Mathematische Software

Vorlesung, zuerst gehalten im Sommersemester 2015

Tomas Sauer

Version 1.0
Letzte Änderung: 15.4.2016

Statt einer Leerseite ... 0

Denn die Doktrin der geschlechtergerechten Sprache macht das Lesen solchermassen „gerechter“ Texte nicht nur fast unerträglich. Sie basiert auch auf einem linguistischen Grundirrtum, weil es das biologische Geschlecht mit dem grammatischen Genus gleichsetzt.

C. Wirz, „Neusprech für Fortgeschrittene“, *NZZ Online, 12.7.2013*

Die wahren Analphabeten sind schließlich diejenigen, die zwar lesen können, es aber nicht tun. Weil sie gerade fernsehen.

L. Volkert, *SZ–Online, 11.7.2009*

$$\frac{(a+b)!}{a!b!} \geq \sqrt{\frac{(a+b)^{a+b}}{a^a b^b}}.$$

And it didn't stop being magic just because you found out how it was done.

T. Pratchett, *Wee Free Men*

Wissen Sie, was ich an der Tugend nicht mag? Wer sich ehrlos verhält und nicht erwischt wird, der hat davon einen Vorteil. Wer sich aber tugendsam verhält, der hat in den meisten Fällen gar nichts davon. Das ist der eigentliche Skandal!

H. Martenstein, *Zeit Online, 19.1.2012*

Tomas Sauer
Lehrstuhl für Mathematik mit Schwerpunkt Digitale Bildverabeitung
Universität Passau
Innstr. 43
94032 Passau

0 Inhaltsverzeichnis

> *The Vertrauensmann is man we are trusting. Not yesterday, maybe not tomorrow. But today we are trusting him for ever.*
>
> J. le Carré, *A Perfect Spy*

1 Grundlagen

Die Vorlesung *Mathematische Software* soll zeigen, wie man Software nutzen kann, um mathematisches Know–How zu nutzen, insbesondere, wenn es nur darum geht, Dinge auszurechnen.

1.1 Wo gibt's die Software?

Zuerst einmal die Verfügbarkeit und Homepages der Software[1]:

Matlab ist ein kommerzielles Produkt, das von der Firma MathWorks, deutsche Homepage

```
de.mathworks.com/
```

vertrieben wird. Trotz der relativ teueren Lizensierung ist es ein de facto Standard in der Industrie, Studentenlizenzen können zumeist zu relativ günstigen Konditionen erworben werden. Die Universität Passau verfügt über eine hinreichende Zahl von sogenannten *Classroom–Lizenzen* zu Ausbildungszwecken, die auf dem Rechnerpool der FIM installiert sind und dort genutzt werden können.

Octave ist eine OpenSource–Version von `Matlab`, die *weitgehend* kompatibel ist und unter

```
www.octave.org
```

zu finden ist. Dort kann man auch das Handbuch (Eaton *et al.*, 2009) herunterladen. Nach ersten Versuchen 1988 wird `Octave` seit 1992 professionell entwickelt und hat sich zu einer sehr stabilen und professionellen Software entwickelt. Unter

```
octave.sourceforge.net
```

gibt es darüberhinaus eine Vielzahl von Toolboxen, teilweise kompatibel mit `Matlab`–Toolboxen, die die Funktionalität von `Octave` erweitern.

[1] Stand: April 2015.

R ist als OpenSource–Version der Statistik–Sprache `S` (entwickelt von den Bell Laboratories) entwickelt worden und kann unter

`www.r-project.org`

gefunden werden. `R` (R Core Team, 2013) ist ein GNU–Projekt, wie auch bei `Octave` besteht auch die Möglichkeit, das System mit selbstgeschriebenen Funktionen in `C` oder `C++` interagieren zu lassen.

Maxima ist ein Computer–Algebra–System (CAS) und ebenfalls OpenSource, diesmal aus dem Projekt `Macsyma`, das bereits in den späten 1960ern am MIT entwickelt wurde und das seine beiden populären (aber kommerziellen) Nachfolger `Maple` und `Mathematica` stark beeinflusst hat. Zu finden ist es unter

`maxima.sourceforge.net`

und es ist, im Gegensatz zu den anderen Programmen, in Lisp geschrieben. Zu `Maxima` gibt es eine Benutzeroberfläche namens `wxmaxima`, die (mindestenst) auf allen Linuxsystemen läuft.

Alle Programme sind in vielen Linux–Distributionen bereits enthalten, beispielsweise in Ubuntu 14.

1.2 Was gibt's noch?

Ein paar weitere Programme für mathematische Zwecke sind:

- `CoCoA` Ein CAS, ebenfalls OpenSource, aus Italien, das besonders stark auf Idealtheorie zielt.
- `Singular` Ebenfalls ein CAS, allerdings für wirklich substantielle Anwendungen, vor allem in algebraischer Geometrie. Nichts zum einfach mal rumspielen.
- `GAP` Ein System für *Computational Discrete Algebra*, insbesondere zum Arbeiten mit (endlichen) Gruppen.
- `Scilab` Ein teilweise `Matlab`–ähnliches System, das mit `Xcos` auch über ein System zur Simulation verfügt, das **Simulink** ähnelt.

All diese Programme sind[2] kostenfrei verfügbar und teilweise in Linux–Distributionen enthalten, `Scilab` ist aber kein OpenSource–Projekt.

[2]Stand April 2015, sowas kann sich leider immer mal wieder ganz schnell ändern.

Aber vor lauter Klammern und Fußnoten verstand er kein Wort, und wenn er gewissenhaft mit den Augen den Sätzen folgte, war ihm, als drehe eine alte, knöcherne Hand ihm das Gehirn in Schraubenwindungen aus dem Kopfe.

R. Musil, *Die Verwirrungen des Zöglings Törleß*

2 Matlab

Als erstes System wollen wir uns `Matlab` bzw. `Octave` ein wenig genauer ansehen.

2.1 Sprachelemente

2.1.1 Grundrechenarten

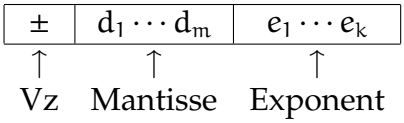
`Matlab` rechnet **numerisch**, also mit Zahlen in einer *endlichen* Darstellung

$$\pm\left(\sum_{j=1}^{m} d_j B^{-j}\right) B^e, \qquad d_j \in \{0,\ldots,B-1\},\ e \in E,$$

die wir auch als

$$\pm.\, d_1 \cdots d_m \ \times\ B^e$$

schreiben können. Die Basis B kann gemäß IEEE–Standard entweder $B = 2$ oder $B = 10$ sein und sieht schematisch folgendermaßen aus



Der Exponent wird in einem „ganz normalen“ vorzeichenbehafteten Ganzzahlformat (Einer- oder Zweierkomplement?) dargestellt. Achtung: damit wird der Exponentenbereich normalerweise nicht symmetrisch sein. Aufgrund dieser Darstellung sind **Rundungsfehler** möglich.

Beginnen wir mit ganz einfachen Rechnungen, hier der Einfachheit halber in `Octave`

```
octave> 2 + 2
ans =  4
octave> 2 + 2;
```

Wird eine Zeile mit dem Semikolon ; abgeschlossen, so wird die Ausgabe des Ergebnisses unterdrückt, andernfalls wird das in der Variablen ans angezeigte Ergebnis angezeigt. ans kann man weiterverwenden:

```
octave> 2+2
ans =  4
octave> ans * 3
ans =  12
```

Ergebnisse von Rechenoperationen können auch Variablen zugewiesen werden:

```
octave> x = 1/6;
octave> (5*x+x) - sqrt( x^2 + 10*x^2 + 25*x^2 )
ans =   -1.1102e-16
```

Hier sehen wir mal einen **Rundungsfehler** in freier Wildbahn: Eigentlich berechnen wir ja

$$5x + x - \sqrt{x^2 + 10x^2 + 25x^2} = 6x - \sqrt{(x + 5x)^2} = 6x - 6x = 0.$$

Der Standard–Rundungsfehler, siehe (Higham, 2002; Sauer, 2013), ist in der Variablen eps abgespeichert. Hier gilt

```
octave> (1+eps)-1
ans =    2.2204e-16
octave> (2+eps)-2
ans = 0
```

das heißt, gegen die Zahl 2 ist der Fehler nicht mehr messbar, und zwar gerade gegen 2:

```
octave> x=2-eps; (x+eps) - x
ans =    2.2204e-16
```

Wir können also auch mehrere Befehle in eine Zeile schreiben, wenn wir sie durch ein Semikolon trennen. Zahlen müssen nicht reell sein, sie können auch komplex sein, wobei i für die komplexe Einheit steht:

```
octave> x = (1+i); y=(2-i); x*y
ans =  3 + 1i
```

Der Absolutbetrag wird auf die Situation, also auf reell oder komplex[3] angepasst:

```
octave> abs(-1)
ans =  1
octave> abs(-1+i)
ans =  1.4142
```

Auch Vergleichsoperatoren liefern, ähnlich wie in C, eine Ergebnis, das weiterverarbeitet werden kann. Hier der klassische Prozessortest[4]

```
octave> x = .1; 10*x == 1
ans =  1
```

[3] $|x + iy| = \sqrt{x^2 + y^2}$.

[4] Diese Berechnung kann je nach Prozessor den Wert 0 oder 1 liefern und beides ist richtig.

2.1.2 Matrizen und Vektoren

Matrizen und Vektoren sind die Standardobjekte in `Matlab`. Tatsächlich steht der Name nämlich für *Matrix Laboratory*. Deswegen bietet `Matlab` auch eine Vielzahl von Möglichkeiten, auf Matrizen und Vektoren zuzugreifen. Ein **Zeilenvektor** ist von der Form

```
octave> x = [ 1,2,3,4,5 ]
x =

   1   2   3   4   5
```

wobei die Kommata auch weggelassen werden können, ein **Spaltenvektor** wird als

```
octave> x = [ 1;2;3;4;5 ]
x =

   1
   2
   3
   4
   5
```

geschrieben. Eine komplette Matrix ist

```
octave> A = [ 1,2 ; 3,4 ]
A =

   1   2
   3   4
```

Matrizen werden immer *zeilenweise* eingegeben, sonst gibt es eine Fehlermeldung

```
octave> A = [ 1;3 , 2;4 ]
error: vertical dimensions mismatch (1x1 vs 1x2)
```

man kann allerdings auch Matrizen aus Spaltenblöcken zusammensetzen:

```
octave> A = [ [ 1;3] [ 2;4 ] ]
A =

   1   2
   3   4
```

Die **Transposition** einer Matrix erfolgt durch den Apostroph

```
octave> A = [ 1 2 ; 3 4 ]; B = A'
B =

   1   3
   2   4
```

wobei das so nicht ganz richtig ist, denn eigentlich ist es die **Hermitesche** Matrix, da komplex konjugiert wird:

```
octave> A = [ 1+i 2 ; 3 4+i ]; B = A'
B =

   1 - 1i   3 - 0i
   2 - 0i   4 - 1i
```

Will man wirklich die Transponierte[5], so heißt der Operator „.'“: oder man verwendet die Funktion `transpose`:

```
octave> A = [ 1+i 2 ; 3 4+i ]; B = A.'
B =

   1 + 1i   3 + 0i
   2 + 0i   4 + 1i
```

Zeilenvektoren mit ganzzahligen Einträgen bekommt man mit `(a:b)` bzw. `(a:inc:b)`, wobei `inc` auch negativ sein kann, dann erhält man die Zahlen in absteigender Reihenfolge

```
octave> (1:10)
ans =

    1    2    3    4    5    6    7    8    9   10

octave> (1:2:10)
ans =

    1    3    5    7    9
```

Matrix–Vektor–Produkte sind definiert[6] wann immer die Dimensionen passen und werden einfach als * geschrieben:

```
octave> [ 1 2 ; 3 4 ] * [ -1 -2 ; -3 -4 ]
ans =

   -7  -10
  -15  -22
```

Das gilt insbesondere für Zeilen und Spaltenvektoren, für die man das **innere Produkt** oder **Skalarprodukt**

[5] Also *ohne* Konjugation.

[6] Zur Erinnerung: Das Produkt einer $m \times r$–Matrix A und einer $r \times n$–Matrix B ist eine $m \times n$–Matrix, deren Komponenten sich als

$$(AB)_{jk} = \sum_{p=1}^{r} a_{jp} b_{pk}, \qquad j = 1, \ldots, m,\ k = 1, \ldots, n, \tag{2.1}$$

ergeben. Aber das haben sicherlich alle gewusst.

```
octave> (1:4) * (1:4)'
ans =  30
```

oder das **äußere Produkt** oder **Tensorprodukt**

```
octave> (1:4)' * (1:4)
ans =

    1    2    3    4
    2    4    6    8
    3    6    9   12
    4    8   12   16
```

berechnen kann, je nachdem, wo man die Transposition hinsetzt. Es kann nicht schaden, sich einmal anhand von (2.1) zu überlegen, was in diesen beiden Fällen passiert.

Besonders clever ist `Matlab`, wenn es darum geht, Teilmatrizen herauszupicken und zu indizieren. Ist `A` eine Matrix–Variable, dann kann man sich mit `A( a:b,c:d )` die Teilmatrix geben lassen, die aus den Zeilen mit Index $a, a+1, \ldots, b$ und den Spalten mit Index $c, c+1, \ldots, d$ besteht. Zusammen mit dem letzten Beispiel bekommen wir dann

```
octave> A = (1:4)' * (1:4); A( 1:2,3:4)
ans =

   3   4
   6   8
```

Aber `1:2` und `3:4` erinnert uns doch an etwas, nämlich an die Vektoren. Und tatsächlich können wir *jeden* Zeilenvektor zur Indizierung verwenden:

```
octave> x=(1:2); y = (3:-1:1); A(x,y)
ans =

   3   2   1
   6   4   2
```

Damit können wir also Zeilen und Spalten einer Matrix sogar permutieren. Und wollen wir eine ganze Zeile oder Spalte, dann schreiben wir einfach nur den Doppelpunkt:

```
octave> A(4,:)
ans =

    4    8   12   16
```

Funktionen können auf Matrizen angewendet werden und werden normalerweise auf die Komponenten der Matrix angewandt:

```
octave> A = [ 1 2 ; 3 4 ]; exp( i*A )
ans =

   0.54030 + 0.84147i  -0.41615 + 0.90930i
  -0.98999 + 0.14112i  -0.65364 - 0.75680i
```

Man kann Matrizen auch komponentenweise verarbeiten, beispielsweise mutlitplizieren, indem man den Punkt davorsetzt

```
octave> A = [ 1 2 ; 3 4 ];  A .* A
ans =

    1    4
    9   16
```

Zum Vergleich das „normale“ Matrixprodukt

```
octave> A = [ 1 2 ; 3 4 ];  A * A
ans =

    7   10
   15   22
```

Analog sind auch `A^2` und `A.^2` zu verstehen. Schöne Tricks sind möglich, wenn man Vergleichsoperatoren auf Matrizen anwendet:

```
octave> A = [ 1 2 ; 3 4 ];  A + ( A > 2 )
ans =

   1   2
   4   5
```

Klar, war hier passiert? `A > 2` liefert eine Matrix, die an Stellen den Wert 1 hat, die die Relation erfüllen und sonst 0.

2.1.3 Kontrollstrukturen und Funktionen

Man kann in `Matlab` auch programmieren. Dazu gibt es Kontrollstrukturen, insbesondere

```
if
% ....
else
% ...
end
```

und verschiedenste Schleifen. Die klassische `for`–Schleife mit ganzzahligem Argument wird wie ein Vektor indiziert. Hier der kleine Gauß:

```
octave> s = 0; for k=1:100 s = s+k; end; s
s =  5050
```

Der Strichpunkt hinter dem `end` ist nur hier als Trennzeichen zwischen den Befehlen nötig, es geht auch

```
octave> s = 0; for k=1:100 s = s+k; end
octave> s
s =  5050
```

Alle Tricks, die man bei der Erstellung von Zeilenvektoren benutzen kann, funktionieren auch bei Schleifen. Es gibt auch eine `while`–Schleife

```
while (...)
% ...
end
```

die im Gegensatz zum `do ... until` Konstrukt auch in `Matlab` und `Octave` einheitlich funktioniert.

Funktionen können in `Matlab` zwar auch direkt definiert werden, mehr Spass machen sie aber als separate Dateien. Eine Funktion ist von der Form

```
function res = Name( ... )
end
```

wobei `res` das Resultat der Funktion ist[7]. Beginnen wir mit dem **Heron–Verfahren**, das mittels der Iteration

$$x_{j+1} = \frac{1}{2}\left(x_j + \frac{a}{x_j}\right), \qquad j \in \mathbb{N}_0, \qquad x_0 = a$$

die **Quadratwurzel** einer Zahl bestimmt. Die Idee ist einfach[8]: Ist x_j ein „geratener" Wert für eine Seitenlänge eines Rechtecks[9] mit Fläche a, dann ist x_{j+1} einfach der Mittelwert zwischen der kurzen und der langen Seite und das konvergiert anschaulich[10] gegen die Seitenlänge des Qudrats mit Fläche a und die ist $\sqrt{a}$. Sehen wir uns Programm 2.1 einmal an. Es werden zwei Parameter übergeben, einmal der Wert `x`, aus dem die Wurzel gezogen werden soll und dazu eine Toleranz `tol`, mit der man vorschreiben kann, welchen relativen Fehler $x_j^2 - a$ machen darf, damit noch weitergerechnet wird. Aufgerufen wird die Funktion[11] einfach unter ihrem Namen:

```
octave> Heron( 2,.1 )
ans =  1.4167
octave> Heron( 2,eps )
ans =  1.4142
```

Eine kleine Warnung: Man kann *nicht* davon ausgehen, daß eine Rechnung immer auf einen Fehler von `eps` kommt und dann kann sich so eine `while`–Schleife auch mal ganz schnell ins Nirwana verabschieden. Eine Funktion kann

[7] Funktionen können mehrere Werte zurückgeben, wenn man es richtig macht, sogar beliebig viele, aber fangen wir doch mal einfach an.

[8] Und findet sich bereits aus babylonischen Keilschrifttafeln von etwa 3000 v.Chr.

[9] Die andere Seite hat dann die Länge $\frac{a}{x_j}$.

[10] Und auch wirklich, also beweisbar.

[11] Die entweder in dem Verzeichnis stehen muss, in dem `Matlab` aufgerufen wurde oder die sich im Suchpfad befinden muss, wie das genau geht, entnimmt man dann der jeweiligen Anleitung.

```
%% Heron.m
%% Heronverfahren
%%
%% x = Startwert
%% t = Toleranz (relativ)

function [y,Iter] = Heron( x,t )
  y = x;
  while ( abs( y^2 - x ) > t*x )
    y = ( y + x/y ) / 2;
  end
end
```

Programm 2.1: Das **Heron–Verfahren** zur Bestimmung der Wurzel einer Zahl.

```
%% Heron2.m
%% Heronverfahren mit Anzahl der Iterationen
%%
%% x = Startwert
%% t = Toleranz (relativ)

function [y,Iter] = Heron2( x,t )
  y = x; Iter = 0;
  while ( abs( y^2 - x ) > t*x )
    y = ( y + x/y ) / 2;
    Iter = Iter + 1;
  end
end
```

Programm 2.2: Das **Heron–Verfahren** mit Angabe der Anzahl der Iterationen.

auch mehrere Werte zurückgeben, die dann in eckigen Klammern aufgelistet werden:

```
function [res1,res2,... ] = Name ( ... )
```

Ein Beispiel wäre ein Heronverfahren, das angibt, wie viele Iterationen benötigt wurden, siehe Programm 2.2.

```
octave> [y,It] = Heron2( 2,.1 )
y =  1.4167
It =  2
octave> [y,It] = Heron2( 2,eps )
y =  1.4142
It =  5
```

Interessanter Nebeneffekt: Die Berechnung von $\sqrt{2}$ in *maximaler Rechengenauigkeit* braucht nur drei Iterationen mehr als die Berechnung auf eine Nachkommastelle. Ruft man eine Funktion mit mehreren Rückgabewerten „naiv“ auf, wird der erste Wert zurückgegeben:

```
octave> Heron2( 2,eps )
ans =  1.4142
```

Noch ein netter Gag am Rande:

```
octave> [y,It] = Heron2( 25,eps )
y =  5
It =  7
octave> 25-y^2
ans = 0
```

Nach sieben Iterationen liefert das Heronverfahren für 25 das *exakte* Ergebnis 5, obwohl die Theorie sagt, daß dies unmöglich ist, das Heron–Verfahren *konvergiert* nur gegen die Wurzel, berechnet sie aber nie exakt. Daß es trotzdem klappt liegt an dem Rundungsfehler. Fazit:

> Rundungsfehler sind nicht immer böse.

Eine einfache Methode, beliebig viele Rückgabewerte zu handhaben ist, diese

```
%% Matn.m
%% Matrix mit Zeilen (1:n) ... (n:2n)
%%
%% n = Groesse

function M = Matn( n )
  M = (1:n);
  for j=2:n
    M = [ M; (j:n+j-1) ];
  end
end
```

Programm 2.3: Berechnung einer Matrix flexibler Größe, bei der einfach Zeilen übereinander gestapelt werden.

in einen Vektor oder eine Matrix anzuordnen und dann einfach dieses Objekt zurückzugeben. Ein Beispiel hierfür ist Programm 2.3, in dem Zeilen gestapelt werden, um so Matrizen variabler Größe zu generieren. Das Ergebnis sieht wie folgt aus:

```
octave> Matn( 3 )
ans =
```

```
1   2   3
2   3   4
3   4   5
```

2.2 Lösen von Gleichungssystemen

Nachdem `Matlab` nun schon so gut im Umgang mit Matrizen ist und eigentlich dafür gestrickt ist, ist es kein Wunder, daß man mit `Matlab` besonders gut lineare Gleichungssysteme behandeln kann.

2.2.1 Lineare Gleichungssysteme in `Matlab`

Zum Lösen eines Gleichungssystems $Ax = y$, $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, benutzt man den Operator \, der eine „Division von links“ (also wie die Division /, nur andersrum) eines Vektors durch eine Matrix darstellen soll:

```
octave> A = [ 1 2 ; 3 4 ]; y = [ 1;1 ]; A\y
ans =

  -1
   1
```

Gleichungssysteme kann man auch mit mehreren, sagen wir k rechten Seiten betrachten, also

$$Ax_j = y_j, \qquad j = 1, \ldots, k.$$

Schreiben wir diese Vektoren in eine Matrix nebeneinander[12], dann ist

$$\mathbb{R}^{m\times k} \ni Y := [y_1 \ldots y_k] = [Ax_1 \ldots Ax_k] = A[x_1 \ldots x_k] = AX, \qquad X \in \mathbb{R}^{n\times k}.$$

Damit können wir also auch Gleichungssysteme mit passenden[13] *matrixwertigen* rechten Seiten betrachten:

```
octave> A\[1 1 ; 1 -1]
ans =

  -1.0000  -3.0000
   1.0000   2.0000
```

löst dann $Ax = y$ für die rechten Seiten $y = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ und $y = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$. Formal heißt das nichts anderes als $AX = Y$. Ist $Y = I$, dann muss X die **Inverse** von A sein:

```
octaveA\eye(2)
ans =

  -2.00000   1.00000
   1.50000  -0.50000
```

[12] So wie es die Notation von `Matlab` ja ohnehin nahelegt.

[13] Man kann es drehen und wenden, meinetwegen auch transponieren, wie man will, man kann mit Matrizen eine Menge anstellen, aber halt nur, wenn die Dimensionen passen.

Die „Division von links“ kennen wir aus der Theorie als Multiplikation $x = A^{-1}y$ mit der Inversen von A, ebenfalls wieder von links, was ebenfalls möglich ist und erst einmal dasselbe Ergebnis liefert:

```
octave> A^(-1)*y
ans =

  -1.00000
   1.00000
```

Die Nullen sollten uns zu einer gewissen Vorsicht warnen und in der Tat ist es offensichtlich nicht dasselbe:

```
octave> A^(-1)*y - A\y
ans =

   2.2204e-16
  -1.1102e-16
```

Sehen wir uns das einmal genauer an: Die **Inverse** A^{-1} einer Matrix erhält man entweder mit

```
octave> inv(A)
ans =

  -2.00000   1.00000
   1.50000  -0.50000
```

oder, wie schon gesehen, mit

```
octave A^(-1)
ans =

  -2.00000   1.00000
   1.50000  -0.50000
```

Diese beiden Operationen sind identisch. Auch hier sind wieder so auffällig viele Nullen zu sehen, die uns sicherlich etwas sagen wollen. In der Tat ist für $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ ja[14]

$$A^{-1} = -\frac{1}{2}\begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix},$$

was erstaunlicherweise am Rechner *exakt* darstellbar wäre. Trotzdem ist

[14] Die leicht zu überprüfende Formel

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

```
octave> inv(A) - [ -2 1 ; 1.5 -.5 ]
ans =

   4.4409e-16  -2.2204e-16
  -2.2204e-16   1.1102e-16
```

nur „numerisch“ Null, was sich auch in

```
octave> A*inv(A) - eye(2)
ans =

   0.0000e+00   0.0000e+00
   8.8818e-16  -4.4409e-16
```

niederschlägt. Und um noch ein wenig zur Verwirrung beizutragen:

```
octave> inv(A) - A\eye(2)
ans =

   2.2204e-16  -2.2204e-16
   0.0000e+00   5.5511e-17
```

Und nur um es klarzustellen: *Exakt* dasselbe Ergebis erhält man bis auf Darstellung auch bei `Matlab`[15]

```
>> A = [ 1 2 ; 3 4 ]; inv(A) - A\eye(2)

ans =

   1.0e-15 *

    0.2220   -0.2220
         0    0.0555
```

Da passiert offenbar was[16] und es ist sinnvoll, das herauszubekommen, was wir im Theorieteil auch tun werden. Davor aber noch ein kurzer Auszug aus „`help inv`“ von `Octave`:

```
In general it is best to avoid calculating the inverse of a matrix
directly.  For example, it is both faster and more accurate to
solve systems of equations (A*x = b) with 'Y = A \ b', rather than
'Y = inv (A) * b'.
```

Doch auch sonst ist das mit der Inversen nicht so klug, wie ein weiteres Extrembeispiel zeigt:

[15] Version R2014a, 64-bit Linux.

[16] Und man sollte sich dabei klarmachen daß es sich hier um eine 2×2–Matrix mit ganzzahligen Einträgen handelt, deren Inverse in jedem Fließpunktformat *exakt* dargestellt werden kann!

```
octave> [ 1 0 ; 0 0 ] \ [ 1;0 ]
ans =

   1
   0
```

Matlab liefert sogar

```
>> [ 1 0 ; 0 0 ] \ [ 1;0 ]
Warning: Matrix is singular to working precision.

ans =

     1
   NaN
```

Klar, die Matrix ist ja auch singulär und nicht invertierbar, obwohl das Gleichungssytem

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

lösbar ist und zwar mit $x = \begin{bmatrix} 1 \\ a \end{bmatrix}$, $a \in \mathbb{R}$. Octave liefert eine spezielle Lösung und zwar[17] diejenige kleinster **Norm**, Matlab erklärt etwas unrichtig das Problem für unlösbar und gibt auch ein etwas seltsames Ergebnis zurück. Hier eine wichtige Lektion:

> Im Falle von singulären Matrizen wird das Verhalten der Programme erratisch und Matlab und Octave können sich unterschiedlich verhalten. Auf eine „falsche Frage“ gibt es eben keine „richtige Antwort“.

Beschäftigen wir uns als nächstes mit der numerischen Stabilität von Lösungen. Dazu betrachten wir die Matrizen $A_t = \begin{bmatrix} 1 & 1 \\ 1 & 1+t \end{bmatrix}$, $t \geq 0$. Für $t = 0$ ist das Gleichungssystem nicht lösbar, und das bekommen wir auch zu spüren:

```
octave> A = [ 1 1 ; 1 1 ]; A\eye(2)
warning: matrix singular to machine precision, rcond = 0
ans =

   0.25000   0.25000
   0.25000   0.25000
```

Das mit eye(2), also der Inversen von A, ist reine Bequemlichkeit. Wir sehen sofort, daß die Lösung keine ist:

[17] Nicht zufällig!

```
octave> ans * A
ans =

   0.50000   0.50000
   0.50000   0.50000
```

ist offenbar nicht wirklich nahe an der Einheitsmatrix, aber immer noch besser als die Antwort von Matlab

```
>> A = [ 1 1 ; 1 1]; A\eye(2)
Warning: Matrix is singular to working precision.

ans =

   Inf  -Inf
  -Inf   Inf

>> A*ans

ans =

   NaN   NaN
   NaN   NaN
```

Übung 2.1 Erklären Sie, warum es zu den NaNs kommt. ◊

Jetzt machen wir die Matrix mal ein bisschen invertierbarer:

```
octave> A = [ 1 1 ; 1 1+eps ]; A\eye(2)
warning: matrix singular to machine precision, rcond = 5.55112e-17
warning: matrix singular to machine precision, rcond = 5.55112e-17
ans =

   0.25000   0.25000
   0.25000   0.25000

octave:4> ans*A
ans =

   0.50000   0.50000
   0.50000   0.50000
```

bzw.

```
>> A = [ 1 1 ; 1 1+eps]; A\eye(2)

ans =

   1.0e+15 *
```

```
    4.5036   -4.5036
   -4.5036    4.5036
```

Aber jetzt sieht man, was passiert: Die `Octave`–Lösungen bleiben stabil, die `Matlab`–Lösungen gehen gegen ∞ und immerhin ist das ein guter Trend:

```
>> ans*A

ans =

     1     0
     0     1
```

Im Moment hat also `Matlab` recht! Mit ein bisschen mehr Störung unten rechts funktioniert dann auch `Octave`

```
octave> A = [ 1 1 ; 1 1+10*eps ]; A\eye(2)
ans =

   4.5036e+14  -4.5036e+14
  -4.5036e+14   4.5036e+14

octave> ans*A
ans =

   1   0
   0   1
```

Gut, es gibt also Probleme, wenn man versucht, unlösbare Gleichungssysteme numerisch zu lösen, was so richtig überraschend das allerdings nicht sein sollte. Viel interessanter ist die Fehlermeldung, die auf ein `rcond` hinweist, was der Reziprokwert der **Konditionszahl**, genauer gesagt, eine Abschätzung dafür, ist. Die Konditionszahl wird in Definition 2.4 eingeführt. Man kann sie mit

```
octave> A = [ 1 1 ; 1 1+3*eps ]; cond(A)
ans =    6.3691e+15
```

bzw.

```
octave> rcond(A)
ans =    1.6653e-16
```

berechnen lassen. `cond` *berechnet* die Konditionszahl $\kappa(A)$, wobei man die Normen noch vorgeben kann, `rcond` liefert hingegen eine *untere Abschätzung* für $\kappa^{-1}(A)$, die schnell und billig berechnet wird, siehe (Anderson *et al.*, 1995), was überhaupt eine sehr interessante Quelle für alle ist, die sich *seriös* für numerische lineare Algebra und wie die Dinge wirklich funktionieren interessieren. Man kann zeigen, daß es absolut keinen Sinn hat, ein Gleichungssystem numerisch zu rechnen, wenn der Reziprokwert der Konditionszahl kleiner ist als die Rechengenauigkeit, also

```
rcond(A) < eps
```

gilt. Das ist die Bedeutung der Warnung, die man auf jeden Fall ernstnehmen sollte.

Das war's dann auch schon zu den quadratischen Gleichungssystemen, die sich recht einfach numerisch lösen lassen, solange sie lösbar sind, was man als „gute Konditionszahl" lesen sollte.

Kommen wir als nächstes zu den überbestimmten Systemen, also Matrizen $A \in \mathbb{R}^{m\times n}$, $m > n$. Hier spielt die rechte Seite eine Rolle, wenn die „gut" ist, dann erhalten wir auch das korrekte Ergebnis:

```
octave> A = [ 1 2; 3 4; 5 6 ]; y = [ 1.5; 3.5; 5.5]; A\y
ans =

   0.50000
   0.50000
```

was sich auch mit kleinen Störungen nicht wesentlich ändert:

```
octave> y = y+10*eps*rand(3,1); A\y
ans =

   0.50000
   0.50000

octave> ans-[.5;.5]
ans =

  -9.4369e-16
   9.9920e-16
```

Man sollte hier schon bedenken, daß das Gleichungssystem $Ax = y$ nicht mehr exakt lösbar ist. Was machen `Matlab` und `Octave` hier? Anstatt das System

$$Ax = y \qquad \Leftrightarrow \qquad Ax - y = 0$$

wird der Fehler in der Norm minimiert:

$$\min_x \|Ax - y\|_2, \qquad \|x\|_2 := \sqrt{x^T x} = \sqrt{\sum_{j=1}^{n} x_j^2}. \tag{2.2}$$

Dieser Prozess ist recht unempfindlich gegenüber kleinen Störungen und liefert ebenfalls wieder ein lineares Gleichungssystem, das man numerisch stabil lösen kann, allerdings dann mit einer QR–Zerlegung, siehe Theorie, allerdings auch nur, wenn die Matrix A den vollen **Rang** n hat, sonst landet man bei exakt denselben Schwierigkeiten wie bei quadratischen Matrizen.

Bei unterbestimmten Gleichungssystemen mit $A \in \mathbb{R}^{m\times n}$, $m < n$ hat man hingegen die Qual der Wahl und muss sich aus den vielen möglichen Lösungen eine auswählen, vorausgesetzt natürlich, daß die Matrix A den Rang m hat. Mit unserer Matrix A aus dem überbestimmten Fall erhalten wir beispielsweise

```
octave:7> A'\eye(2)
ans =

  -1.33333   1.08333
  -0.33333   0.33333
   0.66667  -0.41667
```

Aber wie findet man diese Lösung? Man sucht sich unter allen möglichen Lösungen die „einfachste“ aus, in diesem Fall die mit kleinster Norm:

$$\min \|x\|_2, \qquad Ax = y. \tag{2.3}$$

Nach (2.11) ist die Lösung hiervon[18]

```
octave> ([ eye(3), A; A', zeros(2) ] \ [ zeros( 3,2 ); eye(2) ])(1:3,:)
ans =

  -1.33333   1.08333
  -0.33333   0.33333
   0.66667  -0.41667
```

was übrigens in der Tat eine **Rechtsinverse** von A ist:

```
octave> A'*ans
ans =

   1.00000  -0.00000
   0.00000   1.00000
```

Von links funktioniert die Matrix schon aus Dimensionsgründen nicht. Die Lösungen zu unserer Matrix A sind **Pseudoinverse** aus (2.14) zu A:

```
octave> pinv(A)
ans =

  -1.33333  -0.33333   0.66667
   1.08333   0.33333  -0.41667

octave> pinv(A')
ans =

  -1.33333   1.08333
  -0.33333   0.33333
   0.66667  -0.41667
```

Pseudoinverse funktionieren gut, im Falle von singulären oder fast singulären Matrizen aber nicht wirklich:

[18] Mal in ganz leicht fortgeschrittener `Octave`–Notation.

```
octave> A = [ 1 1 ; 1 1 ]; pinv(A)
ans =

   0.25000   0.25000
   0.25000   0.25000
```

Aber das Ergebnis kommt uns bekannt vor! Und in der Tat:

```
octave> pinv(A) - A\eye(2)
warning: matrix singular to machine precision, rcond = 0
ans =

   0   0
   0   0
```

bei `Matlab` ist das Ergebnis offensichtlich anders.

2.2.2 Lineare Gleichungssysteme: Die Theorie

Ein **lineares Gleichungssystem** ist ein System von m *linearen* Gleichungen in n Unbekannten,

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= y_1, \\ &\vdots \qquad\qquad m, n \in \mathbb{N}, \\ a_{m1}x_1 + \cdots + a_{mn}x_n &= y_m, \end{aligned} \tag{2.4}$$

das sich natürlich viel kompakter in Matrixform

$$Ax = y, \qquad A \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^n,\ y \in \mathbb{R}^m, \tag{2.5}$$

schreiben lässt. Aber wie löst man nun lineare Gleichungssysteme? Dazu erst einmal eine kleine Klassifikation.

Definition 2.1 *Ein lineares Gleichungssystem $Ax = y$, $A \in \mathbb{R}^{m\times n}$ heißt*

1. ***überbestimmt**, wenn $m > n$ ist,*
2. ***quadratisch**, wenn $m = n$ ist,*
3. ***unterbestimmt**, wenn $m < b$ ist.*

*Ist ein Gleichungssystem für jede rechte Seite **eindeutig lösbar**, so muss es quadratisch sein*[19]

Fangen wir also erst einmal mit quadratischen Systemen an. Von **Gauß–Elimination** hat man möglicherweise schon in der Schule oder sonstwo[20] gehört, in der „richtigen“ Numerik geht es aber ein bisschen anders, man arbeitet dort mit **Matrix–Zerlegungen**.

[19] Weiss man aus der Linearen Algebra, (Brieskorn, 1983; Fischer, 1984).
[20] Lineare Algebra?

Beispiel 2.2 (Einfache Gleichungssysteme) *Sortieren wir doch einmal leicht zu lösende quadratische Gleichungssysteme* $Ax = y$ *bezüglich einfacher Matrizen* $A \in \mathbb{R}^{n\times n}$.

1. $A = I$. *Müssen wir hier überhaupt darüber diskutieren? Das Gleichungssystem ist* $x = y$, *woraus man* x *mit etwas Arbeit bestimmen können sollte.*

2. A *ist* ***diagonal****, d.h.*

$$A = \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{nn} \end{bmatrix} \qquad \Leftrightarrow \qquad a_{jj}\, x_j = y_j, \quad j = 1, \ldots, n.$$

   Das Gleichungssystem ist genau dann lösbar[21]*, wenn* $a_{jj} \neq 0$, $j = 1, \ldots, n$, *ist und da die Variablen* ***entkoppelt*** *sind, löst man einfach die einzelnen Gleichungen* $x_j = \frac{y_j}{a_{jj}}$.

3. A *ist eine* ***obere Dreiecksmatrix****, d.h.,*

$$A = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ & \ddots & \vdots \\ & & a_{nn} \end{bmatrix}.$$

   Diese Gleichungssysteme kann man durch ***Rücksubstitution*** *direkt lösen, indem man die letzte Gleichung*

$$a_{nn}x_n = y_n \qquad \Rightarrow \qquad x_n = \frac{1}{a_{nn}} y_n$$

   löst, dann das Ergebnis in die vorletzte einsetzt,

$$a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = y_{n-1} \qquad \Rightarrow \qquad x_{n-1} = \frac{1}{a_{n-1,n-1}} \left(y_{n-1} - a_{n-1,n}x_n\right)$$

   und durch Fortführung dieser Idee auf

$$x_j = \frac{1}{a_{jj}} \left(y_j - \sum_{k=j+1}^{n} x_k \right), \qquad j = n, n-1, \ldots, 1 \tag{2.6}$$

   kommt.

4. A *ist* ***orthogonal****, d.h.* $A^T A = AA^T = I$, *die Zeilen- bzw. Spaltenvektoren der Matrix sind also* ***orthonormal***[22]*. Dann löst man* $Ax = y$ *durch* $x = A^T y$.

Um zu zeigen, wie gut Matlab als Sprache ist, können wir die Iteration (2.6) als Einzeiler

[21] Unabhängig von der rechten Seite.
[22] Der Sprachgebrauch ist hier etwas ungenau, das ist richtig.

```
%% Ruecksubs.m
%% Loesen von Dreiecksmatrizen
%%
%% U = Obere Dreiecksmatrix
%% Y = rechte Seite

function X = Ruecksubs( U,Y )
  [m,n] = size( U );
  X = Y( n,: ) / U(n,n);
  for j=n-1:-1:1
    X = [ ( Y(j,:) - U( j,j+1:n )*X ) / U(j,j); X ];
  end
end
```

Programm 2.4: Die **Rücksubstitution** in knapper `Matlab`–Notation. Eigentlich alles ganz einfach.

```
x = [ ( y(j) - A( j,j+1:n )*x ) / A(j,j); x ];
```

schreiben, siehe Programm 2.4. Die Übertragung von (2.6) auf eine **untere Dreiecksmatrix**

$$L = \begin{bmatrix} \ell_{11} & & \\ \vdots & \ddots & \\ \ell_{n1} & \dots & \ell_{nn} \end{bmatrix}$$

ist entweder klar oder eine schöne und empfehlenswerte Übung. Wir verwenden ab sofort die folgenden Buchstaben:

U,R: obere Dreiecksmatrix (bzw. **Rechtsdreiecksmatrix**),

L: untere Dreiecksmatrix,

Q: orthogonale Matrix,

P: **Permutationsmatrix**, also eine Matrix, die in jeder Zeile und jeder Spalte genau einen von Null verschiedenen Wert hat.

Das Lösen von Gleichungssystemen führt man dann auf Matrixzerlegungen zurück.

Satz 2.3 *Ist $A \in \mathbb{R}^{n\times n}$ eine **invertierbare Matrix**, so kann sie in*

$$PA = LU, \qquad \Rightarrow \qquad A = PLU \tag{2.7}$$

und

$$A = QR \tag{2.8}$$

zerlegt werden.

Für die LU–Zerlegung benötigt man die Permutationsmatrix P, das ist die sogenannte **Spaltenpivotsuche**, siehe (Golub & van Loan, 1996; Sauer, 2013), bei der QR–Zerlegung kann man sie hinzufügen, muss aber nicht, wobei man dann allerdings „von rechts“, also die Spalten permutieren muss.

Übung 2.2 Zeigen Sie:

1. PA liefert eine Vertauschung der *Zeilen*, AP eine Vertauschung der *Spalten* von A.
2. Die Matrix
$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$
ist invertierbar, besitzt aber *keine* Zerlegung $A = LU$.
3. Eine QR–Zerlegung von PA bringt keinen zusätzlichen Gewinn.

◇

Die Zerlegungen kann man mit den Funktionen `lu` und `qr` auch abfragen:

```
octave> A= [ 1 2 ; 3 4 ]; [L,U] = lu(A)
L =

   0.33333   1.00000
   1.00000   0.00000

U =

   3.00000   4.00000
   0.00000   0.66667
```

Hier fallen zwei Dinge auf: Bei beiden Matrizen ist offenbar durch 3 geteilt worden, was das plötzliche Auftreten von Rundungsfehlern erklären kann, und vor allem ist L keine untere Dreiecksmatrix. Ein erweiterter Aufruf bringt die Lösung:

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

   1.00000   0.00000
   0.33333   1.00000

U =

   3.00000   4.00000
   0.00000   0.66667

P =
```

```
Permutation Matrix

   0   1
   1   0
```

Man sieht hier übrigens, daß Permutationsmatrizen, zumindest in `Octave`, ein eigener Datentyp sind. `qr` ist da einfacher

```
octave> [Q,R] = qr(A)
Q =

  -0.31623  -0.94868
  -0.94868   0.31623

R =

  -3.16228  -4.42719
   0.00000  -0.63246
```

und daß Q orthogonal ist sieht man direkt.

Das normale Procedere beim Lösen eines linearen Gleichungssystems besteht in einer LU–Zerlegung $PA = LU$ der Matrix A, wobei die Matrix U auf der Diagonalen immer den Wert 1 hat und $|\ell_{jk}| \leq 1$ ist. Letzteres definiert die Permutationsmatrix P, man spricht hier von **Spaltenpivotsuche**, siehe (Golub & van Loan, 1996; Higham, 2002; Sauer, 2013), die auch numerisch ausgesprochen sinnvoll ist. Dann löst man die beiden Dreieckssysteme

$$y' = Uy, \qquad y'' = Ly'$$

und setzt $x = P^{-1}y'' = P^{T}y''$, wobei man bei Permutationen nie wirklich eine Matrix–Vektor–Multiplikation rechnet, sondern lediglich die Komponenten des Vektors vertauscht, weswegen es sinnvoll ist, Permutationen als eigenen Datentyp zu definieren.

Das erklärt auch das seltsame Verhalten der Inversen aus dem Beispiel $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Die Spaltenpivotsuche vertauscht die beiden Zeilen und man muss bei der **Gauß–Elimination**, die hierbei verwendet wird, durch 3 teilen, was zu Rundungsfehlern führt.
Da Matrizen einen Vektorraum bilden, also vernünftig addiert und mit Zahlen multipliziert werden können, können wir auch Normen für Matrizen definieren und so die Größe von Matrizen messen.

Definition 2.4 (Konditionszahl)

1. *Eine **Matrixnorm** heißt **submultiplikativ**, wenn*

$$\|AB\| \leq \|A\| \, \|B\|$$

für alle A, B *erfüllt ist.*

2. *Die* ***Konditionszahl*** *einer invertierbaren Matrix A zur Norm $\|\cdot\|$ ist definiert als*

$$\kappa(A) := \|A\|\,\|A^{-1}\|. \tag{2.9}$$

Übung 2.3 Zeigen Sie: Es gibt keine *multiplikative* Matrixnorm, d.h. keine Matrixnorm mit $\|AB\| = \|A\|\,\|B\|$. **Hinweis:** Überlegen Sie, welches Normaxiom verletzt werden könnte. ◊

Ist die Matrixnorm submultiplikativ[23] und so normiert, daß $\|I\| = 1$, dann gilt

$$\kappa(A) = \|A\|\,\|A^{-1}\| \geq \|AA^{-1}\| = \|I\| = 1,$$

die bestmögliche Konditionszahl ist also 1. Ansonsten ist die Konditionszahl ein Maß für die Invertierbarkeit von A, je größer die Konditionszahl wird, desto schlechter kann die Matrix invertiert werden. Genauer sagt's uns der folgende, sehr einfach zu beweisende Satz.

Satz 2.5 *Ist A_n, $n \in \mathbb{N}$, eine Folge von invertierbaren Matrizen, die gegen die singuläre*[24] *Matrix A konvergiert, dann ist*

$$\lim_{n\to\infty} \kappa(A_n) = \infty.$$

Beweis: Wären $\kappa(A_n)$ beschränkt, dann wären auch die Folgen $\|A_n\|$ und $\|A_n^{-1}\|$ beschränkt und die Folge A_n^{-1} müsste als beschränkte Folge eine konvergente Teilfolge enthalten[25], für die dann aber die A_n gegen eine invertierbare Matrix konvergieren. Das ist auch schon der Widerspruch zur Annahme. □

Für die überbestimmten Systeme müssen wir die Funktion[26]

$$F(x) = \|Ax - y\|_2^2 = (Ax - y)^T(Ax - y) = x^TA^TAx - 2y^TAx + y^Ty$$

minimieren, was mit der Standardmethode „Ableiten und gleich Null setzen" gemacht wird, siehe (Forster, 1984; Sauer, 2015). Dazu bildet man den Gradienten bezüglich x und setzt ihn gleich Null:

$$0 = \nabla F(x) = 2A^TAx - 2A^Ty \qquad \Leftrightarrow \qquad A^TAx = A^Ty. \tag{2.10}$$

Hat A maximalen Rang, dann ist A^TA eine symmetrische, (strikt) positiv definite Matrix und die **Normalengleichungen** (2.10) haben eine eindeutige Lösung. In Wirklichkeit löst man aber nicht (2.10), denn $\kappa(A^TA) \approx \kappa(A)^2$ und das ist nicht wirklich erwünscht.

Für die Lösung des Minimierungsproblems (2.3) verwendet man **Lagrange–Multiplikatoren**, siehe (Heuser, 1983; Sauer, 2015) und sucht einen Vektor $\lambda \in \mathbb{R}^3$, so daß

$$\begin{aligned} 0 &= x - A^T\lambda \\ y &= Ax \end{aligned}$$

[23] Was nach Übung 2.3 das beste ist, was eine Matrixnorm leisten kann.

[24] Also nicht invertierbare.

[25] Das gilt generell in metrischen Räumen, siehe (Sauer, 2015).

[26] Wir quadrieren, um die Wurzel loszuwerden, die sonst nur stören würde. Da $\sqrt{} : \mathbb{R}_+ \to \mathbb{R}_+$ eine **monotone Funktion** ist, macht das keinen Unterschied bezüglich der Lage des Maximums.

was man zu dem Gleichungssystem

$$\begin{bmatrix} I & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ -\lambda \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix} \tag{2.11}$$

anordnen kann, von dessen Lösung uns die Multiplikatoren λ aber nicht interessieren. Allerdings hiesse das ja immer, daß wie ein $m \times n$–Problem lösen würden, indem wir es erst einmal zu einem $(n+m) \times (n+m)$–Problem aufblasen, was bei einer Lösungskomplexität von $O(n^3)$ für eine $n \times n$–Matrix recht ineffizient wäre. Hier verwendet man eine andere Zerlegung.

Satz 2.6 (SVD) *Jede Matrix $A \in \mathbb{R}^{m \times n}$ hat eine **singuläre Werte–Zerlegung** bzw. **SVD**[27]*

$$A = U \Sigma V^T, \qquad U \in \mathbb{R}^{m \times m},\ V \in \mathbb{R}^{n \times n},\ \Sigma \in \mathbb{R}^{m \times n}, \tag{2.12}$$

mit orthogonalen *Matrizen $U^T U = I$, $V^T V = I$ und einer **Diagonalmatrix** Σ mit*

$$\sigma_{11} \geq \sigma_{22} \geq \cdots \geq \sigma_{kk} \geq 0, \qquad k = \min(m, n), \tag{2.13}$$

und allen anderen Komponenten gleich Null.

Der Eintrag $\sigma_j = \sigma_{jj}$, $j = 1, \ldots, \min(m, n)$, heisst jter **Singulärwert** von A. Die Anzahl der positiven Singulärwerte entspricht dem Rang von A. Die SVD kann man einfach ausrechnen lassen:

```
octave> [U,S,V] = svd( [ 1 1; 1 1 ] )
U =

  -0.70711  -0.70711
  -0.70711   0.70711

S =

Diagonal Matrix

   2   0
   0   0

V =

  -0.70711  -0.70711
  -0.70711   0.70711
```

Definiert man nun für $A \in \mathbb{R}^{m \times n}$ die Matrix

$$A^\dagger := \begin{cases} a_{jk}^{-1}, & a_{jk} \neq 0, \\ 0, & a_{jk} = 0, \end{cases}$$

[27] Für **S**ingular **V**alue **D**ecomposition.

dann ist die **Pseudoinverse** A^+ zu einer Matrix $A = U\Sigma V^T$ als

$$A^+ = V\Sigma^\dagger U^T \tag{2.14}$$

definiert. In `Matlab`/`Octave` liefert `pinv` die Pseudoinverse. Ist $A \in \mathbb{R}^{n\times n}$ quadratisch und invertierbar, dann ist dies äquivalent dazu, daß $\sigma_j > 0$, $j = 1, \ldots, n$, also

$$\Sigma^\dagger = \begin{bmatrix} \sigma_1^{-1} & & \\ & \ddots & \\ & & \sigma_n^{-1} \end{bmatrix} = \Sigma^{-1}$$

und daher

$$A^+A = V\Sigma^\dagger \underbrace{U^T U}_{=I} \Sigma V^T = V \underbrace{\Sigma^{-1}\Sigma}_{=I} V^T = VV^T = I,$$

also $A^+ = A^{-1}$, was den Namen erklärt.

Übung 2.4 Zeigen Sie: Hat $A \in \mathbb{R}^{m\times n}$ maximalen Rang, dann ist A^+ eine **Linksinverse** falls $m \le n$ ist und eine **Rechtsinverse**, falls $m \ge n$ ist. ◊

Die Pseudoinverse ist bei Ingenieuren beliebt, weil sie eine „Inverse" ist, die immer und überall „funktioniert" und nie eine Warnung bringt oder `Inf` oder `NaN` liefert. Allerdings ist ihr Verhalten bei singulären oder fast singulären Matrizen ebenfalls oftmals ein wenig erratisch. Die Ursache des Problems liegt darin, daß in der Definition der Pseudoinversen eine Abfrage auf $= 0$ enthalten ist, was aber in numerischer Rechnung und bei Rundungsfehlern eigentlich nicht sinnvoll ist. Daher muss bei der `pinv` eine Schwelle festgelegt werden und diese kann, muss aber nicht klappen.

2.3 Funktionen, Interpolation und Approximation

In diesem Abschnitt befassen wir uns mit der Interpolation und Approximation von und durch Funktionen und gleichzeitig auch damit, wie man Funktionen in einer und zwei Variablen in `Matlab` plotten und manipulieren kann. Gerade in diesem Punkt, also bei der Grafik, gibt es sicherlich die größten Unterschiede zwischen `Matlab` und `Octave`.

2.3.1 Funktionen und Plots in `Matlab`

Beginnen wir mit einem ganz einfachen Beispiel, nämlich einem Plot des Cosinus[28]. Dazu müssen wir die Funktion erst einmal abtasten:

```
octave> y = cos( (0:.1:10) );
```

Das tastet den Cosinus auf dem Intervall $[0, 10]$ mit Schrittweise 0.1 ab und liefert uns, wie wir bereits gelernt haben, einen **Zeilenvektor** der Länge 101:

```
octave> size(y)
ans =

     1   101
```

[28] Da wissen wir dann wenigstens, wie das Ergebnis aussehen muss.

Den plotten wir nun einfach mit der Funktion `plot`:

```
octave> plot(y)
```

Wenn wir alles richtig gemacht haben, dann öffnet sich ein Fenster mit dem Plot der Cosinusfunktion. Die sind, wie man in Abb. 2.1 sehen kann, durchaus

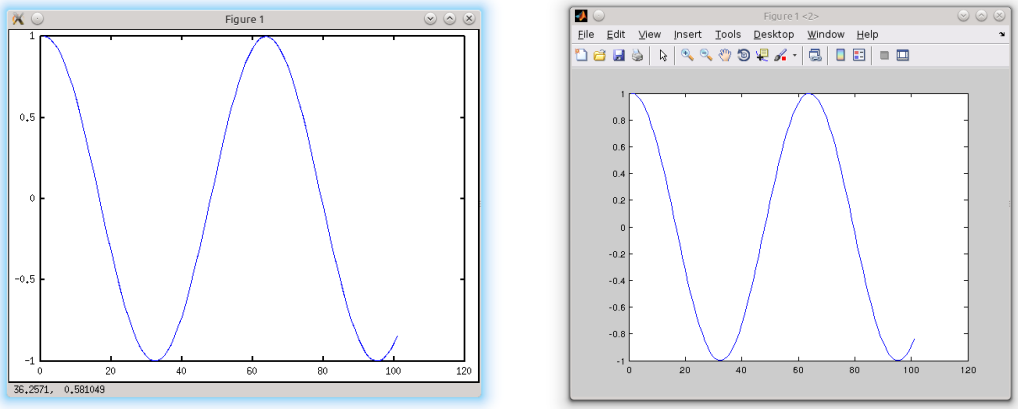


Abbildung 2.1: Plots der Cosinus–Funktion in `Octave` *(links)* und `Matlab` *(rechts)*. Man kann deutlich erkennen, daß es in Matlab deutlich mehr Möglichkeiten gibt, mit dem Plot zu spielen.

unterschiedlich, da `Matlab` deutlich mehr Möglichkeiten bietet, während sich `Octave` im wesentlichen auf eine „Weiterleitung“ an `gnuplot` beschränkt[29]. Die grundsätzliche Funktionalität ist allerdings identisch. Die `plot`–Funktion plottet also einen Vektor, wobei die y–Achse bezüglich dessen Werten skaliert ist, auf der x–Achse hingegen werden die Indizes der Vektorkomponenten aufgetragen. Das kann man natürlich ändern, indem man `Matlab` auch die x–Wert verrät. Dies geht beispielsweise folgendermaßen:

```
octave> x = ( 0:.01:10 ); y = cos(x); plot( x',y' )
```

Die Transposition benötigt man, da `Matlab` die *Spalten* gegeneinander plottet – das ist einfach Konvention. Man kann die beiden Vektoren auch in eine Matrix schreiben, aber dann passiert etwas anderes:

```
octave> plot( [x',y'] )
```

Will man mehrere Plots gleichzeitig haben, so kann man die y–Werte auch in einer Matrix anordnen:

```
octave> x = (0:.1:10)'; plot( x, [ cos(x),sin(x) ] )
```

Noch ein paar wichtige Hilfsmittel:

[29] Allerdings wird gerade an der Grafik aktuell (04/2015) sehr viel gearbeitet.

1. Mit `figure()` kann man ein neues Fenster öffnen, mit `figure(k)` auf Fenster `k` umschalten.

2. Mit `hold on` kann man eine neue Grafik in das alte Fenster plotten lassen:

   ```
   octave> figure(); hold on; plot( x,cos(x) ); plot( x,sin(x) );
   ```

   Mit `hold off` schaltet man das wieder aus.

3. Mit `clf` schliesst man alle Plots.

4. Mit `plot(...,"*")` kann man die Werte auf verschiedene Art und Weise als Punkte plotten lassen:

   ```
   octave> clf; hold on; plot( x,cos(x) );
   octave> plot( (0:10)',cos((0:10)'), "*" )
   ```

5. Mit `linspace` kann man sich ein Intervall in eine vorgegebene Anzahl von Punkten zerlegen lassen, was ganz praktisch ist, wenn man den Bereich $[a, b]$ beispielsweise in 1000 Teile zerlegen lassen möchte:

   ```
   octave> linspace(0,1,4)
   ans =

      0.00000   0.33333   0.66667   1.00000
   ```

   Man gibt zuerst den Bereich vor und dann die Anzahl an gleichverteilten Punkten, die man haben möchte. Nützlicher ist das in Anwendungen wie

   ```
   octave> linspace(0,pi,4)
   ans =

      0.00000   1.04720   2.09440   3.14159
   ```

Was in 1D geht, geht auch in 2D, also mit Flächen. Zu diesem Zweck muss man die z–Werte in einer Matrix anordnen, aber zuerst einmal die x- und y–Werte dazu bestimmen. Dazu ist es gut, sich an die Regel

$$xy^T = \begin{bmatrix} x_1y_1 & \dots & x_1y_n \\ \vdots & \ddots & \vdots \\ x_ny_1 & \dots & x_ny_n \end{bmatrix} \in \mathbb{R}^{n\times n}, \qquad x, y \in \mathbb{R}^n,$$

zu erinnern, mit deren Hilfe man **separable Funktionen** sehr einfach plotten kann, beispielsweise $f(x, y) = \sin(x)\cos(y)$. Die Ergebnisse sind in Abb 2.2 zu sehen, dabei ist der y–Wert farbcodiert. In `Matlab` gibt es ziemlich gute Beleuchtungseffekte, mit denen man Flächen recht gut aussehen lassen kann, siehe Abb. Tatsächlich ist das nicht einmal schwer:

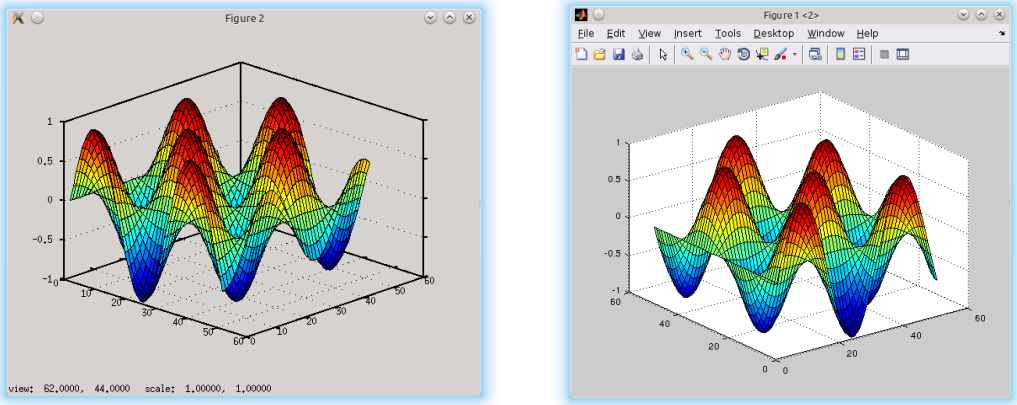


Abbildung 2.2: Plots der Funktion $\cos x \sin y$ in `Octave` *(links)* und `Matlab` *(rechts)*. Sieht erst mal noch alles ziemlich ähnlich aus.

```
>> x = (0:0.1:10)';surface( cos(x)*sin(x)' );
>> light( 'Position', [15,15,7] );
>> shading interp
```

Suchen wir uns mal ein etwas spannenderes Beispiel, nämlich die Funktion

$$f(x,y) = \begin{cases} \dfrac{xy^2}{x^2+y^2}, & (x,y) \neq (0,0), \\ 0, & (x,y) = (0,0), \end{cases} \tag{2.15}$$

die ein typisches Beispiel für eine Funktion ist, die im Ursprung in jede Richtung **richtungsdifferenzierbar** ist, aber dort nicht differenzierbar ist, siehe (Sauer, 2015). Bauen wir uns diese Funktion einmal zusammen. Wir benötigen die beiden Funktionen xy^2 und x^2+y^2. Die erste ist separabel und wir können sie als

```
>> x = linspace( -1,1,500 )';
>> zlr = x * (x.^2 )';
```

konstruieren. Beim Nenner haben wir es mit $(x,y) \mapsto x^2+y^2$ zu tun und dafür müssen wir die Funktione x^2 und y^2 separat behandeln:

```
>> nnr = ( x.^2 )*ones( 1,length(x) ) + ones( length(x),1 )*(x.^2)';
```

Jetzt brauchen wir nur noch komponentenweise zu dividieren, dabei aufpassen, daß wir nicht durch Null dividieren, und dann das Ganze zu plotten:

```
>> f = zlr ./ ( nnr + ( nnr == 0 ) );
>> surface( x,x,f ); light( 'Position', [2,2,5] ); shading interp
```

Das Ergebnis in Abb 2.5 kann sich dann durchaus sehen lassen. Beleuchtungseffekte sind übrigens nicht nur ästhetischer Selbstzweck, sie helfen vor allem dabei, das Krümmungsverhalten von Objekten besser zu erkennen.

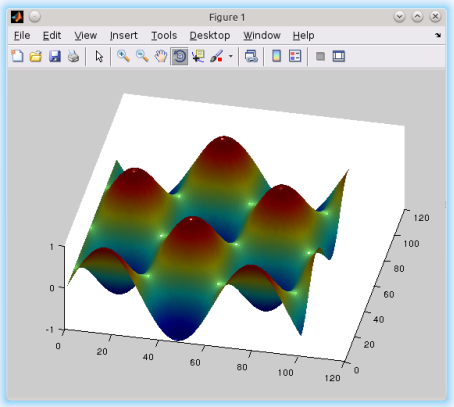


Abbildung 2.3: Die Funktion $\cos x \sin y$ mit Beleuchtungseffekten.

2.3.2 Interpolation aus linearen Funktionenräumen

Als nächstes befassen wir uns mit der Frage der **Interpolation**, also der Rekonstruktion von Funktionen aus Datenwerten. Zu **Stützstellen** $x_1, \ldots, x_n$ und vorgegebenen Werten $y_1, \ldots, y_n$ suchen wir eine Funktion f mit der Eigenschaft

$$f(x_j) = y_j, \qquad j = 1, \ldots, n. \tag{2.16}$$

Die Funktion f heißt **Interpolant** an die Werte y_j an den Stellen x_j. Natürlich gibt es beliebig viele Funktionen, die diese Eigenschaft haben können und die Lösung von (2.16) ist in keinster Weise eindeutig. Ausserdem würden wir die Funktion f gerne auch am Rechner beschreiben können.

Definition 2.7 *Ein **linearer Funktionenraum** zu Funktionen $f_1, \ldots, f_m$, $m \in \mathbb{N}$, ist der Vektorraum aller **Linearkombinationen***

$$f = \sum_{j=1}^{m} a_j f_j = f_a, \qquad a_j \in \mathbb{R}, \quad j = 1, \ldots, m,$$

*wobei der **Koeffizientenvektor** $a = (a_1, \ldots, a_m)$ die Funktion f_a komplett beschreibt, wenn die Funktionen bekannt sind. Der Einfachheit halber nehmen wir an, daß die Funktionen alle **linear unabhängig** sind.*

Übung 2.5 Zeigen Sie, daß $\mathscr{F} := \{f_a : a \in \mathbb{R}^m\}$ ein **Vektorraum** ist. Was ist die Dimension dieses Vektorraums? ◊

Mit ein bisschen `Matlab`–Denkweise können wir das Problem nun angehen: Für $j = 1, \ldots, n$ liefert uns (2.16) die Bedingung

$$y_j = f_a(x_j) = \sum_{k=1}^{m} f_k(x_j)\, a_k = \left(\left[f_k(x_j) : \begin{array}{c} j = 1, \ldots, n \\ k = 1, \ldots, m \end{array} \right] a \right)_j,$$

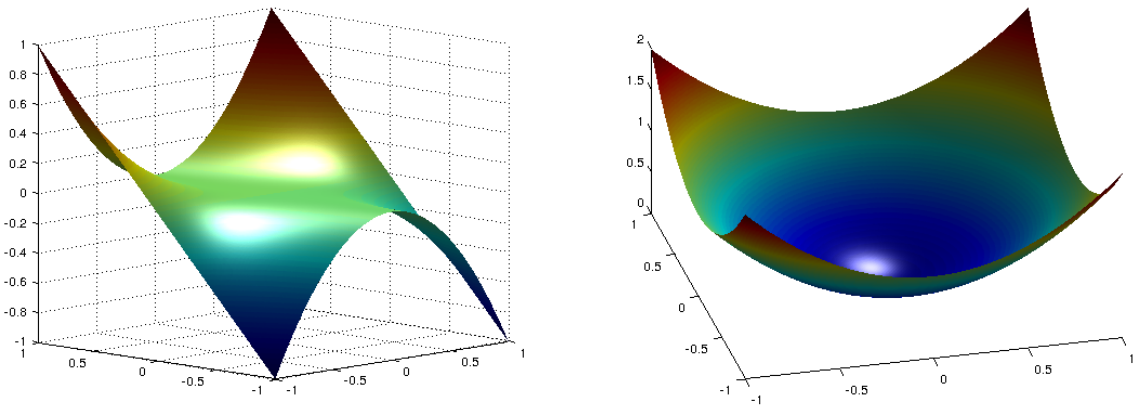


Abbildung 2.4: Zähler und Nenner der Funktion aus (2.15).

also

$$Y = \underbrace{\left[f_k(x_j) : \begin{array}{c} j = 1, \ldots, n \\ k = 1, \ldots, m \end{array} \right]}_{=:F(X)} a, \qquad F = (f_1, \ldots, f_m), \quad X = (x_1, \ldots, x_n). \tag{2.17}$$

Definition 2.8 *Die Matrix* $F(X)$ *heißt* ***Kollokationsmatrix*** *des Interpolationsproblems* (2.16) *zur Basis* F.

Die Lösung eines Interpolationsproblems ist nun einfach: Wir erstellen die Kollokationsmatrix und lösen das System. Erstes Beispiel: Wir interpolieren mit x^j, $j = 0, \ldots, n-1$, an gleichverteilten Stellen in $[0, 1]$:

```
octave> x = linspace( 0,1,7 )';
```

Die Basisfunktionen bilden die **Spalten** der Matrix, also können wir die Kollokationsmatrix mit der einfachen Schleife

```
octave> F = []; for j=1:7 F = [ F (x).^(j-1) ]; end
```

bilden. Und[30] diese Matrix können wir, wenn wir wollen, uns auch als Fläche ansehen:

```
octave> surface( F );
```

Jetzt brauchen wir noch eine rechte Seite, die wir einfach zufällig wählen und können unser Problem lösen:

```
octave> y = rand( 7,1 ); a = F\y;
```

[30] Because we can.

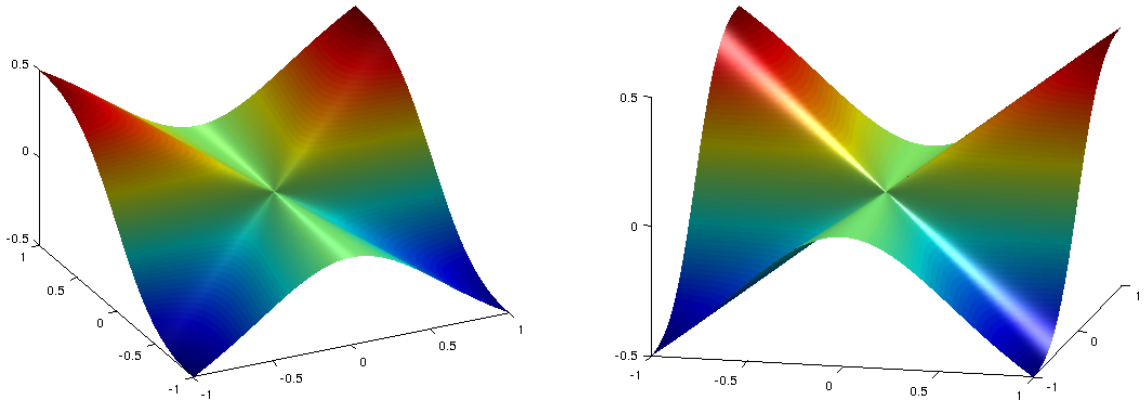


Abbildung 2.5: Die Funktion (2.15) aus zwei verschiedenen Blickwinkeln mit Beleuchtungseffekten, die schon hilfreich sind, wenn man wirklich sehen will, was da passiert.

Jetzt würden wir unsere Funktion gerne plotten, also die Funktionswerte

$$f(\xi_j) = \sum_{k=1}^{m} f_k(\xi_j)\, a_k = \left(F(\Xi)\, a\right)_j, j = 1, \ldots, N,$$

ausgeben, was wieder eine Kollokationsmatrix ist. Also dasselbe Spiel nochmal:

```
octave> X = linspace( 0,1,1000 )';
octave> G = []; for j=1:7 G = [ G, X.^(j-1) ]; end
```

Und das können wir dann recht entspannt plotten:

```
octave> clf; plot( X,G*a );
octave> hold on; plot( x,y,"o" )
```

Dabei trägt die zweite Zeile die zu interpolierenden Werte mit ein, der Interpolant interpoliert. Besonders nett sind die Koeffizienten zu $A = F(X)^{-1}$, also die Spalten $A(:,1), \ldots, A(:,n)$ dieser Matrix. Da[31]

$$F(X)\, A(:,j) = (F(X)A)(:,j) = \left(F(X)F(X)^{-1}\right)(:,j) = I(:,j) = e_j$$

erhält man so Funktionen, die an der j–ten Stützstelle den Wert 1 und sonst überall den Wert 0 haben. Plotten wir sie:

```
octave> clf; plot( X,G*inv(F) );
```

Das Leben kann in `Matlab` und `Octave` sehr einfach sein.

Noch einfacher geht **trigonometrische Interpolation** mit den Funktionen $\cos kx$, $k = 0, 1, \ldots, m-1$. Dann ergibt sich die Kollokationsmatrix[32] als

[31] Ja, man kann auch in `Matlab`–Notation mathematischen Formalismus betreiben.
[32] Nicht vergessen: Spalten sind die Funktionen.

```
octave> x = linspace( 0,pi,11 )'; F = cos( x*(0:10) );
```

und wir können die Elementarlösungen ebenfalls sehr einfach plotten:

```
octave> X = linspace( 0,pi,1000 )'; G = cos( X*(0:10) );
octave> plot( X,G*inv(F) )
```

oder auch als Fläche, dann aber besser in `Matlab`:

```
>> surface( G*inv(F) );
>> light( 'Position',[0,4,3] );
>> shading interp
```

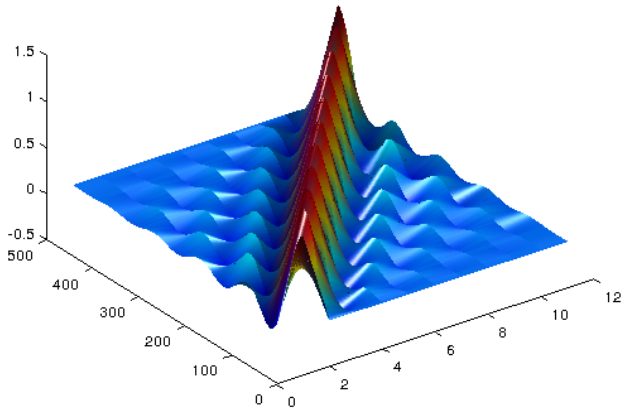


Abbildung 2.6: Die Elementarfunktionen des Cosinus als nettes dreidimensionales Bild mit ein bisschen Beleuchtung. Der inhaltliche Wert hält sich in Grenzen. .

Damit können wir auch schon für jeden linearen Raum interpolieren. Aber:

Achtung: Selbst wenn $m = n$ ist, so muss die Matrix $F(X)$ noch lange nicht invertierbar sein. Das hängt normalerweise sehr stark vom Zusammenspiel der Basisfunktionen und der Stützstellen ab.

Einen Funktionenraum gibt es, mit dem es *immer* gutgeht.

Satz 2.9 *An beliebigen, verschiedenen Punkte* $x_1, \ldots, x_n$ *kann man mit den Monomen* $1, x, \ldots, x^{n-1}$ *immer eindeutig interpolieren.*

Beweis: Interpolieren kann man, weil sich die **Elementarfunktionen**

$$\ell_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}, \qquad j = 1, \ldots, n, \tag{2.18}$$

alle als Linearkombinationen von $1, \ldots, x^{n-1}$ darstellen lassen[33] und die offensichtliche Eigenschaft

$$\sum_{k=1}^{n} y_k \underbrace{\ell_k(x_j)}_{=\delta_{jk}} = y_j$$

haben, also interpoliert $y_1\ell_1 + \cdots + y_n\ell_n$. Wären die Interpolanten nicht eindeutig, dann gäbe es zwei Interpolanten f, g, beide vom Grad $\leq n-1$, deren Differenz an $x_1, \ldots, x_n$ verschwindet, also ein Vielfaches von $(x-x_1)\cdots(x-x_n)$ sein muss. Also:

$$f(x) - g(x) = q(x)\,(x-x_1)\cdots(x-x_n),$$

aber das Polynom auf der linken Seite hat Grad $\leq n-1$, das auf der rechten Seite Grad $\geq n$, es sei denn, $q = 0$. □

Warnung 2: In vielen Fällen, gerade bei Polynomen, kann die **Konditionszahl** der **Kollokationsmatrix** sehr unerfreulich werden.

Übung 2.6 Berechnen Sie das Interpolationspolynom vom Grad 20 an gleichverteilten Punkten. ◇

2.3.3 Approximation und optimale Interpolation

Im letzten Kapitel haben wir stillschweigend angenommen, daß die Kollokationsmatrix quadratisch ist, indem wir einfach dafür gesort haben, daß die Zahl der **Bedingungen**[34] und der **Freiheitsgrade**[35] übereinstimmen. Aber natürlich kann das Gleichungssystem auch **überbestimmt** oder **unterbestimmt** sein, wenn wir zu viele oder zu wenige Punkte haben. Beginnen wir mit dem ersten Fall und tun einfach das, was uns der Befehl `F\y` suggeriert, nämlich eine **Least–Squares–Lösung**. Anstelle der exakten Forderung

$$f(x_j) = y_j \qquad \Leftrightarrow \qquad f(x_j) - y_j = 0 \qquad \Leftrightarrow \qquad \sum_{j=1}^{n} (f(x_j) - y_j)^2 = 0$$

minimieren wir wieder den letzten Ausdruck

$$\|f(X) - Y\|_2^2 := \sum_{j=1}^{n} (f(x_j) - y_j)^2 = 0. \tag{2.19}$$

In der `Matlab`–Routine sehen wir keinen Unterschied, wenn wir mit den ersten fünf Monomen an 15 Punkten „interpolieren“:

```
octave> x = linspace( 0,1,15 )';
octave> F = []; for j=0:4 F = [ F, x.^j ]; end
octave> y = rand( 15,1 ); a = F\y;
```

[33] Es sind Polynome vom **Grad** $\leq n-1$.
[34] Die Anzahl der Gleichungen
[35] Die Anzahl der Koeffizienten für unsere Funktionen.

Was passiert, sehen wir, wenn wir das Ganze plotten:

```
octave> clf; hold on; plot( X,G*a ); plot( x,y,"o" );
```

Der wichtigste Fall einer solchen **Least–Squares–Approximation** an Daten ist die Approximation durch eine **lineare Funktion**, also eine Funktion der Form $f(x) = ax + b$. In der Statistik bezeichnet man das als **lineare Regression**, doch dazu später mehr im R–Teil.

Unterbestimmte Gleichungssysteme treten auf, wenn man beispielsweise ein **inverses Problem** betrachtet, bei dem man einen Mechanismus $x \mapsto Ax$, $A \in \mathbb{R}^{m\times n}$ mit vielen Steuergrößen x und wenigen Messungen $y = Ax$ zu verstehen versucht, was in vielen Fällen gar nicht möglich ist[36]. Hier betrachtet man wieder Minimierungen, gerne von der etwas allgemeineren Form

$$\min_x x^T Bx, \qquad Ax = y, \tag{2.20}$$

mit symmetrischem B was wie in (2.11) zu einem Gleichungssystem

$$\begin{bmatrix} B & A^T \\ A & 0 \end{bmatrix}\begin{bmatrix} x \\ -\lambda \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix} \tag{2.21}$$

führt. Mit $B = I$ ist es wieder das, was der \–Operator sowieso macht. Und so haben wir dann unser unterbestimmtes dejà vu:

```
octave> x = linspace( 0,1,4 )';
octave> F = []; for j=0:7 F = [ F, x.^j ]; end
octave> y = rand( 4,1 ); a = F\y;
```

Und der Plot ist auch nichts neues:

```
octave> X = linspace( 0,1,500 )';
octave> G = []; for j=0:7 G = [ G, X.^j ]; end
octave> clf; hold on; plot( X,G*a ); plot( x,y,"o" );
```

Das schreit nach einem Fazit.

> Interpolationsprobleme mit linearen Räumen können auf lineare Gleichungssysteme in `Matlab` zurückgeführt und gelöst werden. Dabei haben auch über- und unterbestimmte Probleme eine sinnvolle Interpretation.

2.3.4 Glättungsterme

Die Sache mit dem **Glattheitsterm** $x^T Bx$ in (2.20) ist eigentlich keine schlechte Idee: Unter allen Lösungen des Interpolationsproblems wählt man sich diejenige aus, die unter dem quadratischen **Funktional** $x^T Bx$ am „bravsten“ ist. Und

[36] In Bildverarbeitungsanwendungen ist oftmals $n \simeq m^2$ oder schlimmer.

ein bisschen Gutartigkeit könnte ja auch „normale“ Interpolation ganz gut vertragen. Daher betrachtet man oft die Minimierung von Ausdrücken der Form

$$G_\lambda(a) = \sum_{j=1}^{n} (f_a(x_j) - y_j)^2 + \lambda\, a^T B a, \qquad \lambda \geq 0. \tag{2.22}$$

Der erste Ausdruck auf der linken Seite beschreibt die **Datentreue** der resultierenden Funktion, der zweite die **Glattheit** der Funktion und λ balanciert zwischen diesen beiden widersprüchlichen Zielen: Mit $\lambda \to 0$ wird das Ergebnis zum Interpolanten[37], für $\lambda \to \infty$ werden die vorgegebenen Punkte ziemlich egal.

Um (2.22) zu lösen, schreiben wir es zuerst um,

$$\begin{aligned} G_\lambda(a) &= \sum_{j=1}^{n} \left\| F(X)a - Y \right\|_2^2 + \lambda\, a^T B a = (F(X)a - Y)^T (F(X)a - Y) + \lambda\, a^T B a \\ &= a^T F(X)^T F(X) a - 2a^T F(X)^T Y + Y^T Y + \lambda\, a^T B a \\ &= a^T \left(F(X)^T F(X) + \lambda\, B \right) a - 2a^T F(X)^T Y + Y^T Y \end{aligned}$$

und setzen die Ableitung

$$2\left(F(X)^T F(X) + \lambda\, B \right) a - 2F(X)^T Y$$

gleich Null, was zu dem linearen Gleichungssystem

$$F(X)^T Y = \left(F(X)^T F(X) + \lambda\, B \right) a =: A_\lambda a \tag{2.23}$$

führt, das wir wieder einfach lösen können. Zwei Bemerkungen:

1. Die Matrix A_λ ist **symmetrisch**, wenn B symmetrisch ist. Ist B symmetrisch und **positiv semidefinit**[38], dann gilt das auch für A_λ.

2. Ist B positiv semidefinit und gilt

   $$\{x : F(x)x = 0\} \cap \{x : Bx = 0\} = \{0\},$$

   dann ist A_λ sogar **positiv definit**[39] und damit invertierbar.

Im Normalfall führt diese Methode sogar zu einer numerischen Stabilisierung des Gleichungssystems, das für kleine Werte von λ trotzdem relativ ähnliche Lösungen liefert. Dies bezeichnet man als **Tychonov–Regularisierung**, der bekannteste Fall ist

$$Ax = b \qquad \to \qquad (A + \lambda I)x = b.$$

Sehen wir uns ein Beispiel an und kehren wieder zu unseren Polynomen auf $[0, 1]$ zurück. Als **Funktional** verwenden wir nun

$$p \mapsto \int_0^1 \left(p^{(k)}(x) \right)^2 \, dx,$$

[37] Man interessiert sich nur noch für die Datentreue.
[38] Das heisst $x^T Bx \geq 0$.
[39] Also $x^T Ax > 0$ für alle $x \neq 0$.

was für ein Polynom der Form

$$p(x) = \sum_{j=0}^{n} a_j x^j$$

dann[40]

$$\begin{aligned}
\int_0^1 \left(p^{(\ell)}(x)\right)^2 \, dx &= \int_0^1 \left(\frac{d^\ell}{dx^\ell} \sum_{j=0}^{n} a_j x^j \right)^2 \, dx \\
&= \int_0^1 \left(\sum_{j=\ell}^{n} a_j \frac{j!}{(j-\ell)!} x^{j-\ell} \right)^2 \, dx = \int_0^1 \sum_{j,k=\ell}^{n} \frac{j!k!}{(j-\ell)!(k-\ell)!} a_j a_k x^{j+k-2\ell} \, dx \\
&= \sum_{j,k=\ell}^{n} \frac{j!k!}{(j-\ell)!(k-\ell)!} \, a_j a_k \underbrace{\int_0^1 x^{j+k-2\ell} \, dx}_{=\frac{1}{j+k-2\ell+1}} \\
&= \sum_{j,k=\ell}^{n} \underbrace{\frac{j!k!}{(j-\ell)!(k-\ell)!(j+k-2\ell+1)}}_{=:b_{jk}} \, a_j a_k = a^T B a.
\end{aligned}$$

Auch hier gilt wieder: Wer einmal mit `Matlab` infiziert ist, sieht überall Matrizen. Die Matrix B ist symmetrisch und positiv semidefinit, da

$$a^T B a = \int_0^1 \left(p^{(\ell)}(x)\right)^2 \, dx \geq 0.$$

Übung 2.7 Bestimmen Sie die Glättungsmatrix für beliebige Funktionen und berechnen Sie ein Beispiel mit der Cosinusfunktion. ◊

Um die Matrix B aufzustellen betrachten setzen wir $j = \ell + j'$ und $k = \ell + k'$ und erhalten

$$b_{jk} = \frac{(j'+\ell)!(k'+\ell)!}{j'!k'!} \frac{1}{j'+k'+1}, \qquad j,k \geq \ell.$$

Die erste Matrix ist wieder ein Produkt der Form xx^T, die für $\ell = 2$ und $n = 7$ als

```
octave> v = factorial( 2 .+ (0:5 ) ) ./ factorial( 0:5 );
octave> B1 = v'*v;
```

berechnet werden kann und die zweite Matrix eine sogenannte **Hankel–Matrix**, also eine Matrix der Form $a_{jk} = f(j+k)$, die man durch

```
octave> B2 = hankel( 1./(1:6),1./(6:11) )
B2 =
```

[40] Elementare Rechnung ...

```
1.000000   0.500000   0.333333   0.250000   0.200000   0.166667
0.500000   0.333333   0.250000   0.200000   0.166667   0.142857
0.333333   0.250000   0.200000   0.166667   0.142857   0.125000
0.250000   0.200000   0.166667   0.142857   0.125000   0.111111
0.200000   0.166667   0.142857   0.125000   0.111111   0.100000
0.166667   0.142857   0.125000   0.111111   0.100000   0.090909
```

erhalten kann. Dabei gibt `hankel(x,y)` eine Hankel–Matrix, deren erste Zeile[41] durch `x` und deren letzte Zeile durch `y` gegeben ist. Das Ergebnis ist das komponentenweise Produkt der beiden Matrizen mit passenden Nullen:

```
octave>  B = [ zeros(2,8) ; zeros(6,2), B1 .* B2 ];
```

Jetzt brauchen wir nur noch Datenpunkte und die Kollokationsmatrix sowie eine rechte Seite `y`

```
octave> x = linspace( 0,1,10 )';
octave> F = []; for j=0:7 F = [ F,x.^j ]; end
octave> y = rand( 10,1 );
```

Lösen wir das Problem gleich für verschiendene Werte von λ:

```
octave> A = []; for j=-10:0 A = [ A,( G + 4^j * B )\(F'*y) ]; end
```

Und dann plotten wir den Spass wieder:

```
octave> X=linspace(0,1,1000);
octave> FX = []; for j=0:7 FX = [ FX,X.^j ]; end
octave> hold on; plot( X,FX*A ); plot( x,y,"o" );
```

Bemerkung 2.10 *Hätte man weniger Punkte als polynomiale Freiheitsgrade, dann nähert sich das Ergebnis sogar einem echten Interpolanten an.*

2.4 Integration und Optimierung

Jetzt befassen wir uns mit zwei Sachen auf einmal, die aber gar nicht so weit auseinanderliegen, nämlich der Integration und der Optimierung.

2.4.1 Funktionen

Wir wollen uns mit Funktionen befassen, was uns natürlich zuerst einmal zu der Frage bringt, wie man eine **Funktion** $f : X \to Y$ eigentlich darstellt. Dabei hilft uns der Operator `@`:

```
octave> f = @(x) x*cos(x);
octave> f(2)
ans = -0.83229
```

[41] Oder erste Spalte, ganz wie man will.

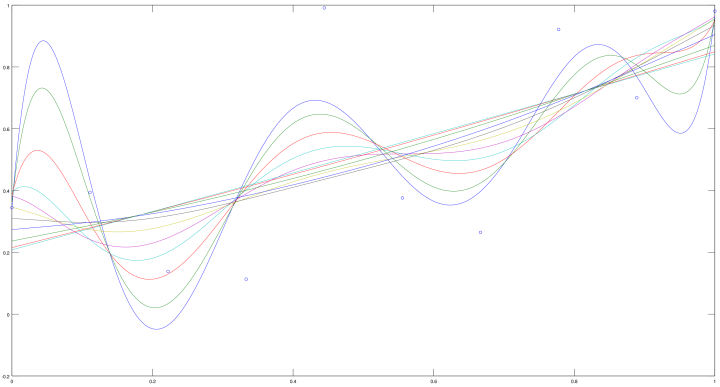


Abbildung 2.7: Die Lösung des Glättungspolynoms. Man sieht, wie sich die Funktione mehr und mehr den vorgegebenen Daten annähern.

Eine Funktion ist also `f = @(...) ...;` wobei in den Klammern die Liste der Funktionsparameter steht und dahinter der zu berechnend Ausdruck. Also können wir die Funktion auch plotten

```
octave> plot( (0:.1:10)', f( (0:.1:10)' ) );
error: operator *: nonconformant arguments (op1 is 101x1, op2 is 101x1)
```

wobei die „richtige“ Fehlermeldung noch deutlich länger ist. Was haben wir also falsch gemacht? Richtig, die Art und Weise wie die Funktion mit Vektoren umgeht, wir wollen ja eine komponentenweise Aktion.

```
octave> f = @(x) x.*cos(x);
octave> plot( (0:.1:10)', f( (0:.1:10)' ) );
```

Und schon klappt's. Auf diese Weise kann man sogar Funktionen an `Matlab`–Funktionen übergeben, siehe Programm 2.5:

```
octave> PlotAFun( f,-10,10,200 );
```

2.4.2 Integration

Bei der **Integration** berechnen wir (erst einmal) ein **bestimmtes Integral**

$$\int_a^b f(x)\,dx$$

wobei `f` eine integrierbare[42] Funktion sein sollte. Obwohl `Matlab`[43] über eine Funktion namens `integral` verfügt, ist der „gemeinsame Nenner“ die Funktion `quad`, was für **Quadratur** steht, den „vornehmen“ Begriff für numerische Integration, (Gautschi, 1997; Sauer, 2013). Die Integralberechnung erfolgt durch Eingabe der Funktion und der Integrationsgrenzen

[42] In welchem Sinn auch immer.
[43] Im Gegensatz zu `Octave`.

```
%% PlotAFun.m
%% Plotte eine Funktion
%%
%% f = funktion
%% a = Anfang, b = Ende
%% N = #Punkte

function PlotAFun( f,a,b,N )
  X = linspace( a,b,N );
  plot( X,f(X) );
end
```

Programm 2.5: Kleine Routine zum Plotten einer Funktion, die in Octave selbst implementiert ist.

```
octave> f = @(x) cos(x); quad(f,-pi,pi)
ans =    2.6159e-16
```

bzw.

```
>> f = @(x) cos(x); quad( f,-pi,pi )
ans =
   4.0143e-09
>> f = @(x) cos(x); integral(f,-pi,pi)
ans =
   2.7756e-16
```

in Matlab, was deutlich bzw. minimal schlechter ist, denn das exakte Integral wäre Null. Daran erkennt man, daß beide Systeme trotz identischer Syntax unterschiedliche Pakete zur numerischen Integration verwenden, bei Octave ist das QUADPACK, ein freies aber sehr gut ausgearbeitetes Paket zur numerischen Berechnung von Integralen. In Octave kann man auch uneigentliche Integrale berechnen, selbst wenn es eine Warnug gibt,

```
octave> f = @(x) cos(x)/(x^2); quad( f,0,Inf )
 ABNORMAL RETURN FROM DQAGI
ans = -2.5712
```

in Matlab tut dies die Funktion integral,

```
>> f = @(x) cos(x)./(x.^2); integral( f,0,Inf )
Warning: Reached the limit on the maximum number of intervals in
use. Approximate bound on error is   3.1e+72.
The integral may not exist, or it may be difficult to approximate
numerically to the requested accuracy.
...
ans =
   3.1423e+72
```

Auch hier wurden einige Zeilen Fehlermeldungen entfernt. Es gibt noch eine Funktion quadgk in beiden Systemen, die eine sogenannte **Gauß–Kronrod–Formel** verwendet und die sogar eine Fehlerabschätzung mitberechnet:

```
octave> f = @(x) cos(x)./(x.^2); quadgk( f,0,Inf )
warning: quadgk: Error tolerance not met.  Estimated error 2.90256e+07
ans =    2.9824e+07
```

Matlab liefert

```
>> f = @(x) cos(x)./(x.^2); quadgk( f,0,Inf )
Warning: Reached the limit on the maximum number of intervals in use.
Approximate bound on error is  1.1e+58. The integral may not
exist, or it may be difficult to approximate numerically.
Increase MaxIntervalCount to 728 to enable QUADGK to continue for
another iteration.

ans =
   1.1164e+58
```

Offensichtlich ist dieses Vorgehen, zumindest in diesem Fall nicht wirklich von Erfolg gekrönt.

Übung 2.8 Berechnen Sie das unbestimmte Integral

$$\int_0^\infty \frac{\cos x}{x^2}\,dx$$

und sagen Sie mir, was dabei herauskommt. ◊

Es gibt noch eine zweite Variante des „Integrierens“, nämlich die, bei der es darum geht, eine **gewöhnliche Differentialgleichung** zu lösen:

$$x'(t) = g(t), x(0) = x_0.$$

Da

$$x(t) = x(0) + \int_0^t x'(\tau)\,d\tau = x_0 + \int_0^x g(\tau)\,d\tau,$$

können wir Differentialgleichungen lösen, wenn wir Funktionen gut integrieren können und umgekehrt[44]. Hier gibt es allerdings wirkliche Unterschiede zwischen den Sytemen. Um Matlab–kompatible Funktionen zu erhalten, muss man in Octave zusätzlich das Paket odepkg nachinstallieren, was einem aber auch gesagt wird:

```
octave> help ode45
error: help: Octave provides lsode for solving differential equations.
For more  information try 'help lsode'.  Matlab-compatible ODE
functions are provided by the odepkg package.  See
<http://octave.sourceforge.net/odepkg/>.
```

[44]Oftmals liefern Funktionen zum nuerischen Lösen von Differentialgleichungen auch sehr gute Methoden zur Integration einer Funktion

Allgemeiner behandelt man ein nichtlineares System von gewöhnlichen Differentialgleichungen

$$x'(t) = f(x,t), \qquad x : \mathbb{R} \to \mathbb{R}^n, \tag{2.24}$$

bei der die **Veränderung** x' also von **Zustand** x und vom **Zeitpunkt** t abhängen darf. Ein Ausdruck der Form (2.24) heißt **Normalform** der gewöhnlichen Differentialgleichung. Ein wesentlicher Aspekt bei der *Verwendung* mathematischer Software besteht auch darin, ein reales Problem in eine maschinentaugliche Normalform zu bringen Sehen wir uns einmal an, wie man nun ein System von gewöhnlichen Differentialgleichungen löst und machen wir uns den Spaß, etwas beinahe realitätsnahes anzusehen, nämlich die sogenannten **Volterra–Gleichungen** für das **Räuber–Beute–Problem**. Dazu gibt es zwei Populationen, x_1, x_2, wobei x_1 die Population der Beutetiere[45] und x_2 die der Räuber[46] beschreibt. Die Differentialgleichungen

$$x_1' = x_1\,(r_1 - \gamma_1 x_2) \qquad x_2' = -x_2\,(\delta_2 - r_2 x_1)$$

beschreiben das Änderungsverhalten, wobei r_1 die Reproduktionsrate der Beutetiere ist, γ_1 die Fressrate der Räuber[47], δ_2 die Sterberate der Räuber, wenn es keine Beutetiere mehr gibt und r_2 die Reproduktionsrate der Räuber pro Beutetier[48]. All diese Werte nimmt man Fall als konstant und positiv an. Damit erhalten wir

$$x' = \begin{bmatrix} r_1 & \\ & -\delta_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} -\gamma_1 \\ r_2 \end{bmatrix} x_1 x_2 =: f(x,t),$$

wobei `f` *nicht* von `t` abhängt. Diese Funktion modellieren wir in `Matlab` bzw. `Octave` als

```
octave> r1 = 2; g1 = .1; d2 = .5; r2 = .1;
octave> f = @(x,t) diag( [r1,-d2] ) *x + [-g1;r2] *prod(x);
```

und lassen mit

```
octave> X = lsode( f,[20,10],(0:.1:20) ); plot(X)
```

das Problem lösen. Bei `Matlab` gibt es zwei kleine Unterschiede: In der Definition der Funktion `f` sind Zeit und Zustand vertauscht

```
>> f = @(t,x) [ x(1)*(r1-g1*x(2));-x(2)*(d2-r2*x(1)) ];
```

und das gleiche gilt auch für die Aufrufparameter:

```
>> [T,X] = ode45( f,[0 20],[20 10]); plot( T,X );
```

[45] Ja, sollte ganzzahlig sein, aber wenn Sie wollen, bekommen Sie im Supermarkt auch ein halbes Hähnchen.

[46] Nichtganzzahlige Werte entstehen durch Freitage und Veganer.

[47] Hängt von der Größe ab und beschreibt, ob man das Verhältnis Elefant zu Fliegenmade oder Garnele zu Blauwal untersucht.

[48] Kennt man von Tauben in Städten.

Gibt man als Zeitbereich nur zwei Werte an, so werden die „Zwischenwerte“ automatisch bestimmt. Dafür gibt es eine Menge spezialisierter Solver, je nach Natur der Differentialgleichung, aber dafür muss man sich dann eben auch mit Differentialgleichungen auseinandersetzen.

Aber wie ist das nun mit Differentialgleichungen höherer Ordnung, also mit Ausdrücken der Form

$$x^{(n)} = f\left(x, x', \ldots, x^{(n-1)}, t\right) \qquad \begin{bmatrix} x(0) \\ x'(0) \\ \vdots \\ x^{(n-1)}(0) \end{bmatrix} = \mathbf{x}_0,$$

die ja nicht direkt der Form (2.24) einer **Differentialgleichung erster Ordnung** enstprechen. Hier behilft man sich mit einem einfachen Trick aus der Theorie und Numerik der gewöhnlichen Differentialgleichungen und setzt

$$y_j(t) := x^{(j)}(t), \qquad j = 0, \ldots, n,$$

was zu der Gleichung

$$y'_{n-1} = y_n = f(y_0, \ldots, y_{n-1}, t) \qquad \begin{bmatrix} y_0(0) \\ y_1(0) \\ \vdots \\ y_{n-1}(0) \end{bmatrix} = \mathbf{x}_0,$$

und den Nebenbedingungen[49]

$$y_{j+1} = y'_j, \quad j = 0, \ldots, n \qquad \Rightarrow \qquad \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix}',$$

was man alles zusammen in das Differentialgleichungssystem

$$\mathbf{y}' = \begin{bmatrix} \begin{bmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \end{bmatrix} \mathbf{y} \\ f(\mathbf{y}, t) \end{bmatrix}, \qquad \mathbf{y}(0) = \mathbf{x}_0, \tag{2.25}$$

mit[50] $\mathbf{y} = (y_0, \ldots, y_{n-1})$.

Das schreit nach einem Beispiel, wofür wir die **Schwingungsgleichung**

$$x''(t) = -\lambda x(t), \qquad x(0) = 1, x'(0) = 0, \lambda > 0, \tag{2.26}$$

heranziehen. Wir haben $n = 2$, also $\mathbf{y} = (y_0, y_1)$ und das System (2.25) ist

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix}' = \begin{bmatrix} y_1 \\ -\lambda y_0 \end{bmatrix}, \qquad \mathbf{y}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Modellieren wir das in `Octave`, dann erhalten wir mit

[49] So *definiert* man ja sogar die höheren Ableitungen einer Funktion.

[50] Hier als **Tupel** geschrieben, nicht als (Spalten-)Vektor, da interessiert uns Transposition dann nicht.

```
octave> l = 1; f = @(x,t) [ x(2); l*x(1) ];
```

die Funktion $f(\mathbf{x}, t)$, die wir mit

```
octave> X = lsode( f,[ 1;0 ], (0:.1:10) );plot( (0:.1:10)',X );
```

berechnen und plotten lassen können. Was dabei angezeigt wird ist die Lösung des *Systems*, also die Funktion *und* ihre Ableitung. Das Bild könnte einem durchaus bekannt vorkommen, denn natürlich ist die Lösung von (2.26) gerade $x(t)$ cos t. Analog erhalten wir in `Matlab` die Lösung

```
>> l = 1; f = @(t,x) [ x(2);-l*x(1) ];
>> [T,X] = ode45( f,[0 10],[1 0]); plot(T,X);
>> [T,X] = ode45( f,[0 10],[0,1]); plot(T,X);
```

Raten Sie, ohne das Programm auszuführen mal, was die zweite Lösung ist. Und wenn wir schon dabei sind: Was liefert

```
>> [T,X] = ode45( f,[0 10],[1,1]); plot(T,X);
```

```
%% LinDGL.m
%% Lose lineare DGL x^(n) = a(1) x + ... + a(n) x^(n-1) + g
%%
%% a = Koeffizienten
%% g = Funktion
%% x0 = Startwert
%% T = Bereich
%% OCTAVE-Version

function X = LinDGL( a,g,x0,T )
  n = length(a);
  A = [ zeros( n-1,1 ), eye(n-1) ];
  f = @(y,t) [ A*y ; a * y + g(y,t) ];
  X = lsode( f,x0,T );
end
```

Programm 2.6: Kleines Skript zur Lösung einer linearen gewöhlichen Differentialgleichung mit konstanten Koeffizienten.

Aus diesen Bestandteilen kann man sich leicht ein Programm zur Lösung einer Differentialgleichung

$$\sum_{j=0}^{n} a_j x^{(j)}(t) = g(t) \qquad \text{bzw.}^{51} \qquad x^{(n)}(t) = \sum_{j=0}^{n-1} a_j x^{(j)}(t) + g(t)$$

basteln, siehe Prog. 2.6.

Was wir bisher betrachte haben bezeichnet man als **Anfangswertproblem**. Bei einem **Randwertproblem** werden am Anfang und Ende des Intervalls Bedingungen vorgegeben, natürlich an jeder Seite weniger als beim Anfangswertproblem. Auch für Randwertprobleme gibt es eigene Verfahren, auf die wir hier aber nicht weiter eingehen wollen.

2.4.3 Optimierung

Die **Optimierung** befasst sich mit der Bestimmung von Maxima oder Minima von Funktionen, also dem Auffinden von Stellen $x \in D$ für eine **Zielfunktion** $f : D \to \mathbb{R}$, so daß

$$f(x) = \min_{y \in D} f(y).$$

Es genügt immer, sich Verfahen für das Finden von Minima anzusehen, denn ist x ein Maximum von $-f$, dann ist x ein Minimum von f.

Optimierungsprobleme kann man auf verschiedene Weisen klassifizieren, einmal nach der Funktion:

1. f ist eine **lineare Funktion**, also[52] $f(x) = c^T x$, dann gibt es eigene Optimierungsverfahren, dazu kommen wir noch.

2. f ist eine **konvexe Funktion**, das heißt,

$$f((1-\alpha)x + \alpha x') \leq (1-\alpha)\, f(x) + \alpha\, f(x'), \qquad \alpha \in [0,1]. \tag{2.27}$$

   f ist eine **konkave Funktion**, wenn $-f$ konvex ist. Eine lineare Funktion ist konvex und konkav. Die Definition (2.27) ist natürlich nur dann sonnvoll, wenn auch D eine **konvexe Menge** ist, d.h., wenn

$$(1-\alpha)x + \alpha x' \in D, \qquad x, x' \in D,\ \alpha \in [0,1].$$

3. Die Funktion kann differenzierbar sein oder nicht. Günstig sind zweimal stetig differenzierbare Funktionen, da sich dann Extrema durch Nullstellen des Gradienten und die Definitheit der **Hesse–Matrix** beschreiben lassen können.

Warum konvexe Funktionen so toll sind, erklärt die folgende einfach zu beweisende Aussage.

Satz 2.11 *Ist* $f : D \to \mathbb{R}$ ***strikt konvex**, d.h., gilt Gleichheit in* (2.27) *nur für*[53] $\alpha \in \{0,1\}$*, dann hat* f *genau ein* ***striktes Minimum*** x *mit* $f(x) < f(x')$, $x' \in D \setminus \{x\}$.

Beweis: Seien $x \neq x'$ zwei Minimalstellen, also $f(x) = f(x') \leq f(y)$, $y \in D$, dann ist für $\alpha \in (0,1)$

$$f\left((1-\alpha)x + \alpha x'\right) < (1-\alpha) f(x) + \alpha \underbrace{f(x')}_{=f(x)} = f(x)$$

und wir hätten ein neues, strikteres Minimum gefunden. □

Man kann aber auch nach den Nebenbedingungen klassifizieren:

1. Das Problem ist **unbeschränkt**, d.h. $D = \mathbb{R}^n$.

[52] Eigentlich $f(x) = c^T x + b$, aber da Konstanten bei der Optimierung keine Rolle spielen, kann man c auch weglassen.

[53] Und da gilt sie trivialerweise.

2. Das Problem ist **beschränkt**, wobei die **Nebenbedingungen** normalerweise in der Form

$$g(x) = 0, \qquad h(x) \leq 0, \qquad g : \mathbb{R}^n \to \mathbb{R}^p, \quad h : \mathbb{R}^n \to \mathbb{R}^q.$$

Extrema mit und ohne Nebenbedingungen für differenzierbare Funktionen kennt man aus der Analysis (Sauer, 2015), in einem Fall setzt man „einfach" die Ableitung $= 0$, im anderen Fall muss man die bereits bekannten **Lagrange–Multiplikatoren** verwenden.

Die eierlegende Wollmilchsau zur *unbeschränkten* Optimierung in `Matlab` und `Octave` ist die Funktion `fminsearch`, die ausgehend von einem **Startpunkt** `x0` ein Minimum einer Funktion sucht:

```
>> f = @(x) cos(x(1))*sin(x(2));
>> fminsearch( f,[ 0 0 ] )
ans =
    0.0000   -1.5708
```

bzw.

```
octave> f = @(x) cos(x(1))*sin(x(2));
octave> fminsearch( f,[ 0 0 ] )
ans =
  -3.1132e-05  -1.5708e+00
```

Obwohl unterschiedliche Ergebnisse herauskommen[54] verwenden beide Programme das sogenannt **Nelder–Mead–Verfahren**, das ausschließlich mit Funktionswerten arbeitet und *keine* Ableitungen verwendet. Diese Verfahren sind zwar universeller, aber normalerweise auch langsamer als spezialisierte Methoden. In erweiterter Form kann man sich nicht nur die Minimalstelle sondern auch den Funktionswert dort ausgeben lassen, `Matlab` lässt auch noch ein Fehlerflag zu:

```
>> [x,fx,exitflag] = fminsearch( f,[ 0 0 ] )
x =
    0.0000   -1.5708
fx =
   -1.0000
exitflag =
     1
>> fx+1
ans =
   5.5657e-10
```

beziehungsweise

[54] In diesem Fall geht der Sieg anscheinend an `Matlab`.

```
octave> [x,fx] = fminsearch( f,[ 0 0 ] )
x =
  -3.1132e-05  -1.5708e+00
fx = -1.00000
octave> fx+1
ans =    5.0781e-10
```

und plötzlich ist das Ergebnis von `Octave` sogar *besser* als das von *Matlab*. Woher das kommt, zeigt die Differenz zur exakten Lösung

```
>> x+[0,pi/2]
ans =
   1.0e-04 *

    0.0426    0.3309
>> norm(ans)
ans =
   3.3364e-05
```

bzw.

```
octave> x+[0,pi/2]
ans =
  -3.1132e-05   6.8119e-06
octave> norm(ans)
ans =    3.1869e-05
```

Die `Octave`–Lösung liegt also geringfügig näher an der Wahrheit als das was `Matlab` produziert. Das sind aber nun wirklich Implementierungsdetails. Das Nelder–Mead–Verfahren ist ein **Suchverfahren** und verwendet nur Punktauswertungen. Dazu verwendet man im $\mathbb{R}^n$ eine Menge von $n+1$ Punkten $x_0, \ldots, x_n$, die ein nichtentartetes Simplex aufspannen sollen, d.h.

$$\det \begin{bmatrix} 1 & \ldots & 1 \\ x_0 & \ldots & 0 \end{bmatrix} \neq 0,$$

vertauscht dann die Punkte so, daß $f(x_0) \geq f(x_j)$, $j = 1, \ldots, n$, und bildet den **Schwerpunkt**

$$x^* = \frac{1}{n} \sum_{j=1}^{n} x_j$$

der „besseren Punkte“, an dem man dann x_0 spiegelt, also Punkte

$$x_\alpha := (1 - \alpha)\, x_0 + \alpha\, x^*, \qquad \alpha > 1,$$

betrachtet. Dann spielt man trickreich mit dem α herum, um eine möglichst große Verbesserung[55] und ersetzt x_0 durch x_α für ein passendes α. Das macht man, bis für eine vorgegebene Toleranz τ eines der folgenden Kriterien erfüllt ist:

[55] Oder zumindest eine Verbesserung, auch damit ist man oftmals schon glücklich.

1. Die Punkte liegen dicht:

$$\|x_j - x_k\| \leq \tau, \qquad j,k = 0,\ldots,n. \tag{2.28}$$

2. Der schlechteste Punkt ist nicht wirklich schlecht:

$$\sum_{j=0}^{n} (f(x_j) - f(x^*))^2 \leq \tau^2. \tag{2.29}$$

3. Die Funktionswerte sind praktisch konstant:

$$\sum_{j=0}^{n} (f(x_j) - \mu)^2 \leq \tau^2, \qquad \mu = \frac{1}{n+1} \sum_{j=0}^{n} f(x_j). \tag{2.30}$$

Damit ist klar, daß es hier sowohl eine Menge von Parametern gibt, die man einstellen kann[56] sowie Implementierungsdetails bei der Bestimmung von α.

Eine wesentliche „Schwäche“, die das Verfahren mit so ziemlich allen iterativen Methoden teilt, ist die Tatsachen, daß es nur **lokal konvergent** ist, alos nur **lokale Extrema** findet, aber kein **globales Minimum**. Ein Beispiel mit der unbeschränkten Funktion

$$f(x) = \|x\| \cos x_1 \ \sin x_2, \tag{2.31}$$

siehe Abb. 2.8 liefert dann auch das Ergebnis

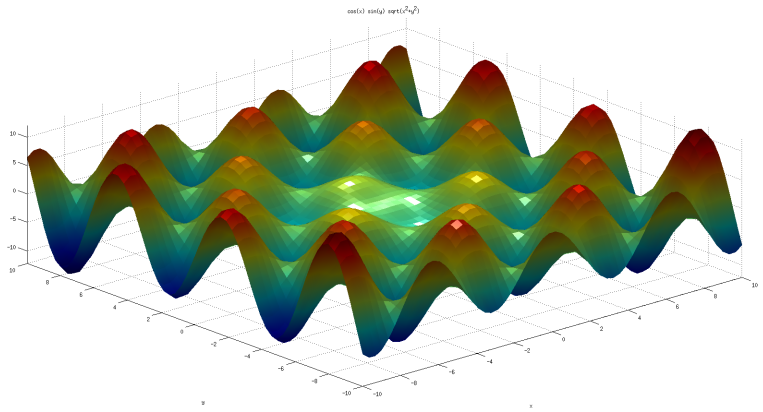


Abbildung 2.8: Die Funktion aus (2.31) mit `ezsurf`–Befehl von `Matlab` geplottet. Handlich ist das schon.

[56] Insbesondere die Toleranz τ und die Anzahl der Iterationsschritte, die das Verfahren maximal durchführen soll.

```
>> f = @(x) cos(x(1))*sin(x(2))*norm(x);
>> [x,fx] = fminsearch( f,[ 0 0 ] )
x =
   -0.0000   -2.0287
fx =
   -1.8197
```

was definitiv nicht das globale Minimum ist.

2.4.4 Lineare Optimierung

Jetzt aber noch zu einem wirklich wichtigen Kapitel, nämlich zu der Frage, wie man ein **lineares Optimierungsproblem** der Form

$$\min_x f(x) = c^T x, \qquad Ax = b,\ x \geq 0. \tag{2.32}$$

löst.

Übung 2.9 Zeigen Sie: Ohne Nebenbedingungen sind linere Optimierungsprobleme mit einer Ausnahme unlösbar. Was ist die Ausnahme? ◊

Die Aufgabenstellung in (2.32) ist eine **Normalform** unter vielen, aber wir verwenden Sie, weil das oftmals in `Octave` integrierte Paket `glpk` darauf basiert. Minimierung und Maximierung sind nur das Vorzeichen von c und Probleme der Form $Ax \leq b$ bzw. $Ax \geq b$ kann man durch Einführen der **Schlupfvariablen** $y = b - Ax$ kompensieren:

$$\min_x f(x,y) = \underbrace{\begin{bmatrix} c^T & 0 \end{bmatrix}}_{=:\widehat{c}^T} \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \underbrace{\begin{bmatrix} A & \pm I \end{bmatrix}}_{=:\widehat{A}} \begin{bmatrix} x \\ y \end{bmatrix} = b, \begin{bmatrix} x \\ y \end{bmatrix} \geq 0. \tag{2.33}$$

Jetzt ist ein Blick auf die Dimensionen angebracht, die durch die Größe der Matrix $A \in \mathbb{R}^{m\times n}$ vorgegeben ist. Das liefert

$$x \in \mathbb{R}^n, \qquad c \in \mathbb{R}^n, \qquad b \in \mathbb{R}^m, \qquad y \in \mathbb{R}^m$$

und damit auch

$$\widehat{c} \in \mathbb{R}^{n+m}, \qquad \widehat{A} \in \mathbb{R}^{m\times m+n}$$

in (2.33). Um also ein Problem der Form

$$\min_x c^T x, \qquad Ax \geq b,\ x \geq 0, \tag{2.34}$$

zu lösen, müssen wir es in `Octave` nur wie in Prog. 2.7 aufbereiten, was man am besten eben gleich in eine kleine Funktion packt.

Übung 2.10 Wie konvertiert man von Form (2.32) in (2.34)? Schreiben Sie ein kleines Analogon zu Prog. 2.7. ◊

Jetzt aber zum Spaßteil der Veranstaltung: Wie bringt man denn überhaupt ein „richtiges“ Problem, also sozusagen eine „Textaufgabe“ in die passende Form, damit unsere Software sie auch für uns lösen kann? Hier ein paar Beispiele, die zum größten Teil aus (Gass, 1970a) stammen.

```
%% OptNormal.m
%% Konvertiere min c'*x, Ax >= b, x >= 0 in NF
%%
%% c = Vektor Zielfunktion
%% A = Matrix
%% b = rechte Seite
%% OCTAVE-Version

function [cc,AA,bb] = OptNormal( c,A,b )
  [m,n] = size(A);
  cc = [ c ; zeros(m,1) ];
  AA = [ A, -eye( m ) ];
  bb = b;
end
```

Programm 2.7: Transformation der Normalform (2.34) in die Normalform (2.32) von `glpk`.

Beispiel 2.12 (Frühstücksplanung) *Eine Hausfrau versucht für Ihre Familie ein optimales Frühstück zusammenzustellen. Dafür stehen ihr*[57] *zwei verschiedene Typen von Getreideflocken*[58]*, nämlich* Crunchies *und* Krispies *zur Verfügung, die zwei Spurenelemente,* ***Thiamin***[59] *und* ***Niacin***[60] *in unterschiedlicher Anzahl enthalten, unterschiedlichen Brennwert in Kalorien liefern und natürlich unterschiedlich teuer sind. Das ideale Frühstück versorgt die Familie mit einem gewissen Mindestmaß an „Vitaminen" und Kalorien und ist dabei natürlich möglichst billig. Die genauen Werte sind in der folgenden Tabelle aufgelistet:*

	Crunchies	*Krispies*	*Benötigt*
Thiamin (in mg)	*0.10*	*0.25*	*1*
Niacin (in mg)	*1.00*	*0.25*	*5*
Kalorien	*110*	*120*	*400*
Preis	*3.8*	*4.2*	

Das Problem ist klar: Was ist die optimale Diät, die diese Randbedingungen erfüllt?

Nun, dieses Problem ist, was die Modellierung angeht, noch richtig einfach, denn wir müssen nur die Bedingungen in Ungleichungsform hinschreiben. Seien dazu x_1 die Menge der verwendeten Crunchies und x_2 die Menge an Krispies, dann erhalten wir die Ungleichungen

$$\begin{aligned} 0.1\,x_1 + .25\,x_2 &\geq 1 \\ x_1 + .25\,x_2 &\geq 5 \\ 110\,x_1 + 120\,x_2 &\geq 400 \end{aligned}$$

[57] Ja, das Beispiel stammt aus den USA.

[58] Auf gut neudeutsch auch als "Cerealien" bezeichnet – das kommt davon, wenn's an Zerebralien mangelt.

[59] Synonym für Vitamin B_1, siehe (Pschyrembel, 1994).

[60] Synonym für Nicotinsäure, die Nebenwirkungen in (Pschyrembel, 1994) liest man besser nicht.

und zu minimieren sind die Kosten $3.8\,x_1 + 4.2\,x_2$. In der[61] Normalform (2.34) erhalten wir somit

$$\min 3.8\,x_1 + 4.2\,x_2, \qquad \begin{bmatrix} 0.1 & 0.25 \\ 1 & 0.25 \\ 110 & 120 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 5 \\ 400 \end{bmatrix},$$

also

```
octave> c = [ 3.8; 4.2 ]; A = [ .1 .25; 1 .25; 110 120 ];
octave> b = [ 1;5;400 ];
octave> [cc,AA,bb] = OptNormal( c,A,b );
```

und wir erhalten die Optimallösung durch

```
octave> glpk( cc,AA,bb )
ans =
     4.44444
     2.22222
     0.00000
     0.00000
   355.55556
```

Die Analyse dieser Ausgabe ist durchaus interessant. Die ersten beiden Komponenten sind die x–Werte der **Optimallösung** $\begin{bmatrix} x \\ y \end{bmatrix}$, wir brauchen also $\frac{40}{9}$ Crunchies und $\frac{20}{9}$ Krispies, aber die anderen drei Einträge sind ebenfalls interessant, denn sie sagen uns, welche Nebenbedingungen *exakt* erfüllt werden und welche Nebenbedingungen übererfüllt werden. Hier sagt uns die Ausgabe, daß wir die benötigten Nährstoffe um den Preis von 355.55... Extrakalorien zusammenbekommen. American Breakfeast ...

Beispiel 2.13 (Kühlschränke) *Eine Firma stellt in zwei Fabriken,* F_1 *und* F_2*, Kühlschränke her, die in den Läden*[62] S_1, S_3, S_3 *verkauft werden sollten. Die Kosten-/Ressourcen–Matrix ist wie folgt:*

	S_1	S_2	S_3	
F_1	8	6	10	11
F_2	9	5	7	14
	10	8	7	

Wie ist der optimale Transport?

Transportprobleme zeichnen sich dadurch aus, daß man sehr viele Variablen hat, die man am zweckmäßigsten *doppelt* indiziert, nämlich als x_{jk}, wobei x_{jk} die Menge bezeichnet, die vom Ausgangspunkt j zum Zielpunkt k transportiert wird. Die Gesamtkosten sind dann immer

$$\sum_{j=1}^{m} \sum_{k=1}^{n} a_{jk} x_{jk}.$$

[61] Normalerweise leichter zugänglichen.
[62] "Shop".

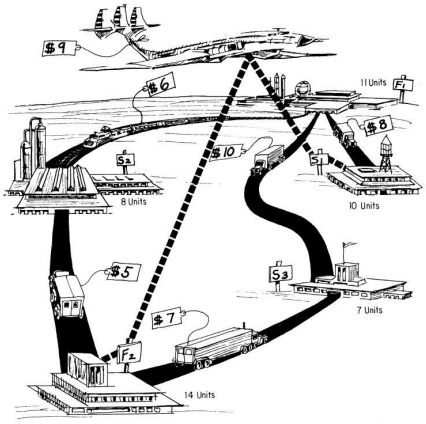


Abbildung 2.9: Die Kühlschränke und deren Transportwege. Aus (Gass, 1970b).

In unserem Beispiel haben wir also die Variablen

$$x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23} \qquad \Rightarrow \qquad x = [x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}]^T$$

auch in einen Vektor angeordnet, indem wir die sogenannte *lexikographische*[63] Ordnung verwenden. Unser Beispiel liefert nun die Nebenbedingungen

$$\begin{array}{ccccccccc}
x_{11} & +x_{12} & +x_{13} & & & & \leq & 11 \\
 & & & x_{21} & +x_{22} & +x_{23} & \leq & 14 \\
x_{11} & & & +x_{21} & & & \geq & 10 \\
 & x_{12} & & & +x_{22} & & \geq & 8 \\
 & & x_{13} & & & +x_{23} & \geq & 7
\end{array}$$

Die ersten beiden Ungleichungen sind die Beschränkungen an die Ressourcen, die anderen drei betreffen das Minimum, das an den Zielpunkten ankommen soll. Damit können wir uns auch schon wieder ans Modellieren machen: Nachdem wir noch ein paar unpassende Vorzeichen umgedreht haben, erhalten wir die folgenden Parameter:

$$A = \begin{bmatrix} 1 & 1 & 1 & & & \\ & & & 1 & 1 & 1 \\ -1 & & & -1 & & \\ & -1 & & & -1 & \\ & & -1 & & & -1 \end{bmatrix}, \qquad b = \begin{bmatrix} 11 \\ 14 \\ -10 \\ -8 \\ -7 \end{bmatrix}, \qquad c = \begin{bmatrix} 8 \\ 6 \\ 10 \\ 9 \\ 5 \\ 7 \end{bmatrix}$$

Da das bereits in `gplk`–Normalform ist, können wir es direkt an den Computer übergeben: Zuerst[64] die Matrix A

[63] Indizes werden angeordnet wir im Lexikon: zuerst ordnet man nach dem ersten Eintrag, dann nach dem zweiten und so weiter.

[64] Die Zeilenumbrüche sind der Übersichtlichkeit halber *nachträglich* eingefügt, liebe Kinder, bitte nicht nachmachen.

```
octave> A = [ ones(1,3), zeros( 1,3 ); zeros(1,3), ones( 1,3 );
             -eye(3),-eye(3)]
A =

   1   1   1   0   0   0
   0   0   0   1   1   1
  -1   0   0  -1   0   0
   0  -1   0   0  -1   0
   0   0  -1   0   0  -1
```

dann b und c,

```
octave> b = [ 11 14 -10 -8 -7 ]; c = [ 8 6 10 9 5 7 ];
```

und schon kann es losgehen:

```
octave> glpk( c,A,b )
ans =
   10
    1
    0
    0
    7
    7
```

was die Transportmatrix

	S_1	S_2	S_3	
F_1	10	1	0	≤ 11
F_2	0	7	7	≤ 14
	≥ 10	≥ 8	≥ 7	

liefert, die sowohl die Nebenbedingungen erfüllt als auch die Transportkosten minimiert. Wenn man genau hinsieht, dann sind Transportmatrizen sehr einfach strukturiert, sie bestehen aus Realisierungen von Zeilensummen und negativen Spaltensummen, also aus $m + n$ Nebenbedingungen in den mn Variablen. Das kann man in ein kleines Programm packen, siehe Prog. 2.8, das zwei nette `Matlab`–Befehle verwendet:

kron berechnet das **Kronecker–Produkt**

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} \in \mathbb{R}^{mp \times nq}, \qquad A \in \mathbb{R}^{m \times n},\ B \in \mathbb{R}^{p \times q}.$$

mit dem Spezialfall

$$I \otimes v^T = \begin{bmatrix} v^T & & \\ & \ddots & \\ & & v^T \end{bmatrix},$$

was für den oberen Teil der Matrix verwendet wird.

```
%% TransMat.m
%% Transportmatrix
%%
%% m,n = Dimensionen

function A = TransMat( m,n )
  A = [ kron( eye(m), ones(1,n) ) ; repmat( -eye( n ), 1,m ) ];
  %% kurz und cool
```

Programm 2.8: Berechnung einer m, n–Transportmatrix $A \in \mathbb{R}^{m+n \times mn}$ unter Verwendung von netten `Matlab`–Features.

`repmat` bildet Blockmatrizen durch einfache Wiederholung einer vorgegebenen Matrix.

Beispiel 2.14 (Noch ein Transportproblem) *Ausrüstungsgegenstände sollen von drei Basen auf fünf andere Basen verteilt werden, wobei die zurückgelegte Gesamtdistanz minimiert werden soll. Die Vorgaben, wie in (Gass, 1970b, S. 20) sind wie folgt:*

	MacDill	*March*	*Davis-Monthan*	*McConnell*	*Pinecastle*	
Oklahoma City	*938*	*1030*	*824*	*136*	*995*	*8*
Macon	*346*	*1818*	*1416*	*806*	*296*	*5*
Columbus	*905*	*1795*	*1590*	*716*	*854*	*8*
	3	*5*	*5*	*5*	*3*	

Das sind jetzt also solide 15 *Variable und da wird's langsam heftig.*

Mit unserer kleinen Funktion reduziert sich das Ganze dann auf

```
octave> A = TransMat( 3,5 ); b = [ 8 5 8 -3 -5 -5 -5 -3 ]';
octave> c = [ 938 1030 824 136 995 346 1818 1416 806 296 905 1795
   1590 716 854 ]';
```

und

```
octave:16> [x,fx] = glpk( c,A,b )
x =
   0
   3
   5
   0
   0
   3
   0
   0
   0
```

```
    2
    0
    2
    0
    5
    1
fx =  16864
```

die Minimallösung hat also eine Gesamtdistanz von 16864 und die einzelnen Transportwerte lassen sich einfach ablesen.

Beispiel 2.15 (Maximaler Transport oder "Fluß" im Netzwerk) *Das Netzwerk aus Abb. 2.10 stelle alle Möglichkeiten dar, mit öffentlichen Verkehrsmitteln von S nach Z zu gelangen, wobei* 1, 2, 3 *die Umsteigepunkte seien. Wieviele Fahrgäste kann man maximal von S nach Z bringen, wenn die Kapazitäten der Verkehrsmittel*[65] *wie in Abb. 2.10 dargestellt sind, und wie muß man die Fahrgäste auf die einzelnen Verkehrsmittel verteilen?*

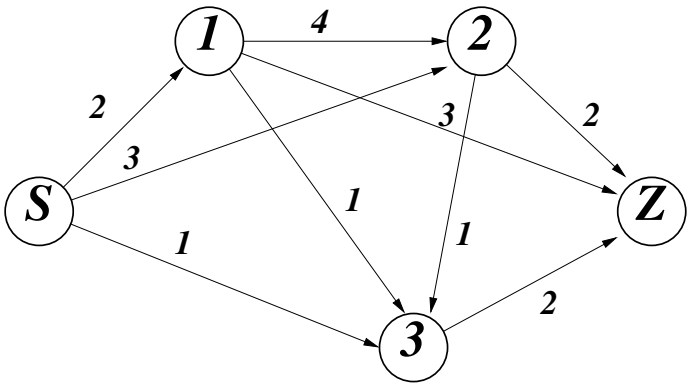


Abbildung 2.10: Das Netzwerk und die Kapazitäten der einzelnen Kanten.

Wieder bezeichnen wir mit x_{jk} die Anzahl der Passagiere, die von Knoten j nach Knoten k fahren, wobei Knoten 0 der Startpunkt und Knoten 4 der Zielpunkt ist. Damit werden die Kapazitätsbeschränkungen, ohne daß wir irgendwie nachdenken müssen, sofort zu Nebenbedingungen:

$$\begin{array}{rcl} x_{01} & \leq & 2 \\ x_{02} & \leq & 3 \\ x_{03} & \leq & 1 \\ x_{12} & \leq & 4 \\ x_{13} & \leq & 1 \\ x_{14} & \leq & 3 \\ x_{23} & \leq & 1 \\ x_{24} & \leq & 2 \\ x_{34} & \leq & 2 \end{array} \qquad (2.35)$$

[65] Sagen wir in der Einheit "100 Fahrgäste".

Das war der einfache Teil. Was wir außerdem noch fordern müssen, ist, daß niemand an einem Umsteigepunkt vergessen wird und dort verhungern muß, daß also alles, was in einen Knoten *hineinfließt*, auch wieder *herausfließen* muß, was wir mathematisch als

$$\sum_j x_{jk} = \sum_j x_{kj}, \qquad \forall k$$

schreiben können: Die Summe über die x_{jk} ist je gerade die Menge, die in den Knoten k hineintransportiert wird und die Summe über die x_{kj} die Menge, die von Knoten k in andere Knoten weitergeleitet wird. In unserem Beispiel entnehmen wir Abb. 2.10 die Nebenbedingungen

$$\begin{array}{rrrrrrrrcl} x_{01} & & & -x_{12} & -x_{13} & -x_{14} & & & = & 0 \\ & x_{02} & & +x_{12} & & & -x_{23} & -x_{24} & = & 0 \\ & & x_{03} & & +x_{13} & & +x_{23} & & -x_{34} = & 0 \end{array} \tag{2.36}$$

und wie wir die in Ungleichungsbedingungen umwandeln, das wissen wir ja schon. Bleibt noch, daß das was wir in das System reinstecken, also was aus S "hinausfließt", auch in Z ankommen muß. Nennen wir diesen Wert t, dann erhalten wir schließlich noch die beiden Nebenbedingungen

$$\begin{array}{rrrrrrcl} -x_{01} & -x_{02} & -x_{03} & & & +t & = & 0 \\ & & & x_{14} & +x_{24} & +x_{34} & -t & = & 0 \end{array} \tag{2.37}$$

Und was ist unser Ziel? Wir wollen ja den *Gesamtfluß* maximieren, also nichts anderes als den Wert t, der gerade in unserer Nebenbedingung aufgetaucht ist. Dazu müssen wir also t als *zusätzliche* Variable einführen und haben unser Problem fertig modelliert. Jetzt müssen wir es nur noch computergerecht aufbereiten, wobei wir t als zusätzliche, zehnte Variable ansetzen. Beginnen wir mit den Erhaltungsbedingungen aus (2.36) und (2.37), die wir in einer 5×10–Matrix modellieren können:

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

   1   0   0  -1  -1  -1   0   0   0   0
   0   1   0   1   0   0  -1  -1   0   0
   0   0   1   0   1   0   1   0  -1   0
  -1  -1  -1   0   0   0   0   0   0   1
   0   0   0   0   0   1   0   1   1  -1
```

Das sieht schon einmal gut aus. Für die Ungleichungsnebenbedingungen (2.35), in denen t ja *nicht* auftaucht benötigen wir noch 9 Schlupfvariablen, die allerdings addiert werden sollen, da die Nebenbegingungen in (2.35) ja von der Form „$\leq$“ sind:

```
octave> B = [ eye(9), zeros( 9,1 ), eye(9) ];
```

Dann setzen wir die beiden Matrizen zusammen, wobei wir berücksichtigen, daß die Schlupfvariablen nicht in den Erhaltungsgleichungen auftauchen:

```
octave> AA = [ A, zeros(5,9) ; B ];
```

Die Zielfunktion ist nur $-t$, denn wir maximieren ja den Fluß, und die rechte Seite betrifft die letzten neun Gleichungen:

```
octave> cc = [ zeros(9,1); -1 ; zeros(9,1) ];
octave> bb = [ zeros(5,1); 2; 3; 1; 4; 1; 3; 1; 2; 2 ];
```

Dann bleibt uns nur noch die Bestimmung der Lösung:

```
octave:21> [x,fx] = glpk( cc,AA,bb )
x =

   2
   3
   1
   0
   0
   2
   1
   2
   2
   6
   0
   0
   0
   4
   1
   1
   0
   0
   0
fx = -6
```

so daß die maximale Kapazität des Netzwerks also 6 ist. Der Lösungsvektor liefert

$$x = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 0 \\ 0 \\ 2 \\ 1 \\ 2 \\ 2 \end{bmatrix} \qquad \text{und} \qquad y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

wovon man die optimale Belegung der Kanten und die dort noch verbliebenen freien Kapazitäten ablesen kann.

2.4.5 Interpolation und Optimierung

Wir hatten es in (2.19) ja bereits mit der Minimierung von Zielfunktionen zu tun, dort mit dem Quadrat der 2–Norm. Jetzt wollen wir uns um „die“ zwei anderen Normen kümmern, nämlich um die ∞–Norm

$$\|f(X) - Y\|_\infty := \max_{j=1,\ldots,n} |f(x_j) - y_j| \tag{2.38}$$

und die 1–Norm

$$\|f(X) - Y\|_1 := \sum_{j=1}^{n} |f(x_j) - y_j| \,. \tag{2.39}$$

Beginnen wir mit (2.38). Dazu setzen wir

$$t := \max_{j=1,\ldots,n} |f(x_j) - y_j| = \max_{j=1,\ldots,n} |(F(X)a)_j - y_j| \,,$$

wobei a der Koeffizientenvektor ist, bezüglich dessen wir minimieren wollen. Damit ergibt sich

$$|(F(X)a)_j - y_j| \le t \qquad \Leftrightarrow \qquad \begin{cases} (F(X)a)_j - y_j & \le & t, \\ -(F(X)a)_j + y_j & \le & t, \end{cases} \tag{2.40}$$

was man in

$$F(X)a - \mathbf{1}t \le y, \qquad -F(X)a - \mathbf{1}t \le -y$$

bzw.

$$\begin{bmatrix} F(X) & -\mathbf{1} \\ -F(X) & -\mathbf{1} \end{bmatrix} \begin{bmatrix} a \\ t \end{bmatrix} \le \begin{bmatrix} y \\ -y \end{bmatrix} \tag{2.41}$$

darstellen kann und das lineare Optimierungsproblem ist dann

$$\min_{a,t} t, \qquad \begin{bmatrix} F(X) & -\mathbf{1} \\ -F(X) & -\mathbf{1} \end{bmatrix} \begin{bmatrix} a \\ t \end{bmatrix} \le \begin{bmatrix} y \\ -y \end{bmatrix}. \tag{2.42}$$

Leider reicht das aber nicht, denn unser Minimierer arbeitet ja immer mit der impliziten Nebenbedingung, daß alle Variablen ≥ 0 sein sollen, was für t kein Problem darstellt, für a aber sehr wohl[66]. Also noch ein Trick: Wir schreiben $a = b - b'$, $b, b' \ge 0$ und erhalten dann am Ende

$$\min_{b,b',t} t, \qquad \begin{bmatrix} -F(X) & F(X) & \mathbf{1} \\ F(X) & -F(X) & \mathbf{1} \end{bmatrix} \begin{bmatrix} b \\ b' \\ t \end{bmatrix} \ge \begin{bmatrix} -y \\ y \end{bmatrix}, \tag{2.43}$$

was wir mittels `OptNormal` dann auch auf die von `glpk` geforderte Form bringen können. Wir kopieren das Präludium aus dem Interpolationskapitel

[66] Wichtiger Lerneffekt: Auch diese Dinge sind Bestandteil der Normalform und müssen beim Aufruf von „Black Box“–Funktionen beachtet werden.

```
octave> x = linspace( 0,1,15 )';
octave> F = []; for j=0:4 F = [ F, x.^j ]; end
octave> y = rand( 15,1 );
```

und setzen

```
octave> c = [ zeros(10,1); 1 ];
octave> A = [ -F F ones(15,1); F -F ones(15,1) ];
octave> b = [ -y ; y ];
octave> [cc,AA,bb] = OptNormal( c,A,b );
octave> [z,fx] = glpk( cc,AA,bb );
```

Die ersten 11 Einträge von z enthalten dann b, b′ und t, die anderen 30 sind Schlupfvariablen. Sehen wir uns den relevanten Teil mal an:

```
octave> z(1:11)
ans =
    1.22888
    0.00000
   14.99384
    0.00000
    5.98306
    0.00000
    5.76869
    0.00000
   15.99418
    0.00000
    0.44493
```

und da finden wir auch[67] den Wert der Zielfunktion wieder (ganz unten). Extrahieren wir nun b und b′ in eine Matrix,

```
octave> b = [ z(1:5) z(6:10) ]
b =
    1.22888    0.00000
    0.00000    5.76869
   14.99384    0.00000
    0.00000   15.99418
    5.98306    0.00000
```

dann sehen wir ebenfalls etwas sehr interessantes, nämlich daß immer nur einer der beiden Werte in jeder Zeile $\neq 0$ ist. Das muss so sein, denn entweder[68] ist $a > 0$ oder $a < 0$ und die „effizienteste“ und „extremalste“ Lösung besteht darin, das nicht durch den anderen Wert unnötig zu kompensieren. Holen wir uns noch a

[67] Keine Überraschung ...

[68] Den Fall $a = 0$ schliessen wir hier einmal elegant aus.

```
octave> a = b(:,1)-b(:,2)
a =

    1.2289
   -5.7687
   14.9938
  -15.9942
    5.9831
```

Das Plotten der Funktion und der Daten ist Routine:

```
octave>  X = linspace( 0,1,500 )';
octave> G = []; for j=0:4 G = [ G, X.^j ]; end
```

Eine interessante Eigenschaft liefert ein anderer Plot:

```
octave> clf; plot( x,F*a - y )
```

der uns den sogenannten **Alternantensatz** der Approximationstheorie illustriert.

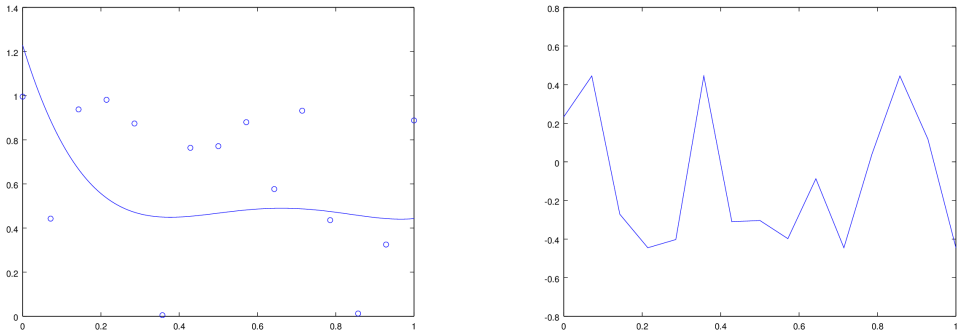


Abbildung 2.11: Plot der Optimallösung in der Maximumsnorm *(links)* und des Fehlers *(rechts)*. Man sieht sehr schön, daß die Extrema des Fehlers *alternierend*, also mit wechselndem Vorzeichen, angenommen werden.

Als letztes Beispiel kommen wir noch zur Norm (2.39), in der wir auch minimieren wollen. Dazu definieren wir die Variablen

$$u_j := (F(X)a)_j - y_j$$

und minimieren die Zielfunktion

$$\sum_{j=1}^{n} |u_j|.$$

Da diese aber nicht **linear** ist, behelfen wir uns mit demselben Trick wie oben, ersetzen u_j durch $u_j - v_j$, $u_j, v_j \geq 0$ und lösen das Minimierungsproblem

$$\min_{u_j,v_j} \sum_{j=1}^{n} u_j + v_j, \qquad (F(X)a)_j - u_j + v_j = y_j, \qquad u_j, v_j \geq 0$$

bzw.

$$\min_{u,v} \mathbf{1}^T \begin{bmatrix} u \\ v \end{bmatrix}, \qquad \begin{bmatrix} F(X) & -I & I \end{bmatrix} \begin{bmatrix} a \\ u \\ v \end{bmatrix} = y, \qquad u, v \geq 0. \tag{2.44}$$

Die Umsetzung ist fast schon erschreckend einfach:

```
octave> A = [ F -eye(15) eye(15) ]; c = [ zeros(5,1); ones( 30,1 ) ];
octave> [z,fz] = glpk( c,A,y );
```

2.5 Simulink

`Simulink` ist eine `Matlab`–Erweiterung, mit der man Systeme simulieren und untersuchen kann. Diese werden aus einfachen Blöcken via Drag-and-Drop zusammengesetzt. Beginnen wir mit einem einfachen Beispiel, nämlich einer Sinusfunktion. Nach dem Starten von `Simulink` von `Matlab` aus

```
>> simulink
```

öffnet sich ein Fenster, aus dem man eine Vielzahl seltsamer Symbole aussuchen kann:

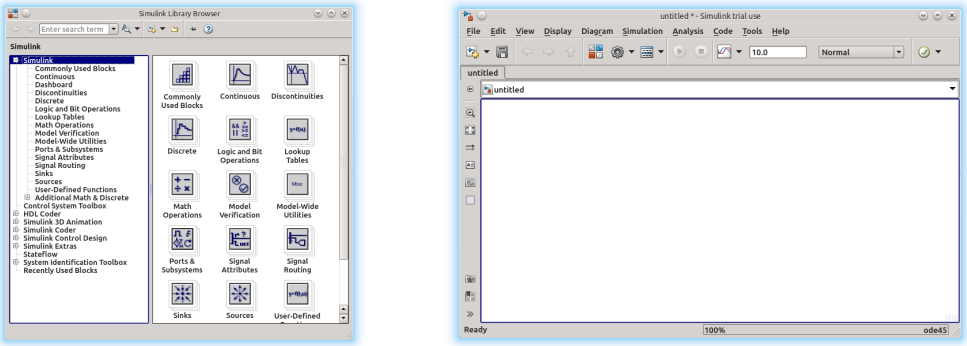


Abbildung 2.12: Startbildschirm von `Simulink` und ein leerer Bildschirm.

Beginnen wir mit einer ganz einfachen Sache, nämlich einer Sinusfunktion, die wir uns ansehen. Dazu kopieren wir aus den `Sources` eine `Sine Wave` und aus den `Sinks` ein `Scope` in das Fenster und verbinden sie. Nachdem wir das System gestartet haben, sehen wir uns das Scope durch einen Doppelklick an, siehe Abb 2.13. In der Tat: eine Sinusfunktion. Mit einem Doppelklick auf die `SineWave` kann man deren Parameter einstellen. Jetzt ein bisschen spannender:

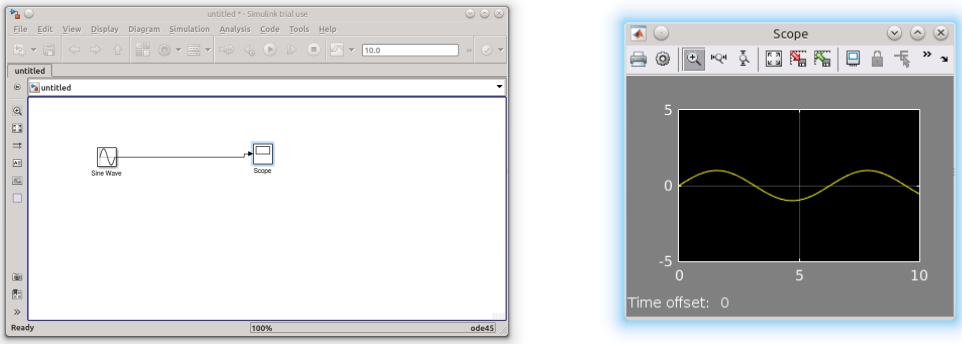


Abbildung 2.13: Einfachstes System und Ausgabe des Scope.

Wir wollen mehrere Dinge gleichzeitig mit unserem Sinus anstellen und beschaffen uns dazu einen `Integrator`, eine `Derivative` und einen `Mux`[69] Den `Mux` stellen wir dann so ein, daß er *drei* Eingänge hat und verdrahten das ganze System:

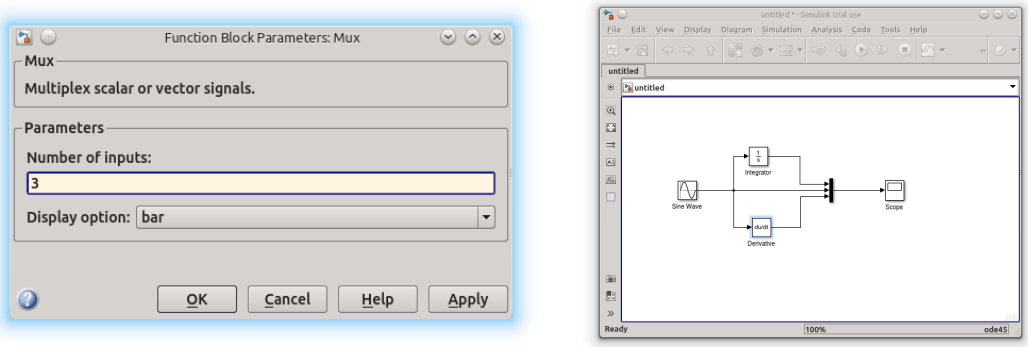


Abbildung 2.14: Einstellung des `Mux` und das Gesamtsystem. Im Scope sind dann drei Funktionen zu sehen.

Der Cosinus, den die Ableitung produziert, hat allerdings am Anfang eine kleine Delle.

Als nächstes Beispiel verwenden wir einen digitalen Filter, der verschiedene Signale verarbeitet.

[69] Aus „Signal Routing“.

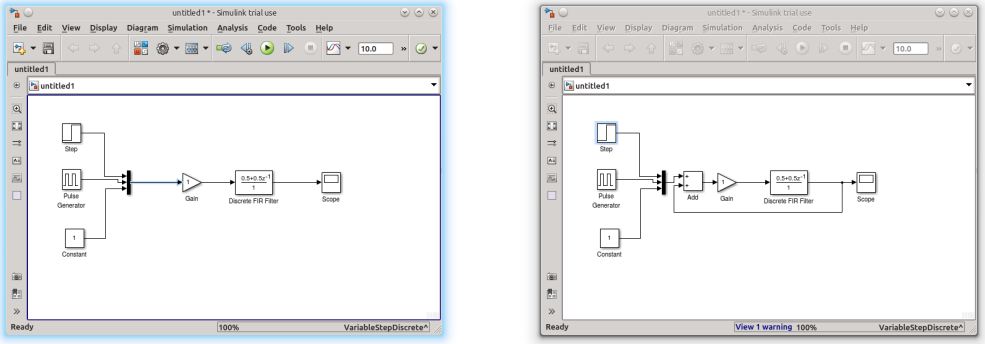


Abbildung 2.15: Filtersystem ohne und mit Rückkopplung.

Das „normale“ System filtert einfach die Daten, das andere, etwas komplexere, arbeitet mit Rückkopplung.

Zum Abschluss eine etwas „anspruchsvolle“ Anwendung, nämlich eine schwingende Feder, die durch die Differentialgleichung

$$\ddot{x} = -\frac{k}{m}x - \frac{b}{m}\dot{x} + g$$

beschrieben ist, wobei m die angebrachte Masse, k die Federkonstante, und b die Dämpfung des Systems ist. Als erstes bauen wir die Regelstrecke der Ableitungen, jede Ableitung niedrigerer Ordnung ergibt sich durch Integration aus der Ableitung höherer Ordnung. Das ergänzt man dann durch die Rückkopp-

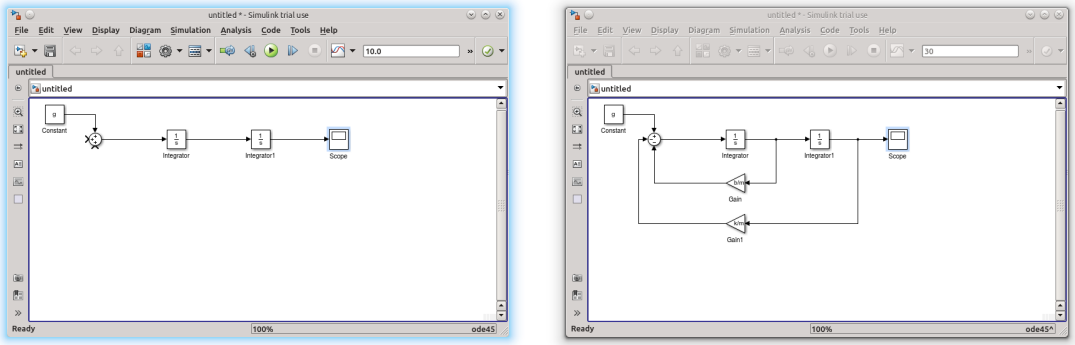


Abbildung 2.16: Das Differentialgleichungssystem und die Rückkopplungen im System.

lungen, die als `Gain` eingetragen werden, siehe Abb. 2.16. Das reicht aber noch nicht aus, die Werte der Variablen müssen ja auch noch spezifiziert werden. Das passiert im `Model Explorer`, den man über `View` aufrufen kann und in dem die Variablen eingestellt werden können, siehe Abb. 2.17. Über den Model Explo-

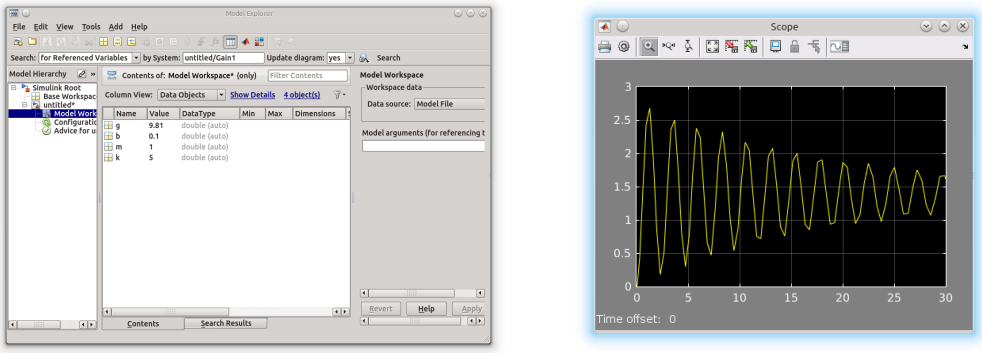


Abbildung 2.17: Der Model Explorer und das Ergebnis für eine bestimmte Einstellung.

rer kann man jetzt die Einstellungen des Systems verändern und entsprechend anderes Verhalten des schwingenden Systems bekommen.

Lieber Freund, du mußt einfach glauben; wenn du einmal zehnmal soviel Mathematik können wirst als jetzt, so wirst du verstehen, aber einstweilen: glauben!

R. Musil, *Die Verwirrungen des Zöglings Törleß*

3 Maxima

Wir kommen jetzt zur **Computeralgebra**, also zum symbolischen Rechnen, bei dem rationale Zahlen exakt in variabler Länge gespeichert werden und Ausdrücke symbolisch manipuliert werden.

3.1 Das System

`Maxima` kann auf verschiedene Arten aufgerufen werden:

1. Als Shell–Anwendung in einer Konsole, einfach mit dem Befehl `maxima`.
2. Als Subanwendung in `emacs`, für den es einen eigenen Maxima–Modus gibt.
3. Als grafische Benutzeroberfläche `wxmaxima`, die durch Verwendung der wx-Widgets auf verschiedensten Betriebssystemen, insbesondere auf Windows und Linux.

Ruft man `Maxima` in einer Shell auf, so erhält man eine Begrüßungsnachricht der Form[70]

```
Maxima 5.30.0 http://maxima.sourceforge.net
using Lisp CLISP 2.49 (2010-07-07)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1)
```

ein Sreenshot von `wxmaxima` findet sich in Abb. 3.1. Da `wxmaxima` in der Bedienung deutlich komfortabler ist und es erlaubt, mit Worksheets zu arbeiten, werden wir in der Folge im wesentlichen mit der Benutzeroberfläche arbeiten. Alle Befehle können zwar auch direkt eingegeben werden, aber die Benutzung ist etwas aufwendiger.

[70] Natürlich mit passender Versionsnummer.

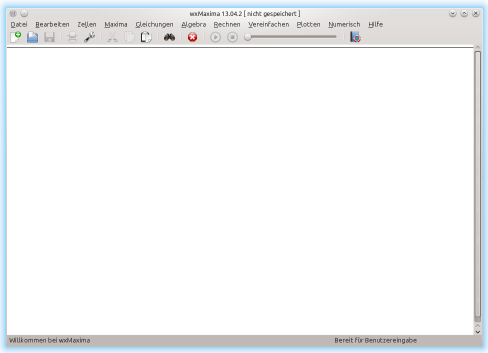


Abbildung 3.1: Sreenshot von `wxmaxima` direkt nach dem Aufruf, also mit leerem Worksheet.

3.2 Einfache Worksheets

Ein **Worksheet** in `wxmaxima` besteht aus Zellen. Eine **Zelle** enthält eine Folge von Befehlen, die in dieser Zelle auszuführen sind. Man kann also zusammenghörige Befehle in einer Zelle kombinieren. Wichtig: Eine Zelle wird nur ausgeführt, wenn man `Shift-Return` eingibt. Worksheets kann man speichern und laden und so immer wieder neu ausführen. Ausserdem kann man in Worksheets auch Textzellen einfügen und so das Worksheet kommentieren. Ein Beispiel findet

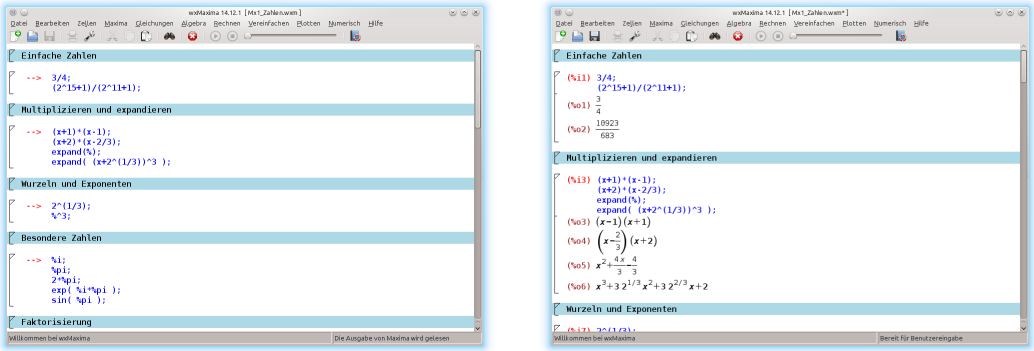


Abbildung 3.2: Ein Worksheet nach dem Laden und nach dem Ausführen der Zellen. Nach unten geht es weiter.

sich in Abb 3.2, wo man ein Worksheet vor und nach dem Ausführen sieht; das Ausführen aller Zellen ist ein einfacher Menüpunkt von `wxmaxima`. Allerdings ist das Worksheet noch deutlich länger als das, was angezeigt wird. Wir arbeiten uns nun durch Worksheets, die diverse Features von `Maxima` erklären und die auch irgendwo bei den Vorlesungsunterlagen zu finden sein sollten. Tatsächlich ist ein Worksheet nichts anderes als eine ACSII–Datei mit `Maxima`–Befehlen.

3.2.1 Zahlen und Genauigkeiten

Beschäftigen wir uns zuerst ein wenig mit Zahlen und der Darstellung symbolischer Objekte. Dazu verwenden wir das Worksheet `Mx1_Zahlen.wxm` und arbeiten uns Schritt für Schritt, genauer Zelle für Zelle, durch. Wir beginnen mit zwei ganzzahligen Rechenoperationen

```
(%i1) 3/4;
(2^15+1)/(2^11+1);
(%o1) 3/4
(%o2) 10923/683
```

Das `%i...` ist die Eingabe mit Nummer und damit auch referenzierbar, das `%o...` sind entsprechend die Ausgaben. Die Ergebnisse sind was man erwartet, man sieht, daß Brüche offensichtlich gekürzt dargestellt werden.

Symbolische Ausdrücke, also Ausdrücke mit Variablen, können ebenfalls verwendet werden, man muss das Ausmultiplizieren aber gegebenenfalls erzwingen:

```
(%i3) (x+1)*(x-1);
(x+2)*(x-2/3);
expand(%);
expand( (x+2^(1/3))^3 );
(%o3) (x-1)*(x+1)
(%o4) (x-2/3)*(x+2)
(%o5) x^2+(4*x)/3-4/3
(%o6) x^3+3*2^(1/3)*x^2+3*2^(2/3)*x+2
```

Der Befehl `expand` sorgt für das Ausmultiplizieren von Ausdrücken, mit `%` greift man auf das Ergebnis der letzten Operation zu.

Wurzeln werden ebenfalls als symbolische Ausdrücke gespeichert[71] und `Maxima` weiss, daß die Potenz einer Wurzel wieder die Ausgangszahl ergibt:

```
(%i7) 2^(1/3);
%^3;
(%o7) 2^(1/3)
(%o8) 2
```

Die komplexe Einheit i und die Zahl π sind eigene Objekte, deren Rechenregeln hinterlegt sind:

```
(%i9) %i;
%pi;
2*%pi;
exp( %i*%pi );
sin( %pi );
(%o9) %i
(%o10) %pi
```

[71] Genauer gesagt, sie werden **adjungiert**.

```
(%o11) 2*%pi
(%o12) -1
(%o13) 0
```

In der Darstellung von `wxmaxima` wird π wirklich als „π“ dargestellt, im Textmodus lediglich als `%pi`, siehe Abb. 3.3.

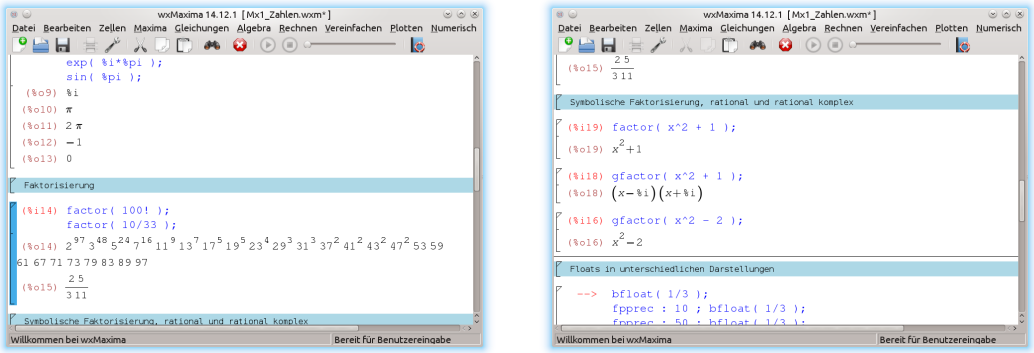


Abbildung 3.3: Spezielle Zahlen und Potenzen von Faktorisierungen in `wxmaxima` *(links)* und die Darstellung der Polynome und Faktorisierungen *(rechts)*.

Zahlen und Brüche können faktorisiert, also in Primfaktoren zerlegt werden, faktorisiert man einen Bruch, so werden Zähler und Nenner separat faktorisiert[72]:

```
(%i14) factor( 100! );
factor( 10/33 );
(%o14) 2^97*3^48*5^24*7^16*11^9*13^7*17^5*19^5*23^4*29^3*31^3
       *37^2*41^2*43^2*47^2*53*59*61*67*71*73*79*83*89*97
(%o15) (2*5)/(3*11)
```

Bei der Darstellung in `wxmaxima` werden die Exponenten eine Zeile weiter oben angezeigt, was zuerst etwas verwirren kann, siehe ebenfalls Abb 3.3.

Zur Faktorisierung von Polynomen gibt es zwei Möglichkeiten, nämlich `factor` und *gfactor*, wobei letzterer auch komplexe Zahlen mit in Betracht zieht:

```
(%i19) factor( x^2 + 1 );
(%o19) x^2+1
(%i18) gfactor( x^2 + 1 );
(%o18) (x-%i)*(x+%i)
(%i16) gfactor( x^2 - 2 );
(%o16) x^2-2
```

[72] Der Umbruch in `%o14` ist nur der Übersicht wegen *nachträglich* eingefügt.

Die Faktorisierungen sind aber *immer* rational, Wurzeln werden nicht erkannt.

Fließpunktzahlen, wie wir sie ja schon aus `Matlab` kennen, gibt es auch und zwar sogar mit beliebiger aber endlicher Genauigkeit, die mit der Variablen `fpprec` gesteuert werden kann:

```
(%i20) bfloat( 1/3 );
fpprec : 10 ; bfloat( 1/3 );
fpprec : 50 ; bfloat( 1/3 );
fpprec : 100; bfloat( 1/3 );
(%o20) 3.333333333333333b-1
(%o21) 10
(%o22) 3.333333333b-1
(%o23) 50
(%o24) 3.3333333333333333333333333333333333333333333333333b-1
(%o25) 100
(%o26) 3.3333333333333333333333333[44 digits]
       3333333333333333333333333b-1
```

Zum Abschluss noch faktorisieren wir ein ganz besonderes Polynom, nämlich des **Wilkinson–Polynom**, in verschiedenen Genauigkeiten. Zur Geschichte dahinter ist (Wilkinson, 1984) eine sehr empfehlenswerte Lektüre. Der besseren Lesbarkeit wegen exportieren wir das Ergebnis der Zelle aus `wxmaxima` als LaTeX–Ausgabe[73]:

```
(%i48) f : (x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)
           *(x-10)$
       f : f*(x-11)*(x-12)*(x-13)*(x-14)*(x-15)*(x-16)*(x-17)*(x-18)
           *(x-19)*(x-20)$
       expand( f );
       expand( bfloat(f) );
       fpprec : 10$ fb : bfloat( f )$
       expand( fb );
```

(%o50) $x^{20}-210\,x^{19}+20615\,x^{18}-1256850\,x^{17}+53327946\,x^{16}-1672280820\,x^{15}+40171771630\,x^{14}-756111184500\,x^{13}+11310276995381\,x^{12}-135585182899530\,x^{11}+1307535010540395\,x^{10}-10142299865511450\,x^9+63030812099294896\,x^8-311333643161390640\,x^7+1206647803780373360\,x^6-3599979517947607200\,x^5+8037811822645051776\,x^4-12870931245150988800\,x^3+13803759753640704000\,x^2-8752948036761600000\,x+2432902008176640000$

(%o51) $1.0b0\,x^{20}-2.1b2\,x^{19}+2.0615b4\,x^{18}-1.25685b6\,x^{17}+5.3327946b7\,x^{16}-1.67228082b9\,x^{15}+4.017177163b10\,x^{14}-7.561111845b11\,x^{13}+1.1310277b13\,x^{12}-1.355851829b14\,x^{11}+1.307535011b15\,x^{10}-1.014229987b16\,x^9+6.30308121b16\,x^8-3.113336432b17\,x^7+1.206647804b18\,x^6-3.599979518b18\,x^5+8.037811823b18\,x^4-1.287093125b19\,x^3+1.380375975b19\,x^2-8.752948037b18\,x+2.432902008b18$

(%o54) $1.0b0\,x^{20}-2.1b2\,x^{19}+2.0615b4\,x^{18}-1.25685b6\,x^{17}+5.3327946b7\,x^{16}-1.67228082b9\,x^{15}+4.017177163b10\,x^{14}-7.561111845b11\,x^{13}+1.1310277b13\,x^{12}-1.355851829b14\,x^{11}+1.307535011b15\,x^{10}-1.014229987b16\,x^9+6.30308121b16\,x^8-3.113336432b17\,x^7+1.206647804b18\,x^6-3.599979518b18\,x^5+8.037811823b18\,x^4-1.287093125b19\,x^3+1.380375975b19\,x^2-8.752948037b18\,x+2.432902008b18$

So richtig wichtig ist das Ergebnis nicht, man sollte nur erkennen, daß die Koeffizienten in der zehnstelligen Version leicht degeneriert sind. Nun kann man

[73] Auch das geht!

mit der Genauigkeit spielen und erhält recht interessante Faktorisierungen. Hier nur die Eingabe:

```
(%i55) fpprec : 11; rat( expand( bfloat( f )));
gfactor( % );
allroots( % );
```

Und zum Schluss noch eine elegantere Art, das Polynom auszurechnen, nämlich über eine Schleife:

```
--> ff : 1$ for j : 1 thru 20 do ff : ff*(x-j)$
expand( ff );
ff;
```

3.2.2 Pakete

`Maxima` ist sehr lange unterwegs und verfügt über eine aktive Community, die zu dem „Basissystem“ eine Menge von Paketen entwickelt hat, die das System massiv bereichern. Sehen wir uns ein paar Beispiele an, das Worksheet dazu ist `Mx2_Pakete.wxm`.

Das erste Paket ist `simplex`, wo das **Simplexverfahren** zur Lösung von linearen Optimierungsproblemen implementiert ist. Geladen wird das Paket folgendermaßen:

```
(%i63) load( "simplex");
(%o63) "/usr/share/maxima/5.32.1/share/simplex/simplex.mac"
```

Die Ausgabe sagt, wo das Paket gefunden wurde. Als Anwendung des Pakets lösen wir nun unser **Diätproblem** aus Beispiel 2.12. Dazu müssen wir die Zielfunktion beschreiben und die Nebenbedingungen in einer **Liste**, gekennzeichnet durch $[\ldots]$, angeben:

```
(%i64) minimize_lp( 3.8*x + 4.2*y,
[ .1*x + .25*y >= 1, x+.25*y >= 5,
110*x+120*y >= 400 ]  );
(%o64) [26.22222222222222,[y=2.222222222222221,x=4.444444444444445]]
```

Als Ausgabe erhalten wir den extremalen, in diesem Falle minimalen, Wert der Zielfunktion und die zugehörigen Paramterwerte. Eigentlich recht komfortabel, oder?

Ein etwas anspruchsvolleres Beispiel sind **orthogonale Polynome**. Das ist eine Familie von Polynomen $p_j \in \Pi_j$ mit der Eigenschaft

$$\int_I p_j(x)\, p_k(x)\, w(x)\, dx = 0, \qquad k < j.$$

Die Polynome hängen natürlich vom Interval $I \subset \mathbb{R}$ und der **Gewichtsfunktion** w ab. Ist $I = [-1, 1]$ und $w = 1$, so spricht man von einem **Legendrepolynom**. Diese kann man nun automatisch generieren lassen, hier die Varianten vom Grad 3 und 5.

```
-->  load( "orthopoly" )$
p3 : legendre_p( 3,x )$
p5 : legendre_p( 5,x )$
allroots( p5 );
expand( legendre_p (4,x ) );
```

Dazu lassen wir über die Funktion `allroots` die Nullstellen eines solchen Polynoms berechnen und stellen voller Begeisterung fest, daß sie alle rell sind und im Intervall $[-1, 1]$ liegen. Das stimmt auch mit der Theorie überein, siehe (Gautschi, 1997; Sauer, 2013). Die Normalisierung ist so gewählt, daß die Polynome an der Stelle 1 den Wert 1 haben, wofür es eine Menge von guten Gründen gibt, siehe (Askey, 1975). Die Integrationseigenschaft kann man auch exemplarisch verifizieren:

```
(%i25) integrate(p3*p5, x, -1, 1);
integrate( p5*p5,x,-1,1 );
(%o25) 0
(%o26) 2/11
```

3.2.3 Plots

Als nächstes sehen wir uns die grafischen Fähigkeiten von `Maxima` an. Per se hat das Programm eigentlich gar keine Grafik, es nutzt aber sehr geschickt[74] die Möglichkeiten von `gnuplot`. Dennoch gibt es eine sehr einfache Syntax für Plots, was im Worksheet `Mx3_Plot.wxm` vorgeführt wird. Wir beginnen mit einem ganz einfachen Plot:

```
(%i27) plot2d( sin(x)/x, [x,-5*%pi,5*%pi] );
plot2d: expression evaluates to non-numeric value somewhere in
        plotting range.
(%o27) "/home/tomas/maxout.gnuplot_pipes"
```

Und in der Tat öffnet sich ein Fenster mit einem Plot der **sinc–Funktion** $f(x) = \frac{\sin x}{x}$ auf dem Intervall $[-5, 5]$, siehe Abb 3.4; die Fehlermeldung bedeutet, daß einmal durch Null geteilt wurde (an $x = 0$), was aber nicht weiter negativ auffällt. Wir können uns auch sinc nochmals als Funktion definieren und dann diese plotten lassen, das grafische Ergebnis ist dasselbe:

```
-->  sinc(x) := sin(x)/x;
plot2d( sinc(x), [x,-5*%pi,5*%pi] );
```

Plots sind auch in 3D möglich, der einfachste Aufruf ist

```
(%i29) plot3d( sin(x)*sin(y)/(x*y), [x,-5*%pi,5*%pi],
[y,-5*%pi,5*%pi] );
(%o29) "/home/tomas/maxout.gnuplot_pipes"
```

[74] Sogar deutlich geschickter als `Octave`.

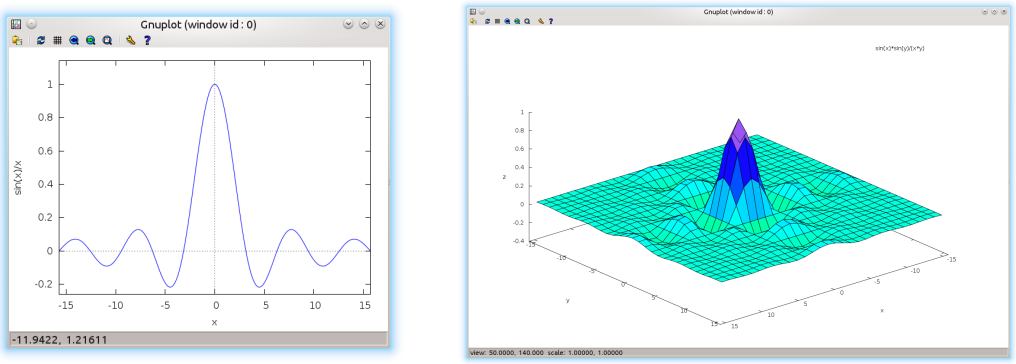


Abbildung 3.4: Plots der sinc–Funktion *(links)* und deren Tensorprodukt *(rechts)*

mit der hübschen, ebenfalls in Abb. 3.4 zu sehenden Funktion. Es gibt darüberhinaus eine Vielzahl von grafischen Methoden; man kann mit `grid` die Auflösung erhöhen:

```
--> plot3d( sin(x)*sin(y)/(x*y), [x,-5*%pi,5*%pi],
[y,-5*%pi,5*%pi], [grid,100,100] );
```

radiale Versionen der Funktion mit oder ohne Linien plotten lassen, siehe Abb 3.5:

```
--> plot3d( sin(sqrt(x^2+y^2))/sqrt(x^2+y^2),
[x,-5*%pi,5*%pi],[y,-5*%pi,5*%pi], [grid,60,60] );
--> nrm(x,y) := sqrt( x^2+y^2 )$
plot3d( sinc(nrm(x,y)), [x,-5*%pi,5*%pi],[y,-5*%pi,5*%pi],
[grid,60,60], [mesh_lines_color,false] );
```

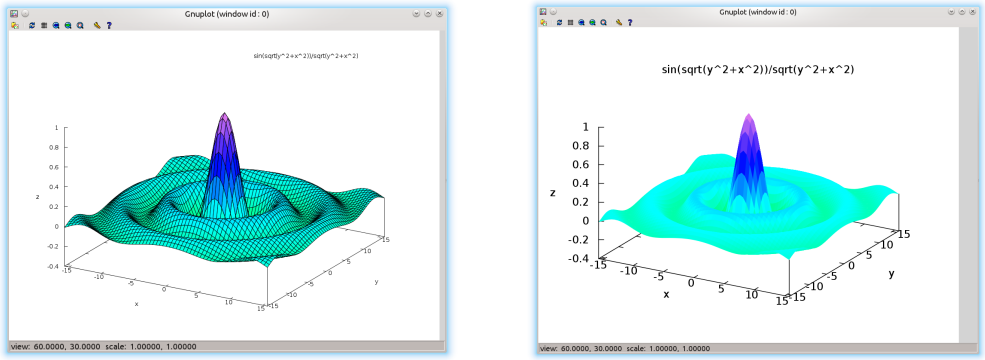


Abbildung 3.5: Radialer sinc mit und ohne Linien.

und es gibt sehr leistungsfähige Routinen für sehr komplexe Funktionen und Fraktale, die sich ebenfalls im Worksheet finden lassen.

3.3 Polynome

Eine der großen Stärke des symbolischen Rechnens ist der formale Umgang mit Polynomen als Funktionen und die Möglichkeit der Durchführung *algebraischer* Operationen im Ring $\Pi = \mathbb{Q}[x]$ der Polynome mit *rationalen* Koeffizienten.

Polynome werden in der sogenannten **CRE–Form** gespeichert, siehe (Schelter, 2001), bei der es einen Hierarchie zwischen den Variablen gibt, wobei man ein Polynom als Polynom in der wichtigsten Variablen betrachtet, dessen Koeffizienten Polynome in den unwichtigen Variablen sind, was gegebenenfalls rekursiv aufgelöst wird. Die „Wichtigkeit" einer Variablen bestimmt sich dabei in aufsteigender alphabetischer Reihenfolge.

Beispiel 3.1 *Das Polynom* $xy^2 + 2xy + y^2 + 3x + 2y + 1$ *wird als*

$$y^2(x+1) + y(2x+2) + y(3x+1)$$

gespeichert.

3.3.1 Manipulation und Faktorisierung

Fangen wir mal an, ein paar *univariate* Polynome zu definieren:

```
(%i8) p1 : x^3-1; p2 : (x+2)*(x-1);
p3 : x^2 - 9/4; p4 : x^2 + 9/4;
(%o8) x^3-1
(%o9) (x-1)*(x+2)
(%o10) x^2-9/4
(%o11) x^2+9/4
```

Wie man sieht, werden die Polynome so gespeichert, wie sie eingegeben wurden, also p_2 in faktorisierter Form, die anderen in ausmultiplizierter Form. Das kann man ändern, indem man die Polynome explizit faktorisiert bzw. ausmultipliziert:

```
(%i12) factor( p1 );
expand( p2 );
factor( p3 );
factor( p4 );
(%o12) (x-1)*(x^2+x+1)
(%o13) x^2+x-2
(%o14) ((2*x-3)*(2*x+3))/4
(%o15) (4*x^2+9)/4
```

Das sieht soweit ganz gut aus, nur bei p_4 hat es nicht geklappt. Was man allerdings sieht ist, daß bei der Faktorisierung der Hauptnenner nach aussen gezogen wurde. Das ist kein Zufall, denn die meisten Faktorisierungsalgorithmen für Polynome funktionieren über $\mathbb{Z}[x]$, also für **Ganzzahlpolynome**, siehe (Gathen & Gerhard, 1999). Will man eine Faktorisierung über den Gaußschen Ganzzahlen, also $\mathbb{Z} + i\mathbb{Z}$, dann verwendet man das Kommando `gfactor`:

```
(%i16) gfactor( p4 );
(%o16) ((2*x-3*%i)*(2*x+3*%i))/4
```

Allerdings funktioneren Faktorisierungen nur rational bzw. über $\mathbb{Z}$, was ja nach obiger Bemerkung ohnehin dasselbe ist:

```
(%i63) factor( x^2-2 );
(%o63) x^2-2
```

Man kann auch über $\mathbb{Q}[y]$ faktorisieren, wobei y als Minimalpolynom den Ausdruck p hat, also $p(y) = 0$. Das sieht beispielsweise folgendermaßen aus:

```
(%i80) factor( x^4+1 );
(%o80) x^4+1

(%i81) factor( x^4+1, a^2-2 ); expand( % );
(%o81) (x^2-a*x+1)*(x^2+a*x+1)
(%o82) x^4-a^2*x^2+2*x^2+1

(%i83) expand( (x^2-sqrt(2)+1) * (x^2+sqrt(2)+1 ) );
(%o83) x^4+2*x^2-1
```

Wir dürfen also nicht einfach annehmen, daß $a = \sqrt{2}$ ist, denn die mittlere Gleichung funktioniert eben nur mit $a^2 = 2$, wenn wir direkt $\sqrt{2}$ einsetzen, dann funktionierte die Faktorisierung nicht. Noch ein schönes Beispiel, bei dem wir ein paar neue Befehle kennenlernen:

```
(%i84) ratsimp( (x^5-1)/(x-1) );
subst( a,x,% );
factor( %th(2), % );
(%o84) x^4+x^3+x^2+x+1
(%o85) a^4+a^3+a^2+a+1
(%o86) (x-a)*(x-a^2)*(x-a^3)*(x+a^3+a^2+a+1)
```

Mit `ratsimp` vereinfachen wir eine rationale Division, mit `%th(k)` verwendet man die Ausgabe der k–letzten Operation[75] und `subst` führt eine Subsitution durch, und zwar wird das zweite Argument im dritten durch das erste ersetzt; das klingt so kompliziert, daß ein Beispiel nicht schlecht ist:

```
(%i97) subst( 2*x,x,p3 );
subst( (x+2)^2,x,p1 ); expand( % );
(%o97) 4*x^2-9/4
(%o98) (x+2)^6-1
(%o99) x^6+12*x^5+60*x^4+160*x^3+240*x^2+192*x+63

(%i118) subst( 1/x,x,p2 );ratsimp(%);
(%o118) (1/x-1)*(1/x+2)
(%o119) -(2*x^2-x-1)/x^2
```

[75] Also `%th(1)` = `%`, `%th(2)` das Ergebnis der vorletzten Operation und so weiter.

Faktorisierungen können auch in $\mathbb{Z}_p$ also modulo p durchgeführt werden, was insbesondere in der Kryptographie sehr hilfreich ist:

```
(%i108) p : x^6+x^2+1$ factor( p );
(%o109) x^6+x^2+1

(%i110) modulus : 13$ factor( p );
modulus : false$ factor( p );
(%o111) (x^2+6)*(x^4-6*x^2-2)
(%o113) x^6+x^2+1
```

Polynome kann man auch mit abstrakten Koeffizienten definieren:

```
(%i55) p : 1$ for j : 1 thru 4 do p : p*(x-c[j])$
p;
expand(p);
ratsimp(p);
```

Diese Koeffizienten kann man auch einem Ausdruck wiederfinden, allerdings muss der zu diesem Zweck erst expandiert werden:

```
(%i60) coeff( p,x,3 ); coeff( expand(p),x,3 );
(%o60) 0
(%o61) -c[4]-c[3]-c[2]-c[1]
```

Polynome kann man addieren, mulitplizieren und weiterverarbeiten:

```
(%i67) p2+p3; p3*p4-p1; factor( p1-p2);
(%o67) x^2+(x-1)*(x+2)-9/4
(%o68) -x^3+(x^2-9/4)*(x^2+9/4)+1
(%o69) (x-1)^2*(x+1)
```

Zum Vereinfachen von Ausdrücken kann man, wie schon gesehen, `ratsimp` verwenden. Allerdings ist das beim ersten Versuch nicht immer vollständig erfolgreich, weswegen es eine Funktion `fullratsimp` gibt, die so lange iterativ vereinfacht[76], bis wirklich keine Verbesserung mehr zu erzielen ist. Ein Beispiel:

```
(%i100) expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
(%o100) ((x^(a/2)-1)^2*(x^(a/2)+1)^2)/x^a-1

(%i101) ratsimp( expr );
(%o101) (x^(2*a)-2*x^a+1)/x^a-1

(%i102) fullratsimp( expr );
(%o102) x^a-1
```

Einen ähnlichen Effekt gibt es auch für Substitutionen, allerdings sind die Funktionen im Paket lrats enthalten:

```
(%i105) load(lrats)$ ratsubst (b*a, a^2, a^3);
fullratsubst (b*a, a^2, a^3);
(%o106) a^2*b
(%o107) a*b^2
```

[76] Mit einer Mischung aus rationalen und anderen Vereinfachungen

3.3.2 Divisionen und Reste

Polynome formen bekanntlich einen **Ring**, in dem man multiplizieren darf, was implizit auch eine Division, zumindest mit Rest liefert. Als **euklidischer Ring** erlauben die Polynome eine Division mit Rest, die sogenannte **euklidische Division** für $f, g \in \Pi$:

$$f = q\,g + r, \qquad \deg r < \deg g.$$

Dies Division wird in `Maxima` durch den Befehl `divide` umgesetzt, der eine **Liste** von zwei Elementen erzeugt, die den Quotienten `q` und den Rest `r` enthält:

```
(%i48) divide( p1,p3 ); rem : %[2];
(%o48) [x,(9*x-4)/4]
(%o49) (9*x-4)/4
```

Zugriff auf Listenelemente geschieht durch „[...]“.

Ein wichtiges Objekt in euklidischen Ringen ist der größte gemeinsame Teiler von zwei Polynomen oder auch nur von zwei Zahlen. Dafür haben wir den `gcd`

```
(%i60) gcd( 27,15 );
(%o60) 3
```

und eine erweiterte Version, den `gcdex`:

```
(%i61) gcdex( 27,15 ); %[1]*27+%[2]*15;
(%o61) [-1,2,3]
(%o62) 3
```

Der `gcd` ist klar, aber was berechnet die andere Funktion? Das ist die Lösung der sogenannten **Bézout–Identität**, die in beliebigen Ringen gilt, nämlich zu $f, g \in R$ die Bestimmung von $p, q \in R$ so daß

$$f\,p + g\,q = \gcd(f, g).$$

Und in der Tat zeigt das obige Beispiel ja, daß genau das passiert. Das kann man ausnutzen um modulo Primzahl p zu invertieren:

$$\gcd(a, p) = 1 \qquad \Leftrightarrow \qquad ax + py = 1 \qquad \Leftrightarrow \qquad a \equiv_p x^{-1}$$

Oder, in `Maxima`, die Berechnung von $1/7$ modulo 13:

```
(%i112) gcdex( 7,13 )$ b : %[1]; mod( b*7,13 );
(%o113) 2
(%o114) 1
```

Das geht auch mit Polynomen, und zwar sowohl der `gcd` als auch die Bezout–Identität:

```
(%i60) gcdex( p1,p2 ); %[1]*p1+%[2]*p2;
(%o60)/R/ [1,-x+1,3*x-3]
(%o61)/R/ 3*x-3
```

In der Tat, der größte gemeinsame Teiler ist $3x - 3$. Man kann den größten gemeinsamen Teiler nutzen, um **mehrfache Nullstellen** eines Polynoms zu eliminieren. Hierzu bemerkt man, daß aus

$$p(x) = (x - \xi)^k q(x), \qquad q(\xi) \neq 0, \quad k > 1,$$

auch

$$\begin{aligned} p'(x) &= k(x - \xi)^{k-1} q(x) + (x - \xi)^k q'(x) = (x - \xi)^{k-1} \left(kq(x) + (x - \xi)q'(x)\right) \\ &=: (x - \xi)^{k-1} \tilde{q}(x) \end{aligned}$$

mit $\tilde{q}(\xi) = kq(\xi) \neq 0$ folgt. Damit besteht der größte gemeinsame Teiler von p und p' gerade aus den Potenzen der mehrfachen Nullstellen und

$$\tilde{p} = \frac{p}{\gcd(p, p')}$$

het dieselben Nullstellen wie p, nur einfach. Das kann die Nullstellensuche massiv vereinfachen. In `Maxima` sieht die Sache dann wie folgt aus:

```
(%i62) p : 4*x^4 + 4*x^3 - 3*x^2 - 4*x - 1$
factor( p );
sqfr( p );
factor( ratsimp( p / gcd( p, ratdiff(p,x) ) ) );
(%o63) (x-1)*(x+1)*(2*x+1)^2
(%o64) (2*x+1)^2*(x^2-1)
(%o65) (x-1)*(x+1)*(2*x+1)
```

3.3.3 Etwas mehr Algebra

Eine etwas spannendere und anspruchsvolle Sache ist die **Resultante** von zwei Polynomen, ein weiterer Bezug zwischen Linearer und Kommutativer Algebra.

Definition 3.2 *Zu zwei Polynomen*

$$p(x) = \sum_{j=0}^{m} p_j x^j, \qquad q(x) = \sum_{j=0}^{n} q_j x^j,$$

vom Grad m *und* n *ist die* ***Sylvestermatrix*** $S(p, q)$ *definiert als die quadratische Matrix*

$$S(p, q) := \begin{bmatrix} p_m & \cdots & p_0 & & \\ & \ddots & \vdots & \ddots & \\ & & p_m & \cdots & p_0 \\ q_n & \cdots & q_1 & q_0 & \\ & \ddots & \vdots & \vdots & \ddots \\ & & q_n & q_{n-1} & \cdots & q_0 \end{bmatrix} \in \mathbb{R}^{m+n \times m+n}, \tag{3.1}$$

in der die Zeile mit den Koeffizienten von p *insgesamt* $\deg q$*–mal, die mit den Koeffizienten von* q *insgesamt* $\det p$*–mal wiederholt wird. Die Determinante der Sylvestermatrix bezeichnet man als* ***Resultante*** *von* p *und* q*:*

$$R(p, q) := \det S(p, q). \tag{3.2}$$

Eine zentrale Eigenschaft der Resultante, die man in der Computeralgebra gerne nutzt ist, daß sie genau dann gleich Null ist, wenn p und q einen gemeinsamen Faktor, also einen nichttrivialen gcd besitzen. Testen wir es.

```
(%i66) resultant( p1,p2,x);
resultant( p1,p3,x );
(%o66) 0
(%o67) -665
```

bzw.

```
(%i68) resultant(2*x^2+3*x+1, 2*x^2+x+1, x);
resultant((x+1)*x, (x+1), x);
(%o68) 8
(%o69) 0
```

Diesen Ansatz kann man nutzen, um in parametrisierten Polynomen die Parameter so zu bestimmen, daß gemeinsame Faktoren auftreten. Hier ein einfaches Beispiel mit einem quadratischen und einem linearen Polynom, das eine der beiden Nullstellen des quadratischen Polynoms teilen soll:

```
(%i70) remvalue(all);
resultant(a*x^2+b*x+1, c*x + 1, x);
solve( %,c );
(%o70) [p1,p2,p3,p4,p,expr,messlrats2,fullratsubstflag,rem,b]
(%o71) c^2-b*c+a
(%o72) [c=-sqrt(b^2-4*a)-b/2,c=(sqrt(b^2-4*a)+b)/2]
```

Das Ergebnis ist natürlich die wohlbekannte Formel für die Nullstellen von quadratischen Polynomen.

Ein weiteres Hilfsmittel ist die **Elimination** von Variablen aus multivariaten Polynomen, also Polynomen in $x, y, z, \ldots$. Die intuitive Idee hierbe ist, daß man, wie im linearen Fall, eine Gleichung nach einer Variablen auflöst und die Ergebnisse in die anderen Gleichungen substituiert, die dann eine Variable weniger. Das ist natürlich etwas naiv gedacht und nicht so einfach, was dahinter steckt, sind sogenannte **Eliminationsideale**, siehe (Cox *et al.*, 1996; Sauer, 2001). Trotzdem kann man es mal versuchen und die Aufrufsyntax ist einfach. Hier ein Beispiel in drei Variablen:

```
(%i73) expr1: 2*x^2 + y*x + z$
expr2: 3*x + 5*y - z - 1$
expr3: z^2 + x - y^2 + 5$
el : eliminate ([expr3, expr2, expr1], [y, z]);
(%o76) [7425*x^8-1170*x^7+1299*x^6+12076*x^5+22887*x^4-5154*x^3
       -1291*x^2+7688*x+15376]
```

Die Elimination hat uns also ein Polynom in einer Variablen beschert, dessen Nullstellen wir nun alle bestimmen könnten, dann wieder in die drei Gleichungen einsetzen würden und dann haben wir nur noch drei Gleichungen in zwei Variablen und so weiter. Prinzipiell ist das dann ein Verfahren, um ein **polynomiales Gleichungssystem** in mehreren Variablen zu lösen. Machen wir's:

```
(%i77) solve( el[1] = 0,x );
allroots( el[1] );
solve( [expr1,expr2,expr3], [x,y,z]);
(%o77) ...
```

Die „Lösung“ lasse wir aber gleich weg, denn der Ausdruck ist ziemlich eindrucksvoll und schwer zu interpretieren. Das ist dann leider das praktische Problem mit den Eliminationssachen.

3.3.4 Etwas andere Brüche

Jetzt noch zwei andere Bruchkonzepte, mit denen man eine Menge anfangen kann. Da beide prinzipiell auf euklidischen Ringen funktionieren, sind sie auch für Polynome geeignet und werden in der Signalverarbeitung auch auf Polynome angewandt.

Definition 3.3 (Kettenbruch) *Ein **Kettenbruch** ist eine rationale Zahl der Form*

$$[a_0; a_1, \ldots, a_n] := a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}}} \tag{3.3}$$

mit Koeffizienten $a_j \in \mathbb{N}$.

Generell kann man Kettenbrüche über beliebigen euklidischen Ringen definieren, also Ringen, in denen es eine Division mit Rest gibt. Das sieht man am besten, wenn man sich ansieht, wie die **Kettenbruchentwicklung** eines rationalen Objekts

$$x := \frac{p}{q}, \qquad p, q \in R$$

für einen euklidischen Ring R bestimmt wird. Dazu bilden wir die Division mit Rest

$$p = a_0 q + r, \qquad r \prec q$$

und erhalten

$$x = \frac{p}{q} = \frac{a_0 q + r}{q} = a_0 + \frac{r}{q} = a_0 + \frac{1}{\frac{q}{r}} =: a_0 + \frac{1}{\frac{p_1}{q_1}},$$

so daß wir nur p_1/q_1 weiter entwickeln müssen. Mit der Iteration

$$p_0 = p,\ q_0 = q, \qquad p_{j+1} = q_j,\ q_{j+1} = p_j - a_j q_j,$$

erhalten wir so die Kettenbruchentwicklung. Wenn man genau hinschaut, steckt darin ein **eulidischer Algorithmus** für den gcd und deswegen gilt die folgende Aussage.

Satz 3.4 *Eine Zahl* x *ist genau dann rational, wenn sie eine endliche Kettenbruchentwicklung hat.*

Kettenbrüche gibt es auch in `Maxima`, die Kettenbruchentwicklung wird mit `cf` bestimmt. Mit `cfdisp` kann man den Kettenbruch anzeigen lassen, mit `cfexpand` die zugehörige rationale Zahl ausgeben:

```
(%i78) a : cf( 7/13 );
cfdisrep( % );
cfexpand( a );
(%o78) [0,1,1,6]
(%o79) 1/(1+1/(1+1/6))
(%o80) matrix([7,1],[13,2])
```

Die erste Spalte von `cfexpand` sind Zähler und Nenner des Kettenbruchs, die zweite Spalte die vorherige **Konvergente**.

Definition 3.5 *Die* k*–te* ***Konvergente*** *des (endlichen oder unendlichen) Kettenbruchs* $[a_0; a_1, \dots]$ *ist* $[a_0; a_1, \dots, a_k]$.

Sehen wir uns das bei unserem Beispiel

$$\frac{7}{13} = [0; 1, 1, 6] = 0 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{6}}}$$

an, da sind die Konvergenten

$$[0;] = 0, \qquad [0; 1] = \frac{1}{1} = 1, \qquad [0; 1, 1] = \cfrac{1}{1 + \cfrac{1}{1}} = \frac{1}{2}, \qquad [0; 1, 1, 6] = \frac{7}{13}.$$

Da unendliche Kettenbrüche schwer zu speichern sind, muss man bei irrationalen Zahlen die Kettenbruchentwicklung rechtzeitig abbrechen, was man über die Variable `cflength` steuert. Hier ein Beispiel mit dem goldenen Schnitt, der übrigens die Entwicklung $[1; 1, 1, \dots]$ hat, was ihn zur irrationalsten Zahl macht, siehe (Khinchin, 1964; Sauer, 2005):

```
(%i81) cflength : 1$ a1 : cf ((1 + sqrt(5))/2);
cflength : 2$ a2: cf ((1 + sqrt(5))/2);
cflength : 10$ a3 : cf ((1 + sqrt(5))/2);
(%o82) [1,1,1,1,2]
(%o84) [1,1,1,1,1,1,1,2]
(%o86) [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
        1,1,1,1,1,2]
```

Algebraische Zahlen, also insbesondere Wurzeln, haben *periodische* Kettenbruchentwiklungen, was man bei $\sqrt{13}$ schön sehen kann:

```
(%i87) cflength : 1$ a1 : cf (sqrt(13));
cflength : 2$ a2: cf (sqrt(13));
cflength : 10$ a3 : cf (sqrt(13));
(%o88) [3,1,1,1,1,6]
(%o90) [3,1,1,1,1,6,1,1,1,1,6]
(%o92) [3,1,1,1,1,6,1,1,1,1,6,1,1,1,1,6,1,1,1,1,6,1,1,1,1,6,
        1,1,1,1,6,1,1,1,1,6,1,1,1,1,6,1,1,1,1,6,1,1,1,1,6]
```

Das Besondere an Kettenbrüchen ist, daß die Konvergenten die Zahl sehr gut approximieren und zwar mit einem Fehler

$$\left|x - \frac{p}{q}\right| \leq \frac{1}{2q^2}, \qquad \frac{p}{q} = [a_0; a_1, \ldots, a_k]. \tag{3.4}$$

Genau genommen gilt dies nur für mindestens eine von zwei aufeinanderfolgenden Konvergenten, aber jeder Bruch, der relativ zu seinem Nenner so genau approximiert wie in (3.4), *muss* eine Konvergente sein, also besser als Konvergente geht nicht. Hier ein paar Beispiele, was das bedeutet:

```
(%i93) cfexpand(a1 ); %[1,1] / %[2,1], numer;
cfexpand( a2 ); %[1,1] / %[2,1], numer;
cfexpand( a3 );%[1,1] / %[2,1], numer;
sqrt(13),numer;
(%o93) matrix([119,18],[33,5])
(%o94) 3.6060606060606
(%o95) matrix([4287,649],[1189,180])
(%o96) 3.605550883095038
(%o97) matrix([12168897861570447,1842222905266249],
       [3375045015828949,510940703520900])
(%o98) 3.605551275463989
(%o99) 3.605551275463989
```

Wir können das Kettebruchspiel auch mit π versuchen. Zuerst die Entwicklung:

```
(%i100) a : cf( rat( ev(%pi,numer) ));
rat: replaced 3.141592653589793 by 80143857/25510582 = 3.141592653589792
(%o100) [3,7,15,1,292,1,1,1,2,1,3,1,14]
```

Man sieht hier allerdings, daß nicht wirklich π, sondern nur eine rationale Näherung entwickelt wird. Trotzdem ist die Approximation gut:

```
(%i101) alen : length(a)$
for i : 1 thru alen do display( cfexpand( rest(a,i-alen  )) );
cfexpand([3])=matrix([3,1],[1,0])
cfexpand([3,7])=matrix([22,3],[7,1])
cfexpand([3,7,15])=matrix([333,22],[106,7])
cfexpand([3,7,15,1])=matrix([355,333],[113,106])
cfexpand([3,7,15,1,292])=matrix([103993,355],[33102,113])
cfexpand([3,7,15,1,292,1])=matrix([104348,103993],[33215,33102])
```

```
cfexpand([3,7,15,1,292,1,1])=matrix([208341,104348],[66317,33215])
cfexpand([3,7,15,1,292,1,1,1])=matrix([312689,208341],[99532,66317])
cfexpand([3,7,15,1,292,1,1,1,2])=matrix([833719,312689],[265381,99532])
cfexpand([3,7,15,1,292,1,1,1,2,1])=matrix([1146408,833719],
   [364913,265381])
cfexpand([3,7,15,1,292,1,1,1,2,1,3])=matrix([4272943,1146408],
   [1360120,364913])
cfexpand([3,7,15,1,292,1,1,1,2,1,3,1])=matrix([5419351,4272943],
   [1725033,1360120])
cfexpand([3,7,15,1,292,1,1,1,2,1,3,1,14])=
   matrix([80143857,5419351],[25510582,1725033])
(%o102) done
```

Oder, ein bisschen schöner, die Konvergenten von π:

```
(%i103) alen : length(a)$
for i : 1 thru alen do
(
tmp : cfexpand( rest(a,i-alen)),
tmp : tmp[1,1]/tmp[2,1],
disp( tmp ), disp( ev(tmp,numer))
);
3
3
22/7
3.142857142857143
333/106
3.141509433962264
355/113
3.141592920353982
103993/33102
3.141592653011902
104348/33215
3.141592653921421
208341/66317
3.141592653467436
312689/99532
3.141592653618936
833719/265381
3.141592653581078
1146408/364913
3.141592653591404
4272943/1360120
3.141592653589389
5419351/1725033
3.141592653589815
80143857/25510582
```

```
3.141592653589792
(%o104) done
```

Das zweite Objekt sind die Partialbrüche, die in der Signalverarbeitung eine wichtige Rolle bei der Behandlung von rationalen Filtern darstellen. Die Funktion hierfür ist `partfrac`. Hier zwei Beispiele:

```
(%i105) partfrac((z^2+2*z+2)/(z^2-2*z+1), z);
(%o105) 4/z-1+5/(z-1)^2+1
```

Formal verbirgt sich dahinter der folgende Satz.

Satz 3.6 *Sei* $f = p/q$ *eine rationale Funktion und seien* $\zeta_1, \dots \zeta_m \in \mathbb{C}$ *die Nullstellen von* q *mit Vielfachheiten* $\mu_1, \dots, \mu_m$*, also*

$$q = \prod_{j=1}^{m} (\cdot - \zeta_j)^{\mu_j} .$$

Dann gibt es Polynome $p_1, \dots, p_m$ *mit* $\deg p_j < \mu_j$*, so daß*

$$f(z) = \sum_{j=1}^{m} \frac{p_j(z)}{(z-\zeta_j)^{\mu_j}}, \qquad z \in \mathbb{C}. \tag{3.5}$$

Hat insbesondere q nur **einfache Nullstellen**, dann besteht die Partialbruchentwicklung nur aus Reziprokwerten linearer Funktionen:

```
(%i106) partfrac( (z^4+13*z^3-5*z^2+z-1)/((z-1)*(z-2)*(z-3)*(z-4)),z);
(%o106) -3/(2*(z-1))+101/(2*(z-2))-389/(2*(z-3))+337/(2*(z-4))+1
```

Hinter allem steckt wieder einmal ein **euklidischer Algorithmus**, was dafür sorgt, daß bei Zähler- und Nennerpolynom mit rationalen Koeffizienten und Nulltstellen[77] auch die Koeffizienten in der Partialbruchentwicklung rational sind

3.4 Analysis

Auch für analytischen Kalkül, also den Umgang mit Funktionen, bietet `Maxima` einiges an Funktionalität, um Funktionen dazustellen, abzuleiten und zu integrieren.

Um mit Funktionen arbeiten zu können, muss man diese zuerst einmal formal definieren:

```
(%i107) f : sin(x)*x + 13*x^2-7/x;
(%o107) x*sin(x)+13*x^2-7/x
```

Will man nun diesen Ausdruck differenzieren und integrieren so muss man natürlich sagen, was hier die Variable ist, in unserem Fall `x`

[77] Zumindest im Nenner.

```
(%i108) diff( f,x );
integrate( %,x );
(%o108) sin(x)+x*cos(x)+26*x+7/x^2
(%o109) x*sin(x)+13*x^2-7/x
```

Integrale gibt es auch in bestimmt, allerdings können bei symbolischer Rechnung die Werte ziemlich uninterpretierbar werden, weswegen man sich oftmals mit der numerischen Auswertung über den Quantor `numer` behelfen muss:

```
(%i110) integrate(f,x,1,2);
%,numer;
(%o110) (3*sin(2)-21*log(2)-6*cos(2)-3*sin(1)+3*cos(1)+91)/3
(%o111) 26.92172549039392
```

Der Befehl `%,numer` wertet hierbei den letzten Ausdruck **numerisch** aus und gibt die entsprechende Näherung zurück.

3.4.1 Grenzwerte

`Maxima` kann Grenzwerte berechnen, wobei man Funktion, Variable und Grenzwert der Variablen angeben muss,

```
(%i112) limit( x*exp(-x),x,0 );
limit( x*exp(-x),x,inf);
limit( x^99*exp(-x),x,inf);
limit( x*exp(-x),x,minf);
(%o112) 0
(%o113) 0
(%o114) 0
(%o115) -inf
```

und kennt auch Tricks wie die **l'Hospital–Regel**[78]

```
(%i116) limit( (exp(x)-1)/x,x,0 );
(%o116) 1
```

Man kann auch einseitige, gerichtete, Grenzwert bestimmen lassen,

```
(%i117) limit( abs(x)/x,x,0,plus);
limit( abs(x)/x,x,0,minus);
(%o117) 1
(%o118) -1
```

was aber nicht immer korrekt klappt:

```
(%i119) limit(x*log(x),x,0,plus);
limit(x*log(x),x,0,minus);
(%o119) 0
(%o120) 0
```

[78] Sooo schwer ist die ja dann auch wieder nicht.

Das zweite Ergebnis ist nicht ganz intuitiv, weil der Logarithmus für negative Werte ja nicht so richtig definiert ist, aber noch korrekt, weil hier dann die komplexe Erweiterung verwendet wird. Aber: es kann auch richtig falsch werden:

```
(%i121) limit( (-2)^x,x,inf);
limit( (-2)^x/x,x,inf);
limit( 2^x/x,x,inf);
(%o121) infinity
(%o122) 0
(%o123) inf
```

Es gibt hier zwei Ausgaben, `infinity` für etwas, das unendlich wird, aber dessen Vorzeichen nicht festlegbar ist, und `inf` für ∞, das „klassische“ Unendlich. Ausserdem gibt es noch `ind` für *indefinit*, was Werte sind, die zwar nicht wirklich festlegbar sind, aber immerhin beschränkt bleiben:

```
(%i124) limit( sin(x),x,inf);
limit( sin(1/x),x,0);
limit( x*sin(1/x),x,0);
limit( x/sin(1/x),x,0);
limit( 1/sin(1/x),x,0);
limit( 1/sin(x),x,0);
(%o124) ind
(%o125) ind
(%o126) 0
(%o127) 0
(%o128) und
(%o129) infinity
```

Noch ein Beispiel, diesmal e^x/x und $(-e)^x/x$ für $x \to \infty$:

```
(%i130) limit( %e^x/x,x,inf);
limit( (-%e)^x/x,x,inf);
(%o130) inf
(%o131) 0
```

Auch hier ist das zweite Ergebnis nicht wirklich nachvollziehbar. Und beim Logarithmus sieht man einen schönen Unterschied zwischen der ein- und zweiseitigen Grenzwerten:

```
(%i132) limit (log(x),x,0 );
limit( log(x),x,0,plus);
limit( log(x),x,0,minus);
(%o132) infinity
(%o133) -inf
(%o134) infinity
```

Bei manchen Folgen kennt `Maxima` sogar den symbolischen Grenzwert und kann damit umgehen:

```
(%i135) limit( (1+x)^(1/x),x,0);
limit( %e^{%i*x},x,%pi);
(%o135) %e
(%o136) %e^({%i*%pi})
```

Und zum Abschluss noch ein paar gemeinere Grenzwerte, die rekursive Funktion bringt dann sogar einen Absturz zustande:

```
(%i137) g(x) := cos(x);
limit( g(x+2*%pi)/g(x),x,inf);
limit( g(x+%pi)/g(x),x,inf);
limit( g(x+1)/g(x),x,inf);
f(x) := f(x-1) + f(x-2);
limit( f(x+1)/f(x),x,inf);
(%o137) g(x):=cos(x)
(%o138) 1
(%o139) -1
(%o140) und
(%o141) f(x):=f(x-1)+f(x-2)
Maxima encountered a Lisp error:
 Error in PROGN [or a callee]: Bind stack overflow.
Automatically continuing.
To enable the Lisp debugger set *debugger-hook* to nil.
```

3.4.2 Ableitungen

Maxima unterscheidet zwischen konkreten Ableitungen, bei denen wirklich versucht wird, Differentialrechnung zu betreiben und formalen Ableitungen, bei denen lediglich ein Differentialoperator in den Ausdruck integriert wird. Das wird durch `diff` bzw. `'diff` ausgedrückt:

```
(%i1) diff( sin(x),x);
diff( sin(x),y);
'diff( sin(x),x);
(%o1) cos(x)
(%o2) 0
(%o3) 'diff(sin(x),x,1)
```

Für Maxima sind **partielle Ableitungen**

```
-->  diff( sin(x*y^2),x);
diff( sin(x*y^2),y);
(%o146) y^2*cos(x*y^2)
(%o147) 2*x*y*cos(x*y^2)
```

ebensowenig ein Problem wie **höhere Ableitungen**:

```
-->  diff( sin(x*y^2),x,7);
diff( sin(x*y^2),y,7);
```

```
diff( sin(x*y^2),x,3,y,7);
(%o148) -y^14*cos(x*y^2)
(%o149) -1344*x^6*y^5*sin(x*y^2)+1680*x^4*y*sin(x*y^2)
   -128*x^7*y^7*cos(x*y^2)+3360*x^5*y^3*cos(x*y^2)
(%o150) -128*x^7*y^13*sin(x*y^2)+43680*x^5*y^9*sin(x*y^2)
   -383040*x^3*y^5*sin(x*y^2)+40320*x*y*sin(x*y^2)
   +4032*x^6*y^11*cos(x*y^2)-199920*x^4*y^7*cos(x*y^2)
   +262080*x^2*y^3*cos(x*y^2)
```

Das gilt natürlich auch symbolisch:

```
(%i4) 'diff( sin(x*y^2),x,3,y,7);
(%o4) 'diff(sin(x*y^2),x,3,y,7)
```

Ist die Funktion nicht konkret spezifiziert, muss natürlich zuerst symbolisch gerechnet werden, durch Einsetzen und Vereinfachen wird es dann oftmals besser:

```
(%i5) diff( exp(f(x)),x,2);
%,f=sin;
ev( %,diff,trigsimp);
diff( exp(sin(x)),x,2);
(%o5) %e^f(x)*('diff(f(x),x,2))+%e^f(x)*('diff(f(x),x,1))^2
(%o6) %e^sin(x)*('diff(sin(x),x,2))+%e^sin(x)*('diff(sin(x),x,1))^2
(%o7) %e^sin(x)*cos(x)^2-%e^sin(x)*sin(x)
(%o8) %e^sin(x)*cos(x)^2-%e^sin(x)*sin(x)
```

Mit `depends` kann man explizit sagen, von welchen Variablen eine Funktion abhängt und dies wird bei den Ableitungen dann auch berücksichtigt:

```
(%i9) depends( f,[x,y] );
diff(f,x);
diff(f,y);
diff(f,z);
(%o9) [f(x,y)]
(%o10) 'diff(f,x,1)
(%o11) 'diff(f,y,1)
(%o12) 0
```

Gradient und **Divergenz** gibt es ebenfalls, und zwar im Paket `vect`:

```
(%i13) load("vect")$
(%i14) grad (x^2 + y^2 + z^2);
express(%);
ev( %,diff);
(%o14) grad(z^2+y^2+x^2)
(%o15) ['diff((z^2+y^2+x^2),x,1),'diff((z^2+y^2+x^2),y,1),
        'diff((z^2+y^2+x^2),z,1)]
(%o16) [2*x,2*y,2*z]
```

grad war der Gradient, die Divergenz erhält man über div:

```
(%i17) div ( [x^2,y^2,z^2]);
express(%);
ev( %,diff);
(%o17) div([x^2,y^2,z^2])
(%o18) 'diff(z^2,z,1)+'diff(y^2,y,1)+'diff(x^2,x,1)
(%o19) 2*z+2*y+2*x
```

Wer die Definition der Divergenz nicht mehr kennt, hier ganz symbolisch:

```
(%i20) remove([f,g,h],dependency);
div( [f,g,h]);
express( % );
ev( %,diff );
dependencies;
(%o20) done
(%o21) div([f,g,h])
(%o22) 'diff(h,z,1)+'diff(g,y,1)+'diff(f,x,1)
(%o23) 0
(%o24) []
```

Mit remove löscht man vorher festgelegte Eigenschaften von Objekten, in diesem konkreten Fall die Abhängigkeiten, also die Variablen, die die Funktion beeinflussen. Der Befehl dependencies hingegen listet alle vorher explizit eingestellten Abhängigkeiten auf. Setzt man Abhängigkeiten, so werden diese in der Divergenz natürlich berücksichtigt:

```
(%i25) depends( [f,g,h],[x,z]);
div( [f,g,h]);
express( % );
ev( %,diff );
dependencies;
(%o25) [f(x,z),g(x,z),h(x,z)]
(%o26) div([f,g,h])
(%o27) 'diff(h,z,1)+'diff(g,y,1)+'diff(f,x,1)
(%o28) 'diff(h,z,1)+'diff(f,x,1)
(%o29) [f(x,z),g(x,z),h(x,z)]
```

Formale Operatoren erlauben eine sehr elegante Formulierung von Differentialgleichungen, die man dann über desolve lösen lassen kann:

```
(%i30) 'diff(f(x),x,2) = -f(x);
atvalue( f(x),x=0,1);
desolve( %th(2),f(x));
(%o30) 'diff(f(x),x,2)=-f(x)
(%o31) 1
(%o32) f(x)=sin(x)*(at('diff(f(x),x,1),x=0))+cos(x)
```

Hierbei haben wir mit `atvalue` festgelegt, daß `f` an der Stelle $x = 0$ den Wert 1 haben soll. Das kann man auch abfragen:

```
(%i33) printprops( f,atvalue );
f(0)=1
(%o33) done
```

3.4.3 Integrale

Was dem einen seine Ableitung ist dem anderen sein Integral. `Maxima` kann sowohl unbestimmte als auch bestimmte Integration, wobei bei der bestimmten Integration einfach die beiden **Integrationsgrenzen** zusätzlich angegeben werden müssen:

```
(%i22) integrate( sin(x),x);
integrate( sin(x),x,0,%pi);
(%o22) -cos(x)
(%o23) 2
```

Komplexere Funktionen und $\pm\infty$ als Integrationsgrenzen sind ebenfalls kein Problem:

```
(%i24) integrate (sin(x)^3, x);
integrate (x/ sqrt (b^2 - x^2), x);
integrate (cos(x)^2 * exp(x), x, 0, %pi);
integrate (x^2 * exp(-x^2), x, minf, inf);
(%o24) cos(x)^3/3-cos(x)
(%o25) -sqrt(b^2-x^2)
(%o26) (3*%e^%pi)/5-3/5
(%o27) sqrt(%pi)/2
```

Der (formale) Hauptsatz der Differential- und Integralrechnung, daß Integration und Differentiation inverse Operationen voneinander sind, wird allerdings nur eingeschränkt wiedergegeben:

```
(%i28) remove(f,dependency);
'diff(f,x);
'integrate(%,x);
ev( %,diff );
depends(f,x);
diff(f,x);
integrate(%,x);
ev( %,diff,integrate );
(%o28) done
(%o29) 'diff(f,x,1)
(%o30) integrate('diff(f,x,1),x)
(%o31) 0
(%o32) [f(x)]
(%o33) 'diff(f,x,1)
```

```
(%o34) integrate('diff(f,x,1),x)
(%o35) integrate('diff(f,x,1),x)
```

Obwohl im zweiten Durchgang die Funktion f als explizit von x abhängig definiert wurde, erhält man trtozdem nicht mehr als eine Integration der formalen Differentiation und eben nicht die Funktion f. Und ja, natürlich können sich f und $\int f'$ immer um eine Konstante unterscheiden, das ist ja bekanntlich unvermeidbar.

Manche Integrale, insbesondere die von komplexeren[79] Exponentialfunktionen liefern als Ergebnis eine **spezielle Funktion**, beispielsweise die **Gammafunktion**. Für diese Funktionen gibt es eine Vielzahl von Rechenregeln, die auch `Maxima` bekannt sind. Ein Beispiel mit der sogenannten unvollständigen Gammafunktion:

```
(%i36) f : %e^(%i * x) + sin(x)/x^2;
diff(f,x);
integrate( %,x );
(%o36) sin(x)/x^2+%e^(%i*x)
(%o37) -(2*sin(x))/x^3+cos(x)/x^2+%i*%e^(%i*x)
(%o38) %e^(%i*x)-(%i*gamma_incomplete(-1,%i*x)
  -%i*gamma_incomplete(-1,-%i*x))/2-%i*gamma_incomplete(-2,%i*x)
  +%i*gamma_incomplete(-2,-%i*x)
```

Hier steht %i für die komplexe Einheit i. Manche Berechnungen funktionieren nur, wenn beispielsweise ein Parameter ein bestimmtes Vorzeichen hat. Dies kann man in Computeralgebrasystemen explizit sagen, in `Maxima` über die Funktion `assume`. Hier ein Integral, nämlich

$$\int_0^\infty \frac{x^a}{(x+1)^{\frac{5}{2}}}\,dx, \qquad a > 1.$$

Für $a < 1$ ist das Integral „einfach“. Hier spielt es eine Rolle, ob a eine ganze Zahl ist oder nicht und genau das wird abgefragt:

```
(%i39) assume (a > 1)$
integrate (x^a/(x+1)^(5/2), x, 0, inf);
"Is "a" an "integer"?"yes;
"Is "2*a-1" positive, negative or zero?"positive;
(%o40) integrate(x^a/(x+1)^(5/2),x,0,inf)
```

Etwas enttäuschend. Und eine numerische Auswertung bringt uns auch nicht weiter:

```
(%i41) ev(%,numer);
(%o41) integrate(x^a/(x+1)^2.5,x,0,inf)
```

[79] Und diese durchaus im doppelten Wortsinn.

Übung 3.1 Experimentieren Sie im Worksheet mit weiteren Möglichkeiten, a zu wählen. ◊

Nochmal zurück zu den gewöhnlichen Differentialgleichungen. Hier ein einfaches Beispiel einer allgemeinen Lösung der **Wellengleichung**

$$f''(x) = -f(x)$$

mit generischer Lösung $f(x) = a \sin x + b \cos x$. In `Maxima` sieht das entsprechend aus[80]:

```
(%i42) remvalue(all)$ remove(all,atvalue)$
desolve( 'diff(f(x),x,2) = -f(x), f(x) );
(%o44) f(x)=sin(x)*(at('diff(f(x),x,1),x=0))+f(0)*cos(x)
```

Für das System

$$\begin{aligned} f''(x) &= \sin x + g'(x) \\ f'(x) - f(x) + x^2 &= 2g''(x) \end{aligned}$$

gibt es auch eine Lösung, aber die schaut man sich besser in `Maxima` selbst an.

```
(%i45) eq1 : 'diff(f(x),x,2) = sin(x) + 'diff(g(x),x);
eq2: 'diff(f(x),x) + x^2 - f(x) = 2*'diff(g(x),x,2);
desolve( [eq1,eq2 ],[f(x),g(x)] );
(%o45) 'diff(f(x),x,2)='diff(g(x),x,1)+sin(x)
(%o46) 'diff(f(x),x,1)-f(x)+x^2=2*('diff(g(x),x,2))
```

Dann noch ein Anfangswertproblem, bei dem die Anfangs- bzw. Randwerte über `atvalue` vorgegeben werden:

```
(%i48) remove(all,atvalue);
atvalue( f(x),x=0,0 );
atvalue( 'diff(f(x),x),x=0,1 );
desolve( 'diff(f(x),x,2) = -f(x), f(x) );
remove(all,atvalue);
atvalue( f(x),x=%pi/2,0 );
printprops( f,atvalue );
desolve( 'diff(f(x),x,2) = -f(x), f(x) );
(%o48) done
(%o49) 0
(%o50) 1
(%o51) f(x)=sin(x)
(%o52) done
(%o53) 0
f(%pi/2)=0
(%o54) done
(%o55) f(x)=sin(x)*(at('diff(f(x),x,1),x=0))+f(0)*cos(x)
```

[80] Mit vorheriger Löschung der Variablen.

und ein parametrisiertes System, dessen Lösung hier aber ebenfalls unterschlagen wird:

```
(%i56) remove(all,atvalue);
atvalue( 'diff(g(x),x),x=0,a );
atvalue(f(x),x=0,1);
atvalue( 'diff(f(x),x),x=0,b );
desolve( [eq1,eq2 ],[f(x),g(x)] );
(%o56) done
(%o57) a
(%o58) 1
(%o59) b
```

3.5 Das Bierkastenproblem

Jetzt befassen wir uns einmal mit der Frage, wie man `Maxima` einsetzen kann, um ein Problem aus der „Praxis“ zu lösen. Es handelt sich hierbei auch um ein echtes wenn auch Problem aus der Robotersteuerung und zwar der Steuerung von Robotern, die mit der Handhabung von Getränkekisten befasst sind, genauer mit Stapeln von Getränkekisten[81], bewegt und zwar über ein Hindernis hinweg. Um dieses Hindernis zu umfahren, muss die Bahn des Roboters durch einen bestimmten Punkt hindurchgeführt werden. Aus technischen Gründen[82] ist die Bahn eine Kombination aus Geradenstücken und Kreisbögen mit *einem* festen Radius `r`. Damit die Bierkästen nicht umfallen muss die Bahn ausserdem im

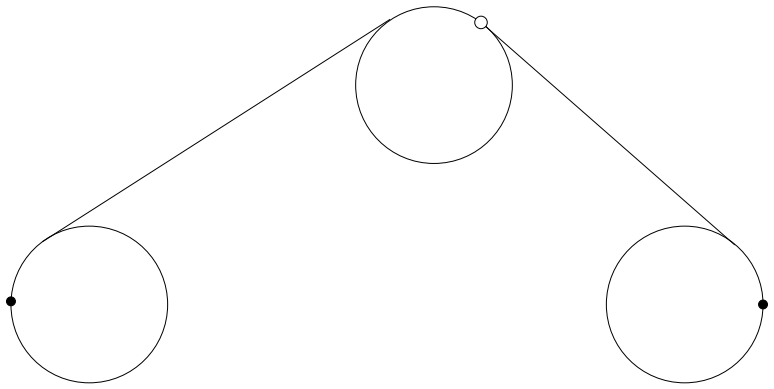


Abbildung 3.6: Eine zulässige Bahn durch den vorgegebenen Punkt. Die Kreise mit dem vorgegebeben Radius sind eingezeichnet.

Anfangs- und Endpunkt eine senkrechte Tangente haben, siehe Abb. 3.6. Die Aufgabe ist nun:

Wie bestimmt man den kürzesten derartigen Weg.

[81] Daher der interne „Codename“ des Projekts: *Das Bierkastenproblem*
[82] Der Roboter ist halt einfach so gebaut.

Dabei überlegt man sich zunächst, daß sich das Problem ziemlich vereinfachen lässt. Da der Pfad senkrecht beginnt und endet, besteht er immer aus einem Halbkreis und zwei Geraden, siehe Abb. 3.7. Die beiden Mittelpunkte sind

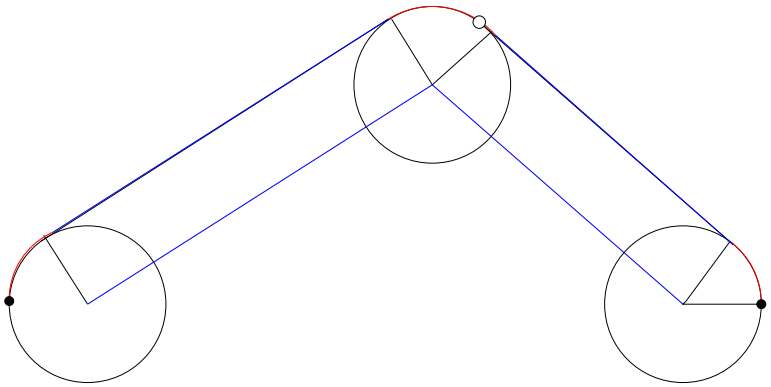


Abbildung 3.7: Die Kreis- und Geradenanteile. Genau genommen interessiert einen also nur ein Pfad vom ersten Kreismittelpunkt über einem Punkt, dessen Abstand zum „Ausweichpunkt“ mit dem Radius übereinstimmt zum Mittelpunkt des zweiten Kreises.

durch Anfangs- bzw. Endpunkt und Radius eindeutig bestimmt, der einzige „Freiheitsgrad“ besteht also in der Wahl des Punktes, an dem sich die beiden Linienstücke treffen und dieser muss auf einem Kreis mit dem Radius r um den Ausweichpunkt c liegen, siehe Abb. 3.8, und zwar so, daß der **Gesamtweg**

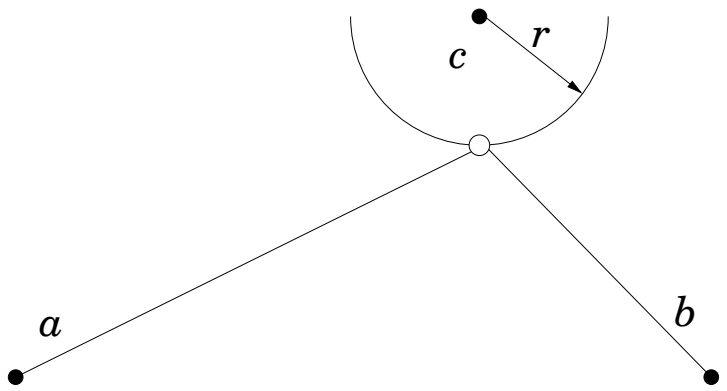


Abbildung 3.8: Das wirkliche Optimierungsproblem.

minimal wird. Das Optimierungsproblem besteht also darin, einen Punkt x auf dem Kreis so zu wählen, daß der Gesamtweg minimiert wird, also

$$\min_x w(x) := \min_x \|x - a\|_2 + \|x - b\|_2 \,, \qquad \|x - c\|_2 = r. \tag{3.6}$$

Das Gemeine an der Sache ist, daß man hier wirklich w, also die Summe der beiden Streckenlängen zu minimieren hat und nicht den ansonsten sehr populären

Wert $\|x - a\|_2^2 + \|x - b\|_2^2$, von dem wir ja aus den Lest–Squares–Problemen wissen, daß er recht einfach zu lösen ist. Dennoch kann man sich leicht überlegen, daß das Minimum *eindeutig* sein muss[83]

Betrachtet man die Menge der Punkte x, für wie $w(x)$ konstant ist, also die Niveaulinien von w, dann ist dies gerade eine Ellipse mit den beiden Brennpunkten a und b. Diese Ellipse hat die beiden Halbachsen α und β, die die Bedingung $\alpha^2 + \beta^2 = \|a - b\|^2$ erfüllen. Jeder Punkt auf der Ellipse hat dann in der Summe den Abstand 2α von den beiden Brennpunkten und die Suche nach dem kürzesten Weg ist äquivalent zum Auffinden der Berührellipse mit der großen Halbachse α, siehe Abb. 3.9. Dazu empfiehlt es sich, ein paar Vereinfa-

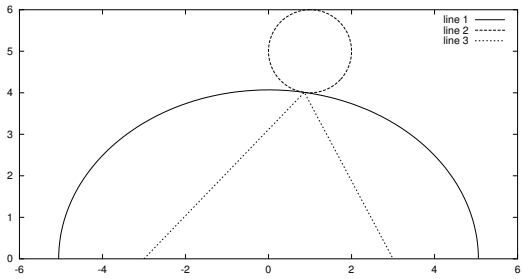


Abbildung 3.9: Berührellipse an die Kugel und der zugehörige kürzeste Weg.

chungen zu machen. Wir setzen $e = \|a - b\|_2$ und wählen $a = [-e, 0]$, $b = [0, e]$ und legen damit den Ursprung in die Mitte zwischen den beiden Punkten. Die Gleichung für die Ellipse ist dann

$$\frac{x^2}{\alpha^2} + \frac{y^2}{\beta^2} = 1, \qquad \alpha^2 - \beta^2 = e^2 = \|a - b\|_2^2, \tag{3.7}$$

und die der Kugel ja bekanntlich

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \tag{3.8}$$

Das quadratische[84] Gleichungssystem aus (3.7) und (3.8) hat im allgemeinen vier Lösungen, beispielsweise, wenn $c = 0$ und $\beta < r < \alpha$ ist. Zwei der Lösungen sind anschaulich sofort klar, nämlich der na"chste Punkt wie in Abb 3.9 und die Berührellipse an den fernsten Punkt des Kreises. Die beiden anderen Lösungen entsprechen Ellipsen mit *negativer* Halbachse, deren „Mittelpunkt" im Unendlichen liegt, was zu Berührhyperbeln an den Kreis führt. Das wird sich auch in unserem Fall nicht ändern, und wir müssten daher α so bestimmen, daß es *zwei komplexe* und *eine doppelte reelle* Lösung des Gleichungssystems gibt, dann

[83] Wie schon der Highlander wusste: „Es kann nur einen geben".

[84] Im dem Sinne, daß alle Gleichungen Polynome vom Grad 2, also quadratisch sind.

berührt die Ellipse den Kreis. Dieses Problem lässt sich tatsächlich mit Methoden der Computeralgebra angehen, ist da aber dann schon ein echter Härtetest, siehe (Sauer, 2001).

Um das Problem ausgehend von dieser geometrischen Überlegung numerisch anzugehen, verwenden wir die parametrischen Darstellungen von Kugel und Ellipse als

$$K(\phi) = c - r \begin{bmatrix} \sin\phi \\ \cos\phi \end{bmatrix}, \qquad E(\psi) = \begin{bmatrix} \alpha \sin\psi \\ \beta \cos\psi \end{bmatrix}, \qquad \phi, \psi \in [-\pi, \pi].$$

Wir müssen also α, ϕ und ψ so bestimmen, daß wir einen Berührpunkt erhalten, daß also die Tangenten

$$K'(\phi) = -r \begin{bmatrix} \cos\phi \\ -\sin\phi \end{bmatrix}, \qquad E'(\phi) = \begin{bmatrix} \alpha \cos\psi \\ -\beta \sin\psi \end{bmatrix}$$

kollinear sind: $K'(\phi) = \lambda E'(\psi)$, $\lambda \neq 0$. Da die Tangentialvektoren nie $= 0$ sind, können wir ohne weiteres $\lambda \neq 0$ fordern. Setzen wir $s_\phi := \sin\phi$, $c_\phi := \cos\phi$ und dasselbe für ψ, dann erhalten wir das Gleichungssystem bestehend aus Schnitt-

$$\begin{aligned} 0 &= c_1 - r\, s_\phi - \alpha\, s_\psi, \\ 0 &= c_2 - r\, c_\phi - \beta\, c_\psi, \end{aligned}$$

und Berührbedingungen

$$\begin{aligned} 0 &= \lambda\, \alpha\, c_\psi - r\, c_\phi, \\ 0 &= \lambda\, \beta\, s_\psi - r\, s_\phi, \end{aligned}$$

zusammen mit den Nebebedingungen

$$\begin{aligned} 0 &= s_\phi^2 + c_\phi^2 - 1, \\ 0 &= s_\psi^2 + c_\psi^2 - 1, \\ 0 &= \alpha^2 - \beta^2 - e^2. \end{aligned}$$

an Sinus, Cosinus und den Abstand. Dieses Gleichungssystem stecken wir nun im `Maxima`, indem wir die Gleichungen von oben mehr oder weniger wortwörtlich abtippen:

```
(%i1) eq1 : x1 - r*s1 - a*s2;
eq2 : x2 - r*c1 - b*c2;
(%o1) x1-a*s2-r*s1
(%o2) x2-c1*r-b*c2
```

Die Berührbedingungen sind

```
(%i3) eq3 : l*a*c2 - r*c1;
eq4 : l*b*s2 - r*s1;
(%o3) a*c2*l-c1*r
(%o4) b*l*s2-r*s1
```

und die Nebenbedinungen aus Sinus und Cosinus kennt man ja:

```
(%i5) eq5 : s1^2 + c1^2 - 1;
eq6 : s2^2 + c2^2 - 1;
(%o5) s1^2+c1^2-1
(%o6) s2^2+c2^2-1
```

Zum Abschluss noch die Abstandsbedingung und dann spezifizieren wir ein paar der Parameter:

```
(%i7) eq7 : a^2 - b^2 - e^2;
e : 5; x1 : 0; x2: 7; r : 3;
(%o7) -e^2-b^2+a^2
(%o8) 5
(%o9) 0
(%o10) 7
(%o11) 3
```

Wir lösen das mit dem **Newton–Verfahren**, das sich im Paket `mnewton` befindet, das dazugeladen werden muss:

```
(%i12) load( mnewton );
(%o12) "/usr/share/maxima/5.32.1/share/mnewton/mnewton.mac"
```

Dem muss man dann nur noch sagen, daß bitteschön *eine* Lösung berechnet werden soll[85]:

```
(%i13) mnewton( [ eq1,eq2,eq3,eq4,eq5,eq6,eq7 ],
       [l,a,b,s1,c1,s2,c2], [1,1,1,1,1,1,1]);
(%o13) [[l=0.46852128566581,a=6.403124237432849,b=4.000000000000001,
        s1=3.685558149192567*10^-27,c1=1.0,
        s2=-1.2949940620108 13*10^-26,c2=1.0]]
```

Diese Lösung hängt ebenso wie die Frage, ob das Verfahren überhaupt konvergiert, davon ab, wir gut der **Startwert**, das dritte Argument der Funktion `mnewton`, gewählt ist. Je nach Startwert[86] liefert das Verfahren dann die gute oder die „böse“ Lösung[87] oder eben gar nicht[88]

Ein Versuch, dieses eigentlich unschuldige Problem mit symbolischen Methoden, also der Funktion `algsys`, zu lösen, rechnet sich einfach zu Tode ...

[85] Das Newton–Verfahren liefert immer nur eine Lösung und auch diese nicht immer, Details in (Gautschi, 1997; Sauer, 2013).

[86] Und es empfiehlt sich, hier einfach mal ein bischen zu experimentieren.

[87] Das ist ja in Ordnung, denn sowoh der nächste als auch der weiteste Punkt sind Lösungen des Gleichungssystems.

[88] Das ist meistens nicht so willkommen.

3.6 „Richtige“ Computeralgebra: Gröbnerbasen

Zum Schluss noch ein paar einfache Bemerkungen zur „richtigen“ Computeralgebra[89]; dazu gibt es eigene Vorlesungen (Sauer, 2001) und einiges an guten Büchern (Cox *et al.*, 1996; Gathen & Gerhard, 1999). Damit wir aber wissen, wovon wir reden, ein bisschen Terminologie.

Für einen **Körper** $\mathbb{K}$ bilden die Polynome $\Pi = \mathbb{K}[x_1, \ldots, x_s]$, also die Menge aller *endlichen* Ausdrücke der Form

$$f(x) = \sum_{\alpha \in \mathbb{N}_0^s} f_\alpha \, x^\alpha, \qquad f_\alpha \in \mathbb{K}, \tag{3.9}$$

eine **Algebra**: Man kann Polynome addieren und mit Körperelementen multiplizieren[90], man kann sie aber auch miteinander multiplizieren und erhält wieder Polynome. Zur Notation: In (3.9) verwenden wir Multiindizes, wobei ein **Multiindex** $\alpha = (\alpha_1, \ldots, \alpha_s) \in \mathbb{N}_0^s$, eine Verallgemeinerung des Index bzw. Exponenten bei univariaten Polynomen ist. Die **Länge** eines Multiindex ist $|\alpha| = \alpha_1 + \cdots + \alpha_s$ und ein **Monom** ist ein Ausdruck der Form

$$x^\alpha = x_1^{\alpha_1} \cdots x_s^{\alpha_s}.$$

Anders gesagt:

> Ein Polynom ist eine *endliche* Linearkombination von Monomen.

Definition 3.7 *Ein **Ideal** $\mathscr{I} \subseteq \Pi$ ist eine Teilmenge von Π, die abgeschlossen ist unter*

1. *Addition von Idealelementen:* $f, g \in \mathscr{I} \Rightarrow f + g \in \mathscr{I}$,
2. *Multiplikation mit* beliebigen *Polynomen:* $f \in \mathscr{I}, p \in \Pi \Rightarrow f \cdot f \in \mathscr{I}$.

Zu einer endlichen Menge $F \subset \Pi$ von Polynomen kann man nun den **Abschluss** unter diesen beiden Operationen bilden, indem man die Menge

$$\langle F \rangle := \left\{ \sum_{f \in F} p_f \, f : p_f \in \Pi \right\} \tag{3.10}$$

bildet. Diese Menge ist nun abgeschlossen unter Addition und Multiplikation mit beliebigen Polynomen, also ein Ideal, und sie ist das *kleinste* Ideal, das H enthält[91]

Definition 3.8 *Die Menge $\langle F \rangle$ heißt von F **erzeugtes Ideal** und ist $\mathscr{I} = \langle F \rangle$, dann nennt man F eine **Basis** von $\mathscr{I}$.*

[89] Zumindest gibt es Leute, die unter **Computeralgebra** im wesentlichen die symbolische Manipulation von Polynomen in mehreren Variablen verstehen, insbesondere in Hinblick auf das Lösen polynomialer Gleichungssysteme.

[90] Das liefert einen $\mathbb{K}$–**Vektorraum**.

[91] Wegen ebendieser Abgeschlossenheit.

Dieses Konzept ermöglicht es prinzipiell, mit Idealen auf dem Computer zu rechnen, indem man sie durch eine endliche Basis repräsentiert und nur mit diesen Basen arbeitet. Daß dies zulässig ist, garantiert der Hilbertsche **Basissatz**, siehe (Cox *et al.*, 1996), der besagt, daß jedes Polynomideal eine endliche Basis besitzt.

Eines der einfachsten Probleme, die man im Kontext der algorithmischen Idealtheorie lösen möchte, ist das **Ideal Membership Problem**: Gegeben eine endlich Menge $F \subset \Pi$ und $g \in \Pi$

1. entscheide, ob $g \in \langle F \rangle$.
2. bestimme Polynome g_f, $f \in F$, so daß

$$g = \sum_{f \in F} g_f \, f. \tag{3.11}$$

Dieses Problem kann man nun wieder auf **Division mit Rest** zurückführen, indem man eine Darstellung

$$g = \sum_{f \in F} g_f \, f + r \tag{3.12}$$

bestimmt oder zu bestimmen sucht, bei der genau dann $r = 0$ ist, wenn $g \in \langle F \rangle$ ist. Man kann nun wirklich den euklidischen Algorithmus recht naiv verallgemeinern, siehe (Cox *et al.*, 1996), stellt dann aber schnell fest, daß das für allgemeine Basen nicht funktioniert. Hier kommt nun die **Gröbnerbasis** ins Spiel, bei der alles wieder gut ist. Wir fassen die wichtigsten Eigenschaften zusammen.

Satz 3.9 (Gröbnerbasis)

1. *Jedes Ideal hat eine endliche Gröbnerbasis.*
2. *Eine Gröbnerbasis kann aus einer endlichen Basis des Ideals effizient*[92] *berechnet werden*[93].
3. *Ist F eine Gröbnerbasis, so liefert ein entsprechender euklidischer Algorithmus eine Darstellung der Form (3.12) mit $r = 0$ genau dann wenn $g \in \langle F \rangle$.*

Fazit: Mit Gröbnerbasen ist die Welt wieder in Ordnung. So gesehen ist es kein Wunder, wenn sie auch in `Maxima` integriert sind und zwar im Paket `grobner`[94]. Das laden wir in gewohnter Weise

```
(%i1) load( grobner );
Loading maxima-grobner $Revision: 1.6 $ $Date: 2009-06-02 07:49:49 $
(%o1) "/usr/share/maxima/5.32.1/share/contrib/Grobner/grobner.lisp"
```

Als erstes sehen wir uns an, daß naive Division mit Rest nicht funktionert. Dazu gibt es eine Funktion `poly_pseudo_divide`, die ein Polynom durch eine Liste von Polynomen[95] teilt[96] und dann Darstellung, Rest und weitere Statistiken

[92] Man kann über „effizient“ streiten, denn die Bestimmung einer Gröbnerbasis kann sehr lange dauern.

[93] Das ist der sogenannte **Buchberger–Algorithmus** aus (Buchberger, 1965).

[94] Mit Umlauten hat die Welt so ihre Probleme ...

[95] Achtung: Reihenfolge!

[96] Man muss noch angeben, was in den Ausdrücken die Variablen sind.

ausgibt:

```
(%i2) poly_pseudo_divide( x^2*y, [ x*y-2,x^2+2*y-1 ], [x,y] );
poly_pseudo_divide( x^2*y, [ x^2+2*y-1,x*y-2 ], [x,y] );
(%o2) [[x,0],2*x,1,1]
(%o3) [[y,0],y-2*y^2,1,1]
```

Und in der Tat: Man erhält zwei Darstellungen der Form (3.12), aber sowohl Darstellung als auch Rest hängen offenbar von der Reihenfolge der Polynome in der Liste ab. Und manchmal auch nicht:

```
(%i4) poly_pseudo_divide( x^3+y^3+1, [ x*y-2,x^2+2*y-1 ], [x,y] );
poly_pseudo_divide( x^3+y^3+1, [ x^2+2*y-1,x*y-2 ], [x,y] );
(%o4) [[-2,x],y^3+x-3,1,2]
(%o5) [[x,-2],y^3+x-3,1,2]
```

Interessiert man sich nur für die Reste der Division, so kann man eine Funktion `poly_normal_form` verwenden, die nur diese Reste oder eben die **Normalform** modulo Ideal zurückliefert. Für die beiden Beispiele erhalten wir je nach Anordnung unserer Divisoren die Ergebnisse

```
(%i6) poly_normal_form( x^2*y, [ x*y-2,x^2+2*y-1 ], [x,y] );
poly_normal_form( x^3+y^3+1, [ x*y-2,x^2+2*y-1 ], [x,y] );
(%o6) 2*x
(%o7) y^3+x-3
```

und

```
(%i8) poly_normal_form( x^2*y, [ x^2+2*y-1,x*y-2 ], [x,y] );
poly_normal_form( x^3+y^3+1, [ x^2+2*y-1,x*y-2 ], [x,y] );
(%o8) y-2*y^2
(%o9) y^3+x-3
```

Und jetzt der „Zaubertrick“: Verwendet man eine **Gröbnerbasis** des Ideals, die man mit der Funktion bestimmen kann, dann erhält man

```
(%i10) poly_normal_form( x^2*y,
         poly_grobner([ x^2+2*y-1,x*y-2 ],[x,y]),[x,y] );
poly_normal_form( x^2*y,
  poly_grobner([ x*y-2,x^2+2*y-1 ],[x,y]),[x,y] );
(%o10) y-2*y^2
(%o11) 2*y^2-y
```

und da Algebraiker hier nicht abergläubisch sind[97], ist das nun derselbe Rest. Die Gröbnerbasen lassen sich ebenfalls berechnen und sie hängen tatsächlich unwesentlich von der Reihenfolge ab:

```
(%i12) poly_grobner( [ x^2+2*y-1,x*y-2 ], [x,y] );
poly_grobner( [ x*y-2,x^2+2*y-1 ], [x,y] );
(%o12) [2*y+x^2-1,x*y-2,2*y^2-y+2*x,-2*y^3+y^2-4]
(%o13) [x*y-2,2*y+x^2-1,-2*y^2+y-2*x,2*y^3-y^2+4]
```

[97] Mit anderen Worten: Vorzeichen sind ihnen egal. Zumindest an dieser Stelle

Genau dieses Artefakt des Buchberger–Algorithmus führt dann auch zu den unterschiedlichen Vorzeichen bei der Normalform.

Ein paar Bemerkungen noch zum Buchberger–Algorithmus:

1. In seiner „Urform“ ist der Algorithmus, der auf der **Reduktion** von **Syzygien**[98] basiert und in (Cox *et al.*, 1996) wirklich sehr gut und verständlich beschrieben ist, sehr langsam und ineffizient und es gibt eine Vielzahl von Verbesserungen, die in einer konkreten Implementierung enthalten sein können oder eben nicht, was in Sachen Laufzeit ganz gewaltige Unterschiede machen kann.

2. Generell ist die **Komplexität** des Buchberger–Algorithmus aber als sehr groß bekannt, es gibt Worst–Case–Abschätzungen, die *doppelt exponentiell* sind, also Laufzeiten der Ordnung 2^{2^n} haben, wobei n die Anzahl der Elemente in der Idealbasis ist. Und es gibt Beispiele, bei denen diese Komplexität auch wirklich auftritt.

3. Der Buchberger–Algorithmus basiert auf dem Konzept der **Termordnung** und sowohl die Laufzeit als auch das Aussehen der Gröbnerbasis hängen davon ab.

Definition 3.10 *Unter einer **Termordnung** versteht man eine **totale Ordnung** auf der Menge $\mathbb{N}_0^s$ der Multiindizes bzw. die dadurch induzierte totale Ordnung der Monome x^α.*

Beispiel 3.11 (Termordnungen) *Die bekanntesten Beispiele für Termordnungen, geschrieben als $\alpha \prec \beta$, sind*

1. *die **lexikographische Termordnung „lex“** mit*

$$\alpha \prec_\ell \beta \qquad \Leftrightarrow \qquad \begin{cases} \alpha_j = \beta_j, & j = 1, \ldots, k-1, \\ \alpha_k < \beta_k, \end{cases} \tag{3.13}$$

2. *die **revers lexikographische Termordnung „revlex“** oder **„invlex“** mit*

$$\alpha \prec_r \beta \qquad \Leftrightarrow \qquad \begin{cases} \alpha_j = \beta_j, & j = k+1, \ldots, s, \\ \alpha_k < \beta_k, \end{cases} \tag{3.14}$$

3. *die **graduiert lexikographische Termordnung „gradlex“** oder **„grlex“** mit*

$$\alpha \prec \beta \qquad \Leftrightarrow \qquad |\alpha| < |\beta| \quad \textit{oder} \quad |\alpha| = |\beta|, \textit{ und } \alpha \prec_\ell \beta, \tag{3.15}$$

4. *die **graduiert revers lexikographische Termordnung „grevlex“** mit*

$$\alpha \prec \beta \qquad \Leftrightarrow \qquad |\alpha| < |\beta| \quad \textit{oder} \quad |\alpha| = |\beta|, \textit{ und } \alpha \prec_r \beta. \tag{3.16}$$

[98] Dieses Wort musste einmal fallen.

Die Termordnung kann man in `Maxima` zumindest aus den obigen vier Möglichkeiten auswählen, und war mit der Variablen `poly_monomial_order`, die entsprechend zu setzen ist. Dies führt dann in unserem Beispiel auch zu unterschiedliche Gröbnerbasen:

```
(%i14) poly_monomial_order : lex$
       poly_grobner( [ x^2+2*y-1,x*y-2 ], [x,y] );
poly_monomial_order : grlex$ p
   poly_grobner( [ x^2+2*y-1,x*y-2 ], [x,y] );
poly_monomial_order : grevlex$
   poly_grobner( [ x^2+2*y-1,x*y-2 ], [x,y] );
poly_monomial_order : invlex$
   poly_grobner( [ x^2+2*y-1,x*y-2 ], [x,y] );
(%o15) [2*y+x^2-1,x*y-2,2*y^2-y+2*x,-2*y^3+y^2-4]
(%o17) [2*y+x^2-1,x*y-2,2*y^2-y+2*x]
(%o19) [2*y+x^2-1,x*y-2,2*y^2-y+2*x]
(%o21) [2*y+x^2-1,x*y-2,x^3-x+4]
```

Interessant ist der letzte Fall mit **invlex**, denn da enthält die Basis das Polynom x^3-x+4, das ein Polynom ausschliesslich in der einen Variablen x ist. Tatsächlich enthält die lexikographische Gröbnerbasis immer auch eine Gröbnerbasis für Polynome in weniger Variablen in dem Ideal, das sogenannte **Eliminationsideal**, mit dessen Hilfe man Abhängigkeiten zwischen den Variablen eliminieren kann. Und wie sieht das im **lex**–Fall aus? Ganz genauso, hier finden wir $-2*y^3+y^2-4$, so daß die Invertierung der Reihenfolge der Variablen nur festlegt, welche Variable „bleiben" darf und welche eliminiert wird. Bei mehr als zwei Variablen könne man diese nach Bedarf eventuell auch noch permutieren. Auch hier gilt wieder: Details in einer Spezialvorlesung.

Jetzt noch ein besonderes Beispiel, der Schnitt der beiden Ellipsen

$$\begin{aligned} f(x) &= \frac{1}{3}x^2+\frac{2}{3}y^2-1 \\ g(x) &= \frac{2}{3}x^2+\frac{1}{3}y^2-1 \end{aligned} \tag{3.17}$$

die sich genau in den Punkten $(\pm1,\pm1)$ schneiden[99], wie man leicht nachrechnen kann. Definieren wir die beiden:

```
(%i22) f : 1/3*x^2 + 2/3*y^2 - 1;
g : 2/3*x^2 + 1/3*y^2 - 1;
(%o22) (2*y^2)/3+x^2/3-1
(%o23) y^2/3+(2*x^2)/3-1
```

Um das Problem zu lösen, können wir etweder die Funktion `solve` verwenden,

```
(%i24) solve( [ f = 0,g = 0],[x,y] );
(%o24) [[x=-1,y=-1],[x=-1,y=1],[x=1,y=-1],[x=1,y=1]]
```

[99] Das ist die Schranke aus dem **Satz von Bézout** aus der algebraischen Geometrie, weshalb man hier von **complete intersection** spricht.

oder, das es sich ja um **algebrische Gleichungen** handelt, auch die Funktion `algsys`

```
(%i25) algsys([f, g], [x,y]);
(%o25) [[x=-1,y=-1],[x=-1,y=1],[x=1,y=-1],[x=1,y=1]]
```

In beiden Fällen erhält man *sofort* ein richtiges Ergebnis, was will man also mehr?

Sehen wir uns nun die Gröbnerbasen zu dem Problem an, so stellen wir fest, daß die **Termordnung** hier gar keine Rolle spielt:

```
(%i26) poly_monomial_order : grlex$ poly_grobner( [f,g], [x,y] );
poly_monomial_order : lex$ poly_grobner( [f,g], [x,y] );
(%o27) [2*y^2+x^2-3,y^2+2*x^2-3,y^2-1]
(%o29) [2*y^2+x^2-3,y^2+2*x^2-3,y^2-1]
```

Das gilt *beweisbar* für jede beliebige Termordnung, solange die Schnittpunkte nur auf einem **Gitter** liegen, d.h. von der Form

$$x_\alpha = (x_{1,\alpha_1}, \ldots, x_{s,\alpha_s}), \qquad X_j := \{x_{j,k} : k = 1, \ldots, n_j\} \subset \mathbb{C},$$

die man als **Tensorprodukt** $X_1 \otimes \cdots \otimes X_s$ bezeichnet.

Jetzt stören wir unser Problem etwas, indem wir die eine Ellipse um einen Winkel a drehen:

```
(%i30) fa : f,x=cos(a)*x + sin(a)*y,y=-sin(a)*x + cos(a)*y;
(%o30) (sin(a)*y+cos(a)*x)^2/3+(2*(cos(a)*y-sin(a)*x)^2)/3-1
```

Die resultierende Lösung ist dann nicht sonderlich hilfreich:

```
(%i33) solve( [fa,g],[x,y] );
(%o33) []
```

denn natürlich gibt es, zumindest für kleine Winkel, auf jeden Fall noch Lösungen, die aus Stetigkeitsgründen auch ungefähr $(\pm 1, \pm 1)$ sein sollten. Allerdings ist es auch nicht fair[100], zu erwarten, daß eine Universallösung für alle Winkel geliefert werden kann. Na gut, setzen wir den Winkel explizit fest, dann erhalen wir

```
(%i35) fa2 : fa,a=1/10000;
solve( [fa2,g],[x,y] );
fa3 : fa,a=1/1000,numer;
solve( [fa3,g],[x,y] );
(%o35) (sin(1/10000)*y+cos(1/10000)*x)^2/3
  +(2*(cos(1/10000)*y-sin(1/10000)*x)^2)/3-1
(%o36) []

(%o37)
```

[100] Wenn man wie bei `Maple` und `Mathematica` allerdings viel Geld für die Wundersoftware bezahlt hat, dann ist die Erwartung eine andere und deswegen wird auch viel Arbeit in Routinen investiert, die dann halt wenigstens irgendwas liefern.

```
 %0.66666666666666*(0.99999950000004*y-9.999998333333416*10^-4*x)^2
+0.33333333333333*(9.999998333333416*10^-4*y+0.99999950000004*x)^2-1
rat: replaced 0.33333333333333 by 1/3 = 0.33333333333333
rat: replaced 0.99999950000004 by 11999995/12000001 =
     0.99999950000004
rat: replaced 9.999998333333416E-4 by 8555999/8556000426 =
     9.999998333333416E-4
rat: replaced 0.66666666666666 by 2/3 = 0.66666666666666
rat: replaced -9.999998333333416E-4 by -8555999/8556000426 =
     -9.999998333333416E-4
rat: replaced 0.99999950000004 by 11999995/12000001 =
     0.99999950000004
(%o38) []
```

Im ersten Fall geht immer noch nichts, denn algebraisch–symbolische Methoden benötigen *rationale* Koeffizienten, bei einer *numerischen* Lösungsanfrage mit dem Zauberwort `numer` gibt's dann zumindest eine Lösung.

Der explizit algebraische Löser versucht eine exakte Lösung des Problems zu finden und landet bei

```
(%i39) algsys( [fa3,g],[x,y] );
(%o39) []
```

Das liegt daran, daß er mit den Sinus- und Cosinuswerten nicht symbolisch rechnen kann, allen Additionstheoremen zum Trotz. Wendet man allerdings `ratsimp` auf das Polynom an, dann werden die numerischen Werte in rationale Zahlen konvertiert und die Verfahren liefern ein Ergebnis:

```
(%i46) fa4 : ratsimp( fa3 );
algsys( [fa4,g], [x,y] );
solve( [fa4 = 0, g=0 ], [x,y]);
```

Das Resultat geben wir hier allerdings nicht wieder, es ist ein klein wenig länglich, gibt einem aber das gute Gefühl, der Computereinsatz hätte sich gelohnt.

3 Literatur

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D. (1995). *LAPACK User's Guide*. SIAM, second edition.

Askey, R. (1975). *Orthogonal Polynomials and Special Functions*, volume 21 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM.

Brieskorn, E. (1983). *Lineare Algebra und Analytische Geometrie I*. Vieweg.

Buchberger, B. (1965). *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal*. PhD thesis, Innsbruck.

Cox, D., Little, J., O'Shea, D. (1996). *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer–Verlag, 2. edition.

Eaton, J. W., Bateman, D., Hauberg, S. (2009). *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform. ISBN 1441413006.

Fischer, G. (1984). *Lineare Algebra*. Vieweg.

Forster, O. (1984). *Analysis 2*. Vieweg, 5. edition.

Gass, S. I. (1970a). *An Illustrated Guide to Linear Programming*. McGraw–Hill. Republished by Dover 1990.

Gass, S. I. (1970b). *An Illustrated Guide to Linear Programming*. McGraw–Hill. Republished by Dover, 1990.

Gathen, J. v. z., Gerhard, J. (1999). *Modern Computer Algebra*. Cambridge University Press.

Gautschi, W. (1997). *Numerical Analysis. An Introduction*. Birkhäuser.

Golub, G., van Loan, C. F. (1996). *Matrix Computations*. The Johns Hopkins University Press, 3rd edition.

Heuser, H. (1983). *Lehrbuch der Analysis. Teil 2*. B. G. Teubner, 2. edition.

Higham, N. J. (2002). *Accuracy and stability of numerical algorithms*. SIAM, 2nd edition.

Khinchin, A. Y. (1964). *Continued fractions*. University of Chicago Press, 3rd edition. Reprinted by Dover 1997.

Pschyrembel (1994). *Klinisches Wörterbuch*. Walter de Gruyter & Co, 257 edition.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Sauer, T. (2001). Computeralgebra. Vorlesungsskript, Justus–Liebig–Universität Gießen, Universität Passau.

Sauer, T. (2005). Kettenbrüche und Approximation. Vorlesungsskript, Justus–Liebig–Universität Gießen. http://www.math.uni-giessen.de/tomas.sauer.

Sauer, T. (2013). Einführung in die Numerische Mathematik. Vorlesungsskript, Universität Passau.

Sauer, T. (2015). Analysis 2. Vorlesungsskript, Universität Passau.

Schelter, W. (2001). Maxima - a Computer Algebra System. http://maxima.sourceforge.net.

Wilkinson, J. H. (1984). The perfidious polynomial. In Golub, G. H., editor, *Studies in Numerical Analysis*, volume 24 of *MAA Studies in Mathematics*, pages 1–28. The Mathematical Association of America.

3 Index