Programmieren mit Scilab

Sebastian Becker

30. September 2014

Inhaltsverzeichnis

Kapitel 1

Einführung

Das Ziel dieses Kurses ist das Erlernen von grundlegenden Kenntnissen und das Sammeln von ersten Erfahrungen im Umgang mit SCILAB. Dabei werden die wesentlichen Grundzüge der Programmiersprache dargestellt und im kurzen erläutert. Dieses Skript kann aufgrund des Umfangs keinen vollständigen Überblick über alle Elemente der Sprache und den Umgang mit SCILAB bieten, stellt aber in jedem Fall eine breite Basis für die Benutzung bereit. Aufgrund der syntaktischen Verwandschaft mit MATLAB wird dem Leser auch ein sehr leichter Einstieg in diese Sprache ermöglicht.

1.1 Über SCILAB

SCILAB ist eine Programmiersprache in Verbindung mit einer großen Anzahl von numerischen Algorithmen zum Lösen einer Vielzahl von numerischen Problemen. Das SCILAB Projekt wurde 1990 von Forschern der INRIA and École nationale des ponts et chaussées (ENPC) ins Leben gerufen. Seit Version 5 steht Scilab unter der GPL-kompatiblen Lizenz welche den freien Einsatz der Software ermöglicht. SCILAB ist in 13 verschiedenen Sprachen erhältlich und wird bereits an vielen Lehranstalten und Universitäten eingesetzt.

1.2 Installation

SCILAB ist für die Betriebssysteme Windows, GNU/Linux und MacOSX verfügbar. Unter der Adresse

```
http://www.scilab.org/download
```

sind entsprechende Installationspakete für 32 und 64-bit Plattformen erhältlich. Auch eine genaue Installationsanleitung für das jeweilige System findet sich dort. Der Quellcode von SCILAB ist dort ebenso frei zugäglich so das auch eigene Versionen für nicht untersützte Betriebssysteme erstellt werden können.

1.3 Benutzeroberfläche

Dieser Abschnitt beschreibt nur kurz die wesentlichen Bedienelemete von SCILAB. Weitere Details zu deren Verwendung werden auch in den späteren Kapiteln ge-

geben.

1.3.1 Konsole

Nach dem Starten von SCILAB erscheint zunächst die Konsole. Da SCILAB eine interpretierte Programmiersprache ist, ist das der natürliche Weg um Befehle abzusetzen. Die Befehlszeile ist mit

```
-->
```

gekennzeichnet. Abbildung 1.1 zeigt die Konsole nach dem Starten von SCILAB.

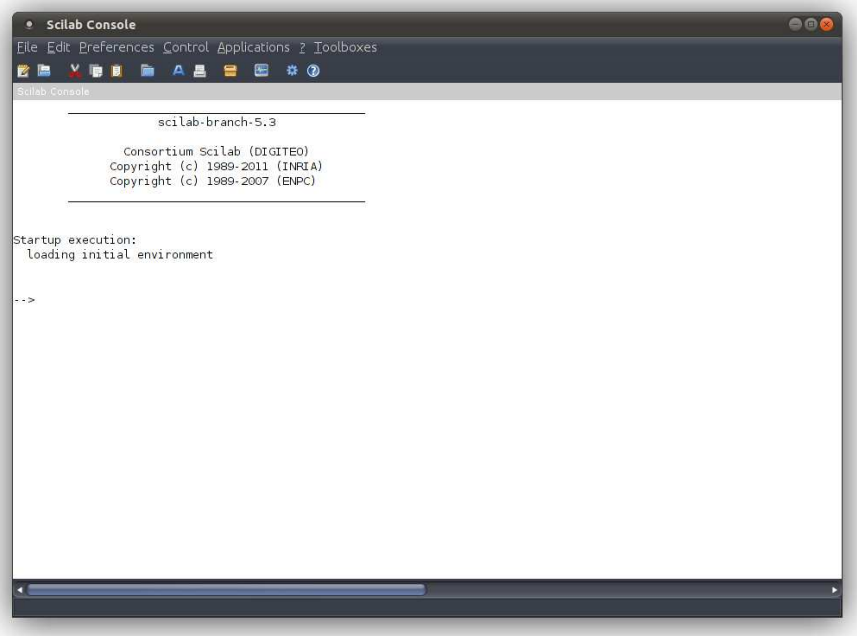

Abbildung 1.1: Die Konsole

1.3.2 Editor

Im Editor können Scripte und Funktionen bearbeitet werden. Er bietet einfaches Syntaxhighlighting für wesentliche Sprachelemete von SCILAB. Durch die Eingabe von

```
-->editor()
```

in der Konsole kann der Editor gestartet werden. Alternativ kann er über das Menü `Applications->SciNotes` aufgerufen werden. Abbildung 1.2 zeigt den in SCILAB integrierten Editor.

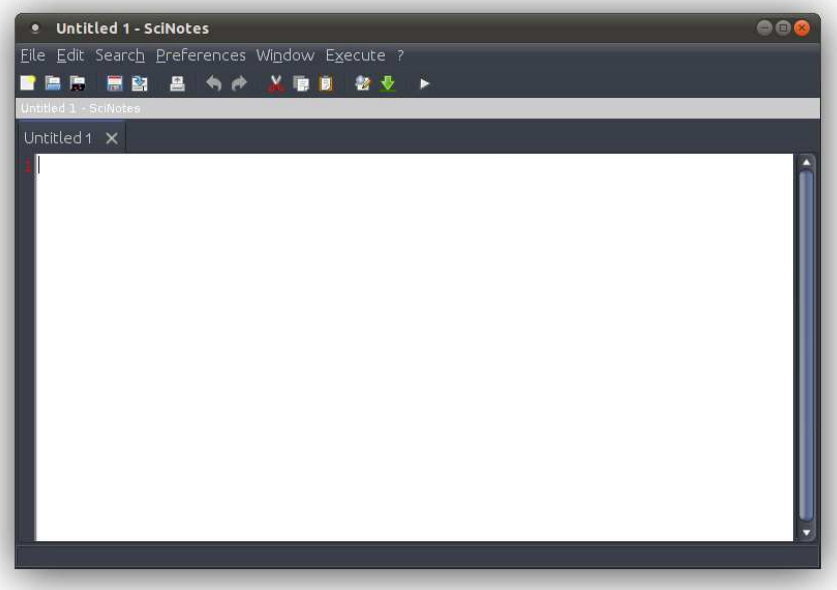

Abbildung 1.2: Der Editor

1.3.3 Hilfe

Die erste Anlaufstelle bei Fragen ist die integrierte Hilfe. Die Hilfe beinhaltet alle Befehle und alle Sprachelemete von Scilab und kann deswegen als Referenz dienen. Oftmals sind Beispiele enthalten, welche die Benutzung verdeutlichen. Der Aufruf erfolgt über Menü `?->Scilab Help` oder die Taste `F1`. Alternativ kann

```
-->help
```

in die Konsole eingegeben werden. Kennt man die Funktion zu der man Hilfe benötigt kann man diese auch als Argument mit übergeben. Zum Beispiel zeigt der Befehl

```
-->help sin
```

die Hilfe zur Sinusfunktion an. Abbildung 1.3 zeigt die Hilfe für die Sinusfunktion.

1.4 Literatur

Weitere Hilfe sowie ein Wiki, Tutorials und Videos sind unter der Adresse

`http://www.scilab.org/product/man`

erhältlich. Eine ausführliche Liste mit Büchern und Publikationen zu Scilab ist unter der Adresse

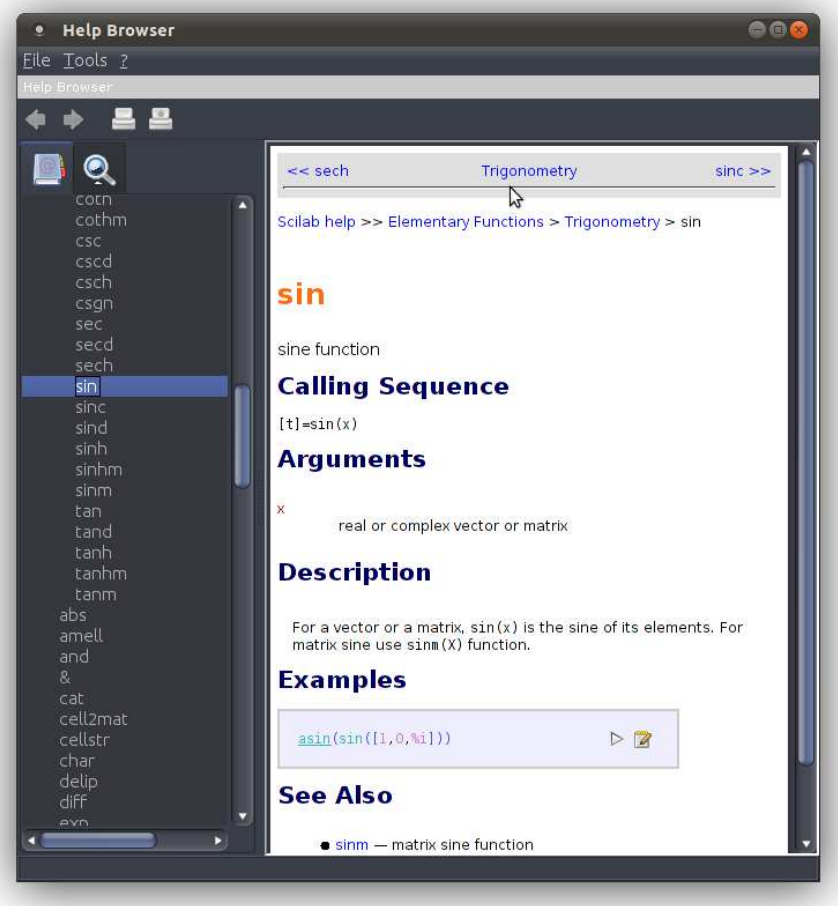

Abbildung 1.3: Die Hilfe

`http://www.scilab.org/publications`

zu finden.

1.5 Andere mathematische Softwarepakete

Es sind zahlreiche weitere mathematische Softwarepakete und Bibliotheken zum Teil frei erhältlich. Die folgende Liste gibt nur eine kleine Auswahl wieder:

- MATLAB - numerische Programmiersprache, ähnlich zu SCILAB
- GNU Octave - numerische Programmiersprache, ähnlich zu SCILAB
- maple - Computeralgebra-System, speziell für Funktional-Analysis
- gap - Computeralgebra-System, speziell für Algebra

- femlab - Finite Elemente Software zum Lösen von partiellen Differenzialgleichungen
- LAPACK - numerische Verfahren der linearen Algebra (direkte Methoden)
- SCALAPACK - numerische Verfahren der linearen Algebra für größere Probleme (direkte Methoden)
- ARPACK - numerische Verfahren der linearen Algebra (direkte Methoden)
- NAG - Bibliothek numerischer Verfahren (FORTRAN)
- NAG - Bibliothek paralleler numerischer Verfahren (C++/MPI)

Aus dieser Liste sind vor allem die ersten beiden, MATLAB und Octave, wichtig, da sie sich in der Syntax kaum von SCILAB unterscheiden. Ein Benutzer, der eines der drei Programme beherrscht, wird sich in allen drei zurechtfinden.

Kapitel 2

Datentypen

SCILAB kennt mehrere unterschiedliche Datentypen. Zum einen skalare Objekte wie Zahlen, Konstanten, Booleans (true oder false), Strings und Polynome. Aus diesen skalaren Datentypen lassen sich Matrizen definieren. Zum anderen gibt es weitere Basis-Objekte wie Listen, cells, structs und Funktionen. Mit dem Befehl `typeof(o)` kann der jeweilige Datentyp eines Objektes `o` abgefragt werden.

Dieses Kapitel soll eine Übersicht über die genannten Datentypen bieten und deren Verwendung in SCILAB anhand von Beispielen verdeutlichen.

2.1 Skalare Datentypen

2.1.1 Zahlen und spezielle Konstanten

Zahlen gehöhren in SCILAB zu den Konstanten, also z.B. ganze Zahlen, rationale Zahlen, irrationale Zahlen und komplexe Zahlen. Die Zuweisung einer Zahl zu einer Variablen erfolgt in SCILAB auf folgende Weise:

```
-->a=1;
```

Hierbei steht `a` für den Namen der neuen Variable und `=` ist der Zuweisungsoperator in SCILAB. Nun haben wir eine Variable `a` mit dem Wert `1` erzeugt. Diese Variable kann nun verwendet werden. Die Eingabe von

```
-->a
 a  =

    1.
```

gibt den Wert der Variable wieder. Grundsätzlich unterscheidet SCILAB zwischen Groß- und Kleinschreibung und somit repräsentieren `a` und `A` also zwei unterschiedliche Variablen falls sie angelegt wurden.

Bemerkung 1. *Wird ein Befehl mit* `;` *beendet so wird die nachfolgende Ausgabe unterdrückt. Wird das Semikolon weggelassen so erfolgt eine entsprechende Ausgabe wie z.B. in obigem Beispiel.*

Zusätzlich kennt SCILAB spezielle Konstanten denen ein `%` vorangestellt ist. Mit diesen speziellen Konstanten können nun z.B. komplexe Zahlen eingegeben werden. Tabelle 2.1 enthält einige wichtige spezielle Konstanten.

`%i`	$\sqrt{-1}$
`%pi`	π
`%e`	$\exp(1)$, e^1
`%eps`	die Maschinengenauigkeit des Rechners
`%nan`	not a number
`%inf`	infinity, ∞

Tabelle 2.1: Spezielle Konstanten in SCILAB

Die folgenden Beispiele sollen die Verwendung von Zahlen und speziellen Konstanten verdeutlichen:

```
-->b=1.65
 b  =

    1.65

-->c=3.8 + 2.5*%i
 c  =

    3.8 + 2.5i

-->d=b + c
 d  =

    5.45 + 2.5i

-->e=2*%pi
 e  =

    6.2831853

-->f=1e-1
 f  =

    0.1
```

Mit allen Zahlen und vordefinierten Konstanten kann in SCILAB wie mit einem Taschenrechner gerechnet werden. Die Operationen `+`, `-`, `*`, `/` und `^` (Potenz) werden automatisch unterstützt.

Bemerkung 2. SCILAB *erkennt automatisch ob eine Zahl ganzzahlig oder reell ist. Es ist jedoch auch eine explizite Konvertierung möglich. Zum Beipiel erhält man durch den Aufruf von* `int(1.5)` *die Ganzzahl* 1.

Bemerkung 3. *Entgegen der allgemeinen Notation in der Mathematik kann in* SCILAB *das Multiplikationszeichen* `*` *nicht weggelassen werden, z.b.* 2π *und* `2*%pi`. *Die Notation* `2e-1` *bedeutet* $2 \cdot 10^{-1}$.

`a==b`	true, falls `a` gleich `b`
`a~=b`, `a<>b`	true, falls `a` ungleich `b`
`<`	true, falls `a` kleiner `b`
`<=`	true, falls `a` kleiner gleich `b`
`>`	true, falls `a` größer `b`
`>=`	true, falls `a` größer gleich `b`
`a&b`	logisches UND
`a|b`	logisches ODER
`~a`	Negation

Tabelle 2.2: Vergleichs-Operatoren in SCILAB

2.1.2 Booleans

Dieser Datentyp beschreibt das Ergebnis einer logischen Aussage, also wahr (true) oder falsch (false). So liefert zum Beispiel das Prüfen auf Gleichheit mittels des binären Operators `==` ein Boolean zurück, je nachdem ob die verglichenen Objekte gleich oder ungleich sind. Tabelle 2.2 listet die Operatoren auf, die ein Boolean als Rückgabewert haben. Zusätzlich gibt es noch die speziellen Konstanen `%t` für true und `%f` für false. Des weiteren gibt es noch Operatoren, die direkt auf Booleans angewandt werden können. Dazu gehöhren das logische UND `&`, das logische ODER `|` und die Negation `~`.

Hierzu einige Beipiele:

```
-->1==1
 ans  =

  T

-->b=2<2
 b  =

  F

-->b & 3>=2
 ans  =

  F

-->~b
 ans  =

  T
```

Bemerkung 4. *Jede Variable vom Typ Boolean kann implizit in eine Zahl konvertiert werden. Dabei wird* `false` *in* 0 *und* `true` *in* 1 *umgewandelt. Umgekehrt kann jede Zahl implizit in ein Boolean konvertiert werden. Dabei steht* 0 *für* `false` *und alle anderen Zahlen für* `true`*.*

`t+s`	verknüpft die Strings `s` und `t`
`lenght(s)`	Länge der Zeichenkette `s`
`part(s, index)`	gibt die Zeichen an Position `index` zurück
`strindex(s, t)`	gibt die Position des Strings `t` im String `s` zurück
`strsubst(s, t, r)`	ersetzt in String `s` das Vorkommen von `t` durch `r`

Tabelle 2.3: String Funktionen in SCILAB

2.1.3 Strings

Auch Zeichenketten können in SCILAB definiert und in Variablen gespeichert werden. Die Zeichenkette muss dazu mit einfachen oder doppelten Anführungszeichen umschlossen werden. Das folgende Beispiel definiert zwei Variablen `s` und `t`:

```
-->s='Scilab'
 s  =

 Scilab

-->t="Tutorial"
 t  =

 Tutorial
```

SCILAB stellt bereits eine Vielzahl von Funktionen zum Arbeiten mit Strings zur Verfügung. Tabelle 2.3 listet einige eingebaute String Funktionen auf.

Zusätzlich lassen sich mittels `string(arg)` Zahlen in Strings konvertieren und mittels `evstr(arg)` Strings in Zahlen. Das nachfolgende Listing soll die Verwendung von Zeichenketten veranschaulichen:

```
-->s + " " + t
 ans  =

 Scilab Tutorial

-->length(s)
 ans  =

    6.

-->part(s, 1)
 ans  =

 S

-->part(s, [1:3])
 ans  =

 Sci
```

```
-->s + string(5)
 ans  =

 Scilab5
```

2.1.4 Polynome

Polynome können in SCILAB auf unterschiedliche Weisen definiert werden. Zum einen gibt es das spezielle Polynom `%s`, welches für das einfache Polynom $P(s) = s$ steht. Damit kann jedes beliebige Polynom gebildet werden, z.B. $P(s) = 3s^2 - 5s + 1$ durch:

```
-->3*%s^2 - 5*%s + 1
 ans  =

               2
    1 - 5s + 3s
```

Die zweite Möglichkeit ist, das Polynom mit Hilfe der Funktion

```
p = poly(arg1, arg2, arg3)
```

zu definieren. Das erste Argument `arg1` sind die Nullstellen oder Koeffizienten des Polynoms, das zweite Argument gibt den Variablennamen an. Das dritte Argument `arg3` entscheidet ob die Nullstellen (`arg3='roots'`) oder die Koeffizienten (`arg3='coeff'`) des Polynoms angegeben wurden. Das folgende Beispiel definiert zwei Polynome, das erste über die Nullstellen, das zweite mittels Koeffizienten:

```
-->p = poly([1 2], 'x', 'roots')
 p  =

              2
    2 - 3x + x

-->p = poly([1 2], 'x', 'coeff')
 p  =

    1 + 2x
```

Natürlich unterstützt SCILAB auch die gewöhnlichen Operationen auf Polynomen, wie `+`, `-`, `*` und `/`. In Tabelle 2.4 sind weitere Funktionen für Polynome aufgelistet. Eine genaue Beschreibung sowie eine vollständige Liste findet sich in der Hilfe.

2.2 Vektoren und Matrizen

2.2.1 Definition

Mit all den im vorherigen Abschnitt vorgestellten, skalaren Datentypen lassen sich Vektoren und Matrizen in SCILAB bilden. Matrizen sind die wichtigsten

`coeff(p)`	Koeffizientenmatrix des Polynoms
`degree(p)`	Grad des Polynoms
`derivat(p)`	Ableitung des Polynoms
`roots(p)`	Nullstellen des Polynoms
`factors(p)`	Faktorisierung des Polynoms
`simp(p)`	Vereinfachung des Polynoms

Tabelle 2.4: Funktionen auf Polynomen in Scilab

Objekte in Scilab und werden fast immer gebraucht. Dabei wird ein Vektor nicht anders behandelt als eine einzeilige oder einspaltige Matrix. Eine Matrix wird mit zwei eckigen Klammern (`[ ]`) definiert, zwischen denen der eigentliche Inhalt beschrieben wird. Die Elemente einer Zeile werden mit einem Komma oder Leerzeichen getrennt, die Spalten mittels eines Semikolon. Das folgende Listing definiert eine Matrix `A`, eine Matrix `B`, sowie einen Zeilenvektor `b`.

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

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

-->B = [1 2;...
-->3 4]
 B  =

    1.    2.
    3.    4.

-->b = [4 6 8]
 b  =

    4.    6.    8.
```

Bemerkung 5. *Soll ein Befehl über mehrere Zeilen umgebrochen werden, so müssen drei Punkte (`...`) verwendet werden. Das Bestätigen durch* `return` *nach den drei Punkten führt nicht zur sofortigen Ausführung des Befehls sondern setzt dessen Eingabe in der nächsten Zeile fort.*

Scilab kennt einige weitere Befehle zur Erzeugung Matrizen. Tabelle 2.5 gibt einen Überblick über diese Befehle, wobei `x` für die Anzahl der Zeilen und `y` für die Anzahl der Spalten steht. Für Vektoren steht noch die Funktion

```
linspace(x1,x2 [,n])
```

zur Verfügung. Sie gibt einen Vektor zurück dessen Elemente äquidistant zwischen den beiden Zahlen `x1` und `x2` sind. Der optionale Parameter `n` bestimmt hier die Anzahl der Elemente. Wird er weggelassen so werden automatisch 100 Elemente erzeugt. Es gibt noch eine weitere Syntax zur Erzeugung von Spaltenvektoren: `[a:c]` und `[a:b:c]`. Hier gibt `a` das erste Element des Vektors an

`eye(x,y)`	Identitätsmatrix mit `x` Zeilen und `y` Spalten
`ones(x,y)`	Matrix mit 1 überall
`zeros(x,y)`	Matrix mit 0 überall
`rand(x,y)`	Zufallsmatrix

Tabelle 2.5: Matrix Erzeuger in SCILAB

und `c` eine obere Schranke. Das optionale Argument `b` gibt das Inkrement an, wird es weggelassen verwendet SCILAB automatisch 1. Als Rückgabe erhält man eine Vektor der alle Elemente von `a` bis `c` die dem Inkrement entsprechen. Das folgende Listing illustriert die eben vorgestellten Funktionen zur Erzeugung von Vektoren und Matrizen.

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

    1.    0.    0.
    0.    1.    0.
    0.    0.    1.

-->R = rand(2,2)
 R  =

    0.0683740    0.6623569
    0.5608486    0.7263507

-->v = linspace(1,2,3)
 v  =

    1.    1.5    2.

-->w = [1:5]
 w  =

    1.    2.    3.    4.    5.

-->x = [1:1.5:5]
 x  =

    1.    2.5    4.
```

Bemerkung 6. *Auch eine* 0×0*-Matrix ist zulässig und wird mit* `[]` *definiert.*

2.2.2 Umgang mit Vektoren und Matrizen

Nachdem nun bekannt ist, wie Vektoren und Matrizen in SCILAB definiert werden widmet sich dieser Abschnitt dem Umgang mit eben diesen Objekten. Der Zugriff auf einzelne Elemente einer Matrix `A` erfolgt mittels `A(x,y)`. Wiederum steht `x` für die Zeile, `y` für die Spalte. Es ist zu beachten, dass im Unterschied

zu anderen Programmiersprachen wie z.B. C oder Java die Indices in SCILAB immer bei 1 und nicht bei 0 beginnen.

Auch der Zugriff auf ganze Vektoren oder Sub-Matrizen ist möglich. Dazu muss `A(x1:x2,y1:y2)` verwendet werden mit der Lesart von Zeile `x1` bis `x2` und von Spalte `y1` bis `y2`. Als Index kann auch `:` angegeben werden, was dann jeweils alle Zeilen- bzw. Spalten-Einträge bedeutet. Zuweisung von Werten zu einzelnen Elementen ist ebenso möglich.

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

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

-->A(3,2)
 ans  =

    8.

-->A(1,1:2)
 ans  =

    1.    2.

-->A(1,:)
 ans  =

    1.    2.    3.

-->A(2:3,2:3)
 ans  =

    5.    6.
    8.    9.

-->A(1,1) = 9
 A  =

    9.    2.    3.
    4.    5.    6.
    7.    8.    9.

-->A(:,3) = [11;11;11]
 A  =

    9.    2.    11.
    4.    5.    11.
    7.    8.    11.
```

Die Referenzierung per Index kann auch vom Ende aus beginnend angegeben

werden. Dazu wird der $-Operator verwendet. Es kann dann bequem auf den letzten Eintrag einer Spalte oder Zeile zugegriffen werden und entsprechend erhält man für `$-1` den vorletzten Eintrag usw.. Auch das Anhängen von Werten ist mit dem $-Operator problemlos möglich.

```
-->A = [1 2 3];

-->A($)
 ans  =

    3.

-->A($-1)
 ans  =

    2.

-->A($+1) = 4
 A  =

    1.    2.    3.    4.
```

2.2.3 Operatoren

Zur Manipulation und Rechnung mit Matrizen stehen in Scilab eine Vielzahl von Operatoren und Funktionen zur Verfügung. Tabelle 2.6 gibt einen Überblick über alle unitären und binären Operatoren. Es ist immer darauf zu achten, das die Dimensionen der Matrizen gemäß der Mathematik konform sind, ansonsten wird eine Fehlermeldung ausgegeben. Eine Sonderrolle nimmt der Lösungsoperator \ ein. Mit seiner Hilfe können lineare Gleichungssysteme der Form $Ax = b$ gelöst werden, falls die Matrix A invertierbar ist. Nachfolgend einige Beispiele zur Verwendung von Matrix Operatoren:

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

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

-->A+ones(3,3)
 ans  =

    2.    3.    4.
    5.    6.    7.
    8.    9.    10.

-->-A
 ans  =

  - 1.  - 2.  - 3.
```

```
  - 4.  - 5.  - 6.
  - 7.  - 8.  - 9.

-->A*A
 ans  =

    30.     36.     42.
    66.     81.     96.
    102.    126.    150.

-->A.*A
 ans  =

    1.      4.      9.
    16.     25.     36.
    49.     64.     81.

-->A.^2
 ans  =

    1.      4.      9.
    16.     25.     36.
    49.     64.     81.

-->A/5
 ans  =

    0.2     0.4     0.6
    0.8     1.      1.2
    1.4     1.6     1.8

-->A = [3 3 5; 6 2 1; 8 5 4];

-->b = [3 4 5]'
 b  =

    3.
    4.
    5.

-->b*b'
 ans  =

    9.      12.     15.
    12.     16.     20.
    15.     20.     25.

-->A\b
 ans  =
```

`-A`	Negation
`A'`	Tranposition und komplexe Konjugation
`A.'`	Tranposition
`A+B`	Addition
`A-B`	Subtraktion
`A*B`	Multiplikation
`A/B`	Division
`A^x`	Potenz, wobei `x` eine Zahl
`A.*B`	elementweise Multiplikation
`A./B`	elementweise Division von rechts
`A.\B`	elementweise Division von links
`A.^B`	elementweise Potenz
`A.^x`	elementweise Potenz, wobei `x` eine Zahl
`A\b`	Lösungsoperator, wobei `b` ein Vektor

Tabelle 2.6: Matrix Operatoren in SCILAB

```
   0.8387097
 - 0.8064516
   0.5806452
```

2.2.4 Funktionen auf Matrizen

Es gibt Funktionen die speziell auf Matrizen oder Vektoren arbeiten um Informationen abzufragen, die Gestalt zu verändern oder bestimme Werte zu errechnen. Eine sehr nützliche Funktion ist `size`, die die Dimensionen einer Matrix abfragt:

```
n = size (A, sel)
```

Der Funktion wird eine Matrix `A` übergeben, sowie eine Auswahl `sel`, die angibt ob die Anzahl der Zeilen (`sel=1` oder `sel="r"`), die Anzahl der Spalten (`sel=2` oder `sel="c"`) oder die Anzahl aller Elemente (`sel="*"`) abgefragt werden soll. Der Rückgabewert `n` enthält dann den jeweiligen Wert.

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

-->size(A, 1)
 ans  =

    3.

-->size(A, 2)
 ans  =

    3.

-->size(A, "*")
 ans  =
```

```
    9.
```

Die nächsten beiden Funktionen erlauben das Ändern von Form und Gestalt von Matrizen und geben jeweils eine neue Matrix zurück.

```
y = matrix(mat, n, m)
y = resize_matrix(mat, n, m, [type])
```

Die Funktion `matrix` ändert hierbei die Anzahl der Zeilen bzw. der Spalten der Eingabematrix `mat`. Wichtig ist, das die Anzahl aller Elemente vor und nach der Änderung übereinstimmen. Ansonsten wird ein Fehler ausgegeben. Die `resize_matrix` Funktion verhält sich ähnlich, jedoch müssen hier die Anzahl der Gesamtelemente nicht übereinstimmen. Die neue Matrix `y` wird mit den bereits in der Matrix `mat` vohandenen Einträgen an den Stellen `(i,j)` besetzt. Überschüssige Elemente werden einfach abgeschnitten und fehlende Elemente werden durch 0 ersetzt. Der optionale Parameter `type` gibt einen skalaren Datentyp an, in den die Werte der Eingabematrix konvertiert werden sollen.

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

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

-->matrix(A, 1, 9)
 ans  =

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

-->resize_matrix(A, 4, 1)
 ans  =

    1.
    4.
    7.
    0.

-->resize_matrix(A, 4, 3)
 ans  =

    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
    0.    0.    0.
```

In Tabelle 2.7 sind weitere Matrixfunktionen aufgeführt. Wird der optionale Parameter `s` für die Funktionen `sum`, `cumsum`, `prod` und `cumprod` weggelassen so arbeiten diese Funktionen auf allen Elementen oder spaltenweise. Die möglichen Werte sind `"r"` oder `1` für Zeilen und `"c"` oder `2` für Spalten. Bei der `norm` Funktion für eine Matrix gibt der optionale Parameter die Norm an. Der Standardwert

`sum(x [, s])`	Summe über Dimension `s`
`cumsum(x [, s])`	kumulierte Summe über Dimension `s`
`prod(x [, s])`	Produkt über Dimension `s`
`cumprod(x [, s])`	kumuliertes Produkt über Dimension `s`
`norm(x [, p])`	Matrixnorm
`diag(x, [, k])`	Diagonalmatrix
`tril(x)`	untere Dreiecksmatrix, Werte unterhalb der Diagonalen
`triu(x)`	obere Dreiecksmatrix, Werte oberhalb der Diagonalen
`min(x)`	Zeilen-/Spaltenminima
`max(x)`	Zeilen-/Spaltenmaxima
`sort(x)`	Sortieren
`trace(x)`	Spur der Matrix
`det(x)`	Determinante der Matrix
`orth(x)`	Orthonormal Basis der Matrix

Tabelle 2.7: Weitere Matrixfunktionen

ist `p=2`, also die Spektralnorm. Die weiteren Werte sind `p=1` (Spaltensummennorm), `p=%inf` (Zeilensummennorm) und `p='fro'` (Frobeniusnorm). Die `diag` Funktion liefert eine Diagonalmatrix zurück wobei `k` den Index der Diagonalen angibt. Der Standard ist 0 und gibt die Hauptdiagonale.

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

-->sum(A)
 ans  =

    45.

-->prod(A,1)
 ans  =

    28.    80.    162.

-->cumsum(A,2)
 ans  =

    1.    3.     6.
    4.    9.     15.
    7.    15.    24.

-->norm(A)
 ans  =

    16.848103
```

2.2.5 Dünnbesetzte Matrizen

Zum Abschluß des Abschnitts über Matrizen werden hier noch dünnbesetzte Matrizen, sogenannte sparse Matrizen vorgestellt. Diese Matrizen sind sehr hilfreich, falls sehr große Matrizen behandelt werden. Oft treten in der Mathematik Probleme auf, bei denen z.B. Bandmatrizen benötigt werden. In diesen Fällen sind meist nur eine geringe Anzahl an Einträgen verschieden von 0. Bei dünnbesetzten Matrizen werden nur diese Einträge gespeichert, was viel Platz sparen kann. Ebenso gibt es spezielle Algorithmen, die auf solchen Matrizen besonders schnell arbeiten können. Diese werden in einem späteren Kapitel noch kurz vorgestellt. Die Definition einer sparse Matrix erfolgt mit Hilfe des `sparse` Befehls:

```
sp = sparse(X)
sp = sparse(ij, v [, mn])
```

Im ersten Fall steht `X` für eine Matrix (kann auch dünnbesetzt sein) aus der eine dünnbesetzte Matrix erstellt werden soll. Im zweiten Fall gibt `ij` eine zweispaltige Matrix mit den Indices an, die besetzt werden sollen. Der Vektor `v` gibt dann die Werte an, die zum befüllen benutzt werden sollen. Die optionale zweielementige Matrix `mn` gibt die Dimension der neuen Matrix an. Natülich funktionieren alle im vorangegangenen Abschnitt vorgestellten Funktionen auch mit dünnbesetzten Matrizen. Zum Umwandeln einer dünnbesetzten Matrix in eine "normale" Matrix kann der `full(sp)` Befehl verwendet werden.

```
-->sp = sparse([0 1 0; 0 0 0; 1 0 0])
 sp  =

(    3,    3) sparse matrix

(    1,    2)        1.
(    3,    1)        1.

-->full(sp)
 ans  =

    0.    1.    0.
    0.    0.    0.
    1.    0.    0.
```

2.3 Standard Funktionen

Funktionen bilden einen weiteren wichtigen Datentyp in Scilab. Damit ist es möglich Funtionen an Variablen zu zuweisen und diese z.B. als Argument eines weiteren Funktionsaufrufes zu übergeben. Bevor im nächsten Kapitel ausführlich beschrieben wird, wie eigene Funktionen geschrieben werden können soll dieser Abschnitt einige in Scilab vorhandene Funktionen erläutern. In Tabelle 2.8 werden die trigonometrischen Funktionen bezüglich des Bogenmaßes vorgestellt und Tabelle 2.9 zeigt die trigonometrischen Funktionen bezüglich des Gradmaßes. Diese Funktionen lassen sich auch auf Matrizen anwenden, wobei sie dort

sin	cos	tan	cot
asin	acos	atan	acot
sinh	cosh	tanh	coth
asinh	acosh	atanh	acoth

Tabelle 2.8: Trigonometrische Funktionen bezgl. des Bogenmaßes

sind	cosd	tand	cotd
asind	acosd	atand	acotd

Tabelle 2.9: Trigonometrische Funktionen bezgl. des Gradmaßes

elementweise ausgeführt werden. Die Benutzung erfolgt wie gewohnt, wobei die Funktionen jeweils ein Argument haben:

```
-->sin(%pi/2)
 ans  =

    1.


-->sind([90 360])
 ans  =

    1.    0.
```

Zum Abschluss seien hier noch einige weitere standard Funktionen in Tabelle 2.10 aufgeführt. Diese lassen sich ebenfalls auf Matrizen anwenden. Eine vollständige Liste aller Funktion kann, wie bereits erwähnt, der Hilfe entnommen werden.

2.4 Listen

Die bisher betrachteten Objekte waren stets homogen. Die skalaren Objekte haben jeweils genau einen Typ und Matrizen besitzen Elemente von genau einem Typ. Das Mischen von Datentypen in Matrizen ist nicht zulässig. Um diese Einschränkung zu umgehen und einen Container für verschiedenste Objekte zu haben, gibt es in SCILAB Listen. Listen sind eine Aneinanderkettung von beliebigen Objekten. Es können Objekte an beliebiger Stelle eingefügt oder gelöscht werden. Das Erzeugen einer neuen Liste erfolgt mittels

exp(a)	Exponentialfuntion	log(a)	natürlicher Logarithmus
log2(a)	Logarithmus zur Basis 2	log10(a)	dekadischer Logarithmus
real(a)	Realteil	imag(a)	Imaginärteil
abs(a)	Betrag	sign(a)	Vorzeichen

Tabelle 2.10: Weitere standard Funktionen

`L = list(a1,....an)`

für beliebige Objekte `a1`, ..., `an`. Die folgenden Operationen auf dieser Liste sind dann möglich:

- Extraktion von Werten mit `[x,y,z...]=L(v)` wobei `v` ein Vektor von Indices ist. Alle Werte erhält man mit `L(:)`.
- Einfügen eines Wertes `a` an Index `i` mit `L(i)=a`.
- Anhängen eines Wertes `a` am Ende der Liste mit `L($+1)=e`.
- Einfügen eines Wertes `a` am Anfang der Liste mit `L(0)=a`.
- Löschen des Eintrags an Stelle `i` mit `L(i)=null()`. Die Indices der anderen Einträge werden nicht verändert. Statt dessen enthält die Liste dann einen leeren Eintrag an Stelle `i`.
- Anzahl der Elemente mit `size(L)`.
- Zusammenfügen zweier Listen mit `L3 = lstcat(L1,L2)`.

Das folgende Beispiel verdeutlicht die Verwendung von Listen:

```
-->L = list('Anfang', [1 2; 3 4], 'Ende');

-->L(1)
 ans  =

 Anfang

-->L(0) = 'noch davor'
 L  =


       L(1)

 noch davor

       L(2)

 Anfang

       L(3)

    1.    2.
    3.    4.

       L(4)

 Ende
```

Neben den normalen Listen stehen noch zwei weitere Arten von Listen zur Verfügung, `tlist` und `mlist`. Hier wird nur der Datentyp `tlist` für eine getypte Liste betrachtet. Der Datentyp `mlist` ist sehr ähnlich. Eine genaue Beschreibung findet sich in der Hilfe. Bei einer getypten Liste werden zusätzlich Typnamen mit angegeben und erlauben dann benutzerspezifische Operationen und benannten Zugriff auf diese Liste. Die allgemeine Syntax für eine getype Liste lautet dann:

```
L = tlist(typ,a1,....an)
```

Hier ist `typ` ein String oder eine Vektor mit Strings. Der erste Eintrag beschreibt immer den Typ der Liste, die restlichen, falls angegeben, die Namen der Felder.

```
-->t = tlist(["meineListe","erster","zweiter"], [], []);

-->typeof(t)
 ans  =

 meineListe

-->t.erster = 'erster Eintrag';

-->t('zweiter') = [1 2; 3 4];

-->t
 t  =


       t(1)

!meineListe  erster  zweiter  !

       t(2)

 erster Eintrag

       t(3)

    1.    2.
    3.    4.

-->t.zweiter
 ans  =

    1.    2.
    3.    4.

-->definedfields(t)
 ans  =

    1.    2.    3.
```

2.5 Cell-Arrays

Ein Block (cell) besteht aus einem Array von leeren Matrizen. Mit dem Befehl

```
c = cell(m1, m2, ..., mn)
```

kann ein cell-Array der Dimensionen `m1`, ..., `mn` erzeugt werden. Zusätzlich können die Einträge selbst wieder cell-Arrays sein. Der Zugriff auf einzelne Matrizen erfolgt, ähnlich wie bei getypten Listen, mittels des `entries` Feldes. Zur Prüfung ob es sich bei einer Variablen um ein cell-Array handelt kann die `iscell(c)` Funktion genutzt werden. Das folgende Beispiel erstellt ein cell-Array und ändert im Anschluß dessen Einträge.

```
-->c = cell(2, 2)
 c  =


!{}  {}  !
!        !
!{}  {}  !

-->c(1,1).entries = 'erste Matrix';

-->c(2,1).entries = 'zweite Matrix';

-->c(1,2).entries = 1:3;

-->c(2,2).entries = (1:3)';

-->c
 c  =


!"erste Matrix"   [1,2,3]  !
!                          !
!"zweite Matrix"  [1;2;3]  !

-->iscell(c)
 ans  =

  T

-->c(2,2).entries(1)
 ans  =

    1.
```

2.6 Structs

Eine Struktur (struct) in SCILAB ist einer getypten Liste ebenfalls sehr ähnlich. Ein struct besitzt benannte Felder, die beliebige Objekte als Wert annehmen

dürfen. Ein Zugriff auf einzelne Felder erfolgt immer per Feldnamen. Das nächste Beispiel erzeugt ein struct, das ein Datum repräsentiert.

```
->s = struct('Tag', 20, 'Monat', 'September', 'Jahr', 2011)
 s  =

   Tag: 20
   Monat: "September"
   Jahr: 2011
```

Eine Prüfung auf den Datentyp struct kann wieder mit dem Befehl `isstruct(s)` erfolgen. Zusätzlich gibt die Funktion `fieldnames(s)` die Namen der definierten Felder zurück und `isfield(s, field)` prüft ob das Feld `field` in der Struktur `s` definiert ist. Abschließend noch einige Zeilen Code zur obigen Struktur:

```
->fieldnames(s)
 ans  =

!Tag    !
!       !
!Monat  !
!       !
!Jahr   !

-->s.Tag
 ans  =

    20.

-->s.Stunde = 10;

-->s.Minute = 30;

-->s.Sekunde = 22
 s  =

   Tag: 20
   Monat: "September"
   Jahr: 2011
   Stunde: 10
   Minute: 30
   Sekunde: 22
```

Kapitel 3

Scripte und Funktionen

Scripte und Funktionen werden im wesentlichen dazu verwendet um größere Anweisungsblöcke wiederverwendbar zu machen. Zusätzlich erlauben sie eine bessere Behandlung von Programmierfehlern. Dieses Kapitel zeigt wie Scripte und Funktionen in SCILAB erstellt werden, deren Benutzung, die Behandlung von Fehlern und das Debuggen.

3.1 Scripte

Ein Script ist in SCILAB nichts anderes als eine Auflistung von mehreren Befehlen. Ein Script besitzt weder Eingabe- noch Rückgabeparameter und verwendet statt dessen globale Variablen die auch nach dem Beenden weiterhin sichtbar sind. Das heißt, ein Script kann auf alle Variablen zugreifen, die vor der Ausführung vorhanden waren und kann diese somit auch überschreiben. Nach dem Ausführen stehen zusätzlich alle Variablen die im Script definiert wurden zur Verfügung.

Um ein neues Script zu erstellen öffnet man ein neues File im Editor. Der Inhalt dieses Files kann aus allen gültigen SCILAB Befehlen, Funktionen und Kommentaren bestehen, die dann der Reihe nach abgearbeitet werden. Um die Funktionsweise von Scripten zu demonstrieren folgt nun ein einfaches Beispiel, welches im Anschluss ausführlich erläutert wird.

```
//ein einfaches Script
disp('Das ist ein erstes, einfaches Script.');

//pruefen ob die Variable A bereits existiert
if isdef('A') then
  disp('Die Variable A existiert bereits.');
  disp('Sie wird nun ueberschreiben.');
else
  disp('Die Variable A existiert noch nicht.');
  disp('Sie wird nun angelegt.');
end

A = [1 2 3];
```

Das Script prüft zuerst mit dem `isdef` Befehl ob eine Variable mit dem Namen `A` bereits vorhanden ist und ein entsprechender Text ausgegeben. Die Verwendung von *if-then-else* wird in Abschnitt 4.1.1 genauer beschrieben. Um dieses Script nun auszuführen muss es zunächst unter dem Dateinamen `simplescript.sce` abgespeichert werden. Mit dem `exec(path [,mode])` Befehl wird es dann ausgeführt. Dabei steht `path` für den Pfad und Dateinamen. Der optionale Parameter `mode` steuert die Ausgabe des Script-Quelltextes. Wird er weggelassen wird der gesamte Quelltext vor der Ausführung ausgegeben, wird `-1` angegeben wird nichts ausgegeben. Die weiteren Werte können in der Hilfe nachgelesen werden.

```
-->exec('/../home/math/becker/Desktop/scilab/simplescript.sce', -1)

 Das ist ein erstes, einfaches Script.

 Die Variable A existiert noch nicht.

 Sie wird nun angelegt.

-->A
 A  =

    1.    2.    3.
```

Eine erneute Ausführung des Scripts ergibt dann:

```
-->exec('/../home/math/becker/Desktop/scilab/simplescript.sce', -1)

 Das ist ein erstes, einfaches Script.

 Die Variable A existiert bereits.

 Sie wird nun ueberschreiben.
```

Das Überschreiben von Variablen kann gewollt sein, kann jedoch auch zu Fehlern führen falls eine Variable überschrieben wird, die später noch verwendet werden soll. Grundsätzlich ist es daher zu empfehlen mit Funktionen zu Programmieren die im nächsten Abschnitt vorgestellt werden.

Bemerkung 7. *Das obige Script verwendet Kommentare die mit* `//` *beginnen. Kommentare dienen der reinen Lesbarkeit von Quellcode und haben keinen Einfluss auf die Ausführung.*

3.2 Funktionen

Im Gegensatz zu Scripten können Funktionen zwar auf globale Variablen zugreifen aber diese nicht überschreiben. Variablendefinitionen und Wertzuweisungen sind nur lokal, also innerhalb der Funktion, sichtbar. Um dies zu veranschaulichen wird zunächst eine Funktion betrachtet die weder Eingabe- noch Rückgabeparameter verwendet. Diese Funktion soll das gleiche leisten wie das Script aus dem vorangegangenen Abschnitt. Um das Script in eine Funktion zu wandeln muss folgende Datei im Editor erstellt werden und unter dem Namen `simplefunction.sci` gespeichert werden:

```
function simplefunction()
  //eine einfache Funktion
  disp('Das ist eine erste, einfache Funktion.');

  //pruefen ob die Variable A bereits existiert
  if isdef('A') then
    disp('Die Variable A existiert bereits.');
    disp('Sie wird nun ueberschreiben.');
  else
    disp('Die Variable A existiert noch nicht.');
    disp('Sie wird nun angelegt.');
  end

  A = [1 2 3];
endfunction
```

Anschliessend muss die Funktion geladen werden um verwendet werden zu können. Dies geschied wieder mit Hilfe des `exec` Befehls. Dann kann die Funktion mit `simplefunction()` aufgerufen werden.

```
-->clear

-->exec('/../home/math/becker/Desktop/scilab/simplefunction.sci',-1)

-->simplefunction()

 Das ist eine erste, einfache Funktion.

 Die Variable A existiert noch nicht.

 Sie wird nun angelegt.

-->A
  !--error 4
Undefined variable: A
```

Wie zu erkennen ist, ist die Variable `A` nach dem Aufruf der Funktion nicht sichtbar und kann somit im Anschluss nicht verwendet werden. Ebenso können globale Variablen nicht überschrieben werden:

```
-->A = 0
 A  =

    0.

-->simplefunction()

 Das ist eine erste, einfache Funktion.

 Die Variable A existiert bereits.
```

```
 Sie wird nun ueberschreiben.

-->A
 A  =

    0.
```

Die Funktion erkennt zwar, das die Variable `A` bereits existiert und weist ihr innerhalb der Funktion einen neuen Wert zu. Nach Aufruf der Funktion steht aber immer noch der alte Wert in der Variablen `A`.

Bemerkung 8. *Anstatt Scripte und Funtionen mit dem* `exec` *Befehl zu starten bzw. zu laden können diese auch im Editor über das Menü* `Execute->Execute Into Scilab` *bzw.* `Execute->Load Into Scilab` *gestartet oder geladen werden. Eine weitere Möglichkeit zum Laden von Funktionen ist die Benutzung der* `getf` *Funktion. Diese ist aber bereits als obsolete gekennzeichnet und wird in späteren Versionen von* Scilab *nicht mehr zur Verfügung stehen.*

3.2.1 Eingabe- und Rückgabeparameter

Da die Funktion im vorangegangenen Abschnitt keinerlei Parameter verwendet wird nun die allgemeine Syntax von Funktionen betrachtet.

```
function [out1, out2, ...] = functionname(in1, in2, ...)
  statements;
endfunction
```

Eine selbstgeschriebene Funktion muss immer mit dem Schlüsselwort `function` beginnen. Danach kommen die Rückgabeparameter die auch, wie oben, ganz weggelassen werden können. Wird nur ein Rückgabeparameter verwendet können die eckigen Klammern auch entfallen. Nun folgt der Name unter dem die Funktion später bekannt sein soll. In den runden Klammern nach dem Namen werden die Eingabeparameter definiert. Die Namen der Parameter können wie immer frei gewählt werden. Eine Funktion endet immer mit dem Schlüsselwort `endfunction`. Innerhalb der Funktion können die Eingabeparameter wie jede andere Variable verwendet werden. Die Rückgabeparameter werden gesetzt indem man ihnen einen Wert zuweist.

Bemerkung 9. Scilab *erlaubt auch die Verwendung einer variablen Anzahl von Eingabe- und Rückgabeparametern. Dazu muss das letzte Element der Eingabeparameter* `varargin` *lauten, das der Rückgabeparameter* `varargout`*.*

Um die Verwendung von Eingabe- und Rückgabeparameter zu illustrieren wird nun die Fakultät betrachtet. Eine rekursive Definition dieser Funktion lautet:

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

für alle $n \in \mathbb{N}$. Aus dieser Definition lässt sich der Programmcode für eine Funktion `myfactorial` sehr leicht ableiten. Er lautet dann:

```
function f = my_factorial( n )
  if n==0 then
```

```
    f = 1;
  else
    f = n * my_factorial( n-1 );
  end
endfunction
```

Es ist gut zu sehen, dass die Funktion den Eingabeparameter `n` sowie den Rückgabeparameter `f` verwendet. Falls `n` ungleich 0 ist, ruft sich die Funktion wieder selbst auf und der Rückgabeparameter wird in der Berechnung verwendet. Ein Vergleich mit der in SCILAB eingebauten Fakultätsfunktion `factorial(n)` zeigt, das beide Funktionen den gleichen Wert zurückliefern.

```
-->factorial(5)
 ans  =

    120.

-->my_factorial(5)
 ans  =

    120.
```

Zum Abschluss folgt nun noch eine Funktion die zwei Rückgabeparameter verwendet um den Zugriff darauf zu verdeutlichen.

```
function [A, B] = out_test()

  A = [ 1 2 3 ];
  B = 0;

endfunction
```

Diese Funktion macht nichts weiter als den Wert der beiden Rückgabeparameter zu setzten. Um nun beide Rückgabewerte zu erhalten müssen auch beim Aufruf beide Werte an Variablen zugewiesen werden. Ansonsten wird nur der erste Wert zurückgegeben.

```
-->[X, Y] = out_test()
 Y  =

    0.
 X  =

    1.    2.    3.

-->out_test()
 ans  =

    1.    2.    3.
```

3.2.2 Geschachtelte Funktionen

Funktionen in SCILAB können beliebig geschachtelt werden (nested functions). Diese Funktionen sind nach aussen hin nicht sichtbar, sondern nur in der jeweils

umgebenen Funktion. Ebenso sind die darin verwendeten Variablen lokal und nur innerhalb der geschachtelten Funktion sichtbar. Zusätzlich muss die Definition vor dem ersten Aufruf erfolgen. Das folgende, einfache Beispiel verdeutlicht dies:

```
function y = nested()

  function x = inner()
    x = 5;
  endfunction

  y = 5*inner();

endfunction
```

3.2.3 inline-Funktionen

Mit Hilfe des `deff` Befehls ist es in Scilab möglich einfache Funktionen ohne den Umweg über eine eigene Datei zu definieren. Dies ist speziell dann interessant, wenn für einmalige Anwendungen eine Funktion definiert werden muss, die durch einfache Scilab Ausdrücke darstellbar ist. Solch eine Funktion kann dann genau wie jede andere Funktion auch verwendet werden. Als Beispiel soll die Funktion $x \mapsto x^2 - 3$ dienen.

```
-->deff('y=whatever(x)', 'y=x.^2-3')

-->whatever(2)
 ans  =

    1.
```

3.2.4 Funktionen als Parameter

Genauso wie andere Werte lassen sich auch Funktionen in Variablen speichern wie folgendes Beispiel zeigt, das die Sinusfunktion in einer neuen Variable `mysin` speichert.

```
-->mysin = sin
 mysin  =


-->mysin(%pi/2)
 ans  =

    1.
```

Das ermöglicht es auch Funktionen als Eingabeparameter für andere Funktionen zu verwenden. Das ist besonders nützlich um z.B. mathematische Routinen für beliebige Funktionen zu definieren. Als Beispiel soll das diskrete Minimum und Maximum für eine beliebige Funktion auf einem bestimmten Intervall gesucht werden. Dazu wird zunächst eine neue Funktion betrachtet, die als Eingabeparameter die Intervallgrenzen `a` und `b`, die Feinheit der Diskretisierung dieses

Intervalls `s`, sowie die Funktion `f` die auf diesem Intervall ausgewertet werden soll, besitzt.

```
function [mi,ma] = my_minmax(a,b,s,f)

  //Erzeugen des Intervalls
  X = a:(b-a)/s:b;
  //Auswerten der Funktion auf dem Intervall
  Y = f(X);
  //Minimum
  mi = min(Y);
  //Maximum
  ma = max(Y);

endfunction
```

Der Aufruf von `my_minmax` kann dann für beliebige Funktionen erfolgen:

```
-->[mi, ma] = my_minmax(0, 10, 1000, sin)
 ma  =

    0.9999997
 mi  =

  - 0.9999971

-->[mi, ma] = my_minmax(0, 10, 100000, sin)
 ma  =

    1.
 mi  =

  - 1.

-->deff('y=quadrat(x)','y=x.^2')

-->[mi, ma] = my_minmax(0, 10, 1000, quadrat)
 ma  =

    100.
 mi  =

    0.
```

3.3 Fehlerbehandlung

Tritt während der Ausführung eines Scripts oder einer Funktion ein unerwarteter Fehler auf so wird die Ausführung an dieser Stelle gestoppt und der nachfolgende Programmcode wird nicht ausgeführt. In vielen Fällen ist es jedoch wünschenswert, das das Programm trotzdem fortfährt und zum Beispiel mit

dem nächsten Datensatz weiterarbeitet. Ein Exception-Handling wie in anderen Programmiersprachen ist auch in SCILAB möglich. Dazu muss der Programmcode, bei dem ein Fehler auftreten kann, von einem try-catch-Block umgeben werden:

```
try
  statements;
catch
  statements;
end
```

Der Programmcode im catch-Block wird nur ausgeführt falls im try-Block ein Fehler fliegt, ansonsten wird dieser Abschnitt übersprungen und nur die Befehle im try-Block interpretiert. Informationen über den aufgetretenen Fehler kann man mit den Befehlen aus Tabelle 3.1 abrufen. Eine genaue Beschreibung der Funktionen sowie eine Tabelle der Fehlercodes findet sich in der Hilfe. Die folgende Beispielsfunktion demonstriert die Fehlerbehandlung in SCILAB. Sie nimmt zwei Zahlen als Eingabeparameter und dividiert diese. Falls das zweite Argument 0 ist wird selbstverständlich ein Fehler erzeugt.

```
function x = fehler_test(a, b)
  x=0;
  try
    x = a/b;
  catch
    disp('Ein Fehler trat auf !');
    disp(lasterror());
  end
endfunction
```

Ein Aufruf der Funktion für 1 und 0 ergibt dann:

```
-->fehler_test(1,0)

 Ein Fehler trat auf !

 Division by zero...
 ans  =

    0.
```

Es können auch eigene Fehlermeldungen erzeugt werden. Dazu muss der Befehl `error(msg)` genutzt werden. Warnungen können mit `warning(msg)` ausgegeben werden.

```
-->error('Mein Fehler');
                      !--error 10000
Mein Fehler



-->lasterror
```

`iserror()`	Testet ob ein Fehler aufgetreten ist
`laterror()`	Gibt den letzten Fehler zurück der aufgetreten ist
`errclear()`	Leert den Fehlerspeicher
`errcatch()`	Kann auch zum Fehler fangen verwendet werden

Tabelle 3.1: Funktionen zur Fehlerbehandlung

```
 ans  =

 Mein Fehler


-->warning('Achtung!')
WARNING: Achtung!
```

3.4 Debugging

Bei der Fehlersuche in einem Programm ist es oftmals wünschenswert den Programmcode Zeile für Zeile auszuführen und den Inhalt der Variablen zu betrachten. Auch in Scilab ist debugging möglich. Dazu müssen Breakpoints im Quelltext gesetzt werden. An diesen Stellen wird die Ausführung des Programms später gestoppt und der Nutzer hat weitere Möglichkeiten zur Interaktion.

In Scilab werden Breakpoints mit dem `pause` Befehl gesetzt. Um dies zu illustrieren wird noch einmal die Fakultätsfunktion aus Abschnitt 3.2.1 betrachtet und entsprechende Breakpoints gesetzt.

```
function f = my_factorial( n )
  if n==0 then
    pause
    f = 1;
  else
    pause
    f = n * my_factorial( n-1 );
  end
endfunction
```

Der Aufruf der Funktion wird nun immer bei den `pause` Befehlen stoppen. Nach dem Anhalten erscheint ein neue Eingabeaufforderung die nun eine Nummer trägt. Mit `resume` kann die ausführung nun fortgesetzt werden. Mit `abort` wird sie abgebrochen. Durch Angabe des Variablennamens wird der Inhalt der Variablen angezeigt. Ein Aufruf der Funktion `my_factorial(2)` kann dann wie folgt aussehen:

```
-->my_factorial(2)


-1->n
 n  =
```

2.

-1->resume

-1->disp(n)

1.

-1->abort

Kapitel 4

Schleifen und Verzweigungen

In diesem Kapitel werden die wichtigsten Konstrukte zur Steuerung von Programmabläufen erläutert. Der erste Abschnitt behandelt bedingte Verzweigungen im Programm und im zweiten Abschnitt werden Schleifen zur Wiederholung von Programmabschnitten diskutiert.

4.1 Verzweigungen

4.1.1 *if-then-else*

Die *if-then-else* Verzweigung testet einen boolschen Ausdruck und führt dann, entsprechend ob die Aussage `true` oder `false` ist, einen Anweisungsblock aus. Die allgmeine Syntax für if-else Verzweigungen in SCILAB ist wie folgt:

```
if condition then
  statements;
elseif condition then
  statements;
else
  statements;
end
```

Hierbei steht `condition` für einen beliebigen boolschen Ausdruck, also z.B. `1<2`, eine boolsche Variable oder eine Funktion die einen boolschen Wert zurück gibt. Für `statements` kann ein oder mehrere SCILAB-Befehle stehen. Die Anweisungen `elseif` und `else` sind optional, wobei `elseif` auch mehrfach verwendet werden darf. Die Verzweigung `else` darf jedoch höchstens einmal verwendet werden und wird dann betreten, falls alle anderen Konditionen nicht erfüllt sind, also `false` ergeben. Verweigungen können auch beliebig verschachtelt werden, was durch das nun folgende Beispiel verdeutlicht wird.

```
if a == 1 then
  disp('a ist gleich 1');
elseif a > 1 then
```

```
  if a > 2 & a < 3 then
    disp('a ist groesser 2 und kleiner 3');
  else
    disp('a ist groesser 1 und kleiner gleich 2 oder groesser gleich 3');
  end
else
  disp('a ist kleiner 1');
end
```

4.1.2 *select case*

Die *select-case* Verzweigung ist hilfreich, wenn ein Ausdruck mehrere vordefinierte Werte annehmen kann. Die allgemeine Syntax lautet:

```
select expression
case exp1 then
  statements;
case exp2 then
  statements;
...
else
  statements;
end
```

In diesem Fall steht `expression` für einen beliebigen Ausdruck, z.B eine Variable oder einen Funktionsaufruf mit Rückgabewert. Entsprechend stehen `exp1`, `exp2`, usw. für Werte die dieser Ausdruck annehmen kann. Das `else` ist wieder optional. Die Eigenschaft, das SCILAB eine ungetypte Programmiersprache ist kommt an dieser Stelle besonders zum tragen. Das wirklich auf jeden beliebigen Wert geprüft werden kann, veranschaulicht das nächste Beispiel.

```
select a
case 1 then
  disp('a ist die Zahl 1');
case 'ABC' then
  disp('a ist der String ABC');
case [0 1;1 0] then
  disp('a ist die 2x2 Einehitsmatrix');
else
  disp('a ist irgendwas anderes');
end
```

4.2 Schleifen

4.2.1 *for*

Dieser Schleifentyp eignet sich besonders dann, wenn eine bestimmte Anzahl an Durchgängen benötigt wird. Im Allgemeinen wird eine *for*-Schleife mit dieser Syntax definiert:

```
for i=values
```

```
  statements;
end
```

Die Variable `i` wird als Laufvariable bezeichnet und ist innerhalb der Schleife sichtbar, das heißt sie kann nur dort verwendet werden. Mit `values` werden die Werte definiert, die die Laufvariable annehmen kann. Im einfachsten Fall ist die Laufvariable `i` eine Zahl. Dazu muss `values` als Vektor definiert sein:

```
for i=[1 2 5 8]
  disp(i);
end
```

Die Laufvariable `i` nimmt in diesem Fall also die Werte $1, 2, 5, 8$ an. Aber auch komplexere Werte für die Laufvariable sind denkbar. Wird `values` beispielsweise als Matrix definiert, so nimmt die Laufvariable jeweils eine Zeile dieser Matrix als Wert an:

```
for m=[1 0; 0 1]
  disp(m);
end
```

In diesem Beispiel wird die Schleife zweimal durchlaufen und `m` nimmt die Werte `[1 0]` und `[0 1]` an. Fü Listen kann zusätzlich über alle Einträge iteriert werden. Für eine Liste `L` ist dann folgendes gültig:

```
for e=L
  disp(e);
end
```

4.2.2 *while*

Im Gegensatz zur *for*-Schleife gibt es bei der *while*-Schleife keine vorbestimmte Anzahl an Durchläufen. Stattdessen testet *while* eine Bedingung auf `true` oder `false` und entscheidet dann, ob der Schleifenrumpf ausgeführt wird.

```
while condition
  statements;
end
```

Hierbei steht `condition` wieder für den boolschen Ausdruck der getestet werden soll.

Bemerkung 10. *Bei der while-Schleife ist es problemlos möglich Endlosschleifen zu programmieren.* Scilab *erkennt nicht, ob es sich bei einer Schleife um eine Endlosschleife handelt und wird den Schleifenrumpf unendlich oft ausführen. Um dennoch die Kontrolle über* Scilab *zurück zuerhalten kann mit der Tastenkombination* `Strg+x` *die Ausführung unterbrochen werden. Anschließend kann sie mit dem Befehl* `abort` *beendet werden.*

4.2.3 *break* und *continue*

Um noch mehr Kontrolle über die Ausführung von Schleifen zu haben kennt Scilab die Befehle `break` und `continue`. Beim `break` Befehl wird die Ausführung

der Schleife sofort beendet. Sind mehrere Schleifen inneinander verschachtelt, so wird die Schleife abgebrochen, in der der Befehl ausgeführt wird. Der `continue` Befehl veranlasst das Programm zum Beginn der Schleife zu springen. Der verbleibende Schleifenrumpf wird für diesen Durchgang dann übersprungen. Das nun folgende Beipiel soll noch einmal die Verwendung der beiden Schleifentypen und der Befehle `break` und `continue` verdeutlichen:

```
x = 5;

for i=0:5

  disp('i = ' + string(i));

  if i<5 then
    continue;
  end

  x = i;
  while %t

    x = x - 1;
    disp('x = ' + string(x));

    if x==0 then
      break;
    end

  end

end
```

Kapitel 5

Graphik

Die Visualisierung von Daten ist eine wichtige Aufgabe bei der Analyse von Daten und der Erstellung von Berichten. SCILAB beeinhaltet eine Vielzahl eingebauter Funktionen zur Erstellung von Funktionsgraphen, Charts und anderer Graphiken. Aus diesem Grund beschränkt sich dieses Kapitel auf eine Auswahl dieser Befehle.

5.1 Graphikfenster

Um einen ersten Eindruck der Graphikfunktionen zu erhalten, kann

```
-->surf()
```

in der Konsole eingegeben werden. Dieser Befehl ohne Argumente erzeugt einen 3D-Oberflächenplot mit Beispieldaten. Es öffnet sich ein neues Graphikfenster wie Abbildung 5.1 zeigt. Jedes Fenster trägt eine Nummer zur Identifizierung. Um ein leeres Graphikfenster zu erzeugen muss die Funktion

```
f = figure(num)
```

benutzt werden. Das Argument `num` gibt die Nummer des neuen Fensters an. Das neue Fenster wird dann automatisch zum aktiven Fenster mit dem dann gearbeitet werden kann. Mit `close` kann es entsprechend geschlossen werden. In der Standardeinstellung wird jeder neue Plot in das gerade aktive Graphikfenster gezeichnet, egal ob bereits ein Plot vorhanden ist. So können zum Beispiel mehrere Funktionsgraphen in ein Fenster gezeichnet werden. Manchmal kann es jedoch wünschenswert sein, das das aktive Fenster vorher geleert wird. Mit den folgenden beiden Befehlen kann der Standardwert geändert werden:

```
da=gda();
da.auto_clear = 'on'
```

Mit dem Wert `'off'` wird der Standard wieder hergestellt. Der Befehl `scf(num)` selektiert das aktive Fenster und `clf` leert das aktive Fenster. Zusätzlich kann mit

```
subplot(m,n,p)
```

das Fenster in $m \times n$ einzelne Graphiken aufgebrochen werden, wobei `p` das aktive Unterfenster angibt.

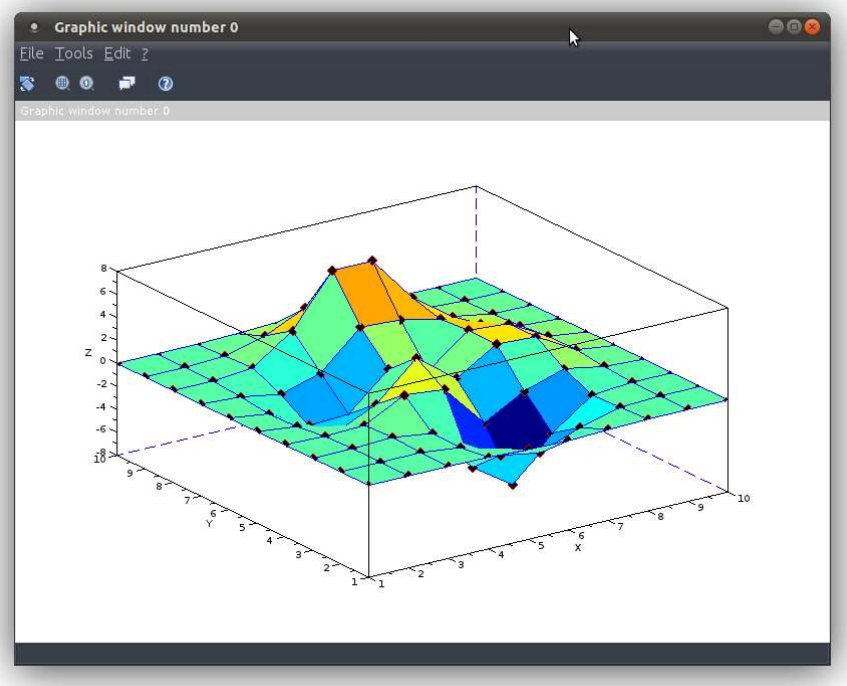

Abbildung 5.1: Das Graphikfenster

5.2 Der plot-Befehl

Der `plot` Befehl stellt den wohl am häufigsten verwendeten Befehl zur Erstellung von Grafiken in Scilab dar. Er erzeugt einen einfachen 2D-Funktionsgraphen nach Übergabe von entsprechenden x-y Wertepaaren. Zunächst einmal muss die genaue Syntax erläutert werden.

```
plot(Y)
plot(X,Y)
plot(X1,Y1,...,XN,YN)
```

Im ersten Fall nimmt die `plot` Funktion eine Zahl, einen Vektor oder eine Matrix `Y` als erstes Argument. Ist `Y` eine $n \times k$ Matrix, so werden k Linien der Form $i = 1, 2, \ldots, n \rightarrow Y_{i,j}$ für $j = 1, 2, \ldots k$ in einer Graphik dargestellt. Jeder dieser Graphen besteht aus geraden Verbindungsstücken zwischen zwei Werten. Falls `Y` ein Vektor ist, so wird dieser immer wie eine $n \times 1$ Matrix behandelt und falls `Y` nur eine Zahl ist, erhält man nur einen Punkt.

Im zweiten Fall werden mit `X` direkt die x-Werte und mit `Y` die y-Werte des Graphen angegeben. Hierbei muss `X` die gleiche Anzahl an Zeilen und Spalten haben wie `Y`. Die einzelnen Linien sind wiederum durch die Spalten definiert und werden angezeigt wie im ersten Fall.

Der dritte Fall erzeugt Linien für jedes `Xi-Yi` Wertepaar. Die Anzeige erfolgt wie in den ersten beiden Fällen. In Abbildung 5.2 ist der Graph für einen einfachen `plot`-Befehl dargestellt:

```
-->X = 0:0.1:2*%pi;

-->plot(X, sin(X), X, cos(X))
```

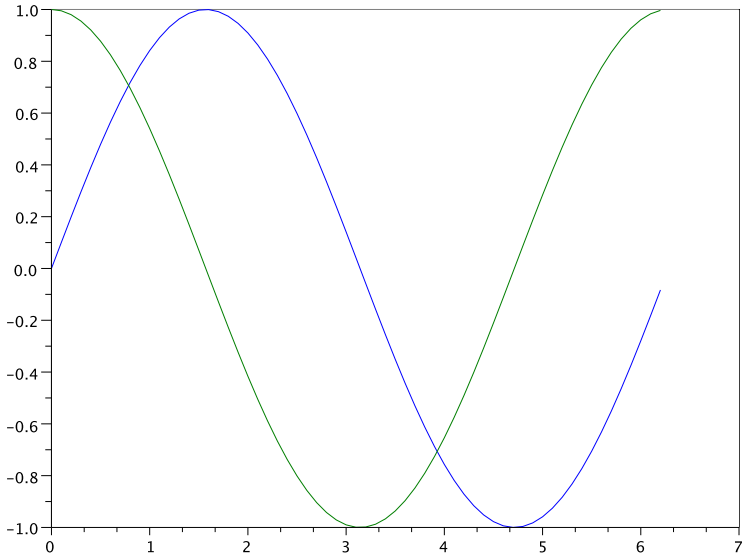

Abbildung 5.2: Ein einfacher Plot

Um das Aussehen der Linien genauer zu bestimmen können dem `plot` Befehl weitere Argumente übergeben werden:

```
plot(Y,<LineSpec>,<GlobalProperty>)
plot(X,Y,<LineSpec>,<GlobalProperty>)
plot(X1,Y1,<LineSpec1>,X2,Y2,<LineSpec2>,...
    XN,YN,<LineSpecN>,<GlobalProperty1>,<GlobalProperty2>,...
    <GlobalPropertyM>)
```

Das Argument `<LineSpec>` ist optional und beschreibt die Art und Weise wie die zuletzt erzeugten Linien des Graphen angezeigt werden. Die Angabe der Eigenschaft besteht dabei aus einem String, der verschiedene Attribute setzt. Es ist möglich den Stil der Linie (siehe Tabelle 5.1), die Farbe der Linie (siehe Tabelle 5.2) sowie den Typ der Marker (siehe Tabelle 5.3) zu ändern.

Das Argument `<GlobalProperty>` ist ebenfalls optional und setzt globale Eigenschaften für alle Linien in der Graphik. Eventuell vorher gesetzte `<LineSpec>` Attribute werden dabei überschrieben. Die Übergabe der Eigenschaften erfolgt als Schlüssel-Wert Paar. Eine genaue Beschreibung dieser Paare kann in der Hilfe gefunden werden. Im Nachfolgenden Beispiel wird zum Beispiel die Linienstärke verändert. Das Ergebnis wird in Abbildung 5.3 gezeigt.

```
-->X = -5:0.1:5;
```

```
-->plot(X, exp(X), 'b', 'LineWidth', 3)

-->plot(X, X.^3, 'r--', X, X.^2, 'k:.', 'markersize', 2)
```

`-`	eine durchgezogene Linie
`--`	eine gestrichelte Linie
`:`	eine gepunktete Linie
`-.`	eine strichpunktierte Linie

Tabelle 5.1: Verschiedene Linienarten

`r`	rot	`g`	grün
`b`	blau	`c`	cyan
`m`	magenta	`y`	gelb
`k`	schwarz	`w`	weiß

Tabelle 5.2: Verschiedene Linienfarben

`+`	Pluszeichen
`o`	Kreis
`*`	Stern
`.`	Punkt
`x`	Kreuz
`'square'` oder `'s'`	Quadrat
`'diamond'` oder `'d'`	Diamand
`o`	Kreis
`^`	Dreieck nach oben
`v`	Dreieck nach unten
`>`	Dreieck nach rechts
`<`	Dreieck nach links
`'pentagram'`	Pentagramm
`'none'`	nichts (default)

Tabelle 5.3: Verschiedene Marker

5.3 Weitere plot-Befehle

Scilab kennt noch viele weitere Befehle um Graphiken zu erzeugen. Oftmals verwenden diese die gleiche Syntax wie der `plot` Befehl. Manchmal unterscheiden sie sich aber auch. Eine genaue Beschreibung aller Befehle ist hier natürlich nicht möglich. Der Vollständigkeit wegen sind jedoch in Tabelle 5.4 weitere Funktionen abgedruckt.

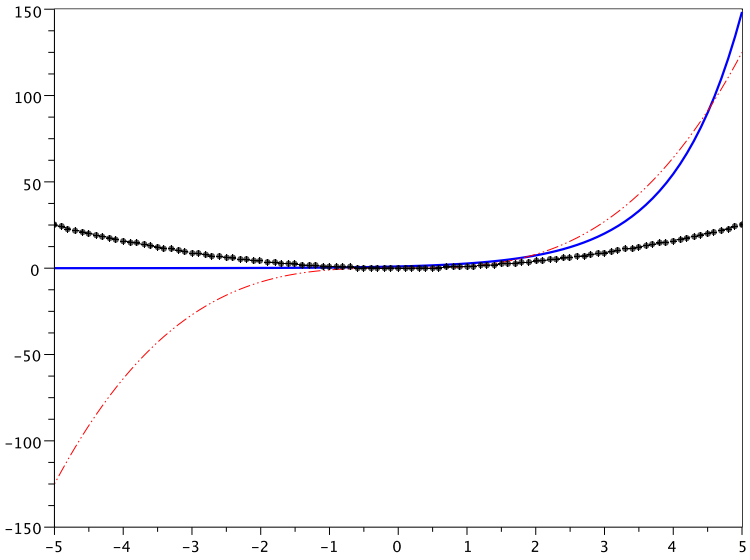

Abbildung 5.3: Ein weiterer Plot

`plot2d`	2D-Plot, ähnlich wie `plot`
`plot3d`	3D-Plot einer Oberfläche
`plot3df`	3D-Plot einer Oberfläche definiert durch eine Funktion
`surf`	3D-Plot einer Oberfläche
`contour`	Höhenlinien einer Oberfäche
`bar`	Barchart
`pie`	Piechart

Tabelle 5.4: Weitere plot-Befehle

5.4 Beschriftung und Legende

Um erstellte Graphiken weiter anzupassen, bietet SCILAB eine Vielzahl von Befehlen, um den Titel, die Achsenbeschriftungen, die Legende sowie einfache Beschriftungen zu ändern. Nachfolgend ein grober Überblick über einige Befehle.

- `title(my_title)`: setzt die Überschrift der Graphik
- `xlabel(x_label)`: setzt die Beschriftung der x-Achse
- `ylabel(y_label)`: setzt die Beschriftung der y-Achse
- `zlabel(z_label)`: setzt die Beschriftung der z-Achse
- `xtitle(title,[x_label,[y_label,[z_label]]])`: setzt alle Beschriftungen

- `xstring(x,y,str)`: setzt einen Text an die Stelle `x`, `y`
- `legend(string1,string2, ...)`: fügt eine Legende in der Reihenfolge der erzeugten Linien hinzu
- `colorbar(umin, umax)`: stellt eine Farbtafel mit Skalierung auf

Seit Version 5.2 erlaubt Scilab die Verwendung von LaTeX oder MathML in den meisten obigen Befehlen. Somit können auch mathematische Formulierungen und Symbole einfach mit in die Graphik aufgenommen werden. Das Ändern von weiteren Eigenschaften wie z.B. Schriftgröße und Schriftfarbe ist etwas aufwendiger und wird nur kurz im abschließenden Beispiel dieses Abschnitts demonstriert. Für eine detailierte Beschreibung sei wieder auf die Hilfe verwiesen.

Nun noch ein ausführliche Beispiel um die in diesem Abschnitt vorgestellten Befehle zu illustrieren. Das Ergebnis des Scripts wird in Abbildung 5.4 gezeigt.

```
//neue Graphik
figure
//Die x-Werte
X = 0:0.1:2*%pi;
//Der erste Graph - sin(X)
//Farbe rot, gestrichelt, Linienstaerke 2
plot(X, sin(X), 'r--', 'LineWidth', 2)
//Der zweite Graph - cos(X)
//Farbe schwarz, durchgezogen, Linienstaerke 1
plot(X, cos(X), 'k')
//Hinzufuegen des Titels
title('Sinus und Kosinus')
//Beschriftung der Achsen, mit Latex
xlabel("$x$")
ylabel("$y$")
//Legende mit Latex
legend("$x \mapsto \sin(x)$", "$x \mapsto \cos(x)$")
//Textgroesse der Ueberschrift aendern
a=gca();
a.title.font_size = 6;
```

5.5 Export

Der Export von Graphiken macht deren Wiederverwendung sehr einfach. Scilab bietet mehrere Formate in die eine Graphik exportiert werden kann. In Tabelle 5.5 sind die verschiedene Exportbefehle sowie deren Ausgabeformate aufgelistet. Grundsätzlich sollte ein vektorielles Dateiformat gewählt werden um Qualitätsverluste zu vermeiden. Die Benutzung dieser Befehle folgt immer dem gleichen Muster:

```
xs2pdf(num, filename)
```

Hierbei gibt `num` wieder die Nummer des Fensters an und `filename` den Dateinamen. Alternativ können Graphiken auch über das Menü `File->Export to...` im Grafikfenster exportiert werden was wohl den leichtesten Weg dastellt.

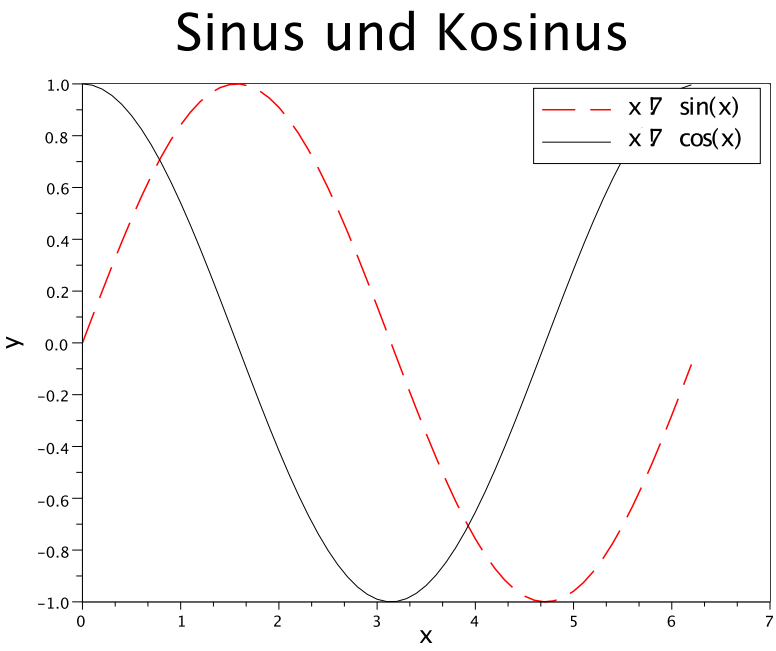

Abbildung 5.4: Ein ausführliches Beispiel

vektoriell	
`xs2png`	exportiert nach PNG
`xs2pdf`	exportiert nach PDF
`xs2svg`	exportiert nach SVG
`xs2eps`	exportiert nach Encapsulated Postscript
`xs2ps`	exportiert nach Postscript
`xs2emf`	exportiert nach EMF (nur Windows)
Bitmap	
`xs2fig`	exportiert nach FIG
`xs2gif`	exportiert nach GIF
`xs2jpg`	exportiert nach JPG
`xs2bmp`	exportiert nach BMP
`xs2ppm`	exportiert nach PPM

Tabelle 5.5: Funktionen zum Export von Graphiken

Kapitel 6

Eingabe und Ausgabe

6.1 Speichern und Lesen von Variablen

Für das Speichern und Lesen von Variablen stellt SCILAB zwei sehr komfortable Funktionen bereits. Zum einen `save` zum Speichern und `load` zum Lesen. Die allgemeine Syntax der beiden Funktionen lautet

```
save(filename [,x1,x2,...,xn])
load(filename [,x1,x2,...,xn])
```

wobei `filename` für den Namen der Datei steht in die geschrieben oder von der gelesen werden soll. Die optionalen Parameter `x1,...,xn` stehen für die Variablen. Werden diese weggelassen, so werden alle momentan verfügbaren Variablen in der Datei gespeichert bzw. alle Variablen aus der Datei gelesen. Das folgende Beispiel erzeugt zwei Variablen `A` und `b` und speichert diese in einer Datei `test.dat`. Anschliessend werden diese Variablen wieder geladen.

```
-->A = [1 2 3 4 5; 6 7 8 9 10];

-->b = [1 2];

-->save('test.dat', A, b);

-->clear

-->A
  !--error 4
Undefined variable: A


-->load('test.dat', 'A', 'b');

-->A
 A  =

    1.    2.    3.    4.    5.
    6.    7.    8.    9.    10.
```

Bemerkung 11. *Da die Variablen vor dem Laden noch nicht zur Verfügung steht, ist es wichtig die Variablennamen in einfache Anführungszeichen zu sezten, also als String zu übergeben. Ansonsten wirft* Scilab *einen Fehler, da die angefragten Variablen noch nicht vorhanden sind.*

Bemerkung 12. *Bei der Benutzung von* `save` *werden die Daten binär in die Datei geschrieben. Eine anschliessende Bearbeitung mit Hilfe eines Texteditors ist nicht möglich und zersört im schlimmsten Fall den Inhalt der Datei.*

6.2 Unformatierte Ausgabe

Um Strings und Variablen ohne ein spezifisches Format auszugeben, kann die Funktion

```
print(unit,x1,[x2,...xn])
```

verwendet werden. Die Besonderheit an dieser Funktion ist, das die Ausgabe sowohl in einer Datei als auch auf dem Bildschirm erfolgen kann. Entscheidend hierfür ist der Wert der für `unit` übergeben wird. Wird ein Dateiname angegeben, so werden die Variablen `x1,...,xn`, im Gegensatz zu `save`, als ASCII Text in diese Datei geschrieben. Wird für `unit` die Konstante `%io(2)` oder `6` angegeben so erfolgt die Ausgabe auf dem Bildschirm.

```
-->A = [1 2 3 4 5; 6 7 8 9 10];

-->b = [1 2];

-->print(%io(2), A, b)
 b  =

    1.    2.
 A  =

    1.    2.    3.    4.    5.
    6.    7.    8.    9.    10.

-->print('test.txt', A, b)
```

Bemerkung 13. *Genau wie bei* `disp`*, wird bei* `print` *die letzte Variable zuerst geschrieben.*

6.3 Arbeiten mit Dateien

Das folgende Kommando erlaubt das öffnen einer Datei:

```
[unit [,err]]=file('open', file-name [,status] [,access [,recl]] [,format])
```

Die möglichen Parameter dieser Funktion sind folgende:

`file-name`: String, der Name der Datei die geöffnet werden soll

`status`: String, der Status der Datei die geöffnet werden soll
`'new'`: Datei muss nicht existieren, eine neue Datei (default)
`'old'`: Datei muss existieren
`'unknown'`: unbekannter Status
`'scratch'`: Datei wird nach Beenden der Session gelöscht

`access`: String, der Typ des Zugriffs auf die Datei
`'sequential'`: sequentieller Zugriff (default)
`'direct'`: direkter Zugriff

`format`: String
`'formatted'`: eine formatierte Datei (default)
`'unformatted'`: binäres Format

`recl`: Integer, die Länge der Einträge in Bytes
wenn `access='direct'`

`unit`: Integer, der Filedescriptor der geöffneten Datei

`err`: Integer, Fehlernummer falls das öffnen fehlschlägt. Falls `err` weggelassen wird, wird statt dessen eine Fehlermeldung ausgegeben.

Ist eine Datei geöffnet und man kennt den Filedescriptor kann das Kommando auch als `file(action, unit)` verwendet werden. Hierbei steht `action` für einen der folgenden Strings:

`'close'`: schliesst die Datei

`'rewind'`: setzt den Zeiger auf den Anfang der Datei

`'backspace'`: setzt den Zeiger auf den Anfang des letzten Eintrages

`'last'`: setzt den Zeiger ans Ende des letzten Eintrages

6.3.1 Schreiben

In diesem Abschnitt wird die `write` Funktion beschrieben. Sie dient hauptsächlich dazu Matrizen zu schreiben, eine nach der anderen. Deshalb ist es oftmals besser seine Daten vorher zu einer Matrix zusammenzufügen und dann nur diese zu schreiben. Zuerst wieder die allgemeine Syntax:

```
write(unit,a,[format])
```

Der Filedescriptor `unit` muss auf eine geöffnete Datei zeigen und `a` steht für die Matrix oder einen String der geschrieben werden soll. Der optionale Parameter `format` ist ein String der die Formatierung in einem FORTRAN Format angibt.

Eine genaue Beschreibung dieses Formats wird hier nicht gegeben. Statt dessen werden einige Beispiele zur Formatierung angegeben.

```
-->A = rand(2,3);

-->B = rand(2,3);

-->C = A+B;

-->u = file('open', '/home/math/becker/Desktop/data.txt', 'new');

-->write(u,'Matrix A', '(a)');

-->write(u,A, '(3f10.6)');

-->write(u,'Matrix B', '(a)');

-->write(u,B, '(3(f10.6,2x))');

-->write(u,'Matrix C', '(a)');

-->write(u,C, '(3(f10.6,3x))');

-->file('close', u);
```

Der Inhalt der Datei sieht dann folgendermaßen aus:

```
Matrix A
  0.726351  0.544257  0.231224
  0.198514  0.232075  0.216463
Matrix B
  0.883389    0.307609    0.214601
  0.652513    0.932962    0.312642
Matrix C
  1.609739     0.851866     0.445825
  0.851028     1.165036     0.529105
```

Bemerkung 14. *Erst nach dem Schliessen der Datei werden die Daten physikalisch geschrieben. Wird die Datei nicht geschlossen, bleibt sie leer.*

6.3.2 Lesen

Der Gegenpart zum `write` Befehl ist das Kommando

```
[x] = read(unit,m,n,[format])
```

Das `x` steht für den Namen der Variablen in die gelesen werden soll, `unit` ist wieder der Filedescriptor der geöffneten Datei. Die beiden Zahlen `m` und `n` geben die Anzahl der Zeilen bzw. Spalten an, die von der Matrix eingelesen werden sollen. Das `format` beschreibt wieder das Format der zu lesenden Daten. Angenommen es existiert eine Datei `data.txt` mit folgendem Inhalt:

```
   1.0        2.0        3.0
   4.0        5.0        6.0
   7.0        8.0        9.0
   10.0       11.0       12.0
```

Falls die Anzahl der Zeilen bekannt ist (`n=4` hier) kann die Matrix mit folgenden Befehlen eingelesen werden:

```
-->u = file('open', '/home/math/becker/Desktop/data.txt', 'old');

-->B = read(u,4,3);

-->B
 B  =

    1.     2.     3.
    4.     5.     6.
    7.     8.     9.
    10.    11.    12.

-->file('close', u);
```

Falls die Anzahl der Zeilen nicht bekannt ist, so kann `n=-1` gesetzt werden. Damit wird die Datei bis zum Ende gelesen. Die Anzahl der Spalten muss in jedem Fall angegeben werden.

6.4 Lesen von der Tastatur

Zum Abschluss dieses Kapitels wird noch das Lesen von Tastatureingaben behandelt. Das geschied ebenfalls mit dem `read` Befehl aus dem vorangegangen Abschnitt. Der Filedescriptor muss in diesem Fall auf `%io(1)` gesetzt werden und `n` sowie `m` sollte jeweils auf `1` gesetzt werden um einzelne Werte einzulesen. Andernfalls ist die Eingabe recht kompliziert. Die folgende Funktion liest zwei Zahlen von der Tastatur, addiert diese und gibt das Ergebnis aus:

```
function my_keyboard_add()

  disp('Gib die erste Zahl ein:');
  x = read(%io(1), 1, 1);
  disp('Die eingegebene Zahl war: ' + string(x));
  disp('Gib die zweite Zahl ein:');
  y = read(%io(1), 1, 1);
  disp('Die eingegebene Zahl war: ' + string(y));
  z=x+y;
  disp('Die Summe von ' + string(x) + ' und '...
             +  string(y) +' ist ' + string(z));

endfunction
```

Die Ausführung ergibt dann folgende Ausgabe:

```
-->my_keyboard_add()

 Gib die erste Zahl ein:
-->2

 Die eingegebene Zahl war: 2

 Gib die zweite Zahl ein:
-->3

 Die eingegebene Zahl war: 3

 Die Summe von 2 und 3 ist 5
```

Kapitel 7

Numerische Verfahren

In diesem Kapitel werden nur einige der in SCILAB vorhandenen numerischen Verfahren vorgestellt. Eine genaue Beschreibung der Algorithmen bleibt Vorlesungen der Numerik vorbehalten.

7.1 Methoden der linearen Algebra

7.1.1 Matrix Zerlegungen

Die nachfolgenden Zerlegungen sind wichtiger Grundbestandteil der numerischen linearen Algebra. Diese Verfahren werden z.B. in Demmel [1], Golub und van Loan [2] und Watkins [4] ausführlich besprochen.

- `lu` LR-Zerlegung: Mittels der Gauß-Elimination mit Zeilenpivotierung werden Matrizen L,U und P erzeugt mit $LU = PA$. Um alle drei Matrizen zu erhalten, ist der Befehl `[L U P] = lu(A)` zu verwenden.
- `qr` QR-Zerlegung: Die QR-Zerlegung liefert eine Zerlegung beliebiger Matrizen in eine orthogonale Matrix Q und eine obere Dreiecksmatrix R. Diese Operation ist zum Beispiel hilfreich beim Lösen von kleinste-Quadrate-Aufgaben.
- `chol` Cholesky-Zerlegung: Die Cholesky-Zerlegung ist ein Spezialfall der LR-Zerlegung für symmetrische positiv definite Matrizen. Dies spart viel Rechenaufwand und ist numerisch stabiler.
- `spec` Eigenwert-Zerlegung: Berechnet die Jordan-Zerlegung einer Matrix. Ein weiteres Verfahren ist auch `schur` zur Berechnung der Schur-Form einer Matrix.
- `svd` Singularwert-Zerlegung: Die Singularwerte einer Matrix A sind die Quadratwurzeln der Eigenwerte von $A^T A$. Allerdings wird die Berechnung anders durchgeführt.

Hier noch eine Beispiel einer LU-Zerlegung:

```
-->A = [ 2 8 17; 4 10 4; 4 4 -8];
```

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

    0.    1.    0.
    0.    0.    1.
    1.    0.    0.
 U  =

    4.    10.    4.
    0.  - 6.   - 12.
    0.    0.     9.
 L  =

    1.     0.     0.
    1.     1.     0.
    0.5  - 0.5    1.
```

7.1.2 Iterative Verfahren zum Lösen von linearen Gleichungssystemen

Wie bereits in einem früheren Kapitel beschrieben, treten in manchen Anwendungen sehr große und dünnbesetzte Matrizen auf. Für solche Matrizen sind die Lösungsoperatoren für Gleichungen `/` und `\` meist nicht sehr effizient. Statt dessen werden iterative Methoden verwendet. Auch die nachfolgende Liste ist sehr unvollständig. Neben der Hilfe ist auch das Buch von Youcef Saad [3] sehr nützlich.

- `pcg` Preconditioned Conjugate Gradient: Eines der besten iterativen Verfahren zum Lösen dünnbesetzter linearer Gleichungssysteme, wenn die Matrix positiv definit ist.
- `gmres` Generalized Minimum Residual: Zum Lösen beliebiger nicht singulärer Gleichungssysteme.
- `gmr` Quasi Minimum Residual: Zum Lösen indefiniter, nicht singulärer, symmetrischer Gleichungssysteme.

7.2 Interpolation

Scilab bietet auch verschiedene Methoden zur Interpolation von Funktionen.

- `interpln`: lineare Interpolation
- `linear_interpn`: n-dimensionale lineare Interpolation
- `splin`: Kubische Splines
- `splin2d`: Bikubische 2D-Splines
- `splin3d`: 3D-Splines
- `interp`: Auswerten von Splines

7.3 Quadratur

Die numerische Berechnung eines bestimmten Integrals wird als Quadratur bezeichnet. Auch hierfür stellt SCILAB effiziente Methoden bereit.

- `integrate`: Berechnet das Integral einer Funktion.
- `intg`: Ähnlich wie `integrate` mit anderer Syntax.
- `int2d`: Zweidimensionale Integration.
- `int3d`: Dreidimensionale Integration.
- `inttrap`: Integration über Messwerte mittels Trapezoid-Interpolation.
- `intsplin`: Integration über Messwerte mittels Spline-Interpolation.

```
-->integrate('sin(x)', 'x', 0, %pi)
 ans  =

    2.

-->deff('y=my_sin(x)','y=sin(x)');

-->intg(0, %pi, my_sin)
 ans  =

    2.
```

7.4 Stochastik

Es stehen auch viele Hilfsmittel aus der Stochastik zur Verfügung:

- `rand`: Liefert gleichverteilte oder normalverteilte Zufallsvariablen.
- `mean`: Berechnet den Erwartungswert.
- `st_deviation`: Berechnet die Standardabweichung.
- `variance`: Berechnet die Varianz.
- `corr`: Berechnet die Korrelation und Kovarianz.

```
-->rand(4,1,'uniform')
 ans  =

    0.2164633
    0.8833888
    0.6525135
    0.3076091

-->rand(4,1,'normal')
 ans  =
```

```
  - 1.4061926
  - 1.0384733
  - 1.7350313
    0.5546874
```

7.5 Weitere Methoden

Zum Abschluß dieses Kapitel noch einige weitere Methoden die in Scilab implementiert sind:

- `fsolve`: Nullstellensuche für eine gegebene Funktion.
- `fminsearch`: Mehrdimensionale Minimierung.
- `ode`: Solver für gewöhnliche Differentialgleichungen.
- `fft`: Fast Fourier Transformation.
- `ifft`: Inverse Fast Fourier Transformation.

```
-->deff('y=f(x)','y=exp(x)-exp(1)');

-->fsolve(0, f)
 ans  =

    1.
```

Literaturverzeichnis

[1] Demmel, J. W. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[2] Golub, G. H., and Van Loan, C. F. *Matrix computations*, third ed. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 1996.

[3] Saad, Y. *Iterative methods for sparse linear systems*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2003.

[4] Watkins, D. S. *Fundamentals of matrix computations*, third ed. Pure and Applied Mathematics (Hoboken). John Wiley & Sons Inc., Hoboken, NJ, 2010.