Brandenburgische
Technische Universität
Cottbus

Lehrstuhl Mathematik, insbesondere
Numerische und Angewandte Mathematik
Prof. Dr. L. Cromme

Hinweise zur Arbeit mit Matlab

MATLAB ist ein Softwarepaket zur Berechnung, Visualisierung und auch zur Programmierung technisch-wissenschaftlicher, insbesondere mathematischer Ausdrücke. MATLAB wird derzeit durch die Firma „The Mathworks“ (`www.mathworks.com`) vertrieben und weiterentwickelt.

Nachfolgend sollen nur einige allgemeine, sehr knapp gehaltene Hinweise zu MATLAB gegeben werden, wie sie zur Lösung von Programmieraufgaben zu Numerikvorlesungen und Modellierungspraktika hilfreich sein können. In erster Linie sollen die nachfolgenden Zeilen eine Hilfestellung für alle sein, die bisher nur selten mit Matlab gearbeitet haben, mit den allgemeinen Grundlagen (wie z.B. der Umgang mit Matrizen und den dazugehörigen Rechenoperationen) aber bereits vertraut sind.

Weiterführende Informationen und Details sind der umfangreichen Matlab-Dokumentation zu entnehmen. Alternativ sei zum Beispiel auf die unten angegebenen Bücher sowie die angegebenen Internetlinks verwiesen. Diese behandeln zwar zum Teil ältere Matlab-Versionen, jedoch bestehen zwischen älteren und neueren Versionen bei den grundlegenden Anweisungen kaum nennenswerte Unterschiede.

1 Allgemeine Grundlagen

Nach dem Start[1] von MATLAB erscheint ein Fenster mit dem Namen `Command Window`. (Bem.: Sollte das `Command Window` nicht sichtbar sein, so kann es über das Menü `View` aufgerufen werden.) Hier können hinter dem `>>`-Zeichen MATLAB-Kommandos eingegeben werden, wobei auf weiter zurückliegende Eingaben mit den „Auf/Ab“-Pfeiltasten zugegriffen werden kann. Man kann die Ergebnisse der Berechnungen Variablen mit beliebigen Namen zuweisen, ist kein Variablenname vorhanden, dann generiert MATLAB die Variable `ans` und weist dieser das Ergebnis zu. Die Zuweisung erfolgt mit Hilfe des Gleichheitszeichens. Grundsätzlich werden die Berechnungsergebnisse angezeigt, dies kann aber durch anhängen eines Semikolons am Ende des Befehls unterdrückt werden. Jede Eingabe ist mit ENTER abzuschließen. Wie in jedem ähnlichen Softwarepaket werden auch hier die normalen Operatoren (`+,-,*,/`) unterstützt. Variablennamen dürfen maximal 31 Zeichen lang sein, zusätzlich wird zwischen Groß- und Kleinschreibung unterschieden. Bei Dezimalzahlen ist ein Punkt als Trennzeichen zu verwenden.

1.1 Matrizen

Grundlegend ist für die Arbeit mit Matlab der Umgang mit Matrizen. Matrizen können auf verschiedene Weisen eingeben werden, zum Beispiel:

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

oder

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

oder

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

[1] In den Rechnerpools des Instituts für Mathematik der BTU Cottbus wird Matlab einfach durch Eingabe von „Matlab plus ENTER“ im Eingabefenster (Host) gestartet.

In allen drei Fällen erhält man als Ergebnis die Matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}.$$

Bei einzelnen Werten handelt es sich also im Prinzip um (1×1)-Matrizen, bei denen allerdings die eckigen Klammern nicht mit eingegeben werden müssen. Eine $(m \times n)$ Nullmatrix erhält man durch den Befehl `zeros(m,n)`. Auf einzelne Elemente der Matrix wird durch Angabe des Zeilen- und Spaltenindex in Klammern hinter dem Variablennamen zugegriffen, d.h. z.B. `A(1,1)`. Auf gleiche Weise können Werte zugewiesen werden. Zum Zugriff auf die i-te Zeile wird der Spaltenindex durch einen Doppelpunkt ersetzt, d.h. `A(1,:)` liefert die erste Zeile von A. Analog erhält man durch `A(:,1)` die erste Spalte von A. Teile von Spalten oder Zeilen werden durch Angabe der entsprechenden Spalten- bzw. Zeilenindizes referenziert, z.B.: `A(1:2,:)` liefert die ersten zwei Zeilen von A und `A(:,[1,3])` liefert die erste und dritte Spalte von A und.

Eine „leere“ Matrix wird durch die Symbolik `[]` ausgedrückt, zur Transposition einer Matrix gibt es die Funktion `transpose.m`. Bei der Arbeit mit Matrizen ist stets auf zueinander „passende“ Zeilen- und Spaltendimensionen zu achten! Zur Addition und Subtraktion werden einfach die Operatoren `+` bzw. `-` verwendet. Die Matrixmultiplikation $C = A \cdot B$ mit $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times k}$ mit dem Ergebnis $C \in \mathbb{R}^{m \times k}$ mit $c_{ij} = \sum_{l=1}^{n} a_{il} \cdot b_{lj}$, $i = 1, \ldots, m$, $j = 1, \ldots, k$, wird durch die Eingabe

```
C = A*B;
```

realisiert. Für quadratische Matrizen $A \in \mathbb{R}^{n \times n}$ kann statt `A*A` auch `A^2` eingegeben werden. Im Unterschied dazu steht die *elementweise* Multiplikation von $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{m \times n}$, was durch die Eingabe

```
C = A.*B;
```

realisiert wird. Die Elemente der Ergebnismatrix $C \in \mathbb{R}^{m \times n}$ sind hierbei definiert durch $c_{ij} = a_{ij} \cdot b_{ij}$ für $i = 1, \ldots, m$ und $j = 1, \ldots, n$. Analog sind die elementweise Division (`C = A./b`) bzw. die elementweise Potenz (z.B. `C = A.^2`) durchführbar.

1.2 Zeichen(ketten)

Hilfreich kann auch die Verwendung von Zeichenketten sein, die durch je ein Hochkomma am Anfang und am Ende kenntlich gemacht werden, zum Beispiel

```
zk = 'Heute ist das Wetter schöner als gestern';
```

Bei Zeichenketten handelt es sich um Vektoren (also spezielle Matrizen) vom MATLAB-Datentyp `char`. Es kann also auf einzelne Elemente (also Zeichen) mit den für Matrizen beschriebenen Methoden zugegriffen werden. Zur Bearbeitung von Zeichen(ketten) stellt Matlab diverse Funktionen bereit, auf die hier aber nicht näher eingegangen werden soll.

1.3 Cell-Arrays

Matrizen zeichnen sich dadurch aus, dass ihre Einträge alle vom gleichen Datentyp sind, d.h. z.B. numerische Werte. Um zum Beispiel Matrizen unterschiedlichen Typs und unterschiedlicher Größe im Rahmen der Programmereigenschaften von Matlab zwischen Funktionen austauschen zu können (siehe unten), ist es aus Gründen der Übersichtlichkeit ggf. hilfreich, diese nicht alle einzeln, sondern in kompakter Form zu transportieren. Eine Möglichkeit besteht dabei in der Verwendung von sog. `cell`-Arrays. Wie die Bezeichnung bereits vermuten läßt, handelt es sich hierbei um eine Sonderform einer Matrix. `Cell`-Arrays zeichnen sich dadurch aus, dass ihre Einträge unterschiedliche Größen und unterschiedliche Datentypen haben können. Anschaulich kann man also dafür auch den Begriff des „Datencontainers“ verwenden.

Ein leeres $n \times m$ `cell`-Array (also bestehend aus n Zeilen und m Spalten) wird wie folgt erzeugt:

```
C = cell(n,m);
```

Der Zugriff (ein- und auslesen) von einzelnen Elementen erfolgt mit Hilfe von *geschweiften* Klammern, dazu ein Beispiel:

```
C = cell(1,4);
C{1,1} = [1,2,3];
C{1,2} = [1,2,3 ; 4,5,6; 7,8,9];
C{1,3} = cell(2,3);
C{1,4} = 'zeichenkette';
```

Ist $n = 1$ bzw. $m = 1$, so kann die Indizierung der einzigen Zeile bzw. Spalte auch weggelassen werden, d.h. mit

```
C = cell(1,4);
C{1} = [1,2,3];
C{2} = [1,2,3 ; 4,5,6; 7,8,9];
C{3} = cell(2,3);
C{4} = 'zeichenkette';
```

wird das gleiche Ergebnis erzielt wie oben. Weitere äquivalente Eingaben (für das obige Beispiel) sind die folgenden beiden Möglichkeiten:

```
C = {[1,2,3] , [1,2,3 ; 4,5,6; 7,8,9] , cell(2,3) , 'zeichenkette'};
```

oder

```
C = {};
C = [C , {[1,2,3]}];
C = [C , {[1,2,3 ; 4,5,6; 7,8,9]}];
C = [C , {cell(2,3)},{'zeichenkette'}];
```

Die letztere Möglichkeit ist vor allem dann hilfreich, wenn man die genaue Länge des `cell`-Vektors `C` vorab nicht kennt. Ist $n \neq 1$ und gleichzeitig $m \neq 1$, so gilt wie bei den oben behandelten Matrizen, dass jede der n Zeilen auch m Spalten haben muss. Auch das Auslesen einzelner Einträge erfolgt mit geschweiften Klammern, z.B.

```
Z = C{4};
```

Achtung! Der Zugriff auf Einträge eines `cell`-Arrays mit *runden* Klammern bewirkt, dass von den entsprechend referenzierten Einträgen nur eine Kopie angefertigt wird. So ist bezogen auf das obige Beispiel das durch

```
P = C(4);
```

erzeugte Objekt `P` ebenfalls ein `cell`-Array mit $n = m = 1$. Der Zugriff auf seinen Inhalt ist wieder über geschweifte Klammern möglich, d.h.

```
Z = P{1};
```

1.4 Datenstrukturen

Eine weitere Möglichkeit, Daten unterschiedlichen Typs und unterschiedlicher Größe in einem Objekt zusammenzufassen besteht in der Verwendung von sog. Datenstrukturen (Matlab-Datentyp `struct`). Dabei hat die Datenstruktur einen (variablen) Namen, dem beliebig viele Felder (Objekte) mit (festen) Variablennamen zugeordnet sind, d.h allgemein:

```
objekt.feld1 = ...
objekt.feld2 = ...
objekt.feld3 = ...
...
```

Der Inhalt des oben als Beispiel behandelten `cell`-Arrays könnte als Datenstruktur damit wie folgt gespeichert werden:

```
objekt.vektor = [1,2,3];
objekt.matrix = [1,2,3 ; 4,5,6; 7,8,9];
objekt.datencontainer = cell(2,3);
objekt.name = 'zeichenkette';
```

Der Zugriff auf die einzelnen Felder der Datenstruktur `objekt` erfolgt einfach durch die Angabe des Feldnamens, zum Beispiel

```
a = objekt.vektor;
z = objekt.name;
```

Eine Zuweisung der Gestalt

```
kopieDerDatenstruktur = objekt;
```

bewirkt, dass eine Kopie der Datenstruktur `objekt` erzeugt wird, wobei die Variablennamen der Felder der Datenstruktur unverändert bleiben, d.h. bezogen auf das Beispiel haben die Datenstrukturen `objekt` und `kopieDerDatenstruktur` die gleichen Felder `vektor, matrix, datencontainer` und `name`.

Die Felder einer Datenstruktur haben außerdem keinen festen Datentyp, d.h. dieser kann jederzeit durch Zuweisung eines beliebigen Objektes (Matrix, numerischer Wert, Zeichenkette, Datenstruktur, `cell`-Array, ...) geändert werden. (Hinweis: Analoges gilt auch für die oben genannten `cell`-Arrays.) Dazu ein Beispiel: Das Feld `datencontainer` der Datenstruktur `kopieDerDatenstruktur` hat im obigen Beispiel ein `cell`-Array zum Inhalt. Die Anweisung

```
kopieDerDatenstruktur.datencontainer = objekt;
```

hinterlegt im Feld `datencontainer` der Struktur `kopieDerDatenstruktur` eine Kopie der Datenstruktur `objekt`, d.h. das Feld `datencontainer` wird damit nun selbst zur Datenstruktur.

2 Funktionen

MATLAB stellt eine Vielzahl einfacher und komplexerer Funktionen zur Verfügung. Zu den einfachen gehören zum Beispiel `sin, cos, exp, sqrt, sum, mean, setdiff`, komplexere sind zum Beispiel Funktionen zur Lösung von Optimierungsproblemen wie etwa `polyfit, fminsearch, lsqnonlin` usw.

2.1 Definition einer Funktion

Es besteht aber auch die Möglichkeit, selbst Funktionen zu programmieren, die sog. `m`-Files. Diese lassen sich mit Hilfe eines (beliebigen) Texteditors schreiben und haben folgenden Grundaufbau: Der Kopf der Funktion definiert diese und sieht wie folgt aus:

```
function[rueckgabeparameter] = funktionsname(eingabeparamater1,eingabeparameter2,...)
```

Soll die Funktion keinen Rückgabewert haben, dann lässt man den Rückgabeparameter einfach weg, die eckigen Klammern müssen aber stehen bleiben, d.h. zum Beispiel

```
function[] = funktionsname(eingabeparamater1,eingabeparameter2,...)
```

Sollen der Funktion keine Eingabeparameter übergeben werden, dann lässt man diese einschließlich der runden Klammern einfach weg, zum Beispiel

```
function[rueckgabeparameter] = funktionsname
```

Dem Funktionskopf folgt optional eine Beschreibung (wobei jeder Zeile ein Prozentzeichen `%` vorangestellt werden muss, womit diese Zeilen als Kommentar identifiziert werden!), woran sich unmittelbar der Anweisungsteil anschließt, wo die Berechnungen durchgeführt werden. Soll ein Ergebnis zurückgegeben werden, so ist dieses der Variablen `rueckgabeparameter` zuzuweisen. Im Anweisungteil können alle beliebigen MATLAB-Anweisungen verwendet werden, insbesondere auch andere Funktionen selbst.

2.2 Programmierung eigener Funktionen

Durch einige Anweisungen wird „richtiges“ programmieren möglich, zum Beispiel die wichtige `if-else` Anweisung zur Fallunterscheidung mit der (Grund)syntax

```
if Bedingung1
   Anweisungen
elseif Bedingung2
   Anweisungen
else
   Anweisungen
end
```

Als Bedingungen sind dabei logische Ausdrücke zu verwenden, die zum Beispiel durch Vergleich zweier Werte entstehen und die Werte 0 oder 1 annehmen können. Als Vergleichsoperatoren sind dabei am wichtigsten: `==` (Gleichheit), `<=` bzw. `<` (kleiner gleich bzw. kleiner), `>=` bzw. `>` (größer gleich bzw. größer). Für das logische NICHT sollte die `not`-Funktion verwendetet werden. Logische Bedingungen können (allerdings immer nur je zwei, durch Komma getrennt) durch die Funktionen `and` und `or` miteinander verknüpft werden. Wichtig ist auch die `for`-Schleife mit folgender Syntax:

```
for index=Anfangswert:Schrittweite:Endwert
   Anweisungen
end
```

Wird die Schrittweite nicht angegeben, dann wird diese automatisch auf 1 gesetzt. Die Variable `index` kann im Anweisungsteil wie eine „normale“ Variable verwendet werden, ihr Wert wird mit jedem Schritt um die angegebene Schrittweite erhöht. Weitere Kontrollanweisungen zum Programmieren sind analog zu vielen Programmiersprachen ebenfalls verwendbar, wie zum Beispiel die `while`-Schleife, `switch-case`-Anweisung, `try-catch`-Anweisung usw.

Die definierte Funktion wird unter ihrem Namen und mit der Dateiendung `m` gespeichert, zum Beispiel würde die Funktion

```
funktion[summe] = summiere(vektor)
   summe = 0;
   for i=1:length(vektor)
      summe = summe + vektor(i);
   end
```

unter dem Dateinamen `summiere.m` gespeichert. Möchte man die Funktion aufrufen, so muss zunächst im Fenster `Current Directory` das Verzeichnis, in dem die Datei gespeichert wurde, als aktuelles Arbeitsverzeichnis aufgerufen werden. Sollten selbstgeschriebene Funktionen ihrerseits weitere selbstgeschriebene Funktionen verwenden, dann müssen diese im gleichen Verzeichnis liegen. (Bem.: Sollte das `Current Directory`-Fenster nicht sichtbar sein, so kann es über das Menü `View` aufgerufen werden.) Der Aufruf der Funktion erfolgt nun über den Dateinamen und Übergabe der entsprechenden Parameter, aber OHNE die Dateiendung `m`, zum Beispiel

```
s = summiere([1,2,3]);
```

Der Variablen `s` wird dann das Ergebnis zugewiesen, hier $s = 6$.

Oft kennt man im Voraus die genaue Anzahl von Parametern nicht, die einer Funktion zum Zeitpunkt ihres Aufrufs übergeben werden. Umgekehrt kann es hilfreich sein, dass auch die Anzahl der Rückgabewerte variabel ist. Hierfür stellt Matlab die Variablen `varargin` und `varargout` zur Verfügung, bei denen es sich um `cell`-Arrays handelt. Die Anzahl der an eine Funktion übergebenen Parameter läßt sich mit dem Befehl `nargin`,

die der ausgehenden Parameter mit `nargout` bestimmen. Neben `varargin` können auch weitere Parameter übergeben werden, wobei zu beachten ist, dass `varargin` immer am Ende der Parameterliste steht, d.h. die Definition

```
funktion[wert] = testfkt(a,b,varargin,c)
```

ist *falsch* und würde bei einem Funktionsaufruf zu einer Fehlermeldung führen! Richtig muss die Funktion wie folgt definiert werden:

```
funktion[wert] = testfkt(a,b,c,varargin)
```

Bei der Anzahl der eingehenden Parameter zählt der Befehl `nargin` *alle* übergebenen Variablen mit und beschränkt sich nicht nur auf das `cell`-Array `varargin`. Beispiele zur eben definierten Funktion `testfkt.m`: Beim Aufruf `testfkt(1,2,3)` liefert `n=nargin` den Wert `n=3`, wobei `length(varargin)=0` ist, d.h. varargin ist leer. Dagegen liefert der Aufruf `testfkt(1,2,3,4,5,6)` bei Anwendung von `n=nargin` den Wert `n=6`, wobei `length(varargin)=3` ist, d.h. varargin enthält drei Elemente. Die Elemente in `varargin` können einen beliebigen Typ haben. Man sollte stets darauf achten, dass innerhalb der Funktion selbst mit Hilfe von Kontrollanweisungen immer auf die variable Anzahl von Parametern reagiert wird. Analog gilt dies auch für die Rückgabewerte im Datencontainer `varargout`.

2.3 Logische Indizierung

Ein sehr effizientes Hilfsmittel bei der Programmierung eigener Funktionen stellt die sog. logische Indizierung dar, mit der schnell alle Elemente (bzw. ihre Indizes) eines Vektors identifiziert werden können, die bestimmte Eigenschaften (erklärt durch logische Ausdrücke) erfüllen. Dazu ein Beispiel: Aus dem Vektor

```
v = [1:1:50,75:1:90,100];
```

sollen alle durch fünf teilbaren Zahlen isoliert und einer Variablen v2 zugeordnet werden. Mit Hilfe der üblichen `for`- und `if-else`-Anweisungen würde dies zum Beispiel wie folgt aussehen:

```
v2 = [];
for (i=1:length(v))
  if (mod(v(i),5) == 0)
    v2 = [v2 , v(i)];
  end
end
```

Die elementweise Abfrage, ob das i-te Element von `v` ohne Rest durch fünf teilbar ist sowie die Aktualisierung und damit Neudefinition des Vektors `v2` ist recht zeitintensiv. Dies ist in diesem Beispiel zwar sicherlich am Rechner kaum messbar, spielt aber für größere Probleme mit längeren Vektoren (Matrizen) und häufigen Abfragen eine wichtige Rolle. Schneller wird ein entsprechendes Programm unter Verwendung der logischen Indizierung, die für das Beispiel wie folgt aussieht:

```
indizes = 1:1:length(v);          % Indizes aller Elemente von v
ind = indizes(mod(v,5) == 0);     % Indizes aller Elemente von v, die durch 5 teilbar sind
v2 = v(ind);                      % alle durch 5 teilbaren Elemente von v
```

Falls kein Element des Vektor `v` die geforderte logische Bedingung erfüllen würde, so würde `ind = []` als Ergebnis zurückgegeben werden (was aber im Beispiel natürlich nicht der Fall ist ...). Kombinationen von einzelnen logischen Ausdrücken lassen sich wie üblich mit Hilfe der Funktionen `and, or` und `not` realisieren.

2.4 Reellwertige Funktionen

Zur Auswertung, Differentiation und Integration reellwertiger Funktionen stellt Matlab mit der Symbolic Toolbox sehr vielfältige Möglichkeiten bereit. Der Umgang damit sei am Beispiel der Funktion $f(x) = xe^x$ und ihrer Ableitung $f'(x) = xe^x + e^x$ angeschnitten:

```
syms x                              % Definition der Variablen
f = x*exp(x);                       % Definition der Funktion f
fstrich = diff(f,x,1)               % erste Ableitung von f nach x
bildf = subs(f,x,-1:0.0001:1);      % Auswertung von f auf dem Gitter -1:0.0001:1;
wertfstrich = subs(fstrich,x,10);   % Auswertung von fstrich in x=10;
```

Analog werden Funktionen in mehreren Veränderlichen definiert, zum Beispiel $f(x) = xe^{yz} + zx$:

```
syms x y z              % Definition der Variablen
f = x*exp(y*z); + z*x;  % Definition der Funktion f
fx = diff(f,x,1);       % erste Ableitung von f nach x
fy2 = diff(f,y,2);      % zweite Ableitung von f nach y
fz5 = diff(f,z,5);      % fünfte Ableitung von f nach z
```

Eine Vielzahl von Funktionen der Symbolic Toolbox bieten sehr einfache Möglichkeiten zum Umgang mit reellwertigen und auch komplexen Funktionen. Ein entscheidender Nachteil kann aber sein, dass das „symbolische Rechnen“ sehr zeitaufwändig wird, zum Beispiel wenn eine Funktion zur Laufzeit eines Matlab-Programm sehr oft oder an sehr vielen Stellen ausgewertet werden muss. Dazu ist es besser, die Funktion separat und mit Hilfe von Matrizen zu definieren, am Beispiel der Funktion $f(x) = xe^x$ sieht das dann wie folgt aus, wobei der Übergabeparameter `x` ein Vektor ist:

```
funktion[werte] = testfkt(x)
  werte = x.*exp(x);
```

Analog für die Ableitung:

```
funktion[ablwerte] = testfktAbleitung(x)
  ablwerte = x.*exp(x) + exp(x);
```

Es kann sinnvoll sein, die Berechnung von Funktionswerten der Funktion selbst und ihrer Ableitung gleichzeitig zu berechnen, dann kann man die beiden Funktionen auch zusammenfassen:

```
funktion[werte,ablwerte] = testfkt2(x)
  werte = x.*exp(x);
  ablwerte = x.*exp(x) + exp(x);
```

Beim Aufruf der Funktion mit `werte = testfkt2(x)` werden dann nur die Funktionswerte $f(x)$ zurückgegeben, während mit `[werte,ablwerte] = testfkt2(x)` sowohl $f(x)$ (im Vektor `werte`) als auch $f'(x)$ (im Vektor `ablwerte`) zurückgegeben werden. Mit Hilfe der logischen Indizierung lassen sich bequem auch stückweise definierte Funktionen definieren. Beispiel für die stetige, aber in $x = 0$ nicht stetig differenzierbare Funktion

$$f(x) = \begin{cases} x & , \quad x \leq 0 \\ x^2 & , \quad x > 0 \end{cases}$$

Die entsprechende Matlab-Funktion zur Auswertung der Funktion f und bei Bedarf auch zur Auswertung ihrer ersten Ableitung kann zum Beispiel wie folgt aussehen:

```
funktion[werte,ablwerte] = testfkt3(x)
  n = length(x);
  indizes = 1:n;
  werte = zeros(1,n);
  ablwerte = zeros(1,n);
  ind = indizes(x < 0);
  werte(ind) = x(ind);
  ablwerte(ind) = 1;
  ind = indizes(x == 0);
  werte(ind) = 0;
  ablwerte(ind) = NaN;
  ind = werte(x < 0);
  werte(ind) = x(ind).^2;
  ablwerte(ind) = 2*x(ind);
```

Da die Ableitung in $x = 0$ nicht existiert, wurde der entsprechende Werte hier gleich der Matlab-Konstanten `NaN` (=**N**ot-**a**-**N**umber) gesetzt.

2.5 Allgemeine Auswertung von Matlab-Funktionen

Zur Auswertung einer beliebigen Funktion

```
funktion[y1,y2,y3,...] = funktionsname(x1,x2,x3,...)
  ...
```

gibt es die Funktion `feval.m`, der mit Hilfe des `@`-Operators ein Verweis auf die Funktion sowie ihre Parameter übergeben werden, d.h. der Standardaufruf

```
[y1,y2,y3,...] = funktionsname(x1,x2,x3,...)
```

kann auch durch

```
[y1,y2,y3,...] = feval(@funktionsname,x1,x2,x3,...)
```

ersetzt werden. Diese allgemeine Form des Funktionsaufrufs ist zum Beispiel dann vorteilhaft und/oder notwendig, wenn die genaue Festlegung, welche Funktion verwendet werden soll, immer erst zur Laufzeit erfolgt, entweder durch den Anwender oder durch Matlab selbst. Ein Beispiel dafür ist die Matlab-Funktion `lsqnonlin.m` zur Lösung von Optimierungsproblemen in der l_2-Norm, der nach eben beschriebenen Verfahren eine beliebige Zielfunktion zur Minimierung übergeben werden kann.

3 Einlesen und Ausgabe von/in `xls`- und `csv`-Dateien

Für das Einlesen von Matrizen, die in `xls`-Dateien (EXCEL-Dateien) abgespeichert sind, gibt es den Befehl `xlsread`, dem der Dateiname als Zeichenkette zu übergeben ist, z.B.

```
matrix = xlsread('..\Daten\werte.xls');,
```

wobei in dieser Form immer nur das erste Arbeitsblatt eingelesen wird. Möchte man weitere Arbeitsblätter einlesen, so ist dessen Name als Zeichenkette mit anzugeben, zum Beispiel

```
matrix = xlsread('..\Daten\werte.xls','Tabelle2');.
```

Den beiden vorangegangenen Eingaben ist gemeinsam, dass bei ihnen nur die numerischen Werte aus den entsprechenden Arbeitsblättern eingelesen werden. Fall es keine numerischen Werte gibt, dann wird eine leere Matrix `[]` übergeben. Möchte man auch alle Zeichenketten mit einlesen, dann nutzt man beispielsweise die Syntax

```
[matrix,zeichenkette] = xlsread('..\Daten\werte.xls');,
```

wobei dann die Variablen `matrix` und `zeichenkette` zur Weiterverarbeitung separat verwendet werden können. Zeichenketten werden dabei in Datencontainern vom MATLAB-Datentyp `cell` abgelegt. Gibt es keine Zeichenkette, dann wird ein leerer Datencontainer {} zugewiesen.

Neuere Matlab-Versionen erlauben es auch, Daten in `xls`-Dateien zu schreiben, wofür es die Funktion `xlswrite.m` gibt. In älteren MATLAB-Versionen ist diese Möglichkeit allerdings noch nicht vorhanden.

Alternativ kann man Matrizen aus `csv`-Dateien (Trennzeichen-getrennte Datei, entspricht dem Inhalt von EXCEL-Tabellen als Textdatei, nur mit dem Unterschied, das eine Spalte von der anderen durch ein Trennzeichen voneinander abgegrenzt wird) einlesen. Dies erfolgt mit Hilfe der Funktion `csvread.m`, der der Pfad und der Name der einzulesenden Datei als Zeichenkette als Parameter zu übergeben ist, zum Beispiel

```
matrix = csvread('..\Daten\werte.csv');,
```

Die einzulesende Datei darf aber ausschließlich numerische Werte enthalten, also keinerlei Zeichenketten oder ähnliche Dinge! Weiterhin ist zu beachten, dass die Funktion `csvread.m` als Trennzeichen nur ein Komma akzeptiert und für Dezimalzahlen muss ein Punkt verwendet werden. Hierbei kann es ggf. Probleme geben, denn z.B. bei der EXCEL-Version EXCEL 97 wurden `csv`-Dateien genau in dieser Form angelegt, während die Version EXCEL 2000 als Trennzeichen ein Semikolon und für Dezimalzahlen ein Komma verwendet. Letztere

Variante kann man aber auch mit EXCEL 97 öffnen, während die durch Kommata getrennte Variante durch EXCEL 2000 nicht geöffnet werden kann. Diese Umstände sind auch dann zu beachten, wenn man mittels des Befehls `csvwrite` Matrizen in eine `csv`-Datei schreiben möchte, zum Beispiel

```
matrix = csvwrite('..\Daten\dateiname.csv',matrix);
```

4 Grafik

Es werden viele Möglichkeiten zur Darstellung von zweidimensionalen (und auch dreidimensionalen) Daten bereitgestellt. Grafiken werden stets in seperaten Fenstern dargestellt. Für zweidimensionale Anwendungen am wichtigsten ist der `plot(DB,W,...)`-Befehl, dem stets (mindestens) der Definitionsbereich `DB` und die Werte `W` der darzustellenden Funktion als Vektoren gleicher Länge übergeben werden, wobei der i-te Eintrag von `DB` und der i-te Eintrag von `W` einzelne Punkte definieren, zwischen denen zur Darstellung linear interpoliert wird (Polygonzug), d.h. z.B. zur Darstellung der Funktion $f(x) = x$ im Intervall $[0, 100]$ genügt es den Befehl

```
plot([0,100],[0,100]);
```

einzugeben. Zur Darstellung nichtlinearer Funktionen ist es also notwendig, eine größere Anzahl von dichter beieinander liegenden Punkten für `DB` und `W` anzugeben, da der Kurvenverlauf sonst den Eindruck eines Polygonzuges hinterläßt. Eine „glatte“ Darstellung der Sinus-Funktion wird zum Beispiel wie folgt erhalten:

```
DB = 0:0.001:2*pi;
plot(DB,sin(DB));
```

Optional können der Funktion weitere Parameter (z.B. Farbe, Positionsangaben, usw.) übergeben werden, worauf hier nicht näher eingegangen werden soll.

Literatur

[1] Angermann,A.: Matlab - Simulink - Stateflow, Oldenbourg, München 2002

[2] Benker,H.: Mathematik mit MATLAB, Springer, Berlin-Wien 2000

[3] Katzenbeisser,S.; Überhuber,C.W.: Matlab 6.5, Springer, Berlin-Wien 2002

Internetverweise

- `http://www.home.uni-osnabrueck.de/phertel/num/eml.pdf`
- `http://www.fluid.tuwien.ac.at/322042?action=AttachFile&do=get&target=matlab-primer.ps`
- `http://www.esi.ac.at/~susanne/MatlabSkriptum.pdf`
- `http://people.inf.ethz.ch/arbenz/MatlabKurs/matlabintro.pdf`
- `http://www.ti3.tu-harburg.de/~haerter/PraktikumI/schramm.pdf`
