# Programmieren mit Matlab

- Verwenden von *Skripten* und *Funktionen*
- Nutzen des Editors und des Debuggers
- Kontrollstrukturen (`for`, `while`, `if`, etc.)
- Bildschirmausgabe
- Lesen und Schreiben von Dateien

# Matlab Skripte

Speichern mehrere Matlab Befehle in einer Textdatei (Dateiendung `*.m`)

```
% dies ist ein Kommentar
a = 3*pi;
b = sin(a/2);
c = b^2;
```

Sämtliche Variablen werden im Workspace angelegt

Unterdrückung der Bildschirmausgabe mit abschließendem *Semicolon* `;`

Zeilenweise Kommentare beginnen mit einem Prozentzeichen `%`

Ausführen der Matlab Skripte

- Eingabe des Dateinamens in Kommandozeile
- Drücken des Run-Buttons in der Symbolleiste des Editors
- Drücken der `<F5>` Taste

# Steuerung des Programmablaufes

## Die `if`-Anweisung

```
if x == 4
    disp('x ist Vier')
else
    disp('x ist nicht Vier')
end
```

Zur Realisierung verschachtelter `if`-Statements existiert das Schlüsselwort `elseif`

## Die `switch-case`-Anweisung

```
switch(x)
    case 1
        disp('x ist Eins')
    case 2
        disp('x ist Zwei')
    otherwise
        disp('x ist weder Eins noch Zwei')
end
```

# Vergleichsoperatoren in Matlab I

| Operator | Vergleich |
|---|---|
| == | gleich |
| ~= | ungleich |
| > | größer |
| >= | größer oder gleich |
| < | kleiner |
| <= | kleiner oder gleich |

Ergebnis von Vergleichsoperatoren ist wahr (`1`) oder falsch (`0`)

```
>> 1 == 1
ans =
  logical
   1
```

# Vergleichsoperatoren in Matlab II

Vergleichsoperatoren funktionieren elementweise

```
>> [7, 4, 2] == [7, 8, 2]
ans =
  1×3 logical array
   1   0   1
```

Prüfen ob alle Werte eines Vektors wahr (d.h. nicht Null) sind

```
>> all([1, 2, 3])
ans =
  logical
   1
```

Prüfen ob mindestens einer der Werte eines Vektors wahr (d.h. nicht Null) sind

```
>> any([0, 2, 0])
ans =
  logical
   1
```

# Verknüpfung der Ergebnisse von Vergleichen

| Operator | Vergleich |
|---|---|
| & & | AND |
| & | AND (auch für Vektoren) |
| \| \| | OR |
| \| | OR (auch für Vektoren) |
| ~ | NOT |

# Iterationen I

Iteration über einen Vektor mittels der `for`-Schleife

```
for i = [3, 4, 5]
    ...
end
```

Iterationsvektor kann während Schleifendurchlauf nicht verändert werden

Häufig lassen sich `for`-Schleifen auch durch Matrixoperationen ersetzen (Vorteil bei Ausführungsgeschwindigkeit)

Beispiel:

- Gegeben Vektor `x` der Länge 10
- Berechnung eines Vektors `y`, der die Quadrate von `x` enthält

```
% for-Schleife (umständlich)
y = [];
for i = 1:length(x)
    y(i) = x(i)^2;
end

% Matrix-Operation (schnell)
y = x.^2;
```

# Iterationen II

Die `while`-Schleife: Wiederholung des Schleifenkörpers solange eine Bedingung erfüllt ist

```
x = 0;
while x < 5
    ...
    x = x+1;
end
```

Das Konstrukt der `do-while`-Schleife (wie z.B. in der Programmiersprache C) existiert in Matlab nicht!

Abbruch von `for`- oder `while`-Schleifen über Schlüsselwort `break`

```
for x = 1:10
    if y == 5
        break
    end
    ...
end
```

Sofortiger Sprung zur nächsten Iteration mittels Schlüsselwort `continue`

```
while x < 100
    if y == 5
        continue
    end
    ...
end
```

# Matlab Funktionen

Programmkonstrukt zur Strukturierung wiederkehrender Aufgaben

Funktionen werden in Textdateien abgelegt (Dateiendung `*.m`)

Aufbau einer Funktion

```
function c = hypothenuse(a, b)
    c_quadrat = a^2 + b^2;
    c = sqrt(c_quadrat);
end
```

Aufruf der Funktion

```
>> hypothenuse(3, 4)
ans =
     5
```

*Vorsicht:* Funktionen werden über Dateinamen identifiziert und nicht über den Funktionsnamen!

Nur eine Funktion pro Datei möglich (Ausnahme: Hilfsfunktionen nur innerhalb der Datei verwendbar)

# Weitere Eigenschaften von Matlab Funktionen

Variablen, die innerhalb der Funktion angelegt werden

- befinden sich im *Function-Workspace*
- werden nach Beendigung der Funktion verworfen

Mehrere Über- und Rückgabeargumente möglich

Erster Blockkommentar dient als Hilfetext

Datei der Funktion muss im aktuellen Verzeichnis oder im Matlab-Suchpfad liegen

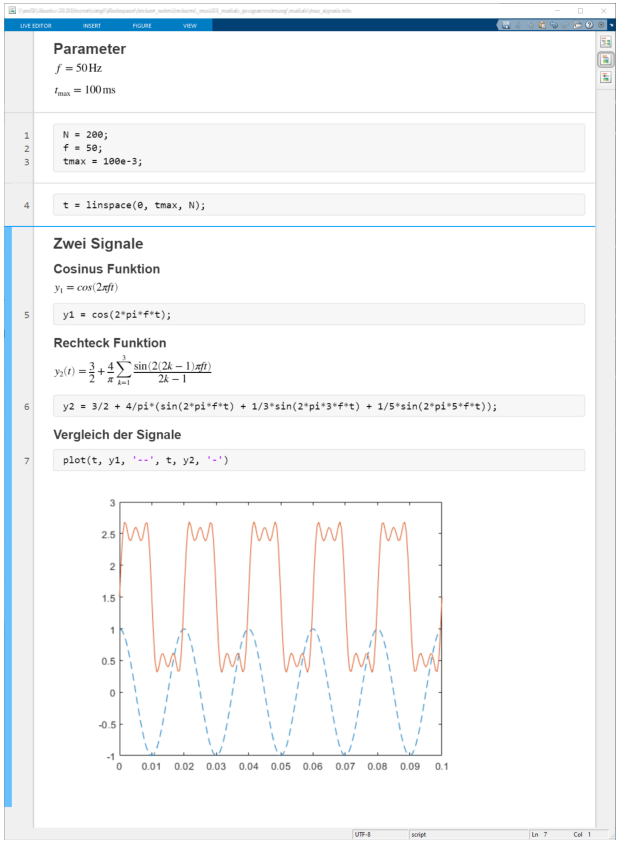
# Beispiel einer Matlab Funktion zur Berechnung von Nullstellen eines Polynoms

```
% nullstellen(c, b, a)
% Berechnung der reellen Nullstellen eines Polynoms
% a*x^2 + b*x + c = 0
% Falls a nicht angegeben ist, wird a=1 angenommen
function [x1, x2] = nullstellen(c, b, a)
    narginchk(2, 3)
    nargoutchk(1, 2)

    if nargin == 2
        a = 1;
    end

    if diskriminante(a, b, c) < 0
        error('Dieses Polynom besitzt keine rein reellen Nullstellen')
    end

    x1 = (-b + sqrt(diskriminante(a, b, c)))/(2*a);

    if nargout == 2
        x2 = (-b - sqrt(diskriminante(a, b, c)))/(2*a);
    end
end

function D = diskriminante(a, b, c)
    D = b^2 - 4*a*c;
end
```

# Matlab Live-Scripts

Matlab Live-Scripts kombinieren die interaktive Verwendung der Konsole mit einem Skript

Kombination verschiedener Darstellungsmöglichkeiten (Überschriften, mathematische Formeln, Plots, etc.)



# Ausgabe von Zeichenketten

## Schreiben von Text in die Konsole

- Schreiben einer Zeile: `disp("Hello World!")`
- Ausgabe eines formatierten Strings: `fprintf("Variable x = %d\n", x)`

## Schreiben von Text in eine Datei

- Öffnen einer Datei: `fid = open("path_and_filename", "w");`
- Schreiben in die Datei: `fprintf(fid, "Variable x = %d\n", x)`
- Schließen der Datei: `fclose(fid)`

# Einlesen von Text und Daten

## Lesen von Textdateien

- Öffnen einer Datei: `fid = open("path_and_filename", "r");`
- Lesen von der Datei: `fscanf(fid, "%s %d")`
- Schließen der Datei: `fclose(fid)`

## Lesen von der Tastatur

- Lesen und Auswerten von Matlab-Ausdrücken: `x = input("Gibt bitte ein Zahl ein: ")`

# Anbinden strukturierter Datenformate

## Lesen und Schreiben von Matlab-Dateien

- Proprietäres Dateiformat für Matlab-Variablen und -Objekte: `*.mat`
- Schreiben des gesamten Workspace in eine Datei: `save("filename.mat")`
- Lesen der Variablen und Objekte von einer mat-Datei: `load("filename.mat")`

## Anbinden weitere Dateiformate

- CSV Dateien (*comma speparated values*): `csvread()` und `csvwrite()`
- Excel Dateien: `xlsread()` und `xlswrite()`
- JSON Dateien (*java script object notation*): `jsondecode()` und `jsonencode()`
- XML Dateien: `xmlread()` und `xmlwrite()`

# Zusammenfassung

- Unterschied zwischen *Skript* und *Funktion*
- Verwenden des Editors und des Debuggers
- Kontrollstrukturen
- Lesen und Schreiben von Dateien