Programmieren in der Verhaltens- und Neurophysiologie

Anfängerkurs für angewandtes Programmieren mit MATLAB

Natalja Gavrilov
natalja.gavrilov@student.uni-tuebingen.de

Maria Moskaleva
maria.moskaleva@gmail.com

Prof. Dr. Andreas Nieder
andreas.nieder@uni-tuebingen.de

LS Tierphysiologie
Institut für Neurobiologie
Universität Tübingen
Auf der Morgenstelle 28
72076 Tübingen

WS 2013/14

Inhaltsverzeichnis

0 Organisatorisches **5**
0.1 Wichtige Termine . . . 5
0.2 Übungsblätter . . . 5
0.3 Klausur . . . 5
0.4 Benotung . . . 5

1 Einführung **5**
1.1 Wozu Programmieren? . . . 5
1.2 Warum MATLAB? . . . 6
1.3 MATLAB zu Hause . . . 6
1.4 Weiterführende Informationen zu MATLAB . . . 6

2 MATLAB Benutzeroberfläche **7**

3 Variablen **8**
3.1 Definition der Variable . . . 8
3.2 Berechnungen mit Variablen . . . 10
3.3 Löschen von Variablen . . . 11

4 MATLAB-Programme **11**

5 Input und Output **13**
5.1 Semikolon in MATLAB . . . 13
5.2 Output . . . 14
5.3 Input . . . 14

6 Datentypen **15**

7 Funktionen **17**

8 Das Hilfe-System von MATLAB **18**

9 Fehlerfreies Programmieren **19**

10 Vektor **24**
10.1 Konstruktion von Vektoren . . . 24
10.2 Vordefinierte Vektoren . . . 27
10.3 Operationen mit Vektoren . . . 28

11 Logisches indizieren **30**
11.1 Formulierung von Bedingungen mit *Vergleichsoperatoren* . . . 30
11.2 Logisches indizieren . . . 31
11.3 Prüfung von logischen Ausdrücken mit *logischen Operatoren* . . . 32

12 Matrix **33**
12.1 Größe einer Matrix . . . 33
12.2 Konstruktion einer Matrix . . . 33

12.3 Indizieren einer Matrix . . . 34
12.4 Löschen und Verbinden von Matrizen . . . 36
12.5 Operationen mit Matrizen . . . 37
12.6 Einlesen von Daten . . . 38

13 Kontrollstrukturen beim Programmieren **39**
13.1 Verzweigungen . . . 39
13.2 Programmschleifen . . . 44
13.2.1 For-Schleifen . . . 44
13.2.2 Häufige Fehler bei Schleifen und Matrizen . . . 49
13.2.3 While-Schleifen . . . 50
13.2.4 Ablaufunterbrechung der Schleifen . . . 51
13.3 Geschachtelte Schleifen . . . 53

14 Plot **55**
14.1 Plotting . . . 55
14.2 Labelling . . . 58
14.3 Veränderung der Achsen und mehrere Abbildungen . . . 59
14.4 Plot-Tools . . . 60

0 Organisatorisches

0.1 Wichtige Termine

Der Kurs findet vom 18.11.2013 bis 26.11.2013 ganztägig statt. Die Klausur ist am 6.12.2013.

0.2 Übungsblätter

Jeder Kursteilnehmer bearbeitet 5 Übungsblätter. Jedes Übungsblatt wird einzeln bewertet. Die Übungsblattnote ist die mittlere Note der 5 Übungsblätter.

0.3 Klausur

Die Klausur findet am 6.12.2013 um 10 Uhr im Kursraum statt. Bearbeitungszeit ist 120 min. Die Aufgaben werden am PC mit MATLAB bearbeitet. Es sind **keine** Hilfsmittel (Skript, eigene Aufzeichnung, eigene MATLAB-Programme) zugelassen.

0.4 Benotung

Die Benotung setzt sich aus der Klausurnote (70%) und der Übungsblattnote (30%) zusammen.

1 Einführung

1.1 Wozu Programmieren?

Ein Computer-Programm schreiben zu können ist für fast jede Art der wissenschaftlichen Arbeit unumgänglich. Für das Erforschen neuer Themengebiete muss man häufig erst geeignete Werkzeuge entwickeln oder bereits vorhandene Programme für die Ansprüche eigener Arbeit anpassen und erweitern. Will man zum Beispiel in einem einfachen Verhaltensexperiment verschiedene Bilder zeigen und die Antworten und Reaktionszeiten der Versuchsteilnehmer aufnehmen, muss man mit dem Computer

- die Stimuli herstellen,
- die Präsentation der Stimuli im Experiment kontrollieren,
- die Daten aufnehmen, sortieren, statistisch analysieren und grafisch darstellen.

Wer nicht selber programmieren kann, ist auf Hilfe angewiesen oder kann nur Probleme lösen, die andere bereits zuvor gelöst haben.

Programmieren kann aber auch Spaß machen oder euch helfen, Probleme die auch „von Hand“ lösbar sind, deutlich schneller und eleganter zu lösen.

1.2 Warum MATLAB?

MATLAB ist eine einfach zu erlernende Programmiersprache. MATLAB bietet eine interaktive Benutzeroberfläche und einen großen Umfang an vordefinierten Funktionen, die das Programmieren erleichtern. Ein weiterer Vorteil ist die komfortable Visualisierung von Daten.

1.3 MATLAB zu Hause

MATLAB ist auf den wichtigsten Rechnersystemen verfügbar, erfordert allerdings eine Lizenz. Das ZDV stellt es allen Studenten kostenlos zur Verfügung:

Geht auf: **http://www.zdv.uni-tuebingen.de/** → **Dienstleistungen** → **Software** → **CampusSoftware-Portal**. Nachdem ihr euch mit eurem ZDV-Login eingeloggt habt, könnt ihr die aktuelle MATLAB-Version herunterladen.

1.4 Weiterführende Informationen zu MATLAB

Internet

Die Webseite der Firma MathWorks, die MATLAB entwickelt, bietet aktuelle Informationen zur Programmiersprache und den möglichen Toolboxes.

http://www.mathworks.de/

Für die fortgeschrittenen Programmierer pflegt MATLAB eine File-Exchange- und Newsgroup-Seite, die Lösungen zu einigen bereits gelösten Programmieraufgaben bietet.

http://www.mathworks.com/matlabcentral/

Bücher

Hanselman, D.C. and Littlefield, B.L. Mastering MATLAB 7, 2005

Rosenbaum, D. A. MATLAB for Behavioral Scientists, 2007

2 MATLAB Benutzeroberfläche

Die MATLAB-Benutzeroberfläche besteht aus 4 Bereichen (s. Abb. 1).

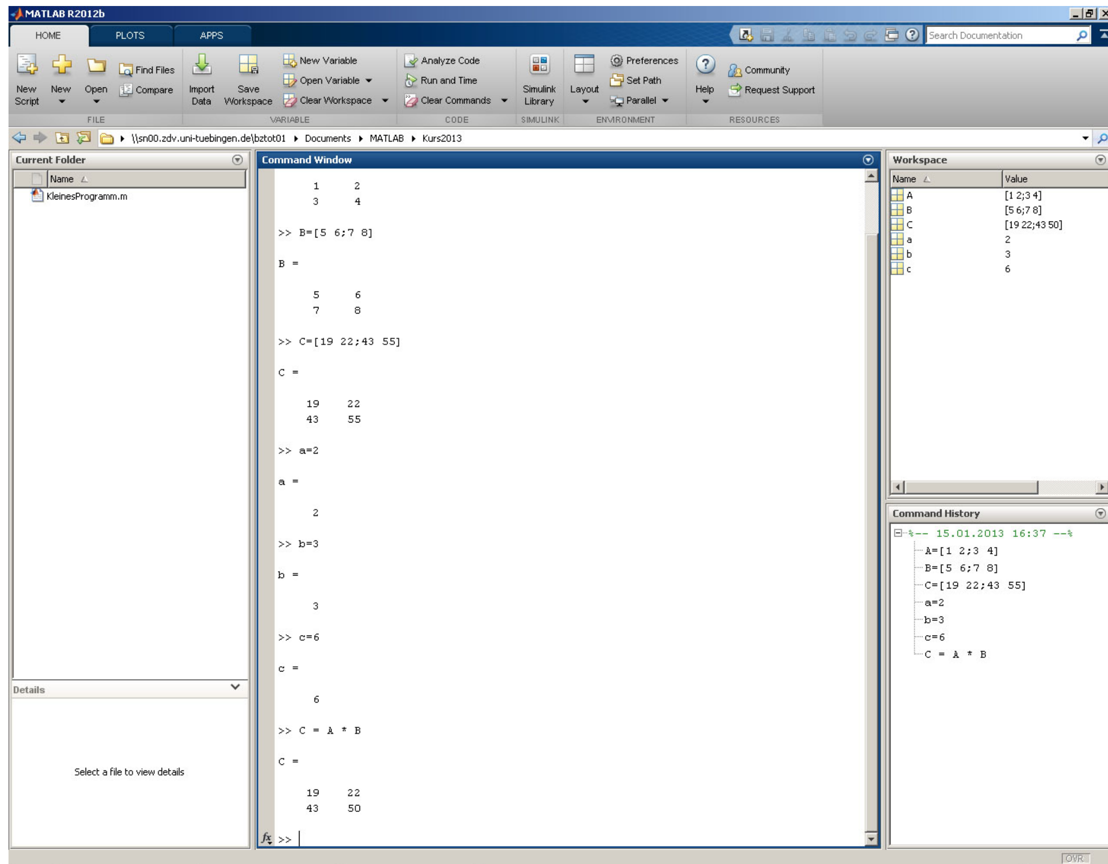


Abbildung 1: Die MATLAB-Benutzeroberfläche.

1. Eingabefenster (*Command Window*): Arbeitsbereich, in dem man mit MATLAB interagiert. Befehle, die MATLAB ausführen soll, werden in der >> Eingabeaufforderung (*prompt*) eingegeben. Wenn die Eingabe beendet ist drückt man **Enter**. MATLAB führt die Befehle aus und gibt das Ergebnis aus. **Tipp:** Die vorangehende Eingabe bekommt man mit der **Cursor**-Taste nach oben. Das ist zum Beispiel praktisch, wenn man einen Tippfehler ausbessern möchte. Wiederholtes Betätigen der **Cursor**-Tasten nach oben oder nach unten ermöglicht ein Scrollen durch die vorangehenden Eingaben.

2. Eingabegeschichte (*Command History*): Zeigt die bisher genutzten Befehle an. Durch Doppelklicken auf ein Befehl wird dieser direkt ausgeführt.

3. Arbeitsspeicher (*Workspace*): Zeigt eine Liste der MATLAB-Variablen an, die gerade verwendet werden. Durch Doppelklicken auf eine Variable kann diese betrachtet, editiert oder gespeichert werden.

4. Aktuelles Verzeichnis (*Current Folder*): Zeigt Dateien des aktuellen MATLAB-Verzechnisses an. Hier werden Skripte und Variablen gespeichert und geladen.

Tipp: Die ursprüngliche Fensterkonfiguration kann mit **Layout →Default** wieder hergestellt werden.

Aufgabe 2.1

MATLAB kann als Taschenrechner verwendet werden.

(a) Gobo will seine Schafe scheren und möchte sich dafür 3 Scherscheren zu je 30.95 Euro kaufen. Außerdem benötigt er 4 Paar Handschuhe für 13 Euro das Paar und eine Thermoskanne für 22 Euro. Wieviele Gegenstände kauft Gobo und wieviel kosten sie insgesamt?
Hinweis: Das Dezimaltrennzeichen ist in MATLAB ein Punkt (wie in 30.95 Euro).
(b) Mokey will sich ein neues Auto (Fiat 500) kaufen. Für ihr altes Auto bekommt sie noch 1 250 Euro. Der neue Fiat 500 kostet 10 000 Euro. Auf ihrem Sparbuch befinden sich aktuell 7 020 Euro. Wieviel muss Mokey noch sparen bevor sie sich das neue Auto leisten kann?
(c) Interpretiere die sichtbaren Inhalte in den MATLAB-Fenstern.

3 Variablen

3.1 Definition der Variable

Will man einen Wert mehrmals benutzen, lohnt es sich eine Variable dafür anzulegen, z. B. `kaufpreis = 10000`. Eine Variable hat immer einen Wert (10000) und einen Variablennamen (kaufpreis). Die Zuweisung von Werten erfolgt in der Form
Variablenname = Wert .

```
>> zahl = 5

zahl = 5

>> anderezahl = 2

anderezahl = 2
```

Danach sind die definierten Werte unter den gewählten Namen gespeichert. Man kann sie jetzt im *Workspace* sehen, jeden Wert unter seinem Namen abrufen und für weitere Berechnungen nutzen:

```
>> zahl

zahl = 5

>> kaufpreis

kaufpreis = 10000

>> billiger = kaufpreis - 1000

billiger = 9000
```

Wird ein Wert nicht zugewiesen, speichert MATLAB diesen in der Variable `ans`.

```
>> 5

ans = 5
```

Konventionen für die Variablennamen

- Variablennamen müssen mit einem Buchstaben beginnen und dürfen nur aus Buchstaben (keine Umlaute), Ziffern und dem Unterstrich _ zusammengesetzt sein (z. B. FavoriteMovie_1, Leute_Raum3, X2345dovb).
- Groß- und Kleinschreibung in den Variablennamen wird unterschieden (zahl, Zahl, zaHl, ZAHL können verschiedene Variablen bezeichnen).
- Von MATLAB reservierte Ausdrücke dürfen nicht als Variablennamen verwendet werden (z. B. `for, end, if, function, else`).

Tipp: Verwende sinnvolle Variablennamen, die dir helfen, dich an die Bedeutung der zugewiesenen Information zu erinnern. Die Wahl der Variablennamen kann auch helfen den gespeicherten Datentyp zu markieren. Zum Beispiel sollten die Namen der Variablen, denen nur eine Zahl zugewiesen wird, nur aus kleinen Buchstaben bestehen (anzahl, tage, radius); Vektoren und Matrizen sollten mit einem Großbuchstaben beginnen (ReaktionsZeiten, DatenTabelle, ResultListe).

Aufgabe 3.1

(a) Gib folgende Befehle im *Command Window* ein und interpretiere die Ausgaben im *Command Window*, die Veränderungen im *Workspace* bzw. die Fehlermeldungen.

```
>> Käfig_1 = 5

>> anzahlzellen = 3892

>> Anzahlzellen
```

```
>> summe

>> for = 0

>> 7 = a
```

(b) MATLAB enthält eine Reihe vordefinierter Variablen. Man kann die Werte dieser Variablen anzeigen und benutzen wie die Variablen, die man selber definiert.

Variable	**Definition**
`ans`	Speicherung des Wertes der letzten Berechnung, wenn dieser Wert keiner anderen Variable zugewiesen wurde.
`pi`	π
`inf`	(∞) unendlich (z. B. 1/0)
`NaN`	Not a Number. Werte die aus ungültigen Berechnungen wie 0/0 oder ∞/∞ resultieren.
`i`	Imaginäre Einheit: Wurzel aus -1.

Gib die vordefinierten Variablen in dein *Command Window* ein und kommentiere die Ausgaben.

3.2 Berechnungen mit Variablen

Variablen und Zahlen können mit Operatoren verknüpft werden + , - , * , / , ^ (Potenz). Dabei können Ausdrücke mit Hilfe von ( ) geklammert werden. Die Auswertung erfolgt in der Reihenfolge Potenzen, Punktrechnungen, Strichrechnungen. Gleichrangige Ausdrücke werden von links nach rechts ausgewertet.

Aufgabe 3.2

(a) Löse die Aufgabe 2.1 mit Hilfe von Variablen.
(b) Führe folgende Berechnungen durch:

```
>> a = 300 / (zahl + 5) ^ 2

>> b = a * anderezahl

>> w = a * (zahl- b) ^ anderezahl

>> w = a * zahl- b ^ anderezahl

>> w = (a * zahl- b ^ anderezahl
```

3.3 Löschen von Variablen

Beim Umgang mit Variablen muss man aufpassen, weil vorhandene Variablen durch gleichnamige Variablen überschrieben werden (beachte die Änderungen im *Workspace*).

```
>> wichtig = 1333983.389722

>> wichtig = 9
```

Nicht mehr verwendete Variablen können durch folgende Befehle gelöscht werden:

Befehle	**Beschreibung**
`clear`	Löscht alle Variablen aus dem *Workspace* und gibt den Speicherplatz frei.
`clear all`	Löscht alle Variablen, Funktionen und debugging breakpoints (siehe 12.3 Fehlerfreies Programmieren).
`clear name1 name2 ...`	Löscht mehrere, ausgewählte Variablen.

Eine Möglichkeit, alle Ausgaben aus dem *Command Window* zu löschen, bietet die Anweisung `clc`, wobei die Variablen immernoch im *Workspace* gespeichert bleiben.

4 MATLAB-Programme

Bislang haben wir nur kurze Aktionen durchgeführt, indem wir MATLAB-Befehle in das *Command Window* eingegeben haben. Um mehrere Befehle hintereinander auszuführen, können diese zusammen in ein MATLAB-Skript geschrieben werden und so nach und nach größere Programme erstellt werden. Um ein MATLAB-Skript zu erstellen gehe auf **New Script**. MATLAB erstellt eine leere Datei und öffnet diese im *Editor*. Hier können die MATLAB-Befehle Zeile für Zeile eingetragen werden.

```
clear all

clc

r = 4

U = 2 * pi * r

A = pi * r ^ 2
```

MATLAB-Skripte können unter **Save**→**Save as** gespeichert werden und erhalten den Suffix `.m`. Skriptnamen dürfen keine Punkte enthalten, keine Umlaute, keine Sonderzeichen (außer Unterstriche) und dürfen nicht mit einer Zahl anfangen. Der Skriptname darf nicht mit einem der Variablennamen in der Datei übereinstimmen.

Hinweis: Wenn Skripte in einem neu angelegten Ordner gespeichert werden, muss der Ordner zu MATLAB hinzugefügt werde. Gehe dazu auf **Home→Set Path→Add with Subfolders**.

Das gesamte MATLAB-Programm kann mit dem **Editor→Run**-Symbol in der Taskleiste des Editor-Fensters (s. Abb. 2) oder mit der **F5**-Taste ausgeführt werden. MATLAB liest das Skript Zeile für Zeile und führt jede Zeile als Befehl aus. Die Ausgabe der Berechnungen erfolgt im *Command Window*.

Tipp: Man kann einen Teil des Skripts ausführen, indem man diesen Teil markiert und nach einem Klick mit der rechten Maustaste den Punkt **Evaluate Selection** aussucht oder mit der Funktionstaste **F9** ausführt.

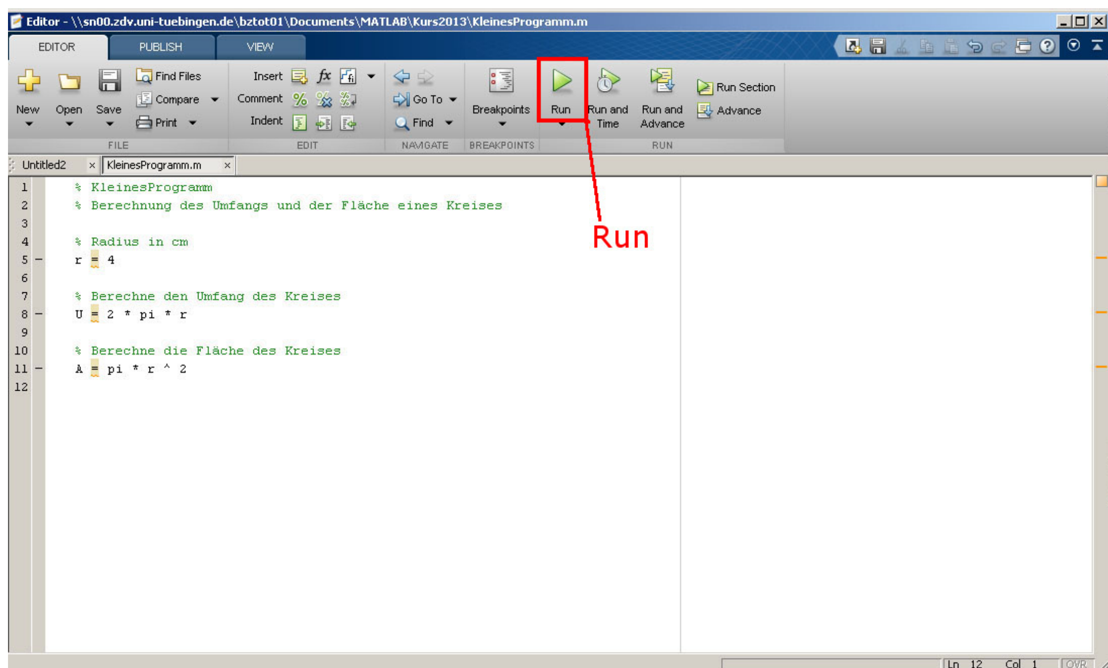


Abbildung 2: Das Editor-Fenster.

Um einen besseren Überblick über Programme zu behalten und sie auch später noch zu verstehen, können und sollen Kommentare eingefügt werden. Zeilen, die mit einem %-Zeichen beginnen, werden von MATLAB ignoriert und nicht ausgeführt.

Tipp: Beginne jedes MATLAB-Skript mit einer Kommentarzeile in der der Skriptname steht. Danach sollte eine kurze Beschreibung des Programms folgen. Jeder wichtige Berechnungsschritt und Variable sollte kommentiert werden.

```
% Kreisberechnungen

% Berechnung des Umfangs und der Fläche eines Kreises
```

```
clear all

clc

r = 4 % Radius in cm

% berechne den Umfang des Kreises

U = 2 * pi * r

% berechne die Fläche des Kreises

A = pi * r ^ 2
```

5 Input und Output

Bis jetzt haben wir MATLAB wie einen Taschenrechner verwendet, sodass nach jeder Berechnung oder Zuweisung von Variablen MATLAB das Ergebnis automatisch im *Command Window* angezeigt hat. Die meisten Programme, die man benutzt, benötigen allerdings bestimmte Eingabewerte (*Input*), führen damit Berechnungen durch und zeigen zum Schluss nur die relevanten Ergebnisse (*Output*). Dabei wird die Ausgabe der Zwischenergebnisse unterdrückt.

5.1 Semikolon in MATLAB

Um eine Ausgabe zu unterdrücken, sollte am Ende jeder Zeile ein Semikolon (`;`) eingefügt werden. In diesem Fall führt MATLAB die Berechnungen und Zuweisungen der Variablen aus (vgl. *Workspace*), sodass diese später verwendet werden können, zeigt das Ergebnis aber nicht an.

```
clear all

clc

r = 4; % Radius in cm

% berechne den Umfang des Kreises

U = 2 * pi * r;

% berechne die Fläche des Kreises

A = pi * r ^ 2;
```

5.2 Output

Für eine Ausgabe im MATLAB *Command Window* eignet sich der Befehl `fprintf()`. Hier werden wir nur auf ausgewählte Formen der Ausgaben eingehen. Um einen Text auszugeben wird in den Klammern der Text in `''` eingegeben.

```
>> fprintf('Hier steht der Text! \n');

Hier steht der Text!
```

Bemerke, hier unterdrückt ein Semikolon nicht die Ausgabe. Wichtig ist das `\n` am Ende der Eingabe, da dieses Zeichen MATLAB dazu auffordert, eine neue Zeile zu starten. Anderenfalls würde die nächste Ausgabe exakt in der gleichen Zeile folgen.

```
>> fprintf('Hier steht der Text!');

>> fprintf('Das soll in die nächste Zeile!!!');

Hier steht der Text!Das soll in die nächste Zeile!!!
```

Wenn eine Variable in die Textausgabe eingebettet sein soll, kann `%g` als Formatierungszeichen eingefügt werden. Dieses wird bei der Ausgabe durch die erforderliche Variable ersetzt.

```
>> result = 50 + 48 + 9939;

>> fprintf('Das Ergebnis beträgt: %g \n', result);

Das Ergebnis beträgt: 10037
```

Textausgaben im *Command Window* können auch mit Hilfe der Funktion `disp('Text')` gemacht werden.

5.3 Input

Das Abfragen von Eingabewerten des Programmbenutzers kann mit dem Befehl `input('Text')` erfolgen und kann einer Variable zugeschrieben werden.

```
>> jahr = input('In welchem Jahr bist du geboren? ')

In welchem Jahr bist du geboren? 1987

jahr = 1987
```

Beispiel

```
clear all
```

```
clc

% Radius in cm

r = input ('Gib einen Radius in cm ein: ');

% berechne den Umfang des Kreises

U = 2 * pi * r;

% berechne die Fläche des Kreises

A = pi * r ^ 2;

fprintf('Bei einem Radius von %g cm \n Fläche des Kreises: %g \n
Umfang des Kreises %g.', r, A, U );
```

Hinweis: Wenn ein Programm gestartet wurde, wird es in einer Zeile mit einer Nutzerabfrage solange pausiert, bis etwas eingegeben wurde. Häufig passiert es, dass man das Programm neu startet, ohne zu merken, dass noch auf einer Nutzereingabe gewartet wurde. Das neu gestartete Programm wird erst ausgeführt, wenn das alte durch die Nutzereingabe(n) beendet wurde. Mit **Strg+C** (im Fenster *Command Window*) lässt sich das laufende Programm sofort abbrechen.

Aufgabe 5.1

Programmiere ein kleines Programm. Denk dir eine Zahl aus und multipliziere diese mit 2, addiere 5, multipliziere mit 50 und addiere 1757. Wenn du in diesem Jahr schon Geburtstag hattest, addiere 1 hinzu, ansonsten 0. Subtrahiere schließlich dein Geburtsjahr (vierstellig). (**Tipp:** Benutze den Input-Befehl für die Zahleneingabe, die Geburtsjahr- und Geburtstagsabfrage und gib mit `fprintf` das Ergebnis aus.)

6 Datentypen

Bisher haben wir MATLAB verwendet, um Berechnungen mit Zahlen anzustellen. Normale Zahlen werden in MATLAB als *double* bezeichnet. Im vorherigen Abschnitt kam bereits ein weiterer Datentyp zum Einsatz, die Zeichenketten oder *strings*. Strings sind Ketten aus Buchstaben oder Zahlen, den *characters* (abgekürzt mit *char*), mit denen man zunächst keine Berechnungen durchführen kann. Ein dritter wichtiger Datentyp sind *logicals*. Eine logische Zahl kann nur die Werte 0 (*false*) oder 1 (*true*) annehmen. Für eine Übersicht über die Datentypen s. Tab. 1.

Variablen können wie gewohnt Zahlen zugeordnet werden. Um einer Variable ein Zeichen zuzuweisen, wird das Zeichen stets zwischen einfache Anführungszeichen (z. B. `'s'`) gesetzt. Man kann eine logische Zahl erzeugen, indem man einer Variable den Wert `true` oder `false` zuweist.

Datentyp	Bezeichnung in MATLAB	Speicherverbrauch	Beispiele
Zahl	*double*	8 Byte	`1`, `5.345`
Zeichen	*char*	2 Byte	`'s'`, `'1'`
Logische Zahl	*logical*	1 Byte	`true`, `false`

Tabelle 1: Wichtige Datentypen in MATLAB.

Beispiel

```
>> a = 'a'

a = a

>> b = 5

b = 5

>> c = true

c = 1

>> d = false

d = 0
```

Beachte die jeweilige Ausgabe von MATLAB. Mit dem Befehl `whos` oder `whos` *`Variable`* kann der Datentyp aller Variablen im *Workspace* oder einer bestimmten Variable angezeigt werden. Auch im *Workspace*-Fenster wird der Datentyp jeder Variable in der Spalte *Class* angezeigt.

```
>> whos

 Name  Size  Bytes  Class
 a     1x1       2  char
 b     1x1       8  double
 c     1x1       1  logical
 d     1x1       1  logical
```

Beachte, dass auch Zahlen als `char` eingegeben werden können. **Vorsicht:** Diese Zahlen werden von MATLAB als `char` behandelt und Berechnungen mit solchen Variablen führen zu falschen Ergebnissen!

```
>> e = '5'

e = 5

>> whos e
```

```
Name  Size  Bytes  Class
e     1x1       2  char
```

Wenn man mehrere Zeichen hintereinander zu einer Zeichenkette zusammenfügt, entsteht ein *String*. Der Befehl `fprintf()`, den wir im vorherigen Kapitel kennen gelernt haben, benutzt für die Textausgabe Strings.

Beispiel

```
>> zeichenkette = 'Hallo Welt! \n';

>> fprintf(zeichenkette);

Hallo Welt!
```

7 Funktionen

Funktionen sind kleine Programme, die meist eine Eingabe bekommen und daraus meist eine Ausgabe berechnen. Funktionen kennt man schon aus der Mathematik. Die Funktion $f(x) = 12x + 3$ berechnet z. B. für jede Eingabe x einen dazugehörigen Ausgabewert f(x) auf eine bestimmte, durch die Funktion festgelegte Art. Statt f(x) könnten wir so eine Funktion in MATLAB auch einfach steilegerade(x) nennen. Wir lernen in diesem Kurs nur, Funktionen zu benutzen, und nicht selbst zu schreiben. Nachdem also irgendjemand so eine Funktion geschrieben hat, benutzt man sie wie folgt: die Eingabe(n) kommen in runde Klammern hinter den Funktionsnamen, will man das Ergebnis verwenden muss man es einer Variable zuweisen:

```
>> x = 1;

>> y1 = steilegerade(x)

y1 = 15

>> x2 = 2;

>> steilegerade(x2)

ans = 27
```

Man kann die Funktion also mit allen möglichen Werten aufrufen, und die Ergebnisse in irgendwelchen Variablen speichern. In MATLAB gibt es schon eine große Menge nützlicher Funktionen vordefiniert:

- mathematische Funktionen:

`cos, sin, tan`	Kosinus, Sinus, Tangens
`exp, pow2`	Exponentialfunktion mit Basis exp bzw. 2
`log, log10, log2`	Logarithmus zur Basis exp, 10 bzw. 2
`sqrt, realsqrt`	Quadratwurzel bzw. Einschränkung auf reelle positive Zahlen
`round, floor, ceil`	runden, abrunden, aufrunden
`mod, rem, sign`	Modul, Divisionsrest, Vorzeichen

- Funktionen für Berechnungen aus mehreren Zahlen (siehe Vektor-Kapitel)

`mean`	Mittelwert bilden
`sort`	Elemente sortieren
`sum`	Summe bilden

- Funktionen zur Konvertierung verschiedener Datentypen

`str2num`	String in Zahlwert umwandeln
`num2str`	Zahl in String verwandeln
`logical`	Zahl in logische Werte umwandeln

Eine Übersicht der verfügbaren Operatoren bzw. elementaren Funktionen bekommt man mit Hilfe der Befehle

```
>> help matlab/ops
```

```
>> help matlab/elfun
```

8 Das Hilfe-System von MATLAB

Das zentrale Hilfe-System von MATLAB wird über **Help→Documentation** (**F1**) aufgerufen. Hier auf **MATLAB** klicken. Hier sind Hilfe-Themen im Browser-Stil aufgelistet. Unter **Getting Started** kann man verschiedene Tutorials und Einführungsvideos anschauen.

Ein Vorteil von MATLAB ist die Verfügbarkeit von vielen vordefinierten Funktionen, die das Rechnen und Progammieren erleichtern. Man findet Funktionen, indem man durch die Themen im Hilfeportal klickt. Einen Überblick über die verfügbaren Funktionen findet man, wenn man ganz unten in der MATLAB-Hilfe auf **MATLAB Functions** klickt. Hier sind alle MATLAB-Funktionen nach Kategorien aufgelistet. Man kann sich auch eine alphabetische Liste anzeigen lassen. Außerdem kann man unter **Search Documentation** nach Funktionen suchen. Dabei werden auch andere Funktionen oder Dokumentationsseiten gefunden, in denen die gesuchte Funktion Verwendung findet.

Funktionen sind meist recht intuitiv nach dem, was sie leisten, benannt (z. B. `log` berechnet den Logarithmus). Klickt man einen Funktionsnamen in der Hilfe an, so sieht

man die Beschreibung im rechten Fenster.

Um die Hilfe für eine bestimmte Funktion direkt im *Command Window* einzusehen, kann man dort

```
>> help gesuchte Funktion
```

eingeben.

Eine ausführlichere Hilfe mit mehr Beispielen erhält man jedoch im Hilfe-Browser, den man für eine bestimmte Funktion wie folgt aufrufen kann:

```
>> doc gesuchte Funktion
```

Tipp: Mit **F1** wird eine Pop-up Version der MATLAB-Hilfe aufgerufen. Wenn der Cursor auf einer MATLAB-Funktion in einem Skript platziert ist, ruft **F1** direkt die Hilfeseite zu der entsprechenden Funktion auf.

Aufgabe 8.1

(a) Suche `log, sqrt` und `exp`. Verwende die Funktionen mit den Zahlen 1, 3 und 5.
(b) Lasse dir die Funktionshilfe für die Funktionen in deinem letzten Skript anzeigen.
(c) Mit welcher Funktion kann man die n-te Wurzel einer Zahl bestimmen? Rufe die entsprechende Funktionshilfe auf und finde heraus, wie man diese Funktion benutzt. Bestimme die 2-te, 3-te und 4-te Wurzel aus 16.

9 Fehlerfreies Programmieren

Die größte Herausforderung beim Programmieren ist es, sicherzustellen, dass das Programm wirklich das macht, was man will. Es ist einfach herauszufinden, dass das Programm nicht macht, was es soll, falls MATLAB eine *Error*-Nachricht ausgibt. Andere Fehler können besser versteckt sein – nur weil das Programm durchläuft, heißt das noch nicht, dass es auch das macht, was es soll. Um Programme zu korrigieren geht man schrittweise vor – diese Tätigkeit nennt man *Debugging*. Idealerweise sollte Debugging natürlich nicht notwendig sein, wenn man von vorneherein alles richtig gemacht hat. Die Realität sieht leider anders aus und Debugging ist eine wichtige Aktivität, die alle Programmierer beherrschen sollten.

Für den Aufbau von Computerprogrammen gelten die selben Grundsätze wie z. B. beim Aufbau einer Messapparatur, o. ä. Stellt euch vor, ihr müsst einen komplizierten Versuchsaufbau zusammenbauen. Wer erst alles aufbaut, dann ausprobiert, und dann einfach alles mögliche verändert, wenn es nicht funktioniert, wird nie zum Ende kommen und bald frustriert sein. Hier muss man Schritt für Schritt vorgehen, einzelne Elemente

hinzufügen und nach jeden Schritt testen, ob bis dahin alles funktioniert. Wenn es am Ende trotzdem nicht funktioniert, verändert man immer eine einzige Sache und checkt nach jeder Veränderung, was sich getan hat. Das gleiche gilt beim Programmieren.

Lesbarkeit

Das wichtigste für fehlerfreies Programmieren ist ein gut lesbares Programm. Außerdem hilft das anderen Leuten, oder euch selbst nach einigen Monaten, euren Code schneller zu verstehen.

Gute Lesbarkeit erreicht man vor allem durch

- korrekte Einrückung des Codes
- konsequente Benutzung von Variablen statt hartgecodeten Zahlen
- sinnvolle Variablennamen
- Kommentare

Damit längere Programme übersichtlich werden, sollten Einrückungen der Programmblöcke vorgenommen werden. Dazu markiert man einen Teil oder das gesamte MATLAB-Skript und drückt dann die Tastenkombination **Strg+I**. Noch passiert dabei nicht viel, sobald wir aber Schleifen und Verzweigungen benutzen erhöht die automatisches Einrücken die Übersichtlichkeit enorm!

Man sollte so wenig wie möglich konkrete Werte „hartcoden“ d. h. beim Programmieren etwas festlegen, das man später nur noch schwer ändern kann. Als Faustregel gilt: eigentlich sollten gar keine richtigen Zahlen z. B. `EinzelSchulden = 85/12` in eurem Code vorkommen, sondern nur Variablen, mit sinnvollen Namen. Dann versteht man später leichter, warum dort nochmal eine 12 stehen sollte. Außerdem definiert man die Variable einmal im Programm, und kann sie dann viele Male benutzen. Falls sich einmal etwas ändert, z. B. eine weitere Person sich am Gruppengeburtstagsgeschenk beteiligen möchte, muss man nur eine einzige Variable ändern und nicht alle Stellen in Code suchen, an denen eine 12 vorkommt:

```
AnzahlPersonen = 12;

GesamtKosten = 85;

EinzelSchulden = GesamtKosten/AnzahlPersonen;

AnzahlColaflaschen = 5*AnzahlPersonen;
```

Schrittweises Vorgehen beim Programmieren

Bevor man überhaupt anfängt eine Zeile Code zu schreiben, überlegt man sich (bei größeren Problemen ruhig auch mit Zettel und Stift), wie das Problem zu lösen ist. Fast jede Aufgabe kann man in kleinere Teilaufgaben zerlegen, die man dann nacheinander lösen und einzeln testen kann. Man sollte also möglichst immer kurze, einfache Programme schreiben, die man leicht überblicken kann. Wenn man mit so einem Teilprogramm fertig ist, testet man es zunächst ausgiebig (z. B. „wenn der Radius 3 ist sollte hier 8 rauskommen. Was passiert wenn ich für den Radius 0 eingebe?“, etc.). Wenn alle Tests erfolgreich verlaufen und das Programm genau das macht, was ihr wolltet (und nur dann!) geht man zum nächsten Schritt: man vergisst alles darüber, wie das Teilprogramm nochmal genau funktioniert, und benutzt jetzt einfach seine Ergebnisse – schließlich haben wir gut getestet und können unserem Programm vertrauen. Im Code schreiben wir über diesen Block einen Kommentar, der erklärt, was in dem Teil des Codes gemacht wird, und fangen dann einen neuen Block an (später kann man auch lernen, alle Teilprobleme in einzelnen Funktionen zu verpacken). Ein weiterer Vorteil dieser Art des Programmierens ist, dass wir die Lösungen von Teilproblemen, also Stücke von unserem Code, eventuell für andere Aufgaben wiederverwenden können.

Beispiel

Es soll das Volumen eines Kegels mit Radius 3 und Höhe 3 berechnet werden. Wir könnten das Problem in einer Zeile Code lösen:

```
volumen = 1/3 * 3^2 * pi * 3;
```

Hierbei macht man allerdings leicht Fehler, außerdem fällt es uns nach etwas zeitlichem Abstand schwer, diesen Code wieder zu verstehen. Wenn wir die Aufgabe für einen anderen Radius lösen wollen, müssen wir uns wieder komplett reindenken, um zu wissen, welche 3 man ändern muss. Und wenn wir z. B. das Volumen eines Zylinders berechnen wollen, müssen wir wieder komplett von vorne anfangen. Besser gehen wir so vor:

Wir zerlegen das Problem in seine Teilaufgaben: zuerst die Fläche des Kreises berechnen, dann das Volumen des Kegels. Jetzt lösen und testen wir jeden Schritt einzeln:

```
radius = 3;

hoehe = 3;


% berechnung der Kreisfläche

kreisflaeche = radius^2 * pi;
```

```
% berechnung des volumens von kegel und zylinder

kegelvolumen = 1/3 * kreisflaeche * hoehe;

zylindervolumen = kreisflaeche * hoehe;
```

Dieser Code ist zwar länger als die erste Lösung, vielleicht hat es auch ein bisschen länger gedauert ihn aufzuschreiben, aber jetzt können wir ihn für alles Mögliche benutzen und jederzeit leicht einzelne Parameter ändern, ohne die allgemeinen Berechnungsformeln weiter unten im Code anschauen oder verstehen zu müssen. Bei dieser Aufgabe wirkt das vielleicht ein wenig übertrieben, aber bei größeren Programmen wird diese Vorgehensweise unumgänglich.

Schrittweises Verbessern von Fehlern

Wenn wir so vorgehen, wie im vorigen Absatz beschrieben, sollten eigentlich keine größeren Fehler auftauchen.

Wenn man bereits weiß, welches Teilstück des Codes fehlerhaft ist, sollte man dieses zunächst isolieren. Man kann alles, was man nicht braucht, auskommentieren – so ist es nicht weg, aber wird nicht ausgeführt. Markiere dafür den Teil im Code und drücke **Strg+R** (zurück bekommt man es wieder mit **Strg+T**). Variablen, die das fehlerhafte Stück Code zur Ausführung benötigt, kann man zwischenzeitlich mit Spielzeugvariablen ersetzen: Wollen wir eigentlich eine Berechnung auf einer riesigen Matrix ausführen, und der Code muss 10 Minuten laufen, bevor wir den Fehler nochmal anschauen können, ersetzen wir diese Matrix durch eine deutlich kleinere Matrix, damit wir den Code immer und immer wieder kurz ausprobieren können.

Wenn wir den Fehler so reproduzieren konnten, gilt es, die fehlerhafte Stelle im Code weiter zu isolieren. Hierfür müssen wir Zwischenergebnisse im *Command Window* ausgeben lassen, indem wir die Semikolons weglassen, oder `print` Befehle benutzen. Man kann auch Teile des Codes (und sogar Teile von einzelnen Zeilen) markieren und durch F9 im Command Window auswerten lassen.

In solchen Situationen ist auch das Debugging Werkzeug hilfreich.

MATLABs Debugging-Werkzeug

Das Debugging-Werkzeug von MATLAB ist in den Editor integriert. Man kann sogenannte Breakpoints an alle Zeilen setzen, an der das Programm stoppen soll (siehe Abb. 3). Klicke dazu einfach auf den Strich links neben der entsprechenden Zeile. So kann man testen, ob der Fehler bis dahin auftritt, oder erst später im Programm. Bei jedem Breakpoint stoppt das Programm und im *Command Window* erscheint `K>>` . Jetzt kann man *Workspace*-Variablen abfragen, Werte ändern, etc., um den Bug zu finden und

zu fixen. Das Programm läuft weiter, nach dem man **Return** gedrückt hat.

Achtung! Die Ã„nderungen, die man im Debugging-Modus vornimmt, werden erst nach dem Speichern wirksam.

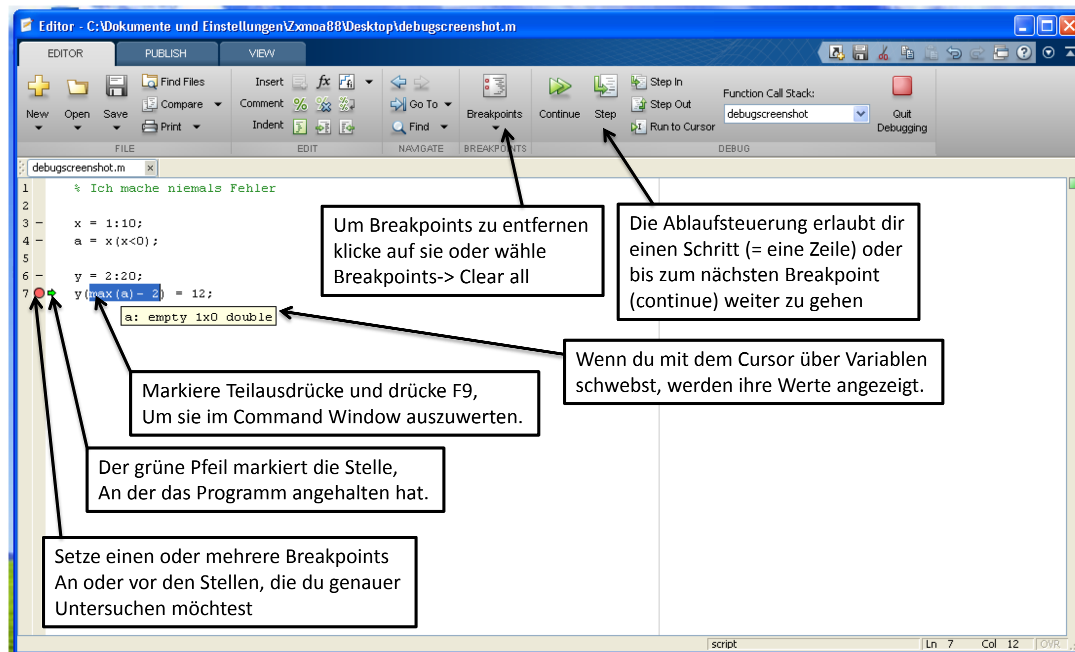


Abbildung 3: Debugging

Häufige Fehlermeldungen

- `Undefined function or variable xxx`: MATLAB kennt ein von euch benutztes Wort nicht. Meistens wegen eines Tippfehlers, oder weil eine Variable benutzt wird, bevor sie definiert wurde. Außerdem könnte man bei einem String die Anführungszeichen vergessen haben, sodass MATLAB das Wort als Schlüsselwort interpretiert.

- `Unknown function for input type xxx`: Meistens kann man den Teil mit „Input type“ ignorieren. So wie beim vorigen Fehler kennt MATLAB den Namen eurer Funktion nicht, aus den oben genannten Gründen. Falls Funktionen, die bisher immer funktioniert haben, plötzlich nicht mehr gefunden werden, ist man häufig im falschen Ordner, bzw. der Ordner ist nicht auf dem MATLAB-Suchpfad gespeichert.

Weitere Fehlerquellen:

Das Ergebnis einer Rechenoperation führt zu einem nicht definierten Ergebnis (`NaN`), z. B. bei Division 0/0. Alle Operation an `NaN`s geben `NaN`s zurück. Sollte `NaN` ein mögliches Ergebnis sein, ist es sinnvoll die logische Funktion **`isnan`** zu verwenden, um `NaN`s aufzuspühren und passend zu behandeln.

10 Vektor

Der Vorteil von MATLAB ist, dass große Datensätze leicht verarbeitet werden können. Datensätze können z. B. als eine Liste von Zahlen dargstellt werden, d. h. einem *Vektor*.

10.1 Konstruktion von Vektoren

In einen Vektor können viele Elemente gleichen Datentyps geschrieben werden, z. B. Alter, Geld, Messerergebnisse eines Versuchs, etc. Es gibt Spalten- und Zeilenvektoren.

Erzeugung eines Zeilenvektors in MATLAB:

```
>> A = [1 2 3]

A = 1 2 3
```

Erzeugung eines Spaltenvektors in MATLAB:

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

A =
1
2
3
```

Wie gewohnt wird der Vektor bei dieser Zuweisung in der Variable `A` gespeichert.

Aufgabe 10.1

(a) Konstruiere einen Zeilenvektor, der deinen Geburtstag darstellt (`[`*`Tag Monat Jahr`*`]`).
(b) Konstruiere einen Spaltenvektor, in dem nur gerade Zahlen zwischen 0 und 10 stehen.

Um Aufgabe (b) zu lösen gibt es einen Trick. Bei MATLAB können Vektoren mit „geregeltem“ Abstand vereinfacht eingegeben werden. Dazu verwendet man den Doppelpunkt-Operator ‚`:`‘.

```
>> X = 0:2:10

X = 0 2 4 6 8 10
```

Die Schreibweise `0:2:10` bedeutet: Beginne mit 0 und zähle 2 dazu, dann zähle wieder 2 dazu, usw., bis die Grenze 10 erreicht ist. Diese Anweisung ist bei der Eingabe von Vektoren mit vielen Elementen sehr nützlich. Man kann die Anweisung `0:2:10` auch in eckigen Klammern schreiben: `[0:2:10]`. Die Schrittweite (*increment*) darf negativ sein:

```
>> U = 29:-2:0

U = Columns 1 through 8
29 27 25 23 21 19 17 15
Columns 9 through 15
13 11 9 7 5 3 1
```

Dieser Vektor hat 15 Elemente, d. h. 15 „Spalten" (*columns*). **Beachte:** Das letzte Element in diesem Beispiel ist 1. Zieht man 2 immer wieder von 29 ab, so trifft man nie auf 0. Die letzte Zahl gößer 0, die man erhält, ist 1.

Es gibt eine weitere hilfreiche Funktion zur Erzeugung von Vektoren:

```
>> s = linspace(5,-3,10);

>> s'

ans =
5.0000
4.1111
3.2222
2.3333
1.4444
0.5556
-0.3333
-1.2222
-2.1111
-3.0000
```

Das Kommando `linspace` erzeugt einen Vektor mit 10 Elementen, die zwischen 5 und -3 (inklusive) liegen. `linspace` wählt die Elemente so, dass alle den gleichen (linearen) Abstand voneinander haben. Eine ï¿½hnliche Funktion ist `logspace`, welche einen logarithmischen Abstand zwischen den einzelnen Elementen liefert. Möchte man aus einem Spaltenvektor einen Zeilenvektor (oder umgekehrt) machen, kann der Vektor mittels `'` transponiert werden.

Um ein bestimmtes Element eines Vektors auszugeben oder für weitere Berechnungen zu verwenden, bestimme die Position an der das gesuchte Element im Vektor steht, und schreibe `U(`*`Position`*`)`, z. B.:

```
>> U(3)

ans = 25
```

Eine weitere Verwendung des Doppelpunkt-Operators besteht darin, sich mehrere oder alle Elemente, die in einem Vektor stehen, ausgeben zu lassen. Hier werden z. B. die Werte in den Positionen 12 bis 15 gesucht:

```
>> U(12:15)

ans = 7 5 3 1
```

Wenn man nicht weiß, wie lang ein Vektor ist, und man sich ab einer bestimmten Position alle Elemente des Vektors bis zum Schluss ausgeben lassen möchte, kann man den Befehl `end` benutzen.

```
>> U(12:end)

ans = 7 5 3 1
```

Man kann die Vektorelemente, die man sich ausgeben lassen möchte, beliebig kombinieren:

```
>> U([3:5,8,end-1])

ans = 25 23 21 15 3
```

Um alle Elemente ausgeben zu lassen, schreibt man `U(:)`.

Nicht nur der Zugriff auf Elemente ist wichtig, sondern auch das Verändern von Vektoren kann notwendig sein. Einträge müssen verändert, gelöscht oder Vektoren zu größeren Vektoren zusammengefügt werden. Um einen oder mehrere Einträge eines Vektors zu ändern kann man den entsprechenden Positionen neue Werte zuweisen.

```
>> X = [0:2:10];

>> X(2) = 100

X = 0 100 4 6 8 10
```

Ein Element eines Vektors kann man löschen, indem man diesem Element einen leeren Wert `[]` zuweist.

```
>> X(2)=[]

X = 0 4 6 8 10
```

Beachte: Der Vektor `X` ist nun um ein Element kürzer!

Einem Vektor können auch mehrere Elemente hinzugefügt werden. Sei `a = 4` und `A = [1 2 3]`. Um den Vektor `A` zu erweitern, schreibe

```
>> A = [A a]
```

```
A = 1 2 3 4.
```

Beachte: Der Vektor `A` ist nun um ein Element länger!

Aufgabe 10.2

(a) Bilde die Folge der Zahlen

1 2 3 4 5 6 ...
2 4 6 8 10 ...
1998 1999 ...
30 27 24 ...
(jeweils 10 Elemente)

(b) Rufe jeweils das 5. Element der erzeugten Vektoren aus (a) auf. Erzeuge mit diesen Elementen einen neuen Zeilen- und Spaltenvektor.
(c) Bilde Vektoren, die aus den letzten 3 Elementen der Vektoren aus (a) bestehen.
(d) Erstelle (geschickt) folgende Vektoren

1 2 3 4 5 6 7 55 9 10
2 4 6 8 12 14 16 18 20
1 2 3 4 5 6 7 11 12 13 14 15

10.2 Vordefinierte Vektoren

Es gibt Vektoren, die man immer wieder braucht und die mühsam konstruiert werden müssten. Dazu gibt es bei MATLAB vordefinierte Vektoren, z. B. `ones(1,n)` bzw. `ones(n,1)`. Dies ist ein Zeilen- bzw. Spaltenvektor der Länge n, der als Einträge nur 1 enthält.

```
>> ones(1,4)

ans = 1 1 1 1

>> ones(4,1)

ans =


1

1

1

1
```

Aufgabe 10.3

Suche in der MATLAB-Hilfe die Funktion `zeros` und `rand`, benutze sie zur Erzeugung eines Zeilen- und Spaltenvektors und erkläre ihren Output.

10.3 Operationen mit Vektoren

MATLAB kennt arithmetische Standardoperatoren, die man in der Vektorrechnung anwenden kann, wie Addition (`+`) und Subtraktion (`-`), Multiplikation (`*`) und Division (`/`), sowie Potenzierung (`^`). Bei diesen Operationen muss man unterscheiden, ob man einen Vektor mit einem *Skalar* (einer einfachen Zahl) verrechnet, oder mit einem zweiten Vektor gleicher Länge.

Bei einer Addition, Subtraktion, Multiplikation oder Division eines Vektors mit einem Skalar wird der Skalar mit jedem Element des Vektors einzeln verrechnet.

Bei einer Addition oder Subtraktion zweier Vektoren gleicher Länge werden die Einträge der Vektoren elementweise addiert bzw. subtrahiert. Der Multiplikationsoperator `*` führt dagegen das *Skalarprodukt* aus, d. h. das Rechenergebnis einer solchen Operation ist ein Skalar. Dabei ist es nur möglich einen Zeilenvektor mit einem Spaltenvektor gleicher Länge zu multiplizieren. Möchte man zwei Zeilen- oder zwei Spaltenvektoren gleicher Länge elementweise miteinander multiplizieren, dividieren oder potenzieren, schreibt man einen Punkt vor den entsprechenden Operator, d. h. `.*`, `./`, bzw. `.^`.

Aufgabe 10.4

(a) Erstelle einen Zeilen- und einen Spaltenvektor mit 20 Elementen, die jeweils den Wert 4 besitzen. Löse diese Aufgabe einmal durch Addition, einmal durch Multiplikation und einmal durch Potenzierung.
(b) Multipliziere `A = [3 3 3 4]` mit `B = [5 5 2 1]` elementweise.
(c) Multipliziere die Vektoren `A` und `C = B'` (Skalarprodukt). Was unterscheidet das Ergebnis von dem Produkt aus (b)?

Die meisten MATLAB-Funktionen lassen sich direkt mit Vektoren verwenden. Möchte man beispielsweise den Logarithmus nicht nur einer Zahl, sondern einer ganzen Liste von Zahlen berechnen, so kann man die `log`-Funktion mit einem Vektor aufrufen. Ausgegeben wird dann ein Vektor gleicher Länge, bei dem jeder Wert des Eingabevektor logarithmiert wurde. Andere Funktionen berechnen bestimmte Größen aus allen Werten des Eingabevektors. Die Funktion `mean` berechnet zum Beispiel das arithmetische Mittel aus allen Werten des Eingabevektors und gibt eine Zahl, den Mittelwert, aus.

Aufgabe 10.5

(a) Berechne den natürlichen Logarithmus von den ganzen Zahlen 1 bis 10.
(b) Berechne die Werte des Sinusfunktion an den Stellen 0, pi/2, pi, 3pi/2, 2pi, ..., 10pi.
(c) Berechne den Mittelwert der Zahlen 3.7, 4.5, 2.9, 3.1, 4.6.

(d) Sortiere den Vektor `[3 1 7 2 5 6 1 8]` der Größe nach.

Aufgabe 10.6

Für diese Aufgabe werdet ihr neue Funktionen brauchen. Versucht diese in der Hilfe oder im Internet zu finden.

Wembley hat in einem Turnwettberwerb mitgemacht. Sie hat die folgenden Punkte bei 4 Disziplinen bekommen:

Bodenturnen: 8.9, 8.8, 8.8, 8.7, 8.9
Schwebebalken: 8.9, 8.9, 8.7, 8.6, 8.5
Barren: 9.5, 9.3, 9.3, 9.25, 8.9
Bock: 8.9, 8.7, 8.2, 9.1, 9.0

Wembleys Endpunktstand für jede Disziplin ist die durchschnittliche Punktzahl, die man erhält, wenn man die Bestpunktzahl und die schlechteste entfernt und dann den Mittelwert bildet. Schreibe ein Programm, das die Endpunktzahl in jeder Disziplin einzeln berechnet. Berechne ihren gesamten Endpunktstand in allen Disziplinen zusammen.

Aufgabe 10.7

In folgender Tabelle stehen die Personenanzahlen, die in einem bestimmten europäischen Land regelmäßig das Internet nutzen, sowie die Facebook-Nutzer des Landes (Quelle: www.internetworldstats.com, Stand 2011). Bestimme den Anteil an Internet-Nutzern für jedes Land, sowie den Anteil der Facebook-Nutzern für jedes Land in Prozent. Bestimme außerdem den Anteil der Facebook-Nutzer an den Internet-Nutzern jedes Landes. Erstelle dafür zunächst drei Vektoren, in denen die Einwohnerzahl, die Zahl der Internet-Nutzer bzw. die Zahl der Facebook-Nutzer gelistet sind (achte auf die Reihenfolge).

Land	Einwohner (Mio.)	Internet-Nutzer (Mio.)	Facebook-Nutzer (Mio.)
Russland	138.739	59.7	4.648
Deutschland	81.471	65.125	19.459
Türkei	78.785	35.000	29.459
Frankreich	65.102	45.262	22.713
UK	62.698	51.442	29.880
Italien	61.016	30.026	19.806
Spanien	46.754	29.093	14.409
Polen	38.441	22.452	6.363
Griechenland	10.76	4.970	3.407
Ungarn	9.976	6.176	3.358
Schweden	9.088	8.397	4.403
Schweiz	7.639	6.152	2.655
Finnland	5.259	4.480	2.023
Norwegen	4.691	4.431	2.466
Island	0.311	0.301	0.207

Welche Länder haben prozentual den höchsten bzw. den niedrigsten Anteil an Internet-Nutzern? Welchen Wert hat Deutschland? Welche Länder liegen bei den Facebook-Nutzern vorn und hinten und wie sieht es in Deutschland aus? In welchem Land haben die meisten Internet-Nutzer ein Facebook-Profil? Was ist der durchschnittliche Anteil an Internet-Nutzern? Der durchschnittliche Anteil an Internet-Nutzern mit Facebook-Profil?

11 Logisches indizieren

11.1 Formulierung von Bedingungen mit *Vergleichsoperatoren*

MATLAB erlaubt den Vergleich von Variablen und Zahlen zur Prüfung numerischer Aussagen. MATLAB gibt eine `1` (`true`) zurück, wenn die mathematische Aussage richtig ist und eine `0` (`false`), wenn die Aussage nicht stimmt. Folgende *Vergleichsoperatoren* stehen zur Verfügung: <, >, <=, >=, == und ∼=. **Hinweis:** Das Zeichen „=“ ist der Zuweisungsoperator . Um das mathematische „ist gleich“ zu verwenden ist das Zeichen „==“ erforderlich.

Aufgabe 11.1

Definiere die Variablen `a = 56; b = 43; z = 139.4; w = -40.4;` und führe die folgenden Vergleiche durch.

```
>> b > a

>> z < a

>> 500 >= a

>> z + w == a + b

>> 50 ~= a + b
```

Vergleichsoperatoren können auch verwendet werden, um Vektoren gleicher Größe miteinander oder mit einem skalaren Wert zu vergleichen. Dabei wird jedes Element eines Vektors mit dem Element in der gleichen Position in dem zweiten Vetktor bzw. dem Skalar einzeln verglichen. Das Ergebnis ist ein *logischer Vektor* gleicher Größe.

Aufgabe 11.2

Gib folgende Befehle ein und interpretiere das Ergebnis.

```
>> A = [1:5];

>> B = [5:-1:1];
```

```
>> A == B

>> A ~= B

>> A < B

>> A > B

>> A >= B

>> B > 3.7

>> L = A < B

>> whos L
```

Manchmal ist es interessant, ob es ein Element des Vektors gibt, welches die angegebene Bedingung erfüllt, oder ob alle Elemente die Relation erfüllen. Schaue dir dazu in der Hilfe die Befehle `any()` und `all()` an.

Aufgabe 11.3

(a) Der folgende Vektor enthält fiktive Daten eines Reaktionszeit-Experiments. In dem Vektor stehen die mittleren Reaktionszeiten (in ms) der Versuchspersonen:
`RTData = [390 347 866 549 589 641 777 702];`
Wie viele Versuchspersonen hatten eine mittlere Reaktionszeit unter 600 ms? **Hinweis:** Benutze logische Vektoren und den Befehl `sum`.
(b) Im folgenden Vektor stehen die Prozent richtig beantworteten Versuche der selben Versuchspersonen wie in (a):
`PCData = [.45 .32 .98 .67 .72 .50 .77 .68]`
Wie viele Versuchspersonen waren besser als 60 %?

11.2 Logisches indizieren

Logische Vektoren können genutzt werden, um Vektoren gleicher Länge zu indizieren. Ausgegeben wird ein Vektor mit den Werten des Urspungsvektors, an deren Stelle der logische Vektor den Wert 1 (*true*) hatte. Die Werte des Ursprungsvektors, an deren Stelle der logische Vektor den Wert 0 (*false*) hatte, werden also nicht benutzt.

Beispiel

```
>> A = [1:5];

>> B = [5:-1:1];

>> L = A < B
```

```
>> A(L)

ans =

     1     2

>> L = A > B

>> A(L)

ans =

     4     5
```

Aufgabe 11.4

Betrachte die Daten des Reaktionszeit-Experiments in Aufgabe 11.3. Gib mit Hilfe von logischen Vektoren die Reaktionszeiten aus, die größer als 600 ms sind. Gib die Performance der Versuchspersonen aus, die beser als 60 % waren.

11.3 Prüfung von logischen Ausdrücken mit *logischen Operatoren*

Vergleichsoperatoren können durch *logische Operatoren* verknüpft werden. Auf diese Weise könnten logische Ausdrücke überprüft werden. Logische Operatoren: und (`&`), oder (`|`), exklusives oder (`xor()`).

Bed. 1	Bed. 2	Bed. 1 `&` Bed. 2	Bed. 1 `|` Bed. 2	`xor(`Bed. 1`,` Bed. 2`)`
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Aufgabe 11.5

Gib folgende Befehle ein und interpretiere das Ergebnis.

```
>> K = [1:4];

>> S = (K>1) & (K<4)

>> T = (K==2) | (K>3)

>> V = (K==2) | (K<3)

>> W = xor((K==2), (K<3))
```

Aufgabe 11.6

Betrachte die Daten des Reaktionszeit-Experiments in Aufgabe 11.3. Wieviele Teilnehmer hatten eine Reaktionszeit unter 600 ms und waren gleichzeitig besser als 60 %? Was für Reaktionszeiten und Performancewerte hatten diese Versuchspersonen?

12 Matrix

Wie auch bei Vektoren muss man sich eine Matrix so gut wie nie als echte mathematische Matrix vorstellen. Wir verwenden sie ganz einfach, um mehrere Zahlen in einer Art Tabelle zusammen zu fassen.

12.1 Größe einer Matrix

Die Matrix

```
A =

1 2 3

4 5 6
```

zum Beispiel enthält 6 Elemente in 2 Zeilen und 3 Spalten. Wir sprechen von einer 2×3 Matrix. **Achtung: Man erwähnt immer zuerst die Anzahl der Zeilen, dann der Spalten!** Das ist für alle Operationen mit Matrizen besonders wichtig, und eine Verwechslung von Zeilen und Spalten ist eine der häufigsten Fehlerquellen.

Eigentlich ist also alles eine Matrix: eine einzelne Zahl ist eine 1×1 Matrix, ein Zeilenvektor mit 5 Elementen eine 1×5 Matrix, ein Spaltenvektor eine 5×1 ein Matrix. Es ist jedoch einfacher, diese Matrizen weiterhin Zahl oder Vektor zu nennen, und erst ab mindestens 2 Dimensionen von einer Matrix zu sprechen.

12.2 Konstruktion einer Matrix

Um eine Matrix bei MATLAB einzugeben, werden die einzelnen Zeilen durch ; abgetrennt:

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

A =

1 2 3
```

```
4 5 6

7 8 9
```

Wie bei Vektoren muss man auch bei Matrizen nicht immer alle Elemente per Hand eingeben, sondern kann den Doppelpunkt-Operator verwenden.

```
>>  A= [1:3;4:6;7:9]

A =

1 2 3

4 5 6

7 8 9
```

Vorsicht Fehler: Alle Zeilen und alle Spalten müssen genau gleich viele Elemente haben, damit sie unter- bzw. nebeneinander passen. Die Eingabe:

```
>>  B= [1:3;4:7;8:9]
```

führt also zu einem Fehler. Probiere es aus!

12.3 Indizieren einer Matrix

Auf die einzelnen Elemente der Matrix kann zugegriffen werden, indem man den Eintrag an einer bestimmten Position abruft, genau wie bei Vektoren. Da wir eine zweidimensionale Matrix haben, brauchen wir jetzt zwei Indizes, durch ein Komma getrennt, um eine Position in der Matrix genau zu beschreiben. In diesem Beispiel ist die Position der Zahl 7 durch ihren Zeilenplatz 3 und ihren Spaltenplatz 1 gekennzeichnet. Man nennt immer zuerst die Zeile, dann die Spalte des Elements, das man ausgeben will!

```
>> A(3,1)

ans = 7
```

Um die gesamte Spalte auszuwählen, in der die 7 steht, kann man wieder den Doppelpunkt-Operator benutzen:

```
>> A(:,1)

ans =

1

4

7
```

Auch alle Kombinationen des Doppelpunkt-Operators, die wir schon bei Vektoren gesehen haben, können benutzt werden, z.B.

```
>> A(1:2,2:3)

ans =

2 3

5 6
```

Natürlich kann man auch wieder beliebige einzelne Elemente oder logische Matrizen der gleichen Grvoße als Indizes benutzen.

So wie auch bei Vektoren können wir nun den durch die Indizes spezifizierten Elementen etwas zuweisen:

```
>> A(2,3) = 100

A =

1 2 3

4 5 100

7 8 9


>> A(:,3) = 1:3

A =

1 2 1

4 5 2

7 8 3
```

Vorsicht Fehler: Dummerweise kann man die Elemente einer Matrix auch nacheinander mit nur einem Index ansprechen, z. B. A(5). Das macht aber selten Sinn, deshalb gewöhnen wir uns gleich an, immer die Zeile und Spalte des gewünschten Elements anzugeben. Es ist allerdings wichtig zu wissen: Falls man doch mal einen Index vergisst, gibt MATLAB keine Fehlermeldung aus, sondern liefert einen Wert der Matrix aus, aber selten den, den man eigentlich gemeint hat! Ein schwierig zu findender Fehler.

12.4 Löschen und Verbinden von Matrizen

Soll eine ganze Zeile oder Spalte gelöscht werden, so setze `[]`. Soll z. B. die zweite Zeile der Matrix A gelöscht werden:

```
>> A(2,:) = []

ans =

1 2 3

7 8 9
```

Man kann 2 Matrizen oder mehr zu einer größeren Matrix zusammenzusetzen, entweder in der 1. Dimension (also untereinander) oder in der 2. Dimension (also nebeneinander). Auch hier muss wieder alles passen. Für eine vertikale Konkatenation (`vertcat` in den MATLAB Fehlermeldungen) muss die Anzahl der Spalten in beiden Matrizen gleich sein, für eine horizontale Konkatenation (`horzcat`) die Anzahl der Zeilen. Ansonsten verbindet man Matrizen, wie man auch einzelne Elemente verbindet: Vertikal mit `;` getrennt, horizontal mit Komma oder einfach Leerzeichen:

```
>> F = [10 11 12; 1 2 3];

>> G = [13 14 15; 4 5 6];

>> V = [F;G]

V =

10 11 12

1 2 3

13 14 15

4 5 6



>> H = [F G]

H =

10 11 12 13 14 15

1 2 3 4 5 6
```

Bevor man Matrizen verbindet, sollte man sich also immer über die Größe der einzelnen Matrizen informieren, um sich zu versichern, dass eine Zusammensetzung möglich ist. Bei größeren Matrizen ist das oft nicht auf den ersten Blick zu sehen. Hierfür gibt es die

Funktion size.

```
>> size(H)

ans = 2 6
```

Das bedeutet, dass die Matrix H aus 2 Zeilen und 6 Spalten besteht. Interessiert man sich nur für die Größe in einer bestimmten Dimension, kann man diese als weiteres Argument an die size-Funktion übergeben. Für die Anzahl der Zeilen heißt es z. B.

```
>> size(H,1)

ans = 2
```

Das gleiche gilt für viele Funktionen, die eine Matrix als Eingabe bekommen (z. B. mean, sum). Fast immer kann man dann auch die gewünschte Dimension angeben, in der die Berechnung durchgeführt werden soll. Funktionen, mit denen man Vektoren erzeugen kann, kann man auch verwenden, um Matrizen bestimmer Größen zu erzeugen (z. B. zeros, ones, rand) . Dazu gibt man zuerst die gewünschte Zeilenanzahl und dann die Spaltenanzahl an (z. B. ones(Zeilen,Spalten)).

12.5 Operationen mit Matrizen

Elemente in Matrizen können mit Hilfe der Operatoren +, -, *, / bzw. ^ addiert, subtrahiert, multipliziert, dividiert bzw. potenziert werden. Die Größen der Matrizen müssen dabei identisch sein, damit klar ist, was von was abgezogen werden soll, etc. Wie auch schon bei Vektoren muss man bei Multiplikation, Division und Pozenzierung den .* bzw ./ bzw .^ Operator verwenden, damit MATLAB die Matrix nicht als mathematische Matrix verwendet und eine echte Matrixmultiplikation durchführt. Wenn einer der Operanden ein Skalar ist, erfolgt die Verknüpfung immer elementweise – hier muss man sich auch keine Sorgen über die Größe machen.

Aufgabe 12.1

(a) Erstelle eine Matrix aus den 3 Vektoren aus der Internet-Aufgabe (Abschnitt 10.7).

Abbildung 4: Die MAGISCHE MATRIX von Albrecht Dürer aus dem Jahr 1514

(b) Gebe die Matrix aus Abbildung 4 in MATLAB ein. Lasse Dir die Elemente der ersten Zeile einzeln und als Zeilenvektor ausgeben.
(c) Summiere alle Elemente der ersten Spalte.
(d) Rufe jede Zeile einzeln auf. Bilde die Summe über alle Zeilen und alle Diagonalelemente (**Tipp:** `diag`).
(e) Finde den größten und kleinsten Eintrag der Dürer-Matrix.
(f) Lasse alle Zahlen der Dürerer-Matrix, die größer als 10 sind, ausgeben.
(g) Verwende `imagesc(`*`Dürer-Matrix`*`)`. Was hat eine Matrix mit Pixel-Bildern zu tun?
(h) Multipliziere die Dürer-Matrix mit sich selbst (Matrixalgebra). Multipliziere sie zudem elementweise. Erkläre den Unterschied.
(i) Erstelle eine Matrix B, die in der ersten Zeile gerade Zahlen von 1 bis 200 enthält und in der zweiten Zeile ungerade. Wie groß ist diese Matrix?

Aufgabe 12.2

Starte die Datei **Schiffeversenken.m** und feuere so viele Schüsse auf die Matrix, bis du die Reihenfolge von Zeilen und Spalten nie wieder verwechselst.

12.6 Einlesen von Daten

Allgemein kann man die Output-Variablen, die im Workspace von MATLAB stehen, als sogenannte mat-Dateien (Endung `.mat`) abspeichern, indem man `save` *`Workspace`* eingibt. So gehen die Resultate nicht verloren und man kann sie jederzeit wieder aufrufen, indem man sie mittels `load` *`Workspace`* wieder lädt. Man kann auch einzelne Variablen oder ein bestimmtes Set von Variablen abspeichern:

```
>> save Variable1
```

Analog werden die abgespeicherten mat-Dateien geladen:

```
>> load Variable1
```

Für mat-Dateien, die *Workspace*-Variablen enthalten, kann man einen Dateinamen angeben.

```
>> save('dateiname.mat','var1','var2','var3')
```

Es werden dann alle angegebenen Variablen unter diesem Namen gespeichert und wieder geladen, d. h. die Datei muss nicht so heißen, wie die darin gespeicherten Variablen.

13 Kontrollstrukturen beim Programmieren

Bisher haben wir MATLAB so kennen gelernt, dass unsere Befehle in einem MATLAB-Skript sequentiell von oben nach unten abgearbeitet werden. Für eine effiziente Formulierung von Algorithmen können Steueranweisungen verwendet werden, die festlegen, in welcher Reihenfolge MATLAB Befehle ausführt. Diese Steueranweisungen bilden die Kontrollstruktur eines Programms. Typische Kontrollstrukturen sind *Verzweigungen* und *Programmierschleifen*.

13.1 Verzweigungen

An einer Verzweigung wird eine Bedingung geprüft, die wahr (*true*) oder falsch (*false*) sein kann. Nur wenn diese Bedingung erfüllt ist, wird die folgende Anweisung ausgeführt. Mit einer solchen Verzweigung kann der Programmfluss (die Abfolge der Ausführung der Befehle) gesteuert werden.

Die einfachste Form einer Verzweigung ist eine einseitig bedingte Anweisung:

```
if Bedingung
    Befehl
    Befehl
    ...
end
```

Die Bedingung ist ein logischer Ausdruck, der ausgewertet wird (s. Kap. 11). Sie hat also immer den Wert `1` (`true`) oder `0` (`false`). Wenn die angegebene Bedingung erfüllt ist, d. h. den Wert `1` (`true`) hat, werden die Befehle im Anweisungsblock ausgeführt, anderenfalls hat die `if`-Anweisung keinen Effekt und das Programm wird direkt mit dem auf den `end`-Befehl folgenden Befehl fortgesetzt.

Beispiel

```
x = 5;
if x ~= 0
    disp('x ist ungleich 0')
end
```

Zweiseitig bedingte Anweisungen ermöglichen, dass abhängig von der Bedingung verschiedene Programmabschnitte ausgeführt werden. Die Auswahl zwischen den Alternativen erfolgt aufgrund einer Bedingung.

```
if Bedingung
    Befehlsblock 1 (ausgeführt wenn Bedingung erfüllt)
else
    Befehlsblock 2 (ausgeführt wenn Bedingung nicht erfüllt)
end
```

Wenn der Wert der Bedingung `1` hat, werden die Befehle aus dem ersten Block ausgeführt, sonst die aus dem zweiten Block. In beiden Fällen wird der Programmablauf nach der Abarbeitung des entsprechenden Befehlblocks mit dem `end`-Befehl folgenden Befehl fortgesetzt.

Beispiel

```
x = 5;
y = 2;
z = 8;
if (y ~= 0) & (x ~= z)
    g = (x - y) / (y * (x - z))
else
    disp('Fehler! Division durch 0!')
end
```

Wenn mehrere, alternative Bedingungen auftreten, können diese separat geprüft werden. Zu jeder Alternative gibt es einen Befehlsblock. Es wird höchstens ein Befehlsblock ausgeführt.

```
if Bedingung 1
    Befehlsblock 1 (ausgeführt wenn Bedingung 1 erfüllt)
elseif Bedingung 2
    Befehlsblock 2 (ausgeführt wenn Bedingung 2 erfüllt)
elseif Bedingung 3
    Befehlsblock 3 (ausgeführt wenn Bedingung 3 erfüllt)
else
```

Befehlsblock 4 (ausgeführt wenn keine der vorangehenden Bedingungen erfüllt ist)
end

Beispiel

```
a = 2;
b = [];
c = 0;
if a < 0
   b = -1
elseif a == 0
   b = 0
elseif c == 0
   b = 1
else
   b = NaN
end
```

Achtung! Die Bedingungen werden von oben nach untengeprüft! Das heißt, wenn die Bedingung a erfüllt ist, wird der entsprechende Block ausgeführt und die if-Schleife verlassen. Es wird nicht mehr geprüft ob andere bedingungen auch erfüllt sind. Im obigen Beispiel: wenn a=0, wird b=0 gesetzt, die Prüfung ob c=0, findet nicht mehr statt.

Bei der Verwendung der `if`-Konstruktion können mehrere voneinander unabhängige Bedingungen ausgewertet werden. Wenn man allerdings die Ausführung bestimmter Befehle von dem Wert einer einzigen Variable abhängig machen möchte, ist oft die *Switch/Case-Konstruktion* günstiger.

`switch` Variable
`case` Fall 1
    Befehlsblock 1
`case` Fall 2
    Befehlsblock 2
`otherwise`
    Befehlsblock 3
`end`

Beispiel

```
x = 2.7; % in cm
unit = 'm'; % gewünschte Ausgabeeinheit
switch unit
    case 'inch'
        y = x / 2.54
    case 'feet'
        y = x / 2.54 / 12
    case 'm'
        y = x / 100
    case 'mm'
        y = x * 10
    case 'cm'
        y = x
    otherwise
        fprintf('Unbekannte Einheit: %s\n',unit)
        y = NaN
end
```

Beispiel

```
t = 20;
switch t
    case 1
        y = 1
    case {2, 3, 4, 20}
        y = 0
end
```

Aufgabe 13.1

Schreibe ein Programm, das eine Zeiteingabe in Millisekunden in eine besser lesbare Einheit umwandelt.

- Fordere den Benutzer auf, eine Zeit in Millisekunden einzugeben.
- Überprüfe, in welchem Wertebereich die Eingabe liegt. Rechne die Zeit entsprechend in Jahre, Tage, Stunden, Minuten oder Sekunden um.
- Gib das Ergebnis aus.

Zum testen kann man die Seite www.unitjuggler.com verwenden.

Aufgabe 13.2

Erstelle ein Programm zur Bestimmung der Anzahl der Tage in einem bestimmten Monat und Jahr.

- Fordere den Benutzer auf, einen Monat und ein Jahr einzugeben.
- Überprüfe, ob es den eingegebenen Monat tatsächlich gibt. Unseren Gregorinischen Kalender gibt es in der Form erst seit 1582. Stelle sicher, dass die eingegebene Jahreszahl nicht kleiner als 1582 ist.
- Nutze die Switch/Case-Konstruktion, um die Anzahl der Tage für bestimmte Monate auszugeben (April, Juni, Sept., Nov.: 30 Tage; Feb.: 28/29 (siehe unten), restliche Monate: 31 Tage).
- Im Februar hängt die Anzahl der Tage davon ab, ob es sich um einen Schaltjahr handelt. Nutze die `if`-Konstruktion, um zu bestimmen, ob das eingegebene Jahr ein Schaltjahr ist. Ein Schaltjahr ist es dann, wenn die Jahreszahl ohne Rest durch 4 teilbar ist. Allerdings sind Säkular-Jahre (Jahre, deren Zahl durch 100 teilbar ist) keine Schaltjahre, es sei denn sie sind ohne Rest durch 400 teilbar. (Nutze den Befehl `rem()`, um zu testen, ob ein Rest bei der Division durch bestimmte Zahlen übrig bleibt.)
- Prüfe, wie viele Tage dein Geburtsmonat nächstes Jahr hat. Prüfe, wie viele Tage der Februar in deinem Geburtsjahr und den Jahren 2000, 2004, 2100, 2200 hatte/haben wird. War dein Geburtsjahr ein Schaltjahr?

13.2 Programmschleifen

Neben den bedingten Anweisungen gibt es in der Programmierung eine weitere wichtige Kontrollanweisung: die Programmschleifen. Darunter versteht man eine Folge von Aktionen oder Teilprogrammen, die physisch nur einmal im Programmcode vorhanden sind, aber mehrmals wiederholt und durchlaufen werden.

13.2.1 For-Schleifen

Eine Schleife besteht grundsätzlich aus einer Anweisung, die den Ein- und Austritt in bzw. aus der Schleife bestimmt, und einem Schleifenkörper (Schleifenrumpf), der die zu wiederholenden Befehle und Anweisungen enthält.

Mit einer `for`-Schleife kann man alle Operationen im Schleifenrumpf so oft wiederholen, wie man vorher festgelegt hat.
`for`-Schleifen haben in MATLAB die Form:

```
for Zählvariable = Bereich
    ...
    Schleifenrumpf
    ...
end
```

Diese `for`-Schleife erzeugt gerade Zahlen:

```
for i = 1:6
    a(i) = 2 * i
end a
```

```
a = 2 4 6 8 10 12
```

Eine `for`-Schleife bildet eine Methode eine Summe von Zahlen zu bilden. Hier ein Beispiel für eine Summenberechnung:

```
summe = 0;
n = 10;
for ind = 1:n
    summe = summe + ind*2
end
summe
```

```
summe = 110
```

Dieses Beispiel scheint vielleicht einfach, aber solche Schleifen, die mit einem Ergebnis aus dem vorherigen Schleifendurchlauf weiterrechnen, sind besonders schwierig zu programmieren (und besonders fehleranfällig). Hier ist es wichtig, bevor man überhaupt etwas hinschreibt, sich zuerst mit Zettel und Stift folgende Dinge genau zu überlegen:

- Angenommen bis zu einem beliebigen Zeitpunkt stimmt schon alles, was soll genau beim nächsten Schritt passieren? (z.B. das Zwischenergebnis soll verdoppelt werden). Welche Variablen werden dafür benutzt? (in diesem Besipiel nur das Zwischenergebnis) Benutzt man die Zählvariable (in diesem Fall nicht). Schreibe diese Anweisung so auf, dass du sie nachher in den Schleifenrumpf kopieren kannst. Schreibe keine unnötigen Anweisungen in den Schleifenrumpf, die eigentlich nur einmal ausgeführt werden müssen!

- Initialisierung: Welche Werte musst du vor dem ersten Schleifendurchlauf festlegen? (da wir schon festgelegt haben, dass wir bei jedem Schritt (also auch beim ersten Schritt!) das Zwischenergebnis verdoppeln werden, muss dieses vor dem ersten Schleifendurchlauf, also vor der for-Schleife auf einen sinnvollen Wert festgesetzt werden, z.B. 1 (warum ist 0 in diesem Fall ungeeignet?))

- Zählvariable: Wie oft muss die Schleife genau durchlaufen werden? Überleg es dir ganz genau, es ist sehr leicht, die Schleife entweder genau einmal zu viel oder einmal zu wenig zu durchlaufen. Ist deine Zählvariable geeignet (wenn du z.B. im Rumpf die Zählvariable als Index benutzen willst, sollte es sich um ganze, positive Zahlen handeln).

Danach kannst du die Schleife ganz leicht aufschreiben.

Da `for`-Schleifen gar nicht gut genug verstanden werden können werden wir jetzt viele Beispiele vorgeben...

Beispiel 1

Wir berechnen die Zahl 12 Fakultät. Die Hilfsvariable `fac` wird das Ergebnis enthalten.

```
n = 12;
fac = 1;
for i = 2:n
    fac = i*fac;
end
```

Anfangs- und Endwert sowie Schrittweite werden bei der `for`-Schleife vor dem ersten Schleifendurchlauf bestimmt und können beim Schleifendurchlauf nicht mehr verändert werden. Im folgenden Beispiel ist der Anfangswert 1, der Endwert 10 und die Schrittweite 2. Die Wertzuweisungen im Schleifenkörper wirken sich auf diese Werte nicht mehr aus. Die Klammerung der drei Ausgabevariablen bewirkt, dass diese hier in nur einer Zeile ausgegeben werden.

Beispiel 2

```
step = 2;
fine = 10;
for tt = step-1:step:fine
   tt = 2*tt;
   step = 2*step;
   fine = fine-1;
   [ tt step fine ]
end
```

```
ans = 2 4 9
ans = 6 8 8
ans = 10 16 7
ans = 14 32 6
ans = 18 64 5
```

Die Ausdrücke, mit denen Anfangs-, Endwert und Schrittweite angegeben werden, müssen vor dem ersten Schleifendurchlauf bereits definierte Werte aufweisen. Das ergibt sich schon daraus, dass bereits vor dem ersten Durchlauf festgestellt werden muss, ob ein solcher überhaupt stattfindet. Bei der Angabe des Schleifenbereiches unterscheidet MATLAB zwischen undefinierten Namen und solchen mit undefinierten bzw. unbrauchbaren Werten. Ersteres führt zu einer Fehlermeldung, letzteres in der Regel dazu, dass die Schleife nicht durchlaufen wird. Probiere die folgenden Beispiele aus (und verwende sie nie in deinem eigenen Code):

Beispiel 3

```
m = [];
for ii = 0:m
   ii
end
```

```
clear all
```

```
for ii=0:m
   ii
end
```

```
m = NaN;
```

```
for ii=0:m
    ii
end
```

```
for ii = 0:Inf
    ii
end
```

For-Schleifen sind ein beliebtes Werkzeug beim Programmieren in MATLAB. Man sollte sich aber vor Augen führen, dass Programme dadurch langsam werden können. Deshalb sollte man Schleifen nur verwenden, wenn man sie wirklich braucht. Um ein Gefühl dafür zu bekommen, wie schnell bzw. langsam for-Schleifen sein können, berechnen wir im Folgenden Laufzeiten von Programmen und vergleichen diese mit Laufzeiten von Programmen, die das gleiche leisten, in denen jedoch keine for-Schleife verwendet wurde. Wir geben ein Beispiel an, wie man Laufzeiten messen kann, wenn man die übrigen Umstände günstig bzw. besonders ungünstig wählt.

Beispiel 4

Wir wollen die ersten N Quadratzahlen ausrechnen und in einem Vektor speichern. Nachfolgend werden drei unterschiedliche Implementierungen angegeben, deren Laufzeit mit Hilfe der Befehle tic (Start einer Stoppuhr) und toc (Anhalten der Stoppuhr) gemessen wird: Implementierung mit einer for-Schleife und schrittweiser Vergrößerung des Vektors:

```
N = 50000;

tic;

squares1 = [];

for i = 1:N

squares1 = [squares1; i^2];

end

toc

Elapsed time is 16.592336 seconds.
```

Implementierung mit einer `for`-Schleife und Präallokation des Vektors:

```
N = 50000;

tic;

squares2 = zeros(N,1);

for i = 1:N

squares2(i) = i^2;

end

toc

Elapsed time is 0.001056 seconds.
```

Im Gegensatz zur ersten Implementierung wird der Vektor hier am Anfang angelegt und ändert dann nicht mehr seine Grösse – das spart enorm viel Zeit.

Implementierung ohne eine `for`-Schleife:

```
N = 50000;

tic;

squares3 = 1:N;

squares3 = squares3.^2;

toc

Elapsed time is 0.000430 seconds.
```

Wenn man gar keine Schleife verwendet sondern mit Vektoren rechnet, spart man noch mehr Zeit. Bei großen Datenmengen kann das ein enormer Unterschied sein!

Aufgabe 13.3

(a) Bilde einen Vektor der nur gerade Zahlen enthält (von 34 bis 104).
(b) Nach der Legende soll der Erfinder des Schachspiels von seinem König eine große Belohnung versprochen bekommen haben. Der Erfinder wünscht sich für das erste Feld des Schachbretts ein Reiskorn, für das zweite zwei, für das dritte vier, usw., in Verdoppelungen bis zum 64. Feld. Der König hält dies für eine kleine Belohnung, tatsächlich treibt ihn seine Zusage in den Ruin. Wieviele Reiskörner schuldet der König dem Erfinder und wieviele Tonnen Reis sind das, wenn man 0.05 g pro Korn annimmt?
(c) Addiere die quadrierten Zahlen zwischen 15 und 55, also 15 zum Quadrat+ 16 zum Quadrat...

13.2.2 Häufige Fehler bei Schleifen und Matrizen

In dem unten stehenden Programm wird eine Matrix M erstellt, in die 100 zehnstellige Zahlenkombinationen geschrieben werden. Leider befindet sich ein Fehler in diesem Programm. Finde ihn mittels des Debugging-Werkzeugs.

```
clear all
clc
for j = 1:100
    M = [];
    r = [rand(1,10)];
    M(j,:) = [r];
end
M
```

- `Error using horzcat: matrix dimensions do not agree`: Dieser Fehler tritt häufig auf, wenn man die Größe einer Matrix innerhalb einer Schleife ändert. Um Matrizen zu verbinden müssen sie in der richtigen Dimension die selbe Größe haben.
- `index exceeds Matrix dimensions`: Der Index, den ihr benutzt, ist größer als die Matrix, auf die zugegriffen wird. Auch das passiert häufig in Schleifen, die einmal zu oft durchlaufen werden.
- `Subscript indices must either be real positive integers or logicals`: Ihr versucht 0, eine negative Zahl, oder Kommazahl als Index zu verwenden. Das geht natürlich nicht. Auch das passiert häufig in Schleifen, wenn man die Zählvariable als Index nimmt, obwohl es keine ganze Zahl ist.

Der Zugriff auf leere Matrizen liefert immer einen Fehler, da die Dimension gleich Null ist. Eine häufige Möglichkeit sich so seinen Fehler einzuhandeln, ist die `find`-Funktion. Falls der leere Output aus der `find`-Funktion benutzt werden soll eine andere Funktion zu indizieren, wird das Ergebnis auch leer sein.

Beispiel

```
>> x = pi*(1:4)

x = 3.1416 6. 2832 9.4249 12.5664

>> i = find (x>20)

i = []
```

```
>> y = 2*x(i)

y =[]
```

Falls erwartet wird, dass y Einträge hat, dann ist es sehr wahrscheinlich, dass ein Fehler auftritt. Wenn man Operationen durchführen möchte, die mÃ¶glicherweise leere Matrizen zurückgeben, kann man die Funktion `isempty` verwenden, um die leere Matrix aufzuspühren und eine default-Operation ausfuhren (z. B. Falls die Matrix leer ist, setzt man ihn defaultmäßig auf 1, sodass das Programm keinen Fehler produziert).

13.2.3 While-Schleifen

Die `while`-Schleife wiederholt einen Anweisungsblock beliebig oft, solange bis eine logische Abbruchbestimmung erfüllt ist. Ist ein solches Abbruchskriterium nicht vorhanden oder fehlerhaft, kommt es zu einer so genannten Endlosschleife, einem typischen Programmfehler: Die Folge von Bedingungen wird unendlich oft wiederholt und findet keinen Abschluss. Endlosschleifen können abgebrochen werden mit der Tastenkombination **Ctrl+c**.

Bei diesem Schleifentyp wird vor jeder Wiederholung die Bedingung überprüft. Solange diese Bedingung wahr (ungleich 0) ist, werden die Anweisungen im Schleifenkörper ausgeführt. Ist die Bedingung von Beginn an nicht erfüllt, so wird der Schleifenkörper nicht durchlaufen und kein einziges Mal ausgeführt. Das Schlüsselwort, das eine Schleife mit vorangestellter Bedinungsprüfung definiert, ist `while`. Dies bedeutet soviel, wie „solange Bedingung wahr ist, wiederhole...“. Eine `while`-Schleife kann oft eine `for`-Schleife ersetzen, wenn man im Schleifenkörper einen Zähler einbaut. Auch am Ende des Schleifenkörpers einer `while`-Schleife muss unbedingt ein `end` stehen.

```
while Bedingung
    Befehl
    Befehl
    ...
end
```

Beispiel: Verdoppeln einer Zahl bis eine Obergrenze erreicht ist:

```
i = 1;
while i<1000
    i = i*2
end
```

Beispiel: Eine Zufallszahl wird erzeugt, sooft wie der Benutzer `'y'` eingibt.

```
reply='y';
while reply=='y'
    rand(1,1)
    reply = input('Do you want more? y/n [y]: ', 's');
end
```

Beispiel: Wie oft gibt das Programm „Hello World“ aus?

```
n = 10;
while n > 0
    disp('Hello World')
    n = n - 1;
end
```

Beispiel: Und hier?

```
n = 1;
while n > 0
    disp('Hello World')
    n = n + 1;
end
```

Aufgabe 13.4

(a) Schreibe ein Programm, das eine natürliche Zahl n abfragt und die Fakultät dieser Zahl berechnet. Nutze dazu die `while`-Schleife.
(b) `While`-Schleifen können oft durch `for`-Schleifen ersetzt werden, wenn die Anzahl der gewünschten Wiederholungen von vorn herein bekannt ist. Führe die gleiche Berechnung mit Hilfe einer `for`-Schleife durch.
(c) Überprüfe deine Berechnung mit den in MATLAB vorhandenen Funktion `factorial()` und `prod()`.
(d) Welches der Verfahren ist das schnellste bei der Ausführung?

13.2.4 Ablaufunterbrechung der Schleifen

In manchen Fällen ist ein vorzeitiger Abbruch oder Verlassen einer `for`- oder `while`-Schleife sinnvoll. Mit dem Befehl `continue` wird innerhalb einer `for`- oder `while`-Schleife

sofort zum nächsten Iterationsschritt gesprungen, alle innerhalb der aktuellen Schleife folgenden Befehle werden übergangen.

Beispiel

```
T=[2 6 5 8 9 3 6 3 9 1 1 9];
summe=0;
for i=1:length(T)
    if T(i)==9 | T(i)==3
        continue
    end
    summe=summe+T(i);
end
    summe = 29;
```

Der Befehl `break` wirkt stärker: Er bricht die gerade aktuelle Schleife ab. Gleiches gilt, falls `break` innerhalb einer Verzweigungsstruktur (`if, switch`) steht. Wird `break` in einem MATLAB-Skript oder einer MATLAB-Funktion auï¿½erhalb einer Schleife aufgerufen, so wird das Skript bzw. die Funktion an dieser Stelle abgebrochen. Bei verschachtelten Schleifen beendet `break` nur die innerste Schleife.

Beispiel

```
T=[2 6 5 8 9 3 6 3 9 1 1 9];
summe=0;
for i=1:length(T)
    if T(i)==9 | T(i)==3
        break
    end
    summe=summe+T(i);
end
    summe = 21;
```

Aufgabe 13.5

Du hast die Aufgabe, jeweils 30 männliche und 30 weibliche Kohlmeisen, Amseln und Rotkehlchen zu fangen, zu beringen und ihr Gewicht zu bestimmen. Schreibe ein Programm, das diese Arbeit für dich erledigt und eine Tabelle zurückgibt, die für die 60 Tiere jeweils die Nummer des Ringes (aufsteigend von 1 bis 60 nach Fangreihenfolge), die Vogelart, das Geschlecht und das Gewicht enthält. Nutze dabei jeweils einen Zahlencode (z. B. Kohlmeise=1, Amsel=2, Rotkehlchen=3), der im Kommentar des Programms genau dokumentiert ist (nutze die Switch/Case-Anweisung). Die Vögel fängt man mit der Funktion **vogelfang.m**, die mit `[Art,Weibchen,Gewicht]=vogelfang` aufgerufen wird und zufällige Vögel ins Netz flattern lässt.

Rückgabewerte: Art ist eine Zeichenkette, Weibchen ist ein Wahrheitswert zur Angabe des Geschlechts, Gewicht ist eine Zahl (in g) Wenn man einen Vogel fängt, den man für seine Statistik nicht braucht (d. h. falls man schon genügend weibliche bzw. männliche Vögel gefangen hat, dann frei lassen, ohne seine Daten aufzunehmen.

Aufgabe 13.6

Die 100 Mitarbeiter der Firma X kommen nur mit eigenem Passwort in ihre Firma. Schreibe ein Programm, welches den Mitarbeiter nach seiner Mitarbeiternummer und seinem Passwort fragt und mit dem zu diesem Mitarbeiter zugehörigen Passwort aus einer Liste vergleicht. Die Mitarbeiternummer soll sooft eingegeben werden, bis sie tatsächlich im Bereich 1–100 liegt. Dabei sollte es möglich sein das Passwort 3 mal falsch einzugeben. Gib dem Benutzer einen Hinweis, falls das Passwort die falsche Länge hat.
(a) Fertige eine Liste für alle 100 Mitarbeiter und ihren zugehörigen Passwörtern an. Die Passwörter sollten 6-Stellig sein und aus 9 zufälligen, sich nicht wiederholenden Ziffern bestehen. **Tipp:** nutze `randperm()`.
(b) Schreibe das Passwort-Programm.

13.3 Geschachtelte Schleifen

Schleifen sowie Verzweigungen können ineinander verschachtelt werden, sodass sich eine Kontrollstruktur im Befehlsblock einer anderen befindet.

Hier ist ein Beispiel, welches zeigt, dass es möglich ist mehrere `for`-Schleifen zu schachteln und eine komplexere Matrix erzeugen. Wir setzen die Elemente der i-ten Zeile und der j-ten Spalte der Matrix A auf die Werte i+j:

```
for zeile = 1:6
    for spalte = 1:3
        A(zeile,spalte) = zeile + spalte;
    end
end
A
```

```
A = 2 3 4
3 4 5
4 5 6
5 6 7
6 7 8
7 8 9
```

Beispiel

Es soll überprüft werden, welche der Zahlen zwischen 3 und 7 Primzahlen sind. Diese werden in der äußeren Schleife durchlaufen:

```
for m=3:7
    for n=2:m-1
        if mod(m,n)~=0
            continue
        end
        fprintf('%d ist keine Primzahl!\n', m)
        break
    end
    if n==m-1
        fprintf('%d ist EINE Primzahl!\n', m)
    end
end
```

In der inneren Schleife wird die gerade zu überprüfende Zahl m jeweils durch alle Zahlen von 2 bis m-1 geteilt und überprüft, ob der Rest `mod(m,n)` ungleich 0 ist. Falls nicht – also der Rest gleich 0 – so ist m durch n teilbar und keine Primzahl, der `continue`-Befehl wird nicht ausgeführt. Andernfalls – also m nicht durch n teilbar – kommt der `continue`-Befehl zur Wirkung, der folgende `fprintf` und der `break`-Befehl werden übersprungen und es wird sofort das nächst höhere n ausprobiert. Ist m nicht durch Zahlen von 2 bis m-1 teilbar, also nur durch 1 und durch sich selber, so ist es eine Primzahl. Die Überprüfung `n==m-1` ist notwendig für den Fall, wenn m keine Primzahl ist, und die innere Schleife mittels `break` verlassen wurde und dann alle nachfolgenden Befehle der äußeren Schleife abgearbeitet werden. Als Ausgabe in MATLAB ergibt sich:

```
3 ist EINE Primzahl!
4 ist keine Primzahl!
5 ist EINE Primzahl!
6 ist keine Primzahl!
7 ist EINE Primzahl!
```

Aufgabe 13.7

In ihrem Gemischtwarenladen verkauft Milo folgende Artikel:
`things = ['dog';'cat';'car';'box'];` jeweils in drei verschiedenen Farben:
`colors = ['blue';'grey';'pink'];` Hilf ihr, mit einer verschachtelten `for`-Schleife, die 12 Schilder für ihr Schaufenster automatisch zu drucken (also 'pink cat' etc).

Aufgabe 13.8

Lade die Datei **sinus.mat**. Sie enthält eine 2×201 Matrix. In der ersten Zeile sind die x-Koordinaten und in der zweiten Zeile die y-Koordinaten einer mit Rauschen versehenen Sinusfunktion. Schreibe ein Programm, das die Funktion glättet (**Hinweis:** Mittele jeweils 5 aufeinanderfolgende Funktionswerte).

14 Plot

14.1 Plotting

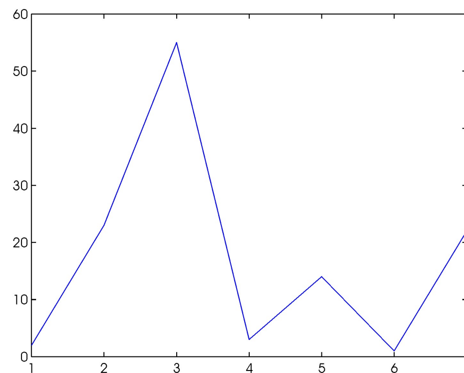
Die relativ einfache Visualisierung von Daten ist ein weiterer Pluspunkt von MATLAB. Die grundlegendste Funktion zur Darstellung von Daten lautet:

```
>>  plot(Daten)
```

Man kann einzelne Punkte, Vektoren und Matrizen plotten. Wenn z. B. in einem Vektor die Höhe eines Insekts relativ zur Ebene gespeichert ist, kann man die Daten plotten und erhält so den rekonstruierten Höhenweg.

Bei MATLAB erhält man den Plot wie folgt:

```
>>  weg = [ 2 23 55 3 14 1 22 ];

>>  plot(weg)
```



Natürlich können wir die Darstellung verschönern. Klicke auf **View→Property Editor** um Eigenschaften der Abbildung anzeigen zu lassen. Jetzt kannst du auf einzelne Elemente klicken (z. B. die Linie) und ihre Farbe, Größe, etc., ändern. Probier ein bisschen rum bis dir dein Plot gefällt. Jetzt solltest du noch auf die Achsen klicken, damit sich das Menü ändert. Hier kannst du die Achsen beschriften und der Abbildung einen Titel geben. Speichere die Grafik ab, wenn sie fertig ist.

Plötzlich erhältst du Daten von einem zweiten Insekt, und möchtest es gerne mit deinem ersten vergleichen. Plotte auch diese Daten und sorge dafür, dass sie genau so aussehen, wie deine erste Abbildung.

```
>> weg2 = [ 1 4 17 5 3 1 ]
```

Schön wäre es, wenn wir das Aussehen des Plots gleich über den `plot`-Befehl kontrollieren könnten, dann müssen wir nicht so viel klicken. Einen Überblick über die Eigenschaften einer Abbildung kann man sich mit der Funktion `get` verschaffen. Dazu weist man dem Plot einer Variable zu.

```
>> h = plot(1:5,1:5);

>> get(h)

    DisplayName: ''
    Annotation: [1x1 hg.Annotation]
    Color: [0 0 1]
    LineStyle: '-'
    LineWidth: 0.5000
    Marker: 'none'
    MarkerSize: 6
    MarkerEdgeColor: 'auto'
    MarkerFaceColor: 'none'
    XData: [1 2 3 4 5]
    YData: [1 2 3 4 5]
    ZData: [1x0 double]
    BeingDeleted: 'off'
    ButtonDownFcn: []
    Children: [0x1 double]
    Clipping: 'on'
    CreateFcn: []
    DeleteFcn: []
    BusyAction: 'queue'
    HandleVisibility: 'on'
    HitTest: 'on'
    Interruptible: 'on'
    Selected: 'off'
    SelectionHighlight: 'on'
    Tag: ''
    Type: 'line'
    UIContextMenu: []
    UserData: []
    Visible: 'on'
    Parent: 175.0018
    XDataMode: 'auto'
    XDataSource: ''
    YDataSource: ''
    ZDataSource: ''
```

Hierbei interessieren uns hauptsächlich die Eigenschaften: `Color, LineStyle, LineWidth, Marker, MarkerSize, MarkerEdgeColor, MarkerFaceColor`, die wir bereits im Menü verändert haben. Wir können diese Eigenschaften auch direkt im `plot`-Befehl kontrollieren:

```
>>  plot(weg,'Color','r','LineWidth',3,'LineStyle',':')
```

Man gibt also immer zuerst den Namen der Eigenschaft an, dann den Wert, mit einem Komma getrennt. Die Reihenfolge der Eigenschaften spielt keine Rolle. Für die Farbe kann man entweder einen Vektor mit 3 Elementen (zwischen 0 und 1) als RGB-Werte angeben, oder man nutzt eine der vordefinierten Farben, die in der folgenden Tabelle zusammen gefasst sind. Sie enthält auch die Optionen für die `Marker`- und `Linestyle`-Eigenschaft.

Symbol	**Farbe**	**Symbol**	**Marker**	**Symbole**	**Linestyle**
`b`	blau	`.`	Punkt	`-`	durchgezogene Linie
`g`	grün	`o`	Kreis	`:`	gepunktete Linie
`r`	rot	`x`	Kreuz	`-.`	Strich-Punkt Linie
`c`	cyan	`+`	Plus	`--`	gestrichelte Linie
`m`	magenta	`*`	Asterisk		
`y`	gelb	`s`	Quadrat		
`k`	schwarz	`d`	Diamand		
`w`	weiß	`v`	Dreieck (nach unten)		
		∧	Dreieck (nach oben)		
		<	Dreieck (nach links)		
		>	Dreieck (nach rechts)		
		`p`	Pentagramm		
		`h`	Hexagramm		

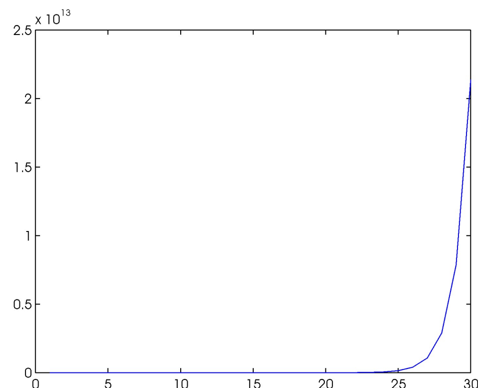
Die Eigenschaften `LineStyle, Marker` und `Color` kann man auch über eine Abkürzung festlegen. Hierzu fasst man die gewünschten Werte dieser Eigenschaften (oder nur ein oder zwei davon), z. B. `'r:*'`, in einem String zusammen, und schreibt diesen **direkt** hinter die zu plottenden Daten. Dahinter kann man dann weitere Eigenschaften festlegen:

```
>>  x =1.5:0.01:4.5

>>  plot([sin(x) -0.4 0.4],[cos(x) 0 0.2],'rx','MarkerSize',40)
```

Nicht nur Vektoren, sondern auch Funktionen und somit deren Verläufe können dargestellt werden. Bleiben wir bei den Insekten. Jede Insektenpopulation zeigt in einem unbegrenzten Lebensraum ein ungebremstes oder exponentielles Wachstum. Idealtypisch verdoppelt sich in konstanten Zeiträumen die Populationsgröße. Die Länge des Zeitraums bestimmt die Wachstumsrate r. Dieses Phänomen lässt sich durch die Funktion $f(x) = re^t$ beschreiben, wobei t die Zeit darstellt. Die Wachstumskonstante r sei 2. Startet man z. B. beim Zeitpunkt 0 und beobachtet die Zunahme der Population täglich

über einen Zeitraum von 30 Tagen, so kann man das Wachstum grafisch darstellen:

```
>> t = 1:30;
>> r = 2;
>> plot(t,r*exp(t));
```



Wenn sich nun die Wachstumsrate ändert und man sich diese Änderung in einer weiteren Abbildung anschauen möchte, ohne die Vorherige zu überschreiben, kann man mit dem Befehl

```
>> figure
```

eine zweite Abbildung aufrufen. Möchte man die Veränderung in die gleiche Abbildung plotten, so gibt man

```
>> hold on
```

ein.

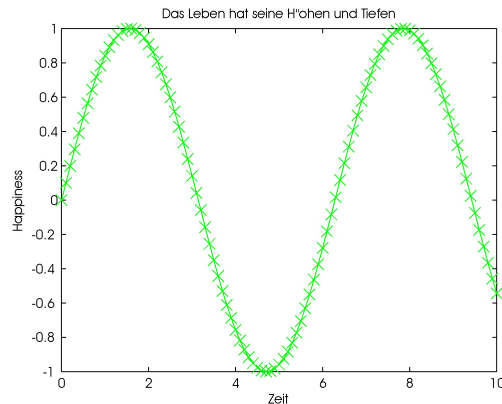
Um zwei oder mehrere Plots in einer Abbildung besser unterscheiden zu können, sollte man z. B. verschiedene Farben benutzen. Mit dem Befehl `close` kann man die aktuelle `figure` schließen. `close all` schließt alle Abbildungen.

14.2 Labelling

Abbildungen bleiben unverständlich solange man nicht ihre Achsenbeschriftungen kennt. Dazu gibt es in MATLAB die Befehle `xlabel` zur x-Achsenbeschriftung, `ylabel` zur y-Achsenbeschriegtung und `title`, um der Grafik eine Überschrift zu verleihen, z. B.

```
>> x = [0:0.1:10];
>> y = sin(x);
>> plot(x,y,'g-x','MarkerSize',12);
>> xlabel('Zeit')
>> ylabel('Happiness')
```

```
>>  title('Das Leben hat seine Höhen und Tiefen')
```



Zudem kann eine Legende zu einer Abbildung hinzugefügt werden (`legend`).

Beispiel

Es sollen die Lernkurven von 4 Versuchspersonen abgebildet werden. Die Versuchspersonen mussten sich Objekte in einer bestimmten Reihenfolge merken und diese dann wieder aufsagen. Es wurden 10 Trials durchgeführt. Die Lernfunktionen der VPs lauten: $VP1 = 0.7 * exp(trial)$, $VP2 = 0.8 * exp(trial)$, $VP3 = 0.9 * exp(trial)$, $VP3 = exp(trial)$. Plotte die Lernfunktionen in unterschiedlichen Farben und einer Linienbreite von 2. Beschrifte die x- und y-Achse. Um eine Legende hinzuzufügen bestimmt man die Reihenfolge der geplotteten Kurven und schreibt die gewünschten Bezeichnungen, hier

```
>>  legend('VP1','VP2','VP3','VP4');
```

Wenn man ein Label an eine bestimmte Stelle in der Abbildung setzen möchte, benutzt man die Funktion `text`. Hier werden drei Eingabe-Argumente benötigt: 1. Horizontale Position an der der Text anfangen soll, 2. die vertikale Position an der der Text starten soll, 3. den eigentlichen Text.

14.3 Veränderung der Achsen und mehrere Abbildungen

In MATLAB kann man mit dem Kommando `axis` die Achsen kontrollieren. Dieses Kommando hat viele Eigenschaften (siehe `help axis`). Wir beschreiben hier nur die gebräulichsten:

```
>>  axis([xmin xmax ymin ymax])
```

setzt die Achsenlimits in der Abbildung. `axis on` schaltet alle Achsenbeschriftungen, etc., an (bzw. `axis off`), `axis auto` setzt alle Achsenskalen zurück auf Voreinstellungen, `axis square` gibt die Achsen quadratisch aus.

Wenn es nicht genügt die vorgegebenen Achsen zu modifizieren, kann man eigene Eigenschaften der Achsen definieren. Dazu benutzt man die Funktion `set`, die der Abbildung

Features zuweisen kann. `set` wird folgendermaßen verwendet

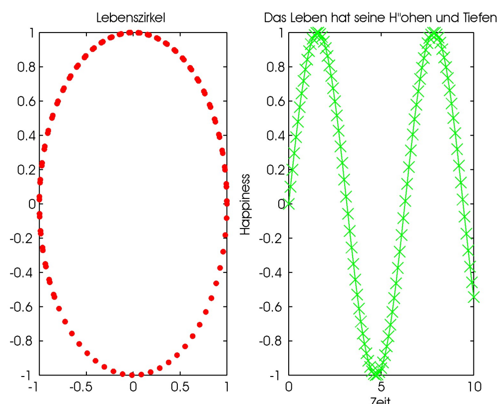
```
>>  set(gca,'XTick',-pi:pi/2:pi)

>>  set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

wobei `gca` für *get current axes* steht, so dass MATLAB weiß, dass es sich um die zuletzt geplottete Abbildung handelt, `XTick` spezifiziert den Ort der Achsenticks und `XTickLabel` benennt die Achseneinheiten.

Ein Figure-Fenster kann mehr als eine Abbildung beinhalten, indem man vor den `plot`-Befehl schreibt wieviele Abbildungen man in der Figure unterbringen will:
`subplot(m,n,p)` unterteilt die Figure in eine m×n Matrix und wählt den p-ten Bereich aus, in den etwas geplottet werden soll.

Beispiel

```
>>  x = [0:0.1:10];

>>  subplot(1,2,1);

>>  plot(cos(x),sin(x),'r.',
    'MarkerSize',12);

>>  title('Lebenszirkel')

>>  y = sin(x);

>>  subplot(1,2,2);

>>  plot(x,y,'g-x', 'MarkerSize',12);

>>  xlabel('Zeit')

>>  ylabel('Happiness')

>>  title('Das Leben hat seine Höhen und Tiefen')
```



14.4 Plot-Tools

Wenn man gar nicht weiß, wie man ein Programm schreiben soll, um ein Objekt zu plotten, kann man die Plotting-Tools von MATLAB benutzen. Markiere dazu die Variable, die du plotten willst im Workspace und klicke auf den Entsprechenden Knopf im Plot-Menü. In der geöffneten `figure` drücke auf **Insert**. Hier findet man Objekte, Pfeile, Striche, etc., die man zeichnen kann. Wähle eins aus und zeichne die gewünschte Größe und Position in die `figure`. Um das gewünschte MATLAB-Skript zu erstellen (ohne selbst zu programmieren), wähle **File→Generate Code**. Hier kann nun nachgeschaut werden, wie ein Programm aussieht, das genau das gezeichnete Objekt generiert.

Aufgabe 14.1

(a) Plotte eine Exponentialfunktion und schreibe : „Dies ist eine Exponentialfunktion!“ direkt an die Kurve.
(b) Ändere die Farbe dieser Abbildung in grau.
(c) Der folgende MATLAB-Code liefert eine Glockenkurve:

```
>>  x = linspace(0,1,200);

>>  a = 6;

>>  b = 6;

>>  y = (x. ∧ a). ∗ ((1-x). ∧ b);

>>  plot(x,y,'k')
```

Verändere den Code so, dass 2 Glockenkurven entstehen, wobei eine um 0.5 Einheiten nach rechts verschoben ist. Wenn das klappt füge 398 weitere Glockenkurven, je um 0.5 Einheiten versetzt, hinzu.
(d) Gegeben sei die Gleichung

```
>>  pcorrect = baserate + learningrate ∗ log(trial);
```

wobei `trial` die Werte 1,2,3,...,200 annehmen kann, `learningrate` kann jede reelle Zahl zwischen 0 und 1 sein, `baserate` ist $\frac{1}{4}$ und `pcorrect` kann 1 nicht übersteigen. Erstelle eine Abbildung, indem du `learningrate` $= 0.02$ setzt. Plotte `pcorrect` als Funktion von `Trial`. Beschrifte die x-Achse mit `trials`, die y-Achse mit `Proportion`. Der Titel der Abbildung sei `Learning`. Die einzelnen Punkte sollen als o-Punkte erscheinen, die mit einer durchgezogenen Linie verbunden sind.
(e) Plotte in die gleiche Abbildung aus (d) zwei weitere Lernkurven mit der `learningrate` 0.02, 0.04, 0.06.
(f) Plotte die Lernkurven aus (e) in 3 Subplots.

Aufgabe 14.2

Stelle die Funktionenschar $f(x) = x^n$ mit $n = 1, 2, ..., 5$ im Intervall $[-3, 3]$ dar.
(a) In einem Plot (nutze `hold on`).
(b) In einem Figure-Fenster (nutze `subplot`).