Eine kurze Einführung in SCILAB
Sommersemester 2015

PD Dr. Thorsten Hüls
Lukasz Targas

10.4.2015

1 SCILAB

SCILAB ist ein quelloffenes interaktives Programm zur Durchführung numerischer Berechnungen, dessen Syntax sich stark an der kommerziellen Software MATLAB orientiert. Die Anwendung kann kostenlos unter `www.scilab.org` heruntergeladen werden. Nach dem Start von SCILAB erscheint folgende Oberfläche.

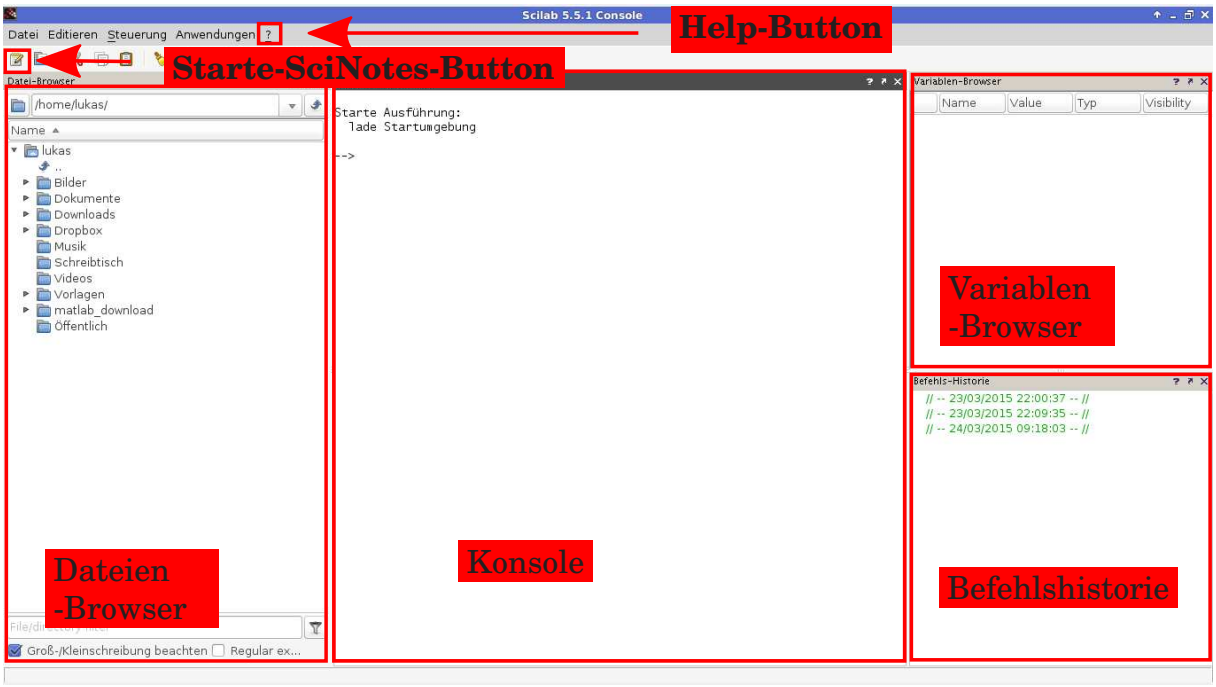

Abbildung: SCILAB-Oberfläche

Diese 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 SCILAB-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.
```

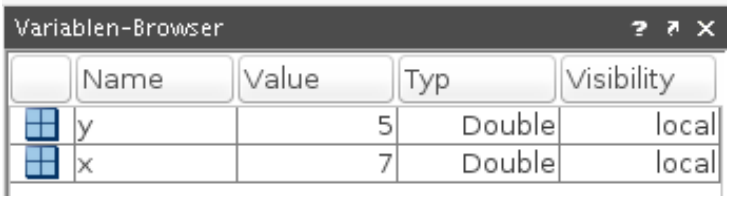

Abbildung: Variablen-Browser

1.2 Editor

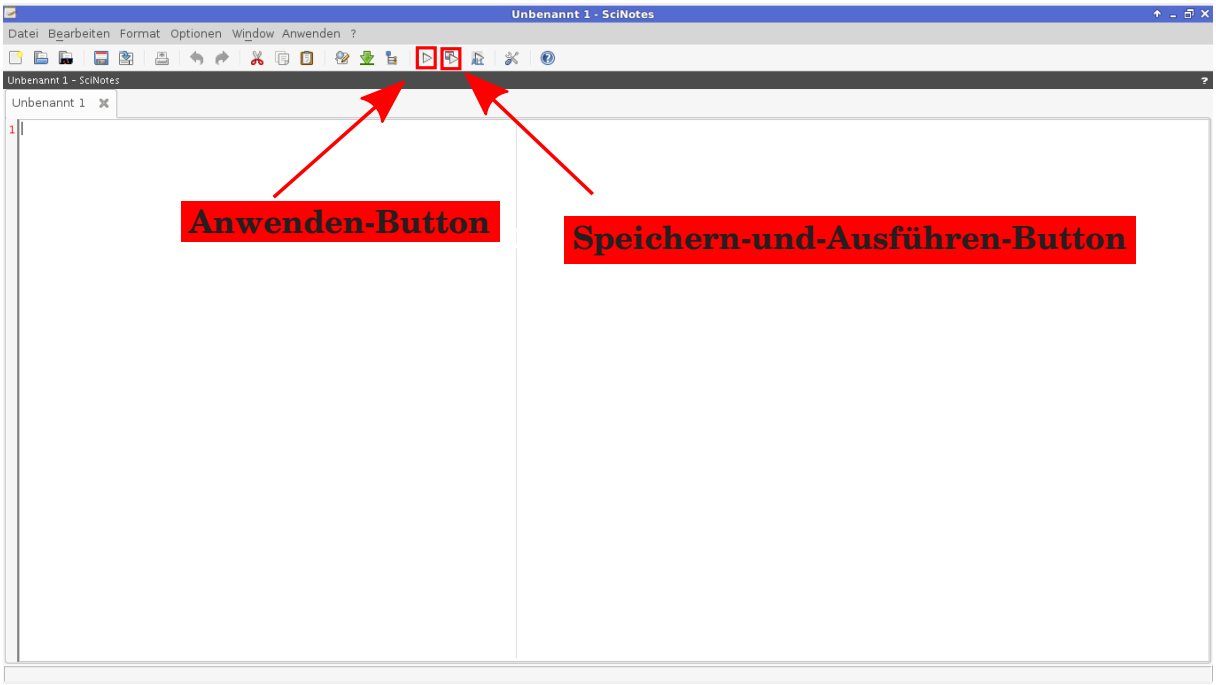

Abbildung: SciNotes

Zur Entwicklung größerer Projekte wird neben der Scilab-Konsole ein Editor benötigt. Dieser ist in Scilab integriert und erlaubt die Erstellung von Scilab-Skripten und Funktionen. Der Scilab-Editor – SciNotes – wird mit dem Befehl: `edit` bzw. `scinotes` in der Konsole oder durch Klick auf den **Starte-SciNotes-Button** gestartet.
Ein Skript ist eine Textdatei `SkriptName.sce`, deren Inhalt zeilenweise abgearbeitet wird. So als ob die Zeilen nacheinander in die Konsole eingegeben würden. Dieses wird ausgeführt, indem man auf den **Anwenden-Button**[1] bzw. **Speichern-und-Ausführen-Button** klickt, oder `exec('Name_des_Skriptes.sce',-1)` in der Konsole eingibt.
Wir wollen uns mit den Grundlagen der Scilab-Syntax vertraut machen, bevor wir eigene Skripte erstellen werden (siehe Abschnitt 2.3).

2 Syntax

2.1 Matrizen

Einer der großen Vorteile von Scilab ist ein bequemer Umgang mit Matrizen. Eine Matrix wird in Scilab in eckigen Klammern zeilenweise eingegeben. Dabei werden die Elemente einer Zeile durch ein Leerzeichen[2] oder ein Komma getrennt. Die Zeilen werden durch ein Semikolon getrennt. Soll die Matrix

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

[1] Zuvor sollte das Skript gespeichert werden, damit die Veränderungen wirksam werden.
[2] Im Folgenden wird diese Methode der Eingabe verwendet.

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 SCILAB 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 SCILAB:

```
--> 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.
```

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  =

    0.1353353    7.3890561
    2.7182818    2.7182818
```

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  =

    0.7158148    2.2745145
    1.1372572    4.1275865
```

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 + [%i %i %i; %i %i %i]
 Z  =

    1. + i      2. + i      3. + i
    4. + i      5. + i      6. + i

--> Z'
 ans  =

    1. - i      4. - i
    2. - i      5. - i
    3. - i      6. - i
```

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

```
--> inv(C)
 ans  =

  - 0.25    0.5
    0.25    0.5
```

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.
```

SCILAB 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 SCILAB

```
--> 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[3]. Zur Erinnerung:

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

Es ist leicht zu sehen, dass

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

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

```
1  function out_var = funktionsName(in_var)
2  // Funktionsbeschreibung (optional)
3  endfunction
```

Quelltext: funktionsName.sci

an. Die Funktion und die Datei müssen an dieser Stelle den gleichen Namen tragen. Mit dem doppelten Slash-Zeichen `//` wird ein Kommentarbereich geöffnet, der bis zum Ende der Zeile gilt. Ein Kommentar wird von SCILAB 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 endfunction
```

Quelltext: fakultaetRekursiv.sci

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.

[3]Unter `https://vimeo.com/91178126` befindet sich eine Videoerklärung, die die wesentlichen Inhalte dieses Abschnitts aufgreift.

Dort wird `y` auf `n` mal `fakultaetRekursiv(n - 1)` gesetzt und somit wird die Funktion mit dem Input `(n - 1)` erneut aufgerufen.
Die Funktion muss vor dem ersten Aufruf geladen werden. Dies kann entweder durch einen Klick auf den **Speichern-und-Ausführen-Button** oder mittels des Befehls `exec('fakultaetRekursiv.sci',-1)` erfolgen. Nun kann sie 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 endfunction
```

Quelltext: fakultaetWhile.sci

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 mittels einer `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 endfunction
```

Quelltext: fakultaetFor.sci

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. Wir erstellen dafür ein SCILAB-Skript[4].

```
1  exec('fakultaetFor.sci',-1);
2  exec('fakultaetWhile.sci',-1);
3  exec('fakultaetRekursiv.sci',-1);
4
5  n = 80;
6
7  tic();                       // starte Zeitmessung
8      for k = 1:1:1000         // mehrmalige Ausfuehrung
9          fakultaetFor(n);
10     end
11 zeit = toc();                // Speichere gemessene Zeit in zeit
12
13 // Erstelle Ausgabe-String. %s : Platzhalter fuer ein String
14                           // %d : Platzhalter fuer eine ganze Zahl
15                           // %f : Platzhalter fuer eine Dezimalzahl
16 msg = sprintf('Fuer %s(%d) benoetigte Zeit betrug %f Sekunden.', '
       fakultaetFor',n,zeit);
17
18 disp(msg);
19
20 tic();
21     for k = 1:1:1000
22         fakultaetWhile(n);
23     end
24
25 zeit = toc();                // Speichere gemessene Zeit in zeit
26
```

[4]Für SCILAB-Skripte hat sich die Endung `*.sce` etabliert.

```
27 msg = sprintf('Fuer %s(%d) benoetigte Zeit betrug %f Sekunden.', '
      fakultaetWhile',n,zeit);
28
29 disp(msg);
30
31 tic();                          // Speichere gemessene Zeit in zeit
32     for k = 1:1:1000
33         fakultaetRekursiv(n);
34     end
35 zeit = toc();
36
37 msg = sprintf('Fuer %s(%d) benoetigte Zeit betrug %f Sekunden.', '
      fakultaetRekursiv',n,zeit);
38
39 disp(msg)
```

Quelltext: vergleich.sce

Mit den `tic`- und `toc`-Befehlen wird die Zeit zwischen den Aufrufen beider Befehle in Sekunden 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. Die gemessene Zeit wird jeweils in der Variablen `zeit` gespeichert. Mittels `sprintf` wird ein Ausgabe-String erstellt. Wir führen `vergleich.sce` aus:

```
-->exec('vergleich.sce',-1)

 Fuer fakultaetFor(80) benoetigte Zeit betrug 0.214000 Sekunden.

 Fuer fakultaetWhile(80) benoetigte Zeit betrug 0.452000 Sekunden.

 Fuer fakultaetRekursiv(80) benoetigte Zeit betrug 1.383000 Sekunden.
```

und sehen, dass die rekursive Variante der Fakultät-Funktion die mit Abstand zeitintensivste ist.
Funktionen können auch innerhalb eines Skriptes definiert werden. Bei mittellangen Projekten – z. B. bei Übungsaufgaben – sorgt diese Variante für eine bessere Lesbarkeit des Codes. Ein Beispiel

```
1  function y = skriptFakultaet(n)
2      y = 1;
3      for k = 1:n
4          y = y * k;
5      end
6  endfunction
7
8  // berechne 1!, 2!, ... , 5!
9
10 for n = 1:5
11     disp(skriptFakultaet(n));
```

```
12 end
```

Quelltext: skriptFunktion.sce

```
--> exec('skriptFunktion.sce',-1)

    1.

    2.

    6.

    24.

    120.
```

Damit ein Skript übersichtlich bleibt, empfiehlt es sich Zeilenumbrüche zu verwenden. Ein Zeilenumbruch wird in SCILAB 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.sce

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

```
--> exec('beispielSkript.sce',-1)

    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.sce

In Zeile 1 wird mittels linspace eine (1×100)-Matrix bzw. ein Zeilenvektor x mit äquidistanten Einträgen zwischen 0 und 3π erstellt. Die Funktion linspace hat den Vorteil, dass der Abstand benachbarter Werte nicht manuell berechnet werden muss. In Zeile 2 wird die sin-Funktion elementweise auf x angewandt und das Ergebnis in dem Vektor y gespeichert. 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.

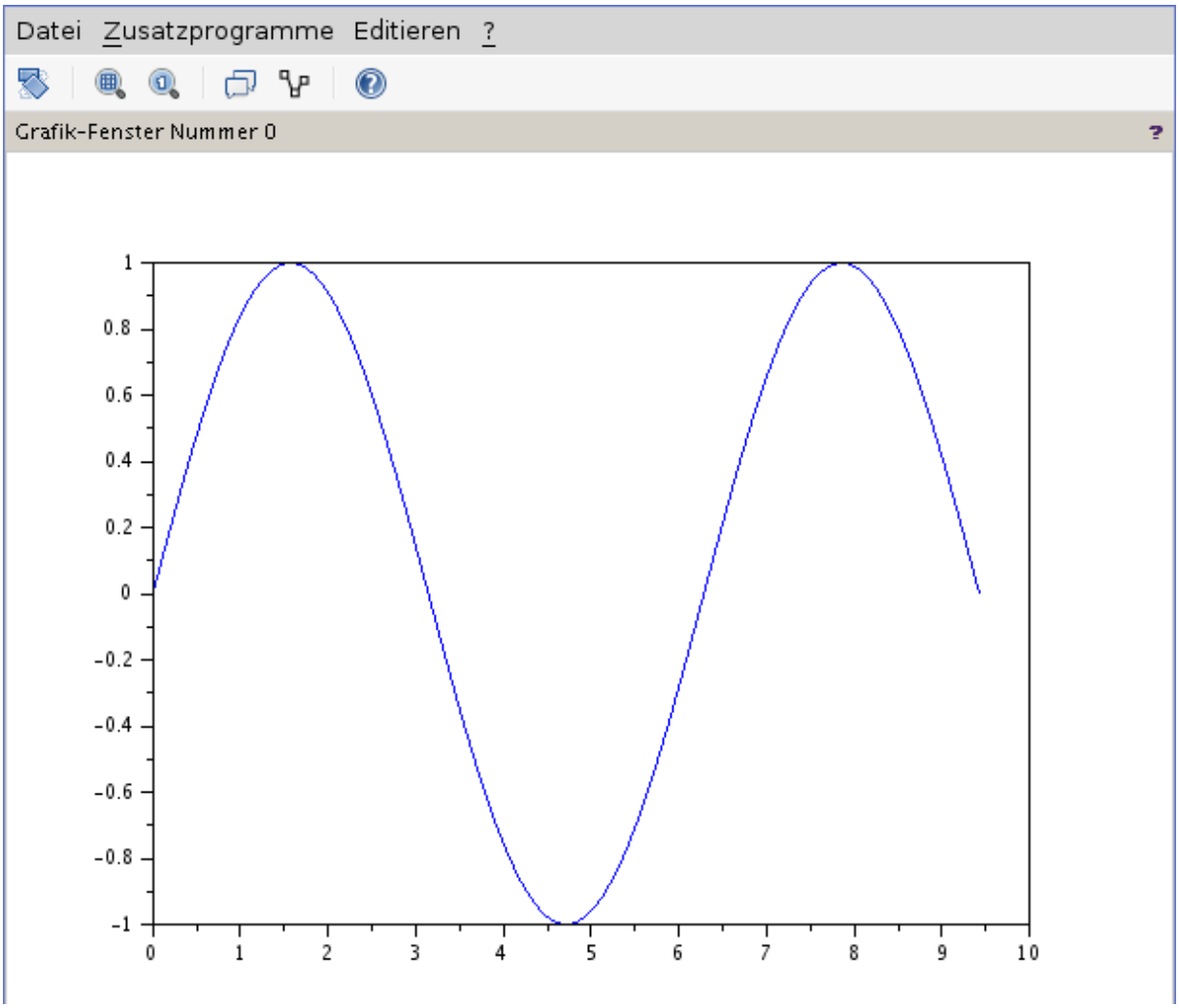

Abbildung: plotMinimalbeispiel.sce

SCILAB 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  plot(x,y1,'Color','r','LineStyle','--','lineWidth',2);
6  plot(x,y2,'k-','lineWidth',3);
7
8  legend('sin(x)','cos(x)');        // Legende
9  title('Zwei Graphen');            // Titel
10 xlabel('x-Achse');                // Beschriftung x-Achse
11 ylabel('y-Achse');                // Beschriftung y-Achse
```

Quelltext: plotErweitert.sce

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.

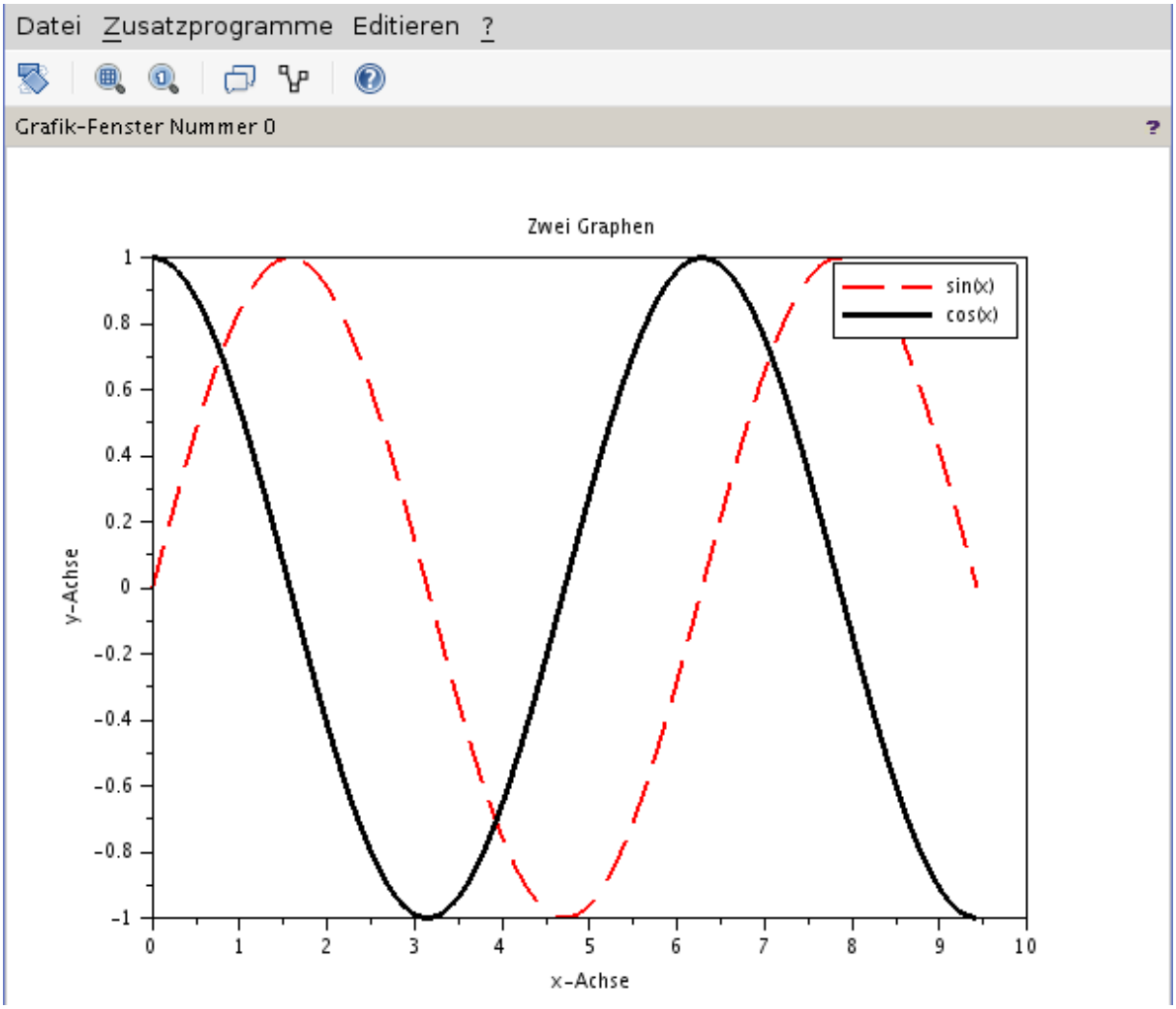

Abbildung: plotErweitert.sce

2.4 Tipps und Tricks

2.4.1 Hilfe

Mittels des **Help-Buttons** gelangen wir zur SCILAB-Dokumentation. Zu vielen dort beschriebenen Funktionen werden Beispiele angegeben.

2.4.2 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.

```
1 n = 10000;
2
```

```
3  a = zeros(1,n);
4
5  tic();
6  for k = 1:n
7      a(k) = k;
8  end
9  zeit = toc();
10 msg = sprintf('Die verstrichene Zeit betraegt %f Sekunden.',zeit);
11 disp(msg);
12
13 tic();
14 for k = 1:n
15     b(k) = k;
16 end
17 zeit = toc();
18 msg = sprintf('Die verstrichene Zeit betraegt %f Sekunden.',zeit);
19 disp(msg);
```

Quelltext: vorreservieren.sce

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 in jedem Schleifendurchlauf um Eins erhöht. Dies macht sich in der Rechenzeit deutlich bemerkbar. Wir führen `vorreservieren.sce` aus.

```
--> exec('vorreservieren.sce',-1)

 Die verstrichene Zeit betraegt 0.083000 Sekunden.

 Die verstrichene Zeit betraegt 0.663000 Sekunden.
```

2.4.3 Vektorisierung

Durch die sogenannte Vektorisierung kann ebenfalls Rechenzeit gespart werden. Dabei werden alle Einträge einer Matrix an eine SCILAB-interne Funktion übergeben und effizient verarbeitet. Im folgenden Code werden die Werte $\sin(x_k)$ für $x_k = 2\pi(k-1)/(10000-1)$, $k = 1, \ldots, 10000$ berechnet.

```
1  n = 10000;
2
3  x = linspace(0,2*%pi,n);
4
5  y1 = zeros(1,n);
6  y2 = zeros(1,n);
7
8  tic();
9  for k = 1:n
10     y1(k) = sin(x(k));
11 end
12 zeit = toc();
```

```
msg = sprintf('Die vrestrichene Zeit betraegt %f Sekunden.',zeit);
disp(msg);

tic();
y2 = sin(x);
zeit = toc();
msg = sprintf('Die verstrichene Zeit betraegt %f Sekunden.',zeit);
disp(msg);
```

Quelltext: vektorisierung.sce

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 SCILAB-Funktion `sin` bestimmt. Auch hier ist ein Rechenzeitunterschied spürbar.

```
--> exec('vektorisierung.sce',-1)

 Die verstrichene Zeit betraegt 0.150000 Sekunden.

 Die verstrichene Zeit betraegt 0.002000 Sekunden.
```

2.4.4 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 – 120 Aufrufe von `plot` mit je einem Punkt – und simultan – ein Aufruf von plot mit 120 Punkten – aufgebaut.

```
m = 120;

tic();
phi = %pi/15;
w = 0;

x = 0;
y = 0;

f = figure('background',[8]);
for k = 1:m
    x = w * cos(w);
    y = w * sin(w);
    w = w + phi;
    plot(x,y,'ko');
end

zeit = toc();
msg = sprintf('Die verstrichene Zeit betraegt %f Sekunden.',zeit);
disp(msg);

```

```
tic();
phi2 = %pi/15;
w2 = 0;

x2 = zeros(1,m);
y2 = zeros(1,m);

for k = 1:m
    x2(k) = w2 * cos(w2);
    y2(k) = w2 * sin(w2);
    w2 = w2 + phi2;
end


f2 = figure('background',[8]);
plot(x2,y2,'ko');

zeit = toc();
msg = sprintf('Die verstrichene Zeit betraegt %f Sekunden.',zeit);
disp(msg);
```

Quelltext: pktPlot.sce

Wir führen das Skript aus.

```
--> exec("pktPlot.sce",-1)

  Die verstrichene Zeit betraegt 4.625000 Sekunden.

  Die verstrichene Zeit betraegt 0.278000 Sekunden.
```

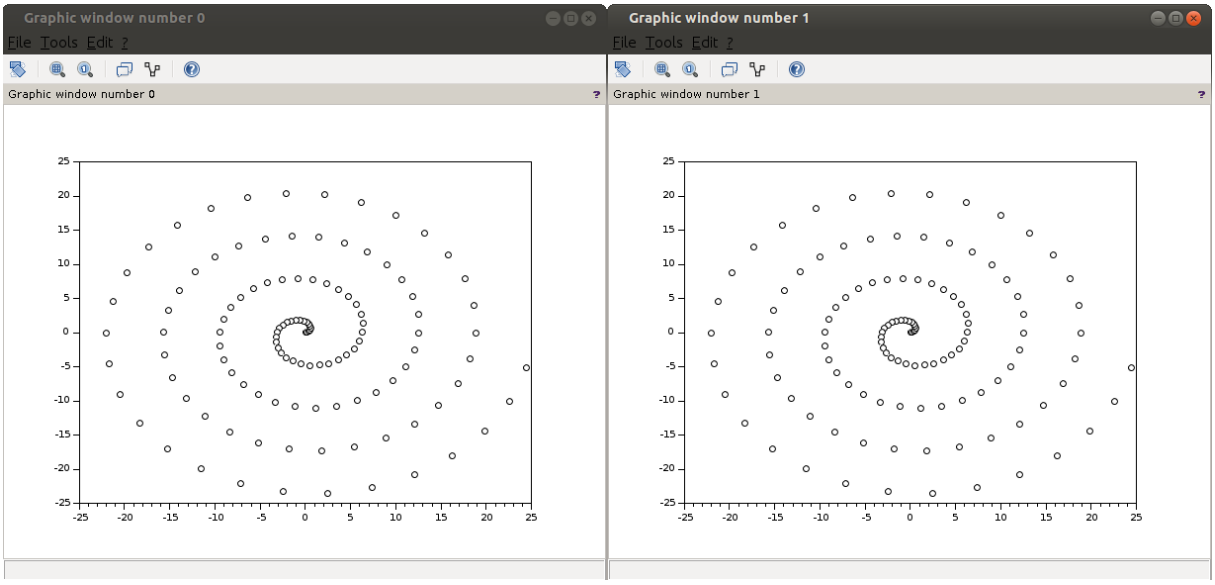

Abbildung: Von `pktPlot.sce` gelieferte Grafik.

2.5 Einige Befehle

Zum Schluss wollen wir einige wichtige SCILAB-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	`%i`

Hier werden die logischen Operatoren aufgelistet.

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