

Einführung in Matlab - Einheit 2

Numerische Lineare Algebra, Programmieren

Jochen Schulz

Georg-August Universität Göttingen 

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

Die p -Norm eines Vektors $x = (x_1, \dots, x_n)$

$$\|x\|_p := \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

(definiert für $p \geq 1$).

- in MATLAB: `norm(x,p)` (Default: $p = 2$)
- $p = \infty$ entspricht der Maximum-Norm

$$\|x\|_\infty = \max_{i=1,\dots,n} |x_i|.$$

Seien $A \in \mathbb{C}^{n \times m}$ und $p \geq 1$. Die *Matrixnorm* ist definiert durch

$$\|A\|_p = \sup_{x \in \mathbb{C}^m \setminus \{0\}} \frac{\|Ax\|_p}{\|x\|_p}.$$

- In MATLAB: `norm(A,p)` (Default $p = 2$).
- $p = \infty$ kann charakterisiert werden durch

$$\|A\|_\infty = \max_{1 \leq j \leq m} \sum_{i=1}^n |a_{ij}|, \quad \text{Zeilensummennorm.}$$

Kondition einer quadratischen Matrix A :

$$\text{cond}_p(A) := \|A\|_p \|A^{-1}\|_p.$$

- In MATLAB: `cond(A,p)` (Default $p = 2$)
- Es gilt $\text{cond}_p(A) \geq 1$.
- Die Kondition mißt die Empfindlichkeit der Lösung x von $Ax = b$ gegenüber Störungen von A und b .
- Ist $\text{cond}_p(A) \gg 1$, so ist die Matrix beinahe singular. Die Matrix ist *schlecht konditioniert*.

- Vektornormen für $x = (1/100)(1, 2, \dots, 100)$

```
>> x = (1:100)/100; [norm(x,1) norm(x,2) norm(x,inf)]  
ans =      50.5000      5.8168      1.0000
```

- Matrixnorm für die Hilbert-Matrix $H = (\frac{1}{i+j-1})_{ij}$

```
>> H = hilb(10); [norm(H,1) norm(H,2) norm(H,inf)]  
ans =      2.9290      1.7519      2.9290
```

- Kondition der Hilbert-Matrix

```
>> H = hilb(10); [cond(H,1) cond(H,2) cond(H,inf)]  
ans =  
      1.0e+13 *  
      3.5354      1.6025      3.5354
```

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

Lineare Gleichungssysteme

Seien $A \in \mathbb{C}^{n \times n}$ und $b \in \mathbb{C}^n$. Das lineare Gleichungssystem

$$Ax = b$$

wird in MATLAB gelöst durch $x=A \backslash b$.

```
>> x = ones(5,1); H = hilb(5); b = H*x; y = (H\b) '
y =
    1.0000    1.0000    1.0000    1.0000    1.0000
```

Warnung: Benutze nie $x=\text{inv}(A)*b$, da das Berechnen von A^{-1} sehr aufwendig sein kann.

Was bedeutet $A \setminus b$?

MATLAB berechnet die LU-Zerlegung von A (Gaussverfahren):

- obere Dreiecksmatrix U
- untere Dreiecksmatrix L mit Einsen auf der Diagonalen

so dass $PA = LU$ gilt (P Permutationsmatrix).

Dann wird das LGS durch Rückwärts- und Vorwärtseinsetzen gelöst
($Lz = Pb$, $Ux = z$)

```
>> [L,U,P]=lu(hilb(5)); norm(P*hilb(5)-L*U)
ans = 2.7756e-17
```

Inverse, Determinante

- Berechnung der Inversen

```
>> A=pascal(3)
```

```
A =
```

1	1	1
1	2	3
1	3	6

```
>> X=inv(A)
```

```
X =
```

3	-3	1
-3	5	-2
1	-2	1

- Berechnung der Determinante

```
>> det(A)
```

```
ans = 1
```

Pseudoinverse

(Moore-Penrose) Pseudoinverse

Sei A singulär, Bestimme X so dass

$$AXA = A, XAX = X, (XA)^* = XA, (AX)^* = AX$$

```
>> pinv(ones(3,3))  
ans =  
    0.1111    0.1111    0.1111  
    0.1111    0.1111    0.1111  
    0.1111    0.1111    0.1111
```

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

Zwei-Punkt-Randwert-Aufgabe

Suche eine Funktion

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

so dass

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

Problem: Es kann i.A. keine geschlossene Lösungsdarstellung angegeben werden.

Ausweg: Approximation der Lösung.

Finite Differenzen Verfahren

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

Differenzenquotient:

$$u''(x_i) \sim \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1}))}{h^2}, \quad h := \frac{1}{n}$$

Einsetzen in $-u''(x) = e^x$ ergibt

$$-u(x_{i-1}) + 2u(x_i) - u(x_{i+1})) = h^2 e^{x_i}, \quad i = 1, \dots, n-1$$

Randbedingungen $\Rightarrow u(x_0) = u(x_n) = 0$.

\Rightarrow Lineares Gleichungssystem für $u(x_1), \dots, u(x_{n-1})$.

Diskretes Problem

Setze $z = (z_1, \dots, z_{n-1})^t = (u(x_1), \dots, u(x_{n-1}))^t$.

Löse das Gleichungssystem $Az = F$ mit

$$A := \begin{pmatrix} 2 & -1 & & & 0 \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ 0 & & & -1 & 2 \end{pmatrix}, \quad F := h^2 \begin{pmatrix} e^{\frac{1}{n}} \\ \vdots \\ e^{\frac{n-1}{n}} \end{pmatrix}.$$

Lösung für $n = 21$

- Zerlegung des Intervalls $[0, 1]$

```
x = 0:(1/21):1
```

- Eliminieren der Randpunkte

```
x_i = x(2:21)
```

- Erzeugen der Matrix A (Übungsaufgabe)

Lösung für $n = 21$

- Berechnen der rechten Seite:

```
F = (1/21)^2*transpose(exp(x_i));
```

- Lösen des linearen Gl.

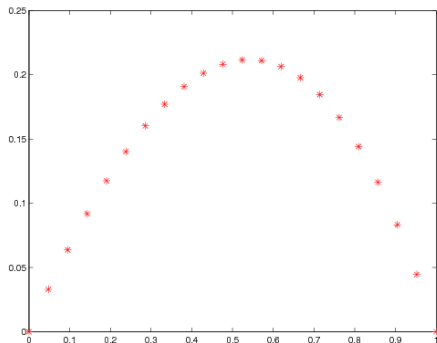
```
z_i = A\F;
```

- Zufügen der Werte am Rand

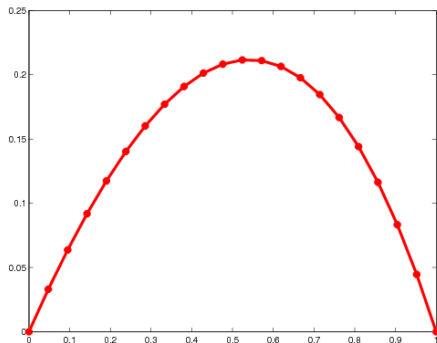
```
>> z = [0; z_i; 0];
```

Lösung für $n = 21$

```
plot(x,z,'r*','MarkerSize',8)
```



```
plot(x,z,'r*-', 'LineWidth',3, 'MarkerSize',8)
```



1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

Eigenwert

Sei $A \in \mathbb{C}^{n \times n}$. $\lambda \in \mathbb{C}$ ist Eigenwert von A , falls ein Vektor $x \in \mathbb{C}^n$ ungleich 0 existiert, so dass $Ax = \lambda x$ gilt. x heißt Eigenvektor.

- $x = \text{eig}(A)$
berechnet die Eigenwerte von A und schreibt sie in den Vektor x .
- $[V, D] = \text{eig}(A)$
 D ist eine Diagonalmatrix mit den Eigenwerten auf der Diagonalen.
Die Spalten von V bilden die zugehörigen Eigenvektoren.

Weitere Zerlegungen

- **QR-Zerlegung:** $[Q,R]=\text{qr}(A)$

$m \times n$ - Matrix A eine Zerlegung $A = QR$ erzeugt, (Q eine unitäre $m \times m$ -Matrix, R eine obere $m \times n$ Dreiecksmatrix).

- **Singularwertzerlegung:** $[U,S,V]=\text{svd}(A)$

$A = U\Sigma V^*$. ($\Sigma \subset \mathbb{C}^{m \times n}$ eine Diagonalmatrix
 $U \subset \mathbb{C}^{m \times m}$, $V \subset \mathbb{C}^{n \times n}$ unitäre Matrizen).

- **Cholesky-Zerlegung:** $R=\text{chol}(A)$

$A = R^*R$ zu einer hermiteschen, positiv definiten Matrix A (R ist eine obere Dreiecksmatrix mit reellen, positiven Diagonalelementen).

- LGS können auch mit Hilfe iterativer Verfahren gelöst werden, z.B. `gmres`, `pcg`, `bicgstab`.
- $A \in \mathbb{C}^{n \times m}$, $n \neq m$ bei $A \setminus b$:
 - $n > m$ (überbestimmter Fall): Least-Square Lösung, d.h. der Ausdruck `norm(A*x-b)` wird minimiert.
 - $n < m$ (unterbestimmter Fall): Grundlösung.

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

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.

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

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.

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
```

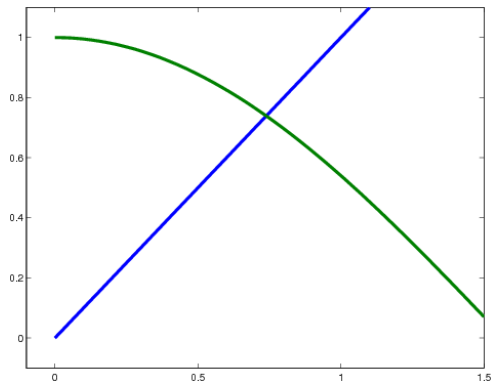
- Matrix als *Ausdruck*

```
>> 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

$$x_f = \cos(x_f)$$

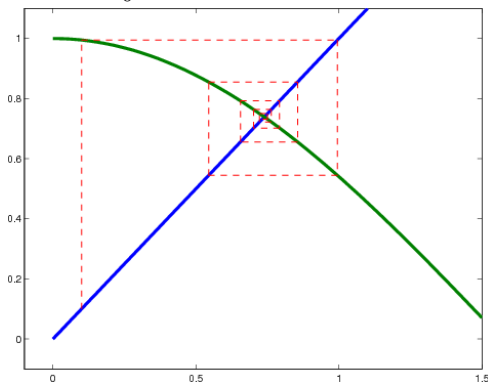


Fixpunkt-Iteration

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;
```

- 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.

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
```

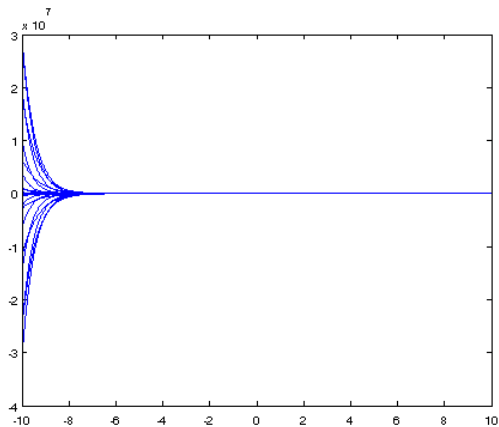
Berechnung von e

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 - Figure



1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

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

- 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]
```

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

$\&$	logisches und	\sim	logisches nicht
$ $	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 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

Rekursive Funktionen

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

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

$$\begin{aligned}n! &= n(n-1)! = n \text{ fak}(n-1) \\&= n(n-1) \text{ fak}(n-2) \\&= \dots = n(n-1) \dots 1\end{aligned}$$

```
function res = fak(n)
% fakultaet    berechnet zu einer gegebenen natuerlichen
%              Zahl n
%              die Fakultaet n!:=1*2*...*n (rekursiv)
%
%              INPUT: natuerliche Zahl n
%              OUTPUT: Fakultaet fak
%      Jochen Schulz 3.9.2010
if (n == 1)
    res = 1;
else
    res = n*fak(n-1);
end
```

```
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;
```

```
% 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

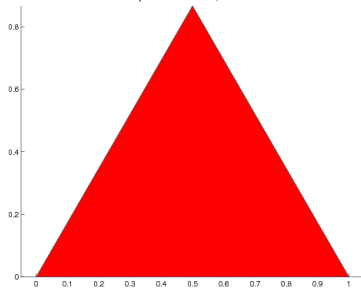
```
function [a,b] = ggt_rekursiv(a,b)
% ggt_rekursiv berechnet den groessten
% gemeinsamen Teiler (ggT)
if a~=b
    if a>b
        a = a-b;
    else
        b = b-a;
    end;
    [a,b] = ggt_rekursiv(a,b);
end;
```

Sierpinski Dreieck

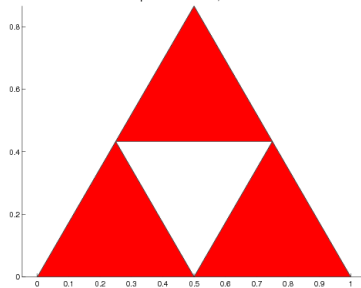
- Wir beginnen mit einem Dreieck mit Eckpunkten P_a , P_b und P_c .
- Wir entfernen daraus das Dreieck, das durch die Mittelpunkte der Kanten entsteht.
- Die verbliebenden drei Dreiecke werden der gleichen Prozedur unterzogen.
- Diesen Prozess können wir rekursiv wiederholen.
- Das Ergebnis ist das Sierpinski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

Sierpinski Dreieck

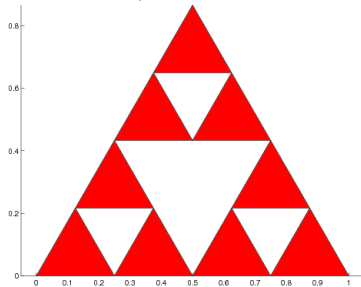
Sierpinski Dreieck, Level =0



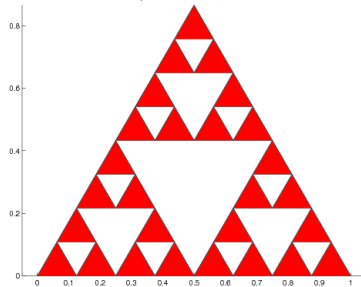
Sierpinski Dreieck, Level =1



Sierpinski Dreieck, Level =2



Sierpinski Dreieck, Level =3



```
% 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'.

1 Numerische Lineare Algebra

- Normen
- Lösen linearer Gleichungssysteme
- Anwendung: Zwei-Punkt-Randwert-Aufgabe
- Bestimmung von Eigenwerten

2 Programmieren

- Schleifen
- Bedingungen
- Rekursionen
- Allgemeines

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. Insbesondere sollten die Eingabe- und Ausgabevariablen genau beschrieben werden.
- 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.