Eine Einführung in Scilab

version 0.999

Bruno Pinçon

Institut Elie Cartan Nancy
E.S.I.A.L.
Université Henri Poincaré
Email: `Bruno.Pincon@iecn.u-nancy.fr`

Übersetzung: Agnes Mainka, Helmut Jarausch
IGPM, RWTH Aachen

Diese Einführung wurde ursprünglich für die Ingenieurstudenten der E.S.I.A.L. (École Supérieure d'Informatique et Application de Lorraine) geschrieben. Sie beschreibt nur einen kleinen Teil der Möglichkeiten von Scilab, im Wesentlichen die Teile, die für die Einführung in die Numerik, die ich unterrichte, gebraucht werden; dies sind:

- Der Umgang mit Matrizen, Vektoren und Gleitkommazahlen
- Programmieren in Scilab
- einfache graphische Ausgaben
- einige dazu nötige wichtige Funktionen (Erzeugung von Zufallszahlen, Lösung von Gleichungen, ...)

Scilab eröffnet viel mehr Möglichkeiten, insbesondere in der optimalen Steuerung, der Signalverarbeitung, der Simulation dynamischer Systeme (mit scicos) usw. Da ich plane, diese Einführung zu vervollständigen, bin ich offen für jederlei Bemerkungen, Vorschläge und Kritik, die zu einer Verbesserung führen; schicken Sie mir diese bitte per Email.

Für diese neue Auflage (ich habe dieses Skript 3 Jahre nicht mehr angefasst) habe ich im Wesentlichen das Kapitel "Graphik" neu gemacht. Dadurch, dass einige Paragraphen an verschiedenen Stellen hinzugekommen sind, ist dieses Skript nicht mehr sehr homogen, aber es existieren mittlerweile weitere Einführungen, die Sie von der *Scilab* Homepage (siehe weiter unten) herunterladen können. Diese Einführung bezieht sich auf Scilab Version 2.7, aber fast alle Beispiele sollten auch unter Scilab 2.6 funktionieren.

Danke

- an Dr. Scilab, der mir oft über seine Benutzergruppen geholfen hat
- an Bertrand Guiheneuf, der mir den magischen "patch" zur Verfügung gestellt hat, der es mir ermöglichte, die Version 2.3.1 von Scilab auf meiner Linux–Maschine zu übersetzen (die Übersetzung der neueren Versionen bereitet keine Probleme auf Linux)
- an meine Kollegen und Freunde, Stéphane Mottelet[1] Antoine Grall, Christine Bernier-Katzentsev und Didier Schmitt
- eine großes Dankschön an Patrice Moreaux für sein sorfältiges Korrekturlesen und die Verbesserungen, die er mir mitgeteilt hat
- an Helmut Jarausch, der diese Einführung ins Deutsche übersetzt hat, und der mich auf einige Fehler hingewiesen hat
- und an alle Leser für ihre Ermutigungen, Bemerkungen und Korrekturen

[1] Danke für die PDF–"Tricks" Stéphane !

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Scilab in wenigen Worten . . . 1
- 1.2 Wie benutzt man diese Einführung? . . . 1
- 1.3 Wie arbeitet man mit Scilab? . . . 2
- 1.4 Wo findet man Informationen über Scilab? . . . 2
- 1.5 Welchen Status (rechtlich gesehen) hat Scilab? . . . 3

2 Der Umgang mit Matrizen und Vektoren **5**
- 2.1 Eine Matrix eingeben . . . 5
- 2.2 Spezielle Matrizen und Vektoren . . . 6
- 2.3 Der Zuweisungsbefehl von Scilab sowie (Matrizen–)Ausdrücke . . . 9
  - 2.3.1 Einige elementare Beispiele für Matrixausdrücke . . . 10
  - 2.3.2 Elementweise Operationen . . . 12
  - 2.3.3 Lösen eines linearen Gleichungssystems . . . 13
  - 2.3.4 Referenzieren, Extrahieren, Zusammenfügen von Matrizen und Vektoren . . . 14
- 2.4 Information über den Arbeitsspeicher (*) . . . 16
- 2.5 Benutzung der Online–Hilfe . . . 17
- 2.6 Visualisieren eines einfachen Graphen . . . 18
- 2.7 Ein Skript schreiben und ausführen . . . 18
- 2.8 Diverse Ergänzungen . . . 19
  - 2.8.1 Einige Kurzschreibweisen für Matrixausdrücke . . . 19
  - 2.8.2 Diverse Bemerkungen zur Lösung linearer Gleichungssysteme (*) . . . 20
  - 2.8.3 Einige zusätzliche Matrix–Grundbefehle (*) . . . 22
  - 2.8.4 Die Funktionen `size` und `length` . . . 27
- 2.9 Übungen . . . 28

3 Programmieren in Scilab **29**
- 3.1 Schleifen . . . 29
  - 3.1.1 Die `for`–Schleife . . . 29
  - 3.1.2 Die `while`–Schleife . . . 30
- 3.2 Bedingte Anweisungen . . . 31
  - 3.2.1 Die „if then else“–Konstruktion . . . 31
  - 3.2.2 Die ‚select case‘–Konstruktion (*) . . . 31
- 3.3 Andere Datentypen . . . 32
  - 3.3.1 Zeichenketten . . . 32
  - 3.3.2 Listen (*) . . . 33
  - 3.3.3 Einige Ausdrücke mit boolschen Vektoren und Matrizen (*) . . . 37
- 3.4 Funktionen . . . 39
  - 3.4.1 Parameterübergabe (*) . . . 41
  - 3.4.2 Debuggen einer Funktion . . . 41
  - 3.4.3 Der Befehl `break` . . . 43
  - 3.4.4 Einige nützliche Grundbefehle für Funktionen . . . 44
- 3.5 Diverse Ergänzungen . . . 47
  - 3.5.1 Länge eines Bezeichners . . . 47

3.5.2 Priorität der Operatoren . . . 47
3.5.3 Rekursivität . . . 48
3.5.4 Eine Funktion ist eine Scilabvariable . . . 48
3.5.5 Dialogfenster . . . 49
3.5.6 Umwandlung einer Zeichenkette in einen Scilabausdruck . . . 50
3.6 Ein– und Ausgabe in Dateien oder das Scilab Fenster . . . 50
3.6.1 FORTRAN–ähnliche Ein– und Ausgabe . . . 51
3.6.2 C–ähnliche Ein– und Ausgabe . . . 54
3.7 Hinweise zur effizienten Programmierung in Scilab . . . 55
3.8 Übungen . . . 60

4 Graphik **63**
4.1 Graphikfenster . . . 63
4.2 Einführung in plot2d . . . 64
4.3 plot2d mit optionalen Parametern . . . 65
4.4 Varianten von plot2d: plot2d2, plot2d3 . . . 69
4.5 Zeichnen von mehreren Kurven, die unterschiedlich viele Punkte haben . . . 71
4.6 Änderungen des Graphik–Kontextes . . . 72
4.7 Zeichnen eines Histogramms . . . 73
4.8 Abspeichern von Graphiken in mehreren Formaten . . . 74
4.9 Einfache Animationen . . . 74
4.10 Flächen . . . 76
4.10.1 Einführung in plot3d . . . 76
4.10.2 Farbgebung . . . 78
4.10.3 `plot3d` und `plot3d1` mit Facetten . . . 79
4.10.4 Zeichnen einer durch $x = f_1(u, v)$, $y = f_2(u, v)$, $z = f_3(u, v)$ definierten Fläche . . . 81
4.10.5 `plot3d` mit Interpolation der Farben . . . 83
4.11 Raumkurven . . . 84
4.12 Diverses . . . 86

5 Einige Anwendungen und Ergänzungen **87**
5.1 Differentialgleichungen . . . 87
5.1.1 Basisanwendung von `ode` . . . 87
5.1.2 Van der Pol noch einmal . . . 88
5.1.3 Weiteres zu `ode` . . . 89
5.2 Erzeugen von Zufallszahlen . . . 91
5.2.1 Die Funktion `rand` . . . 91
5.2.2 Die Funktion `grand` . . . 94
5.3 Klassische Verteilungsfunktionen und ihre Inversen . . . 96
5.4 Einfache stochastische Simulationen . . . 96
5.4.1 Einführung und Notation . . . 96
5.4.2 Konfidenzintervalle . . . 97
5.4.3 Zeichnen einer empirischen Verteilungsfunktion . . . 98
5.4.4 Ein χ^2–Test . . . 99
5.4.5 Kolmogorov–Smirnov–Test . . . 100
5.4.6 Übungen . . . 101

6 Fallstricke **105**
6.1 Elementweise Definition eines Vektors oder einer Matrix . . . 105
6.2 Apropos Rückgabewerte einer Funktion . . . 106
6.3 Ich habe meine Funktion verändert, aber... . . . 107
6.4 Probleme mit `rand` . . . 107
6.5 Zeilenvektoren, Spaltenvektoren... . . . 107
6.6 Vergleichsoperatoren . . . 107
6.7 Komplexe und reelle Zahlen . . . 107

6.8 Scilab–Grundbefehle und –Funktionen 108
6.9 Auswertung von boole'schen Ausdrücken 109

A Lösungen der Übungsaufgaben **111**
A.1 Lösungen zu den Übungen aus Kapitel 2 111
A.2 Lösungen zu den Übungen aus Kapitel 3 112
A.3 Lösungen zu den Übungen aus Kapitel 4 116

Kapitel 1

Einführung

1.1 Scilab in wenigen Worten

Was ist Scilab? Falls Sie schon MATLAB kennen, so besteht eine kurze Antwort darin, dass Scilab ein frei verfügbarer Pseudo–Klon ist (siehe die Details weiter unten), der am I.N.R.I.A. (Institut National de Recherche en Informatique et Automatique) entwickelt[1] worden ist. Es gibt durchaus Unterschiede, aber die Syntax ist fast dieselbe — mit Ausnahme der Graphikroutinen. Wer MATLAB nicht kennt, dem sei kurz gesagt, dass Scilab ein komfortables System für nummerische Rechnungen ist, in dem man auf viele bewährte Methoden dieser Disziplin zurückgreifen kann, z.B.:

- Lösungen linearer Gleichungssystem (auch dünn besetzter)
- Berechnung von Eigenwerten und Eigenvektoren
- Singulärwertzerlegung und Pseudo–Inverse
- schnelle Fourier–Transformation
- mehrere Methoden zur Lösung von (auch steifen) Differentialgleichungen
- mehrere Optimierungsverfahren
- Lösung nichtlinearer Gleichungssysteme
- Erzeugung von Zufallszahlen
- mehrere Methoden der linearen Algebra für optimale Steuerungen

Auf der anderen Seite stellt Scilab ein ganzes Arsenal von Graphikbefehlen zur Verfügung, elementare, wie das Zeichnen von Polygonen und das Einlesen der Koordinaten des Maus–Zeigers, aber auch viel komplexere (zum Visualisieren von Kurven und Flächen); außerdem eine einfache, mächtige und komfortable Programmiersprache, die Matrizen als integralen Bestandteil hat. Wenn man seine Programme in Scilab austestet, so geht dies i.A. viel schneller, da man in einfacher Weise seine Variablen anzeigen lassen kann — dies ist wie mit einem Debugger. Wenn jedoch die Rechenzeiten zu groß werden (Scilab benutzt einen Interpreter), so kann man die rechenintensiven Unterprogramme in C oder FORTRAN 77 schreiben und leicht in Scilab integrieren.

1.2 Wie benutzt man diese Einführung?

Beginnen Sie beim ersten Lesen mit Kapitel 2, wo ich erkläre, wie man Scilab als Matrizen–Rechner benutzt: es reicht, die angegebenen Beispiele zu studieren; dabei können Sie die mit einem Stern (*) gekennzeichneten Abschnitte überspringen. Wenn Sie die Graphik interessiert, können Sie die ersten Beispiele im Kapitel 4 ausprobieren. Das Kapitel 3 erläutert die Grundzüge der Programmierung in Scilab. Ich habe begonnen, ein fünftes Kapitel zu schreiben, das einige Anwendungen sowie Fallstricke

[1] Scilab verwendet viele Routinen, die teilweise von Netlib–Routinen abgeleitet wurden.

zeigt — typische Fehlerquellen bei der Benutzung von Scilab (schicken Sie mir bitte Ihre Hinweise!). Überdies sei darauf hingewiesen, dass sich die Graphik–Umgebung in Scilab (das primäre Fenster, die Graphikfenster,...) in der Unix– und der Windows–Version leicht unterscheidet, d.h. die Druckknöpfe und die Menus sind nicht in gleicher Weise angeordnet. In dieser Einführung beziehen sich einige Details (wie man was in einem Menu aktiviert) auf die Unix–Version, aber Sie werden kaum Schwierigkeiten haben, das Äquivalent in der Windows–Version zu finden.

1.3 Wie arbeitet man mit Scilab?

Ganz am Anfang kann man Scilab einfach wie einen Taschenrechner benutzen, der Operationen mit Vektoren und Matrizen (und natürlich auch Skalaren) — reell oder komplex — beherrscht, und der Kurven und Flächen graphisch darstellen kann. Bei dieser Art von Anwendung benötigen Sie nur das Programm Scilab. Ziemlich schnell kommt man jedoch dazu, dass man Skripts (eine Reihe von Scilab–Befehlen) oder Funktionen benötigt. Dann arbeitet man parallel mit einem Text–Editor, z.B. emacs (Unix und Windows), wordpad (Windows) oder auch nedit, vi (Unix) ...

1.4 Wo findet man Informationen über Scilab?

Im Folgenden wird angenommen, dass Sie die Version 2.7 (oder neuer) von Scilab besitzen. Für Auskünfte aller Art konsultiere man die Scilab–Homepage:

`http://scilabsoft.inria.fr`,

wo Sie insbesondere Zugang zu verschiedenen Dokumentationen, Beiträgen anderer Benutzer, usw. haben.

Die "Scilab–Gruppe" hat (zwischen Ende 1999 und 2001) ungefähr 20 Artikel im "Linux magazine" geschrieben. Viele Aspekte von Scilab, von denen die meisten nicht in dieser Einführung erwähnt werden, wurden dort behandelt; daher empfehle ich diese. Diese Artikel können von der URL

`http://www.saphir-control.fr/articles/`,

bezogen werden.

Scilab hat außerdem eine Benutzergruppe im Usenet; dies ist der geeignete Ort zum Stellen von Fragen, für Vorschlägen, Bug–reports, Lösungsvorschläge für dort gestellte Fragen, usw.:

`comp.sys.math.scilab`

Alle Mitteilungen dieses Forums werden archiviert und können über die Scilab–Homepage eingesehen werden, indem man auf Scilab newsgroup archive[2] klickt. Weiterhin können Sie von der Homepage unter einer der Rubriken Books and Articles on Scilab oder Scilab Related Links auf weitere Dokumente zugreifen, z.B.

- l'introduction de B. Ycart (Démarrer en Scilab) ;
- "Scilab Bag Of Tricks" von Lydia E. van Dijk und Christoph L. Spiel, das sich an Personen mit guten Vorkenntnissen in Scilab wendet (die Pflege dieses Buches wurde leider von einigen Jahren abrupt beendet)
- Travaux Pratiques sur Scilab classes par themes — Projekte, die von Studenten des ENPC realisiert wurden.
- une introduction à l'informatique en utilisant Scilab (`http://kiwi.emse.fr/SCILAB/`).

Aber es gibt sicherlich noch weitere Resourcen, von denen einige Ihnen vielleicht mehr zusagen als diese Einführung.

[2] es handelt sich um einen Link auf ein Google–Archiv

1.5 Welchen Status (rechtlich gesehen) hat Scilab?

Diejenigen, die die Freie Software kennen (i.A. unter der GPL–Lizenz), könnten sich fragen, ob Scilab "frei verfügbar und kostenlos" ist. Dazu lese man den Beitrag, den Dr. Scilab in einer Mitteilung im Scilab–Forum geschrieben hat:

> *Scilab: is it really free?*
>
> Yes it is. Scilab is not distributed under GPL or other standard free software copyrights (because of historical reasons), but Scilab is an Open Source Software and is free for academic and industrial use, without any restrictions. There are of course the usual restrictions concerning its redistribution; the only specific requirement is that we ask Scilab users to send us a notice (email is enough). For more details see Notice.ps or Notice.tex in the Scilab package.
>
> Answers to two frequently asked questions: Yes, Scilab can be included a commercial package (provided proper copyright notice is included). Yes, Scilab can be placed on commercial CD's (such as various Linux distributions).

Kapitel 2

Der Umgang mit Matrizen und Vektoren

Dieser erste Teil stellt Elemente vor, die es ermöglichen, Scilab wie einen Taschenrechner für Matrizen zu benutzen.

Um Scilab zu starten, reicht es, Folgendes auf der Kommandozeile einzugeben[1]:

```
scilab
```

Falls alles gut geht, erscheint am Bildschirm das Scilab-Fenster mit der Menüleiste oben (sie bietet insbesondere den Zugriff auf `Help` und `Demos`) gefolgt von dem Schriftzug scilab und dem Prompt (`-->`), der Ihre Befehle erwartet:

```
                    ==========
                    scilab-2.7
        Copyright (C) 1989-2003 INRIA/ENPC
                    ==========
Startup execution:
  loading initial environment

-->
```

2.1 Eine Matrix eingeben

Einen der Basistypen von Scilab stellen Matrizen von reellen oder komplexen Zahlen dar (in Wirklichkeit sind es „Gleitkommazahlen“). Die einfachste Weise, eine Matrix (oder einen Vektor oder einen Skalar, die im Grunde nichts anderes sind als spezielle Matrizen) in der Scilab-Umgebung zu definieren, ist die Liste ihrer Elemente über die Tastatur einzugeben, unter Einhaltung folgender Konventionen:

- Die Elemente in einer Zeile sind durch Leerzeichen oder Kommata getrennt.
- Die Liste der Elemente muss mit eckigen Klammern [ ] umschlossen sein.
- Jede Zeile, bis auf die letzte, muss mit einem Semikolon beendet werden.

Z.B. produziert der Befehl

```
-->A=[1 1 1;2 4 8;3 9 27]
```

folgende Ausgabe:

[1] oder auf einen dafür vorgesehenen Menupunkt oder Icon zu klicken

```
 A  =
!   1.    1.    1.  !
!   2.    4.    8.  !
!   3.    9.    27. !
```

Natürlich wird die Matrix für eine mögliche spätere Anwendung im Speicher gehalten. Für den Fall, dass eine Anweisung mit einem Semikolon beendet wird, erscheint das Ergebnis nicht auf dem Bildschirm. Versuchen Sie beispielsweise:

```
-->b=[2 10 44 190];
```

Um den Inhalt des Zeilenvektors b sichtbar zu machen, muss einfach folgendes eingegeben werden:

```
-->b
```

Und die Antwort von Scilab ist folgende:

```
 b =
!  2.  10.  44.  190. !
```

Eine sehr lange Anweisung, die sich über mehrere Zeilen erstreckt, kann mit drei Punkten am Ende jeder Zeile geschrieben werden:

```
-->T = [ 1 0 0 0 0 0 ;...
-->      1 2 0 0 0 0 ;...
-->      1 2 3 0 0 0 ;...
-->      1 2 3 0 0 0 ;...
-->      1 2 3 4 0 0 ;...
-->      1 2 3 4 5 0 ;...
-->      1 2 3 4 5 6 ]
```

was folgendes ergibt:

```
 T  =
!   1.    0.    0.    0.    0.    0. !
!   1.    2.    0.    0.    0.    0. !
!   1.    2.    3.    0.    0.    0. !
!   1.    2.    3.    4.    0.    0. !
!   1.    2.    3.    4.    5.    0. !
!   1.    2.    3.    4.    5.    6. !
```

Um eine komplexe Zahl einzugeben, benutzt man die folgende Syntax (bei Eingabe eines Skalars kann auf die eckigen Klammern `[]` verzichtet werden):

```
-->c=1 + 2*%i
 c  =
    1. + 2.i

-->Y = [ 1 + %i , -2 + 3*%i ; -1 , %i]
 Y  =
!   1. + i    - 2. + 3.i !
! - 1.          i        !
```

2.2 Spezielle Matrizen und Vektoren

Es gibt Funktionen, mit deren Hilfe verschiedene spezielle Matrizen und Vektoren konstruiert werden können. Eine erste Liste soll den Überblick über einige dieser Funktionen geben (über andere wird im weiteren Verlauf gesprochen bzw. sie sind unter `Help` zu finden):

Einheitsmatrizen

Um eine Einheitsmatrix der Dimension (4,4) zu erhalten:

```
-->I=eye(4,4)
 I  =
!   1.    0.    0.    0. !
!   0.    1.    0.    0. !
!   0.    0.    1.    0. !
!   0.    0.    0.    1. !
```

Die Argumente der Funktion `eye(n,m)` sind einmal die Anzahl der Zeilen n und außerdem die Anzahl der Spalten m der Matrix (*Bem:* für $n < m$ (resp. $n > m$) erhält man die Matrix der kanonischen Surjektion (resp. Injektion) von $\mathbb{K}^m$ nach $\mathbb{K}^n$.)

Diagonalmatrizen und Zugriff auf die Diagonale einer Matrix

Um eine Diagonalmatrix zu erhalten, deren Diagonalelemente die Einträge eines Vektors sind:

```
-->B=diag(b)
 B  =
!   2.    0.     0.     0.   !
!   0.    10.    0.     0.   !
!   0.    0.     44.    0.   !
!   0.    0.     0.     190. !
```

Bem: Dieses Beispiel soll veranschaulichen, dass Scilab zwischen Groß- und Kleinschreibung unterscheidet. Geben Sie ein kleines `b` ein, um sich klarzumachen, dass dieser Vektor immer noch in der Programmumgebung existiert.

Auf eine Matrix angewandt, erlaubt die Funktion `diag` deren Hauptdiagonale in Form eines Spaltenvektors zu extrahieren:

```
-->b=diag(B)
 b  =
!   2.   !
!   10.  !
!   44.  !
!   190. !
```

Diese Funktion lässt auch ein zweites optionales Argument zu (vgl. Übungen).

Matrizen aus Nullen und Einsen

Die Funktionen `zeros` und `ones` erlauben jeweils das Erzeugen von Matrizen mit Null- bzw. Einseinträgen. Wie bei der Funktion `eye` ergeben sich auch hier die Argumente aus der gewünschten Anzahl von Zeilen und Spalten. Beispiel:

```
-->C = ones(3,4)
 C  =
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
```

Man kann aber auch den Namen einer Matrix, die bereits in der Umgebung definiert wurde, als Argument verwenden, und alles läuft so, als ob man die zwei Dimensionen dieser Matrix angegeben hätte:

```
-->O = zeros(C)
 O  =
!   0.    0.    0.    0. !
!   0.    0.    0.    0. !
!   0.    0.    0.    0. !
```

Zugriff auf den Teil unterhalb oder oberhalb der Diagonalen einer Matrix

Die Funktionen `triu` und `tril` erlauben jeweils das Extrahieren der oberen (u für upper) bzw. der unteren (l für lower) Dreiecksmatrix, beispielsweise:

```
-->U = triu(C)
 U  =
!   1.    1.    1.    1. !
!   0.    1.    1.    1. !
!   0.    0.    1.    1. !
```

Zufallsmatrizen

Die Funktion `rand` (zu der wir noch zurückkommen werden) ermöglicht das Erzeugen einer Matrix, die mit Pseudozufallszahlen gefüllt ist (entsprechend einer Gleichverteilung auf $[0, 1)$, aber es ist ebenso möglich, eine Normalverteilung zu verwenden und außerdem den Startwert der Zufallsfolge vorzugeben):

```
-->M = rand(2, 6)
 M  =
!   0.2113249    0.0002211    0.6653811    0.8497452    0.8782165    0.5608486 !
!   0.7560439    0.3303271    0.6283918    0.6857310    0.0683740    0.6623569 !
```

Vektoren mit konstantem Inkrement

Um einen (Zeilen-)Vektor x mit n Komponenten, die zwischen x_1 und x_n gleichmäßig verteilt sind (d.h. $x_{i+1} - x_i = \frac{x_n - x_1}{n-1}$, n Stellen, also $n-1$ Intervalle...), zu erzeugen, verwendet man die Funktion `linspace`:

```
-->x = linspace(0,1,11)
 x  =
!   0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8    0.9    1. !
```

Eine entsprechende Anweisung ermöglicht es, einen (Zeilen-)Vektor zu erzeugen, indem mit einem Anfangswert für die erste Komponente gestartet, das Inkrement (zwischen zwei Komponenten) vorgeschrieben und somit alle anderen Komponenten gebildet werden, bis eine festgesetzte obere Grenze erreicht ist:

```
-->y = 0:0.3:1
 y  =
!   0.    0.3    0.6    0.9 !
```

Die Syntax ist demzufolge: `y = anfangswert:inkrement:obere_grenze`. Arbeitet man mit ganzen Zahlen, gibt es keine Probleme (außer es handelt sich um sehr große Zahlen...), die Grenze festzulegen — sie entspricht dann der letzten Komponente:

```
-->i = 0:2:12
 i  =
!   0.    2.    4.    6.    8.    10.    12. !
```

Im Fall reeller Zahlen (approximiert durch Gleitkommazahlen) ist es problematischer, dadurch dass

(i) das Inkrement evtl. nicht genau auf eine Binärzahl fällt (z.B. $(0.2)_{10} = (0.00110011\ldots)_2$) und es somit in der Maschinendarstellung zu einer Rundung kommt

(ii) und die nummerischen Rundungsfehler sich je nach Berechnung der Einträge anhäufen.

Zum Beispiel:

```
-->xx = 0:0.05:0.60
 ans  =
! 0.   0.05   0.1   0.15   0.2   0.25   0.3   0.35   0.4   0.45   0.5   0.55 !
```

Bem: Je nach der CPU des Rechners kann man unterschiedliche Ergebnisse erhalten (d.h. u.U. mit 0.6 als zusätzlichem Eintrag). Häufig ist das Inkrement gleich 1 — in diesem Fall kann man es weglassen:

```
-->ind = 1:5
 ind  =
!   1.    2.    3.    4.    5. !
```

Schließlich, wenn das Inkrement positiv (bzw. negativ) und `obere_grenze < anfangswert` (bzw. `obere_grenze > anfangswert`) ist, erhält man einen Vektor ohne Komponenten (!), ein Scilab–Objekt, das „leere Matrix“ heißt (vgl. Abschnitt: Einige zusätzliche Matrix–Grundbefehle):

```
-->i=3:-1:4
 i  =
     []

-->i=1:0
 i  =
     []
```

2.3 Der Zuweisungsbefehl von Scilab sowie (Matrizen–)Ausdrücke

Scilab ist eine Sprache mit einfacher Syntax (vgl. nachfolgendes Kapitel), deren Befehle folgende Form annehmen:

```
variable = expression
```

oder einfacher

```
expression
```

Im letzteren Fall wird der Wert `expression` durch eine Defaultvariable `ans` bestimmt. Ein Scilab–Ausdruck kann ganz einfach eine Reihe von skalaren Ausdrücken sein, wie man sie in gängigen Programmiersprachen findet. Ebensogut kann dieser Ausdruck aus Matrizen und Vektoren zusammengesetzt sein, was den Anfänger beim Erlernen dieser Art von Sprachen oft durcheinanderbringt. Die skalaren Ausdrücke folgen üblichen Regeln: für (reelle, komplexe) nummerische Operationen stehen fünf Operatoren zur Verfügung: `+`, `-`, `*`, `/` und `^` (Potenzieren) sowie eine Reihe von klassischen Funktionen (vgl. Tabelle (2.1) als eine unvollständige Auflistung). Es sei darauf hingewiesen, dass all die Funktionen, die an die Γ–Funktion gebunden sind, nur ab der Version 2.4 verfügbar sind. Ferner bietet Scilab auch spezielle Funktionen an, unter denen man Bessel–Funktionen, elliptische Funktionen usw. finden kann.

So kann man, um eine Matrix koeffizientenweise zu erzeugen, neben Konstanten (die nichts anderes als elementare Ausdrücke sind) beliebige Ausdrücke verwenden, die einen skalaren (reellen oder komplexen) Wert haben, z.B.:

```
-->M = [sin(%pi/3) sqrt(2) 5^(3/2) ; exp(-1) cosh(3.7) (1-sqrt(-3))/2]
 M  =
!   0.8660254    1.4142136    11.18034          !
!   0.3678794    20.236014    0.5 - 0.8660254i !
```

(*Bem.:* Dieses Beispiel zeigt eine potenzielle Gefahr auf: man hat die Quadratwurzel einer negativen Zahl berechnet, aber Scilab verhält sich so, als ob es mit einer komplexen Zahl zu tun hätte und gibt nur eine der beiden Wurzeln als Ergebnis zurück).

abs	Absolutbetrag
exp	Exponentialfunktion
log	natürlicher Logarithmus
log10	10er–Logarithmus
cos	Cosinus (Argument im Bogenmaß)
sin	Sinus (Argument im Bogenmaß)
sinc	$\frac{sin(x)}{x}$
tan	Tangens (Argument im Bogenmaß)
cotg	Cotangens (Argument im Bogenmaß)
acos	arccos
asin	arcsin
atan	arctg
cosh	ch
sinh	sh
tanh	th
acosh	argch
asinh	argsh
atanh	argth
sqrt	Quadratwurzel
floor	ganzzahliger Anteil (kleiner) $E(x) = (\lfloor x \rfloor) = n \Leftrightarrow n \leq x < n+1$
ceil	ganzzahliger Anteil (größer) $\lceil x \rceil = n \Leftrightarrow n-1 < x \leq n$
int	ganzzahliger Anteil (abgeschnitten) : $int(x) = \lfloor x \rfloor$ für $x > 0$ und $= \lceil x \rceil$ sonst
erf	Fehlerfunktion $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
erfc	komplementäre Fehlerfunktion $ercf(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2} dt$
gamma	$\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$
lngamma	$\ln(\Gamma(x))$
dlgamma	$\frac{d}{dx} \ln(\Gamma(x))$

Tabelle 2.1: Einige Standardfunktionen von Scilab

2.3.1 Einige elementare Beispiele für Matrixausdrücke

Alle üblichen Operationen sind für Matrizen verfügbar: die Summe zweier Matrizen gleicher Dimension, das Produkt (sofern ihre Dimensionen miteinander vereinbar sind: $(n,m) \times (m,p) \ldots$), das Produkt aus einem Skalar und einer Matrix usw... Hier sind einige Beispiele (für die ein Teil der zuvor erzeugten Matrizen verwendet wird). *Bem.:* bei dem Text einer Zeile, der jeweils hinter `//` zu finden ist, handelt es sich lediglich um einen Kommentar zum Programm. Er enthält einige Bemerkungen und Erklärungen und braucht nicht abgeschrieben werden!

```
-->D = A + ones(A)   // vorher A eingeben, um den Inhalt dieser Matrix zu überprüfen
 D  =
!   2.    2.     2.  !
!   3.    5.     9.  !
!   4.    10.    28. !

-->A + M             // Summe unmöglich (3,3) + (2,3) : was sagt Scilab ?
      !--error     8
inconsistent addition

-->E = A*C           // C ist eine 3x4-Matrix  mit Einsereinträgen
 E  =
!   3.     3.     3.     3.  !
!   14.    14.    14.    14. !
!   39.    39.    39.    39. !
```

```
--> C*A   // Matrixprodukt unmöglich (3,4)x(3,3) : was sagt Scilab ?
     !--error    10
inconsistent multiplication

--> At = A'  // die Transponierte erhält man durch Hintenanstellen eines Apostrophs
 At  =
!   1.    2.    3.  !
!   1.    4.    9.  !
!   1.    8.    27. !

--> Ac = A + %i*eye(3,3)  // hier eine Matrix aus komplexen Koeffizienten
 Ac  =
!   1. + i      1.          1.        !
!   2.          4. + i      8.        !
!   3.          9.          27. + i   !

--> Ac_adj = Ac' // im komplexen Fall ergibt ' die Adjunkte (transponiert konjugiert)
 Ac_adj  =
!   1. - i      2.          3.        !
!   1.          4. - i      9.        !
!   1.          8.          27. - i   !

-->x = linspace(0,1,5)' // ein Spaltenvektor
 x  =
!   0.   !
!   0.25 !
!   0.5  !
!   0.75 !
!   1.   !

-->y = (1:5)' // auch ein Spaltenvektor
 y  =
!   1. !
!   2. !
!   3. !
!   4. !
!   5. !

-->p = y'*x   // Skalarprodukt (x | y)
 p  =
    10.

-->Pext = y*x' // man erhält eine 5x5-Matrix ((5,1)x(1,5)) vom Rang 1 : warum ?
 Pext  =
!   0.    0.25    0.5    0.75    1. !
!   0.    0.5     1.     1.5     2. !
!   0.    0.75    1.5    2.25    3. !
!   0.    1.      2.     3.      4. !
!   0.    1.25    2.5    3.75    5. !

--> Pext / 0.25   // man kann eine Matrix durch einen Skalar dividieren
 ans  =
!   0.    1.    2.     3.     4.  !
```

```
!   0.    2.    4.     6.     8.  !
!   0.    3.    6.     9.     12. !
!   0.    4.    8.     12.    16. !
!   0.    5.    10.    15.    20. !

--> A^2    // Potenzieren einer Matrix
 ans  =
!   6.      14.     36.  !
!   34.     90.     250. !
!   102.    282.    804. !

--> [0 1 0] * ans   // man kann die Variable ans verwenden (die das letzte
-->                 // Ergebnis enthält, das nicht einer Variablen zugewiesen wurde)
 ans  =
!   34.    90.    250. !

--> Pext*x - y + rand(5,2)*rand(2,5)*ones(x) + triu(Pext)*tril(Pext)*y;
--> // geben Sie ans ein, um das Ergebnis zu sehen
```

Eine andere sehr interessante Eigenschaft betrifft übliche Funktionen (siehe Tabelle 2.1), die sich auch auf Matrizen elementweise anwenden lassen: Wenn `f` eine solche Funktion beschreibt, dann ist `f(A)` die Matrix $[f(a_{ij})]$. Einige Beispiele:

```
-->sqrt(A)
 ans  =
!   1.           1.    1.        !
!   1.4142136    2.    2.8284271 !
!   1.7320508    3.    5.1961524 !

-->exp(A)
 ans  =
!   2.7182818    2.7182818    2.7182818 !
!   7.3890561    54.59815     2980.958  !
!   20.085537    8103.0839    5.320D+11 !
```

Bem.: Bei Matrixfunktionen (im Unterschied zu jenen, die elementweise definiert sind), z.B. der Exponentialfunktion, wird an den Namen der Funktion ein `m` angehängt. Will man also die Exponentialfunktion von A erhalten, lautet der Befehl: :

```
-->expm(A)
 ans  =
   1.0D+11 *

!   0.5247379    1.442794     4.1005925 !
!   3.6104422    9.9270989    28.213997 !
!   11.576923    31.831354    90.468498 !
```

2.3.2 Elementweise Operationen

Will man unter Anwendung der elementweise Operationen zwei Matrizen A und B gleicher Dimension multiplizieren oder dividieren, so benutzt man die Operatoren `.*` und `./` . Der Ausdruck `A.*B` ergibt die Matrix $[a_{ij}b_{ij}]$ und `A./B` die Matrix $[a_{ij}/b_{ij}]$. Genauso verhält es sich beim Potenzieren — durch das

Verwenden des Postfixoperators `.^` kann jeder Koeffizient potenziert werden: `A.^p` ergibt die Matrix $[a_{ij}^p]$. Man versuche zum Beispiel:

```
-->A./A
ans =
!  1.  1.  1. !
!  1.  1.  1. !
!  1.  1.  1. !
```

Bemerkungen:

- Solange `A` keine quadratische Matrix ist, wird `A^n` im Sinne von elementweise funktionieren; ich rate dennoch, `A.^n` zu verwenden, denn eine solche Schreibweise ist geeigneter, die dahintersteckende Absicht auszudrücken;
- Wenn `s` ein Skalar ist und `A` eine Matrix, so ergibt `s.^A` die Matrix $[s^{a_{ij}}]$.

2.3.3 Lösen eines linearen Gleichungssystems

Um ein lineares Gleichungssystem mit einer quadratischen Matrix zu lösen, benutzt Scilab eine LU–Zerlegung mit partieller Pivotisierung und anschließend die Lösung zweier Dreieckssysteme. Dies ist jedoch für den Anwender bei Benutzung des Operators `\` transparent. Versuchen Sie selbst:

```
-->b=(1:3)'     //ich erzeuge eine rechte Seite b
 b  =
!   1. !
!   2. !
!   3. !

-->x=A\b        // es wird Ax=b gelöst
 x  =
!   1. !
!   0. !
!   0. !

-->A*x - b    // ich überprüfe das Ergebnis durch das Berechnen des Residuenvektors
 ans  =
!   0. !
!   0. !
!   0. !
```

Man kann sich diesen Befehl einprägen, indem man die Ausgangsgleichung $Ax = y$ im Kopf habend von links mit A^{-1} multipliziert (was man durch eine Division von links durch A symbolisiert), woher auch die in Scilab verwendete Syntax stammt. Hier haben wir ein exaktes Ergebnis erhalten, doch im Allgemeinen gibt es wegen der Gleitkommaarithmetik Rundungsfehler:

```
-->R = rand(100,100); // verwende ein Semikolon, um den Bildschirm nicht mit Zahlen zu
                      // überschwemmen

-->y = rand(100,1);   // s.o.

-->x=R\y;             // Lösung von Rx=y

-->norm(R*x-y)    // Norm ermöglicht das Berechnen der Vektor-/ Matrixnorm
                  // (defaultmäßig die 2-Norm (euklidische oder hermitesche Norm))
 ans  =
    1.134D-13
```

Bem.: Sie werden nicht zwingend dasselbe Ergebnis erhalten, wenn sie nicht genauso mit der Funktion `rand` umgehen wie ich es tue... Wenn die Lösung eines linearen Gleichungssystems fragwürdig erscheint, gibt Scilab einige Informationen aus, die den Anwender darüber in Kenntnis setzen (vgl. Ergänzungen zu Lösungen von linearen Gleichungssystemen).

2.3.4 Referenzieren, Extrahieren, Zusammenfügen von Matrizen und Vektoren

Die Koeffizienten einer Matrix können mit Hilfe ihrer Indizes referenziert werden, die in runden Klammern angegeben werden. Zum Beispiel:

```
-->A33=A(3,3)
 A33  =
    27.

-->x_30 = x(30,1)
 x_30  =
  - 1.2935412

-->x(1,30)
        !--error    21
invalid index


-->x(30)
 ans  =
  - 1.2935412
```

Bem.: Wenn die Matrix aus einem Spaltenvektor besteht, kann man sich darauf beschränken, ein Element lediglich durch seinen Zeilenindex anzugeben, und entsprechend für einen Zeilenvektor.

Einer der Vorteile einer Sprache wie Scilab besteht darin, dass Untermatrizen mühelos extrahiert werden können. Für den Anfang einige einfache Beispiele:

```
-->A(:,2)   // um die zweite Spalte zu extrahieren
 ans  =
!   1. !
!   4. !
!   9. !

-->A(3,:)   // die dritte Zeile
 ans  =
!   3.    9.    27. !

-->A(1:2,1:2)  // die Hauptuntermatrix der Ordnung 2
 ans  =
!   1.    1. !
!   2.    4. !
```

Die allgemeine Syntax lautet: Ist A eine Matrix der Größe (n, m) und sind $v1 = (i_1, i_2, \ldots, i_p)$ und $v2 = (j_1, j_2, \ldots, j_q)$ zwei Indexvektoren (Zeilen– oder Spaltenvektor), für deren Werte gilt $1 \leq i_k \leq n$ und $1 \leq j_k \leq m$, dann ist `A (v1,v2)` eine Matrix (der Dimension (p, q)), die durch die Schnittpunkte der Zeilen $i_1, i_2, \ldots, i_p$ und der Spalten $j_1, j_2, \ldots, j_q$ gebildet wird. Beispiele:

```
-->A([1 3],[2 3])
 ans  =
!   1.    1.  !
!   9.    27. !
```

```
-->A([3 1],[2 1])
 ans  =
!   9.    3. !
!   1.    1. !
```

Im Allgemeinen handelt es sich in der Praxis um einfache Extraktionen, wie die eines zusammenhängenden Blocks oder die einer (oder mehrerer) Spalte(n) oder Zeile(n). In diesem Fall benutzt man den Ausdruck `i_anf:inkr:i_ende`, um Indexvektoren zu erzeugen, sowie das Zeichen `:`, um alle Indizes der entsprechenden Dimension auszuwählen (vgl. die ersten Beispiele). Auf die folgende Art und Weise erhält man die aus erster und dritter Zeile gebildete Untermatrix:

```
-->A(1:2:3,:)    // oder auch A([1 3],:)
 ans  =
!   1.    1.    1.  !
!   3.    9.    27. !
```

Betrachten wir nun das Zusammensetzen von Matrizen aus Teilmatrizen, was das Assemblieren (d.h. das Aneinanderfügen) mehrerer Matrizen ermöglicht, um so eine neue (größere) Matrix zu erhalten. Betrachten wir dazu ein Beispiel: man stelle sich eine Matrix mit folgender Blockzerlegung vor:

$$A = \left(\begin{array}{c|ccc} 1 & 2 & 3 & 4 \\ \hline 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \\ 1 & 16 & 81 & 256 \end{array}\right) \qquad = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

Wir werden zunächst die Untermatrizen $A_{11}, A_{12}, A_{21}, A_{22}$ definieren:

```
-->A11=1;

-->A12=[2 3 4];

-->A21=[1;1;1];

-->A22=[4 9 16;8 27 64;16 81 256];
```

Schließlich erhalten wir A durch das Zusammenfügen dieser vier Blöcke:

```
-->A=[A11 A12; A21 A22]
A =
!  1.  2.  3.   4.   !
!  1.  4.  9.   16.  !
!  1.  8.  27.  64.  !
!  1.  16. 81.  256. !
```

Die Syntax ist die gleiche, als ob unsere Untermatrizen einfache Skalare wären (natürlich wird Kompatibilität bezüglich der Zeilen– und Spaltenzahl der unterschiedlichen Blöcke erwartet...).

Es existiert eine besondere Syntax, um die Gesamtheit der Zeilen bzw. Spalten einer Matrix zu löschen: Ist $v = (k_1, k_2, \ldots, k_p)$ ein Indexvektor, der die Nummern der Zeilen bzw. der Spalten einer Matrix M bestimmt, dann löscht `M(v,:) = []` die Zeilen $k_1, k_2, \ldots, k_p$ von M und `M(:,v) = []` die Spalten $k_1, k_2, \ldots, k_p$. Schließlich löscht `u(v) = []` für einen (Zeilen– bzw. Spalten–)vektor u die entsprechenden Einträge.

Zur Anordnung der Elemente einer Matrix

Scilab speichert Matrizen, indem es die Spalten hintereinander packt; dies beeinflusst viele Funktionen (z.B. `matrix`, mit dessen Hilfe man die Gestalt von Matrizen ändern kann). Speziell bei Operationen, die einen Teil einer Matrix auslesen bzw. einfügen, ist es möglich, diese implizite Anordnung zu benutzen, indem man nur einen Vektor von Indizes angibt (statt zwei für die Angabe der Zeile und Spalte). Hier einige Beispiele mit der zuletzt definierten Matrix `A`

```
-->A(5)
 ans  =
    2.

-->A(5:9)
 ans  =
!   2.  !
!   4.  !
!   8.  !
!   16. !
!   3.  !

-->A(5:9) = -1  // hier werden die Element 5 bis 9 eingefügt
 A  =
!   1.  - 1.  - 1.     4.   !
!   1.  - 1.    9.     16.  !
!   1.  - 1.    27.    64.  !
!   1.  - 1.    81.    256. !
```

2.4 Information über den Arbeitsspeicher (*)

Es genügt, den folgenden Befehl anzugeben:

```
-->who
your variables are...

 Anew      A         A22       A21      A12 A11   x_30      A33       x
 y         R         b         Pext     p Ac_adj    Ac     At        E
 D         cosh      ind       xx       i linspace  M     U       O
 zeros     C         B         I        Y c   T     startup   ierr
 scicos_pal          home      PWD      TMPDIR  percentlib     fraclablib
 soundlib  xdesslib  utillib   tdcslib  siglib  s2flib    roblib    optlib    metalib
 elemlib   commlib   polylib   autolib  armalib alglib    mtlblib   SCI       %F
 %T        %z        %s        %nan     %inf old   newstacksize        $
 %t        %f        %eps      %io      %i %e   %pi
 using      14875 elements  out of    1000000.
          and         75 variables out of       1023
```

und es erscheinen:

- Variablen, welche in der Umgebung angegeben wurden: `Anew, A, A22, A21`, ..., `b` in der umgekehrten Reihenfolge ihrer Erzeugung. Im Grunde war die Matrix A die erste erzeugte Variable, aber wir haben ihre Dimensionen in dem Beispiel zum Zusammenfügen von Matrizen erhöht (von (3,3) auf (4,4)). In einem solchen Fall wird die Anfangsvariable gelöscht und mit neuer Dimension neu erzeugt. Dies ist ein wichtiger Punkt, der im Zusammenhang mit der Programmierung in Scilab nocheinmal Erwähnung findet;

- die Namen der Scilab–Bibliotheken (die mit `lib` enden) und ein Funktionsname: `cosh`. In Wirklichkeit gelten die (in Scilab programmierten) Funktionen und Bibliotheken für Scilab als Variablen; *Bem.:* Die in Fortran 77 und in C programmierten Scilabprozeduren werden „Scilab–Grundbefehle“ genannt und gelten nicht als Scilabvariablen; im weiteren Text gebrauche ich gelegentlich fälschlicherweise den Terminus „Grundbefehl“, um auf Scilabfunktionen (in Scilab programmiert) hinzuweisen, die in der Standardumgebung angeboten werden;

- vordefinierte Konstanten wie π, e, die imaginäre Einheit i, die Maschinengenauigkeit *eps* und die beiden anderen klassischen Konstanten der Gleitkommaarithmetik *nan* (not a number) und inf (für ∞); diese Variablen, die notwendigerweise mit `%` beginnen, können nicht gelöscht werden;

- eine wichtige Variable `newstacksize`, die standardmäßig der Größe des Heap (d.h. des verfügbaren Speichers) entspricht.

- Ferner zeigt Scilab die Anzahl der gebrauchten 8–Byte–Wörter, die verfügbare Speicherkapazität, die Anzahl der verwendeten Variablen, sowie deren maximal erlaubte Anzahl.

Man kann mit Hilfe des Befehls `stacksize(Anzahl_8bytes)` die Größe des Heap verändern, wobei Anzahl_8bytes die gewünschte neue Größe angibt; derselbe Befehl `stacksize()` ohne Argument ermöglicht, die Größe des Heaps sowie die maximal erlaubte Anzahl von Variablen herauszukriegen. Schließlich verwendet man den Befehl `clear v1`, wenn man eine Variable `v1` aus der Umgebung loswerden (und so Speicherplatz wiedergewinnen) will. Der Befehl `clear` an sich löscht alle Ihre Variablen; wenn Sie also lediglich die Variablen `v1, v2, v3` löschen wollen, müssen Sie den Befehl `clear v1 v2 v3` benutzen.

2.5 Benutzung der Online–Hilfe

Man erhält sie durch das Anklicken des `Help`–Buttons im Scilabfenster... Seit Version 2.7 sind die Hilfeseiten in `html` formatiert[2]. Voreingestellt ist ein ziemlich einfacher html–Browser (, der nicht ganz so schön aussehende Seiten erzeugt), aber man kann einen anderen Browser mit folgendem Trick wählen[3]

```
-->global %browsehelp; %browsehelp=[];
-->help
```

Die erste Zeile modifiziert die globale Variable `%browsehelp`, die jetzt eine leere Matrix enthält. Nach der Eingabe `help` wird Ihnen eine Auswahl verschiedener Browser vorgeschlagen (z.B. kann man unter Linux mozilla auswählen). Weiterhin können Sie diese Variable in Ihrer Konfigurationsdatei `.scilab` in Ihrem Homeverzeichnis setzen[4].

Wenn Sie einfach das Kommando `help` eingeben (oder auf einen entsprechenden Button klicken), wird eine html–Seite angezeigt, die eine Einteilung aller Hilfeseiten in bestimmte Rubriken (`Scilab Programming, Graphic Library, Utilities and Elementary functions,...`) anzeigt. Indem man auf eine spezielle Rubrik klickt, bekommt man eine Liste aller zu dieser Rubrik gehörenden Funktionen samt einer Kurzbeschreibung. Indem man nun auf eine dieser Funktionen klickt, erhält man die Hilfeseite dieser Funktion.

Wenn Sie den Namen der Funktion, die Sie interessiert, kennen, können Sie direkt `help` *Funktionsname* im Scilab Kommandofenster eingeben.

Schließlich erlaubt es Ihnen der Befehl `apropos` *Schlüsselwort* bzw. das `help` Untermenu `Apropos` alle Hilfeseiten nach dem *Schlüsselwort* zu durchsuchen.

Zur Zeit kann es schwierig sein, sich mit den Rubriknamen zurechtzufinden (z.B. ist Elementary functions eine “Rumpelkammer”, die eine neue Einteilung verdient), man zögere daher nicht `apropos` zu benutzen.

[2] Das Originalformat ist eigentlich `xml`, aus dem die `html` Seiten erzeugt werden.

[3] in neueren Versionen von Scilab enthält das Pop–Up Menu “Help” einen Unterpunkt “Configure”

[4] Sie müssen u.U. die Datei `SCI/macros/util/browsehelp.sci` modifizieren, wenn Ihr bevorzugter Browser nicht in dieser Auswahl enthalten ist.

2.6 Visualisieren eines einfachen Graphen

Angenommen, wir wollen die Funktion $y = e^{-x}\sin(4x)$ für $x \in [0, 2\pi]$ visualisieren. Wir können zuallererst mit Hilfe der Funktion `linspace` eine Unterteilung des Intervalls vornehmen:

```
-->x=linspace(0,2*%pi,101);
```

dann die Funktionswerte für alle Komponenten der Unterteilung ausrechnen, was dank der Vektor–Anweisungen keine Schleife erfordert:

```
-->y=exp(-x).*sin(4*x);
```

und schließlich:

```
-->plot(x,y,'x','y','y=exp(-x)*sin(4x)')
```

wobei die drei letzten Zeichenketten (jeweils eine Legende für die Abszisse, eine für die Ordinate und ein Titel) optional sind. Der Befehl ermöglicht eine Kurve zu zeichen, die durch Punkte verläuft, deren Koordinaten durch die Vektoren gegeben sind — `x` für die Abszisse und `y` für die Ordinate. Da die Punkte durch Geradenabschnitte verbunden sind, wird der Verlauf umso genauer, je zahlreicher die Punkte sind.

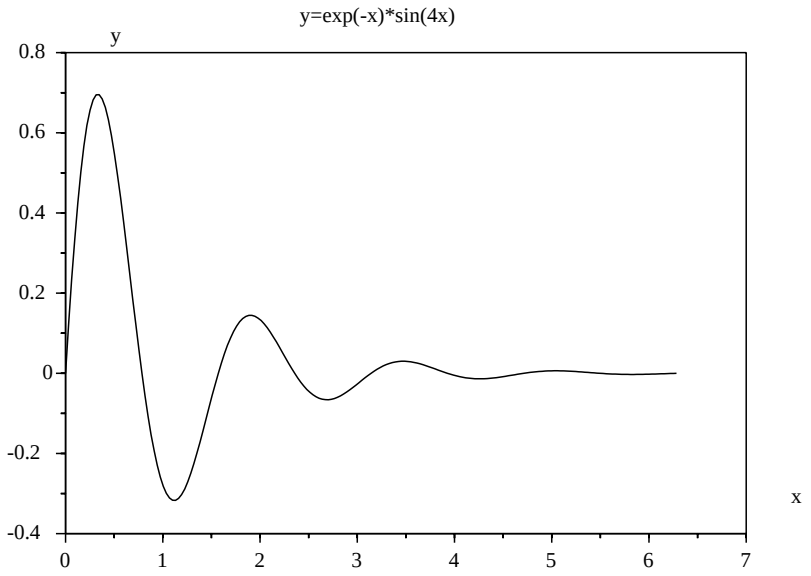

Abbildung 2.1: Eine einfache Zeichnung

Bem.: Dieser Befehl hat Einschränkungen; man kann z.B. nur eine Kurve zeichnen. Im Kapitel über Graphiken werden wir lernen, mit `plot2d`, das viel leistungsfähiger ist, umzugehen.

2.7 Ein Skript schreiben und ausführen

Man kann eine Reihe von Befehlen in eine Datei `datname` schreiben und sie durch folgende Anweisung ausführen:

```
-->exec('datname')    // oder auch exec("datname")
```

Eine bequemere Methode besteht darin, das Untermenu `File Operations` auszuwählen, welches nach Anklicken des Button `File` im Scilabfenster erscheint. Man erhält daraufhin ein Menu, welches das Auswählen seiner Datei gestattet (evtl. nach Änderung des aktuellen Verzeichnisses), und es bleibt solange geöffnet, bis man auf den Button `Exec` klickt. Als Beispielskript greifen wir auf den Graph der Funktion $e^{-x}\sin(4x)$ zurück, wobei das Visualisierungsintervall $[a, b]$ sowie dessen Diskretisierung gewählt werden können. Ich schreibe also in eine Datei, die z.B. `script1.sce` heißt, folgende Scilab–Befehle:

```
// mein erstes Scilab-Skript

a = input(" Wert von a eingeben: ");
b = input(" Wert von b eingeben: ");
n = input(" Anz. der Intervalle n : ");

// Abszissenberechnung
x = linspace(a,b,n+1);

// Ordinatenberechnung
x = exp(-x).*sin(4*x);

// eine kleine Graphik
plot(x,y,'x','y','y=exp(-x)*sin(4x)')
```

Bem.:

1. Um Scilab–Skript–Dateien leicht als solche erkennen zu können, ist es ratsam, Dateinamen mit der Endung `.sce` zu verwenden (enthält eine Datei Funktionen, so hat ihr Name die Endung `.sci`).

2. Manche Editoren können mit einem Eingabemodus ausgestattet werden, der das Schreiben von Scilabprogrammen erleichtert (→ Scilab–Homepage). Für `emacs` existieren zwei verschiedene, der bessere davon ist von Alexander Vigodner; die neueste Version kann von der URL:

   `http://www.geocities.com/avezunchik/scilab.html`

   bezogen werden.

3. Ein Skript dient oft als Hauptprogramm einer Anwendung, die in Scilab geschrieben wurde.

2.8 Diverse Ergänzungen

2.8.1 Einige Kurzschreibweisen für Matrixausdrücke

Wir haben bereits gesehen, dass die Multiplikation einer Matrix mit einem Skalar von Scilab in natürlicher Weise unterstützt wird (dasselbe gilt für die Division einer Matrix durch einen Skalar). Dagegen verwendet Scilab weniger offensichtliche Kurzschreibweisen, wie die Addition eines Skalars und einer Matrix. Der Ausdruck `M + s`, wobei `M` eine Matrix und `s` ein Skalar ist, ist eine Kurzform für

```
M + s*ones(M)
```

d.h. dass der Skalar zu jedem Element der Matrix hinzuaddiert wird.

Eine andere Kurzform: Wenn man in einem Ausdruck der Form `A./B` (der normalerweise die elementweise Divison zweier Matrizen derselben Dimension bezeichnet) `A` ein Skalar ist, so wird der Ausdruck zu einer Kurzform für

```
A*ones(B)./B
```

Man erhält also die Matrix $[a/b_{ij}]$. Diese Kurzformen ermöglichen eine in vielen Fällen kompaktere Schreibweise (vgl. Übungen). Ist beispielsweise f eine in der Programmumgebung definierte Funktion, x ein Vektor und s eine skalare Variable, dann ist

```
s./f(x)
```

ein Vektor derselben Größe wie x, dessen i–te Komponente gleich $s/f(x_i)$ ist. Es scheint, als könne man für die Berechnung eines Vektors mit den Komponenten $1/f(x_i)$ schreiben:

```
1./f(x)
```

Weil aber `1.` syntaktisch gleich mit `1` ist, entspricht das Ergebnis nicht den Erwartungen. Der geeignete Weg, zu erhalten was man will, besteht darin, die Zahl mit runden Klammern zu umschließen oder ein Leerzeichen zwischen die Zahl und den Punkt einzufügen:

```
(1)./f(x)       // oder auch 1 ./f(x)
```

2.8.2 Diverse Bemerkungen zur Lösung linearer Gleichungssysteme (*)

1. Hat man mehrere rechte Seiten, kann man nach folgendem Schema verfahren:

```
-->y1 = [1;0;0;0]; y2 = [1;2;3;4];  // zwei rechte Seiten (man kann mehrere
-->                                 // Anweisungen in eine Zeile schreiben)
-->X=A\[y1,y2]     // Zusammenfügen von y1 und y2
 X  =
!   4.          - 0.8333333 !
! - 3.           1.5       !
!   1.3333333   - 0.5       !
! - 0.25          0.0833333 !
```

   Die erste Spalte der Matrix X stellt die Lösung des linearen Gleichungssystems $Ax^1 = y^1$ dar, während die zweite der Lösung von $Ax^2 = y^2$ entspricht.

2. Wir haben bereits gesehen, dass wenn A eine quadratische $n \times n$–Matrix und b ein Spaltenvektor mit n Komponenten (also eine Matrix der Größe (n,1)) ist, uns dann

```
x = A\b
```

   die Lösung des linearen Gleichungssystems $Ax = b$ liefert. Wird die Matrix A als singulär erkannt, gibt Scilab eine Fehlermeldung zurück. Zum Beispiel:

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

-->b=[1;1];

-->A\b
    !--error    19
singular matrix
```

   Erweist sich die Matrix A dagegen als schlecht konditioniert (oder evtl. schlecht äquilibriert), wird ein Resultat geliefert, aber sie ist begleitet von einer Warnmeldung mit einer Abschätzung der Inversen der Konditionszahl ($cond(A) = ||A||\ ||A^{-1}||$):

```
-->A=[1 2;1 2+3*%eps];
-->A\b
 warning
 matrix is close to singular or badly scaled.
 results may be inaccurate. rcond =   7.4015D-17

 ans  =
!   1. !
!   0. !
```

Ist Ihre Matrix hingegen nicht quadratisch, hat aber dieselbe Anzahl von Zeilen wie die rechte Seite, wird Scilab Ihnen eine Lösung zurückgeben (einen Spaltenvektor, dessen Dimension gleich der Spaltenzahl von A ist) ohne i.A. eine Fehlermeldung anzuzeigen. In dem Falle, dass die Gleichung $Ax = b$ keine eindeutige Lösung[5] besitzt, kann man immer einen eindeutigen Vektor x auswählen, der bestimmte Eigenschaften erfüllt (x von minimaler Norm und Lösung von $\min ||Ax - b||$). In diesem Fall wird die Lösung einem anderen Algorithmus übertragen, der es (evtl.) ermöglicht, diese Pseudo–Lösung[6] zu finden. Der Nachteil besteht darin, dass, wenn Sie einen Fehler in der Definition Ihrer Matrix eingebaut haben (Sie haben z.B. eine zusätzliche Spalte definiert, und Ihre Matrix hat nun die Größe (n,n+1)), Sie es u.U. nicht direkt merken. Nehmen wir nocheinmal das vorige Beispiel:

```
-->A(2,3)=1         // Unfug
A =
!  1.  2.  0. !
!  1.  2.  1. !

-->A\b
 ans  =
!   0.         !
!   0.5        !
! - 3.140D-16 !

-->A*ans - b
 ans  =
   1.0D-15 *

! - 0.1110223 !
! - 0.1110223 !
```

Abgesehen davon, dass dieses Beispiel vor den Konsequenzen solchen Unfugs warnt, ist es darüber hinaus aufschlussreich, was folgende Punkte betrifft:

- `x = A\y` erlaubt also auch das Lösen eines least–squares–Problems (wenn die Matrix nicht maximalen Rang hat, ist es besser `x = pinv(A)*b` zu benutzen, die Pseudoinverse, die durch Singulärwertzerlegung von A berechnet wird (diese Zerlegung erhält man mit Hilfe der Funktion `svd`));
- Die Anweisung `A(2,3)=1` (der dumme Fehler...) ist im Grunde eine Kurzform für

  ```
  A = [A,[0;1]]
  ```

  D.h. Scilab stellt fest, dass Sie die Matrix A (durch eine dritte Spalte) ergänzen wollen, ihm aber ein Element fehlt. In diesem Fall wird mit Nullen aufgefüllt.
- Das Element an Position (2,2) ist normalerweise gleich $2 + 3\,\epsilon_m$, wegen Rundungsfehlern nur beinahe. Nun kann die Maschinenkonstante (ϵ_m) definiert werden als die größte Zahl, für die $1 \oplus \epsilon_m = 1$ in Gleitkommaarithmetik[7] gilt. Als Konsequenz sollte $2 \oplus 3\epsilon_m > 2$ sein, aber das Ausgabefenster zeigt 2 an. Das liegt an dem standardmäßig benutzten Format, was man aber mit der Anweisung `format` ändern kann:

[5] Seien (m, n) die Dimensionen von A (mit $m \neq n$), dann existiert eine eindeutige Lösung genau dann, wenn $m > n$, $KerA = \{0\}$ und schließlich $b \in ImA$ (diese letzte Bedingung wird die Ausnahme sein, wenn b beliebig aus K^m gewählt ist); in allen anderen Fällen gibt es entweder keine Lösung oder unendlich viele.

[6] In den schwierigen Fällen, d.h. wenn die Matrix nicht von maximalem Rang ist ($rg(A) < min(n, m)$, wobei m und n die beiden Dimensionen sind), ist es besser, diese Lösung über die Pseudoinverse von A zu berechnen (`x = pinv(A)*b`).

[7] Tatsächlich kann jede reelle Zahl x mit der Eigenschaft $m \leq |x| \leq M$ durch eine Gleitkommazahl $fl(x)$ mit $|x - fl(x)| \leq \epsilon_m|x|$ codiert werden, wobei m und M jeweils die kleinste bzw. größte positive Zahl sind, die mithilfe der normalisierten Gleitkommadarstellung codierbar sind.

```
-->format('v',19)

-->A(2,2)
ans =
    2.0000000000000009
```

wohingegen die Anzeige standardmäßig `format('v',10)` entspricht (siehe `Help` für die Bedeutung der Argumente).

- Wenn man die „Lösung“ von $Ax = b$ ausrechnet, ohne eine besondere Variable anzugeben, bedient sich Scilab der Variablen `ans`, die ich dann benutzen kann, um das Residuum $Ax - b$ auszurechnen.

3. Mit Scilab kann man außerdem direkt ein lineares System des Typs $xA = b$ lösen, wobei x und b Zeilenvektoren sind und A eine quadratische Matrix (transponiert ergibt sich ein klassisches lineares System $A^\top x^\top = b^\top$). Es reicht so zu tun, als ob man von rechts mit A^{-1} multipliziert (man symbolisiert diese Operation durch eine Division von rechts durch A):

```
x = b/A
```

Und genauso wie vorher gibt Scilab eine Lösung zurück, wenn A eine rechteckige Matrix ist (Spaltenanzahl ist identisch mit b); man muss jedoch auch hier aufpassen.

2.8.3 Einige zusätzliche Matrix–Grundbefehle (*)

Summe und Produkt der Koeffizienten einer Matrix, die leere Matrix

Um die Koeffizienten einer Matrix zu addieren, benutzt man `sum`:

```
-->sum(1:6)   // 1:6 = [1 2 3 4 5 6] : man sollte also 6*7/2 = 21  erhalten !!!!!
 ans  =
    21.
```

Diese Funktion lässt ein zusätzliches Argument zu, um die zeilen– bzw. spaltenweise Addition vorzunehmen:

```
-->B = [1 2 3; 4 5 6]
 B  =
!   1.    2.    3. !
!   4.    5.    6. !

-->sum(B,"r")  // addiert spaltenweise -> man erhält eine Zeile
 ans  =
!   5.    7.    9. !

-->sum(B,"c") // addiert zeilenweise -> man erhält eine Spalte
 ans  =
!   6.  !
!   15. !
```

Es gibt ein sehr praktisches Objekt, die „leere Matrix“, die man folgendermaßen definiert:

```
-->C =[]
 C  =
     []
```

Die leere Matrix interagiert mit anderen Matrizen nach folgenden Regeln: `[] + A = A` und `[]*A = []`. Wenn man nun die Funktion `sum` auf die leere Matrix anwendet, erhält man selbstverständlich das Ergebnis

```
-->sum([])
 ans  =
    0.
```

identisch mit der in der Mathematik üblichen Konvention für die Summenbildung:

$$S = \sum_{i \in E} u_i = \sum_{i=1}^{n} u_i \quad \text{für } E = \{1, 2, \ldots, n\}$$

Wenn die Menge E leer ist, gilt per Konvention $S = 0$.

Analog zur Summenbildung verfügt man über die Funktion `prod`, um das Produkt von Elementen einer Matrix zu bilden:

```
-->prod(1:5)     // man muss 5! = 120 erhalten
 ans  =
    120.

-->prod(B,"r")  // Geben Sie B ein, um nochmal diese Matrix zu sehen...
 ans  =
!   4.    10.    18. !

-->prod(B,"c")
 ans  =
!   6.   !
!   120. !

-->prod(B)
 ans  =
    720.

-->prod([])
 ans  =
    1.
```

Man erhält wiederum die in der Mathematik übliche Konvention:

$$\prod_{i \in E} u_i = 1, \quad \text{für } E = \emptyset.$$

akkumulierende Summen und Produkte

Die Funktionen `cumsum` et `cumprod` berechnen die akkumulierte Summe bzw. Produkt eines Vektors oder einer Matrix:

```
-->x = 1:6
 x  =
!   1.    2.    3.    4.    5.    6. !

-->cumsum(x)
 ans  =
!   1.    3.    6.    10.    15.    21. !

-->cumprod(x)
 ans  =
!   1.    2.    6.    24.    120.    720. !
```

Bei einer Matrix wird die Akkumulation spaltenweise durchgeführt:

```
-->x = [1 2 3;4 5 6]
 x  =
!   1.    2.    3. !
!   4.    5.    6. !

-->cumsum(x)
 ans  =
!   1.    7.     15. !
!   5.    12.    21. !
```

Und wie bei den Funktionen `sum` und `prod` kann man kummulative Summen und Produkte sowohl zeilenweise als auch spaltenweise durchführen:

```
-->cumsum(x,"r")   // spaltenweise Teil--Summen !
 ans  =
!   1.    2.    3. !
!   5.    7.    9. !

-->cumsum(x,"c")   // zeilenweise Teil--Summen !
 ans  =
!   1.    3.    6.  !
!   4.    9.    15. !
```

Hier zeigt sich die ungeschickte Wahl der Optionen, die gewisse Funktionen auf Zeilen oder Spalten operieren lässt[8] : für die Funktionen `sum` und `prod` scheint es, als wenn die Form des Ergebnisses bezeichnet wird: `sum(x,"r")` liefert eine Zeile (r für row, Zeile auf englisch), d.h. die Funktion operiert auf Spalten — mnemotechnisch widersprüchlich!

Minimum und Maximum eines Vektors oder einer Matrix

Die Funktionen `min` und `max` liefern das Minimum und Maximum bzw. einen Vektor der Minima und Maxima einer Zeile/Spalte einer Matrix. In Bezug auf das zweite Argument funktionieren sie genauso wie `sum` und `prod`. Sie gestatten einen zweiten Funktionswert, der den ersten Index angibt, an dem das Minimum bzw. Maximum angenommen wird. Hier einige Bespiele:

```
-->x = rand(1,5)
 x  =
!   0.7738714    0.7888738    0.3247241    0.4342711    0.2505764 !

-->min(x)
 ans  =
    0.2505764

-->[xmin, imin] = min(x)
 imin  =
    5.              // das Minimum wird bei x(5) angenommen
 xmin  =
    0.2505764

-->y = rand(2,3)
 y  =
!   0.1493971    0.475374     0.8269121 !
!   0.1849924    0.1413027    0.7530783 !

-->[ymin, imin] = min(y)
```

[8] `sum`, `prod`, `cumsum`, `cumprod`, `mean`, `st_deviation`, `min`, `max`

```
 imin  =
!   2.    2. !      // das Minimum wird bei y(2,2) angenommen
 ymin  =
    0.1413027

-->[ymin, imin] = min(y,"r")  // Minima der Spalten
 imin  =
!   1.    2.    2. !          // => die Minima sind  y(1,1) y(2,2) y(2,3)
 ymin  =
!   0.1493971    0.1413027    0.7530783 !

-->[ymin, imin] = min(y,"c")  // Minima der Zeilen
 imin  =
!   1. !           // die Minima sind  y(1,1)
!   2. !           //                  y(2,2)
 ymin  =
!   0.1493971 !
!   0.1413027 !
```

Mittelwert und Standardabweichung

Die Funktionen `mean` und `st_deviation` erlauben die Berechnung des/der Mittelwerte bzw. Standardabweichung(en) eines Vektors oder einer Matrix; dabei wird die Standardabweichung gemäß folgender Formel berechnet:

$$\sigma(x) = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{1/2} , \; \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

```
-->x = 1:6
 x  =
!   1.    2.    3.    4.    5.    6. !

-->mean(x)
 ans  =
    3.5

-->st_deviation(x)
 ans  =
    1.8708287
```

Ebenso erhält man die Mittelwerte (bzw. Standardabweichungen) der Zeilen oder Spalten einer Matrix:

```
-->x = [1 2 3;4 5 6]
 x  =
!   1.    2.    3. !
!   4.    5.    6. !

-->mean(x,"r")   // die Mittelwerte jeder Spalte
 ans  =

!   2.5    3.5    4.5 !

-->mean(x,"c")   // die Mittelwerte jeder Zeile
 ans  =

!   2. !
!   5. !
```

Die Form einer Matrix ändern

Die Funktion `matrix` erlaubt, eine Matrix umzugestalten, indem man ihr neue Dimensionen vorgibt (wobei sich die Anzahl der Koeffizienten insgesamt nicht ändert).

```
-->B = [1 2 3; 4 5 6]
 B  =
!   1.    2.    3. !
!   4.    5.    6. !

-->B_new = matrix(B,3,2)
 B_new  =
!   1.    5. !
!   4.    3. !
!   2.    6. !
```

Sie arbeitet, indem sie die Koeffizienten Spalte für Spalte anordnet. Eine ihrer Anwendungen ist das Transformieren eines Zeilen– in einen Spaltenvektor und umgekehrt. Es sei noch auf eine Kurzform hingewiesen, die eine Matrix `A` (Zeilen– und Spaltenvektoren eingeschlossen) in einen Spaltenvektor `v` zu transformieren gestattet: `v = A(:)`, Beispiel:

```
-->A = rand(2,2)
 A  =
!   0.8782165    0.5608486 !
!   0.0683740    0.6623569 !

-->v=A(:)
 v  =
!   0.8782165 !
!   0.0683740 !
!   0.5608486 !
!   0.6623569 !
```

Vektoren mit logarithmischem Abstand

Gelegentlich braucht man einen Vektor mit einer logarithmischen Inkrementierung für die Komponenten (d.h. derart, dass das Verhältnis zweier aufeinanderfolgender Komponenten konstant ist: $x_{i+1}/x_i = Const$): in diesem Fall kann man die Funktion `logspace` verwenden; `logspace(a,b,n)` ermöglicht einen solchen Vektor mit n Komponenten zu erhalten, wobei die erste und letzte jeweils 10^a und 10^b sind.

```
-->logspace(-2,5,8)
 ans  =
!   0.01    0.1    1.    10.    100.    1000.    10000.    100000. !
```

Eigenwerte und –vektoren

Die Funktion `spec` ermöglicht das Berechnen der Eigenwerte einer (quadratischen!) Matrix:

```
-->A = rand(5,5)
 A  =
!   0.2113249    0.6283918    0.5608486    0.2320748    0.3076091 !
!   0.7560439    0.8497452    0.6623569    0.2312237    0.9329616 !
!   0.0002211    0.6857310    0.7263507    0.2164633    0.2146008 !
!   0.3303271    0.8782165    0.1985144    0.8833888    0.312642  !
!   0.6653811    0.0683740    0.5442573    0.6525135    0.3616361 !
```

```
-->spec(A)
 ans  =
!   2.4777836              !
! - 0.0245759 + 0.5208514i !
! - 0.0245759 - 0.5208514i !
!   0.0696540              !
!   0.5341598              !
```

und gibt das Ergebnis in Form eines Spaltenvektors zurück (Scilab verwendet die QR–Methode, die darin besteht, eine iterative *Schur*–Zerlegung der Matrix zu erhalten). Die Eigenvektoren erhält man mit `bdiag`. Für ein verallgemeinertes Eigenwertproblem können Sie die Funktion `gspec` verwenden.

2.8.4 Die Funktionen `size` und `length`

`size` ermöglicht, die beiden Dimensionen (Anzahl der Zeilen und der Spalten) einer Matrix zu bestimmen:

```
-->[nr,nc]=size(B)   // B ist die Matrix der Größe (2,3) aus dem vorigen Beispiel
 nc  =
    3.
 nr  =
    2.

-->x=5:-1:1
 x  =
!   5.    4.    3.    2.    1. !

-->size(x)
 ans  =
!   1.    5. !
```

wogegen `length` die Anzahl der Elemente einer (reellen oder komplexen) Matrix liefert. Genauso erhält man für einen Zeilen– oder Spaltenvektor unmittelbar die Anzahl seiner Komponenten:

```
-->length(x)
 ans  =
    5.

-->length(B)
 ans  =
    6.
```

Tatsächlich werden diese beiden Grundbefehle vor allem innerhalb von Funktionen benutzt, um die Größe von Matrizen und Vektoren zu bestimmen, wodurch vermieden wird, diese als zusätzliche Argumente übergeben zu müssen. Beachten Sie, dass man auch mit `size(A,'r')` (oder `size(A,1)`) und `size(A,'c')` (oder `size(A,2)`) die Anzahl der Zeilen (rows) und der Spalten (columns) der Matrix A erhalten kann.

2.9 Übungen

1. Definieren Sie die folgende Matrix der Ordnung n (die Details der Funktion `diag` sehen Sie unter `Help` nach):

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

2. Sei A eine quadratische Matrix; was ergibt `diag(diag(A))` ?

3. Die Funktionen `tril` (bzw. `triu`) ermöglichen, das untere (bzw. obere) Dreieck einer Matrix zu extrahieren. Definieren Sie eine beliebige quadratische Matrix A (z.B. mit `rand`) und konstruieren Sie mittels einer einzigen Anweisung eine untere Dreiecksmatrix T, so dass $t_{ij} = a_{ij}$ für $i > j$ (die Bereiche unterhalb der Diagonalen von A und T sind gleich) und so dass $t_{ii} = 1$ (T hat eine Einheitsdiagonale).

4. Sei `X` eine Matrix (oder ein Vektor...), die in der Umgebung definiert wurde. Schreiben Sie eine Anweisung, die das Berechnen einer Matrix `Y` (derselben Größe wie `X`) erlaubt, deren Elemente an der Position (i, j) gleich $f(X_{ij})$ sind:

   (a) $f(x) = 2x^2 - 3x + 1$

   (b) $f(x) = |2x^2 - 3x + 1|$

   (c) $f(x) = (x - 1)(x + 4)$

   (d) $f(x) = \frac{1}{1+x^2}$

5. Zeichnen Sie einen Graphen zu der Funktion $f(x) = \frac{\sin x}{x}$ für $x \in [0, 4\pi]$ (Schreiben Sie ein Skript).

6. Eine kleine Illustration des Gesetzes der großen Zahlen: Erzeugen Sie mit Hilfe der Funktion `rand` n Realisationen einer Gleichverteilung in Form eines Vektors x, berechnen Sie die akkumulierten Mittelwerte des Vektors, d.h. den Vektor $\bar{x}$, dessen n Komponenten sich gemäß $\bar{x}_k = \frac{1}{k} \sum_{i=1}^{k} x_i$ ergeben, und zeichnen Sie den Verlauf der so erhaltenen Folge. Testen Sie dies mit immer größeren Werten von n.

Kapitel 3

Programmieren in Scilab

Scilab verfügt — neben eingebauten Grundbefehlen, die eine sehr kompakte, der mathematischen ähnliche Formulierung erlaubt — über eine einfache, aber ausreichend umfangreiche Programmiersprache. Der wesentliche Unterschied im Vergleich zu den gängigen Programmiersprachen (C, C++, Pascal, ...) besteht darin, dass die Variablen nicht deklariert werden: im bisherigen Umgang mit Scilab haben wir zu keinem Zeitpunkt weder die Größe der Matrix noch den Typ ihrer Elemente (reell, komplex...) angegeben. Es ist der Interpreter, der, wenn er auf ein neues Objekt stößt, diese Aufgabe übernimmt. Dieses Merkmal ist vom Standpunkt der Informatik in Verruf geraten (zurecht, vgl. das alte Fortran), denn es führt zu schwer auffindbaren Fehlern. Im Fall einer Sprache wie Scilab (oder MATLAB) stellt dies nicht so viele Probleme dar, weil die Vektor–/Matrixnotation und die vorhandenen Grundbefehle die Anzahl der Variablen und der Programmzeilen erheblich einzuschränken erlauben (z.B. ermöglichen Matrixanweisungen, viele Schleifen zu vermeiden). Ein weiterer Vorteil dieser Art von Sprache besteht darin, dass sie über Graphikbefehle verfügt (was vermeidet, sich mit einem Graphikprogramm oder einer –bibliothek rumschlagen zu müssen) und darin, dass Scilab ein Interpreter ist (was ermöglicht, ziemlich leicht — ohne auf einen Debugger angewiesen zu sein — Fehler aufzuspüren). Der Nachteil dagegen ist, dass ein in Scilab geschriebenes Programm langsamer (sogar viel langsamer) ist, als wenn man es in C geschrieben hätte (das Programm in C hingegen beansprucht fünfzigmal soviel Zeit zum Schreiben und Austesten). Außerdem ist es im Fall von Anwendungen, in denen die Anzahl der Daten wirklich beträchtlich ist (z.B. 1000×1000–Matrizen), besser, auf eine kompilierte Sprache zurückzugreifen. Ein wichtiger Punkt bezüglich der Schnelligkeit der Ausführung betrifft die Tatsache, dass man so programmieren sollte, dass man möglichst viele der verfügbaren Grundbefehle und Matrixbefehle verwendet (d.h.: die Schleifen werden auf ein Minimum reduziert)[1]. Tatsächlich ruft Scilab in einem solchen Fall eine kompilierte (Fortran–)Routine auf, und so arbeitet sein Interpreter weniger... Um vom Besten beider Welten zu profitieren (d.h. von der Schnelligkeit einer kompilierten Sprache und vom Komfort einer Programmierumgebung wie der von Scilab), hat man die Möglichkeit, Unterprogramme aus Fortran (77) oder C in Scilab einzubinden.

3.1 Schleifen

Es existieren zwei Schleifentypen: die `for`–Schleife und die `while`–Schleife.

3.1.1 Die `for`–Schleife

Die `for`–Schleife iteriert über die Komponenten eines Zeilenvektors:

```
-->v=[1 -1 1 -1]
-->y=0; for k=v, y=y+k, end
```

Die Anzahl der Iterationen ist durch die Anzahl der Komponenten des Zeilenvektors gegeben[2], und in der i–ten Iteration ist der Wert von `k` gleich `v(i)`. Damit die Scilabschleifen den Schleifen der folgenden Art ähneln:

[1] siehe Abschnitt „Hinweise zur effizienten Programmierung in Scilab“

[2] Ist der Vektor eine leere Matrix, dann findet keine Iteration statt.

for $i := i_{beg}$ **to** i_{end} **step** i_{step} **do :**
Folge von Befehlen
end for

reicht es, `i_beg:i_step:i_end` als Vektor zu benutzen, und wenn das Inkrement `i_step` gleich 1 ist, kann es —wie bereits gesehen — weggelassen werden. Die vorige Schleife kann dann ganz einfach in der folgenden Form geschrieben werden:

```
-->y=0; for i=1:4, y=y+v(i), end
```

Einige Bemerkungen:

- Eine Schleife kann auch über eine Matrix iterieren. Die Anzahl der Iterationen ist gleich der Anzahl der Spalten der Matrix und die Schleifenvariable in der i–ten Iteration ist gleich der i-ten Spalte der Matrix. Hier ein Beispiel dazu:

  ```
  -->A=rand(3,3);y=zeros(3,1); for k=A, y = y + k, end
  ```

- Die exakte Syntax sieht folgendermaßen aus:

  ```
  for variable = Matrix, Folge der Befehle, end
  ```

  wobei die Befehle durch Kommata voneinander getrennt sind (oder durch Semikolons, falls man nicht will, dass das Ergebnis der Zuweisungsbefehle am Bildschirm erscheint). In einem Skript (oder einer Funktion) ist der Zeilenende jedoch äquivalent zum Komma, was zu einer Darstellung in der folgenden Form führt:

  ```
  for  variable  =   Matrix
     Befehl 1
     Befehl 2
     ........
     Befehl n
  end
  ```

  Den einzelnen Anweisungen können jeweils Semikolons folgen (auch hier wieder, um die Anzeige am Bildschirm zu unterdrücken)[3].

3.1.2 Die `while`–Schleife

Sie gestattet eine Folge von Befehlen zu wiederholen, solange eine Bedingung wahr ist, z.B.:

```
-->x=1 ; while x<14,x=2*x, end
```

Es sei auf die folgenden Vergleichsoperatoren hingewiesen :

==	gleich
<	kleiner
>	größer
<=	kleiner gleich
>=	größer gleich
~= oder <>	ungleich

und darauf, dass Scilab logische bzw. boolsche Konstanten `%t` oder `%T` für wahr und `%f` oder `%F` für falsch besitzt. Man kann boolsche Matrizen und Vektoren definieren. Die logischen Operatoren sind:

[3]Dies gilt lediglich für ein Skript, denn in einer Funktion wird das Resultat des Zuweisungsbefehls selbst dann nicht angezeigt, wenn dem Befehl kein Semikolon folgt; dieses Standardverhalten kann mit der Anweisung `mode` modifiziert werden.

&	und
|	oder
~	nicht

Die Syntax von `while` sieht folgendermaßen aus:

```
while Bedingung, Befehl_1, ... ,Befehl_N , end
```

oder auch (in einem Skript oder einer Funktion):

```
while Bedingung
   Befehl_1
   ............
   Befehl_N
end
```

wobei jeder `Befehl_k` von einem Semikolon gefolgt sein kann; unter `Bedingung` versteht man einen Ausdruck, der einen boolschen Wert hat.

3.2 Bedingte Anweisungen

Auch hier gibt es zwei Arten: den „if then else“ und den „select case“.

3.2.1 Die „if then else“–Konstruktion

Zunächst ein Beispiel:

```
-->if x>0 then, y=-x,else,y=x,end    // die Variable x muss definiert sein
```

Genauso wie in einem Skript oder einer Funktion, sind hier die Kommata zur Trennung nicht obligatorisch, wenn statt dessen ein Zeilenende steht. Wie in anderen Programmiersprachen kann der `else`–Teil weggelassen werden, wenn nichts geschehen soll in dem Fall, dass die Bedingung falsch ist. Schließlich können die Schlüsselwörter `else` und `if` miteinander verbunden werden, wenn der `else`–Teil mit einem anderen ‚if then else‘ fortfährt, was dann zu folgender Darstellung führt:

```
if Bedingung_1  then
   Folge 1 der Befehle
elseif Bedingung_2  then
   Folge 2 der Befehle
.........................
elseif  Bedingung_N  then
   Folge N der Befehle
else
   Folge N+1 der Befehle
end
```

Genau wie bei der `while`–Schleife ist jede `Bedingung` ein Ausdruck, der einen boolschen Wert zurückgibt.

3.2.2 Die ‚select case‘–Konstruktion (*)

Hier ein Beispiel (mit verschiedenen Werten der Variable `num` zu testen)[4]

```
-->num = 1, select num, case 1, y = 'Fall 1', case 2, y = 'Fall  2',...
-->else, y = 'anderer Fall', end
```

das sich in einem Skript oder in einer Funktion eher folgendermaßen schreibt:

[4]Die Variable `y` ist vom Typ Zeichenkette, vgl. nächster Abschnitt.

```
// man nimmt hier an, dass die Variable num wohldefiniert ist
select  num
case 1 y = 'Fall 1'
case 2 y = 'Fall 2'
else   y = 'anderer Fall'
end
```

Hier testet Scilab nacheinander die Gleichheit der Variable `num` mit den verschiedenen möglichen Fällen (1 oder 2), und sobald Gleichheit besteht, werden die (dem Fall) entsprechenden Befehle ausgeführt, und dann verlässt man die Konstruktion. Das optionale `else` ermöglicht, eine Folge von Befehlen in dem Fall durchzuführen, dass alle vorangegangenen Tests gescheitert sind. Die Syntax dieser Konstruktion sieht folgendermaßen aus:

```
select variable_test
case Ausdruck_1
    Folge 1 von Befehlen
.........................
case Ausdruck_N
    Folge N von Befehlen
else
    Folge N+1 von Befehlen
end
```

wobei das, was `Ausdruck_i` genannt wird, ein Ausdruck ist, der einen Wert zurückgibt, welcher mit dem Wert der Variable `Variable_Test` verglichen wird (in den meisten Fällen sind diese Ausdrücke Konstanten oder Variablen). Tatsächlich ist diese Konstruktion zur folgenden `if`–Konstruktion äquivalent:

```
if variable_test =  Ausdruck_1 then
   Folge 1 von Befehlen
.......................
elseif variable_test = Ausdruck_N then
   Folge N von Befehlen
else
   Folge N+1 von Befehlen
end
```

3.3 Andere Datentypen

Bis jetzt haben wir folgende Datentypen betrachtet:

1. Matrizen (Vektoren und Skalare) aus reellen[5] oder komplexen Zahlen;
2. boolsche Ausdrücke (Matrizen, Vektoren und Skalare);

Es gibt noch andere Datentypen — darunter Zeichenketten und Listen.

3.3.1 Zeichenketten

Im Beispiel zur `select`–Konstruktionen ist die Variable `y` vom Typ Zeichenkette. In Scilab werden sie von Apostrophen oder (englischen) Anführungszeichen umschlossen, und wenn eine Zeichenkette ein solches Zeichen enthält, müssen diese verdoppelt werden. Um der Variablen `ist_doch_klar` die folgende Zeichenkette zuzuweisen:

```
Scilab -- ist's nicht "cool" ?!
```

benutzt man:

[5] ganze Zahlen wurden als Gleitkommazahlen betrachtet

```
-->ist_doch_klar = "Scilab -- ist''s nicht ""cool"" ?!"
```

oder:

```
-->ist_doch_klar = 'Scilab -- ist''s nicht ""cool"" ?!'
```

Man kann auch aus Zeichenketten bestehende Matrizen definieren:

```
-->Ms = ["a" "bc" "def"]
 Ms  =
!a  bc  def  !

-->size(Ms)  // um die Dimensionen zu erhalten
 ans  =
!   1.    3. !

-->length(Ms)
 ans  =
!   1.    2.    3. !
```

Beachten Sie, dass sich `length` nicht genauso verhält wie bei einer Zahlenmatrix: im Fall einer aus Zeichenketten bestehenden Matrix `M` gibt `length(M)` eine Matrix desselben Formats, jedoch mit ganzen Zahlen zurück; die Koeffizienten an der Position (i, j) geben die Anzahl der Zeichen der Zeichenkette an der Position (i, j) an.

Für das Zusammenfügen von Zeichenketten wird einfach der Operator + verwendet:

```
-->s1 = 'abc'; s2 = 'def'; s = s1 + s2
s =
 abcdef
```

und Teilzeichenketten werden mit Hilfe der Funktion `part` erzeugt:

```
-->part(s,3)
ans =
 c

-->part(s,3:4)
ans =
 cd
```

Das zweite Argument der Funktion `part` ist also ein Indexvektor (oder einfach eine ganze Zahl), der die Nummern der Zeichen, welche extrahiert werden sollen, angibt.

3.3.2 Listen (*)

Eine Liste ist einfach eine Aufzählung von Scilab–Objekten (Matrizen oder Skalare, die aus rellen oder komplexen Zahlen oder Zeichenketten bestehen, boolsche Ausdrücke, Listen, Funktionen, ...). Es gibt zwei Arten von Listen: die „einfachen“ und die „typisierten“. Es folgt ein Beispiel für eine einfache Liste:

```
-->L=list(rand(2,2),["Wäre ich doch schon" " mit dieser Übersetzung fertig"],[%t;%f])
 L  =
       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

       L(2)
```

```
!Wäre ich doch schon   mit dieser Übersetzung fertig  !

       L(3)

! T !
! F !
```

Es wurde gerade eine Liste definiert, deren erstes Element eine 2×2–Matrix, das zweite Element ein aus Zeichenketten bestehender Vektor und das dritte ein boolscher Vektor ist. Im Folgenden werden einige Basisoperationen für Listen vorgestellt:

```
-->M = L(1)  // Extrahieren des ersten Eintrags
 M  =
!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

-->L(1)(2,2) = 100;  // Änderung des ersten Eintrags

-->L(1)
 ans  =
!   0.2113249    0.0002211 !
!   0.7560439    100.      !

-->L(2)(4) = " am liebsten noch vor Beginn der Vorlesungszeit!"; // Änderung des
                                                                 // zweiten Eintrags

-->L(2)
 ans  =
!Wäre ich doch schon   mit dieser Übersetzung fertig   am liebsten noch vor Beginn
der Vorlesungszeit!  !

-->L(4)="um einen vierten Eintrag hinzuzufügen"
 L  =
       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

!Wäre ich doch schon  mit dieser Übersetzung fertig   am liebsten, noch vor Beginn
der Vorlesungszeit!  !

       L(3)

! T !
! F !

       L(4)

  um einen vierten Eintrag hinzuzufügen

-->size(L)   // Aus wievielen Elementen besteht die Liste?
 ans  =
    4.
```

```
-->length(L)  // s.o.
 ans  =
    4.

-->L(2) = null()  // Löschen des zweiten Eintrags
 L  =
       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

! T !
! F !

       L(3)

  um einen vierten Eintrag hinzuzufügen

 -->Lbis=list(1,1:3)    // eine andere Liste wird definiert
 Lbis  =
       Lbis(1)

    1.

       Lbis(2)

!   1.    2.    3. !

-->L(3) = Lbis  // das dritte Element von L ist jetzt auch eine Liste
 L  =
       L(1)
 !   1.    2.    3. !

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

! T !
! F !

       L(3)


        L(3)(1)

    1.

        L(3)(2)

!   1.    2.    3. !
```

Wenden wir uns den „typisierten“ Listen zu. Bei diesen Listen ist das erste Element eine Zeichenkette, die den Typnamen dieser Liste angibt (dies erlaubt es, einen neuen Datentyp und für diesen Typ entsprechende Operatoren zu definieren), die folgenden Elemente können irgendwelche Scilab–Objekte sein. Im Grunde kann dieses erste Element auch ein Vektor aus Zeichenketten sein; dessen erstes Element liefert den Typnamen der Liste und die anderen können dazu dienen, die verschiedenen Elemente der Liste zu referenzieren (anstatt ihrer Nummern in der Liste). Ein Beispiel dazu: man will einen Polyeder (dessen Seitenflächen alle dieselbe Anzahl von Kanten haben) darstellen. Zu diesem Zweck speichert man die Koordinaten aller Ecken in einer Matrix (vom Format (3, Eckenanzahl)) ab, die durch die Zeichenkette *Koord* referenziert wird. Dann beschreibt man durch eine Matrix (vom Format (Seitenanzahl, Anzahl Ecken pro Seite und die Ecken jeder Seitenfläche)): für jede Seite werden die Nummern der Ecken, welche diese bilden, in der Reihenfolge angegeben, so dass die Normale gemäß der Rechte–Hand–Regel nach außen weist.

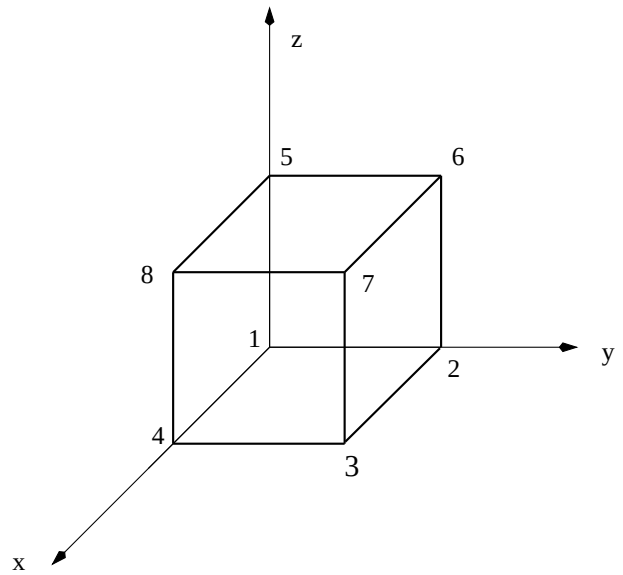

Abbildung 3.1: Nummerierung der Ecken eines Würfels

Dieser Listeneintrag wird durch die Zeichenkette *Seite* referenziert. Für einen Würfel (vgl. Abbildung 3.1) könnte das so aussehen:

```
P=[ 0 0 1 1 0 0 1 1;...     // die Koordinaten der Ecken
    0 1 1 0 0 1 1 0;...
    0 0 0 0 1 1 1 1];

Ecken = [1 5 3 2 1 1;...  // Eckpunkte der Seiten
         2 8 7 6 5 4;...
         3 7 8 7 6 8;...
         4 6 4 3 2 5];
```

Nun muss nur noch eine typisierte Liste gebildet werden, die die ganzen Informationen über den Würfel enthält:

```
-->Wuerfel = tlist(["Polyeder","Koordinaten","Seite"],P,Ecken)
 Wuerfel  =
       Wuerfel(1)

!Polyeder  Koordinaten  Seite  !

       Wuerfel(2)
```

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

       Wuerfel(3)

!   1.    5.    3.    2.    1.    1. !
!   2.    8.    7.    6.    5.    4. !
!   3.    7.    8.    7.    6.    8. !
!   4.    6.    4.    3.    2.    5. !
```

Anstatt die erzeugenden Elemente durch Nummern anzugeben, kann man die entsprechende Zeichenkette benutzen, Beispiel:

```
-->Wuerfel.Koordinaten(:,2)
 ans  =
!   0. !
!   1. !
!   0. !

-->Wuerfel.Seite(:,1)
 ans  =
!   1. !
!   2. !
!   3. !
!   4. !
```

Abgesehen von dieser Besonderheit wird mit typisierten Listen so wie mit anderen umgegangen. Ihr Vorteil besteht darin, dass man die Operatoren `+`, `-`, `/`, `*`, usw. für eine `tlist` vorgegebenen Typs definieren (d.h. überladen) kann (vgl. eine zukünftige Version dieser Dokumentation oder noch besser: die Online–Dokumentation auf der Scilab–Homepage).

3.3.3 Einige Ausdrücke mit boolschen Vektoren und Matrizen (*)

Der boolsche Typ eignet sich für bestimmte Matrixmanipulationen, von denen einige von einer praktischen Kurzschreibweise profitieren. Wenn man reelle Matrizen gleicher Größe mittels eines der Vergleichsoperatoren (`<`, `>`, `<=`, `>=`,`==`, `~=`) miteinander vergleicht, erhält man eine boolsche Matrix gleichen Formats; der Vergleich findet elementweise statt, zum Beispiel:

```
-->A = rand(2,3)
 A  =
!   0.2113249    0.0002211    0.6653811 !
!   0.7560439    0.3303271    0.6283918 !

-->B = rand(2,3)
 B  =
!   0.8497452    0.8782165    0.5608486 !
!   0.6857310    0.0683740    0.6623569 !

-->A < B
 ans  =
! T T F !
! F F T !
```

Ist aber eine der beiden Matrizen ein Skalar, dann ist `A < s` eine Kurzform für `A < s*one(A)`:

```
-->A < 0.5
 ans  =
! T T F !
! F T F !
```

Die boolschen Operatoren lassen sich auch auf diesen Matrixtyp, stets im elementweisen Sinne, anwenden:

```
-->b1 = [%t %f %t]
 b1  =
! T F T !


-->b2 = [%f %f %t]
 b2  =
! F F T !

-->b1 & b2
 ans  =
! F F T !

-->b1 | b2
 ans  =
! T F T !

-->~b1
 ans  =
! F T F !
```

Außerdem gibt es sehr praktische Funktionen, die das Vektorisieren von Tests ermöglichen:

1. `bool2s` transformiert eine boolsche Matrix in eine Matrix gleichen Formats, wobei der logische Wert von `%T` zu 1 und der von `%F` zu 0 transformiert wird:

   ```
   -->bool2s(b1)
    ans  =
   !   1.    0.    1. !
   ```

2. Die Funktion `find` gestattet es, die Indizes der Koeffizienten eines boolschen Vektors (oder Matrix) mit dem Wert `%T` zu finden:

   ```
   -->v = rand(1,5)
    v  =
   !   0.5664249    0.4826472    0.3321719    0.5935095    0.5015342 !

   -->find(v < 0.5)  // v < 0.5 liefert einen boolschen Vektor
    ans  =
   !   2.    3. !
   ```

Eine Anwendung dieser Funktionen wird weiter unten (vgl. Hinweise zur effizienten Programmierung in Scilab) gezeigt. Zu guter Letzt bestehen die Funktionen `and` und `or` jeweils aus dem logischen Produkt (und) und der logischen Addition (oder) aller Elemente einer boolschen Matrix. Man erhält also einen boolschen Skalar. Diese beiden Funktionen lassen ein optionales Argument zu, um die Operation auf den Zeilen oder Spalten vorzunehmen.

3.4 Funktionen

Die übliche Methode, eine Funktion in Scilab zu definieren, besteht darin, sie in eine Datei zu schreiben, die übrigens mehrere Funktionen enthalten kann (sie werden dann beispielsweise thematisch oder nach Art der Anwendung sortiert). Jede Funktion muss mit dem folgenden Befehl beginnen:

```
function [y1,y2,y3,...,yn]=Funktionsname(x1,....,xm)
```

wobei die `xi` Eingabeargumente, die `yj` Ausgabeargumente sind. Darauf folgen die Anweisungen, die zur Funktion gehören und die (letztendlich) aus den Eingabeargumenten die Ausgabeargumente berechnen. Eine Funktion endet mit dem Schlüsselwort `endfunction`. *Bem.:* Es hat sich eingebürgert, den Namen von Dateien, die Funktionen enthalten, mit dem Suffix `.sci` zu versehen. Hier ein erstes Beispiel:

```
function y = fact1(n)
  // die Fakultät:  n muss eine natürliche Zahl sein
  y = prod(1:n)
endfunction
```

Angenommen, man hätte diese Funktion in die Datei `facts.sci`[6] hineingeschrieben. Damit Scilab sie erkennen kann, muss die Datei mit folgendem Befehl geladen werden:

```
getf("facts.sci")    //  oder kürzer  getf facts.sci   oder exec("facts.sci")
```

was genauso wie bei einem Skript mit Hilfe des Menus `File operations` geschehen kann (nach der Auswahl der Datei muss man den Button `getf` anklicken). Man kann diese Funktion auch in der Kommandozeile verwenden (aber auch in einem Skript oder einer anderen Funktion):

```
-->m = fact1(5)
 m  =
    120.

-->n1=2; n2 =3; fact1(n2)
 ans  =
    6.

-->fact1(n1*n2)
 ans  =
    720.
```

Bevor weitere Beispiele vorgestellt werden, bedarf es einiger Präzisierungen des Vokabulars. Beim Schreiben einer Funktion heißen das Ausgabeargument `y` und das Eingabeargument `x` *formale Argumente*. Wenn diese Funktion hinter dem Prompt, in einem Skript oder einer anderen Funktion benutzt wird,

```
arg_s = fact1(arg_e)
```

heißen die Argumente *effektive Argumente*. In der ersten Anwendung ist das effektive Eingabeargument eine Konstante (`5`), in der zweiten eine Variable (`n2`) und in der dritten ein Ausdruck (`n1*n2`). Die Korrespondenz zwischen effektiven und formalen Argumenten (was man Parameterübergabe nennt) kann auf unterschiedliche Art und Weise vollzogen werden (vgl. nächsten Abschnitt zur Präzisierung der Parameterübergabe in Scilab).

Als zweites Beispiel nehmen wir die Lösung einer quadratischen Gleichung:

```
function [x1,x2] = Quad_Gleichung(a, b, c)
   // berechne die Wurzeln von   a x^2 + b x + c = 0
   // a, b et c  können reell oder komplex sein,  a  muss ungleich 0 sein
   delta = b^2 - 4*a*c
```

[6] gemäß Konvention haben Dateien, die Funktionen enthalten, die Endung `.sci` (und Skripte die Endung `.sce`)

```
      x1 = (-b - sqrt(delta))/(2*a)
      x2 = (-b + sqrt(delta))/(2*a)
   endfunction
```

Hier drei Versuche mit dieser Funktion:

```
-->[r1, r2] = Quad_Gleichung(1,2,1)
 r2  =
  - 1.
 r1  =
  - 1.

-->[r1, r2] = Quad_Gleichung(1,0,1)
 r2  =
    i
 r1  =
  - i

-->Quad_Gleichung(1,0,1)  // Aufruf ohne Zuweisung
 ans  =
  - i
```

Man beachte, dass in dem dritten Aufruf nur eine Wurzel (die erste) zurückgegeben wurde. Das ist normal, da man den Wert des Funktionsaufrufes nicht einer Variablen zugewiesen hat, im Gegensatz zu zwei Rückgabewerten bei den ersten beiden Aufrufen. In diesem Falle benutzt Scilab standardmäßig die Variable `ans`, um (nur das erste) Ergebnis des Funktionsaufrufes zu speichern (der zweite geht verloren).

Hier ein drittes Beispiel: es handelt sich um die Auswertung des Polynoms an der Stelle t, das in der Newtonbasis dargestellt ist (*Bem.:* mit $x_i = 0$ erhält man die kanonische Basis):

$$p(t) = c_1 + c_2(t - x_1) + c_3(t - x_1)(t - x_2) + \ldots + c_n(t - x_1)\ldots(t - x_{n-1}).$$

Indem man die gemeinsamen Faktoren ausnutzt und von rechts nach links rechnet (hier mit $n = 4$):

$$p(t) = c_1 + (t - x_1)(c_2 + (t - x_2)(c_3 + (t - x_3)(c_4))),$$

erhält man den Horner–Algorithmus:

(1) $p := c_4$

(2) $p := c_3 + (t - x_3)p$

(3) $p := c_2 + (t - x_2)p$

(4) $p := c_1 + (t - x_1)p.$

Durch Verallgemeinerung auf beliebiges n und den Einsatz einer Schleife, erhält man in Scilab:

```
function p=myhorner(t,x,c)
  // Auswerten des Polynoms c(1) + c(2)*(t-x(1)) + c(3)*(t-x(1))*(t-x(2)) +
  //                          ... + c(n)*(t-x(1))*...*(t-x(n-1))
  // mit dem Horner-Algorithmus
  n=length(c)
  p=c(n)
  for k=n-1:-1:1
     p=c(k)+(t-x(k))*p
  end
endfunction
```

Sind die Vektoren `coef` und `xx` und die reelle Zahl `tt` in der Aufrufumgebung der Funktion wohldefiniert (hat der Vector `coef` m Komponenten, dann muss `xx` mindestens $m-1$ Komponenten haben, wenn es keine Probleme geben soll...), dann weist die Anweisung

```
val = myhorner(tt,xx,coef)
```

der Variable `val` den Wert

$$coef_1 + coef_2(tt - xx_1) + \cdots + coef_m \prod_{i=1}^{m-1} (tt - xx_i)$$

bis auf nummerische Rundungsfehler zu. Zur Erinnerung: Der Befehl `length` gibt das Produkt aus den zwei Dimensionen einer (Zahlen–)Matrix zurück, und folglich im Fall eines (Zeilen– bzw. Spalten)Vektors seine Komponentenzahl. Dieser Befehl (und der Befehl `size`, der die Zeilen– und Spaltenanzahl zurückgibt) erlaubt es, dass man die Dimension der Datenstrukturen (Matrizen, Listen, ...) nicht als Argumente einer Funktion zu übergeben braucht.

3.4.1 Parameterübergabe (*)

Seit der Scilab–Version 2.4 werden die Eingabevariablen per Referenz übergeben; werden diese doch in der Funktion verändert, so wird zuvor eine Kopie erstellt (die Eingabeparameter können also nicht modifiziert werden). Man kann dies benutzen, um gewisse die Abarbeitung gewisser Funktionen zu beschleunigen. Das folgende konstruierte Beispiel zeigt deutlich die Kosten der Parameterübergabe bei Parametern mit großem Speicherbedarf:

```
function [w] = toto(v, k)
  w = 2*v(k)
endfunction

function [w] = titi(v, k)
  v(k) = 2*v(k)
  w = v(k)
endfunction

// Testskript
m = 200000;
nb_rep = 2000;
x = rand(m,1);
timer(); for i=1:nb_rep; y = toto(x,300); end; t1=timer()/nb_rep
timer(); for i=1:nb_rep; y = titi(x,300); end; t2=timer()/nb_rep
```

Auf meiner Maschine ergibt sich:

```
 t1  =  0.00002
 t2  =  0.00380
```

Am Ende eines Funktionsaufrufes werden alle lokalen Variablen zerstört. Ein anderer Punkt: In einer Funktion haben Sie (nur lesenden) Zugriff auf alle Variablen von höheren Niveaus, d.h. alle Variablen, die vor dem Aufruf dieser Funktion außerhalb aller Funktionen oder in einer Funktion der Aufrufskette bisher angelegt worden sind. Man könnte also lesend globale Variablen benutzen, obwohl dies nicht empfehlenswert ist. Wenn Sie dagegen eine globale Variable verändern, wird eine neue interne Variable (in der Funktion) erzeugt; die Variable mit gleichem Namen vom höheren Niveau wird nie verändert.

3.4.2 Debuggen einer Funktion

Um eine Funktion zu debuggen, kann man als erstes die Funktion `disp(v1,v2, ...)` benutzen, die es ermöglicht, den Wert der Variablen `v1`, `v2`, ... in *umgekehrter* Reihenfolge zu visualisieren (aus diesem

Grund wurde die Zeichenkette `'c = '` in der Anweisung `disp(c,'c = ')` aus dem vorigen Beispiel an die zweite Stelle gesetzt). Im zweiten Schritt können Sie eine oder mehrere `pause`–Anweisungen an strategische Stellen der Funktion stellen. Wenn Scilab auf diese Anweisung trifft, hält die Ausführung des Programms an, und Sie können den Wert aller bereits definierten Variablen aus dem Scilab–Fenster heraus (der Scilab–Prompt `-->` wandelt sich in `-1->` um) überprüfen. Wenn Ihre Betrachtungen abgeschlossen sind, setzt das Kommando `resume` die unterbrochene Ausführung der Anweisung fort (evtl. bis zur nächsten `pause`).

Da es mühsam ist, in allen zu debuggenden Funktionen `pause` Anweisungen einzufügen, gibt es eine Möglichkeit, Haltepunkte (englisch breakpoint) mit der Anweisung

`setbpt(`*Funktions_Name* `[`, *Zeilennummer* `])`

anzugeben, bei der man den Namen (als Zeichenkette) der Funktion und ggf. in dieser eine Zeilennummer (voreingestellt ist 1) angibt, wo die Funktion angehalten werden soll. Wenn Scilab bei der Ausführung auf einen solchen Haltepunkt trifft, so gibt es die Zeilennummer aus und verhält sich genauso, als ob dort eine `pause` Anweisung *hinter* dieser Zeile gestanden hätte. Nachdem man sich gewisse Variablen hat anzeigen lassen, kann man mit `resume` die Ausführung der Funktion fortsetzen. Ein solcher Haltepunkt kann explizit wieder mit:

`delbpt(`*Funktions_Name* `[`, *Zeilennummer* `])`

entfernt werden. Gibt man keine Zeilennummer an, so werden alle Haltepunkte in dieser Funktion gelöscht. Mittels `dispbpt()` kann man sich alle Haltepunkte anzeigen lassen. Dies soll anhand der Funktion `Quad_Gleichung` von früher demonstriert werden.

```
function [x1,x2] = Quad_Gleichung(a, b, c)
   // berechne die Wurzeln von   a x^2 + b x + c = 0
   // a, b et c  können reell oder komplex sein,  a  muss ungleich 0 sein
   delta = b^2 - 4*a*c
   x1 = (-b - sqrt(delta))/(2*a)
   x2 = (-b + sqrt(delta))/(2*a)
endfunction
```

Wir nehmen an, dass diese Funktion in Scilab geladen ist (mittels `getf` oder `exec`):

```
-->setbpt("Quad_Gleichung")   // ein erster Haltepunkt
-->[r1,r2] = Quad_Gleichung(1,2,7)  // Aufruf => Haltepunkt
Stop after row     1 in function Quad_Gleichung:

-1->a    // einige Variable werden überprüft
 a  =
    1.

-1->b
 b  =
    2.

-1->dispbpt()   // Anzeige der Haltepunkte
 breakpoints of  function :Quad_Gleichung
          1

-1->setbpt("Quad_Gleichung",5)  // Hinzufügen eines weiteren Haltepunkte in Zeile 5

-1->x1  // x1 ist noch nicht definiert !
x1
  !--error     4
undefined variable : x1

-1->resume   // Fortsetzung bis zum nächsten Haltepunkt
Stop after row     5 in function Quad_Gleichung :
```

```
-1->x1     // jetzt kann man x1 anzeigen lassen
 x1  =
   - 1. - 2.4494897i

-1->x2    // x2 noch nicht definiert (erst in Zeile 6 dieser Funktion)
x2
  !--error     4
undefined variable : x2

-1->dispbpt()   // Anzeige der beiden Haltepunkte
 breakpoints of  function :Quad_Gleichung
         1
         5

-1->resume   // man fährt fort und verlässt damit die Funktion
 r2  =
   - 1. + 2.4494897i
 r1  =
   - 1. - 2.4494897i

-->delbpt("Quad_Gleichung")  // Entfernen aller Haltepunkte
```

3.4.3 Der Befehl `break`

Er erlaubt es, eine `for`– oder `while`–Schleife abzubrechen, indem die Kontrolle an die Anweisung übergeben wird, die dem `end` folgt, welches das Ende der Schleife markiert[7]. Er kann dazu dienen, andere Schleifentypen zu simulieren, solche mit einer Abbruchbedingung am Ende (wie etwa `repeat ... until` in Pascal) und solche mit einer Abbruchbedingung in der Mitte (arg...) oder dazu, Ausnahmefälle zu behandeln, die den normalen Ablauf der `for`– oder `while`–Schleife (z.B. ein Pivotelement fast Null im Gaußalgorithmus) nicht erlauben. Angenommen, man will eine Schleife mit der Abbruchbedingung am Ende simulieren:

```
repeat
  Folge von Befehlen
until Bedingung
```

wobei `Bedingung` ein Ausdruck ist, der einen boolschen Wert zurückgibt (man verlässt die Schleife, wenn die Abbruchbedingung wahr ist). Man kann also in Scilab schreiben:

```
while %t    // Anfang der Schleife
  Folge von Befehlen
  if Bedingung then, break, end
end
```

Es gibt auch Fälle, in denen das Benutzen von `break` zu einer natürlichen, lesbaren und kompakten Lösung führt. Dazu ein Beispiel: man will in einem aus Zeichnketten bestehenden Vektor nach Indizes des ersten Wortes suchen, welches mit dem Buchstaben l anfängt. Dafür wird eine Funktion geschrieben (die 0 zurückgibt, wenn keine der Zeichenketten mit dem betreffenden Buchstaben beginnt). Das Benutzen der `while`–Schleife (ohne also `break` zu verwenden) kann zur folgenden Lösung führen:

```
function ind = Suche2(v,l)
  n = max(size(v))
  i = 1
  gefunden = %f
  while ~gefunden & (i <=  n)
     gefunden = part(v(i),1)  == l
     i =  i + 1
```

[7] wenn die Schleife in einer anderen verschachtelt ist, erlaubt `break` lediglich die innere Schleife zu verlassen

```
  end
  if gefunden then
     ind = i-1
  else
     ind = 0
  end
endfunction
```

Greift man auf `break` zurück, erhält man folgende natürliche Lösung (die aber wenig konform mit den harten Regeln der reinen strukturierten Programmierung ist):

```
function ind  = Suche1(v,l)
 n   = max(size(v))
 ind =  0
 for i=1:n
    if part(v(i),1) == l then
       ind = i
       break
    end
 end
endfunction
```

Zur Erinnerung: Man sollte die Funktion `size` benutzen, auch wenn die Funktion `length` auf einen Vektor[8] zugeschnitten zu sein scheint; das kommt daher, dass `length` anders reagiert, wenn die Einträge der Matrix oder des Vektors Zeichenketten sind (`length` gibt eine Matrix gleicher Grösse zurück, wobei jeder Eintrag die Anzahl der Zeichen der entsprechenden Zeichenkette angibt).

3.4.4 Einige nützliche Grundbefehle für Funktionen

Abgesehen von `length` und `size`, die ermöglichen, die Dimensionen der Datenstrukturen herauszufinden, sowie von `pause`, `resume`, `disp` zum Debuggen, können auch andere Funktionen wie `error`, `warning`, `argn` oder auch `type` und `typeof` nützlich sein.

Die Funktion `error`

Sie ermöglicht, den Ablauf einer Funktion abrupt zu unterbrechen und gleichzeitig eine Fehlermeldung anzuzeigen; die folgende Funktion berechnet die Fakultät $n!$, wobei darauf geachtet wird, dass $n \in \mathbb{N}$ gilt:

```
function f = fact2(n)
 // Berechnung der Fakultät einer positiven ganzen Zahl
 if (n - floor(n) ~=0) | n<0 then
    error('Fehler in fact2: das Argument muss eine natürliche Zahl sein')
 end
 f = prod(1:n)
endfunction
```

und hier das Ergebnis für zwei Argumente:

```
-->fact2(3) ans =
    6.

-->fact2(0.56)

 !--error 10000 Fehler in fact2: das Argument muss eine natürliche Zahl sein
 at line 6 of function fact called by : fact(0.56)
```

[8] Wenn man einen Code will, der unabhängig davon funktioniert, ob `v` ein Zeilen– oder Spaltenvektor ist, kann man nicht mehr `size(v,'r')` oder `size(v,'l')` benutzen, daher `max(size(v))`.

Die Funktion `warning`

Sie lässt eine Meldung am Bildschirm erscheinen, ohne den Ablauf der Funktion zu unterbrechen:

```
function f = fact3(n)
 // Berechnung der Fakultät einer positiven Zahl
 if (n - floor(n) ~=0) | n<0  then
    n = floor(abs(n))
    warning('das Argument ist keine natürliche Zahl: es wird '+sprintf("%d",n)+"!"
    berechnet')
 end
 f = prod(1:n)
endfunction
```

was zum Beispiel liefert:

```
-->fact3(-4.6)
WARNING:das Argument ist keine natürliche Zahl: es wird 4! berechnet
 ans  =
    24.
```

Wie bei der Funktion `error` ist auch das einzige Argument der Funktion `warning` eine Zeichenkette. Hier wurde eine Verkettung von drei Zeichenketten benutzt, wobei eine der Zeichenketten mit Hilfe der Funktion `sprintf` erhalten wurde, welche die Umwandlung von Zahlen in Zeichenketten gemäß einem bestimmten Format gestattet.

Die Funktionen `type` und `typeof`

Diese erlauben den Typ einer Variablen `v` zu erkennen. `type(v)` gibt eine ganze Zahl zurück, während es sich bei `typeof(v)` um eine Zeichenkette handelt. Es folgt eine Tabelle, die alle bereits bekannten Datentypen zusammenfasst:

Typ von `v`	`type(v)`	`typeof(v)`
Matrix aus rellen oder komplexen Zahlen	1	constant
boolsche Matrix	4	boolean
Matrix aus Zeichenketten	10	string
Liste	15	list
typisierte Liste	16	Typ der Liste
Funktion	13	function

Ein Anwendungsbeispiel: will man seine Fakultäts–Funktion im Falle eines Aufrufs mit einem ungeeigneten Eingabeargument absichern, so kann man z.B. zo vorgehen:

```
function f = fact4(n)
 // eine (gegen Fehleingaben) ein bisschen abgeschirmtere Fakultäts-Funktion
 if type(n) ~= 1 then
   error("Fehler in fact4 : das Argument hat den falschen Typ\dots")
 end
 [nr,nc]=size(n)
 if (nr ~= 1) | (nc ~= 1) then
   error("Fehler in fact4 : das Argument darf keine Matrix sein\dots")
 end
 if (n - floor(n) ~=0) | n<0  then
    n = floor(abs(n))
    warning('Das Argument ist keine natürliche Zahl: es wird '+sprintf("%d",n)
```

```
            +"! berechnet")
 end
 f = prod(1:n)
endfunction
```

Die Funktion `argn` und optionale Parameter

`argn` gestattet es, die Anzahl der aktuellen Eingabe– und Ausgabeargumente einer Funktion bei deren Aufruf zu bestimmen. Man benutzt sie in der Form: :

```
[lhs,rhs] = argn()
```

`lhs` (für left hand side) liefert die Anzahl der aktuellen Ausgabeargumente, und `rhs` (für right hand side) liefert die Anzahl der aktuellen Eingabeargumente.

Sie erlaubt es, im Wesentlichen eine Funktion mit optionalen Eingabe– und Ausgabeargumenten zu schreiben (die Funktionen `type` oder `typeof` können auch hier hilfreich sein). Ein Anwendungsbeispiel wird weiter unten gegeben (Eine Funktion ist eine Scilabvariable).

Um auf einfache Weise Funktionen mit optionalen Parametern zu schreiben, besitzt Scilab eine nützliche Funktionalität, um die Zuordnung von formalen und effektiven Argumenten zu bestimmen. In dem folgenden Beispiel hat die Funktion einen klassischen und zwei optionale Parameter:

```
function [y] = argopt(x, coef_mult, coef_add)
   // zur Illustration der Zuordnung von formalen und effektiven Argumenten
   [lhs, rhs] = argn()
   if rhs < 1 | rhs > 3 then
      error("falsche Anzahl von Parametern")
   end

   if ~exists("coef_mult","local") then
      coef_mult = 1
   end

   if ~exists("coef_add","local") then
      coef_add = 0
   end

   y = coef_mult*x + coef_add
endfunction
```

Mit Hilfe der Funktion `exists` testet man, ob die Parameter `coef_mult` und `coef_add` definiert sind[9]. Falls nicht, so kann man diese Parameter mit Voreinstellungen versehen. Der Vorteil der Syntax *formaler_Parameter = effektiver_Parameter* liegt darin, dass nun die Reihenfolge der Angabe dieser Parameter keine Rolle mehr spielt. Ein Beispiel:

```
-->y1 = argopt(5)
 y1  =
    5.

-->y2 = argopt(5, coef_add=10)
 y2  =
    15.

-->y3 = argopt(5, coef_mult=2, coef_add=6)
 y3  =
    16.

-->y4 = argopt(5, coef_add=6, coef_mult=2) // y4 sollte gleich y3 sein
 y4  =
    16.
```

[9]Der zusätzliche Parameter schränkt den Test auf in der Funktion definierte Namen ein; dies ist wichtig, falls es in einer Funktion der Aufrufkette eine Variable gleichen Namens gibt.

3.5 Diverse Ergänzungen

3.5.1 Länge eines Bezeichners

Scilab berücksichtigt nur die 24 ersten Buchstaben eines jeden Bezeichners:

```
-->a2345678901234567890123345 = 1
 a234567890123456789012345  =
    1.

-->a2345678901234567890123346 = 2
 a234567890123456789012345  =
    2.
```

Man kann zwar mehr Buchstaben verwenden, aber nur die 24 ersten sind von Bedeutung.

3.5.2 Priorität der Operatoren

Sie ist recht natürlich (aus diesem Grund erscheinen diese Regeln womöglich nicht in der Online–Hilfe...). Wie üblich haben die nummerischen Operatoren[10] (`+ - * .* / ./ \ ^ .^ '`) eine höhere Priorität als die Vergleichsoperatoren (`< <= > >= == ~=`). Die untere Übersicht fasst die nummerischen Operatoren nach absteigender Priorität geordnet zusammen:

'
^ .^
* .* / ./ \
+ -

Im Fall boolscher Operatoren hat „nicht“ (~) Vorrang vor dem „und“ und dieses Vorrang vor dem „oder“ (|). Erinnert man sich nicht mehr an die Priorität, dann setzt man runde Klammern, was übrigens beim Lesen eines Ausdrucks, der aus mehreren Termen besteht, helfen kann (mit Leerzeichen dazwischen)... Für weitere Details siehe „Scilab Bag Of Tricks“. Einige Bemerkungen:

1. Wie in den meisten Sprachen, ist das unäre - nur am Anfang eines Ausdrucks zugelassen, d.h. dass die Ausdrücke des folgenden Typs (wobei *op* für einen beliebigen nummerischen Operator steht):

   Operand op - Operand

   nicht erlaubt sind. Man muss also Klammern benutzen:

   Operand op (- Operand)

2. Für einen Ausdruck der Art

   Operand op1 Operand op2 Operand op3 Operand

   in dem die Operatoren dieselbe Priorität haben, geschieht die Auswertung i.A. von links nach rechts:

   ((Operand op1 Operand) op2 Operand) op3 Operand

   Die Ausnahme bildet der Potenzoperator:

   `a^b^c` wird von rechts nach links ausgewertet: `a^(b^c)`

   auf die Art und Weise also, wie sie der mathematischen Konvention entspricht: a^{b^c}.

[10] es gibt noch weitere, die nicht in dieser Liste aufgeführt sind

3. Im Gegensatz zu C geschieht die Auswertung boolscher Ausdrücke der folgenden Form:

$$a \text{ oder } b$$
$$a \text{ und } b$$

zunächst durch die Auswertung der boolschen Unterausdrücke a und b , bevor Scilab zu „oder“ im ersten und „und “ im zweiten Fall schreitet (im Fall, dass a ‚wahr‘ für den ersten (bzw. ‚falsch‘ für den zweiten) Fall zurückgibt, hätte Scilab eigentlich auf die Auswertung von dem boolschen Ausdruck b verzichtet können). Dies untersagt bestimmte Kurzschreibweisen, die in C verwendet werden. Der Test in der folgenden Schleife beispielsweise:

```
while i>0 & temp < v(i)
  v(i+1) = v(i)
  i = i-1
end
```

(wobei `v` ein Vektor und `temp` ein Skalar ist), würde einen Fehler für $i = 0$ erzeugen, weil der zweite Ausdruck `temp < v(i)` trotzdem ausgewertet wird (die Vektorindizes fangen stets mit 1 an!).

3.5.3 Rekursivität

Eine Funktion kann sich selbst aufrufen. Im Folgenden werden zwei elementare Beispiele vorgestellt (das zweite demonstriert eine schlechte Anwendung der Rekursivität):

```
function f=fact(n)
   // Fakultät in der rekursiven Variante
   if n <= 1 then
      f = 1
   else
      f = n*fact(n-1)
   end
endfunction

function f=fib(n)
   // Berechnung des n-ten Gliedes der Fibonnaci-Folge :
   // fib(0) = 1, fib(1) = 1, fib(n+2) = fib(n+1) + fib(n)
   if n <= 1 then
      f = 1
   else
      f = fib(n-1) + fib(n-2)
   end
endfunction
```

3.5.4 Eine Funktion ist eine Scilabvariable

Eine Funktion, die in der Scilabsprache programmiert ist[11], ist eine Variable vom Typ ‚Funktion‘, und sie kann insbesondere als Argument einer anderen Funktion übergeben werden. An dieser Stelle ein kleines Beispiel dazu[12]: Öffnen Sie eine Datei, um die folgenden zwei Funktionen zu schreiben:

```
function y = f(x)
  y = sin(x).*exp(-abs(x))
endfunction

function Zeichne_Funktion(a, b, funktion, n)
```

[11] siehe den Abschnitt „Scilab–Grundbefehle und –Funktionen“ im Kapitel „Fallstricke“
[12] das bis auf seinen pädagogischen Wert nutzlos ist: vgl. `fplot2d`

```
    // n ist ein optionales Argument; im Fall, dass es fehlt, wird n=61 gesetzt
    [lhs, rhs] = argn()
    if rhs == 3 then
       n = 61
    end
    x = linspace(a,b,n)
    y = funktion(x)
    plot(x,y)
  endfunction
```

Dann geben Sie diese Funktionen in der Umgebung ein und probieren schließlich folgendes aus:

```
-->Zeichne_Funktion(-2*%pi,2*%pi,f)
-->Zeichne_Funktion(-2*%pi,2*%pi,f,21)
```

Eine interessante Möglichkeit, die Scilab bietet, besteht darin, dass man eine Funktion mit Hilfe des Kommandos `deff` direkt in der Umgebung (ohne sie in eine Datei schreiben und dann mit `getf` laden zu müssen) mit folgendermaßen Syntax definieren kann:

```
deff('[y1,y2,...]=Name_der_Funktion(x1,x2,....)',text)
```

wobei `text` ein aus Zeichenketten bestehender Spaltenvektor ist, der die aufeinanderfolgenden Anweisungen einer Funktion darstellt. Man hätte beispielsweise Folgendes verwenden können:

```
deff('y=f(x)','y = sin(x).*exp(-abs(x))')
```

um die erste der beiden obigen Funktionen zu definieren. Tatsächlich ist diese Möglichkeit in mehreren Fällen interessant:

1. in einem Skript, das eine einfache Funktion benutzt, die ziemlich oft modifiziert werden muss; in diesem Fall kann man die Funktion mit `deff` in dem Skript definieren, was das Hantieren mit einer anderen Datei vermeidet, in der man sonst in gewohnter Weise die Funktion definieren würde; **man kann seit Version 2.6 jedoch in einem Skript mehrere Funktionen definieren**, was viel praktischer ist, als mit der Funktion `deff`. In einem Skript kann man nun schreiben:

   ```
   // Definition der in diesem Skript benutzten Funktionen
   function
      ....
   endfunction
   function
      ....
   endfunction
   // Ende des Skripts
   ```

2. Die wirklich interessante Möglichkeit besteht darin, dass man einen Teil des Codes in dynamischer Weise definieren kann: Man erstellt eine Funktion mit Hilfe verschiedener Elemente, die aus vorherigen Berechnungen und/oder einer Eingabe von Daten (mittels einer Datei oder interaktiv (vgl. Dialogfenster)) hervorgegangen sind. In diesem Sinn siehe auch die Funktionen `evstr` und `execstr` ein bisschen weiter unten.

3.5.5 Dialogfenster

Im Beispiel zum Skript aus dem Kapitel 2 haben wir die Funktion `input` gesehen, die erlaubt, einen Parameter über das Scilabfenster interaktiv einzugeben. Zum anderen ermöglicht die Funktion `disp` die Anzeige von Variablen am Bildschirm (stets im Scilabfenster). Tatsächlich gibt es eine Reihe von Funktionen, die das Anzeigen von Dialogfenstern, Menus und der Dateienauswahl gestatten: `x_choices`, `x_choose`, `x_dialog`, `x_matrix`, `x_mdialog`, `x_message` und `xgetfile`. Siehe `Help` zu Details zu diesen Funktionen (die Hilfe zu einer Funktion enthält immer mindestens ein Beispiel).

3.5.6 Umwandlung einer Zeichenkette in einen Scilabausdruck

Es ist oft nützlich, einen Scilabausdruck, der in Form einer Zeichenkette gegeben ist, auswerten zu können. Zum Beispiel gibt die Mehrzahl der vorigen Funktionen Zeichenketten zurück, was sich gleichermaßen als praktisch erweist, um Zahlen zurückzugeben, weil man auch Scilabausdrücke verwenden kann (z.B. `sqrt(3)/2, 2*%pi`, ...). Der Befehl, der diese Umwandlung zulässt, heißt `evstr`, beispielsweise:

```
-->c = "sqrt(3)/2"
 c  =
 sqrt(3)/2

-->d = evstr(c)
 d  =
    0.8660254
```

In der Zeichenkette können Sie bereits definierte Scilabvariablen benutzen:

```
-->a = 1;
-->b=evstr("2 + a")
 b  =
    3.
```

und diese Funktion ist auch auf eine Matrix aus Zeichenketten anwendbar[13]:

```
-->evstr(["a" "2" ])
 ans  =
!   1.    2. !

-->evstr([" a + [1 2]" "[4 , 5]"])
 ans  =
!   2.    3.    4.    5. !

-->evstr(["""a""" """b"""])   // Umwandlung einer Zeichenkette in eine Zeichenkette
 ans  =
!a  b  !
```

Es gibt auch eine Funktion `execstr`, die gestattet, einen in Form einer Zeichenkette gegebenen Scilabbefehl auszuführen:

```
-->execstr("A=rand(2,2)")

-->A
 A  =
!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !
```

3.6 Ein– und Ausgabe in Dateien oder das Scilab Fenster

Scilab besitzt zwei Ein–/Ausgabesysteme (FORTRAN–ähnlich oder C–ähnlich). Man sollte unbedingt vermeiden, beide Systeme bei der Ein–/Ausgabe auf ein und dieselbe Datei zu mischen.

[13] und auch auf eine Liste; sehen Sie unter Help nach

3.6.1 FORTRAN–ähnliche Ein– und Ausgabe

Im zweiten Kapitel haben wir gesehen, wie man eine reelle Matrix mit Hilfe einer einzigen Anweisung `read` bzw. `write` aus einer Datei lesen bzw. schreiben kann. Genauso ist es möglich, einen aus Zeichenketten bestehenden Spaltenvektor zu schreiben und zu lesen:

```
-->v = ["Scilab is free";"Octave is free";"Matlab is ?"];

-->write("toto.dat",v,"(A)")  // sehen Sie sich den Inhalt der Datei toto.dat an

-->w = read("toto.dat",-1,1,"(A)")
 w  =
!Scilab is free  !
!                !
!Octave is free  !
!                !
!Matlab is ?     !
```

Was die Schreibweise betrifft, so fügt man `write` einfach ein drittes Argument hinzu, was dem Fortran–Format entspricht: es besteht aus einer Zeichenkette, die einen (oder mehrere) Ausgabebezeichner (getrennt durch Kommata, falls es mehrere gibt) enthält, der von runden Klammern eingeschlossen ist: `A` besagt, dass man eine Zeichenkette schreiben will. Bezüglich des Lesens ist zu berücksichtigen, dass die zweiten und dritten Argumente jeweils der Anzahl der Zeilen (-1 um bis ans Ende der Datei zu gelangen) und Spalten (hier 1) entsprechen. Für reelle Matrizen können sie übrigens ebenso ein Format hinzufügen (eher beim Schreiben), um präzise die Art und Weise kontrollieren zu können, in der die Daten geschrieben werden.

Im Großen und Ganzen sind die Möglichkeiten von Scilab in diesem Bereich exakt dieselben wie diejenigen von Fortran 77; Sie können also ein Buch über diese Sprache lesen, falls Sie mehr erfahren möchten[14]. Im Folgenden werden einige Beispiele gegeben, die einzig und allein Textdateien im sequentiellen Zugriff benutzen.

Eine Datei öffnen

Dies erreicht man mit der Anweisung `file`, deren (vereinfachte) Syntax so lautet:

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

wobei

- `file-name` eine Zeichenkette ist, die den Namen der Datei enthält (der eventuell der Pfad vorangestellt ist, welcher zu dieser Datei führt, falls sie sich nicht in dem Verzeichnis befindet, auf welches Scilab weist, dieses Verzeichnis lässt sich mit dem Befehl `chdir` wechseln;
- `status` eine der folgenden Zeichenketten ist:
  - `"new"`, um eine neue Datei zu öffnen (falls diese bereits existiert, wird eine Fehlermeldung ausgelöst);
  - `"old"`, um eine existierende Datei zu öffnen (falls diese nicht existiert, wird ebenfalls ein Fehler gemeldet);
  - `"unknow"`, um eine neue Datei zu erzeugen, falls noch keine existiert; andernfalls wird die entsprechende Datei geöffnet;

  Im Falle, dass `status` nicht angegeben ist, benutzt Scilab `"new"` (dies ist der Grund, dass das Schreiben in eine Datei mittels einer einzigen Anweisung `write` versagt, falls die Datei bereits existiert).

[14] Sie könen kostenlos das Buch von Clive Page auf dem ftp–Server `ftp.star.le.ac.uk` erhalten: es befindet sich im Verzeichnis `/pub/fortran` in der Datei `prof77.ps.gz`

- `unit` eine ganze Zahl ist, die es ermöglicht, die Datei im weiteren Verlauf bei Lese–/Schreibvorgängen zu identifizieren (mehrere Dateien können gleichzeitig geöffnet sein).

- Ein Fehler beim Öffnen einer Datei kann erkannt werden, wenn das Argument `err` vorhanden ist; andernfalls behandelt Scilab den Fehler als fatal. Eine fehlerfreie Ausführung entspricht dem Wert 0; wenn also dieser Wert ungleich 0 ist, gibt die Anweisung `error(err)` eine Fehlermeldung zurück, die uns mehr über den Fehler erfahren lässt: häufig erhält man err=240, was bedeutet:

```
-->error(240)
          !--error   240
File error(240) already exists or directory write access denied
```

  Will man interaktiv eine Datei auswählen, benutze man `xgetfile`, was ein Navigieren durch den Verzeichnisbaum ermöglicht.

Schreiben und Lesen in einer geöffneten Datei

Angenommen, man hätte mit Erfolg eine Datei geöffnet: auf diese bezieht man sich mit der ganzen Zahl `unit`, welche durch `file` zurückgegeben wurde. Existiert die Datei bereits, findet das Lesen und Schreiben normalerweise am Anfang der Datei statt. Will man jedoch ans Ende der Datei schreiben, muss man zuvor mit dem Befehl `file("last", unit)` dorthin positionieren; falls Sie aus irgendeinem Grund wieder zum Anfang der Datei gelangen wollen, verwenden Sie `file("rewind", unit)`.

Hier ein erstes Beispiel: man will eine Datei schreiben, die eine Liste von Kanten in der Ebene beschreibt, d.h. wenn man n Punkte $P_i = (x_i, y_i)$ und m Kanten hat, kann jede Kante als ein Segment $\overrightarrow{P_iP_j}$ beschrieben werden, indem man einfach die Nummer (aus der Tabelle der Punkte) des Anfangs– und Endpunktes angibt (hier i bzw. j). Für diese Datei wählt man das folgende Format:

```
eine Textzeile
n
x_1 y_1
.......
x_n y_n
m
i1 j1
.....
im jm
```

Die Textzeile enthält Anmerkungen. Danach gibt eine ganze Zahl Aufschluss über die Anzahl der Punkte. Als Nächstes werden die Koordinaten dieser Punkte angegeben. Die Zeile darauf informiert über die Anzahl der Kanten und wieder die nächste über die Endpunkte jeder Kante. Angenommen, unsere Textzeile wäre in der Variablen `text`, die Punkte in der Matrix `P` vom Format $(n, 2)$ und schließlich die Endpunkte der Kanten in der Matrix `Endpunkte` vom Format $(m, 2)$ enthalten, dann würde das Schreiben der Datei mittels folgender Anweisungen erfolgen:

```
write(unit,text)              // Schreiben der Textzeile
write(unit,size(P,1))         // Schreiben der Punkteanzahl
write(unit,P)                 // Schreiben der Koordinaten der Punkte
write(unit,size(Endpunkte,1)) // Schreiben der Kantenanzahl
write(unit,Endpunkte)         // Schreiben der Endpunkte
file("close",unit)            // Schließen der Datei
```

und man erhielte dieses Resultat:

```
irgendein Polygon
    5.0000000000000
   0.28553641680628   0.64885628735647
   0.86075146449730   0.99231909401715
   0.84941016510129    5.0041977781802D-02
   0.52570608118549   0.74855065811425
   0.99312098976225   0.41040589986369
    5.0000000000000
    1.0000000000000    2.0000000000000
    2.0000000000000    3.0000000000000
    3.0000000000000    4.0000000000000
    4.0000000000000    5.0000000000000
    5.0000000000000    1.0000000000000
```

was nicht sehr ansehnlich ist, weil man das Ausgabeformat nicht präzisiert hat. Der Nachteil besteht darin, dass ganze Zahlen von Scilab als Gleitkommazahlen behandelt werden[15], sie werden in einem Format geschrieben, das auf Gleitkommazahlen basiert. Außerdem wird einer Zeichenkette ein Leerzeichen vorangestellt (das in der Zeichenkette `text` nicht enthalten ist). Um ein besseres Ergebnis zu erhalten, muss man diese Fortran–Formate hinzufügen:

- für eine ganze Zahl verwendet man `Ix`, wobei `x` eine strikt positive ganze Zahl ist, die die Länge des Feldes angibt, in welches die Zahlen (rechtsbündig) als Zeichen geschrieben werden;
- für Gleitkommazahlen ist `Ex.y` das passende Format, wobei `x` die Gesamtlänge des Feldes und `y` die Länge der Mantisse bezeichnet; die Ausgabe nimmt dann folgende Form an: `[Vorzeichen]0.MantisseE[Vorzeichen]Exponent`. Für Gleitkommazahlen doppelter Genauigkeit liefert die Umrechnung ins Dezimalsystem ungefähr 16 signifikante Stellen, der Exponent liegt (ungefähr) zwischen -300 und +300, was eine Gesamtlänge von 24 Zeichen ergibt. Man kann also das Format `E24.16` benutzen (je nach der Größe einer Zahl und der gewünschten Darstellung können andere Formate besser geeignet sein);
- um das Leerzeichen vor der Zeichenkette zu vermeiden, kann man das Format `A` benutzen.

Wenden wir uns nun nocheinmal dem vorigen Beispiel zu; man erhält eine „ansehnlichere“ Ausgabe, wenn das Obengenannte berücksichtigt wird (unter der Annahme, dass es weniger als 999 Punkte und Kanten gibt):

```
write(unit,text,"(A)")               // Schreiben der Textzeile
write(unit,size(P,1),"(I3)")         // Schreiben der Punktanzahl
write(unit,P,"(2(X,E24.16))")        // Schreiben der Koordinaten der Punkte
write(unit,size(Endpunkte,1),"(I3)") // Schreiben der Kantenanzahl
write(unit,Endpunkte,"(2(X,I3))")    // Schreiben der Endpunkte
file("close",unit)                   // Schließen der Datei
```

(das Format `X` erzeugt ein Leerzeichen; außerdem wird ein Wiederholungfaktor verwendet: `2(X,E24.16)` bedeutet, dass man in die gleiche Zeile zwei Felder hintereinander schreiben will, die ein Leerzeichen gefolgt von einer aus 24 Zeichen bestehenden Gleitkommazahl enthalten), was Folgendes ergibt:

```
irgendein Polygon
  5
   0.2855364168062806E+00   0.6488562873564661E+00
   0.8607514644972980E+00   0.9923190940171480E+00
   0.8494101651012897E+00   0.5004197778180242E-01
   0.5257060811854899E+00   0.7485506581142545E+00
```

[15] Ab der Version 2.5 gibt es jedoch die `integer`–Typen int8, int16 und int32; siehe `Help`.

```
 0.9931209897622466E+00   0.4104058998636901E+00
5
 1   2
 2   3
 3   4
 4   5
 5   1
```

Um dieselbe Datei zu lesen, kann man die folgende Sequenz benutzen:

```
texte=read(unit,1,1,"(A)")      // Lesen der Textzeile
n = read(unit,1,1)              // Lesen der Punkteanzahl
P = read(unit,n,2)              // Lesen der Koordinaten der Punkte
m = read(unit,1,1)              // Lesen der Kantenanzahl
Endpunkte = read(unit,m,2)      // Lesen der Endpunkte
file("close",unit)              // Schließen der Datei
```

Wenn Sie diese paar Beispiele aufmerksam verfolgt haben, werden Sie festgestellt haben, dass das Schließen einer Datei mit Hilfe der Anweisung `file("close",unit)` erfolgt.

Zu guter Letzt können Sie im Scilabfenster lesen und schreiben, indem Sie jeweils `unit = %io(1)` und `unit = %io(2)` verwenden. Bezüglich des Schreibens kann man also eine sorgfältigere Darstellung als mit der Funktion `disp` erhalten (siehe ein Beispiel im Kapitel „Fallstricke“ Abschnitt „Scilab–Grundbefehle und –Funktionen“ (Aufrufskript der MonteCarlo–Funktion)).

3.6.2 C–ähnliche Ein– und Ausgabe

Zum Öffnen oder Schließen von Dateien benutzt man `mopen` und `mclose`, die grundlegenden Funktionen sind:

`mprintf, mscanf`	Schreiben / Lesen in das Scilab Fenster
`mfprintf, mfscanf`	Schreiben / Lesen in eine Datei
`msprintf, msscanf`	Schreiben / Lesen in eine Zeichenkettenvariable

Im Folgenden erkläre ich nur die Funktion `mprintf`. Das folgende Skript erläutert die wichtigsten Fälle:

```
n = 17;
m = -23;
a = 0.2;
b = 1.23e-02;
c = a + %i*b;
s = "Kuckuck";
mprintf("\n\r  n = %d, m = %d", n, m); // %d für ganze Zahlen
mprintf("\n\r  a = %g", a);           // %g für reelle Zahlen
mprintf("\n\r  a = %e", a);           // %e für reelle Zahlen (Exponentialnotation)
mprintf("\n\r  a = %24.16e", a);      // Angabe der Dezimalstellen
mprintf("\n\r  c = %g + i %g", real(c), imag(c)); // Ausgabe komplexer Zahl
mprintf("\n\r  s = %s", s);           // %s für eine Zeichenkette
```

Wenn Sie dieses Skript mit einem « ; » am Ende der Anweisung `exec` ausführen (also `exec("demo_mprintf.sce");`, wenn Ihr Skript so heißt), so erhalten Sie folgende Ausgabe:

```
n = 17, m = -23
a = 0.2
a = 2.000000e-01
a =   2.0000000000000001e-01
c = 0.2 + i 0.0123
s = Kuckuck
```

Ein paar Erklärungen:

- das `\n\r` erzeugt ein Zeilenende (unter Unix reicht `\n`) : benutzen Sie dies nicht, wenn Sie keinen Zeilenvorschub haben wollen!
- `%`x sind Formatierungsanweisungen, d.h. sie geben an, wie die zugehörige Variable (Ausdruck) ausgegeben wird:
  1. `%d` für ganze Zahlen
  2. `%e` für reelle Zahlen im Exponentialformat. Die Ausgabe hat die Form: $[-]c_0.c_1 \ldots c_6\mathrm{e}\pm d_1 d_2[d_3]$, d.h. mit einem Minuszeichen (falls der Wert negativ ist), einer Ziffer vor dem Dezimalpunkt, 6 Ziffern danach, der Buchstabe e, das Vorzeichen des Exponenten und schließlich 2 (notfalls 3) Stellen des Exponenten. Um mehr Stellen auszugeben, gibt man zwei durch einen Punkt getrennte Zahlen an; die erste gibt die Gesamtlänge der Ausgabe an, die zweite die der Anzahl der Ziffern nach dem Dezimalpunkt.
  3. `%g` Wählt eine Ausgabe mit oder ohne Exponentialdarstellung, je nachdem, was kürzer ist.
  4. `%5.3f` gibt den Wert in Festkommadarstellung mit 5 Stellen insgesamt und 3 Stellen nach dem Dezimalpunkt aus.
  5. `%s` benutzt man bei der Ausgabe von Zeichenketten.
- zu jeder Formatanweisungen sollte genau eine Ausgabewert (Variable, Ausdruck oder Konstante) folgen, z.B. wenn Sie 4 Werte ausgeben wollen, so sollte auch 4 Formate angegeben werden:

```
mprintf("   %d1 ..... %d4 ", expr1, expr2, expr3, expr4);
```

Frage: Die Ausgabe von 0.2 mit dem Format `%24.16e` erscheint merkwürdig. Antwort: das liegt darin, dass 0.2 nicht exakt in der Binärdarstellung des Rechners dargestellt werden kann; bei der Umwandlung in das Dezimalsystem (nur zur Ausgabe) wird dann u.U. der Rundungsfehler sichtbar.

3.7 Hinweise zur effizienten Programmierung in Scilab

Es folgen nun Beispiele mit einigen Tricks, die man kennen sollte. Man versucht eine Vandermonde–Matrix zu berechnen:

$$A = \begin{bmatrix} 1 & t_1 & t_1^2 & \ldots & t_1^n \\ 1 & t_2 & t_2^2 & \ldots & t_2^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 & \ldots & t_m^n \end{bmatrix}$$

Hier ein erster ziemlich einfacher Code:

```
function A=vandm1(t,n)
  // Berechnung der Vandermonde-Matrix A=[a(i,j)] 1<= i <= m
  //                                              1<= j <= n+1
  // wobei a(i,j) = ti^(j-1)
  // t muss ein Spaltenvektor mit n Komponenten sein
  m=size(t,'r')
  for i = 1:m
    for j = 1:n+1
      A(i,j) = t(i)^(j-1)
    end
  end
endfunction
```

Da man grundsätzlich die Größe von Matrizen und anderen Objekte in Scilab nicht deklariert, ist es nicht erforderlich, das Endformat unserer Matrix A: $(m, n+1)$ zu nennen. Da sich die Matrix nach und nach im Laufe der Rechnung vergrößert, muss Scilab mit diesem Problem umgehen können (für $i = 1$ ist A ein Zeilenvektor mit j Komponenten; für $i > 1$ ist A eine $i \times (n+1)$–Matrix; man hat also insgesamt $n + m - 1$ Änderungen der Dimensionen von A. Wenn man dagegen eine Pseudodeklaration der Matrix (durch die Funktion `zeros(m,n+1)`) verwendet:

```
function A=vandm2(t,n)
  // wie bei vandm1 jedoch mit einer Pseudodeklaration für A
    m=size(t,'r')
  A = zeros(m,n+1)     // Pseudodeklaration
  for i = 1:m
    for j = 1:n+1
      A(i,j) = t(i)^(j-1)
    end
  end
endfunction
```

gibt es dieses Problem nicht mehr, und man gewinnt deutlich:

```
-->t = linspace(0,1,1000)';
-->timer(); A = vandm1(t,200); timer()
 ans  =
    6.04

-->timer(); A = vandm2(t,200); timer()
 ans  =
    1.26
```

Man kann versuchen, diesen Code ein wenig zu optimieren, indem A mit `ones(m,n+1)` initialisiert wird (man vermeidet die Berechnung der ersten Spalte), indem man nur Multiplikationen gemäß $a_{ij} = a_{ij-1} \times t_i$ ausgeführt (was die Berechnung der Potenz vermeidet), oder sogar durch Vertauschung der beiden Schleifen, was aber nicht viel bringt. Eine gute Methode, A zu konstruieren, besteht darin, eine Vektoranweisung dafür zu verwenden:

```
function A=vandm3(t,n)
  // gute Methode: Verwende die Matrixschreibweise
  m=size(t,'r')
  A=ones(m,n+1)
  for i=1:n
     A(:,i+1)=t.^i
  end
endfunction

function A=vandm4(t,n)
  // wie bei vandm3, mit einer kleinen Optimierung
  m=size(t,'r')
  A=ones(m,n+1)
  for i=1:n
     A(:,i+1)=A(:,i).*t
  end
endfunction
```

und man macht die Funktion so signifikant schneller:

```
-->timer(); A = vandm3(t,200); timer()
 ans  =
    0.05

-->timer(); A = vandm4(t,200); timer()
 ans  =
    0.02
```

Nun ein zweites Beispiel: es handelt sich um die Auswertung einer Hut–Funktion (vgl. Abbildung

(3.2) in mehreren Punkten (diese Werte seien in einem (reellen) Vektor bzw. einer Matrix gespeichert):

$$\phi(t) = \begin{cases} 0 \text{ für } t \le a \\ \frac{t-a}{b-a} \text{ für } a \le t \le b \\ \frac{c-t}{c-b} \text{ für } b \le t \le c \\ 0 \text{ für } t \ge c \end{cases}$$

Da diese Funktion stückweise definiert ist, benötigt ihre Auswertung in einem Punkt in der Regel mehrere

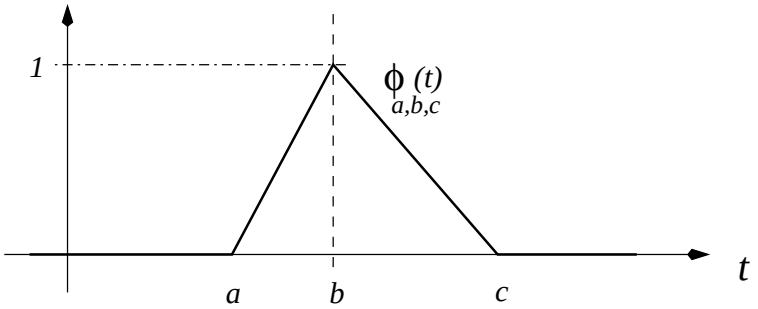

Abbildung 3.2: Die Hut–Funktion

Tests. Soll diese Arbeit in vielen Punkten vorgenommen werden, müssen diese Tests vektorisiert werden, um zu vermeiden, dass der Interpreter diese Aufgabe übernimmt. Hier ein erster Versuch:

```
function y=phi1(t,a,b,c)
  // wertet die Hut--Funktion (mit Parametern a, b und c) auf dem Vektor
  // (sogar der Matrix) t elementweise aus.
  // a,b und c müssen a < b < c erfüllen
  [n,m] = size(t)
  y = zeros(t)
  for j=1:m, for i=1:n
     if t(i,j) > a then
        if t(i,j) < b then
           y(i,j) = (t(i,j) - a)/(b - a)
        elseif t(i,j) < c then
           y(i,j) = (c - t(i,j))/(c - b)
        end
     end
  end, end
endfunction
```

liefert das Ergebnis:

```
-->a = -0.2 ; b=0 ; c=0.15;
 -->t = rand(200000,1)-0.5;

-->timer(); y1 = phi1(t,a,b,c); timer()
 ans  =
    2.46
```

während die folgenden Codes[16]:

```
function y=phi2(t,a,b,c)
  // wertet die Hut--Funktion (mit Parametern a, b und c) auf dem Vektor
  // (sogar der Matrix) t elementweise aus.
  // a,b und c müssen a < b < c erfüllen
  Ist_in_a_b = bool2s( (a < t) & (t <= b) )
```

[16] in denen es die Funktion `bool2s` ermöglicht, eine boolsche in eine reelle Matrix (true ergibt 1 und false 0) umzuwandeln

```
  Ist_in_b_c = bool2s( (b < t) & (t < c ) )
  y = Ist_in_a_b .* (t - a)/(b - a) + Ist_in_b_c .* (c - t)/(c - b)
endfunction

function y=phi3(t,a,b,c)
  // wie bei phi2 mit einer kleinen Optimierung
  t_le_b = ( t <= b )
  Ist_in_a_b = bool2s( (a < t) & t_le_b )
  Ist_in_b_c = bool2s( ~t_le_b & (t < c) )
  y = Ist_in_a_b .* (t - a)/(b - a) + Ist_in_b_c .* (c - t)/(c - b)
endfunction
```

schneller sind:

```
-->timer(); y2 = phi2(t,a,b,c); timer()
 ans  =
    0.12

-->timer(); y3 = phi3(t,a,b,c); timer()
 ans  =
    0.12

-->timer(); y4 = phi4(t,a,b,c); timer()  // Definition weiter unten
 ans  =
    0.1
```

Bemerkungen:

- die kleine Optimierung für `phi2` ergibt keine Verbesserung (im Gegensatz zu einer früheren Version von Scilab auf einer anderen Maschine)
- Die Funktion `phi4` benutzt die Funktion `find`, was viel natürlicher und einfacher ist: auf einen boolschen Vektor `b` angewandt, gibt sie einen Vektor zurück, der Indizes `i` enthält, für die gilt: `b(i)=%t` (bzw. eine leere Matrix, wenn alle Komponenten den Wert %f haben). Beispiel:

  ```
  -->x = rand(1,6)
   x  =
  !   0.8497452    0.6857310    0.8782165    0.0683740    0.5608486    0.6623569 !

  -->ind = find( 0.3<x & x<0.7 )
   ind  =
  !   2.    5.    6. !
  ```

  Auf eine boolsche Matrix `A` angewandt, erhalten Sie dieselbe Liste, wobei zu berücksichtigen ist, dass die Matrix ein „langer“ Vektor ist, in dem die Elemente von `A` spaltenweise angeordnet sind. Es ist jedoch möglich, mittels eines zweiten Ausgabeargumentes Vektoren von Zeilen– und Spaltenindizes zu erhalten:

  ```
  -->A = rand(2,2)
   A  =
  !   0.7263507    0.5442573 !
  !   0.1985144    0.2320748 !

  -->[ir,ic]=find(A<0.5)
   ic  = ! 1.    2. !
   ir  = ! 2.    2. !
  ```

Nun die Funktion `phi4`, die `find` verwendet:

```
function y=phi4(t,a,b,c)
  // hier ist die Funktion find sehr hilfreich
  t_le_b = ( t <= b )
  ist_in_a_b = find( a<t & t_le_b )
  ist_in_b_c = find( ~t_le_b & t<c )
  y = zeros(t)
  y(ist_in_a_b) = (t(ist_in_a_b) - a)/(b - a)
  y(ist_in_b_c) = (c - t(ist_in_b_c))/(c - b)
endfunction
```

Fazit: Falls Ihre Berechnungen zu langsam werden, versuchen Sie sie zu vektorisieren. Ist dieses Vektorisieren nicht möglich oder unzureichend, bleibt nichts anderes übrig, als die entscheidenden Abschnitte in C oder in Fortran 77 zu schreiben.

3.8 Übungen

1. Schreiben Sie eine Funktion, um ein lineares Gleichungssystem zu lösen, wobei die Matrix eine obere Dreiecksmatrix ist. Man benutze die Anweisung `size`, die die beiden Dimensionen einer Matrix zurückgibt:

   ```
   -->[n,m]=size(A)
   ```

   In einem ersten Schritt programmiere man den klassischen Algorithmus unter Benutzung von zwei Schleifen, dann versuche man, die innere Schleife durch eine Matrixanweisung zu ersetzen. Um Ihre Funktion zu testen, können Sie eine Matrix aus Zufallszahlen erzeugen und davon nur den oberen Dreiecksteil mit der Anweisung `triu` verwenden:

   ```
   -->A=triu(rand(4,4))
   ```

2. Die Lösung des Systems gewöhnlicher Diferrentialgleichungen erster Ordnung

   $$\frac{dx}{dt}(t) = Ax(t), \quad x(0) = x_0 \in \mathbb{R}^n, \quad x(t) \in \mathbb{R}^n, \quad A \in \mathcal{M}_{nn}(\mathbb{R})$$

   kann man mithilfe der Matrix–Exponentialfunktion erhalten:

   $$x(t) = e^{At}x_0$$

   Obwohl Scilab über eine Funktion verfügt, die die Matrix–Exponentialfunktion berechnet (`expm`), bleibt ohne Zweifel noch etwas zu tun. Man möchte die Lösung für $t \in [0, T]$ wissen. Dafür kann man diese zu einer ausreichend großen Zahl von im Intervall $[0, T]$ gleichverteilten Zeitpunkten, $t_k = k\delta t, \quad \delta t = T/n$, berechnen und die Eigenschaften der Exponentialfunktion ausnutzen, um die Rechnung zu erleichtern:

   $$x(t_k) = e^{Ak\delta t}x_0 = e^{k(A\delta t)}x_0 = (e^{A\delta t})^k x_0 = e^{A\delta t}x(t_{k-1})$$

   Es reicht also einzig und allein, die Exponentialfunktion der Matrix $A\delta t$ zu berechnen und dann n Matrix–Vektor–Multiplikationen auszuführen, um $x(t_1), x(t_2), \ldots, x(t_n)$ zu erhalten. Schreiben Sie ein Skript, um die Differentialgleichung (eine gedämpfte Schwingung)

   $$x'' + \alpha x' + kx = 0, \text{ z.B. mit } \alpha = 0.1, k = 1, x(0) = x'(0) = 1$$

   zu lösen, die offenbar in Form eines Systems von zwei Gleichungen erster Ordnung geschrieben werden kann. Zum Schluss visualisiere man x als Funktion der Zeit, anschließend die Trajektorie im Phasenraum. Man kann mit dem Befehl `xset("window",window-number)` von einem Graphikfenster zum nächsten wechseln. Zum Beispiel:

   ```
   --> // Ende der Berechnungen
   --> xset('window',0) // wählt das Graphikfenster 0
   --> Anweisungen für den ersten Graphen (der in Fenster 0 angezeigt wird)
   --> xset('window',1) // wählt das Graphikfenster 1
   --> Anweisungen für den zweiten Graphen (der in Fenster 1 angezeigt wird)
   ```

3. Schreiben Sie eine Funktion `[i,info]=intervall_von(t,x)`, welche das Intervall i mit $x_i \leq t \leq x_{i+1}$ mit Hilfe der Binärsuche bestimmt (die Einträge des Vektors x seien so, dass $x_i < x_{i+1}$ gilt). Falls $t \notin [x_1, x_n]$, so soll die boolesche Variable `info` gleich `%f` (und `%t` im umgekehrten Fall) sein.

4. Schreiben Sie die Funktion `myhorner` um für den Fall, dass das Argument `t` eine Matrix ist (die Funktion soll eine Matrix `p` derselben Größe wie `t` zurückliefern, wobei jeder Koeffizient (i, j) der Auswertung des Polynoms in `t(i,j)` entspricht).

5. Schreiben Sie eine Funktion `y = signal_fourier(t,T,cs)`, die den Anfang einer Fourierreihe zurückgibt unter Benutzung der Funktionen

$$f_1(t,T) = 1,\ f_2(t,T) = \sin(\frac{2\pi t}{T}t),\ f_3(t,T) = \cos(\frac{2\pi t}{T}), f_4(t,T) = \sin(\frac{4\pi t}{T}),\ f_5(t,T) = \cos(\frac{4\pi t}{T}), \cdots$$

anstelle der Exponentialfunktion. T ist ein Parameter (die Periode), und das Signal wird (außer durch seine Periode) durch den Vektor `cs` charakterisiert; seine Komponenten sind in der Basis $f_1, f_2, f_3, \cdots$ zu verstehen. Man beschaffe sich die Anzahl der zu verwendenden Funktionen mithilfe der Funktion `length`, die man auf `cs` anwendet. Es ist ratsam eine Hilfsfunktion `y=f(t,T,k)` zu verwenden, um $f_k(t,T)$ zu berechnen. Abschließend soll das alles auf einen Vektor (oder eine Matrix) von Zeitpunkten `t` so angewendet werden können, dass man ein solches Signal einfach visualisieren kann:

```
--> T = 1 // eine Periode...
--> t = linspace(0,T,101) // die Zeitpunkte ...
--> cs =  [0.1 1 0.2 0 0  0.1] // ein Signal mit einem konstanten,
--> // einem T-periodischen, keinem 2T-periodischen aber
--> // einem 4T-periodischen Anteil
--> y = signal_fourier(t,T,cs); // Berechnung des Signals
--> plot(t,y) // und eine Zeichnung...
```

6. Hier eine Funktion, um das Vektorprodukt zweier Vektoren zu berechnen:

```
function v=prod_vect(v1,v2)
  // Vektorprodukt v = v1 /\ v2
  v(1) = v1(2)*v2(3) - v1(3)*v2(2)
  v(2) = v1(3)*v2(1) - v1(1)*v2(3)
  v(3) = v1(1)*v2(2) - v1(2)*v2(1)
endfunction
```

Vektorisieren Sie diesen Code derart, dass in einer Funktion `function v=prod_vect_v(v1,v2)` die Vektorprodukte $v^i = v_1^i \wedge v_2^i$ berechnet werden, wobei v^i, v_1^i und v_2^i die i–te Spalte von $3\times n$–Matrizen bezeichnet, welche diese Vektoren enthält.

7. *Treffen sie sich?* : Herr A und Fräulein B haben ausgemacht, sich zwischen 17 und 18 Uhr zu treffen. Beide treffen rein zufällig zwischen 17 und 18 Uhr ein — mit einer gleichverteilten Wahrscheinlichkeit und stochastisch unabhängig. Fräulein B wartet 5 Minuten, bevor sie wieder geht, Herr A wartet 10 Minuten.

   (a) Wie groß ist die Wahrscheinlichkeit, dass sie sich treffen? (Antw. 67/288)

   (b) Bestimmen Sie diese Wahrscheinlichkeit approximativ mit Hilfe einer Simulation: schreiben Sie eine Funktionen `p = rvd(m)`, die die empirische Wahrscheinlichkeit bei m Realisationen berechnet.

   (c) Experimentieren Sie mit Ihrer Funktion mit immer größeren Werten von m.

Kapitel 4

Graphik

Auf diesem Gebiet besitzt Scilab zahlreiche Möglichkeiten, die von den einfachen Grundbefehlen[1] bis hin zu komplexen Funktionen reichen, welche mit einer einzigen Anweisung alle Arten von Graphiken zu zeichnen vermögen. Im Folgenden wird nur ein kleiner Teil dieser Möglichkeiten erläutert. *Bemerkung:* Für diejenigen, die die MATLAB–Graphikfunktionen[2] kennen, hat Stéphane Mottelet eine Bibliothek von Scilabfunktionen geschrieben, um Plots wie in MATLAB zu erstellen; diese ist unter folgender Adresse zu finden:

`http://www.dma.utc.fr/~mottelet/scilab/`

4.1 Graphikfenster

Wenn man eine Anweisung wie `plot, plot2d, plot3d ...` startet, wählt Scilab das Fenster Nr. 0 für die Zeichnung, falls kein anderes Fenster aktiviert ist. Wird ein weiterer Graph gezeichnet, so überlagert er i.A. den ersten[3], und man muss vorher das Graphikfenster löschen, was sich entweder mit Hilfe des Menus dieses Fensters machen lässt (Eintrag `clear` im Menu `File`), oder im Scilabfenster mit Hilfe des Befehls `xbasc()`. Im Grunde kann man mit Hilfe folgender Anweisungen sehr leicht mit mehreren Graphikfenstern umgehen:

`xset("window",num)`	das aktive Fenster wird das Fenster Nummer `num`; existiert dieses Fenster nicht, wird es von Scilab erzeugt.
`xselect()`	bringt das aktive Fenster in den Vordergrund; existiert kein Graphikfenster, erzeugt Scilab eines.
`xbasc([num])`	löscht das Graphikfenster Nummer `num`; wird `num` weggelassen, löscht Scilab das aktive Fenster.
`xdel([num])`	zerstört das Graphikfenster Nummer `num`; wird `num` weggelassen, zerstört Scilab das aktive Fenster.

Ganz allgemein, wenn man ein aktives Fenster (mit `xset("window",num)`) ausgewählt hat, erlaubt ein ganzer Satz von `xset("name",a1,a2,...)` Anweisungen alle Parameter dieses Fensters zu setzen: `"name"` bezeichnet i.A. den Parametertyp, der eingestellt werden soll, wie beispielsweise `font` für den verwendeten Zeichensatz (für diverse Titel und Beschriftungen), `"thickness"` für die Strichstärke, `"colormap"` für die Farbpalette usw., gefolgt von einem oder mehreren Argumenten für die eigentliche Einstellung. Die Gesamtheit dieser Parameter bildet das, was man einen Graphikkontext nennt (jedes Fenster kann also seinen eigenen Graphikkontext haben). Details über diese (ziemlich zahlreichen) Parameter erfahren Sie mittels `Help` in der Rubrik `Graphic Library`, aber die Mehrzahl von ihnen kann man auch interaktiv über ein Graphikmenu einstellen, welches nach dem Kommando `xset()` erscheint (Bemerkung: dieses Menu zeigt auch die Farbpalette (aber man kann sie hier nicht ändern)). Schließlich erlaubt der Satz von `[a1,a2,...]=xget('name')`–Anweisungen verschiedene Parameter des Graphikkontextes zu bestimmen.

[1] Beispiele: Zeichnen von Rechtecken, Polygonen (gefüllt oder ungefüllt), Herausfinden von Koordinaten des Maus–Zeigers
[2] im Allgemeinen einfacher als die von Scilab!
[3] außer bei `plot`, das automatisch den Inhalt eines aktiven Fensters vorher löscht

4.2 Einführung in plot2d

Wir haben bereits die einfache Anweisung `plot` betrachtet. Will man jedoch mehrere Kurven zeichnen, ist es besser, sich auf `plot2d` zu konzentrieren. Eine einfache Anwendung sieht folgendermaßen aus:

```
x=linspace(-1,1,61)'; // die Abszissen (als Spaltenvektor)
y = x.^2;   // die Ordinaten (ebenfalls als Spaltenvektor)
plot2d(x,y) // --> braucht Spaltenvektoren!
```

Fügen wir nun eine andere Kurve hinzu:

```
ybis = 1 - x.^2;
plot2d(x,ybis)
xtitle("Kurven")  // erzeugt einen Titel
```

Die nächste Kurve passt nicht mehr in diesen Maßstab[4]:

```
yter = 2*y;
plot2d(x,yter)
```

Man stellt fest, dass Scilab die Skalierung an die dritte Kurve anpasst[5] und die ersten beiden jetzt in diesem neuen Maßstab zeichnet. Dies scheint völlig normal zu sein, aber diese Eigenschaft gibt es erst seit Version 2.6 von Scilab (davor erschien die 3. Kurve deckungsgleich zur zweiten).

Es ist jedoch möglich, dass alle drei Kurven gleichzeitig gezeichnet werden:

```
xbasc() // für's Löschen
plot2d([x x x],[y ybis yter]) // Zusammenfügen von Matrizen ...
xtitle("Kurven","x","y")      // Titel und die Beschriftung der beiden Achsen
```

und Sie werden etwas erhalten, was der Abbildung 4.1 ähnelt.

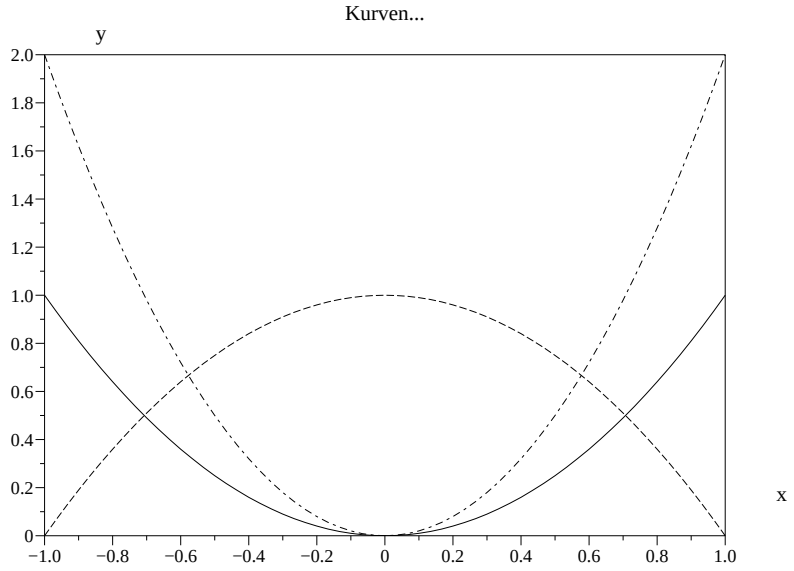

Abbildung 4.1: Die Funktionen x^2, $1 - x^2$ und $2x^2$

Um also gleichzeitig mehrere Kurven zu zeichnen, benutzt man `plot2d(Mx,My)`, wobei `Mx` und `My` zwei Matrizen derselben Größe sind; die Anzahl der Kurven ist gleich der Spaltenanzahl `nc`, und die i–te Kurve ergibt sich aus den Vektoren `Mx(:,i)` (ihre Abszissen) und `My(:,i)` (ihre Ordinaten). In dem Falle, dass die Abszissen übereinstimmen (wie in meinem Beispiel), gibt man diese einfach nur einmal an, anstatt sie `nc` mal zu duplizieren (`plot2d(x,My)` statt `plot2d([x x .. x],My)`).

[4]Der Maßstab beinhaltet die Längen des angezeigten Rechteckes und ggf. weitere Eigenschaften.

[5]Der Maßstab passt sich immer automatisch so an, dass alle Kurven vollständig sichtbar sind.

4.3 plot2d mit optionalen Parametern

Die allgemeine Form lautet:

`plot2d(Mx,My <,opt_arg>*)`

wobei `<,opt_arg>*` die (eventuell vorhandene) Liste von optionalen Parametern bezeichnet. `opt_arg` hat dabei die Form[6]:

Schlüsselwort=Wert

Die Reihenfolge der optionalen Parameter spielt keine Rolle. In den folgenden Beispielen probieren wir die wichtigsten Optionen aus:

1. **Auswahl der Farbe und einer Legende**: im letzten Beispiel hat Scilab die 3 verschiedenen Kurven in 3 verschiedenen Farben dargestellt, es hat die (Standard–)Farben 1, 2 und 3 benutzt. Eine (unvollständige) Liste ist

1	schwarz	5	rot	23	violett
2	blau	6	lila	26	(kastanien)braun
3	hellgrün	13	dunkelgrün	29	rosa
4	hellblau	16	türkis	32	gelborange

   aber die Anweisung `xset()` zeigt Ihnen alle Farben[7]. Man wählt die Farbe mit der Option *style=vect* aus, wobei *vect* ein Zeilenvektor ist, dessen i–te Komponente die (Nummer der) Farbe der i–ten Kurve bestimmt. Die Legende wählt man mittels *leg=str* aus, wobei *str* eine Zeichenkette der Form `"leg1@leg2@..."` und leg_i die Legende der i–ten Kurve ist; siehe das folgende Beispiel (und Abb. (4.2)):

```
x = linspace(0,15,200)';
y = besselj(1:5,x);
xbasc()
plot2d(x, y, style=[2 3 4 5 6], leg="J1@J2@J3@J4@J5")
xtitle("Die Besselfunktionen J1, J2,...","x","y")
```

   und da die Reihenfolge der optionalen Parameter keine Rolle spielt, hätte man genauso auch

```
plot2d(x, y, leg="J1@J2@J3@J4@J5", style=[2 3 4 5 6])
```

   verwenden können.

2. **Kurven mit Markierungssymbolen**: Manchmal möchte man einzelne Punkte einer Kurve mit einem Markierungssymbol versehen; dazu benutzt man einen `style` mit einem Wert zwischen 0 und −9, siehe die folgende Tabelle:

style	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
symbole	·	+	×	⊕	♦	♢	△	▽	♣	◯

   Versuchen Sie z.B. (siehe Abb. (4.3)):

```
x = linspace(0,2*%pi,40)';
y = sin(x);
yp = sin(x) + 0.1*rand(x,"normal"); // Gauss--verteilte Störung
xbasc()
plot2d(x, [yp y], style=[-2 2], leg="y=sin(x)+Störung@y=sin(x)")
```

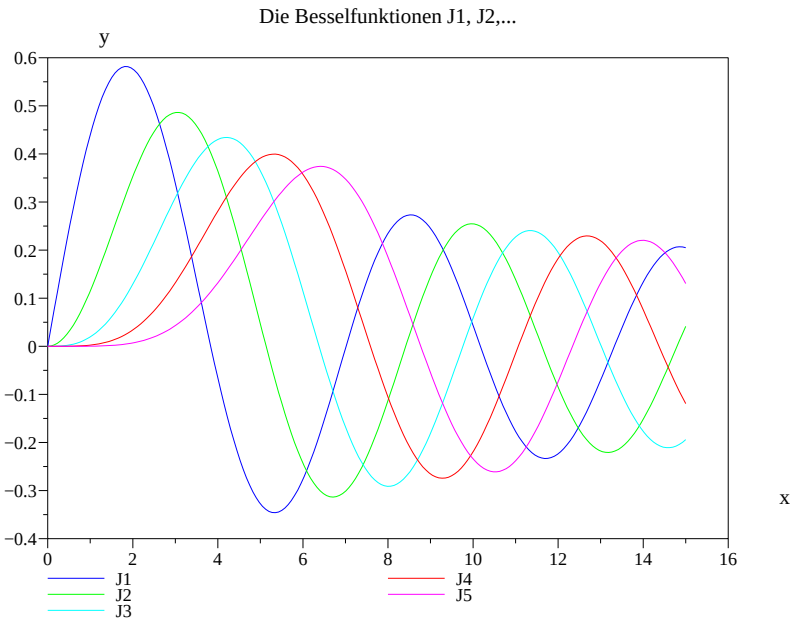

Abbildung 4.2: Wahl der Farbe und Legende

3. **Festlegung des Abbildungsmaßstabes**: Im folgenden Beispiel, das u.a. einen Kreis (siehe Abb. (4.4)) enthält, ist es nötig, eine isometrische Skalierung zu wählen, d.h. man erzwingt, dass beide Achsen mit demselben Maßstab skaliert werden. Die Option *frameflag=Wert*, hier mit dem Wert 4, erzwingt eine isometrische Skalierung, die aus den Minima und Maxima der Daten berechnet wird:

```
t = linspace(0,2*%pi,60)';
x1 = 2*cos(t); y1 = sin(t);                 // eine Ellipse
x2 = cos(t);   y2 = y1;                     // ein Kreis
x3 = linspace(-2,2,60)'; y3 = erf(x3);      // die Fehlerfunktion
Legende="Ellipse@Kreis@Fehlerfunktion";  // die Legenden
plot2d([x1 x2 x3],[y1 y2 y3],style=[1 2 3], frameflag=4, leg=Legende)
xtitle("noch mehr Kurven ...","x","y")
```

In einigen Fällen muss man *frameflag* noch durch die Option *rect* ergänzen, z.B., wenn man selbst den Maßstab/Ausschnitt wählen will, anstatt es der Automatik von `plot2d` zu überlassen. Dazu benutzt man $rect = [x_{min}, y_{min}, x_{max}, y_{max}]$ in Zusammenhang mit der Option $frameflag = 1$ wie in dem folgenden Beispiel:

```
x = linspace(-5,5,200)';
y = 1 ./(1 + x.^2);
xbasc()
plot2d(x, y, frameflag=1, rect=[-6,0,6,1.1])
xtitle("Die Funktion von Runge")
```

und schließlich alle möglichen Werte von *frameflag*:

[6] *formales_Argument=effektives_Argument*

[7] die Standardpalette ist nicht gerade optimal, da einige Farbe sich sehr ähneln, aber wir werden weiter unten sehen, wie man sie ändern kann.

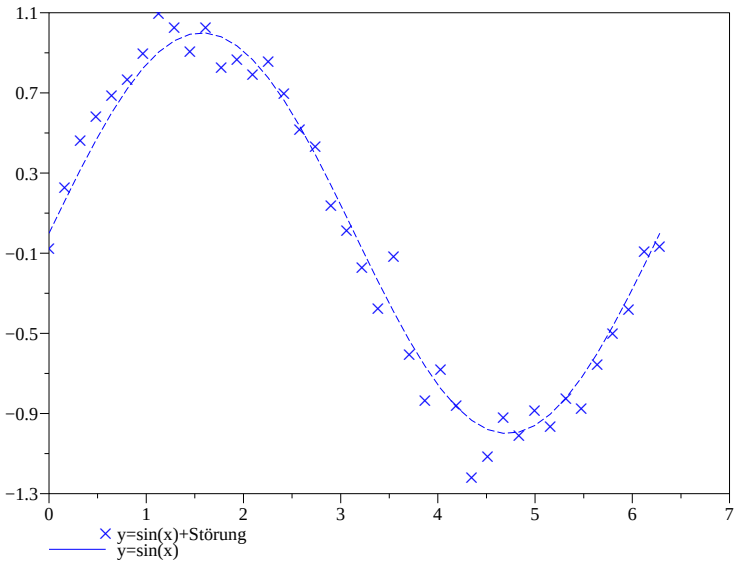

Abbildung 4.3: Zeichnung mit einfachem Strich und nicht verbundenen Markierungssymbolen

`frameflag=0`	Benutzung des vorherigen Maßstabes oder der Voreinstellung
`frameflag=1`	der Maßstab wird mittels *rect* angegeben
`frameflag=2`	der Maßstab errechnet sich aus dem Max und Min von `Mx` und `My`
`frameflag=3`	isometrischer Maßstab abgeleitet von *rect*
`frameflag=4`	isometrischer Maßstab abgeleitet aus dem Max und Min von `Mx` und `My`
`frameflag=5`	wie 1, aber ggf. mit einer Anpassung für eine schöne Unterteilung
`frameflag=6`	wie 2, aber ggf. mit einer Anpassung für eine schöne Unterteilung
`frameflag=7`	wie 1, aber die bisherigen Kurven werden neu gezeichnet
`frameflag=8`	wie 2, aber die bisherigen Kurven werden neu gezeichnet

Bem.: In den Fällen 4 und 5 wird der Abbildungsmaßstab eventuell so modifiziert, dass die Graduierung (, die immer den Anfang und das Ende einer Achse enthält) "schön" ist (, d.h. die Werte der Unterteilungen lassen sich mit möglichst wenigen Dezimalen darstellen).

4. **Festlegung der Lage der Achsen**: dies erreicht man mit der Option *axesflag=Wert* In dem folgenden Beispiel (siehe Abb. (4.5)) habe ich den Wert 5 gewählt, damit sich die Achsen im Punkte $(0,0)$ [8] schneiden; weiterhin wird kein umschließender Rahmen gezeichnet:

```
x = linspace(-14,14,300)';
y = sinc(x);
xbasc()
plot2d(x, y, style=2, axesflag=5)
xtitle("Die sinc Funktion")
```

Und hier alle Werte von *axesflag* im Überblick:

`axesflag=0`	ohne Rahmen, Achsen und Unterteilungen
`axesflag=1`	mit Rahmen, Achsen und Unterteilungen (x–Achse unten, y–Achse links)
`axesflag=2`	mit Rahmen, aber ohne Achsen und Unterteilungen
`axesflag=3`	mit Rahmen, Achsen und Unterteilungen (x–Achse unten, y–Achse rechts)
`axesflag=4`	ohne Rahmen, aber mit Unterteilungen und Achsen in der Mitte
`axesflag=5`	ohne Rahmen, aber mit Unterteilungen und Achsen durch $(0,0)$

[8] falls der Ursprung sichtbar ist.

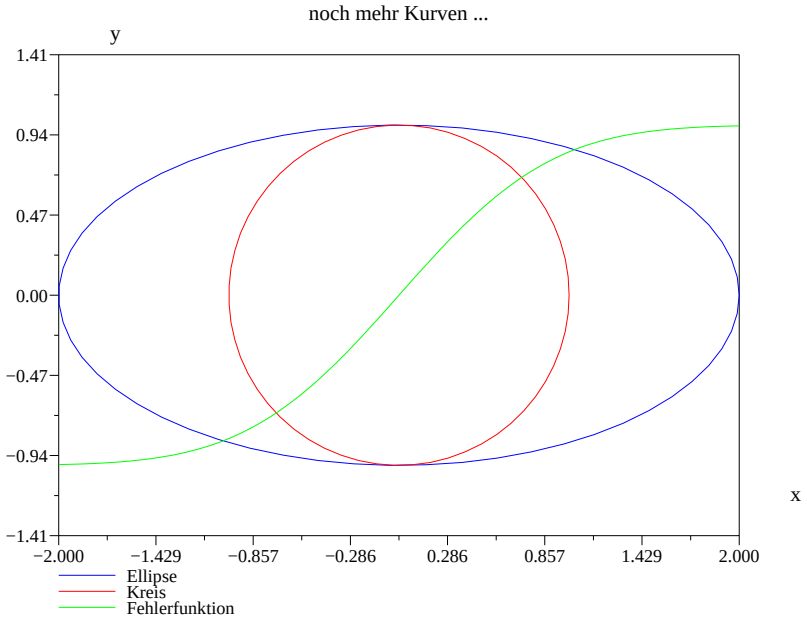

Abbildung 4.4: Ellipse, Kreis und Fehlerfunktion

5. **Benutzung eines logarithmischen Maßstabes**: man benutzt die Option *logflag=str*, wobei *str* eine Zeichenketten aus genau 2 Zeichen ist, die (nur) die Werte “n” (nicht logarithmisch) und “l” (logarithmisch) haben können; dabei bezieht sich das erste Zeichen auf die x–Achse, das zweite auf die y–Achse, z.B.:

```
x = logspace(0,4,200)';
y = 1 ./x;
xbasc()
subplot(1,2,1)
   plot2d(x, y, style=2, logflag= "ln")
   xtitle("logflag=""ln""")
subplot(1,2,2)
   plot2d(x, y, style=2, logflag= "ll")
   xtitle("logflag=""ll""")
```

Dieses Bespiel zeigt uns außerdem, wie man mehrere Zeichnungen in gleichen Graphikfenster ausgeben kann, indem man die Anweisung *subplot(m, n, num)* benutzt. Dabei gibt der Parameter m die Anzahl der vertikalen Unterteilungen (m gleiche Teile) und n die Anzahl der horizontalen Unterteilungen an. Weiter ist *num* die Nummer des Teilfensters der $m \times n$ kleinen Fenster; die Nummerierung verläuft von links nach rechts und von oben nach unten — also hat das Teilfenster in der Position (i, j) die Nummer[9] $n \times (i-1) + j)$ — am Besten man probiert es aus, z.B. mit:

```
xbasc()
subplot(1,2,1)
   titlepage("links")
subplot(3,2,2)
   titlepage(["rechts";"oben"])
subplot(3,2,4)
   titlepage(["rechts";"in der Mitte"])
subplot(3,2,6)
```

[9] und nicht wie u.U. noch in der Hilfeseite angegeben $m \times (i-1) + j$

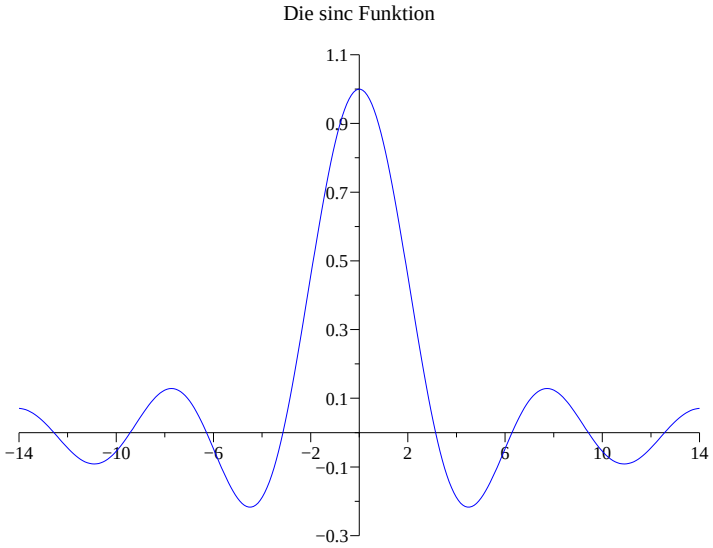

Abbildung 4.5: Platzierung der Achsen mittels $axesflag = 5$

```
   titlepage(["rechts";"unten"])
xselect()
```

Das Graphikfenster wird durch einen vertikalen Schnitt (links/rechts) in zwei Teile zerlegt, dessen rechter Teil wiederum durch horizontale Schnitte in drei Teile zerlegt wird. Man muss sich nur klarmachen, dass `subplot` eine Anweisung ist, (nur) einen Teil des Graphikfenster für die folgenden Plot–Anweisungen auszuwählen.

6. **das Schlüsselwort strf** existiert aus Kompatibilität mit älteren Versionen von Scilab. Es ersetzt `frameflag` und `axesflag` und enthält einen Schalter (Flag), der anzeigt, ob eine Legende benutzt wird oder nicht. Der Wert von *strf* ist eine Zeichenketten `"xyz"` von genau 3 Zeichen mit:

   x $= 0$ (keine Legende) oder 1 (mit Legende in Kombination mit *leg=Wert*)

   y ein Wert zwischen 0 und 9, welcher dem Wert von *frameflag* entspricht

   z ein Wert zwischen 0 und 5, welcher dem Wert von *axesflag* entspricht

   Man muss dieses Schlüsselwort kennen, da viele Graphikfuntionen noch nicht die Schlüsselworte *frameflag* und *axesflag* zulassen. Es ist außerdem ganz praktisch, wenn man zu einer schon aufgebauten Graphik noch etwas hinzufügen will, ohne den Maßstab zu ändern, oder einen (neuen) Rahmen zu zeichnen; man schreibt kurz `strf="000"` anstelle von `frameflag=0, axesflag=0` .

4.4 Varianten von plot2d: plot2d2, plot2d3

Man benutzt sie genauso wie `plot2d`, mit der gleichen Syntax und denselben Optionen.

1. **plot2d2** zeichnet eine Treppenfunktion. Anstatt die Punkte (x_i, y_i) und (x_{i+1}, y_{i+1}) durch einen Strecke zu verbinden, zeichnet `plot2d2` eine horizontale Strecke zwischen (x_i, y_i) und (x_{i+1}, y_i) und dann eine vertikale Strecke zwischen (x_{i+1}, y_i) und (x_{i+1}, y_{i+1}), z.B. (siehe Abb. (4.7)) :

```
n = 10;
x = (0:n)';
y = x;
```

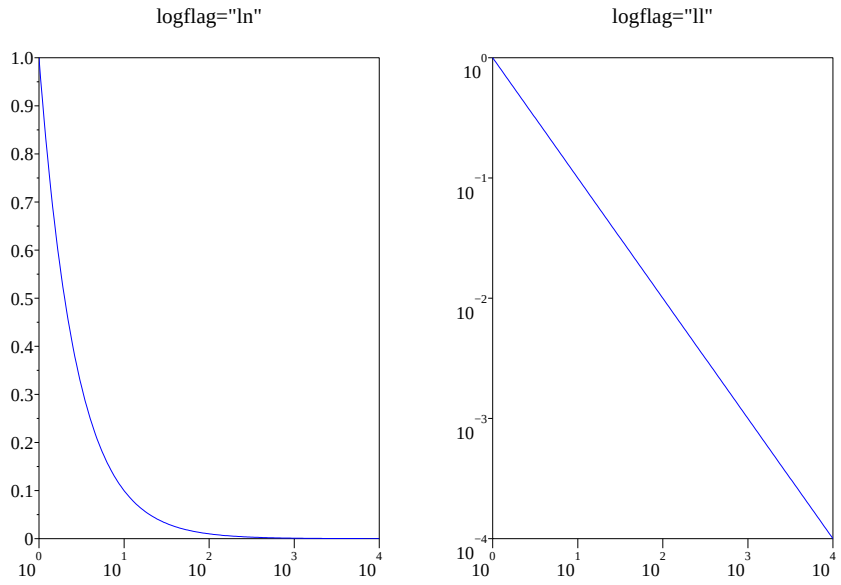

Abbildung 4.6: Darstellungen mit dem Parameter *logflag*

```
xbasc()
plot2d2(x,y, style=2, frameflag=5, rect=[0,-1,n+1,n+1])
xtitle("plot2d2")
```

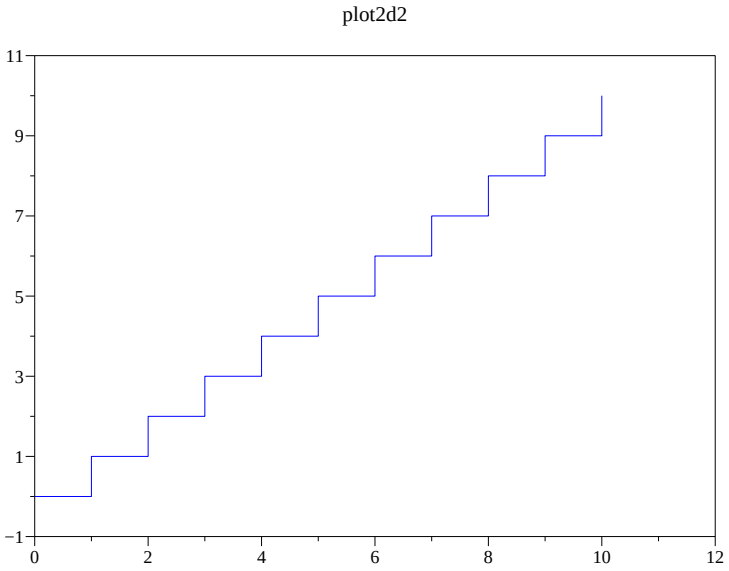

Abbildung 4.7: Illustration von plot2d2

2. **plot2d3** erstellt ein Balkendiagramm: für jeden Punkt (x_i, y_i) zeichnet `plot2d3` eine vertikale Strecke zwischen $(x_i, 0)$ und (x_i, y_i); ein Beispiel sei (siehe Abb. (4.8)):

```
n = 6;
x = (0:n)';
y = binomial(0.5,n)';
```

```
xbasc()
plot2d3(x,y, style=2, frameflag=5, rect=[-1,0,n+1,0.32])
xtitle("Wahrscheinlichkeiten der Binomialverteilung B(6,1/2)")
```

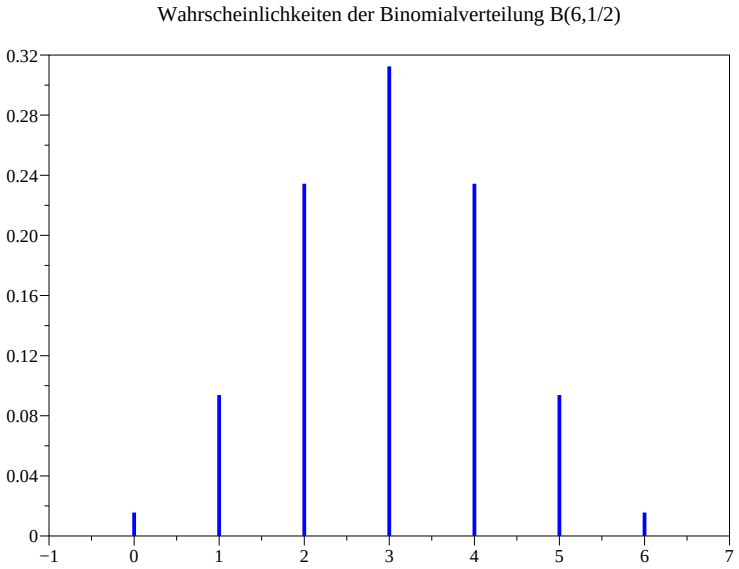

Abbildung 4.8: Illustration von plot2d3

4.5 Zeichnen von mehreren Kurven, die unterschiedlich viele Punkte haben

Mit `plot2d` und seinen Varianten kann man nicht mehrere Kurven zeichnen, die nicht mit der gleichen Anzahl von Intervallen diskretisiert worden sind. Man ist also gezwungen, mehrere Funktionsaufrufe zu verwenden. Seit Scilab Version 2.6 gibt es dabei keine bösen Überraschungen, da standardmäßig (`frameflag=8`) die vorher gezeichneten Kurven bei jedem weiteren Funktionsaufruf neu skaliert und gezeichnet werden. Wenn man jedoch die vollständige Kontrolle über die verwendete Skalierung haben will, muss man dies beim Aufruf der ersten Plot–Funktion festlegen und dann bei allen weiteren Aufrufen `frameflag=0` benutzen[10]; hier ein Beispiel (siehe Abb. (4.9)):

```
x1 = linspace(0,1,61)';
x2 = linspace(0,1,31)';
x3 = linspace(0.1,0.9,12)';
y1 = x1.*(1-x1).*cos(2*%pi*x1);
y2 = x2.*(1-x2);
y3 = x3.*(1-x3) + 0.1*(rand(x3)-0.5);  // wie y2 mit einer Störung
ymin = min([y1 ; y2 ; y3]); ymax = max([y1 ; y2 ; y3]);
dy = (ymax - ymin)*0.05;
rect = [0,ymin - dy,1,ymax+dy];   // Zeichenbereich
xbasc()                           // Löschen vorheriger Graphiken...
plot2d(x1,y1,1,"011"," ",rect)    // erster Aufruf, der die Skalierung bestimmt
plot2d(x2,y2,2,"000")             // zweiter Aufruf und
plot2d(x3,y3,-1,"000")            // dritter Aufruf: mit vorheriger Skalierung
xtitle("Kurven...","x","y")
```

Bem.: Probieren Sie dieses Beispiel auch mit *frameflag=1* anstelle von *frameflag=5* aus!
Man kann hier also nicht jede Kurve mit einer Legende versehen, aber man kann dies mit elementaren Graphikfunktionen erreichen.

[10] Dies ist die einzige Methode, wenn man eine isometrische Skalierung will.

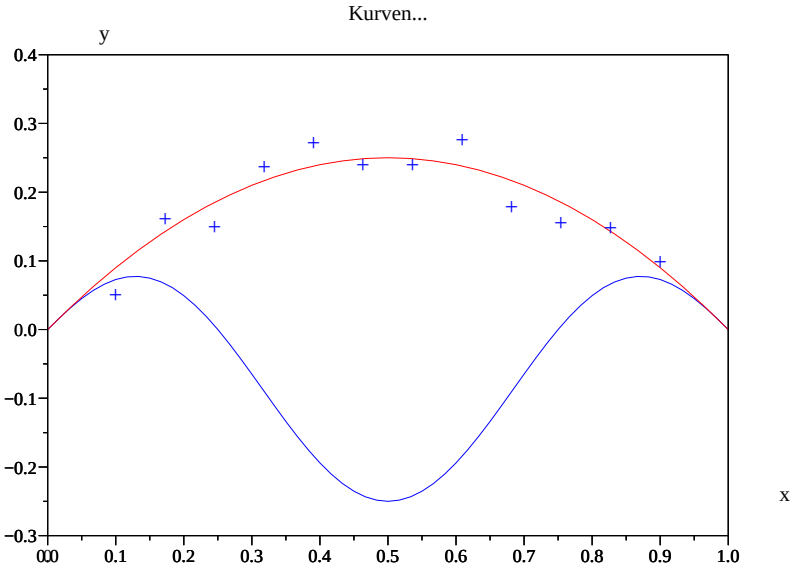

Abbildung 4.9: noch mehr Kurven...

4.6 Änderungen des Graphik–Kontextes

Als Sie die bisherigen Beispiele ausprobiert haben, kam Ihnen sicher einmal der Wunsch, die Größe von Symbolen und Zeichen oder den verwendeten Zeichensatz (font), z.B. Fettdruck, im Titel zu ändern.

1. **Zeichensätze** ändert man mit:

   ```
   xset("font",font_id,fontsize_id)
   ```

   wobei `font_id` und `fontsize_id` ganze Zahlen sind, die einen Zeichensatz bzw. die Größe des Zeichensatzes auswählen. Den z.Z. benutzten Zeichensatz erhält man mittels:

   ```
   f=xget("font")
   ```

   mit einem Vektor `f`, wobei `f(1)` die Nummer des Zeichensatzes und `f(2)` die Zeichengröße enthält. Die Zeichengröße kann man mittels `xset("font size,size_id)` setzen und mit `fontsize_id= xget("font size")` abfragen.

   Die folgende Tabelle gibt die z.Z. gültige Nummerierung der Zeichensätze an:

Zeichensatz	Courier	Symbol	Times	Times-Italic	Times-Bold	Times-Bold-Italic
Nummer	0	1	2	3	4	5

Größe	8 pts	10 pts	12 pts	14 pts	18 pts	24 pts
Nummer	0	1	2	3	4	5

   Bemerkungen:

   - der Zeichensatz Courier hat einen unveränderlichen Zeichenabstand.
   - der Zeichensatz Symbol ermöglicht z.B. griechische Zeichen (p ergibt π, a ergibt α, etc.).
   - Times ist der Standardzeichensatz in einer Größe von 10 Punkten.

2. **Größe von Symbolen** ändert man mit

```
xset("mark size",marksize_id)
```

und man kann die z.Z. benutzte Größe mit

```
marksize_id = xget("mark size")
```

abfragen. Wie bei den Größen der Zeichensätze sind auch hier die Angaben 0 bis 5 zugelassen, wobei 0 die voreingestellte Größe ist.

3. **Strichstärken** ändert bzw. erhält man mit

```
xset("thickness",thickness_id)
thickness_id = xget("thickness")
```

thickness_id ist dabei ganzzahlig und entspricht der Zahl der Pixel, die zum Zeichnen eines Zeichens verwendet werden (standardmäßig 1). Ein klassisches Problem dabei ist, dass dieser Parameter auch auf die Strichstärke beim Zeichnen des umgebenden Rahmens und der Einteilung der Achsen wirkt. Als Ausweg kann man die Kurven zunächst ohne Rahmen und Graduierungen (`axesflag=0`) zeichnen, dann setzt man die Strichstärke zurück und zeichnet im alten Maßstab (`frameflag=0`), jetzt mit Rahmen und Graduierungen, etwas, was nicht in der Zeichnung erscheint (, weil es außerhalb des sichtbaren Bereiches liegt, wie z.B. einen einzigen Punkt mit den Koordinaten $(-\infty, -\infty)$)

```
xset("thickness", 3)
plot2d(x, y, axesflag=0, ...)
xset("thickness", 1)
plot2d(-%inf, -%inf, frameflag=0, ...)
```

Der zweite Aufruf dient also nur dazu, den Rahmen und die Graduierungen zu erzeugen.

4.7 Zeichnen eines Histogramms

Die adäquate Scilabfunktion heißt `histplot`, und ihre Syntax lautet:

```
histplot(n, X, <,opt_arg>*)
```

wobei

- `n` entweder eine ganze Zahl oder ein Zeilenvektor (mit $n_i < n_{i+1}$) ist
  1. Wenn `n` ein Zeilenvektor ist, werden die Daten gemäß den k Klassen $C_i = [n_i, n_{i+1})$ aufgeteilt (der Vektor `n` hat also $k+1$ Komponenten)
  2. Wenn `n` eine ganze Zahl ist, werden die Daten in n äquidistante Klassen aufgeteilt:

$$C_1 = [c_1, c_2],\ C_i =]c_i, c_{i+1}],\ i = 2, ..., n,\ \text{mit} \begin{cases} c_1 = min(X),\ c_{n+1} = max(X) \\ c_{i+1} = c_i + \Delta C \\ \Delta C = (c_{n+1} - c_1)/n \end{cases}$$

- `X` der Datenvektor (Zeile oder Spalte) ist.
- `<,opt_arg>*` Optionen genauso wie `plot2d` enthalten kann. Hat die neue Option *normalization = Wert* (boolesche Konstante, Variable oder Ausdruck) den Wert `%T` (Voreinstellung), so hat das Histogramm das Integral 1, sodass es eine Verteilungsdichte approximiert. Im anderen Fall (`Wert=%F`) entspricht die Höhe eines Balkens der Anzahl derjenigen Komponenten des Vektors `X`, die in dieses Intervall fallen; genauer berechnet sich die Höhe des Balkens zu dem Intervall C_i (mit der Zahl der Daten m und $\Delta C_i = n_{i+1} - n_i$):

$$\begin{cases} \frac{\text{card } \{X_j \in C_i\}}{m \Delta C_i} & \text{falls } normalization = \texttt{\%T} \\ \text{card } \{X_j \in C_i\} & \text{falls } normalization = \texttt{\%F} \end{cases}$$

Hier ein kleines Beispiel, immer noch mit der Normalverteilung (vgl. Abb. 4.10):

```
X = rand(100000,1,"normal"); klassen = linspace(-5,5,21);
histplot(klassen,X)
// diesem überlagert man den Graph der Dichte der N(0,1)--Verteilung
x = linspace(-5,5,60)'; y = exp(-x.^2/2)/sqrt(2*%pi);
 plot2d(x,y, style=2, frameflag=0, axesflag=0)
```

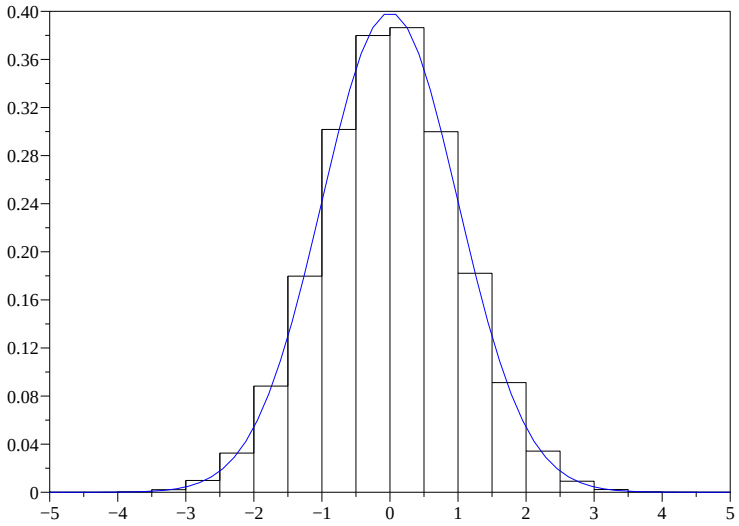

Abbildung 4.10: Histogramm einer Stichprobe von $N(0,1)$–verteilten Zufallszahlen

4.8 Abspeichern von Graphiken in mehreren Formaten

Dies ist mit Hilfe des Menus `File` des Graphikfensters sehr einfach; der Menupunkt `Export` bietet ein weiteres Menu verschiedener Möglichkeiten zu (Varianten von) Postscript sowie das `fig`–Format, was Ihnen ermöglicht, Ihre Graphik mit dem Graphikprogramm `xfig` nachzubearbeiten. Ab der Version 2.5 können Sie auch `gif`–Dateien exportieren.

4.9 Einfache Animationen

Es ist leicht, kleine Animationen mit Scilab zu realisieren, wenn man Doppelpuffer (double buffer) benutzt; diese verhindern ein Flimmern, wenn man Teilbilder vor einem festen Hintergrund überlagert, um eine zeitliche Entwicklung darzustellen. Außerdem gibt es verschiedene Treiber, die die Anzeige am Bildschirm ermöglichen[11]:

- `Rec` speichert alle im Fenster vorgenommenen Graphikoperationen und ist der Standardtreiber.
- `X11` begnügt sich lediglich damit, Graphiken anzuzeigen. (Es ist also nicht möglich zu zoomen.)

Für eine Animation, die viele Bilder enthält, ist es ratsamer, den letzteren zu verwenden, was man mit der Anweisung `driver("X11")` erreichen kann (mit `driver("Rec")` kehrt man zum Standardtreiber zurück). Bei Benutzung von Doppelpuffern (double buffer) wird jedes Teilbild zunächst nur im Hauptspeicher erzeugt (sogen. Pixmap) und erst dann zur Anzeige gebracht, wenn es vollständig ist. Dies geschieht in etwa nach folgendem Muster:

[11] und darüber hinaus welche zum Erzeugen einer Postscript, Fig oder GIF–Datei.

```
driver("X11")      // noch keine Graphikoperationen
xset("pixmap",1)   // wähle den double buffer Modus
...........        // ggf.\ eine Aufruf zur Festlegung des Maßstabes
for i=1:Anz_Zeichnungen
   xset("wwpc")    // Löschen der Pixmap
   .......
   .......         // Erstellung des i--ten Teilbildes
   .......
   xset("wshow")   // Anzeige der Pixmap
end
xset("pixmap",0)   // Abschalten des double buffer Modus
driver("Rec")      // Rückkehr zum Standardtreiber
```

Bemerkung: Man muss nicht unbedingt `xset("wwpc")` benutzen, das die Pixmap vollständig löscht; man kann z.B. nur einen rechteckigen Teil mit Hilfe von `xclea` löschen, sodass man statische Teile des Bildes nicht immer wieder neu generieren muss. Beim Zeichnen werden in der Regel bisherige Pixel durch neue *ersetzt*; man kann jedoch statt dessen eine boolesche Kombination aus dem alten und neuen Wert setzen. Dazu verwendet man die Funktion `xset('alufunction',num)`, wobei `num` ein ganzzahliger Kode ist, der die gewünschte boolesche Operation auswählt; schauen Sie unter *Help* oder in den Demos nach!

Als Beispiel lassen wir den Schwerpunkt eines Rechteckes (der Länge L und der Breite l) auf einem Kreis mit Radius r um dem Mittelpunkt $(0, 0)$ rotieren, wobei sich das Rechteck zusätzlich noch um seine eigene Achse dreht. Folgende Details bedürfen noch einer Erklärung:

- die Anweisung `plot2d` legt nur den (isometrischen) Maßstab fest.
- `xset("background",1)` wählt die Hintergrundfarbe 1 (schwarz in der Standardpalette); man sollte zusätzlich mit `xbasr()` die neue Hintergrundfarbe wirksam werden lassen.
- die Zeichnung besteht aus dem Aufruf der Funktion `xfpoly` gefolgt von `xpoly`, um den Rand des Rechtecks zu zeichnen (hier wurde mit `xset("thickness",3)` eine Strichstärke von 3 Pixeln gewählt). Jedesmal wird mit `xset("color",num)` die verwendete Farbe geändert.
- die Anweisung `xset("default")` setzt den Graphik–Kontext in seine Voreinstellung zurück, also erhält `pixmap` den Wert 0, `thickness` den Wert 1, `background` seinen Standardwert, u.s.w.

```
n = 4000;
L = 0.6; l = 0.3; r = 0.7;
Anz_Umdrehungen = 4;
t = linspace(0,Anz_Umdrehungen*2*%pi,n)';
xg = r*cos(t); yg = r*sin(t);
xy = [-L/2  L/2 L/2 -L/2;...   // die 4 Randpunkte
      -l/2 -l/2 l/2  l/2];

xselect()
driver("X11")
xset("pixmap",1)
plot2d(%inf,%inf, frameflag=3, rect=[-1,-1,1,1], axesflag=0)
xset("background",1); // schwarzer Hintergrund
xbasr()               // aktualisiert die Hintergrundfarbe
xset("thickness",3)   // Strichstärke von 3 Pixeln

xset("font",2,4)
for i=1:n
   xset("wwpc")
   theta = 3*t(i);
   xyr = [cos(theta) -sin(theta);...
          sin(theta)  cos(theta)]*xy;
   xset("color",2)
   xfpoly(xyr(1,:)+xg(i), xyr(2,:)+yg(i))
   xset("color",5)
   xpoly(xyr(1,:)+xg(i), xyr(2,:)+yg(i),"lines",1)
```

```
    xset("color",32)
    xtitle("einfache Animation")
    xset("wshow")   // bringe die Pixmap zur Anzeige
end
driver("Rec")       // zurück zum Standardtreiber
xset("default")     // setze den Graphik-Kontext auf seine Voreinstellung
```

4.10 Flächen

`plot3d` ist die allgemeine Anweisung zum Zeichnen von Flächen[12]. Was die Darstellung Ihrer Fläche durch Facetten betrifft, erlauben diese beiden Anweisungen, jede einzelne Facette in einer anderen Farbe zu zeichnen. Seit der Scilab Version 2.6 ist es auch möglich, für dreieckige oder viereckige Facetten (nur) die Farbe der Ecken festzulegen, diese werden dann im Inneren der Facette interpoliert.

4.10.1 Einführung in plot3d

Ist Ihre Fläche durch eine Gleichung der Art $z = f(x, y)$ gegeben, ist es besonders einfach, sie über einem rechteckigen Parametergebiet darzustellen. Im folgenden Beispiel stelle ich die Funktion $f(x, y) = cos(x)cos(y)$ für $(x, y) \in [0, 2\pi] \times [0, 2\pi]$ dar:

```
x=linspace(0,2*%pi,31);   // Diskretisierung in x (und dieselbe auch in y)
z=cos(x)'*cos(x);  // z-Werte: eine Matrix z(i,j) = f(x(i),y(j))
plot3d(x,x,z)  // das Bild
```

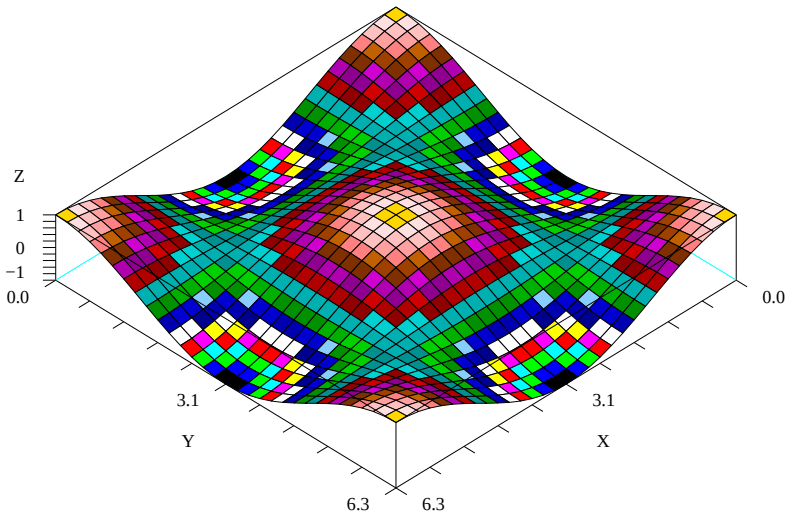

Abbildung 4.11: die Funktion $z = cos(x)cos(y)$

Sie werden etwas erhalten, was der Abbildung 4.11 ähnelt[13]. Allgemein benutzt man:

```
plot3d(x,y,z <,opt_arg>*)
plot3d1(x,y,z <,opt_arg>*)
```

[12] `plot3d1` wird beinahe auf identische Art und Weise verwendet, und erlaubt, Farben entsprechend dem Wert der dritten Koordinate z zu verwenden.

[13] außer, dass Farben mit `plot3d1` verwendet wurden (für dieses Dokument wurden sie in unterschiedliche Graustufen umgewandelt), und dass die Blickrichtung etwas anders gewählt ist.

Hierbei haben — wie bei `plot2d` — die Optionen die Form *Schlüsselwort=Wert*. In der einfachsten Form sind `x` und `y` zwei Zeilenvektoren (Dimension $(1, nx)$ und $(1, ny)$), welche den diskretisierten $x-$ und $y-$Werten entsprechen; `z` ist eine $nx \times ny$–Matrix so, dass $z_{i,j}$ die Höhe über dem Punkt (x_i, y_j) angibt.

Mögliche Optionen sind:

1. `theta=`*val_theta* und `alpha=`*val_alpha* sind zwei Winkel (in Grad), die die Blickrichtung in sphärischen Koordinaten festlegen (sei O der Mittelpunkt einer alles einschließenden Kugel und Oc die Blickrichtung der Kamera, dann ist $\alpha = Winkel(Oz, Oc)$ und $\theta = Winkel(Ox, Oc')$, wobei Oc' die Projektion von Oc auf die Ebene Oxy ist.

2. `leg=`*val_leg* spezifiziert die Beschriftung der Achsen, z.B. `leg="x@y@z"`, wobei `@` als Trennzeichen benutzt wird.

3. `flag=`*val_flag*, wobei *val_flag* ein Vektor mit drei Komponenten `flag=[mode type box]` ist:

   (a) der Parameter `mode` darüber entscheidet, ob die verdeckten Flächen gezeichnet werden oder nicht:

      i. mit `mode > 0` werden die verdeckten Flächen entfernt[14], das Parameternetz bleibt sichtbar.

      ii. mit `mode = 0` erhält man ein "Drahtnetzmodell" der Fläche.

      iii. mit `mode < 0` werden die verdeckten Flächen entfernt und es wird kein Parameternetz gezeichnet.

   Die "positive" Seite einer Fläche (Definition weiter unten) wird in der Farbe mit der Nummer $|mode|$ gezeichnet; die Farbe (Voreinstellung: 4) der gegenüberliegenden Seite kann man mittels `xset("hidden3d",colorid)` setzen.

   (b) der Parameter *type* legt den Maßstab fest:

type	verwendeter Maßstab
0	zuletzt benutzter Maßstab (Voreinstellung)
1	Maßstab wird durch `ebox` festgelegt
2	der Maßstab errechnet sich aus dem Max und Min der Daten
3	isometrischer Maßstab abgeleitet von `ebox`
4	isometrischer Maßstab abgeleitet aus dem Max und Min der Daten
5	Variante von 3
6	Variante von 4

   (c) der Parameter `box` kontrolliert die Umgebung rund um den Graphen:

box	Wirkung
0	nur die Fläche selbst wird gezeichnet
2	die Achsen unter der Fläche werden gezeichnet
3	wie 2 mit einem Kasten um die Fläche herum
4	wie 3, jetzt mit einer Graduierung der Achsen

4. `ebox=`*val_ebox* legt die Größe des umgebenden Kastens fest, *val_ebox* ist ein Vektor mit 6 Komponenten $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$.

Hier ein kleines Skript, in dem fast alle Parameter von `plot3d` gebraucht werden. Es handelt sich um eine Animation, die Ihnen helfen soll, die Änderung der Blickrichtung mit den Parametern `theta` und `alpha` zu verstehen. Im Skript benutze ich `flag=[2 4 4]`; dies bewirkt:

- *mode* = 2 Die Fläche (positive Seite) wird in der Farbe Nr. 2 zusammen mit dem Parameternetz gezeichnet.

- *type* = 4 liefert eine isometrische Skalierung, die den Daten angepasst wird; dies ist wie *type* = 3 mit dem Parameter *ebox*, der sich aus den Minima und Maxima der Daten ergäbe.

[14] Dies wird z.Z. dadurch erreicht, dass die am weitesten vom Betrachter entfernten Facetten zuerst gezeichnet werden.

- *box* = 4 Es wird eine umgebende Quader gezeichnet, und die Achsen tragen Graduierungen.

```
x=linspace(-%pi,%pi,31);
z=sin(x)'*sin(x);
n = 200;
theta = linspace(30,390,n);  // einmal im Kreis
alpha = [linspace(60,0,n/2) linspace(0,80,n/2)]; // erst nach oben
                                                 // dann nach unten
xselect()
xset("pixmap",1)   // Aktivierung des Doppelpuffers
plot3d(x,x,z,theta(1),alpha(1),"x@y@z",[2 2 4])
xtitle("Variation der Blickrichtung mit dem Parameter theta")
xset("wshow")
driver("X11")
// variiere theta
for i=1:n
  xset("wwpc") // löscht den aktuellen Puffer
   plot3d(x,x,z,theta=theta(i),alpha=alpha(1),leg="x@y@z",flag=[2 4 4])
  xtitle("Variation der Blickrichtung mit dem Parameter theta")
  xset("wshow")
end
// variiere alpha
for i=1:n
  xset("wwpc") // löscht den aktuellen Puffer
   plot3d(x,x,z,theta=theta(n),alpha=alpha(i),leg="x@y@z",flag=[2 4 4])
  xtitle("Variation der Blickrichtung mit dem Parameter alpha")
  xset("wshow")
end
xset("pixmap",0)  // Rückkehr zum normalen Modus
driver("Rec")     // Rückkehr zum Standardtreiber
```

4.10.2 Farbgebung

Sie können die beiden vorigen Beispiele noch einmal ausprobieren, wobei Sie `plot3d` durch `plot3d1` ersetzen, welches die Farben entsprechend dem z-Wert auswählt. Ihre Fläche wird einem Mosaik ähneln, weil die Standardfarbpalette nicht kontinuierlich ist. Eine Farbpalette wird durch eine Matrix `C` der Dimension `(Anz_Farben,3)` gegeben; die i–te Zeile spezifiziert die Intensität (eine reelle Zahl zwischen 0 und 1) der Farbe Rot, Grün bzw. Blau. Mit der Anweisung `xset("colormap",C)` aktiviert man diese Palette für das aktuelle Graphikfenster. Es gibt zwei Funktionen, `hotcolormap` und `greycolormap`, die (fast) kontinuierliche Farbpaletten liefern[15]. Kurze Bemerkung: Wird die Farbpalette verändert, nachdem ein Graph gezeichnet wurde, werden Sie die Änderungen nicht unmittelbar in Ihrer Zeichnung wiederfinden (was normal ist). Es reicht, zum Beispiel, die Größe des Graphikfensters zu verändern bzw. den Befehl `xbasr(Nr_Fenster)` anzugeben, um das erneute Zeichnen zu veranlassen (unter Benutzung der neuen Farbpalette). Jetzt noch einmal das erste Beispiel

```
x = linspace(0,2*%pi,31);
z = cos(x)'*cos(x);
C = hotcolormap(32); // die hotcolormap Palette mit 32 Abstufungen
xset("colormap",C)
xset("hidden3d",30)  // Farbe Nummer 30 für die negativen Seite der Fläche
xbasc()
plot3d1(x,x,z, flag=[1 4 4])  // probieren Sie dies auch mit flag=[-1 4 4] aus
```

Bemerkung: Bei `plot3d1` wird nur das Vorzeichen des Parameters *mode* ausgewertet (bei *mode* ≥ 0 erscheint das Parameternetz, bei *mode* < 0 dagegen nicht).

[15] siehe auch den Abschnitt Contributions auf der Homepage von Scilab.

4.10.3 `plot3d` und `plot3d1` mit Facetten

Um diese Funktionen in einem allgemeineren Kontext zu benutzen, müssen Sie Ihre Fläche durch Facetten beschreiben. Dies geschieht mit drei Matrizen `xf, yf, zf` mit der Dimension (`anz_Ecken_pro_Seite, anz_Seiten`), wobei `xf(j,i),yf(j,i),zf(j,i)` die Koordinaten der j–ten Ecke der i–ten Facette sind. Modulo dieser kleinen Änderung verhalten sich diese Funktionen wie vorher bzgl. der übrigen Argumente:

```
plot3d(xf,yf,zf <,opt_arg>*)
```

Beachten Sie, dass die Orientierung der Facetten anders ist, als man es sonst gewöhnt ist (siehe Abb. 4.12).

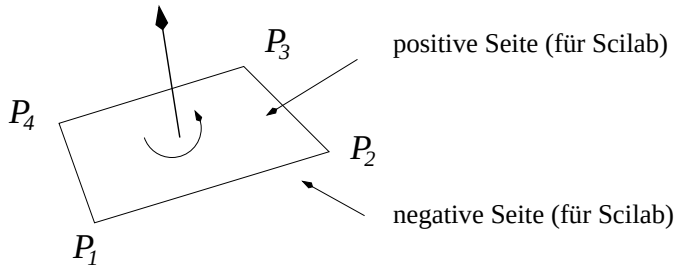

Abbildung 4.12: Orientierung von Facetten in Scilab

Man legt für jede Facette eine (eigene) Farbe fest, indem man als drittes Argument (statt `zf`) eine Liste angibt: `list(zf,colors)`, wobei `colors` ein Vektor der Dimension `Anz_Facetten` und `colors(i)` die Nummer der Farbe (in der aktuellen Palette) der i–ten Facette ist.

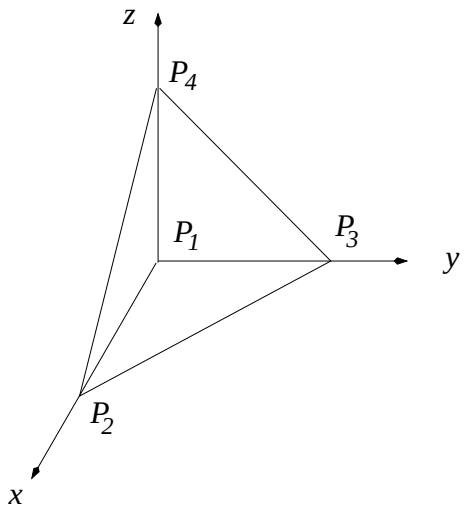

Abbildung 4.13: ein Tetraeder

Wir visualisieren nun die Seiten des Tetraeders (siehe Abb. 4.13), für den

$$P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},\ P_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ P_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ P_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

gilt; weiter definieren wir die Seitenflächen so, dass die äußere Normale mit der positiven Orientierung von Scilab zusammenfällt:

$$f_1 = (P_1, P_2, P_3),\ f_2 = (P_2, P_4, P_3),\ f_3 = (P_1, P_3, P_4),\ f_4 = (P_1, P_4, P_2)$$

Damit erhält man:

```
//    f1 f2 f3 f4
xf = [ 0  1  0  0;
       1  0  0  0;
       0  0  0  1];
yf = [ 0  0  0  0;
       0  0  1  0;
       1  1  0  0];
zf = [ 0  0  0  0;
       0  1  0  1;
       0  0  1  0];

xbasc()
plot3d(xf,yf,list(zf,2:5), flag=[1 4 4], leg="x@y@z",alpha=30, theta=230)
xselect()
```

Mit diesen Parametern sollten Sie ein Bild wie Abb. 4.14 erhalten. Sie haben vielleicht bemerkt, dass `plot3d` eine einfache Orthogonal–Projektion statt einer realistischeren Zentral–Projektion verwendet.

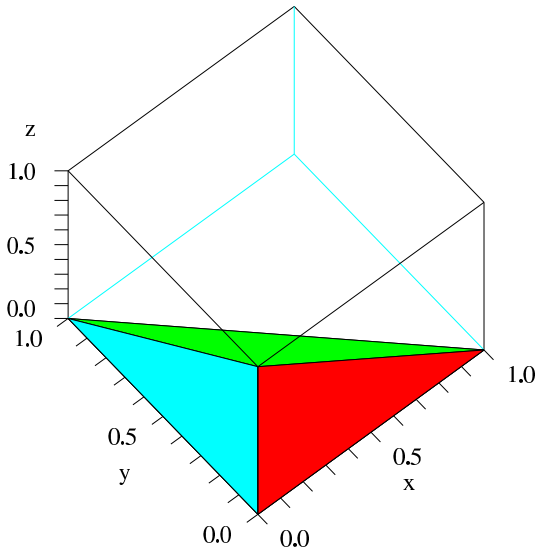

Abbildung 4.14: Tetraeder mit Scilab gezeichnet

Man kann Facetten mit den folgenden beiden Funktionen erhalten:

- `eval3dp` und `nf3d` für Flächen in Parameterform: $x = x(u,v), y = y(u,v), z = z(u,v)$ (siehe 4.10.4)
- `genfac3d` für Flächen, die durch $z = f(x,y)$ gegeben sind (ein Beispiel geben wir weiter unten (4.10.5)).

Wenn Ihre Fläche (Polyeder) auf die Art und Weise gegeben ist, wie in unserem Würfelbeispiel zu typisierten Listen, so kann man sie nicht einfach mit `plot3d` visualisieren; dann kann man die folgende Funktion benutzen:

```
function [xf,yf,zf] = Poyeder_Facetten(P)
   // transformiere die Polyeder-Struktur des Bespiels
   // zu den typisierten Listen in eine Darstellung, die
   // für plot3d geeignet ist.
   [ne, nf] = size(P.Seite)  // ne : Zahl der Ecken pro Facette
                             // nf : Zahl der Facetten
   xf=zeros(P.Seite); yf=zeros(P.Seite); zf=zeros(P.Seite)
   for j=1:ne
      num =  connect(ne+1-j,:) // Ändern der Orientierung
```

```
        xf(j,:) = P.coord(1, num)
        yf(j,:) = P.coord(2, num)
        zf(j,:) = P.coord(3, num)
      end
   endfunction
```

Damit können wir das Objekt Wuerfel von früher wie folgt zeichnen:

```
[xf,yf,zf] = Poyeder_Facetten(Wuerfel);
plot3d(xf,yf,list(zf,2:7), flag=[1 4 0],theta=50,alpha=60)
```

4.10.4 Zeichnen einer durch $x = f_1(u, v)$, $y = f_2(u, v)$, $z = f_3(u, v)$ definierten Fläche

Dazu nehme man eine Diskretisierung des Parameterbereiches und berechne die Facetten mit der Funktion `eval3dp`. Aus Effizienzgründen sollte Ihre Funktion, die die Parametrisierung der Fläche definiert, "vektoriell" sein. Wenn $(u_1, u_2, \ldots, u_m)$ und $(v_1, v_2, \ldots, v_n)$ die Diskretisierungen eines Parameterrechteckes sind, so wird Ihre Funktion nur einmal mit zwei großen Vektoren der Länge $m \times n$ gerufen:

$$U = (\underbrace{u_1, u_2, \ldots, u_m}_{1}, \underbrace{u_1, u_2, \ldots, u_m}_{2}, \ldots\ldots, \underbrace{u_1, u_2, \ldots, u_m}_{n})$$
$$V = (\underbrace{v_1, v_1, \ldots, v_1}_{m \text{ fach } v_1}, \underbrace{v_2, v_2, \ldots, v_2}_{m \text{ fach } v_2}, \ \ldots\ \ldots\ , \underbrace{v_n, v_n, \ldots, v_n}_{m \text{ fach } v_n})$$

Daraus sollte Ihre Funktion 3 Vektoren X, Y und Z der Länge $m \times n$ gemäß

$$X_k = x(U_k, V_k), Y_k = y(U_k, V_k), Z_k = z(U_k, V_k)$$

berechnen und zurückgeben — es folgen einige Beispiele[16], die mit `eval3dp` benutzt werden können:

```
function [x,y,z] = Torus(theta, phi)
  // klassische Parametrisierung des Torus mit den Radien R und r und der Achse Oz
  R = 1; r = 0.2
  x = (R + r*cos(phi)).*cos(theta)
  y = (R + r*cos(phi)).*sin(theta)
  z = r*sin(phi)
endfunction

function [x,y,z] = Schraub_Torus(theta, phi)
  // Parametrisierung eines Schraub-Torus
  R = 1; r = 0.3
  x = (R + r*cos(phi)).*cos(theta)
  y = (R + r*cos(phi)).*sin(theta)
  z = r*sin(phi) + 0.5*theta
endfunction

function [x,y,z] = moebius(theta, rho)
  // Parametrisierung eines Möbius-Bandes
  R = 1;
  x = (R + rho.*sin(theta/2)).*cos(theta)
  y = (R + rho.*sin(theta/2)).*sin(theta)
  z = rho.*cos(theta/2)
endfunction

function [x,y,z] = verbeulter_Torus(theta, phi)
  // Parametrisierung eines Torus, dessen kleiner Radius mit theta variiert
```

[16] Sie können die Funktion erst ganz normal schreiben und dann `*` bzw. `/` durch `.*` bzw. `./` ersetzen.

```
  R = 1; r = 0.2*(1+ 0.4*sin(8*theta))
  x = (R + r.*cos(phi)).*cos(theta)
  y = (R + r.*cos(phi)).*sin(theta)
  z = r.*sin(phi)
endfunction
```

und hier ein Beispiel, das die letzte Fläche graphisch darstellt:

```
// Skript zum Zeichnen einer Fläche, die durch Parametergleichungen definiert ist
theta = linspace(0, 2*%pi, 160);
phi = linspace(0, -2*%pi, 20);
[xf, yf, zf] = eval3dp(verbeulter_Torus, theta, phi); // Berechnung der Facetten
xbasc()
plot3d(xf,yf,zf)
xselect()
```

Sollten Sie Farben verwenden, sie aber nicht erhalten, so ist die Orientierung nicht richtig: es reicht dann, die Orientierung von einem der beiden Diskretisierungsvektoren des Parametergebietes umzukehren.

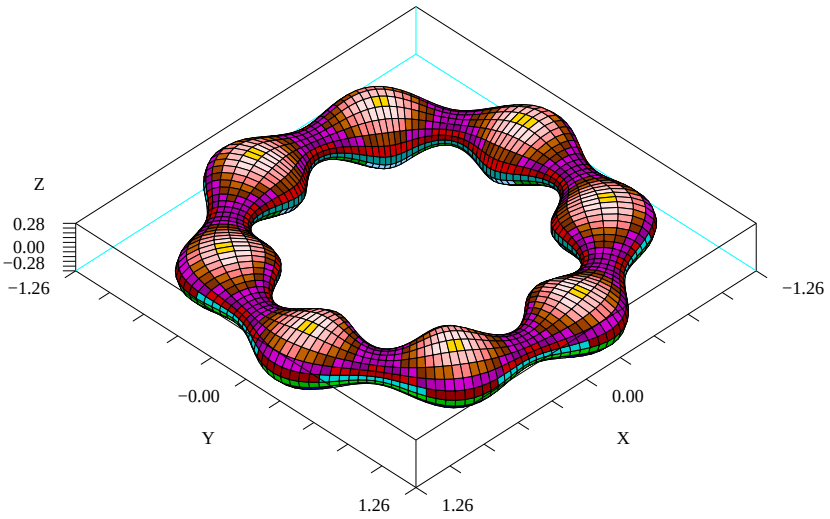

Abbildung 4.15: Ein verbeulter Torus

Die Funktion `nf3d` ist ähnlich zu `eval3dp`, aber, gegeben eine Diskretisierung von u und v, sollten Sie die Matrizen X, Y, Z so berechnen, dass

$$X_{i,j} = x(u_i, v_j)$$
$$Y_{i,j} = y(u_i, v_j)$$
$$Z_{i,j} = z(u_i, v_j)$$

gilt. Die Facetten erhalten Sie dann mit `[xf,yf,zf] = nf3d(X,Y,Z)`. Als Beispiel diene das Möbius–Band:

```
nt = 120;
nr = 10;
rho = linspace(-0.5,0.5,nr);
theta = linspace(0,2*%pi,nt);
R = 1;
X = (R + rho'*sin(theta/2)).*(ones(nr,1)*cos(theta));
```

```
Y = (R + rho'*sin(theta/2)).*(ones(nr,1)*sin(theta));
Z = rho'*cos(theta/2);
[xf,yf,zf] = nf3d(X,Y,Z);
xbasc()
plot3d(xf,yf,zf, flag=[2 4 6], alpha=60, theta=50)
xselect()
```

Bemerkung: Ich musste die Funktion `ones` benutzen, um die richtigen Matrizen zu erhalten, was die Funktion etwas schwieriger aussehen lässt; die Funktion `eval3dp` ist einfacher zu benutzen.

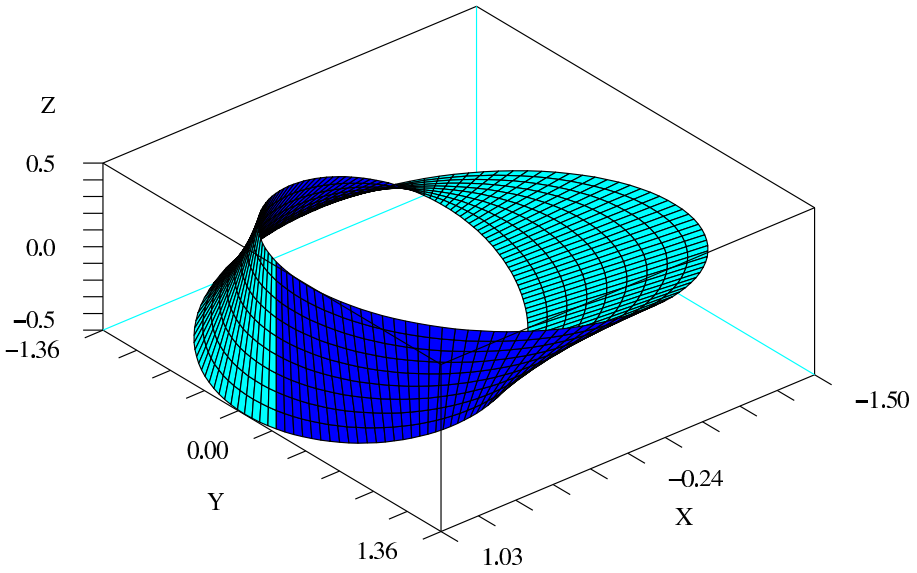

Abbildung 4.16: Moëbius-Band

4.10.5 `plot3d` mit Interpolation der Farben

Seit Version 2.6 ist es möglich, jeder Ecke einer Facette eine Farbe zuzuordnen. Dazu reicht es, eine Matrix `colors` der gleichen Dimension anzugeben wie die Matrizen `xf, yf, zf`, die die Beschreibung der Facetten festlegen, d.h. `colors(i,j)` ist die Farbe der i–ten Ecke der j–ten Seite. Als drittes Argument von `plot3d` gibt man jetzt eine Liste an:

```
plot3d(xf,yf,list(zf,colors) <,opt_arg>*)
```

Nun ein erstes Beispiel, das mit `plot3d` eine Fläche ohne Parameternetz, aber mit

- einer Farbe pro Seite im linken Bild
- einer Farbe pro Ecke im rechten Bild

zeichnet. Zur Berechnung der Farbe benutze ich eine kleine Funktion, die eine lineare Zuordnung der Werte zu Farben der aktuellen Farbpalette vornimmt (ich benutze die Funktion `dsearch`, die es seit Version 2.7 gibt, aber es ginge auch ohne). Beachten Sie die Verwendung der Funktion `genfac3d` zur Berechnung der Facetten.

```
// Bespiel zu plot3d mit Interpolation der Farben
function [col] = Farb_Zuordnung(val)
   // ordne jeder Komponente von val einen Farbwert zu
   n1 = 1                     // Nummer der 1. Farbe
   n2 =  xget("lastpattern") // Nummer der letzten Farbe
   Anz_Farben = n2 - n1 + 1
   classes = linspace(min(val),max(val),Anz_Farben)
   col = dsearch(val, classes)
endfunction
```

```
x=linspace(0,2*%pi,31);
z=cos(x)'*cos(x);
[xf,yf,zf] = genfac3d(x,x,z);
xset("colormap",graycolormap(64)) // Auswahl einer Graustufenpalette

zmeanf = mean(zf,"r");
zcolf = Farb_Zuordnung(zmeanf);
zcols = Farb_Zuordnung(zf);

xbasc()
xset("font",6,2)  // der Zeichensatz 6 (helvetica) ist nur in einer
                  // neueren Version von Scilab vorhanden
subplot(1,2,1)
   plot3d(xf,yf,list(zf,zcolf), flag=[-1 4 4])
   xtitle("Eine Farbe pro Seite")
subplot(1,2,2)
   plot3d(xf,yf,list(zf,zcols), flag=[-1 4 4])
   xtitle("Eine Farbe pro Ecke")
xselect()
```

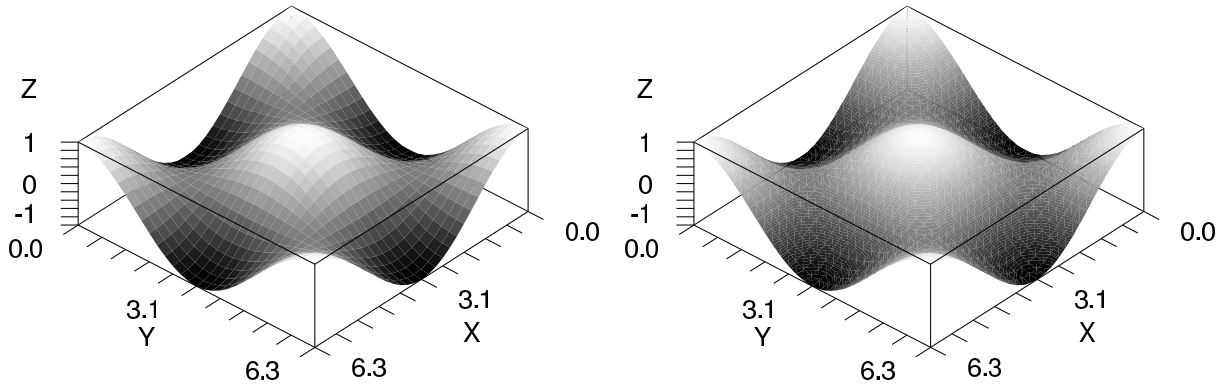

Abbildung 4.17: Mit und ohne Interpolation der Farben

4.11 Raumkurven

`param3d` dient als Basisanweisung, wenn man eine solche Kurve zeichnen will. Hier ist ein klassisches Beispiel für eine Helix:

```
t = linspace(0,4*%pi,100);
x = cos(t); y = sin(t) ; z = t;
param3d(x,y,z)  // Löschen Sie evtl. das Graphikfenster mit xbasc()
```

Da dieser Befehl nur eine einzige Kurve anzeigen kann, werden wir uns nun auf `param3d1` konzentrieren, das viel leistungsfähiger ist. Hier die entsprechende Syntax:

```
param3d1(x,y,z <,opt_arg>*)
param3d1(x,y,list(z,colors) <,opt_arg>*)
```

Die Matrizen `x, y` und `z` müssen vom gleichen Format (`np,nc`) sein, und die Anzahl der Kurven (`nc`) ist durch ihre Spaltenanzahl (wie für `plot2d`) gegeben. Die optionalen Parameter sind die gleichen wie die der Anweisung `plot3d`, mit Ausnahme der nicht vorhandenen Optionen `flag` und `mode`. Weiter ist `colors` ein Vektor, der den Stil der Ausgabe jeder Kurve angibt (genauso wie für `plot2d`), d.h. wenn `colors(i)` eine positive ganze Zahl ist, wird die i–te Kurve mit der i–ten Farbe der aktuellen Farbpalette gezeichnet (bzw. mit unterschiedlichen Strichtypen auf einem Schwarzweißmedium), während man für einen Wert zwischen -9 und 0 eine Anzeige von (nicht verbundenen) Punkten erhält, die diese Kurven durch entsprechende Symbole darstellt. Hier ein Beispiel, das zur Abbildung 4.18 führt:

```
t = linspace(0,4*%pi,100)';
x1 = cos(t); y1 = sin(t) ; z1 = 0.1*t;  // eine Helix
x2 = x1 + 0.1*(1-rand(x1));
y2 = y1 + 0.1*(1-rand(y1));
z2 = z1 + 0.1*(1-rand(z1));
xbasc();
xset("font",2,3)
param3d1([x1 x2],[y1 y2],list([z1 z2], [1,-9]), flag=[4 4])
xset("font",4,4)
xtitle("Schraubenlinie mit Perlen")
```

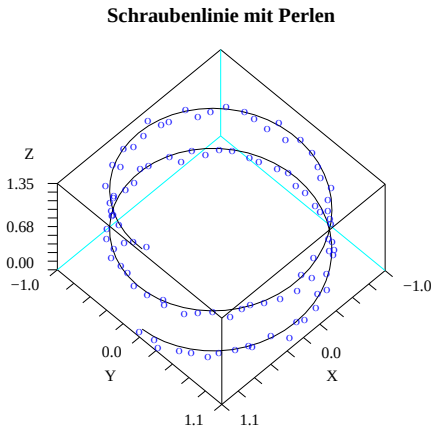

Abbildung 4.18: Kurve und Punkte im Raum

Wie bei `plot2d` ist man auch hier gezwungen, den Befehl mehrmals aufzurufen, wenn die zu zeichnenden Kurven nicht dieselbe Anzahl von Punkten haben. Im Folgenden wird ein Skript vorgestellt, das zeigt, wie zwei Gruppen von Punkten mit verschiedenen Symbolen bzw. Farben gezeichnet werden können:

```
n = 50;        // Anzahl der Punkte
P = rand(n,3); // Zufallszahlen
// Daten des umschließenden Quaders
ebox = [0 1 0 1 0 1];
// Trennen der Punkte in zwei Gruppen, um zu zeigen, wie man verschiedene
// Symbole und Farben erhalten kann
m = 30;
P1 = P(1:m,:) ; P2 = P(m+1:n,:);

xbasc()
// erste Gruppe von Punkten
xset("color",2)  // blau mit der Standardfarbpalette
param3d1(P1(:,1),P1(:,2),list(P1(:,3), -9), alpha=60, theta=30,...
         leg="x@y@z", flag=[3 4], ebox=ebox)
          // flag=[3 4] : 3 -> isometrische Skalierung basierend auf ebox
          //              4 -> umgebender Quader und Graduierungen
                  // für die 2. Gruppe
xset("color",5)  // rot  mit der Standardfarbpalette
param3d1(P2(:,1),P2(:,2),list(P2(:,3), -5), flag=[0 0])
          // -5 markiert mit einer kleinen Raute
          // [0 0]  : Maßstab des vorherigen Aufrufes
xset("color",1)  // Schwarz für Titel und Beschriftung
xtitle("Punkte...")
xselect()
```

4.12 Diverses

Es gibt noch mehr Graphik–Grundbefehle, darunter:

1. `contour2d` und `contour`, um Isolinien einer Funktion $z = f(x, y)$ zu zeichnen, die auf einem Rechteck definiert ist;
2. `grayplot` und `Sgrayplot`, die gestatten, die Werte einer solchen Funktion durch Farben darzustellen;
3. `fec` entspricht den beiden vorherigen Anweisungen für eine Funktion, die auf einer ebenen Triangulierung definiert ist;
4. `champ` zeichnet ein Vektorfeld in 2D.
5. und schließlich mehrere Graphikfunktion, wenn man statt diskreten Daten eine Funktion (geschrieben in Scilab) als Parameter übergeben will; deren Namen fangen alle mit dem Buchstaben f an: `fplot2d, fcontour2d, fplot3d, fplot3d1, fchamp`,....

Um sich die vielfältigen Möglichkeiten[17] vor Augen zu führen, reicht es, die Rubrik `Graphic Library` des Hilfesystems zu durchstöbern. Sie können sich auch die Bibliothek von Enrico Ségré beschaffen,

`http://www.weizmann.ac.il/~fesegre/`

die die Graphikfunktionen von Scilab ergänzen, und einige Aufgaben erleichtern.

[17] Dabei kann man leicht untergehen.

Kapitel 5

Einige Anwendungen und Ergänzungen

Dieses Kapitel soll Ihnen zeigen, wie man bestimmte Probleme aus dem Bereich der Numerischen Analysis mit Scilab löst (eigentlich z.Z. nur Differentialgleichungen), und es bringt einige Ergänzungen, um einfache stochastische Simulationen zu ermöglichen.

5.1 Differentialgleichungen

Scilab stellt ein sehr leistungsfähiges Interface zum nummerischen(d.h. approximativen) Lösen von Differentialgleichungen mit dem Grundbefehl `ode` bereit. Gegeben sei eine Differentialgleichung mit einer Anfangsbedingung:

$$\begin{cases} u' = f(t,u) \\ u(t_0) = u_0 \end{cases}$$

wobei $u(t)$ ein Vektor aus $\mathbb{R}^n$ ist, f eine Funktion der Gestalt $\mathbb{R} \times \mathbb{R}^n \longrightarrow \mathbb{R}^n$, und $u_0 \in \mathbb{R}^n$. Man nimmt an, dass die Bedingungen für Existenz und Eindeutigkeit der Lösung bis zu einem Zeitpunkt T erfüllt sind.

5.1.1 Basisanwendung von `ode`

In seiner elementarsten Funktionsweise ist es sehr einfach zu benutzen: man muss eine rechte Seite f als Scilabfunktion mit folgender Syntax schreiben:

```
function f = MeineRechteSeite(t,u)
   // hier der Programmcode, der die Komponenten von f als Funktion von
   // t und den Komponenten von u enthält
endfunction
```

Bem.: Auch wenn die Gleichung autonom ist, muss t dennoch als erstes Argument von `MeineRechteSeite` angegeben werden; z.B. für die rechte Seite der Van–der–Pol–Gleichung:

$$y'' = c(1-y^2)y' - y$$

die man in ein System von zwei Differentialgleichungen erster Ordnung umformuliert, indem man $u_1(t) = y(t)$ und $u_2(t) = y'(t)$ setzt:

$$\frac{d}{dt}\begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} = \begin{bmatrix} u_2(t) \\ c(1-u_1^2(t))u_2(t) - u_1(t) \end{bmatrix}$$

```
function f = VanDerPol(t,u)
  // rechte Seite für die Van-der-Pol-Gleichung (c = 0.4)
  f(1) = u(2)
  f(2) = 0.4*(1 - u(1)^2)*u(2) - u(1)
endfunction
```

Um die Gleichung von t_0 bis T mit Anfangswert u_0 (ein Spaltenvektor) zu lösen (integrieren) und die Lösung zu den Zeitpunkten $t(1) = t_0$, $t(2)$, ..., $t(m) = T$ zu erhalten, ruft man `ode` folgendermaßen auf:

```
t = linspace(t0,T,m);
U = ode(u0,t0,t,MeineRechteSeite)
```

Man erhält also eine „Matrix“ `U` vom Format (n, m), so dass `U(i,j)` eine approximative Lösung von $u_i(t(j))$ (die i–te Komponente zum Zeitpunkt $t(j)$) ist. *Bem.:* Die Anzahl der Komponenten, die man für `t` nimmt (die Zeitpunkte, für die man die Lösung erhält) hat nichts mit der Genauigkeit der Berechnung zu tun. Dies kann man mit anderen Parametern steuern (die Standardwerte haben). Außerdem verbergen sich hinter `ode` mehrere mögliche Algorithmen, die es erlauben, sich an verschiedene Situationen anzupassen... Um ein besonderes Verfahren auszuwählen, muss man beim Aufruf einen Parameter hinzufügen (vgl. `Help`). In der Regel (d.h. ohne eines dieser Verfahren explizit auszuwählen) wählt `ode` eine intelligente Strategie, bei der es zu Beginn ein Adams–Prädiktor–Korrektor–Verfahren verwendet, aber dann auch in der Lage ist, diesen Algorithmus gegen das Gear–Verfahren auszuwechseln, falls sich die Gleichung als steif[1] erweist. Hier ein vollständiges Beispiel der Van–der–Pol–Gleichung. Da in diesem Fall der Phasenraum eine Ebene ist, kann man bereits eine Vorstellung von der Dynamik entwickeln, indem man einfach das Vektorfeld in einem $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$–Rechteck mit dem Graphikbefehl `fchamp` zeichnet, dessen Syntax folgendermaßen aussieht:

```
fchamp(MeineRechteSeite,t,x,y)
```

wobei `MeineRechteSeite` der Name einer Scilabfunktion ist, die die rechte Seite der Differentialgleichung darstellt, `t` der Zeitpunkt, für den das Feld gezeichnet werden soll (im oft üblichen Fall einer autonomen Gleichung setzt man einen Dummy–Wert, z.B. 0, ein) und `x` sowie `y` Zeilenvektoren mit nx bzw ny Komponenten sind, die die Punkte des Gitters angeben, in denen die Pfeile gezeichnet werden, die das Vektorfeld repräsentieren.

```
// 1.) Zeichnung des Vektorfeldes zur Van-der-Pol-Gleichung
n = 30;
delta = 5
x = linspace(-delta,delta,n); // hier y = x
xbasc()
fchamp(VanDerPol,0,x,x)
xselect()

// 2.) Lösung der Differentialgleichung
m = 500 ; T = 30 ;
t = linspace(0,T,m);  // Zeitpunkte, für die man die Lösung erhält
u0 = [-2.5 ; 2.5];    // Anfangsbedingung
u = ode(u0, 0, t, VanDerPol);
plot2d(u(1,:)',u(2,:)',2,"000")
```

5.1.2 Van der Pol noch einmal

In diesem Abschnitt wird man sich eine graphische Möglichkeit von Scilab zunutze machen, um soviele Trajektorien zu erhalten wie gewünscht, ohne das vorige Skript mit einem anderen u_0–Wert neu ausführen zu müssen. Außerdem wird für die Zeichnungen die isometrische Skalierung verwendet werden. Nach der Anzeige des Vektorfeldes wird jede Anfangbedingung mit einem Klick auf die linke Maustaste[2] festgelegt, nachdem der Mauszeiger auf die gewünschte Anfangsbedingung positioniert wurde. Diese graphische Möglichkeit wird von dem Grundbefehl `xclick` ermöglicht, dessen vereinfachte Syntax die folgende ist:

```
[c_i,c_x,c_y]=xclick();
```

Scilab erwartet also einen „Graphik–Event“ vom Typ „Mausklick“, und wenn dieser Event stattfindet, erhält man die Position des Mauszeigers (in der aktuellen Skalierung) mit `c_x` und `c_y` sowie die Nummer der Maustaste:

[1] Um es kurz zu machen: man sagt, dass eine Differentialgleichung steif ist, wenn sich diese mit den (mehr oder weniger) expliziten Methoden schwer integrieren lässt

[2] wie in einem der Artikel über Scilab im „Linux Magazine“ vorgeschlagen

Wert für `c_i`	Maustaste
0	links
1	Mitte
2	rechts

Im Skript führt das Klicken der rechten Maustaste zum Verlassen der Event–Schleife. Schließlich nimmt man noch einige Verschönerugen derart vor, dass für jede Trajektorie die Farbe gewechselt wird (die Tabelle `Farbe` erlaubt es, geeignete Farben aus der Standardfarbpalette auszuwählen). Um eine isometrische Skalierung zu erhalten, ruft man `fchamp` mit einer Option wie bei `plot2d` (man muss `strf=`*Wert_strf* benutzen, da die Parameter *frameflag* und *axesflag* nicht unterstützt werden[3]. Letzte Verschönerung: man zeichnet einen kleinen Kreis, um die Anfangsbedingung zu markieren. Um das gesamte Graphikfenster auszunutzen, benutzt man einen rechteckigen Phasenraum. Letzte Bemerkung: wenn das Vektorfeld erscheint, können Sie das Graphikfenster maximieren! Durch mehrmaliges Klicken erhält man die Abbildung 5.1. Alle Trajektorien konvergieren gegen eine periodische Bahn, was für diese Gleichung ein erwartetes theoretisches Verhalten darstellt.

```
// 1.) Zeichnung des Vektorfeldes zur Van-der-Pol-Gleichung
n = 30;
delta_x = 6
delta_y = 4
x = linspace(-delta_x,delta_x,n);
y = linspace(-delta_y,delta_y,n);
xbasc()
fchamp(VanDerPol,0,x,y, strf="041")
xselect()

// 2.) Lösung der Differentialgleichung
m = 500 ; T = 30 ;
t = linspace(0,T,m);
Farben = [21 2 3 4 5 6 19 28 32 9 13 22 18 21 12 30 27]  // 17 Farben
num = -1
while %t
  [c_i,c_x,c_y]=xclick();
  if c_i == 0 then
    plot2d(c_x, c_y, style=-9, strf="000")  // ein o markiert die Anfangsbedingung
    u0 = [c_x;c_y];
    u = ode(u0, 0, t, VanDerPol);
    num = modulo(num+1,length(Farben));
    plot2d(u(1,:)',u(2,:)', style=Farben(num+1),  strf="000")
  elseif c_i == 2 then
    break
  end
end
```

5.1.3 Weiteres zu `ode`

Im zweiten Beispiel wird der Grundbefehl `ode` mit einer rechten Seite verwendet, die einen zusätzlichen Parameter zulässt, und wir werden selbst die Toleranzen für die Zeitschrittsteuerung des Lösers festlegen. Hier ist unsere neue Differentialgleichung (*die Brusselator–Gleichung*):

$$\begin{cases} \frac{du_1}{dt} = 2 - (6+\epsilon)u_1 + u_1^2 x_2 \\ \frac{du_2}{dt} = (5+\epsilon)u_1 - u_1^2 u_2 \end{cases}$$

die einen einzigen kritischen Punkt $P_{stat} = (2, (5+\epsilon)/2)$ besitzt. Wenn der Parameter ϵ von einem strikt negativen in einen positiven Wert übergeht, wechselt dieser stationäre Punkt sein Verhalten (aus einem stabilen Zustand wird er instabil, für $\epsilon = 0$ tritt ein Phänomen auf, das *Hopf*verzweigung heißt). Man interessiert sich für die Trajektorien zu den Anfangsbedingungen in der Nähe dieses Punktes. Hier die Funktion, die die rechte Seite berechnet:

[3] außer bei neueren Versionen von Scilab

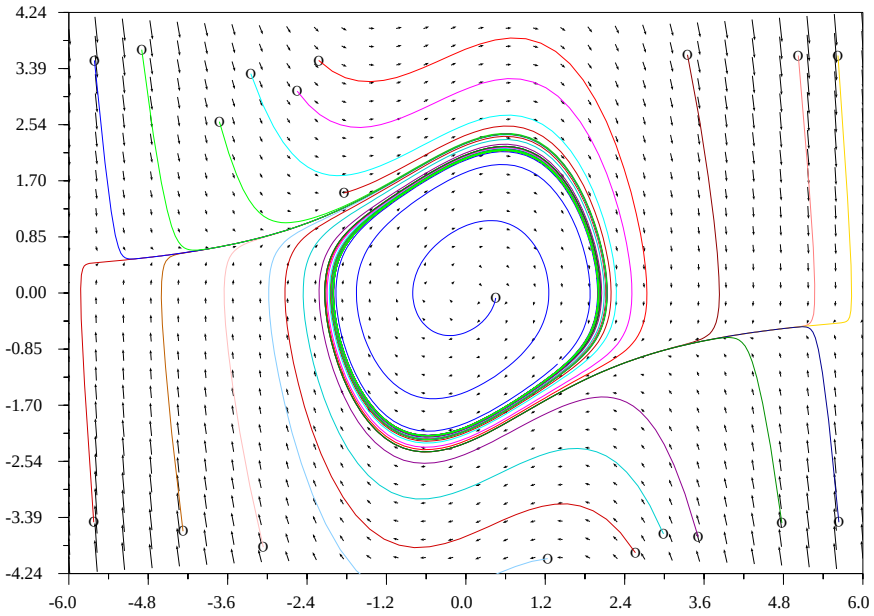

Abbildung 5.1: Einige Trajektorien im Phasenraum der Van–der–Pol–Gleichung

```
function f = Brusselator(t,u,eps)
  //
  f(1) = 2 - (6+eps)*u(1) + u(1)^2*u(2)
  f(2) = (5+eps)*u(1) - u(1)^2*u(2)
endfunction
```

Um den zusätzlichen Parameter übergeben zu können, ersetzt man beim Aufruf von `ode` den Namen der Funktion (hier `Brusselator`) durch eine Liste, die aus dem Namen der Funktion und dem oder den zusätzlichen Parametern besteht:

```
x = ode(x0,t0,t,list(MeineRechteSeite, par1, par2, ...))
```

In diesem Fall:

```
x = ode(x0,t0,t,list(Brusselator, eps))
```

und man verfährt genauso, um das Feld mit `fchamp` zu zeichnen.

Um die Toleranzen für den lokalen Fehler des Lösers festzulegen, fügt man die Parameter `rtol` und `atol` vor dem Namen der rechte–Seite–Funktion (oder der Liste, die durch diese und die zusätzlichen Parameter der Funktion gebildet wird) hinzu. In jedem Zeitschritt, $t_{k-1} \to t_k = t_{k-1} + \Delta t_k$, berechnet der Löser eine Schätzung des lokalen Fehlers e (d.h. des Fehlers in diesem Zeitschritt mit $v(t_{k-1}) = U(t_{k-1})$ als Anfangsbedingung):

$$e(t_k) \simeq U(t_k) - \left(\int_{t_{k-1}}^{t_k} f(t, v(t))dt + U(t_{k-1}) \right)$$

(der zweite Term ist die exakte Lösung, die von der nummerischen Lösung $U(t_{k-1})$ ausgeht, die im vorigen Schritt erhalten wurde) und vergleicht diesen Fehler mit der Toleranz, die durch die beiden Parameter `rtol` und `atol` gebildet wird; sind dies zwei Vektoren der Länge n , so gilt

$$tol_i = rtol_i * |U_i(t_k)| + atol_i,\ 1 \le i \le n$$

und im Falle zweier Skalare

$$tol_i = rtol * |U_i(t_k)| + atol,\ 1 \le i \le n$$

Wenn $|e_i(t_k)| \leq tol_i$ für jede Komponente gilt, wird der Schritt akzeptiert, und der Löser berechnet den neuen Zeitschritt derart, dass das Kriterium für den zukünftigen Fehler eine Chance hat, erfüllt zu werden. Im gegenteiligen Fall wird ab t_{k-1} mit einem etwas kleineren Zeitschritt neu gelöst (, sodass der nächste Test des lokalen Fehlers mit größerer Wahrscheinlichkeit erfüllt wird). Da der Löser Merhschrittverfahren verwendet, justiert er neben dem Zeitschritt auch die Ordnung des Verfahrens, um eine möglichst hohe Effizienz zu erreichen. Standardmäßig werden die Werte $rtol = 10^{-5}$ und $atol = 10^{-7}$ benutzt (außer wenn via `type` ein Runge–Kutta–Verfahren ausgewählt hat). Wichtiger Hinweis: Der Löser kann sehr wohl bei der Integration scheitern...

Hier ein mögliches Skript; die einzige zusätzliche Verschönerung besteht darin, den kritischen Punkt mit einem kleinen schwarzen Quadrat zu markieren, das man mit dem Graphik–Grundbefehl `xfrect` erhält:

```
// die Brusselator-Gleichung
eps = -4
P_stat = [2 ; (5+eps)/2];
// Grenzen für die Zeichnung des Vektorfeldes
delta_x = 6; delta_y = 4;
x_min = P_stat(1) - delta_x; x_max = P_stat(1) + delta_x;
y_min = P_stat(2) - delta_y; y_max = P_stat(2) + delta_y;
n = 20;
x = linspace(x_min, x_max, n);
y = linspace(y_min, y_max, n);
// 1.) Zeichnung des Vektorfeldes
xbasc()
fchamp(list(Brusselator,eps),0,x,y, strf="041")
xfrect(P_stat(1)-0.08,P_stat(2)+0.08,0.16,0.16) // Markierung des kritischen Punktes
xselect()

// 2.) Lösung der Differentialgleichung
m = 500 ; T = 5 ;
rtol = 1.d-09; atol = 1.d-10; // Toleranzen für den Löser
t = linspace(0,T,m);
Farben = [21 2 3 4 5 6 19 28 32 9 13 22 18 21 12 30 27]
num = -1
while %t
  [c_i,c_x,c_y]=xclick();
  if c_i == 0 then
    plot2d(c_x, c_y, style=-9, strf="000")  // ein o markiert die Anfangsbedingung
    u0 = [c_x;c_y];
    u = ode(u0, 0, t, rtol, atol, list(Brusselator,eps));
    num = modulo(num+1,length(Farben));
    plot2d(u(1,:)',u(2,:)', style=Farben(num+1), strf="000")
  elseif c_i == 2 then
    break
  end
end
```

5.2 Erzeugen von Zufallszahlen

5.2.1 Die Funktion `rand`

Bis zu diesem Zeitpunkt diente sie im Wesentlichen dazu, Matrizen und Vektoren zu füllen... Diese Funktion verwendet den folgenden linearen Kongruenz–Generator[4]:

$$X_{n+1} = f(X_n) = (aX_n + c) \bmod m,\ n \geq 0, \quad \text{mit} \begin{cases} m = 2^{31} \\ a = 843314861 \\ c = 453816693 \end{cases}$$

[4] laut dem, was aus dem Sourcecode hervorgeht

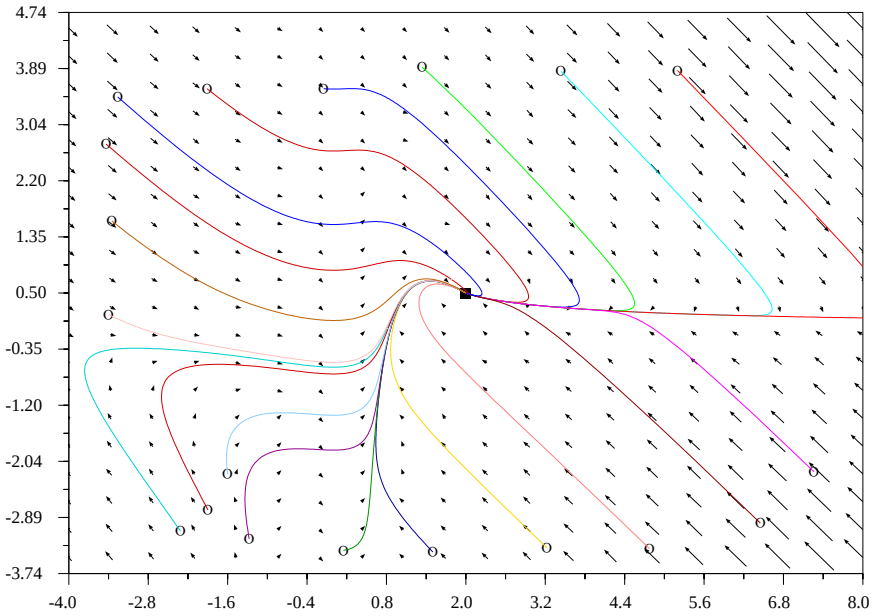

Abbildung 5.2: Einige Trajektorien im Phasenraum der Brusselator–Gleichung ($\epsilon = -4$)

Ihre Periode ist m (dies bedeutet, dass f eine zyklische Permutation auf $[0, m-1]$ ist). Zu beachten ist, dass alle Zufallszahlengeneratoren auf Computern vollkommen deterministische Folgen erzeugen, welche (für gute Generatoren) entsprechend einer bestimmten Anzahl von statistischen Tests (nur) zufällig aussehen. Um reelle Zahlen im Intervall $[0, 1[$ zu erhalten, dividiert man die erhaltenen ganzen Zahlen durch m (und erhält einen Generator für reelle Zahlen, die einer Gleichverteilung auf $[0, 1[$ zu folgen scheinen). Der Anfangsterm einer Folge wird oft Keim (seed) genannt, und dieser ist standardmäßig $X_0 = 0$. Daher liefert der erste Aufruf von `rand` (der erste Koeffizient, falls man eine Matrix oder einen Vektor erhält) immer:

$$u_1 = 453816693/2^{31} \approx 0.2113249$$

Es ist jedoch möglich, jederzeit den Keim (seed) mit der folgenden Anweisung zu ändern:

```
rand("seed",Keim)
```

wobei `Keim` eine ganze Zahl im (abgeschlossenen) Intervall $[0, m-1]$ ist. Oft benötigt man einen mehr oder weniger zufälligen Keim (seed) (um nicht jedesmal dieselben Zahlen zu bekommen), und eine Möglichkeit besteht darin, das Datum und die Uhrzeit zu verwenden und damit den Keim zu erzeugen. Scilab besitzt eine Funktion `getdate`, die einen Vektor mit neun ganzen Zahlen liefert (bezüglich der Details siehe `Help`), u.a. folgende:

- die zweite Zahl gibt den Monat (1-12) an,
- die sechste den Tag des Monats (1-31),
- die siebte die Stunde (0-23),
- die achte die Minuten (0-59),
- und die neunte die Sekunden (0-59).

Um einen Keim zu erhalten, kann man z.B. diese Zahlen addieren:

```
v = getdate()
rand("seed", sum(v([2 6 7 8 9])))
```

Den aktuell benutzten Keim kann man mit

```
Keim = rand("seed")
```

erfragen. Ausgehend von der Gleichverteilung auf $[0, 1[$ kann man auch andere Verteilungen erhalten; `rand` stellt ein Interface bereit, das erlaubt, eine Normalverteilung (mit dem Mittelwert 0 und der Varianz 1) zu erhalten. Um von der einen in die andere zu wechseln, verfährt man wie folgt:

```
rand("normal")  // für die Normalverteilung
rand("uniform") // um zur Gleichverteilung zurückzukehren
```

Voreingestellt ist eine Gleichverteilung, aber es ist klug, sich in jeder Simulation zu versichern, dass `rand` das liefert, was man erwartet, indem man eine der beiden Anweisungen benutzt. Man kann übrigens die aktuelle Verteilung mit folgender Anweisung bestimmen:

```
verteilung=rand("info")  // verteilung ist eine der beiden Zeichenketten "uniform"
                         // oder "normal"
```

Es sei daran erinnert, dass `rand` auf unterschiedliche Art und Weise verwendet werden kann:

1. `A = rand(n,m)` füllt die Matrix `A` der Größe (n,m) mit Zufallszahlen;
2. ist `B` eine bereits definierte $n \times m$–Matrix, dann erhält man mit `A = rand(B)` dasselbe (man braucht also vorher nicht die Dimensionen von `B` zu bestimmen);
3. schließlich liefert `u = rand()` eine einzelne Zufallszahl.

In den ersten beiden Fällen kann man ein zusätzliches Argument hinzufügen, um die Verteilung festzulegen: `A = rand(n,m,verteilung)`, `A = rand(B,verteilung)`, wobei `verteilung` eine der beiden Zeichenketten `"normal"` oder `"uniform"` ist.

Einige kleine Anwendungen mit `rand`

Ausgehend von der Gleichverteilung ist es einfach, eine Zahlenmatrix der Größe (n, m) zu erhalten, gemäß

1. einer Gleichverteilung auf $[a, b[$:

   ```
   X = a + (b-a)*rand(n,m)
   ```

2. einer Gleichverteilung auf den ganzen Zahlen im Intervall $[n_1, n_2]$:

   ```
   X = floor(n1 + (n2+1-n1)*rand(n,m))
   ```

   (man zieht reelle Zahlen aus dem reellen Intervall $[n_1, n_2 + 1[$ gemäß einer Gleichverteilung und nimmt dann deren ganzzahligen Anteil).

Nach Bernouilli kann man vorhersagen, wie häufig bei N Versuchen ein Ereignis mit der Erfolgswahrscheinlichkeit p eintreten wird.

```
erfolg = rand() < p
```

Damit haben wir eine einfache[5] Methode zur Simulation der Binomialverteilung $B(N, p)$:

```
X = sum(bool2s(rand(1,N) < p))
```

(`bool2s` wandelt die Erfolge in 1 um, und es bleibt nur noch, diese mit `sum` zu addieren). Da Iterationen in Scilab langsam verlaufen, kann man auf direktem Wege einen (Spalten–)Vektor erhalten, der m Realisierungen dieser Verteilung enthält:

```
X = sum(bool2s(rand(m,N) < p),"c")
```

[5] aber für große N ineffiziente

Es ist vorteilhafter, die Funktion `grand` zu benutzen, die eine leistungsfähigere Methode verwendet. Wenn Sie andererseits diesen kleinen Trick[6] benutzen, ist es einfach, diese als Scilabfunktionen zu codieren. Hier eine kleine Funktion zum Simulieren der geometrischen Verteilung (Anzahl nötiger Bernouilli–Tests, um einen Erfolg zu erzielen)[7]:

```
function X = G(p)
  // geometrische Verteilung
  X = 1
  while rand() > p   // Misserfolg
    X = X+1
  end
endfunction
```

Schließlich erhält man ausgehend von der Normalverteilung $\mathcal{N}(0,1)$ die Normalverteilung $\mathcal{N}(\mu,\sigma^2)$ (Mittelwert μ und Standardabweichung σ) durch:

```
rand("normal")
X = mu + sigma*rand(n,m) // um eine n x m - Matrix solcher Zahlen zu erhalten
// oder auch in einer einzigen Anweisung: X = mu + sigma*rand(n,m,"normal")
```

5.2.2 Die Funktion `grand`

Für komplexe Simulationen, die viele Zufallszahlen verwenden, ist die Standardfunktion `rand` mit ihrer Periode $2^{31}(\simeq 2.147\ 10^9)$ vielleicht etwas zu einfach. Man sollte daher `grand`, das auch die Simulation aller klassischen Verteilungen erlaubt, vorziehen. Man benutzt `grand` wird fast in derselben Weise wie `rand`, d.h. man kann eine der folgenden beiden Syntaxen verwenden (für den zweiten Fall muss offensichtlich die Matrix `A` im Moment des Aufrufs definiert sein):

```
grand(n,m,verteilung, [p1, p2, ...])
grand(A,verteilung, [p1, p2, ...])
```

mit einer Zeichenkette `verteilung`, die die Verteilung festlegt, und optionalen Parametern. Einige Beispiele (m Stichproben in Form eines Spaltenvektors):

1. eine Gleichverteilung über den ganzen Zahlen eines großen Intervalles $[1, m[$:

   ```
   X = grand(m,1,"lgi")
   ```

   wobei m vom Basisgenerator abhängt (Voreinstellung: $m = 2^{32}$ (siehe weiter unten));

2. eine Gleichverteilung über den ganzen Zahlen im Intervall $[n1, n2]$:

   ```
   X = grand(m,1,"uin",k1,k2)
   ```

   (es muss $k2 - k1 \leq 2147483561$ gelten, andernfalls wird eine Fehlermeldung erzeugt);

3. für die Gleichverteilung auf $[0, 1[$:

   ```
   X = grand(m,1,"def")
   ```

4. für die Gleichverteilung auf $[a, b[$:

   ```
   X = grand(m,1,"unf",a,b)
   ```

5. für die Binomialverteilung $B(N, p)$:

[6] Im Allgemeinen benutzt man eher die Funktion `grand`, die die meisten klassischen Verteilungen liefert.
[7] Die Funktion ist für kleines p ineffizient.

```
X = grand(m,1,"bin",N,p)
```

6. für die geometrische Verteilung $G(p)$:

```
X = grand(n,1,"geom",p)
```

7. für die Poissonverteilung mit dem Mittelwert μ:

```
X = grand(m,1,"poi",mu)
```

8. für die Exponentialverteilung mit dem Mittelwert λ:

```
X = grand(n,1,"exp",lambda)
```

9. für die Normalverteilung mit dem Mittelwert μ und der Standardabweichung σ:

```
X = grand(m,1,"nor",mu,sigma)
```

Weitere Verteilungen findet man in den Hilfeseiten.

Seit Version 2.7 besitzt `grand` verschiedene Basisgeneratoren (, die gleichverteilte ganze Zahlen liefern (*lgi*–Verteilung)). Standardmäßig benutzt `grand` den *Mersenne Twister*, der eine riesengroße Periode der Länge 2^{19937} hat; es gibt insgesamt 6 Generatoren[8]. Um mit diesen Generatoren zu arbeiten, benutzt man die Anweisungen:

`Name_gen = grand("getgen")`	gibt den Namen des aktuellen Generators an
`grand("setgen",Name_gen)`	wählt `Name_gen` als aktuellen Generator
`Zustand = grand("getsd")`	gibt den internen Zustand des aktuellen Generators zurück
`grand("setsd",e1,e2,..)`	setzt den internen Zustand des aktuellen Generators

Die Dimension des internen Zustand hängt vom Typ des Generators ab: eine ganze Zahl für *urand* und bis zu 624 ganze Zahlen plus einem Index für den *Mersenne Twister*[9]. Wenn Sie (bei einer Simulation) einen bestimmten Ausgangszustand exakt wiederherstellen wollen, müssen Sie den Anfangszustand (vor der Simulation) des benutzten Generators kennen und auf die eine oder andere Weise sichern; Beispiel:

```
grand("setgen","kiss")   // kiss wird der aktuelle Generator
e = [1 2 3 4];           // einzustellender Anfangszustand
                         // (dieser besteht aus 4 ganzen Zahlen)
grand("setsd",e(1),e(2),e(3),e(4));    // so wird er gesetzt
grand("getsd")           // sollte den Vektor e liefern
X = grand(10,1,"def");   // 10 Zahlen
s1 = sum(X);
X = grand(10,1,"def");   // weitere 10 Zahlen
s2 = sum(X);
s1 == s2                 // i.A. unterscheidet sich s1 von s2

grand("setsd",e(1),e(2),e(3),e(4));   // Rückkehr zum Ausgangszustand
X = grand(10,1,"def");   // wieder 10 Zahlen
s3 = sum(X);
s1 == s3                 // s1 und s3 sollten jetzt übereinstimmen
```

[8] `"mt"` ,`"kiss"`, `"clcg4"`, `"clcg2"`, `"fsultra"`, und `"urand"`; der letzte ist der Generator von `rand`.

[9] Dennoch existiert eine Initialisierungsfunktion mit nur einem ganzzahligen Parameter.

5.3 Klassische Verteilungsfunktionen und ihre Inversen

Diese Funktionen sind oft bei statistischen Tests (χ^2_r, ...) nützlich, weil sie Folgendes zu berechnen gestatten:

1. die Verteilungsfunktion in einem oder mehreren Punkten;
2. ihre Inverse in einem oder mehreren Punkten;
3. einen der Parameter der Verteilung, wenn die anderen und ein Paar $(x, F(x))$ gegeben sind.

Unter `Help` werden Sie sie in der Rubrik „Cumulative Distribution Functions“ finden, und alle diese Funktionen fangen mit den Buchstaben `cdf` an. Nehmen wir z.B. die Normalverteilung $\mathcal{N}(\mu, \sigma^2)$; die Funktion, die uns hier interessiert, heißt `cdfnor` und hat die folgende Syntax:

1. `[P,Q]=cdfnor("PQ",X,mu,sigma)`, um $P = F_{\mu,\sigma}(X)$ und $Q = 1 - P$ zu erhalten; `X`, `mu` und `sigma` können Vektoren (gleicher Größe) sein, und man erhält dann für `P` und `Q` Vektoren mit $P_i = F_{\mu_i,\sigma_i}(X_i)$;
2. `X=cdfnor("X",mu,sigma,P,Q)`, um $X = F^{-1}_{\mu,\sigma}(P)$ zu erhalten (genau wie oben können die Argumente Vektoren gleicher Größe sein; man erhält dann $X_i = F^{-1}_{\mu_i,\sigma_i}(P_i)$);
3. `mu=cdfnor("Mean",sigma,P,Q,X)`, um den Mittelwert zu erhalten;
4. und schließlich `sigma=cdfnor("Std",P,Q,X,mu)`, um die Standardabweichung zu erhalten.

Diese beiden letzten Formen funktionieren auch, wenn die Argumente Vektoren gleicher Größe sind.
Bemerkungen:

- Die Funktionen arbeiten intern mal mit p und mal mit $q = 1 - p$, um auszunutzen, dass die Genauigkeit in Bereichen, wo p nahe 0 bzw. 1 ist, unterschiedlich ist.
- Die Zeichenkette für die inverse Verteilungsfunktion ist nicht immer `"X"` — sehen Sie in den entsprechenden Hilfeseiten nach!

5.4 Einfache stochastische Simulationen

5.4.1 Einführung und Notation

Häufig besteht eine Simulation in erster Linie darin, einen Vektor

$$x^m = (x_1, ..., x_m)$$

zu bestimmen, dessen Komponenten als Realisation gleichverteilter Zufallsvariablen $X_1, X_2, ..., X_m$ angesehen werden können. (Wir bezeichnen im Folgenden mit X eine allgemeine Zufallsvariable der gleichen Verteilung.) In der Praxis erhält man den Vektor x^m direkt oder indirekt aus den Funktionen `rand` oder `grand`[10].

Ausgehend von einer Stichprobe x^m möchte man einige charakteristische Größen der zugrundeliegenden Verteilung wie den Erwartungswert, die Varianz, die Verteilungsfunktion (oder Dichte) approximieren. Oder, wenn eine parametrische Verteilung (vermutlich) vorliegt, zur Bestimmung seiner Parameter; oder, wenn die Parameter bekannt sind, um mit statischtischen Test zu verifizieren, dass die vorliegende Stichprobe die Realisierung einer Zufallsvariable gemäß der vorliegenden Verteilung ist, etc... In der Mehrzahl der uns interessierenden Fälle kennen wir die exakten theoretischen Resultate und die Simulation dient der Illustration eines Resultates oder (mathematischen) Satzes, einer Methode oder eines Algorithmus.

[10] In der Statistik erhält man in den meisten Fällen Stichproben durch Messungen in der Physik (Temperatur, Druck, ...), biometrische Daten (Größe, Gewicht), Umfragen, etc ... Die Daten werden in einer Datei oder Datenbank gespeichert. Einige Softwarepakete (wie z.B. R) erlauben das Hantieren mit solchen Daten, aber dies geht z.Z. nicht mit Scilab; dennoch gibt es zahlreiche solcher Simultionen, z.B., um das Verhalten von bestimmten Systemen in Abhängigkeit von zufälligen (Störungen von) Eingangsdaten zu studieren, oder um rein deterministische Probleme zu behandeln, für die nummerische Methoden viel zu aufwändig oder gar undurchführbar sind.

5.4.2 Konfidenzintervalle

Hat man — ausgehend von der vorliegenden Stichprobe — eine Schätzung des Erwartungswertes (in Scilab mittels `x_bar_m = mean(xm)`), so möchte man ein Intervall I_c (häufig mit Mittelpunkt $\bar{x}_m$) kennen, für das

$$E[X] \in I_c \text{ mit der Wahrscheinlichkeit } 1-\alpha$$

gilt, wobei häufig $\alpha = 0.05$ oder $\alpha = 0.01$ ist (95%– oder 99%–Konfidenzintervalle). Das fundamentale Mittel zur Herleitung solcher Intervalle ist der zentrale Grenzwertsatz. Mit

$$\bar{X}_m = \frac{1}{m}\sum_{i=1}^{m} X_i$$

erhält man die Zufallsvariable des Mittelwertes, (von dem $\bar{x}_m$ eine Realisierung ist,) die nach dem Gesetz der großen Zahlen gegen $E[X]$ "konvergiert". Der zentrale Grenzwertsatz sagt (unter bestimmten Bedingungen) aus, dass

$$\lim_{m\to+\infty} P(a < \frac{\sqrt{m}(\bar{X}_m - E[X])}{\sigma} \leq b) = \frac{1}{\sqrt{2\pi}}\int_a^b e^{-t^2/2}dt$$

(, wobei man $Var[X] = \sigma^2$ gesetzt hat.) Für m genügend groß kann man diese Wahrscheinlichkeit durch den Grenzwert[11] approximieren:

$$P(a < \frac{\sqrt{m}(\bar{X}_m - E[X])}{\sigma} \leq b) \approx \frac{1}{\sqrt{2\pi}}\int_a^b e^{-t^2/2}dt$$

Sucht man "symmetrische" Konfidenzintervalle, so hat man:

$$\frac{1}{\sqrt{2\pi}}\int_{-a}^{a} e^{-t^2/2}dt = F_{N(0,1)}(a) - F_{N(0,1)}(-a) = 2F_{N(0,1)}(a) - 1 = 1-\alpha$$

und daher:

$$a_\alpha = F^{-1}_{N(0,1)}(1-\frac{\alpha}{2}) \quad \text{oder auch} \quad -F^{-1}_{N(0,1)}(\frac{\alpha}{2})$$

was man in Scilab folgendermaßen schreibt:

```
a_alpha = cdfnor("X", 0, 1, 1-alpha/2, alpha/2)
```

Damit erhält man schließlich[12] mit einer Wahrscheinlichkeit von $1-\alpha$ (, wenn die Approximation durch den Grenzwert korrekt ist):

$$E[X] \in [\bar{x}_m - \frac{a_\alpha\sigma}{\sqrt{m}}, \bar{x}_m + \frac{a_\alpha\sigma}{\sqrt{m}}]$$

Das Problem liegt darin, dass man i.A. die Varianz nicht kennt. Man benutzt dann eine obere Schranke oder eine Schätzung gemäß:

$$S_m = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(X_i - \bar{X}_m)^2}$$

Indem man die Varianz σ durch s_m ersetzt (,wobei s_m eine Realisierung von S_m ist), erhält man rein empirisch ein Konfidenzintervall.

Es gibt jedoch einige Sonderfälle:

1. Wenn die X_i einer Normalverteilung $N(\mu, \sigma^2)$ genügen, so gilt

$$\sqrt{m}\frac{\bar{X}_m - \mu}{S_m} \sim t(m-1)$$

[11] Für gewisse Verteilungen hat man dafür Kriterien, z.B., wenn $X \sim Ber(p)$, so ist der Grenzwert genügend genau, sobald $\min(mp, m(1-p)) > 10$ gilt.

[12] Für ein 95%–Konfidenzintervall ist $a_\alpha \simeq 1.96$, was man oft durch 2 approximiert.

wobei $t(k)$ die Student–Verteilung mit k Freiheitsgraden ist. In diesem Falle existieren die eben angewandten Approximation nicht mehr (Approximation durch den Grenzwert, Approximation der Variation) und man erhält

$$\mu \in [\bar{x}_m - \frac{a_\alpha s_m}{\sqrt{m}}, \bar{x}_m + \frac{a_\alpha s_m}{\sqrt{m}}] \text{ mit der Wahrscheinlichkeit } 1-\alpha$$

wobei s_m die Stichprobenvarianz (`sm = st_deviation(xm)` in Scilab) ist, und a_α sich mit Hilfe der Student–Verteilung (anstelle der $N(0,1)$–Verteilung) berechnet:

$$a_\alpha = F^{-1}_{t(m-1)}(1 - \frac{\alpha}{2})$$

was man in Scilab[13] mit

```
a_alpha = cdft("T", m-1, 1-alpha/2, alpha/2)
```

erhält.

2. Wenn die Varianz eine Funktion des Erwartungswertes ist (Bernouilli–, Poisson–, Exponential–Verteilung,...), so kommt man ohne eine Approximation der Varianz aus, und man erhält das Konfidenzintervall durch Lösung einer Ungleichung.

5.4.3 Zeichnen einer empirischen Verteilungsfunktion

Die Verteilungsfunktion von X ist die Funktion

$$F(x) = \text{Wahrscheinlichkeit, dass } X \leq x$$

und die empirische Verteilungsfunktion, die durch die Stichprobe x^m definiert ist, ist gegeben durch

$$F_{x^m}(x) = card\{x_i <= x\}/m$$

Es ist eine Treppenfunktion, die sich einfach berechnen lässt, wenn man den Vektor x^m aufsteigend sortiert (man hat also $F_{x^m}(x) = i/m$ für $x_i \leq x < x_{i+1}$). Der Standardsortieralgorithmus von Scilab ist die Funktion `sort`, welche in absteigender Reihenfolge sortiert[14]. Um den Vektor x^m in aufsteigender Reihenfolge zu sortieren, benutzt man also:

```
xm = - sort(-xm)
```

Die Funktion `plot2d2` erlaubt nun, diese Funktion in einfacher Weise zu zeichnen, z.B. der folgende Programmcode:

```
function empirische_Verteilung(xm)
   //  zeichnet die (empirische) Verteilungsfunktion, die zu der
   //  Stichprobe xm gehört
   m = length(xm)
   xm = - sort(-xm(:))
   ym = (1:m)'/m
   plot2d2(xm, ym, leg="empirische Verteilungsfunktion")
endfunction
```

der einen kleinen Trick enthält: `xm(:)` ermöglicht es, dass die Funktion auch mit einem Zeilenvektor als Eingabe funktioniert.

Im nächsten Beispiel verwenden wir die Normalverteilung $\mathcal{N}(0,1)$ — siehe Abbildung 5.3.

[13] Siehe die entsprechende Hilfeseite, da die Funktionen `cdf...` keine besonders regelmäßige Syntax haben; so ist z.B. `"X"` nicht immer der Parameter für die inverse Verteilungsfunktion, hier ist es `"T"`.

[14] siehe auch die Funktion `gsort`, die mehr kann

```
m = 100;
xm = grand(m,1,"nor",0,1);
xbasc()
empirische_Verteilung(xm);  // Bild der empirischen Verteilungsfkt.
// Daten zum Zeichnen der exakten Verteilungsfkt.
x = linspace(-4,4,100)';
y = cdfnor("PQ", x, zeros(x), ones(x));
plot2d(x, y, style=2)      // man fügt dem ersten Bild diese Kurve hinzu
xtitle("Exakte und empirische Verteilungsfunktion")
```

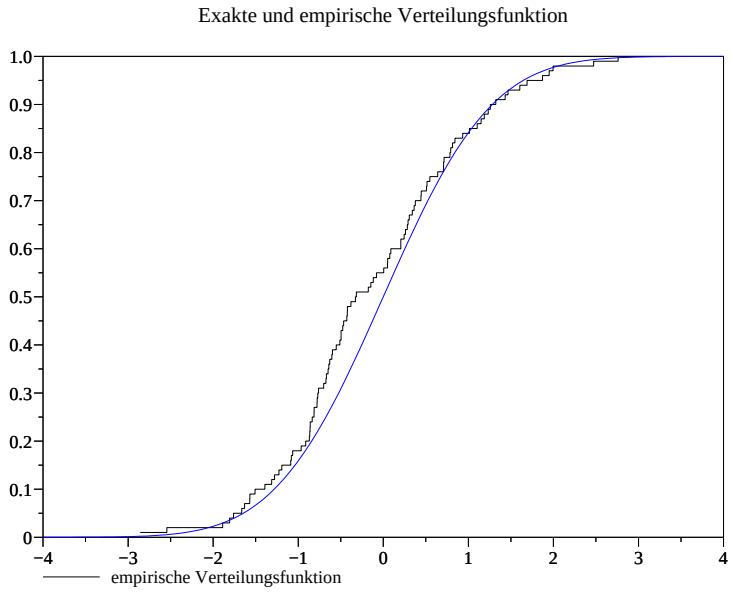

Abbildung 5.3: Exakte und empirische Normalverteilungsfunktion

5.4.4 Ein χ^2–Test

Sei nun $x^m = (x_1, ..., x_m)$ unsere zu analysierende Stichprobe. Wir untersuchen die Hypothese $\mathcal{H}$, dass die Zufallsvariablen $(X_1, ..., X_m)$ die zugrundeliegende Verteilung $\mathcal{L}$ haben. Wir können die elementaren statistischen Größen (wie Mittelwert und Varianz) berechnen und mit den theoretischen Werten, die zu $\mathcal{L}$ gehören, vergleichen. Bei genügend guter Übereinstimmung möchte man dann ein statistisches Testverfahren benutzen. Der χ^2–Test eignet sich für eine diskrete Verteilung mit endlich vielen Werten. Nimmt man z.B. an, dass die Verteilung $\mathcal{L}$ durch $\{(v_i, p_i), 1 \leq i \leq n\}$ gegeben ist, so besteht der Test darin, zunächst die Größe

$$y = \frac{\sum_{i=1}^{n}(o_i - mp_i)^2}{mp_i}$$

zu berechnen, wobei o_i angibt, wieviele x_j den Wert v_i haben. Sodann vergleicht man dieses y mit einem Fraktil (Schwellenwert) y_α. Der Test stützt[15] die Hypothese, wenn $y \leq y_\alpha$ gilt.

Ersetzt man in der Formel für y die Stichprobe $x_1, \ldots, x_m$ durch die Zufallsvariablen $X_1, \ldots, X_m$[16], so erhält man eine Zufallsvariable Y, die (für genügend großes m) approximativ eine χ^2–Verteilung mit $n-1$ Freiheitsgraden hat. Das Fraktil y_α erhält man daher gemäß

$$y_\alpha = F^{-1}_{\chi^2_{n-1}}(1-\alpha)$$

wobei man meist $\alpha = 0.05$ oder sogar $\alpha = 0.01$ wählt. In Scilab erhält man das Fraktil mittels

[15] Rein intuitiv erwartet man im Falle einer korrekten Hypothese, dass o_i in etwa mp_i ist. Daher wird man die Hypothese verwerfen, wenn der Wert von y zu groß ist ($y > y_\alpha$).

[16] mit Hilfe der vektoriellen Zufallsvariable $O = (O_1, \ldots, O_n)$, die einer Multinomialverteilung genügt

```
y_alpha = cdfchi("X", n-1, 1-alpha, alpha)
```

Die Häufigkeiten o_i bestimmt man mit Scilab folgendermaßen:

```
occ = zeros(n,1);
for i=1:n
   occ(i) = sum(bool2s(xm == v(i))); // oder length(find(xm==v(i)))
end
if sum(occ) ~= m then, error("Anzahlfehler"), end
```

Damit erhält man die Größe y (mit vektorieller Schreibweise[17]):

```
y = sum( (occ - m*p).^2 ./ (m*p) )
```

wobei vorausgesetzt wird, dass `p` (der Vektor der diskreten Wahrscheinlichkeiten von $\mathcal{L}$) die gleiche Gestalt wie `occ` hat (hier ein Spaltenvekor — siehe meine Definition von `occ`).
Bemerkungen:

- Die Approximation durch die χ^2–Verteilung ist nur für genügend große m zulässig; eine häufig geforderte Bedingung ist $m\,p_{min} > 5$ ($p_{min} = \min_i p_i$). Man sollte diese testen und den Anwender darauf hinweisen, wenn sie nicht erfüllt ist.
- Es ist einfach, diese Berechnungen in einer Funktion zusammenzufassen.
- Bei einer stetigen Verteilung lässt sich der Test anwenden, wenn man die Werte zu Gruppen zusammenfasst, die je in einem Intervall liegen. Für die Gleichverteilung $U_{[0,1]}$ auf dem Intervall $[0, 1]$ teilt man dieses in n äquidistante Teile. In ähnlicher Weise kann man bei einer diskreten Verteilung mit unendlich vielen Werten die Schwänze der Verteilung zu Gruppen zusammenfassen. Auch für diskrete endliche Verteilungen, für die die Bedingungen des Tests nicht erfüllt sind, kann man so vorgehen.
- Wenn man eine parametrische Verteilung hat, deren Parameter erst noch durch die Daten geschätzt werden müssen, so muss man die Zahl der Freiheitsgrade der χ^2–Verteilung um die Zahl dieser Parameter erniedrigen. Wenn z.B. eine $B(n-1, p)$–Verteilung vorliegt und man p durch $p = \bar{x}_m/(n-1)$ schätzt, so erhält man für das Fraktil $y_\alpha = F^{-1}_{\chi^2_{n-2}}(1-\alpha)$ anstelle von $y_\alpha = F^{-1}_{\chi^2_{n-1}}(1-\alpha)$.

5.4.5 Kolmogorov–Smirnov–Test

Dieser Test ist natürlicher als der χ^2–Test, wenn die erwartete Verteilung eine stetige Verteilungsfunktion besitzt. Es sei X eine reelle Zufallsvariable, deren Verteilung eine stetige Verteilungsfunktion F besitzt, und $X_1,\ X_2, \ldots, X_m$ seien m voneinander unabhängige Kopien. Ausgehend von Realisierungen dieser X_i (sagen wir der Stichprobenvektor $x^m = (x_1, \ldots, x_m)$) kann man eine empirische Verteilungsfunktion konstruieren, die für $(m \to +\infty)$ gegen die exakte Verteilungsfunktion konvergiert. Der Kolmogorov–Smirnov–Test misst die Abweichung der empirischen von der exakten Verteilungsfunktion. Er vergleicht

$$k_m = \sqrt{m} \sup_{-\infty<x<+\infty} |F(x) - F_{x^m}(x)|$$

mit einem "zulässigen" Wert. Ersetzt man den Stichprobenvektor durch die vektorielle Zufallsvariable, so wird k_m ebenfalls eine Zufallsvariable, die wir mit K_m bezeichnen. Die Theorie sagt uns, dass diese im Grenzwert die folgende Verteilung hat:

$$\lim_{m\to+\infty} P(K_m \le x) = H(x) = 1 - 2\sum_{j=1}^{+\infty}(-1)^{j-1}e^{-2j^2x^2}$$

Wie beim χ^2–Test wird man die Hypothese verwerfen, wenn k_m zu groß wird, genauer, wenn

$$k_m > H^{-1}(1-\alpha)$$

[17] Zerlegen Sie zur Übung diese Anweisung, um zu verstehen, wie und warum sie funktioniert.

mit (z.B.) $\alpha = 0.05$ oder $\alpha = 0.01$ gilt. Benutzt man die Approximation $H(x) \simeq 1 - 2e^{-2x^2}$, so erhält man für das Fraktil k_α

$$k_\alpha = \sqrt{\frac{1}{2}\ln(\frac{2}{\alpha})}$$

Die Berechnung von k_m bereitet keine Schwierigkeiten, wenn der Vektor $(x_1,\ x_2,\ \ldots,\ x_m)$ sortiert ist. Nimmt man dies an und beachtet, dass

$$\sup_{x\in[x_i,x_{i+1}[} F_{x^m}(x) - F(x) = \frac{i}{m} - F(x_i), \quad \text{und} \quad \sup_{x\in[x_i,x_{i+1}[} F(x) - F_{x^m}(x) = F(x_{i+1}) - \frac{i}{m}$$

gilt, so lassen sich leicht die folgenden beiden Werte berechnen:

$$k_m^+ = \sqrt{m} \sup_{-\infty<x<+\infty} (F_{x^m}(x) - F(x)) = \sqrt{m} \max_{1\le j\le m} (\frac{j}{m} - F(x_j))$$

$$k_m^- = \sqrt{m} \sup_{-\infty<x<+\infty} (F(x) - F_{x^m}(x)) = \sqrt{m} \max_{1\le j\le m} (F(x_j) - \frac{j-1}{m})$$

Daraus erhält man $k_m = \max(k_m^+,\ k_m^-)$.

5.4.6 Übungen

Ist der Würfel manipuliert?

Sie führen 200 mal den folgenden Versuch durch: Sie werfen den Würfel solange, bis Sie eine 1 erhalten, jedoch höchstens 10 mal. Sie erhalten z.B. die folgenden Ergebnisse:

Zahl der benötigten Würfe	1	2	3	4	5	6	7	8	9	10	≥ 11
Zahl solcher Versuche	36	25	26	27	12	12	8	7	8	9	30

d.h., Sie erhielten in 36 Versuchen sofort beim ersten Wurf eine 1 und in 25 Versuchen kam die 1 (erst) beim zweiten Wurf, u.s.w.

Erstellen Sie einen χ^2–Test zur Beantwortung dieser Frage.

Polya–Urne

Sie ziehen N Kugeln aus einer Polya–Urne. Diese enthält zu Beginn r rote und g grüne Kugeln. Bei jedem Ziehen entnimmt man zufällig eine Kugel der Urne und legt sie zusammen mit c weiteren Kugeln der gleichen Farbe zurück in die Urne. Bezeichnet man mit X_k den Anteil der grünen Kugeln nach k Zügen und mit G_k die Zahl der grünen Kugeln, so gilt

$$X_0 = \frac{g}{g+r}, G_0 = g.$$

Wählt man $g = r = c = 1$, so erhält man

1. $E(X_N) = E(X_0) = X_0 = 1/2$
2. X_N ist auf $\{\frac{1}{N+2}, \ldots, \frac{N+1}{N+2}\}$ gleichverteilt.
3. Für $N \to +\infty$, konvergiert X_N fast überall gegen die Gleichverteilung auf $[0, 1)$.

Führen Sie eine Simulation durch, die die ersten beiden Aussagen bestätigt.

1. Um verschiedene Simulationen vornehmen zu können, kann man eine Funktion mit dem Parameter N programmieren, die N aufeinanderfolgende Ziehungen vornimmt und dann X_N und G_N zurückgibt:

```
function [XN, GN] = Polya_Urne(N)
  // Simulation von N Ziehungen aus einer "Polya-Urne":
  GN = 1 ; G_plus_R = 2 ; XN = 0.5
  for i=1:N
    u = rand()               // Ziehung einer Kugel
    G_plus_R = G_plus_R + 1  // das macht eine Kugel mehr
    if (u <= XN) then        // man hat eine grüne Kugel gezogen (dies geschieht
                             // mit einer Wahrscheinlichkeit von XN
                             // und 1-XN bei einer roten Kugel)
      GN = GN + 1
    end
    XN = GN / G_plus_R       // man aktualisiert den Anteil der grünen Kugeln
  end
endfunction
```

Um aussagekräftige Statistiken durchzuführen, wird diese Funktion sehr oft aufgerufen, und da die Iterationen in Scilab langsam sind, schreiben Sie eine Funktion, die in der Lage ist, m Simulationen "parallel" durchzuführen (hier handelt es sich nicht um wahre Parallelität im Sinne der Informatik — vielmehr vektorisiert man diese Funktion unter Benutzung der in Scilab sehr effizienten Matrixoperationen). Sie können die Funktion `find` benutzen, um die Fälle ausfindig zu machen, in denen eine grüne Kugel gezogen wird.

2. Schreiben Sie ein Skript, um durch Simulation die Aussage über die Erwartungswerte (samt einem Konfidenzintervall) zu bestätigen.

3. Fahren Sie in Ihrem Skript fort, welches mit Hilfe des χ^2–Tests die Hypothese überprüft, dass der Zufallsvariablen X_N eine Gleichverteilung auf $\{\frac{1}{N+2}, \ldots, \frac{N+1}{N+2}\}$ zugrunde liegt.

4. Versuchen Sie, das Vorgehen graphisch zu illustieren; zeichnen Sie z.B. die erhaltene empirische und die exakte Verteilungsfunktion in einer Graphik; in einer zweiten Graphik zeichnen Sie die χ^2–Verteilung mit einer vertikalen Linie beim Fraktil und dem Wert, den der Test ergibt.

Brownsche Bewegung

Für den folgenden stochastischen Prozess (U bezeichne die Gleichverteilung auf $[0, 1]$)

$$X_n(t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} (1_{\{U_i \leq t\}} - t)$$

gilt für festes $t \in]0, 1[$

$$\lim_{n \to +\infty} X_n(t) = Y(t) \sim \mathcal{N}(0, t(1-t))$$

Wir möchten dieses Resultat mit Hilfe von Simulationen veranschaulichen.

Aufgabenstellung:

1. Schreiben Sie eine Scilab Funktion `function X = Brownsche_Bewegung(t,n)`, die eine Realisierung von $X_n(t)$ liefert. Im Folgenden rufen wir diese Funktion m–fach mit einem immer größer[18] werdenden Werten von n auf; man kann daher keine (einfache) Schleife benutzen.

2. Schreiben Sie ein Skript, dass die Simulation der Brownschen Bewegung durchführt: führen Sie m Simulationen von $X_n(t)$ (mit großem n) durch und veranschaulichen Sie graphisch die Konvergenz (in der Verteilung) (, indem Sie in einer Graphik die empirische und die exakte Verteilungsfunktion $Y(t)$ überlagern und damit einen Kolmogorov–Smirnov–Test durchführen). Sie können die Funktion `Brownsche_Bewegung` an den Anfang dieses Skriptes setzen, damit Sie nur mit einer Datei arbeiten müssen.

[18] um $Y(t)$ zu approximieren.

Bemerkung: Es ist nicht einfach, geeignete Werte von m und n auszuwählen: Wenn m größer wird, so gibt es einen Punkt, an dem der Test versagt, da er "spürt", dass n nicht groß genug ist (, da die Normalverteilung nur im Limes $n \to +\infty$ angenommen wird). Man muss daher dann auch n vergrößern. Probieren Sie bei $t = 0.3$ die Werte $n = 1000$ und $m = 200, 500, 1000$, mit denen der Test funktioniert (die Hypothese wird fast nie abgelehnt). Aber mit $m = 10000$ ist der Test fast immer negativ. Erhöht man n (z.B. mit $n = 10000$, aber dann wird die Simulation auf einem PC (Stand 2003) recht langsam), so geht der Test wieder meist positiv aus.

Kapitel 6

Fallstricke

Dieser Teil versucht Fehler, die bei der Benutzung von Scilab häufig auftreten, zu erfassen...

6.1 Elementweise Definition eines Vektors oder einer Matrix

Dieser Fehler ist einer der häufigsten. Betrachten wir das folgende Skript:

```
K = 100  // der einzige Parameter meines Skriptes
for k=1:K
   x(i) = irgend etwas
   y(i) = etwas anderes
end
plot(x,y)
```

Wenn man sein Skript zum ersten Mal ausführt, definiert man die beiden Vektoren `x` und `y` auf die natürliche Art und Weise und alles scheint zu funktionieren... Es gibt aber schon einen kleinen Fehler, denn Scilab definiert bei jeder Iteration die Dimensionen der Vektoren neu (es weiß nicht, dass ihre Endgröße (K,1) ist). Zu beachten ist ebenfalls, dass standardmäßig Spaltenvektoren erzeugt werden. Beim zweiten Ausführen (der Parameter `K` wurde verändert) sind die Vektoren `x` und `y` bekannt und solange `k` kleiner als 100 (der Anfangswert von `K`) ist, begnügt es sich damit, den Wert der Komponenten zu ändern. Folglich erhalten wir in Abhängigkeit vom Wert von `K`:

- `K < 100`, dann haben unsere Vektoren `x` und `y` immer noch 100 Komponenten (allein die ersten `K` Stück sind verändert worden) und die Zeichnung sieht nicht wie beabsichtig so aus.
- `K > 100`, dann hat man anscheinend keine Probleme (ausgenommen der Tatsache, dass die Größe der Vektoren jedes Mal von neuem ab der 101. Iteration verschieden ist).

Das Beste ist, man initialisiert die Vektoren `x` und `y` gemäß:

```
x = zeros(K,1) ; y = zeros(K,1)
```

Dann treten diese Fehler nicht mehr auf. Unser Skript liest sich dann folgendermaßen:

```
K = 100  // der einzige Parameter meines Skriptes
x = zeros(K,1); y = zeros(K,1);
for k=1:K
   x(i) = irgend etwas
   y(i) = etwas anderes
end
plot(x,y)
```

6.2 Apropos Rückgabewerte einer Funktion

Angenommen, man hätte eine Scilabfunktion programmiert, die zwei Argumente zurückgibt, z.B.:

```
function [x1,x2] = resol(a,b,c)
  // Lösung einer Gleichung zweiten Grades  a x^2 + b x + c = 0
  // verbesserte Formel für mehr nummerische Robustheit
  // (indem man die Subtraktion zweier fast gleicher Zahlen vermeidet)
  if (a == 0) then
    error("Der Fall a=0 wird nicht behandelt!")
  else
    delta = b^2 - 4*a*c
    if (delta < 0) then
      error("Der Fall delta < 0 wird nicht behandelt")
    else
      if (b < 0) then
        x1 = (-b + sqrt(delta))/(2*a) ; x2 = c/(a*x1)
      else
        x2 = (-b - sqrt(delta))/(2*a) ; x1 = c/(a*x2)
      end
    end
  end
endfunction
```

Ganz allgemein wird, wenn man eine Funktion im Scilabfenster folgendermaßen aufruft,

```
-->resol(1.e-08, 0.8, 1.e-08)
 ans  =
  - 1.250D-08
```

das Resultat der Variable `ans` zugewiesen. Aber `ans` ist skalar und da diese Funktion zwei Werte zurückgibt, wird nur der erste an `ans` zugewiesen. Um beide Werte zu erhalten, benutzt man die Syntax:

```
-->[x1,x2] = resol(1.e-08, 0.8, 1.e-08)
 x2  =
  - 80000000.
 x1  =
  - 1.250D-08
```

Eine andere sehr böse Falle ist die folgende: Angenommen, man stellt eine kleine Studie über die mit diesen Formeln erhaltene Genauigkeit an (gegenüber jener, die man mit klassischen Formeln erhält). Für einen Satz von Werten für a und c (z.B. $a = c = 10^{-k}$, wobei man mehrere Werte für k nimmt), wird man alle Wurzeln berechnen und sie in zwei Vektoren zur späteren Analyse abspeichern. Es scheint natürlich zu sein, in dieser Weise zu verfahren:

```
  b = 0.8;
  kmax = 20;
  k = 1:kmax;
  x1 = zeros(kmax,1); x2=zeros(kmax,1);
  for i = 1:kmax
    a = 10^(-k(i)); // c = a
    [x1(i), x2(i)] = resol(a, b, a); // FEHLER !
end
```

Aber dies funktioniert nicht[1] (außer die Funktion gibt nur einen einzigen Wert zurück). Man muss daher in zwei Schritten vorgehen:

```
  [rac1, rac2] = resol(a, b, a);
  x1(i) = rac1; x2(i) = rac2;
```

[1] außer in neueren Scilab–Versionen

6.3 Ich habe meine Funktion verändert, aber...

alles scheint wie vor der Änderung zu sein! Sie haben vielleicht vergessen, die Änderung mit Hilfe Ihres Editors zu speichern. Viel wahrscheinlicher ist aber, dass Sie vergessen haben, die Datei, welche diese Funktion enthält, wieder in Scilab mit der Anweisung `getf` (oder `exec`) zu laden ! Ein kleiner Trick: Ihre Anweisung `getf` (oder `exec`) ist sicher nicht sehr weit im Befehls–Logbuch, drücken sie also die Taste ↑, bis Sie sie finden[2].

6.4 Probleme mit `rand`

Standardmäßig liefert `rand` Zufallszahlen gemäß der Gleichverteilung auf $[0, 1)$, aber man kann die Normalverteilung $\mathcal{N}(0, 1)$ mit `rand("normal")` erhalten. Will man wieder die Gleichverteilung erhalten, darf man die Anweisung `rand("uniform")` nicht vergessen. Mit Sicherheit kann man dieses Problem vermeiden, wenn man bei jedem Aufruf die Verteilung angibt (siehe voriges Kapitel).

6.5 Zeilenvektoren, Spaltenvektoren...

In einem Kontext von Matrizen haben sie eine genaue Bedeutung, aber für andere Anwendungen scheint es normal, keinen Unterschied zu machen und eine Funktion so zu schreiben, dass sie in beiden Fällen funktioniert. Will man jedoch schnelle Berechnungen erzielen, ist es ratsamer Matrixausdrücke zu verwenden, anstatt Iterationen zu benutzen; dann muss man jedoch zwischen Spalten– und Zeilenvektoren unterscheiden. Man kann die Funktion `matrix` in der folgenden Art und Weise benutzen:

```
x = matrix(x,1,length(x)) // um einen Zeilenvektor zu erhalten
x = matrix(x,length(x),1) // um einen Spaltenvektor zu erhalten
```

Einen Spaltenvektor kann man auch einfach mit der folgenden Kurzform erhalten:

```
x = x(:)
```

6.6 Vergleichsoperatoren

In gewissen Fällen lässt Scilab das Symbol `=` als Vergleichsoperator zu:

```
-->2 = 1
 Warning: obsolete use of = instead of ==
    !
 ans  =
  F
```

aber es ist besser, stets das Symbol `==` zu benutzen.

6.7 Komplexe und reelle Zahlen

Alles ist in Scilab darauf ausgelegt, reelle und komplexe Zahlen in derselben Weise zu benutzen. Dies ist ziemlich praktisch, kann aber auch zu gewissen Überraschungen führen, beispielsweise dann, wenn Sie eine reelle Funktion ausserhalb ihres Definitionsbereiches (z.B. $\sqrt{x}$ und $\log(x)$ für $x < 0$, $\text{acos}(x)$ und $\text{asin}(x)$ für $x \notin [-1, 1]$, $\text{acosh}(x)$ für $x < 1$) auswerten, denn Scilab gibt die Auswertung der ins Komplexe fortgesetzten Funktion zurück. Um festzustellen, ob man es mit einer reellen oder komplexen Variable zu tun hat, kann man die Funktion `isreal` benutzen:

[2] Am besten benutzen Sie den in neueren Versionen von Scilab eingebauten Editor.

```
-->x = 1
 x  =
    1.

-->isreal(x)
 ans  =
  T

-->c = 1 + %i
 c  =
    1. + i

-->isreal(c)
 ans  =
  F

-->c = 1 + 0*%i
 c  =
    1.

-->isreal(c)
 ans  =
  F
```

6.8 Scilab–Grundbefehle und –Funktionen

In diesem Dokument wurden die Termini Grundbefehl und Funktion mehr oder weniger in gleicher Weise benutzt, um Prozeduren zu bezeichnen, die von der aktuellen Scilabversion angeboten werden. Es gibt jedoch einen grundlegenden Unterschied zwischen einem Grundbefehl, der in Fortran 77 oder in C codiert ist, und einer Funktion (auch Makro genannt), die in der Scilabsprache codiert ist: Eine solche Funktion gilt als eine Scilabvariable und kann dementsprechend als Parameter einer anderen Funktion übergeben werden. Seit Version 2.7 sind auch die Grundbefehle eine Art von Variablen (vom Typ `fptr`); dennoch existieren — außer in neueren Versionen — einige Probleme.

Welcher Art diese Probleme sind, zeigt das folgende Beispiel: Eine kleine Funktion, um ein Integral nach der *Monte–Carlo–Methode* zu berechnen:

```
function [I,sigma]=MonteCarlo(a,b,f,n)
   //              /b
   //  Approx. von |  f(x) dx mit der Monte-Carlo-Methode
   //              /a
   //  man erhält ausserdem die empirische Varianz
   xm = grand(m,1,"unf",a,b)
   ym = f(xm)
   im = (b-a)*mean(ym)
   sm = (b-a)*st_deviation(ym)
endfunction
```

Die Funktion erwartet eine Scilab–Funktion als Argument `f`, jedoch sollte sie auch mit einer eingebauten mathematischen Funktion[3] aufrufbar sein. Aber ein Versuch mit einer eingebauten Funktion (z.B. `exp`) schlägt fehl:

```
-->[I,sigma]=MonteCarlo(0,1,exp,10000)  //  Bug
```

Man kann dieses Problem (, das in aktuellen Versionen von Scilab nicht mehr vorhanden ist), vermeiden, indem man die Funktion `MonteCarlo` mit `getf` und der Option `"n"` lädt:

```
-->getf("MonteCarlo.sci","n")
```

[3] z.B. `sin`, `exp` und viele andere sind eingebaute Funktionen.

6.9 Auswertung von boole'schen Ausdrücken

Im Gegensatz zur Sprache C (oder `C++`) werden boole'sche Ausdrücke der Form

$$a \text{ oder } b$$
$$a \text{ und } b$$

erst ausgewertet, *nachdem* die Teilausdrücke a und b ausgewertet worden sind. Ergibt nun die Auswertung von a den Wert `%T` im ersten bzw. `%F` im zweiten Fall, bräuchte man b gar nicht mehr auswerten, da das Gesamtergebnis schon feststeht — dennoch wertet Scilab b aus, was zu Problemen führt, wenn dabei ein Fehler auftritt; z.B.

```
x= 0
if  x ~= 0  & 1/x > 1, disp("x < 1"), end
               !--error    27
division by zero...
```

That 's all Folks...

Anhang A

Lösungen der Übungsaufgaben

A.1 Lösungen zu den Übungen aus Kapitel 2

1. ```
   --> n  = 5 // um einen Wert für n festzulegen...
   --> A = 2*eye(n,n) - diag(ones(n-1,1),1) - diag(ones(n-1,1),-1)
   ```

   Eine schnellere Methode besteht darin, die Funktion `toeplitz` zu verwenden:

   ```
   -->n=5; // um einen Wert für n festzulegen...
    -->toeplitz([2 -1 zeros(1,n-2)])
   ```

2. Wenn A eine $n \times n$–Matrix ist, dann gibt `diag(A)` einen Spaltenvektor zurück, der die Elemente der Diagonalen von A enthält (also einen Spaltenvektor der Dimension n). `diag(diag(A))` gibt dann eine quadratische Diagonalmatrix der Ordnung n zurück, mit den gleichen Diagonalelementen wie die Ausgangsmatrix.

3. Hier eine Möglichkeit:

   ```
   --> A = rand(5,5)
   --> T = tril(A) - diag(diag(A)) + eye(A)
   ```

4. (a) ```
   --> Y = 2*X.^2 - 3*X + ones(X)
   --> Y = 2*X.^2 - 3*X + 1   // benutzt eine Kurzschreibweise
   --> Y = 1 + X.*(-3 + 2*X) // hier mit dem Hornerschema
   ```

   (b) ```
   --> Y = abs(1 + X.*(-3 + 2*X))
   ```

   (c) ```
   --> Y = (X - 1).*(X + 4)  // benutzt eine Kurzschreibweise
   ```

   (d) ```
   --> Y = ones(X)./(ones(X) + X.^2)
   --> Y = (1)./(1 + X.^2) // mit Kurzformen
   ```

5. Hier das Skript:

   ```
   n = 101;                          // für die Diskretisierung
   x = linspace(0,4*%pi,n);
   y  =  [1  , sin(x(2:n))./x(2:n)]; // vermeidet die Division durch Null
   plot(x,y,"x","y","y=sin(x)/x")
   ```

6. Ein mögliches Skript (zum Experimentieren mit verschiedenen Werten von n) ist:
   ```

```
  n = 2000;
  x = rand(1,n);
  xbar = cumsum(x)./(1:n);
  plot(1:n, xbar, "n","xbar", ...
      "Gesetz der großen Zahl: xbar(n) -> 0.5  für  n -> + oo")
```

A.2 Lösungen zu den Übungen aus Kapitel 3

1. Der klassische Algorithmus hat zwei Schleifen:

```
function  x =  sol_tri_sup1(U,b)
  //
  //  Lösung von   Ux  =  b , wobei U eine obere Dreiecksmatrix ist.
  //
  //  Bemerkung: Dieser Algo. funktioniert im Falle vielfacher rechter Seiten
  //  (jede rechte Seite entspricht einer Spalte von b)
  //
  [n,m] =  size(U)
  // einige Überprüfungen...
  if  n  ~=  m then
     error('Die Matrix ist nicht quadratisch!')
  end
  [p,q] =  size(b)
  if p ~=  m then
     error('rechte Seite inkompatibel')
  end
  // Anfang des Algo.
  x = zeros(b) // man reserviert Platz für x
  for i = n:-1:1
     summe = b(i,:)
     for j =  i+1:n
        summe = summe -  U(i,j)*x(j,:)
     end
     if U(i,i) ~= 0 then
        x(i,:) = summe/U(i,i)
     else
        error('Matrix nicht invertierbar')
     end
  end
endfunction
```

Hier eine Version, die eine einzige Schleife benutzt:

```
function x = sol_tri_sup2(U,b)
  //
  //  siehe sol_tri_sup1, außer dass ein bisschen mehr
  //  Matrixnotation verwendet wird
  //
  [n,m] = size(U)
  //  einige Überprüfungen...
  if  n ~= m then
     error('Die Matrix ist nicht quadratisch!')
  end
```

```
  [p,q] = size(b)
  if p ~= m then
     error('Rechte Seite inkompatibel')
  end
  // Anfang des Algo.
  x = zeros(b) // man reserviert Platz für x
  for i = n:-1:1
     summe = b(i,:) - U(i,i+1:n)*x(i+1:n,:) // siehe Kommentar am Ende
     if U(i,i) ~= 0 then
        x(i,:) = summe/U(i,i)
     else
        error('Matrix nicht invertierbar')
     end
  end
endfunction
```

Kommentar: Bei der ersten Iteration (die $i = n$ entspricht) sind die Matrizen `U(i,i+1:n)` und `x(i+1:n,:)` leer. Sie entsprechen einem Objekt, das in Scilab wohldefiniert ist (die leere Matrix), die so [] notiert wird. Die Addition mit einer leeren Matrix ist definiert und liefert `A = A + []`. Bei der ersten Iteration erhält man also `summe = b(n,:) + []`, d.h. dass `summe = b(n,:)` ist.

2.
```
//
// Skript zur Lösung von x" + alpha*x' + k*x = 0
//
// Um die Gleichung in ein System 1. Ordnung umzuwandeln,
// setzt man  X(1,t) = x(t) und X(2,t) = x'(t)
//
// Man erhält dann  X'(t) = A X(t)  mit  A = [0 1;-k -alpha]
//
k =  1;
alpha =  0.1;
T = 20;    //  Endzeitpunkt
n   = 100; //  Zeitdiskretisierung: Das Intervall [0,T] wird
           //  in n Intervalle zerlegt
t  = linspace(0,T,n+1); // die Zeitpunkte X(:,i) entsprechen X(:,t(i))
dt = T/n;  // Zeitschrittweite
A = [0 1;-k -alpha];
X = zeros(2,n+1);
X(:,1) = [1;1]; // die Anfangsbedingungen
M = expm(A*dt); // Berechnung der Exponentialfunktion von A dt

// die Rechnung
for i=2:n+1
  X(:,i) =  M*X(:,i-1);
end

// Anzeige der Ergebnisse
xset("window",0)
xbasc()
xselect()
plot(t,X(1,:),'Zeit','Ort','Kurve x(t)')
xset("window",1)
xbasc()
xselect()
plot(X(1,:),X(2,:),'Ort','Geschw.','Trajektorie im Phasenraum')
```

3.
```
function [i,info]=intervall_von(t,x)
   // binäre Suche des Intervalls i mit  x(i)<=  t < x(i+1)
   // Ist t nicht in [x(1),x(n)] enthalten, wird info = %f zurückgegeben
   n=length(x)
   if (t<x(1)) | (t>x(n)) then
      info = %f
      i = 0 // Standardwert
   else
      info = %t
      i_beg=1
      i_end=n
      while  i_end - i_beg > 1
         itest = floor((i_end + i_beg)/2 )
         if ( t >= x(itest) ) then i_beg= itest, else, i_end=itest, end
      end
      i=i_beg
   end
endfunction
```

4.
```
function p=myhorner(t,x,c)
   // Auswertung des Polynoms  c(1)+c(2)*(t-x(1))+c(3)*(t-x(1))*(t-x(2))+...
   // durch den Horner-Algorithmus...
   // t ist ein Vektor von Zeitpunkten (oder eine Matrix)
   n=length(c)
   p=c(n)*ones(t)
   for k=n-1:-1:1
      p=c(k)+(t-x(k)).*p
   end
endfunction
```

5. Erzeugen einer abgebrochenen Fourierreihe:

```
function y=signal_fourier(t,T,cs)
  // Diese Funktion gibt ein T-periodisches Signal zurück.
  // t  : ein Vektor von Zeitpunkten, für welche man
  //      das Signal y berechnet ( y(i) entspricht t(i) )
  // T  : die Periode des Signals
  // cs : ist ein Vektor, der die Amplitude jeder Funktion f(i,t,T) liefert
  l=length(cs)
  y=zeros(t)
  for j=1:l
     y=y + cs(j)*f(j,t,T)
  end
endfunction

//-------------------------------------------------------------------

function y=f(i,t,T)
  // die trigonometrischen Polynome für ein Signal der Periode T:
  // falls i gerade  : f(i)(t)=sin(2*pi*k*t/T)  (mit k=i/2)
  // falls i ungerade: f(i)(t)=cos(2*pi*k*t/T)  (mit k=floor(i/2)),
  // wobei insbesondere f(1)(t)=1, was unten als besonderer Fall
  // behandelt wird, obwohl es nicht notwendig ist.
  // t ist ein Vektor von Zeitpunkten
  if i==1 then
```

```
    y=ones(t)
  else
    k=floor(i/2)
    if modulo(i,2)==0 then
      y=sin(2*%pi*k*t/T)
    else
      y=cos(2*%pi*k*t/T)
    end
  end
endfunction
```

6. Das Treffen: Seien T_A und T_B die Ankunftszeiten von Herrn A und Fräulein B. Dies sind zwei stochastisch unabhängige Zufallsvariablen der Verteilung $U([17,18])$. Das Treffen findet statt, wenn $[T_A, T_A+1/6] \cap [T_B, T_B+1/12] \neq \emptyset$. Ein Experiment stellt eine Realisation der vektorwertigen Zufallsvariablen $R = (T_A, T_B)$ dar, die — gemäß unserer Annahmen — auf $[17,18] \times [17,18]$ gleichverteilt ist. Die Wahrscheinlichkeit eines Treffens ist das Verhältnis des Flächeninhaltes des folgenden Bereiches (hier findet das Treffen statt)

$$\begin{cases} T_A \leq T_B + 1/12 \\ T_B \leq T_A + 1/6 \\ T_A, T_B \in [17,18] \end{cases}$$

und der Fläche des Quadrats(1), womit man $p = 67/288$ erhält. Zur Berechnung der Wahrscheinlichkeit kann man annehmen, dass das Treffen zwischen Mitternacht und ein Uhr stattfindet. Führt man m Simulationen durch, so ist die Wahrscheinlichkeit gleich dem Quotienten der Anzahl derjenigen Simulationen, bei denen das Treffen stattfindet, geteilt durch m; dies geht z.B. so:

```
function p = rdv(m)
   tA = rand(m,1)
   tB = rand(m,1)
   Treffen = tA+1/6 > tB  &  tB+1/12 > tA;
   p = sum(bool2s(Treffen))/m
endfunction
```

man kann auch die Funktion `grand` benutzen, die im Kapitel “Anwendungen und Ergänzungen” beschrieben wird:

```
function p = rdv(m)
   tA = grand(m,1,"unf",17,18)
   tB = grand(m,1,"unf",17,18)
   Treffen = tA+1/6 > tB  &  tB+1/12 > tA;
   p = sum(bool2s(Treffen))/m
endfunction
```

A.3 Lösungen zu den Übungen aus Kapitel 4

Ist der Würfel manipuliert?

Bezeichnet man mit J die Zufallsvariable, die den Ausgang des Experimentes beschreibt, so ist diese gemäß einer geometrischen Verteilung $G(p)$ mit $p = 1/6$ verteilt, falls der Würfel nicht manipuliert wurde. Fasst man alle Versuche, die (echt) mehr als 10 Würfe benötigt hätten, zu einer Klasse zusammen (siehe Aufgabenstellung), so erhalten wir mit $q = 1 - p = 5/6$ die folgenden Wahrscheinlichkeiten:

$$P(J = 1) = p,\ P(J = 2) = qp,\ P(J = 3) = q^2p,\ ....,\ P(J = 10) = q^9p,\ P(J > 10) = q^{10}$$

Andererseits enthält die gegebene Tabelle direkt die (zugehörigen) Häufigkeiten; dies führt zu dem Skript:

```
m= 200;
occ= [36 ; 25 ; 26 ; 27 ; 12 ; 12 ; 8 ; 7 ; 8 ; 9 ; 30];
p = 1/6; q = 5/6;
pr = [p*q.^(0:9) , q^10]';
y = sum( (occ - m*pr).^2 ./ (m*pr) );
y_fraktil = cdfchi("X", 10, 0.95, 0.05);
mprintf("\nWürfeltest :\n")
mprintf("y = %g, y_fraktil (95 \%) = %g\n", y, y_fraktil)
mprintf("%d*min(pr) = %g\n", m, m*min(pr))
```

Man erhält $y = 7.73915$ während $y_{fraktil} = 18.307$ ist; daher scheint der Würfel korrekt zu sein. Jedoch ist $200 \times \min(pr) \simeq 6.5$ zu klein für die Anwendbarkeit des Tests (man muss daher noch mehr Experimente machen).

Polya–Urnen–Funktion

```
function [XN, VN] = parallele_Polya_Urne(N,m)
   VN = ones(m,1) ; V_plus_R = 2 ; XN = 0.5*ones(m,1)
   for i=1:N
      u = grand(m,1,"def")     // Ziehen einer Kugel (aus jeder der m Urnen)
      V_plus_R = V_plus_R + 1  // das macht eine Kugel mehr
      ind = find(u <= XN)      // findet Nummern der Urnen, aus denen
                               // man eine grüne gezogen hat
      VN(ind) = VN(ind) + 1    // erhöht die Anzahl der grünen Kugeln in diesen Urnen
      XN = VN / V_plus_R       // aktualisiert das Verhältnis der grünen Kugeln
   end
endfunction
```

Das Polya–(Scilab)–Skript

```
// Polya-Simulation:
N = 10;
m = 5000;
[XN, VN] = parallele_Polya_Urne(N,m);

// 1.) berechne den Erwartungswert und das Konfidenzintervall
EN = mean(XN);            // empirischer Erwartungswert
sigma = st_deviation(XN); // empirische Varianz
delta =  2*sigma/sqrt(m); // Inkrement für das empirische Intervall (2 statt 1.9599..)
mprintf("\n  E exakt = 0.5\n");
mprintf("  E geschätzt = %g\n",EN);
mprintf("  95\%-ges (empirisches) Konfidenzintervall : [%g,%g]\n",EN-delta,EN+delta);

// 2.) chi-Quadrat-Test
alpha = 0.05;
p = 1/(N+1);  //  theoretische Wahrscheinlichkeit für jedes Resultat (Gleichverteilung)
occ = zeros(N+1,1);
for i=1:N+1
```

```
  occ(i) = sum(bool2s(XN == i/(N+2)));
end
if sum(occ) ~= m then, error(" Fehler..."), end  // kleiner Test
Y = sum( (occ - m*p).^2 / (m*p) );
Y_fraktil = cdfchi("X",N,1-alpha,alpha);
mprintf("\n chi-Quadrat-Test: \n")
mprintf(" -------------- \n")
mprintf(" durch den Test erhaltener Wert : %g\n", Y);
mprintf(" Schwellwert, der nicht überschritten werden sollte : %g\n", Y_fraktil);
if (Y > Y_fraktil) then
  mprintf(" vorläufiges Ergebnis: Hypothese verworfen!\n")
else
  mprintf(" vorläufiges Ergebnis: Hypothese nicht verworfen!\n")
end

// 3.) graphische Darstellung
function d = chi2_Dichte(X,N)
   d = X.^(N/2 - 1).*exp(-X/2)/(2^(N/2)*gamma(N/2))
endfunction

haeufigkeiten = occ/m;
ymax = max([haeufigkeiten ; p]); rect = [0 0 1 ymax*1.05];
xbasc(); xset("font",2,2)
subplot(2,1,1)
   plot2d3((1:N+1)'/(N+2), haeufigkeiten, style=2 , ...
           frameflag=1, rect=rect, nax=[0 N+2 0 1])
   plot2d((1:N+1)'/(N+2),p*ones(N+1,1), style=-2, strf="000")
   xtitle("empirische Häufigkeiten (vertikale Linien) und " + ...
          "exakte Wahrscheinlichkeiten (Kreuze)")
subplot(2,1,2)
   // chi-Quadrat-Dichte in N Freiheitsgraden
   X = linspace(0,1.1*max([Y_fraktil Y]),50)';
   D = chi2_Dichte(X,N);
   plot2d(X,D, style=1, leg="chi-Quadrat-Dichte", axesflag=2)
   // ein vertikaler Strich für Y
   plot2d3(Y, chi2_Dichte(Y,N), style=2, strf="000")
   xstring(Y,-0.01, "Y")
   // ein vertikaler Strich für  Y_fraktil
   plot2d3(Y_fraktil, chi2_Dichte(Y_fraktil,N), style=5, strf="000")
   xstring(Y_fraktil,-0.01, "Y_fraktil")
   xtitle("Wert von Y im Vergleich zum Fraktil")
```

Hier ein (mögliches) Resultat mit $N = 10$ und $m = 5000$ (siehe Abb. (A.1)):

```
 E exakt = 0.5
 E geschätzt = 0.500267
 95%-ges (empirisches) Konfidenzintervall: [0.492724,0.507809]
chi-Quadrat-Test:
--------------
durch den Test erhaltener Wert: 5.1276
Schwellwert, der nicht überschritten werden sollte: 18.307
vorläufiges Ergebnis: Hypothese nicht verworfen!
```

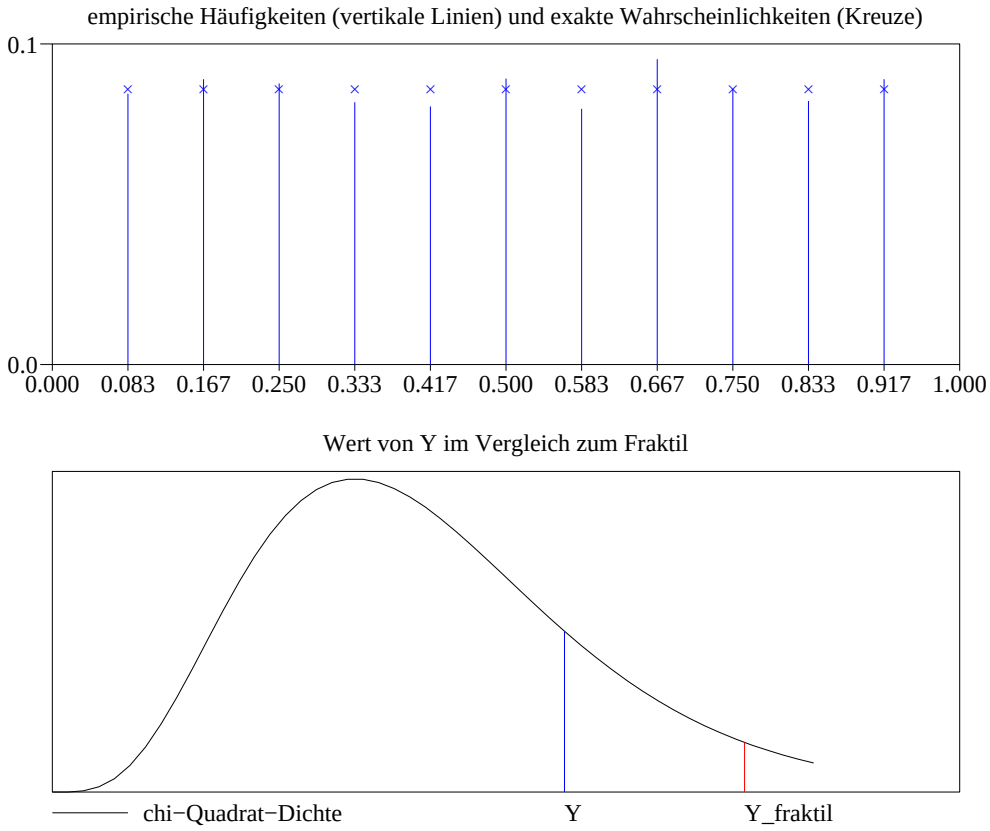

Abbildung A.1: Demonstration des χ^2 Tests anhand der Polya–Urne

Brownsche Bewegung

Die Funktion

```
function X = Brownsche_Bewegung(t,n)
   X = sum(bool2s(grand(n,1,"def") <= t) - t)/sqrt(n)
endfunction
```

Das Skript

Dieses Skript zeigt nach m Simulationen der Zufallsvariablen $X_n(t)$ den Graphen der empirischen Verteilungsfunktion, überlagert diesen dann mit der erwarteten Verteilungsfunktion (für $n = +\infty$) und führt schließlich den Kolmogorov–Smirnov–Test durch:

```
t = 0.3;
sigma = sqrt(t*(1-t));  // erwartete Standardabweichung
n = 4000;               // "großes" n
m = 2000;               // Anzahl der Simulationen
X = zeros(m,1);         // Initialisierung des Vektors der Realisierungen
for k=1:m
   X(k) = Brownsche_Bewegung(t,n);  // Schleife zum Berechnen der Realisierungen
end

// das Bild der empirischen Verteilungfunktion
X = - sort(-X); // sortieren
prcum = (1:m)'/m;
xbasc()
plot2d2(X, prcum)
x = linspace(min(X),max(X),60)';               // die Abszissen und
[P,Q]=cdfnor("PQ",x,0*ones(x),sigma*ones(x));  // die Ordinaten für die exakte
                                               // Funktion
```

```
 plot2d(x,P,style=2, strf="000")                // zum ersten Bild dazu

 // Bereitstellen des KS-Tests
 alpha = 0.05;
 FX = cdfnor("PQ",X,0*ones(X),sigma*ones(X));
 Dplus = max( (1:m)'/m - FX );
 Dmoins = max( FX - (0:m-1)'/m );
 Km = sqrt(m)*max([Dplus ; Dmoins]);
 K_fraktil = sqrt(log(1/alpha)/2) - 1/(6*sqrt(m)) ;

// Anzeige des Resultats
//
 mprintf(" KS-Test: \n")
 mprintf(" -------- \n")
 mprintf("  durch den Test erhaltener Wert: %g\n", Km);
 mprintf("  Fraktil, das nicht überschritten werden sollte: %g\n",K_fraktil);
 if (Km > K_fraktil) then
    mprintf("  vorläufiges Ergebnis: Hypothese verworfen!\n")
 else
    mprintf("  vorläufiges Ergebnis: Hypothese nicht verworfen!\n")
 end
```

Für $n = 1000$ und $m = 4000$ erhält man z.B.:

```
KS-Test:
--------
 durch den Test erhaltener Wert: 0.793264
 Fraktil, das nicht überschritten werden sollte: 1.22015
 vorläufiges Ergebnis: Hypothese nicht verworfen!
```

Index

übliche mathematische Funktionen, 9

boolsche Operatoren, 30

Erzeugen von Zufallszahlen
    grand, 94
Erzeugen von Zufallszahlen
    rand, 91

komplexe Zahlen
    eine komplexe Zahl eingeben, 6

Laden und Speichern von Dateien, 51
Listen, „typisierte“, 36

Matrizen
    Eine Matrix verändern, 26
Matrizen
    Eigenwerte, 26
    Ein lineares Gleichungssystem lösen, 13
    eine Matrix eingeben, 5
    Elementweise Operationen, 12
    Lösen eines linearen Gleichungssystems, 20
    leere Matrix `[]`, 22
    leere Matrix `[]` , 9
    Summe, Produkt, Transponieren, 10
    Zusammenfügen und Extrahieren, 14

Programmieren
    Zuweisung, 9
Programmierung
    break, 43
    eine Funktion direkt definieren, 49
    for–Schleife, 29
    Funktionen, 39
Programmierung
    eine Anweisung fortsetzen , 6
    konditionaler Ausdruck: if then else, 31
    konditionaler Ausdruck: select case, 31

Scilab–Grundbefehle
    `file`, 51
    `timer`, 56
Scilab–Grundbefehle
    `argn`, 46
    `bool2s`, 38
    `cumprod`, 23
    `cumsum`, 23
    `diag`, 7
    `error`, 44
    `evstr`, 50
    `execstr`, 50
    `expm`, 12
    `eye`, 7
    `find`, 38
    `input`, 18
    `length`, 27
    `linspace`, 8
    `logspace`, 26
    `matrix`, 26
    `mean`, 25
    `ode`, 87
    `ones`, 7
    `plot`, 18
    `prod`, 23
    `rand`, 8
    `read`, 54
    `size`, 27
    `spec`, 26
    `st_deviation`, 25
    `stacksize`, 17
    `sum`, 22
    `triu tril`, 8
    `type` und `typeof`, 45
    `warning`, 45
    `who`, 16
    `write`, 52
    `zeros`, 7

Vergleichsoperatoren, 30