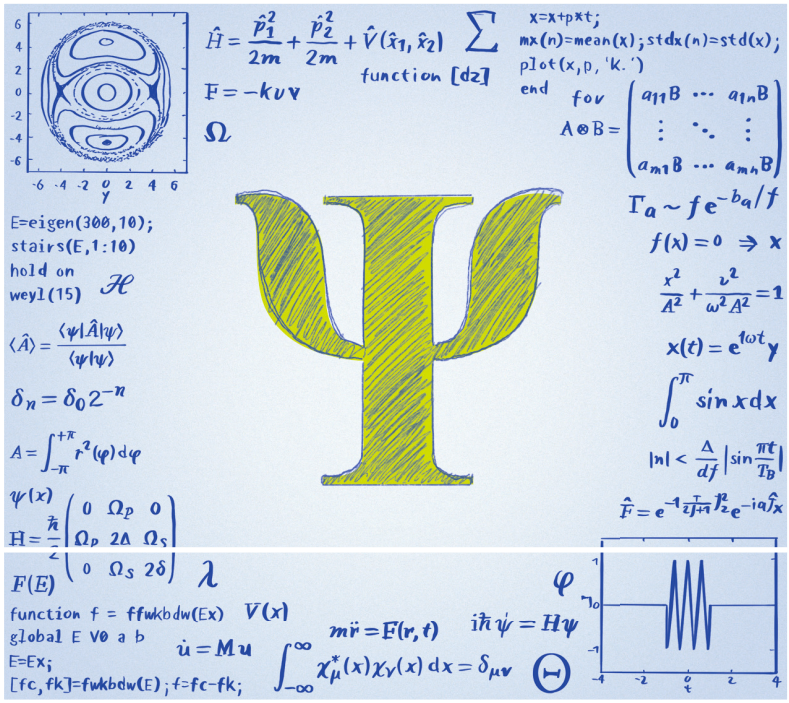
Hans Jürgen Korsch

Numerische Physik mit Octave und Matlab

Klassische Mechanik und Quantenmechanik



HANSER

Korsch

Numerische Physik mit Octave und Matlab

Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial. Geben Sie dazu einfach diesen Code ein:

plus-x4g55-wtkbm

plus.hanser-fachbuch.de

Bleiben Sie auf dem Laufenden!

Hanser Newsletter informieren Sie regelmäßig über neue Bücher und Termine aus den verschiedenen Bereichen der Technik. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter

www.hanser-fachbuch.de/newsletter

Hans Jürgen Korsch

Numerische Physik mit Octave und Matlab

Klassische Mechanik und Quantenmechanik

HANSER

Der Autor:

Prof. Dr. Hans Jürgen Korsch
Technische Universität Kaiserslautern
Fachbereich Physik

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de abrufbar.

Internet: www.hanser-fachbuch.de

Lektorat: Frank Katzenmayer
Herstellung: Frauke Schafft
Covergestaltung: Max Kostopoulos
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Titelbild: © Max Kostopoulos
Satz: Hans Jürgen Korsch
Druck und Bindung: CPI books GmbH, Leck
Printed in Germany

Print-ISBN 978-3-446-47026-2
E-Book-ISBN 978-3-446-47316-4

Vorwort

Numerische Methoden sind heute in der Physik weit verbreitet und nicht nur unter dem Schlagwort „Computational Physics“ unterzubringen. In vielen Lehrveranstaltungen haben sie inzwischen neben den herkömmlichen analytischen Methoden ihren Platz gefunden. Das lässt sich hauptsächlich auf den extremen Leistungszuwachs der PCs zurückführen, mit denen (fast) jeder in der Lage ist, auf seinem Schreibtisch schnelle numerische Berechnungen durchzuführen, die noch vor wenigen Jahrzehnten den Hochleistungsrechnern vorbehalten waren. Diese Entwicklung beruht aber auch auf der Verfügbarkeit moderner Programmierumgebungen, mit denen solche Rechnungen bequem und effizient durchzuführen sind, und deren grafische Darstellungsmöglichkeiten auch noch Vergnügen bereiten.

Hier ist in erster Linie das Softwarepaket **MATLAB** zu nennen, „eine kommerzielle Software des US-amerikanischen Unternehmens MathWorks zur Lösung mathematischer Probleme und zur grafischen Darstellung der Ergebnisse. Matlab ist vor allem für numerische Berechnungen mithilfe von Matrizen ausgelegt, woher sich auch der Name ableitet: MATrix LABoratory.“ Diesen Text und mehr dazu findet man bei https://de.wikipedia.org/wiki/Matlab.

Daneben existieren kostenlose freie Alternativen, die MATLAB als Programmiersprache verwenden und die Funktionalität von MATLAB nachbilden. Sie sind zum großen Teil codekompatibel zu MATLAB. Neben dem Softwarepaket **FreeMat** ist hier insbesondere **GNU Octave** zu nennen (mehr dazu findet man unter https://de.wikipedia.org/wiki/GNU_Octave). Alle Programme dieses Buches sind unter Octave entwickelt worden (Octave Version 5.1.0).

Warum MATLAB? Dieses Softwarepaket bietet viele Vorteile:

- Die Programmiersprache ist intuitiv und leicht zu erlernen.
- Kleine Programme können sehr schnell entwickelt, implementiert und benutzt werden.
- Viele mathematische Strukturen und Methoden sind vorhanden und lassen sich direkt benutzen.
- Eine leistungsfähige grafische Ausgabe ermöglicht vielfältige Illustrationen.
- MATLAB ist als Standard-Software in Wissenschaft und Industrie weit verbreitet und an den meisten Universitäten verfügbar.

Dieses Buch behandelt ausgewählte Themen der Klassischen Mechanik und der Quantenmechanik mit numerischen Methoden. Es ist jedoch *kein* Lehrbuch …

- der Programmiersprache MATLAB, jedoch wird als Hilfestellung für Einsteiger eine kurze Einführung in MATLAB im Kapitel 1 vorangestellt.
- der Numerischen Mathematik oder Physik. Elementare numerische Methoden, die in den folgenden Anwendungen verwendet werden, sind im Kapitel 2 zu finden.
- der Theoretischen Physik, hier also der Klassischen Mechanik und der Quantenmechanik, dazu fehlt es an dem notwendigen systematischen Aufbau, den mathematischen Herleitungen und insbesondere der Vollständigkeit.

Ziel dieses Buches ist es vielmehr, wichtige Aspekte der Quantentheorie vorzustellen, die in nahezu jedem Kurs in Quantenmechanik vorkommen oder vorkommen könnten, und durch numerische Berechnungen und grafische Darstellungen in sinnvoller Weise zu ergänzen. Dazu werden kleine Programme vorgestellt, die das leisten können, ergänzt durch viele Aufgaben und Anregungen zum eigenen Experimentieren, sei es durch Erweiterungen der vorliegenden Programme oder durch eigene Neuentwicklungen. Auf diese Weise lernt man einerseits wichtige numerische Techniken kennen und ihre Umsetzung in ein Programm, und andererseits erweitert man seine MATLAB-Kenntnisse. Das ist aber nur ein Nebeneffekt, denn es ist die Überzeugung des Autors, dass man so ein besseres Verständnis der betrachteten physikalischen Systeme und ihrer theoretischen Modellierung gewinnt.

Der Aufbau des Buches und die Auswahl der hier behandelten Systeme ist sehr subjektiv gefärbt und beruht auf den Vorlieben des Autors sowohl in der Forschung als auch in der Lehre. Insbesondere wird ein Brückenschlag zwischen klassischer Mechanik und Quantenmechanik versucht, indem Unterschiede und Ähnlichkeiten herausgearbeitet werden. Die Themenwahl spannt einen weiten Bogen von der klassischen Einteilchendynamik bis hin zu quantenmechanischen Vielteilchenproblemem, wie zum Beispiel Bose-Einstein-Kondensaten. Behandelt werden neben den bekannten regulären Systemen auch solche mit (nichtlinearer) klassisch chaotischer Dynamik und ihre Signaturen in der (linearen) Quantenmechanik, also Fragen des Quantenchaos.

Wir beginnen im Kapitel 3 mit einfachsten Systemen der Klassischen Mechanik, wie der schräge Wurf sowie harmonische und anharmonische Schwingungen. Komplizierter wird die chaotische Dynamik in Kapitel 4. Nach diesem einführenden Training unserer numerischen Fertigkeiten beginnt endlich das Hauptthema, die Quantenmechanik. Die Kapitel 5 und 6 widmen sich elementaren ein- und mehrdimensionalen Quantensystemen, erforschen deren Energiespektren sowie die Eigenfunktionen in Orts-, Impuls- und Phasenraumdarstellungen. Die quantenmechanische Zeitentwicklung ist Thema der folgenden Kapitel 7 bis 10, wobei unter anderem Bloch-Oszillationen, Fragen des Quantenchaos und zerfallende Systeme behandelt werden. Zwei Anhänge beschreiben detailliert mathematische Methoden der Fourier-Transformation und des Kronecker-Produkts.

Die Aufgaben im Text sind einfach gehalten und sollen dazu anregen, mit den vorgegebenen Programmen zu arbeiten, sie weiter auszubauen oder neue Programme zu erstellen. Dies trägt einerseits dazu bei, die Programmiertechniken auszubauen, führt aber andererseits zu einem besseren Verständnis des betrachteten physikalischen Systems. In allen Fällen findet man ausführliche Lösungen am Ende jedes Kapitels.

Der vorliegende Text beruht auf den Vorlesungen des Autors zu Themen der Numerischen Physik und der Theoretischen Physik an der TU Kaiserslautern. Der Autor dankt den ehemaligen Mitgliedern seiner Arbeitsgruppe für viele Anregungen und Kommentare, insbesondere Dr. Friederike Trimborn-Witthaut und Dr. Dirk Witthaut, mit denen gemeinsam erste Versionen dieses Textes entwickelt wurde, und die viele Programmideen beigetragen haben. Sicher haben auch viele Leserinnen und Leser noch Verbesserungs- und Ergänzungsvorschläge, die sie bitte an

h.j.korsch@gmail.com

senden können. Eine aktuelle Korrekturliste und weitere Informationen findet man unter

https://plus.hanser-fachbuch.de/ .

Mein Dank gilt auch dem Carl Hanser Verlag für die Bereitschaft, dieses Buch in sein Verlagsprogramm aufzunehmen, und seinem Lektorat. Dabei haben mich wieder einmal Frau Christina Kubiak und Herr Frank Katzenmayer mit ihrer kompetenten Betreuung und vielen Verbesserungsvorschlägen unterstützt.

Kaiserslautern, November 2021 Hans Jürgen Korsch

Die Programme

In dem gesamten Buch finden sich viele MATLAB-Programme, die im Text gelistet sind. In den meisten Fällen sind diese m-Files sehr kurz und (hoffentlich) leicht verständlich. Hier als Beispiel das Programm **eigendw.m** , das die quantenmechanischen Eigenwerte eines Doppeltopfpotentials berechnet (mehr dazu auf Seite 151):

```
% eigendw.m - Eigenwerte Doppeltopfpotential
N = 100; n=1:N-1; V0 = 9;
m = sqrt(n);
x = 1/sqrt(2)*(diag(m,-1)+diag(m,1));
p = i/sqrt(2)*(diag(m,-1)-diag(m,1));
H = p^2/2+x^2/2+V0*expm(-x^2);
E = sort(eig(H)); E(1:12)
```

Um ein Programm wie dieses selbst auszuführen, was eigentlich unabdingbar ist, kann man natürlich den kurzen Text abtippen oder, falls man mit dem E-Book arbeitet, einfach per Drag and Drop mit der Maus übertragen. Es gibt aber noch eine bequeme Alternative:

Die Listings der 156 MATLAB-Programme stehen auf der Webseite des Hanser Verlags unter https://plus.hanser-fachbuch.de/ zur Verfügung. Den Zugangscode finden Sie auf der ersten Seite des Buches.

Dort findet man auch die acht Programme `bloch1av.m`, `blochflip.m`, `blochbz.m`, `eigfunp.m`, `honher2.m`, `honher3.m`, `honher4.m` und `holindV.m`, deren Aufbau zwar im Text erklärt wird, die dort jedoch nicht gelistet sind.

Literatur

Auf die oft üblichen detaillierten Angaben der Quellenliteratur zu den einzelnen Themen wird in dem vorliegenden Buch weitgehend verzichtet, abgesehen von einigen Hinweisen zur Originalliteratur in Fußnoten. Die folgenden Lehrbücher können jedoch dieses Buch ergänzen und helfen, Lücken zu füllen und zusätzliche Kenntnisse zu vermitteln.

MATLAB: Die Literatur zum Programmieren mit MATLAB ist sehr umfangreich, was angesichts der vielfältigen Möglichkeiten dieses Softwarepakets nicht verwunderlich ist. In diesem Buch

wird in Kapitel 1 nur eine sehr kurze Einführung gegeben. Für weitgehendere Information zum Programmieren in MATLAB sind die folgenden Bücher empfohlen:

- A. Bosl: *Einführung in MATLAB/Simulink*, Carl Hanser Verlag, 2020
- W. Schweizer: *MATLAB® kompakt*, Oldenbourg Verlag, 2008
- U. Stein: *Programmieren mit MATLAB*, Carl Hanser Verlag, 2017

Numerische Mathematik: Das vorliegende Buch ist, wie schon erwähnt, keine Einführung in die numerischen Methoden der Mathematik, obwohl es sinnvoll erschien, einige der grundlegenden Techniken im Kapitel 2 vorzustellen, insbesondere solche, die in den folgenden Programmen angewandt werden. Mehr Details findet man in den folgenden Büchern:

- G. Gramlich, W. Werner: *Numerische Mathematik mit Matlab*, dpunkt.verlag, 2000
- M. Knorrenschild: *Numerische Mathematik*, 2. Auflage, Carl Hanser Verlag, 2021
- W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical Recipes*, Cambridge University Press, London, 2007

Computer-unterstützte Physik: In diesem Buch wird versucht, grundlegende Phänomene der Physik mithilfe numerischer Experimente und grafischer Darstellungen zu illustrieren und verständlicher zu machen. Das ist natürlich kein Alleinstellungsmerkmal und wird in einer ganzen Reihe von Büchern mit verwandter Thematik in ähnlicher Weise verfolgt. Hier eine Auswahl davon:

- P. L. DeVries: *Computerphysik*, Spektrum Akadem. Verlag, Heidelberg, 1994
- P. Hertel: *Mathematikbuch zur Physik*, Springer-Verlag, 2009
- H. J. Korsch, H. J. Jodl, T. Hartmann: *Chaos - A Program collection for the PC*, Springer-Verlag, 2008
- W. Schweizer: *Simulation physikalischer Systeme*, De Gruyter, 2017
- U. Wolff: *Computational Physics I und II*, Vorlesungsskript, Humboldt-Universität zu Berlin, 2012

Quantenmechanik: Obwohl der Hauptteil des vorliegenden Buches Themen aus der Quantenmechanik gewidmet ist, soll hier nicht einmal ansatzweise versucht werden, grundlegende Lehrbücher der Quantenmechanik anzuführen, und die Leserinnen und Leser sollten ihre eigenen Favoriten heranziehen. Es lässt sich allerdings nicht leugnen, dass einige der angesprochenen Themen engen Bezug zu Texten des Autors zu diesem Themenbereich haben:

- H. J. Korsch: *Mathematik der Quantenmechanik*, Carl Hanser Verlag 2019
- H. J. Korsch: *Physik mit* 2×2*-Matrizen*, Carl Hanser Verlag 2020
- H. J. Korsch: *Mathematik mit* 2×2*-Matrizen*, Carl Hanser Verlag, 2020

Inhalt

1 MATLAB – eine kurze Einführung

MATLAB® und auch Octave oder FreeMat sind umfangreiche Programmpakete, die vielerlei Möglichkeiten bieten. Eine enorme Vielfalt an Strukturen und Programmen für die Entwicklung numerischer Simulationen werden bereitgestellt. Dieses Kapitel kann nur kurz die grundlegenden Konzepte und Techniken darstellen. Ausführliche Einführungen geben die Bücher *Einführung in MATLAB/Simulink* von Angelika Bosl oder *Programmieren mit MATLAB* von Ulrich Stein, und eine kompakte und vollständige Übersicht über MATLAB findet man in dem Buch *MATLAB® kompakt* von Wolfgang Schweizer (siehe Literaturverzeichnis auf Seite 7).

1.1 MATLAB als Taschenrechner

Zunächst wollen wir die grundlegenden Funktionen kennenlernen und benutzen MATLAB als einen besseren Taschenrechner. An der Eingabeaufforderung im Kommandofenster, die in MATLAB durch >> gekennzeichnet ist, kann man Befehle oder mathematische Ausdrücke eingeben. Bei einem Druck auf die Eingabetaste werden diese ausgeführt bzw. ausgewertet:

```
>> sin(pi/4)^2 * 84
ans =  42.000
```

Neben den Grundrechenarten beherrschen die Programmpakete eine Vielzahl von elementaren oder höheren mathematischen Funktionen wie den Sinus im obigen Beispiel, den Hyperbolischen Sinus `sinh`, den Inversen Sinus `asin`, die Quadratwurzel `sqrt`, die Exponentialfunktion `exp` oder die Gammafunktion `gamma`. Eine Übersicht über diese „built-in functions“ findet man beispielsweise in Octave, wenn man in der Menüleiste den Menüpunkt *Hilfe* → *Dokumentation* → *Funktions-Index* wählt, oder im Web unter

https://de.mathworks.com/help/matlab/elementary-math.html

Das Ergebnis einer Berechnung wie oben kann man nun einer Variablen zuweisen, mit der man dann weiter rechnen kann:

```
>> a = sin(pi/4)^2
a = 0.5000
>> b = 84
b = 84
>> a*b
ans = 42.0000
```

Hier wird das Ergebnis jeweils auf dem Bildschirm ausgegeben. Dies ist oft sinnlos und verlangsamt größere Rechnungen mit vielen Zwischenschritten signifikant, und man kann diese

Ausgabe mit einem Semikolon am Ende des Befehls unterdrücken. Dann ergibt sich statt des obigen Beispiels eine deutlich übersichtlichere Ausgabe:

```
>> a = sin(pi/4)^2;
>> b = 84;
>> a*b
ans = 42.0000
```

oder noch kürzer auf einer Zeile

```
a = sin(pi/4)^2; b = 84; a*b
```

Wenn man in einer solchen einzeiligen Liste eine Größe auf dem Bildschirm ausgeben will, dann ersetzt man den Semikolon durch ein Komma.

Wie man in dem obigen Beispiel sieht, sind einige Variable bereits vordefiniert. So kann man zum Beispiel den Wert der Variable `pi` abfragen und erhält:

```
>> pi
ans = 3.1416
```

Man beachte, dass standardmäßig nur die ersten vier Ziffern nach dem Dezimalpunkt angezeigt werden. Die Variable `pi` gibt π jedoch wesentlich genauer an, nämlich bis auf 15 Stellen nach dem Dezimalpunkt. Zu einer solchen Ausgabe kann man mit `format long` wechseln (und mit `format short` wieder zurück auf das Kurzformat).

Es kann auch problemlos mit komplexen Zahlen gerechnet werden. Dazu ist in den Variablen `i` oder `1i` sowie `j` die imaginäre Einheit vordefiniert:

```
>> i
ans = 0 + 1.0000i
>> i^2
ans = -1
```

Man sollte man daher die Variablen `i` und `j` *nicht* anderweitig verwenden! Beispielsweise macht man oft den Fehler in einer Schleife `i` als Zählvariable zu verwenden. Wenn man danach mit komplexen Zahlen rechnen will, führt dies oft zu unsinnigen Ergebnissen. Daher verwendet man sicherheitshalber für die imaginäre Einheit besser die Bezeichnung als `1i`.

Eine Stärke von MATLAB ist es, dass man ohne weiteres mit Vektoren und Matrizen arbeiten kann. Dies benötigen wir natürlich für Rechnungen in der Linearen Algebra, aber auch für einfaches Listen von Einträgen ist das sehr praktisch. Als einführendes Beispiel definieren wir zwei Spaltenvektoren (man beachte dabei das Semikolon!) und berechnen ihr Skalar- und Kreuzprodukt:

```
>> x = [1;2;3]
x =
     1
     2
     3
>> y = [4;5;6];
>> y(2)
ans = 5
>> dot(x,y)
ans = 32
>> cross(x,y)
```

```
ans =
    -3
     6
    -3
```

Man kann mit diesen Vektoren rechnen, sie also mit einer Zahl multiplizieren (wie `4*x`) und sie addieren wie `x+y` falls sie die gleiche Länge haben.[1] In diesem Beispiel sieht man auch, wie man auf einzelne Einträge eines Vektors oder einer Matrix zugreifen kann. Dem Namen der Variable folgt in runden Klammern die Position, hier bezeichnet also `y(2)` den zweiten Eintrag des Vektors `y`.

Mit `z = [1,2,3]` oder `z = [1 2 3]` erhält man einen Zeilenvektor und mit einen Apostroph wandelt man einen Spaltenvektor in einen Zeilenvektor um und umgekehrt. Es gilt also für unser Beispiel `z' = x` und `x' = z`. Die Norm erhält man mit `norm(x)` und mit `sum(x)` die Summe aller Elemente. Bei einer Matrix `a` summiert `sum(a)` über die Elemente jeder Spalte und ergibt einen Zeilenvektor der Spaltensummen. Mit `sum(sum(a))` erhält man dann die Summe aller Matrixelemente.

Aufgabe 1.1 (Lösung Seite 30): Bilden Sie mit den Vektoren `x = [1;2;3]` und `y = [4;5;6]` aus dem Text die Ausdrücke `x'`, `x'*y`, `x'*x`, `x*y'`, `x+3` `z = x+i z'` und `z'*z`. Überlegen Sie sich *vor* jeder Berechnung, was sie ergeben wird!

Bei der Definition von Vektoren und Matrizen werden die verschiedenen Einträge in einer Zeile mit einem Komma abgegrenzt; die Zeilen werden mit einem Semikolon abgeschlossen. Dabei muss eine Matrix immer zeilenweise eingegeben werden wie in dem folgenden Beispiel:

```
>> S = [cos(pi/4),sin(pi/4),0;-sin(pi/4),cos(pi/4),0;0,0,1]
S =
    0.7071    0.7071         0.0000
   -0.7071    0.7071         0.0000
    0.0000    0.0000         1.0000
```

Diese Matrix `S` beschreibt eine Drehung um die z-Achse um einen Winkel von $\pi/4 \mathrel{\widehat{=}} 45°$.

Die mathematischen Operationen `+,-,*,/` werden nun nach den Regeln der Matrixmultiplikation interpretiert. Man kann also ganz einfach die Matrix `S` mit dem Vektor `x` multiplizieren:

```
>> S*x
ans =
    2.1213
    0.7071
    3.0000
```

Das Matrixprodukt `S*S` oder `S^2` ergibt die Hintereinanderschaltung der beiden Drehungen, also eine Drehung um 90°.

Oft kann es aber auch sinnvoll sein, zwei Vektoren oder Matrizen *elementweise* miteinander zu multiplizieren. In diesem Fall ersetzt man die Operationen `*` und `/` durch die elementweisen Operationen `.*` und `./` (jeweils mit einem vorangestellten Punkt). Wenn wir also etwa ver-

[1] Vorsicht jedoch bei Addition einer Zahl zu einem Vektor. Hier wird diese Zahl zu *jeder* Komponente des Vektors addiert!

suchen, mit `x*y` zwei Spaltenvektoren zu multiplizieren, erscheint eine Fehlermeldung. Eine *elementweise* Multiplikation wie `x.*y` ist jedoch möglich, wenn beide Faktoren von gleicher Art sind.

Als eine kleine Übung sollte man zwei 2×2-Matrizen `a` und `b` erzeugen und die beiden Produkte `a*b` und `a.*b` miteinander vergleichen. Welches der Produkte ist kommutativ?

Die elementweise Multiplikation benötigt man zum Beispiel, wenn man dieselbe Funktion auf eine ganze Reihe von Werten anwenden will. Das folgende Beispiel zeigt, wie man numerisch das Maximum einer Funktion der Variable `z` bestimmt:

```
>> z = [-5:0.01:+5];
>> f = -z.^4+2*z.^2+3*z;
>> [fmax,ind] = max(f)
fmax = 4.4347
ind = 627
>> z(ind)
ans = 1.2600
```

In diesem Beispiel wurde zunächst der Zeilenvektor z mit Einträgen im Abstand von 0.01 zwischen -5 und +5 definiert. Definitionen diese Art, mit vielen Einträgen mit gleichem Abstand lassen sich bequem mit einem Doppelpunkt implementieren. Alternativ kann man für diese Definition auch den Befehl `linspace` verwenden. So erzeugt zum Beispiel

```
>> z = linspace(-5,+5,1001);
```

einen Vektor mit 1001 äquidistanten Einträgen im Intervall $[-5,5]$. Dann wird für jeden dieser Werte die Funktion $f(z) = -z^4 + 2z^2 + 3z$ berechnet. Dazu wird dann eine *elementweise* Operation benötigt, und daher wird dem Exponent-Zeichen ein Punkt vorangestellt. Schließlich wird das absolute Maximum der Funktion $f(z)$ mithilfe des Befehls `max` gesucht. Dieser gibt zwei Informationen zurück, den maximalen Wert von `f`, der in der Variablen `fmax` gespeichert wird, und den Index `ind` dieses Maximums. Also ist der 627-te Eintrag des Vektors `f` gerade gleich `4.4347` und dies ist der maximale Wert von `f`. Will man nun wissen, für welchen Wert von `z` das Maximum angenommen wird, so kann man mit `z(ind)` den Wert von `z` an genau dieser Position abfragen.

Bei der Arbeit mit vielen Variablen sind die Befehle `who` und `whos` hilfreich, die die Namen und Eigenschaften aller Variablen zurückgeben. Zur Probe geben wir hier noch die komplexe Zahl `u = 2+i` ein und dann den Befehl `whos` mit dem folgenden Resultat:

```
Variables in the current scope:
   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        S           3x3                         72  double
        ans         1x1                          8  double
        f           1x1001                    8008  double
        fmax        1x1                          8  double
        ind         1x1                          8  double
   c    u           1x1                         16  double
        x           3x1                         24  double
        y           3x1                         24  double
        z           1x1001                    8008  double
Total is 2012 elements using 16104 bytes
```

Der Eintrag `Size` zeigt nun, dass die Variablen `ans`, `fmax`, `ind` und `u` jeweils 1×1-Matrizen sind, also Skalare. Das Spalte `Attr` mit dem Attribut `c` bei `u` gibt an, dass diese Variable komplex ist. Weiterhin sind `x` und `y` jeweils 3×1-Matrizen, also Spaltenvektoren, `z` und `f` sind 1×1001-Matrizen, also Zeilenvektoren und die Variable `S` ist eine 3×3-Matrix.

Die Länge eines Vektors `z` erhält man mit dem Befehl `length`:

```
>> length(z)
ans = 1001
```

Allgemeiner ermittelt man die Dimensionen einer Variable mit dem Befehl `size`:

```
>> size(S)
ans = 3     3
```

Wie schon erwähnt, lassen sich auch Matrizen passender Größe multiplizieren. Die Adjungierte $\mathbf{A}^\dagger$ einer Matrix $\mathbf{A}$, also die komplex konjugierte Transponierte, erhält man mit `A'`, ihre Determinante mit `det(A)` und ihre Inverse mit `inv(A)`, falls die letzten beiden Operationen durchführbar sind.

Der Befehl `I = eye(n)` erzeugt eine $n \times n$-Einheitsmatrix `I`, `zeros(n)` eine Nullmatrix, `ones(n)` eine Matrix mit lauter Einsen, `magic(n)` ein magisches Quadrat, `rand(n)` eine Zufallsmatrix mit gleichverteilten Matrixelementen zwischen null und eins, und `randn(n)` liefert eine Zufallsmatrix mit normalverteilten Einträgen. Entsprechend arbeiten `eye(n,m)`, `eye(size(A))` oder `zeros(n,m)` und `zeros(size(A))` usw.

Aufgabe 1.2 (Lösung Seite 31): Schreiben Sie das Gleichungssystem

$$3x+2y+4z=-5\ , \quad x+3y+2z=13\ , \quad 4x+2y-7z=-13$$

in Matrixform $\mathbf{Ax}=\mathbf{b}$ und berechnen Sie die Lösung $\mathbf{x}$.
Lösen Sie genauso ein Gleichungssystem mit einer $n \times n$-Zufallsmatrix $\mathbf{A}$ und dem Vektor $\mathbf{b}=(1,2,3,\ldots,n)^T$ für $n=100$.
Erzeugen Sie ein magisches Quadrat und überprüfen Sie, dass alle Spalten- und Zeilensummen den gleichen Wert haben.

Die Variable `x` kann man mit dem Befehl `clear x` löschen, was sinnvoll sein kann, wenn x viel Speicherplatz belegt. *Alle* Variablen löscht man mit `clear`, jedoch sollte man dabei vorsichtig sein!

1.2 Hilfe und Dokumentation

Die MATLAB -Programmpakete beherrschen eine beachtlich große Menge von Funktionen und Befehlen. Man kann damit (fast) alles berechnen, nur muss man erst einmal herausfinden wie.

Ein umfangreiche Hilfe und Dokumentation findet man, indem man entweder in der Menüleiste den Menüpunkt *Hilfe* → *Dokumentation* anwählt oder auf der Kommandozeile den Befehl `doc` eingibt. Dann erscheint der Hilfe-Browser in einem neuen Fenster. In diesem Fenster kann man nun auf der linken Seite die verschiedenen Themen durchblättern, der entsprechende Hilfetext wird auf der rechten Seite angezeigt.

Meistens benötigt man jedoch nur eine Hilfe zu einem speziellen Befehl. Dazu gibt es zwei Möglichkeiten: Nach dem Befehl `help befehlsname` erscheint ein Hilfetext im Kommandofenster oder durch `doc befehlsname` öffnet man den Hilfe-Browser mit dem Hilfetext zu dem Befehl.

Oft kennt man den korrekten Namen eines benötigten Befehls jedoch noch nicht. In diesem Fall kann der Befehl `lookfor begriff` gute Dienste leisten. An der Stelle von `begriff` kann jetzt ein beliebiger Begriff stehen, wie der Name einer mathematischen Funktion (in Englisch). Das Programm `lookfor` wird dann den Hilfetext aller Befehle nach diesem `begriff` durchsuchen, sodass man in der Regel den richtigen Befehl zurückgeliefert bekommt. Will man zum Beispiel den Binomialkoeffizienten $\binom{n}{k}$ berechnen, muss man zunächst den richtigen Befehl finden. Mithilfe des Befehls `lookfor binomial` geht das sehr schnell.

Aufgabe 1.3 (Lösung Seite 31): Suchen Sie den MATLAB-Befehl zur Berechnung der Fakultät $n!$ und berechnen Sie $9!$ auf diese Weise.

Eine ganz simple Methode sollte hier nicht unerwähnt bleiben: Sehr oft hilft Google, zum Beispiel durch Eingabe von `matlab fakultät`. Bitte ausprobieren!

1.3 Skripte und Funktionen

Will man eine umfangreiche Simulation durchführen oder mit mehreren Personen an einem Problem arbeiten, ist es natürlich undenkbar, alle Befehle immer wieder einzeln im Kommandofenster einzugeben. Man benötigt stattdessen **Skripte** und **Funktionen**. Wir werden beide im Folgenden als Programme bezeichnen.

Ein **Skript** ist nichts anderes als eine Menge an Befehlen, die man in dem Editor-Fenster in eine Datei schreibt, und dann mit der Dateiendung `.m` abspeichert. Nun kann man dieses Skript ausführen, indem man im Kommandofenster in das entsprechende Verzeichnis wechselt und den Namen des Skripts eingibt (ohne die Dateiendung `.m`). Die Befehle im Skript werden dann genau so ausgeführt, als ob man sie in der Kommandozeile nacheinander eingegeben hätte. Hier als Beispiel ein Skript, das unter dem Namen **funmax1.m** abgespeichert ist:

```
z = [-2.5:0.01:+2.5];
f = -z.^4+2*z.^2+3*z;
[fmax,pos] = max(f)
zmax = z(pos)
plot(z,f)
```

Um es ausführen zu können, müssen wir jedoch erst in das richtige Verzeichnis wechseln. Dies kann man im Kommandofenster mit dem Befehl `cd` (Kurzform von „change directory“) durchführen, oder man verwendet den Verzeichnis-Browser. Diesen öffnet man mit einer Schaltfläche wie beispielsweise `Aktuelles Verzeichnis`. Das Skript wird dann durch Eingabe des Befehls `funmax1` ausgeführt und das Ergebnis wird im Kommandofenster dargestellt. Das Skript enthält auch noch den Befehl `plot(z,f)`, der die Werte von `f` in Abhängigkeit von `z` grafisch darstellt.

Oft möchte man aber, dass das Ergebnis der Berechnung nicht einfach im Kommandofenster ausgegeben wird, sondern für weitere Rechnungen zur Verfügung steht und zum Beispiel einer Variablen zugewiesen werden kann. Zu diesem Zweck verwendet man eine **Funktion** (englisch *function*) statt eines Skripts. Um die Verwendung einer Funktion zu illustrieren, ändern wir das obige Beispiel entsprechend ab

```
function [zmax,fmax] = funmax2(z)
f = -z.^4+2*z.^2+3*z;
[fmax,pos] = max(f);
zmax = z(pos);
```

und speichern sie als **funmax2.m** Eine Funktion beginnt mit dem Schlüsselwort `function`, danach folgt in eckigen Klammern eine Liste derjenigen Variablen, die am Ende zurückgegeben werden sollen. In diesem Fall sind dies `zmax` und `fmax`. Dann folgt ein Gleichheitszeichen, dann der Name der Funktion und in runden Klammern eine Liste der Variablen, die der Funktion von außerhalb übergeben werden, in diesem Fall ist es die Variable z. Man ruft eine Funktion immer mit dem Namen der Datei auf, nicht mit dem Namen, den man in der ersten Zeile verwendet. Man sollte darauf achten, dass beide Namen gleich sind, sonst kann man leicht die Übersicht verlieren.

Die so definierte Funktion kann man dann im Kommandofenster wie folgt verwenden:

```
>> x = [-3:0.01:+3];
>> [xmax,fmax] = funmax2(x)
xmax = 1.2600
fmax = 4.4347
```

Die Eingabevariable `x` kann man jetzt im Nachhinein frei wählen, und man schafft so eine große Flexibilität der Funktion im Gegensatz zu einem Skript. Die Werte der Ausgabevariablen werden im Kommandofenster den Variablen `xmax` und `fmax` zugewiesen. Sie stehen also für weitere Berechnungen zur Verfügung, nicht aber die Variablen, die innerhalb der Funktion benutzt wurden.

Die Übergabe von Variablen an eine Funktion lässt sich natürlich durch ihre Argumentliste bewerkstelligen, aber manchmal möchte man dies vermeiden. Dann kann man auch einige Variablen als **global** erklären. Sind dies beispielsweise die Parameter `par1` und `par2`, so gibt man im Hauptprogramm vor dem Funktionsaufruf den Befehl `global par1 par2` ein. Der gleiche Befehl muss dann auch im Funktionsprogramm erscheinen. Danach stehen dort diese Parameter zur Verfügung.

Schließlich bleibt zu sagen, dass Skripte und Funktionen eine ganz erhebliche Länge und Komplexität annehmen können. In diesem Fall ist eine ausführliche **Kommentierung** unerlässlich! Alles, was in einer Zeile auf ein Prozentzeichen `%` folgt, wird als Kommentar interpretiert. Davon sollte man reichlich Gebrauch machen und den Programmcode dort erklären. Insbesondere bei der Zusammenarbeit in Gruppen muss man seinen eigenen Programmcode so kommentieren, dass er auch für Andere verständlich ist.

1.4 Kontrollstrukturen

Kontrollstrukturen werden verwendet, um den Ablauf eines Computerprogramms zu steuern. Die wichtigen Elemente sind dabei **`if`-Abfragen**, in denen Anweisungen nur durchgeführt werden, wenn eine Bedingung erfüllt ist, und **Schleifen**, in denen Anweisungen wiederholt ausgeführt werden.

Eine **`if`-Abfrage** hat die Struktur

```
if (Bedingung)
  Anweisungen1;
else
  Anweisungen2;
end
```

Sie beginnt mit dem Schlüsselwort `if`. Daraufhin wird die angegebene `Bedingung` ausgewertet. Ist diese *erfüllt* („wahr"), so wird der erste Block von Anweisungen (`Anweisungen1`) ausgeführt. Dann folgt das Schlüsselwort `else` und ein zweiter Block mit Anweisungen (`Anweisungen2`). Diese werden nur dann ausgeführt, wenn die Bedingung *nicht erfüllt* („falsch") ist. Die `if`-Abfrage wird mit dem Schlüsselwort `end` abgeschlossen.

Die Bedingung ist in der Regel ein logischer Ausdruck, wie zum Beispiel $a > b$, der wahr oder falsch sein kann. Allerdings wird auch jede von Null verschiedene Zahl in einem solchen Fall als „wahr" interpretiert, eine Null hingegen als „falsch". Der zweite Teil (`else` und `Anweisungen2`) kann auch weggelassen werden.

Betrachten wir ein Beispiel. Ein Programm soll die geometrische Reihe

$$\sum_{n=0}^{\infty} q^n = \begin{cases} \frac{1}{1-q} & \text{für } |q| < 1 \\ \infty & \text{für } |q| \geq 1 \end{cases} \tag{1.1}$$

auswerten. Dies ist aber nur möglich für $|q| < 1$, denn sonst divergiert die Reihe. Daher verwenden wir eine `if`-Abfrage, um diese Bedingung zu testen. Ist sie erfüllt, wird die Reihe ausgewertet, andernfalls gibt das Programm die Meldung zurück, dass die Reihe divergiert:

```
>> q = 0.9;
if (abs(q) < 1)
  summe = 1/(1-q)
else
  fprintf('Die Reihe divergiert.')
end
```

Hier hat die Variable `q` den Wert 0.9 und die Bedingung `abs(q) < 1` ist wahr. Also wird der Befehl `summe = 1/(1-q)` ausgeführt, um die unendliche Summe zu berechnen. Hat die Variable `q` den Wert 1.1, ist `abs(q) < 1` falsch, sodass nun `fprintf('Die Reihe divergiert.')` ausgeführt wird. Dieser Befehl gibt einen Text auf der Kommandozeile aus.

Ebenso wichtige Strukturen sind **Schleifen**. Dabei wird ein Block von Anweisungen entweder solange wiederholt, wie der Programmierer vorgibt (eine `for`-Schleife) oder solange wie eine gewisse Bedingung erfüllt ist (eine `while`-Schleife). Die allgemeine Form der **`for`-Schleife** ist

```
for Zaehlvariable = Vektor
  Anweisungen;
end
```

Die `Zaehlvariable` nimmt nacheinander alle Werte an, die durch den `Vektor` vorgegeben sind. Für jeden dieser Werte werden nacheinander die Anweisungen ausgeführt. Die `for`-Schleife endet dann immer mit dem Schlüsselwort `end`. Als Beispiel betrachten wir die Berechnung der Fakultät $n!$ einer Zahl n. Dabei wird eine Operation, hier die Multiplikation, für eine gegebene Anzahl von Schritten (genau n mal) wiederholt. Damit kann man die Berechnung mit einer `for`-Schleife durchführen:

```
>> n = 9;
>> fak = 1;
>> for k = 1:n
fak = fak*k;
end
>> fak
fak = 362880
```

In diesem Beispiel wird die Fakultät der Variable `n` berechnet, die vorher mit dem Wert 9 initialisiert wurde. Dann wird die Variable `fak` für die Fakultät initialisiert, in diesem Fall wird sie gleich eins gesetzt. In der eigentlichen `for`-Schleife nimmt die Variable `k` nacheinander alle Werte des Vektors `1:n` an. Dieser Befehl mit dem Doppelpunkt erzeugt einen Vektor mit Einträgen im Abstand 1 im Intervall $[1,9]$. (Hier könnte man auch die Schreibweise `1:1:n` oder `[1:1:n]` verwenden.) In jedem Schritt wird die Variable `fak` mit dem neuen Wert von `k` multipliziert. So erhält man schließlich das Ergebnis $9! = 362880$ (vgl. dazu auch Aufgabe 1.3).

Kann oder will man die Anzahl der Wiederholungen nicht vorher festlegen, sondern stattdessen an eine Bedingung knüpfen, so verwendet man die **`while`-Schleife**. Die allgemeine Syntax dieser Schleife ist

```
while (Bedingung)
  Anweisungen;
end
```

Die Anweisungen werden dabei so lange wiederholt ausgeführt, wie die Bedingung erfüllt ist. Dabei muss man darauf achten, dass die Bedingung nicht ewig erfüllt bleibt, denn sonst wird die Schleife immer und immer wieder ausgeführt, und das Programm endet niemals. In einem solchen Fall hilft nur noch ein **manueller Abbruch** der Schleife: Dazu drückt man die Tastenkombination `Ctrl + C` bzw. `Strg + C`.

Im folgenden Beispiel betrachten wir die Folge $x_{n+1} = x_n/2$ mit $x_1 = 1$, für die wir die Summe aller Folgenglieder berechnen wollen. Wir berechnen also die Werte von x_n in einer Schleife und summieren diese Werte. Der Algorithmus muss aber irgendwann ein Ende finden. Daher legen wir fest, dass die Schleife nur so lange ausgeführt wird, wie $x_n > 10^{-6}$ gilt. Danach ändert sich die Zahl nur noch marginal. Dieses Verfahren wird dann folgendermaßen implementiert:

```
xn = 1;
>> summe = 0;
>> while (xn > 1e-6)
summe = summe+xn;
xn = xn/2;
end
>> summe
summe = 2.0000
```

Zunächst werden die Variablen `xn` und `summe` initialisiert. Dann folgt die `while`-Schleife mit der Bedingung `xn > 1e-6`. Im letzteren Ausdruck steht dabei das `e-6` für eine Multiplikation mit 10^{-6}. So kann man sehr große und sehr kleine Zahlen eingeben. Innerhalb der Schleife führt dieses Programm dann wiederholt zwei Anweisungen durch: Zunächst wird der aktuelle Wert von `xn` zu der Variable `summe` addiert, dann wird das nächste Folgenglied berechnet. Die `while`-Schleife wird mit dem Schlüsselwort `end` abgeschlossen. Nach der Ausführung der Schleife enthält die Variable `summe` den Wert 2, den Summenwert der geometrischen Reihe (1.1) für $q = 1/2$.

Man sollte wissen, dass Schleifen in der Ausführung recht langsam sind, Matrixoperationen jedoch sehr schnell. Daher hat sich eine gewisse Kunst entwickelt, Schleifen weitgehend zu vermeiden und durch Matrixoperationen zu ersetzen. Dies vermindert jedoch in aller Regel die Lesbarkeit eines Programms. Die folgende Aufgabe vermittelt einen ersten Eindruck von der Geschwindigkeit der Ausführung einer Schleifenstruktur. Dabei sollte man von der Möglichkeit Gebrauch machen, die Rechenzeit eines Rechenvorganges zu ermitteln. Mit `tic` und `toc` wird die Rechenzeit (CPU-Zeit) für die Ausführung eines Programmteils zwischen den Kommandos `tic` und `toc` ausgegeben, beispielsweise als `Elapsed time is 23.2963 seconds.`

Aufgabe 1.4 (Lösung Seite 31): Erstellen Sie ein Programm, ein Skript, das zwei $n \times n$ Matrizen **A** und **B** erzeugt (beispielsweise als Zufallsmatrizen `A = rand(n)` und `B = rand(n)`) und die Matrixmultiplikation **C** = **AB** explizit nach der bekannten Regel

$$C_{\ell m} = \sum_k A_{\ell k} B_{km}$$

für die Matrixelemente durchführt. Vergleichen Sie die Rechenzeit mit der einer normalen Matrixmultiplikation `A*B` für $n = 3$ und $n = 100$.

Als letztes Negativbeispiel hier ein Programm zur Berechnung der Sinus-Funktion:

```
n = 100;
dx = 2*pi/n;
for k = 1:n
  x(k) = k*dx;
  y(k) = sin(x(k));
end
```

Kürzer und wesentlich(!) schneller ist die schleifenfreie Version

```
x = linspace(0,2*pi,1024);
y = sin(x);
```

1.5 Dateneingabe und -ausgabe

Bisher haben wir alle Daten zumeist fest in einem Skript angegeben, ohne die Möglichkeit einer späteren Eingabe. Als Ausgabe stand uns nur eine Möglichkeit zur Verfügung, die Ausgabe des Wertes einer Variable. Mehr Möglichkeiten bieten die Befehle `input` und `fprintf`.

Der Befehl `input` wird benutzt, um während der Laufzeit eines Skriptes oder einer Funktion Daten vom Benutzer abzufragen. Damit gibt man eine Frage auf der Kommandozeile aus und der Nutzer muss eine Eingabe machen. Dies verdeutlicht das folgende Beispiel:

```
>> alter = input('Wie alt bist du ? ')
Wie alt bist du ? 29
alter = 29
```

Um den Wert einer Variable auf der Kommandozeile auszugeben haben wir bisher einfach bei der Berechnung das Semikolon am Ende weggelassen. Diese Möglichkeit wird oft verwendet, ist jedoch sehr beschränkt. Was macht man zum Beispiel, wenn ein Skript einen Text zurückgeben soll? Dazu steht der Befehl `fprintf` zur Verfügung, mit dem man zum Beispiel einen Text auf der Kommandozeile ausgeben kann durch:

```
>> fprintf('MATLAB ist super\n');
MATLAB ist super
```

Der Text `MATLAB ist super` muss dabei in einfachen Anführungszeichen eingeschlossen sein. Weiterhin haben wir das Zeichen `\n` verwendet, das einen Zeilenumbruch erzwingt.

Außerdem kann `fprintf` auch den Wert einer Variable zurückgeben, wobei man angeben muss, in welchem Format dies geschehen soll. Dies verdeutlicht das folgende Beispiel:

```
>> fprintf('pi auf drei Stellen: \%1.3f \n',pi);
pi auf drei Stellen: 3.142
```

In diesem Beispiel ist `%1.3f` ein Platzhalter für den Wert der Variable, die später angegeben wird. Hier ist das die Variable `pi`. In diesem Platzhalter bedeutet dann `f`, dass der Wert als Fließkommazahl ausgegeben wird, und `1.3` besagt, dass genau eine Ziffer vor dem Dezimalpunkt ausgegeben wird und drei Ziffern danach. Mehr Informationen zu den verschiedenen Formaten erfährt man in der Hilfe mit dem Befehl `help fprintf`.

Oft wird man jedoch größere Mengen an Daten aus einer Datei einlesen oder in einer Datei abspeichern wollen. Dies ist recht einfach möglich. Der Befehl `save dateiname.mat` speichert *alle* aktuellen Variablen in einer Datei mit dem Namen `dateiname.mat` im aktuellen Verzeichnis ab. Umgekehrt lädt der Befehl `load dateiname.mat` alle Variablen aus der Datei `dateiname.mat` ein. Will man nur wenige Variablen abspeichern, so stellt man die Namen der Variablen hinter den Dateinamen:

```
>> save dateiname.mat variable1 variable2
```

Manchmal ist auch eine andere Form des `save`-Befehls nützlich, der die Namen als Strings übergibt:

```
>> save('dateiname.mat','variable1','variable2');
```

Diese Form wird insbesondere gebraucht, wenn die Namen selbst wieder in einer Variable abgespeichert sind.

Hier wird zum Speichern der Daten ein eigenes Format (`mat`) verwendet, in dem alle Informationen über die Variablen gesichert werden. Dabei werden die Daten komprimiert.

Außerdem kann MATLAB mit einfachen ASCII-Dateien umgehen, also mit einfachen Textdateien. Dazu muss man jedoch explizit angeben, dass dieses Format gewählt werden soll, wie in dem folgenden Beispiel:

```
>> M = [1,2,3;4,5,6;7,8,9]
M =
```

```
     1     2     3
     4     5     6
     7     8     9
>> save('test.txt','M','-ascii');
```

Beim Abspeichern einer Matrix im ASCII-Format werden alle Einträge durch Leerzeichen bzw. Zeilenumbrüche getrennt. Die Datei `test.txt`, die im obigen Beispiel erzeugt wurde, sieht dann so aus:

```
1.0000000e+00   2.0000000e+00   3.0000000e+00
4.0000000e+00   5.0000000e+00   6.0000000e+00
7.0000000e+00   8.0000000e+00   9.0000000e+00
```

Das Abspeichern von vielen Variablen in einer einzigen Datei ist so aber nicht unbedingt zu empfehlen, denn die Dateien werden schnell sehr unübersichtlich, da keine Variablennamen gespeichert werden.

Liest man eine solche ASCII-Datei wieder ein, werden *alle* Einträge in nur eine Matrix geschrieben, die den Namen der Datei trägt:

```
>> load('test.txt','-ascii')
>> whos
  Name      Size            Bytes  Class     Attributes
  test      3x3                72  double
>> test
test =
     1     2     3
     4     5     6
     7     8     9
```

1.6 Grafikausgabe

Für die Verwendung von MATLAB in der Physik sprechen weiterhin die vielseitigen und einfach zu benutzenden Programme zur grafischen Ausgabe. Auch viele Physiker, die ihre eigentlichen Simulationen in C oder Fortran programmieren, nutzen MATLAB daher zur Auswertung und Darstellung der Ergebnisse. Eine Übersicht über die Kommandos zur Erstellung und Manipulation von Grafiken erhält man mit den Kommandos `help graph2d` und `help graph3d`.

Wir wollen die wichtigsten Kommandos anhand eines Beispielprogramms einführen, das die plancksche Strahlungsformel

$$W(\nu,T) = \frac{8\pi h\nu^3}{c^3}\,\frac{1}{\mathrm{e}^{h\nu/k_\mathrm{B}T}-1} \tag{1.2}$$

in Abhängigkeit von der Frequenz ν und der Temperatur T plottet. Dabei ist h das plancksche Wirkungsquantum, k_B die Boltzmann Konstante und c die Lichtgeschwindigkeit. Wie in Bild 1.1 gezeigt, erzeugt das Programm **planck.m** sowohl eine zweidimensionale Darstellung von $W(\nu,T)$ als auch eine eindimensionale für eine feste Temperatur $T = 600K$.

```
function []= planck()
% grafische Darstellung der planckschen Strahlungsformel
h = 6.6261e-34; kB = 1.3807e-23; c = 299792458;
```

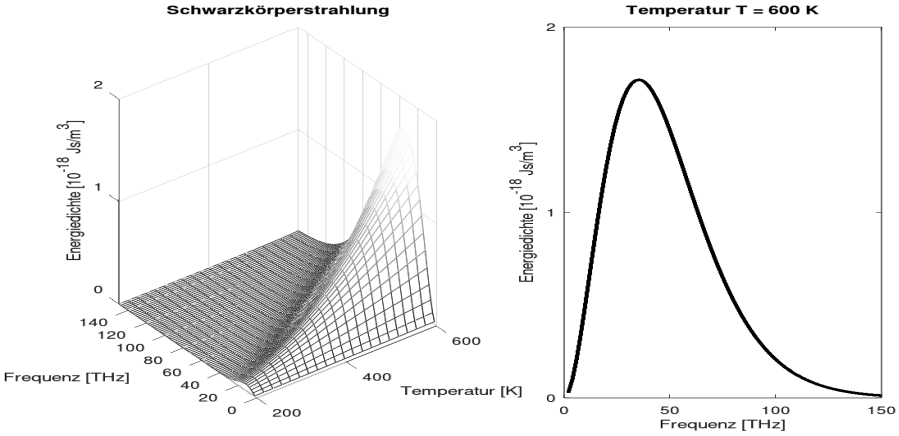


Bild 1.1 Spektrum eines schwarzen Körpers: Gezeigt ist die spektrale Energiedichte (1.2) in Abhängigkeit der Frequenz ν und der Temperatur T (links) und für eine feste Temperatur von 600 K (rechts). Dieses Bild wurde mit dem Programm `planck.m` erzeugt, das im Text erläutert ist.

Zunächst werden die physikalischen Konstanten definiert, die in die Strahlungsformel (1.2) eingehen. Dabei ist darauf zu achten, dass alle Werte konsistent in einem Einheitensystem, hier den SI-Einheiten, angegeben werden.

```
T0 = [200:20:600];
nu0 = [0:2:150]*10^12;
[T,nu] = meshgrid(T0,nu0);
W = 8*pi*h*nu.^3/c^3./(exp(h*nu/kB./T)-1);
```

In den Vektoren `T0` und `nu0` werden die Werte der Temperatur und der Frequenz definiert, für die die Strahlungsformel ausgewertet werden soll. Der Befehl `meshgrid` erzeugt nun aus den beiden unterschiedlich großen Vektoren zwei gleich große Matrizen, indem die Werte der Temperatur und der Frequenz entsprechend oft wiederholt werden. Man erhält zwei gleich große Matrizen `T` und `nu` mit

$$\texttt{T}(\texttt{n},:) = \texttt{T0} \quad \text{und} \quad \texttt{nu}(:,\texttt{m}) = \texttt{nu0} \tag{1.3}$$

für alle Zeilen `n` und alle Spalten `m`. Mit diesen Matrizen können wir schließlich die plancksche Strahlungsformel (1.2) auswerten und erhalten die Energiedichte `W`.

Damit kommen wir endlich zum eigentlichen Thema diese Abschnitts, dem Erstellen einer dreidimensionalen Grafik. Zunächst teilen wir das Grafikfenster mit dem Befehl `subplot` in zwei Teile auf. Dabei geben die ersten beiden Argumente die Anzahl der Unterteilungen in vertikaler bzw. horizontaler Richtung an, das dritte Argument wählt in diesem Fall den linken Teil aus. Die eigentliche Grafik wird dann mit dem Befehl `mesh` erzeugt. Dieser stellt die Energiedichte `W` in Abhängigkeit von `T` und `nu` in Form eines Gitternetzes dreidimensional dar. Die drei Argumente stehen dabei für die Koordinaten einzelner Gitterpunkte in x-, y- bzw. z-Richtung. Schließlich legt der Befehl `colormap` die Farbskala an, in der das Gitternetz eingefärbt wird:

```
subplot(1,2,1);
mesh(T,nu/1e12,W*1e18);
```

```
colormap(hot);
```

Hier wurde mit `colormap(hot)` eine Farbskala gewählt, die von schwarz über rot und gelb bis hin zu weiß variiert. Mehr dazu findet man mit `help colormap`.

Mit dem Befehl `set` können nun die Attribute dieser Grafik manipuliert werden, wie hier mit

```
set(gca,'xlim',[200 600],'xtick',[200:200:600]);
set(gca,'ylim',[0 150],'ztick',[0:50:150]);
set(gca,'zlim',[0 2],'ztick',[0:1:2]);
```

Das erste Argument `gca` gibt dabei nur an, dass der aktuelle Graph manipuliert wird. Dann folgen in einer beliebig langen Liste jeweils zunächst der Name eines Attributs und dann sein neuer Wert. So stellen wir mit dem Attribut `xlim` zum Beispiel ein, dass auf der x-Achse die Werte von 200 bis 600 dargestellt werden sollen.[2]

Zum Beschriften der Achsen stehen die Befehle `xlabel`, `ylabel` und `zlabel` zur Verfügung. Zudem kann man mit dem Befehl `title` eine Titelzeile einfügen:

```
xlabel('Temperatur [K]');
ylabel('Frequenz [THz]');
zlabel('Energiedichte [10^{-18} Js/m^3]');
title('Schwarzkörperstrahlung');
```

Zusätzlich wollen wir noch die Strahlungsformel für einen festen Wert der Temperatur grafisch darstellen. Dazu wählen wir mit `subplot` zunächst den rechten Teil des Grafikfensters aus. Der Graph der Funktion $W(\nu)$ wird dann mit dem Befehl `plot` geplottet. Die ersten beiden Argumente geben dabei die x- und die y-Werte als Vektoren an. Das dritte Argument gibt den Stil des Graphs an: Der Bindestrich steht für eine durchgezogene Linie, der Buchstabe `k` wählt schwarz als Farbe für diese Linie aus.

```
subplot(1,2,2);
plot(nu/1e12,W(:,end)*1e18,'k-','linewidth',2);
```

Schließlich kann man, wie bei dem Befehl `set`, noch weitere Attribute mit Namen und Wert angeben. Hier setzen wir die Linienbreite auf zwei Punkte, etwas dicker als normal:

```
set(gca,'xlim',[0 150],'xtick',[0:50:150]);
set(gca,'ylim',[0 2],'ytick',[0:1:2]);
xlabel('Frequenz [THz]');
ylabel('Energiedichte [10^{-18} Js/m^3]');
title(['Temperatur T = ' num2str(T(end,end)) ' K']);
```

Zuletzt werden auch hier die x- und y-Achse mit dem Befehl `set` skaliert und mit `xlabel` bzw. `ylabel` beschriftet. Beachtung verdient hier vor allem die Erstellung des Titels mit dem Befehl `title`. Diesen fest einzugeben, ist oft viel zu unflexibel, insbesondere wenn man mehrere Plots für unterschiedliche Werte eines Parameters erstellen will. Um den Titel dynamisch zu erzeugen, lesen wir den aktuellen Wert der Temperatur aus der Variable `T` aus. Dies ergibt hier die Zahl 600, die wir noch mit dem Befehl `num2str` in die Zeichenkette 600 umwandeln müssen. Dann werden einzelne Teile des Titels wie Zahlen in einem Zeilenvektor mithilfe der eckigen Klammern `[ ]` zusammengefügt.

[2] Das Gegenstück zu `set` ist der Befehl `get`, mit dem sich die aktuellen Werte der Grafik-Attribute abfragen lassen.

Als ein weiteres Beispiel für die grafischen Möglichkeiten soll die Oberfläche eines **Torus** dargestellt werden, die man durch

$$x = (1 + r\cos\varphi)\cos\theta\,, \quad y = (1 + r\cos\varphi)\sin\theta\,, \quad z = r\sin\theta \tag{1.4}$$

mit $0 \le \theta, \varphi < 2\pi$ parametrisieren kann. Für $r = 0$ ergibt das einen Einheitskreis in der (x, y)-Ebene, parametrisiert durch den Winkel θ, der für $r > 0$ zu einer Röhre mit Radius r aufgeweitet wird. Bei einem Schnitt für einen festen Wert von θ erhält man als Schnittkurve einen Kreis mit Radius r, parametrisiert durch den Winkel φ.

In dem Programm **torus.m** wird die Torusoberfläche für $r = 0.4$ dargestellt. Dabei werden mit $n = 80$ die Winkel diskretisiert und die Toruspunkte x, y, z berechnet. Ihre grafische Darstellung übernimmt das Grafikprogramm `surf`.

```
% torus.m - Torus-Grafik
r = 0.4; n = 80;
theta = 2*pi*(0:n)/n; phi = 2*pi*(0:n)'/n;
x = (1+r*cos(phi))*cos(theta);
y = (1+r*cos(phi))*sin(theta);
z = r*sin(phi)*ones(size(theta));
surf (x, y, z); hold on
a = (1+r)/sqrt(2)+0.5; axis([-a,a,-a,a,-a,a]);
```

Im zweiten Teil des Programms wird eine Bahnkurve auf der Torusoberfläche berechnet mit einer Umlaufffrequenz $\omega = 3\sqrt{2}$ im Winkel φ und einer Frequenz eins im Winkel θ.

```
t = 0:0.01:26; om=3*sqrt(2);
phit = t*om; thetat = t;
xt = (1+r*cos(phit)).*cos(thetat);
yt = (1+r*cos(phit)).*sin(thetat);
zt = r*sin(phit).*ones(size(thetat));
plot3(xt,yt,zt,'r'); axis off
colormap(gray); shading flat
```

Hier ist das Frequenzverhältnis irrational mit der Konsequenz, dass für große Werte der Variable `t`, der „Zeit“, die Bahn den gesamten Torus überdeckt. Für ein rationales Frequenzverhältnis ist die Bahn periodisch. Bitte ausprobieren!

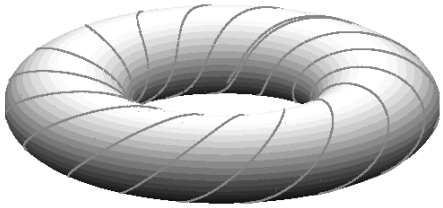


Bild 1.2 Oberfläche eines Torus mit der Bahn eines klassischen Teilchens

In dem Programm oben bewirkte der Befehl `hold on`, dass die folgende Grafikausgabe im gleichen Bildfenster erfolgt, ohne die vorherigen Daten zu löschen; mit `hold off` lässt sich dies wieder ausschalten. Ein neues Bildfenster lässt sich mit `figure` öffnen. Die Bildfenster werden dann fortlaufend nummeriert und mit `figure 2` oder `figure(2)` kann man das Bildfenster Nummer 2 aktivieren. Mit `close` schließt man das aktive Bild und mit `close all` alle Bilder.

1.7 Einige Tipps und Tricks

Grafische Voreinstellungen: In den Programmen werden in grafischen Darstellungen Kurven mit der voreingestellten Strichstärke gezeichnet. Das Gleiche gilt auch für Beschriftungen. Falls man das ändern möchte, gibt man einfach vor dem Programmstart Befehle wie

```
set(0,'DefaultLineLineWidth',2)
set(0,'DefaultaxesLineWidth',2)
set(0,'DefaultaxesFontSize', 26)
```

ein. Das wirkt dann auf *alle* folgenden Plots.

Zugriff auf Teile einer Matrix: Bisher wurde mit `a(2,3)` auf genau ein Matrixelement der Matrix `a` zugegriffen, nämlich das in der zweiten Zeile und der dritten Spalte. Das kann man aber noch ausbauen. Durch `a([2:6],3)` erhält man einen Spaltenvektor, der die fünf Elemente $a_{2,3}$ bis $a_{6,3}$ enthält, mit `a([2:2:6],3)` einen Spaltenvektor mit den Elementen $a_{2,3}$, $a_{4,3}$, $a_{6,3}$, und `a([2:4],[1:2])` erzeugt eine 3×2-Untermatrix.

Wenn Sie die Aufgabe hätten, die mittlere 3×3-Matrix einer 5×5-Matrix `a` durch eine Einheitsmatrix zu ersetzen, so können Sie das durch `a([2:4],[2:4])=eye(3)` erreichen.

Mit `diag(a)` erhält man einen Spaltenvektor mit den Einträgen der Diagonale einer Matrix `a`. Wenn man so auf einen Vektor zugreift, entsteht eine Diagonalmatrix mit diesem Vektor auf der Diagonale. Folglich liefert `diag(diag(a))` eine Diagonalmatrix, die nur die Diagonale von `a` enthält. Durch `diag(a,1)` greift man auf die erste Nebendiagonale von `a` zu.

Weitere hier interessierende Befehle sind `triu(a)` zur Erzeugung der oberen Dreiecksmatrix, `tril(a)` für die untere und `triu(a,1)` für die Dreiecksmatrix oberhalb der ersten Nebendiagonale. Der Rest wird jeweils mit Nullen gefüllt.

Logische Ausdrücke: Bei der Diskussion von Schleifen haben wir schon mit logischen Ausdrücken wie `a>b` gearbeitet. Ist ein solcher Ausdruck wahr, hat er den Wert `1`, sonst `0`. Die Eingabe von `5>4` ergibt also als Antwort eine `1`, die Eingabe `5<4` eine `0`, `4==4` (gleich) oder `4>=4` (größer oder gleich) oder `4<=4` (kleiner oder gleich) eine `1` und `4~=4` (ungleich) eine `0`.

Man kann mit logischen Ausdrücken wie `a` und `a` rechnen und die folgenden sechs Operationen ausführen:

```
and    or    not   xor   any   all
```

Für die ersten drei Operationen gibt es auch die Kurzformen `&` und `|` sowie die Negation `~`.Man sollte sich mit diesen Operationen vertraut machen, indem man sie ausprobiert. Beispielsweise können wir zwei Matrizen erzeugen wie

```
a = [1 2;4 1]        b = [3 1;1 1]
```

und dann `c = (a>1)` ausführen, mit dem Ergebnis $c = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, genau wie für `c = (a>b)`. Dann ergibt `not(c)` oder `~c` die Negation $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$.

Mit `any(b==1)` wird für jeder Spalte von `b` abgefragt, ob irgendein Element gleich eins ist. Da das für alle Spalten von `b` erfüllt ist, erhalten wir den Zeilenvektor $(1,1)$. Mit `all(b==1)` gilt das, wenn es für alle Elemente erfüllt ist, mit dem Resultat $(0,1)$, und `all(any(b==1))` ist wahr, liefert also eine eins.

Als letzten Test wollen wir die Operation `xor` betrachten, also das exklusive „oder“. Es ergibt „wahr“, wenn eines der Argumente wahr ist, aber nicht beide. Wenn es auf Matrizen operiert, dann wirkt es elementweise. Was liefert hier also `xor(a>1,a==b)`? Wenn Sie jetzt mit $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right)$ geantwortet haben, dann sollten Sie zufrieden sein.