Einführung in MATLAB

Skript zur Vorlesung für Studierende der Masterstudiengänge
Marine Umweltwissenschaften und Umweltmodellierung
an der Carl von Ossietzky Universität Oldenburg
im Wintersemester 2020/2021

Cora Kohlmeier

1. Dezember 2020

Inhaltsverzeichnis

1 Einführung

MATLAB ist ein Softwarepaket für numerische Berechnungen und für die Visualisierung von großen Datenmengen. MATLAB steht für MATRIX LABORATORY. Im Unterschied zu vielen anderen Programmiersprachen arbeitet MATLAB Matrixorientiert. Leider ist MATLAB nicht kostenlos erhältlich. Wer eine ähnliche aber kostenlose Variante sucht, sollte mal bei Python oder Octave nachsehen. Auch R ist eine gute Alternative.
Die folgende Einführung ersetzt keinesfalls einen Programmierkurs, sondern kann nur einen kleinen Einblick in die Funktionsweise von MATLAB geben.
MATLAB kann man auf 2 Arten nutzen

- interaktiv: Anweisungen werden über die Tastatur eingegeben und sofort ausgeführt.
- als Programmiersprache: Anweisungen werden als so genannte m-Files abgespeichert. Im Commandofenster kann der Name des m-Files eingegeben werden (ohne.m). Alle Anweisungen im m-File werden dann nacheinander ausgeführt

2 MATLAB interaktiv

Nach dem Start von MATLAB öffnet sich das Kommandofenster (command window) und es erscheint das Prompt >>. Hier können nun Kommandos eingegeben werden. Mit der Pfeiltaste nach oben kann man das vorherige Kommando wiederholen (history).

2.1 Rechnen mit MATLAB

Die Rechenoperationen +,-,*,/ werden wie üblich verwendet, die Eingabe wird mit ↩ abgeschlossen

```
>> 3+5.5
ans =
    8.5000
```

Hierbei gilt wie üblich Punktrechnung vor Strichrechnung und es dürfen Klammern (rund) verwendet werden. Potenzen werden mit ^ eingegeben. Dezimalkommas werden, wie im Englischen allgemein üblich, als Punkt (decimal point) eingegeben.
Braucht man einen Ausdruck öfters, so kann man ihn als Variable definieren:

```
>> a= 5*5-3^2

a =

    16
```

Folgende Regeln sind bei der Definition von Variablen zu beachten:

- Ein Variablenname darf keine Sonderzeichen außer dem Unterstrich enthalten.
- Das erste Zeichen muss ein Buchstabe sein.

MATLAB unterscheidet zwischen Groß- und Kleinbuchstaben (upper case und lower case letters). Daher sind a und A verschiedene Variablen (man sagt, die Variablennamen sind case sensitive). Ein Semikolon am Ende der Eingabezeile bewirkt, dass MATLAB das Ergebnis zwar ausrechnet, aber nicht ausgibt.
Mit Variablen kann man nun rechnen

```
>> a=8/3
a =
    2.6667
>> b=3/2
b =
    1.5000
>> a*b
ans =
     4
```

2.1.1 Zahlformate

Wie man oben sieht, kürzt MATLAB die Ausgabe ab, rechnet aber genauer. Will man das Ausgabeformat ändern so geht dies mit `format`:

```
>> format long
>> a
a =
   2.666666666666667
```

Die kurze Ausgabe erhält man wieder mit `format short`.

2.1.2 Strings

Strings oder auch Zeichenketten werden häufig zur Beschriftung von Grafiken benötigt. Zeichenketten werden in einfache Anführungszeichen gesetzt. Will man beide Zeichenketten verbinden so geht dies mit strcat:

```
s1='Hello ';
s2='World!';
>> strcat(s1,s2)
ans =
HelloWorld!
```

Will man das Leerzeichen nach Hello erhalten, so geht dies mit

```
s1='Hello ';
s2='World!';
>> [s1 s2]
ans =
Hello World!
```

2.2 Vordefinierte Funktionen

Es gibt - wie auch in jedem Taschenrechner - vordefinierte Funktionen, z.B. die Wurzelfunktion (square root)

```
a = sqrt(2)/2
a =
0.7071
```

Im folgenden sind einige wichtige Funktionen angegeben:

Befehl	Funktion	Bemerkung
sin	Sinus	Argument in Bogenmaß
cos	Cosinus	Argument in Bogenmaß
tan	Tangens	Argument in Bogenmaß
atan	Arcustangens	Wertebereich [$-\pi/2$, $\pi/2$]
log	Logarithmus	zur Basis e
log10	Logarithmus	zur Basis 10
pi	π	=3.1415927...

2.3 Vektoren

Eine besondere Stärke hat MATLAB bei der Verarbeitung von Matrizen und Vektoren.
Zeilenvektoren werden wie folgt definiert:

```
>> v1=[1 2 3 4 5 6 7 8 9 10]
v1 =
     1     2     3     4     5     6     7     8     9    10
```

Einfacher kann man diesen Vektor auch mit

```
v1=1:10
```

erzeugen. Will man auch Zwischenwerte haben, so geht dies mit

```
>> v1=1:0.5:3
v1 =
    1.0000    1.5000    2.0000    2.5000    3.0000
```

Die Schreibweise `1:0.5:3` bedeutet: beginne mit 1 und zähle so oft 0.5 dazu, bis die Grenze 3 erreicht ist.
Einzelne Werte eines Vektors erhält man mit

```
>> v1(2)
ans =
    1.5000
```

Man sieht, das der erste Index eines Vektors 1 ist[1].
Ganze Bereiche eines Vektors erhält man durch die Angabe des Indexbereichs:

[1] Bei C und einigen anderen Programmiersprachen ist der erste Index 0

```
>> v1=1:0.5:3;
>> v1(2:4)
ans =
    1.5000    2.0000    2.5000
```

oder auch

```
>> v1=1:0.5:3;
>> v1(2:end)
ans =
    1.5000    2.0000    2.5000    3.0000
```

Ein Spaltenvektor erzeugt man wie folgt.

```
>> v2=[4;5;6]
v2 =
     4
     5
     6
```

Will man einen Zeilenvektor in einen Spaltenvektor umwandeln, so geht dies mit

```
>> v1=1:0.5:3;
>> v1'
ans =
    1.0000
    1.5000
    2.0000
    2.5000
    3.0000
```

Dies gilt umgekehrt natürlich genauso.
Bei MATLAB ist es auch möglich, Funktionen auf Vektoren los zulassen, z.B.

```
>> v1=1:0.5:3;
>> sqrt(v1)
ans =
    1.0000    1.2247    1.4142    1.5811    1.7321
```

Die Länge eines Vektors erhält man mit length:

```
>>v1=[1 2 3];
>> length(v1)
ans =
     3
```

2.3.1 Rechnen mit Vektoren

Addition und Subtraktion von Vektoren

```
>> v1=[1 2 3];
>> v2=[4 5 6 ];
>> v1+v2
ans =
     5     7     9
>> v1-v2
ans =
    -3    -3    -3
```

Hierbei müssen `v1` und `v2` natürlich Vektoren desselben Typs sein, d.h. gleich lang und entweder Zeilen- oder Spaltenvektoren.

Multiplikation mit einem Skalar

Die Multiplikation mit einem Skalar (Zahl) erfolgt elementweise:

```
>> v1=[1 2 3];
>> v1*0.5
ans =
    0.5000    1.0000    1.5000
```

Multiplikation zweier Vektoren

Einen Zeilenvektor kann man mit einem Spaltenvektor multiplizieren, wenn sie die gleiche Länge haben:

```
>> v1=[1 2 3];
>> v2=[1;0;2];
>> v1*v2
ans =
     7
```

In MATLAB gibt es aber zu diesen bekannten mathematischen Operationen noch weitere, so genannte elementweise Operationen.

Elementweise Addition

```
>> v1=[1 2 3];
>> v1+7
ans =
     8     9    10
```

Elementweise Multiplikation

Mit einem Punkt vor dem * (`.*`) wird die Operation elementweise ausgeführt:

```
>> v1=[1 2 3];
>> v2=[1 0 3];
>> v1.*v2
ans =
     1     0     9
```

Hierbei müssen `v1` und `v2` Vektoren desselben Typs sein, d.h. gleich lang und entweder Zeilen- oder Spaltenvektoren. Analog ist die elementweise Division (`./`) Potenzierung (`.^`) definiert.

Übung

`a = [1 4 6]; b = [-1 2 1];`
Überlege, was MATLAB ausgeben wird:
`a+b, a*2,a/2, a+3, a*b, a.*b, a./b`

2.4 Matrizen

Ähnlich wie Vektoren, werden Matrizen definiert. Hierbei werden die Zeilen durch ; getrennt.

```
>> A=[1 2 3; 4 5 6]
A =
     1     2     3
     4     5     6
```

Die Dimension der Matrix erhält man durch

```
>> A=[1 2 3; 4 5 6];
>> size(A)
ans =
     2     3
```

Hier bei gibt die erste Zahl die Anzahl der Zeilen, die zweite die Anzahl der Spalten an.
Die Transponierte erhält man durch

```
>> A=[1 2 3; 4 5 6];
>> A'
ans =
     1     4
     2     5
     3     6
```

Beim Zugriff auf einzelne Matrixelemente gibt der erste Index die Zeile, der zweite die Spalte an:

```
>> A=[1 2 3; 4 5 6];
>> A(2,1)
ans =
     4
```

Es ist auch möglich auf Bereiche zuzugreifen:

```
>> A=[1 2 3; 4 5 6];
>> A(2,2:3)
ans =
     5     6
```

Hier wird in Zeile zwei auf die Spalten 2 bis 3 zugegriffen.
Will man auf eine gesamte Spalte zugreifen, so geht dies mit :

```
>> A=[1 2 3; 4 5 6];
>> A(:,2)
ans =
     2
     5
```

Der Doppelpunkt bedeutet hier also: alle Zeilen. Analog kann man auch auf eine Zeile zugreifen.
Häufig benötigt man eine mit Nullen oder Einsen gefüllte Matrix. Die geht mit

```
>> zeros(2,3)
ans =
     0     0     0
     0     0     0
>> ones(2,3)
ans =
     1     1     1
     1     1     1
```

So kann man dann auch Zeilen oder Spaltenvektoren einer bestimmten Länge erzeugen:

```
>> zeros(1,3)
ans =
     0     0     0
>> ones(3,1)
ans =
     1
     1
     1
```

2.4.1 Rechnen mit Matrizen

Analog zum Rechnen mit Vektoren, kann man mit Matrizen rechnen, wobei man dabei auf die Dimensionen achten muss.

Analog der mathematischen Rechenregeln können Matrizen miteinander addiert, mit einem Skalar multipliziert, subtrahiert und multipliziert werden:

```
>> A=[1 2;3 4];
>> C=[2 3;8 0];
>> A+C
ans =
     3     5
    11     4
```

```
>> 10*A
ans =
    10    20
    30    40
>> B=[1 1 2;5 5 0];
>> A*B
ans =
    11    11     2
    23    23     6
```

Bei Dimensionen, die nicht zueinander passen, gibt es eine Fehlermeldung:

```
>> A=[1 2 ;3 4];
>> B=[1 1 2;5 5 0];
>> B*A
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Übung

1. `A = [1 4 5; 2 0 1; 2 3 6];` Überlege, was MATLAB ausgeben wird:

   `A', A(1,:), A(:,3), A(2,2), size(A)`

2. `B = rand(4,5)` erzeugt eine 4x5 Matrix mit zufällig bestimmten Elementen. Ersetze in der so erhaltenen Matrix B das erste Element in der ersten Zeile durch 0 und streiche die gesamte dritte Spalte von B.

3. Erzeuge eine 2x6 Matrix A mit Zufallselementen. Bilde daraus

   - eine Matrix B, die aus den ersten drei Spalten von A besteht
   - eine Matrix C, die aus der zweiten und fünften Spalte von A besteht.

3 Nutzung von m-Files

Hat man viele Anweisungen nacheinander auszuführen, ist es sinnvoll, alle Anweisungen in eine Datei zu schreiben und zu speichern. Der Dateiname muss dabei die Extension (das nach dem Punkt) `m` haben, z.B. `myfirst.m`. Ein solches so genanntes m-File darf alle MATLAB-Befehle enthalten. Eine solches m-File nennt man häufig auch MATLAB-Skript. MATLAB bietet einen Editor, um m-Files zu erzeugen. Mit File $\rightarrow$ New$\rightarrow$ M-File wird der Editor geöffnet und man kann ein Programm (mehrere Anweisungen) eingeben und speichern. Der Name des m-Files darf nicht mit einer MATLAB Funktion übereinstimmen, also z.B.nicht `sqrt.m`.
Wird der Name des m-Files im Kommandofenster als Befehl eingegeben (`myfirst`), so werden alle Befehle im File nacheinander ausgeführt. Wichtig hierbei ist, dass sich MATLAB im richtigen Pfad befindet. Wurde das m-File z.B. unter
`C:\Eigene Dateien\mymatlab`
gespeichert, so muss das aktuelle Verzeichnis entsprechend gewählt sein, damit MATLAB das Skript auch findet. Gegebenenfalls muss man unter `Current Directory` das richtige Verzeichnis auswählen. Im folgenden werden wir alle Programme als m-Files speichern, damit uns nichts verloren geht!!

4 Programmieren mit MATLAB

Was nützt einem nun eine Programmiersprache? Rechnungen kann ein Taschenrechner doch viel bequemer ausführen!

Häufig muss man ähnliche Rechenschritte oft hintereinander ausführen. Will man z.B. die Zahlen von eins bis 100 aufaddieren (und kennt die Gaußsche Formel nicht), so ist das sehr mühsam. Mit einer Programmiersprache ist das auch ohne Gaußsche Formel sehr einfach.

In einer Programmiersprache gibt es viele Befehle, die einem das Leben leicht machen. Um Befehle mehrfach hintereinander auszuführen gibt es so genannte Schleifen[1].

[1] Zu MATLAB und Schleifen siehe auch Abschnitt 4.2

`for`-Schleife

```
for i=1:4
    i
end
```

Das Beispiel zeigt die so genannte `for`- Schleife. Man wählt eine Laufvariable (hier `i`), die den Zahlenbereich von 1 bis 4 durchläuft.
Beim Programmlauf wird in der `for`-Zeile `i` auf 1 gesetzt, dann werden die eingerückten Anweisungen abgearbeitet und dann wieder nach oben gesprungen. Dies wird so oft wiederholt bis `i` den Endwert erreicht hat.
Nun können wird die Zahlen von 1 bis 10 addieren:

```
summe=0;
for  i =1:10
   summe= summe+i;
end

disp(summe)
```

In der ersten Zeile wird die Variable `summe` initialisiert. Dies ist wichtig, da innerhalb der Schleife die Variable auf der rechten Seite auftritt. Vergisst man die Initialisierung, hat `summe` keinen Wert und kann folglich auch nicht verrechnet werden. In der Schleife wird der aktuelle Wert von `i` zum aktuellen Wert von `summe` addiert und das Ergebnis in `summe` gespeichert. Nach Verlassen der Schleife wird das Ergebnis mit `disp` ausgegeben.
Man sieht, dass die Zeile `summe= summe+i;` eingerückt ist. Dies ist in MATLAB zwar nicht zwingend erforderlich, erhöht aber die Lesbarkeit von Programmen enorm. Man rückt alles ein, was innerhalb der Schleife steht.
Als nächstes wollen wird das Programm hübsch machen. Es soll uns nach einer Zahl fragen, bis zu der alle natürlichen Zahlen von 1 an aufaddiert werden sollen, und das Ergebnis ausgeben.

input

```
disp('Dieses Programm berechnet die Summe der Zahlen von 1 bis n')
n=input('Bitte n eingeben:')

summe=0;
```

```
for  i =1:n
   summe= summe+i;
end

disp('Ergebnis:'),disp(summe)
```

Die erste Zeile gibt einen String aus. Ein String ist eine Zeichenkette. Diese wird in Gänsefüßchen gesetzt (ob einfache oder Doppelte ist egal, es muss nur am Anfang und am Ende gleich sein). In der zweiten Zeile wird auch ein String ausgegeben und auf die Eingabe von n über die Tastatur gewartet. Im Ausgabefenster erscheint bis hierhin:

```
Dieses Programm berechnet die Summe der Zahlen von 1 bis n
Bitte n eingeben:
```

Nun kann man z.B. 5 eingeben. Danach wird das Programm automatisch weiter abgearbeitet und es erscheint

```
Ergebnis:
             15
```

Der Befehl `input()` wartet also auf die Eingabe, die mit der Eingabetaste abgeschlossen wird, und schreibt das Ergebnis in die Variable `n`. Die Zahl muss eine ganze Zahl sein. Eine ganze Zahl heißt auch Integer-Wert oder kurz Integer.
Bis jetzt muss man für n eine natürliche Zahl eingeben. Gibt man z.B. 5.5 ein (da Programmiersprachen der englischen Notation folgen, werden Kommazahlen wie beim Taschenrechner mit einem Punkt statt einem Komma eingegeben), so erhält man dasselbe Ergebnis. Die Schleife läuft bis zur größten ganzen Zahl, die kleiner als 5.5 ist. Wie man dies vermeiden, muss man zuvor abfragen, ob die Eingabe eine zulässige Zahl ist.
Dazu müssen wir zuerst überprüfen, ob der ganzzahlige Anteil von `n` gleich `n` selbst ist. Der ganzzahlige Anteil von `n` wird durch `floor(n)` berechnet. Zur Abfrage braucht man eine so genannte `if`-Anweisung:

`if`-Anweisung

```
disp('Dieses Programm berechnet die Summe der Zahlen von 1 bis n')
n=input('Bitte n eingeben:');
```

```
if n==floor(n)
    summe=0;
    for  i =1:n
      summe= summe+i;
    end
    disp('Die Summe beträgt:');
    disp(summe);
end
```

Hinter `if` steht eine Aussage, die wahr oder falsch sein kann. Ist die Aussage wahr, dann wird alles, was zwischen der if-Zeile und end steht, ausgeführt. Die zwei Gleichheitszeichen hintereinander bedeuten, dass abgefragt wird, ob die Werte von n und floor(n) gleich sind. Man kann auch auf ungleich (∼=), größer (>), kleiner(<), größer oder gleich (>=) und kleiner oder gleich (<=) abfragen. Jetzt fehlen noch zwei Sachen. Erstens wollen wird, dass es eine Meldung gibt, wenn ein falsches `n` eingegeben wird, und zweitens müssen wir abfangen, dass ein n kleiner als 1 eingegeben wird. Es müssen also zwei Bedingungen gleichzeitig erfüllt sein, nämlich `n>1` und `n==floor(n)`. Dies erreicht man durch

```
...
if  n==floor(n) & n>1
...
```

Hierbei werden die nachfolgenden Anweisungen nur dann ausgeführt, wenn beide Aussagen gleichzeitig wahr.Das Zeichen `&` kennzeichnet ein logisches UND. Ein logisches ODER wird durch `|` erreicht, ein logisches NOT durch `~`.
Eine Meldung soll erscheinen, wenn mindestens eine der beiden Aussagen falsch ist. Dann ist aber nach der Aussagenlogik auch die gesamte Aussage falsch. Man muss im Programm also eine Meldung ausgeben, wenn die Aussage nach `if` falsch ist . Dies erreicht man, wenn man die `if` Bedingung um eine `else` Anweisung erweitert.

`else` und `&`

```
disp('Dieses Programm berechnet die Summe der Zahlen von 1 bis n')
n=input('Bitte n eingeben:');
```

```
if n==floor(n) & n>1
    summe=0;
    for  i =1:n
      summe= summe+i;
    end
    disp('Die Summe beträgt:');
    disp(summe);
else
   disp('Die Eingabe ist unzulässig!')
end
```

Alles, was nun zwischen `else` und `end` steht, wird ausgeführt, wenn die Aussage nach `if` falsch ist.
Nun wollen wir das Programm ja nicht jedes mal neu starten, wenn eine falsche Eingabe gemacht wurde.

`while`-Schleife

Hierzu gibt es eine weitere Schleifenart, die so genannte `while`-Schleife. Bei einer `while`-Schleife werden alle Anweisungen, die nach der while Zeile eingerückt sind solange ausgeführt, solange die Bedingung zwischen `while` und `:` wahr ist.

```
a=1;
while a<1.3
   a=a+0.1;
   disp(a)
end
```

Die Ausgabe des Programms ist

```
    1.1000
    1.2000
    1.3000
```

Der letzte Schleifendurchlauf findet bei a=1.2 statt. Dann wird aber a noch um 0.1 erhöht und erst dann ausgegeben. Damit können wir das Programm verbessern:

```
disp('Dieses Programm berechnet die Summe der Zahlen von 1 bis n')
ok=0;
while ok==0
      n=input('Bitte n eingeben:');
      if n==floor(n) & n>1
          summe=0;
          for  i =1:n
            summe= summe+i;
          end
          disp('Die Summe beträgt:');
          disp(summe);
          ok=1;
      else
        disp('Die Eingabe ist unzulässig!')
      end
end
```

Wenn eine zulässige Zahl eingegeben wurde, wird `ok` auf 1 gesetzt und die Schleife damit beendet. Anderenfalls wird nach erneuter Eingabe gefragt.
Zu guter Letzt sollten wir das Programm noch kommentieren

4.1 Kommentare

Ganz besonders wichtig in Programmen sind Kommentare. Kommentare dienen dem Programmierer und besonders Leuten, die mit einem fremden Programm arbeiten sollen, dazu, das Programm zu verstehen. Alles innerhalb einer Zeile, was hinter `%` steht, wird als Kommentare gewertet und von MATLAB ignoriert. Jedes Programm sollte mindestens einen Kommentarkopf enthalten, der das Programm beschreibt, den Autor und das Datum der letzten Änderung beinhaltet. Weiter Kommentare sind sinnvoll, wenn Programmteile komplizierter werden.

4.2 MATLAB und Schleifen

Schleifen sollte man in MATLAB nur verwenden, wenn es unbedingt notwendig ist. Meistens findet sich eine elegantere Lösung, die viel schneller abgearbeitet

werden kann. Will man z.B. einen Vektor der Länge 100000 erzeugen, der an der Stelle i den Wert sin(i/pi) enthält, so kann man dies mit

```
for i=1:100000
   x(i)=sin(i/pi);
end
```

erreichen. Dies ist aber sehr langsam, da für jedes i der Speicherplatz für x(i) bereitgestellt werden muss. Schneller ist es, vorab den Speicherplatz mit einer Anweisung zu belegen und erst dann die Werte zuzuweisen:

```
n=100000;
x=zeros(1,n);
for i=1:n
   x(i)=sin(i/pi);
end
```

Noch viel schneller und im Sinne von MATLAB ist folgende Lösung:

```
i=1:100000;
x=sin(i/pi);
```

hierbei ist `i` ein Vektor der die Zahlen von 1 bis 100000 enthält.

4.3 Hilfe

MATLAB hat sehr gute Hilfefunktionen. Unter `Help->Product Help` findet man eigentlich alles was MATLAB kann. Manchmal ist es etwas mühsam, den richtigen Suchbegriff zu finden. Mit etwas Übung geht es aber immer besser! Weiß man nicht genau, was eine Funktion tut, bekommt man z.B. auch mit der Eingabe

```
help sqrt
```

im Kommandofenster Hilfe zur Funktion `sqrt`. Weiß man aber nicht, wie eine Funktion heißt, kann man die Keywordsuche lookfor benutzen, z.B.

```
lookfor root
```

5 Grafik mit MATLAB

5.1 2D-Plots

Eine der Stärken von MATLAB ist die Grafikfähigkeit. Wir wollen nun die Funktion Sinus im Bereich von [–π, π] plotten. Möchte man die Anzahl der gezeichneten Datenpunkte vorgeben, so erzeugt man sich einen Vektor x für die unabhängige Variable z.B. mit `linspace`:

```
x = linspace(-pi,pi,30);
```

Der Vektor x besteht aus 30 Werten im gleichen Abstand, wobei der erste Wert –π und der letzte Wert π ist. Fehlt die Angabe für die Anzahl der Werte, so werden 100 Werte erzeugt. (Vergleiche `x = linspace(-pi,pi,30)` und `x=-pi:0.2:pi`)
Wie in Abschnitt 2.2 beschrieben, erhalten wir mit `sin` die gewünschten Werte.

```
x = linspace(-pi,pi,30);
y=sin(x)
plot(y)
```

Ein Grafikfenster wird geöffnet und der Sinus wird dargestellt. Leider ist die x-Achse aber von 0 bis 30 beschriftet. Dies wird man los, indem man im plot-Befehl die x-Werte explizit angibt:

```
x = linspace(-pi,pi,30);
y=sin(x)
plot(x,y)
```

Hierbei macht MATLAB automatisch die Achse ”hübsch”. Will man die Werte im Bereich von [–π, π] sehen so kann man anschließend

```
axis([-pi pi -1 1])
```

eingeben. genauso kann man nachträglich einen Titel setzen

```
title('Mein erster Plot')
```

Text einfügen

```
text(1,0.5,'Geht sogar {\it kursiv} !')
```

und Achsenbeschriftungen hinzufügen.

```
xlabel('Zeit')
ylabel('Wert')
```

Will man den Graphen in rot haben:

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

oder mit roten Sternen haben:

```
plot(x,y,'r*');
```

oder mit beidem:

```
plot(x,y,'-r*');
```

Die Farbkennungen sind r (red), r (blue), g (green) und k (black). Weitere sind möglich, siehe Hilfe.
Wollen wir den Cosinus im gleichen Graphen haben, so geht dies auf verschiedene Arten. Entweder gleich im Plot-Befehl

```
x = linspace(-pi,pi,30);
y=sin(x)
z=cos(x)
plot(x,y,x,z)
```

wobei jeweils x,y Paare angegeben werden müssen oder man zeichnet ihn nachträglich. Dies geht mit dem Befehl `hold on`:

```
...
plot(x,y)
hold on
plot(x,z)
```

der bewirkt, dass beim nächsten Plot das Fenster nicht vorher gelöscht wird. Hier bei wird die Farbe dann nicht automatisch gewechselt. Man muss diese also explizit angeben. Nutzt man den Befehl `hold all` wird die Linienfarbe beim 2. Plot gewechselt. Die beiden Befehle werden durch `hold off` aufgehoben.
Zusätzlich kann man dann noch eine Legende hinzufügen `legend('Sinus','Cosinus')`

```
legend('Sinus','Cosinus')
```

Der Befehl muss als Argumente jeweils eine Zeichenkette für jeden Graphen erhalten.
Will man eine weiteres Grafikfenster haben, so geht dies mit `figure`

```
...
figure(1)
plot(x,y)
figure(2)
plot(x,z)
```

Will man verschiedene Graphen in einem Plot, so geht dies mit `subplot`, wobei zuerst die Anzahl der Zeilen und Spalten angegeben wird und danach der gerade aktuelle Plot. Folgendes Beispiel erzeugt einen Plot mit zwei Zeilen und einer Spalte

```
figure(1)
subplot(2,1,1)
title('Sinus')
plot(x,y)
subplot(2,1,2)
plot(x,z)
title('Cosinus')
```

MATLAB hat noch viel mehr Möglichkeiten. Eine extensive Nutzung der Hilfe ist hier unbedingt notwendig!!

Übung

Überlege, wie bei folgendem Beispiel die Plots angeordnet sind

```
clf
x = linspace(-pi,pi,30);
y=sin(x)
z=cos(x)
figure(1)
subplot(2,3,5)
plot(x,y)
title('Sinus')
subplot(2,3,1)
```

```
plot(x,z)
title('Cosinus')
```

5.2 3D-Plots

Einige Beispiele für 3D Plots: Mit

```
p=peaks;
```

erhält man eine Matrix (`size(p)`), die man sich mit

```
mit pcolor(p)
```

oder

```
imagesc(p)
```

ansehen kann Einen Skala erhält man mit

```
colorbar
```

Eine 3D-Darstellung erhält man mit

```
mesh(p)
```

oder

```
surf(p)
```

Man kann die Darstellung im Fenster mit der Maus drehen, wenn man im Fenster auf den Dreh-Button drückt.

6 Kleinigkeiten, aber wichtig

6.1 Variablen-Monitoring

Variablen, die während einer MATLAB-Session definiert wurden, behalten ihren Wert bis sie einen neuen zugewiesen bekommen. Welchen Wert bzw. welche Dimension sie haben, kann im Workspace-Fenster überprüft werden. Bei Arrays ist es besonders nützlich, dass ein Doppelklick auf eine Variable den Array-Editor öffnet und das gesamte Array anzeigt.

6.2 Löschen alter Variablen

Will man eine Variable wieder los werden, geht dies mit dem Befehl

```
clear <Variablenname>
```

Mit `clear all` werden alle Variablen im Speicher gelöscht. Sinnvoll ist es, diesen Befehl als ersten im m-File auszuführen, um eventuell noch vorhandene Variablen zu löschen. Mit `clc` kann man das Kommandofenster leeren, mit `close all` alle Grafikfenster schliessen.

Sinnvoll ist es nach dem Programmkopf (Kommentar, der Autor, Programmbeschreibung und Bearbeitungsdatum beinhaltet) die Befehlsfolge

```
clear all
close all
clc
```

auszuführen, um ”altes” los zu werden.

6.3 Unterbrechung der Programmlaufs

Mit der Tastenkombination Strg-C kann ein laufendes Programm abgebrochen werden.

7 Was man sonst so noch braucht

7.1 Ein wenig Statistik

In folgenden sind einige wichtige Statistikbefehle aufgeführt:

Aufruf	Bedeutung
mean(X)	Mittelwert der Elemente Vektors X
median(X)	Median der Elemente Vektors X
std(X)	Standardabweichung (Stichprobe)
std(X,1)	Standardabweichung (Grundgesamtheit)
sum(X)	Summe der Elemente Vektors X
cumsum(X)	kumulierte Summe
diff(X)	Differenz benachbarter Elemente
max(X),min(X)	Minimum, Maximum

7.2 Polynomfit

Zur Funktionsweise siehe Hilfe
Beispiel:

```
clear all
figure(1)
x = linspace(-pi,pi,30);
y=sin(x);
p=polyfit(x,y,3);
plot(x,y,x,polyval(p,x));
```

Übung

Verändere im Beispiel den Grad des Polynoms (3. Argument von polyfit)

7.3 Zufallszahlen

Zufallszahlen erhält man mit dem Befehl rand..

```
rand(1)
```

liefert eine Zufallszahl zwischen 0 und 1. Mit

```
rand(1,10)
```

erhält man einen Vektor mit 10 Zufallszahlen, mit

```
rand(3,3)
```

eine 3x3 Matrix mit Zufallszahlen.
Gleichverteilte Zufallszahlen im Bereich von -a bis a bekommt man dann mit

```
2*a*(rand(1)-0.5)
```

Ganzzahlige Zufallszahlen von 1 -n erhält man mit

```
ceil(rand(1)*n)
```

Benötigt man normalverteilte Zufallszahlen so geht dies mit `randn` Gibt man den Mittelwert m und die Standardabweichung sig vor, so erhält man mit

```
m+sig *randn(1)
```

m,sig-normalverteilte Zufallsvariablen.

7.4 Histogramm

Mit hist kann die Häufigkeitsverteilung eines Vektors ausgegeben werden

```
a=rand(1,100);
hist(a)
```

Standardmäßig werden 10 Größenklassen erzeugt. Die kann mit einem weiteren Argument verändert werden:

```
hist(a,20)
```

Für weiter Beispiele siehe Hilfe!

Übung

Variiere die Vektorlänge und die Anzahl der Klassen.

7.5 Unterprogramme

Nun werden Programme recht schnell unübersichtlich. Daher ist es gut Funktionen zu schreiben, die einen Teil der Berechnungen ausführen und das Ergebnis zurückgeben.
Wollen wir z.B. die Kreisfläche für verschiedene Radien berechnen, so ist es hilfreich eine Funktion zu schreiben, die ganz allgemein die Kreisfläche als Funktion des Radius zurück gibt :Da MATLAB aber keine Funktionen in Skripten

zulässt, müssen wir uns zunächst eine Funktion aus unserem Skript machen. Heißt unser m-File z.B. mykreis so schreiben wir als erstes

```
function mykreis
end
```

in unser m-File. Diese Funktion wird dann von MATLAB aufgerufen, wenn wir `mykreis` im Kommandofenster eingeben. Innerhalb dieser Funktion ist es nun erlaubt, weitere Funktionen zu definieren. Wir bauen uns nun eine Funktion `kreis`, die den Wert `area` an die aufrufende Funktion zurück gibt. Als Argument benötigt die Funktion der Radius `rad`. Die `area` wird dann durch `area= pi*rad^2` berechnet Mit `kreis(2)` erhalten wir dann die Kreisfläche eines Kreises mit Radius 2.

```
function mykreis
clear all
function area=kreis(rad)
area=pi*rad^2;
end
kreis(2)
end
```

Gibt man nun im Kommandofenster `mykreis` ein, so erhält man die Ausgabe

```
>> mykreis
ans =
   12.5664
```

7.6 In eine Datei schreiben/aus einer Datei lesen

Will man Daten in eine Datei schreiben, so muss man zuerst die Datei zum Schreiben öffnen. Dies geht mit

```
fid = fopen('out.txt', 'w');
```

Hier wird nun die Datei out.txt zum Schreiben (w) geöffnet. Wenn Sie existiert, wird sie überschrieben, wenn nicht, wird sie angelegt. Zurück kommt ein Identifier (`fid`), den man nun ansprechen kann.
Nachdem alles weggeschrieben ist, muss die Datei mit

```
fclose(fid)
```

geschlossen werden.
Will man nun also x-Werte und zugehörige y-Werte in eine Datei schreiben, so geht dies wie folgt:

```
x = 0:.1:1;
y = [x; exp(x)];
fid = fopen('exp.txt', 'w');
fprintf(fid, '%f %f\n', y);
fclose(fid)
```

Mit der Anweisung `fprintf` werden die Werte in die Datei geschrieben. Hierzu muss man den Identifier , das Ausgabeformat und die Daten, die geschrieben werden sollen, angeben. Das Ausgabeformat beinhaltet für x und y eine Angabe über das Format hier `%f`, was bedeutet, dass es Fließkommazahlen werden sollen.
Diese Datei kann man nun auch wieder auslesen:

```
a=load('exp.txt')
```

Will man in anderen Formaten ein- bzw. auslesen, findet man dies in der Hilfe!

Übung

Was macht das `\n` in der Formatanweisung?

7.7 Differentialgleichungen

Das Lösen von Differentialgleichungen geht mit `ode23` oder `ode45`. Siehe Hilfe!

8 Fazit

Damit haben wir einige Grundlagen der Programmierung zusammen. In anderen Programmiersprachen läuft es ganz ähnlich, auch wenn die Befehle mei-

stens etwas anders heißen. Wichtig dabei ist, dass einige Programmiersprachen, wie MATLAB, zwischen **Groß- und Kleinschreibung unterscheiden**, andere nicht. Auch lassen viele Sprachen **keine Sonderzeichen wie ä oder ß** zu. Matrizen und Vektoren sind eine Spezialität von MATLAB. Man findet sie nur in einigen Programmiersprachen, wie z.B. Python, wieder. Meistens müssen dazu spezielle Zusatzpakte geladen werden.
Die Grafikfähigkeit der verschiedenen Sprachen ist sehr unterschiedlich. Daher ist es wichtig zu verstehen, wie man Ergebnisse in Dateien schreibt. Man ist dann unabhängig von der jeweiligen Sprache und kann die Ergebnisse anderweitig weiterverarbeiten.
Diese Einführung ersetzt keinen Programmierkurs reicht aber für die Übungen aus.