Eine (extrem) kurze Einführung in MATLAB

von Prof. Dr. Dr. Daniel J. Strauss*

Vorwort

Das Seminar *Eine kurze Einführung in MATLAB* soll einen ersten Einblick in das Scientific Computing System MATLAB geben. Vorrangiges Ziel ist es, den Hörer dieser Vorlesung in die Lage zu versetzen, sich Sachverhalte der Numerischen Mathematik selbständig und auf experimenteller Basis auf dem Computer zu veranschaulichen. Um dieses Ziel zu erreichen, gliedert sich das Seminar in theoretische Erläuterungen zu MATLAB und praktische Übungen am Computer.

Desweiteren soll das Seminar einen Überblick über den Leistungsumfang von MATLAB geben, was im zeitlichen Rahmen dieses Seminars aber nur sehr oberflächlich geschehen kann. Dennoch soll der Teilnehmer abschätzen können, welche Lösungskonzepte MATLAB für spezifische Probleme bereitstellt. Der Teilnehmer soll ebenfalls erlernen, wie er sich selbständig in weitere – im Seminar nicht behandelte – Funktionen des Systems ohne größeren Aufwand einarbeiten kann. Kenntnisse in höheren Programmiersprachen werden nicht benötigt.

*Ursprünglich am 16. November 2000 als ergänzendes Seminar zur Vorlesung Numerik I an der Universität Mannheim erstellt.

1 Allgemeines

MATLAB ist ein sehr leistungsfähiges Softwaresystem der Firma MathWorks für Berechnungen in Wissenschaft und Technik und hat bereits internationale Standards im Bereich des Scientific Computing gesetzt. Die Bezeichnung MATLAB steht stellvertretend für *MATrix LABoratory*, was die Grundgedanken dieses Systems wiederspiegelt: Die grundlegenden Elemente sind Matrizen, d.h. MATLAB ist ein Matrix-orientiertes System. Ursprünglich wurde mit der Entwicklung von MATLAB ein einheitlicher Zugang zu den Softwaresystemen *EISPACK* und *LINPACK* verfolgt. Diese Systeme ermöglichten einen sicheren und portablen Zugang zu Problemen der linearen Algrabra und wurden in den 70'er Jahren entwickelt.

MATLAB wurde im Lauf der Jahre ständig weiterentwickelt und stellt heute dem Anwender für viele Probleme der Numerik ausgefeilte Lösungskonzepte unmittelbar bereit. Es vereint unter einer einheitlichen Benutzeroberfläche die 3 Hauptkomponenten:

BERECHNUNG: MATLAB stellt eine Sammlung von ausgetesteten Algorithmen bereit. Auf Standardalgorithmen der Numerischen Mathematik, z.B. Eigenwertberechnungen oder die schnelle Fourier–Transformation, kann so bequem per Funktionsaufruf zugegriffen werden. Dem Anwender bleibt somit die aufwendige Programmierung von Standardroutinen in einer universellen Programmiersprache wie z.B. C oder FORTRAN erspart. Die nicht zu unterschätzende Fehlerquelle einer ad hoc–Programmierung kann somit ebenfalls umgangen werden.

VISUALISIERUNG: MATLAB verfügt über sehr ausgereifte Techniken zur Visualisierung von Daten (2D, 3D, Kontur-Plots etc.). Die Hinzunahme von sekundären Visualisierunssystemen wie z.B. ORIGIN entfällt somit.

PROGRAMMIERUNG: MATLAB beinhaltet eine eigene hohe Programmiersprache. Der Anwender ist somit in der Lage die Funktionalität von MATLAB ständig zu erweitern, wobei natürlich auf die bereitgestellten Algorithmen zurückgegriffen werden kann. MATLAB-Programme werden als `m-files` bezeichnet.

Neben diesen 3 Hauptkomponenten tragen die folgenden Punkte zu dem heutigen Erfolg von MATLAB bei:

Syntax: MATLAB verfügt über eine benutzerfreundliche und intuitive Syntax die auf der mathematischen Notation basiert.

Toolboxen: MATLAB kann durch sogenannte Toolboxen erweitert werden. Dies sind im wesentlichen Sammlungen von `m-files` die für bestimmte spezialisierte Anwendungen ausgetestete Lösungskonzepte bereitstellen, z.B. Bildverarbeitung (*Image Processing Toolbox* oder auch die *Wavelet Toolbox*), Kommunikationstechnik (*Communications Toolbox*), Neuronale Netze (*Neural Network Toolbox*), Finanzmathematik (*Financial Toolbox*), Statistik (*Statistics Toolbox*) oder auch für die Signalverarbeitung (*Signal Processing Toolbox*) – um nur einige zu nennen. Die *Signal Processing Toolbox* ermöglicht z.B. das Design von digitalen Filtern einzig und allein durch die Angabe einiger gewünschter Parameter wie etwa die Grenzfrequenzen. Die oben genannten Toolboxen werden kommerziell von MathWorks Inc. vertrieben. Dem Anwender wird natürlich weiterhin die Möglichkeit geboten, ein Konglomerat von selbstgeschriebenen `m-files` zu einer Toolbox zu vereinen. Weiterhin bieten auch viele Universitäten eigens entwickelte MATLAB Toolboxen – für wissenschaftliche Zwecke – kostenfrei via Internet an.

SIMULINK: SIMULINK ist eine ergänzende Symbol-orientierte, interaktive Begleitsoftware von MATLAB zur Simulation linearer und nichtlinearer Systeme. Sie ermöglicht dem Ingenieur das Erstellen von komplexen

Systemen – bequem auf visueller Basis – über Blockschaltbilder, wodurch oftmals ein erheblicher Programmieraufwand vermieden werden kann. In SIMULINK wird die Struktur eines Systems graphisch mit Funktionsblöcken nachgebildet, deren Parameter frei gewählt werden können, z.B. die Koeffizienten eines digitalen Filters. So kann ein System z.B. mit einem Signalgenerator erregt werden und der dynamische Verlauf an bestimmten Punkt des Systems betrachtet werden. Grob gesagt, die Programmierung des Quelltextes wird hier durch eine mausgesteuerte und graphische Oberfläche ersetzt wobei natürlich weiterhin – aber versteckt – MATLAB Funktionen aufgerufen werden.

Der Erfolg von SIMULINK hat dazu geführt, daß sog. *Blocksets* entwickelt wurden die ebenfalls zusätzliche Symbolbibliotheken für bestimme Anwendungen zur Verfügung stellen, z.B. mit Kopplung an die entsprechende Toolbox. Weiterhin bietet der sog. *Real Time Workshop* die Möglichkeit, die erstellten Blockdiagramme direkt in C–Code zu übersetzen, der auf einer Vielzahl von real time Systemen abgearbeitet werden kann oder z.B. zur weiteren Portierung auf einen DSP (*Digital Signal Processor*) dienen kann.

Symbolisches Rechnen: Die *(extended) Symbolic Math Toolbox* versetzt MATLAB in die Lage symbolisch zu rechnen, d.h. mit Formeln analytisch zu rechnen, ähnlich wie man es von den bekannten Computer–Algebrasystemen MACSYMA und MAPLE kennt. Diese spezielle Toolbox basiert auf MAPLE und stellt die Fähigkeiten diese Systems uneingeschränkt zur Verfügung.

Graphische Benutzeroberflächen: MATLAB bietet durch das sog. GUIDE (*Graphical User Interface Development Environment*) die Möglichkeit zur schnellen Erstellung von graphischen Benutzeroberflächen. Binnen Minuten können komplexe graphische Applikationen erzeugt werden. Viele Toolboxen enthalten ebenfalls Anwendungen die über eine graphische Be-

nutzeroberfläche gesteuert werden können.

Schnittstelle zu anderen Programmiersprachen: Das *Application Program Interface* ermöglicht das dynamische Einbinden (dynamically linked) von vorhandenen C oder FORTRAN Codes in `m-files` über sog. `mex-files` (MATLAB *Executable files*). Neben der Möglichkeit vorhandene Routinen in C oder FORTRAN zu nutzen, können Algorithmen die z.B. sehr viele Schleifen benötigen in C oder FORTRAN implementiert werden um Vorteile bei der Laufzeit des Programms zu erreichen. Letzteres ist allerdings nur für bestimmte Spezialanwendungen nötig, die extrem zeitkritisch sind.

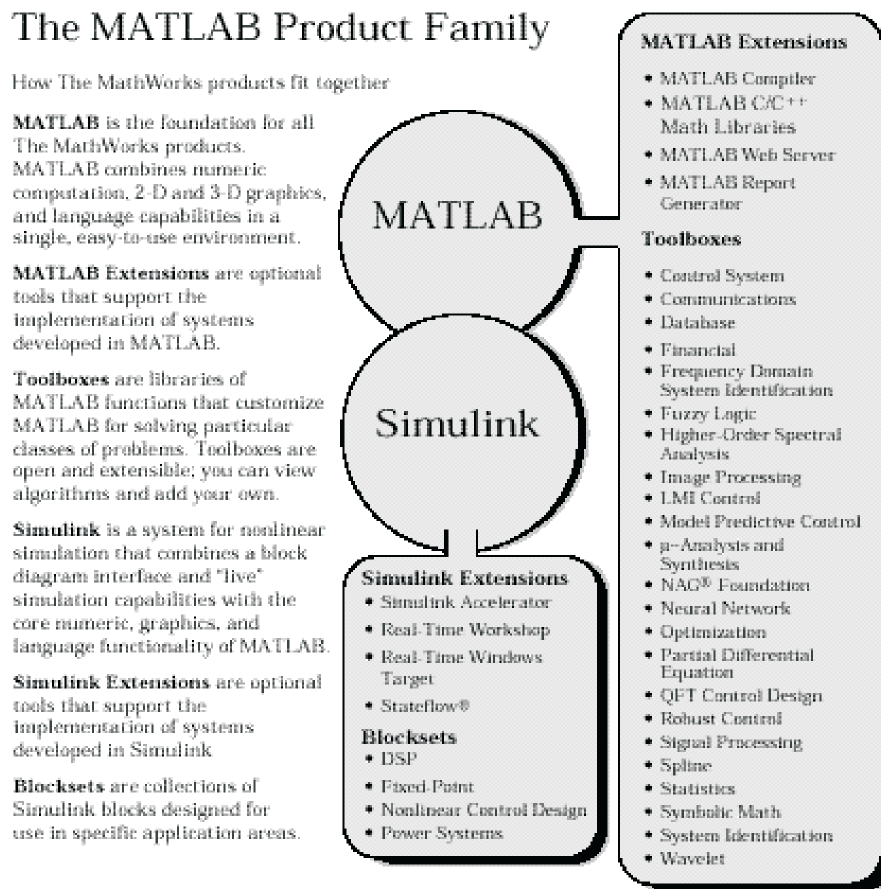

Abbildung 1: MATLAB und Zusatzsoftware

Weiterhin bietet MathWorks den *MATLAB Compiler*, *C/C++ Math Library* und die *C/C++ Graphics Library* an. Diese Ergänzungen können separat oder als Teil der *MATLAB C/C++ Compiler Suite* erworben werden und ermöglichen die Übersetzung von `m-files` in C/C++ Routinen oder das Einbinden von MATLAB Funktionen in C/C++ (auch von Graphiken). Desweiteren können hiermit *stand–alone* Applikationen realisiert werden, d.h. Software die unabhängig von MATLAB ist. Für zeitkritische Anwendungen, können auch vorhandene `m-files` in C/C++ übersetzt werden und dann wieder dynamisch über `mex-files` in `m-files` eingebunden werden, was aber – wie zuvor schon bemerkt – nur für Spezialanwendungen nötig ist. Mit diesen Ergänzungen ist es ebenfalls möglich C Funktionen für SIMULINK zu nutzen, d.h. an bestimmte Funktionsblöcke zu koppeln.

Bevor im folgenden auf die MATLAB inhärenten Hilfsfunktionen eingegangen wird, sollen kurz eine weitere, wichtige Informationsquelle angegeben werden. Dies ist die URL der Firma MathWorks:

`http://www.mathworks.com`

Hier sind ständig Informationen zu neusten Produktentwicklungen zu MATLAB verfügbar, sowie weitere kostenlose Serviceangebote z.B. diverse downloads, Informationsblätter wie *News and Notes* oder den *MATLAB-Digest* via email. Alle Handbücher zu MATLAB, SIMULINK, Toolboxen etc. werden auf der URL:

`http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml`

kostenlos in PDF zum download angeboten. In den Handbüchern zu den Toolboxen findet sich auch oftmals eine gute und prägnante Darstellung der zugrundeliegenden Mathematik und viele Anwendungsbeispiele.

In MATLAB selbst gibt es verschiedene Möglichkeiten sich Informationen zu Befehlen, Syntax etc. zu verschaffen:

- Das *helpwin* System, das im interaktiven Modus (s. folgendes Kapitel) einfach über die Eingabe `helpwin` gestartet wird.
- Das *helpdesk* System, das auf HTML basiert. Dieses System kann ebenfalls durch die Eingabe `helpdesk` gestartet werden.
- Der `help` Befehl. Mit `help` *Name der Funktion* verschafft man sich Informationen über eine Funktion.
- Der `lookfor` Befehl. Mit `lookfor` *Stichwort* können Funktionen anhand eines Stichwortes gesucht werden.

Beispiele zu diesen Hilfsfunktionen gibt das folgende Kapitel. Eine ausführliche Zusammenstellung von MATLAB Funktionen ist im Anhang zu finden.

2 Interaktives Arbeiten in MATLAB

2.1 Grundsätzliches

Nach dem Start der Software erscheint ein Kommandofenster das die Verbindung zum MATLAB–Interpreter herstellt. Hier wartet das Prompt

```
>>
```

auf Ihre Eingabe.

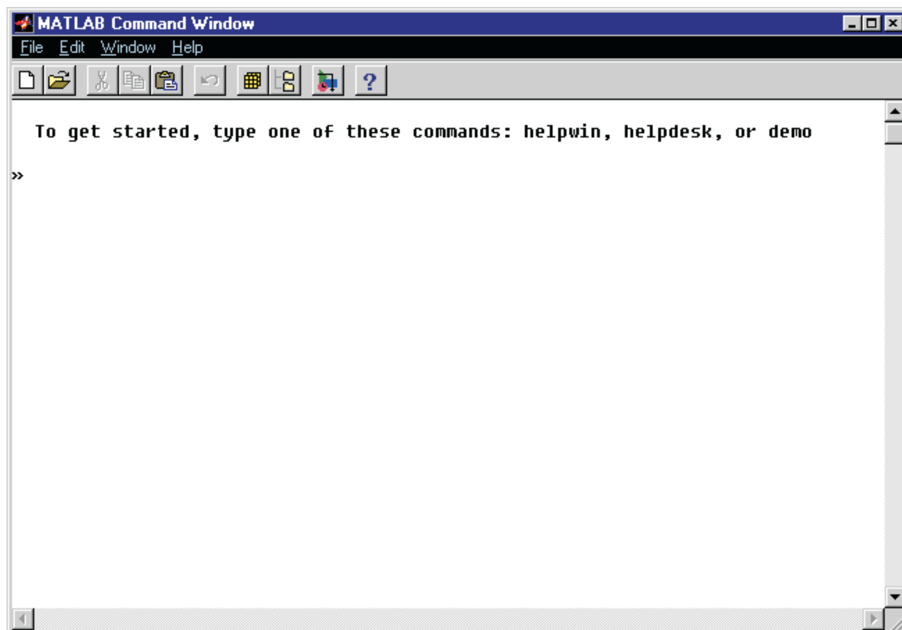

Abbildung 2: Das Kommandofenster unter Windows

Nun kann z.B. eine arithmetische Rechenoperation zwischen Skalaren durchgeführt werden. Hier gilt die übliche Notation:

`+` Addition; `-` Subtraktion; `*` Multiplikation; `/` Division.

Weitere Operatoren u.a. auch Vergleichsoperatoren, d.h. kleiner, größer und gleich, sowie die Syntax sämtlicher Funktionen, z.B. trigonometrische Funktionen, sind im Anhang zu finden. Gibt man z.B.

```
>> 4.5+5.5
```

ein und betätigt die *Enter*– oder *Return*–Taste, so erhält man als Ausgabe:

```
ans=24.2
```

MATLAB hat die Berechnung ausgeführt. Weiterhin wurde die Variable `ans` (von *answer*) erzeugt, die das Ergebnis der Berechnung speichert. Gibt man nun

```
>> erg=25

erg =

    25
```

ein, so ist im *Workspace* von MATLAB eine Variable `erg` mit dem Wert 25 angelegt worden.

Unterdrückung der Ausgabe

Wenn die Eingabe durch ein Semikolon abgeschlossen wird, kann die Ausgabe unterdrückt werden. Dies ist insbesondere bei der Zuweisung von sehr großen Variablen wie z.B. großen Matrizen empfehlenswert. Sollten Sie es einmal vergessen: Mit `ctrl` und `c` können in MATLAB laufende Operationen abgebrochen werden.

Eingabe langer Zeilen

Wenn besonders lange Ausdrücke eingegeben werden müssen, reicht oftmals eine Zeile nicht aus. Wenn MATLAB mitgeteilt werden soll, daß ein Ausdruck in der folgenden Zeile fortgesetzt wird, so kann dies durch drei Punkte ... geschehen.

Die Variablen des Workspace

Die im *Workspace* gespeicherten Variablen können mit `who` oder für eine ausführlichere Ausgabe mit `whos` gesichtet werden: Das Kommando `whos` führt nach der bisherigen Variablenzuweisung zu der folgenden Ausgabe:

```
 Name      Size         Bytes  Class


  ans       1x1                8  double array
  erg       1x1                8  double array


Grand total is 2 elements using 16 bytes
```

Die Bezeichnung `size` gibt die Größe des Feldes (*array*) der Variablen an. Ein 2D–Feld ist eine Matrix, also der Grundbaustein von MATLAB. Vektoren und Skalare werden als Spezialfälle von Matrizen aufgefaßt. Eine Matrix der Größe $n \times 1$ ist ein Zeilenvektor mit n Elementen. Eine Matrix der Größe $1 \times n$ ist ein Spaltenvektor mit n Elementen. Im obigen Fall liegt eine Matrix der Größe 1×1, also ein skalarer Wert vor.

`Bytes` gibt den benötigten Speicherplatz und `class` den Datentyp an. Unter anderem sind dies

```
double, char, sparse, uint8, cell, struct.
```

In diesem Seminar werden wir uns fast ausschließlich auf Felder konzentrieren, die vom Datentyp `double` sind und reelle oder komplexe Elemente beinhalten. Zeichenketten, sog. strings, als Variablen sind ebenfalls möglich. Diese müssen mit Hochkommta zugewiesen werden, z.B. `var='Zeichenkette'` und sind vom Datentyp char.

Komplexe Zahlen

Arithmetische Operationen mit komplexen Zahlen können in MATLAB ebenso ausgeführt werden wie mit reellen Zahlen, d.h. es werden die gleichen Operatoren verwendet. Zu beachten ist hier, daß für die imaginäre Einheit sowohl `i` als auch `j` vorgesehen sind, d.h. $i = j = \sqrt{-1}$. Weiterhin gilt für eine komplexe Zahl z:

`abs`	Absolutwert (Betrag, Modul) $\|z\|$
`real`	Realteil $Re(z)$
`imag`	Imaginärteil $Im(z)$
`angle`	Winkel (Argument) $arg(z)$
`conj`	Konjugiert komplex $\bar{z}$
`abs(z)*exp(i*angle(z))`	$\|z\| \cdot e^{i \cdot arg(z)}$

Beispiel:

```
>> z=4+5*i

z =

   4.0000+ 5.0000i

>> abs(z)

ans =

    6.4031

>> angle(z)

ans =

    0.8961

>> abs(z)*exp(i*angle(z))

ans =

   4.0000+ 5.0000i
```

In MATLAB sind neben `ans`, `i` und `j` die folgenden Variablen vordefiniert:

`eps`	Maschinengenauigkeit
`pi`	Kreiszahl π
`inf`	Unendlich ∞
`NaN`	Not–a–Number
`clock`	Aktuelle Uhrzeit und Datum
`date`	Aktuelles Datum
`flops`	Zählt die Gleitpunktopertionen
`nargin`	Anzahl der Eingabeargumente einer Funktion
`nargout`	Anzahl der Ausgabeargumente einer Funktion
`inputname`	Eingabeargumentname
`computer`	Identifiziert Computer

An dieser Stelle soll bemerkt werden, daß die vordefinierten Variablen überschrieben werden können, d.h. setzt man z.B. `clock=5`, so ist in dem *workspace* `clock` mit 5 belegt. Die weiteren Befehle / Funktionen sind ebenfalls wichtig für die Arbeit im interaktiven Modus:

`help`	Hilfe
`lookfor`	Informationssuche anhand eines Stichwortes
`clear`	Löscht die Variablen des *workspace*
`clear all`	Löscht alle Variablen (auch globale)
`load`	Laden von Daten
`save`	Speichern von Daten
`print`	Drucken und Speichern von Graphiken
`cd` oder `pwd`	Wechselt ein Verzeichnis
`delete`	Löscht ein file
`dir` oder `ls`	Zeigt den Inhalt eins Verzeichnisses
`exit`	MATLAB beenden

Wie zuvor schon erwähnt, kann man sich mit `help` Informationen zu Funktionen und deren Syntax verschaffen, z.B.

```
>> help dir

 DIR List directory.
 DIR directory_name lists the files in a directory. Pathnames
 and wildcards may be used.  For example, DIR *.m lists all
 the M-files in the current directory.

 D = DIR('directory_name') returns the results in an M-by-1
 structure with the fields:
     name  -- filename
     date  -- modification date
     bytes -- number of bytes allocated to the file
     isdir -- 1 if name is a directory and 0 if not

 See also WHAT, CD, TYPE, DELETE.
```

Also wenn Sie nicht genau wissen was ein Funktion genau ausführt oder sich über die Syntax nicht im klaren sind, hilft Ihnen `help` weiter. Mit `lookfor` können Sie sich Informationen zu einem Stichwort ansehen, z.B.

```
>> lookfor directory
ADDPATH Add directory to search path.
CD      Change current working directory.
DIR List directory.
LS List directory.
PWD Show (print) current working directory.
RMPATH Remove directory from search path.
```

```
WHAT List MATLAB-specific files in directory.
ISDIR  True if argument is a directory.
 README file for the uitools directory
FILESEP Directory separator for this platform.
MATLABROOT Root directory of MATLAB installation.
TEMPDIR Get temporary directory.
DPSSDIR  Discrete prolate spheroidal sequence database
directory.
MIDPREFS Choose directory for idprefs.mat,
the ident start-up info file.
```

Mit den Befehlen `dir` etc. können Sie sich in MATLAB ähnlich bewegen und files verwalten wie z.B. in DOS oder UNIX.

2.2 Operationen mit Matrizen

Wie zuvor schon erwähnt, der Grundbaustein von MATLAB ist eine rechteckige Matrix und alle anderen Variablen sind Spezialfälle davon. Daher gelten die folgenden Aussagen ebenfalls für Skalare und Vektoren. Es gibt die folgenden Möglichkeiten Matrizen auf den MATLAB–*workspace* zu bekommen:

- Wir geben die Elemente einer Matrix explizit ein
- Wir erzeugen sie über vordefinierte MATLAB Funktionen
- Wir erhalten sie als Ergebnis eigner Funktionen
- Wir lesen sie von einem Datenträger ein

Aus Gründen der Übersichtlichkeit werden die folgenden Operationen anhand einer 3×3 Matrix exemplarisch vorgeführt. Explizit geben wir eine Matrix in der folgenden Weise ein:

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

m =

     1     2     3
     4     5     6
     7     8     9

>> m=[1,2,3;4,5,6;7,8,9]

m =

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

Wie zu erkennen ist, erzeugen beide Anweisungen eine 3×3 Matrix. Die Elemente werden also in eckige Klammern gefaßt und durch Semikolon, Komma oder blank getrennt. Zu beachten ist, daß Spalten sowohl durch Komma oder blank getrennt werden können. Einzelne Elemente der Matrix werden durch runde Klammern (Zeilenelmente, Spaltenelemente) angesprochen, z.B.

```
>> m(2,2)

ans =

     5
```

ist das Element von Zeile 2 und Spalte 2. Durch die folgende Syntax können ganze Zeilen der Spalten einer Matrix separat angesprochen werden:

```
>> vz=m(1,:)

vz =

     1     2     3

>> vs=m(:,1)

vs =

     1
     4
     7
```

Wie in diesem Beispiel zu erkennen ist, wurde der Zeilenvektor `vz` und der Spaltenvektor `vs` aus der Matrix `m` extrahiert. Mit `m(:)` werden alle Elemente von `m` angesprochen und aneinandergereiht als Spaltenvektor ausgegeben.

Mit der Funktion `size` können Sie die Anzahl der Zeilen und Spalten einer Matrix ermitteln:

```
>> [Zeilen,Spalten]=size(m)

Zeilen =

     3

Spalten =

     3
```

`Zeilen` und `Spalten` sind hier Variablen, denen die entsprechenden Werte zugewiesen werden. Weiterhin kann – zusätzlich zu dem Befehl `size` – die Länge eines Vektors mit dem Befehl `length` ermittelt werden:

```
>> length(vz)

ans =

     3
```

Sehr leicht können in MATLAB Zeilenvektoren in der folgenden Weise erzeugt werden: `v = von : Schrittweite : bis`.
Beispiel:

```
>> v=1:4

v =

     1     2     3     4

>> v=1:0.5:4

v =

    1.0000 1.5000  2.0000  2.5000  3.0000  3.5000 4.0000
```

Zu beachten ist, daß MATLAB eine Schrittweite von 1 verwendet wenn keine explizite Angabe gemacht wird. Aus der Matrix `m` kann nun eine Untermatrix `um` in der folgenden Weise extrahiert werden:

```
>> um=m(1:2,2:3)

um =

     2     3
     5     6
```

Also die Zeilenelemente von 1 bis 2 und die Spaltenelemente von 2 bis 3. Mit der Funktion `linspace` können Vektoren erzeugt werden, indem man den Wert der ersten, letzten, sowie die Anzahl der Komponenten (drittes Argument im Aufruf) vorgibt. Beispiel:

```
a=linspace(0,pi,10)

a =
  Columns 1 through 7
  0   0.3491  0.6981  1.0472  1.3963  1.7453  2.0944
  Columns 8 through 10
    2.4435  2.7925  3.1416
```

Mit der Funktion `logspace` ist es möglich Vektoren zu erzeugen deren Komponenten einen logarithmischen Abstand haben. In MATLAB sind mehrere Funktionen implementiert, die spezielle Matrizen erzeugen können. Hier sind einige wichtige Matrizen aufgelistet:

`zeros`	Nullmatrix
`eye`	Einheitsmatrix
`diag`	Diagonal Matrix und die Diagonalen einer Matrix
`hilb`	Hilbert–Matrix
`toeplitz`	Toeplitz–Matrix
`ones`	Matrix deren Elemente alle 1 sind
`rand`	Gleichverteilte Zufallsmatrix
`randn`	Normalverteilte Zufallsmatrix
`[]`	Leere Matrix

Beispiel: Toeplitz:

```
>> t=toeplitz(1:4)

t =

     1     2     3     4

     2     1     2     3

     3     2     1     2

     4     3     2     1
```

Weitere Informationen erhalten Sie wiederum mit `help` *name der Matrix.* Mit `isequal` können Matrizen auf Gleichheit kontrolliert werden. Mit `isempty` kann überprüft werden, ob eine Matrix leer ist. In MATLAB können die folgenden Matrixoperationen durchgeführt werden:

`A+B`	Addition $A + B$
`A-B`	Subtraktion $A - B$
`A*B`	Multiplikation AB
`A/B`	rechte “Division” löst $XA = Y$ nach X
`A\B`	linke “Division” löst $AX = Y$ nach X
`A^p`	Potenzieren A^p
`A'`	Konjugiert Transponieren A^T_*
`A.'`	Transponieren A^T
`kron(A,B)`	Kronecker Tensorprodukt $A \otimes B$
`inv(A)`	Inverse der Matrix A

Beispiel:

```
>> m1=[1,2;3,4]


m1 =


     1     2

     3     4


>> m2=[5,6;7,8]


m2 =


     5     6

     7     8


>> m=m1*m2
```

```
m =

    19    22
    43    50
```

Die “Division” von Matrizen stellt keine Matrixoperation im Sinne der linearen Algebra dar und muß daher genauer erläutert werden. Sei `A` eine $n \times n$ Matrix und `b` ein Spaltenvektor der Länge n, dann ist

```
x=A\b
```

die Lösung des linearen Gleichungssystems `Ax=b`. Diese “Linksdivision” löst das Gleichungssystem mit Hilfe des Gauß–Verfahrens.

Elementweise Operationen

Weiterhin ist es möglich, die Operationen `*`,`^`,`/` und `\` elementweise durchzuführen. Hierfür sind die folgenden Operatoren notwendig:

`A.*B`	Multiplikation
`A./B`	rechte Division
`A.\B`	linke Division
`A.^p`	Potenzieren

```
>> m1=[1,2;3,4]

m1 =

     1     2
     3     4

>> m2=[5,6;7,8]
```

```
m2 =

     5     6
     7     8

>> m=m1.*m2

m =

     5    12
    21    32
```

Ein weiteres Beispiel für elementweise Operationen ist Verwendung von Matrizen in mathematischen Funktionen.
Beispiel:

```
>> a=[0,1;log(5),-1];
>> exp(a)
ans =
    1.0000    2.7183
    5.0000    0.3679
```

Sparse–Matrizen

Im allgemeinen geht MATLAB davon aus, daß dicht besetzte Matrizen vorliegen. Sind aber von einer Matrix viele Elemente Null (z.B. 95%), dann spricht man von einer dünn besetzten oder sparse–Matrix. Man kann in MATLAB zwischen zwei verschiedenen Speichermodi: `full` und `sparse` wählen, wobei `full` standardmäßig eingestellt ist. Im sparse–Modus werden hierbei nur die von Null verschiedenen Elemente als eindimensionales Feld

mit ihren Zeilen– und Spaltenindizes abgespeichert. Mit Hilfe des sparse–Modus können Algorithmen in bezug auf Speicher– und Zeitaufwand verbessert werden.
Beispiel:

```
>> a=[1,2,0;4,0,0;0,8,0]


a =


     1     2     0
     4     0     0
     0     8     0


>> f=sparse(a)


f =
   (1,1)        1
   (2,1)        4
   (1,2)        2
   (3,2)        8
```

Weiter Funktionen zum Thema "sparse–Matrizen" sind im Anhang unter *Sparse Matrix Functions* zu finden.

2.3 Visualisierung in MATLAB

2D–Plots

Die `plot` Funktion ist eine der grundlegenden Graphikfunktionen in MATLAB. Mit ihr können lineare 2D–Darstellungen realisiert werden. Sind `x` und `y` zwei Vektoren mit `l=length(x)=length(y)` dann öffnet der Befehl

`plot(x,y)` ein Graphikfenster[1] und stellt die Elemente des Vektors `y` gegen die Elemente des Vektors `x` dar. Die einzelnen Punkte (`x(n)`,`y(n)`), $n = 1, 2, \ldots, l$ werden durch Linen verbunden, so daß ein Polygonzug entsteht. Beispiel:

```
>> x=0:0.1:4*pi;
>> y=sin(x);
>> plot(x,y)
```

Mehrere Plots in einem einzigen Graph können Sie z.B. in der folgenden Weise realisieren:

```
>> x=0:0.1:4*pi;
>> y1=sin(x);
>> y2=sin(x-1);
>> y3=2*sin(x);
>> plot(x,y1,x,y2,x,y3)
```

In diesem Zusammenhang soll auch auf die Funktion `hold` verweisen werden. Mit `hold on` erreicht man, daß weitere Graphen in ein bestehendes Graphikfenster hinzugefügt werden. Mit `hold off` kann man wieder in den Standardmodus zurückkehren.

In den obigen Beispielen legt MATLAB den Darstellungsbereich selbst fest. Mit dem Befehl `axis` (s. `help axis`) können sie diesen jedoch selbst festlegen. Weiterhin können Sie u.a. mit `title`, `xlabel` und `ylabel` die Plots beschriften. Beispiel (siehe auch Abbildung 3):

```
>> x=0:0.1:4*pi;
>> y1=sin(x);
```

[1] mit dem Befehl figure können beliebig viele Graphikfenster geöffnet werden. Mit Subplot kann ein Graphikfenster zerlegt werden

```
>> y2=sin(x-1);
>> y3=2*sin(x);
>> plot(x,y1,x,y2,x,y3);axis([0,4*pi,-2.5,2.5]);...
title('test plot');xlabel('x-values');ylabel('Amplitude');
```

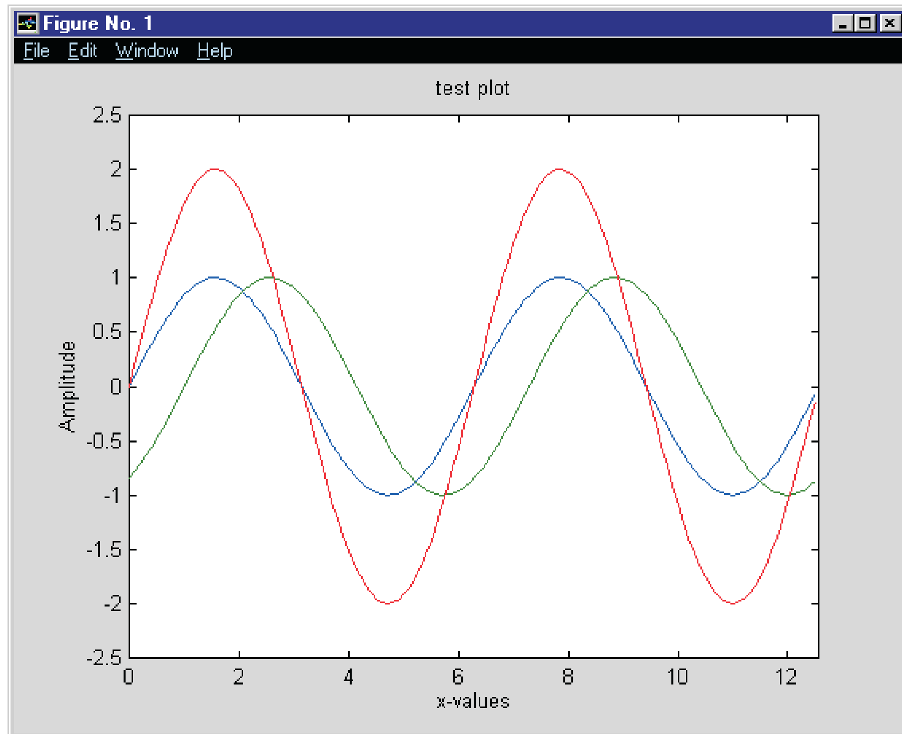

Abbildung 3: Beispiel zur plot Funktion

An dieser Stelle sei darauf hingewiesen, daß in MATLAB auszugebende stings, d.h. Zeichenketten, immer in Hochkommata gehören. Weiterhin können auch andere Darstellungsformen gewählt werden, z.B.

```
plot(x,y1,'b.')
```

stellt die einzelnen Elemente von `y` gegen `x` als blaue Punkte dar. Für eine Auflistung der verschiedenen Darstellungsformen mittels `plot` siehe `help plot`. Mit MATLAB können natürlich auch u.a. Balkendiagramme

(`bar`), Kreisdiagramme (`pie`) oder auch Histogramme (`hist`) geplottet werden. Parametrisierte Kurven können mit `ezplot` dargestellt werden. Für das plotten von Funktionen bietet sich die Funktion `fplot` besonders an. Einen Überblick über alle Plotfunktionen kann sich mit `help graph2d` für 2D Plots und mit `help graph3d` für 3D Plots verschafft werden. Mit `help specgraph` erhalten Sie eine Liste von Spezialplots. Im folgenden soll nur noch kurz auf 3D Plots eingegangen werden.

3D–Plots

Damit MATLAB die Funktionswerte einer Funktion zweier unabhängiger Variablen darstellen kann, muß ein 2D Gitter in der x–y Ebene erzeugt werden. Ein 2D Gitter wird in MATLAB über zwei Matrizen definiert. Eine Matrix beinhaltet die x–Koordinaten und die andere die y–Koordinaten. Die Erstellung derartiger Gittermatizen kann in MATLAB durch die Funktion `meshgrid` automatisiert werden. Als Beispiel soll nun die Funktion $z = f(x,y) = e^{-x^2-y^2}$ über dem Quadrat $[-2,2] \times [-2,2]$ dargestellt werden mit der `mesh` oder `surf` Funktion (siehe auch Abbildung 4):

```
>> xx=-2:0.01:2; yy=xx; [X,Y]=meshgrid(xx,yy);...
Z=exp(-X.^2-Y.^2); mesh(X,Y,Z);
```

In diesem Zusammenhang soll auch auf die Funktion `surf` verwiesen werden. Mit dem Befehl `plot3` können 3D Kurven erstellt werden, z.B.

```
>> t=.01:.01:30*pi;x=cos(t);y=sin(t);...
z=sqrt(t);plot3(x,y,z);
```

Skalierungen und Achsenbeschriftungen können natürlich auch für 3D Plots erzeugt werden (siehe `help graph3d`).

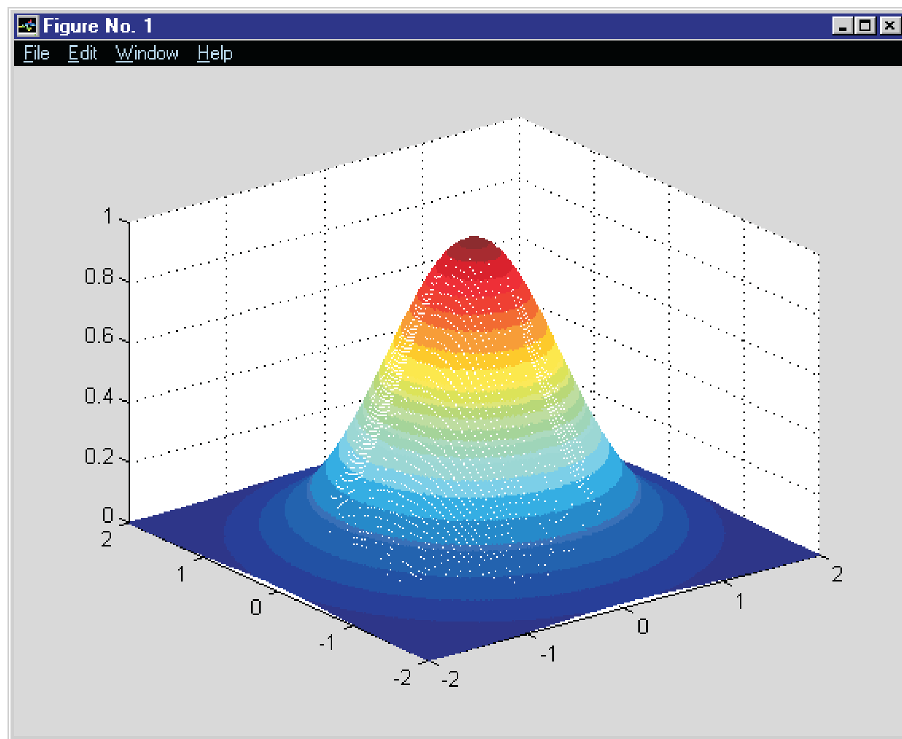

Abbildung 4: Beispiel zur meshgrid und mesh Funktion

Ausgabe von Graphiken

Mit dem Befehl `print` können Graphiken in verschiedenen Formaten in Dateien oder auf dem Drucker ausgegeben werden.

Beispiel: Die folgende Anweisung speichert die aktuelle Graphik (*figure*) unter dem Namen `test.eps` im EPS (*encapsulated postscript*) – Format.

```
>> print -deps test.eps
```

Einlesen von Bildern

Matrizen können als Bilder dargestellt werden, wobei die einzelnen Elemente einer Matrix die Helligkeit oder die Farbe der betreffenden Bildpunkte bezeichnen.

Beispiel: Im Verzeichnis `matlab\toolbox\matlab\demos` finden sich einige Beispielbilder, die ebenfalls die zugehörigen colormaps enthalten, z.B.

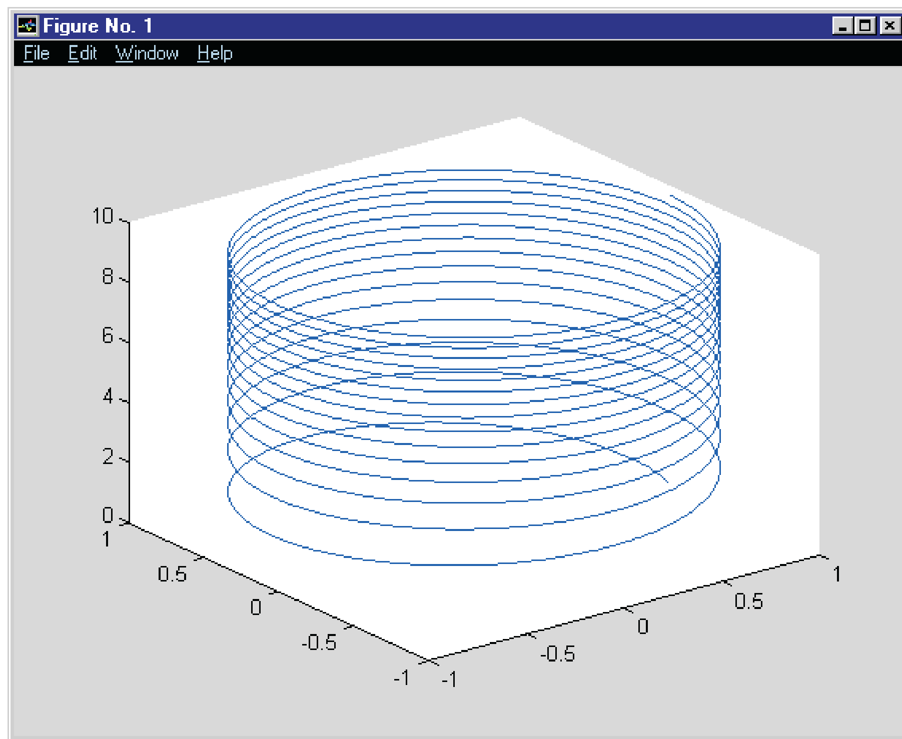

Abbildung 5: Beispiel zur plot3 Funktion

clown.mat, earth.mat, durer.mat und gatlin.mat. Sie können z.B. das Bild gatlin.mat laden und sich die entsprechenden Variablen ansehen:

```
>> load gatlin
>> whos
  Name            Size           Bytes  Class

  X             480x640        2457600  double array
  caption         2x63             252  char array
  map            64x3             1536  double array

Grand total is 307518 elements using 2459388 bytes
```

Die Matrix `X` beschriebt die Bildpunkte. Die Variable `map` ist die zugehörige colormap. Der char–array `caption` beinhaltet Informationen über das Bild:

```
>> caption

caption =

1964 photo from the Gatlinburg Conference on Numerical Algebra.
Wilkinson, Givens, Forsythe, Householder, Henrici, and Bauer.
```

Darstellen können Sie das Bild nun mit den folgenden Befehlen:

```
>> image(X);colormap(map);
```

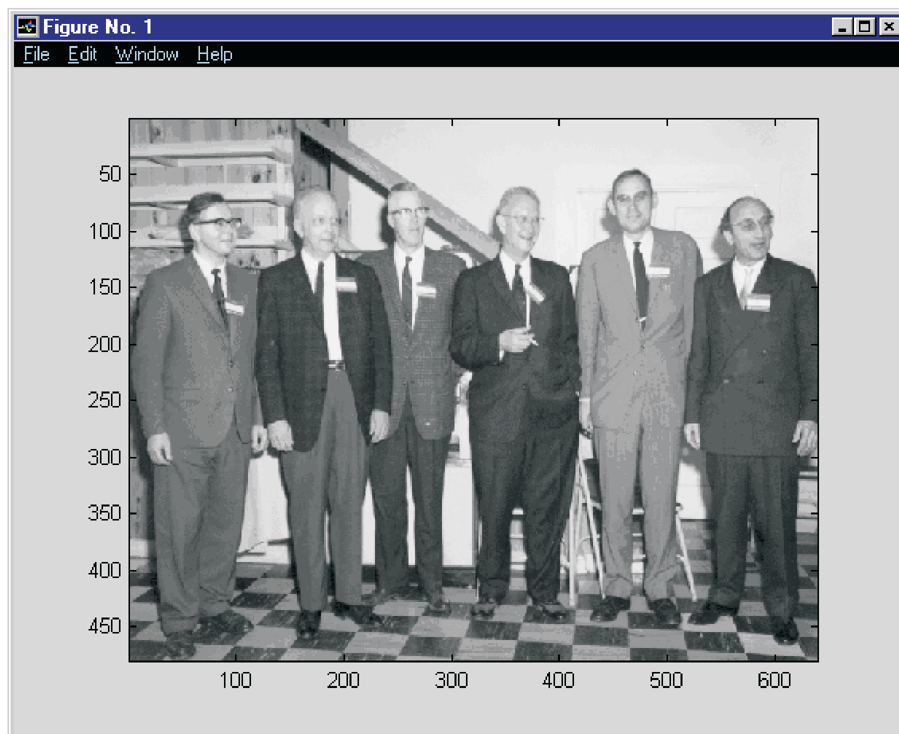

Abbildung 6: Beispiel zur Darstellung eines Bildes

3 Programmierung in MATLAB

Im interaktiven Modus wurden bisher Anweisungen zeilenweise eingeben und verarbeitet. Dieses Vorgehen ist zur Erstellung von Algorithmen natürlich wenig zweckmäßig. Hierfür eignen sich MATLAB–Programme, sog. `m-files`, die mit einem Editor erstellt werden können. MATLAB stellt einen eignen Editor zur Verfügung. Sie können diesen öffnen, indem Sie vom Kommandofenster aus `File->New->M-file` betätigen. Diese `m-files` werden als `*.m` Dateien auf dem Datenträger hinterlegt. Man unterscheidet zwischen zwei Arten von `m-files`: den `script-files` und den `function-files`. Das Erstellen dieser files bildet den Kern bei der Arbeit mit MATLAB.

3.1 Script–Files

Ein `script-file` ist eine Folge von MATLAB–Anweisungen die sukzessiv abgearbeitet wird. Ein `script-file` wird ausgeführt, indem es – ohne das Suffix `.m` – aufgerufen wird.
Beispiel: Mit dem Editor wurde das folgende `m-file` erstellt

```
s1=sin(linspace(0,4*pi,10));
s2=sin(linspace(0,4*pi,20));
s3=sin(linspace(0,4*pi,100));

subplot(311);plot(s1)
subplot(312);plot(s2)
subplot(313);plot(s3)
```

und unter dem Namen `prog1.m` in das aktuelle Arbeitsverzeichnis gespeichert. Der Aufruf `prog1` im interaktiven Modus führt nun dazu, daß das Programm sukzessiv abgearbeitet wird. Es wird ein Fenster geöffnet und 3 ver-

schieden abgetastete Sinus–Signale dargestellt (siehe Abbildung 7). In einem

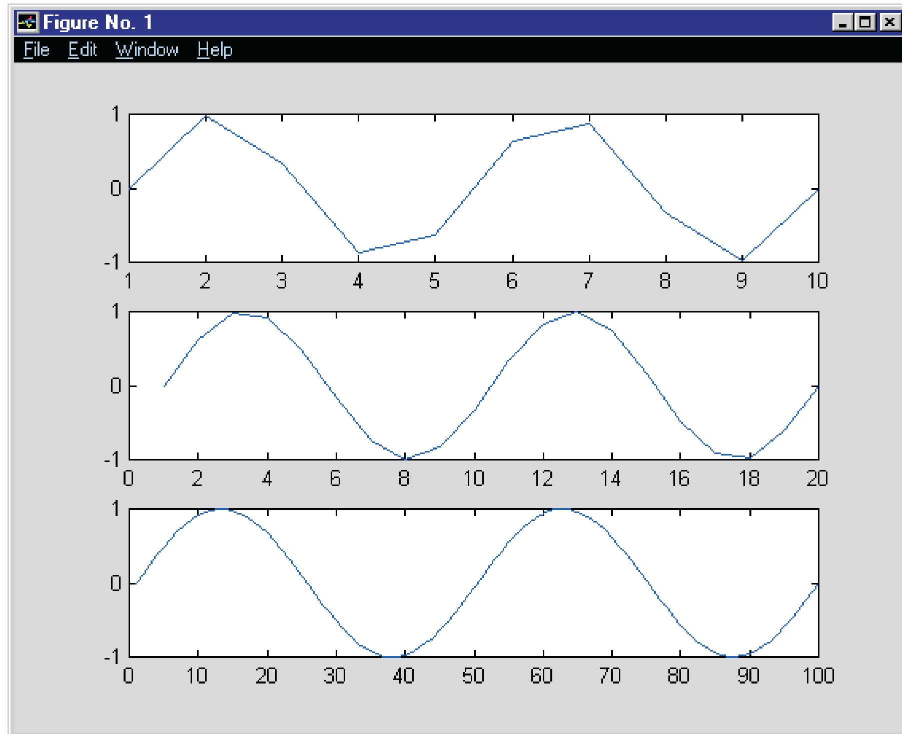

Abbildung 7: Darstellung durch prog1

derartigen `script-file` kann natürlich jede beliebige MATLAB-Funktion aufgerufen werden.

3.2 Function–Files

MATLAB ist ein offenes System dessen Funktionalität ständig durch den Benutzer erweitert werden kann. Oftmals sind Funktionen bzw. Befehle wünschenswert, die MATLAB nicht direkt bereitstellt. In diesem Fall können eigene Funktionen geschrieben werden, sog. `function-files`, und die vorhandenen MATLAB-Funktionen ergänzen. In `function-files` sind Variablen sog. lokale Variable, da sie nur in dem `function-file` präsent sind. Die Übergabe einzelner Variablen erfolgt über eine Parameterliste im Funk-

tionsaufruf. Dies geschieht analog zum Aufruf von Standard–MATLAB–Funktionen. Der Rumpf eines `function-files` besitzt die folgende Struktur:

```
function [out_1,...,out_n]=Funktionsname(in_1,...,in_m)
        Folge von Anweisungen
```

Das Schlüsselwort `function` kennzeichnet das `function-file`. Die `m` Variablen `in_1,...,in_m` werden der Funktion als Eingabeargumente übergeben. Die Funktion liefert die `n` Ausgabeargumente `out_1,...,out_n` zurück. Eine Funktion die keine Eingabe– oder Ausgabeargumente besitzt, ist ebenfalls zulässig.

Beispiel: Der folgenden Funktion mit dem Namen `test` werden zwei Vektoren `a` und `b` übergeben. Als Ausgabe erhält man den Vektor `c=a+b` sowie die eukldische Norm von `c`, also ein Skalar.

```
function [c,nc] = test(a,b)
c=a+b;
nc=norm(c);
```

Im interaktiven Modus führt nun der Aufruf dieser Funktion zu der folgenden Ausgabe:

```
>> a=[1;2;3;4];b=[5;6;7;8];
>> [c,nc]=test(a,b)


c =


     6
     8
    10
```

```
    12


nc =

   18.5472
```

Etwas 'eleganter' kann die Funktion wie folgt realisiert werden:

```
function [c,nc] = test(a,b)
% Diese Funktion berechnet c=a+b und die euklidsche Norm von c
% Syntax: [c,norm_von_c] = test(a,b)
% Es werden jeweils 2 Eingabe- und Ausgabeargumente verlangt
if (nargin ~=2 | nargout~=2)
    help test
else
c=a+b;
nc=norm(c);
end
```

Durch `%` wird in `m-files` ein Kommentar gekennzeichnet, d.h. eine Zeile die nicht interpretiert wird. Die ersten Kommentarzeilen von Funktionen können mit `help` abgefragt werden:

```
>> help test

  Diese Funktion berechnet c=a+b und die euklidsche Norm von c
  Syntax: [c,norm_von_c] = test(a,b)
  Es werden jeweils 2 Eingabe- und Ausgabeargumente verlangt
```

Die `if` – `else` Anweisung hat zur Folge, daß das Programm nur ausgeführt wird, wenn genau 2 Eingabe– und Ausgabeargumente vorliegen. Anderenfalls wird `help test` aufgerufen. Auf diese Programmflußsteuerungen wird im nächsten Kapitel eingegangen.

Sollen mehrere Funktionen mit der gleichen Variable arbeiten, so muß diese Variable als `global` gekennzeichnet werden (bevor sie von einer Funktion aufgerufen wird). Das Wort `global` ist in diesem Fall als Präfix für die Variable zu verwenden.

Im allgemeinen sollten folgende Regeln beachtet werden, wenn eine Funktion erstellt werden soll:

- Name des `m-files` und Funktionsname müssen übereinstimmen.
- Das Wort `function` muß in der ersten Zeile stehen.
- Die Kommentarzeilen nach der ersten Zeile dienen als Hilfstext. Weiterhin wird die erste Kommentarzeile mit dem `lookfor`–Befehl analysiert.
- In einem `function-file` können `script-files` aufgerufen werden. Dieses `script-file` wird dann nur im *Function-Workspace* ausgewertet.
- Mit `nargin` und `nargout` kann die Anzahl der Eingabe– und Ausgabeargumente festgestellt werden.
- Eine Inter-Funktionskommunikation ist über `globale` Variablen möglich.

3.3 Steuerstrukturen

Steuerstrukturen dienen zur Kontrolle des Programmflusses. Wie viele andere Programmiersprachen auch, bietet MATLAB die Möglichkeit den sequentiellen Ablauf eines Programms durch *Schleifen* (iterative Anweisungen) und *Verzweigungen* (bedingte Anweisungen) zu manipulieren.

3.3.1 Schleifen

Die for–Schleife

Bei for–Schleifen kann festgelegt werden, wie oft eine Folge von Anweisungen ausgeführt werden soll. Da for–Schleifen in MATLAB eine etwas andere Struktur besitzen als z.B. in C, soll mit der allgemeinen Syntax begonnen werden:

```
for Variable = Matrix
       Folge von Anweisungen
end
```

Die Variable ist in jedem Schleifendurchlauf ein Skalar oder ein Spaltenvektor. Die Folge von Anweisungen wird so oft ausgeführt, wie die Matrix Spalten besitzt. Nach jeder Iteration wird die Variable gleich der nächsten Spalte der Matrix gesetzt.
Beispiel:

```
M=[1 2; 3 4];
for k=M
   e=k
end
```

Im ersten Durchlauf wird dem Vektor `e` die erste Spalte der Matrix zugewiesen. Im zweiten Durchlauf die zweite Spalte:

```
e =

     1
     3



e =

     2
     4
```

Im allgemeinen wird man for–Schleifen jedoch in der folgenden Weise anwenden:

```
for Variable = von : Schrittweite : bis
       Folge von Anweisungen
end
```

Einer Variable wird also ein Anfangswert `von` (Skalar) zugewiesen. Nachdem die Folge von Anweisungen einmal ausgeführt wurde, wird die Variable um den Wert `Schrittweite` (Skalar) erhöht und die Folge von Anweisungen wird erneut ausgeführt. Dies geschieht solange, wie die Variable kleiner gleich `bis` (Skalar) ist bzw. die Anzahl der Elemente des Spaltenvektors `von:Schrittweite:bis` ist äquivalent zur Anzahl der Schleifendurchläufe.

Als Beispiel soll das folgende `script-file` betrachtet werden:

```
for k=1:1:5
   e(k)=k/10;
end
e
```

Wird dieses file gestartet, so erhält man die folgende Ausgabe:

```
e =


    0.1000    0.2000    0.3000    0.4000    0.5000
```

Die for–Schleife wurde also 5 mal durchlaufen und dem Vektor `e` die Werte `k/10` zugewiesen. Zu beachten ist, daß es keinen Einfluß hat, wenn die Variable während der Durchläufe neu gesetzt wird:

```
M=[1 2; 3 4];
for k=1:1:2
   k
   k=100
end
```

Ausgabe:

```
k =
     1
k =
   100
k =
     2
k =
   100
```

Die Variable wird also nach jedem Durchlauf neu zugewiesen und die Schleife wird nicht abgebrochen, obwohl `k`=100 gesetzt wird, was ja größer als 2 ist. Dies ist notwendig, um zu gewährleisten, daß sich die Anzahl der Durchläufe nach der Anzahl der Spalten richtet: Hier also nach der Anzahl der Elemente

des Spaltenvektors `1:1:2` (bzw. `1:2`, da die Schrittweite von 1 ja als default vorgegeben ist).

while–Schleifen

Bei while–Schleifen wird die Anzahl der Wiederholungen einer Folge von Anweisungen durch Vergleichsoperatoren, Vergleichsfunktionen und logische Bedingungen kontrolliert.

Vergleichsoperatoren:

`<`	kleiner
`<=`	kleiner gleich
`>`	größer
`>=`	größer gleich
`==`	gleich
`~=`	ungleich

Vergleichsfunktionen:

`any`	WAHR, wenn irgend ein Element ungleich 0 ist
`all`	WAHR, wenn alle Elemente ungleich 0 sind
`find`	Findet Indizes von Elementen ungleich 0

Sehr nützlich kann hier die `find` Anweisung eingesetzt werden, z.B. `find(x>50)` findet alle Elemente des Vektors die größer als 50 sind. Logische Bedingungen sind

`&`	UND
`|`	ODER
`~`	NICHT

Eine while–Schleife besitzt grundsätzlich die folgende Struktur:

```
while Ausdruck
   Folge von Anweisungen
end
```

Die while–Schleife wird als eine abweisende Schleife bezeichnet, da der Ausdruck ausgewertet wird, bevor die Folge von Anweisungen ausgeführt wird. Ist der Ausdruck (für alle Elemente) ungleich 0 (WAHR), so wird die Folge von Anweisungen ausgeführt. Ist der Ausdruck 0 (FALSCH), so werden die Anweisungen nicht abgearbeitet und das Programm wird in der ersten Zeile nach der Schleife fortgesetzt.

Beispiel:

```
k=0;
while k<=5
   k=k+1;
end
k
```

Als Ausgabe erhält man `k=6`. Erst wenn `k` den Wert 6 annimmt, ist der Ausdruck `k<=5` FALSCH und die Schleife wird beendet, d.h. die Anweisungen nicht mehr ausgeführt.

Mit `break` kann sowohl die while– als auch die for–Schleife abgebrochen bzw. vorzeitig beendet werden. Liegen geschachtelte Schleifen vor, so springt der `break` Befehl direkt in die übergeordnete Schleife. Mit `return` kann man bei beiden Schleifen in das Programm, aus dem die Funktion aufgerufen wurde, zurückkehren.

Allgemeines zu Schleifen in MATLAB

MATLAB ist in der Verarbeitung von Schleifen nicht sehr effizient. Deshalb sollten in MATLAB Schleifen vermieden werden, wenn dies möglich ist. Viele Operationen, die in universellen Programmiersprachen Schleifen benötigen, können in MATLAB durch die Vektor–Matrix Operationen ausgeführt werden. Mann nennt diese Äquivalentverarbeitung durch Vektor–Matrix Operationen *Vektorisierung*.

Ein einfaches Beispiel:

```
a=[1,2,3,4,5];
b=[6,7,8,9,10];
for k=1:length(a)
   c(k)=a(k)+b(k);
end
c
```

Das obige Programm addiert mit Hilfe einer for–Schleife die Vektoren `a` und `b`. In MATLAB kann dies einfach durch die Operation `c=a+b` realisiert werden. Für viele Operationen bei denen normalerweise Schleifen involviert werden müssen, stellt MATLAB schon entsprechende Funktionen bereit. So kann z.B. die Aufsummierung eines Vektors einfach über den Befehl `sum`[2] erfolgen.

Wenn dennoch auf Schleifen zurückgegriffen werden muß und in diesen Schleifen Matrizen bzw. Vektoren erzeugt werden (z.B. oben der Vektor `c`), dann sollten die zu erzeugenden Matrizen bzw. Vektoren zuvor initialisiert werden. Dies kann z.B. durch den Befehl `zeros` geschehen[3]. Die Vorinitialisierung hat ein besseres Speichermanagement zur Folge und führt zu effizienteren Algorithmen. Das obige Beispiel lautet mit Vorinitialisierung:

```
a=[1,2,3,4,5];
b=[6,7,8,9,10];
c=zeros(1,length(a)); % Initialisierung:
                      % Spaltenvektor mit length(a) Nullelementen
for k=1:length(a)
```

[2]siehe auch Anhang: Column–Wise Data Analysis

[3]Es ist oftmals sinnvoll die Matrizen bzw. Vektoren mit Null zu initialisieren, z.B bei Summationen

```
   c(k)=a(k)+b(k);
end
c
```

3.3.2 Verzweigungen

Die if–Anweisung

Die if–Anweisung wertet einen logischen Ausdruck aus und läßt eine Gruppe von Anweisungen ausführen, wenn er WAHR ist. Die allgemeine (mehrseitige) Syntax ist:

```
if Ausdruck1
   Folge von Anweisungen
elseif Ausdruck2
   Folge von Anweisungen
elseif Ausdruck3
   Folge von Anweisungen
   .
   .
   .
else
   Folge von Anweisungen
end
```

Bei der elseif–Anweisung werden die Ausdrücke sequentiell ausgewertet. Es wird nur die Folge von Anweisungen nach dem ersten wahren Ausdruck ausgeführt, danach ist die komplette Kette beendet. Wenn keiner der Ausdrücke WAHR ist, wird die Folge von Anweisungen nach der else–Anweisung ausgeführt.

Beispiel: Das Programm

```
for k=1:5
   if k==1
      sprintf('k= %i, %s',k,'Text1') %Anweisung 1
   elseif k==4
      sprintf('k= %i, %s',k,'Text2') %Anweisung 2
   else
      sprintf('k= %i, %s',k,'Text3') %Anweisung 3
   end
end
```

bewirkt die folgende Ausgabe:

```
ans =
k= 1, Text1
ans =
k= 2, Text3
ans =
k= 3, Text3
ans =
k= 4, Text2
ans =
k= 5, Text3
```

Wie zu erkennen ist, wird für $k = 1$ Anweisung 1 ausgeführt (Text 1 wird ausgegeben[4]). Für $k > 1, k \neq 4$ wird Anweisung 3, d.h. die Anweisung nach `else` ausgeführt. Nur für $k = 4$ wird – so wie es die elseif–Anweisung fordert – Anweisung 2 ausgeführt. Für eine zweiseitige Auswahl gilt einfach:

```
if Ausdruck
```

[4]Die formatierte Ausgabe eines Textes kann mit 'sprintf' erfolgen. In diesem Zusammenhang soll auch auf den Befehl 'disp' verwiesen werden

```
   Folge von Anweisungen
else
   Folge von Anweisungen
end
```

Die switch–Anweisung

Die switch–Anweisung führt dazu, daß Folgen von Anweisungen entsprechend dem Wert einer Variablen oder eines Ausdrucks ausgeführt werden. Mit den Befehlen `case` und `otherwise` werden diese Folgen eingegrenzt. Wichtig ist, daß nur die erste Übereinstimmung ausgeführt wird und dann switch–Umgebung verlassen wird. Die allgemeine Syntax lautet:

```
switch Variable
   Folge von Anweisungen
case Konstante1
   Folge von Anweisungen
case Konstante2
   Folge von Anweisungen
   .
   .
   .
otherwise
    Folge von Anweisungen
end
```

Beispiel:

```
switch var
  case 2
     disp('Fall 1: var ist gleich 2')
```

```
  case {4,5}
     disp('Fall 2: var ist gleich 4 oder var ist gleich 5')
  otherwise
     disp('Weder Fall 1 noch Fall 2')
end
```

Zu beachten ist, daß die switch–Anweisung mehrere Bedingungen in einem case–Fall behandeln kann, wenn die Bedingungen in einer sog. Zelle (durch die geschweiften Klammern erkennbar) zusammengefaßt wird (siehe in obigen Beispiel Fall 2). Die switch–Anweisung ist besonders zur Realisierung von Auswahltafeln o.ä. geeignet.

3.4 Einlesen und Speichern von Daten

In MATLAB können Daten in binärem MATLAB– und lesbaren ASCII-Format eingelesen und gespeichert werden. Binärdateien haben den Vorteil, daß sie schneller gelesen und geschrieben werden können als ASCII-Files und weniger Speicherplatz auf einem Datenträger benötigen. ASCII-Dateien haben hingegen den Vorteil, daß sie leicht – außerhalb von MATLAB – erzeugt werden können und mit einem beliebigen Editor bearbeitet werden können.

High–Level Befehle

Mit den Befehlen `load` und `save` stellt MATLAB high–level Befehle zum lesen und speichern von Daten bereit, die sowohl Binär – als auch ASCII–Format nutzen. Beispiel Binärformat:

```
>> a=[1,2,3;4,5,6;7,8,9];
>> save a
>> clear
```

```
>> load a
>> a
a =
     1     2     3
     4     5     6
     7     8     9
```

Ausführlich hieße es `save a.mat -mat` bzw. `load a.dat -mat`. MATLAB verwendet das Binärformat (mit dem Suffix mat) jedoch standardmäßig.
Beispiel ASCII–Format:

```
>> a=[1,2,3;4,5,6;7,8,9];
>> save a.dat -ascii
>> clear
>> load a.dat -ascii
>> a
a =
     1     2     3
     4     5     6
     7     8     9
```

Der Befehl `save` kann in beiden Fällen die Datei auch unter einem beliebigen Namen speichern. Mit dem obigen Beispiel kann dies in der folgenden Form geschehen:

```
>> a=[1,2,3;4,5,6;7,8,9];
>> save matrix.dat a -ascii
>> clear
>> load matrix.dat
```

```
>> whos
  Name      Size           Bytes  Class
  a         3x3               72  double array
Grand total is 9 elements using 72 bytes
```

Low–Level Befehle

Die low–level Befehle `fprintf`, `fscanf`, `fwrite` und `fread` können, ähnlich wie in C/C++, zum formatierten Einlesen und Speichern von Daten genutzt werden. Im folgenden soll nur auf die häufig verwendeten Funktionen zum Schreiben und Lesen von ASCII-Dateien eingegangen werden: `fprintf` und `fscanf`. Beispiel:

```
a=[1 2 3;4 5 6;7 8 9];
file=fopen('matrix.dat','w'); %File wird zur geoeffnet (schreiben)
[z,s]=size(a);
for m=1:z
   for n=1:s
      fprintf(file,'%g ',a(m,n)); %g bezeichnet floating point
   end
   fprintf(file,'\n'); %Zeilentrennung
end
fclose(file);
clear a;
file=fopen('matrix.dat','r');
a=fscanf(file,'%g',[z,s]);
a
```

Zur Spezifikation der verschiedenen Datentypen (integer, floating point etc.) sowie zur Darstellung der allgemeinen Syntax wird auf den *Helpdesk* verwiesen. Die Verwendung von low–level Befehlen macht immer dann Sinn,

wenn die Daten nicht mit einer rechteckigen Matrix bzw. mit einem Vektor beschrieben werden können, z.B. bei der Verwendung von bestimmten Header–Dateien. In der gleichen Form können die Befehle `fwrite` und `fread` für binäre Dateien verwendet werden.

4 Übung I

1. Es sei:

   $$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} -1 & 8 \\ 0 & -5 \\ 5 & -4 \end{pmatrix}, \mathbf{C} = \begin{pmatrix} -4 & 8 \\ 1 & -7 \end{pmatrix},$$

   $$\mathbf{D} = \begin{pmatrix} 5 & 6 & 10 \\ 1 & -7 & 0 \\ 4 & 9 & 1 \end{pmatrix}$$

   Berechnen Sie:

   1.1 $\mathbf{A} + \mathbf{B}$

   1.2 $\mathbf{C}^2$

   1.3 $\mathbf{DA}$

   1.4 $\mathbf{D}^T$

   1.5 $\mathbf{AB}$ (elementweise)

2. Es sei:

   $z_1 = -1 + 5i$ und $z_2 = 0.9e^{0.8i}$

   Berechnen Sie:

   2.1 $z = z_1 + z_1$

   2.2 $z = z_1 z_2$

   2.3 $z = \frac{z_1}{z_2}$

   2.4 Erzeugen Sie eine sog. 3×3 DFT (Diskrete Fourier Transformation)–Matrix.

   $$\mathbf{F}_3 := \left(e^{-2\pi ijk/3}\right)_{j,k=0}^{2}$$

2.5 Zeigen Sie, daß $\frac{1}{\sqrt{3}}\mathbf{F}_3$ eine unitäre Matrix[5] ist.

3. Plotten Sie den Graph der Runge–Funktion $f(x) = \frac{1}{1+25x^2}$ in dem Intervall $[-1, 1]$ (beliebige, aber sinnvolle Anzahl von Stützstellen) ! Beschriften Sie die Achsen !

4. Plotten Sie den Graph der Funktion $f(x, y) = (-x^2 + y^2)sin(y)$ über $[-2, 2] \times [-2, 2]$ mit `surf` (beliebige, aber sinnvolle Anzahl von Stützstellen) !

5. Erzeugen Sie eine 4×4 Hilbert–Matrix[6] $\mathbf{H}_4$!

   Aufgaben:

   5.1 Extrahieren Sie eine beliebige 2×2 Untermatrix !

   5.2 Extrahieren Sie die Diagonalelemente !

   5.3 Extrahieren Sie den zweiten Spaltenvektor !

   5.4 Extrahieren Sie den vierten Zeilenvektor !

   5.5 Berechnen Sie $\mathbf{b} = \mathbf{H}_4\mathbf{1}_4$, mit $\mathbf{1}_4 = (1, 1, 1, 1)^T$.

   5.6 Lösen Sie das Gleichungssystem $\mathbf{H}_4\mathbf{x} = \mathbf{b}$ nach dem Vektor $\mathbf{x}$ auf und berechnen Sie den relativen Fehler[7] von $\mathbf{x}$ als Näherung von $\mathbf{1}_4$.

   5.7 Wiederholen Sie Schritt 3.5 und 3.6 mit einer 50×50 Hilbert–Matrix $\mathbf{H}_{50}$ und $\mathbf{1}_{50}$.
   Was fällt auf ?

---

[5] Eine Matrix $\mathbf{M}$ mit komplexen Elementen für die gilt $\overline{\mathbf{M}}^T = \mathbf{M}^{-1}$ oder $\mathbf{M}\overline{\mathbf{M}}^T = E$ heißt unitäre Matrix. Bei reellen Matrizen fällt der Begriff "unitär" mit dem Begriff "orthogonal" zusammen.

[6] Zur Definition der Hilbert–Matrix siehe help hilb.

[7] Der relative Fehler eines Vektors $\mathbf{a}$ als Näherung eines Vektors $\mathbf{b}$ lautet: $\lambda = \frac{||\mathbf{a}-\mathbf{b}||_2}{||\mathbf{b}||_2}$, wobei $|| \cdot ||$ die euklidsche Norm bezeichnet (siehe help norm)

6. Gegeben sei der Vektor $\mathbf{a} = (5, 4, 2, 1, 3, 6, 7, 8, 9)$.

   Aufgaben:

   6.1 Bilden Sie die Summe über die Komponenten des Vektors !

   6.2 Berechnen Sie den Mittelwert des Vektors !

   6.3 Sortieren Sie den Vektor (aufsteigend) !

   Hinweis: Verwenden Sie die entsprechenden MATLAB Funktionen (siehe Skriptum / Anhang).

7. In MATLAB können Gleitpunktoperationen (flops) mit dem Befehl `flops` gezählt werden. Mit `flops(0)` wird der Zähler auf Null gesetzt. Die Ausführungszeit von Berechnungen kann mit den Befehlen `tic` und `toc` ermittelt werden.

   7.0 Löschen Sie die Variablen auf dem Workspace !

   7.1 Erzeugen Sie eine 1000×1000 Random–Matrix $\mathbf{R}$ mit `rand` !

   7.2 Bilden Sie eine 1000×1000 sparse–Bandmatrix $\mathbf{S}$ indem Sie mit `spdiags` die Diagonalen `-100:1:100` von $\mathbf{R}$ extrahieren !

   7.2 Stellen Sie die Matrix $\mathbf{S}$ mit dem Befehl `imagesc` dar ! Die entsprechende Skalierung erhalten Sie mit `colorbar`.

   7.3 Speichern Sie die Matrix $\mathbf{S}$ unter dem Namen $\mathbf{F}$ im full–Modus.

   7.4 Schauen Sie sich die Variablen auf dem Workspace an !
   Was fällt auf ?

   7.5 Ermitteln Sie die Anzahl der flops und die Berechnungszeit für die Operationen $\mathbf{F}^2$ und $\mathbf{S}^2$!
   Was fällt auf ?

Viel Spaß !

5 Übung II

1. Schreiben Sie ein Programm, das für Zufallszahlen x_k in [0,1], $k = 1, 2, \dots$ den Wert $y = \frac{x_k}{k}$ solange berechnet, wie gilt $x_k < 0.99$. Geben Sie die Werte x_k und y aus.

2. Programmieren Sie das Romberg–Verfahren mit Romberg–Schrittweiten $h_j = 2^{-j}$ $(j = 1 \dots , M)$ zur Integration einer gegebenen Funktion f über dem Intervall $[0, 1]$.

   Testen Sie das Verfahren für die folgenden Funktionen bis $M = 5$:

   i) $f(x) = x^{\frac{1}{4}}$
   $(\int_0^1 x^{\frac{1}{4}}\, dx = 0.8)$

   ii) $f(x) = \cos(\pi x) e^{-\pi x}$
   $(\int_0^1 \cos(\pi x) e^{-\pi x}\, dx = (1 + e^{-\pi})/(2\pi) = 0.1660326514\dots)$

   Geben Sie das Ergebnis in der Form

$$\begin{array}{ccc} T_{00} & & \\ T_{01} & T_{11} & \\ T_{02} & T_{12} & T_{22} \\ \vdots & \vdots & \dots \end{array}$$

   an! Hinweis. Sie können das Beispiel aus der Vorlesung zum Test Ihres Programmes verwenden. Nutzen Sie function–files.

3. Schreiben Sie ein Computerprogramm zur Lösung der folgenden Aufgabe!

   3.1 Berechnen Sie die Werte $p_{10}(\hat{x}_j)$ $j = 0, \dots , 200$ mit $\hat{x}_j := -1 + \frac{j}{100}$ des Interpolationspolynoms p_{10} der Funktion

$$f(x) = \frac{1}{1 + 25x^2}$$

an den Stützstellen $x_k := -1 + \frac{k}{5}$ $(k = 0, \ldots, 10)$.

3.2 Lösen Sie die gleiche Aufgabe wie in a) mit den Stützstellen $x_k := \cos \frac{(2k+1)\pi}{2 \cdot 11}$ $(k = 0, \ldots, 10)$. Das Interpolationspolynom sei q_{10}.

3.3 Plotten Sie f, p und q_{10} !

Hinweis: Ich finde die Aufgabe am einfachste mit dem Neville-Algorithmus. Aber Sie können natürlich auch andere Algorithmen verwenden.

Viel Spaß !