Einführung in MATLAB/Octave

Dr. Falk Ebert

4. Dezember 2017

MATLAB (MATrix LABoratory)[1] wurde Ende der 1970er Jahre entwickelt um Berechnungen mit Vektoren und Matrizen ohnen großen Programmieraufwand durchführen zu können. Mit der Zeit wurde dieses Programm aber immer leistungsfähiger und ist mittlerweile aus der Arbeit von Naturwissenschaftlern und Ingenieuren kaum noch wegzudenken. Für die Anwendung in der Schule ist es zum einen unhandlich groß und auch zu teuer. Eine kostenfreie open-source Alternative ist *Octave*[2]. Diese hat zwar nicht den kompletten Leistungssumfang wie das Original, ist aber für den Gebrauch in der Schule mehr als ausreichend. Man kann *Octave* auch ohne Istallation nutzen, indem man einen Online-Client[3] nutzt.
Die Folgenden Kapitel sind als Einführung zum Selbstlernen gedacht. Dazu empfiehlt es sich, die Anweisungen in den Boxen tatsächlich mal einzutippen und die genauen Rückmeldungen des Programms zu beobachten.

Inhaltsverzeichnis

[1] `http://www.mathworks.com/products/matlab.html`
[2] `http://www.gnu.org/software/octave/`
[3] `http://octave-online.net/`

1 Erste Schritte

1.1 Variablen und Werte

Gib folgende Zeilen in den MATLAB/Octave Befehlsbildschirm ein. Jede Zeile entspricht einem Befehl, der nach Drücken der ENTER-Taste ausgeführt wird.

```
x
x=42
x
```

Zumindest ist das der Plan. Bereits die erste Zeile wird eine Fehlermeldung bringen (Zumindest sollte sie das). Das Programm beschwert sich, dass `'x'`nicht definiert wäre. Das stimmt, denn (wenn nichts anderes vorher schief gegangen ist) bisher wurde noch gar nichts definiert. Also definieren wir `x` gleich in der zweiten Zeile. MATLAB/Octave wird das Definierte artig wiederholen. Und in der dritten Zeile etwas mit der Variablen `x` anzufangen wissen und deren Wert erneut ausgeben.
Die Zuweisung eines Wertes zu einer Variablen funktioniert von rechts nach links. Der Variablenname muss links stehen, der zugewiesene Wert rechts. `1=x` wird also einen Fehler bringen. Die mathematisch falsche Aussage `x=x+1` ist aber absolut zulässig, um den Wert von `x` um 1 zu erhöhen - vorausgesetzt, `x` ist bereits definiert. Sonst kommt wieder ein Fehler.
Variablennamen beschränken sich nicht nur auf Kleinbuchstaben. Kombinationen von Buchstaben, Ziffern und Unterstrichen sind erlaubt, solange keine Ziffer an erster Stelle steht. Die folgenden Definitionen sind alle zulässig.

```
xyz=1
Sinn_des_Lebens_des_Universums_und_des_ganzen_Rests = 42
___sogar_das_geht = 2
```

Sollte `'x'` irgendwann nicht mehr gebraucht werden, kann es mit `clear x` komplett gelöscht werden. Probiere folgendes aus:

```
x=1, y=2, z=3
who
clear x
who
clear all
who
```

Dieses Beispiel zeigt gleich mehrere Dinge. Zum einen werden in der ersten Zeile mehrere Variablendefinitionen in eine Zeile gepackt. Dies kann in vielen Situationen sinnvoll werden. Wir werden bei Schleifen darauf zurückkommen. Bitte probiere diese Zeilen erneut aus, aber setze hinter jede Variablendefinition ein Semikolon „`;`“ statt eines Kommas. Probiere ebenfalls aus, was ein Semikolon nach `who` bewirkt. Zur Vereinfachung kann man auch alte Zeilen mit der ⇑-Taste zurückholen und gegebenenfalls nochmal ändern.
Der Befehl `who` gibt einen Überblick über die aktuell verwendeten Variablen. Im MATLAB/Octave GUI kann das auch in einem separaten Fenster dargestellt werden. In MATLAB ist es außerdem möglich, den Befehl `why` zu verwenden. Dieses Easteregg gibt es in Octave nicht. Dafür spuckt dort `fact` mehr oder weniger verlässliche Fakten aus dem Open Source-Universum aus.

Mit `clear <Variablenname>` kann eine Variable gelöscht werden. `clear all` löscht nicht ganz unerwartet alle aktuell definierten Variablen.

Bisher haben wir nur natürliche Zahlen verwendet. Natürlich ist es auch möglich, negative Zahlen durch das Voranstellen eines `-` zuzuweisen. Rationale Zahlen werden mit einem Punkt statt eines Kommas notiert. Es ist ebenfalls möglich, eine Zehnerpotenz in der Ingenieurschreibweise einzugeben. Dazu wird zwischen Mantisse und Exponent ein `e` oder `E` gesetzt.

```
x=-42
y=4.2
z=-4.2e-3 % dies entspricht -4,2*10^(-3)
q=1/42
```

In der dritten Zeile steht ein durch `%` gekennzeichneter Kommentar. Alles was in einer Zeile nach dem Prozentzeichen folgt, wird von MATLAB/Octave einfach ignoriert. Das ist hier in der Kommandozeile nicht sehr sinnvoll, aber wenn später Scripts geschrieben werden, hilft das, diese für den Nutzer lesbar zu halten.
Die Variable `q` wird über eine Berechnung als $\frac{1}{42}$ definiert. Mehr dazu kommt später. Die Variablen `x`, `y` und `z` konnten nach ihrer Eingabe noch komplett dargestellt werden. `q` ist ein unendlicher Dezimalbruch und kann *nicht* komplett wiedergegeben werden. MATLAB/Octave speichert allerdings mehr als nur die dargestellten 4 Nachkommastellen. Die Ausgabe ist nur zur besseren Lesbarkeit auf 5 Stellen (eine Vorkomma- und 4 Nachkommastellen begrenzt). Mit den folgenden `format` Anweisungen kann das geändert werden. Probiere jede einzelne aus und gib jeweils `q` im Anschluss aus. Notiere die Unterschiede.

```
format long
format long e
format rat %rat steht fuer rational
format short e
format short
```

Mit der letzten Zeile ist der Normalzustand wieder hergestellt.
Die scheinbare Fähigkeit, mit Brüchen zu rechnen ist nur eine Illusion - aber eine hilfreiche. Tatsächlich werden von jeder eingegebenen Zahl etwa die ersten 16 signifikanten Kommastellen (also keine führenden Nullen) und der Exponent gespeichert. Das heißt, dass jede im Rechner gespeicherte Zahl nur in der Nähe der tatsächlich gewollten liegt. Konkret gilt, dass zu jedem zugewiesenen x eine Zahl `x` im Rechner gespeichert wird, für die gilt

$$\mathtt{x} \in [x - \varepsilon \cdot x,\, x + \varepsilon \cdot x].$$

Das ε ist die sogenannte Maschinengenauigkeit und kann in MATLAB/Octave mit `eps` ausgegeben werden. Diese fehlerhafte Zuweisung ist keine Krankheit von MATLAB/Octave sondern liegt einfach darin begründet, dass ein Computer zur Speicherung einer Zahl nur begrenzten Speicher zur Verfügung hat. Bei den hier verwendeten doppeltgenauen Kommazahlen sind das 64 Bit. Die daraus folgende Genauigkeit reicht für die meisten Anwendungen aus. Aber man sollte im Hinterkopf behalten, dass MATLAB/Octave kein Computer-Algebra-System ist[4], sondern nur ein extrem, leistungsfähiger Taschenrechner. Dieser speichert

[4]Man kann mittels des `syms` Kommandos eins daraus machen. Aber das geht hier zu weit.

leicht verfälschte Zahlen und durch Rechnen mit solchen falschen Zahlen können teilweise absurde Ergebnisse entstehen.

```
1234567890123457-1234567890123456 %noch alles OK
12345678901234568-12345678901234567 %unerwartet?
1234567890123456.1-1234567890123456.0
1234567890123456.2-1234567890123456.1 %sehr merkwuerdig!
```

Zur Erklärung des Ergebnisses der letzten Zeile kann man sich vor Augen führen, dass $\frac{0{,}25}{1234567890123456.2} \approx 2{,}025 \cdot 10^{-16}$ und damit extrem nah an der Maschinengenauigkeit ε liegt.

1.2 Vordefinierte Funktionen

Wir haben bereits einige Berechnungen durchgeführt. Die Grundrechenarten werden mit `+`, `-`, `*`, `/` beschrieben und es darf nach Lust und Laune mit `(` und `)` geklammert werden. Potenzieren geht mit `^`. Die Quadratwurzel wird mit `sqrt` gezogen. Der Betrag (Absolutwert) einer Zahl wird mit `abs` ermittelt. Weitere vordefinierte Funktionen sind `exp` und `log` für die natürliche Exponential- und Logarithmusfunktion sowie `sin`, `cos`, `tan` für die trigonometrischen Funktionen und `asin`, `acos`, `atan` für deren Umkehrfunktionen. Die Kreiszahl π ist als `pi` bereits vordefiniert. Man kann diesen Wert aber abändern und mit `pi=3` ganz im biblischen Sinn[5] weiterrechnen. Die eulersche Zahl e ist als `exp(1)` aufrufbar.

```
4*(1/1-1/3+1/5-1/7) %naehungsweise pi
4*atan(1) % =pi
exp(pi)-exp(1)^pi %sollte 0 ergeben, aber ...
```

Häufig wird man auch Werte runden müssen, speziell wenn es später darum geht, Indizes von Matrizen zu definieren.

```
round(pi)
round(3.5) % kaufmaennisches Runden
floor(pi) % stets abrunden
ceil(pi) % steht fuer ceiling also aufrunden
```

1.3 Komplexe Zahlen

MATLAB/Octave können ebenfalls korrekt mit komplexen Zahlen umgehen. Diese können mit Real- und Imaginärteil eingegeben werden. Die imaginäre Einheit $\sqrt{-1} \stackrel{!}{=} i$ ist dabei als `i` (und zur Freude der Elektrotechniker auch als `j`) vordefiniert. Auch diese Variablen können umdefiniert werden. Da diese beiden Buchstaben aber gern auch als Laufvariablen in Schleifen verwendet werden, ist das eine beliebte Quelle von Fehlern in Scripts, die mit komplexen Zahlen in Schleifen rechnen.

[5] `http://www.welt.de/welt_print/article1620503/Die-Zahl-Pi-taucht-schon-in-der-Bibel-auf.html`

```
x=sqrt(-4)
y=sqrt(x)
real(y)
imag(y)
z=sqrt(y)
z'
abs(z)
arg(z)
exp(i*pi) %die Euler-Identitaet[6] funktioniert auch
```

Hier wird `y` auf den Wert $0 + 2i$ gesetzt. Die Funktionen `real` und `imag` liefern jeweils den Real- und Imaginärteil von `y`. `z` wird dann durch nochmaliges Wurzelziehen aus `y` mit $1 + i$ initialisiert. Das Apostroph „`'`“ hinter `z` liefert die zu `z` komplex konjugierte Zahl. Mit `abs` bestimmt man wie gehabt den Betrag einer Zahl und `arg` liefert das Argument im Bogenmaß. In diesem Fall ist $\arg(1 + i) = 45° = \frac{\pi}{4}$.

1.4 Grenzen der Rechnerei

Etwas, das MATLAB/Octave fast schon auf eine Stufe mit Chuck Norris stellt[7], ist die Tatsache, dass man durch Null teilen kann. Konkret gilt $\frac{1}{0} \stackrel{!}{=} \infty$ und konsequenterweise $\frac{-1}{0} = -\infty$. Für ∞ ist die Variable `inf` vorgesehen. Auch diese kann ebenso wie `i` und `pi` umdefiniert werden und zu absurden Fehlern führen. Tatsächlich ist `inf` aber einfach nur alles was größer als die größte maschinell darstellbare Zahl ist. Mit `inf` kann aber tatsächlich noch halbwegs sinnvoll weitergerechnet werden, solange keine undefinierten Ausdrücke wie $\infty - \infty$ oder $\frac{\infty}{\infty}$ auftreten.

```
1/0 %Hail to the Chuck!
realmax() %die groesste darstellbare Zahl
realmax()*(1+eps) % etwas mehr ist schon zu viel
1/inf %nicht unerwartet
inf+inf % konsistent mit den Grenzwertsaetzen
inf*inf % konsistent mit den Grenzwertsaetzen
inf-inf % undefiniert
inf/inf % undefiniert
```

Die Ergebnisse der letzen beiden Zeilen sind `NaN`, was für „Not a Number“ steht und letztendlich heißt, dass irgendwo irgendeine Berechnung ziemlich schief gegangen ist. Ein Script, das anfängt, `NaN` auszuspucken, hat wahrscheinlich einen Fehler.

Wir haben jetzt einige erste Schritte unternommen und einige grundlegende Prinzipien kennengelernt. Im nächsten Abschnitt wird es konkret um das MAT von MATLAB gehen, also die Fähigkeit, mit Matrizen zu rechnen.

[6] `http://de.wikipedia.org/wiki/Eulersche_Formel#Eulersche_Identit.C3.A4t`
[7] `http://www.youtube.com/watch?v=awmtY3dBui4`

1.5 Aufgaben

A1.1 Definiere den Wert von `pi` um. Beobachte, wie sich die Ausgabe von `who` ändert.

A1.2 Was macht der Befehl `clear` mit vordefinierten Werten wie `pi`, die umgeändert wurden?

A1.3 Es gibt eine weitere vordefinierte Variable namens `ans`. Finde heraus, was sie beinhaltet.

A1.4 In MATLAB ist die Länge eines Variablennamens begrenzt. Finde heraus, wie lang eine Variable sein darf. Wie sieht die Situation bei Octave (normal/online) aus?

A1.5 Bestimme den exakten Wert von i^i. Prüfe, ob MATLAB/Octave das korrekt berechnet.

A1.6 Prüfe weitere Ausdrücke mit `inf` auf ihre Vertrglichkeit mit den Grenzwertsätzen. Gibt es Unterschiede zwischen MATLAB und Octave (normal/online)?

A1.7* Warum wurde als nächstgrößere Zahl von `realmax()` nicht `realmax()+1` verwendet?

2 Matrizen

2.1 Eingabe

Das Abrufen einer nicht definierten Variablen, z.B. `a` wird zu einem Fehler führen. Aber selbst bei definerten Variablen können unerwartete Fehler auftauchen.

```
a=1
a(5) %genau die Fehlermeldung lesen!
a(5)=42 %Ausgabe beachten
a(5)
a(3)
a(6) %jetzt nicht mehr so unerwartet
b=a'
```

Durch die Zahl in Klammern nach einer Variablen nimmt MATLAB/Octave an, dass es sich bei der Variablen um ein Feld/Array, bzw. mathematisch um einen Vektor handelt. Wird erstmalig einem Element dieses Vektors ein Wert zugewiesen, wird automatisch die Variable als Vektor erzeugt, der groß genug ist. Alle nicht explizit definierten Einträge werden mit 0 initialisiert. Auch ist ein solcher Vektor standardmäßig ein Zeilenvektor. In der letzten Zeile taucht ein Apostroph auf. Hier wird der Vektor `b` als der zu `a` transponierte Vektor definiert. Das Transponieren mittels „'“ ist die einfachste Möglichkeit, einen Spaltenvektor zu erzeugen. Eine weitere Möglichkeit ist, den Vektor als eine sehr schmale Matrix aufzufassen.
Eine Matrix ist ein zweidimensionales Zahlenfeld. Als solches muss sie mit zwei Indizes (nicht Indexe!) angesprochen werden. Die einfachste Möglichkeit, einen

Wert einer Matrix zu setzen ist dabei, diesen einzeln einzugeben. Dabei ist der erste Index die Zeile und der zweite Index die Spalte.

```
A(2,3)=23
A(4,2)=42
```

Das Ergebnis sollte eine Matrix mit 3 Zeilen, 4 Spalten und jeder Menge Nullen sein. Wir haben zwei Werte konkret definiert, nämlich $A_{2,3}$ und $A_{4,2}$. Bei der Belegung der Werte wird eine Matrix erzeugt, die groß genug ist, den neuen Wert aufzunehmen. Der Rest wird mit Nullen aufgefüllt. Das wird allerdings mühselig, wenn die Matrix mehr als nur ein paar Nicht-Null-Elemente enthält. Dafür gibt es eine weitere einfache Methode. Man kann eine Matrix auch als Liste von Werten in eckigen Klammern eingeben. Dies erfolgt zeilenweise, wobei einzelne Werte einer Zeile durch Komma oder Leerzeichen getrennt sind. Ein Semikolon „`;`“ markiert einen Zeilenumbruch und damit die Eingabe einer neuen Zeile.

```
C=[1 2 3;4,5,6] % Komma oder Leerzeichen als Trenner in
Zeilen C' % ' transponiert die Matrix D=[2 2; 3 3 3] %
Fehlermeldung lesen!
```

Die Definition von `C` sollte problemlos funktionieren. Bei `D` tritt ein Fehler auf, weil die erste Zeile nur 2 und die zweite Zeile 3 Elemente hat.

2.2 Besondere Matrizen

Die Eingabe einer großen Einheitsmatrix ist mit keiner der bisher genannten Varianten komfortabel. Glücklicherweise sind gewisse häufig vorkommende Matrizentypen über einfache Funktionen leicht verfügbar.

```
eye(3) %3x3 Einheitsmatrix
ones(4) %nur ein Argument
ones(2,4) %zwei Argumente
zeros(4)
zeros(5,3) % wie bei den Einsmatrizen
rand
rand(3)
rand(2,4) % gleichverteilte Werte aus [0,1]
```

2.3 Bereichsoperator „`:`“

Eine einfache und nützliche Möglichkeit eine Folge von Zahlen zu erstellen, ist die Benutzung des Operators „`:`“. Dieser kann grob als *bis* gelesen werden. `1:10` erzeugt die Folge von Zahlen von 1 *bis* 10. Will man größere Schritte haben, kann man eine ähnliche Konstruktion benutzen.

```
1:10 % das Beispiel aus dem Text
1:2:10 % jetzt in 2er-Schritten
10:1 % wird nicht funktionieren
10:-1:1 % jetzt wird abwaerts gezaehlt
0:0.1:1 % geht auch mit Bruechen
```

Diese neue Erkenntnis kann man jetzt benutzen, um nicht nur einzelne Werte sondern ganze Bereiche von Matrizen zu belegen.

```
A=eye(6), A(1:3,4:6)=ones(3)
x=ones(1,10), x(1:2:10)=zeros(1,5)
A(1,:) % Doppelpunkt ohne Begrenzung heisst ALLES
A(:,2) % hier also die zweite Spalte
```

Sollte eine Rechnung eine Matrix unbekannter Länge erzeugen, kann man diese Dimension von MATLAB/Octave ausgeben lassen.

```
A=[1,2,3;4,5,6] % nur zum Testen
size(A) % liefert [Zeilen, Spalten]
[m,n] =size(A) % jetzt stehen die Werte in m und n
m
n
length(A) % liefert den groesseren der beiden Werte zurueck
x=1:10
x(length(x)) % liefert den letzten Wert von x
x(end) % oder einfach so
A(end,1) % unten links
A(:,end) % liefert die letzte Spalte
```

Hilfreiche Befehle, eine Matrix zu formen sind auch `diag`, `tril` (*tri*angle, *l*ower) und `triu` (*tri*angle, *u*pper). Dabei ist `diag` in zwei Varianten nutzbar. Zum einen liefert es die Diagonalelemente einer Matrix als Vektor zurück. Mit einem Vektor als Argument liefert es die Matrix, die genau diese Elemente auf der Diagonalen hat (und sonst Nullen).

```
v=1:5 A=diag(v) % v auf der Diagonalen
B=diag(v,1) % v auf der oberen 1. Nebendiagonalen
L=tril(ones(5)) % ein Dreieck aus Einsen
R=triu(ones(5)) % noch ein Dreieck
```

2.4 Rechnen mit Matrizen

Sind die Matrizen einmal definiert, dann kann man einfach und komfortabel mit ihnen rechnen.

```
A=[1,2;2,4]
B=[1,1;-1,1]
A+B % selbsterklaerend
A*B % selbsterklaerend
inv(B) % die Inverse von B
inv(A) % Fehler, weil A singulaer
A*inv(B)
A/B % Kurzform fuer A*inv(B)
B\A %Kurzform fuer inv(B)*A
c=[1;3]
x=B\c % loest das Gleichungssystem B*x=c
```

3 Scripts und Funktionen

Häufig will man nicht nur einen Befehl auf eine Matrix anwenden sondern eine ganze Reihe von Manipulationen. Dies kann man in der Kommandozeile dadurch erreichen, dass man die Befehle, durch Komma getrennt hintereinanderschreibt. Nach Drücken der Eingabetaste werden alle Befehle nacheinander abgearbeitet und die jeweiligen Zwischenergebnisse ausgegeben. Trennt man die einzelnen Befehle statt durch Komma durch Semikolon, dann wird die Einfgabe unterdrückt. Dies ist unbedingt zu beachten, wenn man umfangreichere Berechnungen durchführt. Häufig ist dann die Ausgabe der Zwischenergebnisse nicht erwünscht und verlangsamt die Berechnung nur massiv.
Hat sich jetzt ein Fehler in die Zeile mit den vielen Befehlen eingeschlichen, dann darf man alles erneut eintippen. (Oder man erinnert sich daran, was die Pfeil-nach-oben-Taste bewirkt.) Viel einfacher ist es, statt dessen, ein *Script* anzulegen. Dies ist nichts anderes als eine Textdatei, in die man alle Befehle hineinschreibt - genauso, wie sie in der Kommandozeile eingegeben werden. Diese Datei speichert man unter einem geeigneten Namen, der aber auf `.m` endet. Dieses sogenannte m-File kann dann von MATLAB/Octave wie ein Befehl ausgeführt werden. Der Befehl hat den gleichen Namen, wie das m-File ohne Endung.

```
% diesen Text als ecksumme.m abspeichern!
[m,n]=size(A); % Zeilen- und Spaltenzahl bestimmen
e=A(1,1)+A(1,n)+A(m,1)+A(m,n) % Summe der Eckelemente von A
```

Nach dem Speichern dieser Datei kann man jetzt durch `ecksumme` die Summe der Elemente in den Ecken der Matrix `A` ausrechnen lassen. Wichtig ist dabei, dass man sich in dem Verzeichnis befindet, in dem das m-File liegt. Gegebenenfalls muss man den Pfad über das Pfadmenu wechseln. Alternativ kann man auch die `PATH` Variable entsprechend setzen. Aber das führt hier zu weit.

```
A=[1,2,3;4,5,6;7,8,9]
ecksumme % klappt
B=ones(5)
ecksumme % klappt nicht
A=ones(5)
ecksumme % klappt wieder
```

Es sollte nicht überraschen, dass (wenn alles richtig eingetippt wurde) der erste Aufruf von `ecksumme` den korrekten Wert 20 liefert. Der zweite Aufruf liefert aber nicht die Ecksumme 4 von `B` sondern wieder 20. Erst der dritte Aufruf liefert den korrekten Wert. Das wiederum ist nicht verwunderlich, wenn man sich das Script `ecksumme.m` genauer ansieht. Dort wird explizit die Ecksumme von `A` berechnet. Der Inhalt von `B` ist vollkommen egal. Das Script ist bisher auch nichts anderes als eine Zusammenfassung von Befehlen, wie sie in der Kommandozeile eingegeben werden. Wir ändern das m-File `ecksumme.m` jetzt folgendermaßen um:

```
% diesen Text als ecksumme.m abspeichern!
function e=ecksumme(A) % Das ist neu!
[m,n]=size(A); % Zeilen- und Spaltenzahl bestimmen
e=A(1,1)+A(1,n)+A(m,1)+A(m,n) % Summe der Eckelemente von A
```

und probieren gleich wieder.

```
A=[1,2,3;4,5,6;7,8,9]
ecksumme % klappt nicht mehr
ecksumme(A) % klappt wieder
B=ones(5)
ecksumme(B) % klappt auch
```

Die erste Zeile des neuen m-Files kennzeichnet dies jetzt als eine *Funktion*. Konkret steht hier, dass `ecksumme` ein Argument in Klammern erwartet und das im weiteren Verlauf als `A` verwendet wird. Alle Änderungen an Variablen, die in der Funktion passieren, sind vollkommen unabhängig von dem, was in der Kommandozeile gemacht wird. Weiterhin wird `e` als Ausgabe deklariert. Das heißt, im Verlauf der Funktion muss eine Variable `e` mit einem Wert belegt werden, der später an die Kommandozeile zurückgegeben wird. Passiert das nicht, treten schwer identifizierbare Fehler auf. Ändere die Anweisung

```
e=A(1,1)+A(1,n)+A(m,1)+A(m,n)
```

zu `f=...`(Rest der Zeile); versuche die Tests erneut und merke Dir, welche Rückmeldungen kommen. Einen Rückgabewert zu vergessen, ist ein beliebter Fehler beim Programmieren. MATLAB kommt uns da entgegen und warnt. Octave bleibt stumm und lässt uns im Dunkeln.

3.1 Schleifen

```
% diesen Text als diagsumme.m abspeichern!
function s=diagsumme(A) % Funktionskopf
[m,n]=size(A); % Zeilen- und Spaltenzahl bestimmen
s=0; % s initialisieren
for i=1:n % Schleife: i laeuft von 1 bis n
  s=s+A(i,i); % schrittweise Diagonalelemente addieren
end % Ende der Schleife
```

In diesem Beispiel wird wieder die Größe von `A` bestimmt. Dann wird einer Laufvariablen `i` nacheinander je ein Wert von 1 bis `n` (Spaltenzahl) zugewiesen. Statt `1:n` könnte jeder andere Vektor dort stehen. `for i=[1,23,42]` lässt `i` nacheinander die Werte 1, 23 und 42 annehmen. Mit jedem dieser Werte wird der Schleifenkörper bis zum `end` durchlaufen und dessen Inhalt ausgeführt. Hier wird zu einer Variablen `s` jeweils das `i`-te Diagonalelement von `A` addiert. Wird die Initialisierung `s=0` in Zeile 3 vergessen, dann gibt es in Zeile 5 kein `s` zu dem addiert werden kann und es wird ein Fehler ausgegeben. Probiere es aus.

```
A=[1,2,3;4,5,6;7,8,9]
diagsumem(A) % korrekt
diagsumme(A(:,1:2)) % nur die ersten 2 Spalten
diagsumme(A(1:2,:)) % nur die oberen 2 Zeilen
```

Dieses Beispiel zeigt, dass das Programm größtenteils korrekt funktioniert. Nur im dritten Beispiel wird ein Fehler angezeigt, weil versucht wird, auf das (3,3) Diagonalelement zuzugreifen. Bei dem Beispiel, in dem nur die ersten zwei Zeilen von `A` verwendet werden, existiert dieses aber nicht. Das Problem liegt darin,

dass wir die Schleife bis `n`, also zur Spaltenzahl laufen lassen. Ist eine Matrix breiter als hoch (mehr Spalten als Zeilen), werden zwangsläufig solche Probleme auftreten.

3.2 Abfragen

```
% diesen Text als diagsumme.m abspeichern!
function s=diagsumme(A) % Funktionskopf
[m,n]=size(A); % Zeilen- und Spaltenzahl bestimmen
s=0; % s initialisieren
if (m<n) % ab hier neu
  grenze=m;
else % Alternative
  grenze=n
end %Ende der Abfragen
for i=1:grenze % Schleife: i laeuft von 1 bis n
  s=s+A(i,i); % schrittweise Diagonalelemente addieren
end % Ende der Schleife
```

Die neu eingefügten Zeilen sind eine `if-else` Abfrage. Diese haben im einfachsten Fall die Form:

```
if (Bedingung) ... end
```

Ist die Bedingung beim Erreichen der Abfrage erfüllt, wird der Block bis zum `end` ausgeführt, andernfalls übersprungen. Die hier auftretende Form

```
if (Bedingung) ... else ... end
```

lässt eine Alternative zu. Der Block zwischen `else` und `end` wird ausgeführt, wenn die Bedingung nicht erfüllt ist.

```
% diesen Text als fakultaet.m abspeichern!
function f=fakultaet(n) % Funktionskopf
if (n==0)
  f=1;
else
  f=n*fakultaet(n-1)
end
```

Hier werden zwei weitere Dinge gezeigt. Zum einen sieht man in Zeile 2 eine Gleichheitsabfrage. Diese wird durch `==` realisiert. Achtung! Ein einfaches `=` ist eine Zuweisung und wird in diesem Beispiel eine kryptische Fehlermeldung liefern. Zum zweiten wird in Zeile 5 die Funktion `fakultaet` von sich selbst aufgerufen, also rekursiv. Überlege, warum das in diesem Fall keine Endlosschleife liefert! Außerdem sollte am Ende der 5. Zeile unbedingt ein Semikolon angefügt werden. Probiere es aus!

Da kommt noch mehr!