# Einführung in Matlab

## Einheit 2

Jochen Schulz

Georg-August Universität Göttingen

6. September 2009

# Aufbau

# Aufbau

1. **Programmieren mit MATLAB**
   - Motivation
   - Skript-Files
   - Function-Files

2. **Programmieren - Teil II**
   - Schleifen
   - Bedingungen
   - Rekursionen

# Zwei-Punkt-Randwert-Aufgabe

Suche eine Funktion

$$u : [0,1] \quad \rightarrow \quad \mathbb{R},$$

so dass

$$\begin{aligned} -u''(x) &= f(x), \quad x \in (0,1) \\ u(0) &= u(1) = 0 \end{aligned}$$

Lösen mit Finite-Differenzen-Verfahren

Diskretisierung: $0 = x_0 < \cdots < x_n = 1$ mit $x_i = \frac{i}{n}$

# Implementierung in MATLAB

$n = 101$, $f \equiv 1$

```
>> x = 0:(1/101):1;
>> x_i = x(2:101);
>> A = diag(2*ones(1,100),0)...
   +diag(-1*ones(1,99),-1)...
   +diag(-1*ones(1,99),1);
>> F = (1/101)^2*ones(100,1);
>> z_i = A\F;
>> z = [0; z_i; 0];
>> plot(x,z,'r*-');
```

# Motivation

Probleme:

- Bei jeder Änderung von $n$ muss alles erneut im interaktiven Modus eingegeben werden.
- Abrufen der Befehle bei späteren Sitzungen ist kaum möglich.
- Bei komplexen Algorithmen wird es unübersichtlich.

Ausweg: Die Befehlsfolge wird in einer Datei abgelegt. MATLAB arbeitet dann sukzessive die einzelnen Kommandos ab.

# randwertaufgabe.m

```
%-----------------------------------
%     randwertaufgabe.m
%
%  berechnet mit Finiten Differenzen die Lösung u von
%  -u''=f in (0,1), u(0)=u(1)=0
%
%  Gerd Rapin           1.11.2003
%------------------------------------------

% Anzahl Stützstellen
n = 5;

% Erzeugen des Gitters
x = 0:(1/n):1;
x_i = x(2:n);
```

# randwertaufgabe.m

```
% Aufstellen des lin. Gls.
A = diag(2*ones(1,n-1),0)...
   +diag(-1*ones(1,n-2),-1)...
   +diag(-1*ones(1,n-2),1);
F = (1/n)^2*ones(n-1,1); % rechte Seite für f=1

% Lösen des lin. Gls.
z_i = A\F;

% Darstellen der Lösung
z = [0; z_i; 0];
plot(x,z,'r*-');
```

# Erzeugen eines Programms

- Starten des Editors: `>> edit datei_name` öffnet die Datei `datei_name`.
- Speichern der Datei mit Hilfe des Menüs: `File->Save` bzw. `File->Save As`.

Achtung: Alle MATLAB-Dateien haben die Endung '.m'. Man spricht deswegen auch von *m*-Files.

# Starten von Programmen

- Befindet man sich im selben Verzeichnis wie das Programm `name.m`, so kann man das Programm starten durch Eingabe von `name`.
- Danach durchsucht MATLAB die in `path` angegebenen Verzeichnisse nach dem Programm.
- Mit dem Befehl `addpath pfadname` kann man eigene Suchpfade hinzufügen.
- Durch `rmpath pfadname` kann man Suchpfade entfernen.

# Graph eines Polynoms

Aufgabe:

Zeichnen Sie den Graphen eines Polynoms

$$p(x) = \sum_{i=0}^{N} a_i x^i, \quad a_i \in \mathbb{R}$$

Problem:

Zu Werten $(x_i)_{i=1}^n$ muß man $(p(x_i))_{i=1}^n$ berechnen, d.h. Funktionswerte müssen sehr oft berechnet werden.

Lösung:

Es gibt Funktionen in MATLAB.

# Skalare Version

```
function y=ausw_poly1(a,x)
%--------------------------------------------------
% ausw_poly berechnet den Funktionswert von
%           p(x)=a_1 +a_2 x + a_3 x^2+ ... +a^n x^(n-1)
%           INPUT:   a Vektor der Koeffizienten
%                    x  auszuwertender Punkt
%           OUTPUT: y  Funktionswert (y=p(x))
%  Gerd Rapin              1.11.2003
%----------------------------------------------------

n = length(a);
aux_vector = x.^(0:n-1);
y = aux_vector*transpose(a);
```

# Vektorielle Version

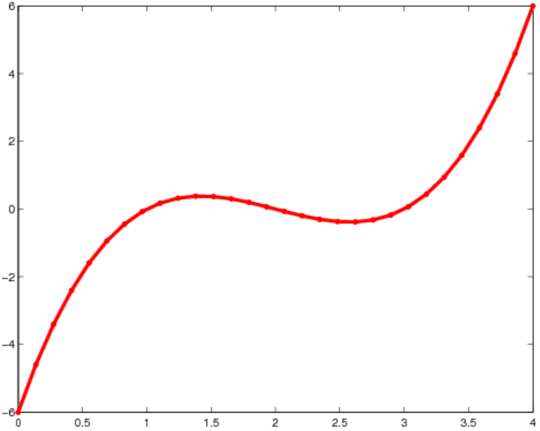
```
function y=ausw_poly2(a,x)
%----------------------------------------------------
% ausw_poly berechnet den Funktionswert von
%           p(x)=a_1 +a_2 x + a_3 x^2+ ... +a_n x^(n-1)
%           INPUT:  a Vektor der Koeffizienten
%                   x Vektor der auszuwertenden Punkte
%           OUTPUT: y Vektor der Funktionswerte
%  Gerd Rapin      1.11.2003
%------------------------------------------------------

n = length(a);
k = length(x);
A = repmat(transpose(x),1,n);
B = repmat(0:(n-1),k,1);

y = (A.^B)*transpose(a);
```

# Plotten des Polynoms

```
%-----------------------------------
%     plot_poly.m
%
%  zeichnet den Graphen eines Polynoms
%
%
%  Gerd Rapin             1.11.2003
%------------------------------------------

% Koeffizienten
a = [9 0 -10 0 1];

x = linspace(0,4,30); % Betrachte [0,4]
y = ausw_poly2(a,x);

% Plotten
plot(x,y,'r*-','LineWidth',3,'MarkerSize',4)
```

# Plotten des Polynoms

$$p(x) = (x-1)(x-2)(x-3)$$



# Aufbau

# Struktur von Skript-Files

- Skript-Files bestehen aus einer Sequenz von Befehlen, die nacheinander abgearbeitet werden.
- Files werden mit der Endung '.m' gespeichert.
- Gestartet wird das Programm `name.m` durch Eingabe von `name`.
- Kommentare beginnen mit `%`.

# Struktur von Skript-Files II

- Am Anfang des Files soll als Kommentar der Name des Programms, eine kurze Beschreibung, Name des Autors und das Erstellungsdatum stehen.
- operiert auf Daten im *Workspace*.
- Beschreibung des Skript-Files erhält man mit

```
>> help plot_poly
 ------------------------------------

      plot_poly.m
   zeichnet den Graphen eines Polynoms
   Gerd Rapin             1.11.2003
 -------------------------------------------
```

# Aufbau

1. **Programmieren mit MATLAB**
   - Motivation
   - Skript-Files
   - Function-Files
2. **Programmieren - Teil II**
   - Schleifen
   - Bedingungen
   - Rekursionen

# Struktur von Function-Files

Beispiel: 'my-file.m'

```
function [Out_1,..,Out_k]=my-file(In_1,..,In_l)
% Beschreibung der Funktion
 ..
Out_1=..
 ..
Out_k=..
```

Wichtig: Funktionsname muss identisch sein mit dem Dateinamen.

# Function-Files

- Funktionen sind mit Kommentaren zu versehen, was das Programm macht, welche Input- und Output-Argumente es hat, und wann und von wem es erstellt wurde.
- Variablen werden nur lokal gehalten; die Variablen des Workspace sind innerhalb des Workspace nicht verfügbar; im Programm definierte Variablen werden nicht im Workspace gespeichert.
- Soll keine Variable zurückgegeben werden, so besteht die erste Zeile aus

```
function myfile(In_1,..,In_k)
```

# Verwalten von m-Files

- `doc name` startet das grafische Hilfefenster mit einem ausführlichen Hilfetext zu dem jeweiligen Programm.
- `lookfor name` sucht nach dem Stichwort `name` in den Kommentaren zu den Funktionen. Ansonsten kann auch das grafische Hilfefenster zur Rate gezogen werden.
- `what` zeigt die m-Files im aktuellen Verzeichnis an.
- `type name` zeigt den Inhalt von `name.m` im 'Command Window' an.
- `which name` gibt den genauen Pfad an, in dem die Funktion `name.m` gespeichert ist.

# Priorität beim Programmaufruf

1. Testet, ob der Name eine Variable ist.
2. Testet, ob der Name eine Unterfunktion ist. Eine Unterfunktion ist ein Programm, das in derselben Datei wie der Aufruf steht.
3. Testet, ob das Programm im aktuellen Verzeichnis steht.
4. Testet, ob der Name eine *private function* ist.
5. Testet, ob das Programm im Suchpfad enthalten ist.

# Aufbau

1. **Programmieren mit MATLAB**
   - Motivation
   - Skript-Files
   - Function-Files

2. **Programmieren - Teil II**
   - Schleifen
   - Bedingungen
   - Rekursionen

# Gültigkeitsbereich von Variablen

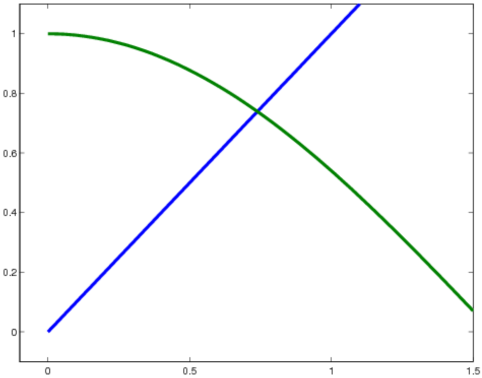
- Variablen in Skript-Files benutzen den globalen Workspace, d.h. bereits vorhandene Variablen können direkt benutzt oder überschrieben werden. Sie sind gültig bis sie explizit gelöscht werden.
- Variablen in Function-Files sind nur innerhalb der Funktion definiert und werden bei Verlassen der Funktion gelöscht. Variablen des globalen Workspace können nicht benutzt werden.

# Aufbau

# Fixpunkt

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

$$x_f = \cos(x_f)$$

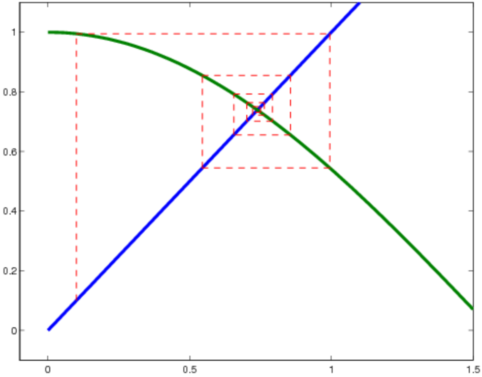


# Fixpunkt-Iteration

Fixpunkt-Iteration

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

bei geeignetem Startwert $x_0$.



# Implementierung

```
% 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

- Durch `figure` wird ein Grafik-Fenster gestartet.
- Mittels `hold on` werden alle Grafiken in einem Fenster übereinander gezeichnet.
- Im Standardmodus wird bei jedem Grafikbefehl die bestehende Grafik gelöscht und durch die neue Grafik ersetzt.
- Mittels `hold off` wird zurück in den Standardmodus gewechselt.

# for - Schleife

```
for variable = Ausdruck
  Befehle
end
```

Bemerkungen:

- Der Ausdruck ist normalerweise von der Form `i:s:j`.
- Die *Befehle* werden eingerückt.
- auch weitere Schleifen-Konstrukte wie `while` und `switch` sind verfügbar.

# 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
```

# Vandermonde-Matrix

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

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

# Implementierung 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
```

## Aufbau

# Quadratische Gleichung

$$\left\{ \begin{array}{l} \text{Suche } x \in \mathbb{R}, \text{ so dass} \\ x^2 + px + q = 0 \end{array} \right.$$

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

- Es gibt in MATLAB logische Variablen. Der 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]
```

| | | |
|---|---|---|
| `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

| | | | |
|---|---|---|---|
| & | logisches und | ~ | logisches nicht |
| \| | logisches oder | xor | 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
```

# Bedingung

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 $ggT(a, b) = ggT(a, b - a)$ 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
```

# *break* and *continue*

- 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
```

# Aufbau

# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen.
Bei jedem Aufruf wird ein neuer lokaler Workspace erzeugt.

**Beispiel:** Fakultät: $n! = \mathsf{fak}(n)$

$$\begin{aligned} n! &= n(n-1)! = n\,\mathsf{fak}(n-1) \\ &= n(n-1)\,\mathsf{fak}(n-2) \\ &= \cdots = n(n-1)\cdots 1 \end{aligned}$$

# Fakultät - rekursiv

```
function res = fak(n)
if (n == 1)
    res = 1;
else
    res = n*fak(n-1);
end
```

# Fakultät - direkt

```
function fak = fak_it(n)
% fakultaet   berechnet zu einer gegebenen natuerlichen Zahl n
%             die Fakultaet n!:=1*2*...*n
%
%              INPUT: natuerliche Zahl n
%              OUTPUT: Fakultaet fak
%   Gerd Rapin  10.11.

fak = 1;
for i = 1:n
  fak = fak*i;
end;
```

# Fakultät - Zeitvergleich

```
% fak_vergleich.m
% iterativ
tic
for i = 1:100
  fak_it(20);
end
time1 = toc;
fprintf('\nZeitverbrauch direktes Verfahren: %f',time1);
% rekursiv
tic
for i = 1:100
  fak(20);
end
time2 = toc;
fprintf('\nZeitverbrauch rekursives Verfahren: %f\n',time2);
```

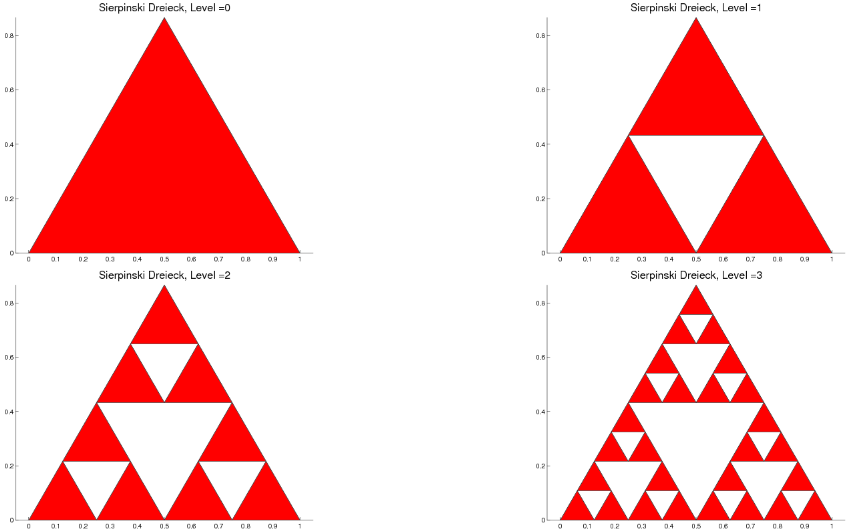
# rekursive Implementierung GGT

```
function [a,b] = ggt_rekursiv(a,b)
% ggt_rekursiv berechnet den goessten
% gemeinsamen Teiler (ggT)
if a~=b
  if a>b
    a = a-b;
  else
    b = b-a;
  end;
  [a,b] = ggt_rekursiv(a,b);
end;
```

# Sierpinski Dreieck

- Wir beginnen mit einem Dreieck mit Eckpunkten $P_a$, $P_b$ und $P_c$.
- Wir entfernen daraus das Dreieck, das durch die Mittelpunkte der Kanten entsteht.
- Die verbliebenden drei Dreiecke werden der gleichen Prozedur unterzogen.
- Diesen Prozess können wir rekursiv wiederholen.
- Das Ergebnis ist das Sierpinski Dreieck.

# Sierpinski Dreieck



# Implementierung

```
%  sierpinski_plot.m
level=7;

ecke1=[0;0];
ecke2=[1;0];
ecke3=[0.5; sqrt(3)/2];

figure; axis equal;
hold on;
sierpinski(ecke1,ecke2,ecke3,level);
hold off;
title(['Sierpinski Dreieck, Level =' ...
       num2str(level)],'FontSize',16);
```

# Implementierung

```
function sierpinski(ecke1,ecke2,ecke3,level)
% Teilt das Dreieck auf in 3 Dreiecke (level>0)
% Plotten des Dreiecks (level=0)

if level == 0
    fill([ecke1(1),ecke2(1),ecke3(1)],...
     [ecke1(2),ecke2(2),ecke3(2)],'r');
else
    ecke12 = (ecke1+ecke2)/2;
    ecke13 = (ecke1+ecke3)/2;
    ecke23 = (ecke2+ecke3)/2;
    sierpinski(ecke1,ecke12,ecke13,level-1);
    sierpinski(ecke12,ecke2,ecke23,level-1);
    sierpinski(ecke13,ecke23,ecke3,level-1);

end;
```

# Zeichnen von Polygonen

Ein Polygon sei durch die Eckpunkte $(x_i, y_i)_{i=1}^{n}$ gegeben. Dann kann er in MATLAB durch den Befehl

```
fill(x,y,char)
```

dargestellt werden. `char` gibt die Farbe des Polygons an, z.B. rot wäre 'r'.

# Warnung

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;
```

# Guter Stil in MATLAB

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

# Guter Stil in MATLAB

- 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.
- Potentielle Fehler sollten, soweit möglich, aufgefangen werden. Speziell sollten die Eingabeparameter der Funktionen geprüft werden.