Numerik

Ingenieurinformatik Teil 2, Sommersemester 2026

David Straub

Gliederung

1. **Einführung in Matlab**
2. **Arbeiten mit Arrays**
3. **Funktionen und Kontrollstrukturen**
4. **Analysis** (Polynome, Ableitung, Integration, …)
5. **Lineare Algebra** (Gleichungssysteme, Eigenwerte, …)
6. **Differentialgleichungen**
7. **Einführung in Simulink**

3. Funktionen und Kontrollstrukturen

Überblick

Funktionen und Kontrollstrukturen sind aus Python bekannt:

- Funktionen mit Parametern und Rückgabewerten
- `if / elif / else`, `for`, `while`, `break`, `continue`

Die Konzepte sind in Matlab dieselben – die **Syntax** unterscheidet sich.

Ziel: Unterschiede zur Python-Syntax kennenlernen und typische Fallstricke vermeiden.

Skripte und Funktionen

Skripte

Ein **Skript** ist eine `.m`-Datei mit einer Folge von Befehlen – ähnlich einem Python-Skript.

Wichtig: Ein Matlab-Skript teilt den **Workspace** mit dem Command Window – im Gegensatz zu einer Funktion, die einen eigenen Workspace hat.

```
% quadrat.m
x = 4;
y = x ^ 2
```

```
>> x = 10            % im Command Window gesetzt
>> quadrat           % Skript aufrufen
y =
    16              % x aus dem Workspace wurde überschrieben!
```

> Ein Skript kann alle Workspace-Variablen lesen, verändern und löschen. Eine Funktion dagegen hat einen **eigenen Workspace**.

Funktionen – Python vs. Matlab

	Python	**Matlab**
Definition	`def umfang(r):`	`function umf = umfang(r)`
Rückgabe	`return 2 * math.pi * r`	`umf = 2 * pi * r;`
Abschluss	Einrückung	`end`
Datei	beliebiger Name	**muss** `umfang.m` heißen
`pi`	`import math` nötig	eingebaut

```
% umfang.m
function umf = umfang(r)
% UMFANG: Kreisumfang berechnen
% umf = umfang(r)  –  r: Radius
    umf = 2 * pi * r;
end
```

Mehrere Rückgabewerte

In Python gibt man ein Tupel zurück. In Matlab werden Ausgabevariablen **im Funktionskopf** deklariert:

```
% kreis.m
function [umf, fl] = kreis(r)
    umf = 2 * pi * r;
    fl  = pi * r .^ 2;
end
```

```
>> [u, f] = kreis(3)
u =   18.8496
f =   28.2743

>> [u, ~] = kreis(3)   % zweiten Rückgabewert ignorieren
```

> **Kapselung:** Variablen in der Funktion sind lokal – aus dem Workspace nicht sichtbar, und umgekehrt.

Funktionen – Ausprobieren ?

Übersetzen Sie diese Python-Funktion nach Matlab:

```
def widerstand(L, A):
    """R = rho * L / A  (Kupfer, rho = 1.68e-8 Ohm*m)"""
    rho = 1.68e-8
    return rho * L / A
```

- Was muss der Dateiname sein?
- Rufen Sie die Funktion für `L=10, A=1e-6` auf (10 m Kupferdraht, 1 mm²).
- Funktioniert die Funktion für Vektoren (z. B. verschiedene Längen)?

Lokale Funktionen

Alternative zu separaten `.m`-Dateien: Hilfsfunktionen direkt in ein Skript schreiben – kein eigener Dateiname nötig.

Wann sinnvoll? Wenn eine Funktion nur in einem bestimmten Skript gebraucht wird und nicht allgemein wiederverwendbar ist.

```
% berechnung.m
r = 5;
fprintf('Umfang: %.4f\n', umfang(r))

function umf = umfang(r)    % lokale Funktion – nur hier aufrufbar
    umf = 2 * pi * r;
end
```

Lokale Funktionen – Hinweis

In älterer Literatur und in Matlab-Dokumentation findet man zwei Begriffe:

- **Subfunctions** – alter Begriff (vor R2016a)
- **Lokale Funktionen** *(local functions)* – aktueller Begriff

Beides meint dasselbe.

Position im Skript: - Ab **R2024a**: Lokale Funktionen können an beliebiger Stelle im Skript stehen - **Vor R2024a**: Lokale Funktionen müssen **nach** dem gesamten Skriptcode stehen

Was passiert hier? (1) 🤔

```
% beispiel1.m
function f1(x)         % lokale Funktion
    x = 2 * x
end

a = [1, 2];
f1(a)
a
```

Was ist der Wert von `a` nach dem Aufruf?

Was passiert hier? (2) 🤔

```
% beispiel2.m
function y = f2(x)     % lokale Funktion
```

```
    x = 2 * x;
    y = x;
end

a = [1, 2];
b = f2(a)
a
```

Was sind die Werte von `a` und `b`?

Was passiert hier? (3) 🤔

```
% beispiel3.m
function y = quadrat(x)    % lokale Funktion
    y = x ^ 2;
end

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

Was ist das Ergebnis? Warum?

Was passiert hier? (4) 🤔

```
% beispiel4.m
function umf = umfang(r)   % lokale Funktion
    r = 2 * pi * r;
    umf = r;
end

r = 10;
umfang(3)
r
```

Was ist der Wert von `r` im Workspace nach dem Aufruf?

Kontrollstrukturen

Übersicht: Syntax-Vergleich

Konzept	**Python**	**Matlab**
`if`-Zeile	`if cond:`	`if cond`
`if`-Block	Einrückung	`end`
`elif`	`elif`	`elseif`
`for`-Schleife	`for k in range(1, n+1):`	`for k = 1:n … end`
`while`-Schleife	`while cond:`	`while cond … end`

Alle Blöcke enden mit `end` – anders als in Python, wo die Einrückung den Block bestimmt.

Vergleichsoperatoren und Verzweigungen

Vergleichs- und logische Operatoren (Skalare)

Vergleiche – fast identisch zu Python, außer `~=`:

Matlab	`<`	`>`	`<=`	`>=`	`==`	`~=`
Python	`<`	`>`	`<=`	`>=`	`==`	`!=`

Logische Operatoren für Skalare (in `if`- und `while`-Bedingungen):

Matlab	Python	Bedeutung
`&&`	`and`	UND, short-circuit
`||`	`or`	ODER, short-circuit
`~`	`not`	NICHT

`if` – Syntax

```
if Bedingung
    Anweisungen
elseif Bedingung-2
    Anweisungen-2
else
    Anweisungen-sonst
end
```

```
note = 2;
if note == 1
    disp('sehr gut')
elseif note <= 3
    disp('bestanden')
else
    disp('nicht bestanden')
end
```

Python: `elif` → Matlab: `elseif`. Zu jedem `if` gehört genau ein `end`.

`if` – Ausprobieren

Schreiben Sie `betrag.m` – Betrag einer Zahl **ohne** `abs`:

$$|x| = \begin{cases} x & x \geq 0 \\ -x & \text{sonst} \end{cases}$$

- Testen Sie mit `betrag(-5)`, `betrag(0)`, `betrag(3)`.
- Funktioniert `betrag([1, -2, 3])`? Warum (nicht)?

  `if` in Matlab wertet eine **einzelne Bedingung** aus – keine automatische elementweise Auswertung bei Arrays.

Vergleichsoperatoren für Arrays

Vergleiche mit Arrays

Vergleichsoperatoren (`<`, `>`, `==`, `~=`, …) wirken **elementweise** auf Arrays und liefern ein **logical array**:

```
x = [1, 5, 3, 8, 2];
x > 4           % [0  1  0  1  0]
```

Logische Operatoren für Arrays (elementweise):

Matlab	Python/NumPy	Bedeutung
`&`	`&`	UND
`|`	`|`	ODER
`~`	`~`	NICHT

Beispiel: Logische Indizierung

Aus Kapitel 2 – jetzt auch mit logischen Operatoren kombinierbar:

```
x = [1, 5, 3, 8, 2];
x(x > 4)            % [5  8]       – Elemente > 4
x(~(x > 4))         % [1  3  2]    – Elemente ≤ 4
x(x > 2 & x < 6)    % [5  3]       – zwischen 2 und 6
sum(x > 4)          % 2            – Anzahl der Treffer
x(x > 4) = 0        % [1  0  3  0  2]  – Elemente ersetzen
```

Logische Indizierung ist oft kürzer und schneller als eine `for`-Schleife mit `if`.

Was passiert hier? 🤔

```
x = [1, 5, 3, 8, 2];
x(x > 2 & x < 6)
```

```
x(~(x > 4))
```

```
x(x < 2 | x > 6)
```

Was ist jeweils das Ergebnis?

Short-Circuit: && und ||

Auswertung stoppt, sobald Ergebnis feststeht – verhindert Fehler:

```
v = [];
result = ~isempty(v) && v(1) > 10   % sicher: v(1) wird nicht ausgewertet
result = ~isempty(v) & v(1) > 10    % FEHLER: Index out of bounds!
```

`&` und `|` short-circuiten in `if`/`while` zwar auch – aber können bei nicht-skalaren Ausdrücken unerwartete Ergebnisse liefern.

> Verwenden Sie in `if`/`else` **immer** `&&` und `||`! `&` und `|` nur für elementweise Array-Operationen

`for`-Schleifen

`for`-Schleife

Python: `for k in range(1, n+1)` → Matlab: `for k = 1:n`

```
n = 10;
summe = 0;
for k = 1:n
    summe = summe + k;
end
disp(summe)   % 55
```

Schrittweite angeben:

```
for k = 10:-1:1          % rückwärts
for k = 0:0.1:1          % Gleitkomma-Schritte
```

> Verwenden Sie `k` statt `i` oder `j` als Zählvariable – in Matlab sind `i` und `j` die imaginäre Einheit!

`for` – Ausprobieren

Schreiben Sie `fakultaet.m` mit einer `for`-Schleife:

$$n! = 1 \cdot 2 \cdots n, \qquad 0! = 1$$

- Was gibt `fakultaet(0)` zurück? Was sollte es ergeben?
- Vergleichen Sie mit `factorial(n)`.
- Ab welchem `n` erhalten Sie `Inf`?

Bonus: Lösen Sie dieselbe Aufgabe in einer Zeile ohne Schleife.

while-Schleifen

while-Schleife

```
while Bedingung
    Anweisungen
end
```

```
x = 5; xn = 10;
while abs(xn - x/xn) > 1e-10
    xn = 0.5 * (xn + x/xn);   % Heron-Iteration
end
```

Solange die Bedingung `true` ist, wird der Block wiederholt – wie in Python.

break und continue

break – Schleife sofort verlassen:

```
for k = 1:100
    if k^2 > 50, break; end
end
```

continue – aktuellen Durchlauf überspringen:

```
for k = 1:10
    if mod(k,2) == 0, continue; end
    disp(k)   % gibt 1 3 5 7 9 aus
end
```

Funktioniert in `for`- und `while`-Schleifen – identisch zu Python.

Schleifen vs. Vektoroperationen

Matlab ist für vektorisierte Berechnungen optimiert – Schleifen sind oft **deutlich** langsamer:

```
n = 1e6;  x = rand(1, n);

tic
for k = 1:n, y(k) = sin(x(k)); end   % Schleife
```

```
toc   % z.B. 0.8 s

tic
y = sin(x);   % vektorisiert
toc   % z.B. 0.01 s
```

Wann Schleifen sinnvoll? Wenn jede Iteration vom Ergebnis der vorherigen abhängt (z. B. Heron, numerische Integration).

Zusammenfassung & Ausblick

Funktionen - `function [out] = name(in) ... end` – Dateiname = Funktionsname - Mehrere Rückgabewerte: `function [a, b] = name(x)` - Eigener Workspace, Call by Value, Dokumentation mit `%`

Kontrollstrukturen – alle Blöcke enden mit `end` - Vergleiche: `<  >  <=  >=  ==  ~=` - Logisch: `&&  ||` (Skalare), `&  |` (Arrays), `~` (Negation) - `if / elseif / else` - `for k = start:schritt:ende`, `while`, `break`, `continue`