Kurzeinführung in Scilab

Alexander Stoffel
Institut für Nachrichtentechnik
Fakultät für Informations-, Medien- und Elektrotechnik
Fachhochschule Köln

13. April 2015

Inhaltsverzeichnis

1 Der Anfang ist nicht schwer **6**

2 Einbinden einer Bibliothek **6**

3 Konstante, Variable und elementare Funktionen **8**

4 Grafische Darstellung elementarer Funktionen und komplexer Zahlen **10**

5 Vektoren **11**

6 Matrizen **13**

7 Lineare Gleichungssysteme **16**

8 Eigenwerte und Eigenvektoren **19**

9 Wie Vektoren und Matrizen einfach erzeugt werden **20**

10 Elementweise Operationen **22**

11 Grafik **24**

12 Zeichenketten **32**

13 Lesen und Schreiben von Daten **34**

14 FFT **36**

15 Programmierung **37**

16 Funktionen **40**

17 Listen **41**

18 Polynome und rationale Funktionen **42**

19 Numerische Integration **47**

20 Lösung nichtlinearer Gleichungen **48**

21 Gewöhnliche Differentialgleichungen **49**

22 Wahrscheinlichkeitsrechnung und Statistik **51**

23 Behandlung von Tondateien **53**

24 Werkzeuge für Wavelets **54**
24.1 Zur Bibliothek 54
24.2 Skalierungsfunktionen und Wavelets 54
24.3 Wavelet-Transformationen für eindimensionale Signale 55

24.3.1 Einstufige Transformationen 55
24.3.2 Mehrstufige Transformationen 56
24.4 Behandlung von Bilddaten 58
24.4.1 Wavelet-Transformationen für Bilddaten 58
24.4.2 Ein- und Ausgabe von Bilddaten 59
24.5 Das Rechnen mit Filtern 61
24.6 Experimente zur Kodierung 62

Vorbemerkung

Das Program „Scilab“ erlaubt es, mit einer einfachen Syntax (d.h mit einfachen Regeln) numerische (zahlenmäßige) Rechnungen durchzuführen und dabei insbesondere Methoden der linearen Algebra (Vektoren und Matrizen) anzuwenden. Eine besondere Reservierung von Speicherplatz ist hierzu nicht notwendig (im Gegensatz zu C). Es ist kommando-orientiert, man muss also einfach kurze Kommandos in eine Kommandozeile eintippen. Die Kommandos werden (hinter dem Prompter „`-->`“) eingetippt und interpretiert (also nicht kompiliert wie bei C). Die Ausgabe erfolgt sofort automatisch am Bildschirm bzw. in einem Grafik-Fenster (für die Grafik-Kommandos). Was hinter „`//`“ steht, wird von Scilab ignoriert (ist also Kommentar).

Es existieren Versionen für alle gängigen Plattformen. Sie sind alle *kostenlos* erhältlich auf

`http://www.scilab.org`

Dort sind auch Links zu verschiedenen Einführungen, die hier vorliegende Kurzeinführung ist nur als Ergänzung zu den dort erhältlichen gedacht. Die Syntax wird hier im wesentlichen anhand von Beispielen von Eingaben in das Scilab-Eingabefenster erklärt. Beispielprogramme zu verschiedenen Themen, nützliche Hilfsfunktionen und die dazugehörigen Hilfsdateien kann man sich von

`http://alexanderstoffel.selfip.org/scilab.html`

herunterladen.

Die vorliegende Version der Kurzeinführung ist zuletzt zur Version 5.5.2 sporadisch überarbeitet worden. Dabei sind auch sicherlich nicht alle Neuerungen, die sich inzischen ergaben, berücksichtigt worden. Sie enthält auch sicher noch viele Tippfehler und sonstige Mängel, insbesondere hinsichtlich des Zeilenumbruchs. Der Leser möge dies entschuldigen.

1 Der Anfang ist nicht schwer

Zum Beginn der Arbeit mit Scilab sollte man sich — nach der Installation — auf seinem Rechner ein Unterverzeichnis einrichten, in dem man alle Dateien aufbewahrt, die bei der Arbeit mit Scilab notwendig und nützlich sind. Man sollte dieses Verzeichnis als Arbeitsverzeichnis bei Scilab einstellen, dann erspart man sich das Eintippen von Pfaden. Insbesondere die Beispielprogramme gehen davon aus. dass die benötigten Hilfsdateien alle in diesem Unterverzeichnis vorhanden sind. Das eingestellte Arbeitsverzeichnis kann man sich durch das Kommando `pwd` (print working directory, wie unter UNIX) ausgeben lassen. Ändern kann man das Arbeitsverzeichnis durch das Kommando `chdir('`*Pfadname*`')`, also beispielsweise

```
chdir('/Users/alexander/scilab-ma')
```

Bei erfolgreichem Wechsel des Arbeitsverzeichnisses wird eine 0 zurückgegeben, die Rückgabe einer 1 zeigt an, dass ein Fehler aufgetreten ist (z.B. das Verzeichnis gar nicht existiert).

Zum Beginn ist zu empfehlen, sich das Beispielprogramm `elebei.sce` anzuschauen und es auszuführen. Es ist auf

```
http://alexanderstoffel.selfip.org/scimat/elembei.sce
```

erhältlich. Textdateien mit ausführbaren Scilab-Kommandos wie die Datei `elebei.sce` (die Autoren von Scilab empfehlen hierfür die Endung `.sce`) kann man durch das Kommando `exec('elebei.sce')` (oder mit dem entsprechenden anderen Namen) als Programme ausführen lassen. Die Datei `elebei.sce` enthält am Anfang das Kommando „`mode(7)`“ , das die Ausführung bremst und nur jeweils eine Zeile ausführt. Weiter geht es dann erst, wenn man die Zeilenvorschub- oder die Enter-Taste betätigt. Über die angegebene Web-Adresse sind noch eine Reihe weiterer Beispielprogramme zu spezielleren Themen erhältlich.

Zu den einzelnen Kommandos erhält man über das Help-Menu oder durch Eingabe von `help` *Kommandoname* eine Dokumentation (also beispielsweise durch Eintippen von `help chdir`).

Für umfangreichere Arbeiten ist es sinnvoll, nicht die einzelnen Kommandos im Kommandofenster einzutippen, sondern sie in einer Datei zu sammeln und mit einem geeigneten Dateinamen mit der empfohlenen Endung `.sce` abzuspeichern. Scilab stellt hierfür einen eigenen Editor zur Verfügung, der das Arbeiten auch durch geeignete Farbdarstellung unterstützt. Man kann die so erstellten Programme auch direkt aus dem Editor heraus starten.

2 Einbinden einer Bibliothek

In der Datei `elembei.sce`, die auf

```
http://alexanderstoffel.selfip.org/scimat/elembei.sce
```

zur Verfügung gestellt wird, befinden sich eine Reihe nützlicher Funktionen, die speziell für die Mathematik-Lehrveranstaltungen der ersten beiden Studienjahre konzipiert sind. Man kann diese Funktionen (z.b. `eplot`) aus dieser Datei in ein anderes Programm kopieren. Auf Dauer ist ein derartiges Vorgehen sehr umständlich und nachteilig. Alle Quellprogramme in einer Datei machen diese schnell unübersichtlich; außerdem belegt das Laden aller Funktionen durch `exec` unnötig Speicherplatz. Professioneller ist das Arbeiten mit

Bibliotheken. Das sind konkret Unterverzeichnisse, die eine ganz bestimmte Struktur haben, die unter anderem alle Quellprogramme der Funktionen enthalten, und zwar jedes in einer gesonderten Datei, die *Funktionsname*`.sci` heißt. Eine Vielzahl derartiger Bibliotheken sind im Internet verfügbar. Sie haben in der Regel eine ähnliche Struktur und können auf ähnliche Weise installiert und geladen werden.

Die in der großen Datei `elembei.sce` vorhandenen Hilfsfunktionen sind in der Werkzeugbox `mathlib` vorhanden, und das durch Auspacken der Archivdatei `mathlib.zip` erhältlich ist. Diese kann von der Webadresse

`http://alexanderstoffel.selfip.org/mathlib.html`

heruntergeladen werden. Zur Installation der Bibliothek gehen Sie wie folgt vor:

(a) Packen Sie die Archivdatei aus. Sie erhalten ein Unterverzeichnis mit Namen `mathlib`.

(b) Schieben Sie dieses Unterverzeichnis an eine sinnvolle Stelle Ihrer Festplatte (z.B. als Unterverzeichnis in Ihr übliches Scilab-Arbeitsverzeichnis).

(c) Zur Installation führen Sie das Programm `builder.sce` aus (auf der obersten Ebene dieses Unterverzeichnisses, durch Eintippen von

`exec('`*Pfad...*`/mathlib/builder.sce')`

oder mit dem Menu „File“). Anschließend führen Sie analog das Programm `loader.sce` aus (ebenfalls auf der obersten Ebene dieses Unterverzeichnisses). Damit sind die Funktionen der Bibliothek verfügbar. U.a. finden Sie danach im Unterverzeichnis `macros` neben den Quelldateien (Dateinamen `*.sci`) die übersetzten Funktionsunterprogramme (Dateinamen `*.bin`).

Wenn Sie Scilab neu starten (oder alle Variablen mit dem Befehl `clear` gelöscht haben), dann können Sie die Bibliothek durch Aufrufen des Programms `loader.sce` laden. Das Aufrufen von `builder.sce` ist nur einmal zur Installation notwendig. Der große Vorteil dieser Vorgehensweise ist, dass Sie damit eine umfangreiche Dokumentation zu den geladenen Funktionen in die on-line-Dokumentation von Scilab mit einbinden. Sie können nicht nur für die von Scilab zur Verfügung gestellten Funktionen, sondern auch für die der Bibliothek ausführliche Informationen durch Eintippen von `help` *Funktionsname* erhalten. Oder Sie werfen einen Blick in das Handbuchkapitel „elem. math. library“ (ganz am Schluss des Inhaltsverzeichnisses, wenn Sie anschließend keine weitere Bibliothek geladen haben).

Wenn man eine Bibliothek immer benutzen möchte, ist es lästig, bei jedem neuen Start von Scilab das Programm `loader.sce` auszuführen. Hierzu kann man eine Startdatei anlegen, die Scilab-Kommandos enthält, die bei jedem Start von Scilab ausgeführt wird. Den Pfad des Verzeichnisses, in dem Scilab nach einer Startdatei sucht, erhalten Sie durch eintippen von `SCIHOME`. Dort legt Scilab auch andere Dateien ab (u.a. die, in der die eingetippten Befehle gespeichert werden, die Sie durch Betätigen der Taste `Cursor nach oben` zurückholen können). Die Startdatei muss den Namen `.scilab` tragen (beachten Sie den Punkt am Anfang). Beachten Sie außerdem, dass Scilab beim allerersten Aufruf ein Unterverzeichnis mit dem Namen `.Scilab` anlegt; beim Autor ist der komplette Pfad zur Startdatei

```
/Users/alexander/.Scilab/scilab-5.2.0/.scilab
```

Achten Sie darauf, dass diese Datei am Schluss ein Zeichen für Zeilenvorschub enthalten muss, sonst wird die letzte Zeile nicht gelesen. Eine Kommentarzeile am Ende kann ein derartiges Unglück stets verhindern. Hier als Beispiel der Inhalt der Startdatei des Autors:

```
 mode(-1);
// eigene Startup-Datei fuer Scilab
chdir('/Users/alexander/scilab_w')
write(%io(2),'Arbeitsverzeichnis ist scilab_w')
exec('mathlib/loader.sce',-1)
//============================================
```

Sie können in der Startdatei auch Variablen definieren, z.B. können Sie durch `pi=%pi` ein Umgehen mit π wie bei MATLAB erreichen.

3 Konstante, Variable und elementare Funktionen

```
-->%pi
   %pi =
   3.1415927
   -->cos(%pi/2)
   ans =
   6.123E-17
```

Rundungsfehler! Scilab rechnet numerisch! Die elementaren Funktionen heißen wie üblich. Ausnahme: `log` für den natürlichen Logarithmus, `cotg` statt cot und `asin, acos, atan` für die Arcusfunktionen sowie `asinh, acosh, atanh` für die Areafunktionen. Nach dem Einbinden der Hilsprogramme durch `exec('Beispiel.sci')` stehen sie auch unter dem sonst üblichen Namen zur Verfügung.
`%i` ist die imaginäre Einheit, Potenzen a^p werden durch `a^p` ausgerechnet:

```
   -->%i
   %i =
   i
   -->%i^2
   ans =
   - 1.
```

`%e` ist die Eulersche Zahl:

```
   -->%e
   %e =
   2.7182818
```

`exp(x)` bewirkt (bis auf Rundungsfehler) dassselbe wie `%e^x`:

```
   %e^2.5-exp(2.5)
   ans =
   - 1.776E-15
```

Die Wurzel erhält man durch `sqrt`, gegebenenfalls erhält man den komplexen Hauptwert der Wurzel:

```
   -->sqrt(-1)
```

```
ans =
i
```

Beispiele für das Rechnen mit komplexen Zahlen::

```
-->
w=-4+3*%i
w =
- 4. + 3.i
-->
abs(w) // absoluter Betrag
ans =
5.
-->
real(w) // Realteil
ans =
- 4.
-->
imag(w) // Imaginärteil
ans =
3.
-->
conj(w) // konjugiert komplexe Zahl
ans =
- 4. - 3.i
-->
z=-1-%i
z =
- 1. - i
-->
[r,phi]=polar(z) // Polarkoordinaten (Absolutbetrag u. Phase)
phi =
- 2.3561945
r =
1.4142136
-->
exec('Beispiel.sci');// Einbinden der Hilfsprogramme
-->
arg(z) // Phase
ans =
- 2.3561945
-->
ln(z) // Hauptwert des Logarithmus
ans =
.3465736 - 2.3561945i
```

Wie diese Beispiele zeigen, brauchen Variable nicht deklariert zu werden, man weist ihnen einfach einen Wert zu. Bei unerklärlichen Fehlermeldungen kann es manchmal hilfreich sein, eine Variable vor ihrer Weiterbenutzung durch `clear` *Variablenname* zu löschen.
Das Kommando „`who`" liefert eine Liste aller Variablen einschließlich derer, die von Scilab vordefiniert sind (z.B. `%pi`). Das Kommando „`clear`" ohne Angabe einer Variablen macht

alle vorgenommenen Deklarationen und Zuweisungen rückgängig, danach hat man nur die von Scilab vordefinierten Größen zur Verfügung:

```
-->clear
-->who
your variables are...
scicos_pal with_tk demolist %helps LANGUAGE MSDOS
home PWD TMPDIR xdesslib percentlib polylib
intlib elemlib utillib statslib alglib siglib optlib
autolib roblib soundlib metalib armalib tkscilib tdcslib
s2flib mtlblib SCI %F %T %z %s
 %t %f %eps %io  using 5292 elements out of 1000000.
and 47 variables out of 1791
your global variables are...
LANGUAGE %helps demolist %browsehelp %toolboxes
 using 1471 elements out of 1661.
and 7 variables out of 255
```

4 Grafische Darstellung elementarer Funktionen und komplexer Zahlen

Nach Einbinden der kleinen Funktionsbibliothek durch
`exec('Beispiel.sci')`
steht durch das Kommando

`eplot('`*arithmetische Anweisung*`',`$[a,b]$`,n)`

die Möglichkeit zur Verfügung, sich ganz einfach grafische Darstellungen elementarer Funktionen zu erzeugen. Die arithmetische Anweisung muss der Scilab-Syntax entsprechen. Die Variable muss x genannt sein. Das Intervall ist in der Form $[a, b]$ anzugeben. Der letzte Parameter n legt die Zahl der Plotpunkte fest. Er kann weggelassen werden, dann wird er automatisch auf $n = 400$ gesetzt. Mögliche Aufrufe sind also

```
eplot('sin(9*x)+sin(10*x)',[0,20])
eplot('5-8*x^2+x^3',[-3,9],300)
eplot('1/x',[0.25,4])
eplot('exp(-x^2)',[-5,5])
eplot('sin(x)/x',[-20,20])
```

Dabei hat man beim letzten Aufruf Glück gehabt, dass die Plotpunkte gerade so gewählt wurden, dass keine Division durch 0 auftritt. Sonst erfolgt nämlich ein Abbruch durch Fehlermeldung. Man hätte hier besser die Anweisung „`sin(x)/x`“ durch das unter Scilab bereits definierte „`sinc(x)`“ ersetzt (dann wird für $x = 0$ automatisch der Funktionswert 1 genommen). Rationale Funktionen werden allerdings sehr unschön dargestellt, wenn Pole im angegebenen Intervall liegen und wenn die Nullstelle im Nenner nicht „getroffen“ wird.

Zur grafischen Darstellung von Funktionen zweier Variabler steht in `Beispiel.sci` die Hilfsfunktion

`eplot2('`*Anweisung m.* `x` *und* `y', [`*min., max. x-Wert; min., max. y-Wert* `])`

zur Verfügung. So wird beispielsweise durch `eplot2('%e^(-x^2-y^2)',[-2,2;-2,2])` eine grafische Darstellung der Funktion $f(x, y) = e^{-x^2-y^2}$ erzeugt (siehe auch das Beispiel in Abb. 1). Ein dritter Parameter kann die Zahl der Plotpunkte (je Achse) verändern.

Wird er wie im obigen Beispiel weggelassen, so ist diese mit 20 festgelegt (das führt zu 400 Plotpunkten in der xy-Ebene).

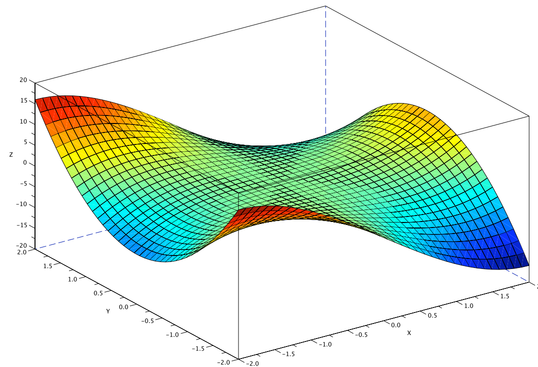


Abbildung 1: grafische Darstellung der Funktion $f(x, y) = x^3 - 3xy^2$ mit der Funktion `eplot2`

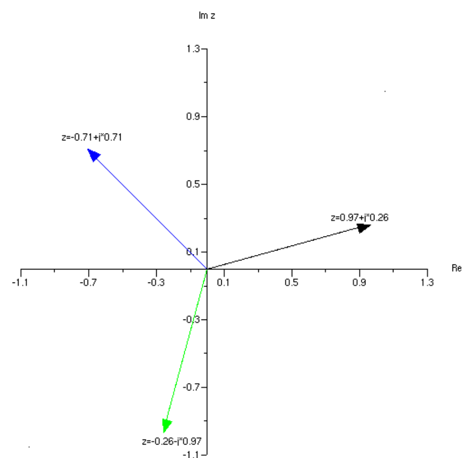
Nach Einbinden der Bibliothek `Beispiel.sci` kann eine komplexe Variable `z` durch das Kommando `zeiger(z)` als Zeiger in der Gaußschen Zahlenebene grafisch dargestellt werden. Es kann auch eine komplexe Konstante übergeben werden, beispielsweise

```
zeiger(-0.5-0.7*%i)
```

Mehrere komplexe Zahlen können in eckigen Klammern, durch Kommas getrennt, an das Kommando `zeiger` übergeben werden, die Zahl ist dabei nicht begrenzt. Sie werden dann in einer grafischen Darstellung abgebildet, beispielsweise erhält man durch

```
z1=%e^(%i*%pi/12)
z2=%e^(%i*9*%pi/12)
z3=%e^(%i*17*%pi/12)
zeiger([z1,z2,z3])
```

das Bild



5 Vektoren

Vektoren werden in eckigen Klammern eingegeben:

```
a=[-1; 2; -4] // Eingabe mit eckigen Klammern
```

```
 a  =

! - 1. !
!   2. !
! - 4. !
```

Auf die Komponenten kann einfach zugegriffen werden:

```
-->a(3)
ans =
- 4.
```

Die Komponenten können auch einfach abgeändert werden:

```
a(2)=7  // \"Anderung von Komponenten

 a  =

! - 1. !
!   7. !
! - 4. !
```

`norm(a)` berechnet die Länge (den Betrag) $|\vec{a}| = \sqrt{\sum_{k=1}^{3} a_k^2}$

```
-->norm(a)
ans =
8.1240384
```

Multiplikation mit einem Sklalar und Addition erfolgt so, wie man es erwartet:

```
-->
-2*a  // Multiplikation mit einem Skalar
 ans  =

!   2.  !
! - 14. !
!   8.  !

-->
b=[1; -5; 3]
 b  =

!   1. !
! - 5. !
!   3. !

-->
a+b // Addition von zwei Vektoren
 ans  =

!   0. !
!   2. !
! - 1. !
```

Skalarprodukt (gehört zu Scilab) und Vektorprodukt bei den Hilfsprogrammen

```
a'*b // Skalarprodukt
 ans  =

  - 48.
vecprod(a,b) // Vektorprodukt
 ans  =

!   1. !
! - 1. !
! - 2. !
```

Statt Spaltenvektoren kann man auch Zeilenvektoren benutzen (Trennung durch Leerzeichen oder Komma):

```
c=[2 3 -4.5]// Zeilenvektor
 c  =

!   2.    3.  - 4.5 !
c(3) // Zugriff auf die 3. Komponente
 ans  =

  - 4.5
d=[-2,-1,4] // Zeilenvektor
 d  =

! - 2.  - 1.    4. !
```

Aber ein Zeilenvektor kann nicht zu einem Spaltenvektor addiert werden (genausowenig können Vektoren unterschiedlicher Länge addiert werden, dies ergibt die Fehlermeldung `"inconsistent addition"`).

6 Matrizen

Matrizen werden zeilenweise eingegeben:

```
-->A=[1 2 0;0 -2 2]
 A  =

!   1.    2.    0. !
!   0.  - 2.    2. !

-->B=[5 0;0 2;1 -1]
 B  =

!   5.    0. !
!   0.    2. !
!   1.  - 1. !
```

Matrixmultiplikation mit „*“:

```
-->A*B
 ans  =

!   5.    4. !
!   2.  - 6. !

-->B*A
 ans  =

!   5.    10.    0. !
!   0.  - 4.     4. !
!   1.    4.   - 2. !
```

Matrix angewandt auf Vektor ebenfalls als Matrixmultiplikation

```
-->A*c
ans =
! 4. !
! 4. !
```

`size` liefert einen Vektor, dessen erste Komponente die Zahl der Zeilen und dessen zweite Komponente die Zahl der Spalten enthält; diese können getrennt abgefragt werden:

```
-->s=size(A)
s =
! 2. 3. !
-->s(2)
ans =
3.
```

Vektoren werden einfach als entsprechende Matrizen aufgefaßt:

```
-->size(b)
ans =
! 1. 3. !
-->size(c)
ans =
! 3. 1. !
```

Auf Matrixelemente kann zugegriffen werden:

```
-->A(2,3)
ans =
2.
```

und sie können verändert werden

```
-->B(2,1)=1
 B  =

!   5.    0. !
!   1.    2. !
!   1.  - 1. !
```

Ein Apostroph ' liefert die transponierte Matrix

```
-->B'
 ans  =

!   5.    1.    1. !
!   0.    2.  - 1. !
```

und macht aus einem Zeilenvektor einen Spaltenvektor

```
-->b
 b  =

!   1.  - 5.    3.5 !

-->b'
 ans  =

!   1.  !
! - 5.  !
!   3.5 !
```

`length` liefert die Zahl der Matrixelemente und damit die Zahl der Komponenten eines Vektors, unabhängig, ob Zeilen- oder Spaltenvektor:

```
-->length(A)
ans =
6.
-->length(c)
ans =
3.
-->length(c')
ans =
3.
-->
```

Die inverse Matrix erhält man durch `inv`:

```
-->C=[1 2 1;0 1 1;2 1 2]
 C  =

!   1.    2.    1. !
!   0.    1.    1. !
!   2.    1.    2. !

-->inv(C)
 ans  =

!   0.3333333  - 1.    0.3333333 !
!   0.6666667    0.  - 0.3333333 !
! - 0.6666667    1.    0.3333333 !
```

Einen Zeilenvektor einer Matrix erhält man durch

```
-->C(3,:)
 ans  =

!   2.    1.    2. !
```

und einen Spaltenvektor durch

```
-->C(:,2)

 ans  =

!   2. !
!   1. !
!   1. !
```

Die **Determinante** einer quadratischen Matrix erhält man durch `det(A)`, den **Rang** einer Matrix erhält man durch `rank(A)`.

7 Lineare Gleichungssysteme

Zur Lösung von linearen Gleichungssystemen ist zunächst die linke Seite als Matrix und die rechte Seite als Spaltenvektor einzugeben. Zur Lösung des Gleichungssystems

$$\begin{array}{rrrcl} 2x_1 & +3x_2 & -5x_3 & = & 10 \\ 4x_1 & +8x_2 & -3x_3 & = & 19 \\ -6x_1 & +x_2 & +4x_3 & = & 11 \end{array}$$

hat man also einzugeben

```
A=[2 3 -5;4 8 -3;-6 1 4],b=[10;19;11]
```

Zur Lösung des linearen Gleichungssystems gibt es zwei Kommandos

```
-->x=A\b
 x  =
  - 2.
    3.
  - 1.

-->y=linsolve(A,-b)
 y  =
  - 2.
    3.
  - 1.
```

Beachten Sie das Minuszeichen beim Aufruf von `linsolve` (diese Funktion geht davon aus, dass beim Gleichungssystem alles auf die linke Seite gebracht ist, die Zahlen auf der rechten Seite ändern dabei ihr Vorzeichen). Sie sollten mit der Lösung `x` bzw. `y` linearer Gleichungssysteme unbedingt den Rechner die Probe durchführen lassen:

```
-->rx=A*x-b
 rx  =
    0.
    0.
    0.
-->ry=A*y-b
 ry  =
    0.
  - 3.553E-15
    5.329E-15
```

Der Vektor `rx` bzw. `ry` darf nur Nullen als Komponenten enthalten, wenn die Lösung korrekt ist. Beachten Sie, dass dies aufgrund der unvermeidlichen Rundungsfehler nicht immer möglich ist, aber überlegen Sie, ob die auftretenden Zahlen als Rundungsfehler akzeptiert werden können! Bei sehr vielen Komponenten kann man sich den Betrag (geometrisch die Länge) durch `norm(rx)` ausrechnen lassen. Ohne Rundungsfehler sollte der Betrag 0 sein!

```
-->A=[0 3 1;-1 2 5;3 0 -13];b=[3;-7;22]; // System unloesbar
-->x=A\b
 warning
 matrix is close to singular or badly scaled. rcond =    3.0935E-18
computing least squares solution. (see lsq)
 x  =
    0.
    1.3351648
  - 1.7197802
-->rx=A*x-b
 rx  =
  - 0.7142857
    1.0714286
    0.3571429
-->norm(rx)
 ans  =
    1.3363062
-->y=linsolve(A,-b)
WARNING:Conflicting linear constraints!
 y  =
     []
```

Bei singulären Systemen erhält man bei der Berechnung von `A\b` eine entsprechende Warnung „matrix is close to singular or badly scaled“, dann wird bei Version 4.0 eine Lösung ausgegeben, die `norm(A*x-b)` zum Minimum macht; das Minimum muss aber nicht bei 0 liegen, wie die Probe zeigt! `linsolve` gibt einen leeren Vektor als Lösung aus, wenn das System unlösbar ist. Zur Bestimmung der Lösungsmenge bei Systemen mit unendlich vielen Lösungen (auch bei unterbestimmten Systemen) benutzt man `linsolve` mit zwei Rückgabewerten: `[x,y]=linsolve(A,-b)`. Wenn man einen Vektor **y** zurückerhält, dann ist die Lösungsmenge des Gleichungssystems

$$\mathbb{L} = \{\mathbf{x} + t \cdot \mathbf{y} \mid t \in \mathbb{R} \text{ beliebig } \}$$

```
-->A=[3 2 -1;2 -1 3;1 3 -4];b=[2;0;2];
-->[x,y]=linsolve(A,-b)
 y  =
    0.3580574
  - 0.7877264
  - 0.5012804
 x  =
    0.4102564
    0.2974359
  - 0.1743590
```

Für dieses Beispiel (aus Abschnitt 3.2 des Skriptes zur Linearen Algebra) erhält man also als Lösungsmenge die Menge der Punkte auf der Geraden

$$\begin{pmatrix} 0.4102564 \\ 0.2974359 \\ -0.1743590 \end{pmatrix} + t \begin{pmatrix} 0.3580574 \\ -0.7877264 \\ -0.5012804 \end{pmatrix}$$

Auch hier kann man für einzelne Zahlenwerte von t die Probe machen:

```
-->norm(A*(x+5*y)-b)
 ans  =
    1.884E-15
```

Beachten Sie, dass bei der vektoriellen Geradengleichung die beiden Vektoren nicht eindeutig bestimmt sind (und die hier numerisch ermittelten Vektoren nicht mit den im Skript angegebenen übereinstimmen). Der Richtungsvektor $\mathbf{y}$ ist ein Vielfaches des im Skript angegebenen:

```
-->norm(y-y(3)*[-5/7;11/7;1])
 ans  =
    1.110E-16
```

Wenn nach dem Aufruf von `[x,y]=linsolve(A,-b)` die zweite Rückgabegröße `y` eine Matrix mit mehreren Spaltenvektoren $\mathbf{y}_1$, $\mathbf{y}_2, \ldots \mathbf{y}_k$ ist, dann ist die Lösungsmenge

$$\mathbb{L} = \{\mathbf{x} + t_1 \cdot \mathbf{y}_1 + t_2 \cdot \mathbf{y}_2 + \cdots + t_k \cdot \mathbf{y}_k \mid t_1, t_2, \ldots t_k \in \mathbb{R} \text{ beliebig } \}$$

Die Spaltenvektoren erhält man in Scilab durch `y(:,1)`, `y(:,2)`,....

```
-->A=[-6 3 -12;2 -1 4;-10 5 -20];b=[-7.5;2.5;-12.5]
 b  =
  - 7.5
    2.5
  - 12.5
-->[x,y]=linsolve(A,-b)
 y  =
    0.8553948  - 0.2789689
  - 0.2001644  - 0.9551519
  - 0.4777385  - 0.0993036
 x  =
    0.2380952
  - 0.1190476
    0.4761905
```

Die Lösungsmenge ist hier also gegeben durch die Punkte der Ebene

$$\begin{pmatrix} 0.2380952 \\ -0.1190476 \\ 0.4761905 \end{pmatrix} + t_1 \begin{pmatrix} 0.8553948 \\ -0.2001644 \\ -0.4777385 \end{pmatrix} + t_2 \begin{pmatrix} -0.2789689 \\ -0.9551519 \\ -0.0993036 \end{pmatrix}$$

und man kann die Probe für bestimmte Zahlenwerte der Parameter machen, z.B. für $t_1 = 5$ und $t_2 = -7$

```
-->norm(A*(x+5*y(:,1)-7*y(:,2))-b)
 ans  =
    2.709E-14
```

Auf diese Weise kann man mit `linsolve` auch die Lösungsmenge unterbestimmter Gleichungssysteme bestimmen.

8 Eigenwerte und Eigenvektoren

Der Aufruf von `spec(A)` liefert die Eigenwerte der quadratischen Matrix `A` als Spaltenvektor zusammengefaßt. Wenn die Matrix `A` diagonalisierbar ist, so erhält man durch

```
-->[D,X]=bdiag(A)
```

die zugehörige Diagonalmatrix `D`, die entlang der Hauptdiagonale die Eigenwerte von `A` enthält. Weiterhin erhält man die Matrix `X`, die durch

$$\mathbf{D} = \mathbf{X}^{-1}\mathbf{A}\mathbf{X}$$

die Diagonalmatrix aus der ursprünglichen Matrix erzeugt. Diese Matrix `X` enthält in ihren Spalten die Eigenvektoren von `A`, d.h. in der 1. Spalte von `X` steht der Eigenvektor zum Eigenwert D_{11}, in der 2. Spalte steht der Eigenvektor zum Eigenwert D_{22} usw. Im Programm Scilab erhält man die Eigenvektoren also durch

```
-->X(:,1) // Eigenvektor zu D(1,1)
```

analog für die andern Eigenvektoren. So erhält man für die Matrix

$$\mathbf{A} = \begin{pmatrix} 3 & 2 & 2 & 2 \\ -10 & -6 & 3 & -11 \\ 0 & 0 & 1 & -12 \\ 0 & 0 & 1 & 8 \end{pmatrix}$$

die Matrizen

$$\mathbf{X} = \begin{pmatrix} 1.4142136 & 11.313708 & -.6246110 & 2.8284271 \\ -2.8284271 & -28.284271 & .2180246 & 0. \\ 0. & 0. & -.7071068 & 4.2426407 \\ 0. & 0. & .1767767 & -1.4142136 \end{pmatrix}$$

und

$$\mathbf{D} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix}$$

Die 4. Spalte von **X** enthält also den Eigenvektor zum Eigenwert 5. Man beachte, dass bei einer Probe durch

```
A*X(:,4)-D(4,4)*X(:,4)
```

man nicht den Nullvektor erhält, sondern durch Rundungsfehler einen Vektor mit Komponenten, deren Betrag sehr klein ist (versions- und rechnerabhängig, Größenordnung 10^{-14}). Ebenso erhält man durch

```
inv(X)*A*X
```

nicht exakt, sondern nur bis auf Rundungsfehler die von der Funktion `bdiag` gelieferte Matrix `D`.

Bei komplexen Eigenwerten einer ansonsten reellen Matrix `A` ist die Matrix `D` nicht diagonal, sondern enthält Blöcke von 2×2-Matrizen, die dieselben Eigenwerte haben wie jeweils komplex konjugierte Eigenwerte von `A`. Da dies nicht die gewünschten Eigenvektoren liefert, hilft man sich in diesem Fall, indem man die Matrix für Scilab einfach komplex macht, indem man statt `A` die Matrix `A+0*%i` eingibt. Dann enthält `X` die komplexen Komponenten der Eigenvektoren.

Wenn die Matrix `A` nicht diagonalisierbar ist, dann liefert `[D,X]=bdiag(A)` eine Matrix `D`, die nicht diagonal ist, also außerhalb der Hauptdiagonale nichtverschwindende Matrixelemente hat. Dann stehen in den entsprechenden Spalten von `X` keine Eigenvektoren. So erhält man für

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$$

die Matrizen

$$\mathbf{X} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ und } \mathbf{D} = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$$

die

$$\mathbf{D} = \mathbf{X}^{-1}\mathbf{A}\mathbf{X}$$

erfüllen. Wir haben $\det(\mathbf{A}-\lambda\mathbf{E}) = (2-\lambda)^2$, also ist $\lambda = 2$ algebraisch zweifacher Eigenwert von **A**. Doch nur in der ersten Spalte von **X** steht ein Eigenvektor. Alle Eigenvektoren zum Eigenwert 2 sind Vielfache dieses Eigenvektors, der Vektor der zweiten Spalte von **X** ist kein Eigenvektor von **A**.

9 Wie Vektoren und Matrizen einfach erzeugt werden

Matrizen und Vektoren komponentenweise zu berechnen, ist mühsam und zeitaufwendig. Scilab stellt eine Reihe nützlicher Makros zur Verfügung. So für die Einheitsmatrix

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

den Nullvektor (aufgefaßt als Matrix!)

```
-->x0=zeros(1,4)
x0 =
! 0. 0. 0. 0. !
```

einen Vektor mit lauter Einsen (und so mit konstanten Komponenten)

```
-->c=ones(1,4)
c =
! 1. 1. 1. 1. !
-->c2=-0.2*c
c2 =
! - .2 - .2 - .2 - .2 !
```

Linear ansteigende (oder abfallende) Werte erhält man durch

`linspace(`*erster Wert, letzter Wert, Zahl der Komponenten* `)`

```
-->x=linspace(0,1,6)
x =
! 0. .2 .4 .6 .8 1. !
```

Man kann auch die Schrittweite festlegen durch die Eingabe

Startwert:*Schrittweite*: *nicht zu überschreitender/unterschreitender Wert*

```
-->x2=0:0.3:2
 x2  =

!   0.      .3      .6      .9    1.2    1.5    1.8 !

-->x3=1:-0.2:0
 x3  =

!   1.      .8      .6      .4      .2 !
```

Bei `x3` hat einem bereits der Rundungsfehler einen Streich gespielt, für die letzte Komponente 0.0, die man noch erwartet, wird die festgelegte Schranke bereits unterschritten! Läßt man die *Schrittweite* weg, so wird als Default-Wert 1 angenommen:

```
-->l=1:5
 l  =

!   1.    2.    3.    4.    5. !
```

Vektoren können aneinandergefügt werden („;“ verhindert die Ausgabe am Schirm)

```
-->u1=[2 1]; u2=[-1 0]; u3=[17 4];
-->u=[u1 u2 u3]
 u  =
!   2.    1.  - 1.    0.    17.    4. !

-->v1=[2;-1;3]; v2=[0;-5;2]; v3=[17;4;21];

-->vc=[v1;v2;v3]
 vc  =

!   2.  !
! - 1.  !
!   3.  !
!   0.  !
```

```
! - 5.  !
!   2.  !
!   17. !
!   4.  !
!   21. !
```

Spaltenvektoren können zu Matrizen zusammengefügt werden;

```
-->Vm=[v1 v2 v3]
 Vm  =

!   2.     0.     17. !
! - 1.   - 5.     4.  !
!   3.     2.     21. !
```

10 Elementweise Operationen

Matrizen (und damit Vektoren) können als Argumente in elementare Funktionen eingesetzt werden. Das ist insbesondere für Vektoren nützlich.

```
-->x=0:%pi/6:2*%pi; // ";" unterdrueckt die Ausgabe des Ergebnisses
-->y=sin(x);
-->x(2),y(2)  // 2 Befehle in derselben Zeile durch "," zu trennen
 ans  =
     .5235988
 ans  =
     .5
```

Vorsicht ist bei Multiplikation und Division angebracht. Diese werden von Scilab grundsätzlich **nicht** element- oder komponentenweise interpretiert. Der Operator `*` wird stets als Matrixmultiplikation verstanden. Wenn `x` ein Spaltenvektor, also eine $(n \times 1)$-Matrix ist, dann erhält man durch `x'*x` das Skalarprodukt des Vektors mit sich selbst, durch `x*x'` eine aus `x` konstruierte $(n \times n)$-Matrix vom Rang 1 und bei Eingabe von `x*x` die Fehlermeldung „Inkonsistente Multiplikation“. Element- bzw. komponentenweise Multiplikation bei Vektoren erfordert zwingend den Zusatz eines Punktes, also durch `.*`

```
-->a=[1 0 -3]; b=[2 1 4]; a.*b
ans =
! 2. 0. - 12. !
```

erhält man das komponentenweise Produkt.

Der Schrägtstrich `/` stellt eine besonders gefährliche „Falle“ dar. Hier bestimmt Scilab die Lösung eines **Gleichungssystems** (gegebenenfalls die Näherungslösung). Wenn **b** ein Zeilenvektor ist (also eine $(1 \times n)$-Matrix, dann erhält man als Ergebnis von `1/b` den Spaltenvektor $\frac{1}{\mathbf{b}\mathbf{b}^{\mathrm{T}}}\mathbf{b}^{\mathrm{T}}$, also dasselbe Ergebnis wie mit `b'/(b*b')`. Dies erklärt nach Eingabe von `b=[2 1 4];1/b` die Ausgabe

```
 ans  =
    0.0952381
    0.0476190
    0.1904762
```

Auch bei Spaltenvektoren im „Nenner“ erhält man ähnlich überraschende Ergebnisse. So liefert `b=[2 1 4];1/b'` die Ausgabe

```
ans  =
    0.    0.    0.25
```

Man könnte meinen, die Eingabe von `1./b` liefert die komponentenweisen Kehrwerte, aber weit gefehlt: man erhält dieselbe Ausgabe wie bei `1/b`. Dies ist eine Besonderheit von Scilab und liegt an der Priorität der Zuordnung des Punktes als Dezimalpunkt, denn `1./b` wird als `(1.)/b` und damit als `1.0/b` interpretiert. Um tatsächlich die komponentenweisen Kehrwerte zu erhalten, muss man `1.0./b` oder `1 ./b` oder (1)./b eingeben. Ergebnis

```
-->1.0./b
ans =
! .5 1. .25 !
```

„Normale“ Potenzen (ohne Zusatz eines Punktes) sind nur für Skalare und quadratische Matrizen definiert. Zur Berechnung der punktweisen Potenz ist analog zur Multiplikation zusätzlich ein Punkt anzugeben.

```
-->b^.2
 ans  =
    4.    1.    16.
```

Beachten Sie, dass bei quadratischen Matrizen die punktweise Potenz ein anderes Ergebnis liefern kann als die „normale“, die über die Matrixmultiplikation definiert ist. Für

```
-->A=[0 2;0 0]
 A  =
    0.    2.
    0.    0.
```

erhält man als Ergebnis der „normalen“ Potenz

```
-->A^2
 ans  =
    0.    0.
    0.    0.
```

und als Ergebnis der punktweisen Potenz

```
-->A.^2
 ans  =
    0.    4.
    0.    0.
```

Bei früheren Versionen von Scilab wurde bei nichtquadratischen Matrizen beim Weglassen des Punktes die punktweise Potenz gebildet (die „normale“ ist dann nicht definiert). Ab Version 5.5.2 führt in diesem Fall das Weglassen des Punktes zu einer Fehlermeldung.

Real- und Imaginärteil sowie der konjugiert-komplexe Wert können elementweise gebildet werten, indem man den ganzen Vektor als Argument in die entsprechende Funktion einsetzt. `real` bildet den Real-, `imag` den Imaginärteil `conj` das Konjugiert-Komplexe der Komponenten des Vektors:

```
-->x=0:2*%pi/3:2*%pi
```

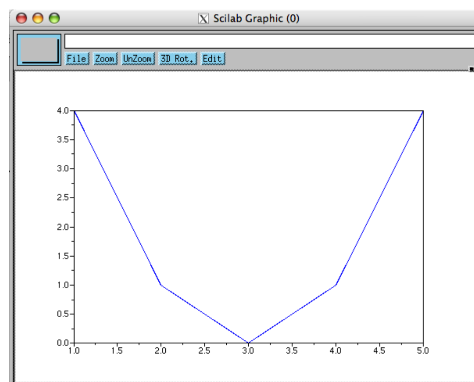
```
x =
! 0. 2.0943951 4.1887902 6.2831853 !
-->z=exp(%i*x)
z =
! 1. - .5 + .8660254i - .5 - .8660254i 1. - 2.449E-16i !
-->real(z)
ans =
! 1. - .5 - .5 1. !
-->imag(z)
ans =
! 0. .8660254 - .8660254 - 2.449E-16 !
-->conj(z)
ans =
! 1. - .5 - .8660254i - .5 + .8660254i 1. + 2.449E-16i !
```

11 Grafik

Die wichtigste Funktion für zweidimensionale Grafik, `plot`, ist beim Übergang zu Version 4.0 in ihrem Verhalten weitgehend an die entsprechende Funktion von MATLAB angeglichen worden (abweichend zu früheren Versionen). Ein Zeilenvektor

```
-->y=(-2:2)^2
y =
! 4. 1. 0. 1. 4. !
```

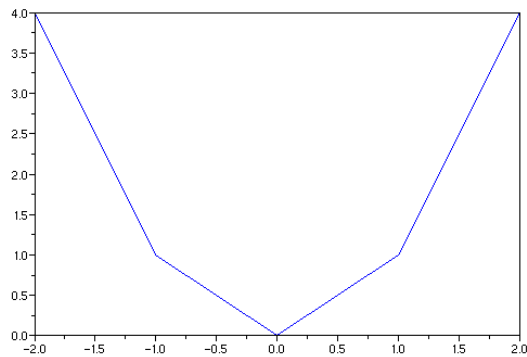
kann durch `plot(y)` grafisch dargestellt werden. Man erhält das folgende Grafikfenster:



Grafisch dargestellt werden in x-Richtung die Nummern der Komponenten des Vektors (hier 1 bis 5), in y-Richtung die Werte der Komponenten. Die dadurch entstehenden Punkte werden durch Geraden verbunden. Maßstab und Beschriftung werden automatisch festgelegt. Die Grafik kann durch `clf()` oder `clf` wieder gelöscht werden. Erneutes Aufrufenn von `plot` löscht die vorherige Grafik nicht, die neue Grafik wird zusätzlich zur alten gezeigt. Bei Angabe von zwei Vektoren x, y werden die Punkte $\big(x(k), y(k)\big)$ dargestellt, d.h. durch Geraden verbunden, d.h.

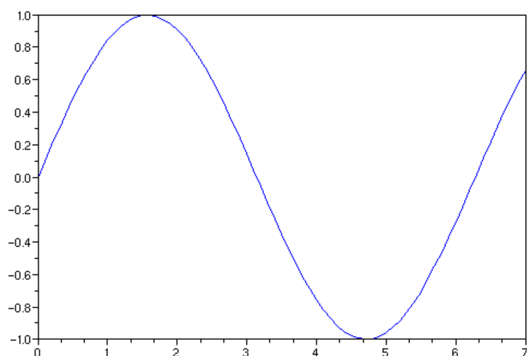
```
-->x=-2:2
x =
! - 2. - 1. 0. 1. 2. !
```

```
-->plot(x,y)
```
liefert



Durch Berechnen einer dichter liegenden Menge von Punkten kann man Funktionen grafisch darstellen:

```
-->x=0:0.1:7; y1=sin(x);plot(x,y1)
```



Die Funktion `plot` ist beim Aufruf mit einem oder zwei Vektoren als Argument völlig unempfindlich, ob diese Vektoren Zeilen- oder Spaltenvektoren sind. Beim Aufruf mit zwei Vektoren müssen diese nur gleich lang sein. Man kann ein Bild durch mehrfachen Aufruf aufbauen, da das alte nicht gelöscht wird:

```
y2=cos(x); plot(x,y1); plot(x,y2)
```

Wenn man mehrere Kurven durch einen einzigen Aufruf in einem Bild darstellen will, dann müssen die darzustellenden Daten als Spaltenvektoren zu einer Matrix „zusammengebaut" werden. Da durch `x=0:0.1:7;` und `y1=sin(x);` sowie `y2=cos(x);` Zeilenvektoren entsehen, müssen diese transponiert werden (mit einem Apostroph). Man erhält also durch `plot([y1',y2'])` dieselben Kurven wie durch einen nacheinandererfolgten Aufruf `plot(y1);plot(y2);`, allerdings werden beim Aufruf mit einer Matrix die beiden Kurven in unterschiedlichen Farben dargestellt.

Wenn man in waagrechter Richtung nicht die Nummern der Komponenten der Vektoren, sondern die zugehörigen x-Werte dargestellt haben will, dann muss man diese durch einen Spaltenvektor als erstes Argument an `plot` übergeben. Dieser muss genauso viele Komponenten haben wie die Spaltenvektoren der Matrix, die im zweiten Argument übergeben wird. In unserem Beispiel erhält man also die grafische Darstellung von $\sin(x)$ und $\cos(x)$ durch den Aufruf `plot(x',[y1',y2'])`; dann wird auch akzeptiert, dass die Koordinaten als Zeilenvektoren übergeben werden (was für `plot` am ersten Argument

ersichtlich ist). In unserem Beispiel erzeugt also `plot(x,[y1;y2])` dieselbe Grafik (beachten Sie das Semikolon zwischen `y1` und `y2`!). Wenn also das erste Argument von `plot` ein Zeilenvektor ist, dann müssen die Zeilenvektoren der Matrix im zweiten Argument genauso viele Komponenten haben.

Für grafische Darstellungen von Funktionen erzeugt man zunächst einen Vektor `x` für die Abtastwerte des Arguments (beispielsweise durch `x=linspace(-4,4,800);` oder mit dem Doppelpunkt-Operator, siehe Abschnitt 9). Dann kann man einen Vektor `y` mit den Funktionswerten erstellen. Hierzu ist von Vorteil, dass die meisten in Scilab bereits vorhandenen Funktionen Vektoren als Argument zulassen und dann die Funktionswerte komponentenweise berechnen und als Vektor zurückgeben. So kann man mit einem geeigneten Vektor `x` mit `y=sin(x)` die Funktionswerte berechnen und durch `plot(x,y)` grafisch darstellen. Bei selbstgeschriebenen Funktionsunterprogrammen oder einer direkten Berchnung mit arithmetischen Anweisungen sollte man darauf achten, dass man die arithmetischen Operationen punktweise definiert. Hier ist insbesondere die in Abschnitt 10 beschriebene „Falle" mit dem Operator `/` zu beachten! So erhält man durch `y=1/(1+x^2);` keinesfalls eine richtige Berechnung der Werte der Funktion $f(x) = \frac{1}{1+x^2}$, wenn der Eingabeparameter `x` ein Vektor ist! Die richtige grafische Darstellung dieser Funktion erhält man durch

```
x=linspace(-4,4,800);  y=1.0./(1+x^2); plot(x,y);
```

Die komponentenweise Berechnung der Funktionswerte mit Vektoren als Argumenten in einem selbstgeschriebenen Programm kann schwierig sein, beispielsweise wenn umfangreiche Fallunterscheidungen notwendig sind (auch bei Fallunterscheidungen ist eine vektorielle Berechnung möglich mit der Funktion `find`, siehe hierzu Abschnitt 15). „Zur Not"kann man die Funktionswerte in einer Schleife berechnen, wenn beispielsweise die Funktion `fun` nur Skalare als Eingabeparameter akzeptiert:

```
for k=1:length(x);y(k)=fun(x(k));end;
```

Da dies umständlich und rechenzeitaufwändig ist (Schleifen sind langsam), gibt es in Scilab zwei einfache Möglichkeiten, eine derartige Möglichkeit der Berechnung der Funktionswerte zu umgehen: Die Berechnung in der obigen Schleife kann ersetzt werden durch `y=feval(x,fun);` oder man übergibt den Funktionsnamen an die Funktion `plot` durch den Aufruf `plot(x,fun)`. Bei beiden Möglichkeiten wird die Funktion `fun` nur für skalare Argumente ausgewertet. So erhält man für die — nur für skalare `x` richtige — Berechnung von $f(x) = \frac{1}{1+x^2}$ durch

```
function [y]=fun(x); y=1/(1+x^2); endfunction;
```

sowie `x=linspace(-5,5,800); plot(x,fun)` oder `y=feval(x,fun); plot(x,y)` eine korrekte grafische Darstellung.

Durch Angabe eines weiteren Arguments, das eine Zeichenkette enthält, kann der Darstellungsstil verändert werden. So wird der Graph der Funktion mit `plot(x,y1,'r')` in **roter** Farbe dargestellt. Entsprechend wird durch `'b'` **blau**, durch `'g'` **grün** sowie durch `'k'` **schwarz** ausgewählt.

Durch andere Zeichen kann der Zeichenstil verändert werden. Durch `plot(x,y1,'--')` wird die Kurve **gestrichelt** dargestellt. Mit `plot(x,y1,'x')` werden statt einer durchgezogenen Kurve die durch die Komponenten von `x` und `y1` gegebenen Punkte durch eine Markierung in der Form eines $\times$ dargestellt. Dies kann für einzelne Messpunkte sowie für Abtastwerte oder allgemein diskrete Daten sinnvoll sein. Analog führt ein Punkt

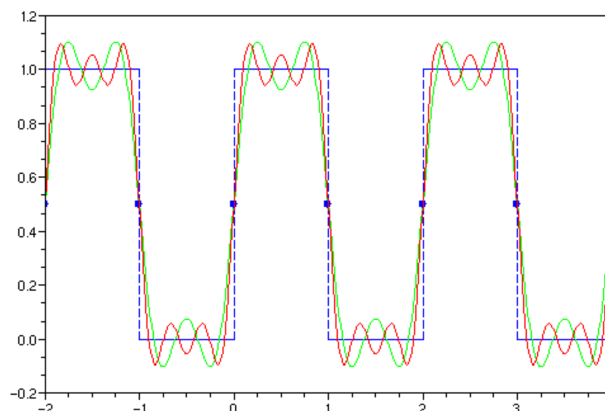
'.' zu Markierungen durch ausgefüllte Kreise •, die Buchstaben 'o' und 'O' führen zu Markierungen zu Kreisen der Form ∘.

Beide Arten von Parametern können zusammen angegeben werden. Dabei kommt es auf die Reihenfolge nicht an. So führen plot(x,y1,'rx') und plot(x,y1,'xr') beide zu einer Darstellung einzelner Punkte durch rote Kreuze ×.

Das folgende kleine Beispielprogramm erzeugt ein Bild, das die Annäherung einer Rechteckschwingung durch trigonometrische Polynome zeigt. Durch

```
xmin=-2;xmax=4;
for k=xmin:(xmax-1);// Rechteckschwingung
   y=0.5*(1+(-1)^k);
   plot([k,k+1],[y,y]); // waagrechte Linien
   plot([k,k],[0,1],'--'); //gestrichelte senkrechte Linien
   plot([k,k+1],[0.5,0.5],'.');// Funktionswerte an Sprungstellen
end; // for k
x=linspace(xmin,xmax,400); // Abtastpunkte
y3=0.5+2*sin(%pi*x)/%pi+2*sin(3*%pi*x)/(3*%pi);// trig. Polynom
y5=y3+2*sin(5*%pi*x)/(5*%pi); // bessere Naeherung
plot(x,y3,'g'); // 1. Naeherung in gruen
plot(x,y5,'r'); // bessere Naeherung in rot
```

entsteht das Bild



Wenn man statt des Standardrahmens ein Achsenkreuz im Ursprung wünscht, so kann man dies — nach Einbinden der Funktionen in Beispiel.sci oder nach Laden der Bibliothek mathlib — erreichen durch Aufruf der Funktion centax oder centax(). Diese Funktion kann vor oder nach der Funktion plot aufgerufen werden, sie muss allerdings nach jedem Löschen des Grafik-Fensters durch clf neu aufgerufen werden. Außerdem steht in Beispiel.sci bzw. in mathlib die Funktion putaxlab zur Verfügung, mit der an die mit centax zentrierten Achsen Beschriftungen angebracht werden können. Das folgende Beispiel zeigt ihren Gebrauch:

```
x=linspace(-4,4,400); plot(x,sin(x)); centax; putaxlab('x','y=sin(x)');
```

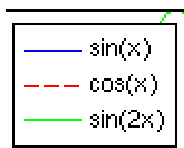
Mit dem Kommando xgrid(*Farbnummer*) vor oder nach dem Aufruf von plot erhält man ein Gitter von Parallelen zu den Achsen in der durch die positive ganze Zahl *Farbnummer* bestimmten Farbe. Welche Zahl zu welcher Farbe gehört, kann man sich durch das Kommando getcolor interaktiv anzeigen lassen. Wird *Farbnummer* weggelassen, dann sind die Gitterlinien schwarz.

Ein weiteres Grafik-Fenster kann man durch den Funktionsaufruf `scf;` öffnen und zum aktiven Grafik-Fenster machen (in dem die nächste Grafik erscheint). Scilab nummeriert die Grafik-Fenster automatisch von 0 an beginnend. Durch `scf(0);` kann man dann das ursprüngliche Fenster wieder zum aktiven erklären. Durch `clf(n)` kann man das Fenster mit der Nummer n löschen, durch `scf(n);` aktivieren. Die nächste grafische Darstellung erfolgt dann in diesem Fenster. Durch `show_window(n);` wird das Fenster mit der Nummer `n` vor die andern Fenster gestellt (als würde man es anklicken); mit `show_window();` wird das gerade aktive Grafikfenster nach vorne gestellt. So kann man mit verschiedenen Fenstern arbeiten.

Durch `xtitle('`*Abbildungstitel*`');` erhält die Grafik eine Überschrift. Hat man mehrere Kurven in unterschiedlichen Farben in einer Abbildung, dann ist eine Beschrifung mit dem Makro `legend` sehr nützlich, wie das folgende Beispiel verdeutlicht:

```
x=linspace(-4,4,400);
plot(x,sin(x)); plot(x,cos(x),'r--');plot(x,sin(2*x),'g');
legend('sin(x)','cos(x)','sin(2x)');
```

erzeugt in der rechten oberen Ecke des Bildes die folgende Beschriftung



Aus dieser Beschriftung werden die Farbe und der Darstellungsstil (wie gestrichelt) für die einzelnen Kurven ersichtlich. Entscheidend ist dafür, dass die Reihenfolge der Argumente von `legend` mit der Reihenfolge beim Erstellen der Grafik übereinstimmt. Die Reihenfolge ist auch entscheidend, wenn man das Bild mit einem einzigen Aufruf von `plot` durch Übergabe einer Matrix erstellt, also hier durch `plot(x,[sin(x);cos(x);sin(2*x)]);` dann ist die Reihenfolge der y-Werte in der Matrix maßgebend.

Oft ist es sinnvoll, ein Programm anzuhalten, um dem Benutzer Gelegenheit zu geben, die Grafik zu betrachten. Das Kommando `halt` unterbricht die Ausführung, bis der Benutzer das Kommandofenster aktiviert und dann die Eingabetaste betätigt. Etwas komfortabler ist das Kommando

```
x_dialog('weiter?','ja');
```

Es öffnet ein kleines Dialogfenster, und das Programm läuft erst weiter, wenn der Benutzer in „OK“ geklickt hat.

Man kann eine erzeugte Grafik abspeichern, indem man auf dem Feld „File“ in die Option „Export“ geht, und dann ein Dateiformat, beispielsweise „PNG“ auswählt sowie Dateinamen und Pfad festlegt. Man kann auch durch ein entsprechendes Kommando den Inhalt eines Grafikfensters abspeichern, was insbesondere in einem längeren Programm nützlich sein kann. So kann man durch

```
xs2png(0,'Ergebnisse/meinbild.png')
```

den Inhalt des Grafik-Fensters Nr. 0 als PNG-Datei im Unterverzeichnis `Ergebnisse` (Unterverzeichnis des aktuellen Arveitsverzeichnisses) abspeichern.

Zur Ergänzung wird hier noch die Routine `plot2d` vorgestellt, die keine entsprechende Funktion bei MATLAB hat. Mit einem Argument oder zwei Argumenten für die darzustellenden Daten wird dasselbe bewirkt wie bei `plot`, auch hierbei wird das alte Bild

durch den Aufruf nicht gelöscht. Durch Angabe von mehr als zwei Argumenten kann man den Stil der Darstellung verändern, allerdings nach etwas andern Regeln als bei `plot`. Mit dem Aufruf `plot2d(x,y1,axesflag=5)` erhält man statt des Standardrahmens ein Achsenkreuz (statt durch den gesonderten Aufruf von `centax`), was vielfach übersichtlicher ist.

Mit `plot2d` kann ein logarithmischer Maßstab gewählt werden. Durch Übergabe des Arguments `logflag='nl'` wird ein lorithmischer Maßstab für die y-Achse gewählt. Das Kommando

```
x=linspace(-20,20,800);plot2d(x,exp(x),logflag='nl');
```

bewirkt die grafische Darstellung einer Geraden. Entsprechend wird mit `logflag='ln'` ein logarithmischer Maßstab für die x-Achse gewählt. Für die Erstellung entsprechender grafischer Darstellungen ist die Funktion `logspace` nützlich; sie erzeugt einen Vektor mit logarithmisch gleich verteilten Punkten: `x=logspace(-3,3,5)` liefert einen Vektor mit 5 Punkten zwischen 10^{-3} und 10^{3}, nämlich

$$\mathbf{x} = \begin{pmatrix} 0.001 & 0.0316228 & 1 & 31.622777 & 1000 \end{pmatrix}$$

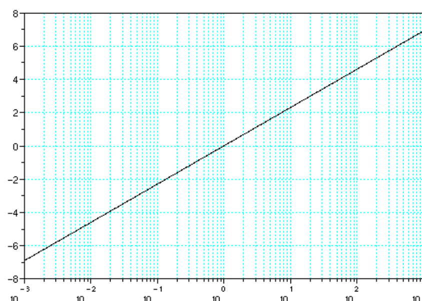
Der Logarithmus zur Basis 10 dieser Werte ist gleichverteilt zwischen -3 und $+3$:

$$\begin{pmatrix} -3 & -1.5 & 0 & 1.5 & 3 \end{pmatrix}$$

Mit den Kommandos

```
x=logspace(-3,3,800);plot2d(x,log(x),logflag='ln');xgrid(4);
```

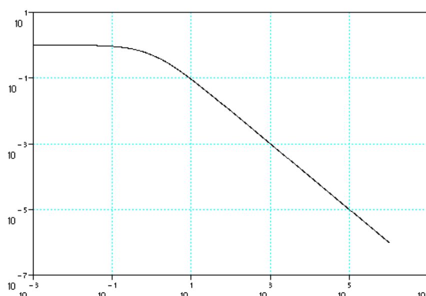
erzeugt man die folgende grafische Darstellung der Funktionswerte des natürlichen Logarithmus



Einen logarithmischen Masßstab in beide Achsrichtungen erhält man mit `logflag=ll`. So liefert

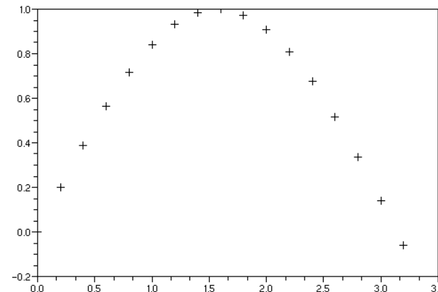
```
x=logspace(-3,6,800);y=1.0./(1+x);plot2d(x,y,logflag='ll');xgrid(4);
```

die folgende grafische Darstellung



Die Funktion `plot2d` hat sehr viele weitere Möglichkeiten. Es ist zu empfehlen, sich hierüber durch `help plot2d` zu informieren; hier noch ein Beispiel, was zur Darstellung von Messwerten nützlich sein kann:

```
-->x=0:0.2:3.2;y=sin(x); plot2d(x,y,style=-1)
```

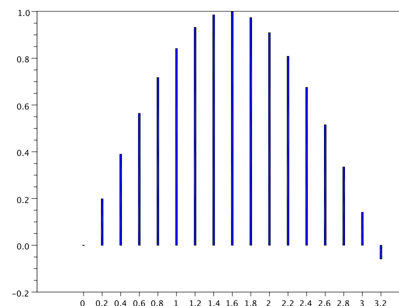


Bis auf die Farbe (`plot` wählt für die Markierung der Punkte blau) erhält man dasselbe Bild durch `plot(x,y,'+')`.

Von `plot2d` sind weitere Varianten verfügbar. Vertikale Striche (für diskrete Signale nicht unüblich) kann man durch `plot2d3(x,y,axesflag=5)` erhalten. Horizontale Striche bis zum nächsten Punkt zeichnet man mit `plot2d2(x,y,axesflag=5)`.

Für diskrete Werte kann auch ein **Balkendiagramm** sinnvoll sein. Dieses erhält man durch

```
x=0:0.2:3.2;y=sin(x);bar(x,y,0.1);
```



Der dritte Parameter von `bar` legt die Dicke der Balken fest; er kann weggelassen werden, dann erhält er den Default-Wert $0,8$.

Grafische Darstellungen von Funktionen von 2 Variablen erhält man sehr leicht nach Einbinden der Bibliothek `mathlib` mit dem Makro `eplot2`, wie schon in Abschnitt 4 erläutert wurde (siehe auch Abb. 1). Wenn man dieses Makro nicht benutzen möchte, dann muss man zur grafischen Darstellung zunächst eine Matrix mit den Matrixelementen $z_{ik} = f(x_i, y_k)$ für die Punkte erstellen, für die man die Fläche $z = f(x, y)$ grafisch darstellen möchte. Beachten Sie bei dieser Konvention, dass die Zeilennummer der Matrix zur Nummer der x-Koordinaten gehört. Dies kann man beispielsweise für $[-2, 2] \times [-2, 2]$ mit einem Gitter mit 20×20 Punkten und $z = x^3 - 3xy^2$ („Affensattel" genannt) durch

```
function [z]=fun(x,y);
  z=x^3-3*x*y^2;
endfunction; // Affensattel
//==================================
```

```
n=20; m=20;
x=linspace(-2,2,m); y=linspace(-2,2,n);
for i=1:m;
  for k=1:n
    z(i,k)=fun(x(i),y(k);
  end; // for i
end; // for k
```

erreichen. Man kann hier jedoch die beiden ineinandergeschalteten Schleifen elegant durch `z=feval(x,y,fun);` ersetzen!

Für die eigentliche grafische Darstellung wird hier die Funktion `surf` empfohlen, da sie fast dieselbe Syntax wie die entsprechende MATLAB-Funktion hat. Allerdings hat sie den Nachteil, dass sie — im Gegensatz zur Funktion `feval` — die Konvention benutzt, dass die Zeilennummer der Matrix `z` zu den y-Koordinaten gehört. Das muss dadurch kompensiert werden, dass man die transponierte Matrix `z'` übergibt. Man erzeugt also durch `surf(x,y,z')` die grafische Darstellung. Allerdings sind die Farben so nicht besonders geeignet. Eine wesentliche Verbesserung erhält man durch vorheriges Aufrufen von `xset("colormap",jetcolormap(100));`. Hier also zusammen die Anweisungen, mit denen man eine grafische Darstellung der durch `fun` gegebenen Funktion erhalten kann (so ist auch das Quellprogramm von `eplot2` aufgebaut):

```
n=20;m=20;
x=linspace(-2,2,m); y=linspace(-2,2,n);
z=feval(x,y,fun);
xset("colormap",jetcolormap(100));
surf(x,y,z'); // transponierte Matrix zu uebergeben
```

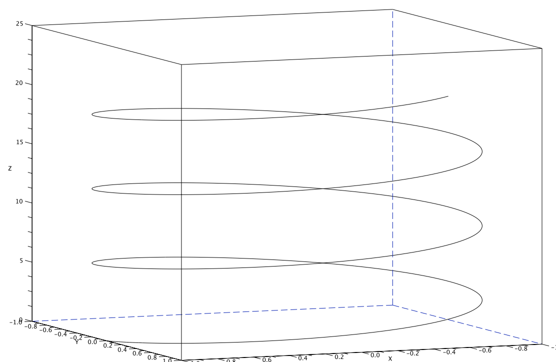


Abbildung 2: Beispiel für eine Kurve in $\mathbb{R}^3$

Kurven in $\mathbb{R}^3$, die durch eine Parameterdarstellung der Form $x = f_1(t)$, $y = f_2(t)$, $z = f_3(t)$, gegeben sind, können mit der Funktion `param3d` dargestellt werden, wie das folgende Beispiel zeigt:

```
t=linspace(0,21,800);
x=cos(t); y=sin(t); z=t;
param3d(x,y,z);
```

Das Ergebnis ist in Abb. 2 gezeigt. Eine Geradengleichung $\vec{x}(t) = \vec{a} + t \cdot \vec{b}$ ist ein wichtiger Spezialfall. `param3d` verbindet die durch (x_k, y_k, z_k) gegebenen Punkte mit einer Geraden. Für die Darstellung einer Geraden für $t \in [t_a, t_b]$ genügt es also, die Koordinaten des Anfangspunkts $\vec{x}(t_a)$ und des Endpunkts $\vec{x}(t_b)$ an `param3d` zu übergeben, wie hier gezeigt ist:

```
a=[-2;1;5]; b=[2;-3;2]; // Vektoren aus Geradengleichung
t=[-2,2]; // Parameterintervall
xveca=a+t(1)*b; // Anfangspunkt
xvecb=a+t(2)*b; // Endpunkt
xmat=[xveca,xvecb]; // Matrix aus den 2 Punkten
param3d(xmat(1,:),xmat(2,:),xmat(2,:));
```

Dabei wurden der Ortsvektor des Anfangs- und Endpunktes zu einer Matrix zusammengesetzt, so dass in deren 1. Zeile die x-Koordinaten, in der 2. Zeile die y-Koordinaten und in der 3. Zeile die z-Koordinaten stehen. Diese Zeilen sind dann als Parameter an `param3d` zu übergeben.

Wenn die Anfangs- und Endpunkte eines Geradenstücks durch die Ortsvektoren $\vec{x}_a$ und $\vec{x}_e$ (bzw. in Scilab durch die Vektoren `xa` und `xe`) gegeben sind, dann erhält man das Geradenstück auch einfach durch `param3d([xa(1),xe(1)],[xa(2),xe(2)],[xa(3),xe(3)])`.

12 Zeichenketten

Zeichenketten können einer Variablen zugewiesen werden. Durch `prognam='sindemo.sce'` oder `prognam="sindemo.sce"` kann der Variablen `prognam` der Text des Dateinamens zugewiesen werden, das Programm kann dann statt durch `exec('sindemo.sce')` durch `exec(prognam)` aufgerufen werden. Durch „+“ können Zeichenketten zusammengesetzt werden. Durch `pfad='/Users/alexander/scilab-ma/'` und `vollnam=pfad+prognam` erhält die Variable `vollnam` den Pfad mit dem Dateinamen. Wie diese Beispiele zeigen, können Zeichenketten sowohl zwischen Hochkommata `'` als auch zwischen Anführungszeichen `"` eingeschlossen werden.

Mit der Funktion `string` können Zahlenvariable in Zeichenketten umgewandelt werden. So erhält man durch `gleichung='x='+string(%pi)` eine Zeichenkette mit dem Inhalt `x=3.1415927`. Zeichenketten können auch zu Matrizen zusammengesetzt werden. Durch die Eingabe `opzei=['+','-';'*','/']` bekommt man eine (2×2)-Matrix, als Echo am Bildschirm wird

```
 opzei  =
!+  -  !
!      !
!*  /  !
```

angezeigt. Man kann dann auf einzelne Matrixelemente gezielt zugreifen, also `opzei(2,1)` ausgeben oder verändern, wie das bei Zahlenmatrizen üblich ist. Eine Matrix von Zahlen kann durch die Funktion `string` in eine Matrix von Zeichenketten umgewandelt werden. Man kann dies zum Erstellen von Tabellen ausnutzen. So erhält man durch

```
x=-%pi:%pi/3:%pi; y=sinc(x);
Tabelle=string([x', y'])
```

eine (7×2)-Matrix von Zeichenketten erzeugen. Durch

```
Tabelle=['x','sinc(x)';Tabelle]
```

erhält man die Ausgabe

```
 Tabelle  =

!x           sinc(x)    !
!                       !
!-3.1415927  0          !
!                       !
!-2.0943951  0.4134967  !
!                       !
!-1.0471976  0.8269933  !
!                       !
!-2.220E-16  1          !
!                       !
!1.0471976   0.8269933  !
!                       !
!2.0943951   0.4134967  !
!                       !
!3.1415927   0          !
```

Mit `size(Tabelle)` kann man die Größe einer derartigen Matrix von Zeichenketten wie gewohnt abfragen und erhält hier 7 Zeilen und 2 Spalten. Im Gegensatz zu Matrizen von Zahlen erhält man für die ursprüngliche Tabelle (nur die Zahlen ohne die Überschrift) bei `l=length(Tabelle)` die Ausgabe

```
 l  =
    10.    1.
    10.    9.
    10.    9.
    1.     1.
    9.     9.
    9.     9.
    9.     1.
```

In dieser Matrix steht die jeweilige Anzahl der Zeichen in der Zeichenkette des betreffenden Matrixelements, beispielsweise enthält `Tabelle(3,2)` die Zeichenkette `'0.8269933'` mit 9 Zeichen. Man kann auch direkt mit `length('Zeichenkette')` die Zahl der Zeichen in einer Zeichenkette erfragen. Beachten Sie, dass dagegen `length(%pi)` als Ergebnis 1 liefert.

Besondere Vorsicht ist jedoch angebracht, wenn die Zeichenkette Sonderzeichen (nicht 7-Bit Ascii) enthält. Scilab arbeitet standardmäßig mit UTF-8, es versteh „zur Not" aber auch ISO-Latin 1. Ab Version 5.3.3 erhält man (könnte allerdings plattformabhängig sein) für `length('fr')` die Zahl 4 und nicht 3, denn das „ü" benötigt in URF-8 zwei Byte.

Auf einzelne Zeichen einer Zeichenkette kann man mit der Funktion `part` zugreifen: `zei='zeichenkette'; part(zei,4);` liefert das 4. Zeichen;

```
part(zei,(length(zei)-4):length(zei))
```

liefert die letzten 5 Zeichen, hier also das Ergebnis `kette`.

Auch hier ist große Vorsicht mit Sonderzeichen angebracht. Mit `part('für',3)` erhält man keinesfalls den Buchtaben „r“, sondern das Zeichen, das dem 2. Byte der UTF-8-Darstellung von „ü“, nämlich der dezimalen Zahl 188, entspricht, allerdings — da nur 1 Byte vorhanden ist — als ISO-Latin 1 interpretiert, und das ist das Zeichen ¼. Das „ü“ bekommt man mit `part('für',2:3)`.

Wenn ein Zeichen, beispielsweise ein Leerzeichen, zur Trennung einzelner Abschnitte einer Zeichenkette verwendet wurde, wie in

```
zei='Stoffel Alexander Professor';
```

dann kann man die einzelnen Teile, die durch das Trennzeichen `' '` getrennt werden, durch `posmat=tokenpos(zei,' ')` ermitteln und erhält die Matrix

```
posmat  =
  1.     7.
  9.     17.
  19.    27.
```

deren Zeilen den Indexbereich der durch das Leerzeichen getrennnten Bestandteile angeben. Diese erhält man beispielsweise durch

```
name=part(zei,posmat(1,1):posmat(1,2))
vorname=part(zei,posmat(2,1):posmat(2,2))
titel=part(zei,posmat(3,1):posmat(3,2))
```

Auch hier ist entsprechende Vorsicht angebracht, wenn der Text Sonderzeichen wie Umlaute enthält.

Von Excel kann man sich Tabellen als Textdateien abspeichern lassen, deren Spalten „Tab-getrennt“ sind. Dann kann man diese Datei zeilenweise einlesen und die einzelnen Zeilen mit `posmat=tokenpos(zeile,ascii(9));` und entsprechenden Anweisungen wie oben bearbeiten. Dabei wurde die Funktion `ascii(n)` benutzt, die das entsprechende ASCII-Zeichen zur durch `n` gegebenen Zahl erzeugt; `ascii(9)` liefert das Tabulatorzeichen.

Übergibt man umgekehrt ein einzelnes Zeichen oder eine Zeichenkette an die Funktion `ascii`, dann erhält man einen Zeilenvektor mit dem entsprechenden ASCII-Kode. So liefert `c=ascii('abc')` die Ausgabe 97. 98. 99.

An dieser Stelle ist eine Anmerkung zur Kodierung angebracht, die Einzelheiten können hier jedoch plattformabhängig sein! Standardmäßig benutzt Scilab UTF-8. So erhält man durch Eingabe in die Kommandozeile von `zei1='ü';ascii(zei1)` die Ausgabe 195. 188., anderseits liefert `zei2=ascii(252)` ebenfalls das Zeichen `ü`. „Zur Not“ versteht Scilab auch Iso Latin 1. Wenn man aber nun mit der Abfrage `if zei1==zei2 then`... arbeitet, dann wird man feststellen, dass der boolesche Ausdruck `zei1==zei2` den Wert „falsch“ hat, denn intern sind die beiden in der Ausgabe nicht zu unterscheidenden Zeichen sehr wohl verschieden. Man sollte also eine UTF-8-kodierte Datei nicht mit einem in Iso Latin 1 abgespeicherten Scilab-Programm bearbeiten — oder umgekehrt — sonst können unerwartete Effekte auftreten.

13 Lesen und Schreiben von Daten

Hier werden nur die einfachste Möglichkeit zum Lesen aus Dateien und Schreiben in Dateien besprochen. Die zu lesende Datei muss grundsätzlich nach der letzten Zeile ei-

ne Leerzeile enthalten, sonst wird die letzte Zeile von Scilab nicht gelesen. Die Datei `'Beispiel.dat'`, die folgende Daten (als ASCII-Zeichen, am Ende eine Leerzeile!) enthält

```
1.5 2
-17 5
6 -2
8 3.05
```

kann durch `A=read('Beispiel.dat',4,2);` ausgelesen werden. Hierzu muss sie im aktuellen Arbeitsverzeichnis stehen (sonst muss der Pfad explizit angegeben werden). Die Zahlen stehen anschließend in der Matrix `A` zur Verfügung. Das 2. Argument von `read` legt die Zeilenzahl, das dritte die Spaltenzahl fest. Wenn man die Zahl der Zeilen nicht kennt, so kann man einfach die ganze Datei auslesen, indem man das entsprechende Argument -1 setzt:

```
-->B=read('Beispiel.dat',-1,2);
-->size(B)
ans =
! 4. 2. !
```

Die Zahl der Spalten muss jedoch angegeben werden.

Zur Ausgabe kann man entsprechend das Kommando `write('Ausgabe.dat',A)` verwenden. Dabei ist zu beachten, dass lange Matrixzeilen umgebrochen werden und die Spalten der Matrix in einem „Flattersatz" ausgegeben werden. Ein ansprechenderes Erscheinungsbild erhält man durch `print('Ausgabe.dat',A)`. Dabei erfolgt die Ausgabe ähnlich der am Bildschirm, auch mit dem Zusatz zu Beginn „`A=`". Aber auch hier werden lange Zeilen umgebrochen. Für die Erstellung von Tabellen kann es daher sinnvoll sein, eher mit Spaltenvektoren zu arbeiten oder die Matrix gegebenenfalls zu transponieren. So kann man sich durch

```
x=-%pi:%pi/3:%pi; y=sinc(x);
Tabelle=[x' y']
print('Tabelle.txt',Tabelle)
```

eine Funktionstabelle der Funktion $f(x) = \mathrm{sinc}(x)$ erstellen, die man ausdrucken oder mit andereren Programmen weiterverarbeiten kann. Noch eleganter kann man die Matrix mit `string` in eine Matrix von Zeichenketten umwandeln (wie dies in Abschnitt 12 erklärt ist). Dann kann man nämlich noch eine Überschrift hinzufügen. Mit dem Beispiel aus Abschnitt 12 erhalten wir insgesamt für die Abspeicherung einer Tabelle von Funktionswerten:

```
x=-%pi:%pi/3:%pi; y=sinc(x);
Tabelle=string([x' y']);
Tabelle=['x','sinc(x)';Tabelle];
print('Tabelle.txt',Tabelle)
```

Die Zahl der ausgegebenen Nachkommastellen (am Bildschirm und mit `print`) kann mit dem Kommando `format(`n`)` verändert werden, wobei n eine positive ganze Zahl ist, die die Zahl der ausgegebenen Stellen festlegt. Die Standardeinstellung erreicht man mit $n = 10$, ein größeres n führt zu mehr, ein kleineres zu weniger ausgegebenen Stellen.

Ein nützlicher Hinweis für die Benutzer von LaTeX: Mit `Atex=prettyprint(A)` kann man die Matrix `A` in eine Zeichenkette `Atex` umwandeln, die die Anweisungen zur Darstellung der Matrix in LaTeX enthält. Durch `write('matrixA.tex',Atex)` kann man diese

Anweisungen dann in einer Datei abspeichern und weiterverarbeiten. Und wer lieber mit plain TeX arbeitet, erhält durch `Aplain=prettyprint(A,'tex')` die entsprechende Zeichenkette.

Leider kann man immer noch auf plattformbedingte Probleme stoßen: Ein Zeilenvorschub wurde dateiintern beim alten Mac (bis System 9) durch ein hexadezimales `0D` (carriage return), unter Unix durch `0A` (line feed) und unter Windows durch `0D0A` (carriage return und line feed) bewirkt. Wenn trotz korrekter Anweisung `B=read`... und im angegebenen Pfad vorhandener Datei mit korrekten Daten die Matrix `B` leer ist (Ausgabe von [ ]), dann kann dies daran liegen, dass die falschen Zeilenvorschübe in der Datei sind. Hier kann man Abhilfe schaffen, indem man die Datei in einen geeigneten Editor einliest und die alte Datei dann mit denselben Daten überschreibt. So liest Scilab 5.2.2 (immer noch) beim Mac keine Dateien mit (alten) Mac-Zeilenumbrüchen. Hier empfielt sich die Umwandlung in Unix-Zeilenumbrüche, beispielsweise durch Einlesen in den Editor BBedit und Rausschreiben mit der Option „Line Breaks: Unix“. Man kann die Umwandlung von (alten) Mac-Zeilenumbrüchen in Unix-Umbrüche auch mit dem Terminal-Befehl

`tr '\r' '\n' <` *Eingabedatei* `>` *Ausgabedatei*

durchführen. Auch ältere Excel-Versionen für Mac Os X schreiben beim Export in Texdateien noch diese uralten Zeilenvorschübe, so dass das Problem auch mit „neu“erzeugten Dateien auftreten kann.

Wenn eine entsprechende Datei schon existiert, so wird diese mit `write` oder `print` nicht überschrieben, der Befehl verursacht dann eine Fehlermeldung. Will man die alte Datei überschreiben, dann muss man sie mit dem Befehl `mdelete(`*Dateiname*`)` explizit löschen. Dieses Löschkommando verursacht keine Fehlermeldung, wenn die zu löschende Datei gar nicht existiert, man kann es also in Programmen vorsorglich vor einem Schreibkommando anbringen, wenn man alte Daten auf jeden Fall überschreiben will. Professioneller ist es natürlich, abzufragen, ob die Datei schon existiert. Mit

`info=fileinfo(`*Dateiname*`)`

erhält man verschiedene Informationen über die Datei in einem Vektor `info` mit 13 Komponenten, wenn die Datei schon existiert. Wenn sie nicht existiert, dann ist `info` eine leere Matrix, `length(info)` hat dann den Wert 0, andernfalls 13. Diesen Wert kann man abfragen und dann den Benutzer (beispielsweise über `x_dialog`) fragen, ob die Datei überschrieben werden soll.

Wenn die Datei sich nicht im aktuellen Arbeitsverzeichnis befindet, dann muss man an die Funktionen `read`, `write`, `mdelete`, `fileinfo` den Pfad übergeben. Dabei kann der Pfad relativ vom aktuellen Pfad angegeben werden, oder es kann der vollständige absolute Pfad angegeben werden.

14 FFT

Für die schnelle Fourier-Transformation gibt es die Routine
`fft(`*Vektor*`)`;
die inverse Transformation erhält man mit
`ifft(`*Vektor*`)`;
Bei ihr wird der Normierungsfaktor $\frac{1}{n}$ angebracht, wobei n die Gesamtzahl der Komponenten des Vektors ist. Beachten Sie, dass infolge von Rundungsfehlern die inverse

Transformation nicht exakt die ursprünglichen Signalwerte zurückliefert. Das folgende Programm zeigt die Verwendung von `fft`. Aufgrund des Abtasttheorems von Shannon ist nur die erste Hälfte der berechneten transformierten Wertte von praktischem Interesse; nur diese sind daher grafisch dargestellt.

```
x0=zeros(1,32); x1=ones(1,32);
x=[x0 x1 x0 x1 x0 x1 x0 x1]; // Rechtecksignal
X=fft(x);
X(5) // die Werte sind komplex!
xr=ifft(X); // inverse Transformation
fehler=norm(x-xr) // nicht Null, Rundungsfehler!
N=length(X)/2; // Shannon!
a=abs(X(1:N));
clf;
plot(0:(N-1),a,'x');
for k=1:N;
   plot([k-1,k-1],[0,a(k)]);
end;
centax; // mathlib hierfuer erforderlich
```

Die erzeugte grafische Darstellung ist in Abb. 3 gezeigt.

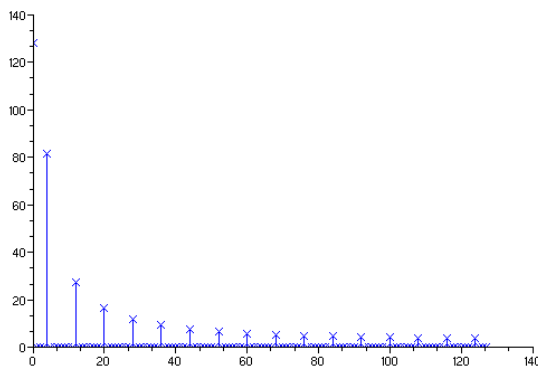


Abbildung 3: Absolutbeträge des Ergebnisses der FFT angewandt auf ein Rechtecksignal (nur die erste Hälfte der Werte ist gezeigt)

15 Programmierung

Man kann Scilab-Kommandos in einer Datei abspeichern und dann durch Eingabe von `exec('`*Dateiname*`')` ablaufen lassen. Wenn man als Beginn einer solchen Datei das Kommando `mode(7)` stehen hat, dann wird jeweils nur nach Drücken der `Return`-Taste die nächste Programmzeile abgearbeitet. Die erste Zeile darf jedoch nicht eine reine Kommentarzeile sein, sondern muss ein Scilab-Kommando enthalten! Für die Benennung von Programm-Dateien wird von den Scilab-Autoren die Endung `.sce` empfohlen. Als Beispiel für eine solche Programm-Datei können Sie sich den Inhalt von `elebei.sce` anschauen. Es ist zum Programmieren dringend zu empfehlen, mit einer solchen Datei zu arbeiten und die Kommandos nicht einzeln in der Kommandozeile einzugeben!

Schleifen können nach dem Schema

`for` *index=Zeilenvektor, Anweisung 1, Anweisung2,* ..., `end`

programmiert werden. Der Index durchläuft alle Komponenten des Zeilenvektors. Berechnung von 9! als Beispiel:

```
-->n=1;
-->for k=2:9,n=n*k,end
n =
2.
n =
6.
n =
24.
n =
120.
n =
720.
n =
5040.
n =
40320.
n =
362880.
```

Beachten Sie, dass statt des Zeilenvektors der Indexbereich auch in Form einer Matrix angegeben werden kann:

`for` *index=Matrix, Anweisung 1, Anweisung2,* ..., `end`

Dann durchläuft die durch *index* definierte Variable die **Spalten** der Matrix! Wenn versehentlich statt eines Zeilenvektors ein Spaltenvektor für den zu durchlaufenden Indexbereich angegeben wurde, dann hat dies sehr unerwünschte Folgen!

Eine andere Möglichkeit für Schleifen ist

`while` Bedingung, *Anweisung 1, Anweisung2,* ..., `end`

Beispiel:

```
-->n=1;k=1;
-->while k<6,k=k+1,n=n*k,end
```

Achtung: Schleifen sind zeitaufwendig. Man sollte sie vermeiden, indem man die komponentenweisen Operationen für Vektoren verwendet. Also

```
-->for k=1:4, x(k)=k^2,end
 x  =
    1.
 x  =
!   1. !
!   4. !
 x  =
!   1. !
!   4. !
!   9. !
 x  =
!   1.  !
```

```
!   4.  !
!   9.  !
!   16. !
```

ist zu vermeiden. Man sieht an diesem Beispiel, wie Scilab die Länge des Vektors bei jedem Schritt anpasst. Das schluckt Rechenzeit. Durch vorherige Reservierung von Speicherplatz (was von der Syntax her nicht notwendig ist), beispielsweise durch `x=zeros(4,1)`, spart man bereits ein wenig Zeit. Hier geht es noch erheblich schneller. Dieselben Komponenten erhält man durch

```
-->y=(1:4)^2
 y  =

!   1.    4.    9.    16. !
```

Wenn man einen Spaltenvektor möchte, so erreicht man dies durch Transponieren (mit dem Apostroph ').

Da die Bearbeitung von Schleifen unter Scilab erhebliche Rechenzeit beansprucht, lohnt es sich, etwas Mühe zur Programmierung aufzuwenden, um Schleifen zu vermeiden. Hierzu ist die Hilfsfunktion `sum(x)`, die die Summe der Komponenten des Vektors `x` berechnet, sehr nützlich. Das Produkt der Komponenten erhält man mit `prod(x)`. So berechnet man (für natürliche Zahlen n) die Fakultät $n!$ am einfachsten mit `prod(2:n)`.

Die Berechnung der Partialsumme für die harmonische Reihe

$$\sum_{k=1}^{100000} \frac{1}{k}$$

dauert bei einer Berechnung über eine Schleife mit dem Kommando (wobei vorher `s=0` und `n=100000` gesetzt wurde)

```
timer();for k=1:n, s=s+1/k;end;t=timer();s,t
```

1,26 Sekunden, was an der Variablen `t` abzulesen ist, denn `timer` liefert die Zeit seit dem letzten Aufruf. Ohne Schleife mit dem Aufruf

```
timer();x=1:n;s=sum(1.0./x);t=timer();s,t
```

dauert die Berechnung unter Verwendung der komponentenweise Division mit `./` nur 0,02 Sekunden! Für die Berechnung der alternierenden harmonischen Reihe kann man mit einem kleinen Trick ebenfalls eine Schleife vermeiden. Mit der naiv programmierten Schleife

```
timer();for k=1:n, s=s+(-1)^(k+1)*1/k;end;t=timer();s,t
```

braucht man 2,1 Sekunden, mit

```
timer();x=1.0./(1:n);x(2:2:n)=-x(2:2:n);s=sum(x);t=timer();s,t
```

dagegen nur 0,06 Sekunden! Dabei wurde ausgenutzt, dass man durch einen Vektor `2:2:n` alle geraden Indizes erhält, und der Vektor `x(2:2:n)` der Vektor ist, der aus den Komponenten von `x` mit geradem Index gebildet wird. Mit dem Kommando `x(2:2:n)=-x(2:2:n)` erhalten somit alle Komponenten mit geradem Index ein negatives Vorzeichen, und dies ohne jede Schleife und Verzweigung.

Abfragen mit Verzweigungen kann man mit dem Schlüsselwort `if` nach folgendem Muster programmieren:

`if` *Ausdruck* `then` *Kommandos*
`else` *Kommandos*

```
end
```
beispielsweise
```
if (x>0 & x<1) then y=x
else y=0
end
```
Für Verzweigungen mit `case` wird auf die übersichtliche Darstellung verwiesen, die man durch `help case` erhält.

Häufig sind Abfragen notwendig in großen Vektoren oder Matrizen nach Eigenschaften bestimmter Matrixelemente (z.B. ob ihr Wert größer als 255 ist). Hier wirken sich Schleifen über alle Indizes als sehr zeitaufwändig aus. Abhilfe kann hier das Makro `find` liefern, das es in solchen Fällen ermöglicht, auf Schleifen zu verzichten. Das Argument von `find` ist eine boolsche Matrix und der Rückgabewert ist ein Zeilenvektor von Indizes, an denen die Matrix den Wert wahr (`%T`) hat. Diese Indizes beziehen sich dabei auf eine spaltenweise Nummerierung der Matrix. Das Kommando

```
wo=find(A>255);
```

liefert also einen Zeilenvektor von Indizes, an denen das entsprechende Matrixelement größer als 255 ist. Mit `length(wo)` erhält man sofort die Zahl der Matrixelemente, die größer als 255 sind. Meist will man die entsprechenden Matrixelemente abändern, beispielsweise den Wert auf 255 setzen („clipping“). Dies kann durch

```
A(wo)=255;
```

geschehen. Zu beachten ist dabei, dass man zwischendurch die Größe von `A` nicht verändert hat und wirklich dieselbe Matrix benutzt (und nicht etwa zu einer Submatrix übergeht). Intern ist `A(wo)` ein Spaltenvektor; die Zuweisung eines konstanten Wertes an einen Spaltenvektor ist von der Syntax her korrekt. Damit hat man ohne jede Schleife alle Matrixelemente, die größer als 255 sind, auf 255 gesetzt. Ein weiteres Beispiel soll die Verwendung von `find` verdeutlichen. Mit

```
x=linspace(-1,2,800);
sx=size(x);
y=zeros(sx(1),sx(2));
wo=find( (0<=x)&(x<1) );
y(wo)=1;
plot(x,y);
```

wird eine Funktionstabelle der Skalierungsfunktion des Haar-Wavelets ($\varphi(x) = 1$ falls $x \in [0,1[$ und $\varphi(x) = 0$ falls $x \notin [0,1[$) erstellt und zu einer grafischen Darstellung benutzt. Diese Anweisungen funktionieren auch, wenn `x` ein Spaltenvektor ist (`linspace` liefert einen Zeilenvektor). Beachten Sie, dass durch `y=zeros(sx(1),sx(2));` sichergestellt ist, dass die Matrix `y` genau dieselbe Größe wie die Matrix `x` hat.

Wenn man im Zweifel ist, ob derartige Tricks wirklich das liefern, was man haben möchte, ist zu empfehlen, die Kommandos an einem kleinen Vektor oder einer kleinen Matrix zu testen und alle Zwischenergebnisse am Bildschirm auszugeben.

16 Funktionen

Funktionsmakros können gesondert in eigenen Dateien stehen, sie können aber auch in Programmdateien stehen oder im Kommandofenster eingegeben werden, wenn sie mit

dem Schlüsselwort „`endfunction`“ beendet werden. Es dürfen mehrere Funktionsmakros in einer Datei stehen. Außerdem wird dringend empfohlen, am Ende der Datei eine Kommentarzeile anzubringen, da Scilab (je nach Version) Dateien nur bis zum letzten Zeilenvorschub liest (und so die letzte Zeile eventuell nicht gelesen wird, was bei einem Kommentar nicht schlimm ist).
Jedes Funktionsmakro muss mit einer Zeile der Form

`function [`*Rückgabegrößen*`]=`*Funktionsname*`(`*Eingabegrößen*`)`

beginnen. Es dürfen mehrere Rückgabegrößen vorkommen, die dann mit Kommata getrennt werden. Ebenso dürfen mehrere Variable oder Eingabegrößen vorkommen, die ebenfalls durch Kommata zu trennen sind. In den folgenden Zeilen stehen dann die Kommandos, mit denen die Rückgabegrößen aus den Eingabegrößen berechnet werden. Das Funktionsmakro soll mit dem Schlüsselwort `endfunction` beendet werden. Dann kann es auch durch das Kommando `exec` geladen werden und es kann mitten in einem ausführbaren Programm (vor dem ersten Aufruf) stehen. Ein Beispiel soll dies erläutern.
In der Datei `Beispiel.sci` steht unter anderem folgende Funktion (die Abfragen, die überprüfen, ob die Eingabegrößen sinnvoll sind, wurden zur besseren Übersichtlichkeit entfernt) :

```
function [c]=vecprod(a,b)
c=zeros(3,1);
c(1)=a(2)*b(3)-a(3)*b(2);
c(2)=a(3)*b(1)-a(1)*b(3);
c(3)=a(1)*b(2)-a(2)*b(1);
endfunction
// Vektorprodukt
```

In eckigen Klammern steht die Rückgabegröße, hier das Ergebnis des Vektorprodukts, der Vektor `c`, die Eingabegrößen stehen in runden Klammern hinter dem Funktionsnamen. Der Aufruf kann beispielsweise in der Form `u=[1;-2;5];v=[-2;4;1];x=vecprod(u,v)` erfolgen. Sie können sich sehr viele weitere Beispiele für Funktionsunterprogramme in der Datei `Beispiel.sci` anschauen.

Bevor man die Funktionen aufruft, müssen diese Makros jedoch eingebunden werden. Dies geschieht mit dem Kommando `exec('`*Dateiname*`')` (die Makros sind mit `endfunction` zu beenden) im Scilab-Kommando-Fenster oder über das entsprechende Menu. Dabei muss die Datei im im aktuellen Arbeitsunterverzeichnis stehen (oder der Pfad muss mit angegeben werden). Anschließend können die in der Datei stehenden Funktionsunterprogramme benutzt werden. Man kann sie aber auch einfach in ein ausführbares Programms stellen (vor dem ersten Aufruf), das man dann durch `exec` ausführt.

17 Listen

Zahlen und Zeichenketten, können zu Vektoren und Matrizen zusammengefaßt werden. Will man jedoch verschiedene Objekte, z.B. eine Zeichenkette, einen Vektor und eine Matrix zusammenfassen, so kann man dies in der Form einer Liste tun. Dies kann beispielsweise so erfolgen:

```
Text='Aufgabe 35)a)';A=[-3,15,-9;2,-12,5;-1,7,-5];b=[9;2;4];
LGS=list(Text,A,b)
```

Dann kann man durch `LGS(2)` beispielsweise auf die ganze Matrix oder durch `LGS(2)(1,1)` auf ein einzelnes Matrixelement zugreifen oder als viertes Listenelement die Lösung des

Gleichungssystems hinzufügen: `LGS(4)=linsolve(LGS(2),-LGS(3))`.

Listen werden intern bei der Behandlung von Polynomen und rationalen Funktionen benutzt. Die kleine Funktionsbibliothek `Beispiel.sci` enthält Funktionen, die das Ergebnis einer Partialbruchzerlegung in Form einer Liste zurückgeben.

18 Polynome und rationale Funktionen

Scilab bietet die Möglichkeit, für Polynome und rationale Funktionen nicht-numerische, d.h. rein formelmäßige Rechnungen durchzuführen. Ein einfacher Ausgangspunkt ist das Kommando

```
x=poly(0,'x')
```

das den elementaren Polynom-Baustein $f(x) = x$ definiert. Damit wird x als Polynomvariable festgelegt. Dann können durch Kommandos der Art

```
p=x^4-13*x^2+36
q=x^4+2*x^3-2*x^2-6*x+5
```

kompliziertere Polynome einfach definiert werden. Dann können durch arithmetische Ausdrücke der Form `f=p*q` und g=5*p+3*q neue Polynome definiert werden. Dabei hat man auf eine Besonderheit zu achten: Der Exponent der Polynomvariablen `x` darf nicht komplex sein, auch nicht mit verschwindendem Imaginärteil. So führt die Eingabe `a=2+0*% i; f=x^a` zur Fehlermeldung „`invalid exponent`“. Nun wird man kaum eine solche wenig sinnvolle Eingabe vornehmen. Aber man sollte an diese Besonderheit denken, wenn man darüber rätselt, warum die Eingabe `b=[%i,2]; f=x^b(2)` zu derselben Fehlermeldung führt, wo doch `b(2)` reell zu sein scheint. Es ist aber in der Tat intern komplex, da die zweite Komponente eines Vektors intern von demselben Typ ist wie die erste, und die ist sichtbar komplex. Abhilfe schafft in diesem Fall die Anweisung `f=x^real(b(2))`. Zum Glück besteht diese Empfindlichkeit nicht bei rellen Zahlen als Basis, da man hierfür auch die Potenz für komplexe Exponenten definieren kann: $3^{2+j} = e^{(2+j)\ln 3}$.
Die Ableitung des Polynoms `p` erhält man durch `h=derivat(p)`. Das Kommando

```
u=roots(q)
```

liefert einen Spaltenvektor `u`, dessen Komponenten alle Nullstellen des Polynoms `q` (auch die komplexen) enthalten, und zwar ihrer Vielfachheit nach.

Polynome können auch durch Vorgabe der Nullstellen definiert werden. Hierzu definiert man zunächst einen Vektor, dessen Komponenten die Nullstellen sind (mit der entsprechenden Vielfachheit). Das Polynom $(x-1)^4$ wird durch

```
r2=ones(1,4)
q2=poly(r2,'s') // Polynom mit vorgegebenen Nullstellen
}\noindent
oder
{\obeylines\tt
q2=poly(r2,'s','roots') // ohne 3. Argument Option "roots"
```

definiert. Das einfache Kommando `x=poly(0,'x')`, das uns `x` als Polynomvariable liefert, ist so erklärbar: es liefert das Polynom, das 0 als einzige Nullstelle hat, und das ist das Polynom $f(x) = x$. Ohne Angabe des 3. Arguments ist also bei tt poly die Option `'roots'` gewählt.

Man kann ein Polynom auch dadurch definieren, dass man die Koeffizienten durch einen Vektor mit der Option `'coeff'` vorgibt:

```
c=[5 -6 -2 2 1]
q3=poly(c,'x','coeff') // Vektor c mit Koeffizienten
```

erzeugt das Polynom $5-6x-2x^2+x^4$. Die Komponente `c(k)` liefert also den Koeffizienten zu x^{k-1}. Ein Koeffizientenvektor mit n Koeffizienten definiert also ein Polynom $(n-1)$. Grades, wenn $c_n \neq 0$. Man kann den entsprechenden Vektor der Koeffizienten durch das Kommando

```
coeff(q3) // liefert Koeffizienten zurueck
```

erhalten.

Die Polynomdivision, die bei gegbenem Zählerpolynom `pz` und Nennerpolynom `pn` einen Quotienten `q` und einen Rest `r` liefert, wird mit dem Aufruf

```
[r,q]=pdiv(pz,pn) // Polynomdivision
q*pn+r-pz// Probe
```

durchgeführt. Bei der Probe müßte das Nullpolynom das Ergebnis sein (bis auf Ausdrücke, die durch Rundungsfehler zustandekommen, also sehr betragskleine Koeffizienten haben). Der Aufruf von `pdiv(pz,pn)` allein liefert nur das Quotientenpolynom `q` ohne das Restpolynom.

Das Ausrechnen des Werts eines Polynoms `p` mit dem Hornerschema erfolgt durch `horner(p,a)`, wobei das Argument `a` eine skalare Konstante oder ein anderes Polynom sein darf. `horner(p,3)` liefert den zahlenmäßigen Funktionswert $p(x)$ für $x = 3$. Aus dem Polynom x^4 erhält man durch das Kommando `horner(x^4,1+x)` das Polynom $(1+x)^4$, das dann in der Form $1+4x+6x^2+4x^3+x^4$ angezeigt wird. Dieses Kommando erlaubt auch einen Variablenwechsel:

```
Polynoms s=poly(0,'s') // anderer Polynombaustein
ps=horner(p,s) // erlaubt Variablenwechsel
```

Der Variablenwechsel kann auch durch das Kommando `pt=varn(p,'t')` durchgeführt werden, `pt` ist dann dasselbe Polynom wie `p`, lediglich die Variable `x` ist durch `t` ersetzt. Mit nur einem Argument liefert diese Funktion den Variablennamen, `varn(pt)` liefert also den Variablennamen `t`. Dies kann für Funktionsunterprogramme nützlich sein, wenn man nur das Polynom `p` als Argument weitergeben möchte, nicht jedoch den zugehörigen Variablennamen. Die Funktion `horner` läßt komplexe Argumente zu, also `horner(p,1+%i)` liefert den Funktionswert $p(x)$ für $x = 1 + j$. Nicht zugelassen sind jedoch Vektoren. Zur grafischen Darstellung eines Polynoms möchte man jedoch den Funktionswert für einen Vektor als Argumente komponentenweise berechnen. Dies ist mit der Funktion `freq` möglich. Diese ist eigentlich für rationale Funktionen vorgesehen. Für Polynome setzt man einfach das zweite Argument, das das Nennerpolynom enthalten muss, auf den Wert 1. Das erste Argument ist das Polynom, dessen Wert man berechnen will, das 3. Argument das Funktionsargument, und dieses darf auch ein Vektor sein (Zeilen- oder Spaltenvektor). Als Ergebnis erhält man stets einen Zeilenvektor mit den Funktionswerten des Polynoms. `freq(p,1,3)` liefert also den Funktionswert für $x = 3$, `freq(p,1,1+%i)` liefert den Funktionswert für $x = 1 + j$. Eine grafische Darstellung des Polynoms für $-5 \leq x \leq 5$ erhält man durch

```
t=-5:0.05:5;
y=freq(p,1,t); // freq erlaubt Vektoren (nicht horner!)
plot(t,y) // so graph. Darstellung von Polynomen
```

Das Kommando `f=polfact(p)` liefert eine Produktzerlegung des Polynoms `p` in reelle Polynome ersten und zweiten Grades, die Faktoren werden als Zeilenvektor `f` übergeben. Die erste Komponente `f(1)` enthält eine Konstante, und zwar den Koeffizienten zum Term mit der höchsten auftretenden Potenz (also den Koeffizienten von x^n, wenn `p` ein Poly-

nom n. Grades ist). Die weiteren Komponenten enthalten Polynome ersten oder zweiten Grades, so dass das Polynom `p` das Produkt aller Komponenten von `f` ist. Man kann die Probe machen, indem man `prod(f)-p` berechnen läßt. Es muss bis auf Terme, die von Rundungsfehlern herrühren, das Nullpolynom sein.

Durch das Kommando `[f,a]=factors(p)` erhält man ebenfalls eine Produktzerlegung des Polynoms `p` in reelle Polynome ersten und zweiten Grades. Hier wird der Faktor der höchsten Potenz in der Variablen `a` übergeben, und die Polynomfaktoren sind Elemente der Liste `f`, sind also durch `f(1), f(2), f(3)`,... erhältlich.

Rationale Funktionen können einfach durch ein Kommando der Art `f=p/q` definiert werden. (Man sollte möglichst nur Polynome mit demselben Variablennamen verwenden, sonst kann es später zu Fehlern führen!). Dabei wird automatisch gekürzt:

```
p=x^4-13*x^2+36
q=x^2+3*x-10
f=p/q // automatisch gekuerzt
```

liefert automatisch die gekürzte gebrochen rationale Funktion

$$f(x) = \frac{-18 - 9x + 2x^2 + x^3}{5 + x}$$

Die Ableitung einer rationalen Funktion erhält man durch `derivat(f)`. Das Zählerpolynom einer gebrochen rationalen Funktion erhält man durch `numer(f)`, das Nennerpolynom durch `denom(f)`.

Funktionswerte erhält man durch `freq(p,q,a)`, wobei das dritte Argument `a` eine reelle oder komplexe Konstante oder ein Vektor ist. Obwohl bei der Definition der Faktor aus der gemeinsamen Nullstelle weggekürzt wird, führt der Aufruf der gemeinsamen Nullstelle mit `freq` (im obigenBeispiel `freq(p,q,2)`) zu einer Fehlermeldung (Division durch Null). Abhilfe schafft die ausschließliche Benutzung der gekürzten rationalen Funktion: `freq(numer(f),denom(f),2)` liefert den korrekten Funktionswert an der Definitionslücke. Das Kommando `horner(f,2)` liefert ebenso den korrekten Funktionswert für die Definitionslücke, diese Funktion kann also auch zur Berechnung der Funktionswerte von rationalen Funktionen eingesetzt werden, sie läßt aber im Gegensatz zu `freq` keinen Vektor als Argument zu. Beachten Sie, dass bei `freq` Zähler und Nenner getrennt als Argumente anzugeben sind, bei `horner` wird die rationale Funktion `f` als Argument angegeben. Für grafische Darstellungen ist `freq` praktischer, da es Vektoren als Argumente erlaubt.

Mit `f0=clean(f)` werden in einer rationalen Funktion oder einem Polynom `f` alle Terme beseitigt, deren Koeffizient sehr klein ist und der durch Rundungsfehler entstanden ist (standardmäßig kleiner als 10^{-10}). Dies kann beim Durchführen von Proben (z.B. nach Produktzerlegungen oder Polynomdivisionen) nützlich sein.

Wenn das Argument eine quadratische Matrix **A** ist, dann liefert der Aufruf von `poly` das charakteristische Polynom $\det(x \cdot \mathbf{E} - \mathbf{A})$:

```
A=[3 2 2 2;-10 -6 3 -11;0 0 1 -12;0 0 1 8]
c=poly(A,'x') // liefert charakteristisches Polynom
det(x*eye(4,4)-A) // charakteristisches Polynom
```

Für die Partialbruchzerlegung stehen in der Funktionssammlung in `Beispiel.sci` zwei Funktionsunterprogramme zur Verfügung, `pbzr` verwendet ausschließlich relle Polynome, bei den einzelnen Termen der Zerlegung können also im Nenner quadratische Polynome ohne reelle Nullstellen vorkommen (nützlich für die Integration), `pbzc` liefert eine Zer-

legung, bei der im Nenner nur Polynome ersten Grades vorkommen mit gegebenenfalls komplexen Nullstellen. Der Aufruf erfolgt durch

```
pbr=pbzr(f,eps)
```

oder für die komplexe Variante

```
pbc=pbzc(f,eps)
```

Dabei ist `f` eine echt gebrochen rationale Funktion und `eps` eine Fehlerschranke. Alle Größen, deren Betrag kleiner als `eps` ist, werden als 0 behandelt, insbesondere werden Pole, die sich betragsmäßig um weniger als `eps` unterscheiden, als mehrfach angesehen. Da Scilab numerisch rechnet, sollte aufgrund des unvermeidlichen Rundungsfehlers `eps` nicht zu klein gewählt werden. Die Rückgabe erfolgt in Form einer Liste. Eine Liste ist eine Scilab-Variable, die als Listenelemente mehrere Größen unterschiedlichen Typs enthalten kann, deren gesamte Anzahl durch `size(pbr)` oder `length(pbr)` erfragt werden kann. Die einzelnen Elemente der Liste erhält man durch `pbr(k)`, wobei die Variable oder Konstante `k` Werte zwischen 1 und `length(pbr)` annehmen kann. Beispielsweise enthält nach dem obigen Aufruf der Funktion `pbzr` und nach Eingabe von

```
b=pbr(k)
```

die Variable `b` die Daten für den `k`. Term der Partialbruchzerlegung, also den `k`. Partialbruch. Die Variable `b` ist dann selbst ein Zeilenvektor, der für die reelle Variante folgende Informationen erhält:

`b(1)` enthält den Zähler des Partialbruchs in der Form eines Polynoms
`b(2)` enthält den Nenner des Partialbruchs ohne Exponenten
`b(3)` enthält den Exponenten im Nenner des Partialbruchs.

Den `k`. Partialbruch erhält man durch

```
b=pbr(k); b(1)/b(2)^coeff(b(3))
```

Da der Vektor `b` von Scilab als Vektor von Polynomen abgespeichert ist, wird die 3. Komponente `b(3)` automatisch als Polynom (0. Grades) aufgefaßt. Der direkte Aufruf von `b(2)^b(3)` würde daher zu einer Fehlermeldung führen, durch `coeff(b(3))` erhält man den Zahlenwert des Exponenten. Gibt man beispielsweise

```
x=poly(0,'x');
eps=1.0E-7;
f=(x^3-10*x^2+7*x-3)/((x-1)^2*(x^2+4*x+5));
pbr=pbzr(f,eps)
```

ein, so erhält man die Ausgabe

```
 pbr  =
       pbr(1)
                                          2
  - 3.9999999 + 1.7x     5 + 4x + x      1
       pbr(2)
  - 0.5000000   - 1 + x     2
       pbr(3)
  - 0.7000000   - 1 + x     1
```

Die Partialbruchzerlegung hat also 3 Terme, und man kann aus der Ausgabe ablesen:

$$\frac{x^3-10x^2+7x-3}{(x-1)^2(x^2+4x+5)} = \frac{1,7x-4}{x^2+4x+5} - \frac{0,5}{(x-1)^2} - \frac{0,7}{x-1}$$

Zur Probe kann man die ursprüngliche Funktion aus den Partialbrüchen durch

```
fr=0;
for k=1:length(pbr);
    b=pbr(k);
    fr=fr+b(1)/b(2)^coeff(b(3));
end; // for k
```

zurückberechnen und dann mit der noch vorhandenen ursprünglichen Funktion `f` vergleichen, beispielsweise, indem man `abwei=f-fr` eingibt. Diese Differenz ist infolge der Rundungsfehler eine gebrochen rationale Funktion, deren Zählerpolynom nur betragsmäßig sehr kleine Koeffizienten enthalten sollte.

Man kann diese Probe auch einfacher durch Aufruf von `fehler=pbzr_probe(f,pbr)` durchführen. Man erhält dann in der Ausgabe wie oben `f-fr` und als Rückgabewert den Betrag (die Länge) des Koeffizientenvektors des Zählers von `f-fr` als Maß dafür, ob das Ergebnis akzeptabel ist — oder etwas schief gelaufen ist. Typischer Wert für dieses Beispiel ist 3.179D-15.

Man hat zu beachten, dass sich bei der Berechnung von Nullstellen von Polynomen Rundungsfehler so unglücklich addieren können, dass man mit der Partialbruchzerlegung Probleme bekommt, denn hierfür ist eine Produktzerlegung des Nenners und damit eine Nullstellenbestimmung notwendig. Dies kann bei „harmlos“ aussehenden Beispielen wie

$$f(x) = \frac{x^2 - 5x + 8}{(x-1)^3(x+3)}$$

vorkommen. Mit der vernünftigen Fehlerschranke `eps=1.0E-8` kann man hierfür in einer Warnung die Ausgabe „Faktorzerlegung Nenner“

$$1 \qquad 3 + x \qquad 1.0000136 - 2.0000136x + x^2 \qquad - 0.9999864 + x$$

erhalten. Hier ist etwas schief gelaufen, denn das quadratische Polynom ist bis auf Rundungsfehler $(1-x)^2$, und dies führt im weiteren Verfahren zu einem Widerspruch und zu einem Abbruch. Die Partialbruchzerlegung wird dann mit dem zehnfachen Wert für die Fehlerschranke `eps` so lange neu begonnen, bis dieser Fehler beseitigt werden kann, vorausgesetzt, dass `eps`<0.01 ist. Bei dem hier angegebenen Beispiel (beachten Sie, dass Rundungsfehler rechnerabhängig sein können) wurde die Partialbruchzerlegung erst bei einer Fehlerschranke `eps=0.0001` erfolgreich durchgeführt. Die Probe mit `pbzr_probe` führte dann zu einem Rückgabewert von 0.0001934 als Maß für den numerischen Fehler.

Mit der komplexen Variante erhält man

```
-->pbc=pbzc(f,eps)
pbc =
pbc(1)
! - 0.7    1.     1. !
pbc(2)
! - 0.5     1.    2. !
pbc(3)
! 0.85 + 3.7i    - 2. + i     1. !
pbc(4)
! 0.85 - 3.7i    - 2. - i     1. !
```

Auch bei der komplexen Variante erhält man das Ergebnis in Form einer Liste. Die einzelnen Elemente der Liste erhält man durch `pbc(k)`, wobei die Variable oder Konstante `k`

Werte zwischen 1 und `length(pbc)` annehmen kann. So enthält nach dem obigen Aufruf der Funktion `pbc` und nach Eingabe von

```
b=pbc(k)
```

die Variable `b` die Daten für den `k`. Term der Partialbruchzerlegung, also den `k`. Partialbruch. Die Variable `b` ist dann selbst ein Zeilenvektor, der für die komplexe Variante folgende Informationen erhält (etwas unterschiedlich gegenüber der reellen Variante):

`b(1)` enthält den Koeffizienten des Partialbruchs als komplexe Zahl
`b(2)` enthält die Nullstelle des Nenners des Partialbruchs
`b(3)` enthält den Exponenten im Nenner des Partialbruchs.

Den `k`. Partialbruch erhält man durch

```
b=pbc(k); b(1)/(x-b(2))^real(b(3))
```

Dabei ist `x` die Polynomvariable. Die Angabe `real(b(3))` ist notwendig, weil der Exponent `b(3)` intern als komplexe Zahl abgespeichert ist (mit Imaginärteil 0), was im Exponenten zu einer Fehlermeldung führt. Somit erhält man aus der obigen Scilab-Ausgabe die Partialbruchzerlegung

$$\frac{x^3-10x^2+7x-3}{(x-1)^2(x^2+4x+5)} = -\frac{0,7}{x-1} - \frac{0,5}{(x-1)^2} + \frac{0,85+3,7j}{x+2-j} + \frac{0,85-3,7j}{x+2+j}$$

Auch hier kann man zur Probe die ursprüngliche Funktion zurückberechnen durch

```
fr=0;
d=size(pbc);
for k=1:d(1);
    b=pbc(k);
    fr=fr+b(1)/(x-b(2))^real(b(3));
end; // for k
```

und durch Ausgabe der Differenzfunktion `abwei=f-fr` sich einen Eindruck vom Rundungsfehler verschaffen.

Diese Probe kann hier mit `numerr=pbzc_probe(f,pbc)` automatisch durchgeführt werden. Man erhält in der Ausgabe die Differenzfunktion `f-fr` und als Rückgabewert die Länge des Koeffizientenvektors des Zählers als Maß für den numerischen Fehler. Typischer Wert für das angegebene Beispiel ist 0.0000324.

19 Numerische Integration

Hierfür stehen zwei Kommandos zur Verfügung. „`integrate`“ verlangt als Argument einen arithmetischen Ausdruck, die Integrationsvariable und die Grenzen:

```
integrate('%e^(-x^2)','x',-10,10)
```

berechnet $\int_{-10}^{+10} e^{-x^2} dx$. Das Kommando „`intg`“ ist dagegen nur möglich für Funktionen, die intern vom Typ „function“ sind. Der Funktionsname (ohne Funktionsvariable) und die Grenzen sind als Argument anzugeben.

```
intg(1,2,ln)
```

berechnet $\int_{1}^{2} \ln x dx$ (wenn vorher `exec('Beispiel.sci')` eingegeben wurde und damit die Funktion `ln` definiert ist). Dieses sehr bequeme Integrationsprogramm `intg` (man braucht nur die Grenzen und den Funktionsnamen anzugeben) funktioniert allerdings nur mit Funktionen, die vom Benutzer geschrieben wurden oder von den Autoren von

Scilab in Scilab geschrieben und als „external“ eingebunden wurden. Einige häufig benutzte Funktionen wie sin sind jedoch kein „external“, sie sind in FORTRAN oder C geschrieben. Diese können also nicht als Argument in `intg` auftauchen. Dann muss man sich mit `integrate` helfen oder dieselbe Funktion mit einem Funktionsmakro (mit den Schlüsselwörtern `function` und `endfunction`) mit einem neuen Namen umprogrammieren. Funktionen, die als „external“ benutzt werden können, erkennt man durch eine Abfrage mit der Funktion `typeof(`*Funktionsname*`)`: Wenn man die Antwort „`function`“ erhält, kann der Funktionsname als `external` verwandt werden, nicht jedoch, wenn man die Antwort „`fptr`“ erhält (damit ist vermutlich function pointer gemeint).

20 Lösung nichtlinearer Gleichungen

Will man beispielsweise die Extremwerte der Funktion $\mathrm{sinc}(x) = \frac{\sin x}{x}$ falls $x \neq 0$ und $\mathrm{sinc}(0) = 1$ berechnen, so kommt man auf die Gleichung $\tan x = x$, für die man außer der Lösung $x = 0$ durch explizite Rechnung keine weiteren Lösungen mehr bekommt. Der grafischen Darstellung von $\mathrm{sinc}(x)$ oder auch von $\tan(x)$ sieht man jedoch an, dass es unendlich viele Lösungen geben muss. Wenn man derartige Gleichungen mit Scilab lösen möchte, dann hat man sie zunächst in die Form $f(x) = 0$ zu bringen. Weiterhin muss man eine Näherungslösung `xstart` bestimmen (z.B. aus der grafischen Darstellung), in deren Nähe die Lösung gesucht wird. Die Funktion `f` muss man als Scilab-Funktion programmieren (in einer eigenen Datei, die dann mit `exec` einzubinden ist, oder mit einem Funktionsmakro mit den Schlüsslewörtern `function` und `endfunction`. Dann erhält man durch `x=fsolve(xstart,f)` eine numerische Näherung für eine Lösung der Gleichung $f(x) = 0$ nahe dem vorgegebenen Startwert.

In unserem Beispiel kann man aus der grafischen Darstellung von $\mathrm{sinc}(x)$ einen Startwert `xstart=4` ablesen. Die Kommandos

```
function [y]=fun(x); y=tan(x)-x; endfunction; x=fsolve(xstart,fun)
```

liefern dann eine weitere Extremstelle $x \approx 4,4934095$ (in der Tat ein lokales Minimum) und die Probe liefert beispielsweise 8.882E-16 als Funktionswert von $\tan(x)-x$. Der Startwert `xstart=8` liefert eine weitere Lösung $x \approx 10.904122$, diesmal ein lokales Maximum. Dieses Beispiel zeigt sehr gut, dass die Lösung im allgemeinen vom Startwert abhängt.

Ein von den Anwendungen her interessanteres Beispiel ist die Funktion

$$g(x) = \frac{1}{\pi}\left(\pi - x + \frac{1}{2}\sin(2x)\right)$$

Sie gibt die relative Leistung in Abhängigkeit des Zündwinkels x beim Phasenanschnitt an (nur interessant für $0 \leq x < \pi$. In der Praxis ist man jedoch an der Umkehrfunktion interessiert, d.h. man möchte statt $r = g(x)$ den Zündwinkel x bei vorgegebener relativer Leistung r bestimmen. Wenn man die Gleichung explizit auflösen könnte, dann hätte man die Umkehrfunktion $x = g^{-1}(r)$. Dies ist rechnerisch jedoch nicht (mit Hilfe von elementaren Funktionen) möglich. Um die Funktion `fsolve` zu benutzen, muss man alles auf eine Seite bringen und Lösungen x der folgenden Gleichung suchen:

$$f(x,r) = \frac{1}{\pi}\left(\pi - x + \frac{1}{2}\sin(2x)\right) - r = 0$$

Man kann nun für vorgegebenes r, z.B. $r = 0,2$ wie beim vorherigen Beispiel vorgehen und einfach als Startwert $x_{\mathrm{Start}} = \frac{\pi}{2}$ wählen:

```
function [y]=fun(x); y=(1/%pi)*(%pi-x+0.5*sin(2*x))-0.2; endfunction;
xstart=%pi/2; x=fsolve(xstart,fun)
```
liefert die Näherung $x \approx 2.0850232$. Für die Praxis wäre es sehr hinderlich, für jeden Wert von r die Definition der Funktion zu ändern. Man möchte doch lieber die Funktion von zwei Variablen abhängen lassen:
```
function [y]=fun(x,r); y=(1/%pi)*(%pi-x+0.5*sin(2*x))-r; endfunction
```
Dies ist in der Tat möglich. Dann hat man jedoch den Wert der zweiten Variablen r an `fsolve` weiterzugeben. Dies geschieht in Form einer Liste. Die Kommandos
```
r=0.2; x=fsolve(xstart,list(fun,r))
```
liefern dieselbe Lösung wie vorhin. Nun kann man sich aber die Lösungen in Abhängigkeit von `r` auch zeichnen lassen. Für die grafische Darstellung mit `eplot` muss man die Variable r umbenennen und mit `x` bezeichnen (weil die Funktion `eplot` dies so will, aber für die Berechnung in Funktionsmakros kommt es beim Aufruf nur auf die Reihenfolge der Variablen, nicht auf die Benennung der Variablen im rufenden Programm an). In der Tat liefert das Kommando (nach Einbinden von `Beispiel.sci`)
```
eplot('fsolve(xstart,list(fun,x))',[0,1])
```
eine grafische Darstellung der Umkehrfunktion $g^{-1}(r)$ für $r \in [0,1]$. Allerdings hat man da etwas Glück gehabt, dass die Funktion `fsolve` für jeden Wert von r die Lösung bei dem vorgegegebenen Startwert $x_{\text{Start}} = \frac{\pi}{2}$ findet. Dies ist bei diesem Anwendungsbeispiel unproblematisch. Die ursprüngliche Funktion $g(x)$ kann man übrigens durch `eplot('fun(x,0)',[0,%pi])` grafisch darstellen.

21 Gewöhnliche Differentialgleichungen

Anfangswertprobleme gewöhnlicher Differentialgleichungen können von Scilab näherungsweise numerisch gelöst werden. Bei Differentialgleichungen erster Ordnung der Form

$$y' = f(t, y)$$

ist dies besonders einfach. Die rechte Seite muss in Form eines Funktionsunterprogramms zur Verfügung stehen (vorher zu laden oder im selben Programm enthalten). Ein Beispiel soll dies erläutern. Für die Differentialgleichung

$$y' = 1 + y^2$$

steht am Anfang des Beispielprogramms `numdgl.sce`, das auf
```
http://alexanderstoffel.selfip.org/scimat/numdgl.sce
```
erhältlich ist, die folgende Funktion
```
function f=epyq(t,y)
f=1+y^2
endfunction
```
Die Variable `t` muss aufgeführt werden, auch wenn die Differentialgleichung nicht explizit zeitabhängig ist. Die diskreten Zeitpunkte, für die die Lösung $y(t)$ zu berechnen ist, sind in Form eines Zeilenvektors zur Verfügung zu stellen, beispielsweise durch
```
t=linspace(0,1.5,50);
```
Die eigentliche Lösung erfolgt schließlich durch
```
y=ode(0,0,t,epyq);
```
Dabei ist das erste Argument von `ode` der Anfangswert von $y(t_0)$, das zweite der Anfangszeitpunkt t_0, der sinnvollerweise mit der ersten Komponente `t(1)` des Vektors `t`

übereinstimmen sollte. Das vierte Argument ist der Name einer Scilab-Funktion. Nach dem Aufruf steht die Näherungslösung im Zeilenvektor `y` zur Verfügung, der (im Erfolgsfall) dieselbe Länge wie der Vektor `t` hat. Bei numerischen Problemen kann auch der Abbruch mit einer Fehlermeldung erfolgen. Dann kann durch `size(y)` oder `length(y)` die tatsächliche Länge bis zum Fehlerabbruch abgefragt werden. Bei Erfolg kann man sich dann durch `plot(t,y)` die Lösung anschauen.

Scilab arbeitet mit verschiedenen numerischen Verfahren, die automatisch ausgewählt werden oder vom Benutzer durch einen ausführlicheren Aufruf der Routine `ode` festgelegt werden können. Hierfür wird auf die Handbücher verwiesen.

Differentialgleichungen höherer Ordnung müssen in ein System von Differentialgleichungen erster Ordnung umgewandelt werden. Die unbekannten Funktionen und die rechten Seiten dieses Systems können dann als Vektoren aufgefaßt werden. Die zu lösende Differentialgleichung *n.Ordnung* muss also in die Form gebracht werden

$$\frac{d}{dt}\vec{y}(t) = \vec{f}(t, \vec{y}(t))$$

mit

$$\vec{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \dots \\ y_n(t) \end{pmatrix}$$

Analog sind dabei die n Funktionen der rechten Seite zu einem Vektor zusammengefaßt:

$$\vec{f}(t, \vec{y}) = \begin{pmatrix} f_1(t, \vec{y}) \\ f_2(t, \vec{y}) \\ \dots \\ f_n(t, \vec{y}) \end{pmatrix}$$

Dies wird von Scilab unterstützt: der Rückgabewert einer Funktion kann ein Vektor sein. Das Vorgehen soll am Beispiel der Differentialgleichung des mathematischen Pendels für große Auslenkungen

$$y'' + \omega_0^2 \sin y = 0$$

erläutert werden. Die Substitution $y_1(t) := y(t)$, $y_2(t) := y'(t)$ liefert

$$\frac{d}{dt}\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \begin{pmatrix} f_1(t, \vec{y}(t)) \\ f_2(t, \vec{y}(t)) \end{pmatrix}$$

mit

$$f_1(t, \vec{y}) = y_2 \qquad \text{und} \qquad f_2(t, \vec{y}) = -\omega_0^2 \sin y_1$$

Für das Zahlenbeispiel $\omega_0 = 2\pi$ kann die Funktion der rechten Seite $\vec{f}(t, \vec{y})$ folgendermaßen in Scilab programmiert werden (steht ebenfalls am Anfang der Datei `numdgl.sce`):

```
function [f]=mape(t,y)
f(1)=y(2)
f(2)=-4*%pi^2*sin(y(1))
endfunction
```

Die beiden Anfangsbedingungen müssen der Routine `ode` in Form eines Vektors als erstes Argument zur Verfügung gestellt werden. Ansonsten ändert sich nichts, wie das Aufrufbeispiel

```
y0=[0.5;0];t=linspace(0,2,50);
y=ode(y0,0,t,mape);
plot(t,y(1,:))
```

zeigt. Die Sonderrolle der Zeitvariablen t wird deutlich am Beispiel der Differentialgleichung

$$y'' + 4\pi^2(1 - 0,1\cos(4\pi t))y = 0$$

mit der Scilab-Funktion

```
function [f]=pare(t,y)
f(1)=y(2)
f(2)=-4*%pi^2*(1-0.1*cos(4*%pi*t))*y(1)
endfunction
```

Der folgende Aufruf

```
y0=[0.1; 0];
t=linspace(0,8,100);
y=ode(y0,0,t,pare);
plot(t,y(1,:))
```

illustriert den physikalischen Effekt der Parameterresonanz.

22 Wahrscheinlichkeitsrechnung und Statistik

Zufallszahlen kann man mit der Funktion `rand` erzeugen. Zunächst hat man die Verteilung einzustellen: Mit dem Aufruf `rand('uniform')` wählt man eine Gleichverteilung im Intervall $[0,1]$, anschließend liefern Anweisungen der Art `x=rand()` gleichverteilte Zufallszahlen zwischen 0 und 1. Mit `rand('normal')` wird die Standardnormalverteilung eingestellt, danach liefern Anweisungen der Form `x=rand()` Zufallszahlen, die normalverteilt sind mit Erwartungswert 0 und Varianz 1. Mit A=`rand`(m,n) erhält man eine $m \times n$-Matrix **A**, deren Matrixelemente Zufallszahlen mit der eingestellten Verteilung sind. So liefert `x=rand(1,2000)` einen Zeilenvektor mit 2000 Zufallszahlen der eingestellten Verteilung. Abb. 4 zeigt als Ergebnis die Histogramme von 2000 auf diese Weise erzeugten Zufallszahlen für beide mögliche Verteilungen.

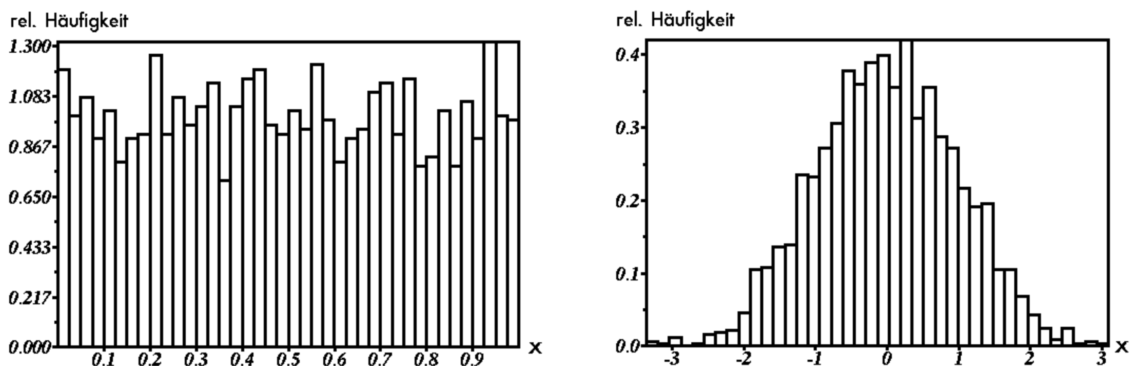


Abbildung 4: Histogramme von 2000 mit `rand` erzeugten Zufallszahlen, links gleichverteilte, rechts standardnormalverteilte Zufallszahlen

Das Verfahren, das von `rand` verwendet wird, gilt heute als veraltet. Es wird daher empfohlen, für wissenschaftliche Zwecke (beispielsweise Diplomarbeiten) die etwas

aufwändigere Funktion `grand` zu nehmen. `y=grand(k,m,'bin',n,p)` erzeugt eine $k \times m$-Matrix `y`, deren Matrixelemente nach der Binomialverteilung mit den Parametern n und p verteilt sind, analog `y=grand(k,m,'poi',mu)` Poisson-verteilte Zufallszahlen mit Erwartungswert $\mu =$ `mu`. Normalverteilte Zufallszahlen mit Erwartungswert $\mu =$ `mu` und Standardabweichung $\sigma =$ `sigma` erhält man durch `y=grand(k,m,'nor',mu,sigma)`. Abb. 5 zeigt als Ergebnis die Histogramme von 2000 mit `grand` erzeugten Zufallszahlen für die Binomial- und die Poissonverrteilung. Weitere Verteilungen können ebenfalls eingestellt werden. Näheres hierzu siehe das Help-Menu (erhältlich mit `help grand`).

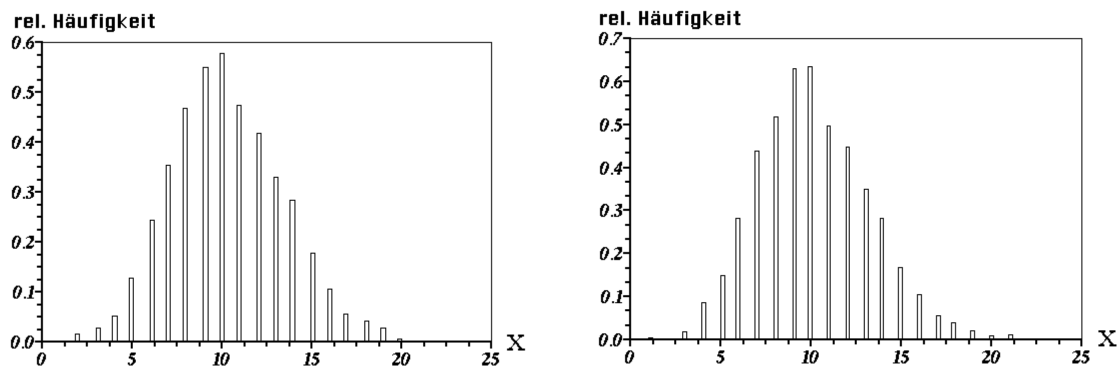


Abbildung 5: Histogramme von 2000 mit `grand` erzeugten Zufallszahlen, links nach der Binomialverteilung mi $n = 1024$ und $p = 0,01$, rechts nach der Poissonverteilung mit $\mu = 10,24$ verteilte Zufallszahlen

Histogramme von Daten, die als Vektor y vorliegen, können in der Form `histplot(n,y)` erzeugt werden. Dabei gibt die positive ganze Zahl n die Zahl der Klassen an. Statt n kann auch ein Vektor x übergeben werden, die Klassen sind dann durch $[x(k), x(k+1)]$ definiert. Diese Funktion wurde bei der Erstellung von Abb. 4 und Abb. 5 verwandt.

Die Verteilungsfunktionen der üblichen Verteilungen sind direkt verfügbar. So erhält man durch `Fx=cdfnor("PQ",t,mu,sigma)` den Funktionswert $F_X(t)$ der Standardnormalverteilung zum Erwartungswert $\mu =$ `mu` und zur Standardabweichung $\sigma =$ `sigma`. Den Funktionswert x der Umkehrfunktion an der Stelle q, also das Quantil der Ordnung q, erhält man durch `x=cdfnor("X",mu,sigma,q,1-q)`. Es erfüllt $F_X(x) = q$.

Für die Binomialverteilung erhält man die Wahrscheinlichkeiten

$$p_k = P\big(\{X = k\}\big) = \binom{n}{k} p^k (1-p)^{n-k}$$

durch den Aufruf `pb=binomial(p,n)`. Der Vektor `pb` enthält dann in der $(k+1)$. Komponenten den Wert der Wahrscheinlichkeit p_k. Die Verschiebung ist notwendig, da die Indizes bei Scilab von 1 an gezählt werden (also `pb(0)` zu einer Fehlermeldung führen würde). Die Funktion `cdfbin` berechnet jedoch kontinuierliche Näherungswerte der Verteilungsfunktion, die entsprechende Funktion interpoliert die diskreten Funktionswerte der Verteilungsfunktion, siehe hierzu die Hinweise im Help-Menu.

Für die grafische Darstellung diskreter Verteilungen ist die Funktion `bar` nützlich, siehe hierzu das Beispiel am Ende von Abschnitt 11. Für weitere Funktionen zur Wahrscheinlichkeitsrechnung und Statistik wird auf das Help-Menu verwiesen.

23 Behandlung von Tondateien

Scilab ermöglicht die Verarbeitung von Tondateien im Waveform Audio File Format (WAVE) mit der Dateiendung `.wav`. Das Lesen erfolgt durch

```
[x,sf,bits]=wavread('Dateiname.wav');
```

wobei die Zeichenkette '*Dateiname*' auch eine Pfadangabe enthalten kann. Die Abtastwerte des Tonsignals steht dann in der Matrix `x` mit zwei Zeilen (bei Stereo-Signalen) oder einem Zeilenvektor zur Verfügung. `sf` gibt die Abtastfrequenz an, hier ist der Wert 44100 üblich, und die Variable `bits` gibt die Zahl der Bits je Abtastwert an. Hier ist 16 ein verbreiteter Wert

Mit der Funktion

```
wavwrite(x,sf,bits,'Dateiname.wav');
```

können Tonsignale wieder in eine Datei geschrieben werden, auch hier kann die Zeichenkette '*Dateiname*' eine Pfadangabe enthalten. Die Parameter `sf` und `bits` haben dieselbe Bedeutung wie beim Lesen. Wird der Parameter `bits` weggelassen, so wird hierfür der Default-Wert 16 genommen, was durchaus üblich ist. Der Parameter `sf` sollte jedoch angegeben werden, denn der Default-Wert hierfür ist 22050, was inzwischen kaum noch üblich ist. Das Weglassen dieses Parameters beim Schreiben kann also beim Wiederabspielen einer verarbeiteten Audio-Datei zu einer Überraschung führen! Das Schreiben von Dateien und das anschließende Lesen kann mit Rundungsfehlern verbunden sein, dabei kann ein relativer Fehler von 0.00009 auftreten (das ist fast ein Promille).

Für die weiteren Funktionen zur Verarbeitung von Tondaten wird auf das Kapitel „Sound file handling“ in den Hilfe-Seiten verwiesen.

24 Werkzeuge für Wavelets

24.1 Zur Bibliothek

Für Wavelets steht auf

`http://alexanderstoffel.selfip.org/wavelib.html`

eine Archivdatei `wavelib.zip` zur Verfügung, die analog zu den Anweisungen für die Bilbliothek `mathlib` in Abschnitt 2 heruntergeladen und installiert werden kann. Anschließend stehen verschiedene Makros zur Verfügung, die für die Lehrveranstaltung „Wavelets“ benötigt werden. Zur Wavelet-Transformation wird hier auf das Skript „Wavelets und Filterbänke“ verwiesen, das auf

`http://alexanderstoffel.selfip.org/mapdf/wavelet.pdf`

erhältlich ist.

Einen Überblick über die Funktionen der Bibliothek erhält man am einfachsten über das Inhaltsverzeichnis des Help-Browsers (linke Spalte des Fensters). Für jedes Kapitel erhält man eine Übersicht der verfügbaren Funktionen. Im Verzeichnis `wavelib` sind — entsprechend den Kapiteln im Help-Browser — Unterverzeichnisse mit Beispielprogrammen. Zu jeder Funktion der Bibliothek steht im zum Kapitel gehörigen Unterverzeichnis `examples` ein Beispielprogramm, der Dateiname ist gebildet nach dem Schema *Funktionsname*+`ex.sce`. So steht zur Funktion `psihut` im Verzeichnis `examples/functions` ein Beispielprogramm `psihutex.sce`, das man über das Menu „Ausführen“ aufrufen kann.

Über die Variable `WLHOME` ist der Pfad zu den Unterverzeichnissen der Bibliothek erhältlich. Man kann also das Beispielprogramm auch durch Eintippen des Befehls

```
exec(WLHOME+'/examples/functions/psihutex.sce')
```

ausführen. Durch Eintippen von `wavelib` bekommt man alle Funktionsnamen dieser Bibliothek aufgelistet, eine ausführliche Beschreibung erhält man auch durch Eintippen von `help` *Funktionsname*. Hier wird daher nur eine grobe Übersicht gegeben über einige besonders häufig benötigte Makros.

24.2 Skalierungsfunktionen und Wavelets

Die Funktionswerte des CDF(3,5)-Wavelets können durch `y=psicdf35(x)` ausgerechnet werden, dabei darf `x` ein Vektor sein. Eine grafische Darstellung kann man sich also leicht durch

```
x=-3:0.01:4; y=psicdf35(x); plot(x,y);
```

erzeugen. Die zugehörige Skalierungsfunktion erhält man durch `y=phicdf35(x)`. Entsprechend erhält man mit `psihut` das CDF(2,2)-Wavelet und mit `psihaar` das Haar-Wavelet. Die zugehörigen Skalierungsfunktionen bekommt man, indem man jeweils „`psi`“ durch „`phi`“ ersetzt.

24.3 Wavelet-Transformationen für eindimensionale Signale

24.3.1 Einstufige Transformationen

Durch `y=sigwt(x,'Haar')` erhält man die normierte Haar-Wavelet-Transformierte des Vektors `x`. Der Hochpassausgang (Koeffizienten $d(k)$) steht in der oberen Hälfte des Vektors `y`, also bei einer Länge N des Vektors `x` in den Speicherplätzen $\frac{N}{2}+1$ bis N, der Tiefpassausgang (Koeffizienten $s(k)$) in den Speicherplätzen 1 bis $\frac{N}{2}$ (N muss gerade sein). Dies ist in Abbildung 6 gezeigt. Dabei ist die Nummerierung auf die übliche von 0 bis $N-1$ korrigiert (gegenüber der in `Scilab` von 1 bis N). Die ursprünglichen Werte (inverse Transformation) erhält man durch `xr=sigwtinv(y,'Haar')` zurück (inverse Transformation). Beachten Sie, dass aufgrund von Rundungsfehlern das so zurückgewonnene Signal `xr` nicht exakt mit dem ursprünglichen Signal `x` übereinstimmt!

Wählt man als zweiten Parameter in `sigwt` den Namen eines anderen Wavelets, so wird die entsprechende Wavelet-Transformation durchgeführt, beispielsweise erhält man durch `y=sigwt(x,'Hut')` die Wavelet-Transformation bezüglich des Wavelets CDF(2,2), analog mit erhält man die zugehörige inverse Transformation durch `xr=sigwtinv(y,'Hut')`. Welche Wavelet-Transformationen derzeit verfügbar sind, ist Tabelle 1 zu entnehmen.

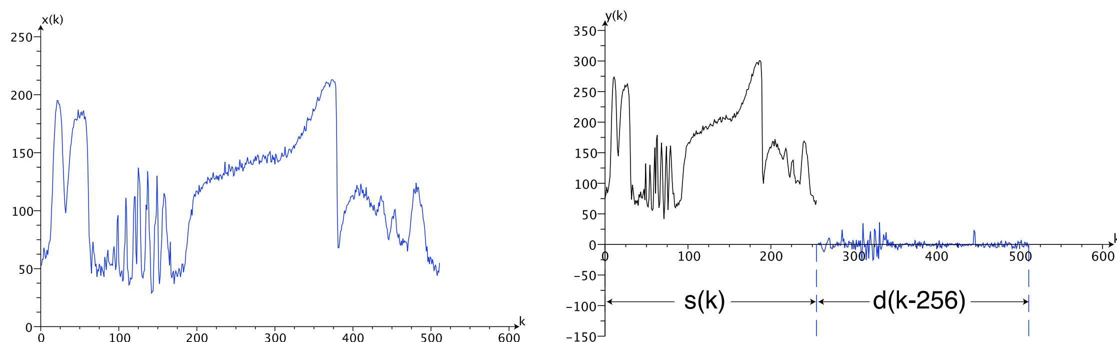


Abbildung 6: eindimensionales Signal `x` (links, 500. Zeile des Testbildes „Lena“) und die Wavelet-Transformierte `y=sigwt(x,'Haar')` (rechts)

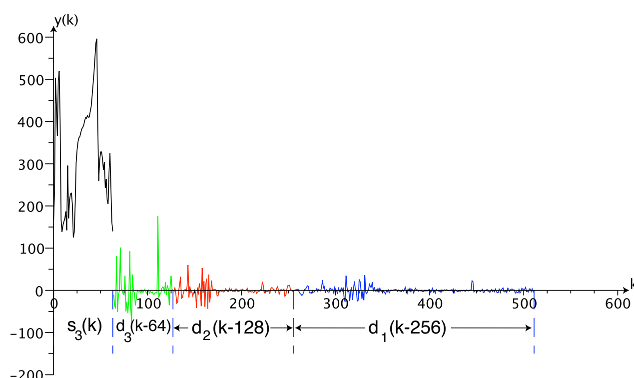


Abbildung 7: Ergebnis der dreistufigen Filterbanktransformation `y=sigwt(x,'Haar',3)` angewandt auf dasselbe Eingangssignal wie in Abb. 6

Wavelet	zweiter Parameter	Optionen	Default
Haar-Wavelet	Haar	n,u	n
CDF(2,2), Hut	CDF(2,2), CDF22, Hut, hat	rr, zp, vm	rr
CDF(3,5)	CDF(3,5), CDF35	rr, zp	rr
DD(4,2), Binlet	DD(4,2), DD42, Bin	rr, zp, vm	rr
DD(4,4)	DD(4,4), DD44	rr, zp	rr
DD(8,6)	DD(8,6), DD86	rr, zp	rr
AI(3,3) (av. int.)	AI(3,3), AVI(3,3), AI33, AVI33	rr, zp	rr
AI(5,3) (av. int.)	AI(5,3), AVI(5,3), AI53, AVI53	rr, zp	rr
AI(5,5) (av. int.)	AI(5,5), AVI(5,5), AI55, AVI55	rr, zp	rr
FBI, 9/7	FBI	rr, zp	rr
D4 (orthogonal)	D4, Daub4, D(4)	rr, zp, pe, gs(1d)	gs(1d), rr(2d)
D6 (orthogonal)	D6, Daub6, D(6)	rr, zp, pe	pe
Symlet	Sym	pe, rr, zp	pe
QN(4), Quincunx	Quisu, Quincunx	wird ignoriert	

Tabelle 1: Verfügbare Wavelettransformationen, der 2. Parameter und gegebenenfalls der 4. Parameter (Spalte „Optionen“) sind als Zeichenkette zwischen Hochkommas oder Anführungszeichen zu übergeben. Dabei spielt Groß- oder Kleinschreibung keine Rolle, statt 'DD42' kann also auch 'dd42' übergeben werden. Beachten Sie auch die Hinweise in Tabelle 2.

24.3.2 Mehrstufige Transformationen

Den Tiefpassausgang der 1. Stufe (also die erste Hälfte des Vektors von `y=sigwt(x,'Haar')`) kann man als Eingang erneut auf die Filterbank geben und diesen Vorgang wiederholen. Dies kann einfacher durch Aufruf von `y=sigwt(x,'Haar',p)` geschehen. Dabei wird der Ausgang des Hochpassfilters jeweils in der verbleibenden Hälfte des Vektors mit den höheren Indizes abgespeichert, der Ausgang des Tiefpassfilters in der mit den kleineren Indizes. Auf diesen wird dann die nächste Stufe der Transformation angewandt. `x` ist das Eingangssignal, `p` die Gesamtzahl der Stufen. Damit dies glatt aufgeht, muss die Gesamtzahl der Komponenten des Eingangsvektors `x` ein Vielfaches von 2^p sein. Man kommt damit zu folgender Anordnung der Filterausgänge im zurückgegebenen Vektor `y`:

Typ	Speicherplätze		
$d_1(k)$	$\frac{N}{2}+1$	$\cdots$	N
$d_2(k)$	$\frac{N}{4}+1$	$\cdots$	$\frac{N}{2}$
$d_3(k)$	$\frac{N}{8}+1$	$\cdots$	$\frac{N}{4}$
$\vdots$		$\vdots$	
$d_p(k)$	$\frac{N}{2^p}+1$	$\cdots$	$\frac{N}{2^{p-1}}$
$s_p(k)$	1	$\cdots$	$\frac{N}{2^p}$

Dabei sind $d_m(k)$ die Hochpassausgänge, $s_p(k)$ ist der verbleibende Tiefpassausgang, steigender Index m bedeutet eine gröbere Skalierungsstufe. Dies ist in Abb. 7 verdeutlicht. Beachten Sie, dass `Scilab` die Indizes von 1 bis N durchnummeriert. In den Abbildungen sind die Indizes auf die übliche Nummerierung von 0 bis $N-1$ korrigiert.

Durch die inverse Transformation `xr=sigwtinv(y,'Haar',p)` erhält man die ursprünglichen Werte zurück. Auch hier darf man aufgrund von Rundungsfehlern keine exakte Übereinstimmung mit den ursprünglichen Werten `x` erwarten!

Ersetzt man den Parameter `'Haar'` durch `'Hut'`, dann erhält man ebenfalls die mehrstufige Wavelet-Transformation (bzw. ihre Inverse) bezüglich des Wavelets CDF(2,2), analog mit `y=sigwtm(x,'Bin',p)` sowie `xr=sigwtinv(y,'Bin',p)` die Transformation bezüglich des 9/7-Binlets (Filter DD(4,2)). Entsprechend können auch die andern Parameter aus der 2. Spalte von Tabelle 1 verwandt werden. Bei der Zeichenkette wird Groß- und Kleinschreibung ignoriert, es kann also auch `'fbi'` statt `'FBI'` als zweiter Parameter angegeben werden.

Als vierten Parameter kann man an die Funktionen `sigwt` und `sigwtinv` eine Zeichenkette übergeben, die bei fast allen Wavelet-Transformationstypen eine bestimmte Variante der Behandlung der Daten am Rand auswählt. Die möglichen Zeichenketten sind in Tabelle 1 in der Spalte „Optionen“angegeben. Beim Haar-Wavelet entscheidet der dritte Parameter darüber, ob die normierte Transformation oder die nicht normierte Transformation mit den Filterkonstanten $\pm\frac{1}{2}$ und ± 1 durchgeführt wird. Beachten Sie, dass das Quincunx-Neville Filter QN(4) zu einer zweidimensionalen Transformation gehört, also 'Quisu' für `sigwt` und `sigwtinv` nicht als 2. Eingangsparameter zugelassen ist. (Hierfür wird bei Benutzung von `imwt` und `imwtinv` der 4. Parameter ignoriert).

Und hier die Bedeutung des 4. Eingangsparameters für `sigwt` und `sigwtinv` (gilt entsprechend auch für die Transformationen von Bilddaten mit `imwt` und `imwtinv`):

'rr' (reflection with repetition) Symmetrische Fortsetzung am Rand mit Wiederholung. Dies ist für die meisten Wavelet-Typen der Default-Wert.

'zp' (zero padding) Fortsetzung durch 0 am Rand (d.h. in den Formeln werden Werte, die sich beispielsweise in $s(-1)$ auf außerhalb des Bereichs der gegebenen Daten beziehen, durch 0 ersetzt.

'pe' Periodische Fortsetzung der Daten am Rand. Diese Option ist (gegenwärtig) nur für die orthogonalen Daubechies-Wavelets D4, D6 und das Symlet implementiert, sie ist bei D6 und beim Symlet der Default-Wert.

'vm' (vanishing moments) Spezielle Randfilter sorgen dafür, dass Signale der Form $x(k) = k^n$ für $n < p$ bei einer p-fachen Nullstelle von $H(z)$ in $z = -1$ auch am Rand im Hochpassteil auf 0 führen, analog bei Signalen der Form $x(k) = (-1)^k k^n$ für den Tiefpassanteil. Diese Option ist nur für das Hut-Wavelet und das DD(4,2)-Wavelet implementiert und führt bei mehrfacher Anwendung (3. Parameter in `sigwt` größer als 1) zur numerischen Instabilität.

'gs' Gram-Schmidt-Orthogonalisierung. Diese Option ist gegenwärtig nur für eindimensionale Signale für D4 implementiert, dort aber der Default-Wert.

'n' normierte Transformation, nur für das Haar-Wavelet möglich, es werden die Filter mit den Filterkonstanten $\pm\frac{1}{2}\sqrt{2}$ benutzt. Dies ist auch der Default-Wert für das Haar-Wavelet.

'u' nicht normierte (unnormierte) Transformation nur für das Haar-Wavelet und nur für die Transformation eindimensionaler Signale möglich, es werden die Filter mit den Filterkonstanten $\pm\frac{1}{2}$ und ± 1 benutzt.

Wenn kein 4. Parameter übegeben wird, dann wird der jeweilige Default-Wert genommen. Dies ist für die Praxis zu empfehlen, wenn nicht Experimente zum Einfluss der Randbedingungen durchgeführt werden. Werden ungültige Werte als 4. Eingangsparameter übergeben, so werden diese durch den jeweiligen Default-Wert ersetzt — in der Regel mit einer entsprechenden Warnung. Für das Symlet führen andere Optionen als der Default-Wert `'pe'` zu numerischen Problemen und werden daher nur mit einer entsprechenden Warnung ausgeführt. Beachten Sie auch, dass Sie in jedem Fall den 3. Parameter übergeben müssen, wenn Sie einen andern als den Default-Wert für die Randbedingungsoption auswählen möchten. Wenn nur eine Transformationsstufe ausgeführt werden soll, dann ist eine 1 zu übergeben. So wird beispielsweise die Binlet-Transformation einmal mit der Option „vanishing moments“ mit `y=imwt(x,'DD(4,2)',1,'vm')` aufgerufen.

Für Einzelheiten wird auf Kapitel 3.4 des Skriptes „Wavelets und Filterbänke“ verwiesen. Die jeweils implementierten Optionen mit den zugehörigen Default-Werten sind in Tabelle 2 zusammengestellt.

Wavelet	Optionen	Default	Hinweise
Haar-Wavelet	n,u	n	für Bilder nur n
CDF(2,2), Hut	rr, zp, vm	rr	
CDF(3,5)	rr, zp	rr	
DD(4,2), Binlet	rr, zp, vm	rr	
DD(4,4) und DD(8,6)	rr, zp	rr	
AI(n,m) (average interpolating)	rr, zp	rr	
FBI, 9/7	rr, zp	rr	
D4 (orthogonal)	rr, zp, pe, gs (1d)	gs (1d), rr (2d)	
D6 (orthogonal)	rr, zp, pe	pe	
Symlet	rr, zp, pe	pe	nur pe ratsam
QN(4), Quincunx-Neville	wird ignoriert		nur für Bilddaten

Tabelle 2: Hinweise zur Option, die als 4. Parameter als Zeichenkette an **`sigwt, sigwtinv, imwt`** und **`imwtinv`** übergeben werden kann. Beachten Sie auch Tabelle 1.

24.4 Behandlung von Bilddaten

24.4.1 Wavelet-Transformationen für Bilddaten

Analog zur Transformation eindimensionaler Signale erhält man die einstufige Haar-Wavelet-Transformation für Bilddaten durch `W=imwt(A,'Haar')`, die inverse Transformation durch `Ar=imwtinv(W,'Haar')`. Dabei sind die Bilddaten in der Matrix `A` bereitzustellen, deren Zeilen- und Spaltenzahl gerade sein muss. Die Waveletkoeffizienten stehen dann in einer gleich großen Matrix `W` in der üblichen Reihenfolge (links oben das HH-Band als Ergebnis einer Tiefpassfilterung in Zeilen- und Spaltenrichtung).

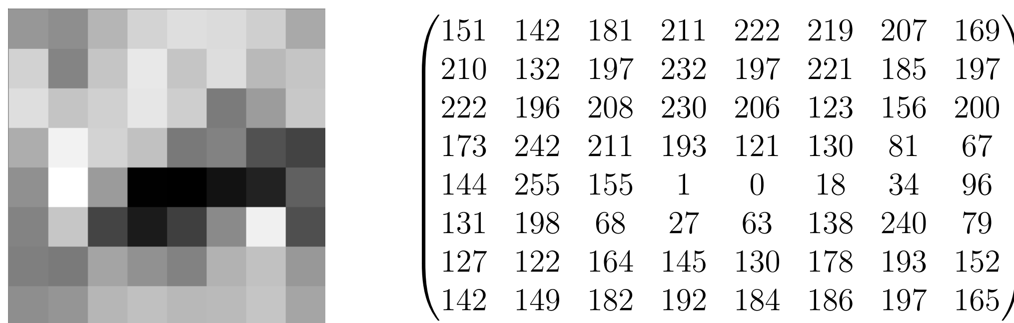
Durch `W=imwt(A,'Haar',p)` erhält man eine p-stufige Haar-Wavelet-Transformation, die zugehörige Inverse durch `Ar=imwtinvm(W,'Haar',p)`. Die Zeilen- und Spaltenzahl der Bildmatrix `A` muss dabei durch 2^p teilbar sein, `p` gibt die Zahl der Stufen an. Man erhält ganz analog wie für die eindimensionalen Signale die übrigen Wavelet-Transformationen, indem man die Zeichenkombination `'Haar'` durch die jeweilige Zeichenkette entsprechend ersetzt, die in Tabelle 1 angegeben ist.

Für Bilddaten gibt es auch Wavelet-Transformationen, die keine entsprechende Transformation für eindimensionale Signale besitzen. Eine einfache derartige Transformation ist implementiert, allerdings nur für den Fall, dass die Zahl der Transformationsstufen gerade ist. Man hat hierfür als 2. Parameter an `imwt` und `imwtinv` die Zeichenkette `'Quisu'` oder `'Quincunx'` zu übergeben. Beachten Sie, dass hierfür nur eine gerade Anzahl von Transformationen möglich ist, der dritte Eingangsparameter von `imwt` und `imwtinv` muss also angegeben werden und **gerade** sein.

24.4.2 Ein- und Ausgabe von Bilddaten

`[A,head] = reimppm('`*Dateiname*`')` liest die Bilddaten einer Bilddatei im PBM-, PGM- oder PPM-Format in die Matrix `A`; `head` enthält nach dem Aufruf den Header der Datei als Textmatrix. Beachten Sie, dass die Bilddaten nach dem Aufruf in der Matrix als Zahlen vom Typ `uint8` vorliegen, also als positive ganze Zahlen mit maximal 8 Bit. Das kann bei der Weiterverarbeitung zu Problemen bis hin zu scheinbar unerklärlichen Fehlern führen. Vor einer rechnerischen Weiterverarbeitung ist daher zu empfehlen, die Daten durch `A=double(A)` in den „normalen“ Datentyp von Scilab umzuwandeln. Wenn die Bilddatei nicht im aktuellen Arbeitsverzeichnis liegt, dann ist der Pfad zu der Bilddatei mit anzugeben, beispielsweise in der Form `[A,header]=reimppm('Testbilder/boats512.pgm');`, wenn die Bilddatei im Unterverzeichnis `Testbilder` des aktuellen Verzeichnisses liegt. Der vollständige Dateiname, also beispielsweise mit der Endung „`.pgm`“ muss angegeben werden.

Wird nur ein Rückgabeparameter angegeben, so enthält dieser die Bilddaten, und die Übergabe des Headers entfällt. In Abb 8 ist ein Graustufenbild und die mit `reimppm` gewonnene Matrix gezeigt. Die Funktion `reimppm` ist eine leicht modifizierte Version eines Programms von Ramine Nikoukhah.



$$\begin{pmatrix} 151 & 142 & 181 & 211 & 222 & 219 & 207 & 169 \\ 210 & 132 & 197 & 232 & 197 & 221 & 185 & 197 \\ 222 & 196 & 208 & 230 & 206 & 123 & 156 & 200 \\ 173 & 242 & 211 & 193 & 121 & 130 & 81 & 67 \\ 144 & 255 & 155 & 1 & 0 & 18 & 34 & 96 \\ 131 & 198 & 68 & 27 & 63 & 138 & 240 & 79 \\ 127 & 122 & 164 & 145 & 130 & 178 & 193 & 152 \\ 142 & 149 & 182 & 192 & 184 & 186 & 197 & 165 \end{pmatrix}$$

Abbildung 8: Lesen von Bilddateien (Beispiel links) mit `reimppm`, rechts die gelesene Matrix mit den Bilddaten (0 entspricht schwarz, 255 weiß)

`writepgm('`*Dateiname*`',A)` schreibt die Daten der Matrix `A` als Graustufenwerte interpretiert in eine Bilddatei mit dem Namen *Dateiname*`.pgm`. Daher darf der String '*Dateiname*' die Endung „`.pgm`“ nicht mit enthalten. Er darf aber einen Pfad enthalten, wenn das Abspeichern im aktuellen Verzeichnis nicht erwünscht ist. Die Matrixelemente von `A` dürfen nicht außerhalb des Intervalls von 0 bis 255 liegen.

`imview(A)` zeigt die Daten der Matrix `A` als Graustufenwerte interpretiert in einem Grafikfenster. Auch hier dürfen die Matrixelemente von `A` nicht außerhalb des Intervalls

von 0 bis 255 liegen. `imview` ändert die Grafik-Einstellungen, insbesondere die Farbskala (`colormap`). Um das übliche Verhalten der Grafikprogramme zurückzuerhalten, empfiehlt es sich, nach dem Aufruf von `imview` alle Grafikeinstellungen durch `sdf();` zurückzusetzen. Das folgende Programmbeispiel

```
A=zeros(512,512);
for i=1:512;for k=1:512;
     A(i,k)=255*abs(sin(i*%pi/256)*cos(k*%pi/256));
end;end;
imview(A)
```

liefert das in Abb. 9 gezeigte Bild. Die Funktion `imview` ist eine leicht modifizierte Version eines Programms von Ramine Nikoukhah. Die Übergabe einer Zeichenkette als optionalen zweiten Parameter an `imview` bewirkt eine Überschrift über dem Bild, so kann im gerade gegebenen Programmbeispiel `imview(A)` durch `imview(A,'Testbild, Funktionswerte')` ersetzt werden.

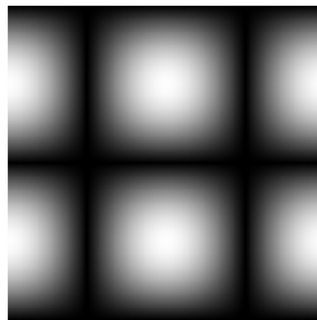


Abbildung 9: Mit `imview` grafisch dargestellte (512×512)-Matrix mit Matrixelementen $a_{ik} = \left|255 \cdot \sin\left(\frac{i\pi}{256}\right) \cos\left(\frac{k\pi}{256}\right)\right|$

`B=grvalues(A)` bringt die Matrixelemente der Matrix `A` in das Intervall $[0, 255]$. Dies ist nützlich vor einer Weiterverwendung der Daten mit `imview` oder `writepgm`. Dabei werden die Daten so transformiert, dass das Minimum bei 0 und das Maximum bei 255 liegt. Die Transformation der Werte erfolgt nach folgendem Schema: Sei $d_{\min}$ der Wert des kleinsten, $d_{\max}$ der Wert des größten Matrixelements, $c := \frac{1}{d_{\max} - d_{\min}}$, $h := -d_{\min} \cdot c$. Dann erfolgt die Berechnung der Matrixelemente b_{ik} von `B` aus den Elementen a_{ik} von `A` nach dem Schema

$$b_{ik} = 255(c \cdot a_{ik} + h)$$

Das maximale Matrixelement wird so auf 255, das minimale auf 0 abgebildet. Wenn $d_{\max} - d_{\min}$ numerisch nahe 0 ist, werden alle Matrixelemente von `B` auf 127 gesetzt.

Die Anwendung von `grvalues` wird durch folgendes Programmbeispiel demonstriert, in dem die Waveletkoeffizienten nach einer Haar-Wavelet-Transformation in Grauwerte umgerechnet werden, und zwar jedes der vier Subbänder einzeln.

```
[A,header]=reimppm('Testbilder/Marseille.pgm');
```

```
W=imhaarwt(A);
bdi=size(W);
nrh=bdi(1)/2; nch=bdi(2)/2;
// gray values separately for the subbands:
HH=grvalues(W(1:nrh,1:nch));
GG=grvalues(W(nrh+1:2*nrh,nch+1:2*nch));
HG=grvalues(W(1:nrh,nch+1:2*nch));
GH=grvalues(W(nrh+1:2*nrh,1:nch));
B=[HH,HG;GH,GG];
clf(); imview(B);
```

Die resultierenden Teilmatrizen werden wieder zu einer Matrix zusammengesetzt. Das Ergebnis ist in Abb. 10 zusammen mit dem ursprünglichen Bild gezeigt.

Abbildung 10: Mit `grvalues` in Grauwerte umgewandelte und mit `imview` dargestellte Waveletkoeffizienten (Haar-Wavelet, rechts) des links gezeigten Bildes

`B=grayclip(A)` wandelt die Zahlenwerte in der Matrix `A` ebenfalls in den Bereich $[0, 255]$ um, jedoch durch „Abschneiden“ oder „clipping“. Werte über 255 werden einfach auf 255 und negative Werte auf 0 gesetzt, die Werte, die sich bereits in $[0, 255]$ befinden, bleiben unverändert. Dies empfiehlt sich, wenn man die Werte außerhalb von $[0, 255]$ als Ausreißer oder Ausnahmefälle betrachten kann (z.B. beim Gibbs-Phänomen) und deswegen die übrigen Werte nicht verändern möchte.

24.5 Das Rechnen mit Filtern

Da Scilab erlaubt, mit Polynomen und gebrochen rationalen Funktionen intuitiv umzugehen (siehe hierzu den Abschnitt 18), ist es möglich, einige Rechnungen mit z-Transformierten von Filtern durchzuführen. Hierzu dienen vor allem folgende Funktionen

`F = ztrans(f,fkmin,z)` führt die z-Transformation aus. Dabei ist `f` ein Vektor mit den Filterkonstanten. `fkmin` gibt den Index der Filterkonstante `f(1)` an. `z` ist die Polynomvariable. Beim FIR-Filter, das durch $f(\pm 2) = 1$, $f(\pm 1) = 4$, $f(0) = 6$ gegeben ist, hat man also die Parameter durch `f=[1 4 6 4 1]` und `fkmin=-2` zu initialisieren. Bei der z-Transformierten ist für die Polynomvariable der Buchstabe `z` üblich; dies erreicht man durch die Initialisierung `z=poly(0,'z')`. Die Rückgabevariable `F` enthält dann die z-Transformierte als rationale Funktion. Dies ist hinsichtlich der Ausgabe reichlich unglücklich. Eine bessere Lösung wäre zweifellos, einen eigenen Typ in Scilab für Laurent-Polynome einzuführen. Rationale Funktionen können

jedoch in Scilab problemlos miteinander multipliziert werden, und das ist für die Hintereinanderausführung von Filteroperationen wesentlich.

`[f,fkmin] = iztrans(F)` führt die inverse z-Transformation durch. Der Eingangsparameter `F` muss eine rationale Funktion sein, die nur eine Potenz von z im Nenner hat, sonst ist sie nicht z-Transforierte eines FIR-Filters. Die Rückgabe erfolgt so, wie die Eingabe in der z-Transformation mit `ztrans` zu erfolgen hat: `f` ist ein Vektor mit den Filterkonstanten, `fkmin` gibt den Index der Filterkonstante `f(1)` an.

`H = compos(F,G)` berechnet die zusammengesetzte Funktion (Komposition) $F(G(Z))$ von zwei Laurent-Polynomen $F(z)$ und $G(z)$. Bei den meisten Wavelet-Anwendungen wird diese Funktion zur Berechnung von Ausdrücken der Form $H(-z^{-1})$ benutzt. Dies kann durch `compos(H,-z^(-1))` erfolgen, wenn `z` durch `z=poly(0,'z')` initialisiert wurde.

`tabelle = tabfiltco(F)` liefert eine Tabelle der Filterkoeffizienten für das Laurent-Polynom `F(z)` (in Scilab als rationale Funktion abgespeichert). Die Tabelle besteht aus einer Matrix mit zwei Spalten, in der ersten Spalte stehen die Indizes, in der zweiten die Koeffizienten.

`showfilt(F)` gibt das Laurent-Polynom wie ein Polynom an der Konsole aus (mit negativen Exponenten). Man sollte das Fenster soweit vergrößern, dass die Ausgabe in eine Zeile passst. Mit `disp(F)` wird dagegen $F(z)$ als rationale Funktion mit einer Potenz z^p im Nenner ausgegeben. In einem zweiten, optionalen Parameter kann eine Zeichenkette für den Namen des Laurent-Polynoms übergeben werden. So ergibt `showfilt(z^2*(1+z^(-1))^5,'H');` die Ausgabe

```
        2     1            -1      -2     -3
H(z)=z  + 5z  + 10 + 10z   + 5z    + z
```

24.6 Experimente zur Kodierung

Hierzu stehen folgende Funktionen zur Verfügung:

`x = scanWL(W,p)` bringt die Waveletkoeffizienten von Bildern, wie man sie beispielsweise durch `W = imwt(A,'FBI',steps);` erhält und die als Matrix `W` vorliegen, in einen Spaltenvektor. Dies geschieht so, dass man durch eine Lauflängenkodierung eine möglichst große Einsparung an Zeichen erreicht. Zunächst wird das HH-Subband zeilenweise „gescannt“. Danach werden die Werte der HG Subbänder spaltenweise in den Ergebnisvektor `x` übertragen, das Subband zur gröbsten Skalierungsstufe kommt zuerst. Anschließend werden entsprechend die Werte der GH-Subbänder zeilenweise in `x` übertragen. Schließlich werden die Werte der GG-Subbänder in diagonaler Zickzackreihenfolge übertragen, wie dies in Abb. 11 gezeigt ist. Auch hier kommen die Subbänder zur gröbsten Skalierungsstufe zuerst.

`[y,mfv] = runlength(x,mfv)` nimmt eine Lauflängenkodierung der Zahlendaten im Vektor `x` vor. Der optionale Parameter `mfv` gibt dabei an, welches Zeichen durch seine Lauflänge kodiert werden soll. Sinnvoll ist hierfür natürlich nur ein sehr häufig vorkommender Wert. Wenn `mfv` als Eingabeparameter fehlt, dann wird dieser Parameter als der am häufigsten vorkommende Wert in `x` berechnet. Bei gleich häufig

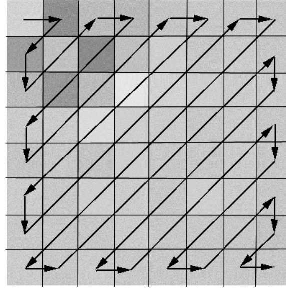


Abbildung 11: Scannen von Subbändern GG durch das Makro `scanWL`

vorkommenden Werten wird der größte gewählt. Das Ergebnis wird in der zweispaltigen Matrix `y` übergeben; der Wert der ersten Spalte gibt die Zahl der Werte `mfv` an, der Wert der 2. Spalte gibt den nächsten von `mfv` verschiedenen Wert an. Ausnahme: wenn `mfv` der letzte Wert in `x` ist, dann wird in der ersten Spalte der letzten Zeile von `y` die um 1 verminderte Lauflänge angegeben, und die zweite Spalte enthält `mfv`.
Beispiel:

```
x=[0,0,0,4,0,0,,5,0,0,0,0,2,0,0];y=runlength(x)
```

liefert für `y`

$$\begin{matrix} 3 & 4 \\ 2 & 5 \\ 4 & 2 \\ 1 & 0 \end{matrix}$$

B = deadzone(A,d,mfv) simuliert eine Totzonenquantisierung. Dabei ist der optionale Parameter `d` der Wert der Schwelle, also die halbe Länge der Totzone. Der Default-Wert von `d` ist 1. Der Wert `mfv` legt den Mittelpunkt der Totzone fest. In den meisten Anwendungsfällen wird dies 0 sein. Wird er nicht angegeben, dann wird er als der häufigste Wert in `A` bestimmt (bei gleich häufigen Werten wird der größte genommen). `A` ist die Matrix oder der Vektor, der die diskreten Daten enthält. Alle Matrixelemente `A(i,k)` oder Vektorkomponenten $A(k)$, die

$$|\texttt{A(i,k)-mfv}| \leq \texttt{d}$$

erfüllen, werden auf `mfv` gesetzt; das Ergebnis wird in `B` abgespeichert. `B` ist also eine Matrix oder ein Vektor derselben Größe wie `A`. Der Nutzen der Totzonenquantisierung ist eine größere Effektivität der Lauflängenkodierung, da die Häufigkeit des am häufigsten vorkommenden Wertes erhöht wird. Gebräuchlicher Wert für den Parameter `d` ist die Stufenhöhe der Quantisierung, so dass diese Stufenhöhe für Werte nahe Null bzw. `mfv` verdoppelt wird.

en = entrop(x) berechnet die Entropie der diskreten Daten im Vektor oder der Matrix `x`. Bezeichnet man mit p_k die relative Häufigkeit des Wertes mit der Nummer k in `x`, dann ist die Entropie

$$-\sum_k p_k \log_2(p_k)$$

wobei die Summe über alle in `x` vorkommenden Werte erfolgt. Der Logarithmus zur Basis 2 kann durch $\log_2(p_k) = \ln(p_k)/\ln(2)$ berechnet werden. Wenn $p_k = 0$,

dann ist der nicht definierte Wert $p_k \log_2(p_k)$ durch Null zu ersetzen, das ist der Grenzwert $\lim\limits_{x \to 0} x \cdot \log_2(x)$. Die Berechnung der Entropie erlaubt eine Abschätzung des Speicherbedarfs bei der Kodierung der Daten in `x`. Die Entropie gibt den theoretischen Mindestspeicherbedarf in Bit je Wert von `x` bei optimaler Kodierung an. Der gesamte Speicherbedarf in Bit für `x` ist also größer als `length(x)*entrop(x)`.