es trotzdem selten die richtige Wahl? (a) Weil es rechnerisch instabil ist. (b) Weil es das schlechteste Verhältnis von Rendite zu Risiko hat und zwangsläufig vollständig in die renditestärkste Einzelanlage geht — also gar nicht mehr diversifiziert ist. (c) Weil CVXPY dort keine Lösung mehr findet.

2. Warum ist das Minimum-Varianz-Portfolio in der Praxis oft robuster als das Maximum-Sharpe-Portfolio? (a) Weil es weniger Nebenbedingungen hat. (b) Weil es **ohne Renditeschätzung** auskommt. Renditen sind die mit Abstand am unzuverlässigsten geschätzte Eingangsgröße; wer sie nicht braucht, umgeht das Problem. (c) Weil es immer die höhere Rendite erzielt.

3. Ihr optimiertes Portfolio legt 44 % in einen einzigen von zwölf Titeln. Was ist Ihre erste Vermutung? (a) Dieser Titel ist tatsächlich deutlich besser als die anderen. (b) Der Optimierer folgt einem Schätzfehler in den erwarteten Renditen. Prüfen Sie, wie stark die Gewichte auf kleine Datenänderungen reagieren, und begrenzen Sie Einzelpositionen. (c) Die Kovarianzmatrix ist falsch skaliert.

19.10 Selbsttest

Antworten: Anhang A.

1. Warum senkt Diversifikation das Risiko, ohne Rendite zu kosten?
2. Welches Portfolio erhält man für $\lambda = 0$, und warum braucht es keine Renditeprognose?
3. Warum ist die Sharpe Ratio nicht konvex, und wie behebt die Korn-Transformation das?
4. Was muss man bei der Korn-Transformation mit den Nebenbedingungen tun?
5. Warum ist die Gleichgewichtung ein ernstzunehmender Vergleichsmaßstab?

19.11 Zusammenfassung

- **Diversifikation ist das einzige Gratis-Mittagessen:** Risiko sinkt, Rendite bleibt. Aber: In Krisen steigen Korrelationen, genau wenn man die Streuung braucht.
- **λ steuert zwischen GMV und Renditemaximierung.** Das GMV benötigt keine Renditeprognose und ist deshalb robuster.
- **Die Korn-Transformation** macht aus der nicht-konvexen Sharpe-Maximierung ein QP — alle Nebenbedingungen müssen dabei mit κ mitskaliert werden.
- **Sektoren immer über Namen definieren**, nie über Positionsindizes.
- **Die Obergrenze der Effizienzgrenze muss berechnet werden**, nicht geraten — sonst bricht die Kurve stillschweigend ab.

- **Vergleichen Sie immer mit 1/N.** Wer die naive Gleichgewichtung nicht schlägt, hat Schätzrauschen optimiert.
- **Renditeschätzungen sind die schwächste Stelle des ganzen Modells.** Der Standardfehler einer Renditeschätzung aus einem Jahr Tagesdaten ist rund doppelt so groß wie der geschätzte Wert selbst; für eine belastbare Schätzung bräuchte man über ein Jahrzehnt. Deshalb ist „keine Renditeschätzung brauchen“ (GMV) wirksamer als jede Verbesserung der Schätzung.
- **Die Effizienzlinie ist eine Speisekarte, keine Empfehlung.** Welcher Punkt richtig ist, entscheidet die Risikotragfähigkeit — eine unternehmerische, keine mathematische Frage.

Ausblick. Kapitel 20 ersetzt die Varianz durch ein Risikomaß, das die tatsächlichen Verlustränder erfasst — und fügt Transaktionskosten hinzu.

Kapitel 20: Tail-Risiko, CVaR und Transaktionskosten

Kapitel auf einen Blick

Worum geht es? Um zwei Schwächen des Markowitz-Modells: Es unterschätzt Extremverluste und ignoriert die Kosten des Umschichtens. Beide lassen sich mit konvexen Mitteln beheben.

Voraussetzungen: Kapitel 19, Kapitel 5 (LP-Formulierungen).

Danach können Sie: VaR und CVaR unterscheiden, den CVaR nach Rockafellar/Uryasev als lineares Programm formulieren, Turnover über die L_1-Norm bestrafen — und begründen, warum VaR-Kennzahlen nicht über Einheiten addiert werden dürfen.

Zeitbedarf: ca. 5 Stunden.

Programme:
`VaR_CVaR_Demo.py`
`CVaR_Portfolio.py`

Notebook: `Notebooks_04/cvar.ipynb`

20.1 In 5 Minuten gelöst

In 5 Minuten gelöst: Zwei Anlagen, dasselbe Risiko?

Zwei Anlagen wurden über 100 Tage beobachtet. An 94 Tagen legten **beide** um 0,5 % zu. Sie unterscheiden sich nur in den sechs schlechten Tagen:

- **Anlage A:** sechsmal −3 %.
- **Anlage B:** fünfmal −3 % — und einmal **−40 %**.

```
import numpy as np

gut = np.full(94, 0.5)
A = np.concatenate([gut, np.full(6, -3.0)])
```

```python
B = np.concatenate([gut, np.full(5, -3.0), [-40.0]])

for name, r in [("Anlage A", A), ("Anlage B", B)]:
    schwelle = np.percentile(r, 5)              # 5-%-Quantil
    var = -schwelle                             # Value at Risk
    cvar = -r[r <= schwelle].mean()             # Mittel der schlimmsten 5 %
    print(f"{name}: VaR(95) {var:5.2f} %   CVaR(95) {cvar:5.2f} %")
```

Ausgabe:

```
Anlage A: VaR(95)  3.00 %   CVaR(95)  3.00 %
Anlage B: VaR(95)  3.00 %   CVaR(95)  9.17 %
```

Der Value at Risk ist für beide Anlagen exakt gleich: 3,00 %. Ein Risikobericht, der nur den VaR ausweist, würde die beiden als gleich riskant einstufen.

Dabei kann Anlage B an einem einzigen Tag **40 %** verlieren. Das ist kein Randfall, den man übersehen darf — es ist der Fall, wegen dessen es Risikomanagement gibt.

Der Grund für diese Blindheit steckt in der Definition:

Maß	Was es beantwortet	Was es dabei übersieht
VaR (95 %)	„Welchen Verlust überschreite ich an höchstens 5 % der Tage?“	**Alles**, was jenseits dieser Schwelle passiert
CVaR (95 %)	„Wie hoch ist der Verlust *im Mittel*, **wenn** es schiefgeht?“	nichts im Schwanz

Der VaR ist ein **Quantil** — er markiert eine Grenze und schaut nicht dahinter. Ob hinter der Grenze −3 % oder −40 % liegen, ändert ihn nicht. Der CVaR mittelt genau über diesen Bereich und macht den Unterschied sichtbar: 3,00 % gegen 9,17 %.

 Merksatz Der VaR sagt Ihnen, **wie oft** es schiefgeht. Der CVaR sagt Ihnen, **wie schlimm** es dann ist. Für die Frage, ob ein Unternehmen einen Verlust überlebt, zählt ausschließlich die zweite.

Und es kommt noch besser. Der CVaR ist nicht nur aussagekräftiger, er ist auch mathematisch handlicher: Er lässt sich als **lineares Programm** minimieren, während die Minimierung des VaR ein nicht-konvexes Problem mit vielen lokalen Optima ist (Kapitel 11). Das ist ein seltener Glücksfall — das bessere Maß ist hier zugleich das leichter optimierbare. Wie das geht, zeigt Abschnitt 20.4.

20.2 Lernziele

Nach diesem Kapitel können Sie …

1. … erklären, warum Marktrenditen keine Normalverteilung haben und was daraus folgt.
2. … VaR und CVaR definieren und begründen, warum nur der CVaR kohärent ist.
3. … das Rockafellar-Uryasev-Theorem anwenden, um den CVaR linear zu formulieren.
4. … Transaktionskosten über die L_1-Norm modellieren.
5. … Einheiten konsistent halten und Nebenbedingungen vektorisieren.
6. … erklären, warum zwei Anlagen mit identischem VaR völlig verschiedene Extremverluste haben können.
7. … die Folgen fehlender Subadditivität für die Verteilung von Risikobudgets benennen.

20.3 Die zwei Schwächen des Markowitz-Modells

Erstens: die Normalverteilungs-Illusion. Die Varianz behandelt Aufwärts- und Abwärtsschwankungen gleich und unterstellt implizit symmetrische, dünn auslaufende Verteilungen. Reale Marktrenditen haben aber **fette Ränder** (*fat tails*) und **negative Schiefe**: Extreme Verluste treten deutlich häufiger auf, als eine Normalverteilung vorhersagt.

Zweitens: Reibungsblindheit. Ein ungedämpftes Mean-Variance-Modell schichtet bei minimalen Schätzänderungen das gesamte Portfolio um. Ohne Berücksichtigung von Gebühren, Spreads und Steuern frisst der Umschlag (*turnover*) den theoretischen Mehrertrag auf.

```
#!/usr/bin/env python3

# VaR_CVaR_Demo.py
"""
Kapitel CVaR: Fat Tails, VaR und CVaR anschaulich.

Teil 1: Wie oft treten "unmoegliche" Tage wirklich auf?
Teil 2: Warum ist der VaR nicht subadditiv - ein Gegenbeispiel zum Nachrechnen.
"""

import numpy as np
from scipy import stats


def var_quantil(verluste: np.ndarray, alpha: float = 0.95) -> float:
    """VaR = Quantil der Verlustverteilung (Verluste positiv, Gewinne negativ)."""
    return float(np.quantile(verluste, alpha))


def cvar_rockafellar(verluste: np.ndarray, alpha: float = 0.95) -> float:
    """
    CVaR ueber die Rockafellar-Uryasev-Formel:
        CVaR = min_gamma { gamma + 1/(1-alpha) * E[max(Verlust - gamma, 0)] }
```

```
    WICHTIG: Der naheliegende Weg "Mittelwert aller Werte >= VaR" ist FALSCH,
    sobald die Verteilung Atome hat (z. B. genau zwei moegliche Verluste).
    Dann liegt der VaR selbst auf einem Atom, und der Vergleich '>=' erfasst
    zu viel Wahrscheinlichkeitsmasse. Die Formel unten behandelt das korrekt -
    und ist zugleich genau der Ausdruck, den wir im Abschnitt 'Value at Risk
    und Conditional Value at Risk' optimieren.
    """
    kandidaten = np.unique(verluste)          # Optimum liegt immer auf einem Datenpunkt
    return float(min(g + np.mean(np.maximum(verluste - g, 0.0)) / (1.0 - alpha)
                     for g in kandidaten))


if __name__ == "__main__":
    rng = np.random.default_rng(2026)

    # --- Teil 1: Fat Tails (analytisch, nicht simuliert) -----------------
    print("=" * 88)
    print("  TEIL 1: WIE OFT TRITT DAS 'UNMOEGLICHE' EIN?")
    print("=" * 88)
    print("Vergleich: Normalverteilung gegen t-Verteilung mit 3 Freiheitsgraden")
    print("(beide auf Standardabweichung 1 normiert).\n")

    t_verteilung = stats.t(df=3)
    skalierung = t_verteilung.std()            # auf Varianz 1 bringen

    print(f"{'Ereignis':<22} {'Normal':>14} {'t (df=3)':>14} {'Faktor':>11} "
          f"{'Normal: 1 Tag in':>18}")
    print("-" * 88)
    for k in [3, 4, 5, 6]:
        p_normal = 2 * stats.norm.sf(k)                    # beidseitig
        p_t = 2 * t_verteilung.sf(k * skalierung)
        print(f"Abweichung > {k} Sigma  {p_normal*100:>13.6f} % {p_t*100:>13.6f} % "
              f"{p_t/p_normal:>10.1f}x {1/p_normal/252:>15,.0f} Jahre")

    print("\nDeutung: Ein 5-Sigma-Tag ist unter Normalverteilung ein Ereignis von")
    print("etwa einmal in 6.900 Jahren. Reale Aktienmaerkte liefern solche Tage")
    print("mehrfach pro Jahrzehnt. Wer allein mit Varianz steuert, plant fuer")
    print("eine Welt, in der Crashs praktisch nicht vorkommen.")

    # --- Teil 2: VaR ist nicht subadditiv --------------------------------
    print("\n" + "=" * 88)
    print("  TEIL 2: WARUM DER VaR KEIN KOHAERENTES RISIKOMASS IST")
    print("=" * 88)
    print("Zwei unabhaengige Anleihen, je 100 EUR Nominal.")
    print("Jede faellt mit 4 % Wahrscheinlichkeit aus (Verlust 100),")
    print("sonst zahlt sie 2 EUR Kupon (Verlust -2).\n")
```

```
    ziehungen = 2_000_000
    verlust_a = np.where(rng.random(ziehungen) < 0.04, 100.0, -2.0)
    verlust_b = np.where(rng.random(ziehungen) < 0.04, 100.0, -2.0)
    verlust_ab = verlust_a + verlust_b

    print(f"{'':<28} {'VaR 95%':>12} {'CVaR 95%':>12}")
    print("-" * 88)
    werte = {}
    for name, v in [("Anleihe A allein", verlust_a),
                    ("Anleihe B allein", verlust_b),
                    ("Portfolio A+B", verlust_ab)]:
        werte[name] = (var_quantil(v), cvar_rockafellar(v))
        print(f"{name:<28} {werte[name][0]:>12.2f} {werte[name][1]:>12.2f}")

    var_summe = werte["Anleihe A allein"][0] + werte["Anleihe B allein"][0]
    cvar_summe = werte["Anleihe A allein"][1] + werte["Anleihe B allein"][1]
    print(f"{'Summe der Einzelwerte':<28} {var_summe:>12.2f} {cvar_summe:>12.2f}")

    var_port, cvar_port = werte["Portfolio A+B"]
    print("-" * 88)
    print(f"VaR:  Portfolio {var_port:7.2f}  vs. Summe {var_summe:7.2f}  -> "
          f"{'VERLETZT die Subadditivitaet!' if var_port > var_summe else 'subadditiv'}")
    print(f"CVaR: Portfolio {cvar_port:7.2f}  vs. Summe {cvar_summe:7.2f}  -> "
          f"{'subadditiv (kohaerent)' if cvar_port <= cvar_summe + 1e-6 else 'verletzt'}")

    print("\nDeutung: Einzeln betrachtet meldet der VaR fuer jede Anleihe einen")
    print("GEWINN von 2 EUR - denn mit 96 % Wahrscheinlichkeit passiert nichts,")
    print("und 4 % liegen unterhalb der 5-%-Schwelle. Im Portfolio steigt die")
    print("Wahrscheinlichkeit mindestens eines Ausfalls auf 7,8 % und damit UEBER")
    print("die Schwelle - der VaR springt auf 98. Er behauptet also, Streuung")
    print("habe das Risiko erhoeht. Das ist oekonomisch unsinnig und der Grund,")
    print("warum die Bankenaufsicht mit Basel III auf den Expected Shortfall")
    print("umgestellt hat.")
    print("=" * 88)
```

Erwartete Ausgabe (gekürzt):

```
========================================================================================
  TEIL 1: WIE OFT TRITT DAS 'UNMOEGLICHE' EIN?
========================================================================================
Ereignis                        Normal       t (df=3)      Faktor   Normal: 1 Tag in
----------------------------------------------------------------------------------------
Abweichung > 3 Sigma       0.269980 %     1.384683 %        5.1x               1 Jahre
Abweichung > 4 Sigma       0.006334 %     0.616537 %       97.3x              63 Jahre
Abweichung > 5 Sigma       0.000057 %     0.323904 %     5649.8x           6,922 Jahre
Abweichung > 6 Sigma       0.000000 %     0.190127 %   963561.0x       2,011,101 Jahre

========================================================================================
```

```
  TEIL 2: WARUM DER VaR KEIN KOHAERENTES RISIKOMASS IST
================================================================================
                                  VaR 95%     CVaR 95%
--------------------------------------------------------------------------------
Anleihe A allein                    -2.00        79.59
Anleihe B allein                    -2.00        79.66
Portfolio A+B                       98.00       101.33
Summe der Einzelwerte               -4.00       159.24
--------------------------------------------------------------------------------
VaR:  Portfolio   98.00  vs. Summe   -4.00  -> VERLETZT die Subadditivitaet!
CVaR: Portfolio  101.33  vs. Summe  159.24  -> subadditiv (kohaerent)
================================================================================
```

Die 6-Sigma-Zeile setzt die Sache ins Verhältnis: Unter Normalverteilung wäre ein solcher Tag ein Ereignis von einmal in **zwei Millionen Jahren**. Unter der t-Verteilung mit drei Freiheitsgraden — die realen Aktienrenditen deutlich näher kommt — passiert er etwa alle **zwei Jahre**. Der Faktor beträgt fast **eine Million**.

Und Teil 2 zeigt das Grundproblem des VaR an einem Beispiel, das Sie von Hand nachrechnen können: Einzeln meldet er für jede Anleihe einen *Gewinn* von 2 €, im Portfolio einen *Verlust* von 98 €. Diversifikation hätte demnach das Risiko um 102 € erhöht. Der CVaR dagegen verhält sich korrekt: 101,33 € im Portfolio gegenüber 159,24 € bei getrennter Betrachtung — die Streuung **senkt** das Risiko, wie es sein muss.

> **Eine Falle bei der CVaR-Berechnung** Der naheliegende Weg — „Mittelwert aller Verluste ≥ VaR“ — ist **falsch**, sobald die Verteilung **Atome** hat (also einzelne Werte mit positiver Wahrscheinlichkeit, wie hier die zwei möglichen Ausgänge). Der VaR liegt dann selbst auf einem Atom, und der Vergleich `>=` erfasst zu viel Wahrscheinlichkeitsmasse. Im Beispiel oben liefert dieser naive Schätzer für Anleihe A den Wert 2,08 statt der korrekten 79,59 — ein Fehler um Faktor 38.
>
> Die Rockafellar-Uryasev-Formel behandelt Atome von sich aus korrekt. Verwenden Sie sie auch dann, wenn Sie „nur schnell“ einen CVaR ausrechnen wollen.

20.4 Value at Risk und Conditional Value at Risk

Value at Risk (VaR_α): Der Verlust, der mit Wahrscheinlichkeit α nicht überschritten wird.

Conditional Value at Risk (CVaR_α, auch *Expected Shortfall*): Der **durchschnittliche Verlust in den schlimmsten** $(1-\alpha)$ **Prozent** der Fälle.

> **Der Unterschied in einem Satz** Der VaR sagt: *„In 95 % der Tage verlieren Sie höchstens 1,86 %.“* Der CVaR sagt: *„Und wenn es doch schiefgeht, verlieren Sie im Mittel 2,99 %.“*
>
> Beide Zahlen stammen aus der Verteilung oben — nachzurechnen mit `bilder_04/erzeuge_var_cvar.py`.
>
> Der VaR ist eine **Schwelle**, der CVaR ein **Mittelwert dahinter**. Der VaR sagt nichts darüber, wie schlimm es hinter der Schwelle wird — ob dort 2,4 % oder 40 % stehen, ist ihm gleich.

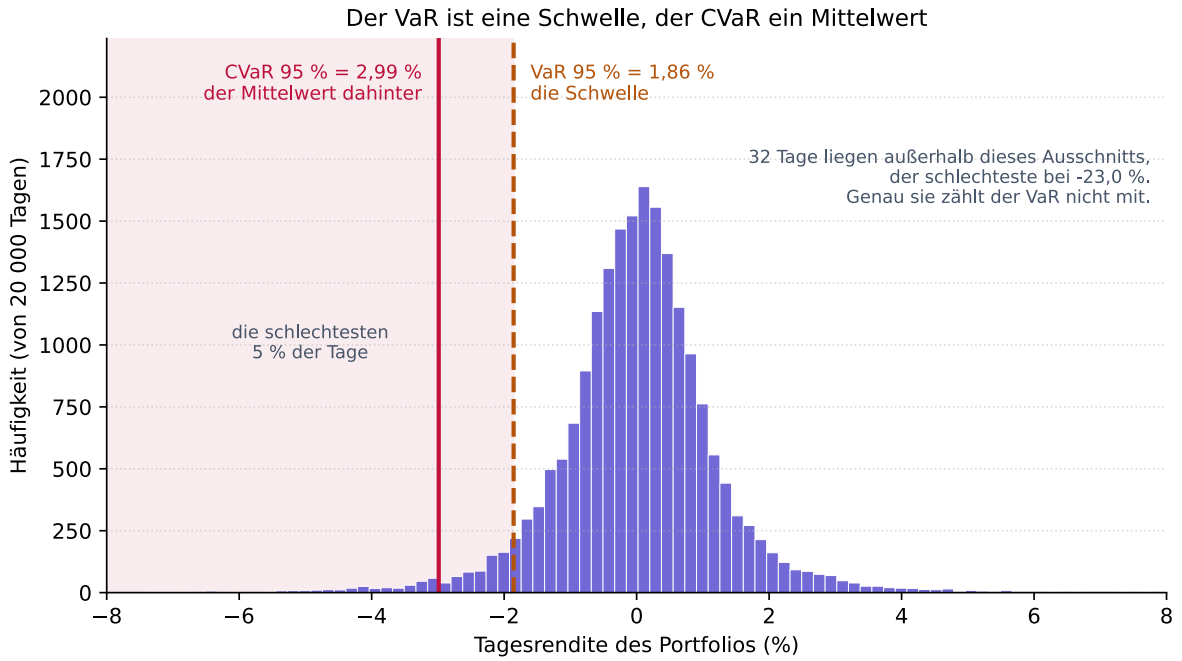

Abbildung 26: Abb. 20.1: Der VaR ist eine Schwelle, der CVaR ein Mittelwert. 20 000 simulierte Tagesrenditen aus dem Fat-Tail-Modell dieses Kapitels (t-Verteilung mit drei Freiheitsgraden). Der schraffierte Bereich sind die schlechtesten 5 % der Tage; der VaR markiert nur ihren Rand, der CVaR ihren Mittelwert. Erzeugt von `bilder_04/erzeuge_var_cvar.py`.

In der Stichprobe oben liegt der schlechteste Tag bei **−23,0 %**; am VaR von 1,86 % ändert dieser eine Tag **nichts**, am CVaR sehr wohl.

Eigenschaft	VaR	CVaR
Berücksichtigt Verlusthöhe im Rand	☐ nein	☐ ja
Subadditiv ($\rho(A+B) \le \rho(A)+\rho(B)$)	☐ nein	☐ ja
Kohärentes Risikomaß	☐ nein	☐ ja
Konvex und optimierbar	☐ nein	☐ ja
Regulatorischer Standard	bis Basel II	**ab Basel III**

☐ **Eine verbreitete Ungenauigkeit** Der CVaR wird manchmal „**streng** konvex“ genannt. Das ist zu stark: Der CVaR ist konvex, in der Szenario-Darstellung sogar **stückweise linear** — und damit gerade **nicht** streng konvex. Genau das ist sein praktischer Vorteil: Stückweise linear heißt, er lässt sich als **lineares Programm** lösen.

Das Rockafellar-Uryasev-Theorem (2000)

Rockafellar und Uryasev zeigten, dass sich der CVaR über S Szenarien exakt als lineares Programm formulieren lässt:

$$\mathrm{CVaR}_{\alpha}(\mathbf{w}) = \min_{\gamma,\, \mathbf{u} \geq 0} \quad \gamma + \frac{1}{S(1-\alpha)} \sum_{s=1}^{S} u_s$$

$$\text{u. d. N.} \qquad u_s \;\geq\; -\mathbf{R}_s^{\top}\mathbf{w} - \gamma \quad \forall s, \qquad u_s \geq 0 \quad \forall s$$

 Formel-Lesehilfe — der Trick in drei Schritten * γ ist eine **Hilfsvariable**, die im Optimum automatisch den VaR annimmt. Man muss ihn also **nicht vorher kennen** — das ist der eigentliche Durchbruch. * $-\mathbf{R}_s^{\top}\mathbf{w}$ ist der **Verlust** im Szenario s (Rendite mit negativem Vorzeichen). * $u_s \geq \text{Verlust}_s - \gamma$ zusammen mit $u_s \geq 0$ bedeutet: $u_s = \max(\text{Verlust}_s - \gamma,\ 0)$ — der **Überschuss über die Schwelle**. Liegt der Verlust unter γ, ist $u_s = 0$ und das Szenario zählt nicht.

Ohne Formel gesagt: „Wähle eine Schwelle γ. Zähle für jedes Szenario, wie weit der Verlust darüber hinausgeht. Der CVaR ist die Schwelle plus der gemittelte Überschuss — und zwar für diejenige Schwelle, bei der diese Summe minimal wird."

Warum u_s automatisch das Maximum wird: Die Zielfunktion minimiert die Summe der u_s. Jedes u_s wird also so klein wie möglich gedrückt — bis an die Grenze, die die beiden Ungleichungen erlauben. Das ist genau das Maximum der beiden Untergrenzen.

 Formel-Übersetzer: jedes Zeichen der Zielfunktion

Die Lesehilfe oben erklärt den *Trick*. Hier steht, was die einzelnen Zeichen **bedeuten** — vor allem der Bruch, an dem die meisten Leser hängen bleiben:

Mathematik	Alltagssprache
α	Das **Konfidenzniveau**, üblich 0,95 oder 0,99. *Nicht* der Randanteil.
$1-\alpha$	Der **Randanteil**: die schlechtesten 5 % (bzw. 1 %) der Fälle. Das ist der Teil, um den es geht.
S	Die Anzahl der durchgerechneten Szenarien — Handelstage, Simulationsläufe, historische Perioden.
$S(1-\alpha)$	**Wie viele Szenarien im Rand liegen.** Bei $S = 2\,000$ und $\alpha = 0{,}95$: genau 100.
$\frac{1}{S(1-\alpha)} \sum_s u_s$	Kein Mittelwert über *alle* Szenarien, sondern über die **Randszenarien allein**. Genau deshalb steht dort nicht $\frac{1}{S}$.
γ	Die Schwelle, ab der ein Verlust zum Randfall wird — im Optimum der VaR.

Mathematik	Alltagssprache
$\gamma + \frac{1}{S(1-\alpha)} \sum_s u_s$	„Schwelle **plus** durchschnittlicher Überschuss darüber“ — und das ist der CVaR.

In einem Satz: *Der CVaR ist der Mittelwert der schlimmsten* $1-\alpha$ *Prozent — der Bruch vor der Summe sorgt allein dafür, dass durch die richtige Anzahl geteilt wird.*

 Die teuerste Verwechslung des Kapitels > α und $1-\alpha$ zu vertauschen führt zu **keiner Fehlermeldung**. Das Modell > rechnet weiter, nur eben über die falsche Menge. An 2 000 simulierten Tagesrenditen mit > Fat Tails gemessen: > > | Rechnung | gemittelt über | CVaR | > | — | — | — | > | richtig ($\alpha = 0{,}95$, Rand 5 %) | 100 von 2 000 Szenarien | **3,43 %** | > | vertauscht (Rand 95 %) | 1 900 von 2 000 Szenarien | **0,18 %** | > > Ein Faktor **19** — und die zweite Zahl ist kein schlecht geschätztes Randrisiko, sondern > überhaupt kein Randmaß mehr: Sie mittelt über fast alle Tage und schließt die schlimmen > gerade nicht ein. Der größte Einzelverlust der Stichprobe beträgt 21,4 %. > > **Die Gegenprobe kostet eine Zeile:** Ist der berechnete CVaR nicht deutlich größer als > der mittlere Verlust, stimmt α nicht.

20.5 Transaktionskosten über die L_1-Norm

Sei $\mathbf{w}_{\text{alt}}$ das bestehende Portfolio und $\mathbf{w}$ das neue Ziel. Der **Umschlag** (*turnover*) ist:

$$\text{Turnover} = \sum_{i=1}^{N} |w_i - w_{\text{alt},i}| = \|\mathbf{w} - \mathbf{w}_{\text{alt}}\|_1$$

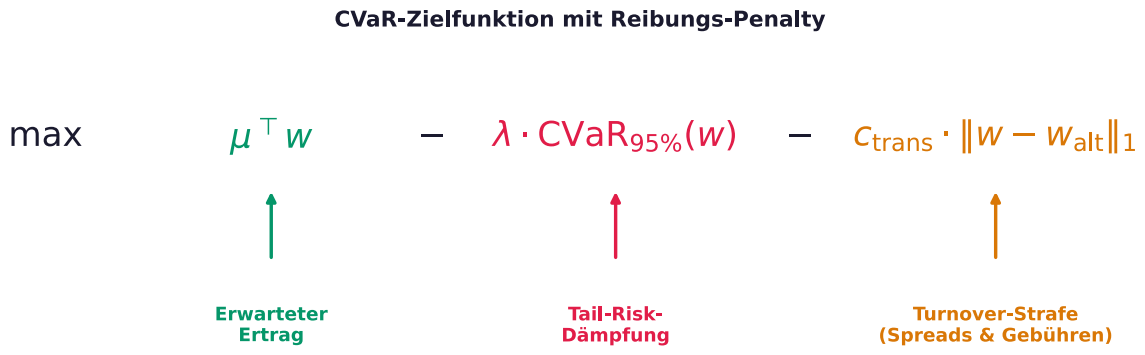

Abbildung 27: Abb. 20.2: CVaR-Zielfunktion mit Reibungs-Penalty

 Formel-Lesehilfe Die L_1-Norm ist die Summe der **Beträge**. Sie misst, wie viel Prozent des Portfolios insgesamt bewegt werden — Käufe und Verkäufe zusammen.

Ohne Formel gesagt: „Wenn du 5 % von A verkaufst und 5 % von B kaufst, hast du 10 % Umschlag und zahlst darauf Gebühren.“

Warum L_1 und nicht L_2? Die L_1-Norm ist konvex (also optimierbar) und erzeugt zusätzlich **dünn besetzte Änderungen**: Sie bevorzugt wenige große Umschichtungen gegenüber vielen kleinen. Das entspricht genau dem, was man in der Praxis will — nicht 50 Kleinstorders mit je 3 € Mindestgebühr.

In CVXPY schreibt man einfach `cp.norm1(w - w_alt)`; intern wird das in lineare Hilfsvariablen zerlegt.

20.6 Implementierung: CVaR-Portfolio mit Reibung

Einheiten konsistent halten

Ein häufiger Fehler verrechnet eine **annualisierte** Rendite gegen einen **täglichen** CVaR:

```
mu = returns_df.mean().values * 252              # ANNUALISIERT
cvar = gamma + (1/(S*(1-alpha))) * cp.sum(u)     # TAEGLICH
objective = cp.Maximize(mu @ w - lambda_risk * cvar - trans_costs)
```

Der Risikoterm ist dadurch faktisch um Faktor 252 zu schwach gewichtet: `lambda_risk = 1.5` bedeutet dann effektiv eine Risikoaversion von $1{,}5/252 \approx 0{,}006$. Das Modell ist nicht falsch im Sinne von unlösbar — aber der Parameter bedeutet nicht, was er zu bedeuten scheint, und lässt sich daher nicht sinnvoll einstellen.

Dieses Programm rechnet deshalb durchgehend auf Tagesbasis und annualisiert erst in der Ausgabe.

```
#!/usr/bin/env python3

# CVaR_Portfolio.py
"""
Kapitel CVaR: CVaR-Optimierung mit L1-Transaktionskosten via CVXPY.

Eigenschaften:
  * Spaltenreihenfolge erzwungen
  * Einheiten konsistent (alles taeglich, Annualisierung nur in der Ausgabe)
  * Szenario-Nebenbedingungen VEKTORISIERT statt in einer Python-Schleife
    (eine Matrixbedingung statt S einzelner Constraints - deutlich schneller)
  * Vergleich CVaR- gegen Varianz-Optimierung
  * Nachrechnung von VaR/CVaR aus den realisierten Szenarien
"""

import time

import cvxpy as cp
import numpy as np
import pandas as pd
import yfinance as yf

TICKER = ["AAPL", "MSFT", "NVDA", "AMZN", "JNJ", "PFE", "JPM", "GS", "XOM", "CVX"]
```

```
ALPHA = 0.95            # Konfidenzniveau: schlechteste 5 % der Tage
MAX_GEWICHT = 0.25
GEBUEHRENSATZ = 0.002   # 0,2 % je Einheit Turnover (Spread + Brokerage)
RISIKOAVERSION = 1.5    # bezogen auf TAEGLICHE Groessen
HANDELSTAGE = 252


def lade_renditen():
    ende = pd.Timestamp.today().normalize()
    start = ende - pd.DateOffset(years=2)
    roh = yf.download(TICKER, start=start, end=ende, auto_adjust=True, progress=False)
    if roh.empty:
        raise SystemExit("Download fehlgeschlagen (Netz, Ticker oder Rate-Limit pruefen).")

    if isinstance(roh.columns, pd.MultiIndex):
        kurse = roh["Close"][TICKER].dropna()        # erzwingt eigene Spaltenreihenfolge
    else:
        kurse = roh[["Close"]].dropna()
        kurse.columns = TICKER
    assert list(kurse.columns) == TICKER, "Spaltenreihenfolge weicht ab!"
    return kurse.pct_change().dropna()


def optimiere_cvar(R, w_alt, vektorisiert=True):
    """
    Maximiere:  taegliche Rendite - lambda * CVaR - Transaktionskosten
    Alle Groessen TAEGLICH.
    """
    S, N = R.shape
    mu_taeglich = R.mean(axis=0)

    w = cp.Variable(N, nonneg=True)
    gamma = cp.Variable()                  # wird im Optimum zum VaR
    u = cp.Variable(S, nonneg=True)        # Ueberschuss ueber die Schwelle

    cvar = gamma + (1.0 / (S * (1.0 - ALPHA))) * cp.sum(u)
    turnover = cp.norm1(w - w_alt)
    kosten = GEBUEHRENSATZ * turnover

    ziel = cp.Maximize(mu_taeglich @ w - RISIKOAVERSION * cvar - kosten)

    bedingungen = [cp.sum(w) == 1, w <= MAX_GEWICHT]
    if vektorisiert:
        # EINE Matrixbedingung statt S einzelner - deutlich schneller
        bedingungen.append(u >= -(R @ w) - gamma)
    else:
        for s in range(S):                 # Alternative: S einzelne Constraints (langsamer)
            bedingungen.append(u[s] >= -R[s] @ w - gamma)
```

```
    problem = cp.Problem(ziel, bedingungen)
    problem.solve()
    if problem.status not in ("optimal", "optimal_inaccurate"):
        raise SystemExit(f"CVaR-Problem nicht loesbar: {problem.status}")
    return w.value, float(gamma.value), float(cvar.value), float(turnover.value)


def optimiere_varianz(R, w_alt):
    """Klassisches Mean-Variance zum Vergleich - ebenfalls taeglich gerechnet.

    ACHTUNG, DCP-Falle: Die Standardabweichung ist hier NICHT als
    cp.sqrt(cp.quad_form(w, sigma)) formulierbar. cp.sqrt ist konkav und
    verlangt ein konkaves Argument; quad_form ist konvex - CVXPY lehnt den
    Ausdruck mit einem DCPError ab, und zwar voellig zu Recht (Anhang
    Fehlerdiagnose). cp.psd_wrap() hilft dagegen nicht: Es behebt eine
    NUMERISCHE Beanstandung an sigma, keine Regelverletzung im Aufbau.

    Der Standardweg ist die Cholesky-Zerlegung sigma = L L^T. Damit gilt
    w' sigma w = ||L^T w||^2, also ist die Standardabweichung die 2-Norm
    eines AFFINEN Ausdrucks - konvex und damit regelkonform.
    """
    N = R.shape[1]
    mu_taeglich = R.mean(axis=0)
    sigma = np.cov(R, rowvar=False, ddof=1)
    # Der winzige Diagonalzuschlag faengt den Fall ab, dass sigma numerisch
    # nur halbdefinit ist (mehr Titel als Handelstage, doppelte Spalten).
    L = np.linalg.cholesky(sigma + 1e-12 * np.eye(N))

    w = cp.Variable(N, nonneg=True)
    ziel = cp.Maximize(mu_taeglich @ w
                       - RISIKOAVERSION * cp.norm2(L.T @ w)
                       - GEBUEHRENSATZ * cp.norm1(w - w_alt))
    problem = cp.Problem(ziel, [cp.sum(w) == 1, w <= MAX_GEWICHT])
    problem.solve()
    if problem.status not in ("optimal", "optimal_inaccurate"):
        raise SystemExit(f"Varianz-Problem nicht loesbar: {problem.status}")
    return w.value


def realisierte_kennzahlen(w, R):
    """VaR und CVaR direkt aus den Szenarien - unabhaengige Gegenprobe."""
    portfoliorenditen = R @ w
    verluste = -portfoliorenditen
    var = float(np.quantile(verluste, ALPHA))
    cvar = float(verluste[verluste >= var].mean())
    return var, cvar, float(portfoliorenditen.mean()), float(portfoliorenditen.std(ddof=1))
```

```
if __name__ == "__main__":
    renditen = lade_renditen()
    R = renditen.values
    S, N = R.shape
    w_alt = np.ones(N) / N                # Ausgangslage: Gleichgewichtung

    t0 = time.perf_counter()
    w_cvar, var_modell, cvar_modell, turnover = optimiere_cvar(R, w_alt, vektorisiert=True)
    dauer_vektor = time.perf_counter() - t0

    w_var = optimiere_varianz(R, w_alt)

    print("=" * 90)
    print("         CVaR-PORTFOLIO-OPTIMIERUNG MIT TRANSAKTIONSKOSTEN")
    print("=" * 90)
    print(f"Datenbasis: {S} Handelstage, {N} Titel | Konfidenzniveau "
          f"{ALPHA*100:.0f} % | Loesungszeit {dauer_vektor:.2f} s\n")

    # --- Gegenprobe: Modellwerte gegen realisierte Szenariowerte ---------
    var_real, cvar_real, mu_real, sd_real = realisierte_kennzahlen(w_cvar, R)
    print("--- Gegenprobe: stimmen Modell und Szenarien ueberein? ---")
    print(f"  VaR  aus dem Modell (gamma): {var_modell*100:7.4f} %  |  "
          f"aus den Szenarien: {var_real*100:7.4f} %")
    print(f"  CVaR aus dem Modell:         {cvar_modell*100:7.4f} %  |  "
          f"aus den Szenarien: {cvar_real*100:7.4f} %")
    assert abs(cvar_modell - cvar_real) < 1e-4, "CVaR stimmt nicht mit den Szenarien!"
    print("  -> Der Rockafellar-Uryasev-Trick liefert exakt den empirischen CVaR.")

    # --- Kennzahlen beider Portfolios ------------------------------------
    print(f"\n{'Portfolio':<24} {'Rendite p.a.':>13} {'Vola p.a.':>11} "
          f"{'VaR 95% (Tag)':>15} {'CVaR 95% (Tag)':>16} {'Turnover':>10}")
    print("-" * 90)
    for name, w in [("CVaR-optimiert", w_cvar), ("Varianz-optimiert", w_var),
                    ("Gleichgewichtung", w_alt)]:
        v, c, m, s = realisierte_kennzahlen(w, R)
        to = float(np.abs(w - w_alt).sum())
        print(f"{name:<24} {m*HANDELSTAGE*100:>12.2f} % "
              f"{s*np.sqrt(HANDELSTAGE)*100:>10.2f} % {v*100:>14.3f} % "
              f"{c*100:>15.3f} % {to*100:>9.1f} %")

    print("\nHinweis zur Annualisierung: Renditen werden mit 252 skaliert,")
    print("Volatilitaeten mit sqrt(252). Fuer VaR/CVaR ist eine solche Skalierung")
    print("nur unter starken Annahmen (Unabhaengigkeit, kein Drift) zulaessig -")
    print("sie werden hier deshalb bewusst als TAGESwerte ausgewiesen.")

    # --- Allokationstabelle ----------------------------------------------
    print("\n--- Allokation ---")
```

```
    tabelle = pd.DataFrame({
        "Ticker": TICKER,
        "vorher": [f"{v*100:5.1f} %" for v in w_alt],
        "CVaR-opt.": [f"{v*100:5.1f} %" for v in w_cvar],
        "Handel": [f"{(w_cvar[i]-w_alt[i])*100:+6.1f} %" for i in range(N)],
        "Varianz-opt.": [f"{v*100:5.1f} %" for v in w_var],
    })
    print(tabelle.to_string(index=False))
    print(f"\nTurnover {turnover*100:.1f} % -> Transaktionskosten "
          f"{GEBUEHRENSATZ*turnover*100:.3f} % des Portfoliowerts")

    # --- Laufzeitvergleich vektorisiert vs. Schleife ----------------------
    if S <= 600:                      # bei sehr vielen Szenarien zu langsam
        t0 = time.perf_counter()
        optimiere_cvar(R, w_alt, vektorisiert=False)
        dauer_schleife = time.perf_counter() - t0
        print(f"\n--- Laufzeit: {S} Nebenbedingungen aufbauen ---")
        print(f"  vektorisiert (u >= -(R @ w) - gamma): {dauer_vektor:6.2f} s")
        print(f"  Schleife ueber Szenarien (V01):       {dauer_schleife:6.2f} s "
              f"({dauer_schleife/dauer_vektor:.1f}x langsamer)")
    print("=" * 90)
```

□ Code-Durchgang: Vektorisierung

Die entscheidende Zeile ist

```
bedingungen.append(u >= -(R @ w) - gamma)     # EINE Matrixbedingung
```

statt

```
for s in range(S):
    bedingungen.append(u[s] >= -R[s] @ w - gamma)    # S einzelne Bedingungen
```

Beide beschreiben **dasselbe Modell**. Der Unterschied liegt im Aufbau: CVXPY muss im zweiten Fall 500 einzelne Ausdrucksbäume erzeugen, prüfen und zusammensetzen. Bei einem Backtest mit 60 Rebalancings (Kapitel 21) summiert sich das erheblich.

Die allgemeine Regel: Wenn Sie in einer Modellierungssprache eine Python-Schleife über Datenzeilen schreiben, prüfen Sie, ob sich dasselbe als Matrixoperation ausdrücken lässt. Der Solver rechnet ohnehin mit Matrizen — der Umweg über einzelne Ausdrücke kostet nur Aufbauzeit.

□ Die Standardabweichung ist in CVXPY keine Wurzel

Im Vergleichsmodell `optimiere_varianz()` steht die Standardabweichung als

```
L = np.linalg.cholesky(sigma + 1e-12 * np.eye(N))
risiko = cp.norm2(L.T @ w)                    # = sqrt(w' sigma w)
```

und **nicht** als das Naheliegende, `cp.sqrt(cp.quad_form(w, sigma))`. Der naheliegende Ausdruck ist nicht DCP und wird von CVXPY mit einem `DCPError` abgelehnt: `cp.sqrt` ist konkav und verlangt deshalb ein **konkaves** Argument, `quad_form` ist aber konvex. Die Regel wird verletzt, obwohl die Funktion als ganze mathematisch völlig harmlos ist — $\sqrt{w^\top \Sigma w}$ ist konvex, DCP kann es nur nicht *sehen*.

Wichtig ist die Fehlerdiagnose dahinter, weil sie leicht in die Irre geht: `cp.psd_wrap(sigma)` sieht nach der Lösung aus und ist keine. Es unterdrückt die **numerische** Beanstandung, dass Σ nicht als positiv semidefinit erkannt wird — gegen die **strukturelle** Regelverletzung im Aufbau hilft es nicht.

Der Ausweg ist immer derselbe: Zerlegen Sie $\Sigma = LL^\top$ (Cholesky). Dann ist $w^\top \Sigma w = \|L^\top w\|_2^2$, und die Standardabweichung wird zur 2-Norm eines **affinen** Ausdrucks — konvex, regelkonform und für den Solver sogar die bessere Formulierung, weil sie direkt ein Kegelproblem ist. Der winzige Diagonalzuschlag 10^{-12} fängt den Fall ab, dass Σ numerisch nur halbdefinit ist (mehr Titel als Handelstage, duplizierte Spalten). Die vollständige Fehlertabelle steht in Anhang C.

 Zur Annualisierung von VaR und CVaR Ein häufiger Fehler skaliert den täglichen CVaR mit $\sqrt{252}$. Diese **Wurzel-Zeit-Regel** gilt streng nur für **Standardabweichungen** unabhängig identisch verteilter Größen ohne Drift. Der CVaR ist ein Erwartungswert über einen Verteilungsrand — für ihn ist die Regel eine grobe Näherung, die bei fetten Rändern und Autokorrelation systematisch danebenliegt.

Dieses Buch weist VaR und CVaR deshalb als Tageswerte aus und benennt die Skalierungsproblematik ausdrücklich. Wer Mehrtageshorizonte braucht, simuliert sie (Monte-Carlo aus Kapitel 12) statt zu skalieren.

20.7 Übungsaufgaben

Lösungen: Abschnitt A.20.

Aufgabe 20.1 — **VaR und CVaR ablesen.** Zehn Tagesverluste (in %): −1,2, 0,3, 2,1, −0,8, 5,4, 1,1, −2,0, 0,6, 8,9, −0,4 (positiv = Verlust). Bestimmen Sie $\mathrm{VaR}_{80\%}$ und $\mathrm{CVaR}_{80\%}$ von Hand.

Aufgabe 20.2 — **Warum kohärent?** Erklären Sie in eigenen Worten, warum ein Risikomaß subadditiv sein sollte. Was bedeutet es wirtschaftlich, wenn diese Eigenschaft verletzt ist?

Aufgabe 20.3 — **Rockafellar-Uryasev nachvollziehen.** Zeigen Sie für die drei Szenarien mit Verlusten $(1, 4, 9)$ und $\alpha = 2/3$: (a) Was ist der empirische CVaR? (b) Werten Sie $\gamma + \frac{1}{S(1-\alpha)} \sum_s \max(\text{Verlust}_s - \gamma, 0)$ für $\gamma \in \{0, 1, 4, 5, 9\}$ aus. (c) Bei welchem γ ist der Ausdruck minimal, und stimmt das Minimum mit (a) überein?

Aufgabe 20.4 — **Einheiten prüfen.** Angenommen, ein Modell verrechnet Jahresrendite gegen Tages-CVaR (siehe die Warnung in Abschnitt 20.6). Berechnen Sie: Welchem „effektiven“ täglichen λ entspräche `lambda_risk = 1.5` dadurch? Welchen Wert müsste man stattdessen setzen, um dieselbe Wirkung wie `RISIKOAVERSION = 1.5` im konsistenten Tagesmodell zu erzielen?

Aufgabe 20.5 □□□ — Risikoaversion kalibrieren. Variieren Sie `RISIKOAVERSION` von 0 bis 20 und tragen Sie Rendite, Volatilität, CVaR und Turnover gegeneinander auf. (a) Wie sieht die „Effizienzgrenze“ im Rendite-CVaR-Raum aus? (b) Bei welchem Wert entspricht das CVaR-Portfolio ungefähr dem Varianz-Portfolio? (c) Welchen Wert würden Sie einem konservativen Stiftungsfonds empfehlen — und wie begründen Sie ihn ohne Fachjargon?

Aufgabe 20.6 □□□ — Turnover-Grenze statt Strafe. Ersetzen Sie den Kostenterm durch eine **harte Grenze** $\|\mathbf{w} - \mathbf{w}_{\text{alt}}\|_1 \le \tau$ und variieren Sie $\tau \in \{0{,}05;\ 0{,}1;\ 0{,}25;\ 0{,}5;\ 1{,}0\}$. (a) Wie verändert sich die erreichbare Rendite? (b) Was ist der Vorteil einer Grenze gegenüber einer Strafe — und was der Nachteil? (c) Wann würden Sie was einsetzen?

20.8 Finde den Denkfehler

□ Finde den Denkfehler: Das Risikobudget, das durch Diversifikation stieg

Eine Bank vergibt Risikobudgets je Abteilung. Zwei Kreditabteilungen halten je eine Unternehmensanleihe über 100 € Nominal mit **4 % Ausfallwahrscheinlichkeit** (bei Ausfall Totalverlust, sonst 2 € Kupon). Der Risikocontroller misst für jede einzeln:

$$\mathrm{VaR}_{95}(\mathrm{A}) = -2{,}00\,€, \qquad \mathrm{VaR}_{95}(\mathrm{B}) = -2{,}00\,€$$

Ein *negativer* Verlust, also ein Gewinn. Auf dem 95-%-Niveau erscheint jede Anleihe risikolos, beide Abteilungen bekommen grünes Licht. Der Controller notiert als Gesamtrisiko die Summe: $-4{,}00\,€$.

Dann werden die Positionen in einem gemeinsamen Buch zusammengelegt. Der VaR des Gesamtbuchs beträgt **+98,00 €**.

Ihre Aufgabe:

(a) Wie kann die Zusammenlegung zweier **unabhängiger** Positionen das gemessene Risiko von $-4\,€$ auf $+98\,€$ treiben? Rechnen Sie die entscheidende Wahrscheinlichkeit aus.

(b) Warum meldet der VaR für eine einzelne Anleihe einen Gewinn, obwohl sie mit 4 % Wahrscheinlichkeit vollständig ausfällt?

(c) Welche Eigenschaft eines Risikomaßes wird hier verletzt? Was bedeutet das konkret für ein Unternehmen, das Risikobudgets auf Abteilungen verteilt und wieder zusammenzählt?

(d) Der CVaR liefert 79,59 € und 79,66 € einzeln, aber nur 101,33 € zusammen — statt der Summe von 159,24 €. Warum kann ihm das Verhalten des VaR nicht passieren?

Die Zahlen stammen aus Teil 2 von `VaR_CVaR_Demo.py` oben. Auflösung: Abschnitt A.20.

□ Eine Falle beim Nachrechnen

Wenn Sie den CVaR selbst nachrechnen wollen: Der naheliegende Weg `verluste[verluste >= var].mean()` ist **falsch**, sobald die Verteilung Atome hat — und hier hat sie genau zwei mögliche Werte. Der VaR liegt dann selbst auf einem Atom, und der Vergleich `>=` erfasst weit mehr Wahrscheinlichkeitsmasse als die vorgesehenen 5 %.

Deshalb verwendet `cvar_rockafellar()` oben die Optimierungsformel statt der naiven Mittelung. Wer stattdessen exakt abzählt, muss die schlechtesten $\lceil (1-\alpha)N \rceil$ Szenarien nehmen — nicht alle, die eine Schwelle überschreiten.

Diese Falle ist besonders unangenehm, weil das falsche Ergebnis plausibel aussieht. Sie fällt nur auf, wenn man wie hier eine **unabhängige Gegenprobe** hat: Ein CVaR, der die Subadditivität verletzt, kann nicht stimmen.

 Merksatz Ein Risikomaß, dessen Werte man nicht addieren darf, ist als Steuerungsgröße unbrauchbar — denn genau das tut jede Organisation mit Kennzahlen: Sie verteilt sie auf Einheiten und zählt sie wieder zusammen. Der CVaR ist **kohärent** und erlaubt das; der VaR nicht. Aus diesem Grund hat die Bankenaufsicht mit Basel III vom VaR auf den *Expected Shortfall* umgestellt — dieselbe Größe, die hier CVaR heißt.

20.9 Micro-Quiz

 Micro-Quiz 20: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Zwei Anlagen haben denselben VaR(95) von 3 %. Was wissen Sie über ihre Extremverluste? (a) Sie sind gleich hoch — der VaR misst genau das. (b) **Nichts.** Der VaR ist ein Quantil: Er markiert die Schwelle und sagt nichts darüber, was dahinter liegt. Die eine kann dort −3 %, die andere −40 % haben. (c) Die Anlage mit der höheren Schwankung hat auch die höheren Extremverluste.

2. Warum lässt sich der CVaR als lineares Programm minimieren, der VaR aber nicht? (a) Der VaR ist nicht differenzierbar. (b) Der CVaR lässt sich nach Rockafellar/Uryasev über eine Hilfsvariable und Schlupfterme als konvexes — sogar lineares — Problem schreiben. Die VaR-Minimierung ist dagegen nicht konvex und hat viele lokale Optima. (c) Der CVaR ist immer größer als der VaR.

3. Sie sollen das Risiko von drei Handelsbüchern zu einer Gesamtkennzahl zusammenfassen. Welches Maß nehmen Sie? (a) VaR — er ist in der Praxis verbreiteter und leichter zu erklären. (b) CVaR, weil er **subadditiv** ist: Die Gesamtkennzahl kann nie größer werden als die Summe der Einzelkennzahlen, Diversifikation wird also korrekt belohnt. (c) Beide liefern dasselbe, solange die Bücher unabhängig sind.

20.10 Selbsttest

Antworten: Anhang A.

1. Was misst der CVaR, was der VaR nicht misst?
2. Warum ist der CVaR konvex, aber nicht streng konvex?
3. Welche Rolle spielt die Hilfsvariable γ bei Rockafellar/Uryasev?
4. Warum bevorzugt die L_1-Strafe wenige große statt vieler kleiner Umschichtungen?
5. Warum sollte man tägliche und annualisierte Größen nicht in einer Zielfunktion mischen?

20.11 Zusammenfassung

- **Reale Renditen haben fette Ränder.** Ein 5-Sigma-Tag ist unter Normalverteilung ein Jahrtausendereignis — an echten Märkten passiert er alle paar Jahre.
- **Der VaR ist ein Quantil und schaut nicht dahinter.** Zwei Anlagen mit identischem VaR können Extremverluste von −3 % und −40 % haben. Der VaR sagt, **wie oft** es schiefgeht; der CVaR, **wie schlimm** es dann ist.
- **Der VaR ist nicht kohärent:** Er kann Diversifikation als Risikoerhöhung ausweisen. Deshalb die regulatorische Umstellung auf den Expected Shortfall. Praktische Folge: Man darf VaR-Kennzahlen **nicht über Einheiten addieren** — und genau das tut jede Organisation mit Kennzahlen.
- **Das bessere Maß ist hier zugleich das leichter optimierbare** — ein seltener Glücksfall. Der CVaR wird zum LP, die VaR-Minimierung bleibt nicht-konvex.
- **Rockafellar/Uryasev** machen den CVaR über eine Hilfsvariable γ linear optimierbar, ohne den VaR vorher zu kennen.
- **Transaktionskosten gehören ins Modell**, nicht in eine nachgelagerte Rechnung — sonst schichtet der Optimierer bei jedem Rauschen um.
- **Einheiten konsistent halten.** Tägliche Renditen zu täglichem Risiko; annualisiert wird erst in der Ausgabe.
- **Vektorisieren Sie Szenario-Bedingungen** — das spart bei wiederholten Läufen erheblich Zeit.

Ausblick. Kapitel 21 fügt alles zusammen: Datenpipeline, Signal, Optimierung, Rebalancing und ein Walk-Forward-Backtest ohne Lookahead-Bias.

Kapitel 21: Die vollständige quantitative Handelsmaschine

Kapitel auf einen Blick

Worum geht es? Um das Zusammenführen von allem: Datenpipeline, Signal, Optimierung, Rebalancing, Kostenverbuchung und ein Backtest, der nicht lügt.

Voraussetzungen: Kapitel 18 bis Kapitel 20.

Danach können Sie: Eine Walk-Forward-Backtest-Architektur bauen, Lookahead-Bias vermeiden, Ergebnisse gegen eine Benchmark bewerten und typische Backtest-Fallen erkennen.

Zeitbedarf: ca. 6 Stunden.

Programme:
`QuantitativeTradingEngine.py`
`Backtest_Fallen.py`
`Data_Snooping.py`

 Keine Anlageberatung. Dieses Kapitel demonstriert Methodik an realen Daten, keine handelbare Strategie. Abschnitt 21.7 erklärt ausführlich, warum ein guter Backtest noch lange keine funktionierende Strategie ist.

Notebook: `Notebooks_04/handelsmaschine.ipynb`

21.1 In 5 Minuten gelöst

 In 5 Minuten gelöst: Dieselben Tage, ein anderes Schicksal

Zwei Strategien über zehn Jahre. Sie bestehen aus **exakt denselben Tagesrenditen** — nur in anderer Reihenfolge. Bei der einen sind die 250 schlechtesten Tage über den ganzen Zeitraum verstreut, bei der anderen liegen sie am Stück.

```
import numpy as np

rng = np.random.default_rng(1)
tage = rng.normal(0.0005, 0.010, 2520)          # zehn Jahre Tagesrenditen

gestreut = tage.copy()
rng.shuffle(gestreut)

sortiert = np.sort(tage)                         # dieselben Tage, neu geordnet:
schlecht, rest = sortiert[:250], sortiert[250:]  # die 250 schlechtesten ...
rng.shuffle(rest)
geballt = np.concatenate([rest[:1000], schlecht, rest[1000:]])   # ... am Stueck

def kennzahlen(r):
    verlauf = np.cumprod(1 + r)
    rueckschlag = (verlauf / np.maximum.accumulate(verlauf) - 1).min()
    return verlauf[-1]**(252/len(r)) - 1, r.mean()/r.std()*np.sqrt(252),
     ↪  rueckschlag

for name, r in [("gestreut", gestreut), ("geballt", geballt)]:
    jahr, sharpe, rueckschlag = kennzahlen(r)
    print(f"{name:9} Jahresrendite {jahr*100:5.2f} %  Sharpe {sharpe:4.2f}  "
          f"max. Rueckschlag {rueckschlag*100:6.1f} %")
```

Ausgabe:

```
gestreut  Jahresrendite  8.88 %  Sharpe 0.61  max. Rueckschlag  -19.0 %
geballt   Jahresrendite  8.88 %  Sharpe 0.61  max. Rueckschlag  -98.7 %
```

Rendite und Sharpe-Kennzahl sind identisch — sie müssen es sein, denn beide Reihen enthalten dieselben Zahlen. Nur die Reihenfolge unterscheidet sich.

Der maximale Rückschlag beträgt trotzdem 19,0 % gegen **98,7 %**. Die zweite Strategie verliert zwischenzeitlich fast das gesamte Kapital und erholt sich danach wieder auf denselben Endstand.

Rechnerisch sind beide gleichwertig. Praktisch ist die zweite tot: Bei −98 % gibt es keinen Anleger mehr, kein Mandat und kein Handelsdesk, das die Erholung noch erlebt.

> **Merksatz** Rendite und Sharpe-Kennzahl mitteln über die Zeit und verlieren dabei die **Reihenfolge**. Genau in der Reihenfolge steckt aber, ob eine Strategie überlebt. Der maximale Rückschlag misst nicht das statistische Risiko, sondern das menschliche: *Wie schlimm sah es zwischendurch aus?* Berichten Sie ihn immer mit — eine Rendite allein ist keine Aussage.

Dasselbe gilt weit außerhalb der Finanzwelt. Ein Produktionsplan, der im Jahresmittel optimal ist, aber im März drei Wochen lang die Liefertreue reißt, wird im April abgeschaltet — bevor sich der Mittelwert einstellen kann. Wo immer Sie einen Durchschnitt berichten, berichten Sie auch den **schlimmsten Zwischenzustand**.

21.2 Lernziele

Nach diesem Kapitel können Sie …

1. … die sechs Bausteine einer quantitativen Engine benennen und ihre Reihenfolge begründen.
2. … einen Walk-Forward-Backtest ohne Lookahead-Bias implementieren.
3. … Rebalancing-Termine korrekt auf Handelstage abbilden.
4. … Transaktionskosten und Gewichtsdrift realistisch verbuchen.
5. … Backtest-Ergebnisse kritisch prüfen und die typischen Selbsttäuschungen erkennen.
6. … den maximalen Rückschlag als eigenständige Kennzahl berichten und begründen, warum Rendite und Sharpe allein nichts über den Verlauf aussagen.
7. … die Zahl der ausprobierten Varianten protokollieren und die Signifikanzschwelle entsprechend korrigieren.

21.3 Die Architektur

#	Baustein	Aufgabe	Kapitel
1	**Datenpipeline**	Kurse laden, bereinigen, Reihenfolge sichern	11
2	**Signal (Alpha)**	Renditeerwartung μ schätzen	11
3	**Risikomodell**	Kovarianz bzw. Szenarien bereitstellen	11

#	Baustein	Aufgabe	Kapitel
4	**Optimierungskern**	Gewichte unter Restriktionen bestimmen	12, 13
5	**Rebalancing**	Umsetzen, Kosten verbuchen, Drift fortschreiben	13
6	**Auswertung**	Kennzahlen, Benchmark-Vergleich, Diagnose	14

 Die goldene Regel des Backtestens Zum Zeitpunkt *t* darf ausschließlich Information verwendet werden, die zum Zeitpunkt *t* tatsächlich vorlag. Jede Verletzung — und sei sie einen einzigen Tag groß — macht Ergebnisse systematisch zu gut. Der Fehler ist besonders tückisch, weil er keine Fehlermeldung erzeugt, sondern nur die Rendite verbessert.

21.4 Rebalancing-Termine richtig bestimmen

 Ein stiller, häufiger Fehler

```
self.rebalance_dates = self.prices.resample("MS").first().index
...
if current_date in self.rebalance_dates:
```

`resample("MS")` erzeugt **Kalender-Monatsanfänge** — den 1. Januar, den 1. Februar und so fort. Der Kursindex enthält aber nur **Handelstage**. Fällt der Monatserste auf ein Wochenende oder einen Feiertag, ist er kein Element des Index, die Bedingung wird nie wahr, und das Rebalancing **entfällt kommentarlos**.

Messung auf einem reinen Wochentagskalender über fünf Jahre:

	Termine
geplant	61
tatsächlich ausgeführt	44 (72 %)
stillschweigend verworfen	**17 (28 %)**

Mit echten Börsenfeiertagen fallen weitere Termine aus. Der Backtest testete also eine **andere Strategie als die beschriebene** — und lieferte trotzdem plausibel aussehende Kennzahlen.

Die Korrektur:

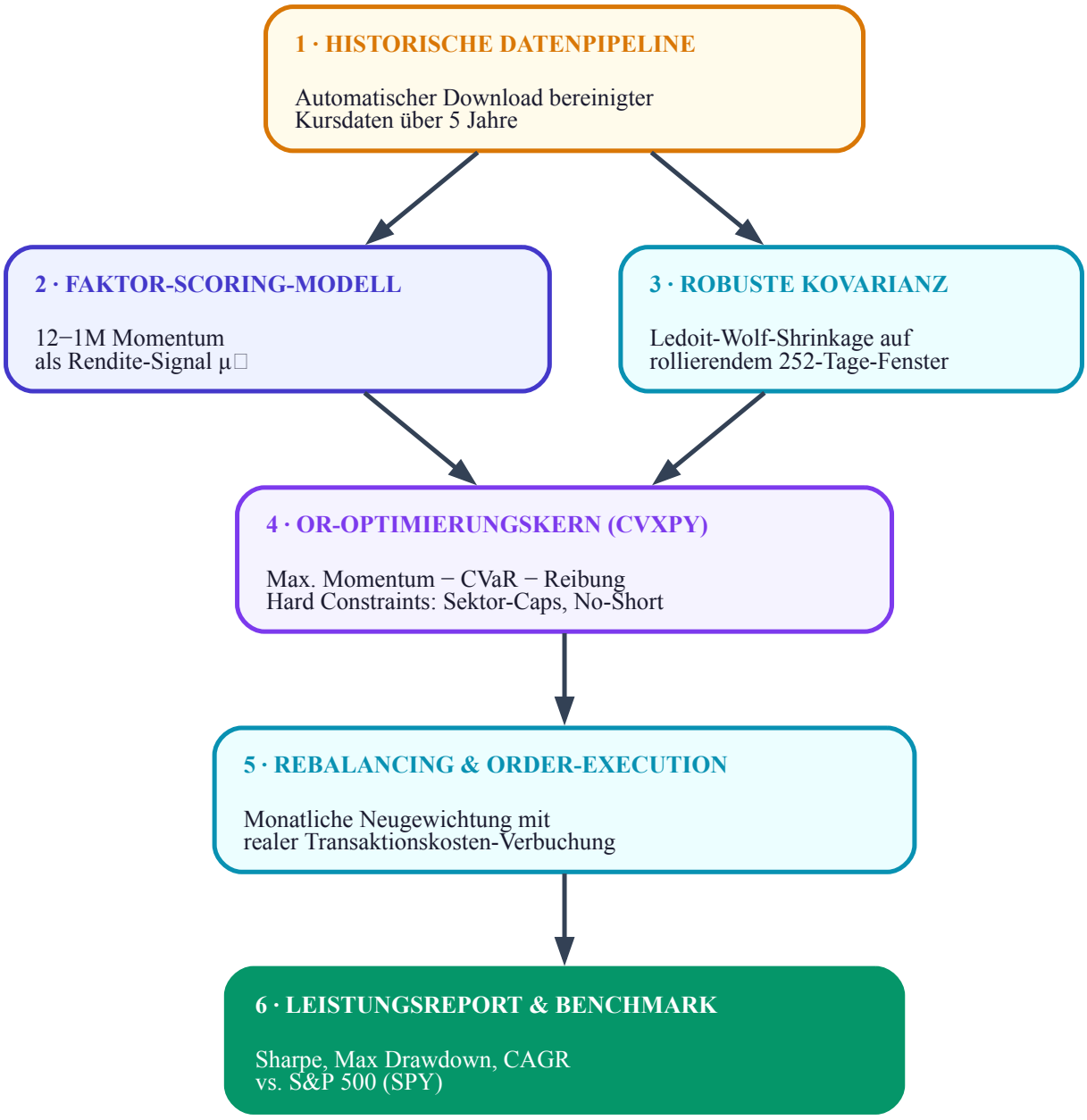

Abbildung 28: Abb. 21.1: Überblick der quantitativen Engine

```
monat = self.kurse.index.to_period("M")
self.rebalancing_termine = self.kurse.index[~monat.duplicated()]   # erster
 ↪  HANDELStag
assert self.rebalancing_termine.isin(self.kurse.index).all()
```

`~monat.duplicated()` markiert den jeweils **ersten vorhandenen** Tag jedes Monats — also garantiert einen Handelstag.

 Die allgemeine Lehre Der gefährlichste Fehler ist der, der nichts kaputt macht. Ein Absturz wird bemerkt. Ein Backtest, der stillschweigend 28 % seiner Entscheidungen auslässt, wird gelobt. Bauen Sie deshalb in jeden Backtest **Zähler** ein: Wie viele Rebalancings waren geplant, wie viele wurden ausgeführt? Weichen die Zahlen ab, stimmt etwas nicht.

21.5 Die Engine

```
#!/usr/bin/env python3

# QuantitativeTradingEngine.py
"""
Kapitel Handelsmaschine: Vollstaendige quantitative Handelsmaschine mit
Walk-Forward-Backtest und OR-Optimierungsschicht.

Eigenschaften:
  * Rebalancing-Termine sind garantiert Handelstage; die Zahl der
    tatsaechlich ausgefuehrten Umschichtungen wird ausgewiesen
  * Spaltenreihenfolge erzwungen
  * Einheiten konsistent (taegliche Groessen im Modell)
  * Szenario-Nebenbedingungen vektorisiert
  * Lookahead-Selbsttest eingebaut
  * Zusaetzliche Kennzahlen: Calmar, Trefferquote, Turnover, Kostenanteil
"""

import os

import cvxpy as cp
import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.covariance import LedoitWolf
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt

OUTPUT_DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)), "output")
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
HANDELSTAGE = 252


class QuantitativeHandelsmaschine:
    """Walk-Forward-Backtest mit CVaR-Optimierung und Transaktionskosten."""

    def __init__(self, universum, benchmark="SPY", jahre=5,
                 startkapital=100_000.0, gebuehrensatz=0.0015,
                 max_gewicht=0.20, alpha=0.95, risikoaversion=1.2,
                 warmup=HANDELSTAGE):
        self.universum = list(universum)
        self.benchmark = benchmark
        self.jahre = jahre
        self.startkapital = startkapital
        self.gebuehrensatz = gebuehrensatz
        self.max_gewicht = max_gewicht
        self.alpha = alpha
        self.risikoaversion = risikoaversion
        self.warmup = warmup

        self.kurse = None
        self.benchmark_kurse = None
        self.renditen = None
        self.rebalancing_termine = None

    # -- 1. Datenpipeline ------------------------------------------------
    def lade_daten(self):
        ende = pd.Timestamp.today().normalize()
        start = ende - pd.DateOffset(years=self.jahre)
        alle = self.universum + [self.benchmark]

        print(f"Lade {len(self.universum)} Titel + Benchmark ({self.benchmark}) "
              f"ueber {self.jahre} Jahre ...")
        roh = yf.download(alle, start=start, end=ende, auto_adjust=True, progress=False)
        if roh.empty:
            raise SystemExit("Download fehlgeschlagen (Netz/Ticker/Rate-Limit pruefen).")

        if isinstance(roh.columns, pd.MultiIndex):
            daten = roh["Close"][alle].dropna()      # erzwingt eigene Spaltenreihenfolge
        else:
            daten = roh[["Close"]].dropna()
            daten.columns = alle
        assert list(daten.columns) == alle, "Spaltenreihenfolge weicht ab!"

        self.kurse = daten[self.universum]
        self.benchmark_kurse = daten[self.benchmark]
        self.renditen = self.kurse.pct_change().dropna()
```

```
        # --- erster HANDELStag je Monat, nicht Kalendermonatsanfang -----
        monat = self.kurse.index.to_period("M")
        self.rebalancing_termine = self.kurse.index[~monat.duplicated()]
        assert self.rebalancing_termine.isin(self.kurse.index).all(), \
            "Rebalancing-Termine sind keine Handelstage!"

        print(f"  {len(self.kurse)} Handelstage, "
              f"{len(self.rebalancing_termine)} Rebalancing-Termine "
              f"(alle sind Handelstage)")

    # -- 2. Signal ---------------------------------------------------------
    def alpha_signal(self, stichtag, rueckblick=HANDELSTAGE):
        """
        12-1-Momentum: Kursentwicklung der letzten 12 Monate unter Auslassung
        des letzten Monats (kurzfristige Umkehr herausrechnen).
        Rueckgabe: TAEGLICHE Renditeerwartung.
        """
        historie = self.kurse.loc[:stichtag]
        if len(historie) < rueckblick:
            return np.zeros(len(self.universum))

        p_alt = historie.iloc[-rueckblick]
        p_neu = historie.iloc[-21]                    # vor ca. einem Monat
        momentum = (p_neu / p_alt - 1.0).values

        # Z-Score, damit das Signal skalenfrei wird
        streuung = momentum.std()
        score = (momentum - momentum.mean()) / (streuung + 1e-9)

        # In eine taegliche Renditeerwartung uebersetzen:
        # Basis 4 % p.a. plus 6 % p.a. je Standardabweichung Momentum.
        mu_jaehrlich = np.maximum(0.04 + 0.06 * score, 0.0)
        return mu_jaehrlich / HANDELSTAGE

    # -- 3./4. Risikomodell und Optimierung --------------------------------
    def optimiere(self, stichtag, w_alt, rueckblick=HANDELSTAGE):
        historie = self.renditen.loc[:stichtag].iloc[-rueckblick:]
        if len(historie) < 100:
            return w_alt, False

        R = historie.values
        S, N = R.shape
        mu_taeglich = self.alpha_signal(stichtag, rueckblick)

        w = cp.Variable(N, nonneg=True)
        gamma = cp.Variable()
        u = cp.Variable(S, nonneg=True)
```

```
        cvar = gamma + (1.0 / (S * (1.0 - self.alpha))) * cp.sum(u)
        kosten = self.gebuehrensatz * cp.norm1(w - w_alt)
        ziel = cp.Maximize(mu_taeglich @ w - self.risikoaversion * cvar - kosten)

        bedingungen = [cp.sum(w) == 1,
                       w <= self.max_gewicht,
                       u >= -(R @ w) - gamma]          # vektorisiert
        problem = cp.Problem(ziel, bedingungen)
        try:
            problem.solve(solver=cp.CLARABEL)
        except cp.error.SolverError:
            return w_alt, False

        if problem.status not in ("optimal", "optimal_inaccurate") or w.value is None:
            return w_alt, False

        gewichte = np.maximum(w.value, 0.0)
        return gewichte / gewichte.sum(), True

    # -- 5. Backtest ---------------------------------------------------------
    def backtest(self):
        print("Starte Walk-Forward-Backtest ...")
        N = len(self.universum)
        gewichte = np.ones(N) / N

        werte = [self.startkapital]
        datumsliste = [self.renditen.index[self.warmup]]
        gewichtshistorie = []
        geplant = versucht = ausgefuehrt = 0
        kosten_gesamt = 0.0
        turnover_gesamt = 0.0

        for t in range(self.warmup, len(self.renditen) - 1):
            heute = self.renditen.index[t]
            morgen = self.renditen.index[t + 1]

            ist_termin = heute in self.rebalancing_termine or t == self.warmup
            if ist_termin:
                geplant += 1
                versucht += 1
                neue_gewichte, erfolg = self.optimiere(heute, gewichte)
                if erfolg:
                    ausgefuehrt += 1
                    turnover = float(np.abs(neue_gewichte - gewichte).sum())
                    gebuehr = werte[-1] * turnover * self.gebuehrensatz
                    werte[-1] -= gebuehr
                    kosten_gesamt += gebuehr
                    turnover_gesamt += turnover
```

```
                gewichte = neue_gewichte
                gewichtshistorie.append((heute, gewichte.copy()))

            # Rendite von MORGEN auf die HEUTE festgelegten Gewichte anwenden
            tagesrenditen = self.renditen.iloc[t + 1].values
            portfoliorendite = float(gewichte @ tagesrenditen)
            werte.append(werte[-1] * (1.0 + portfoliorendite))
            datumsliste.append(morgen)

            # Gewichte driften mit den Kursen weiter
            gedriftet = gewichte * (1.0 + tagesrenditen)
            gewichte = gedriftet / gedriftet.sum()

        print(f"  Rebalancing: {geplant} geplant, {ausgefuehrt} ausgefuehrt "
              f"({ausgefuehrt/max(geplant,1)*100:.0f} %)")
        if ausgefuehrt < geplant:
            print(f"  WARNUNG: {geplant - ausgefuehrt} Optimierungen sind "
                  f"fehlgeschlagen - Ursache pruefen!")
        print(f"  Kumulierter Turnover: {turnover_gesamt*100:.0f} % | "
              f"Transaktionskosten insgesamt: {kosten_gesamt:,.2f} EUR")

        ergebnis = pd.DataFrame({"Portfolio": werte}, index=datumsliste)
        benchmark = self.benchmark_kurse.loc[datumsliste]
        ergebnis["Benchmark"] = benchmark / benchmark.iloc[0] * self.startkapital
        return ergebnis, gewichtshistorie, kosten_gesamt

    # -- 6. Auswertung ------------------------------------------------------
    @staticmethod
    def kennzahlen(reihe, risikofrei=0.02):
        renditen = reihe.pct_change().dropna()
        jahre = (reihe.index[-1] - reihe.index[0]).days / 365.25
        cagr = (reihe.iloc[-1] / reihe.iloc[0]) ** (1 / jahre) - 1
        vola = renditen.std() * np.sqrt(HANDELSTAGE)
        sharpe = (cagr - risikofrei) / vola if vola > 0 else np.nan
        drawdown = ((reihe - reihe.cummax()) / reihe.cummax()).min()
        calmar = cagr / abs(drawdown) if drawdown < 0 else np.nan
        trefferquote = float((renditen > 0).mean())
        return {"CAGR": cagr, "Volatilitaet": vola, "Sharpe": sharpe,
                "Max Drawdown": drawdown, "Calmar": calmar,
                "Trefferquote": trefferquote, "Jahre": jahre}

    def bericht(self, ergebnis, kosten_gesamt):
        k_port = self.kennzahlen(ergebnis["Portfolio"])
        k_bench = self.kennzahlen(ergebnis["Benchmark"])

        print("\n" + "=" * 84)
        print("          LEISTUNGSREPORT DER QUANTITATIVEN ENGINE")
        print("=" * 84)
```

```
        print(f"Zeitraum: {ergebnis.index[0]:%Y-%m-%d} bis {ergebnis.index[-1]:%Y-%m-%d} "
              f"({k_port['Jahre']:.1f} Jahre) | Startkapital "
              f"{self.startkapital:,.0f} EUR\n")

        zeilen = [
            ("Endkapital", f"{ergebnis['Portfolio'].iloc[-1]:,.0f} EUR",
             f"{ergebnis['Benchmark'].iloc[-1]:,.0f} EUR"),
            ("CAGR (Jahresrendite)", f"{k_port['CAGR']*100:6.2f} %",
             f"{k_bench['CAGR']*100:6.2f} %"),
            ("Volatilitaet p.a.", f"{k_port['Volatilitaet']*100:6.2f} %",
             f"{k_bench['Volatilitaet']*100:6.2f} %"),
            ("Sharpe Ratio (rf=2 %)", f"{k_port['Sharpe']:6.2f}",
             f"{k_bench['Sharpe']:6.2f}"),
            ("Maximum Drawdown", f"{k_port['Max Drawdown']*100:6.2f} %",
             f"{k_bench['Max Drawdown']*100:6.2f} %"),
            ("Calmar Ratio", f"{k_port['Calmar']:6.2f}", f"{k_bench['Calmar']:6.2f}"),
            ("Trefferquote (Tage)", f"{k_port['Trefferquote']*100:6.1f} %",
             f"{k_bench['Trefferquote']*100:6.1f} %"),
        ]
        print(pd.DataFrame(zeilen, columns=["Kennzahl", "OR-Strategie", "Benchmark"])
              .to_string(index=False))

        anteil = kosten_gesamt / self.startkapital
        print(f"\nTransaktionskosten: {kosten_gesamt:,.0f} EUR "
              f"= {anteil*100:.2f} % des Startkapitals "
              f"= {anteil/k_port['Jahre']*100:.2f} % p.a.")
        print("Ohne diese Kosten waere die ausgewiesene Rendite entsprechend hoeher -")
        print("genau deshalb gehoeren sie in den Backtest und nicht daneben.")
        print("=" * 84)
        return k_port, k_bench

    def zeichne(self, ergebnis):
        plt.figure(figsize=(11, 6))
        plt.plot(ergebnis.index, ergebnis["Portfolio"], "b-", lw=2.2,
                 label="OR-Handelsmaschine")
        plt.plot(ergebnis.index, ergebnis["Benchmark"], "k--", lw=1.5, alpha=0.75,
                 label=f"Benchmark ({self.benchmark})")
        plt.title("Walk-Forward-Backtest: OR-Strategie gegen Benchmark", fontsize=12)
        plt.xlabel("Datum")
        plt.ylabel("Depotwert in EUR")
        plt.grid(True, linestyle=":", alpha=0.6)
        plt.legend(loc="upper left")
        plt.tight_layout()
        ziel = os.path.join(OUTPUT_DIR, "trading_engine_backtest.png")
        plt.savefig(ziel, dpi=150)
        print(f"Chart gespeichert unter '{ziel}'")
```

```
def lookahead_selbsttest(maschine):
    """
    Prueft, dass das Signal zum Zeitpunkt t NUR Daten bis t verwendet.
    Methode: kuenstlich die Zukunft veraendern und pruefen, ob sich das
    Signal von heute dadurch aendert. Tut es das, liegt Lookahead vor.
    """
    stichtag = maschine.kurse.index[len(maschine.kurse) // 2]
    signal_original = maschine.alpha_signal(stichtag)

    original_kurse = maschine.kurse.copy()
    zukunft = maschine.kurse.index > stichtag
    maschine.kurse.loc[zukunft] *= 3.0            # Zukunft massiv veraendern
    signal_manipuliert = maschine.alpha_signal(stichtag)
    maschine.kurse = original_kurse               # zuruecksetzen

    identisch = np.allclose(signal_original, signal_manipuliert)
    print(f"Lookahead-Selbsttest: Signal bleibt bei manipulierter Zukunft "
          f"{'UNVERAENDERT (gut)' if identisch else 'VERAENDERT - LOOKAHEAD-BIAS!'}")
    assert identisch, "Das Signal verwendet Zukunftsdaten!"


if __name__ == "__main__":
    UNIVERSUM = ["AAPL", "MSFT", "NVDA", "GOOGL",
                 "JNJ", "UNH", "PFE",
                 "JPM", "BAC", "GS",
                 "XOM", "CVX",
                 "PG", "KO", "COST"]

    maschine = QuantitativeHandelsmaschine(UNIVERSUM, benchmark="SPY", jahre=5)
    maschine.lade_daten()
    lookahead_selbsttest(maschine)
    ergebnis, gewichte, kosten = maschine.backtest()
    maschine.bericht(ergebnis, kosten)
    maschine.zeichne(ergebnis)
```

Code-Durchgang: die vier kritischen Stellen

1. Rebalancing-Kalender.

```
monat = self.kurse.index.to_period("M")
self.rebalancing_termine = self.kurse.index[~monat.duplicated()]
```

`to_period("M")` bildet jeden Handelstag auf seinen Monat ab; `~duplicated()` behält jeweils den **ersten** — also garantiert einen realen Handelstag.

2. Die Zeitachse im Backtest.

```
tagesrenditen = self.renditen.iloc[t + 1].values      # MORGEN
portfoliorendite = gewichte @ tagesrenditen            # HEUTE festgelegt
```

Die Gewichte werden am Tag t aus Daten bis t bestimmt und auf die Rendite von $t + 1$ angewendet. **Diese eine Indexverschiebung ist der Unterschied zwischen einem ehrlichen und einem geschönten Backtest.**

3. Der Lookahead-Selbsttest. Er verdreifacht künstlich alle Kurse **nach** dem Stichtag und prüft, ob sich das Signal von *vor* dem Stichtag dadurch ändert. Ändert es sich, greift der Code auf die Zukunft zu. Dieser Test kostet zehn Zeilen und findet eine Fehlerklasse, die sonst praktisch unsichtbar bleibt. **Übernehmen Sie ihn in jedes Backtest-Projekt.**

4. Gewichtsdrift.

```
gedriftet = gewichte * (1.0 + tagesrenditen)
gewichte = gedriftet / gedriftet.sum()
```

Zwischen zwei Rebalancings verändern sich die Gewichte von selbst — gestiegene Titel bekommen mehr Gewicht. Wer das weglässt, unterstellt implizit **tägliches** kostenloses Rebalancing und überschätzt die Strategie.

 Zur Reproduzierbarkeit Dieses Programm lädt Live-Daten über ein relatives Fünfjahresfenster. **Ihre Zahlen werden von jeder abgedruckten Zahl abweichen** — je nach Abrufdatum und Marktlage. Bewerten Sie deshalb nicht die absoluten Werte, sondern die **Struktur**: Wurden alle Rebalancings ausgeführt? Besteht der Lookahead-Test? Sind die Transaktionskosten plausibel?

21.6 Die fünf Selbsttäuschungen des Backtestens

```
#!/usr/bin/env python3

# Backtest_Fallen.py
"""
Kapitel Handelsmaschine: Wie leicht man sich selbst betruegt.

Demonstriert an SIMULIERTEN Daten (also ohne jede echte Prognosekraft),
wie gross die scheinbare Ueberrendite durch typische Backtest-Fehler wird.
Weil die Daten reines Rauschen sind, ist JEDE positive Ueberrendite ein Artefakt.
"""

import numpy as np
import pandas as pd


def erzeuge_zufallsmarkt(tage=1260, titel=15, seed=7):
    """Reines Rauschen: kein Titel hat echte Prognosekraft."""
    rng = np.random.default_rng(seed)
    renditen = rng.normal(0.0003, 0.015, size=(tage, titel))
    index = pd.bdate_range("2020-01-01", periods=tage)
    return pd.DataFrame(renditen, index=index,
```

```
                        columns=[f"T{i:02d}" for i in range(titel)])


def strategie(renditen, lookahead=False, survivorship=False,
              ohne_kosten=False, gebuehr=0.0015):
    """Einfache Momentum-Strategie mit optional eingebauten Fehlern."""
    fenster = 60
    kapital = [100.0]
    gewichte = np.ones(renditen.shape[1]) / renditen.shape[1]

    daten = renditen
    if survivorship:
        # FEHLER: nur die im Nachhinein besten Titel ins Universum lassen
        beste = renditen.sum().nlargest(renditen.shape[1] // 2).index
        daten = renditen[beste]
        gewichte = np.ones(len(beste)) / len(beste)

    for t in range(fenster, len(daten) - 1):
        if t % 21 == 0:                                  # monatlich
            if lookahead:
                # FEHLER: Signal nutzt die Rendite von MORGEN
                signal = daten.iloc[t + 1].values
            else:
                signal = daten.iloc[t - fenster:t].mean().values

            neue = (signal > np.median(signal)).astype(float)
            neue = neue / neue.sum() if neue.sum() > 0 else gewichte
            if not ohne_kosten:
                kapital[-1] *= (1 - np.abs(neue - gewichte).sum() * gebuehr)
            gewichte = neue

        kapital.append(kapital[-1] * (1 + float(gewichte @ daten.iloc[t + 1].values)))
    return np.array(kapital)


def cagr(verlauf, jahre):
    return (verlauf[-1] / verlauf[0]) ** (1 / jahre) - 1


if __name__ == "__main__":
    renditen = erzeuge_zufallsmarkt()
    jahre = len(renditen) / 252
    markt = 100 * (1 + renditen.mean(axis=1)).cumprod().values

    print("=" * 84)
    print("  DIE FUENF SELBSTTAEUSCHUNGEN - GEMESSEN AN REINEM RAUSCHEN")
    print("=" * 84)
    print("Die Daten sind zufaellig erzeugt. Es gibt KEINE echte Prognosekraft.")
```

```
    print("Jede Ueberrendite unten ist daher ein reines Artefakt.\n")
    print(f"{'Variante':<44} {'CAGR':>9} {'ggue. Markt':>13}")
    print("-" * 84)

    basis = cagr(markt, jahre)
    print(f"{'Markt (Gleichgewichtung, Referenz)':<44} {basis*100:>8.2f} % "
          f"{0.0:>12.2f} %")

    varianten = [
        ("Ehrlich: Signal aus Vergangenheit, mit Kosten", {}),
        ("FEHLER 1: Lookahead (Signal kennt morgen)", {"lookahead": True}),
        ("FEHLER 2: Survivorship (nur Gewinner im Universum)", {"survivorship": True}),
        ("FEHLER 3: Transaktionskosten ignoriert", {"ohne_kosten": True}),
        ("FEHLER 1+2+3 kombiniert", {"lookahead": True, "survivorship": True,
                                     "ohne_kosten": True}),
    ]
    for name, argumente in varianten:
        verlauf = strategie(renditen, **argumente)
        wert = cagr(verlauf, jahre)
        print(f"{name:<44} {wert*100:>8.2f} % {(wert-basis)*100:>+12.2f} %")

    print("-" * 84)
    print("FEHLER 4 (Overfitting) und FEHLER 5 (Data Snooping) lassen sich so nicht")
    print("zeigen - sie entstehen erst durch WIEDERHOLTES Probieren. Faustregel:")
    print("Wer 20 Strategien testet, findet auch in reinem Rauschen eine mit")
    print("Signifikanz auf dem 5-%-Niveau. Genau das ist der Punkt.")
    print("=" * 84)
```

Erwartete Ausgabe:

```
====================================================================================
  DIE FUENF SELBSTTAEUSCHUNGEN - GEMESSEN AN REINEM RAUSCHEN
====================================================================================
Die Daten sind zufaellig erzeugt. Es gibt KEINE echte Prognosekraft.
Jede Ueberrendite unten ist daher ein reines Artefakt.

Variante                                        CAGR   ggue. Markt
------------------------------------------------------------------------------------
Markt (Gleichgewichtung, Referenz)               5.61 %         0.00 %
Ehrlich: Signal aus Vergangenheit, mit Kosten    6.71 %        +1.10 %
FEHLER 1: Lookahead (Signal kennt morgen)       20.96 %       +15.35 %
FEHLER 2: Survivorship (nur Gewinner im Univ.)  18.71 %       +13.11 %
FEHLER 3: Transaktionskosten ignoriert           7.77 %        +2.16 %
FEHLER 1+2+3 kombiniert                         25.41 %       +19.80 %
------------------------------------------------------------------------------------
```

Lesen Sie diese Tabelle sehr genau. Die zugrunde liegenden Daten sind **reiner Zufall** — es gibt keinerlei Prognosekraft, kein Signal, keine Struktur. Trotzdem:

- **Lookahead** erzeugt eine scheinbare Überrendite von **15,4 Prozentpunkten pro Jahr**. Ein einziger falscher Index — `iloc[t+1]` statt `iloc[t]` — verwandelt Rauschen in eine scheinbar brillante Strategie.
- **Survivorship-Bias** liefert **13,1 Prozentpunkte** allein dadurch, dass man das Universum im Nachhinein auf die Gewinner beschränkt.
- Selbst die **ehrliche** Variante zeigt noch +1,1 Prozentpunkte — reines Rauschen, das aber ohne Weiteres als „leichte Outperformance“ verkauft werden könnte.

Die Lehre: Wenn Sie in einem Backtest eine Überrendite von 15 % sehen, ist die erste Frage nicht „Wie kann ich das nutzen?“, sondern **„Wo ist mein Fehler?“**. Und wenn Sie keinen finden, suchen Sie weiter — die Wahrscheinlichkeit, dass Sie tatsächlich eine Goldader entdeckt haben, ist deutlich kleiner als die, dass ein Index verrutscht ist.

Die fünf Fallen im Einzelnen:

#	Falle	Was passiert	Gegenmittel
1	**Lookahead-Bias**	Entscheidungen nutzen Daten, die es noch nicht gab	Zeitindex prüfen; Selbsttest wie oben
2	**Survivorship-Bias**	Nur heute existierende Titel im Universum	Historische Indexzusammensetzung verwenden
3	**Kostenblindheit**	Gebühren, Spread, Slippage ignoriert	Kosten im Modell **und** in der Verbuchung
4	**Overfitting**	Parameter auf die Vergangenheit getrimmt	Out-of-Sample-Zeitraum strikt zurückhalten
5	**Data Snooping**	Viele Varianten getestet, beste berichtet	Zahl der Versuche protokollieren; Deflated Sharpe Ratio

 Die unbequeme Wahrheit Falle 5 ist die gefährlichste, weil sie sich nicht im Code zeigt, sondern im **Arbeitsprozess**. Wer zwanzig Parameterkombinationen durchprobiert und die beste berichtet, hat keine Strategie gefunden, sondern eine Zufallsschwankung ausgewählt. Marcos López de Prado hat gezeigt, dass die meisten veröffentlichten Backtests genau daran scheitern. **Protokollieren Sie, wie viele Varianten Sie getestet haben** — und berichten Sie diese Zahl mit.

21.7 Vom Backtest zum Betrieb

Ein guter Backtest ist eine **notwendige**, keine hinreichende Bedingung. Was zwischen Backtest und echtem Einsatz liegt:

Schritt	Frage	Typisches Ergebnis
Papierhandel	Funktioniert es auf Live-Daten ohne Geld?	3–6 Monate, oft schlechter als der Backtest
Kapazitätsanalyse	Bewegt meine eigene Order den Markt?	Begrenzt das einsetzbare Volumen
Regimewechsel-Test	Hält es in Krisen?	Backtest über 2008, 2020 separat auswerten
Betriebsrisiken	Was, wenn der Datenfeed ausfällt?	Fallback-Strategie, Monitoring
Regulatorik	MiFID II, Best Execution, Dokumentation	Protokollpflichten

 Der Realitätsabschlag Als Faustregel gilt: **Die reale Performance liegt deutlich unter dem Backtest** — durch Slippage, verzögerte Ausführung, Datenrevisionen und schlicht dadurch, dass die Zukunft nicht die Vergangenheit ist. Wer einen Backtest mit Sharpe 1,5 hat, sollte real mit deutlich weniger rechnen. Eine Strategie, die im Backtest nur knapp die Benchmark schlägt, wird es real nicht tun.

21.8 Übungsaufgaben

Lösungen: Abschnitt A.21.

Aufgabe 21.1 — **Lookahead erkennen.** Welche dieser Zeilen enthalten Lookahead-Bias? Begründen Sie: (a) `signal = kurse.loc[:heute].pct_change().mean()` (b) `signal = kurse.pct_change().mean()` (c) `vola = renditen.std()` (berechnet einmal über den gesamten Zeitraum) (d) `w = optimiere(renditen.loc[:heute]); rendite = w @ renditen.loc[heute]` (e) `universum = [t for t in alle if kurse[t].notna().all()]`

Aufgabe 21.2 — **Kennzahlen deuten.** Strategie A: CAGR 12 %, Vola 22 %, MaxDD −35 %. Strategie B: CAGR 8 %, Vola 9 %, MaxDD −12 %. Berechnen Sie Sharpe (bei $r_f = 2\,\%$) und Calmar. Welche würden Sie einem Pensionsfonds empfehlen und warum?

Aufgabe 21.3 — **Rebalancing-Kalender prüfen.** Bauen Sie den Fehler nach: Ersetzen Sie den korrigierten Kalender durch `resample("MS").first().index`. Zählen Sie, wie viele Rebalancings ausfallen, und vergleichen Sie die Backtest-Kennzahlen beider Varianten.

Aufgabe 21.4 — **Rebalancing-Frequenz.** Variieren Sie die Frequenz: täglich, wöchentlich, monatlich, quartalsweise, jährlich. (a) Wie ändern sich Rendite, Turnover und Kosten? (b) Ab welcher Frequenz fressen die Kosten den Ertrag auf? (c) Was ist Ihre Empfehlung bei einem Gebührensatz von 0,15 %? Bei 0,5 %?

Aufgabe 21.5 □□□ **— Krisenverhalten.** Werten Sie den Backtest getrennt für Teilzeiträume aus (z. B. je Kalenderjahr). Erstellen Sie eine Tabelle mit CAGR, Vola und MaxDD je Jahr für Strategie und Benchmark. (a) In welchen Phasen schlägt die Strategie die Benchmark? (b) Ist das Muster stabil oder zufällig? (c) Was folgt daraus für die Beurteilung der Gesamtkennzahl?

Aufgabe 21.6 □□□ **— Data Snooping messen.** Testen Sie 30 Varianten Ihrer Strategie (verschiedene Rückblickfenster, Risikoaversionen, Positionsgrenzen). Berichten Sie: (a) die beste Sharpe Ratio, (b) die mittlere Sharpe Ratio über alle Varianten, (c) die Sharpe Ratio der besten Variante **auf einem zurückgehaltenen Zeitraum**. Was lernen Sie aus der Differenz zwischen (a) und (c)?

21.9 Finde den Denkfehler

`Backtest_Fallen.py` zeigt drei Fehler, die **im** Backtest stecken: Lookahead, Survivorship, vergessene Kosten. Alle drei kann man einem Programm ansehen, wenn man es liest.

Die beiden übrigen Selbsttäuschungen sind anders gebaut. Sie entstehen nicht im Code, sondern in der **Arbeitsweise davor** — und hinterlassen im Programm keine Spur. Ein Backtest, der zwanzigmal nachjustiert wurde, sieht am Ende genauso sauber aus wie einer, der beim ersten Versuch funktioniert hat.

> □ **Finde den Denkfehler: Die Strategie mit dem hochsignifikanten Ergebnis**
>
> Eine Analystin entwickelt über mehrere Wochen eine Handelsstrategie. Sie probiert verschiedene Signalfenster, Schwellenwerte, Filter und Haltedauern durch — insgesamt rund **hundert Varianten**. Die beste testet sie sauber: kein Lookahead, korrektes Universum, Transaktionskosten berücksichtigt.
>
> Das Ergebnis über fünf Jahre: **Sharpe-Kennzahl 1,13**, p-Wert **0,0094**. Der Backtest ist handwerklich einwandfrei — jede der drei Fallen aus `Backtest_Fallen.py` wurde vermieden. Sie stellt die Strategie vor.
>
> **Ihre Aufgabe:** (a) Der Backtest ist korrekt. Warum ist das Ergebnis trotzdem wertlos? (b) Sehen Sie sich unten die letzte Spalte der Tabelle an: Sie bleibt konstant bei 5 %, obwohl die Zahl der Versuche um das Tausendfache wächst. Wieso ist das kein Widerspruch, sondern der Kern des Problems? (c) Welche Zahl fehlt in ihrem Bericht — eine Zahl, die in keinem Backtest-Ergebnis steht und die sie selbst aufschreiben müsste? (d) Nennen Sie drei Gegenmittel und begründen Sie, warum das dritte nur **einmal** verwendbar ist.
>
> *Auflösung: Abschnitt A.21.*

Das folgende Programm macht den Effekt messbar. Es testet Strategien, die **nachweislich** keinerlei Vorteil haben — ihre Tagesergebnisse sind Zufallszahlen mit Erwartungswert exakt null — und zeigt, wie gut die jeweils beste davon aussieht, allein als Funktion der Anzahl der Versuche.

```
#!/usr/bin/env python3

# Data_Snooping.py
"""
Kapitel Handelsmaschine: Die Selbsttaeuschung, die man einem Backtest nicht ansieht.
```

```
Backtest_Fallen.py zeigt drei Fehler, die IM Backtest stecken: Lookahead,
Survivorship, vergessene Kosten. Alle drei kann man einem Programm ansehen,
wenn man es liest.

Die beiden anderen - Overfitting und Data Snooping - entstehen nicht im
Backtest, sondern in der ARBEITSWEISE davor. Sie hinterlassen im Code keine
Spur. Ein Backtest, der zwanzigmal angepasst wurde, sieht am Ende genauso
sauber aus wie einer, der beim ersten Versuch funktioniert hat.

Dieses Programm macht den Effekt messbar. Es testet Strategien, die
NACHWEISLICH keinerlei Prognosekraft haben (ihre taeglichen Ergebnisse sind
Zufallszahlen mit Erwartungswert exakt null), und zeigt, wie gut die jeweils
BESTE davon aussieht - allein als Funktion der Anzahl der Versuche.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import numpy as np
from scipy import stats

HANDELSTAGE = 1260              # fuenf Jahre
TAGESSCHWANKUNG = 0.010         # 1 % taeglich
WIEDERHOLUNGEN = 300            # damit die Tabelle nicht vom Zufall abhaengt
NIVEAU = 0.05

RNG = np.random.default_rng(42)


def bewerte(ergebnisse: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """Sharpe-Kennzahl und einseitiger p-Wert je Strategie.

    ergebnisse hat die Form (Strategien, Handelstage). Der p-Wert prueft die
    Nullhypothese 'mittleres Tagesergebnis ist null' - also genau die Frage,
    die ein Backtest beantworten soll.
    """
    mittel = ergebnisse.mean(axis=1)
    streuung = ergebnisse.std(axis=1, ddof=1)
    sharpe = mittel / streuung * np.sqrt(252)
    t_wert = mittel / (streuung / np.sqrt(ergebnisse.shape[1]))
    return sharpe, stats.t.sf(t_wert, ergebnisse.shape[1] - 1)


def probiere(anzahl_strategien: int) -> tuple[float, float, float, float]:
    """Liefert (beste Sharpe, ihr p-Wert, Anteil scheinbar signifikanter,
    Anteil, der auch die Bonferroni-Huerde nimmt) - gemittelt ueber
```

```
    WIEDERHOLUNGEN unabhaengige Durchgaenge."""
    beste_sharpe, beste_p, anteil_signifikant, ueberlebt_bonferroni = [], [], [], []

    for _ in range(WIEDERHOLUNGEN):
        # Der springende Punkt: Erwartungswert exakt 0. Keine dieser
        # Strategien hat einen echten Vorteil - per Konstruktion.
        ergebnisse = RNG.normal(0.0, TAGESSCHWANKUNG,
                                (anzahl_strategien, HANDELSTAGE))
        sharpe, p_werte = bewerte(ergebnisse)

        beste = int(sharpe.argmax())
        beste_sharpe.append(float(sharpe[beste]))
        beste_p.append(float(p_werte[beste]))
        anteil_signifikant.append(float((p_werte < NIVEAU).mean()))
        # Bonferroni: Wer n Tests macht, muss die Huerde durch n teilen.
        ueberlebt_bonferroni.append(
            float((p_werte < NIVEAU / anzahl_strategien).mean()))

    return (float(np.mean(beste_sharpe)), float(np.mean(beste_p)),
            float(np.mean(anteil_signifikant)),
            float(np.mean(ueberlebt_bonferroni)))


if __name__ == "__main__":
    print("=" * 78)
    print("  WIE GUT SIEHT DIE BESTE VON N STRATEGIEN AUS,")
    print("  WENN KEINE EINZIGE ETWAS TAUGT?")
    print("=" * 78)
    print(f"{HANDELSTAGE} Handelstage, taegliche Ergebnisse mit Erwartungswert "
          f"EXAKT null.")
    print(f"Jede Zeile ist der Mittelwert aus {WIEDERHOLUNGEN} unabhaengigen "
          f"Durchgaengen.\n")

    print(f"{'getestet':>9} {'beste Sharpe':>14} {'ihr p-Wert':>12} "
          f"{'scheinbar sign.':>16}")
    print("-" * 78)
    ergebnisse = {}
    for anzahl in (1, 10, 100, 1000):
        sharpe, p_wert, anteil, _ = probiere(anzahl)
        ergebnisse[anzahl] = (sharpe, p_wert, anteil)
        print(f"{anzahl:9d} {sharpe:14.2f} {p_wert:12.4f} {anteil * 100:15.1f} %")
    print("-" * 78)

    sharpe_100, p_100, _ = ergebnisse[100]
    sharpe_1000, p_1000, _ = ergebnisse[1000]

    print("\nLesen Sie die dritte Spalte von unten nach oben:")
    print(f"  Wer {1000} Varianten durchprobiert, findet eine mit p = {p_1000:.4f} -")
```

```
    print("  ein Wert, den jede Zeitschrift als 'hochsignifikant' druckt.")
    print(f"  Wer {100} probiert, findet p = {p_100:.4f} und eine Sharpe-Kennzahl")
    print(f"  von {sharpe_100:.2f} - damit geht man zum Vorgesetzten.")
    print("  Und keine dieser Strategien hat auch nur den Hauch eines Vorteils.")
    print()
    print("Die LETZTE Spalte bleibt dagegen konstant bei rund 5 %. Das ist kein")
    print("Widerspruch, sondern die Definition: Ein Test zum 5-%-Niveau irrt in")
    print("5 % der Faelle. Was sich aendert, ist nicht der Anteil, sondern die")
    print("ANZAHL - und weil man nur die beste Strategie praesentiert, sieht man")
    print("von allen Fehlversuchen genau einen: den erfolgreichsten.")

    print("\n" + "=" * 78)
    print("  WAS MAN DAGEGEN TUN KANN")
    print("=" * 78)
    print("1. ZAEHLEN. Notieren Sie, wie viele Varianten Sie ausprobiert haben -")
    print("   auch die verworfenen, auch die 'nur mal schnell'. Ohne diese Zahl")
    print("   ist kein p-Wert und keine Sharpe-Kennzahl interpretierbar.")
    print()
    print("2. KORRIGIEREN. Die Bonferroni-Korrektur teilt die Huerde durch die")
    print("   Zahl der Versuche:")
    for anzahl in (10, 100, 1000):
        print(f"     bei {anzahl:5d} Versuchen: p < {NIVEAU / anzahl:.5f} statt "
              f"p < {NIVEAU:.2f}")
    print("   Sie ist konservativ, aber sie ist ehrlich. Fuer Handelsstrategien")
    print("   gibt es verfeinerte Varianten (Deflated Sharpe Ratio nach Bailey")
    print("   und Lopez de Prado), die dasselbe Prinzip verfolgen.")
    print()
    print("3. ZURUECKHALTEN. Legen Sie einen Zeitraum beiseite, BEVOR Sie")
    print("   anfangen, und ruehren Sie ihn bis zum Schluss nicht an. Er ist")
    print("   der einzige Test, den Sie nicht durch Probieren verderben koennen -")
    print("   und er ist genau einmal verwendbar.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  WIE GUT SIEHT DIE BESTE VON N STRATEGIEN AUS,
  WENN KEINE EINZIGE ETWAS TAUGT?
==============================================================================
1260 Handelstage, taegliche Ergebnisse mit Erwartungswert EXAKT null.
Jede Zeile ist der Mittelwert aus 300 unabhaengigen Durchgaengen.

 getestet   beste Sharpe   ihr p-Wert  scheinbar sign.
------------------------------------------------------------------------------
        1          -0.00       0.5101             6.0 %
       10           0.69       0.0885             4.9 %
      100           1.13       0.0094             4.8 %
     1000           1.44       0.0010             5.0 %
```

```
--------------------------------------------------------------------------------

Lesen Sie die dritte Spalte von unten nach oben:
  Wer 1000 Varianten durchprobiert, findet eine mit p = 0.0010 -
  ein Wert, den jede Zeitschrift als 'hochsignifikant' druckt.
  Wer 100 probiert, findet p = 0.0094 und eine Sharpe-Kennzahl
  von 1.13 - damit geht man zum Vorgesetzten.
  Und keine dieser Strategien hat auch nur den Hauch eines Vorteils.

Die LETZTE Spalte bleibt dagegen konstant bei rund 5 %. Das ist kein
Widerspruch, sondern die Definition: Ein Test zum 5-%-Niveau irrt in
5 % der Faelle. Was sich aendert, ist nicht der Anteil, sondern die
ANZAHL - und weil man nur die beste Strategie praesentiert, sieht man
von allen Fehlversuchen genau einen: den erfolgreichsten.

================================================================================
  WAS MAN DAGEGEN TUN KANN
================================================================================
1. ZAEHLEN. Notieren Sie, wie viele Varianten Sie ausprobiert haben -
   auch die verworfenen, auch die 'nur mal schnell'. Ohne diese Zahl
   ist kein p-Wert und keine Sharpe-Kennzahl interpretierbar.

2. KORRIGIEREN. Die Bonferroni-Korrektur teilt die Huerde durch die
   Zahl der Versuche:
     bei    10 Versuchen: p < 0.00500 statt p < 0.05
     bei   100 Versuchen: p < 0.00050 statt p < 0.05
     bei  1000 Versuchen: p < 0.00005 statt p < 0.05
   Sie ist konservativ, aber sie ist ehrlich. Fuer Handelsstrategien
   gibt es verfeinerte Varianten (Deflated Sharpe Ratio nach Bailey
   und Lopez de Prado), die dasselbe Prinzip verfolgen.

3. ZURUECKHALTEN. Legen Sie einen Zeitraum beiseite, BEVOR Sie
   anfangen, und ruehren Sie ihn bis zum Schluss nicht an. Er ist
   der einzige Test, den Sie nicht durch Probieren verderben koennen -
   und er ist genau einmal verwendbar.
================================================================================
```

 Merksatz Ein Backtest-Ergebnis ist ohne die Angabe **„wie viele Varianten wurden probiert?“** nicht interpretierbar. Diese Zahl steht in keinem Programm und in keiner Kennzahl — sie existiert nur im Kopf der Person, die die Arbeit gemacht hat. Genau deshalb ist sie die erste Frage, die man einem vorgelegten Backtest stellen sollte, und die erste Zahl, die man beim eigenen mitschreiben muss.

21.10 Micro-Quiz

Micro-Quiz 21: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Zwei Strategien haben dieselbe Jahresrendite und dieselbe Sharpe-Kennzahl. Was wissen Sie über ihren Verlauf? (a) Er ist im Wesentlichen gleich — beide Kennzahlen zusammen beschreiben ihn vollständig. (b) **Nichts.** Beide Kennzahlen mitteln über die Zeit und verlieren die Reihenfolge. Dieselben Tagesrenditen können zu einem Rückschlag von 19 % oder von 98 % führen. (c) Die Strategie mit der höheren Schwankung hat den größeren Rückschlag.

2. Sie haben 50 Strategievarianten getestet und die beste erreicht p = 0,01. Wie bewerten Sie das? (a) Hochsignifikant — die Wahrscheinlichkeit eines Zufallsbefunds liegt bei 1 %. (b) Bei 50 Versuchen ist ein p-Wert von 0,01 unauffällig: Schon rein zufällig erwartet man etwa 2,5 Treffer unter dem 5-%-Niveau. Die Bonferroni-Schwelle läge bei 0,001. (c) Der p-Wert ist für Handelsstrategien grundsätzlich ungeeignet.

3. Warum ist ein zurückgehaltener Testzeitraum nur einmal** verwendbar?**** (a) Weil die Daten danach veraltet sind. (b) Weil man ab dem Moment, in dem man das Ergebnis kennt und daraufhin etwas ändert, wieder anfängt zu probieren — der Zeitraum ist dann Teil der Suche geworden und nicht mehr unabhängig. (c) Weil sich die Kennzahlen bei wiederholter Berechnung numerisch verschlechtern.

21.11 Selbsttest

Antworten: Anhang A.

1. Was ist die goldene Regel des Backtestens?
2. Warum sind Kalender-Monatsanfänge als Rebalancing-Termine falsch?
3. Wie funktioniert der Lookahead-Selbsttest, und warum ist er so wirksam?
4. Warum muss man Gewichtsdrift modellieren?
5. Warum ist Data Snooping gefährlicher als ein Programmierfehler?

21.12 Zusammenfassung

- **Sechs Bausteine** in fester Reihenfolge: Daten, Signal, Risiko, Optimierung, Rebalancing, Auswertung.
- **Die goldene Regel:** Zum Zeitpunkt t nur Information von t. Der Selbsttest mit manipulierter Zukunft prüft das automatisch.
- **Rebalancing-Termine müssen Handelstage sein** — sonst fallen bis zu 28 % still aus.
- **Zählen Sie mit:** geplante gegen ausgeführte Rebalancings, kumulierter Turnover, Gesamtkosten. Abweichungen sind Fehlerhinweise.

- **Fünf Selbsttäuschungen:** Lookahead, Survivorship, Kostenblindheit, Overfitting, Data Snooping. Die ersten drei stecken im Code und sind lesbar; die letzten beiden entstehen in der **Arbeitsweise** und hinterlassen im Programm keine Spur.
- **Ohne die Zahl der ausprobierten Varianten ist kein Backtest-Ergebnis interpretierbar.** Auf reinem Rauschen liefert die Beste von hundert Versuchen im Mittel Sharpe 1,13 bei p = 0,0094. Protokollieren Sie mit, korrigieren Sie die Schwelle (Bonferroni, Deflated Sharpe) — und halten Sie einen Zeitraum zurück, den Sie genau **einmal** verwenden.
- **Rendite und Sharpe verlieren die Reihenfolge.** Dieselben Tagesrenditen können zu einem Rückschlag von 19 % oder 98 % führen. Berichten Sie den maximalen Rückschlag immer mit.
- **Ein guter Backtest ist notwendig, nicht hinreichend.** Zwischen ihm und dem Betrieb liegen Papierhandel, Kapazitätsanalyse und Betriebsrisiken.

Ausblick. Kapitel 22 zieht die Summe aus allen Teilen: die typischen Praxisfallen und der Weg vom Skript zum produktiven System.

Synthese Teil IV — Zwei Domänen, dieselbe Mathematik

Sechs Kapitel, zwei Branchen, die sich für unvergleichbar halten. Die Werkstatt plant Maschinen, das Depot plant Titel — und beide lösen dasselbe Problem: **knappe Mittel auf konkurrierende Verwendungen verteilen, unter Unsicherheit.**

Dieselbe Struktur, andere Namen

Werkstatt und Kraftwerk	Depot und Handel	Was mathematisch dahintersteht
Produkt, Maschinenstunde	Titel, Kapital	Entscheidungsvariable
Deckungsbeitrag	erwartete Rendite	linearer Zielterm
Kapazität, Liefervertrag	Positionsgrenze, Sektorlimit	Nebenbedingung
Rüstkosten, Anfahrkosten	Transaktionskosten	Fixkosten mit Binärvariable (B1)
Lastabwurf, Konventionalstrafe	Tail-Verlust	Straf- bzw. Risikoterm
Wind- und Nachfrageszenarien	Renditeszenarien	Szenariomenge

Und wo die Analogie endet, benennt Kapitel 16 an drei Stellen ausdrücklich — das ist der wertvollere Teil. Eine Analogie, die man nicht begrenzen kann, ist keine Erkenntnis, sondern eine Redewendung.

Was dieser Teil gemessen hat

Behauptung	Gemessen	Wo
„Planen mit der Prognose ist gut genug.“	Der Erwartungswert-Plan kostet **149 % mehr** (874 870 € statt 351 356 €) und wirft in **28 von 40** Szenarien Last ab	Kapitel 17
„Dann sparen wir eben beim Brennstoff.“	Genau das tut er: 12 725 € gespart, **536 239 €** Lastabwurf bezahlt — das 42-fache. Die Anfahrkosten sind in beiden Plänen gleich	Kapitel 17
„Der VaR sagt, wie riskant es ist.“	VaR **1,86 %**, CVaR **2,99 %** — und der schlechteste Tag der Stichprobe (**−23,0 %**) ändert am VaR **nichts**	Kapitel 20
„Die Kovarianzmatrix ist eine Eingabe wie jede andere.“	Aus kurzen Reihen geschätzt ist sie schlecht konditioniert; die Optimierung setzt dann auf Schätzrauschen	Kapitel 18

Drei Fehler, die dieser Teil verhindert

1. **Bei Markowitz einsteigen.** Wer Kapitel 18 überspringt, optimiert seine eigenen Schätzfehler und wundert sich über Gewichte, die niemand halten würde.
2. **Den VaR für ein Risikomaß halten.** Er sagt, **ob** die Schwelle gerissen wird, nicht **wie schlimm** es dahinter aussieht — und er ist nicht kohärent. Optimiert wird der CVaR.
3. **Trägheit unterschätzen.** Im Kraftwerkspark ist nicht die Prognosegüte das Problem, sondern dass sich ein Block mit acht Stunden Mindestlaufzeit um 18 Uhr nicht mehr herbeirufen lässt. Dieselbe Trägheit heißt im Depot Transaktionskosten.

Wenn Sie nur eines mitnehmen

 Wer die Struktur erkennt, kann sein Werkzeug mitnehmen, wenn er die Branche wechselt. Wer nur die Bibliothek kennt, fängt jedes Mal von vorn an — und wer die Grenzen der Analogie nicht kennt, überträgt irgendwann auch das, was nicht überträgt.

Teil V: Praxis

Kapitel 22: Praxisfallen und der Weg zum produktiven Einsatz

Kapitel auf einen Blick

Worum geht es? Um alles, was zwischen einem funktionierenden Notebook und einem System steht, das Menschen tatsächlich benutzen.

Voraussetzungen: Die vorangegangenen Kapitel — dieses Kapitel zieht die Summe.

Danach können Sie: Unlösbarkeit systematisch diagnostizieren und auffangen, Ergebnisse erklärbar machen, Laufzeiten begrenzen und ein OR-System betriebsfähig aufsetzen.

Zeitbedarf: ca. 4 Stunden.

Programme:
`Infeasibility_Diagnose.py`
`Erklaerbarkeit.py`
`Constraint_Attribution.py`
`Betriebsueberwachung.py`
`or_kern.py`
`Solverwechsel_CPSAT_HiGHS.py`

Notebook: `Notebooks_04/praxisfallen.ipynb`

22.1 In 5 Minuten gelöst

Die vorangegangenen vierzehn Kapitel enden fast alle mit derselben Empfehlung: *Prüfen Sie das Ergebnis gegen die Wirklichkeit.* Dieses Kapitel beginnt damit, diese Prüfung als wiederverwendbaren Baustein hinzuschreiben.

In 5 Minuten gelöst: Die Abnahmeprüfung, die in jedes Modell gehört

```python
def pruefe_loesung(x, A, b, kosten, zielwert, ganzzahlig=(), toleranz=1e-6):
    """Prueft eine Loesung gegen die Anforderungen - OHNE den Solver zu fragen."""
    beanstandungen = []

    verbrauch = A @ x                                    # 1. Werden Grenzen
    ↪ eingehalten?
    for i, (ist, grenze) in enumerate(zip(verbrauch, b)):
        if ist > grenze + toleranz:
            beanstandungen.append(f"Bedingung {i}: {ist:.4f} > {grenze:.4f}")

    for j in ganzzahlig:                                 # 2. Ist ganzzahlig auch
     ↪ ganzzahlig?
        if abs(x[j] - round(x[j])) > toleranz:
            beanstandungen.append(f"x[{j}] = {x[j]!r} ist nicht ganzzahlig")
```

```
    nachgerechnet = float(kosten @ x)                    # 3. Stimmt der Zielwert?
    if abs(nachgerechnet - zielwert) > toleranz * max(1.0, abs(nachgerechnet)):
        beanstandungen.append(
            f"Zielwert {zielwert:.4f} passt nicht zu den Daten
             ↪  ({nachgerechnet:.4f})")

    return beanstandungen
```

Zwölf Zeilen, drei Prüfungen — und jede davon hätte einen der Denkfehler dieses Buches gefunden:

Prüfung	fängt	aus
Grenzen einhalten	die pro Produkt statt insgesamt formulierte Kapazität	Abschnitt 1.10
Grenzen einhalten	die vergessene Ladungsdimension im Routing	Abschnitt 8.7
Ganzzahligkeit	Binärvariablen, die bei 10^{-8} hängen (Big-M)	Abschnitt 6.10
Zielwert nachrechnen	Fixkosten, die im Modell nicht verbucht wurden	Abschnitt 6.10

 Merksatz Entscheidend ist, was diese Funktion **nicht** benutzt: kein Solver-Objekt, keine Modellvariable, keine Dimension aus der Bibliothek. Sie bekommt nur die ausgegebene Lösung und die Anforderungen — und rechnet unabhängig nach. Eine Prüfung, die dieselben Bausteine verwendet wie das Modell, prüft das Modell gegen sich selbst und findet nichts.

Und was sie nicht kann. Sie prüft, ob die gefundene Lösung **zulässig** ist. Sie kann nicht prüfen, ob es eine **bessere** gäbe — genau daran ist die Prüfung in Abschnitt 13.7 gescheitert, wo das Modell einen korrekten Kostenbetrag für einen unnötig teuren Plan meldete. Dafür braucht es ein zweites, unabhängiges Verfahren oder eine bekannte Schranke. Die restlichen Abschnitte dieses Kapitels handeln davon, was im Betrieb sonst noch schiefgeht — und wie man es bemerkt.

22.2 Lernziele

Nach diesem Kapitel können Sie …

1. … die fünf typischen Praxisfallen benennen und je ein Gegenmittel nennen.
2. … ein unlösbares Modell in einen „am wenigsten schlechten“ Plan verwandeln.
3. … Solver-Ergebnisse erklärbar machen (*Explainable OR*).
4. … für jede Nebenbedingung beziffern, was sie den Plan kostet — und begründen, warum der Schattenpreis diese Frage nur für die *nächste* Einheit beantwortet.
5. … Zeitlimits und Optimalitätslücken sinnvoll setzen.
6. … die Architektur einer produktionsreifen OR-Plattform skizzieren.

7. … eine **modellunabhängige Abnahmeprüfung** schreiben, die eine Lösung gegen die Anforderungen prüft statt gegen das Modell.
8. … Status, Gap und Zeitausschöpfung protokollieren und begründen, warum eine konstante Laufzeit bei Zeitlimit ein Warnsignal ist.
9. … ein Modell in Domänenschicht, Modellbauer und Lösungs-DTO trennen — und begründen, warum sich das nicht wegen des Solverwechsels lohnt, sondern wegen der Prüfbarkeit.
10. … die Statuswerte verschiedener Solverbibliotheken auf eine gemeinsame Sprache abbilden.

22.3 Die fünf typischen Praxisfallen

DIE 5 FALLSTRICKE IN DER PRAXIS	
1. INFEASIBILITY-FALLE	Unlösbare Modelle durch starre, widersprüchliche Hard Constraints
2. ERKLÄRBARKEITS-LÜCKE	Mangelnde Akzeptanz wegen fehlender Begründungen (Black-Box)
3. OVERFITTING / REGIME-SHIFT	Perfekt auf die Vergangenheit optimiert – scheitert in Krisen
4. DATENVERZUG / LOOKAHEAD	Signale nutzen Daten, die zum Handlungszeitpunkt fehlten
5. LAUFZEIT-EXPLOSION	NP-schwere Modelle verfehlen reale Timeouts bei Wachstum

Abbildung 29: Abb. 22.1: Die fünf typischen Praxisfallen

Falle 1 — Infeasibility

Wenn Parameter unglücklich zusammentreffen — drei Mitarbeitende gleichzeitig krank, kein qualifizierter Ersatz —, meldet der Solver schlicht `INFEASIBLE`.

- **Falsch:** Das System stürzt ab oder zeigt „Fehler“. Der Disponent steht ohne Plan da.
- **Richtig: Hierarchische Relaxation.** Jede harte Bedingung, die im Notfall gebrochen werden darf, erhält eine teure Schlupfvariable. So liefert der Solver immer einen Plan — und zeigt zugleich präzise, **wo** es klemmt.

```
#!/usr/bin/env python3

# Infeasibility_Diagnose.py
"""
Kapitel Praxisfallen: Aus INFEASIBLE einen Notfallplan machen.

Demonstriert die hierarchische Relaxation an einem Dienstplan, der in der
harten Fassung unloesbar ist, und zeigt, wie der Solver die Ursache benennt.
"""

from ortools.sat.python import cp_model

SLOTS = ["Mo frueh", "Mo spaet", "Di frueh", "Di spaet"]
FAECHER = ["Mathematik", "Physik", "Mathematik", "Chemie"]   # Chemie kann niemand!
```

```
PERSONAL = ["Alice", "Bob", "Carla"]
QUALIFIKATION = {
    "Alice": {"Mathematik", "Physik"},
    "Bob":   {"Mathematik"},
    "Carla": {"Physik", "Mathematik"},
}
MAX_PRO_PERSON = 2
STRAFE_UNBESETZT = 10_000          # sehr teuer, aber nicht unmoeglich
STRAFE_UEBERLAST = 500             # teuer, aber billiger als Ausfall


def plane(harte_fassung: bool):
    """harte_fassung=True: klassisch (kann INFEASIBLE werden).
       harte_fassung=False: mit Schlupfvariablen (immer loesbar)."""
    modell = cp_model.CpModel()
    x = {(p, s): modell.NewBoolVar(f"x_{p}_{s}")
         for p in PERSONAL for s in range(len(SLOTS))}

    strafen = []
    unbesetzt = {}
    ueberlast = {}

    for s in range(len(SLOTS)):
        if harte_fassung:
            modell.AddExactlyOne(x[p, s] for p in PERSONAL)
        else:
            # Schlupfvariable: der Slot DARF unbesetzt bleiben - gegen hohe Strafe
            unbesetzt[s] = modell.NewBoolVar(f"unbesetzt_{s}")
            modell.AddExactlyOne([x[p, s] for p in PERSONAL] + [unbesetzt[s]])
            strafen.append(unbesetzt[s] * STRAFE_UNBESETZT)

    # Qualifikation bleibt IMMER hart - fachfremder Unterricht ist keine Option
    for s, fach in enumerate(FAECHER):
        for p in PERSONAL:
            if fach not in QUALIFIKATION[p]:
                modell.Add(x[p, s] == 0)

    for p in PERSONAL:
        last = sum(x[p, s] for s in range(len(SLOTS)))
        if harte_fassung:
            modell.Add(last <= MAX_PRO_PERSON)
        else:
            # Ueberlast erlaubt - aber teuer
            ueberlast[p] = modell.NewIntVar(0, len(SLOTS), f"ueberlast_{p}")
            modell.Add(last <= MAX_PRO_PERSON + ueberlast[p])
            strafen.append(ueberlast[p] * STRAFE_UEBERLAST)

    if strafen:
```

```
        modell.Minimize(sum(strafen))

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = 5.0
    status = loeser.Solve(modell)
    return loeser, status, x, unbesetzt, ueberlast


if __name__ == "__main__":
    print("=" * 78)
    print("  VARIANTE A: KLASSISCH MIT LAUTER HARTEN BEDINGUNGEN")
    print("=" * 78)
    loeser, status, *_ = plane(harte_fassung=True)
    print(f"Solver-Status: {loeser.StatusName(status)}")
    if status == cp_model.INFEASIBLE:
        print("Das System kann keinen Plan liefern. Der Anwender erfaehrt NICHT,")
        print("welche Regel das Problem verursacht - nur, dass es nicht geht.\n")

    print("=" * 78)
    print("  VARIANTE B: MIT HIERARCHISCHER RELAXATION")
    print("=" * 78)
    loeser, status, x, unbesetzt, ueberlast = plane(harte_fassung=False)
    print(f"Solver-Status: {loeser.StatusName(status)} | "
          f"Strafkosten: {loeser.ObjectiveValue():.0f}\n")

    print(f"{'Slot':<12} {'Fach':<12} {'Zuweisung':<18} {'Bemerkung'}")
    print("-" * 78)
    for s, slot in enumerate(SLOTS):
        if loeser.Value(unbesetzt[s]):
            print(f"{slot:<12} {FAECHER[s]:<12} {'-- UNBESETZT --':<18} "
                  f"Kein qualifiziertes Personal verfuegbar")
        else:
            person = next(p for p in PERSONAL if loeser.Value(x[p, s]))
            print(f"{slot:<12} {FAECHER[s]:<12} {person:<18}")

    print("\n--- Diagnose ---")
    for s, slot in enumerate(SLOTS):
        if loeser.Value(unbesetzt[s]):
            fach = FAECHER[s]
            qualifiziert = [p for p in PERSONAL if fach in QUALIFIKATION[p]]
            print(f"  '{slot}' ({fach}): {len(qualifiziert)} qualifizierte Personen "
                  f"{qualifiziert if qualifiziert else '-> URSACHE: niemand kann dieses
                   ↪ Fach'}")
    for p in PERSONAL:
        if loeser.Value(ueberlast[p]):
            print(f"  {p} arbeitet {loeser.Value(ueberlast[p])} Stunden ueber der Grenze.")

    print("\nDer Anwender bekommt jetzt einen Plan PLUS eine konkrete Ursache -")
```

```
    print("und kann handeln: Vertretung von aussen holen, Stunde verlegen,")
    print("oder die Klasse zusammenlegen.")
    print("=" * 78)
```

Falle 2 — Der Black-Box-Effekt

Ein Planer oder Trader lehnt ein System ab, wenn er die Zuweisungen nicht nachvollziehen kann. **Akzeptanz ist keine Nebensache, sondern Voraussetzung für den Betrieb.**

Jede Entscheidung braucht einen **Constraint-Trace**:

> *„Warum bekommt Person A die Stunde und nicht Person B?“ → „Weil B heute bereits zwei Vertretungsstunden hat und bei einer dritten die gesetzliche Ruhezeit von 11 Stunden unterschritten würde. A hat 80 Strafpunkte weniger.“*

Bei Portfolioentscheidungen: **Kostenzerlegung** ausweisen — Alpha-Ertrag, Risikoprämie, Transaktionskostenstrafe (siehe das Muster in Kapitel 7 und Kapitel 20).

> **Die Faustregel für Erklärbarkeit** Für jede Entscheidung sollten Sie drei Fragen beantworten können: 1. **Warum diese Lösung?** — Zielfunktionswert und seine Bestandteile. 2. **Warum nicht die naheliegende Alternative?** — Welche Bedingung verhindert sie? 3. **Was würde die Lösung verbessern?** — Schattenpreise: welcher Engpass, welcher Wert.
>
> Kapitel 5 liefert für Frage 3 das Werkzeug. Nutzen Sie es — Schattenpreise sind in Managementgesprächen oft wertvoller als die Lösung selbst.

Ein vollständiger Constraint-Trace beantwortet alle drei Fragen der Faustregel für eine einzelne Zuweisung — inklusive einer Unterscheidung, die in der Praxis oft übersehen wird: eine Bedingung kann **blockieren**, ohne **etwas zu kosten**.

```
#!/usr/bin/env python3

# Erklaerbarkeit.py
"""
Kapitel Praxisfallen: Ein Constraint-Trace, der die drei Fragen der Faustregel fuer
Erklaerbarkeit (Abschnitt 'Die fuenf typischen Praxisfallen') fuer eine
einzelne Zuweisung beantwortet:
Warum diese Loesung? Warum nicht die Alternative? Was wuerde sie verbessern?
"""

from ortools.sat.python import cp_model

PERSONAL = ["Anna", "Ben", "Clara"]
BEREITS_STUNDEN = {"Anna": 1, "Ben": 2, "Clara": 0}   # diese Woche schon geleistet
MAX_STUNDEN = 2                                        # gesetzliche Obergrenze
STRAFE_VORBELASTUNG = 50                               # pro bereits geleisteter Stunde
STRAFKOSTEN_WUNSCH = {"Anna": 0, "Ben": 30, "Clara": 100}


def loese(max_stunden: int):
```

```
    modell = cp_model.CpModel()
    x = {p: modell.NewBoolVar(f"x_{p}") for p in PERSONAL}
    modell.AddExactlyOne(x.values())

    kosten = []
    for p in PERSONAL:
        if BEREITS_STUNDEN[p] + 1 > max_stunden:
            modell.Add(x[p] == 0)                        # hartes Ausschlusskriterium
        strafe = BEREITS_STUNDEN[p] * STRAFE_VORBELASTUNG + STRAFKOSTEN_WUNSCH[p]
        kosten.append(strafe * x[p])

    modell.Minimize(sum(kosten))
    loeser = cp_model.CpSolver()
    status = loeser.Solve(modell)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        return None
    gewinner = next(p for p in PERSONAL if loeser.Value(x[p]))
    return gewinner, loeser.ObjectiveValue()


def gesamtkosten(p: str) -> int:
    return BEREITS_STUNDEN[p] * STRAFE_VORBELASTUNG + STRAFKOSTEN_WUNSCH[p]


if __name__ == "__main__":
    gewinner, kosten_opt = loese(MAX_STUNDEN)

    print("=" * 78)
    print("  CONSTRAINT-TRACE: Warum bekommt", gewinner, "die Vertretungsstunde?")
    print("=" * 78)

    print("\n1. WARUM DIESE LOESUNG?")
    v, w = BEREITS_STUNDEN[gewinner] * STRAFE_VORBELASTUNG, STRAFKOSTEN_WUNSCH[gewinner]
    print(f"   {gewinner}: Vorbelastung {v} Strafpunkte + Wunsch {w} Strafpunkte "
          f"= {v + w} Strafpunkte insgesamt (Minimum).")

    print("\n2. WARUM NICHT DIE NAHELIEGENDEN ALTERNATIVEN?")
    for p in PERSONAL:
        if p == gewinner:
            continue
        if BEREITS_STUNDEN[p] + 1 > MAX_STUNDEN:
            print(f"   {p}: AUSGESCHLOSSEN - hat bereits {BEREITS_STUNDEN[p]} Stunden, "
                  f"eine dritte würde die Obergrenze von {MAX_STUNDEN} Stunden verletzen "
                  f"(harte Bedingung, nicht verhandelbar).")
        else:
            differenz = gesamtkosten(p) - kosten_opt
            print(f"   {p}: zulässig, aber {differenz:.0f} Strafpunkte teurer als "
                  f"{gewinner} ({gesamtkosten(p):.0f} statt {kosten_opt:.0f}).")
```

```
    print("\n3. WAS WUERDE DIE LOESUNG VERBESSERN?")
    gewinner_gelockert, kosten_gelockert = loese(MAX_STUNDEN + 1)
    if kosten_gelockert < kosten_opt:
        print(f"   Obergrenze auf {MAX_STUNDEN + 1} Stunden gelockert: {gewinner_gelockert} "
              f"uebernimmt jetzt zu {kosten_gelockert:.0f} Strafpunkten "
              f"({kosten_opt - kosten_gelockert:.0f} weniger als heute).")
        print("   -> Diese Bedingung hat tatsaechlich einen Preis.")
    else:
        print(f"   Obergrenze auf {MAX_STUNDEN + 1} Stunden gelockert: Ergebnis bleibt bei "
              f"{gewinner} mit {kosten_gelockert:.0f} Strafpunkten (unveraendert).")
        blockiert = ", ".join(p for p in PERSONAL
                              if BEREITS_STUNDEN[p] + 1 > MAX_STUNDEN and p != gewinner)
        print(f"   -> Die Obergrenze blockiert zwar {blockiert}, kostet aber gerade NICHTS:")
        print("   Selbst ohne sie waere die Loesung dieselbe.")
        print("   Blockierend und kostenrelevant sind zwei verschiedene Dinge.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  CONSTRAINT-TRACE: Warum bekommt Anna die Vertretungsstunde?
==============================================================================

1. WARUM DIESE LOESUNG?
   Anna: Vorbelastung 50 Strafpunkte + Wunsch 0 Strafpunkte = 50 Strafpunkte insgesamt
    ↪  (Minimum).

2. WARUM NICHT DIE NAHELIEGENDEN ALTERNATIVEN?
   Ben: AUSGESCHLOSSEN - hat bereits 2 Stunden, eine dritte würde die Obergrenze von 2
    ↪  Stunden verletzen (harte Bedingung, nicht verhandelbar).
   Clara: zulässig, aber 50 Strafpunkte teurer als Anna (100 statt 50).

3. WAS WUERDE DIE LOESUNG VERBESSERN?
   Obergrenze auf 3 Stunden gelockert: Ergebnis bleibt bei Anna mit 50 Strafpunkten
    ↪  (unveraendert).
   -> Die Obergrenze blockiert zwar Ben, kostet aber gerade NICHTS:
   Selbst ohne sie waere die Loesung dieselbe.
   Blockierend und kostenrelevant sind zwei verschiedene Dinge.
==============================================================================
```

Der letzte Punkt ist der eigentliche Lerneffekt: Die Obergrenze schließt Ben zwar technisch aus, aber selbst wenn er dürfte, wäre er wegen seiner Vorbelastung immer noch teurer als Anna. **Eine bindende Bedingung zu lockern hilft nur, wenn sie tatsächlich die kostenrelevante ist** — sonst ist die Mühe umsonst.

Derselbe Unterschied kehrt auf Modellebene wieder, und dort lässt er sich in Euro beziffern: Abschnitt 22.4 misst für jede Bedingung, was sie den ganzen Plan kostet — und baut daraus einen Bericht, den man vorlegen kann.

Falle 3 — Regimewechsel und Schätzfehler

Ein Modell, das auf einer mehrjährigen Aufwärtsphase kalibriert wurde, kennt keine Liquiditätskrise und keine Zinswende.

- **Robuste Schätzung** statt roher Historie: Ledoit-Wolf-Shrinkage (Kapitel 18), Szenarien (Kapitel 12).
- **Harte Obergrenzen** für Einzelpositionen und Sektoren — auch dann, wenn der Solver rechnerisch alles in einen Titel legen möchte. Die Grenze kostet Ertrag im Normalfall und rettet im Ernstfall.
- **Rollierende Neuschätzung** statt einmaliger Kalibrierung.

Falle 4 — Lookahead- und Survivorship-Bias

Ausführlich in Kapitel 21 behandelt. Die Kurzfassung: Beide Fehler machen Backtests systematisch zu gut, ohne eine Fehlermeldung zu erzeugen. Der Lookahead-Selbsttest aus Abschnitt 21.5 findet den ersten automatisch.

Falle 5 — Laufzeitexplosion

Exakte Optimalität kann bei MILP oder CP-SAT Minuten bis Stunden dauern — und die Laufzeit wächst nicht linear mit der Problemgröße.

Einsatzszenario	Empfohlenes Zeitlimit	Akzeptabler MIP-Gap
Interaktive Oberfläche	5–10 Sekunden	2–5 %
Batch tagsüber	1–5 Minuten	1–2 %
Nachtlauf	30–60 Minuten	0,1–1 %
Strategische Planung	Stunden	exakt anstreben

 Warum ein Gap von 2 % fast immer genügt Der MIP-Gap misst den Abstand zwischen der besten gefundenen Lösung und der besten **beweisbaren Schranke**. Ein Gap von 2 % heißt: „Diese Lösung ist höchstens 2 % vom theoretischen Optimum entfernt."

Dem gegenüber steht die **Datenunsicherheit**: Ihre Nachfrageprognose ist um ±10 % genau, Ihre Fahrzeiten um ±15 %. Die letzten 2 % Optimalität in einem Modell zu erkämpfen, dessen Eingangsdaten um 10 % schwanken, ist verlorene Zeit. **Optimieren Sie nicht genauer, als Ihre Daten sind.**

22.4 Constraint Attribution: welche Bedingung kostet wie viel?

Der Constraint-Trace beantwortet „warum **diese** Zuweisung?". Der Deletion Filter in Anhang C beantwortet „warum geht es **gar nicht**?". Der häufigste Fall in der Praxis liegt dazwischen, und für ihn hat bisher niemand ein Werkzeug bekommen:

> *Das Modell rechnet. Der Plan ist zulässig. Und trotzdem ist er enttäuschend — nur weiß niemand, woran es liegt.*

Die Antwort heißt **Constraint Attribution**: Jede Bedingungsgruppe wird einzeln gelockert, und gemessen wird, was sich am Zielwert ändert. Das ist dasselbe Verfahren wie der Deletion Filter — nur mit **Kosten** statt Zulässigkeit als Kriterium.

Gerechnet wird auf derselben Fabrik wie die Konfliktsuche in Anhang C, aber **nach der Reparatur**: Dort war die Lackierkapazität von 150 Stunden der Kern des Widerspruchs ($2 \cdot 40 + 3 \cdot 30 = 170 > 150$). Eine zweite Schicht hat sie auf 210 gebracht, der Mindestumsatz wurde auf 10 000 € gesenkt. Jetzt ist das Modell lösbar — und die Frage lautet nicht mehr „warum nicht?“, sondern „was kostet uns was?“.

```
#!/usr/bin/env python3

# Constraint_Attribution.py
"""
Kapitel Praxisfallen: Welche Bedingung kostet wie viel - und was sagt man
dem Management?

Zwischen den beiden bekannten Faellen klafft eine Luecke:

  * Erklaerbarkeit.py beantwortet "warum diese eine Zuweisung?"
  * Konfliktsuche.py (Anhang Fehlerdiagnose) beantwortet "warum geht es gar
    nicht?"

Der haeufigste Fall in der Praxis liegt dazwischen: Das Modell rechnet, der
Plan ist zulaessig - und trotzdem unbefriedigend. Niemand weiss, WORAN es
liegt. Genau das beantwortet Constraint Attribution.

Gerechnet wird auf derselben Fabrik wie in Konfliktsuche.py, nur nach der
Reparatur: Dort war die Lackierkapazitaet von 150 Stunden der Kern des
Widerspruchs (2*40 + 3*30 = 170 > 150). Eine zweite Schicht hat sie auf 210
gebracht, und der Mindestumsatz wurde auf 10.000 EUR gesenkt. Jetzt ist das
Modell loesbar - und die Frage lautet nicht mehr "warum nicht?", sondern
"was kostet uns was?".

Gezeigt werden vier Dinge:
  1. Welche Bedingungen binden - und dass "bindend" nicht "teuer" heisst.
  2. Dass der Schattenpreis eine MOMENTAUFNAHME ist: hochgerechnet auf eine
     realistische Sonderschicht verspricht er hier das Doppelte des
     tatsaechlichen Nutzens.
  3. Dass ein Wunsch aus dem Vertrieb nicht teuer, sondern unmoeglich sein
     kann - und welche Bedingungen ihn gemeinsam blockieren.
  4. Wie aus alledem ein Bericht in Alltagssprache wird.

Benoetigt: numpy, scipy
"""

from __future__ import annotations

import textwrap
```

```
import numpy as np
from scipy.optimize import linprog

PRODUKTE = ["Rahmen", "Gehaeuse", "Deckel", "Traeger", "Halter"]
DECKUNGSBEITRAG = np.array([35.0, 28.0, 9.0, 22.0, 6.0])   # EUR je Stueck

# Dieselben Bedingungen wie in Konfliktsuche.py, mit den beiden Reparaturen.
# Der sprechende Name ist keine Kosmetik: Ohne ihn ist kein Bericht moeglich,
# der jemandem ausserhalb der Modellierung etwas sagt.
BEDINGUNGEN: list[tuple[str, list[float], str, float]] = [
    ("Kapazitaet Montage",       [3, 2, 1, 4, 2],      "<=",   400),
    ("Kapazitaet Lackieren",     [2, 3, 0, 1, 0],      "<=",   210),
    ("Kapazitaet Pruefung",      [1, 1, 1, 1, 1],      "<=",   300),
    ("Liefervertrag Rahmen",     [1, 0, 0, 0, 0],      ">=",    40),
    ("Liefervertrag Gehaeuse",   [0, 1, 0, 0, 0],      ">=",    30),
    ("Liefervertrag Deckel",     [0, 0, 1, 0, 0],      ">=",    20),
    ("Liefervertrag Traeger",    [0, 0, 0, 1, 0],      ">=",    25),
    ("Marktgrenze Rahmen",       [1, 0, 0, 0, 0],      "<=",   120),
    ("Marktgrenze Gehaeuse",     [0, 1, 0, 0, 0],      "<=",    90),
    ("Marktgrenze Deckel",       [0, 0, 1, 0, 0],      "<=",   150),
    ("Marktgrenze Halter",       [0, 0, 0, 0, 1],      "<=",   200),
    ("Mindestumsatz",            [90, 70, 30, 60, 20], ">=", 10000),
    ("Sortimentsbreite",         [0, 0, 1, 1, 1],      ">=",   100),
    ("Lackierbudget Schicht 2",  [0, 1, 0, 1, 0],      "<=",    55),
]

# Was liesse sich im Ernstfall wirklich bewegen - und um wie viel? Nicht "eine
# Einheit", sondern der Schritt, den ein Planer tatsaechlich gehen kann: eine
# Sonderschicht, ein nachverhandelter Vertrag. Genau an dieser Groesse
# scheitert das blosse Hochrechnen des Schattenpreises.
STELLSCHRAUBEN: dict[str, tuple[float, str]] = {
    "Kapazitaet Montage":      (+50, "eine Sonderschicht Montage"),
    "Kapazitaet Lackieren":    (+30, "eine Sonderschicht Lackieren"),
    "Kapazitaet Pruefung":     (+50, "eine Sonderschicht Pruefung"),
    "Lackierbudget Schicht 2": (+10, "10 Stunden mehr Budget in Schicht 2"),
    "Liefervertrag Traeger":   (-10, "10 Stueck weniger Abnahmepflicht Traeger"),
    "Liefervertrag Gehaeuse":  (-10, "10 Stueck weniger Abnahmepflicht Gehaeuse"),
}

WUNSCH_MENGE = 55.0      # der Vertrieb haette gern mehr Rahmen
WUNSCH_PRODUKT = "Rahmen"


def loese(bedingungen: list[tuple[str, list[float], str, float]]):
    """Maximiert den Deckungsbeitrag. linprog minimiert, daher das Vorzeichen."""
    matrix, rechte_seite = [], []
    for _, koeffizienten, richtung, grenze in bedingungen:
```

```
        zeile = np.array(koeffizienten, dtype=float)
        matrix.append(zeile if richtung == "<=" else -zeile)
        rechte_seite.append(grenze if richtung == "<=" else -grenze)
    return linprog(-DECKUNGSBEITRAG,
                   A_ub=np.array(matrix), b_ub=np.array(rechte_seite),
                   bounds=[(0, None)] * len(PRODUKTE), method="highs")


def geaendert(name: str, delta: float,
              basis=None) -> list[tuple[str, list[float], str, float]]:
    """Dieselben Bedingungen, eine davon um delta verschoben."""
    return [(n, k, r, g + delta if n == name else g)
            for n, k, r, g in (basis or BEDINGUNGEN)]


def auswertung(ergebnis, bedingungen=None):
    """Schlupf und Schattenpreis je Bedingung."""
    x = ergebnis.x
    dual = ergebnis.ineqlin.marginals
    zeilen = []
    for i, (name, koeff, richtung, grenze) in enumerate(bedingungen or BEDINGUNGEN):
        links = float(np.array(koeff, dtype=float) @ x)
        schlupf = (grenze - links) if richtung == "<=" else (links - grenze)
        zeilen.append((name, links, richtung, grenze, schlupf, abs(dual[i])))
    return zeilen


def blockierer(bedingungen, zusatz) -> list[str]:
    """Welche Bedingungen verhindern 'zusatz' - einzeln weggelassen.

    Dasselbe Vorgehen wie der Deletion Filter in Konfliktsuche.py, nur mit
    umgekehrtem Vorzeichen: Dort wird gesucht, was den Widerspruch AUSMACHT,
    hier, was einen Wunsch VERHINDERT. Wird das Modell ohne eine Bedingung
    loesbar, gehoert sie zum Konflikt.
    """
    gefunden = []
    for i, (name, _, _, _) in enumerate(bedingungen):
        rest = [c for j, c in enumerate(bedingungen) if j != i] + [zusatz]
        if loese(rest).success:
            gefunden.append(name)
    return gefunden


def euro(betrag: float) -> str:
    return f"{betrag:,.2f} EUR"


if __name__ == "__main__":
```

```
basis = loese(BEDINGUNGEN)
db_basis = -basis.fun
plan = dict(zip(PRODUKTE, basis.x))

print("=" * 78)
print("  CONSTRAINT ATTRIBUTION: Was kostet uns welche Bedingung?")
print("=" * 78)
print(f"Status: {basis.message.split('(')[0].strip()}")
print(f"Deckungsbeitrag des Plans: {euro(db_basis)}\n")
print("  " + "  ".join(f"{p}: {m:.1f}" for p, m in plan.items()))

# --- 1. Wer begrenzt den Plan? ----------------------------------------
print("\n" + "=" * 78)
print("  (1) Welche Bedingungen binden - und was kostet die naechste Einheit?")
print("=" * 78)
print(f"{'Bedingung':<24} {'genutzt':>9} {'Grenze':>7} {'Schlupf':>8} "
      f"{'Schattenpreis':>11}")
print("-" * 78)
bindend = []
for name, links, richtung, grenze, schlupf, dual in auswertung(basis):
    marke = ""
    if abs(schlupf) < 1e-6:
        bindend.append((name, dual))
        marke = "  <-- bindend"
    print(f"{name:<24} {links:>9.1f} {grenze:>7.0f} {schlupf:>8.1f} "
          f"{dual:>11.2f}{marke}")

ohne_preis = [n for n, d in bindend if d < 1e-9]
print(f"\n  {len(bindend)} Bedingungen binden. Aber: "
      f"{', '.join(ohne_preis)} bindet")
print("  mit einem Schattenpreis von 0,00 EUR - sie beruehrt den Plan, ohne ihn")
print("  zu verteuern. Bindend und kostenrelevant sind zwei verschiedene Dinge.")

# --- 2. Der Schattenpreis als Momentaufnahme --------------------------
print("\n" + "=" * 78)
print("  (2) Was der Schattenpreis verschweigt")
print("=" * 78)
print("Der Schattenpreis gilt fuer die NAECHSTE Einheit. Ein Planer kauft aber")
print("keine Einheit, sondern eine ganze Schicht. Beides gegenuebergestellt:\n")
preise = {name: dual for name, *_, dual in auswertung(basis)}
print(f"{'Stellschraube':<24} {'Preis':>9} {'Schritt':>7} "
      f"{'hochgerechnet':>13} {'gemessen':>9} {'Abw.':>9}")
print("-" * 78)
wirkung = {}
for name, (delta, _) in STELLSCHRAUBEN.items():
    ergebnis = loese(geaendert(name, delta))
    echt = (-ergebnis.fun) - db_basis if ergebnis.success else 0.0
    linear = preise[name] * abs(delta)
```

```
        wirkung[name] = echt
        print(f"{name:<24} {preise[name]:>9.2f} {delta:>+7.0f} "
              f"{linear:>13.2f} {echt:>9.2f} {echt - linear:>+9.2f}")

    # Wo endet der Gueltigkeitsbereich? Schrittweise nachfahren.
    abweichler = max(STELLSCHRAUBEN,
                     key=lambda n: preise[n] * abs(STELLSCHRAUBEN[n][0]) - wirkung[n])
    delta_max = STELLSCHRAUBEN[abweichler][0]
    grenze_gueltig = 0.0
    schritt = delta_max / 60.0
    vorher = db_basis
    for i in range(1, 61):
        d = schritt * i
        e = loese(geaendert(abweichler, d))
        db = -e.fun if e.success else vorher
        if db - vorher > 1e-9:
            grenze_gueltig = d
        vorher = db
    print(f"\n  '{abweichler}': hochgerechnet "
          f"{preise[abweichler] * abs(delta_max):.2f} EUR, tatsaechlich "
          f"{wirkung[abweichler]:.2f} EUR.")
    print(f"  Der Preis von {preise[abweichler]:.2f} EUR gilt nur bis "
          f"+{grenze_gueltig:.0f} Stunden. Jede weitere Stunde")
    print("  dieser Sonderschicht bringt exakt nichts - der Engpass ist dann ein anderer.")

    nach_preis = sorted(STELLSCHRAUBEN, key=lambda n: -preise[n])[0]
    nach_wirkung = max(wirkung, key=wirkung.get)
    print(f"\n  Und die Ranglisten drehen sich: Nach Schattenpreis fuehrt")
    print(f"  '{nach_preis}' ({preise[nach_preis]:.2f} EUR je Einheit),")
    print(f"  nach tatsaechlicher Wirkung '{nach_wirkung}' "
          f"({wirkung[nach_wirkung]:.2f} EUR).")
    print("  Der Schattenpreis sagt, was eine Einheit wert ist - nicht, wie viele")
    print("  Einheiten zu haben sind.")

    # --- 3. Der Wunsch aus dem Vertrieb -----------------------------------
    print("\n" + "=" * 78)
    print(f"  (3) Was-waere-wenn: mindestens {WUNSCH_MENGE:.0f} "
          f"{WUNSCH_PRODUKT} statt {plan[WUNSCH_PRODUKT]:.1f}")
    print("=" * 78)
    wunsch = (f"Wunsch Vertrieb {WUNSCH_PRODUKT}",
              [1 if p == WUNSCH_PRODUKT else 0 for p in PRODUKTE], ">=", WUNSCH_MENGE)
    mit_wunsch = loese(BEDINGUNGEN + [wunsch])
    schuldige: list[str] = []
    engpass = noetig = None
    wunsch_gratis = False

    if mit_wunsch.success:
        print(f"Erfuellbar, kostet {euro(db_basis - (-mit_wunsch.fun))}.")
```

```
    else:
        schuldige = blockierer(BEDINGUNGEN, wunsch)
        print("Der Wunsch ist nicht teuer - er ist UNMOEGLICH.\n")
        print(f"Blockiert wird er von {len(schuldige)} Bedingungen gemeinsam. "
              f"Faellt eine davon")
        print("weg, ist er erfuellbar; keine allein ist 'der' Grund:")
        for name in schuldige:
            e = loese([c for c in BEDINGUNGEN if c[0] != name] + [wunsch])
            print(f"   ohne '{name}': moeglich, Deckungsbeitrag {euro(-e.fun)}")

        engpass = next((n for n in schuldige if n in STELLSCHRAUBEN), schuldige[0])
        for schritt in range(1, 61):
            if loese(geaendert(engpass, schritt) + [wunsch]).success:
                noetig = schritt
                break
        if noetig is not None:
            e = loese(geaendert(engpass, noetig) + [wunsch])
            ohne = loese(geaendert(engpass, noetig))
            wunsch_gratis = abs((-e.fun) - (-ohne.fun)) < 1e-6
            print(f"\n   Mit +{noetig} bei '{engpass}' wird der Wunsch erfuellbar:")
            print(f"   Deckungsbeitrag {euro(-e.fun)} "
                  f"({-e.fun - db_basis:+,.2f} gegenueber heute).")
            if wunsch_gratis:
                print(f"   Und ohne den Wunsch ergaebe dieselbe Lockerung "
                      f"{euro(-ohne.fun)} bei")
                print(f"   {ohne.x[PRODUKTE.index(WUNSCH_PRODUKT)]:.0f} "
                      f"{WUNSCH_PRODUKT} - der Wunsch kostet also NICHTS.")
                print("   Er war nie das Problem. Der Engpass war es.")

    # --- 4. Der Bericht ------------------------------------------------
    # Aus denselben Zahlen, die oben in Tabellen stehen, wird hier Prosa. Kein
    # Satz enthaelt eine Zahl, die nicht vorher gerechnet wurde - das ist der
    # ganze Trick und zugleich die Bedingung dafuer, dass der Bericht nicht
    # irgendwann leise falsch wird.
    print("\n" + "=" * 78)
    print("  (4) Und so steht es in der Vorlage fuer die Abteilungsleiterrunde")
    print("=" * 78)

    teuerste = sorted(((n, d) for n, d in bindend if d > 0), key=lambda t: -t[1])
    gross = STELLSCHRAUBEN[nach_wirkung][1]
    klein = STELLSCHRAUBEN[abweichler][1]

    absaetze = [
        f"Der Plan bringt {euro(db_basis)} Deckungsbeitrag. Begrenzt wird er von "
        f"{len(bindend)} Bedingungen, von denen {len(teuerste)} tatsaechlich Geld "
        f"kosten; \"{ohne_preis[0]}\" beruehrt den Plan, ohne ihn zu verteuern.",

        f"Teuerste Bindung ist \"{teuerste[0][0]}\" mit "
```

```
        f"{teuerste[0][1]:.2f} EUR je Einheit, gefolgt von \"{teuerste[1][0]}\" "
        f"mit {teuerste[1][1]:.2f} EUR.",

        f"Die groesste einzelne Verbesserung bringt {gross}: "
        f"{euro(wirkung[nach_wirkung])} mehr Deckungsbeitrag. {klein[0].upper()}{klein[1:]} "
        f"dagegen bringt nur {euro(wirkung[abweichler])} statt der rechnerischen "
        f"{euro(preise[abweichler] * abs(delta_max))} - ab +{grenze_gueltig:.0f} "
        f"Stunden begrenzt uns etwas anderes. Wer die ganze Schicht bezahlt, "
        f"bezahlt die zweite Haelfte umsonst.",
    ]

    if schuldige and noetig is not None:
        namen = ", ".join(f'"{s}"' for s in schuldige)
        satz = (f"Zum Wunsch des Vertriebs ({WUNSCH_MENGE:.0f} {WUNSCH_PRODUKT} statt "
                f"{plan[WUNSCH_PRODUKT]:.1f}): Er ist mit der heutigen Ausstattung "
                f"nicht erfuellbar - nicht aus Kostengruenden, sondern weil {namen} "
                f"ihn gemeinsam blockieren. Mit +{noetig} Stunden bei "
                f'"{engpass}" wird er moeglich')
        satz += (", und er kostet dann nichts: Der Plan waehlt diese Menge von "
                 "sich aus." if wunsch_gratis else ".")
        absaetze.append(satz)

    for absatz in absaetze:
        print()
        print(textwrap.fill(absatz, width=76))
    print("\n" + "=" * 78)
```

Ausgabe:

```
==============================================================================
  CONSTRAINT ATTRIBUTION: Was kostet uns welche Bedingung?
==============================================================================
Status: Optimization terminated successfully.
Deckungsbeitrag des Plans: 3,930.00 EUR

  Rahmen: 47.5  Gehaeuse: 30.0  Deckel: 97.5  Traeger: 25.0  Halter: 0.0

==============================================================================
  (1) Welche Bedingungen binden - und was kostet die naechste Einheit?
==============================================================================
Bedingung                  genutzt  Grenze  Schlupf Schattenpreis
------------------------------------------------------------------------------
Kapazitaet Montage           400.0     400      0.0        9.00  <-- bindend
Kapazitaet Lackieren         210.0     210      0.0        4.00  <-- bindend
Kapazitaet Pruefung          200.0     300    100.0        0.00
Liefervertrag Rahmen          47.5      40      7.5        0.00
Liefervertrag Gehaeuse        30.0      30      0.0        2.00  <-- bindend
Liefervertrag Deckel          97.5      20     77.5        0.00
```

```
Liefervertrag Traeger         25.0      25      0.0       18.00  <-- bindend
Marktgrenze Rahmen            47.5     120     72.5        0.00
Marktgrenze Gehaeuse          30.0      90     60.0        0.00
Marktgrenze Deckel            97.5     150     52.5        0.00
Marktgrenze Halter             0.0     200    200.0        0.00
Mindestumsatz              10800.0   10000    800.0        0.00
Sortimentsbreite             122.5     100     22.5        0.00
Lackierbudget Schicht 2       55.0      55      0.0        0.00  <-- bindend

  5 Bedingungen binden. Aber: Lackierbudget Schicht 2 bindet
  mit einem Schattenpreis von 0,00 EUR - sie beruehrt den Plan, ohne ihn
  zu verteuern. Bindend und kostenrelevant sind zwei verschiedene Dinge.

================================================================================
  (2) Was der Schattenpreis verschweigt
================================================================================
Der Schattenpreis gilt fuer die NAECHSTE Einheit. Ein Planer kauft aber
keine Einheit, sondern eine ganze Schicht. Beides gegenuebergestellt:

Stellschraube                Preis Schritt hochgerechnet  gemessen      Abw.
--------------------------------------------------------------------------------
Kapazitaet Montage            9.00     +50        450.00    450.00     +0.00
Kapazitaet Lackieren          4.00     +30        120.00     60.00    -60.00
Kapazitaet Pruefung           0.00     +50          0.00      0.00     +0.00
Lackierbudget Schicht 2       0.00     +10          0.00      0.00     +0.00
Liefervertrag Traeger        18.00     -10        180.00    180.00     +0.00
Liefervertrag Gehaeuse        2.00     -10         20.00     18.00     -2.00

  'Kapazitaet Lackieren': hochgerechnet 120.00 EUR, tatsaechlich 60.00 EUR.
  Der Preis von 4.00 EUR gilt nur bis +15 Stunden. Jede weitere Stunde
  dieser Sonderschicht bringt exakt nichts - der Engpass ist dann ein anderer.

  Und die Ranglisten drehen sich: Nach Schattenpreis fuehrt
  'Liefervertrag Traeger' (18.00 EUR je Einheit),
  nach tatsaechlicher Wirkung 'Kapazitaet Montage' (450.00 EUR).
  Der Schattenpreis sagt, was eine Einheit wert ist - nicht, wie viele
  Einheiten zu haben sind.

================================================================================
  (3) Was-waere-wenn: mindestens 55 Rahmen statt 47.5
================================================================================
Der Wunsch ist nicht teuer - er ist UNMOEGLICH.

Blockiert wird er von 3 Bedingungen gemeinsam. Faellt eine davon
weg, ist er erfuellbar; keine allein ist 'der' Grund:
   ohne 'Kapazitaet Lackieren': moeglich, Deckungsbeitrag 3,990.00 EUR
   ohne 'Liefervertrag Gehaeuse': moeglich, Deckungsbeitrag 3,948.00 EUR
   ohne 'Liefervertrag Traeger': moeglich, Deckungsbeitrag 4,320.00 EUR
```

```
   Mit +15 bei 'Kapazitaet Lackieren' wird der Wunsch erfuellbar:
   Deckungsbeitrag 3,990.00 EUR (+60.00 gegenueber heute).
   Und ohne den Wunsch ergaebe dieselbe Lockerung 3,990.00 EUR bei
   55 Rahmen - der Wunsch kostet also NICHTS.
   Er war nie das Problem. Der Engpass war es.

================================================================================
  (4) Und so steht es in der Vorlage fuer die Abteilungsleiterrunde
================================================================================

Der Plan bringt 3,930.00 EUR Deckungsbeitrag. Begrenzt wird er von 5
Bedingungen, von denen 4 tatsaechlich Geld kosten; "Lackierbudget Schicht 2"
beruehrt den Plan, ohne ihn zu verteuern.

Teuerste Bindung ist "Liefervertrag Traeger" mit 18.00 EUR je Einheit,
gefolgt von "Kapazitaet Montage" mit 9.00 EUR.

Die groesste einzelne Verbesserung bringt eine Sonderschicht Montage: 450.00
EUR mehr Deckungsbeitrag. Eine Sonderschicht Lackieren dagegen bringt nur
60.00 EUR statt der rechnerischen 120.00 EUR - ab +15 Stunden begrenzt uns
etwas anderes. Wer die ganze Schicht bezahlt, bezahlt die zweite Haelfte
umsonst.

Zum Wunsch des Vertriebs (55 Rahmen statt 47.5): Er ist mit der heutigen
Ausstattung nicht erfuellbar - nicht aus Kostengruenden, sondern weil
"Kapazitaet Lackieren", "Liefervertrag Gehaeuse", "Liefervertrag Traeger"
ihn gemeinsam blockieren. Mit +15 Stunden bei "Kapazitaet Lackieren" wird er
moeglich, und er kostet dann nichts: Der Plan waehlt diese Menge von sich
aus.

================================================================================
```

Drei Befunde, und keiner steht im Solverergebnis

Erstens: bindend heißt nicht teuer. Fünf Bedingungen binden, aber „Lackierbudget Schicht 2“ tut es mit einem Schattenpreis von **0,00 €**. Sie berührt den Plan, ohne ihn zu verteuern — wer sie nachverhandelt, gewinnt nichts. Das ist derselbe Unterschied, den `Erklaerbarkeit.py` oben an einer einzelnen Zuweisung zeigt, hier im Dualwert eines LP.

Zweitens: Der Schattenpreis ist eine Momentaufnahme. Er gilt für die **nächste** Einheit. Ein Planer kauft aber keine Einheit, sondern eine Schicht. Für die Lackiererei verspricht die Hochrechnung 4,00·30 = 120 €; gemessen kommen **60 €**. Der Preis von 4,00 € gilt exakt bis **+15 Stunden**, danach begrenzt etwas anderes — **die zweite Hälfte der Sonderschicht wäre bezahlt und wirkungslos.**

Und die Ranglisten drehen sich dabei um: Nach Schattenpreis führt der Trägervertrag mit 18,00 € je Stück, nach tatsächlicher Wirkung die Montage-Sonderschicht mit 450 €. Der Schattenpreis sagt, was **eine** Einheit wert ist — nicht, wie viele Einheiten zu haben sind. Für die Frage „welchen Hebel ziehen wir?“ ist das der entscheidende Unterschied.

Drittens: Ein Wunsch kann unmöglich statt teuer sein. Der Vertrieb hätte gern 55 Rahmen statt 47,5. Das Modell antwortet nicht mit einem Preis, sondern mit `INFEASIBLE` — und drei Bedingungen blockieren den Wunsch **gemeinsam**: Fällt eine davon weg, ist er erfüllbar, keine allein ist „der“ Grund. Genau die Struktur, die Anhang C für den unlösbaren Fall beschreibt, hier auf einen Wunsch angewandt.

Die Pointe steht am Ende: Mit +15 Stunden Lackierkapazität wird der Wunsch erfüllbar — und kostet dann **nichts**. Der Plan wählt diese Menge von sich aus, sobald der Engpass weg ist. Der Konflikt bestand nie zwischen Vertrieb und Produktion. Er bestand zwischen dem Vertrieb und der Lackiererei, und niemand wusste das.

> **Die drei Fragen, jetzt auf Modellebene** Die Faustregel oben stellt sie für eine einzelne Zuweisung. Constraint Attribution stellt dieselben drei für den **ganzen Plan**: 1. **Warum dieser Zielwert?** — welche Bedingungen binden, und welche davon kosten Geld. 2. **Warum nicht besser?** — was eine realistische Lockerung wirklich bringt, gemessen statt hochgerechnet. 3. **Was ist mit meinem Wunsch?** — erfüllbar und zu welchem Preis, oder unmöglich und warum.

Vom Zahlenwerk zur Vorlage

Alle drei Antworten stehen bis hierhin in Tabellen. Tabellen liest niemand in einer Abteilungsleiterrunde. Teil (4) des Programms baut aus **denselben** Zahlen einen Text — und das ist weniger trivial, als es aussieht:

- **Kein Satz enthält eine Zahl, die nicht vorher gerechnet wurde.** Auch nicht die Rangfolge, auch nicht die Namen der blockierenden Bedingungen. Sobald sich das Modell ändert, ändert sich der Bericht mit — er kann nicht leise falsch werden.
- **Der sprechende Name jeder Bedingung ist die Voraussetzung dafür.** `A_ub[7]` ergibt keinen Satz. Deshalb tragen die Bedingungen in beiden Programmen einen Namen, und deshalb ist das keine Kosmetik.

> **Was ein solcher Bericht nicht leistet** Er erklärt das **Modell**, nicht die **Wirklichkeit**. „Liefervertrag Träger kostet 18 € je Stück“ heißt: *in diesem Modell, mit diesen Deckungsbeiträgen*. Ist ein Deckungsbeitrag falsch geschätzt, ist der Bericht überzeugend **und** falsch — die gefährlichste Kombination. Ein generierter Text macht ein Modell nicht richtiger, nur schwerer anzuzweifeln. Deshalb gehört zu jedem solchen Bericht die Angabe, worauf er beruht.

22.5 Architektur einer produktionsreifen OR-Plattform

In einer professionellen Umgebung ist der Optimierer kein Skript, sondern ein **versionierter, zustandsloser Dienst**:

Schicht	Aufgabe	Typische Werkzeuge
Quellsysteme	ERP, Personalplanung, Marktdaten-Feeds	REST, Datenbanken, Broker-APIs
Datenqualität	Validierung, Plausibilität, unveränderlicher Snapshot	Pydantic, Pandera, Great Expectations

Schicht	Aufgabe	Typische Werkzeuge
Optimierungs-Worker	Modell bauen, lösen, Timeout, Fallback	Celery/Redis, OR-Tools, HiGHS, CVXPY
Disposition	Ergebnis prüfen, ändern, freigeben	Web-UI, Dashboard, Freigabeworkflow
Ausführung	Kalender-Sync, Benachrichtigung, Order-Routing	Kalender-APIs, Messaging, Broker
Betrieb	Monitoring, Audit-Trail, Alarme	Logging, Metriken, Versionierung

Fünf Prinzipien, die sich in der Praxis bewährt haben:

1. **Snapshot-Prinzip.** Jeder Optimierungslauf arbeitet auf einem unveränderlichen Datenschnappschuss mit eigener ID. Nur so ist ein Ergebnis später reproduzierbar — und im Streitfall belegbar.
2. **Zustandslosigkeit.** Der Worker hält keinen Zustand; Eingabe und Ausgabe sind Daten. Das macht Skalierung und Wiederholung trivial.
3. **Mensch in der Schleife.** Der Optimierer **schlägt vor**, ein Mensch **entscheidet** — zumindest in der Einführungsphase. Das baut Vertrauen auf und fängt Modellfehler ab.
4. **Fallback-Strategie.** Was passiert, wenn der Solver kein Ergebnis liefert? Antwort: letzter gültiger Plan, regelbasierte Notlösung, Alarm. Nie: nichts.
5. **Versionierung.** Modellversion, Parametersatz und Solver-Version gehören ins Protokoll jedes Laufs. Ohne das lässt sich nicht klären, warum das Ergebnis von letzter Woche anders aussah.

   Der häufigste Projektfehler Nicht das Modell scheitert, sondern die **Einführung**. Ein technisch überlegener Plan, den die Disponentin nicht versteht und dem sie nicht traut, wird umgangen — sie plant weiter in ihrer Tabelle. Rechnen Sie mindestens so viel Zeit für Erklärbarkeit, Schulung und schrittweise Einführung ein wie für das Modell selbst.

22.6 Der gemeinsame Unterbau: `or_kern.py`

Die fünf Prinzipien oben beschreiben eine Plattform. Der Weg dorthin beginnt aber viel kleiner — mit der Frage, welche Teile eines Optimierungsprogramms **immer dieselben** sind.

Sehen Sie sich die Programme dieses Buchs an: Jedes liest Daten ein, baut ein Modell, wertet einen Solverstatus aus und prüft das Ergebnis. Nur der mittlere Schritt ist wirklich problemspezifisch. Die anderen drei schreibt man in jedem Projekt neu — und macht dabei jedes Mal dieselben Fehler.

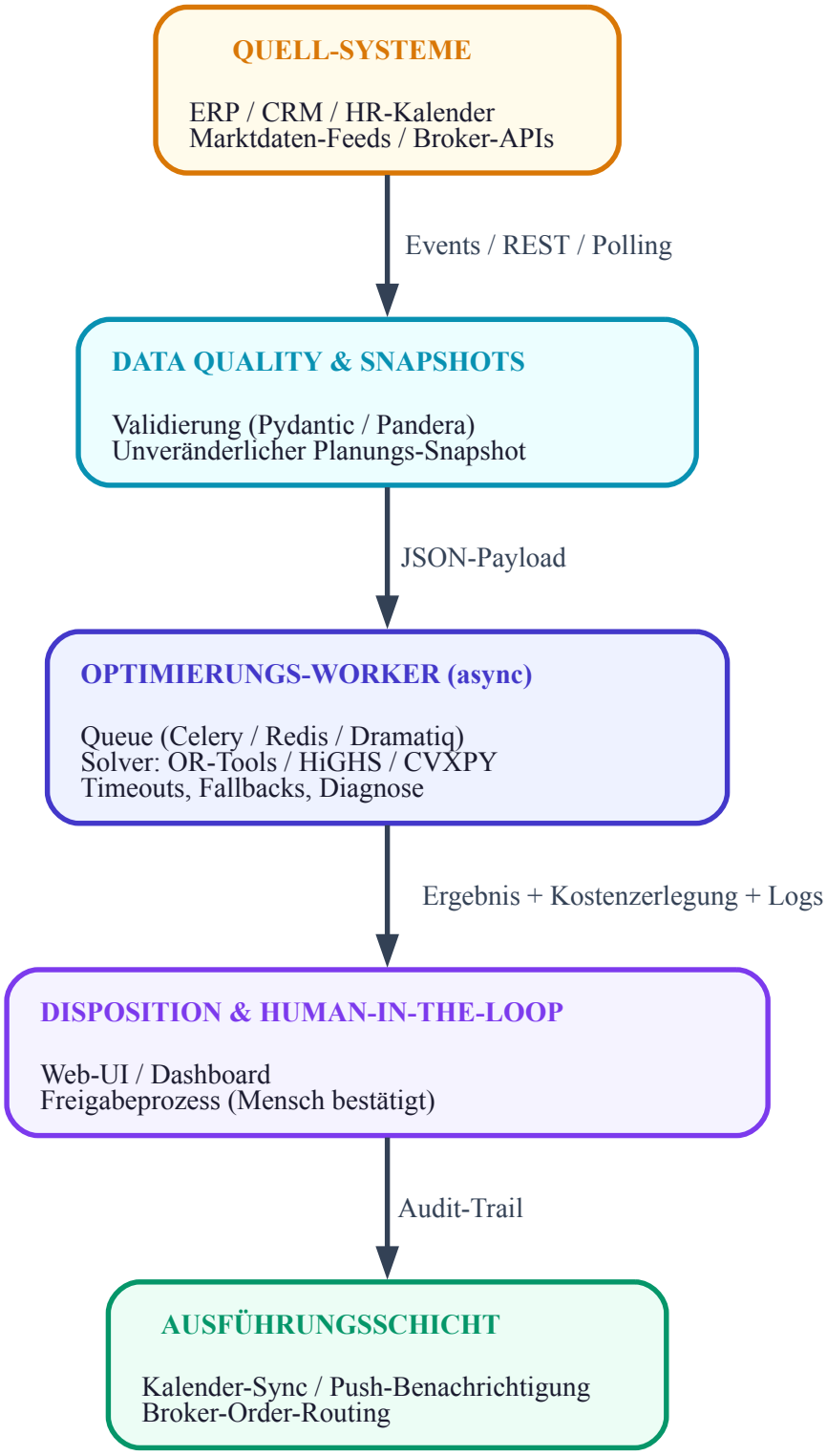

Abbildung 30: Abb. 22.2: Architektur einer produktionsreifen OR-Plattform

Die vier Schichten

Rohdaten (Excel, CSV, ERP) ⟶ Domänenmodell (geprüft, solverfrei) ⟶ Modellbauer (solverabhängig) ⟶ Lösung (DTO, solverfrei)

Schicht	Weiß nichts von …	Gewinn
Domänenmodell	Solvern, Matrizen, Variablen	Datenfehler schlagen beim **Einlesen** zu, wo man sie noch zuordnen kann
Modellbauer	Datenherkunft, Berichtsformat	Solverwechsel betrifft genau **eine** Datei
Lösung (DTO)	wie gerechnet wurde	Auswertung, Test und Bericht funktionieren für jeden Solver gleich
Abnahmeprüfung	Modell und Solver	prüft gegen die **Anforderung**, nicht gegen sich selbst

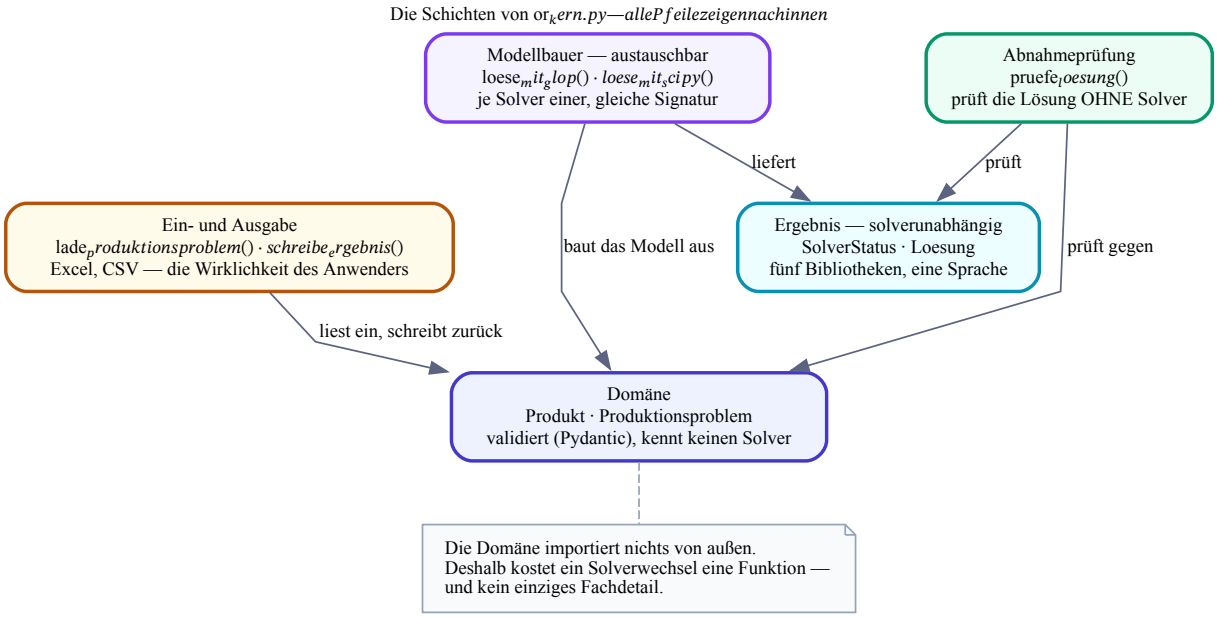

Abbildung 31: Abb. 22.3: Dieselben Schichten als Abhängigkeitsbild. Die Aussage steckt in der Pfeilrichtung: Alles zeigt nach innen auf die Domäne, und die Domäne zeigt auf nichts. Erzeugt von `bilder_04/erzeuge_architektur_diagramme.py`.

 Merksatz Die Trennung lohnt sich nicht wegen des Solverwechsels — den macht man selten. Sie lohnt sich, weil jede Schicht dadurch **einzeln prüfbar** wird. Das Domänenmodell testet man ohne Solver, die Abnahmeprüfung ohne Modell, den Bericht ohne Daten.

Ein Enum für fünf Bibliotheken

Der unscheinbarste, aber wirksamste Teil ist die Statusübersetzung. Dieselbe Aussage heißt in den fünf im Buch verwendeten Bibliotheken:

Bedeutung	pywraplp	CP-SAT	HiGHS	SciPy	CVXPY
beweisbar optimal	`Solver.OPTIMAL`	`cp_model.OPTIMAL`	`"Optimal"`	`status == 0`	`"optimal"`
zulässig, nicht bewiesen	`Solver.FEASIBLE`	`cp_model.FEASIBLE`	`"Time limit reached"`	—	`"optimal_inaccurate"`
es gibt keine Lösung	`Solver.INFEASIBLE`	`cp_model.INFEASIBLE`	`"Infeasible"`	`status == 2`	`"infeasible"`
Modell ist fehlerhaft	`Solver.ABNORMAL`	`cp_model.MODEL_INVALID`	`"Model error"`	`status == 4`	—

Fünf Schreibweisen für dieselben vier Aussagen. Wer direkt darauf prüft, bindet seinen gesamten Auswertungscode an eine Bibliothek — und muss ihn beim Wechsel überall anfassen.

Eine Besonderheit beim Import

`or_kern.py` importiert von sich aus **keine** Solverbibliothek. Die Statusübersetzer laden ihre Bibliothek erst beim Aufruf. Das ist kein Stilmittel, sondern notwendig: `ortools` und `highspy` vertragen sich nicht im selben Prozess (Abschnitt 3.5). Ein Modul, das beide importierte, wäre mit keinem von beiden benutzbar.

```
#!/usr/bin/env python3

# or_kern.py
"""
Kapitel Praxisfallen: Der gemeinsame Unterbau fuer produktionsreife OR-Modelle.

Alle Beispiele dieses Buchs loesen dieselben vier Aufgaben immer wieder:
Daten einlesen und pruefen, ein Modell bauen, den Solverstatus auswerten, die
Loesung gegen die Wirklichkeit kontrollieren. Dieses Modul zieht diese vier
Aufgaben aus den Einzelprogrammen heraus.

    Rohdaten (Excel/CSV)
        -> Domaenenmodell (Pydantic, geprueft)
            -> Modellbauer (solverabhaengig)
                -> Loesung (DTO, solverunabhaengig)
                    -> Abnahmepruefung

Der Sinn dieser Trennung: Die Domaenenschicht weiss nichts von Solvern, die
Loesungsschicht weiss nichts vom Modellaufbau. Wer den Solver wechselt, tauscht
genau EINEN Baustein aus - der Rest bleibt.
```

```
WICHTIG ZUM IMPORT: Dieses Modul importiert von sich aus KEINE Solver-
bibliothek. Der Grund steht im Kapitel Oekosystem: ortools und highspy vertragen sich
nicht im selben Prozess. Die Statusuebersetzer laden ihre Bibliothek erst,
wenn sie aufgerufen werden - so bleibt or_kern.py mit jedem Solver benutzbar.

Benoetigt: pydantic (v2), numpy; optional pandas und openpyxl fuer Excel.
"""

from __future__ import annotations

from enum import Enum
from typing import Annotated, Any, Sequence

import numpy as np
from pydantic import BaseModel, Field, model_validator

# Wiederverwendbare Feldtypen. Sie tragen die Pruefung im Typ, nicht im Code -
# damit gilt sie ueberall, wo der Typ verwendet wird.
PositiveZahl = Annotated[float, Field(gt=0)]
NichtNegativ = Annotated[float, Field(ge=0)]


# --- 1. Solverstatus: eine Sprache fuer fuenf Bibliotheken -------------------

class SolverStatus(str, Enum):
    """Was ein Solverlauf ergeben hat - unabhaengig davon, wer gerechnet hat.

    Die fuenf im Buch verwendeten Bibliotheken benennen dasselbe Ergebnis
    unterschiedlich: 'Optimal', 'OPTIMAL', 2, 'optimal'. Wer darauf direkt
    prueft, bindet seinen Auswertungscode an eine Bibliothek.
    """
    OPTIMAL = "optimal"              # beweisbar bestmoeglich
    ZULAESSIG = "zulaessig"          # brauchbare Loesung, Beweis fehlt
    UNZULAESSIG = "unzulaessig"      # es gibt keine Loesung (Aussage ueber das Modell)
    UNBESCHRAENKT = "unbeschraenkt"  # Ziel waechst ins Unendliche
    ZEITLIMIT = "zeitlimit"          # abgebrochen, nichts Brauchbares gefunden
    FEHLERHAFT = "fehlerhaft"        # das Modell ist kein gueltiges Modell
    UNBEKANNT = "unbekannt"          # alles Uebrige

    @property
    def brauchbar(self) -> bool:
        """Habe ich etwas in der Hand, das ich ausfuehren kann?"""
        return self in (SolverStatus.OPTIMAL, SolverStatus.ZULAESSIG)

    @property
    def modellfehler(self) -> bool:
        """Liegt die Ursache im Modell (und nicht in der Rechenzeit)?"""
        return self in (SolverStatus.UNZULAESSIG, SolverStatus.UNBESCHRAENKT,
```

```
                        SolverStatus.FEHLERHAFT)


def status_von_pywraplp(rohstatus: int) -> SolverStatus:
    """OR-Tools linear_solver (GLOP, SCIP, CBC)."""
    from ortools.linear_solver import pywraplp
    zuordnung = {
        pywraplp.Solver.OPTIMAL: SolverStatus.OPTIMAL,
        pywraplp.Solver.FEASIBLE: SolverStatus.ZULAESSIG,
        pywraplp.Solver.INFEASIBLE: SolverStatus.UNZULAESSIG,
        pywraplp.Solver.UNBOUNDED: SolverStatus.UNBESCHRAENKT,
        pywraplp.Solver.ABNORMAL: SolverStatus.FEHLERHAFT,
        pywraplp.Solver.NOT_SOLVED: SolverStatus.ZEITLIMIT,
    }
    return zuordnung.get(rohstatus, SolverStatus.UNBEKANNT)


def status_von_cpsat(rohstatus: int) -> SolverStatus:
    """OR-Tools CP-SAT (siehe Kapitel CP-SAT, alle fuenf Faelle)."""
    from ortools.sat.python import cp_model
    zuordnung = {
        cp_model.OPTIMAL: SolverStatus.OPTIMAL,
        cp_model.FEASIBLE: SolverStatus.ZULAESSIG,
        cp_model.INFEASIBLE: SolverStatus.UNZULAESSIG,
        cp_model.MODEL_INVALID: SolverStatus.FEHLERHAFT,
        cp_model.UNKNOWN: SolverStatus.ZEITLIMIT,
    }
    return zuordnung.get(rohstatus, SolverStatus.UNBEKANNT)


def status_von_highs(statustext: str) -> SolverStatus:
    """HiGHS ueber highspy - hier kommt der Status als Text."""
    zuordnung = {
        "Optimal": SolverStatus.OPTIMAL,
        "Time limit reached": SolverStatus.ZULAESSIG,   # meist mit Loesung
        "Solution limit reached": SolverStatus.ZULAESSIG,
        "Infeasible": SolverStatus.UNZULAESSIG,
        "Unbounded": SolverStatus.UNBESCHRAENKT,
        "Primal infeasible or unbounded": SolverStatus.UNZULAESSIG,
        "Model error": SolverStatus.FEHLERHAFT,
    }
    return zuordnung.get(statustext, SolverStatus.UNBEKANNT)


def status_von_scipy(ergebnis: Any) -> SolverStatus:
    """scipy.optimize.linprog und milp liefern ein Ergebnisobjekt mit .status."""
    zuordnung = {
        0: SolverStatus.OPTIMAL,
```

```
        1: SolverStatus.ZEITLIMIT,        # Iterations-/Zeitgrenze
        2: SolverStatus.UNZULAESSIG,
        3: SolverStatus.UNBESCHRAENKT,
        4: SolverStatus.FEHLERHAFT,       # numerische Schwierigkeiten
    }
    return zuordnung.get(int(ergebnis.status), SolverStatus.UNBEKANNT)


def status_von_cvxpy(statustext: str) -> SolverStatus:
    """CVXPY - Statuszeichenketten wie 'optimal' oder 'infeasible'."""
    zuordnung = {
        "optimal": SolverStatus.OPTIMAL,
        "optimal_inaccurate": SolverStatus.ZULAESSIG,
        "infeasible": SolverStatus.UNZULAESSIG,
        "infeasible_inaccurate": SolverStatus.UNZULAESSIG,
        "unbounded": SolverStatus.UNBESCHRAENKT,
        "unbounded_inaccurate": SolverStatus.UNBESCHRAENKT,
    }
    return zuordnung.get(statustext, SolverStatus.UNBEKANNT)


# --- 2. Die Loesung als solverunabhaengiges Datenobjekt ----------------------

class Loesung(BaseModel):
    """Das Ergebnis eines Solverlaufs, so wie es weiterverarbeitet wird.

    Bewusst OHNE Referenz auf Solver, Modell oder Variablen: Ein Bericht, ein
    Test oder eine Weiterverarbeitung soll nicht wissen muessen, womit
    gerechnet wurde.
    """
    status: SolverStatus
    werte: dict[str, float] = Field(default_factory=dict)
    zielwert: float | None = None
    schranke: float | None = None
    laufzeit: float | None = None
    # Schattenpreise je Nebenbedingung, sofern der Solver sie liefert
    # (nur bei kontinuierlichen Problemen, siehe Kapitel LP). Auch das ist
    # ein solverunabhaengiger Begriff und gehoert deshalb hierher.
    schattenpreise: dict[str, float] = Field(default_factory=dict)

    @property
    def gap(self) -> float | None:
        """Relativer Abstand zwischen gefundener Loesung und bewiesener Schranke.

        Das ist die Zahl fuer den Bericht: keine Schaetzung, sondern eine
        Zusage - schlechter als das kann die Loesung nicht sein.
        """
        if self.zielwert is None or self.schranke is None:
```

```
            return None
        nenner = max(abs(self.zielwert), 1e-9)
        return abs(self.zielwert - self.schranke) / nenner

    def als_bericht(self) -> str:
        """Eine Zeile fuers Betriebsprotokoll - siehe Betriebsueberwachung.py."""
        teile = [f"Status: {self.status.value}"]
        if self.zielwert is not None:
            teile.append(f"Zielwert: {self.zielwert:,.2f}")
        if self.gap is not None:
            teile.append(f"Gap: {self.gap:.2%}")
        if self.laufzeit is not None:
            teile.append(f"Zeit: {self.laufzeit:.2f}s")
        return " | ".join(teile)


# --- 3. Domaenenmodell: die Fakten, bevor ein Solver sie sieht ---------------

class Produkt(BaseModel):
    """Ein Produkt mit Deckungsbeitrag und Ressourcenverbrauch."""
    name: str = Field(min_length=1)
    deckungsbeitrag: float
    verbrauch: dict[str, NichtNegativ]

    model_config = {"frozen": True}      # Stammdaten aendern sich nicht im Lauf


class Produktionsproblem(BaseModel):
    """Produktionsprogrammplanung - das Modell aus Kapitel Einfuehrung.

    Die Pruefungen stehen hier und nicht im Solvercode. Damit gelten sie
    unabhaengig davon, mit welcher Bibliothek spaeter gerechnet wird - und sie
    schlagen beim EINLESEN zu, wo der Fehler noch zuzuordnen ist.
    """
    produkte: list[Produkt] = Field(min_length=1)
    kapazitaeten: dict[str, PositiveZahl]

    @model_validator(mode="after")
    def pruefe_ressourcen(self) -> "Produktionsproblem":
        namen = [p.name for p in self.produkte]
        if len(set(namen)) != len(namen):
            doppelt = sorted({n for n in namen if namen.count(n) > 1})
            raise ValueError(f"Produktnamen kommen mehrfach vor: {doppelt}")

        for produkt in self.produkte:
            unbekannt = set(produkt.verbrauch) - set(self.kapazitaeten)
            if unbekannt:
                raise ValueError(
```

```
                    f"Produkt '{produkt.name}' verbraucht Ressourcen ohne "
                    f"Kapazitaetsangabe: {sorted(unbekannt)}")
        return self

    @property
    def ressourcen(self) -> list[str]:
        """Feste Reihenfolge - siehe die Spaltenfalle in Kapitel Finanzdaten."""
        return sorted(self.kapazitaeten)

    def verbrauchsmatrix(self) -> np.ndarray:
        """Zeilen = Ressourcen, Spalten = Produkte (Matrixform, Kapitel Fundament)."""
        return np.array([[p.verbrauch.get(r, 0.0) for p in self.produkte]
                         for r in self.ressourcen])

    def kapazitaetsvektor(self) -> np.ndarray:
        return np.array([self.kapazitaeten[r] for r in self.ressourcen])

    def deckungsbeitragsvektor(self) -> np.ndarray:
        return np.array([p.deckungsbeitrag for p in self.produkte])


# --- 4. Abnahmepruefung: Loesung gegen Anforderung, ohne Solver --------------

def pruefe_loesung(problem: Produktionsproblem, loesung: Loesung,
                   ganzzahlig: Sequence[str] = (),
                   toleranz: float = 1e-6) -> list[str]:
    """Prueft eine Loesung gegen die Anforderungen - OHNE den Solver zu fragen.

    Diese Funktion darf ausdruecklich KEIN Solverobjekt und keine
    Modellvariable benutzen. Eine Pruefung aus denselben Bausteinen wie das
    Modell prueft das Modell gegen sich selbst und findet nichts (siehe die
    Denkfehler in den Kapiteln Graphen und MILP).

    Liefert eine Liste von Beanstandungen; leer heisst bestanden.
    """
    beanstandungen: list[str] = []

    if not loesung.status.brauchbar:
        return [f"Kein verwertbares Ergebnis (Status: {loesung.status.value})"]

    fehlend = [p.name for p in problem.produkte if p.name not in loesung.werte]
    if fehlend:
        return [f"Loesung enthaelt keine Werte fuer: {fehlend}"]

    mengen = np.array([loesung.werte[p.name] for p in problem.produkte])

    if (mengen < -toleranz).any():
        negativ = [p.name for p, m in zip(problem.produkte, mengen) if m < -toleranz]
```

```
        beanstandungen.append(f"negative Mengen bei: {negativ}")

    verbrauch = problem.verbrauchsmatrix() @ mengen
    for ressource, ist, grenze in zip(problem.ressourcen, verbrauch,
                                      problem.kapazitaetsvektor()):
        if ist > grenze + toleranz:
            beanstandungen.append(
                f"{ressource}: Verbrauch {ist:,.3f} ueber Kapazitaet {grenze:,.3f}")

    for name in ganzzahlig:
        wert = loesung.werte[name]
        if abs(wert - round(wert)) > toleranz:
            beanstandungen.append(f"'{name}' = {wert!r} ist nicht ganzzahlig")

    if loesung.zielwert is not None:
        nachgerechnet = float(problem.deckungsbeitragsvektor() @ mengen)
        if abs(nachgerechnet - loesung.zielwert) > toleranz * max(1.0, abs(nachgerechnet)):
            beanstandungen.append(
                f"Zielwert {loesung.zielwert:,.4f} passt nicht zu den Mengen "
                f"(nachgerechnet {nachgerechnet:,.4f})")

    return beanstandungen


# --- 5. Ein Modellbauer je Solver - austauschbar ----------------------------

def loese_mit_glop(problem: Produktionsproblem) -> Loesung:
    """Modellbauer fuer OR-Tools GLOP (kontinuierlich)."""
    import time
    from ortools.linear_solver import pywraplp

    solver = pywraplp.Solver.CreateSolver("GLOP")
    menge = {p.name: solver.NumVar(0, solver.infinity(), p.name)
             for p in problem.produkte}
    # Die Nebenbedingung wird gemerkt - ohne die Referenz gibt es spaeter
    # keinen Schattenpreis (siehe Kapitel LP).
    bedingung = {}
    for ressource in problem.ressourcen:
        bedingung[ressource] = solver.Add(
            sum(menge[p.name] * p.verbrauch.get(ressource, 0.0)
                for p in problem.produkte)
            <= problem.kapazitaeten[ressource], name=ressource)
    solver.Maximize(sum(menge[p.name] * p.deckungsbeitrag for p in problem.produkte))

    t0 = time.perf_counter()
    rohstatus = solver.Solve()
    laufzeit = time.perf_counter() - t0
```

```
    status = status_von_pywraplp(rohstatus)
    if not status.brauchbar:
        return Loesung(status=status, laufzeit=laufzeit)
    return Loesung(status=status,
                   werte={name: v.solution_value() for name, v in menge.items()},
                   zielwert=solver.Objective().Value(),
                   schranke=solver.Objective().Value(),
                   schattenpreise={r: b.dual_value() for r, b in bedingung.items()},
                   laufzeit=laufzeit)


def loese_mit_scipy(problem: Produktionsproblem) -> Loesung:
    """Derselbe Fall mit scipy.optimize.linprog - anderer Solver, gleiche Loesung.

    Beachten Sie, wie wenig hier steht: Die Daten kommen fertig geprueft aus
    dem Domaenenmodell, die Auswertung geht ans gemeinsame Loesung-Objekt.
    Genau das ist der Ertrag der Trennung.
    """
    import time
    from scipy.optimize import linprog

    t0 = time.perf_counter()
    ergebnis = linprog(c=-problem.deckungsbeitragsvektor(),      # linprog minimiert
                       A_ub=problem.verbrauchsmatrix(),
                       b_ub=problem.kapazitaetsvektor(),
                       bounds=(0, None), method="highs")
    laufzeit = time.perf_counter() - t0

    status = status_von_scipy(ergebnis)
    if not status.brauchbar:
        return Loesung(status=status, laufzeit=laufzeit)
    # linprog rechnet minimierend mit negierten Kosten - die Dualwerte
    # muessen entsprechend zurueckgedreht werden.
    return Loesung(status=status,
                   werte={p.name: float(x)
                          for p, x in zip(problem.produkte, ergebnis.x)},
                   zielwert=float(-ergebnis.fun),
                   schranke=float(-ergebnis.fun),
                   schattenpreise={r: float(-m) for r, m in
                                   zip(problem.ressourcen,
                                       ergebnis.ineqlin.marginals)},
                   laufzeit=laufzeit)


# --- 6. Ein- und Ausgabe: Tabellen sind die Wirklichkeit ---------------------
#
# In den meisten Betrieben liegen die Zahlen in einer Tabellenkalkulation
# (Kapitel Einfuehrung). Diese beiden Funktionen sind die einzige Stelle im
```

```
# Modul, die das weiss - alles andere arbeitet mit dem Domaenenmodell.
#
# pandas und openpyxl werden bewusst erst hier importiert: Wer or_kern nur
# fuer Statusauswertung und Pruefung benutzt, soll sie nicht installieren
# muessen.

STAMMSPALTEN = ("Produkt", "Deckungsbeitrag")


def lade_produktionsproblem(pfad: str, blatt_produkte: str = "Produkte",
                            blatt_kapazitaeten: str = "Kapazitaeten"
                            ) -> Produktionsproblem:
    """Liest eine Excel-Mappe und liefert ein geprueftes Domaenenmodell.

    Alle Spalten ausser 'Produkt' und 'Deckungsbeitrag' gelten als
    Verbrauchsspalten - eine neue Ressource ist damit eine neue SPALTE in der
    Mappe und erfordert keine Codeaenderung.

    Die eigentliche Pruefung passiert nicht hier, sondern im Konstruktor von
    Produktionsproblem. Diese Funktion uebersetzt nur Tabelle in Objekt; was
    ein gueltiges Problem ist, steht an genau einer Stelle.
    """
    import pandas as pd

    produkte_tabelle = pd.read_excel(pfad, sheet_name=blatt_produkte)
    kapazitaeten_tabelle = pd.read_excel(pfad, sheet_name=blatt_kapazitaeten)

    fehlende = [s for s in STAMMSPALTEN if s not in produkte_tabelle.columns]
    if fehlende:
        raise ValueError(f"Blatt '{blatt_produkte}': Spalten fehlen: {fehlende}")
    if produkte_tabelle.isna().any().any():
        zeilen = produkte_tabelle[produkte_tabelle.isna().any(axis=1)].index.tolist()
        raise ValueError(f"Blatt '{blatt_produkte}': leere Zellen in Zeile(n) {zeilen}")

    ressourcen = [s for s in produkte_tabelle.columns if s not in STAMMSPALTEN]
    produkte = [
        Produkt(name=str(zeile.Produkt),
                deckungsbeitrag=float(zeile.Deckungsbeitrag),
                verbrauch={r: float(getattr(zeile, r)) for r in ressourcen})
        for zeile in produkte_tabelle.itertuples()
    ]
    kapazitaeten = {str(z.Ressource): float(z.Verfuegbar)
                    for z in kapazitaeten_tabelle.itertuples()}

    return Produktionsproblem(produkte=produkte, kapazitaeten=kapazitaeten)


def schreibe_ergebnis(pfad: str, problem: Produktionsproblem,
```

```
                      loesung: Loesung) -> None:
    """Schreibt Plan und Kennzahlen als zwei Blaetter zurueck nach Excel.

    Die Fachabteilung bekommt das Ergebnis im Format, das sie ohnehin benutzt.
    Akzeptanz entscheidet ueber Projekterfolg (Kapitel Praxisfallen).
    """
    import pandas as pd

    if not loesung.status.brauchbar:
        raise ValueError(f"Nichts zu schreiben - Status: {loesung.status.value}")

    mengen = np.array([loesung.werte[p.name] for p in problem.produkte])
    plan = pd.DataFrame({
        "Produkt": [p.name for p in problem.produkte],
        "Menge": mengen,
        "Deckungsbeitrag": mengen * problem.deckungsbeitragsvektor(),
    })

    verbrauch = problem.verbrauchsmatrix() @ mengen
    kennzahlen: dict[str, float | str] = {
        "Status": loesung.status.value,
        "Gesamtdeckungsbeitrag": loesung.zielwert or 0.0,
    }
    for ressource, ist, grenze in zip(problem.ressourcen, verbrauch,
                                      problem.kapazitaetsvektor()):
        kennzahlen[f"{ressource}: Verbrauch"] = float(ist)
        kennzahlen[f"{ressource}: Auslastung %"] = float(100.0 * ist / grenze)
        if ressource in loesung.schattenpreise:
            kennzahlen[f"{ressource}: Schattenpreis"] = \
                loesung.schattenpreise[ressource]

    kennzahlen_tabelle = pd.DataFrame({
        "Kennzahl": list(kennzahlen),
        "Wert": [round(w, 2) if isinstance(w, float) else w
                 for w in kennzahlen.values()],
    })

    with pd.ExcelWriter(pfad, engine="openpyxl") as mappe:
        plan.round(2).to_excel(mappe, sheet_name="Plan", index=False)
        kennzahlen_tabelle.to_excel(mappe, sheet_name="Kennzahlen", index=False)


if __name__ == "__main__":
    # Die Schreinerei aus Kapitel Einfuehrung - jetzt als Domaenenmodell.
    schreinerei = Produktionsproblem(
        produkte=[
            Produkt(name="Tisch", deckungsbeitrag=240.0,
                    verbrauch={"Montagestunden": 3.0, "Plattenmaterial": 6.0}),
```

```
            Produkt(name="Stuhl", deckungsbeitrag=60.0,
                    verbrauch={"Montagestunden": 1.0, "Plattenmaterial": 1.0}),
        ],
        kapazitaeten={"Montagestunden": 150.0, "Plattenmaterial": 240.0},
    )

    print("=" * 78)
    print("  DERSELBE FALL, ZWEI SOLVER, EIN AUSWERTUNGSCODE")
    print("=" * 78)

    for name, bauer in [("OR-Tools GLOP", loese_mit_glop),
                        ("scipy / HiGHS", loese_mit_scipy)]:
        loesung = bauer(schreinerei)
        beanstandungen = pruefe_loesung(schreinerei, loesung)
        print(f"\n{name}")
        print(f"  {loesung.als_bericht()}")
        for produkt, menge in loesung.werte.items():
            print(f"    {produkt:<10} {menge:8.2f}")
        print(f"  Abnahmepruefung: "
              f"{'bestanden' if not beanstandungen else beanstandungen}")

    print("\n" + "=" * 78)
    print("  WAS DIE PRUEFUNGEN ABFANGEN")
    print("=" * 78)

    faelle = [
        ("Kapazitaet 0", dict(
            produkte=schreinerei.produkte,
            kapazitaeten={"Montagestunden": 0.0, "Plattenmaterial": 240.0})),
        ("Ressource ohne Kapazitaet", dict(
            produkte=[Produkt(name="Regal", deckungsbeitrag=130.0,
                              verbrauch={"Lackieren": 2.0})],
            kapazitaeten={"Montagestunden": 150.0})),
        ("doppelter Produktname", dict(
            produkte=[schreinerei.produkte[0], schreinerei.produkte[0]],
            kapazitaeten=schreinerei.kapazitaeten)),
    ]
    for beschreibung, daten in faelle:
        try:
            Produktionsproblem(**daten)
            print(f"  {beschreibung:<28} NICHT erkannt (!)")
        except Exception as fehler:
            meldung = str(fehler).splitlines()
            kern = next((z.strip() for z in meldung
                         if "Value error" in z or "greater than" in z), meldung[-1])
            print(f"  {beschreibung:<28} abgefangen: {kern[:44]}")

    print("\nAlle drei scheitern beim EINLESEN - nicht erst beim Loesen und")
```

```
    print("schon gar nicht erst im Bericht. Das ist der ganze Zweck der")
    print("Domaenenschicht.")
    print("=" * 78)
```

Erwartete Ausgabe:

```
==============================================================================
  DERSELBE FALL, ZWEI SOLVER, EIN AUSWERTUNGSCODE
==============================================================================

OR-Tools GLOP
  Status: optimal | Zielwert: 10,800.00 | Gap: 0.00% | Zeit: 0.00s
    Tisch         30.00
    Stuhl         60.00
  Abnahmepruefung: bestanden

scipy / HiGHS
  Status: optimal | Zielwert: 10,800.00 | Gap: 0.00% | Zeit: 0.00s
    Tisch         30.00
    Stuhl         60.00
  Abnahmepruefung: bestanden

==============================================================================
  WAS DIE PRUEFUNGEN ABFANGEN
==============================================================================
  Kapazitaet 0                abgefangen: Input should be greater than 0 [type=greater
  Ressource ohne Kapazitaet   abgefangen: Value error, Produkt 'Regal' verbraucht Ress
  doppelter Produktname       abgefangen: Value error, Produktnamen kommen mehrfach vo

Alle drei scheitern beim EINLESEN - nicht erst beim Loesen und
schon gar nicht erst im Bericht. Das ist der ganze Zweck der
Domaenenschicht.
==============================================================================
```

□ Code-Durchgang

Stelle	Was passiert	Warum so
`PositiveZahl = Annotated[float, Field(gt=0)]`	Prüfung im **Typ**, nicht im Code	Sie gilt damit überall, wo der Typ verwendet wird — man kann sie nicht vergessen.
`SolverStatus.brauchbar`	fragt „habe ich etwas in der Hand?“	Nicht „war der Solver erfolgreich?“. `ZULAESSIG` ist im Betrieb meist völlig ausreichend (Abschnitt 6.8).
`SolverStatus.modellfehler`	trennt Modell- von Zeitproblemen	Genau die Unterscheidung, die Abschnitt 7.8 als häufigste Verwechslung benennt.

Stelle	Was passiert	Warum so
`model_validator(mode="after")`	prüft Zusammenhänge **zwischen** Feldern	Einzelne Felder prüft Pydantic selbst; dass jedes Produkt nur bekannte Ressourcen verbraucht, muss man sagen.
`ressourcen` als sortierte Liste	feste Spaltenreihenfolge	Die Vertauschungsfalle aus Kapitel 18 wird hier strukturell unmöglich.
`loese_mit_glop` / `loese_mit_scipy`	zwei Modellbauer, ein Auswertungscode	Der Solverwechsel betrifft **eine** Funktion. Alles danach — Bericht, Prüfung, Test — bleibt unverändert.
die drei Fehlerfälle am Ende	Kapazität 0, unbekannte Ressource, doppelter Name	Alle drei scheitern beim **Einlesen**. Ohne Domänenschicht fielen sie erst beim Lösen auf — oder gar nicht.
`import pandas` **in** der Funktion	Excel-Schicht als optionale Abhängigkeit	Wer `or_kern` nur für Statusauswertung und Prüfung benutzt, soll pandas nicht installieren müssen.
`schattenpreise` im DTO	Dualwerte gehören zur Lösung, nicht zum Solver	Beide Modellbauer liefern hier identische Werte (40 / 20) — der Begriff ist solverunabhängig, die Abfrage nicht.

 Merksatz Ein Fehler, der beim Einlesen auffliegt, kostet Minuten. Derselbe Fehler, der erst im Bericht auffällt, kostet Tage — und derselbe Fehler, der **nie** auffällt, kostet am meisten. Validierung ist deshalb keine Fleißarbeit, sondern die günstigste Stelle im ganzen Ablauf.

22.7 Der Solverwechsel in der Praxis

Der vorige Abschnitt endet mit einer Behauptung: Ein Solverwechsel betreffe genau **einen** Baustein. Das ist die Art Satz, die in jedem Architekturvortrag vorkommt und die man deshalb nicht glauben sollte, bevor sie jemand vorgeführt hat.

Also führen wir sie vor — und zwar am unbequemsten Paar, das dieses Buch zu bieten hat: **CP-SAT gegen HiGHS**. Die beiden sind grundverschieden. CP-SAT ist ein Constraint-Programming-Solver, der ausschließlich ganzzahlig rechnet und Restriktionen als Ausdrücke entgegennimmt. HiGHS ist ein klassischer Branch-and-Bound-Löser, der eine Koeffizientenmatrix erwartet. Und sie lassen sich nicht einmal gemeinsam importieren (Kapitel 3).

Die Aufgabe ist eine kapazitierte Standortplanung: Aus sechs möglichen Lagern sind diejenigen zu eröffnen, mit denen zwölf Kunden am günstigsten beliefert werden. Jeder Kunde wird von genau einem Lager bedient, jedes eröffnete Lager kostet eine Fixgebühr, und die Kapazität reicht je Lager für 174 der insgesamt 387 Paletten — es müssen also mindestens drei Lager öffnen.

Der Aufbau folgt dem Muster aus `or_kern.py`:

Baustein	Solverabhängig?	Zeilen
`Standortproblem` (Pydantic-Domänenmodell)	nein	38
`baue_und_loese_mit_cpsat()`	**ja**	39
`baue_und_loese_mit_highs()`	**ja**	52
`pruefe_zuordnung()` (Abnahmeprüfung)	nein	36
Bericht und Vergleich	nein	Rest

Ausgetauscht wird eine Zeile: welche der beiden Funktionen aufgerufen wird. Prüfung und Bericht bekommen davon nichts mit, weil beide Funktionen dasselbe `Loesung`-Objekt liefern.

Warum zwei Prozesse?

`ortools` und `highspy` bringen beide eine eigene HiGHS-Kopie mit und kollidieren beim gemeinsamen Import (Kapitel 3). Das Programm startet sich deshalb für jeden Solver selbst noch einmal als Kindprozess und lässt sich das Ergebnis als **JSON** zurückgeben:

```
print(MODELLBAUER[sys.argv[1]](problem).model_dump_json())   # im Kindprozess
Loesung.model_validate_json(...)                            # im Hauptprozess
```

Das ist der zweite, weniger offensichtliche Ertrag des DTOs: Ein Lösungsobjekt ohne Solverbezug ist nicht nur eine Sprachregelung, sondern ein **Datenformat**. Es überlebt eine Prozessgrenze, passt in eine Warteschlange und lässt sich protokollieren. Ein `pywraplp.Solver`-Objekt kann das alles nicht.

```
#!/usr/bin/env python3

# Solverwechsel_CPSAT_HiGHS.py
"""
Kapitel Praxisfallen: Denselben Fall einmal mit CP-SAT und einmal mit HiGHS rechnen.

Die Trennung aus or_kern.py behauptet, ein Solverwechsel koste genau EINEN
Baustein. Dieses Programm loest diese Behauptung ein. Aufgabe ist eine
Standortplanung: Welche Lager oeffnen wir, und wer beliefert welchen Kunden?

    Standortproblem  (Pydantic, geprueft)   <- gemeinsam
        baue_und_loese_mit_cpsat()          <- der EINE Baustein
        baue_und_loese_mit_highs()          <- ... in zwei Ausfuehrungen
    Loesung          (DTO)                  <- gemeinsam
    pruefe_zuordnung()                      <- gemeinsam
    berichte()                              <- gemeinsam
```

```
Die beiden Modellbauer sind 39 und 52 Zeilen lang - sie sind der gesamte
solverabhaengige Teil des Programms. Alles andere wird zweimal benutzt und
einmal geschrieben.

WARUM ZWEI PROZESSE? ortools und highspy bringen beide eine eigene
HiGHS-Kopie mit und lassen sich auf vielen Systemen nicht gemeinsam
importieren (Kapitel Oekosystem). Das Hauptprogramm startet deshalb fuer jeden
Solver einen eigenen Python-Prozess und laesst sich die Loesung als JSON
zurueckgeben - das DTO ist nicht nur eine Sprachregelung, sondern ein
Datenformat, das eine Prozessgrenze ueberlebt.

Aufruf:
    python3 Solverwechsel_CPSAT_HiGHS.py            # beide, mit Vergleich
    python3 Solverwechsel_CPSAT_HiGHS.py cpsat      # nur der Kindprozess
    python3 Solverwechsel_CPSAT_HiGHS.py highs

Benoetigt: numpy, pydantic, ortools, highspy (jeweils im eigenen Prozess)
"""

from __future__ import annotations

import multiprocessing
import time
from concurrent.futures import ProcessPoolExecutor

import numpy as np
from pydantic import BaseModel, Field, model_validator

from or_kern import Loesung, SolverStatus, status_von_cpsat, status_von_highs

ZEITLIMIT = 30.0


# --- 1. Domaenenmodell: dieselben Fakten fuer beide Solver ------------------

class Standortproblem(BaseModel):
    """Kapazitierte Standortplanung mit Einzelbelieferung.

    Nach dem Muster von Produktionsproblem in or_kern.py: Die Pruefungen
    stehen im Konstruktor, nicht im Solvercode - sie gelten damit fuer beide
    Solver, und sie schlagen beim Einlesen zu.

    Alle Kosten sind ganzzahlig (Euro). Das ist keine Bequemlichkeit, sondern
    Voraussetzung: CP-SAT rechnet ausschliesslich ganzzahlig.
    """
    lager: list[str] = Field(min_length=1)
    kunden: list[str] = Field(min_length=1)
```

```
    fixkosten: list[int]        # je Lager, faellt bei Eroeffnung an
    kapazitaet: list[int]       # je Lager, in Paletten
    bedarf: list[int]           # je Kunde, in Paletten
    transport: list[list[int]]  # [Lager][Kunde], Kosten der Belieferung

    @model_validator(mode="after")
    def pruefe_masse(self) -> "Standortproblem":
        n, m = len(self.lager), len(self.kunden)
        if len(self.fixkosten) != n or len(self.kapazitaet) != n:
            raise ValueError(f"fixkosten/kapazitaet muessen {n} Eintraege haben")
        if len(self.bedarf) != m:
            raise ValueError(f"bedarf muss {m} Eintraege haben")
        if len(self.transport) != n or any(len(z) != m for z in self.transport):
            raise ValueError(f"transport muss {n} x {m} sein")
        if sum(self.kapazitaet) < sum(self.bedarf):
            raise ValueError(f"Gesamtkapazitaet {sum(self.kapazitaet)} deckt den "
                             f"Gesamtbedarf {sum(self.bedarf)} nicht")
        return self

    def schluessel(self, i: int, j: int) -> str:
        """Variablenname im Loesung-DTO - beide Modellbauer benutzen ihn."""
        return f"{self.lager[i]}->{self.kunden[j]}"


def beispielproblem(saat: int = 11) -> Standortproblem:
    """Sechs moegliche Lager, zwoelf Kunden - klein genug fuer beide Solver."""
    rng = np.random.default_rng(saat)
    lager = [f"Lager_{k}" for k in "ABCDEF"]
    kunden = [f"Kunde_{k:02d}" for k in range(1, 13)]
    bedarf = rng.integers(10, 60, len(kunden))
    return Standortproblem(
        lager=lager,
        kunden=kunden,
        fixkosten=rng.integers(3000, 9000, len(lager)).tolist(),
        kapazitaet=[int(bedarf.sum() * 0.45)] * len(lager),
        bedarf=bedarf.tolist(),
        transport=rng.integers(200, 1800, (len(lager), len(kunden))).tolist(),
    )


# --- 2. Der eine Baustein, der sich aendert: der Modellbauer -----------------

def baue_und_loese_mit_cpsat(problem: Standortproblem) -> Loesung:
    """CP-SAT: Bool-Variablen, ganzzahlige Koeffizienten, Minimize."""
    from ortools.sat.python import cp_model

    n, m = len(problem.lager), len(problem.kunden)
    modell = cp_model.CpModel()
```

```
    y = [modell.NewBoolVar(f"offen_{i}") for i in range(n)]
    x = {(i, j): modell.NewBoolVar(f"liefert_{i}_{j}")
         for i in range(n) for j in range(m)}

    for j in range(m):                                  # jeder Kunde genau einmal
        modell.AddExactlyOne(x[i, j] for i in range(n))
    for i in range(n):
        for j in range(m):                              # nur aus offenen Lagern
            modell.AddImplication(x[i, j], y[i])
        modell.Add(sum(problem.bedarf[j] * x[i, j] for j in range(m))
                   <= problem.kapazitaet[i] * y[i])     # Kapazitaet

    modell.Minimize(
        sum(problem.fixkosten[i] * y[i] for i in range(n))
        + sum(problem.transport[i][j] * x[i, j] for i in range(n) for j in range(m)))

    loeser = cp_model.CpSolver()
    loeser.parameters.max_time_in_seconds = ZEITLIMIT
    loeser.parameters.num_workers = 1
    loeser.parameters.random_seed = 1
    status = status_von_cpsat(loeser.Solve(modell))
    if not status.brauchbar:
        return Loesung(status=status, laufzeit=loeser.WallTime())

    return Loesung(
        status=status,
        werte={problem.schluessel(i, j): float(loeser.Value(x[i, j]))
               for i in range(n) for j in range(m)},
        zielwert=loeser.ObjectiveValue(),
        schranke=loeser.BestObjectiveBound(),
        laufzeit=loeser.WallTime())


def baue_und_loese_mit_highs(problem: Standortproblem) -> Loesung:
    """HiGHS: dieselben Restriktionen als Ungleichungszeilen einer Matrix."""
    import highspy

    n, m = len(problem.lager), len(problem.kunden)
    anzahl_x = n * m                                    # Spalten 0..n*m-1
    spalte_y = lambda i: anzahl_x + i                   # danach die y_i  # noqa: E731

    modell = highspy.Highs()
    modell.setOptionValue("output_flag", False)
    modell.setOptionValue("time_limit", ZEITLIMIT)
    modell.addVars(anzahl_x + n, np.zeros(anzahl_x + n), np.ones(anzahl_x + n))
    for spalte in range(anzahl_x + n):
        modell.changeColIntegrality(spalte, highspy.HighsVarType.kInteger)
    for i in range(n):
```

```
        modell.changeColCost(spalte_y(i), float(problem.fixkosten[i]))
        for j in range(m):
            modell.changeColCost(i * m + j, float(problem.transport[i][j]))

    for j in range(m):                                  # jeder Kunde genau einmal
        index = np.array([i * m + j for i in range(n)], dtype=np.int32)
        modell.addRow(1.0, 1.0, n, index, np.ones(n))
    for i in range(n):
        for j in range(m):                              # x_ij - y_i <= 0
            modell.addRow(-highspy.kHighsInf, 0.0, 2,
                          np.array([i * m + j, spalte_y(i)], dtype=np.int32),
                          np.array([1.0, -1.0]))
        index = np.array([i * m + j for j in range(m)] + [spalte_y(i)],
                         dtype=np.int32)                # Kapazitaet
        werte = np.concatenate([np.array(problem.bedarf, dtype=float),
                                [-float(problem.kapazitaet[i])]])
        modell.addRow(-highspy.kHighsInf, 0.0, m + 1, index, werte)

    t0 = time.perf_counter()
    modell.run()
    laufzeit = time.perf_counter() - t0

    status = status_von_highs(modell.modelStatusToString(modell.getModelStatus()))
    if not status.brauchbar:
        return Loesung(status=status, laufzeit=laufzeit)

    loesungswerte = modell.getSolution().col_value
    info = modell.getInfo()
    return Loesung(
        status=status,
        werte={problem.schluessel(i, j): float(loesungswerte[i * m + j])
               for i in range(n) for j in range(m)},
        zielwert=info.objective_function_value,
        schranke=info.mip_dual_bound,
        laufzeit=laufzeit)


MODELLBAUER = {"cpsat": baue_und_loese_mit_cpsat, "highs": baue_und_loese_mit_highs}


# --- 3. Alles Weitere ist wieder gemeinsam -----------------------------------

def pruefe_zuordnung(problem: Standortproblem, loesung: Loesung,
                     toleranz: float = 1e-6) -> list[str]:
    """Prueft die Loesung gegen die Anforderung - ohne Solver, ohne Modell."""
    if not loesung.status.brauchbar:
        return [f"kein verwertbares Ergebnis ({loesung.status.value})"]
```

```
    n, m = len(problem.lager), len(problem.kunden)
    zuordnung = np.array([[loesung.werte[problem.schluessel(i, j)]
                           for j in range(m)] for i in range(n)])
    beanstandungen: list[str] = []

    if (np.abs(zuordnung - np.round(zuordnung)) > toleranz).any():
        beanstandungen.append("Zuordnungen sind nicht 0/1")
    zuordnung = np.round(zuordnung)

    for j, kunde in enumerate(problem.kunden):
        if abs(zuordnung[:, j].sum() - 1.0) > toleranz:
            beanstandungen.append(f"{kunde} wird {zuordnung[:, j].sum():.0f}-mal beliefert")

    beliefert = zuordnung @ np.array(problem.bedarf, dtype=float)
    for i, lagername in enumerate(problem.lager):
        if beliefert[i] > problem.kapazitaet[i] + toleranz:
            beanstandungen.append(f"{lagername}: {beliefert[i]:.0f} Paletten ueber "
                                  f"Kapazitaet {problem.kapazitaet[i]}")

    if loesung.zielwert is not None:
        offen = beliefert > toleranz
        nachgerechnet = (np.array(problem.fixkosten, dtype=float) @ offen
                         + (np.array(problem.transport, dtype=float) * zuordnung).sum())
        if abs(nachgerechnet - loesung.zielwert) > 0.5:
            beanstandungen.append(f"Zielwert {loesung.zielwert:,.0f} passt nicht zur "
                                  f"Zuordnung (nachgerechnet {nachgerechnet:,.0f})")
    return beanstandungen


def geoeffnete_lager(problem: Standortproblem, loesung: Loesung) -> list[str]:
    m = len(problem.kunden)
    return [name for i, name in enumerate(problem.lager)
            if any(loesung.werte[problem.schluessel(i, j)] > 0.5 for j in range(m))]


def loese_in_eigenem_prozess(name: str, problem: Standortproblem) -> Loesung:
    """Laesst genau einen Modellbauer in einem frischen Prozess rechnen.

    'spawn' statt des Linux-Standards 'fork': Der Kindprozess startet mit
    einem leeren Interpreter und importiert nur den Solver, den SEIN
    Modellbauer braucht. max_tasks_per_child=1 sorgt dafuer, dass der Pool
    seinen Arbeiter nicht wiederverwendet - sonst saessen beim zweiten Aufruf
    wieder beide Bibliotheken im selben Prozess.

    Hin und zurueck wandert das Domaenenmodell bzw. das Loesungs-DTO. Beide
    kennen keinen Solver, sind also serialisierbar - genau dafuer sind sie da.
    """
    with ProcessPoolExecutor(
```

```
            max_workers=1,
            mp_context=multiprocessing.get_context("spawn"),
            max_tasks_per_child=1) as pool:
        return pool.submit(MODELLBAUER[name], problem).result(timeout=300)


if __name__ == "__main__":
    problem = beispielproblem()

    # --- Beide Solver anstossen und vergleichen --------------------------
    print("=" * 82)
    print("  DERSELBE FALL, ZWEI SOLVER - UND EIN AUSWERTUNGSCODE")
    print("=" * 82)
    print(f"Standortplanung: {len(problem.lager)} moegliche Lager, "
          f"{len(problem.kunden)} Kunden, {sum(problem.bedarf)} Paletten Bedarf.")
    print(f"Kapazitaet je Lager: {problem.kapazitaet[0]} Paletten "
          f"-> mindestens 3 Lager noetig.\n")

    loesungen: dict[str, Loesung] = {}
    for name, beschriftung in [("cpsat", "OR-Tools CP-SAT"),
                               ("highs", "HiGHS (highspy)")]:
        loesung = loesungen[name] = loese_in_eigenem_prozess(name, problem)
        beanstandungen = pruefe_zuordnung(problem, loesung)

        print(f"{beschriftung}")
        print(f"  {loesung.als_bericht()}")
        print(f"  eroeffnete Lager: {', '.join(geoeffnete_lager(problem, loesung))}")
        print(f"  Abnahmepruefung:  "
              f"{'bestanden' if not beanstandungen else beanstandungen}")

    # --- Was der Vergleich zeigt -----------------------------------------
    zielwerte = [loesung.zielwert for loesung in loesungen.values()]
    print("-" * 82)
    print(f"Zielwertdifferenz: {abs(zielwerte[0] - zielwerte[1]):.6f} EUR")

    gleich_belegt = all(
        round(loesungen["cpsat"].werte[s]) == round(loesungen["highs"].werte[s])
        for s in loesungen["cpsat"].werte)
    print(f"Identische Zuordnung: {'ja' if gleich_belegt else 'nein'}")

    assert abs(zielwerte[0] - zielwerte[1]) < 0.5, "Die Solver widersprechen sich!"
    assert all(l.status is SolverStatus.OPTIMAL for l in loesungen.values())

    print("\n" + "=" * 82)
    print("  WAS DER WECHSEL GEKOSTET HAT")
    print("=" * 82)
    print("Ausgetauscht wurde EINE Funktion. Domaenenmodell, Abnahmepruefung und")
    print("Bericht sind woertlich dieselben - sie sehen den Solver nie.")
```

```
    print()
    print("Nicht umsonst ist der Wechsel trotzdem:")
    print("  * CP-SAT rechnet ausschliesslich GANZZAHLIG. Alle Kosten sind hier")
    print("    deshalb int. Wer in Euro und Cent rechnet, skaliert vorher auf Cent -")
    print("    und muss das im Bericht wieder zuruecknehmen.")
    print("  * HiGHS braucht die Restriktionen als Matrixzeilen, CP-SAT nimmt sie")
    print("    als Ausdruecke. Das ist der Grund, warum der HiGHS-Modellbauer")
    print("    laenger ist, obwohl er dasselbe Modell beschreibt.")
    print("  * Beide Bibliotheken bringen eine eigene HiGHS-Kopie mit und lassen")
    print("    sich nicht gemeinsam importieren - daher die zwei Prozesse.")
    print()
    print("Der Ertrag: Beide beweisen denselben optimalen Zielwert, und die")
    print("Entscheidung zwischen ihnen ist eine Frage der Laufzeit geworden -")
    print("nicht eine Frage, wie viel Code man neu schreiben muss.")
    print()
    print("Verglichen wird deshalb der ZIELWERT, nicht der Plan: Gibt es mehrere")
    print("gleich teure Loesungen, darf jeder Solver eine andere davon liefern.")
    print("Hier stimmen sie zufaellig ueberein - darauf zu testen waere trotzdem")
    print("ein unzuverlaessiger Test (siehe JobShop_Intervalle.py).")
    print("=" * 82)
```

Erwartete Ausgabe (Laufzeiten hardwareabhängig):

```
==========================================================================================
  DERSELBE FALL, ZWEI SOLVER - UND EIN AUSWERTUNGSCODE
==========================================================================================
Standortplanung: 6 moegliche Lager, 12 Kunden, 387 Paletten Bedarf.
Kapazitaet je Lager: 174 Paletten -> mindestens 3 Lager noetig.

OR-Tools CP-SAT
  Status: optimal | Zielwert: 22,010.00 | Gap: 0.00% | Zeit: 0.07s
  eroeffnete Lager: Lager_A, Lager_B, Lager_D
  Abnahmepruefung:  bestanden
HiGHS (highspy)
  Status: optimal | Zielwert: 22,010.00 | Gap: 0.00% | Zeit: 0.02s
  eroeffnete Lager: Lager_A, Lager_B, Lager_D
  Abnahmepruefung:  bestanden
------------------------------------------------------------------------------------------
Zielwertdifferenz: 0.000000 EUR
Identische Zuordnung: ja

==========================================================================================
  WAS DER WECHSEL GEKOSTET HAT
==========================================================================================
Ausgetauscht wurde EINE Funktion. Domaenenmodell, Abnahmepruefung und
Bericht sind woertlich dieselben - sie sehen den Solver nie.

Nicht umsonst ist der Wechsel trotzdem:
```

```
  * CP-SAT rechnet ausschliesslich GANZZAHLIG. Alle Kosten sind hier
    deshalb int. Wer in Euro und Cent rechnet, skaliert vorher auf Cent -
    und muss das im Bericht wieder zuruecknehmen.
  * HiGHS braucht die Restriktionen als Matrixzeilen, CP-SAT nimmt sie
    als Ausdruecke. Das ist der Grund, warum der HiGHS-Modellbauer
    laenger ist, obwohl er dasselbe Modell beschreibt.
  * Beide Bibliotheken bringen eine eigene HiGHS-Kopie mit und lassen
    sich nicht gemeinsam importieren - daher die zwei Prozesse.

Der Ertrag: Beide beweisen denselben optimalen Zielwert, und die
Entscheidung zwischen ihnen ist eine Frage der Laufzeit geworden -
nicht eine Frage, wie viel Code man neu schreiben muss.

Verglichen wird deshalb der ZIELWERT, nicht der Plan: Gibt es mehrere
gleich teure Loesungen, darf jeder Solver eine andere davon liefern.
Hier stimmen sie zufaellig ueberein - darauf zu testen waere trotzdem
ein unzuverlaessiger Test (siehe JobShop_Intervalle.py).
================================================================================
```

Was dieser Vergleich belegt — und was er kostet

Belegt: Beide Solver beweisen denselben Zielwert von 22 010 €. Domänenmodell, Abnahmeprüfung und Bericht sind wörtlich dieselben; sie sehen den Solver nie. Die Entscheidung zwischen CP-SAT und HiGHS ist damit zu einer Laufzeitfrage geworden — man kann beide messen, statt sich vorher festlegen zu müssen.

Kostet: Der Wechsel ist nicht gratis, und es wäre unehrlich, das zu verschweigen.

- **CP-SAT rechnet ausschließlich ganzzahlig.** Alle Kosten im `Standortproblem` sind deshalb `int`. Wer in Euro und Cent rechnet, skaliert vor dem Modellaufbau auf Cent — und muss die Skalierung im Bericht zurücknehmen. Diese Umrechnung ist eine klassische Fehlerquelle; sie gehört an genau eine Stelle und in einen Test.
- **Der HiGHS-Modellbauer ist ein Drittel länger**, obwohl er dasselbe Modell beschreibt. Der Unterschied ist nicht Qualität, sondern Schnittstelle: `AddImplication(x, y)` gegen eine Matrixzeile mit den Koeffizienten $+1$ und -1.
- **Verglichen wird der Zielwert, nicht der Plan.** Gibt es mehrere gleich teure Lösungen, darf jeder Solver eine andere davon liefern — hier stimmen sie überein, aber darauf zu testen wäre unzuverlässig. Denselben Fehler zeigt `JobShop_Intervalle.py` in Kapitel 7.

Merksatz Die Frage „welcher Solver ist der beste?“ ist schlecht gestellt. Die gute Frage lautet: „Wie teuer ist es, den Solver zu wechseln?“ Wer sie mit *eine Funktion* beantworten kann, muss die erste Frage nie entscheiden — er misst.

22.8 Eine Checkliste vor dem Produktivgang

Arbeiten Sie diese Liste ab, bevor ein OR-System in Betrieb geht:

Modell - [ ] Alle Nebenbedingungen sind dokumentiert — mit fachlicher Begründung, nicht nur als Code. - [ ] Für jede harte Bedingung ist geprüft: Muss sie wirklich hart sein? - [ ] Es gibt eine Relaxationsstrategie für den Infeasible-Fall. - [ ] Big-M-Werte sind so klein wie möglich gewählt. - [ ] Die Zielfunktion ist in konsistenten Einheiten formuliert.

Daten - [ ] Spaltenreihenfolge und Datentypen werden nach jedem Import geprüft. - [ ] Fehlende Werte und Ausreißer haben eine definierte Behandlung. - [ ] Kovarianzmatrizen werden auf positive Definitheit geprüft. - [ ] Datenstände sind versioniert und reproduzierbar.

Validierung - [ ] Das Ergebnis wurde gegen eine unabhängige Rechnung geprüft (Handrechnung, zweiter Solver, Simulation). - [ ] Alle Nebenbedingungen werden nach dem Lösen per `assert` verifiziert. - [ ] Es gibt einen Vergleich gegen eine naive Referenzstrategie. - [ ] Bei Backtests: Lookahead-Selbsttest bestanden, Zahl der Versuche protokolliert.

Betrieb - [ ] Zeitlimit und akzeptierter Gap sind festgelegt und begründet. - [ ] Es gibt eine Fallback-Strategie und Alarmierung. - [ ] Jedes Ergebnis ist mit Snapshot-ID, Modellversion und Solver-Version protokolliert. - [ ] Die Ergebnisse sind erklärbar (Kostenzerlegung, Schattenpreise, Constraint-Trace).

22.9 Weiterführende Literatur und Roadmap

Grundlagen und lineare/ganzzahlige Optimierung * Bertsimas & Tsitsiklis: *Introduction to Linear Optimization* — Standardwerk zu Simplex, Dualität, Polyedertheorie. * Wolsey: *Integer Programming* — Branch-and-Cut, Schnittebenen, Formulierungsstärke.

Konvexe Optimierung * Boyd & Vandenberghe: *Convex Optimization* — Pflichtlektüre; frei verfügbar. Die theoretische Basis von CVXPY.

Constraint Programming und Scheduling * Google OR-Tools Dokumentation und das CP-SAT-Primer von Laurent Perron.

Quantitative Finanzmathematik * López de Prado: *Advances in Financial Machine Learning* — Backtest-Overfitting, Denoising, Deflated Sharpe Ratio. * Cornuéjols, Peña & Tütüncü: *Optimization Methods in Finance*.

Wie es weitergeht

Der nächste Schritt nach diesem Kurs ist selten ein weiteres Verfahren, sondern die **Verankerung im Betrieb**: reproduzierbare Datenpipelines, versionierte Modelle, Monitoring der Lösungsqualität, bewusster Umgang mit Zeitlimits.

Wenn Sie doch tiefer in die Methodik wollen, sind das die lohnendsten Richtungen:

Richtung	Wofür	Einstieg
Column Generation / Branch-and-Price	sehr große Zuschnitt- und Dienstplanprobleme	Kapitel 10, Wolsey
Benders-Zerlegung	zweistufige stochastische Modelle mit vielen Szenarien	Birge & Louveaux
Approximate Dynamic Programming / RL	hochdimensionale Zustandsräume	Powell: *Reinforcement Learning and Stochastic Optimization*
Konische Optimierung (SOCP, SDP)	robuste Portfolios, Ellipsoid-Unsicherheit	Boyd & Vandenberghe, Kapitel 5
Multi-Objective Optimization	echte Zielkonflikte ohne Gewichtung	Ehrgott: *Multicriteria Optimization*

22.10 Übungsaufgaben

Lösungen: Abschnitt A.22.

Aufgabe 22.1 □ — Hart oder weich, revisited. Nennen Sie für ein Vertretungsplanungssystem je zwei Bedingungen, die (a) zwingend hart bleiben müssen, (b) unbedingt weich sein sollten, (c) diskutabel sind. Begründen Sie.

Aufgabe 22.2 □ — Zeitlimit wählen. Für welche Szenarien setzen Sie welches Zeitlimit und welchen Gap? (a) Taxi-Disposition in Echtzeit, (b) Wochendienstplan, freitags erstellt, (c) Jahresproduktionsplanung, (d) Portfolio-Rebalancing monatlich.

Aufgabe 22.3 □□ — Relaxation einbauen. Nehmen Sie Ihr Modell aus der Aufgabe *Eigener Dienstplan* (Abschnitt 7.9) und machen Sie es `INFEASIBLE`-sicher: Schlupfvariablen für unbesetzte Schichten, gestaffelte Strafen. Testen Sie mit einem bewusst überlasteten Szenario.

Aufgabe 22.4 □□ — Erklärbarkeit implementieren. Erweitern Sie ein beliebiges Modell aus dem Kurs um einen `erklaere(loesung)`-Report, der die drei Fragen aus Abschnitt 22.3 beantwortet.

Aufgabe 22.5 □□ — Den Bericht in die Irre führen. `Constraint_Attribution.py` erzeugt seinen Bericht aus gerechneten Zahlen — richtig ist er deshalb noch nicht. (a) Setzen Sie den Deckungsbeitrag von *Deckel* von 9 € auf 20 €. Wie ändern sich Plan, Rangliste und Bericht? Welche Sätze bleiben wörtlich stehen, obwohl sie jetzt etwas anderes bedeuten? (b) Der Bericht nennt „Liefervertrag Träger" als teuerste Bindung. Formulieren Sie den Satz so um, dass ein Leser erkennt, **worauf** die Aussage beruht. (c) Welche Angabe müsste das Programm zusätzlich ausgeben, damit ein Empfänger die Belastbarkeit selbst einschätzen kann?

Aufgabe 22.6 □□□ — Gap gegen Laufzeit. Nehmen Sie das MILP-Portfolio aus Kapitel 6 und vergrößern Sie es auf 50 Anlagen. Messen Sie Laufzeit und Zielwert bei Gap-Vorgaben von 10 %, 5 %, 2 %, 1 %, 0,1 % und 0 %. Stellen Sie den Zusammenhang grafisch dar und leiten Sie eine Empfehlung ab.

Aufgabe 22.7 □□□ — Post-Mortem schreiben. Suchen Sie sich einen der im Buch besprochenen Fehler aus (z. B. die Vorzeichenfalle in Abschnitt 5.7, die ungültige Kovarianzmatrix in Kapitel 11, die Sektor-

Positionsindizes in Kapitel 19, die vermischten Einheiten in Abschnitt 20.6 oder die Rebalancing-Termine in Abschnitt 21.4) und schreiben Sie ein einseitiges Post-Mortem: Was war der Fehler? Warum fiel er nicht auf? Welche Prüfung hätte ihn gefunden? Welche Regel leiten Sie daraus für eigene Projekte ab?

22.11 Finde den Denkfehler

Falle 5 — die Laufzeitexplosion — zeigt sich im Betrieb selten als Absturz. Sie zeigt sich als **gar nichts**.

> **Finde den Denkfehler: Das Modell, das seit einem Jahr nicht mehr optimiert**
>
> Ein Standortplanungsmodell geht 2024 in Betrieb. Es läuft jede Nacht, hat drei Sekunden Zeitbudget und liefert zuverlässig die Kosten des Tagesplans. Das Betriebsteam überwacht Abstürze und Laufzeitspitzen — beides tritt nie auf.
>
> Zwei Jahre später ist das Unternehmen von 15 auf 80 Lager gewachsen. Am Code hat niemand etwas geändert; es gab ja keinen Anlass. Das Protokoll sieht so aus:
>
> | Zeitpunkt | Kosten |
> |---|---|
> | 2024-Q1 | 40 484 |
> | 2024-Q3 | 51 042 |
> | 2025-Q1 | 64 053 |
> | 2025-Q3 | 85 791 |
> | 2026-Q1 | 108 582 |
>
> Die Kosten steigen — das Unternehmen wächst schließlich. Alles unauffällig.
>
> Tatsächlich liefert das Modell seit **Anfang 2025** nicht mehr das Optimum, sondern eine Lösung, die zuletzt **17,2 %** davon entfernt ist.
>
> **Ihre Aufgabe:** (a) Warum ist die gleichbleibende Laufzeit hier kein Zeichen von Stabilität, sondern das Symptom? (b) Welche drei Größen fehlen im Protokoll? (c) Welche der drei warnt am **frühesten** — und warum ist gerade sie so wertvoll? (d) Das Team bemerkt den Fehler 2026. Wie hoch waren die Mehrkosten, und warum lässt sich das nachträglich nicht mehr genau beziffern?
>
> *Auflösung: Abschnitt A.22.*

```
#!/usr/bin/env python3

# Betriebsueberwachung.py
"""
Kapitel Praxisfallen: Das Modell, das still aufgehoert hat zu optimieren.

Ein Optimierungsmodell geht in Betrieb und laeuft jede Nacht. Es liefert
zuverlaessig eine Zahl, die Job-Dauer bleibt konstant, es gibt keine
Fehlermeldung. Alles sieht gut aus.
```

```
Waehrenddessen waechst das Unternehmen. Aus 15 Lagern werden 40, dann 80. Das
Zeitlimit von drei Sekunden - vor zwei Jahren grosszuegig bemessen - reicht
irgendwann nicht mehr. Der Solver bricht ab und liefert die beste Loesung, die
er bis dahin gefunden hat.

Das ist nicht falsch. Es ist sogar richtig so. Aber es ist etwas anderes als
das, was das Modell einmal geleistet hat - und niemand merkt es, weil im
Protokoll nur die Kosten stehen.

Dieses Programm zeigt die Verschlechterung und die Ueberwachung, die sie
sichtbar macht.

WICHTIG: ortools wird hier nicht importiert (Konflikt mit highspy, siehe
Kapitel Oekosystem).

Benoetigt: numpy, highspy
"""

from __future__ import annotations

import time
from dataclasses import dataclass

import numpy as np
import highspy

ZEITLIMIT = 3.0                 # das Budget, das der Nachtjob hat
GAP_WARNSCHWELLE = 0.02         # ab 2 % Gap wollen wir es wissen
ZEITAUSLASTUNG_WARNSCHWELLE = 0.9


@dataclass
class Laufprotokoll:
    """Was nach JEDEM Produktivlauf protokolliert gehoert.

    Die meisten Systeme schreiben nur 'kosten'. Genau deshalb faellt eine
    schleichende Verschlechterung ueber Monate nicht auf.
    """
    zeitpunkt: str
    lager: int
    kunden: int
    status: str
    kosten: float
    gap: float
    laufzeit: float
    knoten: int
```

```
    @property
    def warnungen(self) -> list[str]:
        meldungen = []
        if self.status != "Optimal":
            meldungen.append(f"nicht beweisbar optimal ({self.status})")
        if self.gap > GAP_WARNSCHWELLE:
            meldungen.append(f"Gap {self.gap:.1%} ueber Schwelle "
                             f"{GAP_WARNSCHWELLE:.0%}")
        if self.laufzeit > ZEITAUSLASTUNG_WARNSCHWELLE * ZEITLIMIT:
            meldungen.append(f"Zeitbudget zu {self.laufzeit / ZEITLIMIT:.0%} "
                             f"ausgeschoepft")
        return meldungen


def plane_netzwerk(n_lager: int, n_kunden: int, zeitlimit: float,
                   saat: int = 7) -> tuple[str, float, float, float, int]:
    """Standortplanung wie in Kapitel MILP, nur mit wachsender Groesse."""
    rng = np.random.default_rng(saat)
    fixkosten = rng.uniform(3000, 9000, n_lager)
    transport = rng.uniform(5, 60, (n_lager, n_kunden))
    bedarf = rng.uniform(10, 60, n_kunden)
    kapazitaet = np.full(n_lager, bedarf.sum() * 0.22)

    modell = highspy.Highs()
    modell.setOptionValue("output_flag", False)
    modell.setOptionValue("time_limit", zeitlimit)

    anzahl_x = n_lager * n_kunden
    unendlich = highspy.kHighsInf
    modell.addVars(anzahl_x, np.zeros(anzahl_x), np.full(anzahl_x, unendlich))
    modell.addVars(n_lager, np.zeros(n_lager), np.ones(n_lager))
    for i in range(n_lager):
        modell.changeColIntegrality(anzahl_x + i, highspy.HighsVarType.kInteger)
        modell.changeColCost(anzahl_x + i, fixkosten[i])
        for j in range(n_kunden):
            modell.changeColCost(i * n_kunden + j, transport[i, j])

    for j in range(n_kunden):
        index = np.array([i * n_kunden + j for i in range(n_lager)], dtype=np.int32)
        modell.addRow(bedarf[j], bedarf[j], len(index), index, np.ones(len(index)))
    for i in range(n_lager):
        index = np.array([i * n_kunden + j for j in range(n_kunden)] + [anzahl_x + i],
                         dtype=np.int32)
        werte = np.concatenate([np.ones(n_kunden), [-kapazitaet[i]]])
        modell.addRow(-unendlich, 0.0, len(index), index, werte)

    t0 = time.perf_counter()
    modell.run()
```

```
    laufzeit = time.perf_counter() - t0
    info = modell.getInfo()
    return (modell.modelStatusToString(modell.getModelStatus()),
            info.objective_function_value, info.mip_gap, laufzeit,
            info.mip_node_count)


if __name__ == "__main__":
    # So ist das Unternehmen ueber zwei Jahre gewachsen.
    entwicklung = [("2024-Q1", 15, 40), ("2024-Q3", 25, 70),
                   ("2025-Q1", 40, 110), ("2025-Q3", 60, 160),
                   ("2026-Q1", 80, 220)]

    print("=" * 84)
    print("  WAS DAS PROTOKOLL ZEIGT - UND WAS ES ZEIGEN SOLLTE")
    print("=" * 84)
    print(f"Derselbe Nachtjob, unveraendert, mit {ZEITLIMIT:.0f} Sekunden Zeitlimit.\n")

    protokolle = []
    for zeitpunkt, lager, kunden in entwicklung:
        status, kosten, gap, laufzeit, knoten = plane_netzwerk(
            lager, kunden, ZEITLIMIT)
        protokolle.append(Laufprotokoll(zeitpunkt, lager, kunden, status,
                                        kosten, gap, laufzeit, knoten))

    print("So sieht das ueblich gefuehrte Protokoll aus:\n")
    print(f"  {'Zeitpunkt':<10} {'Kosten':>12}")
    print("  " + "-" * 24)
    for p in protokolle:
        print(f"  {p.zeitpunkt:<10} {p.kosten:>12,.0f}")
    print("\n  -> Die Kosten steigen. Das Unternehmen waechst ja auch.")
    print("     Nichts an dieser Tabelle deutet auf ein Problem hin.")

    print("\n" + "-" * 84)
    print("So sieht ein vollstaendiges Protokoll aus:\n")
    print(f"  {'Zeitpunkt':<10} {'Groesse':>10} {'Status':>20} {'Kosten':>11} "
          f"{'Gap':>8} {'Zeit':>7}")
    print("  " + "-" * 80)
    for p in protokolle:
        groesse = f"{p.lager}x{p.kunden}"
        print(f"  {p.zeitpunkt:<10} {groesse:>10} {p.status:>20} "
              f"{p.kosten:>11,.0f} {p.gap * 100:7.2f} % {p.laufzeit:6.2f}s")

    print("\n" + "-" * 84)
    print("Und so sehen die Warnungen aus, die daraus folgen:\n")
    for p in protokolle:
        if p.warnungen:
            print(f"  {p.zeitpunkt}:  " + "; ".join(p.warnungen))
```

```
        else:
            print(f"  {p.zeitpunkt}:  in Ordnung")

    letzte = protokolle[-1]
    erste_warnung = next(p for p in protokolle if p.warnungen)

    print("\n" + "=" * 84)
    print("  WAS DA PASSIERT IST")
    print("=" * 84)
    print(f"Seit {erste_warnung.zeitpunkt} erreicht der Job das Zeitlimit und liefert")
    print("nicht mehr das Optimum, sondern die beste bis dahin gefundene Loesung.")
    print(f"Im letzten Lauf betraegt der Abstand zum Bestmoeglichen "
          f"{letzte.gap:.1%}.")
    print()
    print("Der Job ist nicht abgestuerzt. Er hat keine Fehlermeldung erzeugt.")
    print("Die Laufzeit ist sogar bemerkenswert STABIL geblieben - genau deshalb,")
    print("weil das Zeitlimit greift. Ein Ueberwachungssystem, das auf Abstuerze")
    print("und Laufzeitspitzen achtet, sieht hier nichts.")
    print()
    print("Drei Zahlen gehoeren deshalb in jedes Protokoll eines Optimierungsjobs:")
    print("  * der STATUS   - 'Optimal' oder etwas anderes?")
    print("  * der GAP      - wie weit ist die Loesung vom Bestmoeglichen entfernt?")
    print("  * die LAUFZEIT im Verhaeltnis zum Limit - wie nah am Anschlag?")
    print()
    print("Die dritte ist die frueheste Warnung: Sie steigt, lange bevor der Gap")
    print("sichtbar wird, und gibt Zeit zum Handeln.")
    print("=" * 84)
```

Erwartete Ausgabe (Zeiten hardwareabhängig):

```
====================================================================================
  WAS DAS PROTOKOLL ZEIGT - UND WAS ES ZEIGEN SOLLTE
====================================================================================
Derselbe Nachtjob, unveraendert, mit 3 Sekunden Zeitlimit.

So sieht das ueblich gefuehrte Protokoll aus:

  Zeitpunkt        Kosten
  ------------------------
  2024-Q1          40,484
  2024-Q3          51,042
  2025-Q1          64,053
  2025-Q3          85,791
  2026-Q1         108,582

  -> Die Kosten steigen. Das Unternehmen waechst ja auch.
     Nichts an dieser Tabelle deutet auf ein Problem hin.
```

```
--------------------------------------------------------------------------------
So sieht ein vollstaendiges Protokoll aus:

  Zeitpunkt     Groesse               Status      Kosten      Gap    Zeit
  ------------------------------------------------------------------------------
  2024-Q1         15x40              Optimal      40,484    0.00 %   0.57s
  2024-Q3         25x70              Optimal      51,042    0.00 %   1.50s
  2025-Q1        40x110   Time limit reached      64,053    8.41 %   3.00s
  2025-Q3        60x160   Time limit reached      85,791   14.99 %   3.01s
  2026-Q1        80x220   Time limit reached     108,582   17.16 %   3.01s

--------------------------------------------------------------------------------
Und so sehen die Warnungen aus, die daraus folgen:

  2024-Q1:  in Ordnung
  2024-Q3:  in Ordnung
  2025-Q1:  nicht beweisbar optimal (Time limit reached); Gap 8.4% ueber Schwelle 2%;
   ↪  Zeitbudget zu 100% ausgeschoepft
  2025-Q3:  nicht beweisbar optimal (Time limit reached); Gap 15.0% ueber Schwelle 2%;
   ↪  Zeitbudget zu 100% ausgeschoepft
  2026-Q1:  nicht beweisbar optimal (Time limit reached); Gap 17.2% ueber Schwelle 2%;
   ↪  Zeitbudget zu 100% ausgeschoepft

================================================================================
  WAS DA PASSIERT IST
================================================================================
Seit 2025-Q1 erreicht der Job das Zeitlimit und liefert
nicht mehr das Optimum, sondern die beste bis dahin gefundene Loesung.
Im letzten Lauf betraegt der Abstand zum Bestmoeglichen 17.2%.

Der Job ist nicht abgestuerzt. Er hat keine Fehlermeldung erzeugt.
Die Laufzeit ist sogar bemerkenswert STABIL geblieben - genau deshalb,
weil das Zeitlimit greift. Ein Ueberwachungssystem, das auf Abstuerze
und Laufzeitspitzen achtet, sieht hier nichts.

Drei Zahlen gehoeren deshalb in jedes Protokoll eines Optimierungsjobs:
  * der STATUS   - 'Optimal' oder etwas anderes?
  * der GAP      - wie weit ist die Loesung vom Bestmoeglichen entfernt?
  * die LAUFZEIT im Verhaeltnis zum Limit - wie nah am Anschlag?

Die dritte ist die frueheste Warnung: Sie steigt, lange bevor der Gap
sichtbar wird, und gibt Zeit zum Handeln.
================================================================================
```

 Merksatz Ein Optimierungsjob, der ein Zeitlimit hat, wird bei wachsenden Daten **nicht langsamer** — er wird **schlechter**. Genau deshalb greift die übliche Betriebsüberwachung nicht: Sie achtet auf Laufzeit und Abstürze, und beides bleibt unauffällig. Protokollieren Sie

Status, Gap und Zeitausschöpfung bei jedem Lauf, oder Sie erfahren nie, wann Ihr Modell aufgehört hat zu optimieren.

22.12 Micro-Quiz

Micro-Quiz 22: Drei Fragen zum Selbstcheck

Genau eine Antwort ist jeweils richtig. Auflösung in Anhang A.

1. Ihr Nachtjob läuft seit Monaten konstant 3,0 Sekunden bei einem Zeitlimit von 3 Sekunden. Was schließen Sie daraus? (a) Das System ist bemerkenswert stabil. (b) Das Zeitlimit greift jedes Mal. Der Solver bricht ab, statt fertig zu werden — die konstante Laufzeit ist das Symptom, nicht die Beruhigung. (c) Der Rechner ist ausgelastet und sollte aufgerüstet werden.

2. Eine Prüffunktion soll kontrollieren, ob ein Tourenplan die Fahrzeugkapazitäten einhält. Woher nimmt sie die geladenen Mengen? (a) Aus der Ladungsdimension des Routing-Modells — dort stehen sie ja bereits. (b) Sie summiert die Bedarfe der Kunden aus dem **ausgegebenen Tourenplan** auf. Alles andere fragt das Modell, ob es sich an sich selbst hält. (c) Aus der Zielfunktion, sofern die Kapazität dort bepreist ist.

3. Ein Modell meldet `INFEASIBLE`, nachdem eine neue Regel aufgenommen wurde. Was ist der produktivste erste Schritt? (a) Das Zeitlimit erhöhen. (b) Die neue Regel wieder entfernen und in Ruhe abwarten. (c) Die harten Regeln hierarchisch lockern — mit teuren Strafkosten statt Verboten —, damit der Solver benennt, **welche** Regel den Widerspruch erzeugt, statt nur zu sagen, dass es einen gibt.

22.13 Selbsttest

Antworten: Anhang A.

1. Wie verwandelt man ein unlösbares Modell in einen brauchbaren Notfallplan?
2. Welche drei Fragen muss ein erklärbares OR-System beantworten können?
3. Warum ist ein MIP-Gap von 2 % in der Praxis meist ausreichend?
4. Was ist das Snapshot-Prinzip und wozu dient es?
5. Was ist der häufigste Grund, aus dem OR-Projekte scheitern?
6. Eine Bedingung bindet, ihr Schattenpreis ist 0. Was heißt das — und was folgt daraus für eine Nachverhandlung?
7. Warum kann die Rangfolge nach Schattenpreis eine andere sein als die Rangfolge nach tatsächlichem Nutzen einer Lockerung?

22.14 Zusammenfassung

- **Infeasibility ist ein Entwurfsproblem, kein Solverproblem.** Wer alles hart formuliert, erhält irgendwann eine Fehlermeldung statt eines Plans.
- **Erklärbarkeit entscheidet über Akzeptanz.** Schattenpreise, Kostenzerlegung und Constraint-Traces sind keine Kür.
- **Bindend heißt nicht teuer.** Eine Bedingung kann den Plan berühren und trotzdem 0,00 € kosten. Nur die kostenrelevanten lohnen eine Nachverhandlung.
- **Der Schattenpreis gilt für die nächste Einheit, nicht für die nächste Schicht.** Hochgerechnet auf einen realistischen Schritt überschätzt er den Nutzen — im Beispiel um das Doppelte. Wer wissen will, was ein Hebel bringt, **misst** die Lockerung, statt zu multiplizieren; die beiden Ranglisten können sich dabei umdrehen.
- **Ein Wunsch kann unmöglich statt teuer sein** — und dann ist die richtige Antwort nicht ein Preis, sondern die Liste der Bedingungen, die ihn gemeinsam blockieren.
- **Ein generierter Bericht macht ein Modell nicht richtiger, nur schwerer anzuzweifeln.** Er erklärt das Modell, nicht die Wirklichkeit — deshalb gehört dazu, worauf er beruht.
- **Optimieren Sie nicht genauer als Ihre Daten.** Ein Gap von 2 % ist bei ±10 % Datenunsicherheit irrelevant.
- **Der Optimierer schlägt vor, der Mensch entscheidet** — jedenfalls anfangs.
- **Reproduzierbarkeit braucht Snapshots und Versionierung**, nicht guten Willen.
- **Projekte scheitern an der Einführung, nicht am Modell.**
- **Schreiben Sie die Abnahmeprüfung als eigenen Baustein** — sie darf keinen Solver und keine Modellvariable anfassen, sonst prüft sie das Modell gegen sich selbst. Sie belegt Zulässigkeit, nicht Optimalität; für Letztere braucht es ein zweites Verfahren oder eine bekannte Schranke.
- **Trennen Sie Domäne, Modellbau und Lösung** (`or_kern.py`). Der Gewinn liegt nicht im Solverwechsel, sondern darin, dass jede Schicht einzeln prüfbar wird — und dass Datenfehler beim **Einlesen** auffliegen statt im Bericht.
- **Protokollieren Sie Status, Gap und Zeitausschöpfung bei jedem Lauf.** Ein Job mit Zeitlimit wird bei wachsenden Daten nicht langsamer, sondern schlechter — und eine konstante Laufzeit am Limit ist deshalb ein Warnsignal, keine Beruhigung.

Kapitel 23: Testen, Messen, Ausliefern

Kapitel auf einen Blick

Worum geht es? Um die drei Fragen, die zwischen einem funktionierenden Modell und einem System stehen, auf das sich jemand verlässt: *Woher weiß ich, dass es stimmt? Woher weiß ich, dass es schnell genug ist? Wie kommt es zu den Leuten, die es brauchen?*

Voraussetzungen: Kapitel 22, insbesondere `or_kern.py` aus Abschnitt 22.6 — dieses Kapitel testet genau dieses Modul.

Danach können Sie: ein Optimierungsmodell testen, obwohl Sie die richtige Antwort nicht kennen; Ihre eigene Testsuite auf Lücken prüfen; einen Solververgleich aufsetzen, dem man

glauben kann; und das Modell als Dienst betreiben, ohne dass die erste große Anfrage ihn blockiert.

Zeitbedarf: ca. 5 Stunden.

Programme:
`test_or_kern.py`
`Mutationstest.py`
`Benchmark_Skalierung.py`
`Optimierungsdienst.py`

Notebook: `Notebooks_04/testing.ipynb`

23.1 In 5 Minuten gelöst

In 5 Minuten gelöst: drei Zeilen, die einen echten Fehler fangen

Die Schreinerei aus Kapitel 1, nur mit etwas knapperen Kapazitäten. Der Solver liefert einen Plan, halbe Tische gibt es nicht — also wird gerundet.

```
import numpy as np
from scipy.optimize import linprog

verbrauch_je_stueck = np.array([[3.0, 1.0],    # Montagestunden je Tisch/Stuhl
                                [6.0, 1.0]])   # Plattenmaterial je Tisch/Stuhl
kapazitaet = np.array([140.0, 232.0])
deckungsbeitrag = np.array([240.0, 60.0])

loesung = linprog(-deckungsbeitrag, A_ub=verbrauch_je_stueck, b_ub=kapazitaet,
                  bounds=(0, None), method="highs")
plan = np.round(loesung.x)                     # "Halbe Tische gibt es nicht"

verbrauch = verbrauch_je_stueck @ plan         # DER TEST
assert (verbrauch <= kapazitaet + 1e-6).all(), \
    f"Plan {plan} braucht {verbrauch}, verfuegbar sind {kapazitaet}"
print("Abnahme bestanden:", plan)
```

Ausgabe:

```
Traceback (most recent call last):
  File "schnellstart.py", line 14, in <module>
    assert (verbrauch <= kapazitaet + 1e-6).all(), \\
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: Plan [31. 48.] braucht [141. 234.], verfuegbar sind [140. 232.]
```

Und jetzt der Punkt. Der Solver hat richtig gerechnet: 30,67 Tische und 48 Stühle sind das Optimum des LP. Das Runden hat den Plan kaputt gemacht — 31 Tische brauchen eine Montagestunde mehr, als es gibt. Ohne die drei Zeilen wäre dieser Plan in die Fertigung gegangen, und dort hätte am Monatsende eine Stunde gefehlt, die niemand einer Zeile Python zugeordnet hätte.

Bemerkenswert ist, **was** der Test prüft. Er prüft nicht, ob 30,67 das Optimum ist — das könnte er gar nicht, dafür bräuchte er einen zweiten Solver. Er prüft die Eigenschaft, die jede brauchbare Lösung haben muss: *Sie hält die Kapazität ein.* Diese Verschiebung — **von der Frage nach dem richtigen Wert zur Frage nach den notwendigen Eigenschaften** — ist der Schlüssel zum ganzen Kapitel.

> **Merksatz** Ein Optimierungsmodell lässt sich fast nie gegen den richtigen Wert testen. Gegen seine Eigenschaften lässt es sich immer testen.

Warum funktioniert das? Weil die Prüfung aus einer **anderen Richtung** kommt als das Modell. Der Solver bekam die Matrix als Nebenbedingung, der Test benutzt sie als Nachrechnung des fertigen Plans. Dass beide dieselbe Matrix verwenden, ist dabei die Grenze des Verfahrens — darauf kommt Abschnitt 23.9 zurück.

23.2 Lernziele

Nach diesem Kapitel können Sie …

1. … die vier Testarten benennen, die bei Optimierungsmodellen tragen, wenn der Sollwert unbekannt ist.
2. … eine Testsuite so aufbauen, dass sie beide Modellbauer durchläuft statt nur einen.
3. … mit einem **Mutationstest** herausfinden, welche Fehler Ihre Suite durchgehen lässt.
4. … einen Solververgleich aufsetzen, der Aufbau und Lösen trennt, Zielwerte gegeneinander prüft und seine eigene Reichweite benennt.
5. … begründen, warum ein Optimierungsdienst zweistufig sein muss, und messen, ob Threads oder Prozesse die richtige Wahl sind.

23.3 Warum Optimierungsmodelle schwer zu testen sind

Bei einer gewöhnlichen Funktion schreibt man den erwarteten Wert hin:

```
assert steuer(50_000) == 12_345.60
```

Bei „der beste Produktionsplan für 400 Aufträge” gibt es niemanden, der die Antwort unabhängig ausrechnen könnte. Gäbe es ihn, bräuchte man den Solver nicht. Damit fällt die naheliegende Testform aus — und mit ihr die Illusion, man könne Optimierungscode wie Geschäftslogik testen.

Was bleibt, sind vier Ersatzformen:

Art	Was geprüft wird	Beispiel aus diesem Kapitel
Eigenschaften	Was jede zulässige Lösung erfüllen muss	Kapazitäten eingehalten, Zielwert passt zu den Mengen
Invarianten	Was das Ergebnis *nicht* ändern darf	Produktreihenfolge, Währungseinheit, ein Produkt mit Deckungsbeitrag 0

Art	Was geprüft wird	Beispiel aus diesem Kapitel
Regression	Eine kleine Instanz mit von Hand belegtem Optimum	die Schreinerei: 30 Tische, 60 Stühle, 10 800 €
Fehlerfälle	Dass Unsinn abgewiesen wird — und die Prüfung wirklich anschlägt	kaputte Lösungen an `pruefe_loesung()` verfüttern

Die Testart, die am häufigsten fehlt

Die vierte. Fast jede Suite prüft, dass gute Eingaben gute Ergebnisse liefern; kaum eine prüft, dass **schlechte Eingaben** auffliegen.

Das ist besonders bei Optimierungsmodellen fatal, weil die Abnahmeprüfung selbst Code ist, der falsch sein kann. Eine `pruefe_loesung()`, die versehentlich immer eine leere Liste zurückgibt, sieht in einer grünen Suite genauso aus wie eine, die funktioniert. Man merkt es erst, wenn sie etwas hätte finden sollen — also im Betrieb.

Die Tests unten füttern die Prüfung deshalb mit Lösungen, die absichtlich falsch sind: zu viel verbraucht, negative Mengen, ein Zielwert, der nicht zu den Mengen passt. Jeder dieser Tests verlangt, dass die Prüfung **anschlägt**.

23.4 Die Testsuite

```
#!/usr/bin/env python3

# test_or_kern.py
"""
Kapitel Testen: Eine Testsuite fuer ein Optimierungsmodell.

Das Grundproblem beim Testen von Optimierungsmodellen: Man kennt die richtige
Antwort nicht. Bei einer Funktion `steuer(brutto)` schreibt man den erwarteten
Wert hin. Bei "der beste Produktionsplan fuer 400 Auftraege" gibt es niemanden,
der ihn unabhaengig ausrechnen koennte - sonst braeuchte man den Solver nicht.

Deshalb testet man nicht den WERT, sondern vier andere Dinge:

  1. EIGENSCHAFTEN  Die Loesung haelt jede Nebenbedingung ein, und der
                    ausgewiesene Zielwert passt zu den Mengen.
  2. INVARIANTEN    Was das Ergebnis NICHT aendern darf: die Reihenfolge der
                    Produkte, die Waehrungseinheit, ein zusaetzliches Produkt
                    mit Deckungsbeitrag 0.
  3. REGRESSION     Eine kleine Instanz mit von Hand nachgerechnetem Optimum.
  4. FEHLERFAELLE   Unsinnige Eingaben werden abgewiesen, unloesbare Modelle
                    als unloesbar erkannt - und die Abnahmepruefung schlaegt
                    tatsaechlich an, wenn man ihr eine kaputte Loesung gibt.
```

```
Der vierte Punkt ist der wichtigste und wird am haeufigsten vergessen: Eine
Pruefung, die noch nie etwas gefunden hat, ist keine Pruefung, sondern eine
Vermutung. Die Tests unten fuettern `pruefe_loesung()` deshalb absichtlich mit
falschen Loesungen und verlangen, dass sie anschlaegt.

Aufruf:
    pytest test_or_kern.py -v
    python3 test_or_kern.py          # ruft pytest selbst auf

Benoetigt: pytest, numpy, pydantic, ortools, scipy (ueber or_kern)
"""

from __future__ import annotations

import pytest

from or_kern import (Loesung, Produkt, Produktionsproblem, SolverStatus,
                     loese_mit_glop, loese_mit_scipy, pruefe_loesung,
                     status_von_scipy)


# --- Die Instanz, gegen die getestet wird ---------------------------------

def schreinerei() -> Produktionsproblem:
    """Der Fall aus Kapitel Einfuehrung - klein genug fuer die Handrechnung.

    Optimum: 30 Tische, 60 Stuehle, Deckungsbeitrag 10.800 EUR.
    Beide Ressourcen sind voll ausgelastet (3*30 + 1*60 = 150,
    6*30 + 1*60 = 240).
    """
    return Produktionsproblem(
        produkte=[
            Produkt(name="Tisch", deckungsbeitrag=240.0,
                    verbrauch={"Montagestunden": 3.0, "Plattenmaterial": 6.0}),
            Produkt(name="Stuhl", deckungsbeitrag=60.0,
                    verbrauch={"Montagestunden": 1.0, "Plattenmaterial": 1.0}),
        ],
        kapazitaeten={"Montagestunden": 150.0, "Plattenmaterial": 240.0},
    )


# Beide Modellbauer durchlaufen JEDEN Test. Ein Test, der nur mit einem Solver
# laeuft, prueft die Bibliothek mit - nicht das Modell.
MODELLBAUER = [loese_mit_glop, loese_mit_scipy]
NAMEN = ["glop", "scipy"]
```

```
@pytest.fixture(params=MODELLBAUER, ids=NAMEN)
def bauer(request):
    return request.param


# --- 1. Eigenschaften -----------------------------------------------------

def test_loesung_haelt_alle_nebenbedingungen_ein(bauer):
    """Die Abnahmepruefung darf nichts zu beanstanden haben."""
    problem = schreinerei()
    loesung = bauer(problem)
    assert loesung.status is SolverStatus.OPTIMAL
    assert pruefe_loesung(problem, loesung) == []


def test_zielwert_passt_zu_den_mengen(bauer):
    """Der ausgewiesene Zielwert wird unabhaengig nachgerechnet.

    Klingt trivial, ist es nicht: Wer die Zielfunktion im Modell anders
    zusammensetzt als im Bericht (etwa Gebuehren einmal abgezogen, einmal
    nicht), merkt es sonst nie.
    """
    problem = schreinerei()
    loesung = bauer(problem)
    nachgerechnet = sum(p.deckungsbeitrag * loesung.werte[p.name]
                        for p in problem.produkte)
    assert loesung.zielwert == pytest.approx(nachgerechnet, abs=1e-6)


def test_beide_solver_liefern_dasselbe():
    """Der Kern der Architektur aus Kapitel Praxisfallen, als Test."""
    problem = schreinerei()
    glop, scipy_ = loese_mit_glop(problem), loese_mit_scipy(problem)
    assert glop.zielwert == pytest.approx(scipy_.zielwert, abs=1e-6)
    for ressource in problem.ressourcen:
        assert glop.schattenpreise[ressource] == pytest.approx(
            scipy_.schattenpreise[ressource], abs=1e-6)


# --- 2. Invarianten -------------------------------------------------------

def test_produktreihenfolge_aendert_nichts(bauer):
    """Dieselben Daten in anderer Zeilenreihenfolge - gleiches Ergebnis.

    Das faengt die Spaltenvertauschungsfalle aus Kapitel Finanzdaten ab: ein
    Modell, das Positionen statt Namen benutzt, faellt hier durch.
    """
    problem = schreinerei()
```

```
    gedreht = Produktionsproblem(produkte=list(reversed(problem.produkte)),
                                 kapazitaeten=problem.kapazitaeten)
    assert bauer(problem).zielwert == pytest.approx(bauer(gedreht).zielwert)


def test_waehrungseinheit_skaliert_linear(bauer):
    """Deckungsbeitraege in Cent statt Euro: Zielwert mal 100, Mengen gleich.

    Der Test prueft nicht die Mathematik - die ist offensichtlich -, sondern
    die NUMERIK. Wer schlecht skalierte Modelle baut (Kapitel Fundament),
    bekommt hier Abweichungen weit ueber der Toleranz.
    """
    problem = schreinerei()
    in_cent = Produktionsproblem(
        produkte=[Produkt(name=p.name, deckungsbeitrag=p.deckungsbeitrag * 100,
                          verbrauch=p.verbrauch) for p in problem.produkte],
        kapazitaeten=problem.kapazitaeten)
    basis, skaliert = bauer(problem), bauer(in_cent)
    assert skaliert.zielwert == pytest.approx(basis.zielwert * 100, rel=1e-9)
    for p in problem.produkte:
        assert skaliert.werte[p.name] == pytest.approx(basis.werte[p.name],
                                                       abs=1e-6)


def test_produkt_ohne_deckungsbeitrag_aendert_das_optimum_nicht(bauer):
    """Ein Produkt, das nichts einbringt, darf den Zielwert nicht heben."""
    problem = schreinerei()
    mit_nullprodukt = Produktionsproblem(
        produkte=problem.produkte + [
            Produkt(name="Muster", deckungsbeitrag=0.0,
                    verbrauch={"Montagestunden": 1.0, "Plattenmaterial": 1.0})],
        kapazitaeten=problem.kapazitaeten)
    assert bauer(mit_nullprodukt).zielwert <= bauer(problem).zielwert + 1e-6


# --- 3. Regression ----------------------------------------------------------

def test_bekanntes_optimum(bauer):
    """Von Hand nachgerechnet (Kapitel Einfuehrung, Handrechnung).

    Das ist der einzige Test, der einen ZAHLENWERT festschreibt - und er darf
    es, weil dieser Wert von Hand belegt ist. Fuer grosse Instanzen gibt es
    diesen Test nicht; dort tragen die Eigenschaften und Invarianten.
    """
    loesung = bauer(schreinerei())
    assert loesung.zielwert == pytest.approx(10_800.0, abs=1e-6)
    assert loesung.werte["Tisch"] == pytest.approx(30.0, abs=1e-6)
    assert loesung.werte["Stuhl"] == pytest.approx(60.0, abs=1e-6)
```

```
def test_schattenpreise_bekannt(bauer):
    """40 EUR je Montagestunde, 20 EUR je Einheit Plattenmaterial."""
    loesung = bauer(schreinerei())
    assert loesung.schattenpreise["Montagestunden"] == pytest.approx(40.0, abs=1e-6)
    assert loesung.schattenpreise["Plattenmaterial"] == pytest.approx(20.0, abs=1e-6)


# --- 4. Fehlerfaelle: schlaegt die Pruefung ueberhaupt an? ----------------

def test_pruefung_findet_kapazitaetsverletzung():
    """Eine Loesung, die zu viel verbraucht, MUSS beanstandet werden."""
    problem = schreinerei()
    kaputt = Loesung(status=SolverStatus.OPTIMAL,
                     werte={"Tisch": 50.0, "Stuhl": 60.0},   # 3*50+60 = 210 > 150
                     zielwert=50 * 240.0 + 60 * 60.0)
    beanstandungen = pruefe_loesung(problem, kaputt)
    assert any("Montagestunden" in b for b in beanstandungen)


def test_pruefung_findet_falschen_zielwert():
    """Mengen und Zielwert passen nicht zusammen - der haeufigste stille Fehler."""
    problem = schreinerei()
    kaputt = Loesung(status=SolverStatus.OPTIMAL,
                     werte={"Tisch": 30.0, "Stuhl": 60.0},
                     zielwert=99_999.0)
    assert any("Zielwert" in b for b in pruefe_loesung(problem, kaputt))


def test_pruefung_findet_negative_mengen():
    problem = schreinerei()
    kaputt = Loesung(status=SolverStatus.OPTIMAL,
                     werte={"Tisch": -5.0, "Stuhl": 60.0},
                     zielwert=-5 * 240.0 + 60 * 60.0)
    assert any("negativ" in b for b in pruefe_loesung(problem, kaputt))


def test_pruefung_findet_gebrochene_ganzzahligkeit():
    """Die Falle aus Kapitel MILP: 0,99999998 ist nicht 1."""
    problem = schreinerei()
    fast_ganz = Loesung(status=SolverStatus.OPTIMAL,
                        werte={"Tisch": 29.4, "Stuhl": 61.8},
                        zielwert=29.4 * 240.0 + 61.8 * 60.0)
    beanstandungen = pruefe_loesung(problem, fast_ganz, ganzzahlig=["Tisch"])
    assert any("ganzzahlig" in b for b in beanstandungen)
```

```
def test_pruefung_verweigert_unbrauchbaren_status():
    """Ohne verwertbares Ergebnis wird gar nicht erst gerechnet."""
    problem = schreinerei()
    ohne = Loesung(status=SolverStatus.UNZULAESSIG)
    beanstandungen = pruefe_loesung(problem, ohne)
    assert len(beanstandungen) == 1 and "unzulaessig" in beanstandungen[0]


@pytest.mark.parametrize("daten, stichwort", [
    (dict(produkte=[Produkt(name="Tisch", deckungsbeitrag=240.0,
                            verbrauch={"Montagestunden": 3.0})],
          kapazitaeten={"Montagestunden": 0.0}), "greater than"),
    (dict(produkte=[Produkt(name="Regal", deckungsbeitrag=130.0,
                            verbrauch={"Lackieren": 2.0})],
          kapazitaeten={"Montagestunden": 150.0}), "Kapazitaetsangabe"),
    (dict(produkte=[], kapazitaeten={"Montagestunden": 150.0}), "at least 1"),
])
def test_unsinnige_eingaben_werden_beim_einlesen_abgewiesen(daten, stichwort):
    """Drei Fehler, die NICHT erst beim Loesen auffallen duerfen."""
    with pytest.raises(Exception) as fehler:
        Produktionsproblem(**daten)
    assert stichwort in str(fehler.value)


def test_doppelter_produktname_wird_abgewiesen():
    produkt = Produkt(name="Tisch", deckungsbeitrag=240.0,
                      verbrauch={"Montagestunden": 3.0})
    with pytest.raises(Exception) as fehler:
        Produktionsproblem(produkte=[produkt, produkt],
                           kapazitaeten={"Montagestunden": 150.0})
    assert "mehrfach" in str(fehler.value)


@pytest.mark.parametrize("rohstatus, erwartet", [
    (0, SolverStatus.OPTIMAL),
    (1, SolverStatus.ZEITLIMIT),
    (2, SolverStatus.UNZULAESSIG),
    (3, SolverStatus.UNBESCHRAENKT),
    (4, SolverStatus.FEHLERHAFT),
    (99, SolverStatus.UNBEKANNT),
])
def test_statusuebersetzung_scipy(rohstatus, erwartet):
    """Die Uebersetzertabelle selbst - inklusive des unbekannten Falls."""
    class Ergebnis:
        status = rohstatus
    assert status_von_scipy(Ergebnis()) is erwartet
```

```
def test_status_eigenschaften_sind_konsistent():
    """brauchbar und modellfehler duerfen sich nie ueberschneiden."""
    for status in SolverStatus:
        assert not (status.brauchbar and status.modellfehler)


# Die beiden folgenden Tests gab es zuerst nicht. Sie sind entstanden, weil
# Mutationstest.py zwei eingebaute Fehler UEBERLEBEN liess - siehe den
# Abschnitt "Wer testet die Tests?".

def test_nur_optimal_und_zulaessig_gelten_als_brauchbar():
    """ZEITLIMIT ist NICHT brauchbar - der Solver hat nichts gefunden.

    Der Unterschied entscheidet, ob ein Nachtjob einen Plan ausliefert oder
    Alarm schlaegt. Ein aufgeweichtes 'brauchbar' faellt sonst nirgends auf.
    """
    assert SolverStatus.OPTIMAL.brauchbar
    assert SolverStatus.ZULAESSIG.brauchbar
    for status in (SolverStatus.ZEITLIMIT, SolverStatus.UNZULAESSIG,
                   SolverStatus.UNBESCHRAENKT, SolverStatus.FEHLERHAFT,
                   SolverStatus.UNBEKANNT):
        assert not status.brauchbar, f"{status.value} darf nicht brauchbar sein"


def test_ressourcenreihenfolge_haengt_nicht_an_der_eingabe():
    """Zwei Mappings mit gleichem Inhalt, andere Einfuegereihenfolge.

    Die Verbrauchsmatrix muss zeilenweise identisch sein. Ohne diese
    Zusicherung passt die Matrix irgendwann nicht mehr zum
    Kapazitaetsvektor - die Vertauschungsfalle aus Kapitel Finanzdaten,
    nur eine Ebene tiefer.
    """
    produkte = schreinerei().produkte
    vorwaerts = Produktionsproblem(
        produkte=produkte,
        kapazitaeten={"Montagestunden": 150.0, "Plattenmaterial": 240.0})
    rueckwaerts = Produktionsproblem(
        produkte=produkte,
        kapazitaeten={"Plattenmaterial": 240.0, "Montagestunden": 150.0})
    assert vorwaerts.ressourcen == rueckwaerts.ressourcen
    assert (vorwaerts.verbrauchsmatrix() == rueckwaerts.verbrauchsmatrix()).all()
    assert (vorwaerts.kapazitaetsvektor() == rueckwaerts.kapazitaetsvektor()).all()


if __name__ == "__main__":
    import sys
    sys.exit(pytest.main([__file__, "-v", "--tb=short", "-p", "no:cacheprovider"]))
```

Code-Durchgang

Stelle	Was passiert
`@pytest.fixture(params=MODELLBAUER, ids=NAMEN)`	Jeder Test läuft **zweimal** — einmal mit GLOP, einmal mit SciPy. Ein Test, der nur einen Solver sieht, prüft die Bibliothek mit statt das Modell
`test_beide_solver_liefern_dasselbe`	Die Architektur aus Kapitel 22 als Test: gleicher Zielwert **und** gleiche Schattenpreise
`test_waehrungseinheit_skaliert_linear`	prüft nicht Mathematik, sondern **Numerik** — schlecht skalierte Modelle (Kapitel 2) fallen hier durch
`test_produktreihenfolge_aendert_nichts`	die Spaltenvertauschungsfalle aus Kapitel 18, als Invariante
`test_pruefung_findet_*`	vier Tests, die die **Prüfung** prüfen
`@pytest.mark.parametrize` bei der Statusübersetzung	sechs Fälle in einer Tabelle, inklusive des unbekannten Rohstatus
`if __name__ == "__main__": pytest.main(...)`	damit die Datei auch ohne `pytest`-Aufruf läuft — sie gehört zur Programmsammlung des Buchs

Der Lauf ist unspektakulär, und das ist der Punkt:

```
$ pytest test_or_kern.py -q
.................................                                    [100%]
33 passed in 0.5s
```

23.5 Wer testet die Tests?

33 grüne Tests. Das ist noch kein Ergebnis.

Eine Testsuite, die nie rot war, ist keine Prüfung, sondern eine Vermutung — sie könnte aus lauter `assert True` bestehen und sähe genauso aus. Die Frage ist nicht, ob die Tests laufen, sondern ob sie etwas **fangen** würden.

Das lässt sich messen. Man baut absichtlich Fehler in den Code ein und schaut, ob die Suite rot wird. Jede Mutation hat genau zwei mögliche Ausgänge:

Ausgang	Bedeutung
getötet	Mindestens ein Test wird rot — diesen Fehler hätte die Suite gefunden.
überlebt	Alle Tests bleiben grün — diesen Fehler hätte sie **durchgehen lassen**.

```
#!/usr/bin/env python3

# Mutationstest.py
"""
Kapitel Testen: Wer testet die Tests?

`test_or_kern.py` besteht aus 31 Tests, und alle sind gruen. Das ist noch kein
Ergebnis. Eine Testsuite, die nie rot war, ist keine Pruefung, sondern eine
Vermutung - sie koennte aus lauter `assert True` bestehen und saehe genauso aus.

Dieses Programm prueft die Pruefung. Es baut nacheinander KLEINE, gezielte
Fehler in `or_kern.py` ein - jeder einzelne ist ein Fehler, den man beim
Programmieren tatsaechlich macht - und laesst die Testsuite darauf los. Fuer
jede Mutation gibt es genau zwei moegliche Ausgaenge:

    GETOETET    Mindestens ein Test wird rot. Die Suite haette diesen Fehler
                in der Praxis gefunden.
    UEBERLEBT   Alle Tests bleiben gruen. Die Suite haette diesen Fehler
                DURCHGEHEN LASSEN - hier fehlt ein Test.

Ueberlebende Mutationen sind das eigentliche Ergebnis: Sie zeigen die Luecken,
und zwar ohne dass man sie erraten muss.

Das Verfahren heisst Mutationstest und ist alt; neu ist nur, wie gut es zu
Optimierungsmodellen passt. Dort kann man den Zielwert meist nicht gegen einen
bekannten Sollwert pruefen - also weiss man ohne so ein Experiment nicht, ob
die Ersatzpruefungen (Eigenschaften, Invarianten) ueberhaupt greifen.

WICHTIG: Es wird nichts veraendert. Jede Mutation laeuft in einem eigenen
temporaeren Verzeichnis mit einer Kopie von or_kern.py.

Benoetigt: pytest; im selben Verzeichnis or_kern.py und test_or_kern.py
"""

from __future__ import annotations

import os
import shutil
import subprocess
import sys
import tempfile

# (Name, gesuchter Text, Ersatz) - jede Zeile ein realistischer Fehler.
MUTATIONEN = [
    ("Nebenbedingung umgedreht (<= wird >=)",
     "<= problem.kapazitaeten[ressource], name=ressource)",
     ">= problem.kapazitaeten[ressource], name=ressource)"),
```

```
    ("Maximierung wird Minimierung",
     "solver.Maximize(sum(menge[p.name] * p.deckungsbeitrag",
     "solver.Minimize(sum(menge[p.name] * p.deckungsbeitrag"),

    ("Vorzeichen der Dualwerte vergessen",
     "schattenpreise={r: float(-m) for r, m in",
     "schattenpreise={r: float(m) for r, m in"),

    ("Abnahmepruefung aufgeweicht (Toleranz 1e-6 -> 1.0)",
     "toleranz: float = 1e-6) -> list[str]:",
     "toleranz: float = 1.0) -> list[str]:"),

    ("Kapazitaetspruefung uebersprungen",
     "if ist > grenze + toleranz:",
     "if False:"),

    ("Pruefung auf doppelte Produktnamen entfernt",
     "if len(set(namen)) != len(namen):",
     "if False:"),

    ("Ganzzahligkeitspruefung entfernt",
     "if abs(wert - round(wert)) > toleranz:",
     "if False:"),

    ("Zielwertabgleich entfernt",
     "if abs(nachgerechnet - loesung.zielwert) > toleranz * max(1.0, abs(nachgerechnet)):",
     "if False:"),

    ("ZEITLIMIT gilt faelschlich als brauchbar",
     "return self in (SolverStatus.OPTIMAL, SolverStatus.ZULAESSIG)",
     "return self in (SolverStatus.OPTIMAL, SolverStatus.ZULAESSIG,\n"
     "                        SolverStatus.ZEITLIMIT)"),

    ("Ressourcenreihenfolge nicht mehr stabil",
     "return sorted(self.kapazitaeten)",
     "return list(self.kapazitaeten)"),
]


def fuehre_suite_aus(verzeichnis: str) -> tuple[int, int, str]:
    """Laesst pytest im Verzeichnis laufen; liefert (bestanden, fehlgeschlagen)."""
    ergebnis = subprocess.run(
        [sys.executable, "-m", "pytest", "test_or_kern.py", "-q",
         "--tb=no", "-p", "no:cacheprovider"],
        cwd=verzeichnis, capture_output=True, text=True, timeout=600)
    letzte = [z for z in ergebnis.stdout.strip().splitlines() if z.strip()]
    zeile = letzte[-1] if letzte else ""
    bestanden = fehlgeschlagen = 0
```

```
    for teil in zeile.replace("=", " ").split(","):
        for wort in teil.split():
            if wort.isdigit():
                zahl = int(wort)
            elif wort.startswith("passed"):
                bestanden = zahl
            elif wort.startswith("failed") or wort.startswith("error"):
                fehlgeschlagen += zahl
    return bestanden, fehlgeschlagen, zeile


def mutiere_und_pruefe(quelle: str, ziel: str, suche: str,
                       ersatz: str) -> tuple[int, int, str] | None:
    """Legt eine mutierte Kopie an und laesst die Suite darauf laufen."""
    with tempfile.TemporaryDirectory() as verzeichnis:
        text = open(quelle, encoding="utf-8").read()
        if text.count(suche) != 1:
            return None                       # Muster passt nicht (mehr)
        open(os.path.join(verzeichnis, "or_kern.py"), "w",
             encoding="utf-8").write(text.replace(suche, ersatz))
        shutil.copy(ziel, verzeichnis)
        return fuehre_suite_aus(verzeichnis)


if __name__ == "__main__":
    hier = os.path.dirname(os.path.abspath(__file__))
    quelle = os.path.join(hier, "or_kern.py")
    suite = os.path.join(hier, "test_or_kern.py")
    for pfad in (quelle, suite):
        if not os.path.exists(pfad):
            raise SystemExit(f"Nicht gefunden: {pfad}\n"
                             f"Beide Dateien muessen im selben Verzeichnis liegen.")

    print("=" * 84)
    print("  MUTATIONSTEST: WUERDE DIE SUITE DIESE FEHLER FINDEN?")
    print("=" * 84)

    with tempfile.TemporaryDirectory() as v:
        shutil.copy(quelle, v)
        shutil.copy(suite, v)
        bestanden, fehlgeschlagen, _ = fuehre_suite_aus(v)
    print(f"Ausgangslage: {bestanden} Tests, {fehlgeschlagen} rot.\n")
    if fehlgeschlagen:
        raise SystemExit("Die Suite ist schon ohne Mutation rot - erst das reparieren.")

    print(f"{'eingebauter Fehler':<48} {'rote Tests':>11} {'Urteil':>12}")
    print("-" * 84)
```

```
    getoetet, ueberlebt = [], []
    for name, suche, ersatz in MUTATIONEN:
        ergebnis = mutiere_und_pruefe(quelle, suite, suche, ersatz)
        if ergebnis is None:
            print(f"{name:<48} {'-':>11} {'nicht anwendbar':>12}")
            continue
        _, rot, _ = ergebnis
        urteil = "GETOETET" if rot else "UEBERLEBT"
        (getoetet if rot else ueberlebt).append(name)
        print(f"{name:<48} {rot:>11} {urteil:>12}")

    print("-" * 84)
    quote = len(getoetet) / max(1, len(getoetet) + len(ueberlebt)) * 100
    print(f"Getoetet: {len(getoetet)} von {len(getoetet) + len(ueberlebt)} "
          f"({quote:.0f} %)")

    print("\n" + "=" * 84)
    print("  WAS DAS ERGEBNIS BEDEUTET")
    print("=" * 84)
    if ueberlebt:
        print("Diese Fehler haette die Suite DURCHGEHEN LASSEN:\n")
        for name in ueberlebt:
            print(f"  * {name}")
        print("\nJeder ueberlebende Eintrag ist eine Testluecke - und zwar eine")
        print("gefundene, keine vermutete. Das ist der ganze Zweck des Verfahrens:")
        print("Es sagt einem, welchen Test man als naechstes schreiben sollte,")
        print("statt dass man raten muss.")
    else:
        print("Alle eingebauten Fehler wurden gefunden. Das heisst nicht, dass die")
        print("Suite vollstaendig ist - nur, dass sie diese zehn Fehler faengt.")
    print()
    print("Die Quote selbst ist keine Kennzahl fuer ein Dashboard. Zehn von Hand")
    print("gewaehlte Mutationen sind keine Stichprobe aus der Menge aller")
    print("moeglichen Fehler. Was zaehlt, ist die LISTE der Ueberlebenden.")
    print("=" * 84)
```

Der erste Lauf fand zwei Lücken

Die Suite hatte zunächst 31 Tests. Der Mutationstest sagte dazu:

```
Nebenbedingung umgedreht (<= wird >=)                    7     GETOETET
Maximierung wird Minimierung                             3     GETOETET
Vorzeichen der Dualwerte vergessen                       2     GETOETET
Abnahmepruefung aufgeweicht (Toleranz 1e-6 -> 1.0)       1     GETOETET
Kapazitaetspruefung uebersprungen                        1     GETOETET
Pruefung auf doppelte Produktnamen entfernt              1     GETOETET
Ganzzahligkeitspruefung entfernt                         1     GETOETET
Zielwertabgleich entfernt                                1     GETOETET
```

```
ZEITLIMIT gilt faelschlich als brauchbar                    0    UEBERLEBT
Ressourcenreihenfolge nicht mehr stabil                     0    UEBERLEBT
```

Zwei Lücken, und beide waren ernst.

Die erste: Ändert man `SolverStatus.brauchbar` so, dass auch `ZEITLIMIT` als brauchbar gilt, merkt es keiner der 31 Tests. Genau diese Unterscheidung entscheidet aber, ob ein Nachtjob einen Plan ausliefert oder Alarm schlägt (Kapitel 22, `Betriebsueberwachung.py`).

Die zweite: Ersetzt man `sorted(self.kapazitaeten)` durch `list(...)`, hängt die Zeilenreihenfolge der Verbrauchsmatrix wieder an der Einfügereihenfolge des Mappings — die Falle aus Kapitel 18, eine Ebene tiefer. Auch das fiel niemandem auf.

Beide Tests stehen jetzt in `test_or_kern.py`; sie tragen einen Kommentar, der sagt, woher sie kommen. Danach:

```
================================================================================
  MUTATIONSTEST: WUERDE DIE SUITE DIESE FEHLER FINDEN?
================================================================================
Ausgangslage: 33 Tests, 0 rot.

eingebauter Fehler                                rote Tests       Urteil
--------------------------------------------------------------------------------
Nebenbedingung umgedreht (<= wird >=)                      7     GETOETET
Maximierung wird Minimierung                               3     GETOETET
Vorzeichen der Dualwerte vergessen                         2     GETOETET
Abnahmepruefung aufgeweicht (Toleranz 1e-6 -> 1.0)           1     GETOETET
Kapazitaetspruefung uebersprungen                          1     GETOETET
Pruefung auf doppelte Produktnamen entfernt                1     GETOETET
Ganzzahligkeitspruefung entfernt                           1     GETOETET
Zielwertabgleich entfernt                                  1     GETOETET
ZEITLIMIT gilt faelschlich als brauchbar                   1     GETOETET
Ressourcenreihenfolge nicht mehr stabil                    1     GETOETET
--------------------------------------------------------------------------------
Getoetet: 10 von 10 (100 %)

================================================================================
  WAS DAS ERGEBNIS BEDEUTET
================================================================================
Alle eingebauten Fehler wurden gefunden. Das heisst nicht, dass die
Suite vollstaendig ist - nur, dass sie diese zehn Fehler faengt.

Die Quote selbst ist keine Kennzahl fuer ein Dashboard. Zehn von Hand
gewaehlte Mutationen sind keine Stichprobe aus der Menge aller
moeglichen Fehler. Was zaehlt, ist die LISTE der Ueberlebenden.
================================================================================
```

□ Was die Quote nicht ist

„100 %" ist keine Kennzahl fürs Dashboard. Zehn von Hand gewählte Mutationen sind keine Stichprobe aus der Menge aller möglichen Fehler — sie sind eine Liste von Fehlern, die *mir* eingefallen sind. Eine Mutation, an die niemand gedacht hat, taucht auch nicht als Lücke auf.

Der Wert des Verfahrens liegt nicht in der Zahl, sondern in der **Liste der Überlebenden**. Sie sagt einem, welchen Test man als nächstes schreiben sollte — statt dass man raten muss.

23.6 Ein Vergleich, dem man glauben kann

Solververgleiche stehen in jedem Blog, und die meisten sind wertlos. Nicht, weil falsch gemessen wurde, sondern weil zu wenig dazugesagt wird.

```
#!/usr/bin/env python3

# Benchmark_Skalierung.py
"""
Kapitel Testen: Ein Vergleich, dem man glauben kann.

Solververgleiche stehen in jedem Blog, und die meisten sind wertlos - nicht
weil falsch gemessen wurde, sondern weil zu wenig dazugesagt wird. Dieses
Programm misst dasselbe Transportproblem in drei Groessen mit vier
Bibliotheken und haelt sich dabei an fuenf Regeln, die den Unterschied machen:

  1. EIGENER PROZESS je Bibliothek. Nicht nur wegen des Importkonflikts
     zwischen ortools und highspy (Kapitel Oekosystem) - auch, damit der
     Speicherverbrauch der einen nicht in der Messung der anderen auftaucht.
  2. AUFBAU UND LOESEN GETRENNT messen. Bei grossen Instanzen ist der Aufbau
     des Modells in Python regelmaessig teurer als das Loesen. Wer nur die
     Gesamtzeit misst, optimiert am Ende die falsche Haelfte.
  3. ZIELWERTE GEGENEINANDER PRUEFEN. Eine Bibliothek, die schneller ist und
     etwas anderes ausrechnet, hat den Vergleich nicht gewonnen. Diese Pruefung
     ist der wichtigste Teil des Programms.
  4. SPEICHER MITMESSEN. Bei 10.000 Variablen entscheidet oft er und nicht die
     Zeit darueber, was auf einer Maschine noch laeuft.
  5. DIESELBE INSTANZ fuer alle. Feste Saat, kein Neuwuerfeln zwischendurch.

Und die Einschraenkung, die dazugehoert: Gemessen wird EIN Problemtyp in EINER
Formulierung auf EINER Maschine. Das Ergebnis ist keine Rangliste der Solver,
sondern eine Entscheidungshilfe fuer genau diesen Fall. Wer es verallgemeinert,
macht denselben Fehler wie jemand, der aus einem Backtest auf die Zukunft
schliesst (Kapitel Handelsmaschine).

Benoetigt: numpy; in den Kindprozessen scipy, highspy, ortools, cvxpy
"""

from __future__ import annotations

import multiprocessing
```

```
import resource
import time
from concurrent.futures import ProcessPoolExecutor

import numpy as np

GROESSEN = [(10, 10), (32, 32), (100, 100)]   # (Lager, Kunden) -> 100 / 1.024 / 10.000
 ↪  Variablen


# Instanz und Speichermessung stehen als gewoehnliche Funktionen hier - nicht
# in einem String, den ein Kindprozess ausfuehrt. Jede Messfunktion baut die
# Instanz aus derselben Saat neu auf, damit ueber die Prozessgrenze nichts
# reist, was das Ergebnis verfaelschen koennte.

def instanz(m: int, n: int):
    rng = np.random.default_rng(20)
    kosten = rng.integers(5, 95, (m, n)).astype(float)
    angebot = rng.integers(50, 150, m).astype(float)
    bedarf = angebot.sum() * rng.dirichlet(np.ones(n))
    return kosten, angebot, bedarf


def speicher_mb() -> float:
    # ru_maxrss ist unter Linux in Kilobyte. Gemessen wird der Kindprozess -
    # deshalb muss jede Messung einen eigenen bekommen.
    return resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024


def messe_scipy(m: int, n: int):
    from scipy.optimize import linprog
    kosten, angebot, bedarf = instanz(m, n)
    t0 = time.perf_counter()
    c = kosten.reshape(-1)
    A_ub = np.zeros((m, m * n)); A_eq = np.zeros((n, m * n))
    for i in range(m):
        A_ub[i, i * n:(i + 1) * n] = 1.0
    for j in range(n):
        A_eq[j, j::n] = 1.0
    aufbau = time.perf_counter() - t0
    t0 = time.perf_counter()
    r = linprog(c=c, A_ub=A_ub, b_ub=angebot, A_eq=A_eq, b_eq=bedarf,
                bounds=(0, None), method="highs")
    loesen = time.perf_counter() - t0
    return float(r.fun), aufbau, loesen, speicher_mb()


def messe_highspy(m: int, n: int):
```

```
    import highspy
    kosten, angebot, bedarf = instanz(m, n)
    t0 = time.perf_counter()
    h = highspy.Highs(); h.setOptionValue("output_flag", False)
    h.addVars(m * n, np.zeros(m * n), np.full(m * n, highspy.kHighsInf))
    for k in range(m * n):
        h.changeColCost(k, float(kosten.reshape(-1)[k]))
    for i in range(m):
        idx = np.arange(i * n, (i + 1) * n, dtype=np.int32)
        h.addRow(-highspy.kHighsInf, float(angebot[i]), n, idx, np.ones(n))
    for j in range(n):
        idx = np.arange(j, m * n, n, dtype=np.int32)
        h.addRow(float(bedarf[j]), float(bedarf[j]), m, idx, np.ones(m))
    aufbau = time.perf_counter() - t0
    t0 = time.perf_counter(); h.run(); loesen = time.perf_counter() - t0
    return h.getInfo().objective_function_value, aufbau, loesen, speicher_mb()


def messe_ortools(m: int, n: int):
    from ortools.linear_solver import pywraplp
    kosten, angebot, bedarf = instanz(m, n)
    t0 = time.perf_counter()
    s = pywraplp.Solver.CreateSolver("GLOP")
    x = [[s.NumVar(0, s.infinity(), f"x{i}_{j}") for j in range(n)]
         for i in range(m)]
    for i in range(m):
        s.Add(sum(x[i]) <= float(angebot[i]))
    for j in range(n):
        s.Add(sum(x[i][j] for i in range(m)) == float(bedarf[j]))
    s.Minimize(sum(float(kosten[i, j]) * x[i][j]
                   for i in range(m) for j in range(n)))
    aufbau = time.perf_counter() - t0
    t0 = time.perf_counter(); s.Solve(); loesen = time.perf_counter() - t0
    return s.Objective().Value(), aufbau, loesen, speicher_mb()


def messe_cvxpy(m: int, n: int):
    import cvxpy as cp
    kosten, angebot, bedarf = instanz(m, n)
    t0 = time.perf_counter()
    x = cp.Variable((m, n), nonneg=True)
    problem = cp.Problem(cp.Minimize(cp.sum(cp.multiply(kosten, x))),
                         [cp.sum(x, axis=1) <= angebot,
                          cp.sum(x, axis=0) == bedarf])
    aufbau = time.perf_counter() - t0
    t0 = time.perf_counter(); problem.solve(); loesen = time.perf_counter() - t0
    return float(problem.value), aufbau, loesen, speicher_mb()
```

```
ANSAETZE = {"scipy.linprog": messe_scipy, "highspy": messe_highspy,
            "ortools/GLOP": messe_ortools, "cvxpy": messe_cvxpy}


def messe(funktion, m: int, n: int):
    """Fuehrt eine Messfunktion in einem FRISCHEN Prozess aus.

    'spawn' und max_tasks_per_child=1 zusammen garantieren, was Regel 1
    verlangt: Jede Messung sieht einen leeren Interpreter. Ohne das
    zweite wuerde der Pool seinen Arbeiter wiederverwenden - dann waere
    der Speicherwert der zweiten Bibliothek um die erste zu hoch, und
    ortools und highspy saessen im selben Prozess.
    """
    with ProcessPoolExecutor(
            max_workers=1,
            mp_context=multiprocessing.get_context("spawn"),
            max_tasks_per_child=1) as pool:
        try:
            return pool.submit(funktion, m, n).result(timeout=600), None
        except Exception as fehler:
            return None, str(fehler).strip().splitlines()[-1][:60]


if __name__ == "__main__":
    print("=" * 92)
    print("  SKALIERUNGSVERGLEICH: TRANSPORTPROBLEM, VIER BIBLIOTHEKEN")
    print("=" * 92)
    print("Jede Zeile ein eigener Prozess. Zeiten und Speicher sind "
          "hardwareabhaengig,")
    print("die Zielwerte und ihr Verhaeltnis zueinander nicht.\n")

    for m, n in GROESSEN:
        kopf = f"--- {m} Lager x {n} Kunden = {m * n:,} Variablen "
        print(kopf + "-" * max(3, 92 - len(kopf)))
        print(f"  {'Bibliothek':<16} {'Zielwert':>14} {'Aufbau':>9} "
              f"{'Loesen':>9} {'Anteil':>8} {'Speicher':>10}")
        print("  " + "-" * 72)
        zielwerte = {}
        for name, funktion in ANSAETZE.items():
            werte, fehler = messe(funktion, m, n)
            if werte is None:
                print(f"  {name:<16} nicht verfuegbar: {fehler}")
                continue
            ziel, aufbau, loesen, speicher = werte
            zielwerte[name] = ziel
            anteil = aufbau / (aufbau + loesen) * 100
            print(f"  {name:<16} {ziel:>14,.2f} {aufbau:>8.3f}s "
```

```
                f"{loesen:>8.3f}s {anteil:>7.0f}% {speicher:>9.0f} MB")

        # Die wichtigste Zeile: Rechnen alle dasselbe aus?
        spanne = max(zielwerte.values()) - min(zielwerte.values())
        bezug = max(abs(v) for v in zielwerte.values())
        print(f"  {'':16} Spannweite der Zielwerte: {spanne:.2e} "
              f"(relativ {spanne / bezug:.1e})")
        if spanne / bezug > 1e-6:
            print("  ACHTUNG: Die Bibliotheken widersprechen sich - "
                  "der Zeitvergleich ist wertlos.")
        print()

    print("=" * 92)
    print("  WAS MAN AUS SO EINER TABELLE ABLESEN DARF - UND WAS NICHT")
    print("=" * 92)
    print("DARF man ablesen:")
    print("  * Die Spalte 'Anteil' - wie viel der Zeit in den AUFBAU geht statt")
    print("    ins Loesen. Wenn dort 80 % stehen, ist ein schnellerer Solver die")
    print("    falsche Antwort; dann gehoert das Modell vektorisiert aufgebaut")
    print("    (Kapitel Oekosystem).")
    print("  * Die Groessenordnung des Speicherbedarfs. Sie entscheidet, was auf")
    print("    einer bestimmten Maschine ueberhaupt laeuft.")
    print("  * Wie sich beides mit der Groesse ENTWICKELT. Der Trend ist")
    print("    uebertragbarer als der Absolutwert.")
    print()
    print("NICHT ablesen darf man:")
    print("  * 'Bibliothek X ist schneller als Y.' Gemessen wurde EIN")
    print("    Problemtyp in EINER Formulierung. Ein MILP, ein QP oder eine")
    print("    andere Modellierung desselben Problems koennen die Reihenfolge")
    print("    umdrehen.")
    print("  * Etwas ueber Ihre Maschine. Diese Zahlen stammen von einer")
    print("    anderen. Der Sinn des Programms ist, dass Sie es auf Ihrer")
    print("    laufen lassen.")
    print("=" * 92)
```

Erwartete Ausgabe (Zeiten und Speicher hardwareabhängig, die Zielwerte nicht):

```
============================================================================================
  SKALIERUNGSVERGLEICH: TRANSPORTPROBLEM, VIER BIBLIOTHEKEN
============================================================================================
Jede Zeile ein eigener Prozess. Zeiten und Speicher sind hardwareabhaengig,
die Zielwerte und ihr Verhaeltnis zueinander nicht.

--- 10 Lager x 10 Kunden = 100 Variablen ---------------------------------------------------
  Bibliothek             Zielwert    Aufbau    Loesen   Anteil   Speicher
  ------------------------------------------------------------------------
  scipy.linprog         13,509.48    0.000s    0.004s       1%        79 MB
  highspy               13,509.48    0.001s    0.002s      36%        44 MB
```

```
  ortools/GLOP          13,509.48    0.003s    0.001s      80%        56 MB
  cvxpy                 13,509.48    0.001s    0.009s       9%       229 MB
                  Spannweite der Zielwerte: 1.33e-06 (relativ 9.8e-11)

--- 32 Lager x 32 Kunden = 1,024 Variablen ------------------------------------------------
  Bibliothek             Zielwert    Aufbau    Loesen   Anteil   Speicher
  ------------------------------------------------------------------------
  scipy.linprog         35,744.25    0.000s    0.009s       3%        81 MB
  highspy               35,744.25    0.004s    0.005s      44%        45 MB
  ortools/GLOP          35,744.25    0.014s    0.002s      85%        58 MB
  cvxpy                 35,744.25    0.001s    0.017s       5%       231 MB
                  Spannweite der Zielwerte: 9.54e-05 (relativ 2.7e-09)

--- 100 Lager x 100 Kunden = 10,000 Variablen ---------------------------------------------
  Bibliothek             Zielwert    Aufbau    Loesen   Anteil   Speicher
  ------------------------------------------------------------------------
  scipy.linprog         72,220.34    0.004s    0.072s       5%       122 MB
  highspy               72,220.34    0.026s    0.032s      45%        50 MB
  ortools/GLOP          72,220.34    0.141s    0.043s      77%        68 MB
  cvxpy                 72,220.34    0.001s    0.108s       1%       242 MB
                  Spannweite der Zielwerte: 3.65e-04 (relativ 5.0e-09)

================================================================================================
  WAS MAN AUS SO EINER TABELLE ABLESEN DARF - UND WAS NICHT
================================================================================================
DARF man ablesen:
  * Die Spalte 'Anteil' - wie viel der Zeit in den AUFBAU geht statt
    ins Loesen. Wenn dort 80 % stehen, ist ein schnellerer Solver die
    falsche Antwort; dann gehoert das Modell vektorisiert aufgebaut
    (Kapitel Oekosystem).
  * Die Groessenordnung des Speicherbedarfs. Sie entscheidet, was auf
    einer bestimmten Maschine ueberhaupt laeuft.
  * Wie sich beides mit der Groesse ENTWICKELT. Der Trend ist
    uebertragbarer als der Absolutwert.

NICHT ablesen darf man:
  * 'Bibliothek X ist schneller als Y.' Gemessen wurde EIN
    Problemtyp in EINER Formulierung. Ein MILP, ein QP oder eine
    andere Modellierung desselben Problems koennen die Reihenfolge
    umdrehen.
  * Etwas ueber Ihre Maschine. Diese Zahlen stammen von einer
    anderen. Der Sinn des Programms ist, dass Sie es auf Ihrer
    laufen lassen.
================================================================================================
```

Die interessanteste Spalte heißt „Anteil"

Sie sagt, wie viel der Zeit in den **Aufbau** des Modells geht statt ins Lösen — und die Zahlen sind unbequem:

Bibliothek	Anteil Aufbau bei 10 000 Variablen
ortools/GLOP	rund 3/4
highspy	rund die Hälfte
scipy.linprog	wenige Prozent
cvxpy	~1 %

(Die genauen Prozentwerte schwanken von Lauf zu Lauf — es sind Verhältnisse zweier Zeitmessungen. Die Größenordnungen sind stabil.)

Bei OR-Tools gehen drei von vier Sekunden dafür drauf, in Python 10 000 Variablenobjekte anzulegen und Nebenbedingungen daraus zusammenzusetzen. Wer an dieser Stelle einen schnelleren *Solver* sucht, sucht in der falschen Hälfte — die Antwort ist ein vektorisierter Modellaufbau (Kapitel 3).

Die zweite unbequeme Zahl steht ganz rechts: **CVXPY braucht rund 240 MB**, unabhängig von der Problemgröße — das ist der Preis seiner Modellierungsschicht. Auf einem Rechner mit knappem Speicher entscheidet diese Spalte und nicht die Laufzeit darüber, was überhaupt läuft.

> **Was diese Tabelle nicht sagt**
>
> Sie sagt **nicht** „ortools ist langsamer als scipy". Gemessen wurde *ein* Problemtyp in *einer* Formulierung auf *einer* Maschine. Ein MILP statt eines LP, eine andere Modellierung desselben Problems oder ein anderer Rechner können die Reihenfolge umdrehen — und bei MILPs tun sie es regelmäßig.
>
> Übertragbar ist der **Trend**, nicht der Absolutwert: dass der Aufbauanteil mit der Größe wächst, dass CVXPY einen konstanten Speichersockel hat. Der Zweck des Programms ist, dass Sie es auf Ihrer Maschine mit Ihrem Problem laufen lassen.

23.7 Das Modell als Dienst

Ein Modell hinter eine HTTP-Schnittstelle zu hängen sieht nach einer Fingerübung aus. Es gibt aber einen Unterschied, der die ganze Bauform bestimmt:

> Eine gewöhnliche Anfrage dauert Millisekunden. Eine Optimierung dauert Sekunden bis Minuten — und manchmal länger, als jemand warten will.

Damit scheidet die naheliegende Lösung aus. Wer den Solver direkt im Request-Handler aufruft, baut einen Dienst, der bei der ersten großen Instanz in den Timeout des Reverse Proxy läuft und bei zehn gleichzeitigen Anfragen alle Arbeiter blockiert.

Die tragfähige Bauform ist **zweistufig**:

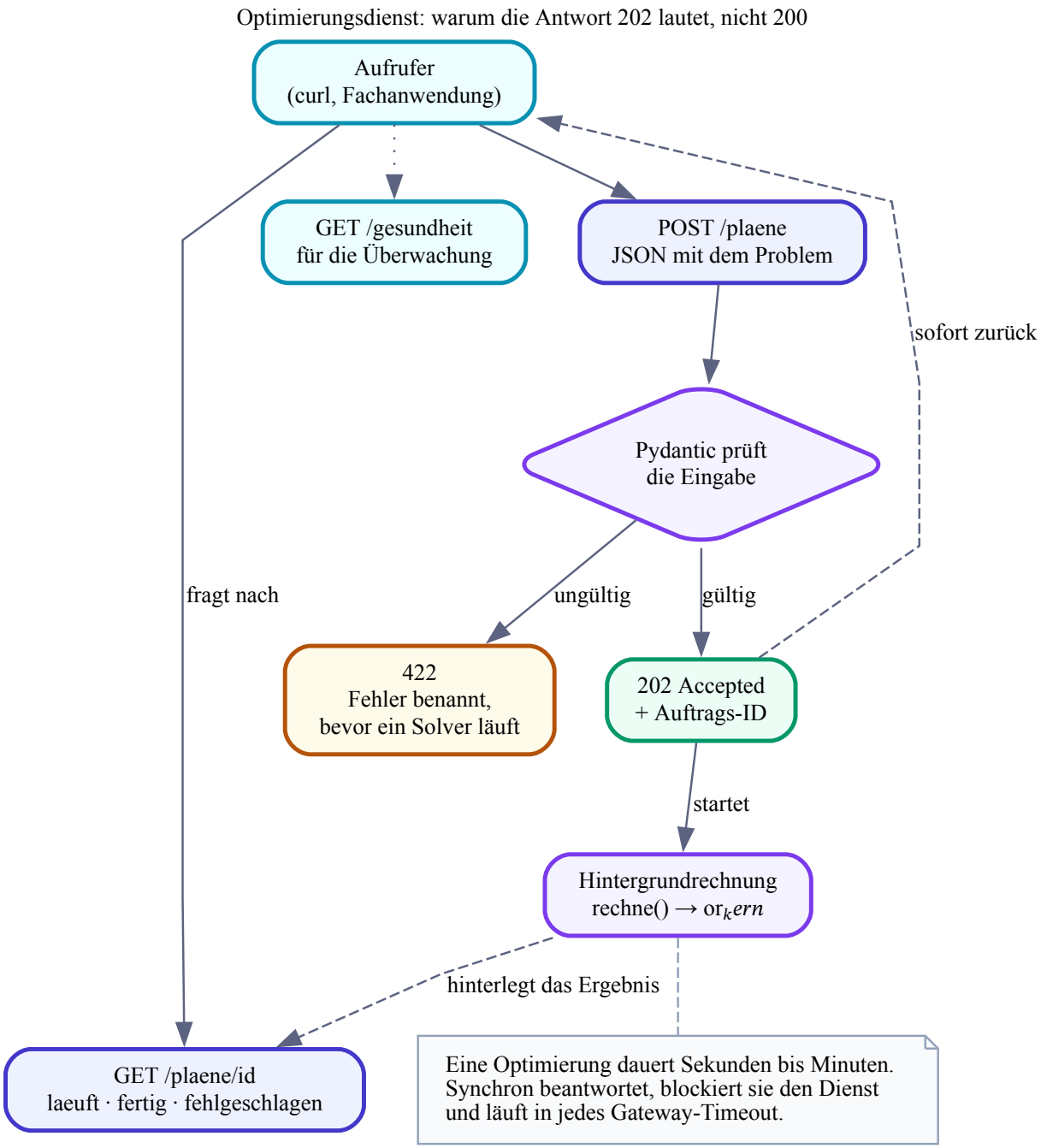

Abbildung 32: Abb. 23.1: Der Ablauf des Optimierungsdienstes. Die Validierung weist eine fehlerhafte Eingabe ab, bevor überhaupt ein Solver startet; die Rechnung selbst läuft im Hintergrund, und der Aufrufer bekommt sofort eine Auftrags-ID. Erzeugt von `bilder_04/erzeuge_architektur_diagramme.py`.

Aufruf	Was passiert
`POST /plaene`	nimmt an, **prüft**, gibt sofort `202 Accepted` und eine Auftragsnummer zurück
`GET /plaene/{id}`	sagt, wie weit es ist — und liefert am Ende die Lösung samt Status und Gap

```
#!/usr/bin/env python3

# Optimierungsdienst.py
"""
Kapitel Testen: Das Modell als Dienst - und warum es kein normaler Endpunkt ist.

Ein Optimierungsmodell hinter eine HTTP-Schnittstelle zu haengen sieht nach
einer Fingeruebung aus. Es gibt aber einen Unterschied, der alles bestimmt:

    Eine gewoehnliche Anfrage dauert Millisekunden.
    Eine Optimierung dauert Sekunden bis Minuten - und manchmal laenger,
    als jemand warten will.

Damit scheidet die naheliegende Bauform aus. Wer den Solver direkt im
Request-Handler aufruft, baut einen Dienst, der bei der ersten grossen Instanz
in einen Timeout des Reverse Proxy laeuft, und bei zehn gleichzeitigen Anfragen
alle Arbeiter blockiert.

Die tragfaehige Bauform ist deshalb ZWEISTUFIG:

    POST /plaene         nimmt den Auftrag an, prueft ihn, gibt sofort
                         202 Accepted und eine Auftragsnummer zurueck
    GET  /plaene/{id}    sagt, wie weit es ist - und liefert am Ende die
                         Loesung samt Status und Gap

Drei Dinge, die dabei leicht untergehen und hier bewusst sichtbar sind:

  1. Das Domaenenmodell aus or_kern.py ist zugleich das API-Schema. Eine
     unsinnige Anfrage wird von FastAPI mit 422 abgewiesen, bevor irgendein
     Solver startet - dieselbe Pruefung wie beim Excel-Import.
  2. Die Antwort enthaelt STATUS und GAP, nicht nur Zahlen. Ein Aufrufer, der
     nur die Mengen bekommt, kann nicht unterscheiden, ob er ein bewiesenes
     Optimum oder einen Zeitlimit-Abbruch in der Hand haelt.
  3. Jeder Auftrag hat ein ZEITLIMIT. Ohne das belegt eine einzige unguenstige
     Instanz einen Arbeiter auf unbestimmte Zeit.

Dieses Programm laeuft ohne Server: Der Selbsttest unten benutzt den
TestClient von FastAPI und spricht die Anwendung direkt an. Fuer den echten
Betrieb steht am Ende, was sich aendert.
```

```
Aufruf:
    python3 Optimierungsdienst.py          # Selbsttest, kein Server noetig
    uvicorn Optimierungsdienst:app         # echter Server auf Port 8000

Benoetigt: fastapi, httpx (fuer den Selbsttest), pydantic, ortools (ueber or_kern)
"""

from __future__ import annotations

import time
import uuid
from concurrent.futures import ThreadPoolExecutor
from enum import Enum

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

from or_kern import (Loesung, Produktionsproblem, loese_mit_glop,
                     pruefe_loesung)

ZEITLIMIT_SEKUNDEN = 30.0
ARBEITER = 2


class Auftragsstand(str, Enum):
    WARTET = "wartet"
    LAEUFT = "laeuft"
    FERTIG = "fertig"
    GESCHEITERT = "gescheitert"


class Auftragsantwort(BaseModel):
    """Was der Aufrufer beim Abholen bekommt.

    Bewusst NICHT nur die Mengen: 'stand' und die Felder aus 'loesung'
    (Status, Gap) sind der Unterschied zwischen einer Zahl und einer
    belastbaren Auskunft.
    """
    id: str
    stand: Auftragsstand
    eingegangen: float
    laufzeit: float | None = None
    loesung: Loesung | None = None
    beanstandungen: list[str] = Field(default_factory=list)
    fehler: str | None = None


app = FastAPI(title="Optimierungsdienst",
```

```
            summary="Produktionsplanung als Auftrag, nicht als Abfrage")

# Fuer das Buchbeispiel: Auftragsbuch im Speicher, Arbeiter im selben Prozess.
# Was daran im echten Betrieb nicht reicht, steht unten.
AUFTRAEGE: dict[str, Auftragsantwort] = {}
POOL = ThreadPoolExecutor(max_workers=ARBEITER)


def rechne(auftrags_id: str, problem: Produktionsproblem) -> None:
    """Laeuft im Arbeiterthread - nie im Request-Handler."""
    antwort = AUFTRAEGE[auftrags_id]
    antwort.stand = Auftragsstand.LAEUFT
    start = time.perf_counter()
    try:
        loesung = loese_mit_glop(problem)
        antwort.loesung = loesung
        # Dieselbe Abnahmepruefung wie ueberall sonst. Ein Dienst, der sie
        # weglaesst, liefert Fehler schneller aus als ein Mensch sie faende.
        antwort.beanstandungen = pruefe_loesung(problem, loesung)
        antwort.stand = (Auftragsstand.FERTIG if loesung.status.brauchbar
                         and not antwort.beanstandungen
                         else Auftragsstand.GESCHEITERT)
        if not loesung.status.brauchbar:
            antwort.fehler = f"Solverstatus: {loesung.status.value}"
        elif antwort.beanstandungen:
            antwort.fehler = "Abnahmepruefung fehlgeschlagen"
    except Exception as fehler:                       # noqa: BLE001
        antwort.stand = Auftragsstand.GESCHEITERT
        antwort.fehler = f"{type(fehler).__name__}: {fehler}"
    finally:
        antwort.laufzeit = time.perf_counter() - start


@app.post("/plaene", status_code=202, response_model=Auftragsantwort)
def auftrag_annehmen(problem: Produktionsproblem) -> Auftragsantwort:
    """Nimmt an, prueft, gibt sofort zurueck.

    Der Typ 'Produktionsproblem' im Parameter ist der ganze Trick: FastAPI
    validiert die Anfrage damit gegen das Domaenenmodell und antwortet bei
    Unsinn mit 422, ohne dass hier eine Zeile Pruefcode steht.
    """
    auftrags_id = str(uuid.uuid4())
    AUFTRAEGE[auftrags_id] = Auftragsantwort(
        id=auftrags_id, stand=Auftragsstand.WARTET, eingegangen=time.time())
    POOL.submit(rechne, auftrags_id, problem)
    return AUFTRAEGE[auftrags_id]
```

```
@app.get("/plaene/{auftrags_id}", response_model=Auftragsantwort)
def auftrag_abholen(auftrags_id: str) -> Auftragsantwort:
    if auftrags_id not in AUFTRAEGE:
        raise HTTPException(status_code=404, detail="Unbekannter Auftrag")
    return AUFTRAEGE[auftrags_id]


@app.get("/gesundheit")
def gesundheit() -> dict[str, object]:
    """Was ein Ueberwachungssystem abfragt - siehe Betriebsueberwachung.py."""
    offen = sum(1 for a in AUFTRAEGE.values()
                if a.stand in (Auftragsstand.WARTET, Auftragsstand.LAEUFT))
    return {"zustand": "bereit", "auftraege_gesamt": len(AUFTRAEGE),
            "offen": offen, "arbeiter": ARBEITER}


# --- Selbsttest ohne Server -------------------------------------------------

SCHREINEREI = {
    "produkte": [
        {"name": "Tisch", "deckungsbeitrag": 240.0,
         "verbrauch": {"Montagestunden": 3.0, "Plattenmaterial": 6.0}},
        {"name": "Stuhl", "deckungsbeitrag": 60.0,
         "verbrauch": {"Montagestunden": 1.0, "Plattenmaterial": 1.0}},
    ],
    "kapazitaeten": {"Montagestunden": 150.0, "Plattenmaterial": 240.0},
}

DOCKERFILE = """\
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY or_kern.py Optimierungsdienst.py .
# Ein Arbeiter je CPU-Kern - der Solver rechnet selbst schon parallel,
# mehr Prozesse machen ihn nicht schneller, sondern langsamer.
CMD ["uvicorn", "Optimierungsdienst:app", "--host", "0.0.0.0", "--port", "8000"]
"""


def warte_auf_ergebnis(klient, auftrags_id: str, grenze: float = 30.0) -> dict:
    ende = time.time() + grenze
    while time.time() < ende:
        antwort = klient.get(f"/plaene/{auftrags_id}").json()
        if antwort["stand"] in ("fertig", "gescheitert"):
            return antwort
        time.sleep(0.02)
    raise TimeoutError("Auftrag wurde nicht fertig")
```

```
if __name__ == "__main__":
    from fastapi.testclient import TestClient

    print("=" * 80)
    print("  DAS MODELL ALS DIENST")
    print("=" * 80)

    with TestClient(app) as klient:
        # 1. Gueltiger Auftrag
        print("\n1. Auftrag einreichen")
        angenommen = klient.post("/plaene", json=SCHREINEREI)
        auftrags_id = angenommen.json()["id"]
        print(f"   POST /plaene            -> {angenommen.status_code} "
              f"{'Accepted' if angenommen.status_code == 202 else ''}")
        print(f"   Auftragsnummer          -> UUID, {len(auftrags_id)} Zeichen")
        # Der entscheidende Punkt, und er ist pruefbar: Die Antwort ist da,
        # BEVOR es eine Loesung gibt. Genau dafuer ist 202 gedacht.
        print(f"   Loesung schon dabei?    -> "
              f"{'ja' if angenommen.json()['loesung'] else 'nein - genau so soll es sein'}")

        fertig = warte_auf_ergebnis(klient, auftrags_id)
        loesung = fertig["loesung"]
        print("\n2. Ergebnis abholen")
        print(f"   Stand                   -> {fertig['stand']}")
        print(f"   Solverstatus            -> {loesung['status']}")
        print(f"   Zielwert                -> {loesung['zielwert']:,.2f} EUR")
        print(f"   Mengen                  -> " + ", ".join(
            f"{name} {wert:.0f}" for name, wert in loesung["werte"].items()))
        print(f"   Schattenpreise          -> " + ", ".join(
            f"{name} {wert:.0f}" for name, wert in loesung["schattenpreise"].items()))
        print(f"   Abnahmepruefung         -> "
              f"{'bestanden' if not fertig['beanstandungen'] else fertig['beanstandungen']}")

        # 2. Unsinnige Anfrage - muss abgewiesen werden, BEVOR gerechnet wird
        print("\n3. Unsinnige Anfragen (der Solver startet gar nicht erst)")
        for beschreibung, aenderung in [
                ("Kapazitaet 0", {"kapazitaeten": {"Montagestunden": 0.0}}),
                ("keine Produkte", {"produkte": []}),
                ("Ressource ohne Kapazitaet",
                 {"produkte": [{"name": "Regal", "deckungsbeitrag": 130.0,
                                "verbrauch": {"Lackieren": 2.0}}]})]:
            anfrage = {**SCHREINEREI, **aenderung}
            antwort = klient.post("/plaene", json=anfrage)
            print(f"   {beschreibung:<24} -> {antwort.status_code} "
                  f"{'Unprocessable Content' if antwort.status_code == 422 else ''}")
```

```
    print("\n4. Unbekannter Auftrag")
    print(f"   GET /plaene/gibtsnicht  -> "
          f"{klient.get('/plaene/gibtsnicht').status_code} Not Found")

    print("\n5. Gesundheitsabfrage")
    print(f"   GET /gesundheit         -> {klient.get('/gesundheit').json()}")

# --- Warum ein Thread je Auftrag genuegt ----------------------------
print("\n" + "-" * 80)
print("6. Warum Threads hier reichen (und wann nicht)\n")
from concurrent.futures import ThreadPoolExecutor as Pool

# Wichtig: eine Rechnung mit ECHTER Last. Ein Modell, das in zehn
# Millisekunden fertig ist, misst nur den Aufwand fuers Threadstarten -
# der erste Entwurf dieser Messung ist genau daran gescheitert und zeigte
# eine Verlangsamung, wo in Wirklichkeit eine Beschleunigung steht.
def eine_rechnung() -> None:
    import numpy as np
    from ortools.linear_solver import pywraplp
    rng = np.random.default_rng(3)
    n = 260
    solver = pywraplp.Solver.CreateSolver("SCIP")
    x = [solver.IntVar(0, 1, f"x{i}") for i in range(n)]
    gewicht = rng.integers(10, 60, n)
    wert = rng.integers(10, 60, n)
    solver.Add(sum(int(gewicht[i]) * x[i] for i in range(n))
               <= int(gewicht.sum() * 0.5))
    for _ in range(30):
        auswahl = rng.choice(n, 40, replace=False)
        solver.Add(sum(x[int(i)] for i in auswahl) <= 12)
    solver.Maximize(sum(int(wert[i]) * x[i] for i in range(n)))
    solver.Solve()

t0 = time.perf_counter()
for _ in range(4):
    eine_rechnung()
seriell = time.perf_counter() - t0
t0 = time.perf_counter()
with Pool(max_workers=4) as p:
    list(p.map(lambda _: eine_rechnung(), range(4)))
parallel = time.perf_counter() - t0

print(f"   vier Rechnungen nacheinander -> {seriell:6.2f} s")
print(f"   vier Rechnungen in Threads   -> {parallel:6.2f} s "
      f"(Faktor {seriell / parallel:.1f})")
print()
if parallel < seriell * 0.75:
    print("   Die Laeufe ueberlappen sich. ortools rechnet in C++ und gibt")
```

```
        print("   den GIL waehrend Solve() frei - deshalb genuegt hier ein")
        print("   Threadpool, es braucht keine eigenen Prozesse.")
    else:
        print("   Keine Ueberlappung - hier waeren Prozesse noetig.")
    print()
    print("   ACHTUNG, das gilt nicht allgemein: Eine in reinem Python")
    print("   geschriebene Heuristik (Kapitel Metaheuristiken) haelt den GIL")
    print("   die ganze Zeit. Fuer sie braucht derselbe Dienst einen")
    print("   ProcessPoolExecutor statt eines Threadpools.")

    print("\n" + "=" * 80)
    print("  WAS SICH IM ECHTEN BETRIEB AENDERT")
    print("=" * 80)
    print("Dieses Beispiel haelt das Auftragsbuch im Speicher und rechnet in")
    print("Threads desselben Prozesses. Das reicht zum Zeigen und fuer einen")
    print("einzelnen Rechner - nicht darueber hinaus:")
    print()
    print("  * NEUSTART LOESCHT ALLES. Auftragsbuch in eine Datenbank oder eine")
    print("    Warteschlange (Redis, RabbitMQ), nicht in ein dict.")
    print("  * ZWEI INSTANZEN KENNEN EINANDER NICHT. Sobald der Dienst mehr als")
    print("    einmal laeuft, muss die Warteschlange ausserhalb liegen -")
    print("    typischerweise Celery mit Redis als Vermittler.")
    print("  * DIE WAHL THREADS/PROZESSE HAENGT AM SOLVER - siehe die Messung")
    print("    unter Punkt 6. Sie gehoert gemessen, nicht angenommen.")
    print("  * OHNE ZEITLIMIT KEIN DIENST. Jede Instanz bekommt eines, und der")
    print("    Aufrufer erfaehrt im Status, ob es gegriffen hat.")
    print()
    print("Das zugehoerige Dockerfile ist kurz genug, um es ganz zu zeigen:")
    print()
    for zeile in DOCKERFILE.splitlines():
        print(f"    {zeile}")
    print()
    print("(Es wird hier nicht gebaut - das Buch setzt keine laufende")
    print(" Docker-Installation voraus.)")
    print("=" * 80)
```

Erwartete Ausgabe (Laufzeiten hardwareabhängig):

```
================================================================================
  DAS MODELL ALS DIENST
================================================================================

1. Auftrag einreichen
   POST /plaene            -> 202 Accepted
   Auftragsnummer          -> UUID, 36 Zeichen
   Loesung schon dabei?    -> nein - genau so soll es sein

2. Ergebnis abholen
```

```
   Stand                   -> fertig
   Solverstatus            -> optimal
   Zielwert                -> 10,800.00 EUR
   Mengen                  -> Tisch 30, Stuhl 60
   Schattenpreise          -> Montagestunden 40, Plattenmaterial 20
   Abnahmepruefung         -> bestanden

3. Unsinnige Anfragen (der Solver startet gar nicht erst)
   Kapazitaet 0             -> 422 Unprocessable Content
   keine Produkte           -> 422 Unprocessable Content
   Ressource ohne Kapazitaet -> 422 Unprocessable Content

4. Unbekannter Auftrag
   GET /plaene/gibtsnicht  -> 404 Not Found

5. Gesundheitsabfrage
   GET /gesundheit         -> {'zustand': 'bereit', 'auftraege_gesamt': 1, 'offen': 0,
    ↪  'arbeiter': 2}

--------------------------------------------------------------------------------
6. Warum Threads hier reichen (und wann nicht)

   vier Rechnungen nacheinander ->   1.88 s
   vier Rechnungen in Threads   ->   0.54 s (Faktor 3.5)

   Die Laeufe ueberlappen sich. ortools rechnet in C++ und gibt
   den GIL waehrend Solve() frei - deshalb genuegt hier ein
   Threadpool, es braucht keine eigenen Prozesse.

   ACHTUNG, das gilt nicht allgemein: Eine in reinem Python
   geschriebene Heuristik (Kapitel Metaheuristiken) haelt den GIL
   die ganze Zeit. Fuer sie braucht derselbe Dienst einen
   ProcessPoolExecutor statt eines Threadpools.

================================================================================
  WAS SICH IM ECHTEN BETRIEB AENDERT
================================================================================
Dieses Beispiel haelt das Auftragsbuch im Speicher und rechnet in
Threads desselben Prozesses. Das reicht zum Zeigen und fuer einen
einzelnen Rechner - nicht darueber hinaus:

  * NEUSTART LOESCHT ALLES. Auftragsbuch in eine Datenbank oder eine
    Warteschlange (Redis, RabbitMQ), nicht in ein dict.
  * ZWEI INSTANZEN KENNEN EINANDER NICHT. Sobald der Dienst mehr als
    einmal laeuft, muss die Warteschlange ausserhalb liegen -
    typischerweise Celery mit Redis als Vermittler.
  * DIE WAHL THREADS/PROZESSE HAENGT AM SOLVER - siehe die Messung
    unter Punkt 6. Sie gehoert gemessen, nicht angenommen.
  * OHNE ZEITLIMIT KEIN DIENST. Jede Instanz bekommt eines, und der
```

```
    Aufrufer erfaehrt im Status, ob es gegriffen hat.

Das zugehoerige Dockerfile ist kurz genug, um es ganz zu zeigen:

    FROM python:3.12-slim
    WORKDIR /app
    COPY requirements.txt .
    RUN pip install --no-cache-dir -r requirements.txt
    COPY or_kern.py Optimierungsdienst.py .
    # Ein Arbeiter je CPU-Kern - der Solver rechnet selbst schon parallel,
    # mehr Prozesse machen ihn nicht schneller, sondern langsamer.
    CMD ["uvicorn", "Optimierungsdienst:app", "--host", "0.0.0.0", "--port", "8000"]

(Es wird hier nicht gebaut - das Buch setzt keine laufende
 Docker-Installation voraus.)
================================================================================
```

Code-Durchgang: drei Entscheidungen

1. Das Domänenmodell ist das API-Schema. Der Parameter heißt `problem: Produktionsproblem` — mehr steht nicht da. FastAPI validiert die Anfrage damit gegen dieselben Pydantic-Regeln wie der Excel-Import aus Kapitel 1 und antwortet bei Unsinn mit `422`, **bevor** ein Solver startet. Die drei abgewiesenen Anfragen im Selbsttest sind genau die drei Fälle, die `or_kern.py` schon beim Einlesen abfängt.

2. Die Antwort enthält Status und Gap, nicht nur Zahlen. Ein Aufrufer, der nur die Mengen bekommt, kann nicht unterscheiden, ob er ein bewiesenes Optimum oder einen Zeitlimit-Abbruch in der Hand hält. Das `Loesung`-Objekt geht deshalb vollständig durch die Schnittstelle — dasselbe DTO, das in Abschnitt 22.7 schon eine Prozessgrenze überlebt hat.

3. Die Abnahmeprüfung läuft auch hier. `pruefe_loesung()` steht im Arbeiter, nicht nur im Skript. Ein Dienst ohne sie liefert Fehler schneller aus, als ein Mensch sie finden könnte.

Threads oder Prozesse? Eine Messung, keine Meinung

Punkt 6 des Selbsttests misst, ob vier Solverläufe in Threads sich überlappen. Das Ergebnis ist ein Faktor von rund 3,5 bei vier Threads: **OR-Tools rechnet in C++ und gibt den GIL während `Solve()` frei.** Ein Threadpool genügt also, es braucht keine eigenen Prozesse.

Das gilt aber nicht allgemein. Eine in reinem Python geschriebene Heuristik (Kapitel 9) hält den GIL die ganze Zeit — für sie braucht derselbe Dienst einen `ProcessPoolExecutor`. Die Entscheidung gehört gemessen, nicht angenommen.

Der erste Entwurf dieser Messung war falsch

Sie lief zunächst gegen ein LP mit 120 Produkten, das in zehn Millisekunden fertig war. Ergebnis: Faktor 0,7 — die Threads schienen zu *bremsen*. Gemessen wurde aber nicht der GIL, sondern der Aufwand fürs Threadstarten, der bei zehn Millisekunden Rechenzeit alles überdeckt.

Das ist die häufigste Art, eine Parallelisierungsmessung zu verderben: **Die Arbeitseinheit muss groß genug sein, dass der Verwaltungsaufwand daneben verschwindet.** Im Programm steht deshalb jetzt ein MILP mit knapp einer halben Sekunde Rechenzeit — und der Kommentar, warum.

Was im echten Betrieb dazukommt

Das Beispiel hält das Auftragsbuch in einem `dict` und rechnet in Threads desselben Prozesses. Für einen einzelnen Rechner reicht das; darüber hinaus nicht:

Grenze	Was stattdessen
Neustart löscht alle Aufträge	Auftragsbuch in eine Datenbank oder Warteschlange
Zwei Instanzen kennen einander nicht	Warteschlange **außerhalb** des Prozesses — Celery mit Redis als Vermittler
Ein Auftrag kann ewig laufen	Zeitlimit je Instanz, und der Aufrufer erfährt im Status, ob es gegriffen hat
Kein Ort für das Ergebnis	Ergebnisspeicher mit Verfallsdatum statt eines wachsenden `dict`

Das Dockerfile ist kurz genug, um es ganz zu zeigen — es steht im Programm und wird beim Lauf mit ausgegeben. Gebaut wird es hier nicht: Das Buch setzt keine laufende Docker-Installation voraus.

23.8 Übungsaufgaben

Lösungen: Abschnitt A.23.

Aufgabe 23.1 □ **— Den Schnellstart reparieren.** Der gerundete Plan verletzt die Kapazität. Nennen Sie zwei Wege, das zu beheben, und sagen Sie, welcher der richtige ist — und warum Kapitel 6 das ausführlich behandelt.

Aufgabe 23.2 □ **— Eine Invariante mehr.** Schreiben Sie einen Test: Verdoppelt man **alle** Kapazitäten, muss sich der Zielwert verdoppeln und die Mengen ebenfalls. Warum gilt das bei einem LP, aber nicht bei einem MILP?

Aufgabe 23.3 □□ **— Eine eigene Mutation.** Ergänzen Sie `MUTATIONEN` um zwei weitere realistische Fehler — zum Beispiel ein vertauschtes Ungleichheitszeichen in `pruefe_loesung()` oder eine Toleranz mit falschem Vorzeichen. Überleben sie?

Aufgabe 23.4 □□ **— Der Benchmark mit MILP.** Ersetzen Sie im Benchmark das Transportproblem durch dieselbe Aufgabe mit ganzzahligen Mengen. Bleibt die Reihenfolge der Bibliotheken gleich? Was passiert mit der Spalte „Anteil"?

Aufgabe 23.5 □□□ **— Der Dienst mit Zeitlimit.** `ZEITLIMIT_SEKUNDEN` steht im Programm, wird aber nirgends benutzt. Reichen Sie es an den Modellbauer durch, geben Sie im Ergebnis aus, ob es gegriffen hat, und schreiben Sie einen Test, der eine Instanz einreicht, die das Limit überschreitet.

23.9 Finde den Denkfehler

„Die Suite ist grün, das Modell stimmt"

Ein Team liefert ein Produktionsplanungsmodell aus. Die Testsuite hat 33 Tests, der Mutationstest tötet alle zehn eingebauten Fehler, und die Abnahmeprüfung läuft in jedem Nachtlauf mit. Nach drei Wochen im Betrieb meldet die Fertigung, dass ihr regelmäßig Lackierkapazität fehlt.

Die Untersuchung zeigt: In der Stammdatentabelle steht der Lackierverbrauch für Regale mit **0,4 Stunden je Stück**. Tatsächlich sind es 4,0 — beim Anlegen ist die Kommastelle verrutscht.

Sämtliche Tests waren grün. Der Mutationstest tötete alles. Die Abnahmeprüfung hat nie etwas beanstandet.

Warum konnte keine dieser Prüfungen den Fehler finden — und was hätte geholfen?

Ein Hinweis: Die Antwort steht schon im Schnellstart dieses Kapitels, im Absatz „Warum funktioniert das?".

23.10 Micro-Quiz

Drei Fragen

1. Warum lässt sich ein Optimierungsmodell selten gegen einen Sollwert testen? a) Weil Solver nicht deterministisch sind. b) Weil niemand die richtige Antwort unabhängig ausrechnen kann — sonst bräuchte man den Solver nicht. c) Weil Gleitkommazahlen keinen exakten Vergleich erlauben.

2. Eine Mutation „überlebt". Was heißt das? a) Der eingebaute Fehler war harmlos. b) Die Testsuite hätte diesen Fehler durchgehen lassen — hier fehlt ein Test. c) Der Test war zu streng eingestellt.

3. Im Benchmark stehen bei einer Bibliothek 77 % in der Spalte „Anteil". Was folgt daraus? a) Diese Bibliothek hat den langsamsten Solver. b) Der Modellaufbau in Python dominiert; ein schnellerer Solver würde wenig ändern. c) Die Messung ist fehlerhaft, Aufbau darf nie so lange dauern.

23.11 Selbsttest

1. Nennen Sie die vier Testarten für Optimierungsmodelle und je ein Beispiel.
2. Warum läuft in `test_or_kern.py` jeder Test über beide Modellbauer?
3. Was ist der Unterschied zwischen „die Suite ist grün" und „die Suite prüft etwas"?
4. Warum trennt der Benchmark Aufbau und Lösen — was übersieht man sonst?
5. Warum antwortet `POST /plaene` mit `202` und nicht mit `200` samt Lösung?
6. Wann genügt für einen Optimierungsdienst ein Threadpool, wann braucht es Prozesse?

23.12 Zusammenfassung

- Ein Optimierungsmodell lässt sich fast nie gegen den **richtigen Wert** testen, aber immer gegen seine **Eigenschaften**: Zulässigkeit, Invarianten, eine kleine Regressionsinstanz, Fehlerfälle.
- Die am häufigsten fehlende Testart ist die vierte. Eine Abnahmeprüfung, die nie etwas gefunden hat, könnte kaputt sein — man muss sie mit falschen Lösungen füttern, um es zu wissen.
- Jeder Test sollte **beide Modellbauer** durchlaufen. Sonst prüft man die Bibliothek mit statt das Modell.
- **Mutationstests** beantworten die Frage, die eine grüne Suite offenlässt: Würde sie einen Fehler überhaupt bemerken? Wertvoll ist nicht die Quote, sondern die Liste der Überlebenden — hier waren es zwei, und beide führten zu einem neuen Test.
- Ein **Benchmark** muss Aufbau und Lösen trennen, Zielwerte gegeneinander prüfen, in getrennten Prozessen laufen und seine eigene Reichweite benennen. Der Aufbauanteil ist regelmäßig die überraschendere Zahl.
- Ein **Optimierungsdienst** ist zweistufig: annehmen und prüfen sofort, rechnen im Hintergrund, Ergebnis mit Status und Gap abholen. Die Frage Threads oder Prozesse gehört gemessen — bei C++-Solvern genügen Threads, bei Python-Heuristiken nicht.

Synthese Teil V — Vom rechnenden Modell zum benutzten System

Zwei Kapitel, ein Übergang: Das Modell rechnet — und muss jetzt jemand anderem übergeben werden. Was dabei schiefgeht, hat selten mit Mathematik zu tun.

Was schiefgeht, und was dagegen hilft

Was in Produktion passiert	Gegenmittel	Wo
`INFEASIBLE` um 3 Uhr nachts, niemand weiß warum	Relaxation mit gestaffelten Strafkosten (B18); Deletion Filter für die Diagnose	Kapitel 22, Anhang C
Der Disponent lehnt den Plan ab, weil er ihn nicht versteht	Constraint-Trace für die einzelne Zuweisung; Kostenzurechnung für den ganzen Plan	Kapitel 22, Abschnitt 22.4

Was in Produktion passiert	Gegenmittel	Wo
Der Solver läuft ins Zeitlimit und keiner merkt es	Status, Gap und Zeitausschöpfung protokollieren — konstante Laufzeit ist ein Warnsignal	Kapitel 22
Ein Test wird grundlos mal rot, mal grün	auf Zielwert und Regeln prüfen, nicht auf die Gestalt der Lösung; `num_workers = 1` **und** Seed	Abschnitt 7.7, Kapitel 23
Ein Ergebnis lässt sich später nicht mehr nachvollziehen	Snapshot-Prinzip: unveränderlicher Datenstand mit ID je Lauf	Kapitel 22
Das Modell prüft sich selbst	Abnahmeprüfung als eigener Baustein, ohne Solver und ohne Modellvariable	Abschnitt 22.6

Was dieser Teil gemessen hat

Behauptung	Gemessen	Wo
„Der Solver nennt schon den Grund für `INFEASIBLE`."	Er nennt gar nichts. Der Deletion Filter findet einen kleinsten Konflikt — dieses Modell enthält **neun** verschiedene, und welchen man sieht, steuert die Prüfreihenfolge	Anhang C
„Bindende Bedingungen sind die teuren."	Von fünf bindenden Bedingungen kostet eine **0,00 €**	Abschnitt 22.4
„Der Schattenpreis sagt, was der Hebel bringt."	Hochgerechnet 120 €, gemessen **60 €** — er gilt nur bis +15 Stunden. Die Ranglisten nach Preis und nach Wirkung drehen sich um	Abschnitt 22.4
„Mehr Arbeiter sind proportional schneller."	**12,3-fach** bei acht Arbeitern — überlinear, weil verschiedene Strategien statt derselben Suche laufen	Abschnitt 7.7

Drei Fehler, die dieser Teil verhindert

1. **Alles hart formulieren.** Wer keine Regel brechen lässt, bekommt irgendwann eine Fehlermeldung statt eines Plans — und zwar nachts, im Batchlauf.
2. **Genauer optimieren als die Daten sind.** Ein Gap von 2 % ist bei ±10 % Datenunsicherheit bedeutungslos. Die Rechenzeit dafür ist verschenkt.
3. **Einen Bericht für einen Beweis halten.** Ein generierter Text erklärt das **Modell**, nicht die **Wirklichkeit**. Ist eine Eingabe falsch geschätzt, ist er überzeugend **und** falsch — die gefährlichste Kombination.

Wenn Sie nur eines mitnehmen

 Ein Modell, das nur auf Ihrem Rechner und nur mit Ihren Daten läuft, ist ein Prototyp — kein System. Der Unterschied besteht aus drei Dingen, die alle nichts mit dem Solver zu tun haben: einer Abnahmeprüfung, einer Erklärung und einem Protokoll.

Danach: die Projektwerkstatt. Elf Aufträge, jeder mit Datenquellen, Modellskizze, Abnahmekriterien und Stolperfallen — zugeschnitten auf 10 bis 25 Stunden bis zu einem vorzeigbaren Ergebnis.

Projektwerkstatt — elf eigene Anwendungen

 Auf einen Blick

Worum geht es? Um den Schritt, auf den der ganze Kurs zuläuft: **eine eigene Anwendung bauen.** Elf vollständig ausgearbeitete Projektaufträge, jeder mit Datenquellen, Modellskizze, Abnahmekriterien und Stolperfallen. P1–P8 bauen je ein Modell; **P9–P11 setzen dort an, wo eines schon steht** — wenn es zu langsam wird, wenn zwei Ziele gegeneinanderstehen, wenn es jemand anderes benutzen soll.

Voraussetzungen: je Projekt angegeben.

Zeitbedarf: 10–25 Stunden je Projekt.

Wie Sie ein Projekt bearbeiten

Halten Sie sich an diese Reihenfolge — sie ist aus vielen gescheiterten und einigen gelungenen Projekten destilliert:

Phase	Dauer	Ergebnis	Häufigster Fehler
1. Zerlegen	10 %	Ausgefüllte Bausteine-Vorlage (Abschnitt 1.6) auf **Papier**	Sofort programmieren

Phase	Dauer	Ergebnis	Häufigster Fehler
2. Kleinstinstanz	15 %	Modell mit 3–5 Elementen, Lösung **von Hand** geprüft	Gleich mit echten Daten anfangen
3. Daten	20 %	Echte Daten geladen, validiert, dokumentiert	Datenqualität unterschätzen
4. Skalieren	20 %	Volles Modell läuft im Zeitlimit	Zu viele harte Bedingungen
5. Erklärbar machen	20 %	Report, den ein Fachanwender versteht	Ganz weglassen
6. Abnahme	15 %	Kriterien geprüft, Grenzen dokumentiert	„Läuft ja“ als Abnahme

 Die wichtigste Regel Phase 2 ist nicht optional. Ein Modell, das Sie an drei Mitarbeitenden und vier Schichten nicht von Hand nachrechnen können, werden Sie an 200 Mitarbeitenden nie debuggen. Jede Stunde in Phase 2 spart drei in Phase 4.

P1 — Vertretungsplaner für eine Schule

Schwierigkeit: □□ · **Kapitel:** Kapitel 1, Kapitel 6, Kapitel 7, Kapitel 22 · **Zeit:** ca. 15 Stunden

Ausgangslage. Morgens um 7:15 Uhr meldet sich die dritte Lehrkraft krank. Die stellvertretende Schulleitung hat 20 Minuten, um einen Vertretungsplan zu erstellen, der Qualifikationen, Arbeitszeiten, bereits geleistete Vertretungen und persönliche Wünsche berücksichtigt.

Daten. Stundenplan (CSV: Klasse, Tag, Stunde, Fach, Lehrkraft, Raum) — in diesem Projektordner erzeugen die Skripte `stundenplan*.py` genau solche Dateien und eignen sich als Datenquelle. Zusätzlich: Qualifikationsmatrix, Deputate, Abwesenheitsmeldungen.

Modellskizze. * Variablen: $x_{p,s} \in \{0,1\}$ — Person p übernimmt Slot s. * Hart: genau eine Person je Slot; Qualifikation; keine Doppelbelegung; Höchstzahl Vertretungsstunden; Ruhezeiten. * Weich: Vorbelastung, Freistundenlöcher, Fachnähe, Wünsche, Fairness über die Woche. * Solver: CP-SAT.

Abnahmekriterien. - [ ] Läuft in unter 10 Sekunden für eine ganze Schule (60 Lehrkräfte, 30 Slots). - [ ] Liefert **immer** einen Plan — auch wenn nicht alle Slots besetzbar sind (Relaxation). - [ ] Weist je Zuweisung eine Begründung aus. - [ ] Ausgabe als CSV **und** als lesbare Tagestabelle. - [ ] Vergleich gegen die manuelle Lösung: Wie viele Wünsche werden erfüllt?

Stolperfallen. * Zu viele harte Regeln → `INFEASIBLE` an genau dem Tag, an dem man das System braucht. * Fairness nur über einen Tag statt über die Woche → dieselbe Person trifft es ständig. * Akzeptanz: Ohne Begründungsanzeige wird der Plan überschrieben.

Erweiterungen. Mehrtagesplanung; Raumkonflikte; Springerstunden; Schnittstelle zum Schulverwaltungsprogramm.

P2 — Schichtplanung für ein Pflegeteam

Schwierigkeit: □□□ · **Kapitel:** Kapitel 6, Kapitel 7, Kapitel 22 · **Zeit:** ca. 20 Stunden

Ausgangslage. 25 Pflegekräfte, drei Schichten täglich, 28-Tage-Zyklus. Es gelten Arbeitszeitgesetz, Tarifvertrag, Qualifikationsmix und individuelle Wünsche.

Modellskizze. * Variablen: $x_{p,t,s} \in \{0,1\}$ — Person, Tag, Schicht. * Hart: Mindestbesetzung je Schicht **und** Qualifikationsmix (mindestens eine examinierte Kraft); nach Nachtschicht ≥ 2 freie Tage; höchstens 6 Arbeitstage in Folge; monatliche Sollstunden ± 10 %. * Weich: Wunschfrei, gleichmäßige Wochenendverteilung, ungeteilte Dienste, stabile Schichtfolgen (nicht Früh–Nacht–Früh). * Solver: CP-SAT mit Intervall- und Cumulative-Constraints.

Abnahmekriterien. - [ ] Alle gesetzlichen Regeln nachweislich eingehalten (per `assert` geprüft). - [ ] Wunscherfüllungsquote wird ausgewiesen und ist fair verteilt (Gini-Koeffizient). - [ ] Läuft in unter 60 Sekunden. - [ ] Bei Unlösbarkeit: Notfallplan plus benannte Ursache.

Stolperfallen. Der 28-Tage-Zyklus erzeugt viele Variablen (25 × 28 × 3 = 2100) — Symmetriebrechung und gute Suchheuristiken werden wichtig. Fairness über einen Monat ist etwas anderes als Fairness über eine Woche.

P3 — Tourenplanung für einen Lieferdienst

Schwierigkeit: □□ · **Kapitel:** Kapitel 8 · **Zeit:** ca. 12 Stunden

Ausgangslage. Ein regionaler Lieferdienst fährt täglich 40–80 Adressen mit 4 Fahrzeugen an. Kunden haben Zeitfenster, Fahrzeuge Kapazitäten, Fahrer Arbeitszeiten.

Daten. Adressen aus einer CSV; Entfernungen über OpenStreetMap (`osmnx`, `openrouteservice`) oder als Luftlinie mit Umwegfaktor 1,3 als Näherung.

Modellskizze. OR-Tools Routing-Bibliothek mit Kapazitäts- und Zeitdimension; Metaheuristik `GUIDED_LOCAL_SEARCH`, Zeitlimit 30 Sekunden.

Abnahmekriterien. - [ ] Alle Kunden werden innerhalb ihrer Zeitfenster beliefert. - [ ] Vergleich gegen die bisherige manuelle Tourenplanung: Ersparnis in km und Minuten. - [ ] Kartendarstellung der Touren (`folium`). - [ ] Robust gegen Ausfall eines Fahrzeugs (Neuplanung in unter 30 s).

Stolperfallen. Luftlinie unterschätzt Fahrzeiten systematisch — mit realistischem Umwegfaktor arbeiten oder echte Routing-Distanzen holen. Zeitfenster, die physisch nicht erreichbar sind, führen zu „keine Lösung“ ohne Erklärung: Vorabprüfung einbauen (siehe Kapitel 8).

P4 — Standort- und Lagernetzplanung

Schwierigkeit: □□□ · **Kapitel:** Kapitel 6, Kapitel 8 · **Zeit:** ca. 15 Stunden

Ausgangslage. Ein Handelsunternehmen prüft, welche von 12 möglichen Lagerstandorten eröffnet werden sollen, um 60 Filialen zu versorgen.

Modellskizze. Kombiniertes Standort- und Transportproblem: * $y_j \in \{0,1\}$ — Lager j eröffnen (Fixkosten). * $x_{ij} \geq 0$ — Menge von Lager j zu Filiale i. * Kopplung: $x_{ij} \leq M\,y_j$; Kapazität je Lager; Bedarfsdeckung je Filiale. * Ziel: Fixkosten + Transportkosten minimieren.

Abnahmekriterien. - [ ] Sensitivitätsanalyse: Wie verändert sich die Lösung bei ±20 % Transportkosten? - [ ] Was-wäre-wenn: Standort X wird politisch vorgegeben — was kostet das? - [ ] Kartendarstellung mit Zuordnungslinien. - [ ] Amortisationsrechnung über 10 Jahre.

Stolperfallen. Big-M zu groß wählen (Kapitel 6) — hier ist die Lagerkapazität die natürliche Wahl. Fixkosten sind einmalig, Transportkosten laufend: **Barwerte rechnen**, nicht einfach addieren.

P5 — Produktionsplanung mit Rüstkosten

Schwierigkeit: □□□ · **Kapitel:** Kapitel 5, Kapitel 6, Kapitel 7 · **Zeit:** ca. 18 Stunden

Ausgangslage. Eine Fertigung produziert 15 Varianten auf 4 Maschinen. Jeder Produktwechsel kostet Rüstzeit; die Rüstzeit hängt von der Reihenfolge ab.

Modellskizze. Los- und Reihenfolgeplanung: * Variablen: Produktionsmengen, Rüstentscheidungen, Reihenfolge (Intervallvariablen). * Hart: Bedarfsdeckung je Periode, Maschinenkapazität, Mindestlosgrößen. * Ziel: Rüst- + Lager- + Fehlmengenkosten minimieren. * Solver: CP-SAT mit `AddNoOverlap` und reihenfolgeabhängigen Übergangszeiten.

Abnahmekriterien. - [ ] Gantt-Diagramm der Maschinenbelegung. - [ ] Vergleich gegen die aktuelle Praxis (Ersparnis in Rüststunden). - [ ] Schattenpreise: Welche Maschine ist der Engpass, was wäre eine zusätzliche Schicht wert?

P6 — Portfolio-Rebalancer für ein Privatdepot

Schwierigkeit: □□□ · **Kapitel:** Kapitel 18 bis Kapitel 21 · **Zeit:** ca. 20 Stunden

> □ **Keine Anlageberatung.** Dieses Projekt dient dem Methodenverständnis. Setzen Sie kein echtes Geld auf ein selbstgebautes Modell, dessen Grenzen Sie nicht vollständig verstehen.

Ausgangslage. Ein Privatdepot aus 10–15 ETFs und Einzeltiteln soll quartalsweise zurückgeführt werden — unter Berücksichtigung von Ordergebühren, Mindestordergrößen und der Steuerfreibetragsnutzung.

Modellskizze. * Variablen: Zielgewichte w_i; Binärvariablen für „Position wird gehandelt“. * Hart: Vollinvestition, keine Leerverkäufe, Positionsobergrenzen, Mindestordergröße (semikontinuierlich, Kapitel 6), höchstens K Transaktionen je Rebalancing. * Ziel: erwartete Rendite − Risikoterm − Transaktionskosten − Steuerwirkung. * Solver: CVXPY (konvexer Teil) oder MIQP für die Kardinalität.

Abnahmekriterien. - [ ] Toleranzband: Es wird nur gehandelt, wenn die Abweichung > 5 Prozentpunkte beträgt. - [ ] Alle Kosten (Ordergebühr, Spread, Steuer) sind explizit ausgewiesen. - [ ] Backtest über 5 Jahre mit Lookahead-Selbsttest. - [ ] Vergleich gegen: nie rebalancen, jährlich rebalancen, Gleichgewichtung. - [ ] Bericht, den ein Nicht-Fachmann versteht.

Stolperfallen. Steuern sind pfadabhängig (FIFO, Freibetrag) und passen nicht sauber in ein einperiodiges Modell — Näherung wählen und die Näherung **dokumentieren**. Und: Wenn Ihre Strategie die Gleichgewichtung nicht schlägt, ist das ein Ergebnis, kein Misserfolg.

P7 — Risikoreport mit CVaR und Stresstests

Schwierigkeit: □□ · **Kapitel:** Kapitel 12, Kapitel 18, Kapitel 20 · **Zeit:** ca. 12 Stunden

Ausgangslage. Für ein bestehendes Portfolio soll ein monatlicher Risikobericht entstehen.

Inhalte des Berichts. * VaR und CVaR auf 95 % und 99 %, historisch **und** Monte-Carlo-simuliert. * Risikobeiträge je Position (*marginal CVaR*): Wer treibt das Risiko? * Stresstests: Was passiert bei −20 % Aktienmarkt, +200 Basispunkten Zins, Korrelationsanstieg auf 0,9? * Historische Szenarien nachspielen (2008, März 2020). * Konzentrationskennzahlen (Herfindahl-Index, effektive Titelzahl).

Abnahmekriterien. - [ ] Bericht als PDF, automatisch erzeugt. - [ ] Alle Kennzahlen mit zwei unabhängigen Methoden berechnet und verglichen. - [ ] Klartext-Zusammenfassung: „Im schlechtesten Prozent der Monate verlieren Sie typischerweise X €.“

Stolperfallen. Die Wurzel-Zeit-Regel bei CVaR (Kapitel 20) — nicht blind anwenden. Korrelationen sind in Krisen andere als im Mittel: Stresstest mit erhöhten Korrelationen rechnen.

P9 — Wenn der Solver aussteigt: Tourenplanung in Echtgröße

Schwierigkeit: □□□ · **Kapitel:** Kapitel 8, Kapitel 9 · **Zeit:** ca. 18 Stunden

Ausgangslage. P3 hat funktioniert — mit 30 Kunden. Der Auftraggeber kommt mit 400 zurück, und derselbe Code liefert nach zwanzig Minuten noch keine Lösung. Das ist keine Panne, sondern der erwartbare Umschlagpunkt.

Modellskizze. Zwei Verfahren am **selben** Problem, verglichen unter **gleichem Zeitbudget**: * Exakt: MILP oder CP-SAT wie in P3, mit Zeitlimit und protokolliertem Gap. * Heuristisch: Startlösung mit einer Faustregel, dann Verbesserung durch Simulated Annealing oder Large Neighborhood Search (Kapitel 9). * Messgröße ist nicht „wer gewinnt”, sondern **ab welcher Instanzgröße** sich das Blatt wendet.

Abnahmekriterien. - [ ] Eine Tabelle über mindestens vier Instanzgrößen: exakt gegen heuristisch, jeweils Zielwert und Laufzeit. - [ ] Der Umschlagpunkt ist auf ±50 Kunden eingegrenzt und **benannt**. - [ ] Bei kleinen Instanzen wird gegen die bewiesen optimale Lösung geprüft — sonst weiß niemand, wie gut die Heuristik wirklich ist. - [ ] Fester Seed, reproduzierbarer Lauf; das Zugbudget ist fix, nicht die Uhrzeit.

□ **Die Falle, in die hier fast jeder tappt:** die Heuristik nur auf großen Instanzen zu testen, wo man das Optimum nicht kennt. Dann sieht jede Lösung gut aus.

P10 — Zwei Ziele, eine Entscheidung: Kosten gegen CO□

Schwierigkeit: □□ · **Kapitel:** Kapitel 6, Kapitel 14 · **Zeit:** ca. 14 Stunden

Ausgangslage. Die Geschäftsführung will „günstiger **und** grüner". Beides zugleich gibt es nicht — und die übliche Antwort, beide Ziele mit Gewichten zu verrechnen, verdeckt genau die Frage, um die es geht.

Modellskizze. Ein Transport-, Beschaffungs- oder Produktionsmodell mit zwei Zielen: * Erst beide Ziele **einzeln** optimieren — das gibt die Eckpunkte und damit den Rahmen. * Dann die Front über das ε-Constraint-Verfahren abfahren (Kapitel 14): ein Ziel minimieren, das andere als Nebenbedingung schrittweise verschärfen. * Ergebnis ist **keine Lösung, sondern eine Kurve** — plus die Angabe, was jeder eingesparte Kilogramm CO□ an Mehrkosten bedeutet.

Abnahmekriterien. - [ ] Die Pareto-Front ist berechnet und gezeichnet, nicht nur beschrieben. - [ ] Für mindestens zwei Punkte ist der Aufpreis je eingesparter Einheit ausgerechnet. - [ ] Es ist geprüft und dokumentiert, welche Punkte der Front eine **gewichtete Summe niemals** finden würde — und warum. - [ ] Die Entscheidungsvorlage nennt drei Punkte zur Auswahl, nicht dreißig.

P11 — Vom Skript zum Dienst: das Modell übergeben

Schwierigkeit: □□□ · **Kapitel:** Kapitel 22, Kapitel 23 · **Zeit:** ca. 16 Stunden

Ausgangslage. Ihr Modell rechnet. Jetzt soll es jemand anderes benutzen — jemand, der weder Python noch Ihr Notebook kennt, und der es auch dann noch braucht, wenn Sie im Urlaub sind. Nehmen Sie **eines Ihrer eigenen** Projekte P1–P10 als Grundlage; dies ist kein neues Modell, sondern der Schritt danach.

Modellskizze. Vier Schichten, jede einzeln abnehmbar: * **Domäne:** Eingabedaten als validierte Objekte (Pydantic), nicht als lose Dictionaries — ein falscher Wert soll beim Einlesen auffallen, nicht im Solver (Kapitel 22). * **Kern:** Modellaufbau und Lösung hinter einer Funktion, die ein `Loesung`-Objekt zurückgibt und **alle** Statusfälle behandelt. * **Tests:** eine Suite, die auch die Fälle abdeckt, in denen es *keine* Lösung gibt (Kapitel 23). * **Schnittstelle:** ein HTTP-Dienst, der ein JSON entgegennimmt und eines zurückgibt.

Abnahmekriterien. - [ ] Eine ungültige Eingabe wird **vor** dem Solver abgewiesen, mit einer Meldung, die den Fehler benennt. - [ ] Die Testsuite läuft grün und enthält mindestens einen INFEASIBLE-Fall. - [ ] Ein Mutationstest zeigt, dass die Tests eine absichtlich eingebaute Modelländerung auch wirklich bemerken. - [ ] Der Dienst antwortet auf eine Beispielanfrage per `curl` — dokumentiert samt Aufruf. - [ ] Es gibt eine Betriebsseite: Was tun, wenn der Solver `INFEASIBLE` meldet? Der Notfallplan gehört dazu, nicht die Fehlermeldung (Anhang C).

□ **Der Prüfstein** Geben Sie das Projekt einer Person, die nicht dabei war, zusammen mit dem README — und sagen Sie nichts. Was sie nicht allein zum Laufen bringt, ist nicht fertig.

P8 — Freies Projekt aus Ihrem eigenen Umfeld

Schwierigkeit: offen · **Zeit:** 10–25 Stunden

Der Auftrag. Nehmen Sie die Notiz aus der Aufgabe *Eigenes Problem zerlegen* (Kapitel 1) hervor — das Problem aus Ihrem eigenen Alltag oder Beruf. Bauen Sie es.

Bewährte Kandidaten aus früheren Kursen:

Problem	Verfahren	Kapitel
Sitzordnung für eine Hochzeit (Sympathien/Antipathien)	CP-SAT	Kapitel 7
Trainingsplan eines Sportvereins (Hallen, Trainer, Altersgruppen)	CP-SAT	Kapitel 7
Budgetaufteilung auf Projekte mit Abhängigkeiten	MILP	Kapitel 6
Schnittoptimierung für Zuschnitt (Holz, Blech, Stoff)	MILP / Spaltengenerierung	Kapitel 10
Speiseplan unter Nährwert- und Budgetgrenzen	LP	Kapitel 5
Prüfungsplanung (keine Kollisionen, Erholungspausen)	CP-SAT	Kapitel 7
Ladeplanung für E-Fahrzeugflotte (Lastspitzen vermeiden)	MILP + Cumulative	Kapitel 6, Kapitel 7
Bewässerungsplan im Kleingarten (Wasser, Wetterprognose)	Stochastisch	Kapitel 12

Abnahmekriterien (für jedes freie Projekt). - [ ] Die Bausteine-Vorlage ist ausgefüllt und liegt dem Projekt bei. - [ ] Eine Kleinstinstanz ist von Hand nachgerechnet und stimmt mit dem Solver überein. - [ ] Alle Nebenbedingungen werden nach dem Lösen per `assert` geprüft. - [ ] Es gibt einen Vergleich gegen die bisherige (manuelle) Lösung — mit Zahlen. - [ ] Das Ergebnis ist erklärbar: Warum diese Lösung, warum nicht die naheliegende Alternative, was würde sie verbessern? - [ ] Die Grenzen des Modells sind dokumentiert: Was bildet es **nicht** ab?

Eine Bitte zum Schluss

Wenn Sie ein Projekt fertig haben, machen Sie zwei Dinge:

Erstens: Zeigen Sie es jemandem, der nichts von Optimierung versteht. Wenn diese Person nach fünf Minuten sagen kann, was das Programm tut und warum sie dem Ergebnis trauen sollte, haben Sie es richtig gemacht. Wenn nicht, fehlt die Erklärbarkeit — nicht das Verständnis Ihres Gegenübers.

Zweitens: Schreiben Sie auf, was schiefgegangen ist. Jeder, der optimiert, produziert Fehler wie die in diesem Buch besprochenen (siehe z. B. Abschnitt 5.7 oder Anhang C). Wer sie dokumentiert, macht sie kein zweites Mal — und hilft dem Nächsten.

Viel Erfolg.

Anhang A: Lösungen zu allen Übungsaufgaben

> **Wie Sie diesen Anhang benutzen:** Erst rechnen, dann nachsehen. Bei Programmieraufgaben (□□□) ist meist der **Lösungsweg** mit den entscheidenden Codezeilen und den erwarteten Erkenntnissen angegeben, nicht das vollständige Programm — den Rest sollen Sie selbst bauen. Wo Zahlen von Live-Daten abhängen, steht die *Struktur* der erwarteten Antwort.

A.1 Lösungen zu Kapitel „Einführung in Operations Research — Vom Ursprung zur mathematischen Entscheidungsfindung“

1.1 — Analytik-Stufen. (a) deskriptiv (Vergangenheit beschreiben). (b) prädiktiv (Prognose). (c) **präskriptiv** (Handlungsanweisung unter Restriktionen). (d) prädiktiv (Klassifikation). (e) **präskriptiv** — hier kommt das begrenzte Budget ins Spiel, es muss zugeteilt werden. Beachten Sie das Paar (d)/(e): Die Prognose sagt, *wer* kündigen wird; die Optimierung sagt, *wen* man mit dem vorhandenen Geld halten kann.

1.2 — Hart oder weich? (a) **hart** — gesetzlich zwingend. (b) **weich** — Wunsch, mit Strafkosten. (c) **hart** — Patientensicherheit, rechtlich vorgeschrieben. (d) **weich** — Fairnessziel, über Strafterme. (e) **hart**, falls tariflich/gesetzlich fixiert, sonst weich mit sehr hoher Strafe. Faustregel: Hart ist nur, was rechtlich oder physikalisch unmöglich zu verletzen ist.

1.3 — Kombinatorik. (a) $12! = 479\,001\,600$. (b) $479\,001\,600 / 5 \cdot 10^6 \approx 95{,}8$ Sekunden $\approx 1{,}6$ Minuten. (c) $13! = 6\,227\,020\,800$; das sind 1245 s $\approx 20{,}8$ Minuten — **Faktor 13**. Jeder weitere Auftrag multipliziert die Zeit mit der neuen Anzahl. (d) Eine Stunde $= 3600$ s $\rightarrow 1{,}8 \cdot 10^{10}$ Prüfungen. $13! = 6{,}2 \cdot 10^9$ □, $14! = 8{,}7 \cdot 10^{10}$ □. Also **13 Aufträge**.

1.4 — Modell lesen. (a) Variablen $x_1, x_2 \geq 0$; Parameter $(3, 5)$ und die Kapazitäten $(4, 12, 18)$; Zielfunktion $\max 3x_1 + 5x_2$; vier Nebenbedingungen inkl. Nichtnegativität. (b) $(2, 6)$: $2 \leq 4$ □, $12 \leq 12$ □, $6 + 12 = 18 \leq 18$ □ → zulässig, $Z = 36$. $(4, 3)$: $4 \leq 4$ □, $6 \leq 12$ □, $12 + 6 = 18 \leq 18$ □ → zulässig, $Z = 27$. (c) Beste ganzzahlige Lösung ist $(2, 6)$ mit $Z = 36$ — sie ist hier bereits ganzzahlig, weil die Ecke des Polyeders zufällig auf Gitterpunkten liegt.

1.5 — Bäckerei.

```
from ortools.sat.python import cp_model
m = cp_model.CpModel()
x1 = m.NewIntVar(0, 1000, "Brote")
x2 = m.NewIntVar(0, 1000, "Broetchen_10er")
m.Add(5 * x1 + 6 * x2 <= 900)        # Mehl in 100-g-Einheiten (0,5 kg -> 5)
m.Add(4 * x1 + 3 * x2 <= 600)        # Ofenminuten
m.Add(x1 >= 40)                       # Vertrag
m.Maximize(250 * x1 + 300 * x2)       # Deckungsbeitrag in Cent
s = cp_model.CpSolver(); s.Solve(m)
```

Ergebnis: $x_1 = 96$ Brote, $x_2 = 70$ Zehnerpackungen, Deckungsbeitrag **450,00 €**.

Probe: Mehl $0{,}5 \cdot 96 + 0{,}6 \cdot 70 = 48 + 42 = 90$ kg □ (voll ausgelastet); Ofen $4 \cdot 96 + 3 \cdot 70 = 384 + 210 = 594 \le 600$ □; Vertrag $96 \ge 40$ □; $Z = 2{,}5 \cdot 96 + 3 \cdot 70 = 240 + 210 = 450$ □.

Zwei lehrreiche Beobachtungen: 1. Die **Vertragsbedingung** $x_1 \ge 40$ **ist nicht bindend** — der Solver backt freiwillig 96 Brote. Wer aus der Aufgabenstellung schließt, die Mindestmenge werde „gerade so“ erfüllt, liegt falsch. 2. Das **kontinuierliche** Optimum $(40;\ 116{,}67)$ liefert **denselben** Zielwert 450. Das Problem hat also mehrere optimale Lösungen — die Zielfunktion verläuft parallel zur Mehlrestriktion ($2{,}5/0{,}5 = 5 = 3{,}0/0{,}6$). Prüfen Sie das nach: Genau deshalb ist hier auch die ganzzahlige Lösung ohne Verlust erreichbar.

1.6 — Sensitivität durch Ausprobieren. Schleife über `RAM_GESAMT in range(54, 73, 2)`, jeweils Modell neu lösen. (a) Ab 62 GB steigt der Gewinn nicht mehr — dann bindet die **vCPU**-Grenze, RAM ist nicht mehr der Engpass. (b) Der Zuwachs je GB ist der **Schattenpreis** (Kapitel 5). Solange RAM bindet, liegt er bei 31,25 €/GB; danach fällt er auf 0.

1.7 — Eigenes Problem. Individuell. Prüfkriterien: Sind die Variablen wirklich *entscheidbar* (nicht bereits festgelegt)? Hat die Zielfunktion eine **Einheit**? Ist jede harte Bedingung wirklich unverhandelbar?

Finde den Denkfehler — Die Schreinerei verdoppelt ihren Gewinn

(a) **Nachrechnen.** Der Plan lautet 40 Tische und 150 Stühle: $40 \cdot 3 + 150 \cdot 1 = \mathbf{270}$ Montagestunden bei 150 verfügbaren, und $40 \cdot 6 + 150 \cdot 1 = \mathbf{390}$ m² Material bei 240 verfügbaren. Beide Vorräte sind fast doppelt überzogen — der Plan ist in der Werkstatt nicht ausführbar.

(b) **Der Fehler.** Die Nebenbedingungen stehen **innerhalb** der Produktschleife. Dadurch entsteht je Produkt eine eigene Kapazitätsgrenze („der Tisch allein darf 150 Stunden verbrauchen“, „der Stuhl allein darf 150 Stunden verbrauchen“) statt einer **gemeinsamen** Grenze über alle Produkte hinweg. Die Ressource wird so für jedes Produkt neu verteilt — im Modell existiert sie mehrfach.

Übersetzt in die Formelsprache aus Abschnitt 1.6: Programmiert wurde $a_{ij} x_j \le b_i$ für jedes j einzeln, gemeint war $\sum_j a_{ij} x_j \le b_i$. Das Summenzeichen ist der ganze Unterschied.

(c) **Korrektur.** Die Summe über alle Produkte gehört *in* die Bedingung, die Schleife läuft über die **Ressourcen**, nicht über die Produkte:

```
for r, index in [("Montagestunden", 1), ("Plattenmaterial", 2)]:
    s.Add(sum(x[p] * produkt[p][index] for p in produkt) <= vorrat[r])
```

(d) **Automatische Absicherung.** Nach jedem Lösen den tatsächlichen Verbrauch gegen den Vorrat prüfen — mit kleiner Toleranz, weil Solver mit endlicher Genauigkeit rechnen (siehe Kapitel 3):

```
for r, index in [("Montagestunden", 1), ("Plattenmaterial", 2)]:
    verbrauch = sum(x[p].solution_value() * produkt[p][index] for p in produkt)
    assert verbrauch <= vorrat[r] + 1e-6, f"{r} ueberzogen: {verbrauch} > {vorrat[r]}"
```

Diese vier Zeilen sind das wichtigste Werkzeug des Kapitels: Sie prüfen die **Lösung gegen die Wirklichkeit**, nicht gegen das Modell. Ein Modellierungsfehler kann sich vor dem Solver verstecken, vor dieser Prüfung nicht.

Micro-Quiz

1 — (c) präskriptiv. Die Frage lautet „was tun“, und es gibt eine Restriktion (das Restbudget). Dass man für die Bewertung eine Prognose braucht, macht die Frage nicht prädiktiv — die Prognose ist hier **Eingabeparameter**, nicht Ergebnis.

2 — (a) als weiche Bedingung mit Strafkosten. `INFEASIBLE` heißt: Die harten Regeln widersprechen sich, es existiert kein einziger zulässiger Plan. Das ist eine **Modellierungs-**, keine Rechenfrage — ein schnellerer Solver (b) rechnet dieselbe Unlösbarkeit nur schneller nach. Antwort (c) ginge am Problem vorbei, weil nicht die Variablengrenzen, sondern eine Wunschregel den Widerspruch erzeugt.

3 — (b). $20! \approx 2{,}4 \cdot 10^{18}$; Faktor 1000 verkürzt 77 Jahre auf knapp einen Monat — und bei 24 Aufträgen ist man wieder bei Jahrtausenden. Gegen multiplikatives Wachstum ist konstante Beschleunigung machtlos. (a) und (c) sind sachlich falsch: Solver nutzen sehr wohl mehrere Kerne, und die Ungenauigkeit großer Fakultäten ist nicht der Grund für die Rechenzeit.

Selbsttest

1. Über Entscheidungsvariablen bestimmt der Solver; Parameter sind vorgegebene Daten.
2. Weil die Zahl der Kombinationen **multiplikativ** wächst: Faktor 10 Geschwindigkeit verschiebt die machbare Größe nur um wenige Elemente.
3. Entscheidungsvariablen, Parameter, Zielfunktion, Nebenbedingungen.
4. Kein Punkt erfüllt alle Bedingungen gleichzeitig; häufigste Ursache: zu viele oder widersprüchliche **harte** Bedingungen.
5. Weil sie dann nicht in der Modellformulierung sichtbar ist, kein Schattenpreis abgefragt werden kann — und weil eine zu eng gewählte Grenze stillschweigend ein falsches Optimum erzeugt.

A.2 Lösungen zu Kapitel „Das mathematische Fundament — Vektoren, Matrizen, Konvexität“

2.1 — Matrixform lesen. $\max 4x_1 + x_2 + 6x_3$ u. d. N. $x_1 + 2x_2 \le 10$, $x_2 + 3x_3 \le 12$, $x \ge 0$. **3 Variablen, 2 Nebenbedingungen** (plus Nichtnegativität).

2.2 — Standardform. $\min -7x_1 + 2x_2$ u. d. N. $-4x_1 - x_2 \le -20$ (aus „$\ge$“ durch Multiplikation mit -1), $x_1 - x_2 \le 3$ **und** $-x_1 + x_2 \le -3$ (Gleichung als zwei Ungleichungen), $x_1, x_2 \ge 0$.

2.3 — Ecken von Hand. (b) Ecken: $(0,0)$, $(6,0)$, $(4,4)$, $(0,8)$. (c) Z: 0, 12, 20, **24** → Optimum $(0,8)$ mit $Z = 24$. (d) Bei $\max 2x_1 + 2x_2$: $Z(4,4) = 16$, $Z(0,8) = 16$ — die Zielfunktion ist **parallel zur Kante** $x_1 + x_2 = 8$. Es gibt dann **unendlich viele** optimale Lösungen (die ganze Kante), aber weiterhin mindestens eine in einer Ecke — der Fundamentalsatz bleibt gültig.

2.4 — Konvexität. (a) konvex (linear, sogar affin — Grenzfall, auch konkav). (b) konvex ($f'' = 12x^2 \ge 0$). (c) **nicht** konvex, sondern **konkav** ($f'' = -\frac{1}{4}x^{-3/2} < 0$). (d) **nicht** konvex, konkav ($f'' = -1/x^2 < 0$). (e) konvex (Summe konvexer Funktionen; Hesse-Matrix $2\mathbf{I} \succ 0$). (f) konvexe Menge (Kreisscheibe). (g) **konvexe Menge** — der Bereich oberhalb der Hyperbel im positiven Quadranten ist konvex (Achtung, überraschend: die *Funktion* $1/x$ ist konvex, und die Menge $\{x_2 \ge 1/x_1\}$ ist der Epigraph einer konvexen Funktion, also konvex).

2.5 — Positive Semidefinitheit. $\mathbf{P}_1$: Eigenwerte $\approx (3{,}8;\ 9{,}2)$ → PSD □, Korrelation $1/\sqrt{4 \cdot 9} = 0{,}167$ □. $\mathbf{P}_2$: Eigenwerte $\approx (-0{,}6;\ 13{,}6)$ → **nicht PSD**; implizierte Korrelation $7/\sqrt{36} = 1{,}167 > 1$ — unmöglich. $\mathbf{P}_3$: Eigenwerte $(0{,}5;\ 0{,}5;\ 2{,}0)$ → PSD □.

2.6 — Bäckerei visualisieren. Ecken: $(40,0)$, $(150,0)$, $(40, 116{,}67)$ und der Schnittpunkt von Mehl- und Ofengrenze. Beste Ecke ist $(40;\ 116{,}67)$ mit $Z = 450$ (im kontinuierlichen Fall).

2.7 — Eigenwert-Clipping.

```python
def repariere(P):
    lam, V = np.linalg.eigh(P)
    return V @ np.diag(np.maximum(lam, 0.0)) @ V.T
```

Für $\mathbf{P}_2$ ergibt sich eine PSD-Matrix mit Korrelation exakt 1,0 — das Verfahren zieht die unmögliche Korrelation auf den nächstgelegenen zulässigen Wert. Für Kovarianzmatrizen setzt man in der Praxis auf einen kleinen positiven Wert statt auf 0 (`np.maximum(lam, 1e-10)`), damit die Matrix invertierbar bleibt.

Finde den Denkfehler — Die unauffällige Transposition

(a) **Warum nichts auffällt. A** ist quadratisch, also passen die Dimensionen auch transponiert. NumPy prüft Formen, nicht Bedeutungen; der Solver bekommt ein vollkommen zulässiges LP vorgelegt und löst es korrekt — nur eben ein **anderes**. Eine Transposition ist genau dann gefährlich, wenn sie folgenlos *aussieht*: Wäre die Matrix 2×3 gewesen, hätte NumPy sofort einen Dimensionsfehler geworfen.

(b) **Der Plan an den echten Restriktionen.** Mit $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$ und $\mathbf{x} = (5{,}6;\ 0{,}8)$:

$$\mathbf{Ax} = \begin{pmatrix} 1 \cdot 5{,}6 + 2 \cdot 0{,}8 \\ 3 \cdot 5{,}6 + 1 \cdot 0{,}8 \end{pmatrix} = \begin{pmatrix} 7{,}2 \\ 17{,}6 \end{pmatrix} \quad \text{gegen} \quad \mathbf{b} = \begin{pmatrix} 8 \\ 12 \end{pmatrix}$$

Die zweite Ressource ist um **47 % überzogen** (17,6 statt 12). Der Plan ist in der Werkstatt nicht ausführbar.

(c) **Warum „zu klein“ schlimmer ist als „zu groß“.** Ein unerwartet *hoher* Zielwert weckt Misstrauen — man rechnet nach (siehe Abschnitt 1.10). Ein leicht *niedrigerer* Wert wirkt dagegen wie ein normales, etwas enttäuschendes Ergebnis: Niemand prüft nach, warum die Optimierung „nur“ 20,80 € statt der erhofften 22 € bringt. Fehler, die sich als Bescheidenheit tarnen, überleben am längsten.

(d) **Zwei Zeilen, die es aufgedeckt hätten.** Erstens beim Einlesen die Form gegen die Bedeutung prüfen, zweitens nach dem Lösen den Verbrauch gegen den Vorrat:

```
assert A.shape == (len(b), len(c)), "A: Zeilen = Ressourcen, Spalten = Variablen"
assert np.all(A @ r.x <= b + 1e-9), f"Plan verletzt Restriktionen: {A @ r.x} > {b}"
```

Die erste Zeile hilft nur bei rechteckigem **A** — die zweite **immer**. Sie ist die wichtigste Zeile in jedem Optimierungsskript: Sie prüft die Lösung nicht gegen das Modell, sondern gegen die Wirklichkeit, die das Modell abbilden sollte.

Vorbeugend hilft außerdem, Matrizen nie positionsweise abzutippen, sondern über benannte Spalten aus einer Tabelle zu erzeugen — genau das tut `Excel_Bruecke.py` aus Kapitel 1.

Micro-Quiz

1 — (b). Der Fundamentalsatz sagt nur, dass ein Optimum *in einer Ecke angenommen wird*. (a) ist falsch, weil es mehrere optimale Ecken geben kann (wenn die Zielfunktion parallel zu einer Kante verläuft) und dann sogar unendlich viele optimale Punkte auf der Verbindungsstrecke. (c) ist falsch, weil die Zahl der Eckenkandidaten kombinatorisch wächst: Bei 6 Variablen und 4 Ungleichungen plus 6 Nichtnegativitäten sind $\binom{10}{6} = 210$ Systeme zu prüfen — bei 50 Variablen wären es 10^{29}. Der Satz sagt, *wo* man suchen muss, nicht dass die Suche billig ist.

2 — (b). Faustregel $\kappa = 10^k$ ☐ etwa k signifikante Stellen verloren; bei 10^{11} bleiben von 16 rund 5. Das Modell ist deshalb nicht unlösbar (a) — es rechnet nur mit einer Genauigkeit, die den Ergebnissen nicht mehr anzusehen ist. (c) verwechselt die Konditionszahl mit einem Aufwandsmaß; sie sagt nichts über die Iterationszahl.

3 — (c). `int()` schneidet ab: `int(0.99999998) == 0` macht aus einer Ja- eine Nein-Entscheidung (a). Den Wert unverändert weiterzureichen (b) verschiebt das Problem nur in die nachgelagerte Verarbeitung, wo dann irgendwann doch jemand `int()` schreibt. Richtig ist die Prüfung gegen die Toleranz mit Fehlermeldung im Zweifelsfall — die Funktion `sichere_ganzzahl()` aus `Skalierung_Kondition.py`.

Selbsttest

1. Die gewichtete Summe aller Variablen — Ergebnis ist ein **Skalar** (eine Zahl).
2. Weil die Matrix rechteckig ist: Jede Zeile muss so viele Einträge haben wie es Variablen gibt. Eine 0 bedeutet „diese Variable verbraucht nichts von dieser Ressource“.
3. Hat ein LP eine Optimallösung, liegt mindestens eine in einer Ecke. Nutzen: endlich viele Kandidaten statt unendlich vieler Punkte — die Grundlage des Simplex.
4. Weil der zulässige Bereich zu einem **Punktgitter** wird; zwischen zwei zulässigen Gitterpunkten liegen unzulässige Punkte, die Verbindungsstrecke verlässt also die Menge.
5. Das ist kein Bug, sondern das erwartete Verhalten bei **nicht-konvexen** Problemen: lokale Verfahren finden lokale Optima. Abhilfe: Multistart, konvexe Reformulierung oder globale Verfahren.

A.3 Lösungen zu Kapitel „Das Python-Ökosystem für OR — Solver, Bindings und Modellierungsschichten“

3.1 — Solverwahl. (a) CP-SAT (diskrete Zuordnung mit Zeitfenstern). (b) `scipy.optimize.linprog` (klassisches Mischungs-LP, klein). (c) CVXPY (konvexes QP). (d) MIQP — CVXPY mit Binärvariablen und MIQP-fähigem Solver, oder Heuristik. (e) MILP über `highspy` oder CP-SAT (Standortproblem mit Fixkosten). (f) `scipy.optimize.minimize` mit Multistart (nicht konvex).

3.2 — Bäckerei viermal. Alle vier müssen $Z = 448$ (ganzzahlig) bzw. 450 (kontinuierlich) liefern. Denken Sie an die Prozesstrennung, falls `ortools` und `highspy` kollidieren.

3.3 — CSR-Format. `values = [3, 1, 2, 5, 4, 6]`, `indices = [0, 3, 2, 0, 1, 3]`, `starts = [0, 2, 3]`. CSR speichert 6 Werte + 6 Indizes + 3 Startpositionen = 15 Zahlen; die volle Matrix hätte $3 \times 4 = 12$. **Bei dieser winzigen, dicht besetzten Matrix lohnt CSR nicht** — der Vorteil entsteht erst bei großer, dünn besetzter Struktur (z. B. 1000×1000 mit 0,5 % Besetzung: 15 000 statt 1 000 000 Zahlen).

3.4 — Konvexitätsprüfung. $\min x^3$ wirft `DCPError: Problem does not follow DCP rules`, weil x^3 auf $[-2, 2]$ weder konvex noch konkav ist. $\min x^2$ läuft und liefert $x = 0$. Der Unterschied: CVXPY akzeptiert nur Ausdrücke, deren Konvexität es **beweisen** kann — dafür garantiert es das globale Optimum.

3.5 — Laufzeitvergleich. Erwartetes Muster: `linprog` und `highspy` liegen bei kleinen Modellen gleichauf (Overhead dominiert); ab etwa $n \gtrsim 500$ zieht `highspy` davon, weil der Modellaufbau effizienter ist. CVXPY hat den größten festen Aufwand (Ausdrucksbaum-Kompilierung), der bei wiederholten Läufen mit `cp.Parameter` teilweise entfällt.

3.6 — Eigene Entscheidungshilfe. Ergänzungen: Bei kommerzieller Lizenz Gurobi/CPLEX über Pyomo oder CVXPY empfehlen; bei Lesbarkeitsanforderung Pyomo (algebraische Notation, Trennung von Daten und Modell).

Finde den Denkfehler — Der Solver, der angeblich dreimal schneller ist

(a) **Was da alles mitgemessen wird.** Die Stoppuhr läuft ab der ersten Zeile, also mindestens über drei Dinge, die mit Lösegeschwindigkeit nichts zu tun haben:

1. **Der `import`.** CVXPY zieht beim ersten Import seine gesamte Solver-Erkennung hoch — allein das kostet regelmäßig über eine Sekunde. `linprog` steht in einem Prozess, der SciPy ohnehin schon geladen hat, praktisch sofort bereit.
2. **Der Modellaufbau.** CVXPY baut einen Ausdrucksbaum und kompiliert ihn in die Standardform des Solvers. Das ist echter Aufwand — aber Aufbau, nicht Rechnen.
3. **Die Reihenfolge.** CVXPY läuft zuerst und bezahlt dabei alles, was danach im Betriebssystem-Cache liegt: Bibliotheken, Speicherseiten, JIT-Wärme. Tauschen Sie die beiden Blöcke, und die Zahlen verschieben sich allein deshalb.

(b) **Warum das bei 60 Wiederholungen kippt.** Import und Kompilierung fallen **einmal** an, das Lösen **sechzigmal**. Genau dafür hat CVXPY `cp.Parameter`: Man baut das Problem einmal, tauscht nur die Daten und löst erneut, ohne neu zu kompilieren. Gemessen wurde also ausgerechnet der Teil, der in der Zielanwendung fast nicht mehr vorkommt — die Entscheidung beruht auf der unwichtigsten Zahl.

(c) **Eine Messung, die trägt.** Drei Regeln:

```
# 1. Importe VOR die Messung
import cvxpy as cp
from scipy.optimize import linprog

# 2. Aufwaermlauf: einmal alles durchlaufen lassen, Ergebnis verwerfen
loese_klein()

# 3. Aufbau und Loesen GETRENNT messen, ueber mehrere Wiederholungen
t0 = time.perf_counter()
problem = cp.Problem(cp.Minimize(kosten @ x), [A @ x == b])   # Aufbau
t_aufbau = time.perf_counter() - t0

zeiten = []
for _ in range(10):
    t0 = time.perf_counter()
    problem.solve()
    zeiten.append(time.perf_counter() - t0)
print(f"Aufbau {t_aufbau:.3f}s, Loesen Median {statistics.median(zeiten):.3f}s")
```

Und die vierte, wichtigste Regel: **die Größe messen, die später auch läuft.** Ein Vergleich bei 200 Variablen sagt nichts über das Verhalten bei 200 000 — die Kurven schneiden sich oft. Genau diese Trennung führt `Vektorisierte_Modellgenerierung.py` vor, und Kapitel 22 baut daraus eine vollständige Benchmark-Pipeline.

Micro-Quiz

1 — (b) CP-SAT. Es geht um diskrete Zuweisung mit logischen Regeln („nicht gleichzeitig“) — CP-SATs Kerngebiet, und Regeln dieser Art formuliert man dort direkt statt über Big-M-Umwege. (a) scheidet aus, weil CVXPY keine sinnvolle Ganzzahligkeit in dieser Größenordnung bietet; (c) findet bei einem diskreten Problem bestenfalls ein lokales Optimum und hat keinerlei Handhabe für „entweder–oder“-Regeln.

2 — (c) Modellaufbau vektorisieren. 32 von 40 Sekunden fallen an, *bevor* der Solver startet. Ein kommerzieller Solver (a) beschleunigt bestenfalls die verbleibenden 8 Sekunden — selbst bei Faktor 4 gewinnen Sie 6 der 40 Sekunden. Das Zeitlimit (b) betrifft den Abbruch, nicht die Geschwindigkeit. Die Regel dahinter: **erst messen, wo die Zeit hingeht, dann optimieren.**

3 — (b). Alle vier sind Modellierungsschichten über HiGHS. Deshalb liefern sie denselben Zielwert (bis auf Toleranz) und unterscheiden sich in der reinen Rechenzeit kaum — wohl aber im Aufbauaufwand und in der Lesbarkeit. (a) ist falsch: Niemand von ihnen implementiert den Simplex in Python, das wäre um Größenordnungen zu langsam. (c) ist falsch: Nicht-konvexe Probleme global zu lösen kann keines dieser Werkzeuge — dafür braucht es spezialisierte globale Solver (Kapitel 11).

Selbsttest

1. Modellierungsschicht (Python) und Solver-Schicht (C++). Die Trennung erlaubt Solvertausch ohne Modelländerung.
2. CVXPY prüft die Konvexität und lehnt ab, was es nicht garantieren kann; `minimize` prüft nichts und liefert ein lokales Optimum.
3. Werte, Spaltenindizes und Zeilenstartpositionen der Nicht-Null-Einträge. Entscheidend, weil reale Modelle extrem dünn besetzt sind.
4. Wegen des Kompilierungsaufwands des Ausdrucksbaums. Kein Argument dagegen, weil dieser Aufwand einmalig ist, die Lesbarkeit hoch und die Konvexitätsprüfung wertvoll.
5. Bei rein kontinuierlichen Problemen (CP-SAT kennt nur ganze Zahlen) und bei reinen Routing-Problemen (dort ist die Routing-Bibliothek überlegen).

A.4 Lösungen zu Kapitel „Vom Management-Wunsch zum Modell“

4.1 — Die fehlende Lesart. Eine dritte Lesart: **„Der Umsatzanteil der Stammkunden darf nicht unter 30 % fallen.”** Sie zielt weder auf einzelne Aufträge noch auf einzelne Kunden, sondern auf das Gesamtbild. Sie beschreibt einen Betrieb, dem die *Struktur* seines Geschäfts wichtig ist — etwa weil Stammkunden verlässlicher zahlen oder weil eine zu große Abhängigkeit von Neukunden als Risiko gilt. Sie lässt zu, dass ein einzelner Stammkunde in einem Quartal leer ausgeht, solange die Summe stimmt.

Der Punkt der Aufgabe: Alle drei Lesarten sind vertretbar, und keine ist aus dem Satz ableitbar. Die Wahl trifft man — im Zweifel unbewusst.

4.2 — Frage 5 anwenden. Wird die *durchschnittliche* Wartezeit minimiert, lohnt es sich, viele leichte Fälle schnell abzuarbeiten und die aufwendigen nach hinten zu schieben: Ein Fall mit vier Stunden Wartezeit zählt genauso viel wie vier Fälle mit einer Stunde. Der optimale Plan behandelt also bevorzugt Bagatellen und lässt schwere Fälle warten — medizinisch das Gegenteil des Gewollten.

Das ist genau der Ertrag von Frage 5: Der Plan lässt sich *ablehnen*, und aus der Ablehnung folgt die richtige Zielgröße — etwa die maximale Wartezeit je Dringlichkeitsstufe, oder die Einhaltung von Zielzeiten je Triage-Kategorie.

4.3 — Der Kompromiss dazwischen. Je Stammkunde eine Nebenbedingung: Die Summe der zugeteilten Stunden dieses Kunden muss mindestens die Hälfte seiner angefragten Stunden betragen.

```
for kunde in STAMMKUNDEN:
    stunden_kunde = [a[3] if a[1] == kunde else 0 for a in AUFTRAEGE]
    je_kunde_ub.append([-s for s in stunden_kunde])
    je_kunde_b.append(-0.5 * sum(stunden_kunde))
```

Erwartung: Das Ergebnis liegt zwischen Lesart A und B — die Regel bindet stärker als „ein Auftrag genügt”, aber schwächer als „alle Aufträge”. Der Lehrpunkt ist nicht die Zahl, sondern dass sich zwischen zwei Lesarten beliebig fein abstufen lässt, sobald man sie einmal als Ungleichung geschrieben hat.

4.4 — Zwei Ziele gleichzeitig. Bei $\lambda = 0$ ergibt sich der DB-Plan (78 500 € DB), bei großem λ der Umsatzplan (66 000 € DB). Dazwischen kippt es an genau einer Stelle — und zwar **sprunghaft**, nicht allmählich: Weil die Entscheidungsvariablen binär sind, gibt es nur endlich viele Pläne, und der Wechsel erfolgt, sobald ein anderer Plan den höheren gewichteten Wert hat.

Der Kippwert sagt dem Betrieb etwas Konkretes: *„Erst wenn Ihnen ein Euro Umsatz mehr wert ist als λ Euro Deckungsbeitrag, ändert sich Ihr Plan.”* Das ist eine Frage, die ein Kaufmann beantworten kann — anders als „welche Gewichtung hätten Sie gern?“.

4.5 — Übersetzen, wo es nicht eindeutig ist. (a) Drei Lesarten von „gleichmäßig ausgelastet”: **(1) Spannweite minimieren** — $\max_i a_i - \min_i a_i$. Bestraft Ausreißer nach beiden Seiten; ein einzelner Untätiger ist genauso schlimm wie ein Überlasteter. **(2) Den Schlechtestgestellten verbessern** — $\max_i a_i$ minimieren (B11). Interessiert sich nicht dafür, wie ungleich die übrigen liegen, solange die Spitze sinkt. **(3) Abweichung vom Mittel minimieren** — $\sum_i |a_i - \bar{a}|$ (B10). Verteilt die Last am gleichmäßigsten, kann aber die Spitze höher lassen als (2). Wo es sichtbar wird: Bei fünf Monteuren mit 8, 8, 8, 8 und 16 Stunden senkt (2) die 16 zuerst; bei 4, 8, 8, 8, 12 tut (1) am meisten gegen die Spreizung, während (2) kaum etwas findet. Und wenn ein Monteur aus guten Gründen nur halbtags da ist, bestrafen (1) und (3) genau das — (2) nicht. (b) Hart: $w_k \le 3$ für jeden Kunden k. Weich: $w_k \le 3 + s_k$ mit $s_k \ge 0$ und Strafkosten $c \cdot s_k$ in der Zielfunktion (B8, B18). Die Frage an den Betrieb ist die dritte der vier: *Was passiert, wenn es einmal nicht geht?* Führt die Antwort zu „dann müssen wir eben” — weich. Führt sie zu „das darf nicht passieren, dann rufen wir an und verschieben” — dann ist auch das ein Prozess, und die Bedingung bleibt weich, nur mit sehr hohen Strafkosten. Wirklich hart ist sie nur, wenn ein Verstoß rechtlich unmöglich ist. (c) Individuell. Prüfkriterium: Wird eine Wendung gefunden, die **zwei** vertretbare Modelle zulässt — und wird benannt, wer die Wahl treffen muss? Die häufigsten Kandidaten in echten Anforderungen sind „in der Regel”, „zeitnah”, „vergleichbar” und jede Zahl ohne Bezugszeitraum.

4.6 — Die Anforderung schreiben. Eine tragfähige Anforderung enthält mindestens:

- **Entscheidung:** Für jede Anfrage im Planungszeitraum: annehmen oder ablehnen.
- **Ziel:** Deckungsbeitrag maximieren (Umsatz minus Material).

- **Hart:** Die Summe der Maschinenstunden angenommener Aufträge überschreitet die verfügbare Kapazität nicht.
- **Weich:** Je abgelehntem Stammkundenauftrag 6 000 € Strafe. *(Begründung dokumentieren — der Wert ist eine kaufmännische Entscheidung, keine technische.)*
- **Bei Unlösbarkeit:** Kann es nicht geben, weil alle Regeln außer der Kapazität weich sind. Sollte die Kapazität selbst verletzt sein, liegt ein Datenfehler vor → Abbruch mit Meldung, kein Plan.
- **Auszuweisen:** angenommene Aufträge, Deckungsbeitrag, Umsatz, Auslastung, Liste der abgelehnten Stammkundenaufträge mit dem jeweils verdrängten Deckungsbeitrag.

Der letzte Punkt ist der, den man am ehesten vergisst und am dringendsten braucht: Ohne ihn kann niemand nachvollziehen, *warum* ein Stammkunde abgelehnt wurde (Abschnitt 22.3, Erklärbarkeit).

Finde den Denkfehler — „Die Engpassmaschine soll zu mindestens 92 % ausgelastet sein“

Fehler 1: Auslastung ist ein Ergebnis, keine Anforderung.

Die Vorgabe zwingt das Modell, Aufträge anzunehmen, nur damit die Maschine läuft. In einem Monat mit schwacher Nachfrage heißt das: Aufträge mit minimalem oder negativem Deckungsbeitrag werden angenommen, weil die Alternative — Leerlauf — verboten ist. Die Kennzahl wird erfüllt, das Ergebnis leidet. Das ist Goodharts Gesetz in Reinform: *Sobald eine Kennzahl zum Ziel wird, hört sie auf, ein gutes Maß zu sein.*

Fehler 2: Die Abnahme konnte den Fehler gar nicht finden.

Mit den Daten des letzten Quartals reicht die Nachfrage weit über die Kapazität hinaus — der Deckungsbeitragsplan erreicht ohnehin 100 % Auslastung. Die Nebenbedingung ist in diesem Datensatz **nicht bindend**: Sie ändert nichts, also fällt sie nicht auf. Der Abnahmetest prüfte eine Regel, die gar nicht wirkte.

Nachgerechnet mit den Daten dieses Kapitels:

Datenlage	ohne 92-%-Vorgabe	mit 92-%-Vorgabe
Starkes Quartal (800 h Nachfrage bei 300 h Kapazität)	DB 78 500 €, 100 %	**unverändert** DB 78 500 €, 100 %
Schwacher Monat (nur 210 h Nachfrage)	DB 57 000 €, 70 %	**UNZULÄSSIG**

Im schwachen Monat ist die Vorgabe schlicht nicht erfüllbar — es gibt nicht genug Arbeit. Das Modell liefert dann gar keinen Plan mehr, und der Nachtlauf bricht ab.

Was geholfen hätte:

1. **Die Vorgabe nicht als Nebenbedingung, sondern als Kennzahl im Bericht.** Auslastung gehört gemessen, nicht vorgeschrieben.
2. **Wenn sie doch ins Modell muss: weich.** Eine Strafe für Leerlaufstunden macht das Modell nie unlösbar und macht sichtbar, was der Leerlauf kostet.

3. **Abnahme mit einem Datensatz, in dem die Regel bindet.** Eine Nebenbedingung, die im Testdatensatz nichts ändert, ist ungetestet — dasselbe Argument wie beim Mutationstest in Kapitel 23.

Micro-Quiz

1. b) Die Zielfunktion ist eine vollständige Aussage darüber, was zählt. Was nicht darin steht, ist für das Modell nicht „weniger wichtig”, sondern wertlos. (a) und (c) sind frei erfunden — an der Rechengenauigkeit liegt es nicht.

2. b) Eine harte Regel, die einmal nicht erfüllbar ist, macht das ganze Modell unzulässig: Statt eines schlechteren Plans kommt gar keiner. (a) stimmt tendenziell sogar umgekehrt — harte Regeln schränken den Suchraum ein und beschleunigen oft. (c) ist falsch, ändern lässt sich alles; es merkt nur niemand rechtzeitig.

3. b) Bei knapper Kapazität ist die volle Maschine die Voreinstellung, nicht die Leistung. Zu entscheiden ist ausschließlich, *womit* sie gefüllt wird — und genau das misst die Auslastung nicht.

Selbsttest

1. Auf „Was ist Ihnen wichtig?” antwortet jeder Betrieb mit einer Liste, in der alles wichtig ist — das ergibt keine Zielfunktion. „Welche Zahl steht im Jahresbericht?” liefert dagegen genau eine Größe und nebenbei die Information, wonach die Beteiligten beurteilt werden. Das erklärt, warum „Umsatz” gesagt wird, wo „Deckungsbeitrag” gemeint ist.
2. Beispiele: *Anzahl bearbeiteter Tickets* (belohnt das Abarbeiten einfacher Fälle), *Termintreue in Prozent* (belohnt das Aufgeben ohnehin verspäteter Aufträge), *Lagerreichweite* (belohnt Überbestände), *Auslastung* (belohnt lange, unrentable Aufträge).
3. Ab einer hinreichend großen Strafe wählt der Optimierer nie mehr die bestrafte Variante — das Ergebnis ist identisch. Der Unterschied zählt genau dann, wenn die Regel *nicht* erfüllbar ist: Die harte Variante liefert dann nichts, die weiche den am wenigsten schlechten Plan.
4. Lesart 1: **Jeder** Mitarbeiter höchstens zwei Wochenenden — eine Regel je Person. Lesart 2: Im **Durchschnitt** zwei Wochenenden — Ausgleich über das Team möglich. Sie fallen auseinander, sobald jemand krank wird: Unter Lesart 1 bleibt eine Schicht unbesetzt, unter Lesart 2 springt jemand ein drittes Mal ein.
5. Eine brauchbare Antwort nennt die Reihenfolge, in der Regeln aufgegeben werden, und wer informiert wird. „Das kommt nicht vor” ist keine Antwort, weil es eine Prognose über die Zukunft ist, die niemand einhalten kann — und weil der Fall erfahrungsgemäß genau dann eintritt, wenn die Planung am dringendsten gebraucht wird.

A.5 Lösungen zu Kapitel „Lineare Programmierung — Simplex, Dualität und Schattenpreise“

5.1 — Schlupf deuten. (a) Ressourcen 1 und 3 (Schlupf 0). (b) Ja: Überall ist $s_i \cdot y_i = 0$. (c) In Ressource 3 — der höchste Schattenpreis (9,8) bedeutet den größten Grenznutzen.

5.2 — Vorzeichen. Er hat die Zielfunktion für `linprog` negiert und die Dualwerte nicht zurückgedreht. Der korrekte Schattenpreis ist **+45 €**.

5.3 — Simplex von Hand. Starttableau mit s_1, s_2 in der Basis. Erste Iteration: Pivotspalte x_1 (-5), Quotienten $24/6 = 4$ und $6/1 = 6 \rightarrow$ Pivotzeile 1. Nach dem Tausch: $x_1 = 4$, $Z = 20$. Zweite Iteration: Pivotspalte x_2, Pivotzeile 2 $\rightarrow x_1 = 3$, $x_2 = 1{,}5$, $Z = 21$. **Optimum:** $\mathbf{x}^* = (3;\ 1{,}5)$, $Z^* = 21$, Schattenpreise $y^* = (0{,}75;\ 0{,}50)$.

5.4 — Duales Problem. $\min 24y_1 + 6y_2$ u. d. N. $6y_1 + y_2 \geq 5$, $4y_1 + 2y_2 \geq 4$, $y \geq 0$. Lösung $y^* = (0{,}75;\ 0{,}50)$. Probe der Restriktionen: $6 \cdot 0{,}75 + 0{,}5 = 5$ □ (mit Gleichheit, weil $x_1 > 0$), $4 \cdot 0{,}75 + 2 \cdot 0{,}5 = 4$ □ (ebenfalls Gleichheit, weil $x_2 > 0$). Zielwert $24 \cdot 0{,}75 + 6 \cdot 0{,}5 = 18 + 3 = 21 = Z^*$ □ — starker Dualitätssatz bestätigt.

5.5 — Unbeschränktheit. Der Solver wirft „Problem ist unbeschränkt“. Geometrisch: Der zulässige Bereich $\{x_1 - x_2 \leq 5,\ x \geq 0\}$ ist nach oben offen — man kann x_1 und x_2 gemeinsam beliebig wachsen lassen (z. B. $x_1 = x_2 = t$ für $t \rightarrow \infty$), ohne eine Bedingung zu verletzen, und $Z = 2t$ wächst mit. In der Praxis fehlt fast immer eine Kapazitätsgrenze.

5.6 — Gültigkeitsbereich. (a) Bei ca. 66,7 Stunden Prüfkapazität wird die **Lackierzeit** zum Engpass; der Schattenpreis der Prüfung fällt dann. (b) Zukaufen lohnt bis zu dem Punkt, an dem der Schattenpreis unter 18 €/h fällt. (c) Der Gewinnverlauf ist **stückweise linear und konkav**: Jedes Teilstück hat die Steigung des jeweils gültigen Schattenpreises, und die Steigungen werden immer flacher — jede zusätzliche Einheit bringt weniger, weil andere Engpässe nachrücken.

5.7 — Phase 1. Ansatz: Für jede Zeile mit $b_i < 0$ (nach Umformung zu $\geq$) eine künstliche Variable $a_i \geq 0$ einführen, Hilfszielfunktion $\min \sum a_i$ lösen. Ist das Minimum 0, existiert eine zulässige Basislösung, und man startet Phase 2 mit dem erreichten Tableau. Ist es > 0, ist das Problem unzulässig. Für das Beispiel: Optimum $(12;\ 0)$, $Z = 36$.

Finde den Denkfehler — Die 380 000-Euro-Maschine

(a) **Eine Ableitung gilt lokal.** $y_i = \partial Z^* / \partial b_i$ beschreibt die Steigung der Zielfunktion **am aktuellen Punkt**. Sie sagt: „Die *nächste* Stunde ist 50 € wert“ — nicht: „jede der nächsten 2 000 Stunden ist 50 € wert“. Der Werksleiter behandelt eine Ableitung wie eine Konstante und multipliziert sie mit einer großen Zahl. Genau daran scheitert die Rechnung.

(b) **Der wahre Verlauf ist stückweise linear und knickt ab.** Solange der Lackierofen der Engpass bleibt, gilt tatsächlich 50 €/h. Irgendwann ist aber genug Ofenzeit da — dann bindet eine **andere** Ressource, und weitere Ofenstunden bringen nichts mehr. Am Beispiel aus dem Kapitelanfang (Zusatzstunden h am Lackierofen):

h	tatsächlicher Z^*	tatsächlicher Zuwachs	linear hochgerechnet
0	8 200 €	0 €	0 €
20	9 200 €	1 000 €	1 000 €

h	tatsächlicher Z^*	tatsächlicher Zuwachs	linear hochgerechnet
40	10 200 €	2 000 €	2 000 €
60	**11 200 €**	**3 000 €**	3 000 €
100	11 200 €	3 000 €	5 000 €
500	11 200 €	3 000 €	25 000 €
2 000	**11 200 €**	**3 000 €**	**100 000 €**

Bei $h = 60$ ist der Plan bei $(0;\ 80)$ angekommen: Es werden nur noch Rahmen gebaut, und die **Montage** ist zum Engpass geworden. Ab dort fällt der Schattenpreis des Lackierofens auf **0**. Der wahre Jahresnutzen der zweiten Maschine beträgt **3 000 €**, nicht 100 000 € — die Hochrechnung überschätzt um **Faktor 33**. Die Amortisation dauert nicht 3,8 Jahre, sondern rund **127 Jahre**.

Der Fehler geht dabei **immer** in dieselbe Richtung: Die lineare Hochrechnung ist stets zu optimistisch, weil der Schattenpreis mit wachsender Kapazität nur fallen, nie steigen kann. Wer so rechnet, kauft systematisch zu teuer ein.

(c) **Zwei fehlende Prüfungen.** Erstens der **Entartungstest**: Sind mehr Nebenbedingungen aktiv als Variablen vorhanden, ist der Wert 50 € nur einer von vielen möglichen und hätte gar nicht als Einzelwert berichtet werden dürfen (siehe `Toleranzen_und_Entartung.py`). Zweitens der **Gültigkeitsbereich**: Bis zu welcher Kapazitätserhöhung bleibt dieser Schattenpreis überhaupt bestehen? Ohne diese Spanne ist die Zahl nicht interpretierbar.

(d) **Nicht extrapolieren — nachrechnen.** Der zusätzliche Nutzen ergibt sich aus einem zweiten Solverlauf, nicht aus einer Multiplikation:

```
z_basis = -linprog(c, A_ub=A, b_ub=b, bounds=(0, None)).fun
b_neu = b.copy(); b_neu[0] += 2000                # zweiter Lackierofen
z_neu = -linprog(c, A_ub=A, b_ub=b_neu, bounds=(0, None)).fun
print(f"Tatsaechlicher Zusatznutzen: {z_neu - z_basis:,.0f} EUR")   # 3.000 EUR
```

Das ist die allgemeine Regel für jede Sensitivitätsaussage über mehr als eine Grenzeinheit: **Der Schattenpreis zeigt die Richtung, die Neuberechnung liefert den Betrag.** Bei einer Investitionsentscheidung kostet ein zusätzlicher Solverlauf Millisekunden — und hier hätte er 380 000 € gespart.

Micro-Quiz

1 — (b). Genau der Satz vom komplementären Schlupf: $s_i \cdot y_i = 0$. Ist Schlupf vorhanden, muss der Dualwert null sein. (a) hat die Aussage in ihr Gegenteil verkehrt; (c) verwechselt Schlupf mit Auslastung — $s_i = 12$ heißt gerade, dass 12 Einheiten *übrig* sind.

2 — (b) Entartung. 7 aktive Bedingungen bei 5 Variablen bedeuten: Die Ecke ist überbestimmt, es gibt mehrere optimale Dualvektoren, und welchen der Solver zeigt, hängt vom gewählten Verfahren ab. (a) verharmlost genau den Fall, der Fehlentscheidungen produziert; (c) verwechselt Entartung mit Unlösbarkeit — der Plan selbst ist völlig in Ordnung und eindeutig.

3 — (b). Der Schlupf ist Solver-Rauschen in der Größenordnung der Maschinengenauigkeit — rechnerisch null, aber nicht `== 0.0`. Richtig ist `abs(schlupf) < 1e-7`. (a) unterstellt dem Solver einen Fehler,

den er nicht gemacht hat; (c) ist frei erfunden — Schlupfwerte sind bei korrekt aufgestelltem Modell nie negativ, abgesehen von genau diesem Rauschen.

Selbsttest

1. Ungenutzte Kapazität; der Wert 0 kennzeichnet einen **Engpass** (bindende Bedingung).
2. Minimaler Quotient über positive Spalteneinträge. Nähme man die erste Zeile, würde eine Basisvariable negativ — die Lösung wäre unzulässig.
3. $s_i y_i = 0$: Entweder hat eine Ressource Reserven (dann ist sie nichts wert), oder sie ist knapp (dann kann sie einen positiven Preis haben).
4. Weil zur Maximierung die Zielfunktion negiert wurde; die Ableitung nach b_i dreht damit ebenfalls das Vorzeichen.
5. Eine fehlende Kapazitätsbeschränkung — der zulässige Bereich ist in Richtung der Zielfunktion offen.

A.6 Lösungen zu Kapitel „Gemischt-ganzzahlige Optimierung — Diskrete Entscheidungen und Branch-and-Bound“

6.1 — Runden widerlegen. Beispiel: max $x_1 + x_2$ u. d. N. $10x_1 + 10x_2 \le 15$, ganzzahlig. LP-Optimum $Z = 1{,}5$; Abrunden ergibt $(0,0)$ mit $Z = 0$ — **100 % Verlust**. Das ganzzahlige Optimum ist $(1,0)$ mit $Z = 1$. Prinzip: kleine Zahlen plus knappe Kapazität.

6.2 — Big-M wählen. $M = 250$ (die bekannte Kapazität). Bei $M = 10^6$ bleibt das Modell korrekt, aber die LP-Relaxation wird extrem schwach: y_j darf schon bei $x_j/10^6$ liegen, die Schranke ist praktisch wertlos, und Branch-and-Bound muss weit mehr Knoten durchsuchen.

6.3 — Regeln übersetzen. (a) $y_A + y_B \le 1$. (b) $\sum_{j=1}^{4} y_j \ge 2$. (c) $y_1 + y_2 - 1 \le y_{\text{Lager}}$. (d) $500\,y \le x \le 2000\,y$ mit $y \in \{0,1\}$. (e) $\sum_{j=1}^{3} y_j = 1$.

6.4 — Branch-and-Bound. LP-Relaxation der Wurzel: nach Nutzen/Gewicht sortieren ($8/5 = 1{,}6$; $11/7 = 1{,}57$; $6/4 = 1{,}5$; $4/3 = 1{,}33$). Gierig füllen: $x_1 = 1$ (Rest 9), $x_2 = 1$ (Rest 2), $x_3 = 0{,}5 \rightarrow Z_{LP} = 8 + 11 + 3 = 22$. Verzweigen über x_3. Ast $x_3 = 0$: $x_1 = 1, x_2 = 1, x_4 = 2/3 \rightarrow Z = 21{,}67$; weiter verzweigen → beste ganzzahlige Lösung $(1,1,0,0)$ mit $Z = 19$. Ast $x_3 = 1$: $x_1 = 1$, $x_3 = 1$, Rest 5 → $x_2 = 5/7 \rightarrow Z = 21{,}86$; verzweigen führt auf $(1,0,1,1)$ mit $Z = 18$ und $(0,1,1,0)$ mit $Z = 17$. **Optimum:** $(1,1,0,0)$**,** $Z = 19$**.**

6.5 — Kardinalität variieren. (a) Ab $K = 3$ steigt der Ertrag nicht mehr wesentlich, weil bereits drei Positionen à 40 000 € das Budget von 100 000 € abdecken können. (b) Rechenzeit steigt zunächst (mehr Kombinationen), fällt bei großem K wieder (Restriktion bindet nicht mehr). (c) Der faire Preis ist die Differenz der Netto-Erträge zwischen $K = 3$ und $K = 4$ — bei den gegebenen Daten nahe null, weil die Obergrenze von 40 000 € bereits bindet.

6.6 — Big-M-Effekt. Erwartetes Muster: Knotenzahl und Laufzeit steigen deutlich mit M. Bei $M = 10^9$ ist die Relaxation so schwach, dass der Solver kaum noch prunen kann.

6.7 — Standortplanung. Modell wie in Projekt P4. Prüfen Sie: Gesamtkapazität der eröffneten Lager ≥ Gesamtbedarf (140); bei Kapazität 80 je Lager sind mindestens $\lceil 140/80 \rceil = 2$ Lager nötig.

Finde den Denkfehler — Elf Lager, die keine Fixkosten kosten

(a) **Warum die Zahl kleiner ist.** Ein größeres M **lockert** die Nebenbedingung $\sum_j x_{ij} \le M y_i$. Der zulässige Bereich des Modells wird also größer, und in einem größeren Bereich kann das Minimum nur kleiner oder gleich bleiben. Bei einem *korrekt* formulierten Modell dürfte das aber gar nicht auffallen: Solange y_i wirklich binär ist, ist $M \cdot 1 = M$ ohnehin größer als jede mögliche Liefermenge — die Lockerung wäre wirkungslos. Dass sich der Zielwert überhaupt ändert, ist deshalb schon der Beweis, dass y_i **nicht** wirklich binär ist.

(b) **Die Binärvariablen.** Sie stehen bei rund $4 \cdot 10^{-8}$ — nicht bei 0 und nicht bei 1. Die Ganzzahltoleranz von HiGHS (wie der meisten Solver) beträgt 10^{-6}. Alles, was näher als das an einer ganzen Zahl liegt, gilt als ganzzahlig. $4 \cdot 10^{-8}$ ist damit für den Solver eine saubere **0**: „Lager geschlossen“.

Zugleich ist M so groß, dass

$$\sum_j x_{ij} \le \underbrace{7{,}2 \cdot 10^9}_{M} \cdot \underbrace{4 \cdot 10^{-8}}_{y_i} \approx 289$$

immer noch reichlich Liefermenge erlaubt. Das Lager liefert also, ohne offiziell offen zu sein. Der Fachbegriff für dieses Muster ist **trickle flow**: Ein verschwindend kleiner Schaltwert lässt einen großen Fluss durch, weil er mit einer riesigen Zahl multipliziert wird.

(c) **Die Bilanz.** In der „Lösung“ liefern **11 von 12 Lagern** tatsächlich Ware. Verbucht werden dafür **0,00 €** Fixkosten statt der real anfallenden **35 847,91 €**. Der gemeldete Zielwert von 12 441,96 € ist damit nicht etwa etwas zu optimistisch, sondern schlicht falsch: Die tatsächlichen Kosten dieses Plans betragen **48 289,86 €** — fast das Doppelte der korrekten Lösung (26 525,28 €).

Und das ist die eigentliche Gemeinheit: Der Fehler tarnt sich als **Erfolg**. Niemand hinterfragt ein Ergebnis, das 53 % günstiger ausfällt als erwartet — man freut sich darüber.

(d) **Die Prüfung.** Drei Tests, zusammen keine zwanzig Zeilen, die in jedes MILP-Auswertungsskript gehören:

```
# 1. Sind die Binaervariablen wirklich binaer?
assert np.abs(y - np.round(y)).max() <= 1e-6, "y ist nicht ganzzahlig"

# 2. Widerspricht sich die Loesung selbst?
liefert = x.sum(axis=1) > 1e-6
assert not (liefert & (y < 0.5)).any(), "Lager liefert, gilt aber als geschlossen"

# 3. Stimmt der Zielwert mit den Kosten der Loesung ueberein?
echte_kosten = FIXKOSTEN[liefert].sum() + (TRANSPORT * x).sum()
assert abs(echte_kosten - zielwert) < 1e-4 * echte_kosten, \
    f"Zielwert {zielwert} passt nicht zu den echten Kosten {echte_kosten}"
```

Test 3 ist der wichtigste. Er vergleicht nicht Modell mit Modell, sondern **Modell mit Wirklichkeit** — er rechnet die Kosten aus der ausgegebenen Lösung neu aus, ganz ohne Solver. Damit fängt er nicht nur den Big-M-Fehler, sondern jeden Fehler, bei dem die Zielfunktion etwas anderes bilanziert als der Plan tatsächlich kostet.

Vorbeugend gilt weiter die Regel aus dem Kapitel: M so klein wie möglich, hergeleitet aus einer echten Kapazität. Hier wäre das schlicht die Lagerkapazität — mehr kann ein Lager ohnehin nicht ausliefern.

Micro-Quiz

1 — (b). Der Gap $(48\,200 - 47\,100)/48\,200 = 2{,}3\,\%$ ist eine **Garantie**, keine Schätzung: Besser als 47 100 € kann es nachweislich nicht werden, also ist der vorliegende Plan höchstens 2,3 % zu teuer. (a) wirft eine völlig brauchbare Lösung weg — genau der Fehler, den `if status == OPTIMAL: ... else: return None` produziert. (c) verwechselt die Schranke mit einem erreichbaren Wert: 47 100 € ist eine *untere* Schranke, es ist völlig offen, ob ein Plan mit diesen Kosten überhaupt existiert.

2 — (b). Die Laufzeit (a) ist ein realer, aber beherrschbarer Nachteil — man merkt ihn und kann reagieren. Der Trickle Flow dagegen ist **still**: Das Modell meldet `Optimal`, liefert eine Zahl, und niemand sieht, dass die Fixkosten verschwunden sind. Ein Fehler, den man bemerkt, ist immer harmloser als einer, den man nicht bemerkt. (c) ist frei erfunden — negative Kosten entstehen dabei nicht.

3 — (c). Ein Hinweis ist für CP-SAT unverbindlich: Er wird als Startlösung ausprobiert und verworfen, wenn er nicht zulässig ist. Das Ergebnis bleibt in jedem Fall korrekt (b ist also falsch), und `INFEASIBLE` bezieht sich immer auf das Modell, nie auf einen Hinweis (a ist falsch). Praktische Konsequenz: Prüfen Sie Ihre Startlösung selbst auf Zulässigkeit — sonst messen Sie einen Warm-Start-Effekt, den es gar nicht gibt.

Selbsttest

1. Weil die Relaxation **mehr** Lösungen zulässt (alle ganzzahligen plus gebrochene) — das Maximum über einer größeren Menge ist mindestens so groß.
2. Teilproblem unzulässig; Schranke schlechter als der Incumbent; LP-Lösung bereits ganzzahlig.
3. $L\,y \le x \le U\,y$ mit binärem y.
4. Weil die LP-Relaxation dadurch schwächer wird: Die berechnete Schranke liegt weiter vom wahren Optimum entfernt, es kann weniger gekappt werden.
5. Die beste gefundene Lösung ist höchstens 3 % vom beweisbaren Optimum entfernt.

A.7 Lösungen zu Kapitel „Constraint Programming mit CP-SAT — Logik, Scheduling und Zuweisung“

7.1 — Propagation. Aus $x_1 + x_2 = 8$ und $x_1 < x_2$ folgt $x_1 < 4$, also $x_1 \in \{2, 3\}$ (denn $x_2 = 8 - x_1 \le 6$ verlangt $x_1 \ge 2$) und entsprechend $x_2 \in \{5, 6\}$. Aus 36 Kombinationen werden 2 zulässige.

7.2 — Hart oder weich. (a) hart. (b) weich, mittlere Strafe (~50). (c) hart. (d) weich, mittlere Strafe (~80, weil geteilte Dienste stark belasten). (e) hart, falls gesetzlich; sonst weich mit sehr hoher Strafe (~1000).

7.3 — Regeln ergänzen. (a) `for s in (2,3): modell.Add(x["Frau_Albrecht", s] == 0)` (b) Hilfsvariablen `arbeitet_bauer`, `arbeitet_koch` per `AddMaxEquality` an die Zuweisungssummen koppeln, dann `modell.Add(arbeitet_bauer + arbeitet_koch <= 1)`. (c) Belohnung = negative Strafe: `strafterme.append(-30 * folge_var)`, wobei `folge_var` per `AddBoolAnd`/`OnlyEnforceIf` an $x_{p,0} \wedge x_{p,1}$ gekoppelt wird.

7.4 — Infeasibility. Mit `MAX_VERTRETUNGEN = 0` meldet der Solver `INFEASIBLE`. Nach Einbau der Schlupfvariablen lässt der Solver **alle vier** Stunden ausfallen (4 × 10 000 = 40 000 Strafpunkte) — es geht nicht anders. Interessanter wird es bei `MAX_VERTRETUNGEN = 1`: Dann fällt genau die Stunde aus, für die am wenigsten qualifiziertes Personal zur Verfügung steht.

7.5 — Sudoku.

```
x = [[m.NewIntVar(1, 9, f"x{i}{j}") for j in range(9)] for i in range(9)]
for i in range(9): m.AddAllDifferent(x[i])                      # Zeilen
for j in range(9): m.AddAllDifferent([x[i][j] for i in range(9)]) # Spalten
for bi in range(3):
    for bj in range(3):
        m.AddAllDifferent([x[3*bi+i][3*bj+j] for i in range(3) for j in range(3)])
for i in range(9):
    for j in range(9):
        if vorgabe[i][j]: m.Add(x[i][j] == vorgabe[i][j])
```

Etwa 10 Zeilen Modellcode — und der Solver löst jedes Sudoku in Millisekunden. Das ist die Stärke globaler Constraints.

7.6 — Die Grenze der Parallelität finden. (a) Der Gewinn flacht ab und kehrt sich um. Auf einem Rechner mit 24 Kernen gemessen (Faktor gegenüber einem Arbeiter): 8 Arbeiter **10,2**, 16 Arbeiter **12,3**, 24 Arbeiter **9,0**. Wer alle Kerne belegt, ist langsamer als mit zwei Dritteln — dem Betriebssystem, dem Speicherbus und den Suchsträngen untereinander bleibt zu wenig Luft. Die Faustregel „so viele Arbeiter wie Kerne” ist also falsch; die richtige Zahl ist eine Messung, keine Einstellung. (b) **Fünf verschiedene Pläne aus fünf Seeds** — bei `num_workers = 1` und identischem Zielwert 183. Das ist die andere Hälfte der Antwort: Der Seed *wirkt*, aber nur auf den einzelnen Suchstrang. Es sichert also **keiner der beiden Parameter allein** die Reproduzierbarkeit:

	Seed fest	Seed variabel
1 Arbeiter	reproduzierbar	5 Pläne aus 5 Läufen
mehrere Arbeiter	3 Pläne aus 4 Läufen	erst recht nicht

Nur die Kombination trägt. (c) Bei **15 Aufträgen** (60 s Zeitlimit): Ein Arbeiter läuft ins Limit und meldet `FEASIBLE` mit Makespan 200; acht Arbeiter melden `OPTIMAL` mit **demselben** Makespan 200 nach 26,4 s. Bei 13 Aufträgen sind es 11,2 s gegen 2,8 s, bei 14 schon 21,4 s gegen 3,7 s. Bemerkenswert ist, dass beide denselben Wert finden — der Unterschied liegt nicht in der Lösung, sondern im **Beweis**, dass es keine bessere gibt. Genau diese Arbeit teilen sich die parallelen Stränge auf.

7.7 — Job-Shop erweitern. (a) Rüstzeiten: `AddNoOverlap` durch paarweise Disjunktionen mit Übergangszeit ersetzen, oder `AddCircuit` je Maschine mit Übergangsmatrix. (b) Verspätung: `tardiness = MaxEquality(0, ende - faellig)`, in die Zielfunktion. (c) `AddCumulative(intervalle, [1]*n, 2)` statt `AddNoOverlap` für die Doppelmaschine.

7.8 — Wochendienstplan. Siehe Projekt P2. Kernpunkte: Nachtschicht-Folgeregel über `AddImplication`; „höchstens 5 Tage in Folge“ über gleitende Fenster (`sum(x[p,t..t+5]) <= 5`).

Finde den Denkfehler — Die Betriebsvereinbarung, die niemanden interessiert

(a) **Warum die Regel verletzt wird.** Weil sie einen **Preis** hat. Eine weiche Nebenbedingung verbietet nichts — sie verteuert. Der Solver vergleicht in jedem Schritt, was ihn eine Verletzung kostet und was sie ihm einbringt. Bei einem Gewicht von 1 gegen 10 lohnt sich eine Drei-Tage-Serie, sobald sie auch nur ein Zehntel eines Wunsches rettet. Der Solver hat genau das getan, was im Modell steht — er hat nur nicht getan, was gemeint war.

(b) **Was** $(1, 10)$ **wörtlich bedeutet.** Nicht „Wunschfrei ist wichtiger“, sondern:

„Zehn Verstöße gegen die Betriebsvereinbarung sind für uns genauso schlimm wie ein einziger abgelehnter Wunschfrei-Tag. Bis zu diesem Verhältnis tauschen wir gerne.“

So ausgesprochen, würde niemand diesem Satz zustimmen. Genau das ist der Punkt: Gewichte sind **Wechselkurse**, keine Rangfolgen. Sie legen fest, wie viele Einheiten der einen Regel gegen eine Einheit der anderen getauscht werden — und wer sie nach Bauchgefühl setzt, unterschreibt einen Kurs, den er nie gelesen hat.

(c) **Der Kipppunkt.** Die Tabelle aus `Strafgewichte.py`:

Strafe je 3er-Serie	3er-Serien	Wunsch verletzt
1	12	6
2	12	6
5	9	7
10	2	11
30	**0**	**14**
100	0	14

Ab Gewicht 30 verschwinden die Serien vollständig. Der Preis dafür sind **14 statt 6** abgelehnte Wünsche. Beachten Sie: Zwischen 30 und 100 ändert sich nichts mehr — ist die Regel einmal teuer genug, macht noch mehr Strafe keinen Unterschied. Es gibt also keinen Grund, „sicherheitshalber“ 10 000 zu nehmen; das verschlechtert nur die Konditionierung (Abschnitt 2.7).

Bemerkenswert ist außerdem, dass sich zwischen 1 und 2 gar nichts tut, zwischen 5 und 10 dagegen sehr viel. Wer ein einziges Gewicht ausprobiert, weiß nicht, ob er zufällig auf einem Plateau oder direkt an einer Kante steht.

(d) **Die Frage, die vorher zu stellen ist.** Nicht „wie wichtig ist Ihnen diese Regel?“ — darauf lautet die Antwort immer „sehr“. Sondern:

„Wie viele abgelehnte Wunschfrei-Tage nehmen Sie in Kauf, um eine Drei-Tage-Serie zu vermeiden?“

Das ist eine Frage, die eine Zahl als Antwort hat. Und wenn der Auftraggeber sie nicht beantworten kann, legt man ihm die Tabelle aus (c) vor und lässt ihn eine **Zeile** auswählen. Das dauert eine Stunde und ersetzt drei Abstimmungsrunden.

Und wann gehört eine Regel gar nicht in die Zielfunktion? Wenn sie unverhandelbar ist. Gesetzliche Ruhezeiten, Qualifikationsanforderungen, eine Betriebsvereinbarung — das sind **harte** Nebenbedingungen. Dass das Modell dann `INFEASIBLE` melden kann, ist kein Nachteil, sondern die ehrliche Auskunft: *Mit diesem Personalbestand ist kein zulässiger Plan möglich.* Diese Aussage ist für die Klinikleitung wertvoller als ein Plan, der formal optimal ist und die Vereinbarung zwölfmal bricht.

Der Merksatz dazu: **Alles, was einen Preis hat, wird irgendwann gekauft.**

Micro-Quiz

1 — (b) wirkungslos. `INFEASIBLE` ist kein Abbruch, sondern ein **Beweis**: CP-SAT hat gezeigt, dass keine zulässige Lösung existiert. Mehr Zeit (a) oder mehr Arbeiter (c) ändern daran nichts — sie bestätigen dasselbe Ergebnis nur schneller. Der Unterschied zu `UNKNOWN` ist genau dieser: Dort *wurde* nichts gefunden, hier *gibt es* nichts. Die Ursache liegt in den harten Regeln; Anhang C zeigt, wie man die widersprüchliche Teilmenge isoliert.

2 — (b). `AddNoOverlap` ist kompakter *und* propagiert stärker, weil der spezialisierte Propagator alle Intervalle gemeinsam betrachtet statt paarweise. (a) ist falsch: Beide Formulierungen beschreiben dieselbe Menge zulässiger Lösungen, also auch dasselbe Optimum — der Unterschied liegt in der Laufzeit, nicht in der Qualität. (c) ist falsch: CP-SAT rechnet ausschließlich mit ganzen Zahlen, kontinuierliche Zeiten kann es gerade **nicht**.

3 — (b). Mehrfache Optima sind bei Scheduling der Normalfall, kein Fehler. (a) verwechselt Eindeutigkeit mit Korrektheit; (c) unterstellt ein Genauigkeitsproblem, das es nicht gibt — alle gefundenen Pläne haben exakt denselben Makespan. Praktische Konsequenz: `assert plan == erwarteter_plan` besteht mal und scheitert mal, ohne dass sich am Code etwas geändert hat. Prüfen Sie stattdessen `assert makespan == 11` und die Einhaltung aller Regeln. Wer eine reproduzierbare Ausgabe braucht — etwa für ein Buch —, fixiert `num_workers` und `random_seed`.

Selbsttest

1. Sie entfernt Werte aus den Wertebereichen, die aufgrund der Bedingungen unmöglich sind — **bevor** gesucht wird. Dadurch schrumpft der Suchbaum drastisch.
2. Stärkere Propagation: Der spezialisierte Algorithmus betrachtet die Struktur als Ganzes (Matching-Theorie), nicht nur Paare.
3. Eine Variable mit Start, Dauer und Ende; sie garantiert automatisch `start + dauer == ende`.
4. Weil jede zusätzliche harte Bedingung das Risiko von `INFEASIBLE` erhöht — und eine Fehlermeldung dem Anwender nicht hilft.
5. Weil Anwender die Frage „Warum ich?“ stellen. Ohne Antwort wird das System umgangen.
6. Weil der Seed nur festlegt, wie ein **einzelner** Suchstrang würfelt — nicht, welcher von mehreren parallelen zuerst fertig wird. Das entscheidet die Ausführungsreihenfolge, also die Uhr. Abhilfe: zusätzlich `num_workers = 1`. Gemessen: schon zwei Arbeiter liefern bei identischem Seed drei verschiedene Pläne zum selben Zielwert.
7. Weil CP-SAT nicht dieselbe Suche vervielfacht, sondern **verschiedene Strategien** nebeneinander laufen lässt (LP-gestützt, andere Verzweigungsregeln, lokale Suche), die einander ihre Schranken mitteilen. Ein zusätzlicher Arbeiter ist deshalb nicht ein weiterer Kern für dieselbe Arbeit, sondern ein anderes Verfahren — und wenn eines davon früh eine gute Schranke findet, profitieren alle.

A.8 Lösungen zu Kapitel „Graphen, Flüsse und Touren — Min-Cost-Flow, Matching und VRP“

8.1 — Flusserhaltung. $b_i = (15 + 3) - (12 + 8) = -2 \rightarrow$ **Senke** (Nettobedarf 2).

8.2 — Unlösbarkeit. Die Summe aller Flusserhaltungsgleichungen ergibt $\sum_i b_i = 0$ (jede Kante taucht einmal mit $+1$ und einmal mit -1 auf). Ist die Summe ungleich null, widersprechen sich die Gleichungen. Bei Angebotsüberschuss führt man einen künstlichen **Dummy-Senkenknoten** mit dem Restbedarf und Kosten 0 ein.

8.3 — Transportproblem. (a) Angebot $30+25+45 = 100$, Bedarf $25+30+20+25 = 100$ □ (b) Optimale Kosten: **790**. Transportplan:

von nach	L1	L2	L3	L4	Summe
W1	0	10	20	0	30
W2	25	0	0	0	25
W3	0	20	0	25	45
Summe	25	30	20	25	100

Beachten Sie, dass Werk 3 seine günstigste Verbindung (L4 zu 5) voll ausschöpft und W2 ausschließlich L1 beliefert (9), obwohl L4 mit 7 billiger wäre — dort ist der Bedarf bereits von W3 gedeckt. (c) Die Lösung ist ganzzahlig, obwohl nicht gefordert — die Transportmatrix ist **total unimodular**, alle Ecken sind ganzzahlig.

8.4 — Zuordnung mit Verboten. Kosten auf einen sehr hohen Wert setzen (`kosten[2][2] = 1e6`) oder mit `np.inf` arbeiten (bei `linear_sum_assignment` erlaubt). Die Lösung weicht auf die zweitbeste Zuordnung für Carla aus; die Gesamtkosten steigen um die Differenz.

8.5 — Engpass finden. Die Dualwerte der Kapazitätsschranken (`res.upper.marginals` bei `linprog`) zeigen es direkt; alternativ jede Kapazität einzeln um 1 erhöhen und neu rechnen. Im Beispiel ist die Kante `Werk_A → Umschlag` der Engpass (voll ausgelastet, günstigster Weg).

8.6 — VRP variieren. (a) Mit 3 Fahrzeugen werden die Touren länger, die Gesamtfahrzeit steigt leicht, die Auslastung deutlich. (b) Unlösbar, sobald Kapazität × Fahrzeuge < Gesamtbedarf (hier: bei 3 Fahrzeugen à 10 sind $30 < 37 \rightarrow$ unlösbar) **oder** wenn Zeitfenster nicht mehr eingehalten werden können. (c) 1 s liefert meist eine brauchbare, 30 s eine spürbar bessere Lösung — die Metaheuristik verbessert kontinuierlich. (d) Doppelte Servicezeit kann Zeitfenster verletzen → möglicherweise unlösbar.

8.7 — TSP mit MTZ. Erwartetes Ergebnis: Bei 8 Städten löst das MILP in Sekunden. Ab etwa 12–15 Städten wird die schwache MTZ-Relaxation zum Problem, und die Laufzeit steigt stark — während OR-Tools weiterhin in Sekundenbruchteilen sehr gute Touren liefert.

Finde den Denkfehler — Die vergessene Dimension

(a) **Was die Fahrzeuge tun.** Fahrzeuge 1, 2 und 3 fahren **gar nicht** — sie stehen mit null Stopps im Depot. Fahrzeug 4 bedient alle 16 Kunden und lädt dabei **37 Paletten** bei einer Kapazität von 10. Es ist um 270 % überladen. Physikalisch ist dieser Plan nicht ausführbar; im Modell ist er die beste Lösung.

Dass ein einzelnes Fahrzeug alles fährt, ist übrigens völlig folgerichtig: Ohne Ladungsgrenze ist jede zusätzliche Tour reine Zusatzstrecke (der Weg vom Depot zum ersten Kunden und zurück). Ein Optimierer, der nur Kilometer minimiert, wird deshalb **immer** alles auf ein Fahrzeug legen. Das leere Depot ist das Symptom.

(b) **Warum die Bibliothek nichts gemerkt hat.** Weil im Modell keine Ladung existiert. Die Routing-Bibliothek kennt keine Kapazität als Begriff — sie kennt **Dimensionen**: benannte Größen, die sich entlang einer Tour aufsummieren und begrenzt werden können. Distanz ist eine Dimension, Zeit ist eine, Ladung wäre eine.

Dass `KAPAZITAETEN = [10, 10, 10, 10]` im Datenteil steht, ändert daran nichts. Eine Liste in Ihrem Python-Programm ist keine Nebenbedingung. Sie wird erst zu einer, wenn `AddDimensionWithVehicleCapacity` sie dem Modell übergibt. Das ist eine allgemeine Lehre, nicht nur für Routing: **Daten, die kein Constraint anfasst, sind Dekoration.**

(c) **Warum das bessere Ergebnis das gefährliche ist.** Ein Modellfehler, der zu `INFEASIBLE` oder zu absurd hohen Kosten führt, fällt sofort auf. Dieser hier führt zu einem Ergebnis, das **43 % besser** aussieht als die korrekte Lösung — und das verteidigt man gegenüber dem Auftraggeber, statt es zu hinterfragen. Es ist dasselbe Muster wie bei der Big-M-Falle in Abschnitt 6.10: Der Fehler tarnt sich als Erfolg.

Die Regel dazu lautet: **Ein Optimierungsergebnis, das deutlich besser ausfällt als erwartet, ist zuerst ein Fehlerverdacht.** Freuen Sie sich erst, nachdem Sie nachgerechnet haben.

(d) **Die Prüfung — und warum sie unabhängig sein muss.**

```
for fahrzeug, (ladung, kapazitaet) in enumerate(zip(ladungen, KAPAZITAETEN)):
    assert ladung <= kapazitaet, \
        f"Fahrzeug {fahrzeug + 1}: {ladung} Paletten bei Kapazitaet {kapazitaet}"
```

Dabei wird `ladung` **aus der ausgegebenen Tour** aufsummiert — also aus der Liste der angefahrenen Kunden und deren Bedarfen. Entscheidend ist, was man dafür *nicht* benutzt:

```
# FALSCH als Pruefung:
ladung = loesung.Value(routing.GetDimensionOrDie("Ladung").CumulVar(index))
```

Diese Zeile fragt das Modell, ob das Modell sich an sich selbst hält. Existiert die Dimension „Ladung“ gar nicht, stürzt sie ab — und existiert sie, kann sie per Konstruktion nie verletzt sein. Sie prüft also entweder nichts oder gar nichts.

Die allgemeine Regel: Eine Prüfung, die dieselben Bausteine verwendet wie das Modell, prüft das Modell gegen sich selbst. Nützlich ist nur eine Prüfung, die von der **ausgegebenen Lösung** ausgeht und

die Anforderungen der Wirklichkeit unabhängig nachrechnet — genau wie die Kostengegenrechnung in Abschnitt 6.10 und die Verbrauchsprüfung in Abschnitt 1.10.

Micro-Quiz

1 — (b) totale Unimodularität. Jede quadratische Teilmatrix hat Determinante 0, +1 oder −1; bei ganzzahliger rechter Seite sind deshalb alle Ecken des zulässigen Bereichs ganzzahlig. Da das LP-Optimum nach dem Fundamentalsatz in einer Ecke liegt, ist es automatisch 0/1-wertig. (a) unterschätzt das: Es ist kein Zufall, sondern eine Struktureigenschaft, die für **jede** Kostenmatrix gilt. (c) ist falsch — `linprog` rundet nichts.

2 — (b) die Struktur geht verloren. „Höchstens 4 von 12“ ist eine Kardinalitätsregel; ihre Zeile passt nicht in das Schema, das die totale Unimodularität sichert. Die Relaxation kann dann Brüche liefern (etwa vier Monteure zu je 0,75 Überstunden), und man braucht Binärvariablen und Branch-and-Bound. (a) ist die gefährliche Fehlannahme: Die Eigenschaft ist **zerbrechlich**, eine einzige zusätzliche Zeile genügt. (c) verwechselt „schwerer zu lösen“ mit „unlösbar“.

3 — (b) fehlende Dimension. Ohne begrenzende Dimension ist jede zusätzliche Tour reine Zusatzstrecke — ein Optimierer legt dann folgerichtig alles auf ein Fahrzeug. Genau dieses Bild zeigt Abschnitt 8.7. (a) wäre zu prüfen, wenn die Lösung *schlecht* aussähe, nicht wenn sie verdächtig gut ist. (c) ist zwar eine sinnvolle Datenprüfung, erklärt aber nicht, warum drei Fahrzeuge unbenutzt bleiben.

Selbsttest

1. Abfluss minus Zufluss = Knotensaldo; entspricht der Kirchhoffschen Knotenregel.
2. Wegen totaler Unimodularität sind alle Ecken des Polyeders ganzzahlig, und das LP-Optimum liegt in einer Ecke.
3. Isolierte Kreise, die das Depot nicht enthalten. MTZ führt Rangvariablen ein, die entlang benutzter Kanten streng wachsen müssen — in einem Kreis unmöglich.
4. Weil die MTZ-Relaxation sehr schwach ist und spezialisierte Metaheuristiken um Größenordnungen bessere Ergebnisse in kürzerer Zeit liefern.
5. Kapazität < Bedarf (Summen vergleichen); Zeitfenster physisch unerreichbar (Direktfahrt prüfen); zu wenige Fahrzeuge für die Zeitfenster (Fahrzeugzahl variieren).

A.9 Lösungen zu Kapitel „Metaheuristiken — wenn der exakte Solver aussteigt“

9.1 — Die Kurzsichtigkeit von Hand. Ab *Schwarz* wählt die Faustregel: Schwarz → Dunkelrot (26) → Rot (5) → … und muss dann in die helle Gruppe: Rot → Weiß (50) → Elfenbein (2) → Beige (2). Summe **85 Minuten** — also noch einen Deut schlechter als die 84 ab Weiß. Der Grund ist derselbe: Die Regel erwischt den Gruppenwechsel an der teuersten Stelle. Bemerkenswert ist aber, dass der *Startpunkt* kaum etwas ändert. Das ist typisch: Eine kurzsichtige Regel scheitert nicht am Anfang, sondern an der Reihenfolge, in der sie ihre Möglichkeiten verbraucht.

9.2 — Zuggröße und Temperatur. Gesucht ist T mit $e^{-70/T} = e^{-1}$, also $T = 70$. Die getesteten Temperaturen liegen zwischen 0,5 und 8 — allesamt **eine Größenordnung darunter**. Bei $T_0 = 8$ hat eine mediane Verschlechterung die Annahmewahrscheinlichkeit $e^{-70/8} \approx 0{,}00016$. Genau das ist der Punkt der

Warnung im Kapitel: Die verbreitete Regel „20 bis 50 % Annahmequote“ würde hier $T_0 \approx 50$ verlangen, und bei dieser Temperatur ist die Suche ein reiner Zufallslauf.

9.3 — Die teure Bewertung. Statt `delta_verschieben` die Reihenfolge tatsächlich umbauen und `gesamtruestzeit()` neu rechnen. Bei $n = 200$ kostet das rund 200 Matrixzugriffe statt sechs. Erwartung: Die Zugzahl je Sekunde bricht um etwa zwei Größenordnungen ein. Bei gleichem **Zeitbudget** kommt die Suche entsprechend weniger weit; bei gleichem **Zugbudget** ist das Ergebnis identisch (die Bewertung ist ja korrekt, nur langsam). Genau diese Unterscheidung ist der Lehrpunkt: Die Implementierung ändert nicht das Verfahren, sondern nur, wie viel davon in das Zeitbudget passt.

9.4 — Der Umschlagpunkt genauer. Sinnvolle Zwischengrößen sind 250, 300, 350 und 400. Zu erwarten ist ein **verwaschener** Übergang, kein scharfer Punkt: Der exakte Solver verschlechtert sich nicht sprunghaft, sondern sein Gap wächst, und irgendwann liegt seine beste gefundene Lösung über der heuristischen. Wo genau das passiert, hängt zusätzlich vom Zeitlimit ab — mit 300 Sekunden statt 30 verschiebt sich der Punkt nach oben. Die belastbare Aussage ist deshalb nie „ab $n = 380$“, sondern immer „ab $n \approx 380$ **bei diesem Zeitbudget**“.

9.5 — Zerstören nach Maß. Die kostenbasierte Zerstörung entfernt gezielt dort, wo etwas zu holen ist, und findet deshalb pro Runde mehr. Sie hat aber zwei Nachteile: Sie muss die Übergangskosten jedes Mal sortieren, und die entfernten Aufträge liegen über den ganzen Plan verstreut — das Teilproblem für CP-SAT ist damit schwieriger als ein zusammenhängendes Fenster gleicher Größe, weil auch alle Einfügestellen offen sind.

Die Aufgabe ist bewusst so gestellt, dass die Antwort **gemessen** werden muss. Wer nur die Ergebniswerte vergleicht, kann Strategie und Rundenzahl nicht trennen. Die saubere Auswertung protokolliert beides: Verbesserungen **je Runde** (Güte der Strategie) und Runden **je Sekunde** (ihr Preis).

Finde den Denkfehler — „Unsere Heuristik ist 6 % besser“

Es fehlt die Schranke.

Alles, was der Kollege sagt, stimmt. Der Fehler liegt darin, was er *nicht* sagt: Die Ersparnis wird gegen die **bisherige Praxis** gemessen, nicht gegen das **Mögliche**. Damit beantwortet die Zahl eine andere Frage als die, die das Management stellt.

`Metaheuristik_vs_Exakt.py` liefert die fehlende Zahl in der Spalte *Schranke*: **1 768 Minuten**. Der Abstand der vorgestellten Lösung zum Bestmöglichen beträgt also bis zu 25 %, nicht 0 %. In derselben Nacht, in der 640 Stunden gefeiert werden, liegen möglicherweise weitere 2 000 Stunden ungenutzt herum.

Die Konsequenz ist keine Absage an das Projekt — 6,2 % sind echt und werden verdient. Die Konsequenz ist eine ehrliche Fortschreibung: „Wir heben 6,2 % und wissen, dass bis zu einem Viertel noch offen ist. Der nächste Schritt ist LNS.“ Wer die Schranke verschweigt, erklärt das Projekt für abgeschlossen und lässt den größeren Teil liegen.

> Dasselbe Muster in anderer Verkleidung: Kapitel Handelsmaschine, `Data_Snooping.py` — dort fehlt nicht die Schranke, sondern die Zahl der Versuche. Beide Male macht eine weggelassene Kennzahl aus einem korrekten Ergebnis eine irreführende Aussage.

Micro-Quiz

1. b) Eine lokale Suche lebt von der Zahl geprüfter Züge. Die volle Summe kostet $O(n)$ statt $O(1)$ — bei 500 Aufträgen zwei Größenordnungen weniger Züge im selben Zeitbudget. Numerisch ist an der vollen Summe nichts falsch (a), und die Konvergenzaussagen zu Simulated Annealing hängen nicht an der Bewertungsfunktion (c).

2. b) Die angenommenen Verschlechterungen summieren sich. 0,29 % von mehreren hunderttausend Zügen sind einige hundert angenommene Verschlechterungen zu je rund 70 Minuten — genug, um die Suche auf das 2,1-fache der Startlösung zu tragen. Die Annahmewahrscheinlichkeit **sinkt** im Lauf (a ist falsch), und numerisch instabil ist nichts (c).

3. b) Die untere Schranke. Sie ist unabhängig davon, wie gut die gefundene Lösung ist, und die einzige verfügbare Aussage darüber, wie viel Luft noch nach oben ist. (c) klingt plausibel, ist hier aber falsch: Die CP-SAT-Lösung war *schlechter* als die Faustregel und damit als Startlösung unbrauchbar.

Selbsttest

1. Kurzsichtig heißt: Sie bewertet einen Schritt nach seinen Kosten, nicht nach dem, was er übrig lässt. Das ist nicht dasselbe wie schlecht — im Schnellstart trifft sie die ersten drei Schritte genau richtig, und bei 500 Aufträgen ist sie besser als ein abgebrochener exakter Lauf.
2. Weil sich beim Umdrehen eines Teilstücks alle Übergänge **innerhalb** des Stücks umkehren. Bei $s_{i,j} = s_{j,i}$ heben sie sich weg und es bleiben zwei geänderte Kanten; bei $s_{i,j} \neq s_{j,i}$ ändern sich alle, und die Bewertung kostet wieder $O(n)$.
3. Die **untere Schranke** — aus einem exakten Lauf mit Zeitlimit. Ohne sie ist „12 % Verbesserung“ eine Aussage über die Vergangenheit, nicht über die Güte.
4. Erstens **Sättigung**: Das Fenster ist zu klein für die nötigen Umbauten — erkennbar an vielen Runden mit sehr wenigen Verbesserungen. Zweitens **Erschöpfung**: Der Plan ist für diese Nachbarschaft austherapiert — erkennbar daran, dass auch ein größeres Fenster nichts mehr findet. Der erste Fall ruft nach einem größeren Fenster, der zweite nach einer anderen Zerstörungsstrategie.
5. Weil er von der Problemstruktur abhängt (wie gut die Relaxation ist, wie stark die Symmetrie), vom Zeitbudget und von der Qualität der Startheuristik. Für die Standortplanung aus Kapitel MILP liegt er woanders als für die Rüstzeiten hier.

A.10 Lösungen zu Kapitel „Spaltengenerierung“

10.1 — Das Abbruchkriterium. Die Eins ist der Zielfunktionskoeffizient eines Musters: Jedes geschnittene Muster verbraucht **genau eine** Mutterrolle, und die Zielfunktion lautet min $\sum_p x_p$. Die reduzierten Kosten einer neuen Spalte sind $c_p - \pi^\top a_p = 1 - \sum_i \pi_i a_{ip}$; sie sind negativ — die Spalte lohnt sich also —, wenn $\sum_i \pi_i a_{ip} > 1$.

Wäre die Zielfunktion eine andere — etwa „minimiere den Verschnitt in Millimetern” —, stünde dort statt der Eins der Verschnitt des Musters. Die Eins ist also kein Zauberwert, sondern schlicht die Kosten der Spalte.

10.2 — Die Startbasis. Ein Muster, das nur eine einzige Breite enthält, ist immer zulässig: $\lfloor W/b_i \rfloor$ Stücke der Breite i passen in jede Rolle. Mit diesen n Mustern lässt sich **jeder** Bedarf decken — notfalls sehr verschwenderisch, aber zulässig.

Eine leere Startmenge wäre ein Problem, weil das Master-LP dann unzulässig ist: Es gibt keine Spalten, mit denen sich die Bedarfszeilen erfüllen ließen. Ohne zulässige Lösung gibt es keine Dualwerte, und ohne Dualwerte kann das Pricing nicht arbeiten. Die Schleife käme nie in Gang.

(In der Literatur behilft man sich alternativ mit künstlichen Variablen und sehr hohen Strafkosten — dieselbe Idee wie die Big-M-Phase des Simplex aus Kapitel 5.)

10.3 — Die Kennzahl prüfen. Zu erwarten ist ein deutlicher, monotoner Zusammenhang: Je mehr Stücke auf eine Rolle passen, desto kleiner die Ersparnis. Bei zwei Stücken je Rolle entscheidet jede Paarung, und die Faustregel verschenkt regelmäßig ganze Rollen; ab acht bis zehn Stücken ist sie praktisch immer optimal.

Der Grund ist ein Verhältnis: Der Verschnitt einer Rolle ist höchstens so groß wie das kleinste nicht mehr passende Stück. Sind die Stücke klein gegen die Rolle, ist auch dieser Rest klein — und der Gesamtverschnitt nähert sich der theoretischen Untergrenze $\lceil \sum_i b_i d_i / W \rceil$, die jede Methode erreichen muss.

10.4 — Dienstplanung. Eine „Spalte” ist ein **vollständiger zulässiger Wochenplan einer Person**: an welchen Tagen sie welche Schicht übernimmt.

- **Ins Master gehört**, was Personen miteinander koppelt: Jede Schicht muss besetzt sein ($\sum_p a_{sp} x_p \geq$ Bedarf je Schicht s), und jede Person bekommt genau einen Plan.
- **Ins Teilproblem gehört**, was innerhalb einer Person gilt: Höchstarbeitszeit, Ruhezeiten, Qualifikation, keine zwei Schichten am selben Tag, höchstens zwei Wochenenden im Monat.

Das ist der eigentliche Gewinn der Zerlegung: Die komplizierten arbeitsrechtlichen Regeln stehen im Teilproblem und werden dort **einmal je Person** geprüft, statt das Master zu überfrachten. Das Teilproblem ist meist ein kürzester Weg über ein Zeit-Zustands-Netz (Kapitel 8) statt eines Rucksacks.

10.5 — Branch-and-Price. Das Problem: Verzweigt man auf einer Mustervariablen ($x_p \leq 3$ gegen $x_p \geq 4$), so ist diese Bedingung im **Pricing** nicht darstellbar. Das Teilproblem erzeugt Muster, es kennt keine Verzweigungsentscheidungen über einzelne Spalten — es könnte im nächsten Schritt genau das ausgeschlossene Muster wieder erzeugen.

Branch-and-Price löst das, indem es auf **Originalgrößen** verzweigt statt auf Spalten: etwa „in wie vielen Rollen kommen Breite i und Breite j gemeinsam vor?” Solche Bedingungen lassen sich ins Pricing-Teilproblem einbauen, weil sie über die Struktur eines Musters sprechen und nicht über seine Identität.

Für das Kapitel genügt der einfachere Weg: das ganzzahlige Master über die erzeugten Spalten. Es ist nicht garantiert optimal, liegt in der Praxis aber fast immer auf oder dicht bei der Schranke — hier exakt darauf (73 gegen 72,92).

Finde den Denkfehler — „Die LP-Lösung sagt 72,92 — also runden wir auf“

Das Aufrunden kostet sechs Rollen — acht Prozent.

Nachgerechnet mit denselben 38 Mustern:

Vorgehen	Rollen
LP-Lösung (13 Muster im Einsatz, gebrochen)	72,92
jedes Muster einzeln aufgerundet	**79**
ganzzahliges Master über dieselben 38 Muster	**73**

Der Fehler steckt in der Vorstellung, „aufrunden” koste höchstens eine Rolle. Aufgerundet wird nicht **eine** Zahl, sondern **dreizehn** — je Muster eine. Jede einzelne Aufrundung kostet bis zu eine Rolle, und in der Summe sind es sechs.

Die Schranke sagt: *Weniger als 73 Rollen sind unmöglich.* Sie sagt **nicht**: *Jede zulässige Lösung liegt höchstens eine Rolle darüber.* Das ist die Verwechslung von unterer Schranke und Gütegarantie — dieselbe Verwechslung wie beim MIP-Gap in Abschnitt 6.8, nur in die andere Richtung.

Was der Kollege stattdessen hätte tun müssen: das Master ein zweites Mal lösen, diesmal mit `integrality=1` über genau die Spalten, die er ohnehin schon erzeugt hat. Das ist eine Zeile, dauert Millisekunden und liefert 73 statt 79. Er hatte alles dafür bereits vorliegen.

 Die allgemeine Lehre Eine gebrochene LP-Lösung ist kein Plan, sondern eine Schranke. Der Weg zur ganzzahligen Lösung führt über den Solver, nicht über `ceil()`.

Micro-Quiz

1. b) Das Mustermodell kennt keine einzelnen Rollen und damit keine austauschbaren Objekte — die Symmetrie verschwindet. (a) trifft zufällig auch zu, ist aber nicht der Grund; (c) ist falsch, die LP-Lösung ist hier gerade **nicht** ganzzahlig (72,92).

2. b) Ein Rucksackproblem: Nutzen sind die Schattenpreise, Gewichte die Breiten, Kapazität die Rollenbreite. (c) beschreibt eine Heuristik — dann wäre das Verfahren nicht mehr exakt, weil man nie sicher wüsste, ob es wirklich kein lohnendes Muster mehr gibt.

3. b) Die untere Schranke von 33,60 beweist, dass mindestens 34 Rollen nötig sind. Damit weiß man, dass die 35 der Faustregel höchstens eine daneben liegen — und kann aufhören zu suchen. Genau diese Aussage kann eine Heuristik allein nie liefern (Kapitel 9).

Selbsttest

1. Freie Antwort. Typische Fälle: identische Maschinen, identische Fahrzeuge, identische Schichten, identische Behälter. Immer dort, wo mehrere gleichartige Objekte indiziert werden, beschreibt jede Lösung sich selbst in vielen Umbenennungen.
2. Eine Spalte ist ein **Schnittmuster**: Sie sagt für jede bestellte Breite, wie viele Stücke davon aus einer Mutterrolle geschnitten werden. Der Vektor a_p hat eine Zeile je Breite.
3. Aus den **Dualwerten des Master-LPs** — den Schattenpreisen der Bedarfszeilen (Abschnitt 5.6). Sie sagen, was ein zusätzliches Stück jeder Breite an Rollen kostet.
4. Weil das Abbruchkriterium eine Aussage über **alle** Muster macht: Das Pricing sucht das beste unter sämtlichen zulässigen Mustern. Findet es keines mit reduzierten Kosten unter null, gibt es auch keines — die LP-Lösung ist über der vollständigen Spaltenmenge optimal.
5. Die **durchschnittliche Stückzahl je Behälter**, also Behältergröße geteilt durch mittlere Stückgröße. Liegt sie bei zwei bis drei, lohnt sich exakte Optimierung; ab sechs bis acht ist eine Faustregel meist schon so gut wie das Optimum.

A.11 Lösungen zu Kapitel „Quadratische und nichtlineare Optimierung — KKT, Lagrange, Konvexität“

11.1 — Konvexität einordnen. PSD (alle ≥ 0) → **konvex**, aber **nicht streng** konvex; die Lösung ist **nicht notwendig eindeutig**. Der Eigenwert 0 bedeutet eine **flache Richtung**: Entlang des zugehörigen Eigenvektors ändert sich der quadratische Term nicht — es gibt eine Rinne statt eines Punktes.

11.2 — Komplementärer Schlupf. Ja, verträglich: $\lambda_i w_i = 0$ gilt für alle i ($w_2 = 0$ mit $\lambda_2 > 0$; die anderen mit $\lambda = 0$). $\lambda_2 = 0{,}03$ bedeutet: Würde man Titel 2 zwingen, ein kleines positives Gewicht zu tragen, verschlechterte sich der Zielwert um 0,03 je Einheit — die Nichtnegativitätsschranke ist dort **bindend**.

11.3 — KKT von Hand. $\mathcal{L} = x_1^2 + x_2^2 + \lambda(4 - x_1 - x_2)$. Stationarität: $2x_1 = \lambda$, $2x_2 = \lambda \rightarrow x_1 = x_2$. Bindend ($\lambda > 0$): $x_1 + x_2 = 4 \rightarrow x_1 = x_2 = 2$, $\lambda = 4$. Prüfung: $\lambda = 4 > 0$ □, $f = 8$. **Interpretation:** Würde die Forderung auf $\geq 4{,}1$ steigen, stiege f um etwa $4 \cdot 0{,}1 = 0{,}4$. Probe: $2 \cdot (2{,}05)^2 = 8{,}405$ □.

11.4 — Unmögliche Korrelationsmatrix. Eigenwerte $\approx (-0{,}62;\ 1{,}0;\ 2{,}62)$ → **nicht PSD**, also unmöglich. Anschaulich: Wenn A stark **positiv** mit B korreliert und B stark positiv mit C, kann A nicht gleichzeitig stark **negativ** mit C korrelieren — Korrelation ist eingeschränkt transitiv. Formal: $\rho_{AC} \geq \rho_{AB}\rho_{BC} - \sqrt{(1-\rho_{AB}^2)(1-\rho_{BC}^2)} = 0{,}81 - 0{,}19 = 0{,}62$; verlangt waren $-0{,}9$.

11.5 — Gewichtung untersuchen. Erwartetes Muster: Größeres α → mehr Rendite, mehr Konzentration im Titel mit höchstem μ. Größeres β → gleichmäßigere Gewichte, Entropie steigt Richtung $\ln 4 = 1{,}386$. Ab etwa $\beta \approx 0{,}1$ dominiert die Entropie und man nähert sich der Gleichgewichtung. **Empfehlung an einen Ausschuss:** Nicht mit β argumentieren, sondern mit der resultierenden **Maximalposition** — „mit dieser Einstellung liegt keine Position über 40 %“ ist verständlich, „$\beta = 0{,}015$“ nicht.

11.6 — Nicht-Konvexität demonstrieren. Mit der bewusst fehlerhaft konstruierten Matrix finden 20 Startpunkte typischerweise mehrere verschiedene Optima; die „Portfoliovarianz“ $w^\top \Sigma w$ kann bei geeig-

neten Gewichten negativ werden (der zugehörige Eigenvektor liegt allerdings teilweise außerhalb des zulässigen Bereichs $w \geq 0{,}001$, $\sum w = 1$ — deshalb fällt der Fehler bei naiver Prüfung nicht auf).

Finde den Denkfehler — Die Kovarianzmatrix aus dem Controlling

(a) **Ohne zu rechnen.** Stellen Sie sich die drei Anlagen als drei Personen vor, die nebeneinander gehen. A und B gehen fast im Gleichschritt (0,9). B und C gehen fast im Gleichschritt (0,9). Können A und C dann in *entgegengesetzte* Richtungen laufen? Nein — wenn A B folgt und B C folgt, muss A ungefähr auch C folgen. Korrelation ist zwar nicht transitiv im strengen Sinne, aber sie ist **eingeschränkt**: Zwei starke Bindungen zwingen die dritte in einen engen Bereich.

Genau beziffern lässt sich das. Damit eine 3×3-Korrelationsmatrix überhaupt existieren kann, muss gelten:

$$\rho_{AC} \in \left[\rho_{AB}\rho_{BC} - \sqrt{(1-\rho_{AB}^2)(1-\rho_{BC}^2)},\ \rho_{AB}\rho_{BC} + \sqrt{(1-\rho_{AB}^2)(1-\rho_{BC}^2)} \right]$$

Mit $\rho_{AB} = \rho_{BC} = 0{,}9$ ergibt das $[0{,}620;\ 1{,}000]$. Der angesetzte Wert $-0{,}9$ liegt weit außerhalb. **Diese drei Zahlen können nicht gleichzeitig zutreffen** — mindestens eine der drei Quartalsauswertungen misst etwas anderes, als der Analyst annimmt (anderer Zeitraum, andere Frequenz, andere Bereinigung).

(b) **Die Eigenwerte** von **R** lauten $(-0{,}8;\ 1{,}9;\ 1{,}9)$. Ein **negativer** Eigenwert bedeutet: Die Matrix ist nicht positiv semidefinit. Es gibt eine Richtung im Gewichtsraum — hier ungefähr $(0{,}58;\ -0{,}58;\ 0{,}58)$, der Eigenvektor zu $-0{,}8$ —, entlang derer die quadratische Form **negativ** wird. Genau diese Richtung hat `scipy` gefunden: Das Ergebnis $(1{,}5;\ -2{,}0;\ 1{,}5)$ ist ein Vielfaches davon.

Der Test dafür ist eine Zeile und gehört vor jede Risikorechnung:

```
eigenwerte = np.linalg.eigvalsh(S)
assert eigenwerte.min() >= -1e-10, \
    f"Kovarianzmatrix nicht positiv semidefinit: kleinster Eigenwert {eigenwerte.min():.3f}"
```

(c) **Warum das kein Rechenfehler ist.** `scipy` hat exakt die Aufgabe gelöst, die man ihm gestellt hat: „minimiere $\mathbf{w}^\top\mathbf{S}\mathbf{w}$“. Dass dieser Ausdruck bei nicht positiv semidefinitem **S** beliebig negativ werden kann, ist eine Eigenschaft der **Eingabe**, nicht des Verfahrens. `success: True` heißt hier — wie in Abschnitt 11.6 — nur: „Ich bin an einer Stelle angekommen, an der es nicht weiter bergab geht.“ Dass es überhaupt bis $-0{,}254$ bergab ging, lag allein an den Schranken ± 2; ohne sie liefe das Verfahren gegen $-\infty$.

Eine Varianz ist definitionsgemäß eine erwartete quadratische Abweichung und damit ≥ 0. Ein negativer Wert ist kein „besonders gutes Portfolio“, sondern der Beweis, dass die Eingabegrößen keine gemeinsame Wirklichkeit beschreiben.

(d) **Warum die Fehlermeldung das Wertvollste war.** CVXPY prüft vor dem Rechnen, ob das Problem konvex ist. Bei einem nicht positiv semidefiniten **S** ist `quad_form(w, S)` es nicht — und CVXPY verweigert die Arbeit, statt eine bedeutungslose Zahl zu liefern. Der `DCPError` war also keine Einschränkung der Bibliothek, sondern eine **Diagnose**: Er hat auf einen Datenfehler hingewiesen, den drei Quartalsberichte und ein Analyst übersehen hatten.

Der Ausweg auf `scipy` hat das Warnsystem umgangen, nicht das Problem gelöst. Richtig ist:

1. **Prüfen**, ob die Matrix positiv semidefinit ist (Zeile aus (b)) — beim Einlesen, nicht beim Optimieren.
2. **Die Ursache suchen.** Korrelationen aus verschiedenen Quellen, Zeiträumen oder Frequenzen sind fast nie widerspruchsfrei. Schätzen Sie die Matrix aus **einem** konsistenten Datensatz.
3. Ist das nicht möglich, die Matrix **reparieren**: negative Eigenwerte auf null setzen und rekonstruieren (*Eigenwert-Clipping*, siehe die Aufgabe *Konvexität einer Kovarianzmatrix reparieren* in Kapitel 2) oder gleich einen Shrinkage-Schätzer verwenden, der positive Semidefinitheit garantiert (Kapitel 18).

   Der Merksatz dazu: Eine Kovarianzmatrix ist kein Behälter für einzeln geschätzte Zahlen, sondern ein geometrisches Objekt. Nicht jede Kombination von Korrelationen existiert.

Micro-Quiz

1 — (b). Der `DCPError` ist eine inhaltliche Aussage: Für diese Formulierung kann es keine Optimalitätsgarantie geben. Auf `scipy` auszuweichen (a) beseitigt die Meldung, nicht die Ursache — man bekommt dann ein lokales Ergebnis ohne Garantie und ohne Warnung, wie Abschnitt 11.8 zeigt. Toleranzen (c) haben mit Konvexität nichts zu tun. Richtig ist, zwischen zwei bewussten Wegen zu wählen: konvex umformulieren, oder lokal rechnen **und** das im Bericht kenntlich machen.

2 — (b) ein lokales Minimum. `success: True` beschreibt die Konvergenz des Verfahrens, nicht die Qualität des Ergebnisses. Bei Mengenrabatten ist die Zielfunktion nicht konvex, also kann es mehrere lokale Minima geben — im Kapitelbeispiel fünf, mit 8 % Spanne. (a) wäre nur bei einem konvexen Problem richtig. (c) ist zu pessimistisch: Ein lokales Minimum ist es sehr wohl, die Zulässigkeit wird von SLSQP eingehalten.

3 — (c) Schattenpreis. λ^* ist die Ableitung des optimalen Zielwerts nach der rechten Seite der Nebenbedingung — dieselbe Bedeutung wie der Dualwert im LP (Kapitel 5), nur für allgemeine, auch krumme Nebenbedingungen. (a) und (b) verwechseln den Multiplikator mit einer Verletzungszahl beziehungsweise mit dem Schlupf; der Schlupf ist bei einer bindenden Bedingung gerade **null** (komplementärer Schlupf).

Selbsttest

1. Semidefinit → konvex, Optimum global, aber evtl. nicht eindeutig. Definit → streng konvex, Optimum eindeutig.
2. Am Optimum heben sich der „Zug“ der Zielfunktion und der „Druck“ der bindenden Nebenbedingungen exakt auf.
3. Weil eine Ungleichungsschranke nur in eine Richtung wirken kann („drücken“, nicht „ziehen“). Ein negativer Multiplikator hieße, die Bedingung sei gar nicht bindend.
4. $\Sigma = \mathbf{DCD}$ mit Volatilitätsdiagonalmatrix $\mathbf{D}$ und gültiger (PSD) Korrelationsmatrix $\mathbf{C}$.
5. Weil es nur lokale Optima findet; übereinstimmende Ergebnisse aus vielen Startpunkten sind ein Indiz (kein Beweis) für Konvexität.

A.12 Lösungen zu Kapitel „Optimierung unter Unsicherheit — Monte-Carlo, Stochastik, Robustheit“

12.1 — Fluch des Durchschnitts. Weil sich Verzögerungen **fortpflanzen** und die Verteilung rechtsschief ist: Ein Gewerk, das früher fertig wird, beschleunigt selten den Gesamtablauf (der nächste Schritt ist noch nicht bereit), ein verspätetes verzögert alles. Weitere Beispiele: Wartezeiten in Warteschlangen (nichtlinear in der Auslastung), Projektkosten mit Nachträgen.

12.2 — Ansatz wählen. (a) robust (Ausfall ist existenziell, Wahrscheinlichkeiten unzuverlässig). (b) stochastisch (Verteilung bekannt). (c) robust bzw. Extremwertstatistik (seltene Ereignisse, katastrophale Folgen). (d) stochastisch (Szenarien mit Wahrscheinlichkeiten liegen vor). (e) Chance Constraint — die Zusage ist bereits als Quote formuliert („an 99 % aller Wintertage“). Robust wäre hier zu teuer (es gibt immer einen kälteren Tag), der Erwartungswert zu schwach (er sagt nichts über die Zusage).

12.3 — Zweistufig rechnen. (a) Mit Spot = 60: Ableitung bei $x \in (100, 250)$: $40 - 60 \cdot 0{,}5 + 5 \cdot 0{,}5 = 40 - 30 + 2{,}5 = +12{,}5 > 0 \rightarrow$ Kapazität **senken**. Bei $x < 100$: $40 - 60 = -20 < 0 \rightarrow$ erhöhen. Optimum daher $x^* = 100$. (b) Billige Nachbesserung macht Vorhalten unattraktiv. (c) Bei $x = 225$ optimal müsste die Ableitung dort das Vorzeichen wechseln — das ist bei diskreten Szenarien nur zufällig der Fall; der Mittelwert ist nur bei symmetrischen Kosten und stetiger Verteilung optimal.

12.4 — EVPI interpretieren. Selbst eine **perfekte** Prognose wäre nur 8 400 € pro Jahr wert. Eine Lösung für 15 000 € kann sich also **niemals** rechnen — unabhängig von ihrer Güte. Argument: „Der theoretische Maximalnutzen liegt unter dem Preis.“

12.5 — Monte-Carlo erweitern. (a) Servicelevel 95 %: Kapazität = 95 %-Quantil des Bedarfs (ca. 485). Die Zusatzkosten gegenüber dem Kostenoptimum (180) betragen ein Vielfaches — Servicelevel ist teuer, und genau diese Zahl braucht die Geschäftsleitung für die Entscheidung. (b) `cvar = kosten[kosten >= np.quantile(kosten, 0.95)].mean()`. (c) Erwartetes Bild: Bei kleiner Kapazität ist die Kostenverteilung stark rechtsschief (seltene, teure Notzukäufe); bei großer Kapazität schmal und nach rechts verschoben.

12.6 — Den Preis der Zusage selbst bestimmen. (a) Die Kosten je Prozentpunkt steigen weiter steil an; der Mix verschiebt sich dabei fast nur noch zugunsten des Gaskraftwerks, weil Biomasse längst an ihrer Ausbaugrenze liegt. Wind und Sonne bleiben bei einem kleinen Rest — sie senken über ihre Gegenläufigkeit die Norm, tragen zur gesicherten Leistung aber kaum bei. (b) Der Unterschied ist **nicht** numerisch, sondern inhaltlich. Bei 200 MW Biomasse liefert selbst der **Vollausbau** in der Kältewelle nur rund 440 MW — 7,5 % der Szenarien sind schlicht nicht bedienbar, egal wie viel Geld man ausgibt. Das Szenariomodell sieht diese Fälle in den Daten und meldet korrekt `infeasible`, sobald die Zusage über 92 % steigt. Das analytische Modell kennt sie nicht: Für eine Normalverteilung ist jede Zusage unter 100 % erfüllbar, wenn man nur genug Streuung wegkauft. Es meldet `optimal` — und der Plan hält gemessen 87,4 %. Glauben sollte man dem Szenariomodell: Ein `infeasible` ist hier die **richtige** Antwort. Es sagt, dass die Zusage nicht am Budget scheitert, sondern am Kraftwerkspark, und dass nicht mehr Geld hilft, sondern nur eine andere Anlage oder eine kleinere Zusage. (c) Sinngemäß: „99,99 % kosten nicht 5 % mehr als 99 %, sondern ein Vielfaches — der letzte Prozentpunkt ist bereits 4,5-mal so teuer wie der erste. Sagen Sie mir, was ein Ausfalltag das Unternehmen kostet, dann rechne ich aus, welche Quote sich lohnt. Und die Zusage gilt nur für die Wetterlagen, die wir modelliert haben — die Kältewelle gehört hinein.“

12.7 — Budgeted Uncertainty.

```
abzug = cp.sum_largest(cp.multiply(UNSICHERHEIT, w), Gamma)
ziel = cp.Maximize(MU_SCHAETZUNG @ w - abzug - 0.5*LAMBDA*cp.quad_form(w, SIGMA))
```

$\Gamma = 0$ entspricht dem nominalen Fall, $\Gamma = n$ dem vollen Worst Case. Dazwischen steuert Γ die Vorsicht stufenlos — der praktisch nützlichste Bereich liegt meist bei $\Gamma \approx \sqrt{n}$.

Finde den Denkfehler — Warum jedes Projekt zu spät fertig wird

(a) **Warum 14,38 statt 11,34 Tage.** Der Projektleiter hat den Mittelwert der **Einzeldauer** berechnet und ihn für den Mittelwert der **Projektdauer** gehalten. Das sind zwei verschiedene Größen:

$$\mathbb{E}\big[\max(D_1, \dots, D_5)\big] \;\neq\; \max\big(\mathbb{E}[D_1], \dots, \mathbb{E}[D_5]\big)$$

Anschaulich: Das Projekt ist fertig, wenn das **letzte** Gewerk fertig ist. Ein Gewerk, das statt 11 nur 7 Tage braucht, bringt niemandem etwas — die anderen sind ja noch dran. Ein Gewerk, das 17 Tage braucht, hält alle auf. Die guten Ausreißer verpuffen, die schlechten schlagen voll durch.

Damit gilt für das Maximum immer $\mathbb{E}[\max] \geq \max(\mathbb{E})$. Die Differenz ist kein Schätzfehler, sondern eine **systematische Verzerrung** — sie geht immer in dieselbe Richtung, und sie wächst mit der Streuung der Einzelschätzungen.

(b) **Warum 95,5 %.** Damit das Projekt die geplanten 11,33 Tage hält, müssen **alle fünf** Gewerke gleichzeitig ihren Mittelwert unterbieten. Für die Dreiecksverteilung $(6;\ 10;\ 18)$ ist die Wahrscheinlichkeit dafür je Gewerk

$$F(11{,}33) = 1 - \frac{(18 - 11{,}33)^2}{(18-6)(18-10)} = 1 - \frac{44{,}4}{96} = 0{,}537$$

— knapp über der Hälfte, weil die Verteilung rechtsschief ist und der Mittelwert deshalb über dem wahrscheinlichsten Wert (10) liegt. Für alle fünf zusammen bleibt davon:

$$0{,}537^5 = 0{,}045 \quad \Rightarrow \quad \textbf{95,5 \% reißen den Plan.}$$

Genau der Wert, den die Simulation misst. Sehen Sie sich den Exponenten an: Es ist die **Zahl der parallelen Stränge**, die die Erfolgswahrscheinlichkeit auffrisst. Bei einem einzigen Gewerk wären es 53,7 %, bei fünf nur noch 4,5 %.

Damit es 50 % wären, müsste man den **Median der Projektdauer** einplanen, nicht den Mittelwert der Einzeldauern.

Das ist die eigentliche Lehre: Eine Terminzusage ist immer eine Aussage über ein **Quantil**, nie über einen Mittelwert.

(c) **Bei zehn Gewerken** steigt die erwartete Projektdauer auf **15,35 Tage** (90-%-Quantil: 17,00). Der Zuwachs von fünf auf zehn Gewerke ist mit knapp einem Tag deutlich kleiner als der von einem auf

fünf — das Maximum wächst logarithmisch, nicht linear. Aber es wächst, und zwar allein durch das Hinzufügen paralleler Arbeit, ohne dass ein einziges Gewerk langsamer geworden wäre.

Praktische Konsequenz: **Mehr Parallelität macht Projekte nicht nur nicht schneller, sie macht die Terminprognose schlechter.** Wer ein Projekt in mehr parallele Stränge zerlegt, muss den Puffer vergrößern, nicht verkleinern.

(d) **Die Zahl für den Projektplan.** Bei einer geforderten Sicherheit von 90 % gehört das **90-%-Quantil** in den Plan: **16,6 Tage**, aufgerundet 17. Das sind 47 % mehr als die ursprünglich geplanten 11,3 Tage.

```
puffer = np.quantile(projekt, 0.90)
print(f"Termin mit 90 % Sicherheit: {np.ceil(puffer):.0f} Tage")
```

Und die Formulierung gegenüber dem Auftraggeber lautet nicht „der Umbau dauert 17 Tage“, sondern: *„In 9 von 10 Fällen sind wir nach 17 Tagen fertig; im Mittel nach 14,4.“* Diese zwei Zahlen sind eine belastbare Zusage — eine einzelne ist es nie.

Micro-Quiz

1 — (b). Das kritische Verhältnis $1400/(1400+120) = 0{,}921$ gibt an, wie weit man sich auf die günstigere Fehlerseite stellen soll: Bestellt wird das 92,1-%-Quantil des Bedarfs, hier 38 Stück. (a) ignoriert die Kostenasymmetrie und kostet im Kapitelbeispiel 160 % mehr. (c) verwechselt die Fragestellung: Die Kapitalbindung *ist* bereits in den 120 € je überzähligem Stück enthalten — sie rechtfertigt keine zusätzliche Kürzung.

2 — (b). Der Unterschied liegt darin, **was man wissen muss**: Stochastische Optimierung setzt eine Wahrscheinlichkeitsverteilung voraus und minimiert den Erwartungswert; robuste Optimierung kommt mit einer bloßen Bandbreite aus und sichert den ungünstigsten Fall darin ab. Keines ist „genauer“ (a) — sie beantworten verschiedene Fragen. (c) trifft es nicht: Robuste Optimierung betrachtet nicht mehr Szenarien, sondern eine ganze Menge auf einmal, und interessiert sich darin nur für den schlechtesten Punkt.

3 — (c). Bei durchgehender Linearität und ohne nachgelagerte Entscheidung gilt $\mathbb{E}[f(X)] = f(\mathbb{E}[X])$ — dann ist Rechnen mit Mittelwerten korrekt. Sobald aber ein Maximum, ein Minimum, ein Betrag oder eine Nachbesserungsentscheidung auftaucht, gilt das nicht mehr: Genau das sind die beiden Fälle aus Schnellstart (asymmetrische Kosten über `maximum`) und Abschnitt 12.9 (Maximum über parallele Vorgänge). (a) ist die Fehlannahme, um die es im ganzen Kapitel geht; (b) ist zu absolut — es gibt den linearen Fall, in dem es tatsächlich zulässig ist.

Selbsttest

1. Wegen der Jensenschen Ungleichung: Bei konvexen Kosten ist $\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$ — Planung mit dem Mittelwert unterschätzt die Kosten systematisch.
2. Stufe 1 wird **vor** Bekanntwerden des Zufalls festgelegt (eine Entscheidung); Stufe 2 **danach**, je Szenario individuell.
3. Die Differenz zwischen Kosten unter Unsicherheit und Kosten bei perfektem Wissen — eine **Obergrenze** für den Wert jeder Prognoseverbesserung.
4. Weil sie nur eine **Menge** möglicher Werte braucht, nicht deren Verteilung — sie optimiert gegen das schlechteste Element dieser Menge.

5. Für einfache Mengen (Box, Ellipsoid) lässt sich das innere Maximum geschlossen ausrechnen und wird zu einem Abzugsterm; das Gesamtproblem bleibt konvex.
6. Weil die Streuung der Summe $\mathbf{a}^\top \mathbf{x}$ nicht die Summe der Streuungen ist: $\sqrt{\mathbf{x}^\top \Sigma \mathbf{x}}$ enthält die **Kovarianzen**. Ein fester Zuschlag je Variable wäre linear und würde deshalb übersehen, dass sich gegenläufige Größen teilweise aufheben — die Norm belohnt Mischung, ein linearer Aufschlag nicht.
7. Weil die Zusage nur so gut ist wie die unterstellte Verteilung. Die Normalverteilung kennt keine Ereignisse, in denen alle Quellen **gleichzeitig** ausfallen; ihre schwachen Korrelationen unterschätzen genau den Fall, der die Zusage bricht. Die szenariobasierte Formulierung hat diese Fälle in den Daten und trifft die Quote — erkauft mit Mehrkosten und mit dem Risiko, sich an die verwendete Stichprobe anzupassen.

A.13 Lösungen zu Kapitel „Dynamische Programmierung — Die Bellman-Gleichung und Order-Execution“

13.1 — Bausteine. Stufe = Tag 1–5; Zustand = verbleibende Distanz (ggf. plus Erschöpfungsgrad); Aktion = heutige Tagesetappe; Wertfunktion = minimale Restanstrengung. **Nicht** in den Zustand gehören: bereits gelaufene Kilometer (redundant, wenn die Restdistanz bekannt ist) und das Wetter von gestern (ohne Einfluss auf die Zukunft).

13.2 — Optimalitätsprinzip. Weil der Wert eines Zustands nur von den *künftigen* Entscheidungen abhängt, kann man ihn ab dem Ende rekursiv berechnen. Hängen die Kosten zusätzlich von **früheren** Aktionen ab, ist die Markov-Eigenschaft verletzt — die Lösung: die relevante Vergangenheit **in den Zustand aufnehmen** (dann wächst allerdings der Zustandsraum).

13.3 — Rückwärtsinduktion. Mit $C(n) = n^2 + 2n$ und 5 Einheiten in 3 Perioden ergibt sich als optimaler Pfad $2 \to 2 \to 1$ (oder eine Permutation davon) mit Gesamtkosten $8 + 8 + 3 = 19$. Alles auf einmal kostet $25 + 10 = 35$.

13.4 — Rucksack als DP.

```
V = np.zeros((n + 1, kapazitaet + 1))
for i in range(1, n + 1):
    for c in range(kapazitaet + 1):
        V[i][c] = V[i-1][c]
        if gewicht[i-1] <= c:
            V[i][c] = max(V[i][c], V[i-1][c - gewicht[i-1]] + nutzen[i-1])
```

Ergebnis identisch zum MILP (110). Laufzeit $O(n \cdot C)$ — bei kleinen ganzzahligen Kapazitäten schneller als Branch-and-Bound, bei großen oder nicht-ganzzahligen Kapazitäten schlechter (dann ist das MILP überlegen). Man nennt das **pseudopolynomiell**.

13.5 — Risikoaversion kalibrieren. (a) Zwischen $\lambda = 10^{-5}$ (41 %) und 10^{-4} (78 %); genauer etwa bei $\lambda \approx 3 \cdot 10^{-5}$. (b) Höheres $\lambda \to$ höhere Gesamtkosten (man kauft Sicherheit mit Slippage). (c) Iterativ: λ so wählen, dass der Anteil der Marktauswirkungskosten an den Gesamtkosten bei 20 % liegt — im Programm ausrechnen und per Bisektion einstellen.

13.6 — Zustandsraum erweitern. Der Zustand wird zu (Restbestand, Orderbuchtiefe); der Zustandsraum verdreifacht sich, die Rechenzeit ebenfalls. Die Strategie wird **zustandsabhängig**: Bei tiefem Buch wird mehr verkauft, bei dünnem gewartet. Genau diese Adaptivität ist der Mehrwert von DP gegenüber einem festen Pfad.

Finde den Denkfehler — Der Zustand, der zu wenig weiß

(a) **Die fehlende Information.** Die Zeile

```
kosten = (RUESTKOSTEN if p > 0 else 0) + ...
```

berechnet die Rüstkosten allein daraus, ob in *dieser* Periode produziert wird. Die tatsächliche Regel lautet aber: Rüstkosten fallen an, wenn produziert wird **und in der Vorperiode nicht**. Ob in der Vorperiode produziert wurde, steht nirgends im Zustand `(t, lager)` — die Funktion kann es gar nicht wissen.

Das Modell nimmt deshalb an, dass **jede** Produktionsperiode eine Rüstung kostet. Es bestraft laufende Produktion, obwohl sie in Wirklichkeit gratis weiterläuft.

(b) **Der unsichtbare Vorteil.** Der Plan $(3, 1, 4, 2)$ produziert in **allen vier** Perioden. Real fällt dafür genau **eine** Rüstung an (in Periode 0), zusammen 30 €. Das falsche Modell rechnet mit vier Rüstungen, also 120 € — und verwirft den Plan als viel zu teuer. Genau der Vorteil, den dieser Plan hat, ist im Modell nicht darstellbar.

Stattdessen wählt es $(5, 0, 5, 0)$: wenige Produktionsperioden, dafür große Lose. Aus Sicht seiner eigenen Kostenannahme ist das folgerichtig.

Plan	Rüstungen real	Stückkosten	Lagerkosten	**Gesamt real**
$(5, 0, 5, 0)$ — Modellwahl	$2 \times 30 = 60$ €	40 €	10 €	**110 €**
$(3, 1, 4, 2)$ — wahres Optimum	$1 \times 30 = 30$ €	40 €	0 €	**70 €**

57 % Mehrkosten — nicht durch einen Rechenfehler, sondern durch einen zu kleinen Zustand.

(c) **Der korrigierte Zustand** lautet `(t, lager, lief_vorher)` mit einem zusätzlichen Wahrheitswert:

```
@functools.lru_cache(None)
def V(t, lager, lief_vorher):
    ...
    kosten = (RUESTKOSTEN if (p > 0 and not lief_vorher) else 0) + ...
    rest, plan = V(t + 1, neu, p > 0)
```

Damit findet die Rückwärtsinduktion $(3, 1, 4, 2)$ für 70 € — das nachgerechnete Optimum. Der Zustandsraum **verdoppelt** sich (je Lagerstand zwei Varianten, „lief“ und „lief nicht“). Das ist der übliche Preis: Vollständigkeit des Zustands kostet Größe, und genau hier beginnt der Fluch der Dimensionalität.

(d) **Warum der richtige Betrag das Tückische ist.** Man würde erwarten, dass ein falsches Modell auch einen falschen Kostenbetrag ausgibt — dann fiele es beim Nachrechnen auf. Hier nicht: Für

den von ihm gewählten Plan $(5, 0, 5, 0)$ stimmt die Rechnung zufällig, weil in diesem Plan tatsächlich jede Produktionsperiode auf eine Pause folgt. Modellannahme und Wirklichkeit fallen für **genau diese eine Lösung** zusammen.

Eine Prüfung, die nur den ausgegebenen Plan nachrechnet, bestätigt also alles. Der Fehler liegt nicht in der Bewertung der gefundenen Lösung, sondern darin, dass die **bessere Lösung nie in Betracht gezogen wurde**. Solche Fehler findet man nur auf zwei Wegen:

1. **Die Zustandsprobe:** *„Wenn ich nur den Zustand kenne und nicht den Weg dorthin — kann ich dann noch richtig weiterentscheiden?“* Hier lautet die Antwort nein, denn ohne zu wissen, ob gerade produziert wird, sind die Kosten der nächsten Aktion nicht bestimmbar.
2. **Ein zweites, unabhängiges Verfahren.** Bei kleinen Instanzen genügt vollständige Enumeration: $6^4 = 1296$ Pläne sind in Millisekunden durchgerechnet — und hätten die 70 €-Lösung sofort gezeigt.

Micro-Quiz

1 — (b). V_t setzt V_{t+1} voraus: Eine Entscheidung lässt sich erst bewerten, wenn feststeht, was sie für die Zukunft bedeutet. Am Ende ist dieser Wert bekannt ($V_T = g$), deshalb beginnt die Rechnung dort. (a) verwechselt ein mathematisches Erfordernis mit einer Implementierungsfrage; (c) ist frei erfunden und hätte mit der Rechenrichtung ohnehin nichts zu tun.

2 — (b). Das Optimalitätsprinzip sagt, dass jedes **Reststück** einer optimalen Lösung selbst optimal ist. Deshalb genügt es, je Zustand einen einzigen Wert zu speichern, statt alle Wege dorthin zu unterscheiden. (a) ist die verbreitetste Fehldeutung — sie beschriebe gieriges Vorgehen, und genau das scheitert im Schnellstart dieses Kapitels (21 statt 14 Minuten). (c) ist eine Allgemeinplatz-Aussage ohne Bezug zum Prinzip.

3 — (b). Der Rabatt hängt von der bisher kumulierten Menge ab; ohne diese Größe im Zustand kann das Modell die Kosten der nächsten Bestellung nicht bestimmen — dasselbe Muster wie in Abschnitt 13.7. (a) übersieht genau das: Kosten, die von der Vorgeschichte abhängen, **sind** Zustandsinformation. (c) ist zu absolut — DP bleibt anwendbar, der Zustandsraum wird nur größer; ob sich stattdessen ein MILP lohnt, ist eine Frage der Größe, nicht der Eignung.

Selbsttest

1. Jeder Teilabschnitt einer optimalen Strategie ist selbst optimal für den Zustand, in dem er beginnt.
2. Weil V_{t+1} bekannt sein muss, um V_t zu berechnen — und am Ende ist der Wert bekannt.
3. Alles, was die Zukunft beeinflusst, und nichts weiter. Unvollständig ist er, wenn die Kosten oder Übergänge zusätzlich von der Vorgeschichte abhängen.
4. Der Aufwand wächst multiplikativ mit jeder Zustandsdimension. Gegenmittel: gröbere Diskretisierung/Zustandsreduktion, Funktionsapproximation (ADP), Reinforcement Learning.
5. Weil sie eine **unabhängige** Prüfung ermöglicht — Größenordnungsfehler und Vorzeichenfehler fallen sofort auf.

A.14 Lösungen zu Kapitel „Mehrere Ziele — Pareto-Fronten statt Gewichte“

14.1 — Dominanz prüfen. P (14 000 €, 6 000 kg) und Q (13 800 €, 6 100 kg): **keiner dominiert den anderen.** Q ist billiger, P ist sauberer — beide sind pareto-optimal, die Wahl ist eine Präferenzfrage.

R (14 000 €, 6 100 kg) dagegen wird **von beiden dominiert**: gleich teuer wie P, aber schmutziger; schmutziger als Q *und* teurer. R kann man wegwerfen, ohne jemanden zu fragen — und genau das ist der Zweck des Dominanzbegriffs: Er trennt die Fälle, in denen der Modellierer entscheiden darf, von denen, in denen er es nicht darf.

14.2 — Der Schattenpreis in der Praxis. Man geht die Front von unten durch, solange der Schattenpreis unter 0,90 €/kg liegt. Aus der Ausgabe: 0,58 — 0,93 — 0,40 — 0,59 — 0,77 — 0,65 — 0,77 — 1,09 — 1,32. Die Werte steigen nicht monoton, weil die Front bei ganzzahligen Modellen keine glatte Kurve ist; maßgeblich ist deshalb nicht der einzelne Schritt, sondern der **Gesamtschattenpreis** gegenüber dem Kostenminimum.

Der letzte Punkt, dessen Gesamtschattenpreis unter 0,90 € liegt, ist **14 189 € / 5 632 kg**: 646 € Aufpreis für 843 kg Ersparnis, also 0,77 €/kg. Der nächste Punkt kostet 1,09 €/kg und liegt damit über dem internen Preis.

Begründung in einem Satz: *„Für 646 Euro sparen wir 843 Kilogramm CO□ — zu 77 Cent je Kilogramm, während wir intern mit 90 Cent rechnen.“*

14.3 — Mehr Trassen. Mit 8 statt 5 Trassen wird der Engpass schwächer. Zu erwarten ist eine **kürzere** Front: Der Zielkonflikt entsteht ja allein aus der Knappheit; je mehr Trassen, desto näher rücken Kostenminimum und CO□-Minimum zusammen. Im Grenzfall genügend vieler Trassen fallen beide zusammen — die Bahn ist billiger *und* sauberer, also gibt es gar keinen Konflikt mehr und die Front schrumpft auf einen Punkt.

Der Lehrpunkt: **Ein Zielkonflikt ist keine Eigenschaft der Ziele, sondern der Knappheit.** Wer ihn auflösen will, sollte zuerst prüfen, ob sich die Ressource vermehren lässt, statt über Gewichte zu verhandeln.

14.4 — Die Front der Relaxation. Ohne `integrality` ist der zulässige Bereich konvex, und die Pareto-Front liegt vollständig auf ihrer eigenen unteren konvexen Hülle. Es gibt also **keinen** Punkt oberhalb — jeder Punkt der Front ist durch ein passendes Gewicht erreichbar.

Genau das ist der Grund, warum das Problem in Lehrbüchern zur linearen Programmierung nicht vorkommt und in der betrieblichen Praxis ständig: Sobald „welcher Träger“, „welches Lager“ oder „welche Schicht“ entschieden wird, zerfällt der Bereich in einzelne Punkte, und zwischen ihnen entstehen die Einbuchtungen, in denen die nicht gestützten Lösungen liegen.

14.5 — Drei Ziele. Die Front wird zu einer Fläche im dreidimensionalen Zielraum. Das ε-Verfahren braucht dann ein **Gitter** über zwei Schranken statt einer Folge über eine — der Aufwand wächst von $O(k)$ auf $O(k^2)$, bei m Zielen auf $O(k^{m-1})$.

Zwei praktische Folgerungen: Erstens ist bei drei und mehr Zielen die vollständige Front meist nicht mehr bezahlbar; man arbeitet dann mit einer Stichprobe oder mit Metaheuristiken, die eine Front approximieren (NSGA-II, in `pymoo` enthalten). Zweitens ist eine Front mit hunderten Punkten für die Entscheidung

ohnehin unbrauchbar — ab drei Zielen ist die bessere Frage meist, ob sich zwei davon zusammenfassen lassen.

Finde den Denkfehler — „Wir haben die Gewichte sauber kalibriert“

Der Lösungsraum wurde nicht vollständig abgetastet — und zwar aus prinzipiellen Gründen.

500 Gewichte sind nicht zu wenige. Auch 500 000 wären zu wenige. Die gewichtete Summe kann ausschließlich Punkte auf der unteren konvexen Hülle erreichen; alles, was darüber liegt, ist für **jedes** w unerreichbar. Im Kapitelbeispiel sind das 4 von 10 Punkten — 40 % der Alternativen, die die Geschäftsführung nie zu sehen bekam.

Die Behauptung „die Entscheidung hat der Fachbereich getroffen“ ist damit nur halb wahr. Der Fachbereich hat aus einer **vorgefilterten** Auswahl gewählt, und die Vorfilterung stammt vom Team — unbeabsichtigt und unbemerkt, aber sie stammt von dort.

Zum zweiten Teil: Nein, niemand hätte es bemerkt.

Das ist der eigentlich beunruhigende Punkt. Alle vorgelegten Pläne waren pareto-optimal, also einwandfrei. Es fehlte kein *schlechter* Plan, sondern es fehlten *gute*. Ein fehlender Kompromiss hinterlässt keine Spur: Es gibt keine Fehlermeldung, keinen auffälligen Wert, keinen Test, der anschlägt. Die Auswahl sah vollständig aus, weil Vollständigkeit von innen nicht überprüfbar ist.

Deshalb genügt es nicht, sorgfältig zu sein — man braucht ein Verfahren mit einer **Vollständigkeitsgarantie**. Das ε-Constraint-Verfahren hat sie: Wenn kein zulässiger Plan mehr existiert, ist die Front nachweislich vollständig.

Was das Team hätte tun sollen: ε-Constraint statt Gewichtsraster. Es hätte die vollständige Front geliefert — mit zehn statt 500 Solverläufen.

Micro-Quiz

1. b) Geometrisch heißt „gewichtete Summe minimieren“: eine Gerade von links unten an die Punktwolke schieben. Sie berührt immer einen Eckpunkt der unteren konvexen Hülle; Punkte darüber werden nie zuerst getroffen. (a) klingt plausibel und ist falsch — ein feineres Raster ändert nichts, wie die Gegenprobe über die Hülle im Programm zeigt.

2. b) Ein Lauf für das Kostenminimum, danach ein Lauf je Frontpunkt, und ein letzter, der unzulässig ist und die Schleife beendet. Der Aufwand hängt also an der **Länge der Front**, nicht an der Feinheit eines Rasters — das ist der praktische Vorteil des Verfahrens.

3. b) Es gibt mehrere kostenminimale Pläne, und der Solver gibt normalerweise einen beliebigen davon zurück. Die lexikografische Formulierung sucht unter allen den saubersten — die CO□-Ersparnis kostet dann tatsächlich nichts. (c) ist ein guter Verdacht, trifft hier aber nicht zu: Auch bei exakt gleichem Budget bliebe der Effekt.

Selbsttest

1. Freie Antwort. Ein tragfähiges Beispiel enthält zwei Größen, die man nicht in dieselbe Einheit bringen kann, ohne eine Wertung vorzunehmen — etwa Personalkosten gegen Reaktionszeit oder Lagerbestand gegen Lieferfähigkeit.
2. Bei einem LP ist der zulässige Bereich konvex; die Pareto-Front liegt vollständig auf ihrer eigenen konvexen Hülle, und die gewichtete Summe erreicht jeden Punkt. Bei einem MILP zerfällt der Bereich in einzelne Punkte, zwischen denen Einbuchtungen entstehen — und die dort liegenden Lösungen sind unerreichbar.
3. Erstens ist w kein Regler zwischen 0 und 1, sondern ein **Wechselkurs** mit einer Einheit: Euro je Kilogramm. „0,5“ heißt also „ein Kilogramm CO□ ist mir 50 Cent wert“ — eine sehr konkrete und keineswegs neutrale Aussage. Zweitens gibt es keinen neutralen Wert, weil weite Gewichtsbereiche dieselbe Lösung liefern und es dazwischen springt.
4. Man optimiert nur das erste Ziel und schreibt dem zweiten eine Obergrenze vor. Diese Grenze setzt man zunächst auf den Wert, der beim reinen Kostenoptimum ohnehin anfällt, und drückt sie dann Schritt für Schritt weiter herunter. Sobald kein zulässiger Plan mehr existiert, hat man alle Kompromisse.
5. Den **Schattenpreis** — Aufpreis geteilt durch Ersparnis, hier in Euro je Kilogramm. Ohne ihn ist die Front eine Liste von Zahlenpaaren; mit ihm ist sie mit dem internen CO□-Preis vergleichbar und damit entscheidbar.

A.15 Lösungen zu Kapitel „Predict-then-Optimize“

15.1 — Andere Preise. Der Überhang kostet nur noch 1,50 €, die Fehlmenge weiterhin 6 €. Das kritische Verhältnis steigt von $6/9 = 0{,}667$ auf $6/7{,}5 = 0{,}80$. Die Bestellmenge verschiebt sich also **nach oben** — man bestellt jetzt das 80-%-Quantil statt des 66,7-%-Quantils.

Das ist die richtige Richtung und leicht zu merken: Je billiger der Überhang, desto großzügiger darf man ansetzen. Im Grenzfall kostenloser Entsorgung ($c_+ \to 0$) geht das kritische Verhältnis gegen 1 — man bestellt so viel, dass praktisch nie etwas fehlt.

15.2 — Wann ist der Mittelwert richtig? Nur wenn $c_- = c_+$ **und** die Nachfrageverteilung symmetrisch ist. Dann liegt das kritische Verhältnis bei 0,5, das 50-%-Quantil ist der Median, und bei Symmetrie fällt der Median mit dem Mittelwert zusammen.

Beide Bedingungen sind unrealistisch. Fehlmenge und Überhang kosten fast nie dasselbe — bei Frischware ist der Überhang teurer, bei Ersatzteilen die Fehlmenge um Größenordnungen. Und Nachfrageverteilungen sind meist rechtsschief. Der Mittelwert ist damit die Ausnahme, nicht die Regel — er wird nur benutzt, weil Regressionsmodelle ihn standardmäßig liefern.

15.3 — Die Kennzahl der Prognoseabteilung. Die Rangfolge bleibt: Die Punktprognose gewinnt auch beim MAPE, weil beide Maße den *Erwartungswert* belohnen — der MAPE etwas anders gewichtet, aber ebenfalls symmetrisch in dem Sinne, dass er nicht weiß, dass Unterschätzung teurer ist als Überschätzung.

Das ist kein Zufall, sondern der Kern des Kapitels: **Jedes rein statistische Fehlermaß ignoriert die Kostenasymmetrie.** Man kann das Problem nicht lösen, indem man das Fehlermaß wechselt; man muss

die Kosten selbst messen. (Die einzige Ausnahme ist der *Pinball Loss* — genau das Maß, das die Quantilregression minimiert, und das ist eben kein allgemeines Prognosemaß, sondern eines für ein bestimmtes Quantil.)

15.4 — Zwei Quantile. Das Band ist an **Aktionstagen** deutlich breiter — dort ist die Streuung im Modell fast viermal so groß. Genau das ist die Information, die eine Punktprognose plus fester Zuschlag wegwirft.

Ein praktischer Nebeneffekt: Ein solches Band ist die verständlichste Form, Unsicherheit an die Disposition zu berichten. „Zwischen 130 und 210” sagt einem Menschen mehr als „170 ± Sicherheitszuschlag”.

15.5 — Der Wert der Merkmale. Zu erwarten ist, dass die **Kosten stärker steigen als der MSE**. Der Grund: Das Merkmal „Aktion” trägt zwei verschiedene Informationen — es verschiebt den Erwartungswert (um 45 Stück) *und* die Streuung (von 8 auf 30). Der MSE bemerkt nur den ersten Teil; die Entscheidungskosten spüren beide.

Der Lehrsatz dahinter: **Der Wert eines Merkmals hängt davon ab, wofür man es benutzt.** Eine Merkmalsauswahl, die nach MSE-Beitrag sortiert, wirft möglicherweise genau die Merkmale weg, die für die Entscheidung am wichtigsten sind — nämlich die, die etwas über die Unsicherheit sagen.

Finde den Denkfehler — „Wir haben die Prognose um 18 % verbessert“

Der Sicherheitszuschlag stammt aus den Residuen des alten Modells.

Das ist die vollständige Antwort auf den zweiten Teil, und sie ist präziser als „bessere Prognose kann schlechter sein”. Der Ablauf im Einzelnen:

1. Das alte Modell hatte größere Residuen, also einen größeren Sicherheitszuschlag.
2. Der Zuschlag wurde nie neu berechnet — er ist „seit Jahren unverändert”.
3. Das neue Modell prognostiziert besser, also **näher am Erwartungswert**. Auf denselben Zuschlag addiert ergibt das eine niedrigere Bestellmenge als vorher.
4. Aber der richtige Zuschlag hängt an der Reststreuung des *aktuellen* Modells und am kritischen Verhältnis, nicht an der Gewohnheit.

Es ist also nicht die bessere Prognose, die schadet, sondern die **nicht mitgezogene zweite Stufe**. Die beiden Stufen wurden unabhängig gepflegt, obwohl sie voneinander abhängen.

Was das Team hätte messen müssen: die Entscheidungskosten. Sie hätten auf denselben Testdaten die Bestellmengen beider Modelle durchgerechnet und die Newsvendor-Kosten verglichen — eine Zahl in Euro, die die Disposition sofort verstanden hätte. Der Fehler wäre vor dem Produktivgang aufgefallen statt sechs Wochen danach.

Und die eigentliche Konsequenz: Der Zuschlag gehört gar nicht in die Disposition, sondern ins Modell. Wer das kritische Quantil direkt schätzt, kann diesen Fehler nicht machen — es gibt keinen zweiten, separat gepflegten Parameter mehr.

Micro-Quiz

1. b) Die Kostenasymmetrie verschiebt das Optimum vom Erwartungswert zum kritischen Quantil. (a) und (c) sind reale Probleme, aber nicht dieses: Selbst bei perfekt bekannter, normalverteilter Nachfrage wäre der Erwartungswert die falsche Bestellmenge.

2. b) Ein jährlich neu berechneter Zuschlag ist besser als ein veralteter, bleibt aber **für alle Tage gleich**. Die Unsicherheit hängt hier von den Merkmalen ab (Aktionstage), und das kann ein einzelner Wert nicht abbilden — im Kapitel gemessen: 3,5 gegen 12,3 Stück. (c) ist zu pauschal; Residuen sind ein brauchbarer Schätzer, nur eben ein globaler.

3. b) Ein einzelner Vergleich ohne Streuungsangabe ist keine belastbare Aussage. (a) verwechselt die Diagnose mit einem Rezept — 730 Testtage hat in der Praxis kaum jemand, und die Antwort ist Kreuzvalidierung statt einer Mindestzahl. (c) ist übertrieben: Der MSE ist ein brauchbares Maß für *Prognosegüte*, nur eben nicht für *Entscheidungsgüte*.

Selbsttest

1. Die erste Stufe wird auf ein statistisches Maß trainiert (MSE), die zweite erzeugt ökonomische Kosten mit asymmetrischer Struktur. An der Naht geht die Information über die **Unsicherheit** verloren: Weitergereicht wird eine einzelne Zahl, obwohl die Entscheidung die ganze Verteilung bräuchte.
2. Das **kritische Quantil** $c_-/(c_- + c_+)$ der Nachfrageverteilung. Der Wert kommt aus den Kosten der Entscheidung, nicht aus den Daten — er ist bekannt, bevor man die erste Zeile Code schreibt.
3. Weil die Unsicherheit **heteroskedastisch** ist: Sie hängt selbst von den Merkmalen ab. Ein einziger Zuschlag ist an ruhigen Tagen zu groß und an unruhigen zu klein.
4. „Wie ändern sich die **Entscheidungskosten**?“ — und, als Zusatz, „ist der Sicherheitszuschlag mitgezogen worden?“
5. So viele, dass der Unterschied zwischen den Modellen größer ist als die Schwankung des Maßes zwischen verschiedenen Zeitfenstern. Wie viele das sind, kann man nur messen — durch Kreuzvalidierung oder rollierende Auswertung.

A.16 Lösungen zu Kapitel „Die Strukturbrücke — dieselbe Mathematik, zwei Welten“

16.1 — Die dritte Ressource. „Liquidität” — wie schnell sich eine Position ohne Kursabschlag verkaufen lässt — entspricht in der Werkstatt der **Umrüstbarkeit** oder der **Vorlaufzeit**: Wie schnell lässt sich die Fertigung von diesem Produkt wieder wegdrehen, wenn der Auftrag storniert wird? In beiden Fällen ist es eine knappe Größe, die nichts mit dem Ertrag zu tun hat und trotzdem die Auswahl einschränkt.

Im Modell ist es schlicht eine weitere Zeile in `kapazitaeten` und ein weiterer Eintrag in jedem `verbrauch`. Genau das ist der Punkt des Kapitels: Eine neue Ressource kostet keinen Modellcode.

16.2 — Den Schattenpreis lesen. Bis zu **602,60 €** für 10 Einheiten (10 × 60,26 €) lohnt sich der Kauf — darüber nicht.

Die Einschränkung aus Abschnitt 5.9: Der Schattenpreis gilt nur **lokal**, solange sich die optimale Basis nicht ändert. Nach einer gewissen Erhöhung wird eine andere Nebenbedingung bindend (hier voraussicht-

lich das Kapital), und ab dort ist die zusätzliche Einheit weniger wert. Bei 10 Einheiten auf 90 ist das eine Erhöhung um 11 % — schon groß genug, dass man es nachrechnen sollte, statt zu extrapolieren.

Zweite Einschränkung: Bei **Entartung** ist der Schattenpreis nicht eindeutig. Verschiedene Solver können dann verschiedene, gleichermaßen gültige Werte liefern — auch das steht in Abschnitt 5.9.

16.3 — Die Brücke rückwärts. Shrinkage hilft, wenn man eine **Kovarianzmatrix** aus wenigen Beobachtungen schätzt. Bei Lieferzeiten tritt genau dasselbe Problem auf, sobald man die *Korrelationen zwischen* Lieferanten braucht — und die braucht man, sobald Lieferanten gemeinsame Ursachen haben: derselbe Hafen, dasselbe Vorprodukt, dieselbe Region.

Bei 40 Lieferanten hat die Kovarianzmatrix 820 zu schätzende Einträge. Wer dafür 36 Monatswerte hat, schätzt 820 Zahlen aus 36 Beobachtungen — dasselbe Missverhältnis wie bei Aktienrenditen, mit denselben Folgen (`Kovarianz_Falle.py`).

Benötigt würden: Lieferzeit-Zeitreihen je Lieferant über denselben Zeitraum, gleich getaktet. Genau daran scheitert es in der Praxis meist — die Daten liegen in Bestellvorgängen, nicht in einer Matrix.

16.4 — CVaR mit Ganzzahligkeit. Man ergänzt Binärvariablen y_j je Lieferant und die Kopplung $w_j \le y_j$ sowie $\sum_j y_j \le 3$. Da CVXPY gemischt-ganzzahlige Probleme unterstützt, genügt `cp.Variable(n, boolean=True)`.

Zu erwarten ist ein **höherer** CVaR: Der zulässige Bereich wird kleiner, und die Feinabstimmung über sechs Lieferanten fällt weg. Der Aufschlag ist der Preis der Vertragswirklichkeit — und genau die Zahl, die man dem Einkauf vorlegt, wenn dort jemand sagt, drei Lieferanten seien genug.

16.5 — Die eigene Brücke. Freie Antwort. Eine gute Bearbeitung nennt das Modell, die Entsprechung und **die Zeile der Grenztabelle, die im Weg steht** — meist ist es die dritte (Teilbarkeit) oder die erste (gemessen gegen geschätzt). Wenn keine im Weg steht, ist die Übertragung wahrscheinlich zu oberflächlich geprüft.

Finde den Denkfehler — „Das ist doch dasselbe Problem“

Fehler 1: Varianz ist hier das falsche Risikomaß — genau der Fall, für den es CVaR gibt.

Markowitz minimiert die **Varianz**, und die bestraft Abweichungen nach *beiden* Seiten gleich. Ein Lieferant, der manchmal zwei Tage zu früh liefert, erhöht die Varianz genauso wie einer, der zwei Tage zu spät liefert — obwohl das eine harmlos und das andere teuer ist.

Schwerer wiegt die Verteilungsform: Lieferzeiten haben einen **einseitig fetten Rand**. Im Normalfall schwankt es um wenige Tage, im seltenen Ausfall sind es zwei Wochen. Die Varianz mittelt diesen Rand weg; sie ist bei genau diesen Verteilungen am unzuverlässigsten. Das Kapitel führt die CVaR-Funktion nicht ohne Grund über *beide* Welten — die Übertragung wäre richtig gewesen, nur eben mit dem richtigen Risikomaß.

Fehler 2: Die Teilbarkeit — die dritte Zeile der Grenztabelle.

Markowitz liefert stetige Anteile: 7,3 % bei Lieferant A, 4,1 % bei B. Ein Lieferantenportfolio funktioniert so nicht. Es gibt Mindestabnahmemengen, Rahmenverträge, Qualifizierungskosten je Lieferant und eine praktische Obergrenze, wie viele Lieferanten der Einkauf betreuen kann. Aus dem QP wird ein **MIQP**

mit Kardinalitätsbedingung und Mindestmengen (Kapitel 6, Muster 5) — und dessen Lösung sieht anders aus als die gerundete stetige.

Was das Kapitel dazu sagt, und was nicht. Fehler 2 steht ausdrücklich in der Grenztabelle. Fehler 1 nicht — er ist ein Fehler *innerhalb* der Finanzwelt, den die Kollegin mit übernommen hat: Auch dort ist die Varianz für Verteilungen mit fetten Rändern das falsche Maß (Kapitel 20). Wer eine Methode überträgt, überträgt eben auch ihre Schwächen mit.

Micro-Quiz

1. b) Beide Probleme haben dieselbe Struktur: knappe Größen auf konkurrierende Verwendungen verteilen. (a) ist falsch — `Produktionsproblem` ist streng typisiert und validiert beim Einlesen; genau deshalb ist es aussagekräftig, dass die Depotdaten durchkommen. (c) ist frei erfunden.

2. b) Der Schattenpreis ist der Dualwert der Nebenbedingung: der zusätzliche Zielwert je zusätzlicher Einheit — lokal gültig. (a) verwechselt ihn mit der Auslastung, (c) mit den Kosten.

3. c) Die Teilbarkeit. In der Produktion sind Entscheidungen ganzzahlig (halbe Maschinen, halbe Schichten, halbe Lieferanten gibt es nicht), an den Finanzmärkten sind Anteile normal. Deshalb ist Teil II voller MILP und Teil IV fast frei davon — es liegt an der Welt, nicht an der Methode.

Selbsttest

1. Montagestunden ↔ Kapital; Plattenmaterial ↔ Risikobudget; Deckungsbeitrag je Stück ↔ erwarteter Ertrag je 1 000 €. (Weitere: Mindestlosgröße ↔ Mindestordergröße, Rüstkosten ↔ Ordergebühr, Sortimentsbreite ↔ Kardinalitätsgrenze.)

2. Ein LP sieht eine Matrix, einen Kapazitätsvektor und einen Zielvektor — Bedeutung kommt darin nicht vor. Das ist hier ein Vorteil, weil derselbe geprüfte, getestete Code beide Domänen bedient; man erbt die Verlässlichkeit mit.

3. „Stellen Sie sich vor, Ihre Engpassmaschine wäre nicht die Fräse, sondern eine Vorschrift: Sie dürfen nur eine bestimmte Menge Risiko in den Büchern haben. Der Schattenpreis sagt dann, was eine Lockerung dieser Vorschrift wert wäre — genau wie bei einer zusätzlichen Maschinenstunde.”

4. **Fette Ränder** (seltene, aber sehr große Abweichungen). Die Standardabweichung mittelt sie weg und bestraft außerdem Abweichungen nach oben genauso wie nach unten; CVaR sieht ausschließlich auf den schlechten Rand.

5. (a) Sind die Zahlen gemessen oder geschätzt — und wie groß ist der Schätzfehler im Verhältnis zu den Unterschieden? (b) Ist der datengenerierende Prozess stabil, oder reagiert er auf das Modell? (c) Sind die Entscheidungen teilbar oder ganzzahlig?

A.17 Lösungen zu Kapitel „Supply-Chain und Energieeinsatz unter Unsicherheit“

17.1 — Der Umschlagpunkt. Bei 60 MW statt 120 MW halbiert sich die Grenzkostendifferenz je Stunde (83 €/MWh × 60 MW = 4 980 €/h). Der Umschlagpunkt verdoppelt sich auf **7,7 Stunden**.

Das ist plausibel: Je kleiner die abzudeckende Leistung, desto länger dauert es, bis der Grenzkostenvorteil des großen Blocks seine Anfahrkosten einspielt. Ein Kernblock lohnt sich für eine kleine Restlast noch weniger als für eine große — was erklärt, warum Grundlastblöcke gerade dann unwirtschaftlich werden, wenn viel Wind einspeist und nur eine kleine Restlast bleibt.

17.2 — Die fehlende Nebenbedingung. Analog zum Anfahren braucht man eine Abfahr-Indikatorvariable $b_{k,t} \geq u_{k,t-1} - u_{k,t}$ und dann:

$$u_{k,\tau} \leq 1 - b_{k,t} \qquad \text{für } \tau = t, \ldots, t + M_k - 1$$

In Worten: Wer abfährt, bleibt M_k Stunden aus.

Der reale Sachverhalt: Ein abgeschalteter Dampfblock kühlt aus und darf aus werkstofftechnischen Gründen nicht sofort wieder hochgefahren werden — die Temperaturwechsel würden das Material schädigen. Bei Gasturbinen ist die Mindeststillstandszeit kurz oder null, bei Kernblöcken beträgt sie viele Stunden.

17.3 — Wie viele Szenarien? Zu erwarten ist, dass der Commitment-Plan ab einer gewissen Szenarienzahl stabil bleibt — oft schon bei 20 bis 40. Der Grund: Die erste Stufe ist binär und **grob**; sie kann nur ganze Blockstunden verschieben. Feine Unterschiede in der Szenarienmenge ändern daran nichts mehr.

Der Zusammenhang mit Abschnitt 15.6 ist der interessante Teil: Dort ging es um die Zahl der **Beobachtungen** bei einer Bewertung, hier um die Zahl der **Szenarien** in einem Modell. Beide Male lautet die Frage nicht „wie viele sind genug¿‘, sondern „ab wann ändert sich die Antwort nicht mehr?” — und beide Male beantwortet man sie, indem man es ausprobiert, statt eine Faustregel zu übernehmen.

17.4 — Der Wert eines Speichers. Der Speicher braucht Variablen für Ladung, Entladung und Füllstand je Szenario und Stunde, mit der Bilanz $F_t = F_{t-1} + \eta\, L_t - E_t/\eta$ und Grenzen für Leistung und Kapazität. Er gehört in die **zweite** Stufe: Wann geladen wird, darf sich am Tag selbst entscheiden.

Zu erwarten ist ein deutlicher Rückgang der erwarteten Kosten, und zwar aus zwei Quellen: Er verschiebt Energie aus billigen in teure Stunden *und* er ersetzt Vorhaltung — ein Speicher ist in Sekunden verfügbar, ein Kernblock in acht Stunden. Der zweite Effekt ist der größere und wird bei einer Betrachtung ohne Szenarien komplett übersehen.

Der Wert je MWh Kapazität ergibt sich als Kostenersparnis geteilt durch 400 MWh. Er ist mit den Investitionskosten vergleichbar — dieselbe Rechnung wie der Schattenpreis in Kapitel 5, nur über einen ganzen Tag.

17.5 — Wenn es größer wird. Mit 20 Blöcken und 168 Stunden hat die erste Stufe 3 360 Binärvariablen statt 120, und die Szenarienkopplung vervielfacht die kontinuierlichen Variablen. Zu erwarten ist, dass der MIP-Gap nach dem Zeitlimit nicht mehr auf null geht.

Zwei Wege aus Kapitel 9: **Erstens** — und in der Praxis meist ausreichend — den Gap akzeptieren; 1 bis 2 % sind bei Unit Commitment üblich und betriebswirtschaftlich belanglos. **Zweitens** LNS: Das Com-

mitment eines Tages herausbrechen und exakt neu optimieren, während der Rest der Woche festbleibt. Das ist genau die Struktur aus `Large_Neighborhood_Search.py` — ein zusammenhängendes Fenster zerstören und mit dem exakten Solver reparieren.

Was **nicht** funktioniert: eine reine Metaheuristik ohne Solver. Die Fahrweise ist bei gegebenem Commitment ein LP mit tausenden Variablen; die will man nicht heuristisch lösen.

Finde den Denkfehler — „Wir rechnen mit dem P50-Szenario“

Fehler 1: Der Median ist keine Absicherung, sondern eine Wette auf die Hälfte der Fälle.

„Die Hälfte fällt besser aus, die Hälfte schlechter, im Mittel gleicht sich das aus” — der letzte Halbsatz ist falsch. Er würde stimmen, wenn die Kosten **symmetrisch** um den Median lägen. Sie tun es nicht: Zu viel Erzeugung kostet ein paar tausend Euro Brennstoff, zu wenig kostet Lastabwurf zu 3 000 €/MWh. Das ist dieselbe Asymmetrie wie beim Newsvendor (Kapitel 15) — nur um Größenordnungen schärfer.

Im Programm gemessen: Der Erwartungswert-Plan (nahe am P50) führt in **28 von 40 Szenarien** zum Lastabwurf und kostet 149 % mehr. Nichts daran gleicht sich aus.

Fehler 2: Eine prozentuale Leistungsreserve löst das falsche Problem.

Sehen Sie sich an, was der zweistufige Plan tatsächlich verändert hat: Er hat **sechs zusätzliche Blockstunden** vorgehalten — er fährt einen Block früher an und lässt ihn länger laufen. Das ist eine Entscheidung über *Verfügbarkeit*, nicht über *Leistung*.

Eine Reserve von „5 % der Last” beschreibt dagegen eine Leistungsmenge. Sie hilft, wenn ein laufender Block etwas mehr liefern muss. Sie hilft **nicht**, wenn der benötigte Block gar nicht am Netz ist — und genau das ist der Fall, wenn der Wind ausbleibt und ein Kernblock mit acht Stunden Mindestlaufzeit um 18 Uhr nicht mehr herbeigerufen werden kann.

Die Reserve ist also nicht zu klein, sondern **von der falschen Art**. Was fehlt, sind nicht Megawatt, sondern *angefahrene* Megawatt.

Was das Team hätte tun sollen: Die Szenarien ins Modell nehmen, statt sie durch einen Repräsentanten plus Pauschalzuschlag zu ersetzen. Die Rechenzeit dafür liegt hier bei gut einer Sekunde.

Micro-Quiz

1. b) Anfahrkosten und Mindestlaufzeiten koppeln die Stunden. Die Merit-Order ist eine Sortierung und kennt keine Kopplung über die Zeit; sie ist für einen einzelnen Zeitpunkt richtig. (a) und (c) sind reale Themen, aber nicht der Grund für diese Aussage.

2. b) Die erste Stufe ist binär. Eine Kapazität lässt sich anteilig anpassen, ein Kraftwerk nicht zu 30 % anfahren — deshalb ist die Vorabentscheidung hier unwiderruflich grob. (a) ist eine Größenfrage, kein struktureller Unterschied; (c) trifft nur auf die dritte Variante zu.

3. b) Bei 3 000 €/MWh ist Vorhaltung schon im Erwartungswert billiger als Lastabwurf; die Schranke ist dann nicht bindend. Erst ein zu niedrig angesetzter Schaden macht Abwurf rechnerisch attraktiv, und dort greift sie. (a) und (c) unterstellen numerische Probleme, die es nicht gibt.

Selbsttest

1. Weil die Entscheidung einer Stunde die folgenden bindet: Anfahrkosten fallen einmal an, Mindestlaufzeiten erzwingen den Weiterbetrieb. Ein Regal füllt man unabhängig von der Reihenfolge, einen Kraftwerkspark nicht.
2. „Here and now” ist das **An/Aus je Block und Stunde** — binär, für alle Szenarien gleich, am Vorabend festgelegt. „Wait and see” ist die **Fahrweise**, also die Leistung jedes laufenden Blocks; sie darf je Szenario verschieden sein.
3. Weil die zu knappe Entscheidung **nicht revidierbar** ist. Ein Plan auf den Mittelwert hält in der Hälfte der Fälle zu wenig vor, und bei einer anpassbaren Größe wäre das halb so schlimm. Ein nicht angefahrener Block steht auch dann nicht zur Verfügung, wenn der Preis auf 3 000 € steigt.
4. Sie ändert etwas, wenn der Schaden im Zielfunktionsterm zu niedrig bewertet ist — dann nimmt der risikoneutrale Plan Schäden in Kauf, die man nicht hinnehmen will. Ist der Schaden realistisch bepreist, erzwingt schon die Erwartungswertminimierung die Absicherung, und die Schranke ist nicht bindend.
5. **46 € je vermiedener MWh.** Entstanden als Quotient aus dem Kostenaufschlag der Absicherung (844 € je Tag) und der dadurch vermiedenen Fehlmenge im CVaR der schlechtesten 10 % (18,4 MWh). Die Zahl ist mit dem volkswirtschaftlichen Schaden eines Ausfalls vergleichbar — und damit verhandelbar.

A.18 Lösungen zu Kapitel „Finanzdaten-Modellierung — Renditen, Kovarianz und Shrinkage“

18.1 — Renditeart. (a) diskret. (b) logarithmisch. (c) logarithmisch (bzw. diskret kumuliert über Produkt). (d) logarithmisch (statistische Eigenschaften).

18.2 — Mittelwertfalle. (a) Arithmetisches Mittel: $(60 - 40 + 60 - 40)/4 = +10\,\%$. (b) $10\,000 \cdot 1{,}6 \cdot 0{,}6 \cdot 1{,}6 \cdot 0{,}6 = 10\,000 \cdot 0{,}9216 = \mathbf{9216}$ € — ein **Verlust**. (c) Geometrisches Mittel: $0{,}9216^{1/4} - 1 = -2{,}0\,\%$ p. a. Der Unterschied zwischen $+10\,\%$ und $-2\,\%$ ist der Grund, warum Fondswerbung gern arithmetische Mittel zeigt.

18.3 — Konditionszahl. Erwartetes Muster: Bei $T = 40 < N = 30\ldots$ (hier $T > N$, aber knapp): sehr große Konditionszahl, kleinster Eigenwert nahe null. Bei $T = 100$: deutlich besser. Bei $T = 1000$: stabil. Faustregel $T \geq 5N$ bis $10N$.

18.4 — Spaltenreihenfolge. `raw["Close"].columns` ist alphabetisch; `raw["Close"][tickers]` stellt die eigene Reihenfolge her. Existiert ein Ticker nicht, wirft `[tickers]` einen `KeyError` — **das ist erwünscht**: lieber ein lauter Fehler als eine stille Verschiebung.

18.5 — Beide Ziele vergleichen. Erwartetes Ergebnis: Das Konstant-Korrelations-Ziel schneidet bei Aktien meist etwas besser ab, weil es die unterschiedlichen Einzelvolatilitäten erhält. Der Unterschied ist aber kleiner als der Unterschied zwischen „mit“ und „ohne“ Shrinkage — die Wahl des Ziels ist zweitrangig gegenüber der Entscheidung, überhaupt zu schrumpfen.

18.6 — Error-Maximizer messen. (a)/(b) Erwartetes Ergebnis: Die Gleichgewichtung schlägt die Stichproben-Optimierung bis etwa $T/N \approx 5$–10; Ledoit-Wolf schlägt sie schon früher. (c) **Praktische**

Folgerung: Bei knapper Datenlage ist 1/N ein ernstzunehmender Kandidat, und jede Optimierung muss sich daran messen lassen.

Finde den Denkfehler — Das risikofreie Portfolio

(a) **Der Rang.** Eine aus T Beobachtungen geschätzte Kovarianzmatrix hat höchstens den Rang $T-1$ — hier also **19** statt der nötigen 30. Geometrisch heißt das: Die 20 beobachteten Renditevektoren spannen nur einen 19-dimensionalen Unterraum des 30-dimensionalen Anlageraums auf. Es bleiben **11 Richtungen** übrig, über die die Daten schlicht **nichts** aussagen.

In genau diesen Richtungen misst die Matrix eine Varianz von exakt null. Nicht, weil dort kein Risiko wäre — sondern weil sie blind dafür ist. Der kleinste Eigenwert im Programm beträgt $-6{,}5 \cdot 10^{-20}$: numerisch null, mit einem Vorzeichen, das es gar nicht geben dürfte.

(b) **Warum der Optimierer das findet.** Weil er genau danach sucht. Die Aufgabe lautet „minimiere $\mathbf{w}^\top \mathbf{S}\mathbf{w}$“ — und es gibt Richtungen, in denen dieser Ausdruck **null** ist. Ein Optimierer, der Freiheit hat, wird sie unfehlbar ansteuern.

Das ist der Kern des **Error-Maximizer-Effekts**: Ein Optimierer sucht nicht nach der besten Anlage, sondern nach dem größten Fehler in Ihren Schätzungen. Je mehr Freiheit Sie ihm geben, desto gründlicher findet er ihn. Er tut damit genau das, worum Sie ihn gebeten haben — die Schwäche liegt in den Daten, nicht im Verfahren.

(c) **Der Hebel von 3,8.** Die Summe der Beträge aller Gewichte beträgt 3,8, obwohl sie sich zu 1 summieren. Das heißt: Es wird in erheblichem Umfang leerverkauft. Für je 100 € Kapital werden rund 240 € gekauft und 140 € leerverkauft.

Ein solches Portfolio ist nicht nur riskant, es ist auch praktisch kaum umsetzbar: Leerverkäufe kosten Leihgebühren, binden Sicherheiten und sind für viele Mandate schlicht untersagt. Ledoit-Wolf senkt den Hebel auf 1,8 — das Portfolio wird nebenbei **handelbarer**, nicht nur stabiler.

Praktischer Hinweis: Eine Nebenbedingung `w >= 0` (keine Leerverkäufe) wirkt schon für sich als kräftige Regularisierung. Sie ist oft die billigste verfügbare Absicherung gegen diesen Fehler.

(d) **Warum 1/N gewinnt — und was das allgemein bedeutet.** Die Gleichgewichtung schätzt **nichts**. Sie hat deshalb auch keinen Schätzfehler, den ein Optimierer ausnutzen könnte. Der rohe Optimierer dagegen bezahlt seine theoretische Überlegenheit mit einer Anfälligkeit für Rauschen, die bei $T < N$ jeden Vorteil auffrisst. Dieser Befund ist in der Literatur breit belegt (DeMiguel, Garlappi, Uppal 2009) und für Optimierungsfachleute unbequem.

Die Lehre reicht weit über die Finanzwelt hinaus:

> **Jedes Optimierungsergebnis ist höchstens so gut wie die Daten, auf denen es beruht — und anders als ein Mensch hinterfragt ein Optimierer diese Daten nie.**

Dasselbe Muster tritt überall auf, wo geschätzte Größen in ein Modell gehen: Bearbeitungs- zeiten aus zwanzig Stichproben, Ausfallraten aus drei Vorfällen, Nachfrageprognosen aus einer kurzen Historie. Drei Gegenmittel, in dieser Reihenfolge:

1. **Mehr Daten** — die einzige echte Lösung, sofern verfügbar.

2. **Schrumpfen** — die Schätzung zu einer stabilen, groben Struktur ziehen (Ledoit-Wolf, Regularisierung, Bayessche Priors).
3. **Die Freiheit begrenzen** — Nebenbedingungen wie „keine Leerverkäufe“, Ober- und Untergrenzen je Position, maximale Abweichung von einer Referenzlösung. Was der Optimierer nicht darf, kann er auch nicht falsch machen.

Und immer: **außerhalb des Schätzzeitraums prüfen.** Ein Modell, das nur im eigenen Datenfenster gut aussieht, sagt nichts aus. Genau diese Trennung führt Kapitel 21 als Walk-Forward-Verfahren aus.

Micro-Quiz

1 — (b) bei 75 %. $1{,}0 \cdot 0{,}5 \cdot 1{,}5 = 0{,}75$. Prozentuale Änderungen verknüpfen sich multiplikativ; ein Verlust wiegt schwerer als ein gleich großer Gewinn, weil er von einer größeren Basis abgeht und der Gewinn auf eine kleinere aufsetzt. (a) ist die verbreitetste Fehlvorstellung überhaupt im Umgang mit Renditen. (c) verwechselt die Rechnung mit einem zweiten Halbierungsschritt.

2 — (b). Es geht um die **Richtung der Summation**: Über die Zeit addieren sich logarithmische Renditen ($\ln(a \cdot b) = \ln a + \ln b$), über die Anlagen eines Portfolios die diskreten ($R_p = \sum_i w_i R_i$). Keine der beiden ist „genauer“ (a) — sie sind exakt ineinander umrechenbar. (c) ist frei erfunden; mit der Länge des Zeitraums hat die Wahl nichts zu tun.

3 — (b). 40 Beobachtungen für 50 Anlagen ergeben eine Matrix vom Rang höchstens 39 — singulär, mit mindestens 11 Richtungen scheinbar null Risikos. (a) verkennt, dass es nicht auf die absolute Zahl der Beobachtungen ankommt, sondern auf ihr **Verhältnis** zur Zahl der Anlagen; die Faustregel lautet $T \gtrsim 10 \cdot N$. (c) wäre eine Notlösung, die Information wegwirft — Shrinkage nutzt alle 50 Anlagen und ist praktisch immer die bessere Wahl.

Selbsttest

1. Log-Renditen sind Differenzen von Logarithmen — die addieren sich über die Zeit. Diskrete Renditen sind lineare Anteile am Kapital — die addieren sich über gewichtete Positionen.
2. Die Optimierung sucht die Richtungen kleinster geschätzter Varianz — genau jene, deren Eigenwerte am stärksten nach unten verzerrt sind. Sie optimiert dadurch in das Schätzrauschen hinein.
3. Der Rang von $\mathbf{X}^\top\mathbf{X}$ ist höchstens $T < N$ — die Matrix ist nicht invertierbar, das GMV-Problem hat unendlich viele Lösungen.
4. Stichprobenmatrix (unverzerrt, verrauscht) mit strukturiertem Ziel (verzerrt, stabil). δ wird analytisch so bestimmt, dass der erwartete quadratische Fehler minimal wird.
5. Skalierte Einheitsmatrix (sklearn) und Konstant-Korrelations-Ziel (LW 2003).

A.19 Lösungen zu Kapitel „Die moderne Portfoliotheorie nach Markowitz“

19.1 — Diversifikationseffekt.

(w_1, w_2)	μ_p	σ_p	Sharpe ($r_f = 2\,\%$)
(1,0; 0,0)	10,00 %	25,00 %	0,320
(0,5; 0,5)	8,00 %	12,74 %	0,471
(0,3; 0,7)	7,20 %	10,08 %	**0,516**
(0,0; 1,0)	6,00 %	12,00 %	0,333

Rechenweg für (0,5; 0,5): $\sigma_p^2 = 0{,}25 \cdot 0{,}0625 + 0{,}25 \cdot 0{,}0144 + 2 \cdot 0{,}25 \cdot (-0{,}2) \cdot 0{,}25 \cdot 0{,}12 = 0{,}015625 + 0{,}0036 - 0{,}003 = 0{,}016225$, also $\sigma_p = 12{,}74\,\%$.

Beste Sharpe Ratio unter den vier Kandidaten: (0,3; 0,7). Das exakte Optimum liegt bei $w_1 = 0{,}318$. **Bemerkenswert:** Die Mischung (0,3; 0,7) hat mit 10,08 % eine **geringere** Volatilität als *beide* Einzeltitel (25 % und 12 %) — genau der Diversifikationseffekt, den die negative Korrelation ermöglicht.

19.2 — Lambda deuten. Von $\lambda = 0$ (GMV, linker unterer Punkt der Kurve) wandert die Lösung entlang der Effizienzgrenze nach rechts oben, bis sie bei $\lambda \to \infty$ im Titel mit der höchsten Rendite endet (bzw. an der Positionsobergrenze).

19.3 — Korn-Transformation. $\mathrm{SR}(c\mathbf{w}) = \frac{c\mathbf{w}^\top \mu - r_f}{\sqrt{c^2 \mathbf{w}^\top \Sigma \mathbf{w}}}$ — bei einem Portfolio mit $\sum w_i = 1$ und Überrenditen geschrieben als $\mathbf{w}^\top(\mu - r_f \mathbf{1})$ kürzt sich c heraus. Mit der Bedingung $\sum w_i = 1$ ist die Skala jedoch **fixiert**, man kann also nicht frei skalieren. Die Transformation ersetzt diese Bedingung durch $\sum y_i = \kappa$ mit freiem κ und normiert stattdessen die Überrendite auf 1 — dadurch wird die Skala wieder frei und das Problem konvex.

19.4 — Restriktionen kosten. Erwartetes Muster: Sharpe Ratio sinkt monoton mit strengerer Grenze. Unlösbar wird es bei $w_{\max} < 1/n$ — dann kann die Summe der Gewichte 1 nicht mehr erreicht werden.

19.5 — Den Sektorfehler nachstellen. (a) AAPL, AMZN, CVX, GS statt AAPL, MSFT, NVDA, AMZN. (b) Die Gewichte unterscheiden sich deutlich, weil die eigentlich zu begrenzenden Tech-Titel frei laufen. (c) **Nein** — die Kennzahlen sehen völlig plausibel aus. Genau das macht den Fehler so gefährlich.

19.6 — Kardinalität. Erwartung: Die Sharpe Ratio sinkt leicht, die Rechenzeit steigt deutlich (MIQP statt QP). Bei $K \geq 5$ und $w_{\max} = 0{,}20$ ist die Restriktion praktisch nicht mehr bindend.

19.7 — Out-of-Sample. Typisches Ergebnis: Max-Sharpe schneidet in der zweiten Hälfte **schlechter** ab als in der ersten — es hat Schätzrauschen mitoptimiert. GMV ist stabiler (keine Renditeprognose nötig), 1/N oft überraschend gut. Mit Ledoit-Wolf verbessern sich GMV und Max-Sharpe spürbar. **Praxisfolgerung:** Renditeprognosen sind viel unzuverlässiger als Risikoschätzungen — Modelle, die ohne sie auskommen, sind robuster.

Finde den Denkfehler — Zwölf gleiche Anlagen, ein sehr ungleiches Portfolio

(a) **Woher die Spanne kommt.** Aus reinem Rauschen. Der Mittelwert von 250 Beobachtungen einer Größe mit Standardabweichung $\sigma = 1{,}2\,\%$ hat selbst noch den **Standardfehler**

$$\frac{\sigma}{\sqrt{T}} = \frac{0{,}012}{\sqrt{250}} = 0{,}00076 = 0{,}076\,\%$$

Das ist fast das **Doppelte** der wahren Rendite von 0,040 %. Bei zwölf unabhängigen Schätzungen liegen höchster und niedrigster Wert typischerweise rund drei Standardfehler auseinander — gemessen wurden 0,243 Prozentpunkte, also das 6,1-Fache des wahren Werts.

Die Anlagen sind identisch. Die *Schätzungen* sind es nicht — und der Optimierer sieht nur die Schätzungen.

(b) **Wie viele Beobachtungen nötig wären.** Damit der Standardfehler klein gegen die wahre Rendite wird, etwa $\sigma/\sqrt{T} \le \mu/2$:

$$T \ge \left(\frac{2\sigma}{\mu}\right)^2 = \left(\frac{2 \cdot 0{,}012}{0{,}0004}\right)^2 = 60^2 = 3600 \text{ Beobachtungen}$$

Das sind rund **14 Jahre Tagesdaten** — für eine einzige Anlage, unter der Annahme, dass sich ihre Eigenschaften in dieser Zeit nicht ändern. Genau daran scheitert die Renditeschätzung grundsätzlich, nicht nur bei diesem Beispiel.

(c) **Warum Renditen schlimmer sind als Kovarianzen.** Zwei Gründe:

1. **Statistisch.** Eine Varianz konvergiert deutlich schneller als ein Mittelwert. Grob: Der relative Fehler einer Varianzschätzung fällt mit $\sqrt{2/T}$, während der einer Renditeschätzung mit $\sigma/(\mu\sqrt{T})$ fällt — und σ/μ ist bei Aktien typischerweise 30 und größer. Bei denselben 250 Beobachtungen ist die Kovarianz brauchbar und der Mittelwert nicht.
2. **Strukturell.** Die Renditen stehen im **linearen** Teil der Zielfunktion. Eine kleine Änderung von μ_i verschiebt die Lösung sofort und ungedämpft; der quadratische Risikoterm wirkt dagegen ausgleichend. Deshalb reagieren die Gewichte auf Renditeschätzfehler viel heftiger als auf Kovarianzfehler.

Zusammen ergibt das das beobachtete Bild: 38 % Konzentration in einer Anlage, obwohl alle zwölf identisch sind.

(d) **Drei Gegenmittel.**

1. **Renditeschätzung ganz vermeiden.** Das **Minimum-Varianz-Portfolio** minimiert nur $\mathbf{w}^\top \Sigma \mathbf{w}$ und braucht überhaupt kein μ. Damit fällt die unzuverlässigste Eingangsgröße ersatzlos weg. Das ist das wirksamste Mittel, weil es das Problem nicht abmildert, sondern **beseitigt** — man kann eine Größe nicht falsch schätzen, die man nicht verwendet.
2. **Stark schrumpfen.** Wenn Renditen gebraucht werden, zieht man sie kräftig zum Gesamtmittel (James-Stein) oder zu einer Gleichgewichtsannahme (Black-Litterman) — deutlich stärker als bei Kovarianzen, aus den Gründen unter (c).

3. **Freiheit begrenzen.** Obergrenzen je Position (etwa 15 %), Sektorgrenzen, maximale Abweichung von einer Referenzgewichtung. Was der Optimierer nicht darf, kann er auch nicht auf Rauschen setzen — dasselbe Rezept wie in Abschnitt 18.8.

Und die Kontrollfrage für den Alltag: Rechnen Sie Ihr Modell mit den Daten des halben Zeitraums und dann mit denen der anderen Hälfte. Wenn die Gewichte dabei stark springen, optimieren Sie Rauschen — unabhängig davon, wie gut die Kennzahlen im Schätzzeitraum aussehen.

Micro-Quiz

1 — (b). Das renditestärkste Portfolio auf der Effizienzlinie hat notwendig das schlechteste Rendite-Risiko-Verhältnis (im Kapitelbeispiel 0,44 gegenüber 0,75 beim Optimum) und besteht vollständig aus der renditestärksten Einzelanlage — Diversifikation findet dort gar nicht mehr statt. (a) und (c) sind sachlich falsch: Der Punkt ist numerisch völlig unproblematisch und wird von CVXPY ohne Weiteres gefunden.

2 — (b) es braucht keine Renditeschätzung. Renditen sind die mit Abstand unzuverlässigste Eingangsgröße (Standardfehler größer als der geschätzte Wert selbst); wer sie nicht benötigt, umgeht das Problem vollständig. (a) trifft nicht zu — die Nebenbedingungen sind dieselben. (c) ist falsch: Das Minimum-Varianz-Portfolio erzielt *erwartungsgemäß* weniger Rendite; sein Vorteil liegt in der Verlässlichkeit, nicht in der Höhe.

3 — (b) Schätzfehler. Eine Konzentration von 44 % auf einen von zwölf Titeln ist das typische Bild eines Optimierers, der einem Rauschsignal folgt — im Kapitelbeispiel entsteht sie sogar dann, wenn alle Anlagen nachweislich identisch sind. (a) mag zutreffen, ist aber die unwahrscheinlichere Erklärung und muss belegt werden, nicht angenommen. (c) beschriebe ein anderes Fehlerbild: Eine falsch skalierte Kovarianzmatrix führt zu unplausiblen Risikowerten, nicht zu Konzentration.

Selbsttest

1. Weil sich Schwankungen teilweise ausgleichen (Korrelationsterm), während sich die Renditen linear mitteln.
2. Das GMV. Es benötigt nur Σ, keine Renditeprognose — und Renditeprognosen sind die unzuverlässigste Zutat.
3. Sie ist ein Quotient. Die Korn-Transformation normiert den Zähler auf 1, wodurch das Maximieren des Quotienten zum Minimieren des Nenners wird — ein QP.
4. **Alle mit κ mitskalieren:** aus $w_i \le c$ wird $y_i \le c\kappa$.
5. Weil sie keinerlei Schätzung benötigt und damit keinen Schätzfehler enthält — sie schlägt optimierte Portfolios out of sample überraschend häufig.

A.20 Lösungen zu Kapitel „Tail-Risiko, CVaR und Transaktionskosten“

20.1 — VaR und CVaR von Hand. Sortierte Verluste: $-2{,}0; -1{,}2; -0{,}8; -0{,}4; 0{,}3; 0{,}6; 1{,}1; 2{,}1; 5{,}4; 8{,}9$. Das 80 %-Quantil ist der 8. Wert: $\text{VaR}_{80\%} = 2{,}1$. $CVaR_{80%} = $ Mittel der schlechtesten 20 % $= (5{,}4 + 8{,}9)/2 = \mathbf{7{,}15}$.

20.2 — Subadditivität. Ein Risikomaß sollte Diversifikation nie bestrafen: Das Risiko eines zusammengelegten Portfolios darf nicht größer sein als die Summe der Einzelrisiken. Wird das verletzt, hätte eine Bank einen Anreiz, Portfolios künstlich aufzuspalten, um Eigenkapitalanforderungen zu senken — ökonomisch unsinnig.

20.3 — Rockafellar-Uryasev nachvollziehen. (a) Schlechtestes Drittel von $(1, 4, 9)$ ist $\{9\} \to \text{CVaR} = 9$. (b) Mit $\frac{1}{S(1-\alpha)} = \frac{1}{3 \cdot (1/3)} = 1$: $\gamma = 0$: $0 + (1 + 4 + 9) = 14$. $\gamma = 1$: $1 + (0 + 3 + 8) = 12$. $\gamma = 4$: $4 + (0 + 0 + 5) = \mathbf{9}$. $\gamma = 5$: $5 + 4 = 9$. $\gamma = 9$: $9 + 0 = 9$. (c) Minimum ab $\gamma = 4$ bei **9** — identisch mit (a) □. (Das Minimum wird auf einem ganzen Intervall angenommen, weil die Funktion stückweise linear ist — genau der Grund, warum der CVaR nicht *streng* konvex ist.)

20.4 — Einheiten prüfen. In einem Modell, das annualisierte Rendite gegen täglichen CVaR verrechnet, wirkt 1,5 effektiv als $1{,}5/252 \approx 0{,}006$ auf Tagesbasis — der Risikoterm ist also um Faktor 252 zu leicht gewichtet. Um dieselbe Wirkung wie `RISIKOAVERSION = 1.5` im konsistenten Tagesmodell zu erzielen, hätte man dort `lambda_risk = 1.5 * 252 = 378` setzen müssen.

20.5 — Risikoaversion kalibrieren. (a) Konkav, ähnlich der Markowitz-Frontier, aber im Rendite-CVaR-Raum. (b) Bei Normalverteilung entspricht CVaR-Optimierung ungefähr der Varianz-Optimierung; die Ergebnisse divergieren umso stärker, je schiefer die Verteilung ist. (c) Empfehlung ohne Fachjargon: „Bei dieser Einstellung liegt der durchschnittliche Verlust an den schlechtesten fünf Prozent der Tage bei X Prozent — bei einer erwarteten Rendite von Y Prozent.“ Das ist entscheidbar; „$\lambda = 4$“ ist es nicht.

20.6 — Grenze statt Strafe. (a) Mit engerer Grenze sinkt die erreichbare Rendite; bei $\tau = 0$ bleibt das Altportfolio. (b) **Vorteil der Grenze:** garantierte Obergrenze für die Handelsaktivität — wichtig, wenn Liquidität oder Compliance eine harte Schranke verlangen. **Nachteil:** Sie kann das Modell unlösbar machen und ignoriert, dass ein sehr lohnender Trade den Aufwand wert wäre. (c) Strafe bei ökonomischer Abwägung, Grenze bei regulatorischen oder Liquiditätsvorgaben.

Finde den Denkfehler — Das Risikobudget, das durch Diversifikation stieg

(a) **Die entscheidende Wahrscheinlichkeit.** Bei zwei unabhängigen Anleihen mit je 4 % Ausfallwahrscheinlichkeit fällt **mindestens eine** aus mit

$$1 - (1 - 0{,}04)^2 = 1 - 0{,}9216 = 0{,}0784 = \mathbf{7{,}84}\,\%$$

Das ist **mehr als 5 %** — und damit rutscht der Ausfall in genau den Bereich, den der VaR(95) betrachtet. Einzeln lag er mit 4 % darunter und blieb unsichtbar.

Der Sprung von −4 € auf +98 € ist also kein Rechenfehler, sondern die korrekte Antwort auf eine schlecht gestellte Frage.

(b) **Warum eine ausfallgefährdete Anleihe „Gewinn“ meldet.** Der VaR(95) ist das 95-%- Quantil der Verlustverteilung: der Verlust, der an höchstens 5 % der Fälle überschritten wird. Bei einer

einzelnen Anleihe passiert in 96 % der Fälle nichts (Kupon +2 €, also Verlust −2 €). Das 95-%-Quantil liegt damit **mitten im guten Bereich** — bei −2 €.

Der Totalverlust findet in den restlichen 4 % statt, also **jenseits** der Schwelle. Der VaR schaut dort nicht hin. Er ist nicht falsch berechnet; er beantwortet schlicht eine andere Frage als die, die das Risikomanagement stellt.

(c) **Verletzt wird die Subadditivität** — die Forderung

$$\rho(A + B) \leq \rho(A) + \rho(B)$$

Auf Deutsch: Ein Portfolio darf nie riskanter sein als seine Teile zusammen; Diversifikation darf nicht schaden. Ein Maß, das das erfüllt (zusammen mit drei weiteren Eigenschaften), heißt **kohärent**.

Was das praktisch bedeutet, ist gravierender, als es klingt. Eine Organisation tut mit Kennzahlen immer dasselbe: Sie verteilt sie auf Einheiten und zählt sie wieder zusammen. Genau das ist beim VaR unzulässig:

- **Budgets lassen sich nicht aufteilen.** Zwei Abteilungen, die je ihr VaR-Limit einhalten, können zusammen weit darüber liegen.
- **Es entstehen Fehlanreize.** Eine Position, deren Verlustwahrscheinlichkeit knapp unter dem VaR-Niveau liegt, erscheint im Bericht als risikolos — je katastrophaler und seltener, desto unsichtbarer. Wer nach VaR gesteuert wird, hat einen direkten Anreiz, genau solche Positionen aufzubauen.
- **Diversifikation wird bestraft statt belohnt** — das Gegenteil dessen, wozu Risikosteuerung da ist.

(d) **Warum der CVaR das nicht kann.** Der CVaR mittelt über den **gesamten** Schwanz, statt an seiner Grenze stehenzubleiben. Damit sieht er den Ausfall in jedem der drei Fälle, und das Ergebnis verhält sich wie erwartet: 79,59 € und 79,66 € einzeln, 101,33 € zusammen statt 159,24 € — die Diversifikation **senkt** das Risiko um rund ein Drittel.

Formal folgt die Subadditivität daraus, dass der CVaR sich als **Maximum über Erwartungswerte** schreiben lässt (Darstellungssatz für kohärente Risikomaße), und Maxima von Erwartungswerten sind stets subadditiv. Anschaulicher: Ein Mittelwert über eine Menge verhält sich gutartig, wenn man Mengen zusammenlegt; ein Quantil nicht — es kann springen, sobald sich die Reihenfolge der Szenarien ändert.

Die praktische Konsequenz ist die Umstellung der Bankenaufsicht mit Basel III vom VaR auf den *Expected Shortfall* — dieselbe Größe, die hier CVaR heißt. Und für Ihre eigenen Modelle, auch außerhalb der Finanzwelt: Wo immer Sie eine Risikokennzahl über Einheiten aggregieren wollen, prüfen Sie zuerst, ob das Maß das überhaupt zulässt.

Micro-Quiz

1 — (b) nichts. Der VaR ist ein Quantil: Er markiert die Schwelle und sagt nichts über den Bereich dahinter. Genau das zeigt der Schnellstart dieses Kapitels — zwei Anlagen mit identischem VaR von 3,00 %, aber CVaR 3,00 % gegen 9,17 %. (a) verwechselt die Schwelle mit dem, was hinter ihr liegt. (c) ist ebenfalls falsch: Anlage B hat zwar die höhere Schwankung, aber die Schwankung allein sagt nichts über die Form des Schwanzes — genau deshalb reicht auch die Varianz als Risikomaß nicht aus.

2 — (b). Rockafellar und Uryasev zeigen, dass sich der CVaR als Minimum über eine Hilfsvariable γ schreiben lässt; mit Schlupfvariablen für die Terme $\max(\cdot, 0)$ wird daraus ein gewöhnliches LP. Die VaR-Minimierung ist dagegen nicht-konvex (das Quantil springt) und hat viele lokale Optima — dasselbe Problem wie in Abschnitt 11.6. (a) ist zwar zutreffend, aber nicht der entscheidende Grund; (c) ist eine wahre Aussage ohne Bezug zur Optimierbarkeit.

3 — (b) CVaR wegen der Subadditivität. Nur ein subadditives Maß erlaubt es, Kennzahlen über Einheiten zusammenzufassen, ohne dass Diversifikation bestraft wird. (a) verkennt, dass gerade die Verbreitung des VaR das Problem ist — er wird laufend addiert, obwohl er es nicht darf. (c) ist falsch, und der Denkfehler dieses Kapitels ist das Gegenbeispiel: Die beiden Anleihen dort sind **gerade unabhängig**, und genau deshalb tritt die Verletzung auf.

Selbsttest

1. Die **Höhe** der Verluste jenseits der Schwelle — der VaR kennt nur die Schwelle selbst.
2. Er ist in der Szenario-Darstellung **stückweise linear**; das Minimum kann auf einem ganzen Intervall angenommen werden.
3. Sie ist eine freie Hilfsvariable, die im Optimum automatisch den VaR annimmt — man muss ihn also nicht vorher kennen.
4. Weil die L_1-Norm dünn besetzte Lösungen begünstigt (im Gegensatz zur L_2-Norm, die viele kleine Änderungen bevorzugt).
5. Weil der Gewichtungsparameter dann eine andere Bedeutung hat, als er zu haben scheint — das Modell ist nicht mehr kalibrierbar.

A.21 Lösungen zu Kapitel „Die vollständige quantitative Handelsmaschine“

21.1 — Lookahead erkennen. (a) sauber (`:heute`). (b) **Lookahead** — gesamter Zeitraum. (c) **Lookahead** — Volatilität über den ganzen Zeitraum enthält Zukunft. (d) sauber — Gewichte aus Daten bis heute, angewendet auf heute (idealerweise auf morgen). (e) **Survivorship-Bias** — das Universum wird danach gefiltert, wer über den gesamten Zeitraum Daten hat.

21.2 — Kennzahlen deuten. A: Sharpe $= (12 - 2)/22 = 0{,}45$, Calmar $= 12/35 = 0{,}34$. B: Sharpe $= (8 - 2)/9 = 0{,}67$, Calmar $= 8/12 = 0{,}67$. **Empfehlung: B** — für einen Pensionsfonds ist der Drawdown entscheidend, weil laufende Auszahlungen in einer Verlustphase Substanz vernichten. B ist in beiden risikoadjustierten Maßen besser.

21.3 — Rebalancing-Kalender. Etwa 25–30 % der Termine fallen aus. Die Kennzahlen ändern sich messbar — in welche Richtung, ist zufällig. Genau das ist der Punkt: Der Fehler verzerrt, ohne aufzufallen.

21.4 — Rebalancing-Frequenz. Typisches Muster: Turnover und Kosten steigen etwa linear mit der Frequenz, der Bruttoertrag verbessert sich nur unterproportional. Bei 0,15 % Gebühren ist monatlich meist vertretbar, bei 0,5 % eher quartalsweise. Ein **Toleranzband** (nur handeln bei Abweichung > x %) schlägt fast immer die feste Frequenz.

21.5 — Krisenverhalten. Erwartung: Die Überrendite ist selten stabil; oft stammt sie aus wenigen Perioden. Folgerung: Eine gute Gesamtkennzahl kann von einer einzigen glücklichen Phase getragen sein — **immer** nach Teilzeiträumen aufschlüsseln.

21.6 — Data Snooping messen. Erwartetes Ergebnis: Die beste In-Sample-Sharpe-Ratio liegt deutlich über dem Mittelwert aller Varianten; auf dem zurückgehaltenen Zeitraum fällt sie Richtung Mittelwert zurück. Die **Differenz zwischen (a) und (c) ist der Selektionseffekt** — genau das, was die Deflated Sharpe Ratio korrigieren soll.

Finde den Denkfehler — Die Strategie mit dem hochsignifikanten Ergebnis

(a) **Warum ein korrekter Backtest wertlos sein kann.** Der Backtest misst genau das, was er messen soll: die Wertentwicklung *dieser einen* Strategie. Die Frage, die beantwortet werden soll, lautet aber anders — nämlich: *„Ist dieses Ergebnis besser, als es Zufall erklären kann?“*

Und für diese Frage ist entscheidend, dass die Strategie als **Beste aus hundert** ausgewählt wurde. Ein p-Wert von 0,0094 bedeutet: „Wenn diese Strategie keinen Vorteil hätte, sähe sie in 0,94 % der Fälle so gut aus.“ Bei hundert Versuchen erwartet man aber rund fünf Ergebnisse unter dem 5-%-Niveau — allein durch Zufall. Genau das zeigt die Tabelle: Auf reinem Rauschen liefert die Beste von hundert im Mittel Sharpe 1,13 und p = 0,0094. **Dieselben Zahlen.**

Der Fehler steckt nicht im Backtest, sondern in der **Auswahl**. Deshalb ist er auch durch noch so sorgfältiges Programmieren nicht zu verhindern.

(b) **Warum die konstante Spalte der Kern ist.** Ein Test zum 5-%-Niveau irrt in 5 % der Fälle — das ist seine Definition, nicht sein Fehler. Der Anteil bleibt deshalb bei jedem Stichprobenumfang gleich.

Was sich ändert, ist die **absolute Zahl**:

getestet	falsche Treffer (erwartet)
1	0,05
10	0,5
100	**5**
1000	**50**

Und nun kommt der entscheidende Schritt: Berichtet wird immer nur **die beste** Strategie. Von den fünf zufälligen Treffern bei hundert Versuchen sieht der Vorgesetzte genau einen — den erfolgreichsten. Die 99 verworfenen Varianten tauchen in keiner Präsentation auf. Der Auswahlprozess ist unsichtbar, das Ergebnis sichtbar.

(c) **Die fehlende Zahl ist die Anzahl der Versuche.** Sie steht in keinem Backtest, in keiner Kennzahl und in keinem Programm — sie existiert nur im Kopf der Person, die die Arbeit gemacht hat. Ohne sie ist weder der p-Wert noch die Sharpe-Kennzahl interpretierbar.

Deshalb: **Führen Sie ein Protokoll.** Jede probierte Variante, auch die schnell verworfenen, auch die „nur mal geschaut“. Diese Liste ist kein bürokratischer Ballast, sondern die Voraussetzung dafür, dass das Endergebnis überhaupt eine Aussage hat.

(d) **Drei Gegenmittel.**

1. **Zählen.** Siehe (c). Das kostet nichts und ist die Grundlage für alles Weitere.
2. **Korrigieren.** Die **Bonferroni-Korrektur** teilt die Signifikanzschwelle durch die Zahl der Versuche: Bei 100 Varianten muss $p < 0{,}0005$ gelten statt $p < 0{,}05$. Der Wert 0,0094 aus dem Beispiel besteht diese Hürde deutlich **nicht**. Für Handelsstrategien gibt es verfeinerte Verfahren (*Deflated Sharpe Ratio* nach Bailey und López de Prado), die dasselbe Prinzip verfolgen und dabei die Korrelation zwischen den Varianten berücksichtigen.
3. **Zurückhalten.** Legen Sie einen Zeitraum beiseite, **bevor** Sie anfangen, und rühren Sie ihn nicht an.

Warum das dritte Mittel nur einmal wirkt: Der zurückgehaltene Zeitraum ist genau so lange ein unabhängiger Test, wie er keinerlei Einfluss auf Ihre Entscheidungen hatte. In dem Moment, in dem Sie das Ergebnis sehen und daraufhin *irgendetwas* ändern — einen Parameter, ein Fenster, auch nur die Auswahl unter mehreren fertigen Kandidaten —, ist er Teil der Suche geworden. Ab dann misst er nicht mehr die Strategie, sondern wieder Ihre Anpassungsfähigkeit.

Ein zweiter Blick auf denselben Testzeitraum ist deshalb kein „nochmal prüfen“, sondern der 101. Versuch. Wer ihn braucht, braucht neue Daten — oder muss warten, bis die Zukunft welche liefert. Das ist unbequem und der Grund, warum diese Regel so oft gebrochen wird.

Micro-Quiz

1 — (b) nichts. Beide Kennzahlen sind Mittelwerte über die Zeit und blind für die Reihenfolge. Der Schnellstart dieses Kapitels zeigt es an **denselben** Tagesrenditen, nur anders angeordnet: identische Rendite (8,88 %), identische Sharpe (0,61), Rückschlag 19 % gegen 98,7 %. (a) ist die verbreitete Fehlannahme, die dazu führt, dass Rückschläge gar nicht berichtet werden. (c) ist falsch — die Schwankung ist in beiden Reihen exakt gleich, sie enthalten ja dieselben Zahlen.

2 — (b) unauffällig. Bei 50 Versuchen erwartet man rein zufällig $50 \cdot 0{,}05 = 2{,}5$ Ergebnisse unter dem 5-%-Niveau; ein p-Wert von 0,01 ist in dieser Menge nichts Besonderes. Die Bonferroni-Schwelle liegt bei $0{,}05/50 = 0{,}001$ — davon ist 0,01 um den Faktor zehn entfernt. (a) interpretiert den p-Wert so, als wäre es der einzige Test gewesen — genau der Denkfehler dieses Kapitels. (c) ist zu pauschal: Der p-Wert ist brauchbar, sofern man die Zahl der Versuche berücksichtigt.

3 — (b). Unabhängigkeit ist keine Eigenschaft der Daten, sondern des **Verfahrens**: Sobald das Ergebnis eine Entscheidung beeinflusst hat, ist der Zeitraum Teil der Suche. (a) trifft die Sache nicht — auch frische Daten wären nach der ersten Verwendung verbraucht. (c) ist frei erfunden; an der Berechnung ändert sich bei Wiederholung nichts.

Selbsttest

1. Zum Zeitpunkt t darf nur Information verwendet werden, die zu t vorlag.
2. Weil sie oft auf Wochenenden oder Feiertage fallen und dann nicht im Handelstage-Index enthalten sind — das Rebalancing entfällt still.
3. Man verändert künstlich die Daten **nach** dem Stichtag und prüft, ob sich das Signal **davor** ändert. Wirksam, weil er die Fehlerklasse mechanisch abdeckt statt auf Aufmerksamkeit zu setzen.
4. Weil sich die Gewichte zwischen Rebalancings durch Kursbewegungen von selbst verändern. Ohne Drift unterstellt man implizit tägliches kostenloses Rebalancing.
5. Weil er im Arbeitsprozess entsteht und im Code unsichtbar ist — man sieht dem Programm nicht an, wie viele Varianten verworfen wurden.

A.22 Lösungen zu Kapitel „Praxisfallen und der Weg zum produktiven Einsatz“

22.1 — Hart oder weich, revisited. (a) hart: gesetzliche Ruhezeit, Qualifikationspflicht. (b) weich: individuelle Wunschtage, Vermeidung von Freistunden. (c) diskutabel: Höchstzahl Vertretungsstunden pro Tag (tariflich vs. Notfall), gleichmäßige Wochenendverteilung.

22.2 — Zeitlimit wählen. (a) 1–3 s, Gap 5–10 % (Echtzeit schlägt Optimalität). (b) 1–5 min, Gap 1–2 %. (c) Stunden, Gap ~0 % (einmalige, folgenreiche Entscheidung). (d) 30–60 s, Gap 1 % — die Datenunsicherheit ist ohnehin größer.

22.3 — Relaxation einbauen. Muster wie in `Infeasibility_Diagnose.py`: Schlupfvariable je Mindestbesetzung, Strafe deutlich über allen weichen Zielen, aber endlich. Wichtig: **gestaffelte** Strafen, damit der Solver die *am wenigsten schmerzhafte* Verletzung wählt.

22.4 — Erklärbarkeit. Report-Struktur: (1) Zielwert und Aufschlüsselung nach Bestandteilen; (2) je Entscheidung die bindenden Bedingungen; (3) Schattenpreise der knappsten Ressourcen mit Handlungsempfehlung.

22.5 — Den Bericht in die Irre führen. (a) Der Plan kippt: *Rahmen* fällt von 47,5 auf **40** — das Minimum aus dem Liefervertrag —, *Deckel* steigt von 97,5 auf **120**, die Marktgrenze. Der Deckungsbeitrag springt von 3 930 € auf **5 190 €**. Mit ihm wechselt die Engpassstruktur: „Kapazität Lackieren” bindet **nicht mehr**, dafür bindet jetzt „Liefervertrag Rahmen” — und zwar als *Kosten* von 25 € je Stück, weil der Vertrag zur Herstellung eines inzwischen unattraktiven Produkts zwingt. Alle Schattenpreise steigen deutlich (Träger von 18 € auf 58 €). Der entscheidende Punkt der Aufgabe: Der Satz *„Teuerste Bindung ist Liefervertrag Träger”* steht **wörtlich unverändert** im Bericht — obwohl die Zahl dahinter sich mehr als verdreifacht hat und ein anderer Engpass die Fabrik begrenzt. Ein Bericht, der sich nicht ändert, ist kein Beweis dafür, dass sich nichts geändert hat. (b) Etwa: *„Unter der Annahme eines Deckungsbeitrags von 9 € je Deckel ist ‚Liefervertrag Träger’ die teuerste Bindung (18 € je Stück).”* Die Aussage bekommt damit ihre Voraussetzung mit — und wird angreifbar, was sie sein soll. (c) Mindestens eines von beidem: den **Gültigkeitsbereich** jedes Schattenpreises (bis wohin gilt er?), oder das Ergebnis einer **Sensitivitätsrechnung** über die unsichersten Eingangsgrößen — etwa „bei ±20 % Deckungsbeitrag Deckel bleibt die Rangfolge/kippt sie”. Das Programm rechnet den Gültigkeitsbereich für die auffälligste Stellschraube bereits aus; ihn für alle auszuweisen ist eine kleine Erweiterung.

22.6 — Gap gegen Laufzeit. Erwartetes Muster: Von 10 % auf 2 % kostet wenig Zeit; von 1 % auf 0 % kann die Laufzeit um Größenordnungen steigen, ohne dass sich der Zielwert nennenswert verbessert. Empfehlung: Gap dort ansetzen, wo die Kurve knickt — typischerweise 1–2 %.

22.7 — Post-Mortem. Bewertungskriterien: Wird zwischen **Symptom**, **Ursache** und **fehlender Prüfung** unterschieden? Ist die abgeleitete Regel allgemein genug, um beim nächsten Projekt zu helfen (z. B. „Nach jeder Vorzeichenumkehr eine numerische Gegenprobe“), aber konkret genug, um überprüfbar zu sein?

Finde den Denkfehler — Das Modell, das seit einem Jahr nicht mehr optimiert

(a) **Warum die konstante Laufzeit das Symptom ist.** Ein Solver, der fertig wird, braucht so lange, wie das Problem eben dauert — und diese Zeit wächst mit den Daten. Eine Laufzeit, die über Jahre exakt bei 3,00 Sekunden liegt, kann deshalb nur eines bedeuten: Der Job wird nicht fertig, sondern **abgeschnitten**.

Genau darin liegt die Tücke. Die übliche Betriebsüberwachung achtet auf zwei Dinge: Abstürze und Laufzeitspitzen. Hier gibt es weder das eine noch das andere — im Gegenteil, das Zeitlimit sorgt für eine bemerkenswert gleichmäßige Laufzeit. Das System sieht *besser* aus als eines, das ehrlich langsamer wird.

Es ist der umgekehrte Fall zum Alarm, den man erwartet: **Ein Optimierungsjob mit Zeitlimit wird bei wachsenden Daten nicht langsamer, sondern schlechter.** Und Qualität steht in keinem Betriebsprotokoll.

(b) **Die drei fehlenden Größen:**

Größe	Was sie verrät
Status	„Optimal“ oder „Time limit reached“ — die direkteste Auskunft überhaupt
Gap	Wie weit die gelieferte Lösung höchstens vom Bestmöglichen entfernt ist
Zeitausschöpfung	Laufzeit im Verhältnis zum Limit

Nur die Kosten zu protokollieren, ist das Äquivalent dazu, bei einem Messgerät den Anzeigewert abzulesen und die Fehlermeldung daneben zu ignorieren.

(c) **Am frühesten warnt die Zeitausschöpfung.** Sie ist eine **stetige** Größe und beginnt zu steigen, lange bevor das Limit erreicht wird — im Beispiel von 0,57 s über 1,50 s bis zum Anschlag. Status und Gap springen dagegen erst, wenn es bereits passiert ist: Bis 2024-Q3 lauten sie „Optimal“ und 0,00 %, ab 2025-Q1 plötzlich „Time limit“ und 8,4 %.

Eine Warnung bei 50 % Ausschöpfung hätte hier rund ein halbes Jahr Vorlauf gegeben — genug, um in Ruhe zu reagieren: Zeitlimit anheben, Modell straffen, oder einen bewussten Zielgap festlegen (etwa „1 % genügt uns“), statt eine unbekannte Verschlechterung hinzunehmen.

> **Faustregel:** Warnen Sie bei 50 % Zeitausschöpfung, alarmieren Sie bei 80 %. Und protokollieren Sie den Gap immer — auch dann, wenn er null ist. Eine Reihe von Nullen ist die beste Nachricht, die ein Betriebsprotokoll enthalten kann.

(d) **Warum sich die Mehrkosten nicht beziffern lassen.** Der Gap von 17,2 % ist eine **Obergrenze**, keine Messung: Er besagt, dass die gefundene Lösung höchstens 17,2 % über dem theoretisch Bestmöglichen liegt. Ob sie tatsächlich 17 % oder nur 3 % darüber liegt, weiß niemand — dazu müsste man das Optimum kennen, und genau das hat der Job ja nicht berechnet.

Nachträglich lässt sich das nur teilweise klären: Man kann die alten Instanzen erneut lösen, diesmal ohne Zeitlimit, und die Differenz ausrechnen. Was sich **nicht** rekonstruieren lässt, sind die Entscheidungen, die auf den schlechteren Plänen beruhten — welches Lager wurde nicht geschlossen, welche Tour wurde ein Jahr lang unnötig gefahren. Diese Kosten sind angefallen und nicht mehr zuzuordnen.

Das ist das eigentliche Argument für die Überwachung: Nicht, dass ein Gap von 17 % schlimm wäre — er kann völlig hinnehmbar sein —, sondern dass man ihn **kennen** muss, um darüber zu entscheiden. Ein bewusst akzeptierter Gap ist eine Managemententscheidung; ein unbemerkter ist ein Betriebsrisiko.

Micro-Quiz

1 — (b) das Zeitlimit greift. Eine Laufzeit, die exakt dem Limit entspricht und über Monate konstant bleibt, ist kein Stabilitätsbeleg, sondern zeigt, dass der Solver jedes Mal abgebrochen wird. (a) ist die Fehldeutung, um die es im Denkfehler geht. (c) läge nahe, wenn die Laufzeit *schwankte* oder das Limit überschritte — hier ist die Ursache aber im Modell und in der Datenmenge zu suchen, nicht in der Hardware.

2 — (b) aus dem ausgegebenen Tourenplan. Eine Prüfung muss von der **Lösung** ausgehen und die Anforderungen unabhängig nachrechnen. (a) fragt die Ladungsdimension — also ausgerechnet den Baustein, dessen Fehlen der Fehler war (siehe Abschnitt 8.7); existiert sie nicht, stürzt die Prüfung ab, existiert sie, kann sie per Konstruktion nie verletzt sein. (c) prüft gar nichts: Ob eine Kapazität in der Zielfunktion bepreist ist, sagt nichts darüber, ob sie eingehalten wurde.

3 — (c) hierarchisch lockern. `INFEASIBLE` sagt nur, *dass* ein Widerspruch existiert, nicht *welche* Regeln ihn erzeugen. Ersetzt man harte Verbote durch sehr teure Strafkosten, liefert der Solver wieder eine Lösung — und die verletzten Regeln stehen benannt in der Kostenzerlegung. Genau das tut `Infeasibility_Diagnose.py`. (a) hilft nicht: `INFEASIBLE` ist ein Beweis, kein Abbruch. (b) verschiebt das Problem und verliert die Gelegenheit, die Ursache zu finden, solange sie noch frisch ist.

Selbsttest

1. Durch hierarchische Relaxation: Schlupfvariablen mit hohen, gestaffelten Strafkosten für die verletzbaren Bedingungen.
2. Warum diese Lösung? Warum nicht die Alternative? Was würde sie verbessern?
3. Weil die Datenunsicherheit (oft ±10 %) die verbleibende Optimalitätslücke bei Weitem übersteigt.
4. Jeder Lauf arbeitet auf einem unveränderlichen, mit ID versehenen Datenstand — nur so sind Ergebnisse reproduzierbar und belegbar.
5. Die **Einführung**: mangelnde Akzeptanz, weil Ergebnisse nicht nachvollziehbar sind.
6. Sie berührt den Plan, ohne ihn zu verteuern — die Lösung liegt genau auf ihrer Grenze, wäre aber auch ohne sie dieselbe (Entartung). Für eine Nachverhandlung folgt: **nichts tun.** Der Aufwand brächte 0 €. „Bindend” ist ein geometrischer, „teuer” ein wirtschaftlicher Befund.
7. Weil der Schattenpreis nur für die **nächste** Einheit gilt und sein Gültigkeitsbereich endet, sobald ein anderer Engpass bindend wird. Realistische Lockerungen sind unterschiedlich groß — eine

Sonderschicht bringt 50 Einheiten, ein nachverhandelter Vertrag 10. Der Preis je Einheit sagt daher nichts darüber, welcher Hebel insgesamt am meisten bringt; dafür muss man die Lockerung rechnen, nicht den Preis multiplizieren.

Zurück zum **Wegweiser** *oder weiter zu* **Anhang B — Modellierungsmuster**

A.23 Lösungen zu Kapitel „Testen, Messen, Ausliefern“

23.1 — Den Schnellstart reparieren. Zwei Wege: (a) **Abrunden statt runden** — `np.floor` statt `np.round`. Das ist immer zulässig, weil weniger produzieren nie eine Kapazität sprengt, kostet aber Deckungsbeitrag und ist im Allgemeinen **nicht** die optimale ganzzahlige Lösung. (b) **Ganzzahlig modellieren** — `linprog(..., integrality=1)` bzw. `IntVar`. Das ist der richtige Weg.

Warum das ein eigenes Kapitel wert ist (Kapitel 6): Runden ist nicht nur ungenau, sondern kann *beliebig* danebenliegen. Es gibt Instanzen, bei denen die gerundete LP-Lösung nicht nur suboptimal, sondern unzulässig ist — genau das zeigt der Schnellstart — und andere, bei denen zwischen gerundetem LP und echtem Optimum Welten liegen (`Runden_Gegenbeispiel.py`).

23.2 — Eine Invariante mehr.

```
def test_kapazitaeten_verdoppeln(bauer):
    problem = schreinerei()
    doppelt = Produktionsproblem(
        produkte=problem.produkte,
        kapazitaeten={r: 2 * k for r, k in problem.kapazitaeten.items()})
    basis, gross = bauer(problem), bauer(doppelt)
    assert gross.zielwert == pytest.approx(2 * basis.zielwert, rel=1e-9)
```

Bei einem **LP** gilt das exakt: Der zulässige Bereich wird um den Faktor 2 gestreckt, und weil die Zielfunktion linear ist, skaliert das Optimum mit. Bei einem **MILP** gilt es nicht: Die Ganzzahligkeitsbedingung skaliert nicht mit. Aus 30,67 wird beim Verdoppeln 61,33 — und ob dazwischen eine bessere ganzzahlige Lösung liegt, hängt vom Einzelfall ab. Der Test wäre dort also falsch.

23.3 — Eine eigene Mutation. Zwei lohnende Kandidaten:

```
("Kapazitaetspruefung mit falschem Vergleich",
 "if ist > grenze + toleranz:", "if ist < grenze - toleranz:"),
("Toleranz mit falschem Vorzeichen",
 "if (mengen < -toleranz).any():", "if (mengen < toleranz).any():"),
```

Die erste wird getötet (`test_pruefung_findet_kapazitaetsverletzung`). Die zweite ist interessanter: Sie macht die Prüfung *strenger* statt schwächer — eine Lösung mit einer Menge von exakt 0 würde als negativ beanstandet. Ob sie überlebt, hängt daran, ob die Suite eine Instanz enthält, in der ein Produkt mit Menge 0 vorkommt. In der Schreinerei ist das nicht der Fall — die Mutation überlebt also und zeigt eine echte Lücke: Es fehlt eine Testinstanz, in der ein Produkt nicht produziert wird.

23.4 — Der Benchmark mit MILP. Zu erwarten ist, dass sich die Reihenfolge ändert. Beim reinen LP entscheidet vor allem der Modellaufbau in Python; beim MILP verschiebt sich das Gewicht zum

Lösen, und dort spielen die Branch-and-Bound-Heuristiken der Bibliotheken gegeneinander. Die Spalte „Anteil” sollte bei allen deutlich **fallen** — nicht weil der Aufbau schneller würde, sondern weil das Lösen langsamer wird. Genau deshalb steht im Kapitel, dass die Tabelle nichts über MILPs sagt.

23.5 — Der Dienst mit Zeitlimit. Der Modellbauer in `or_kern.py` nimmt bisher kein Zeitlimit entgegen — das ist der erste Schritt (bei GLOP `solver.SetTimeLimit(millisekunden)`). Danach im Arbeiter durchreichen und den Status auswerten: Liefert der Solver `ZEITLIMIT`, ist `stand` weiterhin `gescheitert`, aber mit einer anderen Begründung als bei `UNZULAESSIG` — der Unterschied zwischen „rechne länger” und „ändere das Modell” (Kapitel 6).

Für den Test braucht es eine Instanz, die das Limit reißt. Ein LP eignet sich schlecht dafür; nehmen Sie ein MILP mit einigen hundert Binärvariablen und ein Limit von 0,1 Sekunden.

Finde den Denkfehler — „Die Suite ist grün, das Modell stimmt”

Alle Prüfungen benutzen dieselbe falsche Zahl.

Die Abnahmeprüfung rechnet den Verbrauch mit `problem.verbrauchsmatrix()` nach — also mit denselben 0,4 Stunden, mit denen der Solver gerechnet hat. Der Plan ist *bezogen auf die hinterlegten Daten* vollkommen korrekt: Er hält jede Kapazität ein, der Zielwert passt zu den Mengen, jede Invariante gilt. Der Fehler steckt nicht im Modell, sondern **in der Wirklichkeit dahinter**.

Kein Test dieser Welt findet das, solange er aus derselben Datenquelle liest. Genau darauf weist der Schnellstart hin: Die Prüfung kommt aus einer anderen *Richtung*, aber nicht aus einer anderen *Quelle*.

Was geholfen hätte — drei Dinge, keines davon ein Unit-Test:

1. **Plausibilitätsgrenzen auf den Stammdaten.** „Lackierzeit je Stück zwischen 0,5 und 20 Stunden” ist eine fachliche Aussage, die man in das Pydantic-Modell schreiben kann. 0,4 wäre beim Einlesen aufgeflogen — dieselbe Idee wie die Kapazitätsprüfung `PositiveZahl`, nur mit fachlichen statt technischen Grenzen.
2. **Abgleich von Plan und Wirklichkeit.** Der geplante Lackierverbrauch gegen den tatsächlich gebuchten aus der Betriebsdatenerfassung, einmal pro Woche. Eine systematische Abweichung um den Faktor 10 fällt in der ersten Woche auf. Das ist die Überwachung aus Kapitel 22, angewandt auf die Eingabe statt auf den Solver.
3. **Vieraugenprinzip bei Stammdatenänderungen.** Unspektakulär und wirksam.

   Die allgemeine Lehre Tests prüfen die Übereinstimmung von Code und Absicht. Ob die **Daten** stimmen, ist eine andere Frage, und sie wird nicht im Testrahmen beantwortet, sondern durch Plausibilitäts- grenzen beim Einlesen und durch den Abgleich mit der Wirklichkeit im Betrieb.

Micro-Quiz

1. b) Es gibt keine unabhängige Quelle für die richtige Antwort — sonst bräuchte man den Solver nicht. (a) ist bei festgelegtem Seed und Zeitlimit meist beherrschbar; (c) löst man mit `pytest.approx`, das ist kein grundsätzliches Hindernis.

2. b) Überlebt heißt: Die Suite bleibt grün, obwohl der Code jetzt falsch ist. Sie hätte diesen Fehler durchgehen lassen. (a) verwechselt „von den Tests nicht bemerkt" mit „harmlos" — die beiden Überlebenden im Kapitel waren gerade nicht harmlos.

3. b) Gut drei Viertel der Zeit gehen in den Modellaufbau in Python. Ein schnellerer Solver würde am verbleibenden Viertel ansetzen. (a) ist der naheliegende Fehlschluss: Gemessen wurde nicht der Solver, sondern die Bindung davor.

Selbsttest

1. **Eigenschaften** (Kapazitäten eingehalten), **Invarianten** (Produktreihenfolge ändert nichts), **Regression** (die Schreinerei mit 10 800 €), **Fehlerfälle** (die Prüfung schlägt bei einer kaputten Lösung an).
2. Weil ein Test, der nur GLOP sieht, nicht unterscheiden kann, ob eine Eigenschaft vom Modell oder von der Bibliothek kommt. Läuft er über beide, muss die Aussage im Modell liegen.
3. „Grün" heißt nur, dass kein Test fehlgeschlagen ist — das gilt auch für eine Suite aus lauter `assert True`. „Prüft etwas" heißt, dass die Tests bei einem eingebauten Fehler rot würden; genau das misst der Mutationstest.
4. Weil sonst nicht erkennbar ist, welche Hälfte die Zeit kostet. Im Kapitel gehen bei OR-Tools 78 % in den Aufbau — wer nur die Gesamtzeit sieht, wechselt den Solver und ändert damit fast nichts.
5. Weil die Rechnung Sekunden bis Minuten dauert. Eine synchrone Antwort läuft in den Timeout des Reverse Proxy und blockiert währenddessen einen Arbeiter. `202` heißt „angenommen, noch nicht fertig" — genau die richtige Aussage.
6. Ein Threadpool genügt, wenn der Solver den GIL freigibt — das tun die C++-Bibliotheken (OR-Tools, HiGHS) während `Solve()`, im Kapitel mit Faktor 3,5 bei vier Threads gemessen. Eine in reinem Python geschriebene Heuristik hält den GIL; für sie braucht es Prozesse.

Anhang B: Katalog der Modellierungsmuster

Wofür dieser Anhang gedacht ist: Sie sitzen vor einem konkreten Problem und wissen nicht, wie Sie eine bestimmte Regel in ein Modell bekommen. Suchen Sie hier das Muster, das zu Ihrer Formulierung passt. Jeder Eintrag nennt die Regel in Alltagssprache, die mathematische Formulierung, den Code und die Fallstricke.

Übersicht

#	Muster	Umgangssprachlich	Wo im Buch
Logische Schalter			
B1	Aktivierungsschalter	„Wenn genutzt, dann Fixkosten“	Kapitel 6
B2	Semikontinuierlich	„Entweder 0 oder mindestens L“	Kapitel 6
B3	Implikation	„Wenn A, dann auch B“	Kapitel 6
B4	Entweder-Oder	„A oder B, aber nicht beides“	Kapitel 6
B5	Exklusiv-Oder	„Genau eines von N“	Kapitel 7
B6	Kardinalität	„Höchstens K von N“	Kapitel 6
B7	Bedingte Kopplung	„Wenn A und B, dann C“	Kapitel 6
Mengen und Grenzen			
B8	Weiche Grenze	„Möglichst nicht über X“	Kapitel 7
B9	Gestaffelte Preise	„Erste 100 Stück billiger“	—
B10	Absolutbetrag	„Abweichung nach oben wie unten“	Kapitel 20
B11	Min/Max in der Zielfunktion	„Den Schlechtesten verbessern“	Kapitel 7
B12	Verhältnis-Bedingung	„Anteil mindestens 30 %“	Kapitel 19
B25	Mindestabnahme im Zeitraum	„Entweder gar nicht oder 500 im Jahr“	Kapitel 16
B27	Budgetlimit	„Mehr als 2 Mio. gibt es nicht“	Kapitel 6
Zeit und Reihenfolge			
B13	Vorrangbeziehung	„B erst nach A“	Kapitel 7
B14	Nichtüberlappung	„Eine Maschine, ein Job“	Kapitel 7
B15	Kumulative Ressource	„Höchstens 3 gleichzeitig“	Kapitel 7
B16	Gleitendes Fenster	„Höchstens 5 Tage in Folge“	Kapitel 7
B17	Umrüstkosten	„Wechsel kostet extra“	Kapitel 7
B26	Rüstzeit als Kapazität	„Umbauen kostet Maschinenstunden“	Kapitel 17

#	Muster	Umgangssprachlich	Wo im Buch
Robustheit und Diagnose			
B18	Schlupf gegen Unlösbarkeit	„Regel notfalls brechen“	Kapitel 22
B19	Hierarchische Ziele	„Erst A, dann B optimieren“	Kapitel 22
B20	Symmetriebrechung	„Gleiche Objekte nicht doppelt zählen“	Kapitel 10
B21	Worst-Case-Abzug	„Gegen Schätzfehler absichern“	Kapitel 12
Netzwerke			
B22	Flusserhaltung	„Was reinkommt, geht raus“	Kapitel 8
B23	Zuordnung 1:1	„Jeder genau eine Aufgabe“	Kapitel 8
B24	Subtour-Eliminierung	„Keine isolierten Kreise“	Kapitel 8

Umgekehrte Richtung: Wer von einem Satz aus der Besprechung kommt und das Muster sucht, findet in Abschnitt 4.6 ein Lexikon, das genau so herum aufgebaut ist — und die Wendungen benennt, bei denen es keine eindeutige Übersetzung gibt.

Die Spalte „Wo im Buch“ nennt das Kapitel, in dem das Muster **im Zusammenhang** vorkommt — nicht die einzige Stelle, an der es taugt. Die Nummern entstehen beim Bauen; im Quelltext dieses Anhangs steht keine einzige.

Logische Schalter

B1 — Aktivierungsschalter (Fixkosten)

Regel. „Wenn überhaupt etwas produziert wird, fallen Rüstkosten F an.“

$$x \le M\,y, \qquad y \in \{0,1\},\ x \ge 0$$

Zielfunktion: $\cdots + F\,y$

```
modell.Add(x <= M * y)              # CP-SAT
# LP/MILP:  Zeile  x - M*y <= 0
```

□ M **so klein wie möglich** — idealerweise die ohnehin vorhandene Kapazitätsgrenze von x. Zu großes M macht die LP-Relaxation wertlos (Abschnitt 6.5).

B2 — Semikontinuierliche Variable

Regel. „Entweder gar nicht oder mindestens L (und höchstens U).“

$$L\,y \le x \le U\,y, \qquad y \in \{0,1\}$$

Typisch für Mindestordergrößen, Mindestlosgrößen, Mindestabnahmemengen.

B3 — Implikation

Regel. „Wenn A gewählt wird, muss auch B gewählt werden.“

$$y_A \le y_B$$

```
modell.AddImplication(y_a, y_b)     # CP-SAT, gleichwertig und lesbarer
```

Für die Umkehrung („nur wenn“) einfach vertauschen. Für Äquivalenz: $y_A = y_B$.

B4 — Entweder-Oder (disjunktive Bedingung)

Regel. „Es muss $f(\mathbf{x}) \le b_1$ **oder** $g(\mathbf{x}) \le b_2$ gelten.“

$$f(\mathbf{x}) \le b_1 + M(1-y), \qquad g(\mathbf{x}) \le b_2 + My$$

```
# CP-SAT: viel eleganter ueber Reifizierung
modell.Add(f_ausdruck <= b1).OnlyEnforceIf(y)
modell.Add(g_ausdruck <= b2).OnlyEnforceIf(y.Not())
```

B5 — Exklusiv-Oder

Regel. „Genau eine der Optionen wird gewählt.“

$$\sum_{j=1}^{N} y_j = 1$$

```
modell.AddExactlyOne(y)             # bzw. AddAtMostOne / AddBoolOr
```

B6 — Kardinalität

Regel. „Höchstens (mindestens, genau) K von N.“

$$\sum_{j} y_j \le K \qquad (\ge K,\ = K\ )$$

B7 — Bedingte Kopplung (Konjunktion)

Regel. „Wenn A **und** B, dann auch C."

$$y_A + y_B - 1 \le y_C$$

Prüfen Sie die vier Fälle: nur bei $y_A = y_B = 1$ erzwingt die Ungleichung $y_C \ge 1$.

Für „Wenn A **oder** B, dann C": $y_A \le y_C$ **und** $y_B \le y_C$.

Mengen und Grenzen

B8 — Weiche Grenze mit Strafkosten

Regel. „Möglichst nicht über b — wenn doch, kostet es."

$$f(\mathbf{x}) \le b + s, \qquad s \ge 0$$

Zielfunktion: $\cdots + c_{\text{Strafe}} \cdot s$

Das ist das wichtigste Muster überhaupt für praxistaugliche Modelle (siehe B18).

B9 — Gestaffelte Preise (stückweise linear)

Regel. „Die ersten 100 Stück kosten 5 €, danach 8 €."

Variablen je Stufe: $x = x_1 + x_2$ mit $0 \le x_1 \le 100$, $x_2 \ge 0$; Kosten $5x_1 + 8x_2$.

> □ Das funktioniert **nur bei steigenden** Preisen (konvexe Kostenfunktion) ohne Binärvariablen — der Optimierer füllt dann automatisch erst die billige Stufe. Bei **fallenden** Preisen (Mengenrabatt, konkav) braucht man Binärvariablen je Stufe, sonst „schummelt" das Modell.

B10 — Absolutbetrag / Abweichung

Regel. „Die Abweichung vom Zielwert soll klein sein, egal in welche Richtung."

$$|x - z| \le d \quad \Longleftrightarrow \quad x - z \le d \ \text{ und } \ z - x \le d$$

In der **Zielfunktion** (Minimierung von $|x - z|$) genügt:

$$\min d \quad \text{u. d. N.} \quad x - z \le d, \quad z - x \le d$$

```
turnover = cp.norm1(w - w_alt)      # CVXPY macht das automatisch
```

B11 — Min/Max in der Zielfunktion

Regel. „Der am schlechtesten gestellte Beteiligte soll möglichst gut dastehen“ (Maximin/Fairness).

$$\max t \quad \text{u. d. N.} \quad t \le f_i(\mathbf{x}) \;\; \forall i$$

```
modell.AddMaxEquality(max_var, liste)   # CP-SAT
modell.AddMinEquality(min_var, liste)
# Fairness ueber die Spannweite:
modell.Add(spannweite == max_var - min_var)   # dann minimieren
```

B12 — Verhältnis-Bedingung

Regel. „Der Anteil von Gruppe G soll mindestens 30 % betragen.“

$$\frac{\sum_{i\in G} x_i}{\sum_i x_i} \ge 0{,}3 \quad\Longleftrightarrow\quad \sum_{i\in G} x_i \ge 0{,}3 \sum_i x_i$$

 Brüche immer wegmultiplizieren — ein Quotient von Variablen ist nichtlinear und meist nicht konvex. Nach dem Umstellen ist die Bedingung linear.

B25 — Mindestabnahmemenge über einen Zeitraum

Regel. „Entweder wir arbeiten mit diesem Lieferanten gar nicht, oder wir nehmen ihm im Jahr mindestens 500 Einheiten ab.“

Ein **Schalter für den ganzen Zeitraum**, nicht je Periode:

$$Q\,y \le \sum_t x_t \le U\,y, \qquad y \in \{0,1\},\; x_t \ge 0$$

```
# y_s ist EINE Variable je Lieferant - nicht eine je Lieferant UND Periode
y = {s: modell.NewBoolVar(f"vertrag_{s}") for s in lieferanten}
for s in lieferanten:
    jahresmenge = sum(x[s, t] for t in perioden)
    modell.Add(jahresmenge >= MINDESTMENGE[s] * y[s])
    modell.Add(jahresmenge <= JAHRESKAPAZITAET[s] * y[s])   # koppelt x an y
```

 Die häufigste Verwechslung: $L\,y_t \le x_t$ je Periode (B2) ist ein *anderes* Modell — es verlangt in jeder einzelnen Periode eine Mindestmenge und ist erheblich strenger. Wer den Jahresvertrag so formuliert, erzeugt ein unlösbares Modell und sucht den Fehler dann in den Daten.

 Die obere Kopplung nicht vergessen. Ohne $\sum_t x_t \le U\,y$ kann das Modell $y = 0$ setzen und trotzdem einkaufen — der Vertrag gilt dann als nicht geschlossen, die Ware fließt aber. Das ist der *Trickle Flow* aus Abschnitt 6.5, nur andersherum. U ist die Jahreskapazität des Lieferanten, keine runde Zahl.

B27 — Budgetlimit

Regel. „Alle Maßnahmen zusammen dürfen 2 Mio. € nicht überschreiten."

$$\sum_i c_i x_i \le B$$

Mit $x_i \in \{0,1\}$ ist das ein **Rucksackproblem** — dasselbe Muster, das Kapitel 6 an `Rucksack.py` vorrechnet und Kapitel 10 als Pricing-Teilproblem wiederverwendet.

```
modell.Add(sum(KOSTEN[i] * x[i] for i in massnahmen) <= BUDGET)
# Mehrere Toepfe: je Topf eine Zeile - NICHT die Summe ueber alle Toepfe
for topf, grenze in BUDGETS.items():
    modell.Add(sum(KOSTEN[i] * x[i] for i in massnahmen
                   if TOPF[i] == topf) <= grenze)
```

 Der Schattenpreis des Budgets ist die Zahl, nach der die Geschäftsführung fragt: „Was bringt der nächste Euro?" Bei einem **LP** ist er direkt ablesbar. Bei **Ganzzahligkeit gibt es ihn nicht** — die Dualwerte der Relaxation sind keine gültige Antwort. Rechnen Sie stattdessen mit erhöhtem Budget neu und vergleichen Sie die Zielwerte (Abschnitt 5.9).

 Die LP-Relaxation verspricht zu viel. Sie darf die letzte Maßnahme anteilig kaufen und liefert deshalb eine Schranke, die spürbar über dem tatsächlich Erreichbaren liegen kann. Wer sie als Prognose berichtet, verspricht Geld, das nicht kommt.

Zeit und Reihenfolge

B13 — Vorrangbeziehung

Regel. „Arbeitsgang B darf erst beginnen, wenn A fertig ist."

$$\text{start}_B \ge \text{ende}_A$$

Mit Mindestwartezeit w: $\text{start}_B \ge \text{ende}_A + w$.

B14 — Nichtüberlappung

Regel. „Eine Maschine bearbeitet nur einen Job gleichzeitig."

```
modell.AddNoOverlap([intervall_1, intervall_2, ...])
```

In MILP bräuchte man je Paar eine Disjunktion (B4) — bei k Jobs sind das $\binom{k}{2}$ Konstruktionen. **Nehmen Sie hier CP-SAT.**

B15 — Kumulative Ressource

Regel. „Zu keinem Zeitpunkt dürfen mehr als 3 Arbeiten gleichzeitig laufen“ bzw. „die Stromlast darf 500 kW nie übersteigen“.

```
modell.AddCumulative(intervalle, bedarfe, kapazitaet)
```

B16 — Gleitendes Fenster

Regel. „Höchstens 5 Arbeitstage in Folge“ / „mindestens 2 freie Tage je Woche“.

$$\sum_{u=t}^{t+5} x_u \le 5 \qquad \forall t$$

```
for t in range(len(tage) - 5):
    modell.Add(sum(x[p, u] for u in range(t, t + 6)) <= 5)
```

B17 — Umrüst- bzw. Wechselkosten

Regel. „Ein Produktwechsel kostet Rüstzeit.“

Hilfsvariable $z_t \in \{0,1\}$ = „in Periode t wird gewechselt“:

$$z_t \ge y_{j,t} - y_{j,t-1} \qquad \forall j,t$$

Zielfunktion: $\cdots + c_{\text{Ruest}} \sum_t z_t$. Für **reihenfolgeabhängige** Rüstzeiten: `AddCircuit` mit Übergangsmatrix.

B26 — Rüstzeit als Kapazitätsverbrauch

Regel. „Das Umrüsten kostet nicht nur Geld, es kostet **Maschinenstunden** — und die fehlen dann für die Produktion.“

B17 verbucht den Wechsel in der *Zielfunktion*. Sobald die Maschine ausgelastet ist, gehört er zusätzlich in die *Kapazitätszeile*:

$$\sum_j a_j\, x_{j,t} \;+\; \sum_j r_j\, y_{j,t} \;\le\; C_t \qquad \forall t$$

mit a_j Stückzeit, r_j Rüstzeit, $y_{j,t} \in \{0,1\}$ = „Produkt j läuft in Periode t“ (gekoppelt über B1: $x_{j,t} \le M\, y_{j,t}$).

```
for t in perioden:
    modell.Add(sum(STUECKZEIT[j] * x[j, t] for j in produkte)
               + sum(RUESTZEIT[j] * y[j, t] for j in produkte) <= KAPAZITAET[t])
```

> **Wer die Rüstzeit nur als Kosten führt, erhält Pläne, die in der Halle nicht laufen.** Das Modell verteilt die Produktion dann auf viele kleine Lose, weil ein zusätzlicher Wechsel zwar etwas kostet, aber keine Zeit verbraucht — die Rechnung geht auf dem Papier auf und in der Schicht nicht.

Reihenfolgeabhängig (r_{ij} statt r_j — von Weiß auf Schwarz ist schneller als umgekehrt) ist es kein Kapazitätsproblem mehr, sondern ein Rundreiseproblem: `AddCircuit` mit der Übergangsmatrix als Kantengewicht (B24, Kapitel 7).

Robustheit und Diagnose

B18 — Schlupfvariablen gegen Unlösbarkeit

Das wichtigste Muster für den Produktivbetrieb.

Regel. „Diese Bedingung soll gelten — aber lieber ein schlechter Plan als gar keiner."

$$\sum_p x_{p,s} + u_s = 1, \qquad u_s \in \{0, 1\}$$

Zielfunktion: $\cdots + 10\,000 \cdot u_s$

Die Strafe muss **hoch genug** sein, dass der Solver sie nur im Notfall in Kauf nimmt, aber **endlich**, damit es überhaupt eine Lösung gibt. Faustregel: eine Größenordnung über der Summe aller weichen Ziele.

B19 — Hierarchische Ziele (lexikografisch)

Regel. „Erst die Besetzung sicherstellen, dann die Fairness optimieren."

Variante A — Gewichtung: Strafen um Größenordnungen staffeln ($10\,000 \gg 100 \gg 1$). Einfach, aber bei extremen Skalenunterschieden numerisch heikel.

Variante B — Zweistufig lösen (sauberer):

```
# Stufe 1: nur das Hauptziel
modell.Minimize(unbesetzte_schichten)
loeser.Solve(modell)
bestwert = loeser.ObjectiveValue()

# Stufe 2: Hauptziel fixieren, Nebenziel optimieren
modell.Add(unbesetzte_schichten <= int(bestwert))
modell.Minimize(unfairness)
loeser.Solve(modell)
```

B20 — Symmetriebrechung

Regel. „Drei identische Maschinen — der Solver soll nicht alle Vertauschungen durchprobieren."

$$\text{start}_1 \le \text{start}_2 \le \text{start}_3 \qquad \text{bzw.} \qquad \sum_j x_{1j} \ge \sum_j x_{2j}$$

Ohne Symmetriebrechung durchsucht Branch-and-Bound $k!$ gleichwertige Lösungen. **Eine einzige Ordnungsbedingung kann die Laufzeit um Größenordnungen senken.**

B21 — Worst-Case-Abzug (robuste Formulierung)

Regel. „Rechne nicht mit dem geschätzten Wert, sondern mit dem ungünstigsten plausiblen."

Bei Box-Unsicherheit $\mu_i \in [\hat{\mu}_i - \delta_i, \hat{\mu}_i + \delta_i]$ und $w \geq 0$:

$$\hat{\mu}^\top \mathbf{w} \longrightarrow \hat{\mu}^\top \mathbf{w} - \delta^\top \mathbf{w}$$

Abgestuft (Bertsimas/Sim): nur die Γ ungünstigsten gleichzeitig:

```
abzug = cp.sum_largest(cp.multiply(delta, w), Gamma)
```

Netzwerke

B22 — Flusserhaltung

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{k:(k,i)\in E} x_{ki} = b_i \qquad \forall i$$

$b_i > 0$ Quelle, $b_i < 0$ Senke, $b_i = 0$ Umschlagknoten. **Voraussetzung:** $\sum_i b_i = 0$ — sonst unlösbar (Dummy-Knoten einführen).

B23 — Zuordnung 1:1

$$\sum_j x_{ij} = 1 \;\forall i, \qquad \sum_i x_{ij} = 1 \;\forall j, \qquad x_{ij} \geq 0$$

> **Ganzzahligkeit nicht fordern!** Die Matrix ist total unimodular; ein LP-Solver liefert automatisch 0/1-Lösungen. Für reine Zuordnungen ist `scipy.optimize.linear_sum_assignment` (Ungarischer Algorithmus, $O(n^3)$) noch deutlich schneller.

B24 — Subtour-Eliminierung

MTZ (einfach, aber schwach):

$$u_i - u_j + C x_{ij} \leq C - d_j \qquad \forall i \neq j$$

Besser in der Praxis: `AddCircuit` in CP-SAT oder die Routing-Bibliothek von OR-Tools.

Ein Wort zur Auswahl

Wenn mehrere Muster passen, entscheiden Sie nach dieser Reihenfolge:

1. **Gibt es ein globales Constraint dafür?** (`AddAllDifferent`, `AddNoOverlap`, `AddCumulative`, `AddCircuit`) → nehmen Sie es. Es propagiert stärker und ist lesbarer.
2. **Kommt man ohne Big-M aus?** (z. B. B3 statt B4) → ja, dann so.
3. **Muss es wirklich hart sein?** → sonst B8/B18.
4. **Ist M so klein wie möglich?** → prüfen.

Anhang C: Fehlerdiagnose-Handbuch

Wofür dieser Anhang gedacht ist: Etwas funktioniert nicht. Suchen Sie unten das Symptom, das Sie beobachten, und arbeiten Sie die Diagnoseschritte der Reihe nach ab.

Programme:
`Konfliktsuche.py`

Symptom-Schnellübersicht

Symptom	Abschnitt
Solver meldet `INFEASIBLE`	C1
`INFEASIBLE`, aber keine einzelne Bedingung ist schuld	C1, Deletion Filter
Solver meldet `UNBOUNDED`	C2
Solver läuft ewig / Timeout	C3
Ergebnis ist offensichtlich unsinnig	C4
Ergebnis ändert sich bei kleinsten Datenänderungen stark	C5
Schattenpreise sehen falsch aus	C6
Zwei Solver liefern verschiedene Ergebnisse	C7
CVXPY wirft `DCPError`	C8
`ImportError` bei ortools/highspy	C9
Backtest sieht zu gut aus	C10
Ergebnisse sind falsch beschriftet	C11

C1 — INFEASIBLE

Bedeutung. Es gibt **keinen einzigen Punkt**, der alle Bedingungen gleichzeitig erfüllt. Das ist eine Aussage über Ihr Modell, nicht über den Solver.

Diagnose in fünf Schritten

Schritt 1 — Trivialprüfungen von Hand. Rechnen Sie die offensichtlichen Bilanzen nach, **bevor** Sie den Solver befragen:

```
assert summe_angebot >= summe_bedarf, "Angebot deckt Bedarf nicht"
assert anzahl_personal * max_schichten >= anzahl_schichten, "zu wenig Personal"
for slot, fach in enumerate(faecher):
    qualifiziert = [p for p in personal if fach in qualifikation[p]]
    assert qualifiziert, f"Fuer Slot {slot} ({fach}) gibt es niemanden"
```

Erfahrungsgemäß findet dieser Schritt **die Mehrzahl** aller Fälle.

Schritt 2 — Bedingungen einzeln abschalten. Kommentieren Sie Bedingungsgruppen nacheinander aus. Wird das Modell lösbar, ist die zuletzt entfernte Gruppe (mit-)verantwortlich.

Schritt 3 — Schlupfvariablen einbauen (das wirksamste Mittel).

```
# statt:  modell.Add(bedingung)
schlupf = modell.NewIntVar(0, obergrenze, f"schlupf_{name}")
modell.Add(linke_seite <= rechte_seite + schlupf)
strafen.append(10_000 * schlupf)
```

Nach dem Lösen zeigen die Schlupfvariablen mit Wert > 0 **präzise, welche Bedingung** wo und um wie viel verletzt werden musste. Das ist zugleich die produktionstaugliche Lösung (Muster B18).

Schritt 4 — Zeitfenster und Erreichbarkeit prüfen. Bei Scheduling und Routing: Ist jeder Termin überhaupt physisch erreichbar?

```
for kunde, (frueh, spaet) in enumerate(zeitfenster):
    assert distanz[depot][kunde] <= spaet, f"Kunde {kunde} nicht rechtzeitig erreichbar"
```

Schritt 5 — Rundungs- und Einheitenfehler. Sind Kapazitäten in Stunden, Verbräuche aber in Minuten? Rechnen CP-SAT-Modelle mit ganzen Zahlen, wo Sie Nachkommastellen brauchen? Ein klassischer Fall: `sum(w) == 1` mit Gleitkommazahlen — nutzen Sie eine Toleranz oder rechnen Sie in Ganzzahlen (Promille).

Wenn die fünf Schritte nicht reichen: den Konflikt einkreisen

Warum Schritt 2 so oft ins Leere läuft. Bedingungen einzeln abzuschalten funktioniert nur, solange es *einen* Schuldigen gibt. Überlagern sich zwei unabhängige Widersprüche, bleibt das Modell nach jeder einzelnen Abschaltung unlösbar — die Suche endet mit null Treffern, obwohl beide Widersprüche unverändert im Modell stehen. Genau dann steht man vor einem Modell mit tausenden Restriktionen und einem Statuscode.

Der Deletion Filter dreht die Frage um. Er fragt nicht „ist *diese* Bedingung schuld?“, sondern „wird *diese* Bedingung für den Widerspruch überhaupt gebraucht?“ — und das ist eine Frage, die sich beantworten lässt:

> Nimm eine Bedingung versuchsweise heraus. Bleibt der Rest unlösbar, wurde sie nicht gebraucht: weg damit, endgültig. Wird der Rest lösbar, war sie beteiligt: sie bleibt.

Nach genau einem Durchlauf über alle n Bedingungen ist die übrig gebliebene Menge **unreduzierbar** — entfernt man aus ihr irgendeine Bedingung, ist der Rest lösbar. Das ist ein *Irreducible Infeasible Subset*, kurz IIS. Kommerzielle Solver bieten das als fertige Funktion an (`computeIIS` bei Gurobi, der Conflict Refiner bei CPLEX); im Open-Source-Werkzeugkasten dieses Buchs gibt es sie nicht — sie ist aber in zwanzig Zeilen selbst geschrieben, und sie kostet n Aufrufe des Solvers statt 2^n durchprobierter Teilmengen.

```
#!/usr/bin/env python3

# Konfliktsuche.py
"""
Anhang Fehlerdiagnose: Den Konflikt finden, der INFEASIBLE verursacht.

Ein Deletion Filter (auch: IIS, Irreducible Infeasible Subset) isoliert aus
einem unloesbaren Modell eine kleinste widerspruechliche Teilmenge von
Bedingungen: Nimmt man aus ihr auch nur eine einzige Bedingung heraus, ist
der Rest wieder loesbar.

Das Programm zeigt sechs Dinge:
  1. Der Solver meldet INFEASIBLE - und sonst nichts.
  2. Die naive Suche ("jede Bedingung einmal weglassen") findet hier gar nichts.
  3. Der Deletion Filter findet einen Konflikt in n Solveraufrufen.
  4. Ein Konflikt ist selten einer: nach der Reparatur folgt der naechste.
  5. "Den" kleinsten Konflikt gibt es nicht - dieses Modell enthaelt neun.
  6. Welchen davon man zu sehen bekommt, steuert die Pruefreihenfolge.

Abgrenzung zu Infeasibility_Diagnose.py (Kapitel Praxisfallen): Dort geht es
um die Relaxation - das Modell soll trotz Widerspruch eine brauchbare Antwort
liefern. Hier geht es um die Diagnose - welche Bedingungen widersprechen sich
ueberhaupt. Beides zusammen ergibt den Umgang mit INFEASIBLE in Produktion.

Benoetigt: numpy, scipy
"""
```

```
from __future__ import annotations

import numpy as np
from scipy.optimize import linprog

PRODUKTE = ["Rahmen", "Gehaeuse", "Deckel", "Traeger", "Halter"]

# Jede Bedingung traegt einen sprechenden Namen - das ist keine Kosmetik,
# sondern die Voraussetzung dafuer, dass der Befund lesbar wird.
# (Name, Koeffizienten je Produkt, Richtung, rechte Seite)
BEDINGUNGEN: list[tuple[str, list[float], str, float]] = [
    ("Kapazitaet Montage",       [3, 2, 1, 4, 2],      "<=",   400),
    ("Kapazitaet Lackieren",     [2, 3, 0, 1, 0],      "<=",   150),
    ("Kapazitaet Pruefung",      [1, 1, 1, 1, 1],      "<=",   300),
    ("Liefervertrag Rahmen",     [1, 0, 0, 0, 0],      ">=",    40),
    ("Liefervertrag Gehaeuse",   [0, 1, 0, 0, 0],      ">=",    30),
    ("Liefervertrag Deckel",     [0, 0, 1, 0, 0],      ">=",    20),
    ("Liefervertrag Traeger",    [0, 0, 0, 1, 0],      ">=",    25),
    ("Marktgrenze Rahmen",       [1, 0, 0, 0, 0],      "<=",   120),
    ("Marktgrenze Gehaeuse",     [0, 1, 0, 0, 0],      "<=",    90),
    ("Marktgrenze Deckel",       [0, 0, 1, 0, 0],      "<=",   150),
    ("Marktgrenze Halter",       [0, 0, 0, 0, 1],      "<=",   200),
    ("Mindestumsatz",            [90, 70, 30, 60, 20], ">=", 12000),
    ("Sortimentsbreite",         [0, 0, 1, 1, 1],      ">=",   100),
    ("Lackierbudget Schicht 2",  [0, 1, 0, 1, 0],      "<=",    55),
]

# Welche Bedingungen koennte ein Planer im Ernstfall wirklich veraendern?
# Kapazitaeten lassen sich durch Sonderschichten dehnen, Liefervertraege und
# Marktgrenzen nicht.
VERHANDELBAR = {"Kapazitaet Montage", "Kapazitaet Lackieren",
                "Kapazitaet Pruefung", "Lackierbudget Schicht 2"}

aufrufe = 0          # zaehlt jeden Solveraufruf mit - der Preis des Verfahrens


def ist_loesbar(auswahl: list[int]) -> bool:
    """Gibt es einen Punkt, der ALLE Bedingungen aus 'auswahl' erfuellt?

    Es wird nur Zulaessigkeit geprueft, keine Zielfunktion optimiert: die
    Zielfunktion ist konstant null. INFEASIBLE haengt nie an der Zielfunktion.
    """
    global aufrufe
    aufrufe += 1
    matrix, rechte_seite = [], []
    for i in auswahl:
        _, koeffizienten, richtung, grenze = BEDINGUNGEN[i]
        zeile = np.array(koeffizienten, dtype=float)
```

```
        # linprog kennt nur "<=", also ">=" durch Negation umdrehen
        matrix.append(zeile if richtung == "<=" else -zeile)
        rechte_seite.append(grenze if richtung == "<=" else -grenze)
    ergebnis = linprog(np.zeros(len(PRODUKTE)),
                       A_ub=np.array(matrix), b_ub=np.array(rechte_seite),
                       bounds=(0, None), method="highs")
    return ergebnis.status == 0


def deletion_filter(auswahl: list[int]) -> list[int]:
    """Verkleinert eine unloesbare Menge zu einer kleinsten unloesbaren Menge.

    Der Kern des Verfahrens ist eine einzige Regel: Nimm eine Bedingung
    versuchsweise heraus. Bleibt der Rest unloesbar, wurde sie fuer den
    Widerspruch nicht gebraucht - sie darf endgueltig weg. Wird der Rest
    loesbar, war sie beteiligt und muss bleiben.

    Nach genau einem Durchlauf ueber alle Bedingungen ist das Ergebnis
    unreduzierbar: n Solveraufrufe statt 2^n Teilmengen.
    """
    rest = list(auswahl)
    for i in list(auswahl):
        probe = [j for j in rest if j != i]
        if not ist_loesbar(probe):
            rest = probe
    return rest


def namen(auswahl: list[int]) -> list[str]:
    return [BEDINGUNGEN[i][0] for i in auswahl]


def teil_1_der_befund(alle: list[int]) -> None:
    print("=" * 68)
    print("1. Was der Solver sagt")
    print("=" * 68)
    print(f"Modell: {len(PRODUKTE)} Variablen, {len(BEDINGUNGEN)} Bedingungen")
    status = "OPTIMAL" if ist_loesbar(alle) else "INFEASIBLE"
    print(f"Status: {status}")
    print("\nMehr ist es nicht. Der Solver nennt keine Ursache, weil es die eine")
    print("Ursache nicht gibt: Unloesbarkeit ist eine Eigenschaft von Mengen von")
    print("Bedingungen, nicht von einzelnen Bedingungen.")


def teil_2_die_naive_suche(alle: list[int]) -> None:
    print("\n" + "=" * 68)
    print("2. Die naheliegende Idee - und warum sie hier scheitert")
    print("=" * 68)
```

```
    print("Jede Bedingung einmal weglassen und schauen, ob es dann geht:\n")
    treffer = [i for i in alle if ist_loesbar([j for j in alle if j != i])]
    for i in alle:
        print(f"   ohne {BEDINGUNGEN[i][0]:<26} {'loesbar' if i in treffer else 'weiter
        ↪  unloesbar'}")
    print(f"\nGefundene Schuldige: {len(treffer)}")
    print("Keine einzelne Bedingung ist schuld. Genau das ist der Normalfall -")
    print("und der Grund, warum diese Suche in der Praxis so oft im Nichts endet.")


def teil_3_der_filter(alle: list[int]) -> list[int]:
    global aufrufe
    print("\n" + "=" * 68)
    print("3. Der Deletion Filter")
    print("=" * 68)
    vorher = aufrufe
    konflikt = deletion_filter(alle)
    kosten = aufrufe - vorher
    print(f"{kosten} Solveraufrufe -> Konflikt aus {len(konflikt)} von "
          f"{len(BEDINGUNGEN)} Bedingungen:\n")
    for name in namen(konflikt):
        print(f"   * {name}")
    print("\nProbe auf Unreduzierbarkeit - jede einzelne davon weglassen:")
    for i in konflikt:
        rest_loesbar = ist_loesbar([j for j in konflikt if j != i])
        print(f"   ohne {BEDINGUNGEN[i][0]:<26} {'loesbar' if rest_loesbar else 'UNLOESBAR -
        ↪  nicht minimal!'}")
    print(f"\nAufwand: {kosten} Aufrufe. Alle Teilmengen durchzuprobieren waeren "
          f"2^{len(BEDINGUNGEN)} = {2 ** len(BEDINGUNGEN):,} gewesen.".replace(",", "."))
    return konflikt


def teil_4_der_naechste_konflikt(alle: list[int], konflikt: list[int]) -> None:
    print("\n" + "=" * 68)
    print("4. Ein Konflikt ist selten einer")
    print("=" * 68)
    entfernt = konflikt[0]
    rest = [i for i in alle if i != entfernt]
    print(f"Angenommen, '{BEDINGUNGEN[entfernt][0]}' laesst sich verhandeln")
    print("und wird aus dem Modell genommen. Dann ist das Modell ...")
    if ist_loesbar(rest):
        print("... loesbar. Fertig.")
        return
    print("... immer noch unloesbar. Der Filter erneut:\n")
    zweiter = deletion_filter(rest)
    for name in namen(zweiter):
        print(f"   * {name}")
    gemeinsam = set(zweiter) & set(konflikt)
```

```
    print(f"\nUeberschneidung mit dem ersten Konflikt: {len(gemeinsam)} Bedingungen")
    print("Ein zweiter, unabhaengiger Widerspruch, den der erste verdeckt hat.")
    print("Deshalb ist die Konfliktsuche eine Schleife, kein einzelner Aufruf:")
    print("reparieren, neu suchen, bis das Modell loesbar ist.")


def teil_5_die_reihenfolge(alle: list[int]) -> None:
    print("\n" + "=" * 68)
    print("5. Es gibt nicht DEN Konflikt")
    print("=" * 68)
    print("Derselbe Filter, nur eine andere Pruefreihenfolge - 200-mal gewuerfelt:\n")
    zufall = np.random.default_rng(0)
    haeufigkeit: dict[tuple[int, ...], int] = {}
    for _ in range(200):
        ordnung = [int(i) for i in zufall.permutation(alle)]
        schluessel = tuple(sorted(deletion_filter(ordnung)))
        haeufigkeit[schluessel] = haeufigkeit.get(schluessel, 0) + 1
    for schluessel, anzahl in sorted(haeufigkeit.items(), key=lambda p: -p[1]):
        print(f"   {anzahl:3d}x  {len(schluessel)} Bedingungen: "
              f"{', '.join(namen(list(schluessel)))}")
    print(f"\n{len(haeufigkeit)} verschiedene minimale Konflikte in EINEM Modell.")
    print("Der Filter liefert *einen* kleinsten Konflikt, nicht *den* kleinsten -")
    print("den gibt es nicht. Alle oben sind gleichermassen korrekt.")


def teil_6_den_befund_steuern(alle: list[int]) -> None:
    print("\n" + "=" * 68)
    print("6. Den Befund brauchbar machen")
    print("=" * 68)
    print("Der Filter wirft heraus, was er zuerst in die Hand bekommt. Wer die")
    print("unveraenderlichen Bedingungen zuerst pruefen laesst, bekommt sie eher")
    print("aus dem Befund heraus - und sieht dafuer mehr Stellschrauben.\n")
    fest = [i for i in alle if BEDINGUNGEN[i][0] not in VERHANDELBAR]
    frei = [i for i in alle if BEDINGUNGEN[i][0] in VERHANDELBAR]
    for titel, ordnung in (("unveraenderliche zuerst", fest + frei),
                           ("veraenderliche zuerst  ", frei + fest)):
        konflikt = sorted(deletion_filter(ordnung))
        stellschrauben = [i for i in konflikt if BEDINGUNGEN[i][0] in VERHANDELBAR]
        print(f"   {titel} -> {len(konflikt)} Bedingungen, davon "
              f"{len(stellschrauben)} veraenderlich")
        print(f"      {', '.join(namen(konflikt))}")
    print("\nBeide Befunde sind wahr. Nur einer davon nennt dem Planer etwas,")
    print("das er tatsaechlich tun kann.")


if __name__ == "__main__":
    alle = list(range(len(BEDINGUNGEN)))
    teil_1_der_befund(alle)
```

```
    teil_2_die_naive_suche(alle)
    konflikt = teil_3_der_filter(alle)
    teil_4_der_naechste_konflikt(alle, konflikt)
    teil_5_die_reihenfolge(alle)
    teil_6_den_befund_steuern(alle)
    print("\n" + "=" * 68)
    print(f"Insgesamt {aufrufe} Solveraufrufe fuer die gesamte Diagnose.")
    print("=" * 68)
```

Die Ausgabe:

```
====================================================================
1. Was der Solver sagt
====================================================================
Modell: 5 Variablen, 14 Bedingungen
Status: INFEASIBLE

Mehr ist es nicht. Der Solver nennt keine Ursache, weil es die eine
Ursache nicht gibt: Unloesbarkeit ist eine Eigenschaft von Mengen von
Bedingungen, nicht von einzelnen Bedingungen.

====================================================================
2. Die naheliegende Idee - und warum sie hier scheitert
====================================================================
Jede Bedingung einmal weglassen und schauen, ob es dann geht:

   ohne Kapazitaet Montage         weiter unloesbar
   ohne Kapazitaet Lackieren       weiter unloesbar
   ohne Kapazitaet Pruefung        weiter unloesbar
   ohne Liefervertrag Rahmen       weiter unloesbar
   ohne Liefervertrag Gehaeuse     weiter unloesbar
   ohne Liefervertrag Deckel       weiter unloesbar
   ohne Liefervertrag Traeger      weiter unloesbar
   ohne Marktgrenze Rahmen         weiter unloesbar
   ohne Marktgrenze Gehaeuse       weiter unloesbar
   ohne Marktgrenze Deckel         weiter unloesbar
   ohne Marktgrenze Halter         weiter unloesbar
   ohne Mindestumsatz              weiter unloesbar
   ohne Sortimentsbreite           weiter unloesbar
   ohne Lackierbudget Schicht 2    weiter unloesbar

Gefundene Schuldige: 0
Keine einzelne Bedingung ist schuld. Genau das ist der Normalfall -
und der Grund, warum diese Suche in der Praxis so oft im Nichts endet.

====================================================================
3. Der Deletion Filter
====================================================================
```

```
14 Solveraufrufe -> Konflikt aus 3 von 14 Bedingungen:

   * Kapazitaet Lackieren
   * Liefervertrag Rahmen
   * Liefervertrag Gehaeuse

Probe auf Unreduzierbarkeit - jede einzelne davon weglassen:
   ohne Kapazitaet Lackieren       loesbar
   ohne Liefervertrag Rahmen       loesbar
   ohne Liefervertrag Gehaeuse     loesbar

Aufwand: 14 Aufrufe. Alle Teilmengen durchzuprobieren waeren 2^14 = 16.384 gewesen.

======================================================================
4. Ein Konflikt ist selten einer
======================================================================
Angenommen, 'Kapazitaet Lackieren' laesst sich verhandeln
und wird aus dem Modell genommen. Dann ist das Modell ...
... immer noch unloesbar. Der Filter erneut:

   * Kapazitaet Montage
   * Liefervertrag Traeger
   * Mindestumsatz
   * Lackierbudget Schicht 2

Ueberschneidung mit dem ersten Konflikt: 0 Bedingungen
Ein zweiter, unabhaengiger Widerspruch, den der erste verdeckt hat.
Deshalb ist die Konfliktsuche eine Schleife, kein einzelner Aufruf:
reparieren, neu suchen, bis das Modell loesbar ist.

======================================================================
5. Es gibt nicht DEN Konflikt
======================================================================
Derselbe Filter, nur eine andere Pruefreihenfolge - 200-mal gewuerfelt:

    67x  3 Bedingungen: Kapazitaet Lackieren, Liefervertrag Rahmen, Liefervertrag Gehaeuse
    32x  4 Bedingungen: Kapazitaet Montage, Kapazitaet Lackieren, Marktgrenze Deckel,
 ↪  Mindestumsatz
    20x  4 Bedingungen: Kapazitaet Montage, Liefervertrag Traeger, Mindestumsatz,
 ↪  Sortimentsbreite
    19x  4 Bedingungen: Kapazitaet Montage, Liefervertrag Traeger, Mindestumsatz,
 ↪  Lackierbudget Schicht 2
    19x  4 Bedingungen: Kapazitaet Montage, Liefervertrag Traeger, Marktgrenze Gehaeuse,
 ↪  Mindestumsatz
    17x  5 Bedingungen: Kapazitaet Lackieren, Kapazitaet Pruefung, Liefervertrag Gehaeuse,
 ↪  Marktgrenze Deckel, Mindestumsatz
    11x  4 Bedingungen: Kapazitaet Montage, Liefervertrag Deckel, Liefervertrag Traeger,
 ↪  Mindestumsatz
```

```
     8x  4 Bedingungen: Kapazitaet Montage, Liefervertrag Rahmen, Liefervertrag Traeger,
↪  Mindestumsatz
     7x  4 Bedingungen: Kapazitaet Montage, Kapazitaet Lackieren, Liefervertrag Traeger,
↪  Mindestumsatz

9 verschiedene minimale Konflikte in EINEM Modell.
Der Filter liefert *einen* kleinsten Konflikt, nicht *den* kleinsten -
den gibt es nicht. Alle oben sind gleichermassen korrekt.

======================================================================
6. Den Befund brauchbar machen
======================================================================
Der Filter wirft heraus, was er zuerst in die Hand bekommt. Wer die
unveraenderlichen Bedingungen zuerst pruefen laesst, bekommt sie eher
aus dem Befund heraus - und sieht dafuer mehr Stellschrauben.

   unveraenderliche zuerst -> 4 Bedingungen, davon 2 veraenderlich
      Kapazitaet Montage, Kapazitaet Lackieren, Marktgrenze Deckel, Mindestumsatz
   veraenderliche zuerst   -> 3 Bedingungen, davon 1 veraenderlich
      Kapazitaet Lackieren, Liefervertrag Rahmen, Liefervertrag Gehaeuse

Beide Befunde sind wahr. Nur einer davon nennt dem Planer etwas,
das er tatsaechlich tun kann.

======================================================================
Insgesamt 2874 Solveraufrufe fuer die gesamte Diagnose.
======================================================================
```

Vier Dinge sind daran wichtig.

1. **Die Zulässigkeitsprüfung braucht keine Zielfunktion.** `linprog` bekommt einen Nullvektor als Ziel. Unlösbarkeit hängt nie an der Zielfunktion — wer beim Diagnostizieren die echte Zielfunktion mitschleppt, bezahlt Rechenzeit für nichts.
2. **Gefiltert werden nur die aufgeführten Bedingungen, nicht die Variablenschranken.** Die Nichtnegativität steht in `bounds` und bleibt in jedem Teilmodell stehen. Sitzt Ihr Widerspruch in den Schranken (`x >= 5` als Bound statt als Zeile), findet ihn der Filter nicht — schreiben Sie solche Grenzen dann als benannte Bedingung.
3. **Sprechende Namen sind kein Luxus.** Der Befund ist genau so brauchbar wie die Namen, die darin vorkommen. `constraint_47, constraint_112, constraint_9` ist kein Befund.
4. **Der Filter liefert *einen* kleinsten Konflikt, nicht *den* kleinsten.** Dieses Modell mit vierzehn Bedingungen enthält neun verschiedene minimale Konflikte; welchen man zu sehen bekommt, entscheidet allein die Prüfreihenfolge. Das ist kein Mangel des Verfahrens, sondern eine Eigenschaft des Problems.

Aus Punkt 4 folgt der eigentliche Praxisgriff. Weil der Filter bevorzugt das hinauswirft, was er zuerst in die Hand bekommt, überlegen Sie vor dem Lauf, welche Bedingungen Sie im Ernstfall tatsächlich ändern könnten — und lassen Sie die *unveränderlichen* zuerst prüfen. Dann bleiben eher die Stellschrauben

im Befund stehen. Im Beispiel oben liefert die Reihenfolge „Verträge und Marktgrenzen zuerst“ einen Befund mit zwei veränderlichen Kapazitäten, die umgekehrte Reihenfolge einen mit nur einer. Beide Befunde sind wahr; nur der erste nennt dem Planer etwas, das er tun kann.

 Was der Filter kostet Ein Konflikt kostet *n* Solverläufe. Bei 14 Bedingungen sind das 14 — bei 50 000 Restriktionen mit je zehn Sekunden Lösungszeit wären es knapp sechs Tage. Für große Modelle filtert man deshalb nicht einzeln, sondern **gruppenweise**: erst über Bedingungs*blöcke* (alle Kapazitäten, alle Verträge, alle Zeitfenster) laufen, dann den Filter nur noch innerhalb des einen Blocks anwenden, der übrig bleibt. Das ist derselbe Algorithmus auf einer gröberen Ebene und senkt die Zahl der Läufe um Größenordnungen.

Das Verhältnis zur Relaxation. Der Deletion Filter beantwortet die Frage *„was widerspricht sich?“*; die Schlupfvariablen aus Schritt 3 beantworten die Frage *„was tun wir jetzt?“*. Beide gehören in ein Produktionssystem, aber an verschiedene Stellen: der Filter in die Entwicklung und in die Fehlersuche, die Relaxation in den Betrieb (`Infeasibility_Diagnose.py`, Abschnitt 22.3). Ein System, das im Ernstfall den Konflikt *benennt* und trotzdem einen Notfallplan *liefert*, hat beides.

 Die eigentliche Lehre `INFEASIBLE` in Produktion ist ein **Entwurfsfehler**, kein Betriebsfehler. Ein System, das im Ernstfall nur „geht nicht“ sagt, ist wertlos. Bauen Sie Schlupfvariablen von vornherein ein (Abschnitt 22.3).

C2 — UNBOUNDED

Bedeutung. Die Zielfunktion lässt sich unbegrenzt verbessern.

Praktisch immer eine dieser drei Ursachen:

1. **Eine Kapazitätsbedingung fehlt.** Die häufigste Ursache. Prüfen Sie: Gibt es für jede Variable eine Obergrenze — entweder explizit oder implizit über eine Ressource?
2. **Vorzeichenfehler.** Sie minimieren, wo Sie maximieren wollten (oder umgekehrt), und der Zielwert läuft in die falsche Richtung davon.
3. **Freie Variablen ohne Nichtnegativität.** `bounds=[(None, None)]` statt `[(0, None)]`.

```
# Diagnose: kuenstliche Schranke einziehen und sehen, wohin es laeuft
bounds = [(0, 1e6)] * n
res = linprog(...)
gross = [j for j in range(n) if res.x[j] > 1e5]
print(f"Diese Variablen laufen an die Kunstschranke: {gross}")
```

Die so gefundenen Variablen sind die, denen eine echte Beschränkung fehlt.

C3 — Zu langsam

Diagnosereihenfolge

1. Ist es überhaupt das Lösen?

```
import time
t0 = time.perf_counter(); modell_bauen(); t1 = time.perf_counter()
loesen();                                 t2 = time.perf_counter()
print(f"Aufbau {t1-t0:.2f}s | Loesen {t2-t1:.2f}s")
```

Bei CVXPY und bei Schleifen über Szenarien ist oft der **Aufbau** der Engpass, nicht der Solver (siehe Abschnitt 20.6).

2. Big-M zu groß? Der häufigste Grund für explodierende MILP-Laufzeiten. Setzen Sie M auf die kleinste gültige Schranke (Abschnitt 6.5).

3. Symmetrie im Modell? Identische Maschinen, austauschbare Mitarbeitende, gleichwertige Fahrzeuge — der Solver durchsucht alle Vertauschungen. Abhilfe: Ordnungsbedingungen (Muster B20).

4. Falsche Solverfamilie? Zuordnungsprobleme mit MILP, Routing mit CP-SAT von Hand nachgebaut, konvexe Probleme mit `scipy.optimize.minimize` — jeweils Größenordnungen langsamer als das passende Werkzeug (Abschnitt 3.6).

5. Wird vektorisiert?

```
# langsam:                        # schnell:
for s in range(S):                constraints.append(u >= -(R @ w) - gamma)
    constraints.append(u[s] >= -R[s] @ w - gamma)
```

6. Gap akzeptieren.

```
loeser.parameters.relative_gap_limit = 0.02     # CP-SAT
loeser.parameters.max_time_in_seconds = 30.0
```

Bei ±10 % Datenunsicherheit sind die letzten 2 % Optimalität verlorene Zeit.

C4 — Unsinniges Ergebnis

Der Solver hat immer recht — bezogen auf das Modell, das Sie ihm gegeben haben. Wenn das Ergebnis unsinnig ist, beschreibt Ihr Modell nicht das Problem, das Sie meinen.

Checkliste

- ☐ **Einheiten.** Euro gegen Cent? Stunden gegen Minuten? Jahres- gegen Tagesgrößen? (Ein Beispiel dafür finden Sie in Abschnitt 20.6.)
- ☐ **Vorzeichen.** Minimieren Sie, wo Sie maximieren wollten? Ist der Ertrag negativ eingetragen?
- ☐ **Fehlende Bedingung.** Formulieren Sie in Worten, was am Ergebnis falsch ist — meistens ist genau das die vergessene Nebenbedingung.
- ☐ **Variablenbedeutung.** Ist x_{ij} „Person i macht Aufgabe j“ oder umgekehrt? Bei asymmetrischen Matrizen fällt das nicht auf.
- ☐ **Index-Reihenfolge.** Zeilen und Spalten vertauscht? Siehe C11.

Die wirksamste Gegenmaßnahme

```
# Nach JEDEM Lösen: alle Regeln explizit nachprüfen
assert abs(w.sum() - 1) < 1e-6,       "Budget nicht eingehalten"
assert w.min() > -1e-9,               "negatives Gewicht"
assert w.max() <= grenze + 1e-9,      "Positionsgrenze verletzt"
for sektor, idx in sektoren.items():
    assert w[idx].sum() <= limit[sektor] + 1e-9, f"Sektor {sektor} verletzt"
```

Diese Zeilen kosten Sekunden und fangen die Fehlerklasse ab, die der Solver **nicht** melden kann: dass Ihr Modell etwas anderes beschreibt, als Sie glauben.

C5 — Instabile Lösung

Symptom. Ein zusätzlicher Handelstag, und das Portfolio sieht völlig anders aus.

Ursachen und Abhilfen:

Ursache	Prüfung	Abhilfe
Schlecht konditionierte Kovarianzmatrix	`np.linalg.cond(Sigma)`	Ledoit-Wolf-Shrinkage (Kapitel 18)
Zu wenige Beobachtungen	T/N berechnen	Historie verlängern oder Universum verkleinern
Alternativoptima	Zielfunktion bei mehreren Lösungen gleich?	Regularisierung: kleinen L_2-Term addieren
Fehlende Turnover-Dämpfung	Turnover messen	L_1-Strafe (Abschnitt 20.5)

```
eig = np.linalg.eigvalsh(Sigma)
print(f"Kondition {eig.max()/eig.min():.0f} | kleinster EW {eig.min():.2e}")
# Faustregel: Kondition > 1000 ist bedenklich, > 10000 kritisch
```

Derselbe Fehler außerhalb der Finanzwelt

Instabilität ist kein Portfolioproblem, sondern ein Skalierungsproblem — und es trifft jedes Modell, in dem Größen sehr verschiedener Ordnung nebeneinanderstehen. Euro-Beträge (10^7) und Tonnen-Angaben (10^{-3}) in derselben Matrix erzeugen eine Konditionszahl, die aus einer Datenunsicherheit von 0,1 % eine Lösungsänderung von 100 % machen kann. Die Konditionszahl ist genau die Obergrenze dieses Verstärkungsfaktors.

```
print(f"Kondition der Restriktionsmatrix: {np.linalg.cond(A):.1e}")
# Grobe Peilung: Wertebereiche der Koeffizienten anschauen
print(f"kleinster / groesster Betrag: {np.abs(A[A != 0]).min():.1e} "
      f"/ {np.abs(A).max():.1e}")
```

Klaffen die Beträge um mehr als drei, vier Größenordnungen auseinander, rechnen Sie in anderen Einheiten (Tausend Euro statt Euro, Kilogramm statt Tonnen) oder skalieren Sie Zeilen und Spalten automatisch. `Skalierung_Kondition.py` zeigt beides samt Ruiz-Equilibrierung (Abschnitt 2.7).

C6 — Falsche Dualwerte

Symptom. Schattenpreise sind negativ, wo sie positiv sein sollten — oder null bei einer offensichtlich knappen Ressource.

Die drei Prüfungen

1. Vorzeichenkonvention. Haben Sie zur Maximierung negiert?

```
schattenpreise = -res.ineqlin.marginals    # bei linprog nach Negation
```

Genau das ist die Vorzeichenfalle aus Abschnitt 5.7.

2. Komplementärer Schlupf.

```
for s, y in zip(res.slack, schattenpreise):
    assert abs(s * y) < 1e-6, "Komplementaerer Schlupf verletzt!"
```

Eine Ressource mit Reserve **muss** Schattenpreis 0 haben.

3. Numerische Gegenprobe — der zuverlässigste Test.

```
b_plus = b.copy(); b_plus[i] += 1.0
zuwachs = loese(b_plus) - loese(b)
assert abs(zuwachs - schattenpreise[i]) < 1e-6
```

Dieser Test ist **unabhängig von jeder Vorzeichenkonvention**. Nutzen Sie ihn.

4. Wenn alle drei Prüfungen bestehen und der Wert trotzdem seltsam ist: Entartung. Die drei Prüfungen oben setzen voraus, dass es *einen* richtigen Schattenpreis gibt. Bei einem entarteten Optimum — mehr aktive Nebenbedingungen als Variablen — gibt es den nicht. Der Wert ist dann nicht falsch, sondern **mehrdeutig**: zwei korrekte Solver liefern für dasselbe Optimum verschiedene Dualwerte, und die nume-

rische Gegenprobe aus Schritt 3 schlägt fehl, weil der Zuwachs nach links und nach rechts unterschiedlich ausfällt.

```
aktiv = int(np.sum(res.slack < 1e-9))          # wie viele Bedingungen sind straff?
if aktiv > len(res.x):
    print(f"Entartet: {aktiv} aktive Bedingungen bei {len(res.x)} Variablen.")
    print("Einen einzelnen Schattenpreis zu berichten waere hier irrefuehrend.")
```

Die belastbare Antwort ist dann keine Zahl, sondern eine **Spanne** über alle optimalen Dualwerte — `Toleranzen_und_Entartung.py` rechnet sie vor (Abschnitt 5.9).

C7 — Widersprüchliche Solver

Symptom. `linprog` sagt 530, CVXPY sagt 529,8.

Diagnose: 1. **Ist es nur Toleranz?** Unterschiede unter 10^{-6} relativ sind normal. Vergleichen Sie **nie** mit `==`, sondern mit `np.isclose()`. 2. **Ist das Modell wirklich identisch?** Häufigster Fall: In einer Formulierung fehlt eine Bedingung oder das Vorzeichen einer Ungleichung ist gedreht. 3. **Alternativoptima.** Verschiedene Lösungsvektoren bei **gleichem** Zielwert sind kein Widerspruch — das Problem hat mehrere Optima (semidefinite Matrix, parallele Zielfunktion). 4. **Ist eines der Ergebnisse gar nicht optimal?** Status prüfen: `optimal_inaccurate` bedeutet, der Solver hat aufgegeben.

C8 — DCPError

```
cvxpy.error.DCPError: Problem does not follow DCP rules.
```

Bedeutung. CVXPY kann die Konvexität Ihres Ausdrucks nicht beweisen — es lehnt ab, statt ein Ergebnis ohne Garantie zu liefern. **Das ist ein Feature.**

Problematischer Ausdruck	Warum	Lösung
`x * y` (beide Variablen)	bilinear nicht konvex	umformulieren oder MILP/NLP
`a / x`		`cp.inv_pos(x)` bei $x > 0$
`cp.sqrt(cp.quad_form(w, S))`	`cp.sqrt` ist **konkav** und verlangt ein konkaves Argument — `quad_form` ist konvex. Der Ausdruck ist nie DCP, egal wie sauber S ist	Cholesky $S = LL^{\mathsf{T}}$, dann `cp.norm2(L.T @ w)`: die 2-Norm eines **affinen** Ausdrucks (siehe `CVaR_Portfolio.py`)

Problematischer Ausdruck	Warum	Lösung
`cp.quad_form(w, S)` bei numerisch unsauberem S	Eigenwerte knapp unter null durch Rundung — CVXPY erkennt S nicht als PSD	`cp.psd_wrap(S)`, **wenn** S nachweislich PSD ist (z. B. eine Kovarianzmatrix). Behebt nur diese numerische Beanstandung, keine Regelverletzung im Aufbau
Quotient (z. B. Sharpe Ratio)	nicht konvex	Korn-Transformation (Abschnitt 19.4)
`cp.log(x)` maximieren	konkav — das ist **erlaubt**	in `cp.Maximize` verwenden

```
print(problem.is_dcp())                    # False?
for c in problem.constraints:
    print(c.is_dcp(), c)                   # zeigt die Schuldige
print(problem.objective.is_dcp())
```

C9 — Importfehler

```
ImportError: .../highspy/_core...so: undefined symbol: _ZN5Highs13releaseMemoryEv
```

Ursache. `ortools` und `highspy` bringen beide eine eigene HiGHS-Kopie mit; sie lassen sich auf vielen Systemen **nicht im selben Prozess** importieren (siehe Abschnitt 3.5). Der Konflikt entsteht auch **indirekt**: `cvxpy` importiert ein installiertes `highspy` bei der Solver-Erkennung selbst mit — ein Skript, das erst `cvxpy` und dann `ortools` importiert, crasht daher mit derselben Meldung.

Abhilfen (in dieser Reihenfolge): 1. Nur eines von beiden im selben Skript verwenden. 2. Getrennte Prozesse — ein `ProcessPoolExecutor` mit `mp_context="spawn"` und `max_tasks_per_child=1`, siehe `Ein_System_Vier_Ansaetze.py`. 3. Auf `highspy` verzichten: HiGHS ist ohnehin Backend von `scipy.optimize.linprog` und CVXPY. 4. Getrennte virtuelle Umgebungen.

C10 — Verdächtig guter Backtest

Faustregel: Eine Sharpe Ratio über 2 bei einer einfachen Strategie ist fast immer ein Fehler, kein Fund.

Prüfreihenfolge

1. Lookahead-Selbsttest.

```
signal_vorher = berechne_signal(stichtag)
kurse.loc[kurse.index > stichtag] *= 3.0
signal_nachher = berechne_signal(stichtag)
assert np.allclose(signal_vorher, signal_nachher), "LOOKAHEAD-BIAS!"
```

2. Zeitindizes prüfen. `iloc[t]` gegen `iloc[t+1]` — die Gewichte von heute gehören auf die Rendite von morgen.

3. Survivorship. Wurde das Universum nach Kriterien gefiltert, die erst am Ende bekannt waren (`dropna()` über den gesamten Zeitraum!)?

4. Kosten. Sind Gebühren, Spread und Slippage verbucht?

5. Wie viele Varianten haben Sie getestet? Bei 20 Versuchen findet man auch in reinem Rauschen eine „signifikante“ Strategie (Abschnitt 21.6).

6. Nach Teilzeiträumen aufschlüsseln. Stammt die gesamte Überrendite aus einem einzigen Quartal?

C11 — Vertauschte Spalten

Symptom. Ergebnisse sind plausibel, aber falsch beschriftet — oder eine Sektorgrenze greift auf die falschen Titel.

Ursache. `yfinance` und viele andere Datenquellen liefern Spalten **alphabetisch sortiert**, nicht in der Reihenfolge Ihrer Anfrage.

```
# IMMER nach dem Import:
kurse = roh["Close"][tickers].dropna()          # erzwingt die Reihenfolge
assert list(kurse.columns) == tickers, (
    f"Reihenfolge weicht ab!\n erwartet: {tickers}\n erhalten: {list(kurse.columns)}")

# Und Gruppen NIE ueber Positionsindizes definieren:
SEKTOREN = {"Technologie": ["AAPL", "MSFT", "NVDA", "AMZN"]}       # gut
tech_idx = [tickers.index(t) for t in SEKTOREN["Technologie"]]     # bricht bei Tippfehler
tech_indices = [0, 1, 2, 3]                                        # SCHLECHT
```

Die goldene Diagnoseregel

Wenn Sie nicht weiterkommen: **Verkleinern Sie das Problem, bis Sie es von Hand nachrechnen können.** Drei Mitarbeitende, zwei Schichten. Zwei Aktien, drei Tage. Fast jeder Modellierungsfehler wird an einer Instanz sichtbar, deren richtige Antwort Sie kennen — und fast keiner wird an einer Instanz sichtbar, deren Antwort Sie nicht unabhängig prüfen können.

Anhang D: Spickzettel der Solver

Wofür dieser Anhang gedacht ist: Sie wissen, was Sie modellieren wollen, und suchen nur noch, wie die gewählte Bibliothek es schreibt. Jede Seite hat denselben Aufbau — Modell, Variablen, Nebenbedingungen, Lösen, **alle** Statusfälle, Lösung auslesen, Stolpersteine. So lassen sich die Seiten nebeneinanderlegen.

Alle Schnipsel lösen dasselbe Problem — das Produktionsprogramm aus Abschnitt 3.1 mit dem bekannten Optimum **530** und den Schattenpreisen **12** und **1**:

$$\max\ 10x_1 + 15x_2 + 25x_3 \quad \text{u.d.N.} \quad x_1 + x_2 + 2x_3 \le 40,\ \ 2x_1 + 3x_2 + x_3 \le 50,\ \ x \ge 0$$

Damit ist jeder Schnipsel selbstprüfend: Kommt bei Ihnen etwas anderes als 530 heraus, liegt es an der Übertragung, nicht am Modell. (Die Ausnahme ist CP-SAT — ein rein stetiges LP ist dort das falsche Werkzeug; die Seite zeigt stattdessen die CP-SAT-eigenen Bausteine.)

Was dieser Anhang *nicht* ist. Kein Vergleich und keine Empfehlung. Welche Bibliothek für welche Aufgabe taugt, steht in Abschnitt 3.6; denselben Fall in vier Bibliotheken *nebeneinander* zeigt `Ein_System_Vier_Ansaetze.py`, die beiden Modellierungssprachen `Modellierungsschichten.py`. Hier geht es allein ums Nachschlagen.

D1 — SciPy: `linprog` und `milp`

Wofür. Die Einstiegsschicht: keine zusätzliche Installation, HiGHS als Unterbau, ideal für lineare und gemischt-ganzzahlige Probleme in Matrixform. **Wofür nicht:** alles, was sich nicht als Matrix schreiben lässt, und jede nichtlineare Zielfunktion.

Lineares Programm

```
import numpy as np
from scipy.optimize import linprog

# linprog MINIMIERT immer -> zum Maximieren die Zielfunktion negieren
ergebnis = linprog(
    c=[-10.0, -15.0, -25.0],                      # Zielkoeffizienten
    A_ub=[[1, 1, 2], [2, 3, 1]], b_ub=[40, 50],    # A_ub @ x <= b_ub
    A_eq=None, b_eq=None,                          # Gleichungen, falls vorhanden
    bounds=[(0, None)] * 3,                        # je Variable (unten, oben)
    method="highs")

if ergebnis.status == 0:
    print(f"optimal: {-ergebnis.fun:.2f}")         # Vorzeichen zuruecknehmen!
    print(f"x = {np.round(ergebnis.x, 4)}")
    print(f"Schattenpreise: {-ergebnis.ineqlin.marginals}")
elif ergebnis.status == 2:
    print("INFEASIBLE - kein zulaessiger Punkt")
elif ergebnis.status == 3:
    print("UNBOUNDED - Zielfunktion unbeschraenkt")
elif ergebnis.status == 1:
    print("Iterations- oder Zeitlimit erreicht")
else:
    print(f"numerisches Problem (status {ergebnis.status}): {ergebnis.message}")
```

Ganzzahlig: `milp`

```
import numpy as np
from scipy.optimize import milp, LinearConstraint, Bounds

# Ganzzahlig: dasselbe Problem, aber x muss ganzzahlig sein
ergebnis = milp(
    c=[-10.0, -15.0, -25.0],                      # auch milp MINIMIERT
    constraints=LinearConstraint([[1, 1, 2], [2, 3, 1]], -np.inf, [40, 50]),
    integrality=[1, 1, 1],                         # 0 = stetig, 1 = ganzzahlig
    bounds=Bounds(0, np.inf))

if ergebnis.status == 0:
    print(f"optimal: {-ergebnis.fun:.2f}  x = {np.round(ergebnis.x).astype(int)}")
    print(f"MIP-Gap: {ergebnis.mip_gap:.4f}")
elif ergebnis.status == 1:
    print("Zeitlimit - beste gefundene Loesung nutzen, Gap pruefen")
elif ergebnis.status == 2:
    print("INFEASIBLE")
elif ergebnis.status == 3:
    print("UNBOUNDED")
```

```
else:
    print(f"kein Ergebnis: {ergebnis.message}")
```

Die drei häufigsten Stolpersteine

1. **`linprog` minimiert immer.** Zum Maximieren `c` negieren — und beim Ausgeben des Zielwerts das Vorzeichen wieder zurücknehmen. Dieselbe Negation dreht auch die Schattenpreise (Abschnitt 5.7).
2. **`status == 0` prüfen, nicht `res.success` allein.** `success` ist bei `status == 1` (Limit erreicht) `False`, obwohl eine brauchbare Lösung vorliegen kann.
3. **`bounds` gilt je Variable.** `bounds=(0, None)` setzt alle Variablen gleich; `bounds=[(0, None), (0, 10), ...]` einzeln. Wer die Liste vergisst, bekommt stillschweigend überall dieselbe Schranke.

D2 — HiGHS über `highspy`

Wofür. Derselbe Solver wie unter SciPy, aber direkt gesteuert: Optionen, Warm-Starts, inkrementelles Ändern eines bestehenden Modells. **Wofür nicht:** schnelles Hinschreiben — die CSR-Matrixübergabe ist fehleranfällig.

```
import numpy as np
import highspy

h = highspy.Highs()
h.setOptionValue("output_flag", False)            # Solverprotokoll abschalten
h.setOptionValue("time_limit", 60.0)
h.setOptionValue("mip_rel_gap", 0.01)             # 1 % Gap genuegt

# Variablen: Anzahl, Untergrenzen, Obergrenzen
h.addVars(3, np.zeros(3), np.full(3, highspy.kHighsInf))
h.changeObjectiveSense(highspy.ObjSense.kMaximize)
for spalte, wert in enumerate([10.0, 15.0, 25.0]):
    h.changeColCost(spalte, wert)

# Zeilen im CSR-Format: starts[i] = Beginn von Zeile i in indices/values
h.addRows(2, np.full(2, -highspy.kHighsInf), np.array([40.0, 50.0]), 6,
          np.array([0, 3], dtype=np.int32),                # starts
          np.array([0, 1, 2, 0, 1, 2], dtype=np.int32),    # Spaltenindizes
          np.array([1.0, 1.0, 2.0, 2.0, 3.0, 1.0]))        # Koeffizienten
# Ganzzahligkeit: h.changeColsIntegrality(...) mit highspy.HighsVarType.kInteger

h.run()
status = h.getModelStatus()
if status == highspy.HighsModelStatus.kOptimal:
    print(f"optimal: {h.getInfo().objective_function_value:.2f}")
    print(f"x = {np.round(h.getSolution().col_value[:3], 4)}")
    print(f"Schattenpreise: {np.round(h.getSolution().row_dual[:2], 4)}")
elif status == highspy.HighsModelStatus.kInfeasible:
```

```
    print("INFEASIBLE")
elif status == highspy.HighsModelStatus.kUnbounded:
    print("UNBOUNDED")
elif status == highspy.HighsModelStatus.kTimeLimit:
    print(f"Zeitlimit, Gap {h.getInfo().mip_gap:.3f}")
else:
    print("kein Optimum:", h.modelStatusToString(status))
```

Die drei häufigsten Stolpersteine

1. **Nicht zusammen mit `ortools` importieren.** Beide bringen eine eigene HiGHS-Kopie mit; im selben Prozess endet das in `undefined symbol` (Anhang C, C9). Auch `cvxpy` zieht `highspy` bei der Solver-Erkennung mit hinein.
2. **Das CSR-Format stimmt oder es stimmt still nicht.** `starts` hat so viele Einträge wie Zeilen, `indices` und `values` so viele wie Nichtnullen. Ein falscher `starts`-Eintrag erzeugt ein *anderes*, aber lösbares Modell — es fällt nur durch ein falsches Ergebnis auf.
3. **`output_flag` abschalten**, sonst überschwemmt das Solverprotokoll jede Ausgabe.

D3 — OR-Tools: `pywraplp`

Wofür. Bequeme algebraische Schreibweise für LP und MILP mit umschaltbarem Backend (`GLOP`, `SCIP`, `CBC`, `SAT`). **Wofür nicht:** Scheduling und kombinatorische Bedingungen — dafür ist CP-SAT (D4) da.

```
from ortools.linear_solver import pywraplp

# "GLOP" = LP, "SCIP" oder "CBC" = MILP, "SAT" = CP-SAT als MILP-Backend
loeser = pywraplp.Solver.CreateSolver("GLOP")
if loeser is None:
    raise SystemExit("Solver nicht verfuegbar")
loeser.SetTimeLimit(60_000)                        # Millisekunden!

unendlich = loeser.infinity()
x = [loeser.NumVar(0, unendlich, f"x{j}") for j in range(3)]
# ganzzahlig: loeser.IntVar(0, unendlich, "n") | binaer: loeser.BoolVar("b")

loeser.Add(x[0] + x[1] + 2 * x[2] <= 40)
loeser.Add(2 * x[0] + 3 * x[1] + x[2] <= 50)
loeser.Maximize(10 * x[0] + 15 * x[1] + 25 * x[2])

status = loeser.Solve()
if status == pywraplp.Solver.OPTIMAL:
    print(f"optimal: {loeser.Objective().Value():.2f}")
    print(f"x = {[round(v.solution_value(), 4) for v in x]}")
elif status == pywraplp.Solver.FEASIBLE:
    print("zulaessig, nicht bewiesen optimal (Zeitlimit)")
elif status == pywraplp.Solver.INFEASIBLE:
    print("INFEASIBLE")
```

```
elif status == pywraplp.Solver.UNBOUNDED:
    print("UNBOUNDED")
else:
    print("ABNORMAL / NOT_SOLVED - Modell oder Solver pruefen")
```

Die drei häufigsten Stolpersteine

1. **`CreateSolver` liefert `None`**, wenn der Backend-Name unbekannt oder nicht gebaut ist — immer prüfen, statt am `None` später zu scheitern.
2. **`SetTimeLimit` erwartet Millisekunden**, nicht Sekunden. Ein `SetTimeLimit(60)` bricht nach einer sechzigstel Sekunde ab.
3. **`FEASIBLE` ist kein `OPTIMAL`.** Bei Zeitlimit liefert der Solver eine gültige, aber möglicherweise schlechte Lösung — den Gap mitberichten, nicht die Zahl allein.

D4 — CP-SAT: `cp_model`

Wofür. Scheduling, Zuordnung, Reihenfolgen, alles Kombinatorische mit globalen Bedingungen. **Wofür nicht:** stetige Größen — CP-SAT rechnet ausschließlich ganzzahlig. Wer Nachkommastellen braucht, skaliert (Cent statt Euro, Promille statt Anteil).

```
from ortools.sat.python import cp_model

modell = cp_model.CpModel()

# Variablen - CP-SAT rechnet ausschliesslich mit GANZEN Zahlen
x = modell.NewIntVar(0, 100, "x")                  # untere, obere Schranke, Name
y = modell.NewIntVar(0, 100, "y")
b = modell.NewBoolVar("b")                         # 0/1

# Nebenbedingungen
modell.Add(2 * x + 3 * y <= 50)
modell.Add(x >= 5).OnlyEnforceIf(b)                # gilt nur, wenn b wahr ist
modell.AddAllDifferent([x, y])                     # globale Bedingung
modell.AddMaxEquality(z := modell.NewIntVar(0, 100, "z"), [x, y])

modell.Maximize(10 * x + 15 * y - 3 * z)

loeser = cp_model.CpSolver()
loeser.parameters.max_time_in_seconds = 10.0
loeser.parameters.num_workers = 1                  # 1 = reproduzierbar
loeser.parameters.random_seed = 1
status = loeser.Solve(modell)

if status == cp_model.OPTIMAL:
    print(f"optimal: {loeser.ObjectiveValue():.0f}  x={loeser.Value(x)} y={loeser.Value(y)}")
elif status == cp_model.FEASIBLE:
    print(f"zulaessig, nicht bewiesen optimal - Gap-Schranke: {loeser.BestObjectiveBound()}")
```

```
elif status == cp_model.INFEASIBLE:
    print("INFEASIBLE - Bedingungen widersprechen sich")
elif status == cp_model.MODEL_INVALID:
    print("Modellfehler:", modell.Validate())
else:
    print("UNKNOWN - Zeit abgelaufen, ohne eine Loesung zu finden")
print(f"Laufzeit {loeser.WallTime():.3f}s, {loeser.NumBranches()} Verzweigungen")
```

Die drei häufigsten Stolpersteine

1. **Alles ist ganzzahlig.** `0.5 * x` gibt es nicht. Skalieren Sie das ganze Modell auf eine feinere Einheit, statt zu runden.
2. **`UNKNOWN` heißt nicht `INFEASIBLE`.** Es heißt: Die Zeit war zu knapp, um überhaupt etwas zu finden. Die beiden zu verwechseln ist einer der teuersten Fehler in Produktion — das Modell wird für widersprüchlich erklärt, obwohl es lösbar ist.
3. **Ohne `num_workers = 1` ist der Lauf nicht reproduzierbar.** Mehrere Suchstränge finden je nach Zeitverlauf verschiedene, gleich gute Lösungen. Für Tests und für abgedruckte Ausgaben Worker und Seed festnageln.

D5 — CVXPY

Wofür. Konvexe Probleme: quadratische Ziele, Normen, CVaR, alles mit Regularisierungstermen. **Wofür nicht:** große kombinatorische Modelle — der Aufbau der Ausdrücke wird dann selbst zum Engpass.

```
import numpy as np
import cvxpy as cp

x = cp.Variable(3, nonneg=True)                    # nonneg=True statt x >= 0
gewichte = cp.Variable(3)
ganzzahlig = cp.Variable(3, integer=True)          # macht daraus ein MIP

ziel = cp.Maximize(np.array([10.0, 15.0, 25.0]) @ x)
bedingungen = [np.array([[1, 1, 2], [2, 3, 1]]) @ x <= np.array([40.0, 50.0])]

problem = cp.Problem(ziel, bedingungen)
if not problem.is_dcp():                           # VOR dem Loesen pruefen
    print("nicht DCP:", [c for c in bedingungen if not c.is_dcp()])
problem.solve()

if problem.status == cp.OPTIMAL:
    print(f"optimal: {problem.value:.2f}  x = {np.round(x.value, 4)}")
    print(f"Schattenpreis: {np.round(bedingungen[0].dual_value, 4)}")
elif problem.status == cp.OPTIMAL_INACCURATE:
    print("Loesung numerisch unsicher - Skalierung pruefen, anderen Solver testen")
elif problem.status == cp.INFEASIBLE:
    print("INFEASIBLE")
elif problem.status == cp.UNBOUNDED:
```

```
    print("UNBOUNDED")
else:
    print("Solverfehler:", problem.status)
```

Die drei häufigsten Stolpersteine

1. **DCP-Regeln vor dem Lösen prüfen.** `problem.is_dcp()` und dann die einzelnen Bedingungen — das nennt die Schuldige, statt einen `DCPError` ohne Ort zu werfen (Anhang C, C8).
2. **`OPTIMAL_INACCURATE` ist kein Erfolg.** Der Solver hat aufgegeben und meldet das leise. Diesen Fall immer eigens behandeln.
3. **Der Aufbau kann teurer sein als das Lösen.** Schleifen über Szenarien durch Vektorausdrücke ersetzen; `Parameter` statt Neuaufbau, wenn sich nur Zahlen ändern.

D6 — Modellierungssprachen: Pyomo und Linopy

Wofür. Beide trennen *Modell* von *Solver*: dasselbe Modell läuft ohne Änderung unter HiGHS, CBC, Gurobi. Pyomo denkt in **Mengen und Indizes** wie eine mathematische Formulierung; Linopy denkt in **beschrifteten Arrays** und baut Nebenbedingungen als Matrixoperation statt in Python-Schleifen. **Wofür nicht:** ein Modell mit zehn Nebenbedingungen — dort ist der Aufwand höher als der Nutzen.

Pyomo

```
import pyomo.environ as pyo

modell = pyo.ConcreteModel()
modell.J = pyo.RangeSet(0, 2)                        # Indexmenge
modell.x = pyo.Var(modell.J, domain=pyo.NonNegativeReals)
# ganzzahlig: domain=pyo.NonNegativeIntegers | binaer: domain=pyo.Binary

ertrag = {0: 10.0, 1: 15.0, 2: 25.0}
modell.ziel = pyo.Objective(expr=sum(ertrag[j] * modell.x[j] for j in modell.J),
                            sense=pyo.maximize)
modell.montage = pyo.Constraint(
    expr=modell.x[0] + modell.x[1] + 2 * modell.x[2] <= 40)
modell.pruefung = pyo.Constraint(
    expr=2 * modell.x[0] + 3 * modell.x[1] + modell.x[2] <= 50)
modell.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)   # fuer Schattenpreise

ergebnis = pyo.SolverFactory("appsi_highs").solve(modell)
zustand = ergebnis.solver.termination_condition
if zustand == pyo.TerminationCondition.optimal:
    print(f"optimal: {pyo.value(modell.ziel):.2f}")
    print(f"x = {[round(pyo.value(modell.x[j]), 4) for j in modell.J]}")
    print(f"Schattenpreis Montage: {modell.dual[modell.montage]:.4f}")
elif zustand == pyo.TerminationCondition.infeasible:
    print("INFEASIBLE")
elif zustand == pyo.TerminationCondition.unbounded:
```

```
    print("UNBOUNDED")
elif zustand == pyo.TerminationCondition.maxTimeLimit:
    print("Zeitlimit")
else:
    print("kein Optimum:", zustand)
```

Linopy

```
import linopy
import numpy as np
import pandas as pd
import xarray as xr

# Benannte Indizes statt blosser Listen - sonst heissen die Achsen "dim_0"
produkt = pd.Index(["Rahmen", "Gehaeuse", "Deckel"], name="produkt")
ressource = pd.Index(["Montage", "Pruefung"], name="ressource")

modell = linopy.Model()
modell.add_variables(lower=0, coords=[produkt], name="menge")
x = modell.variables["menge"]
# ganzzahlig: integer=True | binaer: binary=True

ertrag = xr.DataArray([10.0, 15.0, 25.0], coords=[produkt])
verbrauch = xr.DataArray([[1.0, 1.0, 2.0], [2.0, 3.0, 1.0]],
                         coords=[ressource, produkt])
vorrat = xr.DataArray([40.0, 50.0], coords=[ressource])

modell.add_objective((ertrag * x).sum(), sense="max")
# EINE Zeile, so viele Nebenbedingungen wie 'ressource' Eintraege hat:
modell.add_constraints((verbrauch * x).sum("produkt") <= vorrat, name="kapazitaet")

modell.solve(solver_name="highs", output_flag=False)

if modell.termination_condition == "optimal":
    print(f"optimal: {modell.objective.value:.2f}")
    print(x.solution.to_series().round(4).to_dict())
    print("Schattenpreise:",
          modell.constraints["kapazitaet"].dual.to_series().round(4).to_dict())
elif modell.termination_condition == "infeasible":
    print("INFEASIBLE")
elif modell.termination_condition == "unbounded":
    print("UNBOUNDED")
else:
    print("kein Optimum:", modell.status, modell.termination_condition)
```

Die drei häufigsten Stolpersteine

1. **Der Solver ist ein eigenes Programm.** `SolverFactory("appsi_highs")` scheitert, wenn HiGHS nicht auffindbar ist — die Fehlermeldung nennt dann das Modell, nicht die fehlende Installation.
2. **Schattenpreise kommen nur auf Anforderung.** Bei Pyomo braucht es `Suffix(direction=IMPORT)` *vor* dem Lösen; wer ihn vergisst, bekommt einen `KeyError` statt einer Warnung.
3. **Bei Linopy die Indizes benennen** (`pd.Index(..., name="produkt")`). Ohne Namen heißen die Achsen `dim_0`, und jede spätere Zuordnung wird zum Ratespiel.

Die gemeinsame Regel

Alle sechs Seiten haben denselben längsten Abschnitt: die **Statusauswertung**. Das ist kein Zufall. Ein Solveraufruf hat nie zwei Ausgänge, sondern mindestens fünf — optimal, zulässig ohne Beweis, unlösbar, unbeschränkt, abgebrochen. Code, der nur `if erfolgreich:` prüft, verwechselt früher oder später „keine Lösung gefunden“ mit „es gibt keine Lösung“, und diese Verwechslung merkt niemand, bis sie teuer wird.

Anhang E: Glossar und Literatur

Glossar

A

- **Almgren-Chriss-Modell** — Standardmodell der optimalen Orderausführung. Es löst den Zielkonflikt zwischen Marktauswirkung bei schnellem Handeln und Volatilitätsrisiko bei langsamem Handeln. → Abschnitt 13.4
- **Alternativoptima** — Mehrere Lösungen mit identischem Zielfunktionswert. Tritt auf, wenn die Zielfunktion parallel zu einer Kante des Polyeders verläuft oder die Matrix nur semidefinit ist. → Abschnitt 11.3

B

- **Bellman-Gleichung** — Rekursionsgleichung der dynamischen Programmierung: Der Wert eines Zustands ist die Summe aus den unmittelbaren Kosten der besten Aktion und dem Wert des Folgezustands. → Abschnitt 13.3
- **Big-M-Methode** — Modellierungstrick, der logische Bedingungen über eine hinreichend große Konstante M an eine Binärvariable koppelt. M sollte **so klein wie möglich** gewählt werden, da große Werte die Relaxation aufweichen und die Laufzeit verschlechtern. → Abschnitt 6.5
- **Binärvariable** — Entscheidungsvariable mit Wertebereich $\{0, 1\}$; sie schaltet Fixkosten, Kapazitäten oder logische Alternativen an und aus. → Abschnitt 1.6
- **Bipartites Matching** — Zuordnung zwischen zwei disjunkten Mengen mit maximalem Nutzen oder minimalen Kosten; klassisch gelöst durch den Ungarischen Algorithmus. → Abschnitt 8.4

- **Branch-and-Bound** — Exaktes Verfahren für ganzzahlige Probleme: Der Suchraum wird rekursiv zerlegt (Branching), Zweige werden verworfen, sobald ihre Relaxation schlechter ist als die beste bekannte Lösung (Bounding). → Abschnitt 6.4
- **Budgeted Uncertainty** — Robustheitsansatz nach Bertsimas/Sim: Höchstens Γ von n Parametern nehmen gleichzeitig ihren ungünstigsten Wert an. → Abschnitt 12.6

C

- **Calmar Ratio** — Jahresrendite geteilt durch den Betrag des maximalen Drawdowns. Ergänzt die Sharpe Ratio um die Verlustperspektive. → Abschnitt 21.5
- **Conditional Value at Risk (CVaR)** — Erwarteter Verlust in den schlechtesten Szenarien jenseits des VaR. Kohärent, subadditiv und nach Rockafellar/Uryasev exakt als lineares Programm formulierbar. → Abschnitt 20.4
- **Conflict Learning** — Technik moderner SAT-Solver, aus jedem Widerspruch eine Sperrklausel abzuleiten, damit dieselbe Sackgasse nicht erneut betreten wird. → Abschnitt 7.3
- **Constraint Programming (CP)** — Paradigma, das nicht über Zielfunktionsgradienten, sondern über logische Verträglichkeit sucht: Constraints schränken Wertebereiche ein, bis eine zulässige Belegung gefunden ist. → Abschnitt 7.3
- **Constraint Propagation** — Kernmechanismus von CP-Solvern: Aus einer Zuweisung werden unmögliche Werte anderer Variablen sofort entfernt, wodurch der Suchbaum schrumpft. → Abschnitt 7.3
- **CP-SAT** — Constraint-Programming-Solver von OR-Tools, der ein Modell in boolesche Erfüllbarkeit übersetzt und mit Propagation, Conflict Learning und paralleler Suche löst. → Abschnitt 7.3
- **CVXPY** — Modellierungssprache für konvexe Optimierung: schreibt das Problem in mathematiknaher Notation, prüft die Konvexität und reicht es an einen Solver weiter. → Abschnitt 3.4

D

- **Dualitätstheorie** — Jedem Optimierungsproblem (Primal) steht ein Dualproblem gegenüber. Der starke Dualitätssatz besagt, dass beide im Optimum denselben Zielfunktionswert besitzen. → Abschnitt 5.6
- **Dynamische Programmierung (DP)** — Lösungsprinzip für mehrstufige Entscheidungen: Das Problem wird in Zustände und Stufen zerlegt und rückwärts gelöst. → Abschnitt 13.3

E

- **Efficient Frontier** — Kurve aller Portfolios, die zu gegebenem Risiko die höchste erwartete Rendite liefern. → Abschnitt 19.4
- **Entscheidungsvariable** — Die vom Solver frei wählbare Größe eines Modells. Ihr Wertebereich (kontinuierlich, ganzzahlig, binär) bestimmt die Problemklasse. → Abschnitt 1.6
- **Error-Maximizer-Effekt** — Eigenschaft der Mean-Variance-Optimierung, Schätzfehler zu verstärken statt auszugleichen: Sie sucht genau die Richtungen, deren Varianz am stärksten unterschätzt wurde. → Abschnitt 18.4

- **EVPI (Expected Value of Perfect Information)** — Differenz zwischen den Kosten unter Unsicherheit und den Kosten bei perfektem Wissen; Obergrenze für den Wert jeder Prognoseverbesserung. → Abschnitt 12.5
- **Explainable OR** — Nachvollziehbarmachung von Solver-Ergebnissen über Schattenpreise, aktive Restriktionen und Kostenzerlegung. → Abschnitt 22.3

F

- **Fat Tails** — Verteilungsränder, die dicker auslaufen als bei der Normalverteilung: Extremereignisse sind deutlich häufiger, als das Normalmodell vorhersagt. → Abschnitt 20.3
- **Fluch der Dimensionalität** — Exponentielles Wachstum des Zustandsraums mit jeder zusätzlichen Zustandsdimension; begrenzt die dynamische Programmierung. → Abschnitt 13.5
- **Fluch des Durchschnitts** (*Flaw of Averages*) — Systematischer Fehler beim Planen mit Erwartungswerten statt mit Verteilungen; folgt aus der Jensenschen Ungleichung. → Abschnitt 12.3
- **Flusserhaltung** — Bedingung, dass an jedem Knoten Abfluss minus Zufluss dem Saldo des Knotens entspricht — das Kirchhoff-Gesetz der Netzwerkoptimierung. → Abschnitt 8.3
- **Fundamentalsatz der linearen Optimierung** — Das Optimum eines lösbaren LP wird stets in mindestens einer Ecke des zulässigen Polyeders angenommen. → Abschnitt 2.4

G

- **Gemischt-ganzzahlige Optimierung (MILP)** — Lineares Modell mit mindestens einer ganzzahligen oder binären Variablen; NP-schwer. → Abschnitt 6.3
- **Global Minimum Variance Portfolio (GMV)** — Portfolio kleinstmöglicher Varianz. Es benötigt **keine Renditeprognose** und ist deshalb robuster gegen Schätzfehler. → Abschnitt 19.4
- **Globale Constraints** — Vorgefertigte Bausteine wie `AllDifferent`, `NoOverlap` oder `Cumulative`, die häufige Strukturen kompakt ausdrücken und spezialisierte Propagatoren besitzen. → Abschnitt 7.4

H

- **HiGHS** — Offener Hochleistungs-Solver für LP, MILP und QP; Backend von `scipy.optimize.linprog` und CVXPY, direkt ansprechbar über `highspy`. → Abschnitt 3.4

I

- **Infeasibility** — Zustand eines Modells ohne zulässige Lösung. In der Praxis über Schlupfvariablen mit hohem Strafgewicht aufzufangen. → Abschnitt 22.3
- **Intervallvariable** — Variable mit Start, Dauer und Ende, die in CP-SAT eine Aktivität beschreibt und Überschneidungsverbote auf Ressourcen ermöglicht. → **Abschnitt 7.4**

J

- **Jensensche Ungleichung** — Für konvexe Funktionen gilt $\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$; die formale Grundlage des Fluchs des Durchschnitts. → Abschnitt 12.3

K

- **Kanonische Standardform** — Einheitliche Matrixschreibweise eines LP als $\min \mathbf{c}^\top \mathbf{x}$ u. d. N. $\mathbf{A}\mathbf{x} \leq \mathbf{b}$, $\mathbf{x} \geq 0$. → Abschnitt 2.3

- **Kardinalitätsbeschränkung** — Obergrenze für die Anzahl gleichzeitig aktiver Entscheidungen; über die Summe der zugehörigen Binärvariablen formuliert. → Abschnitt 6.5
- **Karush-Kuhn-Tucker-Bedingungen (KKT)** — Notwendige Optimalitätsbedingungen restringierter Probleme: Stationarität, primale und duale Zulässigkeit sowie komplementärer Schlupf. Bei konvexen Problemen zugleich hinreichend. → Abschnitt 11.4
- **Kohärentes Risikomaß** — Risikomaß mit den Eigenschaften Monotonie, Subadditivität, positive Homogenität und Translationsinvarianz. Der CVaR erfüllt sie, der VaR nicht. → Abschnitt 20.4
- **Kombinatorische Explosion** — Überproportionales Wachstum des Lösungsraums mit der Problemgröße. → Abschnitt 1.4
- **Komplementärer Schlupf** — Bedingung $s_i \cdot y_i = 0$: Eine Nebenbedingung ist entweder aktiv oder ihr Multiplikator verschwindet. → Abschnitt 5.6
- **Konditionszahl** — Verhältnis von größtem zu kleinstem Eigenwert; misst, wie stark sich kleine Datenänderungen auf die Inverse auswirken. → Abschnitt 18.6
- **Konvexität** — Eigenschaft, bei der jede Verbindungsstrecke zweier Punkte innerhalb der Menge bzw. unterhalb des Funktionsgraphen liegt. Bei konvexen Problemen ist jedes lokale Optimum global. → Abschnitt 2.5
- **Korn-Transformation** — Umformung der nicht-konvexen Sharpe-Ratio-Maximierung in ein konvexes QP durch Homogenisierung. → Abschnitt 19.4
- **Kovarianzmatrix** — Matrix der paarweisen Kovarianzen. Steuert im Markowitz-Modell den Diversifikationseffekt; bei vielen Titeln und wenigen Beobachtungen notorisch schlecht konditioniert. → Abschnitt 18.4

L

- **Lagrange-Multiplikator** — Gewicht, mit dem eine Nebenbedingung in die Lagrange-Funktion eingeht; sein Optimalwert entspricht dem Schattenpreis. → Abschnitt 11.4
- **Ledoit-Wolf-Shrinkage** — Schrumpfung der Stichprobenkovarianz in Richtung eines strukturierten Ziels; der optimale Faktor wird analytisch bestimmt. `scikit-learn` verwendet die skalierte Einheitsmatrix als Ziel. → Abschnitt 18.5
- **Lineare Programmierung (LP)** — Optimierung einer linearen Zielfunktion unter linearen Nebenbedingungen mit kontinuierlichen Variablen. → Abschnitt 5.3
- **Logarithmische Rendite** — Stetige Rendite als Logarithmus des Preisverhältnisses; **über die Zeit** additiv. → Abschnitt 18.3
- **Lookahead-Bias** — Fehler, bei dem Informationen einfließen, die zum Entscheidungszeitpunkt nicht vorlagen. → Abschnitt 21.6

M

- **Market Impact** — Preisverschlechterung, die eine eigene Order durch ihr Volumen auslöst; wächst überproportional mit der Handelsgeschwindigkeit. → Abschnitt 13.4
- **Maximum Drawdown** — Größter prozentualer Rückgang vom bisherigen Höchststand. → Abschnitt 21.5
- **Min-Cost-Flow-Problem (MCNFP)** — Kostengünstigster Transport durch ein Netzwerk unter Kapazitäts- und Flusserhaltungsbedingungen; total unimodular, daher ganzzahlige LP-Lösungen. → Abschnitt 8.3

- **MIP-Gap** — Relativer Abstand zwischen bester gefundener Lösung und bester bekannter Schranke. → Abschnitt 22.3
- **Moderne Portfoliotheorie (Markowitz)** — Rahmenwerk, das ein Portfolio über das Zusammenspiel von erwarteter Rendite und Kovarianz bewertet. → Abschnitt 19.4
- **Monte-Carlo-Simulation** — Erzeugung vieler Zufallsszenarien, um Kennzahlen empirisch zu schätzen. **Bewertet**, optimiert aber nicht. → Abschnitt 12.4
- **MTZ-Formulierung** — Miller-Tucker-Zemlin-Bedingungen, die über Rangvariablen Kurzzyklen ausschließen. → Abschnitt 8.5

N

- **Nebenbedingung (Constraint)** — Gleichung oder Ungleichung, die zulässige von unzulässigen Lösungen trennt. Harte müssen erfüllt sein, weiche werden über Strafterme lediglich bestraft. → Abschnitt 1.6
- **Nichtlineare Programmierung (NLP)** — Problemklasse mit nichtlinearer Ziel- oder Nebenbedingungsfunktion. Ohne Konvexität liefern Verfahren wie SLSQP nur lokale Optima. → Abschnitt 11.5

O

- **Operations Research (OR)** — Disziplin, die reale Entscheidungsprobleme in mathematische Modelle überführt und mit exakten oder heuristischen Algorithmen löst. → Abschnitt 1.3
- **Optimalitätsprinzip** — Grundsatz von Bellman: Jede Teilpolitik einer optimalen Politik ist ihrerseits optimal für den erreichten Zustand. → Abschnitt 13.3

P

- **Pivotisierung** — Basiswechsel im Simplex-Verfahren. → Abschnitt 5.4
- **Polyeder** — Schnittmenge endlich vieler Halbräume; die geometrische Gestalt des zulässigen Bereichs eines LP. → Abschnitt 2.4
- **Positiv (semi-)definit** — Eigenschaft einer symmetrischen Matrix, deren Eigenwerte alle > 0 (definit) bzw. ≥ 0 (semidefinit) sind. Definit □ streng konvex, eindeutige Lösung; semidefinit □ konvex, evtl. mehrere Lösungen. → Abschnitt 11.3
- **Präskriptive Analytik** — Analysestufe, die vorschreibt, welche Handlung unter den gegebenen Bedingungen die beste ist. → Abschnitt 1.3

Q

- **Quadratische Programmierung (QP)** — Optimierung einer quadratischen Zielfunktion unter linearen Nebenbedingungen. → Abschnitt 11.3

R

- **Rebalancing** — Periodische Rückführung des Portfolios auf die Zielgewichte. → **Abschnitt 21.4**
- **Regime-Shift** — Strukturbruch in den Daten, nach dem historisch geschätzte Momente ihre Gültigkeit verlieren. → Abschnitt 22.3
- **Relaxation** — Absichtliches Weglassen einschränkender Forderungen — typischerweise der Ganzzahligkeit —, um eine schnell berechenbare Schranke zu gewinnen. → Abschnitt 6.4

- **Robuste Optimierung** — Auslegung auf den ungünstigsten Fall innerhalb einer Unsicherheitsmenge; verlangt keine Wahrscheinlichkeiten. → Abschnitt 12.6
- **Rockafellar-Uryasev-Theorem** — Ergebnis, das die CVaR-Minimierung in eine konvexe Hilfsfunktion überführt und die Optimierung des Tail-Risikos mit linearen Solvern erlaubt. → Abschnitt 20.4

S

- **Schattenpreis** — Optimalwert einer Dualvariablen: Um wie viel ändert sich der Zielwert, wenn die zugehörige Ressource um eine Einheit erweitert wird. → Abschnitt 5.6
- **Schätzfehler** — Abweichung geschätzter Momente von den wahren Werten. → Abschnitt 18.4
- **Schlupfvariable** — Nichtnegative Hilfsvariable, die eine Ungleichung in eine Gleichung überführt; ihr Wert zeigt die ungenutzte Reserve. → Abschnitt 5.3
- **Schnittebenenverfahren (Cutting Planes)** — Zusätzliche gültige Ungleichungen, die gebrochene LP-Lösungen abschneiden, ohne ganzzahlige Punkte zu verlieren. → Abschnitt 6.4
- **Sensitivitätsanalyse** — Untersuchung, in welchem Bereich sich Koeffizienten und Kapazitäten ändern dürfen, ohne die Struktur der Optimallösung zu verändern. → Abschnitt 5.8
- **Sharpe Ratio** — Überrendite über den risikofreien Zins je Einheit Volatilität. → Abschnitt 19.4
- **Simplex-Algorithmus** — Verfahren von Dantzig, das von Ecke zu Ecke wandert und dabei den Zielfunktionswert monoton verbessert. → Abschnitt 5.4
- **SLSQP** — *Sequential Least Squares Programming*; gradientenbasiertes Verfahren in `scipy.optimize.minimize` für nichtlineare Probleme. → Abschnitt 11.5
- **Subadditivität** — Eigenschaft $\rho(A + B) \leq \rho(A) + \rho(B)$: Diversifikation darf das Risiko nicht erhöhen. Der VaR verletzt sie. → Abschnitt 20.4
- **Survivorship-Bias** — Verzerrung durch Auswahl nur der heute noch existierenden Titel. → Abschnitt 21.6
- **Symmetriebrechung** — Zusätzliche Ordnungsbedingungen, die verhindern, dass der Solver gleichwertige Vertauschungen mehrfach durchsucht. → Anhang B, Muster B20

T

- **Totale Unimodularität** — Eigenschaft einer Matrix, bei der jede quadratische Teilmatrix die Determinante 0, $+1$ oder -1 hat. Folge: Alle Ecken des Polyeders sind ganzzahlig — Ganzzahligkeit muss nicht gefordert werden. → Abschnitt 8.4
- **Transaktionskosten** — Beim Umschichten anfallende Kosten aus Gebühren, Spread und Market Impact; über eine L_1-Strafe modellierbar. → Abschnitt 20.5
- **Turnover** — Summe der absoluten Gewichtsänderungen einer Umschichtung. → Abschnitt 20.5

U

- **Unsicherheitsmenge** — Vorab definierter Bereich möglicher Parameterwerte, gegen dessen ungünstigstes Element eine robuste Lösung abgesichert wird. → Abschnitt 12.6

V

- **Value at Risk (VaR)** — Verlustschwelle, die mit vorgegebener Wahrscheinlichkeit nicht überschritten wird. Sagt nichts über die Verlusthöhe dahinter und ist nicht subadditiv. → Abschnitt 20.4

- **Vehicle Routing Problem (VRP)** — Verallgemeinerung des TSP auf mehrere Fahrzeuge mit Depot, Kapazitäten und — in der Variante VRPTW — Zeitfenstern. → Abschnitt 8.5
- **Volatilität** — Standardabweichung der Renditen, üblicherweise auf ein Jahr skaliert. → **Abschnitt 18.3**

W

- **Walk-Forward-Backtest** — Rollierende Auswertung, bei der Parameter stets nur auf Vergangenheitsdaten geschätzt und auf dem folgenden Zeitraum getestet werden. → **Abschnitt 21.5**
- **Wurzel-Zeit-Regel** — Skalierung der Volatilität mit $\sqrt{T}$. Gilt streng nur für Standardabweichungen unabhängiger Größen ohne Drift — für VaR/CVaR nur als grobe Näherung. → Abschnitt 20.6

Z

- **Zielfunktion** — Der zu minimierende oder maximierende Ausdruck, der die Entscheidungsvariablen zu einer einzigen Bewertungszahl verdichtet. → Abschnitt 1.6
- **Zulässiger Bereich (Feasible Region)** — Menge aller Punkte, die sämtliche Nebenbedingungen gleichzeitig erfüllen. → Abschnitt 2.4

Literaturverzeichnis

Mathematische Optimierung — Grundlagen

- **Bertsimas, D. & Tsitsiklis, J. N. (1997):** *Introduction to Linear Optimization.* Athena Scientific, Belmont. *Standardwerk zur geometrischen und algebraischen Theorie des Simplex-Verfahrens, Dualität, Sensitivitätsanalyse und Netzwerkflüssen.*
- **Dantzig, G. B. (1963):** *Linear Programming and Extensions.* Princeton University Press. *Das historische Originalwerk des Erfinders des Simplex-Algorithmus.*
- **Wolsey, L. A. (2020):** *Integer Programming* (2. Aufl.). Wiley, Hoboken. *Branch-and-Bound, Branch-and-Cut, Polyedertheorie, Formulierungsstärke.*
- **Hillier, F. S. & Lieberman, G. J. (2021):** *Introduction to Operations Research* (11. Aufl.). McGraw-Hill. *Breite Einführung inkl. Warteschlangen, DP und Entscheidungsmodellen.*

Konvexe, nichtlineare und robuste Optimierung

- **Boyd, S. & Vandenberghe, L. (2004):** *Convex Optimization.* Cambridge University Press. *Referenzwerk für konvexe Mengen, KKT, QP und SDP; theoretische Basis von CVXPY. Frei verfügbar unter stanford.edu/~boyd/cvxbook/.*
- **Ben-Tal, A., El Ghaoui, L. & Nemirovski, A. (2009):** *Robust Optimization.* Princeton University Press.
- **Bertsimas, D. & Sim, M. (2004):** *The Price of Robustness.* In: *Operations Research* 52(1), S. 35–53. *Quelle des Budgeted-Uncertainty-Ansatzes.*
- **Bellman, R. (1957):** *Dynamic Programming.* Princeton University Press.

Constraint Programming und Scheduling

- **Rossi, F., van Beek, P. & Walsh, T. (2006):** *Handbook of Constraint Programming.* Elsevier.
- **Perron, L. & Furnon, V.:** *OR-Tools CP-SAT Solver Documentation.* developers.google.com/optimization/cp

Quantitative Finanzmathematik

- **Markowitz, H. (1952):** *Portfolio Selection.* In: *The Journal of Finance* 7(1), S. 77–91.
- **Rockafellar, R. T. & Uryasev, S. (2000):** *Optimization of Conditional Value-at-Risk.* In: *Journal of Risk* 2(3), S. 21–42.
- **Ledoit, O. & Wolf, M. (2003):** *Improved estimation of the covariance matrix of stock returns with an application to portfolio selection.* In: *Journal of Empirical Finance* 10(5), S. 603–621. *Quelle des Konstant-Korrelations-Ziels.*
- **Ledoit, O. & Wolf, M. (2004):** *A well-conditioned estimator for large-dimensional covariance matrices.* In: *Journal of Multivariate Analysis* 88(2), S. 365–411. *Quelle des von* `scikit-learn` *verwendeten Ziels (skalierte Einheitsmatrix).*
- **Almgren, R. & Chriss, N. (2000):** *Optimal Execution of Portfolio Transactions.* In: *Journal of Risk* 3(2), S. 5–39.
- **DeMiguel, V., Garlappi, L. & Uppal, R. (2009):** *Optimal Versus Naive Diversification.* In: *Review of Financial Studies* 22(5), S. 1915–1953. *Die Studie, die 1/N als ernstzunehmenden Vergleichsmaßstab etablierte.*
- **López de Prado, M. (2018):** *Advances in Financial Machine Learning.* Wiley. *Backtest-Overfitting, Deflated Sharpe Ratio, Denoising.*
- **Cornuéjols, G., Peña, J. & Tütüncü, R. (2018):** *Optimization Methods in Finance* (2. Aufl.). Cambridge University Press.

Software und Dokumentation

- **Google OR-Tools** — developers.google.com/optimization · Quellcode: github.com/google/or-tools
- **Diamond, S. & Boyd, S. (2016):** *CVXPY: A Python-embedded modeling language for convex optimization.* In: *Journal of Machine Learning Research* 17(83), S. 1–5. · www.cvxpy.org
- **Huangfu, Q. & Hall, J. A. J. (2018):** *Parallelizing the dual revised simplex method.* In: *Mathematical Programming Computation* 10(1), S. 119–142. · highs.dev
- **SciPy** — docs.scipy.org/doc/scipy/reference/optimize.html
- **scikit-learn, Covariance Estimation** — scikit-learn.org/stable/modules/covariance.html

Verbände und Normen

- **INFORMS** — www.informs.org
- **GOR (Gesellschaft für Operations Research e. V.)** — deutschsprachiger Fachverband.
- **Basel Committee on Banking Supervision:** *Minimum capital requirements for market risk* (Basel III, FRTB) — Grundlage der Umstellung von VaR auf Expected Shortfall.
- **Europäische Union:** MiFID II / ESMA-Leitlinien zum algorithmischen Handel.

Stichwortverzeichnis

A

Absolutbetrag (Modellierungsmuster), 700
Adjusted Close, 457
Aktivierungsschalter (Modellierungsmuster), 698
Almgren-Chriss-Modell, 379, 731
Alternativoptima, 731
Approximate Dynamic Programming, 584

B

Bedingte Kopplung (Konjunktion), 700
Bellman, Richard, 375
Bellman-Gleichung, 386, 731
Benders-Zerlegung, 584
Big-M-Methode, 174, 731
Binärvariable, 731
Bipartites Matching, 251, 731
Black-Box-Effekt, 544
Branch-and-Bound, 172, 732
    Bounding und Pruning, 173
    Branching, 173
    Incumbent, 173
Branch-and-Cut, 88, 172
Brute Force, 32
Budgeted Uncertainty, 361, 732
Budgetlimit, 702

C

Calmar Ratio, 732
CDCL, 88, 206
    Conflict Learning, 243, 732
Chance Constraint, 361
Column Generation, 297, 584
Conditional Value at Risk (CVaR), 502, 732
Constraint Attribution, 548
Constraint Programming (CP), 206, 732
Constraint Propagation, 206, 732
Constraint-Trace, 544
CP-SAT, 732
CP-SAT-Solver, 206
CSR-Format, 99
CVRPTW, 254
CVXPY, 88, 328, 732

D

Dantzig, George, 37, 139
Data Snooping, 529
DCPError, 720
Deletion Filter, 708
Diskrete Rendite, 457
Diversifikation, 477
Dual Simplex, 88
Duales Problem, 146
Dualitätstheorie, 732
Dynamische Programmierung, 375, 732
    Approximate Dynamic Programming, 386

E

Ecke (Extrempunkt), 63
Effizienzgrenze, 480, 732
Entartung, 155
    und Dualwerte, 719
Entscheidungsvariable, 38, 732
Entscheidungsvektor, 59
Entweder-Oder-Bedingung, 699
Error-Maximizer-Effekt, 456, 732
EVPI, 358, 733
Exklusiv-Oder, 699
Expected Shortfall, 502
Explainable OR, 733

F

Fallback-Strategie, 558
Falsche Dualwerte (Fehlerbild), 719
Fat Tails, 499, 733
Fluch der Dimensionalität, 386, 733
Fluch des Durchschnitts, 347
Flusserhaltung, 246, 705, 733
Fundamentalsatz der linearen Optimierung, 65, 733

G

Genetischer Algorithmus, 297
Gestaffelte Preise (stückweise linear), 700
Gini-Koeffizient, 631

Gleitendes Fenster, 703
Globale Constraints, 209, 733
  AddAllDifferent, 243
  AddCumulative, 243
  AddNoOverlap, 243
Globales Minimum, 66
GMV (Global Minimum Variance), 496, 733
Goldene Regel des Backtestens, 536
Gradient, 320
Gurobi, 172

H
Herfindahl-Index, 633
Hierarchische Relaxation, 541
HiGHS, 172, 733

I
IIS (Irreducible Infeasible Subset), 708
Implikation (Modellierungsmuster), 699
Importfehler (ortools und highspy), 721
Infeasibility, 39, 707, 733
Instabile Lösung (Fehlerbild), 718
Interior-Point-Verfahren, 88
Intervallvariable, 206, 733

J
Jensensche Ungleichung, 350, 733
Job-Shop-Scheduling, 218

K
Kanonische Standardform, 733
Kapazitätsvektor, 60
Kardinalitätsbeschränkung, 177, 699, 734
KKT-Bedingungen, 323, 734
Kohärentes Risikomaß, 734
Kombinatorische Explosion, 56, 734
Komplementärer Schlupf, 150, 324, 734
Konditionszahl, 74, 719, 734
Konische Optimierung (SOCP, SDP), 361, 584
Konvexität, 65, 320, 734
  Funktion, 66
  Menge, 65
  streng konvex, 320
Korn-Transformation, 476, 734
Kostenzerlegung, 544
Kovarianzmatrix, 320, 734
Kumulative Ressource, 703

L
Lagrange-Funktion, 323
Lagrange-Multiplikator, 734
Large Neighborhood Search, 288
Laufzeitexplosion (Fehlerbild), 717
Ledoit-Wolf-Shrinkage, 463, 734
Lexikografische Optimierung, 402, 704
Lineare Programmierung (LP), 734
Linopy, 101
linprog, 151
Logarithmische Rendite, 457, 734
Lokales Minimum, 66
Lookahead-Bias, 516, 734
LP-Relaxation, 173

M
Makespan, 218
Marchenko-Pastur-Gesetz, 459
Markowitz-Modell (Mean-Variance), 476, 735
Marktauswirkung, 379, 734
Matrixform, 86
Matrix-Vektor-Produkt, 60
Maximin-Fairness, 701
Maximum Drawdown, 734
Mensch in der Schleife, 558
Metaheuristik, 270
MILP, 168, 733
Min-Cost-Flow, 247, 734
Mindestabnahmemenge, 701
MINLP, 340
MIP-Gap, 177, 185, 735
Monte-Carlo-Simulation, 351, 735
MTZ-Formulierung, 254, 705, 735
Multi-Objective Optimization, 584

N
Nebenbedingung, 39, 735
  hart, 39
  weich, 39
Newsvendor-Problem, 347
Nichtlineare Programmierung (NLP), 735

Nichtüberlappung, 702
NP-schwer, 172

O
Operations Research, 32, 735
Optimalitätsprinzip, 374, 735
OR-Tools, 31
Overfitting, 529

P
Parameter (Modell), 38
Pareto-Front, 392
Pivotisierung, 735
Polars, 106
Polyeder, 63, 735
Positiv (semi-)definit, 735
Präskriptive Analytik, 32, 735
Predict-then-Optimize, 408
Primales Problem, 146
Pyomo, 100

Q
Q-Learning, 387
Quadratische Programmierung (QP), 320, 735

R
Rebalancing, 735
Regime-Shift, 735
Reinforcement Learning, 386
Relaxation, 735
Robuste Optimierung, 358, 736
    Worst-Case-Abzug, 705
Rockafellar-Uryasev-Theorem, 504, 736
Routing-Bibliothek, 267
Rucksackproblem, 175, 702
Rückwärtsinduktion, 374
Ruiz-Equilibrierung, 76
Rüstzeit, 703

S
SAT (Boolean Satisfiability), 206
Satz von Birkhoff und von Neumann, 251
Schattenpreis, 137, 138, 736
    Vorzeichenfalle, 151
Schätzfehler, 736
Schlupfvariable, 138, 736
    gegen Unlösbarkeit, 704
Schnittebenen, 173, 736
    Gomory-Cuts, 173
SCIP, 172
Semikontinuierliche Variable, 174, 699
Sensitivitätsanalyse, 151, 736
Shannon-Entropie, 328
Sharpe-Ratio, 480, 736
Simplex-Algorithmus, 37, 139, 736
    Dantzig-Regel, 139
    Pivotspalte, 139
    Pivotzeile, 139
Simulated Annealing, 271
Skalarisierung, 393
Skalarprodukt, 59
SLSQP, 328, 736
Snapshot-Prinzip, 558
Starker Dualitätssatz, 147
Stichproben-Kovarianzmatrix, 459
Stochastische Programmierung, 354
Strafkosten, 39
Subadditivität, 503, 736
Subtour, 254
    Eliminierung, 705
Survivorship-Bias, 529, 736
Symmetriebrechung, 704, 736

T
Tabu-Suche, 297
Tangentialportfolio, 480
Technologiematrix, 59
Totale Unimodularität, 246, 736
Transaktionskosten, 736
Trickle Flow, 650
TSP (Traveling Salesperson Problem), 254
Turnover (Umschlag), 505, 736
TWAP, 385

U
Umrüstkosten (Modellierungsmuster), 703
Unbounded (Fehlerbild), 716
Ungarischer Algorithmus, 267
Unsicherheitsmenge, 736

Unsinniges Ergebnis (Fehlerbild), 717

V

Value at Risk (VaR), 502, 736
Vehicle Routing Problem (VRP), 737
Verdächtig guter Backtest (Fehlerbild), 721
Verhältnis-Bedingung, 701
Vertauschte Spalten (Fehlerbild), 722
Volatilität, 737
Vorrangbeziehung, 702

W

Walk-Forward-Backtest, 516, 737
Warm-Start, 191
Weiche Grenze mit Strafkosten, 700
Widersprüchliche Solver (Fehlerbild), 720
Wurzel-Zeit-Regel, 737

Z

Zielfunktion, 39, 737
Zufallsmatrizentheorie, 459
Zulässiger Bereich, 63, 737
Zuordnungsproblem (1:1), 705
Zustandslosigkeit (OR-Plattform), 558
Zustandsraum, 386
Zwei-Phasen-Simplex-Methode, 647
Zweischichtige Architektur, 88