

Was man wissen sollte

In diesem Buch werden wir ständig elementaren mathematischen Konzepten begegnen, die der Leser bereits kennen sollte, die ihm aber möglicherweise nicht mehr sofort präsent sind.

In diesem einführenden Kapitel wollen wir diese Kenntnisse auffrischen und neue, für das Wissenschaftliche Rechnen wichtige Konzepte einführen. Wir werden die Bedeutung und die Nützlichkeit dieser mit Hilfe des Programmes MATLAB (Matrix Laboratory) entdecken. In Abschnitt 1.6 werden wir eine kurze Einführung in MATLAB geben, für eine vollständige Beschreibung wollen wir auf die Handbücher [HH00] und [Mat04] verweisen.

In diesem Kapitel fassen wir also die wichtigsten Resultate aus Analysis, Algebra und Geometrie zusammen, aufbereitet für deren Anwendung im Wissenschaftlichen Rechnen.

1.1 Reelle Zahlen

Während allen bekannt sein dürfte, was die Menge der reellen Zahlen \mathbb{R} ist, ist es vielleicht weniger klar, auf welche Art und Weise Computer mit ihnen umgehen. Einerseits ist es unmöglich, auf einer Maschine (mit endlichen Ressourcen) die Unendlichkeit der reellen Zahlen darzustellen, und man wird sich damit zufrieden geben müssen, nur eine Untermenge \mathbb{F} von \mathbb{R} mit endlicher Dimension darstellen zu können, die wir die Menge der *Floating-Point-Zahlen* nennen. Andererseits hat \mathbb{F} andere Eigenschaften als die Menge \mathbb{R} , wie wir in Abschnitt 1.1.2 sehen werden. Der Grund hierfür ist, daß jede reelle Zahl x von der Maschine durch eine gerundete Zahl dargestellt wird, die wir mit $fl(x)$ bezeichnen und *Maschinenzahl* nennen. Diese stimmt nicht notwendigerweise mit dem Ausgangswert x überein.

1.1.1 Wie wir sie darstellen

Um uns des Unterschieds zwischen \mathbb{R} und \mathbb{F} bewußt zu werden, wollen wir anhand einiger Experimente in MATLAB zeigen, wie ein Computer (zum Beispiel ein PC) mit reellen Zahlen umgeht. Wir verwenden MATLAB anstelle einer anderen Programmiersprache (wie etwa Fortran oder C) nur der Einfachheit halber; das Ergebnis hängt nämlich in erster Linie von der Arbeitsweise des Computers ab und nur zu einem sehr kleinen Teil von der verwendeten Programmiersprache.

Betrachten wir die rationale Zahl $x = 1/7$, deren Dezimaldarstellung $0.\overline{142857}$ ist. Beachte, daß der Punkt den Dezimalteil vom ganzen Teil trennt und statt eines Kommas steht. Diese Darstellung ist unendlich, denn nach dem Punkt stehen unendlich viele Ziffern ungleich null. Um auf einer Maschine diese Zahl darzustellen, setzen wir nach dem *Prompt* **>>** (dem Symbol **>>**) den Bruch $1/7$ und erhalten

```
>> 1/7
ans =
    0.1429
```

als Ergebnis, also eine Zahl, die offenbar nur aus vier Dezimalziffern besteht, die letzte sogar falsch im Vergleich zur vierten Ziffer der reellen Zahl. Geben wir nun $1/3$ ein, erhalten wir 0.3333 , wo auch die vierte Ziffer exakt ist. Dieses Verhalten ist dadurch zu erklären, daß die reellen Zahlen am Computer *gerundet* werden, es wird also nur eine feste Anzahl von Dezimalziffern gespeichert und die letzte aufgerundet, falls die darauffolgende Ziffer größer oder gleich 5 ist.

Natürlich können wir uns fragen, wie sinnvoll es ist, nur vier Dezimalziffern für die Darstellung der reellen Zahlen zu verwenden. Wir können aber beruhigt sein, die Zahl die wir „gesehen“ haben, ist nur eines der möglichen *Output*-Formate des Systems, und dieses stimmt nicht mit seiner internen Darstellung überein. Diese verwendet 16 Dezimalziffern (und weitere korrigierende, aber das wollen wir hier nicht weiter ausführen). Dieselbe Zahl nimmt je nach Formatangabe diverse Formen an; zum Beispiel sind für $1/7$ einige mögliche *Output*-Formate:

format

```
format long    ergibt 0.14285714285714,
format short e "    1.4286e - 01,
format long e  "    1.428571428571428e - 01,
format short g "    0.14286,
format long g  "    0.142857142857143.
```

Einige dieser Darstellungen, etwa das Format **long e**, stimmen besser mit der internen Darstellung überein. Im allgemeinen speichert ein Computer eine reelle Zahl auf folgende Weise:

$$x = (-1)^s \cdot (0.a_1a_2 \dots a_t) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}, \quad a_1 \neq 0, \quad (1.1)$$

wobei s entweder 0 oder 1 ist, die positive ganze Zahl β größer gleich 2 die *Basis*, m eine ganze Zahl, die *Mantisse*, deren Länge t der maximalen Anzahl an speicherbaren Ziffern a_i (zwischen 0 und $\beta - 1$) entspricht, und e eine ganze Zahl, der *Exponent*. Das Format `long e` ähnelt dieser Darstellung am meisten (`e` steht für Exponent und dessen Ziffern samt Vorzeichen stehen gleich rechts des Zeichens `e`). Die Maschinenzahlen in Format (1.1) heißen *Floating-Point-Zahlen*, weil die Position des Dezimalpunktes variabel ist.

Die Bedingung $a_1 \neq 0$ verhindert, daß dieselbe Zahl mehrere Darstellungen besitzt. Ohne diese Bedingung könnte $1/10$ zur Basis 10 nämlich auch durch $0.1 \cdot 10^0$ oder $0.01 \cdot 10^1$ dargestellt werden.

Die Menge \mathbb{F} ist also vollständig durch die Basis β , die Anzahl signifikanter Stellen t und durch das Intervall (L, U) (mit $L < 0$ und $U > 0$), in dem der Exponent e variieren kann, bestimmt. Diese Menge wird auch mit $\mathbb{F}(\beta, t, L, U)$ bezeichnet: MATLAB verwendet $\mathbb{F}(2, 53, -1021, 1024)$ (tatsächlich entsprechen die 53 signifikanten Stellen zur Basis 2 den 15 von MATLAB im `format long` angezeigten signifikanten Stellen zur Basis 10).

Glücklicherweise ist der unvermeidbare *Roundoff-Fehler*, der entsteht, wenn man eine reelle Zahl $x \neq 0$ durch ihren Vertreter $fl(x)$ in \mathbb{F} ersetzt, im allgemeinen klein, da

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} \epsilon_M, \quad (1.2)$$

wobei $\epsilon_M = \beta^{1-t}$ dem Abstand zwischen 1 und der nächsten *Floating-Point-Zahl* ungleich 1 entspricht. Beachte, daß ϵ_M von β und t abhängt. In MATLAB erhält man ϵ_M mit dem Befehl `eps` und es ist $\epsilon_M = 2^{-52} \simeq 2.22 \cdot 10^{-16}$. Beachte, daß wir in (1.2) den *relativen Fehler* von x abgeschätzt haben, der sicher aussagekräftiger als der *absolute Fehler* $|x - fl(x)|$ ist, denn dieser zieht ja die Größenordnung von x nicht mit in Betracht.

eps

Die Zahl 0 gehört nicht zu \mathbb{F} , da zur Darstellung dieser in (1.1) $a_1 = 0$ ist: sie wird also extra behandelt. Da überdies L und U endlich sind, lassen sich keine Zahlen mit beliebig großem oder kleinem Absolutbetrag darstellen. So ist die größte bzw. kleinste Zahl von \mathbb{F}

$$x_{min} = \beta^{L-1}, x_{max} = \beta^U (1 - \beta^{-t}).$$

Mit den Befehlen `realmin` und `realmax` ist es in MATLAB möglich, diese Werte festzustellen:

realmin
realmax

$$\begin{aligned} x_{min} &= 2.225073858507201 \cdot 10^{-308}, \\ x_{max} &= 1.7976931348623158 \cdot 10^{+308}. \end{aligned}$$

Eine Zahl kleiner als x_{min} erzeugt einen sogenannten *Underflow* und wird entweder wie 0 oder auf besondere Weise (siehe [QSS02], Kapitel

2) behandelt. Eine positive Zahl größer als x_{max} erzeugt hingegen einen **Inf** *Overflow* und wird in der Variablen **Inf** (am Computer die Darstellung des positiv Unendlichen) gespeichert.

Die Tatsache, daß x_{min} und x_{max} die Extrema eines sehr breiten Intervalls der reellen Zahlengeraden sind, soll uns nicht täuschen: die Zahlen von \mathbb{F} sind sehr dicht gestreut in der Nähe von x_{min} , und immer dünner gestreut in der Nähe von x_{max} . Dies können wir sofort überprüfen, indem wir x_{max-1} und x_{min+1} in \mathbb{F} betrachten:

$$\begin{aligned}x_{max-1} &= 1.7976931348623157 \cdot 10^{+308} \\x_{min+1} &= 2.225073858507202 \cdot 10^{-308},\end{aligned}$$

es ist also $x_{min+1} - x_{min} \simeq 10^{-323}$, während $x_{max} - x_{max-1} \simeq 10^{292}$ (!). Der relative Abstand bleibt jedenfalls klein, wie wir aus (1.2) sehen können.

1.1.2 Wie wir mit Floating-Point-Zahlen rechnen

Kommen wir nun zu den elementaren Operationen in \mathbb{F} : da \mathbb{F} nur eine Teilmenge von \mathbb{R} ist, besitzen diese nicht alle Eigenschaften der in \mathbb{R} definierten Operationen. Genauer gesagt bleibt die Kommutativität der Addition (also $fl(x+y) = fl(y+x) \forall x, y \in \mathbb{F}$) und der Multiplikation ($fl(xy) = fl(yx) \forall x, y \in \mathbb{F}$) erhalten, aber die Eigenschaften Assoziativität und Distributivität sowie die Eindeutigkeit der Zahl 0 gehen verloren.

Um zu sehen, daß die Zahl 0 nicht mehr eindeutig ist, ordnen wir einer Variablen **a** einen beliebigen Wert, etwa 1, zu und führen folgendes Programm aus:

```
>> a = 1; b=1; while a+b ~= a; b=b/2; end
```

In diesem wird die Variable **b** in jedem Schritt halbiert, solange die Summe von **a** und **b** ungleich (=) **a** ist. Würden wir mit reellen Zahlen rechnen, dann würde das Programm niemals abbrechen; in unserem Fall hingegen bricht es nach einer endlichen Anzahl von Schritten ab und liefert für **b** folgenden Wert: $1.1102e-16 = \epsilon_M/2$. Es existiert also mindestens eine Zahl **b** ungleich 0, so daß **a+b=a**. Das ist deswegen möglich, da die Menge \mathbb{F} aus voneinander isolierten Elementen besteht. Summieren wir also zwei Zahlen **a** und **b** mit **b**<**a** und **b** kleiner als ϵ_M , werden wir stets **a+b** gleich **a** erhalten.

Die Assoziativität wird dann verletzt, wenn sich ein *Overflow* oder ein *Underflow* ereignet. Nehmen wir zum Beispiel **a**=1.0e+308, **b**=1.1e+308 und **c**=-1.001e+308 und summieren auf zwei verschiedene Arten. Wir erhalten

$$a + (b + c) = 1.0990e + 308, (a + b) + c = \text{Inf}.$$

Das kann passieren, wenn man zwei Zahlen mit annähernd gleichem Absolutbetrag, aber verschiedenem Vorzeichen addiert. In diesem Fall kann das Ergebnis ziemlich ungenau sein und man spricht von der *Auslöschung signifikanter Stellen*. Führen wir zum Beispiel in MATLAB die Operation $((1+x) - 1)/x$ mit $x \neq 0$ durch, deren exakte Lösung klarerweise für alle $x \neq 0$ gleich 1 ist. Wir erhalten hingegen

```
>> x = 1.e - 15; ((1 + x) - 1)/x
ans =
    1.1102
```

Wie wir sehen, ist das Ergebnis sehr ungenau, der absolute Fehler sehr groß.

Ein weiteres Beispiel für die Auslöschung signifikanter Stellen liefert die Auswertung der Funktion

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \quad (1.3)$$

in 401 äquidistanten Punkten des Intervalls $[1 - 2 \cdot 10^{-8}, 1 + 2 \cdot 10^{-8}]$. Wir erhalten den in 1.1 abgebildeten chaotischen Graphen (der wahre Verlauf ist der von $(x - 1)^7$, also einer auf diesem kleinen Intervall um $x = 1$ annähernd konstanten Funktion gleich null). In Abschnitt 1.4 werden wir die Befehle kennenlernen, mit denen wir den Graphen erzeugt haben.

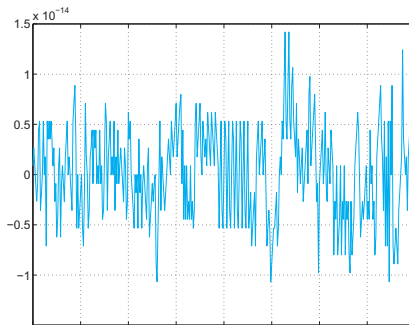


Abb. 1.1. Oszillierender Verlauf der Funktion (1.3), bedingt durch die Auslöschung signifikanter Stellen

Beachte schließlich, daß in \mathbb{F} sogenannte unbestimmte Formen wie etwa $0/0$ oder ∞/∞ nicht vorkommen: in Berechnungen ergeben sie eine sogenannte *Not-a-Number* (NaN in MATLAB), für die die üblichen Rechenregeln nicht gelten. NaN

Bemerkung 1.1 Es ist richtig, daß die Rundungsfehler im allgemeinen klein sind, werden sie aber innerhalb von langen und komplexen Algorithmen wiederholt, können sie katastrophale Auswirkungen haben. Zwei außergewöhnli-

che Ereignisse waren die Explosion der Ariane-Rakete am 4. Juni 1996, verursacht durch einen *Overflow* im Bordcomputer, und der Absturz einer amerikanischen Patriot-Rakete während des ersten Golfkrieges im Jahr 1991 auf eine amerikanische Kaserne aufgrund eines Rundungsfehlers in der Berechnung ihrer Schußbahn.

Ein weniger folgenschweres, aber dennoch erschreckendes Beispiel ergibt sich aus der Berechnung der Folge

$$z_2 = 2, z_{n+1} = 2^{n-1/2} \sqrt{1 - \sqrt{1 - 4^{1-n} z_n^2}}, n = 2, 3, \dots, \quad (1.4)$$

die für n gegen unendlich gegen π konvergiert. Verwenden wir MATLAB für die Berechnung von z_n , stellen wir fest, wie der relative Fehler zwischen π und z_n für die ersten 16 Iterationen sinkt, dann aber aufgrund von Rundungsfehlern wieder steigt (siehe Abbildung 1.2).

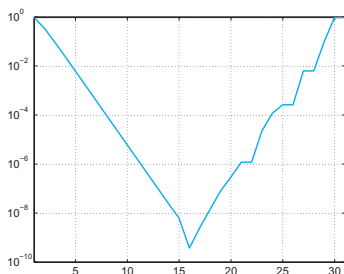


Abb. 1.2. Logarithmus des relativen Fehlers $|\pi - z_n|/|\pi|$ in Abhängigkeit der Iterationsanzahl n



Siehe Aufgaben 1.1–1.2.

1.2 Komplexe Zahlen

Die komplexen Zahlen, deren Menge mit dem Symbol \mathbb{C} bezeichnet wird, sind von der Form $z = x + iy$, wobei $i = \sqrt{-1}$ die komplexe Einheit (also $i^2 = -1$) ist, während $x = \operatorname{Re}(z)$ bzw. $y = \operatorname{Im}(z)$ Realteil bzw. Imaginärteil von z sind. Im allgemeinen werden diese am Computer als Paare reeller Zahlen dargestellt.

Falls nicht anders definiert, bezeichnen beide MATLAB-Variablen `i` und `j` die imaginäre Einheit. Um eine komplexe Zahl mit Realteil `x` und Imaginärteil `y` einzuführen, genügt es, einfach `x+i*y` zu schreiben; alternativ kann man auch den Befehl `complex(x,y)` verwenden. Wir erinnern auch an die trigonometrische Darstellung einer komplexen Zahl z , für die

$$z = \rho e^{i\theta} = \rho(\cos \theta + i \sin \theta) \quad (1.5)$$

gilt, wobei $\rho = \sqrt{x^2 + y^2}$ der Betrag der komplexen Zahl ist (den man mit dem Befehl `abs(z)` erhält) und θ das Argument, also der Winkel zwischen dem Vektor z mit den Komponenten (x, y) und der x-Achse. θ erhält man mit dem Befehl `angle(z)`. Die Darstellung (1.5) ist also

$$\text{abs}(z) * (\cos(\text{angle}(z)) + i * \sin(\text{angle}(z))).$$

Die Polardarstellung (also in Funktion von ρ und θ) einer oder mehrerer komplexer Zahlen erhält man mit dem Befehl `compass(z)`, wobei z eine komplexe Zahl oder ein Vektor komplexer Zahlen sein kann. So erhält man zum Beispiel durch Eingabe von

```
>> z = 3+i*3; compass(z);
```

den in Abbildung 1.3 dargestellten Graphen.

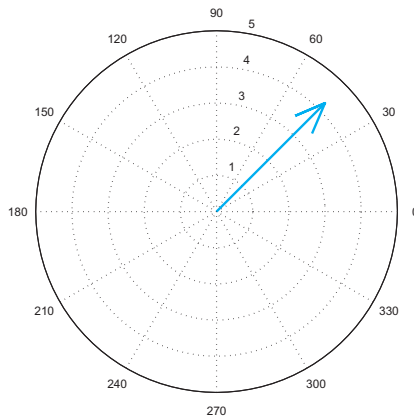


Abb. 1.3. Output des MATLAB-Befehls `compass`

Ist eine komplexe Zahl z gegeben, kann man deren Realteil (bzw. Imaginärteil) mit dem Befehl `real(z)` (bzw. `imag(z)`) extrahieren. Schließlich erhält man die komplex konjugierte Zahl $\bar{z} = x - iy$ von z mit dem Befehl `conj(z)`.

In MATLAB werden alle Operationen unter der impliziten Annahme, daß Ergebnis und Operatoren komplex sind, ausgeführt. So ist es nicht verwunderlich, daß MATLAB für die Kubikwurzel von -5 mit dem Befehl `(-5)^(1/3)` statt der reellen Zahl $-1.7099\dots$ die komplexe Zahl $0.8550 + 1.4809i$ liefert.

Das Symbol \wedge bedeutet Potenzieren. Tatsächlich sind alle Zahlen der Form $\rho e^{i(\theta+2k\pi)}$ mit ganzem k nicht von z unterscheidbar. Berechnen wir jetzt $\sqrt[3]{z}$, erhalten wir $\sqrt[3]{\rho} e^{i(\theta/3+2k\pi/3)}$, das heißt die drei verschiedenen Wurzeln

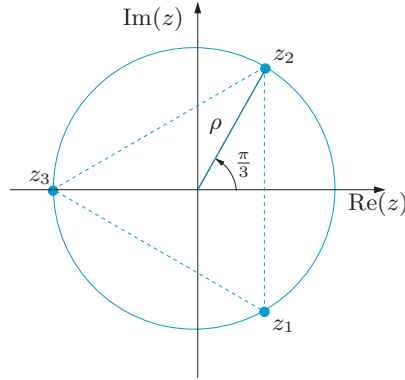


Abb. 1.4. Darstellung der Kubikwurzeln der reellen Zahl -5 in der Gauß-Ebene

$$z_1 = \sqrt[3]{\rho}e^{i\theta/3}, \quad z_2 = \sqrt[3]{\rho}e^{i(\theta/3+2\pi/3)}, \quad z_3 = \sqrt[3]{\rho}e^{i(\theta/3+4\pi/3)}.$$

MATLAB wählt dann jene aus, der als erstes begegnet wird, falls die komplexe Ebene, ausgehend von der reellen Achse, im Gegenuhrzeigersinn aufgespannt wird. Da die Polardarstellung von $z = -5$ gleich $\rho e^{i\theta}$ mit $\rho = 5$ und $\theta = -\pi$ ist, sind die drei Wurzeln

$$z_1 = \sqrt[3]{5}(\cos(-\pi/3) + i \sin(-\pi/3)) \simeq 0.8550 - 1.4809i,$$

$$z_2 = \sqrt[3]{5}(\cos(\pi/3) + i \sin(\pi/3)) \simeq 0.8550 + 1.4809i,$$

$$z_3 = \sqrt[3]{5}(\cos(-\pi) + i \sin(-\pi)) \simeq -1.71.$$

Die zweite Wurzel ist jene, die von MATLAB ausgewählt wird (siehe Abbildung 1.4 für die Darstellung von z_1 , z_2 und z_3 in der Gauß-Ebene).

Schließlich erhalten wir aus (1.5) die Euler-Formel

$$\cos(\theta) = \frac{1}{2} (e^{i\theta} + e^{-i\theta}), \quad \sin(\theta) = \frac{1}{2i} (e^{i\theta} - e^{-i\theta}). \quad (1.6)$$

1.3 Matrizen

Seien n und m zwei positive ganze Zahlen. Eine Matrix A mit m Zeilen und n Spalten ist eine Menge von $m \times n$ Elementen a_{ij} mit $i = 1, \dots, m$, $j = 1, \dots, n$, dargestellt durch folgende Tabelle

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}. \quad (1.7)$$

In kompakter Form schreiben wir $A = (a_{ij})$. Sind die Elemente von A reelle Zahlen, schreiben wir $A \in \mathbb{R}^{m \times n}$; $A \in \mathbb{C}^{m \times n}$, falls sie komplex sind. Ist $n = m$, nennt man die Matrix *quadratisch* der Dimension n . Eine Matrix mit nur einer Spalte ist ein *Spaltenvektor*, während eine Matrix mit nur einer Zeile ein *Zeilenvektor* ist.

Um in MATLAB eine Matrix einzugeben, müssen wir nur die Elemente von der ersten bis zur letzten Zeile eingeben und das Ende jeder Zeile mit `;` markieren. So erzeugt etwa der Befehl

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

die Matrix

```
A =
     1     2     3
     4     5     6
```

mit zwei Zeilen und drei Spalten.

Auf den Matrizen sind einige elementare Operationen definiert. Falls $A = (a_{ij})$ und $B = (b_{ij})$ zwei $(m \times n)$ -Matrizen sind, dann ist:

1. die *Summe* zweier Matrizen A und B die Matrix $A + B = (a_{ij} + b_{ij})$. Die Nullmatrix der Dimension $m \times n$ (bezeichnet mit 0 und erzeugt durch `zeros(m,n)`) ist die Matrix mit ausschließlich Nulleinträgen; zeros
2. das *Produkt* einer Matrix A und einer (reellen oder komplexen) Zahl λ die Matrix $\lambda A = (\lambda a_{ij})$.

Etwas Beachtung müssen wir dem *Matrixprodukt* schenken. Dieses kann nur gebildet werden, falls A und B die Dimension $m \times p$ bzw. $p \times n$ haben, mit einer beliebigen ganzen Zahl p . Das Produkt ist dann eine Matrix C der Dimension $m \times n$ mit den Elementen

$$c_{ij} = \sum_{k=1}^p a_{ik} b_{kj}, \text{ für } i = 1, \dots, m, \quad j = 1, \dots, n.$$

Das neutrale Element der Multiplikation ist eine quadratische Matrix, die *Einheitsmatrix*, und wird mit I bezeichnet; sie hat entlang der Diagonalen Einträge gleich 1 und Elemente gleich 0 sonst (in MATLAB wird die Matrix mit dem Befehl `eye(n)` konstruiert, n ist dabei eye die Dimension).

Summen- und Produktbildung für Matrizen erfolgt in MATLAB mit denselben Operationen wie für reelle Zahlen. Es ist zum Beispiel

```
>> A=[1 2 3; 4 5 6]; B=[7 8 9; 10 11 12]; C=[13 14; 15 16; 17 18];
>> A+B
ans =
     8    10    12
    14    16    18
>> A*C
```

```
ans =
```

```
    94    100
   229    244
```

Der Versuch, die Summe von **A** und **C** zu bilden, führt zu folgender Diagnosemeldung

```
>> A+C
??? Error using ==> +
Matrix dimensions must agree.
```

Ein analoges Resultat würde man beim Versuch erhalten, **A** mit **B** zu multiplizieren.

Die *Inverse* einer quadratischen Matrix der Dimension n , also die eindeutige Matrix $X = A^{-1}$, so daß $AX = XA = I$, existiert nur, falls die *Determinante* von **A** ungleich null ist, also die Zeilenvektoren der Matrix **A** linear unabhängig sind. Die Berechnung der Inversen erfolgt mit dem Befehl `inv(A)`, während der Befehl für die Berechnung der Determinante `det(A)` lautet. An dieser Stelle wollen wir daran erinnern, daß die Determinante einer quadratischen Matrix eine durch folgende Rekursion (Laplace-Regel) definierte Zahl ist

$$\det(A) = \begin{cases} a_{11} & \text{falls } n = 1, \\ \sum_{j=1}^n \Delta_{ij} a_{ij} & \text{für } n > 1, \forall i = 1, \dots, n, \end{cases} \quad (1.8)$$

wobei $\Delta_{ij} = (-1)^{i+j} \det(A_{ij})$ und A_{ij} die Matrix ist, die man durch Streichung der i -ten Zeile und j -ten Spalte aus der Matrix **A** erhält. (Das Ergebnis hängt nicht von der gewählten Zeile i ab.)

Falls $A \in \mathbb{R}^{1 \times 1}$, setzen wir $\det(A) = a_{11}$, falls $A \in \mathbb{R}^{2 \times 2}$, erhält man

$$\det(A) = a_{11}a_{22} - a_{12}a_{21},$$

während für $A \in \mathbb{R}^{3 \times 3}$

$$\begin{aligned} \det(A) = & a_{11}a_{22}a_{33} + a_{31}a_{12}a_{23} + a_{21}a_{13}a_{32} \\ & - a_{11}a_{23}a_{32} - a_{21}a_{12}a_{33} - a_{31}a_{13}a_{22}. \end{aligned}$$

Falls hingegen $A = BC$, dann ist $\det(A) = \det(B) \cdot \det(C)$.

Sehen wir uns ein Beispiel für die Berechnung der Inversen einer (2×2) -Matrix und ihrer Determinante an:

```
>> A=[1 2; 3 4];
>> inv(A)
ans =
   -2.0000    1.0000
```

```

1.5000 -0.5000
>> det(A)
ans =
-2

```

Ist die Matrix singulär, meldet uns MATLAB das Problem und gibt eine Diagnosemeldung zurück, gefolgt von einer Matrix mit Einträgen Inf:

```

>> A=[1 2; 0 0];
>> inv(A)
Warning: Matrix is singular to working precision.
ans =
    Inf    Inf
    Inf    Inf

```

Die Berechnung der Inversen und der Determinante ist für bestimmte Klassen von quadratischen Matrizen besonders einfach. So etwa bei den *Diagonalmatrizen*, für die die a_{kk} mit $k = 1, \dots, n$ die einzigen Elemente ungleich null sind. Diese Elemente bilden die sogenannte Hauptdiagonale der Matrix und es ist

$$\det(A) = a_{11}a_{22} \cdots a_{nn}.$$

Die Diagonalmatrizen sind nicht singulär, falls $a_{kk} \neq 0$ für alle k . In diesem Fall ist die Inverse eine Diagonalmatrix mit den Elementen a_{kk}^{-1} .

In MATLAB ist die Konstruktion einer Diagonalmatrix der Dimension n einfach, es genügt der Befehl `diag(v)`, wobei v ein Vektor der Dimension n ist, der nur die Diagonalelemente enthält. Mit dem Befehl `diag(v,m)` hingegen wird eine quadratische Matrix der Dimension $n+\text{abs}(m)$ erzeugt, die in der m -ten oberen Nebendiagonalen (oder unteren Nebendiagonalen, falls m negativ ist) die Elemente des Vektors v enthält. diag

Ist etwa $v = [1 \ 2 \ 3]$, so ist

```

>> A=diag(v,-1)
A =
    0    0    0    0
    1    0    0    0
    0    2    0    0
    0    0    3    0

```

Andere Matrizen, für die die Berechnung der Determinante einfach ist, sind die *obere Dreiecksmatrix* und die *untere Dreiecksmatrix*; eine quadratische Matrix der Dimension n heißt untere Dreiecksmatrix (bzw. obere Dreiecksmatrix), falls alle Elemente oberhalb (bzw. unterhalb) der Hauptdiagonalen null sind. Die Determinante ist dann einfach das Produkt der Diagonalelemente.

Mit den MATLAB-Befehlen `tril(A)` und `triu(A)` ist es möglich, aus der Matrix **A** der Dimension **n** deren untere bzw. obere Dreiecksmatrix zu extrahieren. In der erweiterten Form `tril(A,m)` bzw. `triu(A,m)`, mit **m** zwischen $-n$ und **n**, lassen sich Dreiecksmatrizen unterhalb bzw. oberhalb der einschließlich *m*-ten Nebendiagonalen extrahieren.

Betrachten wir zum Beispiel die Matrix **A** = [3 1 2; -1 3 4; -2 -1 3], dann erhalten wir mit dem Befehl **L1=tril(A)** die untere Dreiecksmatrix

```
L1 =
     3     0     0
    -1     3     0
    -2    -1     3
```

Geben wir hingegen **L2=tril(A,1)** ein, erhalten wir folgende Matrix

```
L2 =
     3     1     0
    -1     3     4
    -2    -1     3
```

Eine spezielle Matrixoperation ist das *Transponieren*: ist eine Matrix $A \in \mathbb{R}^{n \times m}$ gegeben, bezeichnen wir mit $A^T \in \mathbb{R}^{m \times n}$ ihre transponierte Matrix, die man durch Vertauschen der Zeilen und Spalten von **A** erhält. Falls in MATLAB **A** eine reelle Matrix ist, bezeichnet **A'** ihre Transponierte. Falls $A = A^T$, dann heißt **A** *symmetrisch*.

1.3.1 Vektoren

Wir bezeichnen Vektoren mit Fettbuchstaben. So bezeichnet **v** stets einen Spaltenvektor, dessen *i*-te Komponente wir mit v_i bezeichnen. Falls ein Vektor *n* reelle Komponenten hat, schreiben wir $\mathbf{v} \in \mathbb{R}^n$.

MATLAB behandelt Vektoren wie spezielle Matrizen. Um einen Spaltenvektor zu definieren, müssen wir in eckigen Klammern nur die Werte der einzelnen Komponenten des Vektors, getrennt durch Strichpunkte, eingeben, während wir für einen Zeilenvektor nur die Werte, getrennt durch Leerzeichen oder Beistriche, eingeben müssen. So definieren zum Beispiel die Befehle **v = [1;2;3]** und **w = [1 2 3]** einen Spalten- bzw. Zeilenvektor der Dimension 3. Der Befehl **zeros(n,1)** (bzw. **zeros(1,n)**) erzeugt einen Spaltenvektor (bzw. Zeilenvektor) der Dimension **n** mit nur Nullelementen; diesen bezeichnen wir mit **0**. In analoger Weise erzeugt der Befehl **ones(n,1)** einen Spaltenvektor mit Einselementen, mit **1** bezeichnet.

Unter den Vektoren sind besonders die voneinander *linear unabhängig* wichtig; ein System von Vektoren $\{\mathbf{y}_1, \dots, \mathbf{y}_m\}$ heißt linear unabhängig, falls die Beziehung

$$\alpha_1 \mathbf{y}_1 + \dots + \alpha_m \mathbf{y}_m = \mathbf{0}$$

gilt. Diese kann nur erfüllt werden, falls alle Koeffizienten $\alpha_1, \dots, \alpha_m$ null sind. Eine Menge n linear unabhängiger Vektoren $\mathcal{B} = \{\mathbf{y}_1, \dots, \mathbf{y}_n\}$ in \mathbb{R}^n (oder \mathbb{C}^n) bildet eine *Basis* von \mathbb{R}^n (bzw. \mathbb{C}^n) und besitzt die Eigenschaft, daß jeder Vektor \mathbf{w} von \mathbb{R}^n in eindeutiger Weise als

$$\mathbf{w} = \sum_{k=1}^n w_k \mathbf{y}_k$$

geschrieben werden kann. Die Zahlen w_k heißen *Komponenten* von \mathbf{w} bezüglich der Basis \mathcal{B} . Zum Beispiel besteht die kanonische Basis des \mathbb{R}^n aus den Vektoren $\{\mathbf{e}_1, \dots, \mathbf{e}_n\}$, wobei die i -te Komponente von \mathbf{e}_i gleich 1 ist und die übrigen Komponenten gleich 0 sind. Das ist nicht die einzige Basis des \mathbb{R}^n , aber jene, die wir im allgemeinen verwenden werden.

Was die Operationen zwischen Vektoren gleicher Dimension betrifft, wollen wir hier besonders auf das *Skalarprodukt* und das *Vektorprodukt* hinweisen. Ersteres ist definiert als

$$\mathbf{v}, \mathbf{w} \in \mathbb{R}^n, (\mathbf{v}, \mathbf{w}) = \mathbf{w}^T \mathbf{v} = \sum_{k=1}^n v_k w_k,$$

wobei $\{v_k\}$ und $\{w_k\}$ die Komponenten von \mathbf{v} bzw. \mathbf{w} sind. In MATLAB kann diese Operation zwischen Vektoren mit dem Befehl `w'*v` (oder mit dem eigenen Befehl `dot(v,w)`) durchgeführt werden, wobei der Apostroph den Vektor \mathbf{w} transponiert. Die Länge (oder der Betrag) eines Vektors \mathbf{v} ist dann gegeben durch

dot

$$\|\mathbf{v}\| = \sqrt{(\mathbf{v}, \mathbf{v})} = \sqrt{\sum_{k=1}^n v_k^2}$$

und kann mit dem Befehl `norm(v)` berechnet werden. Das Vektorprodukt zweier Vektoren $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ ist hingegen gegeben durch den Vektor $\mathbf{u} \in \mathbb{R}^n$ (bezeichnet mit $\mathbf{u} = \mathbf{v} \times \mathbf{w}$ oder $\mathbf{u} = \mathbf{v} \wedge \mathbf{w}$), der sowohl zu \mathbf{v} als auch zu \mathbf{w} orthogonal steht und dessen Länge $|\mathbf{u}| = |\mathbf{v}| |\mathbf{w}| \sin(\alpha)$ ist, wobei α der Winkel zwischen \mathbf{v} und \mathbf{w} ist. In MATLAB wird das Vektorprodukt mit dem Befehl `cross(v,w)` berechnet. Die Visualisierung von Vektoren in MATLAB erfolgt in \mathbb{R}^2 mit dem Befehl `quiver` und in \mathbb{R}^3 mit `quiver3`.

norm

cross
quiver
quiver3

Oft kommen in den vorgestellten MATLAB-Programmen Operationen zwischen Vektoren vor, die von einem Punkt angeführt werden, wie zum Beispiel in `x.*y` oder `x.^2`. Diese Schreibweise bedeutet, daß die Operation nicht im herkömmlichen Sinn durchgeführt wird, sondern komponentenweise. So ergibt `x.*y` nicht das Skalarprodukt zweier Vektoren \mathbf{x} und \mathbf{y} , sondern einen Vektor, dessen i -te Komponente gleich $x_i y_i$ ist. Definieren wir zum Beispiel die Vektoren

.*
.^

```
>> v = [1; 2; 3]; w = [4; 5; 6];
```

so sind das Skalarprodukt und das komponentenweise Produkt

```
>> w'*v
```

```
ans =
```

```
32
```

```
>> w.*v
```

```
ans =
```

```
4    10    18
```

Beachte, daß das Produkt $w*v$ nicht definiert ist, falls die Vektoren die falsche Dimension haben.

Ein Vektor $v \in \mathbb{R}^n$ mit $v \neq 0$ ist der zur komplexen Zahl λ gehörige *Eigenvektor* der Matrix $A \in \mathbb{R}^{n \times n}$, falls

$$Av = \lambda v.$$

Die Zahl λ heißt *Eigenwert* von A . Die Berechnung der Eigenwerte einer Matrix ist im allgemeinen recht kompliziert; nur für Diagonalmatrizen und Dreiecksmatrizen entsprechen die Eigenwerte den Diagonalelementen.



Siehe Aufgaben 1.3–1.5.

1.4 Reelle Funktionen

Reelle Funktionen werden Hauptgegenstand einiger Kapitel dieses Buches sein. Im speziellen wollen wir für eine auf einem Intervall (a, b) definierte Funktion f deren Nullstellen, ihr Integral und ihre Ableitung berechnen und annähernd ihren Verlauf kennen.

fplot

Der Befehl `fplot(fun,lims)` zeichnet in einem Intervall, dessen Grenzen die beiden Komponenten des Vektors `lims` sind, den Graphen der Funktion `fun` (in einer Zeichenkette gespeichert).

Wollen wir zum Beispiel $f(x) = 1/(1+x^2)$ auf $(-5, 5)$ darstellen, schreiben wir

```
>> fun = '1/(1+x^2)'; lims=[-5,5]; fplot(fun,lims);
```

Alternativ können wir direkt

```
>> fplot('1/(1+x^2)',[-5 5]);
```

eingeben. Die erhaltene Darstellung ist eine Annäherung des Graphen von f (mit einer Toleranz von 0.2%), die sich aus der Auswertung der Funktion auf einer Menge von nicht äquidistanten Abszissenwerten ergibt. Um die Genauigkeit der Darstellung zu erhöhen, können wir `fplot` folgendermaßen aufrufen:

>> `fplot(fun,lims,tol,n,LineStyle)`

wobei `tol` die gewünschte relative Toleranz ist. Der Parameter `n` (≥ 1) garantiert, daß der Graph der Funktion mit mindestens `n+1` Punkten gezeichnet wird; `LineStyle` gibt hingegen die Art (oder die Farbe) der gezeichneten Linie an (zum Beispiel `LineStyle='--'` für eine unterbrochene Linie, `LineStyle='r-.'` für eine rote Strichpunktlinie). Sollen die voreingestellten (*Default*-) Werte für jede dieser Variablen verwendet werden, kann man eine leere Matrix (in MATLAB `[]`) übergeben.

Den Wert $f(x)$ in einem Punkt x (oder auf einer Menge von Punkten, wieder gespeichert in einem Vektor x) erhält man nach Eingabe von x mit dem Befehl `y=eval(fun)`. In y werden die zugehörigen Ordinaten gespeichert. Um diesen Befehl zu verwenden, muß die Zeichenkette `fun` (die den Ausdruck f bestimmt) ein Ausdruck in der Variablen x sein. Anderenfalls müssen wir den Befehl `eval` durch den Befehl `feval` ersetzen.

`eval`

`feval`

Um schließlich wie in Abbildung 1.1 ein Referenzgitter einzuzichnen, müssen wir nach dem Befehl `fplot` den Befehl `grid on` eingeben.

`grid`

1.4.1 Nullstellen

Aus dem Graphen einer Funktion können wir, wenn auch nur annähernd, deren *Nullstellen* ablesen; wir erinnern daran, daß α eine Nullstelle von f ist, falls $f(\alpha) = 0$. Eine Nullstelle wird außerdem *einfach* genannt, falls $f'(\alpha) \neq 0$, anderenfalls *mehrfach*.

Oft ist die Berechnung von Nullstellen kein Problem. Falls die Funktion ein Polynom mit reellen Koeffizienten vom Grad n ist, also von der Form

$$p_n(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n = \sum_{k=0}^n a_kx^k, \quad a_k \in \mathbb{R}, \quad a_n \neq 0,$$

kann man für $n = 1$ (der Graph von p_1 ist eine Gerade) die einzige Nullstelle $\alpha = -a_0/a_1$ leicht berechnen, genauso wie die zwei Nullstellen α_+ und α_- für $n = 2$ (der Graph von p_2 ist eine Parabel)

$$\alpha_{\pm} = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0a_2}}{2a_2}.$$

Es ist auch bekannt, daß es für $n \geq 5$ keine allgemeinen Formeln gibt, mit denen man in endlich vielen Schritten die Wurzeln eines beliebigen Polynoms p_n berechnen kann.

Auch die Anzahl der Nullstellen einer Funktion ist auf elementare Weise *a priori* nicht feststellbar; die Ausnahme sind dabei wieder die Polynome, für die die Anzahl von (reellen oder komplexen) Nullstellen gleich dem Grad des Polynoms ist. Außerdem weiß man, daß, falls ein

Polynom mit reellen Koeffizienten eine komplexe Wurzel $\alpha = x + iy$ hat, auch die komplex konjugierte $\bar{\alpha} = x - iy$ von α eine Wurzel des Polynoms ist.

Die Berechnung einer (nicht aller) Nullstelle einer Funktion f , in `fun` gespeichert, in der Nähe eines reellen oder komplexen Wertes `x0` kann in `fzero` MATLAB mit dem Befehl `fzero(fun,x0)` erfolgen. Neben dem Wert der approximierten Nullstelle wird auch das Intervall ausgegeben, in dem die Funktion die Nullstelle gesucht hat. Rufen wir den Befehl hingegen mit `fzero(fun,[x0 x1])` auf, wird die Nullstelle von `fun` im Intervall mit den Grenzen `x0,x1` gesucht, vorausgesetzt, f wechselt zwischen `x0` und `x1` das Vorzeichen.

Betrachten wir zum Beispiel die Funktion $f(x) = x^2 - 1 + e^x$; aus einer graphischen Untersuchung sehen wir, daß sie zwei Nullstellen im Intervall $(-1, 1)$ hat. Um diese zu berechnen, führen wir folgende Befehle aus:

```
>> fun='x^2 - 1 + exp(x)';
>> fzero(fun,1)
Zero found in the interval: [-0.28, 1.9051].
ans =
    6.0953e-18
>> fzero(fun,-1)
Zero found in the interval: [-1.2263, -0.68].
ans =
   -0.7146
```

Da wir am Graphen gesehen haben, daß sich eine Nullstelle in $[-1, -0.2]$, die andere in $[-0.2, 1]$ befindet, können wir auch schreiben:

```
>> fzero(fun,[-0.2 1])
Zero found in the interval: [-0.2, 1].
ans =
   -4.2658e-17
>> fzero(fun,[-1 -0.2])
Zero found in the interval: [-1, -0.2].
ans =
   -0.7146
```

Wie wir sehen, ist das für die erste Wurzel erhaltene Ergebnis nicht gleich dem vorher berechneten (obwohl beide annähernd gleich null sind). Der Grund hierfür ist, daß der in `fzero` implementierte Algorithmus in beiden Fällen jeweils verschieden funktioniert. In Kapitel 2 werden wir einige Verfahren zur Berechnung der Nullstellen einer Funktion kennenlernen.

1.4.2 Polynome

Wie wir schon gesehen haben, sind Polynome recht spezielle Funktionen. Im folgenden bezeichnen wir mit \mathbb{P}_n die Menge der Polynome vom Grad n . Für diese gibt es in MATLAB eine Reihe spezieller Befehle, die in der Toolbox `polyfun` zusammengefaßt sind.

Wenden wir uns den wichtigsten zu. Der erste Befehl, `polyval`, dient zur Auswertung eines Polynoms in einem oder mehreren Punkten und bedarf zweier Vektoren `p` und `x` als Eingabeparameter. In `p` sind die Koeffizienten des Polynoms in der Reihenfolge von a_n bis a_0 gespeichert, in `x` hingegen die Abszissenwerte, in denen wir das Polynom auswerten wollen. Das Ergebnis kann in einem Vektor `y` gespeichert werden, indem wir

```
>> y = polyval(p,x)
```

schreiben. Zum Beispiel können wir für das Polynom $p(x) = x^7 + 3x^2 - 1$ dessen Werte in den äquidistanten Knoten $x_k = -1 + k/4$, $k = 0, \dots, 8$ mit folgenden Befehlen berechnen:

```
>> p = [1 0 0 0 0 3 0 -1]; x = [-1:0.25:1];
>> y = polyval(p,x)
y =
Columns 1 through 7
    1.0000    0.5540   -0.2578   -0.8126   -1.0000   -0.8124   -0.2422
Columns 8 through 9
    0.8210    3.0000
```

Natürlich könnte man für die Auswertung eines Polynoms auch den Befehl `fplot` verwenden; dieser ist aber im allgemeinen recht unbequem zu verwenden, weil wir in der Zeichenkette, die die zu zeichnende Funktion definiert, den analytischen Ausdruck des Polynoms und nicht nur seine Koeffizienten angeben müssen.

Das Programm `roots` dient hingegen zur angenäherten Berechnung der Nullstellen eines Polynoms und verlangt als Eingabeparameter nur den Vektor `p`. Wir möchten noch einmal daran erinnern, daß α Nullstelle von p genannt wird, falls $p(\alpha) = 0$ ist, oder, äquivalent dazu, *Wurzel* der Gleichung $p(x) = 0$. Zum Beispiel können wir für das Polynom $p(x) = x^3 - 6x^2 + 11x - 6$ die Nullstellen folgendermaßen berechnen:

```
>> p = [1 -6 11 -6]; format long;
>> roots(p)
ans =
    3.000000000000000
    2.000000000000000
    1.000000000000000
```

In diesem Fall erhalten wir die exakten Nullstellen.



Aber nicht immer ist das Ergebnis so genau: so erhalten wir etwa für das Polynom $p(x) = (x-1)^7$, dessen einzige Nullstelle $\alpha = 1$ ist, folgende Nullstellen (einige davon sind sogar komplex)

```
>> p = [1 -7 21 -35 35 -21 7 -1];
>> roots(p)
ans =
    1.0088
    1.0055 + 0.0069i
    1.0055 - 0.0069i
    0.9980 + 0.0085i
    0.9980 - 0.0085i
    0.9921 + 0.0038i
    0.9921 - 0.0038i
```

Eine mögliche Erklärung dieses Verhaltens ist die Fortpflanzung der Rundungsfehler bei der Berechnung der Nullstellen des Polynoms. Die Koeffizienten von p können nämlich alternierende Vorzeichen haben, was zu großen Fehlern führen kann, da signifikante Stellen ausgelöscht werden.

conv Weiters können wir mit dem Befehl `p=conv(p1,p2)` die Koeffizienten des aus dem Produkt der beiden Polynome mit den Koeffizienten `p1` und `p2` erhaltenen Polynoms berechnen. Der Befehl `[q,r]=deconv(p1,p2)` hingegen berechnet die Koeffizienten `q` und den Rest `r` nach Division von `p1` durch `p2`, so daß `p1 = conv(p2,q) + r` ist.

Betrachten wir zum Beispiel die Polynome $p_1(x) = x^4 - 1$ und $p_2(x) = x^3 - 1$ und berechnen Produkt und Quotienten:

```
>> p1 = [1 0 0 0 -1];
>> p2 = [1 0 0 -1];
>> p=conv(p1,p2)
p =
    1    0    0   -1   -1    0    0    1
>> [q,r]=deconv(p1,p2)
q =
    1    0
r =
    0    0    0    1   -1
```

Wir erhalten die Polynome $p(x) = p_1(x)p_2(x) = x^7 - x^4 - x^3 + 1$, $q(x) = x$ und $r(x) = x - 1$, so daß $p_1(x) = q(x)p_2(x) + r(x)$.

polyint Schließlich liefern die Befehle `polyint(p)` und `polyder(p)` die Koeffizienten der Stammfunktion (die in $x = 0$ verschwindet) bzw. die Koeffizienten der Ableitung des Polynoms, dessen Koeffizienten wiederum durch die Komponenten des Vektors `p` gegeben sind.

Wir fassen die oben kennengelernten Befehle in Tabelle 1.1 noch einmal zusammen: in dieser bezeichnet `x` einen Vektor mit Abszissenwerten,

während \mathbf{p} , \mathbf{p}_1 und \mathbf{p}_2 Vektoren sind, die die Koeffizienten der Polynome P , P_1 und P_2 enthalten.

Tabelle 1.1. Die wichtigsten MATLAB-Befehle zu Polynomen

Befehl	Ergebnis
<code>y=polyval(p,x)</code>	\mathbf{y} = Werte von $P(x)$
<code>z=roots(p)</code>	\mathbf{z} = Wurzeln von P , so daß $P(z) = 0$
<code>p=conv(p1,p2)</code>	\mathbf{p} = Koeffizienten des Polynoms $P_1 P_2$
<code>[q,r]=deconv(p1,p2)</code>	\mathbf{q} = Koeffizienten von Q , \mathbf{r} = Koeffizienten von R so daß $P_1 = Q P_2 + R$
<code>y=polyder(p)</code>	\mathbf{y} = Koeffizienten von $P'(x)$
<code>y=polyint(p)</code>	\mathbf{y} = Koeffizienten von $\int_0^x P(t) \, dt$

Der Befehl `polyfit` ermöglicht die Berechnung der $n + 1$ Koeffizienten eines Polynoms p vom Grad n , wenn man die Werte des Polynoms p in $n + 1$ verschiedenen Punkten kennt (siehe Abschnitt 3.1.1). `polyfit`

1.4.3 Integration und Differentiation

Die zwei folgenden Resultate sind fundamental für die Integral- und Differentialrechnung, wir werden sie in diesem Buch noch öfters gebrauchen:

- 1. der *Hauptsatz der Differential- und Integralrechnung*: falls f eine stetige Funktion im Intervall $[a, b)$ ist, dann ist

$$F(x) = \int_a^x f(t) \, dt$$

eine differenzierbare Funktion, *Stammfunktion* von f genannt, und erfüllt $\forall x \in [a, b)$:

$$F'(x) = f(x);$$

- 2. der *erste Mittelwertsatz der Integralrechnung*: falls f eine stetige Funktion im Intervall $[a, b)$ ist und $x_1, x_2 \in [a, b)$, dann $\exists \xi \in (x_1, x_2)$, so daß

$$f(\xi) = \frac{1}{x_2 - x_1} \int_{x_1}^{x_2} f(t) \, dt.$$

Auch wenn eine Stammfunktion existiert, kann es oft schwierig oder unmöglich sein, sie zu berechnen. So spielt es zum Beispiel keine Rolle, ob wir wissen, daß die Stammfunktion von $1/x$ gleich $\ln|x|$ ist, wenn wir den Logarithmus dann nicht effektiv berechnen können. In Kapitel 4 werden wir Approximationsverfahren kennenlernen, mit denen wir das Integral einer stetigen Funktion mit gewünschter Genauigkeit berechnen können, ohne ihre Stammfunktion zu kennen.

Wir wollen daran erinnern, daß eine Funktion f , definiert auf einem Intervall $[a, b]$, im Punkt $\bar{x} \in (a, b)$ ableitbar oder differenzierbar ist, falls der folgende Grenzwert existiert und endlich ist:

$$f'(\bar{x}) = \lim_{h \rightarrow 0} \frac{f(\bar{x} + h) - f(\bar{x})}{h}. \quad (1.9)$$

Die geometrische Interpretation der Ableitung als Steigung der Tangente der Funktion f in \bar{x} spielt eine wichtige Rolle in der Herleitung des Newton-Verfahrens zur Approximation der Wurzeln von f . Wir sagen, daß eine auf dem ganzen Intervall $[a, b]$ stetig differenzierbare Funktion zum Raum $C^1([a, b])$ gehört. Im allgemeinen sagt man, eine bis zur Ordnung p (ein positives Ganzes) stetig differenzierbare Funktion gehört zu $C^p([a, b])$. Im speziellen gehört eine nur stetige Funktion zu $C^0([a, b])$.

Ein weiteres Ergebnis aus der Analysis, das wir oft verwenden werden, ist der *Mittelwertsatz*, der besagt: falls $f \in C^1([a, b])$, dann existiert ein Punkt $\xi \in (a, b)$, so daß

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}.$$

Schließlich wollen wir daran erinnern, daß eine Funktion mit bis zur Ordnung $n + 1$ stetigen Ableitungen in x_0 in einer Umgebung von x_0 durch das sogenannte *Taylor-Polynom vom Grad n* im Punkt x_0 approximiert werden kann:

$$\begin{aligned} T_n(x) &= f(x_0) + (x - x_0)f'(x_0) + \dots + \frac{1}{n!}(x - x_0)^n f^{(n)}(x_0) \\ &= \sum_{k=0}^n \frac{(x - x_0)^k}{k!} f^{(k)}(x_0). \end{aligned}$$

diff In MATLAB kann man mit den Befehlen **diff**, **int** und **taylor** aus der Toolbox **symbolic** analytisch die Ableitung, das unbestimmte Integral (also die Stammfunktion) und das Taylor-Polynom von einfachen Funktionen berechnen. Haben wir den Ausdruck der Funktion, die man manipulieren will, in der Zeichenkette **f** gespeichert, dann berechnet **diff(f,n)** die **n**-te Ableitung, **int(f)** das Integral und **taylor(f,x,n+1)** das Taylor-Polynom vom Grad **n** in einer Umgebung von $x_0 = 0$. Die Variable **x** muß mit dem Befehl **syms x** *symbolisch*

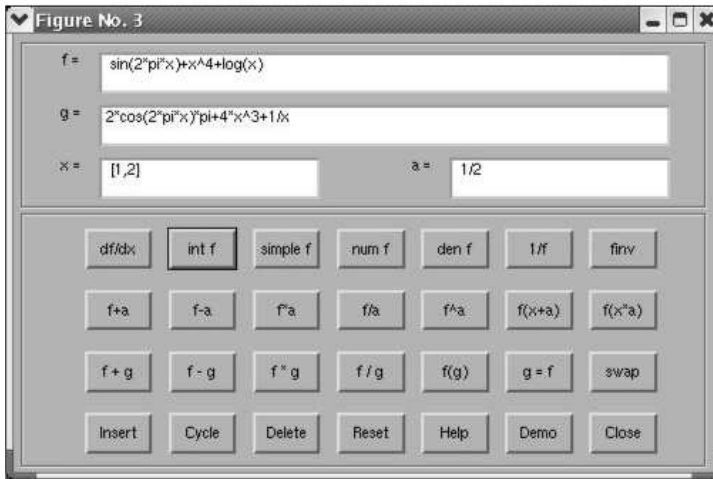


Abb. 1.5. Graphische Oberfläche von funtool

deklariert werden. Auf diese Weise kann sie algebraisch manipuliert werden, ohne ausgewertet zu werden.

Nehmen wir zum Beispiel an, wir wollen die Ableitung, das unbestimmte Integral und das Taylor-Polynom fünfter Ordnung der Funktion $f(x) = (x^2 + 2x + 2)/(x^2 - 1)$ berechnen. Dazu genügen folgende Befehle:

```
>> f = '(x^2+2*x+2)/(x^2-1)';
>> syms x
>> diff(f)
(2*x+2)/(x^2-1)-2*(x^2+2*x+2)/(x^2-1)^2*x
>> int(f)
x+5/2*log(x-1)-1/2*log(1+x)
>> taylor(f,x,6)
-2-2*x-3*x^2-2*x^3-3*x^4-2*x^5
```

Mit dem Befehl `simple` kann man die von `diff`, `int` und `taylor` erzeugten Ausdrücke vereinfachen. Mit dem Befehl `funtool` kann man schließlich über die in Abbildung 1.5 dargestellte graphische Oberfläche Funktionen symbolisch manipulieren und deren wichtigste Eigenschaften untersuchen.

`simple`
`funtool`



Siehe Aufgaben 1.6–1.7.

1.5 Irren ist nicht nur menschlich

Während es im Lateinischen heißt „errare humanum est“, müssen wir für das Wissenschaftliche Rechnen eingestehen, daß Irren unvermeidbar ist.

Wie wir nämlich gesehen haben, verursacht die bloße Darstellung reeller Zahlen am Computer bereits Fehler. Wir müssen also versuchen, nicht Fehler zu vermeiden, sondern mit ihnen zu leben und sie unter Kontrolle zu halten.

Im allgemeinen können wir bei der Approximation und Lösung eines physikalischen Problems verschiedene Stufen von Fehlern unterscheiden (siehe Abbildung 1.6).

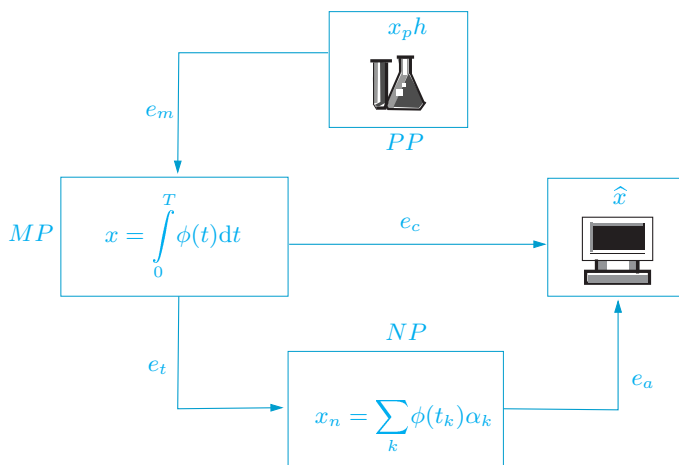


Abb. 1.6. Verschiedene Arten von Fehlern im Laufe einer Berechnung

Auf höchster Stufe stehen die Fehler e_m , die man begeht, wenn man die physikalische Realität (PP steht für physikalisches Problem und $x_p h$ ist dessen Lösung) durch ein mathematisches Modell (MP, dessen Lösung x ist) ersetzt. Die Anwendbarkeit des mathematischen Modells wird durch diese Fehler begrenzt, diese entziehen sich nämlich der Kontrolle des Wissenschaftlichen Rechnens.

Das mathematische Modell (zum Beispiel ausgedrückt durch ein Integral wie in Abbildung 1.6, durch eine algebraische Gleichung, eine Differentialgleichung oder ein lineares oder nichtlineares System) ist im allgemeinen nicht analytisch lösbar. Dessen Lösung mittels Computeralgorithmen führt dann zu Rundungsfehlern, die sich fortpflanzen können. Diese Fehler bezeichnen wir mit e_a . Zu diesen kommen noch weitere Fehler dazu, denn jede Operation im mathematischen Modell, die einen Grenzübergang erfordert, kann ja am Computer nicht exakt durchgeführt werden, sondern nur angenähert werden. Denken wir etwa an die Berechnung der Summe einer Reihe. Diese können wir nur annähern, indem wir die unendliche Reihe nach bereits endlich vielen Gliedern abbrechen. Wir müssen also ein numerisches Problem NP einführen, dessen Lösung x_n sich um einen Fehler e_t von x unterscheidet, der *Abbruchfehler*

(*truncation error*) genannt wird. Diese Fehler treten in mathematischen Problemen endlicher Dimension (wie zum Beispiel der Lösung eines linearen Systems) nicht auf. Die Fehler e_a und e_t ergeben zusammen den *Berechnungsfehler* (*computational error*) e_c , der in unserem Interesse steht.

Wenn wir mit x die exakte Lösung des mathematischen Modells und mit \hat{x} die numerische Lösung bezeichnen, ergibt sich der absolute Rechenfehler aus

$$e_c^{abs} = |x - \hat{x}|,$$

während der relative (falls $x \neq 0$)

$$e_c^{rel} = |x - \hat{x}|/|x|$$

ist, wobei $|\cdot|$ der Absolutbetrag (oder ein anderes Maß je nach der Bedeutung von x) ist.

Ein numerischer Prozeß besteht im allgemeinen in der Approximation eines mathematischen Modells in Funktion eines positiven Diskretisierungsparameters h . Falls nun der numerische Prozeß die Lösung des mathematischen Modells liefert, wenn h gegen 0 strebt, dann sagen wir, daß der numerische Prozeß *konvergent* ist. Falls der absolute oder relative Fehler in Funktion von h durch

$$e_c \leq Ch^p \tag{1.10}$$

beschränkt ist, wobei C eine von h und p unabhängige (im allgemeinen ganze) Zahl ist, dann sagen wir, daß das Verfahren *konvergent von der Ordnung p* ist. Oft kann man das Symbol \leq sogar durch das Symbol \simeq ersetzen, und zwar falls neben der oberen Schranke (1.10) auch eine untere Schranke $C'h^p \leq e_c$ existiert, wobei C' eine andere Konstante ($\leq C$) unabhängig von h und p ist.

Beispiel 1.1 Approximieren wir die Ableitung einer Funktion f in einem Punkt \bar{x} mit dem Inkrement (1.9). Falls f in \bar{x} differenzierbar ist, strebt der Fehler, den wir durch Ersetzen von $f'(\bar{x})$ durch das Inkrement gemacht haben, gegen 0, falls h gegen 0 geht. Wie wir in Abschnitt 4.1 sehen werden, kann dieser nur dann durch Ch nach oben abgeschätzt werden, falls $f \in C^2$ in einer Umgebung von \bar{x} .

In unseren Konvergenzuntersuchungen werden wir oft Graphen lesen müssen, die den Fehler als Funktion von h auf logarithmischer Skala wiedergeben, auf der Abszissenachse $\log(h)$ und auf der Ordinatenachse $\log(e_c)$. Der Vorteil dieser Darstellung ist leicht einzusehen: falls $e_c \simeq Ch^p$, dann ist $\log e_c \simeq \log C + p \log h$. Folglich ist p in logarithmischer Skala gleich der Steigung der Geraden $\log e_c$, und falls wir nun zwei Verfahren vergleichen wollen, hat jenes mit der Geraden mit größerer Steigung die höhere Ordnung. Um in MATLAB Graphen in logarithmischer Skala zu zeichnen, genügt der Befehl `loglog(x,y)`, wobei

\mathbf{x} und \mathbf{y} die Vektoren sind, die die Abszissen- bzw. Ordinatenwerte der darzustellenden Daten enthalten.

`loglog`

In Abbildung 1.7 sind die zu zwei verschiedenen Verfahren gehörenden Graphen des Fehlerverlaufs dargestellt. Jenes in durchgezogener Linie ist erster, jenes in unterbrochener Linie zweiter Ordnung.

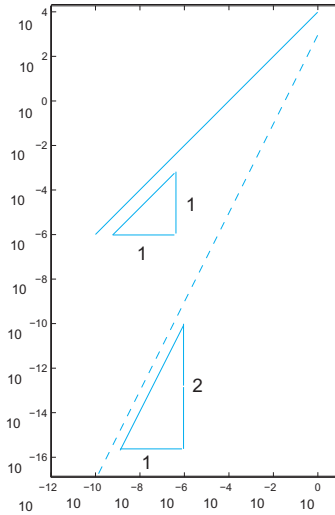


Abb. 1.7. Graphen in logarithmischer Skala

Eine andere Möglichkeit zur Bestimmung der Konvergenzordnung eines Verfahrens, für das man den Fehler e_i für gewisse Werte h_i , mit $i = 1, \dots, N$, des Diskretisierungsparameters kennt, beruht auf der Annahme $e_i \simeq Ch_i^p$ mit C unabhängig von i . Dann kann man p über die Werte

$$p_i = \log(e_i/e_{i-1}) / \log(h_i/h_{i-1}), \quad i = 2, \dots, N \quad (1.11)$$

abschätzen. Eigentlich ist der Fehler ja eine nicht berechenbare Größe, die von der Unbekannten des Problems selbst abhängt. Deshalb müssen wir berechenbare Größen einführen, die zur Abschätzung des Fehlers verwendet werden können, sogenannte *Fehlerschätzer*. In den Abschnitten 2.2, 2.3 und 4.3 werden wir Beispiele für diese kennenlernen.

1.5.1 Sprechen wir über Kosten

Im allgemeinen wird ein Problem am Computer über einen Algorithmus gelöst, also einer Vorschrift in Form eines endlichen Textes, die in eindeutiger Weise alle zur Lösung des Problems notwendigen Schritte präzisiert.

Unter dem *Rechenaufwand* (*computational costs*) eines Algorithmus verstehen wir normalerweise die Anzahl an arithmetischen Operationen, die für dessen Ausführung notwendig sind. Oft wird die Geschwindigkeit eines Computers über die maximale Anzahl von *Floating-Point-Operationen*, die er in einer Sekunde ausführen kann, in *Flops* und seinen Vielfachen (Megaflops gleich 10^6 *flops*, Gigaflops gleich 10^9 *Flops*, Teraflops gleich 10^{12} *Flops*) gemessen. Die zur Zeit schnellsten Computer können mehr als 136 Teraflops verarbeiten. In MATLAB war es bis Version 5.3 möglich, mit dem Befehl `flops` die ungefähre Zahl der von einem Programm durchgeführten *Floating-Point-Operationen* zu erfahren. Seit Version 6 ist dies nicht mehr möglich.

flops

Im allgemeinen muß man die Anzahl arithmetischer Operationen nicht kennen, es genügt, die Größenordnung in Funktion eines Parameters d , der an die Dimension des zu lösenden Problems gebunden ist, zu quantifizieren. So sagen wir, daß ein Algorithmus *konstante* Komplexität hat, falls er eine von d unabhängige, also $\mathcal{O}(1)$ Anzahl von Operationen erfordert, *lineare*, falls er $\mathcal{O}(d)$ Operationen und allgemein *polynomiale* Komplexität hat, falls er $\mathcal{O}(d^m)$ Operationen erfordert, mit positivem Ganzen m . Einige Algorithmen haben *exponentielle* ($\mathcal{O}(c^d)$ Operationen) oder *faktorielle* Komplexität ($\mathcal{O}(d!)$ Operationen).

Wir erinnern daran, daß das Symbol $\mathcal{O}(d^m)$ (das gelesen wird als „groß O von d^m “) für „verhält sich für große d wie eine Konstante mal d^m “ steht.

Beispiel 1.2 (das Matrix-Vektor Produkt) Sei A eine quadratische Matrix der Dimension n ; wir wollen den rechnerischen Aufwand eines gewöhnlichen Algorithmus zur Berechnung des Produkts $A\mathbf{v}$ mit $\mathbf{v} \in \mathbb{R}^n$ bestimmen. Die Berechnung der j -ten Komponente des Produktes ist gegeben durch

$$a_{j1}v_1 + a_{j2}v_2 + \dots + a_{jn}v_n$$

und erfordert die Bildung von n Produkten und $n-1$ Summen. Man benötigt also $n(2n-1)$ Operationen zur Berechnung aller n Komponenten. Dieser Algorithmus benötigt also $\mathcal{O}(n^2)$ Operationen und hat somit quadratische Komplexität bezüglich des Parameters n . Mit derselben Vorgangsweise sind $\mathcal{O}(n^3)$ Operationen zur Berechnung des Produkts zweier Matrizen der Ordnung n nötig. Es gibt allerdings einen Algorithmus, Algorithmus von Strassen genannt, der „nur“ $\mathcal{O}(n^{\log_2 7})$ Operationen benötigt, und einen anderen, der auf Winograd und Coppersmith zurückgeht und nur $\mathcal{O}(n^{2.376})$ Operationen benötigt.

Beispiel 1.3 (die Berechnung der Determinante) Wie wir bereits gesehen haben, kann die Determinante einer Matrix der Dimension n mit der Rekursionsformel (1.8) berechnet werden.

Man kann zeigen, daß der zugehörige Algorithmus faktorielle Komplexität bezüglich n hat, also $\mathcal{O}(n!)$ Operationen benötigt. Man bedenke, daß derart komplexe Algorithmen selbst auf den zur Zeit schnellsten Computern nur für kleine n ausgeführt werden können. Ist zum Beispiel $n = 24$, so würde ein Rechner, der ein Petaflop (also 10^{15} Operationen in der Sekunde) durchführen könnte, etwa 20 Jahre für diese Berechnung benötigen. Wir sehen also, daß die alleinige Steigerung der Rechenleistung noch nicht bedeutet, daß jedes Problem gelöst werden kann; effizientere numerische Verfahren müssen untersucht und angewandt werden. Zum Beispiel gibt es einen rekursiven Algorithmus, der die Berechnung der Determinante auf die Berechnung von Matrizenprodukten zurückführt, der nur mehr eine Komplexität von $\mathcal{O}(n^{\log_2 7})$ Operationen hat, falls man den eben erwähnten Algorithmus von Strassen anwendet (siehe [BB96]).

Die Anzahl der von einem Algorithmus benötigten Operationen ist also ein wichtiger Parameter bei dessen theoretischer Untersuchung. Sobald aber ein Algorithmus in einem Programm implementiert ist, können andere Faktoren mit eine Rolle spielen, die die Effektivität beeinflussen können (wie zum Beispiel der Zugriff auf den Speicher). Ein Maß für die Geschwindigkeit eines Programmes ist die Zeit, die für dessen Ausführung benötigt wird, also die *CPU-Zeit* (CPU steht für *central processing unit*). Es geht also um die Zeit, die vom Prozessor des Rechners benötigt wird, um ein bestimmtes Programm auszuführen, ohne die Wartezeit zu berücksichtigen, die für das Laden von Daten (die sogenannte *Input-Phase*) oder zum Speichern der erhaltenen Resultate (*Output-Phase*) vergeht. Die Zeit hingegen, die vom Beginn der Ausführung bis zum Ende des Programms vergeht, wird *Elapsed-Time* genannt. In MATLAB wird die CPU-Zeit mit dem Befehl `cputime` gemessen, die *Elapsed-Time* (stets in Sekunden) mit dem Befehl `etime`.

Beispiel 1.4 Messen wir die Ausführzeit (elapsed time) für die Berechnung des Produktes einer quadratischen Matrix mit einem Vektor. Dazu führen wir folgende Befehle aus:

```
>> n = 4000; step = 50; A = rand(n,n); v = rand(n); T = [ ];
sizeA = [ ]; count = 1;
>> for k = 50:step:n
    AA = A(1:k,1:k); vv = v(1:k)';
    t = cputime; b = AA*vv; tt = cputime - t;
    T = [T, tt]; sizeA = [sizeA,k]; count = count + 1;
end
```

Mit dem Befehl `a:step:b`, der in der `for`-Schleife auftritt, werden alle Zahlen der Form `a+step*k` mit ganzem `k` von 0 bis `kmax`, für die `a+step*kmax` kleiner gleich `b`, erzeugt (in unserem Fall `a=50`, `b=4000` und `step=50`). Der Befehl `rand(n,m)` initialisiert eine Matrix $n \times m$, deren Elemente Zufallszahlen sind. Schließlich werden in den Komponenten des Vektors `T` die CPU-Zeiten zur Ausführung eines jeden Matrix-Vektor-Produkts gespeichert. `cputime` gibt die von MATLAB zur Ausführung jedes einzelnen Prozesses benötigte Gesamtzeit zurück. Die zur Ausführung eines einzelnen Prozesses benötigte Zeit ist also die Differenz zwischen der aktuellen CPU-Zeit und jener vor Start des gerade betrachteten Prozesses und wird in unserem Fall in der Variablen `t` gespeichert. Der Graph in Abbildung 1.8 (erhalten mit dem Befehl `plot(sizeA,T,'o')`) zeigt, wie die CPU-Zeit etwa proportional zum Quadrat der Dimension `n` der Matrix steigt.

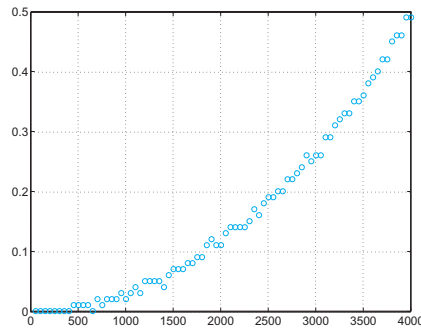


Abb. 1.8. Zur Ausführung eines Matrix-Vektor-Produkts benötigte CPU-Zeit (in Sekunden) in Funktion der Dimension n der Matrix auf einem PC mit 2.53 GHz

1.6 Einige Worte zu MATLAB



MATLAB ist eine Programmumgebung für Wissenschaftliches Rechnen und Visualisierung, geschrieben in der Programmiersprache C und vertrieben von The MathWorks (siehe www.mathworks.com). Der Name steht für *MATrix LABoratory*, denn ursprünglich wurde das Programm dazu entwickelt, sofort auf speziell für die Matrizenrechnung entwickelte Softwarepakete zugreifen zu können.

Ist MATLAB installiert (die ausführende Programmdatei befindet sich im allgemeinen im Unterverzeichnis *bin* des Hauptverzeichnisses *matlab*), hat man nach dessen Ausführung Zugang zu einer Arbeitsumgebung, die vom *Prompt* `>>` charakterisiert wird. Führen wir nun also MATLAB aus, erscheint

`>>`

< M A T L A B >
 Copyright 1984-2004 The MathWorks, Inc.
 Version 7.0.1.24704 (R14) Service Pack 1
 September 13, 2004

Using Toolbox Path Cache. Type "help toolbox_path_cache" for more info.

To get started, select "MATLAB Help" from the Help menu.

>>

Alles, was wir nach dem *Prompt* schreiben, wird nach Drücken der Eingabetaste *Enter* (oder *Return*) sofort von MATLAB interpretiert.¹

MATLAB überprüft nun, ob unsere Eingabe einer definierten Variablen oder einem MATLAB-Programm oder -Befehl entspricht. Scheitert diese Überprüfung, gibt MATLAB eine Fehlermeldung aus. Andernfalls wird der Befehl ausgeführt, der unter Umständen einen *Output* erzeugt. In beiden Fällen bietet uns das System am Ende wieder den *Prompt* an und wartet auf eine neue Eingabe. Das Programm wird mit dem Befehl **quit** (oder **exit**) und *Enter* beendet. Von nun an verstehen wir unter dem Ausführen eines Befehls immer auch dessen Bestätigung mit der *Enter*-Taste und verwenden die Ausdrücke Befehl, Funktion oder Programm in äquivalenter Weise.

Ein spezieller Fall ergibt sich, falls der eingegebene Befehl eine in MATLAB elementar definierte Struktur ist (etwa Zahlen oder durch einfache Anführungszeichen definierte Zeichenketten). Diese werden sofort erkannt und im *Output* in der *Default*-Variablen **ans** (für *Answer*) ausgegeben. Geben wir zum Beispiel die Zeichenkette 'Haus' ein, erhalten wir

```
>> 'Haus'
ans =
Haus
```

Geben wir jetzt eine andere Zeichenkette oder Zahl ein, nimmt **ans** diesen neuen Wert an.

Um diesen *Output* zu unterdrücken, genügt es, nach der Variablen oder dem Befehl einen Strichpunkt zu setzen. So erscheint nach der Ausführung von 'Haus'; lediglich der *Prompt*, der Wert der Variablen wird aber wieder der Variablen **ans** zugewiesen.

Im allgemeinen werden in MATLAB Variablen mit dem Befehl **=** zugewiesen. Wollen wir zum Beispiel die Zeichenkette 'Haus' der Variablen **a** zuweisen, schreiben wir einfach

¹ Folglich muß ein MATLAB-Programm nicht wie in anderen Programmiersprachen, etwa Fortran oder C, kompiliert werden, auch wenn MATLAB den Compiler **mcc** enthält, um Programme schneller ausführen zu können.

```
>> a='Haus';
```

Wie wir sehen, muß der Typ der Variablen `a` im Gegensatz zu anderen Programmiersprachen nicht deklariert werden, MATLAB erledigt das für uns automatisch und dynamisch. Dadurch können derselben Variablen nach der Reihe Werte verschiedenen Typs zugewiesen werden. Wollen wir etwa der vorhin initialisierten Variablen `a` jetzt die Zahl 5 zuweisen, müssen wir nichts anderes schreiben als `a=5`. Diese extrem einfache Bedienbarkeit hat aber ihren Preis. Wir wollen zum Beispiel eine Variable `quit` nennen und sie gleich 5 setzen. Wir haben also eine Variable erzeugt, die genauso heißt wie der MATLAB-Befehl `quit`; dadurch können wir den Befehl `quit` nicht mehr ausführen, da MATLAB für dessen Interpretation zunächst überprüft, ob es sich dabei um eine Variable handelt, und erst dann, ob es sich um einen vordefinierten Befehl handelt. Wir müssen es also unbedingt vermeiden, Variablen oder Programmen bereits in MATLAB *vordefinierte* Variablen oder Programme zuzuweisen. Um wieder auf den Befehl `quit` zugreifen zu können, kann man mit `clear`, gefolgt vom Namen der Variablen (in diesem Fall `quit`), die aktuelle Definition der Variablen wieder löschen.

`clear`

Mit dem Befehl `save` werden alle Variablen der aktuellen Sitzung (der sogenannte *Base-Workspace*) in der binären Datei `matlab.mat` gespeichert. Analog dazu können wir mit dem Befehl `load` alle in der binären Datei `matlab.mat` gespeicherten Variablen wiederherstellen. Der Dateiname, unter dem wir die Variablen speichern (aus dem wir sie laden), kann auch vom Benutzer definiert werden, indem man nach dem Befehl `save` (`load`) den Dateinamen bestimmt. Wollen wir schließlich nur einige Variablen, etwa `v1`, `v2` und `v3` in eine Datei mit Namen `area.mat` speichern, genügt der Befehl `save area v1 v2 v3`.

`save`

`load`

Mit dem Befehl `help` können wir alle verfügbaren Befehle und vordefinierten Variablen, auch die der sogenannten *Toolboxes*, Sammlungen speziellerer Befehle, ansehen. Unter diesen wollen wir an die elementaren Funktionen Sinus (`sin(a)`), Cosinus (`cos(a)`), Quadratwurzel (`sqrt(a)`) sowie die Exponentialfunktion (`exp(a)`) erinnern.

`help`

`sin cos`
`sqrt exp`

Es gibt außerdem spezielle Zeichen, die nicht im Namen einer Variablen oder eines Befehls vorkommen dürfen, etwa die elementaren Operatoren für Addition, Subtraktion, Multiplikation und Division (`+`, `-`, `*` und `/`), die logischen Operatoren *and* (`&`), *or* (`|`), *not* (`~`), die Verhältnisoperatoren größer (`>`), größer oder gleich (`>=`), kleiner (`<`), kleiner oder gleich (`<=`), gleich (`==`). Schließlich darf ein Name mit keiner Zahl beginnen und keine Klammern oder Interpunktionszeichen enthalten.

`+ - * /`
`& | ~ >`
`>= < <= ==`

1.6.1 MATLAB-Statements

MATLAB enthält auch eine spezielle Programmiersprache, mit der der Benutzer neue Programme schreiben kann. Auch wenn wir diese nicht

beherrschen müssen, um die in diesem Buch vorhandenen Programme auszuführen, so erlaubt sie uns doch, diese zu verändern oder neue zu erstellen. Die Programmiersprache MATLAB besitzt verschiedene Befehle und Konstrukte (*Statements*), hier wollen wir auf jene für *Abfragen* und *Schleifen* eingehen.

Das *Statement* für eine *if-elseif-else*-Abfrage hat folgende allgemeine Form

```
if cond(1)
    statement(1)
elseif cond(2)
    statement(2)
.
.
.
else
    statement(n)
end
```

wobei `cond(1)`, `cond(2)`, ... MATLAB-Befehle sind, die die Werte 0 oder 1 (falsch oder wahr) annehmen können. Die gesamte Konstruktion erlaubt die Ausführung jenes *Statements*, das zur ersten Bedingung mit dem Wert 1 gehört. Falls alle Bedingungen falsch sind, wird das letzte *Statement*, `statement(n)`, ausgeführt. Falls also Bedingung `cond(k)` falsch ist, wird `statement(k)` nicht ausgeführt und übersprungen.

Um zum Beispiel die Wurzeln des Polynoms $ax^2 + bx + c$ zu berechnen, können wir folgende Befehle verwenden (der Befehl `disp()` zeigt lediglich an, was zwischen den Klammern steht):

```
>> if a~=0
    sq = sqrt(b*b - 4*a*c);
    x(1) = 0.5 * (-b + sq)/a;
    x(2) = 0.5 * (-b - sq)/a;
elseif b~=0
    x(1) = -c/b;
elseif c~=0
    disp('Unmögliche Gleichung');
else
    disp('Die eingegebene Gleichung ist eine Identität')
end
```

(1.12)

Beachte, daß MATLAB die Gesamtkonstruktion erst ausführt, sobald sie mit dem *Statement* `end` abgeschlossen wird.

In MATLAB stehen uns zwei Arten von Schleifen zur Verfügung: `for` (ähnlich dem `do` in Fortran oder dem `for` in C) und `while`. Eine `for`-Schleife wiederholt eine Anweisung für alle in einem Zeilenvektor

enthalten Indizes. Um etwa die ersten sechs Elemente einer Fibonacci-Folge $\{f_i = f_{i-1} + f_{i-2}, i \geq 2\}$ mit $f_1 = 0$ und $f_2 = 1$ zu berechnen, kann man folgende Befehle verwenden

```
>> f(1) = 0; f(2) = 1;
>> for i = [3 4 5 6]
    f(i) = f(i-1) + f(i-2);
end
```

Der Strichpunkt wird auch dazu verwendet, mehrere MATLAB-Befehle in einer Zeile voneinander zu trennen. Beachte, daß die **for**-Anweisung auch durch die äquivalente Anweisung `>> for i = 3:6` ersetzt werden kann. Die **while**-Schleife hingegen wird solange ausgeführt, bis eine logische Anweisung erfüllt ist. So sind zum Beispiel auch folgende Anweisungen für obiges Problem möglich

```
>> f(1) = 0; f(2) = 1; k = 3;
>> while k <= 6
    f(k) = f(k-1) + f(k-2); k = k + 1;
end
```

Andere, weniger gebräuchliche *Statements* sind **switch**, **case** und **otherwise**: für deren Erklärung verweisen wir den Leser auf das entsprechende *help*.

1.6.2 Programmieren in MATLAB

In diesem Abschnitt wollen wir kurz erklären, wie man MATLAB-Programme schreibt. Ein neues Programm muß in einer Datei gespeichert werden, die *m-File* genannt wird und die Endung **.m** trägt. Diese Datei kann dann in MATLAB aufgerufen werden, muß sich aber in einem der Ordner (*Directories*) befinden, in denen MATLAB automatisch nach m-Files sucht. Eine Liste dieser Ordner einschließlich des aktuellen Ordners kann mit dem Befehl **path** ausgegeben werden. Wie man dieser Liste einen Ordner hinzufügt, steht im entsprechenden *Help*. path

Bei Programmen ist es wichtig, zwischen *Scripts* und *Functions* zu unterscheiden. Erstere sind einfache Sammlungen von MATLAB-Befehlen ohne Input/Output-Schnittstelle. Zum Beispiel werden die Befehle (1.12) zusammen in einem m-File, etwa mit Namen **equation.m**, abgespeichert, zu einem *Script*. Um dieses auszuführen, genügt es, nach dem Prompt `>>` einfach den Befehl **equation** einzugeben. Im folgenden geben wir zwei Anwendungsbeispiele:

```
>> a = 1; b = 1; c = 1;
>> equation
ans =
-0.5000 + 0.8660i -0.5000 - 0.8660i
```

```
>> a = 0; b = 1; c = 1;
>> equation
ans =
    -1
```

Da wir keine Input/Output-Schnittstelle haben, sind alle im *Script* verwendeten Unbekannten Teil der aktuellen Arbeitssitzung und werden daher auch erst nach der expliziten Anweisung **clear** gelöscht. Diese Eigenschaft ist nicht sehr befriedigend, sobald man komplizierte Programme mit vielen temporären Variablen und wenigen Input- und Output-Variablen erstellen möchte. Nach Programmende sind die Output-Variablen auch die einzigen, die man noch behalten oder abspeichern möchte. Aus diesem Grund greift man auf eine viel flexiblere Programmform zurück, auf die sogenannte *Function*. Eine *Function* ist wieder in einem m-File definiert, zum Beispiel mit Namen **name.m**, besitzt aber eine genau festgelegte Input-/Output-Schnittstelle, eingeführt mit dem Befehl **function**

```
function [out1,...,outn]=name(in1,...,inm)
```

wobei out_1, \dots, out_n die Output-Variablen und in_1, \dots, in_m die Input-Variablen sind.

Die folgende Datei **det23.m** ist ein Beispiel für eine *Function*; in ihr wird eine neue *Function*, nämlich **det23**, definiert, die die Determinante einer Matrix der Dimension 2 oder 3 nach der Formel aus Abschnitt 1.3 berechnet:

```
function [det]=det23(A)
%DET23 berechnet die Determinante einer quadratischen Matrix
% der Dimension 2 oder 3
[n,m]=size(A); if n==m
    if n==2
        det = A(1,1)*A(2,2)-A(2,1)*A(1,2);
    elseif n == 3
        det = A(1,1)*det23(A[2,3],[2,3]))-A(1,2)*det23(A([2,3],[1,3]))+...
            A(1,3)*det23(A[2,3],[1,2]));
    else
        disp(' Nur (2x2)- oder (3x3)-Matrizen!');
    end
else
    disp(' Nur quadratische Matrizen! ');
end
return
```

... Die Zeichen ... bedeuten, daß eine Anweisung in der nächsten Zeile fortgesetzt wird, das Zeichen % kommentiert eine Zeile aus. Die Anweisung $A([i,j],[k,1])$ erlaubt die Konstruktion einer (2×2) -Matrix,

deren Elemente im Schnitt der i -ten und j -ten Zeilen mit den k -ten und l -ten Spalten der Ausgangsmatrix A liegen.

Sobald man eine *Function* aufruft, erzeugt MATLAB einen lokalen Arbeitsbereich (den *Function's Workspace*), in dem alle innerhalb der *Function* aufgerufenen Variablen angelegt werden. Folglich können sich die in einer *Function* enthaltenen Anweisungen nicht auf eine im *Base-Workspace* deklarierte Variable beziehen, außer sie werden der Funktion als Input übergeben.² Alle in einer *Function* verwendeten Variablen sind verloren, sobald die Funktion beendet ist, außer sie gehören zu den Output-Parametern.

Normalerweise ist eine *Function* beendet, sobald die letzte Anweisung abgearbeitet wurde. Mit einem **return Statement** kann man die Funktion schon früher verlassen. Um etwa den Wert $\alpha = 1.6180339887\dots$, den Grenzwert für $k \rightarrow \infty$ des Verhältnisses f_k/f_{k-1} von Fibonacci-Folgen zu approximieren, und zwar indem wir solange iterieren, bis sich zwei aufeinanderfolgende Brüche um weniger als 10^{-4} unterscheiden, können wir folgende *Function* konstruieren

return

```
function [golden,k]=fibonacci
f(1) = 0; f(2) = 1; goldenold = 0; kmax = 100; tol = 1.e-04;
for k = 3:kmax
    f(k) = f(k-1) + f(k-2);
    golden = f(k)/f(k-1);
    if abs(golden - goldenold) <= tol
        return
    end
    goldenold = golden;
end
return
```

Das Programm wird entweder nach **kmax=100** Iterationen abgebrochen, oder nachdem sich der Absolutbetrag der Differenz zweier aufeinanderfolgender Iterierter um weniger als **tol=1.e-04** unterscheidet. Wir können die *Function* folgendermaßen ausführen

```
>> [alpha,niter]=fibonacci
alpha =
    1.61805555555556
niter =
    14
```

Nach 14 Iterationen gibt die *Function* also einen Wert zurück, der mit dem exakten α die fünf ersten signifikanten Stellen gemeinsam hat.

² Es gibt noch eine dritte Art von *Workspace*, den *Global Workspace*, in dem alle **global** definierten Variablen gespeichert werden. Diese Variablen können auch dann in einer *Function* verwendet werden, wenn sie nicht Input-Parameter sind.

Die Anzahl der Ein- und Ausgabeparameter einer MATLAB-*Function* ist beliebig. Die *Function* `fibonacci` können wir zum Beispiel folgendermaßen abändern

```
function [golden,k]=fibonacci(tol,kmax)
if nargin == 0
    kmax = 100; tol = 1.e-04; % default Werte
elseif nargin == 1
    kmax = 100; % default Wert fuer kmax
end
f(1) = 0; f(2) = 1; goldenold = 0;
for k = 3:kmax
    f(k) = f(k-1) + f(k-2);
    golden = f(k)/f(k-1);
    if abs(golden - goldenold) <= tol
        return
    end
    goldenold = golden;
end
return
```

`nargin`
`nargout`

Die *Function* `nargin` zählt die Input-Parameter (`nargout` die Output-Parameter). In dieser Version der *Function* `fibonacci` können wir entweder die maximale Anzahl erlaubter Iterationen (`kmax`) und eine spezielle Toleranz `tol` vorschreiben, oder die in der *Function* festgelegten *Default*-Werte (`kmax=100` und `tol=1.e-04`) beibehalten, indem wir keine Input-Parameter übergeben. So können wir die *Function* folgendermaßen verwenden

```
>> [alpha,niter]=fibonacci(1.e-6,200)
alpha =
    1.61803381340013
niter =
    19
```

Durch die Wahl einer kleineren Toleranz für das Abbruchkriterium haben wir eine neue Approximation berechnet, die mit dem exakten α gar acht signifikante Ziffern gemeinsam hat. Die *Function* `nargin` kann auch außerhalb einer *Function* verwendet werden, um die maximale Anzahl von Input-Parametern einer *Function* abzufragen. Hier ein Beispiel:

```
>> nargin('fibonacci')
ans =
    2
```

inline **Bemerkung 1.2 (inline functions)** Der Befehl `inline`, dessen einfachste Syntax `g=inline(expr,arg1,arg2,...,argn)` ist, deklariert eine *Function* `g`, die von den Zeichenketten `arg1,arg2,...,argn` abhängt. Die Zeichenkette

`expr` enthält den Ausdruck von `g`. So deklariert `g=inline('sin(r)','r')` die Funktion $g(r) = \sin(r)$. In der abgekürzten Schreibweise `g=inline(expr)` wird implizit angenommen, daß der Ausdruck `expr` nur von der Variablen `x` abhängt. Ist eine *Inline-Function* einmal deklariert, so läßt sie sich auf jeder beliebigen Menge von Variablen mittels des Befehls `feval` auswerten. Um zum Beispiel `g` in den Punkten `z=[0 1]` auszuwerten, können wir schreiben.

```
>> feval('g',z);
```

Wie wir sehen, muß im Gegensatz zum Befehl `eval` mit `feval` der Variablenname (`z`) nicht notwendigerweise mit dem mit `inline` zugewiesenen symbolischen Namen (`r`) übereinstimmen.

Nach dieser kurzen Einleitung möchten wir dazu einladen, MATLAB mit Unterstützung des *Help* weiter zu erkunden. So erhält man etwa mit dem Befehl `help for` nicht nur eine ausführliche Beschreibung dieser Anweisung, am Ende wird auch auf ähnliche Befehle (im Fall von `for` auf `if`, `while`, `switch`, `break`, `end`) verwiesen. Mit *Help* können wir also fortwährend unsere MATLAB-Kenntnisse erweitern.

Siehe Aufgaben 1.8–1.11.



1.7 Was wir nicht erwähnt haben

Eine systematischere Behandlung der *Floating-Point*-Zahlen findet sich in [QSS02] oder in [Ueb97].

Für die Behandlung der rechnerischen Komplexität und von Algorithmen im allgemeinen verweisen wir zum Beispiel auf [Pan92].

Für eine systematische Einführung in MATLAB verweisen wir den interessierten Leser auf das Handbuch von MATLAB [HH00] oder [Mat04], aber auch auf Spezialliteratur wie etwa [HLR01] oder [EKH02].

1.8 Aufgaben

Aufgabe 1.1 Aus wie vielen Zahlen besteht die Menge $\mathbb{F}(2, 2, -2, 2)$? Wie groß ist ϵ_M für diese Menge?

Aufgabe 1.2 Zeige, daß die Menge $\mathbb{F}(\beta, t, L, U)$ genau $2(\beta-1)\beta^{t-1}(U-L+1)$ Zahlen enthält.

Aufgabe 1.3 Konstruiere in MATLAB eine obere und eine untere Dreiecksmatrix der Dimension 10 mit Einträgen 2 auf der Hauptdiagonalen und -3 auf der zweiten oberen und unteren Nebendiagonalen.

Aufgabe 1.4 Welche MATLAB-Anweisungen sind notwendig, um die dritte und siebte Zeile bzw. die vierte und achte Spalte der in Aufgabe 1.3 konstruierten Matrix zu vertauschen?

Aufgabe 1.5 Zeige, daß folgende Vektoren in \mathbb{R}^4 linear unabhängig sind:

$$\mathbf{v}_1 = [0 \ 1 \ 0 \ 1], \mathbf{v}_2 = [1 \ 2 \ 3 \ 4], \mathbf{v}_3 = [1 \ 0 \ 1 \ 0], \mathbf{v}_4 = [0 \ 0 \ 1 \ 1].$$

Aufgabe 1.6 Gib folgende Funktionen in MATLAB ein und berechne mit der Toolbox für symbolisches Rechnen deren erste und zweite Ableitungen und das unbestimmte Integral:

$$g(x) = \sqrt{x^2 + 1}, f(x) = \sin(x^3) + \cosh(x).$$

Aufgabe 1.7 Sei ein Vektor \mathbf{v} der Dimension n gegeben. Mit der Anweisung `poly` `c=poly(v)` können wir die $n + 1$ Koeffizienten eines Polynoms vom Grad n berechnen, dessen Koeffizient x^n gleich 1 ist und dessen Nullstellen genau die in \mathbf{v} gespeicherten Werte haben. Man wird erwarten, daß `v = roots(poly(c))` ist. Versuche, `roots(poly([1:n]))` zu berechnen, wobei n von 2 bis 25 geht, und kommentiere die erhaltenen Ergebnisse.

Aufgabe 1.8 Schreibe ein Programm, das folgende Folge berechnet

$$I_0 = \frac{1}{e}(e - 1),$$

$$I_{n+1} = 1 - (n + 1)I_n, \text{ für } n = 0, 1, \dots, 21.$$

Da wir wissen, daß $I_n \rightarrow 0$ für $n \rightarrow \infty$, kommentiere die erhaltenen Ergebnisse.

Aufgabe 1.9 Erkläre das Verhalten der in MATLAB berechneten Folge (1.4).

Aufgabe 1.10 Betrachten wir folgenden Algorithmus zur Berechnung von π : erzeuge n Paare (x_k, y_k) von Zufallszahlen zwischen 0 und 1. Von diesen berechne die Anzahl m jener Punkte, die ins erste Viertel des Kreises mit Mittelpunkt im Ursprung und Radius 1 fallen. Offensichtlich ist π der Grenzwert der Folge $\pi_n = 4m/n$. Schreibe ein Programm, das diese Folge berechnet und überprüfe den Fehler für steigende n .

Aufgabe 1.11 Schreibe ein Programm, das den Binomialkoeffizienten $\binom{n}{k} = n!/(k!(n-k)!)$ berechnet, wobei n und k natürliche Zahlen mit $k \leq n$ sind.



<http://www.springer.com/978-3-540-25005-0>

Wissenschaftliches Rechnen mit MATLAB

Quarteroni, A.; Saleri, F.

2006, X, 269 S., Softcover

ISBN: 978-3-540-25005-0