CompMath

Elias Karabelas

28. Oktober 2020

Matrizen

Matrix $\boldsymbol{A}$ definiert als „Rechteck von Zahlen“

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{pmatrix}, \qquad a_{ij} \in \mathbb{R}, \qquad i = 1, \dots, m,\ j = 1, \dots, n,$$

$\boldsymbol{A}[i,j] = a_{ij}$

Man kann auch sagen $\boldsymbol{A} \in \mathbb{R}^{m \times n}$. m ist Anzahl „Zeilen“ und n ist Anzahl „Spalten“. Spezialfälle: **Zeilenvektor** $\vec{v}^\top \in \mathbb{R}^{1 \times n}$ ist eine Matrix mit 1 Zeile und n Spalten, **Spaltenvektor** $\vec{v} \in \mathbb{R}^{m \times 1} = \mathbb{R}^m$ ist eine Matrix mit m Zeilen und 1 Spalte.
Die quadratische Matrix ($m = n$) $\boldsymbol{I}$ definiert durch

$$I[i,j] = \begin{cases} 1 & \text{wenn } i = j \\ 0 & \text{sonst} \end{cases}$$

heißt m-dimensionale **Einheitsmatrix** (nur 1 auf der Diagonale sonst 0).

Rechenregeln

Wenn Matrizen $\boldsymbol{A}$, $\boldsymbol{B}$ gleiche Anzahl Spalten und gleiche Anzahl Zeilen haben macht folgendes Sinn

$$\boldsymbol{C} = C[i,j] = \lambda \boldsymbol{A} \pm \mu \boldsymbol{B} = \lambda a_{ij} \pm \mu b_{ij}, \lambda, \mu \in \mathbb{R}.$$

Diese Rechenregeln sind auch kommutativ! Anders ist es für die Multiplikation: Geht nur für $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ und $\boldsymbol{B} \in \mathbb{R}^{k \times n}$.

$$\boldsymbol{C} = C[i,j] = (\boldsymbol{A} \cdot \boldsymbol{B})[i,j] := \sum_{s=1}^{k} a_{is} b_{sj}$$

Achtung: Diese Operation ist **nicht** kommutativ, $\boldsymbol{A} \cdot \boldsymbol{B} \neq \boldsymbol{B} \cdot \boldsymbol{A}$.

Ein Spezialfall des Matrix-Matrix Produktes ist das Matrix-Vektor-Produkt: Für $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ und $\vec{x} \in \mathbb{R}^{n \times 1}$ haben wir

$$\begin{aligned} \boldsymbol{A}\vec{x} &= \vec{y} \in \mathbb{R}^m, \\ y[i] &= \sum_{j=1}^{n} a_{ij} x[j]. \end{aligned}$$

Weiters interessant ist die **transponierte Matrix** zur Matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ $\boldsymbol{A}^\top \in \mathbb{R}^{n \times m}$ definiert durch

$$\boldsymbol{A}^\top[i, j] := \boldsymbol{A}[j, i].$$

Inverse Matrix

Verallgemeinert das Konzept des „Bruchs“ auf Matrizen. Für eine Zahl weiß man dass die Gleichung

$$x \cdot y = 1$$

für ein gegebenes $x \in \mathbb{R}$ durch $y = \frac{1}{x}$ gelöst wird. Für Matrizen heißt das, wir suchen eine **quadratische Matrix** $\boldsymbol{A}^\dagger$ sodass

$$\begin{aligned} \boldsymbol{A} \cdot \boldsymbol{A}^\dagger &= \boldsymbol{I} \\ \boldsymbol{A}^\dagger \cdot \boldsymbol{A} &= \boldsymbol{I} \end{aligned}$$

Die so definierte Matrix heißt **inverse Matrix** und wird auch als $\boldsymbol{A}^{-1}$ bezeichnet.

Zusammenhang lineare Gleichungssysteme und Matrizen

Beispiel aus der Schule

$$\begin{aligned} x + y + z &= 4 \\ 2x - 3y + z &= 1 \\ x - z &= 2 \end{aligned}$$

anders geschrieben

$$\begin{aligned} 1 * x + 1 * y + 1 * z &= 4 \\ 2 * x - 3 * y + 1 * z &= 1 \\ 1 * x + 0 * y - 1 * z &= 2 \end{aligned}$$

Dies kann man jetzt als Matrix-Vektor Produkt schreiben

$$\begin{aligned}
\boldsymbol{A}\vec{x} &= \vec{f}, \\
\boldsymbol{A} &= \begin{pmatrix} 1 & 1 & 1 \\ 2 & -3 & 1 \\ 1 & 0 & 1 \end{pmatrix}, \\
\vec{x} &= \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \\
\vec{f} &= \begin{pmatrix} 4 \\ 1 \\ 2 \end{pmatrix}.
\end{aligned}$$

Die Lösung dieses Gleichungssystems kann mit der inversen matrix realisiert werden

$$\vec{x} = \boldsymbol{A}^{-1}\vec{f}$$

Strukturierte Programmierung mit Matlab / Octave

Um den Programmfluss besser zu steuern bedarf es einiger Konstrukte die das Leben erleichtern:

Alternativen-Steuerung (IFs) Oft kommt es vor, dass ein Programm abhängig vom Zustand einer Variablen unterschiedliche Tätigkeiten tun soll. Als Beispiel denken Sie sich dass Ihr Programm für 0 etwas tun soll und für 1 etwas anders ⇒ `if`-Statement

```
if x ~= 0
   awesome_func_that_handles_that_case(x)
end
```

oder

```
if x > 0 && mod(x,2) == 0
   awesome_func_that_handles_that_case(x)
end
```

Das kann mitunter zuwenig sein, man möchte auch steuern wenn das Gegenteil eintrit ⇒ `if-else`

```
if x ~= 0
   awesome_func_that_handles_this_case(x)
else
   awesome_func_that_handles_that_case(x)
end
```

Manchmal hat man nochmehr möglichkeiten ⇒ `if-elseif-else`

```
if x == 0
   awesome_func_that_handles_zero_case(x)
elseif x == 1
    awesome_func_that_handles_first_case(x)
elseif x == 2
    awesome_func_that_handles_second_case(x)
else
   awesome_func_that_handles_alternative
end
```

Zählobjekte, Schleifen Wenn man wiederholende Prozesse in seinem Code hat bieten sich Schleifen an. Beispiel Summe der ersten n Zahlen. Dumme variante:

```
sum = 1;
sum = 1 + 2;
sum = 1 + 2 + 3;
%etc...
```

Eine Schleife kann hier helfen

```
sum = 0;
n = 100;
% k ist Zählvariable existiert nur innerhalb der for schleife
for k = 1:1:n
    sum = sum + k;
end
%etc...
```

Ein anderer Schleifen-Typ ist die `while`-Schleife (auch abweisende schleife genannt). Diese wiederholt etwas solange ein Kriterium erfüllt ist

```
sum = 0;
n = 100;
k = 1;
% k ist Zählvariable existiert nur innerhalb der for schleife
while k <= n
    sum = sum + k;
    k = k + 1;
end
%etc...
```