Octave in 16 Beispielen

In dieser Sammlung soll anhand von Beispielen das Programm Octave näher gebracht werden. Die verwendeten Skripte ab Beispiel 10 sind in extra Ordnern mit der Beispielnummer als Namen zu finden. Die Skripte können in einem eigenen Ordner gespeichert und bearbeitet werden. In dem Glossar im Anhang, sind einige der hier verwendeten Befehle kurz zusammengefasst.

Beispiel 1:
Es sollen die einfachsten Grundrechenarten und das Anlegen einer Variable kennen gelernt werden. Einfache Berechnungen können auch direkt im sogenannten Befehlsfenster eingegeben und durchgeführt werden. Über dem Befehlsfenster befindet sich der sogenannte (Skript-) Editor. Dort können ganze Programme (-teile) geschrieben werden. Den Editor lernen wir in Beispiel 10 genauer kennen.
Es werden folgende Schritte im Befehlsfenster durchgeführt:

1. Variable *x* wird angelegt. Variablen werden standardmäßig im double-Format gespeichert:

   >> x=10

2. „Ergebnis“ aus voriger Zeile wird automatisch von Octave angezeigt:

   x = 10

3. Um das Anzeigen des Ergebnisses zu unterdrücken, kann das Semikolon verwendet werden:

   >> x=10;

4. Vorher festgelegte Variablen können für spätere Berechnungen genutzt werden, da sie im sogenannten workspace gespeichert werden:

   *>> a=2*x*

5. Ausgabe mit zuvor festgelegtem Ergebnisnamen:

   a=20

6. Octave kann auch als einfacher Taschenrechner genutzt werden. Es muss kein Variablenname für das Ergebnis festgelegt werden. Es wird dann standardmäßig unter *ans* gespeichert:

   >> 60(a/2)*

7. Ausgabe mit automatisch vergebenem Ergebnisnamen:

   ans = 600

8. Auch *ans* ist ein normaler Variablenname und kann als solcher verwendet werden. Anstatt eines Kommas muss der Punkt verwendet werden:

   *>> y=ans*0.5*

9. Ergebnis:
   y = 300

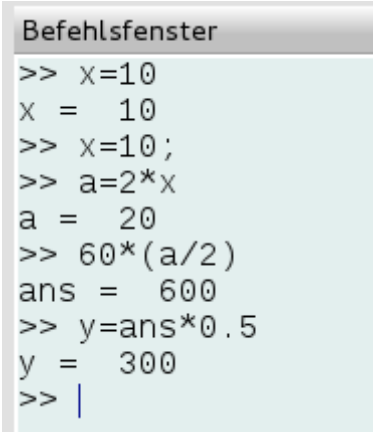

Abbildung 1: Gesamter Code

Beispiel 2:
Es sollen ein paar der Octaveintern festgelegten Konstanten kennengelernt werden. Es wird die Wurzel gezogen, quadriert und eine komplexe Zahl angelegt. Zusätzlich werden einige Hilfsfunktionen präsentiert.

1. Die Konstanten π und *e* werden ausgegeben:

   >> pi
   ans = 3.1416
   >> e
   ans = 2.7183

2. Es wird die Wurzel aus 10 gezogen:

   >> y= sqrt(10)
   y = 3.1623

3. Das Ergebnis der vorigen Berechnung wird quadriert:

   >> a=y^2
   a = 10.000

4. Es wird die komplexe Zahl *i* ausgegeben, ihr Betrag ermittelt und eine weitere komplexe Zahl namens *kompl* definiert:

>> i
ans = 0 + 1i
>> abs(i)
ans = 1
>>kompl= 2+ 3i
kompl = 2 + 3i

5. Es können alle im workspace gespeicherten Variablen mit dem Befehl *who* angezeigt werden:

   >> who
   Variables in the current scope:
   a ans kompl y

6. Mit *clear* bzw. *clear all* können einzelne bzw. alle Variablen gelöscht werden:

   >> clear ans
   Variables in the current scope:
   a kompl y

7. Möchte man mehr über einen Befehl oder eine Funktion erfahren, kann dies folgendermaßen im Befehlsfenster allgemein realisiert werden:

   >> help Befehl

   Ocatve gibt dann eine kleine Informationsübersicht mit kurzen Erklärungen und Anwendungsbeispielen zu *Befehl*.

Befehlsfenster

```
>> pi
ans =  3.1416
>> e
ans =  2.7183
>> y=sqrt(10)
y =  3.1623
>> a=y^2
a =  10.000
>> i
ans =  0 + 1i
>> abs(i)
ans =  1
>> kompl= 2+3i
kompl =  2 + 3i
>> who
Variables in the current scope:

a      ans    kompl  y
>> clear ans
>> who
Variables in the current scope:

a      kompl  y
```

Abbildung 2: Gesamter Code

Beispiel 3:
Es werden Zeilen- und Spaltenvektoren angelegt und verändert. Einige Standardvektoren werden erzeugt.

1. Der Zeilenvektor *x* wird auf zwei verschiedene Weisen angelegt:

```
>> x=[1 2 3]
x =
     1     2     3
>>x=[1,2,3]
x =
     1     2     3
```

2. Der Spaltenvektor *y* wird auf zwei verschiedene Weisen angelegt (Schritt 2: Transponieren eines Zeilenvektors):

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

3. Der erste Eintrag des in den vorigen Schritten angelegten *x*-Vektors und der letzte Eintrag des *y*-Vektors soll angezeigt werden. Dann soll die erste Stelle des *x*-Vektors zu 7 verändert werden:

```
>>x(1)
ans = 1
>>y(end)
ans = 3
>>x(1)=7
x =
     7     2     3
```

4. Es wird die Länge des Vektors *x* ermittelt. Es sollen Vektoren, die sich über längere Bereiche erstrecken, mittels Doppelpunkt-Operator definiert werden. Zusätzlich werden für diese unterschiedliche Schrittgrößen festgelegt. Der Vektor *c* soll von 6 auf 2 in Einer-Schritten runtergezählt werden:

```
>>length(x)
ans = 3
>>a=[2:6]
a =
     2     3     4     5     6
>>b=[2:2:6]
```

```
b =
    2     4     6
>>c=[6:-1:2]
c =
    6     5     4     3     2
```

5. Die Standardvektoren aus Nullen und Einsen werden erzeugt:

```
>>ones(3,1)
ans =
     1
     1
     1
>>zeros(1,2)
ans =
    0     0
```

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

   1   2   3

>> x=[1,2,3]
x =

   1   2   3

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

   1
   2
   3
```

Abbildung 5: Code zu Schritt 1 und 2

```
>> y=[1 2 3 ]'
y =

   1
   2
   3

>> x(1)
ans =  1
>> y(end)
ans =  3
>> x(1)=7
x =

   7   2   3

>> length(x)
ans =  3
```

Abbildung 3: Code zu Schritt 3 und 4

```
>> a=[2:6]
a =

   2   3   4   5   6

>> b=[2:2:6]
b =

   2   4   6

>> c=[6:-1:2]
c =

   6   5   4   3   2
```

Abbildung 4: Code zu Schritt 4

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

   1
   1
   1

>> zeros(1,2)
ans =

   0   0
```

Abbildung 6: Code zu Schritt 5

Beispiel 4:
Vektoren sollen multipliziert und potenziert werden. Zusätzlich werden Skalar- und Kreuzprodukt gebildet.

1. Es werden die Vektoren *x= 1 2 3* und *y= 4 2 1* angelegt. *z* ist die Differenz dieser beiden Vektoren:

   ```
   >> x=[1 2 3];
   >> y=[4 2 1];
   >> z=x-y
   z =
       -3   0   2
   ```

2. Vektor *a* ist das Doppelte des Vektors *z:*

   ```
   >> a=2*z
   a =
        -6   0   4
   ```

3. Mit dem Punkt-Operator können Anweisungen komponentenweise ausgeführt werden. Das kann z.B. zum komponentenweisen Potenzieren eines Vektors genutzt werden:

   ```
   >> b=a.^2
   b =
       36    0   16
   ```

4. Es soll das Skalarprodukt von *a* und *b* gebildet werden. Dies geschieht mit der *dot*-Anweisung:

   ```
   >> dot(a,b)
   ans = -152
   ```

5. Das Kreuzprodukt von *x* und *z* wird mit dem *cross*-Befehl gebildet:

```
>> cross(x,z)
ans =
     4   -11    6
```

```
>> x=[1 2 3];
>> y=[4 2 1];
>> z=x-y
z =

  -3   0   2

>> a=2*z
a =

  -6   0   4

>> b=a.^2
b =

   36    0   16

>> dot(a,b)
ans = -152
>> cross(x,z)
ans =

    4  -11    6
```

Abbildung 7: Gesamter Code

Beispiel 5:
Es sollen Vektoren dividiert werden. Des Weiteren soll über das Befehlsfenster zur Eingabe aufgefordert und die Antwort verarbeitet (hier: angezeigt) werden. Zum Schluss wird die formatierte Ausgabe gezeigt.

1. Es wird der Vektor *a=[1 2 3]* angelegt und halbiert:

```
>> a=[1 2 3];
>> b=a/2
b =
0.50000   1.00000   1.50000
```

2. Der Vektor *c=[4 5 6]* wird angelegt und *a* und *c* in unterschiedlicher Weise dividiert:

```
>> c=[4 5 6];
>> a/c
ans =  0.41558
>> a\c
ans =
    0.28571   0.35714   0.42857
```

0.57143 0.71429 0.85714
0.85714 1.07143 1.28571

Dabei entspricht a/c = a * c^{-1} und a\c = a^{-1} * c

3. Es wird über das Befehlsfenster dazu aufgefordert eine Zahl einzugeben und diese anschießend angezeigt.

>> x=input('Geben Sie eine Zahl ein:')
Geben Sie eine Zahl ein:2016
x = 2016
>> y=disp(x)
y = 2016

4. Es wird die Zahl *zahl=359* definiert. Sie wird mit der formatierten Ausgabe *fprintf* ausgegeben.

>> zahl=359;
>> fprintf('Die Zahl lautet: %d\n', zahl)
Die Zahl lautet: 359
>> fprintf('Die Zahl lautet: %7.3f\n', zahl)
Die Zahl lautet: 359.000

Die Anhabe hinter dem Prozentzeichen definiert den Typ der einzugebenden Variable. *d* steht für eine Ganzzahl und *f* für eine Gleitkommazahl. 7.3 bedeutet, dass die Zahl insgesamt 7 Stellen belegt, 3 davon sind Nachkommastellen.

```
>> a=[1 2 3];
>> b=a/2
b =

   0.50000   1.00000   1.50000

>> c=[4 5 6];
>> a/c
ans =  0.41558
>> a\c
ans =

   0.28571   0.35714   0.42857
   0.57143   0.71429   0.85714
   0.85714   1.07143   1.28571
```

Abbildung 8: Code zu Schritt 1 und 2

```
>> x=input('Geben Sie eine Zahl ein:')
Geben Sie eine Zahl ein:2016
x =  2016
>> y=disp(x)
y =  2016
```

Abbildung 9: Code zu Schritt 3

```
>> zahl=359;
>> fprintf('Die Zahl lautet: %d\n', zahl)
Die Zahl lautet: 359
>> fprintf('Die Zahl lautet: %7.3f\n', zahl)
Die Zahl lautet: 359.000
```

Abbildung 10: Code zu Schritt 4

Beispiel 6:
Es soll eine Matrix aus einem Zeilen- und einem Spaltenvektor und direkt gebildet werden. Es wird die Matrizen-Dimension abgefragt. Anschließend werden die Grundrechenarten durchgeführt.

1. Es wird eine Matrix aus dem Zeilenvektor *a=[1 2 3]* und dem Spaltenvektor *b=[1;2;3]* gebildet werden:

```
>> a=[1 2 3];
>> b=[1;2;3];
>> b*a
ans =
    1   2   3
    2   4   6
    3   6   9
```

2. Es werden zwei Matrizen *c* und *d* direkt gebildet:

```
>> c=[1 2 3; 4 5 6]
c =
    1   2   3
    4   5   6
>> d=[1 2; 3 4; 5 6]
d =
    1   2
    3   4
    5   6
```

3. Es werden drei Matrizen *a,b* und *c* gebildet und ihre Dimensionen abgefragt:

```
>> a=[1 2 4; 6 8 10]
a =
    1    2    4
    6    8   10
>> b=[3 7 6; 0 6 4]
```

```
b =
    3   7   6
    0   6   4
>> c=[5 6; 5 9; 3 7]
c =
    5   6
    5   9
    3   7
>> size(a)
ans =
    2   3
>> size(b)
ans =
    2   3
>> size(c)
ans =
    3   2
```

Die erste Zahl der Dimension ist die Anzahl der Zeilen, die zweite die der Spalten.

4. Zwei Matrizen *a* und *b* werden addiert und subtrahiert:

```
>> a=[10 15 20; 30 5 40]
a =
    10   15   20
    30    5   40
>> b=[20 30 40; 5 15 25]
b =
    20   30   40
    5   15   25
>> a+b
ans =
    30   45   60
    35   20   65
>> a-b
ans =
   -10  -15  -20
    25  -10   15
```

Für die Addition und Subtraktion müssen die Dimensionen der Matrizen übereinstimmen.

5. Zwei Matrizen werden multipliziert und dividiert:

```
>> a=[1 2 3; 4 5 6]
a =
    1   2   3
    4   5   6
>> b=[1 2; 3 4; 5 6]
b =
    1   2
```

```
    3   4
    5   6
>> a*b
ans =
    22   28
    49   64
>> c= [4 6 8; 10 12 14]
c =
    4    6    8
    10   12   14
>> d=[20 10 15; 50 60 2]
d =
    20   10   15
    50   60    2
>> c/d
ans =
    0.3665027  -0.0035058
    0.6790387   0.0231269
>> c\d
ans =
    4.16667   12.77778   -7.72222
    1.66667    2.77778   -0.72222
    -0.83333   -7.22222    6.27778
```

Grundrechenarten haben gleiche Funktionsweise wie bei Vektoren

```
>> a=[1 2 3];
>> b=[1;2;3];
>> b*a
ans =

   1   2   3
   2   4   6
   3   6   9

>> c=[1 2 3; 4 5 6]
c =

   1   2   3
   4   5   6

>> d=[1 2; 3 4; 5 6]
d =

   1   2
   3   4
   5   6
```

Abbildung 11: Code zu Schritt 1 und 2

```
>> a=[1 2 4; 6 8 10]
a =

    1    2    4
    6    8   10

>> b=[3 7 6; 0 6 4]
b =

    3    7    6
    0    6    4

>> c=[5 6; 5 9; 3 7]
c =

    5    6
    5    9
    3    7

>> size(a)
ans =

    2    3

>> size(b)
ans =

    2    3

>> size(c)
ans =

    3    2
```

Abbildung 13: Code zu Schritt 3

```
>> a=[10 15 20; 30 5 40]
a =

   10   15   20
   30    5   40

>> b=[20 30 40; 5 15 25]
b =

   20   30   40
    5   15   25

>> a+b
ans =

   30   45   60
   35   20   65

>> a-b
ans =

  -10  -15  -20
   25  -10   15
```

Abbildung 12: Code zu Schritt 4

```
Befehlsfenster
>> a=[1 2 3; 4 5 6]
a =

   1   2   3
   4   5   6

>> b=[1 2; 3 4; 5 6]
b =

   1   2
   3   4
   5   6

>> a*b
ans =

   22   28
   49   64

>> c= [4 6 8; 10 12 14]
c =

    4    6    8
   10   12   14

>> d=[20 10 15; 50 60 2]
d =

   20   10   15
   50   60    2
```

Abbildung 14: Code zu Schritt 5 (Teil 1)

```
>> c/d
ans =

   0.3665027  -0.0035058
   0.6790387   0.0231269

>> c\d
ans =

    4.16667   12.77778   -7.72222
    1.66667    2.77778   -0.72222
   -0.83333   -7.22222    6.27778
```

Abbildung 15: Code zu Schritt 5 (Teil 2)

Beispiel 7:
Es werden ein paar spezielle Matrizen erzeugt. Des Weiteren werden einige Matrizen-Operationen benutzt.

1. Es werden Null-, Einser- und Einheitsmatrizen erzeugt:

```
>> N=zeros(2)
N =
    0   0
    0   0
>> N2=zeros(3,2)
N2 =
    0   0
    0   0
    0   0
```

```
>> E= ones(1,5)
E =
     1   1   1   1   1
>> I=eye(2)
I =
Diagonal Matrix
     1   0
     0   1
>> I2=eye(2,3)
I2 =
Diagonal Matrix
     1   0   0
     0   1   0
```

2. Es werden Matrizen erzeugt, deren Diagonale mit beliebigen Zahlen und der Rest mit Nullen belegt werden:

```
>> D=diag([1 5])
D =
Diagonal Matrix
     1   0
     0   5
>> D2=diag([1 6 28])
D2 =
Diagonal Matrix
     1    0    0
     0    6    0
     0    0   28
```

3. Es wird eine 3x3-Matrix erzeugt. Es wird ihre Länge, ihre Determinante, ihr Inverse, ihr Rang und die Eigenwerte bestimmt:

```
>> A=[52 96 12; 26 48 3; 4 2 6];
>> length(A)
ans =  3
>> det(A)
ans = -840
>> inv(A)
ans =
    -0.33571   0.65714   0.34286
     0.17143  -0.31429  -0.18571
     0.16667  -0.33333   0.00000
>> rank(A)
ans =  3
>> eig(A)
ans =
     100.4831
     -1.2376
     6.7546
```

```
>> N=zeros(2)
N =

   0   0
   0   0

>> N2=zeros(3,2)
N2 =

   0   0
   0   0
   0   0

>> E= ones(1,5)
E =

   1   1   1   1   1

>> I=eye(2)
I =

Diagonal Matrix

   1   0
   0   1

>> I2=eye(2,3)
I2 =

Diagonal Matrix

   1   0   0
   0   1   0
```

Abbildung 16: Code zu Schritt 1

```
>> D=diag([1 5])
D =

Diagonal Matrix

   1   0
   0   5

>> D2=diag([1 6 28])
D2 =

Diagonal Matrix

    1    0    0
    0    6    0
    0    0   28
```

Abbildung 17: Code zu Schritt 2

```
>> A=[52 96 12; 26 48 3; 4 2 6];
>> length(A)
ans =  3
>> det(A)
ans = -840
>> inv(A)
ans =

  -0.33571   0.65714   0.34286
   0.17143  -0.31429  -0.18571
   0.16667  -0.33333   0.00000

>> rank(A)
ans =  3
>> eig(A)
ans =

   100.4831
    -1.2376
     6.7546
```

Abbildung 18: Code zu Schritt 3

Beispiel 8:
Es soll ein Polynom erstellt werden. Anschließend werden beliebige Werte für *x* eingesetzt und der Wert des Polynoms berechnet. Dann werden für das Polynom die Nullstellen bestimmt und umgekehrt ein Polynom aus gegebenen Nullstellen erstellt. Es wird die Polynom-Multiplikation und -Division durchgeführt. Am Ende wird das Polynom differenziert.

1. Ein Polynom *p* wird erstellt und für *x=3* ausgewertet. Es werden die Nullstellen des Polynoms bestimmt:

   ```
   >> p=[5 6 2]
   p =
       5   6   2
   >> x=3;
   >> polyval(p,x)
   ans =  65
   >> roots(p)
   ans =
       -0.60000 + 0.20000i
       -0.60000 – 0.20000i
   ```

   Das erstellte Polynom sieht wie folgt aus:
   $p= 5 * x^2 + 6 * x + 2$

2. Aus den gegeben Nullstellen *x1=5* und *x2=3* wird ein Polynom erstellt:

   ```
   >> p2=poly([5 3])
   p2 =    1   -8   15
   ```

Das erzeugte Polynom sieht wie folgt aus:
p2= 1 * x² - 8 * x + 15

3. Die zwei Polynome *p* und *p2* werden differenziert:

```
>> polyder(p2)
ans =
    2  -8
>> polyder(p)
ans =
    10    6
```

4. Die beiden Polynome werden multipliziert und dividiert:

```
>> conv(p,p2)
ans =
    5  -34   29   74   30
>> deconv(p,p2)
ans =  5
```

Die Multiplikation sieht wie folgt aus: (5 * x² + 6 * x + 2) * (1 * x² - 8 * x + 15)
Die Division sieht wie folgt aus: (5 * x² + 6 * x + 2) / (1 * x² - 8 * x + 15)

```
>> p=[5 6 2]
p =

   5   6   2

>> x=3;
>> polyval(p,x)
ans =  65
>> roots(p)
ans =

  -0.60000 + 0.20000i
  -0.60000 - 0.20000i

>> p2=poly([5 3])
p2 =

    1   -8   15

>> polyder(p2)
ans =

   2  -8

>> polyder(p)
ans =

   10    6
```

Abbildung 19: Code zu Schritt 1, 2 und 3

```
>> conv(p,p2)
ans =

    5  -34   29   74   30

>> deconv(p,p2)
ans =  5
```

Abbildung 20: Code zu Schritt 4

Beispiel 9:
Es soll eine Sinusfunktion und eine Exponentialfunktion (logarithmisch) graphisch in einem Plot dargestellt werden.

1. Es wird die Sinusfunktion in Abhängigkeit von *x* erstellt. Dabei soll der Sinus von 0 bis 10 abgebildet werden.

   ```
   >> x=linspace(0,10,100);
   >> y=sin(x);
   >> plot(x,y)
   ```

   linspace erzeugt 100 äquidistante Werte zwischen 0 und 10.

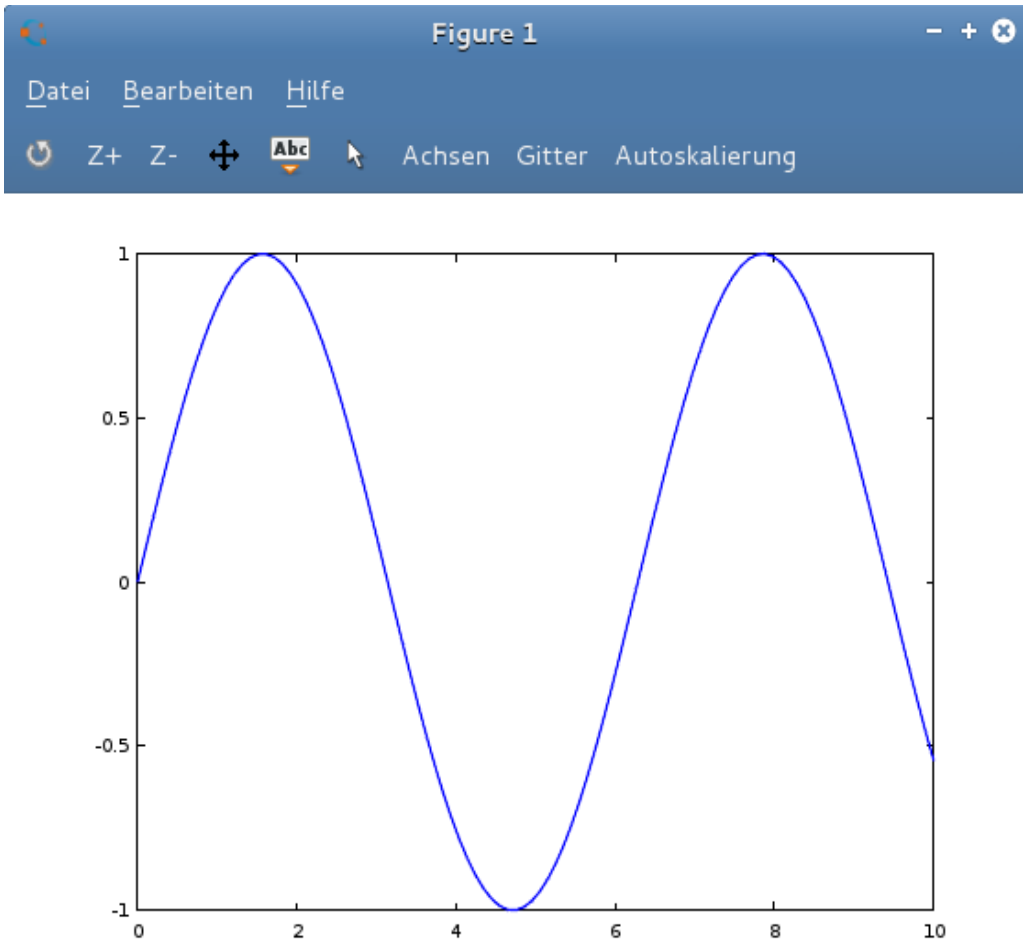

Abbildung 21: Figure zu Schritt 1

2. Der Plot der gleichen Funktion soll nun Achsenbeschriftungen, einen Titel und eine neue Einteilung der Achsen bekommen:

   ```
   >> x=linspace(0,10,100);
   >> y=sin(x);
   >> plot(x,y);
   >> xlabel('Zeit');
   >> ylabel('Amplitude');
   >> title('Erster Plot');
   ```

```
>> axis([0 10 -1.5 1.5]);
```

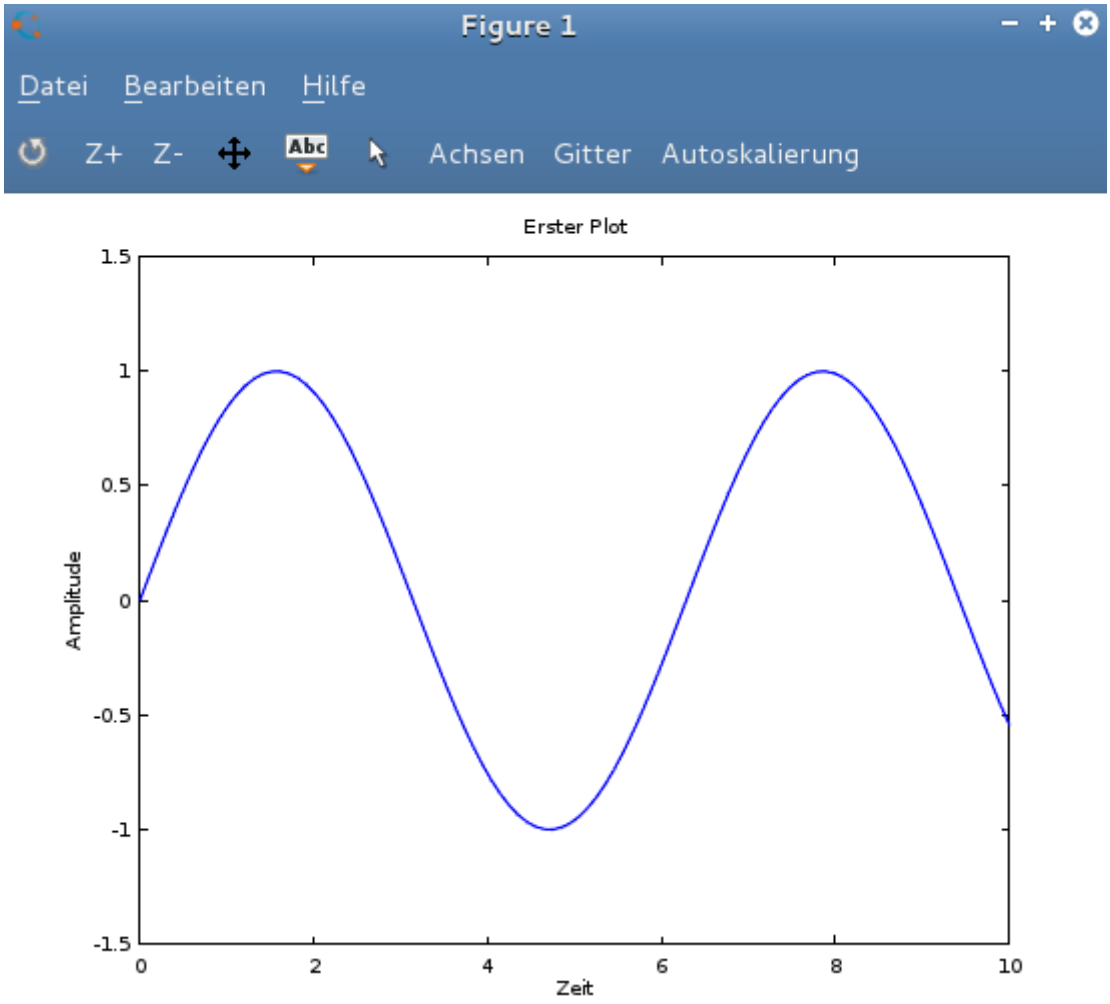

Abbildung 22: Figure zu Schritt 2

3. Es wir eine Exponentialfunktion *y=exp(0.5*x)* erstellt, wobei *x* von 0 bis 10 in 100 (äquidistanten) Schritten laufen soll. Die y-Achse soll logarithmisch dargestellt werden:

```
>> x=linspace(0,10,100);
>> y=exp(0.5*x);
>> plot(x,y);
>> semilogy(x,y);
```

semilogy bzw. *semilogx* erzeugen bei der jeweils angegeben Achse eine logarithmische Teilung. Möchte man beide Achsen gleichzeitig logarithmisch auftragen, ist die Funktion *loglog(x,y)* zu wählen.

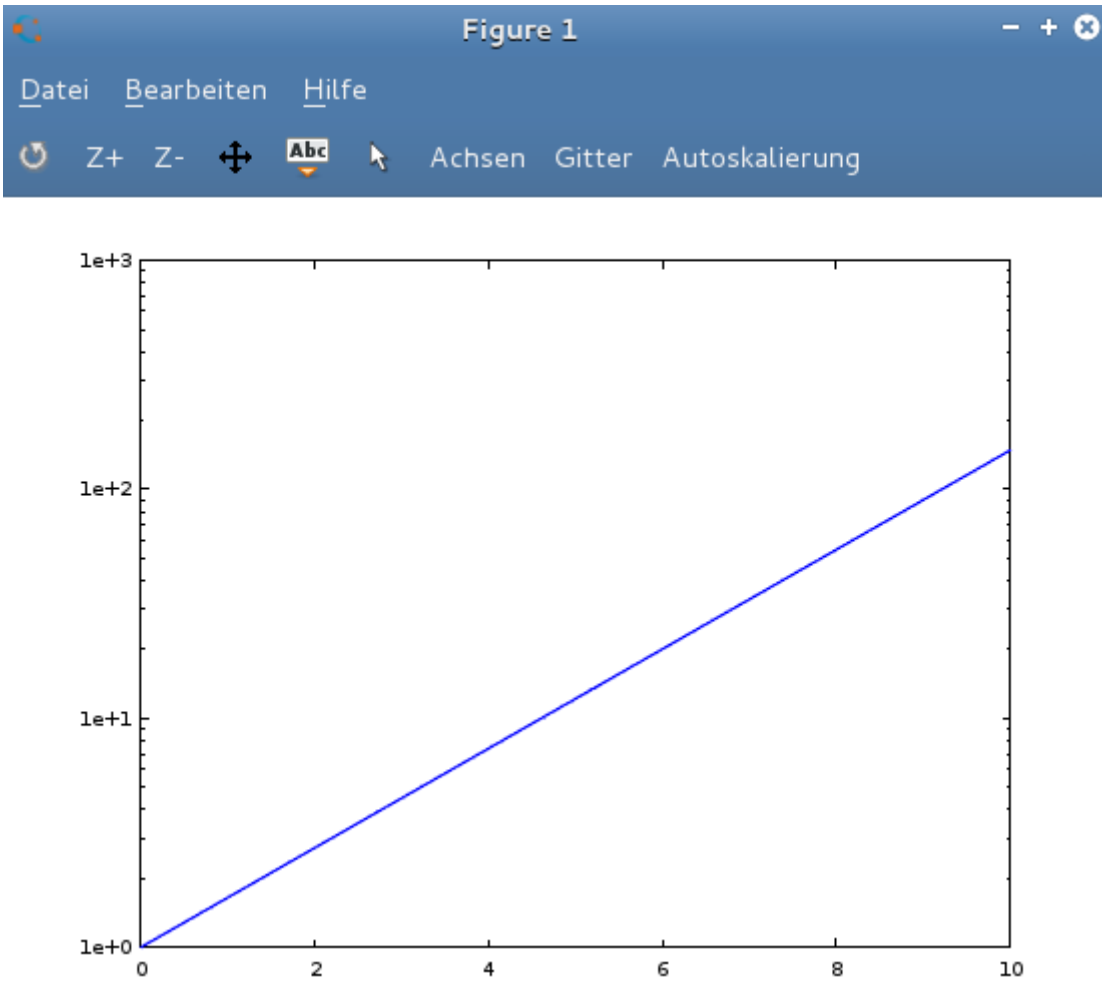

Abbildung 23: Figure zu Schritt 3

```
>> x=linspace(0,10,100);
>> y=sin(x);
>> plot(x,y)
```
Abbildung 24: Code zu Schritt 1

```
>> x=linspace(0,10,100);
>> y=sin(x);
>> plot(x,y)
>> xlabel('Zeit');
>> ylabel('Amplitude');
>> title('Erster Plot');
>> axis([0 10 -1.5 1.5]);
```
Abbildung 25: Code zu Schritt 2

```
>> x=linspace(0,10,100);
>> y=exp(0.5*x);
>> plot(x,y);
>> semilogy(x,y);
```
Abbildung 26: Code zu Schritt 3

Beispiel 10:
Ein Skript soll angelegt und benutzt werden. In Skripten können aufwändigere Befehl (-ketten) geschrieben werden.

1. Anlegen eines Skriptes:
   Das obere Fenster (Editor) dient der Eingabe und dem Aufrufen der Skripte. Zum Anlegen eines neuen Skriptes kann der Datei-Button gedrückt und „Neues Skript“ gewählt werden. Skripte werden als .m-Dateien gespeichert, hier z.B. *erstesSkript.m*. Wichtig ist dabei, dass das Skript im selben Verzeichnis wie Octave gespeichert ist. Das Skript kann sonst nicht auf die in Octave definierten Variablen zugreifen. Ein häufiger Fehler besteht somit auch darin, dass nicht der richtige Speicherpfad zu Beginn gewählt wurde. Unter „Aktuelles Verzeichnis“ kann in Octave der richtige Pfad eingestellt werden.

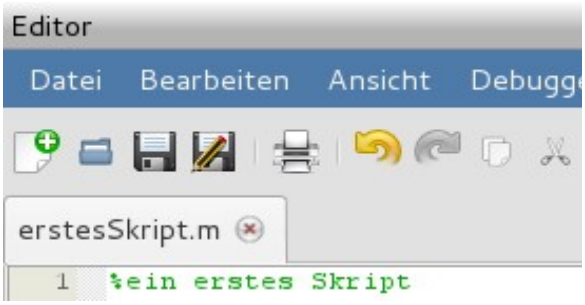

Abbildung 27: Skript im Editor zu Schritt 1

   Ein Sternchen vor dem Dateinamen bedeutet, dass das Skript nach Änderungen noch nicht abgespeichert wurde.

2. Erstellen eines Skriptes:
   Nun wird das Skript mit Befehlen „gefüllt“. Hier soll das Skript lediglich zwei Zahlen addieren, subtrahieren und ausgeben. Die Prozentzeichen kennzeichnen Kommentare und werden somit von Octave nicht beachtet.

```
%ein erstes Skript
a=10;
b=20;
c=a+b
d=b-a
disp(c);
disp(d);
```

3. Aufrufen des Skriptes:
   Im Befehlsfenster kann nun das Skript unter seinem Speicher-Namen aufgerufen und somit durchführt werden.

```
>> erstesSkript
c =  30
d =  10
    30
    10
```

Auch hier gilt, dass ein Semikolon die Ausgabe unterdrückt.

4. Es wird ein neues Skript namens *BEingabe* angelegt. Dabei soll im Befehlsfenster zur Eingabe des Namen aufgefordert werden und der Name anschließend angezeigt werden.

```
%Skript zur Benutzer_innen-Eingabe
x=input('Bitte geben Sie Ihren Namen ein: ', 'str');
disp(x);
```

Damit ein Wort eingegeben werden kann, muss das Format String gewählt werden, indem in die Klammer der Hinweis *'str'* geschrieben wird. Alternativ kann dafür auch *'s'* genutzt werden.

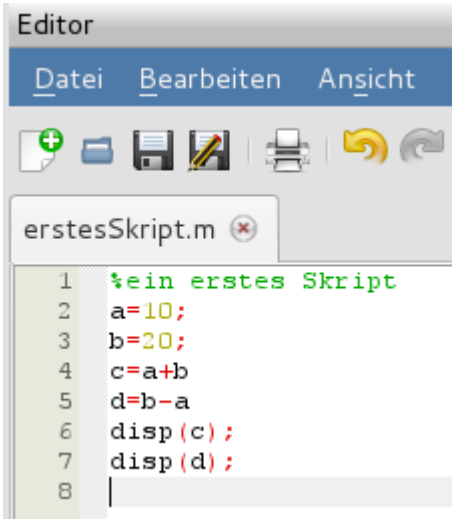

Abbildung 28: Skript zu Schritt 1 bis 3

```
>> erstesSkript
c =  30
d =  10
 30
 10
```

Abbildung 29: Befehlsfenster zu Schritt 1 bis 3

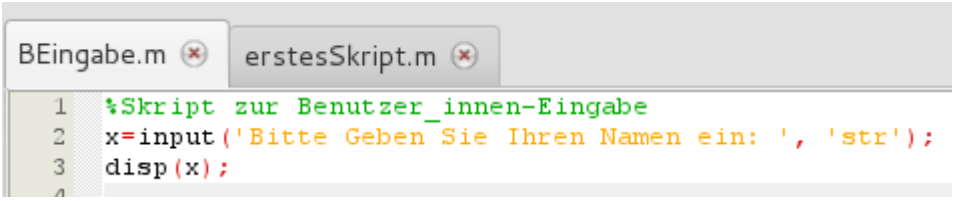

Abbildung 30: Skript zu Schritt 4

```
>> BEingabe
Bitte Geben Sie Ihren Namen ein: Mustermensch
Mustermensch
```

Abbildung 31: Befehlsfenster zu Schritt 4

Beispiel 11:
Es werden Funktionen definiert und abgerufen.

1. Anlegen einer Funktion:
   Über den Datei-Button im Editor kann „Neue Funktion“ gewählt. Es wird zur Eingabe des Funktionsnamens aufgefordert. Hier wurde der Name *ErsteFunktion* gewählt. Es wird automatisch der „Rahmen“ einer Funktion erstellt und die Funktion mit dem eingegeben Namen als m-Datei gespeichert. Alternativ kann das gesamte Gerüst auch direkt im Editor per Hand eingegeben werden.
   Die Variablen, die in der Funktion definiert werden, sind lokal und gehören nicht zum workspace von Octave. Von dort kann also nur per *function* auf diese Variablen zugegriffen werden.
   Allgemein ist eine Funktion wie folgt aufgebaut:

   ```
   function [output-Variable] = Name (input-Variablen)
       %Hier wird reingeschrieben, was die Funktion machen soll.
   endfunction
   ```

   in unserem Beispiel:

   ```
   function [retval] = ErsteFunktion (input1, input2)

   endfunction
   ```

   Im Befehlsfenster erscheint bei erfolgreichem Anlegen:

   ```
   >> edit ErsteFunktion.m
   ```

2. Funktion schreiben:
   Die Funktion soll zwei Variablen *out1* und *out2* ausgeben und eine Variable *x* bekommen. *x* soll erst quadriert und dieses Ergebnis dann wiederum mit *x* multipliziert werden.

   ```
   function [out1, out2] = ErsteFunktion (x)
       out1 = x^2;
       out2 = out1*x;
   endfunction
   ```

3. Funktion im Befehlsfenster abrufen:
   Nun wird die Funktion aufgerufen, Namen für die output-Variabeln festgelegt und ein Wert für *x* übergeben.

   ```
   >> [x1, x2] = ErsteFunktion(3)
   x1 =  9
   x2 =  27
   ```

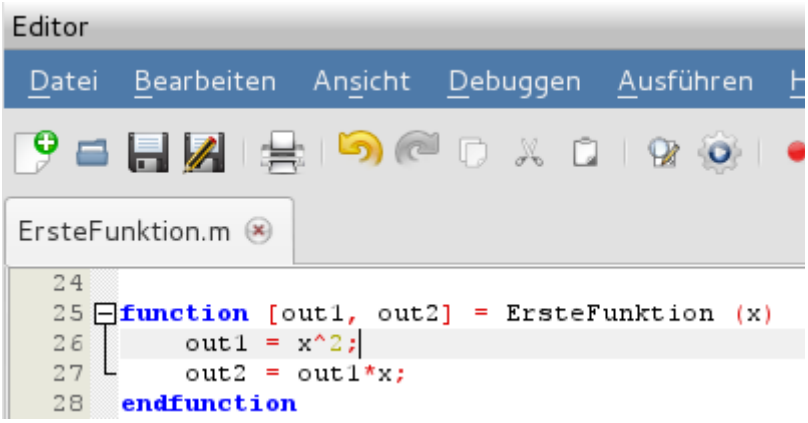

Abbildung 32: Skript zu Schritt 2

```
>> [x1, x2] = ErsteFunktion(3)
x1 =  9
x2 =  27
```

Abbildung 33: Befehlsfenster zu Schritt 2

Beispiel 12:
Es werden Kontrollstrukturen genutzt für bedingte und wiederholte Ausführung eines Programmabschnitts.

1. if-Bedingung:
   Hinter das Schlüsselwort *if* wird die zu erfüllende Bedingung geschrieben. Ist diese wahr, wird alles zwischen *if* und *end* ausgeführt. Alle verwendeten Variablenamen müssen definiert sein. In einem Skript namens *Kontrollstr* wird zunächst die if-Bedingung geschrieben. Dabei soll eine Zahl ausgegeben werden, wenn sie einen festgelegte Zahlenwert hat. *x* und *y* werden für unser Beispiel vor Beginn der Schleife auf einen willkürliche Wert gesetzt:

   ```
   %if zum Prüfen einer Bedingung
   x=10;
   y=20;

   if x==10
       disp(x);
   end

   if x<y
       disp(y);
   end
   ```

   Der Aufruf im Befehlsfenster erfolgt über das Skript:

   ```
   >> Kontrollstr
       10
       20
   ```

2. elseif-Bedingung:
   Zur Auswahl zwischen mehreren Bedingungen wird die elseif-Bedingung genutzt. Sie ist

ähnlich wie die if-Bedingung aufgebaut. Die Bedingungen stehen hinter *if* und *elseif*. *elseif* kann dabei mehrfach verwendet werden, um mehr als eine Alternative zu definieren. Trifft keine der vorher festgelegten Bedingungen zu, wird die unter *else* stehende Anweisung ausgeführt. Auch dieser Code wird dem Skript *Kontrollstr* hinzugefügt. *a* und *b* sind wieder willkürlich festgelegt:

```
a=15;
b=65;

if a==65
    disp('a=65');
elseif b==65
    disp('b=65');
else
    disp('nichts von beidem');
end
```

Aufruf erfolgt wieder im Befehlsfenster:

```
>> Kontrollstr
b=65
```

3. for-Schleife:
   Die for-Schleife wird verwendet, wenn die Anzahl der Durchläufe bekannt ist oder vorher ermittelt werden kann.In unserem Beispiel soll eine sogenannte Laufvariable *i* von 0 bis 5 laufen. Ist diese Bedingung erfüllt, soll *i* erst dargestellt und dann um 1 erhöht werden:

```
for i=0:5
    disp(i);
    i=i+1;
end
```

4. while-Schleife:
   Bei der while-Schleife wird die festgelegte Bedingung vor durchlaufen der Schleife geprüft. Die in der Schleife enthaltenen Anweisungen werden solange ausgeführt, bis die Bedingung nicht mehr erfüllt wird. In unseren Fall soll, solange *i* kleiner als 10 ist, die Variable *a* um 1 und anschließend *i* um 3 erhöht werden. Dann soll *a* ausgegeben werden. Vor der Schleife werden *i* und a definiert. *i=1* steht hier lediglich ohne Semikolon, damit der Beginn der while-Schleife besser im Befehlsfenster zu erkennen ist.

```
a=1;
i=1
while i<10
    a=a+i;
    i=i+3;
    disp(a);
end
```

```
%if zum Prüfen einer Bedingung
x=10;
y=20;

if x==10
    disp(x);
end

if x<y
    disp(y);
end

%elseif zur Auswahl zwischen mehreren Alternativen
a=15;
b=65;

if a==65
    disp('a=65');
elseif b==65
    disp('b=65');
else
    disp('nichts von beidem');
end
```

Abbildung 34: Skript zu Schritt 1 und 2

```
%for-Schleife
for i=0:5
  disp(i);
  i=i+1;
end

%while-Schleife
a=1;
i=1
while i<10
  a=a+i;
  i=i+3;
  disp(a);
end
```

Abbildung 36: Skript zu Schritt 3 und 4

```
>> Kontrollstr
 10
 20
b=65
0
 1
 2
 3
 4
 5
i =  1
 2
 6
 13
```

Abbildung 35: Befehlsfenster zu allen Schritten

Beispiel 13:
Es soll eine Excel-Datei importiert und angezeigt werden.

1. Excel-Datei importieren:
   Die Excel-Datei *OctaveTest2.xlsx* muss ebenfalls am selben Ort wie Octave gespeichert sein. Es wird ein Skript namens *E1* erzeugt und gespeichert. Darin wird die Funktion *xlsread* zum direkten Einlesen von Excel-Dateien verwendet. Es wird der Dateiname inklusive Format-Endung angegeben.

   x = xlsread('OctaveTest2.xlsx');

Dabei ist darauf zu achten, dass die Excel-Datei im Format *.xlsx* und nicht *.xls* gespeichert ist!

2. Zum Anzeigen des Inhaltes kann die bereits bekannte Funktion *disp* verwendet werden.

   disp(x);

   Die Datei kann auch direkt ohne *disp* angezeigt werden, indem man das Semikolon hinter der Zeile aus Schritt 1 entfallen lässt.

3. Im Befehlsfenster wird das Skript wie gewohnt unter seinem Speichernamen aufgerufen und gibt folgendes aus:

```
>> E1
    1    6    7
    2    7    9
    3    8   11
    4    9   13
    5   10   15
```

```
1  %Excel-Date importieren und anzeigen
2
3  x = xlsread('OctaveTest2.xlsx');
4  disp(x);
```

Abbildung 37: Skript zu Schritt 1 bis 3

```
>> E1
    1    6    7
    2    7    9
    3    8   11
    4    9   13
    5   10   15
```

Abbildung 38: Befehlsfenster zu Schritt 3

Beispiel 14:
Es soll eine Text-Datei namens *OctaveTest1.txt* importiert werden. Dafür wird ein kurzes Skript namens *V1* geschrieben.

1. Die Text-Datei *OctaveTest1.txt* muss zunächst geöffnet (*fopen*) und geschlossen (*fclose*) werden:

   a=fopen('OctaveTest1.txt', 'rt');

   fclose(a);

Es wird der Dateiname inklusive Format-Endung und der Befehl *'rt'* für „read text“ (alternativ gibt es auch *bt* für binäre Dateien) angegeben.

2. Zwischen diesen beiden Zeilen können nun Befehle zum verarbeiten angegeben werden. Hier soll sie nur importiert werden:

   A=fscanf(a, '%c', [1, Inf]);

   Zunächst wird das auszugebende Element genannt, dann welches Format gelesen werden soll (hier *c* für „Character“) und dann ggf. welcher Bereich gelesen werden soll (hier von der 1. Stelle bis zum „Ende“, was dem Befehl *Inf* entspricht).

3. Nach schließen der Datei, soll ihr Inhalt angezeigt werden:

   y=A(1:end)

   Alternativ kann dafür auch die Funktion *disp(A)* genutzt werden.

4. Im Befehlsfenster wird das Skript wie gewohnt mit dem entsprechenden Namen aufgerufen und es wird der Inhalt der Datei (hier der Satz *Es hat geklappt!*) angezeigt:

   >> V1
   y = Es hat geklappt!

```
1  %Skript zum Importieren einer Text-Datei
2
3  a=fopen('OctaveTest1.txt', 'rt');
4  A=fscanf(a, '%c', [1, Inf]);
5  fclose(a);
6  y=A(1:end)
```

Abbildung 39: Gesamtes Skript

```
>> V1
y = Es hat geklappt!
```

Abbildung 40: Befehlsfenster

Beispiel 15:
Programmteile sollen als Excel- und als Text-Datei gespeichert werden. Dazu wird ein kleines Skript namens *sp1* geschrieben.

1. Der Programmteil *B=rand(100,100)* soll als Excel-Datei gespeichert werden. Die *rand* Funktion erstellt eine 100x100-Matrix gefüllt mit Zufallszahlen zwischen 0 und 1. Dafür kann unter Windows die Funktion *xlswrite* genutzt werden. Allgemein kann allerdings der Programmteil auch im csv-Format gespeichert, was sich anschließend in eine Excel-Datei umwandeln lässt. Zum Erstellen der Datei wird deswegen hier die Funktion *csvwrite* genutzt:

   B=rand(100,100);
   csvwrite('Zufall.csv', B)

Zuerst wird der Name und das Format der späteren Datei festgelegt. Dann wird der zu speichernde Programmabschnitt benannt. Das Skript muss nun lediglich noch im Befehlsfenster aufgerufen werden. Octave speichert nun eine Datei *Zufall.csv* im aktuellen Verzeichnis ab.

2. Der String *Mal gucken, ob es klappt.* soll nun als Textdatei gespeichert werden. Dafür öffnet (*fopen*) und schließt (*fclose*) man eine Datei. Dazwischen wird der Inhalt mit *fputs* geschrieben:

```
M = 'Mal gucken, ob es klappt.';
d = fopen ('Test.txt' , 'w');
fputs(d, M);
fclose (d);
```

Bei *fopen* wird zuerst der spätere Dateiname festgelegt und dass in diesen etwas geschrieben wird (*'w'* für „write“). Mit *fputs* wird nun dieser Befehl angesprochen und der zu speichernde String benannt. Abschließend muss die nun erzeugte Datei noch geschlossen werden. Das Skript wird im Befehlsfenster aufgerufen und ausgeführt. Octave speichert nun eine Datei *Text.txt* im aktuellen Verzeichnis ab.

```
1  %Speichern/Exportieren von Dateien/Programmteilen
2
3  %------------Speichern als Excel-Datei:------------
4
5  %A=rand(100,100);
6  %xlswrite('Zufall.xlsx', A)
7
8  %------> funktioniert nur unter Windwos
9
10 %csv-Datei kann anschließend in Excel-Datei umgewandelt werden
11 B=rand(100,100);
12 csvwrite('Zufall.csv', B)
13
14
15 %------------Speichern als Text-Datei:------------
16 M = 'Mal gucken, ob es klappt.';
17 d = fopen ('Test.txt' , 'w');
18 fputs (d, M);
19 fclose (d);
```

Abbildung 41: Skript inklusive Alternative unter Windows

```
>> sp1
>>
```

Abbildung 42: Befehlsfenster

Beispiel 16:
Über ein Cluster soll ein Job (z.B. eine aufwendige Simulation) parallel auf mehreren Kernen durchgeführt werden. Dazu wird das sogenannte „Message Passing Interface“ (kurz MPI) verwendet. Dies ist ein Standard, welcher den Informationsaustausch bei paralleler Berechnung auf verschiedenen Computersystemen beschreibt. Zusätzlich wird eine Software zur Planung dieser Jobs benötigt, in diesem Fall das „Portabel Batch System“ (kurz PBS).
Der gesamte Prozess läuft grundsätzlich wie folgt ab: Mit dem Befehl *qsub* wird die cmd-Datei, die die Anweisungen für das Batchsystem enthält, aufgerufen. Dann wird zuerst geprüft, ob die in der Datei festgelegte Anzahl an Knoten (node) und Kernen (core) zur Verfügung steht. Anschließend wird festgelegt, welche der verfügbaren Knoten und Kerne zur Berechnung des Jobs genutzt werden sollen. Diese werden reserviert und in eine Datei (*PBS_NODEFILE*) geschrieben. Am Ende der cmd-Datei steht die Funktion *mpirun*. Mit diesem wird das angegebene Programm (hier Octave) und die darin beschriebene Berechnung (hier *BeispielAufruf01*) aufgerufen und auf den in der Datei *PBS_NODEFILE* festgelegten Kernen durchgeführt. Am Ende stehen in einer output-Datei die Ergebnisse des Jobs.
Wie dieser Ablauf anhand eines Beispiels in einen konkreten Quellcode übersetzt werden kann, wird in der folgenden Anleitung nun genauer erklärt:

1. Zunächst kann geprüft werden, ob das MPI-Package in Octave geladen ist. Dazu wird der Befehl *pkg list* verwendet. Dieser kann einfach direkt in das Befehlsfenster eingegeben werden:

   ```
   >> pkg list
   ```

   Es erscheint eine Auflistung aller geladenen Packages. Ist das MPI-Package geladen, so erscheint es mit einem Stern markiert und weiteren Informationen wie der geladenen Version in einer Liste im Befehlsfenster:

   ```
   mpi *|    1.2.0 | /usr/share/octave/packages/mpi-1.2.0
   ```

   Sollte das Package nicht geladen sein, kann dieses mit dem Befehl *pkg load mpi* nachgeholt werden.
   Dieser gesamte Schritt ist optional und wird in diesem Beispiel dadurch umgangen, dass das Laden des MPI-Packages direkt im Quellcode verankert wird.

2. Es wird die cmd-Datei erstellt, mit welcher der durchzuführende Job gesteuert („geplant“) wird. Diese Datei wird in einem Editor, z.B. Geany, geschrieben. Die hier verwendete cmd-Datei trägt den Namen *MPI_DemoAufruf.cmd*. Aufgrund ihrer Länge wurde die Zeile 25 aufgeteilt. Der unten angefügte Teil ab *octave –no-gui* steht im Quellcode ebenfalls in Zeile 25 (hinter *$CORES*).

MPI_DemoAufruf.cmd

```
#PBS -o output.dat
#PBS -j oe
#PBS -q scc-queue1
#PBS -N DemoHallo
#PBS -l nodes=2:ppn=4,walltime=01:00:00

#ulimit -l
cat $PBS_NODEFILE
. /etc/profile.d/modules.sh

#module add OpenMPI/1.8.8-GNU-4.9.3-2.25

export CORES_PER_NODE=4
export CORES=8

export PARNODES=`wc -l $PBS_NODEFILE |gawk '{print $1}'`
export UNIQNODES=`cat $PBS_NODEFILE |gawk '{print $1}' | uniq | wc -l`

cd $PBS_O_WORKDIR

export CPPFLAGS="-I/usr/include/eigen3 $(mpicxx -showme:compile)"
export LD_PRELOAD=/usr/lib/openmpi/lib/libmpi.so


mpirun -v -x CPPFLAGS -x LD_PRELOAD -machinefile $PBS_NODEFILE -np $CORES
```

Abbildung 43: Code der Datei MPI_DemoAufruf.cmd

```
octave --no-gui -q --no-history --eval 'pkg load mpi; BeispielAufruf01();'
```

Abbildung 44: Restliche Zeile 25 der Datei MPI_DemoAufruf.cmd

Bei den Zeilen, die mit *#PBS* beginnen, handelt es sich um Anweisungen an das verwendete Batchsystem. Die einzelnen Codezeilen haben dabei die folgende Funktion:

Zeile Nr.	**Code**	**Funktion**
1	*#PBS -o output.dat*	Erzeugt eine Datei mit dem Namen *output* und dem Dateiformat *.dat*. In diese Datei werden die Ergebnisse des Jobs geschrieben.
2	*#PBS -j oe*	Alle Fehler werden mit in die output-Datei geschrieben. Optional können die Fehler und die Ergebnisse auch in unterschiedlichen Dateien gespeichert werden.
3	*#PBS -q scc-queuel*	Queue (Reihe), in die der Job eingreiht werden soll (hier die Reihe mit dem Namen *scc-queuel*)
4	*#PBS -N DemoHallo*	Unter diesem Namen wird der Job während der Durchführung angezeigt (hier *DemoHallo*)
5	*#PBS -l nodes=2:ppn=4, walltime=01:00:00*	nodes=2 bedeutet, dass für die Berechnung dieses Jobs zwei Knoten reserviert werden. ppn=4 (processor per node) legt die Anzahl der reservierten Prozessorkerne pro Knoten fest (hier 4). walltime=01:00:00 legt die Zeit fest, nach der das Programm spätestens abbricht, sollte bspw. ein Fehler auftreten. Das Format ist dabei: hh:mm:ss (Stunde:Minute:Sekunde).

Tabelle 1: Erklärung des Codes der cmd-Datei Zeile 1-5

In Zeile 7-22 werden für das System spezifische Einstellungen festgelegt. Diese werden in der folgenden Tabelle genauer erklärt:

Zeile Nr.	**Code**	**Funktion**
7	*#ulimit -l*	*ulimit* begrenzt die durch *-l* genauer festgelegten verwendeten Programmressourcen.
8	*cat $PBS_NODEFILE*	*cat:* Der Inhalt der angegeben Datei wird als Argument übergeben, gelesen und anschließend in einem zusammenhängenden Datenstrom ausgegeben. *$*-Zeichen: kennzeichnet Umgebungsvariablen *PBS_NODEFILE:* Enthält eine Liste der für den durchzuführenden Job reservierten Kerne.
9	*. /etc/profile.d/modules.sh*	Es werden Umgebungsmodule verwendet.
11	*#module add OpenMPI/1.8.8-GNU-4.9.3-2.25*	*module add* fügt das angegebene Modul der Umgebung hinzu (hier: *OpenMPI* in der hinter dem Backslash angezeigten Version)
13 *14*	*export CORES_PER_NODE=4* *export CORES=8*	In diesen beiden Zeilen werden, wie in Zeile 5, die verwendeten Kerne pro Knoten und die Gesamtzahl der Kerne definiert. Diese Zeilen müssen bei Änderungen in Zeile 5 entsprechend angepasst werden!
16	*export PARNODES=* *`wc -l $PBS_NODEFILE* *|gawk '{print $1}'`*	Mit *PARNODES* wird die Anzahl der verwendeten Prozessorkerne festgelegt bzw. ausgelesen. Hier wird dabei auf das bereits zuvor verwendete *PBS_NODEFILE* verwiesen.
17	*export UNIQNODES=* *`cat $PBS_NODEFILE* *|gawk '{print $1}' | uniq | wc -l`*	*UNIQNODES s*orgt dafür, dass jeder Kern nur einmal verwendet wird.
19	*cd $PBS_O_WORKDIR*	Es wird in das Verzeichnis, in dem der später aufgerufene *qsub*-Befehl laufen soll, gewechselt.
21	*export CPPFLAGS=* *"-I/usr/include/eigen3* *$(mpicxx -showme:compile)"*	*Flag*: Wird zur Kennzeichnung bestimmter Zustände verwendet, eine Art „Marker“, *CPPFLAGS:* Flags für den Präprozessor (Schritt vor dem eigentlichen Kompilieren), *mpicxx:* verweist auf einen in C++ geschriebenes MPI-Programm, *showme:compile:* gibt Flags aus, die der C++-Compiler bereit gestellt hätte
22	*export LD_PRELOAD=* */usr/lib/openmpi/lib/libmpi.so*	*LD_PRELOAD* zwingt die danach angegebene Programmbibliothek vor der Durchführung des Programms geladen zu werden.

Tabelle 2: Erklärung des Codes der cmd-Datei Zeile 7-22

Von besonderen Interesse ist die letzte Zeile (Nr. 25) der cmd-Datei. Hier wird das MPI genutzt. Hier werden alle für die Durchführung des Jobs relevanten Informationen übermittelt. Dabei haben die einzelnen Befehle die folgenden Funktionen:

Code	**Funktion**
mpirun	Starten des Jobs auf mehreren Prozessorkernen
-v	Damit werden alle Schritte und Kommandos, die der Compiler durchführt angezeigt (ist optional).
-x CPPFLAGS	*-x* wird verwendet um (mehrere) Umgebungsvariablen zur Parallelrechnung zu exportieren. In diesem Falls werden die Flags des Präprozessors weitergegeben.
-x LD_PRELOAD	Hier werden entsprechend die vor der eigentlichen Durchführung des Programms zu ladenden Programmbibliotheken weitergegeben.
-machinefile $PBS_NODEFILE	*machinefile* entnimmt der Datei *PBS_NODEFILE* die Liste der möglichen Kerne, auf denen der Job laufen kann.
-np $CORES	*np* legt die Anzahl der Kerne fest. In diesem Fall wird diese zuvor in Zeile 14 definiert.
octave	Octave wird aufgerufen.
--no-gui	Es wird auf den aufgerufenen Prozessorkernen keine Benutzeroberfläche (GUI = Graphical User Interface) von Octave geöffnet.
-q	Es wird die Ausgabe von Informationen während der Durchführung des Jobs unterdrückt (ist optional).
--no-history	Es wird keine Liste der verwendeten Befehle angelegt.
--eval 'pkg load mpi; BeispielAufruf01();'	Mit dieser Anweisung wird in Octave das MPI_Package geladen und die Funktion *BeispielAufruf01()* aufgerufen. Soll eine andere Funktion, Datei oder ein anderes Skript durchgeführt werden, muss an dieser Stelle der entsprechende Name angegeben werden.

Tabelle 3: Erklärung des Codes der cmd-Datei Zeile 25

3. Nun kann sich dem eigentlichen Octave-Skript zugewendet werden. In unserem Beispiel soll lediglich der folgende Text ausgegeben werden:

Prozessor: x von y sagt Hallo!

Für diesen Job, ist die Leistung mehrerer Kerne eigentlich nicht nötig. Es handelt sich dabei lediglich um ein Beispiel zur Erklärung des Codes.
x und *y* sind hier Zahlen. Es wird jedem Kern bei der Durchführung des Jobs eine Zahl (hier der Rang) zugewiesen. Diese Zahl lassen wir uns gemeinsam mit der Gesamtzahl an Kernen in der output-Datei ausgeben. Der gesamte Ocatve-Code ist in der folgenden Abbildung zu sehen:

BeispielAufruf01.m

```
1  function BeispielAufruf01()
2
3  MPI_Init();                          % Beginn der MPI-Umgebung
4  CW = MPI_Comm_Load("NEWORLD");
5  my_rank = MPI_Comm_rank (CW);
6  p = MPI_Comm_size (CW);
7
8  pause(30);
9  disp(['Prozessor: ' num2str(my_rank) ' von ' num2str(p) ' sagt Hallo!']);
10
11 MPI_Finalize();                      % Ende der MPI-Umgebung
12
13 endfunction
```

Abbildung 45: Skript namens *BeispielAufruf01*

In unserem Beispiel wird eine Funktion namens *BeispielAufruf01* geschrieben. Weitere Informationen zu Funktionen sind in Beispiel 11 zu finden. Die notwendige MPI-Umgebung zur parallelen Durchführung muss immer gestartet und beendet werden. Zwischen diesen beiden Befehlen kann der durchzuführende Job definiert werden. In unserem Beispiel wird der Rang des Prozessorkerns abgefragt und ausgegeben. Die Pause dient lediglich als Platzhalter für aufwendigere Jobs. In der folgenden Tabelle wird der Code Zeile für Zeile erklärt:

Zeile Nr.	**Code**	**Funktion**
3	*MPI_Init()*	Startet die MPI-Umgebung
4	*MPI_Comm_Load("NEWORLD")*	Es wird dem sogenannten Communicator ein Objekt namens *NEWORLD* übergeben.
5	*MPI_Comm_rank (CW)*	Gibt den sogenannten Rang des gerade verwendeten Prozessorkerns an den Communicator (hier CW) weiter.
6	*MPI_Comm_size (CW)*	Gibt Größe des Communicators wieder.
8	*pause()*	Legt bei der Ausführung des Jobs eine Pause von x Sekunden ein. Diese dient hier lediglich als Platzhalter für aufwendigere Berechnungen.
9	*disp()*	Bei der Durchführung des Programms wird der in Klammern stehende Text ausgegeben.
9	*num2str()*	Wandelt eine Zahl in einen String um. Dies ist hier nötig, da *disp()* nur Strings und keine Zahlen ausgeben kann.
11	*MPI_Finalize()*	Beendet die MPI-Umgebung

Tabelle 4: Erklärung des Codes der Datei *BeispielAufruf01*

4. Nun ist der gesamte Quellcode geschrieben und der Job kann durchführt werden. Dazu wird ein Terminal (bspw. das *MATE*-Terminal) aufgerufen. Es muss zunächst in das Verzeichnis gewechselt werden, in welchem sich die cmd- und die Octave-Datei befinden. Dazu wird der Befehl *cd Verzeichnispfad* genutzt. Anstelle des Wortes *Verzeichnispfad* muss natürlich der Pfad der verwendeten Dateien eingefügt werden.
   Nun kann der Job mit dem Befehl *qsub MPI_DemoAufruf.cmd* durchgeführt werden. Es wird im Terminal also nur die cmd-Datei aufgerufen. Der Aufruf der Octave-Datei ist schließlich schon in der cmd-Datei enthalten. Nun werden die zu Beginn erklärten Programmschritte abgearbeitet.
   In der folgenden Abbildung sind der Verzeichniswechsel und das Starten des Jobs zu sehen:

```
ll376706@fb11-nx-main:~/octave_beispiele_cluster$ cd /home/st/l/ll376706/octave_
beispiele_cluster/DemoProgramm
ll376706@fb11-nx-main:~/octave_beispiele_cluster/DemoProgramm$ qsub MPI_DemoAufr
uf.cmd
10789.fb11-nx-main.fh-muenster.de
```

Abbildung 46: Der Wechsel in das richtige Verzeichnis und Starten des Jobs über das Terminal

   Der Job hat bei der Durchführung die Kennnummer 10789 erhalten.
   Mit dem Befehl *pbstop* kann eine Liste aller Knoten und Kerne aufgerufen werden. In dieser ist zu sehen, auf welchen Kernen der Job durchgeführt wird. Dies ist in der folgenden Abbildung zu sehen:

```
     Job#  Username  Queue    Jobname    Nodes   S  Elapsed/Requested
l = 10789 ll376706  scc-queu DemoHallo      2   R       --/   00:10
```

Abbildung 47: Jobdetails, die durch den Befehl *pbstop* angezeigt werden.

   In unserem Beispiel wurde der Job also auf den zwei Knoten mit den Nummern 09 und 10 und dort jeweils auf den ersten vier Kernen durchgeführt:

```
                ------------------------------------------------------------
fb11-nx-node09  l l l l . . . . . .   . . . . . . . . . .   . . . .
                ------------------------------------------------------------

                1 2 3 4 5 6 7 8 9 0   1 2 3 4 5 6 7 8 9 0   1 2 3 4 5 6 7 8 9 0
                ------------------------------------------------------------
fb11-nx-node10  l l l l . . . . . .   . . . . . . . . . .   . . . .
                ------------------------------------------------------------
```

Abbildung 48: Übersicht der Verteilung des Jobs über die Kerne mittels des Befehls *pbstop*.

   Hier ist auch der für den Job vergebene Name *DemoHallo* wiederzufinden. Diese Ansicht kann mit der Tastenkombination *Strg* und *C* verlassen werden.

5. Die Ergebnisse und die Fehler des Jobs werden in die output-Datei geschrieben. Diese befindet sich im selben Verzeichnis wie die cmd- und die Octave-Datei. Für diesen Job wird in der folgenden Abbildung nur das eigentliche Ergebnis dargestellt:

```
185   Prozessor: 1 von 8 sagt Hallo!
186   Prozessor: 2 von 8 sagt Hallo!
187   Prozessor: 5 von 8 sagt Hallo!
188   Prozessor: 7 von 8 sagt Hallo!
189   Prozessor: 4 von 8 sagt Hallo!
190   Prozessor: 6 von 8 sagt Hallo!
191   Prozessor: 0 von 8 sagt Hallo!
192   Prozessor: 3 von 8 sagt Hallo!
```

Abbildung 49: Ergebnis des Jobs mit der Octave-Datei *BeispielAufruf02*

Da im Quellcode keine Reihenfolge der Prozessorkerne festgelegt wurde, sind sie auch in der output-Datei ungeordnet. Die Nummerierung des Rangs startet bei 0 und nicht bei 1.

6. Als Variante des Octave-Skriptes wird hier *BeispielAufruf02* noch zusätzlich aufgeführt. Anders als im ersten Beispiel sollen hier die Prozessorkerne bei der Ausgabe nach ihrem Rang geordnet werden. Dazu wird ihre Rangnummer mit einem Faktor multipliziert. Das Ergebnis wird als Pause bei der Durchführung des Jobs eingefügt (Codezeile 8).
   Nun muss lediglich in der cmd-Datei in Zeile 25 der Name des aufgerufenen Skriptes in *BeispielAufruf02()* geändert werden. Ansonsten bleiben Code und Durchführung gleich.

BeispielAufruf02.m

```
1   function BeispielAufruf02()
2
3   MPI_Init();                     % Begin der MPI-Umgebung
4   CW = MPI_Comm_Load("NEWORLD");
5   my_rank = MPI_Comm_rank (CW);
6   p = MPI_Comm_size (CW);
7
8   pause(0.2*my_rank);
9   disp(['Prozessor: ' num2str(my_rank) ' von ' num2str(p) ' sagt Hallo!']);
10  pause(20);
11
12  MPI_Finalize();                 % Ende der MPI-Umgebung
13
14  endfunction
```

Abbildung 50: Skript namens *BeispielAufruf02*

Das Ergebnis dieses Jobs zeigt die folgende Abbildung:

```
185   Prozessor: 0 von 8 sagt Hallo!
186   Prozessor: 1 von 8 sagt Hallo!
187   Prozessor: 2 von 8 sagt Hallo!
188   Prozessor: 3 von 8 sagt Hallo!
189   Prozessor: 4 von 8 sagt Hallo!
190   Prozessor: 5 von 8 sagt Hallo!
191   Prozessor: 6 von 8 sagt Hallo!
192   Prozessor: 7 von 8 sagt Hallo!
```

Abbildung 51: Ergebnis des Jobs mit der Octave-Datei *BeispielAufruf02*

Die Sortierung der Prozessorkerne hat funktioniert.

Beispiel 17:
Es soll mit Octave zunächst eine E-Mail mit dem Ergebnis einer einfachen Berechnung an eine einzugebende E-Mail-Adresse verschickt werden. Es soll zuvor gefragt werden, ob diese losgeschickt werden soll.

1. Es wird eine Funktion mit dem Namen *MailVar* erstellt, die das Ergebnis der Berechnung *y=x+7* als Mail verschicken soll (Anlegen einer Funktion siehe Beispiel 11). Der Wert von *x* soll gewählt werden können. Der grobe Aufbau der Funktion zur Berechnung sieht wie folgt:

   function MailVar(x)
   y=x+7;
   endfunction

   Im Befehlsfenster kann nun mit z.B. *MailVar(3)* die Funktion aufgerufen werden. Bei diesem Beispiel ergibt sich *y=10* als Ergebnis im Befehlsfenster.

1. Nun wird zunächst der Code zum Verschicken der Mail geschrieben. Dies läuft über die system-Funktion. Mit dieser Funktion lassen sich Subprozesse starten und kontrollieren. Soll also z.B. ein anderes Programm mit Octave gestartet und das Ergebnis wieder an Octave zurückgegeben werden, kann dafür diese Funktion genutzt werden. In unserem Fall wird das MATE-Terminal im Hintergrund gestartet und zum Verschicken der Mail genutzt. Es könnte also auch der Inhalt von *system* direkt in das Terminal eingegeben und ausgeführt werden. Der hier verwendete Code sieht wie folgt aus:

   system(["echo 'Das Ergebnis der Berechnung ist: '" ys " |mail -s 'Ergebnis einer Berechnung' " mail]);

   Die [ ]-Klammern zusammen mit den „“-Anführungszeichen bedeuten, dass es sich hier um einen String handelt (Syntax der system-Funktion).
   Die folgenden Erklärungen sind nun also keine eigentlichen Octave-Befehle, sondern funktionieren so im Terminal:
   Mit dem *echo*-Befehl wird der Inhalt oder der angegebene String wiedergegeben, in diesem Fall *Das Ergebnis der Berechnung ist:* . Wichtig ist hierbei die Kennzeichnung des Strings mit einem Apostroph an seinem Anfang und an seinem Ende. Das *ys* ist hier der Name der Variable, die das Ergebnis der Berechnung enthält. Mit dem Code |*mail* wird nun gesagt, dass eine E-Mail verschickt werden soll. Mit -s *'Ergebnis einer Berechnung'* wird der E-Mail eine Betreffzeile hinzugefügt, die die Worte *Ergebnis einer Berechnung* enthält. Auch hier muss der String wieder durch Apostrophe eingeleitet und beendet werden. Abschließend wird die E-Mail-Adresse als Variable *mail* übergeben. Diese Adresse soll ebenfalls auf Nachfrage im Befehlsfenster eingegeben werden. Handelt es sich um mehrere E-Mail-Adressen, können sie durch Kommata getrennt voneinander angegeben werden.

2. Es wird die Frage, ob die Mail verschickt werden soll oder nicht programmiert. Hierbei handelt es sich um eine einfache Ja-Nein-Frage. Für solche Fälle hält Octave eine eingebaute Funktion namens *yes_or_no(' ')* bereit. In die Klammern wird die Frage geschrieben, die im Befehlsfenster ausgegeben werden soll. In unserem Beispiel ist das die folgende Frage: *Wollen Sie das Ergebnis der Berechnung als Mail verschicken?* . Das Leerzeichen am Ende der Frage dient nur der Übersichtlichkeit im Befehlsfenster, damit das von Octave automatisch hinzugefügte *(yes or no)* nicht direkt nach dem Fragezeichen steht.

Dieser Codeabschnitt sieht nun wie folgt aus:

```
ant=yes_or_no('Wollen Sie das Ergebnis der Berechnung als Mail verschicken? ');
```

Wird im Befehlsfenster *yes* eingegeben, wird *ant* auf 1 gesetzt, wird *no* eingegeben entsprechend auf 0. Daraus lässt sich die Bedingung für die if-Kontrollstrutkur erzeugen (mehr zu Kontrollstrukturen in Beispiel 12). Wird *yes* eingegeben, ist also *ant=1*, so soll eine E-Mail verschickt werden. Wird *no* eingegeben, soll diese entsprechend nicht verschickt werden.

```
if ant==1
    system([ "echo 'Das Ergebnis der Berechnung ist: '" ys " |mail -s 'Ergebnis einer
    Berechnung' " mail ]);
    disp(['Es wurde eine Mail an ' mail ' verschickt.']);
else
    disp('Es wurde keine Mail verschickt.');
end
```

3. Es muss noch die Eingabeaufforderung für die E-Mail-Adresse erfolgen. Dazu kann die input-Funktion verwendet werden.

   ```
   mail=input('Geben Sie Ihre E-Mail-Adresse jetzt ein. Bei mehreren Adressen trennen Sie
   diese durch Kommata: ','s');
   ```

   Es erfolgt die Eingabeaufforderung im Befehlsfenster mit dem Hinweis *Geben Sie Ihre E-Mail-Adresse jetzt ein. Bei mehreren Adressen trennen Sie diese durch Kommata:* . Die anschließend eingegebene(n) Adresse(n) werden als String ('s') in der Variable *mail* gespeichert.

4. Abschließend muss lediglich noch die folgende Zeile dem Code hinzugefügt werden:

   ```
   ys=num2str(y);
   ```

   Die system-Funktion kann nur Strings verarbeiten. Daher muss das Ergebnis *y*, das als Zahl vorliegt, noch in einen String umgewandelt werden. Dies führt die Funktion *num2str()* durch.

5. Der komplette Code wird als Skript unter dem Namen *MailVar.m* gespeichert und ist im folgenden Bild zu sehen:

MailVar.m

```
1  function MailVar(x)
2
3  y=x+7;
4  ys=num2str(y);
5
6  ant=yes_or_no('Wollen Sie das Ergebnis der Berechnung als Mail verschicken? ');
7
8  if ant==1
9    mail=input('Geben Sie Ihre E-Mail-Adresse jetzt ein. Bei mehreren Adressen
10   trennen Sie diese durch Kommata: ','s');
11   system([ "echo 'Das Ergebnis der Berechnung ist: '" ys
12   " |mail -s 'Ergebnis einer Berechnung' " mail ]);
13   disp(['Es wurde eine Mail an ' mail ' verschickt.']);
14 else
15   disp('Es wurde keine Mail verschickt.');
16 end
17
18 %system("cat /home/st/1/11376706/octave_beispiele_cluster/InhaltMail.txt |
19 %mail -s 'Eine Datei mit Octave verschicken' 11376706@fh-muenster.de");
20
21 endfunction
```

Abbildung 52: Skript zu Schritt 1 bis 4

Achtung: Zeile 9 und 10 müssen im Code unbedingt in eine gemeinsame Zeile geschrieben werden! Sie sind hier nur zu Übersichtszwecken getrennt worden. Das gleiche gilt für Zeile 11 und 12.

6. Der Aufruf samt Aus- und Eingabe im Befehlsfenster sieht beispielhaft wie folgt aus:

```
>> MailVar(10)
Wollen Sie das Ergebnis der Berechnung als Mail verschicken? (yes or no) no
Es wurde keine Mail verschickt.
>> MailVar(11)
Wollen Sie das Ergebnis der Berechnung als Mail verschicken? (yes or no) yes
Geben Sie Ihre E-Mail-Adresse jetzt ein. Bei mehreren Adressen trennen Sie diese
 durch Kommata: 11376706@fh-muenster.de
Es wurde eine Mail an 11376706@fh-muenster.de verschickt.
```

Abbildung 53: Befehlsfenster zu Schritt 6

7. Im Fall des zweiten Aufrufs *MailVar(11)* ist *x=11* und *y* muss entsprechend 18 sein. Der Inhalt der empfangen Mail sieht damit wie folgt aus:

```
Das Ergebnis der Berechnung ist: 18
```

Abbildung 54: Inhalt der E-Mail

8. Soll der Inhalt einer Datei als Anhang verschickt werden, kann dies mittels der Codezeilen 18 und 19 erfolgen. Diese müssen dann jedoch in einer gemeinsamen Zeile ohne die beiden Prozentzeichen geschrieben werden. Anstelle des echo-Befehls in Zeile 11 muss der cat-Befehl verwendet werden. Dieser liest den Inhalt der danach angegebenen Datei aus. Damit diese gefunden wird, muss nach diesem Befehl der Speicherpfad der Datei angegeben werden. In unserem Beispiel handelt es sich um eine einfache Text-Datei mit belanglosem

Inhalt, die an die direkt angegebene E-Mail-Adresse gesendet wird. Die erhaltene E-Mail sieht wie folgt aus:

```
Ich bin ein Text, der einfach nur zum Fuellen der Mail dient.
Ich bin ein Absatz mit sinnlosem Inhalt. Ich bin wieder ein Text, der einfach nur
zum Fuellen dient.
Ich bin eine kleine Rechnung: zahl = 5 + 4 = 9
```

Abbildung 55: Inhalt der durch eine externe Datei gefüllten E-Mail