Einführung in Matlab
Grundlagen

- **Matlab auf Uni-Seite herunterladen** (dort ist auch eine Installationsanleitung): **https://software.uni-oldenburg.de/software/matlab/**
- Ausführliche **Dokumentation und Hilfe** in Matlab selbst und auf **http://de.mathworks.com/**

1 Gleitpunkt-Zahlen, Variablen, Vektoren, Arrays, Plots

1.1 Grundlagen für Gleitpunkt-Zahlen

- Im **Command Window** können **Befehle eingegeben** und durch **Drücken der Eingabetaste ausgeführt** werden, z.B. einfache Rechnungen.

```
>> 3+5
ans =
     8
```

- **Übliche Rechenregeln** werden eingehalten, **mehrere Anweisungen** können **durch Komma getrennt** werden

```
>> 3+4*5, (3+4)*5
ans =
    23
ans =
    35
>> 2*0.1^2, 2e-2  % abkürzende Darstellung mit 10er-Exponent 2e-2=2*10^(-2)
ans =
    0.0200
ans =
    0.0200
>> 1/2/3  % wird von links nach rechts abgearbeitet
ans =
    0.1667
```

- **alte Befehle können mit Pfeiltasten** nochmal übernommen werden (auch gezielt nach Eingeben der ersten paar Zeichen), oder auch direkt aus der **Command History** mit Doppelklick.
- Matlab arbeitet mit der üblichen Hardware-Arithmetik **double** = 64 bit = ca. **16 signifikante Dezimalstellen** für **Gleitpunkt-Zahlen** (floating point), zeigt aber nur 4 Stellen nach dem Komma an. Wechsel der Darstellung mit `format long/short`.

```
>> format long
>> 1/2/3
ans =
   0.166666666666667
>> pi
ans =
   3.141592653589793
>> format short
```

- **Abstand zur nächstgrößeren Gleitpunkt-Zahl**, **Größte darstellbare Gleitpunkt-Zahl**, ± **Unendlich**: ±`Inf` (z.B. bei Überlauf), **Not a Number**: `NaN`

```
>> eps(1), eps(100), eps(1e16)
ans =
   2.2204e-16
ans =
   1.4211e-14
ans =
     2
>> realmax
ans =
  1.7977e+308
>> 2*realmax
ans =
   Inf
>> inf-3, inf-inf, -1/0, 0/0
ans =
   Inf
ans =
   NaN
ans =
  -Inf
ans =
   NaN
```

- Matlab kennt alle **elementaren Funktionen** wie $\sin, \exp, \ln, \sqrt{\cdot}$ (allerdings ist `log` der natürliche Logarithmus, `ln` gibt es nicht, aber z.B. auch `log10`). Eine **Liste findet man in der Hilfe unter Matlab->Mathematics->Elementary Math**

```
>> sin(pi)
ans =
   1.2246e-16
>> log(exp(1))
ans =
     1
>> sqrt(2)
ans =
    1.4142
```

1.2 Variablen

- Das **zuletzt berechnete Ergebnis** wird automatisch in Variable `ans` im **Workspace** gespeichert.

```
>> ans
ans =
    1.4142
```

- Eigene **Variablen definieren und Werte zuweisen** durch

```
Var_Name=Ausdruck
```

wobei der Variablenname mit einem Buchstaben anfangen muss, gefolgt von Buchstaben, Zahlen oder Unterstrich, wobei Groß-und Kleinschreibung unterschieden wird (genaueres in Hilfe unter Matlab->Language Fundamentals->Entering Commands->Concepts->Variable Names).
Die **Maximale Variablenlänge** ist

```
>> namelengthmax
ans =
    63
```

- **Rechnen mit Variablen**

```
>> x=1, y=2, z=x+y
x =
     1
y =
     2
z =
     3
```

- **Wert der Variable wird erst nach erneuter Zuweisung geändert**

```
>> x+1
ans =
     2
>> x
x =
     1
>> x=x+1
x =
     2
```

1.3 Vektoren, Arrays

- **Vektoren** sind **1-dimensionale Arrays** und werden **durch eckige Klammern erzeugt**. Dabei werden (Zeilen-)Einträge durch Komma oder Leerzeichen getrennt. **Länge/Anzahl der Einträge** erhält man mit `length` oder `numel`

```
>> x=[4,5,6], x=[4 5 6]
x =
     4     5     6
x =
     4     5     6
>> length(x), numel(x)
ans =
     3
ans =
     3
```

- **Auf Einträge eines Vektors mit runden Klammern zugreifen**. In Matlab hat **erster Eintrag den Index 1 und nicht 0**! Letzter Eintrag geht auch mit `end`.

```
>> a=x(1), b=x(end), c=x(0)
a =
     4
b =
     6
Subscript indices must either be real positive integers or
logicals.
```

- **Positiv-Ganzzahlige Vektoren** können **auch als Indices** verwendet werden. So kann man mehrere Einträge zuweisen oder abrufen.

```
>> a=x([3,1,2,2])
a =
     6     4     5     5
>> a(4)=1
a =
     6     4     5     1
>> a([4,2])=1
a =
     6     1     5     1
>> a([4,2])=[1,0]
a =
     6     0     5     1
>> a([4,2])=[1,0,2]
In an assignment  A(:) = B, the number of elements in A and B must
be the same.
```

- **Gleichgroße Vektoren komponentenweise addieren, mit Skalar addieren oder multiplizieren**

```
>> 2*x, x-1, x+[1,2,3], x+[1,2]
ans =
     8    10    12
ans =
     3     4     5
ans =
     5     7     9
Matrix dimensions must agree.
```

- **Gleichgroße Vektoren elementweise multiplizieren, dividieren, potenzieren** mit `.*` und `./` und `.^` (oder auch mit Skalar), **Elementare Funktionen wirken automatisch elementweise**

```
>> x.*x, x.^2, 2.^x, x.^x, 1./x, x./x, exp(x)
ans =
    16    25    36
ans =
    16    25    36
ans =
    16    32    64
ans =
           256          3125         46656
ans =
    0.2500    0.2000    0.1667
ans =
     1     1     1
ans =
   54.5982  148.4132  403.4288
```

- Mehrere **Vektoren lassen sich zusammenfassen**

```
>> x=[[1 2], zeros(1,4), 2*ones(1,3)]
x =
     1     2     0     0     0     0     2     2     2
```

- **Doppelpunkt-Operator** `a:b` erzeugt Zeilenvektor mit Einträgen `a, a+1, a+2,...,b`. Entsprechend `a:d:b` für Abstand `d` statt `1`.

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
>> x(end:-2:1)
ans =
    10     8     6     4     2
```

- **Command Window löschen** mit `clc` und **Ausgabe unterdrücken mit Semikolon** am Ende.

```
>> x=1:1e4
>> clc
>> x=1:1e4;
```

1.4 Einfache Plots

- Mit `plot(x,y)` werden die Punkte mit x-Koordinaten im Array `x` und y-Koordinaten im Array `y` geplottet (und standardmäßig mit geraden Linien verbunden).

```
>> x=-5:5; y=x.^2;
>> plot(x,y) % Punkte mit geraden Linien verbunden
>> plot(x,y,'o') % Punkte als Kreise, nicht verbunden
```

- Mit `hold('on')...hold('off')` kann **in die selbe Figure dazu geplottet** werden.

```
>> hold('on')
>> plot(y,x,'ro-') % rote Kreise mit Linien verbunden
>> plot([-5,25],[-5,25],'g','LineWidth',3) % dickere grüne Linie
>> hold('off')
```

- Mit `figure(n)`, `close(n)` kann die **neue Figure** `n` **geöffnet bzw. geschlossen** werden.

```
>> figure(2)
>> plot(y,x,'ro-','MarkerSize',10)
>> close(2)
```

- Weiteres zu `plot` und **Erklärung zu allen Befehlen im Hilfemenü**.

2 Editor, Skripte, Funktionen

- **Im Editor** können **sowohl Skripte als auch Funktionen** eingegeben und gespeichert werden.

2.1 Skripte

- Mehrere Befehle als **Skript im Editor** eingeben:
  - **Editor öffnen**: Durch **"New Script"-Icon** anklicken, oder im Command Window `edit` eingeben.

    ```
    n=5; % n=100
    x=linspace(0,2*pi,n); % Vektor mit n äquidistanten Einträgen von 0 bis 2*pi
    plot(x,sin(x),x,cos(x)); % Zwei Plots in selbe Figure
    legend('sinus','cosinus'); % Legende
    grid('on') % Gitterlinien
    %% Neue Section: mit Icon 'Run Section' einzeln ausführen
    grid('off')
    ```

    (Nach einem **Prozentzeichen** stehen **Kommentare**.)
    (**Nach genau 2 Prozentzeichen folgt eine neue Section**.)
  - **Skript speichern**, hier z.B. als `test_plot.m`
  - **Skript ausführen**: Durch **"Run"-Icon** anklicken,
    oder im Command Window **Name wie Befehl eingeben** (ohne File-Endung ".m"):

    ```
    >> test_plot
    ```

  - **Einzelne Section ausführen**: Durch Mausklick gewünschte Section und dann **"Run Section"-Icon** anklicken

2.2 Funktionen

- Eine **Funktion** wird im Editor allgemein erzeugt durch

  ```
  function [out1, out2, ...] = myfun(in1, in2, ...)
  ...
  end
  ```

  Dabei sind
  - `function`: **Schlüsselwort**, an dem Matlab erkennt, dass es sich um ein Funktions- und kein Skript-File handelt
  - `myfun`: **Funktionsname**, welcher mit dem File-Namen übereinstimmen sollte
  - `[out1, out2, ...]` Liste der **Ausgabevariablen** in **eckigen Klammern** (können bei nur einer oder keiner Variablen weggelassen werden), kann beliebig lang und auch leer sein.
    Bemerkung: Hier wird durch die eckigen Klammern kein Array erzeugt.
  - `(in1, in2, ...)` Liste der **Eingabevariablen** in **runden Klammern** (können nur bei keiner Variablen weggelassen werden), kann beliebig lang und auch leer sein.
  - Danach können beliebige Matlab-Befehle folgen.
  - **Vorzeitig verlassen** kann man eine Funktion mit `return` oder einer **Fehlermeldung** `error`.
  - Das `end` am **Funktionsende** ist optional, empfiehlt sich aber immer (und muss im Zusammenhang mit eingenesteten Funktionen auch verwendet werden).

- **Beispiel**: Die Funktion `mittel` zum Berechnen des **arithmetischen Mittels zweier Zahlen**

```
function a=mittel(x,y)
% arithmetisches Mittel
a=(x+y)/2;
end
```

- **Speichern** als `mittel.m`

- Kommentare können an beliebigen Stellen innerhalb der Funktion benutzt werden. Dabei erscheinen **alle Kommentarzeilen bis zur ersten echten Befehlszeile als Hilfe** mit `help` oder `doc`.

```
>> help('mittel')
  arithmetisches Mittel
```

- **Aufruf der Funktion erfolgt entsprechend dem Funktionskopf**

```
>> a=mittel(3,4)
a =
    3.5000
```

Dadurch werden der Funktion `mittel` an `x` und `y` die Werte `3` bzw. `4` übergeben und das Ergebnis der Variablen `a` zugewiesen. Hier wird bei Aufruf im Command Window das Ergebnis im Workspace gespeichert.

- **Funktionen haben einen eigenen Arbeitsspeicher**. In der Funktion definierte Variablen sind **lokal** und werden beim Verlassen der Funktion gelöscht. Eingabevariablen werden per Adresse übergeben und zusätzlicher Speicher nur bei Veränderung in Anspruch genommen. **Beim Aufruf der Funktion müssen die Variablen nicht den selben Namen wie im Funktionskopf haben**.

```
>> clear('a') % löscht Variable a im Workspace
>> b=mittel(3,4)
b =
    3.5000
% a wird nur lokal berechnet und nicht im Worspace abgelegt
```

- Die **Variablen können von beliebigem Typ sein**, sofern alle in der Funktion vorkommenden Ausdrücke ausgewertet werden können, hier also z.B. auch Vektoren gleicher Länge für `x` und `y`.

```
>> b=mittel([1 2],[3 4])
b =
     2     3
```

- Nun erweitern wir die Funktion `mittel` so, dass **zusätzlich noch das geometrische und harmonische Mittel** berechnet werden

```
function [a,g,h]=mittel2(x,y)
% arithmetisches, geometrisches, harmonisches Mittel
a=(x+y)/2;
g=sqrt(x.*y);
h=1./(1./x+1./y);
end
```

und weisen die Ergebnisse zu

```
>> [a,g,h]=mittel2([1 2],[3 4])
a =
     2     3
g =
    1.7321    2.8284
h =
    0.7500    1.3333
```

Wie man sieht, wird hier durch die eckigen Klammern um die Ausgabevariablen beim Funktionsaufruf kein gemeinsames Array erzeugt, sondern allen Ausgabevariablen einzeln das entsprechende Ergebnis zugewiesen (hier jeweils ein Vektor der Länge 2).

3 for-Schleife

- Eine **for-Schleife** dient zum **wiederholten Ausführen von Anweisungen**:

  ```
  for var = a:b                                  % oder auch var = a:c:b
     Anweisungen
  end
  ```

  Beschreibung:

  - Die `Anweisungen` werden in einer Schleife ausgeführt, wobei die Variable `var` **nacheinander** die Werte `a:b` bzw. `a:c:b` durchläuft.
  - Eine `for`-Schleife kann **vorzeitig** mit `break` **beendet** werden. Ein `continue` bewirkt, dass die folgenden Anweisungen ignoriert werden und führt die `for`-Schleife mit der nächsten Iteration fort.

- **Beispiele**:

  - dreimalige **Bildschirmausgabe**

    ```
    >> for k=1:3 disp('Hallo'); end
    Hallo
    Hallo
    Hallo
    ```

  - Die **Anweisungen können von der Schleifen-Variablen abhängen**

    ```
    >> for k=1:3 disp(k); end
         1
         2
         3
    ```

  - Eine Funktion zur Berechnung der **Summe der ersten n natürlichen Zahlen**

    $$s_n = \sum_{k=1}^{n} k = 1 + 2 + \ldots + n$$

    ```
    function s=summe(n)
    s=0;
    for k=1:n
       s=s+k;
    end
    end
    ```

    Testen:

    ```
    >> n=5; s=summe(n), n*(n+1)/2
    s =
        15
    ans =
        15
    ```

– **Skalarprodukt zweier Vektoren** $x, y \in \mathbb{R}^n$

$$\langle x, y\rangle = \sum_{k=1}^{n} x_k \cdot y_k = x_1 \cdot y_1 + \ldots + x_n \cdot y_n$$

```
function s=skalarprodukt(x,y)
n=numel(x);
s=0;
for k=1:n
    s=s+x(k)*y(k);
end
end
```

Testen:

```
>> x=[1 2 3]; y=[3 0 -1]; s=skalarprodukt(x,y)
s =
     0
```

– **Elementweises Produkt zweier Vektoren**: Für $x, y \in \mathbb{R}^n$ sei $p \in \mathbb{R}^n$ mit

$$p_k = x_k \cdot y_k \quad , \quad k = 1, \ldots, n$$

```
function p=produkt_elementweise(x,y)
n=numel(x);
p=zeros(1,n); % optional; reserviert Speicher, dadurch schneller
for k=1:n
    p(k)=x(k)*y(k);
end
end
```

Testen:

```
>> x=[1 2 3]; y=[3 0 -1]; p=produkt_elementweise(x,y)
p =
     3     0    -3
```

Die Anweisung `p=zeros(1,n)` ist optional. Dadurch wird ein Vektor aus lauter 0'en erzeugt, der die gleiche Länge wie `x` hat. Diese **Reservierung von Speicherplatz** vor Beginn der `for`-Schleife empfiehlt Matlab für eine **schnellere Performance**, da ansonsten in jedem Schleifendurchlauf die Größe des Arrays geändert wird. In der Tat macht sich dies **bei größeren Arrays bemerkbar**, wie wir mit einer **Zeitmessung** mit `tic,toc` überprüfen können.

```
>> x=ones(1,1e6); y=3*x;   % ones: erzeugt Array mit lauter 1'en
>> tic; p=produkt_elementweise(x,y); toc   % ohne Speicherreservierung
Elapsed time is 0.119069 seconds.
>> tic; p=produkt_elementweise(x,y); toc   % mit Speicherreservierung
Elapsed time is 0.022213 seconds.
```

Am schnellsten ist aber die eingebaute Matlab-Operation `.*`

```
>> tic; p=x.*y; toc
Elapsed time is 0.003829 seconds.
```

4 Logische Ausdrücke und Variablen

- Matlab weist **logischen Ausdrücken** den Wert 1 bzw. `true` für "wahr" und den Wert 0 bzw. `false` für "falsch" zu. **Logische** Ausdrücke können dann auch **Variablen** zugewiesen werden.

```
>> a=3<4, b=true, c=false(1,3)
a =
  logical
   1
b =
  logical
   1
c =
  1x3 logical array
   0   0   0
```

- Mit Hilfe von `whos` sieht man auch, dass Matlab zwischen den logischen und numerischen 1'en und 0'en unterscheidet, allerdings **benötigt Matlab 1 Byte statt wie meist üblich nur 1 Bit** pro logischer Variable! (Dafür können logische und numerische Variablen kombiniert werden, s.u.)

```
>> d=1; whos('a','d')
  Name      Size            Bytes  Class     Attributes

  a         1x1                 1  logical
  d         1x1                 8  double
```

- **Relationen** werden elementweise überprüft. Die Dimensionen der Arrays müssen dabei übereinstimmen. Ausnahme: Vergleich eines Skalars mit jedem Element eines Array.

`a < b` , `a <= b`	`a` kleiner(gleich) `b`
`a > b` , `a >= b`	`b` größer(gleich) `b`
`a == b`	`a` gleich `b`
`a ~= b`	`a` ungleich `b`

```
>> a=[1 2 3]; b=[-1 4 3]; c=2; a<=b, c==a, c~=a
ans =
  1x3 logical array
   0   1   1
ans =
  1x3 logical array
   0   1   0
ans =
  1x3 logical array
   1   0   1
```

- In **logischen Ausdrücken** wird jedem von 0 verschiedenen **numerischen Wert** der Wahrheitsgehalt 1 zugeordnet, nur die numerische 0 wird zur logischen Null. Mit dem Befehl `logical` kann **ausdrücklich ein logisches Array erzeugt** werden.

```
>> logical([1 0 2])
ans =
  1x3 logical array
   1   0   1
```

Umgekehrt können die logischen 1’en und 0’en auch als numerische 1’en und 0’en in Rechnungen verwendet werden, z.B. zum Erzeugen abschnittsweise definierter Funktionen wie

$$y(x) = \begin{cases} x & , \quad x < 0 \\ x^2 & , \quad x \geq 0 \end{cases}$$

```
>> x=linspace(-1,1,100); y=(x<0).*x+(x>=0).*x.^2; plot(x,y)
```

- Logische Ausdrücke `a`, `b` können (elementweise) durch **logische Operatoren** verknüpft werden.

`a & b` , `and(a,b)`	`a` UND `b`
`a | b` , `or(a,b)`	`a` ODER `b`
`~a` , `not(a)`	NICHT `a`
`xor(a,b)`	ENTWEDER `a` ODER `b`

```
>> a=[0 0 1 1]; b=[0 1 0 1]; a&b, a|b, ~a, xor(a,b)
ans =
  1x4 logical array
   0   0   0   1
ans =
  1x4 logical array
   0   1   1   1
ans =
  1x4 logical array
   1   1   0   0
ans =
  1x4 logical array
   0   1   1   0
```

- Mit `all` und `any` kann man überprüfen, ob **alle bzw. irgendein Eintrag eines Arrays wahr** sind.

```
>> all(a), any(a)
ans =
  logical
   0
ans =
  logical
   1
```

- **Short-circuit** `&&` und `||`: In `a && b` und `a || b` wird Ausdruck b nicht mehr ausgewertet, falls der Wahrheitsgehalt der Aussage nach Auswertung von Ausdruck a schon eindeutig ist (Matlab arbeitet Ausdrücke von links nach rechts ab). Somit können z.B. große oder ungewünschte Rechnungen vermieden werden.
Vorsicht: Alle Ausdrücke, die im Zusammenhang mit den short-circuit-Operatoren ausgewertet werden, müssen skalarwertig sein (im Gegensatz zu `&` und `|`).

- Die **Priorität beim Auswerten** der logischen Operatoren steht in folgender Tabelle. Es ist aber dennoch empfehlenswert, immer zu Klammern.

Operator	Priorität
`~`	höchste
`&`	
`|`	
`&&`	
`||`	niedrigste

Dementsprechend ist `a | b & c` gleichbedeutend mit `a | (b & c)`, letzteres liest sich aber besser.

5 if-Abfrage

- Die allgemeine Form einer **if-Abfrage** ist

```
if Ausdruck_1  (wahr)
   Anweisungen_1
elseif Ausdruck_2  (wahr)    % optional
   Anweisungen_2
...
elseif Ausdruck_n  (wahr)    % optional
   Anweisungen_n
else                         % optional
   Anweisungen_n+1
end
```

Beschreibung:

  - Es werden solange die Ausdrücke `Ausdruck_1,...,Ausdruck_n` nacheinander überpüft, bis einer wahr ist. Ist dann `Ausdruck_k` wahr, so werden die `Anweisungen_k` ausgeführt (und nur diese), und alle folgenden Ausdrücke werden nicht mehr überprüft. Ist kein Ausdruck wahr, so werden die `Anweisungen_n+1` nach `else` ausgeführt.
  - `elseif` und `else` sind optional. Es kann mehrere `elseif`-Abfragen geben, aber nur ein `else`. **Matlab empfiehlt im if/elseif Ausdruck** die Verwendung von **short-circuit `&&` und `||`**.

- **Beispiele:**

  - **Vorzeichenfunktion** $\mathrm{sign}(x) = \begin{cases} -1 & ,x < 0 \\ 0 & ,x = 0 \\ 1 & ,x > 0 \end{cases}$

```
function v=vorzeichen(x)
if x<0
    v=-1;
elseif x>0
    v=1;
else
    v=0;
end
end

>> v=vorzeichen(3)
v =
     1
>> v=vorzeichen(-3)
v =
    -1
>> v=vorzeichen(0)
v =
     0
```

Wir **erweitern** die Funktion `vorzeichen` noch so, dass sie **elementweise für ein Array** gilt

```
function v=vorzeichen_elementweise(x)
n=numel(x);
v=zeros(1,n);
for k=1:n
```

```
    if x(k)<0
        v(k)=-1;
    elseif x(k)>0
        v(k)=1;
    else
        v(k)=0;
    end
end
end
```

```
>> x=[-2 0 3]; v=vorzeichen_elementweise(x)
v =
    -1     0     1
```

Es ginge **in Matlab auch sehr kurz und schnell mit einer Kombination aus elementweisen logischen und numerischen Operationen**.

```
>> v=(x>0)-(x<0)
v =
    -1     0     1
```

Die **entsprechende Matlab-Funktion** ist übrigens `sign`.

```
>> v=sign(x)
v =
    -1     0     1
```

– **Finde für ein Array `x` den Index `ind` eines Elementes**, welches den Wert `a` hat. Falls kein Element den Wert `a` hat, soll `ind=0` zurückgegeben werden.

```
function ind=finde(a,x)
n=numel(x);
ind=0;
for k=1:n
    if x(k)==a
        ind=k;
        break % da Eintrag gefunden, kann for-Schleife beendet werden
    end
end
end
```

Zum Testen erzeugen wir einen **Vektor mit 8 ganzzahligen Zufallszahlen** zwischen 0 und 5

```
>> x=randi([0,5],1,8)
x =
     4     5     0     5     3     0     1     3
>> ind=finde(6,x)
ind =
     0
>> ind=finde(0,x)    % hier ist break auskommentiert
ind =
     6
>> ind=finde(0,x)    % hier mit break
ind =
     3
```

6 while-Schleife

- Eine **while-Schleife** dient zum **bedingten wiederholten Ausführen von Anweisungen**:

```
while Ausdruck (wahr)
   Anweisungen
end
```

Beschreibung:

  - Solange der `Ausdruck` wahr ist, werden die `Anweisungen` ausgeführt.
  - **Matlab empfiehlt im while-Ausdruck** die Verwendung von **short-circuit `&&` und `||`**.
  - Eine while-Schleife kann **vorzeitig** mit `break` **beendet** werden. Ein `continue` bewirkt, dass die folgenden Anweisungen ignoriert werden und führt die while-Schleife mit der nächsten Iteration fort.

- **Beispiele:**

  - Im folgenden Skript `test_while.m` wird der Anwender so lange aufgefordert, eine Zahl zwischen 5 und 10 einzugeben, bis er dies auch tut. Die **Tastatur-Eingabe einer Zahl** erfolgt mit dem Befehl `input`.

```
weitermachen=true;
while weitermachen
   x=input('Geben␣Sie␣bitte␣eine␣Zahl␣zwischen␣5␣und␣10␣ein:');
   if (x>5) && (x<10)
        weitermachen=false;
        disp('Danke!')
   end
end

>> test_while
Geben Sie bitte eine Zahl zwischen 5 und 10 ein:4
Geben Sie bitte eine Zahl zwischen 5 und 10 ein:11
Geben Sie bitte eine Zahl zwischen 5 und 10 ein:6
Danke!
```

Vorsicht: Manchmal erzeugt man aus Versehen eine **Endlosschleife**, bei der das Abbruchkriterium nie erfüllt wird. Ein **laufendes Programm kann mit der Tastenkombination STRG+C abgebrochen** werden. Dies kann man hier testen, indem man im Skript `test_while.m` die Zeile `weitermachen=false;` auskommentiert.

  - **Jede for-Schleife lässt sich auch als while Schleife schreiben**.

```
>> for k=1:3 disp(k); end
>> k=1; while k<=3 disp(k); k=k+1; end
     1
     2
     3
```

Vorsicht: Eine **häufige Fehlerquelle** sind Variablen, die man nach dem Durchlaufen einer Schleife weiterverwendet. Hier hat `k` nach dem Schleifen-Durchlauf den Wert `k=4`.

```
>> k
k =
     4
```

Möchte man eigentlich mit dem Wert `k=3` weiterrechnen, so könnte man dies z.B. auch so erreichen

```
>> k=0; while k<3 k=k+1; disp(k); end
     1
     2
     3
>> k
k =
     3
```

– **Babylonisches Wurzelziehen**: Die **iterativ erzeugte Folge** $x_1, x_2, x_3, \dots$ mit

$$x_1 \quad \text{ist beliebiger positiver Startwert}$$
$$x_k := \frac{1}{2}\left(x_{k-1} + \frac{a}{x_{k-1}}\right) \quad , \quad \text{für } k \geq 2$$

konvergiert für $k \to \infty$ gegen $\sqrt{a}$, d.h. die Zahlen x_k nähern für größer werdende k die Zahl $\sqrt{a}$ immer besser. Dies **veranschaulichen** wir uns zuerst mal mit einer Funktion, die **zunächst mit einer for-Schleife** die ersten n Folgenglieder $x_1, \dots, x_n$ berechnet und zusammen als Array ausgibt.

```
function x=babylon_test(a,x1,n)
x=zeros(1,n);
x(1)=x1;
for k=2:n
   x(k)=(x(k-1)+a/x(k-1))/2;
end
end
```

Zur Näherung von $\sqrt{2}$ verwenden wir $a = 2$, Startwert $x1 = 100$ und **berechnen und plotten die ersten** $n = 10$ **Folgenglieder**:

```
>> a=2; x1=100; n=10; x=babylon_test(a,x1,n);
>> plot(1:n,x,'o',1:n,sqrt(2)*ones(1,n))
```

Der Fehler $|x_n - \sqrt{2}|$ (**Betrag des Abstandes** von x_n zu $\sqrt{2}$) ist nach $n = 10$ Iterationen hier

```
>> format long
>> sqrt(a), x(n), abs(x(n)-sqrt(a))    % abs ist Betragsfunktion
ans =
   1.414213562373095
ans =
   1.414215014050053
ans =
     1.451676957975323e-06
```

Für den **besseren Startwert** $x1 = 1$ erhalten wir **nach gleicher Anzahl an Iterationen** auch eine **bessere Näherung**

```
>> a=2; x1=1; n=10; x=babylon_test(a,x1,n);
>> plot(1:n,x,'o',1:n,sqrt(2)*ones(1,n))
>> sqrt(a), x(n), abs(x(n)-sqrt(a))
ans =
   1.414213562373095
ans =
   1.414213562373095
ans =
     2.220446049250313e-16
```

In der Tat ist die **Näherung schon nach wenigen Iterationen sehr gut**

```
>> sqrt(a), x(5), abs(x(5)-sqrt(a))
ans =
   1.414213562373095
ans =
   1.414213562374690
ans =
      1.594724352571575e-12
```

Deshalb wollen wir **nun mit einer while-Schleife statt einer festen Anzahl an Iterationen eine gewünschte Genauigkeit $\epsilon > 0$ vorgeben**. Da wir $\sqrt{a}$ mit dem Algorithmus berechnen wollen, können wir nicht direkt überprüfen, ob schon $|x_n - \sqrt{a}| \leq \epsilon$ gilt. Aber z.B. können wir überprüfen, ob $|x_n^2 - a| \leq \epsilon$ gilt. Für $a \geq 1$ gilt dann nämlich $\sqrt{a} \geq 1$, und (da hier alle $x_n > 0$ bleiben) damit auch $(x_n + \sqrt{a}) \geq 1$ und folglich

$$|x_n - \sqrt{a}| = \frac{|(x_n - \sqrt{a}) \cdot (x_n + \sqrt{a})|}{x_n + \sqrt{a}} = \frac{|x_n^2 - a|}{x_n + \sqrt{a}} \leq |x_n^2 - a| \leq \epsilon$$

In der while-Schleife wollen wir solange iterieren, wie die **gewünschte Genauigkeit noch nicht erreicht** ist. Da hier nur die letzte Iterierte von Interesse ist, verwenden wir auch kein Array (bessere Performance: weniger Speicherplatz und höhere Geschwindigkeit). Ausserdem geben wir zusätzlich noch die benötigte Anzahl an Iterationen aus.

```
function [x,iter]=babylon(a,x1,epsilon)
x=x1;
iter=0;
while abs(x^2-a) > epsilon
    x=(x+a/x)/2;
    iter=iter+1;
end
end
```

Test für $a = 2$ bzw. $a = 10$ für gleichen Startwert $x1 = 1$ und Genauigkeit $\epsilon = 10^{-6}$:

```
>> a=2; [x,iter]=babylon(a,1,1e-6), abs(x-sqrt(a))
x =
   1.414213562374690
iter =
      4
ans =
      1.594724352571575e-12
>> a=10; [x,iter]=babylon(a,1,1e-6), abs(x-sqrt(a))
x =
   3.162277665175675
iter =
      5
ans =
      5.007295467152062e-09
```

7 Mehr zu Funktionen

- **Beim Funktionsaufruf müssen weder alle Eingabevariablen übergeben noch alle Ausgabevariablen verlangt werden**, sondern es kann von hinten weggelassen werden, z.B. bei unserer vorher schon geschriebenen Funktion `[a,g,h]=mittel2(x,y)`

```
>> a=mittel2(1,3)
a =
     2
>> mittel2(1,3)
ans =
     2  % ans erhält den Wert der ersten Ausgabevariablen
```

- Bei den **Ausgabevariablen können auch am Anfang oder mittendrin welche nicht verlangt werden**; dies geschieht mit Hilfe der **Tilde ~ an den entsprechenden Stellen**.

```
>> [a,~,h]=mittel2(1,3)
a =
     2
h =
    0.7500
```

- Beim **Weglassen von Eingabevariablen** muss darauf geachtet werden, dass diesen dennoch Werte zugewiesen werden, wenn Sie z.B. in Rechnungen benutzt werden. Dabei ist oft der Befehl `nargin` nützlich, der die **Anzahl der übergebenen Eingabevariablen** liefert. Mit `nargout` kann die **Anzahl der verlangten Ausgabevariablen** abgefragt werden. **Matlab zählt die Tilde ~ mit!**

```
function [a,g,h]=mittel2narginout(x,y)
% arithmetisches, geometrisches, harmonisches Mittel
if nargin==0
    x=1;
    y=3;
end
a=(x+y)/2;
g=sqrt(x.*y);
if nargout>2
    h=1./(1./x+1./y);
else
    disp('h wurde nicht verlangt')
end
end

>> mittel2narginout
h wurde nicht verlangt
ans =
     2
>> [a,~,h]=mittel2narginout
a =
     2
h =
    0.7500
```

7.1 Unterfunktionen

- **Unterfunktionen (Subfunctions oder Local Functions):**
  Ein Funktionsfile kann **nach der Hauptfunktion noch Unterfunktionen** enthalten. Unterfunktionen werden einfach wie eigene Funktionen **in beliebiger Reihenfolge hinter die Hauptfunktion geschrieben, alles in ein gemeinsames .m-File**.

- **Beispiel:** Somit lässt sich die Funktion `mittel` auch so schreiben:

```
function [a,g]=mittel_local(x,y)
a=arithmetisch(x,y);
g=geometrisch(x,y);
end
function a=arithmetisch(x,y)
a=(x+y)/2;
end
function g=geometrisch(x,y)
g=sqrt(x.*y);
end
```

  Test:

```
>> [a,g]=mittel_local(2,8)
a =
     5
g =
     4
```

- **Auf Unterfunktion** kann man allerdings **von außen nicht zugreifen**, sondern nur von der Hauptfunktion (einschliesslich evtl. eingenesteten Funktionen) und allen Unterfunktionen des selben Funktionsfiles.

```
>> g=geometrisch(2,8)
Undefined function or variable 'geometrisch'.
```

- Jede Unterfunktion hat ihren **eigenen Arbeitsspeicher**, unabhängig von der Hauptfunktion und den anderen Unterfunktionen (im **Unterschied zu eingenesteten Funktionen**). Unterfunktionen sind bzgl. der Variablen unabhängiger als eingenestete Funktionen und können deshalb auch **ohne Gefahr Variablen gleichen Namens wie z.B. die Hauptfunktion** verwenden, ohne diese (unabsichtlich) in der Hauptfunktion zu verändern; allerdings müssen benötigte Variablen auch übergeben werden.

7.2 Eingenestete Funktionen

- **Eingenestete Funktionen (Nested Functions): Innerhalb einer Funktion** können auch eingenestete Funktionen definiert werden:

```
function out_A = A(in_A)
...
   function out_B = B(in_B)
   ...
   end
...
end
```

- **Jede Funktion muss dabei mit end beendet werden.** Es können **beliebig viele eingenestete Funktionen** definiert werden, auch selbst wieder innerhalb von eingenesteten Funktionen, **allerdings nicht innerhalb von Kontrollstrukturen** wie if-Abfragen oder for-und while-Schleifen.

- **Aufrufen** kann man eingenestete Funktionen nicht von ausserhalb der Hauptfunktion, aber
  - **vom direkt höheren Level** (s.unten: A kann B und D aufrufen, aber weder C noch E)
  - **von einer Funktion gleichen Levels innerhalb der selben Eltern-Funktion** (B kann D aufrufen und umgekehrt)
  - **von einer Funktion beliebig niedrigeren Levels** (C kann A, B und D aufrufen, aber nicht E)

```
function A                  % Hauptfunktion
B; D;
   function B               % eingenestet in A
   A; C; D;
      function C            % eingenestet in B
      A; B; D
      end
   end
   function D               % eingenestet in A
   A; B; E;
      function E            % eingenestet in D
      A; B; D;
                    end
   end
end
```

- **Reichweite der Variablen:** Auch eingenestete Funktionen haben ihren **eigenen Arbeitsspeicher**. **Zusätzlich** gilt: **Eingenestete Funktionen haben Zugriff (lesen und verändern) auf alle Variablen der Funktionen beliebig höheren Levels, in der sie eingenested sind**. Umgekehrt kann eine (eingenestete) Funktion **auch auf Variablen** zugreifen, die in einer **in ihr enthaltenen eingenesteten Funktion beliebig niedrigeren Levels deklariert** werden. Ausgabevariablen einer eingenesteten Funktion sind allerdings zunächst lokal und müssen erst explizit in der enthaltenden Funktion zugewiesen werden, um Zugriff darauf zu erhalten.

- **Im Editor** werden Variablen, die von mehreren Funktionen genutzt werden, **farblich kenntlich gemacht**, und beim **Drüberfahren mit dem Mauszeiger** erscheint die "Warnung": **The scope of variable spans multiple functions.**

- **Beispiel:** Somit lässt sich die Funktion `mittel` auch so schreiben:

```
function [a,g]=mittel_nested(x,y)
a=arithmetisch;
g=geometrisch;
    function a=arithmetisch
        a=(x+y)/2;
    end
    function g=geometrisch
        g=sqrt(x.*y);
    end
end
```

oder auch so:

```
function [a,g]=mittel_nested2(x,y)
arithmetisch;
geometrisch;
    function arithmetisch
        a=(x+y)/2;
    end
    function geometrisch
        g=sqrt(x.*y);
    end
end
```

7.3 Rekursive Funktionen

- **Funktionen können sich auch selbst aufrufen**. Dabei muss sichergestellt sein, dass keine unendlichen Rekursionen auftreten.
  Beispiele:

- Berechnung der **Fakultät**

$$n! = \prod_{k=1}^{n} k = n \cdot (n-1)!$$

```
function f=fakultaet(n)
if n==1
    f=1;
    return; % Funktion wird vorzeitig verlassen
end
f=n*fakultaet(n-1);
end
```

Die Zuweisung `f=n*fakultaet(n-1)` funktioniert, da wegen den Erklärungen am Anfang von Abschnitt 7 der Rückgabewert von `fakultaet` ohne vorherige Zuweisung verwendet werden kann (ist hier also abkürzend für `f=fakultaet(n-1); f=n*f`)

Test: (Die **entsprechende Matlab-Funktion** ist übrigens **factorial**).

```
>> fakultaet(5), factorial(5)
ans =
   120
ans =
   120
```

- **Bemerkung:** Ein **Rekursionsaufruf kann oft durch eine geeignete Schleife ersetzt werden** (dies kann effizienter sein). Für die Fakultät ginge also auch

```
function f=fakultaet2(n)
f=1;
if n==1
    return;
end
for k=2:n
   f=k*f;
end
end
```

- **Array von Zahlen aufsteigend mit Selection Sort sortieren**:
  - Hat das Array **nur 1 Element**, so ist **nix zu tun**.
  - **Sonst finde Position des minimalen Eintrags**.
  - Ist minimaler Eintrag **nicht an erster Stelle, so tausche** diesen **mit dem ersten Eintrag**.
  - **Wiederhole** das gleiche nun **mit dem restlichen Array**.

  Beispiel:

$$\underline{4}, 2, \overset{\text{min}}{\underline{1}}, 3 \quad \rightarrow \quad 1| \overset{\text{min}}{\underline{2}}, 4, 3 \quad \rightarrow \quad 1, 2|\underline{4}, \overset{\text{min}}{\underline{3}} \quad \rightarrow \quad 1, 2, 3|4$$

```
function x=selection_sort(x)
n=numel(x);
if n==1
    return
end
ind=finde_min(x,n);
if ind>1
    x=tausche(x,1,ind);
end
x(2:n)=selection_sort(x(2:n));
end
% Unterfunktionen
function ind=finde_min(x,n)
ind=1;
for k=2:n
    if x(k)<x(ind)
        ind=k;
    end
end
end
function x=tausche(x,ind1,ind2)
a=x(ind1);
x(ind1)=x(ind2);
x(ind2)=a;
end
```

  Zum **Testen und Fehlersuchen** sind auch der **Debugger und Profiler** nützlich, siehe dazu die **nächsten beiden Unterabschnitte**.

```
>> x=randi([1,5],1,9)
x =
     1     5     5     3     3     1     5     1     1
>> x=selection_sort(x)
x =
     1     1     1     1     3     3     5     5     5
```

- Array von Zahlen aufsteigend mit **Quick Sort** sortieren:
  - Hat das Array **nur 1 Element**, so ist **nix zu tun**.
  - **Sonst teile Array in zwei Teile**, die selbst auch **nochmal mit Quick Sort sortiert** werden-
  - Das **Teilen** geht so: Wähle ein **Pivotelement p**, z.B. das **erste Element des Arrays**. Setze dann alle **Einträge kleinergleich p links von p**, und alle **Einträge größergleich p rechts von p** (durch **geeignetes Tauschen mit p**).

Beispiel: zunächst **Grundidee**

teilen mittels Pivotelement	$\overline{3},4,5,2,1,3,6,1 \quad \rightarrow \quad 2,1,1,\overline{3},4,5,3,6$
nochmal mit linkem Teil-Array	$\overline{2},1,1 \quad \rightarrow \quad 1,1,\overline{2}$
nochmal mit rechtem Teil-Array	$\overline{4},5,3,6 \quad \rightarrow \quad 3,\overline{4},5,6$

nun **genauer für Unterfunktion teile**

$$\xrightarrow{\text{suchen}} \overline{3},\overset{il}{\underline{4}},5,2,1,3,6,\overset{ir}{\underline{1}} \xrightarrow{\text{tauschen}} \overline{3},\underline{1},5,2,1,3,6,\underline{4} \xrightarrow{\text{weitersuchen}} \overline{3},1,\overset{il}{\underline{5}},2,\overset{ir}{\underline{1}},3,6,4$$

$$\xrightarrow{\text{tauschen}} \overline{3},1,\underline{1},2,\underline{5},3,6,4 \xrightarrow{\text{weitersuchen}} \overline{3},1,1,\overset{il=ir}{\underline{2}},5,3,6,4 \xrightarrow{\text{mit p tauschen}} \underline{2},1,1,\overline{3},5,3,6,4$$

(Bemerkung: Je nach Implementierung können die Einträge der Teilarrays in unterschiedlicher Reihenfolge auftauchen, hier erhalten wir z.B. auch eine andere Reihenfolge als oben in der ersten Zeile zur Grundidee.)

```
function x=quick_sort(x,anfang,ende)
if anfang<ende
    % finde endgültige Position des Pivotelementes
    [x,ind]=teile(x,anfang,ende);
    % sortiere Einträge links davon
    x=quick_sort(x,anfang,ind-1);
    % und rechts davon
    x=quick_sort(x,ind+1,ende);
end
end
% Unterfunktionen
function [x,ind]=teile(x,anfang,ende)
p=x(anfang); % Pivotelement
il=anfang+1;
ir=ende;
while il<ir
    % suche von rechts bis Eintrag<p gefunden
    while ir>anfang && x(ir)>=p
        ir=ir-1;
    end
    % suche von links bis Eintrag>p gefunden
    while il<ir && x(il)<=p
        il=il+1;
    end
    % tausche die beiden, falls nicht schon an selber Position
    if il<ir
        x=tausche(x,il,ir);
    end
end
% stelle sicher, dass p zwischen beiden Teilen liegt
if x(ir)<p
    x=tausche(x,ir,anfang);
end
% und gib dann den (neuen) Index von p zurück
ind=ir;
end
function x=tausche(x,ind1,ind2)
a=x(ind1);
```

```
x(ind1)=x(ind2);
x(ind2)=a;
end
```

Test

```
>> x=randi([1,5],1,9)
x =
     3     4     2     5     5     2     2     5     4
>> x=quick_sort(x,1,numel(x))
x =
     2     2     2     3     4     4     5     5     5
```

Ein Laufzeitvergleich zeigt, dass Quick Sort bei Arrays mit weitgehend zufällig verteilten Einträgen wesentlich schneller ist als Selection Sort. Beide haben zwar im worst case Komplexität $\mathcal{O}(n^2)$ bei einem n-elementigen Array, aber für Quick Sort kann man zeigen, dass die Komplexität im Mittel nur $\mathcal{O}\big(n \cdot \log(n)\big)$ ist.

```
>> x=rand(1,1e4);    % 10000 gleichverteilte Zufallszahlen im Intervall [0,1]
>> tic; xq=quick_sort(x,1,numel(x)); toc
Elapsed time is 0.009112 seconds.
>> tic; xs=selection_sort(x); toc
Elapsed time is 0.737609 seconds.
```

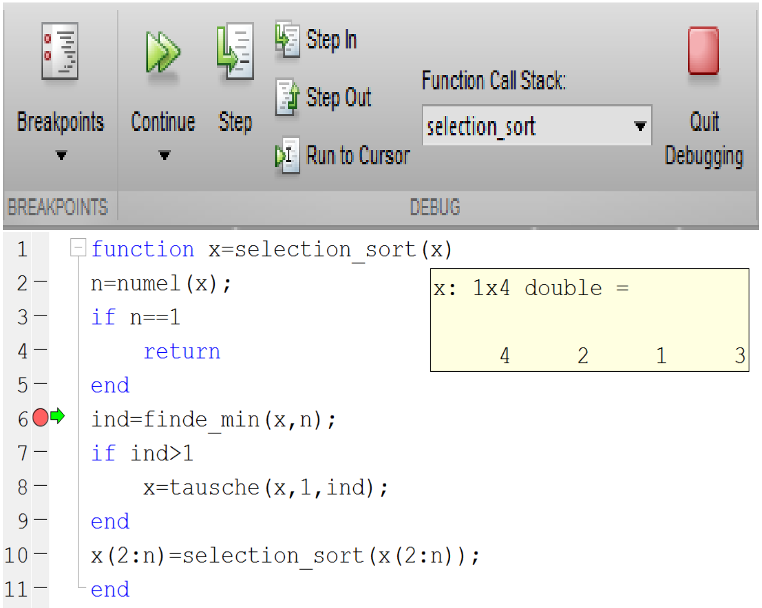
Bei **weitgehend vorsortierten Einträgen** ist auch **kaum ein Unterschied** zu merken.

```
>> x=ones(1,1e4);    % Vektor mit 10000 Einsen
>> tic; xq=quick_sort(x,1,numel(x)); toc
Elapsed time is 0.401207 seconds.
>> tic; xs=selection_sort(x); toc
Elapsed time is 0.731163 seconds.
```

7.4 Debugger

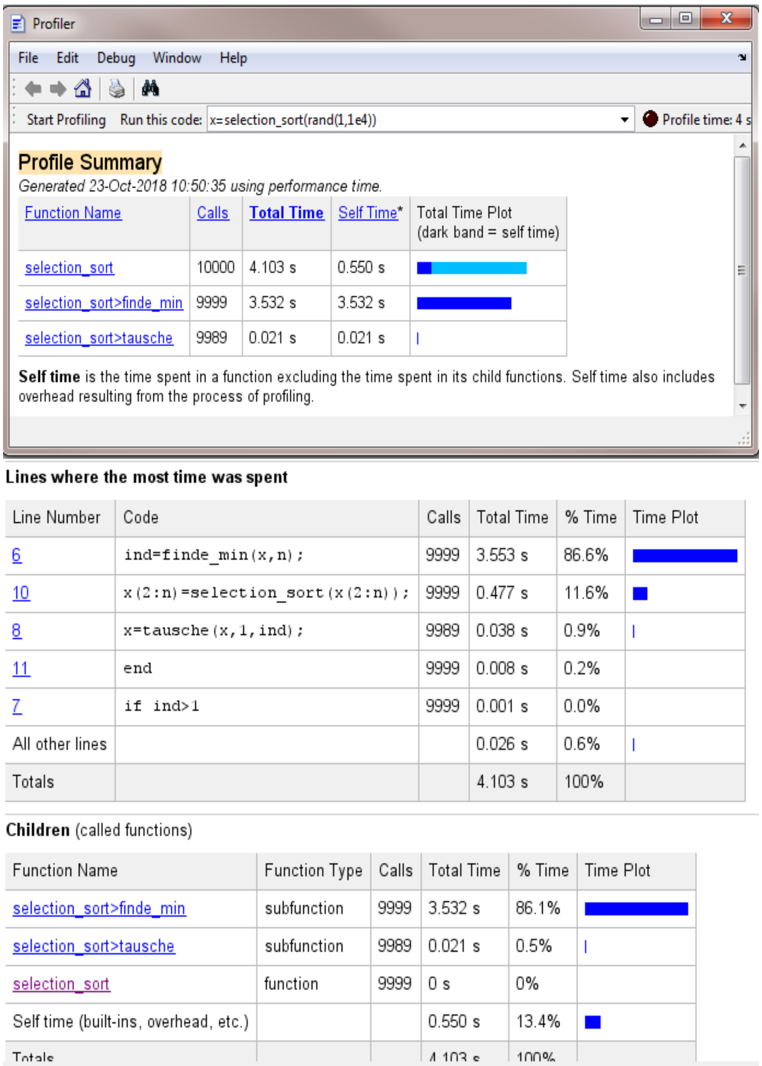
Ein **interaktiver Debugger im Editor** erleichtert die **Fehlersuche**:

- Beim Setzen von **Breakpoints** läuft Programm nur bis zu dieser Zeile (**Setzen/Löschen durch linken Mausklick neben Zeile**).
- **Inhalt von Variablen** ansehen durch **drüberfahren mit Mauszeiger**.
- Mit **Step** wird **nächste Zeile** ausgeführt.
- Mit **Step In** wird **in Funktion (z.B. Unterfunktion) gesprungen**, und mit **Step Out** wieder raus.
- Mit **Continue** gehts weiter **bis zum nächsten Breakpoint**.
- Mit **Run to Cursor** gehts weiter bis zum Cursor.
- **Unter Breakpoints** kann man zum Beispiel **alle Breakpoints löschen**.
- Debugger **verlassen** mit **Quit Debugging** (wichtig bevor man das Programm nochmal ändern möchte).



7.5 Profiler

Genauere Laufzeitangaben auch bzgl. den Unterfunktionen erhält man mit dem Profiler: **Run and Time-Button drücken**, oder >> `profile('viewer')` eingeben, und dann Funktion aufrufen.



Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
selection_sort	10000	4.103 s	0.550 s	
selection_sort>finde_min	9999	3.532 s	3.532 s	
selection_sort>tausche	9989	0.021 s	0.021 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
6	`ind=finde_min(x,n);`	9999	3.553 s	86.6%	
10	`x(2:n)=selection_sort(x(2:n));`	9999	0.477 s	11.6%	
8	`x=tausche(x,1,ind);`	9989	0.038 s	0.9%	
11	`end`	9999	0.008 s	0.2%	
7	`if ind>1`	9999	0.001 s	0.0%	
All other lines			0.026 s	0.6%	
Totals			4.103 s	100%	

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
selection_sort>finde_min	subfunction	9999	3.532 s	86.1%	
selection_sort>tausche	subfunction	9989	0.021 s	0.5%	
selection_sort	function	9999	0 s	0%	
Self time (built-ins, overhead, etc.)			0.550 s	13.4%	
Totals			4.103 s	100%	

8 Function Handle und Anonyme Funktionen

8.1 Function Handle

- Ein **Function Handle** ist ein **"Griff" auf eine Matlab-Funktion**, der **genauso verwendet** werden kann **wie die Funktion selbst**. Ein Function Handle wird mit dem `@`-Zeichen erzeugt.

  ```
  var = @myfun
  ```

  Dabei sind

  – `var`: beliebiger **Name der Variable**, der die Funktion zugewiesen wird

  – `myfun`: **Funktionsname** (**ohne Pfad-Angabe**, dieser wird automatisch vollständig gespeichert),

- **Beispiel:** Der folgende Aufruf erzeugt eine Variable `f` als Function Handle, welche die gleichen Eigenschaften wie die Matlab-Funktion `sin` hat.

  ```
  >> f=@sin
  f =
    function_handle with value:
      @sin
  >> f([0,pi/2])
  ans =
       0     1
  >> x=linspace(0,2*pi,1000); plot(x,f(x))
  ```

- **Function Handle können einer anderen Funktion als Eingabevariable übergeben werden.** Mit folgender Funktion `zeichnen` lassen sich dann einfach Funktionsgraphen über einem Intervall plotten.

  ```
  function zeichnen(f,a,b)
  x=linspace(a,b,1000);
  plot(x,f(x)); grid('on')
  end
  ```

  ```
  >> zeichnen(f,0,2*pi)
  >> zeichnen(@sqrt,0,1) % hier wird direkt ein Function Handle auf sqrt übergeben
  ```

- **Auch Funktionen in m.-Files** können **mit einem Function Handle** versehen werden. Ein Function Handle kann dann **genauso aufgerufen werden wie die Funktion selbst**.

  ```
  function y=myfun(x)
  y=x.^2.*sin(100*x);
  end
  ```

  ```
  >> zeichnen(@myfun,0,1)
  >> h=@zeichnen; h(@atan,-10,10)
  ```

- **Viele Matlab-Funktionen arbeiten mit Function Handles**, z.B.

- **Numerische Integration** $\int_a^b f(x)\,dx$ mit `integral`:

  ```
  >> integral(@sin,0,pi)
  ans =
      2.0000
  ```

- **Finden von Nullstellen** einer Funktion, d.h. finde $x \in \mathbb{R}$ mit $f(x) = 0$, mit `fzero`:
  Kennt man **zwei Funktionswerte** $f(a)$ und $f(b)$ **mit verschiedenem Vorzeichen**, so kann man eine **Nullstelle im Intervall** $[a, b]$ **finden** mit

```
>> x=fzero(@sin,[3,4])
x =
    3.1416
```

  Mehr Information über die Iterationen des Verfahrens erhält man z.B. mit

```
>> options=optimset('Display','iter');
>> x=fzero(@sin,[3,4],options)

 Func-count    x          f(x)             Procedure
    2               3       0.14112        initial
    3         3.15716    -0.0155695        interpolation
    5         3.14159   -1.86917e-09       interpolation
    7         3.14159    1.22465e-16       interpolation

Zero found in the interval [3, 4]
x =
    3.1416
```

  Alternativ kann man auch **nur einen Startwert** verwenden. Dann wird zunächst ein geeignetes Intervall gesucht, auf dem ein Vorzeichenwechsel stattfindet.

```
>> x=fzero(@sin,3,options)

Search for an interval around 3 containing a sign change:
 Func-count    a          f(a)             b          f(b)        Procedure
    1               3       0.14112              3       0.14112   initial inter
    3         2.91515      0.224515        3.08485     0.0567094   search
    7         2.83029      0.306295        3.16971    -0.0281093   search

Search for a zero in the interval [2.83029, 3.16971]:
 Func-count    x          f(x)             Procedure
    7         3.16971    -0.0281093        initial
    9         3.14159   -5.41432e-08       interpolation
   12         3.14159    1.22465e-16       interpolation

Zero found in the interval [2.83029, 3.16971]
x =
    3.1416
```

- **Minimieren einer Funktion** auf Intervall, d.h. finde $x \in [a, b]$ mit $f(x) = \min_{t\in[a,b]} f(t)$, mit `fminbnd`:

```
>> [x,fx]=fminbnd(@sin,3,6,options)

 Func-count     x          f(x)         Procedure
    2          4.8541     -0.989976      golden
    8         4.71242            -1      parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.0e-04
```

```
x =
    4.7124
fx =
   -1.0000
>> 3*pi/2
ans =
    4.7124
```

Vorsicht: Mit `fminbd` **kann nicht garantiert werden, dass tatsächlich ein globales Minimum gefunden** wurde. **Oft werden nur lokale Minima gefunden**, wie wir bei unserer eigenen Funktion `myfun` sehen können:

```
>> zeichnen(@myfun,0,1)
>> [x,fx]=fminbnd(@myfun,0,1)
x =
    0.6758
fx =
   -0.4564
>> [x,fx]=fminbnd(@myfun,0.95,1)
x =
    0.9898
fx =
   -0.9795
```

Bei **so einfachen Funktionen** könnte man natürlich auch **einfach von sehr vielen Funktionswerten das Minimum bestimmen**

```
>> x=linspace(0,1,1e4); [fx,k]=min(myfun(x))
fx =
   -0.9795
k =
        9898
>> x(k)
ans =
    0.9898
```

Ansonsten ist aber `fminbd` **hilfreich, wenn z.B. Funktionsauswertungen teuer** sind (aufwendige Algorithmen, Messungen,...) **oder** eine **lokale Minimumstelle mit sehr hoher Genauigkeit** bestimmt werden soll, denn `fminbd` versucht **mit möglichst wenig Funktionsauswertungen** ein Minimum zu finden.

8.2 Anonyme Funktionen

- Es gehen auch **Function Handle auf Anonyme Funktionen, d.h. einfache Funktionen ohne eigenes .m-File.**

  ```
  var = @(in1, in2,...) Anweisung
  ```

  Dabei sind

  – `var`: **Name der Variable**, der das Function Handle zugewiesen wird (optional).

  – `(in1, in2, ...)`: **Liste der Eingabevariablen** in runden Klammern, kann beliebig lang und auch leer sein (dann leere Klammern `()` benutzen).

  – `Anweisung`: Ein **einfacher Matlabausdruck** (keine Kontrollausdrücke wie `if,while,...` oder mehrere durch Semikolon/Komma getrennte Anweisungen).

- **Beispiel**

```
>> f=@(x) x.^2
f =
  function_handle with value:
    @(x)x.^2
>> zeichnen(f,-1,1)
>> zeichnen(@(x) 1./x,0.1,10);
```

- **Aus einem Function Handle lassen sich auch neue Function Handle erzeugen**, z.B. um eine Funktion an der x-Achse zu spiegeln oder daraus neue Funktionen zu basteln

```
>> f=@(x) -f(x)
f =
  function_handle with value:
    @(x)-f(x)
>> zeichnen(f,-1,1)
>> f=@(x) (f(x-1)+1).^2; zeichnen(f,-1,3)
```

- Somit lassen sich auch gut **Funktionen mit Parametern übergeben**, z.B. wenn wir verschiedene Parabeln plotten wollen. Zunächst erzeugen wir ein Function Handle mit mehreren Eingabevariablen für eine allgemeine Parabel $p(x) = a \cdot x^2 + b \cdot x + c$.

```
>> p=@(x,a,b,c) a*x.^2+b*x+c;
```

  Zum Beispiel erhalten wir speziell für $p(x) = 2x^2 + 1$ den Wert $p(0) = 1$ durch

```
>> p(0,2,0,1)
ans =
     1
```

  Dann erzeugen wir aus p spezielle Parabeln als Function Handle die nur noch von x abhängen, und die Werte der Parameter fest gewählt werden.

```
>> q=@(x) p(x,2,0,1);
>> zeichnen(q,-1,1)
>> zeichnen(@(x) p(x,0,1,0),-1,1)
```

- **Beispiel** Die folgende Funktion `bisektion` nähert eine **Nullstelle einer Funktion mit Hilfe der sukzessiven Intervallhalbierung** des Ausgangsintervalls $[a, b]$ bis die Nullstelle gefunden oder eine gewisse Genauigkeit erreicht ist (hier Verwenden wir sowohl die Breite des Intervalls als auch die Größe des Funktionswertes). Dabei wird zunächst angenommen, dass die Funktion f am linken Intervallende a negativ und am rechten Intervallende b positiv ist. Die Funktionen müssen dabei als Function Handle übergeben werden.

```
function x=bisektion(f,a,b,epsilon)
% Findet Nullstelle einer Funktion mittels Intervallhalbierung
% Annahme: f(a)<0 und f(b)>0
x=(a+b)/2;
fx=f(x);
while (abs(b-a)>epsilon) && (abs(fx)>epsilon)
    if fx<0
        a=x;
    else
        b=x;
    end
    x=(a+b)/2;
```

```
    fx=f(x);
end
end
```

Suchen wir damit eine Nullstelle von $f(x) = \frac{\sin(x)}{x}$ in geeignetem Intervall.

```
>> f=@(x) sin(x)./x;
>> zeichnen(f,0.2,6*pi);
>> x0=bisektion(f,4,8,1e-6)
x0 =
     6.2832
>> f(x0)
ans =
   4.0728e-07
```

9 Zeichenketten

- Eine **Zeichenkette (String)** ist ein **Array vom Typ char** (character), und lässt sich mit dem einfachen Anführungszeichen `'` (über `#`) am Anfang und Ende erzeugen. Das einfache Anführungszeichen selbst wird dabei durch zwei einfache Anführungszeichen `''` (ohne Leerzeichen) erzeugt.

```
>> s1='hallo', s2='2', s3='peter''s␣auto'
s1 =
    'hallo'
s2 =
    '2'
s3 =
    'peter's auto'
```

- **Jedes Zeichen benötigt 2 Byte**=16 bit Speicher (also gibt es theoretisch $2^{16} = 65536$ verschiedene Zeichen).

```
>> whos('s1','s2')
  Name      Size            Bytes  Class    Attributes

  s1        1x5                10  char
  s2        1x1                 2  char
```

- Mit `isa` kann man **den Typ einer Variablen überprüfen**.

```
>> isa(s2,'char'), isa(2,'char'), isa(2,'double')
ans =
  logical
   1
ans =
  logical
   0
ans =
  logical
   1
```

- Genau **wie bei numerischen Arrays** kann man auf **einzelne Zeichen mittels Indizierung** zugreifen und **Strings zu einem größeren Array zusammenfassen**.

```
>> s1(end:-1:1)
ans =
    'ollah'
>> s4=[s1,'welt']
s4 =
    'hallowelt'
```

- **Zeichen 0 bis 127 entsprechen den Standard ASCII-Zeichen**. Mit `char` erhält man beginnend mit 32 (entspricht Leerzeichen) davon die “sichtbaren”,

```
>> char(32:127)
ans =
    '␣!"#...'
```

und z.B. ist **10 die Nummer des Sonderzeichens für Zeilenumbruch**.

```
>> s=['a',char(10),'b']
>> disp(s)
a
b
```

- Umgekehrt erhält man die **Nummer eines Zeichens** mit `double`

```
>> double('äöüÄÖÜß')
ans =
   228   246   252   196   214   220   223
```

- Diese Nummern werden **auch in mathematischen Operationen und Vergleichen** verwendet.

```
>> double('Aab'), char('a'+1), 'aa'<'Ab'
ans =
    65    97    98
ans =
    'b'
ans =
  1x2 logical array
   0   1
```

- **Vergleich zweier Strings** macht man am besten mit `strcmp`, da mit == nur Strings (Arrays) gleicher Länge elementweise verglichen werden können

```
>> 'zwei'=='drei'
ans =
  1x4 logical array
   0   0   1   1
>> 'zwei'=='dre'
Matrix dimensions must agree.
>> strcmp('zwei','dre')
ans =
  logical
   0
```

- Für **Eingabe eines Strings** statt Zahl wird `input` zusätzlich mit `,'s'` aufgerufen.

```
>> x=input('Eingabe:','s')
Eingabe:eins
x =
    'eins'
>> x=input('Eingabe:','s')
Eingabe:1
x =
    '1'
```

- Die **Umwandlung eines "Zahl"-Strings in eine numerische Zahl und umgekehrt** geht dann mit `str2num` und `num2str`

```
>> y=str2num(x)+1
y =
     2
>> disp(['y␣ist␣',num2str(y)])
y ist 2
```

- Für **anwenderfreundliche Eingaben mathematischer Funktionen** ist auch die **Umwandlung eines Strings in ein Function Handle** mit `str2func` nützlich

```
>> x=input('Eingabe:','s')
Eingabe:t.^2
x =
    't.^2'
>> s=['@(t)',x]
s =
    '@(t)t.^2'
>> f=str2func(s)
f =
  function_handle with value:
    @(t)t.^2
>> f(-2)
ans =
     4
```

- Die bequeme **formatierte Bildschirmausgabe** geht auch mit `fprintf`.

```
>> fprintf('n=%5d und\nx=%5.2f und\nc=%5s\n',100,pi,'null')
n=  100 und
x= 3.14 und
c= null
```

Dabei gilt

  - `\n` bewirkt einen **Zeilenumbruch**
  - die **Prozentzeichen** werden **durch die Werte am Ende in der angegebenen Reihenfolge ersetzt**,
  - die `5` nach dem Prozentzeichen gibt die **(minimale) Breite des Textfeldes für die Werte** an, welches dann bei Bedarf mit führenden Leerzeichen gefüllt wird,
  - das `d` steht für **ganze Zahlen**,
  - `.2f` steht für **Gleitpunkt-Zahlen mit Angabe von** `2` **Nachkommastellen**,
  - und `s` steht für einen **String**.

- Auch **in Textdateien** kann man mit `fprintf` **schreiben**. Dabei werden **Dateien** mit `fopen, fclose` **geöffnet (erzeugt) bzw. geschlossen**.
Schreiben wir z.B. eine Matlab-Funktion, die **Funktionswerte tabellarisch in eine Textdatei** schreibt.

```
function textdatei(name,x,fx)
fileID=fopen([name,'.txt'],'w'); % Öffnet/Erzeugt Datei zum Überschreiben
fprintf(fileID,'%7s | %7s \n','x','f(x)');
for k=1:numel(x)
    fprintf(fileID,'%7.4f | %7.4f \n',x(k),fx(k));
end
fclose(fileID); % Schließt Datei
end
```

```
>> x=linspace(0,2,5); textdatei('parabel',x,x.^2)
>> open('parabel.txt') % im Editor öffnen
```

Bemerkung: Statt Text-Dateien mit der Dateiendung .txt kann man z.B. **auch Latex-Dateien mit .tex** und **Matlab-Dateien mit .m** erzeugen.

10 Vergleiche mit switch

- Die allgemeine Form von **Vergleichen mit switch** ist

```
switch var
   case Wert_1
      Anweisungen_1
         ...
   case Wert_n
      Anweisungen_n
   otherwise  % optional
      Anweisungen_n+1
end
```

Beschreibung:

  – Es wird solange nacheinander überprüft, ob Variable `var` den Wert `Wert_1,...,Wert_n` hat, bis zum ersten mal Gleichheit gilt. Hat dann `var` den Wert `Wert_k`, so werden die `Anweisungen_k` ausgeführt (und nur diese), und nicht mehr weiter überprüft.
  – `otherwise` ist optional. Hat `var` keinen der angegebenen Werte, so werden die `Anweisungen_n+1` nach `otherwise` ausgeführt.
  – Es kann **entweder mit Zahlen oder Strings verglichen** werden.

- **Beispiel** zum Plotten von **Balken- und Kuchen-Diagrammen**

```
function switchdemo(x,auswahl)
switch auswahl
    case 'balken'
        bar(x)
    case 'kuchen'
        pie(x)
    otherwise
        pie3(x);
end
end
```

Zum Testen erzeugen wir einen **Vektor mit** 5 **im Intervall** $[0, 1]$ **gleichverteilten Zufallszahlen**

```
>> x=rand(1,5)
>> switchdemo(x,'balken')
>> switchdemo(x,'kuchen')
>> switchdemo(x,'kuch')
```

- **Switch-Vergleiche lassen sich** natürlich auch **immer als if-Abfragen schreiben**. Hier können wir z.B. die Funktion `strcmp` verwenden, die zwei Strings auf Gleichheit überprüft. Allerdings sind **Switch-Vergleiche oft angenehm im Programmcode zu lesen**.

```
function ifdemo(x,auswahl)
if strcmp(auswahl,'balken')
    bar(x)
elseif strcmp(auswahl,'kuchen')
    pie(x)
else
    pie3(x);
end
end
```

11 Matrizen

- **Matrizen sind 2-dimensionale Arrays** und werden wie Vektoren **durch eckige Klammern erzeugt**. Dabei werden **Zileneinträge durch Komma oder Leerzeichen**, und **Spalteneinträge durch Semikolon getrennt**.

```
>> A=[1 2 3;4 5 6]
A =
     1     2     3
     4     5     6
>> numel(A)  % Gesamtzahl aller Einträge
ans =
     6
>> [m,n]=size(A) % Anzahl der Zeilen m und Spalten n
m =
     2
n =
     3
```

- **Transponieren** (d.h. mache aus Zeilen Spalten) geht mit `'` oder `transpose`.

```
>> A'
ans =
     1     4
     2     5
     3     6
>> x=1:3, x'
x =
     1     2     3
ans =
     1
     2
     3
```

- **Matrizen gleicher Dimension** kann man **komponentenweise addieren, mit Skalar addieren oder multiplizieren**.

```
>> 2*A, A+1, A-A
ans =
     2     4     6
     8    10    12
ans =
     2     3     4
     5     6     7
ans =
     0     0     0
     0     0     0
```

- **Matrizen gleicher Dimension** kann man **elementweise multiplizieren, dividieren, potenzieren** mit `.*` und `./` und `.^` (oder auch mit Skalar), und **elementare Funktionen wirken automatisch elementweise**.

```
>> A.*A, A.^2, 2.^A, 1./A, exp(A)
ans =
     1     4     9
    16    25    36
```

```
ans =
     1     4     9
    16    25    36
ans =
     2     4     8
    16    32    64
ans =
    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
ans =
    2.7183    7.3891   20.0855
   54.5982  148.4132  403.4288
```

- Produktzeichen `*` ohne Punkt entspricht **Matrix/Vektor-Multiplikation nach Rechenregeln bei passender Dimension** "$(m \times n) \cdot (n \times k) = (m \times k)$".

```
>> A*x
Error using  *
Inner matrix dimensions must agree.
>> A*x'
ans =
    14
    32
>> A*A'
ans =
    14    32
    32    77
>> A'*A
ans =
    17    22    27
    22    29    36
    27    36    45
>> x*x'
ans =
    14
>> x'*x
ans =
     1     2     3
     2     4     6
     3     6     9
```

- Bei **qadratischen Matrizen** Q ist `^` ohne Punkt **potenzieren** im Sinne von $Q^k = \underbrace{Q \cdots Q}_{k\text{-mal}}$.

```
>> A^2
Error using  ^
One argument must be a square matrix and the other must be a
scalar. Use POWER (.^) for elementwise power.
>> Q=[1 2;3 4]
Q =
     1     2
     3     4
>> Q*Q, Q^2, Q*Q*Q, Q^3
ans =
     7    10
```

```
    15    22
ans =
     7    10
    15    22
ans =
    37    54
    81   118
ans =
    37    54
    81   118
```

- **Matrizen passender Dimensionen** lassen sich zu größeren Matrizen **zusammenfassen**.

```
>> M=[eye(4),zeros(4,2);3*ones(2,2),rand(2,3),[5;6]]
M =
    1.0000         0         0         0         0         0
         0    1.0000         0         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000         0         0
    3.0000    3.0000    0.8147    0.1270    0.6324    5.0000
    3.0000    3.0000    0.9058    0.9134    0.0975    6.0000
```

- **Matrizen speichern und laden**

```
>> save('matrix','M')
>> clear('M')
>> load('matrix')
>> M
M = ...
```

- Auf **einzelne Einträge in Matrizen** kann man **mit runden Klammern zugreifen**. Dabei gibt es **mehrere Möglichkeiten**

  – Ein **Indexpaar (Zeilenindex, Spaltenindex)**. Dabei bedeutet Doppelpunkt ganze Zeile bzw. Spalte

```
>> A=[1 2 3;4 5 6]
A =
     1     2     3
     4     5     6
>> A(1,3), A(2,1), A(2,:), A(2,[1 1 3 2]), A([2,1],[1 1 3 2])
ans =
     3
ans =
     4
ans =
     4     5     6
ans =
     4     4     6     5
ans =
     4     4     6     5
     1     1     3     2
```

- **Lineare Indizierung mit nur einem Index**, so **als würden alle Spalten der Matrix untereinander gehängt** werden zu einem langen Vektor (dies lässt sich auch mit dem Doppelpunktoperator explizit erzeugen)

```
>> B=A(:)
B =
     1
     4
     2
     5
     3
     6
>> A(3), B(3)
ans =
     2
ans =
     2
>> A([3 2]), B([3 2]), A([3;2])
ans =
     2     4
ans =
     2
     4
ans =
     2
     4
```

- **Logische Indizierung mit logischen Matrizen gleicher Dimension**. Das ist **nützlich um alle Einträge zu suchen/verändern, die einer Bedingung genügen**.

```
>> ind=(A<=4)
ind =
  2x3 logical array
   1   1   1
   1   0   0
>> B=A(ind)   % Reihenfolge entspricht dabei linearer Indizierung
B =
     1
     4
     2
     3
>> A(ind)=0
A =
     0     0     0
     0     5     6
```

- **In Matlab (= Matrix Laboratory)** sind **obige Matrix-Operationen und sehr viele weitere Algorithmen für und mit Matrizen effizient und robust implementiert**, z.B.
- **Lineare Gleichungssysteme lösen mit Backslash-Operator** "\", z.B.

$$x_1 + 2x_2 = 5$$
$$2x_1 + x_2 = 1$$

in Matrix-Vektor-Form

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$$

d.h. $A \cdot x = y$ mit Koeffizientenmatrix $A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ und Rechter-Seite-Vektor $y = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$.

```
>> A=[1 2;2 1], y=[5;1], x=A\y
A =
     1     2
     2     1
y =
     5
     1
x =
    -1
     3
```

zur Kontrolle die Probe

```
>> A*x
ans =
     5
     1
```

- **Determinante und inverse Matrix** A^{-1} einer (invertierbaren) **quadratischen Matrix** A

```
>> det(A)
ans =
    -3
>> B=inv(A)
B =
   -0.3333    0.6667
    0.6667   -0.3333
>> B*A
ans =
     1     0
     0     1
```

- **LR-Zerlegung** $P \cdot A = L \cdot R$ mit Permutationsmatrix P, linker unterer Dreiecksmatrix L (mit Einsen auf Diagonale) und rechter oberer Dreiecksmatrix R. Im Englischen heißt es **LU-Decomposition** für lower und upper.

```
>> [L,U,P]=lu(A)
L =
    1.0000         0
    0.5000    1.0000
U =
    2.0000    1.0000
         0    1.5000
P =
     0     1
     1     0
>> P*A, L*U
ans =
     2     1
     1     2
ans =
     2     1
     1     2
```

- **Eigenwerte und Eigenvektoren einer quadratischen Matrix** A, d.h. finde Zahlen λ und Vektoren v mit $A \cdot v = \lambda \cdot v$

```
>> [V,D]=eig(A)
V =                        % Spalten von V sind Eigenvektoren
   -0.7071    0.7071
    0.7071    0.7071
D =                        % Auf Diagonale von D stehen Eigenwerte
    -1     0
     0     3
>> lambda=diag(D)
lambda =
    -1
     3
>> A*V(:,1), A*V(:,2), 3*V(:,2)
ans =
    0.7071
   -0.7071
ans =
    2.1213
    2.1213
ans =
    2.1213
    2.1213
>> V*D*V'    % Es gilt A=V*D*V'
ans =
    1.0000    2.0000
    2.0000    1.0000
```

- **Numerische Lösung von Optimierungsproblemen**, z.B.

- **Barkeeper-Problem als Lineares Programm**. Welche der folgenden Cocktails sollten zubereitet werden um den Gewinn zu maximieren

  - Daiquiri für 6 Euro: 4.5 cl weißer Rum, 3 cl Cointreau, Zitronensaft, Zuckersirup, Eis
  - Kamikaze für 5 Euro: 3 cl Wodka, 3 cl Cointreau, Zitronensaft, Limonensirup, Eis
  - Long Island Ice Tea für 7.50 Euro: 2 cl Wodka, 2 cl weißer Rum, 2 cl Gin, 2 cl Cointreau, Zitronensaft, Orangensaft, Cola, Eis

  wenn die Spirituosen in folgenden Mengen vorhanden sind: 6 l weißer Rum, 7 l Cointreau, 5 l Wodka und 3 l Gin?

  - Anzahl der Cocktails als **Variablen**: x_1 Daiquiri, x_2 Kamikaze, x_3 Long Island Ice Tea
  - Maximiere **Zielfunktion** Gewinn: $\max\limits_{x_1,x_2,x_3} 6 \cdot x_1 + 5 \cdot x_2 + 7.5 \cdot x_3$
  - unter den **Nebenbedingungen** in ml:

$$
\begin{aligned}
\text{(Rum)} \quad & 45 \cdot x_1 + 20 \cdot x_3 \leq 6000 \\
\text{(Cointreau)} \quad & 30 \cdot x_1 + 30 \cdot x_2 + 20 \cdot x_3 \leq 7000 \\
\text{(Wodka)} \quad & 30 \cdot x_2 + 20 \cdot x_3 \leq 5000 \\
\text{(Gin)} \quad & 20 \cdot x_3 \leq 3000 \\
\text{und} \quad & x_1, x_2, x_3 \geq 0
\end{aligned}
$$

– **Für den Matlab-Löser** `linprog` muss dies **als Minimierungsproblem in Matrix-Vektor-Form** geschrieben werden:

$$(-)\min_x \langle f\,,x\rangle \quad \text{unter den NB} \quad A\cdot x\le b \quad , \quad x\ge 0$$

mit $x=(x_1,x_2,x_3)$, $f=-(6,5,7.5)$, $b=(6000,7000,5000,3000)$ und

$$A=\begin{pmatrix}45 & 0 & 20\\ 30 & 30 & 20\\ 0 & 30 & 20\\ 0 & 0 & 20\end{pmatrix}$$

wobei wir keine oberen Schranken (upper bounds) für x und keine Gleichungen (equalities), sondern nur Ungleichungen haben

```
>> f=-[6,5,7.5]; b=[6000,7000,5000,3000]; lb=zeros(size(f)); ub=[];
>> A=[45,0,20; 30,30,20; 0,30,20; 0,0,20]; Aeq=[]; beq=[];
>> [x,fx]=linprog(f,A,b,Aeq,beq,lb,ub)
x =
   66.6667
   66.6667
  150.0000
fx =
  -1.8583e+03
>> A*x
ans =
   1.0e+03 *
    6.0000
    7.0000
    5.0000
    3.0000
```

Will man tatsächlich **ganzzahlige Lösungen** garantieren, so kann man `intlinprog` verwenden, bei dem zusätzlich die Indizes der als ganzahlig zu behandelnden Variablen übergeben werden können, hier also alle drei. (**Bemerkung:** Das Finden ganzzahliger Lösungen ist **i.a. algorithmisch wesentlich aufwändiger und daher bei großen Problemen langsamer!**)

```
>> intcon=1:3; % integer constraints=ganzzahlige Variablen
>> [x,fx]=intlinprog(f,intcon,A,b,Aeq,beq,lb,ub)
x =
   67.0000
   67.0000
  149.0000
fx =
  -1.8545e+03
>> A*x
ans =
   1.0e+03 *
    5.9950
    7.0000
    4.9900
    2.9800
```

- **Sensornetzwerk zur Lokalisierung als nichtlineares Ausgleichsproblem (nonlinear least squares).**

Gegeben sind n **Sensoren mit** den **bekannten Ortskoordinaten** $(p_{i,1}, p_{i,2}) \in \mathbb{R}^2$ für $i = 1, \ldots, n$. **Gesucht ist die Position** (x_1, x_2) **eines Objektes**, welches **zum unbekannten Zeitpunkt** t_0 ein **Signal mit bekannter Geschwindigkeit** $v = 1$ aussendet. Sensor i empfängt zur Zeit t_i dieses Signal. Zwischen den gemessenen Zeiten t_i und zurückgelegten Signal-Wegen s_i besteht der Zusammenhang

$$s_i = v \cdot (t_i - t_0) = t_i - t_0 \quad (\text{für } v = 1)$$

Wir schreiben dies zunächst als mehrdimensionales Nullstellenproblem mit den 3 Unbekannten $x = (x_1, x_2, t_0)$ und n Gleichungen

$$0 = f_i(x) := s_i - t_i + t_0 = \sqrt{(p_{i,1} - x_1)^2 + (p_{i,2} - x_2)^2} - t_i + t_0 \quad \text{für } i = 1, \ldots, n$$

Da die gemessenen Zeiten aber oftmals fehlerbehaftet sein können (atmosphärisches Rauschen etc.), formulieren wir dies als nichtlineares Ausgleichsproblem

$$\min_x \sum_{i=1}^{n} \big(f_i(x)\big)^2$$

Der **Matlab-Löser** `lsqnonlin` **verlangt dabei ein Function-Handle auf eine Funktion mit** Vektor x als Eingabevariable und **Vektor der Funktionswerte** $f_i(x)$ **als Rückgabevariable**. Hier übergeben wir zusätzlich noch die Sensorpositionen als Matrix

$$p = \begin{pmatrix} p_{1,1}, & p_{1,2} \\ \vdots & \vdots \\ p_{n,1}, & p_{n,2} \end{pmatrix}$$

(d.h. die erste Spalte enthält die x-Koordinaten und die zweite Spalte enthält die y-Koordinaten der Punkte), und die gemessenen Zeiten als Vektor $t = (t_1, \ldots, t_n)$.

```
function f=sensorfunktion(x,p,t)
f=sqrt((p(:,1)-x(1)).^2+(p(:,2)-x(2)).^2)-t+x(3);
end
```

Zum **Testen** setzen wir die **Sensoren in die Ecken des Quadrats** $[-1, 1]^2$ und lassen **darin zufällig die Position des Objektes** erzeugen. Auch der **Zeitpunkt** t_0 soll **zufällig** im Intervall $[0, 1]$ gewählt werden.

```
>> p=[1,1;1,-1;-1,-1;-1,1]
p =
     1     1
     1    -1
    -1    -1
    -1     1
>> x=[2*(rand(1,2)-0.5),rand(1)]   % Position x(1),x(2), Zeitpunkt t0=x(3)
x =
    0.3115   -0.9286    0.8491
```

Nun brauchen wir **noch eine Funktion, die** uns zu den gegebenen Sensorpositionen, Position des Objektes und Startzeit t_0 **die gemessenen Sensorzeiten** t **erzeugt** mit

$$t_i = \sqrt{(p_{i,1} - x_1)^2 + (p_{i,2} - x_2)^2} + t_0 \quad \text{für } i = 1, \ldots, n$$

```
function t=sensorzeit(x,p)
t=sqrt((p(:,1)-x(1)).^2+(p(:,2)-x(2)).^2)+x(3);
end
```

```
>> t=sensorzeit(x,p)
t =
    2.8969
    1.5413
    2.1626
    3.1814
```

Da der **Matlab-Löser** `lsqnonlin` **auch noch einen (sinnvollen) Startwert** verlangt, übergeben wir **hier der Einfachheit halber den Nullvektor**.

```
>> x_berechnet=lsqnonlin(@(x) sensorfunktion(x,p,t),zeros(1,3))
x_berechnet =
    0.3115   -0.9286    0.8491
```

Jetzt verrauschen wir die gemessenen Zeiten noch zufällig, so dass für den relativen Fehler $\frac{|t_i - t_{verrauscht,i}|}{|t_i|} \leq 0.1$ gilt, d.h. einem maximalen Fehler von 10%.

```
>> t_verrauscht=(1+0.2*(rand(size(t))-0.5)).*t
t_verrauscht =
    3.1484
    1.5964
    2.2740
    3.3361
>> x_berechnet=lsqnonlin(@(x) sensorfunktion(x,p,t_verrauscht),zeros(1,3))
x_berechnet =
    0.3203   -1.0669    0.9304
```

Das **Ergebnis** ist also **immer noch brauchbar**.

12 Cell Arrays

- Ein **Cell Array** ist ein **Array von Zellen**, in denen man **verschiedene Datentypen** speichern kann. **Erzeugt** wird ein Cell Array **mit geschweiften Klammern** `{...}`, oder auch z.B. vor Schleifen initialisiert mit `cell`.

```
>> C={}
C =
  0x0 empty cell array
>> C=cell(1,3)
C =
  1x3 cell array
    {0x0 double}    {0x0 double}    {0x0 double}
```

- Das ist z.B. **nützlich um Vektoren/Matrizen unterschiedlicher Dimension gemeinsam** in einer Variable speichern zu können, **oder mehrere Strings oder Function Handles**.

```
>> C={1:3,[1 2;3 4]}
C =
  1x2 cell array
    {1x3 double}    {2x2 double}
>> C={'eins','zwei'}
C =
  1x2 cell array
    {'eins'}    {'zwei'}
>> C={@(x) x.^2, @sin}
C =
  1x2 cell array
    {@(x)x.^2}    {@sin}
```

- Allerdings **benötigen Cell Arrays mehr Speicher** und erlauben **keine effizienten Array-Operationen wie numerische Arrays**.

```
>> C={1,2}, c=[1,2]
C =
  1x2 cell array
    {[1]}    {[2]}
c =
     1     2
>> whos('C','c')
  Name      Size            Bytes  Class     Attributes

  C         1x2               240  cell
  c         1x2                16  double
```

- Für die **Indizierung** gibt es zwei Möglichkeiten: **Cell Indexing** und **Content Indexing**.
  - **Cell Indexing**: Mit Indices in **runden Klammern** `(...)` hat man Zugriff auf die angegebenen **Zellen als Ganzes**. So kann man z.B. einen Teil des Cell Arrays zuweisen, wie bei Matrizen mit Indexpaaren, linearer oder logischer Indizierung.

```
>> x=linspace(-1,1,5); C={x,'LineWidth';x.^2,10}
C =
  2x2 cell array
    {1x5 double}    {'LineWidth'}
    {1x5 double}    {[       10]}
```

```
>> D=C(1,:)
D =
  1x2 cell array
    {1x5 double}    {'LineWidth'}
>> D=C([false,true;false,true])
D =
  2x1 cell array
    {'LineWidth'}
    {[       10]}
```

Auf die Inhalte selbst kann man so aber nicht zugreifen.

```
>> D=C(4)
D =
  1x1 cell array
    {[10]}
>> D+1
Undefined operator '+' for input arguments of type 'cell'.
```

– **Content Indexing**: Mit Indices in **geschweiften Klammern** `{...}` hat man Zugriff auf die **Inhalte der angegebenen Zellen**.

```
>> D=C{4}
D =
    10
>> D+1
ans =
    11
```

Mit **Content Indexing** hat man allerdings so direkt **nur Zugriff auf den Inhalt einer einzelnen Zelle** (dies rührt daher, dass die Inhalte ja ganz verschiedene Typen sein können). **Mehrere Inhalte kann man z.B. durch geeignete for-Schleife abrufen, oder an eine durch Kommata getrennte Liste übergeben**, wie bei den Rückgabevariablen von Funktionen.

```
>> C{:}    % alle Inhalte anschauen (lineare Indizierung)
ans =
   -1.0000   -0.5000         0    0.5000    1.0000
ans =
    1.0000    0.2500         0    0.2500    1.0000
ans =
    'LineWidth'
ans =
    10
>> [a,b,c,d]=C{:}    % alle Inhalte einzeln zuweisen
a =
   -1.0000   -0.5000         0    0.5000    1.0000
b =
    1.0000    0.2500         0    0.2500    1.0000
c =
    'LineWidth'
d =
    10
>> [a,b]=C{:,2}    % nur Inhalte der zweiten Zeile einzeln zuweisen
a =
    'LineWidth'
b =
```

```
        10
    >> plot(C{:})    % alle Inhalte einzeln an Funktion übergeben
```

- Wie bei anderen Arrays kann man **Cell Arrays passender Dimensionen auch zu einem größeren Cell Array zusammenfassen**.

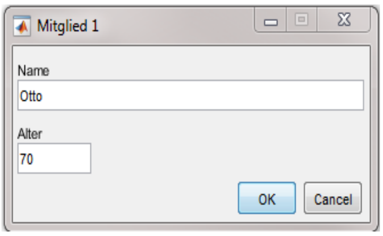
```
>> D=[C,C(:,2)]
D =
  2x3 cell array
    {1x5 double}    {'LineWidth'}    {'LineWidth'}
    {1x5 double}    {[        10]}    {[        10]}
```

- **Vorsicht:** Geschweifte Klammern im selben Beispiel erzeugen dagegen ein 1×2-Cell Array, dessen Zellen wieder Cell Arrays sind (hier dürften die Dimensionen auch verschieden sein).

```
>> D={C,C(:,2)}
D =
  1x2 cell array
    {2x2 cell}    {2x1 cell}
```

- **Viele Matlab-Funktionen verwenden Cell Arrays**, insbesondere als Cell Arrays von Strings. Zum **Beispiel** `inputdlg` zum Erzeugen einer **Dialog-Box mit Eingabefeldern** bestimmter Breite.

```
>> C=inputdlg({'Name','Alter'},'Mitglied␣1', [1 50; 1 10])
C =
  2x1 cell array
    {'Otto'}
    {'70'  }
>> name=C{1}, alter=str2num(C{2})
name =
    'Otto'
alter =
    70
```



- **Variable Anzahl von Ein-/und Ausgabevariablen von Funktionen** mit Hilfe von `varargin` und `varargout`:

  Der allgemeinen Funktionskopf lässt sich durch `varargin` und `varargout` ergänzen. Beide sind jeweils optional, müssen aber **am Ende der jeweiligen Variablenliste** stehen (bzw. allein).

```
function [out1, ..., varargout] = myfun(in1, ..., varargin)
```

  In `varargin` werden **alle optional übergebenen Eingabevariablen als Cell Array zusammengefasst**. Ebenso müssen **alle optional angeforderten Ausgabevariablen als Cell Array in `varargout` zugewiesen werden**.

  Beispiele: Wir erweitern unsere Funktion `switchdemo` (siehe Vergleiche mit switch) so, dass **beliebig viele Eingabepaare** der Form `x,auswahl` übergeben werden können

```
function switchdemo_var(varargin)
n=numel(varargin); % hier auch: n=nargin;
for k=1:n/2
    auswahl=varargin{2*k};
    x=varargin{2*k-1};
    figure(k);
    switch auswahl
        case 'balken'
            bar(x)
        case 'kuchen'
            pie(x)
        otherwise
            pie3(x);
    end
end
```

  Test:

```
>> switchdemo_var(rand(1,7),'kuchen',rand(1,10),'balken')
```

  Bei der nächsten Funktion kann eine **beliebige Anzahl quadratischer Zufallsmatrizen zurückverlangt** werden, deren Größe bis zur angeforderten Anzahl wächst.

```
function varargout=zufallsmatrizen
    n=nargout;
    varargout=cell(1,n);
    for k=1:n
       varargout{k}=rand(k);
    end
end
```

```
>> [A,B,C]=zufallsmatrizen
A =
    0.2511
B =
    0.6160    0.3517
    0.4733    0.8308
C =
    0.5853    0.2858    0.3804
    0.5497    0.7572    0.5678
    0.9172    0.7537    0.0759
```

13 Structures

- Auch **eine Structure kann beliebige Datentypen enthalten**, welche **in Feldern** (Fields) gespeichert werden. Im Unterschied zu Cell Arrays werden die Felder aber **nicht numerisch indiziert, sondern mit dem Feldnamen**.

- Eine **Structure** lässt sich durch `var=struct(Feldname_1,Inhalt_1,...,Feldname_n,Inhalt_n)` **erzeugen**:

```
>> Mitglied1=struct('Name','Otto','Alter',70)
Mitglied1 =
  struct with fields:
     Name: 'Otto'
    Alter: 70
```

- **Auf den Inhalt der Felder wird mit dem entsprechenden Feldnamen zugegriffen** (`var.Feldname`)

```
>> Mitglied1.Alter=Mitglied1.Alter+1;
>> Mitglied1.Alter
ans =
    71
```

- **Neue Felder lassen sich einfach hinzufügen**...

```
>> Mitglied1.Funktion=@(x) x.^2
Mitglied1 =
  struct with fields:
        Name: 'Otto'
       Alter: 71
    Funktion: @(x)x.^2
>> Mitglied1.Funktion(-1)
ans =
     1
```

- ... **und entfernen**

```
>> Mitglied1=rmfield(Mitglied1,'Funktion')
Mitglied1 =
  struct with fields:
     Name: 'Otto'
    Alter: 71
```

- Bei **Structure Arrays** muss **jedes Element des Arrays eine Structure mit gleicher Anzahl von gleichnamigen Feldern** haben. Allerdings dürfen die **Felder gleichen Namens Inhalte verschiedenen Typs** haben.

```
>> Mitglied2=struct('Name','Ute','Alter','jung')
Mitglied2 =
  struct with fields:
     Name: 'Ute'
    Alter: 'jung'
>> Mitglied=[Mitglied1,Mitglied2]
Mitglied =
  1x2 struct array with fields:
    Name
    Alter
```

```
>> numel(Mitglied)
ans =
     2
>> Mitglied(2).Alter
ans =
    'jung'
```

Es ginge **auch direkt mit**

```
>> Mitglied=struct('Name',{'Otto','Ute'},'Alter',{70,'jung'})
Mitglied =
  1x2 struct array with fields:
    Name
    Alter
```

- Die **Feldnamen in einem Cell Array** erhält man mit `fieldnames`.

```
>> f=fieldnames(Mitglied)
f =
  2x1 cell array
    {'Name' }
    {'Alter'}
```

- Die **Feldnamen** lassen sich auch **mit einem String angeben** `var.(string)`, wobei `string` einen **String für einen zulässigen Feldnamen** liefert (also z.B. erstes Zeichen ein Buchstabe und keine Zahl) und **in runden Klammern** stehen muss.

```
>> s=datestr(now,'mmm_dd_yyyy_HH_MM_SS','local')
s =
    'Nov_22_2018_09_13_03'
>> preis.s=10
preis =
  struct with fields:
    s: 10
>> preis.(s)=10
preis =
  struct with fields:
                       s: 10
    Nov_22_2018_09_13_03: 10
>> preis.(datestr(now,'mmm_dd_yyyy_HH_MM_SS','local'))=15
preis =
  struct with fields:
                       s: 10
    Nov_22_2018_09_13_03: 10
    Nov_22_2018_09_14_53: 15
```

- **Auch Structures werden von vielen Matlab-Funktionen genutzt**, z.B. beim **Laden von gespeicherten Variablen**

```
>> a=[1 2]; b='hallo'; save('testdaten','a','b')
>> s=load('testdaten')
s =
  struct with fields:
    a: [1 2]
    b: 'hallo'
```

- Als **Beispiel** noch eine **Structure für ein Polygon, dessen Ecken als Matrix gespeichert werden**, wobei die erste Zeile die x-Koordinaten und die zweite Zeile die y-Koordinaten enthält. Ausserdem werden **noch Drehwinkel, Position und Farbe** angegeben, z.B. ein rotes Dreieck mit den Eckpunkten $(0,0)$, $(1,0)$ und $(0,1)$.

```
>> p=struct('Ecken',[0,1,0;0,0,1],'Position',[0;0],'Winkel',0,'Farbe','r')
p =
  struct with fields:

       Ecken: [2x3 double]
    Position: [2x1 double]
      Winkel: 0
       Farbe: 'r'
```

  mit der folgenden Funktion können wir das Polygon ausgefüllt plotten lassen, wobei die **Drehung um den Ursprung durch eine Drehmatrix** erzeugt wird.

```
function male(p)
phi=p.Winkel;
D=[cos(phi), -sin(phi);sin(phi), cos(phi)];
e=D*p.Ecken;
fill(p.Position(1)+e(1,:),p.Position(2)+e(2,:),p.Farbe);
axis('equal');
end
```

  Nun **plotten** wir das Polygon, dann **drehen und verschieben** wir es und **plotten es nochmal**.

```
>> male(p)
>> p.Winkel=pi/8; p.Position=[2;3]
p =
  struct with fields:

       Ecken: [2x3 double]
    Position: [2x1 double]
      Winkel: 0.3927
       Farbe: 'r'
>> male(p)
```

14 Huffman-Kodierung

- Ziel der **Huffman-Kodierung** ist die **verlustlose Datenkompression** (lossless compression).
- **Standard-Kodierung**: Meist wird **jedes Textzeichen mit einer gleichen Anzahl von Bits abgespeichert**. In Matlab benötigt jedes Zeichen 2 Bytes = 16 Bits, das sind $2^{16} = 65536$ verschiedene Zeichen.
- **Beobachtung**: In Texten kommen **einige Zeichen** (z.B. “e”) meist wesentlich **häufiger** vor als andere (z.B. “y”; zumindest in deutschen Texten).
- **Idee der Huffman-Kodierung**: Verwende **nur wenige Bits für die häufigsten Zeichen** und mehr Bits für die weniger häufigen. Wir verdeutlichen das Vorgehen am Beispiel von $n = 4$ verschiedenen Zeichen

  $$z_k \quad , \quad k = 1, \ldots, n$$

  Bei einer Standard-Kodierung braucht jedes Zeichen 2 Bits (z.B. $z_1 : 00$, $z_2 : 01$, $z_3 : 10$, $z_4 : 11$).
  Annahme: Man benutzt Texte, in denen **Zeichen** z_k **mit relativer Häufigkeit**

  $$w_k \in [0,1] \quad , \quad \sum_{k=1}^{n} w_k = 1$$

  auftaucht. Z.B. tauchen in dem 10 Buchstaben langen Text **“KAFFEE AFFE”** (hier ohne Leerzeichen gezählt) die Zeichen

  $$z_1 = K \quad , \quad z_2 = A \quad , \quad z_3 = E \quad , \quad z_4 = F$$

  mit den relativen Häufigkeiten

  $$w_1 = 0.1 \quad , \quad w_2 = 0.2 \quad , \quad w_3 = 0.3 \quad , \quad w_4 = 0.4$$

  auf.

- **Konstruiere daraus folgendermaßen einen Baum:**

  (1) Betrachte die **Zeichen** z_k **als Blätter/Knoten mit Wert** w_k.

  (2) Hänge die **beiden Knoten** z_{k_1}, z_{k_2} mit **kleinsten Werten an einen neuen Knoten** z_{neu} und **gib diesem den Wert**

  $$w_{neu} = w_{k_1} + w_{k_2}$$

  Gib dabei dem **Zweig von** z_{neu} **zu** z_{k_1} den **Wert 0** und dem **Zweig von** z_{neu} **zu** z_{k_2} den **Wert 1** falls $w_{k_1} \leq w_{k_2}$.

  (3) Betrachte nur noch die **Menge der restlichen Knoten ohne** z_{k_1}, z_{k_2} **aber vereinigt mit** z_{neu}, und **wiederhole (2) und (3) bis alle Knoten verarbeitet** sind.

- **Verdeutlichung an obigem Beispiel:**

```
z1=K  z2=A    z3=E    z4=F
0.1   0.2     0.3     0.4
 \     /      /        /
 0\   /1     /        /
   \/       /        /
   0.3     /        /
     \    /        /
     1\  /0       /      (hier ginge auch 0\  /1)
       \/        /
       0.6      /
         \     /
         1\   /0
           \ /
            *
```

Die **Bitfolge** für Zeichen z_k erhält man nun **von der Wurzel $*$ bis zum jeweiligen Zeichen**. Hier also:

$$z_1 = K : 110 \quad , \quad z_2 = A : 111 \quad , \quad z_3 = E : 10 \quad , \quad z_4 = F : 0$$

Die **durchschnittliche Bitlänge** für die betrachteten Texte mit dieser **Huffman-Kodierung** ist folglich **geringer**

$$w_1 \cdot 3 + w_2 \cdot 3 + w_3 \cdot 2 + w_4 \cdot 1 = 0.1 \cdot 3 + 0.2 \cdot 3 + 0.3 \cdot 2 + 0.4 \cdot 1 = 1.9 < 2$$

Man benötigt damit also im Durchschnitt nur $1.9/2 = 95\%$ des ursprünglichen Speicherbedarfs. Das Wort **"AFFE"** hätte hiermit den **Code 1110010**, d.h. **nur 7 statt 8 Bits**.

- **Noch ein Beispiel:**

```
z1     z2      z3   z4     z5
0.1   0.15     02  0.25   0.3
 \     /      /     \     /
 0\   /1     /      0\   /1
   \/       /          \/
   0.25    /          0.55
     \    /          /
     1\  /0         /
       \/          /
       0.45       /
         \       /
         0\   /1
           \ /
            *
```

d.h.

$$z_1 : 010 \quad , \quad z_2 : 011 \quad , \quad z_3 : 00 \quad , \quad z_4 : 10 \quad , \quad z_5 : 11$$

Die durchschnittliche Bitlänge ist

$$0.1 \cdot 3 + 0.15 \cdot 3 + 0.2 \cdot 2 + 0.25 \cdot 2 + 0.3 \cdot 2 = 2.25$$

Da nach dem ersten Schritt z_{neu} nach Zusammenfügen von z_1 und z_2 den gleichen Wert $0.1 + 0.15 = 0.25$ wie z_4 hat, könnte man je nach Sortierung auch den folgenden Baum erhalten:

```
z1      z2       z3    z4    z5
0.1    0.15      02   0.25  0.3
 \     /         \     /   /
 0\   /1         0\   /1  /
   \/              \/   /
   0.25          0.45  /
      \             \/
       \            /\
        \          /  \
         \        /    \
         0\     /1      \0
           \   /         \
           0.55---------*
                    1
```

d.h.

$$z_1 : 100 \quad , \quad z_2 : 101 \quad , \quad z_3 : 00 \quad , \quad z_4 : 01 \quad , \quad z_5 : 11$$

Der **Baum** ist demnach zwar **nicht eindeutig**, die **durchschnittliche Bitlänge ist aber die selbe**.

- **In Matlab** lassen sich solche **Bäume t recht anschaulich mit Hilfe von Structures** erzeugen. Dabei wählen wir die Feldnamen a für 0 und b für 1, da Feldnamen nicht mit einer Zahl anfangen dürfen. Der **Baum aus dem ersten Beispiel** lässt sich dann **schrittweise** folgendermaßen **zunächst per Hand erzeugen**.

```
>> t.a='K'; t.b='A'
t =
  struct with fields:

    a: 'K'
    b: 'A'
>> t.b=t; t.a='E'
t =
  struct with fields:

    a: 'E'
    b: [1x1 struct]
>> t.b=t; t.a='F'
```

Zu den Zeichen gelangen wir dann jeweils, indem wir **ausgehend von der Wurzel den entsprechenden Zweigen des Baumes folgen**.

```
>> [t.b.b.a, t.b.b.b, t.b.a, t.a]
ans =
    'KAEF'
```

- **Dekodierung**: Sei **Code** c **als Array von 0'en und 1'en** gegeben, z.B. wie oben für “AFFE”

```
>> c=[1 1 1 0 0 1 0];
```

Daraus lässt sich einfach **ein Array mit Buchstaben a und b** machen.

```
>> char('a'+c)
ans =
    'bbbaaba'
```

Dann können wir den **ursprünglichen Text** s erhalten, indem wir:

- Den **Code durchlaufen**,
- dabei **ausgehend von der Wurzel den Zweigen des Baumes t solange mit den Anweisungen 0 bzw. a, und 1 bzw. b folgen, bis wir bei einem Zeichen angelangt** sind,
- dann **wieder zur Wurzel springen und wiederholen bis Code ganz durchlaufen**,
- und dabei **alle auftretenden Zeichen aneinanderhängen**.

```
function s=dekodiere(c,t)
m=numel(c);
c=char('a'+c);
s=[];
i=0;
% durchlaufe Code bis Ende
while i<m
    i=i+1;
    % durchlaufe Baum
    knoten=t; % Wurzel
    folge_zweig(knoten.(c(i)));
end
% eingenestete Funktion teilt c,s,i mit Hauptfunktion
    function folge_zweig(knoten)
        if isa(knoten,'struct')
            i=i+1;
            folge_zweig(knoten.(c(i)));
        else
            % knoten ist Blatt/Zeichen
            s=[s,knoten];
        end
    end
end
```

Ein **Test** liefert

```
>> s=dekodiere(c,t)
s =
    'AFFE'
```

- **Automatisches Erzeugen eines Huffmann-Baumes: Version 1 mit Structure t**
  - Alle **verwendeten Zeichen werden als String** `z` übergeben, die entsprechenden **relativen Häufigkeiten als Vektor** `w`.
  - **Zunächst** werden **im Cell Array** `knoten` die **einzelnen Zeichen als anfängliche Knoten (Blätter) abgelegt**.
  - In der **for-Schleife** werden die **Schritte (2) und (3) der Huffman-Baum-Konstruktion wiederholt**: Die **beiden Knoten mit kleinstem Wert zu neuem Knoten mit Summe der Gewichte als neuem Wert zusammenfassen**, und **nur noch neue Knotenmenge betrachten**.
  - Der **letzte übgriggebliebene Knoten ist** dann die **Wurzel unseres Baumes** `t`.

```
function t=huffman_baum(z,w)
n=numel(w);
knoten=cell(1,n);
for k=1:n
    knoten{k}=z(k);
end
for k=1:n-1
    [w_min,ind]=mink(w,2); % die 2 kleinsten w mit Indices
    neuerknoten.a=knoten{ind(1)};
    neuerknoten.b=knoten{ind(2)};
    knoten{ind(1)}=neuerknoten; % wird ersetzt
    w(ind(1))=w_min(1)+w_min(2);
    w(ind(2))=inf; % Wert unendlich (wird nicht mehr als kleinster gewählt)
end
t=neuerknoten;
end
```

Wir **erzeugen damit zum Test den Baum aus dem ersten Beispiel** und vergleichen mit obiger Dekodierung (**Vorsicht:** je nachdem wie die Matlab-Funktion `mink` bei gleichen Einträgen den kleineren wählt, könnte auch der alternative Baum erzeugt werden und der Code müsste entsprechend geändert werden, d.h. `c=[1 0 1 0 0 1 1]` statt `c=[1 1 1 0 0 1 0]`).

```
>> z='KAEF'; w=[0.1 0.2 0.3 0.4];
>> t=huffman_baum(z,w);
>> s=dekodiere(c,t)
s =
    'AFFE'
```

- **Erzeugen der elementaren Codewörter**: Um zu wissen welche Bitfolge (“elementares Codewort”) zu Zeichen z_k gehört, müssen wir den ganzen Baum von der Wurzel zum Zeichen durchlaufen und die den Zweigen zugeordneten Bits aneinanderreihen. Damit werden wir uns **in der Übung** beschäftigen.

- **Speicherung von Bitfolgen**: Matlab benötigt intern zur Speicherung von logischen “Bits” auch 1 Byte=8 Bits.

```
>> c=logical(1)
>> whos('c')
  Name      Size            Bytes  Class      Attributes
  c         1x1                 1  logical
```

Somit würde man in Matlab eigentlich nix gewinnen.

Allerdings kann man **logische Arrays z.B. als .pbm-File (“portable bitmap”) speichern**. Hier z.B. ein zufälliger Code aus logischen 0’en und 1’en der Länge $8 * 5000 = 40000$, der zur Speicherung also auch nur 40000 Bits=5000 Byte (5 kB) benötigen sollte:

```
>> c=logical(randi([0 1],1,8*5000));
>> imwrite(c,'c_logical.pbm');
>> imfinfo('c_logical.pbm')
ans =
  struct with fields:
                ...
          FileSize: 5011  % in Bytes
          ...

>> c=imread('c_logical.pbm');
```

Dabei gibt `FileSize` die Größe in Bytes an (hier werden also zusätzlich noch 11 Bytes zur Speicherung von Bildinformationen benötigt).

- Will man mit dem selben Huffmann-Baum mehrere lange Codes behandeln, so ist obiges **Vorgehen mit Huffmann-Baum als Structure** eigentlich ausreichend, da dieser dann ja nur einmal erzeugt werden muss. Ausserdem ist das prinzipielle Vorgehen **sehr allgemein** und lässt sich **einfach auf beliebige Bäume mit mehreren Zweigen übertragen**. Will man jedoch jeden Code mit seinem eigenen Huffmann-Baum versehen, so ist der **Speicherbedarf des Huffmann-Baumes als Structure aber unverhältnismäßig hoch**, was man schon bei unserem Minibaum sieht.

```
>> whos('t')
  Name       Size              Bytes  Class     Attributes

  t          1x1                1064  struct
```

- Da wir es hier aber mit einem **speziellen binären Baum mit bekannter Anzahl von Knoten** zu tun haben, können wir den Huffmann-Baum **auch als Array** t erzeugen. Eigentlich **benötigt** man ja auch **nur die Indices der Zeichen**, und nicht die Zeichen selbst. Daher machen wir folgendes:
  - **t als Matrix mit 2 Spalten und n-1 echten Zeilen (und n gedachten Zeilen)**.
  - **Spalte 1** entspricht **Zweig 0**, und **Spalte 2** entspricht **Zweig 1**.
  - **Zeilenindex** ist auch **Index eines Knotens**.
  - In den **echten Zeilen 1 bis n-1** stehen die **neuen Knoten**, wobei **Knoten 1 die Wurzel** sein soll. In Spalte 1 steht Index des an Zweig 0 hängenden Knotens, und in Spalte 2 steht Index des an Zweig 1 hängenden Knotens.
  - In den **gedachten Zeilen n bis 2n-1** stehen die **Zeichen-Knoten/Blätter**, wobei dann **Zeichenindex=Zeilenindex-n+1** gilt.

Für unser erstes Beispiel erhalten wir

	Index	Spalte 1 (Zweig 0)	Spalte 2 (Zweig 1)
echte	1 (Wurzel)	7	2
Zeilen	2	6	3
von t	3 (n-1)	4	5
	(4) (n)	(Zeichen 1=K)	
gedachte	(5)	(Zeichen 2=A)	
Zeilen	(6)	(Zeichen 3=E)	
	(7) (2n-1)	(Zeichen 4=F)	

- **Automatisches Erzeugen eines Huffmann-Baumes: Version 2 mit Array t**
  - Es werden **nur die relativen Häufigkeiten als Vektor `w` übergeben**.
  - **Zunächst** erhalten die **neuen n-1 Knoten den Wert unendlich**, und die **gedachten n Knoten/Blätter die Häufigkeiten als Wert**.
  - In der **for-Schleife** werden die **Schritte (2) und (3) der Huffman-Baum-Konstruktion wiederholt**. Wir **laufen dabei rückwärts von n-1 bis 1**, so dass **am Ende die Wurzel in Zeile 1** steht.
  - **Abgearbeitete Knoten erhalten den Wert unendlich.**

```
function t=huffman_baum2(w)
n=numel(w);
t=zeros(n-1,2);
w=[inf*ones(n-1,1);w(:)];
for k=n-1:-1:1
    [w_min,ind]=mink(w,2);
    t(k,:)=ind;
    w(k)=w_min(1)+w_min(2);
    w(ind)=inf;
end
end
```

Wir erzeugen damit zum Test den Baum aus dem ersten Beispiel und sehen, dass der **Baum als Array wesentlich weniger Speicherplatz** benötigt. (Vorsicht: auch hier könnte der alternative Baum entstehen)

```
>> t2=huffman_baum2(w)
t2 =
     7     2
     6     3
     4     5
>> whos('t2')
  Name      Size            Bytes  Class     Attributes

  t2        3x2                48  double
```

- Wenn wir wissen, dass die **Indices nur kleine ganze Zahlen** sind (z.B. $2n - 1 \leq 255$), so können wir auch **entsprechende Datentypen verwenden**, z.B. `uint8` (**unsigned integer 8 bit, d.h. ganze Zahlen von 0 bis 255)**, und dadurch **noch mehr Speicherplatz sparen**.

```
>> t2=uint8(t2)
t2 =
  3x2 uint8 matrix
   7   2
   6   3
   4   5
>> whos('t2')
  Name      Size            Bytes  Class    Attributes

  t2        3x2                 6  uint8
```

- Schreiben wir noch die **entsprechende Funktion** `dekodiere2(c,t,z)` **zum Dekodieren mit t als Array**, wobei wir hier zusätzlich noch die Zeichen übergeben müssen, da diese selbst ja nicht mehr wie zuvor im Baum gespeichert sind, sondern nur noch ihre Indices. Nun gelangen wir **von einem neuen Knoten zum nächsten Knoten**, indem wir den **entsprechenden Index aus Spalte 1 bei Zweig 0 bzw. aus Spalte 2 bei Zweig 1** nehmen. **Ob** wir beim Durchlaufen des Baumes **an einem Zeichen/Blatt angekommen** sind, erkennen wir nun daran, dass ein **Index größergleich n** ist.

```
function s=dekodiere2(c,t,z)
n=numel(z);
m=numel(c);
c=1+c;
s=[];
i=0;
% durchlaufe Code bis Ende
while i<m
    i=i+1;
    % durchlaufe Baum
    k=1; % Wurzel
    folge_zweig(t(k,c(i)));
end
% eingenestete Funktion teilt s,c,i,t,z,n mit Hauptfunktion
    function folge_zweig(k)
        if k<n
            i=i+1;
            folge_zweig(t(k,c(i)));
        else
            % knoten ist Blatt/Zeichen
            s=[s,z(k-n+1)];
        end
    end
end
```

Test

```
>> s=dekodiere2(c,t2,z)
s =
    'AFFE'
```

- **Mit Huffman-Kodierung** lassen sich nicht nur Texte, sondern **auch Bilder komprimieren**. Daher gehen wir noch kurz auf Darstellung von Bildern ein.

15 Bilder

- Viele Farbbilder sind **RGB-Bilder**, bei denen die **Pixelfarbe durch 3 Intensitäts-Werte für Rot, Grün und Blau** angegeben wird. **In Matlab** lässt sich dies einfach durch **3-dimensionale** $M \times N \times 3$**-Arrays** realisieren, wobei die Pixel den Einträgen einer $M \times N$-Matrix mit 3 Farbwerten entsprechen. Die **Farbwerte** können **als Gleitpunkt-Zahlen** aus $[0, 1]$ angegeben werden (dann entspricht `[0,0,0]` schwarz und `[1,1,1]` weiß) **oder auch als ganzzahlige Werte**, z.B. `uint8` aus $[0, 255]$ (dann entspricht `[0,0,0]` schwarz und `[255,255,255]` weiß, und es gibt $256^3 = 16777216$ also **mehr als 16 Millionen verschiedene Farben (true color)**). Ein solches Array lässt sich dann mit `image` als Farbbild anschauen.

```
>> X=zeros(7,10,3);
>> x=linspace(0,1,10); % 10 wachsende Intensitäts-Werte aus [0,1]
>> X(1,:,1)=x; X(2,:,2)=x; X(3,:,3)=x; % erste Zeile nur Rot-Werte,
                          % zweite nur Grün-Werte, dritte nur Blau-Werte
>> X(4,:,[1,2])=[x;x]'; X(5,:,[1,3])=[x;x]'; X(6,:,[2,3])=[x;x]';
                          % jeweils 2 Farb-Werte gemischt
>> X(7,:,:)=[x;x;x]'; % alle 3 Farbwerte gleich ergibt Grauwerte
>> image(X);
>> X8=uint8(255*X)); image(X8) % als uint8-Bild (automatisch gerundet)
```

- **Bilder** lassen sich einfach **einlesen** mit `imread`.

```
>> imfinfo('trailer.jpg')  % uint8-Bild von Matlab
ans =
  struct with fields:
           ColorType: 'truecolor'
        CodingMethod: 'Huffman'
>> X=imread('trailer.jpg');
>> image(X)
>> R=X; R(:,:,[2,3])=0; image(R) % nur Rot-Werte
```

- Oftmals gibt es aber auch **Bilder mit Farbkarten (colormap; indexed image)**. Dies sind **in Matlab Matrizen**, bei denen **jeder Eintrag als Index in eine Farbkarte** dient. Solche Bilder brauchen oft **weniger Speicherplatz**. Die **Farbkarten** selbst sind einfach $K \times 3$**-Matrizen**, bei denen **in jeder Zeile ein RGB-Wert** steht. Wir keine spezielle Farbkarte angegeben, so wird eine Default-Farbkarte verwendet.

```
>> I=1:10;  % Bild mit Farb-Indices aus [1,10]
>> image(I); colorbar % Farbbalken und Default-Farbkarte
```

- Nun eine **eigene Farbkarte** aus Rot/Grün gemischt.

```
>> c=zeros(10,3); c(:,[1,2])=[x;x]'; colormap(c)
```

- **Vorsicht: Bei Farbkarten beginnen Indices als Gleitpunktzahlen wie gewohnt ab 1, aber bei ganzzahligen Datentypen wie uint8 ab 0.** (Das macht hier Sinn, damit ganzzahlige Werte einfach aus $[0, 255]$ übernommen werden können, ohne jedesmal eine 1 addieren zu müssen).

```
>> I=1:10; image(uint8(I)); colorbar; colormap(c)
>> I=0:9; image(uint8(I)); colorbar; colormap(c)
```

- In Matlab gibt es auch einige **vorgefertigte Farbkarten**, z.B.

```
>> c=winter(10); colormap(c) % 10 winterliche Werte
>> c=gray(10); colormap(c) % 10 Grau-Werte
```

- **Falls ein Bild nicht alle Farbwerte einer Farbkarte ausnutzt**, so kann man auch `imagesc` verwenden. Damit werden die **Werte** automatisch so **skaliert**, dass die Farbkarte ganz genutzt wird, d.h. kleinster (größter) Wert im Bild entspricht dann auch erstem (letztem) Farbwert der Farbkarte. Dies **erhöht den Kontrast**.

```
>> c=gray(50); colormap(c)
>> c=gray(50); imagesc(I); colorbar; colormap(c)
```

- Mit `rgb2gray` lassen sich **RGB-Bilder in Grauwertbilder umwandeln** (durch die Umrechnungsformel Grauwert=$0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$), und mit `ind2rgb` lassen sich **Bilder mit Farbkarte in RGB-Bilder umwandeln**.

```
>> Xg=rgb2gray(X); % Trailer-Bild von oben
>> [min(Xg(:)), max(Xg(:))]
ans =
  1x2 uint8 row vector
     0   255
>> image(Xg); colormap(gray(256))
>> colormap(summer(256))
>> Xrgb=ind2rgb(Xg,summer(256));
>> image(Xrgb)
```

- Auch **Bilder mit Farbkarte** lassen sich einfach **einlesen** mit `imread`.

```
>> info=imfinfo('trees.tif')  % Diese .tif-Datei enthält 8 Bilder
info =
  8x1 struct array with fields:
        ...
>> info(1)
ans =
  struct with fields:
                    ColorType: 'indexed'
                     Colormap: [256x3 double]
                MaxSampleValue: 255
                MinSampleValue: 0
>> [X,c]=imread('trees.tif');  % erstes Bild der .tif-Datei
>> image(X); colormap(c)
>> [X,c]=imread('trees.tif',2);  % zweites Bild
>> image(X); colormap(c)
```

- Bei RGB-und **Grau-Wert-Bildern** wird **keine Farbkarte mitgeliefert**.

```
>> imfinfo('coins.png')
ans =
  struct with fields:
                ColorType: 'grayscale'
>> [X,c]=imread('coins.png');
>> c
c =
     []
>> image(X); colormap(gray(256))
```

16 Haar Wavelet Transformation

- Wavelets werden vielfältig in der **Signal-und Bildverarbeitung** eingesetzt. Hier werden wir am Beispiel des Haar Wavelet Ideen zur **Kantenerkennung und verlustbehafteten Datenkompression** illustrieren (edge detection, lossy compression).

- Die **1D-HWT transformiert ein Signal gerader Länge** n

  $$x = (x_1, x_2, \ldots, x_n)$$

  in zwei Anteile

  $$x \mapsto (s|d)$$

  je halber Signallänge $n/2$

  $$s = (s_1, s_2, \ldots, s_{n/2}) \quad , \quad d = (d_1, d_2, \ldots, d_{n/2})$$

  wobei in s die **Summen** und in d die **Differenzen aufeinanderfolgender Einträge** in x stehen

  $$\begin{aligned} s_1 &= x_1 + x_2 \quad , \quad s_2 = x_3 + x_4 \quad \ldots \\ d_1 &= x_2 - x_1 \quad , \quad d_2 = x_4 - x_3 \quad \ldots \end{aligned}$$

  bzw. allg.

  $$\begin{aligned} s_k &= x_{2k} + x_{2k-1} \quad , \quad k = 1, \ldots, n/2 \\ d_k &= x_{2k} - x_{2k-1} \quad , \quad k = 1, \ldots, n/2 \end{aligned}$$

  Die folgende Funktion `HWT1D` liefert zu gegebenem x den transformierten Vektor

  $$y = (s|d) = (s_1, s_2, \ldots, s_{n/2}, d_1, d_2, \ldots, d_{n/2})$$

```
function y=HWT1D(x)
n=numel(x);
y=zeros(1,n);
for k=1:n/2
    y(k)=x(2*k)+x(2*k-1);
    y(n/2+k)=x(2*k)-x(2*k-1);
end
end
```

  zum Beispiel

```
>> x=[1 5 10 10 6 80 100 90];
>> y=HWT1D(x)
y =
     6    20    86   190     4     0    74   -10
>> s=y(1:4), d=y(5:8)
s =
     6    20    86   190
d =
     4     0    74   -10
```

- **Summen** haben einen **glättenden Effekt** und **Differenzen detektieren Änderungen/Kanten**,

```
>> figure(1);
>> subplot(2,1,1);plot(1:8,x,'o-',1.5:2:7.5,s/2,'x:')
>> subplot(2,1,2);stem(1.5:2:7.5,d)
```

- Die **Transformation ist umkehrbar**, denn

$$\begin{aligned}\frac{s_k+d_k}{2} &= \frac{(x_{2k}+x_{2k-1})+(x_{2k}-x_{2k-1})}{2}=x_{2k} \quad , \quad k=1,\dots,n/2\\ \frac{s_k-d_k}{2} &= \frac{(x_{2k}+x_{2k-1})-(x_{2k}-x_{2k-1})}{2}=x_{2k-1} \quad , \quad k=1,\dots,n/2\end{aligned}$$

Die folgende Funktion `IHWT1D` liefert zum transformierten Vektor $y=(s|d)$ den ursprünglichen Vektor x.

```
function x=IHWT1D(y)
n=numel(y);
x=zeros(1,n);
for k=1:n/2
    x(2*k)=(y(k)+y(k+n/2))/2;
    x(2*k-1)=(y(k)-y(k+n/2))/2;
end
end
```

Test:

```
>> IHWT1D(y)
ans =
     1     5    10    10     6    80   100    90
```

- **2D-HWT:** Die (Rück-)Transformation können wir **auf Matrizen (Bilder) nacheinander zeilen- und spaltenweise** anwenden. Dazu dienen folgende Funktionen `HWT2D` und `IHWT2D`.

```
function Y=HWT2D(X)
[m,n]=size(X);
Y=zeros(m,n);
for i=1:m
    Y(i,:)=HWT1D(X(i,:));
end
for j=1:n
    Y(:,j)=HWT1D(Y(:,j));
end
end

function X=IHWT2D(Y)
[m,n]=size(Y);
X=zeros(m,n);
for i=1:m
    X(i,:)=IHWT1D(Y(i,:));
end
for j=1:n
    X(:,j)=IHWT1D(X(:,j));
end
end
```

- Als **Beispiel** nehmen wir das in der Bildverarbeitung verbreitete **“Cameraman”-Bild**.

```
>> imfinfo('cameraman.tif')
ans =
  struct with fields:
                      Width: 256
                     Height: 256
                   BitDepth: 8
                  ColorType: 'grayscale'
  PhotometricInterpretation: 'BlackIsZero'
             MaxSampleValue: 255
             MinSampleValue: 0
>> I=imread('cameraman.tif');
>> c=gray(256);
>> figure(2); image(I); colormap(c)
```

- `I` ist ein **Grauwertbild mit uint8-Werten** im Bereich $[0, 255]$. Da bei der **HWT Differenzen** gebildet werden, können wir **nicht sinnvoll mit uint8-Werten rechnen**, deshalb machen wir **daraus double-Werte**.

```
>> z=uint8(1); [z-2,z+255]
ans =
  1x2 uint8 row vector
     0   255
>> X=double(I);
>> figure(2); image(X+1); colormap(c)
```

- Schauen wir uns die **HWT** an, wenn wir **zunächst nur die Zeilen transfomieren** (durch auskommentieren der zweiten `for`-Schleife in `HWT2D`). **Damit die negativen Werte bei den Differenzen nicht abgeschnitten werden, verwenden wir nun imagesc statt image.**

```
>> Y=HWT2D(X);
>> figure(3); imagesc(Y); colormap(c)
```

- In der **linken Hälfte ist nun eine gemittelte Version des Originalbildes** zu sehen und in der **rechten Hälfte Intensitäts-Sprünge**. Da wir zunächst nur horizontale Differenzen gebildet haben, sind die Sprünge besonders deutlich an den **vertikalen Kanten**.

- **dann Zeilen und Spalten transfomieren** (Kommentare weg und wiederholen).

```
>> Y=HWT2D(X);
>> figure(3); imagesc(Y); colormap(c)
```

  Im **linken unteren Viertel sind die Sprünge erwartungsgemäß besonders deutlich an horizontalen Kanten** zu sehen, da man hier vertikale Differenzen nimmt. Nicht ganz so deutlich hat man noch diagonale Sprünge im rechten unteren Viertel.

- **Testen** wir noch die **inverse Transformation**.

```
>> figure(4); imagesc(IHWT2D(Y)); colormap(c)
```

- **Kantendetektion:**
  Wenn wir **zuerst das gemittelte Bild auf Null setzen** und **dann die inverse Transformation** anwenden, erhält man ein **Bild der Sprünge (horizontal, vertikal und diagonal zusammen)**. **Hier interessiert uns nur der Betrag der Sprünge** (und nicht das Vorzeichen).

```
>> Z=Y; Z(1:128,1:128)=0; D=abs(IHWT2D(Z));
>> figure(4); imagesc(D); colormap(c)
```

  Um eine **(logische) Maske der Kanten** zu erhalten, geben wir uns einen **Threshold** $\tau > 0$ **für die Stärke des Sprunges** vor, und setzen alle Einträge $> \tau$ auf 1 (und alle Einträge $\leq \tau$ auf 0).

```
>> Dmax=max(D(:)); tau=0.3*Dmax; K=D>tau;
>> figure(4); image(K); colormap(gray(2)) % Kanten weiß
>> figure(4); image(~K); colormap(gray(2)) % Kanten schwarz
>> Dmax=max(D(:)); tau=0.1*Dmax; K=D>tau;
>> figure(4); image(~K); colormap(gray(2))
```

- **(Verlustbehaftete) Datenkompression:**
  Wenn wir Daten automatisch komprimieren wollen, brauchen wir ein **Maß, das uns sagt, welche Daten wichtiger bzw. nicht so wichtig sind**, so dass wir die unwichtigen vernachlässigen können. Dazu betrachten wir einen Vektor (Signal,Bild)

$$x = (x_1, x_2, \ldots, x_n)$$

  Diesen **sortieren wir vom betragsmäßig größten zum betragsmäßig kleinsten Element**.

$$x \mapsto v = (v_1, v_2, \ldots, v_n) \quad , \quad v_1 \geq v_2 \geq \ldots \geq v_n \geq 0$$

  bilden die euklidische Norm

$$\|v\|^2 = \sum_{k=1}^{n} v_k^2 \quad \left(= \|x\|^2 \right)$$

  und setzen $e_1 = \frac{v_1^2}{\|v\|^2}$, $e_2 = \frac{v_1^2+v_2^2}{\|v\|^2}, \ldots$, d.h.

$$e_j = \frac{\sum_{k=1}^{j} v_k^2}{\|v\|^2} \quad , \quad j = 1, \ldots, n$$

  Den Vektor $e = (e_1, \ldots, e_n)$ bezeichnet man als die **kumulierte Energie** von x. Der **Eintrag** e_j **gibt an, wieviel Prozent der Gesamtenergie in den ersten** j **betragsmäßig größten Einträgen von** x **konzentriert sind**. Typischerweise verwendet man die kumulierte Energie folgendermaßen zur **Datenkompression**:

  – Gib vor, **wieviel Prozent der Gesamtenergie** behalten werden soll.
  – **Identifiziere** die **Einträge in** x**, die diesen Anteil erzeugen**.
  – Setze die **übrigen Einträge auf Null** $\rightsquigarrow$ diese müssen dann (im Prinzip) nicht gespeichert werden, bzw. die Null taucht bei Huffman-Kodierung sehr häufig auf.

- **In den Übungen schreiben wir dazu eine Funktion** `[e,ind]=energie(x)`, welche sowohl die kumulierte Energie `e` als auch den Indexvektor `ind` der Sortierung liefert, so dass gilt `v=abs(x(ind))` (ist auch für Matrizen mit linearer Indizierung geeignet), z.B.

```
>> x=[1 -4 0 2];
>> [v,ind]=sort(abs(x),'descend')
v =
     4     2     1     0
```

```
ind =
     2     4     1     3
>> [e,ind]=energie(x)
e =
    0.7619    0.9524    1.0000    1.0000
ind =
     2     4     1     3
```

- **Vergleichen wir die kumulierte Energie des Originals mit dem HW-transformierten Bild**,

```
>> [e,ind]=energie(X);
>> [eHWT,indHWT]=energie(Y);
>> figure(5);n=numel(e);plot(1:n,e,1:n,eHWT);legend('original','HWT')
>> eHWT(15000)
ans =
    0.9957
```

so sehen wir, dass die **ersten 15000 betragsgrößten Werte des transformierten Bildes** `Y` **schon über 99% der Gesamtenergie** ausmachen.

- Die **restlichen Einträge** können wir **auf Null setzen und** erhalten **nach Rücktransformation ein immer noch gutes Bild**.

```
>> [~,indHWT]=sort(abs(Y(:)),'descend'); % wieder zum mitmachen
>> Y2=Y; Y2(indHWT(15001:end))=0; X2=IHWT2D(Y2);
>> figure(6); imagesc(X2); colormap(c)
```

- Da wir **dadurch nun eventuell auch Werte außerhalb unseres ursprünglichen Bereiches** erhalten, **passen wir die Werte noch an**, und machen Sie **auch wieder ganzzahlig**.

```
>> xmin=min(X(:)),xmax=max(X(:))
xmin =
     7
xmax =
   253
>> x2min=min(X2(:)),x2max=max(X2(:))
x2min =
   -9.2500
x2max =
  267.5000
>> X3=uint8(xmin+(xmax-xmin)*(X2-x2min)/(x2max-x2min));
>> figure(7); image(X3); colormap(c)
```

- Vergleichen wir noch den **Speicherbedarf mit Huffman-Codierung der transformierten Bilder mit und ohne Nullsetzen**.

```
>> ymin=min(Y(:)),ymax=max(Y(:))
ymin =
  -406
ymax =
        1002
>> z=ymin:ymax; % Zeichen
>> w=haeufigkeit(Y,z); % aus Übung
>> t=huffman_baum(z,w); % Version 1 mit Structure
>> cw=codewort(t); % aus Übung
>> yc=kodiere(Y,cw); % aus Übung
```

```
>> w2=haeufigkeit(Y2,z);
>> t2=huffman_baum(z,w2);
>> cw2=codewort(t2);
>> y2c=kodiere(Y2,cw2);
>> ybits=numel(yc), y2bits=numel(y2c)
ybits =
      469577
y2bits =
      203880
>> y2bits/ybits
ans =
    0.4342
```

Mit Nullsetzen werden also **nur ca. 40% des Speicherplatzes ohne Nullsetzen benötigt** (und **ca. 45% des Speicherplatzes des ursprünglichen Bildes mit Huffman-Kodierung**).

- Erklärung: **Betragsmäßig kleine Einträge** haben **beim Original-Bild und HW-transformierten Bild** eine **unterschiedliche Bedeutung**.
  - Beim **Original-Bild** sind es die **Pixel mit geringer Intensität**, also **fast schwarz**.
  - Beim **HW-transformiertem Bild** sind es einerseits intensitätsschwache Mittelwerte, und andererseits **bei den Differenzen kleine Sprünge**. Setzt man letztere auf Null, **vernachlässigt man kleine Details im Bild**, die aber **für das Auge nicht so wichtig** sind.
- **K-fache HWT:**
  Auf die **Summen** (linkes oberes Viertel) **des transformierten Bildes** `Y` können wir **nochmal die HWT** anwenden.

```
>> Y3=HWT2D(Y(1:128,1:128));
>> figure(7); imagesc(Y3); colormap(c)
```

- Allerdings haben wir dann die Differenzdaten aus der ersten Transformation vernachlässigt. Deshalb plotten wir alles in **ein Bild mit allen Transformationsdaten** und erhalten so das typische Bild der **2-fachen HWT**.

```
>> figure(8)
>> imagesc([Y3,Y(1:128,129:end);Y(129:end,1:128),Y(129:end,129:end)])
>> colormap(c)
```

- Das können wir **mehrmals wiederholen**...und erhalten die **K-fache HWT**. Damit auch diese invertierbar bleibt, müssen die **Differenzdaten aus allen Schritten beibehalten** werden. Für ein Ausgangssignal x der Länge 8 erhält man z.B.

$$\begin{array}{ll} & (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \\ K = 1: & (s_1^1, s_2^1, s_3^1, s_4^1 \,|\, d_1^1, d_2^1, d_3^1, d_4^1) \\ K = 2: & (s_1^2, s_2^2 \,|\, d_1^2, d_2^2 \,|\, d_1^1, d_2^1, d_3^1, d_4^1) \\ K = 3: & (s_1^3 \,|\, d_1^3 \,|\, d_1^2, d_2^2 \,|\, d_1^1, d_2^1, d_3^1, d_4^1) \end{array}$$

d.h. man wendet die **1D-HWT nacheinander an auf ganzen Vektor, linke Hälfte, linkes Viertel,...**
Bei einer Matrix wendet man die **2D-HWT nacheinander an auf ganze Matrix, linkes oberes Viertel, linkes oberes 16-tel,...**
Manchmal erhält man mit der K-fachen HWT (für kleine K=2,3) noch bessere Kompressionsraten.

17 Graphische Objekte

- **Figures, Axes, geplottete Linien etc. sind graphische Objekte mit Eigenschaften, die man lesen und verändern kann.** Die Namen der Eigenschaften folgen dem Objektnamen nach einem Punkt (analog zu Structures).

```
>> f1=figure(1)
f1 =
  Figure (1) with properties:
      Number: 1
       Color: [0.9400 0.9400 0.9400]
    Position: [680 558 560 420]
       Units: 'pixels'

  Show all properties
```

- Die **Farbe ist hier in RGB-Werten** angegeben. Wir **ändern die Farbe zu rot**, und **holen die Figure** mit `figure(f1)` **in den Vordergrund**.

```
>> f1.Color=[1,0,0]
>> figure(f1)
```

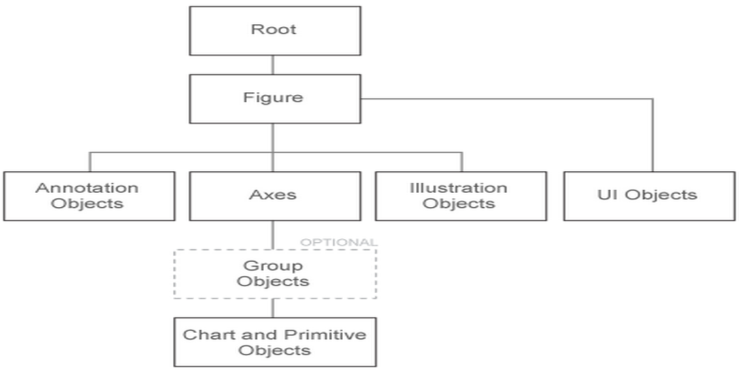
- **Alle Eigenschaften** sehen wir, wenn wir **auf “Show all properties” klicken** oder mit `get(f1)`

```
  Show all properties
                     Color: [1 0 0]
               CurrentAxes: [0x0 GraphicsPlaceholder]
                    Parent: [1x1 Root]
                    Resize: 'on'
                  ...
```

- **Eigenschafte Resize** legt fest, ob man die **Größe der Figure ändern kann**.

```
>> f1.Resize='off';
>> f1.Resize='on';
```

- Alle **graphischen Objekte** sind **hierarchisch angeordnet**:



- **Jede Figure hängt am Root-Objekt**, wie man unter der **Eigenschaft Parent** sieht.

```
>> r=f1.Parent
r =
  Graphics Root with properties:
          CurrentFigure: [1x1 Figure]
             ScreenSize: [1 1 1920 1080]
       MonitorPositions: [2x4 double]
                  Units: 'pixels'
```

- Die aktuelle Bilschirmgröße 1920 mal 1080 und **Position der Figure sind in 'pixels'-Einheiten** angegeben mit Ursprung [1,1] links unten und [x Pixel nach rechts, y Pixel nach oben, Breite, Höhe]. **Oft ist es bequemer in relativen Einheiten zu rechnen** mit Ursprung [0,0] links unten, [1,1] rechts oben und [x nach rechts, y nach oben, Breite, Höhe].

```
>> f1.Units='normalized';
>> f1.Position=[1/2, 1/2, 1/4, 1/8];
```

- **Manche Eigenschaften** sind allerdings **nur zum Lesen und können nicht verändert werden**.

```
>> r.ScreenSize=[1 1 800 600]
You cannot set the read-only property 'ScreenSize' of Root.
```

- **Alle graphischen Objekte sind Handle-Objekte**, so dass **bei Zuweisung** einer Variablen kein neues Objekt erzeugt wird, sondern **nur ein Handle auf das selbe Objekt übergeben** wird (im Unterschied zu den bisherigen Datentypen; und es muss auch kein `@`-Symbol wie bei Function Handles verwendet werden).

```
>> f2=f1;
>> f2.Color=[0 0 0];
>> f1.Color
ans =
     0     0     0
```

- Wenn wir eine **Figure schließen**, werden **nicht automatisch auch die Handle-Variablen gelöscht**, sie zeigen aber nur noch auf eine gelöschte Figure. Die Handle-Variablen können wir dann mit `clear` löschen.

```
>> close(f2)
>> f1
f1 =
  handle to deleted Figure
>> clear('f1','f2')
```

- Erzeugen wir **2 neue Figures** und setzen **in f1 eine Axes zum Plotten** rein.

```
>> f1=figure(1); f2=figure(2);
>> a1=axes(f1)
a1 =
  Axes with properties:
             XLim: [0 1]
             YLim: [0 1]
```

- Die **aktuelle (zuletzt verwendete) Figure/Axes** erhält man mit **gcf/gca**

```
>> fc=gcf  % get current figure
fc =
  Figure (2) with properties:
>> ac=gca  % get current axes
ac =
  Axes with properties:
             XLim: [0 1]
             YLim: [0 1]
```

- Nun **ändern wir den Plotbereich** auf x-und y-Achse,

```
>> a1.XLim=[-4,4]; a1.YLim=[-2,2];
```

- **machen Koordinatenachsen so, dass Mittelpunkt im Ursprung**,

```
>> a1.XAxisLocation='origin'; a1.YAxisLocation='origin';
```

- **ändern die Ticks der x-Achse**,

```
>> a1.XTick=[-pi,-pi/2,0,pi/2,pi];
```

- **wählen eigene Beschriftung der Ticks mit Latex und ändern Schriftgröße**,

```
>> a1.TickLabelInterpreter='Latex';
>> a1.XTickLabel={'$-\pi$','$-\frac{\pi}{2}$',...
       '$0$\qquad','$\frac{\pi}{2}$','$\pi$'};
>> a1.FontSize=20;
```

- **plotten ein Line-Objekt in die Axes a1**,

```
>> x=linspace(-pi,pi,100);
>> l1=line(a1,'XData',x,'YData',sin(x)) % in Axes a1
l1 =
  Line with properties:
              Color: [0 0 0]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
              XData: [1x100 double]
              YData: [1x100 double]
              ZData: [1x0 double]
```

- **ändern das Aussehen des Line-Objekts**,

```
>> l1.Color='g'; l1.LineWidth=3; l1.LineStyle=':';
```

- und **kopieren unsere Axes** `a1` **mitsamt Kindern (hier Line-Objekt) in andere Figure** `f2`.

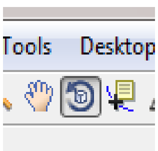
```
>> a2=copyobj(a1,f2);
```

- **Mit copyobj** werden tatsächlich **neue Objekte mit gleichen Eigenschaften** erzeugt, und **nicht nur die Handle übergeben**.

```
>> l1.YData=cos(x);
>> l2=a2.Children;
>> l2.XData=cos(x);
>> a2.DataAspectRatio=[1 1 1]; % Gleiche Längenverhältnisse der Achsen,
                               % damit hier der Kreis auch wie einer aussieht
```

- Nach **Drücken des Rotate-Buttons in Figure 2** und Bewegen der Mouse sehen wir, dass eigentlich **alles 3D Objekte** sind. Ändern wir also noch die **Z-Komponenten unseres Line-Objektes**.

```
>> l2.ZData=0.5*sin(6*x);
```



- Setzen wir **in Figure f1** noch eine **Textbox als Annotation-Objekt**. Dabei soll die **Schrift mit der Größe der Figure größer oder kleiner** werden.

```
>> t1=annotation(f1,'textbox')
t1 =
  TextBox with properties:
           FontSize: 10
          FontUnits: 'points'
             String: {''}
           Position: [0.3000 0.3000 0.1000 0.1000]   % bzgl. Figure
              Units: 'normalized'
>> t1.String={'Zeile␣1','Zeile␣2'};
>> t1.FontUnits='normalized'; t1.FontSize=0.1;
>> t1.Position=[0,1,0.2,-0];  % Linke obere Ecke der Figure (ohne Menüleiste)
>> t1.FitBoxToText='on';
```

- Und **ein Text-Objekt in Axes a1**.

```
>> t2=text(a1,pi,0.5,'hier')
t2 =
  Text (hier) with properties:
              Position: [3.1416 0.5000 0]
                 Units: 'data'  % Position wie Datenpunkte
>> t2.Rotation=-45;  % Winkel in Grad
```

- Und **löschen beide wieder**.

```
>> delete([t1,t2])  % lösche Textbox und Text
>> t1  % Handle-Variablen existieren noch
t1 =
  handle to deleted TextBox
>> clear('t1','t2')  % Variablen löschen
```

- Bei den **Matlab-Funktionen zum Plotten** kann man sich auch die **Handle der Objekte zurückgeben lassen.**

```
>> a2.NextPlot='add';  % wie hold('on')
>> l3=plot3(a2,cos(3*x),sin(3*x),x/pi/2);
>> l3.Visible='off';
>> l3.Visible='on';
```

- Das **direkte Ändern der Eigenschaften** wie YData etc. ist aber **effizienter als** jedesmal den **plot-Befehl** zu verwenden, da dieser viele Aufgaben auf einmal übernimmt. So können wir auch einfach **kleine Filme erzeugen**, indem wir zunächst mit `getframe` eine Bildfolge speichern (alles in eine Zeile, oder als Skript im Editor),

```
>> figure(f1); p1=f1.Position;    % Position und Größe merken
   for t=1:10 l1.YData=cos(x-pi/5*t); M(t)=getframe(f1); end
```

und dann mit `movie` z.B. **dreimal mit 10 fps (frames per second) abspielen**.

```
>> f3=figure(3); f3.Position=p1; movie(f3,M,3,10)
```

- Den **Film** können wir auch **als Video-Datei speichern**, indem wir zuerst ein **VideoWriter-Objekt erzeugen, öffnen, beschreiben und schließen**, und uns dann **außerhalb von Matlab die Video-Datei anschauen**.

```
>> v=VideoWriter('testfilm.avi')
v =
  VideoWriter
  ...
>> v.FrameRate=10;
>> open(v)
>> writeVideo(v,M)
>> close(v)
```

- Setzen wir **in Axes a1** noch ein **Patch-Objekt**,

```
>> p1=patch(a1,'Vertices',[0 0;1 0;0 1;1 1],'Faces',[1 2 4 3])
p1 =
  Patch with properties:
    FaceColor: [0 0 0]
    FaceAlpha: 1
    ButtonDownFcn: ''
    UserData: []
>> p1.FaceColor=[1 0 0]; p1.FaceAlpha=0.3; % rot, transparent
```

- und schreiben **für die ButtonDownFcn eine Callback-Funktion**, mit der wir **per Mausklick die Farbe ändern** können. Die **Callback-Funktionen** müssen hier **immer als erste Eingabevariable das auslösende Objekt** und als **zweite Eingabevariable eine Ereignis-Variable** besitzen (und **optional weitere Eingabevariablen**).
Den **aktuellen Farb-Zustand speichern wir unter der Eigenschaft UserData, die alle graphischen Objekte besitzen**.

```
function patch_mausklick(p,ereignis)
% ereignis wird zwar hier nicht benötigt,
% Matlab verlangt aber diese Syntax für Callbacks
if p.UserData % hier logical
    p.FaceColor=[1 0 0]; % rot
    p.UserData=false;
else
    p.FaceColor=[0 0 1]; % blau
    p.UserData=true;
end
end
```

Test:

```
>> p1.UserData=false;
>> p1.ButtonDownFcn=@(p,e) patch_mausklick(p,e);
   % wird bei Mausklick auf Patch aufgerufen
```

- Schreiben wir **noch eine Callback-Funktion**, mit der wir **das Patch-Objekt nach Anklicken mit der Maus bewegen** können. Dazu benötigen wir sowohl die **ButtonDownFcn des Patch-Objektes** als auch die **WindowButtonMotionFcn der Figure**, da diese die Mausbewegung abfragt. Die **zusätzlichen Eingabevariablen für die WindowButtonMotionFcn** können gemeinsam **mit dem Function-Handle in einem Cell-Array übergeben** werden. Als **aktuelle Mausposition** verwenden wir die **CurrentPoint-Eigenschaft der Axes**. Da **bei Überlappung**

mehrerer Patches dasjenige ausgewählt wird, welches im Children-Array der Axes a1 an Position 1 steht, schreiben wir den Callback so, dass das **angeklickte Patch** nun auch **an die erste Position des Children-Arrays** gesetzt wird.
(Bemerkung: Die nicht benötigten Eingabevariablen empfiehlt Matlab durch eine Tilde zu ersetzen).

```
function patch_mausklick2(p,~)
a=p.Parent;
f=a.Parent;
if p.UserData
    f.WindowButtonMotionFcn='';
    p.UserData=false;
else
    f.WindowButtonMotionFcn={@(f,e,a,p) wbmf(f,e,a,p),a,p};
    p.UserData=true;
    % mache p zum ersten Kandidat für Mausklick bei Überlappung
    c=a.Children;
    for k=1:numel(c)
       if p==c(k) % zeigen Handle auf selbes Objekt?
           a.Children([1,k])=c([k,1]);
           break;
       end
    end
end
end
% Unterfunktion
function wbmf(~,~,a,p)
maus_position=a.CurrentPoint(1,1:2);
% nun linke untere Ecke auf Spitze des Mauszeigers
p.Vertices(:,1)=p.Vertices(:,1)+(maus_position(1)-p.Vertices(1,1));
p.Vertices(:,2)=p.Vertices(:,2)+(maus_position(2)-p.Vertices(1,2));
drawnow; % versuche direkt zu zeichnen
end
```

Test:

```
>> p1.UserData=false;
>> p1.ButtonDownFcn=@(p,e) patch_mausklick2(p,e);
>> a1.SortMethod  % Bestimmt Reihenfolge der Objekte im Vordergrund
ans =
    'childorder'  % Standard in 2D-Ansicht
```

- **Beim Kopieren werden die Callback-Funktionen nicht mit kopiert.**

```
>> p2=copyobj(p1,a1);
>> p2.FaceColor=[0 1 0];
>> p2.UserData=false;
>> p2.ButtonDownFcn=@(p,e) patch_mausklick2(p,e);
```

- Man kann auch einfach **beliebig berandete Patch-Objekte** erzeugen, **auch in 3D**.

```
>> p3=patch(a1,'XData',cos(x),'YData',sin(x))
p3 =
  Patch with properties:
        Faces: [1x100 double] % in eingegebener Reihenfolge wie X/YData
     Vertices: [100x2 double] % ebenso
>> p3.ZData=x; p3.FaceAlpha=0.1;
```

- **Beliebig gefärbte Flächen (auch mit Texturen)** könnte man durch viele kleine Patch-Objekte erzeugen. Einfacher geht es **mit Surface-Objekten**. Erzeugen wir **einen rechteckigen Parameterbereich mit meshgrid, zunächst wenig Gitterpunkte zum Verdeutlichen.**

```
>> f2=figure; % ohne Angabe einer Nummer wird nächste Figure geöffnet
>> a2=axes(f2); a2.DataAspectRatio=[1 1 1];
>> x=linspace(-1,1,3); y=linspace(-1,1,5); % x-,y-Koordinaten der Gitterpunkte
>> [X,Y]=meshgrid(x,y) % Erzeuge alle Gitterpunkte (alle Kombinationen)
X =
    -1      0      1
    -1      0      1
    -1      0      1
    -1      0      1
    -1      0      1
Y =
   -1.0000   -1.0000   -1.0000
   -0.5000   -0.5000   -0.5000
         0         0         0
    0.5000    0.5000    0.5000
    1.0000    1.0000    1.0000
>> Z=X.^2+Y.^2; s1=surface(a2,X,Y,Z) % X,Y,Z haben gleiche Dimension
s1 =
  Surface with properties: ...
```

- Nun **mehr Gitterpunkte und Wechsel der Colormap**.

```
>> delete(s1);
>> x=linspace(-1,1,20); y=linspace(-1,1,20); [X,Y]=meshgrid(x,y);
>> Z=X.^2+Y.^2; s1=surface(a2,X,Y,Z);
>> f2.Colormap=gray(7);
```

- **In CData stehen die Indices in die Colormap.** Standardmäßig gilt CData=ZData. Dies können wir aber ändern.

```
>> s1.CData=rand(size(s1.ZData)); % gleiche Größe wie ZData
```

- Wir wählen eine **einheitliche Farbe** und setzen ein **Licht-Objekt in die Axes**. Standardmäßig sitzt die **Lichtquelle im Unendlichen** und **sendet parallele Strahlen aus Richtung Position**.

```
>> s1.FaceColor='r';
>> li=light(a2)
li =
  Light with properties:
       Color: [1 1 1]
       Style: 'infinite'
    Position: [1 0 1]
```

- Nun setzten wir die **Lichtquelle lokal an eine Position, von der Sie in alle Richtungen strahlt**. Wenn wir Sie genau **in den Brennpunkt des Paraboloids** setzen, sehen wir den Effekt eines **Parabolspiegels/-Scheinwerfers**. Wir **ändern** auch die **Beleuchtungsart**, damit die **einzelnen Facetten nicht gleichmäßig beleuchtet** werden (schöner für gekrümmte Objekte).

```
>> li.Style='local';
>> li.Position=[0,0,1];
>> s1.EdgeColor='none';
>> s1.FaceLighting='gouraud';
>> li.Position=[0,0,1/4]; % Brennpunkt
```

- **Mit einer Texturemap** (diese muss nicht die gleiche Dimension wie ZData haben). Da Bilddaten als Matrix/Array in Matlab in der x-y-Ebene mit `image` standardmäßig mit Zeilen von oben nach unten geplottet werden, die y-Koordinate hier aber von “unten” nach “oben” wächst, vertauschen wir noch die Zeilen des Bildes.

```
>> C=imread('trailer.jpg');
>> s1.FaceColor='texturemap'; s1.CData=C(end:-1:1,:,:);
   % mit vertauschten Zeilen, damit Bild nicht auf dem Kopf steht
```

- Parametrisierung in **Zylinderkoordinaten**,

```
>> s1.EdgeColor='k';
>> phi=linspace(0,3/2*pi,50); z=linspace(0,2,5); [Phi,Z]=meshgrid(phi,z);
>> s1.XData=cos(Phi); s1.YData=sin(Phi); s1.ZData=Z;
```

- und **Kugelkoordinaten**.

```
>> phi=linspace(0,3/2*pi,50); theta=linspace(pi,pi/4,20); % von unten nach oben
    [Phi,Theta]=meshgrid(phi,theta);
>> s1.XData=cos(Phi).*sin(Theta); s1.YData=sin(Phi).*sin(Theta);
    s1.ZData=cos(Theta);
```

- In **3D-Ansicht** ist die **SortMethod der Axes standardmäßig depth** (vorne ist, was näher zur Kamera).

```
>> a2.SortMethod
ans =
    'depth'
```

- **Verschiedene graphische Objekte** lassen sich in einem **graphics-Array** zusammenfassen. Die **Speicherreservierung geht mit gobjects**.

```
>> ga=[f2,a2,s1]
ga =
  1x3 graphics array:

    Figure     Axes       Surface
>> ga(3).FaceColor='none'; % nur Gitterlinien
ga=gobjects(2,2)
ga =
  2x2 GraphicsPlaceholder array:

    GraphicsPlaceholder    GraphicsPlaceholder
    GraphicsPlaceholder    GraphicsPlaceholder
```

18 User Interface Objekte

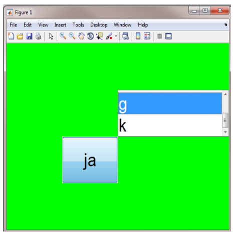
- Graphische Objekte und ihre Callbacks kann man auch gut zum **Programmieren benutzerfreundlicher Software** verwenden. Um nicht alles selbst machen zu müssen, gibt es **in Matlab schon einige vorgefertigte UI-Objekte** (User-Interface).

- Zum **Beispiel ein Toggle-Button**, der bei jedem Anklicken zwischen 2 Zuständen wechselt. Der **aktuelle Zustand 0 oder 1 steht in Eigenschaft Value**.

```
>> f=figure;
>> u1=uicontrol(f,'Style','togglebutton')
u1 =
  UIControl with properties:
            Style: 'togglebutton'
           String: ''
         Callback: ''
            Value: 0
            Units: 'pixels'
>> u1.String='nein'; u1.FontSize=30;
>> u1.Units='normalized'; u1.Position=[1/4,1/4,1/4,1/4];
```

- Dazu schreiben wir noch einen **passenden Callback**.

```
function toggle(u,~)
if u.Value
    u.String='ja';
else
    u.String='nein';
end
end
```



Test:

```
>> u1.Callback=@(u,e) toggle(u,e);
```

- Noch ein **Beispiel: Listbox, bei der man Einträge einer Liste auswählen kann**. Die **Liste besteht aus Cell-Array von Strings, und hier ist Value der Index des Eintrags**.

```
>> u2=uicontrol(f,'Style','listbox')
u2 =
  UIControl with properties:
            Style: 'listbox'
           String: ''
         Callback: ''
            Value: 1
>> u2.String={'r','b','g','k'}; u2.FontSize=30;
>> u2.Units='normalized'; u2.Position=[1/2,1/2,1/2,1/4];
```

- Dazu schreiben wir noch einen **passenden Callback**.

```
function listbox(u,~)
f=u.Parent;
f.Color=u.String{u.Value};
end
```

Test:

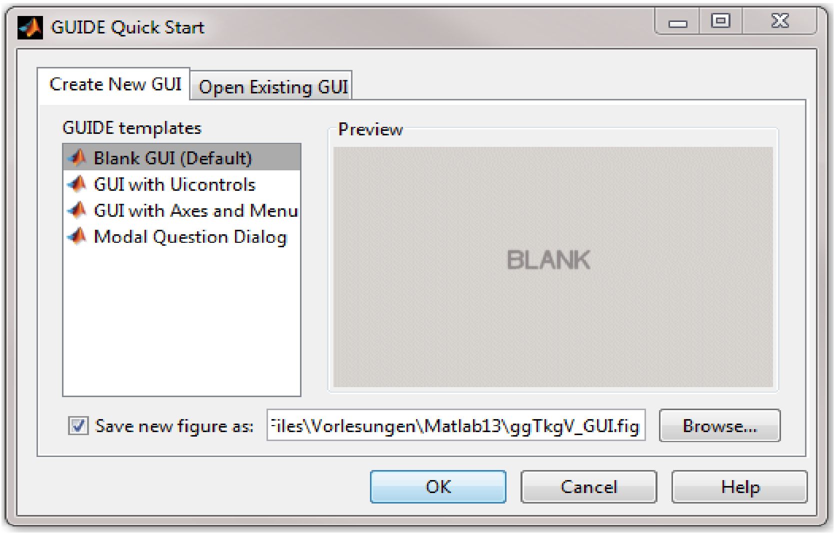
```
>> u2.Callback=@(u,e) listbox(u,e);
```

19 GUI’s mit GUIDE

- **Mit Hilfe von GUIDE** lassen sich in Matlab auch **interaktiv graphische Benutzeroberflächen (GUI)** erstellen.

- Hier wollen wir **beispielhaft eine GUI zum Berechnen des größten gemeinsamen Teilers (ggT) und kleinsten gemeinsamen Vielfachen (kgV) zweier Zahlen** erstellen.

- **Gestartet** wird GUIDE durch Anklicken von **New -> App -> GUIDE** in der Symbolleiste oder durch den Befehl `guide` im Command Window. Für eine neue GUI dann **Blank GUI** wählen und **unter gewünschtem Dateinamen** `ggTkgV_GUI.fig` **speichern**. Dabei werden dann **automatisch ein .fig-File und ein .m-File** gespeichert.

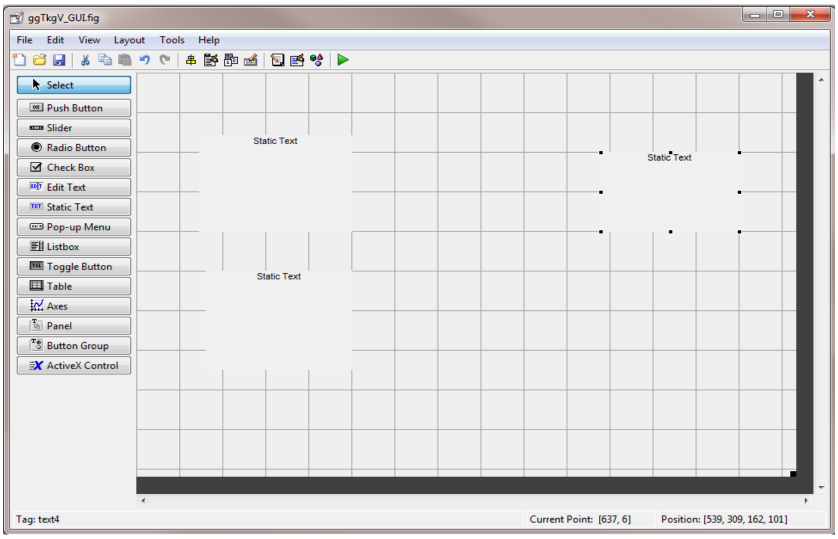


- **Im .m-File** wird der **Programm-Code** der GUI erzeugt, den wir **nach unseren Wünschen ergänzen** können. Unser .m-File beginnt hier mit

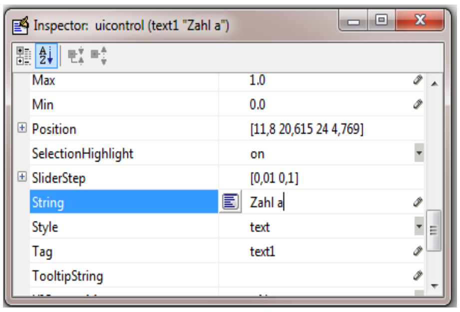
```
function varargout = ggTkgV_GUI(varargin)
% GGTKGV_GUI MATLAB code for ggTkgV_GUI.fig
...
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @ggTkgV_GUI_OpeningFcn, ...
                   'gui_OutputFcn',  @ggTkgV_GUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
...
```

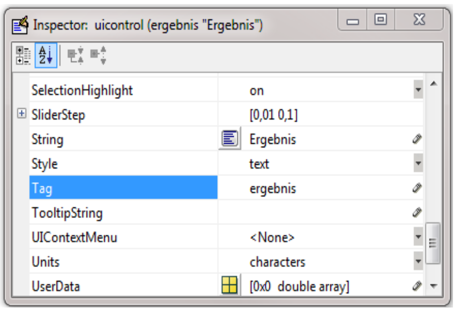
- Die **GUI** können wir **interaktiv gestalten durch Einsetzen der Komponenten/Objekte wie Text Box, Push Button, Slider usw. in den Gitterbereich**. Größe und Position der Objekte lassen sich beliebig verändern. Die **Größe des Gitterbereiches wird auch die Größe unserer GUI-Figure** sein und lässt sich **durch Ziehen des schwarzen Kastens in der rechten unteren Ecke verändern**.

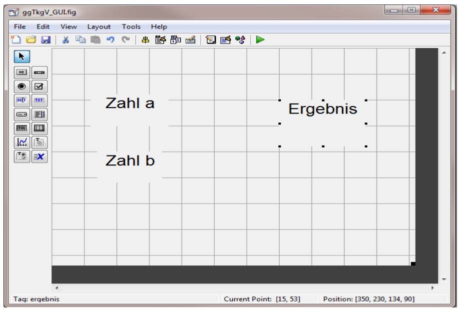
- Hier setzen wir **3 Text Boxen** (Icon TXT-Static Text auswählen).



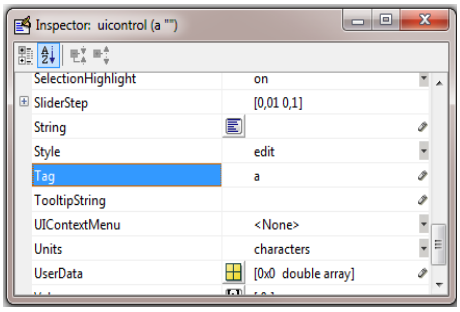
- **Doppelklick auf ein Objekt öffnet den Property Inspector**. Darin sieht man **alle Eigenschaften des Objekts**, z.B. FontSize, String,... und kann diese verändern. **Der Name des Objekts ist unter Eigenschaft Tag**. Unter diesem Namen erkennt man das Objekt dann **auch im .m-File**. Hier ändern wir **bei allen Textboxen** die **FontSize auf 20**, den **jeweiligen String zu Zahl a, Zahl b bzw. Ergebnis** (und passen gegebenenfalls die Größe unserer Objekte an), und **bei der letzten auch den Tag zu ergebnis**.

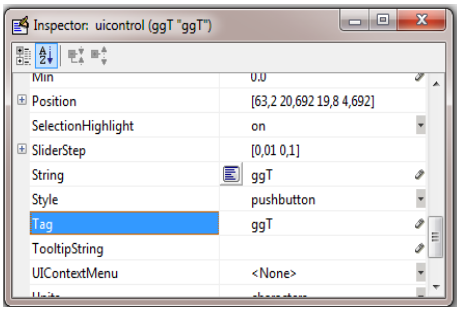


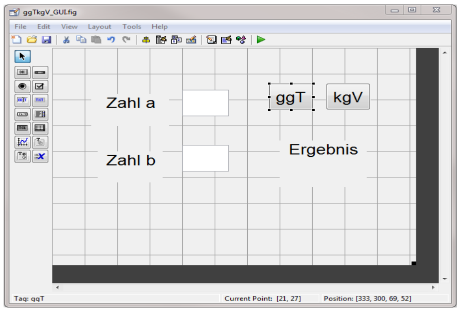




- Wir ergänzen unsere GUI durch **2 Eingabeboxen (EDIT-Edit Text)** und **2 Push Buttons (OK)** mit **FontSize 20, jeweiligem Tag a, b, ggT, kgV und String blank, blank, ggT, kgV**. Dabei können wir ein **Objekt kopieren durch rechter Mausklick auf Objekt -> copy**.







- **Unsere GUI soll folgendes leisten**: Man kann **in die Felder neben Zahl a und Zahl b jeweils eine natürliche Zahl eingeben und bei Drücken des Push Buttons ggT bzw. kgV wird dann das entsprechende Ergebnis berechnet und in der Text Box das Ergebnis ausgegeben**. Um **auf ein Ereignis reagieren** zu können (z.B. “Drücken eines Push Buttons” oder “Bestätigen der Eingabe in Eingabefeld”), kann man die entsprechende **Callback-Funktion des Objekts im .m-file** abändern. Diese wird ins .m-File als Unterfunktion eingefügt durch **rechter Mausklick auf Objekt -> View Callbacks -> Callback**. In unserem .m-File also z.B.

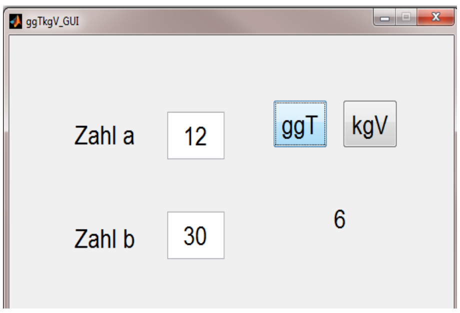
```
...
% --- Executes on button press in ggT.
function ggT_Callback(hObject, eventdata, handles)
% hObject    handle to ggT (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
...
```

- Die **Structure handles** im .m-File **enthält alle Objekte mit Feldname=Tag**. Zusätzlich ist **hObject auch das aktuelle auslösende Objekt des Callbacks**.

- Wir schreiben nun die **Callbacks für die ggT-und kgV-Push Buttons**.

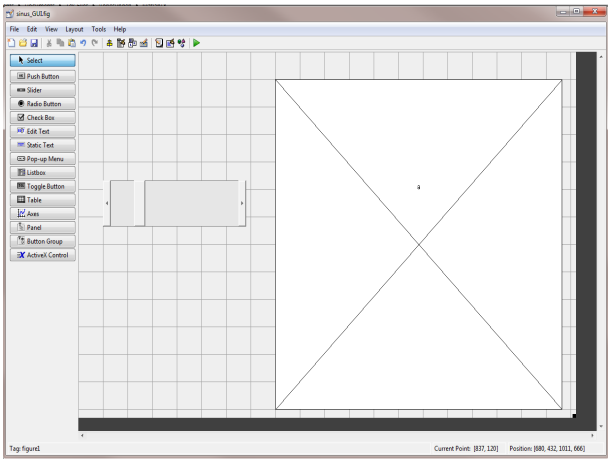
```
% --- Executes on button press in ggT.
function ggT_Callback(hObject, eventdata, handles)
% handles    structure with handles and user data (see GUIDATA)
a=str2num(handles.a.String);
b=str2num(handles.b.String);
ggT=gcd(a,b);  % Matlab-Funktion (g)reatest (c)ommon (d)ivisor
handles.ergebnis.String=num2str(ggT);

% --- Executes on button press in kgV.
function kgV_Callback(hObject, eventdata, handles)
% handles    structure with handles and user data (see GUIDATA)
a=str2num(handles.a.String);
b=str2num(handles.b.String);
kgV=lcm(a,b);  % Matlab-Funktion (l)east (c)ommon (m)ultiple
handles.ergebnis.String=num2str(kgV);
```

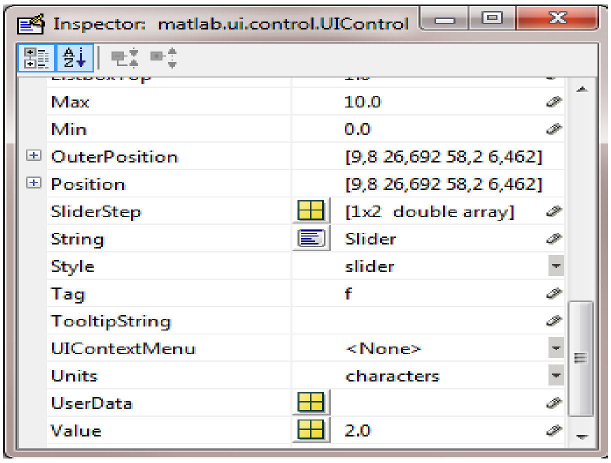
- **Testen der GUI** durch **Drücken des Pfeil-Icon oder Tools -> Run**.



- **Zum Schluss** können wir die **GUI als in Matlab einzelnes ausführbares .m-File exportieren durch File -> Export**. In dem **exportierten .m-File steht dann der komplette Programmcode der GUI** (den man auch ohne Hilfe von GUIDE so eingegeben könnte, um die GUI ganz alleine zu erzeugen...). Allerdings wird **manchmal auch noch ein .mat-file** erzeugt, indem **zusätzliche Daten gespeichert** werden, so dass man doch noch mal 2 Files hat.

- Noch ein **Beispiel** `sinus_GUI` zum **Plotten einer Sinusfunktion** $\sin(2\pi f \cdot x)$, **deren Frequenz f sich durch einen Slider (Schiebebalken) verändern lässt**:
  - **Axes:** `tag a`
  - **Slider:** `Max 10, Min 0, Value 2, tag f.`

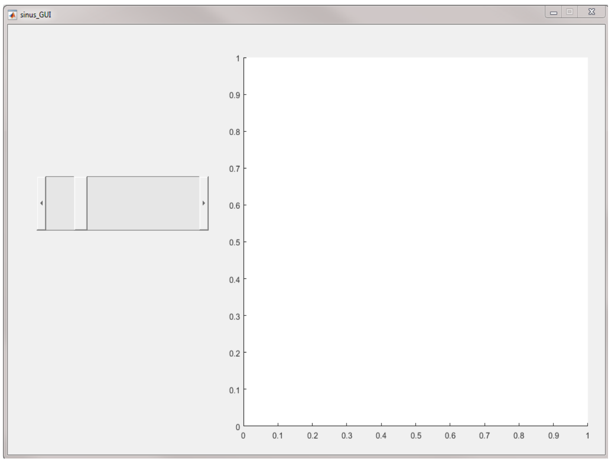


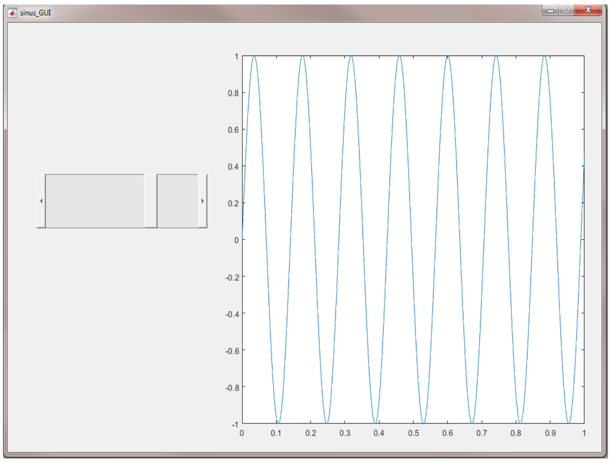


- **Callback des Slider** ändern.

```
% --- Executes on slider movement.
function f_Callback(hObject, eventdata, handles)
f=hObject.Value;
x=linspace(0,1,1000);
y=sin(2*pi*f*x);
plot(handles.a,x,y);
```

und **testen.**





- Wünschenswert wäre, dass schon **beim Erscheinen unserer GUI** der Sinus mit der voreingestellten Frequenz f=2 geplotted wird. Dazu dient die **OpeningFcn-Funktion der GUI direkt hinter dem Initialisierungscode**. Wir können einfach obigen Code kopieren, aber **Vorsicht**: Hier müssen wir `f=handles.f.Value` statt `f=hObject.Value` schreiben!

```
% --- Executes just before sinus_GUI is made visible.
function sinus_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to sinus_GUI (see VARARGIN)
```

```
% Choose default command line output for sinus_GUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes sinus_GUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);
f=handles.f.Value;  % hier statt f=hObject.Value
x=linspace(0,1,1000);
y=sin(2*pi*f*x);
plot(handles.a,x,y);
```

- **Daten auslesen können wir z.B. mit der Eigenschaft UserData der GUI-Figure**. Das **Handle auf die GUI-Figure ist in handles.output** gespeichert. Wir **erweitern den Callback des Slider** um zwei Zeilen: Wir **speichern x und y in der Structure mydata und übergeben diese als UserData**.

```
% --- Executes on slider movement.
function f_Callback(hObject, eventdata, handles)
...
mydata.x=x; mydata.y=y;
handles.output.UserData=mydata;
```

Wir **rufen unsere GUI mit Rückgabewert des Handles im Command Window auf**,

```
>> h=sinus_GUI
h =
  Figure (figure1) with properties:
        Name: 'sinus_GUI'
```

...**ändern die Frequenz mit dem Slider und plotten im Command Window mit den aktuellen Werten**.

```
>> mydata=h.UserData
mydata =
  struct with fields:
    x: [1x1000 double]
    y: [1x1000 double]
>> plot(mydata.x,mydata.y);
```

- **Daten einlesen können wir z.B. wie gewohnt, indem wir diese als Eingabevariablen übergeben**. Diese werden dann **in dem Cell Array varargin** zusammengefasst, auf das wir **in der OpeningFcn Zugriff** haben. Wir **wollen beim Starten unserer GUI schon eine Frequenz f übergeben lassen und ergänzen entsprechend die OpeningFcn**.

```
% --- Executes just before sinus_GUI is made visible.
function sinus_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
...
% uiwait(handles.figure1);
if ~isempty(varargin)
    handles.f.Value=varargin{1};
end
f=handles.slider1.Value;
...
```

Test mit f=8 (f zwischen 0 und 10; unser default-Wert war f=2).

```
>> h=sinus_GUI(8);
```

Vorsicht: GUIDE interpretiert in seinem automatisch erstellten .m-File einen **optional übergenenen String schon auf andere Weise!** wie folgt (**macht** also ein **Function Handle daraus**)

```
function varargout = sinus_GUI(varargin)
...
%      SINUS_GUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in SINUS_GUI.M with the given input arguments.
...
% Begin initialization code - DO NOT EDIT
...
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
...
```

20 Objektorientierte Programmierung

- Hier eine sehr kurze Einführung zur **Objektorientierten Programmierung (OOP)** mit Matlab. Mehr dazu unter:
  Matlab-Hilfe -> Advanced Software Development -> Object Oriented Programming.

- **Vorbemerkung**: Alle **Datentypen in MATLAB sind Klassen mit gewissen Eigenschaften und Methoden**, z.B. gehören numerische Werte standardmäßig zur Klasse `double` und können addiert werden. Eine **“Variable von einem gewissen Typ”**, oder anders ausgedrückt ein **“Objekt einer bestimmten Klasse”** lässt sich **mit dem zugehörigen Konstruktor erzeugen**. Dies ist eine **Funktion mit dem gleichen Namen wie die Klasse**, also z.B. `int8` oder `uint8` (ganze Zahlen aus $\{-128, \ldots, 127\}$ bzw. $\{0, \ldots, 255\}$).

```
>> x=int8(-3), y=uint8(-3)
x =
  int8
   -3
y =
  uint8
   0
```

  Dabei kennen diese **verschiedenen Klassen** beide eine **Methode (Funktion)** `plus` **mit dem selben Namen**, und MATLAB ruft die entsprechende Methode der jeweiligen Klasse anhand der übergebenen Argumente auf. Dies nennt man auch **Überladen von Funktionen**. Das +**Zeichen** wird hierbei von MATLAB als **Ersatz für die Funktion** `plus` erkannt.

```
>> x+x, plus(x,x), y+y, plus(y,y)
ans =
  int8
   -6
ans =
  int8
   -6
ans =
  uint8
   0
ans =
  uint8
   0
```

20.1 Value-Klassen

- Die Grundklassen in Matlab sind **Value-Klassen**, bei denen **wie bei numerischen Datentypen einer Variablen bei Zuweisung der Wert (ganzes Objekt) übergeben** wird und nicht nur ein Handle darauf wie bei graphischen Objekten (dafür gibt es in Matlab allgemein die Handle-Klassen).

- Damit lassen sich einfach **neue (numerische) Datentypen definieren**, indem man geeignete **Rechenregeln als Methoden der Klasse** implementiert. Dabei lassen sich z.B. auch die **arithmetischen Operationen** für `+,-,*,/` **überladen**, so dass man bequem `z=x+y` statt `z=myplus(x,y)` schreiben kann.

- **Beispiel**: Hier wollen wir eine **neue Value-Klasse** zum **Rechnen in Restklassen-Körpern**

$$\frac{\mathbb{Z}}{m \cdot \mathbb{Z}} = \{0, 1, \ldots, m-1\} \quad \text{mit } +, -, *, / \text{ und Rechnen modulo } m$$

für Primzahlen m erstellen, z.B. gilt

$$7 = 2 \mod 5 \quad , \quad -1 = 4 \mod 5 \quad , \quad 7 + 8 = 15 = 0 \mod 5 \quad , \quad 3 \cdot 2 = 6 = 1 \mod 5$$

d.h. das **multiplikative Inverse** zu 3 ist

$$3^{-1} = 2 \mod 5$$

und daher auch

$$4/3 = 4 \cdot 3^{-1} = 4 \cdot 2 = 8 = 3 \mod 5$$

- Zur **ganzzahligen Division mit Rest** können wir die Matlab-Funktion `r=mod(z,m)` verwenden

```
>> r=mod([7,-1,15,6],5)
r =
     2     4     0     1
```

- Das **multiplikative Inverse** zu z modulo m lässt sich **rechnerisch mit Hilfe des erweiterten Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers** von z und m bestimmen: Dieser liefert zwei Zahlen x und y mit

$$x \cdot z + y \cdot m = \mathrm{ggT}(z, m) = 1 \quad \text{für } m \text{ Primzahl (und } z \neq 0 \text{ kein Vielfaches von } m)$$

Folglich gilt

$$x \cdot z = 1 \mod m \quad \text{d.h.} \quad x = z^{-1} \mod m$$

Die Matlab-Funktion `[g,x,y]=gcd(z,m)` liefert das Gewünschte.

```
>> [g,x,y]=gcd(1:4,5)
g =
     1     1     1     1
x =
     1    -2     2    -1
y =
     0     1    -1     1
>> mod(x,5)   % benötigt wird hier also nur x
ans =
     1     3     2     4
```

- Wir **definieren** nun für $\frac{\mathbb{Z}}{m \cdot \mathbb{Z}}$ den **neuen Datentyp/Value-Klasse** `restklasse` mit den Eigenschaften `zahl` und `modulo`, und den **üblichen Rechenregeln als Methoden**.
Dies geht **wie gewohnt als .m-File** mit den **Schlüsselwörtern** `classdef`, `properties` und `methods`.
Da wir **hier der Einfachheit halber alle Methoden in das .m-File der Klasse** schreiben, benötigen wir keinen eigenen Klassenordner (dafür **siehe aber auch Matlab-Hilfe zu class folder** `@ClassName`).

```
classdef restklasse
 properties
   zahl
   modulo
 end
 methods
 ...
 end
end
```

- Um Variablen als **Objekte dieser Klasse erzeugen** zu können, benötigt die Klasse eine erzeugende Funktion, den sogenannten Konstruktor. Die **Konstruktor-Funktion hat den gleichen Namen wie die Klasse**. Diese und auch weitere der Klasse zugeordneten Funktionen werden wie gewöhnliche (Unter-)Funktionen geschrieben, nach dem Schlüsselwort `methods`.

```
classdef restklasse
 properties...
 methods
   % Konstruktor
   function r=restklasse(z,m)
   r.zahl=mod(z,m);
   r.modulo=m;
   end
 end
end
```

Damit **erzeugen** wir z.B. eine **Variable** `r` **vom neuen Typ restklasse** in $\frac{\mathbb{Z}}{5\cdot\mathbb{Z}}$.

```
>> r=restklasse(9,5)
r =
  restklasse with properties:
      zahl: 4
    modulo: 5
```

- **Auf Eigenschaften zugreifen und verändern** können wir mit `Variablenname.Eigenschaft` (analog zu Structures).

```
>> r.zahl
ans =
     4
>> r.modulo=3
r =
  restklasse with properties:
      zahl: 4
    modulo: 3
```

- Das **Verändern "per Hand" ist hier sicherlich unerwünscht**, denn wir hätten ja z.B. gerne als `zahl` nur Vertreter aus $\frac{\mathbb{Z}}{3\cdot\mathbb{Z}} = \{0, 1, 2\}$. Wir können aber die **Zugriffsrechte einschränken**, so dass z.B. gilt: **nur die Methoden der Klasse dürfen deren Eigenschaften verändern**.

```
classdef restklasse
 properties (SetAccess=private)
   zahl
   modulo
 end
 methods...
end
```

Testen

```
>> r=restklasse(9,5)
>> r.modulo=3
You cannot set the read-only property 'modulo' of restklasse.
```

Vorsicht: Manchmal werden nach dem Ändern einer Klasse nicht alle Änderungen für die bereits erzeugten Variablen/Objekte übernommen. Diese müssen dann zunächst neu erzeugt werden (und manchmal auch zuvor gelöscht).

- Zu den **Methoden der Klasse** können wir beliebig viele Funktionen hinzufügen. Machen wir zunächst die **Ausgabe etwas schöner, indem wir die Matlab-Funktion disp überladen**.

```
classdef restklasse
 properties...
 methods
   % Konstruktor
         ...
   % weitere Methoden
   function disp(r)
   disp([num2str(r.zahl),' mod ',num2str(r.modulo)]);
   end
 end
end
```

Testen

```
>> r
r =
4 mod 5
```

- Weiter macht es Sinn, die **Funktionen** `plus(r1,r2)`, `minus(r1,r2)`, `mtimes(r1,r2)` **für die Rechenregeln** `r1+r2`, `r1-r2`, `r1*r2` zu **überladen**, die hier **alle analog aufgebaut** sind. Falls dabei `r1` oder `r2` ein numerischer Typ ist, machen wir hier der Einfachheit halber daraus zunächst einen `restklasse`-Typ, so dass der Anwender einfach `2*r+3` mit numerischen Zahlen 2, 3 und restklassen-Variable r modulo 5 eingeben kann anstatt `restklasse(2,5)*r+restklasse(3,5)`.

```
classdef restklasse
 properties...
 methods
 ...
   % weitere Methoden
   function r=plus(r1,r2)
   if ~isa(r1,'restklasse')
      r1=restklasse(r1,r2.modulo);
   elseif ~isa(r2,'restklasse')
      r2=restklasse(r2,r1.modulo);
   end
   m=r1.modulo;
   z=mod(r1.zahl+r2.zahl,m);
   r=restklasse(z,m);
   end
   function r=minus(r1,r2)
   ...
   z=mod(r1.zahl-r2.zahl,m);
   ...
   end
   function r=mtimes(r1,r2)
   ...
   z=mod(r1.zahl*r2.zahl,m);
   ...
   end
 end
end
```

Testen

```
>> r2=restklasse(2,5); r3=restklasse(3,5);
>> r2+r3, r2-r3, r2*r3, r2*r+r3, 2*r+3
ans =
0 mod 5
ans =
4 mod 5
ans =
1 mod 5
ans =
1 mod 5
ans =
1 mod 5
```

- Achtung: **Als Vorzeichen benötigen + und - eigene Funktionen** `uplus` und `uminus`.

```
>> -r
Undefined unary operator '-' for input arguments of type 'restklasse'.
```

```
% weitere Methoden
function r=uplus(r)
% hier steht einfach nix
end
function r=uminus(r)
r=restklasse(-r.zahl,r.modulo);
end
```

Testen

```
>> -r, +r
ans =
1 mod 5
ans =
4 mod 5
```

- Nun das **multiplikative Inverse**

```
% weitere Methoden
function invr=inv(r)
m=r.modulo;
z=r.zahl;
if z==0
   warning('Division durch Null!');
   invr=r;
else
   [~,x,~]=gcd(z,m);
   invr=restklasse(x,m);
end
end
```

Testen

```
>> inv(r), inv(r-r), inv(restklasse(70,101))
ans =
4 mod 5
Warning: Division durch Null!
```

```
> In restklasse/inv (line 55)
ans =
0 mod 5
ans =
13 mod 101
>> 13*70
ans =
   910
```

- Damit können wir **Dividieren, indem wir mit dem Inversen multiplizieren** $r_1/r_2 = r_1 \cdot r_2^{-1}$

```
% weitere Methoden
function r=mrdivide(r1,r2)
if ~isa(r2,'restklasse')
   r2=restklasse(r2,r1.modulo);
end
r=r1*inv(r2);
end
```

Testen

```
>> r2/r3, 2/r3, r2/3
ans =
4 mod 5
ans =
4 mod 5
ans =
4 mod 5
```

- **Potenzieren** (hier nur für ganzzahlige Hochzahlen $n \geq 0$)

```
% weitere Methoden
function rn=mpower(r,n)
if n==0 % Rekursionsanfang
   rn=restklasse(1,r.modulo);
else
   rn=r*r^(n-1); % Rekursion
end
end
```

Testen

```
>> r^0, r^1, r^2
ans =
1 mod 5
ans =
4 mod 5
ans =
1 mod 5
```

- Wir können nun **auch wie gewohnt mit Function-Handle rechnen**

```
>> f=@(x) -x^2+2/x
f =
  function_handle with value:
    @(x)-x^2+2/x
>> f(r)
```

```
ans =
2 mod 5
```

- Den **Vergleich** müssen wir allerdings noch implementieren wegen

```
>> r2==r3
Undefined operator '==' for input arguments of type 'restklasse'.
```

Dazu überladen wir die Funktion eq.

```
% weitere Methoden
function e=eq(r1,r2)
if ~isa(r1,'restklasse')
   r1=restklasse(r1,r2.modulo);
elseif ~isa(r2,'restklasse')
   r2=restklasse(r2,r1.modulo);
end
e=(r1.zahl==r2.zahl);
end
```

Testen

```
>> r2==r3
ans =
  logical
   0
>> r2==-3
ans =
  logical
   1
```

- Für das (effiziente) **Rechnen mit Vektoren/Matrizen, deren Einträge restklasse-Objekte sind**, müssten wir z.B. **auch noch die elementweisen Operationen** wie `.*` implementieren...**und** auch die schon vorhandenen Methoden für den **Umgang mit restklasse-Arrays** geeignet abändern (insbesondere Konstruktor und disp; siehe auch **Matlab-Hilfe zu Object Arrays**).

20.2 Handle-Klassen

- Bei Objekten von Klassen, die von der **Handle-Klasse** erben, wird **bei Zuweisung nur ein Handle auf das Objekt übergeben** (wie bei graphischen Objekten). Deshalb muss das **Handle meist nicht als Rückgabe-Variable einer Funktion** deklariert werden; **wichtige Ausnahme: Die Konstruktor-Methode** muss das Handle zurückgeben. Die Handle-Klasse selbst ist abstrakt, hat keine Eigenschaften, aber **folgende Methoden, die auch von jeder ihrer Unterklassen verwendet werden können**:

```
>> m=methods('handle')
m =
  13x1 cell array
    {'addlistener'}{'delete'}{'eq'}{'findobj'}{'findprop'}{'ge'}
    {'gt'}{'isvalid'}{'le'}{'listener'}{'lt'}{'ne'}{'notify'}
```

- Als **Beispiel** erzeugen wir eine Unterklasse `schrank` der Handle-Klasse. In einen **Schrank** kann man **Dinge reinlegen und rausnehmen**, und die gewünschte **Größe lässt sich nur bei der Herstellung angeben, aber danach nicht mehr ändern**.

```
classdef schrank < handle % erbt von der Handle-Klasse
    properties (SetAccess=immutable) % nur im Konstruktor wählbar
        groesse
    end
    properties (SetAccess=private)
        dinge=0 % default-Wert
    end
    methods
        % Konstruktor
        function s=schrank(g)
           if nargin==0
               g=10;
           end
           s.groesse=g;
        end
        % weitere Methoden
        function rein(s,d)
            s.dinge=min(s.dinge+d,s.groesse);
        end
        function d=raus(s,d)
            d0=s.dinge;
            s.dinge=max(d0-d,0);
            d=d0-s.dinge;
        end
    end
end
```

- **Testen**

```
>> s1=schrank % Größe muss bei obigem Konstruktor nicht übergeben werden
s1 =
  schrank with properties:
    groesse: 10
      dinge: 0
>> s2=s1; rein(s2,5); s1  % s2 und s1 Handle auf gleichen Schrank
s1 =
  schrank with properties:
    groesse: 10
      dinge: 5
>> d=raus(s1,6), s1.dinge
d =
     5
ans =
     0
```

- Die **Größe kann hier nur im Konstruktor festgelegt werden**.

```
>> s1.groesse=5
You cannot set the read-only property 'groesse' of schrank.
>> s3=schrank(5)
s3 =
  schrank with properties:
    groesse: 5
      dinge: 0
```

- Wir können **auch** z.B. die **Methoden eq und delete** verwenden, die unsere **Schrank-Klasse von der Handle-Klasse erbt**.

```
>> s3==s2    % (eq) Zeigen Handle auf gleichen Schrank?
ans =
  logical
   0
>> delete(s2), s1
s1 =
  handle to deleted schrank
```

20.3 Vererbung

- **(Fast) jede Klasse kann vererbt werden**. Die erbende **Unterklasse** kann dann **auch alle (erlaubten) Eigenschaften und Methoden der vererbenden Oberklasse(n)** verwenden (d.h. sofern nicht ausdrücklich einschränkende Attribute gesetzt sind). **Im Konstruktor der erbenden Unterklass**e kann der **Konstruktor einer vererbenden Oberklasse explizit** aufgerufen werden durch `objekt@oberklasse(...)` **oder** er wird **automatisch implizit ohne Eingabevariablen** aufgerufen, in diesen Fall muss der Konstruktor der Oberklasse aber auch so programmiert sein, dass er ohne Eingabevariablen auskommt (z.B. wie bei unserer Schrank-Klasse).

- Als **Beispiel** erzeugen wir eine Unterklasse `geheimschrank` unserer Schrank-Klasse. In einem **Geheimschrank** kann man **Dinge in einem geheimen Fach verstecken**. Weil das nicht jeder wissen soll, setzen wir für die **Eigenschaft** `geheimfach` und die **Methode** `verstecken` **jeweils das Attribut** `Hidden=true`.

```
classdef geheimschrank < schrank % erbt von Schrank-Klasse
    properties (SetAccess=private, Hidden=true)
        geheimfach=0
    end
    methods
        % Konstruktor
        function gs=geheimschrank(gr)
            gs@schrank(gr); % Konstruktor der Oberklasse explizit
        end
    end
    % weitere Methoden
    methods (Hidden=true)
        function verstecke(gs,d)
            gs.geheimfach=gs.geheimfach+raus(gs,d);
        end
    end
end
```

- **Testen**

```
>> gs=geheimschrank(5)
gs =
  geheimschrank with properties:
    groesse: 5
      dinge: 0
% Geheimfach nicht sichtbar (auch Methode verstecken nicht)
>> p=properties(gs)
p =
  2x1 cell array
```

```
    {'groesse'}
    {'dinge'  }
>> isa(gs,'schrank') % Geheimschrank ist also auch Schrank
ans =
  logical
   1
>> rein(gs,4), gs.dinge
ans =
     4
>> verstecke(gs,3), gs.dinge, gs.geheimfach
ans =
     1
ans =
     3
```

20.4 Ereignisse

- **Handle-(Unter-)Klassen können Ereignisse mit `events` definieren und bei deren Eintreten eine Nachricht mit `notify(ausloesendes_objekt,ereignis)` an alle registrierten Zuhörer versenden**.

- Als **Beispiel** erzeugen wir eine weitere Unterklasse `kuehlschrank` unserer Schrank-Klasse. Ein **smarter Kühlschrank ist immer kalt genug und meldet sich, wenn er fast leer ist**. Dazu **überladen** wir die **Methode `raus` der Oberklasse** Schrank. In der neuen Methode unserer Unterklasse wollen wir auch die Methode `raus` der Oberklasse selbst verwenden. Da die Objekte der Unterklasse aber auch als Objekte der Oberklasse erkannt werden, muss die Methode der Oberklasse hier mit `raus@schrank(...)` aufgerufen werden. Ein Konstruktor ist hier nicht unbedingt notwendig wegen dem default-Wert für die eigene neue Eigenschaft `temperatur`, und da der Konstruktor der Oberklasse Schrank implizit ohne Argumente aufgerufen werden kann.

```
classdef kuehlschrank < schrank
    properties (Constant=true)
        temperatur=4
    end
    events (NotifyAccess=private)
        fastleer
    end
    methods
        % Konstruktor hier nicht unbedingt notwendig
        % weitere Methoden
        function d=raus(k,d)
            d=raus@schrank(k,d); % raus von Oberklasse Schrank
            if k.dinge<=k.groesse/3
                notify(k,'fastleer');
            end
        end
    end
end
```

- Zum **Testen** legen wir zunächst nur etwas in den Kühlschrank rein, denn noch reagiert ja niemand auf das Ereignis.

```
>> k=kuehlschrank; rein(k,5); k
k =
```

```
  kuehlschrank with properties:
    temperatur: 4
       groesse: 10
         dinge: 5
```

- Dann schreiben wir noch in ein neues .m-File eine **Callback-Funktion** `reagiere`**, welche bei Eintreten des Ereignisses aufgerufen werden soll**. Callback-Funktionen verlangen die Syntax mit beiden Eingabevariablen auslösendes Objekt und Ereignisobjekt, wobei wir hier das Ereignisobjekt nicht verwenden.

```
function reagiere(k,~)
% Ereignisobjekt hier nicht verwendet
disp('Habe␣wieder␣aufgefüllt.');
rein(k,k.groesse);
end
```

- Zum **Testen** registrieren wir ein **Listener-Objekt als Zuhörer**.

```
>> alex=listener(k,'fastleer',@(k,e) reagiere(k,e))
alex =
  listener with properties:
       Source: {[1x1 kuehlschrank]}
    EventName: 'fastleer'
     Callback: @(k,e)reagiere(k,e)
      Enabled: 1
    Recursive: 0
>> raus(k,2);
Habe wieder aufgefüllt.
>> k.dinge
ans =
    10
```

- Wenn man hier das **Event-Attribut** `NotifyAccess=private` auskommentiert, kann jeder das Ereignis auslösen. Mit diesem Attribut darf nur die kuelschrank-Klasse das Ereignis auslösen, somit kann man also einen **Fehlalarm vermeiden**.

```
>> notify(k,'fastleer')
Habe wieder aufgefüllt.

% Event-Attribut setzen: NotifyAccess=private
>> notify(k,'fastleer')
Error using kuehlschrank/notify
Cannot notify listeners of event 'fastleer' in class 'kuehlschrank'.
```

- **Ein Listener-Objekt** kann **auf mehrere auslösende Objekte** reagieren, **aber für jede Art von Ereignis** braucht es einen **eigenen Listener**.

```
>> k2=kuehlschrank;
>> alex.Source{2}=k2
alex =
  listener with properties:
       Source: {2x1 cell}
>> raus(k2,0);
Habe wieder aufgefüllt.
```

- Man kann auch **von mehreren Klassen erben, z.B. ein kühler Geheimschrank**.

```
classdef kgschrank < kuehlschrank & geheimschrank
   methods
      % Konstruktor
      function kgs=kgschrank(g)
         kgs@geheimschrank(g);
         % Konstruktor von kuehlschrank implizit
      end
   end
end
```

- **Testen**

```
>> kgs=kgschrank(5)
kgs =
  kgschrank with properties:
    temperatur: 4
       groesse: 5
         dinge: 0
>> kgs.geheimfach
ans =
     0
```

20.5 Object-Arrays

- Mehrere **Objekte der gleichen Klasse kann man wie gewohnt zu einem Array zusammenfassen**.

```
>> a=[k,k2]
a =
  1x2 kuehlschrank array with properties:
    temperatur
    groesse
    dinge
>> a(1)
ans =
  kuehlschrank with properties:
    temperatur: 4
       groesse: 10
         dinge: 0
```

- **Vorsicht: Es gibt einige Einschränkungen**, z.B. muss beim Überladen der Bilschirmausgabe mit `disp` dies auch geeignet für Arrays implemtiert werden. Und **Speicher einfach vorreservieren geht nur, wenn Konstruktor ohne Eingabevariablen aufgerufen werden** kann. Nachfolgend wird dann das Array mit Kopien aufgefüllt.

```
>> b(3)=kuehlschrank
b =
  1x3 kuehlschrank array with properties:
    temperatur
    groesse
    dinge
>> c(3)=geheimschrank
Not enough input arguments.
```

```
Error in geheimschrank (line 8)
            gs@schrank(gr); % Konstruktor der Oberklasse explizit
```

- **Objekte verschiedener Klassen lassen sich nicht ohne weiteres zusammenfassen**.

  ```
  >> d=[k,gs]
  Error using horzcat
  The following error occurred converting from geheimschrank to
  kuehlschrank:
  Too many input arguments.
  ```

- Aber **Objekte mit gleicher Oberklasse lassen sich zusammenfassen, wenn diese von der matlab.mixin.Heterogeneous-Klasse erbt**. Wir ergänzen also bei unserer Schrank-Oberklasse die Klassendefinition zu

  ```
  classdef schrank < handle & matlab.mixin.Heterogeneous
  ...
  ```

  und dann funktioniert's

  ```
  >> d=[k,gs]
  d =
    1x2 heterogeneous schrank (kuehlschrank, geheimschrank) array with properties:
      groesse
      dinge
  ```