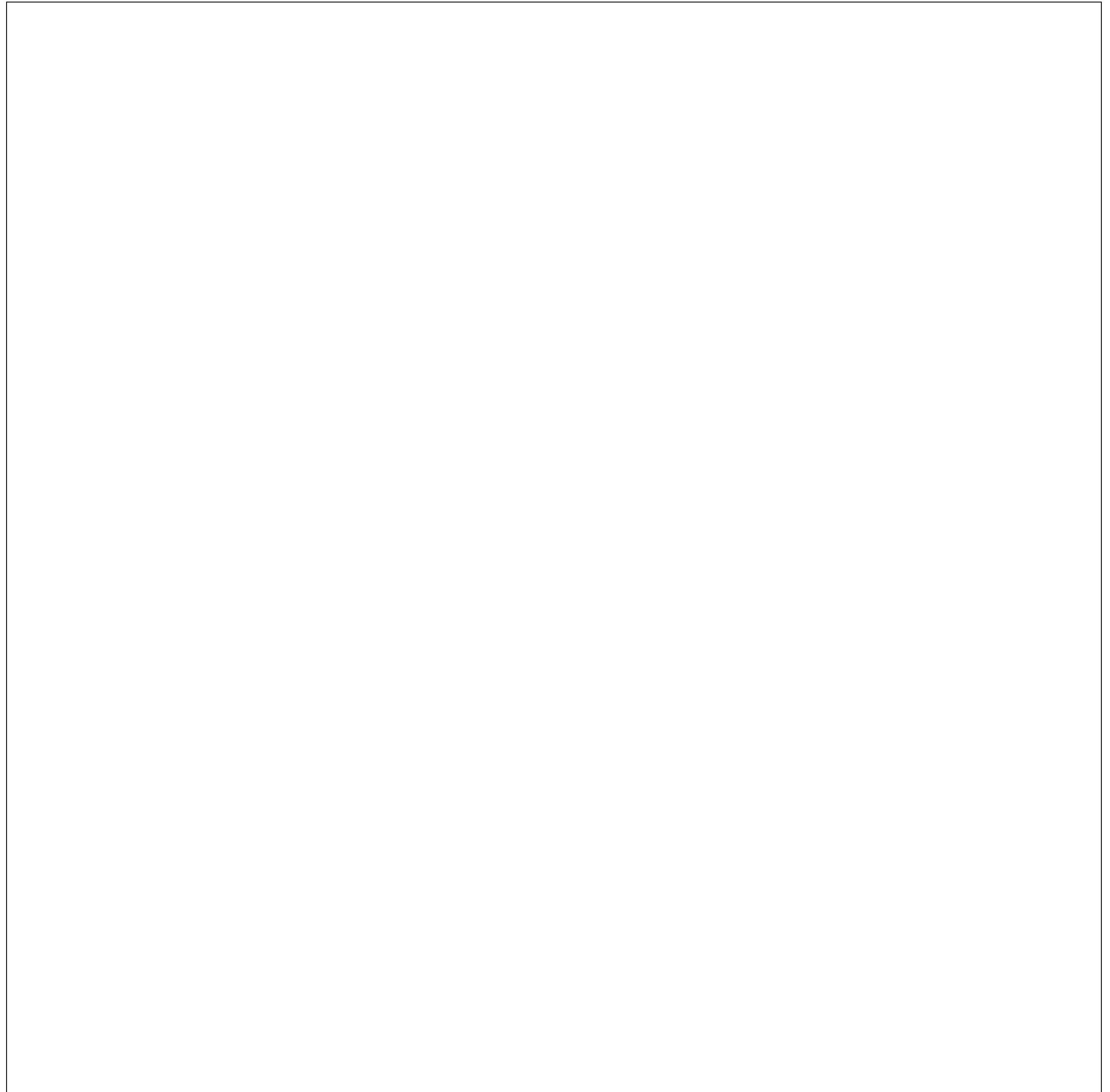
FernUniversität
Gesamthochschule in Hagen

Praktikum unter Benutzung mathematischer Softwarepakete

Autor: K. Veselić
Wissenschaftliche Mitarbeit: I. Nakić
Schrift: R. Cramer



© 2001 Fernuniversität - Gesamthochschule in Hagen 01087 - 2 - 01 - S1

Fachbereich Mathematik

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, bleieben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werks darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Inhalt

0 Einleitung 1

1 Zahlen und Operationen 3

2 Elemente der Programmierung 7

3 Zeilenvektoren 15

4 Matrizen 25

5 Rundungsfehler 35

6 Ausgewählte Aufgaben 39

0 Einleitung

Dieser Kurs soll einen Einstieg zum Umgang mit der modernen mathematischen Hochleistungsoftware bieten. Wir haben uns für den „Matlab-Dialekt“, vornehmlich für das Programm-Paket Matlab* [1] entschlossen. Die Gründe dafür sind vielfach:

- Matlab zeichnet sich durch Eleganz, Praktikabilität und Übersichtlichkeit aus. Es wird von etlichen anderen Softwarepaketen sowie als “Pseudocode“ bei der Beschreibung von Rechenverfahren in vielen Büchern benutzt.
- Matlab besitzt grosse Flexibilität: vom gehobenen Tischrechner für Zahlen und Matrizen bis zum Mittel zur Lösung anspruchvoller technischer Aufgaben.
- Es bietet sehr gute On-Line-Hilfeleistung.
- Komplizierte Programme können in einem Bruchteil der Zeit erstellt werden, die unter “klassischen” Programmiersprachen dazu notwendig wäre.
- Matlab ist schnell erlernbar: nach wenigen Tagen kann man schon anspruchsvolle Aufgaben angehen.
- Matlab* beinhaltet starke Graphik, welche zwar in ihren einfachsten Funktionen schnell erlernbar ist, jedoch – unter wesentlich mehr Lernaufwand – schier unbegrenzte Möglichkeiten der Gestaltung bietet.

All diese Eigenschaften machen Matlab* unter Mathematikern als auch unter Praktikern verschiedener Richtungen (Natur-, Wirtschafts-, Ingenieurwissensschaften) beliebt. Dazu kommt Matlabs Erweiterbarkeit durch selbstgebaute oder von Mathworks Inc. zusätzlich bereitgestellte Funktionen. Die letzteren werden als die sog. „Toolboxes“ angeboten. Die numerischen Verfahren, die von Matlab* eingesetzt werden, gehören zum heutigen Standard. Obwohl Matlab* – mit der entsprechenden Toolbox ausgerüstet – auch sog. “symbolische Operationen” wie Maple oder Mathematica erledigen kann, wollen wir unsere Aufmerksamkeit in diesem Kurs ausschließlich dem klassischen “numerischen” Rechnen widmen. Im Unterschied zum symbolischen lässt das numerische Rechnen ständig Rundungsfehler zu, ist aber für “reale” Anwendungen der einzige Weg und wird es nach unserer Meinung noch lange bleiben.

Technische Voraussetzungen. Bei der Bearbeitung dieses Textes soll ein funktionsfähiger Rechner zur Hand sein mit der (wahlweise) installierten Software:

1. Matlab* 6 (MS-Windows (3.1, 95, 98, NT...), DOS, Sun, Linux, Mac)
2. Matlab* 6, Studentenversion (MS-Windows)
3. Matlab*, ältere Versionen
4. Octave* [2]
5. Web-Browser (Netscape oder Explorer)

---

1 Vertrieben durch The Mathworks Inc. Wir werden das Wort “Matlab” sowohl als Sprachbezeichnung als auch für das Softwareerzeugnis benutzen, im letzteren Fall mit einem Stern versehen.

2 Octave ist ein “open source” Produkt der GNU-Gruppe und kann kostenlos installiert werden. Seine Sprache und Funktionen — einschließlich der elementaren Graphik — sind weitgehend an denen von Matlab* orientiert.

In unserem Kurs sind 1. und 2. als Standard zu betrachten. Parallel zum Lesen sollen Sie alle Programme im Text ausprobieren. Dies kann geschehen:

A Im Lokalbetrieb mit Matlab* oder Octave*, installiert auf Ihrem Rechner. Die Berechnungen erfolgen in der jeweiligen interaktiven Umgebung.

B Über das Internet, indem die Seite mit der Web-Version dieses Kurses geöffnet und dort “Octave-on-line” aktiviert wird. Hier ist kein echtes interaktives Rechnen möglich, die von Ihnen erstellten Programmzeilen werden von unserem Rechner verarbeitet, die Ergebnisse werden an Ihrem Fenster sichtbar. Dieses soll eher als Notlösung betrachtet werden.

Aus Rücksicht gegenüber unserer Leserschaft haben wir bewusst mehrere Software-Lösungen berücksichtigt. Dabei wird es mitunter zu Kompatibilitätsproblemen kommen (selbst verschiedene Matlab*-Versionen sind nicht immer miteinander kompatibel). Wir haben versucht diesem auszuweichen, indem wir uns auf grundlegende Programmierungselemente beschränkt haben. Aus dem gleichen Grund haben wir darauf verzichtet, alle verfügbaren Funktionen tabellarisch aufzulisten. Dafür haben Sie das Handbuch sowie die gut organisierte Funktion „help“.

Unsere Schilderung und Funktionsbeschreibung ist knapp gehalten und ist in keinerlei Weise vollständig. Deshalb sollten Sie jede neu gelernte Funktion – über die Beispiele in unserem Text hinaus – selbständig „erforschen“, indem Sie sie verschiedentlich einsetzen und auf die Hilfe von „help“ zurückgreifen. Weitere Hilfe bietet das entsprechende Handbuch – sowohl für Matlab* als auch für Octave*.

Das mathematische Vorwissen umfasst die Grundzüge der Analysis und der Linearen Algebra. Wo „Ausflüge“ in benachbarte Gebiete gemacht werden (z.B. Differentialgleichungen), werden auch kurze Anleitungen dazu gegeben.

1 Zahlen und Operationen

Der einfachste Dienst von Matlab ist der eines Tischrechners. *Die Grundoperationen* sind `+, -, *, /, \, ^,` genauerhin

Operation	Ausdruck	Bedeutung
Addition	`a + b`	$a + b$
Subtraktion	`a - b`	$a - b$
Multiplikation	`a*b`	ab, $a \cdot b$
Linksdivision	`a\b`	$\frac{b}{a}$, b/a
Rechtsdivision	`a/b`	$\frac{a}{b}$, a/b
Potenz	`a^b`	a^b

Beispiele:

```
>> 5*2 - 8
ans =
     2

>> 5^20/17 + 1
ans =
     5.6098e+12

>> 5^(20/17 + 1)
ans =
     33.2114

>> (-2)^(3/4)
ans =
     -1.1892 + 1.1892i

>> (2+3i)/(1+i)
ans =
      2.5000 + 0.5000i
```

Der Name „`ans`“ ist eine Abkürzung von „answer“ (engl. Antwort). Daraus ersieht man:

- Es gilt drei Typen (Bereiche) von Zahlen: *ganze*, *reelle* und *komplexe*.
- Es gelten die üblichen Präzedenzregeln für die Rechenoperationen : 1. Potenzieren 2. Multiplizieren/Dividieren 3. Addieren/Subtrahieren. Gleichberechtigtes wird von links nach rechts ausgeführt. Die Klammern ändern diese Präzedenzen nach Wunsch.

Die Ausgabe wird von der Funktion *format* gesteuert:

```
>> format
>> 100/3
ans =
     33.3333
```

```
>> format short
>> 100/3
ans =
     33.3333

>> format long
>> 100/3
ans =
     33.33333333333334

>> format short e
>> 100/3
ans =
     3.3333e+001

>> format long e
>> 100/3
ans =
     3.333333333333334e+001
```

Das letzte Format entspricht in etwa der rechnerinternen Speicherung dieser Zahl. Diese letztere ist unabhängig vom gewählten Ausgabe–Format. Bei der Eingabe können beliebige Dezimalstellen verwendet werden.

Zu den beschriebenen Zahlen kennt Matlab noch „uneigentliche“ Zahlen `inf` („das Unendliche“) und `NaN` („not a number“), etwa

```
>> 1.2/0
ans =
    Inf
>> 0/0
ans =
    NaN
```

Die *relationalen Operationen* sind

Operation	Ausdruck	Bedeutung
kleiner	`<`	$<$
größer	`>`	$>$
kleiner oder gleich	`<=`	$\leq$
größer oder gleich	`=>`	$\geq$
gleich	`=`	$==$
ungleich	`~=`	$\neq$

Das Ergebnis dieser Operationen ist bei Erfüllung gleich 1, bei Nichterfüllung gleich 0:

```
>> 5 == 2
ans =
      0

>> 5 < 10
ans =
      1
```

```
>> 5 < 10 & 2 > 3
ans =
      0

>> 5 < 10 | 2 > 3
ans =
      1
```

Dabei kamen noch die logischen Operationen `&` *(und)* bzw. `|` *(oder)* mit ihrer natürlichen Bedeutung vor.

Schon beim Tischrechner ist es zweckmäßig, die Zahlen in Variablen zu speichern, etwa

```
>> a1 = 100/3
a1 =
     33.3333

>> b = 2 < 1
b =
    0
```

Wenn wir es – wie früher oben – versäumen, so wird das Ergebnis in die Variable `ans` gesteckt. Einige Variablen werden beim Starten von Matlab gesetzt:

```
  i,j  (imaginaere Einheit)
  pi = 3.141592653589793
  eps = 2.2204460492503134e--016
```

Sie können jedoch bei Wunsch auch anders belegt werden.

Man merke: Mit `=` wird der Wert einer Variablen **gesetzt**, mit `==` wird die Gleichheit **geprüft**.

Man merke auch: solange `i` als die imaginäre Einheit gesetzt ist, sind die Eingaben `5+2i` und `5+2*i` äquivalent; die erstere liefert die gewünschte komplexe Zahl auch dann, wenn die Variable `i` anders besetzt ist.

Neben den Zahlen kennt Matlab auch Zeichenketten oder *strings*, eingegeben werden sie mit Hochkommata

```
>> a = '2+2 sind 4'
a =
    2+2 sind 4
```

Variablennamen enthalten bis zu 13 alphanumerische Zeichen (Groß– und Kleinbuchstaben unterschieden!), jeweils beginnend mit einem Buchstaben. Einige gültige Variablennamen:

```
a, aaA, B12, Bb_1, b12
```

wobei `B12` und `b12` zwei verschiedene Variablen sind. Ungültig sind Namen wie

```
1A c*d a!2 ;
```

der Name

```
abcd1234567890
```

wird als

```
abcd123456789
```

verstanden. Das gleiche gilt für die Funktionennamen. Man soll vermeiden, zwei Objekten den gleichen Namen zu geben. Insbesondere sollten den Variablen keine reservierten Funktionennamen gegeben werden.

2 Elemente der Programmierung

Sobald eine Zeichenfolge geschrieben und die Zeile mit der Enter–Taste beendet wird, versucht Matlab, sie als Befehl zu verstehen, sodann auszuführen. All die Beispiele im Abschnitt 1 waren gültige Matlab–Befehle.

Das Leerzeichen ist obligatorisch, um die Variablennamen und Funktionennamen voneinander zu trennen. Sobald ein Operationszeichen, Komma, Semikolon oder Doppelpunkt verwendet wird, ist das Leerzeichen nicht mehr nötig, kann aber zwecks Lesbarkeit verwendet werden.

Das Semikolon hinter einem Befehl, der den Wert einer Variablen setzt, unterdrückt deren Anzeige:

```
>> a1 = 5.2;
>> b3 = 6;
>> c = a1*b3
c =
    31.2000
```

Dabei ist die Unterdrückung der Anzeige eher der Regelfall, die Anzeige wird meistens beim Endergebnis oder bei bestimmten Zwischenergebnissen zwecks Fehlersuche verwendet.

Eine Zeile kann mehrere Befehle tragen; sie müssen entweder durch *das Komma* (mit Anzeige) oder das Semikolon (ohne Anzeige) getrennt werden.

```
>> a1 = 5;  b2 = 7*a1;
>> c = 5*b2, z = sin(c)*2;
c =
    175
```

Erscheint dagegen die Zeile für einen Befehl zu kurz oder unübersichtlich, so kann sie durch *drei Punkte* fortgesetzt werden:

```
>> a = 5*8 - 2 ...
*8 - 7
a =
    17
```

Erscheint irgendwo das Zeichen % (*das Kommentarzeichen*), so überspringt Matlab den darauffolgenden Text bis zum Zeilenende.

Die üblichen *Elementarfunktionen* sind

Bezeichnung	Bedeutung
`sin(a)`	$\sin a$
`cos(a)`	$\cos a$
`tan(a)`	$\tan a$
`cot(a)`	$\cot a$
`asin(a)`	$\arcsin a$
usw.	
`abs(a)`	$\|a\|$ (auch komplex)
`exp(a)`	e^a
`log(a)`	$\ln a$
`sinh(a)`	$\sinh a$
usw.	
`conj(a)`	$\bar{a}$
`real(a)`	Re a (Realteil)
`imag(a)`	Im a (Imaginärteil)
`sqrt(a)`	$\sqrt{a}$
`round(a)`	das nächste Ganze
`rand`	Zufallszahl zwischen 0 und 1

Eine vollständige Auflistung gibt der Befehl

```
>> help elfun
```

Die Funktion `help` gibt schnelle Information über alle Matlab–Funktionen und deren Syntax. Der Befehl

```
help
```

bringt eine Übersicht über alle Funktionengruppen, dagegen liefert

```
help xxx
```

Auskunft über die Funktion (oder die Funktionengruppe) `xxx`, etwa `help general`, `help sin`, `help lookfor`, `help help` usw. Bei Octave ist die `help`–Funktion ähnlich organisiert.

2.1 Beispiel. Wir berechnen die Gegenkathete `b` des rechteckigen Dreiecks mit der Hypothenuse `c = 2cm` und den Winkel $\varphi = 35^0$
Es ist $b = c \sin \varphi$ (φ im Bogenmaß). Programm:

```
>> c = 2;
>> phi = 35*pi/180;
>> c*sin(phi)
ans =
       1.1472
```

Erstellen eines Programmes. Die Befehle können im Lokalbetrieb interaktiv, d.h. Zeile für Zeile mit der Enter–Taste abgeschickt werden oder aber sämtlich in eine Datei mit einem Namen *.m *(m–Datei)* aufgeschrieben werden. Das so entstandene Programm kann durch das Eintippen des Dateinamens (die Verlängerung „.m“ wird dabei weggelassen) und der Enter–Taste ausgeführt werden. Ein Beispiel:

```
% Berechnung der Winkel alpha, beta (im Gradmass)
% aus den Katheten a,b eines rechtw. Dreiecks
a = input('Tippen Sie a ein! ');
b = input('Tippen Sie b ein! ');
% die Hypothenuse;
c = sqrt (a^2 + b^2);
% Die Winkel im Bogenmass
alpha_Bog = acos(a/c);
beta_Bog = acos(b/c);
% Die Winkel im Gradmass
alpha = 180 * alpha_Bog/pi;
beta = 180*beta_Bog/pi;
```

Ist dieser Text in die Datei `winkel.m` im aktuellen Verzeichnis gespeichert, so kann die Programmausführung so aussehen:

```
>> winkel
Tippen Sie a ein! 3
Tippen Sie b ein! 4
```

Die Variablen `a,b` können bei Wunsch durchs Eintippen gezeigt werden).

Hier kommt die Funktion `input` vor, der Text zwischen den Hochkomma–Zeichen fordert zur Eingabe auf, diese wird jeweils durch die Enter–Taste abgeschickt.

Die Erfahrung lehrt: besser mehr Kommentartext als weniger! Der erste zusammenhängende Kommentarblock wird mit dem Befehl `help winkel` gezeigt, dort soll knappe Information über das Programm enthalten sein.

Die meisten Matlab-Umgebungen enthalten einen Editor, wo solche Dateien erstellt werden können, die Speicherung erfolgt in der Regel in das aktuelle Verzeichnis. Sonst können Sie mit

`!xxx`

den Editor Ihrer Wahl aufrufen; beim Verlassen kehren Sie in die alte Matlab– (oder Octave–) Umgebung zurück. Besteht die Möglichkeit, mehrere Fenster zu öffnen (Unix, Linux), so können Sie das Editieren ohnehin in einem anderen Fenster erledigen.

Wenn Sie Octave–on–line benutzen, so ist aus technischen Gründen das Anlegen von Dateien auf dem Server ausgeschlossen. Sie schreiben einfach das ganze Programm in das Fenster und führen es durch das Klicken in die Fläche „Ergebnis“ aus.

Die eben beschriebene Art eines Programms nennt sich ein „Skript“. Natürlich kann ein Skript von einem anderen aufgerufen werden. Dabei wächst die Menge der Variablen schnell ins Unübersichtliche. Abhilfe bietet eine andere Art von Programm, die *m–Funktion.*

Unser Programm `winkel` könnte als Funktion so aussehen

```
function [alpha,beta] = fwinkel(a,b)
% function [alpha,beta]=fwinkel(a,b) berechnet
% die Winkel alpha,beta (in Gradmass) aus den
% Katheten a,b eines rechtw. Dreiecks
% Die Hypothenuse
```

```
c = sqrt(a^2 + b^2);
% Die Winkel in Bogenmass
alpha_Bog = acos(a/c);
beta_Bog = acos(b/c);
% Die Winkel in Graden
alpha = alpha_Bog*180/pi;
beta = beta_Bog*180/pi ;
```

Wir vergleichen die beiden Vorgehensweisen:

```
>> clear
>> winkel
    Tippen Sie  a ein! 3
    Tippen Sie  b ein!  4
>> who
Your variables are:
a alpha_Bog  b  beta_Bog
alpha   ans  beta c
>> clear
>> [alph,bet] = fwinkel(3,4);
>> who
Your variables are
alph   bet
```

Wir kommentieren das Vorgehen und die Ergebnisse. Die Funktion `clear` löscht alle Variablen, die Funktion `who` listet alle vorhandenen Variablen auf. Bei `fwinkel` sind alle Zwischenvariablen gelöscht, es bleiben nur diejenigen, die wir als *Ausgabeparameter* in die eckige Klammer gesetzt haben.

Beim Aufruf haben wir diese Variablen nicht `alpha,beta` genannt (was hier ja sinnvoller wäre), sondern anders, um zu zeigen, daß die Namensgebung frei ist, die Ausgabeparameter werden nach *der Reihenfolge* in den eckigen Klammern identifiziert. Entsprechendes gilt auch für die *Eingabeparameter* in den runden Klammern. Ein kleiner Unterschied: die Ausgabeparameter müssen Variablen sein, während die Eingabeparameter sowohl Variablen als auch konkrete Ausdrücke sein können (wie bei unseren Ausführungen oben). Wenn es nur einen Ausgabeparameter gibt, ist die eckige Klammer optional. Es kann natürlich auch Funktionen mit variabler Parameterzahl oder auch ohne jegliche Parameter geben.

Im on-line Betrieb mit Octave sind die Funktionen `clear` und `who` bedeutungslos und eine selbstprogrammierte Funktion muss selbst im Fensterprogramm samt des Ausrufes vorhanden sein:

```
1;
function [alpha,beta] = fwinkel(a,b)
c = sqrt(a^2+b^2);
alpha_Bog = acos(a/c);
beta_Bog = acos(b/c);
alpha = alpha_Bog*180/pi;
beta = beta_Bog*180/pi;
end
[alph,bet] = fwinkel(3,4)
```

Man merkt: (i) es ist eine an sich bedeutungslose erste Zeile vorhanden, sie verhindert lediglich, dass das ganze Programm als eine Funktion verstanden wird, (ii) am Ende des Funktionstextes kommt

noch ein `end` hinzu; dies ist auch dann notwendig, wenn Funktionen als „Unterprogramme“ in einem Skript untergebracht werden. Diese Möglichkeit bieten auch die neueste Version von Matlab, dort sind die Funktionen “geschachtelt”, etwa als `fwink.m`:

```
function [alpha,beta]=fwink(a,b)
[alph,bet] = fwinkel(a,b)
function [alpha,beta] = fwinkel(a,b)
c = sqrt(a^2+b^2);
alpha_Bog = acos(a/c);
beta_Bog = acos(b/c);
alpha = alpha_Bog*180/pi;
beta = beta_Bog*180/pi;
end
```

Bei den älteren Versionen muß jede m-Funktion in einer besonderen Datei abgelegt werden. Funktionen sind bei mehrfachem Gebrauch die richtige Programmart. Allerdings ist es bei der Erstellung einer Funktion oft zweckmäßig, erst ein m–Skript zu schreiben, weil bei evtl. Fehlersuche die Zwischenvariablen nützlich sind. Wenn das Programm erst einmal läuft, fügt man einfach den „Funktionskopf“ hinzu.

Bedingte Befehle. Wir möchten die quadratische Gleichung

$$ax^2 + bx + c = 0$$

lösen, aber nur, wenn sie reelle Lösungen hat. Ein entsprechendes Programm, `quadr.m` genannt, könnte so aussehen:

```
a = rand; b = rand; c = rand;
diskr = b*b - 4*a*c;
if diskr >= 0
   lambda1 = (-b + sqrt(disk))/2/a;
   lambda2 = (-b - sqrt (diskr))/2/a;
end
```

Die Formeln für die Lösungen werden genau dann ausgeführt, wenn die Quadratwurzel reell ausfallen muss. Der `end` - Befehl schließt die Abfolge aller durch das `if` bedingten Befehle ab.

Führen Sie das Programm `quadr.m` mehrmals aus, um beide möglichen Ergebnisse herbeizuführen.

Sie können das Programm auch interaktiv ausführen; nach der `if` - Zeile zeigt das Fehlen des Promptzeichens `>>`, dass Matlab auf das abschließende `end` wartet.

Als Bedingung darf neben `if` irgendeine Zahl stehen – typischerweise wird es eine Relation mit dem Wert 1 (für „wahr“) oder 0 (für „falsch“) sein. (Das `if` reagiert eigentlich auf die Tatsache, ob die Zahl gleich oder ungleich Null ist.)

Wenn sowohl bei Erfüllung als auch bei der Nichterfüllung gewisse Aktionen gemacht werden sollen, dann benutzt man den Befehl `else`. Das Programm `quadr1.m`

```
a = rand; b = rand; c = rand;
diskr = b*b - 4*a*c;
```

```
if diskr >= 0
   lambda_1 = (-b + sqrt(diskr))/2/a;
    lambda_2 = (-b - sqrt(diskr))/2/a;
else
    error('keine reellen Loesungen!')
end % if diskr
```

ist ausführlicher – bei Nichterfüllung der Bedingung hält die Funktion `error` das Programm an und gibt die beigefügte Meldung.

Mit `while` wird ein Block von Befehlen wiederholt, solange die Bedingung erfüllt ist. Betrachten wir das Programm `epsil.m`

```
function z = epsil(a);
if a <= 0
   error(' Ihre Zahl war nicht positiv')
end % if a
while 1 + a/2 > 1
    a = a/2;
end % while 1
z=a ;
```

Die gegebene Zahl a wird solange halbiert, wie die Operation $1 + a/2$ etwas größeres als 1 ergibt.

2.2. Aufgabe. Führen Sie `epsil.m` mit verschiedenen Zahlen a durch und vergleichen Sie das Ergebnis mit der vorhandenen Variable `eps`.

Wir fertigen nun ein etwas umfangreicheres Programm `cbisect.m`, das die Nullstelle der Funktion

$$\cos x - x \ , \quad 0 < x < \pi/2$$

mittels Intervallhalbierung finden soll.

```
function c = cbisect(a,b,tol)
% function x0 = bisect(fn,a,b,tol)
% berechnet die Nullstelle z der Funktion cos(x) - x
% im Intervall a < x < b mit der
% Genauigkeit tol.
if sign(cos(a) - a) == sign(cos(b) - b)
   error('Kein Anzeichen fuer eine Nullstelle! ');
end % if sign
while b - a > tol
    c = (b + a)/2;
    if cos(c) - c == 0
       a = c; b = c;
    else
       if sign(cos(a) - a) ~= sign(cos(c) - c)
          b = c;
       else
          a = c;
       end % if sign
```

```
    end % if fn
end % while b - a
end
```

Kommentar: Der erste `if` – Block ist eine Vorsichtsmaßname – Intervallhalbierung hat wenig Sinn, wenn die Funktion an beiden Intervallenden das gleiche Vorzeichen hat. Der `while` – Befehl läßt die Halbierung solange arbeiten, bis die gewünschte Grenze `tol` erreicht ist. Das nächste `if` überprüft, ob das gewonnene `c` schon Nullstelle ist, wenn ja, dann wird mit `a = c; b = c`; dem `while` mitgeteilt, es soll aufhören. Sonst `(else)` wird überprüft, auf welcher Intervallhälfte die Funktion ihr Vorzeichen ändert; mit dem wird weitergearbeitet. Kurze Kommentare nach den `end` – Befehlen sowie passende Zeileneinzüge verbessern die Lesbarkeit.

2.3 Aufgabe. Führen Sie `c = cbisect(0,pi/2,tol)` aus mit verschiedenen Werten für `tol` und überprüfen Sie das Ergebnis mit `cos(c) - c`.

Der Nachteil von `cbisect` ist, daß es nur die Funktion $\cos(x) - x$ behandelt. Wünschenswert wäre es, ein Intervallhalbierungsprogramm zu haben, das beliebige Funktionen behandelt. Das Programm `bisect.m` tut dies.

```
function c = bisect(fn,a,b,tol)
% function x0 = bisect(fn,a,b,tol)
% berechnet die Nullstelle z der Funktion, deren Namen das
% string fn beinhaltet, im Intervall a < x < b mit der
% Genauigkeit tol.
if sign(feval(fn,a)) == sign(feval(fn,b))
   error('Kein Anzeichen fuer eine Nullstelle! ');
end % if sign
while b - a > tol
    c = (b + a)/2;
    if feval(fn,c) == 0
       a = c; b = c;
    else
       if sign(feval(fn,a)) ~= sign(feval(fn,c))
          b = c;
       else
          a = c;
       end % if sign
    end % if fn
end % while b - a
end
```

Das Neue hier ist der Funktionsaufruf

```
feval(fn,a)
```

womit die Funktion mit dem Namen `fn` an der Stelle `a` berechnet wird. Somit wird eben die gewünschte Flexibilität erreicht, die beliebige Funktionen bearbeiten läßt – eingebaute (wie sin, cos, exp, usw) oder selbstgebaute.

Damit dies das Gleiche bewirken kann, wie `cbisect`, soll noch eine Funktion `cosxx.m` angelegt werden:

```
function y=cosxx(x)
% function y=cosxx(x) berechnet cos(x)-x
y=cos(x)-x;
```

Nun bewirkt

```
>> c = bisect('cosxx',0,pi/2,1e-10);
```

das gleiche wie

```
>> c=cbisect(0,pi/2,1e-10);
```

2.4 Aufgabe. Mit `bisect` finde man Nullstellen von $\sin x$, $\cos x$, $x^5 - 10x^2 + 1$ auf passenden Intervallen. Man vergleiche die Ergebnisse mit denjenigen, die mit der eingebauten Funktion `fzero` erzielt werden (erst `help fzero` eintippen!).

Der `for` - Befehl wiederholt eine Abfolge von Befehlen in einer Schleife, so berechnet

```
for k = 1:3
    a = k^2, b = k^3
end
```

alle Quadrate und Kuben der ganzen Zahlen von 1 bis 3. Das gleiche in absteigender Reihenfolge

```
for k = 3:-1:1
    a = k^2, b = k^3
end
```

Mit `for k=1:0.5:3` werden alle Zahlen zwischen 1 und 3 mit Abstand 0.5 abgezählt. Auch `for` muss stets mit `end` abgeschlossen werden.

2.5 Aufgabe. Finden Sie die Nullstelle von $\cos(x) - x$, indem Sie die Funktionen `bisect` und `cosxx` in ein Skript eintragen (dies kann dann auch mit Octave-on-line ausgeführt werden).

3 Zeilenvektoren

Die Ausdrücke vom Typ `x:y:z`, die wir beim `for`–Befehl kennengelernt haben, besitzen auch selbständige Bedeutung:

```
>> 1:5
ans =
    1 2 3 4 5
>> 1:-0.3:0
ans =
    1.0000 0.7000 0.4000 0.1000
```

Das sind Beispiele von *Zeilen* oder *Zeilenvektoren*. Elementweise wird eine Zeile in eckigen Klammern eingegeben:

```
>> a = [77.1 -pi exp(1)];
```

Einzelne Komponente werden als `a(k)` gewonnen:

```
>> t = a(1)
t =
    77.1
```

Aber auch Teile eines Zeilenvektors können herausgenommen werden:

```
>> a(2:3)
ans =
      -3.1415 2.7183
```

oder noch allgemeiner:

```
>> p = [3 2 1]
>> a(p)
ans =
      2.7183  -3.1416  77.1000
```

Ähnlicherweise können einzelne Elemente oder Teile eines Zeilenvektors neu gesetzt werden:

```
>> a(3) = 1/3
a =
    77.1000  -3.1416  0.3333
>> a(1:2) = [1 1]
a =
    1.000 1.000 0.3333
>> a(3:4) = []
a =
    1.0000   1.0000
```

```
>> a(5) = 5
a =
    1.0000 1.0000 0 0 5.0000
```

Der Ausdruck [] (leere Matrix) diente hier dazu, die Teile des Zeilenvektors zu entfernen. Man merke auch, wie durch die Anweisung `a(5) = 5` sinnvollerweise der ganze Zeilenvektor „redimensioniert“ wurde, indem die Zwischenkomponenten als Nullen gesetzt wurden.

Das Arbeiten mit Zeilen zeichnet sich, wie Sie sehen, durch große Flexibilität aus. Alle elementaren Funktionen aus Abschnitt 2 können auch an Zeilenvektoren angewandt werden mit komponentenweise Wirkung, z.B.

```
sin(1:3)
ans =
     0.8415  0.9093  0.1411
```

Einige nützliche Zeilenvektoren erhält man mit

```
zeros(1,n)       (n Nullen)
ones (1,n)       (n Einsen)
rand(1,n)        (n Zufallszahlen zwischen 0 und 1)
```

Im gleichen Sinne erstrecken sich alle Grundoperationen auf die Zeilenvektoren, wobei die beiden Zeilen gleich lang sein müssen, und dem Operationszeichen wird ein Punkt vorgesetzt (bei `+,-` und den relationalen Operationen ist dieser Punkt optional):

```
>> [2 1] + [3 4]
ans =
      5  5
>> [2 1].*[3 4]
ans =
      6  4
>> [2 1]./[3 4]
ans =
      0.6667   0.2500
>> [2 1].^[3 4]
ans =
      8  1
```

Die gleiche Länge muß nicht sein, wenn ein Operand nur ein Element hat (dies ist dann gleich einer Zahl, auch *Skalar* genannt):

```
>> [2 1 + 1i] + 3
ans =
      5.0000  4.0000 + 1.0000i
>> 4./[2 1]
ans =
      2  4
>> [5 -1 2 1] >= 0
```

```
ans =
      1 0 1 1
>> [2 1] == [2 3]
ans =
      1 0
```

Zeilenvergleiche können auch bei `if`– oder `while`– Befehlen verwendet werden, die Bedingung gilt als erfüllt, wenn **alle** Komponenten wahr (d.h. von Null verschieden) sind! Wollen wir also die Bedingung stellen, die Vektoren a, b sollen nicht gleich sein, so ist der Befehl

```
if a ~= b
```

fehl am Platze; wir sollen stattdessen

```
if any(a ~= b)
```

verwenden. Einige weitere Vektorfunktionen sind

Funktion	Bedeutung
`sum`	Summe aller Komponenten
`prod`	Produkt aller Komponenten
`max`	Maximale Komponente
	(falls komplex, maximaler Realteil)
`min`	Minimale Komponente
	(falls komplex, minimaler Realteil)
`norm`	Euklidische Norm
`sort`	sortiert wachsend
`any`	0, wenn alle Komponenten
	verschwinden, sonst 1
`length`	Länge der Zeile
`mean`	Mittelwert

Beispiele:

```
>> a = [0 -1 1];
>> min(a)
ans =
       -1
>> [z,h] = min(a)
z =
       -1
h =
    2
```

Wir sehen, daß die Funktion `min` zwei Ausgabeparameter haben kann, der erste ergibt den Minimumwert, der zweite die Minimumstelle. Ähnlich ist es mit `max` und `sort`; probieren Sie es selbst

aus!

3.1. Aufgabe. Schreiben Sie einen Einzeiler, der einen gegebenen Zeilenvektor absteigend sortiert!

Die Zeilenvektoren sind sehr geeignet, um Funktionen einer Variablen zu tabellieren oder graphisch darzustellen. Nehmen wir die Parabel

$$y = x^2 - 2x + 1 \ , \quad 0 \leq x \leq 2$$

und die Sinusfunktion

$$y = \sin x \ , \quad 0 \leq x \leq 2 \ .$$

Mit der Wahl `x = 0:0.5:2` haben wir

```
x = 0:0.5:2;
y = x.^2 - 2*x + 1;
y1 = sin(x);
```

Mit `plot(x,y)` wird die erste, mit `plot(x,y1)` die zweite Funktion graphisch dargestellt. Mit

```
plot(x,y,x,y1)
```

erscheinen beide Funktion auf einem Bild. Wir sehen, daß die Graphik die gewonnenen Punkte mit geraden Stücken verbindet. Die Verfeinerung der „Knotenverteilung“, etwa `x = 0:0.1:2`, bringt ein besseres Bild. (Abb.3.1.)

(Abb. 3.1)

Mit `plot(x,y,'O')`, `plot(x,y,'+',x,y1,'.')` lernen Sie mehr über die vielseitige Funktion `plot` (`help plot` ist sehr hilfreich). Die wirklichen Möglichkeiten der Matlab*–Graphik gehen viel weiter.

Ein gut gelungenes Bild verlangt oft etwas Mühe ab. Als Beispiel sei

$$y \ = \ \frac{1}{x^2 + x - 1}, \quad 0 \leq x \leq 1$$

graphisch darzustellen:

```
h = input('Schrittweite h = ');
x = 0:h:1;
y = 1./(x.^2 + x - 1);
plot(x,y)
```

(Abb. 3.2)

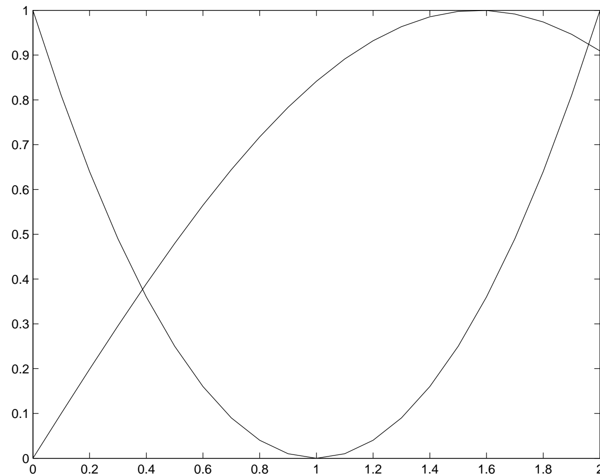


Abbildung 3.0.1

Bei $h = 0.03$ wäre das Bild (Abb.3.2) ganz brauchbar, wenn nicht die schräge Verbindung „von $-\infty$ bis $+\infty$" da wäre. Die Wahl 0.01 oder 0.001 macht die Sache weniger schräg, dafür aber den Funktionsverlauf weniger deutlich, weil auch sehr große Funktionswerte erscheinen müssen (unsere Funktion ist ja singulär bei $x = (\sqrt{5} - 1)/2 \approx 0.6180$).

Eine saubere Lösung ist es, die beiden Zweige der Funktion getrennt darzustellen:

```
h = input('Schrittweite h = ');
x = 0:h:1;
y = 1./(x.^2 + x - 1);
[ymin, imin] = min(y);
[ymax, imax] = max (y);
xminus = x(1:imin);
xplus = x(imax : length(x));
yminus = y(1:imin);
yplus = y(imax : length(y));
plot(xplus,yplus,xminus,yminus);
```

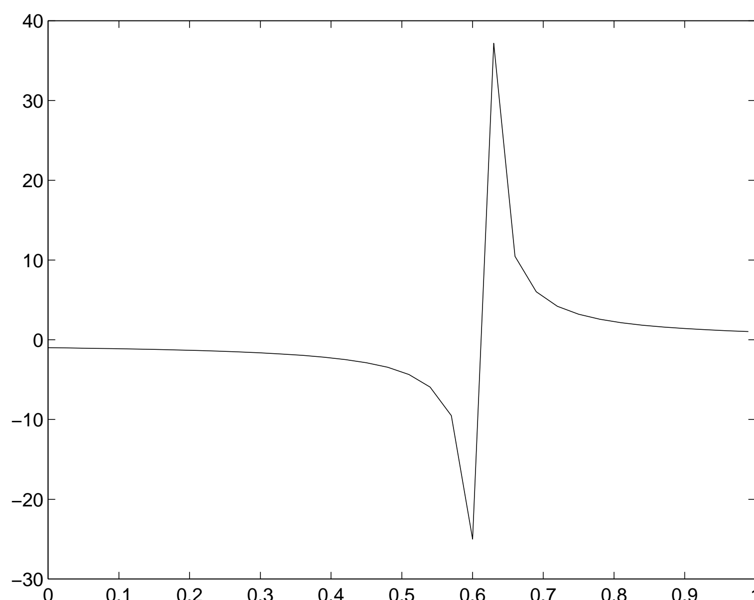


Abbildung 3.0.2

Hier entsteht mit $h = 0.03$ ein recht schönes Bild; man merke den sinnvollen Einsatz der Funktionen `min`, `max`, um die „Sprungstelle“ der Funktion zu bestimmen.

3.2. Beispiel. Wir tabellieren die Funktion

$$y = \begin{cases} \cos x & , \quad 0 \le x \le \pi/2 \\ -x + \pi/2 & , \quad \pi/2 < x \le \pi \end{cases}$$

und stellen sie graphisch dar.

Hier bietet sich die Verwendung von relationalen Operationen an:

```
h = input('Schrittweite h = ');
x = 0:h:pi;
y = cos(x).*(x <= pi/2) + (-x + pi/2).*(x > pi/2);
plot(x,y)
```

Dies ist selbsterklärend, man lasse sich dazu nochmals die Zeilen `x >= pi/2` und `x > pi/2` bei `h = 0.1` zeigen.

Polynome und Interpolation. Ein Polynom $n-$ten Grades

$$y = p(x) \; = a_1 x^n + a_2 x^{n-1} + \ldots + a_n x + a_{n+1}$$

kann bei bekanntem Zeilenvektor der Koeffizienten $[a_1\, a_2 \;\ldots\; a_n\, a_{n+1}]$ und bekannter Zahl x berechnet werden als

```
y = 0; n = length(a) - 1;
for k = 1:n + 1
    y = y + a(k)*x.^(n - k + 1);
end
```

Beim Tabellieren wird x ein Zeilenvektor:

```
n = length(a) - 1;
nx = length(x);
y = zeros(1,nx);
for  k = 1:n + 1
     y=y + a(k)*x.^(n - k + 1);
end
```

Matlab hat dazu eine eingebaute Funktion `polyval` mit der gleichen Wirkung.

3.3. Aufgabe. Vergleichen Sie `polyval` mit unserem obigen Programm an einigen einfachen Beispielen.

Wie man weiß, gibt es bei den Daten

$$\begin{array}{c} x_1 \; < \; x_2 < \quad \cdots \quad < x_n \; < \; x_{n+1} \\ y_1, \; y_2, \quad \cdots \quad , \; y_n, \; y_{n+1} \end{array}$$

stets genau ein Polynom $p(x)$ vom Grad $\leq n$, so daß gilt

$$p(x_i) = y_i \;, \quad i = 1, \ldots, n+1 \;.$$

Wir sagen, das Polynom p *interpoliert* die Daten x_i, y_i . Für die Koeffizienten des interpolierenden Polynoms existieren verschiedene Formeln (Lagrange–, Newton-Formeln usw.). Matlab hat dazu eine Funktion `polyfit`. Die Koeffizienten $a_1, \ldots, a_{n+1}$ werden als Vektorzeile aus den Datenvektoren x, y durch

```
a = polyfit(x,y, length(x)-1)
```

gewonnen.

3.4. Beispiel. Wir interpolieren die Daten

x	1	2	3	4
y	0	1	0	2

```
>> xd = 1:4;
>> yd = [0 1 0 2];
>> x = 1:0.1:4 ;
>> y = polyval(polyfit(xd,yd,length(xd) - 1),x);
>> plot(x,y,xd,yd,'o')
```

(Abb. 3.3)

3.5. Aufgabe. Versuchen Sie, das obige Programm auszuführen, indem Sie den Ausdruck `length(xd - 1)` (sein Wert ist dort ja gleich 3) durch 2 oder 4 ersetzen. Benutzen Sie die `help`-Funktion, um die Ergebnisse zu interpretieren!

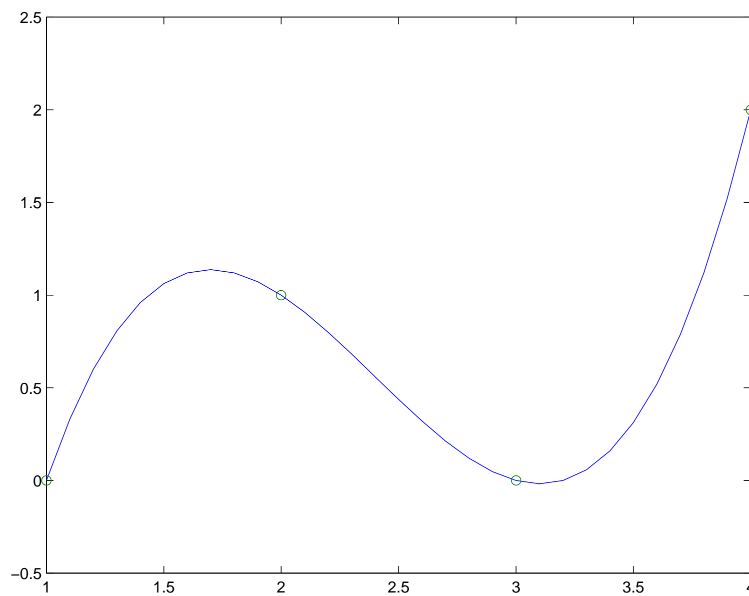


Abbildung 3.0.3

Nullstellensuche. Alle Nullstellen einer stetigen Funktion zu finden, ist nicht immer leicht: das einzige Suchkriterium ist der Vorzeichenwechsel. Es gibt aber Fälle, wo kein Vorzeichenwechsel vorhanden ist (etwa bei $y = x^2$) oder solche, wo das Vorzeichenwechsel auf einem sehr kleinen Intervall stattfindet, das deshalb schwer zu finden ist. Ein Beispiel:

$$y = \frac{\sin x}{x} - 0.1285, \;\; x > 0$$

Tabellarische Untersuchung

```
x = 0.01:0.01:15;
y = sin(x)./x - 0.1285;
plot(x,y)
```

(Abb. 3.4)

ergibt Abb. 3.4. Die eine Nullstelle ist klar sichtbar, die zweite fraglich. Die Vorzeichenuntersuchung

```
x = 0.01:0.01:15;
y = sin(x)./x - 0.1285;
n = length(x);
a = zeros(n);
for k = 1:n - 1
    if sign(y(k)) ~= sign(y(k+1))
                      a(k) = 1;
end % if sign
end % for k
find(a)
```

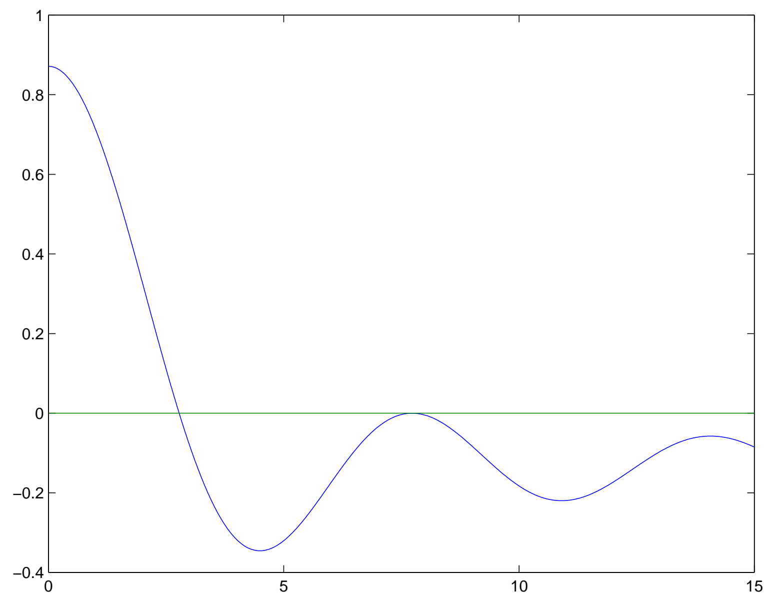


Abbildung 3.0.4

ergibt, wenn ausgeführt,

```
ans =
   277
```

d.h. der Vektor `a` ist nur an der Stelle 277 von Null verschieden. Somit befindet sich die sichere Nullstelle zwischen `x(277)` und `x(278)`. Die Funktion `find` gibt die Stellen (die Indexwerte), wo der gegebene Vektor von null verschieden ist.

3.6. Aufgabe. Schreiben Sie das obige Programm in eine `m`-Datei, führen Sie es aus und finden Sie anschließend die Nullstelle

– mit unserer Funktion `bisect`

– mit der Matlab-Funktion `fzero`

Vergleichen Sie die Ergebnisse.

4 Matrizen

Eine der Hauptstärken von Matlab ist sein sehr flexibler und übersichtlicher Umgang mit Matrizen. Eigentlich ist die Matrix der Grundtyp der Matlab–Variablen. Eine Matrix wird elementweise eingegeben als

```
>> a = [1 2 3; 4 5 6/2]
a =
     1  2  3
     4  5  3
```

oder auch

```
>> a = [1,2,3;4,5,6/2]
a =
     1  2  3
     4  5  3
```

Die Elemente in einer Zeile werden durch das Komma oder das Leerzeichen getrennt, die Zeile endet mit Semikolon. Die Dimensionen von `a` (man sagt auch *den Typ*) erfragt man mit

```
>> size(a)
ans =
      2  3
```

(d.h. 2 Zeilen und 3 Spalten). Die Matrizen mit nur einer Zeile und Spalte sind den entsprechenden Zahlen gleichgesetzt (auch *Skalare* genannt):

```
>> [2]
ans =
      2
```

Die im vorigen Abschnitt vorgestellten Zeilenvektoren sind Matrizen mit einer Zeile.
Die einzelnen Elemente sind als `a(i,j)` etwa

```
>> c = a(1,3)
c =
     3
```

gegeben, können genauso neu gesetzt werden

```
a =
    1  2  3
    4  5  3
>> a(3,3) = -1
a =
    1  2  3
    4  5  3
    0  0 -1
```

(mit dem üblichen Auffüllen durch Nullen).
Die Funktionen **ones**, **zeros** und **rand** ermöglichen schnelles Generieren von Matrizen mit gewünschten Eigenschaften.

```
>> ones(2,3)
ans =
      1  1  1
      1  1  1
>> zeros(2)
ans =
      0  0
      0  0
>> rand(3,1)
ans =
      0.7582
      0.4895
      0.0011
```

Untermatrizen werden mittels passender ganzzahliger Vektoren definiert:

```
>> a =[1 2 3; 4 5 6; 7 8 9];
>>z = a(1:2,2:3)
z =
    2  3
    5  6
>> a(1:2,:)
ans =
      1  2  3
      4  5  6
>> p = [3  1  2];
>> a(p,p)
ans =
      9  7  8
      3  1  2
      6  4  5
>> a(:,3)
ans =
      3
      6
      9
```

Matrizen mit nur einer Spalte sind die *Spaltenvektoren*. Aus bestehenden Matrizen lassen sich schnell neue Matrizen bilden, z.B.

```
>> p = [3 1 2];
>> a = [1 2 3;4 5 6;7 8 9];
>> a = [p;a]
```

```
a =
     3  1  2
     1  2  3
     4  5  6
     7  8  9
>> [a a(:,3)]
ans
    3 1 2 2
    1 2 3 3
    4 5 6 6
    7 8 9 9
>> a(1:2,1:3) = ones(2,3)
a =
    1 1 1
    1 1 1
    4 5 6
    7 8 9
>> a(:,2:3) = [ ]
a =
    1
    1
    4
    7
```

Rechenoperationen mit Matrizen. Die Symbole `+,-,*,` bewirken die üblichen Matrix–Operationen:

```
>> a = [1 2 3; 4 5 6], b = [1 2; 3 4; 5 6];
a =
    1 2 3
    4 5 6
b =
    1 2
    3 4
    5 6
>> c = a*b
c =
    22  28
    49  64
>> b*a
ans =
      9 12 15
     19 26 33
     25 40 51
```

Die Dimensionen müssen passend sein, es sei denn, ein Operand ist Skalar; in diesem Fall wird die Operation an jedem Matrixelement ausgeführt:

```
>> a+2
```

```
ans =
      3  4  5
      6  7  8
>> z = 2*a
z =
    2  4  6
    8 10 12
>> z/2
ans =
     1  2  3
     4  5  6
```

während `2/a` so nicht möglich ist. Die Funktion `inv` liefert die Inverse:

```
>> c = [1 1; 0 1]
c =
     1  1
     0  1
>> z = inv(c)
z =
     1  -1
     0   1
>> z*c
ans =
      1  0
      0  1
```

Besonders mächtig sind die Matrix–Operationen `\` und `/`. Bis auf die Rundungsfehler bewirkt `A\B` dasselbe wie `inv(A)*B` und `A/B` dasselbe wie `A*inv(B)`, jedoch ist der Algorithmus bei `\` und `/` schneller [1] (oft auch genauer) und deshalb absolut vorzuziehen:

```
>> a = rand(200); b = rand(200,1);
>> tic, x = a\b; toc
>> tic, xx = inv(a)*b; toc
>> norm(xx - x)
>> norm(a*x - b), norm(a*xx - b)
```

Die Funktionen `tic`, `toc` imitieren eine Stoppuhr.

Multiplizieren, Dividieren und Potenzieren kann man auch punktweise:

```
>> a = [1 2; 6 4], b = [-1 -1; 2 2]
a =
    1  2
    6  4
b =
    -1  -1
```

1 Die Operation `X=A\B` löst unmittelbar das lineare Gleichungssystem $AX = B$.

```
     2    2
>> a.*b
ans =
      -1   -2
       12   8
>> a./b
ans =
      -1   -2
       3    2
>> a.^b
ans =
      1.0000   0.5000
     36.0000  16.0000
```

während die Operationen `.+`, `.-` gleichbedeutend sind mit `+,-`.

Die Matrix–Potenz ist etwas delikater. Typischerweise ist bei `a^b` `a` eine quadratische Matrix und `b` ein Skalar:

```
>> a = [2 -1; -1 2]
a =
    2   -1
   -1    2
>> c = a^(1/2)
c =
    1.3660  -0.3660
   -0.3660   1.3660
>> c*c
ans =
     2   -1
    -1    2
```

Die Matrix-Potenz kann auch bei reellen Matrizen komplex ausfallen.

Die Elementarfunktionen aus Abschnitt 2 werden an Matrizen stets elementweise ausgeführt:

```
>> exp([1 0 1; 0 3 -1])
ans =
      2.7183   1.0000   2.7183
      1.0000  20.0855   0.3679
```

usw. Die Funktionen wie `max,min,sort` usw. haben bei Matrizen besondere Eigenschaften, die Sie selber ausprobieren können.
Die Matrix wird *transponiert* mit der Funktion `.'`:

```
>> (1:3).'
ans =
      1
      2
      3
>> [i 1].'
```

```
ans =
      0 + 1.0000i
      1.0000
```

Die Funktion ' (adjungiert) bewirkt dazu noch das komplexe Konjugieren:

```
>> [i 1]'
ans =
       0 -1.0000i
       1.0000
```

(an reellen Matrizen stimmen ' und .' überein). Das Skalarprodukt von zwei Spaltenvektoren `x,y` wird als `x'*y` berechnet:

```
>> x = [1;2], y = [3;4]
x =
    1
    2
y =
    3
    4
>> x'*y
ans =
     11
>> x = [i;1];
>> x'*x
ans =
      2
```

so dass der Ausdruck `sqrt(x'*x)` bei einem Zeilenvektor gleichwertig ist mit `norm(x)`. Die Operation ' ist wichtiger und kommt öfter vor als `.'`, daher ist ihr Zeichen einfacher.[2)]

Um den Optimierungsgrad von Matrix-Operationen zu untersuchen, eine kleine Testserie: Wir setzen

```
>> n = 100;
>> a = rand(n) ; b = rand(n);
```

und berechnen das Produkt `a*b` auf fünf Weisen. Erstens

```
for i1 = 1:n
for j1 = 1:n
    c(i1,j1) = 0;
    for k1 = 1:n
        c(i1,j1) = c(i1,j1) + a(i1,k1)*b(k1,j1);
    end
end
end
```

Zweitens: Das Programm wie oben mit der vorangestellten Zeile

2 Die übliche Bezeichnung für die adjungierte Matrix ist A^*, für die transponierte A^T.

```
c = zeros(n);
```

Drittens:

```
c = zeros(n) ;
for i1 = 1:n
for j1 = 1:n
    c(i1,j1) = a(i1,:)*b(:,j1);
end
end
```

Viertens:

```
for j1 = 1:n
    c(:,j1) = a*b(:,j1);
end
```

Fünftens:

```
c = a*b;
```

Vergleichen Sie die Ergebnisse dieser fünf Programme, insbesondere die benötigten Zeiten. Benutzen Sie z.B. die Funktionen `tic`, `toc` oder einfach Ihre Armbanduhr (passen Sie ggf. die Dimension n entsprechend an).

Die kleinsten Quadrate. Der Ausdruck

```
c = A\b
```

hat auch dann einen Sinn, wenn A mehr Zeilen als Spalten hat. Das Ergebnis `c` hat die Eigenschaft, dass die Größe

$$\|Ax - b\|^2 \quad = \quad \sum_j |(Ax - b)_j|^2$$

ihren kleinsten Wert genau für $x = c$ annimmt. Wir sagen, dass das lineare Gleichungssystem

$$Ax = b$$

– das in der Regel unlösbar ist – dadurch *annähernd gelöst wird im Sinne der kleinsten Quadrate.* Es ist bekannt, dass dies äquivalent ist mit dem System

$$A^* Ax = A^* b$$

Man vergleiche selber die Operationen `A\b` und `(A'*A)\(A'*b)`. Die erste hat in der Regel größere Genauigkeit.

4.1. Beispiel. Gegeben sind n Punkte im Raum durch ihre Cartesischen Koordinaten

$$x_i, y_i, z_i, \quad i = 1, \ldots n, \quad n \geq 3 \ .$$

Wir wollen durch diese Punkte möglichst eine Ebene legen:

$$z \ = \ \alpha x + \beta y + \gamma$$

mit

$$z_i \; = \; \alpha x_i + \beta y_i + \gamma \;, \quad i = 1, \ldots, n \;.$$

oder anders geschrieben,

$$Aa \; = \; f \tag{4.1}$$

mit

$$A \;=\; \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix} \;, \quad a \;=\; \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \;, \quad f \;=\; \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \;.$$

Dies wird bei $n = 3$ möglich sein (drei Punkte bestimmen eine Ebene), bei $n > 3$ nur in Ausnahmefällen. Im Sinne der kleinsten Quadrate suchen wir a so, dass

$$\|Aa - f\|^2$$

minimal wird. All dies liefert automatisch die Operation `\`. Wir nehmen also an, dass die Koordinaten in Spaltenvektoren `x, y, z` gespeichert sind. Das Programm `ebene.m`:

```
function a = ebene(x,y,z)
% function ebene (x,y,z); x,y,z sind Zeilenvektoren
% mit den Koordinaten der Punkte, welche durch
% Z = a(1)*X+a(2)\cdot Y + a(3) im Sinne der
% kleinsten Quadrate approximiert werden
A = [x,y,ones(size(x))];
f = z;
a = A\f;
```

z.B.

```
>> x = [1 0 1]';
>> y = [1 1 2]';
>> z = [0 1 2]';
>> a = ebene(x,y,z)
a =
    -1
     2
    -1
```

Die gesuchte Ebene ist

$$z \; = \; -x + 2y - 1 \;.$$

Sind mehr Punkte vorhanden, etwa

```
>> x = [1 0 1 0 2]';
>> y = [1 1 2 1 1]';
>> z = [0 1 2 0 0]';
```

so haben wir

```
>> a = ebene(x,y,z)
a =
    -0.2727
     1.8182
    -1.3636
```

Die Ebene $z = a_1x + a_2y + a_3$ kann mit Hilfe der Funktion `mesh` graphisch dargestellt werden (wir lassen alle Variablen anzeigen):

```
>> xc = 0:0.5:2
xc =
     = 0 0.5000  1.0000  1.5000  2.0000
>> yc = xc'
yc =
     0
     0.5000
     1.0000
     1.5000
     2.0000
>>XC = ones(size(yc))*xc
XC =
     0  0.5000  1.0000  1.5000  2.0000
     0  0.5000  1.0000  1.5000  2.0000
     0  0.5000  1.0000  1.5000  2.0000
     0  0.5000  1.0000  1.5000  2.0000
     0  0.5000  1.0000  1.5000  2.0000
>>YC = yc*ones(size(xc))
YC =
         0         0         0         0         0
    0.5000    0.5000    0.5000    0.5000    0.5000
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.5000    1.5000    1.5000    1.5000    1.5000
    2.0000    2.0000    2.0000    2.0000    2.0000
>> ZC = a(1)*XC + a(2)*YC + a(3);
>> mesh(XC,YC,ZC)
```

Wir sehen, dass für das Zeichnen einer Fläche Matrizen notwendig sind. Genauso wie eine Fläche mit Hilfe von zwei Parametern

$$\begin{aligned} x &= x(u,v) \\ y &= y(u,v) \\ z &= z(u,v) \end{aligned}$$

analytisch dargestellt wird, so braucht `mesh` die Matrizen – die Rolle von u, v in den analytischen Formeln übernehmen die zwei Matrizen–Indizes. Die Gestaltung von Eingabeparametern bei `mesh` ist vielfältig und ermöglicht verschiedenartige Anwendungen. Merken Sie, wie wir die eindimensionalen Daten `xc,yx` in entsprechende „zweidimensionale“ `XC, YC` umgewandelt haben, denn nur so konnte ZC als Matrix gewonnen werden.

4.2. Aufgabe. Stellen Sie mit `mesh` die Funktionen

$$\begin{aligned} z &= x^2 - y^2 \ , \quad -1 < x < 1 \ , \quad -1 < y < 1 \\ z &= \frac{\sin\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}} \ , \quad x^2 + y^2 < 5 \\ z &= \sin x \sin y \quad x^2 + y^2 < 2 \end{aligned}$$

graphisch dar.

5 Rundungsfehler

Anders als die Pakete wie Mathematica oder Maple ist Matlab auf der Basis der „Fließpunkt–Rechnung“ basiert.[1] Das heißt, dass jede Zahl durch eine Dezimalzahl fester Länge ($\approx$ 16 Dezimalstellen) approximiert wird; genauer gesagt, sind es die Binärzahlen bzw. Binärstellen. Nur so können Probleme von realen Größen erst behandelt werden. Das aber bedeutet, dass bei den meisten Operationen Rundungsfehler entstehen. Das nennt sich „numerisches Rechnen“. Eine unmittelbare Kontrolle der Fehler und ihrer Fortpflanzung ist nicht möglich, möglich sind Abschätzungen, oft sehr grobe. Manchmal ist es so, dass gewisse Rechenverfahren (die als zuverlässig gelten) durch theoretische Fehlerabschätzungen nur teilweise gedeckt sind, das Vertrauen in das Verfahren basiert teilweise auf der Erfahrung in der Praxis.

Wir haben in unseren Rechenbeispielen oft gesehen, dass in Ergebnissen anstatt Null irgendeine „kleine Zahl“ vorkam, mit der man sich – weil sie klein war – auch zufriedenstellte. Sobald man aber sagen will, **ab wann** eine Zahl als klein zu gelten hat, kommt man in Schwierigkeiten, die nicht einfach zu bewältigen sind. Dieser Fragenkreis ist einer der Gegenstände der modernen *Numerischen Mathematik*.

Manchmal kann eine bessere Formelwahl die Genauigkeit steigern. Ein Beispiel: die quadratische Gleichung

$$10^{-20}x + x - 3 \;=\; 0\;, \quad -10 < x < 10$$

kann z.B. mit dem Programm `bisect` gelöst werden mit der Lösung $x = 3$. Eine andere Möglichkeit wäre es, die Lösungsformel

$$\lambda_1 \quad = \quad \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

einzusetzen:

```
>> a = 1e-20; b = 1 ; c = -3;
>> (-b + sqrt(b^2 - 4*a*c))/(2*c)
ans =
      0
```

Drastischer kann man die Wirkung der Rundungsfehler kaum zeigen! Das Ergebnis wurde hier als Differenz zweier sehr großer Zahlen erhalten. Da die Länge der Dezimalzahl zu kurz war, ging die Genauigkeit verloren. Die obige Formel kann aber auch als

$$\lambda_1 \quad = \quad \frac{2c}{b + \sqrt{b^2 - 4ac}}$$

umgeschrieben werden. Hier gibt es, wegen $b > 0$ keine gefährliche Subtraktion mehr und es ist

```
>> -2*c/(b + sqrt(b^2 - 4*a*c))
ans =
      3
```

1 Abgesehen davon, kann auch unter Matlab – als Erweiterung – sog. „Symbolisches Rechnen“ durchgeführt werden, wo keine Rundungsfehler stattfinden. Die praktische Einsetzbarkeit dieser Rechenart ist sehr beschränkt.

Unser Programm `quadr1` können wir nunmehr so verbessern:

```
function [l1,l2] = quadr2(a,b,c)
% function [l1,l2] = quadr2(a,b,c) liefert
% die Loesungen l1,l2 der quadratischen
% Gleichung a*x^2+b*x+c=0
diskr = b*b - 4*a*c;
if diskr >= 0
      if b >= 0
           l1 = (-b - sqrt(diskr))/(2*a);
           l2 = 2*c/(-b - sqrt(diskr));
      else
           l1 =(-b + sqrt(diskr))/(2*a);
           l2 = 2*c/(b + sqrt(diskr));
      end % if b
end % if diskr
```

5.1. Aufgabe. Interpretieren Sie das Programm `quadr2`. Führen Sie es unter „extremen Bedingungen“ aus. Setzen Sie $a = 0$ und interpretieren Sie das Ergebnis.

5.2. Aufgabe. Führen Sie die Zeile

```
>> a = 1e-5;  sh = (exp(a) - exp(-a))/2
```

aus. Dies liefert ja $\sinh a$. Holen Sie sich die Zeile zurück mit „Cursor-aufwärts“ und verkleinern Sie den Wert von `a`, etwa zu `1e-10` , `1e-20` . Vergleichen Sie das Ergebnis mit dem der eingebauten Funktion `sinh`. Kommentieren Sie den Unterschied (helfen Sie sich durch `type sinh`).

Nicht nur Subtraktion, sondern auch Addition lauter positiver Zahlen kann relativ große Fehler mit sich bringen:

```
>> a = 1 ;
>> for k = 1:1e+5
     a = a + 1e-17;
   end
>> a
a =
    1
```

Was ist geschehen? Nach so vielen Additionen lauter positiver Zahlen blieb die Variable `a` gleich ihrem Anfangswert 1. Kein Wunder: bei 16-stelliger Genauigkeit bleibt `1 + 1e-17` stets gleich 1. Man kann einwenden, dass wir, anders organisiert, das korrekte Ergebnis erhalten:

```
>> format long
>> 1 + 1e+5*1e-17
ans =
       1.00000000100000
```

Dies ist zwar hier richtig, kann aber bei komplexen Berechnungen nicht so einfach gemacht werden. Betrachten wir schließlich ein lineares Gleichungssystem mit der Matrix

```
>> A = [1+1e-15 1 1; 1 1+1e-15 1; 1 1 1];
>> x = [1;2;3];
>> b = A*x;
```

Nun würden wir als Lösung des Gleichungssystems

$$Ax = b$$

den obigen Vektor `[1; 2; 3]` erwarten. In der Tat ist aber

```
>> x = A\b
x =
     1.0000
     1.6000
     3.4000
```

und das ist ziemlich weit entfernt von der erwarteten Lösung. Hier liegt die Schuld nicht am verwendeten Algorithmus, sondern die Eingangsdaten, genauer die Matrix A, sind „schlecht konditioniert“ im Sinne, dass kleine Veränderungen in den Matrix-Elementen große Veränderungen in der Lösung bewirken. Die Funktion `cond` gibt die sog. *Konditionszahl* einer Matrix an; sie ist stets ≥ 1. Für unsere obige Matrix ist

```
>> cond(A)
ans =
      9.2177e+014
```

Ist `cond(A)` nahe der Zahl `1/eps`, so ist die Genauigkeit der Lösung zweifelhaft.

6 Ausgewählte Aufgaben

Wir wollen hier anhand von einigen etwas anspruchsvolleren Aufgaben den Umgang mit Matlab üben und einige weitere Funktionen kennenlernen. Dabei sind bewußt nicht alle Aufgaben mit kompletten Lösungen versehen.

6.1. Aufgabe. In der $z, x-$Ebene im dreidimensionalen Raum ist die Sinuskurve gegeben

$$x = \sin z \ , \quad -\pi \leq z \leq \pi$$

Rotiert diese Kurve um die $z-$Achse, so entsteht eine Fläche. Stellen Sie diese Fläche graphisch dar.

Lösung. Führen wir die Zylinderkoordinaten r, φ als

$$x = r\cos\varphi \ , \quad y = r\sin\varphi \ , \quad r = \sqrt{x^2 + y^2}$$

ein, so wird die Rotationsfläche durch

$$r = |\sin z|$$

beschrieben. (Man merke: $x = \sin z$ und $x = |\sin z|$ ergeben die gleiche Rotationsfläche!). Parameterdarstellung ist

$$\begin{aligned} x &= |\sin z|\cos\varphi \ , \quad -\pi < \varphi \leq \pi \\ y &= |\sin z|\sin\varphi \ , \\ z &= z \qquad\qquad\qquad -\pi \leq z \leq z \end{aligned}$$

Das Programm `sinrot.m` könnte so aussehen

```
z = -pi:0.1:pi;
phi = z';
Z = ones(size(phi))*z;
PHI=phi*ones(size(z));
X = abs(sin(Z)).*cos(PHI);
Y = abs(sin(Z)).*sin(PHI);
mesh(X,Y,Z)
```

Wollen Sie dieses schöne Bild drucken, so fügen Sie noch die Zeile hinzu

```
print -depsc sinrot.eps
```

Dies erzeugt die Datei `sinrot.eps`, welche beim Drucken verwendet werden kann. Mit `help print` erfahren Sie mehr über Druckvorbereitung von Graphiken bei Matlab*.

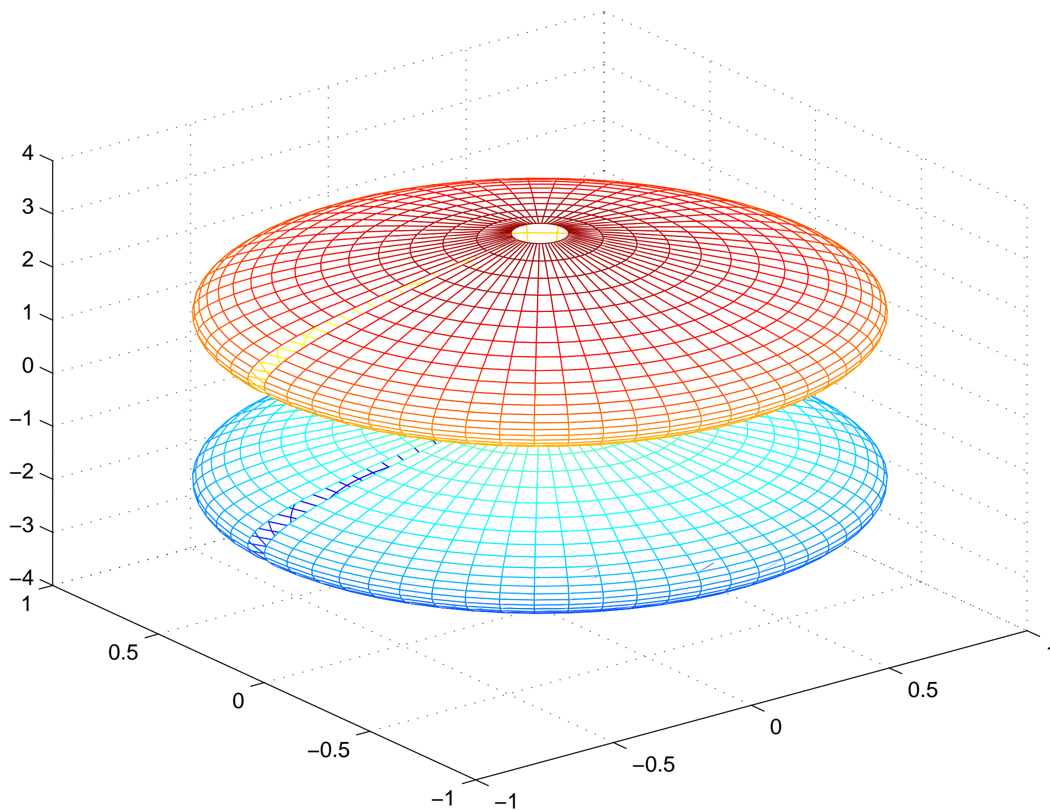


Abbildung 6.0.1

(Abb. 6.1)

6.2 Aufgabe. Finden Sie das Minimum der Funktion

$$z = x + 10\sqrt{x^2 + y^2 + 1} + (y-1)^2 .$$

Lösung. Diese Funktion ist auf ganz $\mathbb{R}^2$ definiert; bevor wir Programme einschalten, müssen wir den Definitionsbereich passend einschränken. Es ist

$$\begin{aligned} z(0,0) &= 11 \\ z &\geq 10|y| \\ z &\geq 9|x| \end{aligned}$$

also können wir uns z.B. auf das Gebiet

$$-2 < x < 2 , \quad -2 < y < 2$$

beschränken (außerhalb dieses Gebietes ist stets $z > 11$).

```
x = -2:0.1:2; y = x';
X = ones(size(y))*x; Y = y*ones(size(x));
Z = X + 10*sqrt(X.^2 + Y.^2 + 1) + (Y - 1).^2;
contour(X,Y,Z)
```

Hier sehen Sie die Niveaulinien, die auf eine Minimumstelle unweit des Ursprungs hindeuten. Nunmehr können wir die Funktion `fminsearch` des Matlab einschalten. Zuerst soll die zu minimierende Funktion in eine Funktion-Datei, z.B. `fn2.m` gespeichert werden. Die Funktion lautet

```
function z = fn2(xx)
z = xx(1) + 10*sqrt(xx(1)^2 + xx(2)^2 + 1) + (xx(2) - 1)^2;
```

Hier entspricht `xx(1)` der Variable x und `xx(2)` der Variable y. Nun wird mit

```
>> x0 = fminsearch('fn2',[0 0]);
```

die Minimumstelle in der Nähe des Ursprungs gesucht. Das Ergebnis ist

```
>> x0
x0 =
    -0.1019   0.1693
```

Das Minimum selbst ist

```
>> fn2(x0)
ans =
      10.7815
```

6.3 Aufgabe. Versuchen Sie, mindestens näherungsweise das obige Minimum zu finden, indem Sie die in der Variable Z enthaltene Funktionstabelle untersuchen (Hinweis: Benutzen Sie die eingebaute Funktion `min`). Können Sie auf diese Weise der Funktion `fminsearch` einen besseren Startwert besorgen?

6.4 Aufgabe. Finden Sie das Minimum von

$$W = 5(x-1)^2(x+1)^2 + (x+y)^2 + (x+y-z)^2$$

Vergleichen Sie das mit der exakten Lösung.

6.5 Aufgabe. Stellen Sie einen Torus graphisch dar. Ein Torus entsteht, wenn ein Kreis in der x,z–Ebene um die $z-$Achse im Raum rotiert. Versuchen Sie es mit den Kreisen

$$\begin{aligned}(x+2)^2 + z^2 &= 1\\ (x-1)^2 + (z-1)^2 &= 2\end{aligned}$$

6.6 Aufgabe. Stellen Sie einen Kreis mit einem eingeschriebenen gleichseitigen Dreieck dar. Sie müssen mit `plot` auskommen!

6.7 Aufgabe. Ein Teilchen fällt in der Luft senkrecht mit der Geschwindigkeit $v(t)$, deren Zeitveränderung v' (die Beschleunigung) wie folgt von v abhängt:

$$v' \;=\; -\alpha v - \beta v^3 - \gamma \;,$$

α, β, γ positive Konstanten. Finden Sie die Funktion auf dem Zeitintervall

$$\begin{array}{c} 0 \;\leq\; t \;\leq\; 0 \quad \text{für} \\ \alpha, \quad \beta \quad = 1 \;, \quad \gamma = 0.5 \end{array}$$

bei der Anfangsgeschwindigkeit

$$v_0 \;=\; v(0) \;=\; 0$$

Stellen Sie alles graphisch dar.

Lösung. Hier soll eine Differentialgleichung gelöst werden. Mit `help funfun` erfahren wir, dass es eine Funktion `ode23` gibt (Ordinary Differential Equation). Mit `help ode23` erfahren wir, dass die „rechte Seite“ der Differentialgleichung in einer Funktion-Datei stehen soll, etwa in `vfn.m`:

```
function vprime = vfn(t,v)
vprime = -v - v^3 -0.5;
```

Nun der `ode23`–Aufruf:

```
>> [t,v] = ode23('vfn',[0,10],0);
>> plot(t,v)
```

Aus der Graphik ersehen wir, dass die Lösung sich für $t \longrightarrow \infty$ einer Konstante nähert. In der Tat, setzen wir in unsere Differentialgleichung $v = v_0 = \text{const.}$, so ergibt sich

$$v_0^3 + v_0 + 0.5 = 0$$

Dafür gibt es keine Lösungsformel, aber eine Matlab Funktion `fzero`. In der Tat haben wir

```
>> v0 = fzero('fnv0', -0.5)
v0 =
    -0.4239
```

Dabei ist die Funktion `fnv0.m`

```
function vprime = fnv0(v)
vprime = -v - v^3 - 0.5;
```

nur „technisch“ verschieden von `vfn.m` – sie hat nur einen Eingabeparameter. Beide Lösungen sind aus

```
plot(t,v,t,v0*ones(size(t)))
```

sichtbar. [1]

6.8 Aufgabe. Auf der Ebene, versehen mit dem Cartesischen Koordinatensystem, bewegen sich zwei Schiffe A und B. Das Schiff A bewegt sich gemäß

$$x_1 = 10t \;, \quad y_1 = 1$$

(x_1, y_1 Abstand in Kilometern, t Zeit in Stunden). Das Schiff B bewegt sich gemäß

$$\begin{aligned} x &= x(t) \quad , \quad x(0) = 0 \\ y &= y(t) \quad , \quad y(0) = 0 \end{aligned}$$

mit der Geschwindigkeit $v = 12$km/h, aber nicht geradlinig wie A, sondern so, daß sein Bug stets in Richtung von A schaut (A wird von B „gejagt“).
Berechnen Sie die Bahn von B und stellen Sie die „Jagd“ graphisch dar.

Lösung. Es ist

$$\begin{aligned} x' &= \alpha(x_1 - x) \\ y' &= \alpha(y_1 - y) \end{aligned}$$

wobei $\alpha > 0$ aus der Forderung

$$x'^2 + y'^2 = v^2$$

zu bestimmen ist:

$$\alpha^2((x_1 - x)^2 + (y_1 - y)^2) = v^2$$

$$\alpha = \frac{v}{\sqrt{(x_1 - x)^2 + (y_1 - y)^2}}$$

Zu lösen ist also das Differentialgleichungssystem

$$\begin{aligned} x' &= \frac{v(x - x_1)}{\sqrt{(x_1 - x)^2 + (y_1 - y)^2}} \;, \quad x'(0) = 0 \\ y' &= \frac{v(y - y_1)}{\sqrt{(x_1 - x)^2 + (y_1 - y)^2}} \;, \quad x'(0) = 0 \end{aligned}$$

1 Bei Octave heißen die Funktionen anders: `lsode` und `fsolve`.