Einführung in MATLAB

Mathematische Software (mit Programmierumgebung):

MAPLE : *Symbolisches* Berechnungswerkzeug (sog. Comuteralgebra-System)
MATLAB: *Numerisches* Berechnungswerkzeug

Grundsätzliche Datenstruktur in MATLAB: Die *Matrix* (MATLAB $\triangleq$ **MAT**rix **LAB**oratory)

Eine Zahl ist insbesondere eine (1×1) – Matrix.
Eine Zeile mit m Elementen ist eine $(1 \times m)$ – Matrix.
Eine Spalte mit n Elementen ist eine $(n \times 1)$ – Matrix.

MATLAB kann nur *endlich viele* Zahlen (sog. Maschinenzahlen) darstellen.
Rundungsfehler sind daher die Folge!

Beispiel: $1{,}2 - 0{,}4 - 0{,}4 - 0{,}4 = 0$
Ergebnis in MATLAB: `-1.1102e-016`

Ursache: Dual gilt z.B.: $0{,}4 = 0,0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ \ldots$ usw
$\Rightarrow$ 0,4 ist dual im Computer nur *näherungsweise* darstellbar

Maschinengenauigkeit in MATLAB: eps = 2.2204e-016.
Maschinengenauigkeit = Abstand von 1 zur nächsthöheren Maschinenzahl.

I. Lineare Gleichungssysteme mit MATLAB

Ein lineares Gleichungssystem lautet in *Matrixschreibweise* so:

ℓ-te Spalte:
Spaltenvektor mit Spaltenindex ℓ

k-te Zeile: Zeilenvektor mit Zeilenindex k

$$\begin{pmatrix} a_{11} & \cdots & a_{1\ell} & \cdots & a_{1m} \\ \vdots & & \vdots & & \vdots \\ a_{k1} & \cdots & a_{k\ell} & \cdots & a_{km} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & a_{n\ell} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_\ell \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ \vdots \\ b_n \end{pmatrix}$$

$$\boldsymbol{A} \quad * \quad \boldsymbol{x} \quad = \quad \boldsymbol{b}$$

I.1. Zur Lösbarkeit linearer Gleichungssysteme:

Rang einer Matrix:

Rang(A) $\underset{def.}{=}$ Anzahl linear unabhängiger Spalten von A

Es gilt: $\boldsymbol{A\,x = b}$ **lösbar** $\Leftrightarrow$ **Rang (A) = Rang (A,b)**

Betrachte A als *lineare Abbildung*: $A: \mathbb{R}^m \to \mathbb{R}^n$

Nullraum (Kern) einer linearen Abbildung:

Null(A) $\underset{def.}{=}$ Kern(A) $\underset{def.}{=}$ $\{x \,/\, A\,x = 0\}$

Es gilt: **dim Kern(A) = m – Rang(A)**

Lösungsmenge L von $Ax = b$:
Ist x_0 eine spezielle Lösung von $Ax = b$, so gilt $\mathbf{L = x_0 + Kern(A) = \{ x_0 + x \;/\; A\,x = 0\}}$

I.2. Lösungsmethoden mit MATLAB

Beispiel 1: Zeilenzahl $n \neq$ Spaltenzahl m (hier $m = 5$ und $n = 3$)

MATLAB-Eingabe im "**Command Window**" (jeweils mit RETURN bestätigen):

```
>> A = [ 2 3 4 5 -8 ; 3 -6 8 5 0 ; 0 0 7 0 -7];
>> b = [2 ; 6 ; 7];
>> Rang_A = rank(A)
>> Rang_Ab = rank([A,b])
```

Es ist Rang(A) = Rang(A,b) = 3 $\Rightarrow$ Ax = b ist lösbar
Bestimme *eine* Lösung x_0 mit MATLAB-Befehl **`xo = pinv(A)*b`**
Bestimme *Nullraum* von A mit MATLAB-Befehl **`Kern_A = null(A)`**

Ausgabe für xo:

```
xo =
   -0.1204
   -0.3600
    0.7400
   -0.3438
   -0.2600
```

Ausgabe für den Nullraum von A:

```
Kern_A =
   -0.8425    0.3271
    0.1987    0.6587
    0.2193    0.4668
    0.3932   -0.1526
    0.2193    0.4668
```

Die MATLAB-Ausgabe für xo ist ein Spaltenvektor mit 5 Einträgen.
Wegen dim Kern(A) = m – Rang(A) = 5 – 3 = 2 gibt es also in der MATLAB-Ausgabe 2 linear unabhängige Spaltenvektoren, welche den Nullraum "aufspannen".

Beispiel 2: Zeilenzahl n = Spaltenzahl m (hier $m = n = 4$)

MATLAB-Eingabe im "**Command Window**" (jeweils mit RETURN bestätigen):

```
>> A = [ 2 3 4 5 ; 3 -6 8 5 ; 0 0 7 -7 ; 1 2 3 4];
>> b = [1; 3 ; -6 ; 7];
>> c = rank(A) - rank([A,b])
>> det_A = det(A)  %Determinante von A bestimmen
```

Es ist c = 0 $\Rightarrow$ $A\,x = b$ ist lösbar

det(A) = 0 $\Leftrightarrow$ A ist singulär $\Rightarrow$ Verfahre gemäß Beispiel 1

det(A) $\neq$ 0 $\Leftrightarrow$ A ist regulär $\Leftrightarrow$ $A\,x = b$ ist *eindeutig* lösbar und es gilt $x = A^{-1}\,b$.

A^{-1} bezeichnet die *Inverse* von A. Sie ist *eindeutig* bestimmt.

Die *eindeutige* Lösung x ergibt sich mit dem MATLAB-Befehl **`x = inv(A)*b`** oder **`x = A\b`** .

Ausgabe für die Determinante: det_A = – 364
Ausgabe für die eindeutige Lösung x :

```
x =
  -10.7582
   -0.4505
    2.1758
    3.0330
```

Die Lösungsausgabe ist also ein Spaltenvektor mit vier Komponenten.

Anmerkung:

Ist Rang(A) $\neq$ Rang(A,b) , so hat Ax = b keine Lösungen.
Mit dem Befehl **`x = pinv(A)*b`** gibt MATLAB dann einen Spaltenvektor aus, der die Gleichung $A\,x = b$ zwar nicht erfüllt aber wenigstens $|\,A\,x - b\,|$ minimiert.

`pinv(A)` bezeichnet die *Pseudoinverse* von A. *Jede beliebige* Matrix besitzt *genau eine* Pseudoinverse.
Ist A *regulär*, dann ist pinv(A) = inv(A). Der Begriff *Pseudoinverse* kann hier nicht weiter vertieft werden.

Bei Eingaben ist zu beachten:

Nach Bestätigung eines MATLAB-Befehls mit RETURN wird der Befehl ausgeführt und das Ergebnis ausgegeben. Wird der Befehl mit einem *Semikolon* abgeschlossen, so wird der Befehl ebenfalls ausgeführt aber das Ergebnis *nicht angezeigt*.

Die voreingestellte Oberfläche von MATLAB kann verändert werden. Die Voreinstellung kann mittels *Desktop* → *Desktop Layout* → *Default* wiederhergestellt werden

Matrizen werden in eckigen Klammern angegeben. Die Elemente der Zeilen sind durch ein Leerzeichen oder alternativ durch ein Komma zu trennen. Die unterschiedlichen Zeilen sind durch ein Semikolon zu trennen.
Wie bereits oben geschehen, kann jeder Matrix ein Name (Variablenname) zugeordnet werden.
Der Befehl `A =[1 2;3 4]` bewirkt, dass die Matrix `[1 2;3 4]` mit dem Namen A angesprochen werden kann. Andernfalls weist MATLAB der Matrix den Namen **`ans`** ("answer") zu.

I.3. Rundungsfehler bei Matrizen und linearen Gleichungssystemen

Vorbemerkung:

MATLAB gibt Zahlen in der Regel mit 4 Nachkommastellen aus (Voreinstellung).
In MATLAB sind dabei Vor- und Nachkommastellen durch einen *Punkt* zu trennen.
Mit dem MATLAB-Befehl **`format long`** erreicht man 15 Nachkommastellen.
Mit dem MATLAB-Befehl **`format short`** erreicht man wieder 4 Nachkommastellen.
Die Voreinstellung ist meist die beste Wahl.

Beispiel: $\underbrace{\begin{pmatrix}1 & 1\\ 1 & 1{,}0000001\end{pmatrix}}_{A}\underbrace{\begin{pmatrix}x_1\\ x_2\end{pmatrix}}_{x}=\underbrace{\begin{pmatrix}2\\ 2{,}0000001\end{pmatrix}}_{b}$

Offenbar ist $x_1 = x_2 = 1$ die eindeutige Lösung , denn es ist det(A) = 0,0000001 ≠ 0 (jedoch nahe bei Null!).
Mit der anschließenden Befehlsfolge gibt MATLAB diese Lösung auch aus:

```
format short; A =[1 1;1 1.0000001]; b =[2;2.0000001]; x = A\b
```

Die Ausgabe ist:

```
x =
    1.0000
    1.0000
```

Bei der Einstellung **`format long`** ergibt sich aber

```
x =
   1.000000002220446
   0.999999997779554
```

In der Einstellung **`format short`** wird dieser Effekt offenbar kaschiert.
Ursache: Weil det(A) "fast" gleich Null ist, ist das obige Beispiel "schlecht konditioniert".
Genauer: Die Matrix A ist schlecht konditioniert.
Der MATLAB-Befehl **`cond(A)`** liefert das Ergebnis ans = 4.000000195305916e+007 .
Es ist also $10^{q} \le \text{cond(A)} \le 10^{q+1}$, wobei hier $q = 7$ ist.
Dies bedeutet (vereinfacht ausgedrückt), dass im Ergebnis für *x* ein Verlust von 7 Stellen möglich ist. Genau dies ist im obigen Ergebnis für *x* bei der Einstellung **`format long`** erkennbar.
Der Begriff *Kondition* kann hier nicht weiter vertieft werden.

I.4. Rechenoperationen mit Matrizen

```
>> A=[2 4;-6 7]
A =
     2     4
    -6     7
```

```
>> B=[3 -9;1 0]
B =
     3    -9
     1     0
```

```
>> AB=A*B
AB =
    10   -18
   -11    54
```

```
>> A_plus_B=A+B
A_plus_B =
     5    -5
    -5     7
```

```
>> A_min_B=A-B
A_min_B =
    -1    13
    -7     7
```

```
>> k=2;kA=k*A
kA =
     4     8
   -12    14
```

Die obigen Operationen beschreiben die üblichen Matrixoperationen *Multiplikation*, *Addition*, *Subtraktion* und *Skalarmultiplikation* (Multiplikation einer Matrix mit einer Zahl).

Die folgenden Operationen gehen von den obigen Matrizen A und B aus und bauen dann aufeinander auf:

```
>> detA=det(A)
detA =
    38
```

```
>> Inv_A=inv(A)
Inv_A =
    0.1842   -0.1053
    0.1579    0.0526
```

Ist die Determinante von A "deutlich ungleich Null", kann die Inverse A^{-1} mit dem Befehl inv(A) genau genug bestimmt werden.

```
>> Inv_AxB=A\B
Inv_AxB =
    0.4474   -1.6579
    0.5263   -1.4211
```

```
>> BxInv_A=B/A
BxInv_A =
   -0.8684   -0.7895
    0.1842   -0.1053
```

"Backslash-Division":
A\B bedeutet A^{-1} B = inv(A)*B
"Slash-Division":
B/A bedeutet BA^{-1} = B*inv(A)

```
>> C=[A;3 3]
C =
     2     4
    -6     7
     3     3
```

```
>> D=[3 3;A]
D =
     3     3
     2     4
    -6     7
```

Erzeugung von C:
Zeile `[3 3]` *unter*halb von A anlegen.
Erzeugung von D:
Zeile `[3 3]` *ober*halb von A anlegen.
Definiert man vorher `z=[3 3]`, so können auch die Befehle `D =[z;A]` und `C =[A;z]` eingegeben werden.

```
>> E=[D,[3;3;3]]
E =
     3     3     3
     2     4     3
    -6     7     3
```

```
>> F=[[3;3;3],D]
F =
     3     3     3
     3     2     4
     3    -6     7
```

Erzeugung von E: Spalte `[3;3;3]` *rechts* von D anlegen.
Erzeugung von F: Spalte `[3;3;3]` *links* von D anlegen.
Mit `z=[3;3;3]` können auch die Befehle `E =[D,z]` und `F =[z,D]` eingegeben werden.

```
>> E(:,2)=[]
E =
     3     3
     2     3
    -6     3
```

```
>> E(1,:)=[]
E =
     2     3
    -6     3
```

Umgekehrter Vorgang angewandt auf Matrix E:
Zuerst 2. Spalte entfernen.
Dann 1. Zeile entfernen.
Analog können in Matrizen beliebige Zeilen oder Spalten entfernt werden.

```
>> FQuadrat=F*F
FQuadrat =
    27    -3    42
    27   -11    45
    12   -45    34
```

```
>> FQuadrat=F^2
FQuadrat =
    27    -3    42
    27   -11    45
    12   -45    34
```

Quadratur im Sinne der bekannten Matrixmultiplikation:
Statt `F*F` kann auch `F^2` eingegeben werden.

```
>> F_plus_2=F+2
F_plus_2 =
     5     5     5
     5     4     6
     5    -4     9
```

```
>> F_minus_2=F-2
F_minus_2 =
     1     1     1
     1     0     2
     1    -8     5
```

Die Addition/Subtraktion zwischen Matrizen und Zahlen (hier z.B. 2) ist i.a. nicht definiert. MATLAB gestattet jedoch solche Befehle: Zu (Von) *jedem Matrixelement* wird die betreffende Zahl addiert (subtrahiert).

```
Zwei_plus_F=2+F
Zwei_plus_F =
     5     5     5
     5     4     6
     5    -4     9
```

```
>> Zwei_minus_F=2-F
Zwei_minus_F =
    -1    -1    -1
    -1     0    -2
    -1     8    -5
```

Analog verhält es sich bei der Bestimmung von 2+F und 2–F , wenn also in der Summe oder Differenz die Zahl vorne steht.

Rechenoperationen komponentenweise (sog. Punktoperationen)
Bei diesen Operationen werden die entsprechenden *Matrixelemente* (- *komponenten*) miteinander verknüpft. Die verknüpften Matrizen müssen dazu "gleich groß" sein, d.h. die Zeilen- und Spaltenzahl muss bei beiden Matrizen übereinstimmen. Man beachte die bereits oben definierten Beispielmatrizen:

```
>> A.*B
ans =
     6   -36
    -6     0
```

Es werden die *entsprechenden Elemente* von A und B *multipliziert*: a_{11} wird mit b_{11} multipliziert, a_{21} mit b_{21} usw. Es wird also $a_{ij}\, b_{ij}$ für $i = 1$ und 2 sowie $j = 1$ und 2 errechnet und diese Produkte sind dann die Elemente der Matrix **ans** .
In MATLAB werden solche komponentenweise Operationen mit *einem Punkt* vor dem Verknüpfungszeichen codiert.

```
>> C.*C
ans =
     4    16
    36    49
     9     9
```

```
>> C.^2
ans =
     4    16
    36    49
     9     9
```

Komponentenweise Matrizen-Quadratur

```
>> C./D
ans =
    0.6667    1.3333
   -3.0000    1.7500
   -0.5000    0.4286
```

```
>> C.\D
ans =
    1.5000    0.7500
   -0.3333    0.5714
   -2.0000    2.3333
```

Komponentenweise Matrizen-Division (Slash und Backslash)
Beachte:
`C.\D = D./C`

```
>> 1./F
ans =
    0.3333    0.3333    0.3333
    0.3333    0.5000    0.2500
    0.3333   -0.1667    0.1429
```

"Reziproke Matrix":
Es wird zu jedem Element f_{ij} von F der Kehrwert $1/f_{ij}$ errechnet, welche dann die Elemente von **ans** sind.

Addition, Subtraktion und Skalarmultiplikation sind ohnehin "Punktoperationen".
Daher sind in MATLAB Eingaben wie z.B. **A.+B** nicht zulässig. Man erhält eine Fehlermeldung.

Transposition von Matrizen

```
>> FT=F'
FT =
     3     3     3
     3     2    -6
     3     4     7
```

```
>> DT=D'
DT =
     3     2    -6
     3     4     7
```

Bei der Matrix-Transposition werden also Zeilen zu Spalten und Spalten zu Zeilen.
Die zu transponierende Matrix ist in MATLAB mit einem Hochkomma zu codieren.

Weitere Rechenoperationen (Man beachte die bereits oben definierten Beispielmatrizen)

Operation	Befehl	Beispiel	Ergebnis
Zeile z herausnehmen	`A(z,:)`	`A(1,:)`	`2 4`
Spalte s herausnehmen	`A(:,s)`	`A(:,2)`	`4` `7`
A rechts neben B	`[B,A]`	`[B,A]`	`3 -9  2  4` `1  0 -6  7`
A links neben B	`[A,B]`	`[A,B]`	`2  4  3 -9` `-6  7  1  0`

Der Doppelpunkt bedeutet jeweils "alle". `A(z,:)` heißt also z-te Zeile und *alle* Spaltenelemente in dieser Zeile.

Operation	Befehl	Beispiel	Ergebnis
Größe einer Matrix	`size(D)`	`size(D)`	`3 2`
Anzahl Zeilen	`size(D,1)`	`size(D,1)`	`3`
Anzahl Spalten	`size(D,2)`	`size(D,2)`	`2`

```
>> [Zeilenzahl,Spaltenzahl]=size(D)
Zeilenzahl =
     3
Spaltenzahl =
     2
```

Will man den Werten 3 und 2 sofort Variablennamen vergeben, so codiere man dies in MATLAB wie linkerhand in der 1. Zeile beschrieben. Die Bestätigung mit RETURN liefert das Ergebnis darunter.

Spezielle Matrizen

Spezielle Matrizen sind beispielsweise die Null-Matrix und die Einheitsmatrix:

```
>> M=zeros(3,3)
M =
     0     0     0
     0     0     0
     0     0     0
```

Erzeugung einer Nullmatrix mit Angabe der Zeilen- und Spaltenanzahl.

```
>> N=ones(3,4)
N =
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

Erzeugung einer Matrix mit Einsen mit Angabe der Zeilen- und Spaltenanzahl.

```
>> E4=eye(4)
E4 =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Erzeugung einer 4-reihigen Einheitsmatrix. Diese ist bekanntlich immer quadratisch.

```
>> x =(0:1:5)
x =
     0  1  2  3  4  5
>> l=length(x)
l =
     6
>> x(6)
ans =
     5
>> x(end)
ans =
     5
```

Diese Codierung erzeugt einen Zeilenvektor (1-zeilige Matrix) gelesen: 0 step 1 until 5. Mit **`length`** misst man die "Vektorlänge" also die Elementezahl der Matrix. Mit **`end`** kann der Wert des letzen Elements der Zeile ermittelt werden. Das ist bedeutsam, wenn die Vektorlänge nicht bekannt ist.

Zusammenbau einer Matrix aus vorgegebenen Matrizen:

```
>> Q=[B A;A B]
Q =
     3    -9     2     4
     1     0    -6     7
     2     4     3    -9
    -6     7     1     0
```

```
>> P=repmat(A,2,1)
P =
     2     4
    -6     7
     2     4
    -6     7
```

Der Befehl **`repmat`** bedeutet "repeat matrix". Matrix P wird also durch "wiederholtes" Anlegen von Matrix A erzeugt:
2-mal in Vertikalrichtung (entspricht der Zeilenzahl)
1-mal in Horizontalrichtung (entspricht der Spaltenzahl).
Die Erzeugung von Q ist unmittelbar einsichtig.

Zugriff auf die Matrix-Elemente:

```
>> R=[2 4 7;-3 0 7;-9 5 2]
R =
     2     4     7
    -3     0     7
    -9     5     2
>> R(3,2)
ans =
     5
```

Der Zugriff auf die *Matrixelemente* erfolgt über die Angabe der Zeilen- und Spaltenindizes.
Man beachte, dass der Zeilenindex *immer zuerst* angegeben wird!
Linkerhand erfolgt mit MATLAB Zugriff auf das Element in der 3. Zeile und in der 2. Spalte von R.
Angesprochen wird es mit **`R(3,2)`**.

```
>> b=[2;6;7;-3]
b =
     2
     6
     7
    -3
>> b(3,1)
ans =
     7
```

Die Spalte b linkerhand ist eine (4×1)-Matrix.
Das 3. Element wird somit mit **`b(3,1)`** angesprochen.

```
>> a=[3 5 7 -6 1]
a =
     3   5   7  -6   1
>> a(1,4)
ans =
    -6
```

Die Zeile a linkerhand ist eine (1×5)-Matrix.
Also wird das vorletzte Element dieser Zeile mit **`a(1,4)`** angesprochen.

I.5. Programmieren in MATLAB
Lösen eines linearen Gleichungssystems mit quadratischer Matrix mit dem *Gauss-Algorithmus*
Einfaches Beispiel: 3 lineare Gleichungen mit 3 Unbekannten

Eliminationschritt 1:
Ziehe von der 2. und 3. Gleichung das 2-fache bzw. das 3-fache der 1. Gleichung ab.

$$\left.\begin{aligned} x_1 + 2x_2 + 3x_3 &= 2 \\ 2x_1 + 3x_2 - x_3 &= 5 \\ 3x_1 + 8x_2 + 10x_3 &= 1 \end{aligned}\right\} \Leftrightarrow \overbrace{\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & -1 \\ 3 & 8 & 10 \end{pmatrix}}^{A} \overbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}^{x} = \overbrace{\begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}}^{b}$$

Eliminationschritt 2:
Ziehe von der 3. Gleichung das (–2)-fache der 2. Gleichung ab

$$\left.\begin{aligned} x_1 + 2x_2 + 3x_3 &= 2 \\ - x_2 - 7x_3 &= 1 \\ 2x_2 + x_3 &= -5 \end{aligned}\right\} \Leftrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -7 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ -5 \end{pmatrix}$$

Rückwärts-Einsetzen:
Berechne zuerst x_3 mit Gleichung 3 dann x_2 mit Gleichung 2 und zuletzt x_1 mit Gleichung 1

$$\left.\begin{aligned} x_1 + 2x_2 + 3x_3 &= 2 \\ - x_2 - 7x_3 &= 1 \\ -13x_3 &= -3 \end{aligned}\right\} \Leftrightarrow \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -7 \\ 0 & 0 & -13 \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 2 \\ 1 \\ -3 \end{pmatrix}}_{c}$$

Beim Gauß-Algorithmus wird die Matrix A mittels Äquivalenzumformungen (Eliminationsschritte) in eine sog. obere "Dreiecksmatrix" U umgewandelt. Sie gestattet dann die Berechnung der Unbekannten "von hinten" (Rückwärts-Einsetzen).

Man erhält rückwärts beginnend mit x_3 :

$x_3 = 3/13 = 0{,}230769...$, $x_2 = -1 - 7x_3 = -34/13 = -2{,}61538...$, $x_1 = 2 - 2x_2 - 3x_3 = 85/13 = 6{,}53846...$

Diese Lösungen erhält man in MATLAB "per Knopfdruck", wenn man dort die Matrix A und die Spaltenmatrix b eintippt und zuletzt den Befehl `x = A\b` mit RETURN bestätigt.

```
>> format long
>> A=[1 2 3;2 3 -1;3 8 10];b=[2;5;1];
>> x=A\b
x =
   6.538461538461539
  -2.615384615384616
   0.230769230769231
```

Mit der Einstellung **`format long`** erhält man 15 Nachkommastellen.
Die Ausgabe für x beginnt mit x_1 .

Wozu also den Gaußalgorithmus programmieren?

Als geeignetes Beispiel zur *Einführung in die* MATLAB-*Programmierung* soll dies trotzdem geschehen.

Im ersten Schritt nehmen wir **A als obere Dreiecksmatrix** an, d.h. das System A x = b sei selbst von der Form U $x = c$. Wir programmieren die o.g. Berechnung der Unbekannten x_k "von hinten", d.h. den **Algorithmus "Rückwärts_Einsetzen"** bei vorgegebener Dreiecksmatrix A.

Voraussetzung: Alle Diagonalelemente a_{kk} von A sind "deutlich" von Null verschieden.

Dazu öffnen wir ein sog. ***script-file*** , einen Programmeditor, wo die Kommandos einzutragen sind. Mit Links-Klick auf das weiße Blatt-Symbol (erstes Symbol der zweiten Menüleiste unterhalb der Menüeinträge *File* und *Edit*) öffnet sich dieser Editor mit der blauen Titel-Leiste "Editor-Untitled" . Bevor wir beginnen und das *script-file* dann starten, sollten wir den mathematischen Ablauf studieren.

Wir betrachten dazu besser ein System A x = b mit 4 Unbekannten. Die 4 Gleichungen lauten dann

$$\left.\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 \\ a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 \\ a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{44}x_4 &= b_4 \end{aligned}\right\} \Rightarrow \left\{\begin{aligned} x_1 &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4\right) \\ x_2 &= \frac{1}{a_{22}}\left(b_2 - a_{23}x_3 - a_{24}x_4\right) \\ x_3 &= \frac{1}{a_{33}}\left(b_3 - a_{34}x_4\right) \\ x_4 &= \frac{1}{a_{44}}\ b_4 \end{aligned}\right.$$

Die Gleichungen für x_1 , x_2 und x_3 formulieren wir in Matrixschreibweise:

$$x_1 = \frac{1}{a_{11}}\left[b_1 - \begin{pmatrix} a_{12} & a_{13} & a_{14} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \\ x_4 \end{pmatrix} \right]$$

$$x_2 = \frac{1}{a_{22}}\left[b_2 - \begin{pmatrix} a_{23} & a_{24} \end{pmatrix} \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} \right]$$

$$x_3 = \frac{1}{a_{33}}\left[b_3 - \underbrace{\left(a_{34}\right)\left(x_4\right)}_{(1\times 1)-\text{Matrizen}} \right]$$

Entscheidend sind die Matrizenprodukte, die von den b_k für k = 1, 2, 3 subtrahiert werden. Die Zahlen a_{34} und x_4 sind hierbei als (1×1)-Matrizen aufzufassen.

Bei der Bestimmung von x_4 entfällt das Matrizenprodukt, weshalb diese Berechnung *gesondert* zu betrachten ist.

Im soeben diskutierten Beispiel ist $n = 4$. Betrachtet man ein lineares Gleichungssystem $A\,x = b$ mit *beliebiger* **oberer ($n \times n$) – Dreiecksmatrix A** (also n Gleichungen mit n Unbekannten), so erkennt man leicht die analoge Berechnungs-Formel

$$\underbrace{x_k = \frac{1}{a_{kk}}\left[b_k - \begin{pmatrix} a_{k,k+1} & \dots & a_{k,n} \end{pmatrix} \begin{pmatrix} x_{k+1} \\ \vdots \\ x_n \end{pmatrix} \right]}_{\text{für } k = 1\,,\,2\,,\,\dots\,,\,n-1} \quad \text{sowie} \quad x_n = \frac{b_n}{a_{nn}}$$

Das skript-file kann nun geschrieben werden:

```
1-  A=[1 2 3;0 -1 -7;0 0 -13];b=[2;1;-3];
2-  [n,n]=size(A); % Größe von A bestimmen
3-  x=zeros(n,1);  % x als Spalte mit n Komponenten dimensionieren
4-  x(n,1)=b(n,1)/A(n,n);% Letzte Komponente von x zuerst bestimmen
5-  disp(['x',num2str(n), ' = ',num2str(x(n,1))]) % und anzeigen
6-  for k=n-1:-1:1 % x von der vorletzten Komponente her berechnen
7-     x(k,1)=(b(k,1)-A(k,k+1:end)*x(k+1:end,1))/A(k,k);
8-     disp(['x',num2str(k), ' = ',num2str(x(k,1))]) % und anzeigen
9-  end
```

Erläuterung:
Zeile1: Hier werden A und b eingegeben. Es ist also $n = 3$.
Zeile2: Um die Größe von A flexibel zu halten, wird hier die Zeilenzahl n ermittelt.
Der Gauß-Algorithmus kann nur auf quadratische Matrizen A angewandt werden, d.h. es muss Zeilenzahl = Spaltenzahl = n sein.
Zeile3: Der zu bestimmende Lösungsspaltenvektor x wird als ($n \times 1$)-Matrix mit Nullen vorinitialisiert.
Zeile4: Löseablauf "von hinten" (Rückwärts Einsetzen) mit der gesonderten Berechnung von $x_3 = b_3 / a_{33}$.
Zeile5: Lösung x_3 anzeigen. "num2str" konvertiert zur Anzeige die Indexzahl n sowie den errechneten Wert x_3 vom Zahlenformat "double" in das Format "string". Die Angaben in Hochkomma sind Text.

Die Zeilen 6 bis 9 enthalten eine Schleife, in welcher rückwärts beginnend von x_{n-1} (hier x_2) die Unbekannten x_k bis x_1 errechnet werden. Zeile 7 beinhaltet die oben dargestellte Berechnungs-Formel.

Anmerkungen:
Der oben beschriebene Algorithmus ist nur erfolgreich, wenn in der Dreiecksmatrix $a_{kk} \neq 0$ für alle $k = 1\,,\,\dots\,,\,n$ gilt. Sonst wäre A singulär und das Gleichungssystem nicht eindeutig lösbar. Die a_{kk} dürfen auch nicht "nahe bei Null" sein, da sonst Rundungsfehler das Ergebnis wesentlich verfälschen!
Wird in der obigen **`for`**-Schleife der Wert $n = 1$ übergeben (was ja der Fall ist, wenn die Matrix A lediglich eine Zahl also eine (1×1)-Matrix ist) , so gilt dort **`k = 0:-1:1`** , d.h. die Schleife ist "leer". Sie wird von MATLAB dann sinngemäß auch nicht ausgeführt.

Das obige script-file wird *vom Script-Editor aus* durch Links-Klick auf das grüne Dreieck (in der Mitte unterhalb der Menüleiste) gestartet. Die Ausgabe der Lösungen erfolgt dann im Command Window:

```
x3 = 0.23077
x2 = -2.6154
x1 = 6.5385
```

Beim Schließen des script-files wird man aufgefordert, einen Namen anzugeben, unter dem das file abgespeichert werden soll. Die Dateiendung ist "m".
Danach erscheint das script-file im "Current Directory", einem Verzeichnis, das im MATLAB-Pfad liegt.

Im zweiten Schritt nehmen wir **A als beliebige ($n \times n$)-Matrix** an, d.h. das System A x = b muss in die Form U $x = c$ mit U als obere Dreiecksmatrix überführt werden. **Wir programmieren die erforderlichen Eliminationsschritte in einem weiteren eigenen file**. Um das Ergebnis mittels "Rückwärts-Einsetzen" weiterverarbeiten zu können, muss der Algorithmus "Rückwärts-Einsetzen" als ***function-file*** angelegt werden. Die dazu notwendige Modifikation des obigen script-files sieht dann so aus:

```
1-  function x=Rueckwaerts_Einsetzen(A,b)
2-  [n,n]=size(A); %Größe von A bestimmen
3-  x=zeros(n,1);  %x als Spalte mit n Komponenten dimensionieren
4-  x(n,1)=b(n,1)/A(n,n); %Letzte Komponente von x zuerst bestimmen
5-  disp(['x',num2str(n), ' = ',num2str(x(n,1))]) % und anzeigen
6-  for k=n-1:-1:1   %x von der vorletzten Komponente her berechnen
7-    x(k,1)=(b(k,1)-A(k,k+1:end)*x(k+1:end,1))/A(k,k);
8-    disp(['x',num2str(k), ' = ',num2str(x(k,1))]) %und anzeigen
9-  end
```

Ein function-file wird also prinzipiell wie ein script-file erstellt. Jedoch kann ein script-file nicht von einem anderen Programm aufgerufen werden, um z.B. Daten an das script-file übergeben zu können. Bei einem *function-file* werden *Eingabe*-Daten (Input) *in* das function-file *hinein* und *Ausgabe*-Daten (Output) *aus* dem function-file *heraus* definiert. Die Syntax:

$$\text{function}\ \underbrace{[\text{Aus1}, \text{Aus2}, ..., \text{Aus N}]}_{\text{Liste mit N Ausgabe-Variablen}} = \text{Funktionsname}\ \underbrace{(\text{Ein1}, \text{Ein2}, ..., \text{Ein M})}_{\text{Liste mit M Eingabe-Variablen}}$$

Die *runde* Klammer bei den Eingabe-Daten ist immer erforderlich, bei den Ausgabe-Daten ist die *eckige* Klammer entbehrlich, wenn die Liste nur *eine* Variable enthält. Das ist oben der Fall: x ist als Lösung des Gleichungssystems die einzige Ausgabe-Variable. A und b sind Eingabe-Daten, die *nach* der Überführung der Matrix A in eine obere Dreiecks-Matrix an das function-file "Rueckwaerts_Einsetzen" übergeben werden. Um das function-file zu starten, sind im Command Window einzugeben:

```
A = [1 2 3;0 -1 -7;0 0 -13];b=[2;1;-3];
x = Rueckwaerts_Einsetzen(A,b)
```

MATLAB gibt aus:

```
x3 = 0.23077
x2 = -2.6154
x1 = 6.5385
x =
    6.5385
   -2.6154
    0.2308
```

Die wiederholte Ausgabe `x = ...` kann unterdrückt werden, indem oben hinter **`function`** der Eintrag `" x = "` weggelassen wird. Lässt man den Eintrag `" x = "` stehen und entfernt stattdessen die Zeilen mit dem **`disp`**-Befehl, so erfolgt nur die Ausgabe `x = ...`

Beachte: Die zunächst *beliebige* ($n \times n$)-Matrix A wird in eine obere Dreiecksmatrix überführt, die hier nicht U sondern wieder mit A bezeichnet wird: Programmtechnisch werden also die alten Inhalte der Variablen A mit neuen Inhalten überschrieben! Dasselbe gilt für b.

Ferner übrigens: Variablen-Namen dürfen nur Buchstaben, Ziffern und den Unterstrich enthalten. Am Anfang muss ein Buchstabe stehen. ä , ö und ü (groß oder klein) sind nicht erlaubt! MATLAB unterscheidet Klein- und Großschreibweise.

Nun zum Algorithmus "**Gauss_Elimination**", den wir ebenfalls als function-file codieren.
Das Programm enthält nur 4 Zeilen! Zum Verständnis müssen wir den mathematischen Hintergrund studieren. Betrachten wir dazu ein lineares Gleichungssysten von n Gleichungen mit n Unbekannten:

$$\begin{array}{l|l|l}
G1: \quad a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 & & \text{Bedingung für G2:} \\
G2: \quad a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 & G2 - (a_{21}/a_{11}) \times G1 & a_{21} - \lambda a_{11} = 0 \\
\vdots \qquad\qquad\qquad\qquad\qquad \vdots & \vdots & \Leftrightarrow \lambda = a_{21}/a_{11} \\
Gn: \quad a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n & Gn - (a_{n1}/a_{11}) \times G1 & \text{usw.}
\end{array}$$

Beim Gaußalgorithmus wird im 1-ten Eliminationsschritt von den Gleichungen *G*2 bis *Gn* jeweils ein geeignetes Vielfaches der Gleichung *G*1 derart subtrahiert, dass in *G*2 bis *Gn* die Koeffizienten a_{21} bis a_{n1} und somit die Unbekannte x_1 verschwindet.

Die neuen $n-1$ Gleichungen *G*2 bis *Gn* enthalten nur noch die $n-1$ Unbekannten x_2 bis x_n und bilden somit ein *reduziertes* Gleichungssystem.
Verwenden wir für das obige Gleichungssystem die *Matrixschreibweise* A x = b also

(1)
$$\underbrace{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}}_{b},$$

so ist die Vorgehensweise im 1-ten Eliminationsschritt identisch mit der *linksseitigen Multiplikation* von A x = b mit der Matrix L in der folgenden Weise:

(2)
$$\underbrace{\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ -a_{21}/a_{11} & 1 & 0 & \cdots & 0 \\ -a_{31}/a_{11} & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ -a_{n1}/a_{11} & 0 & \cdots & 0 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}}_{x} = L \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}}_{b}$$

Wir richten unsere Augen nur auf die Multiplikation der 1. Zeile von L mit *allen* Spalten von A sowie die Multiplikation *aller folgenden* Zeilen von L mit der 1. Spalte von A. Es ergibt sich

(3)
$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22}^{*} & \cdots & a_{2n}^{*} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{*} & \cdots & a_{nn}^{*} \end{pmatrix}}_{LA \underset{\text{rename}}{=} A} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \underbrace{\begin{pmatrix} b_1 \\ b_2^{*} \\ \vdots \\ b_n^{*} \end{pmatrix}}_{Lb \underset{\text{rename}}{=} b}$$

Die mit * versehenen Größen stehen für beliebige Zahlenwerte. Die Matrix LA bezeichnen wir wieder mit A und Lb benennen wir wieder mit b. A und b werden also mit neuen Werten überschrieben.

Der 1-te Eliminationsschritt erzeugt aus der Matrix A in (1) also die neue Matrix A in (3).
Die erste Zeile von A *ändert sich dabei nicht*, die erste Spalte von A unterhalb a_{11} wird in (3) mit Nullen aufgefüllt. Ebenso bleibt in der Spalte *b* das erste Element b_1 unverändert.
Wir beschreiben den Ablauf im 1. Eliminationsschritt in der MATLAB-Sprache.
Für die Matrix L, mit der das Ausgangssystem (1) von links multipliziert wird, gilt

$$L = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix} - \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21}/a_{11} & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ a_{n1}/a_{11} & 0 & \cdots & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}}_{E} - \underbrace{\begin{pmatrix} 0 \\ a_{21}/a_{11} \\ \vdots \\ a_{n1}/a_{11} \end{pmatrix}}_{m} \underbrace{\begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix}}_{e}.$$

E ist die *n*-reihige Einheitsmatrix, *m* eine Spalte und *e* eine Zeile mit jeweils *n* Elementen.

Es folgt sodann

$$LA = \left(E - me\right)A = EA - meA = A - m\begin{pmatrix}1 & 0 & \cdots & 0\end{pmatrix}\begin{pmatrix}a_{11} & \cdots & a_{1n}\\ \vdots & & \vdots\\ a_{n1} & \cdots & a_{nn}\end{pmatrix}$$

mithin

$$LA = A - m\begin{pmatrix}a_{11} & \cdots & a_{1n}\end{pmatrix} \underset{\text{MATLAB}}{=} A - m * A(1\ ,\ :) \underset{\text{rename}}{=} A \tag{4}$$

sowie

$$Lb = \left(E - me\right)b = Eb - meb = b - m\begin{pmatrix}1 & 0 & \cdots & 0\end{pmatrix}\begin{pmatrix}b_1\\ \vdots\\ b_n\end{pmatrix}$$

mithin

$$Lb = b - mb_1 \underset{\text{MATLAB}}{=} b - m * b(1\ ,\ 1) \underset{\text{rename}}{=} b \tag{5}$$

Das function-file für die beschriebene Prozedur sieht somit folgendermaßen aus:

```
1-  function [A,b]=Gauss_Elimination(A,b)
2-  m=A(:,1)/A(1,1); % 1.Spalte von A dividiert durch A(1,1)
3-  m(1,1)=0;        % 1.Element der Spalte Null setzen
4-  A=A-m*A(1,:);    % Matrix A aktualisieren
5-  b=b-m*b(1,1);    % Spalte b aktualisieren
```

Es kann auf beliebige quadratische Matrizen A und Spalten b mit gleicher Zeilenzahl angewandt werden. Es erzeugt aus dem Input A eine neue Matrix A gleichen Formats mit unveränderter 1. Zeile *und Nullen in der* 1. *Spalte* unterhalb der 1. Zeile. Aus der Spalte b entsteht eine neue Spalte b gleichen Formats mit unverändertem 1. Element.

Der 2-te Elimiminationsschritt greift in (3) nur auf das in (3) eingerahmte System (6) zu und *wiederholt*

$$\underbrace{\begin{pmatrix}a_{22}^* & \cdots & a_{2n}^*\\ \vdots & & \vdots\\ a_{n2}^* & \cdots & a_{nn}^*\end{pmatrix}}_{A^{(2)}}\begin{pmatrix}x_2\\ \vdots\\ x_n\end{pmatrix} = \underbrace{\begin{pmatrix}b_2^*\\ \vdots\\ b_n^*\end{pmatrix}}_{b^{(2)}} \tag{6}$$

darauf die Prozedur des 1. Eliminationsschritts. **Die erste Zeile der Matrix $A^{(2)}$** ***ändert sich dabei nicht*****, die erste Spalte von $A^{(2)}$ unterhalb a_{22}^* wird Null. Ebenso bleibt in der Spalte $b^{(2)}$ das *erste Element* b_2^* *unverändert.***

Somit ist das Paar $A^{(2)}, b^{(2)}$ der Input für das obige file im 2-ten Eliminationsschritt.
Der *k*-te Eliminationsschritt überschreibt in A nur die Matrix-Elemente ab der *k*-ten Zeile und *k-ten* Spalte. Letztere wird unterhalb der *k*-ten Zeile mit Nullen aufgefüllt. Die Spalte b wird nur ab dem *k-ten* Element überschrieben.

In der MATLAB-Sprache gilt im 1-ten Eliminationsschritt:
`[A(1:end,1:end),b(1:end,1)]=Gauss_Elimination(A(1:end,1:end),b(1:end,1))`
Im 2. Eliminationsschritt gilt für (6):
`[A(2:end,2:end),b(2:end,1)]=Gauss_Elimination(A(2:end,2:end),b(2:end,1))`
und im *k*-ten Schritt bis $k = n - 1$:
`[A(k:end,k:end),b(k:end,1)]=Gauss_Elimination(A(k:end,k:end),b(k:end,1)).`

Für ein Gleichungssystem (1) mit beispielsweise $n = 4$ sieht von Eliminations-Schritt zu Eliminations-Schritt der Übergang von A zur Dreiecksmatrix so aus:

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}}_{\text{A(1:end,1:end)}}, \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}}_{\text{b(1:end,1)}} \quad \Rightarrow \text{Gauss\_Elimination 1.Schritt} \quad \underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22}^{*} & a_{23}^{*} & a_{24}^{*} \\ 0 & a_{32}^{*} & a_{33}^{*} & a_{34}^{*} \\ 0 & a_{42}^{*} & a_{43}^{*} & a_{44}^{*} \end{pmatrix}}_{\text{A(1:end,1:end)}}, \underbrace{\begin{pmatrix} b_1 \\ b_2^{*} \\ b_3^{*} \\ b_4^{*} \end{pmatrix}}_{\text{b(1:end,1)}}$$

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22}^{*} & a_{23}^{*} & a_{24}^{*} \\ 0 & a_{32}^{*} & a_{33}^{*} & a_{34}^{*} \\ 0 & a_{42}^{*} & a_{43}^{*} & a_{44}^{*} \end{pmatrix}}_{\text{A(2:end,2:end)}}, \underbrace{\begin{pmatrix} b_1 \\ b_2^{*} \\ b_3^{*} \\ b_4^{*} \end{pmatrix}}_{\text{b(2:end,1)}} \quad \Rightarrow \text{Gauss\_Elimination 2.Schritt} \quad \underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22}^{*} & a_{23}^{*} & a_{24}^{*} \\ 0 & 0 & a_{33}^{**} & a_{34}^{**} \\ 0 & 0 & a_{43}^{**} & a_{44}^{**} \end{pmatrix}}_{\text{A(2:end,2:end)}}, \underbrace{\begin{pmatrix} b_1 \\ b_2^{*} \\ b_3^{**} \\ b_4^{**} \end{pmatrix}}_{\text{b(2:end,1)}}$$

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22}^{*} & a_{23}^{*} & a_{24}^{*} \\ 0 & 0 & a_{33}^{**} & a_{34}^{**} \\ 0 & 0 & a_{43}^{**} & a_{44}^{**} \end{pmatrix}}_{\text{A(3:end,3:end)}}, \underbrace{\begin{pmatrix} b_1 \\ b_2^{*} \\ b_3^{**} \\ b_4^{**} \end{pmatrix}}_{\text{b(3:end,1)}} \quad \Rightarrow \text{Gauss\_Elimination 3.Schritt} \quad \underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22}^{*} & a_{23}^{*} & a_{24}^{*} \\ 0 & 0 & a_{33}^{**} & a_{34}^{**} \\ 0 & 0 & 0 & a_{44}^{***} \end{pmatrix}}_{\text{A(3:end,3:end) ObereDreiecksmatrix}}, \underbrace{\begin{pmatrix} b_1 \\ b_2^{*} \\ b_3^{**} \\ b_4^{***} \end{pmatrix}}_{\text{b(3:end,1)}}$$

Man beachte hier nochmals, dass im *k*-ten Eliminationsschritt die ersten *k* Zeilen von A und b gleich bleiben! Nach dem $(n-1)$-ten Schritt (hier also nach dem 3-ten Schritt) ist auf diese Weise A zu einer Dreiecksmatrix geworden, auf welche dann das function-file "`Rueckwaerts_Einsetzen`" angewandt werden kann. Das eigentliche "Hauptprogramm", welches den Gauss-Algorithmus ausführt, lässt sich nun leicht erschließen. Wir codieren es als script-file mit Namen "**`Gauss_Verfahren.m`**":

```
 1-  A=input('Bitte A eingeben: ');
 2-  [m,n]=size(A);   % Format von A bestimmen
 3-  if m~=n          % ~= bedeutet "ungleich"
 4-      error('Die Matrix A ist nicht quadratisch!')
 5-  end
 6-  if n>10
 7-      error('Die Matrix A ist zu groß!')
 8-  end
 9-  b=input('Bitte b eingeben: ');
10-  [p,q]=size(b);   % Format von A bestimmen
11-  if q~=1
12-      error('b muss eine Spalte sein!')
13-  end
14-  if p~=m
15-      error('Die Zeilenzahlen von b und A müssen gleich sein!')
16-  end
17   %-----------Gauss_Elimination starten-------------------------
18-  for k=1:1:n-1
19-      if abs(A(k,k))<eps
20-         error(['Eliminationsschritt ', num2str(k), ...
21-          ': Diagonalelement zu klein oder Null'])
22-      end
23-      disp(['System Ax=b nach Eliminationsschritt ', num2str(k)])
24-      [A(k:end,k:end),b(k:end,1)]= ...
25-      Gauss_Elimination(A(k:end,k:end),b(k:end,1))
26-  end
27-  disp('Die Lösung x lautet:')
28-  x=Rueckwaerts_Einsetzen(A,b)
```

Erläuterungen:
Zu Beginn wird der User mit Hilfe des **`input`**-Befehls aufgefordert, A und b einzugeben (Zeile 1 und 2). Die Eingaben werden in den gleichnamigen Variablen vor dem input-Befehl gespeichert. Anschließend werden die Formate (Zeilen- und Spaltenzahl) von A und b ermittelt (Zeile 3 und 4). Eventuelle Eingabefehler sowie unerwünschte Eingaben werden durch Fehlermedungen dann abgefangen (Zeile 5 bis 16).
Ab Zeile 18 beginnen die Eliminationsschritte. In jedem Eliminationsschritt ruft das *script-file* **`Gauss_Verfahren`** das *function-file* **`Gauss_Elimination`** auf (Zeile 24 und 25), um Schritt für Schritt A in eine obere Dreiecks-Matrix zu überführen. Dabei besteht die Möglichkeit, dass im k-ten Schritt a_{kk} "fast" oder gleich Null wird. Dann muss das Verfahren abgebrochen werden (Zeile 19-22). Geschieht das nicht, so wird das Verfahren bis $k = n - 1$ ausgeführt und A ist dann eine obere Dreiecks-Matrix.
`Gauss_Verfahren` ruft nun das *function-file* **`Rueckwaerts_Einsetzen`** auf , um beginnend mit x_n die Komponenten des Lösungs-Vektors x zu bestimmen (Zeile 27 und 28).

Anmerkung:
Mithilfe der sog. *Pivotsuche* kann dem Effekt $a_{kk} = 0$ oder "fast" 0 begegnet werden. Dabei werden Zeilen und Spalten in A geeignet vertauscht. Darauf sei hier aber verzichtet.

Übrigens:
Kommentare im Programm beginnen stets mit dem %-Zeichen.

Grundwissen über lineare Gleichungssysteme im Zusammenhang mit Matlab

Ein lineares Gleichungssystem von ***n*** **Gleichungen** mit ***m*** **Unbekannten** ist charakterisiert durch die Schreibweise

$$\left.\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m &= b_n \end{aligned}\right\}$$

Formuliert man das Gleichungssystem in *Matrixschreibweise* $A\,x = b$, so erhält man eine Matrix A mit n Zeilen und m Spalten. Die Koeffizienten $a_{i\,k}$ der Unbekannten $x_{\,k}$ erscheinen in A in derselben Anordnung wie sie linkerhand dastehen.

Die ***Matrixschreibweise*** des Gleichungssystems sieht also so aus:

k-te Spalte: Spaltenvektor

i-te Zeile: Zeilenvektor

(1)
$$\underbrace{\begin{pmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1m} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \cdots & a_{ik} & \cdots & a_{im} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nk} & \cdots & a_{nm} \end{pmatrix}}_{\boldsymbol{A}} \underbrace{\begin{pmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_m \end{pmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ b_k \\ \vdots \\ b_n \end{pmatrix}}_{\boldsymbol{b}}$$

A Koeffizienten-Matrix

x = **b** "Spaltenvektoren" (1-spaltige Matrizen)

Üblich ist die Kurzschreibweise ***A x = b*** .
Ist $n = m$, so hat (1) ebensoviele Gleichungen wie Unbekannte, die Matrix A also ebensoviele Zeilen wie Spalten. Eine solche Matrix A heißt *quadratisch*.

Die Zahlen a_{ik} heißen ***Matrixelemente***.
Die Indices i und k geben Auskunft darüber, in welcher *Zeile* und *Spalte* das jeweilige Element steht.
i ist der ***Zeilenindex*** : Das Element steht in der i - ten Zeile.
k ist der ***Spaltenindex*** : Das Element steht in der k - ten Spalte.
Mit den genannten Indices ist also das betreffende Element in der Matrix eindeutig lokalisiert.
Man beachte, daß der erste Index *immer* der Zeilenindex ist. Man merke es sich so: **Zeile zuerst**!

Wie wird das Gleichungssystem (1) gelöst?
Zunächst halten wir fest:

Das Gleichungssystem $A\,x = b$ hat eine einzige oder auch mehrere (dann sind es stets unendlich viele) Lösungen genau dann, wenn gilt

(2)
$$\boldsymbol{Rang}\begin{pmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1m} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \cdots & a_{ik} & \cdots & a_{im} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nk} & \cdots & a_{nm} \end{pmatrix} = \boldsymbol{Rang}\begin{pmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1m} & b_1 \\ \vdots & & \vdots & & \vdots & \vdots \\ a_{i1} & \cdots & a_{ik} & \cdots & a_{im} & b_k \\ \vdots & & \vdots & & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} & \cdots & a_{nm} & b_n \end{pmatrix}$$

Wir schreiben kurz ***Rang A = Rang (A , b)***.
Dies ist eine Voraussetzung für die ***Existenz*** von Lösungen von $A\,x = b$.
Unter dem ***Rang*** einer Matrix versteht man die ***Anzahl der linear unabhängigen Spalten*** in dieser Matrix.

Die Spalten s_1 , s_2 , ... , s_ℓ einer Matrix heißen ***linear unabhängig*** , wenn aus der ***Linearkombination***

$c_1\,s_1 + c_2\,s_2 + \ldots + c_\ell\,s_\ell = 0$ sich $c_1 = c_2 = \ldots = c_\ell = 0$ ergibt.

Salopp kann man sagen, dass keine der genannten Spalten sich mit Hilfe der anderen Spalten darstellen lässt.

In Matlab bestimmt man den Rang einer Matrix A mit dem Befehl **`rank(A)`.**
Zur Angabe von Rang (A , b) ist der Befehl **`rank([A,b])`** zu verwenden.

Wenn das Gleichungssystem $A\,x = b$ wegen **`rank(A)= rank([A,b])`** Lösungen *hat*, so verfährt man zur Angabe der *gesamten Lösungsmenge* L in der folgenden Weise:

(I) Man bestimmt zunächst nur eine *spezielle Lösung* x_0 von $A\,x = b$
(II) Man bestimmt anschließend die gesamte Lösungsmenge von $A\,x = 0$
Diese gesamte Lösungsmenge nennt man den ***Nullraum*** von A oder auch dessen ***Kern*** .
Per definitionem ist also Kern (A) = $\{\, x \;/\; A\,x = 0 \,\}$.
(III) Ist Kern (A) ermittelt, so ergibt sich die gesamte Lösungsmenge L von $A\,x = b$ zu

$$L = x_0 + Kern(A) \underset{def.}{=} \{x_0 + x \,/\, x \in Kern(A)\} = \{x_0 + x \,/\, Ax = 0\}$$

Amerkung:
Das Gleichungssystem $A\,x = b$ heißt *inhomogenes* Gleichungssystem (im Falle $b \neq 0$).
Das Gleichungssystem $A\,x = 0$ heißt *homogenes* Gleichungssystem (in diesem Zusammenhang das zu $A\,x = b$ gehörige homogene Gleichungssystem)
Da b im allgemeinen eine Spaltenmatrix darstellt, so bezeichnet hier das Symbol 0 eine *Spalte*, deren Komponenten als Zahlen *alle* gleich Null sind.

In Matlab bestimmt man eine spezielle Lösung x_0 mit dem Befehl **`A\b`** oder alternativ mit dem Befehl **`pinv(A)*b`**. `pinv(A)` bezeichnet die *Pseudoinverse* von A (dazu später mehr).
Der Kern von A wird mit dem Befehl **`null(A)`** ermittelt.
Beispiel zur Lösung des Gleichungssystems $A\,x = b$ mit

$$\begin{pmatrix} 2 & 3 & 4 & 5 & -8 \\ 3 & -6 & 8 & 5 & 0 \\ 0 & 0 & 7 & 0 & -7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \\ 7 \end{pmatrix} .$$

Vorgehensweise in Matlab:

```
>> A = [ 2 3 4 5 -8 ; 3 -6 8 5 0 ; 0 0 7 0 -7];  % A eingeben
>> b = [2 ; 6 ; 7];                               % b eingeben
>> Rang_A = rank(A)                 % Zum Lösungsnachweis Rang(A)
>> Rang_Ab = rank([A,b])            % und Rang(A,b) bestimmen
>> xo = pinv(A)*b                   % Spezielle Lösung xo bestimmen
>> Kern_A = null(A)                 % Kern(A) bestimmen
```

Es ist Rang(A) = Rang(A,b) = 3 $\Rightarrow$ $A\,x = b$ ist lösbar. Matlab liefert folgende Ausgaben

Ausgabe für xo:

```
xo =
   -0.1204
   -0.3600
    0.7400
   -0.3438
   -0.2600
```

Ausgabe für den Nullraum von A:

```
Kern_A =
   -0.8425    0.3271
    0.1987    0.6587
    0.2193    0.4668
    0.3932   -0.1526
    0.2193    0.4668
```

`xo` ist *ein* spezieller Lösungsdatensatz für die 5 Unbekannten x_1 , ... , x_5 des *inhomogenen* Systems $A\,x = b$.
Für **`Kern_A`** werden zwei sog. *Basislösungen* S_1 und S_2 für das *homogene* System $A\,x = 0$ ausgegeben.
Diese Spalten sind linear unabhängig und es gilt
`Kern_A` = $\{\, c_1\, S_1 + c_2\, S_2 \,/\, c_1 , c_2 \in \mathbb{R} \,\}$.

Die Gesamtheit L der Lösungen von $A\,x = b$ ist somit gegeben durch die *unendliche* Lösungsmenge
L = x_0 + Kern(A) = $\{\, x_0 + c_1\, S_1 + c_2\, S_2 \,/\, c_1 , c_2 \in \mathbb{R} \,\}$.

Der Sonderfall $n = m$

Im folgenden sei $n = m$, das heißt das System $A\,x = b$ umfasse ebensoviele Gleichungen wie Unbekannte.
Die Matrix A enthalte also ebensoviele Zeilen wie Spalten und sei somit *quadratisch*.
Für solche ***quadratischen*** Matrizen - ***und nur für diese*** - ist die sog. ***Determinante*** definiert.

Die ***Berechnung*** von Determinanten erledigt Matlab für uns!
Zum theoretischen Verständnis für den interessierten Leser trotzdem ein kurzer Exkurs über Berechnungsmethoden von Determinanten:

Eine solche Berechnungsmethode ist der ***Laplace'sche Entwicklungssatz*** .
Zu seiner Beschreibung betrachten wir eine Determinante D mit einer beliebig fixierten Zeile und Spalte.

$$D = \begin{vmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1n} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \cdots & a_{ik} & \cdots & a_{in} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nk} & \cdots & a_{nn} \end{vmatrix}$$

i-te Zeile z_i

k-te Spalte s_k

Die Darstellung einer Determinante erfolgt wie im Bild linkerhand.
Die Berechnung von D erfolgt, indem man ***eine*** *beliebige* Zeile ***oder*** Spalte auswählt und dann die folgende Berechnung ausführt:

Laplace'scher Entwicklungssatz :
Für jede gewählte Spalte s_k gilt:

(3) $D = \sum_{i=1}^{n} (-1)^{i+k} a_{ik} D_{ik}$ Man sagt: D wird durch *Entwickeln nach der k-ten Spalte* berechnet.

Für jede gewählte Zeile z_i gilt:

(4) $D = \sum_{k=1}^{n} (-1)^{i+k} a_{ik} D_{ik}$ Man sagt: D wird durch *Entwickeln nach der i-ten Zeile* berechnet.

Die Determinante D_{ik} entsteht durch *Entfernen* der i-ten Zeile und k-ten Spalte aus D . Sie wird eine *Unterdeterminante* von D genannt. $D_{i\,k}$ hat demnach nur noch $n - 1$ Zeilen und Spalten.

Anmerkung:
Der griechische Buchstabe Σ ist ein großes *Sigma* und steht für die *Summe* von Zahlenwerten. Statt $a_1 + a_2 + \ldots + a_n$ schreibt man in der Mathematik $\sum_{k=1}^{n} a_k$ gelesen "Summe a_k von k gleich 1 bis n ".
k heißt *Laufindex* oder *Summationsindex* . Für ihn kann auch ein anderes Symbol statt k gewählt werden.

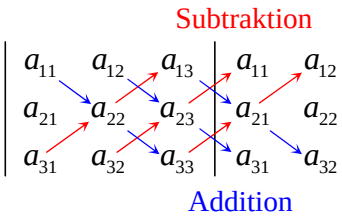
Der Laplace'sche Satz gestattet die Berechnung ***beliebiger*** Determinaten.
Für Determinanten mit **2** oder **3** ***Zeilen und Spalten*** gibt es eine einfachere Regel von Sarrus, die oft nützlich ist :

Sarrus'sche Regel

Es gilt (5) $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$

und (6) $\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{21}a_{12}$.

Die in (6) kompliziert anmutende Regel läßt sich auf folgende Weise bequem merken:



Man schreibe die beiden ersten Spalten nochmals rechts neben die Determinante und multipliziere die drei Elemente jeweils in Pfeilrichtung. Die Produkte in Pfeilrichtung von *linksoben nach rechtsunten* werden *addiert*, die in Pfeilrichtung von *linksunten nach rechtsoben subtrahiert*. Auf diese Weise ergibt sich das obige Resultat für die Determinante. Die Berechnung von Determinanten mit **2** Zeilen und Spalten erfolgt analog nach der Methode linkerhand.

Beispiel :

Es ist die Determinante $D = \begin{vmatrix} 4 & 5 & -2 & 3 \\ 2 & 7 & 0 & 1 \\ 3 & 8 & 0 & 1 \\ 1 & 1 & 3 & 1 \end{vmatrix}$ zu berechnen. Wir *entwickeln* nach der 1-ten Zeile gemäß (5) :

$$D = \begin{vmatrix} \mathbf{4} & \mathbf{5} & \mathbf{-2} & \mathbf{3} \\ 2 & 7 & 0 & 1 \\ 3 & 8 & 0 & 1 \\ 1 & 1 & 3 & 1 \end{vmatrix} = 4(-1)^{1+1}\begin{vmatrix} 7 & 0 & 1 \\ 8 & 0 & 1 \\ 1 & 3 & 1 \end{vmatrix} + 5(-1)^{1+2}\begin{vmatrix} 2 & 0 & 1 \\ 3 & 0 & 1 \\ 1 & 3 & 1 \end{vmatrix} - 2(-1)^{1+3}\begin{vmatrix} 2 & 7 & 1 \\ 3 & 8 & 1 \\ 1 & 1 & 1 \end{vmatrix} + 3(-1)^{1+4}\begin{vmatrix} 2 & 7 & 0 \\ 3 & 8 & 0 \\ 1 & 1 & 3 \end{vmatrix}.$$

Die 4 Unterdeterminanten von D bestimmen wir mit der Regel von Sarrus und erhalten sodann

$$D = 4(24-21) - 5(9-6) - 2(16+7+3-8-2-21) - 3(48-63) = 12-15+10+45 = 52.$$

Nach welcher Zeile oder Spalte man auch entwickelt, es wird sich immer 52 ergeben.
Des Rechenaufwandes wegen hätte man D daher besser nach der 3-ten Spalte entwickeln sollen.

In Matlab wird nach Eingabe einer quadratischen Matrix A die Determinante mit dem Befehl **`det(A)`** ermittelt.

Lösungsmethoden für Gleichungssysteme $A\,x = b$ mit quadratischer Matrix

I. Die Cramer'sche Regel

Die ***Cramer'sche Regel*** verwendet zur Lösung von $A\,x = b$ ***Determinanten*** :

(7) Mit $\det A = \begin{vmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{vmatrix}$ ist $x_k = \dfrac{1}{\det A}\begin{vmatrix} a_{11} & \cdots & \boxed{b_1} & \cdots & a_{1n} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & b_n & \cdots & a_{nn} \end{vmatrix}$ ***Cramer'sche Regel***

k-te Spalte

In der "Zählerdeterminante" ist also die k-te Spalte der Determinante $\det A$ durch den Spaltenvektor b zu ersetzen. Die Lösung fordert $\det A \neq 0$. Eine Matrix, deren Determinate *ungleich* Null ist, heißt ***regulär.***
Man beachte nochmals, daß Determinanten *nur für quadratische Matrizen* (Zeilenzahl = Spaltenzahl) erklärt sind! Offenbar ist

(8) $\quad A\,x = b$ ***eindeutig*** **lösbar** $\Leftrightarrow$ $\det A \neq 0$ $\Leftrightarrow$ **Matrix** A **regulär** .

Will man $A\,x = b$ lösen, so sind die genannten ***Determinanten*** zu bestimmen.
Matrizen A mit det(A) = 0 heißen ***singulär*** .

II. Lösung mit der Inversen

Jede ***reguläre*** Matrix **A** hat eine ***Inverse*** , welche mit $\mathbf{A^{-1}}$ bezeichnet wird. Sie hat das gleiche Format wie A selbst, d.h. sie ist quadratisch mit gleicher Zeilen- und Spaltenzahl wie A.
Ist die Inverse bekannt, so hat die Lösung von $\boldsymbol{A\,x = b}$ die Darstellung $\boldsymbol{x} = \mathbf{A^{-1}}\,\boldsymbol{b}$.

Die ***Berechnung*** der Inversen erledigt Matlab für uns!
Im folgenden soll die Berechnung der Inversen für den interessierten Leser kurz dargestellt werden.

Für eine Matrix $A = \begin{pmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1n} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \cdots & a_{ik} & \cdots & a_{in} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nk} & \cdots & a_{nn} \end{pmatrix}$ geht das so:

1. Schritt:
Man bestimme die Determinante von *A* also det *A* .
2. Schritt:

Für *jedes* Matrixelement $a_{i\,k}$ führe man folgendes aus:

(a) Man streiche die *i*-te Zeile und die *k*-te Spalte

(b) Man berechne die Determinante der *übriggebliebenen* Matrix also die *Unterdeterminante* D_{ik} mit $n - 1$ Zeilen und Spalten

(c) Man berechne die sog. *Adjunkte* $A_{ik} = (-1)^{i+k} D_{ik}$

(d) Man bilde die Matrix $C \underset{def.}{=} \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix}$.

(e) Man *transponiere* die Matrix *C*, das heißt man schreibe die Zeilen von *C* als Spalten und die Spalten von *C* als Zeilen also $C^{\mathrm{T}} \underset{def.}{=} \begin{pmatrix} A_{11} & \cdots & A_{n1} \\ \vdots & & \vdots \\ A_{1n} & \cdots & A_{nn} \end{pmatrix}$.

Dann gilt

(9) $\boldsymbol{A^{-1} = \dfrac{1}{\det A} C^{\mathrm{T}}}$.

Zur Lösung des folgenden Gleichungssystems $A\,x = b$ *mittels Inversion* der Matrix *A* schreiben wir zunächst

$$\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & -1 \\ 3 & 8 & 10 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}}_{b} \quad \Leftrightarrow \quad x = A^{-1} b \quad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & -1 \\ 3 & 8 & 10 \end{pmatrix}^{-1} \begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}$$

Mittels der Sarrusregel (siehe (6) auf Seite 3 unten) ergibt sich det (A) = 13.

Die Inverse von *A* ergibt sich mittels obiger Prozedur (9) zu

$$A^{-1} = \frac{1}{13} \begin{pmatrix} (-1)^{1+1}\begin{vmatrix} 3 & -1 \\ 8 & 10 \end{vmatrix} & (-1)^{1+2}\begin{vmatrix} 2 & -1 \\ 3 & 10 \end{vmatrix} & (-1)^{1+3}\begin{vmatrix} 2 & 3 \\ 3 & 8 \end{vmatrix} \\ (-1)^{2+1}\begin{vmatrix} 2 & 3 \\ 8 & 10 \end{vmatrix} & (-1)^{2+2}\begin{vmatrix} 1 & 3 \\ 3 & 10 \end{vmatrix} & (-1)^{2+3}\begin{vmatrix} 1 & 2 \\ 3 & 8 \end{vmatrix} \\ (-1)^{3+1}\begin{vmatrix} 2 & 3 \\ 3 & -1 \end{vmatrix} & (-1)^{3+2}\begin{vmatrix} 1 & 3 \\ 2 & -1 \end{vmatrix} & (-1)^{3+3}\begin{vmatrix} 1 & 2 \\ 2 & 3 \end{vmatrix} \end{pmatrix}^{\mathrm{T}} = \frac{1}{13} \begin{pmatrix} 38 & -23 & 7 \\ 4 & 1 & -2 \\ -11 & 7 & -1 \end{pmatrix}^{\mathrm{T}} .$$

Ausführung der Transposition liefert:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \underbrace{\frac{1}{13}\begin{pmatrix} 38 & -23 & 7 \\ 4 & 1 & -2 \\ -11 & 7 & -1 \end{pmatrix}^{\mathrm{T}}}_{A^{-1}} \begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix} = \frac{1}{13}\begin{pmatrix} 38 & 4 & -11 \\ -23 & 1 & 7 \\ 7 & -2 & -1 \end{pmatrix}\begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix} = \frac{1}{13}\begin{pmatrix} 85 \\ -34 \\ 3 \end{pmatrix} = \begin{pmatrix} 85/13 \\ -34/13 \\ 3/13 \end{pmatrix}.$$

Für $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ ist gemäß obiger Prozedur $A^{-1} = \frac{1}{\det A}\begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix}$.

Die Inverse A^{-1} ist allgemein charakterisiert durch die Bedingung $A^{-1}A = AA^{-1} = E = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$.

E heißt ***Einheitsmatrix*** und hat das gleiche Format wie A.
Bis auf die ***Hauptdiagonale***, die nur mit 1 besetzt ist, sind alle anderen Matrixelemente gleich Null.

In Matlab wird nach Eingabe einer ***quadratischen*** Matrix A die Inverse mit dem Befehl **`inv(A)`** ermittelt.
Nach Eingabe von *b* wird dann *x* mit dem Befehl **`inv(A)*b`** ermittelt.
Eine Einheitsmatrix mit *n* Zeilen und Spalten wird mit dem Befehl **`eye(n)`** generiert.

Anmerkung:
Stimmen Zeilenzahl und Spaltenzahl von A *nicht* überein, so hat A *keine Inverse*.
Es gibt jedoch für ***jede Matrix A*** eine eindeutige sog. ***Pseudoinverse.*** Bei quadratischen Matrizen sind Inverse und Pseudoinverse identisch.

In Matlab wird nach Eingabe einer ***nicht quadratischen*** Matrix A die PseudoInverse mit dem Befehl **`pinv(A)`** ermittelt. Nach Eingabe von *b* wird dann *x* mit dem Befehl **`pinv(A)*b`** oder **`A\b`** ermittelt (siehe Seite 2).

III. Der Gaußalgorithmus

Betrachtet man die Berechnung der Inverse von A sowie der Determinante von A (welche zusätzlich für die Inverse benötigt wird) genauer, so wird schnell klar, dass der *Rechenaufwand* bei Steigerung der Zeilen- und Spaltenzahl ernorm groß wird! Irgendwann wird dann auch jeder Computer und die darauf vorhandene Software überfordert sein. In der Tat versucht die moderne numerische Mathematik bei "großen" linearen Gleichungssystemen die Berechnung von Determinanten und Inversen zu umgehen. Ein Beispiel hierfür ist der *Gaußalgorithmus*.
Wir betrachten dazu nochmal das obige lineare Gleichungssystem mit den 3 Unbekannten x_1, x_2 und x_3:

$$\left.\begin{array}{ll} \text{(a)} & x_1 + 2x_2 + 3x_3 = 2 \\ \text{(b)} & 2x_1 + 3x_3 - x_3 = 5 \\ \text{(c)} & 3x_1 + 8x_2 + 10x_3 = 1 \end{array}\right\} \quad \underset{\text{Matrix-schreibweise}}{\Leftrightarrow} \quad \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & -1 \\ 3 & 8 & 10 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}}_{b}.$$

Der Gaußalgorithmus verfolgt die Idee, das Gleichungssystem $A\,x = b$ schrittweise in ein *äquivalentes* Gleichungssystem $U\,x = c$ umzuformen derart, daß U eine ***Dreiecksmatrix*** ist.

Dazu werden im 1. Schritt die Matrixelemente 2 und 3 in der 1-ten Spalte von *A* durch eine Null ersetzt.
Im 2. Schritt wird die 8 in der 2-ten Spalte von *A* durch eine Null ersetzt.
Das erfolgt so:

1. Schritt:
Subtrahiere von Gleichung (b) das 2-fache von Gleichung (a)
Subtrahiere von Gleichung (c) das 3-fache von Gleichung (a)
Aus dem obigen Gleichungssystem wird dann das folgende mit den gewünschten Nullen:

$$\left.\begin{array}{lrcl} \text{(d)} & x_1 + 2x_2 + 3x_3 & = & 2 \\ \text{(e)} & -x_2 - 7x_3 & = & 1 \\ \text{(f)} & 2x_2 + x_3 & = & -5 \end{array}\right\} \quad \overset{\text{Matrix-schreibweise}}{\Leftrightarrow} \quad \begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -7 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ -5 \end{pmatrix} .$$

2. Schritt:
Addiere zu Gleichung (f) das 2-fache von Gleichung (e).
Das ergibt die gewünschte Null in der 2-ten Matrixspalte ganz unten:

$$\left.\begin{array}{lrcl} \text{(g)} & x_1 + 2x_2 + 3x_3 & = & 2 \\ \text{(h)} & -x_2 - 7x_3 & = & 1 \\ \text{(i)} & -13x_3 & = & -3 \end{array}\right\} \quad \overset{\text{Matrix-schreibweise}}{\Leftrightarrow} \quad \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -7 \\ 0 & 0 & -13 \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 2 \\ 1 \\ -3 \end{pmatrix}}_{c} .$$

U ist eine sog. ***obere Dreiecksmatrix***. Da das Gleichungssystem $U\,x = c$ durch *Äquivalenzumformung* erhalten wird, hat $U\,x = c$ ***die gleichen Lösungen*** wie das Ausgangssystem $A\,x = b$. Die Lösungen von $U\,x = c$ erhält man bequem, indem man mit der letzten Gleichung (i) beginnt und dann "von unten nach oben" verfährt:

$$\text{(i)} \;\Rightarrow\; x_3 = \frac{3}{13} \;\underset{\text{(h)}}{\Rightarrow}\; x_2 = -1 - 7x_3 = -\frac{34}{13} \;\underset{\text{(g)}}{\Rightarrow}\; x_1 = 2 - 2x_2 - 3x_3 = \frac{85}{13} .$$

Die *Determinante* von A und deren Inverse wird also garnicht benötigt. Dies ist der Vorteil des Gaußalgorithmus bei der Lösung linearer Gleichungssysteme.

Wie oben erwähnt ist das zentrale Anliegen des Gaußalgorithmus die ***Umwandlung der Matrix A in eine Dreiecksmatrix U.*** Dreiecksmatrizen sind quadratisch und sehen so aus:

$$U = \begin{pmatrix} a_{11} & \cdots & \cdots & a_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix} \quad \text{oder} \quad L = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ a_{n1} & \cdots & \cdots & a_{nn} \end{pmatrix}$$

U heißt *obere Dreiecksmatrix* (**U**pper triangular Matrix)
L heißt *untere Dreiecksmatrix* (**L**ower triangular Matrix)

Die Determinanten von Dreiecksmatrizen sind sofort problemlos berechenbar! Für solche Matrizen gilt nämlich:

(10) $\det L = \det U = a_{11} \cdot a_{22} \cdot \ ... \ \cdot a_{nn}$,

wie man mit dem Laplaceschen Entwicklungsatz leicht feststellt.
Gemäß der Sarrusregel (6) ist det (A) = 13. Analog ergibt sich mit (10) ebenfalls det (U) = 13 und das ist kein Zufall. *Es ist stets der Fall*! Hat man mit dem Gaußalgorithmus A in U umgewandelt, so folgt det (A) = det (U), d.h. der Gaußalgorithmus ist auch eine Methode zur (effizienteren) Determinantenberechnung.
Auf den Beweis dieser Tatsache muß hier verzichtet werden.

Merken wir uns also:

1. Der ***Gaußalgorithmus*** wandelt das lineare Gleichungssystem $\boldsymbol{A\,x = b}$ um in ein *äquivalentes* lineares Gleichungssystem $\boldsymbol{U\,x = c}$ mit *Dreiecksmatrix* $\boldsymbol{U}$.
2. Aus $U\,x = c$ ergeben sich *ohne* Kenntnis von det A die Lösungen $x_1 , x_2 , \ ... \ , x_n$ von $A\,x = b$. Anders ausgedrückt: Die Systeme $A\,x = b$ und $U\,x = c$ haben die *gleichen* Lösungen.
3. Es ist **det *A* = det *U*** .

Rechenregeln für Matrizen, Determinanten und Transposition

Zwei Matrizen werden addiert , indem ihre entsprechenden Elemente addiert werden.
Mit der Schreibweise $A = (a_{ik})$ und $B = (b_{ik})$ gilt also $A + B = (a_{ik} + b_{ik})$.

Addiert man eine Matrix $A = (a_{ik})$ n – mal , dann schreibt man $\underbrace{A + \ldots + A}_{n\ -\ \text{mal}} = (na_{ik}) \underset{def.}{=} nA \underset{def.}{=} An$.

Jedes Matrixelement wird also mit der Zahl n multipliziert.

Analog definiert man die ***Multiplikation einer Matrix mit einer beliebigen Zahl*** λ : $\lambda A \underset{def.}{=} (\lambda a_{ik}) \underset{def.}{=} A\lambda$

Zu einer Matrix A lautet somit die ***negative Matrix*** $-A \underset{def.}{=} (-1)\,A = (-a_{ik})$.

Die Subtraktion einer Matrix B von einer Matrix A definiert man als Addition von $-B$ zu A also

Zwei Matrizen werden subtrahiert , indem ihre entsprechenden Elemente subtrahiert werden.
Mit der Schreibweise $A = (a_{ik})$ und $B = (b_{ik})$ gilt also $A - B = (a_{ik} - b_{ik})$.

Beachte: Die Addition und Subtraktion ist nicht nur für quadratische Matrizen sondern auch für solche mit *unterschiedlicher* Zeilen- und Spaltenzahl erlaubt. Es können aber nur Matrizen addiert oder subtrahiert werden, die zueinander "passen": A und B müssen beide die *gleiche Anzahl von Zeilen und Spalten* haben!

Auch bei der ***Multiplikation zweier Matrizen*** A und B müssen diese zueinander "passen":
Die Anzahl der Spalten von A muß gleich der Anzahl der Zeilen von B sein.
Ist diese Anzahl mit m bezeichnet, so kann mit

$$A = (a_{ik}) \underset{def.}{=} \begin{pmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1m} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \cdots & a_{ik} & \cdots & a_{im} \\ \vdots & & \vdots & & \vdots \\ a_{n1} & \cdots & a_{nk} & \cdots & a_{nm} \end{pmatrix} \quad \text{und} \quad B = (b_{kj}) \underset{def.}{=} \begin{pmatrix} b_{11} & \cdots & b_{1j} & \cdots & b_{1p} \\ \vdots & & \vdots & & \vdots \\ b_{k1} & \cdots & b_{kj} & \cdots & b_{kp} \\ \vdots & & \vdots & & \vdots \\ b_{m1} & \cdots & b_{mj} & \cdots & b_{mp} \end{pmatrix}$$

i-te Zeile z_i ; j-te Spalte s_j

die Berechnungsvorschrift für $C = AB = (c_{ij})$

kurz auch so formuliert werden: $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$ und somit $(c_{ij}) = (a_{ik})(b_{kj}) = \left(\sum_{j=1}^{m} a_{ik} b_{kj} \right)$.

Interpretiert man die i-te Zeile von A als **1-*zeilige Matrix*** $\boldsymbol{z_i}$ mit den m einelementigen Spalten und die j-te Spalte von B als **1-*spaltige Matrix*** $\boldsymbol{s_j}$ mit den m einelementigen Zeilen, so ist das Produkt c_{ij} = (i-te Zeile von A) · (j-te Spalte von C) das Matrizenprodukt dieser beiden Matrizen: $\boldsymbol{c_{ij} = z_i s_j}$.

Ein Beispiel zur Matrizenmultiplikation:

$$A = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 4 \\ -3 & 2 & 5 \\ 9 & 7 & 6 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} s_1 & s_2 & s_3 \end{pmatrix} = \begin{pmatrix} 8 & 3 & 0 \\ 4 & 3 & 9 \\ 1 & 2 & 1 \end{pmatrix} .$$

Es gilt

$$AC = \begin{pmatrix} z_1 s_1 & z_1 s_2 & z_1 s_3 \\ z_2 s_1 & z_2 s_2 & z_2 s_3 \\ z_3 s_1 & z_3 s_2 & z_3 s_3 \end{pmatrix} = \begin{pmatrix} 1\cdot 8 + 2\cdot 4 + 4\cdot 1 & 1\cdot 3 + 2\cdot 3 + 4\cdot 2 & 1\cdot 0 + 2\cdot 9 + 4\cdot 1 \\ -3\cdot 8 + 2\cdot 4 + 5\cdot 1 & -3\cdot 3 + 2\cdot 3 + 5\cdot 2 & -3\cdot 0 + 2\cdot 9 + 5\cdot 1 \\ 9\cdot 8 + 7\cdot 4 + 6\cdot 1 & 9\cdot 3 + 7\cdot 3 + 6\cdot 2 & 9\cdot 0 + 7\cdot 9 + 6\cdot 1 \end{pmatrix} = \begin{pmatrix} 20 & 17 & 22 \\ -11 & 7 & 23 \\ 106 & 60 & 69 \end{pmatrix} .$$

Um in der Produktmatrix beispielsweise das Element in der 3-ten Zeile und in der 1-ten Spalte zu erhalten, multipliziere man also die 3-te Zeile mit der 1-ten Spalte im Sinne der Matrixmuliplikation:

$d_{31} = z_3 s_1 = \begin{pmatrix} 9 & 7 & 6 \end{pmatrix} \begin{pmatrix} 8 \\ 4 \\ 1 \end{pmatrix} = 9 \cdot 8 + 7 \cdot 4 + 6 \cdot 1 = 106$. Analog sind die restlichen Elemente zu bestimmen.

Matrizen mit *n* Zeilen und *m* Spalten heißen (*n* , *m*)-Matrizen.

Darf man wie bei der Zahlenmultiplikation die Matrizen bei ihrer Multiplikation *vertauschen*?

Das Beispiel $\begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 0 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 14 \end{pmatrix}$ und $\begin{pmatrix} 2 & 1 \\ 0 & 4 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 4 & 3 \\ 8 & 12 \end{pmatrix}$ zeigt bereits, daß dies ***nicht*** der Fall ist. Bereits hier sei auf diese Tatsache *unbedingt* hingewiesen:

Die Matrizenmultiplikation ist ***nicht kommutativ*** **!** Im allgemeinen gilt $\boldsymbol{AC \neq CA}$ **!**

Zum Schluß seien alle Rechenregeln aufgelistet. Es seien λ und μ beliebige Zahlen.

Matrizenregeln

(a)	$A\,A^{-1} = A^{-1}\,A = E$	Die Inverse ist eindeutig definiert
(b)	$A - A = O$	Nullmatrix: Alle Matrixelemente sind gleich Null
(c)	$A + B = B + A$	Kommutativgesetz der Matrixaddition
(d)	$(A + B) + C = A + (B + C)$	Assoziativgesetz der Matrixaddition
(e)	$\lambda (\mu A) = (\lambda \mu) A$	Assoziativgesetz bei Multiplikation einer Matrix mit Zahlen
(f)	$\lambda (A + B) = \lambda A + \lambda B$	Distributivgesetz bei Multiplikation einer Matrixsumme mit Zahl
(g)	$(\lambda + \mu) A = \lambda A + \mu B$	Distributivgesetz bei Multiplikation einer Zahlensumme mit Matrix
(h)	$\mu (AB) = (\mu A) B = A (\mu B)$	Assoziativgesetz bei Multiplikation einer Zahl mit Matrizen
(i)	$A (B C) = (A B) C$	Assoziativgesetz der Matrixmultiplikation
(j)	$A (B + C) = AB + AC$	Distributivgesetz der linksseitigen Matrixmultiplikation
(k)	$(A + B) C = AC + BC$	Distributivgesetz der rechtsseitigen Matrixmultiplikation
(l)	$A\,O = O\,A = O$	Multiplikation mit Nullmatrix
(m)	$A\,E = E\,A = E$	Multiplikation mit Einheitsmatrix
(n)	$A + O = O + A = A$	Addition mit Nullmatrix
(o)	$1\,A = A\,1 = A$	Multiplikation einer Matrix mit der Zahl 1
(p)	$(A\,B)^{-1} = B^{-1} A^{-1}$	Inversion eines Matrizenprodukts

Determinantenregeln

(q)	$\det AB = \det A \det B$	Determinante eines Matrizenprodukts
(r)	$\det E = 1$	Determinante der Einheitsmatrix
(s)	$\det A^{\mathrm{T}} = \det A$	Determinante der transponierten Matrix
(t)	$\det(A^{-1}) = \frac{1}{\det A}$	Determinante der inversen Matrix

Transpositionsregeln

(u)	$(AB)^{\mathrm{T}} = B^{\mathrm{T}} A^{\mathrm{T}}$	Transposition eines Matrizenprodukts
(v)	$(A + B)^{\mathrm{T}} = A^{\mathrm{T}} + B^{\mathrm{T}}$	Transposition einer Matrixsumme
(w)	$(\lambda A)^{\mathrm{T}} = \lambda\ A^{\mathrm{T}}$	Transposition eines Produktes aus Matrix und Zahl
(x)	$(A^{\mathrm{T}})^{-1} = (A^{-1})^{\mathrm{T}}$	Vertauschbarkeit von Transposition und Inversion

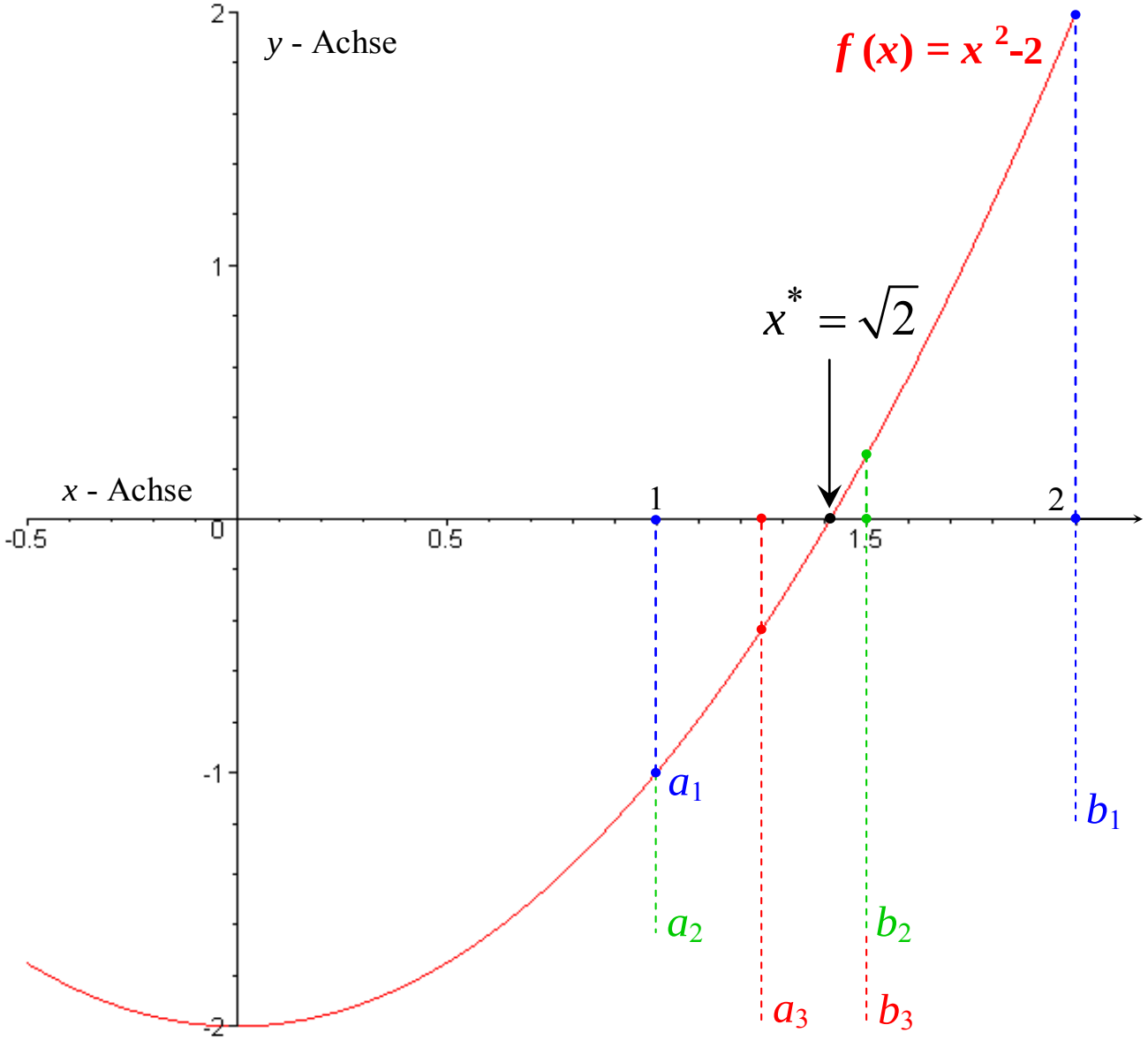
Anmerkung:
In Matlab verwendet man statt T als Transpositionszeichen ein Apostroph: **`A'`**.
Eine Nullmatrix O von *n* Zeilen und *m* Spalten wird mit dem Befehl **`zeros(n,m)`** generiert.

II. Iterationsverfahren

Das *Gaussverfahren* in Kapitel I führt nach *endlich vielen* Schritten zur Lösung und beschränkt sich auf *lineare* Gleichungsysteme. Ein *Iterationsverfahren* ist mit einer *unendlichen Folge* verbunden, welche gegen die gesuchte Lösung *konvergiert*. Da das Verfahren nach endlich vielen Schritten abgebrochen werden muss, erhält man eine Näherungslösung. Diese ist umso genauer, je mehr Schritte (sog. *Iterationsschritte*) ausgeführt werden. Iterative Verfahren sind auch auf *Gleichungssysteme* anwendbar. Dazu erst später.

Wir beschränken uns zunächst auf reellwertige Funktionen $f : \mathbb{R} \to \mathbb{R}$ *einer* reellwertigen Veränderlichen. Die sich ergebenden unendlichen Folgen sind dann reellwertige *Zahlenfolgen*.

II.1 Das Intervallschachtelungsverfahren (sog. Bisektionsverfahren)



Beispiel:
Es ist ein Näherungswert für $\sqrt{2}$, d.h. für die Nullstelle x^* der Parabelfunktion $f(x) = x^2 - 2$ zu bestimmen (siehe obiges Bild). Anders ausgedrückt: Es ist eine Lösung x^* der Gleichung $x^2 - 2 = 0$ zu ermitteln.

Vorgehensweise:
1-ter Iterationsschritt:
Man wähle ein ***Startintervall*** **[a_1 , b_1]** mit $f(a_1)\, f(b_1) < 0$, was bedeutet, dass die gesuchte Nullstelle zwischen a_1 und b_1 liegt: $x^* \in]\, a_1 \,,\, b_1 \,[$.

Man definiere anschließend den 1-ten Näherungswert $x_1 = \frac{a_1 + b_1}{2} = a_1 + \frac{b_1 - a_1}{2}$ als *Mittelpunkt* von $]\, a_1 \,,\, b_1 \,[$ und bestimme $f(x_1)$.

Es ergeben sich 3 Möglichkeiten, von denen genau eine stets zutrifft:
1. Fall: Ist $f(x_1) = 0 \Rightarrow x_1$ ist die gesuchte Nullstelle x^*
2. Fall: Ist $f(a_1)\,f(x_1) < 0 \Rightarrow x^* \in]\,a_1\,,\,x_1\,[$. Definiere dann das ***Folgeintervall*** $[a_2\,,\,b_2] \underset{def.}{=} [a_1\,,\,x_1]$.
3. Fall: Ist $f(x_1)\,f(b_1) < 0 \Rightarrow x^* \in]\,x_1\,,\,b_1\,[$. Definiere dann das ***Folgeintervall*** $[a_2\,,\,b_2] \underset{def.}{=} [x_1\,,\,b_1]$.

2-ter Iterationsschritt:

Man wiederhole das Verfahren am Intervall $[a_2\,,\,b_2]$ mit der Intervallmitte $x_2 = \frac{a_2+b_2}{2} = a_2 + \frac{b_2-a_2}{2}$ usw.

k-ter Iterationsschritt:

Tritt der Fall 1 in keinem Schritt ein, so haben wir im k-ten Schritt das Intervall $[a_k\,,\,b_k]$ mit dessen Mitte $x_k = \frac{a_k+b_k}{2} = a_k + \frac{b_k-a_k}{2}$ als k-tem *Näherungswert* für x^* .

Es gilt $x^* \in]\,a_k\,,\,x_k\,[$ oder $x^* \in]\,x_k\,,\,b_k\,[$ also auf jeden Fall

(1) $$\left| x_k - x^* \right| < \frac{b_k-a_k}{2} = \frac{b_{k-1}-a_{k-1}}{2^2} = \ldots = \frac{b_1-a_1}{2^k} \le \varepsilon \ .$$

ε repräsentiert eine vorgegebene Genauigkeit für den Näherungswert x_k von $x^* = \sqrt{2}$.
Diese Genauigkeit wird wegen des wachsenden Nenners 2^k nach genügend vielen Iterationsschritten erreicht.

Ein simples function-file mit Namen **`Bisektionsverfahren1`** zur Ausführung des Bisektionsverfahrens sieht etwa so aus:

```
 1-  function Bisektionsverfahren1(f,a,b,n)
 2   % Eingabe in der MATLAB-Befehlszeile im Command Window:
 3   % >>Bisektionsverfahren1('x^2-2',0,2,100)
 4-  format long
 5-  f=inline(f);     % f einlesen
 6-  for k=1:n        % n = Anzahl der Bisektionen
 7-     c=(a+b)/2;   % Intervallmitte als k-te Näherung bestimmen
 8-     if f(c)==0   % Wenn c selbst Nullstelle, dann
 9-        break;    % Abbruch: Ausstieg aus der for-Schleife
10-     elseif f(a)*f(c)<0
11-            b=c;  % Folgeintervall definieren
12-         else
13-            a=c;
14-         end
15-     end
16-  end
17-  disp('Ergebnis des Bisektionsverfahrens:')
18-  Nullstelle_von_f=c
```

Hier wird auf das Abfangen unterschiedlicher Fehleingaben mittels geeigneter Fehlermeldungen verzichtet.
Es sollte also ein Startintervall $[a\,,\,b]$ mit $f(a)\,f(b) < 0$ bekannt sein, welches genau eine Nullstelle von f enthält. Mit Hilfe des sog. **`inline`** - Befehls kann f eingelesen werden. Dazu muss f bei der Eingabe im Command Window in Hochkomma angegeben werden (siehe Kommentar im obigen file).
Mit n gibt man die Anzahl der Iterationen an, welche mittels einer **`for`**-Schleife ausgeführt werden.
Allerdings hat man beim Ergebnis keine Information über dessen Genauigkeit.
Abhilfe verschafft der Gebrauch einer **`while`**-Schleife, bei der alle Befehle bis zur zugehörigen **`end`**-Anweisung so lange wiederholt werden, wie eine nach **`while`** angegebene Bedingung *wahr* ist.
Als Bedingung wäre hier geeignet: $b-a \ge \varepsilon$ oder $\left|f(a)\right| \ge \varepsilon \le \left|f(b)\right|$,
d.h. man gibt eine gewisse Genauigkeit ε vor, etwa **`Eps = 1e-15`** .
Die Schleife wird dann so oft wiederholt bis diese Genauigkeit *unterschritten* ist.

Das function-file mit Namen **Bisektionsverfahren2** sieht dann so aus:

```
 1-  function Bisektionsverfahren2(f,a,b)
 2   % Eingabe in der MATLAB-Befehlszeile:
 3   % >>Bisektionsverfahren2('x^2-2',0,2)
 4-  format long
 5-  f = inline(f);   % f einlesen
 6-  Eps = 1e-15;     % Eps-Wert nicht zu klein wählen!
 7-  while (b-a>=Eps ||...          % || bedeutet "oder"
 8-        abs(f(a))>=Eps &&...    % && bedeutet "und"
 9-        abs(f(b))>=Eps)
10-        c=(a+b)/2; % Intervallmitte als k-te Näherung bestimmen
11-        if f(c)==0
12-           break;  % Abbruch, wenn Nullstelle c gefunden
13-        elseif f(a)*f(c)<0
14-           b=c;
15-        else
16-           a=c;
17-        end
18-  end
19-  disp('Ergebnis des Bisektionsverfahrens:')
20-  Nullstelle_von_f=c
```

Hinweis: Wird **Eps** zu klein gewähl, z.B. **Eps = 1e-16** also kleiner als die in MATLAB vorgegebene *Maschinengenauigkeit* **eps = 2.220446049250313e-16**, so wird die **while**-Schleife zur Endlosschleife, da **Eps** nie unterschritten wird, d.h. MATLAB "hängt sich auf ". Bei solchen und ähnlichen Gelegenheiten ist lediglich Strg+Pause zu drücken und MATLAB ist wieder betriebsbereit.

Beide files liefern bei fehlerfreiem Verlauf $x^* = \sqrt{2} = 1.414213562373095$.

Der Vorteil des Bisektionsverfahrens

Bei vorgegebener Genauigkeit ε gemäß (1) konvergiert das Verfahren *stets* zu einer Lösung.

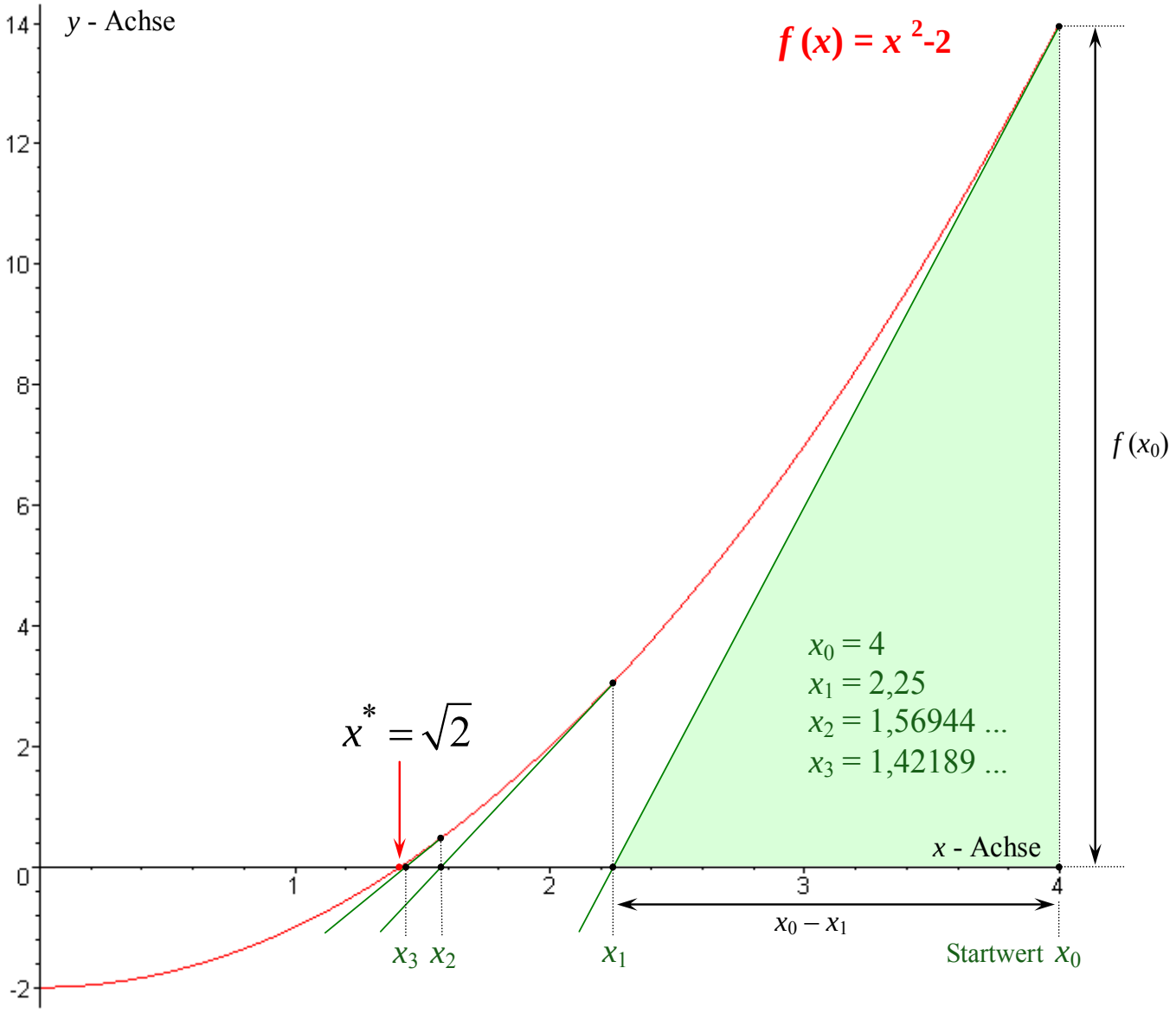
Der Nachteil des Bisektionsverfahrens

Das Verfahren konvergiert *langsam*! Nach erst 50 Iterationsschritten erreicht **Bisektionsverfahren2** mit der Ausgabe von $c = \frac{a+b}{2}$ pro Iterationsschritt (Zeile 10 im function-file) den obigen Wert für $\sqrt{2}$:

1.000000000000000	1.414184570312500	1.414213553071022	1.414213562371515
1.500000000000000	1.414245605468750	1.414213560521603	1.414213562372424
1.250000000000000	1.414215087890625	1.414213564246893	1.414213562372879
1.375000000000000	1.414199829101563	1.414213562384248	1.414213562373107
1.437500000000000	1.414207458496094	1.414213561452925	1.414213562372993
1.406250000000000	1.414211273193359	1.414213561918587	1.414213562373050
1.421875000000000	1.414213180541992	1.414213562151417	1.414213562373078
1.414062500000000	1.414214134216309	1.414213562267833	1.414213562373092
1.417968750000000	1.414213657379150	1.414213562326040	1.414213562373099
1.416015625000000	1.414213418960571	1.414213562355144	1.414213562373096
1.415039062500000	1.414213538169861	1.414213562369696	1.414213562373094
1.414550781250000	1.414213597774506	1.414213562376972	**1.414213562373095**
1.414306640625000	1.414213567972183	1.414213562373334	

Um diese Liste zu erhalten, lasse man im obigen file **Bisektionsverfahren2** in Zeile 10 das Semikolon weg. Ansonsten erscheint nur das Endergebnis **1.414213562373095** mit Kommentar gemäß Zeile 19 und 20. Wünschenswert wäre ein Verfahren, das "rascher" zum gewünschten Ergebnis führt.

II.2 Das Newtonverfahren



Es soll wiederum die Lösung $x^* = \sqrt{2}$ der Gleichung $f(x) = x^2 - 2 = 0$ näherungsweise bestimmt werden. Ausgehend von einem *Startwert* x_0 "in der Nähe" der gesuchten Nullstelle wird eine Folge von Zahlen $x_1, x_2, x_3, \ldots$ konstruiert, welche gegen x^* konvergiert.

Das geht so:
An der Stelle x_0 wird eine *Tangente* an den Graphen von f angelegt, welche die x-Achse an der Stelle x_1 schneidet. Da $f'(x_0)$ die *Tangentensteigung* ist, liest man im obigen Bild am grünen Steigungsdreieck ab:

$$\frac{f(x_0)}{x_0 - x_1} = f'(x_0) \quad \Leftrightarrow \quad \frac{f(x_0)}{f'(x_0)} = x_0 - x_1 \quad \Leftrightarrow \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \; .$$

Indem man x_1 als neuen Startwert interpretiert, bestimmt man analog x_2 usw. Wir erhalten so die mit dem Startwert x_0 beginnende unendliche Zahlenfolge (2), welche offenbar gegen $\sqrt{2}$ konvergiert.

(2) $$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \;\; , \;\; k = 0,1,2,3, \; \ldots}$$ ***Newtonsche Rekursionsformel***

(2) kann allgemein zur Nullstellenbestimmung differenzierbarer Funktionen f verwendet werden.

Wegen $f'(x) = 2x$ ergibt sich hier wegen (2) $x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k} = x_k - \frac{x_k}{2} + \frac{1}{x_k}$ also

(3) $\boxed{x_{k+1} = \frac{x_k}{2} + \frac{1}{x_k} \quad , \quad k = 0,1,2,3, \ \ldots}$ und somit z.B. für $x_0 = 2$:

$$x_1 = \frac{2}{2} + \frac{1}{2} = \frac{3}{2} = 1{,}5$$

$$x_2 = \frac{1{,}5}{2} + \frac{1}{1{,}5} = \frac{17}{12} = 1{,}41\overline{6}$$

$$x_3 = \frac{17/12}{2} + \frac{1}{17/12} = \frac{577}{408} = 1{,}41421568\ldots$$

$$x_4 = \frac{577/408}{2} + \frac{1}{577/408} = \frac{665857}{470832} = 1{,}41421356237469\ldots$$

Bis auf die roten Ziffern stimmen die Nachkommastellen!
Im Gegensatz zum Bisektionsverfahren liefert das Newtonverfahren bereits nach wenigen Iterationsschritten sehr genaue Ergebnisse.

Mit dem function-file **Newtonverfahren_fuer_f** mit Ausgabeparameter **Nullstelle_von_f** kann die Nullstelle $\sqrt{2}$ der Funktion $f(x) = x^2 - 2$ folgendermaßen berechnet werden:

```
 1-  function [Nullstelle_von_f]=Newtonverfahren_fuer_f(xo)
 2-  format long
 3-  Eps=1e-15;                  %Genauigkeit festlegen
 4-  xk=xo;                      %xk mit Startwert xo belegen
 5-  xk1=xo/2+1/xo;              %xk+1 mit x1 belegen
 6-  while abs(xk1-xk)>=Eps      %Abfrage, ob Genauigkeit erreicht
 7-      S=xk1;                  %xk+1 in S zwischenspeichern
 8-      xk1=xk1/2+1/xk1;        %xk+2 bestimmen
 9-      xk=S;                   %xk für nächsten Iterationsschritt
10-  end                         %mit xk+1 belegen
11-  Nullstelle_von_f=xk1;
```

Das file liefert mit Startwert $x_0 = 4$ **(gemäß der obigen Graphik)** bereits nach 6 Iterationsschritten den Wert $\sqrt{2} = 1.414213562373095$. Zeilen 5 und 8 sind gemäß (2) anzuändern, wenn die Nullstelle einer anderen Funktion f bestimmt wird. Lässt man in Zeile 9 das Semikolon weg, so kann man die Liste der Iterationswerte im Command Window einsehen:

```
>> [Nullstelle_von_f]=Newtonverfahren_fuer_f(4)
xk =
   2.250000000000000
xk =
   1.569444444444444
xk =
   1.421890363815143
xk =
   1.414234285940073
xk =
   1.414213562524932
xk =
   1.414213562373095
Nullstelle_von_f =
   1.414213562373095
```

← Dies ist im obigen Bild der Iterationswert x_3 , der schon ganz nahe bei $\sqrt{2}$ liegt.

Der Vorteil des Newtonverfahrens

Bei vorgegebener Genauigkeit ε ergeben sich mit wenigen Iterationsschritten gemäß (1) bereits genaue Ergebnisse, d.h. das Verfahren " *konvergiert schnell* " zu einer Lösung.

Der Nachteil des Newtonverfahrens

Man benötigt die Ableitung $f'(x)$. Bei ungünstiger Wahl des Startwertes x_0 kann das Verfahren versagen und keine Nullstelle liefern. Das ist im hiesigen Beispiel bei $x_0 = 0$ der Fall. Um das zu verhindern, sollte der Startwert *hinreichend nahe* bei der gesuchten Lösung liegen.

II.3 Der Banachsche Fixpunktsatz

Schreiben wir die Rekursionsformel (3) für das Newtonverfahren um in die Form

$x_{k+1} = F(x_k)$ mit $F(x_k) \underset{def.}{=} x_k - \dfrac{f(x_k)}{f'(x_k)}$, so gilt wegen $\lim\limits_{k\to\infty} x_k = x^*$ offenbar

$x^* = \lim\limits_{k\to\infty} x_{k+1} = \lim\limits_{k\to\infty} F(x_k) = F(x^*)$. Das letzte Gleichheitszeichen ist garantiert, wenn F *stetig* ist.

Die Identität $x^* = F(x^*)$ heißt ***Fixpunktgleichung*** und der Wert x^* wird ***Fixpunkt*** von F genannt. Iterationen, wie insbesondere das Newtonverfahren lassen sich auf solche Fixpunktgleichungen zurückführen. Da sie in der Numerischen Mathematik eine wesentliche Rolle spielen, sei im folgenden kurz das Wichtigste zu diesem Thema zusammengestellt. Im Vordergrund steht dabei die Frage, unter welchen Bedingungen eine Iterationsfolge und damit insbesondere das Newtonverfahren *konvergiert* .

II.3.1 Definition

Sei $F : [a\,,\, b] \to \mathbb{R}$ eine auf dem abgeschlossenen Intervall [*a* , *b*] definierte Funktion.
F heißt ***kontrahierend*** auf [*a* , *b*], wenn gilt:

1. $F(x) \in [a,b]$ für alle $x \in [a,b]$, d.h. F bildet [*a* , *b*] *in sich* ab.
2. $\dfrac{|F(x)-F(y)|}{|x-y|} \le L$ für alle $x, y \in [a,b]$ mit einer sog. ***Lipschitz-Konstanten*** $L \in$] 0,1 [.

Punkt 2 bedeutet geometrisch, dass die Steigung *jeder Sekante* des Graphen von F betragsmäßig kleiner oder gleich L ist. Da die *Ableitung* als Tangentensteigung als Grenzwert von Sekantensteigungen erhalten wird, so gilt auch $|F'(x)| \le L < 1$ für alle $x \in [\,a\,,\,b\,]$. In der Tat folgt:

II.3.2 Satz

Sei $F : [a\,,\, b] \to \mathbb{R}$ eine auf dem abgeschlossenen Intervall [*a* , *b*] definierte Funktion.
F ist ***kontrahierend*** auf [*a* , *b*], wenn gilt:

1. $F(x) \in [a,b]$ für alle $x \in [a,b]$, d.h. F bildet [*a* , *b*] in sich ab.
2. F ist ***differenzierbar*** auf [*a* , *b*] mit $|F'(x)| \le L < 1$ für alle $x \in [a\,,\,b]$

Beweis:
Sei ohne Beschränkung der Allgemeinheit $a \le x < y \le b$. Für ein geeignetes $\vartheta \in [\,x\,,\,y\,]$ gilt nach dem Mittelwertsatz $\dfrac{F(x)-F(y)}{x-y} = F'(\vartheta)$ mithin $\dfrac{|F(x)-F(y)|}{|x-y|} = |F'(\vartheta)| \le L$, was zu beweisen war.

Der folgende Satz stellt nun sicher, wann eine Iterationsfolge *konvergiert*:

II.3.3 Banachscher Fixpunktsatz

Sei $F : [a\,,\, b] \to \mathbb{R}$ eine auf dem abgeschlossenen Intervall [*a* , *b*] definierte ***Kontraktion*** .

1. Es existiert ***genau ein*** Fixpunkt $x^* \in [a,b]$, d.h. es gilt $x^* = F(x^*)$.
2. Für jeden Startwert $x_0 \in [a,b]$ konvergiert die durch $x_{k+1} = F(x_k)$, $k = 0,1,2,3,\ ...$ definierte Zahlenfolge gegen x^* .

Beweisskizze:
Wir gehen der Einfachkeit halber davon aus, dass für die Gleichung $x^* = F(x^*)$ eine Lösung x^* existiert, welche mit der Iteration $x_{k+1} = F(x_k)$ bestimmt werden soll. Dazu schätzen wir die Abweichung $|x_k - x^*|$ der *k*-ten Näherung x_k von der Lösung x^* ab. Es ist nämlich:

$$\begin{aligned} |x_k - x^*| &= |F(x_{k-1}) - F(x^*)| \le L\,|x_{k-1} - x^*| \quad \text{weil } F \text{ kontrahierend ist} \\ &= L\,|F(x_{k-2}) - F(x^*)| \le L^2\,|x_{k-2} - x^*| \quad \text{weil } F \text{ kontrahierend ist} \\ &\le \ ... \ \le L^k\,|x_0 - x^*| \ . \end{aligned}$$

Wegen $0 < L < 1$ gilt $\lim\limits_{k\to\infty} L^k = 0 \ \Rightarrow \ \lim\limits_{k\to\infty} L^k\,|x_0 - x^*| = 0 \underset{\text{erst recht}}{\Rightarrow} \lim\limits_{k\to\infty} |x_k - x^*| = 0$ also $\lim\limits_{k\to\infty} x_k = x^*$.

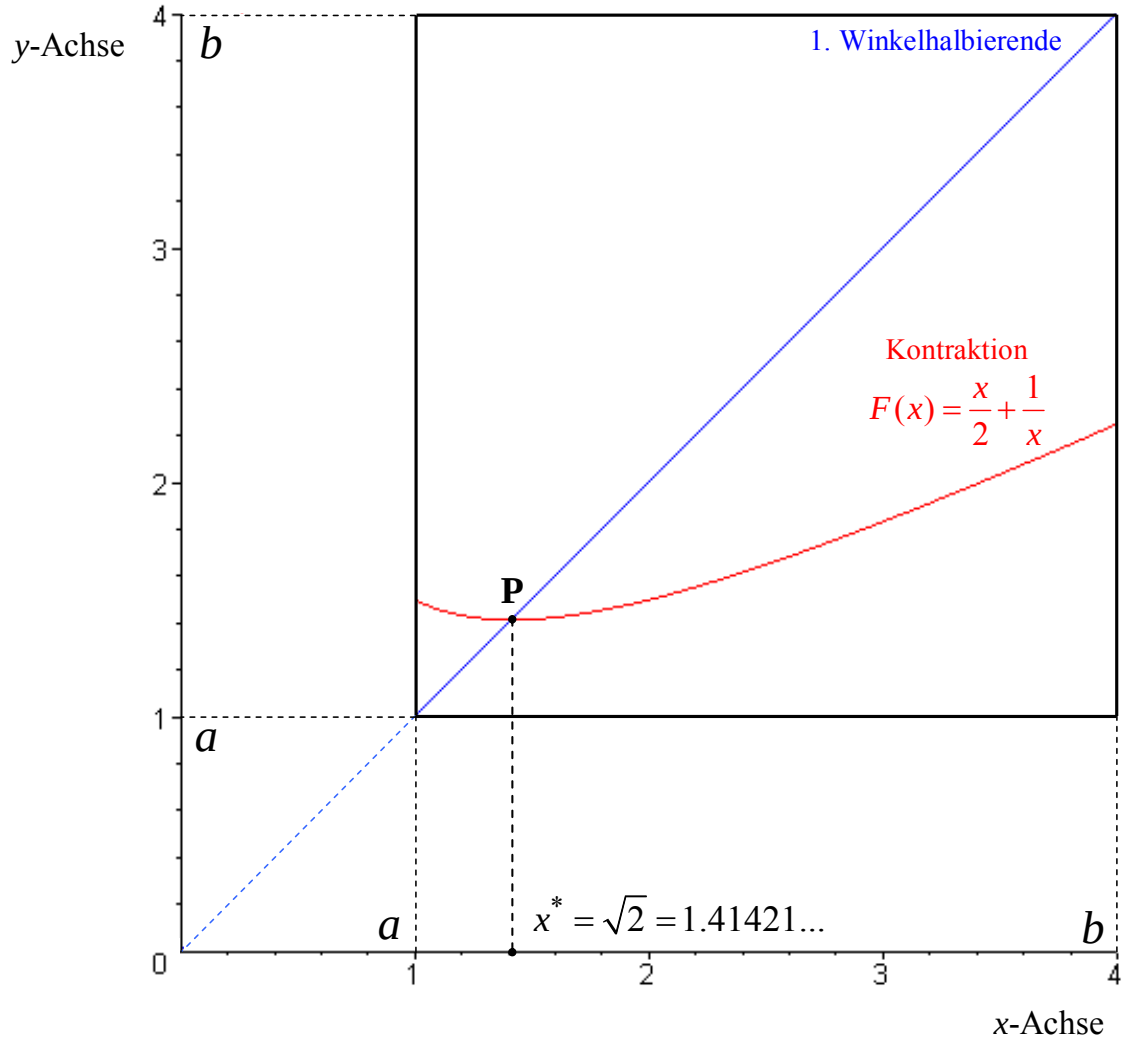
x^* ist der *einzige* Fixpunkt, denn wäre y^* ein weiterer solcher, dann wäre $\left|y^*-x^*\right| = \left|F\left(y^*\right)-F\left(x^*\right)\right| \le L\left|y^*-x^*\right|_{L<1} < \left|y^*-x^*\right|$ also $\left|y^*-x^*\right| \lneq \left|y^*-x^*\right|$ und das ist ein Widerspruch! Damit ist die Eindeutigkeit des Fixpunktes nachgewiesen.

Greifen wir wieder unser *Newtonverfahren* mit der Funktion $f(x) = x^2 - 2$ zur Bestimmung von $\sqrt{2}$ auf. Gemäß der Rekursionsformel (2) folgt $F(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x^2-2}{2x} = \frac{x}{2} + \frac{1}{x}$.

Das Definitionsintervall $[a\,,\,b]$ für F muss die Voraussetzungen zu Satz II.3.2 erfüllen. Das gelingt, wenn wir F auf dem Intervall $[a\,,\,b] = [\,1\,,\,4\,]$ betrachten, worin der zu bestimmende Wert $\sqrt{2}$ liegt. Man prüft dann leicht nach, dass F das Intervall $[\,1\,,\,4\,]$ *in sich* abbildet.

Ferner ist leichtersichtlich $\left|F'(x)\right| = \left|\frac{1}{2} - \frac{1}{x^2}\right| \le \frac{1}{2} < 1$ für alle $x \in [\,1\,,\,4\,]$.

Gemäß Satz II.3.2 ist somit F eine Kontraktion und mit jedem Startwert $x_0 \in [\,1\,,\,4\,]$ konvergiert die durch $x_{k+1} = F(x_k)$, $k = 0,1,2,3,\ \ldots$ definierte Folge gegen $\sqrt{2}$.



Unter Beachtung von Satz II.3.2 Punkt 2 mache man sich anschaulich klar:

Der *Graph* einer *Kontraktion* F ist eine Kurve (im obigen Bild rot), die in einem *Quadrat* liegt, das vom Definitionsintervall $[a\,,\,b]$ (hier $[1\,,\,4]$) im x-y-Koordinatensystem "aufgespannt" wird.
Die *Steigung* des roten Graphen (Ableitung $F'(x)$) ist überall im Definitionsintervall *betragsmäßig geringer* als die Steigung 1 der Winkelhalbierenden (im obigen Bild blau). Letztere verläuft *diagonal* durch das Quadrat. Unter diesen Umständen wird die rote Kurve die blaue Winkelhalbierende im Quadrat *stets* in *genau einem* Punkt P schneiden! Die x-Koordinate von P ist genau der Fixpunkt x^* von F also die Lösung der Fixpunktgleichung $x^* = F(x^*)$.

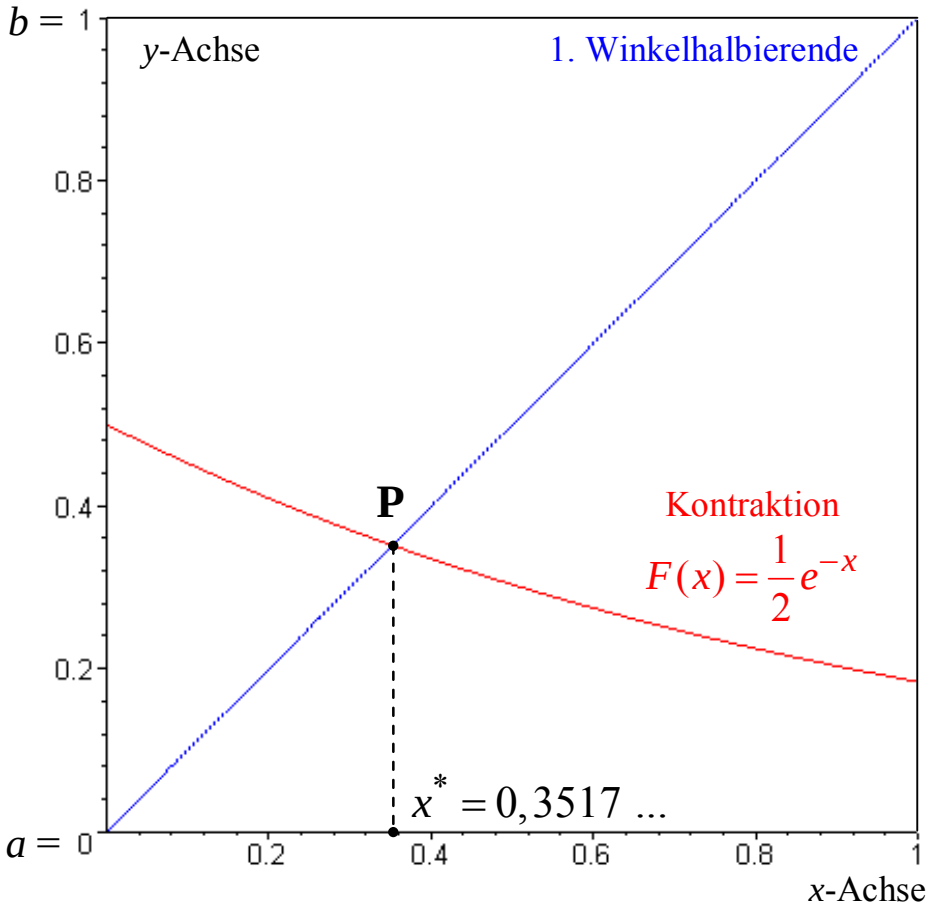
II.3.4 Beispiele mit MATLAB

Die vorangegangenen Überlegungen zeigen, dass die numerische Lösung der Gleichung $f(x) = 0$ auf die Lösung einer Fixpunktgleichung $x^* = F(x^*)$ zurückführt. Im folgenden sollen einige Beispiele dieser Art mit MATLAB gelöst werden. Dabei sollen auch weitere Möglichkeiten, die MATLAB z.B. für die Datenaufbereitung und -ausgabe anbietet, aufgezeigt werden.

Als Beispiel wollen wir die Gleichung $2x\,e^x = 1$ lösen.

Dazu formen wir sie äquivalent in eine *Fixpunktgleichung* um: $x = \frac{1}{2}e^{-x} \underset{def.}{=} F(x)$.

Anschaulich ist der *einzige* Schnittpunkt des Graphen von F mit der 1. Winkelhalbierenden zu ermitteln.



Wir betrachten F auf dem Intervall $[a\,,\,b] = [0\,,\,1]$. F ist dort streng monoton fallend mit $F(0) = \frac{1}{2}$ und $F(1) = \frac{1}{2e} = 0{,}1839\ldots$ Also ist $1 > F(0) \geq F(x) \geq F(1) > 0$ für alle $x \in [0\,,\,1]$, d.h. F bildet das Intervall $[a\,,\,b] = [0\,,\,1]$ *in sich* ab.

Ferner ist F differenzierbar mit $F'(x) = -\frac{1}{2}e^{-x}$ weshalb $|F'(x)| = \frac{1}{2}e^{-x} \leq \frac{1}{2} < 1$ für alle $x \in [a,b]$ gilt.

Gemäß Satz II.3.2 daher ist F eine Kontraktion und nach dem Banachschen Fixpunktsatz II.3.3 konvergiert mit jedem Startwert $x_0 \in [0\,,\,1]$ die durch $x_{k+1} = F(x_k)$, $k = 0,1,2,3,\ \ldots$ definierte Folge gegen die Lösung x^* der Fixpunktgleichung $x = F(x)$ d.h. gegen die Lösung x^* der zur Fixpunktgleichung äquivalenten Gleichung $2x\,e^x = 1$.

Zu beachten ist, dass *kein* Newtonverfahren zur Anwendung kommt! Der hier beschriebene Lösungsweg heißt ***Banachiteration***. Bei diesem Verfahren wird eine zu lösende Gleichung $f(x) = 0$ *äquivalent* in eine *Fixpunktgleichung* $F(x) = x$ derart umgeformt, dass die Voraussetzungen zu Satz II.3.2 erfüllt sind. Der Banachsche Fixpunktsatz garantiert dann die Lösung x^* der Fixpunktgleichung $F(x) = x$ und x^* ist dann auch die Lösung von $f(x) = 0$.

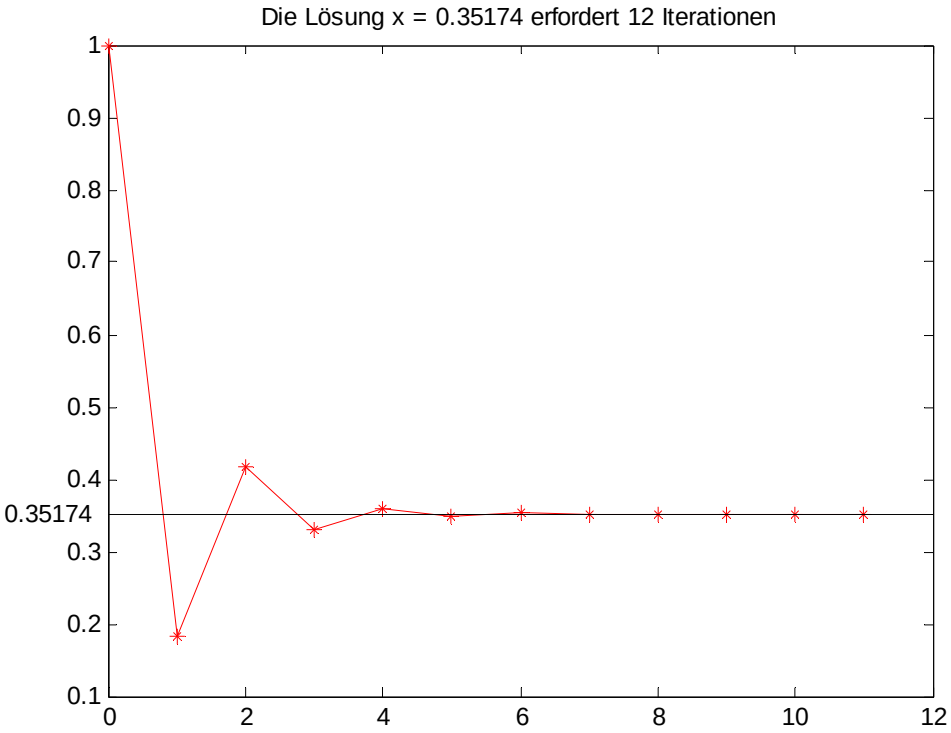
Das folgende script-file führt die beschriebene Banachiteration aus:

```
%Banachiteration für die Fixpunktgleichung x=F(x)=exp(-x)/2.
K=[];X=[];          %Index- und x-Wertematrix initialisieren.
x=1;x_vorher=0;     %Startwert x=1 und Vorgänger x_vorher=0 setzen.
Eps=1e-5;k=0;       %Genauigkeit Eps festlegen,Startwert k=0 setzen.
while abs(x-x_vorher)>=Eps    %Solange Genauigkeit nicht erreicht:
    x_vorher=x;     %Altes x mit aktuellem Wert belegen.
    K=[K,k];        %k zur Indexmatrix hinzufügen.
    X=[X,x];        %Aktuelles x zur x-Wertematrix hinzufügen.
    x=exp(-x)/2;    %x gemäß Fixpunktgleichung aktualisieren.
    k=k+1;          %k um 1 hochzählen.
end
plot(K,X,'red-*')   %Iterationsprozess graphisch veranschaulichen.
hold on
Achse1=[0,k];Achse2=[x,x];
plot(Achse1,Achse2,'black-');
title(['Die Lösung x = ',num2str(x),...
       ' erfordert ',int2str(k),' Iterationen']);
text(-1.5,x,num2str(x));   %Lösung x in der Graphik anzeigen.
hold off
fprintf('Ausgabe der Lösung mit Abweichung <  %.e\n',Eps)
fprintf('Es wurden %u Iterationen benötigt \n',k)     %Iterationen
fprintf('Die Gleichung hat die Lösung x = %.4f\n',x)  %mit Ergebnis
fprintf('Im folgenden die Liste der Iterationen: \n') %ausgeben.
for i=1:length(K) %length(K)=Anzahl der Listeneinträge.
    k=K(i);       %k für eine Listenzeile eintragen.
    x=X(i);       %x für eine Listenzeile eintragen.
    if k<10       %Liste erzeugen mit Angabe von k und xk.
        fprintf('k =  %u ergibt x( %u) = %.4f\n',k,k,x)
    else          %Für k<10 vor k ein Leerzeichen einfügen, um
        fprintf('k = %u ergibt x(%u) = %.4f\n',k,k,x)
    end           %unschöne Einrückungen in der Liste zu vermeiden.
end
```

Die Ergebnisliste erscheint im Command Window

```
Lösung der Gleichung 2*x*exp(x)=1 mit Abweichung <  1e-05
Es wurden 12 Iterationen benötigt
Die Gleichung hat die Lösung x = 0.3517
Im folgenden die Liste der Iterationen:
k =  0 ergibt x( 0) = 1.0000
k =  1 ergibt x( 1) = 0.1839
k =  2 ergibt x( 2) = 0.4160
k =  3 ergibt x( 3) = 0.3298
k =  4 ergibt x( 4) = 0.3595
k =  5 ergibt x( 5) = 0.3490
k =  6 ergibt x( 6) = 0.3527
k =  7 ergibt x( 7) = 0.3514
k =  8 ergibt x( 8) = 0.3519
k =  9 ergibt x( 9) = 0.3517
k = 10 ergibt x(10) = 0.3517
k = 11 ergibt x(11) = 0.3517
```

zusammen mit einer Graphik



Die Graphik linkerhand macht deutlich, wie die Iterationswerte x_k gegen die gesuchte Lösung konvergieren. Die Folgenglieder liegen abwechselnd oberhalb und unterhalb des Grenzwertes x^*. Man spricht dann von einer *alternierenden* Zahlenfolge.

Zum obigen script-file:

In Zeile 2 wird die Ausgabe der Iterationsfolge x_k , $k = 0, 1, 2, ...$ vorbereitet. K ist die Zeilenmatrix der Zahlen k und X die Zeilenmatrix der Folgenglieder x_k . Zu Beginn sind K und X "leere" Matrizen.
In Zeile 3 wird der Startwert $x = x_0$ festgelegt. Er muss im Definitionsintervall $[a , b] = [0 , 1]$ der Kontraktion F liegen. Es wird daher z.B. **`x = 1`** gesetzt. Der "Vorgänger" eines Folgengliedes x_k wird vor Beginn der **`while`**-Schleife mit **`x_vorher = 0`** initialisiert, ferner bekommt der Index k zu Beginn den Wert **`k = 0.`**
In Zeile 4 wird die gewünschte Genauigkeit für die Lösung x^* festgesetzt. 4 Nachkommastellen sollen für x^* ermittelt werden, d.h. der Absolutbetrag $|x_{k+1} - x_k|$ muss den Wert 10^{-5} unterschreiten: **`Eps=1e-5`** .
Solange dies nicht der Fall ist, wird die **`while`**-Schleife in Zeile 5 bis 11 ausgeführt.
Die *erstmalige Ausführung* der **`while`**-Schleife ist wegen **`x = 1`** und **`x_vorher = 0`** sichergestellt: **`x`** und **`x_vorher`** werden darin aktualisiert. **`x_vorher`** muss als erstes mit dem Wert **`x = 1`** belegt werden (Zeile 6) , damit K und X ihre ersten Elemente K(1) = **`k = 0`** und X(1) = **`x = 1`** erhalten (Zeile 7 und 8).
Dann wird **`x`** mit dem Folgewert **`exp(-x)/2`** belegt und der Index vom Initialwert 0 auf 1 hochgezählt.
Nach dem 1. Schleifendurchlauf gilt K=[0] und X=[1] und es ist k = 1.
Nach dem 2. Schleifendurchlauf gilt K = [0 , 1] und $X = [1, \frac{1}{2e}]$ und es ist k = 2 usw.
Ist der letzte Schleifendurchlauf vollzogen, so gibt **`k`** in Zeile 10 die *Elementeanzahl* n von K und X an.
Es ist $K = [0 , 1 , 2 , \ ... \ , n - 1]$ und $X = [x_0 , x_1 , x_2 , \ ... \ , x_{n-1}]$. Gemäß den Vorgaben ergibt sich **`k`** = n = 12.
Der Wert für **`x`** in Zeile 9 wird in diesem *letzten* Schleifendurchlauf nicht mehr in die Matrix X übernommen! Er kann jedoch als Ausgabewert in Zeile 14, 16, 18 und 22 verwendet werden (statt **`X(end)`**, dem letzten Element der Matrix X) und ist gleich 0,35174.
Zur Ausgabe der Graphik dienen die Zeilen 12 bis 19. Der Befehl **`plot(K,X)`** (mit Option **`'red-*'`**) in Zeile 12 erzeugt ein Graphikfenster und verbindet in einem skalierten kartesischen Koordinatensystem die Wertepaare (k , x_k) durch einen geschlossenen Streckenzug. Die zusätzliche Option **`'red-*'`** erzeugt *rote* Streckenzüge und markiert die Punkte (k , x_k) durch *Sterne*. Weiterführende Infos hierzu in der MATLAB-Hilfe unter dem Suchbegriff "plot".
Zeile 14 und 15 erzeugen eine schwarze Parallele zur skalierten Abszisse des Graphikfensters in Höhe des x-Wertes 0,35174 (**`Achse2=[x,x]`**). Die Länge der Parallele ist hier gleich 12 (**`Achse1=[0,k]`**).
Mit dem Befehl **`title`** in Zeile 16 und 17 kann die obige Graphik mit einer Überschrift versehen werden.
Die Option **`num2str(x)`** konvertiert dazu die Gleitkommazahl x zur Anzeige in einen String.
Die Option **`int2str(x)`** konvertiert dazu die ganze Zahl k zur Anzeige in einen String.

Nach Wunsch kann an einer beliebigen Stelle in der Graphik weiterer Text eingefügt werden.
Der Befehl **`text(-1.5,x,num2str(x))`** in Zeile 18 positioniert z.B. den Zahlenwert für x an eine gewünschte Stelle: Dazu ist der Zahlenwert ebenfalls in einen String zu konvertieren, davor durch Komma getrennt sind die Koordinaten für das erste Stringzeichen anzugeben.
Die Befehle **`hold on`** und **`hold off`** in Zeile 13 bzw 19 folgen unmittelbar nach dem plot-Befehl in Zeile 12 und schließen alles ein, was in der Graphik zusätzlich anzuzeigen ist.
Der Befehl **`fprintf`** in Zeile 20 bis 24 gestattet individuelle Möglichkeiten zur Ausgabe von Text und Listen im Command Window. Das betrifft vorallem die Ausgabe von *Zahlen* in unterschiedlichen Formaten *innerhalb eines Textes* mit Hilfe von Steuerzeichen.

Jede Zahlenausgabe ist nach einem Leerzeichen mit dem %-*Zeichen* zu codieren:
Zeile 20: **`%.e`** gibt `1e-005` für **`Eps=1e-5`** aus.
Zeile 21: **`%u`** gibt eine ganze Zahl **`k`** im Text aus.
Zeile 22: **`%.4f`** gibt die Gleitkommazahl **`x`** mit 4 Nachkommastellen aus.
Zeile 23: **`\n`** ist das Kommandozeichen für eine neue Zeile (auch in Zeile 20 bis 22 erforderlich).

Hinter dem Text in Hochkommata (innerhalb des **`fprintf`**-Befehls) steht nach einem weiteren Komma die Variable, deren Wert im Text anzuzeigen ist (nicht in Zeile 23).
Weiterführende Infos hierzu in der MATLAB-Hilfe unter dem Suchbegriff "fprintf".

Die **`for`**-Schleife in Zeile 24 bis 32 gibt schließlich die Liste der Iterationswerte (k , x_k) aus.
Mit length(K) = 12 im obigen script-file sind das 12 Listeneinträge (siehe ebenfalls oben).

Wichtige Anmerkung:

Matlab markiert im obigen script-file in Zeile 7 und 8 die Variablen K bzw. X und gibt bei Position des Cursors auf diese Variablen folgende Quickinfo aus:

"The variable 'K' (bzw. 'X') appears to change size on every loop iteration. Consider preallocating for speed."

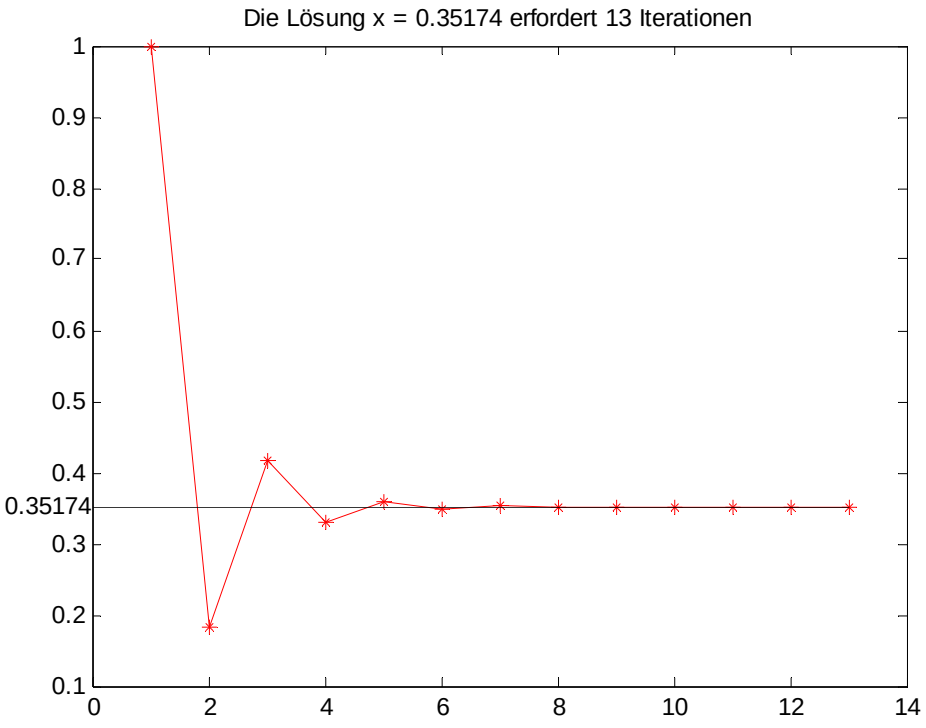
Die Codierung gemäß Zeile 2, 7 und 8 führt zu hoher Rechenleistung mit entsprechender Programmlaufzeit!
Zur Behebung dieses Umstandes "allokiere" man K und X, indem die Zeilen 2 – 11 wie folgt geändert werden:

```
 1  %Banachiteration für die Fixpunktgleichung x=F(x)=exp(-x)/2.
 2-  K(1,1)=1;X(1,1)=1;  %Anfangswerte für Index- und x-Wertematrix.
 3-  x=1;x_vorher=0;     %Startwert x=1 und Vorgänger x_vorher=0 setzen.
 4-  Eps=1e-5;k=1;       %Genauigkeit Eps festlegen, Startwert k=1 setzen.
 5-  while abs(x-x_vorher)>=Eps    %Solange Genauigkeit nicht erreicht:
 6-      x_vorher=x;     %Altes x mit aktuellem Wert belegen.
 7-      x=exp(-x)/2;    %x gemäß Fixpunktgleichung aktualisieren.
 8-      k=k+1;          %k um 1 hochzählen.
 9-      X(1,k)=x;       %Aktuelles x zur x-Wertematrix hinzufügen.
10-      K(1,k)=k;       %k zur Indexmatrix hinzufügen.
11-  end
```

Die Ausgabeliste im Command Window sieht dann so aus

```
Lösung der Gleichung 2*x*exp(x)=1 mit Abweichung <  1e-05
Es wurden 13 Iterationen benötigt
Die Gleichung hat die Lösung x = 0.3517
Im folgenden die Liste der Iterationen:
k =  1 ergibt x( 1) = 1.0000
k =  2 ergibt x( 2) = 0.1839
k =  3 ergibt x( 3) = 0.4160
k =  4 ergibt x( 4) = 0.3298
k =  5 ergibt x( 5) = 0.3595
k =  6 ergibt x( 6) = 0.3490
k =  7 ergibt x( 7) = 0.3527
k =  8 ergibt x( 8) = 0.3514
k =  9 ergibt x( 9) = 0.3519
k = 10 ergibt x(10) = 0.3517
k = 11 ergibt x(11) = 0.3517
k = 12 ergibt x(12) = 0.3517
k = 13 ergibt x(13) = 0.3517
```

zusammen mit der Graphik:



Die Allokierung bereitet Schritt für Schritt K und X als Zeilenmatix vor: `K=K(1,k)` und `X=X(1,k)`. Dabei enthält `k` den Wert, der nach Ablauf der letzten **`while`**-Schleife gemäß Zeile 8 erreicht wird.

Beachte: Statt x_0 lautet hier der Startwert x_1. Entsprechend beginnt die Ausgabeliste mit `k = 1` und `x( 1)`.

allocate (engl.) bedeutet *zuweisen*. In der Informatik ist damit das Zuweisen von Speicherplatz gemeint. Sind nur *wenige* Schritte in einer Iteration zu erwarten, so kann auf eine Allocation jedoch verzichtet werden. Wir werden davon auch Gebrauch machen.

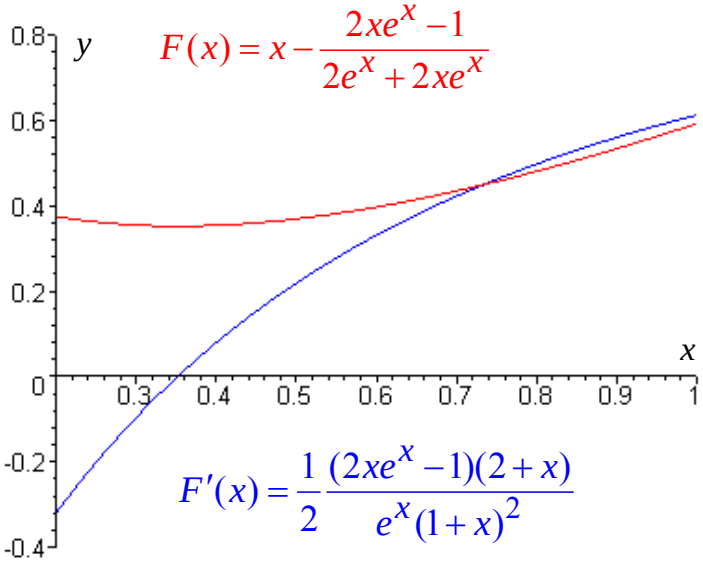
II.3.5 Newtoniteration und Konvergenzgeschwindigkeit:

Die Gleichung $2x\,e^x = 1$ kann auch mit dem *Newtonverfahren* gelöst werden.

Die Lösung x^* von $2x\,e^x = 1$ ist dann als *Nullstelle* der Funktion $f(x) \underset{def.}{=} 2xe^x - 1$ aufzufassen

und statt $F(x) = \frac{1}{2}e^{-x}$ in der Banachiteration ist gemäß der Newtonschen Rekursionsformel (2)

$$F(x) = x - \frac{f(x)}{f'(x)} = x - \frac{2xe^x - 1}{2e^x + 2xe^x}$$ zu setzen.



Das Bild linkerhand zeigt die Graphen von F und der Ableitung F', die mit der Quotientenregel erhalten wird. Wählt man das Intervall $[a\,,\,b] = [0.2\,,\,1]$, so bildet F offensichtlich $[a\,,\,b]$ *in sich* ab. Ferner ist auf dem Intervall die Abschäzung $|F'(x)| < 0{,}8 < 1$ gültig. Gemäß Satz II.3.2 ist somit F auf $[a\,,\,b]$ eine Kontraktion und nach dem Fixpunktatz Satz II.3.3 von Banach gibt es genau eine Lösung x^* der Fixpunktgleichung $x = F(x)$. Diese Lösung ist dann auch Lösung der Gleichung $f(x) = 2xe^{-x} - 1 = 0$ d.h. $2xe^{-x} = 1$.

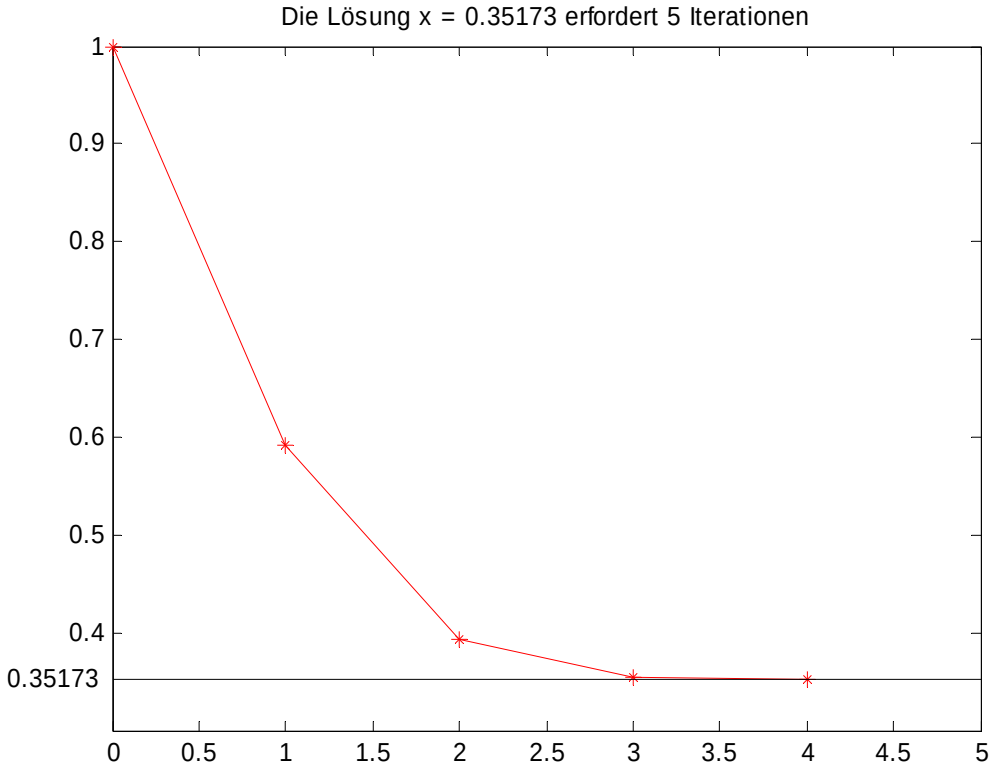
Insbesondere ist die Newtoniteration somit eine Banachiteration.

Zur Ausführung der Newtoniteration mit MATLAB muss im obigen script-file lediglich der Befehl **`x=exp(-x)/2;`** durch den Befehl **`x=x-(2*x*exp(x)-1)/(2*exp(x)+2*x*exp(x));`** ersetzt werden. Ferner ist Zeile 18 anzupassen: **`text(-0.6,x,num2str(x));`**
Der Startwert kann bei **`x = 1`** belassen werden (Zeile 3).

Man erhält dann im Command Window (ohne Allokation) die Listenausgabe

```
Ausgabe der Lösung mit Abweichung < 1E-005
Es wurden 5 Iterationen benötigt
Die Gleichung hat die Lösung x = 0.3517
Im folgenden die Liste der Iterationen:
k =  0 ergibt x( 0) = 1.0000
k =  1 ergibt x( 1) = 0.5920
k =  2 ergibt x( 2) = 0.3939
k =  3 ergibt x( 3) = 0.3532
k =  4 ergibt x( 4) = 0.3517
```

zusammen mit der Graphik:



Die Graphik zeigt eine schnellere Konvergenz der Iterationswerte x_k "von oben her" gegen die gesuchte Lösung $x^* = 0{,}3517$ ... Es sind nur 5 Iterationsschritte notwendig. Die Banachiteration benötigt 12 Schritte. Wegen der "schnelleren Konvergenz" ist das Newtonverfahren daher in der Praxis favorisiert.

Im folgenden sei die "***Konvergenzgeschwindigkeit*** " näher untersucht.

Konvergenz der Banachiteration

Gemäß dem Fixpunktsatz II.3.3 von Banach gilt von Iterationsschritt zu Iterationsschritt für die Kontraktion F die Abschätzung $\left|x_{k+1}-x^*\right| = \left|F\left(x_k\right)-F\left(x^*\right)\right| \le L\left|x_k-x^*\right|$ also $\left|x_{k+1}-x^*\right| \le L\ \left|x_k-x^*\right|$.

d.h. die Verkleinerung des Fehlers $\left|x_{k+1}-x^*\right|$ gegenüber $\left|x_k-x^*\right|$ erfolgt *linear* .

Man sagt: Die Banachiteration *konvergiert linear* .

Konvergenz der Newtoniteration

In wieweit ist die Newtoniteration "schneller"?
Konvergiert das Newtonverfahren *immer* , wenn der Startwert nur nahe genug bei der gesuchten Lösung liegt? Diese Frage hat deswegen Bedeutung, weil im allgemeinen der Nachweis von F als Kontraktion sehr mühsam sein kann (siehe Graphik unter II.3.5). Der folgende Satz (**für besonders Interessierte!**) gibt darauf unter den folgenden Bedingungen Auskunft:

Satz II.3.6 zur Newtoniteration (Hinweis: Der Leser kann den Beweis überspringen!)

Sei $f : [\,c\,,d\,] \to \mathbb{R}$ eine 2-mal stetig differenzierbare Funktion und $x^* \in]\,c\,,d\,[$ eine *einfache* Nullstelle von f, d.h. es gelte $f(x^*) = 0$ und $f'(x^*) \neq 0$. Dann gilt:

1. Liegt der Startwert $x_0 \in]\,c\,,d\,[$ als Anfangsnäherung *nahe genug* bei x^*, so konvergiert die Iterationsfolge $x_{k+1} \underset{def.}{=} x_k - \dfrac{f(x_k)}{f'(x_k)}$, $k = 0,1,2,\ \ldots$ gegen die Nullstelle x^*.
2. Ferner gilt $\left|x_{k+1} - x^*\right| \le K \left|x_k - x^*\right|^2$ für $k = 0\,,\,1\,,\,2\,,\,\ldots$ mit einer Konstanten $K > 0$.
   Man sagt: Die Newtoniteration *konvergiert quadratisch*.

Beweis für Aussage 1:

Wie bereits oben geschehen erhält man für f vermöge $F(x) \underset{def.}{=} x - \dfrac{f(x)}{f'(x)}$ beginnend mit einem Startwert x_0 die Newtoniteration $x_{k+1} = F(x_k) = x_k - \dfrac{f(x_k)}{f'(x_k)}$ für $k = 0\,,\,1\,,\,2\,,\,\ldots$

<u>Wir zeigen, dass gemäß Satz II.3.2 F eine Kontraktion ist.</u>

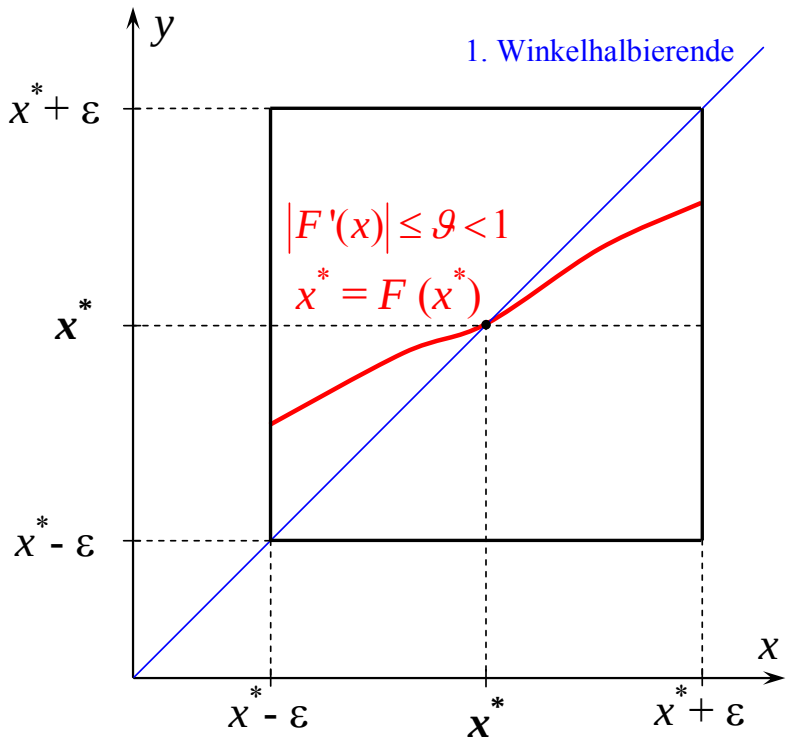
Dazu bestimmen wir zuerst die Ableitung von F:

Mit der Quotientenregel ergibt sich $F'(x) = 1 - \dfrac{f'(x)^2 - f(x)\,f''(x)}{f'(x)^2} = \dfrac{f(x)\,f''(x)}{f'(x)^2}$.

Wegen $f(x^*) = 0$ und $f'(x^*) \neq 0$ folgt $F'(x^*) = 0$.

Da f 2-mal stetig differenzierbar ist, so ist F' stetig in einem Intervall $[x^* - \varepsilon, x^* + \varepsilon] \subset]\,c,d\,[$ mit genügend kleinem $\varepsilon > 0$ und es ist wegen $F'(x^*) = 0$ sodann $|F'(x)| \le \vartheta < 1$ für alle $x \in [x^* - \varepsilon, x^* + \varepsilon]$.

Wegen $F(x^*) = x^* - \dfrac{f(x^*)}{f'(x^*)} \underset{s.o.}{=} x^*$ sieht der Graph von F im wesentlichen so aus:



Wegen $|F'(x)| \le \vartheta < 1$ für alle $x \in [x^* - \varepsilon, x^* + \varepsilon]$ ist die Tangentensteigung des Graphen von F (rot dargestellt) überall im Intervall $[x^* - \varepsilon, x^* + \varepsilon]$ *kleiner* als die Steigung 1 der 1. Winkelhalbierenden (blau dargestellt). Wegen $F(x^*) = x^*$ verläuft die rote Kurve deswegen im linkerhand dargestellten Quadrat, d.h. F bildet das Intervall $[x^* - \varepsilon, x^* + \varepsilon]$ *in sich* ab. Gemäß Satz II.3.2. ist damit F auf dem Intervall $[x^* - \varepsilon, x^* + \varepsilon]$ eine *Kontraktion* und nach dem Banachschen Fixpunktsatz konvergiert für alle Startwerte x_0 im genanten Intervall die Folge $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$, $k = 0,1,2,\ \ldots$

gegen den eindeutigen Grenzwert x^*. Dies zeigt, dass x_0 "nahe genug" bei x^* liegen muss.

Beweis für Aussage 2:

Gemäß dem Satz von Taylor gilt für die Funktion F für alle $x \in [x^* - \varepsilon, x^* + \varepsilon]$

$F(x) = F(x^*) + F'(x^*)(x - x^*) + \frac{1}{2}F''(\xi)(x - x^*)^2$ mit geeignetem fixen $\xi \in [x^* - \varepsilon, x^* + \varepsilon]$.

Wegen $F(x^*) = x^*$ und $F'(x^*) = \frac{f(x^*)f''(x^*)}{f'(x^*)^2} = 0$ folgt $F(x) = x^* + \frac{1}{2}F''(\xi)(x - x^*)^2$

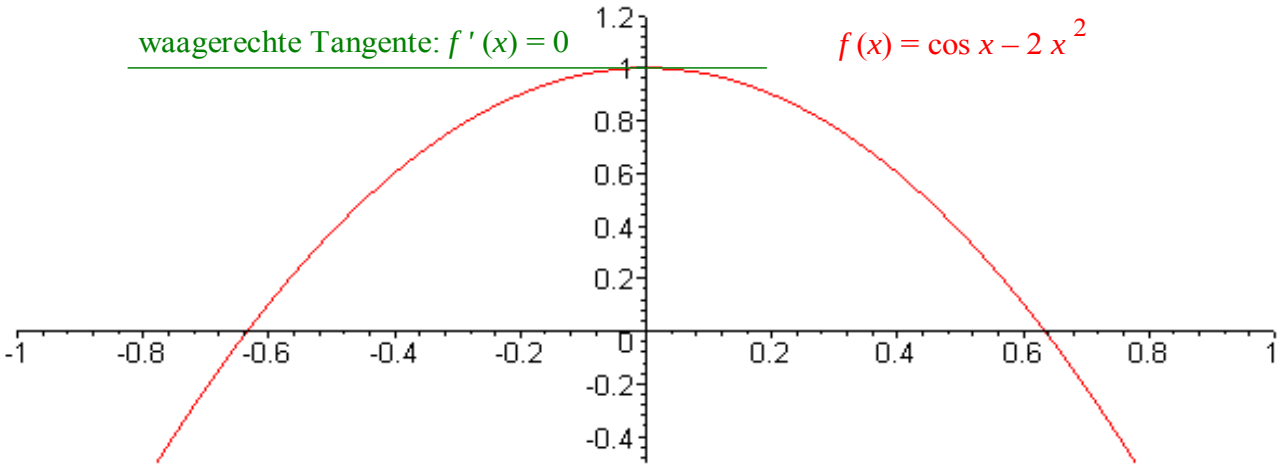
also $F(x) - x^* = \frac{1}{2}F''(\xi)(x - x^*)^2$ also insbesondere $F(x_k) - x^* = \frac{1}{2}F''(\xi)(x_k - x^*)^2 = x_{k+1} - x^*$.

Der Term $\frac{1}{2}F''(\xi)$ ist eine fixe Konstante und wir erhalten somit

$\left|x_{k+1} - x^*\right| = \left|\frac{1}{2}F''(\xi)\right|\left|x_k - x^*\right|^2 \le K\left|x_k - x^*\right|^2$ mit geeignetem $K > 0$, was zu beweisen war.

Das Newtonverfahren konvergiert somit *quadratisch* und ist damit "schneller" also die gewöhnliche obige Banachiteration. Die *Quadratur* einer bereits sehr kleinen Abweichung $\left|x_k - x^*\right|$ bewirkt nämlich nochmals eine deutliche Verkleinerung. Die quadratische Konvergenz heißt auch *Konvergenz 2-ter Ordnung* .
Erfüllt die Nullstelle in Satz II.3.6 zusätzliche Voraussetzungen, so kann das Newtonverfahren mit noch höherer Ordnung konvergieren. Darauf sei hier aber nicht weiter eingegangen. Unter genannten Voraussetzungen konvergiert das Newtonverfahren *zumindest* quadratisch.

Zum Schluss nochmals ein Beispiel einer Newtoniteration *ohne* Ausgabe einer Liste mit Graphik. In diesem Fall sind im script-file unter II.3.4 die Matrizen K und X, die ja für die Liste und die Graphik benötigt wurden überflüssig. Es sind dies die Zeilen 2 , 7 , 8 , 12-19 sowie 23-32.



Als Beispiel soll die Gleichung $f(x) = \cos x - 2x^2 = 0$ gelöst werden. Der achsensymmetrische Graph von f ist im obigen Bild dargestellt. Es gibt nur 2 Nullstellen von f nahe bei $0{,}6$ und $-0{,}6$.
Zur genauen Bestimmung setzen wir gemäß der Newtoniteration

$F(x) = x - \frac{f(x)}{f'(x)} = x - \frac{\cos x - 2x^2}{-\sin x - 4x}$ und beachten $f'(x) = -\sin x - 4x \neq 0$ für $x \neq 0$.

An der Stelle $x = 0$ hat der Graph eine waagerechte Tangente, d.h. dort ist $f'(0) = 0$.

Mit dem Startwert $x = 0$ wird also das Newtonverfahren versagen.
Das script-file zur Auffindung einer Nullstelle von f sieht so aus:

```
 1   %Newtoniteration mit Genauigkeitsvorgabe
 2   %und maximal zulässiger Anzahl von Iterationen
 3-  k=0;k_max=20;
 4-  x=1;x_vorher=0;
 5-  Eps=1e-8;
 6-  while abs(x-x_vorher)>=Eps && k<=k_max   %&& bedeutet "und".
 7-      x_vorher=x;
 8-      x=x-(cos(x)-2*x^2)/(-sin(x)-4*x);
 9-      k=k+1;
10-  end
11-  if k==0
12-      fprintf('Wegen ungünstiger Wahl des Startwertes \n')
13-      fprintf('wurden keine Iterationen ausgeführt! \n')
14-  else
15-      fprintf('Ausgabe der Lösung mit Abweichung < %.e\n',Eps)
16-      fprintf('Es wurden %u Iterationen ausgeführt \n',k)
17-      fprintf('Die Gleichung hat die Lösung x = %.7f\n',x)
18-  end
19-  if k>k_max
20-      fprintf('Die zulässige Iterationenanzahl ist überschritten \n')
21-  end
```

Der Programmlauf (wegen $f'(x) \neq 0$ für $x \neq 0$ mit *quadratischer* Konvergenz) wird auf eine maximale Anzahl von Iterationen beschränkt (Variable **k_max**) , falls man den Startwert **x** nicht nahe genug bei der gesuchten Nullstelle gewählt hat.
Wählt man den Startwert **x=0** , so versagt wie erwähnt das Verfahren wegen der waagerechten Tangente. Die **while**-Schleife wird dann wegen **x-x_vorher=0** nicht ausgeführt, d.h. es bleibt bei **k=0**. In diesem Fall wird die Meldung gemäß Zeile 12 und 13 ausgegeben.
Wählt man den Startwert **x** von 0 verschieden , so ist das Verfahren stets erfolgreich. Es wird die Lösung **x** ausgegeben mit den Meldungen gemäß Zeile 15, 16 und 17, auch wenn **k_max** überschritten wird. Die diesbezügliche zusätzliche Meldung ist in Zeile 20 codiert. Es werden dann 21 Interationen angegeben mit einem ggfs. ungenauen Lösungswert, da keine weiteren Iterationen ausgeführt werden (Zeile 6)

Die Ausgabe im Command Window mit Startwert **x=1** sieht so aus:

```
Ausgabe der Lösung mit Abweichung < 1e-008
Es wurden 5 Iterationen ausgeführt
Die Gleichung hat die Lösung x = 0.6345599
```

Ist beispielsweise **x=-1** also als Startwert negativ, so ergibt sich die Lösung `x =-0.6345599`.
Der Grund ist die Achsensymmetrie der Funktion $f(x) = \cos x - 2x^2$. Weitere Nullstellen für f gibt es wie erwähnt nicht.

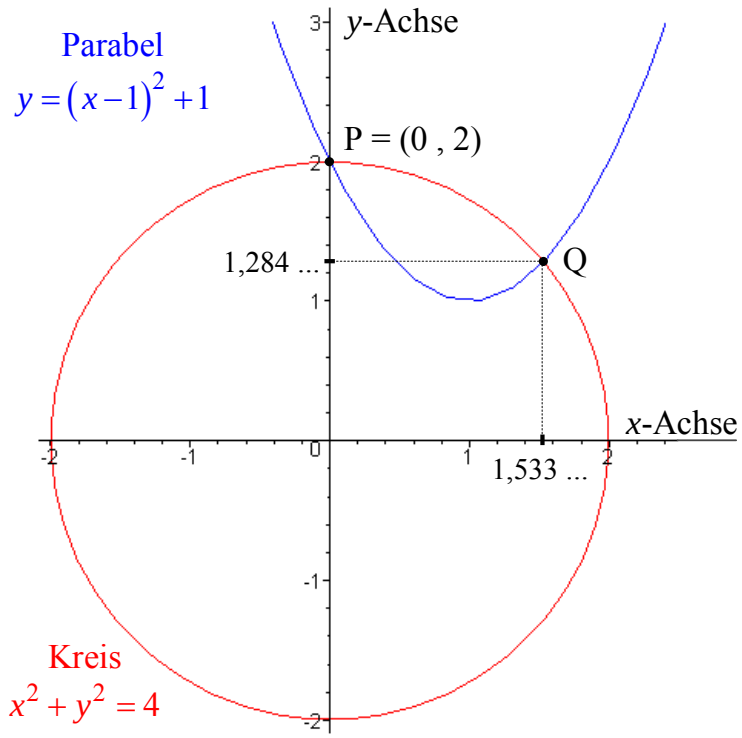
II.4 Das Newtonverfahren zur Lösung von Gleichungssystemen

Bisher haben wir nur *einzelne* Gleichungen numerisch gelöst. Im folgenden soll die numerische Lösung von *Gleichungssystemen* diskutiert werden. Als Beispiel betrachten wir ein Gleichungsystem mit *zwei* Gleichungen:

(4) $(x-1)^2 - y + 1 = 0$ Diese Gleichung stellt eine Parabel dar: $y = (x-1)^2 + 1$

(5) $x^2 + y^2 - 4 = 0$ Diese Gleichung stellt einen Kreis mit Radius 2 dar: $x^2 + y^2 = 4$

Das folgende Bild veranschaulicht die Situation:



Das Gleichungssystem (4),(5) hat genau zwei Zahlenpaare als Lösungen. Es sind dies die beiden Schnittpunkte P und Q von Kreis und Parabel.
Das Zahlenpaar (0 , 2) ist noch leicht zu ermitteln, das andere Zahlenpaar muss jedoch numerisch bestimmt werden.
Um unsere vorherigen Überlegungen anwenden zu können, definieren wir

$$f_1(x,y) \underset{def.}{=} (x-1)^2 - y + 1$$

$$f_2(x,y) \underset{def}{=} x^2 + y^2 - 4$$

und schließlich $f : \mathbb{R}^2 \to \mathbb{R}^2$ durch

$$f(x,y) \underset{def.}{=} \begin{pmatrix} f_1(x,y) \\ f_2(x,y) \end{pmatrix} = \begin{pmatrix} (x-1)^2 - y + 1 \\ x^2 + y^2 - 4 \end{pmatrix}.$$

Unsere Aufgabe ist somit die Lösung der Gleichung $f(x,y) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

Gesucht sind also die "Nullstellen" der Funktion $f : \mathbb{R}^2 \to \mathbb{R}^2$.

Das Newtonverfahren mit der Iterationsvorschrift $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$, $k = 0,1,2, \ldots$ lässt sich auf die Funktion $f : \mathbb{R}^2 \to \mathbb{R}^2$ nun folgendermaßen übertragen:

1. Für x_k ist $\begin{pmatrix} x_k \\ y_k \end{pmatrix}$ zu setzen.
2. Für $f(x_k)$ ist $\begin{pmatrix} f_1(x_k, y_k) \\ f_2(x_k, y_k) \end{pmatrix}$ zu setzen
3. Für $f'(x_k)$ ist $J(x_k, y_k) = \begin{pmatrix} \dfrac{\partial f_1}{\partial x}(x_k, y_k) & \dfrac{\partial f_1}{\partial y}(x_k, y_k) \\ \dfrac{\partial f_2}{\partial x}(x_k, y_k) & \dfrac{\partial f_2}{\partial y}(x_k, y_k) \end{pmatrix}$ zu setzen.

   Da f eine Funktion von 2 Veränderlichen x und y bestehend aus 2 Komponenten f_1 und f_2 ist, so gibt es 4 verschiedene sog. ***partielle Ableitungen*** . Bei Funktionen mehrerer Veränderlicher wird statt dem lateinischen "d" das runde "∂ " verwendet. Die "Ableitung" J von f ist eine *Matrix* und wird als ***Jacobi-Matrix*** bezeichnet.
4. Da $f'(x_k)$ in der Newtoniteration im Nenner steht, also $f(x_k)^{-1}$ benötigt wird, so ist auch hier die ***Inverse der Jacobi-Matrix*** anzuwenden.

Die in Punkt 1 – 4 genannten Übertragungen möge der Leser akzeptieren, eine Begründung kann hier nicht gegeben werden. Dies ist einem fundierten fortgeschrittenen Mathematikstudium vorbehalten.

Für die Funktion $f:\mathbb{R}^2\to\mathbb{R}^2$ lautet die Newtoniteration statt $x_{k+1}=x_k-\dfrac{f(x_k)}{f'(x_k)}$ also so:

$$(6)\qquad \begin{pmatrix} x_{k+1}\\ y_{k+1}\end{pmatrix}=\begin{pmatrix} x_k\\ y_k\end{pmatrix}-\underbrace{\begin{pmatrix} \dfrac{\partial f_1}{\partial x}(x_k,y_k) & \dfrac{\partial f_1}{\partial y}(x_k,y_k)\\ \dfrac{\partial f_2}{\partial x}(x_k,y_k) & \dfrac{\partial f_2}{\partial y}(x_k,y_k)\end{pmatrix}^{-1}\begin{pmatrix} f_1(x_k,y_k)\\ f_2(x_k,y_k)\end{pmatrix}}_{\text{Als Matrizenprodukt zu berechnen}}.$$

Gemäß obiger Definition von f gilt

$\dfrac{\partial f_1}{\partial x}(x,y)=2x-2$, $\dfrac{\partial f_1}{\partial y}(x,y)=-1$, $\dfrac{\partial f_2}{\partial x}(x,y)=2x$ sowie $\dfrac{\partial f_2}{\partial y}(x,y)=2y$ mithin statt (6)

$$(7)\qquad \begin{pmatrix} x_{k+1}\\ y_{k+1}\end{pmatrix}=\begin{pmatrix} x_k\\ y_k\end{pmatrix}-\begin{pmatrix} 2x_k-2 & -1\\ 2x_k & 2y_k\end{pmatrix}^{-1}\begin{pmatrix} (x_k-1)^2-y_k+1\\ x_k^2+y_k^2-4\end{pmatrix}.$$

Die Inverse einer (2 × 2)-Matrix $A=\begin{pmatrix} a_{11} & a_{12}\\ a_{21} & a_{22}\end{pmatrix}$ berechnet sich bekanntlich so:

$$(8)\qquad A^{-1}=\frac{1}{\det A}\begin{pmatrix} a_{22} & -a_{12}\\ -a_{21} & a_{11}\end{pmatrix}=\frac{1}{a_{11}a_{22}-a_{21}a_{12}}\begin{pmatrix} a_{22} & -a_{12}\\ -a_{21} & a_{11}\end{pmatrix}.$$

In (7) liefert das

$$\begin{pmatrix} 2x_k-2 & -1\\ 2x_k & 2y_k\end{pmatrix}^{-1}=\frac{1}{2y_k(2x_k-2)+2x_k}\begin{pmatrix} 2y_k & 1\\ -2x_k & 2x_k-2\end{pmatrix}=\frac{1}{4y_k(x_k-1)+2x_k}\begin{pmatrix} 2y_k & 1\\ -2x_k & 2(x_k-1)\end{pmatrix}$$

und somit vermöge (7) die Newtoniteration:

$$(9)\qquad \boxed{\begin{pmatrix} x_{k+1}\\ y_{k+1}\end{pmatrix}=F(x_k,y_k)\underset{def.}{=}\begin{pmatrix} x_k\\ y_k\end{pmatrix}-\frac{1}{4y_k(x_k-1)+2x_k}\begin{pmatrix} 2y_k & 1\\ -2x_k & 2(x_k-1)\end{pmatrix}\begin{pmatrix} (x_k-1)^2-y_k+1\\ x_k^2+y_k^2-4\end{pmatrix}}$$

für $k=$ 0 , 1 , 2 , ...

Wählt man als Startwert $(x_0,y_0)=(0,1)$, so liefert die Iteration in (9) die Lösung $(x^*,y^*)=(0,2)$ also im obigen Bild den Punkt P. Wählt man als Startwert $(x_0,y_0)=(1,1)$, so liefert die Iteration in (9) die Lösung $(x^*,y^*)=(1.533...\ ,\ 1.284...)$ also im obigen Bild den Punkt Q.

Wir wollen die Berechnung mittels (9) mit MATLAB ausführen. Dazu sind die Gleichungen für x_k und y_k gemäß (9) für $k=$ 0 , 1 , 2 , ... getrennt aufzustellen:

$$(10)\qquad x_{k+1}=F_1(x_k,y_k)\underset{def.}{=}x_k-\frac{2y_k\left((x_k-1)^2-y_k+1\right)+x_k^2+y_k^2-4}{4y_k(x_k-1)+2x_k}$$

$$(11)\qquad y_{k+1}=F_2(x_k,y_k)\underset{def.}{=}y_k-\frac{-2x_k\left((x_k-1)^2-y_k+1\right)+2(x_k-1)\left(x_k^2+y_k^2-4\right)}{4y_k(x_k-1)+2x_k}$$

Ein lauffähiges skript-file, welches bei einer vorgegebenen Genauigkeit lediglich die Anzahl der erforderlichen Iterationen mit dem Lösungspaar (x^*,y^*) ausweist, sieht etwa so aus:

```
 1   %Newtoniteration für ein Gleichungssystem mit 2 Unbekannten
 2-  X=0;Y=1;         %Startwerte xo und yo
 3-  Eps=1e-10;       %Gewünschte Genauigkeit
 4-  for k=0:100
 5-     x=X;         %x=xk im k-ten Schritt (x=xo im Anfangsschritt k=0)
 6-     y=Y;         %y=yk im k-ten Schritt (y=yo im Anfangsschritt k=0)
 7-     X=F1(x,y);   %X=xk+1 ist Nachfolger von x=xk
 8-     Y=F2(x,y);   %Y=yk+1 ist Nachfolger von y=yk
 9-     if max(abs(X-x),abs(Y-y))<Eps
10-         break    %Abbruch, wenn Genauigkeit erreicht
11-     end
12-  end
13-  fprintf('Es wurden %u Iterationen benötigt. Die Lösungen: \n',k)
14-  fprintf('x = %.9f  und  y = %.9f\n',X,Y)
```

Hier wird wahlweise eine **for**-Schleife verwendet mit maximal 100 Schleifendurchläufen. Bei Unterschreiten der Genauigkeit in Zeile 9 wird die **for**-Schleife verlassen und das Ergebnis ausgegeben.
In Zeile 7 und 8 wird auf die Iterationsvorschriften (10) und (11) verwiesen. Statt diese hier einzugeben, was durchaus möglich ist, definiert man sie in eigenen function-files mit Namen "F1" und "F2" (als m-files), damit das obige script-file die function-files mit Zeile 7 und 8 *aufrufen* kann:

```
1-  function X=F1(x,y)
2-  X=x-(2*y*((x-1)^2-y+1)+x^2+y^2-4)/(4*y*(x-1)+2*x);
```

sowie

```
1-  function Y=F2(x,y)
2-  Y=y-(-2*x*((x-1)^2-y+1)+2*(x-1)*(x^2+y^2-4))/(4*y*(x-1)+2*x);
```

Die Startwerte x_0 = **X=0** und y_0 = **Y=1** gemäß Zeile 2 im script-file liefern die Ausgabe

```
Es wurden 5 Iterationen benötigt. Die Lösungen:
x = -0.000000000  und  y = 2.000000000
```

Die Startwerte x_0 = **X=1** (im script-file in Zeile 2 **X=0** durch **X=1** ersetzen) und y_0 = **Y = 1** ergeben

```
Es wurden 6 Iterationen benötigt. Die Lösungen:
x = 1.533176835  und  y = 1.284277537
```

Das obige script-file kann selbst auch als function-file mit z.B. Namen "Newton" codiert werden:

```
 1   %Newtoniteration für ein Gleichungssystem mit 2 Unbekannten
 2-  function Newton(X,Y,F1,F2)
 3-  Eps=1e-10;
 4-  for k=0:100
 5-     x=X;
 6-     y=Y;
 7-     X=F1(x,y);
 8-     Y=F2(x,y);
 9-     if max(abs(X-x),abs(Y-y))<Eps
10-         break
11-     end
12-  end
13-  fprintf('Es wurden %u Iterationen benötigt. Die Lösungen: \n',k)
14-  fprintf('x = %.9f  und  y = %.9f\n',X,Y)
```

Zum Erhalt der Lösungen x und y für die Startwerte $(x_0, y_0) = (0, 1)$ sowie $(x_0, y_0) = (1, 1)$ ist nun im *Command Window* einzugeben:

`>> Newton(0,1,@F1,@F2)` mit Ergebnis `x = -0.000000000  und  y = 2.000000000`
`>> Newton(1,1,@F1,@F2)` mit Ergebnis `x =  1.533176835  und  y = 1.284277537` .

Die Angaben @F1 und @F2 verweisen auf die in den function-files F1 und F2 definierten Funktionen und werden als "function handles" bezeichnet. Sie stellen in MATLAB einen *eigenen Datentyp* dar und sind durch das der betreffenden Funktion vorangestellte @-Zeichen erkennbar. Dazu später mehr.

Soll eine *Liste* der Iterationswerte ausgegeben werden, so kann man auf das script-file zur Banachiteration in II.3.4 zurückgreifen. Man vergleiche dieses file mit dem folgenden script-file:

```
 1   %Newtoniteration für ein Gleichungssystem mit 2 Unbekannten
 2-  K=[];LX=[];LY=[];       %Matrizen für k,x und y initialisieren
 3-  X=1;Y=1;x=0;y=0;k=0;    %Startwerte X=xo,Y=yo,Vorgänger x=0,y=0
 4-  Eps=1e-10;              %Gewünschte Genauigkeit
 5-  while max(abs(X-x),abs(Y-y))>=Eps
 6-      x=X;y=Y;
 7-      K=[K,k];LX=[LX,x];LY=[LY,y];
 8-      X=F1(x,y);Y=F2(x,y);
 9-      k=k+1;
10-  end
11-  fprintf('Es wurden %u Iterationen benötigt. \n',k)
12-  fprintf('Die Lösungen lauten: \n')
13-  fprintf('x = %.9f und y = %.9f\n',X,Y)
14-  fprintf('Im folgenden die Liste der Iterationen \n')
15-  fprintf('für die x-Werte: \n')
16-  for i=1:length(K)       %length(K)=Anzahl der Listeneinträge.
17-       k=K(i);x=LX(i);    %k und xk erzeugen und
18-       if k<10            %in einer Liste ausgeben.
19-           fprintf('x( %u) = %.9f\n',k,x)
20-       else               %Für k<10 vor k ein Leerzeichen
21-           fprintf('x(%u) = %.9f\n',k,x)
22-       end                %einfügen, um unschöne Einrückungen
23-  end                     %in der Liste zu vermeiden.
24-  fprintf('und die y-Werte: \n')
25-  for i=1:length(K)       %Kommentar wie oben.
26-      k=K(i);y=LY(i);     %k und yk erzeugen.
27-      if k<10
28-          fprintf('y( %u) = %.9f\n',k,y)
29-      else
30-          fprintf('y(%u) = %.9f\n',k,y)
31-      end
32-  end
```

```
Es wurden 7 Iterationen benötigt.
Die Lösungen lauten:
x = 1.533176835 und y = 1.284277537
Im folgenden die Liste der Iterationen
für die x-Werte:
x( 0) = 1.000000000
x( 1) = 2.000000000
x( 2) = 1.625000000
x( 3) = 1.537990196
x( 4) = 1.533191986
x( 5) = 1.533176835
x( 6) = 1.533176835
und die y-Werte:
y( 0) = 1.000000000
y( 1) = 1.000000000
y( 2) = 1.250000000
y( 3) = 1.281862745
y( 4) = 1.284270671
y( 5) = 1.284277537
y( 6) = 1.284277537
```

Für die Startwerte $(x_0 , y_0) = (1 , 1)$ ist die Ausgabe die Liste linkerhand. Statt einer **`for`**-Schleife wird hier wahlweise eine **`while`**-Schleife verwendet.
Auch dieses script-file greift gemäß Zeile 8 auf die beiden function-files F1 und F2 zu, um die Iterationen gemäß (10) und (11) auszuführen.
Codiert man das obige script-file als *function-file* mit den Eingabe-Daten **`X,Y,F1,F2`** , so muss in Zeile 3 die Zuweisung **`X=1;Y=1;`** entfallen und zu Beginn des files z.B. **`function Newton2(X,Y,F1,F2)`** codiert werden. Der Aufruf erfolgt dann mit **`Newton(1,1,@F1,@F2)`** im Command Window, um die Liste linkerhand zu erhalten.

II.5 2-dimensionale Banach-Iteration und Newton-Raphson-Iteration

Zu lösen ist das Gleichungssystem

(12) $f_1(x,y)=\frac{1}{2}\sin(xy)-x=0$ und $f_2(x,y)=2y-e^{-x}-4=0$.

Statt des Newtonverfahrens wollen wir (12) zunächst mit einer *Banach-Iteration* lösen. Dazu sind die beiden Gleichungen lediglich in die *Fixpunktform* zu überführen:

(13) $x=F_1(x,y)=\frac{1}{2}\sin(xy)$ und $y=F_2(x,y)=2+\frac{1}{2}e^{-x}$.

Zur Ausführung mit MATLAB kann das obige function-file **`Newton`** (hier umbenannt in **`Banach`**) herangezogen werden

```
 1   %Banachiteration für ein Gleichungssystem mit 2 Unbekannten
 2-  function Banach(X,Y,F1,F2)
 3-  Eps=1e-10;
 4-  for k=0:100
 5-     x=X;
 6-     y=Y;
 7-     X=F1(x,y);
 8-     Y=F2(x,y);
 9-     if max(abs(X-x),abs(Y-y))<Eps
10-         break
11-     end
12-  end
13-  fprintf('Es wurden %u Iterationen benötigt. Die Lösungen: \n',k)
14-  fprintf('x = %.9f  und  y = %.9f\n',X,Y)
```

mit

```
1-  function X=F1(x,y)
2-  X=sin(x*y)/2;
```

sowie

```
1-  function Y=F2(x,y)
2-  Y=2+exp(-x)/2;
```

Dabei lässt mit den Startwerten **`x=0.3`** und **`y=2`** die Konvergenzgeschwindigkeit zu wünschen übrig:

```
>> Banach(0.3,2.0,@F1,@F2)
Es wurden 45 Iterationen benötigt. Die Lösungen:
x = 0.405747526  und  y = 2.333239207
```

Zur Steigerung der Konvergenzgeschwindigkeit (wesentlich weniger als 45 Iterationen!) verwenden wir das 2-dimensionale Newtonverfahren allerdings mit einer in der Praxis üblichen Modifikation:
Gemäß (6) gilt die Iterationsvorschrift

(13) $$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x}(x_k,y_k) & \frac{\partial f_1}{\partial y}(x_k,y_k) \\ \frac{\partial f_2}{\partial x}(x_k,y_k) & \frac{\partial f_2}{\partial y}(x_k,y_k) \end{pmatrix}^{-1}}_{\text{Jacobi-Matrix } \mathbf{J}(x_k\ ,\ y_k)} \begin{pmatrix} f_1(x_k,y_k) \\ f_2(x_k,y_k) \end{pmatrix} \quad , \quad k = 0\ ,\ 1\ ,\ 2\ ,\ \ldots$$

Wir schreiben $\mathsf{x}_k = \begin{pmatrix} x_k \\ y_k \end{pmatrix}$, $\mathsf{x}_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix}$, $\mathsf{f}(\mathsf{x}_k) = \begin{pmatrix} f_1(x_k,y_k) \\ f_2(x_k,y_k) \end{pmatrix}$ und $\mathsf{J}(\mathsf{x}_k)$ für die Jacobi-Matrix. Dann lautet (13) kurz: $\mathsf{x}_{k+1} = \mathsf{x}_k - \mathsf{J}(\mathsf{x}_k)^{-1}\,\mathsf{f}(\mathsf{x}_k)$.

Daraus folgt $\mathsf{x}_{k+1} - \mathsf{x}_k = -\,\mathsf{J}(\mathsf{x}_k)^{-1}\,\mathsf{f}(\mathsf{x}_k)$ oder mittels Multiplikation mit $\mathsf{J}(\mathsf{x}_k)$:

$\mathsf{J}(\mathsf{x}_k)\,(\mathsf{x}_{k+1} - \mathsf{x}_k) = -\,\mathsf{f}(\mathsf{x}_k)$. Mit der "Korrekturgröße" $\mathsf{h}_k = \mathsf{x}_{k+1} - \mathsf{x}_k$

erhalten wir die Vorgehensweise des ***Newton-Raphson-Verfahrens*** :

(14) $\begin{cases} \text{Definiere für } k=0 \text{ den Startvektor } \mathsf{x}_0 \\ \text{Löse das } \textit{lineare Gleichungssystem } \mathsf{J}(\mathsf{x}_k)\,\mathsf{h}_k = -\,\mathsf{f}(\mathsf{x}_k) \text{ für } k = 0, 1, 2, \ldots \text{ nach } \mathsf{h}_k \text{ auf.} \\ \text{Setze } \mathsf{x}_{k+1} = \mathsf{x}_k + \mathsf{h}_k \end{cases}$

Im konkreten Fall ist die Gleichung

(15) $$\underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x}(x_k,y_k) & \frac{\partial f_1}{\partial y}(x_k,y_k) \\ \frac{\partial f_2}{\partial x}(x_k,y_k) & \frac{\partial f_2}{\partial y}(x_k,y_k) \end{pmatrix}}_{\mathbf{J}(\mathbf{x}_k)} \underbrace{\begin{pmatrix} h_{1k} \\ h_{2k} \end{pmatrix}}_{\mathbf{h}_k} = -\underbrace{\begin{pmatrix} f_1(x_k,y_k) \\ f_2(x_k,y_k) \end{pmatrix}}_{\mathbf{b} \underset{def.}{=} \mathbf{f}(\mathbf{x}_k)}$$

nach dem Korrekturvektor h_k aufzulösen. Das geht, wenn **det** $\mathsf{J}(\mathsf{x}_k) \neq \mathbf{0}$ ist:

(16) $$\begin{pmatrix} h_{1k} \\ h_{2k} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x}(x_k,y_k) & \frac{\partial f_1}{\partial y}(x_k,y_k) \\ \frac{\partial f_2}{\partial x}(x_k,y_k) & \frac{\partial f_2}{\partial y}(x_k,y_k) \end{pmatrix}^{-1}}_{\mathbf{J}} \underbrace{\begin{pmatrix} -f_1(x_k,y_k) \\ -f_2(x_k,y_k) \end{pmatrix}}_{\mathbf{b}},$$

Hieraus erhalten wir dann $x_{k+1} = x_k + h_{1k}$ und $y_{k+1} = y_k + h_{2k}$.

Gemäß (12) ergeben sich die Ableitungen in der Jacobi-Matrx J im konkreten Beispiel zu

$$f_{11}(x,y) \underset{def.}{=} \frac{\partial f_1}{\partial x}(x,y) = \frac{1}{2}y\cos(xy) - 1 \quad , \quad f_{12}(x,y) \underset{def.}{=} \frac{\partial f_1}{\partial y}(x,y) = \frac{1}{2}x\cos(xy)$$

$$f_{21}(x,y) \underset{def.}{=} \frac{\partial f_2}{\partial x}(x,y) = e^{-x} \quad , \quad f_{22}(x,y) \underset{def.}{=} \frac{\partial f_2}{\partial y}(x,y) = 2$$

und in $\mathbf{b}$ ist gemäß (12)

$f_1(x,y) = \frac{1}{2}\sin(xy) - x$ und $f_2(x,y) = 2y - e^{-x} - 4$ einzutragen.

Das folgende script-file löst (12) mit dem Newton-Raphson-Verfahren:

```
 1   %Newton-Raphson-Iteration für 2 Gleichungen mit 2 Unbekannten
 2-  X=0.3;Y=2;     %Startwerte xo und yo.
 3-  Eps=1e-10;     %Gewünschte Genauigkeit.
 4-  for k=0:100    %Im k-ten Schritt
 5-     x=X;y=Y;  %Vorgängerwerte mit xk,yk belegen.
 6-     J=[f11(x,y) f12(x,y);f21(x,y) f22(x,y)];
 7-     b=[-f1(x,y);-f2(x,y)];
 8-     h=J\b      %Multiplikation der Inversen von J mit b ergibt h.
 9-     X=X+h(1); %xk+1 berechnen.
10-     Y=Y+h(2); %yk+1 berechnen.
11-     if max(abs(X-x),abs(Y-y))<Eps
12-         break
13-     end
14-  end
15-  fprintf('Es wurden %u Iterationen benötigt. Die Lösungen: \n',k)
16-  fprintf('x = %.9f  und  y = %.9f\n',X,Y)
```

Die Ableitungen in der Matrix J sowie f_1 und f_2 in $\mathbf{b}$ werden von 6 weiteren function-files an das script-file übergeben:

```
function J11=f11(x,y)
J11=y*cos(x*y)/2-1;

function J12=f12(x,y)
J12=x*cos(x*y)/2;

function J21=f21(x,y)
J21=exp(-x);

function J22=f22(x,y)
J22=2;

function b1=f1(x,y)
b1=sin(x*y)/2-x;

function b2=f2(x,y)
b2=2*y-exp(-x)-4;
```

Wird das script-file selbst als function-file codiert etwa mit Namen **Newton_Raphson**, so entfallen in Zeile 2 die Zuweisungen **X=0.3** und **Y=2** . Stattdessen ist dort
function Newton_Raphson(X,Y,f1,f2,f11,f12,f21,f22) zu codieren.
Mit den Komponenten von **J** und **b** als function handles werden auch die Startwerte X und Y an das file **Newton_Raphson** übergeben. Im Command Window ist dazu einzugeben:

```
>> NR(0.3,2,@f1,@f2,@f11,@f12,@f21,@f22)
```

Zur Eingabe der Daten von **J** und **b** gibt es in MATLAB noch eine weitere Möglichkeit:
Man codiert *nur* das obige *script-file* zum Newton-Raphson-Verfahren und *verzichtet* auf die Codierung der 6 function-files zum Erhalt von `f1,f2,f11,f12,f21` und `f22`. Stattdessen gebe man diese in das *Command Window* wie folgt ein:

```
>> f1=@(x,y)sin(x*y)/2-x;
>> f2=@(x,y)2*y-exp(-x)-4;
>> f11=@(x,y)y*cos(x*y)/2-1;
>> f12=@(x,y)x*cos(x*y)/2;
>> f21=@(x,y)exp(-x);
>> f22=@(x,y)2;
```

Jede Zeile ist mit Semikolon abzuschließen und dann mit RETURN zu bestätigen.
Nach Bestätigung der letzten Zeile ist das *script-file* zu starten.
Jede der obigen 6 Zeilen definiert eine *anonymous function mit function handle* .
Allgemeine Syntax: **f = @ (Variable1, Variable2, ...)Formel ;**

Bei dieser Art der Eingabe ist das Haupt-Programm als script-file zu codieren.
Will man bei einem konkreten Gleichungssystem mehrere Anfangswerte für **X** und **Y** testen, so ist es sinnvoller, die obigen 6 function-files zu codieren und das Haupt-Programm als script-file zu starten.
In Zeile 2 sind lediglich die Anfangswerte für **X** und **Y** jeweils zu variieren.

In allen Fällen erhält man nach erfolgreichem Programmlauf die Ausgabe

```
Es wurden 4 Iterationen benötigt. Die Lösungen:
x = 0.405747526  und  y = 2.333239207
```

Statt 45 Iterationen wie bei der Banach-Iteration sind also hier nur 4 Iterationen erforderlich.
Wünscht man eine Listenausgabe mit den Näherungswerten nach jedem Iterationsschritt, so kann man auf den entsprechenden Programmcode im Abschnitt II.4 zurückgreifen (Programmzeilen 14-32).

II.6 Graphische Darstellung mit `ezplot`

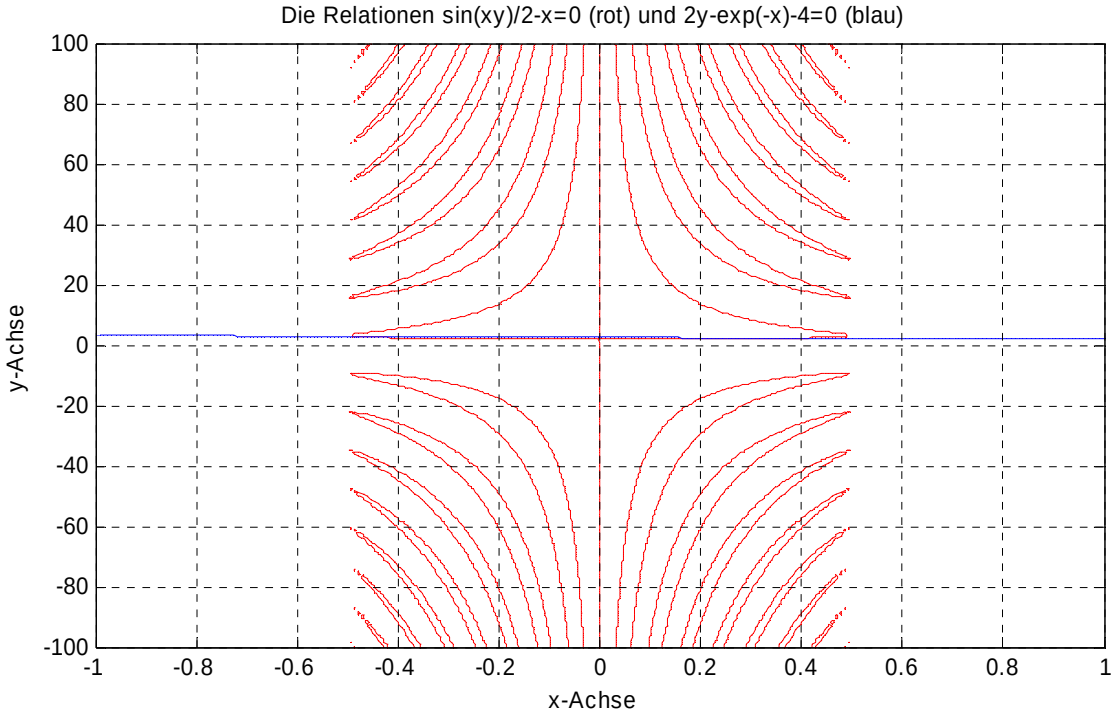
Es stellt sich die Frage, ob das Gleichungssystem (12) noch weitere Lösungen hat. Zur Klärung dieser Frage ist es sinnvoll, die beiden Gleichungen in (12) als Relationen graphisch darzustellen. MATLAB bietet dazu den Befehl **ezplot** an mit der grundlegenden Syntax: **ezplot('Relation',[a,b,c,d])**.
Weiterführende Infos hierzu findet man in der MATLAB-Hilfe unter dem Suchbegriff "ezplot".

Als Relation gebe man in Hochkommata $\frac{1}{2}\sin(xy)-x=0$ sowie $2y-e^{-x}-4=0$ ein.

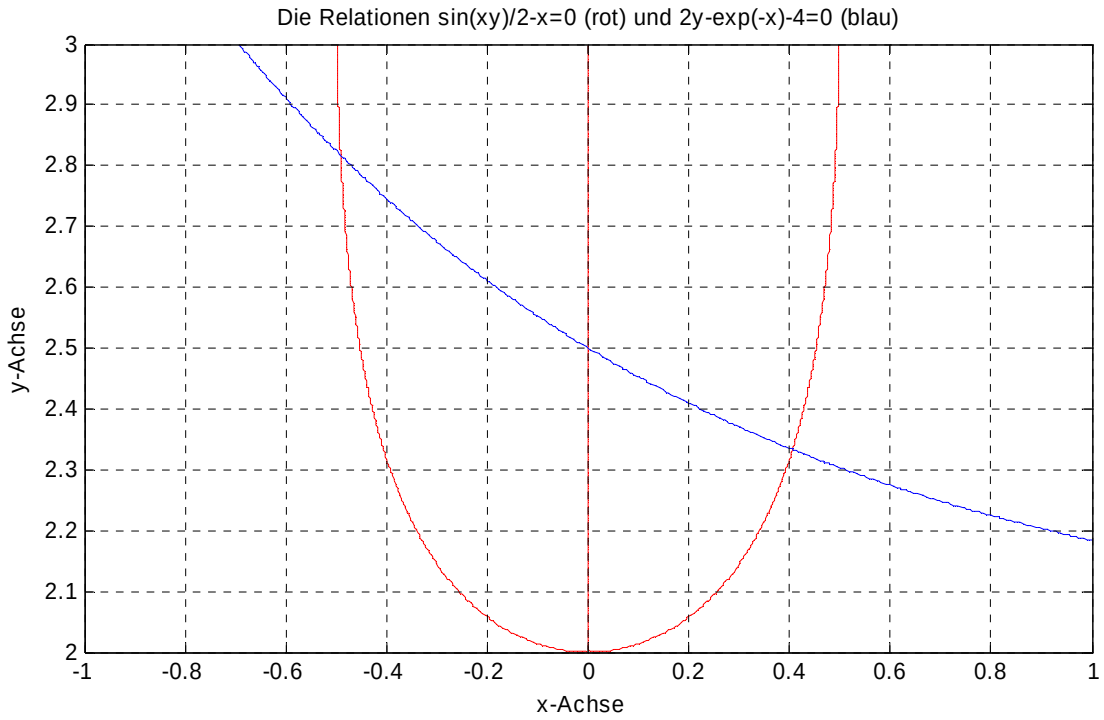
a und **b** bezeichnen die Intervallgrenzen für x und **c** und **d** die Intervallgrenzen für y .
Die Graphiken beider Relationen erhält man in MATLAB z.B. mit dem script-file

```
p1=ezplot('sin(x.*y)/2-x=0',[-1,1,-100,100]);set(p1,'Color','red');
hold on
p2=ezplot('2*y-exp(-x)-4=0',[-1,1,-100,100]);set(p2,'Color','blue');
title('Die Relationen sin(xy)/2-x=0 (rot) und 2y-exp(-x)=0 (blau)');
xlabel('x-Achse');ylabel('y-Achse');grid on
hold off
```

Beide Graphiken werden im Bereich $-1 \le x \le 1$, $-100 \le y \le 100$ in einem Koordinatensystem erzeugt.
Der Befehl **`grid on`** versieht das Koordinantensystem mit einem Raster.
Der Befehl **`set(p1,'Color','red')`** bzw. **`set(p2,'Color','blue')`** ordnet *beiden* Graphiken die Linienfarbe rot bzw. blau zu. Achsenbezeichnungen und Titel werden wie üblich erzeugt.
Das obige script-file liefert folgendes Bild:



Durch Änderung des Bereichs $-1 \le x \le 1$, $-100 \le y \le 100$ in $-1 \le x \le 1$, $2 \le y \le 3$ ergibt sich:



Bei beiden Graphiken (in der unteren genauer) erkennt man, dass es nur *zwei* Lösungspaare (x , y) gibt:
Neben dem Paar `x = 0.405747526 und y = 2.333239207`
gibt es das Paar `x = -0.491316269 und y = 2.817233101` ,
welches mit den Startwerten `X=0.5` und `Y=3` nach 4 Iterationen erreicht wird.

Schlussbemerkung:

Wir haben nur 2 Gleichungen mit 2 Unbekannten betrachtet und darauf die Banach- und Newton-Iteration angewandt. Auf Konvergenzaussagen wie in II.3.1–3 haben wir dabei übrigens verzichtet.
Sowohl die 2-dimensionale Banach-Iteration als auch die 2-dimensionale Newton-Iteration lassen sich *wörtlich* auf den *n*-dimensionalen Fall übertragen.
Für das Gleichungssystem aus *n* Gleichungen mit *n* Unbekannten

$$f\left(x_1,...,x_n\right) \underset{def.}{=} \begin{pmatrix} f_1(x_1,...,x_n) \\ \vdots \\ f_n(x_1,...,x_n) \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$ lautet z.B. die *Newton-Raphson-Iteration* entsprechend (13)

$$(17)\quad \begin{pmatrix} x_{1,k+1} \\ \vdots \\ x_{n,k+1} \end{pmatrix} = F(x_{1,k},...,x_{n,k}) \underset{def.}{=} \begin{pmatrix} x_{1,k} \\ \vdots \\ x_{n,k} \end{pmatrix} - \underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1,...,x_n) & \cdots & \frac{\partial f_1}{\partial x_n}(x_1,...,x_n) \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(x_1,...,x_n) & \cdots & \frac{\partial f_n}{\partial x_n}(x_1,...,x_n) \end{pmatrix}^{-1}}_{\text{Jacobi-Matrix } \mathbf{J}} \begin{pmatrix} f_1(x_1,...,x_n) \\ \vdots \\ f_n(x_1,...,x_n) \end{pmatrix}$$

oder entsprechend (16)

$$(18)\quad \begin{pmatrix} h_{1k} \\ \vdots \\ h_{nk} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1,...,x_n) & \cdots & \frac{\partial f_1}{\partial x_n}(x_1,...,x_n) \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(x_1,...,x_n) & \cdots & \frac{\partial f_n}{\partial x_n}(x_1,...,x_n) \end{pmatrix}^{-1}}_{\mathbf{J}} \underbrace{\begin{pmatrix} -f_1(x_1,...,x_n) \\ \vdots \\ -f_n(x_1,...,x_n) \end{pmatrix}}_{\mathbf{b}}, \quad k = 0, 1, 2, ...$$

Die Konvergenzaussage in Satz II.3.6 erfordert neben $f(x^*) = 0$ die Bedingung $f'(x^*) \neq 0$.
Diese Bedingungen sind hier durch $f(x_1^*,...,x_n^*) = f(\mathbf{x}^*) = 0$ und $\det J(x_1^*,...,x_n^*) = \det \mathbf{J}(\mathbf{x}^*) \neq 0$
zu ersetzen. Liegt der Startvektor x_0 "nahe genug" bei $\mathbf{x}^*$, so konvergiert (17) gegen $\mathbf{x}^*$.
Die Konvergenz ist auch hier quadratisch. Das *n*-dimensionale Banach-Verfahren konvergiert linear.

Die Auffindung eines geeigneten Startvektors x_0 ist vorallem im höherdimensionalen Fall nicht immer leicht. Oft "probiert man", um schnell zum Erfolg zu kommen. Gelegentlich versucht man auch, mit einer Banach-Iteration einen Startvektor soweit zu "verbessern", um dann den "besseren" Wert mit der Newton-Iteration zur gesuchten Lösung zu iterieren.
Dies ist aber einem fundierten fortgeschrittenen Mathematikstudium vorbehalten und kann hier nicht weiter vertieft werden .

III. Fourierreihen

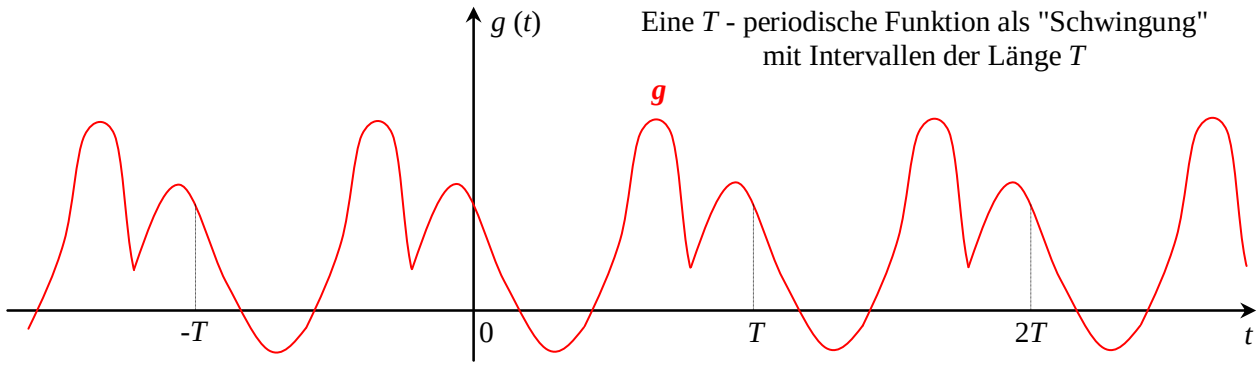
In Physik und Technik (besonders in der Elektrotechnik) sind periodische Vorgänge wie z.B. elektrische *Schwingungen* von besonderer Bedeutung. Deren mathematische Beschreibung erfolgt in der Regel mit *periodischen* Funktionen unter denen die *Sinus*- und *Cosinus*-Funktionen eine fundamentale Rolle spielen. Daher ist die Darstellung periodischer Funktionen mittels Sinus und Cosinus eine mathematische Grundaufgabe. Reihen solcher Art heißen *Fourierreihen* (Jean Baptiste Joseph Fourier, 1768 – 1830). Die Approximation periodischer Funktionen mittels Fourierreihen (unter Verwendung von MATLAB) sei Gegenstand dieses Kapitels. Zunächst einige wichtige theoretische Grundlagen.

III.1 Periodische Funktionen

Eine *auf ganz* $\mathbb{R}$ definierte Funktion g, welche bei konstantem Wert $T > 0$ der Bedingung

(1) $\quad g(t+T) = g(t) \quad$ für alle $\quad t \in \mathbb{R}$

genügt, heißt *periodisch*, genauer *T - periodisch*. T bezeichnet die *Periode* von g.
Teilt man die gesamte reelle *t-Achse* in Intervalle der Länge T ein, so ist der Graph von f auf allen diesen Intervallen derselbe.



Der *Sinus* und der *Cosinus* sind 2π- *periodisch* und die Funktionen $\sin kx$, $\cos kx$, $k = 1, 2, 3, \ldots$ haben die Periode $2\pi / k$. Infolgedessen sind sie ebenfalls 2π-periodisch.

Jede T-periodische Funktion g kann mittels der Substitution $t = x\dfrac{T}{2\pi}$ in eine 2π-periodische Funktion f umgewandelt werden. Definieren wir nämlich $f(x) \underset{def.}{=} g\left(x\dfrac{T}{2\pi}\right)$ so folgt

$$f(x+2\pi) = g\left(\left(x+2\pi\right)\frac{T}{2\pi}\right) = g\left(x\frac{T}{2\pi}+T\right) \underset{g\ T\text{-periodisch}}{=} g\left(x\frac{T}{2\pi}\right) = f(x)$$ und f ist somit 2π-periodisch.

Daher beschränken wir uns im folgenden *ohne Verlust der Allgemeinheit* auf **2π-periodische** Funktionen $f(x)$.

Interpretieren wir t als *Zeit* und $\omega \underset{def.}{=} \dfrac{2\pi}{T}$ als *Kreisfrequenz* mit der Zeitperiode T einer *Schwingung*, so ist $x = \omega t$ und wir schreiben daher gelegentlich auch $f(x) = f(\omega t)$. Der Funktionswert $f(\omega t)$ ist dann als *Amplitude* der Schwingung aufzufassen.

III.2 Fourierreihen und Fourierkoeffizienten

Es sei $f: \mathbb{R} \to \mathbb{R}$ eine beliebige reellwertige 2π-periodische Funktion.
Wir stellen uns die Aufgabe, sie durch eine Reihe der folgenden Form darzustellen:

(2) $$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty}\left(a_k \cos kx + b_k \sin kx\right)$$

Diese Reihe heißt *Fourierreihe* oder auch *trigonometrische Reihe*.
Falls f so darstellbar ist, welche Werte haben dann die Koeffizienten a_0, a_k, b_k für $k = 1, 2, 3, \ldots$?
Zur Bestimmung dieser sog. *Fourierkoeffizienten* gehen wir folgendermaßen vor:

1. Schritt: ***Berechnung der Fourierkoeffizienten b_k***

Es sei $k = 1, 2, 3, \ldots$ und $m = 1, 2, 3, \ldots$ angenommen!
Die Gleichung (2) wird beidseitig mit **sin *mx*** *multipliziert* und anschließend über das Periodenintervall $[0, 2\pi]$ *integriert*. Es folgt

$$f(x)\sin mx = \frac{a_0}{2}\sin mx + \sum_{k=1}^{\infty}\left(a_k \cos kx \sin mx + b_k \sin kx \sin mx\right)$$

$$\Rightarrow \int_0^{2\pi} f(x)\sin mx\ dx = \int_0^{2\pi} \frac{a_0}{2}\sin mx\ dx + \int_0^{2\pi}\sum_{k=1}^{\infty}\left(a_k \cos kx \sin mx + b_k \sin kx \sin mx\right) dx$$

mithin bei Vertauschung von Summe und Integration zunächst

$$(3)\qquad \int_0^{2\pi} f(x)\sin mx\ dx = \frac{a_0}{2}\int_0^{2\pi}\sin mx\ dx + \sum_{k=1}^{\infty}\left(a_k \int_0^{2\pi}\cos kx \sin mx\ dx + b_k \int_0^{2\pi}\sin kx \sin mx\ dx\right).$$

Mittels der *Additionstheoreme* für den Sinus und den Cosinus lassen sich die Integranden $\cos kx \sin mx$ sowie $\sin kx \sin mx$ umformen. Es ist nämlich

$$\sin(m-k)x + \sin(m+k)x = \sin(mx-kx) + \sin(mx+kx)$$
$$= \underbrace{(\sin mx\cos kx - \sin kx\cos mx)}_{\sin(mx-kx)} + \underbrace{(\sin mx\cos kx + \sin kx\cos mx)}_{\sin(mx+kx)} = 2\cos kx \sin mx$$

mithin

$$(4)\qquad \cos kx \sin mx = \frac{1}{2}\left[\sin(m-k)x + \sin(m+k)x\right]$$

sowie

$$\cos(m-k)x - \cos(m+k)x = \cos(mx-kx) - \cos(mx+kx)$$
$$= \underbrace{(\cos mx\cos kx + \sin mx\sin kx)}_{\cos(mx-kx)} - \underbrace{(\cos mx\cos kx - \sin mx\sin kx)}_{\cos(mx+kx)} = 2\sin kx \sin mx$$

mithin

$$(5)\qquad \sin kx \sin mx = \frac{1}{2}\left[\cos(m-k)x - \cos(m+k)x\right].$$

Wegen

$$\int_0^{2\pi}\sin mx\ dx = \left[-\frac{1}{m}\cos mx\right]_0^{2\pi} = -\frac{1}{m}\left[\cos 2m\pi - \cos 0\right] = -\frac{1}{m}\left[1-1\right] = 0$$

$$\int_0^{2\pi}\cos mx\ dx = \left[\frac{1}{m}\sin mx\right]_0^{2\pi} = \frac{1}{m}\left[\sin 2m\pi - \sin 0\right] = \frac{1}{m}\left[0-0\right] = 0$$

für alle *m* ist somit in (3) auch

$$(6.1)\qquad \int_0^{2\pi}\cos kx \sin mx\ dx \underset{(4)}{=} \int_0^{2\pi}\frac{1}{2}\left[\sin(m-k)x + \sin(m+k)x\right] dx = 0 \quad \text{für } \boldsymbol{\text{alle } k, m}$$

$$(6.2)\qquad \int_0^{2\pi}\sin kx \sin mx\ dx \underset{(5)}{=} \int_0^{2\pi}\frac{1}{2}\left[\underbrace{\cos(m-k)x}_{=1 \text{ für } k=m} - \underbrace{\cos(m+k)x}_{=\cos 2mx \text{ für } k=m}\right] dx = 0 \quad \text{für } \boldsymbol{k \neq m}\ .$$

In (3) fallen rechts des Gleichheitszeichens daher alle Integrale bis auf jenes hinter b_k weg und dort bleiben nur die Integrale mit $\boldsymbol{k = m}$ übrig und das erzwingt zunächst für ***alle*** $\boldsymbol{m}$ = 1 , 2 , 3 , ...

$$\int_0^{2\pi} f(x)\sin mx\ dx = b_m \int_0^{2\pi} \sin^2 mx\ dx \underset{(6.2)}{=} b_m \int_0^{2\pi} \frac{1}{2}\left(1-\cos 2mx\right)\ dx = \frac{1}{2} b_m \cdot 2\pi = b_m\ \pi\ .$$

Indem wir k statt m schreiben, ergibt sich schließlich

(7) $$\boxed{b_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\sin kx\ dx \quad \text{für } k = 1\ ,\ 2\ ,\ 3\ ,\ ...}$$

2. Schritt: ***Berechnung der Fourierkoeffizienten*** $\boldsymbol{a_k}$

Es sei $k = 1\ ,\ 2\ ,\ 3\ ,\ ...$ und $m = 0\ ,\ 1\ ,\ 2\ ,\ 3\ ,\ ...$ angenommen!

Für m sei also *zusätzlich* der Wert 0 zugelassen! Die Gleichung (2) wird beidseitig mit $\boldsymbol{\cos mx}$ *multipliziert* und anschließend über das Periodenintervall $[0\ ,\ 2\pi]$ *integriert*. Analog zu (3) ergibt sich

(8) $$\int_0^{2\pi} f(x)\cos mx\ dx = \frac{a_0}{2}\int_0^{2\pi} \cos mx\ dx + \sum_{k=1}^{\infty}\left(a_k \int_0^{2\pi} \cos kx\cos mx\ dx + b_k \int_0^{2\pi} \sin kx\cos mx\ dx \right)$$

Wegen (4) sind alle Integrale hinter b_k gleich Null. Für $m \neq 0$ ist auch das Integral hinter $a_0/2$ gleich Null. Mittels des *Additionstheorems* für den Cosinus lässt sich der Integrand $\cos kx \cos mx$ umformen. Es ist nämlich

$$\cos(m-k)x + \cos(m+k)x = \cos(mx-kx) + \cos(mx+kx)$$

$$= \underbrace{\left(\cos mx\cos kx + \sin mx\sin kx\right)}_{\cos(mx-kx)} + \underbrace{\left(\cos mx\cos kx - \sin mx\sin kx\right)}_{\cos(mx+kx)} = 2\cos kx\cos mx$$

mithin $\cos kx\cos mx = \frac{1}{2}\left[\cos(m-k)x + \cos(m+k)x\right]$.

Das erzwingt

(9) $$\int_0^{2\pi} \cos kx\cos mx\ dx = \int_0^{2\pi} \frac{1}{2}\left[\underbrace{\cos(m-k)x}_{=\,1 \text{ für } k=m} + \underbrace{\cos(m+k)x}_{=\,\cos 2mx \text{ für } k=m} \right] dx = 0 \quad \text{für } \boldsymbol{k \neq m}\ .$$

In (8) fallen rechts des Gleichheitszeichens somit auch alle Integrale hinter a_k bis auf jenes mit $\boldsymbol{k = m}$ weg und das erzwingt zunächst für ***alle*** $\boldsymbol{m}$ = **1** , **2** , **3** , ...

(10) $$\int_0^{2\pi} f(x)\cos mx\ dx = a_m \int_0^{2\pi} \cos^2 mx\ dx \underset{(9)}{=} a_m \int_0^{2\pi} \frac{1}{2}\left(1+\cos 2mx\right)\ dx = \frac{1}{2} a_m \cdot 2\pi = a_m\ \pi$$

und speziell für $\boldsymbol{m = 0}$ wegen $\cos(0 \cdot x) = \cos 0 = 1$:

(11) $$\int_0^{2\pi} f(x)\ dx = \frac{a_0}{2}\int_0^{2\pi} dx = a_0\ \pi\ .$$

Indem wir auch hier k statt m schreiben, lassen sich (10) und (11) zusammenfassen zu

(12) $$\boxed{a_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\cos kx\ dx \quad \text{für } k = 0\ ,\ 1\ ,\ 2\ ,\ 3\ ,\ ...}\ .$$

Mit (7) und (12) kann für jede *integrierbare Funktion* $f : [0\ ,\ 2\pi] \to \mathbb{R}$ formal die Fourierreihe (2) gebildet werden. Da sie eine Summe *unendlich* vieler Summanden ist, stellt sich die Frage nach der *Konvergenz*. Eine für die Technik und Naturwissenschaft befriedigende Antwort lautet:

Satz über die Konvergenz von Fourierreihen:

Sei $f: \mathbb{R} \to \mathbb{R}$ eine 2π–periodische Funktion mit folgenden Eigenschaften:
(a) f ist *stetig differenzierbar* auf $[0\,,\,2\pi]$ bis auf endlich viele Ausnahmestellen $x_1\,,\,x_2\,,\,\ldots\,,\,x_n$ in $[0\,,\,2\pi]$.
(b) Es existieren an allen Ausnahmestellen die *links- und rechtsseitigen Grenzwerte* von f
$f(x_i-) = \lim\limits_{0<h\to 0} f(x_i-h)$ und $f(x_i+) = \lim\limits_{0<h\to 0} f(x_i+h)$, $i = 1\,,\,2\,,\,\ldots\,,\,n$
Die Ableitung f' ist *beschränkt*: $|f'(x)| \le M$ für eine positive Konstante M.

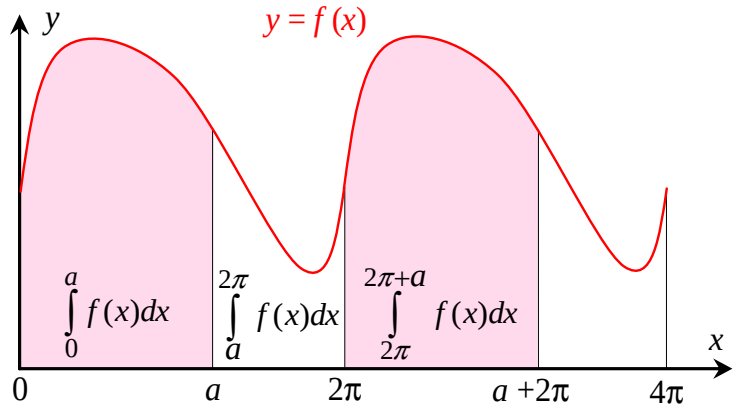
Dann konvergiert die rechte Seite von (2) gegen $f(x)$ für alle $x \in [0\,,\,2\pi]$ *ohne* die Ausnahmestellen, d.h. dort ist (2) als Gleichung erfüllt.
An den Ausnahmestellen konvergiert die rechte Seite von (2) gegen das *arithmetische Mittel* des links- und rechtsseitigen Grenzwertes von f. Ist also $x = x_i$ eine *Unstetigkeitsstelle* (*Sprungstelle*), so gilt statt (2)

$$(13)\qquad \frac{a_0}{2} + \sum_{k=1}^{\infty}\left(a_k \cos kx + b_k \sin kx\right) = \frac{1}{2}\left(f(x_i-) + f(x_i+)\right)$$

Eine Funktion, welche (a) und (b) erfüllt, heißt *stückweise glatt*.

Anmerkung:

Wir nehmen in unseren Überlegungen f als stückweise glatte **2π-periodische** Funktion an.
In (7) und (12) ändern sich die Fourierkoeffizienten nicht, wenn das Integrationsintervall $[0\,,\,2\pi]$ durch ein *beliebiges anderes* Integrationsintervall der Länge 2π ersetzt wird. Gemäß dem Bild linkerhand ist nämlich leicht ersichtlich:



$$\int_0^{2\pi} f(x)\,dx = \int_0^{a} f(x)\,dx + \int_a^{2\pi} f(x)\,dx$$
$$= \int_{2\pi}^{a+2\pi} f(x)\,dx + \int_a^{2\pi} f(x)\,dx$$
$$= \int_a^{2\pi+a} f(x)\,dx$$

So kann beispielsweise in (7) und (12) statt von 0 bis 2π auch von $-\pi$ bis π integriert werden.

Es gilt also auch:

$$(14)\qquad a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos kx\,dx \qquad \text{für } k = 0\,,\,1\,,\,2\,,\,3\,,\,\ldots$$

$$(15)\qquad b_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin kx\,dx \qquad \text{für } k = \quad 1\,,\,2\,,\,3\,,\,\ldots$$

Fourier-Analyse und Fourier-Synthese:

In der Elektrotechnik und Elektronik spielt die Funktion f die Rolle eines periodischen *Signals* mit der Kreisfrequenz $\omega = \frac{2\pi}{T}$. Dabei ist T die *Schwingungsdauer*. Mit $x = \omega t$ schreiben wir für das Signal dann gelegentlich auch $f(x) = f(\omega t)$. Die Darstellung von f als Fourierreihe bedeutet die *Zerlegung des periodischen Signals in eine Summe von Sinus- und Cosinusfunktionen*, d. h. das Signal wird in seine *Frequenzanteile* zerlegt: $f(\omega t) = \frac{a_0}{2} + \sum\limits_{k=1}^{\infty}\left(a_k \cos k\omega t + b_k \sin k\omega t\right)$

Wir sprechen dann von der *Fourieranalyse* des Signals. Umgekehrt beschreibt die *Fourier-Synthese* die Erzeugung beliebiger Signale aus reinen Sinus- und Cosinus-Funktionen.

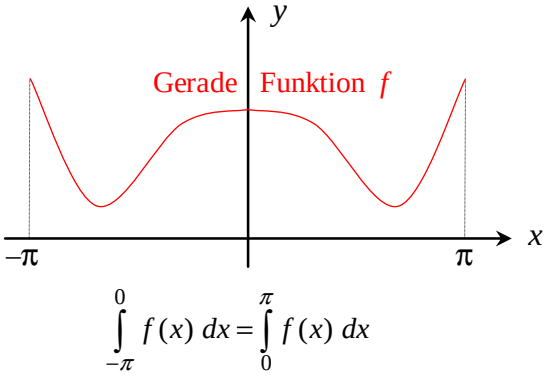
III.3 Beispiele von Fourierreihen mit MATLAB (Fourier-Analyse)

Im folgenden sollen unterschiedliche periodische *Signale* einer Fourieranalyse unterzogen werden. Hierbei soll MATLAB zum Zuge kommen. Zunächst noch ein wichtiger Begriff:

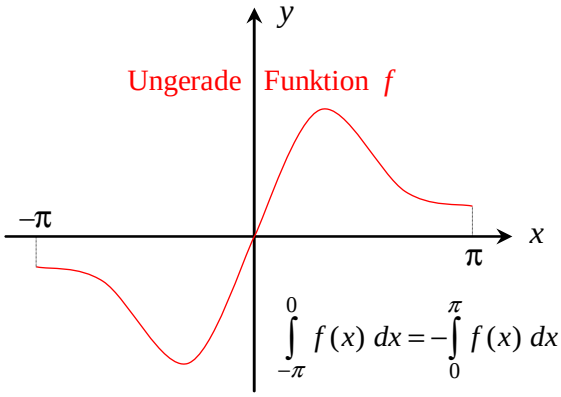
Definition:
Sei $I = [-r, r]$ oder $I =]-r, r[$ ein um den Nullpunkt *symmetrisches* Intervall.
Eine Funktion $f : I \to \mathbb{R}$ heißt *gerade*, falls $f(-x) = f(x)$ und *ungerade*, falls $f(-x) = -f(x)$ ist.
Folgerung: Im folgenden sei ohne Beschränkung der Allgemeinheit $r = \pi$.

Ist f *gerade*, so ist stets $\int_{-\pi}^{\pi} f(x)\,dx = 2\int_{0}^{\pi} f(x)\,dx$. Ist f *ungerade*, so ist stets $\int_{-\pi}^{\pi} f(x)\,dx = 0$.

Am einfachsten wird dies an den folgenden Bildern deutlich:



$$\int_{-\pi}^{0} f(x)\,dx = \int_{0}^{\pi} f(x)\,dx$$



$$\int_{-\pi}^{0} f(x)\,dx = -\int_{0}^{\pi} f(x)\,dx$$

Da der *Sinus ungerade* und der *Cosinus gerade* ist, so folgt:
Ist f *gerade*, so ist der Integrand in (14) *gerade* und der Integrand in (15) *ungerade*.
Ist f *ungerade*, so ist der Integrand in (14) *ungerade* und der Integrand in (15) *gerade*.
Daraus folgt:

Ist f *gerade*, so gilt

(16) $$a_k = \frac{2}{\pi}\int_{0}^{\pi} f(x)\cos kx\,dx \quad \text{und} \quad b_k = 0 \quad \text{für} \quad k = 0,1,2,3...$$

Ist f *ungerade*, so gilt

(17) $$a_k = 0 \quad \text{und} \quad b_k = \frac{2}{\pi}\int_{0}^{\pi} f(x)\sin kx\,dx \quad \text{für} \quad k = 1,2,3,\ldots.$$

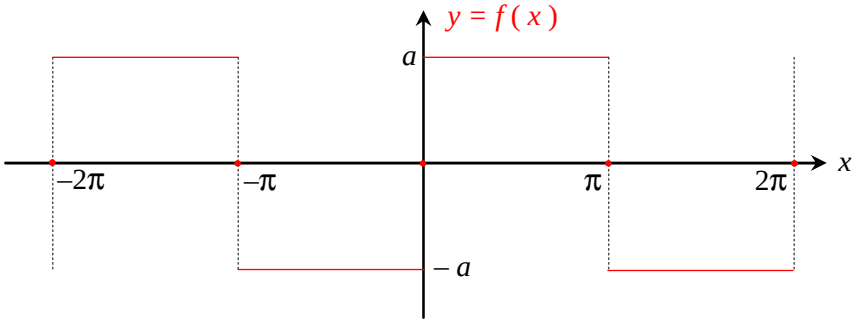
Die Fourierreihe einer *geraden* Funktion ist somit eine reine *Cosinus*-Reihe.
Die Fourierreihe einer *ungeraden* Funktion ist somit eine reine *Sinus*-Reihe.

Nun zu einigen Beispielen mit MATLAB:

Beispiel 1: ***Rechtecksignal***
Gegeben sei $a \neq 0$ und $h :]-\pi, \pi] \to \mathbb{R}$ definiert durch

$$h(x) = \begin{cases} a & \text{für } 0 < x < \pi \\ 0 & \text{für } x = 0,\ x = \pi \\ -a & \text{für } -\pi < x < 0 \end{cases}$$

h sei auf diese Weise zu einer 2π - periodischen Funktion f auf ganz $\mathbb{R}$ fortgesetzt. f ist eine *ungerade* stückweise glatte Funktion, ihre Fourierreihe besteht nur aus Sinusgliedern mit



$$b_k \underset{(17)}{=} \frac{2}{\pi}\int_{0}^{\pi} a\sin kx\,dx$$

$$= -\frac{2a}{\pi}\left[\frac{\cos(kx)}{k}\right]_{0}^{\pi}$$

$$= \begin{cases} 0 & \text{, wenn } k \text{ gerade} \\ \dfrac{4a}{k\pi} & \text{, wenn } k \text{ ungerade} \end{cases}$$

Somit erhalten wir die Fourierreihe des Rechtecksignals in der Form

$$f(x)=\sum_{k=1}^{\infty}\left(b_k\sin kx\right)=\sum_{\substack{k=1\\ k\text{ ungerade}}}^{\infty}\left(\frac{4a}{k\pi}\sin kx\right)=\frac{4a}{\pi}\sum_{k=0}^{\infty}\left(\frac{\sin(2k+1)x}{2k+1}\right)=\frac{4a}{\pi}\left(\frac{\sin x}{1}+\frac{\sin 3x}{3}+\frac{\sin 5x}{5}+\ldots\right)$$

Um zu sehen, wie die Fourierreihe das Rechtecksignal f *approximiert*, brechen wir die Reihe nach einem n-ten Summanden ab und stellen die Funktion

$$f_n(x)=\frac{4a}{\pi}\sum_{k=0}^{n}\left(\frac{\sin(2k+1)x}{2k+1}\right)=\frac{4a}{\pi}\left(\frac{\sin x}{1}+\frac{\sin 3x}{3}+\frac{\sin 5x}{5}+\ldots+\frac{\sin(2n+1)x}{2n+1}\right)$$

im Vergleich zusammen mit $f(x)$ dar.

Dazu schreiben wir in MATLAB ein function-file, welches f_n als Funktion von x zeichnet mit den Eingangsgrößen a und n. Die Rechteckfunktion f sowie f_n werden im Periodenintervall $[0\,,\,2\pi]$ skizziert:

```
 1-  %Fourierapproximation einer Rechteckfunktion
 2-  function Rechteckfunktion(a,n)
 3-  x=0:0.002:2*pi;    %Definitionsbereich [0,2*pi] festlegen
 4-  fn=zeros(size(x)); %fn vorinitialisieren
 5-  for k=0:n          %fn als Summe berechnen
 6-     fn=fn+sin((2*k+1)*x)/(2*k+1);
 7-  end
 8-  fn=fn*(4*a/pi);    %fn-Endsumme mit 4*a/pi multiplizieren
 9-  plot(x,fn)         %Fourierapproximation fn zeichnen
10-  hold on           %Zeichnung zusätzlich vorbereiten mit:
11-  x1=[0,pi];x2=[pi,2*pi];           %Rechteckfunktion f
12-  y1=[a,a];y2=[-a,-a];              %mit Funktionswert a und -a und
13-  xpi=[pi,pi];ypi=[-a,a];           %Senkrechte bei Sprungstelle x=pi
14-  xAchse1=[0,2*pi];xAchse2=[0,0];   %sowie der x-Achse.
15-  plot(x1,y1,'red-',x2,y2,'red-')   %f(x)=a und f(x)=-a zeichnen (rot)
16-  plot(xpi,ypi,'red-')              %Senkrechte bei x=pi zeichnen (rot)
17-  plot(xAchse1,xAchse2,'black-')    %x-Achse zeichnen (schwarz)
18-  text(0.5,3*a/4,'Gibbs-Phänomen'); %Text in Graphik einfügen
19-  title(['Fourierapproximation eines Rechtecksignals der Ordnung ',...
20-      int2str(n), ' beim Signalwert ',num2str(a)]) %Titel einfügen
21-  xlabel('x');ylabel('f(x) und fn(x)');    %Bezeichnung x- und y-Achse
22-  text(pi+2,0.1,'x-Achse');text(0.1,a+0.4,'y-Achse');
23-  legend('fn(x)','f(x)');legend('boxoff'); %Legende ohne Rahmen
24-  hold off
```

In Zeile 9 wird die n-te Fourierapproximation graphisch dargestellt. Der plot-Befehl erzeugt nur die unten dargestellten blauen "Schwingungen".

Die Darstellung der *Rechteckfunktion* im Periodenintervall $[0\,,\,2\pi]$ wird in den Zeilen 11 bis 13 vorbereitet: Man begreife `x1=[0,pi]` und `x2=[pi,2*pi]` im mathematischen Sinne als *Intervalle* $[0\,,\,\pi]$ bzw. $[\pi\,,\,2\pi]$ auf der x-Achse, wo die Rechteckfunktion den Wert a bzw. $-a$ annimmt. Diese Werte werden in MATLAB als "1-elementige Intervalle" `y1=[a,a]` bzw. `y2=[-a,-a]` codiert. `xpi=[pi,pi]` zusammen mit `ypi=[-a,a]` stellen an der Stelle π den "senkrechten Sprung" der Rechteckfunktion dar.

In Zeile 14 wird entsprechend die Darstellung der x-Achse von 0 bis 2π vorbereitet.

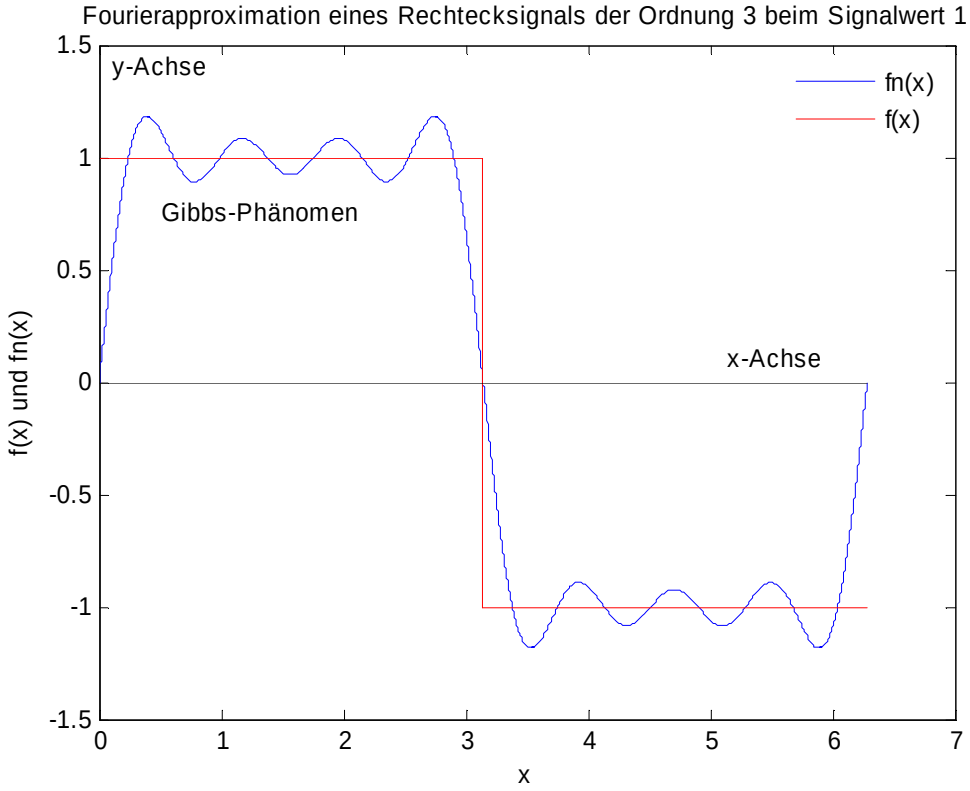
In den Zeilen 15 bis 17 wird nun die Darstellung von x-Achse (in schwarz) und Rechteckfunktion (in rot) *ausgeführt*. Der Strich hinter "red" und "black" erzeugt die hier gewünschte "durchgezogene Linie". Will man die Linie *punktiert* oder *gestrichelt*, so muss – durch `:` bzw – – ersetzt werden.

Mit der Option `text` in Zeile 18 kann an gewünschter Stelle in der Graphik ein Text beigefügt werden.

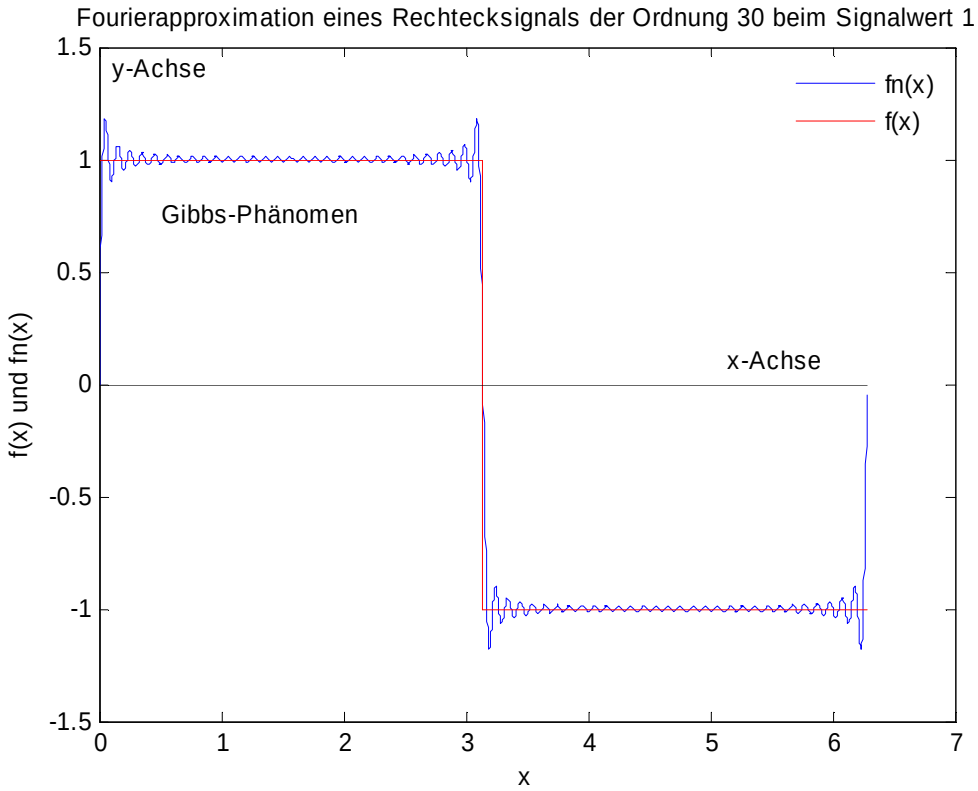
Die Option `title` versieht die Graphik mit einer gewünschten Überschrift ggfs. mit den Eingabewerten n und a als Parameter: n wird als ganze Zahl mit `int2str(n)` in die Überschrift eingefügt. "int" steht für "integer" und "str" für "String". n wird zur Darstellung im Text in das Stringformat konvertiert. Analoges gilt für die beliebig wählbare Zahl a : Sie wird mittels `num2str(a)` in das Stringformat konvertiert. Die Optionen in 21 bis 23 versehen die x- und y-Achse mit gewünschten Textbezeichnungen sowie die gesamte Graphik mit einer Legende mit oder ohne Umrahmung.

Die Befehle `hold on` und `hold off` in Zeile 10 und 24 bewirken, dass alle Befehle dazwischen im *gleichen* Fenster ausgeführt werden wie der plot-Befehl in Zeile 9.

Die folgende Graphik zeigt das Ergebnis des function-files mit den Eingabegrößen $a = 1$ und $n = 5$:



Erhöht man n auf 30, so ergibt sich folgendes Bild:



Die "Überschwinger" sichtbar an den Stellen $x = 0$, π und 2π bezeichnet man als "Gibbs'sches Phänomen" (Josiah Willard Gibb, 1839-1903). Es ist das typische Verhalten von Fourierreihen an Sprungstellen von f. Sie lassen sich auch dann nicht verringern, wenn man versucht, die Funktion durch weitere Summenglieder zu approximieren.

Die Approximation f_n hat beispielsweise an der Stelle $x = \pi$ den Wert 0. Das ist das arithmeitische Mittel der Werte + 1 und – 1, den rechts- und linksseitigen Grenzwerten von f an der Stelle π. So verhält es sich für *alle* n auch an allen anderen Sprungstellen, also insgesamt bei $0 , \pm\pi , \pm 2\pi, \pm 3\pi$ usw (siehe (13) im Satz über die Konvergenz von Fourierreihen).

Beispiel 2: ***Sägezahnsignal***

Gegeben sei $a > 0$ und $h :]-\pi , \pi] \to \mathbb{R}$ definiert durch

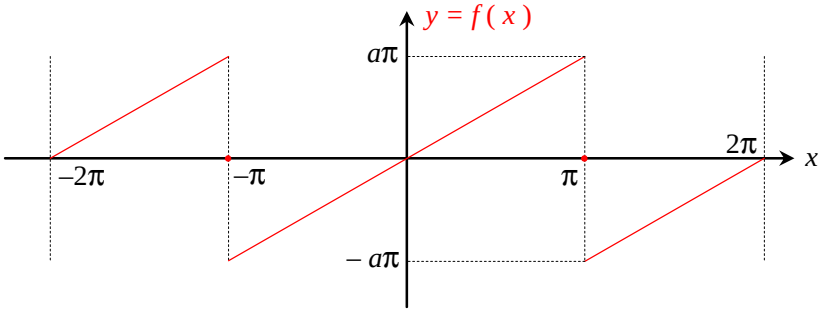
$$h(x) = \begin{cases} a\,x & \text{für } -\pi < x < \pi \\ 0 & \text{für } x = \pi \end{cases}$$

h sei auf diese Weise zu einer 2π - periodischen Funktion f auf ganz $\mathbb{R}$ fortgesetzt.

f ist eine *ungerade* stückweise glatte Funktion, ihre Fourierreihe besteht somit nur aus Sinusgliedern. Mittels Produktintegration lassen sich die Fourierkoeffizienten berechnen. Für $k = 1 , 2 , 3 , \ldots$ folgt

$$b_k \underset{(17)}{=} \frac{2}{\pi}\int_0^{\pi} ax\sin kx\,dx = \frac{2a}{\pi}\int_0^{\pi} x\sin kx\,dx = \frac{2a}{\pi}\left(\left[-x\frac{\cos(kx)}{k}\right]_0^{\pi} - \int_0^{\pi}\frac{-\cos(kx)}{k}\,dx\right)$$

$$= \frac{2a}{\pi}\left(\left[-\pi\frac{\cos(k\pi)}{k}\right] + \frac{1}{k}\underbrace{\left[\frac{1}{k}\sin kx\right]_0^{\pi}}_{=0}\right) = -\frac{2a}{k}\cos(k\pi) = -\frac{2a}{k}(-1)^k = \frac{2a(-1)^{k+1}}{k} \quad .$$



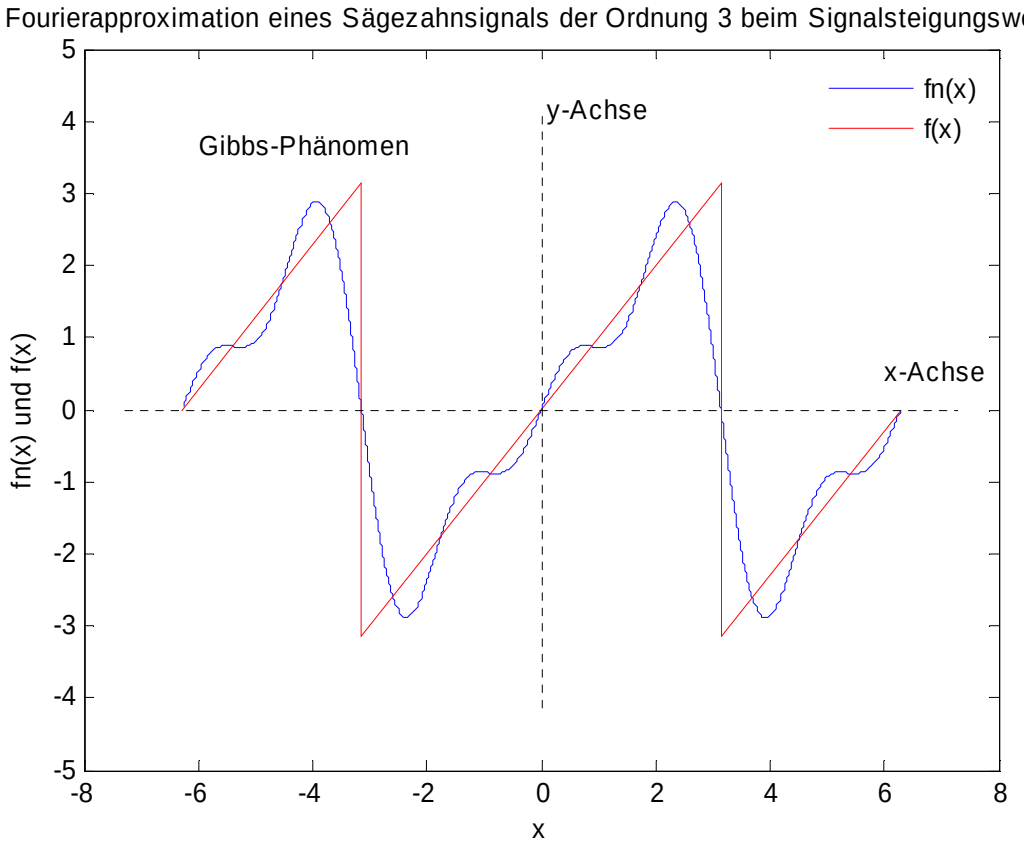
Die Graphik des Sägezahnsignals ist linkerhand dargestellt. Mit den soeben errechneten Fourierkoeffizienten b_k ergibt sich die Fourierreihe des Sägezahlsignals wie folgt:

(18) $$f(x) = \sum_{k=1}^{\infty}\left(b_k \sin kx\right) = \sum_{k=1}^{\infty}\left(\frac{2a(-1)^{k+1}}{k}\sin kx\right) = 2a\left(\frac{\sin x}{1} - \frac{\sin 2x}{2} + \frac{\sin 3x}{3} - \frac{\sin 4x}{4} + - \ \ldots\right) \quad .$$

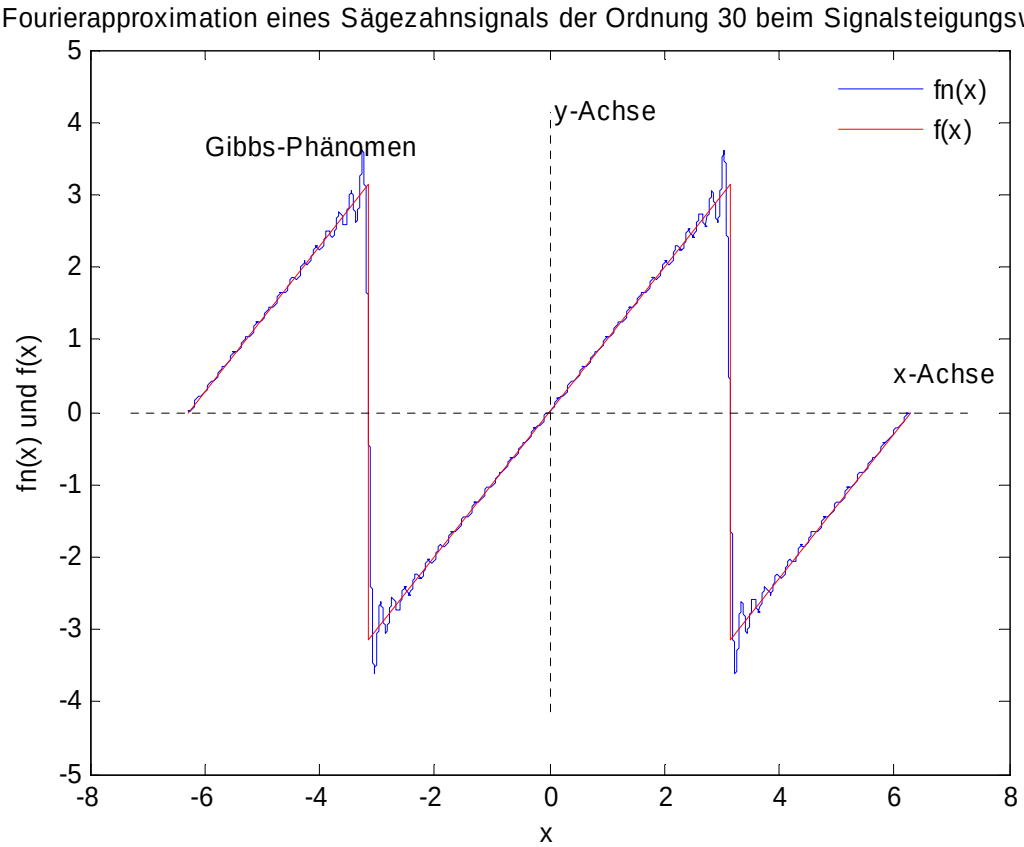
Das dazugehörige function-file in MATLAB ist dem obigen file zur Rechteckfunktion ähnlich:

```
 1-  %Fourierapproximation einer Sägezahnfunktion
 2-  function Saegezahnfunktion(a,n)
 3-  x=-2*pi:0.0002:2*pi;    %Definitionsbereich [-2*pi,2*pi] festlegen
 4-  fn=zeros(size(x));      %fn vorinitialisieren
 5-  for k=1:n               %fn als Summe berechnen
 6-     fn=fn+((-1)^(k+1)/k)*sin(k*x);
 7-  end
 8-  fn=fn*2*a;              %fn-Endsumme mit 2*a multiplizieren
 9-  plot(x,fn)              %Fourierapproximation fn zeichnen
10-  hold on                 %Zeichnung zusätzlich vorbereiten mit:
11-  x1=[-2*pi,-pi];x2=[-pi,pi];x3=[pi,2*pi];  %Sägezahnfunktion f
12-  y1=[0,a*pi];y2=[-a*pi,a*pi];y3=[-a*pi,0]; %mit Funktionswerten und
13-  xpi1=[-pi,-pi];ypi1=[-a*pi,a*pi];         %Unstetigkeitsstelle x=-pi,
14-  xpi2=[pi,pi];ypi2=[-a*pi,a*pi];           %Unstetigkeitsstelle x=+pi
15-  xAchse1=[-2*pi-1,2*pi+1];xAchse2=[0,0];   %sowie der x-Achse.
16-  yAchse1=[0,0];yAchse2=[-a*pi-1,a*pi+1];
17-  plot(x1,y1,'red-',x2,y2,'red-',x3,y3,'red-') %f plotten mit
18-  plot(xpi1,ypi1,'red-',xpi2,ypi2,'red')    %Sprung bei |x|=pi und
19-  plot(xAchse1,xAchse2,'black:',yAchse1,yAchse2,'black:') %x-y-Sstem
20-  text(-6,pi*a+0.5,'Gibbs-Phänomen');       %Text in Graphik einfügen
21-  title(['Fourierapproximation eines Sägezahnsignals der Ordnung ',...
22-      int2str(n), ' beim Signalsteigungswert ',num2str(a)]) %mit Titel
23-  xlabel('x');ylabel('fn(x) und f(x)');     %Bezeichnung der Achsen
24-  text(6,0.5,'x-Achse');text(0.1,a*pi+1,'y-Achse')
25-  legend('fn(x)','f(x)');legend('boxoff');  %Legende ohne Rahmen
26-  hold off
```

Mit den Eingabewerten $a = 1$ und $n = 3$ ergibt sich:



Mit den Eingabewerten $a = 1$ und $n = 30$ erhält man:



Auch hier ist das Gibbs-Phänomen nicht zu übersehen.
Die Sägezahnfunktion beschreibt beispielsweise beim Fernseher die waagerechte Bewegung des Lichtpunktes in Abhängigkeit der Zeit. Da man Sinusschwingungen durch elektrische Schwingkreise erzeugen und überlagern kann, so lässt sich die Lichtpunktbewegung durch die Fourierreihe der Sägezahnfunktion gewinnen.

Anmerkung:

Wählt man in (18) $a = 1$, so hat man für $x \in \,]-\pi\,,\,\pi[$ die Fourierdarstellung

$$x = 2\left(\frac{\sin x}{1} - \frac{\sin 2x}{2} + \frac{\sin 3x}{3} - \frac{\sin 4x}{4} + - \ \ldots\right) \text{ mithin } \quad \frac{x}{2} = \frac{\sin x}{1} - \frac{\sin 2x}{2} + \frac{\sin 3x}{3} - \frac{\sin 4x}{4} + - \ \ldots$$

Mit $x = \frac{\pi}{2}$ liefert das

$$\frac{\pi}{4} = \frac{\sin x}{1} - \frac{\sin 2x}{2} + \frac{\sin 3x}{3} - \frac{\sin 4x}{4} + \frac{\sin 5x}{5} - \frac{\sin 6x}{6} + \frac{\sin 7x}{7} - + \ldots$$

$$= \frac{\overbrace{\sin\left(\frac{\pi}{2}\right)}^{=1}}{1} - \frac{\overbrace{\sin 2\left(\frac{\pi}{2}\right)}^{=0}}{2} + \frac{\overbrace{\sin 3\left(\frac{\pi}{2}\right)}^{=-1}}{3} - \frac{\overbrace{\sin 4\left(\frac{\pi}{2}\right)}^{=0}}{4} + \frac{\overbrace{\sin 5\left(\frac{\pi}{2}\right)}^{=1}}{5} - \frac{\overbrace{\sin 6\left(\frac{\pi}{2}\right)}^{=0}}{6} + \frac{\overbrace{\sin 7\left(\frac{\pi}{2}\right)}^{=-1}}{7} - + \ldots$$

mithin $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - + \ldots = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$, die bekannte *Leibnizreihe*.

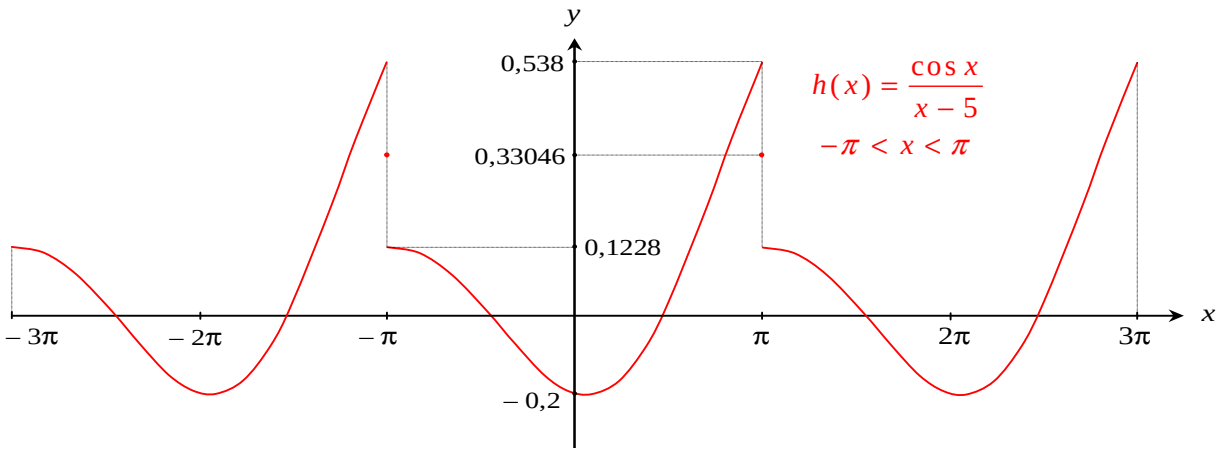
Ähnliche Formeln werden ebenfalls mit Hilfe von Fourierreihen erhalten.

Beispiel 3: ***Signal mit Sinus- und Cosinusgliedern***

Das folgende Signalbeispiel ist weder gerade noch ungerade. Die Fourierreihe wird somit sowohl Sinus- wie Cosinusglieder enthalten.

Es sei $h :]-\pi\,,\,\pi\,[\to \mathbb{R}$ definiert durch $h(x) = \frac{\cos x}{x-5}$ und auf diese Weise zu einer 2π-periodischen stückweise glatten Funktion f fortgesetzt.
Da die Fourierreihe von f an den *Sprungstellen* gegen den Mittelwert des rechts- und linksseitigen Grenzwertes von f konvergiert, sei per definitionem

$$f(\pm k\pi) \underset{def.}{=} \frac{1}{2}\left(\frac{\cos\pi}{\pi-5} + \frac{\cos(-\pi)}{-\pi-5}\right) = \frac{1}{2}\left(\frac{-1}{\pi-5} + \frac{-1}{-\pi-5}\right) = \frac{1}{2}\left(\frac{1}{\pi+5} - \frac{1}{\pi-5}\right) = \frac{5}{25-\pi^2} = 0{,}33046\ \ldots\ .$$



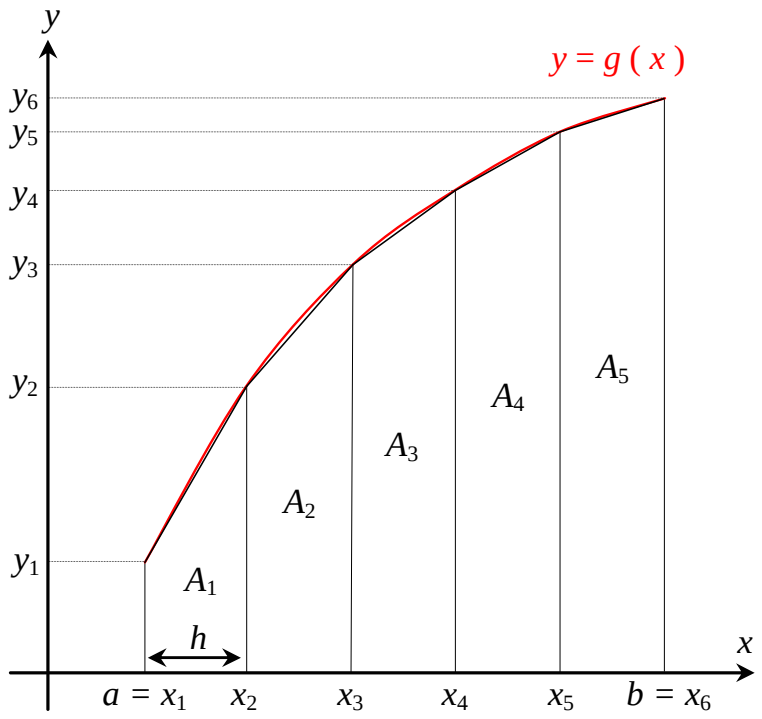
Die Fourierreihe dieser Funktion ist gegeben durch $f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty}\left(a_k \cos kx + b_k \sin kx\right)$ mit

(19) $$a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos kx\ dx = \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{\cos x}{x-5}\cos kx\ dx \quad \text{für } k = 0,1,2,3\ \ldots$$

(20) $$b_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin kx\ dx = \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{\cos x}{x-5}\sin kx\ dx \quad \text{für } k = 1,2,3,\ \ldots \text{ gemäß (14) und (15).}$$

Die Integrale (19) und (20) sind *nicht elementar lösbar*, sodass zu ihrer Bestimmung eine *numerische Integration* herangezogen werden muss! Wir verwenden dazu das *Trapezverfahren* und erläutern dieses an der *allgemeinen Problemstellung* , das Integral $\int_a^b g(x)\,dx$ einer stetigen Funktion g zu ermitteln.

Es handelt sich dabei um die *Fläche*, die der Graph von g mit der x-Achse zwischen den Integrationsgrenzen a und b einschließt.



Im Bild linkerhand sei eine Funktion $g : [\,a\,,\,b\,] \to \mathbb{R}$ dargestellt. Das Intervall $[\,a\,,\,b\,]$ werde mittels der "Schrittweite" $h = \frac{b-a}{5}$ in 5 gleichlange Teilintervalle unterteilt. Mittels der Punkte $(x_1\,,\,y_1)$ bis $(x_6\,,\,y_6)$ erhalten wir 5 Trapeze, deren Flächeninhalte addiert das Integral $\int_a^b g(x)\,dx$ annähern. Die Näherung wird um so besser, je kleiner die Schrittweite h ist. Jedes der 5 Trapeze hat nach der bekannten Trapezformel den Flächeninhalt

$$A_k = h\,\frac{y_k + y_{k+1}}{2}$$

Somit folgt im Bild linkerhand

$$\int_a^b g(x)\,dx \;\approx\; h\frac{y_1+y_2}{2} + h\frac{y_2+y_3}{2} + h\frac{y_3+y_4}{2} + h\frac{y_4+y_5}{2} + h\frac{y_5+y_6}{2} = \frac{1}{2}h\left(y_1 + 2y_2 + 2y_3 + 2y_4 + 2y_5 + y_6\right)$$

mithin bei Unterteilung in beliebige n gleiche Teilintervalle mit der Schrittweite $h = \frac{b-a}{n}$:

$$\int_a^b g(x)\,dx \;\approx\; \frac{1}{2}h\left(y_1 + 2\sum_{k=2}^{n} y_k + y_{n+1}\right) = \frac{1}{2}h\left(g(a) + 2\sum_{k=2}^{n} y_k + g(b)\right) = h\left(\frac{g(a)+g(b)}{2} + \sum_{k=2}^{n} g\left(x_k\right)\right)$$

oder:

(21) $$\boxed{\int_a^b g(x)\,dx = h\left(\frac{g(a)+g(b)}{2} + \sum_{k=2}^{n} g\left(x_k\right)\right) + R(h) \quad \text{mit} \quad \lim_{h\to 0} R(h) = 0}$$ ***Trapezregel***

Je kleiner die Schrittweite, desto genauer der Wert des Integrals, der mit (21) (ohne $R(h)$) erhalten wird.
Die Codierung der Trapezregel in MATLAB erfolgt so:

$h = (b - a) / n$	Schrittweite definieren
$x = a : h : b$	x–Wertebereich als Matrix anlegen
$y = g\ (x)$	y –Wertebereich als Matrix anlegen (*Punktoperation* bei Fixierung von g beachten!)

Integral = h * [(y (1) + y (end)) / 2 + (sum(y) $- y(1) - y$(end))]

Es ist $y\,(1) = g\,(a)$ und y(end) = $g\,(b)$ sowie sum (y) = $y_1 + \ldots + y_{n+1}$

und somit sum(y) $- y(1) - y$(end) = $\sum_{k=2}^{n} y_k = \sum_{k=2}^{n} g\left(x_k\right)$.

Die Berechnung der Fourierkoeffizienten gemäß (19) und (20) wird in eigenen function-files angelegt:
Bei der Berechnung der a_k ist zu beachten, dass a_0 gemäß (2) zu *halbieren* ist:

```
%Fourierkoeffizient ak berechnen
function ak=Fourierkoeffizient_ak(k)
h=pi/1000;
x=-pi:h:pi;
y=(1/pi)*(cos(x)./(x-5)).*cos(k*x);  %Punktoperation beachten!
if k==0
    ak=h*((y(1)+y(end))/2+(sum(y)-y(1)-y(end)))/2;
else
    ak=h*((y(1)+y(end))/2+(sum(y)-y(1)-y(end)));
end
```

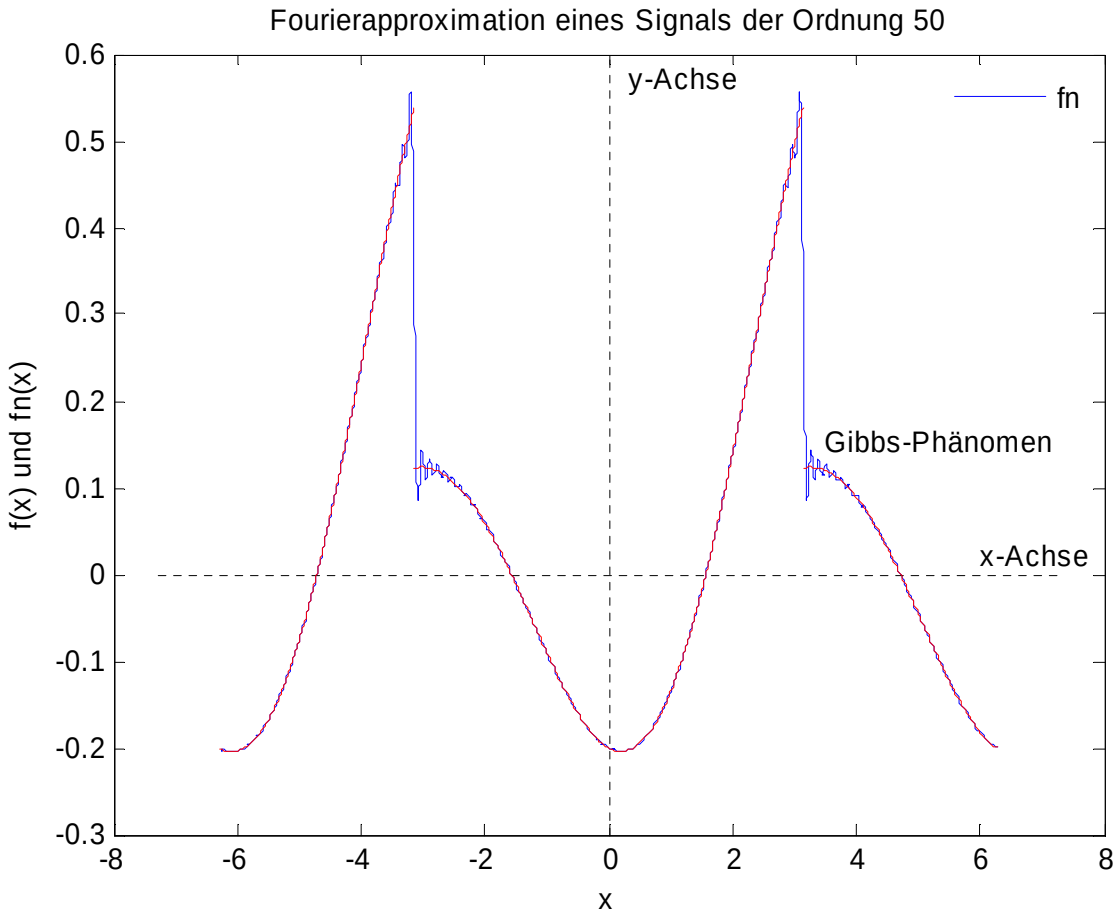
Bei der Berechnung der b_k entfällt die obige if-Abfrage:

```
%Fourierkoeffizient bk berechnen
function bk=Fourierkoeffizient_bk(k)
h=pi/1000;
x=-pi:h:pi;
y=(1/pi)*(cos(x)./(x-5)).*sin(k*x);  %Punktoperation beachten!
bk=h*((y(1)+y(end))/2+(sum(y)-y(1)-y(end)));
```

Das "Hauptprogramm" mit Ergebnis für $n = 50$ sieht so aus:

```
%Fourierapproximation einer 2pi-periodischen Funktion
function Fourierapproximation(n)
x=-2*pi:0.002:2*pi;                 %Bereich [-2*pi,2*pi] festlegen
fn=zeros(size(x));                  %fn vorinitialisieren
for k=0:n                           %fn als Summe berechnen
    fn=fn+Fourierkoeffizient_ak(k)*cos(k*x)+...
          Fourierkoeffizient_bk(k)*sin(k*x);
end
plot(x,fn,'blue')                   %Fourierapproximation fn zeichnen
legend('fn',1);legend('boxoff');    %Legende für fn ohne Rahmen
hold on                             %Zeichnung vorbereiten mit:
xAchse1=[-8,8];xAchse2=[0,0];       %x-Achse
yAchse1=[0,0];yAchse2=[-0.3;0.6];   %y-Achse
plot(xAchse1,xAchse2,'black:')      %x-Achse zeichnen (schwarz)
plot(yAchse1,yAchse2,'black:')      %y-Achse zeichnen (schwarz)
x=-2*pi:0.002:-pi;plot(x,cos(x+2*pi)./(x+2*pi-5),'red-') %f plotten
x=-pi:0.002:pi;plot(x,cos(x)./(x-5),'red')               %im Bereich
x=pi:0.002:2*pi;plot(x,cos(x-2*pi)./(x-2*pi-5),'red-')   %[-2*pi,2*pi]
text(3.5,0.15,'Gibbs-Phänomen');    %Text in Graphik einfügen
title(['Fourierapproximation eines Signals der Ordnung ',...
    int2str(n)])                    %Titel einfügen
xlabel('x')                         %Bezeichnung x-Achse
ylabel('f(x) und fn(x)')            %Bezeichnung y-Achse
hold off
```

Das folgende Bild ergibt sich bei einer Fourierapproximation mit 50 Summanden:



Auch hier ist das Gibbs-Phänomen an den Sprungstellen – π und π gut sichtbar.

IV. Parameteroptimierung

IV.1 Einführung

In der Praxis gibt es zahlreiche Aufgaben, eine Größe, die von mehreren Variablen (sog. Parametern oder Einflussgrößen)) abhängt, zu *minimieren* oder zu *maximieren*. Dazu sind die Parameter "optimal" zu bestimmen. Bei der Kraftstoffregelung eines Verbrennungsmoters beispielsweise sind die "Regelparameter" so zu bestimmen, dass das KFZ einen vorgeschriebenen Fahrzyklus mit *minimalem Kraftstoffverbrauch* durchfährt.

Mathematisch lautet die Aufgabenstellung so:

Gegeben sei eine "Zielfunktion" $f(\mathbf{x}) = f(x_1,\dots x_n) \in \mathbb{R}$ mehrerer Veränderlicher (Einflussgrößen) $x_1,\dots x_n$.
Gesucht ist ein Vektor $\mathbf{x}^* = (x_1^*,\dots x_n^*)$ mit $f(\mathbf{x}^*) = f(x_1^*,\dots x_n^*) =$ Minimum .

Ist f zu *maximieren* , so ist äquivalent dazu $g = -f$ zu minimieren.
Ohne Beschränkung der Allgemeinheit wollen wir uns also auf *Minimierungsaufgaben* beschränken.

Oft ist die *Hauptbedingung* $f(\mathbf{x}^*) = f(x_1^*,\dots x_n^*) =$ Minimum von einer zusätzlichen Forderung (sog. *Nebenbedingung*) begleitet. Im einfachsten Fall lässt sich eine solche Forderung als *Gleichung* in der Form $N(x_1,\dots x_n) = 0$ formulieren. Durch Auflösen dieser Gleichung nach einer der Variablen $x_1,\dots x_n$ und Einsetzen in die Hauptbedingung lässt sich dort die Variablenanzahl um 1 reduzieren.
Bei *mehreren* Nebenbedingungen können entsprechend viele Variablen in der Hauptbedingung eliminiert werden. Maximal sind $n - 1$ Nebenbedingungen möglich.
Nach Auswertung der genannten Nebenbedingungen ändert sich die obige mathematische Aufgabenstellung nicht. Die Zielfunktion ist lediglich um einige Variablen reduziert.

Liegen Nebenbedingungen z.B. in Form von *Ungleichungen* vor, gestaltet sich die Parameteroptimierung anspruchsvoller. Darauf sei hier nicht eingegangen. Infolgedessen und aufgrund der obigen Überlegungen beschränken wir uns auf *Minimierungsaufgaben ohne Nebenbedingungen.*

Ferner betrachten wir nur Zielfunktionen mit $n = 2$ Variablen also $f(\mathbf{x}) = f(x_1, x_2)$ und schreiben dafür $z = f(x, y)$. Gesucht ist im folgenden $\mathbf{x}^* = (x^*, y^*)$, sodass $z = f(\mathbf{x}^*) = f(x^*, y^*) =$ Minimum gilt.

IV.2 Lösung von Minimierungsaufgaben mittels Differentialrechnung

Ist $y = f(x)$ eine Funktion *einer* Veränderlichen, so ist das Lösen der Minimierungsaufgabe $f(x)$ = Minimum aus der Schule bekannt: Man bestimme die Lösung x^* der Gleichung $f'(x) = 0$. Ist $f''(x^*) > 0$, so ist x^* die gesuchte Lösung der Minimierungsaufgabe.

Ist $z = f(x , y)$ eine Funktion *zweier* Veränderlicher, so erfolgt die Lösung der Minimierungsaufgabe $f(x , y)$ = Minimum auf folgende Weise:
Man bestimme die Lösung $\mathbf{x}^* = (x^*, y^*)$ der beiden Gleichungen $f_x(x, y) = 0$ und $f_y(x, y) = 0$.

Ist die Determinante $\begin{vmatrix} f_{xx}(x^*, y^*) & f_{xy}(x^*, y^*) \\ f_{yx}(x^*, y^*) & f_{yy}(x^*, y^*) \end{vmatrix} > 0$ und gilt $f_{xx}(x^*, y^*) > 0 < f_{yy}(x^*, y^*)$,

so ist $\mathbf{x}^* = (x^*, y^*)$ die gesuchte Lösung der Minimierungsaufgabe.

Dabei bezeichnen $f_x = \frac{\partial f}{\partial x}$, $f_y = \frac{\partial f}{\partial y}$, $f_{xx} = \frac{\partial}{\partial x}\frac{\partial f}{\partial x} = \frac{\partial^2 f}{\partial x^2}$, $f_{yy} = \frac{\partial}{\partial y}\frac{\partial f}{\partial y} = \frac{\partial^2 f}{\partial y^2}$

und $f_{yx} = \frac{\partial}{\partial x}\frac{\partial f}{\partial y} = \frac{\partial^2 f}{\partial x \partial y}$, $f_{xy} = \frac{\partial}{\partial y}\frac{\partial f}{\partial x} = \frac{\partial^2 f}{\partial y \partial x}$

die *partiellen Ableitungen* von f nach x und y bis zur 2. Ordnung.

f_{yx} und f_{xy} heißen *gemischte* Ableitungen. Diese sind gleich, wenn die obigen Ableitungen von f sämtlich existieren und stetig sind (Schwarz'sche Identität). Im technischen Bereich ist dies praktisch immer der Fall, sodass wir dies voraussetzen und nicht weiter vertiefen wollen.

Beispiel IV.2.1:

$f(x,y) = x^2 - 2xy + 2y^2 - y$.

Es ist $f_x(x,y) = 2x - 2y \underset{setze}{=} 0$ und $f_y(x,y) = -2x + 4y - 1 \underset{setze}{=} 0 \Rightarrow x = y = \frac{1}{2}$.

Für diese Werte ist $f_{xx} = 2 > 0$, $f_{yy} = 4 > 0$, $f_{xy} = f_{yx} = -2 < 0$ und $\begin{vmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{vmatrix} = \begin{vmatrix} 2 & -2 \\ -2 & 4 \end{vmatrix} = 4 > 0$.

Also besitzt f im Punkt $\mathbf{x}^* = (x^*, y^*) = \left(\frac{1}{2}, \frac{1}{2}\right)$ ein Minimum und dieses ist gleich $f\left(\frac{1}{2}, \frac{1}{2}\right) = -\frac{1}{4} = -0.25$.

Mit Matlab lassen sich die Ergebnisse anschaulich darstellen:

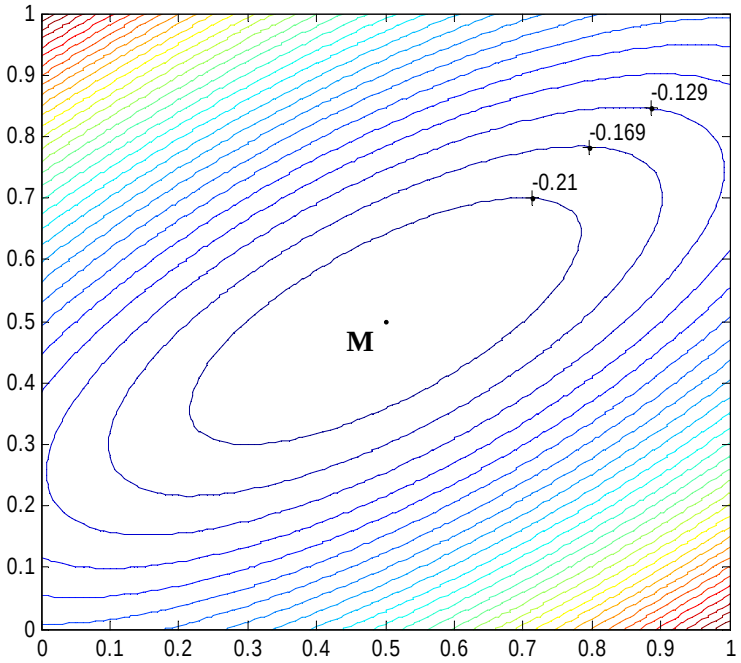


Abb. 1

Das Bild linkerhand zeigt einen "***Contour-Plot*** " von f. Die "Niveaulinien" sind Ellipsen, auf welchen die Funktionswerte einen konstanten Wert haben. Für 3 Niveaulinien sind die konstanten Funktionswerte eingeblendet. Innerhalb der innersten Niveaulinie liegt der Punkt M = (0.5 , 0.5) , wo die Funktion f den Minimalwert -0.25 annimmt.

Punkt M ist mit Bezeichnung *nachträglich eingefügt*.

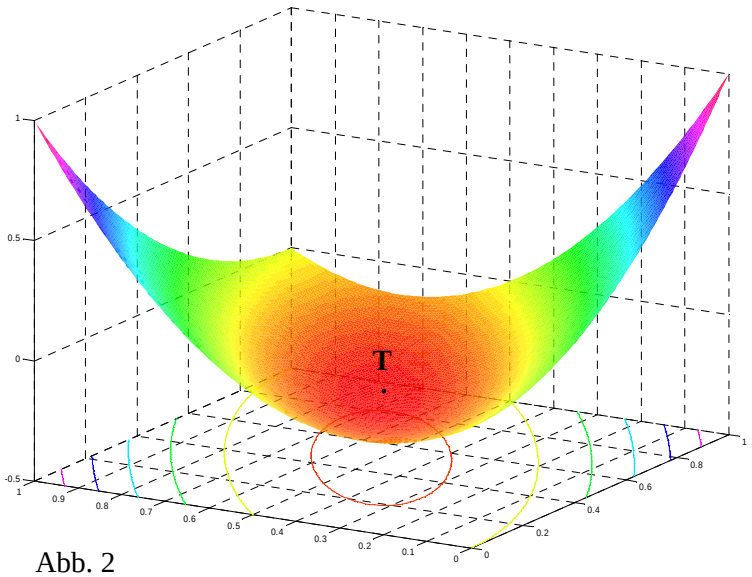


Abb. 2

Dieses Bild linkerhand zeigt einen "***Surface-Plot*** " von f im einem 3-dimensionalen x-y-z-Koordinatensystem. Die "Niveaulinien" sind in die x-y-Ebene projeziert. Deren Farben entsprechen den Farben des Funktionsgraphen. T = (0.5 , 0.5 , – 0.25) ist der "tiefste" Punkt des Graphen, wo f den Minimalwert -0.25 annimmt.

Punkt T ist mit Bezeichnung *nachträglich eingefügt*.

Beispiel IV.2.2:

$f(x,y) = x^2 - 4xy + 2y^2 - y$

Es ist $f_x(x,y) = 2x - 4y \underset{setze}{=} 0$ und $f_y(x,y) = -4x + 4y - 1 \underset{setze}{=} 0 \Rightarrow x = -\frac{1}{2}$, $y = -\frac{1}{4}$.

Für diese Werte ist $f_{xx} = 2 > 0$, $f_{yy} = 4 > 0$, $f_{xy} = f_{yx} = -4 < 0$ und $\begin{vmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{vmatrix} = \begin{vmatrix} 2 & -4 \\ -4 & 4 \end{vmatrix} = -8 < 0$.

Bei *negativer* Determinante liegt stets *kein* Extremum vor!

Wegen $f_x\left(-\frac{1}{2},-\frac{1}{4}\right) = f_y\left(-\frac{1}{2},-\frac{1}{4}\right) = 0$ ist $\mathbf{x}^* = (x^*, y^*) = \left(-\frac{1}{2},-\frac{1}{4}\right)$ ein *Sattelpunkt* und es ist $f\left(-\frac{1}{2},-\frac{1}{4}\right) = \frac{1}{8} = 0.125$. Die folgenden Graphiken veranschaulichen die Ergebnisse:

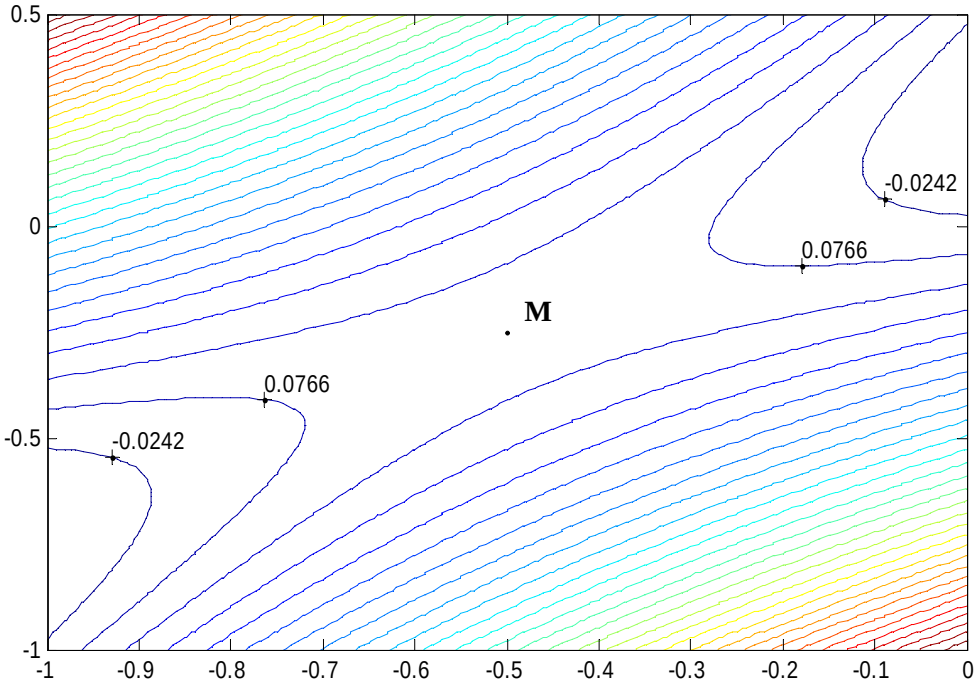


Abb. 3

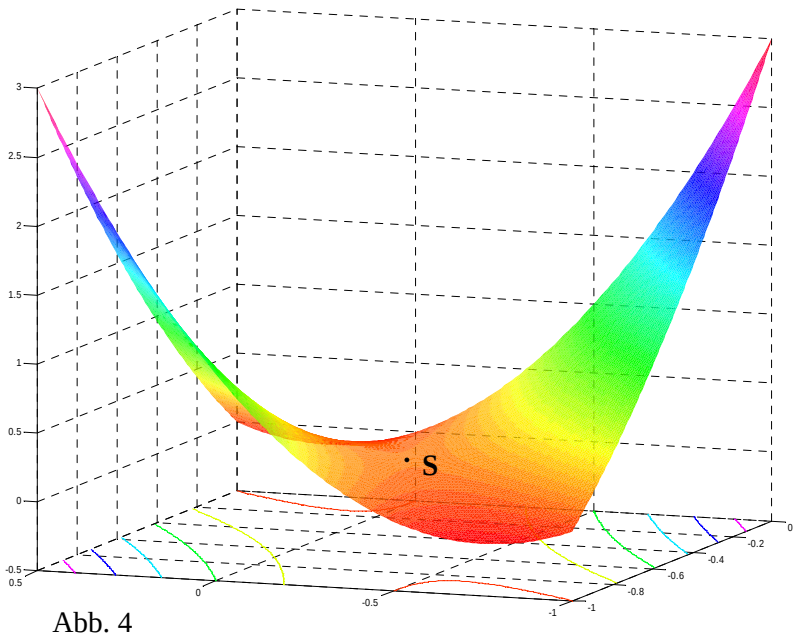


Abb. 4

Die Niveaulinien sind in diesem Falle Hyperbeln. Im obigen Bild ist der Punkt M = (– 0.5 , – 0.25) im Contour-Plot eingetragen.

Das Bild linkerhand zeigt den Surface-Plot mit dem Sattelpunkt S = (– 0.5 , – 0.25 , 0.125). Man erkennt: Die Fläche verläuft ausgehend von S nach links und rechts *aufwärts* sowie nach hinten und vorne *abwärts*. Daher ist der Wert $z = 0.125$ kein Extremum. Die Punkte M und S sind mit Bezeichnung *nachträglich eingefügt*.

IV.3 Graphische Darstellung einer Funktion $z = f(x, y)$

Die obigen Graphiken am Beispiel der vorangegangenen Funktion $f(x,y) = x^2 - 4xy + 2y^2 - y$ gewinnt man mit Matlab mit den folgenden script-files:

```
1   %Darstellung von z=f(x,y) mit Niveaulinien als "Contour-Plot"
2-  [x,y]=meshgrid(-1:0.01:0,-1:0.01:0.5);
3-  z=x.^2-4*x.*y+2*y.^2-y; %Punktoperation erforderlich!
4-  C=contour(x,y,z,30); %Der Plot enthält 30 Niveaulinien (Höhenlinien)
5-  clabel(C,'manual')
```

In Zeile 2 wird mit dem Befehl **`meshgrid`** ein Rechennetz in der *x*-*y*-Ebene im Bereich $-1 \le x \le 0$ und $-1 \le y \le 0.5$ jeweils mit der Schrittweite 0.01 erzeugt. x und y werden dabei als *Matrizen* angelegt.
In Zeile 3 ist demnach die Funktion $z = f(x, y)$ mittels *Punktoperation* als *Matrix* zu definieren.
Zeile 4 erzeugt mit dem Befehl **`contour`** den Contour-Plot in Abb. 3 mit 30 Niveaulinien (Höhenlinien) zwischen dem kleinsten und größten Niveau.
Zeile 5 gestattet dem Benutzer, ausgewählte Niveaulinien mit ihrem Funktionswert zu kennzeichnen. Dazu ist in der Graphik der Cursor (ein Fadenkreuz) auf eine Niveaulinie zu plazieren und dann diese Position mit Linksklick zu bestätigen.
Die Ausführung des script-files erzeugt das obige Bild in Abb. 3 (ohne den Punkt M !).

```
1   %Darstellung des Funktionsgraphen z=f(x,y) als "Surface-Plot"
2-  [x,y]=meshgrid(-1:0.01:0,-1:0.01:0.5);
3-  z=x.^2-4*x.*y+2*y.^2-y; %Punktoperation erforderlich!
4-  surfc(x,y,z);
```

Zeile 2 und 3 sind mit den Zeilen 2 und 3 im ersten script-file identisch.
Zeile 4 erzeugt mit dem Befehl **`surfc`** den Graphen von f (Funktionsfläche) über dem mittels **`meshgrid`** definierten Bereich zusammen mit projezierten Niveaulinien in der *x*-*y*-Ebene.
Die Ausführung des script-files erzeugt das obige Bild in Abb. 4 (ohne den Punkt S !).
Der Befehl **`surf`** (statt **`surfc`**) erzeugt dasselbe Bild *ohne* die Niveaulinien.

Zur besseren Farbgebung des Graphen aktiviere man im betreffenden Graphen-Fenster die Schaltfläche mit dem Pfeil und wähle mittels Rechtsklick auf die Graphik im erscheinenden Kontextmenü den Eintrag *Line Style* . In dessen Untermenü aktiviere man schließlich *none* .

IV.4 Optimierungsverfahren

Die beschriebene Minimumsuche mittels Differentialrechnung ist im allgemeinen zu kompliziert. Numerische Optimierungsverfahren erweisen sich als einfacher und effizienter. Auch in diesen Verfahren ergeben sich die gesuchten Lösungen als Grenzwerte von *Iterationsverfahren*. Im folgenden seien die wichtigsten Verfahren an einfachen Beispielen beschrieben.

IV.4.1 Das Gradientenverfahren

Gesucht ist ein lokales Minimum der Funktion $z = f(x, y)$. Die Ableitung von f ist dessen *Gradient*, also

(1) $\quad \text{grad } f = \nabla f = \begin{pmatrix} f_x \\ f_y \end{pmatrix}$.

Für den Gradienten wollen wir die Bezeichnung "grad" verwenden. Das Symbol ∇ bezeichnet man als *Nabla-Operator* und ist alternativ für den Gradienten gebräuchlich.

Wir behandeln das ***Verfahren des steilsten Abstiegs* (*Steepest -Descent -Verfahren*)** und zeigen zunächst:

Ist $z = f(x, y)$ vorgegeben, so ist an jeder Stelle $\mathbf{x} = (x, y)$ die Richtung des "steilsten Abstiegs" von f durch $-\text{grad } f(\mathbf{x})$ gegeben.

Beweis (für den interessierten Leser und kann daher auch übersprungen werden):

Sei $\mathbf{x_0}$ ein beliebiger Punkt und $\mathbf{s} = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$ ein beliebiger Richtungsvektor mit $|\mathbf{s}| = 1$.

Wir betrachten f längs der Geraden $\mathbf{x} = \mathbf{x}_0 + t\,\mathbf{s}$, $t \in \mathbb{R}$, und leiten $g(t) \underset{def.}{=} f(\mathbf{x}_0 + t\,\mathbf{s})$ nach t ab.

Es folgt $\frac{dg}{dt}(t) = \underbrace{\text{grad } f(\mathbf{x}_0 + t\,\mathbf{s})}_{\text{äussere Ableitung}} \cdot \underbrace{\mathbf{s}}_{\substack{\text{innere}\\ \text{Ableitung}}}$ im Sinne des Skalarprodukts für Vektoren, also

(2) $$\frac{dg}{dt}(t) = \left|\text{grad } f(\mathbf{x}_0 + t\,\mathbf{s})\right| \cdot \underbrace{|\mathbf{s}|}_{=1} \cdot \cos\varphi \ ,$$

wo φ den Winkel bezeichnet, den der Gradient (als Vektor gemäß (1)) mit dem Richtungsvektor $\mathbf{s}$ einschließt. (2) liefert insbesondere für $t = 0$ die Abschätzung

(3) $$\left|\frac{dg}{dt}(0)\right| = \left|\text{grad } f(\mathbf{x}_0)\right| \cdot \underbrace{|\cos\varphi|}_{\le 1} \le \left|\text{grad } f(\mathbf{x}_0)\right| \ .$$

$\frac{dg}{dt}(0)$ ist die Steigung von f in $\mathbf{x_0}$ in Richtung $\mathbf{s}$. Weist der Richtungsvektor $\mathbf{s}$ ausgehend von $\mathbf{x_0}$ nun *in Richtung von* $\text{grad } f(\mathbf{x_0})$, so ist $\varphi = 0$ also $\cos\varphi = 1$ und somit $\frac{dg}{dt}(0) = \left|\text{grad } f(x_0)\right| > 0$.

Wegen (3) weist $\text{grad } f(\mathbf{x}_0)$ somit in die Richtung des *größten Anstiegs* von f.

$-\text{ grad } f(\mathbf{x}_0)$ ist infolgedessen die Richtung des *größen* (*steilsten*) *Abstiegs* von f.

Damit ist die obige Behauptung bewiesen.

Mit der gewonnenen Erkenntnis wird der Algorithmus für das Verfahren des steilsten Abstiegs (**Steepest-Descent-Verfahren)** für eine vorgegebene Funktion $z = f(x, y)$ so aussehen:

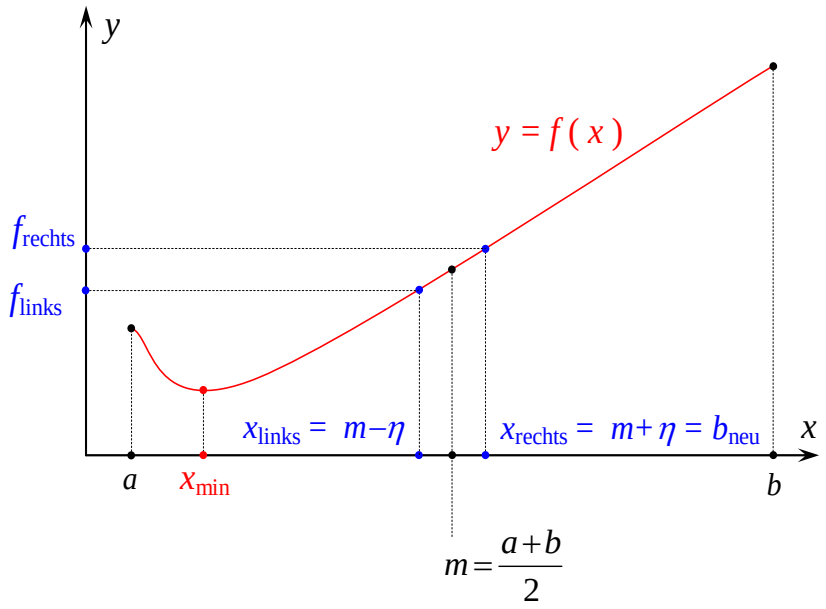
(4.1)	Wähle einen Startpunkt $\mathbf{x_0} = (x_0 , y_0)$.
(4.2)	Bestimme ausgehend von $\mathbf{x_0}$ die Richtung $\mathbf{s_0}$ des steilsten Abstiegs: $\mathbf{s_0} = -\text{ grad } f(\mathbf{x_0})$.
(4.3)	Gehe von $\mathbf{x_0}$ aus solange in die Richtung $\mathbf{s_0}$, wie es weiter "bergab" geht, genauer: Bestimme die erreichte Minimumstelle $\mathbf{x_1}$ von $\{ f(\mathbf{x_0} + t\,\mathbf{s_0}) / t \ge 0 \}$.
(4.4)	Wähle in (4.1) nun $\mathbf{x_0} = \mathbf{x_1}$ und wiederhole damit die Schritte (4.2) und (4.3). In Schritt (4.2): $\mathbf{s_1} = -\text{ grad } f(\mathbf{x_1})$. In Schritt (4.3): Minimumstelle $\mathbf{x_2}$ von $\{ f(\mathbf{x_1} + t\,\mathbf{s_1}) / t \ge 0 \}$. usw.

Das Verfahren liefert eine Iterationsfolge $\mathbf{x_0}$, $\mathbf{x_1}$, $\mathbf{x_2}$, ... , die gegen die Minimumstelle $\mathbf{x}^*$ konvergiert.

IV.4.1.1 Dichotome 1-dimensionale Minimumsuche

Zur Vorbereitung wollen wir das Prinzip der Minimumsuche für eine Funktion $f : [a,b] \to \mathbb{R}$ *einer* Veränderlichen erläutern. f sei *unimodal*, d.h. sie besitze im Intervall [a , b] *genau ein* Minimum an der Stelle $x_{\min} \in [a , b]$. In diesem Zusammenhang wird $[a , b]$ *Unbestimmtheitsintervall* oder *Suchintervall* genannt. Wir sprechen von der sog. "dichotomen 1-dimensionalen Minimumsuche".

Der Algorithmus zur Auffindung von $x_{\min}$ verläuft so:



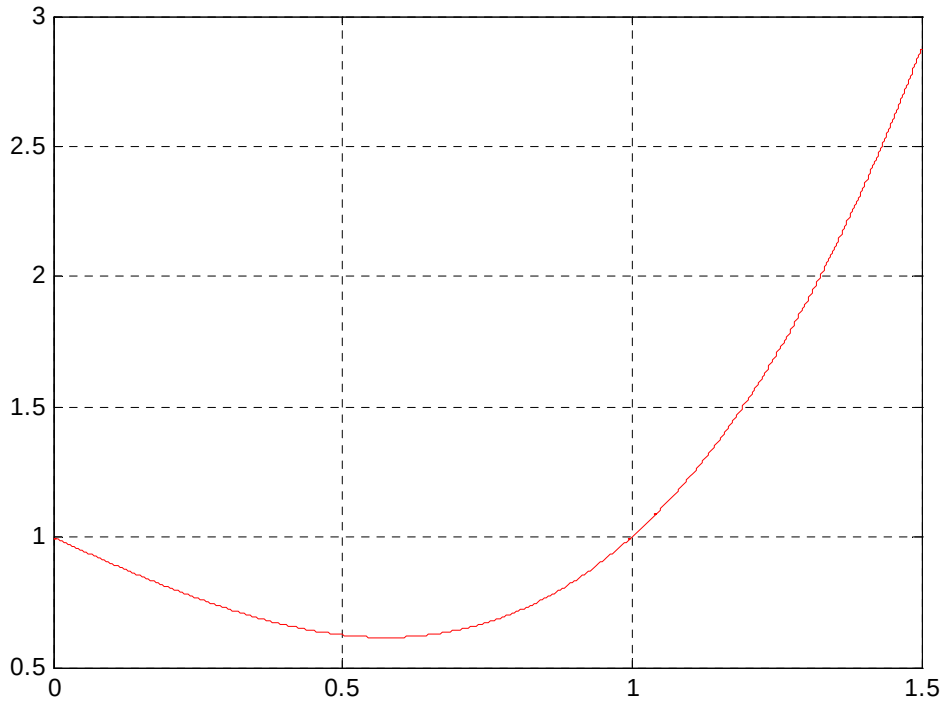
(a) Bilde Mittelpunkt von $[a , b]$
$m = (a + b) / 2$
(b) Wähle "Trennschärfe" $\eta > 0$ mit sehr kleinem Wert und
setze $x_{\text{links}} = m - \eta$
und $x_{\text{rechts}} = m + \eta$
(c) Berechne $f_{\text{links}} = f(x_{\text{links}})$
und $f_{\text{rechts}} = f(x_{\text{rechts}})$
(d) Vergleiche f_{links} mit f_{rechts} :
Wenn $f_{\text{links}} \le f_{\text{rechts}}$
$\Rightarrow$ Setze $b = x_{\text{rechts}}$
Wenn $f_{\text{links}} > f_{\text{rechts}}$
$\Rightarrow$ Setze $a = x_{\text{links}}$
(e) Kehre zurück zu (a) *mit neuen Werten* für a bzw. b und durchlaufe alle Schritte erneut.

Bezeichnen wir die "neuen Werte" mit a_{neu} und b_{neu} , so wird auf diese Weise das Anfangsintervall $[a\,,\,b]$ *in etwa halbiert* zum Intervall $[a_{neu}\,,\,b_{neu}\,]$ und zwar so, dass $x_{min} \in [a_{neu}\,,\,b_{neu}\,]$ gilt usw. x_{min} wird also mittels einer "Intervallschachtelung" ermittelt. Nach n Durchläufen hat sich die Länge des Intervalls (welches x_{min} enthält) ungefähr auf den 2^n- ten Teil der Anfangslänge reduziert.

Vorgehensweise mit MATLAB:

Wir betrachten als Beispiel die Funktion $f(x) = x^3 - x + 1$ im Intervall $\left[0\,,\ \frac{3}{2}\right]$.

Die Graphik von f sieht so aus:



Das folgende script-file erstellt die obige Graphik und ermittelt die Lage des Minimums:

```
 1-  f=@(x)x.^3-x+1;
 2-  a=0;b=3/2;eta=1e-10;k_max=20;
 3-  for k=1:k_max  % Anzahl der Schleifendurchläufe begrenzen
 4-      m=(a+b)/2;x_li=m-eta;x_re=m+eta; % re = "rechts"
 5-      f_li=f(x_li);f_re=f(x_re);       % li = "links"
 6-      if f_li<=f_re
 7-          b=x_re;
 8-      else
 9-          a=x_li;
10-      end
11-  end   % Der letzte Wert von m ist das gesuchte Ergebnis x_min.
12-  m=(a+b)/2;x_min=m;f_min=f(x_min);
13-  fprintf('Die Minimumstelle liegt bei x_min = %.6f\n',x_min);
14-  fprintf('Das Minimum von f beträgt f(x_min) = %.6f\n',f_min);
15-  x=0:0.01:3/2;plot(x,f(x),'red');grid on;
```

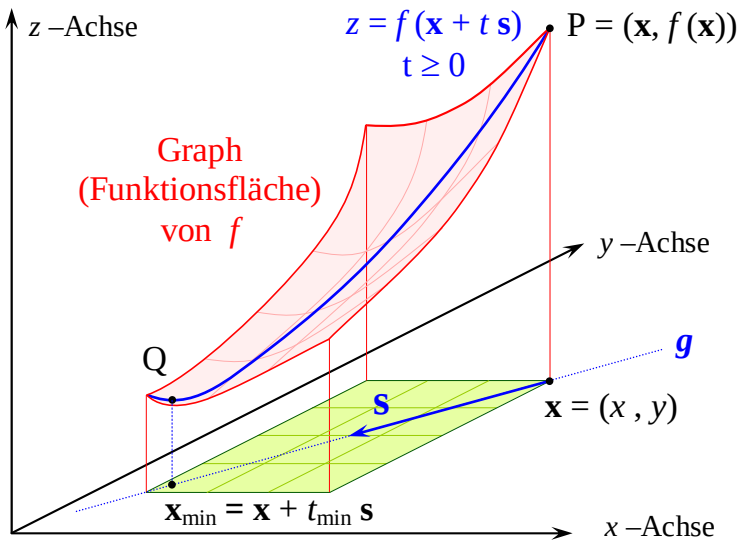
Nach Start des script-files erscheint neben der Graphik im Command-Window das Ergebnis:

```
Die Minimumstelle liegt bei x_min = 0.577351
Das Minimum von f beträgt f(x_min) = 0.615100
```

Der Programm-Code ist direkt aus dem oben beschriebenen Algorithmus ersichtlich.
Man beachte, dass wegen des plot-Befehls in Zeile 15 die Funktion f in Zeile 1 mittels *Punktoperation* deklariert werden muss.

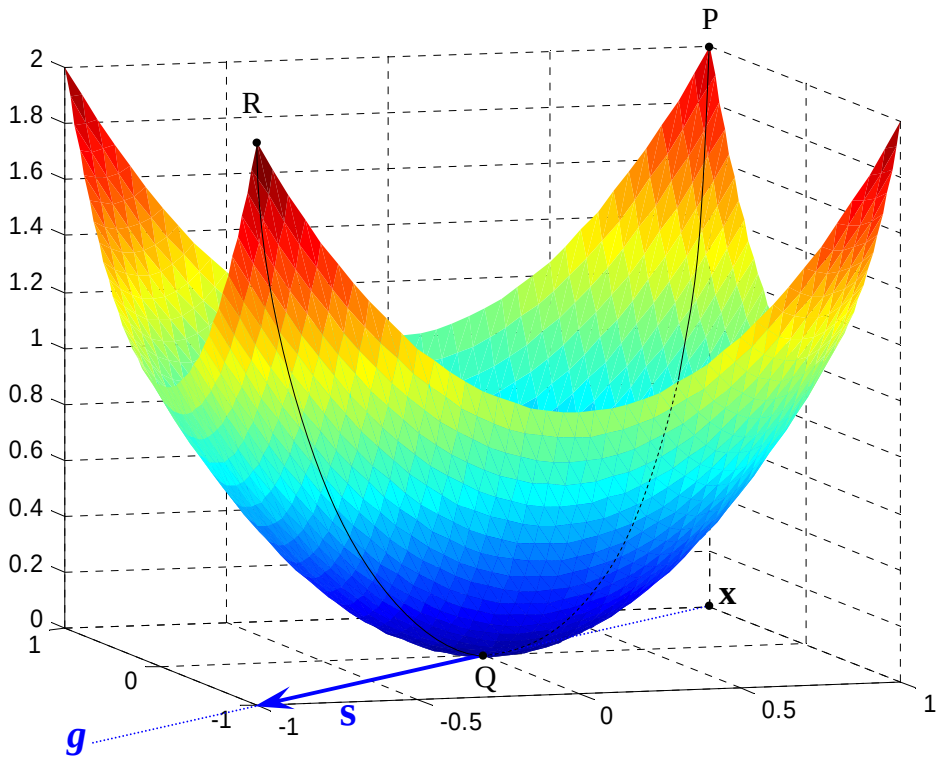
IV.4.1.2 Anwendung der dichotomen 1-dimensionalen Minimumsuche auf Schritt (4.3) des Steepest-Descent-Algorithmus (Dichotome 2-dimensionale Minimumsuche)

Es sei **s** die Richtung des steilsten Abstiegs der Zielfunktion $z = f(x, y)$ in einem fixierten Punkt $\mathbf{x} = (x , y)$.



Die Idee lautet:
Man gehe von **x** aus (auf dem Graphen von f vom Ort P aus) solange in Richtung von **s** , wie es auf dem Graphen weiter "bergab" geht, d.h. man ermittle den tiefsten Punkt Q der Funktionsfläche von f über der Geraden
$g = \{\mathbf{x} + t\ \mathbf{s}\ ,\ t \in \mathbb{R}\}$.
Anders formuliert:
Man bestimme $t = t_{min}$ so, dass gilt:
$f(\mathbf{x} + t\ \mathbf{s})$ = Minimum .
Der tiefste Punkt Q liegt dann über der Stelle $\mathbf{x}_{min} = \mathbf{x} + t_{min}\ \mathbf{s}$.
Für Q selbst gilt $Q = (\mathbf{x}_{min} , f(\mathbf{x}_{min}))$.

Die Funktion $t \to f(\mathbf{x} + t\ \mathbf{s})$ ist eine Funktion der *einzigen* Veränderlichen t, sodass auf diese Funktion die dichotome 1-dimensionale Minimumsuche angewandt werden kann.



Beispiel: $f(x, y) = x^2 + y^2$ für $x \in [-1 , 1]$ und $y \in [-1 , 1]$.

Der Graph von f ist ein *Paraboloid* (Erzeugung mit MATLAB siehe IV.3). Das Minimum liegt offenbar bei $\mathbf{x}_{min} = (0 , 0)$ mit $f_{min} = f(\mathbf{x}_{min}) = 0$. Gemäß dem *obigen* Bild ist also Q = (0 , 0 , 0).
Wir wählen $\mathbf{x} = (1 , 1)$ und $\mathbf{s} = (-1 , -1)$.
Die Punktmenge $\{\ \mathbf{x} + t\ \mathbf{s}\ , t \in [0 , 2]\ \}$ ist die Gerade g von $(1 , 1) = \mathbf{x}$ bis $(-1 , -1)$.
Die Menge $\{\ \mathbf{x} + t\,\mathbf{s}\ ,\ f(\mathbf{x} + t\ \mathbf{s})\ ,\ t \in [0 , 2]\ \}$ ist der Parabelbogen über g von P versus Q nach R.
Auf die Funktion $t \to f(\mathbf{x} + t\ \mathbf{s})$, $t \in [0 , 2]$ sei somit die o.g. Minimumsuche angewandt:

Das function-file bezeichnen wir mit **`Dichotome_2D_Minimumsuche`** , da das Minimum einer Funktion von *zwei* Veränderlichen gesucht wird. Übergeben werden $a = 0$, $b = 2$, $\eta = 10^{-10}$, $k\_max = 20$ sowie $\mathbf{x} = [1\ ,\ 1]$ und $\mathbf{s} = [-1,-1]$. Im Command-Window ist *zuerst* `f=@(x)x(1)^2+x(2)^2;` einzugeben.

```
 1-  function [x_min,f_min]=Dichotome_2D_Minimumsuche(a,b,eta,k_max,x,s,f)
 2-  for k=1:k_max
 3-     m=(a+b)/2;t_li=m-eta;t_re=m+eta;
 4-     f_li=f(x+t_li*s);
 5-     f_re=f(x+t_re*s);
 6-     if f_li<=f_re
 7-         b=t_re;
 8-     else
 9-         a=t_li;
10-     end
11-  end
12-  m=(a+b)/2;x_min=x+m*s;f_min=f(x_min);
```

Mit `[x_min,f_min]=Dichotome_2D_Minimumsuche(0,3,1e-10,100,[1,1],[-1,-1],f)` im Command-Window wird erwartungsgemäß `x_min = 0` sowie `f_min = 0` ausgegeben.

IV.4.1.3 Steepest-Descent-Algorithmus

Das in IV.1 beschriebene Steepest-Descent-Verfahren (Schritte 4.1– 4) kann nun implementiert werden.
Als Funktionsbeispiel verwenden wir $z = f(x,y) = x^2 - 2xy + 2y^2 - y$, $x \in [-2\ ,\ 2]$ und $y \in [-2\ ,\ 2]$.
Das folgende script-file (Name: **`Steepest_Descent_Verfahren`**) erzeugt einen Contour-Plot von f und stellt folgende Daten bereit: Die Funktion f = f sowie die partiellen Ableitungen f1 = f_x und f2 = f_y zur Bestimmung des Gradienten von f, den Startwert $\mathbf{x} = \mathbf{x_0}$, die maximale Schleifenanzahl k_max sowie die Konvergenzschranke Eps = ε für die gewünschte Genauigkeit der Ergebnisse x_min und f_min.

```
 1-  [x,y]=meshgrid(-2:0.01:2,-2:0.01:2);
 2-  z=x.^2-2*x.*y+2*y.^2-y;
 3-  C=contour(x,y,z,20);
 4-  clabel(C,'manual')
 5-  xo=[-1,-1.5];k_max=80;Eps=1e-4;
 6-  f=@(x)x(1)^2-2*x(1)*x(2)+2*x(2)^2-x(2);
 7-  f1=@(x)2*x(1)-2*x(2);
 8-  f2=@(x)-2*x(1)+4*x(2)-1;
 9-  [x_min,f_min,k]=Gradienten_Verfahren(xo,k_max,Eps,f,f1,f2);
10-  fprintf('Das Minimum liegt bei (x_min,y_min) = (%.4f,%.4f)\n',x_min);
11-  fprintf('Der Minimalwert von f beträgt f_min = %.4f\n',f_min);
12-  fprintf('Es wurden %u Iterationen benötigt',k);
```

Das script-file gibt auch die Ergebnisse x_min und f_min für das Minimum aus.

Dazu ruft das script-file das function-file **`Gradienten_Verfahren`** auf , welches ausgehend vom Startvektor $\mathbf{x_0}$ gemäß (4.2) den Richungsvektor $\mathbf{s_0} = -\text{grad}\ f(\mathbf{x_0})$ und daraus gemäß (4.3) den Folgevektor $\mathbf{x_1}$ ermittelt. Die Berechnung erfolgt mit dem function-file **`Dichotome_2D_Minimumsuche`** , welches in jedem Iterationsschritt von **`Gradienten_Verfahren`** aufgerufen wird:

```
 1-  function [x_min,f_min,k]=Gradienten_Verfahren(xo,k_max,Eps,f,f1,f2)
 2-  x=xo;
 3-  for k=1:k_max
 4-     grad_f=[f1(x),f2(x)];
 5-     if abs(grad_f(1))+abs(grad_f(2))<=Eps
 6-         break
 7-     end
 8-     s=-grad_f;
 9-     eta=1e-10;
10-     [x_min,f_min]=Dichotome_2D_Minimumsuche(0,1,eta,100,x,s,f);
11-     x=x_min;
12-  end
```

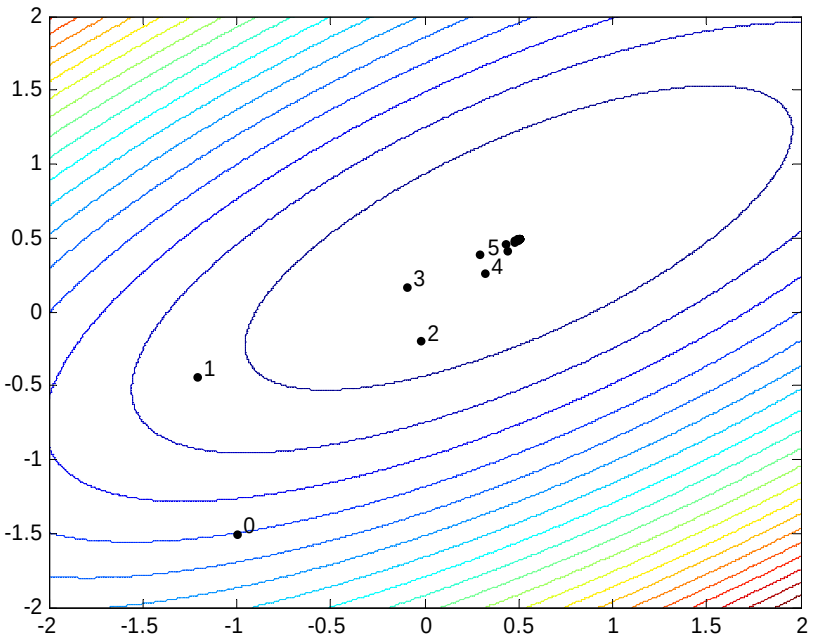
Das function-file **Gradienten_Verfahren** erzeugt eine Iterationsfolge $\mathbf{x_0}$, $\mathbf{x_1}$, $\mathbf{x_2}$, ... , die gegen das gesuchte Minimum $\mathbf{x}^*$ konvergiert. Die Konvergenz ist (ohne Beweis) *linear*. Will man sich die Punkte $\mathbf{x_k}$ im Contour-Plot ansehen, so ergänze man **Gradienten_Verfahren** wie folgt (zusätzliche Zeilen 4 – 9):

```
 1-  function [x_min,f_min,k]=Gradienten_Verfahren(xo,k_max,Eps,f,f1,f2)
 2-  x=xo;
 3-  for k=1:k_max
 4-     hold on;
 5-     plot(x(1),x(2),'black.');
 6-     if k<=6
 7-         text(x(1)+0.04,x(2)+0.04,int2str(k-1));
 8-     end
 9-     hold off;
10-     grad_f=[f1(x),f2(x)];
11-     if norm(grad_f)<=Eps   %Euklidische (Pythagoreische) Norm
12-        break               %als Alternative zur Abfrage
13-     end                    %abs(grad_f(1))+abs(grad_f(2))<=Eps (s.o.)
14-     s=-grad_f;
15-     eta=1e-10;
16-     [x_min,f_min]=Dichotome_2D_Minimumsuche(0,1,eta,100,x,s,f);
17-     x=x_min;
18-  end
```

Mit den Zeilen 6 – 8 werden die ersten 6 Vektoren $\mathbf{x}_0$, ... , $\mathbf{x}_5$ mit ihren Indizes gekennzeichnet, um den Verlauf der Iterationsfolge zu erkennen. Da $\mathbf{x}_0$ als Startvektor das erste Element ist, muss in Zeile 7 `int2str(k-1)` codiert werden. Die Minimumstelle $\mathbf{x}^*$ ist gefunden, wenn grad $f(\mathbf{x}^*) = (0\ ,\ 0)$ ist. Das ist näherungsweise erreicht, wenn in Zeile 11 die *Norm* des Gradienten die Konvergenzschranke erreicht oder unterschreitet. Das Programm wird dann beendet (Zeilen 11 – 13).

Ist $\mathbf{v} = (v_1, v_2, \ ... \ , v_n)$ ein beliebiger Vektor, so errechnet MATLAB dessen *Norm* gemäß der Formel $|\mathbf{v}| = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$. Für *ebene* Vektoren $\mathbf{v} = (v_1, v_2)$ und *räumliche* Vektoren $\mathbf{v} = (v_1, v_2, v_3)$ ist das deren *geometrische Länge* gemäß des Satzes von Pythagoras. Deswegen heißt $|\mathbf{v}|$ "Pythagoreische" oder "Euklidische" Norm.

Zusammenfassung:
Gestartet wird das script-file **Steepest_Descent_Verfahren.** Dieses file ruft das function-file **Gradienten_Verfahren** auf. Das function-file **Gradienten_Verfahren** ruft function-file **Dichotome_2D_Minimumsuche** auf. Die Ausgaben sind der Contour-Plot



sowie die Ergebnisse für das Minimum im Command-Window:

```
Das Minimum liegt bei (x_min,y_min) = (0.4999,0.4999)
Der Minimalwert von f beträgt f_min = -0.2500
Es wurden 23 Iterationen benötigt
```

Verbindet man im Contourplot Punkt 0 mit Punkt 1 und dann Punkt 1 mit Punkt 2 usw. , so fällt auf , dass die Strecken zueinander senkrecht stehen (siehe Contour-Plot).

Beweis (für den interessierten Leser und kann daher auch übersprungen werden):
Offenbar weist die Strecke von Punkt 0 zu Punkt 1 in Richtung von $\mathbf{s_0}$, die Strecke von Punkt 1 zu Punkt 2 in Richtung von $\mathbf{s_1}$ usw. Die Bestimmung von $\mathbf{x_{n+1}}$ aus $\mathbf{x_n}$ erfolgt mittels $\mathbf{x_{n+1}} = \mathbf{x_n} + t_{\mathbf{n}}\ \mathbf{s_n}$.
Dabei ist $t_{\mathbf{n}}$ der Parameter, der ausgehend von $\mathbf{x_n}$ in Richtung $\mathbf{s_n}$ zum neuen Vektor $\mathbf{x_{n+1}}$ führt, wo der Graph von f einen tiefsten Punkt hat, die Steigung dort also gleich 0 ist.
Mit $q(t) \underset{def.}{=} f(x_n + ts_n)$ gilt an der Stelle $t = t_n$ somit

$$q'(t_n) = \frac{dq}{dt}(t_n) = \frac{d}{dt} f(x_n + ts_n)\Big|_{t=t_n} = \operatorname{grad} f(x_n + t_n s_n)\cdot s_n = \operatorname{grad} f(x_{n+1})\cdot s_n \underset{(4.2)}{=} -s_{n+1}s_n = 0 .$$

Es ist also $s_{n+1}s_n = 0$ im Sinne des Skalarprodunkts, d.h. es ist $s_n \perp s_{n+1}$.

Anmerkungen:

Zum "Feintuning" des Steepest-Descent-Verfahrens kann die "Kontrollgröße" $\gamma_n \underset{def.}{=} \frac{s_n\ s_{n+1}}{|s_n|^2}$ mitgeführt werden, die bei jedem Iterationsschritt *nahe bei Null* bleiben sollte.
Je ähnlicher die Niveaulinien einer konzentrischen *Kreisschar* sind, desto effizienter ist das Steepest-Descent_Verfahren.

Vorteile des Steepest-Descent-Verfahrens:
Einfach zu programmieren, Konvergenz unter schwachen Forderungen an die Zielfunktion garantiert.
Nachteile des Steepest-Descent-Verfahrens:
Langsame (lineare Konvergenz). Im obigen Beispiel waren 23 Iterationen nötig. Die Ableitungsfunktionen von f sind zur Berechnung des Gradienten erforderlich.

IV.4.2 Ableitungsfreie Verfahren: Der Hooke-Jeeves-Algorithmus

Bei *ableitungsfreien* Verfahren besteht jeder Iterationsschritt aus zwei Schritten:
1. Schritt: Mustersuche ("pattern search", "Tastphase") zum Auffinden einer geeigneten Suchrichtung.
2. Schritt: Minimierungsschritt in der ermittelten Suchrichtung (Extrapolationsschritt).
Die Verfahren unterscheiden sich hinsichtlich der Ausgestaltung dieser Phasen.

Als Beispiel sei hier das *Hooke-Jeeves-Verfahren* (1960) beschrieben.

Gegeben sei die reellwertige *Zielfunktion* $f(\mathbf{x}) = f(x_1, x_2, \ \dots\ , x_n) \in \mathbb{R}$, d.h. f hänge von n Einflussgrößen ab. Es sei

$\mathbf{x_0}$ ein beliebiger Startvektor

h die vorgegebene (kleine) Schrittweite und

$\mathbf{x_k}$ das k-te Folgenglied der Iterationsfolge $\mathbf{x_0}$, $\mathbf{x_1}$, $\mathbf{x_2}$, ...

Wie wird das nächste Folgenglied $\mathbf{x_{k+1}}$ erzeugt?

1. Schritt: Mustersuche (Tastphase)

Weise $\mathbf{x_k}$ der *Hilfsvariablen* $\mathbf{z}$ zu (d.h. definiere $\mathbf{z} = \mathbf{x_k}$) und bilde die Punkte

$$(5) \qquad \begin{cases} \mathbf{z_+} = \mathbf{z} + h\mathbf{e}_1 \\ \mathbf{z_-} = \mathbf{z} - h\mathbf{e}_1 \end{cases} \quad \text{mit } \mathbf{e}_1 = (1\ \underbrace{0\ \dots\ 0}_{n-1 \text{ mal}})$$

sowie die Funktionswerte $f_- = f(\mathbf{z}_-)$, $f_0 = f(\mathbf{z})$ und $f_+ = f(\mathbf{z}_+)$.

$\mathbf{e}_1$ ist der *n*-dimensionale Einheitsvektor in Richtung der positiven x_1-Achse des *n*-dimensionalen $x_1 - x_2 - \ ... \ - x_n$ – Koordinatensystems, worauf f definiert ist.

Anschaulich werden hier ausgehend von $\mathbf{z} = \mathbf{x_k}$ die Funktionswerte von f mit der Schrittweite h in der positiven und negativen x_1-Achse ermittelt ("abgetastet") und miteinander verglichen. Von den 3 obigen Funktionswerten nehme man dann den *kleinsten* und aktualisiere f_0 und $\mathbf{z}$ wie folgt:

$$\begin{cases} \min\{f_-, f_0, f_+\} \text{ bezeichne das "neue" } f_0 \\ \text{Das zu } \min\{f_-, f_0, f_+\} \text{ gehörige Argument bezeichne das "neue" } \mathbf{z}. \end{cases}$$

Genauer: Ist z.B. $f_+ = \min\{f_-, f_0, f_+\}$, dann ist $\mathbf{z}_+$ das neue $\mathbf{z}$.

Mit dem *aktualisierten* $\mathbf{z}$ kehre man zurück zu (5) und ersetze dort $\mathbf{e}_1$ durch $\mathbf{e}_2 = (0\ 1\ \underbrace{0\ ...\ 0}_{n-2 \text{ mal}})$ und wiederhole das Vorgehen. Die "Abtastung" geschieht jetzt mit der Schrittweite h längs der x_2-Achse, was zu einer *weiteren Aktualisierung* von $\mathbf{z}$ und f_0 führt.

Entprechend verfahre man mit allen übrigen Achsenrichtungen.

Sind alle Koordinatenrichtungen abgearbeitet, so liegt ein Nachbarpunkt $\mathbf{z}$ von $\mathbf{x_k}$ vor, der einen "im bestmöglichen Sinne" kleineren Funktionswert f_0 liefert als $f(\mathbf{x_k})$: $f_0 = f(\mathbf{z}) \le f(\mathbf{x_k})$.

Man beachte aber, dass im *ungünstigsten Fall* das Ergebnis $f(\mathbf{z}) = f(\mathbf{x_k})$ und somit evtl. auch $\mathbf{z} = \mathbf{x_k}$ möglich sein kann. Daher

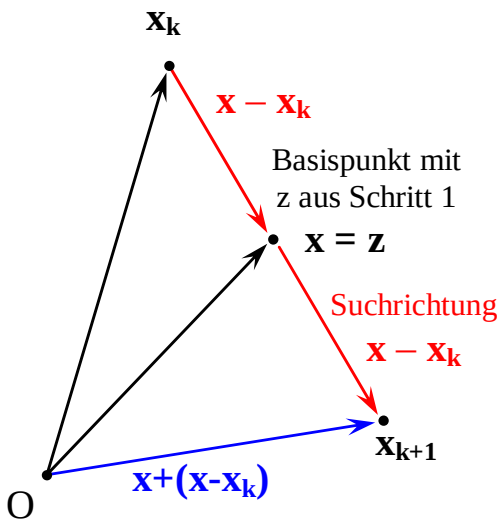
Abfrage:

Ist trotzallem das aktuelle $\mathbf{z} = \mathbf{x_k}$?

Falls ja, so wiederhole man das obige Vorgehen mit *halbierter* Schrittweite $\frac{h}{2}$.

Falls nein, so setze man den "Basispunkt" $\mathbf{x} = \mathbf{z}$ und starte den folgenden Schritt:

2. Schritt: Extrapolationsschritt



Nach dem obigen Vorgehen ist der Basispunkt $\mathbf{x} = \mathbf{z}$ derjenige, der im bestmöglichen Sinn einen kleineren Funktionswert hat, als der Punkt $\mathbf{x_k}$. In diesem Sinne ist somit $\mathbf{x} - \mathbf{x_k}$ (dargestellt als Pfeil linkerhand) die "optimale Richtung", welche zu einer bestmöglichen Verkleinerung des Funktionswertes führt. Das ist das Ergebnis im 1. Schritt.

Im 2. Schritt benutze man diese "Suchrichtung" zum Erhalt des Folgengliedes $\mathbf{x_{k+1}}$:

Ausgehend vom Basispunkt $\mathbf{x}$ definiere man ein *abermals neues* $\mathbf{z}$ vermöge

(6) $\mathbf{z} \underset{def.}{=} \mathbf{x} + (\mathbf{x} - \mathbf{x_k})$ und setze

(7) $f_1 \underset{def.}{=} f(\mathbf{z})$

Falls f_1 das Ergebnis f_0 im 1. Schritt *nochmals unterbietet* (wenn also $f_1 < f_0$ gilt),

so aktualisiere man endgültig: $f_0 \underset{def.}{=} f_1$ sowie $\mathbf{z}$ gemäß (6) und setze $\mathbf{x_{k+1}} \underset{def.}{=} \mathbf{z}$.

Falls nicht, bleiben die Ergebnisse aus Schritt 1 aktuell.

Man definiere dann $\mathbf{x_{k+1}} \underset{def.}{=} \mathbf{x}$ als obigen Basispunkt und verwende das Ergebnis f_0 aus dem 1. Schritt.

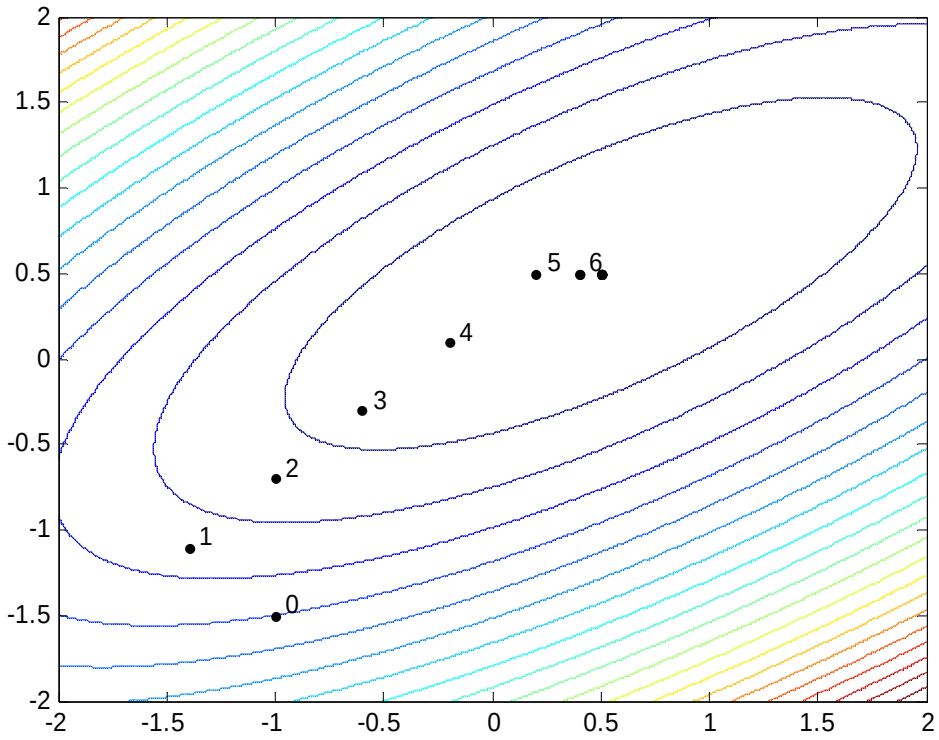
Zur Umsetzung des Hooke-Jeeves-Verfahrens mit MATLAB verwenden wir zum Vergleich mit dem Steepest-Descent-Verfahren wieder die Funktion $f(x,y) = x^2 - 2xy + 2y^2 - y$, $x \in [-1,1]$, $y \in [-1,1]$.
Ein script-file (Name: **Hooke_Jeeves_Verfahren_Start**) stellt für das Hooke-Jeeves-Verfahren die erforderlichen Daten bereit und erzeugt einen Contour-Plot von f. Ableitungen von f werden nicht benötigt. Das script-file ruft das function-file **Hooke_Jeeves_Verfahren** auf, welches die Iterationsfolge $\mathbf{x_0}$, $\mathbf{x_1}$, $\mathbf{x_2}$, ... und daraus x_min und f_min bestimmt und in den Contour-Plot einbettet.
Die Ergebnisse x_min und f_min für das Minimum werden vom script-file im Command-Window ausgegeben.
Zunächst der Code für das script-file:

```
 1-  [x,y]=meshgrid(-2:0.01:2,-2:0.01:2);
 2-  z=x.^2-2*x.*y+2*y.^2-y;
 3-  C=contour(x,y,z,20);
 4-  clabel(C,'manual')
 5-  xo=[-1,-1.5];h=0.2;k_max=100;Eps=1e-4;
 6-  f=@(x)x(1)^2-2*x(1)*x(2)+2*x(2)^2-x(2);
 7-  [x_min,f_min,k]=Hooke_Jeeves_Verfahren(2,xo,h,Eps,k_max,f);
 8-  fprintf('Das Minimum liegt bei (x_min,y_min) = (%.4f,%.4f)\n',x_min);
 9-  fprintf('Der Minimalwert von f beträgt f_min = %.4f\n',f_min);
10-  fprintf('Es wurden %u Iterationen benötigt',k);
```

Das Hooke-Jeeves-Verfahren ist eng an die obige Beschreibung angelehnt und muss daher nicht ausführlich kommentiert werden. Beachte die Variablennamen zp = z + , zm = z _ , fp = f + und fm = f_ .

```
 1-  function [x_min,f_min,k]=Hooke_Jeeves_Verfahren(n,xo,h,Eps,k_max,f)
 2-  E=eye(n);     %E ist die n-reihige Einheits-Matrix
 3-  for k=1:k_max
 4-     hold on;                      %Iterationspunkte in den
 5-     plot(xo(1),xo(2),'black.');   %Contour-Plot einbetten.
 6-     if k<=7 %Die ersten 7 Iterationspunkte mit Indizes kennzeichnen.
 7-         text(xo(1)+0.05,xo(2)+0.05,int2str(k-1));
 8-     end
 9-     hold off;
10-     z=xo;fo=f(z);      %1.Schritt: Tast-Phase
11-     for j=1:n                          %n = Variablenanzahl von f.
12-         zm=z-h*E(j,:);fm=f(zm);        %E(j,:)=Einheitsvektor e_j.
13-         zp=z+h*E(j,:);fp=f(zp);        %Beachte: Hier wird n = 2
14-         if fm<=min(fo,fp)              %vom script-file übernommen.
15-            z=zm;fo=fm;
16-         end
17-         if fp<=min(fo,fm)
18-            z=zp;fo=fp;
19-         end
20-     end
21-     if z==xo           %Abfrage zur Halbierung der Schrittweite h.
22-         h=h/2;
23-         if h<=Eps      %Gewünschte Ergebnisgenauigkeit erreicht,
24-             break      %daher Programmabbruch.
25-         end
26-     else               %2.Schritt: Extrapolation.
27-         x=z;z=x+(x-xo);fz=f(z);
28-         if fz<fo
29-             xo=z;fo=fz;
30-         else
31-             xo=x;
32-         end
33-     end
34-  end
35-  x_min=xo;f_min=fo;    %Ergebnis zur Ausgabe im script-file.
```

Wird das Hooke-Jeeves-Verfahren vom script-file aus gestartet, so erhalten wir folgende Ausgabe:



Im Command-Window erfolgt die Ausgabe:

```
Das Minimum liegt bei (x_min,y_min) = (0.5000,0.5000)
Der Minimalwert von f beträgt f_min = -0.2500
Es wurden 18 Iterationen benötigt
```

Das Ergebnis wird im Vergleich zum Steepest-Descent-Verfahren (23 Iterationen) hier bereits nach 18 Iterationen erreicht.

Vorteile des Hooke-Jeeves-Verfahrens: Robustes und effizientes Verfahren.
Nachteile des Hooke-Jeeves-Verfahrens: Keine theoretisch begründete Konvergenzaussage vorhanden.

Schlussbemerkungen:
Weitere bekannte ableitungsfreie Verfahren sind das *Nelder-Mead-Verfahren* (1965) , das *Rosenbrock-Verfahren* (1960) sowie das *Powell-Verfahren* (1964).