

# Einführung in Matlab - Einheit 2

## Programmieren, Datenstrukturen

Jochen Schulz

Georg-August Universität Göttingen 

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

- Variablen in Skript-Files

- globaler Workspace (d.h. bereits vorhandene Variablen können direkt benutzt oder überschrieben werden)
- gültig bis explizit gelöscht

- Variablen in Function-Files

- innerhalb der Funktion definiert und werden bei Verlassen der Funktion gelöscht.
- Variablen des globalen Workspace können nicht benutzt werden.

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

# for - Schleife

```
for <variable> = <Ausdruck>  
    <Befehle>  
end
```

## Bemerkungen:

- Ausdruck: `i:s:j`.
- Übersichtlichkeit: `Befehle` einrücken.
- Weitere Schleifen-Konstrukte: `while` und `switch`.

# Schleifen - Beispiele

- Berechne  $\sum_{i=1}^{1000} \frac{1}{i}$

```
sum=0; for j=1:1000, sum=sum+1/j; end, sum
```

```
sum = 7.4855
```

- Berechnen dreier Werte

```
for x=[pi/6 pi/4 pi/3], sin(x), end
```

```
ans = 0.5000
```

```
ans = 0.7071
```

```
ans = 0.8660
```

- Matrix als *Ausdruck*

```
for x=eye(3), x', end
```

```
ans = 1 0 0
```

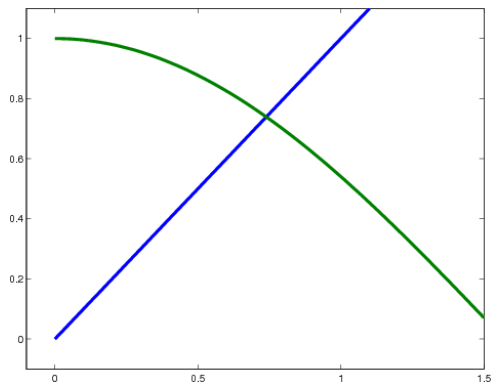
```
ans = 0 1 0
```

```
ans = 0 0 1
```

# Fixpunkt

Suche ein  $x_f \in \mathbb{R}$  so dass

$$x_f = \cos(x_f)$$



Voraussetzung: Abbildung kontrahierend

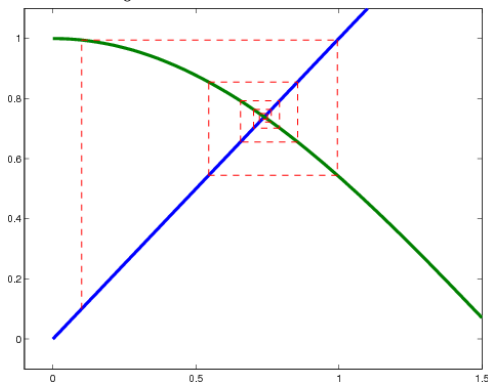
$$|f(x) - f(y)| \leq C|x - y|, \quad C < 1 \quad \forall x, y \in I$$

# Fixpunkt-Iteration

Fixpunkt-Iteration

$$x_{k+1} = \cos(x_k)$$

bei geeignetem Startwert  $x_0$ .



(Funktioniert wenn die Abbildung kontrahierend ist)



# Fixpunkt-Iteration - Implementation

```
% Plot 1
x = linspace(0,1.5,50);
y = cos(x);
plot(x,x,x,y,'LineWidth',3),
axis([-0.1 1.5 -0.1 1.1]);
hold on;
pause; % stoppt bis eine Taste gedrückt wird
z(1) = 0.1; % Anfangswert
it_max = 10; % Iterationsschritte
for i = 1:it_max
    z(i+1) = cos(z(i));
    plot([z(i) z(i)], [z(i) z(i+1)], 'r--', 'LineWidth', 1);
    pause;
    plot([z(i) z(i+1)], [z(i+1) z(i+1)], 'r--', 'LineWidth', 1);
    hold on;
    pause; % stoppt bis eine Taste gedrückt wird
end;
```

# Einige Grafikbefehle

- `figure`  
startet ein Grafik-Fenster.
- `hold on`  
alle Grafiken in einem Fenster werden übereinander gezeichnet.
- `hold off` (Default)  
bestehende Grafik wird gelöscht und durch die neue Grafik ersetzt.

# Vandermonde-Matrix I

Berechne zu einem gegebenen Vektor  $x = (x_1, \dots, x_n)$  die Vandermonde-Matrix

$$V := \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{pmatrix}.$$

# Vandermonde-Matrix II

```
function V = vandermonde2(x)
% vandermonde2 berechnet die Vandermonde Matrix zu einem
%           Vektor x
%
%           INPUT:
%           x Zeilenvektor
%           OUTPUT:
%           V Vandermonde-Matrix
%   Gerd Rapin           8.11.2003

n = length(x);

V = zeros(n,n);
for i = 1:n
    for j = 1:n
        V(i,j) = x(i)^(j-1);
    end
end
end
```

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

# Quadratische Gleichung

$$\begin{cases} \text{Suche } x \in \mathbb{R}, \text{ so dass} \\ x^2 + px + q = 0 \end{cases}$$

Fallunterscheidung für  $d := \frac{p^2}{4} - q$ :

**Fall a)** :  $d > 0$     2 Lösungen:  $x = -\frac{p}{2} \pm \sqrt{d}$

**Fall b)** :  $d = 0$     1 Lösung:  $x = -\frac{p}{2}$

**Fall c)** :  $d < 0$     keine Lösung

# Implementierung

```
function [anz_loesungen, loesungen]=quad_gl(p,q)
%-----
% quad_gl berechnet die Loesungen der quadratischen
% Gleichung  $x^2 + px + q = 0$ 
% INPUT: Skalare p
%          q
%
% OUTPUT: anz_loesungen   Anzahl der Loesungen
%          loesungen      Vektor der Loesungen
%
% Gerd Rapin      8.11.2003
%-----
d=p^2/4-q; % Diskriminante
```

# Implementierung II

```
% 2 Loesungen
if d>0
    anz_loesungen=2;
    loesungen=[-p/2-sqrt(d) -p/2+sqrt(d)];
end

% 1 Loesung
if d==0
    anz_loesungen=1;
    loesungen=[-p/2];
end

% 0 Loesungen
if d<0
    anz_loesungen=0;
    loesungen=[];
end
```



# Logische Operationen

- logische Variablen (Datentyp ist **logical**).
- Variablen dieses Typs sind entweder **TRUE** (1) oder **FALSE** (0).
- Numerische Werte ungleich 0 werden als **TRUE** gewertet.

```
a = (1<2)
```

```
a = 1
```

```
b = ([ 1 2 3 ] < [ 2 2 2 ])
```

```
b =      1      0      0
```

```
whos
```

Name	Size	Bytes	Class
a	1x1	1	logical array
b	1x3	3	logical array

# Vergleichs-Operatoren

$a = [1 \ 1 \ 1]$ ,  $b = [0 \ 1 \ 2]$

Operation	Bedeutung	Ergebnis
$a == b$	gleich	0 1 0
$a ~= b$	ungleich	1 0 1
$a < b$	kleiner	0 0 1
$a > b$	größer	1 0 0
$a <= b$	kleiner oder gleich	0 1 1
$a >= b$	größer oder gleich	1 1 0

Bem: 1 = wahre Aussage, 0 = falsche Aussage

Bem: Komponentenweise Vergleiche sind auch für Matrizen gleicher Größe möglich!

# Logische Operatoren

<code>&amp;</code>	logisches und	<code>~</code>	logisches nicht
<code> </code>	logisches oder	<code>xor</code>	exklusives oder

Beispiele:

```
x=[-1 1 1]; y=[1 2 -3];
```

```
>> (x>0) & (y>0)
ans =
     0     1     0
```

```
>> ~( (x>0) & (y>0))
ans =
     1     0     1
```

```
>> (x>0) | (y>0)
ans =
     1     1     1
```

```
>> xor(x>0,y>0)
ans =
     1     0     1
```

## Einfache Bedingung

```
if  <Ausdruck>  
    <Befehle>  
end
```

## Bed. mit Alternative

```
if  <Ausdruck>  
    <Befehle>  
else  
    <Befehle>  
end
```

Die Befehle zwischen **if** und **end** werden ausgeführt, wenn der *Ausdruck* wahr (**TRUE**) ist. Andernfalls werden (soweit vorhanden) die Befehle zwischen **else** und **end** ausgeführt.

*Ausdruck* ist wahr, wenn alle Einträge von *Ausdruck* ungleich 0 sind.

# While-Schleifen

```
while <Ausdruck>  
    <Befehle>  
end
```

Die Befehle werden wiederholt, so lange die Bedingung *Ausdruck* wahr ist. *Ausdruck* ist wahr, wenn alle Einträge von *Ausdruck* ungleich 0 sind.

**Beispiel:** Berechne  $\sum_{i=1}^{1000} \frac{1}{i}$ .

```
n = 1000; sum = 0; i = 1;  
while (i <= n)  
    sum = sum+(1/i);  
    i = i+1;  
end  
sum
```

# Größter gemeins. Teiler (ggT)

Berechnung des ggT von natürlichen Zahlen  $a$  und  $b$  mit Hilfe des euklidischen Algorithmus

Idee: Es gilt  $\text{ggT}(a, b) = \text{ggT}(a, b - a)$  für  $a < b$ .

Algorithmus:

Wiederhole, bis  $a = b$

- Ist  $a > b$ , so  $a = a - b$ .
- Ist  $a < b$ , so  $b = b - a$

# Implementierung

```
function a = ggt(a,b)
%-----
% ggt berechnet den groessten gemeinsamen Teiler (ggT)
% zweier natuerlichen Zahlen a und b
% INPUT: natuerliche Zahlen a
% b
%
% OUTPUT: ggT
%
% Gerd Rapin 11.11.2003
%-----
while (a ~= b)
    if (a > b)
        a = a-b;
    else
        b =b-a;
    end
end
end
```

- Der Befehl `break` verläßt die `while` oder `for`-Schleife.

```
x=1;
while 1
    xmin=x;
    x=x/2;
    if x==0
        break
    end
end
xmin
```

`xmin = 4.9407e-324`



- Durch `continue` springt man sofort in die nächste Iteration der Schleife, ohne die restlichen Befehle zu durchlaufen.

```
for i=1:10
    if i<5
        continue
    end
    x(i)=i;
end
x
```

x = 0 0 0 0 5 6 7 8 9 10

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

# Operator Rangfolge

Level	Operator
1	Exponent ( $\wedge$ , $\cdot\wedge$ ), <b>transpose</b>
2	logische Verneinung ( $\sim$ )
3	Multiplikation ( $\cdot$ , $\cdot$ ), Division ( $/$ , $\cdot /$ , $\backslash$ , $\cdot \backslash$ )
4	Addition (+), Subtraktion (-)
5	Doppelpunktoperator (:)
6	Vergleichsoperatoren ( $<$ , $>$ , $<=$ , $>=$ , $==$ , $\sim=$ )
7	Logisches und (&)
8	Logisches oder ( )

Bei gleicher Rangfolge wertet MATLAB von links nach rechts aus.

Die Rangfolge kann durch Kammersetzung geändert werden.

Wiederholte Anwendung von Script-Files kann zu Fehlern führen!

## Programm

```
% plotte_sin.m  
  
disp(['Plot der Sinus' ...  
      'Funktion auf [0,10]']);  
n = input(['Plot an ' ...  
          'wievielen Punkten?']);  
x = linspace(0,10,n);  
for i=1:n  
    y(i) = sin(x(i));  
end;  
plot(x,y);
```

## Aufruf

```
>> plotte_sin  
Plot der Sinus Funktion auf [0,10]  
Plot an wievielen Punkten?20  
>> plotte_sin  
Plot der Sinus Funktion auf [0,10]  
Plot an wievielen Punkten?10  
??? Error using ==> plot  
Vectors must be the same lengths.  
  
Error in ==> plotte_sin.m  
On line 9 ==> plot(x,y);
```

# globale Variablen

Mittels des Befehls `global` können Variablen des globalen Workspace auch für Funktionen manipulierbar gemacht werden.

## Funktion

```
function f=myfun(x)
% myfun.m
% f(x)=x^alpha sin(1/x)

global alpha
f=x.^alpha.*sin(1./x);
```

## Plotten

```
% plot_myfun
global alpha
alpha_w=[ 0.4 0. 6 1 1.5
          2];
for i = 1:length(alpha_w)
    alpha = alpha_w(i);
    fplot(@myfun,[0.1,1])
    hold on;
end
hold off;
```

- Alle Programme sollten zu Beginn einen Kommentar enthalten, in dem beschrieben wird, was das Programm macht.
  - Programmbeschreibung
  - Eingabevariablen
  - Ausgabevariablen
  - Beispiele
- Vor und nach logischen Operatoren und = sollte ein Leerzeichen gesetzt werden.
- Man sollte pro Zeile nur einen Befehl verwenden.
- Befehle in Strukturen, wie `if`, `for` oder `while`, sollten eingerückt werden.

- Die Namen der Variablen sollten, soweit möglich, selbsterklärend sein.
- Verfasst man umfangreiche Programme, so sollten M-Funktionen, die eine logische Einheit bilden in einem separaten Unterverzeichnis gespeichert sein. Die Verzeichnisse können durch `addpath` eingebunden werden.
- Potenzielle Fehler sollten, soweit möglich, aufgefangen werden. Speziell sollten die Eingabeparameter der Funktionen geprüft werden.

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings



- **Datentypen** werden bestimmt durch ihre Eigenschaften.
- Einzelne Elemente eines Datentyps werden **Objekte** genannt.
- Ein **Objekt** besteht meist aus drei Teilen: **Bezeichner**, **Referenzen** und **Werte** des Objekts.
- **Variablen** sind Datenobjekte deren Werte während eines Programmablaufs verändert werden können.

- Alle Variablen sind Felder (Array). Ein Skalar ist eine  $1 \times 1$ -Matrix.
- Zuweisung des Datentyps *implizit*.
- Den Datentyp eines Objekts *a* kann durch den Befehl `class(a)` bestimmt werden.

# Datentypen in MATLAB

- Gleitkommazahlen (Komplexe Zahlen)
- Characters und Strings
- Strukturen
- Cell Arrays
- Funktionen
- Sparse Matrizen
- Integer-Zahlen
- Logische Ausdrücke

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

- Standard-Datentyp ist ein Array von Gleitkommazahlen (`double`).
- Abstand von 1 zur nächsten Gleitkommazahl:  $\epsilon = 2^{-52} \sim 2.2 \cdot 10^{-16}$  (vgl. `eps`)
- Sei  $x \in \mathbb{R}$  eine reelle Zahl und  $\tilde{x}$  die Darstellung in MATLAB. Dann gilt für den Rundungsfehler

$$\frac{|x - \tilde{x}|}{|x|} \leq \frac{1}{2}\epsilon.$$

- Die größte bzw. kleinste in MATLAB darstellbare positive Zahl ist in `realmin` bzw. `realmax` gespeichert.

# Ausnahmen

- Ist eine Zahl größer als `realmax`, so meldet MATLAB einen 'Overflow' und gibt als Ergebnis `Inf` zurück.

```
realmax*1.1
```

```
ans =    Inf
```

- Bei Operationen wie  $0/0$  oder  $\infty/\infty$ , erhält man als Ergebnis `NaN` (*Not a Number*).

```
0/0
```

```
Warning: Divide by zero.
```

```
ans =    NaN
```

# Umgang mit NaN und Inf

- Mit Hilfe von `isinf` und `isnan` kann auf  $\infty$  bzw. NaN getestet werden.

```
isnan(0/0), isinf(1.2*realmax)
```

```
ans =     1    ans =     1
```

- Test auf NaN durch `==` ist nicht möglich

```
0/0 == NaN
```

```
ans =     0
```

- Bei Inf ist der Test durch `==` möglich!

```
(1.2*realmax)==Inf
```

```
ans =     1
```

- Ähnlich wie in C gibt es den Datentyp `single`. Es ist eine Darstellung in geringerer Genauigkeit.
- Durch den Befehl `single()` wird eine `double`-Zahl in eine `single`-Zahl konvertiert.
- Arithmetische Operationen mit `double`- und `single`-Objekten ergeben `single`-Objekte.



# Single

```
a = sqrt(2); b = single(a);  
c = a+b; d = a-b
```

```
d =  
    2.4203e-08
```

```
whos
```

Name	Size	Bytes	Class
a	1x1	8	double
b	1x1	4	single
c	1x1	4	single
d	1x1	4	single

```
[realmax, single(realmax)], realmax
```

```
ans =  
    Inf    Inf  
ans =  
    1.7977e+308
```

# Darstellungsformate am Beispiel 1/7

<code>format short</code>	0.1429
<code>format short e</code>	1.4286e-01
<code>format short g</code>	0.14286
<code>format long</code>	0.14285714285714
<code>format long g</code>	0.142857142857143
<code>format long e</code>	1.428571428571428e-01

Das Default-Format ist `short`.

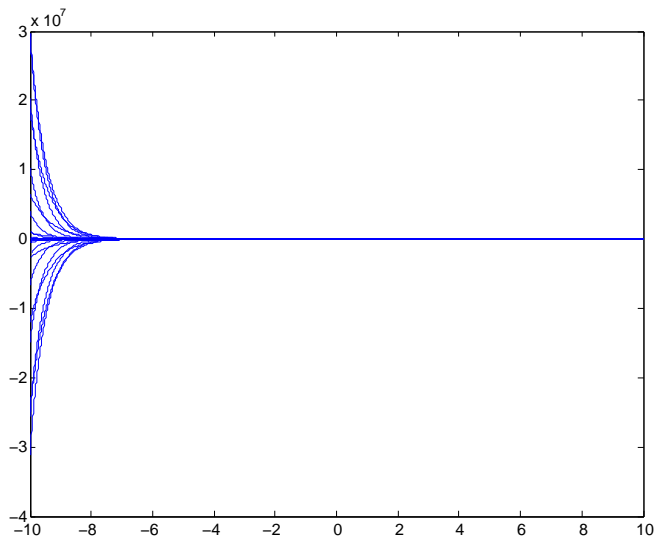
# Beispiel - Berechnung von $e$

Approximation der Exponentialfunktion durch eine Taylor-Reihe

$$P_n(x) = \sum_{j=0}^n \frac{x^j}{j!}$$

```
x = -10:0.01:10; % die x-Werte
expx = exp(x); % die wahre Exponentialfunktion
for n=0:1:25
    % so viele Nullen wie x Elemente hat
    sum=zeros(size(x));
    for j=0:n
        % das berechnet die Partialsumme
        sum=sum+x.^j/factorial(j);
    end
    % plottet relativen Fehler
    plot(x,(sum-expx)./expx);
    % wir plotten alles uebereinander
    hold on
end
```

# Berechnung von $e$ - Relativer Fehler



# Auslöschung

```
% Ausloeschung, mit 6 Dezimalstellen  
format long g % sorgt fuer lange Ausgabezahlen  
x = 0.344152  
xwahr = 0.344152*1.0000001 % Fehler  
relfx = abs(xwahr-x)/xwahr  
y = 0.344135  
z = x-y  
zwahr = xwahr-y  
relfz = abs(z-zwahr)/abs(zwahr) % relativer Fehler von z
```

```
x = 0.344152  
xwahr = 0.3441520344152  
relfx = 9.99999900671778e-08  
y = 0.344135  
z = 1.699999999999892e-05  
zwahr = 1.70344152000124e-05  
relfz = 0.00202033352005498
```

# Komplexe Zahlen

Komplexe Zahlen  $z \in \mathbb{C}$  haben die Form

$$z = x + iy, \quad x, y \in \mathbb{R}$$

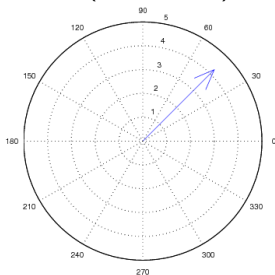
mit  $i = \sqrt{-1}$ .

- $\sqrt{-1}$  ist in MATLAB vordefiniert in den Variablen  $i, j$ .
- Durch `complex(x,y)` kann aus  $x, y \in \mathbb{R}$  die komplexe Zahl  $x + iy$  erzeugt werden.
- Für  $z = x + iy \in \mathbb{C}$  erhält man den Realteil mit  $real(z)$  und den Imaginärteil durch  $imag(z)$ .

# Polarkoordinaten

$$z \in \mathbb{C}, \quad z = re^{i\varphi} = r(\cos \varphi + i \sin \varphi)$$

- `abs(z)` ergibt den Betrag  $r$  von  $z$ .
- $\varphi$  erhält man durch `angle(z)`.
- grafische Darst.: `compass(z)` ( $z = 3 + 3i$ ).



- In diesen Datentypen werden ganze bzw. natürliche Zahlen gespeichert.
- Zur effizienten Speicherung gibt es die Datentypen `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`.
- In den Datentypen, die mit `u` beginnen, werden natürliche Zahlen gespeichert, sonst ganze Zahlen.
- Die abschließende Zahl gibt den Speicherbedarf an. `uint8` benötigt z.B. 8-Bit. (Wertebereich  $0 \dots 2^8 - 1$ ).



# Integer

```
a = int8(20); b = int16(20); c = int8(20);  
a*c, a*b
```

```
ans = 127
```

```
??? Error using ==> mtimes
```

```
Integers can only be combined with integers  
of the same class, or scalar doubles.
```

```
a+0.2
```

```
ans = 20
```

```
a+0.5
```

```
ans = 21
```

```
a*1.54
```

```
ans = 31
```

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

# Structures

## Structures:

Strukturen sind eine Möglichkeit verschiedene Objekte in einer Datenstruktur zu bündeln.

## Beispiel: komplexe Zahlen

```
komp_Zahl.real=1;  
komp_Zahl.imag=1;  
komp_Zahl
```

```
komp_Zahl =
```

```
real: 1  
imag: 1
```

- Alternativ können Strukturen durch

```
struktur = struct('Feld1', <Wert1>, 'Feld2', <Wert2>, ...)
```

definiert werden.

- Ein Feld einer Struktur `struktur` kann durch

```
struc2 = rmfield( <struktur> , 'Feld')
```

gelöscht werden.

# Cell Arrays

## Cell Arrays:

Cell Arrays sind spezielle Matrizen, deren Einträge aus unterschiedlichen Datentypen bestehen können. Erzeugt werden sie durch geschweifte Klammern.

```
C = { 1:10, hilb(4); ...  
      'Hilbert Matrix', pi}
```

```
C =  
      [1x10 double]      [4x4 double]  
      'Hilbert Matrix'   [      3.1416]
```

# Befehle für Cell Arrays

- Zugriff auf Cell-Arrays:

```
C{2,1}
```

```
ans =  
Hilbert Matrix
```

```
C{1,2}(2,3)
```

```
ans =  
0.2500
```

- Durch `celldisp(C)` wird der Inhalt von `C` dargestellt.
- `cellplot(C)` stellt `C` grafisch dar.

## 1 Programmieren

- Schleifen
- Bedingungen
- Allgemeines

## 2 Datenstrukturen

- Zahlen
- Container
- Chars und Strings

# Characters (char) - Zeichen

- Darstellung durch Integer
- Die Werte zwischen 0 und 128 entsprechen den ASCII Werten.
- 2 Bytes Speicherbedarf  $\Rightarrow$  Zahl zwischen 0 und  $2^{16} - 1$

```
s = 'd'
```

```
s = d
```

```
s1 = double(s)
```

```
s1 = 100
```

```
s2 = char(100)
```

```
s2 = d
```



# Strings - Vektor von Zeichen

Die Zeichen werden wiederum durch die ASCII Werte dargestellt.

```
s = 'AB6de*'
```

```
s =  
AB6de*
```

```
sd = double(s)
```

```
sd =  
    65    66    54   100   101    42
```

```
s2 = char(sd)
```

```
s2 =  
AB6de*
```

# Befehle für Strings

- Durch `strcat` werden Strings verbunden, z.B.

```
strcat('Hello', ' world')
```

```
ans = Hello world
```

- `num2str(x,n)` konvertiert `x` in einen String mit `n` signifikanten Stellen. (Default: `n = 4`)
- `int2str(x)` rundet `x` und konvertiert es in einen String.
- `strcmp(s,t)` vergleicht die Strings `s` und `t`.
- Durch `help strfun` erhält man eine Liste aller Befehle im Zusammenhang mit Strings.