Lineare Algebra und Operations Research

Klaus Rheinberger, FH Vorarlberg

10. Mai 2024

Inhaltsverzeichnis

1. Vorwort

1.1. Eckdaten

- Studiengang: Umwelt und Technik der FH Vorarlberg
- Semester: 2, 2. Semesterhälfte
- ECTS-Punkte: 4
- Semesterwochenstunden: 2 Vorlesung + 1 Übungen
- Unterrichtssprache: Deutsch

1.2. Lernergebnisse

Die Studierenden können:

- Zentrale Begriffe der linearen Algebra wie z. B. Vektoren, Matrizen, Vektorraum, lineare Abbildung erklären.
- Algorithmen und Verfahren der Vektor- und Matrizenrechnung geometrisch interpretieren und anwenden. Sie sind in der Lage, das Lösungsverhalten von linearen Gleichungssystemen zu bestimmen und sie ggf. von Hand und mit dem Computer zu lösen.
- Die Möglichkeiten und Grenzen von Linearer- (LP) und Mixed-Integer Linearer (MILP) Programmierung zusammenfassen.
- Zweidimensionale lineare Optimierungsprobleme geometrisch beschreiben und analysieren.
- Ausgehend von der Beschreibung eines Optimierungsproblems dieses in Python modellieren (implementieren) und einen geeigneten Solver (GLPK) zur Lösung des Problems benutzen.
- Die Lösung von LPs und MILPs interpretieren und unterschiedliche Lösungen vergleichen.

1.3. Lehrinhalte

Die Lehrveranstaltung behandelt folgende grundlegende Konzepte und Methoden:

- Vektoren, Matrizen, lineare Gleichungssysteme
- Vektorräume, lineare Abbildungen, Skalarprodukte, Normen
- Determinanten, Eigenwerte und Eigenvektoren
- Klassifizierung von Optimierungsproblemen
- Modellierung und Formulierung von Linearen- und Mixed-Integer Linearen Programmen
- Lösen von Optimierungsproblemen in Python mit Hilfe des Solvers GLPK
- Interpretation der Lösung, evtl. Sensitivitätsanalyse

1.4. Literatur

Einige der folgenden Bücher [BV18; HL20; MG10; Dom+15a; Dom+15b; SC17; Wil13; Pap18; Pap15; LLM21; Str23; LL20; Kle13] sind in der Bibliothek der FH Vorarlberg in Printversion und oder als pdf-Download erhältlich.

- Papula, Lothar (2018): Mathematik für Ingenieure und Naturwissenschaftler Band 1: Ein Lehr- und Arbeitsbuch für das Grundstudium. 15. Auflage, Springer. FHV-Download
- Papula, Lothar (2015): Mathematik für Ingenieure und Naturwissenschaftler Band 2: Ein Lehr- und Arbeitsbuch für das Grundstudium. 14. Auflage, Springer. FHV-Download
- Lay, David; Lay, Steven; McDonald, Judi (2021): Linear Algebra and Its Applications. 6th edition, Pearson Education Limited.
- Strang, Gilbert (2023): Introduction to Linear Algebra. 6th edition. Cambridge University Press.
- Boyd, Stephen; Vandenberghe, Lieven (2018): Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares. 1. Auflage, Cambridge University Press. Online Version
- Hillier, Frederick; Lieberman, Gerald (2020): Introduction to Operations Research. 11th edition. New York, NY: McGraw-Hill Education Ltd.
- Matousek, Jirí; Gärtner, Bernd (2010): Understanding and Using Linear Programming. Springer.
- Domschke, Wolfgang et al. (2015): Einführung in Operations Research. 9. Auflage, Springer Gabler. FHV-Download
- Domschke, Wolfgang et al. (2015): Übungen und Fallbeispiele zum Operations Research. 8. Auflage, Springer Gabler. FHV-Download
- Sioshansi, Ramteen; Conejo, Antonio J. (2017): Optimization in Engineering: Models and Algorithms. 1st edition, Springer.
- Williams, H. Paul (2013): Model Building in Mathematical Programming. 5th edition, Wiley.
- Kallrath, Josef (2013): Gemischt-ganzzahlige Optimierung: Modellierung in der Praxis. 2. Auflage, Springer.
- Python.org
- Linge, Svein; Langtangen, Hans Petter (2020): Programming for Computations - Python: A Gentle Introduction to Numerical Simulations with Python 3.6. 2nd edition, Springer. FHV-Download
- Klein, Philip N. (2013): Coding the Matrix: Linear Algebra through Applications to Computer Science. 1. Auflage, Newtonian Press.

1.5. Notation

Als Dezimaltrennzeichen wird der englische Punkt statt dem deutschen Komma verwendet, da dies auch in der verwendeten Programmiersprache der Fall ist und zu keinen Verwechslungen mit Kommas bei der Angabe von Vektoren und Intervallen führt.

1.6. Impressum

Siehe das Impressum der FH Vorarlberg

Teil I.

Lineare Algebra

2. Vektorrechnung

2.1. Methoden

Motivation

Vektoren werden sehr vielseitig eingesetzt, um Objekte zu modellieren, die mehr als eine Zahl für ihre Beschreibung benötigen. Beispiele: Punkte, Orts- und Verbindungsvektoren, Ereignisse in der Raumzeit, Zeiger, Geschwindigkeiten, Beschleunigungen, (Dreh-)Impulse, Kräfte, Signale, Zeitreihen, Preise, cash flows, (Einkaufs-)Listen, Zustände eines Systems. Vektoren finden Anwendungen in der analytischen Geometrie, der Physik, der Elektrotechnik, der Statik, der Robotik, der Statistik und Data Science, der Wirtschaft und in vielen anderen Gebieten.
Das Rechnen mit Vektoren ist vergleichsweise einfach und daher leicht am Computer zu implementieren. Zudem lassen sich Vektoren und die davon abgeleiteten Objekte und Methoden anschaulich in Ebene und Raum darstellen.

2.1.1. Vektorraum $\mathbb{R}^n$

Ein n-**Vektor** ist eine geordnete Liste von n Zahlen, die **Komponenten** (auch Elemente oder Koordinaten) genannt werden. Ein Vektor a wird typischer Weise als **Spaltenvektor**

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

oder als **Zeilenvektor**

$$a = (a_1, a_2, \dots, a_n)$$

geschrieben. Zwei Vektoren sind gleich, wenn sie in allen Komponenten übereinstimmen. Ein **Nullvektor** ist ein Vektor, dessen Komponenten alle Null sind. Man schreibt verkürzt nur eine Null: $(0, 0, \dots, 0) = 0$.

Es sind auch andere Notationen für Vektoren üblich: mit eckigen statt runden Klammern, in fetter Schrift und mit einem Strichpunkt oder einem vertikalen Strich statt einem Komma zum Trennen der Komponenten. Oft wird auch ein Pfeil über den Vektor gezeichnet.

Achtung bei der Indexierung!

Eine Liste von z. B. 300 10er-Vektoren wird üblicherweise mit $a_1, a_2, \dots, a_{300}$ bezeichnet. Dann

ist der Vektor $a_k \in \mathbb{R}^{10}$ und man schreibt die ihn it seinen Komponenten als

$$a_k = \begin{pmatrix} a_{k,1} \\ a_{k,2} \\ \vdots \\ a_{k,10} \end{pmatrix}.$$

Der $\mathbb{R}^n$ ist die Menge aller n-Vektor mit reellen Komponenten. Er hat die **Dimension** n, da jeder Vektor aus $\mathbb{R}^n$ eindeutig durch n Zahlen, z. B. seine n Komponenten, beschreibbar ist. Es gilt $\mathbb{R}^1 = \mathbb{R}$, der Zahlenstrahl. Die Elemente des $\mathbb{R}^2$ kann man z. B. als Punkte, Ortsvektoren, Verbindungsvektoren oder freie Vektoren der Ebene identifizieren:

- Der Nullvektor ist der (Koordinaten-)Ursprung der Ebene.
- Den Vektor $(3, -2)$ als **Punkt** zu interpretieren, bedeutet, bei den Koordinaten 3 und -2 einen Punkt in das Koordinatensystem einzuzeichnen.
- **Ortsvektoren** zeichnet man als Pfeile, die vom Ursprung ausgehen und zum zugehörigen Punkt zeigen.
- Die zwei Punkte $P = (4, 3)$ und $Q = (5, -2)$ werden durch den **Verbindungsvektor** $\overrightarrow{PQ} = \begin{pmatrix} 5-4 \\ -2-3 \end{pmatrix} = \begin{pmatrix} 1 \\ -5 \end{pmatrix}$ verbunden, den man als Pfeil von P nach Q zeichnet.
- Wenn man den Vektor $\begin{pmatrix} 1 \\ -5 \end{pmatrix}$ nicht bei P ``anbindet'', sondern ihn in der Ebene an jeden Angriffspunkt ``anbinden'' kann, dann interpretiert ihn als **freien Vektor**.

Analog kann man den $\mathbb{R}^3$ mit dem Raum identifizieren. Die Vektoren der Ebene schreibt man gerne als (x, y) oder (a_x, a_y) anstatt mit Indizes (a_1, a_2), und jene des Raums gerne als (x, y, z) oder (a_x, a_y, a_z) anstatt mit Indizes (a_1, a_2, a_3). Wir werden uns in diesem Kapitel bei den Anwendungen meist auf die Dimensionen $n = 2$ (Ebene) und $n = 3$ (Raum) einschränken, die Theorie (mit Ausnahme des Kreuzprodukts) aber weiterhin flexibel für alle $n \in \mathbb{N}$ präsentieren.

Die folgenden zwei Rechenoperationen machen aus der Menge $\mathbb{R}^n$ einen sogenannten **Vektorraum**:

- **Skalarmultiplikation**: Die elementweise Multiplikation eines Skalars (einer Zahl) $\alpha \in \mathbb{R}$ mit allen Vektorkomponenten liefert wieder einen Vektor:

$$\alpha \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \alpha a_1 \\ \alpha a_2 \\ \vdots \\ \alpha a_n \end{pmatrix}$$

- **Addition**: Die elementweise Addition zweier Vektoren liefert wieder einen Vektor:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{pmatrix}$$

Die **Subtraktion** $a - b$ zweier Vektoren ist erklärt als $a + (-1)b$ und entspricht der elementweise Subtraktion der Komponenten.

Die Interpretation dieser Rechenoperationen hängt davon ab, was die Vektoren modellieren! Interpretiert man Vektoren als Pfeile in der Ebene oder im Raum, dann kann man die Skalarmultiplikation als ``Strecken/Stauchen/Umdrehen'', die Addition als ``Pfeil-Aneinanderreihung'' und die Subtraktion mit der ``Spitze minus Schaft''-Regel darstellen, siehe z. B. Wikipedia.

Es gelten die üblichen, intuitiven Rechenregeln: $\alpha(a+b) = \alpha a + \alpha b$, $(\alpha+\beta)a = \alpha a + \beta a$, $(\alpha\beta)a = \alpha(\beta a) = \beta(\alpha a)$ für $\alpha, \beta \in \mathbb{R}$ und $a, b \in \mathbb{R}^n$.

Zwei nicht-Null-Vektoren $a, b \in \mathbb{R}^n$ heißen **parallel**, wenn $b = \alpha a$ und $\alpha > 0$, und **antiparallel**, wenn $\alpha < 0$. In beiden Fällen heißen die Vektoren **kollinear**.

Ein Ausdruck der Form $\alpha a + \beta b$ mit $\alpha, \beta \in \mathbb{R}$ und $a, b \in \mathbb{R}^n$ oder allgemeiner $\alpha_1 a_1 + \alpha_2 a_2 + ... + \alpha_k a_k$ mit $\alpha_i \in \mathbb{R}$ und $a_i \in \mathbb{R}^n$ heißt **Linearkombination** der zwei bzw. k Vektoren. Eine Menge von k Vektoren $a_1, a_2, ..., a_k$ heißt **linear unabhängig**, wenn die Gleichung $\alpha_1 a_1 + \alpha_2 a_2 + ... + \alpha_k a_k = 0$ nur die triviale Lösung $\alpha_i = 0 \; \forall i = 1, ..., k$ hat. Andernfalls heißen die k Vektoren **linear abhängig**, und mindestens einer der vorkommenden Vektoren lässt sich als Linearkombination der restlichen ausdrücken. Denn, wenn z. B. $\alpha_1 \neq 0$, dann kann man aus $\alpha_1 a_1 + \alpha_2 a_2 + ... + \alpha_k a_k = 0$ folgendes schließen:

$$\begin{aligned} \alpha_1 a_1 &= -\alpha_2 a_2 - ... - \alpha_k a_k \\ a_1 &= \frac{1}{\alpha_1}(-\alpha_2 a_2 - ... - \alpha_k a_k) \\ a_1 &= -\frac{\alpha_2}{\alpha_1} a_2 - ... - \frac{\alpha_k}{\alpha_1} a_k. \end{aligned}$$

Zum Beispiel sind zwei kollineare Vektoren linear abhängig und zwei nicht kollineare Vektoren linear unabhängig.

2.1.2. Inneres Produkt

Eine weitere Rechenoperation stellt das **innere Produkt** (auch Skalarprodukt genannt, engl. oft *inner product* oder *dot product*) dar, das für zwei Vektoren $a, b \in \mathbb{R}^n$ durch Multiplizieren der entsprechenden Komponenten und anschließendes Aufsummieren eine Zahl, also ein Skalar, liefert. Wir schreiben das innere Produkt von a mit b als

$$a \cdot b = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = a_1 b_1 + a_2 b_2 + ... a_n b_n.$$

Auch hier sind andere Notationen verbreitet. Es gilt: $a \cdot b = b \cdot a$, $a \cdot (b + c) = a \cdot b + a \cdot c$, $\alpha(a \cdot b) = (\alpha a) \cdot b = a \cdot (\alpha b)$.

Die **Länge** (Norm, Betrag) $\|a\|$ eines Vektors a wird über den verallgemeinerten Satz von Pythagoras als

$$\|a\| := \sqrt{a_1^2 + a_2^2 + ... + a_n^2}$$

definiert und kann mit Hilfe des inneren Produktes als $\|a\| = \sqrt{a \cdot a}$ geschrieben werden. Ein Vektor mit Länge eins heißt **Einheitsvektor**. Falls ein Vektor a eine Länge $\|a\| \neq 0$ hat, kann man ihn durch $\frac{1}{\|a\|} a$ auf Länge eins skalieren und so einen zu a parallelen Einheitsvektor erhalten. Es gilt nämlich die Rechenregel $\|\alpha a\| = |\alpha| \|a\|$ für $\alpha \in \mathbb{R}$.

Der ungerichtete **Winkel** $\varphi \in [0, \pi]$ zwischen zwei nicht-Null-Vektoren a und b kann implizit über die Gleichung

$$a \cdot b = \|a\| \, \|b\| \cos(\varphi) \tag{2.1}$$

definiert und explizit mit $\varphi = \arccos\left(\frac{a \cdot b}{\|a\| \, \|b\|}\right)$ berechnet werden. Der Term $\|b\| \cos(\varphi)$ in Gleichung (2.1) kann geometrisch als die vorzeichenbehaftete **orthogonale** (=rechtwinklige) **Projektion** von a auf b interpretiert werden, siehe Abbildung 2.1. Das innere Produkt $a \cdot b$ ist geometrisch somit gleich der Länge von a mal der Projektion von a auf b.

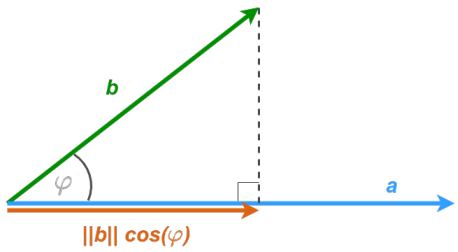


Abbildung 2.1.: Inneres Produkt

Aus dieser geometrischen Interpretation ergeben sich einige wichtige Folgerungen für nicht-Null-Vektoren a und b:

- $a \cdot b > 0 \iff a$ und b bilden einen spitzen Winkel $\varphi \in [0, \frac{\pi}{2})$ im Bogenmaß.
- $a \cdot b = 0 \iff a$ ist normal (orthogonal, rechtwinklig) auf b, also $\varphi = \frac{\pi}{2}$ im Bogenmaß. Man schreibt dies auch als $a \perp b$.
- $a \cdot b < 0 \iff a$ und b bilden einen stumpfen Winkel $\varphi \in (\frac{\pi}{2}, \pi]$ im Bogenmaß.

Da mit dem inneren Produkt Längen und Winkel berechnet werden können, wird durch das innere Produkt die euklidische Geometrie ``berechenbar'', d. h. analytisch, vgl. Kapitel 2.1.4.

In der Ebene $\mathbb{R}^2$ bilden die Vektoren $e_x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ und $e_y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ die sogenannte **Standardbasis** und heißen **Standardbasisvektoren**. Sie sind aufeinander normal und haben die Länge eins. Beides zusammen macht sie zu einem **Orthonormalsystem**. Im Raum $\mathbb{R}^3$ bilden die Standardbasisvektoren $e_x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $e_y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ und $e_z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ ein Orthonormalsystem.

2.1.3. Kreuz- und Spatprodukt

Im dreidimensional Raum $\mathbb{R}^3$ gibt es noch ein weiteres Produkt von zwei Vektoren, das sogenannte Kreuzprodukt, auch Vektorprodukt, vektorielles Produkt oder äußeres Produkt genannt. Es liefert als Ergebnis keine Zahl, sondern wieder einen dreidimensionalen Vektor:

$$a \times b = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ -(a_x b_z - a_z b_x) \\ a_x b_y - a_y b_x \end{pmatrix}$$

Der Vektor $a \times b$ ist geometrisch eindeutig durch die folgenden Eigenschaften bestimmt, vgl. Wikipedia und siehe Abbildung 2.2:

- $(a \times b) \perp a$ und $(a \times b) \perp b$.
- $\|(a \times b)\|$ ist gleich dem Flächeninhalt $\|a\| \, \|b\| \sin(\varphi)$ des von a und b aufgespannten Parallelogramms.
- $(a, b, a \times b)$ bilden ein Rechtssystem (Rechte-Hand-Regel).

Es gelten die folgenden Rechenregeln:

- $a \times (b + c) = a \times b + a \times c$
- $(a + b) \times c = a \times c + b \times c$
- $a \times b = -b \times a$

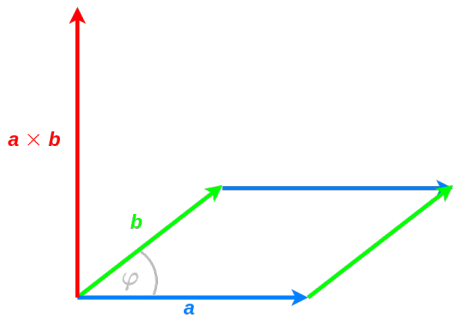


Abbildung 2.2.: Kreuzprodukt

- $\alpha(a \times b) = (\alpha a) \times b = a \times (\alpha b)$

Weiters gilt $a \times b = 0 \iff a$ und b sind kollinear und somit linear abhängig. Das Kreuzprodukt wird zum Beispiel verwendet, um in der Mechanik Drehimpuls und Drehmoment zu definieren und um in der Elektrotechnik die Lorentz-Kraft zu beschreiben.

Mit dem sogenannten **Spatprodukt** kann man das Volumen des durch die drei räumliche Vektoren $a, b, c \in \mathbb{R}^3$ aufgespannten Spats (Parallelepipeds) zu berechnen, siehe Abbildung 2.3.

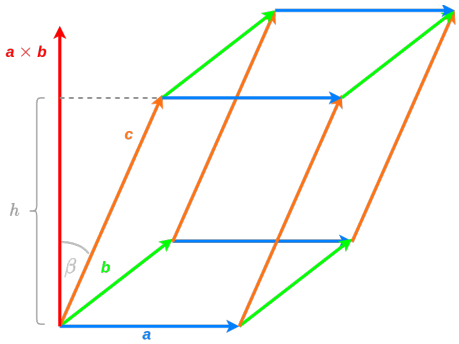


Abbildung 2.3.: Spatprodukt

Das Spatprodukt von a, b und c ist definiert als

$$[a, b, c] := (a \times b) \cdot c.$$

Es gilt $[a, b, c] = (a \times b) \cdot c = \|a \times b\| \|c\| \cos(\beta)$, wobei β den Winkel zwischen $a \times b$ und c bezeichnet. Der Terme $\|a \times b\|$ ist gleich dem Flächeninhalt G des von a und b aufgespannten Parallelogramms, und $\|c\| \cos(\beta)$ ist im Betrag gleich der Höhe h des Spats. Somit erhalten wir $|[a, b, c]| = Gh = V$ dem Volumen des Spats.

Es gilt $[a, b, c] = 0 \iff a, b, c$ sind linear abhängig. $\iff$ Der von a, b, c aufgespannte Spat hat Null Volumen. $\iff a, b, c$ liegen in einer Ebene.

2.1.4. Analytische Geometrie

2.1.4.1. Geraden

Eine **Gerade in der Ebene** kann neben der Form $y = kx + d$ auch in Parameterform und in Normalvektorform angegeben werden:

- **Parameterform**: Jeder Punkt $X \in \mathbb{R}^2$ der Gerade ist die Summe aus einem gegebenen Punkt $P \in \mathbb{R}^2$ der Geraden und einem Vielfachen $\lambda \in \mathbb{R}$ des gegebenen Richtungsvektors $r \in \mathbb{R}^2$:

$$X = P + \lambda r$$

  zum Beispiel:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix} + \lambda \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

- **Normalvektorform**: Gegeben ist ein Punkt $P \in \mathbb{R}^2$ der Geraden und ein Normalvektor $n \in \mathbb{R}^2$ der Geraden. Dann ist der Verbindungsvektor $\overrightarrow{PX}$ von P zu einem Punkt $X \in \mathbb{R}^2$ der Geraden normal auf den Normalvektor n:

$$\begin{aligned} n \cdot \overrightarrow{PX} &= 0 \\ n \cdot (X - P) &= 0 \\ n \cdot X - n \cdot P &= 0 \\ n \cdot X &= n \cdot P \end{aligned}$$

  zum Beispiel:

$$\begin{pmatrix} 2 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$
$$2x + y = 8$$

Bemerkung: In der Ebnen kann man ein Vektor (a_x, a_y) leicht um 90° gegen den Uhrzeigersinn oder 90° im Uhrzeigersinn drehen: $(-a_y, a_x)$ bzw. $(a_y, -a_x)$.

Eine **Gerade im Raum** kann in Parameterform $X = P + \lambda r \in \mathbb{R}^3$ oder als Schnitt zweier Ebenen angegeben werden.

2.1.4.2. Ebenen

Eine **Ebene im Raum** in Parameterform und in Normalvektorform angegeben werden:

- **Parameterform**: Jeder Punkt $X \in \mathbb{R}^3$ der Ebene ist die Summe aus einem gegebenen Punkt $P \in \mathbb{R}^3$ der Ebene, einem Vielfachen $\lambda \in \mathbb{R}$ eines ersten gegebenen Richtungsvektors $a \in \mathbb{R}^3$ und dem Vielfachen $\mu \in \mathbb{R}$ eines zweiten, nicht kollinearen gegebenen Richtungsvektors $b \in \mathbb{R}^3$:

$$X = P + \lambda a + \mu b$$

- **Normalvektorform**: Eine zum Fall der Geraden in der Ebene analoge Argumentation führt zur analogen Gleichung

$$n \cdot X = n \cdot P$$

  im Raum, zum Beispiel:

$$\begin{pmatrix} 3 \\ 1 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix}$$
$$3x + y + 4z = 7$$

2.1.4.3. Kreise und Kugeln

Der **Kreis** in der Ebene mit Radius $r > 0$ und Mittelpunkt $M = (u, v)$ besteht aus allen Punkten $X = (x, y)$ der Ebene, deren Abstand zum Mittelpunkt gleich dem Radius ist, d. h.

$$\begin{aligned} \|\overrightarrow{MX}\| &= r \\ \|X - M\| &= r \\ \|(x-u, y-v)\| &= r \\ \sqrt{(x-u)^2 + (y-v)^2} &= r \\ (x-u)^2 + (y-v)^2 &= r^2 \end{aligned}$$

Eine analoge Argumentation führt zur Gleichung $(x-u)^2 + (y-v)^2 + (z-w)^2 = r^2$ einer **Kugel** im Raum.

2.1.5. Python

Für die Implementierung der Vektorrechnung mit Python verwenden wir das Paket NumPy. Die notwendigen Funktionen werden im Kapitel Python Tutorial eingeführt. Wir verwenden dabei die folgenden, üblichen Abkürzungen:

```
import numpy as np
```

Die zentralen Python-Funktionen sind:

- `np.array([a, b, c])`: erstellt einen Vektor (1-dim. Array) aus der Liste `[a, b, c]` mit den Komponenten `a`, `b` und `c`.
- `np.linalg.norm(v)`: berechnet die Länge des Vektors `v`.
- `np.dot(v, w)` oder `v@w`: berechnet das innere Produkt der Vektoren `v` und `w`.
- `np.arccos(x)`: berechnet den Arcuscosinus von `x` im Bogenmaß.
- `np.cross(v, w)`: berechnet das Kreuzprodukt der Vektoren `v` und `w`.

2.2. Beispiele

2.2.1. Längen, Winkel, Flächeninhalt

Durch die drei Punkte $A = (1, 4, -2)$, $B = (3, 1, 0)$ und $C = (-1, 1, 2)$ werden die Ecken eines Dreiecks festgelegt. Berechnen Sie die Längen der drei Seiten, die Innenwinkel sowie den Flächeninhalt des Dreiecks.

Quelle: [Pap18] Kapitel II, Abschnitt 2 und 3, Aufgabe 17

Lösung: Siehe Abbildung 2.4 und Abbildung 2.5.

Python:

17) Festlegung der Seitenvektoren nach Bild A-18 $(\vec{a} + \vec{b} + \vec{c} = \vec{0})$:

$$\vec{a} = \overrightarrow{BC} = \begin{pmatrix} -4 \\ 0 \\ 2 \end{pmatrix}; \qquad \vec{b} = \overrightarrow{CA} = \begin{pmatrix} 2 \\ 3 \\ -4 \end{pmatrix}$$

$$\vec{c} = \overrightarrow{AB} = \begin{pmatrix} 2 \\ -3 \\ 2 \end{pmatrix}; \qquad |\vec{a}| = \sqrt{20};$$

$$|\vec{b}| = \sqrt{29}; \qquad |\vec{c}| = \sqrt{17}$$

Bild A-18

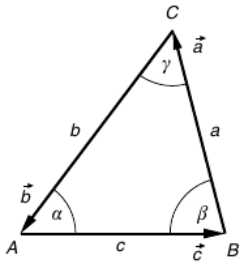


Abbildung 2.4.: Quelle: [Pap18] Kapitel II, Abschnitt 2 und 3, Aufgabe 17

α: Winkel zwischen $-\vec{b}$ und $\vec{c}$; β: Winkel zwischen $\vec{a}$ und $-\vec{c}$

$$\alpha = \arccos\left(\frac{(-\vec{b}) \cdot \vec{c}}{|-\vec{b}| \cdot |\vec{c}|}\right) = \arccos\left(\frac{-\vec{b} \cdot \vec{c}}{|\vec{b}| \cdot |\vec{c}|}\right) = \arccos\left(\frac{13}{\sqrt{29} \cdot \sqrt{17}}\right) = 54{,}16^\circ$$

$$\beta = \arccos\left(\frac{\vec{a} \cdot (-\vec{c})}{|\vec{a}| \cdot |-\vec{c}|}\right) = \arccos\left(\frac{-\vec{a} \cdot \vec{c}}{|\vec{a}| \cdot |\vec{c}|}\right) = \arccos\left(\frac{4}{\sqrt{20} \cdot \sqrt{17}}\right) = 77{,}47^\circ$$

$$\alpha + \beta + \gamma = 180^\circ \quad \Rightarrow \quad \gamma = 180^\circ - \alpha - \beta = 48{,}37^\circ$$

$$\textit{Fläche:}\ A = \frac{1}{2}|\vec{a} \times \vec{b}| = \frac{1}{2}|\vec{a}| \cdot |\vec{b}| \cdot \sin\gamma = \frac{1}{2}\sqrt{20} \cdot \sqrt{29} \cdot \sin 48{,}37^\circ = 9$$

Abbildung 2.5.: Quelle: [Pap18] Kapitel II, Abschnitt 2 und 3, Aufgabe 17

```
A = np.array([ 1, 4, -2])
B = np.array([ 3, 1,  0])
C = np.array([-1, 1,  2])

a = C - B  # Spitze minus Schaft
b = A - C  # Spitze minus Schaft
c = B - A  # Spitze minus Schaft

print(f"Vector a has norm {np.linalg.norm(a):.2f}.")
print(f"Vector b has norm {np.linalg.norm(b):.2f}.")
print(f"Vector c has norm {np.linalg.norm(c):.2f}.")

# Checks and variants:
inner_product_v1 = np.dot(a, b)
inner_product_v2 = a@b
print(f"Check: The inner product of a and b is {inner_product_v1:.2f} = {inner_product_v2:.2f}.")
norm_v1 = np.linalg.norm(a)
norm_v2 = np.sqrt(a@a)
print(f"Check: The norm of a is {norm_v1:.2f} = {norm_v2:.2f}.")

alpha = np.arccos(np.dot(-b, c)/(np.linalg.norm(-b)*np.linalg.norm(c)))
beta = np.arccos(np.dot(-c, a)/(np.linalg.norm(-c)*np.linalg.norm(a)))
gamma = np.arccos(np.dot(-a, b)/(np.linalg.norm(-a)*np.linalg.norm(b)))

print(f"The angle alpha is {np.degrees(alpha):.2f}°.")
```

```python
print(f"The angle beta is {np.degrees(beta):.2f}°.")
print(f"The angle gamma is {np.degrees(gamma):.2f}°.")

area = 0.5*np.linalg.norm(np.cross(a, b))
print(f"The area of the triangle is {area:.2f}.")
```

```
Vector a has norm 4.47.
Vector b has norm 5.39.
Vector c has norm 4.12.
Check: The inner product of a and b is -16.00 = -16.00.
Check: The norm of a is 4.47 = 4.47.
The angle alpha is 54.16°.
The angle beta is 77.47°.
The angle gamma is 48.37°.
The area of the triangle is 9.00.
```

2.2.2. Lineare Abhängigkeit

Zeigen Sie jeweils, dass die Vektoren linear abhängig sind.

1. $a = \begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix}$, $b = \begin{pmatrix} -6 \\ 3 \\ -9 \end{pmatrix}$
2. $a = \begin{pmatrix} 1 \\ 2 \\ 5 \end{pmatrix}$, $b = \begin{pmatrix} -1 \\ -2 \\ 3 \end{pmatrix}$, $c = \begin{pmatrix} 5 \\ 10 \\ 1 \end{pmatrix}$

Quelle: [Pap18] Kapitel II, Abschnitt 2+3, Aufgabe 31

Lösung:

1. Die Vektoren sind eine Vielfaches voneinander: $b = -3a$. Daher sind sie kollinear und linear abhängig.
2. Das Spatprodukt $[a, b, c]$ der drei Vektoren ergibt Null:

$$\begin{aligned} [a, b, c] &= (a \times b) \cdot c = \\ &= \left(\begin{pmatrix} 1 \\ 2 \\ 5 \end{pmatrix} \times \begin{pmatrix} -1 \\ -2 \\ 3 \end{pmatrix} \right) \cdot \begin{pmatrix} 5 \\ 10 \\ 1 \end{pmatrix} = \\ &= \begin{pmatrix} 16 \\ -8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 10 \\ 1 \end{pmatrix} = \\ &= 80 - 80 = 0. \end{aligned}$$

   Daher hat das von ihnen aufgespannte Parallelepiped kein Volumen. Die Vektoren liegen daher in einer Ebene und sind linear abhängig.

Python:

```python
# problem 2:
a = np.array([ 1, 2, 5])
b = np.array([-1,-2, 3])
c = np.array([ 5,10, 1])

print(f"The triple product of a, b, c is {np.dot(np.cross(a, b), c)}.")
```

```
The triple product of a, b, c is 0.
```

2.2.3. Lineare (Un-)Abhängigkeit

Sei $v_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, v_2 = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}$ und $v_3 = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$.

1. Bestimmen Sie, ob die Vektoren v_i linear (un)abhängig sind.
2. Falls möglich finden Sie eine Linearkombination $\alpha_1, \alpha_2, \alpha_3$ nicht Null, so dass $\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = 0$.

Lösung:

Die Vektoren v_i sind linear abhängig, da $[v_1, v_2, v_3] = 0$. Das lineare Gleichungssystem $\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = 0$ für die Koeffizienten $\alpha_i \in \mathbb{R}$ hat daher nicht-triviale Lösungen. In Komponenten ausgeschrieben lautet die Vektorgleichung $\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = 0$

$$\alpha_1 \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \alpha_2 \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} + \alpha_3 \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

In Gleichungsform:

$$\begin{aligned} 1\alpha_1 + 4\alpha_2 + 2\alpha_3 &= 0 \\ 2\alpha_1 + 5\alpha_2 + 1\alpha_3 &= 0 \\ 3\alpha_1 + 6\alpha_2 + 0\alpha_3 &= 0 \end{aligned}$$

Das lineare Gleichungssystem lässt sich z. B. mit dem Gaußschen Eliminationsverfahren lösen, siehe Kapitel Lineare Gleichungssysteme. Der Lösungsraum ist eindimensional und kann z. B. mit α_3 parametrisiert werden:

$$\begin{aligned} \alpha_1 &= 2\alpha_3 \\ \alpha_2 &= -\alpha_3 \\ \alpha_3 &= \text{frei wählbar} \end{aligned}$$

Das lineare Gleichungssystem $\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = 0$ hat daher auch nicht-triviale Lösungen, z. B. mit $\alpha_3 = -1$ gilt $-2v_1 + v_2 - v_3 = 0$.

Python: Siehe Kapitel Lineare Gleichungssysteme.

2.2.4. Gerade und Ebene

Welche Lage haben die Gerade g und die Ebene E zueinander? Bestimmen Sie gegebenenfalls Abstand, Schnittpunkt und Schnittwinkel.

$$g : X = \begin{pmatrix} 5 \\ 3 \\ 6 \end{pmatrix} + \lambda \begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}$$

$$E : \begin{pmatrix} 3 \\ -1 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} x-1 \\ y-1 \\ z-1 \end{pmatrix} = 0$$

Quelle: [Pap18] Kapitel II, Abschnitt 4, Aufgabe 19b

Lösung: Zur Bestimmung der Lage von Gerade und Ebene berechnen wir das innere Produkt des Richtungsvektors der Geraden mit dem Normalvektor der Ebene:

$$\begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ -1 \\ -1 \end{pmatrix} = 6 - 5 - 1 = 0.$$

Die beiden Vektoren sind daher orthogonal zueinander, und die Gerade ist parallel zur Ebene. Um ihren Abstand voneinander zu bestimmen, berechnen wir das innere Produkt des auf Länge eins skalierten Normalvektors der Ebene mit dem Verbindungsvektor zwischen dem gegebenen Punkt der Ebene und dem gegeben Punkt der Geraden:

$$\frac{1}{\sqrt{3^2 + (-1)^2 + (-1)^2}} \begin{pmatrix} 3 \\ -1 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} 5-1 \\ 3-1 \\ 3-1 \end{pmatrix} = \frac{5}{\sqrt{11}} = 1.51.$$

Gerade und Ebene sind somit im Abstand 1.51 parallel.

Python:

```python
inner_product = np.dot(np.array([2, 5, 1]), np.array([3, -1, -1]))
print(f"inner product = {inner_product:.2f}.")

n = np.array([3, -1, -1])
n_unit = n/np.linalg.norm(n)
connection = np.array([5 - 1, 3 - 1, 6 - 1])
d = np.dot(n_unit, connection)
d = np.abs(d)   # distance is always positive
print(f"The distance between the line and the plane is {d:.2f}.")
```

```
inner product = 0.00.
The distance between the line and the plane is 1.51.
```

2.2.5. Geometrische Figuren

Welche geometrischen Figuren werden durch die folgenden Gleichungen beschrieben?

1. $x^2 - 6x + y^2 + 8y = 0$
2. $9x^2 - 36x + 4y^2 + 24y + 36 = 0$

Lösung:

1. Durch Quadratische Ergänzung erhalten wir:

$$\begin{aligned} x^2 - 6x + y^2 + 8y &= 0 \\ (x-3)^2 - 9 + (y+4)^2 - 16 &= 0 \\ (x-3)^2 + (y+4)^2 &= 25 \\ (x-3)^2 + (y+4)^2 &= 5^2 \end{aligned}$$

   Die Gleichung beschreibt einen Kreis mit Mittelpunkt $(3, -4)$ und Radius 5.
2. Durch Quadratische Ergänzung erhalten wir:

$$\begin{aligned} 9x^2 - 36x + 4y^2 + 24y + 36 &= 0 \\ 9(x^2 - 4x) + 4(y^2 + 6y) + 36 &= 0 \\ 9((x-2)^2 - 4) + 4((y+3)^2 - 9) + 36 &= 0 \\ 9(x-2)^2 - 36 + 4(y+3)^2 - 36 + 36 &= 0 \\ 9(x-2)^2 + 4(y+3)^2 &= 36 \\ \frac{(x-2)^2}{4} + \frac{(y+3)^2}{9} &= 1 \end{aligned}$$

   Die Gleichung beschreibt eine verschobene Ellipse mit Mittelpunkt $M = (2, -3)$ und Halbachsen $a = 2$ und $b = 3$.

2.3. Aufgaben

2.3.1. Rechnen im Vektorraum

Gegeben sind die Vektoren $u = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$ und $v = \begin{pmatrix} -3 \\ -1 \end{pmatrix}$.

1. Berechnen Sie $-v$, $-2v$, $u + v$, $u - v$, $u - 2v$, und $\|v\|$ von Hand und in Python.
2. Überprüfen Sie Ihre Ergebnisse grafisch.

Ergebnis: $-v = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$, $-2v = \begin{pmatrix} 6 \\ 2 \end{pmatrix}$, $u + v = \begin{pmatrix} -4 \\ 1 \end{pmatrix}$, $u - v = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ und $u - 2v = \begin{pmatrix} 5 \\ 4 \end{pmatrix}$. $\|v\| = \sqrt{10} \simeq 3.16$.

2.3.2. Linearkombination

Schreiben Sie den Vektor $v = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$ als eine Linearkombination von zwei Vektoren, von denen einer auf der Geraden $y = \frac{x}{2}$ und der andere auf der Geraden $y = 2x$ liegt. Überprüfen Sie Ihr Ergebnis grafisch.

Ergebnis: $v = \frac{4}{3}\begin{pmatrix} 2 \\ 1 \end{pmatrix} + \frac{7}{3}\begin{pmatrix} 1 \\ 2 \end{pmatrix}$.

2.3.3. Winkel

Rechnen Sie zuerst von Hand, und überprüfen Sie Ihr Ergebnis anschließend in Python:

1. Welchen Winkel schließen die Vektoren $a = \begin{pmatrix} 10 \\ -5 \\ 10 \end{pmatrix}$ und $b = \begin{pmatrix} 3 \\ -1 \\ -\frac{1}{2} \end{pmatrix}$ miteinander ein?
2. Zeigen Sie, dass die drei Vektoren $a = \begin{pmatrix} 1 \\ 4 \\ -2 \end{pmatrix}$, $b = \begin{pmatrix} -2 \\ 2 \\ 3 \end{pmatrix}$ und $c = \begin{pmatrix} -1 \\ 6 \\ 1 \end{pmatrix}$ ein rechtwinkliges Dreieck bilden. Welche Vektoren sind Katheten, und welcher Vektor ist die Hypothenuse?

Quelle: [Pap18] Kapitel II, Abschnitt 2+3, Aufgaben 11b und 15

Ergebnis:

1. 51.34°
2. $a \perp b$. Daher ist c die Hypothenuse.

2.3.4. Winkel

Rechnen Sie zuerst von Hand, und überprüfen Sie Ihr Ergebnis anschließend in Python:

1. Welchen Winkel schließen die Vektoren $a = \begin{pmatrix} 3 \\ 1 \\ -2 \end{pmatrix}$ und $b = \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix}$ miteinander ein?
2. Zeigen Sie, dass die Vektoren $a = \begin{pmatrix} -1 \\ 2 \\ 5 \end{pmatrix}$ und $b = \begin{pmatrix} -4 \\ 8 \\ -4 \end{pmatrix}$ zueinander orthogonal sind.

Quelle: [Pap18] Kapitel II, Abschnitt 2+3, Aufgaben 11a und 12a

Ergebnis:

1. 79.92°
2. Das innere Produkt ergibt Null.

2.3.5. Orthonormiertes System

Zeigen Sie von Hand und in Python: Die Vektoren $a = \begin{pmatrix} 1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \end{pmatrix}$, $b = \begin{pmatrix} 1/\sqrt{2} \\ 0 \\ -1/\sqrt{2} \end{pmatrix}$, $c = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}$ bilden ein orthonormiertes System, d. h. die Vektoren stehen paarweise senkrecht aufeinander und besitzen jeweils die Länge 1.

Quelle: [Pap18] Kapitel II, Abschnitt 2-3, Aufgabe 14

Ergebnis: Alle Vektoren sind paarweise orthogonal und haben die Länge 1.

2.3.6. Volumen

Bestimmen Sie von Hand und in Python das Volumen des von den Vektoren $a = \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}$, $b = \begin{pmatrix} 3 \\ 4 \\ 7 \end{pmatrix}$ und $c = \begin{pmatrix} 1 \\ 2 \\ -8 \end{pmatrix}$ gebildeten Parallelepipeds (=Spats). Sind die Vektoren linear unabhängig oder linear abhängig?

Quelle: [Pap18] Kapitel II, Abschnitt 2+3, Aufgabe 28

Ergebnis: $V = 75$. Die Vektoren sind linear unabhängig.

2.3.7. Spatprodukt

Zeigen Sie mit Hilfe des Spatprodukts von Hand und in Python: Die Vektoren $a = \begin{pmatrix} -3 \\ 4 \\ 0 \end{pmatrix}$, $b = \begin{pmatrix} -2 \\ 3 \\ 5 \end{pmatrix}$ und $c = \begin{pmatrix} -1 \\ 3 \\ 25 \end{pmatrix}$ liegen in einer gemeinsamen Ebene.

Quelle: [Pap18] Kapitel II, Abschnitt 2+3, Aufgabe 27a

Ergebnis: $[a, b, c] = 0$

2.3.8. Lineare Abhängigkeit

Zeigen Sie von Hand und in Python: Die Vektoren sind linear abhängig: $a = \begin{pmatrix} 1 \\ 2 \\ 5 \end{pmatrix}$, $b = \begin{pmatrix} -1 \\ -2 \\ 3 \end{pmatrix}$, $c = \begin{pmatrix} 5 \\ 10 \\ 1 \end{pmatrix}$.

Quelle: [Pap18] Kapitel II, Abschnitt 2-3, Aufgabe 31

Ergebnis: $[a, b, c] = 0$

2.3.9. Gerade und Ebene

Gegeben sind eine Gerade g und eine Ebene E:

$$g : X = \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix}$$
$$E : 2x + y + z = 1$$

1. Zeigen Sie von Hand und in Python, ohne einen Schnittpunkt zu berechnen, dass sich die Gerade und die Ebene schneiden.
2. Berechnen Sie von Hand den Schnittpunkt, und von Hand und in Python den Schnittwinkel.

Quelle: [Pap18] Kapitel II, Abschnitt 4, Aufgabe 23:

Ergebnis:

1. Der Richtungsvektor der Geraden und der Normalvektor der Ebene bilden keinen rechten Winkel.
2. Schnittpunkt $S = (-4, -12, 21)$, Schnittwinkel $\varphi = 6.26°$

2.3.10. Gerade und Ebene

Eine Ebene enthält den Punkt $P = (2, 1, 8)$ und hat den Normalvektor $n = \begin{pmatrix} 2 \\ -6 \\ 1 \end{pmatrix}$.

1. Zeigen Sie von Hand und in Python, dass die Gerade $g : X = \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix} + \lambda \begin{pmatrix} 4 \\ 1 \\ -2 \end{pmatrix}$ zu dieser Ebene parallel ist.
2. Berechnen Sie von Hand und in Python den Abstand zwischen Gerade und Ebene.

Ergebnis:

1. Ein Richtungsvektor der Graden g ist gegeben durch $r = \begin{pmatrix} 4 \\ 1 \\ -2 \end{pmatrix}$. Da $n \cdot r = 0$ ergibt, sind die Gerade und die Ebene parallel.
2. Abstand $d = 2.03$.

2.3.11. Schnittpunkt zweier Geraden

Bestimmen Sie von Hand den Schnittpunkt der beiden Geraden

$$\begin{aligned} 2x - 3y &= 8 \\ -x + 4y &= -9 \end{aligned}$$

und überprüfen Sie Ihr Ergebnis grafisch.

Ergebnis: $S = (1, -2)$

2.3.12. Ebenengleichung

Berechnen Sie von Hand die Gleichung jener Ebene, die alle drei Koordinatenachsen im selben Abstand vom Ursprung schneidet und durch den Punkt $P = (3, -4, 7)$ geht.

Ergebnis: $x + y + z = 6$

2.3.13. Abstand zweier Ebenen

Zeigen Sie von Hand und in Python, dass die folgenden beiden Ebenen parallel sind, und berechnen Sie ihren Abstand.

$$E_1 : \text{Punkt } P_1 = (3, 5, 6), \text{Normalvektor } n_1 = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}$$

$$E_2 : \text{Punkt } P_2 = (1, 5, -2), \text{Normalvektor } n_2 = \begin{pmatrix} -3 \\ -9 \\ 6 \end{pmatrix}$$

Ergebnis: Die Ebenen sind parallel, weil die Normalvektoren kollinear sind. Abstand $d = 3.74$.

2.3.14. Mehr Aufgaben

- [Pap18] Kapitel II
  - Abschnitt 2 und 3, Aufgaben 1 - 32
  - Abschnitt 4, Aufgaben 1 - 25
- [Pap20] Kapitel I Vektorrechnung, Abschnitt 1 Vektoroperationen: I1 - I43
- [Pap19] Kapitel I Vektorrechnung, Beispiele 1 - 19

3. Lineare Gleichungssysteme

3.1. Methoden

Motivation

Wenn mehrere lineare Gleichungen von den Variablen erfüllt werden sollen, spricht man von einem linearen Gleichungssystem, das wir oft als **LGS** abkürzen. Englisch: system of linear equations oder linear system. Lineare Gleichungssysteme kommen in vielen Anwendungen vor: Geometrie, Naturwissenschaften, Technik, Informatik, Wirtschaft etc. Sie haben die Vorteile, dass sie mit Hilfe von Matrizen und Vektoren kompakt dargestellt werden können, ihre Lösbarkeit leicht bestimmbar ist und dass es effiziente Lösungsverfahren gibt. Zudem können viele nicht-lineare Probleme näherungsweise durch lineare Gleichungssysteme beschrieben werden, siehe z. B. das Newtonverfahren oder die Finite-Elemente-Methode.

3.1.1. Modellierung

Wie kommt man zu einem linearen Gleichungssystem? Im folgenden betrachten wir einige Beispiele aus verschiedenen Anwendungsgebieten.

3.1.1.1. Geometrie

Wann schneiden sich zwei Geraden in der Ebene, und wie berechnet man ggf. den Schnittpunkt? Jede Gerade kann durch eine lineare Gleichung der Form $ax + by = c$ beschrieben werden. Dabei sind a, b und c gegebene Zahlen. Linear bedeutet hier, dass die Variablen x und y nur mit einer Zahl multipliziert und addiert werden dürfen, aber z. B. nicht potenziert oder multipliziert werden dürfen. Die Gleichung $ax + by = c$ beschreibt alle Punkte (x, y), die auf der Geraden liegen. Wir suchen nun alle Punkte (x, y), die beide Gleichungen erfüllen, also auf beiden Geraden liegen. Diese Schnittpunkte bilden die Lösungsmenge des linearen Gleichungssystems. Unsere geometrische Vorstellung sagt uns, dass es drei Möglichkeiten gibt: Die Geraden schneiden sich in einem Punkt, sie sind parallel und haben keinen Schnittpunkt, oder sie sind identisch und haben unendlich viele Schnittpunkte. Die geometrische Intuition stimmt und lässt auf höhere als zwei Dimensionen verallgemeinern.

Beispiel:

$$\begin{aligned} 2x - 5y &= 8 \\ -x + 7y &= 5 \end{aligned}$$

Die Geraden schneiden sich in einem Punkt, nämlich im Punkt $(x, y) = (9, 2)$.

3.1.1.2. Kinematik

Die Kinematik beschäftigt sich mit der Beschreibung der Bewegung von Körpern. Wenn sich zum Beispiel zwei Flugzeuge in einer Ebene (z. B. auf einer Landkarte) gleichförmig geradlinig bewegen, dann können ihre Positionen zu jedem Zeitpunkt t durch Vektoren $P_1(t) = Q_1 + tv_1$ und $P_2(t) = Q_2 + tv_2$ mit $Q_i, v_i \in \mathbb{R}^2$ für $i = 1, 2$ beschrieben werden, siehe Geradengleichung in Parameterform. Die Punkte Q_i sind ihre Positionen zum Zeitpunkt $t = 0$, und die Vektoren v_i sind die Geschwindigkeiten der Flugzeuge.

Beispiel:

$$P_1(t) = \begin{pmatrix} 1 \\ -2 \end{pmatrix} + t \begin{pmatrix} -3 \\ 4 \end{pmatrix}$$
$$P_2(t) = \begin{pmatrix} -6 \\ 6 \end{pmatrix} + t \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Wir suchen den Schnittpunkt ihrer Bewegungsbahnen und die Zeiten, zu denen die Körper diesen erreichen. Dazu lösen wir die Vektorgleichung $P_1(t_1) = P_2(t_2)$ nach t_1 und t_2. Dies ist gleichwertig mit dem linearen Gleichungssystem

$$\begin{aligned} 1 - 3t_1 &= -6 + t_2 \\ -2 + 4t_1 &= 6 - t_2 \end{aligned}$$

Die Lösung ist gegeben durch $t_1 = 1$ und $t_2 = 4$. Die Bahnen der Flugzeuge scheiden sich also an der Position $P_1(1) = P_2(4) = \begin{pmatrix} -2 \\ 2 \end{pmatrix}$. Das Flugzeug 1 hat diesen Punkt zum Zeitpunkt $t_1 = 1$ erreicht, und das Flugzeug 2 zum Zeitpunkt $t_2 = 4$. Da die Vektorgleichung als lineares Gleichungssystem geschrieben werden kann, ist die Lösungsstruktur dieselbe: Es kann keine, eine oder unendlich viele Lösungen geben.

3.1.1.3. Kräftezerlegung

Um eine gegebene Kraft $F = \begin{pmatrix} 7 \\ -7 \\ 4 \end{pmatrix}$ in ihre Komponenten bzgl. den zwei Richtungen $v_1 = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}$ und $v_2 = \begin{pmatrix} 1 \\ -3 \\ 2 \end{pmatrix}$ zu zerlegen, müssen wir die Vektorgleichung $x_1 v_1 + x_2 v_2 = F$ nach x_1 und x_2 lösen. Die Vektorgleichung

$$x_1 \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} + x_2 \begin{pmatrix} 1 \\ -3 \\ 2 \end{pmatrix} = \begin{pmatrix} 7 \\ -7 \\ 4 \end{pmatrix}$$

ist gleichwertig mit dem linearen Gleichungssystem

$$\begin{aligned} 2x_1 + x_2 &= 7 \\ x_1 - 3x_2 &= -7 \\ -x_1 + 2x_2 &= 4 \end{aligned}$$

Die Lösung ist gegeben durch $x_1 = 2$ und $x_2 = 3$, und die Zerlegung der Kraft lautet $F = 2v_1 + 3v_2$.

Nicht jede Kraft $F \in \mathbb{R}^3$ kann in die gegeben zwei Richtungen $v_1, v_2 \in \mathbb{R}^3$ zerlegt werden. Nur wenn die Kraft in der von den beiden Richtungen aufgespannten Ebene liegt, ist eine Zerlegung möglich. Falls die beiden Richtungen linear abhängig sind, d. h. wenn sie auf einer Geraden liegen, ist einen Zerlegung (falls sie existiert) nicht eindeutig, und es gibt unendlich viele Zerlegungen. Wieder ist die Lösungsstruktur dieselbe: Es kann keine, eine oder unendlich viele Lösungen geben.

3.1.1.4. Elektrischer Schaltkreis

Die Schaltung in Abbildung Abbildung 3.1 hat die Daten $U_{\text{bat},1} = 10.8$ V, $U_{\text{bat},2} = 3.2$ V, $R_1 = 6\ \Omega$, $R_2 = 8\ \Omega$, $R_3 = 4\ \Omega$.

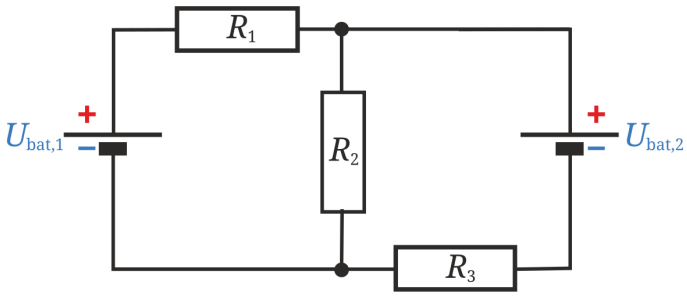


Abbildung 3.1.: Elektrischer Schaltkreis

Die Kirchhoffschen Regeln besagen:

- Die Summe aller Ströme, die in einen Knoten hineinfließen, ist gleich der Summe aller Ströme, die aus dem Knoten herausfließen.
- Die Summe aller Spannungen in einer Masche ist gleich Null.

Diese Regeln wenden wir in Abbildung 3.2 auf die Schaltung an und erhalten das folgende lineare Gleichungssystem.

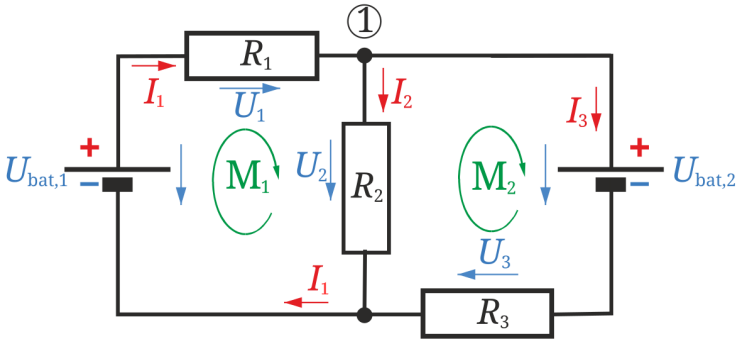


Abbildung 3.2.: Anwendung der Kirchhoffschen Regeln

$$
\begin{aligned}
I_1 - I_2 - I_3 &= 0 \\
I_1 R_1 + I_2 R_2 &= U_{\text{bat},1} \\
I_2 R_2 - I_3 R_3 &= U_{\text{bat},2}
\end{aligned}
$$

Die Lösung ist gegeben durch $I = 1$ A, $I_2 = 0.6$ A, $I_3 = 0.4$ A. Das stimmt mit unserer physikalischen Intuition überein, dass es eine eindeutige Lösung geben sollte.

Quelle und Details: LEIFIphysik

3.1.1.5. Wirtschaft

Sie haben Ihr Moped mit einer Mischung von Superbenzin und E10 getankt. Dabei haben Sie für 5 Liter dieser Mischung insgesamt 6.50 Euro bezahlt. Wie viel Liter sind von jeder Sorte getankt worden, wenn 1 Liter Superbenzin 1.35 EUR und 1 Liter E10 1.20 EUR kosten?

Wir führen die Variablen x_1 und x_2 ein, die die Literzahl von Superbenzin bzw. E10 bezeichnen. Dann können wir die obige Fragestellung als das folgende lineare Gleichungssystem formulieren.

$$\begin{aligned} x_1 + x_2 &= 5 \\ 1.35x_1 + 1.2x_2 &= 6.5 \end{aligned}$$

Lösung: Es wurden $x_1 = \frac{10}{3} \simeq 3.3$ Liter Superbenzin und $x_2 = \frac{5}{3} \simeq 1.7$ Liter E10 getankt.

Bezüglich der allgemeinen Lösungseigenschaften von Problemen dieses Typs halten wir fest: Wenn die Preise von Superbenzin und E10 unterschiedlich sind, dann gibt es eine eindeutige Lösung. Wenn die Preise gleich sind, dann gibt es keine Lösung, wenn die Gesamtmenge mal dem Preis nicht den angegebenen Kosten entspricht, oder unendlich viele Lösungen, wenn diese Inkonsistenz nicht gegeben ist.

Quelle: Aufgabe 7 in Serlo

3.1.1.6. Chemie

Chemische Reaktionsgleichungen beschreiben die Verhältnisse der Reaktanten und Produkte einer gegebenen Stoffumwandlung. Zum Beispiel reagiert bei der Verbrennung das Gas Propan (C_3H_8) mit Sauerstoff (O_2) zu Kohlendioxid (CO_2) und Wasser (H_2O), beschrieben durch die Reaktionsgleichung

$$x_1C_3H_8 + x_2O_2 \to x_3CO_2 + x_4H_2O.$$

Um die Reaktion ins Gleichgewicht zu bringen, gilt es die ganzzahligen Lösungen $x_1, ..., x_4$ zu finden, so dass die Gesamtzahlen von Kohlenstoff-, Wasserstoff- und Sauerstoffatomen auf der rechten und linken Seite der Reaktionsgleichung übereinstimmen. Eine systematische Methode, um das Reaktionsgleichgewicht zu bestimmen, ist die Formulierung einer vektoriellen Gleichung, welche die Zahlen der Atome aller in der Reaktion vorkommenden Elemente beschreibt:

$$x_1 \begin{pmatrix} 3 \\ 8 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix} = x_3 \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} + x_4 \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix}.$$

Die vektorielle Gleichung entspricht den Massenbilanzen für die drei Elemente C, H und O. Wir bestimmen die Lösungsmenge in $\mathbb{R}^4$ und finden die Lösung mit kleinstmöglichen ganzen Zahlen $x_1, ..., x_4$: Das Gaußschen Eliminationsverfahren, das wir später genau durchgehen, liefert, dass das lineare Gleichungssystem unendlich viele Lösungen (genauer: eine 1-dimensionale Lösungsmenge) hat, die z. B. mit x_4 parametrisiert werden kann:

$$\begin{aligned} x_4 &= \text{frei wählbar} \\ x_3 &= \frac{3}{4}x_4 \\ x_2 &= \frac{5}{4}x_4 \\ x_1 &= \frac{1}{4}x_4 \end{aligned}$$

Die Lösung mit kleinstmöglichen ganzen Zahlen ist $x_1 = 1, x_2 = 5, x_3 = 3, x_4 = 4$, also

$$C_3H_8 + 5O_2 \to 3CO_2 + 4H_2O.$$

Dass es unendlich viele Lösungen geben muss, falls die Reaktion chemisch möglich ist, leuchtet ein, da man von den Reaktanten beliebig viele Moleküle im gleichen Verhältnis nehmen kann, um die Reaktion zu starten.

3.1.2. Schreibweisen

3.1.2.1. Gleichungsform

Ein lineares Gleichungssystem mit n Variablen $x_1, ..., x_n$ und m Gleichungen hat die **Gleichungsform**

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + ... + A_{1n}x_n &= b_1 \\ A_{21}x_1 + A_{22}x_2 + ... + A_{2n}x_n &= b_2 \\ &... \\ A_{m1}x_1 + A_{m2}x_2 + ... + A_{mn}x_n &= b_m \end{aligned}$$

3.1.2.2. Matrix-Vektor-Form

Zur Übersichtlichkeit, zur Behandlung im Rahmen der Matrizenrechnung und zur Bestimmung der Lösung am Computer wird das lineare Gleichungssystem in **Matrix-Vektor-Form** geschrieben. Dabei werden die gegebenen Zahlen A_{ij} in eine $m \times n$ Matrix A zusammengefasst, die Variablen x_i in einen Vektor $x \in \mathbb{R}^n$ und die gegebenen Zahlen b_i in einen Vektor $b \in \mathbb{R}^m$ geschrieben. Die Daten des Problems sind also die Matrix A und der Vektor b. Die gesuchte Größe ist der Variablen-Vektor x. Die Matrix-Vektor-Form lautet:

$$\begin{pmatrix} A_{11} & A_{12} & ... & A_{1n} \\ A_{21} & A_{22} & ... & A_{2n} \\ ... & ... & ... & ... \\ A_{m1} & A_{m2} & ... & A_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_m \end{pmatrix}$$

Die Matrix-Vektor-Form kann kompakt als

$$Ax = b$$

geschrieben werden, was eine Vektorgleichung im $\mathbb{R}^m$ ist. Die Matrix A heißt Koeffizientenmatrix. Der Vektor b heißt rechte Seite. Das Produkt Ax von A mit x ist das Ergebnis der sogenannten Matrixmultiplikation, die wir im Abschnitt Matrizenrechnung genauer betrachten werden. Sie ist definiert durch

$$\begin{pmatrix} A_{11} & A_{12} & ... & A_{1n} \\ A_{21} & A_{22} & ... & A_{2n} \\ ... & ... & ... & ... \\ A_{m1} & A_{m2} & ... & A_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{12}x_2 + ... + A_{1n}x_n \\ A_{21}x_1 + A_{22}x_2 + ... + A_{2n}x_n \\ ... \\ A_{m1}x_1 + A_{m2}x_2 + ... + A_{mn}x_n \end{pmatrix}$$

Damit ist die Notation $Ax = b$ nicht nur kompakt, sondern es stehen einem auch die Methoden der Matrizenrechnung zur Verfügung, um das lineare Gleichungssystem zu lösen. Insbesondere ist die Matrixmultiplikation linear, d. h. es gilt $A(x_1 + x_2) = Ax_1 + Ax_2$ und $A(cx) = cAx$ für alle $x_1, x_2, x \in \mathbb{R}^n$ und $c \in \mathbb{R}$.

3.1.2.3. Erweiterte Koeffizientenmatrix

Zum effizienten händischen Berechnen der Lösung verwenden wir noch eine weitere Schreibweise, die **erweiterte Koeffizientenmatrix** $(A|b)$. Dabei werden die Daten des LGS , d. h. die $m \times n$ Koeffizientenmatrix A und der $m \times 1$ Vektor b, in eine $m \times (n+1)$ Matrix $(A|b)$ zusammengefasst:

$$(A|b) = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} & b_1 \\ A_{21} & A_{22} & \dots & A_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ A_{m1} & A_{m2} & \dots & A_{mn} & b_m \end{pmatrix}$$

3.1.3. Begriffe

- Die **Lösungsmenge (Lösungsraum)** eines LGS $Ax = b$ ist die Menge aller Vektoren x, die das LGS erfüllen.
- Ein LGS heißt **homogen**, wenn $b = 0$, d. h. wenn es sich als $Ax = 0$ schreiben lässt.
- Die Lösungsmenge eines homogenen LGS $Ax = 0$ heißt Kern oder Nullraum (engl. null space) der Matrix A.
- Wenn $b \neq 0$, dann heißt das LGS **inhomogen**.
- Wenn $m = n$, d. h. wenn das LGS gleich viele Gleichungen wie Unbekannte (=Variablen) hat, dann spricht man von einem **quadratischen** LGS.
- Wenn $m > n$, dann heißt das LGS **überbestimmt**.
- Wenn $m < n$, dann heißt das LGS **unterbestimmt**.

3.1.4. Lösungsstruktur Teil 1

Die Lösungsmenge eines LGS kann im Allgemeinen leer sein, oder aus genau einer Lösung bestehen, oder unendlich viele Lösungen enthalten. Etwas mehr kann man über inhomogene und homogene LGS sagen:

- Ein homogenes LGS $Ax = 0$ hat immer mindestens eine Lösung, nämlich die triviale Lösung (Nulllösung) $x = 0$. Wenn es eine weitere Lösung gibt, dann gibt es unendlich viele Lösungen.
- Ein inhomogenes LGS $Ax = b$ mit $b \neq 0$ hat entweder keine Lösung, genau eine Lösung oder unendlich viele Lösungen. Der Lösungsraum besteht aus dem Lösungsraum des homogenen LGS $Ax = 0$ und einer beliebigen speziellen (=partikulären) Lösung des inhomogenen LGS $Ax = b$. Das stimmt, weil man zu jeder speziellen Lösung eine beliebige Lösung des homogenen LGS addieren kann, und weil die Differenz zweier spezieller Lösungen eine Lösung des homogenen LGS ist:
$$Ax_1 = b, Ax_0 = 0 \Rightarrow A(x_1 - x_0) = Ax_1 - Ax_0 = b - 0 = b,$$
$$Ax_1 = b, Ax_2 = b \Rightarrow A(x_1 - x_2) = Ax_1 - Ax_2 = b - b = 0.$$

> **Achtung**
>
> Die Dimensionen m und n eines LGS - d. h ob das LGS mehr, weniger oder mehr Gleichungen als Variablen hat - geben keinen Aufschluss darüber, wie viele Lösungen (keine, eine oder unendlich viele) es hat!

3.1.5. Lösungsverfahren

3.1.5.1. Einsetzen und Gleichsetzen

Bei LGS mit sehr kleinen Dimensionen n und m oder speziellen Strukturen kann man durch Einsetzen und Gleichsetzen die Lösung finden, siehe Gleichsetzungsverfahren und Einsetzungsverfahren.

3.1.5.2. Gaußsches Eliminationsverfahren

Das Gaußsche Eliminationsverfahren (GEV) ist dagegen immer anwendbar und ist das Standardlösungsverfahren für LGS. Die Idee des GEV besteht darin, die erweiterte Koeffizientenmatrix $(A|b)$ eines LGS $Ax = b$ durch elementare Umformungen in eine sogenannte Trapezform (Stufenform, Treppenform) zu bringen, aus der die Lösungsmenge leicht berechnet werden kann. Die elementaren Umformungen ändern zwar die Daten (A und b) des LGS aber nicht die Lösungsmenge. Die elementaren Umformungen sind:

- Vertauschen zweier Gleichungen
- Multiplikation einer Gleichung mit einer Zahl $\neq 0$
- Addition einer Gleichung zu einer anderen Gleichung
- Vertauschen zweier Spalten, d. h. Vertauschen zweier Variablen

Eine Matrix T ist in **Trapezform**, wenn sie in der folgenden Form ist:

$$\begin{pmatrix} T_{11} & T_{12} & \dots & T_{1r} & T_{1,r+1} & \dots & T_{1n} \\ 0 & T_{22} & \dots & T_{1r} & T_{2,r+1} & \dots & T_{2n} \\ 0 & 0 & \ddots & \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \dots & T_{rr} & T_{r,r+1} & \dots & T_{rn} \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{pmatrix}$$

Dabei sind $T_{11} \neq 0$, $T_{22} \neq 0$, ..., $T_{rr} \neq 0$. An der Anzahl r an Stufen kann der **Rang** (engl. rank) der Matrix abgelesen werden. Man schreibt $\operatorname{rank}(T) = r$. Der Rang einer Matrix ist definiert als die Anzahl linear unabhängiger Spalten- oder Zeilenvektoren der Matrix. Durch elementare Umformungen ändert sich der Rang einer Matrix nicht.

Beispiel:

$$\begin{aligned} x_1 + 2x_2 - 2x_3 &= 7 \\ 2x_1 + 3x_2 &= 0 \\ 3x_1 + 3x_2 + 6x_3 &= -21 \end{aligned}$$

Wir bringen das LGS, das dem Schnitt von drei Ebenen im Raum entspricht, zuerst von der Gleichungsform in die erweiterte Koeffizientenmatrix:

$$\begin{pmatrix} 1 & 2 & -2 & 7 \\ 2 & 3 & 0 & 0 \\ 3 & 3 & 6 & -21 \end{pmatrix}$$

Wir addieren das -2-fache der ersten Gleichung zur zweiten Gleichung und das -3-fache der ersten Gleichung zur dritten Gleichung:

$$\begin{pmatrix} 1 & 2 & -2 & 7 \\ 0 & -1 & 4 & -14 \\ 0 & -3 & 12 & -42 \end{pmatrix}$$

Wir addieren das -3-fache der zweiten Gleichung zur dritten Gleichung:

$$\begin{pmatrix} 1 & 2 & -2 & 7 \\ 0 & -1 & 4 & -14 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Nun ist die Trapezform erreicht. Man hätte die dritte Gleichung anfangs durch 3 dividieren können, um die Zahlen klein zu halten. Der Rang der Koeffizientenmatrix und der erweiterten Koeffizientenmatrix ist 2. Wir können die Lösungsmenge nun leicht durch Rückwärtseinsetzen bestimmen:

- Die Variable x_3 ist frei wählbar, d. h. sie kann beliebige Werte aus $\mathbb{R}$ annehmen.
- Die Variable x_2 ist durch x_3 bestimmt, d. h. sie kann nicht frei gewählt werden: Aus der zweiten Zeile (Gleichung) $-x_2 + 4x_3 = -14$ folgern wir, dass $x_2 = 4x_3 + 14$.
- Die Variable x_1 ist durch x_2 und x_3 bestimmt, d. h. sie kann nicht frei gewählt werden: Aus der ersten Zeile (Gleichung) $x_1 + 2x_2 - 2x_3 = 7$ folgern wir, dass $x_1 = -2x_2 + 2x_3 + 7 = -6x_3 - 21$.

Die Lösungsmenge wir also aus allen $x = \begin{pmatrix} -6x_3 - 21 \\ 4x_3 + 14 \\ x_3 \end{pmatrix}$ mit $x_3 \in \mathbb{R}$ gebildet. Die Lösungsmenge ist 1-dimensional, da eine Variable frei gewählt werden kann. Geometrisch ist die Lösungsmenge eine Gerade im $\mathbb{R}^3$ mit der Geradengleichung $x = \begin{pmatrix} -21 \\ 14 \\ 0 \end{pmatrix} + x_3 \begin{pmatrix} -6 \\ 4 \\ 1 \end{pmatrix}$.

Allgemeine Strategie zur Umformung auf Trapezform:

1. Man schreibt das LGS als erweiterte Koeffizientenmatrix.
2. Wenn die Zahl in der ersten Spalte und ersten Zeile - das erste Pivotelement - nicht Null ist, dann vertauscht man die erste Zeile mit einer passenden anderen Zeile. Ist dies nicht möglich, vertauscht man die erste Spalte mit einer passenden anderen Spalte der Koeffizientenmatrix.
3. Außer dem trivialen Fall, dass die Koeffizientenmatrix nur Nullen enthält, können wir nun annehmen, dass das erste Pivotelement nicht Null ist. Indem man passende Vielfache der ersten Zeile von den anderen Zeilen subtrahiert, können wir Nullen unter dem ersten Pivotelement erzeugen.
4. Nun verwendet man die Zahl in der zweiten Zeile und zwei Spalten - das zweite Pivotelement - um Nullen unter dem zweiten Pivotelement zu erzeugen. Evtl. müssen davor noch Zeilen- oder Spaltenvertauschungen (aber nicht mit der ersten Zeile oder Spalte) durchgeführt werden, damit das zweite Pivotelement nicht Null ist.
5. Etc.

Wenn man eine allgemeinere Art der Stufenform zulässt, bei der sich in jeder Zeile die Zahl der Variablen auch um mehr als eine verringern kann, dann sind keine Spaltenvertauschungen (und somit auch keine Variablenvertauschungen) notwendig.

3.1.6. Lösungsstruktur Teil 2

Wir betrachten ein LGS $Ax = b$ mit m Gleichungen und n Variablen. Die $m \times n$-Koeffizientenmatrix A habe den Rang r, der sicher kleiner gleich m und kleiner gleich n ist. Dann gilt:

- Falls $r = m$, dann existiert mindestens eine Lösung.
- Falls $r = n$, dann ist eine Lösung, falls sie existiert, eindeutig.
- Falls $r = m = n$, dann existiert genau eine Lösung.

Um die Lösungsstruktur besser beurteilen zu können, muss man auch den Rang der erweiterten Koeffizientenmatrix $(A|b)$ kennen:

- Falls $r = \text{rank}(A) = \text{rank}(A|b)$, dann ist das LGS lösbar:
  - Falls $r = n$, dann hat das LGS genau eine Lösung.
  - Falls $r < n$, dann hat das LGS unendlich viele Lösungen mit $n-r$ frei wählbaren Variablen. Die Lösungsmenge ist also $n-r$-dimensional.
- Falls $\text{rank}(A) < \text{rank}(A|b)$, dann hat das LGS keine Lösung.

Die letzten drei Fälle sind in Abbildung Abbildung 3.3 dargestellt.

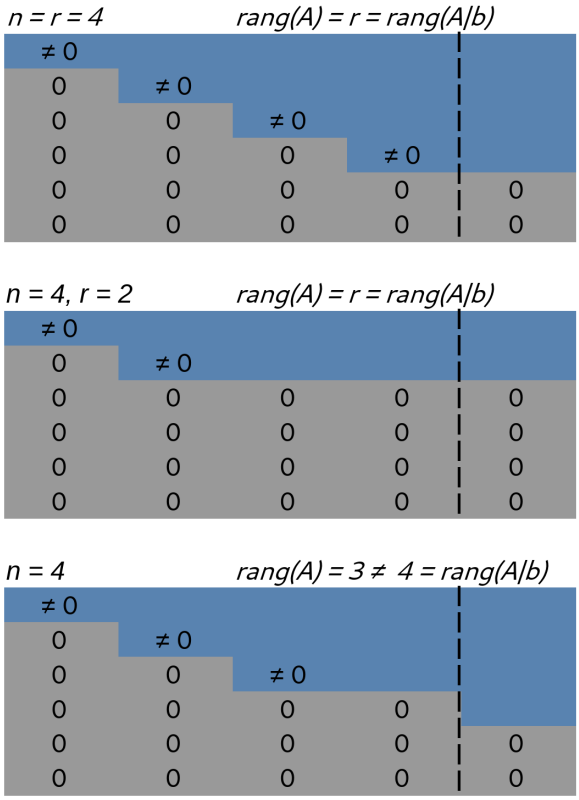


Abbildung 3.3.: Trapezformen

3.1.7. Python

Für die Lösung von LGS mit Python verwenden wir die Pakete

- NumPy, insbesondere das Modul numpy.linalg und von
- SciPy das Modul scipy.linalg.
- Für die grafische Darstellung verwenden wir das Paket matplotlib.

Die notwendigen Funktionen werden im Kapitel Python Tutorial eingeführt. Wir verwenden dabei die folgenden, üblichen Abkürzungen:

```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

Die zentralen Python-Funktionen sind:

- `@`: Operator der Matrixmultiplikation, z. B. `A@x`
- `np.linalg.matrix_rank(A)`: Gibt den Rang der Matrix A zurück.
- `np.linalg.solve(A, b)`: Ist nur für quadratische LGS $Ax = b$ anwendbar! Wenn die $n \times n$-Matrix A maximalen (man sagt: vollen) Rang n hat, liefert `solve` die eindeutige Lösung. Andernfalls liefert `solve` eine Fehlermeldung!

Das heißt, wir können in Python zwar mittels `matrix_rank` die Lösungsstruktur eines LGS bestimmen jedoch bisher nur bestimmte, quadratische LGS direkt lösen. Für die direkte Lösung von allgemeinen LGS in Python werden wir in den kommenden Kapiteln die notwendigen Methoden aber noch kennenlernen.

3.2. Beispiele

3.2.1. GEV eindeutige Lsg.

Wir betrachten das LGS

$$\begin{aligned} 4x_1 - x_2 - x_3 &= 6 \\ x_1 + 2x_3 &= 0 \\ -x_1 + 2x_2 + 2x_3 &= 2 \\ 3x_1 - x_2 &= 3 \end{aligned}$$

Die erweiterte Koeffizientenmatrix ist

$$\begin{pmatrix} 4 & -1 & -1 & 6 \\ 1 & 0 & 2 & 0 \\ -1 & 2 & 2 & 2 \\ 3 & -1 & 0 & 3 \end{pmatrix}$$

Wir vertauschen zuerst die erste und die zwei Zeile und bringen dann die Matrix in mehreren Schritten in die folgende Trapezform:

$$\begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & -1 & -9 & 6 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Der Rang der Koeffizientenmatrix und der erweiterten Koeffizientenmatrix ist 3. Die Lösungsmenge ist also 0-dimensional, d. h. es gibt genau eine Lösung. Diese kann durch Rückwärtseinsetzen bestimmt werden: Die dritte Zeile $-x_3 = 1$ ergibt $x_3 = -1$. Setzen wir dies in die zweite Zeile $-x_2 - 9x_3 = 6$ ein, erhalten wir $x_2 = 3$ Setzt man schließlich die Werte für x_3 und x_2 in die erste Zeile ein erhält man $x_1 = 2$.

Quelle: Papula Bd 2, Seite 81, Beispiel (2)

3.2.2. GEV keine Lsg.

Wir betrachten das LGS

$$\begin{aligned} -x_1 + 5x_2 &= 4 \\ 3x_1 - 5x_2 &= 8 \\ 5x_1 - 7x_2 &= -11 \end{aligned}$$

Die erweiterte Koeffizientenmatrix ist

$$\begin{pmatrix} -1 & 5 & 4 \\ 3 & -5 & 8 \\ 5 & -7 & -11 \end{pmatrix}$$

Diese kann in die folgende Trapezform gebracht werden:

$$\begin{pmatrix} -1 & 5 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & -3 \end{pmatrix}$$

Der Rang der erweiterten Koeffizientenmatrix ist 3, der Rang der Koeffizientenmatrix ist 2. Das LGS hat also keine Lösung.

3.2.3. GEV und Python

Wir lösen das folgende LGS in Python mittels GEV, bestimmen die Lösungsstruktur mit dem Rang und lösen das LGS, falls möglich, mit `np.linalg.solve`:

$$\begin{aligned} 4x_2 - 3x_3 &= 3 \\ -x_1 + 7x_2 - 5x_3 &= 4 \\ -x_1 + 8x_2 - 6x_3 &= 5 \end{aligned}$$

```
A = np.array([[ 0,  4, -3],
              [-1,  7, -5],
              [-1,  8, -6]])
b = np.array([[3],
              [4],
              [5]])
Ab = np.hstack((A, b))  # stack arrays horizontally, i.e., column wise
print('Koeffizientenmatrix A:')
print(A)
print('Rechte Seite b:')
print(b)
print('Erweiterte Koeffizientenmatrix (A|b):')
print(Ab)

# Vertausche 1. und 3. Zeile:
Ab_orig = Ab.copy()  # copy because we will change Ab and want to keep Ab_orig
Ab[0] = Ab_orig[2]
Ab[2] = Ab_orig[0]
print('Erweiterte Koeffizientenmatrix [A|b] nach Zeilentausch')
print(Ab)
```

```python
# Neue 2. Zeile = Z2 - Z1:
Ab[1] = Ab[1] - Ab[0]
print('Erweiterte Koeffizientenmatrix [A|b] nach 1. Zeilenaddition')
print(Ab)

# Neue 3. Zeile = Z3 + 4*Z2:
Ab[2] = Ab[2] + 4*Ab[1]
print('Erweiterte Koeffizientenmatrix [A|b] nach 2. Zeilenaddition')
print(Ab)

# Rücksubstitution:
x3 = Ab[2,3]/Ab[2,2]
x2 = 1/Ab[1,1]*(Ab[1,3] - Ab[1,2]*x3)
x1 = 1/Ab[0,0]*(Ab[0,3] - Ab[0,2]*x3 - Ab[0,1]*x2)
x = np.array([[x1], [x2], [x3]])
print(f"Lösung: x = \n{x}")

# Lösungsstruktur mit Rang:
rA   = np.linalg.matrix_rank(A)
rAb  = np.linalg.matrix_rank(np.hstack((A, b)))
print(f"Rang A = {rA} und Rang [A|b] = {rAb}")

# Lösung mit np.linalg.solve:
# Achtung: verwendbar, weil A quadratisch ist und vollen Rang hat.
sol  = np.linalg.solve(A,b)
print(f"Lösung mit np.linalg.solve: x = \n{sol}")
check = np.dot(A, x)
print(f"Check, dass Ax = \n{check} gleich b = \n{b}")
```

```
Koeffizientenmatrix A:
[[ 0  4 -3]
 [-1  7 -5]
 [-1  8 -6]]
Rechte Seite b:
[[3]
 [4]
 [5]]
Erweiterte Koeffizientenmatrix (A|b):
[[ 0  4 -3  3]
 [-1  7 -5  4]
 [-1  8 -6  5]]
Erweiterte Koeffizientenmatrix [A|b] nach Zeilentausch
[[-1  8 -6  5]
 [-1  7 -5  4]
 [ 0  4 -3  3]]
Erweiterte Koeffizientenmatrix [A|b] nach 1. Zeilenaddition
[[-1  8 -6  5]
 [ 0 -1  1 -1]
 [ 0  4 -3  3]]
Erweiterte Koeffizientenmatrix [A|b] nach 2. Zeilenaddition
[[-1  8 -6  5]
```

```
 [ 0 -1  1 -1]
 [ 0  0  1 -1]]
Lösung: x =
[[ 1.]
 [-0.]
 [-1.]]
Rang A = 3 und Rang [A|b] = 3
Lösung mit np.linalg.solve: x =
[[ 1.]
 [ 0.]
 [-1.]]
Check, dass Ax =
[[3.]
 [4.]
 [5.]] gleich b =
[[3]
 [4]
 [5]]
```

3.2.4. Unendlich viele Lsg.

Wir betrachten das LGS

$$\begin{aligned} x_1 - 3x_2 + x_3 &= 0 \\ -2x_1 + 5x_2 + 5x_3 &= -7 \end{aligned}$$

Die erweiterte Koeffizientenmatrix lautet

$$\begin{pmatrix} 1 & -3 & 1 & 0 \\ -2 & 5 & 5 & -7 \end{pmatrix}$$

Diese kann in die folgende Trapezform gebracht werden:

$$\begin{pmatrix} 1 & -3 & 1 & 0 \\ 0 & -1 & 7 & -7 \end{pmatrix}$$

Die erweiterte Koeffizientenmatrix hat den Rang 2, die Koeffizientenmatrix hat den Rang 2. Die Anzahl der Variablen ist 3. Das LGS hat also 3 - 2 = 1 frei wählbare Variable, nämlich x_3, und somit unendlich viele Lösungen:

$$\begin{aligned} x_2 &= 7 + 7x_3 \\ x_1 &= 3x_2 - x_3 = 21 + 21x_3 - x_3 = 21 + 20x_3 \end{aligned}$$

```python
A = np.array([[ 1,-3, 1],
              [-2, 0, 5]])
b = np.array([[ 0],
              [-7]])
Ab = np.hstack((A, b))
print("Rang von A  =", np.linalg.matrix_rank(A))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
print("Anzahl der Variablen =", np.shape(A)[1])
print("Daher: unendlich (1-dim.) viele Lösungen.")
```

```
Rang von A  = 2
Rang von Ab = 2
Anzahl der Variablen = 3
Daher: unendlich (1-dim.) viele Lösungen.
```

3.2.5. Keine Lösung

Wir betrachten das LGS

$$\begin{aligned} x_1 - 3x_2 &= 0 \\ 2x_1 + 2x_2 &= 7 \\ 5x_1 + x_2 &= 1 \end{aligned}$$

Die erweiterte Koeffizientenmatrix lautet

$$\begin{pmatrix} 1 & -3 & 0 \\ 2 & 2 & 7 \\ 5 & 1 & 1 \end{pmatrix}$$

Diese kann in die folgende Trapezform gebracht werden:

$$\begin{pmatrix} 1 & -3 & 0 \\ 0 & 8 & 7 \\ 0 & 0 & -13 \end{pmatrix}$$

Der Rang der erweiterten Koeffizientenmatrix ist 3, der Rang der Koeffizientenmatrix ist 2. Das LGS hat also keine Lösung.

```python
A = np.array([[ 1,-3],
              [ 2, 2],
              [ 5, 1]])
b = np.array([[0],
              [7],
              [1]])
Ab = np.hstack((A, b))
print("Rang von A  =", np.linalg.matrix_rank(A))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
print("Daher: keine Lösung.")
```

```
Rang von A  = 2
Rang von Ab = 3
Daher: keine Lösung.
```

3.2.6. Eindeutige Lösung

Wir betrachten das LGS

$$\begin{aligned} x_1 - 3x_2 &= 4 \\ -2x_1 &= -2 \\ 5x_1 + x_2 &= 4 \end{aligned}$$

Die erweiterte Koeffizientenmatrix lautet

$$\begin{pmatrix} 1 & -3 & 4 \\ -2 & 0 & -2 \\ 5 & 1 & 4 \end{pmatrix}$$

Diese kann in die folgende Trapezform gebracht werden:

$$\begin{pmatrix} 1 & -3 & 4 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Der Rang der erweiterten Koeffizientenmatrix ist 2, der Rang der Koeffizientenmatrix ist 2. Das LGS hat also genau eine Lösung, nämlich $x_2 = -1$ und $x_1 = 1$.

```
A = np.array([[ 1,-3],
              [-2, 0],
              [ 5, 1]])
b = np.array([[ 4],
              [-2],
              [ 4]])
Ab = np.hstack((A, b))
print("Rang von A  =", np.linalg.matrix_rank(A))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
print("Anzahl der Variablen =", np.shape(A)[1])
print("Daher: genau eine Lösung.")
```

```
Rang von A  = 2
Rang von Ab = 2
Anzahl der Variablen = 2
Daher: genau eine Lösung.
```

3.3. Aufgaben

3.3.1. Verständnisfragen und kurze Aufgaben

1. Wir betrachten das lineare Gleichungssystem

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 7 \\ 3 & 14 & 17 \end{pmatrix} x = \begin{pmatrix} 4 \\ 8 \\ 12 \end{pmatrix}.$$

   Bestimmen Sie anhand des GEV, wie viele Lösungen es für das LGS gibt.
2. Was ist der Rang einer Matrix, und wozu kann man ihn verwenden?
3. Wie bestimmen Sie, ob ein allgemeines LGS $Ax = b$ keine, eine oder unendlich viele Lösungen hat?
4. Gegeben sei das LGS $Ax = b$. Der Rang der Koeffizientenmatrix A sei geringer als der Rang der erweiterten Koeffizientenmatrix $(A|b)$, also $\operatorname{rank}(A) < \operatorname{rank}(A|b)$. Wie viele Lösungen hat das gegebene LGS?

Ergebnisse:

1. unendlich viele Lösungen
2. Siehe Lösungsverfahren.
3. Siehe Lösungsstruktur Teil 2.
4. keine Lösung

3.3.2. Modellierungsbeispiele

Rechnen Sie alle Beispiele aus dem Abschnitt Modellierung von Hand und in Python nach.

3.3.3. Vektorgleichung und lineares Gleichungssystem

Welchem linearen Gleichungssystem entspricht die Vektorgleichung

$$x \begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix} + y \begin{pmatrix} 4 \\ -2 \\ 4 \end{pmatrix} = \begin{pmatrix} -9 \\ 7 \\ 5 \end{pmatrix} ?$$

Bestimmen Sie seine Lösungsmenge mit dem GEV von Hand. Überprüfen Sie Ihr Ergebnis grafisch in der x-y-Ebene.

Ergebnis: Die Lösungsmenge ist die leere Menge. Die drei Geraden schneiden sich nicht in einem Punkt.

3.3.4. Gaußsches Eliminationsverfahren

Bestimmen Sie von Hand mit Hilfe des GEV, ob folgendes lineares Gleichungssystem nicht-triviale Lösungen hat.

$$\begin{aligned} 2x_1 - 5x_2 + 8x_3 &= 0 \\ -2x_1 - 7x_2 + x_3 &= 0 \\ 4x_1 + 2x_2 + 7x_3 &= 0 \end{aligned}$$

Überprüfen Sie Ihr Ergebnis am Computer.

Ergebnis: Das Gleichungssystem hat nicht-triviale Lösungen, nämlich alle Vielfachen von $x = \begin{pmatrix} -17 \\ 6 \\ 8 \end{pmatrix}$.

3.3.5. GEV und Python

Lösen Sie folgende LGS mittels des GEV von Hand. Geben Sie jeweils den Rang der (erweiterten) Koeffizientenmatrix an.

Beispiel A:

$$\begin{aligned} x_1 + x_2 + x_3 &= 1 \\ x_1 + 2x_2 + 2x_3 &= 1 \\ x_1 + 2x_2 + 3x_3 &= 1 \end{aligned}$$

Beispiel B:

$$\begin{aligned} -x_1 + 3x_2 - 2x_3 &= 1 \\ -x_1 + 4x_2 - 3x_3 &= 0 \\ -x_1 + 5x_2 - 4x_3 &= 0 \end{aligned}$$

Beispiel C:

$$\begin{aligned} -x_1 + 3x_2 - 2x_3 &= 4 \\ -x_1 + 4x_2 - 3x_3 &= 5 \\ -x_1 + 5x_2 - 4x_3 &= 6 \end{aligned}$$

Verifizieren Sie die Ergebnisse mittels Python.

Ergebnisse:

- Beispiel A: Rang von A ist 3, und Rang von $(A|b)$ ist 3. Das LGS hat die eindeutige Lösung $x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$.
- Beispiel B: Rang von A ist 2, und Rang von $(A|b)$ ist 3, LGS hat keine Lösung.
- Beispiel C: Rang von A ist 2, und Rang von $(A|b)$ ist 3, LGS hat unendlich viele Lösungen. x_3 ist frei wählbar, $x_2 = 1 + x_3$ und $x_1 = -1 + x_3$.

3.3.6. Existenz und Eindeutigkeit

Gegeben ist das lineare Gleichungssystem:

$$\begin{aligned} 2x_1 - 4x_2 - 2x_3 &= b_1 \\ -5x_1 + x_2 + x_3 &= b_2 \\ 7x_1 - 5x_2 - 3x_3 &= b_3 \end{aligned}$$

Hat das lineare Gleichungssystem für alle Vektoren b eine eindeutige Lösung? Lösen Sie dieses Problem am Computer.

Ergebnis: Das lineare Gleichungssystem $Ax = b$ hat nicht für alle Vektoren b eine eindeutige Lösung, denn der Rang von A ist nur 2 und somit nicht voll.

3.3.7. Rang

Bestimmen Sie von Hand mittels elementarer Umformungen den Rang von

$$A = \begin{pmatrix} 1 & 0 & -1 & 1 \\ 2 & 3 & 1 & 5 \\ 3 & 2 & -1 & 6 \\ 0 & 5 & 5 & 5 \end{pmatrix},$$

und überprüfen Sie das Ergebnis am Computer.

Ergebnis: Rang von A ist 3.

3.3.8. Lösbarkeit

Zeigen Sie von Hand, dass das folgende lineare Gleichungssystem nicht lösbar ist, und überprüfen Sie das Ergebnis am Computer.

$$\begin{aligned} x_1 + x_2 - x_3 &= 2 \\ -2x_1 + x_3 &= -2 \\ 5x_1 - x_2 + 2x_3 &= 4 \\ 2x_1 + 6x_2 - 3x_3 &= 5 \end{aligned}$$

Ergebnis: Rang der Koeffizientenmatrix ist 3, der Rang der erweiterten Koeffizientenmatrix ist 4.

3.3.9. Elektrischer Schaltkreis

Der elektrische Schaltkreis in Abbildung Abbildung 3.4 ist parametrisiert durch: $U_1 = 230$ V, $U_2 = 370$ V, $R_1 = 200$ Ohm, $R_2 = 100$ Ohm und $R_3 = 300$ Ohm.

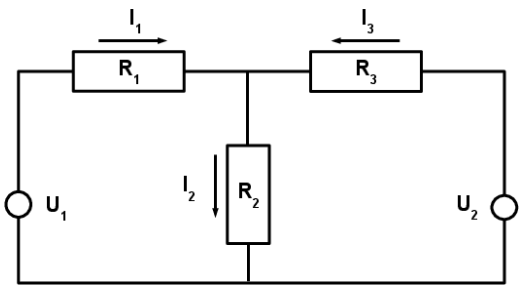


Abbildung 3.4.: Elektrischer Schaltkreis

1. Formulieren Sie das lineare Gleichungssystem für die Stromstärken I_1, I_2, und I_3 in der Form $Ax = b$.
2. Lösen Sie $Ax = b$ händisch.
3. Bestimmen Sie in Python die Lösungsstruktur, und lösen Sie, falls möglich, das LGS in Python mit dem Befehl `np.linalg.solve`.

Ergebnis: $I_1 = 0.5$ A, $I_1 = 1.3$ A, $I_3 = 0.8$ A

3.3.10. Parabel durch drei Punkte

Stellen Sie ein LGS zur Bestimmung der Koeffizienten α, β, γ der Parabel (=quadratische Funktion)

$$y(x) = \alpha + \beta x + \gamma x^2$$

auf, welche durch die Punkte $(1, 1), (2, 2)$ und $(3, 0)$ geht. Lösen Sie anschließend das LGS, und stellen Sie die Lösung grafisch dar.

Ergebnis: $y(x) = -3 + 5.5x - 1.5x^2$.

3.3.11. Diät

Quelle: Lay, David; Lay, Steven; McDonald, Judi (2021): Linear Algebra and Its Applications. 6th edition, Pearson Education Limited.

Die Formel für die Cambridge-Diät, eine beliebte Diät in den 1980er Jahren, basiert auf jahrelanger Forschung. Um die gewünschten Mengen und Verhältnisse der Nährstoffe zu erreichen, musste man eine Vielfalt von Lebensmitteln einbeziehen. Jedes Lebensmittel liefert mehrere der erforderlichen Inhaltsstoffe, jedoch nicht in den genau richtigen Anteilen. So war zum Beispiel fettfreie Milch eine wichtige Eiweißquelle, enthielt aber zu viel Kalzium. Daher wurde Sojamehl für einen Teil des Eiweißes verwendet, da Sojamehl wenig Kalzium enthält. Allerdings enthält Sojamehl zu viel Fett, so dass Molke hinzugefügt wurde, da sie im Verhältnis zum Kalzium weniger Fett liefert. Leider enthält die Molke zu viele Kohlenhydrate … Das folgende Beispiel veranschaulicht das Problem in kleinem Maßstab. In der Tabelle Tabelle 3.1 sind drei der Bestandteile der Diät aufgeführt, zusammen mit den Mengen bestimmter Nährstoffe.

Tabelle 3.1.: Enthaltene Nährstoffmenge (g) pro 100 g der Zutat

Nährstoff	fettfreie Milch	Sojamehl	Molke	Geforderte Nährstoffmenge (g)
Eiweiß	36	51	13	33
Kohlenhydrate	52	34	74	45
Fett	0	7	1.1	3

Wenn möglich, finden Sie eine Kombination aus fettfreier Milch, Sojamehl und Molke, um die geforderten Mengen an Eiweiß, Kohlenhydraten und Fett zu erhalten.

Ergebnis: Die Diät erfordert 27.7 g fettfreie Milch, 39.2 Einheiten Sojamehl und 23.3 Einheiten Molke.

3.3.12. Lösbarkeit

1. Geben Sie ein Zahlenbeispiel für ein lineares Gleichungssystem in Matrixform an, das 2 Gleichungen, 3 Variablen und keine Lösung hat.
2. Geben Sie ein Zahlenbeispiel für ein lineares Gleichungssystem in Matrixform an, das mindestens 2 Gleichungen, mindestens 2 Variablen und einen 1-dimensionalen Lösungsraum hat.

Ergebnis:

1. zwei parallele, aber nicht gleiche Ebenen im Raum
2. zwei gleiche Geraden in der Ebene

3.3.13. Unter- und überstimmte LGS

Lösen Sie von Hand folgende LGS, und geben Sie jeweils den Rang der (erweiterten) Koeffizientenmatrix an:

Beispiel A:

$$\begin{aligned} x_1 + 2x_2 + x_3 + 2x_4 &= 0 \\ 2x_1 + 4x_2 + x_3 + 3x_4 &= 0 \\ 3x_1 + 6x_2 + x_3 + 4x_4 &= 0 \end{aligned}$$

Beispiel B:

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 0 \\ 4x_1 + 2x_2 + x_3 &= 0 \\ 6x_1 + 3x_2 + x_3 &= 0 \\ 8x_1 + 5x_2 + x_3 &= 0 \end{aligned}$$

Beispiel C:

$$\begin{aligned} x_1 + 2x_2 + x_3 + 2x_4 &= 3 \\ 2x_1 + 4x_2 + x_3 + 3x_4 &= 4 \\ 3x_1 + 6x_2 + x_3 + 4x_4 &= 5 \end{aligned}$$

Beispiel D:

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 2 \\ 4x_1 + 2x_2 + x_3 &= 5 \\ 6x_1 + 3x_2 + x_3 &= 8 \\ 8x_1 + 5x_2 + x_3 &= 8 \end{aligned}$$

Verifizieren Sie Ihre händischen Ergebnisse mit Python.

Ergebnisse:

- Beispiel A: x_4 ist frei wählbar, $x_3 = -x_4$, x_2 ist auch frei wählbar, $x_1 = -2x_2 - x_4$.
- Beispiel B: Es gibt nur die triviale Lösung.
- Beispiel C: Lösung aus Beispiel A ist verschoben um $\begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \end{pmatrix}$.
- Beispiel D: $x_3 = -1$, $x_2 = -3$, $x_1 = 3$.

3.3.14. Ill-Conditioned Linear System

Lösen Sie in Python die beiden LGS $Ax = b$

$$\begin{aligned} 0.835x_1 + 0.667x_2 &= 0.168 \\ 0.333x_1 + 0.266x_2 &= 0.067 \end{aligned}$$

und

$$\begin{aligned} 0.835x_1 + 0.667x_2 &= 0.168 \\ 0.333x_1 + 0.266x_2 &= 0.066 \end{aligned}$$

die sich nur im Wert b_2 unterscheiden. Erklären Sie den großen Unterschied in den Lösungen. Eine Koeffizientenmatrix wie in diesem Beispiel heißt *ill-conditioned* und das LGS heißt *ill-conditioned system*.

Ergebnis: Lösung des ersten LGS: $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$, Lösung des zweiten LGS: $\begin{pmatrix} -665.99999998 \\ 833.99999998 \end{pmatrix}$. Die Gleichungen, die zu jeder Zeile gehören, entsprechen fast parallelen Geraden, deren Schnittpunkt durch eine kleine Änderung in b stark beeinflusst wird.

3.3.15. LGS mit Parameter

Bestimmen Sie alle jene Werte für α im LGS

$$\begin{aligned} 2x_1 + 2x_2 + 3x_3 &= 0 \\ 4x_1 + 8x_2 + 12x_3 &= -4 \\ 6x_1 + 2x_2 + \alpha x_3 &= 4 \end{aligned}$$

sodass

1. eine eindeutige Lösung existiert,
2. unendliche viele Lösungen existieren.

Gibt es Werte für α, bei denen das LGS keine Lösung hat?

Ergebnis: $\alpha = 3$ führt auf unendliche Lösungen, $\alpha \neq 3$ führt auf eine eindeutige Lösung. Es existiert immer eine Lösung.

3.3.16. Theoriefragen

1. Bisher wurden nur Änderungen eines LGS durch Operationen an den Zeilen der erweiterten Koeffizientenmatrix betrachtet. Es ist jedoch auch möglich, äquivalente Operationen (Vertauschen, Multiplikation mit einem Skalar, Addition) mit den Spalten durchzuführen. Welche Auswirkung haben die Spaltenoperationen auf die Unbekannten (=Variablen)?
2. Die Lösung eines lösbaren LGS mit 4 Gleichungen und 8 Unbekannten enthält 5 freie Variablen. Wie groß ist der Rang der Koeffizientenmatrix?
3. Welche Struktur hat die Lösungsmenge eines homogenen LGS mit mehr Unbekannten als Gleichungen?
4. Wie lässt sich überprüfen, ob ein LGS $Ax = b$ ill-conditioned ist? Was bedeutet ill-conditioned für die Lösung des LGS?
5. Wie viele unterschiedliche Trapezformen gibt es für eine 3×4 Matrix?
6. Der Rang der $m \times n$ Matrix A sei m. Was bedeutet dies für die Lösungsmenge des LGS $Ax = b$?

Ergebnis:

1. Vertauschen von Spalten entspricht dem Vertauschen der Reihenfolge der Unbekannten. Multiplikation mit einem Skalar ändert die ``Einheit'' der Unbekannten. Addition zweier Spalten ändert die Repräsentation der Unbekannten, vgl. Basiswechsel.
2. 3
3. unendlich viele Lösungen
4. Überprüfung, ob kleine Änderung von Elementen im Vektor b zu großen Änderungen in der Lösung führen. Z. B. problematisch, wenn Elemente in b aus Daten *geschätzt* werden.
5. 15
6. Die erweiterte Koeffizientenmatrix hat auch den Rang m. Das LGS hat entweder genau eine (für $m = n$) oder unendlich viele (für $m < n$) Lösungen. Der Fall $m > n$ ist nicht möglich, da die Koeffizientenmatrix dann nicht den Rang m haben würde.

3.3.17. Lösungsmenge

1. Untersuchen Sie von Hand und am Computer die Lösungsstruktur des linearen Gleichungssystems (d. h. ohne es zu lösen).
2. Bestimmen Sie anschließend im Falle der Lösbarkeit von Hand und am Computer sämtliche Lösungen.

$$\begin{pmatrix} 1 & 2 & -3 \\ 0 & 1 & -1 \\ 2 & 9 & 11 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 8 \\ 50 \end{pmatrix}$$

Ergebnis:

1. Die Koeffizientenmatrix hat den Rang 3, die erweiterte Koeffizientenmatrix hat den Rang 3. Alternativ: Die Koeffizientenmatrix hat eine nicht-Null-Determinante.
2. $x_1 = -11$, $x_2 = 8$, $x_3 = 0$.

3.3.18. Lösungsmenge

Bestimmen Sie von Hand die Lösungsmenge des LGS

$$\begin{pmatrix} -2 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Ergebnis: Das LGS hat die unendlich viele Lösungen $x_1 = x_2 = x_3$ mit x_3 als freien Parameter.

4. Matrizenrechnung

4.1. Methoden

Motivation

Matrizen sind ein Schlüsselkonzept der linearen Algebra und tauchen in fast allen Gebieten der Mathematik auf. Zudem lassen sich Rechnungen mit Matrizen sehr effizient auf Computern durchführen.
Typische Anwendungen: lineare Gleichungssysteme, lineare Abbildungen, Eigenwertprobleme, Optimierung, Datenanalyse, Bildverarbeitung, Künstliche Intelligenz, …

4.1.1. Definitionen

Eine Matrix ist ein rechteckige Anordnung (Tabelle) von Zahlen. Eine $m \times n$-Matrix A hat m Zeilen und n Spalten. Man sagt: Die Matrix hat die Dimension $m \times n$. Oder: Sie ist vom Typ $m \times n$. Die Einträge der Matrix A sind die Zahlen A_{ij}, die in der i-ten Zeile und j-ten Spalte stehen. Die Menge aller $m \times n$-Matrizen wird mit $\mathbb{R}^{m \times n}$ bezeichnet.

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}$$

- Wenn $m = n$, dann heißt die Matrix A quadratisch.
- Wenn $m = 1$, dann heißt die Matrix A Zeilenvektor.
- Wenn $n = 1$, dann heißt die Matrix A Spaltenvektor.
- Wenn $m = n = 1$, dann heißt die Matrix A Zahl oder Skalar.

Gleichheit von Matrizen: Zwei Matrizen sind genau dann gleich, wenn sie dieselben Dimensionen haben und alle ihre Einträge gleich sind. Vergleiche die Gleichheit von Vektoren.

Visualisierung:

- Zahl: Punkt
- Vektor: Strich, vertikal für Spaltenvektor, horizontal für Zeilenvektor
- Matrix: Rechteck

Spalten- und Zeilenansicht von Matrizen: Eine Matrix kann auf folgende Arten gesehen werden:

- als Stapel von Zeilenvektor
- als Stapel von Spaltenvektor
- als 2-dimensionale Anordnung von Zahlen

4.1.2. Rechenoperationen

4.1.2.1. Addition, Skalarmultiplikation, Transposition

- Die **Addition** von Matrizen ist nur für Matrizen mit gleichen Dimensionen definiert und erfolgt dann elementweise, wie bei Vektoren.
- Die **Skalarmultiplikation** einer Matrix mit einer Zahl (=Skalar) erfolgt, auch wie bei Vektoren, elementweise.
- Das **Transponieren** einer Matrix vertauscht die Spalten und Zeilen der Matrix und macht eine $n \times m$-Matrix A zur $m \times n$-Matrix A^T. Die Einträge der transponierten Matrix sind $A^T_{ij} = A_{ji}$. Die transponierte Matrix entsteht durch Spiegelung der Ausgangsmatrix an ihrer Diagonalen (=Elemente mit gleichem Zeilen- und Spaltenindex).

4.1.2.2. Matrixmultiplikation

Die Matrixmultiplikation einer $n \times m$-Matrix mit einer $m \times p$-Matrix liefert eine $n \times p$-Matrix. Die innere Dimension muß übereinstimmen, hier ist sie m. Das Ergebnis hat die äußeren Dimensionen, hier n und p. Die Ergebnismatrix $C = AB$ der Matrixmultiplikation von A mit B hat die Einträge $C_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}$. Das heißt, der Eintrag in der i-ten Zeile und j-ten Spalte von C ist das Skalarprodukt des i-ten Zeilenvektors von A mit dem j-ten Spaltenvektor von B. Hier ein Beispiel für eine 2×3-Matrix A und eine 3×2-Matrix B:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix} =$$
$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} \\ A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} & A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} \end{pmatrix}$$

Ansichten der Matrixmultiplikation:

- Tabellenansicht: Die Einträge der Ergebnismatrix AB sind die inneren Produkte der Zeilenvektoren von A mit den Spaltenvektoren von B.
- Spaltenansicht: Die k-te Spalte von AB ist die Linearkombinationen der Spalten von A mit Koeffizienten aus der k-ten Spalte von B.
- etc.

Produkttypen:

- Zeilenvektor $\times$ Spaltenvektor = Zahl: Das ist gerade das **innere Produkt** von Vektoren $a, b \in \mathbb{R}^n$, die in der Matrizenrechnung als Spaltenvektoren geschrieben werden:

$$a^T b = \begin{pmatrix} a_1 & \dots a_n \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n.$$

- Matrix $\times$ Spaltenvektor = Spaltenvektor
- Zeilenvektor $\times$ Matrix = Zeilenvektor
- Matrix $\times$ Matrix = Matrix

4.1.2.3. Spezielle Matrizen

- **Nullmatrizen** enthalten lauter Nullen. Man schreibt kurz 0, so wie man es auch für Nullvektoren macht. Die Mullmatrix entsprechender Dimension ist das neutrale Element der Addition: $A + 0 = A$.
- **Einheitsmatrizen** sind $n \times n$-Matrizen, also quadratisch, die Einser auf der Diagonalen und ansonsten Nullen haben. Notation I (engl. identity matrix) oder $\mathbb{1}$. Die Einheitsmatrix entsprechender Dimension ist das neutrale Element der Multiplikation: $AI = IA = A$.
- Eine **Diagonalmatrix** ist eine quadratische Matrix, deren Einträge außerhalb der Diagonalen alle Null sind. Die Diagonalelemente können beliebige Zahlen sein.
- **Einsermatrizen** enthalten nur Einser als Elemente.
- Wenn $A^T = A$ gilt, dann heißt die Matrix A **symmetrisch**.
- Wenn $A^T = -A$ gilt, dann heißt die Matrix A **schiefsymmetrisch**.
- Eine obere bzw. untere **Dreiecksmatrix** ist eine quadratische Matrix, deren Einträge unterhalb bzw. oberhalb der Diagonalen alle Null sind.
- Es gibt noch viele weitere spezielle Matrizen, siehe z. B. Wikipedia.

4.1.3. Quadratische Matrizen

4.1.3.1. Determinante

Wir betrachten zuerst das allgemeine, quadratische 2×2-LGS

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 &= b_1 \\ A_{21}x_1 + A_{22}x_2 &= b_2 \end{aligned}$$

das die folgende Matrix-Vektor-Form $Ax = b$ hat:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

Die Koeffizientenmatrix A ist eine quadratische 2×2-Matrix. Mit dem Gaußschen Eliminationsverfahren erhält man die Lösungsformel:

$$\begin{aligned} x_1 &= \frac{b_1 A_{22} - b_2 A_{12}}{A_{11}A_{22} - A_{12}A_{21}} \\ x_2 &= \frac{b_2 A_{11} - b_1 A_{21}}{A_{11}A_{22} - A_{12}A_{21}} \end{aligned}$$

Schlussfolgerungen:

- Falls der Nenner $A_{11}A_{22} - A_{12}A_{21} \neq 0$, dann hat das LGS genau eine Lösung.
- Falls der Nenner $A_{11}A_{22} - A_{12}A_{21} = 0$, dann hat das LGS keine Lösung oder unendlich viele Lösungen.

Die Zahl $A_{11}A_{22} - A_{12}A_{21}$, die die Fallunterteilung bestimmt (determiniert), heißt Determinante der Koeffizientenmatrix A und wird mit $\det(A)$ oder $|A|$ notiert. Die Determinante kann nur für quadratische Matrizen (gleich viele Zeilen wie Spalten) berechnet werden kann.

Beispiel: $A = \begin{pmatrix} 3 & 2 \\ -1 & 4 \end{pmatrix}$ hat die Determinante $\det(A) = 3 \cdot 4 - (-1) \cdot 2 = 14$. Vergleiche das Kreuzprodukt $a \times b$ der zwei räumlichen Vektoren $a = \begin{pmatrix} 3 \\ -1 \\ 0 \end{pmatrix}$ und $b = \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix}$, dessen Länge die Fläche des von a und b aufgespannten Parallelogramms ist: $a \times b = \begin{pmatrix} 0 \\ 0 \\ 14 \end{pmatrix}$, $\|a \times b\| = \sqrt{14^2} = 14$.

Geometrie: Der Betrag der Determinante ist gleich dem Flächeninhalt (allg. dem Volumen) des von den Spaltenvektoren der Matrix aufgespannten Parallelepipeds.

Formeln:

- 2×2-Matrizen: $\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - cb$.
- 3×3-Matrizen:

$$\det \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} =$$
$$A_{11}A_{22}A_{33} + A_{12}A_{23}A_{31} + A_{13}A_{21}A_{32}$$
$$-A_{13}A_{22}A_{31} - A_{12}A_{21}A_{33} - A_{11}A_{23}A_{32}$$

- Für höhere Dimensionen gibt es Formeln, die wir hier aber nicht angeben.

4.1.3.2. Inverse Matrix

Zu einer quadratischen Matrix A kann (muss aber nicht) eine inverse Matrix, geschrieben als A^{-1}, existieren. Sie entspricht dem Kehrwert für Matrizen und hat die definierenden Eigenschaften: $A^{-1}A = I = AA^{-1}$. Falls eine inverse Matrix existiert, dann ist sie eindeutig. Für eine 2×2-Matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ ist die inverse Matrix, falls $\det(A) \neq 0$, durch $A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$ gegeben. Eine quadratische Matrix, die eine Inverse besitzt, heißt reguläre, invertierbare oder nicht singuläre Matrix, andernfalls heißt sie singuläre Matrix.

Falls für ein quadratisches LGS $Ax = b$ die inverse Matrix A^{-1} existiert, liefert die Multiplikation von $Ax = b$ von links mit A^{-1} die eindeutige Lösung $x = A^{-1}b$. Die inverse Matrix von A existiert genau dann, wenn die Determinante von A nicht Null ist. Falls A^{-1} nicht existiert, also $\det(A) = 0$, dann gibt es ein $\tilde{b}$, sodass $Ax = \tilde{b}$ keine Lösung hat, und falls eine Lösung zu $Ax = b$ existiert, dann ist sie nicht eindeutig.

4.1.3.3. Lösungsstruktur quadratischer LGS

Wir betrachten ein quadratisches $n \times n$-LGS $Ax = b$ mit n Gleichungen und n Variablen. Dann gilt:

- Falls $\det(A) \neq 0$, dann hat das LGS genau eine Lösung.
- Falls $\det(A) = 0$, dann hat das LGS keine Lösung oder unendlich viele Lösungen. Die Unterscheidung kann mit den Rängen von A und $(A|b)$ getroffen werden.

Für ein homogenes quadratisches LGS $Ax = 0$ gilt zudem:

- Falls $\det(A) \neq 0$, dann hat das LGS nur die triviale Lösung $x = 0$.

- Falls $\det(A) = 0$, dann hat das LGS unendlich viele Lösungen. Die Lösungsmenge ist $n - \text{rank}(A)$-dimensional.

4.1.4. Rechenregeln

Notation: Zahl α, Matrizen A, B und C

- $\alpha(A + B) = \alpha A + \alpha B$
- $\alpha(AB) = (\alpha A)B = A(\alpha B) = \alpha AB$
- $C(A + B) = CA + CB$
- $(A + B)C = AC + BC$
- *Achtung:* Im Allgemeinen ist AB nicht gleich BA!
- $A(BC) = (AB)C = ABC$
- $(A \pm B)^T = A^T \pm B^T$
- $(\alpha A)^T = \alpha A^T$
- $(AB)^T = B^T A^T$ *Achtung:* Reihenfolge vertauscht!
- $(AB)^{-1} = B^{-1}A^{-1}$ *Achtung:* Reihenfolge vertauscht!
- $(A^{-1})^{-1} = A$
- $(A^{-1})^T = (A^T)^{-1} = A^{-T}$
- $\det(A) \neq 0 \Leftrightarrow A^{-1}$ existiert, d. h. A ist regulär und invertierbar.
- $\det(A) = 0 \Leftrightarrow A^{-1}$ existiert nicht, d. h. A ist singulär und nicht invertierbar.
- $\det(A^T) = \det(A)$
- $\det(AB) = \det(A)\det(B)$
- $\det(I) = 1$
- $\det(A^{-1}) = \det(A)^{-1}$
- Die Determinante einer Dreiecksmatrix ist das Produkt ihrer Diagonalelemente.
- Eine quadratische $n \times n$ Matrix A ist regulär. $\Leftrightarrow$ Der Rang von A ist voll, d. h. gleich n. $\Leftrightarrow$ Die Spaltenvektoren von A sind linear unabhängig und spannen den ganzen $\mathbb{R}^n$ auf.

4.1.5. Lineare Abbildungen

4.1.5.1. Definition

Eine lineare Abbildung (=Funktion) f zwischen den Vektorräumen $\mathbb{R}^n \to \mathbb{R}^m$ ordnet jedem Inputvektor x des Inputraums $\mathbb{R}^n$ genau einen Outputvektor $f(x)$ des Outputraums $\mathbb{R}^m$ derart zu, sodass die Linearitätseigenschaft

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

für alle Zahlen α und β und alle Inputvektoren x und y erfüllt ist.

4.1.5.2. Eigenschaften

- Jede lineare Funktion lässt sich in Matrixform $f(x) = Ax$ für eine eindeutige $m \times n$-Matrix A schreiben.
- Jede Funktion $f(x)$, die sich in Matrixform $f(x) = Ax$ schreiben lässt, ist linear.

Aufgrund dieser Eigenschaften kann eine lineare Abbildungen mit ihrer Matrix, genannt Abbildungsmatrix, gleichgesetzt werden.

4.1.5.3. Beispiele

- Eine skalare, lineare Funktion von $\mathbb{R}^n$ nach $\mathbb{R}$ ist gegeben durch das innere Produkt eines fixen Vektors a mit einem Variablenvektor x:

$$f(x) = a^T x.$$

  Die Konturlinien $a^T x = c$ mit $c \in \mathbb{R}$ sind in der Ebene ($n = 2$) parallele Geraden. Die Null-Konturlinie $a^T x = 0$ geht durch den Ursprung. Die Konturflächen $a^T x = c$ sind im Raum ($n = 3$) parallele Ebenen, und die Null-Konturfläche $a^T x = 0$ geht durch den Ursprung. Der Koeffizientenvektor a ist orthogonal (=rechtwinklig, normal) zu den Konturlinien bzw. Konturflächen.
- Der Drehung in der Ebene um den Winkel α gegen den Uhrzeigersinn entspricht die Drehmatrix $R = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$.
- Der Spiegelung in der Ebene an der 1-Achse entspricht die Matrix $S = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.
- Die Skalierung (Streckung bzw. Stauchung) in der Ebene um den Faktor a in x_1-Richtung und den Faktor b in x_2-Richtung entspricht die Matrix $D = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$.
- Die Projektion in der Ebene auf die 1-Achse entspricht die Matrix $P = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$. Für jede Projektion gilt: P ist symmetrisch und $P^2 = P$.
- Die Projektion in der Ebene auf die 2-Achse entspricht die Matrix $P = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$.
- Die Projektion in der Ebene auf die Gerade durch den Ursprung mit Richtungsvektor v entspricht die Matrix $P = \frac{1}{v^T v} v v^T$.

4.1.5.4. Darstellungen von Vektoren

Eine Basis des Vektorraumes $\mathbb{R}^n$ ist eine Liste von n Vektoren $b_i, i = 1, \dots n$ des $\mathbb{R}^n$, mit der sich jeder Vektor x des Vektorraumes eindeutig als Linearkombination der Basisvektoren b_i darstellen lässt:

$$x = \sum_{i=1}^{n} c_i b_i.$$

Die Koeffizienten c_i heißen Koordinaten von x bezüglich der Basis b_i. Schreibt man die Basisvektoren als Spalten in einen Matrix B und die Koordinaten als Spalten in einen Vektor c, dann gilt

$$x = Bc.$$

Die Matrix B ist regulär, da die Basisvektoren linear unabhängig sind und den ganzen $\mathbb{R}^n$ aufspannen. Die Standardbasis des $\mathbb{R}^n$ besteht aus den Vektoren e_i mit 1 an der i-ten Stelle und sonst Nullen. Bezüglich der Standardbasis sind die Koordinaten eines Vektors x die Einträge vom x, also $x = \sum_{i=1}^{n} x_i e_i$. Die Matrix der Standardbasisvektoren ist die Einheitsmatrix I, und es gilt $x = Ix$. Beispiel für $n = 2$: $x = x_1 e_1 + x_2 e_2 = x_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$.

Wenn man Vektoren mittels zwei verschiedenen Basen beschreibt, spricht man von einer Koordinatentransformation:

$$x = Bc = \tilde{B}\tilde{c}$$

Durch Multiplikation der letzten Gleichung mit der inversen Matrix B^{-1} bzw. $\tilde{B}^{-1}$ erhalten wir Formeln für die Umrechnung der Koordinaten: $c = B^{-1}\tilde{B}\tilde{c}$ und $\tilde{c} = \tilde{B}^{-1}Bc$.

Oft ist es sinnvoll, eine Basis zu wählen, die dem betrachteten Problem angepasst ist, siehe Kapitel Eigenwerte und -vektoren. Wir betrachten eine lineare Abbildung $y = Ax$ und stellen x und y in anderen Basen dar: $x = Bc$ und $y = Cd$. Dann erhält man durch Einsetzen:

$$Cd = ABc \iff d = C^{-1}ABc.$$

Bezüglich den Koordinaten c und d ist die Abbildung also durch die Matrix $C^{-1}AB$ gegeben.

4.1.5.5. Darstellung des Outputs

- Der Outputvektor Ax einer linearen Abbildung A kann als Linearkombination der Spalten von A dargestellt werden. Die Spalten von A sind die Bilder der Standardbasisvektoren e_i. Beispiel: $Ax = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_1 \begin{pmatrix} 1 \\ 3 \end{pmatrix} + x_2 \begin{pmatrix} 2 \\ 4 \end{pmatrix}$.
- Der Outputvektor Ax einer linearen Abbildung A kann auch als Spaltenvektor der durch die Zeilenvektoren von A definierten linearen Abbildungen dargestellt werden. Beispiel: $Ax = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1x_1 + 2x_2 \\ 3x_1 + 4x_2 \end{pmatrix}$.

Wenn man den Output von mehreren Inputvektoren unter einmal berechnen möchte, schreibt man die Inputvektoren als Spalten in eine Matrix X und erhält den Output als AX. Die Spalten von AX sind die Bilder der Spalten von X.

4.1.5.6. Hintereinanderausführung

Wenn $f(x) = Ax$ und $g(y) = By$, dann entspricht der Hintereinanderausführung der linearen Abbildungen $g \circ f$ das Matrixprodukt BA, also $g(f(x)) = BAx$.

Beispiel: Drehen eines Punktes der Ebene zuerst um den Winkel α und anschließend Spiegeln an der 1-Achse entspricht dem Matrixprodukt SR. Achtung: Die andere Reihenfolge liefert nicht dasselbe!

4.1.5.7. Bild, Kern, Rangsatz und LGS

Ein LGS $Ax = b$ kann mithilfe der linearen Abbildung A so interpretiert werden: Gesucht werden alle Inputvektoren x, die via A auf den Outputvektor b abgebildet werden.

Die Menge der Outputvektoren einer linearen Abbildung mit $m \times n$-Matrix A heißt Bild (engl. image, range) von A und wird mit $\mathrm{im}(A)$ bezeichnet. Die Dimension des Bildes ist der Rang der Matrix A:

$$\mathrm{rank}(A) = \dim(\mathrm{im}(A))$$

Ein LGS ist genau dann lösbar, wenn b im Bild von A liegt. Die Lösungsmenge des LGS $Ax = b$ ist dann die Menge aller Inputvektoren x, die auf b abgebildet werden. Wenn der Rang von A gleich der Dimension m des Outputraums ist, dann ist das Bild von A der ganze Outputraum, und das LGS hat für jeden Outputvektor b eine Lösung.

Die Menge der Inputvektoren, die auf den Nullvektor abgebildet werden, ist die Lösungsmenge des homogenen LGS $Ax = 0$ und heißt Kern oder Nullraum (engl. null space) der Matrix A und wird mit $\ker(A)$ bezeichnet. Falls ein LGS $Ax = b$ eine Lösung x_0 hat, kann man zu x_0 jeden Vektor x_h des Nullraums addieren und erhält wieder eine Lösung, denn $A(x_0 + x_h) = Ax_0 + Ax_h = b + 0 = b$.

Die Lösung eines LGS ist also genau dann eindeutig, wenn der Nullraum nur den Nullvektor enthält. Der Nullvektor ist immer enthalten.

Der Rangsatz besagt, dass für jede $m \times n$-Matrix die Dimension des Nullraums plus die Dimension des Bilds gleich die Dimension n des Inputraums ist:

$$\dim(\ker(A)) + \dim(\operatorname{im}(A)) = n.$$

Der Rangsatz kann als Erhaltung der Dimensionen/Freiheitsgrade interpretiert werden: Die Dimension des Inputraums wird auf die Dimension des Bildes und des Nullraums aufgeteilt. Wenn der Rang von A gleich der Dimension n des Inputraums ist, also $\dim(\operatorname{im}(A)) = n$, dann ist wegen des Rangsatzes der Nullraum 0-dimensional und besteht daher nur aus dem Nullvektor. Daher ist eine Lösung des LGS unter diesen Umständen eindeutig.

4.1.6. Python

Die Beispiele und Aufgaben verwenden folgende Python-Bibliotheken, siehe Kapitel Python Tutorial:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

Die zentralen Python-Funktionen sind:

- Für die Matrixmultiplikation kann der Operator `@` verwendet werden
- Die Matrixtransponierte kann mit der Methode `.T` erzeugt werden.
- `np.zeros` erzeugt Nullmatrizen
- `np.eye` erzeugt die Einheitsmatrizen
- `np.ones` Matrizen mit lauter Einsen als Einträgen.
- `np.diag` zum Extrahieren der Diagonale einer Matrix und zum Erzeugen einer Diagonalmatrix
- `np.linalg.det(A)`: Gibt die Determinante der Matrix A zurück.
- `np.linalg.inv` berechnet die inverse Matrix.
- `sp.linalg.null_space(A)` berechnet eine Orthonormalbasis des Nullraums (Kerns) der Matrix A. Die Spalten der zurückgegebenen Matrix sind die Orthonormalbasisvektoren.
- `sp.linalg.orth(A)` berechnet eine Orthonormalbasis des Bildes der Matrix A. Die Spalten der zurückgegebenen Matrix sind die Orthonormalbasisvektoren.

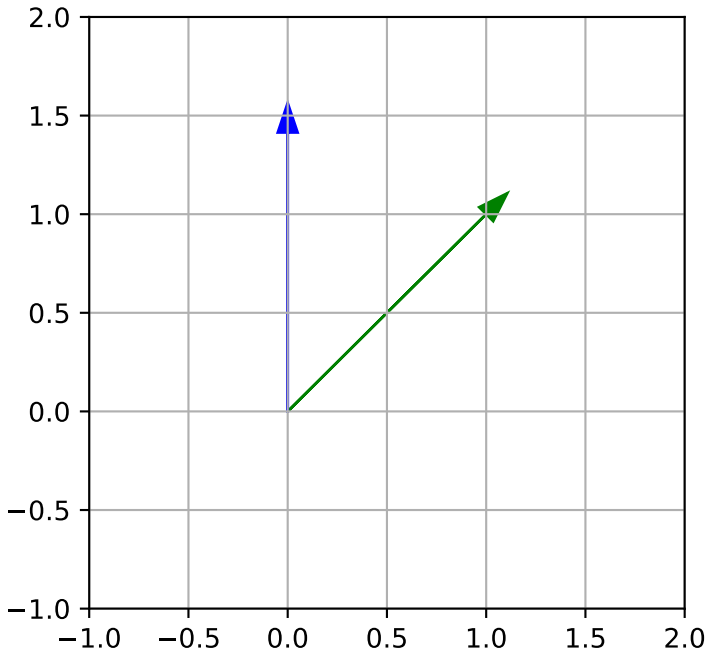
4.2. Beispiele

4.2.1. Drehung

Drehen eines Vektors, der als Pfeil interpretiert wird:

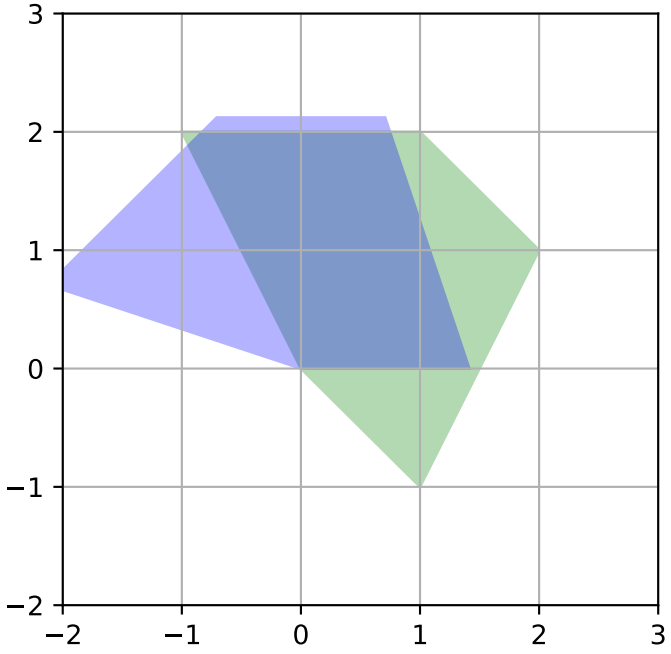
```
alpha = np.pi/4  # 45°
R = np.array([[np.cos(alpha), -np.sin(alpha)],
              [np.sin(alpha),  np.cos(alpha)]])
x = np.array([1, 1])
y = R@x
```

```
plt.figure(figsize=(4,4))
plt.arrow(0, 0, x[0], x[1], head_width=.1, color='green')
plt.arrow(0, 0, y[0], y[1], head_width=.1, color='blue')
plt.xlim(-1, 2)
plt.ylim(-1, 2)
plt.grid(True)
```



Drehen vieler Vektoren, die als Punkte interpretiert werden:

```
# Matrix der Eckpunkte einer Fläche, Eckpunkte als Spaltenvektoren der Matrix:
ecken_orig = np.array([[0, 1, 2, 1,-1],
                       [0,-1, 1, 2, 2]])
ecken_dreh = R@ecken_orig

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot()
ax.add_patch(plt.Polygon(ecken_orig.T, closed=True, color='green', alpha=0.3))
ax.add_patch(plt.Polygon(ecken_dreh.T, closed=True, color='blue', alpha=0.3))
plt.xlim(-2, 3)
plt.ylim(-2, 3)
plt.grid(True)
```



4.2.2. Darstellung in einer Basis

Wir wollen den Vektor $v = \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}$ in der Basis $b_1 = \begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix}$, $b_2 = \begin{pmatrix} 2 \\ 1 \\ 5 \end{pmatrix}$, $b_3 = \begin{pmatrix} 4 \\ -2 \\ 1 \end{pmatrix}$ darstellen. Dazu lösen wir das LGS $Bc = v$ mit der Basismatrix $B = (b_1 \quad b_2 \quad b_3)$:

```
v = np.array([[ 2],
              [ 3],
              [-1]])
B = np.array([[-1,  2,  4],
              [ 3,  1, -2],
              [ 2,  5,  1]])
# Check der Basiseigenschaft: B ist regulär <=> det(B) not 0 <=> Rang von B = 3
print("Rang von B  =", np.linalg.matrix_rank(B))
print("Determinante von B =", np.linalg.det(B))

c = np.linalg.solve(B, v)
print("Koordinaten von v in Basis B: c= \n", c)
```

```
Rang von B  = 3
Determinante von B = 27.0
Koordinaten von v in Basis B: c=
 [[ 3.11111111]
 [-1.88888889]
 [ 2.22222222]]
```

4.2.3. LGS und Nullraum

Nicht-quadratisches, homogenes LGS mit unendlich vielen Lösungen:

```
A = np.array([[ 1,-3, 1],
              [-2, 0, 5]])
b = np.array([[0],
              [0]])
Ab = np.hstack((A, b))
print("Rang von A  =", np.linalg.matrix_rank(A))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
print("Anzahl der Variablen =", np.shape(A)[1])
print("Daher: unendlich viele Lösungen.")

# Der Nullvektor ist die triviale Lsg.
x = np.zeros((3, 1))
print("Ax =\n", A@x)

# Alle Lösungen = Nullraum N:
N = sp.linalg.null_space(A)
# Check, dass die Spalten von N mit A multipliziert Null ergeben:
print("AN =\n", A@N)
```

```
Rang von A  = 2
Rang von Ab = 2
Anzahl der Variablen = 3
Daher: unendlich viele Lösungen.
Ax =
 [[0.]
 [0.]]
AN =
 [[3.33066907e-16]
 [4.44089210e-16]]
```

4.2.4. Quadr. LGS mit eindeutiger Lsg.

Wir betrachten das quadratische LGS

$$\begin{aligned} x_1 - x_2 &= 1 \\ 2x_1 + 3x_2 &= -3 \end{aligned}$$

Determinante der Koeffizientenmatrix ist nicht Null, daher gibt es eine eindeutige Lösung, die `solve` und die Multiplikation mit der inversen Matrix liefert:

```
A = np.array([[ 1,-1],
              [ 2, 3]])
b = np.array([[ 1],
              [-3]])
print(f"Determinante von A = {np.linalg.det(A):.2f}")
print("Daher: genau eine Lösung.")
```

```
print("Rückgabe von solve(A, b):\n", np.linalg.solve(A, b))
print("Rückgabe von inv(A):\n", np.linalg.inv(A))
print("Rückgabe von inv(A)@b:\n", np.linalg.inv(A)@b)
```

```
Determinante von A = 5.00
Daher: genau eine Lösung.
Rückgabe von solve(A, b):
 [[ 0.]
 [-1.]]
Rückgabe von inv(A):
 [[ 0.6  0.2]
 [-0.4  0.2]]
Rückgabe von inv(A)@b:
 [[ 2.22044605e-16]
 [-1.00000000e+00]]
```

4.2.5. Quadr. LGS ohne Lsg.

Wir betrachten das quadratische LGS

$$\begin{aligned} x_1 - 2x_2 &= 1 \\ -2x_1 + 4x_2 &= 3 \end{aligned}$$

Die Determinante der Koeffizientenmatrix ist Null, daher gibt es keine oder unendlich viele Lösungen. Der Befehl `solve` liefert die Fehlermeldung `LinAlgError: Singular matrix`.

```
A = np.array([[ 1, -2],
              [-2,  4]])
b = np.array([[ 1],
              [-3]])
print("Determinante von A =", np.linalg.det(A))
print("Daher: keine oder unendlich viele Lösungen.")
# print("Rückgabe von solve(A,b):\n", np.linalg.solve(A,b)) # -> LinAlgError: Singular matrix
```

```
Determinante von A = 0.0
Daher: keine oder unendlich viele Lösungen.
```

Um in Python zu bestimmen, ob das LGS keine oder unendlich viele Lösungen hat, bestimmen wir den Rang der Koeffizientenmatrix und der erweiterten Koeffizientenmatrix:

```
print("Rang von A  =", np.linalg.matrix_rank(A))
Ab = np.hstack((A, b))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
```

```
Rang von A  = 1
Rang von Ab = 2
```

Das LGS hat also keine Lösung.

4.3. Aufgaben

4.3.1. Verständnisfragen und kurze Aufgaben

1. Geben Sie ein Zahlenbeispiel für ein Matrixprodukt AB, für das BA nicht dasselbe Ergebnis liefert, also $AB \neq BA$.
2. Es seien v und w zwei Spaltenvektoren aus $\mathbb{R}^n$. Welche der folgenden Ausdrücke liefern dasselbe, welche nicht? $v^T w$, vw^T, $w^T v$, wv^T
3. Ist eine Verschiebung in der Ebene, d. h. jedem Punkt x der Ebene wird sein um einen fixen Verschiebungsvektor verschobener Punkt zugeordnet, linear, also als Ax darstellbar? Begründen Sie Ihre Antwort.
4. Wie bestimmen Sie, ob ein quadratisches lineares Gleichungssystem keine, eine oder unendlich viele Lösungen hat?
5. Wie lautet die Rotationsmatrix einer Vektorrotation bei gegebenem Winkel α?
6. Die Rotationsmatrix $R = \begin{pmatrix} -0.707 & -0.707 \\ 0.707 & -0.707 \end{pmatrix}$ wird auf den Vektor $v = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$ angewendet. Wie lautet der resultierende Vektor?
7. Wie können Sie zeigen, dass für die Matrix $\begin{pmatrix} 1 & 4 \\ 8 & 16 \end{pmatrix}$ eine inverse Matrix existiert? Berechnen Sie ggf. die inverse Matrix.
8. Was ist die Determinante einer Matrix, und wozu kann man sie verwenden?

Ergebnisse:

1. Zum Beispiel $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $B = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$
2. $v^T w$ und $w^T v$ liefern dasselbe Ergebnis, nämlich einen Zahl. vw^T und wv^T liefern dasselbe Ergebnis, aber eine Matrix.
3. Nein, da die Verschiebung des Nullvektors nicht der Nullvektor ist. Eine lineare Abbildung bildet den Nullvektor aber immer auf den Nullvektor ab.
4. Siehe Lösungsstruktur quadratischer LGS und Lösungsstruktur Teil 2.
5. $\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$
6. $Rv = \begin{pmatrix} -0.707 & -0.707 \\ 0.707 & -0.707 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} -2.828 \\ 0 \end{pmatrix}$
7. $\det(A) = 16 - 32 = -16 \neq 0$, daher existiert die inverse Matrix. $A^{-1} = \frac{1}{-16} \begin{pmatrix} 16 & -4 \\ -8 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0.25 \\ 0.5 & -0.0625 \end{pmatrix}$
8. Die Determinante einer quadratischen Matrix A ist eine Zahl, die die Fallunterscheidung in der Lösungsstruktur des quadratischen LGS $Ax = b$ ermöglicht. Siehe Determinante.

4.3.2. Vektorrechnung als Matrizenrechnung

1. Stellen Sie in Python die Vektoren $a = \begin{pmatrix} 2 \\ 1 \\ -2 \\ 5 \\ 3 \end{pmatrix}$, $b = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 2 \\ 4 \end{pmatrix}$ und $c = \begin{pmatrix} 0 \\ -2 \\ -1 \\ 3 \\ 6 \end{pmatrix}$ als Spaltenvektoren dar, und berechnen Sie folgende Ausdrücke mittels der Matrizenrechnung:

- $a + 2b - c$
- Summe der inneren Produkte $a \cdot b + a \cdot c$ als $a^T b + a^T c$
- Das innere Produkt $a \cdot a$ als $a^T a$
- $\|a\|$ als $\sqrt{a^T a}$

2. Berechnen Sie mittels der Matrizenrechnung den Winkel zwischen den Spaltenvektoren $a = \begin{pmatrix} -3 \\ 1 \end{pmatrix}$ und $b = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$.

Ergebnis:

- $a + 2b - c = (4, 5, -1, 6, 5)^T$
- $a^T b + a^T c = 58$
- $a^T a = 43$
- $\|a\| \simeq 6.56$
- $\phi \simeq 2.03$ rad $\simeq 116.57°$

4.3.3. Matrixmultiplikation

Gegeben sind die Matrizen

$$A = \begin{pmatrix} 2 & 4 & 1 \\ 1 & 3 & 5 \end{pmatrix}, B = \begin{pmatrix} 3 & 1 & 1 \\ 0 & 2 & 1 \\ -1 & 5 & 1 \end{pmatrix}, \text{ und } C = \begin{pmatrix} -2 & 0 & 3 \\ 2 & 5 & 1 \\ -1 & 1 & 1 \end{pmatrix}.$$

Berechnen Sie von Hand und mit Python ABC.

Ergebnis: $ABC = \begin{pmatrix} 13 & 82 & 37 \\ 59 & 169 & 35 \end{pmatrix}$

4.3.4. Rechnen mit Matrizen

Gegeben sind folgende zwei Matrizen und ein Vektor:

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 0 & -5 & 4 \\ 4 & -3 & 8 \end{pmatrix}, B = \begin{pmatrix} 1 & 2 \\ 0 & 4 \\ 3 & 7 \end{pmatrix} \text{ und } c = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Berechnen Sie folgende Ausdrücke per Hand und überprüfen Sie die Ergebnisse mittels Python:

- AB
- BA
- $B^T A$
- $c^T B$
- $c^T B(BB^T + A)c$

Ergebnisse: $AB = \begin{pmatrix} 10 & 15 \\ 12 & 8 \\ 28 & 52 \end{pmatrix}$, BA ist nicht wohldefiniert, $B^T A = \begin{pmatrix} 13 & -11 & 27 \\ 30 & -45 & 78 \end{pmatrix}$, $c^T B = \begin{pmatrix} 10 & 31 \end{pmatrix}$, $c^T(BB^T + A)c = 1137$

4.3.5. Adjazenz-Matrix

Eine Fluglinie fliegt fünf verschiedene Stationen an. Die möglichen Verbindungen sind nachfolgend dargestellt.

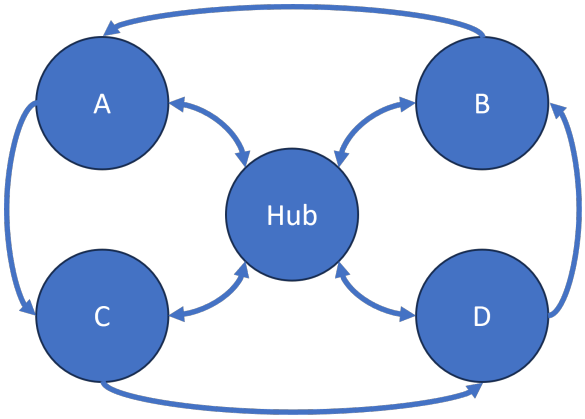


Abbildung 4.1.: Flugplan

Erstellen Sie in Python die Adjazenz-Matrix C für die Fluglinie. Dabei gilt

$$C_{ij} = \begin{cases} 1 \text{ wenn es einen Direktflug von i nach j gibt} \\ 0 \text{ sonst} \end{cases}$$

Um zu bestimmen, wie viele mögliche Flugrouten es zwischen 2 Stationen gibt, kann die Matrixpotenz, also $C^k = C \cdot C \cdot ...$ verwendet werden, siehe z. B. Wikipedia. Das heißt, die Anzahl der möglichen Flugrouten mit *genau einem Umstieg* zwischen zwei beliebigen Stationen sind die Einträge in der Matrix C^2.

Bestimmen Sie in Python die Anzahl der Flugrouten zwischen zwei beliebigen Stationen mit (a) genau einem Umstieg und (b) mit genau zwei Umstiegen. (c) Wie viele Routen gibt es vom Startort A zum Zielort B mit *maximal* zwei Umstiegen?

Ergebnis: (a) C^2, (b) C^3, (c) entsprechendes Element aus $C + C^2 + C^3$

4.3.6. Matrixgleichung

Gegeben sind die Matrizen $A = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$ und $B = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 3 \end{pmatrix}$. Bestimmen Sie in Python die Matrix X, die die Bedingung $X = AX + B$ erfüllt, falls es eine solche Matrix gibt.

Ergebnis: $X = \begin{pmatrix} 2 & 4 \\ -1 & -2 \\ 3 & 3 \end{pmatrix}$

4.3.7. Konturlinien einer linearen Abbildung

Zeichnen Sie von Hand die Konturlinien (Höhenlinien, Isolinien) der linearen Abbildung $f(x) = a^T x$ für den Vektor $a = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$.

Ergebnis: Die Konturlinien sind Geraden, die durch den Nullpunkt gehen und den Vektor a als Normalenvektor haben.

4.3.8. Darstellung in einer Basis

Gegeben ist der Vektor $v = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ und die Basis $B = \begin{pmatrix} 3 & 1 \\ 1 & -2 \end{pmatrix}$.

1. Zeichnen Sie die Basisvektoren $b_1 = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$ und $b_2 = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$ in ein Koordinatensystem.
2. Zeigen Sie, dass B eine Basis ist.
3. Stellen Sie v in der Basis B dar, indem Sie von Hand und am Computer ein entsprechendes LGS lösen.

Ergebnis: Die Basisvektoren sind linear unabhängig und spannen den $\mathbb{R}^2$ auf. $v = Bc = 2 \cdot b_1 - 1 \cdot b_2$

4.3.9. Projektion

Bestimmen Sie die orthogonale Projektion des Punktes $a = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ auf die Gerade, die vom Vektor $v = \begin{pmatrix} -3 \\ 4 \end{pmatrix}$ aufgespannt wird.

1. Zeichnen Sie die Gerade und den Punkt a in ein Koordinatensystem, und schätzen Sie die Projektion ab.
2. Bestimmen Sie die Projektion rechnerisch von Hand und am Computer.

Ergebnis: Die Projektion von a auf die von v aufgespannte Gerade ist $\begin{pmatrix} -0.6 \\ 0.8 \end{pmatrix}$.

4.3.10. PV-Ertrag

Die ebenen Module einer Photovoltaikanlage werden so ausgerichtet, dass sie eine Normalvektor $n = \begin{pmatrix} 5 \\ -5 \\ 7 \end{pmatrix}$ haben. Die Sonne strahlt entlang des Vektors $s = \begin{pmatrix} -10 \\ 1 \\ -5 \end{pmatrix}$. Die Länge des Vektors s gibt die Intensität der Sonneneinstrahlung in $\frac{\mathrm{kW}}{\mathrm{m}^2}$ an. Wie viel der Strahlungsleistung wird von den Modulen pro Quadratmeter aufgenommen? Bearbeiten Sie die folgenden Aufgaben am Computer:

1. Bestimmen Sie die Abbildungsmatrix der orthogonalen Projektion auf die von n aufgespannte Gerade. Berechnen Sie damit die orthogonale Projektion des Vektors s auf diese Gerade. Bestimmen Sie nun den Anteil der Strahlungsleistung, die von den Modulen aufgenommen wird, indem Sie das Verhältnis der Längen der Projektion von s und des Vektors s berechnen.

2. Berechnen Sie den Winkel ϕ zwischen n und s, und bestimmen Sie den Kosinus des Winkels. Klären Sie den Zusammenhang zwischen dem Kosinus des Winkels und dem Anteil der Strahlungsleistung, die von den Modulen aufgenommen wird.

Ergebnis: Der Anteil der Strahlungsleistung, die von den Modulen aufgenommen wird, ist ca. $0.8058 = 80.58\ \%$.

4.3.11. Bild, Kern und Rangsatz

1. Sei P die Abbildungsmatrix der orthogonalen Projektion auf eine Gerade durch den Ursprung im Raum. Welche Dimensionen hat das Bild und der Kern von P? Überprüfen Sie den Rangsatz. Überprüfen Sie Ihr allgemeines Ergebnis an einem Beispiel in Python.
2. Wiederholen Sie die Aufgabe für die orthogonale Projektion auf eine Ebene durch den Ursprung im Raum.

Ergebnis:

1. Der Kern von P ist die Ebene senkrecht zur Geraden, das Bild ist die Gerade. Rangsatz: $\dim(\ker(P)) + \dim(\mathrm{im}(P)) = 2 + 1 = \dim(\mathbb{R}^3) = 3$.
2. Der Kern von P ist die Gerade senkrecht zur Ebene, das Bild ist die Ebene. Rangsatz: $\dim(\ker(P)) + \dim(\mathrm{im}(P)) = 1 + 2 = \dim(\mathbb{R}^3) = 3$.

4.3.12. Determinanten

Berechnen Sie mit dem Multiplikationstheorem für Determinanten $\det(A \cdot B) = \det(A) \cdot \det(B)$ die Determinante des Matrizenproduktes $C = A \cdot B$ von Hand und am Computer für

$$A = \begin{pmatrix} -2 & 3 & 9 \\ 7 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \text{ und } B = \begin{pmatrix} -1 & 1 & 3 \\ 2 & 5 & 9 \\ 0 & 6 & 1 \end{pmatrix}.$$

Ergebnis: $\det(A) \cdot \det(B) = 68 \cdot 83 = 5644$

4.3.13. Inverse Matrix

Zeigen Sie von Hand, dass $A = \begin{pmatrix} 1 & 2 \\ 0.5 & 3 \end{pmatrix}$ eine reguläre Matrix ist, und bestimmen Sie von Hand die inverse Matrix A^{-1}. Überprüfen Sie das Ergebnis, indem Sie AA^{-1} und $A^{-1}A$ berechnen.

Ergebnis: $\det(A) = 2 \neq 0$, $A^{-1} = \begin{pmatrix} 1.5 & -1 \\ -0.25 & 0.5 \end{pmatrix}$

4.3.14. Quadratisches LGS

Wir betrachten das folgende quadratische LGS:

$$\begin{aligned} 2x_1 - 5x_2 + 8x_3 &= 0 \\ -2x_1 - 7x_2 + x_3 &= 0 \\ 4x_1 + 2x_2 + 7x_3 &= 0 \end{aligned}$$

Bestimmen Sie in Python die Lösungsstruktur des LGS, d. h. ob es keine, eine oder unendlich viele Lösungen hat.

Ergebnis: Das LGS hat unendlich viele Lösungen.

4.3.15. Inverse einer 2x2 Matrix

1. Überprüfen Sie, dass die Inverse einer 2x2 Matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ durch folgende Formel gegeben ist:
$$A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \text{ mit } \det(A) = ad - cb.$$
2. Benutzen Sie diese Formel, um von Hand die Inverse der Matrix $A = \begin{pmatrix} 0.5 & 0 \\ 0 & 3 \end{pmatrix}$ zu bestimmen. Beschreiben Sie die Abbildung $f(x) = Ax$ in Worten.

Ergebnis:

1. Berechne $A^{-1}A$ und AA^{-1}.
2. $A^{-1} = \begin{pmatrix} 2 & 0 \\ 0 & \frac{1}{3} \end{pmatrix}$. Sie Abbildung A staucht in x_1-Richtung um Faktor 2 und streckt in x_2-Richtung um Faktor 3.

4.3.16. Rotationen in der Ebene

Betrachten Sie die Rotationsmatrix

$$R(\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}.$$

1. Welche Rotation beschreibt die Abbildung $f(x) = Rx$?
2. Zeigen Sie von Hand, dass $R^T R = I$ gilt, und berechnen Sie $\det(R)$ sowie R^{-1}.

Ergebnisse:

1. Rotation um den Winkel α gegen den Uhrzeigersinn um den Koordinatenursprung.
2. $R(\alpha)^{-1} = R(-\alpha) = R(\alpha)^T$.

4.3.17. Existenz und Eindeutigkeit

Gegeben ist das lineare Gleichungssystem:

$$\begin{aligned} 2x_1 - 4x_2 - 2x_3 &= b_1 \\ -5x_1 + x_2 + x_3 &= b_2 \\ 7x_1 - 5x_2 - 3x_3 &= b_3 \end{aligned}$$

Bestimmen Sie von Hand und am Computer mittels der Determinante der Koeffizientenmatrix, ob das lineare Gleichungssystem für alle Vektoren b eine eindeutige Lösung hat.

Ergebnis: Das LGS hat nicht für alle Vektoren b eine eindeutige Lösung.

4.3.18. Existenz und Eindeutigkeit einer Lsg.

Bestimmen Sie von Hand, ohne das folgende, quadratische LGS zu lösen, ob es eindeutig lösbar ist.

$$\begin{pmatrix} -10 & 3 \\ -5 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -3.5 \\ 9.2 \end{pmatrix}$$

Ergebnis: Das LGS hat eine eindeutige Lösung.

4.3.19. Leontief Modell

Nehmen Sie an, ein Staat ist Standort für drei verschiedene Industriesektoren: Chemische Industrie, Nahrungsmittelindustrie und Ölindustrie. Die Produktion einer Einheit Chemikalien benötigt 0.2 Einheiten von Chemikalien, 0.4 Einheiten Nahrungsmittel und 0.5 Einheiten Öl. Diese Größen werden in die erste Spalte der Verbrauchsmatrix A ein, die den Verbrauch während der Produktion wiedergibt:

$$\begin{pmatrix} \text{Chemie Verbrauch} \\ \text{Nahrungsmittel Verbrauch} \\ \text{Öl Verbrauch} \end{pmatrix} = \begin{pmatrix} 0.2 & 0.3 & 0.4 \\ 0.4 & 0.4 & 0.1 \\ 0.5 & 0.1 & 0.3 \end{pmatrix} \begin{pmatrix} \text{Chemische Produktion} \\ \text{Nahrungsmittel Produktion} \\ \text{Öl Produktion} \end{pmatrix}$$

Bemerkung: Die Verbrauchsmatrix der USA im Jahr 1958 enthielt 83 industrielle Sektoren, heutige Modelle sind bei weitem umfangreicher.

Es stellt sich die Frage, ob die so beschriebene Volkswirtschaft einen Bedarf $d = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$ für chemische Produkte, Nahrungsmittel und Öl decken kann? Dazu muss neben dem Produktionsplan $p = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$ der Verbrauch Ap während der Produktion berücksichtigt werden. Die Netto-Produktion von $p - Ap$ soll daher den Bedarf d decken. Damit gilt es für einen gegebenen Bedarf d einen Produktionsplan p zu finden, so dass folgendes erfüllt ist:

$$\begin{aligned} p - Ap &= d \text{ bzw.} \\ p &= (I - A)^{-1} d \end{aligned}$$

Der Bedarfsvektor d und die Verbrauchsmatrix A dürfen keine negativen Einträge haben, um wirtschaftlich sinnvoll interpretiert werden zu können. Ebenso der Produktionsplan $p = (I - A)^{-1} d$.

Daher lautet die Frage: Wann ist $(I - A)^{-1}$ eine nicht-negative Matrix, d. h. jeder der Einträge ist nicht-negativ.

1. Berechnen Sie mit Python den Produktionsplan für den Bedarf $d = \begin{pmatrix} 4 \\ 2 \\ 3 \end{pmatrix}$. Benutzen Sie dazu den Befehl `solve`.
2. Berechnen Sie mit dem Python-Befehl `inv` den Ausdruck $(I - A)^{-1}$. Ist die Matrix nicht-negativ?
3. Ändern Sie die Verbrauchsmatrix A zu einer nicht-negativen Verbrauchsmatrix B, die den Bedarf nicht decken kann.

Bemerkung: Die Matrix $I - A$ wird Technologiematrix genannt. Sie führt einen Produktionsplan in einen Vektor über, der angibt, welcher Bedarf erfüllt werden kann.

Ergebnisse:

1. $p = \begin{pmatrix} 31.72 \\ 29.68 \\ 31.18 \end{pmatrix}$
2. Ja.
3. Zum Beispiel $B = \begin{pmatrix} 0.2 & 0.9 & 0.4 \\ 0.4 & 0.4 & 0.1 \\ 0.5 & 0.1 & 0.3 \end{pmatrix}$

5. Eigenwerte und Eigenvektoren

5.1. Methoden

Motivation

Der Effekt einer linearen Abbildung, also die Multiplikation eines Vektors x mit einer Matrix A, ist im Allgemeinen unübersichtlich, d. h. der Output Ax hängt vom Input auf eine komplizierte Weise ab. Wäre der Effekt eine reine Skalarmultiplikation λx, so wäre der Output eine Umsaklierung (Streckung, Stauchung, Umkehrung) des Inputs x und deutlich einfacher zu handhaben.
Anwendungen:

- Entkoppeln von Gleichungen: Dynamische Systeme, Differentialgleichungen (-> Eigenfrequenzen, etc.)
- Nicht-lineare Optimierung: Kriterium für lokales Maximum bzw. Minimum
- Hauptachsentransformation, Principal Component Analysis
- PageRank-Algorithmus von Google
- Quantenmechanik
- etc.

5.1.1. Definitionen

Ein Eigenvektor (EV) einer quadratischen $n \times n$-Matrix A ist ein nicht-Null-Vektor $x \in \mathbb{R}^n$, sodass

$$Ax = \lambda x$$

für eine Zahl λ gilt. Die Zahl λ ist der Eigenwert (EW) von A zum Eigenvektor x.

Achtung: Jedes Vielfache eines Eigenvektors ist wieder Eigenvektor zum selben Eigenwert: $A(\alpha x) = \alpha Ax = \alpha\lambda x = \lambda(\alpha x)$.

5.1.2. Spektralzerlegung

Wenn man eine Basis des $\mathbb{R}^n$ aus Eigenvektoren $v_i \in \mathbb{R}^n, i = 1, \dots, n$ bilden kann, dann heißt A diagonalisierbar und die Basis Eigenbasis. Man bildet die Matrix V, die die Basis-Eigenvektoren als Spalten hat, und die Diagonalmatrix D mit den zugehörigen Eigenwerten λ_i auf der Diagonalen. Dann gilt $AV = A(v_1|v_2|\dots|v_n) = (\lambda_1 v_1|\lambda_2 v_2|\dots|\lambda_n v_n) = VD$, also

$$AV = VD$$

und, weil V als Basisvektoren-Matrix invertierbar ist, durch Umformen $A = VDV^{-1}$ und $V^{-1}AV = D$.

Diese Formeln ergeben sich auch aus dem Basiswechsel (Koordinatentransformation) im In- und Outputraum der linearen Abbildung $y = Ax$. Seinen $x = Vc$ und $y = Vd$ mit c und d die Koordinaten von x bzw. y bzgl. der Eigenbasis V. Dann erhält man durch Einsetzen in $y = Ax$ und Multiplikation mit V^{-1}:

$$\begin{aligned} Vd &= AVc = VDc \\ d &= V^{-1}AVc = Dc. \end{aligned}$$

Die Abbildung, die in Standardkoordinaten die Matrix A hat, hat bzgl. den angepassten Eigenkoordinaten die Diagonalmatrix D.

Das Auffinden der EW und EV einer quadratischen Matrix A und die Diagonalisierung $A = VDV^{-1}$ wird auch Spektralzerlegung von A genannt. Weitere wichtige Matrixzerlegungen sind

- QR-Zerlegung
- LU-Zerlegung
- Cholesky-Zerlegung
- SVD-Zerlegung.

5.1.3. Berechnung

Falls x ein Eigenvektor von A zum Eigenwert λ ist, dann gelten:

$$\begin{aligned} Ax &= \lambda x \\ Ax - \lambda x &= 0 \\ Ax - \lambda Ix &= 0 \\ (A - \lambda I)x &= 0. \end{aligned}$$

Da x laut Annahme ein nicht-Null-Vektor ist, hat das letzte Gleichungssystem neben der trivialen Lösung des Nullvektors mit x eine zweite und somit nicht eindeutige Lösung. Daher muss die Determinante der Koeffizientenmatrix Null sein:

$$\det(A - \lambda I) = 0.$$

Andernfalls hätte das letzte Gleichungssystem den Nullvektor als eindeutige Lösung. Aus der Bedingung $\det(A - \lambda I) = 0$ lassen sich alle Eigenwerte berechnen, die auch komplex sein können. Anschließend kann zu jedem Eigenwert λ ein Eigenvektor berechnet werden, indem eine nicht-triviale Lösung von $(A - \lambda)x = 0$ bestimmt wird. Die Bedingung $\det(A - \lambda I) = 0$ heißt charakteristische Gleichung von A.

Beispiel: $A = \begin{pmatrix} -2 & -5 \\ 1 & 4 \end{pmatrix}$, $\det(A - \lambda I) = \det \begin{pmatrix} -2-\lambda & -5 \\ 1 & 4-\lambda \end{pmatrix} = (-2-\lambda)(4-\lambda) - (-5) = \lambda^2 - 2\lambda - 3 = 0$ hat die Lösungen $\lambda_1 = -1$ und $\lambda_2 = 3$. Das sind die Eigenwerte von A.

- Eigenwert $\lambda_1 = -1$: $(A - \lambda_1 I)x = \begin{pmatrix} -1 & -5 \\ 1 & 5 \end{pmatrix} x = 0$ hat unendlich viele Lösungen, z. B. $v_1 = \begin{pmatrix} 5 \\ -1 \end{pmatrix}$. v_1 ist ein Eigenvektor von A zum Eigenwert $\lambda_1 = -1$.
- Eigenwert $\lambda_2 = 3$: $(A - \lambda_2 I)x = \begin{pmatrix} -5 & -5 \\ 1 & 1 \end{pmatrix} x = 0$ hat unendlich viele Lösungen, z. B. $v_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. v_2 ist ein Eigenvektor von A zum Eigenwert $\lambda_2 = 3$.

5.1.4. Python

Die zentrale Python-Funktion ist `numpy.linalg.eig`. Sie liefert die Eigenwerte und Eigenvektoren einer Matrix. Die Eigenwerte werden in einem Vektor und die Eigenvektoren als Matrix mit den zugehörigen Eigenvektoren als Spalten zurückgegeben. Die Eigenvektoren sind auf die Länge 1 normiert, d. h. sie sind Einheitsvektoren.

Für die Beispiele und Aufgaben verwenden wir folgende Python-Bibliotheken, siehe Kapitel Python Tutorial:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

5.2. Beispiele

5.2.1. Berechnung von EW und EV

Für $A = \begin{pmatrix} 0.8 & 0.3 \\ 0.2 & 0.7 \end{pmatrix}$ liefert die Bedingung $\det(A - \lambda I) = 0$ die quadratische Gleichung $\lambda^2 - 1.5\lambda + 0.5 = 0$. Deren Lösungen $\lambda_1 = 1$ und $\lambda_2 = \frac{1}{2}$ sind die Eigenwerte von A.

- Die Eigenvektoren zum Eigenwert λ_1 sind die Lösungen von $(A - \lambda_1 I)x = 0$, in unserem Beispiel ist das $\begin{pmatrix} -0.2 & 0.3 \\ 0.2 & -0.3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Die Eigenvektoren sind alle Vielfachen von z. B. $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$.
- Die Eigenvektoren zum Eigenwert λ_2 sind die Lösungen von $(A - \lambda_2 I)x = 0$, in unserem Beispiel ist das $\begin{pmatrix} 0.3 & 0.3 \\ 0.2 & 0.2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Die Eigenvektoren sind alle Vielfachen von z. B. $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

```
import numpy as np

A = np.array([[0.8, 0.3],
              [0.2, 0.7]])
L, V = np.linalg.eig(A)
print(f"eigenvalues: {L}")
print(f"matrix with eigenvectors in columns:\n{V}")
```

```
eigenvalues: [1.  0.5]
matrix with eigenvectors in columns:
[[ 0.83205029 -0.70710678]
 [ 0.5547002   0.70710678]]
```

5.2.2. Bevölkerungswachstum

Die Population einer Region zerlegt sich in die Altersgruppen

- jünger als 20 Jahre
- 20 bis 39 Jahre und
- über 40 Jahre.

Dynamisches System: Anfangs ist der Generationenstand durch den Vektor $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ gegeben. Eine Generation später ist die Population durch zwei Effekte verändert:

- Reproduktion: $x_1^{\text{neu}} = F_1 x_1 + F_2 x_2 + F_3 x_3$ mit den Fertilitätsraten F_k.
- Überlebensrate: $x_2^{\text{neu}} = P_1 x_1$ and $x_3^{\text{neu}} = P_2 x_2$ mit den Überlebenswahrscheinlichkeiten P_k.

Die Leslie Matrix A liefert x^{neu} für x via $x^{\text{neu}} = Ax$. Betrachten Sie folgendes Beispiel:

$$x^{\text{neu}} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^{\text{neu}} = \begin{pmatrix} F_1 & F_2 & F_3 \\ P_1 & 0 & 0 \\ 0 & P_2 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} .04 & 1.1 & .01 \\ .98 & 0 & 0 \\ 0 & .92 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Aufgaben:

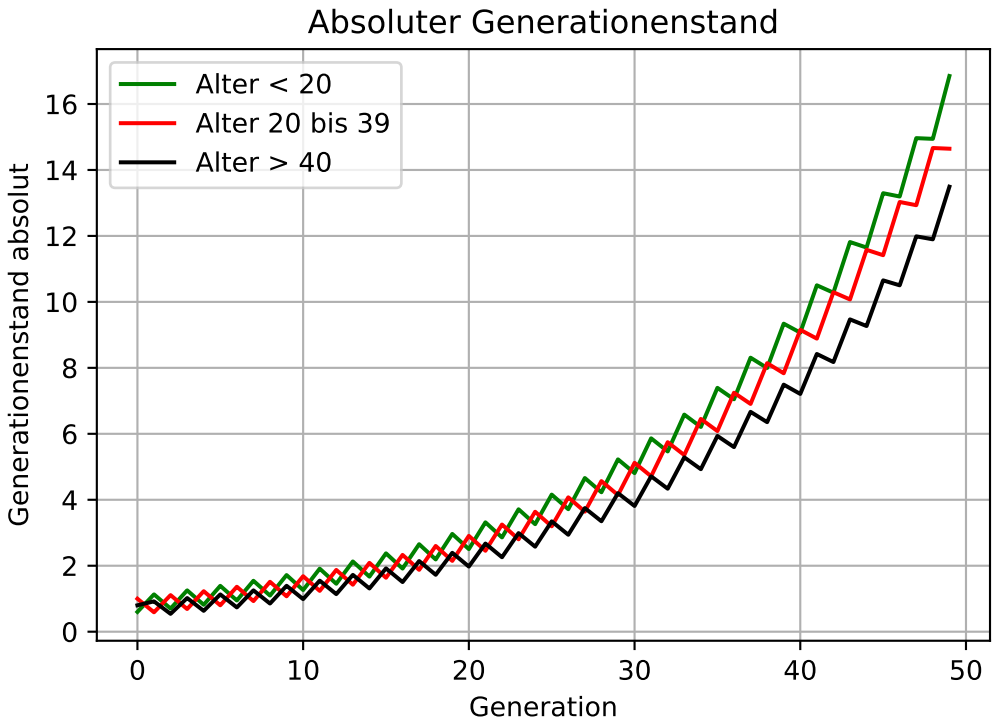
1. Der Generationenstand sei zu Beginn durch $x = \begin{pmatrix} .6 \\ 1 \\ .8 \end{pmatrix}$ [Mio. Menschen] gegeben. Benutzen Sie Python, um die absoluten und relativen Populationsgrößen aller drei Gruppen für die nächsten 30 Generationen zu berechnen und zu plotten.
2. Berechnen Sie die Eigenwerte und Eigenvektoren der Leslie Matrix mithilfe des Python Befehls `eig`, und interpretieren Sie das Resultat. Welcher Eigenwert ist weshalb für die Bevölkerungsentwicklung entscheidend? Wie kann ein zugehöriger Eigenvektor zu diesem Eigenwert zur Prognose der sich abzeichnenden relativen Populationsgrößen verwendet werden?

Lösung:

```
A = np.array([[0.04, 1.1 , 0.01],
              [0.98, 0   , 0   ],
              [0   , 0.92, 0   ]])
print(A)
(L, V) = np.linalg.eig(A)

for k in range(len(L)):
  print(f"Der Eigenwert {L[k]:.2f} hat den Eigenvektor\n {V[:,[k]]}")

x0 = np.array([[0.6, 1, 0.8]]).T
# x0 = -V[:,[0]]
# x0 = V[:,[1]]
# x0 = V[:,[2]]

N = 50
X = np.zeros((3, N))
X[:,[0]] = x0

for k in range(1, N):
    X[:,[k]] = A@X[:,[k-1]]

plt.figure(figsize=(6, 4))
plt.title('Absoluter Generationenstand')
plt.plot(X[0,:], 'g', label='Alter < 20')
plt.plot(X[1,:], 'r', label='Alter 20 bis 39')
```

```
plt.plot(X[2,:], 'k', label='Alter > 40')
plt.xlabel('Generation')
plt.ylabel('Generationenstand absolut')
plt.legend()
plt.grid(True)
```

```
[[0.04 1.1  0.01]
 [0.98 0.   0.  ]
 [0.   0.92 0.  ]]
Der Eigenwert 1.06 hat den Eigenvektor
 [[-0.63392495]
 [-0.58468167]
 [-0.50624747]]
Der Eigenwert -1.01 hat den Eigenvektor
 [[ 0.6083396 ]
 [-0.58784241]
 [ 0.53325813]]
Der Eigenwert -0.01 hat den Eigenvektor
 [[ 7.76398245e-05]
 [-9.09394695e-03]
 [ 9.99958646e-01]]
```



Jeder Anfangszustand x kann in der Eigenbasis darstellt werden: $x = Vy$, wobei y die Koordinaten von x in der Eigenbasis sind. Im Detail bedeutet das, dass

$$x = \sum_{i=1}^{3} y_i v_i$$

mit den Eigenvektoren v_i und den Koordinaten y_i. Um den Zustand n Generationen später zu berechnen, wendet man die Leslie-Matrix A n-mal auf x an:

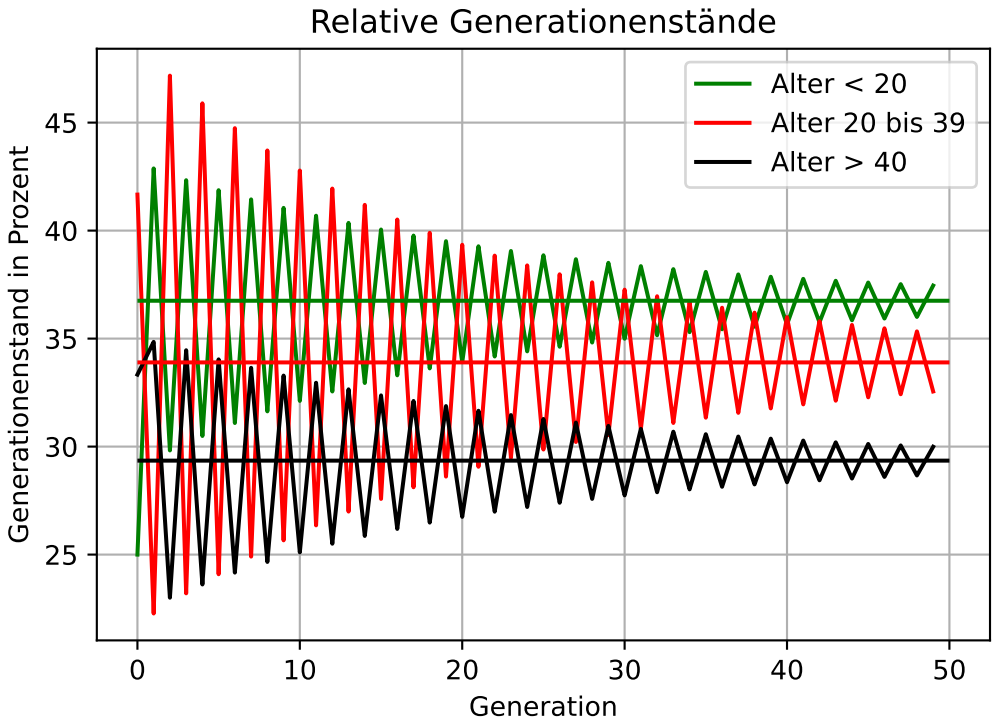
$$x^{(n)} = A^n x = \sum_{i=1}^{3} y_i A^n v_i = \sum_{i=1}^{3} y_i \lambda_i^n v_i.$$

Der Zustand $x^{(n)}$ ist also eine Linearkombination der Eigenvektoren v_i mit den Faktoren $\lambda_i^n y_i$. Der Eigenvektor v_i mit dem größten Eigenwert λ_i bestimmt das langfristige Verhalten der Population.

Umskalieren des Eigenvektors mit größten Eigenwert:

```
E = V/np.sum(V, axis=0)
e1 = E[:,[0]]*100  # rescale the first eigenvector to percentage
print(f"rescaled the first eigenvector:\n{e1}")

plt.figure(figsize=(6, 4))
plt.title('Relative Generationenstände')
plt.plot(X[0,:]/np.sum(X, axis = 0)*100, 'g', label='Alter < 20')
plt.plot(X[1,:]/np.sum(X, axis = 0)*100, 'r', label='Alter 20 bis 39')
plt.plot(X[2,:]/np.sum(X, axis = 0)*100, 'k', label='Alter > 40')
plt.hlines(e1[0,0], 0, N, color='g')
plt.hlines(e1[1,0], 0, N, color='r')
plt.hlines(e1[2,0], 0, N, color='k')
plt.legend()
plt.xlabel('Generation')
plt.ylabel('Generationenstand in Prozent')
plt.grid(True)
```

```
rescaled the first eigenvector:
[[36.75238138]
 [33.89745683]
 [29.3501618 ]]
```



5.3. Aufgaben

5.3.1. Verständnisfragen und kurze Aufgaben

1. Beschreiben Sie die Rechenschritte zur Bestimmung der Eigenvektoren einer Matrix.
2. Welche Eigenwerte hat die Matrix $\begin{pmatrix} 1 & 3 \\ 0 & -2 \end{pmatrix}$?
3. Welche Eigenwerte hat die Matrix $\begin{pmatrix} 1 & -2 \\ 1 & 4 \end{pmatrix}$?

Ergebnisse:

1. Siehe Abschnitt Berechnung
2. $\lambda_1 = 1, \lambda_2 = -2$
3. $\lambda_1 = 2, \lambda_2 = 3$

5.3.2. Eigenwerte und -vektoren

Berechnen Sie Eigenwerte und Eigenvektoren der Matrix $\begin{pmatrix} 3 & 0.25 \\ 1 & 3 \end{pmatrix}$, und überprüfen Sie Ihr Ergebnis in Python.

Ergebnisse: Die Eigenwerte sind $\lambda_1 = 3.5$ und $\lambda_2 = 2.5$. Die Eigenvektoren sind z. B. $v_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ und $v_2 = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$.

5.3.3. Diskrete Dynamische Systeme, Eigenwerte und -vektoren

Jedes Jahr ziehen etwa 10 % der Bevölkerung einer Stadt in die umliegenden Vororte, und etwa 20 % der Vorstadtbevölkerung in die Stadt. Im Jahr 2016 gibt es 10 Millionen Menschen in der Stadt und 8 Millionen Menschen in den Vororten, was wir mit dem Zustandsvektor $x_0 = \begin{pmatrix} 10 \\ 8 \end{pmatrix}$ modellieren.

1. Bestimmen Sie die Übergangsmatrix M, so dass Mx der Einwohnerverteilung ein Jahr nach x entspricht.
2. Analysieren Sie das Langzeitverhalten des dynamischen Systems mit Hilfe der Eigenwerte und -vektoren von M, d. h. wohin tendiert der Zustandsvektor für große Zeiten?

Ergebnisse: Die Übergangsmatrix ist $M = \begin{pmatrix} 0.9 & 0.2 \\ 0.1 & 0.8 \end{pmatrix}$. Der Zustand tendiert für große Zeiten gegen $x = \begin{pmatrix} 12 \\ 6 \end{pmatrix}$.

5.3.4. Ein Räuber-Beute-System

Tief in den Redwood-Wäldern von Kalifornien liefern die Dunkelfuß-Buschratten bis zu 80 % der Nahrung für den Fleckenkauz, den Hauptfeind der Buschratten. Wir verwenden ein lineares dynamisches System, um die Dynamik des Eulen- und Rattenbestands zu modellieren. Das Modell ist in mehrfacher Hinsicht unrealistisch, aber es kann als Ausgangspunkt für die Untersuchung komplizierterer nichtlinearer Modelle verwendet werden. Mit $x_t = \begin{pmatrix} f_t \\ b_t \end{pmatrix}$ bezeichnen wir für den Monat t den Bestand des Fleckenkauzes sowie den Bestand an Buschratten gemessen in Tausenden. Ein Monat später sei der Bestand gegeben durch

$$\begin{aligned} f_{t+1} &= 0.5 f_t + 0.4 b_t, \\ b_{t+1} &= -0.104 f_t + 1.1 b_t. \end{aligned}$$

Der Term $0.5 f_t$ in der ersten Gleichung besagt, dass ohne Buschratten als Nahrung nur die Hälfte der Fleckenkauz jeden Monat überleben wird, während der Term $1.1 b_t$ in der zweiten Gleichung besagt, dass ohne Fleckenkauz als Räuber die Buschrattenpopulation um 10 % pro Monat zunehmen wird. Wenn es viele Buschratten gibt, führt der Term $0.4 b_t$ zu einem Anstieg der Fleckenkauzpopulation, während der negative Term $-0,104 f_t$ den Tod von Buschratten durch den Fleckenkauz beschreibt. Ein Fleckenkauz frisst also pro Monat im Schnitt 104 Buschratten.

Wir beschreiben die Bestandsdynamik kann durch die Vektorgleichung $x_{t+1} = Mx_t$. Die Übergangsmatrix M lautet

$$M = \begin{pmatrix} 0.5 & 0.4 \\ -0.104 & 1.1 \end{pmatrix}.$$

1. Bestimmen Sie in Python die Eigenwerte und Eigenvektoren von M. Skalieren Sie die Eigenvektoren sinnvoll.
2. Bestimmen Sie daraus den langfristigen Bestand der Eulen und Buschratten.
3. Plotten Sie die Bestandsentwicklung der Eulen und Buschratten für die nächsten Monate für unterschiedliche Anfangsbedingungen in der $f - b$-Ebene.

5.3.5. Eigenwerte und -vektoren, Invertierbarkeit, Eindeutigkeit der Lösung

Gegeben ist die Matrix $A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 0 \\ 0 & 1 & 3 \end{pmatrix}$.

1. Berechnen Sie von Hand und am Computer die Eigenwerte und Eigenvektoren von A.
2. Ist die Matrix A invertierbar? Begründen Sie Ihre Antwort.
3. Besitzt das Gleichungssystem $Ax = b$ eine eindeutige Lösung? Begründen Sie Ihre Antwort.

Hinweis:

$$\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{12}a_{21}a_{33} - a_{11}a_{23}a_{32}.$$

Ergebnisse:

1. Eigenwerte von A: $\lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 3$. Zugehörigen Eigenvektoren sind z. B. $v_1 = (1,0,0)^T, v_2 = (1,0,1)^T, v_3 = (-2,1,-1)^T$.
2. Ja, denn die Determinante ist nicht Null.
3. Ja, denn die Determinante ist nicht Null.

5.3.6. Eigenwerte und -vektoren

Bestimmen Sie von Hand und in Python alle Eigenwerte und -vektoren folgender Matrix.

$$\begin{pmatrix} 2 & 7 \\ 7 & 2 \end{pmatrix}$$

Ergebnisse: Eingenwerte von A: $\lambda_1 = -5, \lambda_2 = 9$. Zugehörige Eigenvektoren sind z. B. $v_1 = (-1,1)^T, v_2 = (1,1)^T$.

5.3.7. Eigenwerte und -vektoren

Bestimmen Sie von Hand und in Python alle Eigenwerte und -vektoren folgender Matrix.

$$\begin{pmatrix} -2 & -5 \\ 1 & 4 \end{pmatrix}$$

Quelle: Papula, Band 2, 7.2, S. 128 f.

Ergebnisse:

Eingenwerte von A: $\lambda_1 = -1, \lambda_2 = 3$. Zugehörige Eigenvektoren sind z. B. $v_1 = (-5,1)^T$, $v_2 = (-1,1)^T$.

5.3.8. Eigenwerte und -vektoren

Bestimmen Sie von Hand die Eigenwerte und Eigenvektoren von $A = \begin{pmatrix} 5 & 1 \\ 4 & 2 \end{pmatrix}$.

Ergebnisse: $\lambda_1 = 1, \lambda_2 = 6$, $v_1 = (1,-4)^T$, $v_2 = (1,1)^T$.

5.3.9. Eigenwerte und -vektoren

Bestimmen Sie von Hand und am Computer die Eigenwerte und Eigenvektoren von $A = \begin{pmatrix} -1 & 2 \\ 4 & 1 \end{pmatrix}$?

Ergebnisse: $\lambda_1 = 3, \lambda_2 = -3$, $v_1 = (1, 2)^T$, $v_2 = (1, -1)^T$.

6. Regression

6.1. Methoden

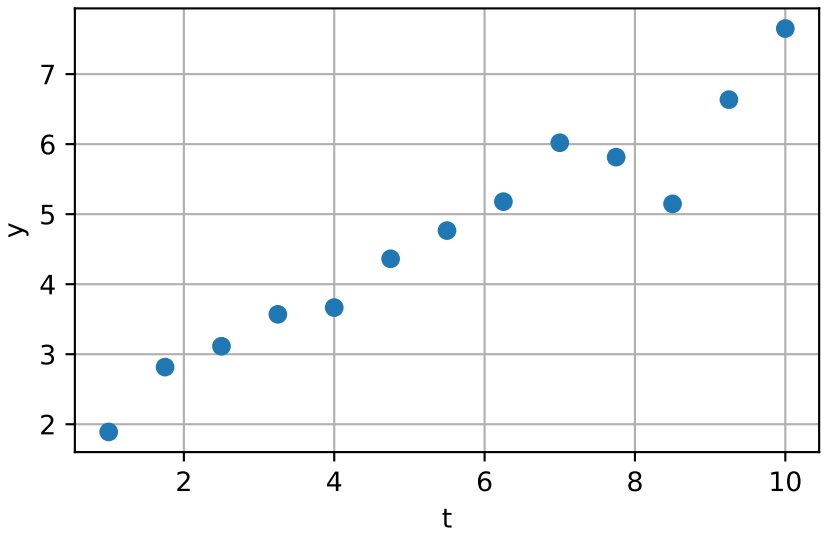
Motivation

Die Regression (Ausgleichsrechnung, Methode der kleinsten Fehlerquadrate, ordinary least squares) ist ein Standardwerkzeug in sehr vielen Disziplinen und hat viele praktische Anwendungen (Datenanalyse, Statistik, Prognoseerstellung, Machine Learning, Beschreibung von Zusammenhängen, Parameterschätzung für Systemidentifikation etc.).
Aus mathematischer Sicht ist die Regression ein quadratisches (und deshalb relativ einfaches) Optimierungsproblem, das immer lösbar ist und unter einfachen Voraussetzungen eine eindeutige, globale Lösung hat. Geometrisch betrachtet ist die Regression die orthogonale Projektion eines Vektors auf eine lineare Hülle von Vektoren.
Geschichte: Die Methode der kleinsten Fehlerquadrate wurde von Carl Friedrich Gauß (1777-1855) und Adrien-Marie Legendre (1752-1833) unabhängig voneinander entwickelt. Legendre veröffentlichte seine Methode 1805 in seinem Buch ``Nouvelles méthodes pour la détermination des orbites des comètes''. Gauß veröffentlichte seine Methode 1809 in seinem Buch ``Theoria motus corporum coelestium in sectionibus conicis solem ambientium''. Link: Wikipedia

6.1.1. Problemstellung

Beispiel Ausgleichsgerade: Wir beginnen mit dem klassischen Beispiel des Fits einer Geraden durch n Datenpunkte.

```
import numpy as np
import matplotlib.pyplot as plt

n = 13                          # number of data points
t = np.linspace(1, 10, num = n) # time points measurements
noise = 0.4*np.random.randn(n)  # noise in measurement values
y = 2 + 0.5*t + noise           # measurement values: line + noise

plt.figure(figsize=(5, 3))
plt.plot(t, y, 'o')
plt.xlabel('t')
plt.ylabel('y')
plt.grid(True)
```



Ziel: Fit einer Geraden $y(t) = d + kt$ durch die Datenpunkte (t_i, y_i), sodass der ``Fehler'', der noch zu definieren ist, klein ist. In anderen Worten: Finde die ``optimalen'' Werte für d und k aus den Datenpunkten.

Vorgehensweise: Wir formulieren das lineare Gleichungssystem

$$\begin{aligned} d + kt_1 &= y_1 \\ d + kt_2 &= y_2 \\ \vdots &= \vdots \\ d + kt_n &= y_n \end{aligned}$$

für die zwei unbekannten Größen d und k als Matrixgleichung $Ax = b$:

$$\begin{pmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_n \end{pmatrix} \begin{pmatrix} d \\ k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

Das lineare Gleichungssystem ist typischer Weise nicht lösbar, da meist keine Gerade durch alle Datenpunkte legbar ist. Stattdessen soll der ``Fehler'' minimiert werden. Wir definieren als i-ten Einzelfehler e_i die y-Differenz zwischen dem Fit $\hat{y}_i := d + kt_i$ und dem zugehörigen Datenwert y_i, d. h. $e_i := \hat{y}_i - y_i = d + kt_i - y_i$. Der Fehlervektor e besteht aus allen n Einzelfehlern. In Matrixnotation lässt sich der Fehlervektor aus dem Fitvektor $\hat{y} = Ax$ und dem Datenvektor y, den wir allgemein als b schreiben, durch $e = \hat{y} - y = Ax - b$ berechnen. Wir verlangen, dass der Fehlervektor minimale Länge haben soll. Das heißt $\|e\| = \sqrt{e_1^2 + e_2^2 + ... + e_n^2}$ soll minimal sein. Das ist äquivalent zur Forderung, dass $\|e\|^2 = e_1^2 + e_2^2 + ... + e_n^2$ minimal sein soll, weil Quadrieren über den nicht-negativen Zahlen eine streng monotone Funktion ist. In Matrixnotation schreibt man $\|e\| = \|Ax - b\|$. Diese Vorgehensweise heißt ``Methode der kleinsten Fehlerquadrate'', andere Bezeichnungen sind ``Regression'' und ``least squares''.

6.1.2. Matrixformulierung

Formulierung des allgemeinen Optimierungsproblems: Das obige Optimierungsproblem lässt sich in Matrixform schreiben als

$$\min \|Ax - b\|.$$

Umgekehrt wird jedes Optimierungsproblem, das sich als min $\|Ax - b\|$ schreiben lässt, auch als Regressionsproblem bezeichnet.

Hinweis: Die Probleme min $\|Ax - b\|$ und min $\|Ax - b\|^2$ sind äquivalent.

Lösung in Matrixform: Falls der Rang der Matrix A maximal ist, dann hat das Regressionsproblem die eindeutige Lösung

$$\hat{x} = (A^T A)^{-1} A^T b.$$

Falls der Rang von A nicht maximal ist, dann gibt es unendlich viele Lösungen $\hat{x}$, die alle denselben optimalen Fit $A\hat{x}$ bilden. Der (bzw. ein) optimale(r) Vektor $\hat{x}$ ist auch Lösung des Gleichungssystems

$$A^T A\hat{x} = A^T b.$$

Die Gleichungen dieses Gleichungssystems heißen Normalgleichungen. Der optimale Fit ergibt sich als $\hat{y} = A\hat{x}$.

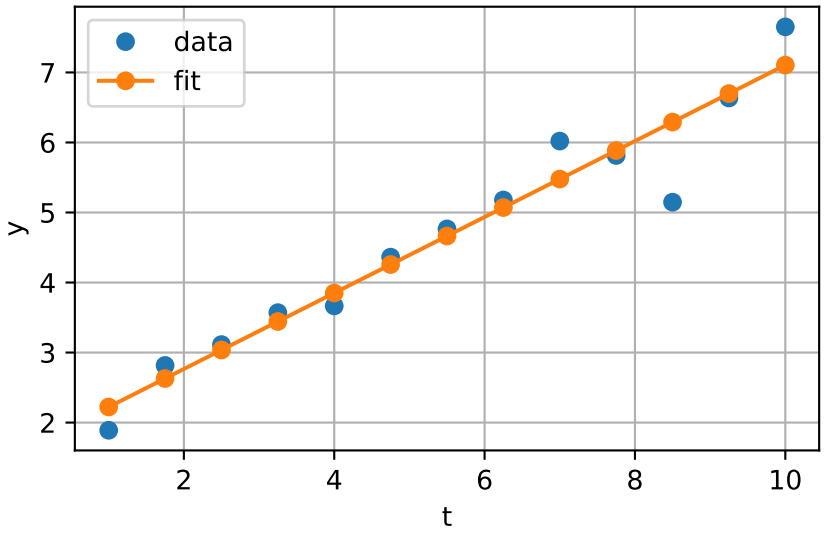
Lösungsvarianten in Python:

- über die Formel $\hat{x} = (A^T A)^{-1} A^T b$
- Lösen der Normalgleichungen mit `np.linalg.solve`
- mit dem Befehl `np.linalg.lstsq`

```
col_of_ones = np.ones(n)
A = np.stack((col_of_ones, t), axis=1)
b = y.reshape(13,1)
# print(f"{A = }")
# print(f"{b = }")

# via Formel:
x_hat_1 = np.linalg.inv(A.T @ A) @ A.T @ b
print(f"x_hat_1 = \n{x_hat_1}")

# via Normalgleichungen
x_hat_2 = np.linalg.solve(A.T @ A, A.T @ b)
print(f"x_hat_2 = \n{x_hat_2}")

# via lstsq
x_hat_3 = np.linalg.lstsq(A, b, rcond=None)[0]
print(f"x_hat_3 = \n{x_hat_3}")

# Fit A*x_hat:
y_hat = A @ x_hat_1  # same as y_hat = x_hat_1[0] + x_hat_1[1]*t

plt.figure(figsize=(5,3))
plt.plot(t, y, 'o', label='data')
plt.plot(t, y_hat, 'o-', label='fit')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid(True)
```

```
x_hat_1 =
[[1.68013423]
```

```
 [0.54249791]]
x_hat_2 =
[[1.68013423]
 [0.54249791]]
x_hat_3 =
[[1.68013423]
 [0.54249791]]
```



6.1.3. Geometrie

Orthogonale Projektion: Wir betrachten das Matrixprodukt Ax als Linearkombination der m Spalten a_i von A mit den Komponenten von x:

$$Ax = x_1 a_1 + x_2 a_2 + ... + x_m a_m$$

Diese Linearkombination der Vektoren a_i im $\mathbb{R}^n$ soll durch optimale Wahl der x_i dem Vektor $b \in \mathbb{R}^n$ möglichst nahe kommen, sodass der Differenzvektor $e = Ax - b$ eine minimale Länge hat.

Für den Fall $n = 3$ und $m = 2$ lässt sich die Situation wie in Abbildung Abbildung 6.1 darstellen:

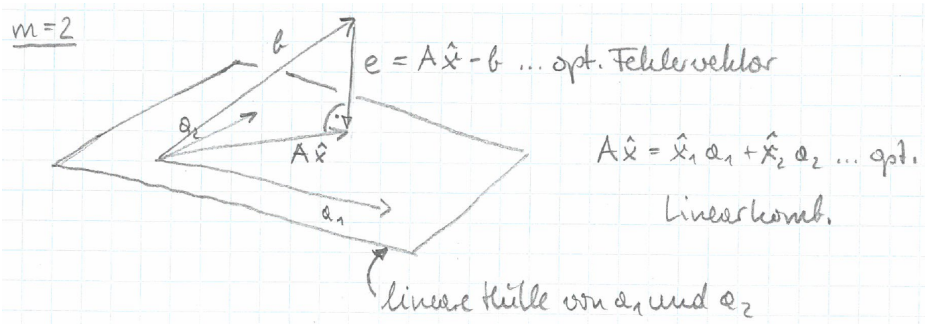


Abbildung 6.1.: Geometrie der Regression

Die Abbildung suggeriert, dass die optimale Linearkombination $\hat{y} = A\hat{x}$ der orthogonalen Projektion von b auf die von den Spalten a_i aufgespannte Ebene entspricht.

Wir zeigen nun, dass diese Intuition in allen Dimensionen n und m stimmt, sofern der Rang der Matrix A maximal ist und deshalb $A^T A$ invertierbar ist. Dazu überprüfen wir, ob der Fit $A\hat{x}$ und sein Fehler $e = A\hat{x} - b$ rechtwinklig sind, indem wir das innere Produkt der beiden berechnen:

$$\begin{aligned}(A\hat{x})^T e &= \hat{x}^T A^T (A\hat{x} - b) \\ &= \hat{x}^T A^T (A(A^T A)^{-1} A^T b - b) \\ &= \hat{x}^T A^T A (A^T A)^{-1} A^T b - \hat{x}^T A^T b \\ &= \hat{x}^T A^T b - \hat{x}^T A^T b = 0.\end{aligned}$$

Der optimale Fit $\hat{y} = A\hat{x}$ ist also orthogonal zum Fehlervektor $e = A\hat{x} - b$. Es gilt allgemein, dass die Länge des Fehlers e genau dann minimal ist, wenn der Fehler zu der von den Spaltenvektoren der Matrix A aufgespannten Hyperebene orthogonal ist. Die orthogonale Projektion von b auf diese Hyperebene ist der optimale Fit $\hat{y} = A\hat{x} = A(A^T A)^{-1} A^T b$, und die Matrix $A(A^T A)^{-1} A^T$ die zugehörige Projektionsmatrix.

Bemerkungen:

- Wenn die Spalten a_i von A linear unabhängig sind und dadurch der Rang von A maximal ist, dann ist jeder Vektor in der von den Spalten aufgespannten linearen Hülle eindeutig als Linearkombination der Spalten schreibbar. Der optimale Fit $A\hat{x}$ ist als orthogonale Projektion in diese lineare Hülle somit eindeutig als Linearkombination der Spalten schreibbar. Die Linearkombinationskoeffizienten $\hat{x}_i$, die die Lösung $\hat{x}$ bilden, sind in diesem Fall also eindeutig.

- Im Fall, dass die Spalten linear abhängig sind und dadurch der Rang von A nicht maximal ist, gibt es immer noch einen eindeutigen optimalen Fit durch die orthogonale Projektion in die lineare Hülle der Spalten, aber er ist nicht eindeutig als Linearkombination der Spalten schreibbar. Es gibt dann unendlich viele optimale $\hat{x}$, die alle denselben optimalen Fit $A\hat{x}$ bilden.

6.1.4. LGS

Für ein LGS $Ax = b$ liefert die Regressionslösung $\hat{x}$ eine Lösung des LGS, falls es mindestens eine Lösung hat. Falls das LGS keine Lösung hat, liefert die Regressionslösung $\hat{x}$ die Least-Squares-Approximation!

Zusammenfassung der Python-Methoden zur Lösung von LGS

- Bestimmung der Lösungsstruktur:
  - Für allgemeine $m \times n$ LGS $Ax = b$ kann die Funktion `np.linalg.matrix_rank` für A und $(A|b)$ verwendet werden, siehe hier.
  - Für quadratische $n \times n$ LGS $Ax = b$ kann die Funktion `np.linalg.det(A)` verwendet werden, siehe hier.
- Bestimmung der Lösungsmenge:
  - Für allgemeine $m \times n$ LGS $Ax = b$ kann die Funktion `np.linalg.lstsq(A, b, rcond=None)[0]` verwendet werden:
    - Falls genau eine Lösung existiert, wird diese zurückgegeben.
    - Falls keine Lösung existiert, wird die Least-Squares-Approximation zurückgegeben!

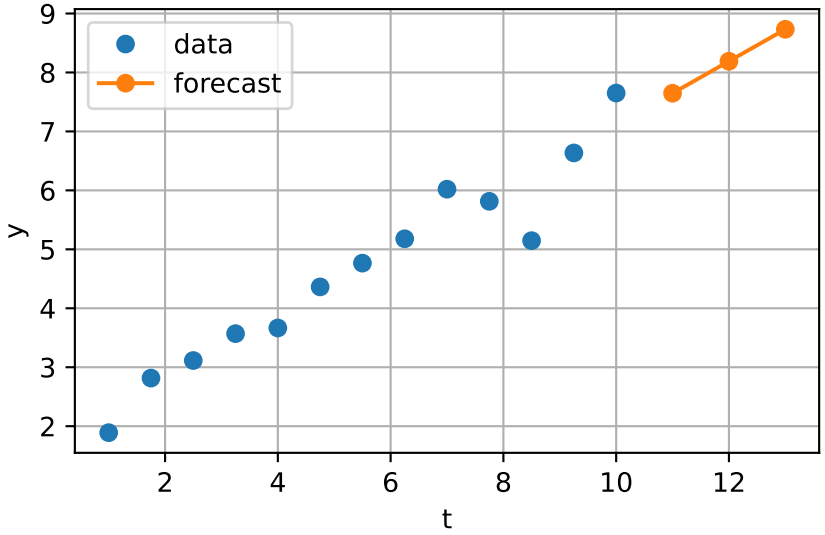
    * Falls unendlich viele Lösungen existieren, wird nur eine Lösung zurückgegeben. Die restlichen Lösungen können erzeugt werden, indem Vektoren des Nullraums von A zu dieser Lösung addiert werden. Der Befehl `sp.linalg.null_space(A)` liefert dafür eine Basis des Nullraums.
  - Für quadratische $n \times n$ LGS $Ax = b$ mit regulärer Matrix A, d. h. $\det(A) \neq 0$, liefert die Funktion `np.linalg.solve(A, b)` die zugehörige eindeutige Lösung. Auch `np.linalg.lstsq` und `np.linalg.inv(A)@b` könnten verwendet werden, aber `np.linalg.solve` ist effizienter.

6.1.5. Anwendungen

6.1.5.1. Prognosen

Regressionen werden sehr oft für Prognosen verwendet: Die Spalten der $n \times m$ Matrix A bilden dabei die jeweils n historischen Werte der m erklärenden Größen. Nach dem Fit an die n historischen Werte b der zu erklärenden Größe stehen die optimalen Koeffizienten im Vektor $\hat{x}$ zur Verfügung. Damit können neue Werte der zu erklärenden Größe berechnet werden, indem die zugehörigen neuen Werte der m erklärenden Größen mit den Koeffizienten des Vektors $\hat{x}$ multipliziert werden.

```
t_new = np.array([11, 12, 13])
col_of_ones_new = np.ones(3)
A_new = np.stack((col_of_ones_new, t_new), axis=1)
y_new = A_new @ x_hat_1  # same as y_new = x_hat_1[0] + x_hat_1[1]*t_new

plt.figure(figsize=(5,3))
plt.plot(t, y, 'o', label='data')
plt.plot(t_new, y_new, 'o-', label='forecast')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid(True)
```



6.1.5.2. Polynomialer Fit

Verallgemeinerung der Ausgleichsgerade als Polynom 1. Ordnung auf z. B. Polynome 3. Ordnung mit gesuchten Koeffizienten c_0, c_1, c_2 und c_3:

$$\begin{aligned} c_0 + c_1 t_1 + c_2 t_1^2 + c_3 t_1^3 &= y_1 \\ c_0 + c_1 t_2 + c_2 t_2^2 + c_3 t_2^3 &= y_2 \\ \vdots &= \vdots \\ c_0 + c_1 t_n + c_2 t_n^2 + c_3 t_n^3 &= y_n \end{aligned}$$

als Matrixgleichung $Ax = b$:

$$\begin{pmatrix} 1 & t_1 & t_1^2 & t_1^3 \\ 1 & t_2 & t_2^2 & t_2^3 \\ \vdots & \vdots & & \\ 1 & t_n & t_n^2 & t_n^3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Die optimalen Koeffizienten sind Lösung des Regressionsproblems $\min \|Ax - b\|$.

6.1.5.3. Transformationen

Beispiel: Das exponentielle Wachstumsmodell

$$n(t) = \alpha^{t-t_0}$$

mit unbekannten Parametern α und t_0 kann durch Logarithmieren zu einem linearen Modell

$$\begin{aligned} \log(n(t)) &= (t - t_0)\log(\alpha) \\ \log(n(t)) &= -t_0 \log(\alpha) + \log(\alpha) t \end{aligned}$$

transformiert werden. Das resultierende lineare Modell kann als Ausgleichsgerade mittels Regression behandelt werden. Dabei entspricht $d = -t_0 \log(\alpha)$ und $k = \log(\alpha)$.

6.1.5.4. Quadratische Zielfunktionen

Jede quadratische, skalare Zielfunktion $g(x)$ mit $x \in \mathbb{R}^n$ kann (bis auf einen für die Optimierung irrelevanten additiven Term) in der Regressionsform $\|Ax - b\|^2$ geschrieben und somit mittels Regression behandelt werden.

Beispiel:

$$\begin{aligned} g(x_1, x_2) =& 13x_1^2 - 30x_1x_2 + 18x_2^2 + 14x_1 - 12x_2 + 10 \\ =& \left\| \begin{pmatrix} 2 & -3 \\ -3 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - \begin{pmatrix} 1 \\ 3 \end{pmatrix} \right\|^2 \end{aligned}$$

6.1.6. Python

Die zentrale Python-Funktion ist `numpy.linalg.lstsq`. Sie liefert einen Vektor zurück, der als ersten Eintrag $\hat{x}$ enthält. Die weiteren Einträge sind für uns nicht relevant.

Für die Beispiele und Aufgaben verwenden wir folgende Python-Bibliotheken, siehe Kapitel Python Tutorial:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

6.2. Beispiele

6.2.1. LGS 1

Nicht-quadratisches, inhomogenes LGS mit unendlich vielen Lösungen. Partikuläre Lösung mit `lstsq`:

```
A = np.array([[ 1,-3, 1],
              [-2, 0, 5]])
b = np.array([[ 0],
              [-7]])
Ab = np.hstack((A, b))
print("Rang von A  =", np.linalg.matrix_rank(A))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
print("Anzahl der Variablen =", np.shape(A)[1])
print("Daher: unendlich viele Lösungen.")

# Eine Lösung erhält man mit lstsq:
x = np.linalg.lstsq(A, b, rcond=None)[0]
print("Eine Lösung ist x = \n{}".format(x))
# Check, dass x eine Lösung ist
print("Ax - b =\n", A@x - b)

# Alle Lösungen erhält man aus x + beliebige Linearkombination der Spalten von N:
r = np.linalg.matrix_rank(A)
N = sp.linalg.null_space(A)
```

```
# Check, dass die Spalten von N mit A multipliziert Null ergeben:
print("AN =\n", A@N)
```

```
Rang von A  = 2
Rang von Ab = 2
Anzahl der Variablen = 3
Daher: unendlich viele Lösungen.
Eine Lösung ist x =
[[ 0.56451613]
 [-0.20322581]
 [-1.17419355]]
Ax - b =
 [[ 4.44089210e-16]
 [-1.77635684e-15]]
AN =
 [[3.33066907e-16]
 [4.44089210e-16]]
```

6.2.2. LGS 2

Nicht-quadratisches, inhomogenes LGS ohne Lösung. `lstsq` liefert eine Least-Squares-Approximation:

```
A = np.array([[ 1,-3],
              [-2, 0],
              [ 5, 1]])
b = np.array([[ 0],
              [-7],
              [ 1]])
Ab = np.hstack((A, b))
print("Rang von A  =", np.linalg.matrix_rank(A))
print("Rang von Ab =", np.linalg.matrix_rank(Ab))
print("Daher: keine Lösung.")

# Achtung : Das mit lstsq berechnete x ist keine Lösung!
x = np.linalg.lstsq(A, b, rcond=None)[0]
print("lstsq liefert x = \n{}".format(x))
print("Ax - b =\n", A@x - b)
```

```
Rang von A  = 2
Rang von Ab = 3
Daher: keine Lösung.
lstsq liefert x =
[[ 0.63513514]
 [-0.02702703]]
Ax - b =
 [[0.71621622]
 [5.72972973]
 [2.14864865]]
```

6.2.3. LGS 3

Das nicht-quadratische LGS mit eindeutiger Lösung. Diese kann mit `lstsq` berechnet werden:

```
A = np.array([[ 1,-3],
              [-2, 0],
              [ 5, 1]])
b = np.array([[ 4],
              [-2],
              [ 4]])
Ab = np.hstack((A, b))
rank_A = np.linalg.matrix_rank(A)
rank_Ab = np.linalg.matrix_rank(Ab)
(m, n) = np.shape(A)

print("Rang von A  =", rank_A)
print("Rang von Ab =", rank_Ab)
print("Anzahl der Variablen =", n)
print("Daher: genau eine Lösung.")

x = np.linalg.lstsq(A, b, rcond=None)[0]
print("eindeutige Lösung ist x = \n{}".format(x))
# Check, dass x eine Lösung ist
print("Ax - b =\n", A@x - b)
```

```
Rang von A  = 2
Rang von Ab = 2
Anzahl der Variablen = 2
Daher: genau eine Lösung.
eindeutige Lösung ist x =
[[ 1.]
 [-1.]]
Ax - b =
 [[-3.10862447e-15]
 [ 0.00000000e+00]
 [ 8.88178420e-16]]
```

6.2.4. Globale Erwärmung

Laden Sie mit dem Befehl die Datei Global_Temperatures_2024.csv in Python.

```
D = np.genfromtxt('daten/Global_Temperatures_2024.csv', delimiter=',')
```

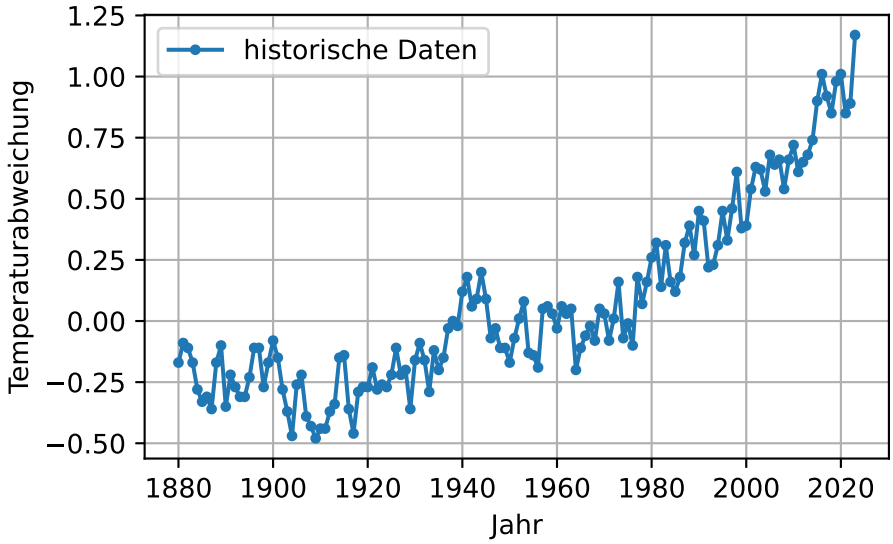
Die Werte stammen von der NASA-Homepage GISS Surface Temperature Analysis. Sie repräsentieren jährliche Mittel der Abweichungen der Oberflächentemperaturen vom Mittel der Jahre 1951 bis 1980. Die erste Spalte enthält die Jahre, die zweite die jährlichen Mittelwerte, die dritte die 5-jährlichen Mittelwerte.

1. Fitten Sie mittels der Methode der kleinsten Fehlerquadrate (Reggression) mindestens zwei sinnvolle Funktionen durch die jährlichen Mittelwerte. Stellen Sie Ihre Ergebnisse auch grafisch dar.

2. Benutzen Sie Ihre mindestens zwei Modelle, um die jährliche mittlere Temperaturabweichung für die nächsten 30 Jahre vorherzusagen. Stellen Sie Ihre Ergebnisse auch grafisch dar.

Lösung:

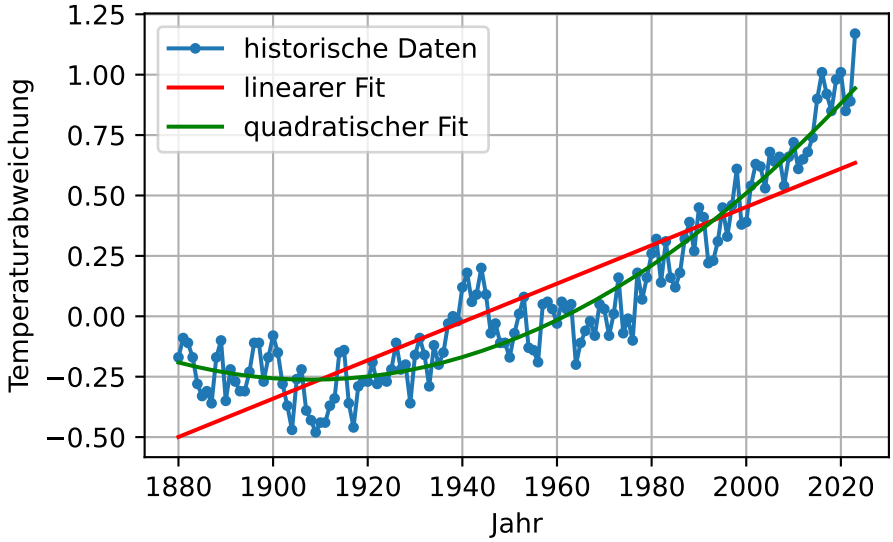
```
D = np.genfromtxt('daten/Global_Temperatures_2024.csv', delimiter=',')
t = D[:,0]
T = D[:,1]
n = len(t)

plt.figure(figsize=(5, 3))
plt.plot(t, T, '.-', label='historische Daten')
plt.xlabel('Jahr')
plt.ylabel('Temperaturabweichung')
plt.legend(numpoints=1, loc='best')
plt.grid(True)
```



Linearer (Gerade) und quadratischer Fit via Least-Squares:

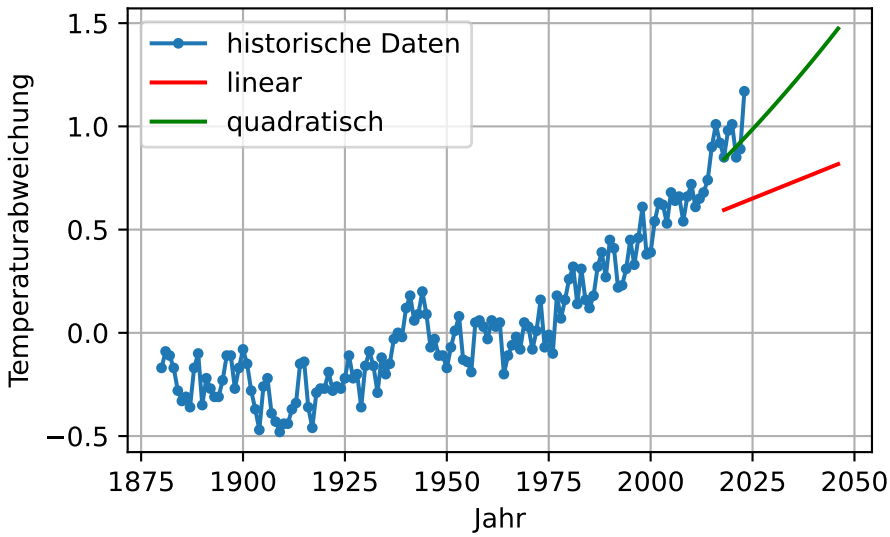
```
A = np.column_stack( (np.ones((n, 1)), t) )
x_linear = np.linalg.lstsq(A, T, rcond=None)[0]
y_fit_linear = A@x_linear

# quadratisch:
A = np.column_stack( (np.ones((n, 1)), t, t**2) )
x_quadr = np.linalg.lstsq(A, T, rcond=None)[0]
y_fit_quadr = A@x_quadr

plt.figure(figsize=(5, 3))
plt.plot(t, T, '.-', label='historische Daten')
plt.plot(t, y_fit_linear, '-r', label='linearer Fit')
plt.plot(t, y_fit_quadr , '-g', label='quadratischer Fit')
plt.xlabel('Jahr')
```

```
plt.ylabel('Temperaturabweichung')
plt.legend(loc='best', numpoints=1)
plt.grid(True)
```



Prognosen:

```
t_ = np.arange(2018, 2047)
n_ = len(t_)

# linear:
A_ = np.column_stack( (np.ones((n_, 1)), t_) )
y_prog_linear = A_ @ x_linear

# quadratisch:
A_ = np.column_stack( (np.ones((n_, 1)), t_, t_**2) )
y_prog_quadr = A_ @ x_quadr

plt.figure(figsize=(5, 3))
plt.plot(t, T, '.-', label='historische Daten')
plt.plot(t_, y_prog_linear, '-r', label='linear')
plt.plot(t_, y_prog_quadr , '-g', label='quadratisch')
plt.xlabel('Jahr')
plt.ylabel('Temperaturabweichung')
plt.legend(numpoints=1, loc = 'best')
plt.grid(True)
```



6.2.5. Prognose von Krebs

Das Breast Cancer Wisconsin (Diagnostic) Data Set enthält Daten von 569 Patientinnen, die Sie aus der Datei wdbc.csv mit dem folgenden Code laden.

```
data = np.genfromtxt('daten/wdbc.csv', delimiter=',')
X = data[:, 2:]
b = data[:, 1]
```

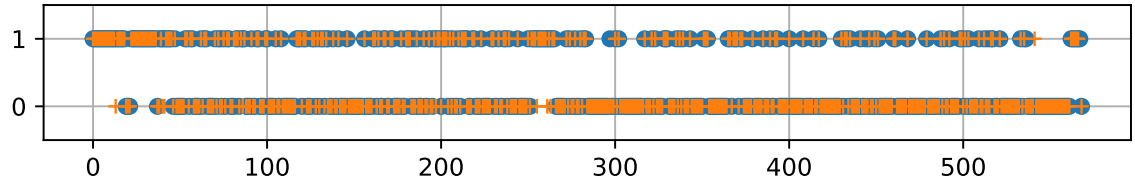
Die 30 Spalten der Matrix X enthalten die Werte von 30 Features, die Aufschluss über die Gut- oder Bösartigkeit des Krebs jeder Patientin (=Zeile) geben können. Die Einträge des Vektors b sind 1 für Patientinnen mit bösartigem Krebs und 0 für Patientinnen mit gutartigem Krebs.

1. Erweitern Sie die Matrix X um eine Spalte mit Einsen zu einer Matrix A.
2. Verwenden Sie eine lineare Regression $Ax \sim b$, um die Gut- bzw. Bösartigkeit des Krebs aller Patientinnen zu fitten. Die gefitteten Werte werden nicht genau 0 oder 1 sein. Ändern Sie diese plausibek auf 0 oder 1 nach der Regression ab.
3. Stellen Sie das Ergebnis grafisch dar, berechnen Sie den Anteil der richtig klassifizierten Patientinnen.

```
n = len(b)
col_of_ones = np.ones((n, 1))
A = np.hstack((col_of_ones, X))

x_hat = np.linalg.lstsq(A, b, rcond=None)[0]
b_hat_raw = A@x_hat   # raw predicted values
b_hat = np.round(b_hat_raw) # round to 0 or 1

plt.figure(figsize=(8, 1))
plt.plot(b, 'o')     # ture values
plt.plot(b_hat, '+') # predicted values
plt.ylim(-0.5, 1.5)
```

```
plt.grid(True)

# accuray (=the fraction of correctly classified samples)
accuracy = sum(b_hat == b)/n
print(f"{accuracy = :.2}")
```

```
accuracy = 0.96
```



In der Praxis wird aus einem Trainingsdatensatz A_{train} und b_{train}, der nur einen Teil der Patientinnendaten enthält, ein Modell $A_{\text{train}}\hat{x} \sim b_{\text{train}}$ gefittet (=gelernt). Dieses Modell wird verwendet, um aus den restlichen Daten A_{test} Prognosen $A_{\text{test}}\hat{x}$ von b_{test} zu erstellen, die dann mit den echten Werten b_{test} verglichen werden. Wieso tut man das?

6.3. Aufgaben

6.3.1. Verständnisfragen und kurze Aufgaben

1. Formulieren Sie das lineare Gleichungssystem $Ax = b$ für den Fit einer quadratischen Funktion durch 10 Datenpunkte.
2. Warum hat eine Regression genau eine Lösung? Hinweis: Argumentieren Sie mit der Geometrie der Regression.
3. Beschreiben Sie die Rechenschritte zum Fit einer Parabel durch 10 Datenpunkte.
4. Erläutern und skizzieren Sie die Vorgehensweise zum Fitten einer Geraden durch eine Datenwolke mittels Regression.
5. Mittels Regression sollen die Koeffizienten eines Polynoms 2. Grades berechnet werden, das die Datenpunkte $(1, 1)$, $(2, 4)$, $(3, 12)$ und $(4, 22)$ möglichst genau annähert. Wie lauten die Matrix A und die Vektoren x und b des Minimierungsproblems min. $||Ax-b||$? Die Koeffizienten selbst müssen nicht berechnet werden.
6. Die Datenpunkte aus Aufgabe 5 sollen mit einer Exponentialfunktion der Form $y = c \cdot e^{at}$ angenähert werden. Transformieren Sie die Funktion, und formulieren Sie anschließend erneut die Matrix A und die Vektoren x und b des Minimierungsproblems min. $||Ax - b||$.

Ergebnisse:

1. Siehe Polynomialer Fit.
2. Weil die Regression einer orthogonalen Projektion entspricht.
3. Siehe Polynomialer Fit.
4. Siehe Methoden.
5. Siehe Polynomialer Fit.

6. Siehe Transformationen. Transformation der Exponentialfunktion $y = c \cdot e^{at}$ zur linearen Funktion $\log(y) = \log(c) + at$. Das lineare Gleichungssystem $Ax = b$ für den Fit einer Exponentialfunktion durch die Datenpunkte (t_i, y_i) lautet

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} \log(c) \\ a \end{pmatrix} = \begin{pmatrix} \log(1) \\ \log(4) \\ \log(12) \\ \log(22) \end{pmatrix}.$$

6.3.2. Polynomiale Regression

Laden Sie mit folgenden Python-Befehlen die Daten der Datei Polyfit.csv in ein array `D`, dessen Spalten als Daten-Vektoren t und y weiterverwendet werden:

```
D = np.genfromtxt('daten/Polyfit.csv', delimiter=';')
y = D[:, 1]
t = D[:, 0]
```

1. Plotten Sie y gegen t.
2. Verwenden Sie die Methode der kleinsten Fehlerquadrate (Regression, Python-Befehl `lstsq`), um ein Polynom vom Grad 4 durch die Punktwolke zu fitten. Erzeugen Sie ein Stamm-Blatt-Diagramm der Koeffizienten. Warum sind die ungeraden Koeffizienten null?
3. Plotten Sie die ursprünglichen Datenpunkte und den polynomialen Fit in eine Grafik.
4. Benutzen Sie die Befehle `numpy.polyfit` und `numpy.poly1d` um die gleichen Resultate zu erzielen.

6.3.3. Mooresches Gesetz

Mehr Information zum Beispiel unter Wikipedia: Mooresches Gesetz.

1. Die Datei Moores_Law.csv enthält die Anzahlen n der Transistoren in 13 Mikroprozessoren und die Jahre derer Einführung t als Spalten. Laden Sie die Datei mit den folgenden Python-Befehlen

```
D = np.genfromtxt('daten/Moores_Law.csv', delimiter=';')
t = D[:, 0]
n = D[:, 1]
```

und plotten Sie n gegen t mit den `matplotlib` Befehlen `plot` und `semilogy`. 2. Erklären Sie, wie die Methode der kleinsten Fehlerquadrate (Regression) verwendet werden kann, um α und t_0 zu bestimmen, so dass

$$n_i \approx \alpha^{t_i - t_0} \text{ for } i = 1, ..., 13$$

gilt. 3. Lösen Sie das Least-Squares-Problem (d.h. die Regressionsaufgabe) in Python mit dem Befehl `lstsq`. Vergleichen Sie Ihre Resultate mit dem Mooreschen Gesetz, das behauptet, dass sich die Anzahl der Transistoren in Mikroprozessoren alle 2 Jahre verdoppelt.

6.3.4. Müllmengen

Die Müllmenge in Millionen Tonnen pro Tag des Landes Lower Slobbovia von 1960 bis 1995 ist in folgender Tabelle wiedergegeben.

Jahr t	1960	1965	1970	1975	1980	1985	1990	1995
Menge y	86.0	99.8	135.8	155.0	192.6	243.1	316.3	469.5

1. Formulieren Sie das OLS-Problem in Matrixform für den besten Fit einer Gerade durch die Datenpunkte.
2. Formulieren Sie das OLS-Problem in Matrixform für den besten Fit eines exponentiellen Wachstumsmodells $y = ce^{\alpha t}$ durch die Datenpunkte.
3. Plotten Sie die Daten, und begründen Sie, welches Modell (linear oder exponentiell) besser geeignet ist.
4. Lösen Sie beide OLS-Probleme, und stellen Sie die Ergebnisse grafisch dar.

6.3.5. Ohmsches Gesetz

Sie haben in einer Messreihe die Spannung U und den Strom I an einem Ohmschen Widerstand gemessen und die folgenden Daten erhalten:

I (A)	0.11	0.15	0.27	0.40	0.50	0.54	0.63	0.89	0.99	1.10
U (V)	1.10	2.08	2.94	4.39	4.87	5.99	7.70	8.30	8.50	10.03

1. Stellen Sie die Daten in Python grafisch dar.
2. Schätzen Sie den Ohmschen Widerstand R, indem Sie mit dem Ohmschen Gesetz $U = R \cdot I$ und einer linearen Regression, die den quadratische U-Fehler minimiert, den optimalen Widerstandswert bestimmen.
3. Schätzen Sie nun den Widerstand R mit einer linearen Regression, die den quadratische I-Fehler minimiert.
4. Vergleichen Sie die beiden Schätzungen. Welche Methode ist besser? Interpretieren Sie die Situation geometrisch. Unter welchen Bedingung liefern die beiden Methoden den selben Schätzwert für R?

Teil II.

Operations Research

7. Überblick

Operations Research (OR) ist ein Teilgebiet der angewandten Mathematik und beschäftigt sich mit der Anwendung mathematischer Methoden zur Lösung von Entscheidungsproblemen - und davon gibt es sehr viele!

Anwendungsgebiete sind unter anderem:

- Ressourcenmanagement
- Logistik
- Produktion
- Verkehr
- Energie
- Finanzen
- Militär
- Spieltheorie

Problemklassen TODO: Entscheidungsprobleme und Optimierungsprobleme: Struktur von Freiheiten und Ziel -> Mathematische Modellierung, Klassifizierung von Optimierungsproblemen

asas

Von den vielen Teilgebieten des OR behandeln wir in diesem Kurs einführend die

- Lineare Optimierung: Englisch *linear programming*, kurz LP
- gemischt-ganzzahlige lineare Optimierung: Englisch *mixed integer linear programming* (MILP) oder *integer (linear) programming* (IP bzw. ILP)

8. Linear Programming

8.1. Methoden

8.1.1. Modellierungsbeispiel

Das Beispiel *Butter und Eiscreme* stammt aus dem Buch *Ferris, Mangasarian, Wright: Linear Programming with MATLAB. SIAM (Society for Industrial and Applied Mathematics), 2008.*

Problemstellung: Ein Bauer hat 3 Kühe, die in Summe 22 Gallonen (1 Gallone $\simeq$ 3.785 Liter) Milch pro Woche geben. Aus der Milch kann er Eiscreme und Butter machen. Er braucht 2 Gallonen Milch für 1 kg Butter und 3 Gallonen Milch für 1 Gallone Eiscreme. Es gibt keine Lagerrestriktionen für Butter. Er kann maximal 6 Gallonen Eiscreme lagern. Er hat 6 Arbeitsstunden pro Woche für die Herstellung zur Verfügung. Für 4 Gallonen Eiscreme benötigt er 1 Stunde, für 1 kg Butter benötigt er ebenfalls 1 Stunde. Die gesamte Produktion kann er zu folgenden Preisen verkaufen (vollständiger Absatz): 5 USD pro Gallone Eiscreme, 4 USD pro kg Butter. Wie viele Gallonen Eiscreme und wie viele kg Butter soll er herstellen, sodass er seinen Profit maximiert?

Modellierung: Entscheidungsvariablen:

- x_1: produzierte Gallonen Eiscreme
- x_2: produzierte kg Butter

Zielfunktion: Maximiere $5x_1 + 4x_2$.

Nebenbedingungen:

- Lagerung von Eiscreme: $x_1 \leq 6$
- Arbeitszeit: $\frac{1}{4}x_1 + x_2 \leq 6$
- verfügbare Milch: $3x_1 + 2x_2 \leq 22$
- Positivität der Variablen: $x_1 \geq 0, x_2 \geq 0$

Matrixform:

$$\begin{aligned} \max\ 5x_1 + 4x_2 \\ \text{s.t. } x_1 &\leq 6 \\ \frac{1}{4}x_1 + x_2 &\leq 6 \\ 3x_1 + 2x_2 &\leq 22 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

8.1.2. Matrixform

Ein lineares Programm (LP) besteht aus

- einer linearen Zielfunktion, die maximiert oder minimiert wird und
- linearen Nebenbedingungen, d. h. linearen Ungleichungen und linearen Gleichungen.

Durch Multiplikation der Zielfunktion mit -1 wird eine Maximierung zu einer Minimierung. Eine $\geq$-Ungleichung wird durch Multiplikation mit -1 zu einer $\leq$-Ungleichung. Daher kann ein LP immer in folgender Form geschrieben werden:

$$\begin{aligned} \min\ & c_1x_1 + c_2x_2 + ... + c_nx_n \\ \text{s. t. } & A_{11}x_1 + A_{12}x_2 + ... + A_{1n}x_n \leq b_1 \\ & \vdots \leq \vdots \\ & A_{m1}x_1 + A_{m2}x_2 + ... + A_{mn}x_n \leq b_m \\ & G_{11}x_1 + G_{12}x_2 + ... + G_{1n}x_n = h_1 \\ & \vdots = \vdots \\ & G_{p1}x_1 + G_{p2}x_2 + ... + G_{pn}x_n = h_p \end{aligned}$$

Mit den entsprechenden Vektoren und Matrizen lässt sich das LP komfortabler als

$$\begin{aligned} \min\ & c^Tx \\ \text{s. t. } & Ax \leq b \\ & Gx = h \end{aligned}$$

schreiben. Die Vektoren c, b und h und die Matrizen A und B enthalten die vollständige Information des LP und können daher als Schnittstellenformat zu Solvern dienen. Wir werden den SciPy-Solver linprog und den in PulP enthaltenen Solver CbC verwenden, siehe Implementierung und PulP.

8.1.3. Lineare Funktionen

Definierende Eigenschaften: Eine lineare Funktion von $\mathbb{R}^n$ nach $\mathbb{R}$ ist gegeben durch das innere Produkt eines fixen Vektors c mit einem Variablenvektor x:

$$f(x) = c^Tx = c_1x_1 + c_2x_2 + ... + c_nx_n.$$

Jede lineare Funktion erfüllt die Linearitätseigenschaft

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y),$$

und jede Funktion, die die Linearitätseigenschaft erfüllt, ist von der Form $f(x) = c^Tx$.

Geometrische Darstellung:

- Die Konturlinien einer linearen Funktion in der Ebene sind parallele Geraden, und die Null-Konturlinie geht durch den Ursprung.
- Die Konturflächen einer linearen Funktion im Raum sind parallele Ebenen, und die Null-Konturfläche geht durch den Ursprung.

Der Koeffizientenvektor c der linearen Funktion $f(x) = c^T x$ ist orthogonal (=rechtwinklig) zu den Konturlinien bzw. Konturflächen.

Beispiel: Konturplot In Abbildung 8.1 ist ein Konturplot der linearen Funktion

$$f : \mathbb{R}^2 \to \mathbb{R} : f(x) = -2x_1 + 3x_2$$

dargestellt. Der Koeffizientenvektor ist $c = \begin{pmatrix} -2 \\ 3 \end{pmatrix}$, d. h. $f(x) = c^T x$.

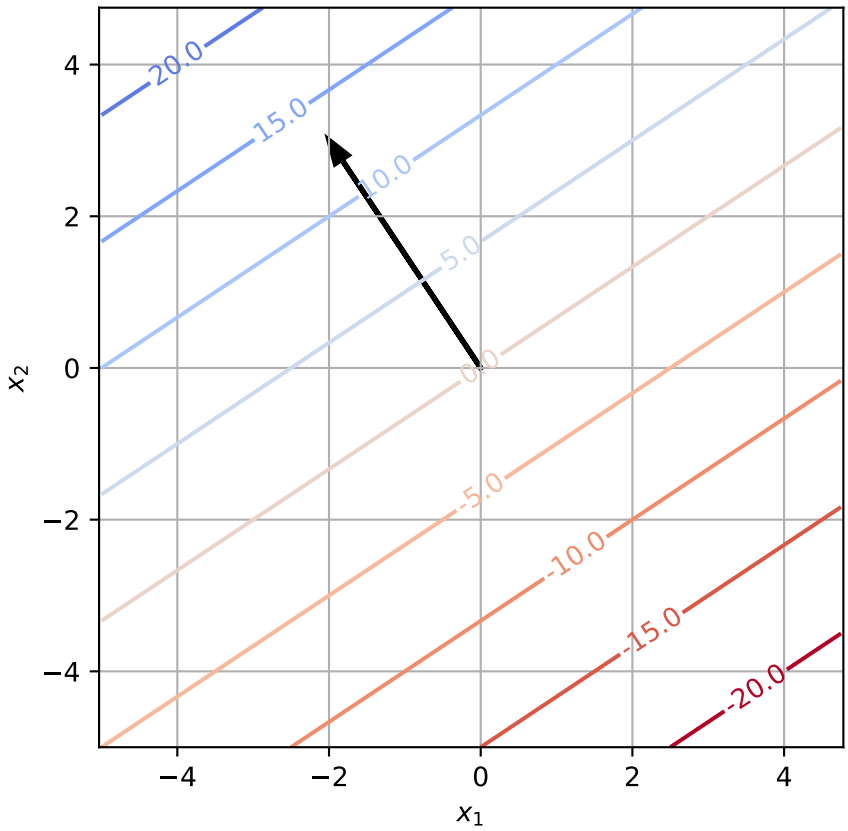


Abbildung 8.1.: Konturlinien einer linearen Funktion

8.1.4. Lineare (Un-)Gleichungen

Eine **lineare Gleichung**

$$c_1 x_1 + c_2 x_2 + ... + c_n x_n = b$$

entspricht der Konturlinie/ebene/etc., allg. **Hyperebene**, $c^T x = b$ der linearen Funktion $f(x) = c^T x$.

Eine **lineare Ungleichung**

$$c_1 x_1 + c_2 x_2 + ... + c_n x_n \leq b$$

definiert in $\mathbb{R}^n$ einen **Halbraum** und kann mittels der linearen Funktion $f(x) = c^T x$ als $c^T x \leq b$ geschrieben werden.

8.1.5. Lösungsstruktur

Wie im Abschnitt Matrixform beschrieben, kann jedes LP mit entsprechenden Vektoren und Matrizen in Vektor-/Matrixform als

$$\begin{aligned} \min\ & c^T x \\ \text{s. t. } & Ax \leq b \\ & Gx = h \end{aligned}$$

geschrieben werden.

8.1.5.1. Begriffe

- Eine **Entscheidung** x des **Entscheidungsraums** $\mathbb{R}^n$ wird **zulässiger Punkt (zulässige Entscheidung, zulässiger Entscheidungsvektor)** genannt, wenn sie alle Nebenbedingungen (hier $Ax \leq b$ und $Gx = h$) erfüllt.
- Der **zulässige Bereich** eines LP ist die Menge aller zulässigen Punkte.
- Der **optimale Wert** eines LP ist das Optimum (Minimum bzw. Maximum) der Zielfunktion unter den Nebenbedingungen. Er kann auch $-\infty$ oder ∞ sein.
- Ein **optimaler Punkt** (eine **Lösung**) eines LP ist ein zulässiger Punkt, der den optimalen Wert erreicht.
- Die **optimale Menge** eines LP ist die Menge aller optimalen Punkte des LP. **Ungleichungsform von LPs:**

Die Formulierung min. $c^T x$ unter $Ax \leq b$ und $Gx = h$ kann man noch weiter vereinfachen, indem die vorkommenden Gleichungen zu $\leq$ Ungleichungen umgeschrieben werden. Zum Beispiel entspricht die Gleichung $3x_1 - 4x_2 = 5$ den beiden Ungleichungen $3x_1 - 4x_2 \leq 5$ und $3x_1 - 4x_2 \geq 5$, von denen die zweite durch Multiplikation mit -1 auch zu einer $\leq$ Ungleichung gemacht wird. Dadurch kann die Gleichung durch die zwei $\leq$ Ungleichungen $3x_1 - 4x_2 \leq 5$ und $-3x_1 + 4x_2 \leq -5$ ersetzt werden.

Jedes LP lässt sich somit auch in der Ungleichungsform

$$\begin{aligned} \min\ & c^T x \\ \text{s. t. } & Ax \leq b \end{aligned}$$

schreiben.

8.1.5.2. Geometrie

- Die Zielfunktion $c^T x$ eines LP ist eine lineare Funktion, deren Konturlinien **Hyperebenen** im Entscheidungsraum $\mathbb{R}^n$ sind.
- Die Matrixungleichung $Ax \leq b$ besteht aus übereinandergestapelten linearen Ungleichungen, die im Entscheidungsraum $\mathbb{R}^n$ jeweils **Halbräume** definieren. Der zulässige Bereich des LP, d. h. die Menge aller Entscheidungen x, die die Bedingung $Ax \leq b$ erfüllen, entspricht somit dem Durchschnitt all dieser Halbräume. Der Durchschnitt von Halbräumen wird **Polyeder** genannt.

Bemerkungen: Ein Polyeder ist *konvex*, d. h. für zwei beliebiege Punkte des Polyeders liegt deren Verbindungslinie vollständig im Polyeder: $\forall x, y \in P = \{x | Ax \leq b\}$ und $\forall \lambda$ mit $0 \leq \lambda \leq 1$ ist $\lambda x + (1-\lambda)y \in P$. Der Ausdruck $\lambda x + (1-\lambda)y$ mit $0 \leq \lambda \leq 1$ wird konvexe Kombination von x und y genannt und ergibt einen Punkt auf der Verbindungslinie zwischen x und y.

8.1.5.3. Fälle

Ein LP fällt in einen der folgenden fünf Fälle.

- Fall A: Es gibt eine eindeutige Lösung an einer Ecke des Polyeders, also genau einen optimalen Punkt. Der optimale Wert ist endlich.
- Fall B: Alle Punkte auf einer endlichen Kante des Polyeders sind optimale Punkte. Es gibt somit ∞-viele Lösungen, die konvexen Kombinationen der optimalen Ecken bilden die optimale Mengen. Der optimale Wert ist endlich.
- Fall C: Alle Punkte auf einer unendlichen Kante des Polyeders sind optimale Punkte. Es gibt somit ∞-viele Lösungen, die keine konvexe Kombinationen von optimalen Ecken sind. Der optimale Wert ist endlich.
- Fall D: Das Polyeder ist leer. Es gibt keine zulässigen Punkte. Die Nebenbedingungen schließen sich aus. Das LP heißt **unzulässig** (engl. infeasible). Der optimale Wert eines Minimierungs-LP ist $+\infty$, jener eines Maximierungs-LP ist $-\infty$.
- Fall E: Das Polyeder ist unbeschränkt in Richtung der Zielfunktion. Das LP heißt **unbeschränkt** (engl. unbounded). Der optimale Wert des Minimierungs-LP ist $-\infty$, jener eines Maximierungs-LP ist $+\infty$.

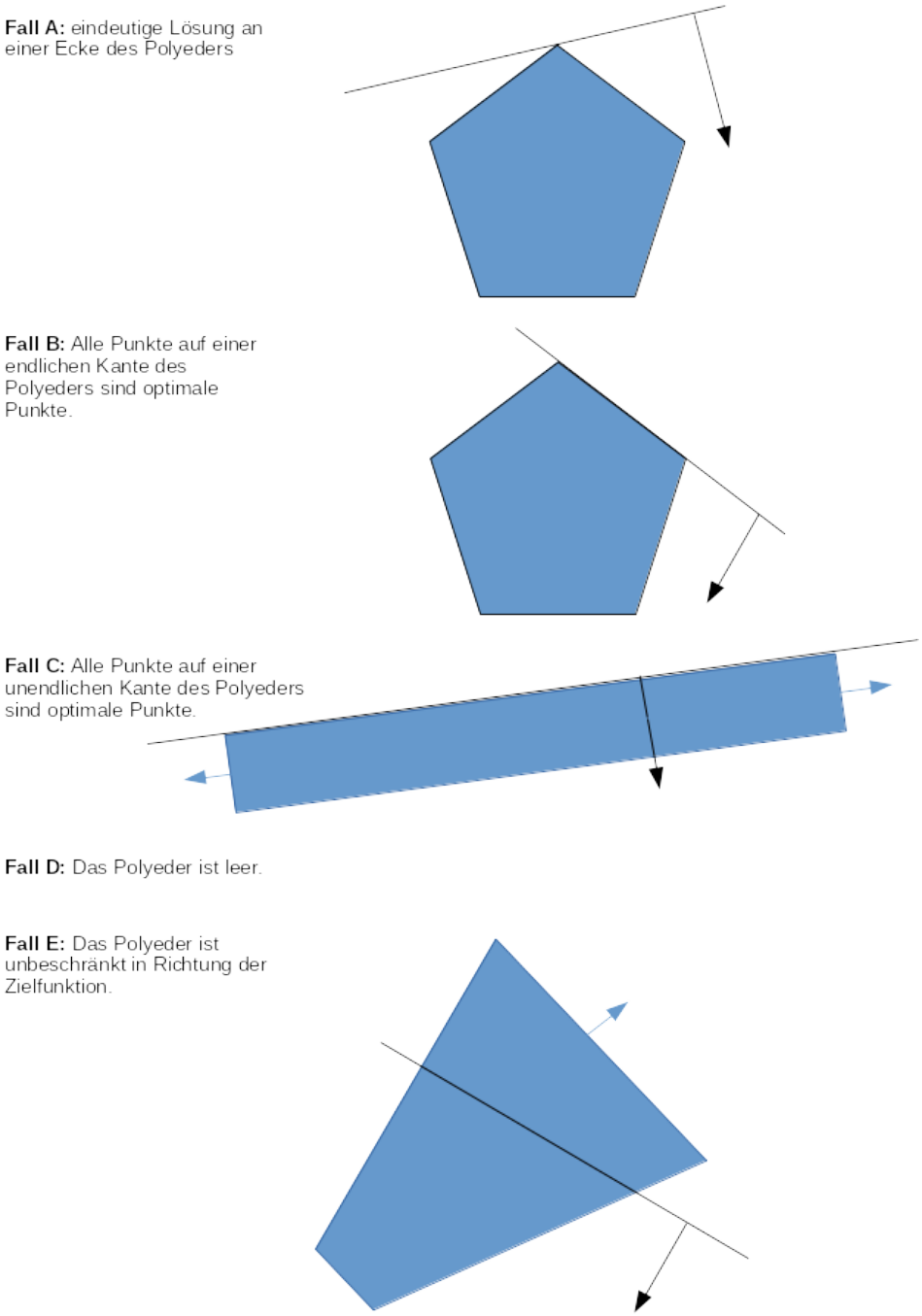


Abbildung 8.2.: LP Loesungsstruktur

Zusammenfassung für Minimierungs-LP:

LP	Polyeder	optimaler Wert	optimale Menge
unzulässig	leer	$+\infty$	leer
unbeschränkt	unbeschränkt in Richtung der Zielfunktion	$-\infty$	leer
andernfalls	andernfalls	endlich	besteht aus einer oder ∞-vielen Lösungen

Vergleich mit Status-Rückgabewerten des SciPy LP-Solvers `linprog`:

```
status : An integer representing the exit status of the optimization:
    0 : Optimization terminated successfully.
    1 : Iteration limit reached.
    2 : Problem appears to be infeasible.
    3 : Problem appears to be unbounded.
    4 : Numerical difficulties encountered.
```

8.1.5.4. Ecken-Theorem

Falls das Polyeder der Nebenbedingungen mindestens eine Ecke hat und der optimale Wert endlich ist, dann gibt es zumindest eine Ecke des Polyeders, die ein optimaler Punkt ist.**

Folgerung: Daher könnte man so vorgehen, dass man zuerst alle Ecken bestimmt und dann jene Ecke(n) mit dem besten Zielfunktionswert als optimale(n) Punkt(e) identifiziert. Diese Vorgehensweise ist für große LP aber nicht effizient.

Simplexalgorithmus: Der Simplexalgorithmus löst ein LP, indem von einer Ecke zu einer nächsten Ecke gesprungen wird, deren Zielfunktionswert besser ist, bis dies nicht mehr möglich ist und dadurch eine Lösung gefunden wurde.

8.1.6. Implementierung

8.1.6.1. Grafisch

Wir setzen das obige Modellierungsbeispiel fort und betrachten zuerst den Zulässigkeitsbereich und die Konturlinien der Zielfunktion. Aus der Abbildung 8.3 erkennen wir, dass wir im Fall A sind. Die eindeutige Lösung lautet $(4, 5)$, und der optimale Wert ist 40. Die optimale Menge besteht aus genau einem Punkt, nämlich $(4, 5)$.

Nun modellieren lösen wir das LP mit dem Solver `linprog` aus dem Paket `SciPy` und anschließend mit dem Paket `PuLP` und dem darin enthaltenen Solver CbC.

8.1.6.2. SciPy

Wir ``füttern'' den Solver `linprog` mit den Daten des Problems, d. h. mit den Vektoren und Matrizen, die die Zielfunktion und die Nebenbedingungen eindeutig beschreiben:

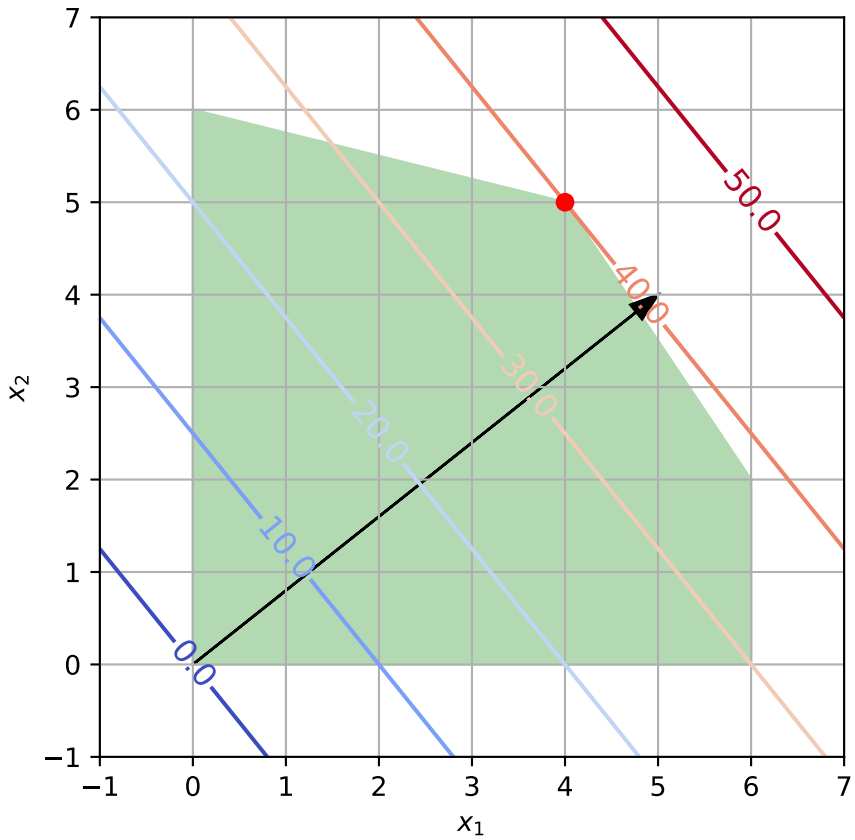


Abbildung 8.3.: Zulässigkeitsbereich und Konturlinien der Zielfunktion

```python
from scipy.optimize import linprog

c = np.array([-5, -4])      # negativ für Minimierung
A = np.array([[   1,  0],   # Lager
              [0.25,  1],   # Zeit
              [   3,  2]])  # Milch
b = np.array([6, 6, 22])
res = linprog(c, A_ub=A, b_ub=b) # by default bounds are (0, None)!
print("Resultate gesamt:\n", res)
print("optimaler Wert:", -res.fun)
print("optimaler Punkt:", res.x)
```

```
Resultate gesamt:
         message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -40.0
              x: [ 4.000e+00  5.000e+00]
            nit: 2
          lower:  residual: [ 4.000e+00  5.000e+00]
                 marginals: [ 0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf]
                 marginals: [ 0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 2.000e+00  0.000e+00  0.000e+00]
                 marginals: [-0.000e+00 -8.000e-01 -1.600e+00]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
optimaler Wert: 40.0
optimaler Punkt: [4. 5.]
```

Varianten und ihre Solverantworten:

Fall B: Alle Punkte auf einer endlichen Kante des Polyeders sind optimale Punkte.

```python
c = np.array([-3, -2])   # Zielfunktion parallel zur Milch-Nebenbedingung
A = np.array([[   1,  0],
              [0.25,  1],
              [   3,  2]])
b = np.array([6, 6, 22])
res = linprog(c, A_ub=A, b_ub=b)
print(res)

# optimaler Wert des zurückgegebenen optimalen Punktes:
print(-np.dot(c, res.x))

# (4,5) ist auch optimaler Punkt:
y = np.array([4, 5])
print(-np.dot(c, y))
```

```
# konvexe Kombination:
lbd = 0.123
print(-np.dot(c, lbd*res.x + (1-lbd)*y))
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: -22.0
              x: [ 6.000e+00  2.000e+00]
            nit: 1
          lower:  residual: [ 6.000e+00  2.000e+00]
                 marginals: [ 0.000e+00  0.000e+00]
          upper:  residual: [       inf        inf]
                 marginals: [ 0.000e+00  0.000e+00]
          eqlin:  residual: []
                 marginals: []
        ineqlin:  residual: [ 0.000e+00  2.500e+00  0.000e+00]
                 marginals: [-0.000e+00 -0.000e+00 -1.000e+00]
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
22.0
22
22.0
```

Fall D: Das Polyeder ist leer. Der optimale Wert eines Minimierungs-LP ist $+\infty$.

```
# Ungleichheitszeichen aller Nebenbedingungen umdrehen:
c = np.array([-5, -4])
A = np.array([[   -1,  0],
              [-0.25,  -1],
              [   -3,  -2]])
b = np.array([-6, -6, -22])
res = linprog(c, A_ub=A, b_ub=b, bounds=(None, 0))  # bounds geändert
print(res)
```

```
        message: The problem is infeasible. (HiGHS Status 8: model_status is Infeasible; primal_sta
        success: False
         status: 2
            fun: None
              x: None
            nit: 0
          lower:  residual: None
                 marginals: None
          upper:  residual: None
                 marginals: None
          eqlin:  residual: None
                 marginals: None
        ineqlin:  residual: None
                 marginals: None
```

Fall E: Das Polyeder ist unbeschränkt in Richtung der Zielfunktion. Der optimale Wert des Minimierungs-LP ist $-\infty$.

```
c = np.array([5, 4])  # Vorzeichen geändert
A = np.array([[   1,  0],
              [0.25,  1],
              [   3,  2]])
b = np.array([6, 6, 22])

res = linprog(c, A_ub=A, b_ub=b, bounds=(None, 0))  # bounds geändert
print(res)
```

```
       message: The problem is unbounded. (HiGHS Status 10: model_status is Unbounded; primal_statu
       success: False
        status: 3
           fun: None
             x: None
           nit: 0
         lower:  residual: None
                marginals: None
         upper:  residual: None
                marginals: None
         eqlin:  residual: None
                marginals: None
       ineqlin:  residual: None
                marginals: None
```

8.1.6.3. PuLP

PuLP ist eine Python-Bibliothek zur benutzerfreundlichen Modellierung und Lösung von LPs, siehe PuLP.

```
import pulp

prob = pulp.LpProblem("Butter_Eiscreme", pulp.LpMaximize)
x1 = pulp.LpVariable("x1", 0, 6)     # 0 <= x1 <= 6, Lager
x2 = pulp.LpVariable("x2", 0, None)  # 0 <= x2
prob += 5*x1 + 4*x2        # Zielfunktion
prob += 0.25*x1 + x2 <= 6  # Arbeitszeit
prob += 3*x1 + 2*x2 <= 22  # Milch
prob.solve()  # Lösen des LP mit Cbc Solver
print("Status:", pulp.LpStatus[prob.status])
print("optimaler Wert:", pulp.value(prob.objective))
print("optimaler Punkt:", x1.varValue, x2.varValue)
```

```
Welcome to the CBC MILP Solver
Version: 2.10.3
Build Date: Dec 15 2019

command line - /home/kr/.local/lib/python3.11/site-packages/pulp/solverdir/cbc/linux/64/cbc /tmp/56
```

```
At line 2 NAME          MODEL
At line 3 ROWS
At line 7 COLUMNS
At line 14 RHS
At line 17 BOUNDS
At line 19 ENDATA
Problem MODEL has 2 rows, 2 columns and 4 elements
Coin0008I MODEL read with 0 errors
Option for timeMode changed from cpu to elapsed
Presolve 2 (0) rows, 2 (0) columns and 4 (0) elements
0  Obj -0 Dual inf 8.9999998 (2)
0  Obj -0 Dual inf 8.9999998 (2)
3  Obj 40
Optimal - objective value 40
Optimal objective 40 - 3 iterations time 0.002
Option for printingOptions changed from normal to all
Total time (CPU seconds):       0.00   (Wallclock seconds):       0.01

Status: Optimal
optimaler Wert: 40.0
optimaler Punkt: 4.0 5.0
```

8.1.7. Schattenpreise

8.1.7.1. Definition

Der Schattenpreis einer Nebenbedingung ist die Änderung des optimalen Wertes bei Lockerung der Nebenbedingung um eine Einheit.

8.1.7.2. SciPy

```
c = np.array([-5, -4])
A = np.array([[   1,  0],    # Lager
              [0.25,  1],    # Zeit
              [   3,  2]])   # Milch
b = np.array([6 + 1, 6, 22]) # Lager um 1 erhöht
res = linprog(c, A_ub=A, b_ub=b)
print(-res.fun)
c = np.array([-5, -4])
A = np.array([[   1,  0],    # Lager
              [0.25,  1],    # Zeit
              [   3,  2]])   # Milch
b = np.array([6, 6 + 1, 22]) # Zeit um 1 erhöht
res = linprog(c, A_ub=A, b_ub=b)
print(-res.fun)
c = np.array([-5, -4])
A = np.array([[   1,  0],    # Lager
              [0.25,  1],    # Zeit
              [   3,  2]])   # Milch
```

```
b = np.array([6, 6, 22 + 1]) # Milch um 1 erhöht
res = linprog(c, A_ub=A, b_ub=b)
print(-res.fun)
```

```
40.0
40.8
41.6
```

8.1.7.3. PuLP

Um die Schattenpreise mit PuLP zu berechnen, verwenden wir eine ausführlichere Formulierung als bisher, vgl. das Beispiel im Abschnitt PuLP.

```
prob = pulp.LpProblem("Butter_Eiscreme", pulp.LpMaximize)

x1 = pulp.LpVariable("x1", 0, None)  # 0 <= x1
x2 = pulp.LpVariable("x2", 0, None)  # 0 <= x2

prob += 5*x1 + 4*x2        # Zielfunktion

constraint_lager = x1 <= 6
prob += constraint_lager, "Lager"

constraint_zeit = 0.25*x1 + x2 <= 6
prob += constraint_zeit, "Zeit"

constraint_milch = 3*x1 + 2*x2 <= 22
prob += constraint_milch, "Milch"

print(prob)

prob.solve(pulp.COIN(msg=False))
print(f"Schattenpreis Lager: {constraint_lager.pi}")
print(f"Schattenpreis Zeit:  {constraint_zeit.pi}")
print(f"Schattenpreis Milch: {constraint_milch.pi}")
```

```
Butter_Eiscreme:
MAXIMIZE
5*x1 + 4*x2 + 0
SUBJECT TO
Lager: x1 <= 6

Zeit: 0.25 x1 + x2 <= 6

Milch: 3 x1 + 2 x2 <= 22

VARIABLES
x1 Continuous
x2 Continuous
```

```
Schattenpreis Lager: -0.0
Schattenpreis Zeit:  0.8
Schattenpreis Milch: 1.6
```

8.2. Beispiele

TODO: Beispiele aus Aufgaben?

TODO: Maximaler Fluss -> Praxisbeispiel

TODO: Schattenpreise

8.3. Aufgaben

TODO: Aufgaben von ETW-AM 3: Aufgaben, Kurztestfragen, Programmierprojekte?, Lösungen in PuLP und linprog angeben, Aufgaben zu Schattenpreisen (erweitern)

8.3.1. Verständnisfragen und kurze Aufgaben

1. Ein LP-Solver liefert als Rückgabe die Meldung *The LP is infeasible (Anm. deutsch: unzulässig).* Was schließen Sie über den zulässigen Bereich des LP?
2. Ein LP-Solver liefert als Rückgabe die Meldung *Optimal value = 123.45.* Wie viele Lösungen hat das LP?
3. Erklären Sie den Begriff Schattenpreis.
4. Ein LP-Solver liefert als Rückgabe die Meldung *The LP is unbounded (Anm. deutsch: unbeschränkt).* Was schließen Sie über den zulässigen Bereich des LP?
5. Erklären Sie die Begriffe ``Polyeder'' und ``optimale Menge eines LP''?
6. Bringen Sie das LP max. $3x_1+4x_2$ s. t. $x_1+x_2=1$ und $2x_1-3x_2 \geq 6$ in die Ungleichungsform min. $c^T x$ s. t. $Ax \leq b$, d. h., bestimmen Sie c, A und b.
7. Bringen Sie das LP max. x_1+x_2 s. t. $3x_1-x_2 \geq 1$ und $2x_1+x_2 \leq 6$ in die Ungleichungsform min. $c^T x$ s. t. $Ax \leq b$, d. h., bestimmen Sie c, A und b.
8. Welche mögliche Lösungsstrukturen (Anzahl der Lösungen und zugehöriger optimaler Wert) haben lineare Optimierungen min $c^T x$ s. t. $Ax \leq b$?
9. Ein LP-Solver liefert als Rückmeldung *The LP is unbounded (deutsch: unbeschränkt).* Welche Aussage können Sie über den zulässigen Bereich treffen, und was ist der optimale Wert des linearen Optimierungsproblems bei einer Minimierung der Zielfunktion?
10. Wie viele Lösungen kann ein lineares Optimierungsproblem mit Nebenbedingungen $Ax \leq b$ haben?
11. Ein Mobilfunkbetreiber möchte seinen monatlichen Gewinn durch einen neuen Handytarif maximieren. Zum Tarif *Speedy* um 10 EUR/Monat kommt der Tarif *Basic* um 5 EUR/Monat dazu. Wie lautet die Zielfunktion des linearen Optimierungsproblems bei einer Formulierung als Minimierungsproblem min. $c^T x$?
12. Für beide Tarife aus Aufgabe 11) wird ein monatliches Datenvolumen von 5 GB angeboten. Die vertraglich zugesicherte Downloadgeschwindigkeit beträgt beim Tarif *Speedy* 30 Mbit/s und beim Tarif *Basic* 5 Mbit/s. Beim Tarif *Speedy* sind darüber hinaus 4 GB EU-Datenroaming inkludiert. Der Mobilfunkbetreiber kann ingesamt ein monatliches Datenvolumen von maximal 1000 GB, eine maximale Downloadgeschwindigkeit von 2000 Mbit/s sowie bis zu 240 GB

EU-Datenroaming bereitstellen. Formulieren Sie sämtliche Nebenbedingungen des linearen Optimierungsproblems in der Form $Ax \leq b$.

13. Lösen Sie das in den Aufgaben 11) und 12) definierte lineare Optimierungsproblem graphisch. Zeichnen Sie den Punkt ein, bei dem sich für den Mobilfunkbetreiber der maximale monatliche Gewinn ergibt. Sollten Sie Aufgabe 4) nicht lösen können, lösen Sie in dieser Aufgabe stattdessen das lineare Optimierungsproblem, das durch das nachfolgende Ungleichungssystem und die Zielfunktion aus Aufgabe 3) gegeben ist:

$$\begin{pmatrix} 0 & 2 \\ 15 & 2 \\ -1 & 0 \\ 0 & -1 \\ 40 & 40 \end{pmatrix} x \leq \begin{pmatrix} 350 \\ 1000 \\ 0 \\ 0 \\ 8000 \end{pmatrix}$$

8.3.2. 2D-Beispiel 1

Zulässigkeitsbereich, Konturlinien, optimaler Wert, optimale Punkte, Implementierung mit `linprog`:

1. Zeichnen Sie auf Papier den Zulässigkeitsbereich für folgende Nebenebedingungen $x_1 + 2x_2 \leq 6$, $2x_1 + x_2 \leq 6$, $x_1 \geq 0$, $x_2 \geq 0$.
2. Zeichnen Sie zusätzlich die Konturlinien der Profitfunktion $f(x) = 2x_1 + 4x_2$.
3. Ermitteln Sie grafisch den optimalen Wert und alle optimalen Punkte des LP, die die Profitfunktion unter den Nebenbedingungen maximieren.
4. Überprüfen Sie das Ergebnis mit `linprog`.

8.3.3. 2D-Beispiel 2

Zulässigkeitsbereich, Konturlinien, optimaler Wert, optimale Punkte, Implementierung mit `linprog`:

1. Zeichnen Sie auf Papier den Zulässigkeitsbereich für folgende Nebenebedingungen $-2x_1 + x_2 \leq 1$, $x_1 + 3x_2 \geq 2$, $x_1 \geq 0$, $x_2 \geq 0$
2. Zeichnen Sie zusätzlich die Konturlinien der Kostenfunktion $f(x) = 3x_1 + 2x_2$.
3. Ermitteln Sie grafisch den optimalen Wert und alle optimalen Punkte des LP, die die Kostenfunktion unter den Nebenbedingungen minimieren.
4. Überprüfen Sie das Ergebnis mit `linprog`.

8.3.4. Förster

Dieses Beispiel stammt aus dem Buch *Linear Programming* von Vasek Chvatal. Ein Förster hat 100 Hektar Wald mit Hartholz und hat folgende zwei Möglichkeiten.

- Option 1: Hartholz fällen und anschließende Regeneration, Kosten 10 EUR/ha, späterer Erlös 50 EUR/ha
- Option 2: Hartholz fällen und anschließend mit Pinien aufforsten, Kosten 50 EUR/ha, späterer Erlös 120 EUR/ha

Die daraus resultierenden späteren Gewinne sind somit 40 EUR/ha und 70 EUR/ha. Er hat momentan 4000 EUR zur Verfügung und fragt sich, wie viele der 100 Hektar er mit welcher Option bewirtschaften soll, um seinen Gewinn zu maximieren.

1. Formulieren Sie das lineare Problem (LP) der Profitmaximierung.
2. Lösen Sie das LP mit graphischen Mitteln durch händisches Rechnen.
3. Lösen Sie das LP in Python mit `linprog` und `pulp`.

8.3.5. Produktionsplanung

Gegeben seien 2 Produkte A und B, die aus dem Rohstoffen R_1, R_2, und R_3 hergestellt werden können. Untenstehende Tabelle zeigt die für die Produktion nötigen Mengen zur Herstellung einer Einheit des jeweiligen Produkts sowie die zur Verfügung stehende Menge an Rohstoffen.

Rohstoff	Produkt A	Produkt B	verfügbare Menge
R_1	5	10	3000
R_2	0	3	750
R_3	4	2	1200

Der erwirtschaftete Profit für eine Einheit des Produkts A ist 200 EUR und 300 EUR für Produkt B.

1. Formulieren Sie das lineare Problem (LP) der Profitmaximierung.
2. Lösen Sie das LP graphisch.
3. Lösen Sie das LP in Python mit `linprog` und mit PuLP.

8.3.6. Kupfer-Zinn Legierung

Drei Klassen Altmetall werden geschmolzen, um 200 kg Kupfer-Zinn Legierung zu erhalten.

- Klasse 1 enthält 80 % Kupfer, 20 % Zinn und kostet EUR 6 pro kg.
- Klasse 2 enthält 40 % Kupfer, 60 % Zinn und kostet EUR 3 pro kg.
- Klasse 3 enthält 10 % Kupfer, 90 % Zinn und kostet EUR 4 pro kg.

Die Legierung soll maximal 60 % Kupfer und 50 % Zinn enthalten.

1. Formulieren Sie das lineare Problem (LP) der Kostenminimierung.
2. Lösen Sie das LP in Python mit `linprog` und mit PuLP.

8.3.7. Fluss in einem Netzwerk

Das folgende Beispiel stammt aus dem Buch *Matousek, Gärtner: Understanding and Using Linear Programming. Springer 2007, section 2.2 p. 14ff.*

Betrachten Sie den Graphen in der Abbildung unten, der beispielsweise ein Energie- oder Informationsnetzwerk modellieren könnte. Er hat die Knoten s (Quelle), a, b, c, d, e und t (Ziel). Die Knoten a, b, c, d und e können keine Energie bzw. Information zwischenspeichern. Die Kanten haben die angegeben Kapazitäten (= maximale Transferraten).

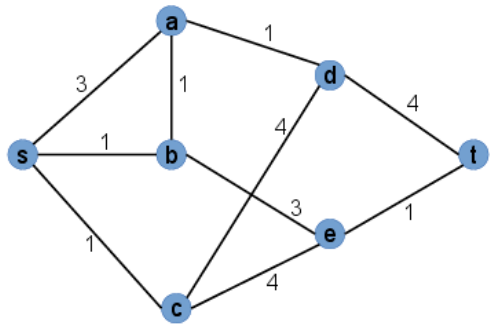


Abbildung 8.4.: Maximaler Fluss

Berechnen Sie in Python mit Hilfe eines LP die maximale Energie- bzw. Informationsübertragungsrate vom Knoten s zum Knoten t.

8.3.8. Tierfuttermischung

Sie sollen die optimalen Mengen von drei Zutaten in einer Tierfuttermischung bestimmen. Das Endprodukt hat mehrere Nährstoffbedingungen zu erfüllen. Die möglichen Zutaten, ihre Nährstoffe (in kg Nährstoff pro kg Zutat) und die Zutatenpreise (in EUR pro Kilogramm) sind in der folgenden Tabelle dargestellt.

Zutat	Kalzium	Eiweiß	Balaststoffe	Preis
Kalk	0.380	0.00	0.00	10.0
Getreide	0.001	0.09	0.02	30.5
Sojamehl	0.002	0.50	0.08	90.0

Die Mischung muss die folgenden Bedingungen erfüllen:

- Kalzium: mindestens 0.8 %, aber nicht mehr als 1.2 %
- Eiweiß: mindestens 8 %
- Balaststoffe: höchstens 5 %

Das Problem besteht darin, jene Zusammensetzung der Futtermittelmischung zu finden, die die Nebenbedingungen erfüllt und die Kosten minimiert. Formulieren Sie diese Problem als LP in der Form:

$$\begin{aligned} &\min c^T x \\ &\text{s. t. } Ax \leq b \\ &\qquad Gx = h. \end{aligned}$$

8.3.9. Energiespeicher

Sie haben 24 Stunden Zeit, um einem Kunden 300 kWh Energie zur Verfügung zu stellen, indem Sie ihm einen anfänglich leeren Energiespeicher befüllen. Die maximale Einspeisleistung beträgt 20 kW, und Energie kann nicht aus dem Speicher zurück ins Energieversorgungsnetz geführt werden. Die Energiepreise sind für jede Stunde durch $c_1, c_2, ..., c_{24}$ gegeben. Der Speicher hat einen Ladewirkungsgrad von 90 %, d. h. 90 % der in den Speicher eingespeisten Energie kann der Kunde tatsächlich nutzen.

Formulieren Sie das Optimerungsproblem, das die gesamtkostenoptimalen Einspeiseenergien für jede Stunde liefert.

8.3.10. Separation von Erzen

Erz aus drei verschiedenen Minen wird von einer Firma vor dem Weitertransport in drei unterschiedliche Güteklassen separiert. Die täglichen Produktionskapazitäten der Minen und ihre täglichen Betriebskosten sind in der folgenden Tabelle angeführt.

Mine	Tonnen/Tag hohe Güte	Tonnen/Tag niedrige Güte	Betriebskosten/Tag in 1000 EUR
1	4	4	20
2	6	4	22
3	1	6	18

Die Firma muss mindestens 54 Tonnen Erz hoher Güte und mindestens 65 Tonnen Erz niedriger Güte pro Woche liefern. Gesucht sind die Betriebstage pro Woche für jede Mine, sodass die Firma ihre Lieferversprechen bei minimalen Kosten einhält.

1. Formulieren Sie dieses lineare Problem in Matrixform.
2. Lösen Sie das Problem in Python mit `linprog` und mit PuLP.

Quelle: Schaum' Outlines: Operations Research, Aufgabe 1.5, S. 5

9. Mixed Integer LP

9.1. Methoden

TODO: Verweis auf Literatur Sioshansi Branch and Bound Cutting Planes, Implementierung mit SciPy und PuLP, keine Schattenpreise

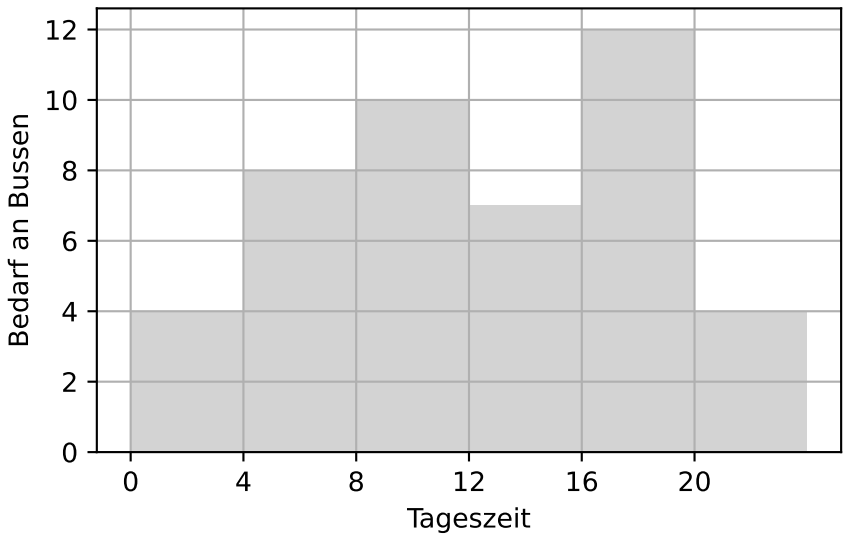
9.2. Beispiele

TODO: Beispiele

9.2.1. Personaleinsatzplanung

Betrachten wir ein Busunternehmen, das eine optimale Personaleinsatzplanung erstellen möchte. Der Bedarf an Bussen variiert, wie in folgender Abbildung dargestellt, aufgrund der Kundennachfrage von Stunde zu Stunde:

```
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt

periods = np.arange(0, 24, 4)
buses = np.array([4, 8, 10, 7, 12, 4])

plt.figure(figsize=(5, 3))
plt.bar(periods, buses, width=4,
  color='lightgray', align='edge')
plt.xticks(periods)
plt.xlabel('Tageszeit')
plt.ylabel('Bedarf an Bussen')
plt.grid(True)
```



Beispielsweise müssen vier Busse von Mitternacht bis 4 Uhr im Einsatz sein, während acht Busse von 4 bis 8 Uhr fahren müssen. Wir gehen davon aus, dass die Nachfrage jeden Tag die gleiche ist.

Sie sollen bestimmen, wie viele Fahrer für jede Startzeit bei Bedarfsdeckung einzuplanen sind. Fahrer arbeiten in 8-Stundenschichten, die um 0, 4, 8, 12, 16 oder 20 Uhr beginnen. Zum Beispiel kann ein Fahrer ab 0 Uhr einen Bus bis 8 Uhr fahren. Ein Fahrer, der um 20 Uhr zu arbeiten beginnt, fährt die letzten vier Stunden des Tages und die ersten vier Stunden des nächsten Tags. Das Ziel besteht darin, die Anzahl der eingesetzten Fahrer und somit die Kosten zu minimieren. Beachten Sie, dass, obwohl ein Fahrer für einen Zeitraum von acht Stunden eingesetzt werden kann, es nicht erforderlich ist, dass er für den gesamten Zeitraum einen Bus fährt. Er könnte auch für vier Stunden innerhalb seines Zeitraums pausieren.

Eine zulässige Entscheidung für dieses Problem besteht darin, 8 Fahrer zum Zeitpunkt 0, 10 Fahrer zum Zeitpunkt 8 und 12 Fahrer zum Zeitpunkt 16 einzusetzen. Diese Entscheidung deckt alle Anforderungen ab und verwendet insgesamt 30 Fahrer, ist aber nicht optimal. Das Problem besteht darin, die kleinst mögliche Anzahl an Fahrern zu finden.

Lösen Sie das LP in Python.

9.3. Aufgaben

TODO: Aufgaben

Teil III.

Computer Tools

10. Python

10.1. Einleitung

Motivation

Wenn man es mit komplexen Modellen, vielen Daten, oder nur numerisch lösbaren Problemen zu tun hat, verwendet man in der Mathematik sehr gerne den Computer (deutsch: Rechner, to compute = berechnen) in Verbindung mit einer passenden Software. Aber auch zum Visualisieren und zum Überprüfungen von Handrechnungen ist der Computer sehr hilfreich.

Warum Python für wissenschaftliches Rechnen? Python ist eine freie, offene, plattformunabhängige, üblicherweise interpretierte, höhere Programmiersprache. Python Code ist gut lesbar und einfach zu lernen. Es gibt sehr viele Pakete und Einsatzgebiete für Python, siehe PyPI - the Python Package Index.

Für den Bereich des wissenschaftlichen Rechnens sind insbesondere folgende **Python-Pakete** nützlich:

- NumPy: NumPy bietet Unterstützung für große, mehrdimensionale Arrays (Vektoren, Matrizen, …) sowie eine große Sammlung von high-level mathematischen Funktionen, um mit diesen Arrays zu arbeiten.
- Matplotlib: eine Grafikbibliothek, die jener von Matlab sehr ähnlich ist
- SciPy: SciPy enthält Module für Optimierung, lineare Algebra, Integration, Interpolation, spezielle Funktionen, Signal- und Bildverarbeitung, Solver für gewöhnliche Differentialgleichungen und andere in Wissenschaft und Technik übliche Aufgaben.

Python erfreut sich einer großen und wachsenden Beliebtheit und besitzt daher eine umfangreiche und breitgefächert **Community**. So ziemlich jedes Problem mit Lösung/en findet man z. B. unter stackoverflow. Der Name bezieht sich übrigens auf die englische Komikergruppe Monty Python.

Zu den **Alternativen** im Bereich wissenschaftliches Rechnen zählen

- Julia: frei, offen und plattformunabhängig
- R: frei, offen und plattformunabhängig
- Matlab: kommerziell
- Mathematica: kommerziell

Im Folgenden finden Sie:

- Informationen zur Installation von Python auf Ihrem Computer,
- Informationen zum Online-Jupyter-Hub der FHV,
- kurze Einführungen in JupyterLab und in jene Python-Kenntnissen, die wir in dieser Lehrveranstaltung verwenden.

10.2. FHV-Jupyter-Hub

Unter jupyter.labs.fhv.at können Sie sich mit Ihrem FHV-Account einloggen und Python in Jupyter Notebooks verwenden. Verwenden Sie zum Einloggen Ihre FHV-Emailadresse ohne den Domänenteil `@students.fhv.at` als Benutzernamen. Von zu Hause müssen Sie zuerst eine VPN-Verbindung zur FHV aufbauen, siehe FHV-Inside/VPN.

10.3. Lokale Installation

Python und Python-Pakete können auf unterschiedliche Weisen lokal installiert werden. Wir beschreiben hier die von vielen empfohlene Installation der Python Distribution Anaconda:

10.3.1. Anaconda

Um die umfangreiche Anaconda Distribution zu installieren, folgen Sie der Dokumentation entsprechend Ihrem Betriebssystem. Sparsamer ist die minimale Version Miniconda, die jedoch nur Terminal-basiert ist. Anaconda ist ein Manager für Umgebungen (engl. environments). Neben Python können auch weitere Sprachen damit installiert werden, was aber für diesen Kurs nicht notwendig ist.

10.3.2. Environments

Die Verwendung von Umgebungen wird empfohlen, um Konflikte zwischen verschiedenen Versionen von Python und Python-Paketen zu vermeiden, ist jedoch für diesen Kurs nicht zwingend erforderlich.

Anstatt den Anaconda Navigator zur Installation einer Umgebung zu verwenden, empfehlen wir die Verwendung der Kommandozeile in einem *Anaconda Prompt* oder *Miniconda Prompt*. Hier ist ein Beispiel: Um eine Umgebung namens `LAOR` zu erstellen, die Python in Version 3.11 und eine Reihe zusätzlicher Pakete enthält, geben Sie den folgenden Befehl im Prompt ein:

```
conda create -n LAOR python=3.11 ipython jupyterlab numpy matplotlib
```

Aktivieren Sie die Umgebung mit `conda activate MarketModelling` und starten Sie JupyterLab mit `jupyter lab`. Deaktivieren Sie die Umgebung nach Abschluss mit `conda deactivate`. Weitere Informationen zu Umgebungen und Conda finden Sie in der Conda-Cheatsheet. Die folgenden Pakete werden für den Kurs benötigt:

- ipython
- jupyterlab
- numpy
- matplotlib

10.4. JupyterLab

Es gibt, grob gesagt, zwei Typen von **Entwicklungsumgebungen**:

- Ein Text-Editor kombiniert mit der Kommandozeile (englisch: command prompt, console, terminal). Entwicklungsumgebungen wie Spyder, Visual Studio Code oder PyCharm kombinieren beides und mehr in einer Applikation.
- Wir verwenden die web-basierte, interaktive, literate programming Entwicklungsumgebung JupyterLab, die neben Python noch viele andere Programmiersprachen mittels Jupyter-Notebooks unterstützt. Im Wikipedia-Eintrag zum Projekt Jupyter steht (Stand 2020-06-18) unter anderem: ``Das Jupyter Notebook hat sich als Benutzeroberfläche für Cloud Computing verbreitet. Große Cloud-Anbieter haben angepasste Tools für Cloud-Anwender entwickelt. Beispiele dafür sind Amazon SageMaker, Googles Colaboratory und Microsofts Azure Notebook.''

Wenn Sie sich unter jupyter.labs.fhv.at einloggen, verwenden Sie bereits JupyterLab. Auf Ihrem Computer starten Sie JupyterLab via dem Anaconda Navigator oder einfacher über eine Kommandozeile mit dem Befehl `jupyter lab`.

Das meiste der browserbasierten Oberfläche ist selbsterklärend. Wenn Sie ein (neues) **Jupyter Notebook** (Dateiendung .ipynb) starten, wird ein **Kernel** gestartet. Der Kernel führt Ihre Python Befehle aus und beinhaltet alle verwendeten Objekte (Variablen, Funktionen, Pakete etc.).

> **Achtung**
>
> Verwenden Sie generell für Ordner- und und Dateinamen keine Umlaute, keine Sonderzeichen und keine Leerzeichen!

Ein Jupyter-Notebook besteht aus einer Liste von **Zellen** (engl. cells). Zellen können im `Command Mode` oder im `Edit Mode` bearbeitet werden. Die wichtigsten zwei Zelltypen sind `Code` und `Markdown`.

Code Zellen:

- Wir schreiben unseren Python Code in die Code-Zellen eines Jupyter-Notebooks. Zum **Ausführen des Codes einer Code-Zelle** drücken Sie `Strg+Return` oder `Shift+Return`. Bei der ersten Variante bleibt der Fokus auf der Code-Zelle, bei der zweiten springt man in die nächste Zelle.
- Der Rückgabewert des letzten Befehls einer Code Input-Zelle wird in einer folgenden Code Output-Zelle ausgegeben. Sie können die Ausgabe mit einem Strichpunkt am Ende des letzten Befehls unterdrücken.
- Kommentare in Code-Zellen beginnen mit dem Rautezeichen `#`.
- Funktionen haben runde Klammern. Hilfe z. B. zur Funktion `print` erhalten Sie durch `help(print)` oder `print?` oder SHIFT-TAB drücken, wenn der Cursor nach der ersten runden Klammer von `print()` steht.
- Eine Liste der von Ihnen definierten Variablen erhalten Sie mit dem magic command `%whos`. Löschen einzelner Variablen, hier z. B. der Variable `x`, erfolgt mit dem Befehl `del(x)`. Löschen aller selber definierten Variablen erfolgt mit dem Befehl `%reset -s`.
- *Tipp:* Verwenden Sie die Tabulator-Vervollständigung beim Coden!
- *Tipp:* Mit der Tastenkombination `Strg+I` öffnet sich der sehr hilfreiche **Contextual Help** Tab.
- *Tipp:* Öffnen Sie eine Console für Ihr Notebook, um außerhalb des Notebooks interaktiv Befehle im Namespace des Notebook-Kernels auszuführen.

Markdown Zellen: Markdown ist eine vereinfachte Auszeichnungssprache, die bereits in der Ausgangsform ohne weitere Konvertierung leicht lesbar ist. Sie können in Markdown sehr leicht folgenden strukturierten Text erstellen:

- Überschirften: Rautesymbol(e) vor der Überschrift
- Listen: mit Minus-, Plus- oder Sternzeichen
- Links: mit Syntax `[Name](URL)`
- Bilder: mit Syntax `![Name](Pfad-zu-Bilddatei)`
- Mathematische Formeln wie z. B. $K = \frac{mv^2}{2}$ via dem LaTeX-Code `$K = \frac{mv^2}{2}$`
- *Tipp:* Cheatsheet

Keyboard Shortcuts

Die Keyboard Shortcuts (Tastenkürzel) sind in den Menüs neben den Auswahlen angegeben. Hier eine persönliche Auswahl:

Shortcut	Effekt
Ctrl+s	save notebook
Ctrl+Shift+q	close and shutdown notebook
Ctrl+f	Find
Enter	enter edit mode
Escape	enter command mode
Ctrl+Enter	run cell
Shift+Enter	run cell and got to cell below
Shift+m	merge selected cells
in edit mode: Ctrl+Shift+-	split cell
in command mode: m	set cell type to markdown
in command mode: y	set cell type to code
in command mode: d d	delete cell
in command mode: z	undo deleting of cell
in command mode: a	insert cell above
in command mode: b	insert cell below
in command mode: x/c/v	cut/copy/paste cells
in command mode: UP/DOWN arrow	selected previous/next cell
in command mode: i+i	interrupt kernel
in command mode: 0+0	restart kernel
in command mode: Shift+l	toggle all line numbers

Bevor Sie **JupyterLab beenden**, vergessen Sie nicht, Ihre Jupyter-Notebooks zu speichern, zu schließen sowie die zugehörigen Kernel herunterzufahren. Zum Beenden von JupyterLab wählen Sie im Menü `File/Shut Down`.

Navigationsbefehle im Dateisystem: Für die Arbeit in einem System-Terminal, dem `Anaconda Command Prompt` oder in einer Codezelle sind oft folgende Navigationsbefehle nützlich:

- `pwd`: print working/current directory
- `ls` oder `dir`: list files in current directory
- `cd DIR`: change into absolute or relative directory DIR
- `cd ..`: change to parent directory

Tipps: Schauen Sie mal in die Gallery of interesting Jupyter Notebooks und in das Jupyter-Tutorial rein.

10.5. Tutorial

Dieses Tutorial zeigt einführend, wie Python zum wissenschaftlichen Rechnen verwendet werden kann. Ausführlichere Einführungen finden Sie z. B. in:

- Python Language Companion to the excellent and free book Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares by Stephen Boyd and Lieven Vandenberghe, Cambridge University Press, 2018.
- Programming for Computations - Python. A Gentle Introduction to Numerical Simulations with Python 3.6. von Svein Linge und Hans Petter Langtangen, 2. Auflage, 2020. => PDF-Download im FHV-Netz möglich!

Wer Python von Grund auf und umfassender lernen möchte, kann z. B. diese Bücher und Links verwenden:

- Python - Der Grundkurs von Michael Kofler, 2. Auflage, 2021
- Official Python 3 documentation
- Python Tutorial bei www.w3schools.com

10.5.1. Python als Taschenrechner

In der Python-Shell können Sie intuitiv Rechnungen eintippen. Hier ein paar Beispiele:

```
(42 + 137)/0.815
```

```
219.6319018404908
```

Achtung: Das Dezimaltrennzeichen ist im Englischen ein Punkt, und Potenzieren wird mit `**` implementiert!

```
1.23**45
```

```
11110.40818513195
```

Um die elementaren mathematischen Funktionen zu verwenden, müssen Sie zuerst, und nur einmal, das NumPy-Paket importieren. Wir verwenden das Kürzel `np`, um die Funktionen von NumPy anzusprechen.

```
import numpy as np

np.sqrt(np.pi/2)
```

```
1.2533141373155001
```

Kommentare beginnen mit einem `#`.

```
# This is a comment.
np.log(1)  # This also.
```

```
0.0
```

10.5.2. Formatierte Ausgabe

Für eine formatierte Ausgabe verwenden wir die Funktion `print` und sogenannte f-Strings:

```
name  = "Klaus"
like_math = True
age = 50
value = 12.3456789

print(f"Hallo! My name is {name}.")
print(f"I {like_math = }")
print(f"I am {age} years old. These are about {age*365} days.")
print(f"A value: {value:.2f}")  # float with two decimal places
```

```
Hallo! My name is Klaus.
I like_math = True
I am 50 years old. These are about 18250 days.
A value: 12.35
```

10.5.3. Listen und Vektoren

In der Mathematik ist ein Vektor des $\mathbb{R}^n$ eine geordnete Liste von n reellen Zahlen. In Python werden allgemeine Listen (nicht nur von Zahlen) mit eckigen Klammern erzeugt:

```
a = [-3, 5.7, 8]
b = [2.1, 0, -8]
c = ["Karl", "Klaus", "Erna", "Maria"]
```

Allerdings ist z. B. die Addition von Python-Listen mit Zahleneinträgen **nicht** wie die Addition von Vektoren in der Mathematik, nämlich elementweise, definiert:

```
a + b
```

```
[-3, 5.7, 8, 2.1, 0, -8]
```

Um den gewünschten Effekt von Addition und Skalarmultiplikation für Vektoren zu erreichen, machen wir solche Python-Listen zu NumPy-Arrays:

```
v = np.array([-3, 5.7, 8])
w = np.array([2.1, 0, -8])

print(v + w)
print(3*w)
```

```
[-0.9  5.7  0. ]
[  6.3   0.  -24. ]
```

Es gibt einige hilfreiche NumPy-Funktionen, um oft gebrauchte Typen von NumPy-Arrays zu erzeugen, z. B.:

```
# 11 evenly spaced numbers over the interval [0, 50]:
x = np.linspace(0, 50, 11)
print(x)
```

```
[ 0.  5. 10. 15. 20. 25. 30. 35. 40. 45. 50.]
```

```
# evenly spaced numbers over the interval [0, 20] with step size 2:
x = np.arange(0, 20, 2)  # Note, that 20 is not included!
print(x)
```

```
[ 0  2  4  6  8 10 12 14 16 18]
```

10.5.4. NumPy-Arrays

Ein Vektor wird als eindimensionales NumPy-Array implementiert, eine Matrix als zweidimensionales NumPy-Array.

```
v = np.array([-3, 5.7, 8])

print(f"{v}")
print(np.ndim(v)) # number of dimensions
print(len(v))     # length, i. e. number of items, not the geometric length!
print(v.size)     # also number of items
```

```
[-3.   5.7  8. ]
1
3
3
```

```
# A matrix is filled row-wise with lists:
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
print(f"{M}")
print(np.ndim(M)) # number of dimensions of the array
print(M.shape)    # shape of the array
print(M.size)     # number of elements in the array
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
2
(2, 3)
6
```

Indexing und slicing of vectors and lists:

```
x = np.array([1, 2.1, -5])

# accessing vector (=1-dim-array) items works the same as with lists:
print(x[0])     # Python starts indexing with 0.
print(x[1:3])   # items from index 1 included until index 3 excluded
print(x[:2])    # items up to index 2 with index 2 excluded
print(x[1:])    # items starting from index 1 with index 1 included
print(x[-1])    # last item
print(x[-2])    # second last item
print(x[-2:])   # from the second last item to the end
print(x[-2:-1]) # should be clear now :-)
```

```
1.0
[ 2.1 -5. ]
[1.  2.1]
[ 2.1 -5. ]
-5.0
2.1
[ 2.1 -5. ]
[2.1]
```

Indexing and slicing of matrices:

```
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
print(f"{M}")

# accessing matrix items works similar:
print("first row, second column:")
print(M[0, 1])
print("first row:")
print(M[0, :])
print("second column:")
print(M[:, 1])
print("second and third column:")
print(M[:, 1:3])
print("first two columns:")
print(M[:, :2])
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
first row, second column:
5.7
first row:
[-3.   5.7  8. ]
second column:
[5.7 0. ]
second and third column:
[[ 5.7  8. ]
 [ 0.  -8. ]]
```

```
first two columns:
[[-3.   5.7]
 [ 2.1  0. ]]
```

Beachten Sie, dass beim Slicing das Rückgabe-Array die angepasste Dimension hat, die sich von der Dimension des Ausgangs-Arrays unterscheiden kann. Hier ein Beispiel, wie man das Slicing so anpasst, dass das Rückgabe-Array die gleiche Dimension wie das Ausgangs-Array hat:

```
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
print(f"{M}")
print("first column as 1-dim array:")
print(M[:, 0])
print("first column as 2-dim array:")
print(M[:, [0]])
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
first column as 1-dim array:
[-3.   2.1]
first column as 2-dim array:
[[-3. ]
 [ 2.1]]
```

Transponieren einer Matrix:

```
print(f"{M}")
print(f"{M.T}")
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
[[-3.   2.1]
 [ 5.7  0. ]
 [ 8.  -8. ]]
```

10.5.5. Vektor- und Matrizenrechnung

```
v = np.array([-1, 5, 0])
w = np.array([ 2, 2, 1])

print("inner product:")
print(f"{v@w = } or {np.dot(v, w) = }")
print("cross product:")
print(f"{np.cross(v, w) = }")

print("Caution: These are element-wise operations:")
print(f"{v*w = }")
print(f"{v/w = }")
print(f"{v**2 = }")
```

```
inner product:
v@w = 8 or np.dot(v, w) = 8
cross product:
np.cross(v, w) = array([  5,   1, -12])
Caution: These are element-wise operations:
v*w = array([-2, 10,  0])
v/w = array([-0.5,  2.5,  0. ])
v**2 = array([ 1, 25,  0])
```

Die Matrixmultiplikation wird wie das innere Produkt mit dem `@` Operator durchgeführt:

```
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
N = np.array([[1, 2],
              [3, 4],
              [5, 6]])
print("M@N = ")
print(f"{M@N}")

print("M@v = ")
print(f"{M@v}")
```

```
M@N =
[[ 54.1  64.8]
 [-37.9 -43.8]]
M@v =
[31.5 -2.1]
```

Berechnung von Rängen, Determinanten, Eigenwerten, Eigenvektoren:

```
M = np.array([[-3, 2, 8],
              [ 2, 0, 1],
              [ 8, 1, 3]])
rank = np.linalg.matrix_rank(M)
print(f"{rank = }")
det = np.linalg.det(M)
print(f"{det = :.2f}")
eigenvalues, eigenvectors = np.linalg.eig(M)
print("1-dim array of eigenvalues:")
print(f"{eigenvalues}")
print("2-dim array of eigenvectors in columns:")
print(f"{eigenvectors}")

print("Some check by computing M*v_1 - lambda_1*v_1")
print("which should be a numerically zero vector:")
result = M@eigenvectors[:, 0] - eigenvalues[0]*eigenvectors[:, 0]
print(f"{result = }")
```

```
rank = 3
det = 23.00
```

```
1-dim array of eigenvalues:
[ 8.97566072 -8.68045912 -0.2952016 ]
2-dim array of eigenvectors in columns:
[[ 0.56706349  0.82346126  0.01872296]
 [ 0.21494363 -0.12599762 -0.96846468]
 [ 0.7951341  -0.55320535  0.2484464 ]]
Some check by computing M*v_1 - lambda_1*v_1
which should be a numerically zero vector:
result = array([ 0.00000000e+00, -1.11022302e-15, -1.77635684e-15])
```

10.5.6. Lineare Gleichungssysteme

Wir lösen das quadratische lineare Gleichungssystem $Ax = b$ und überprüfen zuerst, ob es eine eindeutige Lösung gibt:

```python
A = np.array([[1, 2],
              [3, 4]])
b = np.array([5, 6])

(m, n) = A.shape
print(f"rank(A) = {np.linalg.matrix_rank(A)}")
if np.linalg.matrix_rank(A) == n:
    x = np.linalg.solve(A, b)  # Matrix A must be square and of full-rank!
    print(f"unique solution: {x = }")
else:
    print("no unique solution!")
```

```
rank(A) = 2
unique solution: x = array([-4. ,  4.5])
```

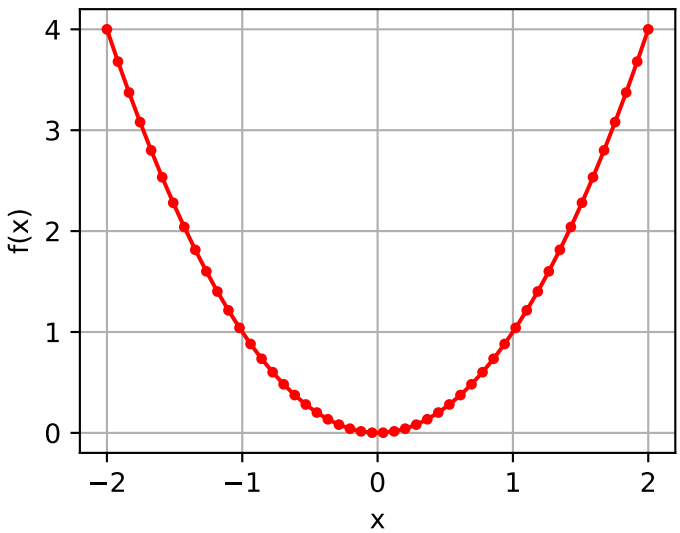
Kern (engl. null space, kernel) und Bild (engl. column space, range, image) einer Matrix:

```python
import scipy as sp

A = np.array([[1, 2, -1],
              [2, 4,  3]])

print("matrix A:")
print(f"{A}")
print(f"{np.linalg.matrix_rank(A) = }.")

print("orthonormal basis for the null space (kernel) of A:")
N = sp.linalg.null_space(A)
print(f"{N}")

print("orthonormal basis for the column space (range, image) of A:")
C = sp.linalg.orth(A)
print(f"{C}")

# the orthonormal bases are given as column vectors.
```

```
matrix A:
[[ 1  2 -1]
 [ 2  4  3]]
np.linalg.matrix_rank(A) = 2.
orthonormal basis for the null space (kernel) of A:
[[ 8.94427191e-01]
 [-4.47213595e-01]
 [-3.33066907e-16]]
orthonormal basis for the column space (range, image) of A:
[[-0.27000134 -0.96285995]
 [-0.96285995  0.27000134]]
```

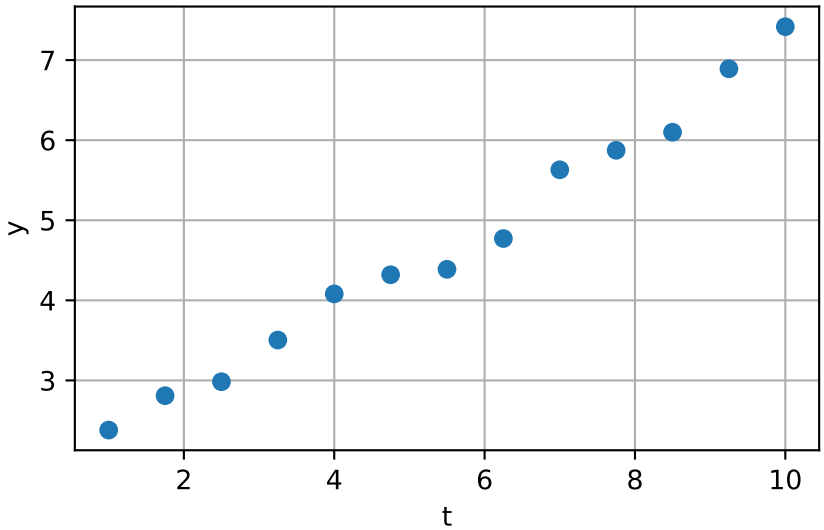
10.5.7. Plot einer Funktion

```python
# define the function f:
def f(x):
    return x**2

# 50 grid points distributed between -2 and 2 with equal spacing:
x = np.linspace(-2, 2, num=50)

# plot the function at the grid points:
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 3))
plt.plot(x, f(x), linestyle='-', color='red', marker='.')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.show()
```



10.5.8. Regression

```
n = 13                                  # number of data points
t = np.linspace(1, 10, num = n)         # time points measurements
noise = 0.2*np.random.normal(size = n)  # noise in measurement values
y = 2 + 0.5*t + noise                   # measurement values: line + noise

plt.figure(figsize=(5, 3))
plt.plot(t, y, 'o')
plt.xlabel('t')
plt.ylabel('y')
plt.grid(True)
```




```
col_of_ones = np.ones(n)
A = np.stack((col_of_ones, t), axis = 1)
b = y.reshape(13, 1)

print(f"{A = }")
print(f"{b = }")
```

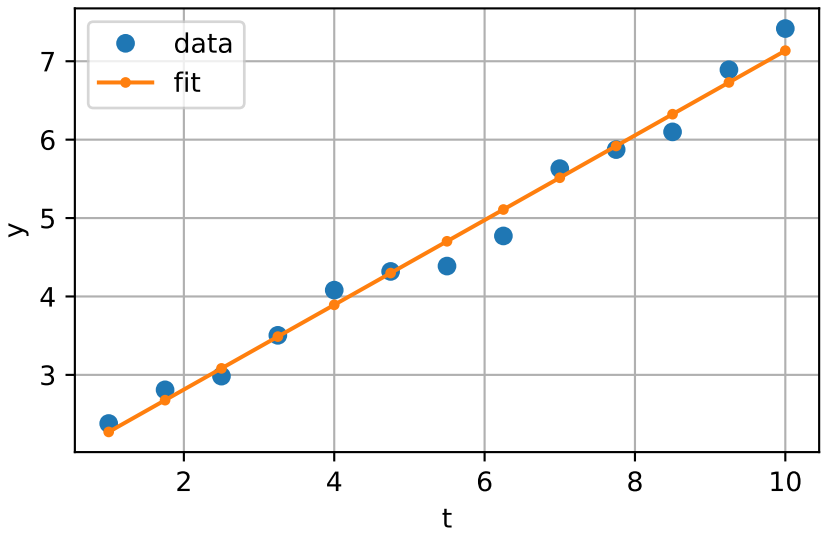
```
A = array([[ 1.  ,  1.  ],
       [ 1.  ,  1.75],
       [ 1.  ,  2.5 ],
       [ 1.  ,  3.25],
       [ 1.  ,  4.  ],
       [ 1.  ,  4.75],
       [ 1.  ,  5.5 ],
       [ 1.  ,  6.25],
       [ 1.  ,  7.  ],
       [ 1.  ,  7.75],
       [ 1.  ,  8.5 ],
```

```
       [ 1.  ,  9.25],
       [ 1.  , 10.  ]])
b = array([[2.37987992],
       [2.80933317],
       [2.98398561],
       [3.50432988],
       [4.08060784],
       [4.31934898],
       [4.38762422],
       [4.77186541],
       [5.63045993],
       [5.87336898],
       [6.09906106],
       [6.89153418],
       [7.41693431]])
```

```python
# via Formel:
x_hat_1 = np.linalg.inv(A.T @ A) @ A.T @ b
print(x_hat_1)

# via Normalgleichungen:
x_hat_2 = np.linalg.solve(A.T @ A, A.T @ b)
print(x_hat_2)

# via lstsq:
x_hat_3 = np.linalg.lstsq(A, b, rcond=None)[0]
print(x_hat_3)
```

```
[[1.73198652]
 [0.54031481]]
[[1.73198652]
 [0.54031481]]
[[1.73198652]
 [0.54031481]]
```

```python
# Fit A*x_hat:
y_hat = A@x_hat_1

plt.figure(figsize=(5, 3))
plt.plot(t, y, 'o', label='data')
plt.plot(t, y_hat, '.-', label='fit')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid(True)
```



10.5.9. For-Schleife

Zinseszinsrechnung:

```
B_0 = 1000.0  # initial balance in EUR
q = 1.05      # interest rate factor
n = 5         # number of years

# range: start 1 is included, stop n + 1 is excluded!
for i in np.arange(1, n + 1, 1):
    B_i = B_0*q**i
    print(f"Balance after {i} years = {B_i:.2f} EUR.")
```

```
Balance after 1 years = 1050.00 EUR.
Balance after 2 years = 1102.50 EUR.
Balance after 3 years = 1157.63 EUR.
Balance after 4 years = 1215.51 EUR.
Balance after 5 years = 1276.28 EUR.
```

10.5.10. While-Schleife

Wir berechnen die Fakultät einer natürlichen Zahl $n > 0$:

```
n = 10
f = 1  # initial value for factorial n!
k = n  # initial decreasing number k
while k > 0:
    print(f"{k = }")
    f = f*k
    k = k - 1
```

```
print(f"{n}! = {f}")
```

```
k = 10
k = 9
k = 8
k = 7
k = 6
k = 5
k = 4
k = 3
k = 2
k = 1
10! = 3628800
```

10.5.11. If-Elif-Else Abfrage

Wir bestimmen das Vorzeichen von ein paar Zahlen:

```
numbers = np.arange(-3, 4, 1)
for n in numbers:
    if n > 0:
        print(f"{n} hat das Vorzeichen +.")
    elif n == 0:
        print(f"{n} hat kein Vorzeichen.")
    else:
        print(f"{n} hat das Vorzeichen -.")
```

```
-3 hat das Vorzeichen -.
-2 hat das Vorzeichen -.
-1 hat das Vorzeichen -.
0 hat kein Vorzeichen.
1 hat das Vorzeichen +.
2 hat das Vorzeichen +.
3 hat das Vorzeichen +.
```

Beachte: Eine If-Elif-Else Abfrage muss nicht zwingend `elif` oder `else` Teile haben.

11. PuLP

11.1. Überblick

Wir verwenden in der Lehrveranstaltung das Python-Paket PuLP. PuLP ist ein Open-Source-Projekt und frei verfügbar. PuLP ist eine algebraische Modellierungssprache zur benutzerfreundlichen Formulierung von mathematischen Optimierungsproblemen. Zum Lösen eines mit PuLP formulierten Optimierungsproblems wird ein Solver benötigt. PuLP beinhaltet den Open Source Solver CbC und unterstützt eine Vielzahl von weiteren Open-Source sowie kommerziellen Solvern, die aber zusätzlich installiert werden müssen.

11.2. Alternativen

In diesem Abschnitt geben wir einen Überblick über andere High-Level Software zur Formulierung von Optimierungsproblemen, weil:

- Die Links führen Sie zu vielen, oft sehr guten Tutorials inkl. Code-Beispielen.
- In Realprojekten ist es wichtig, das passende Software-Setting unter Berücksichtigung folgender Punkte auszuwählen: kommerziell versus open source, Schnittstellen zu Programmiersprachen, Benutzerfreundlichkeit, Performance etc.. Dafür sollte man seinen Überblick über die LP-Software aktuell halten.

Liste von bekannten Algebraischen Modellierungssprachen:

- AMPL: you can find examples e. g. in the AMPL book
- GAMS: for some simple examples see e. g.the GAMS Quick Start Tutorial
- AIMMS: in AIMMS_modeling.pdf you find many examples and tricks
- Mosel: for examples see e. g. the FICO Xpress Optimization Examples Repository
- GNU MathProg where you can find a short example, see also GLPK and this online solver. The GNU MathProg language is a subset of the AMPL language.
- OPL: for examples see e. g. the OPL Language User's Manual
- MPL: you can find examples e. g. here

Für Python gibt es unter anderem folgende Modellierungs- und Entwicklungsumgebungen für (MI)LPs:

- SciPy LP and MILP with `linprog` and `milp`: open source solver and interface, modelling in matrix/vector form
- CVXOPT: open source interface to own solver and other solvers, can also solve quadratic programs, user's guide, see also cvxpy
- gurobipy: interface to commercial Gurobi solver
- Cplex: commercial solver and interface, academic license available
- lp_solve: open source solver and interface
- GLPK interfaces: open source solver and interfaces
- Pyomo: open source interface to different solvers

- PuLP: open source interface to own solver and other solvers documentation, github
- linopy: open source interface to different solvers
- Python-MIP: open source interface to different solvers
- PySCIPOpt: interface to the SCIP Optimization Suite, scipbook
- Optlang: open source interface to different solvers, github
- PYMPROG: open source interface to different solvers

11.3. Solver

Liste der bekanntesten Solver:

- CBC: open source solver
- HiGHs: open source solver
- GLPK: open source solver
- SCIP Optimization Suite: non-commercial solver for mixed integer programming (MIP) and mixed integer nonlinear programming (MINLP)
- lp_solve: open source solver
- Gurobi: commercial solver, academic license available
- Cplex: commercial solver and interface, academic license available
- Xpress: commercial solver
- Ipopt: open source software package fo large-scale nonlinear optimization, documentation

11.4. Installation

- In Anaconda und Miniconda: `conda install -c conda-forge pulp`
- Allgemein mit pip: `pip install pulp`

11.5. Beispiel

Der folgende Code implementiert das blending problem mit PuLP und löst es mit dem CBC-Solver.

11.5.1. Problemstellung

Whiskas wants to make their cat food out of just two ingredients: chicken and beef. The company wants to meet certain nutritional requirements, and do so at a minimum cost. The table below shows the nutritional content of each ingredient per gram .

Nutrient	Chicken	Beef
Protein	0.100	0.200
Fat	0.080	0.100
Fibre	0.001	0.005
Salt	0.002	0.005

The price of chicken is 0.013 USD per gram, and the price of beef is 0.008 USD per gram. The company has the following nutritional requirements for the cat food:

- at least 8.0 % protein
- at least 6.0 % fat
- at most 2.0 % fibre
- at most 0.4 % salt

11.5.2. LP-Formulierung

Decision variables:

- x_1: percentage of chicken in the cat food
- x_2: percentage of beef in the cat food

$$
\begin{aligned}
\min\ 0.013x_1 + 0.008x_2 \\
\text{s. t. } x_1 + x_2 &= 100 \\
0.100x_1 + 0.200x_2 &\geq 8.0 \\
0.080x_1 + 0.100x_2 &\geq 6.0 \\
0.001x_1 + 0.005x_2 &\leq 2.0 \\
0.002x_1 + 0.005x_2 &\leq 0.4 \\
x_1, x_2 &\geq 0
\end{aligned}
$$

11.5.3. Ausführliche Implementierung

Zuerst importieren wir das Paket `pulp`, und dann modellieren wir das Problem in einer ausführlichen Form:

```
import pulp

# Create the 'prob' object to contain the problem data:
prob = pulp.LpProblem("Whiskas_problem", pulp.LpMinimize)
# Note: no space in the name!

# Create variables for beef and chicken with a lower limit of zero:
# help(pulp.LpVariable)  # check the documentation!
x1 = pulp.LpVariable("chicken_percent",
    lowBound=0, upBound=None, cat='Continuous')
x2 = pulp.LpVariable("beef_percent",
    lowBound=0, upBound=None, cat='Continuous')

# The objective function is added to 'prob' first!
obj = 0.013*x1 + 0.008*x2
prob += (obj, "total cost of ingredients per 100 gram")

# The five constraints are added to 'prob' as tupes with two elements:
# the constraint and a string with a name for the constraint.
c1 = x1 + x2 == 100
prob += (c1, "percentages sum")
c2 = 0.100*x1 + 0.200*x2 >= 8.0
prob += (c2, "protein requirement")
```

```
c3 = 0.080*x1 + 0.100*x2 >= 6.0
prob += (c3, "fat requirement")
c4 = 0.001*x1 + 0.005*x2 <= 2.0
prob += (c4, "fibre requirement")
c5 = 0.002*x1 + 0.005*x2 <= 0.4
prob += (c5, "salt requirement")
print(prob)
```

```
Whiskas_problem:
MINIMIZE
0.008*beef_percent + 0.013*chicken_percent + 0.0
SUBJECT TO
percentages_sum: beef_percent + chicken_percent = 100

protein_requirement: 0.2 beef_percent + 0.1 chicken_percent >= 8

fat_requirement: 0.1 beef_percent + 0.08 chicken_percent >= 6

fibre_requirement: 0.005 beef_percent + 0.001 chicken_percent <= 2

salt_requirement: 0.005 beef_percent + 0.002 chicken_percent <= 0.4

VARIABLES
beef_percent Continuous
chicken_percent Continuous
```

Nun lösen wir das Problem mit einem Solver und geben die Ergebnisse aus:

```
# Solve using PuLP's COIN-CBC-Solver or some other installed solver:
# If you set the msg argument to True, then the solver will
# print output as it solves the problem.
prob.solve(pulp.COIN(msg=True))  # CBC is the default solver included in PuLP
# prob.solve(pulp.COIN(msg=False))
# prob.solve(pulp.HiGHS_CMD(msg=False)) # HiGHs is another open source solver
# prob.solve(pulp.GLPK(msg=False))      # GLPK is another open source solver
# prob.solve(pulp.GUROBI(msg=False))    # Gurobi is a commercial solver

# status of the solution:
print(f"status: {pulp.LpStatus[prob.status]}")

# optimal values of the variables:
print("optimal decisions:")
for v in prob.variables():
    print(f"  {v.name} = {v.varValue:.2f} %")

# optimal value:
print(f"optimal cost = {pulp.value(prob.objective):.2f} USD per 100 g")
```

```
Welcome to the CBC MILP Solver
Version: 2.10.11
```

```
Build Date: Oct 26 2023

command line - cbc /tmp/42ddbf9d2a49441491b746eb3236d9d4-pulp.mps -timeMode elapsed -branch -printi
At line 2 NAME          MODEL
At line 3 ROWS
At line 10 COLUMNS
At line 23 RHS
At line 29 BOUNDS
At line 30 ENDATA
Problem MODEL has 5 rows, 2 columns and 10 elements
Coin0008I MODEL read with 0 errors
Option for timeMode changed from cpu to elapsed
Presolve 0 (-5) rows, 0 (-2) columns and 0 (-10) elements
Empty problem - 0 rows, 0 columns and 0 elements
Optimal - objective value 0.96666667
After Postsolve, objective 0.96666667, infeasibilities - dual 0 (0), primal 0 (0)
Optimal objective 0.9666666667 - 0 iterations time 0.002, Presolve 0.00
Option for printingOptions changed from normal to all
Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00

status: Optimal
optimal decisions:
  beef_percent = 66.67 %
  chicken_percent = 33.33 %
optimal cost = 0.97 USD per 100 g
```

11.5.4. Kompakte Implementierung

Nun das selbe in einer kompakteren Form:

```python
prob = pulp.LpProblem("Whiskas_problem", pulp.LpMinimize)
x1 = pulp.LpVariable("x1", 0, None, "Continuous")
x2 = pulp.LpVariable("x2", 0, None, "Continuous")
prob += 0.013*x1 + 0.008*x2
prob += x1 + x2 == 100
prob += 0.100*x1 + 0.200*x2 >= 8.0
prob += 0.080*x1 + 0.100*x2 >= 6.0
prob += 0.001*x1 + 0.005*x2 <= 2.0
prob += 0.002*x1 + 0.005*x2 <= 0.4
print(prob)

prob.solve(pulp.COIN(msg=False))
print(f"status: {pulp.LpStatus[prob.status]}")
print(f"optimal cost = {pulp.value(prob.objective):.2f} USD per 100 g")
print(f"{x1.value() = :.2f} %")
print(f"{x2.value() = :.2f} %")
```

```
Whiskas_problem:
MINIMIZE
0.013*x1 + 0.008*x2 + 0.0
```

```
SUBJECT TO
_C1: x1 + x2 = 100

_C2: 0.1 x1 + 0.2 x2 >= 8

_C3: 0.08 x1 + 0.1 x2 >= 6

_C4: 0.001 x1 + 0.005 x2 <= 2

_C5: 0.002 x1 + 0.005 x2 <= 0.4

VARIABLES
x1 Continuous
x2 Continuous

status: Optimal
optimal cost = 0.97 USD per 100 g
x1.value() = 33.33 %
x2.value() = 66.67 %
```

Literaturverzeichnis

[BV18] Stephen Boyd und Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Englisch. 1. Aufl. Cambridge, UK ; New York, NY: Cambridge University Press, Juni 2018. ISBN: 978-1-316-51896-0. URL: https://web.stanford.edu/~boyd/vmls/.

[Dom+15a] Wolfgang Domschke u. a. *Einführung in Operations Research*. Berlin, Heidelberg: Springer, 2015. ISBN: 978-3-662-48215-5. DOI: 10.1007/978-3-662-48216-2. URL: https://link.springer.com/10.1007/978-3-662-48216-2 (besucht am 02. 02. 2024).

[Dom+15b] Wolfgang Domschke u. a. *Übungen und Fallbeispiele zum Operations Research*. Berlin, Heidelberg: Springer, 2015. ISBN: 978-3-662-48229-2. DOI: 10.1007/978-3-662-48230-8. URL: https://link.springer.com/10.1007/978-3-662-48230-8 (besucht am 02. 02. 2024).

[HL20] Frederick Hillier und Gerald Lieberman. *Introduction to Operations Research*. 11th edition. New York, NY: McGraw-Hill Education Ltd, 2020. 964 S. ISBN: 978-1-260-57587-3.

[Kle13] Philip N. Klein. *Coding the Matrix: Linear Algebra through Applications to Computer Science*. 1. Aufl. Newton, Mass: Newtonian Press, 3. Sep. 2013. 548 S. ISBN: 978-0-615-88099-0.

[LL20] Svein Linge und Hans Petter Langtangen. *Programming for Computations - Python: A Gentle Introduction to Numerical Simulations with Python 3.6*. Englisch. 2. Aufl. Springer, 2020. ISBN: 978-3-030-16876-6.

[LLM21] David Lay, Steven Lay und Judi McDonald. *Linear Algebra and Its Applications, Global Edition*. 6. Aufl. Pearson Education Limited, 4. Feb. 2021. 672 S. ISBN: 978-1-292-35121-6.

[MG10] Jirí Matousek und Bernd Gärtner. *Understanding and Using Linear Programming*. 2007. Aufl. Berlin ; New York: Springer Berlin Heidelberg, 2. Juni 2010. 236 S. ISBN: 978-3-540-30697-9.

[Pap15] Lothar Papula. *Mathematik für Ingenieure und Naturwissenschaftler Band 2: Ein Lehr- und Arbeitsbuch für das Grundstudium*. Deutsch. 14., überarb. u. erw. Aufl. Wiesbaden: Springer Vieweg, Apr. 2015. ISBN: 978-3-658-07789-1.

[Pap18] Lothar Papula. *Mathematik für Ingenieure und Naturwissenschaftler Band 1: Ein Lehr- und Arbeitsbuch für das Grundstudium*. Deutsch. 15., überarb. Aufl. Wiesbaden Heidelberg: Springer Vieweg, Aug. 2018. ISBN: 978-3-658-21745-7.

[Pap19] Lothar Papula. *Mathematik für Ingenieure und Naturwissenschaftler - Anwendungsbeispiele: 222 Aufgabenstellungen mit ausführlichen Lösungen*. Deutsch. 8., überarb. Aufl. Wiesbaden Heidelberg: Springer Vieweg, 2019. ISBN: 978-3-658-24881-9.

[Pap20] Lothar Papula. *Mathematik für Ingenieure und Naturwissenschaftler - Klausur- und Übungsaufgaben: 711 Aufgaben mit ausführlichen Lösungen zum Selbststudium und zur Prüfungsvorbereitung*. Deutsch. 6., erw. u. überarb. Aufl. 2020 Edition. Wiesbaden: Springer Vieweg, 2020. ISBN: 978-3-658-30270-2.

[SC17] Ramteen Sioshansi und Antonio J. Conejo. *Optimization in Engineering: Models and Algorithms*. 1st ed. 2017 Edition. New York, NY: Springer, 3. Juli 2017. 427 S. ISBN: 978-3-319-56767-9.

[Str23] Gilbert Strang. *Introduction to Linear Algebra*. 6th edition. Wellesley, Mass: Cambridge University Pr., 15. Jan. 2023. 430 S. ISBN: 978-1-73314-667-8.

[Wil13] H. Paul Williams. *Model Building in Mathematical Programming 5e*. 5. Hoboken, N.J: Wiley, 22. Feb. 2013. 432 S. ISBN: 978-1-118-44333-0.