Matlab–Kompendium

zur Numerischen Mathematik

Stand:

20. Januar 2009

Vorwort

Dieses Manuskript ist für die TeilnehmerInnen der Vorlesung

"Numerische Mathematik"

an der Universität Göttingen gedacht. Es nimmt Bezug auf das Buch [1]

R. Schaback, H. Wendland: Numerische Mathematik,
Springer–Verlag, 5. Auflage

und bringt die passenden MATLAB–Beispiele nebst einer Einführung in MATLAB. Der Text ist zur Zeit noch sehr rudimentär und wird sich im Laufe des Semesters allmählich zu etwas Sinnvollem mausern.

R. Schaback, 20. Januar 2009

Inhaltsverzeichnis

1 Einführung

Das erste Kapitel des Buches enthält ein Beispiel zur Wärmeleitungsgleichung, das sich natürlich nicht als MATLAB–Einstieg eignet. Wer schon etwas von MATLAB versteht, kann sich das im Abschnitt 1.4 ansehen.

1.1 Einführung in MATLAB

Dieser Textteil ist eine Adaptation eines Bausteins aus der MafIA–Vorlesung.

Hier wird eine kurze Anleitung zur Benutzung von MATLAB gegeben, wobei die praktische Handhabung auf den Göttinger Rechnern im Vordergrund steht.

> **Es wird dringend empfohlen, den folgenden Text direkt am Rechner durchzuarbeiten und die MATLAB–Kommandos sofort auszuprobieren!**

1.1.1 Start von MATLAB

Mit der UNIX–Kommandozeile

```
matlab &
```

auf einem der lokalen Rechner ruft man MATLAB auf. Nach einem schnell verschwindenden Begrüßungsfenster sieht man ein Arbeitsfenster, das u.a. ein *Command Window* enthält, in dem man durch Direkteingabe Kommandos ausführen kann. Man kann aber über die üblichen Menüeinträge (*File/Open*) auch vorgefertigte Kommandosequenzen im *Command Window* ausführen, die man als *m–files* bezeichnet und mit jedem beliebigen ASCII–Texteditor bearbeiten kann. Die auch bei anderen Programmen üblichen Icons zum Erzeugen, Laden und Abspeichern von Dateien sind vorhanden, um eigene *m*-Files zu erstellen.

Hier ist ein simples Beispiel[1], das im folgenden kommentiert werden soll. Man kann die Befehle einzeln (ohne den mit % beginnenden Kommentarteil) in das jeweilige Kommandofenster eingeben, um zu sehen, was passiert.

```
clear all;        % bereinigt die komplette Vorgeschichte
A=[1 0.2 ; -0.3 4]% eine Matrix mit 2 Zeilen und Spalten
                  % ein Semikolon faengt eine neue Zeile an
```

[1] `http://www.num.math.uni-goettingen.de/schaback/teaching/texte/NuMath/matlab/Kapitel01/erst`

```
x=[5; -0.6]       % ein Vektor als Spaltenvektor, 2 Komponenten
z=A*x             % Matrix mal Vektor
B=A*A             % Matrix mal Matrix
rank(A)           % Dimension von Zeilen/Spaltenraum
y=1:7             % ein Folgenstueck als Zeile
z=(1:7)'          % dito als Spalte, transponiert
C=0.1*[1:7;2:8;3:9]% eine 3x7-Matrix, skalar multipliziert
C'*C              % liefert eine 7x7-Matrix
C*C'              % liefert eine 3x3-Matrix
D=ones(2,3)       % Matrix mit Einsen, 2x3
E=eye(5)          % Einheitsmatrix, 5x5
F=exp(-z)         % Operationen bilden Matrizen auf Matrizen ab
G=exp(-C)         % und werden komponentenweise ausgerechnet
G(:,3)            % dritte Spalte
G(2,:)            % zweite Zeile
u=A\x             % loest Gleichung A*u=x
x-A*u             % Test
C+C'              % Fehlermeldung
```

Wie alle anderen Systeme dieser Art arbeitet auch MATLAB als dynamischer Interpreter, d.h. das "Wissen" von MATLAB und die Nutzung des internen Speichers hängt von der Vorgeschichte ab. Die Zuweisung

```
A=B
```

weist dem Bezeichner `A` die Bedeutung zu, die vorher dem Bezeichner `B` zukam. Die vorherige Bedeutung des Bezeichners `A` ist verloren. Eine weitere Zuweisung

```
A=C
```

überschreibt dies durch die Bedeutung des Bezeichners `C`. Das im Beispiel zuerst auftretende Kommando

```
clear all;
```

bereinigt die komplette Vorgeschichte und löscht alle Bedeutungen von Bezeichnern. Das ist zu Beginn eines neuen und unabhängigen *m–files* sinnvoll, damit auch der bisher reservierte Speicher (*workspace* in MATLAB) freigegeben wird. Man kann sich übrigens in einem über das Hauptfenster aufrufbaren Teilfenster stets den aktuellen *workspace* und seine Nutzung ansehen.

Die *Wertzuweisung* durch `=` ist eine unsymmetrische Operation, weil zuerst der Wert der rechten Seite bestimmt wird und dann dem Bezeichner auf

der linken Seite zugewiesen wird. Es ist ein Notations–Verbrechen, für eine unsymmetrische Operation ein symmetrisches Symbol zu verwenden, aber das können wir nicht ändern. In PASCAL war das durch `:=` besser gelöst.

Die Kommandostruktur von MATLAB ist zeilenorientiert (ein Zeile = ein Kommando), wobei man mit ... auf eine Verlängerungszeile gehen kann, wenn nötig. Wenn man ein Kommando mit einem Semikolon abschließt, wird die Ausgabe unterdrückt. Das ist bei großen Matrizen und Vektoren **lebenswichtig**.

Auch wichtig ist die *Hilfefunktion*, die man über das Menü des Hauptfensters aufrufen kann. Sie erlaubt es, komfortabel nach bestimmten Wörtern (nicht nur MATLAB–Kommandos) in der riesigen Dokumentation zu suchen. Man probiere das mal aus, indem man nach `clear` oder `Gauss` sucht.

1.1.2 Erzeugen von Matrizen

In MATLAB sind alle "normalen" Objekte Matrizen von `double`–Zahlen, wobei Vektoren als Matrizen mit einer Spalte, n–Tupel als Matrizen mit einer Zeile und Skalare als 1×1–Matrizen aufgefaßt werden. Im Normalfall arbeiten alle Operationen auf **kompletten Matrizen**. Sonderfälle muß man speziell behandeln. Man tut gut daran, dieses Grundkonzept nicht künstlich zu verwässern, indem man statt mit Vektoren und Matrizen zu arbeiten, auf deren Komponenten zurückgeht. Schleifenprogrammierung ist möglich, sollte aber wie die Pest vermieden werden, wenn sie sich nicht auf komplette Matrizen bezieht.

Kleine Matrizen kann man in MATLAB direkt eingeben, indem man z.B. die Matrix

$$A = \begin{pmatrix} 1 & 0.2 \\ -0.3 & 4 \end{pmatrix}$$

als

```
A=[1 0.2 ; -0.3 4]
```

spezifiziert und sofort ausgibt (kein Semikolon als Abschluß). Die Eingabe geschieht innerhalb der Klammern [ ] zeilenweise mit Leerzeichen als Trennzeichen, wobei das Semikolon eine neue Zeile beginnt. Dann ist auch klar, was

```
x=[5; -0.6]     % ein Vektor als Spaltenvektor, 2 Komponenten
```

bewirkt. Ganz gemäß der MATLAB–Philosophie kann man, wenn die Größen stimmen, in dieser Klammernotation auch Matrizen einsetzen, um z,B. `[A -A]` oder `[A ; 7 -2.3]` zu bilden.

Also:

- Elemente in derselben Zeile werden nur durch Leerzeichen getrennt,
- der Übergang zu einer neuen Zeile beginnt nach einem Semikolon.

Zum Erzeugen von Standardmatrizen gibt es die Befehle

```
zeros(m,n)              % Matrix mit Nullen, m x n
rand(m,n)               % m x n Matrix mit Zufallszahlen in [0,1]
ones(m,n)               % Matrix mit Einsen, m x n
eye(n)                  % Einheitsmatrix, n x n
```

und man kann natürlich auch größere Matrizen aus Dateien einlesen, indem man das Kommando `load` verwendet (man gebe im Kommandofenster `help load` ein, um die genaue Syntax zu sehen).

Einer der wichtigsten Operatoren in MATLAB ist der Doppelpunkt oder *colon*–Operator. Steht er zwischen Skalaren, so erzeugt er Zeilenvektoren von Werten:

```
3:7                     % liefert [3 4 5 6 7]
4:2:9                   % liefert [4 6 8]
0:0.15:1                % liefert [0 0.15 0.3 0.45 0.6 0.75 0.9]
```

Die allgemeine Form `a:h:b` erzeugt einen Zeilenvektor, der von `a` nach `b` geht und dabei versucht, das Inkrement `h` zu verwenden, wobei `h`= 1 angenommen wird, wenn `h` fehlt. Es werden Zahlen `a`,`a+h`, `a+2h` ... erzeugt, aber `b` wird nicht überschritten. Das funktioniert sinngemäß auch für negative `h`. Diese Notation ist extrem hilfreich zur Erzeugung von Wertetabellen, denn Funktionen wie `sin` arbeiten immer komponentenweise auf kompletten Vektoren oder Matrizen:

```
sin(0:0.15:1)           % liefert die Sinuswerte auf
                        % [0 0.15 0.3 0.45 0.6 0.75 0.9]
```

Und wer gerne etwas Graphisches sehen möchte, sollte

```
x=0:0.01:2*pi;
plot(x,sin(x),x,cos(x))
```

versuchen, aber das Semikolon nicht vergessen.

Eine typische Anwendung ist, im Intervall $[a, b] \subset \mathbb{R}$ genau n Punkte $x_1 = a, \ldots, x_n = b$ gleichmäßig zu verteilen. Nach dem Gesetz des Lattenzauns[1] nimmt man dann die Schrittweite $h = (b - a)/(n - 1)$ und schreibt

```
x=a:(b-a)/(n-1):b
```

wobei man ein gewisses Risiko eingeht, wenn a, b und n "unschöne" Zahlen sind. Denn man kann sich bei Gleitkommarechnung nicht sicher sein, ob die Ausrechnung von $a + (n - 1) * ((b - a)/(n - 1))$ nicht geringfügig größer als b herauskommt, und dann fehlt b in der Punkteliste. In der Regel wird aber durch "Abhacken" gerundet, und dann ist der berechnete Wert eher etwas zu klein.

Man kann die *colon*–Notation auch sehr gut auf Indexbereiche anwenden. Zum Beispiel kann man die obere linke 2×3–Teilmatrix aus einer Matrix `A` herausholen und nach `B` speichern mit

```
B=A(1:2,1:3)
```

Aber der Doppelpunkt kann auch als Platzhalter mit der Bedeutung "für alle" stehen. Ist etwa `A` eine $m \times n$–Matrix in MATLAB, so ist `A(:,3)` die dritte Spalte und `A(2,:)` die zweite Zeile von `A`. Und dann ist `A(4:2:9,:)` natürlich die Teilmatrix von `A`, die aus den Zeilen 4, 6 und 8 besteht, denn wir haben oben schon gesehen, daß `4:2:9` die Indexliste `[4 6 8]` produziert.

Ferner gibt es das sehr nützliche Wörtchen `end` mit der Bedeutung "bis zum Ende", mit dem man Reste von Matrizen oder Vektoren bilden kann, z.B. `vec(3:end)` oder `mat(2:end,2:end)`, egal wie groß sie sind.

Kennt man die wahre Größe einer $m \times n$ Matrix `A` nicht (z.B. wenn sie aus einer Datei eingelesen wurde), so kann man sie sich mit

```
[m n]=size(A)
```

holen, denn nach diesem Befehl gibt `m` die Zeilenzahl und `n` die Spaltenzahl an. Will man z.B. eine Nullmatrix `B` bauen, die dieselbe Größe wie `A` hat, so kann man einfach

```
B=zeros(size(A))
```

sagen.

[1] Ein Zaun mit n Latten hat $n - 1$ Zwischenräume.

1.1.3 Verarbeiten von Matrizen

Die drei Kommandos

```
z=A*x           % Matrix mal Vektor
B=A*A           % Matrix mal Matrix
rank(A)         % Dimension von Zeilen/Spaltenraum
```

zeigen, wie einfach man nun in MATLAB mit Matrizen und Vektoren umgeht. Dabei kann man die üblichen Operationen $+, -, *$ zwischen Matrizen verwenden, muß aber aufpassen, ob die Operationen bei den vorliegenden Matrixgrößen auch ausführbar sind, sonst erfolgt eine Fehlermeldung. Es gibt allerdings eine sehr praktische Sonderregelung, wenn einer der Operanden skalar ist. Dann wird die Operation als komponentenweise Skalaroperation ausgeführt. So kann man Konstanten zu Matrizen addieren oder Matrizen mit festen Faktoren komponentenweise multiplizieren. Der Zugriff auf Matrixelemente erfolgt über Indizes in runden Klammern, z.B. mit `A(1,2)` auf A_{12}, sollte aber nur im absoluten Notfall benutzt werden.

Die Transposition einer Matrix wird durch ein nachgestelltes Apostroph bewirkt, z.B. in

```
C=0.1*[1:7;2:8;3:9]  % eine 3x7-Matrix, skalar multipliziert
C'*C                 % liefert eine 7x7-Matrix
C*C'                 % liefert eine 3x3-Matrix
```

Manchmal sind auch noch die **komponentenweisen** Operationen von MATLAB nützlich. Wenn `A`$= (a_{jk})$ und `B`$= (b_{jk})$ zwei Matrizen gleicher Größe sind, so besteht `A.*B` aus der Matrix $(a_{jk} \cdot b_{jk})$. Analog ist `A./B` definiert.

Aber die Stärke von MATLAB liegt in der einfachen Verfügbarkeit höherer Operationen, die das Lösen von Gleichungssystemen, das Bestimmen des Rangs oder des Kerns von Matrizen erlauben. Der Rang von `A` wird mit `rank(A)` abgerufen, während eine Orthonormalbasis des Kerns mit `null(A)` und des Bildes mit `orth(A)` produziert wird. Hier fließt die rechengenauigkeitsbedingte Unsicherheit des Rangentscheids ein (vgl. Abschnitt 4.2 auf Seite 63 des Buches). Die Lösung x eines linearen Gleichungssystems $A \cdot x = b$ bekommt man einfach mit

```
x=A\b                % löst Ax=b
```

unter unsichtbarer Verwendung stabiler Lösungsverfahren, sofern die Voraussetzungen für die Lösbarkeit gegeben sind. Aber das werden wir im Folgenden noch im Detail ansehen.

Wir unterbrechen den Einführungskurs in MATLAB hier, weil wir einfache Probleme schon behandeln können. Schwierigeres kommt später.

1.2 Fehler

Wir wenden uns jetzt dem ersten Kapitel des Buches zu und lernen alles bis Seite 9 Mitte, inklusive dem relativen und absoluten Fehler. Für das Beispiel 1.9 brauchen wir noch etwas mehr MATLAB:

1.2.1 Schleifen in MATLAB

Wir üben hier die Programmierung einfacher Schleifen in MATLAB, obwohl wir vereinbart haben, Schleifen wie die Pest zu vermeiden. Die Konstruktion

```
for n=a:h:b
```

> ... einige Befehle, die für alle Elemente `n`
> des durch `a:h:b` definierten
> Zeilenvektors ausgeführt werden sollen...

```
end
```

ist alles, was wir fürs erste brauchen.

1.2.2 Beispiel 1.9

Wir illustrieren damit das Beispiel 1.9, mit einer ziemlich naiven Programmierung. Es geht um die näherungsweise Berechnung der Exponentialfunktion durch ihre auf $n+1$ Terme abgeschnittene Taylorreihe $P_n(x)$. Wir wenden Schleifen zweimal an: erst für die n in $P_n(x)$ und dann für die Summation über j in

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

Wir lassen n beispielsweise von 0 bis 25 laufen und verwenden also `for n=0:1:25` oder `for n=0:25`. Danach folgt ein `for j=0:n`, und beide brauchen je ein `end`. Da wir das Ganze plotten wollen, vereinbaren wir einen Abszissenbereich `x` und berechnen darauf die Exponentialfunktion. Für die Berechnung von $P_n(x)$ als Summe brauchen wir einen Vektor `sum`, der dieselbe Länge wie `x` hat und alle Summanden $x^j/j!$ aufsammelt. Wir plotten den relativen Fehler, aber durch `hold on` sagen wir MATLAB, dass der folgende Plot in dieselbe Zeichnung gesetzt werden soll.

Hier ist das komplette Programm:

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

Man achte genau darauf, wo hier "gepunktete" Operationen stehen. Der Zeilenvektor `x` aus Abszissenwerten zwischen -10 und 10 wird durch `x.^j` **elementweise** zur j–ten Potenz erhoben. Ergebnis ist ein Vektor, der durch $j!$ =`factorial(j)` elementweise dividiert und dann auf die Summe aufgeschlagen wird. Ebenso beim relativen Fehler: weil `sum` und `expx` Vektoren sind, die elementweise durch einander dividiert werden sollen, müssen wir `./` statt nur `/` verwenden.

Die Floskel

```
Summe=Summe+Inkrement
```

ist typisch für das Aufsummieren. Es wird die "alte" Summe genommen, das Inkrement aufaddiert und dann der Summenvariablen ales neuer Wert zugewiesen, und das Spiel kann von neuem beginnen.

Das Ergebnis der Rechnung ist Abbildung 1. Warum werden die Ergebnisse für negative x groß? Warum sehen sie für positive x klein aus? Sind sie wirklich klein? Oder kann man sie nur nicht sehen? Was passiert, wenn man sich nur positive x ansieht? Man verändere das Programm an den richtigen Stellen, um sich die Lage genauer anzusehen.

1.2.3 Beispiel 1.22

Jetzt ein Beispiel zur Auslöschung, analog zu Beispiel 1.22 auf Seite 15. Dieses Programm ist sehr primitiv, es gibt noch nicht mal Vektoren. Der einzige "Trick" ist `format long g` um dafür zu sorgen, daß die Ausgabe nicht schreibtechnisch verkürzt wird. Dieser Befehl bewirkt kein genaueres

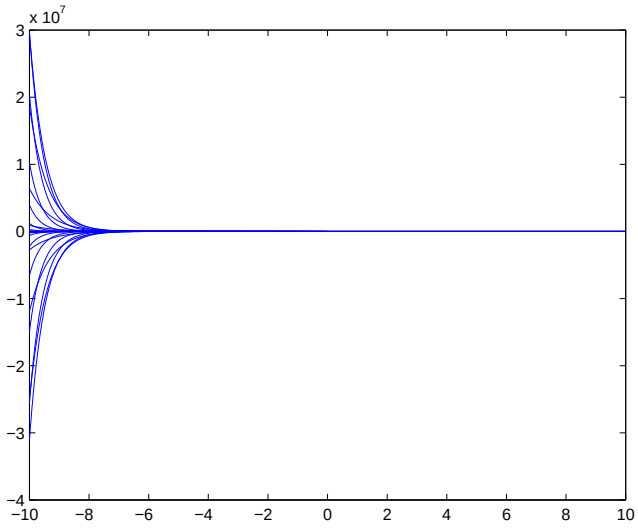

Abbildung 1: Relativer Fehler bei Reihen–Partialsummen

Rechnen, sondern beeinflußt nur die Ausgabe. Ferner verwenden wir den Absolutbetrag $|x|$ =`abs(x)`, den man in MATLAB auch auf Vektoren und Matrizen anwenden kann, wobei er wie üblich komponentenweise wirkt.

```
% Ausloeschung, mit 6 Dezimalstellen
clear all;
format long g % sorgt fuer lange Ausgabezahlen
x=0.344152
xwahr=0.344152*1.0000001 % das ergibt 0.01% relativen Fehler
relfx=abs(xwahr-x)/xwahr
y=0.344135
z=x-y                %
zwahr=xwahr-y                %
relfz=abs(z-zwahr)/abs(zwahr) % relativer Fehler von z
```

Die Ausgabe ist (etwas geschönt)

$$\begin{aligned} x &= 0.344152 \\ xwahr &= 0.3441864152 \\ relfx &= 9.99900009998154e-05 \\ y &= 0.344135 \\ z &= 1.69999999999892e-05 \\ zwahr &= 5.14151999999601e-05 \\ relfz &= 0.669358477648586 \end{aligned}$$

d.h. der relative Fehler verschlechtert sich durch Auslöschung von $1.0e-6$ auf 0.669.

Ein weiteres Beispiel zur Auslöschung.

```
% Ausloeschung
format long g  % das sorgt fuer lange Ausgabe...
x=1.00000000012345 % Wahrer Wert soll 1.0+1.0e-10 sein
y=1.00000000001234 % Wahrer Wert soll 1 sein
% Dann haben x und y einen relativen Fehler von etwa 1.0e-11
z=x-y              % Wahre Differenz ist 1.0e-10
relfx=(x-1-1.0e-10)/(1+1.0e-10) % relativer Fehler von x
relfy=(y-1)/1 % relativer Fehler von y
relfz=abs(1.0e-10-z)/1.0e-10 % relativer Fehler von z
```

mit der Ausgabe

$$\begin{aligned} x &= 1.00000000012345 \\ y &= 1.00000000001234 \\ z &= 1.11110010081461e-10 \\ relfx &= 2.34499169532197e-11 \\ relfy &= 1.23399068741037e-11 \\ relfz &= 0.11110010081461 \end{aligned}$$

Hier steigt der relative Fehler noch dramatischer.

Nach diesem Strickmuster kann man leicht eigene Beispiele bauen.

1.2.4 Maschinengenauigkeit

Die Maschinengenauigkeit ist in MATLAB als `eps` abrufbar. Das ist sehr nützlich, wenn man Abfragen auf $x \neq 0$ ausführen will. In MATLAB wäre dann

`if abs(x)>eps` usw.

angebracht, aber auch das ist oft zu knapp bemessen, wenn x das Ergebnis umfangreicher Rechnungen ist. Man setzt stattdessen in der Praxis ein experimentell gefundenes Vielfaches von `eps` ein, aber die Wahl des Faktors erfordert Erfahrung und Vertrautsein mit dem jeweiligen Algorithmus.

1.2.5 Beispiel 1.23

Zum Vergleich von symmetrischem und unsymmetrischem Differenzenquotienten dient das folgende Programm:

```
% Numerische Differentiation, Tabelle 1.1
clear all
close all
format long g
z=1:20;
h=10.^(-z')
diffq=[h (exp(h)-exp(zeros(size(h))))./h (exp(h)-exp(-h))./(2*h)]
loglog(h,abs(1-diffq(:,2)),h,abs(1-diffq(:,3)))
legend('unsymmetrisch','symmetrisch')
xlabel('h')
ylabel('relativer Fehler')
title('Numerische Differentiation')
```

Es erzeugt die Abbildung 2, die beide numerischen Differentiationen vergleicht. Die numerische Ausgabe dazu ist identisch mit Tabelle 1.1 auf Seite 17 des Buches.

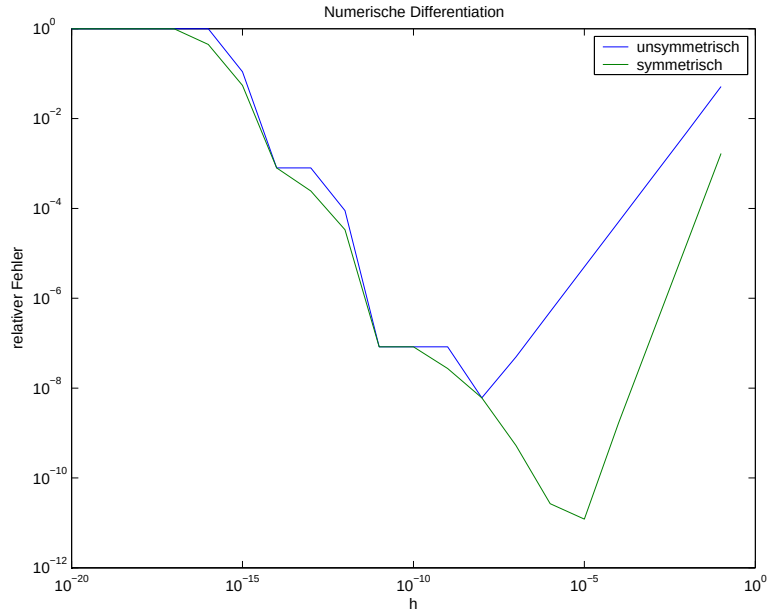

Abbildung 2: Vergleich der beiden Differentiationen

Die Erstellung so eines Programmes kann schrittweise erfolgen. Im Übungsbetrieb wurde dazu folgende Programmieraufgabe gestellt:

Differenzieren Sie $f(x) = \exp(x)$ in $x = 0$ durch den zentralen Differenzenquotienten. Plotten Sie den Approximationsfehler für die Approximation der ersten Ableitung durch den zentralen Differenzenquotienten für die Exponentialfunktion an der Stelle $x = 0$ mit doppelt logarithmischen Achsen und interpetieren sie das Ergebnis.
Tips:

1. Machen Sie das Ganze nicht im Kommandofenster, sondern mit dem MATLAB-Editor in einem m–File.
2. Bauen Sie sich einen Vektor, der eine passende Anzahl von positiven h–Werten $h_1, \ldots, h_n$ enthält.
3. Daraus bauen Sie sich Vektoren, die die Werte $\exp(h_j)$ bzw. $\exp(-h_j)$ enthalten, und dann
4. einen Vektor, der alle zentralen Differenzenquotienten enthält.
5. Berechnen Sie dann den Vektor, der die absoluten Fehler enthält,
6. und plotten Sie ihn gegen den Vektor der h–Werte.
7. Schauen Sie in der Doku nach, wie man einen doppelt logarithmischen Plot macht.
8. Vermutlich werden Sie Gründe haben, Ihre Wahl der h_j noch einmal zu revidieren, um den Effekt klarer herauskommen zu lassen.
9. Schleifen sind verboten.

Eine mögliche Lösung ist

```
clear all;
close all;
h=0.01:0.01:1; % naiver Versuch
h=h.^5% verbesserter Versuch
eplus=exp(h);
eminus=exp(-h);
zentral=(eplus-eminus)./(2*h);
absfehler=abs(zentral-1.0);
% plot(h,absfehler) % naiver Versuch
loglog(h,absfehler) % besserer Versuch
```

mit der Ausgabe in Abbildung 3.

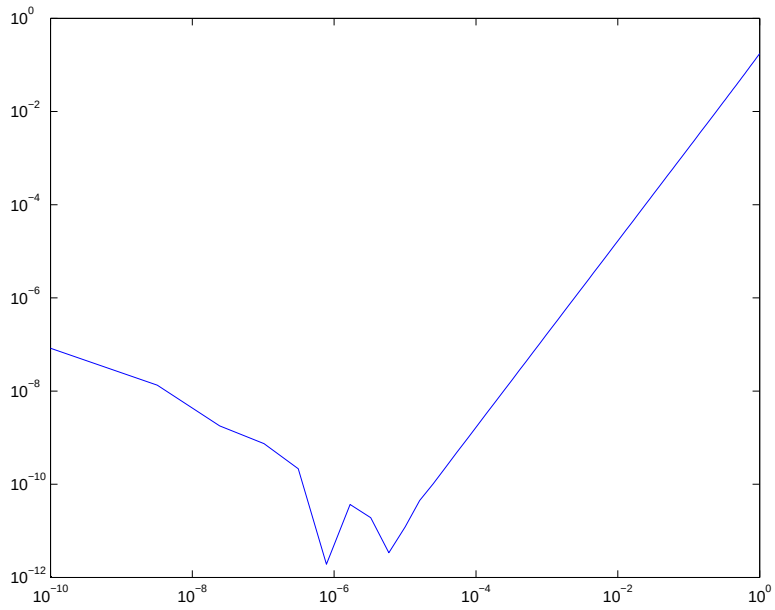

Abbildung 3: Zentraler Differenzenquotient

1.2.6 Beispiel 1.27

Dieses Beispiel zeigt die Instabilität der Standardmethode zur Lösung quadratischer Gleichungen. Unser MATLAB-Programm verwendet die Bezeichnungen p und q des Buches und benutzt die Wurzelfunktion $\sqrt{x}$ =`sqrt(x)` sowie die vordefinierte Konstante π =`pi`.

```
% Beispiel 1.27
clear all;
p=pi;
q=0.0000000000001;  % fuer starke Ausloeschung
ep=1.0e-8;          % wir bringen einen relativen Fehler ep an
pep=p*(1+ep);
qep=q*(1-ep);
% jetzt die instabile Form
x=-p+sqrt(p^2+q)
xep=-pep+sqrt(pep^2+qep)
relf=abs(x-xep)/abs(x)
% und die stabilere
x=q/(p+sqrt(p^2+q))
xep=qep/(pep+sqrt(pep^2+qep))
relf=abs(x-xep)/abs(x)
```

und die geschönte Ausgabe ist

$$\begin{aligned} x &= 1.5987e-014 \\ xep &= 1.5543e-014 \\ relf &= 0.0278 \\ x &= 1.5915e-014 \\ xep &= 1.5915e-014 \\ relf &= 2.0000e-008 \end{aligned}$$

Man sieht, daß die stabile Form den relativen Eingangsfehler nur verdoppelt, während die instabile Form ihn etwa mit 10^6 malnimmt. Auch mit diesem Programm sollte man etwas herumspielen.

1.3 Datenlokalität

Dies bezieht sich auf Seite 22 des Buches. Dort wird empfohlen, alle Rechnungen mit Matrizen und Vektoren so zu organisieren, daß die Rechnung im Speicher *datenlokal* abläuft. In MATLAB wird dies automatisch berücksichtigt, so daß sich entsprechende Effekte nicht massiv auswirken. Bei C/C++–Programmierung ist der Effekt aber dramatisch, und eine Compileroptimierung macht sich sehr bezahlt, denn man kann die Matrixmultiplikation um etwa einen Faktor 10 in der Realzeit beschleunigen, wenn man auf Datenlokalität achtet und die Optimierung einschaltet. Man sehe sich das bei den C/C++–Beispielen an.

1.4 Wärmeleitungsgleichung

Hier ist das m-file, das Abb 1.3 auf Seite 7 des Buches erzeugt.

```
% Lï¿½sung der Wï¿½rmeleitungsgleichung: Bild fï¿½r Buch
clear all;
close all;
x=0:pi/9:pi;   % 10 ï¿½quidistante x-Werte zwischen o und pi
u0=x.*(pi-x);  % Werte der Funktion x*(pi-x) dort
xr=x(2:9);     % Nullen vorn und hinten weglassen
ur=u0(2:9)';   % Nullen vorn und hinten weglassen, als Spalte schreiben
vr=1:8;        % Zeilenvektor aus 1,2,3,4,5,6,7,8
mat=sin(xr'*vr);  % Die Matrix der Sinuswerte
a=mat\ur          % Lï¿½sung des Gleichungssystems. Koeffizienten sind in a.
                  % Ab hier wird das Plotten vorbereitet
xp=0:0.01:pi;     % Plotpunkte fï¿½r den x-Bereich, von 0 bis pi
kx=sin(xp'*vr);   % Die Matrix der Sinuswerte sin(kx_j)
```

```
tt=0:0.01:2;      % Die Plotpunkte fï¿½r den t-Bereich
ex=exp(-tt'*(vr.*vr))*diag(a); % Die Matrix der Exponentialwerte mal den a_k
res=ex*kx';       % Die Resultatmatrix, die zu plotten ist
subplot(2,2,1)
[TA, XA]=meshgrid(tt,xp); % Berechnen der Gitterwerte zum Plotten
surf(TA,XA,res')  % Plotkommando
shading interp    % Fï¿½rben
xlim([0 2]);ylim([0 4]);zlim([0 2]);
xlabel('t');
ylabel('x');
title('Sinusansatz');
clear workspace;
x=0:pi/9:pi   % 10 ï¿½quidistante x-Werte zwischen o und pi
u0=x.*(pi-x)  % Werte der Funktion x*(pi-x) dort
ur=u0(2:9)'   % Weglassen der Nullen vorn und hinten
              % Der DGL-Lï¿½ser wird fï¿½r diese Anfangswerte aufgerufen
[t,y] = ode45(@wlgldglfct,[0 2],ur);  % von t=0 bis t=2, die DGL
              % steht in wlgldglfct.m (siehe dort)
              % Vorbereiten des Plottens
vr=0:pi/9:pi;       % Wir wollen die Nullfunktionen am Rand mitplotten
[m n ]=size(y)
yy=[zeros(m,1)  y  zeros(m,1)]; % deshalb mï¿½ssen wir 2 Nullspalten dazutun
subplot(2,2,2)
[TA, XA]=meshgrid(t,vr);        % Berechnen der Gitterwerte zum Plotten
surf(TA,XA,yy')                 % Plotkommando
title('Differentialgleichungen');
xlabel('t');
ylabel('x');xlim([0 2]);ylim([0 4]);zlim([0 2]);
% Lï¿½sung der Wï¿½rmeleitungsgleichung durch Differenzenverfahren
clear workspace;
x=0:pi/9:pi;   % 10 ï¿½quidistante x-Werte zwischen o und pi
u0=x.*(pi-x);  % Werte der Funktion x*(pi-x) dort
dt=0.07;       % unser delta_t, bei 0.08 klappt es NICHT
t=0:dt:2;      % wir rechnen immer bis 2
[mt, nt]=size(t) % aktuelle Dimension holen
v=zeros(10,nt);  % die Matrix der zu berechnenden Werte
v(:,1)=u0';      % Startwerte in erste Spalte eingesetzt
                 % Hier kommt die Rechenschleife
for k=1:nt-1
    v(2:9,k+1)=v(2:9,k)+(dt*81/(pi*pi))*(v(1:8,k)-2*v(2:9,k)+v(3:10,k));
end
```

```
                        % Ab hier wird geplottet
subplot(2,2,3);
[TA, XA]=meshgrid(t,x); % Berechnen der Gitterwerte zum Plotten
surf(TA,XA,v)           % Plotkommando
title('Differenzen, \delta t=0.07');
xlabel('t');
ylabel('x');xlim([0 2]);ylim([0 4]);zlim([0 2]);
% shading interp
% Lï¿½sung der Wï¿½rmeleitungsgleichung durch Differenzenverfahren
clear workspace;
x=0:pi/9:pi;   % 10 ïï¿½quidistante x-Werte zwischen o und pi
u0=x.*(pi-x);  % Werte der Funktion x*(pi-x) dort
dt=0.08;       % unser delta_t, bei 0.08 klappt es NICHT
t=0:dt:2;      % wir rechnen immer bis 2
[mt, nt]=size(t) % aktuelle Dimension holen
v=zeros(10,nt);  % die Matrix der zu berechnenden Werte
v(:,1)=u0';      % Startwerte in erste Spalte eingesetzt
                 % Hier kommt die Rechenschleife
for k=1:nt-1
    v(2:9,k+1)=v(2:9,k)+(dt*81/(pi*pi))*(v(1:8,k)-2*v(2:9,k)+v(3:10,k));
end
subplot(2,2,4);
                        % Ab hier wird geplottet
[TA, XA]=meshgrid(t,x); % Berechnen der Gitterwerte zum Plotten
surf(TA,XA,v)           % Plotkommando
xlabel('t');
ylabel('x');
xlim([0 2]);ylim([0 4]);zlim([0 2]);
% shading interp
title('Differenzen, \delta t=0.08');
```

2 Eliminationsverfahren

Hier betrachten wir die Lösung linearer Gleichungssysteme

$$Ax = b \text{ mit } A \in I\!R^{n\times n},\ b \in I\!R^n,\ x \in I\!R^n$$

zunächst mit dem konventionellen Gaußschen Eliminationsverfahren.

2.1 Das Eliminationsverfahren von Gauß

2.1.1 Rückwärtseinsetzen

Ist A eine nichtsinguläre obere Dreiecksmatrix, so ist durch die Formel (2.3)

$$x_j = \frac{1}{a_{jj}} \left(b_j - \sum_{k=j+1}^{n} a_{jk} x_k \right), \qquad j = n, \ldots, 1,$$

des Buches auf Seite 20 die Lösung x des Systems $Ax = b$ direkt ausrechenbar. Wir sehen hier eine "rückwarts" von n nach 1 laufende Schleife, aber das ist mit `for j=n:-1:1` kein Problem. Die innere Summe ist ein Skalarprodukt, und zwar von

$$\begin{array}{rcl} (x_{j+1}, \ldots, x_n)^T & = & \texttt{x(j+1:end,1)} \\ (a_{j,j+1}, \ldots, a_{j,n}) & = & \texttt{A(j,j+1:end)} \end{array}$$

wobei wir sicherheitshalber klargestellt haben, daß der MATLAB–Vektor `x` ein Spaltenvektor sein soll. Weil die obigen Vektoren je ein Spalten– und ein Zeilenvektor sind, können wir das Skalarprodukt als Matrizenmultiplikation schreiben. Das liefert den hinteren Teil unseres Beispielprogramms:

```
function x=rwe(A, b)
% Rueckwaertseinsetzen nach Proposition 2.2, Seite 29
% loest A*x=b, wenn A eine nichtsingulaere
% rechte obere Dreiecksmatrix ist
[m n]=size(A); % Groesse von A holen
if m~=n
    error('Matrix ist unsymmetrisch');
end
x=zeros(n,1);  % Loesungsvektor dimensionieren
for j=n:-1:1   % RUECKWAERTS rechnen!!!!
    x(j,1)=(b(j,1)-A(j,j+1:end)*x(j+1:end,1))/A(j,j);
    % das ist die Formel (2.3) des Buches
end
```

MATLAB merkt, daß das Skalarprodukt im Falle $j = n$ gar nicht auszurechnen ist, aber erwartet einen Vektor `x` um die rechte Seite überhaupt nachprüfen zu können. Den Rest des Programms erklären wir als Fortsetzung unseres MATLAB–Crashkurses:

2.1.2 Abfragen in MATLAB

Die Konstruktion

```
if m~=n
     error('Matrix ist unsymmetrisch');
end
```

fragt ab, ob im obigen Fall die $m \times n$–Matrix nicht–quadratisch ist, d.h. ob $m \neq n$ gilt. Wenn ja, wird eine Fehlermeldung ausgegeben und die Rechnung wird brutal beendet. Analog arbeitet man für andere Abfragen, wobei die Abfrage auf Gleichheit mit `==` statt mit `=` gemacht wird, denn `=` ist schon für die Wertzuweisung verbraten. Natürlich kann man in naheliegender Weise auch auf

```
 <     >    <=     >=
```

abfragen und auch die Alternative behandeln, z.B.

```
if m==n
     disp('Matrix ist symmetrisch');
else
     error('Matrix ist unsymmetrisch');
end
```

wobei `disp` als Kurzform von "display" nur eine Meldung an das Kommandofenster gibt, aber nicht die Rechnung abbricht. Natürlich kann man Abfragen schachteln, z.B.

```
if m~=n
     error('Matrix ist unsymmetrisch');
else
   if n>1000
     error('Matrix ist zu gross');
   else
     disp('Matrix ist symmetrisch und nicht zu gross');
   end
end
```

2.1.3 Funktionen als `m`–Files

Jetzt erklären wir den Anfang des Beispielprogramms. Der gesamte Text ist als eine Datei `rwe.m` gespeichert, und die Startzeile

```
function x=rwe(A, b)
```

bewirkt, daß man in jedem anderen MATLAB-Programm, das in demselben Verzeichnis liegt, diese Funktion aufrufen kann. Das werden wir brauchen, denn wir wollen ja das Rückwärtseinsetzen **nach** einer Elimination ausführen.

Die obige Startzeile informiert MATLAB, daß die Funktion `rwe`, die in dieser Datei definiert ist, zwei Eingabe– und einen Ausgabeparameter hat, deren Namen intern als `A`, `b` und `x` definiert sind. Beim Aufruf aus einem anderen MATLAB–Programm kann man die dortigen Namen verwenden, z.B.

```
loesung=rwe(matrix, rechteseite)
```

wenn man sich vorher kunstvoll eine `matrix` als obere Dreiecksmatrix und einen passenden Vektor `rechteseite` gebaut hat. Was dann passiert, nennt man in der Informatik einen *call–by–value*: die Werte der Elemente der `matrix` und des Vektors `rechteseite` werden berechnet und in eine lokale Kopie von `A` und `b` geschrieben, die dann in `rwe.m` als Eingabe dienen. Das `m`–file `rwe.m` arbeitet also immer auf einer eigenen lokalen Kopie, was die Eingabeparameter angeht.

Dann rechnet `rwe.m` eine Lösung im internen Vektor `x` aus, und am Ende werden dessen Komponentenwerte in den Vektor `loesung` des Aufrufs kopiert und stehen dann im aufrufenden Programm zur Verfügung. Natürlich könnte man Zeit sparen, indem man direkt auf `matrix` und `rechteseite` rechnet und das Kopieren spart. Das erfordert *call–by–reference* und ist nichts für Anfänger.

Wir testen `rwe.m` durch das davon unabhängige Programm[1]

```
n=15;
mat=rand(n,n);  % eine n x n Zufallsmatrix
rs =rand(n,1);  % ein Zufallsvektor
for i=1:n       % auf obere Dreiecksform bringen
    for j=i+1:n
        mat(j,i)=0;
    end
end
% und jetzt noch Testausgabe
mat
rs
loes=rwe(mat,rs)
rest=rs-mat*loes
```

und stellen fest, daß $b - Ax$=`rest` fast immer in der Größenordnung der Maschinengenauigkeit liegt. Wir haben dabei bewußt nicht die Bezeichnungen `A` und `b` verwendet. Eine exemplarische Ausgabe wird hier nicht eingebaut,

[1] `http://www.num.math.uni-goettingen.de/schaback/teaching/texte/NuMath/matlab/Kapitel02/rwet`

weil man das besser *live* in der Vorlesung oder am Rechner durchspielt. Das wird für viele der folgenden Beispielprogramme gelten.

2.1.4 Eliminationsschritt

Wir gehen jetzt auf Seite 30 des Buches und finden dort, daß der Standard–Eliminationsschritt auf einer Matrix $A \in I\!R^{n\times n} = (a_{ij})_{1\le i,j\le n}$ sich schreiben läßt als

$$A \Rightarrow L^{(1)}A = \left(E - m^{(1)}e_1^T\right) A = A - m^{(1)}e_1^T A$$

mit dem Vektor

$$m^{(1)} := \left(0,\ \frac{a_{21}}{a_{11}}, \dots \frac{a_{n1}}{a_{11}}\right)^T .$$

Bis auf das erste Element ist das der Vektor

```
m=A(:,1)/A(1,1);
```

als renormierte erste Spalte von `A`. Und dann ist die obige Transformation nichts anderes als

```
A=A-m*A(1,:);
```

wobei natürlich einiges zuviel berechnet wird, aber das soll uns hier nicht stören. Man muß dann nur noch die rechte Seite mittransformieren, und alles in allem bekommen wir das Programm[1]

```
function [A, b]=elimnaiv(A, b)
% ein einzelner naiver Eliminationsschritt, S. 30 des Buches
if abs(A(1,1))<eps
    error('Diagonalelement zu klein')
end
% Dies wird der Vektor m des Skripts
m=A(:,1)/A(1,1);  % renormierte erste Spalte
m(1,1)=0;         % vorne eine Null rein
% Aneu=(E-m*e_1')*A=A-m*A(1,:)
A=A-m*A(1,:);     % dann A transformieren
% Hier wird zuviel gemacht,
% weil die erste Zeile und Spalte von A mittransformiert werden
b=b-m*b(1,1);     % und b transformieren
```

[1] http://www.num.math.uni-goettingen.de/schaback/teaching/texte/NuMath/matlab/Kapitel02/elim

Obwohl natürlich immer noch *call–by–value* vorliegt, haben wir das Programm so geschrieben, dass es bei einem Aufruf der allgemeinen Form

```
[am, bv]=elimnaiv(am,bv)
```

d.h. mit Eingabeparametern=Ausgabeparametern eine Überschreibung der alten Daten macht. Das werden wir gleich ausnutzen.

2.1.5 Gauß–Elimination ohne Pivotisierung

Wir haben oben schon gesehen, wie man Restmatrizen bildet. Man muß also nur noch den obigen Eliminationsschritt auf `A` und dann auf die Restmatrizen `A(k:end,k:end)` für $k = 2, 3, \ldots, n-1$ anwenden. Das packen wir gleich in eine ausführbares `m`–file[1]

```
clear all;
% Gauss-sches Eliminationsverfahren naiv, ohne Pivotisierung
n=5;
A=rand(n,n)    % eine Zufallsmatrix
Asave=A;       % Kopie wird gespeichert
b=rand(n,1)    % ein Zufallsvektor
bsave=b;       % Kopie wird gespeichert
for k=1:n-1    % Schleife ueber Eliminationsschritte
    % das wirkt auf die Restmatrizen
    Eliminationsschritt=k % nur zur Ausgabe,
    % damit man sieht, wo man ist
    % und hier der Standard-Eliminationsschritt
    [A(k:end,k:end), b(k:end,1)]=...
          elimnaiv(A(k:end,k:end),b(k:end,1))
end
x=rwe(A,b)               % Rueckwaertseinsetzen
rest=Asave*x-bsave       % Ergebnistest
x=Asave\bsave            % was macht MATLAB?
restMATLAB=Asave*x-bsave
```

Man sieht, daß das Fehlerverhalten von MATLAB in vielen Fällen ein wenig besser ist, und das liegt daran, daß keine naive Gauß–Elimination gerechnet wird.

[1] `http://www.num.math.uni-goettingen.de/schaback/teaching/texte/NuMath/matlab/Kapitel02/gaus`

2.2 LR–Zerlegung

Die Formeln auf Seite 33 oben, die eine LR–Zerlegung ohne Pivotisierung ausrechnen, lassen sich ganz einfach in ein MATLAB-Programm stecken:

```
function [L, R]=LRnaiv(A)
% naive LR-Zerlegung ohne Pivotisierung
[m, n]=size(A);
if m~=n
   error('Matrix ist unsymmetrisch');
end
L=eye(n);          % wir reservieren die Einhaitsmatrix
R=zeros(n,n);      % und eine Nullmatrix
% jetzt kommen die Formeln auf Seite 33 oben, ohne Aenderung
for i=1:n
    for j=1:i-1
        L(i,j)=(A(i,j)-L(i,1:j-1)*R(1:j-1,j))/R(j,j);
    end
    for j=i:n
       R(i,j)=A(i,j)-L(i,1:i-1)*R(1:i-1,j);
    end
end
```

und das kann man testen mit

```
clear all;
n=5;
A=rand(n,n) % Zufallsmatrix
[L, R]=LRnaiv(A);
L
R
rest=A-L*R
```

2.3 Pivotisierung

Jetzt pivotisieren wir, und zwar durch Zeilenvertauschen und Durchsuchen der ersten Spalte. Man sehe sich dazu das MATLAB–Kommando `max` an. Wie viele andere arbeitet es auf Matrizen spaltenweise, d.h. es berechnet einen Zeilenvektor der Spaltenmaxima. Aber es berechnet nicht nur den Zahlenwert des Maximums, sondern auch den Index, für den das Maximum angenommen wird. Also liefert

```
[vmax imax]=max(abs(A(:,1)))
```

den Index `imax` mit

$$|A(imax, 1)| = \max\{|A(i, 1)| \ : \ 1 \leq i \leq m\}$$

und deshalb ist `imax` der Index der Zeile, die mit der ersten zu vertauschen ist. Man muss aber auch noch die rechte Seite mitvertauschen, weil sich ja die gesamte Gleichungsnumerierung ändert. Das Ganze ist ein m-file

```
function [A, b]=elimpivot(A, b)
% ein einzelner Eliminationsschritt, mit Pivotisierung
% durch Zeilenvertauschen
[vmax imax]=max(abs(A(:,1)'))
if vmax<eps
    error('Pivot zu klein')
end
% und jetzt wird vertauscht
helprow=A(1,:);
A(1,:)=A(imax,:);
A(imax,:)=helprow;
help=b(1,1);
b(1,1)=b(imax,1);
b(imax,1)=help;
% und jetzt tun wir so, als waere nichts passiert:
% Dies wird der Vektor m des Skripts
m=A(:,1)/A(1,1);  % renormierte erste Spalte
m(1,1)=0;         % vorne eine Null rein
% Aneu=(E-m*e_1')*A=A-m*A(1,:)
A=A-m*A(1,:);     % dann A transformieren
% Hier wird zuviel gemacht,
% weil die erste Zeile und Spalte von A mittransformiert werden
b=b-m*b(1,1);     % und b transformieren
```

in dem der Rest genau so läuft wie ohne Pivotisierung. Das wird dann in einem anderen m–file

```
function x=gausspivot(A, b)
% Gauss-Elimination mit Zeilenpivotisierung, fuer A*x=b
[m, n]=size(A);
if m~=n
   error('Matrix ist unsymmetrisch');
```

```
end
for k=1:n-1    % Schleife ueber Eliminationsschritte
    % das wirkt auf die Restmatrizen
    eliminationsschritt=k % nur zur Ausgabe, damit man sieht, wo man ist
    % und hier der Standard-Eliminationsschritt
    [A(k:end,k:end), b(k:end,1)]=elimpivot(A(k:end,k:end),b(k:end,1))
end
x=rwe(A,b)               % Rueckwaertseinsetzen
```

aufgerufen, und wie folgt getestet:

```
clear all;
% Gauss-sches Eliminationsverfahren mit Zeilenpivotisierung
n=5;
A=rand(n,n)    % eine Zufallsmatrix
b=rand(n,1)    % ein Zufallsvektor
x=gausspivot(A, b)
rest=A*x-b     % Ergebnistest

x=A\b;   % MATLAB
restMATLAB=A*x-b
```

2.4 Cholesky–Verfahren

Hier ist alles in ein einziges m–file gepackt, und die Leser sollten das selbst in ein vernünftiges Programmsystem umstricken.

```
clear all;
n=5
A=rand(n,n); % Zufallsmatrix, nicht positiv definit
A=A*A'       % das macht sie positiv (semi-) definit
L=zeros(n,n); % L muss dimensioniert werden
for j=1:n     % hier kommen die Formeln von Seite 40 des Buches
    L(j,j)=sqrt(A(j,j)-L(j,1:j-1)*(L(j,1:j-1))');
    for i=j+1:n
        L(i,j)=(A(i,j)-L(i,1:j-1)*(L(j,1:j-1))')/L(j,j);
    end
end
L
Rest=A-L*L'
```

Aufgabe:

Um beim Cholesky-Verfahren die Wurzelfunktion zu vermeiden, kann man die Matrix A in der Form $A = LDL^T$ mit einer normierten unteren Dreiecksmatrix L und einer Diagonalmatrix D zerlegen. Implementieren Sie einen entsprechenden Algorithmus zur Berechnung von L und D.

Eine denkbare Lösung ist

```
clear all;
n=15
A=2*rand(n,n)-1;     % Zufallsmatrix
A=(A+A')/2;      % das macht sie symmetrisch
L=eye(n);        % L muss dimensioniert werden
D=zeros(1,n);    % D muss dimensioniert werden, hier als Zeile
for j=1:n        % Varianten der Formeln von Seite 40 des Buches,
                 % aus A=L*D*L' folgend
    D(1,j)=A(j,j)-(L(j,1:j-1).*D(1,1:j-1))*L(j,1:j-1)';
    for i=j+1:n
        L(i,j)=(A(i,j)-(L(i,1:j-1).*D(1:j-1))*(L(j,1:j-1)'))/D(1,j);
    end
end
% Testausgabe
% L
Diagonale=D'
Rest=A-L*diag(D)*L';
restnorm=norm(Rest)
```

Hier ist die Lösungsidee. Man man schreibt direkt die Gleichung $A = LDL^T$ in Komponenten hin und bekommt

$$a_{ij} = \sum_{k=1}^{j} L_{ik} D_{kk} L_{jk},\ 1 \leq j \leq i \leq n. \tag{2.4.1}$$

Für $i = j$ ist das

$$a_{jj} = \sum_{k=1}^{j} L_{jk} D_{kk} L_{jk}$$

oder

$$a_{jj} - L_{jj} D_{jj} L_{jj} = a_{jj} - D_{jj} = \sum_{k=1}^{j-1} L_{jk} D_{kk} L_{jk}$$

weil die L-Matrix hier normiert ist. Löst man das nach D_{jj} auf, so bekommt man die erste Zeile in der Schleife. Die zweite Zeile ergibt sich aus 2.4.1 durch Auflösen nach L_{ij} als

$$a_{ij} - L_{ij}D_{jj}L_{jj} = a_{ij} - L_{ij}D_{jj} = \sum_{k=1}^{j-1} L_{ik}D_{kk}L_{jk},\ 1 \leq j+1 \leq i \leq n.$$

3 Störungsrechnung

In diesem Kapitel gibt es erstmal viel Theorie und kaum etwas zu rechnen. Aber hier ist eine kleine Demo zum Kommando `norm` von MATLAB.

```
clear all;
x=[ 1 2 3]
normx2=norm(x)
normx1=norm(x,1)
normxinf=norm(x,Inf)
p=3.5
normxp=norm(x,p)
A=[1 2 3 ; 2 3 4]
normA=norm(A)
normA1=norm(A,1)
normA2=norm(A,2)
normAinf=norm(A,Inf)
normA1fro=norm(A,'fro')
singval=sqrt(abs(eig(A'*A)))
```

Man sieht, daß MATLAB auf Vektoren die p–Normen für $1 \leq p \leq \infty$ beherrscht, wobei man `Inf` ohne Apostrophe ins zweite Argument setzen kann.

Bei Matrizen ist der zweite Parameter p der Normfunktion immer als $\|.\|_{p,p}$ mit $p \in \{1, 2, \infty\}$ zu verstehen, d.h. man hat

p	Norm
1	Spaltensummennorm
2	Spektralnorm
Inf	Zeilensummennorm

aber es gibt auch noch den Wert p=`'fro'` um die Frobeniusnorm zu berechnen.

Beim Lösen eines linearen Gleichungssystems $Ax = b$ kann man aus der Kleinheit des Residuums $r := Ax - b$ nicht darauf schließen, die "Lösung" genau berechnet zu haben. Dazu ein schlichtes Spielprogramm:

```
clear all;
format long g
% Demo zum Loesen linearer Gleichungssysteme
n=2
A=[1 1 ; 1 1.0001]
x=rand(n,1) % ein Zufallsvektor
b=A*x       % also loest x das System b=A*x
Rest_von_x=b-A*x
% wir beschaffen uns einen Stoerungsvektor z
% als Eigenvektor zu einem kleinen Eigenwert
[V, D]=eig(A);
z=10^5*V(:,1)
% und testen mal, wie gut x+z das System loest
Rest_von_x_plus_z=b-A*(x+z)
x
x_plus_z=x+z
```

Was passiert hier? Die Idee ist, ein zufälliges x durch $b := Ax$ zur "wahren" Lösung zu machen, um dann ein möglicht riesiges z zu finden, so daß $x + z$ auch noch "ganz gut" das System löst. Natürlich muss dann Az ziemlich klein sein, und es ist keine schlechte Idee, z als Eigenvektor zum kleinsten Eigenwert von A zu wählen. Aus Jux verpassen wir dem ansonsten bei MATLAB immer auf Norm 1 getrimmten Eigenvektor (er ist im Programm die erste Spalte der kompletten Eigenvektormatrix `V`) noch einen Faktor 10^5 und sehen uns dann an, wie "gut" der Vektor $x+z$ dann noch das System löst. Klar? Bitte durchspielen, und an der Matrix und dem Faktor 10^5 Änderungen anbringen. Wer schon weiß, wie die Kondition von A definiert ist, kann diese mit

```
cond(A)
```

aufrufen.

Aufgabe:
Gegeben seien die Matrizen

$$H_n := \left(\frac{1}{i+j-1}\right)_{i,j=1,\ldots,n} \in I\!R^{n\times n}.$$

Versuchen Sie für $n \in \{5, 11, 20\}$ mit MATLAB das Gleichungssystem $H_n x = b$ zu lösen, wobei $b = H_n(1, \ldots, 1)^T$ verwendet werden soll. Überprüfen Sie für die gefundenen Lösungen die Norm des Residuums und vergleichen Sie die Lösung mit der exakten Lösung $x^* = (1, \ldots, 1)^T$. Was bedeutet das für den Vektor $x - x^*$? Wie kann man mit MATLAB eine "bessere" Lösung des Gleichungssystems bekommen?

Hier ist eine Beispiellösung:

```
clear all;
n=20
H=hilb(n);  % Hilbertmatrix
einsen=ones(n,1); % Einservektor
b=H*einsen        % kuenstliche rechte Seite
x=H\b;            % MATLAB-Loesung
resid=H*x-b;      % Residuum
xfehler=x-einsen; % x - Fehler, absolut=relativ
% Jetzt eine Verbesserung durch
% regularisierte Singulaerwertzerlegung.
% Man sehe sich Seite 69 des Buches an.
[U S V]=svd(H);  % Singulaerwertzerlegung
z=U'*b;          % neue rechte Seite
ep=1.0e-12;      % das Epsilon des Buches
sd=diag(S);      % S-Diagonale rausholen
gutliste=find(abs(sd)>=ep);  % hole die guten Indizes
y=zeros(n,1);                % das wird der Loesungsvektor bei SVD,
                             % die  regularisierte Form von y=S\z
y(gutliste,1)=z(gutliste,1)./sd(gutliste,1);% vektoriell berechnet
xsvd=V*y;                    % das ist die eigentliche SVD-Loesung
xsvdfehler=xsvd-einsen;      % und das ist der x-Fehler
svdresid=H*xsvd-b;           % und das neue Residuum.
% Ausgabe vertikal, vierspaltig, zum Einbinden in den Text
format short g
Ausgabe=...
    [xfehler resid xsvdfehler svdresid]
save ('Aufgabe12.txt','Ausgabe','-ascii')
```

Wie man aus dem obigen Programm ersehen kann, enthält die folgende Wertetabelle, die einfach die Datei `Aufgabe12.txt` einliest, in den ersten beiden Spalten den x–Fehler und das Residuum der MATLAB–Lösung, während die beiden letzten Spalten den x–Fehler und das Residuum der Lösung durch

regularisierte Singulärwertzerlegung enthalten. Man sieht deutlich die Überlegenheit der Singulärwertzerlegung.

```
 1.3495872e-007  8.8817842e-016 -1.0158108e-010  4.8849813e-015
-2.5083520e-005  4.4408921e-016  5.6601610e-009  1.7763568e-015
 1.1115407e-003  0.0000000e+000 -7.2295391e-008  8.8817842e-016
-2.0716807e-002  2.2204460e-016  3.3187527e-007  4.4408921e-016
 2.0201756e-001  0.0000000e+000 -4.8160226e-007  4.4408921e-016
-1.1414087e+000 -6.6613381e-016 -4.5491531e-007  4.4408921e-016
 3.8862784e+000  4.4408921e-016  1.3476823e-006  4.4408921e-016
-7.8416402e+000 -8.8817842e-016  4.2165155e-007  4.4408921e-016
 8.4462231e+000 -4.4408921e-016 -1.2165891e-006 -2.2204460e-016
-3.9071457e+000  2.2204460e-016 -1.2558229e-006 -2.2204460e-016
 5.4270900e+000 -2.2204460e-016  1.0067434e-007  2.2204460e-016
-2.0191420e+001 -6.6613381e-016  1.3436121e-006  2.2204460e-016
 2.3786631e+001 -7.7715612e-016  1.3769967e-006  6.6613381e-016
 8.5771690e+000 -1.1102230e-016  2.3799942e-007  4.4408921e-016
-5.1545313e+001 -4.4408921e-016 -1.1428235e-006  3.3306691e-016
 6.4565711e+001 -2.2204460e-016 -1.6437105e-006  2.2204460e-016
-4.8750732e+001  2.2204460e-016 -6.9269942e-007  2.2204460e-016
 2.6814333e+001 -2.2204460e-016  1.1811426e-006 -1.1102230e-016
-1.0191139e+001 -1.1102230e-016  2.0552140e-006  2.2204460e-016
 1.8829764e+000  1.1102230e-016 -1.4419305e-006  2.2204460e-016
```

Man sollte hier noch auf das `find` Kommando von MATLAB hinweisen. Man wendet es im einfachsten Fall (siehe oben) auf einen logischen Ausdruck an, der für Vektoren gleicher Länge und gleicher Indexmenge definiert ist, und man bekommt die Liste der Indizes, für die der Ausdruck zutrifft. Danach arbeitet man mit dieser Liste geschickt weiter, wobei man sie in andere Vektoren und Matrizen anstelle von `:` einsetzt.

Schließlich wurde oben noch das Kommando `save` verwendet, mit dem man Ausgaben von Matrizen oder Vektoren abspeichern kann.

4 Orthogonalisierungsverfahren

Wir gehen jetzt auf Seite 61 und berechnen den Normaleneinheitsvektor `u` auf einer Spiegelungsebene, die einen Vektor `a` auf ein Vielfaches des ersten Einheitsvektors abbildet. Wir beschaffen uns die Norm von `a` und das Vorzeichen `sg` des ersten Elementes `a(1,1)` um wie im Buch den Vektor `v` auszurechnen. Dann bekommen wir `u` durch Renormierung und sind fertig:

```
function u=householder(a)
% gibt den Normaleneinheitsvektor u
% der Spiegelungsebene zureck,
% mit der man a auf ein Vielfaches
%  des ersten Einheitsvektors spiegelt
norm_a=norm(a,2);
if norm_a<eps
    error('Vektor a zu klein')
end
sg=sign(a(1,1));
v=a;
v(1,1)=v(1,1)+sg*norm_a;
norm_v=norm(v,2);
u=v/norm_v;
trsf=a-2*u*(u'*a);
```

So, und jetzt müssen wir das schön anwenden. Um Q^T zu bekommen, wenden wir alle Householder–Transformationen auch auf die anfängliche Einheitsmatrix an. In dem folgenden Programm ist `a` eine Matrix, und wir arbeiten wie bei der Gauß–Elimination immer auf einer Restmatrix, aus der wir uns das untere Stück der k–ten Spalte holen, um den Normaleneinheitsvektor auszurechnen. Man muß hier ein wenig aufpassen, weil der sich allmählich aufbauende R–Teil in `a` in den Folgeschritten noch verändert. Deshalb werden die Transformationen auf `a` und `q` nicht auf geschickt gewählte Teilmatrizen angewendet, sondern einfach immer auf die vollen Matrizen, und dazu wird der Vektor `u` immer auf volle Länge durch Nullen aufgefüllt. Wie kann man das verbessern?

```
clear all;
% Householderverfahren zur QR-Zerlegung
n=5
a=rand(n,n)
asave=a;  % aufheben, um zu testen
q=eye(n); % wir wollen auch q=Q' berechnen
for k=1:n-1   % Zerlegungsschleife
    u=zeros(n,1);
    u(k:end,1)=householder(a(k:end,k))
    % wir rechnen das ziemlich primitiv vor...
    a=a-2*u*(u'*a)
    q=q-2*u*(u'*q)
end
% Rest berechnen
```

```
rest=asave-q'*a
% testen auf Orthogonalitaet
test=q'*q-eye(n)
test=q*q'-eye(n)
```

Hier ist das Ganze nochmal, als unpivotisierte QR–Zerlegung. Aber wir machen das etwas allgemeiner, indem wir eine nichtquadratische Eingabematrix zulassen.

```
function [q, r]=qrnaiv(a)
% Naives Householderverfahren zur QR-Zerlegung
% fuer nicht notwendig quadratische Matrizen a
[n m]=size(a);
qs=eye(n); % wir wollen auch q=Q' berechnen
mn=min(n,m); % das gibt die Schrittzahl des Verfahrens an
for k=1:mn    % Zerlegungsschleife
    u=zeros(n,1);
    u(k:end,1)=householder(a(k:end,k)) ;
    % wir rechnen das ziemlich primitiv vor...
    a=a-2*u*(u'*a);
    qs=qs-2*u*(u'*qs);
end
r=a;
q=qs';
```

Und es folgt noch ein Treiberprogramm:

```
a=rand(4,6)
[ q r]=qrnaiv(a)
q*r-a
q*q'
q'*q
```

Aufgabe:

Implementieren Sie die QR-Zerlegung mit Pivotisierung durch Spaltenvertauschung und Rangentscheid. Modifizieren Sie dazu die obige Beispielimplementierung der einfachen QR-Zerlegung.

Eine mögliche Lösung ist

```
function [q, a, p, rk]=qrpivot(a)
% Householderverfahren zur QR-Zerlegung
% mit Pivotisierung und Rangentscheid.
% Inputmatrix a darf nichtquadratisch sein.
% Ergebnis: a(:,p(:))=q*r  mit Permutationsvektor p
% und Rang in rk
[m n]=size(a);
p=1:n; % Permutation der Spalten
q=eye(m); % wir wollen auch q=Q' berechnen
qsum=sum(a.*a); % Spaltenquadratsummen als Zeilenvektor
mn=min(m,n);   % Schrittzahl des Verfahrens
for k=1:mn   % Zerlegungsschleife
    % Maximum der laufenden Spaltenquadratsummen
    [qsmax kmax]=max(qsum(1,:));
    % Diese werden upgedatet, und
    % fuer die schon orthogonalisierten
    % Spalten sollten die Quadratsummen etwa Null sein.
    if kmax<k    % wenn das Max. vorne ist,
        % ist irgendwas faul, nicht weiterrechnen
        rk=k-1;
        break;
    end
    if qsmax<eps % ebenso, wenn keine brauchbare
                 % Quadratsumme mehr da ist
        rk=k-1;
        break;
    end
    % Volle Spaltenvertauschung
    help=a(:,kmax);
    a(:,kmax)=a(:,k);
    a(:,k)=help;
    % Permutation veraendern
    hp=p(kmax);
    p(kmax)=p(k);
    p(k)=hp;
    % Quadratsumme auch vertauschen
    hp=qsum(kmax);
    qsum(kmax)=qsum(k);
    qsum(k)=hp;
    % So, jetzt koennen wir normal householden
    u=zeros(m,1);
```

```
    u(k:end,1)=householder(a(k:end,k));
    % wir rechnen wieder mal primitiv
    a=a-2*u*(u'*a);
    q=q-2*u*(u'*q);
    % aber wir muessen noch die Quadratsummen updaten
    qsum=qsum-a(k,:).*a(k,:);
    rk=k;
end
q=q';
```

Fortsetzung der Aufgabe: Für die elliptisch angenommene Flugbahn $x^2 = ay^2+bxy+cx+dy+e$ eines Himmelskörpers wurden über einen Messzeitraum folgende Daten gemessen:

x	1.0249	0.9499	0.8661	0.7734	0.6714	0.5595	0.4371	0.3030	0.1555	0.0075
y	0.3893	0.3229	0.2653	0.2166	0.1772	0.1476	0.1286	0.1214	0.1273	0.1489

Stellen Sie mit diesen Daten das überbestimmte lineare Gleichungssystem für die Koeffizienten a, b, c, d, e auf und lösen Sie das zugehörige lineare Ausgleichsproblem mit Ihrer QR-Implementierung. Plotten Sie die Messwerte zusammen mit der so bestimmten Ellipse. Eine mögliche Lösung ist

```
clear all;
close all;
n=5 % Zahl der Unbekannten
% wir holen die Daten
x=[1.0249  0.9499  0.8661  0.7734  0.6714 ...
        0.5595  0.4371  0.3030  0.1555  0.0075]';
y=[0.3893  0.3229  0.2653  0.2166  0.1772 ...
        0.1476  0.1286  0.1214  0.1273  0.1489]';
% und zaehlen sie ab
m=length(x);
a=[y.^2 x.*y x y ones(m,1)]; % Anpassungsmatrix laut Aufgabe
b=x.^2;  % rechte Seite
[q, a, p, rk]=qrpivot(a);   % Aufruf QR mit Pivotisierung
c=q'*b;           % Transformation im Bildraum
loes=a\c;            % Loesung modulo Permutation
coeff=loes(p(:));    % Rueckpermutierte Loesung
plot(x,y,'rx')    % Datenplot, rote kleine x gesetzt
title('Aufgabe 20')
hold on          % wir wollen auch die Loesung plotten
```

```
yp=0:0.001:2.5;  % diese Grenzen muss man ausprobieren
% Wir loesen nach x auf, und brauchen
% den Radikanden der pq Formel
rad=coeff(1)*yp.^2+coeff(4)*yp+...
      coeff(5)+((coeff(3)+coeff(2)*yp).^2)/4;
% hole die Indizes, wo der Radikand positiv ist
posi=find(rad>=0);
% fuer diese Indizes gibt es je einen x-Wert,
% SEHR INEFFIZIENT gerechnet:
xprechts=(coeff(3)+coeff(2)*yp(posi))/2+sqrt(rad(posi));
xplinks =(coeff(3)+coeff(2)*yp(posi))/2-sqrt(rad(posi));
plot(xprechts,yp(posi))
hold on
plot(xplinks,yp(posi))
legend('Daten','Ellipse')
xlabel('x')
ylabel('y')
```

mit der Ausgabe in Abbildung 4. Das ist ein stark vereinfachter Modellfall dessen, was Gauß leisten mußte, als er aus wenigen Beobachtungsdaten eine Planetoidenbahn ausrechnen wollte.

5 Lineare Optimierung

In der Vorlesung im WS 2008/2009 fiel dieses Kapitel weg. Deshalb gibt es hier auch keine MATLAB-Beispiele.

6 Iterative Verfahren

6.1 Mandelbrotmenge

Zum Banachschen Fixpunktsatz gibt es nicht viel zu analysieren. Wir holen das beim Vergleich mit dem Newtonverfahren nach. Aber es ist auch sehr einfach, ein paar Iterationen mit einem der Standardbeispiele

$$\begin{aligned} F(x) &= \cos(x) \\ F(x) &= \frac{x}{2} + \frac{1}{x} \end{aligned}$$

durchzuführen.

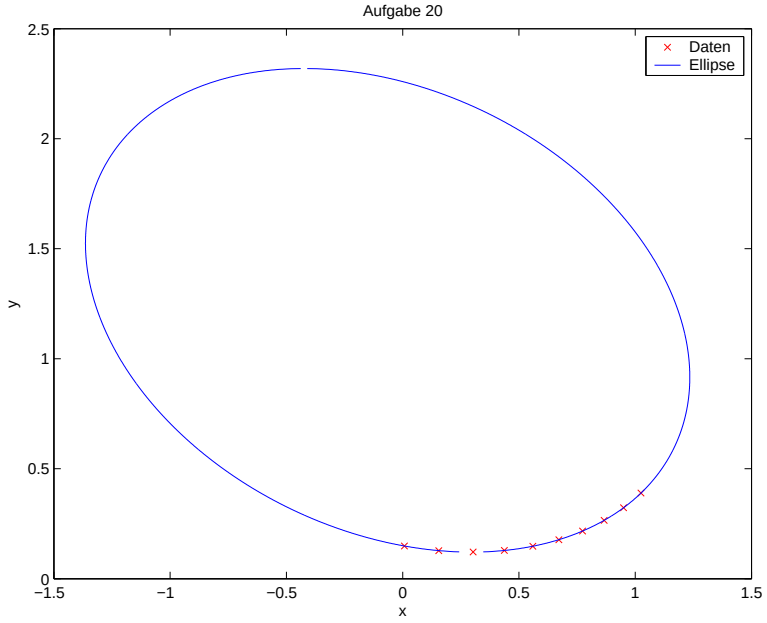

Abbildung 4: Rekonstruktion einer Ellipse

Aber wir wollen demonstrieren, wie nahe Ordnung und Chaos beieinanderliegen. Im Komplexen kann man zu gegebenem $c \in \mathbb{C}$ die Fixpunktiteration mit der Iterationsfunktion

$$F(z) = z^2 + c$$

mit den Fixpunkten

$$z_{1/2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} - c}$$

betrachten. Wir wollen hier nicht die Konvergenzanalyse durchführen, bemerken aber, daß die Iteration für hinreichend kleine $|c|$ in einer Umgebung der Null gegen z_2 konvergiert. Aber es ist absolut nicht klar, was für andere c passiert. Die *Mandelbrotmenge* illustriert das. Sie besteht aus den komplexen Zahlen c, für die man bei Start in c oder in 0 eine beschränkte Folge von Iterierten bekommt, und deshalb besteht ihr Rand aus den Grenzfällen zwischen Konvergenz (von Teilfolgen) und Nicht–Konvergenz. Das folgende simple Programm (Achtung, es erfordert die Image Processing Toolbox von MATLAB) macht einen Plot der Mandelbrotmenge in Abbildung 5, wobei die Punkte des Komplements unterschiedlich eingefärbt werden, und zwar in Abhängigkeit von der Zahl der Schritte, die gebraucht werden, bis man sicher sein kann, daß die Folge gegen Unendlich geht.

```
clear all;
```

```
close all;
% Programm zur Mandelbrotmenge
% Wir denken uns c=x+i*y als komplexe Zahlen.
xmin=-2; % Intervallgrenze x
xmax=1;  % Intervallgrenze x
ymin=-1.3; % Intervallgrenze y
ymax=1.3;  % Intervallgrenze y
np=2000; % Zahl der Punkte pro Dimension
hx=(xmax-xmin)/np;  % relative Schrittweite
hy=(ymax-ymin)/np;  % relative Schrittweite
radius=sqrt(xmin^2+ymin^2); % bei dieser Lage = max(abs(c))
% Grenze zur Abfrage auf Divergenz:
schranke=0.5+sqrt(0.25+radius)
schritt=150;  % max. Schrittzahl zum Entscheiden
% auf Beschraenktheit
[xp yp]=meshgrid(xmin:hx:xmax,ymin:hy:ymax); % Gitter legen
xpv=xp(:); % Gitter als eindimensionale Vektoren hinschreiben
ypv=yp(:);
ipv=zeros(size(xpv));  % hier kommen die Schrittzahlen rein
for j=1:length(xpv)    % Schleife ueber das Gitter
    c=xpv(j)+i*ypv(j);
    z=c;               % Start der Iteration in z=c
    for k=1:schritt
        if abs(z)>schranke  % wenn vorzeitig draussen,
            ipv(j)=k;       % mit Schrittzahl einfaerben
            break;          % und Schluss mit diesem c
        end
        z=z*z+c;       % Iteration
    end
end
% wenn die Iteration beschraenkt bleibt, ist der ipv-Wert Null
ip=reshape(ipv,size(xp));  % zuruecksortieren auf Gitterform
image(ip*2)                % Bild malen, brutale Farbwahl
title('Mandelbrotmenge')
```

Wir werden später beim Newtonverfahren noch einmal sehen, wie kompliziert die Grenze zwischen Konvergenz und Divergenz ist.

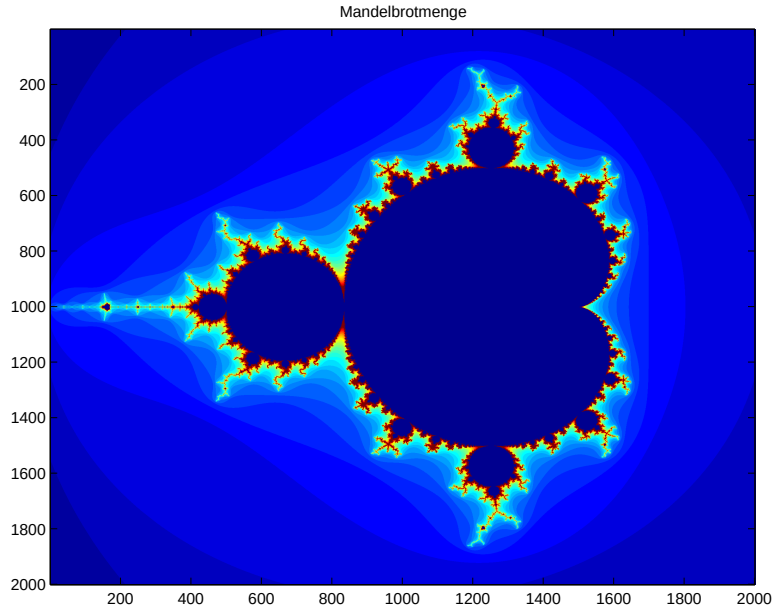

Abbildung 5: Mandelbrotmenge

6.2 Iterative Verfahren für Lineare Gleichungssysteme

Das folgende kleine `m`–File ist das Gesamtschrittverfahren in der Matrix–Vektor–Form

$$x = D^{-1}(b - (L + R)x) = (b - (A - D)x)D^{-1},$$

wobei im Programm `D` nur die Diagonale von D ist und `diag(D)` aus der Diagonale wieder eine volle Matrix D macht. Im Allgemeinen baut `diag` bei Anwendung auf einen Vektor v eine quadratische Diagonalmatrix mit v in der Diagonale, aber bei Anwendung auf eine Matrix wird die Diagonale als Vektor extrahiert. Also ist `a-diag(diag(a))` die Matrix `a` mit auf Null gesetzter Diagonale, und das werden wir weiter unten benutzen.

```
function x=gsvschritt(A, b, x)
% macht einen Schritt des GSV, SEHR naiv programmiert
D=diag(A);
x=(b-(A-diag(D))*x)./D;
```

Beim Einzelschrittverfahren kann man nicht so einfach Vektoren übereinanderschreiben. Eine simple, aber nicht schleifenfreie Version ist

```
function x=esvschritt(A, b, x)
% macht einen Schritt des ESV, SEHR naiv programmiert
[n m]=size(A);
for i=1:n
    x(i,1)=(b(i,1)-A(i,1:i-1)*x(1:i-1,1)...
        -A(i,i+1:end)*x(i+1:end,1))/A(i,i);
end
```

und das entspricht genau (6.12) auf Seite 100 des Buches.

Hier ist ein kleines m–file für das Zeilensummenkriterium:

```
function zsnorm=zsk(a)
% Zeilensummenkriterium = Zeilensummennorm der GSV-Iterationsmatrix
ad=a-diag(diag(a));
s=sum(abs(ad'))./diag(a)';
zsnorm=max(s');
```

Hier ist wieder einmal zu bedenken, dass `sum` immer Spaltensummen berechnet. Jetzt kommt das Sassenfeldkriterium als Umsetzung von Satz 6.15 auf Seite 100:

```
function sf=sassenfeld(a)
% Sassenfeldwert = obere Schranke der Zs-Norm der ESV-Iterationsmatrix
[n m]=size(a);
s=zeros(n,1);
for i=1:n
    u=abs(a(i,1:i-1))*s(1:i-1,1);
    v=sum(abs(a(i,i+1:end)));
    s(i,1)=(u+v)/abs(a(i,i));
end
sf=max(s);
```

Wir vergleichen jetzt mal das Gesamt– mit dem Einzelschrittverfahren. Um die Situation der letzten Aussage von Satz 6.15 zu erhalten, brauchen wir eine Matrix, die das Zeilensummenkriterium erfüllt, und zwar so, dass wir die Norm der Iterationsmatrix des Gesamtschrittverfahrens kontrollieren können. Wir nehmen eine zufällige $n \times n$–Matrix `a` mit Einträgen aus

$[-1, +1]$, aber behalten uns eine Modifikation der Diagonale vor. Wie oben beschrieben, ist `ad=a-diag(diag(a))` die Matrix `a` mit Nulldiagonale, und `s=sum(abs(ad'))` ist der Vektor der absoluten Zeilensummen ohne Diagonale. Wenn wir diesen in die Diagonale schreiben würden, hätte die Iterationsmatrix des Gesamtschrittverfahrens die $\|.\|_{\infty,\infty}$–Norm Eins. Wir wollen aber Konvergenz des Gesamtschrittverfahrens, und deshalb vergrößern wir die Diagonale etwas, nämlich durch Multiplikation mit einer Zahl `1+ep`> 1 und einer komponentenweisen Multiplikation mit einem Zufallsvektor mit Elementen > 1, damit bei späteren Tests das Maximum der Zeilensummen nicht immer in der ersten Zeile angenommen wird.

Dann folgt je eine brutale Ausrechnung der Iterationsmatrizen des Gesamt– und Einzelschrittverfahrens, und es werden sowohl die Normen als auch die Spektralradien berechnet. Die Normen können nicht kleiner sein als der Spektralradius, aber dieser kann deutlich kleiner als die Normen sein. Das gilt besonders für den Sassenfeldwert, der ja nur eine (oft schlechte) obere Schranke für die Norm der Iterationsmatrix des Einzelschrittverahrens ist und oft den Spektralradius dramatisch überschätzt. Das kann man an diversen Beispielläufen schnell sehen.

```
clear all;
close all;
n=120;
% wir bauen eine Matrix mit erfuelltem Zeilensummenkriterium
a=2*rand(n,n)-1;
ad=a-diag(diag(a)); % Diagonale rausnehmen
ep=0.002 % Stoerungsterm: je groesser,
% desto besser die Konvergenz
s=sum(abs(ad'))*(1+ep).*(1+ep*rand(1,n));
% wir bauen eine Diagonale,
% die groessere Elemente als die Rest-Zeilensumme hat
a=ad+diag(s); % und schreiben sie rein
zsnorm=zsk(a) % Zeilensummennorm ausrechnen
% Ab hier werden die Iterationsmatrizen mal genauer untersucht
d=diag(diag(a));   % der Diagonalteil
cgsv=-inv(d)*(a-d);   % Iterationsmatrix GSV
% und der Spektralradius dazu
spektralradiusgsv=max(abs(eig(cgsv)))
R=a;   % wird der R-Teil
L=a;   % wird der L-Teil
for i=1:n
```

```
    for j=1:i
        L(j,i)=0;
        R(i,j)=0;
    end
end
cesv=-inv(d+L)*R;  % Iterationsmatrix des ESV
sass=sassenfeld(a) % dito Sassenfeld-Wert
spektralradiusesv=max(abs(eig(cesv)))
b=rand(n,1);  % Zufallsvektor
x=rand(n,1);  % Zufallsvektor
rlistgsv=[];    % Liste der Fehlernormen fuer gsv
rlistesv=[];    % Liste der Fehlernormen fuer esv
xg=x; % Iterierte des GSV
xe=x; % Iterierte des ESV
for i=1:200   % wir machen hier die Iteration:
    restg=b-a*xg;
    restgnorm=norm(restg);
    rlistgsv=[rlistgsv restgnorm];
    xg=gsvschritt(a, b, xg);
    reste=b-a*xe;
    restenorm=norm(reste);
    rlistesv=[rlistesv restenorm];
    xe=esvschritt(a, b, xe);
end
% und jetzt etwas Plotterei
semilogy(1:length(rlistgsv),rlistgsv,1:length(rlistesv),rlistesv)
legend('GSV','ESV')
title('Residuum')
xlabel('Iterationen')
ylabel('Residuum')
```

Das Programm macht dann bei gleichem zufälligem Startwert eine Anzahl von Iterationen und berechnet jeweils $\|Ax - b\|_2$ für die beiden Iterierten des Gesamt– und Einzelschrittverfahrens. Diese Werte sollten beim Iterieren gegen Null gehen, aber wie schnell? Wir plotten sie im halblogarithmischen Stil. Die Werte werden in dynamische Listen geschrieben, damit man nicht viel ändern muss, wenn man die Iterationsschleife verändert oder einen Abbruch einbaut. Die Listen werden leer initialisiert und dann um je einen Wert verlängert. Das ist nicht besonders effizient, aber pflegeleicht programmiert. Man sehe sich auch die Verschönerungen des Plots durch die vier letzten Kommandos an. Abbildung 6 ist ein Beispiel so eines Plots. Man

sieht, dass erwartungsgemäss das Einzelschrittverfahren das Gesamtschrittverfahren klar aussticht. Das ist sehr oft, aber nicht generell so. Weil sich Rundungsfehler nicht akkumulieren, bleiben die Fehler nach hinreichend vielen Iterationen einigermaßen stabil und schaukeln sich nicht auf.

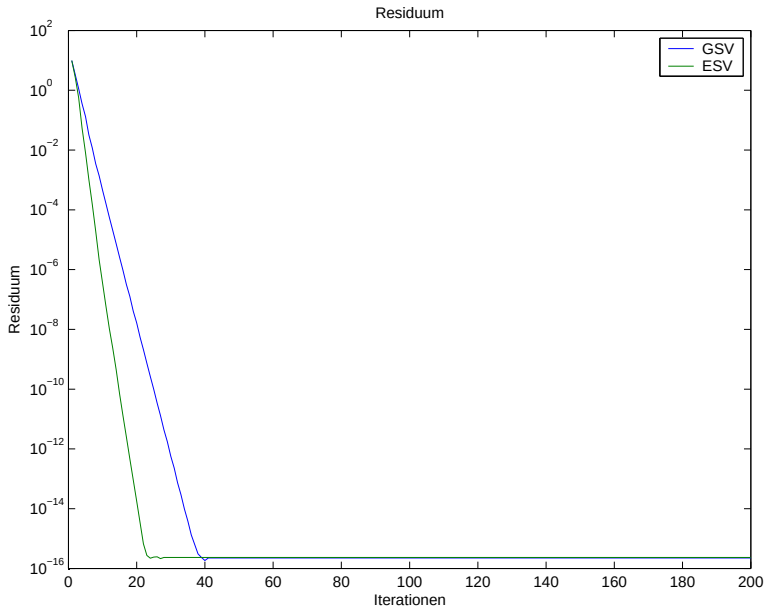

Abbildung 6: GSV–ESV–Test

Jetzt kommt noch die Relaxation des Einzelschrittverfahrens. Das Programm ist eine Variante des vorigen, aber statt der Iterierererei wird ein Plot des Spektralradius des relaxierten Einzelschrittverfahrens in Abhängigkeit vom Relaxationsparameter $\omega \in [0, 2]$ gemacht, und wir bauen uns eine positiv (semi–) definite symmetrische Matrix, denn wir wollen Satz 6.19 des Buches auf Seite 104 illustrieren. Hier haben wir das Auflisten der zu plottenden Werte nicht mit dynamischen Listen, sondern durch plattes Einsetzen in uninitialisierte Vektoren gemacht. Bei hardcore–Programmen macht man das natürlich nicht so hemdsärmelig...

```
clear all;
close all;
% Testen der Relaxation des ESV fuer pos.def. symm. Matrizen
n=37;
a=2*rand(n,n)-1;    % Zufallsmatrix mit Elementen aus [-1,1]
a=a'*a;             % symmetrisieren und pos. def. machen
% a=hilb(n);
```

```
D=diag(diag(a));   % der Diagonalteil
cgsv=-inv(D)*(a-D);   % Iterationsmatrix GSV
zsnorm=zsk(a) % zeilensummennorm GSV ausrechnen
% und der Spektralradius dazu
spektralradiusgsv=max(abs(eig(cgsv)))
R=a;   % wird der R-Teil
L=a;   % wird der L-Teil
for i=1:n
    for j=1:i
        L(j,i)=0;
        R(i,j)=0;
    end
end
cesv=-inv(D+L)*R;  % Iterationsmatrix des ESV
sass=sassenfeld(a) % dito Sassenfeld-Wert
format long g
spektralradiusesv=max(abs(eig(cesv))) % Spektralradius des ESV
% und jetzt testen wir m Relaxationsparameter
% im Intervall [0,2]
m=200;
for i=1:m
    omega=2*i/m;  % das werden unsere omega aus [0,2]
    olist(i)=omega;  % omega-Liste
    % Iterationsmatrix des ESV
    cesv=-inv(D+omega*L)*(D*(1-omega)-omega*R);
    spektralradiusesv=max(abs(eig(cesv)));
    slist(i)=spektralradiusesv;   % in Liste setzen
end
% und jetzt Plotterei
plot(olist,slist)
title('ESV-Relaxation')
xlabel('Relaxationsparameter Omega')
ylabel('Spektralradius der Iterationsmatrix')
% wir holen uns den besten Wert
[optimalwert imin]=min(slist)
opt_omega=2*imin/m
```

Abbildung 7 zeigt einen Beispiellauf. Der scharfe Abfall in einen Optimalwert zwischen 1 und 2 ist typisch. Nach Satz 6.19 muss die Kurve immer unterhalb 1 verlaufen.

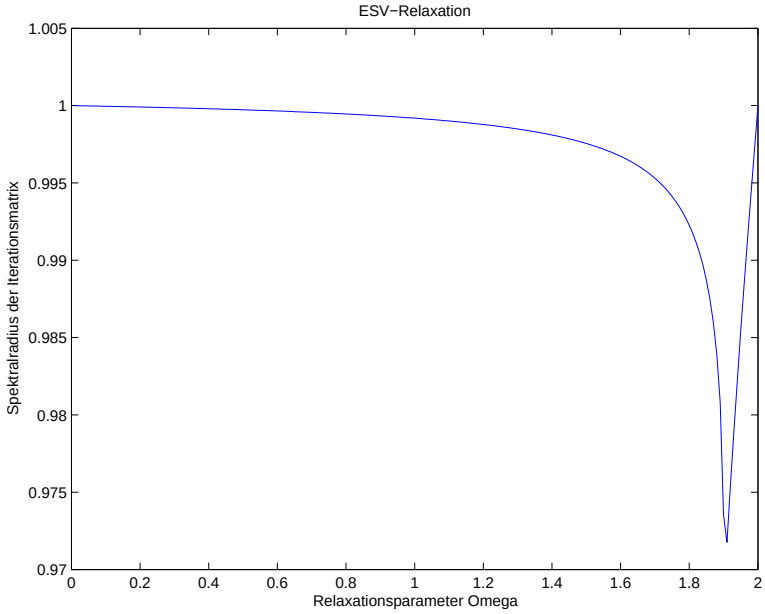

Abbildung 7: ESV–Relaxation

Aufgabe:
Implementieren sie das relaxierte Gesamtschritt- und Einzelschrittverfahren. Lösen Sie mit beiden Verfahren näherungsweise das Gleichungssystem aus Aufgabe 22 und plotten Sie die Lösung. Die Daten können frei gewählt werden, sollen aber austauschbar sein, indem sie über ein getrenntes m-file `fct.m` programmiert werden. Untersuchen den Einfluss des Relaxationsparameters und versuchen Sie diesen zu optimieren.

7 Newton–Verfahren

Aufgabe:
Implementieren Sie das Newton-Verfahren zur Bestimmung einer Nullstelle einer gegebenen Funktion $f : [a, b] \to I\!R$. Die Ableitung von f soll dabei wahlweise exakt angegeben werden oder durch einen einseitigen bzw. zentralen Differenzenquotienten approximiert werden.

Testen Sie das Programm mit der Funktion $f(x) = \sqrt{2 + x} - x$ und $x_0 = 0$. Vergleichen Sie die Konvergenz der drei Varianten anhand der Residuen $f(x_n)$ in einem Diagramm.

Das machen wir später. Hier ist aber zumindestens erst einmal ein schlichtes Vergleichsprogramm zwischen Fixpunktiteration und Newton–Verfahren.

Wir lösen die Gleichung

$$x = \cos(x)$$

die, wie eine einfache Zeichnung ergibt, genau eine Lösung in der Gegend um 0.74 besitzt. Aus Satz 6.3 des Buches folgt die lokale Konvergenz der Fixpunktiteration $x_{i+1} = \cos(x_i)$, wenn dicht bei 0.74 gestartet wird. Mit

$$f(x) := x - \cos(x)$$

bekommt man das Newtonverfahren als

$$x_{i+1} = x_i - \frac{x_i - \cos(x_i)}{1 + \sin(x_i)}.$$

Beides wird unten in einer Schleife 15 mal wiederholt, mit dem Startwert 1. Den Plot der absoluten Fehler in den x–Werten zeigt Abbildung 8. Man sieht die lineare Konvergenz der Fixpunktiteration, und bei etwas gutem Willen auch die quadratische des Newton–Verfahrens, aber wie so oft braucht das Newton–Verfahren nur 5 Schritte um die Maschinengenauigkeit zu erreichen.

```
clear all;
close all;
% Programm zur Loesung von x=cos(x)
xn=1; % Startwert Newton
xf=1; % Startwert Fixpunktiteration
xr=0.739085133215161; % wahre Loesung
xnlist=[xn-xr];   % Fehlerliste Newton
xflist=[xf-xr];   % Fehlerliste Fixpunkt
for i=1:15
    xn=xn-(xn-cos(xn))/(1+sin(xn)); % Newton
    xnlist=[xnlist xn-xr];
    xf=cos(xf);  % Fixpunktiteration
    xflist=[xflist xf-xr];
end
% Listenausgabe
format long
Newton_____________Fixpunktiteration=...
[xnlist' xflist']
% Plotausgabe
semilogy(1:length(xnlist),abs(xnlist),1:length(xflist),abs(xflist))
legend('Newton-Verfahren','Fixpunktiteration',3)
title('Loesung von x=cos(x)')
xlabel('Iterationen')
ylabel('Fehler im X-Wert')
```

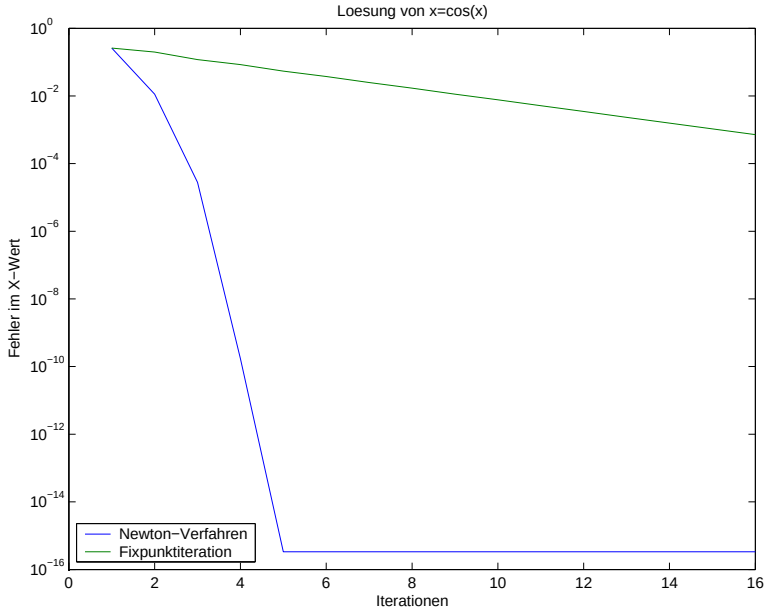

Abbildung 8: Newtonverfahren und Fixpunktiteration

Wir haben schon gelernt, daß das Newtonverfahren nur lokal konvergiert. Bei Problemen mit mehreren Lösungen wird es in je einer Umgebung je einer Lösung gegen genau diese konvergieren, wenn die üblichen lokalen Voraussetzungen gegeben sind. Aber was passiert bei beliebigen Startwerten? Ist wie bei der Mandelbrotmenge aus dem vorigen Kapitel die Grenze zwischen Konvergenz und Nicht–Konvergenz problematisch?

Zur Illustration nehmen wir die Gleichung

$$f(z) = z^3 - 1$$

im Komplexen. Sie hat die drei Einheitswurzeln als Lösungen, die zu der Teilung des Einheitskreises in 3 gleiche Teile mit Innenwinkel 120 Grad gehören. Neben der 1 sind deshalb auch (etwa) $-0.5 \pm i * 0.866$ Lösungen.

Es ist nicht schwer zu verifizieren, daß das Newtonverfahren im Komplexen genauso funktioniert wie im Reellen, und deshalb kann man es relativ simpel programmieren. Wir machen das in dem folgenden Programm innerhalb der `while`–Schleife, aber wir wollen feststellen, gegen welche Wurzel das Verfahren konvergiert, wenn es überhaupt konvergiert. Wir geben deshalb zu gegebenem Startwert nur einen Index zurück, der genau dies angibt.

```
function index=newtonkubcomplex(z)
```

```
% gibt Index der Limeswurzel aus
% 1 fuer (1,0)
% 2 fuer (-0.5,+0.866)
% 3 fuer (-0.5,-0.866)
while abs(z^3-1)>1.0e-12
    z=z-(z^3-1)/(3*z^2);
end
index=0;
if (abs(z-1)<1.0e-6)
    index=1;
end
if abs(z-(-0.5+i*0.866025403))<1.0e-6
    index=2;
end
if abs(z-(-0.5+-i*0.866025403))<1.0e-6
    index=3;
end
```

Dieses Rechenprogramm bauen wir in ein Malprogramm ein:

```
clear all;
close all;
% Programm zur Loesung von z^3=1 im Komplexen
a=1; % Intervallgrenze um Null
h=a*0.001;  % relative Schrittweite
[xp yp]=meshgrid(-a-h/2:h:a+h/2,-a-h/2:h:a+h/2); % Gittererzeugung
xpv=xp(:);
ypv=yp(:);
ipv=zeros(size(xpv));  % hier kommen die Indizes rein
for j=1:length(xpv)
    ipv(j)=newtonkubcomplex(xpv(j)+i*ypv(j)); %  Grenzwerte berechnen
end
ip=reshape(ipv,size(xp));  % zuruecksortieren
image(ip*20)               % brutale Farbwahl
axis square
title('Einzugsbereiche der drei dritten Einheitswurzeln')
```

Man sieht, daß es sich um eine vereinfachte Adaptation des Programms zur Mandelbrotmenge handelt. Weil der Nenner des Newtonverfahrens $f'(z) = 3z^2$ in der Null verschwindet, muß man das Gitter so bauen, daß die Null vermieden wird. Das Ergebnis zeigt Abbildung 9. Man könnte die Farbgebung wesentlich verbessern, indem man nicht nur die drei Grundfarben für die drei

Einzugsbereiche verwendet, sondern sie auch noch aufhellt oder verdunkelt, indem man die Schrittzahl bis zum Erreichen einer die lokale Konvergenz garantierenden Umgebung verwendet. Aber es sollte hier mit möglichst simplen Programmen gearbeitet werden.

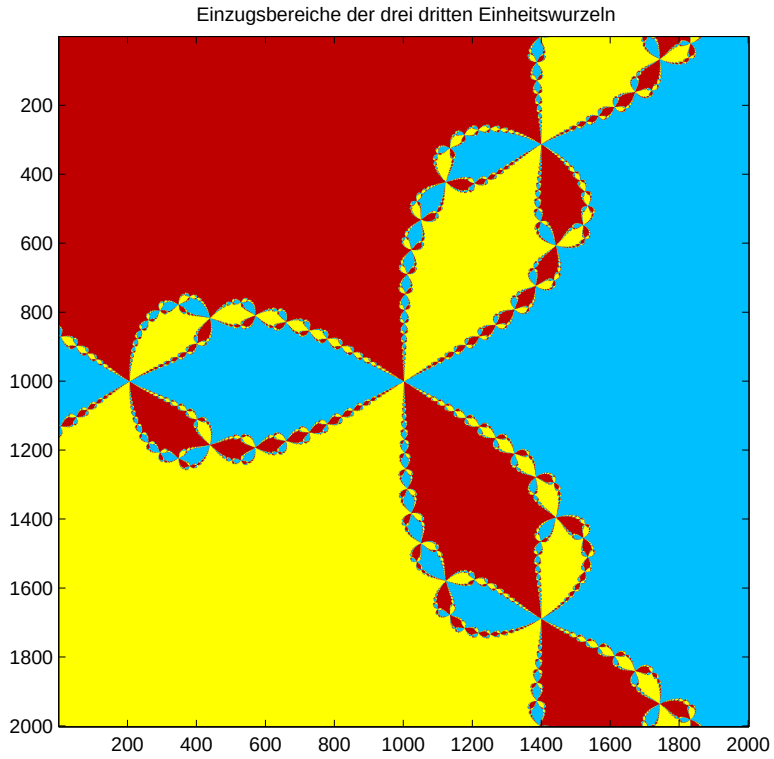

Abbildung 9: Einzugsgebiete der drei Lösungen

Es folgt ein Testprogramm zum Newton–Verfahren für Systeme. Wir packen die nichtlineare Abbildung in ein m–file `Fun.m`

```
function [F, DF] =Fun(x)
% Kombi-Routine fuer das Newtonverfahren,
% berechnet F(x) und die Jacobimatrix DF(x)
% Zwei Ellipsen
F(1,1)=   x(1)^2/1+x(2)^2/4-1;
F(2,1)=   x(1)^2/2+x(2)^2/6-1;
DF(1,1)=2*x(1)/1;
DF(1,2)=2*x(2)/4;
DF(2,1)=2*x(1)/2;
DF(2,2)=2*x(2)/6;
```

Dabei programmieren wir die Iteration so, daß sie den Schnitt zweier Ellipsen

$$\begin{aligned} \frac{x^2}{1} + \frac{y^2}{4} &= 1 \\ \frac{x^2}{4} + \frac{y^2}{1} &= 1 \end{aligned}$$

mit Halbachsen 1 und 2 berechnet, die sich notwendig in vier Punkten schneiden müssen. Das Ganze bauen wir in ein Hauptprogramm

```
clear all;
close all;
% Programm zur Loesung von F(x)=0, Newtonverfahren
% ohne Schrittweitensteuerung
for k=1:350    % wir rechnen viele Einzelfaelle
    xn=4*rand(2,1)-2;  % mit Zufallsstartwert in [-2,2]^2
    xnlist(1,:)=xn'; % hier sammeln wir die Iterationspunkte
    for i=1:1000  % Newtoniteration
        [f, df]=Fun(xn); % Funktionsaufruf
        z=df\f; % Newton-Inkrement
        xn=xn-z; % Update
        if norm(f)<1.0e-10  % Abbruch
            break;
        end
        xnlist(i+1,:)=xn'; % neuen Punkt aufheben
        maxx=i+1;
    end
    % figure
    plot(xnlist(:,1),xnlist(:,2)) % Punktefolge malen
    hold on           % alles uebereinander
    plot(xnlist(1,1),xnlist(1,2),'ro')
    hold on
    plot(xnlist(maxx,1),xnlist(maxx,2),'rx')
    hold on
    axis([-5 5 -5 5]) % mit fester Skalierung
end
title('Newton-Trajektorien ohne Schrittweitensteuerung')
```

von dem zunächst nur die innere Schleife angesehen werden sollte, denn sie ist das eigentliche Newtonverfahren. Aber wir wollen mal sehen, welche der vier Lösungen angesteuert werden, und deshalb verwenden wir 350 zufällige Startwerte in $[-2, 2]^2$, markieren sie mit einem roten Kreis und zeichnen den

Verlauf des Newtonverfahrens als Streckenzug auf, mit einem roten Kreuz am Ende.

Das Ergebnis ist Abbildung 10. Man sieht, daß das Verfahren oft ausbricht in Richtung auf große F–Werte (die Funktion ist komponentenweise eine positiv definite quadratische Form, steigt also gegen Unendlich quadratisch an). Sieht man sich die Jacobimatrix genauer an, so verschwindet ihre Determinante für $x = 0$ und $y = 0$, d.h. auf den Achsen, und bei Start in der Nähe der Achsen gibt es deshalb riesige Inkremente.

Macht man eine Schrittweitensteuerung (das Programm dazu unterdrücken wir erst noch), so bekommt man Abbildung 11. Jetzt können die Werte $\|F(x^j)\|$ nicht mehr fallen, und deshalb springt das Verfahren nicht mehr nach außen weg. Das Intervall $[-2, 2]^2$ wird (bis auf die Achsen) sauber in die vier Einzugsbereiche der vier Lösungen aufgeteilt.

In Abbildung 12 wurde dem Programm ein anderes m–file `Fun.m` untergejubelt, nämlich

```
function [F, DF] =Fun(x)
% Kombi-Routine fuer das Newtonverfahren,
% berechnet F(x) und die Jacobimatrix DF(x)
% Zwei Ellipsen
F(1,1)=   x(1)^2/1+x(2)^2/4-1;
F(2,1)=   x(1)^2/2+x(2)^2/6-1;
DF(1,1)=2*x(1)/1;
DF(1,2)=2*x(2)/4;
DF(2,1)=2*x(1)/2;
DF(2,2)=2*x(2)/6;
```

Hier haben wir zwei Ellipsen, die sich nicht schneiden, es gibt also keine Lösung des Problems. Das Newton–Verfahren mit Schrittweitensteuerung erzeugt eine Folge x^j mit monoton fallenden $\|F(x^j)\|$ und bewegt sich im wesentlichen auf Ellipsenbahnen auf die y–Achse zu. Dort ist aber das Newton–Verfahren undefiniert, und in der Nähe der y–Achse kriecht es mit winzigen Schrittweiten auf die Achse zu, ohne sie je genau zu erreichen.

8 Interpolation mit Polynomen

8.1 Polynomauswertung

In MATLAB werden Polynome p in der Monombasis durch `y=polyval(p,x)` ausgewertet. Dabei ist `p` ein Koeffizientenvektor, beginnend mit dem höchsten

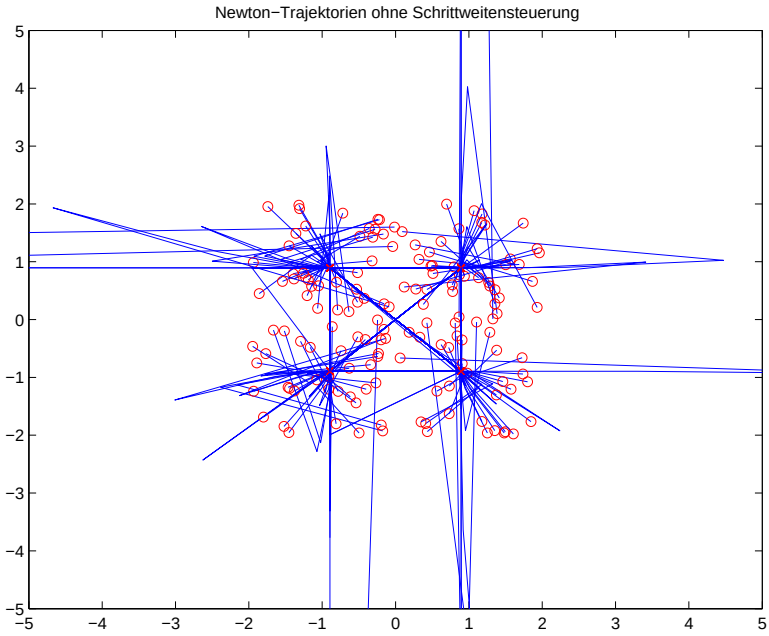

Abbildung 10: Newtonverfahren ohne Schrittweitensteuerung

Koeffizienten, und `x` ist ein Vektor der x–Stellen, an denen ausgewertet werden soll. Das Ergebnis ist ein Vektor $y = p(x)$ der Polynomwerte.

Polynominterpolation wird durch `p=polyfit(x,z,n)` durchgeführt. Dabei gibt `n` den Grad des gewünschten Polynoms an, die x–Werte stehen in `x` und die Funktionswerte in `y`. Das Ergebnis ist ein Koeffizientenvektor `p`, der sich direkt in `polyval(p,...)` einsetzen läßt, um das Polynom auszuwerten. Er beginnt deshalb mit dem höchsten Koeffizienten.

Was passiert, wenn man mehr als $n + 1$ Daten bei einem Polynomgrad n spezifiziert? Dann stellt MATLAB ein überbestimmtes lineares Gleichungssystem auf und macht lineare Ausgleichsrechnung wie in Kapitel 4. Hier ist eine kleine Demo, mit der Ausgabe in Abbildung 13:

```
clear all;
close all;
% Routine zur Illustration von polyval und polyfit
x=(-1:0.001:1);     % Diskretisierung des Intervalls [-1,1]
% z=exp(-x.*x);  % eine Funktion hernehmen
% z=abs(x);        % eine Funktion hernehmen
z=1./(1+25*x.*x);        % eine Funktion hernehmen
q=polyfit(x,z,155)  % sie durch Polynom q auf x fitten,
```

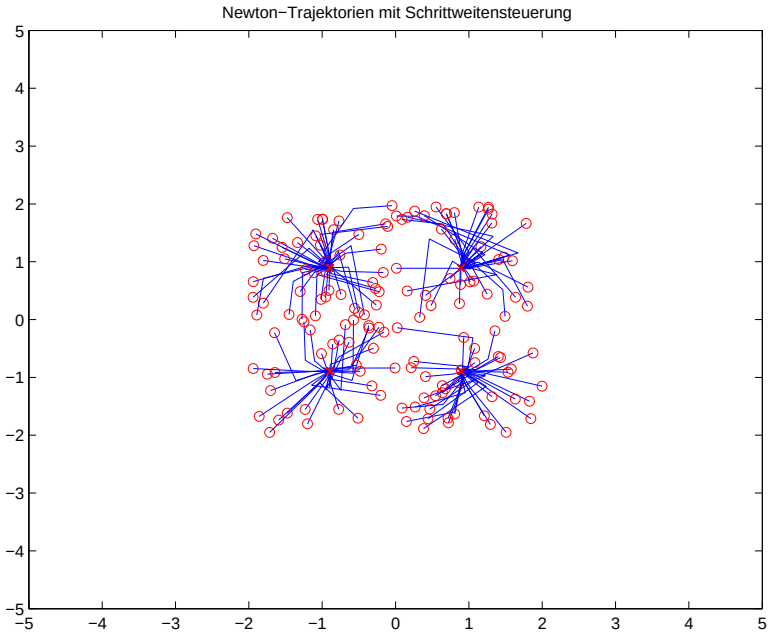

Abbildung 11: Newtonverfahren mit Schrittweitensteuerung

```
% q=mypolyfit(x,z,5); % sie durch Polynom q auf x fitten,
% der Polynomgrad steht im letzten Argument
subplot(2,1,1)   % erster Plot in einem 2x1 Schema
plot(x,z,x,polyval(q,x))
title('Funktion und Polynomapproximation')
legend('Funktion','Polynom')
subplot(2,1,2)  % zweiter Plot in einem 2x1 Schema
plot(x,z-polyval(q,x))
title('Fehler der Approximation')
```

Man sollte damit etwas herumspielen, verschiedene Polynomgrade und Daten von Funktionen unterschiedlicher Glätte nehmen. Es stellt sich heraus, daß so ein Fitting umso besser klappt, je glatter die Funktion ist, die die Daten liefert, aber bei großem Polynomgrad wird das Ganze schnell instabil.

Als Übungsaufgabe könnte man mal eine eigene Version von `p=polyfit(x,z,n)` erstellen, die dasselbe tut.

8.2 Tschebyscheffpolynome

Hier ist ein kleines Demoprogramm zur Berechnung von Tschebyscheffpolynome, mit Ausgabe in Abbildung 14. Man sollte das durch eine Routine

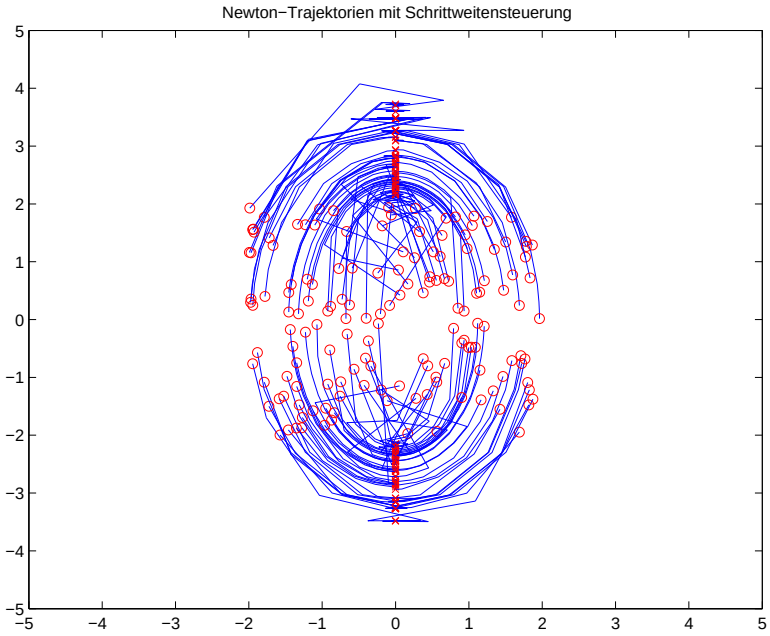

Abbildung 12: Newtonverfahren bei unlösbarem Problem

ersetzen, die keine trigonometrische Funktionen benutzt, sondern stattdessen die Rekursionsformel.

```
clear all;
close all;
n=55; % maximaler Grad
x=(-pi:0.001:0)'; % damit cos(x) in [-1,1] liegt, als Spalte
cx=cos(x);
for i=1:n+1
    val=cos(x*(i-1));
    plot(cx,val); % wir plotten cos((i-1)*x) gegen cos(x)
    hold on
end
title('Tschebyscheff-Polynome')
```

Aufgabe:
Implementieren sie die Polynominterpolation durch naives Lösen des zugehörigen linearen Gleichungssystems. Schreiben Sie dazu eine Funktion `yi = interp(x,y,xi)`, die einen Stützstellenvektor `x`, Funktionswerte in den Stützstellen `y` und eine weitere Stützstellenmenge `xi` als Argumente bekommt, und die interpolierten Werte `yi` an den Stellen `xi` berechnet. Ver-

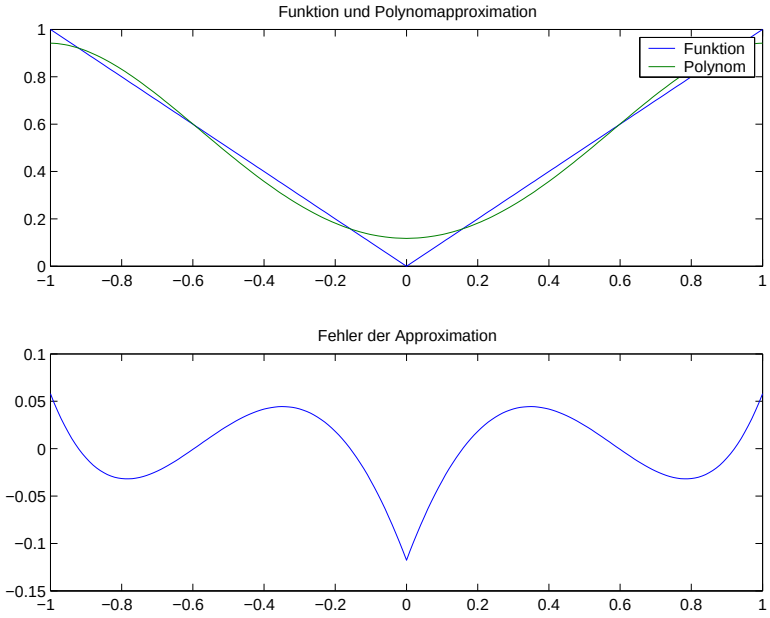

Abbildung 13: Polynomapproximation

wenden Sie einmal die Monome und einmal die Tschebyscheffpolynome als Basis. Inwiefern ist das Verfahren von der Wahl der Basis abhängig und warum würde man praktisch nicht auf diese Weise vorgehen?

Hier ist eine schlichte Lösung:

```
function yi=interpol(x,y,xi)
% naive Polynominterpolation
n=length(x);
ni=length(xi);
mat=zeros(n,n);
mat(:,1)=ones(n,1);
for i=2:n
    mat(:,i)=mat(:,i-1).*x;
end
co=cond(mat)
coeff=mat\y;
mati=zeros(ni,n);
mati(:,1)=ones(ni,1);
for i=2:n
    mati(:,i)=mati(:,i-1).*xi;
end
```

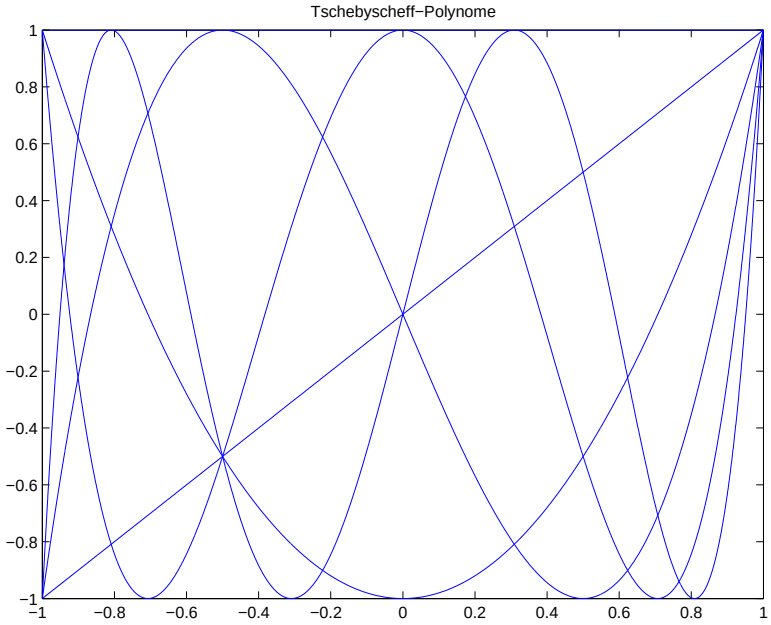

Abbildung 14: Tschebyscheffpolynome

```
yi=mati*coeff;
```

Jetzt dasselbe mit Tschebyscheff-Polynomen:

```
function yi=interTPol(x,y,xi)
% Polynominterpolation, Tschebyscheff-Basis
n=length(x);
ni=length(xi);
mat=zeros(n,n);
mat(:,1)=ones(n,1);
if n>1
    mat(:,2)=x;
end
for i=3:n
    mat(:,i)=2*mat(:,i-1).*x-mat(:,i-2);
end
co=cond(mat)
coeff=mat\y;
mati=zeros(ni,n);
mati(:,1)=ones(ni,1);
if n>1
    mati(:,2)=xi;
```

```
end
for i=3:n
    mati(:,i)=2*mati(:,i-1).*xi-mati(:,i-2);
end
yi=mati*coeff;
```

Aufgabe:
Implementieren Sie die Polynominterpolation nach dem Schema von Neville und Aitken. Verwenden Sie den gleichen Funktionsprototypen `yi = interp(x,y,xi)` wie in Aufgabe 36, so dass beide Funktionen leicht ausgetauscht werden können. Wie verhalten sich die Laufzeiten beider Implementierungen hinsichtlich des Polynomgrads bzw. der Anzahl der auszuwertenden Stellen?

Deshalb jetzt dasselbe mit Neville-Aitken:

```
function yi=interpNA(x,y,xi)
% Neville-Aitken Polynominterpolation
n=length(x)
ni=length(xi)
mati=zeros(ni,n);
for i=1:n
    mati(:,i)=y(i);
end
% Invariante: mati(:,i) interpoliert in xi(i-k),...xi(i).
% Gilt zuerst fuer k=0, dann fuer alle k
for k=1:n-1
    for i=n:-1:k+1
        mati(:,i)=((xi-x(i-k)).*mati(:,i)+(x(i)-xi).*mati(:,i-1))...
            /(x(i)-x(i-k));
    end
end
% Ergebnis fuer k=n-1, i=n.
yi=mati(:,n);
```

Wir sollten jetzt aber noch Beispiele zum Interpolationsfehler zeigen, die sich mit den obigen Routinen leicht herstellen lassen. In allen Fällen verwenden wir als gute Göttinger Lokalpatrioten die Runge-Funktion $1/(1+25x^2)$. Abbildung 15 zeigt den äquidistanten Fall, während Abbildung 16 in den Nullstellen und Abbildung 17 in den Extremstellen des passenden Tschebyscheff-Polynoms interpoliert. Die verwendete Routine ist je eine Variante von

```
clear all;
```

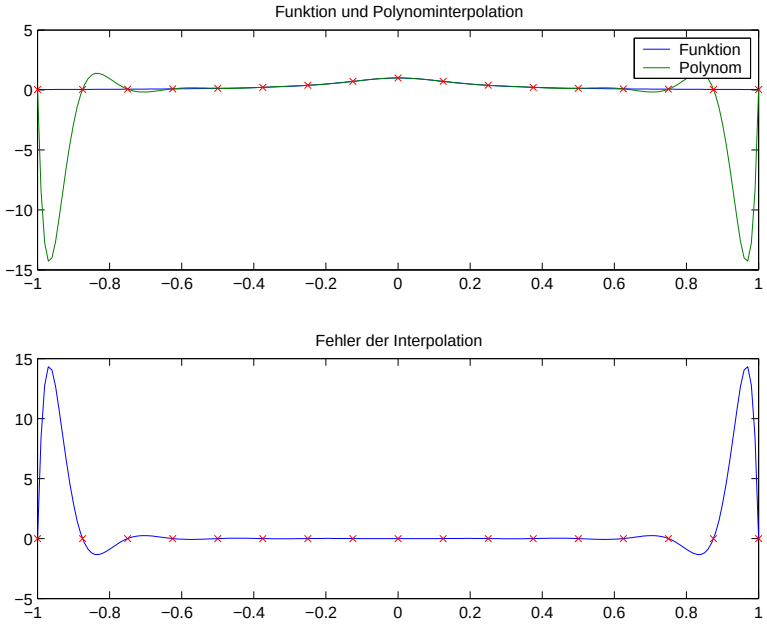

Abbildung 15: Äquidistante Punkte

```
close all;
% Routine zur Illustration von polyval und polyfit, Interpolation
m=16;
xfein=-1:0.01:1;    % Diskretisierung des Intervalls [-1,1] zum Plotten
xdata=-1:2/m:1;     % Diskretisierung des Intervalls [-1,1] aequidistant
% durch m+1 Punkte, also wird Grad = m
% xdata=cos(-pi:pi/m:0); % Tschebyscheff-Extremstellen, BESSER!!!
% xdata=cos((pi*(2*(0:m)+1))/(2*m+2)); % Tschebyscheff-Nullstellen, BESSER!!!
% z=abs(xdata);        % eine Funktion hernehmen, Daten berechnen
% zfein=abs(xfein);    % und die feinen Werte zum Testen
% z=max(0,xdata.^3);        % eine Funktion hernehmen, Daten berechnen
% zfein=max(0,xfein.^3);    % und die feinen Werte zum Testen
z=1./(1+25*xdata.^2);        % eine Funktion hernehmen, Daten berechnen
zfein=1./(1+25*xfein.^2);        % eine Funktion hernehmen
q=polyfit(xdata,z,m) % Daten durch Polynom q auf x fitten,
% der Polynomgrad steht im letzten Argument
subplot(2,1,1)   % erster Plot in einem 2x1 Schema
plot(xfein,zfein,xfein,polyval(q,xfein),xdata,z,'rx')
title('Funktion und Polynominterpolation')
legend('Funktion','Polynom')
subplot(2,1,2)  % zweiter Plot in einem 2x1 Schema
```

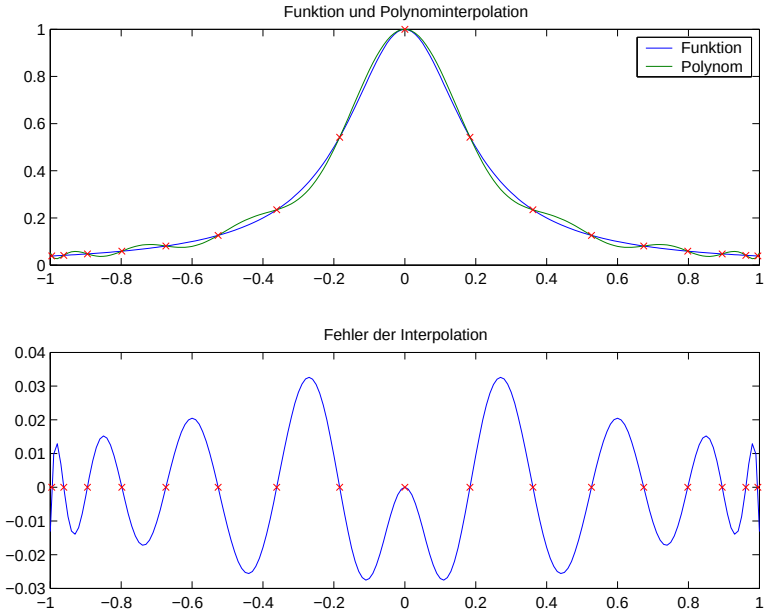

Abbildung 16: Tschebyscheff-Nullstellen

```
plot(xfein,zfein-polyval(q,xfein),xdata,zeros(size(xdata)),'rx')
title('Fehler der Interpolation')
```

9 Numerische Integration

9.1 Interpolationsquadraturen

Wir beginnen mit einem einfachen Programm, das geschlossene Newton-Cotes-Formeln ausrechnet. Die Idee ist simpel, denn es werden einfach die Gleichungen gelöst, die Exaktheit für ausreichend viele Monome fordern.

```
clear all;
close all;
% Routine zur Illustration von Interpolationsquadraturen in [-1,1]
m=58;
%  xdata=-1:2/m:1;     % Diskretisierung des Intervalls [-1,1] aequidistant
% durch m+1 Punkte, also wird Grad = m
% xdata=cos(-pi:pi/m:0); % Tschebyscheff-Extremstellen, BESSER!!!
xdata=cos((pi*(2*(0:m)+1))/(2*m+2)); % Tschebyscheff-Nullstellen, BESSER!!!
mat=zeros(m+1,m+1);
rhs=zeros(m+1,1);
```

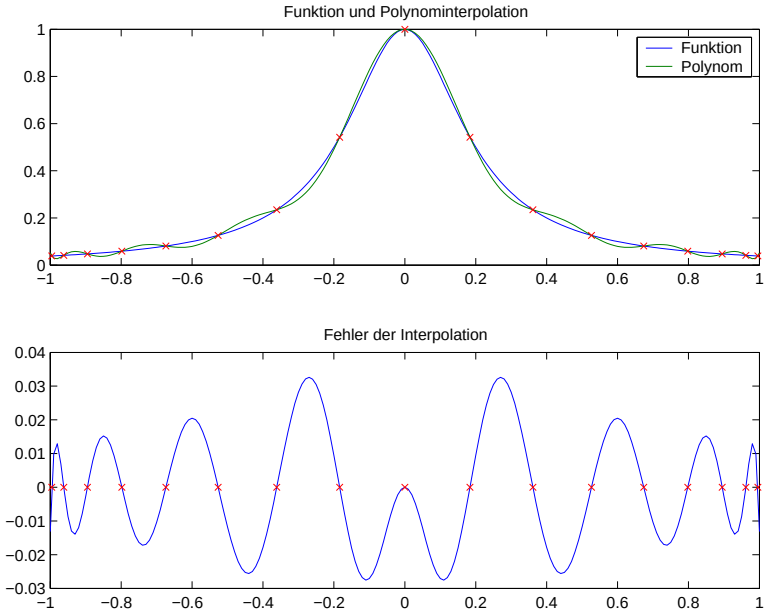

Abbildung 17: Tschebyscheff-Extremstellen

```
for j=1:m+1
    mat(j,:)=xdata.^(j-1);
    if mod(j,2)==1
        rhs(j,1)=2/j;
    else
        rhs(j,1)=0;
    end
end
coef=mat\rhs
plot(xdata,coef,'ro',xdata,zeros(m+1,1),'.')
title('Gewichte der geschlossenen Newton-Cotes-Formel in [-1,+1]')
legend('Gewichte','Stuetzstellen')
```

Man kann verschiedene Punktwahlen einsetzen. Ein Ergebnis für den Grad 8 bei äquidistanten Stützstellen ist Abbildung 18. Man sieht, dass negative Gewichte auftreten.

Nimmt man die Nullstellen des Tschebyscheff-Polynoms T_8, so sind die Gewichte viel besser, siehe Abbildung 19.

9.2 Gauss-Quadratur

Ein Programm zum Zeichnen der Legendre-Polynome fehlt noch...

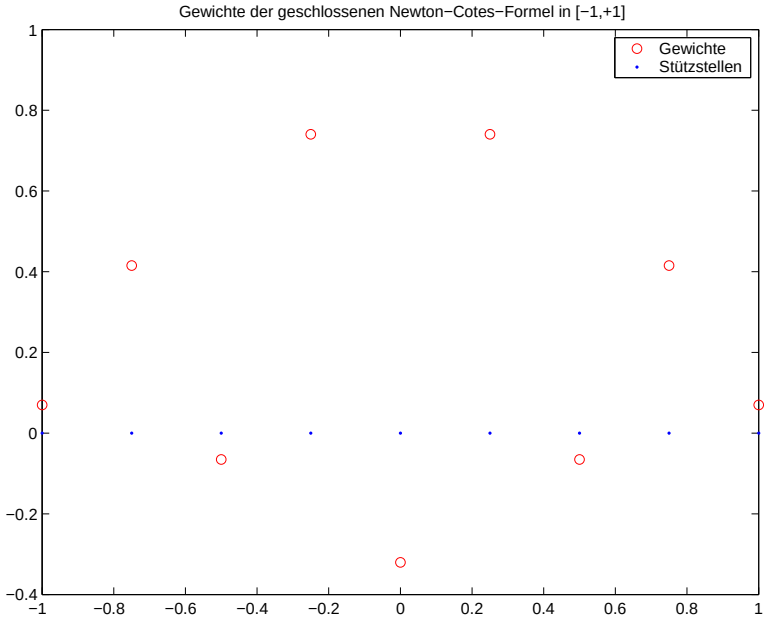

Abbildung 18: Äquidistante Knoten, Newton-Cotes-Gewichte

Hier geben wir nur ein kleines Programm an, das die Integration in den Nullstellen der Tschebyscheff-Polynome ausführt. Mit den n Nullstellen von T_n in $[-1,1]$ ist die einfache Formel

$$\int_{-1}^{1} \frac{g(x)}{\sqrt{1-x^2}} dx \approx \frac{\pi}{n} \sum_{i=1}^{n} g(x_i).$$

exakt für Polynome bis zum Grad $2n-1$. Hier sind alle Gewichte gleich groß, und es handelt sich um die Gaußquadratur zur Gewichtsfunktion $\frac{1}{\sqrt{1-x^2}}$. Das liefert auch eine nicht allzu schlechte Formel

$$\int_{-1}^{1} f(x) dx \approx \frac{\pi}{n} \sum_{i=1}^{n} f(x_i) \sqrt{1-x_i^2}$$

auf $[-1,1]$ mit dem folgenden Programm:

```
clear all;
close all;
% Wir testen die Tschebyscheff-Integration
a=-1;
b=1;
n=1024; % Anzahl der Punkte
```

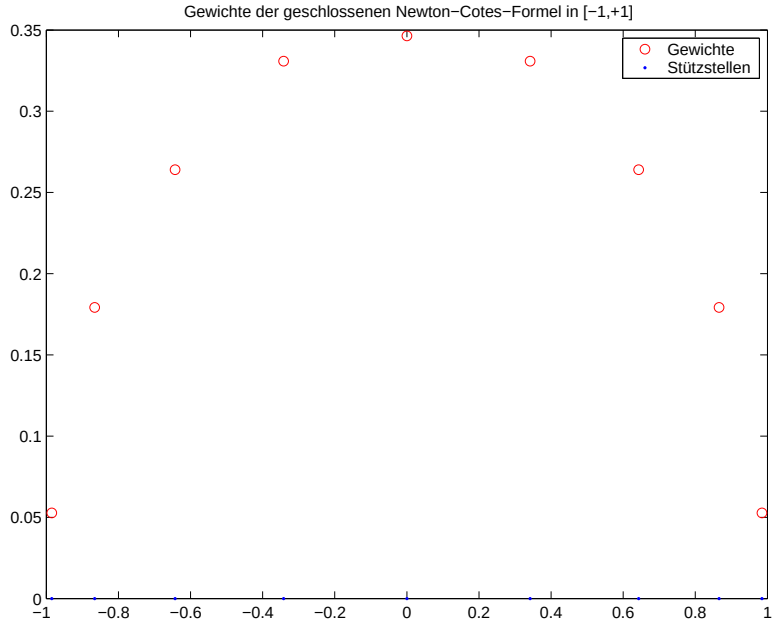

Abbildung 19: Tschebyscheff-Knoten, Newton-Cotes-Gewichte

```
c=pi/n; % das konstante Gewicht
x=cos((2*(1:n)-1)*pi/(2*n)); % die Nullstellen von T_n
fx=fct(x).*sqrt(1-x.*x);  % wir mï¿½ssen f korrigieren
format long g
val=sum(fx')*c            % die Integrationsformel
2*pi                      % der Sollwert, wenn fct=4./(1+x.*x)
```

Wir integrieren in den folgenden Beispielen $4/(1+x^2)$ mit der Stammfunktion $4\arctan(x)$. Wegen $\pi/4 = \arctan(1)$ ist dann das exakte Integral über $[0,1]$ gleich π und das Integral über $[-1,1]$ gleich 2π. Alle Integrationsformeln sind dann mehr schlecht als recht geeignet, π auszurechnen. Mit obigem Programm bekommt man 6.28318687591009 als Näherung für $2\pi = 6.28318530717959$, wobei

```
function val=fct(x)
% val=ones(size(x));
% val=x;
val=4./(1+x.^2);
```

verwendet wurde.

9.3 Zusammengesetzte Trapezregel und Romberg-Integration

Die zusammengesetzte Trapezregel wird implementiert in

```
function val=trapezint(fct,a,b,n)
h=(b-a)/n;
x=a:h:b;
f=fct(x)*h;
f(1)=f(1)/2;
f(end)=f(end)/2;
val=sum(f');
```

und getestet mit

```
a=-1;
b=1;
n=8;
trapezint(@cos,a,b,n)
```

auf dem Kosinus. Man kann das dann in eine primitive Routine für die Romberg-Integration einbauen:

```
clear all;
close all;
% Programm fuer das einfachste Romberg-Verfahren
a=0; % linke Intervallgrenze
b=1; % rechte Intervallgrenze
n=5; % Schrittzahl
m=1; % Wir starten mit 2 Punkten
T=zeros(n,n); % das wird die Dreiecksmatrix mit dem Rombergschema
% Natuerlich kann man das besser machen.....
for k=1:n % Beschaffung der Startwerte, naiv programmiert
    T(k,1)= trapezint(@fct,a,b,m);
    m=2*m;   % Punktverdopplung, d.h. h-Halbierung
end
% jetzt das Rombergschema (9.19) auf Seite 169
for k=1:n-1
    for j=k+1:n
        T(j,k+1)=T(j,k)+(T(j,k)-T(j-1,k))/(4^k-1)
    end
end
format long
T % Ausgabe des Rombergschemas in Fixkommazahlen
save('Romberg.txt','T','-ascii')
```

mit dem Ergebnis

```
3.0000000e+00   0.0000000e+00   0.0000000e+00   0.0000000e+00   0.0000000e+00
3.1000000e+00   3.1333333e+00   0.0000000e+00   0.0000000e+00   0.0000000e+00
3.1311765e+00   3.1415686e+00   3.1421176e+00   0.0000000e+00   0.0000000e+00
3.1389885e+00   3.1415925e+00   3.1415941e+00   3.1415858e+00   0.0000000e+00
3.1409416e+00   3.1415927e+00   3.1415927e+00   3.1415926e+00   3.1415927e+00
```

als Approximation an π, wenn man `fct` in $[0, 1]$ integriert. Man sieht, wie sich die Genauigkeit erst durch die Richardson-Extrapolation steigert.

Aufgabe:
Für praktische Berechnungen von Integralen wird man zunächst das Intervall zerlegen und dann auf die Teilintervalle Quadraturformeln anwenden. Schreiben Sie eine Funktion zur numerischen Integration die so vorgeht. Die Quadraturformel (Stützstellen und Gewichte) sollen dabei bezüglich des Einheitsintervalls angegeben werden und entsprechend auf die Teilintervalle transformiert werden.

Berechnen Sie numerisch $\int_0^\pi \sin(x)\, dx$ mit der Quadraturformel aus Aufgabe 43 für verschieden feine Zerlegungen des Intervalls $[0, \pi]$ und stellen Sie den Fehler grafisch dar. Vergleichen Sie das Ergebnis mit der zusammengesetzten Trapezregel.

Literatur

[1] Schaback, R., and Wendland, H. *Numerische Mathematik*. Springer-Lehrbuch. Springer Verlag, 2004. Fifth edition.