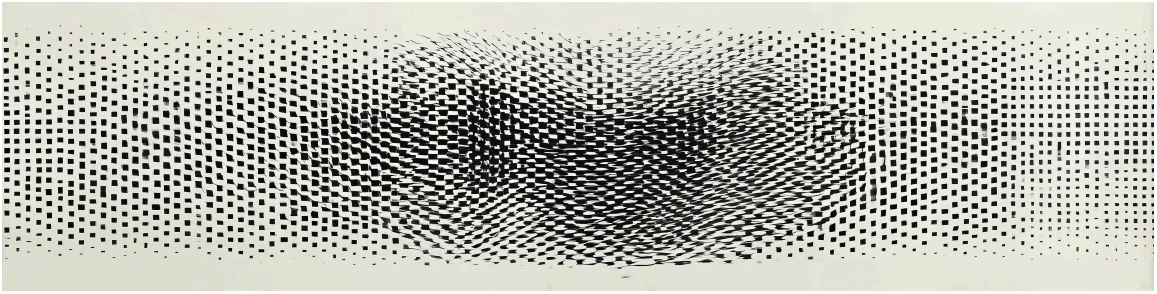
Lineare Algebra

Joern Ploennigs · AI4SC



> The Matrix is everywhere. It is all around us. Even now in this very room
>
> — Morpheus (The Matrix)

Erstellung von Arrays für Vektor, Matrix- und Tensorberechnungen

NumPy ist die führende Bibliothek in Python für Vektor- und Matrixberechnungen. Daten und Berechnungen werden dabei nicht in Python, sondern mit schnellen internen Funktionen und Operationen ausgeführt. Dadurch ist es besonders platzsparend und schnell in der Verarbeitung großer Datensätze.

Matrixberechnungen werden z.B. in der Finite-Elemente-Methode (FEM) genutzt, um Spannungen und Verschiebungen in Tragwerken zu berechnen.

Die Bibliothek NumPy wird normalerweise in Python mit der Abkürzung `np` importiert. Dadurch kann man einfach das kurze `np` im Code schreiben, wenn man die Bibliothek referenziert. Das auch Standard in den meisten Dokumentationen online.

```
# Import von NumPy
import numpy as np
```

Die wichtigste Datenstruktur in **NumPy** sind n-dimensionale Arrays, die sowohl zur Darstellung von Vektoren ($n = 1$) und Matrizen ($n = 2$) verwendet werden. Die Arrays können nicht nur Vektoren sein (Temperaturverlauf entlang eines Stabträgers) sondern auch höhere Dimensionen haben, wie Matrizen (Verformungen eines 2D-Tragwerks) oder Tensoren (Spannungsverteilungen in einem 3D-Bauteil). Diese Tensoren sind insbesondere für komplexe ML-Modelle wie Tiefe Neuronale Netzwerke wichtig.

Arrays in NumPy werden mit `np.array([1, 2, 3, 4, 5])` erstellt. Wir übergeben dabei eine Liste `[1, 2, 3, 4, 5]` der Zahlen.

```
# Beispiel: Messwerte zur Luftfeuchtigkeit in einem Betonbauteil
arr_1d = np.array([88.2, 86.5, 85.1, 83.9, 82.4])
print("Messreihe:")
print(arr_1d)
```

```
Messreihe:
[88.2 86.5 85.1 83.9 82.4]
```

Dabei wird die Liste in ein NumPy `ndarray` umgewandelt. Dies sehen wir, wenn wir den Datentyp prüfen.

```
print("Datentyp des Vektors in Python: ", type(arr_1d))
```

```
Datentyp des Vektors in Python:  <class 'numpy.ndarray'>
```

Intern nutzt NumPy keine Python-Datentypen (`int` in dem Beispiel), sondern C-Datentypen, die effizienter in der Speichernutzung sind und vor allem deutlich schnellere numerische Berechnungen erlauben. Für ML-Anwendungen sind `ndarray`-Objekte meist besser geeignet als Python-Listen, wenn Merkmale oder Matrizen dargestellt werden sollen. Der wichtigste Grund ist, dass `ndarray` homogene numerische Daten kompakt speichert und vektorisierte Operationen wie Addition, Skalierung oder Matrixmultiplikation direkt unterstützt. Python-Listen sind flexibler, aber für numerische Daten langsamer und für lineare Algebra deutlich unhandlicher. Den Datentyp kann man mit dem `dtype` Attribut einsehen.

```
print("Datentyp der Werte im Vektor in NumPy: ", arr_1d.dtype)
```

```
Datentyp der Werte im Vektor in NumPy:  float64
```

Den Datentypen kann man beim Erstellen festlegen. Das ist insbesondere sinnvoll, wenn NumPy den falschen Datentyp erkennt.

```
# Erstellen eines einfachen Vektors als 1D-Arrays mit Reellen Zahl
arr_1d_f = np.array([6, 7, 8, 9, 10], dtype=np.float32)
print("3x1 Vektor:")
print(arr_1d_f)
print("Datentyp der Werte im Vektor in NumPy: ", arr_1d_f.dtype)
```

```
3x1 Vektor:
[ 6.  7.  8.  9. 10.]
Datentyp der Werte im Vektor in NumPy:  float32
```

NumPy unterstützt hierbei viele unterschiedliche Datentypen:

Klasse	Datentyp	Python	NumPy	Beispiel
Numerisch	Ganze Zahl	`int`	`np.int8`, `np.int16`, `np.int32`, `np.int64`	`np.array([1, 2, 3], dtype=np.int64)`
	Natürliche Zahl	`int`	`np.uint8`, `np.uint16`, `np.uint32`, `np.uint64`	`np.array([1, 2, 3], dtype=np.int64)`
	Reelle Zahl	`float`	`np.float16`, `np.float32`, `np.float64`	`np.array([1.0, 2.5, 3.7], dtype=np.float64)`
	Komplexe Zahl	`complex`*	`np.complex64`, `np.complex128`, `np.complex192`, `np.complex256`	`np.array([1 | 2j, 3 | 4j], dtype=np.complex128)`
Logisch	Boolean	`bool`	`np.bool_`	`np.array([True, False, True], dtype=np.bool_)`
Textuell	Textuell	`str`	`np.str_`	`np.array(['hello', 'world'], dtype=np.str_)`
Temporal	Datum und Zeit	`datetime.date`*, `datetime.datetime`*	`np.datetime64`	`np.array(['2022-01-01', '2022-01-02'], dtype=np.datetime64)`
	Zeitdifferenz	`datetime.timedelta`*	`np.timedelta64`	`np.datetime64('2011-06-15T00:00') + np.timedelta64(12, 'h')`
Komplex	Python Objekt	`list`, `tuple`, `dict`, `set`, `object`	`np.object_`	`np.array([1, 'two', 3.0], dtype=np.object_)`

* in separaten Packages verfügbar

Die Gestalt des Arrays können wir mit dem `shape` Attribut abfragen:

```
print("Gestalt des Arrays:", arr_1d.shape)
```

```
Gestalt des Arrays: (5,)
```

Wenn wir eine Matrix (zweidimensionales Array) erstellen wollen, können wir an `np.array` eine Liste von Listen übergeben, mit je einer Unterliste für jede Zeile der Matrix. Wichtig ist, dass jede Unterliste gleich groß ist:

```
# Erstellen einer Matrix als 2D-Arrays
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("3x2 Matrix:")
print(arr_2d)
```

```
3x2 Matrix:
[[1 2 3]
 [4 5 6]]
```

Die Gestalt der Matrix hat sich entsprechend geändert. In unserem Beispiel ist die Shape gleich `(2, 3)`. Das bedeutet, das wir zwei Zeilen und drei Spalten haben.

```
print("Shape des Arrays:", arr_2d.shape)
```

```
Shape des Arrays: (2, 3)
```

Mit `reshape` kann man ein Array in eine andere Form bringen, solange die Anzahl der Elemente gleich bleibt. Das ist für ML sehr wichtig, weil Merkmalsdaten oft als Matrix der Form `(Anzahl Beobachtungen, Anzahl Merkmale)` vorliegen.

```
arr_flat = np.array([1, 2, 3, 4, 5, 6])
arr_matrix = arr_flat.reshape((2, 3))
print("Ursprüngliche Shape:", arr_flat.shape)
print("Neue Shape:", arr_matrix.shape)
print(arr_matrix)
```

```
Ursprüngliche Shape: (6,)
Neue Shape: (2, 3)
[[1 2 3]
 [4 5 6]]
```

Gerade der Unterschied zwischen einem Vektor mit Shape `(n,)` und einer Matrix mit Shape `(n, 1)` oder `(1, n)` ist wichtig, weil sich daraus bei linearen Algebra-Operationen unterschiedliche Ergebnisse ergeben können.

Zur Erzeugung von Standart-Vektoren oder -Matrizen gibt es spezielle Funktionen in NumPy. Um eine Einsmatrix zu erzeugen, nutzt man:

```
# Erstellen eines 3x3 Einsmatrix
ones_arr = np.ones((3,3))
print("3x3 Einsmatrix:")
print(ones_arr)
```

```
3x3 Einsmatrix:
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Nicht zu verwechseln mit der Einheitsmatrix, die zu erstellen ist, mit:

```
# Erstellen eines 3x3 Einheitsmatrix
identiy_arr = np.identity(3)
print("3x3 Einheitsmatrix:")
print(identiy_arr)
```

```
3x3 Einheitsmatrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Genauso lassen sich 0-Matrizen erzeugen, die man oft zur Initialisierung benötigt.

```
# Erstellen eines 3x3 0-Matrix
zeros_arr = np.zeros((3, 3))
print("3x3 Nullmatrix:")
print(zeros_arr)
```

```
3x3 Nullmatrix:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Zufallszahlen lassen sich mit der Unterbibliothek `random` erzeugen. Um eine Zufallsvektor mit gleichverteilten Werten zu erzeugen kann man z.B. die `np.random.random()` Funktion nutzen.

```
# Erstellen eines 3 Zufallsarrays mit gleichverteilten Werten zwischen 0 und 1
rand_arr = np.random.random((3,3))
print("3x3 Zufallsmatrix:")
print(rand_arr)
```

```
3x3 Zufallsmatrix:
[[0.79990991 0.10639009 0.58823711]
 [0.04040813 0.64400177 0.37528203]
 [0.52512377 0.27839681 0.45364338]]
```

Für die Normalverteilung gibt es die Funktion:

```
# Erstellen eines 3 Zufallsarrays mit normalverteilten Werten mit dem
Mittelwert 3 und der Standardabweichung 2
rand_arr_norm = np.random.normal(3, 2, size=(3,3))
print("3x3 Zufallsmatrix:")
print(rand_arr_norm)
```

```
3x3 Zufallsmatrix:
[[ 1.74677435  3.7266711  -1.5618844 ]
 [ 3.25980389  6.46448604  2.14740234]
 [ 2.61352701 -0.19861246  2.6116415 ]]
```

Zugriff und Slicing

Auf einzelne Elemente im Array kann über den numerischen Index zuzugreifen. Zu beachten ist das der ***Index bei 0 anfängt zu zählen***.

```
# Zugriff auf Elemente in einem ndarray
print("Element in der zweiten Zeile und dritten Spalte des 2D-Arrays:",
arr_2d[1, 2])
```

```
Element in der zweiten Zeile und dritten Spalte des 2D-Arrays: 6
```

Damit lassen sich auch Werte über den Index direkt zuweisen.

```
# Ändern von Elementen in einem ndarray
arr_2d[1, 2] = 99
print("Matrix nach Änderung:")
print(arr_2d)
```

```
Matrix nach Änderung:
[[ 1  2  3]
 [ 4  5 99]]
```

Bei großen Matrizen möchte man diese oft zerlegen. NumPy hat einige komfortable Operationen dafür. Möchte man zum Beispiel auf die zweite Zeile zugreifen, nutzt man in der Spalte den `:`-Operator

```
arr_2d[1, :]
```

```
array([ 4,  5, 99])
```

Hierbei werden auch die aus Python bekannten negative Indizes unterstützt, mit denen man auf die letzten Elemente zugreifen kann. Um auf die letzte Zeile zuzugreifen, kann man schreiben:

```
arr_2d[-1, :]
```

```
array([ 4,  5, 99])
```

Den Slices kann man auch Werte zuweisen. Um die letzte Zeile auf `99` zu setzen, schreiben wir:

```
arr_2d[-1, :] = 99
print("Matrix nach Änderung:")
print(arr_2d)
```

```
Matrix nach Änderung:
[[ 1  2  3]
 [99 99 99]]
```

Bei der Analyse von Daten ist es oft wichtig, Daten zu filtern. Wollen wir zum Beispiel alle Werte, die nicht `99` sind erhalten, können wir eine Logische Bedingung auf die gleiche Variable als Index verwenden

```
print("Werte kleiner als 99:")
arr_2d[arr_2d<99]
```

```
Werte kleiner als 99:
```

```
array([1, 2, 3])
```

Vektorisierte Funktionen und Lineare Algebra

Es ist möglich, Schleifen zu verwenden, um Berechnungen mit Numpy-Objekten durchzuführen wie bei der Arbeit mit Listen in Python. Allerdings sollte man stattdessen immer vektorisierte Operationen von Numpy verwenden, wenn möglich. Diese sind deutlich performanter als Schleifen und meist auch zu programmieren und einfacher zu lesen.

Numpy bietet eine Vielzahl von vektorisierten Funktionen und Operatoren, die als universelle Funktionen bezeichnet werden. Zum Beispiel mathematische Operationen für die effiziente Rechnung mit Vektoren und Matrizen, was unteranderem für die lineare Algebra sehr sinnvoll ist.

So kann man die Vektoraddition und skalare Multiplikation einfach mit den vektoriellen Operatoren +, - und * durchführen. Nehmen wir als Beispiel die Vektoren a, b, c:

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
c = np.array([9, 10, 11, 12])
```

Wir können zum Beispiel a und b addieren durch:

```
a + b
```

```
array([ 6,  8, 10, 12])
```

oder eine skalare Multiplikation ausführen mit

```
2 * a
```

```
array([2, 4, 6, 8])
```

und das Quadrat aller Elemente berechnen durch

```
a**2
```

```
array([ 1,  4,  9, 16])
```

Wir können auch zeigen, dass a, b, c nicht linear unabhängig sind, indem wir zeigen, dass: $2b - c - a = 0$

```
2*b - c - a
```

```
array([0, 0, 0, 0])
```

Ähnlich wie wir 1-D-Arrays als Vektoren verwendet haben, können wir 2-D-Arrays als Matrizen verwenden. Die Operationen +, - und * funktionieren wie erwartet für zwei Arten von Operationen - elementweise Addition, Subtraktion und Multiplikation - sowie für die Addition, Subtraktion von Konstanten und Multiplikation mit einem Skalar. Hier sind ein paar Beispiele:

```
A = np.array([[1, 2], [3, 4]])
A + 10  # Addition einer Konstanten
```

```
array([[11, 12],
       [13, 14]])
```

```
A * 10  # Skalare Multiplikation
```

```
array([[10, 20],
       [30, 40]])
```

Es gibt noch ein paar andere praktische Möglichkeiten. `np.diag` erzeugt entweder eine diagonale Matrix eines gegebenen Vektors oder gibt, falls eine Matrix gegeben ist, deren Diagonale zurück:

```
np.diag([1, 2, 3])  # Erzeugt eine Diagonal Matrix mit der angegebenen
Diagonalen
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

```
np.diag(A)  # Gibt die Diagonale der Matrix zurück
```

```
array([1, 4])
```

Auch das Transponieren von Vektoren und Matrizen ist einfach mit `.T` erreichbar

```
A.T  # Transponierte Matrix A
```

```
array([[1, 3],
       [2, 4]])
```

Skalarprodukt und Matrizenmultiplikation

Das Skalarprodukt (Punktprodukt) zweier Vektoren kann mit dem `@`-Symbol als Multiplikationszeichen durchgeführt werden, es gibt auch die Funktion `np.dot()` und die Methode `.dot()`.

Wir erstellen zum Beispiel Vektoren $a = (1, 2)$ und $b = (11, 12)^T$:

```
a = np.array([1, 2])  # Zeilenvektor
b = np.array([[11], [12]])  # Spaltenvektor
```

und berechnen

```
a @ b  # Inneres (Matrix) Produkt
```

```
array([35])
```

Achtung: Elementweise Multiplikation vs. Matrizenmultiplikation

Es ist wichtig zu wissen, dass das gewöhnliche Multiplikationszeichen * kein Matrizenmultiplikation ist, sondern eine elementweise Multiplikation mit bestimmten Regeln zur Behandlung von Dimensionsunterschieden.

Da b transponiert ist ergibt sich zum Beispiel

```
a * b  # elementweises Produkt (Übertragung, da die Abmessungen nicht übereinstimmen)
```

```
array([[11, 22],
       [12, 24]])
```

Dies ist nicht das Matrixprodukt! Beachten Sie auch, dass wir hier keinen Fehler erhalten haben (obwohl das manchmal vorkommen kann), da die Berechnung immer noch gültig ist, nur nicht das, was wir hier erhalten wollen.

Genau wie bei Vektoren kann das Matrizenmultiplikation mit `@` und nicht mit * bei Matrizen durchgeführt werden.

```
a @ A
```

```
array([ 7, 10])
```

Auch ist zu beachten, dass die Matrizenmultiplikation nicht kommutativ ist, also die Reihenfolge der Matrizen bei der Produktbildung nicht vertauscht werden darf und zu anderen Ergebnissen führt.

```
A @ a
```

```
array([ 5, 11])
```

Gleichungssysteme mit linearer Algebra lösen

Gleichungssysteme lassen sich numerisch einfach mit Matrizen und Vektoren lösen. Nehmen wir das folgende Beispiel eines Gleichungssystems mit drei Unbekannte und drei Gleichungen.

Wir können dieses Gleichungssystem auch als Matrizenprodukt darzustellen, in dem wir die bekannten Koeffizienten in der Matrix $\mathbf{A}$ von den unbekannten Variablen im Vektor $\mathbf{x}$ trennen und den bekannten Ergebnisvektor $\mathbf{b}$ zuweisen.

$$\mathbf{Ax} = \mathbf{b}$$

Angewandt auf die obigen Gleichungen (1)-(3) erhalten wir die Matrixform:

$$\begin{bmatrix} 5 & 3 & 0 \\ 1 & 2 & 3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Die Koeffizientenmatrix $\mathbf{A}$ enthält alle Konstanten, die mit Ihren unbekannten Variablen x_1, x_2 und x_3 multipliziert werden. Der Ergebnisvektor $\mathbf{y}$ enthält alle bekannten Konstanten, die nicht mit Ihren unbekannten Variablen x_1, x_2 und x_3 multipliziert werden. Schließlich enthält der Vektor $\mathbf{x} = [x_1, x_2, x_3]$ die unbekannten Werte.

Dies können wir mit linearer Algebra lösen indem wir die obige Gleichung (4) mit der inversen Matrix $\mathbf{A}^{-1}$ von links multiplizieren, um die Gleichung nach $\mathbf{x}$ umzustellen:

Dieses Gleichungssystem ist immer dann lösbar, wenn eine Lösung für die inverse Matrix $\mathbf{A}^{-1}$ existiert, was der Fall ist wenn deren Determinante $\det \mathbf{A} \neq 0$ nicht null ist.

Wenn wir das gelernte auf Numpy ergibt sich folgender Lösungsweg. Zuerst definieren wir unsere Koeffizientenmatrix $\mathbf{A}$ und den Ergebnisvektor $\mathbf{b}$.

```
A = np.array([[5, 3, 0], [1, 2, 3], [1, 1, 1]])
b = np.array([1, 2, 3])
```

Wir prüfen die Determinante der Matrix. Hierfür nutzen wir die Funktion `det()` aus der Bibliothek `np.linalg` für Lineare Algebra in *Numpy*:

```
np.linalg.det(A)  # Gibt die Determinante der Matrix zurück
```

```
np.float64(1.0000000000000002)
```

Die Determinante ist nicht 0, also können wir die Inverse der Matrix berechnen.

Dies geschieht mit der Funktion `inv()` wie folgt:

```
A_inv = np.linalg.inv(A)  # Berechnet die Inverse von A
```

Tip

Nicht alle Matrizen sind invertierbar, und wenn Sie versuchen, die Inverse einer Matrix mit Determinante 0 zu berechnen, wird die Funktion `inv()` eine `LinAlgError`-Ausnahme auslösen.

Letztendlich können wir den gesuchten $\mathbf{x}$-Vektor nach der obigen Gleichung (5) bestimmen

```
x = A_inv @ b
x
```

```
array([ 20., -33.,  16.])
```

Gleichungssysteme formal lösen mit *SymPy*

Bisher haben wir betrachtet, wie wir Gleichungssysteme numerisch lösen können. In manchen Situationen ist es jedoch notwendig eine formale Lösung für eine Berechnung algebraisch zu finden. Hier hilft das Python Paket *SymPy*.

Definieren wir formal unser Gleichungssystem aus Gleichung (3) als

```
from sympy import *

A = MatrixSymbol('A', 3, 3).as_explicit()
x = MatrixSymbol('x', 3, 1).as_explicit()
b = MatrixSymbol('b', 3, 1).as_explicit()
```

So erhalten wir keine numerischen, sondern formale Matrizen und Vektoren.

```
A
```

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{bmatrix}$$

```
x
```

$$\begin{bmatrix} x_{0,0} \\ x_{1,0} \\ x_{2,0} \end{bmatrix}$$

Hierfür können wir jetzt zum Beispiel algebraisch die Lösung der Determinante bestimmen

```
A.det()
```

$$A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2,0} + A_{0,2}A_{1,0}A_{2,1} - A_{0,2}A_{1,1}A_{2,0}$$

oder die Inverse der Matrix

```
A.inv()
```

$$\begin{bmatrix} \frac{A_{1,1}A_{2,2} - A_{1,2}A_{2,1}}{A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2,0} + A_{0,2}A_{1,0}A_{2,1} - A_{0,2}A_{1,1}A_{2,0}} & \frac{-A_{0,1}A_{2,2} + A_{0,2}A_{2,1}}{A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2} \\ \frac{-A_{1,0}A_{2,2} + A_{1,2}A_{2,0}}{A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2,0} + A_{0,2}A_{1,0}A_{2,1} - A_{0,2}A_{1,1}A_{2,0}} & \frac{A_{0,0}A_{2,2} - A_{0,2}A_{2,0}}{A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2} \\ \frac{A_{1,0}A_{2,1} - A_{1,1}A_{2,0}}{A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2,0} + A_{0,2}A_{1,0}A_{2,1} - A_{0,2}A_{1,1}A_{2,0}} & \frac{-A_{0,0}A_{2,1} + A_{0,1}A_{2,0}}{A_{0,0}A_{1,1}A_{2,2} - A_{0,0}A_{1,2}A_{2,1} - A_{0,1}A_{1,0}A_{2,2} + A_{0,1}A_{1,2}A_{2} \end{bmatrix}$$

Auch können wir das Gleichungssystem komplett algebraisch lösen lassen.

```
A.solve(b)
```

$$\begin{bmatrix} \frac{A_{0,1}A_{1,2}b_{2,0}-A_{0,1}A_{2,2}b_{1,0}-A_{0,2}A_{1,1}b_{2,0}+A_{0,2}A_{2,1}b_{1,0}+A_{1,1}A_{2,2}b_{0,0}-A_{1,2}A_{2,1}b_{0,0}}{A_{0,0}A_{1,1}A_{2,2}-A_{0,0}A_{1,2}A_{2,1}-A_{0,1}A_{1,0}A_{2,2}+A_{0,1}A_{1,2}A_{2,0}+A_{0,2}A_{1,0}A_{2,1}-A_{0,2}A_{1,1}A_{2,0}} \\ \frac{-A_{0,0}A_{1,2}b_{2,0}+A_{0,0}A_{2,2}b_{1,0}+A_{0,2}A_{1,0}b_{2,0}-A_{0,2}A_{2,0}b_{1,0}-A_{1,0}A_{2,2}b_{0,0}+A_{1,2}A_{2,0}b_{0,0}}{A_{0,0}A_{1,1}A_{2,2}-A_{0,0}A_{1,2}A_{2,1}-A_{0,1}A_{1,0}A_{2,2}+A_{0,1}A_{1,2}A_{2,0}+A_{0,2}A_{1,0}A_{2,1}-A_{0,2}A_{1,1}A_{2,0}} \\ \frac{A_{0,0}A_{1,1}b_{2,0}-A_{0,0}A_{2,1}b_{1,0}-A_{0,1}A_{1,0}b_{2,0}+A_{0,1}A_{2,0}b_{1,0}+A_{1,0}A_{2,1}b_{0,0}-A_{1,1}A_{2,0}b_{0,0}}{A_{0,0}A_{1,1}A_{2,2}-A_{0,0}A_{1,2}A_{2,1}-A_{0,1}A_{1,0}A_{2,2}+A_{0,1}A_{1,2}A_{2,0}+A_{0,2}A_{1,0}A_{2,1}-A_{0,2}A_{1,1}A_{2,0}} \end{bmatrix}$$

Prinzipiell können wir die Gleichungen (1)-(3) auch direkt algebraisch lösen, ohne sie vorher in eine Matrizendarstellung zu bringen.

```
x1, x2, x3 = symbols("x1, x2, x3")
solve([
    5 * x1 + 3 * x2 - 1,
        x1 + 2 * x2 + 3 * x3 - 2,
        x1 + x2 + x3 - 3
    ], [x1, x2, x3], dict=True)
```

$[\{x_1 : 20, x_2 : -33, x_3 : 16\}]$

Weitere Funktionen in *SimPy* erlauben das Vereinfachen von Formeln. Insbesondere wenn es um komplexe Brüche oder trigonometrische Funktionen geht, hat man nicht alle Ersetzungsmuster im Kopf und kann das von *SimPy* lösen lassen.

```
x = symbols("x")

simplify(sin(x)**2 + cos(x)**2)
```

1

```
simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
```

$x - 1$

In gleicher Weise lassen sich vereinfachte Formen erweitern

```
expand((x + 1)**2)
```

$x^2 + 2x + 1$

Das funktioniert auch für Differentialgleichungen. Von einfachen Lösungen wie

```
diff(cos(x), x)
```

$-\sin(x)$

bis zu komplexen Ausdrücken, wie diese Differentialgleichung zweiter Ordnung

```
f = symbols("f", cls=Function)
```

```
diffeq = Eq(f(x).diff(x, x) - 2*f(x).diff(x) + f(x), sin(x))
diffeq
```

$f(x) - 2\frac{d}{dx}f(x) + \frac{d^2}{dx^2}f(x) = \sin(x)$

die wir mit der Funktion `dsolve` formal lösen können

```
dsolve(diffeq)
```

$f(x) = (C_1 + C_2 x)e^x + \frac{\cos(x)}{2}$

als auch für bestimmte Randbedingungen für $f(x)$ wie $f(0) = 1$ und $f(2) = 3$

```
dsolve(diffeq, ics={f(0): 1, f(2): 3})
```

$f(x) = \left(\frac{x(-e^2 - \cos(2) + 6)}{4e^2} + \frac{1}{2}\right)e^x + \frac{\cos(x)}{2}$

Das funktioniert auch für Differentialgleichungssysteme

```
f, g = symbols("f g", cls=Function)

x = symbols("x")

eqs = [Eq(f(x).diff(x), g(x)), Eq(g(x).diff(x), f(x))]
eqs
```

$\left[\frac{d}{dx}f(x) = g(x), \frac{d}{dx}g(x) = f(x)\right]$

mit der formalen Lösung

```
dsolve(eqs, [f(x), g(x)])
```

$[f(x) = -C_1 e^{-x} + C_2 e^x, g(x) = C_1 e^{-x} + C_2 e^x]$

und einer spezifischen Lösung mit gegebenen Randbedingungen $f(0) = 1$ und $g(2) = 3$

```
dsolve(eqs, [f(x), g(x)], ics={f(0): 1, g(2): 3})
```

$\left[f(x) = \frac{(1+3e^2)e^x}{1+e^4} - \frac{(-e^4+3e^2)e^{-x}}{1+e^4}, g(x) = \frac{(1+3e^2)e^x}{1+e^4} + \frac{(-e^4+3e^2)e^{-x}}{1+e^4}\right]$

Bibliographie