Eine kurze Einführung in MATLAB

1 Grundlegendes

Im Folgenden wollen wir annehmen, dass wir bereits wissen wie wir *MATLAB* starten, d.h., unter LINUX eine Shell-Konsole öffnen (–> System –> Konsole oder über Icon) und `matlab` eingeben. Des Weiteren sollten folgende Symbole klar sein:

`>>` Der *MATLAB*-Prompt, also die Eingabeaufforderung.

% Das Kommentar-Zeichen, alles, was hinter diesem Zeichen steht, wird von *MATLAB* ignoriert.

; Semikolon, wird ein Befehl mit dem Semikolon abgeschlossen, so wird das Ergebnis nicht angezeigt. Kann mehrere Befehle in einer Zeile trennen.

, Komma, trennt mehrere Befehle in einer Zeile, kein Unterdrückung der Ausgabe.

= Der Zuweisungsoperator.

Weiters ist zu beachten, dass *MATLAB* zwischen Groß- und Kleinschreibung unterscheidet und dass die Indizierung bei 1 (nicht bei 0) beginnt! Auch erwähnenswert ist, dass *MATLAB* mit doppelter Genauigkeit rechnet. Optionen und Dokumentation zu einem *MATLAB*-Befehl `dummy` erhält man mittels

```
>> help dummy
```

oder

```
>> doc dummy
```

2 Eingabe von Skalaren, Vektoren und Matrizen

2.1 Explizite Eingabe

Eingabe eines Skalars:

```
>> n = 8
```

Eingabe eines Zeilenvektors:

```
>> x = [ 1 2 3 4 5 6 7 8 ]
```

Diese Eingabe läßt sich wie folgt abkürzen:

```
>> x = 1:n
```

In der obigen Eingabe wird standardmäßig ein Inkrement von 1 angenommen. Natürlich können wir auch beliebige (negative, nicht ganzzahlige) Inkremente benutzen:

```
>> x = 1:0.5:n
>> x = n:-1:1
```

Eingabe eines Spaltenvektors:

```
>> y = [ 1; 2; 3; 4 ]
```

Eingabe einer Matrix:

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

Wollen wir den Eintrag a_{11} der Matrix A ändern, erreichen wir dies mit folgendem Befehl:

```
>> A(1,1) = 99;
```

Zeige den neuen Inhalt der Matrix A an:

```
>> A
```

2.2 Laden aus Dateien

Angenommen in einer Datei `matrix.dat` sind die einzelnen Komponenten der obigen Matrix A in folgender Form abgespeichert:

```
1 2 3
4 5 6
7 8 9
```

Die einzelnen Zeilen von A sind also zeilenweise in `matrix.dat` aufgelistet. Um diese Daten in eine neue Matrixvariable, sagen wir B, einzulesen, verwenden wir den Befehl `load`:

```
>> B = load('matrix.dat');
```

Mit dem obigen Befehl wird eine Variable B der Dimension 3×3 erzeugt, die anschließend mit den Werten aus der Datei gefüllt wird. Gibt man nur

```
>> load('matrix.dat')
```

ein, so wird eine Variable namens `matrix` erzeugt, die die Komponenten aus der Datei enthält.

2.3 Eingebaute *MATLAB*–Funktionen zur Matrix/Vektor Generierung

MATLAB besitzt einige Funktionen, die uns bei der Erzeugung von Matrizen und Vektoren helfen können. Zum Beispiel erzeugt

```
>> E = eye(3,5)
```

eine Einheitsmatrix mit 3 Zeilen und 5 Spalten. `eye(3,3)` würde eine quadratische Einheitsmatrix erzeugen, kann aber durch `eye(3)` abgekürzt werden. Analog dazu können wir mit

```
>> O = ones(5,3)
```

eine Matrix mit 5 Zeilen und 3 Spalten erzeugen, die nur aus Einsen besteht. Wiederum erhalten wir mit `ones(3)` eine quadratische Matrix. Matrizen mit beliebigen Einträgen ungleich 0 erhalten wir mittels Multiplikation des gewünschten Wertes mit `ones`. Zum Beispiel eine Matrix mit lauter π Einträgen:

```
>> pi * ones(3)
```

Der Befehl `zeros` hat die gleiche Funktionalität, nur liefert er lauter Nulleinträge. Eine Matrix, deren Komponenten Zufallszahlen aus dem Intervall $[0, 1]$ sind, liefert uns folgender Befehl:

```
>> R = rand(3)
```

Alle obige Befehlen können wir natürlich benutzen, um mit zum Beispiel

```
>> ones(3,1)
```

einen Spaltenvektor und mit zum Beispiel

```
>> ones(1,3)
```

einen Zeilenvektor zu erzeugen.

3 Matrizen und Matrixoperationen

3.1 Die *Colon*-Notation

Lassen wir uns zuerst die Matrix A anzeigen.

```
>> A
```

Mit der Anweisung

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

erhalten wir die Untermatrix B von A, die aus den Zeilen 1 und 2 (abgekürzt mit `1:2`) und aus den Spalten 2 und 3 (abgekürzt mit `2:3`) zusammengesetzt ist. Eine komplette Zeile (hier die zweite Zeile) liefert der Befehl

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

und analog

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

eine ganze Spalte einer Matrix. Somit könnten wir auch mit

```
>> A(:,:)
```

den gleichen Effekt wie mit `>> A` erzielen. Diese *Colon*-Notation (colon, engl. Doppelpunkt) erlaubt uns gezielt auf Untermatrizen zuzugreifen, um diese zu extrahieren oder zB. zu überschreiben:

```
>> A(1:2,1:2) = eye(2)
```

Einen Spaltenvektor, der durch das Aneinanderreihen der einzelnen Spalten von A entsteht, erhalten wir mittels

```
>> a = A(:)
```

Somit dürfte klar sein, auf welchen Eintrag wir mit `A(4)` zugreifen, nämlich auf A_{12}. Umgekehrt können wir auch einen Spaltenvektor verwenden, um eine bereits vorhandene Matrix zu überschreiben:

```
>> R(:) = a
```

Womit nun die Einträge von A und R identisch sind.

3.2 Matrixoperationen

Nun zu einigen grundlegenden Matrixoperationen:

```
>> A ^ 2
```

berechnet $A \cdot A$, das Quadrat der Matrix A. Das Produkt zweier Matrizen wird klarerweise mit

```
>> A * M
```

und die Summe mit

```
>> A + M
```

berechnet. Für zwei quadratische Matrizen kennt *MATLAB* auch die Funktion

```
>> B = A / M
```

Hier ist B die Matrix die $B \cdot M = A$ erfüllt. Diese Operationen können auch komponentenweise durchgeführt werden. Dazu müssen wir nur dem entsprechenden Operator einen Punkt (`.`) voranstellen:

```
>> A .^ 2
```

erzeugt die Matrix mit den Einträgen $(a_{ij} \cdot a_{ij})_{ij}$, folgender Befehl die Matrix $(a_{ij} \cdot m_{ij})_{ij}$

```
>> A .* M
```

und schließlich $(a_{ij}/m_{ij})_{ij}$:

```
>> A ./ M
```

Die Transponierte von A bilden wir wie folgt:

```
>> A'
```

Wir können auch Operationen zwischen Matrizen und Skalaren benützen.

```
>> n + A
```

Jeder Eintrag der Matrix A wird mit n addiert.

```
>> n * A
```

Jeder Eintrag der Matrix A wird mit n multipliziert.

3.3 Funktionen mit Matrizen

Ist A eine bereits definierte Matrix, dann liefert uns der Befehl

```
>> size(A)
```

die Größe der Matrix. Möglich ist auch

```
>> [m,n] = size(A)
```

was in m die Anzahl der Zeilen und in n die Anzahl der Spalten abspeichert. Weiters liefert der Befehl

```
>> diag(A)
```

die Diagonale von A (als Spaltenvektor). Umgekehrt ergibt für einen Spaltenvektor v (mit n Komponenten) der Befehl

```
>> B = diag(v)
```

eine Diagonalmatrix mit den Elementen von v in der Diagonale.

```
>> B = diag(v,1)
```

liefert uns eine $(n+1)\times(n+1)$ Matrix B mit dem Vektor v in der ersten oberen Nebendiagonale. Ersetzen wir im obigen Befehl `1` mit `-1`, dann würde die erste untere Nebendiagonale belegt werden. Durch

```
>> sum(A)
```

werden die Elemente in den Spalten aufsummiert (ergibt als Ergebnis einen Zeilenvektor). Andere Ergebnisse ermöglichen uns noch die folgenden Befehle:

`>> sum(diag(A))`	(summiert die Diagonalelemente)
`>> sum(A')`	(Elemente in den Zeilen werden summiert)
`>> sum(A(:,3))`	(summiert die Elemente der 3. Spalte)
`>> sum(sum(A))`	(summiert über alle Elemente von A)
`>> diag(diag(A))`	(Diagonalmatrix)
`>> A(:,i) = []`	(löscht die i-te Spalte aus A)

und so weiter ...

3.4 Lineare Gleichungsystemen

Zwecks Einfachheit wollen wir hier nur quadratische Matrizen betrachten. Auch wenn etwas unüblich, benützt *MATLAB* das Divisionszeichen (wie im skalaren Fall), um die allgemeine Lösung eines linearen Systems zu beschreiben. Dafür werden die beiden Divisionsymbole *slash*, (`/`) und *backslash* (\) benützt. Dabei steht `X = A \ B` für die Lösung des Gleichungssystems $AX = B$ und `X = B / A` für die Lösung von $XA = B$. Natürlich müssen

die Dimensionen der vorkommenden Matrizen (Vektoren) übereinstimmen. Wir wollen dies gleich an einem kleinen Beispiel ausprobieren:

```
>> A = rand( 3 )
>> b = ones( 3, 1 )
>> x = A / b
```

Die Inverse einer Matrix A können wir direkt mittels

```
>> inv( A )
```

ausrechnen, was natürlich nur bei invertierbaren Matrizen funktioniert. Die Determinante einer Matrix liefert uns:

```
>> det( A )
```

Im Hintergrund benützt *MATLAB* Gauß'sche Elimination wenn wir `/` oder `\` anwenden. Nähere Informationen erhalten wir mittels `help mldivide` oder `help mrdivide`.

Iterative Methoden zum Lösen von linearen Gleichungssystemen müssen wir explizit aufrufen, wie zum Beispiel das Verfahren der konjugierten Gradienten (*CG*):

```
>> x = pcg( A, b, tol, maxit )
```

Wobei hier die Koeffizientenmatrix A symmetrisch und positiv definit sein sollte. Der Befehl `pcg` realisert nun das CG–Verfahren mit Startvektor $\mathbf{0}$, bis die euklidische Norm des Residuums kleiner als der Parameter `tol` ist, oder die Anzahl der Iterationen `maxit` übersteigt. Eine vorkonditionierte CG–Methode können wir entweder mit

```
>> x = pcg( A, b, tol, maxit, M1, M2 )
```

oder mit

```
>> x = pcg( A, b, tol, maxit, vorkondfunk )
```

anwenden. Bei der ersten Variante wird die Suchrichtung $\mathbf{p}$ aus dem Residuum $\mathbf{r}$ durch $\mathbf{y} = M_1^{-1}\mathbf{r}$ und $\mathbf{p} = M_2^{-1}\mathbf{y}$ bestimmt. Alternativ besteht die Möglichkeit durch die Datei `vorkondfunk.m` eine Funktion zur Verfügung zu stellen, die aus dem Residuum $\mathbf{r}$ die Suchrichtung $\mathbf{p}$ berechnet. Für nicht positiv definite Matrizen stellt *MATLAB* zB. die Methoden *MINRES* und *QMR* zur Vefügung, die mittel `minres` und `qmr` und mit gleichen Parametern wie `pcg` aufgerufen werden. Weitere verwandte Methoden finden wir unter `help pcg` am Ende des Hilfetextes.

Iterative Methode bieten sich insbesonders bei großen dünn besetzten Matrizen an. Dünn besetzte Matrizen können in *MATLAB* durch `sparse` erzeugt werden. Enthält die Matrix A viele Nullelemente, so liefert uns

```
>> S = sparse( A )
```

eine *sparse* Darstellung von A. `sparse( n, m )` würde eine Sparsematrix mit nur Nullelementen liefern, mehr dazu erfahren wir durch `help sparse`.

3.4.1 Kurzes zum Vorkonditioneren

Die Vorkonditionierungs Matrix sollte die Inverse der Matrix A gut approximieren und leicht zu berechnen sein. Ein Möglichkeit bietet uns dafür zB. die unvollständige Cholesky Zerlegung:

```
>> C = cholinc( A, tol )
```

wobei `tol` die Genauigkeit der Zerlegung steuert. `tol = 0` würde die vollständige Choleskyzerlegung liefern und größere Werte von `tol` (`1e-6` bis `1e-1`) liefern dünner besetztere Matrizen C, aber mit schlechterer Vorkonditionierungs Eigenschaften. Die Matrix C können wir dann durch `M1 = C'` und `M2 = C` als Vorkonditionierer verwenden. Bei der unvollständigen LU Zerlegung können wir `M1 = L` und `M2 = U` setzen:

```
>> [ L, U ] = luinc( A, tol )
```

Wobei `tol` hier die gleiche Bedeutung wie bei `cholinc` hat.

3.5 Matrix Faktorisierungen

Die *Cholesky* Faktorisierung $A = R^T R$ stellt eine symmetrische Matrix durch das Produkt einer oberen Dreiecksmatrix R und ihrer Transponierten dar:

```
>> R = chol( A )
```

Die LU Faktorisierung (oder Gauß Elimination) stellt eine beliebige quadratische Matrix als das Produkt einer unteren und oberen Dreiecksmatrix dar:

```
>> [ L, U ] = lu( A )
```

Hier müssen wir beachten, dass L nicht notwendigerweise eine untere Dreicksmatrix ist. Durch den Befehl

```
>> [ L, U, P ] = lu( A )
```

bekommen wir zusätzlich eine Permutationsmatrix P, sodass $PA = LU$. Die *orthogonale*, oder QR, Faktorisierung $A = QR$ stellt eine rechteckige Matrix als das Produkt einer orthogonalen und einer oberen Dreiecksmatrix dar:

```
>> [ Q, R ] = qr( A )
```

Wobei ebensfalls wieder eine Variante mit Pivotierung existiert
(`[ Q, R, P ] = qr( A )`).

4 Bedingte Verzweigungen und Schleifen

MATLAB stellt uns folgende Vergleichsoperatoren zur Verfügung:

MATLAB	math. Bedeutung
`<`	$<$
`<=`	$\leq$
`>`	$>$
`>=`	$\geq$
`==`	$=$
`~=`	$\neq$
`&`	und
`|`	oder
`~`	nicht

4.1 *if*-Abfragen

Die Bedeutung von *if*-Abfragen wollen wir als bekannt vorraussetzen. Wir wollen annehmen, dass alle vorkommenden Variablen in *MATLAB* schon bekannt sind. Als Beispiel einer *if*-Abfrage wollen wir die Vorzeichenfunktion implementieren:

```
>> if x < 0
     s = -1;
   elseif x > 0
     s = 1;
   else
     s = 0;
   end
```

Natürlich sind auch mehrere `elseif`-Teile möglich.

4.2 *for*-Schleifen

Die *for*-Schleife wollen wir anhand zweier Beispiele vorstellen. Zehn Totozahlen sollen zufällig generiert werden und in dem Vektor x gespeichert werden:

```
>> x = [];
>> for i=1:11
     x = [ x; round(2*rand) ];
   end
>> x
```

Selbstverständlich dürfen *for*-Schleifen auch geschachtelt werden:

```
>> for i=1:5
     for j=1:3
       A(i,j) = i+j-1;
     end
   end
```

4.3 *while*-Schleifen

Wiederum wollen wir zwei *while*-Schleifen angeben, wobei das erste Beispiel ein Negativbeispiel ist:

```
>> n = 1;
>> while n > 0
     n = n + 1;
   end
```

Wie kommen wir da blos wieder raus? Mit der Tastenkombination `CRTL+C`. Ein weniger destruktives Beispiel:

```
>> n = 0;
>> while n+1 <= x
     n = n + 1;
   end
>> n
```

Wir wollen hier zu einer positiven Zahl x die kleinste ganze Zahl n mit der Eigenschaft $n \leq x < n + 1$ berechnen.

4.4 Allgemeines

Vorzeitiges Beenden einer Schleife erreichen wir durch den Befehl `break` im Anwendungsteil einer Schleife. Die Schleife wird sofort beendet und zu der Zeile nach `end` gesprungen. Durch das Kommando `continue` wird der aktuelle Schleifendurchlauf abgebrochen und die Schleifenabarbeitung mit dem nächsten Schleifendurchlauf fortgesetzt.

MATLAB ist eine Sprache, die für den Umgang mit Matrizen und Vektoren entwickelt wurde, d.h. Operationen mit Matrizen und Vektoren werden schnell ausgeführt. Schleifen dagegen erfordern wesentlich mehr Rechenzeit und sollten daher, wenn möglich, vermieden werden. Im folgenden wollen wir eine Wertetabelle von $\sin(t)$ erstellen, dies kann also auf verschiedene Arten realisiert werden:

```
tic
i = 0;
for t = 0:0.01:10
  i = i + 1;
  y(i) = sin(t);
end
toc
```

Oder durch folgende zweite Möglichkeit:

```
tic
t = 0:0.01:10;
y = sin(t);
toc
```

Das zweite Beispiel wird wesentlich schneller ausgeführt. Bei dieser Gelegenheit lernen wir auch noch die Befehle `tic` und `toc` kennen, welche die benötigte Rechenzeit, der zwischen ihnen eingeschlossenen Kommandos, bestimmen.

5 Dateien als Programme/Skripte

Dateien mit dem Suffix `.m` spielen eine wichtige Rolle für *MATLAB*. Im folgenden werden wir zwei verschiedene Klassen von `m`-Dateien kennen lernen.

5.1 Skriptdateien

Eine *Skript*datei enthält eine Folge von *MATLAB* Anweisungen. Nehmen wir an wir haben eine Datei namens `dreizeiler.m` erzeugt, die die folgenden drei Zeilen enthält:

```
A = [ 1 2; 3 4 ];
d = det(A)
B = A' * A
```

Dann werden nach dem Aufruf

```
>> dreizeiler
```

die drei Befehlszeilen an *MATLAB* übergeben und ausgeführt. Es macht somit keinen Unterschied, ob wir diese drei Zeilen durch das Laden von `dreizeiler.m` ausführen, oder ob wir die einzelnen Anweisungen während einer Sitzung hinereinander eingeben:

```
>> A = [ 1 2; 3 4 ];
>> d = det(A)
>> B = A' * A
```

Damit sollte die Bedeutung von Skriptdateien klar sein. In Skriptdateien speichert man typischerweise eine Folge von *MATLAB*-Anweisungen ab, die man während einer Sitzung häufiger verwendet, aber die man aus Zeitgründen, nicht immer wiederholt eingeben möchte.

5.2 Funktionsdateien

Mit Hilfe von Funktionsdateien können wir neue eigene Funktionen definieren, die von *MATLAB* während einer Sitzung verwendet werden können. Ebenso können wir mit eigenen Funktionen bestehende Implementierungen von *MATLAB*-Funktionen überlagern. Als Beispiel wollen wir eine Funktion, die zuerst die Länge eines Vektors feststellt und dann den Mittelwert berechnet, erstellen:

```
function mw = mittelwert( u )
% Diese Funktion berechnet den Mittelwert eines Vektors
n = length( u );
mw = sum( u ) / n;
```

Dieses Programm muss in eine Datei names `mittelwert.m` gespeichert werden. Im folgenden Beispiel wollen wir nun diese Funktion aufrufen :

```
>> u = [ 1 2 3 4 5 6 ];
>> mittelwert(u)
```

Mit dem Befehl `help mittelwert` wird die Kommentarzeile ausgegeben.

5.2.1 Unterfunktionen

Jede Funktion, die wir während einer Sitzung aufrufen wollen, muss sich in einer eigenen Datei befinden. Werden jedoch innerhalb der Funktion weitere Funktionen aufgerufen, so können diese in derselben Datei stehen. Solche *Unterfunktionen* können aber von anderen *MATLAB*-Programmen nicht direkt aufgerufen werden. Dazu wollen wir uns folgendes Beispiel betrachten, welches in die Datei `auswert.m` gespeichert wird.

```
function [ mittel, fehler ] = auswert( u )
mittel = mittelwert( u );
fehler = quadrat( u, mittel );
function mw = mittelwert( u )
% Diese Funktion berechnet den Mittelwert eines Vektors
n = length( u );
mw = sum( u ) / n;
function error = quadrat( u, mw )
error = sum( ( u - mw ).^2 )
```

Hier haben wir zwei Unterfunktionen (`mittelwert` und `quadrat`) definiert, die von der eigentlichen Funktion `auswert` aufgerufen werden. Diese Funktion können wir etwa so aufrufen:

```
>> [ a, b ] = auswert( u )
```

Aber `mittelwert` und `quadrat` können wir nicht direkt benutzen, da sie sich in der Datei `auswert.m` befinden.

5.2.2 Funktionen als Parameter

Eine Funktion kann auch als Parameter an eine andere Funktion übergeben werden. Dabei müssen wir jedoch einige Dinge beachten, wie anhand des folgenden Beispiels deutlich werden sollte. In eine Datei `zeichnefunk.m` speichern wir folgende Funktion, die eine andere Funktion zeichnen soll:

```
function x = zeichnefunk( funk, data )
werte = feval( funk, data );
plot( data, werte );
```

Dabei ist `data` ein Vektor, der die Werte aus dem aus dem Definitionsbereich enthält, an denen die Funktion ausgewertet werden soll. Der Parameter `funk` ist die zu zeichnende

Funktion. Der Befehl `feval` wertet die Funktion `funk` an den Punkten `data` aus. Die Funktion `zeichnefunk` wird dann wie folgt aufgerufen:

```
>> t = -3:0.01:3;
>> zeichnefunk( @sin, t );
```

Dadurch zeichnen wir die Sinus-Funktion im Intervall $[-3, 3]$. Zu beachten ist, dass die Übergabe der Funktion `sin` als Parameter `@sin` lautet, also mit vorangestelltem @-Zeichen. Entsprechend müssen wir bei selbst definierten Funktionen vorgehen.