MATLAB Mini-Tutorials

Tobias Kies

Fassung vom 22. Dezember 2017

Inhaltsverzeichnis

Vorwort

Das Ziel dieser Mini-Tutorials ist es, eine kurze Einführung in diejenigen Sprachelemente und Funktionalitäten von MATLAB zu geben, die für den Programmierbetrieb begleitend zur computerorientierten Mathematik und zur numerischen Mathematik wesentlich sind. Die Tutorials versuchen sich besonders an all jene zu richten, die nur über wenig allgemeine Programmiererfahrung verfügen. Sie können allerdings auch für Leser mit fortgeschrittenen Programmierkenntnissen von Interesse sein, um einen schnellen Überlick über die in MATLAB verwendete Syntax zu gewinnen.

Abgesehen davon, dass eine kurze Einführung unmöglich innerhalb eines vertretbaren Rahmens auf MATLAB als Sprache in seinem vollen Umfang eingehen könnte, ist es sogar vielmehr der Fall, dass die folgenden Abschnitte in Teilen bewusst unvollständig und vereinfachend gehalten sind, um den Einstieg zu erleichtern. Falls Sie sich für bestimmte Themen genauer interessieren – sei es aus schierer Neugierde oder weil dieses Dokument Ihnen aus einem anderen Grund nicht weiterhilft – dann kann Ihnen mit großer Wahrscheinlichkeit die Online-Dokumentation von MATLAB unter `https://mathworks.com/help/matlab/index.html` weiterhelfen. Dort werden (vermutlich) alle relevanten Themen ausführlich behandelt und in der Regel auch mit nützlichen Beispielen illustriert.

Fehler gefunden? Hinweise auf Fehler jeglicher Art und Anregungen zum Inhalt werden dankbar unter

tobias.kies@fu-berlin.de

entgegengenommen.

Herzlicher Dank ergeht an dieser Stelle an Eleonore Bach und Maren Fanke für das Korrekturlesen und Mitgestalten der Vorabversion sowie das eifrige Melden von Fehlern.

Teil I.
Die Benutzeroberfläche und grundlegende Sprachelemente

0. MATLAB am Fachbereich starten und beenden

In diesem Abschnitt geht es darum, wie man MATLAB in einem unserer Computerräume auf einem Linux-Rechner startet. Falls Sie MATLAB (oder das kostenfreie Octave) auf

Ihrem persönlichen Rechner verwenden, dann können Sie diesen Abschnitt überspringen.

Aufgabe 1 (Einloggen, MATLAB starten und beenden, ausloggen)**.** Diese Aufgabe zeigt Ihnen wie Sie das Programm MATLAB auf einem Linux-Computer an unserem Fachbereich starten und richtig beenden.

1. Suchen Sie einen Computerraum auf, in dem MATLAB installiert ist. Das ist beispielsweise der Fall für die Computerräume in der Arnimallee 6 und in der Takustraße 9. Um die Poolräume ausfindig zu machen, können Sie die Raumpläne des KVV (`https://kvv.imp.fu-berlin.de/`) nutzen.

2. Suchen Sie einen freien Computer und loggen Sie sich dort ein. Benutzen Sie hierfür die Zugangsdaten Ihres Fachbereichsaccounts. Wichtig: Sie benötigen einen Fachbereichsaccound und nicht nur einen Studierenden- bzw. ZEDAT-Account. Für genauere Informationen zum Fachbereichsaccount siehe `http://www.mi.fu-berlin.de/w/IT/Computeraccess`.

3. Sobald Sie eingeloggt sind, drücken Sie die Tastenkombination `Alt + F2` und führen Sie den Befehl `konsole`, um das Terminal zu starten. Geben Sie anschließend `matlab` ein, um MATLAB zu starten.

   Anmerkung: Mit `matlab &` können Sie MATLAB so starten, dass es nicht das Terminal blockiert. Allerdings wird MATLAB dennoch geschlossen, wenn das Terminal-Fenster geschlossen wird. Alternativ können Sie MATLAB mit `matlab` starten, dann mit `Strg + Z` pausieren und schließlich mit dem Befehl `bg` vom Terminal entkoppeln.

   Anmerkung: Auf Windows-Rechnern können Sie MATLAB starten, indem Sie den entsprechenden Eintrag im Startmenu suchen und anklicken.

4. An unserem Fachbereich steht nur eine begrenzte Anzahl an MATLAB-Lizenzen zur Verfügung, weswegen wir hier auf andere Benutzer Rücksicht nehmen und MATLAB nur so lange wie nötig gestartet lassen. Beenden Sie MATLAB mit Hilfe des Befehls `exit` im Command Window. Wenn Sie MATLAB anderweitig beenden, könnten Lizenzen unter Umständen verloren gehen.

   > **Wichtig:** Wenn Sie mit dem Programmieren fertig sind oder eine längere Pause einlegen, dann vergessen Sie bitte nicht, MATLAB wieder zu beenden. Verwenden Sie dafür den Befehl `exit` in der MATLAB-Konsole, um sicherzugehen, dass die Lizenz ordnungsgemäß freigegeben wird.

5. Wenn Sie den Computer verlassen möchten und MATLAB ordnungsgemäß beendet haben, dann melden Sie sich einfach ab, indem Sie oben rechts auf den Pfeil, dann auf ihren Namen und schließlich auf `Abmelden` (bzw. `Log out`) klicken. Fahren Sie den Rechner nicht herunter.

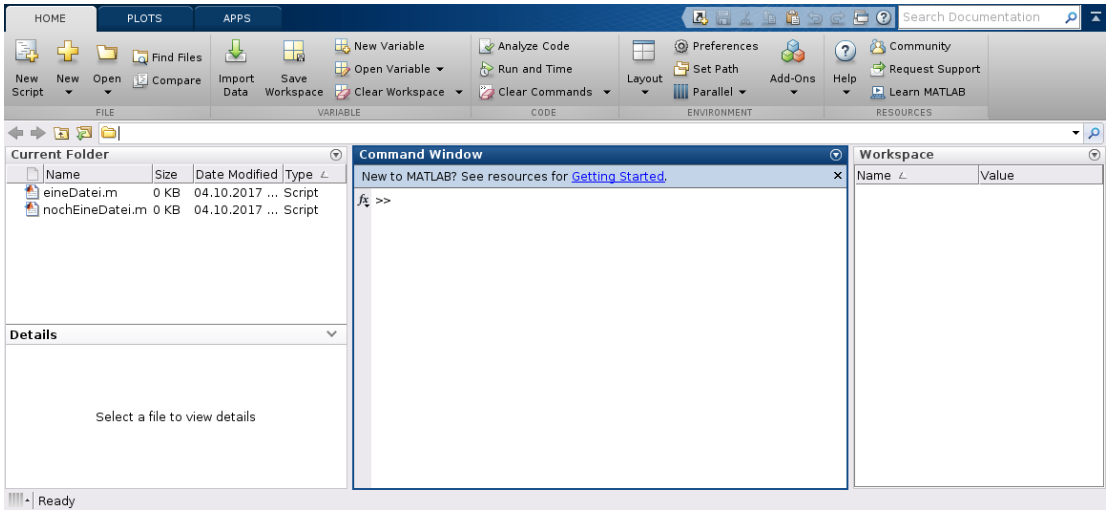


Abbildung 1.1: Beispiel für die Benutzeroberfläche zu Beginn einer Sitzung.

1. Die Benutzeroberfläche und ein erstes Programm

Wir gehen nun davon aus, dass MATLAB bereits gestartet ist. In diesem Abschnitt erklären wir in Kürze die wesentlichen Elemente der Benutzeroberfläche und zeigen, wie man für die folgenden Abschnitte Skript-Dateien erstellt und aufruft.

In Abb. 1.1 sehen Sie wie die Benutzeroberfläche in etwa aussehen sollte. Die wichtigsten Elemente dort sind die *Konsole* (engl. *Command Window*) in der Mitte, der *Datei-Explorer* links, und der *Editor*, der erst erscheint, sobald man eine Datei geöffnet hat.

In der *Konsole* können Sie Befehle eingeben, die von MATLAB dann direkt ausgeführt werden. Außerdem wird dort im Normalfall auch die Ausgabe ihres Programms angezeigt.

Im *Datei-Explorer* finden Sie eine typische Ansicht der Verzeichnisstruktur im Dateisystem vor. Falls Sie MATLAB wie oben erklärt gestartet haben, sollten Sie hier Ihr Heimverzeichnis sehen.

Der *Editor* ist ein typischer Text-Editor, den Sie später zum Bearbeiten Ihrer Programme verwenden können. Im Folgenden werden Sie dort vor allem sogenannte *Skript*-Dateien und später auch *Funktionen* bearbeiten.

Aufgabe 2 (Erste Schritte mit MATLAB)**.**

1. Wählen Sie die *Konsole* aus. Sie können dort bereits Befehle ausführen. Testen Sie beispielsweise die Grundrechenarten, indem sie nacheinander die folgenden vier Befehle eingeben und jeweils mit `Eingabe` (bzw. `Enter`) bestätigen:

   ```
   3+0.5    3-0.5    3*0.5    3/0.5
   ```

2. Wählen Sie nun den *Datei-Explorer* aus. Es wäre ziemlich unordentlich, einfach alle Programme und späteren Datei-Ausgaben im Heimverzeichnis abzuspeichern. Legen Sie deshalb im Datei-Explorer einen neuen Ordner an (mit einem beliebigen Namen; Sie können beispielsweise `matlabMiniTutorials` verwenden).

3. Machen Sie das neue Verzeichnis zum Arbeitsverzeichnis, indem Sie darauf im Datei-Explorer einen Doppelklick ausführen.

   Wichtig: MATLAB sucht Dateien immer ausgehend vom aktiven Arbeitsverzeichnis. Achten Sie daher im Verlauf stets darauf, dass sich Ihre Programme auch im aktiven Arbeitsverzeichnis befinden.

4. Erstellen Sie nun eine *Skript*-Datei, indem Sie an einer leeren Stelle im Datei-Explorer einen Rechtsklick ausführen und anschließend die entsprechende Dialogoption auswählen. Benennen Sie die Datei mit `aufgabe2.m`. Die Datei wird im aktiven Verzeichnis angelegt, und befindet sich somit direkt am richtigen Ort.

5. Machen Sie einen Doppelklick auf die neu erstellte Datei. Sie sollte Sich nun im *Editor* öffnen.

6. In die Datei können Sie nun beliebige Befehle schreiben, die Sie später ausführen möchten. Schreiben Sie beispielsweise den folgenden Code in Ihre Datei – bei dieser Gelegenheit lernen sie auch die besondere Bedeutung der Symbole `%` und `;` in MATLAB kennen.

```
% Mit % leitet man Kommentare ein. Zeilen, die
% ein % beinhalten, werden ab diesem von MATLAB ignoriert.

% Die Ergebnisse der folgende Rechnungen werden
% in der Konsole ausgegeben.
3 + 0.5
3 - 0.5
3 * 0.5
3 / 0.5

% Die Ausgabe von Rechenergebnissen wird in der Konsole
% unterdrueckt, wenn am Ende ein ; steht. Das ist bei
% laengeren Rechnungen nuetzlich, bei denen man sich
% nicht fuer Zwischenergebnisse interessiert.
3+4;      % <- Gibt nichts in der Konsole aus.
```

7. Sobald der Code in der Datei steht und Sie die Datei abgespeichert haben (!), können Sie das Skript ausführen, indem Sie in der Konsole den Namen des Skripts ohne Dateiendung eingeben – hier also `aufgabe2` – und mit `Eingabe` bestätigen. Alternativ können Sie im Editor auch einfach `F5` oder den *Run*-Knopf in der Menuleist drücken. Das Ergebnis sollte etwa so aussehen:

```
>> aufgabe2

ans =

    3.5000


ans =

    2.5000


ans =

    1.5000


ans =

     6
```

Glückwunsch! Sie haben erfolgreich ein Programm in MATLAB implementiert und ausgeführt.

Damit wissen Sie jetzt genug, um auch die nächsten Abschnitte bestreiten zu können. Es ist Ihnen überlassen, ob Sie die Aufgaben im Folgenden nur über die Konsole lösen oder weitere Skript-Dateien anlegen. Für den Fall, dass Sie Ihre Programme anderen zeigen oder zu späterer Zeit selbst wiedersehen möchten, empfehlen wir jedoch, dass Sie für jede Aufgabe jeweils eine Skript-Datei anlegen.

2. Matrix-Variablen: Zuweisung und Verkettung

Allgemein sind *Variablen* eine Art Container für Daten, die zwischengespeichert werden, bevor sie weiterverarbeitet und ggf. ausgegeben werden. Im Fall von MATLAB sind *Matrizen* der wichtigste elementare Daten-Typ, mathematisch gesprochen also Elemente der Räume $\mathbb{R}^{n\times m}$, $n, m \in \mathbb{N}$. [1] Dieser Datentyp schließt auch Skalare, Spaltenvektoren und Zeilenvektoren ein, da diese als (1×1)-, $(n\times 1)$- bzw. als $(1\times n)$-Matrizen aufgefasst werden können. Das folgende Code-Beispiel zeigt einige Möglichkeiten, wie man solche Variablen initialisieren kann.

```
a     = 42.1337;    % Ein Skalar (1x1-Matrix)
b_col = [1; 2; 3]; % Ein (3x1)-Spaltenvektor.
```

[1] In MATLAB ist es auch möglich mit komplexen Matrizen zu rechnen, also Elementen aus $\mathbb{C}^{n\times m}$. Für unsere Zwecke jedoch reellwertige Matrizen ausreichend.

```
3                       % Jedes ; leitet eine neue Zeile ein.
4  b_row = [4  5  6];   % Ein (1x3)-Zeilenvektor.
5  A     = [1  2  3;    % Eine (2x3)-Matrix.
6           4  5  6];   % Der Zeilenumbruch nach dem ; ist optional.
7
8  % Einige Kurzschreibweisen
9  n = 2;
10 k = 3;
11 m = 9;
12
13 seq1 = n:m;          % Inkrementelle Folge.
14                      % Aequivalent zu seq1 = [2 3 4 5 6 7 8 9];
15 seqk = n:k:m;        % k-inkrementelle Folge.
16                      % Aequivalent zu seqk = [2 5 8];
17
18 O = zeros(n,k);      % (n x k)-Nullmatrix.
19                      % Aequivalent zu O = [0 0 0; 0 0 0];
20 E = eye(k);          % Einheitsmatrix.
21                      % Aequivalent zu E = [1 0 0; 0 1 0; 0 0 1];
22
23
24 % Verkettung: Matrizen koennen miteinander "verklebt" werden,
25 % solange die Dimensionen dabei zusammenpassen.
26 [a; b_col]           % Aequivalent zu [42.1337; 1; 2; 3]
27 [a  b_row]           % Aequivalent zu [42.1337 4 5 6]
28 [b_row; A]          % Aequivalent zu [4 5 6; 1 2 3; 4 5 6]
29 % [A b_col]          <- Fehler!
```

Aufgabe 3 (Matrizen zuweisen)**.** Versuchen Sie selbst, einige Variablen wie oben gezeigt zu initialisieren und vergewissern Sie sich, dass das Ergebnis Ihren Erwartungen entspricht. Versuchen Sie auch bewusst, einige Fehler zu erzeugen, um zu sehen , wie MATLAB darauf reagiert.

Sie können übrigens den aktuellen Wert einer Variable einsehen, indem Sie entweder den Variablennamen in die Konsole eingeben und mit `Enter` bestätigen, oder indem Sie die Variablenansicht (*Workspace* genannt) in der Benutzeroberfläche benutzen. Die genaue Platzierung dieser Ansicht kann je nach verwendeter MATLAB-Version variieren.

3. Matrix-Variablen: Arithmetische, logische und relationelle Operatoren

Als nächstes wollen wir mit Matrizen rechnen. Dazu sehen wir uns die Grundoperatoren in MATLAB etwas genauer an.

Zunächst gibt es hier die *arithmetischen Operatoren*, die Sie zum größten Teil bereits früher für Skalare kennengelernt haben.

`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division
`^`	Potenzieren

Diese Operatoren sind auch für Matrizen gültig, indem sie beispielsweise der Addition oder Multiplikation von Matrizen im mathematischen Sinn entsprechen. [2] Wichtig ist dabei wie immer, dass die Dimensionen der Matrizen zueinander kompatibel sind. Eine Ausnahme hiervon ist, dass Operatoren in den meisten Fällen auch definiert sind, solange eine der Matrizen ein Skalar ist – am besten einfach selbst ausprobieren. Außerdem ist zu allen arithmetischen Operatoren anzumerken, dass MATLAB Punkt-vor-Strich-Rechnung beherrscht, es können aber auch Klammern `()` gesetzt werden, um die Auswertungsreihenfolge zu beeinflussen.

Einige einfache Anwendungen sind im folgenden Beispiel dargestellt.

```
% Wir nutzen im Folgenden den 'display' Befehl, um Text und
% Variablen ohne 'ans' in der Konsole auszugeben.

% Vorbereitung
a = 0.5;
b = [1 2 3];
c = [4; 5; 6];
A = [1 2 3; 4 5 6];
B = [1 2; 3 4; 5 6];
C = [1 2; 3 4];

% Addition.
display('Addition zweier Vektoren: b+b');
display(b+b);

display('Addition zweier Matrizen: A+A');
display(A+A);

display('Addition eines Vektors mit einem Skalar: a+b');
display(a+b);

display('Addition einer Matrix mit einem Skalar: a+A');
display(a+A);

```

[2] Die Division `/` bzw. auch `\` im Matrix-wertigen Sinn meint hier das Lösen eines Gleichungssystems samt diversen Verallgemeinerungen. Wir werden hier noch nicht näher darauf eingehen.

```
% Subtraktion.
display('Subtraktion verhaelt sich analog.');

% Multiplikation.
display('Multiplikation zweier Matrizen: B*A');
display(B*A);

display('Multiplikation Matrix mit Spaltenvektor: A*c');
display(A*c);

display('Multiplikation Zeilen- mit Spaltenvektor: b*c');
display(b*c);

display('Multiplikation Spalten- mit Zeilenvektor: c*b');
display(c*b);

display('Multiplikation einer Matrix mit einem Skalar: a*A');
display(a*A);

% Division fuer Matrizen wird hier ausgelassen.

% Potenzieren
display('Das Quadrat einer Matrix: C^2');
display(C^2);
```

Häufig kommt es vor, dass man nicht an einer Matrix-wertigen Operation interessiert ist, sondern stattdessen eine Operation parallel auf alle Elemente einer Matrix einzeln ausführen möchte. Solche *elementweisen arithmetischen Operatoren* sind in der folgenden Liste dargestellt.

`.*`	elementweise Multiplikation
`./`	elementweise Division
`.^`	elementweises Potenzieren

Siehe hierzu auch folgendes Beispiel:

```
% Wir nutzen im Folgenden den 'display' Befehl, um Text und
% Variablen ohne 'ans' in der Konsole auszugeben.

% Vorbereitung
a  = 3;
b  = [1 2 3];
c  = [4 5 6];
A  = [1 2 3; 4 5 6];
B  = [7 8 9; 0 1 2];

```

```
% Elementweise Multiplikation.
display('Multiplikation zweier Zeilenvektoren: b.*c');
display(b.*c);

display('Multiplikation zweier Matrizen: A.*B');
display(A.*B);

% Elementweise Division.
display('Division zweier Vektoren: c./b');
display(c./b);

display('Division zweier Matrizen: B./A');
display(B./A);

% Potenzieren
display('Ein Vektor von Potenzen fuer ein Skalar: a.^b');
display(a.^b);

display('Potenz eines jeden Eintrags einer Matrix bezueglich
    eines Skalars: A.^a');
display(A.^a);
```

Verwandt dazu sind auch die folgenden Operatoren.

'	Adjunktion
.'	Transponieren

Da wir ausschließlich mit reellwertigen Matrizen arbeiten werden, machen diese beiden Operatoren für uns keinen Unterschied.

```
% Wir nutzen im Folgenden den 'display' Befehl, um Text und
% Variablen ohne 'ans' in der Konsole auszugeben.

% Vorbereitung
A = [1 2 3; 4 5 6];

% Adjungieren.
display('Adjungieren der Matrix A.');
display(A');

% Transponieren.
display('Transponieren der Matrix A.');
display(A.');
```

Für die *logischen Operatoren* lernen wir einen weiteren Daten-Typ kennen: Die *logischen Matrizen*. Solche Matrizen entsprechen Elementen aus $\{0,1\}^{n\times m}$, wobei 1 für *wahr*

und 0 für *falsch* steht. In den meisten Fällen werden wir nur logische Skalare benötigen. Die wichtigsten Operatoren für logische Variablen sind:

`&`	logisches Und	(aussagenlogisches $\wedge$)
`|`	logisches Oder	(aussagenlogisches $\vee$)
`~`	Negation	(aussagenlogisches $\neg$)

```
% Wir nutzen im Folgenden den 'display' Befehl, um Text und
% Variablen ohne 'ans' in der Konsole auszugeben.

% Vorbereitung
T = true;   % Logisches Skalar mit 'wahr'
F = false;  % Logischs Skalar mit 'falsch'

A = true(2,3); % Logische (2x3)-Matrix mit 'wahr'-Eintraegen.
B = [T F T; F T F]; % Logische (2x3)-Matrix.

% Skalare Verknuepfungen.
display( 'Logisches Und angewendet auf Skalare.');
display( T & T );
display( T & F );
display( F & T );
display( F & F );

display( 'Logisches Oder angewendet auf Skalare.');
display( T | T );
display( T | F );
display( F | T );
display( F | F );

display( 'Negation angewendet auf Skalare.');
display( ~T );
display( ~F );

% Dasselbe noch einmal Matrix-wertig.
display( 'Logisches Und angewendet auf Matrizen.');
display( A & A );
display( A & B );
display( B & A );
display( B & B );

display( 'Logisches Oder angewendet auf Matrizen.');
display( A | A );
display( A | B );
```

```
38 display( B | A );
39 display( B | B );
40
41 display( 'Negation angewendet auf Matrizen.');
42 display( ~A );
43 display( ~B );
```

Mit *relationellen Operatoren* lassen sich numerische Matrizen in logische Matrizen überführen. Die wichtigsten relationellen Operatoren sind:

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

```
1  % Wir nutzen im Folgenden den 'display' Befehl, um Text und
2  % Variablen ohne 'ans' in der Konsole auszugeben.
3
4  % Vorbereitung
5  a   = 1;
6  b   = 2;
7  A   = [1 2 3; 4 5 6];
8  B   = [1 3 2; 6 5 4];
9
10 % Relationelle operatoren angewendet auf Skalare.
11 display( 'a<b' );
12 display( a<b );
13
14 display( 'a<=b' );
15 display( a<=b );
16
17 display( 'a==b' );
18 display( a==b );
19
20 display( 'a~=b' );
21 display( a~=b );
22
23 display( 'a>=b' );
24 display( a>=b );
25
26 display( 'a>b' );
27 display( a>b );
28
```

```
% Relationelle operatoren angewendet auf Matrizen.
display( 'A<B' );
display( A<B );

display( 'A<=B' );
display( A<=B );

display( 'A==B' );
display( A==B );

display( 'A~=B' );
display( A~=B );

display( 'A>=B' );
display( A>=B );

display( 'A>B' );
display( A>B );
```

Aufgabe 4. Lesen Sie die Programme in diesem Abschnitt durch und überlegen Sie sich, was die Ergebnisse der einzelnen Operationen sind. Führen Sie die Skripte anschließend aus und vergleichen Sie die Resultate mit Ihren Erwartungen.

Aufgabe 5. Versuchen Sie, einige Operationen durchzuführen, die nicht erlaubt sind. Mit welchen Meldungen reagiert MATLAB und wie lässt sich der Fehler beheben?

Aufgabe 6. Definieren Sie in MATLAB die Matrizen

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \qquad B = \begin{pmatrix} 4 & 5 & 6 \\ 3 & 2 & 1 \end{pmatrix}.$$

Berechnen Sie anschließend die Matrix C, die durch

$$C_{ij} = \frac{\frac{1}{2}A_{ij} - (A_{ij} \cdot B_{ij})^3}{1 + A_{ij} + B_{ij}}$$

definiert ist. Vergleichen Sie Ihr Ergebnis mit der Lösung:

$$C = \begin{pmatrix} -10{,}58\overline{3} & -124{,}875 & -583{,}05 \\ -215{,}750 & -124{,}6875 & -26{,}625 \end{pmatrix}$$

Anmerkung: Wir folgen hier und im Folgenden der üblichen Konvention für Matrix-Indizierung, sprich A_{ij} bezeichnet den Eintrag der Matrix A in der i-ten Zeile und j-ten Spalte.

Anmerkung: Falls Sie möchten, dass MATLAB in der Konsole mehr Nachkommastellen anzeigt, dann können Sie den Befehl `format long` verwenden. Mit `format short` kehren Sie wieder zu MATLABs kompakterer Standardeinstellung zurück.

Aufgabe 7. Erstellen Sie in MATLAB eine neue Skript-Datei und definieren Sie dort die Matrizem

$$A = \begin{pmatrix} 0 & 5 \\ 7 & -6 \end{pmatrix}, \qquad B = \begin{pmatrix} 10 & 13 \\ 10 & 18 \end{pmatrix}.$$

1. Berechnen Sie in MATLAB eine logische Matrix `I`, die die positiven Elemente in A anzeigt, und eine logische Matrix `J`, die die Elemente von B anzeigt, die kleiner oder gleich 10 sind. Überprüfen Sie Ihr Ergebnis, indem Sie es mit der Lösung

   $$I = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad J = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$$

   vergleichen.

2. Berechnen Sie unter Verwendung der logischen Matrizen `I` und `J` eine logische Matrix `K`, die alle Elemente anzeigt, für die $A_{ij} > 0$ und $B_{ij} \leq 10$ gilt. Ihr Ergebnis sollte folgendermaßen aussehen:

   $$K = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

3. Berechnen Sie nun unter Verwendung der logischen Matrizen `I` und `J` eine logische Matrix `L`, die alle Elemente anzeigt, für die *entweder* $A_{ij} > 0$ *oder* $B_{ij} \leq 10$ gilt. [3] Das Ergebnis sollte also so aussehen:

   $$L = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

   Anmerkung: Vergessen Sie nicht, dass Sie Klammern `( )` benutzen können, um die Auswertungsreihenfolge von Operationen vorzuschreiben.

4. Matrix-Variablen: Indizierung

Für die meisten Algorithmen wird es nicht ausreichen, einfach nur Elementaroperationen auf Matrizen anzuwenden. Stattdessen werden Sie auf bestimmte Einträge einer Matrix zugreifen wollen, um diese gezielt auszulesen oder mit neuen Daten zu belegen.

Es unterscheiden sich zwei Arten von Indizierung: Die *„klassische“ Indizierung* und die *logische Indizierung*.

Bei der klassischen Indizierung gibt man einfach die Zeile und Spalte des Eintrags an, für den man sich interessiert. Im Vergleich zu vielen anderen Programmiersprachen hat MATLAB die Besonderheit, dass Zeilen und Spalten ab 1 gezählt werden und dass es möglich ist, anstatt nur einzelner skalarer Werte sogar komplette Untermatrizen auf

[3] Gemeint ist hier das logische Entweder-Oder. Für eine kurze Erklärung siehe beispielsweise `https://de.wikipedia.org/wiki/Kontravalenz`.

einmal anzusprechen. Außerdem können Matrizen als Vektoren aufgefasst werden. Dies geschieht, indem die Spalten einer Matrix zu einem langen Spaltenvektor aneinandergereiht werden (dies entspricht auch der Reihenfolge wie die Einträge von MATLAB im Speicher abgelegt werden).

Für die logische Inidizierung wendet man eine logische Matrix auf eine numerische Matrix an, wobei beide Matrizen gleich groß sein müssen. Als Ergebnis erhält man einen Spaltenvektor, der alle Einträge der numerischen Matrix enthält, die in der logischen Matrix mit 1 (*wahr*) belegt sind. Die Einträge sind dabei aufsteigend nach ihrer Spaltennummer und dann nach ihrer Zeilennummer sortiert.

Die verschiedenen Möglichkeiten zur Indizierung werden am besten durch ein Beispiel illustriert.

```
% Vorbereitung.
b    = [1 5 3];
c    = [4; 2; 6];
A    = [1 5 3; 4 2 6; 7 8 9];

i    = 2;
j    = 3;
I    = [1 3];


% Vektoren.
display('Zugriff auf einzelnes Element: b(i), c(i),');
display(b(i));
display(c(i));

display('Zugriff auf Untervektor: b(I), c(I)');
display(b(I));
display(c(I));

display('Zugriff auf das letzte Element: b(end)');
display(b(end));

display('Alle elemente ab dem zweiten: b(2:end)');
display(b(2:end));


% Matrizen
display('Zugriff auf einzelnes Element: A(i,j)');
display(A(i,j));

display('Zugriff auf komplette Zeile: A(i,:)');
display(A(i,:));
```

```
33
34 display('Zugriff auf komplette Spalte: A(:,j)');
35 display(A(:,j));
36
37 display('Zugriff auf mehere Zeilen gleichzeitig: A(I,:)');
38 display(A(I,:));
39
40 display('Zugriff auf letzte Zeile: A(end,:)');
41 display(A(end,:));
42
43 display('Zugriff auf letzte beiden Spalten: A(:,end-1:end)');
44 display(A(:,end-1:end));
45
46 display('Zugriff auf Untermatrix: A(2:end,2:end)');
47 display(A(2:end,2:end));
48
49 display('Matrix als Spaltenvektor: A(:)');
50 display(A(:));
51
52
53 % Logiscche Indizierung.
54 % Vergleiche Abschnitt zu relationellen Operatoren.
55 I   = (A < 5);      % Ermittle  Elemente kleiner als 5.
56 display('Alle Elemente in A, die kleiner als 5 sind:');
57 display(A(I));
58 A(I)    = -A(I);    % Mache all diese Elemente negativ.
59 display('Matrix A nach Modifikation:');
60 display(A);
```

Aufgabe 8. Lesen Sie sich das Programmbeispiel sorgfältig durch und machen Sie sich Gedanken darüber, was die Ergebnisse der einzelnen Operationen sein könnten. Führen Sie das Skript anschließend aus und vergleichen Sie die Ausgabe mit Ihren Erwartungen. Achten Sie dabei auch darauf, ob die Ergebnisse als Zeilen- oder Spaltenvektor zurückgegeben werden.

Aufgabe 9. Versuchen Sie, beim Indizieren einige Fehler zu verursachen. Wie reagiert MATLAB, wenn Sie versuchen, auf Einträge außerhalb des zulässigen Bereichs zuzugreifen?

Aufgabe 10. Definieren Sie `A = magic(10)`. [4]

1. Geben Sie A_{12} aus. Als Ergebnis sollte der Wert 99 ausgegeben werden.

[4] Mit der Funktion `magic(n)` können Sie eine $(n \times n)$-Matrix mit gewissen besonderen Eigenschaften erzeugen. Eine genaue Erklärung finden Sie unter `https://de.mathworks.com/help/matlab/ref/magic.html`.

2. Geben Sie die vierte Spalte von A aus. Als Ergebnis sollten Sie

$$(8, 14, 20, 21, 2, 83, 89, 95, 96, 77)^T$$

erhalten.

3. Geben Sie die erste, zweite und siebte Zeile von A ab der vierten Spalte aus. Als Ergebnis sollten Sie

$$\begin{pmatrix} 8 & 15 & 67 & 74 & 51 & 58 & 40 \\ 14 & 16 & 73 & 55 & 57 & 64 & 41 \\ 89 & 91 & 48 & 30 & 32 & 39 & 66 \end{pmatrix}$$

erhalten.

4. Geben Sie die Untermatrix von A aus, die aus allen Zeilen mit einem geraden Index besteht.

$$\begin{pmatrix} 98 & 80 & 7 & 14 & 16 & 73 & 55 & 57 & 64 & 41 \\ 85 & 87 & 19 & 21 & 3 & 60 & 62 & 69 & 71 & 28 \\ 17 & 24 & 76 & 83 & 90 & 42 & 49 & 26 & 33 & 65 \\ 79 & 6 & 13 & 95 & 97 & 29 & 31 & 38 & 45 & 72 \\ 11 & 18 & 100 & 77 & 84 & 36 & 43 & 50 & 27 & 59 \end{pmatrix}$$

Anmerkung: Erinnern Sie sich an die Kurzschreibweise `n:k:m`.

5. Finden Sie alle Einträge in A, die größer als 15, aber kleiner als 20 sind und multiplizieren Sie diese mit 10. Setzen Sie alle anderen Einträge auf 0. Die neue Matrix A sollte nun so aussehen:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 160 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 190 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 170 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 180 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

5. Funktionen: Einige wichtige Standardfunktionen

Funktionen sind Unterprogramme, die Funktionalitäten übernehmen, die über einfache Elementaroperationen hinausgehen. Funktionen bekommen bei Aufruf im Allgemeinen eine gewisse Anzahl an *Eingabeparametern* übergeben, verarbeiten diese auf irgendeine Weise und geben ihr Ergebnis in Form einer gewissen Anzahl an *Ausgabevariablen* an

den aufrufenden Kontext zurück. Eine Besonderheit von MATLAB in Bezug auf die zur Verfügung gestellten Funktionen ist, dass viele mathematische Funktionen anstatt nur für Skalare auch für allgemeine Matrizen definiert sind. In den meisten Fällen entspricht das einer elementweisen Anwendung der Funktion auf die einzelnen Einträge der Matrix. Zur besseren Illustration zeigen wir auch hier ein Beispiel.

```
% Vorbereitung.
a    = 5;
b    = 2;
x    = [1  2  3];
A    = [1 -2  3; 4 -5  6];

display('Groesse einer Matrix: size(A)');
[n,m]    = size(A);
display(n);
display(m);

display('Laenge eines Vektors: length(x)');
display(length(x));
display(length(x'));

display('abs(A)');           % Betrag
display(abs(A));

display('cos(A), sin(A)');   % Cosinus und Sinus
display(cos(A));
display(sin(A));

display('exp(A)');           % Exponentialfunktion
display(exp(A));

display('max(a,b)');         % Maximum zweier Skalare.
display(max(a,b));

display('max(A)');           % Maximum (spaltenweise)
display(max(A));

display('min(a,b)');         % Minimum zweier Skalare.
display(min(a,b));

display('min(A)');           % Minimum (spaltenweise)
display(min(A));

display('mod(a,b)');         % Teilen mit Rest.
```

```
39 display(mod(a,b));              % Hier: a/b = 5/2 = 2 Rest 1
40                                 % Also: mod(a,b) == 1
41
42 display('sqrt(A)');              % Wurzelfunktion. Gleich zu .^0.5
43 display(sqrt(A));
44
45 % Vektor mit 11 Eintraegen, die gleichmaessig von a bis b
       verteilt sind.
46 display('linspace(a,b,11)');
47 display(linspace(a,b,11));
```

In MATLAB gibt es eine Fülle interessanter Funktionen, die im Rahmen dieser Einführung allerdings unmöglich abgedeckt werden können.

Viele Funktionen werden Sie ganz nebenher kennenlernen, wenn Sie mit MATLAB diverse Aufgaben erledigen möchten, die so alltäglich sind, dass für diese bereits vor-eingebaute Funktionen zur Verfügung gestellt werden. Sagen wir beispielsweise, dass es Teil Ihrer Aufgabe ist, das Produkt der Elemente eines Vektors $v \in \mathbb{R}^n$ zu berechnen, also den Ausdruck $v_1 \cdot v_2 \cdots v_n$. Sie können dies einerseits mittels einer For-Schleife und einem zugegebenermaßen überschaubaren Aufwand erreichen (siehe Abschnitt 8) oder andererseits, indem Sie beispielsweise die Internet-Suche *„matlab product of vector elements“* ausführen (oder zumindest geschickt raten) und zu dem Schluss kommen, dass Ihr Problem auch kurzerhand mittels `prod(v)` zu lösen ist.

Aufgabe 11. Lesen Sie auch hier den Code durch und überlegen Sie sich die Ergebnisse. Führen Sie das Skript anschließend aus und vergleichen Sie Ihre Erwartung mit den tatsächlichen Ergebnissen.

Aufgabe 12. Nutzen Sie eine Informationsquelle Ihrer Wahl, um die folgenden Fragen zu beantworten:

1. Wie kann man die Norm eines Vektors v berechnen?
2. Wie kann man eine Zahl auf- bzw. abrunden?
3. Mit welcher Funktion kann man die Diagonale einer Matrix $A \in \mathbb{R}^{n\times n}$ extrahieren, also den Vektor $(A_{11}, \ldots, A_{nn}) \in \mathbb{R}^n$ erhalten? Wie kann man umgekehrt aus einem Vektor $v \in \mathbb{R}^n$ die Diagonalmatrix

$$\begin{pmatrix} v_1 & & & & 0 \\ & \ddots & & & \\ & & v_i & & \\ & & & \ddots & \\ 0 & & & & v_n \end{pmatrix}$$

erzeugen?

4. Wie kann man eine Matrix erstellen, die aus aneinandergereihten Kopien einer kleineren Matrix besteht? Sprich, wie kann man beispielsweise für $A \in \mathbb{R}^{n\times m}$ und $l, k \in \mathbb{N}$ die $(kn \times lm)$-Matrix

$$\overbrace{\begin{pmatrix} A & \cdots & A \\ \vdots & & \vdots \\ A & \cdots & A \end{pmatrix}}^{l\text{-mal}} \Bigg\} k\text{-mal}$$

definieren?

Anmerkung: Für umfangreichere Ergebnisse lohnt es sich häufig, die Suche in englischer Sprache durchzuführen.

6. Funktionen: Funktion-Dateien und Funktionen-Handles

Für die meisten Aufgaben müssen Sie eigene Funktionen schreiben, die dann von einem Skript heraus aufgerufen werden können. Um eine eigene Funktion zu MATLAB hinzuzufügen, müssen Sie zunächst eine Datei erstellen, die genauso heißt wie die Funktion, die Sie neu erstellen möchten.

> **Um es wirklich noch einmal zu betonen:** Die Datei, die eine Funktion beinhaltet, muss abzüglich der Dateiendung in MATLAB *genauso* heißen wie die Funktion selbst. Außerdem sollten Sie Namenskonflikte vermeiden, sprich, Ihre Funktionen *nicht* gleichnamig zu bereits existierenden Funktionen oder Variablen wählen.

Sie können eine neue Funktionsdatei ganz bequem über den Datei-Explorer anlegen, indem Sie auf eine freie Stelle im aktiven Arbeitsverzeichnis rechtsklicken und die entsprechende Dialogoption `Neue Datei > Funktion` auswählen. Im Folgenden sehen Sie ein Beispiel von einer von MATLAB erstellten Vorlage für eine Funktion mit Namen `myfunction` (in der Datei `myfunction.m`).

```
1 function [ output_args ] = myfunction( input_args )
2     % ...
3 end
```

Eine Funktionendefinition ist im Wesentlichen so aufgebaut, dass man ihren Anfang mit dem `function` Schlüsselwort kennzeichnet und die Ausgabe-Variablen (hier noch `output_args`), den Funktionnamen (hier `myfunction`) sowie die Eingabeparameter (hier noch `input_args`) angibt. Abgeschlossen wird die Definition durch ein `end` Schlüsselwort. Im Block zwischen `function` und `end` steht dann der Code, in dem aus den Eingabeparametern die Ausgabe berechnet wird.

> **Übrigens:** Zur besseren Leserlichkeit ist es üblich, den Code innerhalb eines Blocks, der mit `end` begrenzt wird, etwas einzurücken. Sie können hierfür die `Tabulator`-Taste verwenden.

Hier ein Beispiel für eine konkrete Implementierung einer Funktion `myfunction1` mit zwei Eingabeparametern und einem Ausgabeparameter.

```
function z = myfunction1( x, y )
    % Diese Funktion berechnet x*y und gibt das Ergebnis
    % in z zurueck.
    % Damit diese Funktion auch fuer Matrizen Sinn ergibt,
    % verwenden wir hier die elementweise Multiplikation.
    z   = x.*y;
end
```

Ein Aufruf der Funktion aus der Konsole heraus könnte etwa wie folgt aussehen.

```
>> myfunction1(2,3)
ans =  6
>> myfunction1(1:5,1:5)
ans =

    1    4    9   16   25
```

Und hier ein weiteres Beispiel für eine Implementierung mit einem Eingabeparameter und zwei Ausgabeparametern.

```
function [a,y] = myfunction2( x )
    % Diese Funktion erhaelt eine numerische Matrix x als
    % Eingabeparameter.
    % Die Ausgabe besteht aus
    %   * einer logischen Matrix a, die die negativen Eintraege
    %     in x kennzeichnet und
    %   * einer numerischen Matrix y, die die Quadrate der
    %     Eintraege in x enthaelt.
    a   = (x < 0);
    y   = x.^2;
end
```

Ein Aufruf der Funktion aus der Konsole heraus könnte etwa wie folgt aussehen.

```
>> [u,v] = myfunction2( [-1 2 -3; 4 -5 6] )
u =

    1    0    1
    0    1    0

v =

    1    4    9
   16   25   36
```

In MATLAB gibt es außerdem *Funktionen-Handles*. Diese sind ein weiterer Variablen-Typ, in dem Funktionen gespeichert werden können. Damit ist es insbesondere möglich, einer Funktion eine andere Funktion als Parameter zu übergeben. Außerdem ist hierdurch in MATLAB aber auch ein Mechanismus eingebaut, um einfach strukturierte Funktionen direkt im Code zu definieren, ohne dafür extra eine klassische Funktionendefinition anlegen zu müssen. Sehen Sie die folgenden Beispiele für die Verwendung von Funktionen-Handles.

```
% Funktionen-Handle auf eine existierende Funktion.
f  = @cos;                  % Cosinus.

% Handle auf eine neue Funktion mit einer Eingabevariable.
g  = @(x) x.^2;             % Quadrat.

% Handle auf eine neue Funktion mit zwei Eingabevariablen.
h  = @(x,y) x.^2 + y.^3;    % Polynom in zwei Variablen.

% Funktion, die eine andere Funktion als Parameter nimmt.
F  = @(f) f(1);             % Wertet f an 1 aus.


% Beispiel-Ausfuehrungen.
display(f(pi));
display(g(1:5));
display(h(1,1:4));
display(F(f));                % Aequivalent zu F(@cos)
display(F(g));
```

Die Ausgabe des Programms sieht in etwa so aus:

```
-1

     1     4     9    16    25

     2     9    28    65

  0.54030

  1
```

Aufgabe 13. Definieren Sie ein Funktionen-Handle, das der mathematischen Abbildung

$$f(x, y) = x^2 + y^2$$

entspricht. Stellen Sie dabei sicher, dass ihre Funktion auch für Matrizen sinnvoll ist. Das heißt, für $X, Y \in \mathbb{R}^{n \times m}$ soll $f(X, Y) = Z \in \mathbb{R}^{n \times m}$ mit $Z_{ij} = f(X_{ij}, Y_{ij})$ gelten. Also

beispielsweise

$$X = \begin{pmatrix} 1 & 0 \\ \frac{1}{\sqrt{2}} & 1 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & 1 \\ \frac{1}{\sqrt{2}} & 1 \end{pmatrix}. \qquad f(X,Y) = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}.$$

Anmerkung: Erinnern Sie sich an die elementweise definierten arithmetischen Operatoren.

Aufgabe 14. Erstellen Sie eine Funktion `somewhatmean`. Diese Soll für einen Vektor $v \in \mathbb{R}^n$ und eine Funktion f den Wert $\frac{1}{n}\sum_{i=1}^{n} f(v_i)$ berechnen. Werten Sie Ihre Funktion anschließend für $v = (0, 1, \ldots, 100)$ und $f = \cos$ aus. (Ergebnis $\approx 0{,}00463080585910504$.)

Aufgabe 15. Legen Sie eine Funktionsdatei mit dem Namen `doublethefun.m` an. Diese Funktion soll so definiert sein, dass für ein Funktionen-Handle `f` auch `g = doublethefun(f)` ein Funktionenhandle ist und dass `g(x) = 2*f(x)` gilt. Die Funktion g soll also das Doppelte von f darstellen. Testen Sie Ihre Funktion mit geeigneten Beispielauswertungen auf Richtigkeit.

Anmerkung: Nicht nur Eingabeparameter, sondern auch Ausgabeparameter dürfen Funktionenhandles sein.

7. Kontrollstruktur: If-Abfrage

Bisher haben unsere Programme nur einen linearen Ablauf befolgt: Alle Befehle wurden immer der Reihe nach ausgeführt. Häufig aber soll der weitere Programmablauf von einem Zwischenergebnis oder einem Eingabeparameter abhängen. Ein solches Verzweigen im Programm-Code kann man mit *If-Abfragen* erreichen.

Das Konzept erklärt sich am einfachsten an ein paar Beispielen. Gehen wir etwa davon aus, dass wir vom Benutzer eine skalare numerische Eingabe `a` erhalten haben und dass unser Programm nur für nicht-negative Eingabewerte funktioniert. Mit dem folgenden Code können Sie sicherstellen, dass `a` nicht-negativ ist.

```
% Wir nehmen an, dass a ein numerisches Skalar ist.
% Zu Vorfuehrungszwecken schreiben wir einen Wert vor.
a = -5;

% Falls a negativ ist...
if a < 0
  % ... wechseln wir das Vorzeichen von a.
  a = -a;
end

% Wir geben hier noch einmal den Endwert von a aus.
a
```

Vielleicht möchten Sie den Benutzer aber auch nur für negative Eingaben tadeln und für nicht-negative loben:

```
% Wir nehmen an, dass a ein numerisches Skalar ist.
% Zu Vorfuehrungszwecken schreiben wir einen Wert vor.
a = -5;

% Falls a negativ ist...
if a < 0
  % ... sind wir unzufrieden!
   display('Negative Variablen sind nicht erlaubt!');
%Ansonsten...
else
  % ... ist alles super!
   display('Weiter so!');
end
```

Und schließlich können Sie auch noch auf die moralische Grenzwertigkeit einer Null-Eingabe eingehen.

```
% Wir nehmen an, dass a ein numerisches Skalar ist.
% Zu Vorfuehrungszwecken schreiben wir einen Wert vor.
a = -5;

% Falls a negativ ist...
if a < 0
  % ... sind wir unzufrieden!
   display('Negative Variablen sind verboten!');
% Ansonsten, falls a gleich Null ist....
elseif a == 0
  % ... klingt das auch noch verdaechtig.
   display('Wo soll es denn hingehen?');
% In allen uebrigen Faellen...
else
  % ... ist alles super!
   display('Weiter so!');
end
```

Damit kennen Sie jetzt die Schlüsselwörter `if`, `elseif` und `else`, mit denen Sie nun Ihren Programmfluss mittels Verzweigungen steuern können.

Aufgabe 16. Implementieren Sie die Betragsfunktion `abs` selbst. Legen Sie dazu eine neue Funktion `myabs` an, die für $x \in \mathbb{R}$ mathematisch durch

$$\text{myabs}(x) = \begin{cases} x & \text{für } x \geq 0 \\ -x & \text{für } x < 0 \end{cases}$$

definiert ist. Testen Sie Ihre Implementierung, indem die Ergebnisse von `myabs` für geeignete Eingabeparametern mit denen der Funktion `abs` vergleichen.

Bonus: Schaffen Sie es auch, Ihre Funktion `myabs` so zu implementieren, dass sie für Matrizen X sinnvoll ist, und ohne dabei `for`- oder `while`-Schleifen zu benutzen?

Aufgabe 17. Implementieren Sie die Funktion f, die für $x \in \mathbb{R}$ durch

$$f(x) = \begin{cases} x^2 & \text{für } x < 0 \\ x & \text{für } x \in [0, 2] \\ 0 & \text{für } x > 2 \end{cases}$$

definiert ist. Vergewissern Sie sich von der Richtigkeit Ihrer Implementierung, indem Sie `f` an geeigneten Stellen auswerten.

Aufgabe 18. Für Schaltjahre gilt die folgende Regel: Ein Jahr ist ein Schaltjahr, falls die Jahreszahl durch vier teilbar ist, aber nicht durch 100. Eine Ausnahme gilt, falls die Jahreszahl jedoch durch 400 teilbar ist – dann ist das Jahr trotzdem ein Schaltjahr.

Schreiben Sie eine Funktion `istSchaltjahr`, sodass für eine Ganzzahl `n` durch den Wert `istSchaltjahr(n)` beschrieben Wird, ob die Jahreszahl `n` ein Schaltjahr ist. Sprich, `istSchaltjahr(n)` ist die logische 1 für Schaltjahre `n` und ansonsten die logische 0.

Prüfen Sie Ihre Implementierung auf Richtigkeit, indem Sie die Fuktion mit geeigneten Werten testen, die die verschiedenen Regeln abprüfen.

Anmerkung: Wichtig für diese Aufgabe ist die Funktion `mod`.

8. Kontrollstruktur: For-Schleife

Mit *For-Schleifen* können Sie gleichartige Aufgaben in Ihrem Programm wiederholt ausführen lassen. Die Idee ist, dass man einer For-Schleife einen Container (in der Regel einen Vektor) übergibt, und dass für jedes Element dieses Containers ein gewisser Code ausgeführt wird. Dabei hängt die Reihenfolge der Ausführungen von der Reihenfolge der Elemente im Container ab. Das ist besonders dann wichtig, wenn die einzelnen Schleifendurchläufe voneinander abhängig sind. Wir zeigen das Konzept genauer an einem Beispiel.

```
% Das Standardbeispiel:
% Iteriere ueber alle Zahlen von 1 bis 10...
for i = 1:10
  % ... und berechne das Quadrat der jeweiligen Iterierten.
  display(i^2);
end


% Man kann for-Schleifen auch schachteln.
for i = 1:3
  for j = 1:3
    display(i*j);
  end
```

```
14 end
15
16
17 % Ebenso kann man Schleifen auch nutzen, um Container
18 % rueckwaerts zu durchlaufen.
19 v   = [1 2 3 5];
20 for i = length(v):-1:1
21   display(v(i));
22 end
23
24
25 % Ein Beispiel mit einem eher beliebigen Container.
26 c = 0;
27 for i = [7 4 6 2 8]
28   % Wir addieren die aktuelle Iterierte auf c.
29   c   = c+i;
30 end
31
32 % Wir geben das Endergebnis aus.
33 display('Die Summe aller Iterierten:');
34 display(c);
35
36
37 % Das vorangegangene Beispiel kann man auch so schreiben:
38 v = [7 4 6 2 8];
39 c = 0;
40 for i = 1:length(v)
41   c   = c + v(i);
42
43   % Der Vorteil dieser Schreibweise: Man koennte hier
44   % jetzt noch explizit auf den Index i zugreifen, um
45   % ggf. weitere i-abhaengige Aufgaben zu erledigen.
46 end
```

Solche Schleifen sind besonders dann nützlich, wenn Sie zum Zeitpunkt des Programmierens noch nicht wissen , wie oft eine bestimmte Aufgabe wiederholt werden muss, oder wenn Sie sich einfach etwas Schreibarbeit sparen wollen. Wichtig ist bei For-Schleifen allerdings, dass das Programm während der Ausführung trotzdem irgendwann feststellen muss, wie oft diese Aufgabe wiederholt werden soll.

Ein weiteres Hilfsmittel sind die Schlüsselwörter `break` und `continue`. Mittels `break` wird die aktuelle Schleife abgebrochen und das Programm nach dem nächsten `end` Schlüsselwort fortgesetzt. Mit `continue` brechen Sie hingegen nur die aktuelle Iteration ab und gehen direkt zur nächsten Iteration über.

```
1 v   = [5 3 8 4 0 9 2];
```

```
2
3  % Wir berechnen die Inverse eines jeden Eintrags.
4  for vi = v
5    % Falls vi == 0 ist, ist die Inverse nicht definiert.
6    % Wir brechen die aktuelle Iteration ab und machen
7    % beim naechsten vi weiter.
8    if vi == 0
9      display('vi = 0 kann nicht invertiert werden.');
10     continue;
11   end
12
13   display(1/vi);
14 end
15
16
17 % Wir berechnen die Summe der Inversen
18 % Wir brechen das Verfahren ab, wenn ein Eintrag 0 ist.
19 c = 0;
20 for vi = v
21   % Falls vi == 0 ist, ist die Inverse nicht definiert.
22   % Wir brechen die Schleife ab.
23   if vi == 0
24     display('vi = 0 kann nicht invertiert werden.');
25     display('Setze die Summe zurueck auf 0 und breche ab.');
26     c = 0;
27     break;
28   end
29
30   c = c + vi;
31 end
32
33 display('Endergebnis:');
34 display(c);
```

Aufgabe 19. Lesen Sie die Code-Beispiele durch und versuchen Sie den Programmablauf nachzuvollziehen. Führen Sie das Skript anschließend aus und vergleichen Sie Ihre Erwartung mit den tatsächlichen Ergebnissen.

Aufgabe 20. Implementieren Sie eine Funktion `alternatingSum`, die zu einem Vektor $v \in \mathbb{R}^n$ die alternierende Summe

$$-v_1 + v_2 - v_3 + \cdots$$

berechnet. Prüfen Sie Ihre Implementierung mit geeigneten Testbeispielen. Für $v = (1, 4, 8, 3)$ sollte der Rückgabewert beispielsweise -2 sein.

Aufgabe 21. Implementieren Sie die MATLAB-Funktion `prod` selbst. Definieren Sie dazu eine Funktion `myprod`, die zu einem Eingabevektor $v \in \mathbb{R}^n$ das Produkt seiner Einträge zurückgibt, also

$$v_1 \cdot v_2 \cdot \dots \cdot v_n.$$

Prüfen Sie die Richtigkeit Ihrer Implementierung, indem Sie Ihre Funktion an einigen geeigneten Beispielen testen. Für $v = (8, -2, -10, 1, -2)$ sollte das Resultat beispielsweise -320 sein.

Aufgabe 22. Schreiben Sie eine Funktion `evensum`, die zu einem Vektor $v \in \mathbb{R}^n$ die Summe aller geraden Einträge von v zurückgibt. Verzichten Sie dabei auf die Verwendung von vektorwertiger Indizierung und verwenden Sie auch nicht die Funktion `sum`. Testen Sie Ihre Implementierung an einigen geeigneten Beispielen. Für $v = (-1, 6, -2, 0, -2)$ sollten Sie etwa das Ergebnis 2 erhalten.

Aufgabe 23. Schreiben Sie eine Funktion `sumfun`, die zu einer Matrix $A \in \mathbb{R}^{n \times m}$ den Wert

$$\sum_{i=1}^{n} \sum_{j=1}^{m} i \cdot j \cdot A_{ij}$$

berechnet. Testen Sie wie gewohnt Ihre Implementierung mittels geeigneter Testbeispiele auf Richtigkeit. Für `A = vander(1:3)` sollten Sie beispielsweise den Rückgabewert 82 erhalten.

Anmerkung: Angedacht ist, dass Sie diese Aufgabe mit Hilfe verschachtelter For-Schleifen lösen. Sie können das Problem alternativ aber auch mit einem recht kurzen „Einzeiler“ lösen.

9. Kontrollstruktur: While-Schleife

Eine *While-Schleife* erlaubt ähnlich wie eine For-Schleife das wiederholte Ausführen von Programm-Codes. Der Unterschied zur For-Schleife besteht allerdings darin, dass hier nicht über einen Container iteriert wird, sondern eine Abbruch-Bedingung spezifiziert wird. Hir ein Beispiel:

```
% Dieses Programm ermittelt wie oft man 1 verdoppeln muss,
% bis man mindestens so gross wie x ist.
x        = 213452;
counter  = 0;
multiple = 1;

% Solange multiple kleiner oder gleich x ist...
while multiple < x
  % ... setze den Zaehler um eins hoch...
  counter  = counter+1;
```

```
11
12    % ... und verdopple multiple.
13    multiple  = multiple*2;
14
15    % Falls wir zu viele Verdopplungsschritte gebraucht haben,
16    % ziehen wir die Reissleine und brechen ab.
17    if counter > 50
18      display('Zu viele Verdopplungen. Wir brechen ab.');
19      break;
20    end
21  end
22
23  % Abschliessend praesentieren wir das Ergebnis.
24  display( 'Anzahl durchgefuehrter Verdopplungen:' );
25  display(counter);
```

Auch in While-Schleifen können Sie die Schlüsselwörter `break` und `continue` verwenden, die Sie bereits für For-Schleifen kennengelernt haben.

> **Wichtig:** Achten Sie unbedingt darauf, dass die Abbruchbedingung in endlich vielen Schritten erreicht wird, ansonsten riskieren Sie eine Endlosschleife. Aus diesem Grund ersetzt man in vielen praktischen Anwendungen eine potentielle While-Schleife durch eine For-Schleife, in der man nur eine gewisse Höchstzahl von Iterationen erlaubt.

Aufgabe 24. Versuchen Sie, den Beispielcode zu verstehen und (vielleicht mit Hilfe eines Taschenrechners) das Endergebnis zu bestimmen. Vergleichen Sie Ihre Erwartung anschließend mit dem tatsächlichen Resultat.

Aufgabe 25. Schreiben Sie das Beispielprogramm so um, dass es anstelle einer `while`-Schleife eine `for`-Schleife verwendet.

Teil II.
Weitere Sprachelemente und Funktionen

10. Zeichenketten

Wenn Sie in MATLAB mit Texten oder anderen Zeichenketten arbeiten möchten, benötigen Sie in MATLAB sogenannte `char` Arrays. Im Rahmen dieser Einführung und der CoMa- und Numerik-Vorlesungen reicht es, nur die wesentlichen Grundlagen solcher Zeichenketten zu kennen.

Ein `char` Array lsst sich definieren, indem man den gewünschten Text mit einfachen Anführungszeichen umgibt.

Wichtig: Es müssen wirklich einfache Anführungszeichen sein, also ' und *nicht* ".

Ein weiteres nützliches Feature ist, dass `char` Arrays genauso wie andere Matrizen auch miteinander verkettet werden können. Außerdem ist zu beachten, dass bestimmte Sonderzeichen doppelt geschrieben (oder mit einem „\“ versehen) werden müssen, damit MATLAB den Text richtig interpretieren kann. Siehe dazu auch das folgende Beispiel:

```
% Textvariable definieren.
a   = 'Hallo Welt!';
% Enthaelt nun den Text: Hallo Welt!

% Verknuepfen von Texten.
b   = [a ' Lange nicht gesehen!'];
% Enthaelt nun: Hallo Welt! Lange nicht gesehen!

% Text mit Sonderzeichen.
c   = 'Einfach '', Prozent %% und Backslash \\';
% Enthaelt nun: Einfach ', Prozent % und Backslash \
```

Für weitere Funktionen und Beispiele im Zusammenhang mit Zeichenketten sei auf die Dokumentation

`http://mathworks.com/help/matlab/characters-and-strings.html`

verwiesen. Außerdem finden Sie unter

`https://mathworks.com/help/matlab/matlab_prog/matlab-operators-and-special-characters.html`

im Abschnitt „String and Character Formatting“ eine Liste alle Sonderzeichen.

11. Formatierte Textausgabe

Bisher haben wir zur Ausgabe von Variablen und Texten nur die Funktion `display` aus den Beispielcodes kennengelernt, die allerdings wenig bis keinen Spielraum lässt, um den Text selbst zu formatieren. In diesem Abschnitt erklären wir daher die Funktionen `fprintf` und `diary`, mit der Sie formatierte Textausgaben auf dem Command Window ausgeben und in Dateien schreiben können.

Die wesentliche Idee der formatierten Textausgabe von `fprintf` ist, eine *Formatierungs-Zeichenkette* vorzugeben, die einige Steuerungszeichen enthält sowie Platzhalterzeichen, die mit Variablenwerten substituiert werden können. Die wichtigsten Steuerzeichen sind:

\n	Zeilenumbruch erzeugen
\t	Tabulatur-Einzug erzeugen

Außerdem sind auch folgende Beispiele für Platzhalter relevant:

%6d	Ganzzahl der Länge 6
%8.5f	Fließkommazahl der Länge 8 mit 5 Nachkommastellen
%9.2e	Exponentielle Schreibweise der Länge 9 mit 2 Nachkommastellen
%10.5e	Darstellung 5 signifikanter Stellen in der kompaketeren Version %f bzw. %e
%s	Zeichenkette

Dabei ist zu beachten, dass Zahlen, die länger sind als es vorgegeben ist, extra Platz verbrauchen und nicht nur abgeschnitten werden. Andersherum werden Zahlen, die kürzer als vorgegeben sind, mit Leerzeichen aufgefüllt. Zur Illustration siehe das folgende Beispiel und dessen Ausgabe.

```
% Definiere einige Daten.
a    = 1:5;
b    = [1 -1/3 pi exp(1) 0];
c    = 100*b;

% Definiere die Zeichenkette zur Formatierung.
formatString = '%2d\t%6.3f\t%8.2e\t%8.2g\n';

% Gebe die Daten zeilenweise aus.
for i = 1:5
  fprintf( formatString, a(i), b(i), c(i), c(i) )
end
```

Ausgabe:

```
 1      1.000   1.00e+02        1e+02
 2     -0.333   -3.33e+01         -33
 3      3.142   3.14e+02      3.1e+02
 4      2.718   2.72e+02      2.7e+02
 5      0.000   0.00e+00            0
```

Sie können mit `fprintf` auch direkt in Dateien schreiben. Wichtig hierzu ist, dass sie mit Hilfe von `fopen` ein Handle auf eine Datei erzeugen, das sie anschließend der Funktion `fprintf` als erstes Argument übergeben. Sobald Sie mit den Arbeiten an der Datei fertig sind, können Sie die Ressource einfach mittels `fclose` wieder freigeben.

```
% Definiere einige Daten.
a = 0.5.^(0:4);

% Oeffne Datei zum schreiben.
% (Bereits existierende Dateien werden ueberschrieben.)
fId = fopen( 'listing1104.txt', 'w' );

% Schreibe Text in Datei:
```

```
for i = 1:5
  fprintf( fId, 'Wert fuer Zeile %i ist: %.4f.\n', i, a(i) );
end

% Gebe Dateihandle wieder frei.
fclose(fId);
```

Dies erstellt die Datei `listing1104.txt` mit Inhalt:

```
Wert fuer Zeile 1 ist: 1.0000.
Wert fuer Zeile 2 ist: 0.5000.
Wert fuer Zeile 3 ist: 0.2500.
Wert fuer Zeile 4 ist: 0.1250.
Wert fuer Zeile 5 ist: 0.0625.
```

Die Funktion `diary` erlaubt es, jegliche Textausgaben, die im Command Window angezeigt werden, außerdem auch an eine Textdatei weiterzuleiten.

```
% Waehle Speicherort fuer die Ausgabe.
diary('listing1106.txt');

% Aktiviere den diary Modus.
diary('on');

% Schreibe etwas in die Konsole.
fprintf('Hallo Welt!\n');

% Deaktiviere den diary Modus.
% Text wird nun nicht mehr gespeichert.
diary('off');
```

Dies gibt einerseits den Text „Hallo Welt!“ im Command Window aus und schreibt ihn zugleich in die Datei `listing1106.txt`.

Für eine ausführliche Diskussion von `fprintf` und `diary` sei auf die zugehörigen Dokumentationen

`https://mathworks.com/help/matlab/ref/fprintf.html`

und

`https://mathworks.com/help/matlab/ref/diary.html`

verwiesen.

12. Cell Arrays

Mit *Cell Arrays* ist es möglich, Listen unterschiedlicher oder „exotischer“ Datentypen zu erstellen. Typische Anwendungen können etwa sein, dass man eine Liste von Zeichenketten oder von Funktionenhandles speichern möchte. Ein Cell Array kann wie eine Matrix

definiert werden, nur dass geschweifte Klammern {} anstelle eckiger Klammern [] benutzt werden. Außerdem gibt es bei der Indizierung eine Besonderheit: Verwendet man runde Klammern () beim Indizieren, so ist das Ergebnis stets ein neues Cell Array. Mit runden Klammern kann man also Teil-Arrays des ursprünglichen Arrays erzeugen. Möchte man allerdings auf das eigentliche Element innerhalb eines Cell Arrays zugreifen, so ist stattdessen mittels geschweifter Klammern {} zu indizieren.

```
% Ein Cell Array mit Zeichenketten.
names   = {'Sinus', 'Cosinus', 'Logarithmus'};

% Ein Cell Arrray mit Funktionenhandles.
funcs   = {@sin, @cos, @log};

% Schleife ueber Eintraege.
for i = 1:length(names)
    fprintf( '%s ausgewertet in 1 ergibt %.3f.\n', ...
        names{i}, funcs{i}(1) );
end
```

Die hiervon erzeugte Ausgabe ist:

```
Sinus ausgewertet in 1 ergibt 0.841.
Cosinus ausgewertet in 1 ergibt 0.540.
Logarithmus ausgewertet in 1 ergibt 0.000.
```

Weitere Informationen zu Cell Arrays sind auf der Website

`https://mathworks.com/help/matlab/cell-arrays.html`

zu finden.

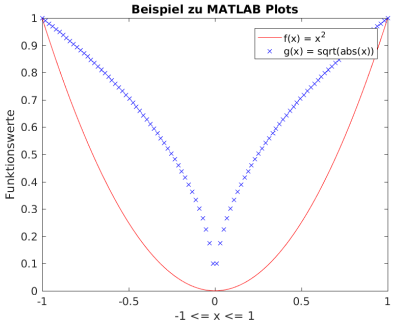
13. Plotten

Eine wichtige Standardaufgabe im Zusammenhang mit mathematischen Algorithmen ist das Plotten von Daten. MATLAB bietet hierzu umfangreiche Funktionen, allerdings wird sich diese Einführung nur darauf beschränken, das Wesentlichste hauptsächlich anhand einiger Code-Beispiele zu illustrieren. Die grundlegende Idee dabei ist einfach: Mit `figure` erstellt man eine neue, leere Abbildung, die anschließend mit Plot-Befehlen befüllt werden kann. Diese Plot-Befehle unterstützen meist eine Vielfalt verschiedener Optionen, um das genaue Erscheinungsbild zu kontrollieren. Möchte man mehrere Plots in derselben Grafik unterbringen, so verwendet man den Befehl `hold on`. Mittels Befehlen wie etwa `title` und `legend` ist es dann möglich, allgemeine Informationen zur Grafik hinzuzufügen. Abschließend kann die erstellte Abbildung mittels der Funktion `print` als Datei abgespeichert werden.

```
% Erstelle ein neues Abbildungs-Handle.
% Wir merken uns dieses in einer Variable.
```

```
3  h = figure;
4
5  % Optional: Mache die Abbildung unsichtbar.
6  set(h, 'Visible', 'off');
7
8  % Definiere die zu plottenden Funktionen.
9  f = @(x) x.^2;
10 g = @(x) sqrt(abs(x));
11
12 % Definiere die Menge aller auszuwertenden Punkte.
13 % Wir waehlen das Intervall [-1,1] mit 100
14 % gleichmaessig verteilten Punkten.
15 X   = linspace(-1,1,100);
16
17 % Plotte die Funktion f ueber X in roter Farbe als Linie.
18 plot( X, f(X), 'color', 'red' );
19
20 % Wir wollen weitere Plots hinzufuegen:
21 hold on;
22
23 % Plotte die Funktion g ueber X in blauer Farbe als Kreuze.
24 plot( X, g(X), 'x', 'color', 'blue' );
25
26 % Fuege einen Titel zur Abbildung hinzu.
27 title( 'Beispiel zu MATLAB Plots' );
28
29 % Beschrifte die Achsen.
30 xlabel( '-1 <= x <= 1' );
31 ylabel( 'Funktionswerte' );
32
33 % Fuege eine Legende hinzu.
34 legend( 'f(x) = x^2', 'g(x) = sqrt(abs(x))' );
35
36 % Speichere die Datei als PNG-Datei ab.
37 print( h, 'BeispielPlot.png', '-dpng' );
38
39 % Optional: Schliesse die Abbildung.
40 close(h);
```

Das Resultat sieht so aus:



Weitere Funktionen und Optionen zur grafischen Darstellung von Daten sind über die Seite

`https://mathworks.com/help/matlab/graphics.html`

in der Dokumentation zu finden.

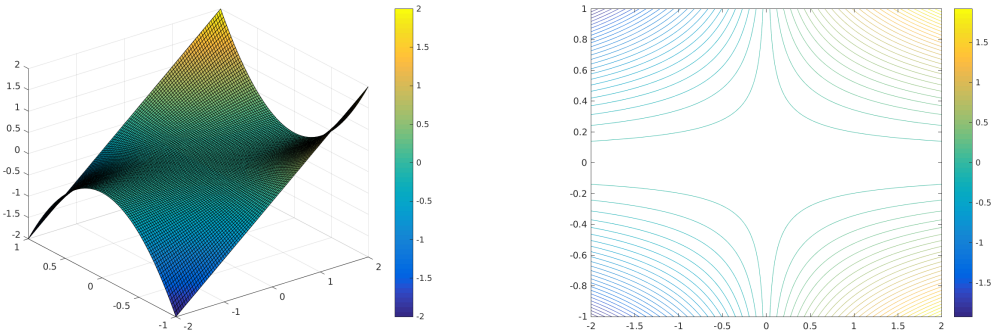
14. Zusatz: 2D/3D-Plots

Im vorangegangenen Abschnitt haben wir einige Grundlagen zum Plotten kennengelernt und ein Beispiel dafür gesehen, wie man eine eindimensionale Funktion $f\colon I \to \mathbb{R}$ plottet, wobei $I \subset \mathbb{R}$ ein Intervall ist. In diesem Abschnitt besprechen wir nun auch ein Beispiel für das Plotten zweidimensionaler Funktionen, genauer für den Spezialfall einer Funktion $f\colon I \times J \to \mathbb{R}$ mit Intervallen I und J. Dazu das folgende Beispiel:

```
% Definiere die zu plottende Funktion.
f = @(x,y) x.*y.^2;

% Definiere die Menge aller auszuwertenden Punkte.
% Wir waehlen das Gebiet [-2,2] x [-1,1] mit 101^2
% gleichmaessig verteilten Punkten.
I      = linspace(-2,2,100);
J      = linspace(-1,1,100);
[X,Y] = meshgrid(I,J);

% Erstelle eine Abbildung und mache diese unsichtbar.
h = figure;
set(h, 'Visible', 'off');

% Plotte den Funktionengraph als 3D-Objekt.
Z = f(X,Y);
surf(X, Y, Z);

% Fuege eine Farb-Legende hinzu.
colorbar;
```

```
21
22 % Speichere die Datei als PNG-Datei ab und schliesse h.
23 print( h, 'BeispielPlot_surf.png', '-dpng' );
24 close(h);
25
26
27 % Ein anderes Beispiel: Plotte nur die Hoehenlinien.
28 h = figure;
29 set(h, 'Visible', 'off');
30 contour(X, Y, Z, 50);
31 colorbar;
32 print( h, 'BeispielPlot_contour.png', '-dpng' );
33 close(h);
```

Die erzeugten Ausgaben sehen anschließend so aus:

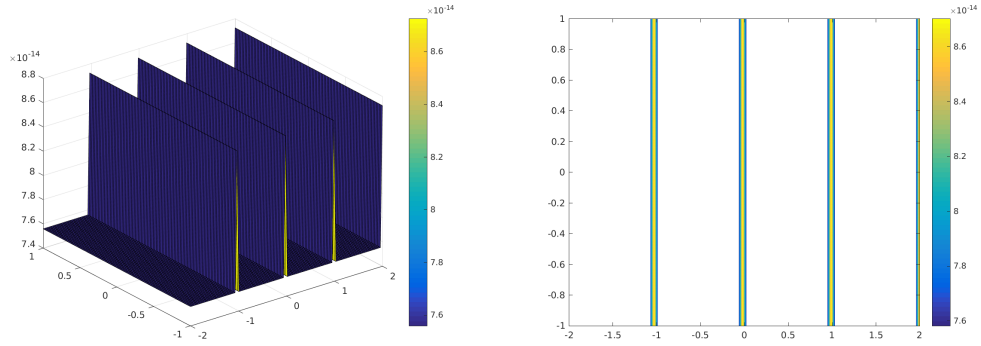


Die wesentliche Idee im Code ist also auch hier wie im eindimensionalen Fall, dass man sich Container mit den Inhalten aller zu plottenden Punkte definiert. Im Beispiel sind das die Matrizen `X`, `Y` und `Z`. Für den Spezialfall, dass die zu plottenden Punkte durch eine Funktion `f` beschrieben werden können und dass diese Funktion in einer geeignet vektorisierten Form vorliegt, kann man die Matrix `Z` durch einen Aufruf der Form `Z = f(X,Y)` berechnen.

Folgendes ist hier zu beachten:

- Der verwendete Plot-Befehl benötigt *Matrizen* `X` und `Y` als Eingabedaten, nicht die Vektoren `I` und `J`. Diese Matrizen kann man aus `I` und `J` generieren, indem man `[X,Y] = meshgrid(I,J)` aufruft. Für mehr Details sei an dieser Stelle auf die MATLAB-Dokumentation verwiesen.

- Man muss aufpassen, dass die zu plottende Funktion wirklich sinnvoll vektorisiert ist. Beispielsweise führt zwar die Definition `f = @(x,y) x*y^2` anstelle der Definition `f = @(x,y) x.*y.^2` zu einem fehlerfreien Code, aber die Ausgabe sieht anders aus als erwartet:



Achtung: Manchmal kann es wesentlich weniger offensichtlich sein, dass die Ausgabe falsch ist.

Abschließend sei noch angemerkt, dass man auch Funktionen plotten kann, die nicht über einem kartesischen Produkt der Form $I \times J$ definiert sind. Ebenso muss man auch nicht immer ein gleichmäßiges Gitter für die Auswertungen verwenden. Dieses Thema führt aber über dieses Skript hinaus, weswegen wir auch hier wieder auf die Dokumentation verweisen. Ein interessantes Stichwort für den Anfang ist hier beipsielsweise die Funktion `trisurf`.

Teil III.
Sonstiges und allgemeine Hinweise

- Als kostenlose Alternative zu MATLAB gibt es auch das Programm Octave, siehe `https://www.gnu.org/software/octave/`. Die Befehle und Syntax von Octave sind in den allermeisten Fällen identisch zu MATLAB, es gibt jedoch auch kleinere Unterschiede. Achten Sie beim Bearbeiten Ihrer Programmieraufgaben daher darauf, dass Ihre finale Version auch unter MATLAB einwandfrei lauffähig ist.

- Unter `Preferences > Keyboard > Shortcuts` können Sie die Tastenkürzel von `Emacs` auf `Windows` umstellen, falls Ihnen letztere vertrauter als erstere sind.

- Nehmen Sie sich die Zeit, um bei Gelegenheit die Liste der verfügbaren Tastenkürzel durchzulesen. Der sicherer Umgang mit Tastenkürzeln ist ein Schlüssel zu einer effizienteren Arbeitsweise.

- Ziehen Sie in Betracht, beim Programmieren das englische QWERTY-Layout für Ihre Tastatur zu verwenden. Im QWERTY-Layout sind Klammern wesentlich einfacher gelegen und können somit auch das Programmieren erleichtern. Unter Debian können Sie das Layout hinzufügen, indem Sie die `Windowstaste` drücken, dann `Region und Sprache` eingeben, auf den entsprechenden Menupunkt klicken

und schließlich `English (USA)` unter `Eingabequellen` hinzufügen. Sie können anschließend mittels `Windowstaste + Leertaste` zwischen dem deutschen und dem englischen Layout hin- und herwechseln. [5]

- Bevor Sie ein Programm abgeben (oder wenn Sie auf Fehlersuche sind), setzen Sie den Befehl `clear variables` an den Anfang der Skript-Datei, das Ihr Programm ausführt, und testen Sie Ihren Code anschließend erneut. Dieser Befehl löscht alle in der Sitzung erstellten Variablen und vermeidet so, dass Ihr Programm auf unerwünschte Weise auf Werte von Variablen zugreift, die es nicht selbst definiert hat.

- Mit `close all` können Sie alle Fenster schließen, die Sie beim Plotten geöffnet haben.

- Mit `Strg + C` können Sie eine laufende Programmausführung abbrechen. Das ist insbesondere dann nützlich, wenn sich Ihr Programm in einer Endlosschleife verfangen hat oder ein Problem unerwartet viel Rechenzeit benötigt.

- Wenn Sie einen ganzen Code-Block auskommentieren möchten, haben Sie zwei Möglichkeiten:
  - Markieren Sie den entsprechenden Block und drücken Sie `Strg + R`. Dadurch wird ein Prozentzeichen `%` an den Anfang jeder Zeile gesetzt. Mittels `Strg + T` können Sie die Prozentzeichen an den Zeilenanfängen ihrer aktiven Auswahl wieder entfernen. (Achtung: Für Octave lauten die Kürzel womöglich anders.)
  - Fügen Sie *vor* den auszukommentierenden Code-Block eine Zeile mit dem Inhalt `%{` und *nach* dem Block eine Zeile mit dem Inhalt `%}` hinzu. Der Code zwischen diesen beiden Zeilen ist nun auskommentiert. Sie können den Vorgang rückgängig machen, indem Sie diese beiden Zeilen wieder entfernen.

- Mit ... (drei Punkte in Folge) am Ende einer Zeile können Sie MATLAB anweisen, den nächsten Zeilenumbruch zu ignorieren und stattdessen die aktuelle Zeile sowie die darauf folgende als eine einzige lange Zeile zu sehen. Dies ist besonders nützlich um Code übersichtlich zu halten, indem es Ihnen beispielsweise erlaubt, Zeilenumbrüche für Funktionsaufrufe mit vielen oder sehr langen Parametern einzubauen.

[5] Unter Linux wird die `Windowstaste` abweichend vom allgemeinen Sprachgebrauch häufig mit `Super` bezeichnet.