Technische Universität Hamburg-Harburg
Institut für Numerische Simulation E-10
Dr. Jens-Peter M. Zemke

Sommersemester 2007

Numerische Verfahren

Übungen und Lösungen, Blatt 0

Lesen Sie bitte den folgenden Text zur Vorbereitung auf die erste Übung. Es kann nicht schaden, dabei auch selbst schon mit Matlab zu experimentieren, sofern sie dieses zur Hand haben. Diese Ausführungen sind eine leicht überarbeitete Version eines Textes von Herrn Dr. Menck aus dem Arbeitsbereich Mathematik, welcher im Sommersemester 2000 verfaßt wurde und wiederum zu einem beträchtlichen Teil auf einem Text von Herrn Prof. Rump beruht, der sich seinerseits auf den **Matlab User's Guide** stützt. Dieser Text ist nur als Einführung gedacht, für nähere Informationen verweisen wir auf den erwähnten **Matlab User's Guide** und auf umfangreiche Literatur im Internet.

1 Was ist Matlab?

Matlab (die Bezeichnung Matlab steht für **Mat**rix **Lab**oratory) ist ein Werkzeug für das numerische Rechnen und zur Visualisierung von numerischen Problemen. Es vereint numerische Analysis, Matrizenrechnung, Signalverarbeitung und Graphik in einer Umgebung, die die Formulierung von Problemen und Lösungen in einer Notation nahe üblicher mathematischer Notation erlaubt.

2 Grundlegendes

Unter anderem ist Matlab eine interaktive Programmiersprache. In diesem Abschnitt werden der grundlegende Datentyp, Elemente zur Steuerung des Programmierflußes und wichtige Funktionen auf den erwähnten Datentypen kurz umrissen.

2.1 Matrizen

Der „klassische" Datentyp in Matlab ist eine *Matrix* mit reellen oder komplexen Elementen. Dabei werden die 1×1–Matrizen als Skalare interpretiert, Vektoren sind in Analogie dazu gegeben als $n \times 1$ und $1 \times n$–Matrizen.

Kleine Matrizen können in expliziten Listen eingegeben werden, die die folgende Form haben:

- Elemente der Liste werden durch Leerzeichen oder Komma getrennt.
- Elemente einer Matrix werden mit eckigen Klammern umgeben.
- Das Ende einer Matrixzeile wird durch ein Semikolon oder durch einen Zeilenwechsel angegeben.

So erzeugt die Eingabe der Kommandozeile

```
A = [1 2 3; 4 5 6; 7 8 9]
```

die Ausgabe

```
A =

     1     2     3
     4     5     6
     7     8     9
```

Matlab speichert die Matrix `A` für den späteren Gebrauch.

2.2 Matrixelemente

Matrixelemente können jeder Matlab–Ausdruck sein. Die Eingabe

```
x = [-1.3 sqrt(3) (1+2+3)*4/5]
```

ergibt das Resultat

```
x =

   -1.3000    1.7321    4.8000
```

Einzelne Matrixelemente werden durch Indizes in runden Klammern referenziert. Mit dem obigen Beispiel ergibt

```
x(5) = abs(x(1))
```

die Ausgabe

```
x =

 -1.3000  1.7321  4.8000  0  1.3000
```

Wie Sie sehen können, wird die Länge von `x` automatisch erhöht, um das neue Element aufzunehmen; die dazwischenliegenden noch undefinierten Elemente werden mit `0` initialisiert.

Große Matrizen können auch aus kleinen Matrizen gebildet werden. So wird die Matrix `A` aus dem obigen Beispiel folgendermaßen um eine Zeile erweitert:

```
r = [10 11 12];
A = [A; r]
```

Dies ergibt die Ausgabe

```
A =

     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

Teilmatrizen kann man durch Angabe der gewünschten Zeilen- und Spaltenindizes referenzieren: So bezeichnet

```
A([1 3],[1 2])
```

die (Unterabschnitts-)Matrix

```
ans =

     1     2
     7     8
```

Bei größeren, gleichmäßig aufgebauten Indexvektoren ist die Verwendung des Doppelpunkt–Operators nützlich, der einer typischen Laufanweisung entspricht. So erzeugt z.B.

```
3:10
```

einen Zeilenvektor mit den Zahlen

```
ans =

   3   4   5   6   7   8   9  10
```

Will man nicht in Einerschritten laufen, so kann man eine Schrittweite als zweiten Parameter einfügen. So ergibt

```
3:2:10
```

beispielsweise den Vektor

```
ans =

     3     5     7     9
```

Bei der Angabe von Indizes bedeutet ein Doppelpunkt ohne umgebende Zahlen den gesamten Bereich. Mit

```
A(1:3,:)
```

sind zum Beispiel die ersten drei Zeilen und alle Spalten der Matrix gemeint, also die ursprüngliche Matrix `A`.

2.3 Anweisungen, Variablen und Funktionen

Matlab interpretiert und evaluiert eingegebene Ausdrücke. Die typische Form einer Matlab–Anweisung hat die Form

```
Variable = Ausdruck
```

oder einfach

```
Ausdruck
```

Ausdrücke können hierbei aus Operatoren, speziellen Zeichen, Funktionen und Variablennamen gebildet werden. Wird ein Ausdruck nicht an eine Variable mit `=` zugewiesen, so erzeugt Matlab eine Variable mit Namen `ans`. Wird eine Anweisung mit Semikolon abgeschlossen, so findet keine Ausgabe des Ergebnisses, wohl aber eine Evaluierung statt. Das Unterdrücken der Ausgabe macht Sinn bei Zwischenausdrücken, die eine längere Ausgabe erzeugen.

Matlab unterscheidet zwischen Groß– und Kleinschreibung. Die Variablen `A` und `a` sind *nicht* dieselben Variablen. Alle Funktionsnamen müssen hingegen klein geschrieben werden.

Ausdrücke können mit den üblichen arithmetischen Operatoren und Vorrangsregeln gebildet werden. Dabei stehen die folgenden Operatoren zur Verfügung:

`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	rechtsseitige Division
`\`	linksseitige Division
`^`	Potenz

Für Matrixoperationen ist es nützlich, zwei Symbole für die Division bereitzustellen. Siehe dazu den Abschnitt über Matrixoperationen. Matlab stellt standardmäßig elementare mathematische Funktionen wie z.B. `abs`, `sqrt`, `log` und `sin` bereit. Weitere Funktionen können mit Hilfe von sogenannten M–Files hinzugefügt werden, vgl. den folgenden Abschnitt.

Matlab–Funktionen können miteinander kombiniert werden. Die Eingabe

```
x = sqrt(log(x));
```

zeigt die Verwendung zweier ineinander geschachtelter Funktionen. Zwei oder mehr Funktionsargumente sind möglich:

```
theta = atan2(y,x);
```

Jedes Argument kann hierbei auch ein Ausdruck sein.

Einige Funktionen liefern zwei oder mehr Resultate zurück. Die Resultatwerte werden in eckige Klammern eingeschlossen und durch Kommata getrennt:

```
[V,Lambda] = eig(A);
```

Matlab verändert niemals Funktionsargumente. Das Funktionsresultat wird immer an die linke Seite einer Zuweisung zurückgegeben.

2.4 M–Files; Kontrollfluß

M–Files sind Dateien mit der Endung `.m`, die Matlab–Kommandos enthalten. Man ruft sie durch Eingabe des Namens ohne Endung auf. Damit M–Files gefunden werden, müssen sie im aktuellen Suchpfad liegen, den man mit dem Befehl `path` anzeigen kann. Ergänzungen lassen sich mit dem Befehl `addpath` vornehmen.

Es ist zu unterscheiden zwischen M–Files vom *Prozedurtyp*, sogenannte Skripte, die lediglich eine Abfolge von Kommandos bündeln, und solchen vom *Funktionstyp*. Die letzteren können mit Parametern aufgerufen werden und Parameter zurückgeben. Sie legen einen eigenen Satz interner Variable an, auf die von der Kommandozeilenebene nicht zugegriffen werden kann. Beispiel: Die Datei `fibfun.m` mit dem Inhalt

```
function ret = fibfun(n)
if n > 2
   ret = fibfun(n-1) + fibfun(n-2);
else
   ret = 1;
end;
```

berechnet (rekursiv) Fibonacci–Zahlen. Die zehnte Fibonacci–Zahl wird z.B. als

```
fib10 = fibfun(10);
```

abgerufen. So würde man das Programm aber selbstverständlich nicht implementieren. Geeigneter wäre eine nicht-rekursive Ausarbeitung.

Im Beispiel erkennt man, daß Matlab auch Sprachelemente bietet, die wie in Programmiersprachen den Fluß des Programms zu kontrollieren gestatten, wie `if`–Abfragen, `for`– und `while`–Schleifen und einiges mehr. Ein Beispiel für eine `for`–Schleife ist

```
for i = 1:n, x(i) = 0; end
```

Dieses Beispiel setzt die ersten `n` Elemente von `x` auf `0`. Das Beispiel ist rein zu Demonstrationszwecken, schneller wäre dieses Ziel z.B. mit der Anweisung

```
x(1:n) = 0;
```

zu erreichen.

Schleifen können ineinander geschachtelt werden:

```
for i = 1:m
   for j = 1:n
      A(i,j) = 1/(i+j-1);
   end
end
```

Zu weiteren Informationen über die Sprachkonstrukte von Matlab kann man sich z.B. über

```
help lang
```

vorarbeiten, vgl. dazu auch den nächsten Abschnitt.

2.5 Online–Hilfe; Informationen zu Matlab

Innerhalb einer laufenden Matlab–Sitzung kann man sich Hilfe mit dem Befehl

```
help
```

verschaffen. Die simple Eingabe des Befehls listet eine Anzahl von Themenbereichen auf, die man wieder als Parameter an `help` weitergeben kann, um Unterthemen angezeigt zu bekommen usw.. Auf der untersten Ebene erhält man dann Hilfetexte zu einzelnen Funktionen oder Stichworten.

Sucht man zum Beispiel nach einer Routine zur Eigenwertberechnung, so kann die Suchreihenfolge

```
help
help matfun
help eig
```

entstehen. (Man darf natürlich auch direkt `help eig` eingeben, wenn man schon weiß, wie die Funktion heißt, oder eine Vermutung hat.)

Für die Suche nach Stichworten gibt es den nützlichen Befehl `lookfor`: Er durchsucht die ersten Zeilen der verschiedenen Hilfetexte nach einem angegebenen Suchbegriff. In unserem Beispiel hätte man z.B. die Funktion `eig` auch über

```
lookfor eigenvalue
```

finden können.

Eine übersichtlichere und weniger an einzelnen Funktionen orientierte Online–Hilfe bietet das Matlab–Helpdesk. Es arbeitet, wenn alles richtig installiert ist, mit dem Standard–Internetbrowser zusammen und wird auf der Matlab–Kommandozeilenebene mit dem Befehl

```
helpdesk
```

gestartet. (Unter Umständen empfiehlt es sich, den Browser vorher gesondert zu starten.)

Weitere Dokumentation zu Matlab wird unter anderem auf den Internetseiten der TU unter der URL

```
http://www.tu-harburg.de/rzt/tuinfo/
       software/numsoft/matlab.html
```

angeboten.

3 Matrixoperationen

Matlab–Matrixoperationen wurden so gestaltet, daß sie der Form nach weitgehend mit den in der Mathematik üblichen Schreibweisen zu realisieren sind.

3.1 Transponation

Das Apostroph bezeichnet das Transponierte einer Matrix. Die Eingabe

```
y = [-1 0 2]'
```

ergibt

```
y =

    -1
     0
     2
```

3.2 Addition und Subtraktion

Addition und Subtraktion sind definiert für Matrizen gleicher Dimension. Im obigen Beispiel ist `A + x` nicht zulässig, aber `A + A'`. Addition und Subtraktion sind auch definiert, wenn ein Operand ein Skalar ist. Dieser Skalar wird zu allen Elementen des anderen Operanden addiert bzw. von ihnen subtrahiert.

Dieses Verhalten ist mit Vorsicht zu genießen, schließlich wird in der Mathematik oft die abkürzende Schreibweise $A - \lambda$ für $A - \lambda I$ verwendet, die der Matlab-Konvention deutlich widerspricht. Aus diesem Grund bietet es sich an, immer die Identität mit hinzuschreiben, sowohl in der Mathematik als auch in Matlab. Die Identität in der Dimension `n` wird durch

```
I = eye(n);
```

erzeugt. Der Name `eye` ist lautmalerisch für die englische Aussprache des Buchstaben I.

3.3 Multiplikation

Matrizen können multipliziert werden, wenn die Spaltenzahl der ersten Matrix gleich der Zeilenzahl der zweiten Matrix ist. Das Produkt ist mit dem in der Mathematik üblichen identisch. Matrix–Vektor–Produkte sind Spezialfälle von Matrix–Matrix–Produkten. Im obigen Beispiel sind sowohl `A*x` als auch `A*A'` zulässig. Es darf aber auch ein Skalar mit einer Matrix oder einem Vektor multipliziert werden wie in `pi*x`.

Beachten Sie bitte den Unterschied zu der *elementweisen* (engl. pointwise) Operation, wie sie im Abschnitt über Array–Operationen vorgestellt wird.

3.4 „Division“

In Matlab existieren zwei Symbole zur Matrixdivision: `\` und `/`. Ist `A` eine reguläre quadratische Matrix, so gilt:

`X = A\B`	löst	`A*X = B`,
`X = B/A`	löst	`X*A = B`.

Division bedeutet hier also die Lösung von linearen Gleichungssystemen. Es werden, wie in der numerischen Mathematik üblich, dabei *keine Inversen* von den auftretenden Matrizen berechnet. Auch die Division durch einen Skalar ist wiederum erlaubt.

4 Operationen auf Arrays

Array–Operationen sind *elementweise* arithmetische Operationen auf Vektoren oder Matrizen. Ein Punkt vor einem Operator zeigt eine Array–Operation an. Der Punkt läßt sich mnemorieren, wenn man sich vergegenwärtigt, daß *pointwise* die englische übersetzung von *elementweise* ist.

4.1 Addition und Subtraktion

Beim Addieren und Subtrahieren entsprechen die Array–Operationen den vorher besprochenen Matrix–Operationen. Das ist natürlich, da die Addition und Subtraktion von Matrizen und Vektoren eben elementweise definiert wird. Die elementweise Addition wird mit `.+`, die elementweise Subtraktion mit `.-` bezeichnet.

4.2 Multiplikation und Division

Die elementweise Multiplikation wird mit dem Symbol `.*` bezeichnet. Bei gleichen Dimensionen von `A` und `B` ist `A.*B` das Array, das durch Multiplikation der einzelnen Elemente von `A` mit den entsprechenden Elementen von `B` entsteht. Ist etwa

```
x = [1 2 3]; y = [4 5 6];
```

dann erzeugt die Eingabe

```
z = x.*y
```

das Ergebnis

```
z =

     4    10    18
```

Das elementweise Produkt zweier Matrizen ist auch unter dem Namen Schur– oder Hadamard–Produkt bekannt.

Die elementweise Division wird mit `A./B` und `A.\B` bezeichnet. Die Eingabe

```
z = x.\y
```

erzeugt das Ergebnis

```
z =

    4.0000    2.5000    2.0000
```

4.3 Potenzierung

Das Symbol `.^` bezeichnet die elementweise Potenz. Mit den obigen Vektoren erzeugt die Eingabe

```
z = x.^y
```

das Ergebnis

```
z =

     1    32   729
```

Der Exponent kann ein Skalar sein:

```
z = x.^2
z =

     1     4     9
```

Ebenso darf die Basis ein Skalar sein.

5 Reelle Funktionen

Funktionen, abgesehen von den bereits in Matlab implementierten, sind meist durch eine Rechenvorschrift gegeben, etwa in Form eines M–Files. Die Datei `quadrat.m` kann z.B. folgendes enthalten:

```
function y = quadrat(x);
y=x.^2;
```

Wenn sie dann im aktuellen Suchpfad von Matlab liegt (also etwa im Arbeitsverzeichnis), kann man sie als „`quadrat`“ aufrufen. Beispiel:

```
x = 2;
y = quadrat(x);
```

Hiernach ist `y == 4`. Wie man sieht, ist das M–File ist zunächst eine abstrakte Zuordnung, die erst per Argumentzuweisung mit Leben gefüllt werden muß.

5.1 Plotbefehle

Wie macht man jetzt einen Plot? Matlab benötigt dazu zwei Arrays, einen Array von x–Werten und einen zugehörigen Array von y–Werten. Standardmäßig sind das (technisch gesehen) Zeilenvektoren.

Beispiel: Sie wollen die Funktion x^2 über $[-1, 2]$ plotten. Sie erzeugen zunächst einen Array mit den x–Werten, z.B. über den Aufruf

```
x = -1:0.2:2;
```

Diese Zeile heißt: Gehe von -1 bis 2 mit der Schrittweite 0.2. Dieser Weg ist mit Vorsicht zu beschreiten. Durch Rundungsfehler

kann es sein, daß weniger oder mehr Werte erzeugt werden, als eigentlich gewünscht. Abhilfe schafft hier die Matlab-Funktion `linspace`. Ein Aufruf der Form `linspace(a,b,n)` erzeugt einen Vektor der Länge `n` von *äquidistanten* Punkten zwischen den Randwerten `a` und `b`. Also verwenden Sie statt obigem Aufruf besser

```
x = linspace(-1,2,16)
```

In beiden Fällen kann man passende y-Werte erzeugen mittels

```
y = x.^2;
```

Der Punkt in `.^` war wichtig, denn jeder x-Wert ist Komponente eines Vektors, und es soll jede Komponente für sich bearbeitet werden. Eine andere Möglichkeit ist

```
y = quadrat(x);
```

mit unserer oben definierten Funktion. Dieses zeigt uns, daß das Argument also auch ein Vektor sein kann (allgemeiner: eine Matrix sein kann). Der gewünschte Plot kann jetzt mittels

```
plot(x,y);
```

erzeugt werden.

Matlab druckt alle (x, y)–Paare und verbindet die aufeinanderfolgenden durch eine gerade Linie. Bei zu grober x–Einteilung ist noch deutlich ein Polygonzug erkennbar. Probieren Sie z.B.

```
x = linspace(-1,2,7);
plot(x,x.^2,'g');
```

Hier bewirkt das `'g'`, daß die Kurve in grün (green) dargestellt wird. Man kann x–Vektoren natürlich auch anders als durch `linspace` oder die Schleifenanweisung erzeugen; das sind nur einige der am häufigsten auftretenden und gleichzeitig bequemsten Möglichkeiten.

5.2 Funktions-Funktionen

Matlab hat einige eingebaute Funktionen zur Funktionenbearbeitung, siehe

```
help funfun
```

Als Beispiel betrachten wir den Befehl `fplot`. Er erzeugt einen Funktionsplot und wählt dazu die x–Punkte selbst. Hierfür hat man nur die Funktionsvorschrift zu liefern, also etwa den Namen des M–Files oder einen Ausdruck. Unsere Aufgabe von oben wäre mit

```
fplot('quadrat(x)',[-1,2]);
```

oder einfach

```
fplot('x^2',[-1,2]);
```

gelöst. Beachten Sie die Striche um den „symbolischen Ausdruck“ `'x^2'`. Die Verwendung von `fplot` sieht zwar sehr praktisch aus, macht aber auch nicht immer, was man will. Als Beispiel hierzu siehe:

```
fplot('tan(x)', [0 2*pi])
```

Hier muß man zunächst die Fensterachsen anpassen, um das Ergebnis zu beurteilen:

```
axis([0 7 -10 10]);
```

Das Ergebnis ist ziemlich indiskutabel. Mit

```
x=0:pi/100:2*pi;
y=tan(x);
plot(x,y);
axis([0 7 -10 10]);
```

bekommt man ein wesentlich vertrauenerweckenderes Bild. (Das Problem ist wahrscheinlich die Beurteilung des Ergebnisses auf Grundlage eines relativen Fehlers, denn der Funktionswert wird ja sehr groß.)

Andere für Sie interessante Funktions-Funktionen sind die folgenden Funktions-Funktionen:

`fmin`: Berechnet das Minimum einer Funktion in einem vorgegebenen Intervall. Als Beispiel: Berechne das Minimum von $x^2 - 3x$ auf dem Intervall $[-1, 2]$ (Ergebnis 1.5):

```
fmin('x^2-3*x',-1,2)
```

`fzero`: Berechnung der Nullstelle einer Funktion bei vorgegebenem Startwert für die Iteration. Als Beispiel folgt der Aufruf für die Berechnung der Nullstelle von $x^2 - 2$ mit dem Startwert 1 (Ergebnis $\sqrt{2}$).

```
fzero('x^2-2',1)
```

Außerdem gibt es Integrationsverfahren, Differentialgleichungslöser etc..

5.3 Themenkreis Plotten

Für weitere Einzelheiten zum Plotten siehe

```
help graph2d
```

Einige Andeutungen:

```
semilogy
```

funktioniert wie „`plot`“, nimmt aber eine semilogarithmische y–Skala.

```
axis
```

nimmt Einfluß auf das Anzeigefenster eines Plots, s.o.. Explizite Angabe:

```
axis([links rechts unten oben])
```

wie oben im Beispiel. Alternativen:

```
axis('auto')
```

wählt eine Standardeinstellung.

```
axis('equal')
```

wählt übereinstimmende x– und y–Skalierungen, was für die Darstellung geometrischer Objekte wie Kreise oft wünschenswert ist.

```
zoom on
```

kann zum manuellen Vergrößern des Bildausschnitts per Maus benutzt werden.

```
hold on
```

bietet sich an, falls mehrere Plots in ein Bild sollen. Immer vorzuziehen ist allerdings der Aufruf von `plot` mit mehreren Argumenten, gemäß

```
plot(x1,y1,c1,...
     x2,y2,c2,...
     x3,y3,c3);
```

Die Ellipsen (...) teilen Matlab nur mit, daß die Eingabezeile noch nicht hier endet, sondern in der nächsten Zeile weitergeht. Diese erweisen sich auch in anderen Fällen als nützlich, wo die Eingabezeilen zu lang und zu unübersichtlich werden.

```
figure
```

öffnet ein neues Bild. Mit

```
figure(Nummer)
```

bearbeitet man (wieder) das Bild mit der angegebenen Nummer.

Es gibt auch noch die Möglichkeit, dreidimensionale Plots zu erstellen, dazu lesen Sie am besten die Hilfeseiten zu den Befehlen `mesh`, `surf` oder `contour`.

6 Erweiterungen

Es gibt zusätzlich zu der eingebauten Funktionalität viele Erweiterungen in Form sogenannter Toolboxen. Toolboxen vereinen mehrere Programme unter einem Gesichtspunkt, so gibt es eine Optimierungstoolbox, eine Toolbox zur Lösung von gewöhnlichen Differentialgleichungen etcpp. Viele dieser Toolboxen sind bereits im Lieferumfang von Matlab enthalten, einige freie Toolboxen findet man im Internet.

Aufgabensammlung

Es folgen nun ein paar Aufgaben, die dazu dienen sollen, Ihnen den Umgang mit Matlab und seinen Sprachkonstrukten zu erleichtern. Diese Aufgaben sind allerdings keine Übungen im eigentlichen Sinne, d.h. Sie lernen aus Ihnen nichts was später in einer Prüfung (mündlich oder Klausur) abgefragt wird. Dieses macht es uns aber möglich, (hoffentlich) interessante Aufgaben zu stellen, die Ihnen den Spaß an Mathematik und Matlab vermitteln. Unnötig zu sagen, daß die Aufgaben deutlich *schwieriger* sind als alle später zu bearbeitenden `;-)`

Aufgabe 1:

Diese Aufgabe soll Ihnen Matlabs Sprachkonstrukte etwas näher bringen. Außerdem lernen Sie einige schwer vermittelbare (und schwer beweisbare) Inhalte über Zufallsmatrizen. Sie sollten das Erlernte aber ruhig im Hinterkopf behalten, falls Sie einmal in die Lage kommen, neue Algorithmen mit „Zufalls“–matrizen testen zu wollen.

- Generieren Sie eine Matrix $A \in \mathbb{R}^{n\times n}$ für ein selbstbestimmtes $n \in \mathbb{N}$, so daß die Einträge zufällig (Gleichverteilung) zwischen $[0, 1]$ verteilt sind (`help rand`). Wählen

Sie n nicht zu klein, wählen Sie z.B. $n = 100$.

- Berechnen Sie alle Eigenwerte von A (`help eig`). Visualisieren Sie diese in einer geeigneten Form (`help plot`).
- „Philosophieren“ Sie über den einen reellen, gut separierten und betragsmaximalen Eigenwert (der ungefähr bei $n/2$ liegen wird).
- Wenn Sie die anderen Eigenwerte ansehen, wird Ihnen auffallen, daß diese sich ungefähr gleichmäßig in einem Kreis mit Radius $\sqrt{n}/2$ um die Null verteilen.
- Generieren Sie eine Matrix $B \in \mathbb{R}^{n\times n}$ für ein selbstbestimmtes $n \in \mathbb{N}$, so daß die Einträge zufällig (Normalverteilung) mit Standardabweichung 3/10 verteilt sind (`help randn`).
- Berechnen Sie alle Eigenwerte von B (`help eig`). Visualisieren Sie diese in einer geeigneten Form (`help plot`).
- Vergleichen Sie die beiden Plots. Was fällt Ihnen auf?

Lösungskommentar zu Aufgabe 1:

Diese Aufgabe dient dazu, Ihnen zu zeigen, daß die Eigenwerte von sogenannten Zufallsmatrizen alles andere als eben „zufällig“ sind. Man kann (unter gewissen, weiter einschränkenden Bedingungen) zeigen:

Girko's Circular Law: Die Eigenwerte einer reellen $n \times n$ Matrix, deren Einträge zufällig in $[-1/\sqrt{n}, 1/\sqrt{n}]$ verteilt sind, überdecken im Limes $n \to \infty$ den Einheitskreis gleichmäßig.

Dieses Überdecken im Limes tritt augenscheinlich schon früher in approximativer Form auf. Siehe dazu auch die Seite bei Wolfram:

`http://mathworld.wolfram.com/GirkosCircularLaw.html`.

Die Matrix B ist von der oben genannten Form, es geht nur eine Skalierung ein. Der Zusammenhang zwischen den Matrizen A und B besteht darin, daß sich A als gestörte Rang-Eins-Matrix auffassen läßt, wobei die Störung beinahe vom oben erwähnten Typ ist:

$$A = \frac{1}{2}(E + \Delta), \qquad E = \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix} \qquad \Delta_{ij} \in [-1, 1]. \tag{1}$$

Die oben angegebene Matrix E ist das äußere Produkt des Vektors e aus lauter Einsen, $E = ee^T$. Damit hat E die Eigenwerte n (einfach) und 0 ($(n-1)$-fach). Damit wird A einen Eigenwert nahe bei $n/2$ haben. Die Matrix $1/2 \cdot \Delta$ ist von ähnlichem Typ wie die Matrix B, also sollten auch die Eigenwertverteilungen ähnlich aussehen. Der angegebene Faktor ist allerdings nicht so leicht zu erklären.

Aufgabe 2:

- Generieren Sie eine reelle symmetrische Tridiagonalmatrix. Sollte Ihnen keine geeignete einfallen, wählen Sie eine Zufallsmatrix (`help rand` und/oder `help hess`).
- Berechnen Sie die Eigenwerte und Eigenvektoren (`help eig`).
- Sortieren Sie die Eigenwerte und Eigenvektoren nach der Größe der Eigenwerte (`help sort`).

- Plotten Sie die (dekadischen) Logarithmen der Absolutbeträge der so sortierten Eigenvektormatrix (`help mesh`, `help abs`, `help log10`).

Lösungskommentar zu Aufgabe 2:

Sie sollten in dieser Aufgabe mit Matlab vertrauter werden und einige für spätere Aufgaben nötige Befehle kennen lernen. Außerdem wäre es schön, wenn Sie feststellen, daß die Eigenvektoren der gut separierten äußeren Eigenwerte sehr stark abklingen (i.e., sehr kleine Einträge für große Indizes haben). Das liegt an den Relationen zwischen Tridiagonalmatrizen, dem Lanczos–Verfahren, orthogonalen Polynomen und Gauß-Quadratur (Auf welche wir leider nicht mehr genauer eingehen werden `;-)`

Das Generieren einer Tridiagonalmatrix, das Sortieren der Eigenvektoren nach der Größe der Eigenwerte mit anschließendem Plot kann dabei (zum Beispiel) wie in dem folgenden kleinen Programm-Fragment skizziert durchgeführt werden:

```
%% generate tridiagonal matrix
B = randn(n);
A = (B+B')/2;
T = hess(A);
%% compute and sort eigenvalues/eigenvectors
[V,Lambda] = eig(T);
[lambda,index] = sort(diag(Lambda));
V = V(:,index);
%% plot eigenvector entries in logarithmic scale
mesh(log10(abs(V)));
```

Dieses Beispiel soll auch wieder die Verwendung einiger neuer Befehle zeigen, es ist nicht notwendig die „beste“ Lösung der Aufgabe. Wenn Sie einen Befehl nicht kennen/verstehen, sollten Sie die Hilfe (`help`) verwenden und dann mit dem Befehl experimentieren.

Aufgabe 3:

Wählen Sie ein beliebiges (aber nicht zu kleines und nicht zu großes) $n \in \mathbb{N}$. Werten Sie die Funktion

$$f(x) \;:=\; 2 - 2\cos(x)$$

an $n+2$ äquidistanten Stellen in $[0, \pi]$ aus, wobei 0 und π vorkommmen. Speichern Sie das Ergebnis in der Variablen `xa` ab. Werten Sie dieselbe Funktion jetzt an den Stellen

$$x_k = \pi \cdot \frac{2k+1}{2n}, \quad k \in \{0, 1, \ldots, n-1\}$$

aus. Speichern Sie das Ergebnis in der Variablen `xb`.

Berechnen und plotten Sie

- die Eigenwerte der $n \times n$ Matrix

$$T := \text{tridiag}(-1, 2, -1) := \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}$$

  gegen die Elemente 2–$(n+1)$ von `xa`. Tipp: `help ones`, `help diag`.

- die Realteile $n+2$ äquidistanter Punkte auf der um zwei skalierten und von plus zwei abgezogenen oberen Hälfte des Einheitskreises gegen `xa` (`help exp`, `help real`, `help linspace`). Der erste Wert sollte Null sein, der letzte Wert sollte Vier sein.

Sollten Sie dann noch nicht genug haben, berechnen und plotten Sie

- die Nullstellen des $n$ten CHEBYSHEV Polynomes (erster Art) für das Intervall $[a, b] := [0, 4]$ gegen `xb` (`help conv`, `help root`, `help fzero`).

Wenn nötig, sortieren Sie die erhaltenen Werte einmal der Größe nach aufsteigend. Vergleichen Sie die erhaltenen Plots. Haben Sie ein Gefühl dafür, warum das Ergebnis gerade so ausfällt?

Falls Sie nicht vertraut sind mit CHEBYSHEV Polynomen, folgt hier eine kurze Zusammenfassung: CHEBYSHEV Polynome auf dem Intervall $[a, b] = [-1, 1]$ werden häufig definiert durch

$$C_n(x) \;:=\; \cos(n \arccos(x)), \quad x \in [-1, 1].$$

Daß diese Definition auch wirklich Polynome erzeugt, folgt aus den Additionstheoremen für trigonometrische Funktionen, es gilt eine sogenannte Drei Term Rekursion:

$$C_0(x) = 1, \qquad C_1(x) = x, \qquad C_{n+1}(x) = 2x \cdot C_n(x) - C_{n-1}(x).$$

Alternativ kann man die CHEBYSHEV Polynome auch in der geschlossenen Form

$$C_n(x) \;=\; \frac{1}{2}\left[\left(x + \sqrt{x^2-1}\right)^n + \left(x - \sqrt{x^2-1}\right)^n\right]$$

angeben. CHEBYSHEV Polynome $C_n^{[a,b]}$ für andere Intervalle $[a, b]$ findet man durch simple lineare Transformation,

$$C_n^{[a,b]}(x) := C_n\left(\frac{2}{b-a}x + \frac{a+b}{a-b}\right).$$

Sie sollten eine der beiden expliziten Darstellungen (oder die Drei Term Rekursion) ausnutzen und dann z.B. mit `fzero` arbeiten. Ein Tipp dazu: Alle Nullstellen sind im Intervall $[-1, 1]$ zu finden. Weiterhin handelt es sich um Polynome, die *ausschließlich* reelle Nullstellen haben.

Die Nullstellen der CHEBYSHEV Polynome sind in gewisser Hinsicht als Knoten bei der Polynominterpolation optimal, denn sie minimieren $\|\omega\|_\infty$, den maximalen Ausschlag des Knotenpolynomes $\omega(x)$.

Lösungskommentar zu Aufgabe 3:

Die beiden Vektoren `xa` und `xb` sollten sich leicht berechnen lassen, anderenfalls folgt hier Quellcode der dieses macht:

```
xa = 2-2*cos(linspace(0,pi,n+2));
xb = 2-2*cos(pi*(2*(0:n-1)+1)/(2*n));
```

Die Eigenwerte der Matrix sollten sich nach dem Vorherigen auch leicht berechnen und plotten lassen. Sicherheitshalber sollte man sie aber noch sortieren, da nicht a priori klar ist, ob `eig` sie sortiert zurück gibt:

```
%% construct T
T = 2*diag(ones(n,1))-diag(ones(n-1,1),1)-diag(ones(n-1,1),-1);
%% compute and sort eigenvalues
xeigT = sort(eig(T));
```

Die Werte auf dem verschobenen, skalierten, oberen Halbkreis lassen sich leicht mittels

```
%% construct upper half-circle
z = exp(i*linspace(0,pi,n+2));
%% scale + shift
z = 2-2*z;
%% compute real part
xcirc = real(z);
```

berechenen. Diese brauchen nicht mehr sortiert werden.

Die Berechnung der Nullstellen erfolgt am elegantesten unter Verwendung der Drei Term Rekursion, `conv` und `roots`:

```
%% initialise
coeffCnmo = [0 1];
coeffCn   = [1 0];
linfacx   = [1 0];

for k = 1:n-1
   %% new coefficient vector (3-term recurrence)
   coeffCnpo = 2*conv(linfacx,coeffCn)-[0,coeffCnmo];
   %% renumber
   coeffCnmo = [0,coeffCn];
   coeffCn   =  coeffCnpo;
end

%% due to ill-conditioning roots may be complex
xroots = sort(real(2-2*roots(coeffCn)));
```

Leider ist diese Form der Berechnung nicht sehr stabil, wie man bei größerem n deutlich sieht. Der Grund ist, daß Polynomnullstellen meist nicht sehr gut durch die Koeffizienten bestimmt sind.

Das Ergebnis sollte (Wunder über Wunder) immer gleich sein. Wer's nicht glaubt, kann es mit dem folgenden Programm-Fragment für einen abschließenden Plot wenigstens numerisch verifizieren:

```
plot(xa(2:n+1),xeigT,'go',...
     xa,xcirc,'r+',...
     xb,xroots,'bs');
```

Für $n = 100$ weicht die blaue „Kurve“ deutlich von der Geraden $g(x) = x$ ab. Fehler treten aber schon bei $n = 30$ sichtbar zu Tage.