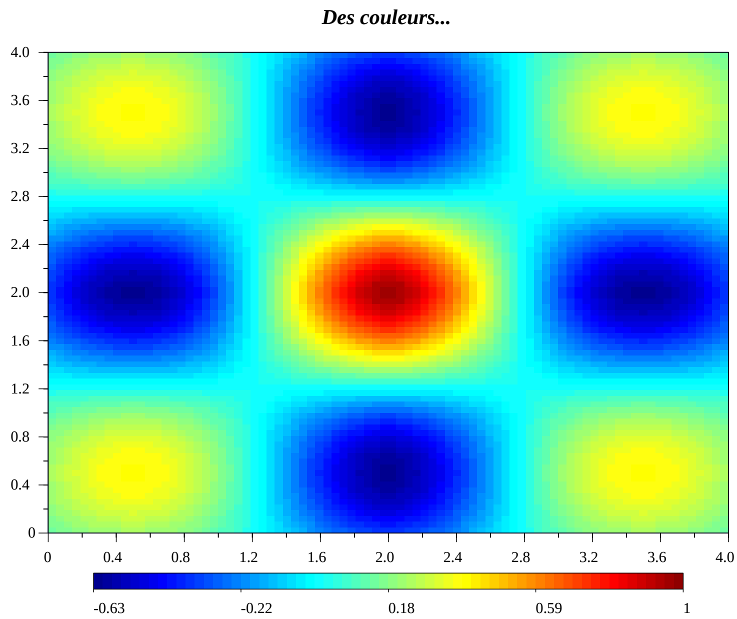
Eine Einführung in Scilab

version 0.997alpha



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

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.

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
- und an alle Leser für ihre Ermutigungen, Bemerkungen und Korrekturen

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

Inhaltsverzeichnis

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 2.6 (oder neuer) von Scilab besitzen. Für Auskünfte aller Art konsultiere man die Scilab–Homepage:

`http://www-rocq.inria.fr/scilab/`,

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

Die "Scilab–Gruppe" schreibt seit Dezember 1999 regelmäßig einen Artikel im "Linux magazine". Viele Aspekte von Scilab, von denen die meisten nicht in dieser Einführung erwähnt werden, wurden dort behandelt; daher empfehle ich diese. 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. Zum Schluss möchte ich Sie auf ein interessantes Dokument "Scilab Bag Of Tricks" von Lydia E. van Dijk und Christoph L. Spiel hinweisen, das Sie unter folgender Adresse finden:

`http://www.hammersmith-consulting.com/scilab/sci-bot/sci-bot.html`

Es ist in verschiedenen Formaten (HTML, SGML, postscript, PDF) verfügbar.

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, das folgende Kommando 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:

```
                                        ===========
                                         S c i l a b
                                        ===========


                                      scilab-2.6
                          Copyright (C) 1989-2001 INRIA


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 es, die Liste ihrer Elemente über die Tastatur einzugeben, unter Einhaltung folgender Konventionen:

- Die Elemente in einer Zeile sind durch Leerzeilen 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.

[1]unter Unix muss sichergestellt werden, dass die Variable `PATH` den Pfad für den Zugriff auf die Software enthält; nach Castor, Pollux et Océanos sollte dieser Pfad `/usr/local/lib/scilab-2.6/bin` sein ; stellen Sie sicher, dass dieser in Ihrer Datei `.login` enthalten ist.

Z.B. produziert der Befehl

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

folgende Ausgabe:

```
 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):

- 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 $\mathbb{K}^n$.)

- 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).

- Die Funktionen `zeros` und `ones` erlauben jeweils das Erzeugen von Matrizen mit Null- bz. 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 Umghebung 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. !
```

- 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. !
```

- 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 !
```

- 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 arithmetischen Einheit 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 Ausdrücke von Matrizen

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.

`abs`	Absolutbetrag
`exp`	Exponentialfunktion
`log`	natürlicher Logarithmus
`log10`	10er–Logarithmus
`cos`	Cosinus (Argument im Bogenmaß)
`sin`	Sinus (Argument im Bogenmaß)
`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

So kann man, um eine Matrix koeffizientenweise zu erzeugen, neben Konstanten (die nichts anderes als elementare Ausdrücke sind) beliebige Ausdrücke verwenden, die einen skalaren (reell oder komplex) 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).

2.3.1 Einige elementare Beispiele für Matrixausdrücke

Alle üblichen Operationen sind auf 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, benutzt man die Operatoren `.*` und `./` ein: `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 haben 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 Vektornorm (und auch der 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, das 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 $v = (k_1, k_2, \dots, k_p)$ 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.

2.4 Daten sichern und einlesen

Hat man eine Matrix (oder einen Vektor) aus reellen Zahlen in der Umgebung definiert, ist es möglich, diese in einer Datei (im ASCII–Format) zu speichern; hier ein Beispiel dazu:

```
-->A      // angebracht, um sich an A zu erinnern

 A  =

!   1.    2.     3.     4.   !
!   1.    4.     9.     16.  !
!   1.    8.     27.    64.  !
!   1.    16.    81.    256. !

-->write('mat.dat',A)     // mat.dat ist der Dateiname
```

Und wenn eine Matrix in einer Datei (in ASCII) gespeichert ist, kann man sie vollständig oder teilweise mit dem Befehl `read` lesen. Auf das vorhergehende Beispiel bezugnehmend:

```
-->Anew=read('mat.dat',2,2)
Anew =

!  1.  2. !
!  1.  4. !

-->Anew=read('mat.dat',4,2)
Anew =

!  1.  2.  !
!  1.  4.  !
!  1.  8.  !
!  1.  16. !

-->Anew=read('mat.dat',4,4)
Anew =

!  1.  2.  3.   4.   !
!  1.  4.  9.   16.  !
!  1.  8.  27.  64.  !
!  1.  16. 81.  256. !
```

Die Argumente dieses Befehls sind jeweils der Dateiname und die Anzahl der Zeilen und Spalten der Matrix (die man lesen will). Ist die genaue Anzahl der Zeilen nicht bekannt, kann man sie durch -1 ersetzen (die Datei wird dann bis zum Ende gelesen). Diejenigen Funktionen, die mit der Funktion `file` verknüpft sind, ermöglichen sehr flexibles Lesen / Schreiben (vgl. Programmierung). Für alle, die C kennen existiert auch eine Nachbildung der Funktionen `fscanf` und `fprintf`.

Man kann auch binär speichern und dann laden — alle oder einen Teil der Variablen, die man mit Hilfe der Befehle `load` und `save` definiert hat:

- `save('datname')` sichert alle Variablen in der Datei `datname`.

- `save('datname',x1,x2,...)` sichert in `datname` nur die Variablen x1, x2, ...
- `load('datname')` lädt alle Variablen, die in `datname` gesichert worden sind.
- `load('datname','x1','x2',...)` lädt nur die Variablen x1, x2, ... unter den in `datname` gespeicherten Variablen.

2.5 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.

2.6 Benutzen der Online–Hilfe

Man erhält sie durch das Anklicken des `Help`–Buttons im Scilabfenster... Es erscheint ein neues Fenster (mit der Überschrift *Scilab Help Panel*), das in drei Teile unterteilt ist (zusätzlich der Button `done`, der dieses Fenster schließt). Der mittlere Teil entspricht der Einteilung aller Funktionen in eine bestimmte Rubriken (`Scilab Programming, Graphic Library, Utilities and Elementary functions,...`), während der erste Teil die Liste aller Funktionen[2] derjenigen Rubrik enthält, die im zweiten Teil invers erscheint... Um eine Rubrik zu wechseln, genügt es, die gewünschte einmal anzuklicken. Will man Details über eine Funktion erhalten, muss man die entsprechende Überschrift anklicken (stets nur einmal); dann erscheint ein neues Fenster mit diesen Details. Um dieses Fenster zu schließen, muss man auf den Button `close window` klicken (aber lassen Sie es geöffnet, solange Sie es brauchen). Wenn man nicht weiß, wo man suchen soll, kann man im dritten Teil — mit `Apropos` überschrieben — das Schlüsselwort angeben und dann auf die `Return`–Taste drücken, um den Suchvorgang zu starten. Bleibt die Suche erfolglos, erscheint die Meldung „`no info for topics ... back to chapter one`“ (die mitteilt, dass der erste Teil des Hilfe–Fensters die Funktionen der Rubrik 1 (Scilab Programming) zeigt). Hatte die Suche Erolg, zeigt der erste Fensterteil alle Funktionen, die das Schlüsselwort in ihrer Überschrift (d.h. im Namen der Funktion oder in der Kurzbeschreibung) enthalten. Zum Beispiel wird die Suche nach dem Schlüsselwort `int` enorm viele Funktionen ausgeben, da ja die Zeichenkette `int` Bestandteil der Wörter interrupt, interface, printing, points usw. ist. Folgt dem Wort `int` ein Leerzeichen, werden weniger Funktionen ausgewählt und die gesuchte Funktion erscheint selbstverständlich in der Auswahlliste. Andererseits können Sie, wenn Sie die Hilfe–Seite eines Scilab–Grundbefehls aufrufen wollen, deren Namen Sie kennen (angenommen sie hieße `funk`), einfach das Kommando `help funk` im Hauptfenster (nach dem Prompt `-->`) eingeben, und die Seite erscheint sofort (ohne das „*Scilab Help Panel*“ benutzen zu müssen).

2.7 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.

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.

2.8 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")
```

[2] für jede Funktion erscheint deren Name gefolgt von einer kurzen Beschreibung

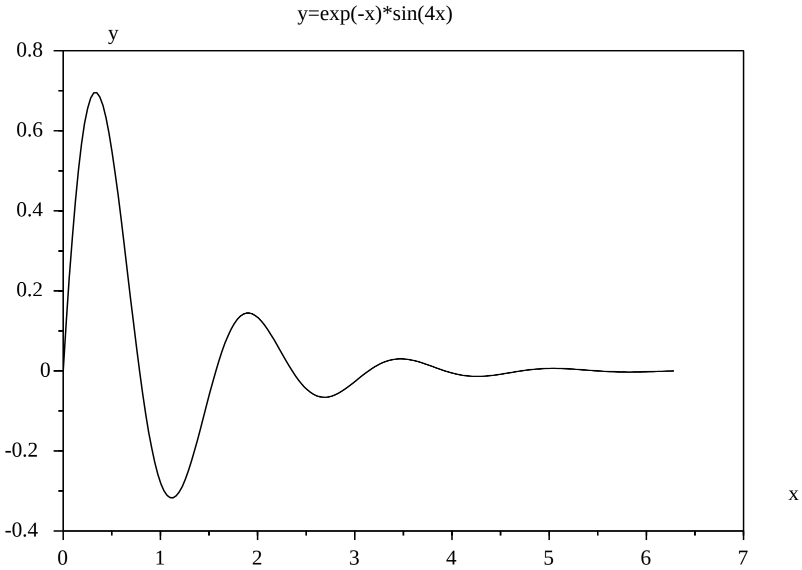


Abbildung 2.1: Eine einfache Zeichnung

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, hat ihr Name die Endung `.sci`). 2.) Der Editor `emacs` kann mit einem Eingabemodus ausgestattet werden (auf der Scilab–Homepage zu erhalten), der das Schreiben von Scilabprogrammen erleichtert. 3.) Ein Skript dient oft als Hauptprogramm einer Anwendung, die in Scilab geschrieben wurde.

2.9 Diverse Ergänzungen

2.9.1 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)
```

2.9.2 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 einzelne 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[3] 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[4] 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
```

[3] 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.

[4] 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`).

```
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[5] 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:

  ```
  -->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.

[5] 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.

2.9.3 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,"row")  // addiert spaltenweise -> man erhält eine Zeile
 ans  =

!   5.    7.    9. !

-->sum(B,"col") // 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,"row")  // Geben Sie B ein, um nochmal diese Matrix zu sehen...
 ans  =

!   4.    10.    18. !

-->prod(B,"col")
 ans  =

!   6.   !
!   120. !

-->prod(B)
 ans  =

    720.

-->prod([])
 ans  =

    1.
```

Man erhält wiederum die in der Mathematik übliche Konvention:

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

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_new = matrix(B,3,2)   // nocheinmal B eingeben...
 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, Beispiel:

```
-->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.

2.9.4 Die Funktionen `size` und `length`

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

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

    3.
 nl  =

    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.

2.10 Übungen

1. Definieren Sie die folgende Matrix der Ordnung n (für die Details über die 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. Was ist die maximale Größe einer quadratischen Matrix, die man in der Scilab–Umgebung erzeugen kann (unter der Annahme, dass es die einzige Variable ist, die man erzeugt, außer der Variable `ans`, welche dazu dient, die Dimensionen dieser Matrix zu berechnen.
   Hilfe: Man kann zuerst `who` eingeben, um die Anzahl der freien Speicherwörter herauszufinden, und man wird feststellen, dass das Erzeugen einer Variablen zwei Wörter im Heap verbraucht (zuzüglich derer für den Inhalt).

Kapitel 3

Die Programmierung in Scilab

Scilab verfügt — neben eingebauten Grundbefehlen(, die eine sehr kopakte, 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). Jedoch in einem Skript (oder einer Funktion) ist der Zeilenende 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, in dem 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, in dem 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 fertg...   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)

!   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.

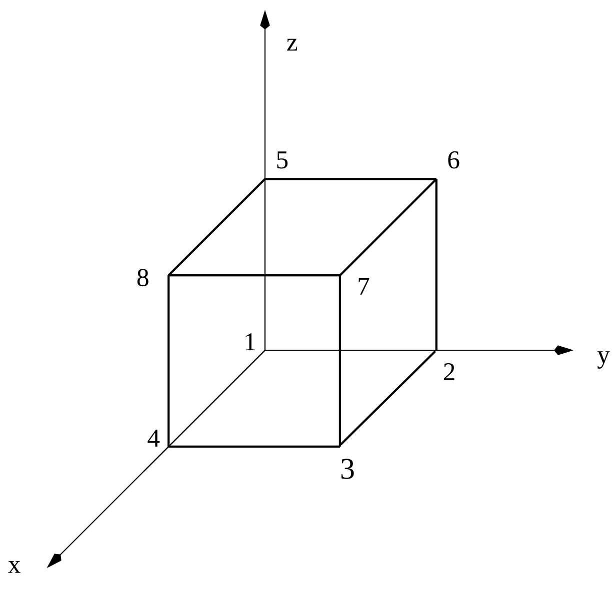


Abbildung 3.1: Nummerierung der Ecken eines Würfels

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 2 3 4; 5 8 7 6; 3 7 8 4;...   // die Seiten
-->        2 6 7 3; 1 5 6 2; 1 4 8 5];
```

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

```
-->Cube = tlist(["Polyeder","Koord","Seite"],P,Ecken)
 Cube  =


       Cube(1)

!Polyeder  Koord  Seite  !

       Cube(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. !

       Cube(3)

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

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

```
-->Cube("Koord")(:,2)
 ans  =

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

-->Cube("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 zukunftige 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. Eine Funktion endet mit dem Schlüsselwort `endfunction`. In dem Fall, dass die Datei mehrere Funktionen enthält, kann auch die Anweisung `function ...` der nächsten Funktion, bzw. das Dateiende diese Rolle spielen. *Bem.:*

- die erste Zeile Ihrer Datei muss auf jeden Fall mit dem Befehl `function ...` beginnen, d.h. dass die erste Funktion einer Datei nicht vor ihrer Kopfzeile kommentiert werden;
- die letzte Anweisung der Datei muss zwangsläufig von einem Zeilenende gefolgt sein, ansonsten nimmt der Interpreter keine Notiz von ihr;
- 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` hineingeschrieben. Damit Scilab sie erkennen kann, muss die Datei mit folgendem Befehl geladen werden:

```
getf('facts.sci')    //  oder kürzer      getf 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 das zweite Beispiel vorgestellt wird, 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).

Hier ein zweites 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 (*)

Bis zur Version 2.3.1 sind alle Eingabevariablen per Wert übergeben worden, d.h. dass eine Kopie der effektiven Argumente stattfand: Ist das effektive Argument eine Matrix, dann entspricht das formale Argument einer Kopie dieser Matrix. Sie können also die Eingabevariablen ändern ohne jegliche Konsequenzen für die entsprechenden Variablen beim Aufruf. So könnte man in obigem Beispiel am Ende den folgenden Befehl hinzufügen:

```
c=ones(c)
```

welcher den Wert der Tabelle `c` lokal verändern würde, aber keinerlei Auswirkung auf die Tabelle `coef` in der Aufrufumgebung hätte. Ab der Version 2.4 werden die Eingabevariablen per Referenz[6] übergeben; werden diese doch in der Funktion verändert, so wird zuvor eine Kopie erstellt. Diese als „copy–on–write“ bekannte Technik demonstriert das folgenden Beispiel:

```
function sz=cp_on_wrt(A,con)
  if con
    A(1,1)=1;
  end
sz=stacksize();
endfunction

A=rand(500,500);
```

[6] im Fall einer Matrix kann einfach die Adresse des ersten Elements sowie die Anzahl der Zeilen und Spalten (und ein Hinweis, der deutlich macht, dass es sich um eine Matrix handelt) angegeben werden

```
first= cp_on_wrt(A,%f);
second=cp_on_wrt(A,%t);

second(2)-first(2)
 ans  =  250004.
```

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. Hier ein Beispiel – man betrachte die folgende Funktion:

```
function  y1 = test(x1)
  y1 =  x1 + c    // man benutzt die globale Variable c (wenn sie existiert ...)
  disp(c,'c  = ') // disp ermöglicht, die Variablen am Bildschirm zu visualisieren
  c(5,2) = 2      // eine interne Variable c wird erzeugt
  disp(c,'c = ')
endfunction
```

benutzt in der folgenden Art und Weise:

```
-->c=0:0.1:1; x= ones(c);
-->y = test(x);

 c =

!  0.  0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1. !

 c =

!  0.  0. !
!  0.  0. !
!  0.  0. !
!  0.  0. !
!  0.  2. !

-->c       // c dürfte nicht verändert worden sein

 c =

!  0.  0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1. !

-->clear c // Löschen der globalen Variable c

-->y = test(x);

 !--error 4 undefined  variable : c at  line 2 of function test  called by : y = test(x);
```

Schließlich werden bei der Rückkehr aus einer Funktion alle internen Variablen (d.h. zu der Funktion gehörig) gelöscht.

Fazit:

1. keine globalen Variablen[7] benutzen;

[7] In bestimmten Fällen ist es einfacher, globale Variablen zu benutzen, und die letzte Scilab–Version (2.5) enthält wichtige Verbesserungen, um damit umzugehen: bezüglich der Details vgl. `global` unter `Help`.

2. keine Eingabeparameter verändern, außer denen, die auch Asugabeparameter sind;

3.4.2 Debuggen einer Funktion

Um eine Funktion zu debuggen, kann man als erstes die Funktion `disp(v1,v2, ...)` benutzen, die ermöglicht es, 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`).

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[8]. 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
  end
  if gefunden then
     ind = i-1
  else
     ind = 0
  end
endfunction
```

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

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[9] 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 zu finden, 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)
```

Die Funktion `warning`

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

[9] 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))`.

```
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...")
  end
  [nl,nc]=size(n)
  if (nl ~= 1) | (nc ~= 1) then
    error("Fehler in fact4 : das Argument darf keine Matrix sein...")
  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`

Sie gestattet, die Anzahl der aktuellen Eingabe– und Ausgabeargumente einer Funktion bei deren Aufruf zu erhalten. Man benutzt sie in der Form :

```
[lhs,rhs] = argn(0)
```

`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).

3.5 Diverse Ergänzungen

3.5.1 Länge eines Bezeichners

Scilab berücksichtigt nur die 24 ersten Buchstaben eines jeden Bezeichners:

```
-->a2345678901234567890123455 = 1
 a234567890123456789012345  =

    1.
-->a2345678901234567890123456 = 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

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

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}.

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))

function ZeichneFunktion(a, b, funktion, n)
  // n ist ein optionales Argument; im Fall, dass es fehlt, wird n=61 gesetzt
  [lhs, rhs] = argn(0)
  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:

```
-->ZeichneFunktion(-2*%pi,2*%pi,f)
-->ZeichneFunktion(-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: mit der klassischen Methode muss man daran denken, die Datei, welche die Funktion enthält, nach jeder Änderung mit `getf` erneut zu laden. Obwohl sich diese Operation automatisieren liesse, indem man die Anweisung `getf` ins Skript schriebe, denke ich, dass die Methode mit `deff` besser geeignet ist, wenn der Code der Funktion(en) kurz ist: Alles ist in ein und derselben Datei (ich denke hier an kleine Anwendungen).

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.

[11] siehe den Abschnitt „Scilab–Grundbefehle und –Funktionen" im Kapitel „Fallstricke"

[12] das bis auf seinen pädagogischen Wert nutzlos ist: vgl. `fplot2d`

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:

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

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

-->A
 A  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !
```

3.5.7 Lesen und Schreiben von Dateien

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:

[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`

– `"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).

- `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
```

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

(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
   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)).

3.5.8 Hinweise zur effizienten Programmierung in Scilab

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

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

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(0,1,600)';
-->timer(); A = vandm1(t,100); timer()
 ans  =

    5.12

-->timer(); A = vandm2(t,100); timer()
 ans  =

    0.94
```

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 verbessert so die Schnelligkeit in signifikanter Weise:

```
-->timer(); A = vandm3(t,100); timer()
 ans  =

    0.04

-->timer(); A = vandm4(t,100); timer()
 ans  =

    0.01
```

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):

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

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

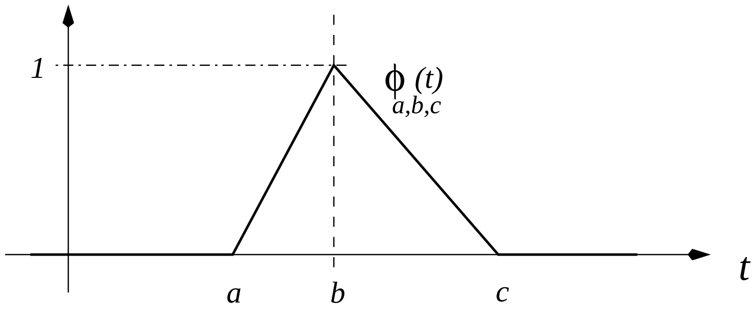


Abbildung 3.2: Die Hut–Funktion

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(100000,1)-0.5;

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

    2.65
```

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) )
  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  =

    0.21

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

    0.22
```

Bemerkung: Für diese Art von Berechnung ist es natürlicher und einfacher, die Funktion `find` zu benutzen: 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. !
```

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

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 Ausgabearguments Vektoren von Zeilen– und Spaltenindizes zu erhalten:

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

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

Nun ein Code für die ϕ–Funktion, der `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

-->timer(); y4 = phi4(t,a,b,c); timer()
 ans  =

    0.17
```

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.

3.6 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 $x(t_1), x(t_2), \dots, x(t_n)$ 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 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.

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

[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

erlaubt der Satz von [a1,a2,...]=xget('name')–Anweisungen verschiedene Parameter des Graphikkontextes zu bestimmen.

4.2 Ebene Kurven

4.2.1 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")  // fügt einen Titel hinzu
```

Noch geht alles gut, denn Scilab wählt dieselbe Skalierung; wenn man aber so fortfährt:

```
-->yter = 2*y;

-->plot2d(x,yter)
```

stellt man fest, dass Scilab die Skalierung verändert (vgl. Skalierung der Ordinaten) und dass diese neue Kurve mit der ersten verschmiltzt (es ist jedoch möglich, die alte Skalierung zu behalten...). Nun werden drei Kurven gleichzeitig gezeichnet:

```
-->xbasc() // für's Löschen

-->plot2d([x x x],[y ybis yter]) // Zusammenfügen von Matrizen ...

-->xtitle("Kurven","x","y") // ein Titel und die Beschriftung der beiden Achsen
```

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

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). Gehen wir nun zur allgemeinen Syntax über:

```
plot2d(Mx,My,[style,strf,leg,rect,nax])
```

wobei die optionalen Argumente folgende Bedeutung haben:

- style ist ein Zeilenvektor der Dimension nc, seine i–te Komponente spezifiziert die Art der i–ten Kurve. Ist style(i) eine positive ganze Zahl k, so wird auf einem Farbmonitor die entsprechende Kurve in der Farbe mit der Nummer k gezeichnet (für einige Farben der Farbpalette vgl. Tabelle 4.1, xset() zeigt Ihnen alle Farben), während man auf einem Schwarzweißmonitor verschiedene Strichmuster[4] sehen wird. Ist dagegen style(i) eine negative Zahl oder Null, so wird jeder Punkt der Kurve durch ein kleines Symbol (Punkt, Dreieck, Kreis, Kreuz, Pluszeichen usw.) dargestellt (die Punkte sind nicht miteinander verbunden); die komplette Liste der möglichen Symbole finden Sie in der Abb. 4.2.

[4] Auf einem Farbmonitor erlaubt die Anweisung xset('use color',0) in einen Schwarzweißmodus zu wechseln und xset('use color',1) in den Farbmodus zurückzukehren

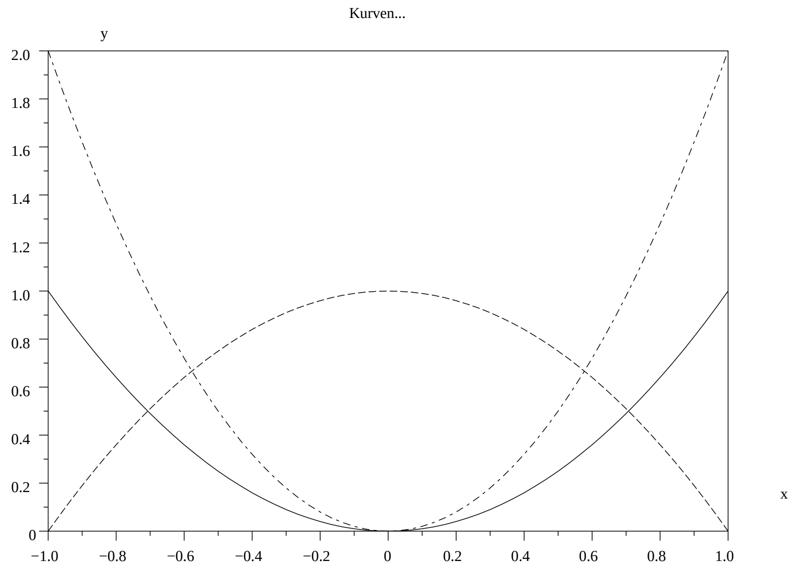


Abbildung 4.1: Die Funktionen x^2, $1 - x^2$ et $2x^2$

- `strf` ist eine Zeichenkette aus drei Zeichen `"xyz"`, so dass:
  - für `x = 1` eine Legende für jede Kurve angegeben wird (gegeben durch `leg`, das in der Liste der Argumente vorhanden sein muss); für alle anderen Werte von `x` erscheint keine Legende;
  - der Buchstabe `y` die Skalierung spezifiziert (d.h. das Rechteck $x_{min}, y_{min}, x_{max}, y_{max}$, welches die Begrenzung des Graphen festlegt):
    * für `y = 0` wird diejenige Skalierung verwendet, die durch den vorigen Aufruf festgelegt wurde;
    * für `y = 1` wird eine Skalierung durch das Arguments `rect` festgelegt;
    * für `y = 2` berechnet Scilab automatisch die Skalierung (indem er die Maxima und Minima von `Mx` und `My` verwendet);
    * mit `y = 3` und `y = 4` kann man eine isometrische Skalierung erhalten (siehe weiter unten);
    * für alle anderen Möglichkeiten siehe `Help`
  - das Zeichen `z` erlaubt die Umgebung des Graphen einzustellen:
    * für `z = 1` ist der Graph von einem Rahmen (dessen Dimensionen $x_{min}, y_{min}, x_{max}, y_{max}$ sind) umgeben, dessen West– (y–Achse) und Südseite (x–Achse) Unterteilungen besitzen (diese werden durch den Parameter `nax` spezifiziert);
    * für `z = 2` ist der Graph einfach von einem Rahmen ($x_{min}, y_{min}, x_{max}, y_{max}$) umgeben;
    * für alle anderen Werte gibt es weder einen Rahmen noch Achsen.
- Der Parameter `leg` ist eine Zeichenkette `'y1@y2@...'`, die eine Legende (vgl. `x = 1`) für jede Kurve liefert, wobei jede Legende von der folgenden durch das Zeichen `@` getrennt wird. Geben Sie n Legenden an, obwohl es nc Kurven gibt ($n < nc$), so werden lediglich die ersten n Kurven mit einer Legende versehen;
- `rect = [xmin,ymin,xmax,ymax]` dient zur Festlegung der Skalierung (vgl. `y = 1`);
- `nax = [nx,Nx,ny,Ny]` dient zur Auswahl der Unterteilung (vgl. `z = 1`); wobei `Nx` die Anzahl der Intervalle auf der x–Achse (man erhält also `Nx+1` nummerische Werte), `nx` die Anzahl der Subintervalle pro Intervall darstellt (so ergibt der Wert von `nx = 2` eine kleine Unterteilung zwischen zwei großen).

Achtung: Falls Sie das i–te Argument brauchen, müssen Sie zwingend alle vorigen Argumente angeben. Auf diese Weise müssen Sie, wenn Sie z.B. Unterteilungen (mit `z = 1` und `nax`) auswählen, alle vorigen Parameter angeben (evtl. mit Dummy–Werten, falls diese nicht gebraucht werden. z.B. `leg = ' '`, `rect = [1:4]`).

Nummer	Farbe
1	schwarz
2	dunkelblau
3	hellgrün
4	hellblau
5	rot
6	lila
26	(kastanien)braun
29	rosa
32	gelborange

Tabelle 4.1: Liste einiger StandardFarben

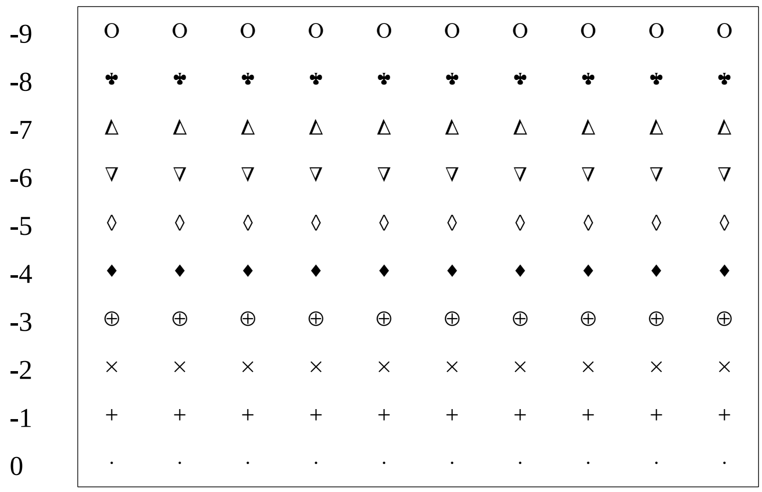


Abbildung 4.2: Die verschiedenen Symbole

Hier ein erstes Beispiel mit fast allen Parametern (vgl. Abbildung 4.3):

```
-->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

-->rect = [-2.05,-1.4,2.05,1.4];          // das Fenster

-->leg="Ellipse@Kreis@Fehler Fkt";        // die Legenden

-->plot2d([x1 x2 x3],[y1 y2 y3],[1:3],"111",leg,rect)

-->xtitle("noch mehr Kurven ...","x","y")
```

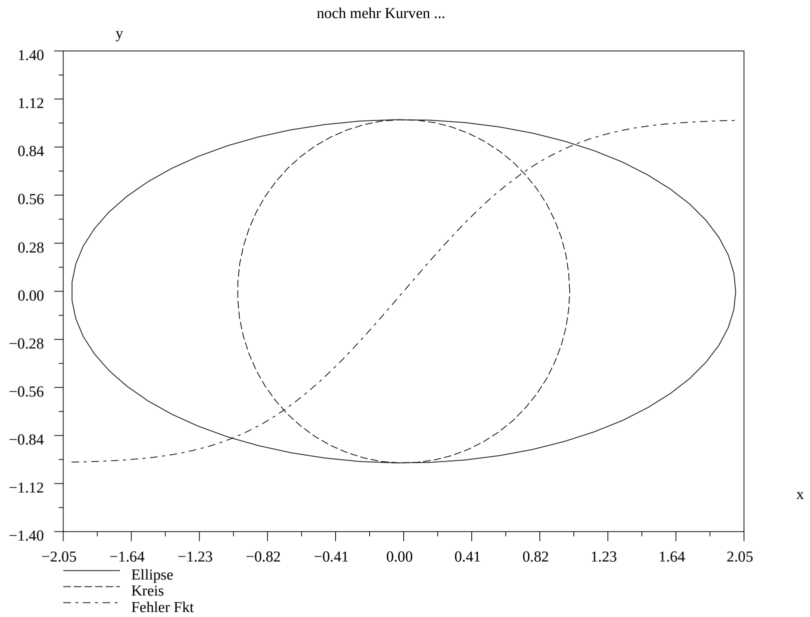


Abbildung 4.3: Ellipse, Kreis und Fehlerfunktion

Im Allgemeinen verwendet man die Anweisung `plot2d` folgendermaßen (wobei `nc` die Anzahl der Kurven ist):

```
plot2d(Mx,My,[1:nc],"121","leg_1@....@leg_nc")
```

Sie erlaubt jede der unterschiedlichen Kurven mit Hilfe einer Legende ausfindig zu machen; für alle wird die Skalierung automatisch berechnet[5]. Es ist u.U. vorteilhafter, die Skalierung mit einem selbst berechneten Vektor `rect` mit `y = 1` einzustellen:

- sei es, um es schöner erscheinen zu lassen, denn die Kurven passen sich automatisch der Dimension des Fensters an; man kann diees folgendermaßen berechnen:

```
x0 = min(Mx); x1 = max(Mx);
y0 = min(My); y1 = max(My);
dx = (x1 - x0)*0.05; dy= (y1 - y0)*0.05;
// und schließlich:
xmin = x0 - dx; xmax = x1 + dx; ymin  = y0 - dy; ymax = y1 + dy;
```

- oder wenn man eine Reihe von Kurven zeichnen will, wobei die i–te Kurve erst am Ende des i–ten Rechenschrittes vorliegt (jede Kurve wird angezeigt, sobald sie verfügbar ist, anstatt das Ende der Berechnungen abzuwarten). Hier ein solches Beispiel:

```
x = linspace(-1,1,101)';
rect = [-1,-1.05,1,1.05];   // Zeichenbereich
t=linspace(2,-2,7)
y = t(1)*x.^2;              // eine erste Berechnung
xbasc()                     // Löschen des aktiven Graphikfensters
xselect()                   // aktives Fenster wird in den Vordergrund gebracht
xtitle("Kurven...","x","y")
plot2d(x,y,1,"011"," ",rect)
// Anzeige
for i=2:7
```

[5]die einfache Anweisung `plot2d(Mx,My)` entspricht `plot2d(Mx,My,[1:nc],'021')`

```
  y = t(i)*x.^2;              // i-te Berechnung
  xpause(2000)                // zur Simulation einer längeren Berechnung
  plot2d(x,y,i,"000")         // man verwendet die bisherige Skalierung (Zeichenbereich)
end
```

Um längere Berechnungen zu simulieren, verwendet man die Funktion `xpause(dt)`, die die Anzeige für die Zeit von ungefähr `dt` ms erstarren lässt (d.h. dass diese Funktion sehr abhängig vom System ist und nicht immer funktioniert).

4.2.2 Zeichnen von mehreren Kurven, die unterschiedlich viele Punkte haben

Man kann mit `plot2d` nicht mehrere Kurven zeichnen, die nicht mit der gleichen Anzahl von Intervallen diskretisiert worden sind. Man ist also gezwungen, erst die Skalierung zu berechnen[6], dann `plot2d` ein erstes Mal aufzurufen, um die Skalierung (mit `y = 1`) festzulegen, alle anderen Aufrufe müssen die vorige Skalierung `y = 0` verwenden. Hier ein Beispiel in Form eines Skripts (vgl. Abbildung 4.4):

```
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(min(y1),min(y2),min(y3));
ymax = max(max(y1),max(y2),max(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 den Zeichenbereich bestimmt
plot2d(x2,y2,2,"000")             // zweiter Aufruf und
plot2d(x3,y3,-1,"000")            // dritter Aufruf: mit vorheriger Skalierung
xtitle("Kurven...","x","y")
```

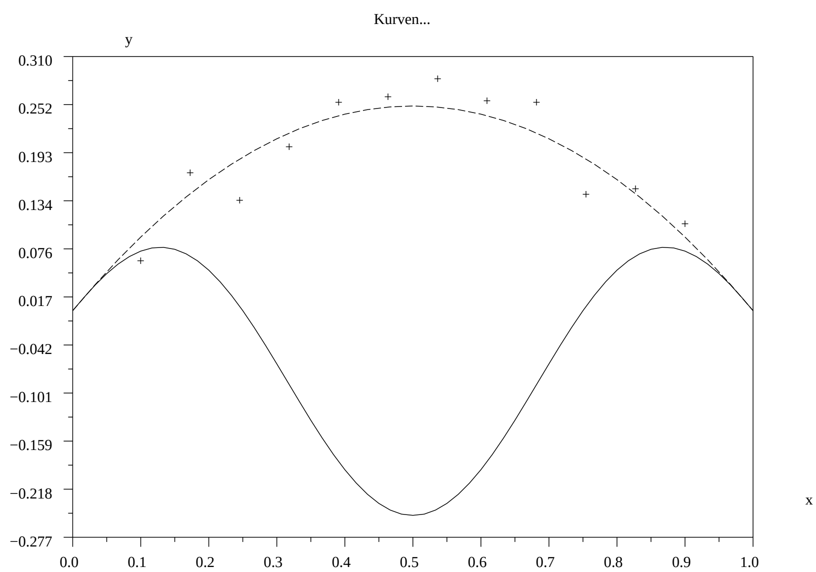


Abbildung 4.4: noch mehr Kurven...

[6] oder aber man schickt als erstes eine Kurve voraus, welche die Skalierung bestimmt (`y = 2`)

Man kann hier also nicht jede Kurve mit einer Legende versehen, aber man kann dies mit elementaren Graphikfunktionen erreichen. Um die Graphiken zu ergänzen, bietet es sich an, diese im Format `fig` zu exportieren, und dann mit dem Graphikprogramms `xfig` weiterzuverarbeiten (vgl. Abspeichern von Graphiken... ein wenig weiter unten).

4.2.3 Zeichnen unter Verwendung einer isometrischen Skalierung

In der Regel werden die Graphiken nicht mit einer isometrischen Skalierung gezeichnet. Wenn Sie also versuchen, einen Kreis zu zeichnen, wird er i.A. eher einer Ellipse ähneln. Indem die Größe des Graphikfensters verändert wird, lässt sich dieses Problem in den Griff kriegen, die Ausgaben in Postscript und anderen Formaten verwenden jedoch die Anfangsskalierung. Es gibt mehrere Möglichkeiten, eine isometrische Skalierung zu erhalten:

1. beim ersten Aufruf von `plot2d` muss der Parameter `y` der Zeichenkette `strf` gleich 3 sein, und man gibt den Zeichenbereich mit dem Argument `rect` an, oder man wählt `y = 4`, damit der Zeichenbereich automatisch berechnet wird (ausgehend von den Minima und Maxima der Punkte); bei den darauffolgenden Aufrufen von `plot2d` muss `y` gleich 0 sein;

2. vor dem Aufruf von `plot2d` wird die Skalierung mit Hilfe der Anweisung `isoview` festgelegt:

   ```
   isoview(xmin, xmax, ymin, ymax)
   ```

   und bei(m) Aufruf(en) von `plot2d` muss der Parameter `y` (der Zeichenkette `strf`) obligatorisch auf 0 gesetzt werden.

Diese beiden Methoden können zusammen mit der Mehrzahl der Grundbefehle für 2D–Zeichnungen verwendet werden. Die isometrische Skalierung wird sowohl beim Verändern des Fensters als auch beim Vergrößern mit dem interaktiven Zoom beibehalten. Hier ein Beispiel mit `isoview` (vgl. Abb. 4.5) :

```
// ein Beispiel für das Benutzen von isoview
// 1.) Daten für das Zeichnen eines Kreises
t = linspace(0,2*%pi,100)';
x = cos(t) ; y = sin(t) ;
// 2.) Daten für das Zeichnen von Punkten "in der Nähe" des Kreises
tt = linspace(0,2*%pi,40)' ;
xx = cos(tt) + 0.2*(rand(tt) - 0.5);
yy = sin(tt) + 0.2*(rand(tt) - 0.5);
// 3.) und los geht's mit der Zeichnung
xbasc()                         // löscht (gegebenenfalls)
isoview(-1.2, 1.2, -1.2, 1.2)   // Einstellen der Skalierung
xset("font",3,1)                // wählt times-italic in 10 pt
plot2d(x,y,2,"001")             // Bild des Kreises
plot2d(xx,yy,-9,"000")          // Bild der Punkte
xset("font",4,2)                // wählt times-bold in 12 pt
xtitle("Beispiel zu isoview: versuchen Sie, das Fenster zu verändern","x","y")
xselect()                       // holt aktuelles Fenster in den Vordergrund
```

Mit der ersten Methode erhält man statt dessen:

```
// 3.) und los geht's mit der Zeichnung
xbasc()                         // löscht (gegebenenfalls)
xset("font",3,1)                // wählt times-italic in 10 pt
plot2d(x,y,2,"031"," ",[-1.2, -1.2, 1.2, 1.2])  // Bild des Kreises
plot2d(xx,yy,-9,"000")          // Bild der Punkte
xset("font",4,2)                // wählt times-bold in 12 pt
xtitle("Beispiel zu isoview: versuchen Sie, das Fenster zu verändern","x","y")
xselect()                       // holt aktuelles Fenster in den Vordergrund
```

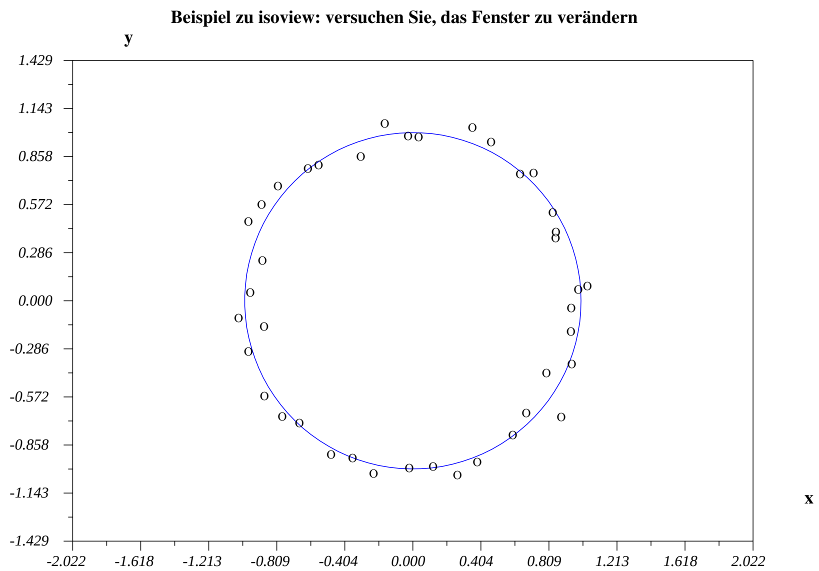


Abbildung 4.5: Ein Kreis mit Datenpunkten...

4.3 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 rund 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.

4.4 Einfache Animationen

Es ist leicht, kleine Animationen mit Scilab zu realisieren, das die Benutzung von Doppelpuffern (double buffer) erlaubt; diese verhindern ein Flimmern und Überlagern von Teilbildern. Außerdem gibt es verschiedene Treiber, die die Anzeige am Bildschirm ermöglichen[7]; der eine (`Rec`) speichert alle im Fenster vorgenommenen Graphikoperationen und ist der Standardtreiber, der andere (`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). Wenn Sie dagegen in `Rec` bleiben, erlaubt Ihnen eine Redimensionierung des Graphikfensters, Ihre Animation nochmals zu sehen. Es folgt ein kleines Skript, welches den Vorteil des Doppelpuffers demonstriert. Man zeigt zu verschiedenen Zeitpunkten (diskretisiert...) die folgende Funktion an:

$$y(x,t) = \cos(t - 2x),\ x \in [0, 2\pi],\quad \text{und } t \in [0, 6\pi].$$

Um von einem Bild in das nächste zu wechseln, löscht man einen Teil des Fensters mit `xclear`. Sie können dies vereinfachen, indem Sie `xset("wwpc")` verwenden, das den aktuellen Puffer komplett löscht (man muss allerdings jedesmal den Titel, die Beschriftungen und Achsen neu zeichnen). Für subtilere Animationen müssen Sie mit Masken arbeiten, indem Sie `xset('alufunction',num)` benutzen, wobei `num` eine ganze Zahl ist, die die Anzeigefunktion bestimmt, vgl. `Help` und die Demos... dieser Teil der Dokumentation muss noch vervollständigt werden!

Hoffentlich ist dieses Skript ausreichend kommentiert, so dass Sie es verstehen können (es ist auf jeden Fall eine gute Übung).

```
// Script zum Testen einer einfachen Animation
```

[7]zuzüglich der Treiber, die erlauben, Zeichnungen in Postscript und in `fig` anzufertigen

```
// 0) verschiedene Initialisierungen

nt =  201;  nx =  101;
T = linspace(0,6*%pi,nt);  // Zeitpunkte
x = linspace(0,2*%pi,nx)'; // Abszissen
rect  = [-0.05,-1.05,2*%pi+0.05,1.05]; // Ausschnitt
xselect() //  holt aktuelles Fenster in den Vordergrund
xbasc()   //  löscht das Fenster

// 1) 1. Methode (schlecht): ohne Benutzung eines Doppelpuffers

// titlepage zeigt eine Überschrift in der Mitte des Graphikfensters
titlepage(["Animationsversuche";"  ";"1. Methode   :";"direkte Anzeige"])
xtitle("(klicken Sie auf das Fenster zum Start der Animation)")
[c_i,c_x,c_y,c_w]=xclick(); // wartet auf den Mausklick
xbasc() //  löscht die Titelseite
xtitle("Eine fortschreitende Welle","x","y")
y =  cos(T(1) - 2*x);
plot2d(x,y,1,"112","y(x,t)=cos(t-2x)",rect)
for i=2:nt
   xpause(10);
   xclea(-0.02,1.02,2*%pi+0.04,2.04) // löscht das Rechteck
   y = cos(T(i) - 2*x);
   plot2d(x,y,1,"000")  // nur Anzeige der Kurve (weder Rahmen noch
end                     // Legende, aber weil man diese nicht gelöscht hat...)

// 2) 2. Methode (besser): Benutzung eines Doppelpuffers

xbasc()
titlepage(["Animationsversuche";" ";"2. Methode   :";"mit Doppelpuffer"])
xtitle("(klicken Sie auf das Fenster zum Start der Animation)")
[c_i,c_x,c_y,c_w]=xclick();
xbasc() // löscht die 2. Titelseite
xset("pixmap",1)  // Benutzung eines Doppelpuffers (die Graphik erscheint nicht direkt
                  // auf dem Bildschirm sondern wird in den Videopuffer (pixmap) geschrieben
xtitle("Eine fortschreitende Welle","x","y")
y = cos(T(1) - 2*x);
plot2d(x,y,1,"112","y(x,t)=cos(t-2x)",rect) // geht in den verdeckten Puffer
xset("wshow")                               // anzeigen: vertausche die beiden Puffer
for i=2:nt
   xpause(10);
   xclea(-0.02,1.02,2*%pi+0.04,2.04)
   y = cos(T(i) - 2*x);
   plot2d(x,y,1,"000")  // geht in den verdeckten Puffer
   xset("wshow")        // anzeigen: vertausche die beiden Puffer
end
xset("pixmap",0) // Rückkehr zum normalen Modus, bei dem die
                 // Graphik direkt auf dem Bildschirm angezeigt wird
```

4.5 Flächen

`plot3d` ist die Anweisung zum Zeichnen von Flächen. `plot3d1` wird beinahe auf identische Art und Weise verwendet und erlaubt, Farben entsprechend dem Wert von z zu verwenden. Was die Darstellung Ihrer Fläche durch Facetten betrifft, erlauben diese beiden Anweisungen, jede einzelne Facette in einer anderen Farbe zu zeichnen.

4.5.1 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 auch in y: dieselbe)
```

```
z=cos(x)'*cos(x);  // z-Werte: eine Matrix z(i,j) = f(x(i),y(j))

plot3d(x,x,z)  // das Bild
```

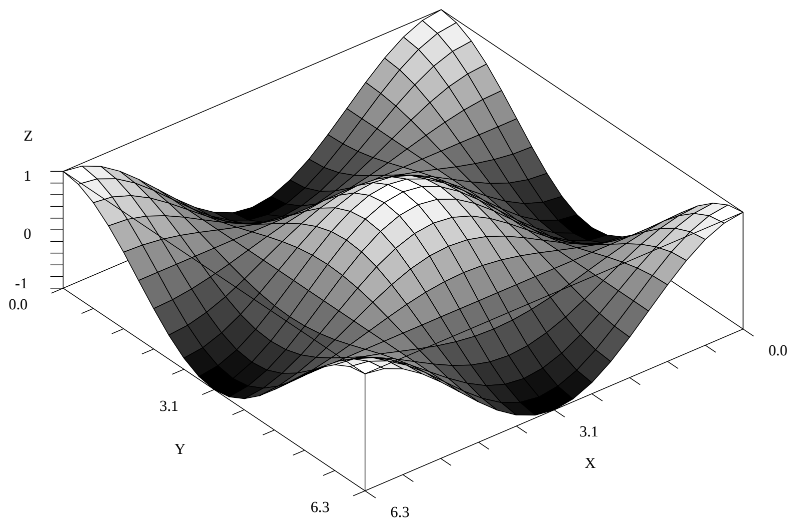


Abbildung 4.6: die Funktion $z = cos(x)cos(y)$

Sie werden etwas erhalten, was der Abbildung 4.6 ähnelt[8]. Hier wurde `plot3d` in seiner einfachsten Syntax verwendet, aber Sie können die Blickrichtung und eine Reihe anderer Parameter näher bestimmen:

```
plot3d(x,y,z [,theta,alpha,leg [,flag,ebox]])
plot3d1(x,y,z [,theta,alpha,leg [,flag,ebox]])
```

wobei gilt:

1. `x` und `y` sind zwei Zeilenvektoren $((1, nx)$ und $(1, ny))$, die der Diskretisierung in x und in y entsprechen;

2. `z` ist eine Matrix $((nx, ny))$, so dass `z(i,j)` die Höhe im Punkt `(x(i),y(j))` angibt;

3. `theta` und `alpha` sind zwei Winkel (in Grad), die die Blickrichtung festlegen;

4. `leg` ist eine Zeichenkette, die für die Legende aller Achsen zuständig ist (z. B. `leg="x@y@z"`);

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

   (a) der Parameter `mode` darüber entscheidet, ob die verdeckten Flächen gezeichnet werden oder nicht: ist `mode > 0`, dann werden die verdeckten Flächen entfernt, ist `mode = 0`, werden sie gezeichnet, und ist `mode < 0`, so wird nur der Schatten der Fläche in der Farbe (Schraffierung) `-mode` gezeichnet(ab der Version 2.3.1);

   (b) ist `type = 0`, wird die vorige Skalierung verwendet, ist `type = 1`, dann bestimmt man die Skalierung näher mit:

   `ebox = [xmin,xmax,ymin,ymax,zmin,zmax]`

   andernfalls wird die Skalierung für alle anderen Werte mit Hilfe der Daten automatisch berechnet;

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

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

- box = 0 ringsherum wird nichts gezeichnet,
- box = 2 die Achsen werden gezeichnet,
- box = 3 ein Kasten umgibt die Fläche...,
- box = 4 ein Kasten und die Achsen...

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:

```
x=linspace(-%pi,%pi,31);
z=sin(x)'*sin(x);
n = 21
theta = linspace(20,80,n);
xbasc(); xselect()
xset("pixmap",1)   // Aktivierung des Doppelpuffers
alpha = linspace(60,30,n);
plot3d(x,x,z,theta(1),alpha(1),"x@y@z",[2 2 4])
xtitle("Variation der Blickrichtung mit dem Parameter theta")
xset("wshow")
// variiere theta
for i=2:n
  xset("wwpc") // löscht den aktuellen Puffer
  plot3d(x,x,z,theta(i),alpha(1),"x@y@z",[2 0 4])
  xtitle("Variation der Blickrichtung mit dem Parameter theta")
  xset("wshow")
end
// variiere alpha
for i=2:n
  xset("wwpc") // löscht den aktuellen Puffer
  plot3d(x,x,z,theta(n),alpha(i),"x@y@z",[2 0 4])
  xtitle("Variation der Blickrichtung mit dem Parameter alpha")
  xset("wshow")
end
xset("pixmap",0)  // Rückkehr zum normalen Modus
```

4.5.2 Farbgebung

Sie können die beiden vorigen Beispiele nocheinmal 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 solche Farbpalette liefert die Funktion hotcolormap ab der Version 2.4. Eine Farbpalette ist eine Matrix der Dimension (anz_Farben,3), die i–te Zeile entspricht der Intensität (zwischen 0 und 1) von Rot, Grün und Blau der i–ten Farbe. Wenn eine derartige Matrix — die wir C nennen — gegeben ist, ermöglicht die Anweisung xset("colormap",C), sie in den Graphikkontext des aktuellen Graphikfensters aufzunehmen. 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). Die folgende kleine Funktion installiert automatisch eine klassische Farbpalette. Für die Anzahl der Farben werden Vielfache von Acht verwendet; ist dies nicht der Fall, weiß sie sich zu helfen. Außerdem können Sie sie ohne Parameter aufrufen — dann wird eine Farbpalette mit 64 Farben installiert.

```
function Farbpalette(anz_Farben)
  // installiert eine klassische Farbpalette mit anz_Farben, falls
```

```
  // anz_Farben ein Vielfaches von 8 ist. Falls nicht, so nimmt man das
  // nächstgrößere Vielfache von 8. Falls das Argument anz_Farben fehlt,
  // so setzt man anz_Farben=64; man begrenzt anz_Farben zudem auf den
  // Bereich  8 bis 128
  [lhs,rhs]=argn(0);
  if rhs == 0 then
    n = 8 ; anz_Farben = 64
  else
    n = ceil(anz_Farben/8)
    if (n < 1) then, n = 1, end      // mindestens   8 Farben
    if (n > 16) then, n = 16, end    // höchstens  128 Farben
    if (8*n ~= anz_Farben) then
       anz_Farben = 8*n
       warning("Die Palette wird " + sprintf("%d",anz_Farben) + " Farben haben")
    end
  end
  rgb = zeros(anz_Farben,3)
  // Hilfsvektor :
  sequence = [(1:2*n)'/(2*n) ; ones(2*n,1); (2*n:-1:1)'/(2*n)]
  rgb(3*n+1:$,1) = sequence(1:5*n)
  rgb(n+1:7*n,2) = sequence
  rgb(1:5*n  ,3) = sequence(n+1:$)
  // jetzt nur noch diese Farbpalette im Graphikkontext installieren
  xset("colormap",rgb)
endfunction
```

4.5.3 plot3d und plot3d1 mit Facetten

Um diese Funktionen in einem allgemeineren Kontext zu benutzen, muss eine Beschreibung Ihrer Fläche mittels Facetten erfolgen. Diese wird dargestellt durch 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 [,theta,alpha,leg [,flag,ebox]])
plot3d1(xf,yf,zf [,theta,alpha,leg [,flag,ebox]])
```

Wenn sie außerdem Farben haben wollen, müssen Sie die Reihenfolge der Ecken einer Facette nach der Rechte–Hand–Regel in Richtung der inneren Normale angeben. Und wenn Sie die Farben für jede Facette vorgeben wollen, muss das dritte Argument eine Liste sein: `list(zf,colors)`, wobei `colors` ein Vektor der Größe `anz_Seiten` ist, `colors(i)` gibt die Nummer der Farbe (gemäß der Palette) für die i–te Facette an.

Greifen wir nun auf den Würfel aus dem Abschnitt über typisierte Listen zurück. Die gegebene Beschreibung entspricht nicht der von `plot3d` erwarteten, aber mit Hilfe einer kleinen Funktion erhält man diese ohne Probleme:

```
function [xf,yf,zf] = trans_polyeder(Polyeder)
  // diese Funktion berechnet eine Beschreibung mittels Facetten
  // aus der klassischen Beschreibung, da diese von plot3d erwartet wird
  [ anz_Seiten, anz_Ecken_pro_Seite ] = size(Polyeder("Seite"))
  xf=zeros(anz_Ecken_pro_Seite, anz_Seiten); yf=zeros(xf); zf=zeros(xf)
  for i=1:anz_Seiten
     for j=1:anz_Ecken_pro_Seite
        Nr_der_Ecke = Polyeder("Seite")(i,anz_Ecken_pro_Seite+1-j)
            // Umkehr der Orientierung
        v = Polyeder("Koord")(:,Nr_der_Ecke)
```

```
      xf(j,i)=v(1); yf(j,i)=v(2); zf(j,i)=v(3)
    end
  end
endfunction
```

und los geht's:

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

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

Cube = tlist(["Polyeder","Koord","Seite"],P,Ecken);
[xf,yf,zf] = trans_polyeder(Cube);
Farben = 2:7;   // Benutzung der Standard-Farbpalette
xbasc(); xselect()
// die kleine Zeichnung:
plot3d(xf,yf,list(zf,Farben),30,60,"x@y@z",[2 1 4],[-0.2,1.2,-0.2,1.2,-0.2,1.2])
```

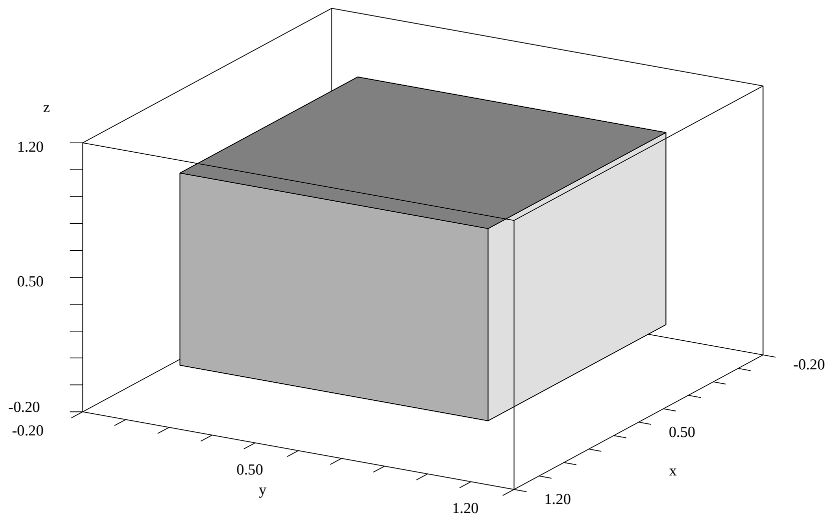


Abbildung 4.7: Ein Würfel

In dieser Anwendung sind die Befehle `plot3d` und `plot3d1` äquivalent. Für einen Polyeder dagegen, bei dem man die Farben gemäß der Höhe haben möchte (ohne sie aber zu berechnen), erledigt `plot3d1` dies automatisch. Letzte Bemerkung: Sind Ihre Facetten dreieckig, müssen Sie so tun, als ob sie viereckig wären. Seien also P_1, P_2, P_3 die drei Ecken einer solchen Facette, so muss diese für Scilab durch vier Ecken P_1, P_2, P_3, P_1 beschrieben werden (dieses Problem ist in der Version 2.5 korrigiert worden).

4.5.4 Zeichnen einer durch $x = f_1(u, v)$, $y = f_2(u, v)$, $z = f_3(u, v)$ definierten Fläche

Einfache Antwort: Nehmen Sie eine Diskretisierung des Parametergebiets und berechnen die Facetten! Es gibt zwar eine Funktion (`eval3dp`), die dies automatisch tut. Aus Effizienzgründen jedoch sollte eine Funktion, die die Parametrisierung Ihrer Fläche definiert, in Form eines Vektors geschrieben sein. Sind $U = (u_1, u_2, \ldots, u_{n_1})$ und $V = (v_1, v_2, \ldots, v_{n_2})$ die Diskretisierungen eines Rechtecks des Parametergebietes, wird Ihre Funktion ein einziges Mal mit zwei „großen" Vektoren der Länge $n_1 n_2$ aufgerufen:

```
Ug = (u1,u2,...,un1, u1,u2,...,un1, ......)
```

```
      |----------| : diese Sequenz wird n2-mal wiederholt

Vg = (v1,v1,.......,v1, v2,v2,.......,v2, ....., vn2 vn2. ... vn2)
      |--n1-mal  v1--|  |--n1-mal  v2--|         |--n1-mal vn2--|
```

Das folgende Beispiel wird Ihnen helfen, die Vektorisierung klassischer Flächen zu verstehen:

```
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 Moebius-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
  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 benutzt:

```
// 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()
```

Wollen Sie Farben verwenden und erhalten sie nicht, ist die Orientierung nicht die richtige: es reicht dann die Orientierung von einem der beiden Diskretisierungsvektoren des Parametergebietes umzukehren.

4.6 Raumkurven

`param3d` dient als Basisanweisung, wenn man eine solche Kurve zeichnen will. Hier ist ein klassisches Beispiel für eine Helix:

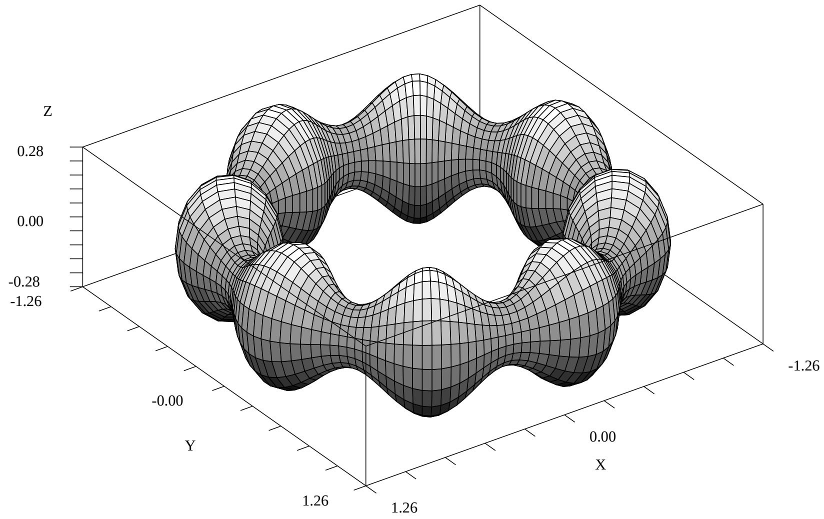


Abbildung 4.8: Ein verbeulter Torus

```
-->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, der viel leistungsfähiger ist. Hier die entsprechende Syntax:

```
param3d1(x,y,z,[theta,alpha,leg,flag,ebox])
param3d1(x,y,list(z,colors),[theta,alpha,leg,flag,ebox])
```

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`. `colors` ist 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 Schwarzweißmonitor), während man für einen Wert zwischen -9 und 0 eine Anzeige von (nicht verbundenen) Punkten erhält, die diese Kurven durch entsprechende Symbole darstellen. Hier ein Beispiel, das Sie zur Abbildung 4.9 führt:

```
-->t = linspace(0,4*%pi,100)';

-->x1 = cos(t); y1 = sin(t) ; z1 = t;  // eine Helix

-->x2 = x1 + 0.2*(1-rand(x1));

-->y2 = y1 + 0.2*(1-rand(y1));

-->z2 = z1 + 0.2*(1-rand(z1));

-->xbasc() ; param3d1([x1 x2],[y1 y2],list([z1 z2],[1 -9]))

-->xset("font",4,3)   // times 14 pt fett

-->xtitle("Schraubenlinie mit Perlen")
```

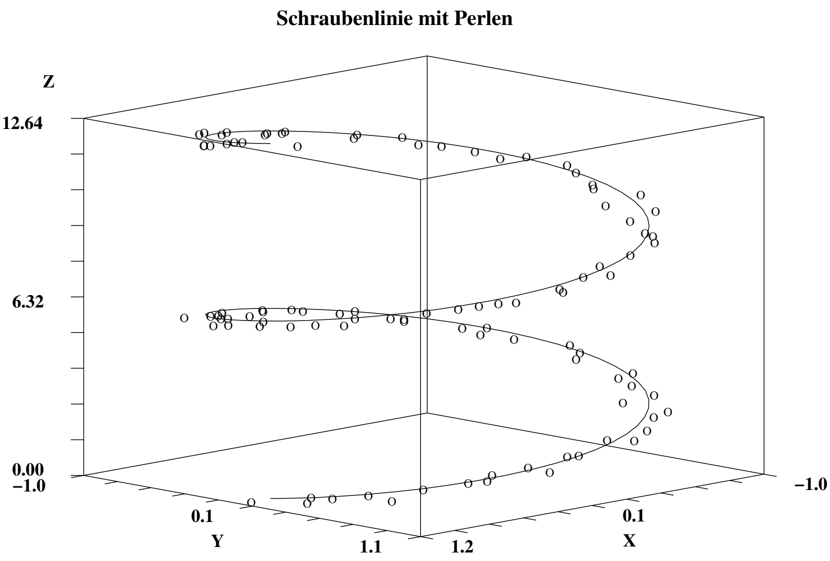


Abbildung 4.9: Kurve und Punkte im Raum

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:

```
// Skript zum Anzeigen von Punkten
n = 50;        // Anzahl der Punkte
P = rand(n,3); // Zufallszahlen
// Berechnung des umschließenden Quaders
xmin = min(P(:,1)); xmax = max(P(:,1)) ; dx = xmax - xmin;
ymin = min(P(:,2)); ymax = max(P(:,2)) ; dy = ymax - ymin;
zmin = min(P(:,3)); zmax = max(P(:,3)) ; dz = zmax - zmin;
xvisu_min = xmin - 0.05*dx ; xvisu_max = xmax +  0.05*dx ;
yvisu_min = ymin - 0.05*dy ; yvisu_max = ymax +  0.05*dy ;
zvisu_min = zmin - 0.05*dz ; zvisu_max = zmax +  0.05*dz ;
ebox = [xvisu_min xvisu_max yvisu_min yvisu_max zvisu_min zvisu_max ] ;

// Trennen der Punkte in zwei Gruppen, um zu zeigen wie die verschiedenen
// Symbole und Farben für die Punkte eingesetzt werden
m = 40;
P1 = P(1:m,:) ; P2 = P(m+1:n,:);

// das Bild
xset("window",0)
xbasc()

// erste Gruppe von Punkten
xset("pattern",2)  // blau  mit der Standardfarbpalette
param3d1(P1(:,1),P1(:,2),list(P1(:,3), -9),60,30,"x@y@z",[1 4],ebox)

// für die zweite Gruppe
xset("pattern",3)  // grün mit der Standardfarbpalette
param3d1(P2(:,1),P2(:,2),list(P2(:,3), -5),60,30," ",[0 4])
        // -5 markiert mit einer kleinen Raute
        // [0 4] ) : um mit der aktuellen Skalierung zu zeichnen
        //           (mit dem ersten Aufruf in param3d1 festgelegt)
xset("pattern",1) // um wieder schwarz als aktuelle Farbe zu erhalten
[font_init] = xget("font")
xset("font",4,3)    // um den Titel in times bold 14 pt zu erhalten...
```

```
xtitle("Punkte...")
xset("font",font_init(1), font_init(2)) // um wieder die Ausgangsschriftart zu erhalten
xselect()
```

4.7 Mehrere Graphen in einem Graphikfenster zeichnen

Zu diesem Zweck verwendet man die Funktion `xsetch`, mit der man eine rechteckige Zone des aktuellen Graphikfensters auszuwählen kann, um das Fenster in mehrere Unterfenster zu zerlegen). Bevor ein Graphikbefehl in ein Unterfenster geschickt wird, wird diese Funktion mit einem einzigen Argument[9] aufgerufen, das aus einem Vektor mit vier Komponenten besteht:

```
xsetech([xul yul dx dy])
```

1. `xul` und `yul` erlauben, die „obere linke" Ecke der Zone zu definieren, die wiederum das Unterfenster definiert;

2. `dx` und `dy` definieren die Breite und die Höhe des Rechecks.

Das gesamte Fenster erhält man mit dem Vektor `[0 0 1 1]` (die y–Achse verläuft von oben nach unten!), eine Unterteilung in zwei Unterfenster gleicher Größe geschieht folgendmaßen:

```
xsetech([0 0 1 0.5])  // Auswahl der oberen Zone
Anweisungen für das obere Bild
xsetech([0 0.5 1 0.5]) // Auswahl der unteren Zone
Anweisungen für das untere Bild
```

Hier ein kleines Skript, in dem das Graphikfenster in vier Unterfenster unterteilt wird und mit dem man die Abbildung 4.10 erhält:

```
// Skript zum Illustrieren der Benutzung von xsetech
x = linspace(0,2*%pi,40);
xset("window",0)
xbasc()
xsetech([0 0 0.5 0.5])     // Bild links oben
titlepage(["    verschiedene Darstellungen";"der Kurve z = cos(x)sin(y)"])
xsetech([0 0.5 0.5 0.5])   // Bild links unten
xtitle("Isolinien mit contour2d")
contour2d(x,x,cos(x')*sin(x),6,1:6,"011"," ",[0 0 2*%pi 2*%pi])
xsetech([0.5 0 0.5 0.5])   // Bild rechts oben
xtitle("3D Ansicht mit plot3d1")
plot3d1(x,x,cos(x')*sin(x))
xsetech([0.5 0.5 0.5 0.5]) // Bild rechts unten
xtitle("2D Ansicht in Farbe mit Sgrayplot")
Sgrayplot(x,x,cos(x')*sin(x))
```

Bemerkung: Für die Anweisungen `plot3d1` und `Sgrayplot` wird die Standardfarbpalette verwendet, aber bei der Konvertierung nach Postscript (bei Auswahl von Schwarzweiß) kann man „kontinuierliche" Graustufen erhalten.

4.8 Diverses

Es gibt noch mehr Graphik–Grundbefehle, darunter:

1. Die Varianten von plot2d, `plot2d1`, `plot2d2`,..., um stückweise konstante Flächen oder vertikale Linien und/oder mit einer logarithmische Skalierung zu zeichnen (siehe einige Beispiele im folgenden Kapitel);

[9] hier wird die vereinfachte Syntax verwendet; es gibt auch Möglichkeiten, die Skalierung einzurichten...

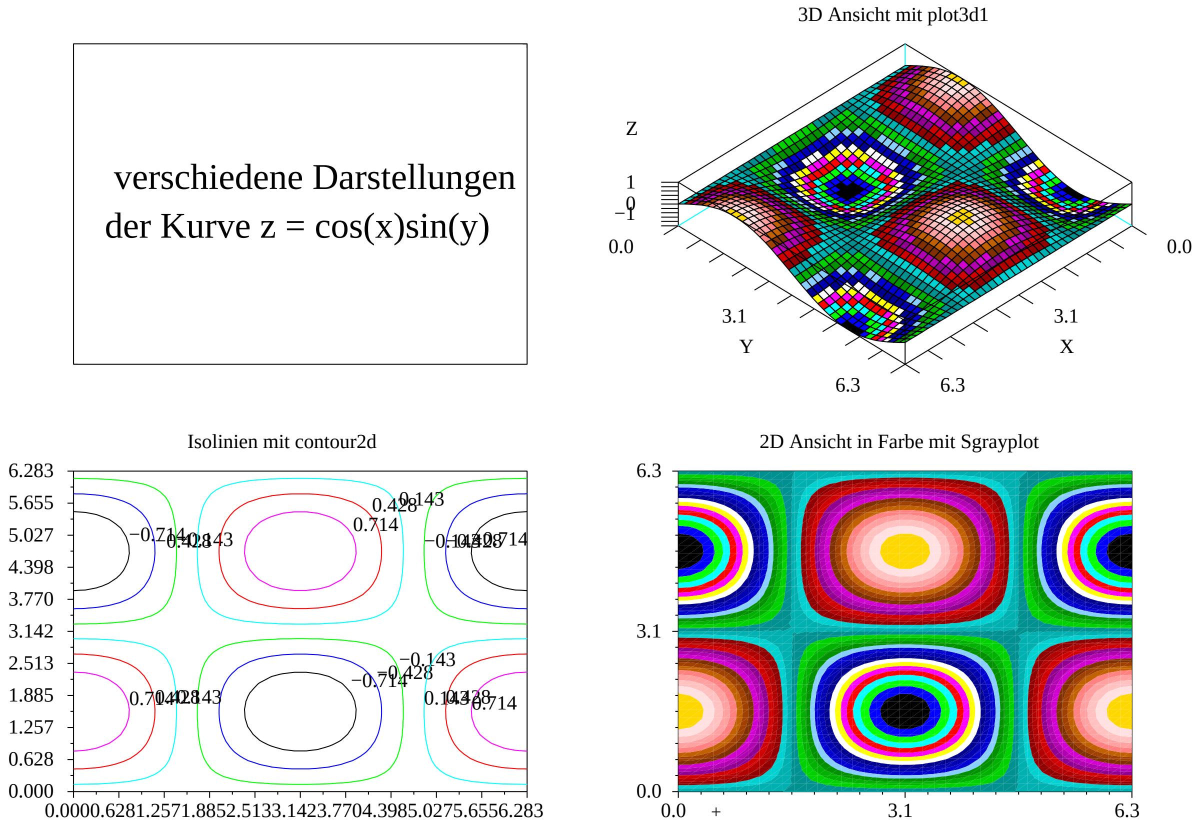


Abbildung 4.10: Abbildung mit xsetech

2. `contour2d` und `contour`, um Isolinien einer Funktion $z = f(x, y)$ zu zeichnen, die auf einem Rechteck definiert ist;

3. `grayplot` und `Sgrayplot`, die gestatten, die Werte einer solchen Funktion durch Farben darzustellen;

4. `fec` entspricht den beiden vorherigen Anweisungen für eine Funktion, die auf einer ebenen Triangulierung definiert ist;

5. `champ` zeichnet ein Vektorfeld in 2D.

Ein letztes Detail: Die Mehrzahl der Graphikfunktionen, auf die ich in diesem Kapitel hingewiesen habe, lassen Varianten zu, die Funktionsgraphen auf direkterem Wege erstellen, wenn man eine Scilabfunktion als Argument angibt. Der Name dieser Funktionen beginnt mit einem f (`fplot2d, fcontour2d, fplot3d, fplot3d1, fchamp`,...). Im folgenden Kapitel wird insbesondere die Anwendung von `fchamp` für die Zeichnung von Vektorfeldern betrachtet, welche zu einer Differentialgleichung gehören (für zwei Dimensionen).

Um sich die vielfältigen Möglichkeiten vor Augen zu führen, reicht es die Rubrik `Graphic Library` des Hilfesystems zu durchstöbern. Sie können sich auch die Bibliothek von Enrico Ségré beschaffen, die einige interessante Graphikfunktionen enthält:

```
http://www.polito.it/~segre/scistuff.html
```

Kapitel 5

Einige Anwendungen und Ergänzungen

Dieses Kapitel soll Ihnen zeigen, wie man bestimmte Probleme aus dem Bereich der Numerischen Analysis mit Scilab löst, und es bringt einige Ergänzungen bezüglich bestimmter Punkte (Erzeugen von Zufallszahlen) an. Erinnern wir uns, dass Scilab über zahlreiche Grundbefehle und Funktionen verfügt, die ermöglichen, Probleme der optimalen Kontrolle, der Signalverarbeitung usw... zu lösen. Es gbit auch verschiedene Toolboxen, die von Anwendern entwickelt wurden (siehe die Rubrik ‚Contributions' auf der Scilab–Homepage).

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 ihrer elementarsten Funktionsweise ist sie 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 trotzdem 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 approximierte 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();
```

[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

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:

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, gebraucht man `fchamp` mit allen seinen Argumenten:

```
fchamp(MeineRechteSeite,t,x1,x2,arfact,rect,strf)
```

wobei `arfact` ein Skalierungsfaktor ist, um die Pfeilspitzen mehr oder weniger fett darzustellen (default 1), `rect` und `strf` haben dieselbe Bedeutung wie für `plot2d` (außer, dass das erste Zeichen ignoriert wird: man kann keine Legende angeben). Die isometrische Skalierung erhält man, wenn der Parameter `y` gleich 3 gesetzt wird. Letzte Verschönerung: man zeichnet einen kleinen Kreis, um die Anfangsbedingung zu markieren, und 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,1,[-delta_x,-delta_y,delta_x,delta_y],"031")
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, -9, "000")  // ein kleiner o zur Markierung der Anfangsbedingung
    u0 = [c_x;c_y];
    u = ode(u0, 0, t, VanDerPol);
    num = modulo(num+1,length(Farben));
    plot2d(u(1,:)',u(2,:)',Farben(num+1),"000")
  elseif c_i == 2 then
    break
  end
end
```

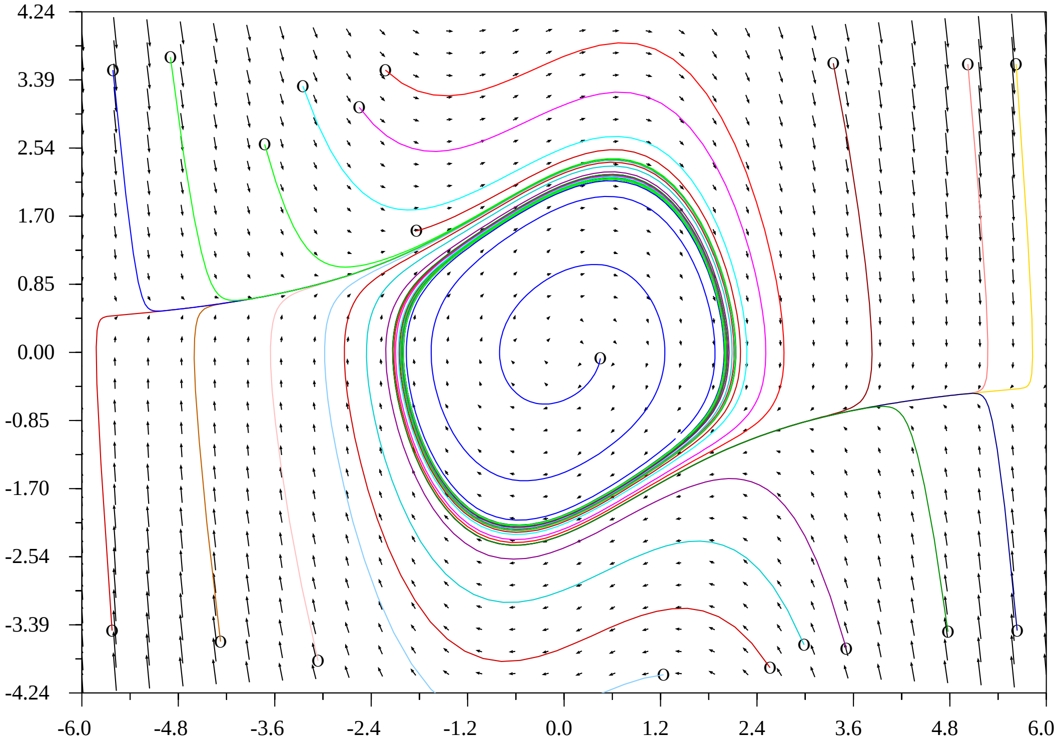


Abbildung 5.1: Einige Trajektorien im Phasenraum der Van–der–Pol–Gleichung

5.1.3 Weiteres zu `ode`

In diesem 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:

```
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 aus 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 $|e_i(t_k)| \le tol_i$ 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 (so dass 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,1,[x_min,y_min,x_max,y_max],"031")
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, -9, "000")  // ein kleiner o zur Markierung der 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,:)',Farben(num+1),"000")
  elseif c_i == 2 then
    break
  end
end
```

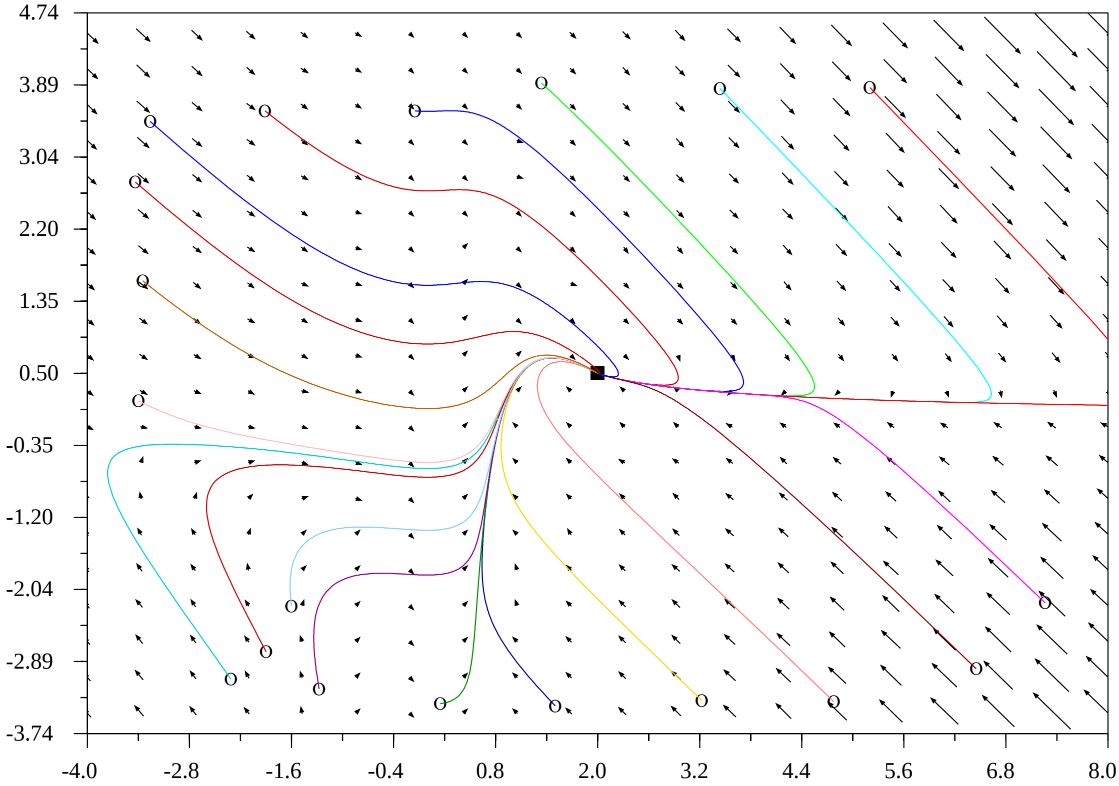


Abbildung 5.2: Einige Trajektorien im Phasenraum der Brusselator–Gleichung ($\epsilon = -4$)

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[3]:

$$X_{n+1} = f(X_n) = (aX_n + c) \bmod m, \; n \geq 0, \quad \text{mit} \left\{ \begin{array}{l} m = 2^{31} \\ a = 843314861 \\ c = 453816693 \end{array} \right.$$

Ihre Periode ist m (dies bedeutet, dass f eine zyklische Pemutation 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, werden die Zahlen miteinander multipiziert[4] (wobei ihnen zuvor 1 hinzugezählt wird, um zu verhindern, dass man jedes 60. Mal 0 als Keim erhält), das erhaltene Resultat wird mit 57 multipliziert, so dass $13 \times 32 \times 24 \times 60 \times 62 \times 57 \approx 2^{31}$ ergibt:

```
v = getdate()
rand("seed", 57*prod(1+v([2 6 7 8 9])))
```

Zu beachten ist, dass man den aktuellen Keim auch mit

```
Keim = rand("seed")
```

bestimmen kann. Ausgehend von der Gleichverteilung auf $[0, 1)$ kann man auch andere Verteilungen erhalten; **rand** stellt ebenfalls 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
```

[3]laut dem, was aus dem Sourcecode hervorgeht

[4]man könnte es ohne Zweifel besser machen...

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 darauf 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.

5.2.2 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 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")
```

Es ist vorteilhafter, die Funktion `grand` zu benutzen, die eine leistungsfähigere Methode verwendet. Wenn Sie andererseits diesen kleinen Trick[5] 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):

[5] Im Allgemeinen benutzt man eher die Funktion `grand`, die die meisten klassischen Verteilungen liefert.

```
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")
```

5.2.3 Zeichnen einer empirischen Verteilungsfunktion

Sei X eine stetige Zufallsvariable (reellwertig), von der man nur m unabhängige Realisierungen kennt, die im Vektor $X^r = (X_1,\ X_2,\ \dots,\ X_m)$ gespeichert sind. Ihre *Verteilungsfunktion* ist die Funktion

$$F(x) = \text{Wahrscheinlichkeit, dass } X \leq x$$

und die empirische Verteilungsfunktion, die durch die Stichprobe X^r definiert ist, ist gegeben durch

$$F_m(x) = card\{X_i <= x\}/m$$

Es ist eine Treppenfunktion, die sich einfach berechnen lässt, wenn man den Vektor X^r in aufsteigend sortiert (man hat also $F_m(x) = i/m$ für $X_i \leq x < X_{i+1}$). Der Standardsortieralgorithmus von Scilab ist die Funktion `sort`, welche in absteigender Reihenfolge sortiert[6]. Um den Vektor X^r in aufsteigender Reihenfolge zu sortieren, benutzt man also:

```
X_r_o = - sort(-X_r)
```

und um zuviel Hantier mit `plot2d` zu vermeiden, gibt es auch noch eine Variante zum Zeichnen von stückweise konstanten Funktionen:

```
plot2d2("onn",x,y,[optionale Argumente wie für plot2d])
```

zeichnet einen konstanten Bereich vom Wert `y(i)` im Intervall `[x(i),x(i+1)]`. Um sicher zu gehen, dass man einen kleinen Bereich (vom Wert 0) vor X^r_{min} und einen anderen (vom Wert 1) nach X^r_{max} hat, kann man folgendermaßen vorgehen:

```
function empirische_Verteilung(X_r)
  //
  //  zeichnet die (empirische) Verteilungsfunktion von X_r,
  //  einem (Zeilen- o. Spalten-)Vektor, der m Realisierungen von X enthält
  //
  m = length(X_r)
  X_r_o = matrix(X_r,m,1)  // um sicher zu gehen, dass man einen Spaltenvektor hat,
                           // damit der Code in beiden Fällen funktioniert
  X_r_o = -sort(-X_r_o)    // X_r_o für X_r ge(o)rdnet
                           // jetzt fügt man auf beiden Seiten je einen Punkt hinzu:
  dx = 0.05*(X_r_o(m) - X_r_o(1))
  X_r_o = [X_r_o(1)-dx ; X_r_o ; X_r_o(m)+dx]
                           // Berechnung der Ordinaten
  y = [0 ; (1:m)'/m ; 1]
                           // und das Bild
```

[6] siehe auch die Funktion `gsort`, die mehr kann

```
  xbasc()
  plot2d2("onn", X_r_o , y, 1, "121", "empirische Verteilungsfunktion")
  xselect()
endfunction
```

Im nächsten Beispiel verwenden wir die Normalverteilung $\mathcal{N}(0,1)$, deren Verteilungsfunktion mit der Fehlerfunktion `erf` von Scilab berechnet werden kann:

$$F(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = \frac{1}{2}(1 + erf(\frac{x}{\sqrt{2}}))$$

```
X = rand(200,1,"normal");  // man zieht 200 Stichproben
empirische_Verteilung(X);  // Bild der empirischen Verteilungsfkt.
                           // Daten zum Zeichnen der exakten Verteilungsfkt.
x = linspace(-5,5,200)';
y = (1 + erf(x/sqrt(2)))/2;
                           // man fügt dem ersten Bild diese Kurve hinzu
plot2d(x,y,2,"000")
```

siehe Abbildung 5.3.

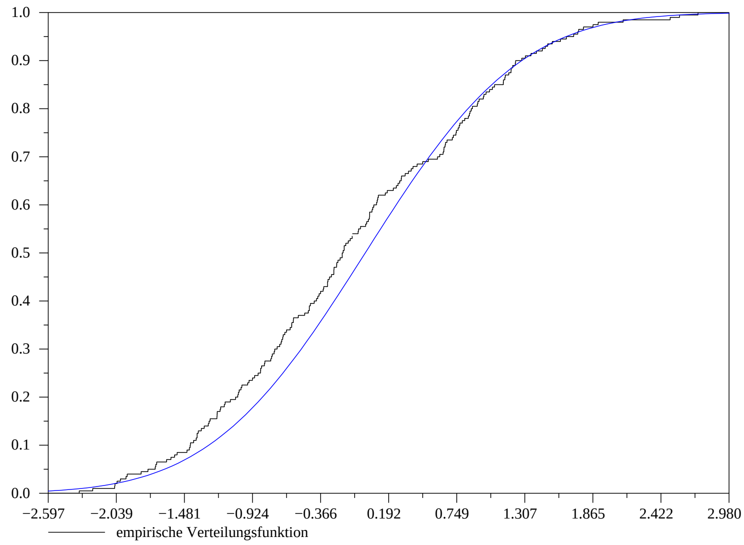


Abbildung 5.3: Exakte und empirische Normalverteilungsfunktion

5.2.4 Zeichnen eines Histogramms

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

```
histplot(n, X,[optionale Argumente wie für plot2d])
```

wobei `n` entweder eine ganze Zahl oder ein Zeilenvektor (mit $n_i < n_{i+1}$) und `X` der (Zeilen– oder Spalten–) Vektor mit den Daten ist, die verarbeitet werden:

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): der Wert dieses Histogramms, der diesem Intervall entspricht, beträgt also (m ist die Anzahl der Daten, und $\Delta C_i = n_{i+1} - n_i$):

$$\frac{\text{card } \{X_j \in C_i\}}{m \Delta C_i}$$

2. Wenn `n` eine ganze Zahl ist, werden die Daten in n äquidistante Klassen aufgeteilt:

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

Hier ein kleines Beispiel, immer noch mit der Normalverteilung (vgl. Abbildung 5.4):

```
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,2,"000")
```

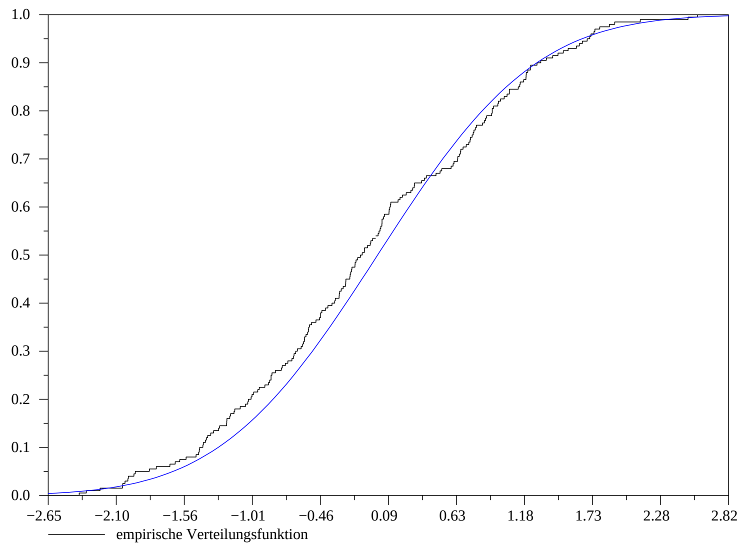


Abbildung 5.4: Histogramm einer Stichprobe von $N(0,1)$–verteilten Zufallszahlen

5.2.5 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. Ab der Version 2.4 ist Scilab mit einem Luxuszufallszahlengenerator ausgestattet, der alle klassischen Verteilungen zu simulieren ermöglicht. **`grand`** wird fast in derselben Weise verwendet wie **`rand`**, d.h. dass man eine der folgenden beiden Syntaxen verwenden kann (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, ...])
```

wobei **`verteilung`** eine Zeichenkette ist, die die Verteilung festlegt, und optionalen Parametern. Einige Beispiele (m Stichproben in Form eines Spaltenvektors):

1. eine Gleichverteilung über den ganzen Zahlen im Intervall $[1, 2147483562]$:

   ```
   X = grand(m,1,"lgi")
   ```

   (dieser Generatortyp bildet die Basis von **`grand`**, siehe weiter unten);

2. eine Gleichverteilung über den ganzen Zahlen im Intervall $[n1, n2]$:

   ```
   X = grand(m,1,"uin",n1,n2)
   ```

   (es muss $n2 - n1 \leq 2147483561$ gelten, andernfalls wird eine Fehlermeldung erzeugt);

3. für die Gleichverteilung auf $(0,1)$ (d.h., dass 0 und 1 kommen nicht vor):

   ```
   X = grand(m,1,"def")
   ```

   (dies wird durch Division der durch den Basisgenerator erhaltenen Zahlen durch 2147483563 erreicht)

4. für die Gleichverteilung auf (a,b) (a und b kommen nicht vor):

   ```
   X = grand(m,1,"unf",a,b)
   ```

5. für die Binomialverteilung $B(N,p)$:

   ```
   X = grand(m,1,"bin",N,p)
   ```

6. für die Poissonverteilung mit dem Mittelwert μ:

   ```
   X = grand(m,1,"poi",mu)
   ```

7. für die Normalverteilung mit dem Mittelwert μ und der Standardabweichung σ:

   ```
   X = grand(m,1,"nor",mu,sigma)
   ```

Es gibt noch weitere Beispiele (siehe die Hilfeseiten).

Der Basis–Generator von `grand` hat eine sehr langen Periode ($2.3\ 10^{18}$); dieser Generator wird in 32 „virtuelle" Generatoren zerlegt, von denen jeder 2^{20} =1 048 576 Zahlenblöcke liefern kann, diese wiederum enthalten jeweils 2^{30} =1 073 741 824 Zahlen aus dem Intervall [1 , 2 147 483 562] (laut `Help`...). Man kann von einem virtuellen Generator zu einem anderen mit

```
grand('setcgn',G)
```

wechseln, wobei $G = 1, 2, \ldots, 32$ ist, und den aktuellen (virtuellen) Generator mit

```
G = grand('getcgn',G)
```

bestimmen (standardmäßig ist $G = 1$), und es gibt auch Möglichkeiten zur Initialisierung (man braucht zwei Keime (seeds)). Kurzum: es gibt viele Möglichkeiten...

5.2.6 Klassische Verteilungsfunktionen und ihre Inversen

Diese Funktionen sind oft bei statistischen Tests (χ_r^2, ...) 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, und 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. So hätte man im Beispiel für das Zeichnen einer Verteilungsfunktion (wobei die Verteilungsfunktion der Normalverteilung $\mathcal{N}(0,1)$ mit der Fehlerfunktion ausgedrückt wird) auf direkterem Wege Folgendes benutzen können:

```
[y,z] = cdfnor("PQ",x,0*ones(x),1*ones(x));
```

5.2.7 Ein χ^2–Test

Im Folgenden wird ein solcher Test an einem Beispiel aus der Schule realisiert, welches darin besteht, N Ziehungen aus der *Polya–Urne* durchzuführen. Diese enthält zu Beginn r rote und g grüne Kugeln, und jede Ziehung besteht darin, zufällig eine Kugel zu ziehen und sie wieder in die Urne mit c Kugeln der gleichen Farbe zurückzulegen. Man notiert mit X_k den Anteil der grünen Kugeln nach k Ziehungen und mit V_k die Anzahl der grünen Kugeln:

$$X_0 = \frac{g}{g+r}, V_0 = g.$$

Wenn man $g = r = c = 1$ wählt, erhält man folgende Resultate, die man per Simulation (für die ersten beiden) bestätigen will:

1. $E(X_N) = E(X_0) = X_0 = 1/2$;
2. X_N folgt einer Gleichverteilung auf $\{\frac{1}{N+2}, \ldots.., \frac{N+1}{N+2}\}$;
3. für $N \to +\infty$, konvergiert X_N f.s. gegen die Gleichverteilung auf $[0,1)$.

Basis–Funktionen(in Scilab) für das Polya–Problem

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 V_N zurückgibt:

```
function [XN, VN] = Polya_Urne(N)
  // Simulation von N Ziehungen aus einer "Polya-Urne":
  //
  VN = 1 ; V_plus_R = 2 ; XN = 0.5
  for i=1:N
    u = rand()              // Ziehung einer Kugel
    V_plus_R = V_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 (1 - XN bei einer roten Kugel)
      VN = VN + 1
    end
    XN = VN / V_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 (wie in allen diesen Sprachen...), kann man eine Funktion schreiben, die in der Lage ist, m Prozesse „parallel" zu simulieren (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). Eine solche Funktion sieht dann so aus (in der die Funktion `find` liefert die Indizes der Komponenten eines boolschen Vektors mit dem Wert %T):

```
function [XN, VN] = parallele_Polya_Urne(N,m)
  // Simulation von m parallelen Prozessen mit je N Ziehungen aus der "Polya-Urne"
  //
  VN = ones(m,1) ; V_plus_R = 2 ; XN = 0.5*ones(m,1)
  for i=1:N
    u = rand(m,1)            // 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

In diesem Skript möchte man durch Simulation den theoretischen vorhergesagten Erwartungswert sowie die Hypothese H, dass „X_N einer Gleichverteilung auf $\{\frac{1}{N+2}, ..., \frac{N+1}{N+2}\}$ folgt", bestätigen. Zu diesem Zweck werden m Simulationen vorgenommen:

1. Man gibt den empirischen Erwartungswert zusammen mit dem empirischen 95%–Fehlerintervall, welches man durch die Berechnung der empirischen Varianz erhält, aus.

2. Um die Hypothese H zu überprüfen, führt man den χ^2–Test durch: man erstellt eine Häufigkeitsverteilung des empirischen Erwartungswertes `occ(i)` für jedes der $N+1$ möglichen Resultate und berechnet dann die Größe
$$Y = \frac{\sum_{i=1}^{N+1}(occ_i - mp_i)^2}{mp_i}$$
die dazu tendiert, groß zu werden, falls die zu prüfende Hypothese nicht stimmt (p_i ist die theoretische Wahrscheinlichkeit, das Resultat i zu erhalten, aber hier erwartet man eine Gleichverteilung für jedes der $N+1$ möglichen Resultate und demzufolge mit $p_i = p = 1/(N+1)$). Man kann zeigten, dass für $m \to +\infty$ Y einer χ^2–Verteilung mit N Freiheitsgraden folgt. Wenn man annimmt, dass die Anzahl der Simulationen m ausreichend groß ist, um nah genug an der erwarteten asymptotischen Verteilung zu sein, verwirft man die Hypothese H, falls $Y > F^{-1}(1-\alpha)$ mit z.B. $\alpha = 0,05$ (wobei F die Verteilungsfunktion von χ^2 mit N Freiheitsgraden ist).

```
// Polya-Simulation :
N = 10;
m = 5000;
[XN, VN] = parallele_Polya_Urne(N,m);
EN = sum(XN)/m;                        // empirischer Erwartungswert
sigma = sqrt(sum((XN - EN).^2)/(m-1)); // empirische Varianz
delta =  2*sigma/sqrt(m);              // Inkrement für das empirische Intervall
// Darstellung des Ergebnisses für den Erwartungswert
write(%io(2)," E exakt = "+string(0.5))
write(%io(2)," E Schätzung = "+string(EN))
write(%io(2)," 95%-ges Konfidenzintervall : ["+string(EN-delta)+","+string(EN+delta)+"]")

// die Tests
```

```
alpha = 0.05;
p = 1/(N+1);  // theoretische Wahrscheinlichkeit für jedes Resultat (Gleichverteilung)

// 1.) Berechnung der Anzahl der Vorkommen eines jeden der N+1 möglichen Resultate:
occ = zeros(N+1,1);
for i=1:N+1
  occ(i) = sum(bool2s(XN == i/(N+2)));
end
// kleine Kontrolle:
if sum(occ) ~= m then, error(" Fehler..."), end

// 2.) Berechnung der Testgröße Y
Y = sum( (occ - m*p).^2 / (m*p) );

// 3.) die grossen Werte von Y müssen verworfen werden, d.h. jene für die gilt
//    F_chi2_Verteilung_in_N_dof(Y) > 1 - Schwelle
//    Der Schwellwert (in Y) wird mit der Inversen der Verteilungsfunktion berechnet:
Y_schwell = cdfchi("X",N,1-alpha,alpha);

// 4.) Anzeige der Ergebnisse
//
write(%io(2)," chi-Quadrat-Test: ")
write(%io(2)," ----------------- ")
write(%io(2)," durch den Test erhaltener Wert: "+string(Y))
write(%io(2)," Schwellwert, der nicht überschritten werden soll: "+string(Y_schwell))
if (Y > Y_schwell) then
  write(%io(2)," vorläufiges Ergebnis: Hypothese verworfen!")
else
  write(%io(2)," vorläufiges Ergebnis: Hypothese nicht verworfen!")
end

// 5.) Zeichnungen...
deff("d = d_chi2(X,N)","d = X.^(N/2 - 1).*exp(-X/2)/(2^(N/2)*gamma(N/2))");
xbasc()
haeufigkeiten = occ/m;
ymax = max([haeufigkeiten; p]);
rect = [0 0 1 ymax*1.05];
xsetech([0,0,1,0.5])
plot2d3("onn",(1:N+1)'/(N+2), haeufigkeiten, 2, "011", " ", rect, [0 N+2 0 10])
plot2d((1:N+1)'/(N+2),p*ones(N+1,1),-2,"000")
xtitle("empirische Häufigkeiten (vertikale Linien) und " + ...
       "exakte Wahrscheinlichkeiten (Kreuze)")
xselect()
xsetech([0.33,0.5,0.33,0.5])
// die chi-Quadrat-Dichte in N Freiheitsgraden wird gezeichnet
X = linspace(0,1.1*max([Y_schwell Y]),50);
D = d_chi2(X,N);
plot2d(X,D,1,"122","chi-Quadrat-Dichte")
plot2d3("gnn",[Y Y_schwell],[d_chi2(Y,N) d_chi2(Y_schwell,N)],[2 3],"000")
xstring(Y_schwell,1.2*d_chi2(Y_schwell,N),"Schwelle")
xtitle("Wert von Y im Vergleich zum Schwellwert")
xselect()
```

Mit $N = 10$ und $m = 5000$ erhält man (vgl. Abbildung 5.5):

```
E exakt = 0.5
E Schätzung = 0.4959167
95%-iges Konfidenzintervall: [0.4884901,0.5033433]

chi-Quadrat-Test:
-----------------
durch den Test erhaltener Wert: 8.3176
Schwellwert, der nicht überschritten werden soll: 18.307038
vorläufiges Ergebnis: Hypothese nicht verworfen!
```

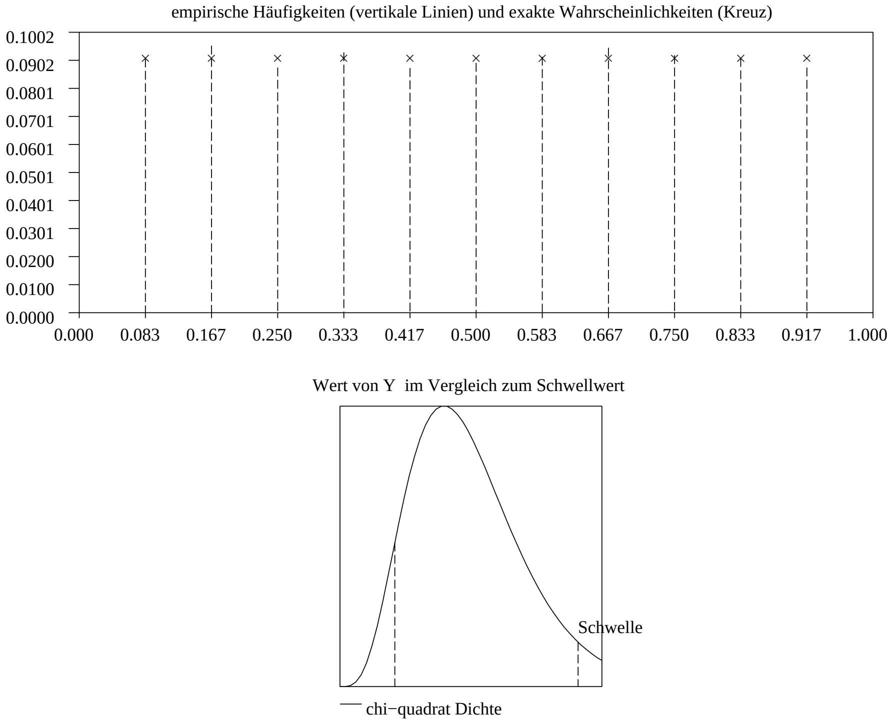


Abbildung 5.5: χ^2–Test für die Polya–Urne

5.2.8 Kolmogorov–Smirnov–Test

Dieser Test ist natürlicher als der χ^2–Test, wenn die erwartete Verteilung eine stetige Verteilungsfunktion besitzt. Sei X eine reelle Zufallsvariable, deren Verteilung eine stetige Verteilungsfunktion F besitzt, und X_1, X_2,..., X_m m unabhängige Realisierungen eines Prozesses, von dem man annimmt, dass er derselben Verteilung folgt wie X (man will diese Hypothese prüfen). Der Test besteht darin, die Abweichung zwischen der exakten und der empirischen Verteilungsfunktion zu messen:

$$K_m = \sqrt{m} \sup_{-\infty < x < +\infty} |F(x) - F_m(x)|$$

und darin, diese mit einem „annehmbaren“ Wert zu vergleichen. Im Grenzfall besitzt die Verteilung der Zufallsvariable K_m die folgende Verteilungsfunktion:

$$\lim_{m \to +\infty} P(K_m \leq x) = H(x) = 1 - 2 \sum_{j=1}^{+\infty} (-1)^{j-1} e^{-2j^2x^2}$$

und genau wie beim χ^2–Test wird die Hypothese verworfen, wenn

$$K_m > H^{-1}(1 - \alpha)$$

mit z.B. $\alpha = 0.05$. Wenn man die Näherung $H(x) \simeq 1 - e^{-2x^2}$ benutzt, dann ist der Schwellwert, der nicht überschritten werden soll, der folgende:

$$K_{schwell} = \sqrt{\frac{1}{2}\ln(\frac{1}{\alpha})}$$

Man verfügt jedoch über einen asymptotischen Ausdruck für diese Schwelle, der der exakten Verteilung, die mit K_m zusammenhängt, entspricht (ausreichend genau für $m > 30$):

$$K_{schwell} = \sqrt{\frac{1}{2}\ln(\frac{1}{\alpha})} - \frac{1}{6\sqrt{m}} + O(\frac{1}{m})$$

Die Berechnung von K_m stellt keine Probleme dar, wenn man den Vektor $X^r = (X_1,\ X_2,\ \ldots,\ X_m)$ sortiert. Angenommen, das Sortieren sei erfolgt, unter Beachtung von

$$\sup_{x\in[X_i,X_{i+1}[} F_m(x) - F(x) = \frac{i}{m} - F(X_i), \text{ und } \sup_{x\in[X_i,X_{i+1}[} F(x) - F_m(x) = F(X_{i+1}) - \frac{i}{m}$$

ist es einfach, diese beiden Größen zu berechnen:

$$K_m^+ = \sqrt{m} \sup_{-\infty<x<+\infty} (F_m(x) - F(x)) = \sqrt{m} \max_{1\le j\le m} (\frac{j}{m} - F(X_j))$$

$$K_m^- = \sqrt{m} \sup_{-\infty<x<+\infty} (F(x) - F_m(x)) = \sqrt{m} \max_{1\le j\le m} (F(X_j) - \frac{j-1}{m})$$

und man erhält dann $K_m = \max(K_m^+,\ K_m^-)$.

Dieser Test wird am folgenden stochastischen Prozess verdeutlicht (wobei U die Gleichverteilung auf $[0,1]$ bezeichnet):

$$X_n(t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} (1_{\{U_i \le t\}} - t)$$

so dass für festes $t \in (0,1)$ gilt

$$\lim_{n\to+\infty} X_n(t) = \mathcal{N}(0, t(1-t))$$

Dafür beabsichtigt man, ein „genügend großes" n zu nehmen, m Simulationen mit festem n durchzuführen und schließlich die Konvergenz gegen die Verteilung zu beobachten; zunächst graphisch (indem die empirische zusammen mit der exakten Verteilungsfunktion gezeichnet wird) und dann indem dieser *Kolmogorov–Smirnov*–Test durchgeführt wird. Mit den in Scilab erlaubten Kurzschreibweisen kann die Hauptfunktion, um eine Realisierung von $X_n(t)$ zu erhalten, in folgender Weise geschrieben werden:

```
function X = Brownsche_Bewegung(t,n)
  //
  X = sum(bool2s(grand(n,1,"def") <= t) - t)/sqrt(n)
endfunction
```

Hier ein mögliches Skript, das nach m Simulationen des Prozesses den Graphen der empirischen Verteilungsfunktion anzeigt, ihn dann mit der erwarteten Verteilungsfunktion überlagert und schließlich den statistischen Test durchführt:

```
  // Brown'sche Bewegung
  // kleine Simulation, um zu veranschaulichen, dass Xn(t) --> N(0,t(1-t)) für n -> oo
  //
  // wobei  Xn(t) = summe_{i=1}^n ( 1_(Ui <= t) - t ) /sqrt(n)
  //
  t = 0.3;
  sigma = sqrt(t*(1-t));  // erwartete Standardabweichung
  n = 1000;               // "großes" n
  m = 4000;               // 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
empirische_Verteilung(X)      // das Bild der empirischen Verteilungfunktion
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,2,"000")                             // man fügt sie dem
                                                // ersten Bild hinzu

// Bereitstellen des KS-Tests
alpha = 0.05
X = - sort(-X); // Sortieren
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_schwell = sqrt(log(1/alpha)/2) - 1/(6*sqrt(m)) ;

// Anzeige des Resultats
//
write(%io(2)," KS-Test: ")
write(%io(2)," -------- ")
write(%io(2)," durch den Test erhaltener Wert: "+string(Km))
write(%io(2)," Schwellwert, der nicht überschritten werden soll: "+string(K_schwell))
if (Km > K_schwell) then
  write(%io(2)," vorläufiges Ergebnis: Hypothese verworfen!")
else
  write(%io(2)," vorläufiges Ergebnis: Hypothese nicht verworfen!")
end
```

Für $n = 1000$ und $m = 4000$ erhält man (vgl. Abbildung 5.6):

```
KS-Test:
--------
durch den Test erhaltener Wert: 1.1204036
Schwellwert, der nicht überschritten werden soll: 1.2212382
vorläufiges Ergebnis: Hypothese nicht verworfen!
```

Zum Abschluss wird für ein festes n die Kurve $X_n(t), t \in [0,1]$ gezeichnet. Es ist nach dem Sortieren der U_i in aufsteigender Reihenfolge (unter der Annahme, dass die U_i zudem alle verschieden und ungleich 0 oder 1 sind und man $U_0 = 0$ und $U_{n+1} = 1$ setzt) ziemlich einfach zu sehen, dass gilt:

$$X_n(t) = \frac{i - nt}{\sqrt{n}}, \quad \text{für } U_i \leq t < U_{i+1}$$

Man kann also diese Kurve mit folgender Funktion berechnen (wobei die Unstetigkeiten durch vertikale Linien verbunden werden):

```
function [] = Bild_Brownsche_Bewegung(n)
  //
  U = -sort(-rand(1,n))
  Xbas = ((0:n-1) - n*U)/sqrt(n)
  Xhaut = ((1:n) - n*U)/sqrt(n)
  absc = [0 ; matrix([U ; U], 2*n, 1) ; 1]
  ord  = [0 ; matrix([Xbas ; Xhaut], 2*n, 1) ; 0]
```

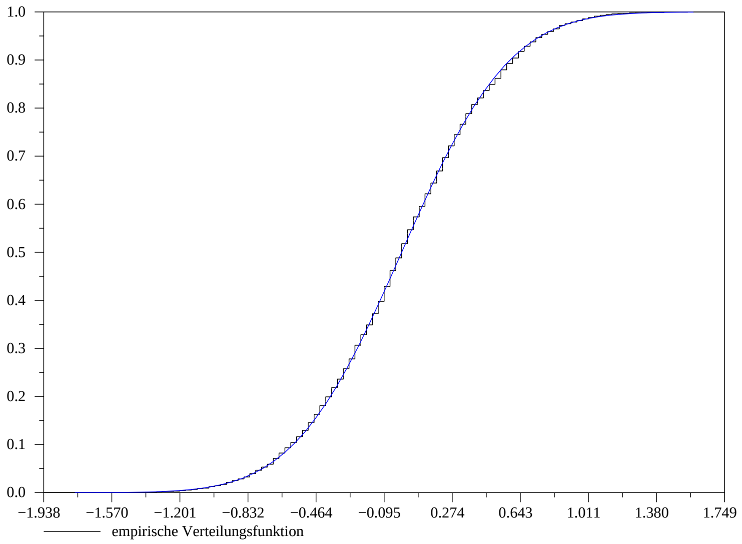


Abbildung 5.6: empirische und exakte Verteilungsfunktion

```
   xbasc()
   plot2d(absc,ord)
endfunction
```

Die Abbildung 5.7 zeigt eine Kurve, die man mit $n = 10000$ erhält; drei aufeinanderfolgende Zooms lassen den fraktalen Charakter der „Grenzkurve“ erkennen.

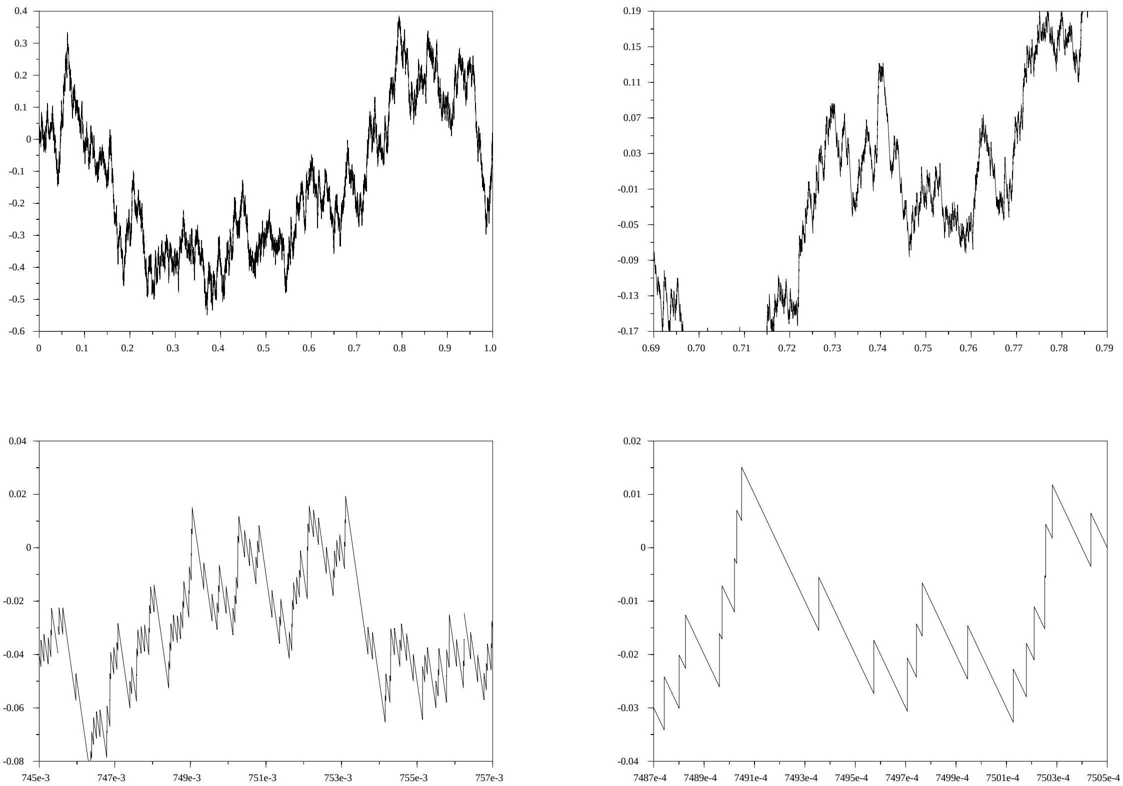


Abbildung 5.7: Brown'sche Bewegung

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, wenn der neue Wert von `K` derart gestaltet ist, dass:

- `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 so aus, wie man es wollte;
- `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).

Die beste Methode ist hier, die Vektoren `x` und `y` vollständig zu definieren mit einer Initialisierung der Art:

```
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 in folgender Weise 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 (wenn die Funktion einen einzigen Wert zurückgibt, ist es dagegen OK). Man muss in zwei Schritten vorgehen:

```
[rac1, rac2] = resol(a, b, a);
x1(i) = rac1; x2(i) = rac2;
```

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` zu laden ! Ein kleiner Trick: Ihre Anweisung `getf` ist sicher nicht sehr weit in der Befehls–Übersicht, drücken sie also die Taste ↑, bis Sie sie finden.

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, sollte 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 dafür anzupassen, dass sie in beiden Fällen funktioniert. Will man jedoch schnelle Berechnungen erzielen, ist es ratsamer, sich Matrixausdrücken zu bedienen, anstatt Iterationen zu benutzen, und dann muss man die eine oder die andere Form auswählen. 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 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

Funktion gilt als eine Scilabvariable, und dementsprechend kann man eine Funktion als Argument einer anderen Funktion übergeben. Bei einem Grundbefehl ist dies anders. Z. B. sind die meisten der üblichen mathematischen Funktionen (exp, cos, sin) Grundbefehle und nicht Scilabfunktionen. Man muss diese Tatsache berücksichtigen, wenn man eine Funktion benutzt, die ein solches Argument erwartet. Hier ist z.B. 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 Standardabweichung
  //  f muss als Scilabfunktion codiert sein,
  //  die ein Argument vom Typ Vektor zulässt.
  //  n ist die Anzahl der verwendeten Zufallszahlen
  u = (b-a)*rand(n,1,"uniform") + a
  y = f(u)
  Mittelwert = sum(y)/n
  I = (b-a)*Mittelwert
  sigma = (b-a) * sqrt(sum( (y - Mittelwert).^2 )/(n-1))
endfunction
```

Angenommen, man wollte diese Funktion testen, indem man die Exponentialfunktion zwischen 0 und 1 integriert; folgende Methode funktioniert nicht:

```
-->[I,sigma]=MonteCarlo(0,1,exp,10000)
                           !--error    25
bad call to primitive :exp
```

Man muss also die Exponentialfunktion als Scilabfunktion codieren, um den gesuchten Effekt wie im folgenden Skript zu erzielen:

```
// Integration nach Monte Carlo
n = 10000 ;
a = 0 ;
b = 1 ;
deff("y=f1(x)","y = exp(x)")  // f1 ist die in Scilab codierte Exponentialfunktion
I_exact = %e - 1;
[I, sigma] = MonteCarlo(a,b,f1,n);
delta = 2*sigma/sqrt(n);
// Ausgabe des Ergebnisses
write(%io(2)," I exakt = "+string(I_exact))
write(%io(2)," I approx. = "+string(I))
write(%io(2),...
" empirisches 95%-Konfidenzintervall : ["+string(I-delta)+","+string(I+delta)+"]")
```

was folgendes Resultat liefert:

```
 I exakt = 1.7182818
 I approx. = 1.7182109
 empirisches 95%-Konfidenzintervall : [1.70843,1.7279917]
```

6.8 Auswertung boolscher Ausdrücke

Im Gegensatz zu C verläuft die Auswertung boolscher Ausdrücke der Form

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

zunächst durch Auswertung der boolschen Unterausdrücke a und b, bevor dann ‚oder' im ersten bzw. ‚und' im zweiten Fall ausgewertet werden. (Im Fall, dass a wahr ist für ‚oder' (und falsch für ‚und'), kann man sich die Auswertung des boolschen Ausdrucks b schenken.) (Vgl. Priorität der Operatoren im Kapitel Programmierung)

6.9 Komplexe und reelle Zahlen

Alles ist in Scilab darauf ausgelegt, mit reellen und komplexen Zahlen in derselben Weise umzugehen! Dies ist ziemlich praktisch, kann aber auch zu gewissen Überraschungen führen, beispielsweise dann, wenn Sie eine reelle Funktion ausserhalb ihres Definitionsbereiches (z.B. $\sqrt{x}$ et $\log(x)$ für $x < 0$, $\mathrm{acos}(x)$ und $\mathrm{asin}(x)$ für $x \notin [-1, 1]$, $\mathrm{acosh}(x)$ für $x < 1$) auswerten, denn Scilab gibt die Auswertung der ins Komplexe fortgesetzten Funktion zurück. Um festsustellen, ob man es mit einer reellen oder komplexen Variable zu tun hat, kann man die Funktion `isreal` benutzen:

```
-->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
```

That 's all Folks...

Anhang A

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)]; // um die Division durch Null zu vermeiden...
   plot(x,y,"x","y","y=sin(x)/x")
   ```

6. Für diese Übung nimmt man an, dass man Scilab gerade gestartet hat[1] (mit dem Befehl `clear` kann man sich überzeugen, dass man ein wenig Platz gewinnt, denn man löscht einige Variablen aus der Anfangsumgebung). Wenn man `who` eingibt, erhält man schließlich die Information

[1] diese Korrektur ist mit der Version 2.3.1 erfolgt
   ```

```
 using 3837 elements out of 1000000.  and 41 variables out of 499
```

Man kann also berechnen:

```
-->floor(sqrt(1000000 - 3837))
ans =

    998.
```

und versuchen eine Matrix der Ordnung 998 zu erzeugen.

```
-->A = eye(998,998);
```

Man gibt nochmals `who` ein:

```
  using 999846 elements out of 1000000.  and 43 variables out of 499
```

Zählen wir zusammen:

```
-->  998*998 + 3837 +  2 + 2  + 1
-->  //  Platz für den Inhalt von A + anfangs gebrauchter Platz
-->  //  +  Variable A +   Variable  ans + Platz für den Inhalt von ans
  ans =

    999846.
```

Bemerkung: man kann nun sozusagen nicht mehr weiterarbeiten (es bleibt ja kein Platz mehr). Um die Größe des Heap zu erhöhen, muss man den Befehl `stacksize` benutzen (z. B. `stacksize(2000000)`, um die Größe standardmäßig zu verdoppeln).

Anhang B

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

//------------------------------------------------------------------

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 „vektorisierte“ Kreuzprodukt:

```
function v=prod_vect_v(v1,v2)
  // das vektorisierte Kreuzprodukt...
  // v1 und v2 müssen 3 x n - Matrizen sein.
  v=zeros(v1)
  v(1,:) = v1(2,:).*v2(3,:) - v1(3,:).*v2(2,:)
  v(2,:) = v2(1,:).*v1(3,:) - v2(3,:).*v1(1,:)
  v(3,:) = v1(1,:).*v2(2,:) - v1(2,:).*v2(1,:)
endfunction
```

Literaturverzeichnis

[1] David GOLDBERG, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol. 23, N. 1, March 1991.

[2] Donald KNUTH, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., vol II, 2nd ed.

[3] Reuven Y. RUBINSTEIN, *Simulation and the Monte Carlo Method*, John Wiley & Sons, Wiley Series in Probability and Mathematical Statistics.

[4] Ernst HAIRER, *Polycopié "Analyse Numérique"*, http://www.unige.ch/math/folks/hairer/polycop.html

Index