Eine kurze Einführung in MATLAB
Sommersemester 2015

PD Dr. Thorsten Hüls
Lukasz Targas

10.4.2015

1 MATLAB

MATLAB (**MAT**rix **LAB**oratory) ist ein kommerzielles interaktives Programm zur Durchführung numerischer Berechnungen. Die Software wird von der Firma MATHWORKS entwickelt. MATLAB wird mit dem Terminal-Befehl: `matlab` gestartet.

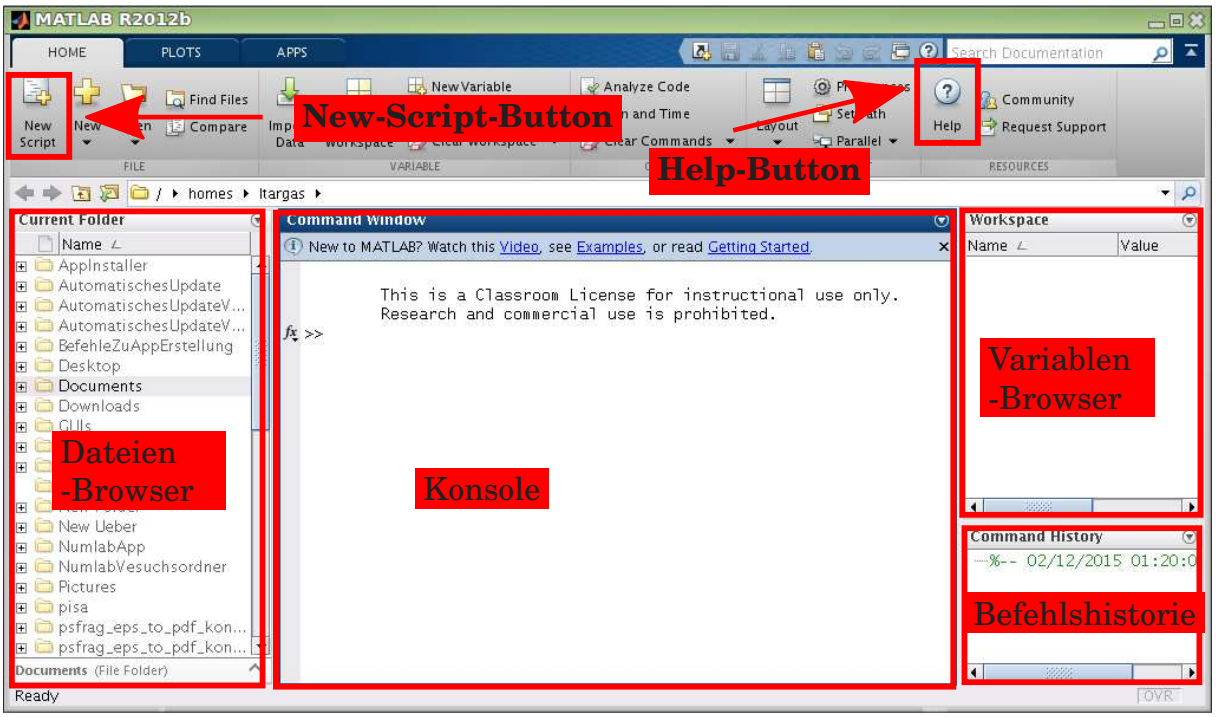


Abbildung: MATLAB-Oberfläche

Die grafische Oberfläche von MATLAB beinhaltet standardmäßig vier Fenster:

- Dateien-Browser: Erstellen, Ausführen, Öffnen, Löschen, Umbenennen, Abrufen der Eigenschaften von Dateien und Ordnern.
- Konsole: Textbasiertes Ein- und Ausgabefenster (siehe Abschnitt 1.1).
- Variablen-Browser: Verwalten von Variablen.
- Befehlshistorie: Dokumentiert alle ausgeführten Eingaben in der Konsole.

1.1 Konsole

Die *Konsole* ist das wichtigste Element. Dort können MATLAB-Befehle wie beispielsweise `2 + 2` ausgeführt werden.

```
>> 2 + 2
ans =  4
```

Falls erwünscht, kann mittels `;` die Ausgabe unterdrückt werden, z. B.

```
>> 2 + 2;
```

Wir können hier Variablen definieren:

```
>> x = 7
x =  7
>> y = 5
y =  5
```

Diese erscheinen dann im Variablen-Browser und können jeder Zeit abgerufen oder verändert werden, z. B.

```
>> x + y
ans =  12
>> y = 3
y =  3
>> x + y
ans =  10
```

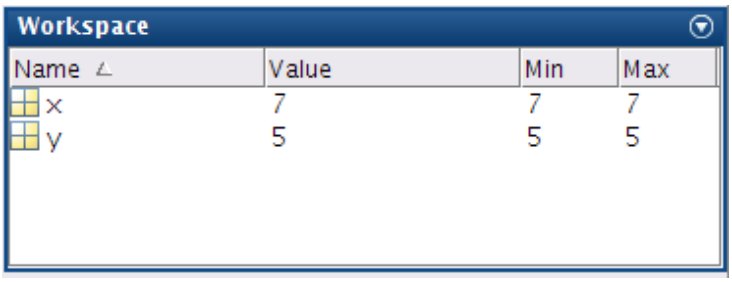


Abbildung: Variablen-Browser

1.2 Editor

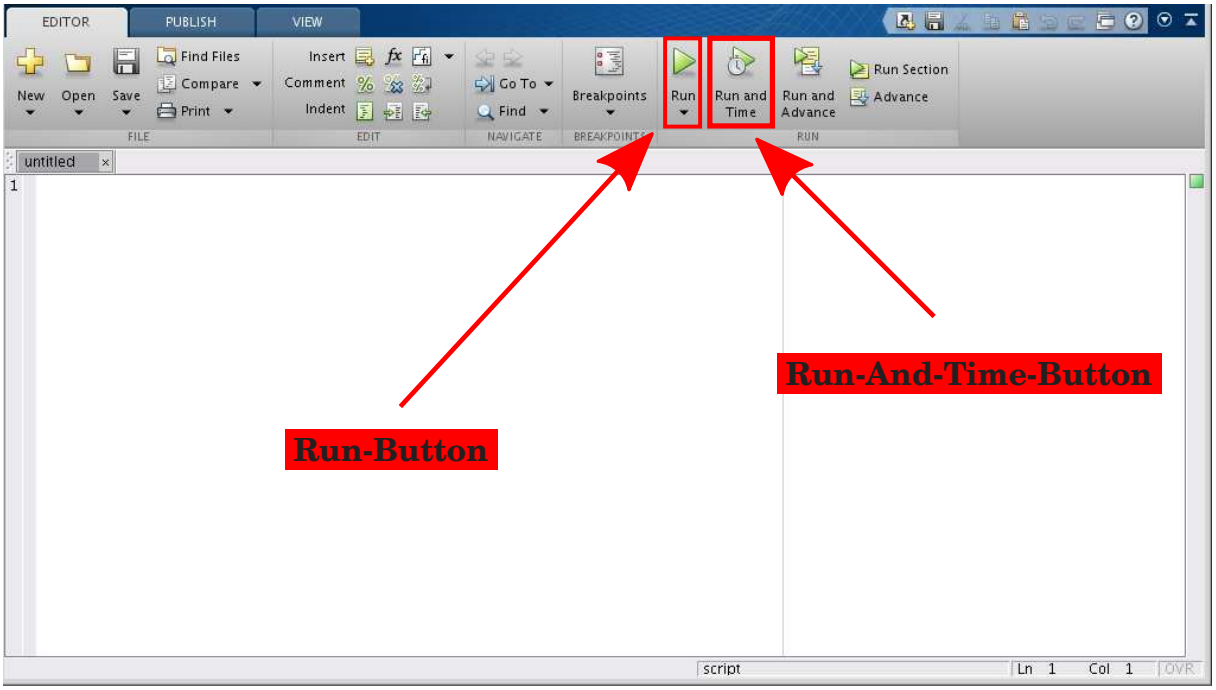


Abbildung: MATLAB-Oberfläche

Zur Entwicklung größerer Projekte wird neben der MATLAB-Konsole ein Editor benötigt. Dieser ist in MATLAB integriert und erlaubt die Erstellung von MAT-

LAB-Skripten und Funktionen. Der MATLAB-Editor wird in der Konsole mit dem Befehl: `edit` oder durch Klick auf den **New-Script-Button** gestartet.
Ein Skript ist eine Textdatei `SkriptName.m`, deren Inhalt zeilenweise abgearbeitet wird. So als ob die Zeilen nacheinander in die Konsole eingegeben würden. Dieses wird ausgeführt, indem man auf den **Run-Button** klickt, oder den Namen des Skriptes (ohne die `.m` -Endung) in der Konsole eingibt. Bevor wir eigene Skripte erstellen werden (siehe Abschnitt 2.3), wollen wir uns mit den Grundlagen der MATLAB-Syntax vertraut machen.

2 Syntax

2.1 Matrizen

Einer der großen Vorteile von MATLAB ist ein bequemer Umgang mit Matrizen. Eine Matrix wird in MATLAB in eckigen Klammern zeilenweise eingegeben. Dabei werden die Elemente einer Zeile durch ein Leerzeichen[1] oder ein Komma getrennt. Die Zeilen werden durch ein Semikolon getrennt. Soll die Matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

unter dem Namen `M` abspeichert werden, so lautet der Befehl:

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

   1   2   3
   4   5   6
```

Außer den gewohnten Matrix-Operationen wie Addition, Multiplikation und Transponierung bietet MATLAB weitere nützliche Funktionen. Diese sollen an Beispielen vorgestellt werden. Seien

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

Wir definieren die Matrizen in MATLAB:

```
>> A = [1 2 3;4 5 6];
>> B = [6 5 4;3 2 1];
>> C = [-2 2;1 1];
```

Mit dem Befehl

```
>> A + B
ans =

   7   7   7
   7   7   7
```

[1] Im Folgenden wird diese Methode der Eingabe verwendet.

erhalten wir die Summe $A+B$. Um das Matrixprodukt $C \cdot A$ zu berechnen, verwenden wir:

```
>> C * A
ans =

   6   6   6
   5   7   9
```

Mittels

```
>> C ^ 2
ans =

   6  -2
  -1   3
```

wird C^2 bestimmt. Tippt man einen Punkt . vor einem Operator, so wird die Operation elementweise durchgeführt:

```
>> A .* B

ans =

    6   10   12
   12   10    6

>> C .^ 2

ans =

   4   4
   1   1
```

Funktionen wie $\sin, \cos, \exp$ etc. können ebenfalls auf einzelne Elemente einer Matrix angewandt werden. Mit dem Befehl

```
>> exp(C)
ans =

   0.13534   7.38906
   2.71828   2.71828
```

erhalten wir die Matrix

$$\begin{pmatrix} \mathrm{e}^{-2} & \mathrm{e}^{2} \\ \mathrm{e}^{1} & \mathrm{e}^{1} \end{pmatrix}.$$

Soll das Matrixexponential

$$\mathrm{e}^{C}$$

bestimmt werden, dann verwenden wir die `expm`-Funktion:

```
>> expm(C)
ans =

   0.71581   2.27451
   1.13726   4.12759
```

Mithilfe des Befehls

```
>> A'
ans =

   1   4
   2   5
   3   6
```

erhalten wir die transponierte Matrix A^T. Im Falle einer komplexwertigen Matrix $Z \in \mathbb{C}^{m,n}$, $m, n \in \mathbb{N}$ erhalten wir mittels `'` die adjungierte Matrix $Z^H = \bar{Z}^T$. Beispiel:

```
>> Z = A + [1i 1i 1i; 1i 1i 1i]

Z =
    1.0000 + 1.0000i    2.0000 + 1.0000i    3.0000 + 1.0000i
    4.0000 + 1.0000i    5.0000 + 1.0000i    6.0000 + 1.0000i
>> Z'

ans =
    1.0000 - 1.0000i    4.0000 - 1.0000i
    2.0000 - 1.0000i    5.0000 - 1.0000i
    3.0000 - 1.0000i    6.0000 - 1.0000i
```

Die Inverse C^{-1} erhalten wir mittels:

```
>> inv(C)
ans =

  -0.25000   0.50000
   0.25000   0.50000
```

Die Software bietet auch eine Möglichkeit lineare Gleichungen zu lösen. Gesucht sei $x \in \mathbb{R}^2$ mit

$$Cx = \begin{pmatrix} 2 \\ 3 \end{pmatrix} .$$

Die Lösung liefert uns folgende Funktion:

```
--> C\[2;3]
ans =

   1
   2
```

MATLAB bietet noch weitere Matrix-Manipulationen. Es ist z. B. möglich zwei (oder mehrere) Matrizen zusammenzufügen. Mittels

```
>> [A B]
ans =

   1   2   3   6   5   4
   4   5   6   3   2   1
```

erhalten wir die Matrix

$$(A\ B)\ .$$

Die Matrix

$$\begin{pmatrix} A \\ B \end{pmatrix}$$

wird auf folgende Weise generiert:

```
>> [A;B]
ans =

   1   2   3
   4   5   6
   6   5   4
   3   2   1
```

Ferner kann auf einzelne Einträge zugegriffen werden. Die Syntax für den Aufruf des Elementes $M_{i,j}$ einer Matrix M lautet:

```
>> M(i,j)
```

Wir können den Eintrag $A_{2,3}$ der Matrix A mit dem Befehl

```
>> A(2,3)
ans = 6
```

abrufen. Die Einträge können auch verändert werden. Ein Beispiel:

```
>> A(1,1) = 8
A =

   8   2   3
   4   5   6
```

Schließlich können wir auf ganze Blöcke zugreifen. Sei

$$D = \begin{pmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{pmatrix}\ .$$

Wir definieren `D` in MATLAB

```
>> D = [16 2 3 13;5 11 10 8;9 7 6 12;4 14 15 1];
```

Mittels

```
>> D(2,:)
ans =

    5   11   10    8
```

können wir die zweite Zeile von D abrufen. Mit dem Befehl

```
>> D(:,3)
ans =
```

```
     3
    10
     6
    15
```

erhalten wir die dritte Spalte von D. Den markierten Bereich aus D

$$\begin{pmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{pmatrix}$$

rufen wir mittels

```
>> D(3:4,2:3)
ans =

     7     6
    14    15
```

auf.

2.2 Schleifen, `if`-Abfragen und Funktionen

Im Folgenden wollen wir eine Funktion schreiben, die bei der Eingabe einer Zahl $n \in \mathbb{N}$ die Ausgabe $n!$ liefert. Zur Erinnerung:

$$\begin{aligned} !: \quad \mathbb{N}_0 &\to \mathbb{N} \\ n &\mapsto n! = \begin{cases} 1 & \text{, falls } n = 0, \\ n \cdot (n-1)! & \text{, sonst.} \end{cases} \end{aligned} \tag{1}$$

Es ist leicht zu sehen, dass

$$n! = \prod_{k=1}^{n} k \tag{2}$$

gilt.
Um in MATLAB eine Funktion zu implementieren, legen wir eine Datei namens `funktionsName.m` mit dem Inhalt

```
1 function out_var = funktionsName(in_var)
2 % Funktionsbeschreibung (optional)
3 end
```

Quelltext: funktionsName.m

an. Die Funktion und die Datei müssen an dieser Stelle den gleichen Namen tragen. Mit dem Prozentzeichen `%` wird ein Kommentarbereich geöffnet, der bis zum Ende der Zeile gilt. Ein Kommentar wird von MATLAB nicht ausgeführt.
Zuerst wollen wir die rekursive Darstellung (1) implementieren, wobei sich unsere Funktion selbst aufruft. Hier der Code:

```
1  function y = fakultaetRekursiv(n)
2  % fakultaetRekursiv(n) liefert n! mittels Rekursion
3
4      if n == 0
5          y = 1;
6      else
7          y = n * fakultaetRekursiv(n - 1);
8      end
9
10 end
```

Quelltext: fakultaetRekursiv.m

In Zeile `1` wird der Name der Funktion definiert, sowie die input-Variable `n` und die output-Variable `y`. In Zeile `10` wird die Funktion beendet. In Zeile `4` wird überprüft, ob die Eingabe `n` gleich null ist. Dies erfolgt mithilfe des doppelten Gleichheitszeichens `==`. Die Abfrage wird mittels `if` begonnen und mittels `end` in Zeile `8` beendet. Lautet die Antwort *ja*, so wird die Zeile `5` ausgeführt, d. h. die output-Variable `y` wird auf `1` gesetzt. Danach wird in die Zeile `8` – hinter `end` – übergegangen. Lautet die Antwort *nein*, geht das Programm in die Zeile `7` – hinter `else` – über. Dort wird `y` auf `n` mal `fakultaetRekursiv(n - 1)` gesetzt und somit die Funktion mit dem Input `(n - 1)` erneut aufgerufen.
Die Funktion kann nun mittels

```
>> fakultaetRekursiv(5)
ans =  120
```

aufgerufen werden.
Die obige Implementierung der Fakultät-Funktion ist zwar elegant, aber eine iterativ implementierte Funktion arbeitet in der Regel schneller (siehe Abschnitt 2.3). Eine iterative Version der Fakultätsfunktion erhalten wir durch die Verwendung von (2). Diese lässt sich mittels einer `while`-Schleife verwirklichen. Hier der Code:

```
1  function y = fakultaetWhile(n)
2  % fakultaetWhile(n) liefert n! mittels einer while-Schleife
3
4      y = 1;
5
6      while n >= 1
7          y = y * n;
8          n = n - 1;
9      end
10
11 end
```

Quelltext: fakultaetWhile.m

In Zeile `4` wird die output-Variable auf `1` gesetzt. In Zeile `6` wird eine `while`-Schleife begonnen, die in Zeile `9` geschlossen wird. Der Inhalt der Schleife – Zeile `7` und Zeile `8` – wird ausgeführt, solange die Bedingung $n \geq 1$ erfüllt ist. Ist die Bedingung von Anfang an nicht erfüllt, so wird die `while`-Schleife übersprungen.

Eine weitere Möglichkeit die Fakultät-Funktion zu implementieren ist eine `for`-Schleife. Hier der Code:

```
1  function y = fakultaetFor(n)
2  % fakultaetFor(n) liefert n! mittels einer for-Schleife
3
4      y = 1;
5
6      for k = 1:1:n
7          y = y * k;
8      end
9
10 end
```

Quelltext: fakultaetFor.m

In Zeile `4` wird die output-Variable `y` auf `1` gesetzt. In Zeile `6` beginnt eine `for`-Schleife, die in Zeile `8` geschlossen wird. Die Syntax einer `for`-Schleife lautet:

```
for laufVariable = Startwert:Schrittweite:Endwert
% Inhalt der Schleife
end
```

In unserem Beispiel durchläuft die Laufvariable `k` die Werte von `1` bis `n` in `1`-er-Schritten.

Anmerkung: Soll die Schrittweite `1` betragen, kann eine verkürzte Schreibweise verwendet werden.

```
for laufVariable = Startwert:Endwert
% Inhalt der Schleife
end
```

2.3 Skripte und `plot`-Funktion

Nun wollen wir die im Abschnitt 2.2 erstellten Funktionen miteinander vergleichen. Hier der Code:

```
1  n = 80;
2
3  tic;                         % starte Zeitmessung
4      for k = 1:1:1000         % mehrmalige Ausfuehrung
5          fakultaetFor(n);
6      end
7  toc;                         % gib gemessene Zeit aus
8
9  tic;
10     for k = 1:1:1000
11         fakultaetWhile(n);
12     end
13 toc;
14
15 tic;
```

```
16      for k = 1:1:1000
17          fakultaetRekursiv(n);
18      end
19  toc;
```

Quelltext: vergleich.m

Mit den `tic`- und `toc`-Befehlen wird die Zeit zwischen den Aufrufen beider Befehle gemessen. Damit können wir überprüfen, wie viel Zeit benötigt wird um $80!$ mit den verschiedenen Ansätzen zu berechnen. Dabei werden die Funktionen jeweils 1000-mal aufgerufen – mithilfe einer `for`-Schleife – um erstens statistische Schwankungen zu minimieren und zweitens um die Rechenzeit proportional zu verlängern. Wir führen `vergleich.m` aus:

```
>> vergleich
Elapsed time is 0.001887 seconds.
Elapsed time is 0.001360 seconds.
Elapsed time is 0.442959 seconds.
```

und sehen, dass die rekursive Variante der Fakultät-Funktion die mit Abstand zeitintensivste ist.

Anmerkung: Um das Zeitverhalten eines MATLAB-Programms genau zu studieren, kann der PROFILER benutzt werden. Dieser wird mit dem **Run-And-Time-Button** des EDITORS gestartet (siehe: Abschnitt 2.4).

Damit ein Skript übersichtlich bleibt, empfiehlt es sich Zeilenumbrüche zu verwenden. Ein Zeilenumbruch wird in MATLAB mittels `...` realisiert. Hier ein Beispiel:

```
1  A = [   1 2; ...     % Zeilenumbruch mittels "..."
2          3 4];       % fuer eine bessere Lesbarkeit
3  B = [   5 6; ...     % des Skriptes
4          7 8];
5  C = A + B;
6  disp(C);            % gib die Werte der Matrix C
7                      % in der Konsole aus
```

Quelltext: beispielSkript.m

In Zeile `6` werden die Werte von der Matrix `C` mittels `disp`-Funktion in der Konsole ausgegeben. Wir führen `beispielSkript.m` aus:

```
>> beispielSkript
    6    8
   10   12
```

Jetzt wollen wir uns mit dem `plot`-Befehl beschäftigen. Wir fangen mit einem Minimalbeispiel an.

```
1  x = linspace(0,3*pi,100);
2  y = sin(x);
3  plot(x,y);
```

Quelltext: plotMinimalbeispiel.m

In Zeile 1 erstellen wir mittels `linspace` eine (1×100)-Matrix bzw. ein Zeilenvektor `x` mit äquidistanten Einträgen zwischen 0 und 3π. Die Funktion `linspace` hat den Vorteil, dass der Abstand benachbarter Werte nicht manuell berechnet werden muss. In Zeile 2 wenden wir die `sin`-Funktion elementweise auf `x` an und speichern das Ergebnis in dem Vektor `y`. In Zeile 3 wird ein Graph der Sinusfunktion erstellt, in dem die Werte von `x` und `y` gegeneinander aufgetragen werden. Obiges Skript liefert folgende Grafik.

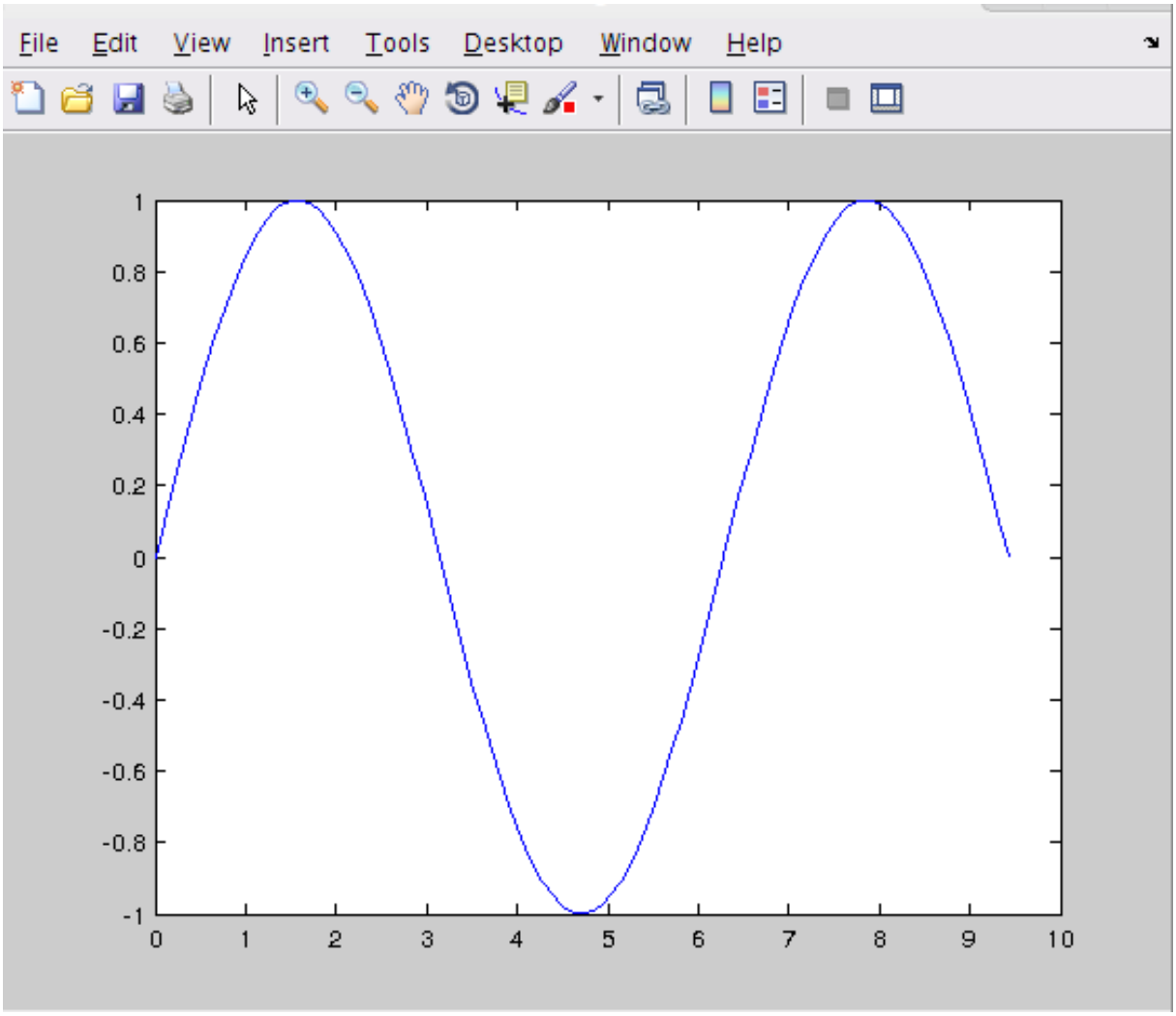


Abbildung: plotMinimalbeispiel.m

MATLAB bietet zahlreiche Möglichkeiten die Grafik zu manipulieren. Hier ein komplexeres Beispiel:

```
1 x = linspace(0,3*pi,100);          % Gitter
2 y1 = sin(x);                       % y1-Werte
3 y2 = cos(x);                       % y2-Werte
4
5 hold on;                           % figure anhalten
6 plot(x,y1,'Color','r','LineStyle','--','lineWidth',3);
7 plot(x,y2,'k-','lineWidth',5);
8
9 legend('sin(x)','cos(x)');         % Legende
```

```
10 title('Zwei Graphen');           % Titel
11 xlabel('x-Achse');               % Beschriftung x-Achse
12 ylabel('y-Achse');               % Beschriftung y-Achse
13
14 hold off;                        % figure nicht mehr anhalten
```

Quelltext: plotErweitert.m

Die Funktion `hold on` erlaubt mehrere Graphen nacheinander zu plotten, ohne die vorigen zu überschreiben. In Zeile `6` wird die `plot`-Funktion mit zusätzlichen Parametern – Farbe, Linienform und Linienstärke – aufgerufen. In Zeile `7` wird eine kompaktere Parameterübergabe vorgestellt, in der Linienfarbe und Linienform in einem String übergeben werden. In den Zeilen `9` bis `12` werden eine Legende, Grafiktitel und Achsenbeschriftung eingestellt.

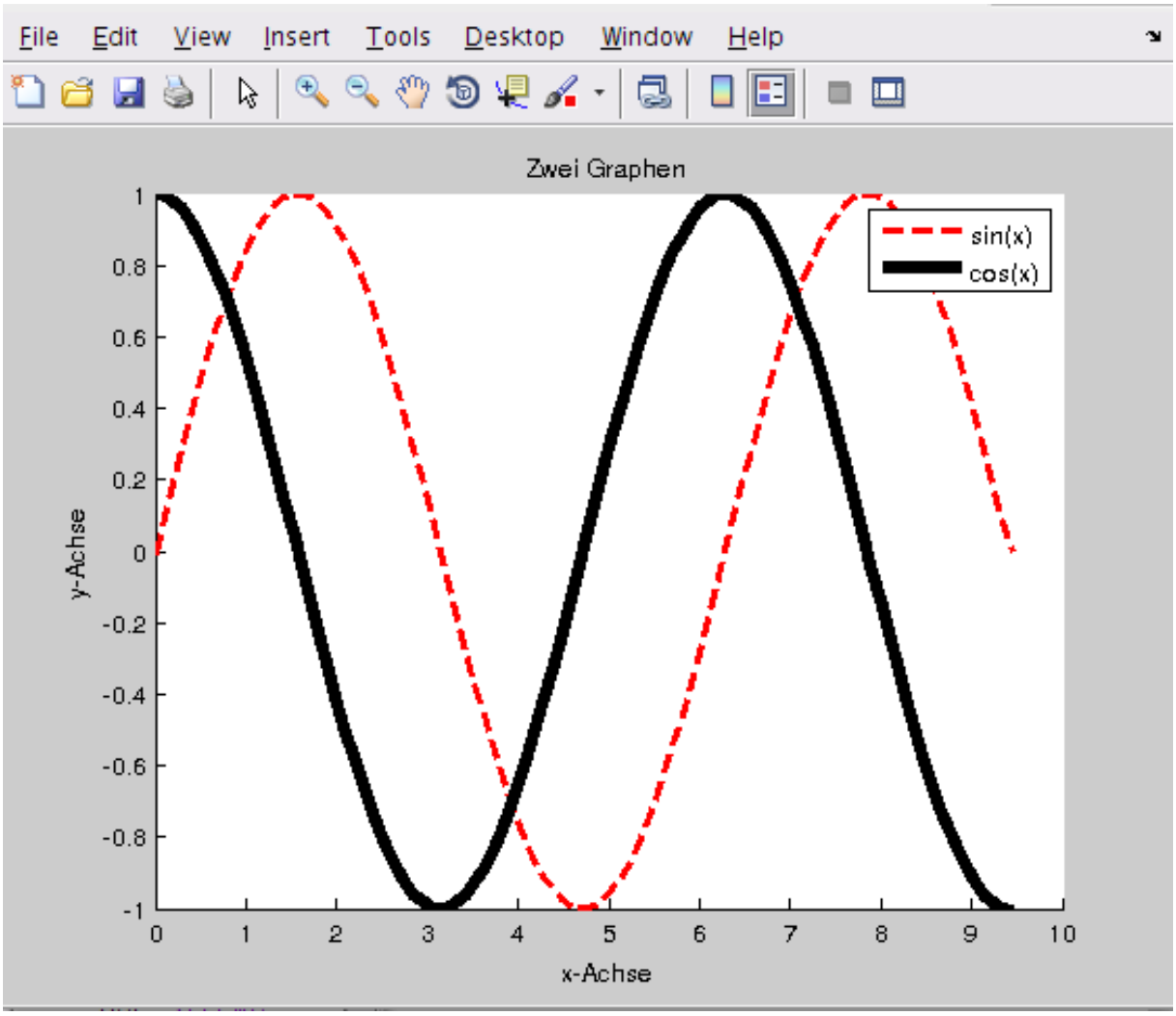


Abbildung: plotErweitert.m

2.4 Profiler

Zur Analyse und Optimierung der Rechenzeit einzelner Programmschritte, bietet sich die Verwendung des *Profilers* an. Der Profiler wird mittels des **Run-And-**

Time-Buttons gestartet. Wir wollen die Profiler-Ausgabe an einem Beispiel nachvollziehen. Dazu ergänzen wir das `vergleich.m`-Skript:

```
n = 60;
n = 80;

if n > 170
    disp('n! kann nicht dargestellt werden.')
else

    tic;                        % starte Zeitmessung
        for k = 1:1:1000        % mehrmalige Ausfuehrung
            fakultaetFor(n);
        end
    toc;                        % gib gemessene Zeit aus

    tic;
        for k = 1:1:1000
            fakultaetWhile(n);
        end
    toc;

    tic;
        for k = 1:1:1000
            fakultaetRekursiv(n);
        end
    toc;

    tic;
        for k = 1:1:1000
            factorial(n);       % Matlab-interne Funtkion
        end
    toc;
end
```

Quelltext: vergleichProfiler.m

In einem Fenster erscheint nun folgendes Bild.

Profile Summary
Generated 02-May-2015 14:22:14 using cpu time.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
vergleichProfiler	1	1.519 s	0.000 s	
fakultaetRekursiv	81000	1.104 s	1.104 s	
fakultaetWhile	1000	0.224 s	0.224 s	
fakultaetFor	1000	0.166 s	0.166 s	
factorial	1000	0.025 s	0.025 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Abbildung: Profile-Ausgabe – Profile Summary

Dort sehen wir, welche Skripte bzw. Funktionen wie oft aufgerufen wurden und wie lange jeweilige Aufrufe gedauert haben. Zu einer detaillierteren Ansicht gelangen wir über den Hyperlink `vergleichProfiler`.

vergleichProfiler (1 call, 1.519 sec)
Generated 02-May-2015 14:22:14 using cpu time.
script in file /home/lukas/MatlabEinfuehrung/Beispiele/vergleichProfiler.m

Copy to new window for comparing multiple runs

Parents (calling functions)
No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
22	`fakultaetRekursiv(n);`	1000	1.104 s	72.7%	
16	`fakultaetWhile(n);`	1000	0.224 s	14.8%	
10	`fakultaetFor(n);`	1000	0.166 s	10.9%	
28	`factorial(n);  % Matlab-i...`	1000	0.025 s	1.6%	
31	`end`	1	0 s	0%	
All other lines			0.000 s	0.0%	
Totals			1.519 s	100%	

Abbildung: Profile-Ausgabe – vergleichProfiler Teil 1

In dem oberen Teil der Ausgabe sehen wir der Reihe nach den Namen der Funktion bzw. hier des Skriptes, das Datum der Berechnung und den Dateipfad. Im weiteren Teil – **Parents** – erfahren wir, dass `vergleichProfiler.m` von keiner Überfunktion aufgerufen wurde. Danach werden die ausgeführten Zeilen nach ihrer Zeitdauer aufgelistet. In diesem Fall verbraucht Zeile `22`, in der `fakultaetRekursiv(n)` berechnet wird, die meiste Rechenzeit. Nun folgt der Abschnitt **Children**:

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
fakultaetRekursiv	function	1000	1.104 s	72.7%	
fakultaetWhile	function	1000	0.224 s	14.8%	
fakultaetFor	function	1000	0.166 s	10.9%	
factorial	function	1000	0.025 s	1.6%	
Self time (built-ins, overhead, etc.)			0 s	0%	
Totals			1.519 s	100%	

Abbildung: Profile-Ausgabe – vergleichProfiler Teil 2

Hier werden die Unterfunktionen von `vergleichProfiler.m` nach ihrer Zeitdauer sortiert.
Im Abschnitt **Code Analyzer results**

Code Analyzer results

Line number	Message
1	The value assigned to variable 'n' might be unused.

Abbildung: Profiler-Ausgabe – vergleichProfiler Teil 3

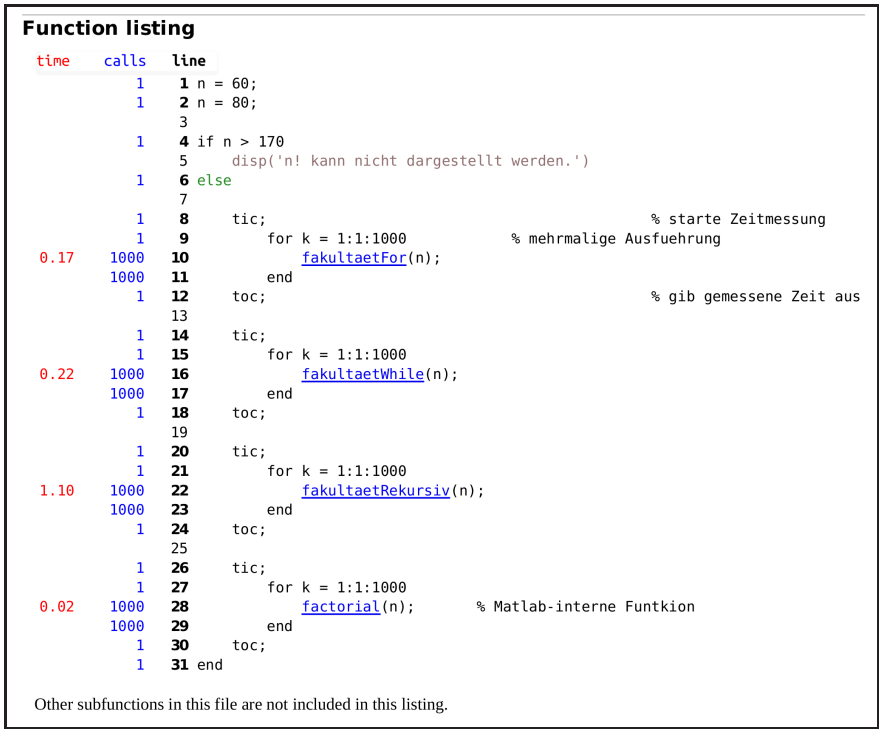
wird uns mitgeteilt, dass der Variablen `n` ein Wert zugewiesen wird, mit dem nicht weiter gerechnet wird. In Zeile `2` wird der Wert von `n` nämlich überschrieben. Folglich weist der Profiler auf Schwächen des Codes hin.
Im Abschnitt **Coverage results**

Coverage results
Show coverage for parent directory

Total lines in function	31
Non-code lines (comments, blank lines)	5
Code lines (lines that can run)	26
Code lines that did run	25
Code lines that did not run	1
Coverage (did run/can run)	96.15 %

Abbildung: Profiler-Ausgabe – vergleichProfiler Teil 4

werden die Zeilen von `vergleichProfiler.m` analysiert. In diesem Fall besteht das Skript aus 31 Zeilen, wovon 5 Leerzeilen sind. Von den übrigen 26 nicht-leeren Zeilen wurden 25 ausgeführt, da Zeile `5` wegen der `if`-Abfrage übersprungen wurde. Das Verhältnis der ausgeführten Zeilen zu den nicht-leeren Zeilen beträgt 96.15 %.
Der letzte Abschnitt ist **Function listing**.



Function listing

time	calls	line	
	1	1	n = 60;
	1	2	n = 80;
		3	
	1	4	if n > 170
		5	disp('n! kann nicht dargestellt werden.')
	1	6	else
		7	
	1	8	tic; % starte Zeitmessung
	1	9	for k = 1:1:1000 % mehrmalige Ausfuehrung
0.17	1000	10	fakultaetFor(n);
	1000	11	end
	1	12	toc; % gib gemessene Zeit aus
		13	
	1	14	tic;
	1	15	for k = 1:1:1000
0.22	1000	16	fakultaetWhile(n);
	1000	17	end
	1	18	toc;
		19	
	1	20	tic;
	1	21	for k = 1:1:1000
1.10	1000	22	fakultaetRekursiv(n);
	1000	23	end
	1	24	toc;
		25	
	1	26	tic;
	1	27	for k = 1:1:1000
0.02	1000	28	factorial(n); % Matlab-interne Funtkion
	1000	29	end
	1	30	toc;
	1	31	end

Other subfunctions in this file are not included in this listing.

Abbildung: Profiler-Ausgabe – vergleichProfiler Teil 5

Hier werden die Aufrufe und Zeitdauer einzelner Zeilen samt Zeileninhalt dargestellt. Die nicht-ausgeführte Zeile `5` wird grau markiert.

Anmerkung: Interessanterweise liefert der Profiler Messzeiten, die von den Messzeiten des `tic-toc`-Befehls abweichen. Verblüffend dabei ist, dass sich, je nach Messmethode, die relativen Messzeiten deutlich unterscheiden. In der folgenden Tabelle werden die Messzeiten verglichen.

Methode	Zeit mit Profiler		Zeit ohne Profiler	
	absolut [s]	relativ [%]	absolut [s]	relativ [%]
`fakultaetFor.m`	0.166	10.9%	0.001225	0.4%
`fakultaetWhile.m`	0.224	14.8%	0.000952	0.31%
`fakultaetRekursiv.m`	1.104	72.7%	0.292274	94.74%
`factorial.m`	0.025	1.6%	0.014046	4.56%

2.5 Debugger

Ein auf den ersten Blick fehlerfreies Skript arbeitet möglicherweise nicht wie gewünscht. Die Schwachstelle des Codes zu identifizieren ist oft eine nicht-triviale Aufgabe. MATLAB bringt ein Werkzeug – *Debugger* – mit, das den Anwender bei der Fehlersuche unterstützt. Die Funktionsweise des Debuggers wollen wir uns an einem Beispiel anschauen.
Das sogenannte Heron-Verfahrens[2] ist eine Vorschrift, mit deren Hilfe die Quadratwurzel $\sqrt{a}$ einer Zahl $a > 0$ bestimmt werden kann. Es gilt

$$\sqrt{a} = \lim_{n \to \infty} x_n \ ,$$

[2] Anwendung des Newton-Verfahrens auf $f(x) = x^2 - a$.
Siehe Wikipedia: `http://de.wikipedia.org/wiki/Heron-Verfahren`.

mit

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}, \quad n = 0, 1, \ldots, \quad x_0 \neq 0\,. \tag{3}$$

Ansatz 1: Die Iteration (3) soll ausgeführt werden, solange $a \neq x_n^2$ gilt. Hier folgt das zugehörige MATLAB-Programm:

```
1  function x = quadratwurzel(a)
2  % Berechne sqrt(a) mithilfe des Heron-Verfahrens
3
4      x = 1;
5
6      while abs(x^2 - a) ~= 0
7          x = (x + a/x)/2;
8      end
9
10 end
```

Quelltext: quadratwurzel.m

Wir testen die Funktion:

```
>> quadratwurzel(9)

ans =

     3
```

Das Ergebnis ist hier offensichtlich richtig. Wir rufen nun

```
>> quadratwurzel(2)
```

auf und stellen fest, dass die Routine in einer Endlosschleife läuft.
Um den Fehler in **Ansatz 1** zu erkennen, verwenden wir den Debugger. Hierzu setzen wir einen *Breakpoint* in dem MATLAB-Editor. Die Zeilen des Codes, an denen ein Breakpoint platziert werden kann, sind mit einem Minuszeichen – markiert. Ein Breakpoint wird durch Klick auf ein Minuszeichen aktiviert.

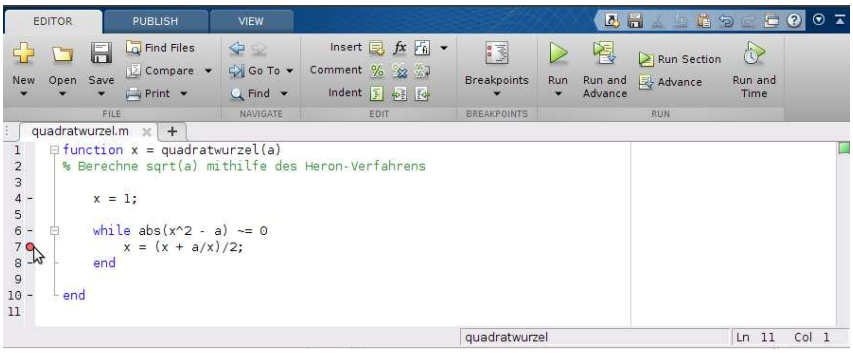


Abbildung: Breakpoint setzen.

Wir setzen einen Breakpoint in Zeile 7. Der Debugger wird beim nächsten Funktionsaufruf von `quadratwurzel.m` gestartet. Wir führen die Funktion in MATLAB-Konsole aus

```
>> quadratwurzel(2)
7            x = (x + a/x)/2;
K>>
```

Die Routine wurde nun in Zeile `7` angehalten. In dem MATLAB-Editor wird die aktuelle Zeile mit einem Pfeil markiert. Außerdem sind dort zusätzliche Buttons erschienen.

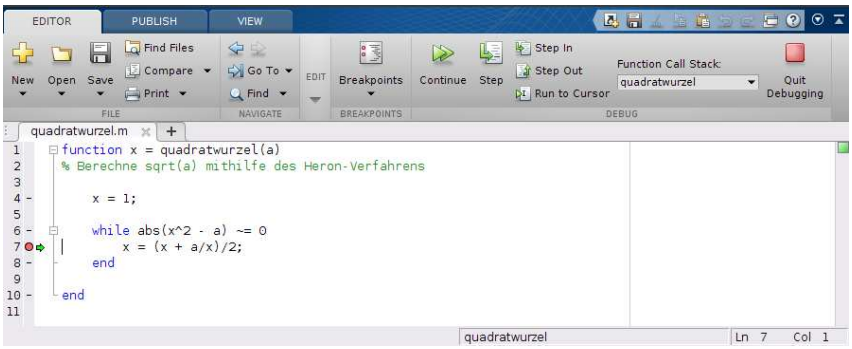


Abbildung: MATLAB-Editor im Debug-Modus.

Durch Klick auf den **Step-Button** wird die nächste Zeile der Funktion ausgeführt. In dem Variablen-Browser oder in einem Tooltip beim Überfahren einer Variablen in MATLAB-Editor können wir die aktuellen Werte der angelegten Variablen ablesen.

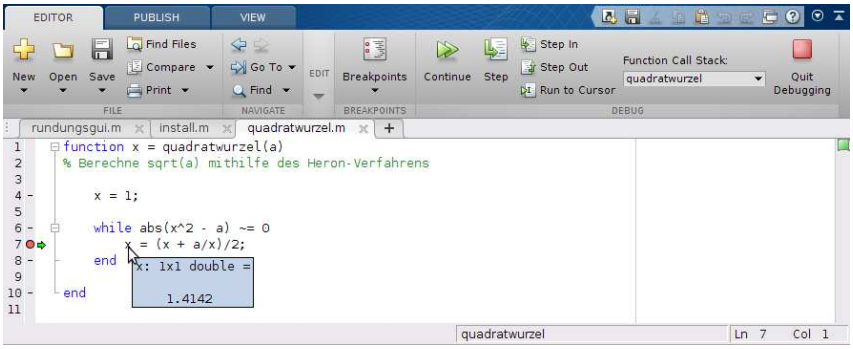


Abbildung: Tooltip beim Überfahren von `x`.

Dort stellen wir fest, dass der Wert von `x` sich $\sqrt{2}$ nähert aber ihn nie erreicht. Daher ist die Bedingung

$$a \neq x_n^2$$

immer erfüllt und die Endlosschleife liegt vor. Folglich muss die Abbruchbedingung in **Ansatz 1** modifiziert werden...

2.6 Tipps und Tricks

2.6.1 Hilfe

MATLAB ist exzellent dokumentiert und weit verbreitet. Mittels des **Help-Buttons** gelangen wir zur Dokumentation. Zu vielen dort beschriebenen werden Beispiele angegeben. Auf der Herstellerseite `www.mathworks.com` gibt es zahlreiche Videotutorials und ein User-Forum. Die Internetadresse `www.gomatlab.de` ist ein erwähnenswertes deutschsprachiges MATLAB-Forum.

2.6.2 Speicherreservierung

Soll eine $m \times n$-Matrix im Laufe eines Programms gefüllt werden, ist es ratsam den Speicher für ihre Einträge zu reservieren. Dies kann mittels des `zeros(m,n)`-Befehls erfolgen, der eine $m \times n$-Nullmatrix erzeugt. Der Vorteil der Speicherreservierung soll an einem Beispiel verdeutlicht werden.

```
n = 100000;

a = zeros(1,n);

tic;
for k = 1:n
    a(k) = k;
end
toc

tic;
for k = 1:n
    b(k) = k;
end
toc
```

Quelltext: vorreservieren.m

Im obigen Skript werden die Vektoren `a` und `b` in einer `for`-Schleife mit aufsteigenden Zahlen gefüllt. Der Vektor `a` wurde vorerst mittels `zeros` mit Nullen besetzt und hat im Laufe der Routine eine feste Größe. Die Größe des Vektors `b` wird dagegen im jeden Schleifendurchlauf um Eins erhöht. Dies macht sich in der Rechenzeit deutlich bemerkbar. Wir führen `vorreservieren.m` aus.

```
>> vorreservieren
Elapsed time is 0.000650 seconds.
Elapsed time is 0.027283 seconds.
```

2.6.3 Vektorisierung

Durch die sogenannte Vektorisierung kann ebenfalls Rechenzeit gespart werden. Dabei werden alle Einträge einer Matrix an eine MATLAB-interne Funktion übergeben und effizient verarbeitet. Im folgenden Code werden die Werte $\sin(x_k)$ für $x_k = 2\pi(k-1)/(100000-1)$, $k = 1, \ldots, 100000$ berechnet.

```
n = 100000;

x = linspace(0,2*pi,n);

y1 = zeros(1,n);
y2 = zeros(1,n);

tic;
for k = 1:n
    y1(k) = sin(x(k));
```

```
11 end
12 toc;
13
14 tic;
15 y2 = sin(x);
16 toc;
```

Quelltext: vektorisierung.m

Diese sollen in den Vektoren `y1` und `y2` gespeichert werden. Die Werte von `y1` werden schrittweise mithilfe einer `for`-Schleife eingetragen. Die Werte von `y2` werden mittels der vektorisierten MATLAB-Funktion `sin` bestimmt. Auch hier ist ein Rechenzeitunterschied spürbar.

```
>> vektorisierung
Elapsed time is 0.003459 seconds.
Elapsed time is 0.000983 seconds.
```

2.6.4 Effizient plotten

Der `plot`-Befehl benötigt relativ viel Zeit. Sollen mehrere Punkte geplottet werden, ist es daher ratsam alle Punkte mit einem Aufruf der `plot`-Funktion darzustellen. Im folgenden Beispiel wird eine Grafik zuerst schrittweise – 3000 Aufrufe von `plot` mit je einem Punkt – und simultan – ein Aufruf von `plot` mit 3000 Punkten – aufgebaut.

```
1  m = 3000;
2
3  tic();
4  phi = pi/120;
5  w = 0;
6
7  x = 0;
8  y = 0;
9
10 f1 = figure();
11  hold on;
12 for k = 1:m
13     x = w * cos(w);
14     y = w * sin(w);
15     w = w + phi;
16     plot(x,y,'ko');
17 end
18 hold off;
19 toc();
20
21 tic();
22 phi2 = pi/120;
23 w2 = 0;
24
25 x2 = zeros(1,m);
```

```
26 y2 = zeros(1,m);
27
28 for k = 1:m
29     x2(k) = w2 * cos(w2);
30     y2(k) = w2 * sin(w2);
31     w2 = w2 + phi2;
32 end
33
34 f2 = figure();
35
36 plot(x2,y2,'ko');
37 toc();
```

Quelltext: pktPlot.m

Wir führen das Skript aus.

```
>>  pktPlot
Elapsed time is 0.801506 seconds.
Elapsed time is 0.170653 seconds.
```

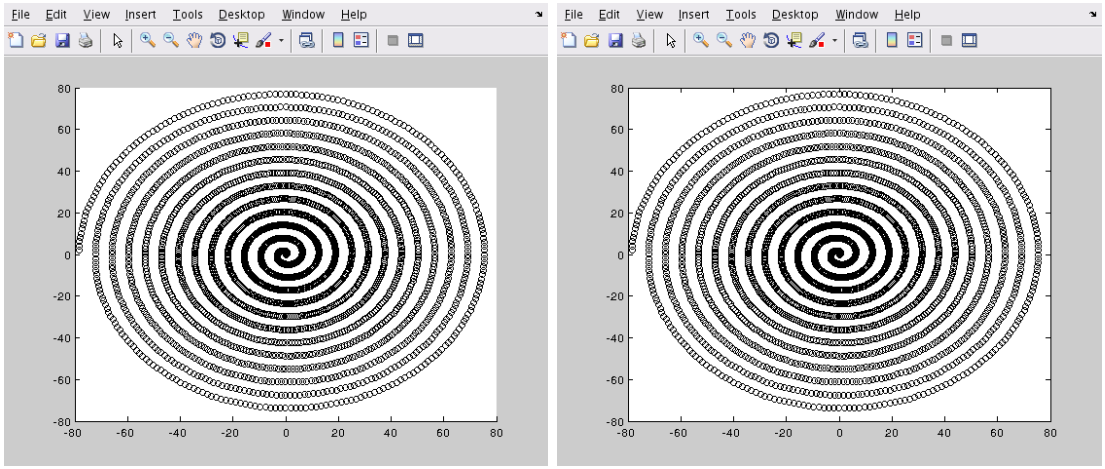


Abbildung: Von `pktPlot.m` gelieferte Grafik.

Ein weiterer Vorteil des simultanen Plots liegt darin, dass in diesem Fall alle dargestellte Punkte als ein Objekt mittels **Tools**/**Edit Plot** verarbeitet werden können.

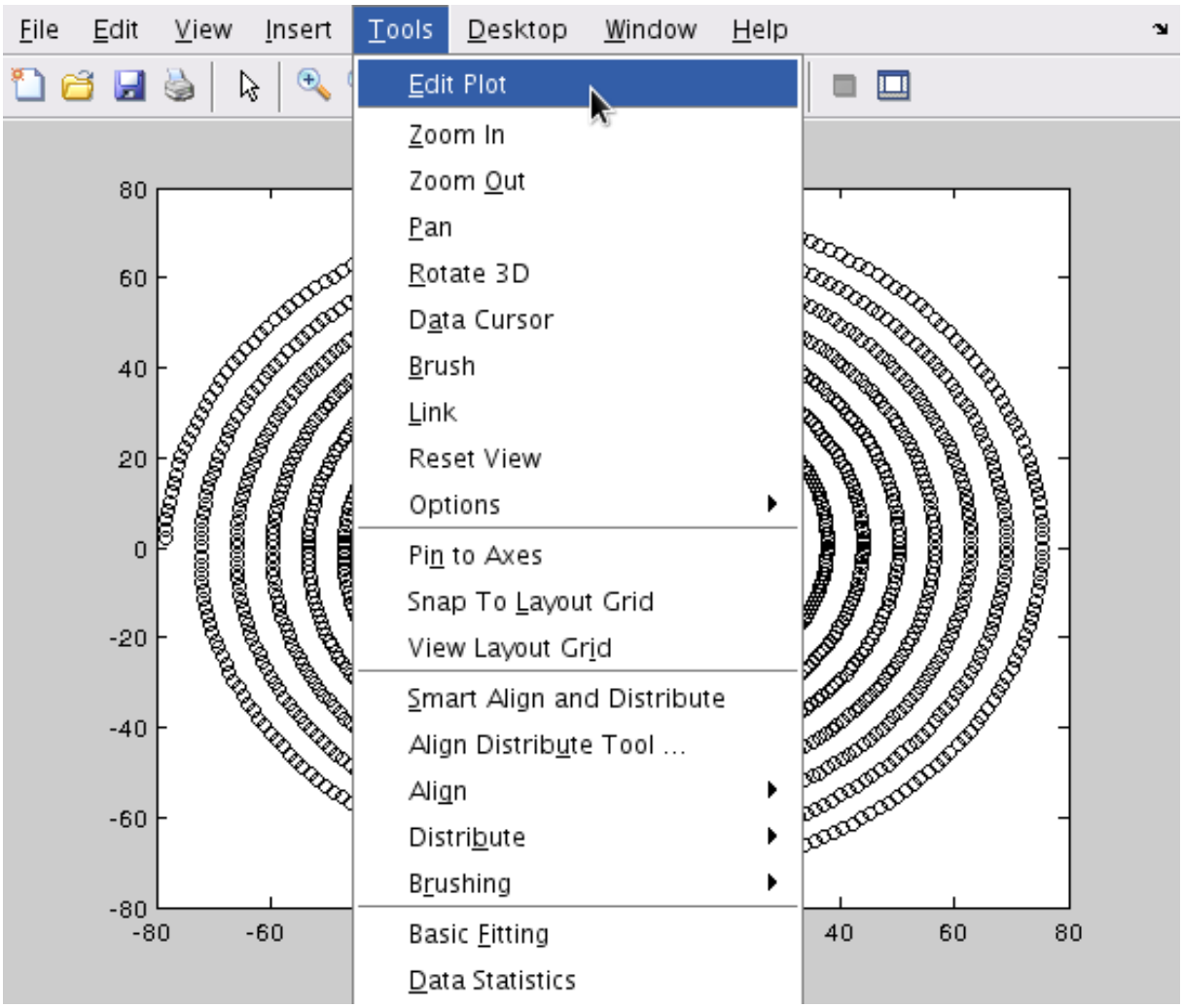


Abbildung: Edit Plot.

2.7 Einige Befehle

Zum Schluss wollen wir einige wichtige MATLAB-Befehle zusammentragen.

Ausdruck	Syntax
Potenz (x^y)	`x^y`
Quadratwurzel ($\sqrt{x}$)	`sqrt(x)`
Euklidische Norm ($\|x\|_2$)	`norm(x)`
Absolutbetrag ($\|x\|$)	`abs(x)`
Determinante	`det()`
Sinus	`sin()`
Cosinus	`cos()`
Tangens	`tan()`
Arcussinus	`asin()`
Arcuscosinus	`acos()`
Arcustangens	`atan()`
Exponentialfunktion	`exp()`
natürlicher Logarithmus	`log()`
Logarithmus zur Basis 10	`log10()`
Kreiszahl π	`pi`
Imaginäre Einheit i	`1i`

Hier werden die logischen Operatoren aufgelistet.

Bedeutung	Syntax
ist gleich	==
ist ungleich	~=
ist kleiner	<
ist kleinergleich	<=
ist größer	>
ist größergleich	>=
und	&&
oder	||
nicht	~
wahr	1
falsch	0