Praktikum "Mathematik am Computer" HS 2020
Prof. H. Harbrecht
R. Brügger, M. Fallahpour, E. Paemurru, C. Santos Universität Basel

---

MATLAB Tutorial

(von Clemens Staub, Michaela Mehlin, HS 2010, überarbeitet von Dennis Tröndle, HS 2017)

MATLAB ist eine Software zur Lösung mathematischer Probleme und zur grafischen Darstellung der Ergebnisse. Matlab ist primär für numerische Berechnungen mithilfe von Matrizen ausgelegt, woher sich auch der Name ableitet: MATrix LABoratory.
Matlab dient im Gegensatz zu Computeralgebrasystemen wie Maple nicht der symbolischen, sondern hauptsächlich der numerischen (zahlenmäßigen) Lösung von Problemen. Programmiert wird unter Matlab in einer proprietären Programmiersprache, die auf dem jeweiligen Computer interpretiert wird. Grosse Programme können aus Skripts oder Funktionen zusammengefügt werden. Durch die vereinfachte, mathematisch orientierte Syntax der Matlab-Skriptsprache und die umfangreichen Funktionsbibliotheken beispielsweise für Statistik, Signal- und Bildverarbeitung ist die Erstellung entsprechender Programme wesentlich einfacher möglich als z. B. unter C.
Matlab wird am Mathematischen Institut der Uni Basel vorwiegend in den Veranstaltungen zur Numerik auf Bachelor- und Masterstufe eingesetzt.
In der Institutsbibliothek und in der UB sind Handbücher zu Matlab zu finden. Weitere Informationen sind auf der Webseite des Herstellers Mathworks zu finden:
`http://www.mathworks.ch/`

1 Operatoren und Werte

Im `Command Window` nimmt Matlab Befehle direkt entgegen. Nach der Eingabeaufforderung, gekennzeichnet durch `>>`, können Befehle eingegeben und durch Drücken von Enter ausgeführt werden. Mit den Pfeiltasten können früher eingegebene Befehle wieder aufgerufen werden. Mit `clc` kann das *Command Window* geleert werden. Versuche einige einfache Berechnungen auszuführen und achte auf die Präzedenz-Regeln der Operatoren. Mit Klammern kannst du diese beeinflussen.

```
>> 1 + 2 - 3 * 4 / 5 ^ 6

ans =

    2.9992

>> 1 + (2 - 3) * 4 / 5 ^ 6

ans =

    0.9997
```

Matlab benutzt die Dezimaldarstellung (mit . als Trennzeichen) für Zahlen und versteht `e` als Zehner-Potenz-Faktor, sowie `i` oder `j` als imaginäre Zahl. Man sollte `e` also nicht für `exp(1)` halten und `i` oder `j` nur als imaginäre komplexe Zahl ($i = \sqrt{-1}$) benutzen und nicht - auch wenn es möglich ist - als Variablennamen.

```
>> 5e-2

ans =

    0.0500

>> i + 1

ans =

   1.0000 + 1.0000i
```

2 Variablen

Das `ans` in der Ausgabe ist eine Variable, in welcher Matlab das letzte Ergebnis speichert. Mit `=` können wir selbst Variablen definieren (nicht mit `:=` wie in Maple). Mit `who` bzw. `whos` sowie im Fenster `Workspace` können wir die aktuell definierten Variablen betrachten.

```
>> x = 1

x =

     1

>> y = 1

y =

     1

>> z = x + y

z =

     2
```

Die Ausgabe kann unterdrückt werden, indem der Befehl mit Semikolon (`;`) abgeschlossen wird. Das sollte man sich angewöhnen und es nur weglassen, wenn man die Ausgabe tatsächlich sehen will.
Matlab kennt verschiedene Typen von Variablen und wählt meist automatisch den geeigneten Typ aus. Mit dem Befehl `whos` kann man den Typ der aktuell definierten Variablen sehen. Dabei können Variablen vom Typ `logical` die Werte `true` und `false` bzw. `1` und `0` aufnehmen, der Typ `int` ganze Zahlen, `double` reelle Zahlen bis zu einer Genauigkeit von grob 16 Dezimalstellen (mit den Befehlen `format long` bzw. `format short` kann man sich 15 oder nur 5 Nachkommastellen anzeigen lassen) und `char` Buchstaben, um nur die wichtigsten zu nennen.

```
>> x = int8(1); y = 1 - i; z = 'Hallo'; a = true; whos
  Name      Size            Bytes  Class      Attributes

  a         1x1                 1  logical
```

```
  x         1x1                 1  int8
  y         1x1                16  double     complex
  z         1x5                10  char
```

Mit `clear x` kann eine bestimmte Variable `x` gelöscht werden. Mit `clear all` der gesamte Workspace. Alle aktuellen Variablen (der sog. *Workspace*) können mit dem Befehl `save MeineDaten.mat` in eine Datei gespeichert werden und mit `load MeineDaten` wieder geladen werden. Achtung, Variablen mit demselben Namen werden dadurch überschrieben! Die Datei *MeineDaten.mat* wird mit dem Befehl oben im aktuellen Verzeichnis gespeichert. Mit dem Befehl `pwd` (für *print working directory*) kann dieses bestimmt und mit `cd` (für *change directory*) geändert werden. (`cd ..` wechselt in das nächsthöhere Verzeichnis). Mit `ls` kann man sich vergewissern, welche Dateien zur Verfügung stehen.

Um den Verlauf der eingegebenen Befehle mitzuschneiden gibt es den Befehl
`diary MeinTagebuch.txt`. Mit `diary off` beendet man die Aufzeichnung. Dies kann praktisch sein, um die Lösungen der ersten Aufgaben zu speichern.

3 Matrizen

Matlab hat seinen Namen - wie bereits erwähnt - deswegen, weil fast alle Variablen in Matlab Matrizen bzw. Arrays (Tabellen oder Felder) sind. Eine eindimensionale Matrix nennen wir gewöhnlich einen *Vektor*. Eine einzelne Zahl kann man sich als eindimensionale Matrix mit einem einzigen Eintrag denken.
Matrizen kann man auf unterschiedliche Weisen erzeugen. Am einfachsten geht das mit eckigen Klammern, wobei man durch ein Komma oder ein Leerzeichen eine neue Spalte und durch ein Semikolon eine neue Zeile erzeugt.

```
>> A = [17, 62, -38; 72, -9, 25]

A =

     17    62   -38
     72    -9    25
```

Die Einträge einer Matrix bzw. eines Vektors können auf mehrere Weisen angesprochen werden. Einerseits können mit natürlichen Zahlen, in runden Klammern direkt auf die Variable folgend, erst die Zeile und dann die Spalte angegeben werden. So ist `A(2,1)` der Wert 72. Andererseits kann mit einer einzigen Zahl ein bestimmter Eintrag bezeichnet werden, da Matlab die Einträge entlang der Spalten durchnummeriert. Zum Beispiel ist `A(5)` gleich -38. Eine weitere Methode auf bestimmte Einträge einer Matrix zuzugreifen ist, dass man eine ihr eine gleich grosse Matrix vom Typ (`logical`) mit `1` an den gewünschten Stellen und `0` überall sonst übergibt. Dazu später mehr.
Eckige Klammern können auch dazu benutzt werden mehrere Matrizen oder Vektoren zu grösseren Matrizen zu kombinieren:

```
>> A = [1 2]; B = [3 4]; [A; B]

ans =

     1     2
     3     4
```

In Matlab kann man mithilfe von `:` einfach einen Vektor mit auf- oder absteigenden Einträgen erstellen:

```
>> a = 1:5

a =

     1     2     3     4     5
```

Mehrere Doppelpunkte geben den Abstand an, den die Einträge eines Vektors haben sollen:

```
>> a = 0:0.25:1

a =

     0    0.2500    0.5000    0.7500    1.0000

>> a = 10:-2:4

a =

     10     8     6     4
```

Diese Schreibweise erzeugt immer Zeilenvektoren, die entsprechenden Spaltenvektoren erhält man durch Transposition mit `'`:

```
>> a = (1:5)'

a =

     1
     2
     3
     4
     5
```

Im Unterschied zu einigen anderen Programmiersprachen steuert man den ersten Eintrag eines Vektors `a` mit `a(1)` und nicht mit `a(0)` an.

Natürlich kann man mit Matrizen auch rechnen. Es muss dabei immer darauf geachtet werden, wie eine Operation auf eine Matrix wirkt. Die meisten Operatoren (also Funktionen wie `exp`, `sin`, `cos` oder Oprationen wie `+` und `-`) wirken elementweise, d.h. die Operation wird für jeden Eintrag separat ausgeführt. `sin(A)` ist dasselbe wie `sin(A(1,1))`, `sin(A(1,2))` usw. Aufpassen muss man bei den `*`, `/`, und `^`. Diese berechnen die zugehörigen Matrixoperationen, d.h. `A*B` ist die aus linearen Algebra bekannte Matrixmultiplikation, `A/B` die Matrixdivision und `A^2` die Matrixpotenz. Die entsprechenden elementweisen Operatoren sind `.*`, `./`, und `.^`(beachte den jeweils vorangehenden Punkt). `A * B` und `A .* B` (und `B * A` !) sind also nicht dasselbe, wenn `A` und `B` Matrizen sind.

```
>> A = [4 6; 8 2]; B = [-0.05 0.15; 0.2 -0.10]; A * B, A .* B

ans =

    1.0000   -0.0000
         0    1.0000


ans =

   -0.2000    0.9000
    1.6000   -0.2000
```

Das heisst u.a. also auch, dass `A * B` nur definiert ist, wenn `A` gleich viele Spalten wie `B` Zeilen hat. Das ist verletzt, wenn die Fehlermeldung `Inner matrix dimensions must agree.` auftritt. `x' * y` berechnet das Skalarprodukt ($x^t y$) von zwei Spaltenvektoren `x` und `y`.

```
>> x = (0:4)'; y = (1:5)';
>> x'*y

ans =

    40
```

Der Ausdruck `x .* y` (also zwei Spaltenvektoren) multipliziert den i-ten Eintrag von `x` mit dem i-ten Eintrag von `y`, ergibt also einen Vektor mit dem Eintrag $x_i \cdot y_i$ an der Stelle i. Stimmen hier nun die Vektoren in der Grösse nicht überein, meldet Matlab den Fehler `Matrix dimensions must agree.`

```
>> x.*y

ans =

    0
    2
    6
    12
    20
```

In neueren Matlab-Versionen mutlipliziert `x .* y'` (ein Spaltenvektor, der mit einem Zeilenvektor eintragsweise multipliziert wird) jeden Eintrag von `x` mit jedem Eintrag von `y` und gibt eine Matrix aus, die an der Stelle (i, j) den Eintrag $x_i \cdot y_j$ hat. Dies heisst auch implizite Erweiterung. Auf die gleiche Art und Weise kann man eintragsweise eine Matrix mit einem Vektor mulitplizieren, wie das Beispiel unten zeigt.

```
>> x.*y'

ans =

    0     0     0     0     0
    1     2     3     4     5
```

```
    2     4     6     8    10
    3     6     9    12    15
    4     8    12    16    20
```

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

ans =

     5    12
    15    24

>> A.*b'

ans =

     5    10
    18    24
```

Wenn man die Multiplikationszeichen nicht als implizite Erweiterung interpretiert, scheint es, als ob man Rechnungen durchführen kann, die gemäss der linearen Algebra verboten sind.
Die Grösse einer Matrix kann mit dem Befehl `size()` (oder `length()` für einen Vektor) bestimmt werden.

4 Funktionen

Matlab bietet eine Vielzahl bereits eingebauter Funktionen. Diese reichen von den elementaren Funktionen (z.B. sin, log, exp, ...) über statistische Tests bis hin zu komplexen numerischen Verfahren. Mit dem Befehl `doc` gelangt man zur ausführlichen Dokumentation, in der man nach geeigneten Funktionen suchen kann. Zu einer spezifischen Funktion bekommt man mit dem Befehl `help` *Funktionsname* weitere Hilfe.
Funktionen haben jeweils Eingabe- und Ausgabewerte. Mehrere davon werden durch Kommas (`,`) getrennt. Manchmal sprechen wir von den Eingabewerten als *Argumente* und von den Ausgabewerten als dem *Funktionswerte*. Erstere folgen der Funktion in runden Klammern. Letztere kann man sich ausgeben lassen oder in eine Variable speichern, indem man hinter die Funktion die Variable schreibt:

```
>> y = cos(3.14)

y =

   -1.0000
```

Funktionen in Matlab können unterschiedlich darauf reagieren, wieviele Argumente oder Ausgabewerte man eingibt bzw. verlangt. Wollen wir beispielsweise die Maxima der Matrix `A = [2 11 5; 7 3 13]` bestimmen, so gibt uns `max(A,[],1)` (oder auch einfach `max(A)`) das Maximum in jeder Spalte, also `[7 11 13]`, und `max(A,[],2)` das Maximum in jeder Zeile, also `[11 13]'`. Verlangen wir bloss einen Ausgabewert, indem wir `x = max(A)` schreiben, erhalten wir wie erwartet den obigen Vektor. Geben wir noch einen zweiten Ausgabewert hinzu,

`[x,y] = max(A)`, so liefert uns `max` noch einen zweiten Vektor, dessen Einträge die Indizes der maximalen Einträge in den Spalten sind. Diese verschiedenen Verwendungsweisen dokumentiert die Hilfe zu einem Befehl (bspw. `help max` oder `doc max`). Es lohnt sich, sich mit dem dort und in allen Hilfeseiten verwendeten Schema vertraut zu machen.

Neben `help` und `doc` ist auch der Befehl `lookfor` nützlich um den Namen einer Funktion zu finden. Matlab bietet auch die Vervollständigung von Befehlen mit der Tabulator-Taste an. `cos` + Tabulator bietet so eine Auswahl aus `cos`, `cosd` (für Ausgaben in Grad statt Radiant) und `cosh` (für den Cosinus Hyperbolicus).

5 Plotten

Zweidimensionale Zeichnungen (sogenannte *Plots*) erstellt man in Matlab, indem man einen Vektor `x` definiert, der in der gewünschte Auflösung oder Feinheit der Abszissenachse (horizontal) die x-Werte enthält, z.B. `x = -3:0.01:3`, und einen Vektor `y`, der entsprechend viele Funktionswerte enthält, z.B. `y = x.^2;`
`plot(x,y)` plottet dann die Parabel in ein neues Fenster. Ein solches Bild nennt Matlab *figure* und überschreibt diese jeweils mit dem neusten Plot. Mit dem Befehl `figure(n)` (mit `n` $\in \mathbb{N}$) kann man im Voraus angeben welchen Plot man bearbeiten und/oder überschreiben möchte.
Plottet man eine weitere Funktion, wird ein bereits existierender Plot überschrieben. Mit `hold on` kann man das Überschreiben verhindern und stattdessen mehrere Plots im gleichen Fenster kombinieren. `hold off` hebt dies wieder auf. Ein Beispiel:

```
>> clear all
>> x = -3:0.01:3;
>> figure(1)
>> plot(x,x.^2);
>> figure(2)
>> plot(x,-x.^2+10);
>> hold on
>> plot(x,x)
>> hold off
>> figure(1)
>> plot(x,cos(x),x,sin(x),'--')
>> hold on
>> plot(x,0,'r-');
>> legend('Cosinus','Sinus')
```

Einen Titel fügt man mit `title('...')`, die x- bzw. y-Achsenbeschriftung mit `xlabel('...')` bzw. `ylabel('...')`
Mehr Informationen gibt wie immer `help plot` und etwas anschaulicher `demo matlab graphics`

6 Logische Ausdrücke

Logische Ausdrücke sind Aussagen wie "A ist grösser als 0" oder "B ist nicht gleich 5", also Ausdrücke, die entweder wahr oder falsch sind. Wie alles in Matlab werden auch logische Ausdrücke in einer Matrix oder einem Vektor dargestellt. Diese sind dann vom Typ `logical`, deren Einträge `1` oder `0` bzw. `true` oder `false` sind. Dazu verbindet man Zahlen und Vektoren oder Matrizen mit `==` (gleich), `~=` (nicht gleich (äussere Negation)), `<`, `>`, `<=`, `>=`.

Vergleichen wir damit zwei Matrizen oder Vektoren A und B, müssen sie selbstverständlich gleich gross sein, da so die jeweils korrespondierenden Einträge $A(i, j)$ und $B(i, j)$ miteinander verglichen werden. Das Resultat ist dann eine gleich grosse Matrix mit `1` oder `0` als Einträgen. Es können aber auch Matrizen mit einer einzigen Zahl verglichen werden. Matlab vergleicht dann jeden Eintrag der Matrix einzeln mit dieser einen Zahl.

```
>> x = 1:10; x >= 5

ans =

     0     0     0     0     1     1     1     1     1     1
```

Matlab kennt auch viele eingebaute Funktionen, welche logische Matrizen liefern. Diese nennt man auch *Prädikate*. Möchte man beispielsweise die Primzahlen zwischen 1 und 10 wissen, so kann man schreiben:

```
>> isprime(x)

ans =

     0     1     1     0     1     0     1     0     0     0
```

Logische Ausdrücke lassen sich mit sog. Junktoren zu komplexen Aussagen verbinden, diese sind: `&` für *und*, `|` für *oder* und `xor` für *entweder-oder*. Eine logische Matrix lässt sich mit `~` (nicht) verneinen, d.h. aus *wahr* wird *falsch* und umgekehrt.
Weitere Befehle kann man in der Dokumentation finden (z.B. Quantoren und *short-circuit* Operatoren).
Logische Ausdrücke sind einerseits nützlich für die `if`-Ausdrücke und `while`-Schleifen, die wir im Abschnitt 8.1 behandeln. Andererseits kann man in Matlab *logische Indizes* benutzen, um mit Vektoren und Matrizen zu arbeiten, d.h. wir können einen Vektor oder eine Matrix A mit einem logischen Vektor oder einer logischen Matrix genau an den Stellen bearbeiten, an denen ein logischer Ausdruck wahr ist. Möchten wir z.B. in einem Vektor alle negativen Zahlen quadrieren, können wir das wie folgt tun:

```
>> x = -5:5;
>> my_indices = x < 0

my_indices =

1x11 logical array

    1   1   1   1   1   0   0   0   0   0   0

>> x(my_indices) = x(my_indices).^2
x =

    25    16     9     4     1     0     1     2     3     4     5
```

7 m-Files

Bis jetzt haben wir Matlab als interaktive Rechenumgebung verwendet. Natürlich wird man grössere Projekte nicht einfach im *Command Window* ausführen. Eine Folge von Befehlen kann

man in sog. *m-Files* abspeichern. Das sind gewöhnliche Textdateien mit der Dateiendung "`.m`". Durch klicken auf *New* → *Script* kann man eine neue m-File erstellen.
Erstellt man beispielsweise eine m-File mit dem Namen `MeineMatrixA.m` und dem Inhalt

```
A = [35,1,6,26,19,24;
    3,32,7,21,23,25;
    31,9,2,22,27,20;
    8,28,33,17,10,15;
    30,5,34,12,14,16;
    4,36,29,13,18,11;];
```

so lässt sich die Matrix A leicht verändern und kann einfach wiederverwendet werden. Wird die Datei `MeineMatrixA.m` nun im aktuellen Verzeichnis abgespeichert, kann die Matrix A mit dem Befehl `MeineMatrixA` in den *Workspace* geladen werden. Bis hierhin unterscheidet sich dies nicht vom Umgang mit `.mat`-Dateien. Erstelle nun folgende m-File und speichere sie im aktuellen Verzeichnis ab.

```
% Was ist die Quadratwurzel von 2?

format long

a = 2

a = (a + 2/a)/2

a = (a + 2/a)/2

a = (a + 2/a)/2
```

Grundsätzlich führt Matlab ein m-File einfach Zeile für Zeile aus, so als würden die Befehle im *Command Window* eingegeben. Dabei sind alle Variablen im *Workspace* zugänglich und dort gespeichert, wie zum Beispiel die Variable a in diesem Skript. Es wird nun wichtig, jede Zeile, die man nicht explizit im *Command Window* ausgegeben haben will, mit einem Semikolon (;) zu beenden, damit man dort den Überblick behält.
Eine m-File kann, wie bereits erwähnt, mit ihrem Namen (ohne `.m`) im *Command Window* aufgerufen werden oder, wenn die Datei im *Editor* von Matlab geöffnet, ist mit der Taste F5 beziehungsweise dem Button *Run*.
Man darf als Dateinamen für m-Files, wie auch bei den Variablen, nicht den Namen einer bereits existierenden Funktion nehmen, sonst wird diese überschrieben. Ebenso darf in einer m-File eine Variable nicht denselben Namen tragen wie die m-File selbst.
Zur besseren Übersicht können in ein m-File Kommentare geschrieben werden. Eine solche Zeile muss mit einem Prozentzeichen (%) beginnen, dann wird sie von Matlab ignoriert. Kommentare sind eine sehr wichtige Sache, will man sicher stellen, dass auch jemand anderes oder man selbst nach einer gewissen Zeit noch versteht, was man programmiert hat.

8 Ablaufkontrolle: if, for, while

Um den Ablauf eines Programms zu bestimmen, kennt Matlab, wie viele andere Programmiersprachen auch, Befehle zur Steuerung: `if`-, `for`- und `while`-Konstrukte.

Diese Sprachelemente kann man auch direkt im *Command Window* eingeben, meistens verwendet man sie aber in m-Files und Matlab-Funktionen.
Ihnen ist allen gemeinsam, dass sie kontrollieren ob oder wie oft ein Abschnitt des Skripts ausgeführt werden soll. Der betreffende Abschnitt wird von dem entsprechenden Schlüsselwort zusammen mit einem `end` umrahmt.
Es kann helfen, die Funktionsweise in einen deutschen Satz zu übersetzen, damit der Ablauf des Programms an dieser Stelle klar wird.

8.1 if-then-else

`if` dient dazu, einen Abschnitt nur unter gewissen Bedingungen auszuführen. Also etwa so: "Wenn dies und das der Fall ist, so tue folgendes".

```
if (x > y)
    disp('x ist groesser als y');
end
```

Ein if-Ausdruck kann auch auf einer Zeile geschrieben werden. Allerdings müssen Ausdrücke, die nicht durch ein Semikolon abgetrennt sind, durch ein Komma abgetrennt werden, z.B.
`if x > y, y = x; end`
Anweisungen, die, wenn die Bedingung falsch ist, *anstelle* der nach `if` aufgeführten Befehle ausgeführt werden sollen, können nach `else` angegeben werden. Unser obiger Satz wird dazu ergänzt um "... und sonst tue folgendes".

```
if (x>y)
    disp('x ist groesser als y');
else
    disp('y ist groesser oder gleich x');
end
```

Wir können auch noch eine weitere Bedingung einbauen mit `elseif`, die wir mit "wenn aber"wieder geben können. Bspw. also "Wenn x grösser ist als y dann tue folgendes ..., wenn aber x gleich y ist, dann tue folgendes ..., sonst tue folgendes ...."

```
if (x>y)
    disp('x ist groesser als y');
elseif (x=y)
    disp('x ist gleich y');
else
    disp('x ist kleiner als y');
end
```

8.2 for

Mit einer `for`-Schleife kann ein Abschnitt beliebig oft wiederholt werden. In jedem Schritt der `for`-Schleife nimmt eine Variable, die sogenannte *Laufvariable*, dabei einen vorher festgelegten Wert an. Übersetzt man die `for`-Schleife in einen deutschen Satz, wäre dies "Für jedes k aus [...] tue dies ..." oder "Tue das Folgende mit jedem k aus [...]".
Als Beispiel gibt der folgende Code in jedem Schritt der `for`-Schleife den Wert der Laufvariable k im *Command Window* aus:

```
for k = 1:5
    disp('Jetzt hat k den Wert:');
    disp(k);
end
```

Beachte, dass k auch Werte aus einem beliebigen Vektor annehmen kann:

```
M = [5,-10,7,3,-1.1];
for k = M
    disp('Jetzt hat k den Wert:');
    disp(k);
end
```

8.3 Effizienz

Die folgende Bemerkung ist so wichtig, dass sie ihren eigenen Abschnitt verdient. `for`-Schleifen sind sehr nützlich und wahrscheinlich das am häufigsten eingesetzte Sprachelement. Da Matlab aber dahingehend optimiert wurde, Matrizen zu bearbeiten, sollte man sich immer überlegen, ob es nicht eine Funktion gibt, die direkt auf den Elementen operiert, anstatt eine `for`-Schleife zu benutzen. D.h., möchte man die Quadrate der ersten fünf Primzahlen berechnen, könnte man das folgendermassen tun:

```
p=[2,3,5,7,11];
for k = 1:5
    p(k)=p(k)^2;
end
```

Wir haben aber den Operator `.^` kennengelernt, dieselbe Operation kann man also auch so ausführen:

```
p=[2,3,5,7,11];
p=p.^2;
```

Dies spart nicht nur Schreibarbeit, sondern vor allem auch Rechenzeit. Bei kleinen Aufgaben ist das natürlich nicht spürbar. Berechnet man jedoch Datensätze mit mehreren tausend Einträgen, so benötigt die `for`-Schleife wesentlich mehr Zeit. Betrachte folgendes Beispiel (`tic` und `toc` messen die Zeit, die in den Codezeilen dazwischen benötigt wurde):

```
x = 1:10000;
tic
for k = x
    y(k) = k^2;
end
toc
tic
    z = x.^2;
toc
```

8.4 while

Der deutsche Merksatz für die `while`-Schleife wäre: "Solange gilt, dass ..., wiederhole folgendes ...". Ähnlich wie bei der `for`-Schleife kann ein Abschnitt damit beliebig oft wiederholt werden, allerdings nur, solange eine bestimmte Bedingng erfüllt ist. Dazu wird ein logischer Ausdruck immer *vor* dem nächsten Durchlauf evaluiert. Sobald dieser den Wahrheitswert `0` bzw. `false` annimmt, wird die Schleife abgebrochen. Man muss hierbei natürlich genau darauf achten, dass sich dieser Ausdruck in dem wiederholten Abschnitt irgendwie verändert, oder dass man die Schleife ggf. mit dem Befehl `break` unterbricht, z.B. in einem `if`-Ausdruck, sonst läuft die `while`-Schleife immer weiter. Man kann mit der `while`-Schleife zum Beispiel $\sqrt{2}$ mit einer Abweichung von 0.001 annähern:

```
>> a = 2;
while abs(a^2-2) > 0.001
    a = (a + 2/a)/2;
end
disp(a);
```

9 Funktionen

Matlab erlaubt es nicht nur, mFiles zu schreiben, welche die bereits eingebauten Funktionen aufrufen, sondern auch das Schreiben von eigenen Funktionen. M-Files, wie wir sie bis jetzt kennen, sind blosse Skripts, die kein Eingabeargumente akzeptieren und keine Ausgabewerte übergeben, sondern bearbeiten einfach den Workspace. Funktionen hingegen können Eingabewerte (Input) entgegennehmen, sie verarbeiten und Ausgabewerte (Output) zurückgeben. Ein einfaches Beispiel einer Funktion ist eine m-File `my_square.m` mit folgendem Inhalt:

```
function y = my_square(x)
    y = x.^2;
end
```

Man kann diese Funktion verwenden wie eine eingebaute Funktion:

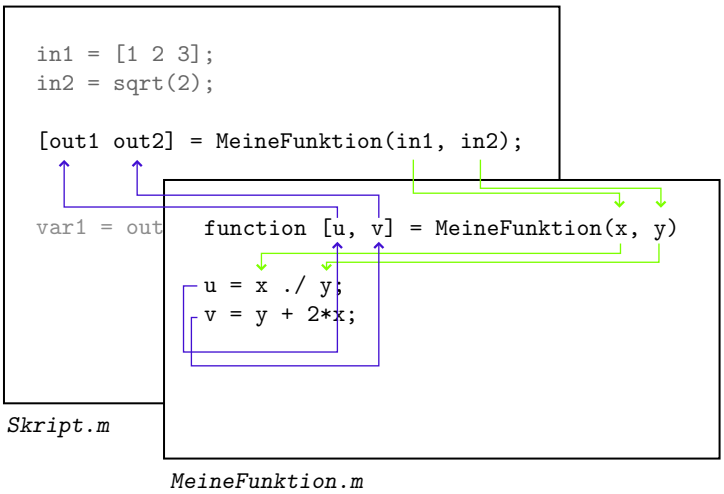
```
y = my_square(5)

y =

    25
```

Probiere dies einmal aus. Es ist unbedingt erforderlich, dass das m-File denselben Namen wie die Funktion hat. Weiter darf der Name natürlich nicht schon für eine andere Funktion in Matlab stehen und auch nicht für eine Variable oder eine `.mat`-Datei verwendet werden.

Ein Funktions-m-File muss immer mit dem Schlüsselwort `function` beginnen, gefolgt von den Ausgabevariablen, dem Funktionsnamen und den Eingabevariablen in der gewünschten Reihenfolge: `output = funktions_name(input)`. Eine Funktion kann auch mehrere Inputs und Outputs haben: `[x, y, z] = fun(a,b,c)`.

Funktionen sind von gewöhnlichen m-Files wesentlich darin unterschieden, dass sie einen eigenen geschützten "Lebensraum" (englisch *Scope*) haben. D.h. in der Funktion `[u,v] = MeineFunktion(x,y)` stehen die Variablen x und y zur Verfügung und werden die Variablen u und v erstellt, sie "leben" aber nicht im *Workspace*, sondern nur innerhalb der Funktion,

wenn diese aufgerufen wird. D.h. wenn ich die Funktion aufrufe, übergebe ich ihr einen Wert oder eine Variable mit einem Wert. Dieser wird durch die Funktion aber nicht verändert und kann auch einen beliebigen Namen haben: `MeineFunktion(z, t)`, `MeineFunktion(2, 3)`, etc. Speichere ich den Output der Funktion in eine weitere Variable, so muss auch diese nicht den in der Funktion angegebenen Namen tragen:
`[out1, out2] = MeineFunktion(in1, in2)`
`out2` ist gewissermassen eine Kopie von v am Ende der Funktion `MeineFunktion`. Mache dir dies mit einem eigenen Beispiel und der folgenden Abbildung klar.



Wer noch raffinierter mit Ein- und Ausgabevariablen umgehen möchte, kann die Dokumentation nach den Befehlen `varargin`, `varargout`, `nargin`, `nargout` durchsuchen. Weiter kennt Matlab auch noch anonyme und Sub-Funktionen. Ebenso lohnt es sich, sich mit den Datentypen `cell` und `struct` vertraut zu machen. Dies würde jedoch den Rahmen dieser Einführung sprengen.