Graphik-Workshop

Klaus Betzler *

Universität Osnabrück

Sommersemester 2004

Inhaltsverzeichnis

* Klaus.Betzler@Physik.Uni-Osnabrueck.de 10. April 2004

1 PostScript

Mathematisch-naturwissenschaftliche Texte, die gedruckt werden sollen (Publikationen, Abschlussarbeiten, Ausarbeitungen zu Praktika oder Übungen), werden überwiegend mit LaTeX erstellt. Als Distributionsformat kann daraus DVI, PS, PDF oder – über LaTeX2HTML o. ä. – HTML generiert werden. Die flexibelste und einheitlichste Art, Graphiken in solche Texte einzubinden, besteht darin, die Graphik aus der Graphik-Anwendung in *Encapsulated-PostScript*[1]-Format (EPS) zu exportieren, und die EPS-Datei in die LaTeX-Texte zu integrieren. Einige Programme, insbesondere solche aus der Windows-Welt, können EPS nicht oder nur fehlerhaft direkt exportieren. In solchen Fällen wird zunächst *PostScript*[1] (PS) erstellt (Ausgabe auf einen PS-Drucker und Umleitung in eine Datei), dann daraus EPS erzeugt – beispielsweise mit *GhostScript* (*GhostView*). Da hierbei bisweilen auch von Hand eingegriffen werden muss, ist es zweckmäßig, sich ein wenig mit den PostScript-Interna vertraut zu machen.

1.1 Grundlagen

PostScript wurde um 1985 von der Firma Adobe als einheitliche Seitenbeschreibungssprache für Drucker entwickelt. Sie wird derzeit praktisch überall als geräteunabhängiges Druck- und Graphikformat verwendet. In der Sprache lassen sich alle Objekte formulieren, die eine Druckseite enthalten kann: Texte, graphische Elemente wie Kurven oder Flächen, Pixel-Bilder. Daneben bietet PostScript auch (fast) alle Möglichkeiten einer einfachen Programmiersprache. Da jedoch keine Sicherheitskonzepte eingebaut sind, kann die unkritische Verwendung problematisch sein[2]. Diese Probleme betreffen allerdings kaum die lokale Verwendung als Zwischenformat oder das direkte Ausdrucken im PS-Format sondern praktisch nur das Betrachten oder indirekte Ausdrucken von unsicheren PostScript-Dateien mit *GhostView* oder *GhostScript*[3]. Die Möglichkeiten der Programmiersprache PostScript kann man nicht nur dazu nutzen, Seiteninhalte zu beschreiben, sondern darüber hinaus auch dazu, graphische Objekte mathematisch zu konstruieren. PostScript wurde bis heute in zwei größeren Schritten erweitert und an die Möglichkeiten moderner Drucker angepasst (Level 1 ⇒ Level 2 ⇒ Level 3), der heutige Stand ist Level 3.

Die Firma Adobe hat mit der Entwicklung von PostScript zwei Handbücher dazu herausgegeben, die PostScript einführend beschreiben und definieren: ein *Tutorial and Cookbook* [1], das allerdings nur PostScript Level 1 umfasst, und ein *Reference Manual* [2], das

[1] PostScript ist ein eingetragenes Warenzeichen der Firma Adobe.

[2] Um druckfertige Publikationen im Internet bereitzustellen, wurde von Adobe das *Portable Document Format* (PDF) entwickelt, das vom Konzept her beträchtlich sicherer ist, außerdem die Möglichkeit bietet, interne und externe Hyperlinks zu integrieren. In diesem Bereich wurde PostScript von PDF weitgehend abgelöst.

[3] *GhostScript* wurde als PostScript-Interpreter entwickelt, um PS-Dateien auch auf Nicht-PS-Druckern ausgeben zu können. Da dabei der PS-Code auf dem Rechner abgearbeitet wird, treten andere Sicherheitsprobleme als an PS-Druckern auf. *GhostScript* wird üblicherweise über das zugehörige Front-End *GhostView* bzw. *GSview* bedient.

in seiner aktuellen Ausgabe den derzeitigen Entwicklungsstand von Level 3 dokumentiert. Umfangreiche Informationen zu PostScript, Bücher dazu im PDF-Format (auch die beiden genannten), Hilfsmittel zur Erstellung von PostScript aus anderen Programmiersprachen und vieles mehr findet man im Internet unter [3].

1.2 Einbinden von EPS-Bildern in LaTeX-Text

EPS-Bilder werden mit der Anweisung `\includegraphics` aus dem Paket *graphicx* in den LaTeX-Text eingebunden. Als Beispiel hier ein mit *Excel* erstelltes Balkendiagramm:

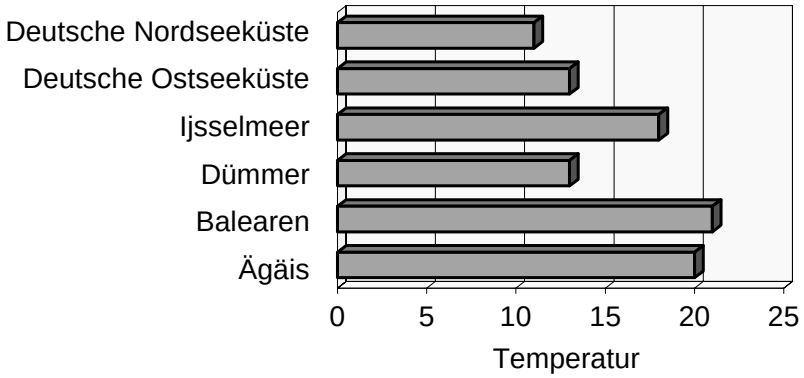


Abbildung 1: Wassertemperaturen 15. Oktober 2000 (Quelle: Neue Osnabrücker Zeitung).

Das nachstehende LaTeX-Fragment veranschaulicht die Verwendung:

```
75  \subsection{Einbinden von EPS-Bildern in LaTeX-Text}
76
77  EPS-Bilder werden mit der Anweisung \verb?\includegraphics? aus
78  dem Paket \emph{graphicx} in den \LaTeX-Text eingebunden. Als
79  Beispiel hier ein mit \emph{Excel} erstelltes Balkendiagramm:
80
81  \begin{figure}[h]
82   \centering\includegraphics[width=0.7\hsize]{ps/wasser0.eps}
83   \caption{Wassertemperaturen 15.~Oktober 2000 (Quelle: Neue
84   Osnabrücker Zeitung).}\label{fig:wasser}
85  \end{figure}
```

Wie in LaTeX üblich, wird alles (Bild, Bildunterschrift und Verweismarke) in eine `figure`-Umgebung gepackt, die als Einheit von LaTeX platziert werden kann (zu dieser Gruppe der *floats* gehört auch die `table`-Umgebung). Die Wunschplatzierung kann als Argument in der eckigen Klammer angegeben werden (h, t, b, p für *h*ere, *t*op, *b*ottom, extra *p*age oder Kombinationen davon).

`\centering` sorgt für eine horizontale Zentrierung. Zum Einbau der Graphik wird die Anweisung `\includegraphics` aus dem *graphicx*-Paket verwendet (im *graphics*-Paket gibt

es diesen Befehl ebenfalls, die beiden sind nicht kompatibel!). Als Zusatzargument kann die gewünschte Breite (*width*) und Höhe (*height*) angegeben werden. Ohne diese Angabe(n) wird das Bild in seiner Originalgröße gesetzt; wird beides angegeben, so wird das Bild entsprechen verzerrt. Im Beispiel wird die Breite auf 70 % der verfügbaren Breite (das ist hier die Textbreite) eingestellt.

1.3 PostScript und Encapsulated PostScript, Probleme

PostScript ist zur direkten Ausgabe auf einen Drucker gedacht, ersetzt somit ältere Druckerkommandosprachen. Eine PostScript-Datei kann in der Regel ohne weitere Zusätze an einen passenden Drucker geschickt werden. Das Encapsulated-PostScript-Format wurde als Ergänzung definiert, um fertige PostScript-Teile skalierbar in andere PostScript-Dateien einbinden zu können. Es entspricht weitgehend PostScript. Abweichend davon kann eine EPS-Datei logischerweise immer nur eine einzelne Seite oder eine einzelne Graphik beinhalten. Darüber hinaus sollte die Größe der Graphik in der Datei definiert sein, um die Skalierung zu erleichtern. Bestimmte Erweiterungen, die in PostScript zulässig sind, sollten im EPS-Dateien nicht verwendet werden, z. B. druckerspezifische Anweisungen.

Ein einseitiges PostScript-Dokument kann meist problemlos in eine EPS-Datei gewandelt werden. Das erledigt *GhostScript* dadurch, dass es die automatisch oder durch Benutzerinteraktion festgestellte Größe der Graphik am Dateianfang als *BoundingBox* vermerkt (Menupunkt *Datei* ⇒ *PS zu EPS* in *GhostView*). Eine solche Umwandlung von PS zu EPS ist immer dann nötig, wenn das benutzte Graphikprogramm nicht oder nur fehlerhaft EPS direkt generieren kann, man 'druckt' dann die Graphik auf einen PostScript-Drucker aus, lenkt die Ausgabe aber auf eine Datei um, erhält so die Graphik im PostScript-Format. Falls in der so erzeugten PostScript-Datei überhaupt eine *BoundingBox* vermerkt wird, entspricht die meist dem Papierformat, nicht der Größe der Graphik. *GhostScript* bestimmt die richtige *BoundingBox* und trägt sie in die Datei ein.

Probleme bei der Verwendung der so erstellten EPS-Datei ergeben sich immer dann, wenn die PS-Datei Druckeranweisungen enthält oder andere Inhalte, die mit der EPS-Verwendung nicht kompatibel sind. Zwei Beispiele solcher fehlerhaften EPS-Dateien sind in Abbildung 2 dargestellt. Im linken Teilbild wurde eine EPS-Datei verwendet, die mit *GhostView* aus einer PS-Datei für HP-Drucker gewandelt wurde, rechts aus einer PS-Datei für Apple Laserwriter.

Links fehlt die Graphik einschließlich aller vor der Graphik stehenden Inhalte. Der Grund liegt in Druckeranweisungen, die ein komplettes *Reset* der PostScript-Maschine bewirken. Ein solches *Reset* ist vor und nach dem Ausdruck einer kompletten Seite sinnvoll, nicht aber vor und nach einem Einzelobjekt. Ein Beispiel für eine derartige Anweisung ist:

```
179  featurebegin{
180  %%BeginFeature: *Duplex None
181    <</Duplex false>> setpagedevice
182  %%EndFeature
```

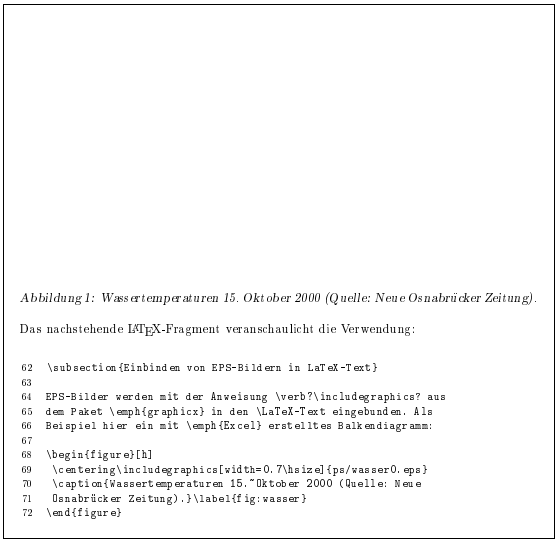


Abbildung 1: Wassertemperaturen 15. Oktober 2000 (Quelle: Neue Osnabrücker Zeitung).

Das nachstehende LaTeX-Fragment veranschaulicht die Verwendung:

```
62  \subsection{Einbinden von EPS-Bildern in LaTeX-Text}
63
64  EPS-Bilder werden mit der Anweisung \verb?\includegraphics? aus
65  dem Paket \emph{graphicx} in den \LaTeX-Text eingebunden. Als
66  Beispiel hier ein mit \emph{Excel} erstelltes Balkendiagramm:
67
68  \begin{figure}[h]
69   \centering\includegraphics[width=0.7\hsize]{ps/wasser0.eps}
70   \caption{Wassertemperaturen 15.~Oktober 2000 (Quelle: Neue
71   Osnabrücker Zeitung).}\label{fig:wasser}
72  \end{figure}
```

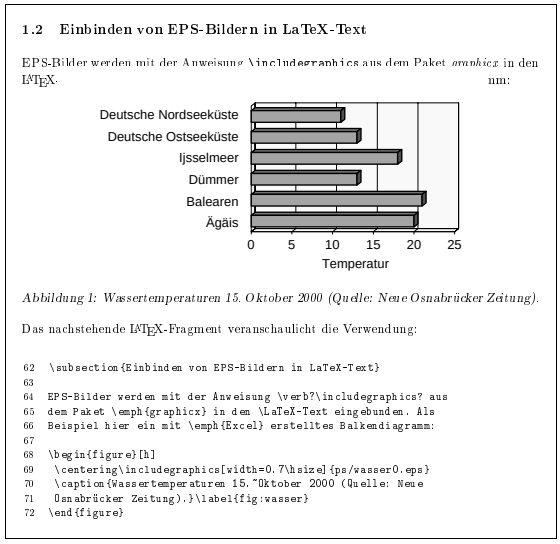


Abbildung 1: Wassertemperaturen 15. Oktober 2000 (Quelle: Neue Osnabrücker Zeitung).

Das nachstehende LaTeX-Fragment veranschaulicht die Verwendung:

```
62  \subsection{Einbinden von EPS-Bildern in LaTeX-Text}
63
64  EPS-Bilder werden mit der Anweisung \verb?\includegraphics? aus
65  dem Paket \emph{graphicx} in den \LaTeX-Text eingebunden. Als
66  Beispiel hier ein mit \emph{Excel} erstelltes Balkendiagramm:
67
68  \begin{figure}[h]
69   \centering\includegraphics[width=0.7\hsize]{ps/wasser0.eps}
70   \caption{Wassertemperaturen 15.~Oktober 2000 (Quelle: Neue
71   Osnabrücker Zeitung).}\label{fig:wasser}
72  \end{figure}
```

Abbildung 2: Probleme bei EPS-Dateien, die aus PS-Dateien gewandelt wurden. Links: Unter Windows für einen HP-Drucker erstelltes PS, rechts: für einen Apple Laserwriter erstelltes PS.

```
183  }featurecleanup
```

oder – bei älteren PostScript-Treibern (Windows NT):

```
78  [{
79  %%BeginFeature: *Duplex None
80    <</Duplex false>> setpagedevice
81  %%EndFeature
82  } stopped cleartomark
```

Der Drucker wird damit auf 'einseitig' eingestellt. Mit der Funktion `setpagedevice` ist jeweils ein implizites *Reset* der Graphikinhalte in der PostScript-Maschine verbunden. Alle derartigen Sequenzen müssen daher in einer EPS-Datei entfernt werden, `featurebegin` ... `featurecleanup` bzw. `[` ... `cleartomark` bilden jeweils die 'Klammern' der Sequenzen[4].

Im rechten Teilbild von Abbildung 2 ist zwar die Graphik an der richtigen Stelle eingebaut, ein Teil des davor stehenden Inhalts ist aber weiß überdeckt. Das Problem wird meist durch eine Sequenz der Form

```
1.000 1.000 1.000 sco 0 0 2479 3508 rf
```

verursacht, die als Basis der Seite eine weiße (`1 1 1 sco`) Rechteckfläche (`x y dx dy rf`) füllt. `sco` und `rf` sind übliche Kürzel für die PostScript-Anweisungen *setcolor* und *rectfill*,

[4] Die Klammerung hat den Zweck, die Sequenzen robust zu machen. Die schließende 'Klammer' ist jeweils eine Anweisung, die dafür sorgt, dass der PostScript-Interpreter (Stack und Fehlerzustand) auf den Zustand vor der öffnenden gesetzt wird.

die zuvor in der PS-Datei definiert werden. Auch in solchen Fällen entfernt man die komplette Sequenz in der EPS-Datei.

Ein weiteres bisweilen auftretendes Problem stellt die Drehung einer PostScript-Graphik dar, beispielsweise dann, wenn die Graphik im Querformat ausgegeben wurde, aber in ein Dokument im Hochformat eingebunden werden soll. Es genügt dann nicht – wie in den

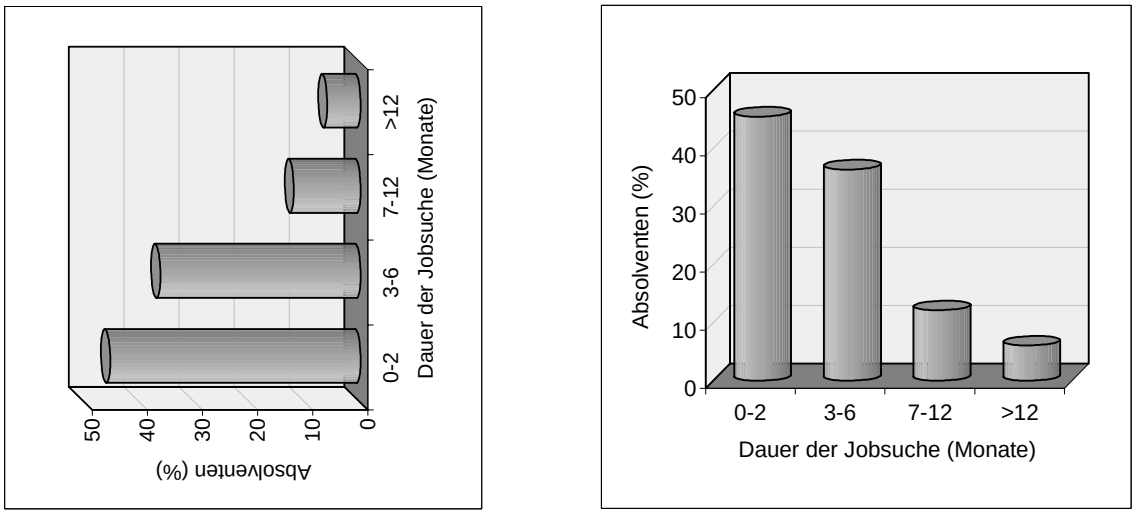


Abbildung 3: Drehung einer PostScript-Graphik um 90°. Links: Original – im Querformat gespeichert, rechts: gedreht.

vorstehenden Beispielen –, bestimmte Sequenzen aus der PS-Datei zu entfernen. Es sind zwei einfache zusätzliche PostScript-Anweisungen einzufügen, etwa wie in dem folgenden Beispiel die beiden Zeilen 129 und 140 vor der Zeile `%%BeginPageSetup`:

```
139  0 800 translate
140  -90 rotate
141  %%BeginPageSetup
```

Zeile 140 rotiert die Graphik um 90°, Zeile 139 bewirkt eine Translation zurück in die Papierfläche. Die Wirkung ist in Abbildung 3 dargestellt. Links das Originalbild, in Querformat gespeichert, rechts das gedrehte Bild mit den beiden eingefügten Zeilen[5].

Die Möglichkeit, graphische Elemente auf diese Weise zu verdrehen, kann auch dazu genutzt werden, aus vorhandenen Objekten graphische Muster zu generieren. Ein Beispiel zeigt Abbildung 4, das Unilogo in neuer Verwendung.

[5] Statt die Drehung im PostScript-Code zu machen, kann man auch die `angle`-Option von `\includegraphics` verwenden (vgl. Dokumentation zum Graphik-Paket in LaTeX).

Abbildung 4: Graphisches Muster aus verdrehten EPS-Bildern (Drehwinkel: $\pm 35°$).

1.4 PostScript, Konzepte

Wenn man mehr als die bisher beschriebenen 'Patches' in PostScript machen will, sollte man sich näher mit den Eigenschaften der Sprache vertraut machen. Hier ist das natürlich nur sehr oberflächlich zu leisten, für einen vertiefteren Einstieg sei auf die in Abschnitt 1.1 genannte Literatur verwiesen.

1.4.1 Interpreter-Sprache

PostScript wird im Drucker oder auf dem Rechner interpretiert, das heißt, die Anweisungen werden in der Reihenfolge abgearbeitet, in der sie in der PS-Datei niedergelegt sind. Funktionen, Parameter etc. können vorab definiert und zur späteren Verwendung im Speicher abgelegt werden. Die interpretierende Arbeitsweise hat zur Folge, dass Syntax-Fehler erst zur Laufzeit entdeckt werden und nicht wie gewohnt in der Kompilierungsphase.

PostScript kann in verschiedenen Modi verwendet werden, in denen jeweils unterschiedlich umfangreiche Möglichkeiten der Sprache genutzt werden können. Bei der Ausgabe auf den **Drucker** muss die Datei oder zumindest eine einzelne Seite vollständig sein, ein Zugriff auf das Dateisystem des Rechners ist nicht möglich, Nebenbedingungen wie Speichergröße sind zu beachten. Bei der Interpretation durch ***GhostScript*** können PS-Anweisungen verwendet werden, die auf das Dateisystem zugreifen, d. h. man kann beispielsweise Daten einlesen und durch eine PS-Funktion am Bildschirm darstellen. Arbeitet man **interaktiv** mit *GhostScript* kann man sich darüber hinaus Informationen ausgeben lassen.

1.4.2 Stack, Postfix-Notation

Die Sprache ist vollständig stackorientiert, es gibt daneben keine 'Register' zum Zwischenspeichern von Variablen. Damit verbunden ist eine strikte Postfix-Notation, wie man sie von alten Taschenrechnern bestimmter Firmen gewohnt ist[6]. Bei dieser Art der Programmierung werden bei jeder Anweisung zunächst die Parameter oder Operanden in der richtigen Reihenfolge auf den Stack gelegt, dann erfolgt der Funktionsaufruf oder Operator. Danach enthält der Stack das Ergebnis der Operation.

Beispiel: $\sin(\frac{2x}{3})$ $\Rightarrow$ `2 x mul 3 div sin.`

[6] Ähnlich ist PostScript in diesem Punkt der Programmiersprache *Forth*. Ebenfalls mit Postfix-Notation arbeitet der BiBTEX-Interpreter.

Die Postfix-Notation erleichtert die Arbeit der Interpreter-Maschine beträchtlich, da der Parser jeweils nur ein Objekt (*token*) gleichzeitig interpretieren muss.

1.4.3 PostScript-Syntax

Eine PostScript-Datei besteht aus einer Folge von Einzelobjekten (*token*), die voneinander durch Zwischenräume (*white space*) getrennt sind. Als Zwischenräume werden u. a. Leerzeichen, Tabulator und Zeilenende interpretiert. Folgen mehrere dieser Zeichen aufeinander, werden sie zu einem Trenner zusammengefasst. Zeichen mit Sonderbedeutung sind alle Klammern sowie Schrägstrich (*slash*) und Prozentzeichen. Klammern kennzeichnen bestimmte Datentypen wie Felder (`[...]`), Zeichenketten (`(...)` und `<...>`) und Funktionsblöcke (`{...}`). Der Schrägstrich veranlasst, dass der Wortlaut eines Symbols an dieser Stelle verwendet wird, nicht, wie sonst üblich, seine Bedeutung. Das Prozentzeichen beginnt, wie in TEX, einen Kommentar.

Abhängig vom Typ des Einzelobjekts entscheidet der Interpreter, was damit geschieht. Direkt angegebene Variable wie Zahlen, Felder oder Zeichenketten werden auf den Stack gelegt, Symbole (Namen) ohne Schrägstrich davor werden im *dictionary* nachgeschlagen (s. u.), Symbole mit Schrägstrich im Wortlaut auf den Stack gelegt. Bei den im *dictionary* nachgeschlagenen Symbolen kann es sich um Variable, Funktionen oder Operatoren handeln, je nach Typ wird die zugehörige Aktion veranlasst.

1.4.4 Dictionaries

Neben dem oben beschriebenen Arbeitsstack verwaltet die PostScript-Maschine einen Teil des Speichers als *dictionary stack*. In diesem logischen 'Nachschlagewerk' sind zunächst alle in der Sprache vordefinierten Funktionen und Operatoren vermerkt. Während der Interpretation einer PS-Datei werden dann alle dort definierten Funktionen und Variablen zusätzlich in den *dictionary stack* übernommen. Bei einem Drucker ist der Teil mit den vordefinierten Funktionen als ROM, der Rest als RAM ausgeführt. Wird im Ausführungsteil der PS-Datei eine Funktion (`mul`) oder Variable (`x`) angefordert, so wird deren Bedeutung bzw. Inhalt in den *dictionaries* nachgeschlagen.

Neben den üblichen Funktionen und Variablen werden auch Fonts – fest eingebaute und nachgeladene – im *dictionary stack* abgelegt. Das Setzen eines Buchstabens entspricht einem Funktionsaufruf.

1.4.5 Graphik-Modell

Das in PostScript verwendete Graphik-Modell entspricht in etwa dem, das man sich intuitiv überlegen würde, um ein Bild zu generieren. Es lässt sich durch die folgenden Begriffe beschreiben:

Current Page: Es wird jeweils nur eine Seite bearbeitet, die *current page*. Erst wenn diese Seite gedruckt ist, folgt die nächste.

Current Path: Ein Linienzug wird zunächst zwischengespeichert, erst wenn der Linienzug vollständig definiert ist, wird entschieden, was damit gemacht wird. So kann beispielsweise eine Polylinie geschlossen werden, bevor sie gezeichnet wird, der Linienzug für eine gefüllte Fläche kann mit den gleichen Anweisungen wie für eine umrandete erstellt werden.

Clipping Path: Ein spezieller Linienzug, der bestimmt, in welchen Bereichen der Seite wirklich 'Farbe' aufgetragen wird. Alle Graphik-Objekte werden entsprechend dem aktuell gültigen *clipping path* zugeschnitten.

1.4.6 Graphik-Objekte in PostScript

Mit den in PostScript verfügbaren Anweisungen können drei Kategorien von graphischen Objekten dargestellt werden: Texte, geometrische Figuren und Pixel-Bilder. Alle diese Objekte lassen sich beliebig skalieren, rotieren, verschieben oder verzerren.

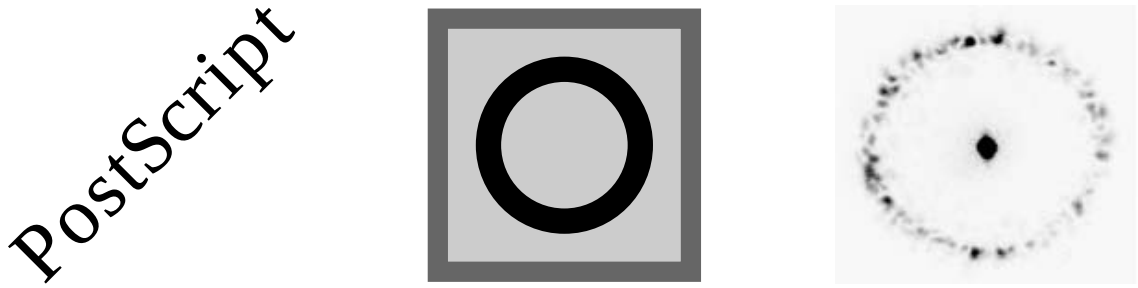


Abbildung 5: Die drei Kategorien von Graphik-Objekten in PostScript: Text, geometrische Figuren, Pixel-Bilder.

Für Texte verfügt jede PostScript-Maschine über mehrere eingebaute Fonts, zusätzliche Fonts können von Anwendungen definiert werden. Geometrische Figuren werden mit Anweisungen für Linien und Kurven konstruiert, sie können als Linienzug mit einstellbarer Linienbreite ausgeführt oder auch ausgefüllt werden. Pixel-Bilder werden als rechteckiger Datenblock gespeichert, die Pixel-Zahl bleibt bei Transformationen ungeändert. Die Vektorobjekte – Texte und geometrische Figuren – werden immer optimal, d. h. mit der Druckerauflösung dargestellt, Pixel-Objekte dagegen vergröbern sich scheinbar bei Vergrößerung.

1.4.7 Koordinatensystem

PostScript verwendet als Basissystem ein kartesisches Koordinatensystem, das den üblichen Darstellungsgewohnheiten entspricht. Die X-Werte (erste Koordinate bei Koordinatenpaa-

ren) werden nach rechts, die Y-Werte(zweite Koordinate) nach oben gemessen. Die verwendeten Maßeinheiten sind Points (1/72 Zoll), sie entsprechen der Auflösung der ersten Matrixdrucker. Der Nullpunkt des Koordinatensystems ist die linke untere Seitenecke. In diesem Basiskoordinatensystem wird z. B. die *BoundingBox* angegeben.

Aus diesem Basiskoordinatensystem können durch geeignete Transformationen (Translation, Rotation, isotrope oder anisotrope Skalierung) beliebige andere Systeme entwickelt werden, um die Entwicklung eines konkreten Graphik-Projekts zu vereinfachen. So kann man nach einer geeignete Skalierung Koordinaten etwa in Millimetern oder Druckerpixeln angeben oder durch Skalierung plus Translation zu Bildschirm-üblichen Koordinaten (Nullpunkt links oben) übergehen (Genaueres dazu in Abschnitt 1.5.8).

1.4.8 Struktur einer PostScript-Datei

Eine PostScript-Datei besteht in der Regel aus mehreren Teilen mit unterschiedlicher Funktion. Beginn und Ende dieser Abschnitte sind durch Pseudokommentare gekennzeichnet (Kommentare beginnen in PostScript wie in TEX mit `%`, Kommentare mit Spezialbedeutung mit `%%`). Meist findet man die Abfolge:

```
%!PS-Adobe-3.0
%%Pages:
%%BoundingBox:
%%EndComments
%%BeginProlog
%%EndProlog
%%BeginSetup
%%EndSetup
%%Page:
%%BeginPageSetup
%%EndPageSetup
%%PageTrailer
%%Trailer
%%EOF
```

Die erste Zeile ist ein spezieller Pseudokommentar, der durch `%!` den Beginn eines PS-Dokuments kennzeichnet, weiterhin ist die Sprachversion vermerkt. Diese vollständige Kennzeichnung sollte aber nur dann verwendet werden, wenn das PS-Dokument strikt der Sprachdefinition und den Gliederungsvorschriften (*Document Structuring Conventions*) von Adobe entspricht. Selbstgeschriebene PS-Dateien sollten sinnvollerweise nur mit `%!` beginnen[7].

`%%BoundingBox:` gefolgt von 4 Integerzahlen informiert über die Graphik-Größe, kann bei PS-Dateien wegfallen, ist bei EPS-Dateien notwendig. Die 4 Integers sind x und y der

[7] *GhostScript* kommt sogar ohne diese Anfangskennung aus, da es nichts anderes als PostScript 'erwartet'. Bei einfachen selbstgeschriebenen PS-Dateien kann man so vorgehen, dass man keine Pseudokommentare verwendet und einige notwendige von *GhostScript* bei der Wandlung nach EPS hinzufügen lässt.

linken unteren sowie x und y der rechten oberen Ecke des begrenzenden Rechtecks. Diese Koordinaten werden in PostScript-Einheiten (1/72 Zoll) von der linken unteren Seitenecke aus gemessen.

Der `Prolog` enthält Definitionen aller Art: Funktionen, die später verwendet werden sollen, Abkürzungen, Variable, Konstanten. `%%EndProlog` trennt den Definitionsteil vom eigentlichen Skript, dem direkt auszuführenden Teil.

Der Skript-Teil beginnt mit globalen Initialisierungen (`Setup`), gefolgt von den einzelnen Seiten des Dokuments, die jeweils mit einem `%%Page:`-Kommentar beginnen. Die einzelnen Seiten können wieder seitenspezifische Initialisierungen enthalten (`PageSetup`).

`%%PageTrailer` und `%%Trailer` markieren das Ende der Seite bzw. des Gesamtdokument, danach können jeweils noch 'Aufräumarbeiten' erfolgen.

Änderungen, insbesondere auch Erweiterungen, von PostScript-Dateien sollten jeweils nur in den richtigen, dafür zuständigen Abschnitten der Datei erfolgen. Die Gliederung der Datei gibt auch Anhaltspunkte dafür, wo Probleme auftreten können. So findet man beispielsweise die oben beschriebenen `setpagedevice`-Aufrufe im `Setup`-Bereich.

1.4.9 Die *BoundingBox*

LaTeX liest eine EPS-Datei genau bis zum `%%BoundingBox:`-Kommentar, um die Originalgröße der Graphik festzustellen. Basierend auf dieser Größe wird die Graphik dann von LaTeX skaliert. Einfacherweise läßt man sich die *BoundingBox* von *GhostView* richtig eintragen. Die so bestimmte *BoundingBox* ist meist etwas zu groß, da *GhostView* offensichtlich eine kleine Sicherheitsmarge berücksichtigt. Nachträgliche Änderungen sind möglich, manchmal auch nötig, um ein Feintuning durchzuführen, beispielsweise wenn zwei ähnliche Graphiken genau nebeneinander in ein Dokument eingefügt werden sollen. Beim manuellen Eintrag ist auf jeden Fall darauf zu achten, dass Integer-Werte eingetragen werden, mit Real-Werten kommen Folgeprogramme nicht zurecht.

Die Werte für die *BoundingBox* können auch (teilweise) negativ sein, die meisten Folgeprogramme kommen damit zurecht. *GhostView* kann solche negativen Werte jedoch nicht bei der Wandlung zu EPS generieren, da dort nur in den Papierbereich gezeichnet wird. Bei der Bestimmung der *BoundingBox* über *GhostView* ist mithin darauf zu achten, dass der gewünschte Bereich nicht über die Seite hinausragt.

1.5 PostScript als Seitenbeschreibungssprache

Die Hauptzielrichtung von PostScript ist die Beschreibung von Objekten, die insgesamt eine Druckseite ausmachen. Ältere Druckersprachen waren meist auf Teilaspekte der Seitenbeschreibung beschränkt; so ist HPGL[8] eine rein vektororientierte Sprache, ursprünglich

[8] *H*ewlett *P*ackard *G*raphics *L*anguage.

für Plotter gedacht, ein von Epson entwickeltes Konzept umfasste nur einige wenige Kurzbefehle für Matrixdrucker. In PostScript können sowohl vektorisierte wie auch Pixel-Objekte formuliert werden. Naturgemäß ist der dazu nötige Befehlsvorrat sehr umfangreich. Er wird darüber hinaus noch ständig erweitert, um neuere Möglichkeiten bei Druckern zu integrieren. So wurde beispielsweise bei der Definition von Level 2 PostScript um die Möglichkeit der Farbdarstellung erweitert. Im Folgenden ein kleiner, sehr unvollständiger Auszug aus den Möglichkeiten, die PostScript bietet.

1.5.1 Linien

Linien werden zunächst als *current path* zwischengespeichert, dann erst vollständig gezeichnet. Unter anderem sind die folgenden Anweisungen zuständig (in Klammern die Zahl der Parameter, die vorher auf den Stack gelegt wird):

moveto (2): Bewegt den Zeichenstift zu einer absoluten Position (`x y moveto`).

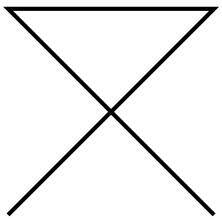
rmoveto (2): Relativbewegung (`dx dy rmoveto`).

lineto (2): Zieht eine Linie ausgehend von der aktuellen Position, absolute Koordinaten (`x y lineto`).

rlineto (2): Relative Koordinaten (`dx dy lineto`).

stroke (0): Zeichnet den *current path*.

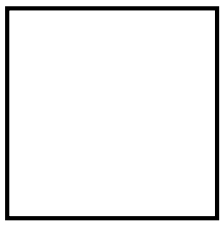
Ein Beispiel zeigt Abbildung 6.



```
100 100 moveto
50 50 rlineto
-50 0 rlineto
50 -50 rlineto
stroke
```

Abbildung 6: Einfacher Linienzug und zugehöriger PostScript-Code.

Um Linienzüge zu schließen, benutzt man **closepath**, das dafür sorgt, dass die Anschlüsse passen (Abbildung 7).



```
100 100 moveto
50 0 rlineto
0 50 rlineto
-50 0 rlineto
closepath
stroke
```

Abbildung 7: Geschlossener Linienzug und zugehöriger PostScript-Code.

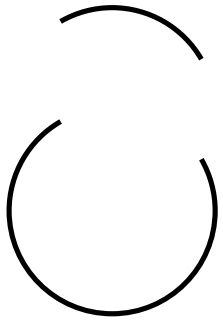
Üblicherweise beginnt man Linienzüge mit **newpath**, das den etwa noch vorhandenen Inhalt des *current path* löscht.

1.5.2 Kurven

Kreisbögen werden mit den folgenden Befehlen gezeichnet:

arc (5): Zeichnet einen Kreisbogen gegen den Uhrzeigersinn, die Parameter sind Mittelpunkt (2), Radius, Anfangs- und Endwinkel (Abbildung 8).

arcn (5): Im Uhrzeigersinn (Abbildung 9).

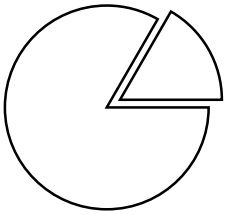


```
100 100 20 30 120 arc
stroke
```

Abbildung 8: Kreisbogen gegen den Uhrzeigersinn von 30 bis 120 Grad.


```
100 100 20 30 120 arcn
stroke
```

Abbildung 9: Kreisbogen im Uhrzeigersinn von 30 bis 120 Grad.



```
100 100 moveto
100 100 40 0 60 arcn
closepath
stroke
5.2 3 translate
100 100 moveto
100 100 40 0 60 arc
closepath
stroke
```

Abbildung 10: Einfaches Tortendiagramm.

1.5.3 Datentypen

In den vorstehenden Beispielen wurden nur Integer-Variable verwendet, dies ist der einfachste Datentyp, den PostScript kennt. Statt der Integer-Variablen können praktisch überall auch reelle Zahlen verwendet werden (wichtige Ausnahme: *BoundingBox*). Weitere wichtige Datentypen:

Felder (arrays): In Feldern können Variable zusammengefasst werden, sie werden definiert durch eckige Klammern. Felder können Variable unterschiedlichen Typs enthalten.

Zeichenketten (strings): In runden Klammern eingeschlossene Folge von Zeichen. In einer solchen Zeichenkette wird das *backslash*-Zeichen (\) verwendet, um Sonderzeichen darzustellen (ähnlich wie in C). Wichtig ist insbesondere die Oktaldarstellung `\nnn` für Umlaute und ähnliches. Daneben können Zeichenketten hexadezimal kodiert werden, sie werden dann in spitze Klammern eingeschlossen (`< ... >`).

Namen (names): Zeichenfolge zur Kennzeichnung eines Objekts in PostScript. Solche Objekte können z. B. eingebaute oder benutzerdefinierte Funktionen oder Variable sein. Ein vorangestellter Schrägstrich (`/`) bedeutet, dass der Name an dieser Stelle als Symbol gemeint ist, ohne Schrägstrich wird der Inhalt, d. h. die Bedeutung eingesetzt (vgl. 1.5.7), die Definition wird in den aktuell eingestellten *Dictionaries* nachgeschlagen.

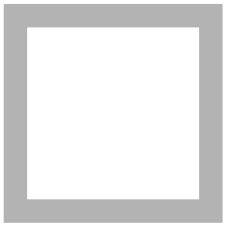
1.5.4 Farbe, Linienbreite

PostScript kann natürlich u. a. auch die Farbe und Breite von Linien einstellen.

setrgbcolor (3): Setzt die aktuellen Rot-, Grün- und Blauwerte. Die drei Parameter müssen Zahlen zwischen 0 und 1 sein.

setgray (1): Weist dem Zeichenstift einen Graustufenwert zu, der Parameter liegt zwischen 0 (Schwarz) und 1 (Weiß).

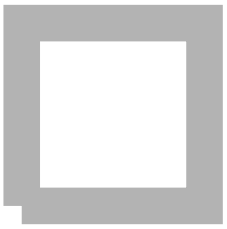
setlinewidth (1): Setzt die Linienbreite in den gerade aktuellen Koordinaten fest.



```
100 100 moveto
50 0 rlineto
0 50 rlineto
-50 0 rlineto
0.7 setgray
6 setlinewidth
closepath
stroke
```

Abbildung 11: Geschlossener Linienzug mit breiten, grauen Linien, PostScript-Code.

Abbildung 12 verdeutlicht die Wirkung von `closepath`, die insbesondere bei größeren Linienbreiten merkbar wird.



```
100 100 moveto
50 0 rlineto
0 50 rlineto
-50 0 rlineto
0 -50 rlineto
0.7 setgray
10 setlinewidth
stroke
```

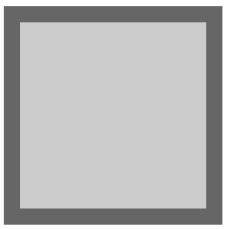
Abbildung 12: Ein Beispiel ohne `closepath`.

1.5.5 Graphik-Status

Der aktuelle Graphik-Status kann mit `gsave` zwischengespeichert, mit `grestore` wieder in den vor `gsave` vorhandenen Zustand zurückgesetzt werden. Das wird beispielsweise nötig, wenn der *current path* mehrfach 'verbraucht' werden soll.

1.5.6 Flächen

Gefüllte Flächen werden mit der Anweisung `fill` erzeugt, dabei wird die durch den *current path* umrandete Fläche mit der aktuellen Farbe bzw. dem aktuellen Graustufenwert ausgefüllt (Abbildung 13).



```
100 100 moveto
50 0 rlineto
0 50 rlineto
-50 0 rlineto
closepath
gsave
0.8 setgray
fill
grestore
4 setlinewidth
0.4 setgray
stroke
```

Abbildung 13: Quadrat, diesmal gefüllt und gerahmt.

1.5.7 Funktionen und Variable

Funktionen und Variable werden im Definitionsteil (Prolog) einer PS-Datei zur späteren Verwendung definiert. Sie können prinzipiell auch überall innerhalb des Ausführungsteils definiert werden, das entspricht jedoch nicht dem von Adobe vorgesehenen sauberen Programmierstil[9]. Die einzige Einschränkung ist, dass definierte Objekte vor ihrer Verwendung vollständig bekannt sein müssen (der Interpreter macht nur einen Durchgang durch den Code). Definiert wird mit dem Operator `def`, Variable `size` und `start` etwa mit:

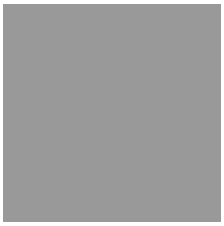
```
4  /size 50 def
5  /start size neg 2 div def
```

Der Funktionsblock einer definierten Funktion wird in geschweifte Klammern eingeschlossen, ein gefülltes Quadrat würde etwa so definiert:

```
7  /box {
8  0 0 moveto
9  start start rmoveto
10 0 size rlineto
11 size 0 rlineto
12 0 size neg rlineto
13 closepath
14 fill
15 } def
```

[9] Wenn im Ausführungsteil noch umfangreiche Definitionen durchgeführt werden, kann das für das Speichermanagement problematisch werden.

Die Operatoren `neg` (Negation des Stackinhalts) und `div` (Division) gehören zu den arithmetischen Operatoren, die in PostScript verfügbar sind, mehr dazu in 1.6. Ihre Funktion hier ist leicht nachzuvollziehen.



```
0.6 setgray
box
```

Abbildung 14: Verwendung der vorher definierten Funktion `box`.

Von Automaten erstellte PostScript-Dokumente zeichnen sich fast immer durch umfangreiche Definitionsblöcke am Dateianfang aus. Das macht die Dateien sehr unübersichtlich und weitgehend unleserlich. Die meisten dieser Definitionen sind Abkürzungen für PostScript-Befehle. So werden für viele PostScript-Kommandos Ein- oder Zwei-Buchstaben-Kürzel definiert, um Platz zu sparen. Meist sind die Definitionen nicht ganz einfach zu erkennen, da zuallererst eine Abkürzung für `def` definiert wird.

1.5.8 Koordinatentransformation

Zur Manipulation von graphischen Objekten können in PostScript beliebige zweidimensionale Koordinatentransformationen programmiert werden. In einfachen Fällen werden sie aus Translationen, Rotationen und Skalierungen zusammengesetzt, die nacheinander ausgeführt werden. Die Anweisungen dazu lauten:

translate (2): Translation in Einheiten des gerade gültigen Koordinatensystems (`tx ty translate`).

rotate (1): Rotation um einen Winkel (gemessen in Grad) gegen den Uhrzeigersinn (`phi rotate`).

scale (2): Skalierung in x- und y-Richtung (`sx sy scale`). Bei isotroper (winkeltreuer) Skalierung sind die beiden Skalierungsfaktoren gleich, im allgemeinen Fall ungleich[10].

Die folgenden Abbildungen 15 und 16 illustrieren die Anwendung der Transformationsbefehle auf das in 1.5.7 beschriebene `box`-Objekt. Abbildung 17 zeigt, wie durch anisotrope Skalierung eine Ellipse gezeichnet werden kann.

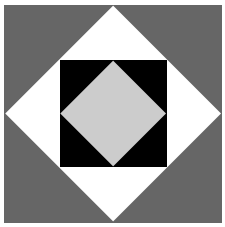
[10] Eine andere, sehr anschauliche Möglichkeit der Skalierung ist es, Makros mit den Namen von Maßeinheiten zu definieren und diese dann bei jeder Längenangabe zu verwenden. Um Maße in Millimetern anzugeben, definiert man

```
/mm { 72 mul 25.4 div } def
```

und kann dann Linien zeichnen mit

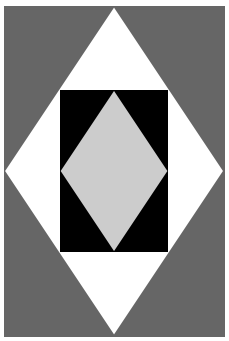
```
15 mm 30 mm rlineto.
```

In solchen Fällen sollte dann keine globale Skalierung angeordnet werden.



```
200 200 translate
0.4 setgray
box
0.7 0.7 scale
45 rotate
1 setgray
box
0.7 0.7 scale
45 rotate
0 setgray
box
0.7 0.7 scale
45 rotate
0.8 setgray
box
```

Abbildung 15: Translation, isotrope Skalierung und Rotation eines Objekts.



```
200 200 translate
2 3 scale
0.4 setgray
box
0.7 0.7 scale
45 rotate
%........
```

Abbildung 16: Anisotrope Skalierung (`2 3 scale`).

```
4 2 scale
40 40 20 0 360 arc
closepath
0.4 setgray
5 setlinewidth
stroke
```

Abbildung 17: Ellipse durch Anisotrope Skalierung (`4 2 scale`).

Der jeweils aktuelle Transformationsstatus von PostScript ist in der *current matrix* zusammengefasst. Diese kann mit `currentmatrix` erfragt, mit `setmatrix` gesetzt werden. Mit `concat` kann die *current matrix* mit einer allgemeinen zweidimensionalen Transformationsmatrix multipliziert werden. Damit können komplexere Transformationen in einer Anweisung definiert werden. Die Syntax des `concat`-Befehls lautet:

`[a b c d t1 t2] concat`,

die sechs Elemente des Parameterfeldes sind definiert durch die allgemeinen Transformationsgleichungen

$$\begin{aligned} x' &= ax + cy + t1 \\ y' &= bx + dy + t2\,, \end{aligned}$$

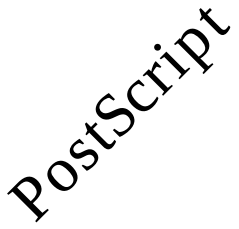
in Matrixschreibweise

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & t1 \\ b & d & t2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} .$$

Mehrfache Transformationen dieser Art können durch die Multiplikation der Transformationsmatrizen aneinandergehängt werden. Sie speziellen Transformationen Translation, Rotation, Skalierung sind durch Sonderfälle der Transformationsmatrix zu definieren, so ist bei einer reinen Translation $a = d = 1$ und $b = c = 0$, bei einer Skalierung $b = c = t1 = t2 = 0$, bei einer Rotation ist $t1 = t2 = 0$ und a, b, c, d ist eine reine Drehmatrix (Sinus- und Cosinuswerte des Drehwinkels).

1.5.9 Text

Text wird in PostScript mit der Anweisung `show` gesetzt, als Parameter wird nur der Text als Zeichenkette übergeben. Font, Zeichengröße, Position und Richtung des Textes müssen vorher festgelegt werden. Die übliche Befehlsabfolge zeigt Abbildung 18.




```
/Palatino-Roman findfont
100 scalefont
setfont
100 100 moveto
45 rotate
(PostScript) show
```

Abbildung 18: Textdarstellung in PostScript.

Die wichtigsten der klassischen PostScript-Fonts sind in Abbildung 19 zusammengestellt.

Times-Roman	Helvetica
Times-Italic	*Helvetica-Oblique*
Times-Bold	**Helvetica-Bold**
Times-BoldItalic	***Helvetica-BoldOblique***
`Courier`	αβγ πφσ ± ∏ ∑ ∞ ♣♠♥♦
`Courier-Bold`	⇒ ⇐ → ← ⌈ ⌊ ↵ ≡ ≠ ≅ ∇

Abbildung 19: Die wichtigsten Font-Familien in PostScript. Der Symbol-Font rechts unten ist nur aufrecht und in normaler Stärke verfügbar.

Englischer Text macht generell keine Schwierigkeiten, dafür ist PostScript gemacht. Benötigt man Umlaute, fängt es an, komplizierter zu werden. Umlaute sind in den PostScript-Zeichensätzen vorhanden, aber über die normale Zeichensatz-Kodierung (*Standard Encoding*) nicht zugänglich. Man benötigt dazu die Kodierung *ISOLatin1*. Da die Zeichen

vorhanden sind, kann man einen der verfügbaren Zeichensätze kopieren, mit der *ISOLatin1Encoding* versehen und mit einem neuen Namen benennen. Das Reference Manual [2] macht dazu folgenden Vorschlag, den man einfach sinngemäß übernehmen kann:

```
/Helvetica-Bold findfont
dup length dict begin
 {1 index /FID ne {def} {pop pop} ifelse} forall
 /Encoding ISOLatin1Encoding def
 currentdict
end
/Helvetica-ISO-Bold exch definefont pop
```

Damit lassen sich nun auch Umlaute verwenden (Abbildung 20).

Abbildung 20: Verwendung von Umlauten und verschiedene Schriftmanipulationen.

Der PostScript-Code für Abbildung 20 lautet:

```
13  /S (OSNABR\334CK) def
14  /Helvetica-ISO-Bold findfont
15  72 scalefont
16  setfont
17  15 200 translate
18  gsave
19  [1 0 2 1.2 0 0] concat
20  0.5 setgray
21  0 0 moveto
22  S show
23  grestore
24  1 setgray
25  0 0 moveto
26  S show
27  0 setgray
28  0 0 moveto
29  S false charpath
30  2 setlinewidth
31  stroke
```

Zunächst wird die Zeichenkette `OSNABRÜCK` definiert, da sie mehrfach verwendet werden soll, sodann der zuvor neu erstellte Fontsatz gewählt. Mit der windschiefen Transformationsmatrix (Zeile 27) wird dann die Schattenschrift erstellt, danach der Schriftzug in weiß (`1 setgray`) darübergelegt. Schließlich wird die Umrandung der Buchstaben in Schwarz

(`0 setgray`) nachgezogen, dazu muss mit `S false charpath`[11] der zugehörige *path* extrahiert werden.

1.5.10 PostScript-Code in TEX-Texten

Mithilfe von DVIPS lassen sich PostScript-Sequenzen wörtlich aus einer TEX- oder LATEX-Quelldatei in das PostScript-Dokument übernehmen. Das kann für einfache Graphiken, Logos, Wasserzeichen oder ähnliches sinnvoll sein. Zuständig für derartige Aktionen ist in TEX immer die `\special`-Anweisung. Sie sorgt dafür, dass der darin verpackte Inhalt von TEX nicht interpretiert sondern wörtlich als Anweisung für den DVI-Treiber in die DVI-Datei übernommen wird. DVIPS kann PostScript-Sequenzen auf zwei Arten einfügen[12]: Mit `\special{ps: XYZ}` wird `XYZ` ungeschützt, d. h. ohne Vor- und Nachspann in die Zieldatei eingefügt, mit `\special{" XYZ}` dagegen geschützt, umrahmt von Anweisungen, die den aktuellen Zustand der PostScript-Maschine retten und wiederherstellen.

Im ersten Fall sollte man genau wissen, was man tut, ein `\special{ps: 0.6 setgray}` wirkt sich direkt auf den folgenden Text aus, bis es mit `\special{ps: 0 setgray}` wieder rückgängig gemacht wird.

Das ‘Anführungszeichen-Special’ hat dagegen etwas andere Probleme: Arbeitet man mit der Sprachoption *german* wie im vorliegenden Skript, so hat das doppelte Anführungszeichen eine Spezialbedeutung, die auch innerhalb des `\special`-Befehls eingesetzt wird. In diesem Fall sollte man ein Makro im Vorspann des LATEX-Dokuments definieren, etwa

```
\newcommand\IncludePS[1]{\special{" #1}},
```

das dann zum Einfügen von PostScript verwendet wird.

Einfache Graphiken, Logos baut man innerhalb einer `picture`-Umgebung ein, mit der man sich den notwendigen Platz verschafft. Abbildung 21 zeigt ein Beispiel: Die `picture`-Sequenz ist in eine `minipage`-Umgebung verpackt. Diese wird in einer `figure`-Umgebung angeordnet, zusammen mit einer zweiten `minipage` mit dem Code, zwischen den beiden dehnbarer Zwischenraum (`\hfill`).

Bei so eingefügtem PostScript-Code wird der Nullpunkt des PostScript-Koordinatensystems an die gerade aktuelle Position der TEX-Maschine verschoben, in obigem Fall ist dies der Nullpunkt der `picture`-Umgebung. Die angegebenen PostScript-Koordinaten werden in Points relativ zur aktuellen TEX-Koordinate gemessen. In TEX heißt die entsprechende Maßeinheit *Big Points* (`bp`), diese Maßeinheit wurde im obigen Beispiel für die `picture`-Koordinaten und für die Größe der `minipage` gewählt, um auf einfache Weise vergleichbare Größenangaben machen zu können.

[11] Der zweite Parameter bei `charpath` legt fest, wofür der so generierte *path* verwendet wird: `true` für `fill` o. ä., `false` für `stroke` o. ä.

[12] Eine sehr detaillierte Beschreibung der in DVIPS verfügbaren Möglichkeiten findet sich in der zum LATEX-Paket gehörenden DVIPS-Dokumentation vom Thomas Rokicki.

```
\begin{minipage}{120bp}
\unitlength1bp
\begin{picture}(120,120)(0,0)
\IncludePS{70 70 35 0 360 arc closepath
  0.3 setgray 25 setlinewidth stroke
  0 0 moveto 70 0 rlineto
  0 70 rlineto -70 0 rlineto
  closepath 0.6 setgray fill}
\end{picture}
\end{minipage}
```

Abbildung 21: Innerhalb des LaTeX-Textes direkt durch PostScript-Anweisungen platzierte Graphik. Rechts der innerhalb einer `picture`*-Umgebung eingefügte PostScript-Code.*

Ein einfaches absolut positioniertes Wasserzeichen wie der Kreis auf der aktuellen Seite kann mit

```
\makeatletter
 \let\old@oddhead\@oddhead
 \def\@oddhead{\old@oddhead
  \IncludePS{matrix defaultmatrix setmatrix
   300 420 120 0 360 arc closepath
   30 setlinewidth 0.9 setgray stroke}
  \gdef\@oddhead{\old@oddhead}}
\makeatother
```

erzeugt werden.

Die etwas kompliziert aussehende Sequenz mit der Definition von `\@oddhead` sorgt dafür, dass die PostScript-Graphik an den Seitenkopf angehängt wird, das aber nur auf der aktuellen Seite, denn `\gdef...` stellt nach dem ersten Aufruf sofort wieder den alten Zustand her. Da der Seitenkopf am Beginn der Seite gesetzt wird, liegt die PostScript-Graphik auf jeden Fall unter dem Seitentext. Will man sie über den Text legen, muss sie auf entsprechende Weise an `\@oddfoot` angehängt werden[13].

Wichtig ist die Redewendung `matrix defaultmatrix setmatrix`[14]. Damit wird dafür gesorgt, dass als Koordinatensystem das Basiskoordinatensystem von PostScript verwendet wird (Points, Nullpunkt links unten auf der aktuellen Seite), die Koordinaten mithin nicht von der aktuellen TeX-Position aus gemessen werden. Diese Teilsequenz ist daher immer dann wichtig, wenn Graphiken auf der Seite absolut positioniert werden sollen.

[13] Das LaTeX-Paket *fancyhdr* bietet vereinfachte Möglichkeiten, Seitenkopf und Seitenfuß in LaTeX-Dokumenten individuell zu gestalten. Man sollte diese Möglichkeiten nutzen, wenn solche Manipulationen häufiger benötigt werden.

[14] Mit `matrix` wird eine leere Transformationsmatrix im Stack bereitgelegt, die mit `defaultmatrix` auf das Basiskoordinatensystem gesetzt wird. Dieses wird dann mit `setmatrix` eingestellt.

Soll das Wasserzeichen auf allen Seiten eines Dokuments erscheinen (durchgängige Kennzeichnung mit 'DRAFT', 'KOPIE', '<DATUM>' o. ä.), wird die PostScript-Sequenz schon im Vorspann des LaTeX-Dokuments an den Seitenkopf oder Seitenfuß gehängt, ohne dass wieder auf den alten Zustand umgeschaltet wird. Bei einseitigen Dokumenten (Briefkopf) ist die Vorgehensweise einfacher, die PostScript-Sequenz wird – der gewünschten Überdeckung entsprechend – an den Anfang oder das Ende des Dokuments gesetzt.

1.6 PostScript als Programmiersprache

Neben den Anweisungen zur reinen Beschreibung von graphischen Inhalten umfasst die Sprachdefinition von PostScript eine Fülle weiterer Operatoren und Funktionen, die PostScript zu einer vollständigen, leistungsfähigen Programmiersprache machen. Man kann dies nutzen, um geometrische Sachverhalte, die mathematisch zu beschreiben sind, konstruktiv zu programmieren.

1.6.1 Stack-Befehle

Verwendet man PostScript nur zur Seitenbeschreibung, so ist es fast nie erforderlich, im Stack Veränderungen vorzunehmen, da Variable im Stack abgelegt und danach sofort verwendet werden (`5 5 rlineto`). Bei komplexeren Abläufen haben die Variablen im Stack eine längere 'Lebensdauer', es kann nötig werden, ihre Reihenfolge zu verändern, sie zu duplizieren oder sie aus dem Stack zu entfernen. Dazu stellt PostScript eine Reihe von Befehlen bereit (in Klammern wieder die Zahl der Parameter):

pop (0) entfernt die oberste Variable aus dem Stack,

exch (0) vertauscht die beiden obersten Elemente,

clear (0) löscht den Stack,

dup (0) dupliziert die oberste Variable im Stack,

index (1) dupliziert ein beliebiges Element im Stack (`0 index` hat die Wirkung von `dup`),

copy (1) kopiert die angegebene Anzahl von Variablen im Stack,

roll (2) rotiert m Variable um n Plätze aufwärts (`m n roll`),

count (0) zählt die Elemente im Stack, das Ergebnis wird auf den Stack gelegt.

1.6.2 Arithmetische und mathematische Operatoren

Bei den arithmetischen und mathematischen Operatoren sind neben den Grundrechenarten für reelle Zahlen (`add`, `sub`, `mul`, `div`, `neg`, `abs`) und den Integer-Operationen `mod` und

`idiv` auch Winkel- (`sin`, `cos`, `atan` mit 2 Operanden für den Bereich 0...360°), Wurzel- (`sqrt`) sowie Logarithmus- (`ln` und `log`) und Exponentialfunktionen (`exp`) verfügbar. Außerdem gibt es alle notwendigen Rundungsfunktionen (`ceiling`, `floor`, `round`, `truncate`) und einen Zufallszahlengenerator (`rand`), der positive 32-Bit-Pseudozufallszahlen erzeugt.

1.6.3 Textausrichtung

In PostScript gibt es keine Kommandos, um Text in bestimmter Ausrichtung zu positionieren (linksbündig, rechtsbündig, zentriert). Das ist auch nicht notwendig, da davon ausgegangen wird, dass diese Funktion vom Programm erledigt wird, das den PostScript-Code erzeugt. Text wird immer linksbündig gesetzt, die linke untere Ecke des Texts wird durch die aktuelle Stiftposition definiert (`x y moveto` vor `(...) show`).

Will man Beschriftungen in selbstgeschriebenen PostScript-Sequenzen richtig positionieren, ist es jedoch oft notwendig, den Text anders als an der linken unteren Ecke auszurichten. Erwünscht sein können Texte rechts oder links neben, über oder unter anderen Objekten. Der Referenzpunkt, an dem der Text ausgerichtet werden soll, ist dann nicht mehr die linke untere Ecke sondern irgendeine andere Position im Textrahmen. Einige der Möglichkeiten sind in Abbildung 22 dargestellt.

Abbildung 22: In einem Rechteck unterschiedlich positionierter Text.

Die Texte werden durch Funktionen positioniert, denen außer dem Text der X- und Y-Wert des jeweiligen Referenzpunktes übergeben wird (die vier Ecken des Rechtecks bzw. bei `CC` der daraus berechnete Mittelpunkt):

```
65  0 setgray
66  (LinksUnten) xl yl LL
67  (LinksOben) xl yh UL
68  (RechtsOben) xh yh UR
69  (RechtsUnten) xh yl LR
70  (Zentriert) xl xh add 2 div yl yh add 2 div CC
```

Bei der Funktion `UR` ist der Referenzpunkt die rechte obere Ecke des Texts:

```
27  /UR {
28   GETSIZE
29   moveto
30   WIDTH neg HEIGHT neg rmoveto
31   show
32  } def
```

Nach einer Bewegung zum Referenzpunkt (`moveto`) wird um die Textbreite (`WIDTH`) nach links und um die Texthöhe (`HEIGHT`) nach unten positioniert (`rmoveto`), dann kann der Text mit `show` an der aktuellen Position gesetzt werden, richtig zur rechten oberen Ecke.

Bei der Funktion `CC` ist der Referenzpunkt die Mitte des Texts:

```
41 /CC {
42  GETSIZE
43  moveto
44  WIDTH 2 div neg HEIGHT 2 div neg rmoveto
45  show
46 } def
```

Relativ positioniert (`rmoveto`) wird hier um jeweils die Hälfte der Textbreite und -höhe.

Breite und Höhe des Textes werden jeweils durch die Funktion `GETSIZE` berechnet:

```
 5 /GETSIZE {
 6  newpath
 7  3 copy
 8  moveto
 9  false charpath pathbbox
10  3 2 roll
11  sub /HEIGHT exch def
12  sub neg /WIDTH exch def
13 } def
```

Darin werden zunächst die drei Parameter kopiert, um sie 'verbrauchen' zu können, der Text wird mit `charpath` in einen *path* gewandelt, dessen *BoundingBox* mit `pathbbox` berechnet wird. Nach dieser Anweisung liegen die Koordinaten der *BoundingBox* – X und Y der linken unteren Ecke sowie X und Y der rechten oberen Ecke – auf dem Stack. Die weiteren Zeilen berechnen daraus `HEIGHT` und `WIDTH` und legen die beiden Größen im *dictionary* ab.

1.6.4 Bit-, Verknüpfungs- und Vergleichsoperatoren

PostScript kennt die bitweisen oder logischen Verknüpfungen `and`, `or`, `not` und `xor`. Die Operanden müssen Integer- oder Boolesche Variable[15] sein, der Ergebnistyp entspricht den Operanden (polymorphe Operationen). Eine weitere in PostScript definierte Bitoperation ist die bitweise Verschiebung mit `bitshift`. Die zwei zugehörigen Integer-Operanden geben an, was um wie viele Bits verschoben wird (positiv ⇒ nach links). Nach `5 1 bitshift` liegt als Ergebnis 10 im Stack, nach `17 -2 bitshift` ist das Ergebnis 4.

[15] Der Datentyp Boolesch kann die beiden Werte `true` und `false` annehmen. Daten dieses Typs sind meist das Ergebnis von Vergleichsoperationen.

Die Vergleichsoperationen in PostScript können generell auf alle Objekte angewendet werden, bei denen die jeweilige Operation Sinn macht. Die Abfrage nach Gleichheit (`eq`) oder Ungleichheit (`ne`) ist mithin auf alle Objekte in PostScript anwendbar. 'Größer'- (`gt`), 'GrößerOderGleich'- (`ge`), 'KleinerOderGleich'- (`le`) und 'Kleiner'-Vergleiche (`lt`) dagegen nur auf numerische oder Zeichenkettenobjekte. Das Ergebnis ist immer Boolesch `true` oder `false`.

1.6.5 Kontrollstrukturen

Zur Steuerung eines Programmablaufs benötigt man Operatoren, die bedingte Ausführungen, Verzweigungen und Schleifen veranlassen. In PostScript sind dafür unter anderem die folgenden Operationen und Schlüsselwörter definiert:

if (2) : Ein Funktionsblock wird ausgeführt, wenn eine Bedingung erfüllt d. h. wahr (`true`) ist: `bool proc if`. Der erste Parameter (`bool`) ist ein Boolescher Ausdruck, beispielsweise das Ergebnis eines Vergleichs, der zweite (`proc`) ein Funktionsblock, der entweder früher definiert wurde (Name) oder direkt definiert wird (`{...}`).

ifelse (3) : Wie `if`, aber mit weiterem Funktionsblock für den `else`-Zweig: `bool if_proc else_proc ifelse`.

for (4) : Die übliche `for`-Schleife: `start incr stop proc for`. Startwert, Inkrement und Endwert der Schleife können beliebige reelle Zahlen sein, abhängig vom Vorzeichen des Inkrements wird die Schleife beendet, wenn der Endwert über- bzw. unterschritten wird. Am Beginn des Funktionsblocks `proc` liegt jeweils der aktuelle Wert der Schleifenvariablen auf dem Stack.

repeat (2) : Wiederholter Ablauf (Anzahl = 1. Parameter) eines Funktionsblocks (2. Parameter): `n proc repeat`.

loop (1) : Endlosschleife (`proc loop`).

exit (0) : Aussprung aus der Endlosschleife.

1.6.6 Beispiel: Interferenz

Graphische Darstellungen bzw. Muster kann man dann relativ kurz und effizient mit `for`-Schleifen programmieren, wenn sich das Muster mathematisch formulieren lässt und geeignet diskretisiert werden kann. Das trifft häufig auf physikalische Zusammenhänge zu. Ein Beispiel ist die Interferenz von zwei kreisförmigen Wellenfeldern, wie man sie von den Versuchen mit der Wellenwanne aus der Einführung in die Experimentalphysik kennt. Das Amplitudenfeld kann mathematisch durch

$$A = A_1 \sin\left(k_1|\vec{r} - \vec{r}_1|\right) + A_2 \sin\left(k_2|\vec{r} - \vec{r}_2|\right)$$

beschrieben werden. Physikalisch nicht ganz richtig, da konstante, nicht von der Entfernung zur Quelle abhängige Amplituden angenommen werden.

Abbildung 23 zeigt die auf Hell-Dunkel-Werte umgesetzte Amplitudenverteilung für relativ nah benachbarte Quellen, die etwa in der Mitte des linken Bildrands angeordnet sind. Es wurden gleiche Einzelamplituden ($A_1 = A_2$) und gleiche Wellenvektorbeträge ($k_1 = k_2$) für die beiden Wellenfelder vorausgesetzt. Man erkennt deutlich die Richtungen konstruktiver und destruktiver Interferenz.

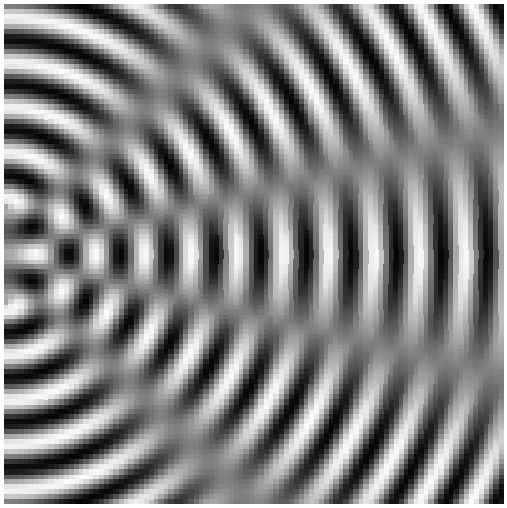


Abbildung 23: Interferenz zweier von benachbarten Punktquellen ausgehenden Sinuswellen.

Das zuständige PostScript-Programm besteht nur aus wenigen Zeilen:

```
 5  /^2 {dup mul} bind def
 6  /ymax 99 def /xmax 99 def
 7  /y1 40 def /y2 59 def /k 40 def
 8
 9  0 1 xmax {
10    /x exch def
11    0 1 ymax {
12      /y exch def
13      x ^2  y1 y sub ^2 add sqrt k mul sin
14      x ^2  y2 y sub ^2 add sqrt k mul sin
15      add
16      0.24 mul 0.5 add setgray x y 1 1 rectfill
17    } for
18  } for
```

Zunächst werden Funktionen und Parameter im *dictionary* abgelegt, dann wird in den geschachtelten `for`-Schleifen die Amplitudenverteilung berechnet. Die Schwärzung (`setgray`) wird proportional zur lokalen Amplitude – mit '`0.24 mul 0.5 add`' abgebildet auf den Bereich $0 \ldots 1$ – gesetzt (`rectfill`).

1.6.7 Rekursive Programmierung

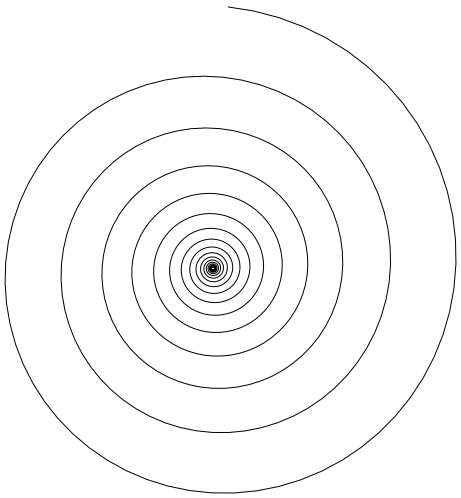
Da Funktionen in Gnuplot vorab definiert werden und erst bei der Ausführung interpretiert werden, kann auch sehr effizient rekursiv programmiert werden. Wichtig dabei ist eine souveräne und konsequente Handhabung des Stacks. Zwei Beispiele:

```
 1 ! = 1
 2 ! = 2
 3 ! = 6
 4 ! = 24
 5 ! = 120
 6 ! = 720
 7 ! = 5040
 8 ! = 40320
 9 ! = 362880
10 ! = 3628800
11 ! = 39916800
12 ! = 479001600
13 ! = 6.22702e+09
14 ! = 8.71783e+10
15 ! = 1.30767e+12
16 ! = 2.09228e+13
17 ! = 3.55687e+14
18 ! = 6.40237e+15
19 ! = 1.21645e+17
20 ! = 2.4329e+18
21 ! = 5.10909e+19
22 ! = 1.124e+21
23 ! = 2.5852e+22
24 ! = 6.20448e+23
25 ! = 1.55112e+25
26 ! = 4.03291e+26
27 ! = 1.08889e+28
28 ! = 3.04888e+29
29 ! = 8.84176e+30
30 ! = 2.65253e+32
31 ! = 8.22284e+33
32 ! = 2.63131e+35
33 ! = 8.68332e+36
34 ! = 2.95233e+38
```

```
 5 /xpos 150 def
 6 /ypos 750 def
 7 /s 40 string def
 8 /displayline {
 9  xpos ypos moveto
10  /ypos ypos 21 sub def
11  dup 10 lt {(  ) show} if
12  s cvs show
13  ( !  =  ) show
14  s cvs show
15 } def
16 /fac {
17  dup dup 1 gt
18   {dup 1 sub fac}
19   {1}
20  ifelse
21  mul dup 3 2 roll
22  displayline
23 } def
24 /Helvetica findfont
25 19 scalefont
26 setfont
27 34 fac
28 pop
```

Abbildung 24: Das Standardbeispiel zur rekursiven Programmierung, die Berechnung der Fakultät einer ganzen Zahl. Die grundlegende Funktion ist `fac`*, die mit einer Integerzahl als Parameter aufgerufen wird und deren Fakultät zurückgibt.* `fac` *ruft sich wie üblich rekursiv mit dekrementiertem Parameter auf, wenn dieser größer als 1 ist. Das Ergebnis des rekursiven Aufrufs wird mit dem vorher duplizierten Parameter multipliziert. Der Rest von* `fac` *sorgt zusammen mit* `displayline` *für die formatierte Ausgabe der ganzen Liste.*

Abbildung 25: Eine logarithmische Spirale, am aktuellen Punkt wird jeweils ein Linienstück (`step`*) angesetzt, das senkrecht auf dem Radiusvektor steht und 1/10 seiner Länge hat.*



```
 5 /step {
 6  2 copy
 7  10 div sub
 8  3 1 roll exch
 9  10 div add
10  2 copy
11  lineto
12  dup 300 lt
13   {step}
14   {pop pop}
15  ifelse
16 } def
17 300 400 translate
18 1 1 moveto
19 1 1 step
20 stroke
```

1.6.8 Felder

Felder (*arrays*) bieten in PostScript die Möglichkeit, mehrere – gleichartige oder unterschiedliche – Objekte zusammenzufassen. Man hat so die Möglichkeit, die Parameter zusammengehöriger Objekte unter einer Bezeichnung zunächst zu definieren und später einheitlich abzuarbeiten. Zum Arbeiten mit Feldern kennt PostScript unter anderem folgende Anweisungen:

`[ ... ]` definiert und initialisiert ein Feld,

array (1) legt ein leeres Feld mit `n` Plätzen im Stack an (`n array`),

length (1) errechnet die Länge (Zahl der Elemente) eines Feldes,

get (2) holt ein Objekt aus dem Feld, nach `array n get` liegt das (n+1)te Objekt auf dem Stack,

put (3) ersetzt ein Objekt im Feld an der Stelle n: `array n obj put`,

getinterval (3) kopiert einen Teilbereich (`array index count getinterval`),

putinterval (3) ersetzt einen Teilbereich (`array index subarray putinterval`),

aload (1) zerlegt das Feld in seine Objekte, legt alle, danach auch das ursprüngliche Feld im Stack ab,

forall (2) führt einen Funktionsblock für jedes Objekt im Feld aus (`array {...} forall`).

1.6.9 Beispiel: Kristallstruktur

Mathematisch einfach zu formulierende 3D-Objekte sind schräg beleuchtete Kugeln, die zur Visualisierung von Kristallstrukturen zusammengesetzt werden können. Zwei Beispiele, die kubische Hoch- und die tetragonale Tieftemperaturphase der Perowskitstruktur[16] zeigt Abbildung 26.

Das PostScript-Programm dafür besteht aus mehreren Teilen. Die zentrale Basisfunktion `DrawSphere` modelliert am Ort (`x,y`) eine beleuchtete Halbkugel mit Radius `r` in der Farbe `color`:

```
51  /DrawSphere {  % usage : x y r color DrawSphere
52  % ***** get parameters, leave x y on stack
53    /color exch def
54    rfac mul /r exch def
```

[16] In dieser Struktur kristallisieren viele Mischoxide des Typs ABO_3, unter anderem die für elektrooptische Anwendungen interessanten Kristalle $BaTiO_3$ und $KNbO_3$. Eine ähnliche Struktur weisen auch die meisten Hochtemperatur-Supraleiter auf.

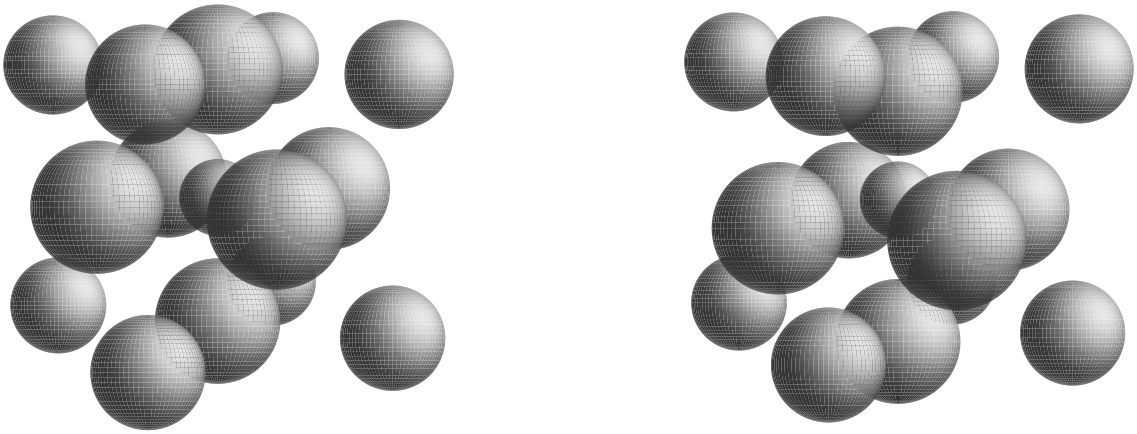


Abbildung 26: Perowskitstruktur ABO_3, links kubische Hochtemperaturphase, rechts tetragonal verzerrte Tieftemperaturphase. In den Würfelecken sitzen die A-Atome, im Zentrum das B-Atom, auf den Flächenmitten die O-Atome (große Kugeln).

```
55    gsave
56    translate      % x y from stack
57  % ***** paint sphere at point 0 0
58  ladelta 90 sub ladelta 90            % latitude loop
59  { /la exch def
60    /cfac la cos lalight cos mul def   % cfac = cos(la)*cos(lalight)
61    /sfac la sin lalight sin mul def   % sfac = sin(la)*sin(lalight)
62    -90 la ladelta sub xy  -90 la xy   % calc 2 starting points
63    lodelta 90 sub lodelta 90          % longitude loop
64    { /lo exch def
65      moveto lineto                    % use 2 points
66      lo la xy  lo la ladelta sub xy   % calc 2 more points
67      4 copy                           % copy for further usage
68      lineto lineto closepath          % close quadrangle
69      4 2 roll                         % exchange points
70  % ***** calculate light intensity: cfac*cos(lo-lolight) + sfac
71      iback
72      lo lolight sub cos cfac mul sfac add
73      dup 0 gt { ^2 ilight mul add } { pop } ifelse
74      gray
75        { setgray }
76        { 1 exch sub color exch bright sethsbcolor }
77      ifelse
78      fill
79    } for
80    pop pop pop pop   % pop 2 points left on stack
81  } for
82  grestore
83  } def
```

Die Modellierungsstrategie ist relativ einfach: Die Halbkugel wird durch Längen- und Breitenkreise in Vierecke zerschnitten. Diese werden der Lichteinstrahlung entsprechend eingefärbt. Da nur die sichtbare Halbkugel modelliert wird, laufen die beiden geschachtelten `for`-Schleifen über die geographische Länge (`longitude`, Variablennamen `lo...`) und die geographische Breite (`latitude`, `la...`) jeweils von -90 bis +90 Grad mit den Schrittweiten `lodelta` und `ladelta`. Die Helligkeit wird aus dem Winkelstand (`lolight` und `lalight`) der entfernten Lichtquelle, der Winkellage des betrachteten Vierecks (`lo`, `la`) und den Intensitäten von Lichtquelle (`ilight`) und Hintergrundbeleuchtung (`iback`) berechnet. Die Halbkugel wird zunächst an der Stelle (`0,0`) modelliert, dann mit `translate` nach (`x,y`) verschoben.

Die mehrfach verwendeten Funktionen `^2` und `xy` (Berechnung der 2D-Koordinaten `x` und `y` aus den 3D-Koordinaten Länge, Breite und Radius) sind vorab definiert:

```
/^2 { dup mul } def
/xy {              % calculate x,y from theta,phi using r
  dup sin r mul           % y = r*sin(phi)
  3 1 roll
  cos r mul exch sin mul  % x = r*cos(phi)*sin(theta)
  exch } def
```

Zur Berechnung der perspektivischen Verzerrung durch Betrachtungsabstand und -winkel wird die Funktion `Sphere` vorgeschaltet, die aus 3D-Koordinaten, Betrachterabstand `zvp` und Drehwinkeln (`xrot`, `yrot`) die von `DrawSphere` erwarteten 2D-Koordinaten berechnet:

```
/Sphere {      % usage : x y z r color Sphere
% ***** save parameters to dictionary
  /color exch def
  /r exch def
  /z exch def
  /y exch def
  /x exch def
% ***** rotate by yrot around y-axis
  /cz z yrot cos mul x yrot sin mul sub def
  /cy y def
  /cx z yrot sin mul x yrot cos mul add def
% ***** rotate by xrot around x-axis
  /z cz xrot cos mul cy xrot sin mul add def
  /y cy xrot cos mul cz xrot sin mul sub def
  /x cx def
% ***** calculate perspective size and position
  /fac zvp zvp z sub div def    % size factor
  /r r fac mul def
  /x x fac mul def
  /y y fac mul def
  x y r color DrawSphere   % call drawing procedure
} def
```

Im Hauptteil schließlich muss dann nur noch das Feld der Atomkoordinaten in der richtigen Reihenfolge (geordnet nach aufsteigendem `z` wegen der richtigen Überdeckung) definiert und in einer Schleife mit `Sphere`-Anweisungen abgearbeitet werden:

```
117  /A { 0.4 red } def    % A ion radius and color
118  /B { 0.3 green } def  % B ion radius and color
119  /O { 0.5 blue } def   % oxygen radius and color
120  /coordinates [        % high temperature cubic phase
121    [ 1 -1 -1  A ]      % array of [ x y z r color ]
122    [ 1  1 -1  A ]
123    [ 0  0 -1  O ]
124    [-1 -1 -1  A ]
125    [-1  1 -1  A ]
126    [ 1  0  0  O ]
127    [ 0  1  0  O ]
128    [ 0  0  0  B ]
129    [ 0 -1  0  O ]
130    [-1  0  0  O ]
131    [ 1 -1  1  A ]
132    [ 1  1  1  A ]
133    [ 0  0  1  O ]
134    [-1 -1  1  A ]
135    [-1  1  1  A ]
136  ] def
137  % **** the loop
138    coordinates { aload pop Sphere } forall
```

Das `forall`-Konstrukt legt nacheinander alle Objekte des `coordinates`-Feldes auf den Stack und führt für jedes den in Klammern definierten Funktionsblock aus. Diese Objekte sind auch wieder Felder (`[...]`), deren Bestandteile mit `aload` einzeln auf den Stack gelegt werden. Bevor `Sphere` aufgerufen wird, wird das von `aload` wieder auf den Stack zurückgelegte Feld mit `pop` verworfen.

1.6.10 Beispiel: Stufenversetzung in einem Kristall

Die oben beschriebenen Funktionen zum Modellieren von Halbkugelflächen lassen sich auch sehr einfach dazu verwenden, periodische Kristallstrukturen zu veranschaulichen. Man muss dabei darauf achten, dass die Graphiken in der richtigen Reihenfolge aufgebaut, d. h. die Schleifen jeweils in der richtigen Laufrichtung programmiert werden. Zwei Beispiele, Stufen auf einem einfach kubischen und auf einem kubisch raumzentrierten Kristallgitter zeigen die Abbildungen 27 und 28.

Das zugehörige ‘Hauptprogramm’ mit den benötigten `for`-Schleifen und einer Koordinatentransformation, die alles vernünftig auf der Seite platziert:

```
119  100 400 translate   % center on page
```

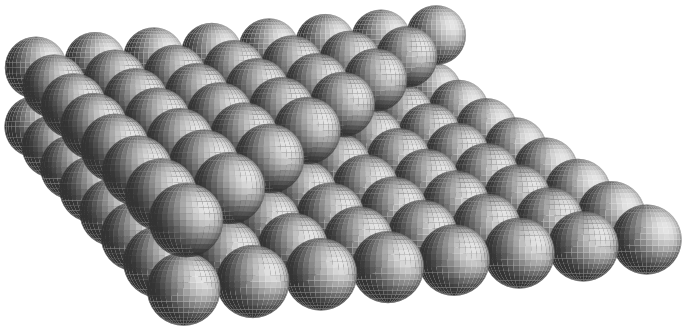


Abbildung 27: In 110-Richtung verlaufende Stufe im einfach kubischen Gitter.

```
120  30 30 scale          % magnify tiny coordinates
121  /rfac 0.5 def
122  % two atomic layers, one full square, one diag. cut
123  1 1 8 {
124    /Z exch def
125    8 -1 1 { 1 Z 1 red Sphere } for
126    9 Z sub -1 1 { 2 Z 1 red Sphere } for
127  } for
```

Mit der Konstanten `rfac` werden innerhalb der Funktion `DrawSphere` alle Radiusangaben multipliziert. Man kann damit global die Kugelradien festlegen.

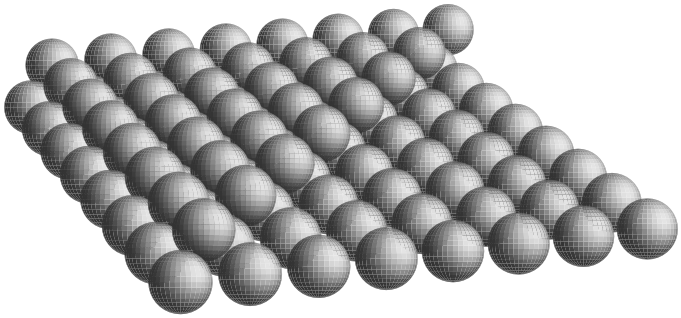


Abbildung 28: In 110-Richtung verlaufende Stufe im kubisch raumzentrierten Gitter.

Die generierende Schleife für das innenzentrierte Gitter:

```
122  /rfac 3 sqrt 4 div def
123  1 1 8 {
124    /Z exch def
125    9.5 Z sub -1 1 { 0.5 Z 0.5 sub 1 red Sphere } for
126    8 -1 1 { 0 Z 1 blue Sphere } for
127  } for
```

Damit sich im kubisch innenzentrierten Gitter die Kugeln berühren, muss ihr Radius (wieder über `rfac` global festgelegt) ein Viertel der Raumdiagonale sein.

2 Gnuplot

Gnuplot ist ein vergleichsweise einfach zu bedienendes Programm zur Visualisierung von Daten und Funktionen. Es ist frei verfügbar [4] und für viele Betriebssysteme übersetzt, die C-Quellen sind allgemein zugänglich. Gut geeignet ist es für 2-dimensionale, bedingt für 3-dimensionale Darstellungen, ist allerdings rein linienorientiert[17], mithin sind komplexere Graphiken wie etwa 3-D-Modellierungen mit eingefärbten Flächenelementen nicht möglich. Zur mathematischen Beschreibung von Graphiken oder zur Umrechnung von Daten ist in Gnuplot ein umfangreicher Befehlsinterpreter eingebaut – mit C-ähnlicher Syntax und einer großen Zahl zusätzlicher mathematischer Funktionen sowie der Möglichkeit,mit komplexen Konstanten zu rechnen. Darüber hinaus ist ein Fit-Progamm integriert, das nach dem Levenberg-Marquardt-Algorithmus [5] arbeitet. Damit lassen sich benutzerdefinierte Funktionen an Datensätze, etwa verrauschte Messdaten, anpassen.

Ausführliche Dokumentation zu Gnuplot in verschiedenen Formaten kann man sich vom zuständigen FTP-Server [4] holen, daneben liegt eine umfangreiche FAQ-Liste im Internet [6]. Eine Kurzeinführung in die Bedienung von Gnuplot wurde vor einiger Zeit vom Rechenzentrum erstellt [7]. Im Folgenden werden daher nur einige ausgewählte Themen behandelt.

2.1 Skripte und Batch-Ausführung

Gnuplot kann sowohl interaktiv – kommandozeilenorientiert, wie auch über Skripte – Textdateien mit Befehlssequenzen, bedient werden. Darüber hinaus sind bei einigen Implementierungen (Windows etc.) umfangreiche graphische Benutzeroberflächen integriert, die die Bedienung über ein Menüsystem zulassen. Trotz dieser einfachen und intuitiven direkten Zugangsmöglichkeit sollte man sich angewöhnen, Graphiken durch Skripte zu generieren, um auf diese Weise reproduzierbar zu arbeiten. Gnuplot erleichtert die Skripterstellung außerordentlich dadurch, dass man die aktuelle, vorher vielleicht durch Einzelanweisungen erstellte Graphik zusammen mit den gerade gültigen Einstellungen von Gnuplot (`set`) als Skript speichern kann (`save`). Dieses Skript lässt sich dann als Basis für Änderungen verwenden. Ist die Graphik am Bildschirm zufriedenstellend, fügt man in das Skript noch zwei Zeilen für Ausgabegerät (`set terminal ...`) und Ausgabedatei (`set output ...`) ein, um die Graphik im gewünschten Format als Datei zu speichern.

Mit fertigen Skripten lässt sich Gnuplot auch als Batch-Anwendung betreiben. Es wird über die Kommandozeile aufgerufen, als Parameter werden die Namen von einer oder mehreren Skriptdateien mitgegeben. Diese Dateien werden von Gnuplot im Hintergrund abgearbeitet, danach beendet sich Gnuplot. Will man das Ergebnis in Gnuplot vor dem Programmende ansehen oder nach dem Ende des Skripts in Gnuplot weiterarbeiten, kann man entweder durch eine `pause`-Anweisung an geeigneter Stelle die Skriptdatei unterbre-

[17] Dies liegt daran, dass Gnuplot ursprünglich zur Verwendung mit verschiedenen Ausgabegeräten, insbesondere auch Stiftplottern, konzipiert wurde.

chen oder durch den speziellen Kommandozeilenparameter "`-`" (Minuszeichen) Gnuplot daran hindern, sich zu beenden.

2.2 Web-Graphiken

Zur Visualisierung von Messdaten auf Webseiten wird als Format meist GIF[18], seltener PNG[19] verwendet. Beide Formate können aus Gnuplot generiert werden, Näheres in der Hilfe zu `set terminal`. GIF und PNG sind Bitmapformate, keine Vektorformate wie PostScript. Daher ist es wichtig, schon bei der Herstellung der Graphiken die Größe anzugeben. Voreingestellt bei beiden ist 640*480. Diese Voreinstellung kann über den `set␣size`-Befehl geändert werden, dabei ist zu beachten, dass die Zahlenwerte Multiplikatoren zur Voreinstellung sind. Bei GIF kann die Bildgröße auch direkt innerhalb des `set␣terminal`-Kommandos eingestellt werden, die Größe wird dort absolut (in Pixeln) angegeben.

Durch die Batch-Fähigkeit von Gnuplot (vgl. 2.1) hat man die Möglichkeit, Graphiken in Webseiten jeweils frisch zu erstellen ('*graphics on the fly*'). Dies ist dann sinnvoll, wenn sich die zugehörigen Daten ständig verändern, z. B. bei einer laufenden Messung, die über das Web kontrolliert wird, oder bei Daten mit zeitaktuellem Bezug wie Wetterdaten. Man lässt sich durch ein geeignetes Skript von Gnuplot bei jedem Zugriff auf die Webseite eine Graphik aus dem aktuellen Datenbestand erstellen[20].

2.3 Gnuplot und PostScript

Für Graphiken, die in LaTeX-Dokumente eingebunden werden sollen, bietet Gnuplot eine ganze Reihe von Formaten an, die an verschiedene LaTeX-Graphik-Erweiterungen angepasst sind. Diese Ausgabeformate ermöglichen die einfache Weiterbearbeitung der Graphik mit TeX-Mitteln. Ein kleiner Nachteil ist allerdings, dass solche Graphiken in der Regel nicht problemlos skaliert werden können, da meist TeX-Fonts mit fester Größe für die Beschriftungen verwendet werden. Andererseits passt die Beschriftung dadurch sehr gut zum Textteil des Dokuments.

Benötigt man skalierbare Graphiken, die in gewohnter Weise in LaTeX-Dokumenten verwendet werden sollen, ist es zweckmäßig, EPS-Format aus Gnuplot zu generieren. Der PostScript-Treiber in Gnuplot wurde zur Version 3.7 erweitert (Option `enhanced`), insbesondere in den Möglichkeiten zur Beschriftung von Graphiken. Man kann nun innerhalb des Beschriftungstextes Schriftgröße und -font verändern, Sonderzeichen wählen, Hoch- und Tiefstellung von Textteilen anordnen. So können in Achsenbeschriftungen beispiels-

[18] Graphics Interchange Format, Compuserve. Mit der dort üblicherweise verwendeten LZW-Komprimierung (Lempel-Ziv Welch) gibt es lizenzrechtliche Probleme, Public-Domain-Programme verwenden daher häufig RLE-Komprimierung (Run Length Encoding).

[19] Portable Network Graphics, bessere Komprimierung als GIF, keine Lizenzprobleme.

[20] Vorsicht bei scheinbar effizienten Browsern. Netscape z. B. tendiert dazu, bei Graphiken nicht die Aktualität zu überprüfen, sondern die alte aus dem Cache zu holen. Man muss dies gegebenenfalls dadurch verhindern, dass man Dateinamen oder besser Namenszusätze ständig verändert.

octal	0	1	2	3	4	5	6	7
\00x								
\01x								
\02x								
\03x								
\04x		!	∀	#	∃	%	&	∍
\05x	(	)	*	+	,	−	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	≅	Α	Β	Χ	Δ	Ε	Φ	Γ
\11x	Η	Ι	ϑ	Κ	Λ	Μ	Ν	Ο
\12x	Π	Θ	Ρ	Σ	Τ	Υ	ς	Ω
\13x	Ξ	Ψ	Ζ	[	∴	]	⊥	_
\14x	‾	α	β	χ	δ	ε	ϕ	γ
\15x	η	ι	φ	κ	λ	μ	ν	ο
\16x	π	θ	ρ	σ	τ	υ	ϖ	ω
\17x	ξ	ψ	ζ	{	|	}	~	
\20x								
\21x								
\22x								
\23x								
\24x		ϒ	′	≤	⁄	∞	*f*	♣
\25x	♦	♥	♠	↔	←	↑	→	↓
\26x	°	±	″	≥	×	∝	∂	•
\27x	÷	≠	≡	≈	…	|	—	↵
\30x	ℵ	ℑ	ℜ	℘	⊗	⊕	∅	∩
\31x	∪	⊃	⊇	⊄	⊂	⊆	∈	∉
\32x	∠	∇	®	©	™	∏	√	·
\33x	¬	∧	∨	⇔	⇐	⇑	⇒	⇓
\34x	◊	〈	®	©	™	∑	⎛	⎮
\35x	⎝	⎡	⎮	⎣	⎧	⎨	⎩	⎮
\36x		〉	⌠	⌡	⎮	⎫	⎭	⎮
\37x	⎠	⎤	⎮	⎦	⎫	⎬	⎭	

Tabelle 1: Das Kodierungsschema für den Symbol-Font in PostScript.

weise griechische Buchstaben aus dem Symbol-Font von Postscript (eine Übersicht über die Kodierung des Fonts gibt Tabelle 1) oder hochgestellte Zahlen (bei Maßeinheiten) verwendet werden.

Die Syntax für die erweiterten PostScript-Möglichkeiten ist eine Mischung aus TEX und PostScript. Blöcke – insbesondere solche mit geändertem Font – werden in geschweifte Klammern eingeschlossen, Hoch- und Tiefstellungen erfolgen TEX-üblich mit "^" und "_". Ist beides – hoch und tief – erforderlich, kann mit `@` eine Box der Breite 0 für eines der beiden Objekte erzwungen werden. Fonts werden mit `/` nach der öffnenden geschweiften Klammer gefolgt von Fontnamen und/oder -größe umgestellt, Sonderzeichen, aber auch alle anderen Zeichen, können in oktaler Schreibweise (PostScript-üblich mit *ooo*) dargestellt werden. Definierter, einer Zeichenfolge entsprechender Zwischenraum (dem `\phantom` in TEXentsprechend) wird mit `&` eingefügt, und falls eines der Spezialzeichen selbst benötigt wird, hilft der Backslash. Beispiele:

`m^2, s^{-1}, x_{ik}`	⇒	m^2, s^{-1}, x_{ik}
`x_1@^2, x@^{1/4}_1y, x^{1/4}@_1y`	⇒	x_1^2, $x_1{}^{1/4}y$, $x_1^{1/4}y$
`{/Symbol w}, {/Symbol \167}`	⇒	ω, ω
`{/=9 abc}, {/Helvetica=14 abc}`	⇒	abc, abc
`<nnn>, <&{nnn}>, \_, \}`	⇒	<nnn>, < >, _, }

Werden in Beschriftungen europäische Sonderzeichen (Umlaute) benutzt, werden diese zwar am Bildschirm meist richtig dargestellt, nicht jedoch in der PostScript-Ausgabe. Dazu muss die richtige Kodierung – *ISOLatin1Encoding* – für PostScript eingestellt werden. Das erledigt die Gnuplot-Anweisung `set encoding iso_8859_1`. Ohne die Umschaltung arbeitet der PostScript-Treiber mit der Voreinstellung *StandardEncoding*, in der nur einige wenige Sonderzeichen zugänglich sind.

2.4 Datenformate, Umrechnung von Daten

Daten für die beiden Plot-Funktionen `plot` und `splot` können als Funktion angegeben oder aus Dateien gelesen werden. Im ersten Fall wird die Zahl der Datenpunkte durch `set samples` oder `set isosamples` eingestellt, bei Dateien kann die Anzahl durch `every`- oder `index`-Anweisungen eingeschränkt werden. Mit `smooth` wird eine Glättungsfunktion durch die Datenpunkte gelegt, die Art der Funktion ist durch ein Zusatzargument anzugeben, die Zahl der Punkte ist wie bei Funktionsplots durch `samples` bestimmt.

Grundsätzlich werden Daten in der benötigten Reihenfolge und Anzahl aus den durch Leerzeichen oder Tabs getrennten Textspalten der Datendatei entnommen, Gnuplot versucht dabei, die Spaltenanordnung vernünftig zu interpretieren. Wird mit `plot` eine Datei

geplottet, die nur eine einzige Spalte mit Zahlenwerten enthält, werden diese Daten als y-Werte interpretiert, bei zwei Spalten dagegen als x-y-Paar.

Diese Voreinstellung von Gnuplot kann für die aktuellen Daten mit `using` verändert werden. Damit kann zum einen die Reihenfolge der Datenspalten verändert werden, zum anderen können Daten vor der graphischen Darstellung umgerechnet werden. So bewirkt `using 2:4` die Interpretation der zweiten Spalte als x-, der vierten Spalte als y-Werte. Wollte man den Sinus der mit $7\pi/180$ multiplizierten Differenz aus vierter und dritter Spalte über dem Quadrat der ersten auftragen, könnte man das mit

```
plot 'data.txt' using ($1*$1):(sin(7*pi/180*$4-$3))
```

erledigen. Statt Gnuplot-spezifischer können an dieser Stelle auch (vorher definierte) eigene Funktionen verwendet werden. Darüber hinaus kann `using` dazu verwendet werden, bestimmte Daten nicht mit in die Graphik aufzunehmen:

```
using 1:(($2>100) ? 1/0 : $2)
```

würde anordnen, dass Werte > 100 unterdrückt werden ("`1/0`" wird nicht geplottet).

Wie in den Beispielen gezeigt, müssen im Argument von `using` Angaben für die im Plot benötigten Variablen stehen, der Doppelpunkt trennt die einzelnen Variablenfelder. Die Angaben können einfache Zahlen sein – die jeweilige Spaltennummer – oder Ausdrücke in runden Klammern, darin die Spaltenangaben mit "`$`".

Zusätzlich kann nach `using` das Format der zu lesenden Daten angegeben werden. Das ist beispielsweise dann nötig, wenn Textspalten überlesen werden sollen oder wenn die Daten durch andere Trenner als Leerzeichen oder Tabs getrennt sind. Die Angabe erfolgt analog zur `scanf`-Syntax und wird in einfache Anführungszeichen gesetzt. So wird mit `using 1:3 '%lf;%lf;%lf'` die erste und dritte Spalte aus einer Datei gelesen, in der das Semikolon als Trenner verwendet wird. Mithilfe solcher Formatanweisungen ist es auch möglich, Kommas als Dezimaltrenner zu verarbeiten, zumindest dann, wenn die Zahl der Nachkommastellen in jeder Spalte konstant ist. Die Datei

```
10,25   14,511
 8,14  129,540
 ...     ...
```

könnte gelesen und geplottet werden mit

```
plot 'data.txt' using ($1+$2/100):($3+$4/1000) '%lf,%lf%lf,%lf' .
```

Für Daten in `splot`-Anweisungen sind noch zwei spezielle, an die Dimensionalität der Graphik angepasste, Formate möglich, die durch die Spezifikationen `binary` und `matrix` gekennzeichnet werden. In beiden Formaten werden die z-Werte eines regulären Gitters matrixartig angegeben, binär oder als Text. Genaueres dazu in der Online-Hilfe.

2.5 Komplexe Zahlen

Gnuplot kann nicht nur mit reellen Größen rechnen, sondern auch mit komplexen. Allerdings ist nur eine Syntax für komplexe Konstanten definiert, ein Zahlenpaar in geschweiften Klammern (`{real,imag}`). Um mit komplexen *Variablen* zu arbeiten, multipliziert man reelle Variable mit geeigneten komplexen Konstanten. Fast alle eingebauten Funktionen und Operatoren kommen mit komplexen Argumenten zurecht. Einige spezielle Funktionen dienen dazu, die für Graphiken benötigten reellen Größen aus den komplexen zu berechnen. So liefert `abs` den Betrag, `arg` die Phase, `real` den Real- und `imag` den Imaginärteil der komplexen Größe.

Komplexes Rechnen erleichtert die mathematische Beschreibung von elektr(on)ischen Schaltungen, am Beispiel eines einfachen Bandpass-Filters (Abbildung 29) sei dies demonstriert.

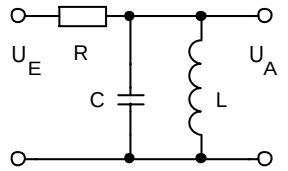


Abbildung 29: RLC-Filter mit LC-Parallelschwingkreis.

Die Durchlass-Charakteristik der Filter-Schaltung ist gegeben durch

$$T = \frac{U_A}{U_E} = \frac{R_C \| R_L}{R + (R_C \| R_L)}\,, \tag{2.1}$$

mit der Parallelschaltung aus C und L

$$R_C \| R_L = \left(j\omega C + \frac{1}{j\omega L} \right)^{-1} \tag{2.2}$$

wird

$$T = \left[1 + R \cdot \left(j\omega C + \frac{1}{j\omega L} \right) \right]^{-1}\,. \tag{2.3}$$

Umgesetzt auf ein Gnuplot-Skript wird daraus:

```
 4  R=1;  L=1;  C=1
 5  j={0,1}
 6  T(w)=1/(1+R*(j*w*C+1/(j*w*L)))
 7  ampl(w)=abs(T(w))
 8  phase(w)=180/pi*arg(T(w))
 9  set xrange [0.001:1000];  set logscale x
10  set xlabel "Frequenz  [ {/Symbol w}/{/Symbol w}_0 ]"
11  set yrange [0:1.05];  set ylabel "Amplitude"
12  set ytics border nomirror norotate autofreq
13  set y2range [-95:95];  set y2label "Phase"
14  set y2tics border nomirror norotate autofreq
15  set terminal postscript eps enhanced "Helvetica" 25
```

```
16 set output "rlcfun.ps"
17 plot ampl(x) title "Amplitude" lw 4, \
18    phase(x) axes x1y2 title "Phase" lw 4
```

In den ersten Zeilen wird die Durchlass-Funktion T erklärt und daraus Amplitude und Phase berechnet. Der Einfachheit halber werden R, L und C jeweils auf den Wert 1 gesetzt, bei anderen Werten muss gegebenenfalls die x-Achse angepasst werden. Die imaginäre Einheit j wird als komplexe Konstante `{0,1}` definiert, mit der dann die komplexen Widerstände ausgerechnet werden. Die Folge von `set`-Anweisungen legt die Achsenskalierungen und -beschriftungen fest, es werden 2 verschiedene y-Achsen und eine logarithmische x-Achse eingestellt, in der x-Achsen-Beschriftung werden spezielle Anweisungen für den PostScript-Enhanced-Treiber verwendet: Font-Umschaltung und Tiefstellung. Der abschließende `plot`-Befehl erzeugt schließlich die Graphik (Abbildung 30). Der Amplitu-

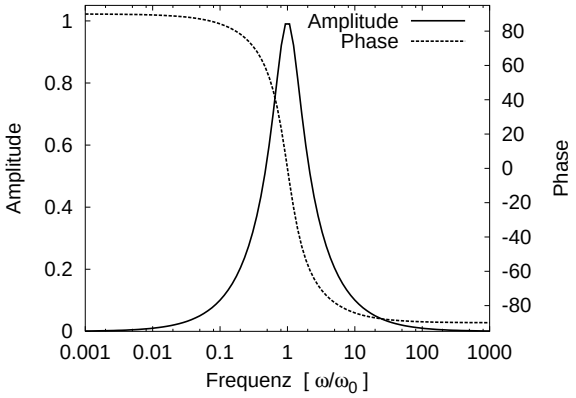


Abbildung 30: Amplituden- und Phasengang für den RLC-Bandpass der Abbildung 29 für $R = L = C = 1$ (in den jeweiligen SI-Einheiten).

denverlauf ist vergleichsweise breit, das heißt, dieser Bandpass ist nicht besonders selektiv. Das liegt an der willkürlichen Festsetzung $R = L = C = 1$, die für reale Anwendungen natürlich verändert werden muss. Abgesehen davon entspricht der Verlauf den Erwartungen: Ein in der logarithmischen Auftragung symmetrisches Maximum bei ω_0 und ein Phasengang von +90 nach -90 Grad mit der Nullstelle bei ω_0.

2.6 Ausgleichskurven

Wird für Messdaten eine funktionale Gesetzmäßigkeit vermutet oder erwartet, dann ist man daran interessiert, die Parameter dieser Funktion möglichst optimal zu bestimmen, um den Verlauf oder Wert der physikalischen Größen mathematisch beschreiben zu können. So wird man Messdaten einer gleichförmigen Bewegung durch eine lineare Funktion (Ausgleichsgerade), einer gleichmäßig beschleunigten Bewegung durch eine quadratische Funktion (Parabel) beschreiben, deren Parameter man jeweils so bestimmt, dass die Abweichung dieser Funktion von den Messdaten insgesamt minimal ist. Der Begriff ‘insgesamt minimal’ ist mathematisch ziemlich unscharf und bedarf näherer Konkretisierung. Es kann damit gemeint sein, dass die Summe der Absolutabweichungen oder die Summe der quadra-

tischen Abweichungen minimiert wird. Darüber hinaus können Messwerte unterschiedlich gewichtet werden, beispielsweise abhängig von ihrer Genauigkeit.

In Gnuplot ist eine Fit-Funktion implementiert, die die Summe der quadratischen Abweichungen nach einer bestimmten Strategie – dem Levenberg-Marquardt-Algorithmus [5] – minimiert. Sind im Datenfile auch Messfehler angegeben, werden die einzelnen Messdaten umgekehrt proportional zum Quadrat des Messfehlers gewichtet. Der Fit-Algorithmus wird mit `fit` aufgerufen, mit den Parametern Datenbereich (fakultativ), Fitfunktion, Datendatei (gegebenenfalls mit Modifizierung – `using`, `every`), und dem Schlüsselwort `via` gefolgt von den anzupassenden Parametern oder dem Namen eine Parameterdatei.

Ein einfaches Beispiel aus der Optik für die Ersetzung von Einzelmesswerten durch eine angepasste Interpolationsfunktion ist die Beschreibung der Wellenlängenabhängigkeit des Brechungsindexes durch einen einfachen funktionalen Zusammenhang. Üblicherweise misst man den Brechungsindex nur bei einigen festen Wellenlängen, die durch die verwendeten Lichtquellen (Spektrallampe) vorgegeben sind. Benötigt man Werte bei anderen Wellenlängen, muss man geeignet dazwischen interpolieren. Es gibt nun die Möglichkeit, stückweise – linear, quadratisch, kubisch,. . . – zu interpolieren, oder einen größeren Bereich durch eine einzige, dann physikalisch zu begründende Anpassungsfunktion mathematisch zu beschreiben. Die Anpassung durch eine Funktion hat den großen Vorteil, dass man mit wenigen Parametern zurecht kommt, außerdem kann man viele, unterschiedlich genaue Messwerte im Fit berücksichtigen. Beim gewählten Beispiel, dem Brechungsindex, ist es recht einfach, die Form der Fitfunktion physikalisch herzuleiten. Beschränkt man sich nämlich auf den sichtbaren Spektralbereich und nahes Infrarot, lässt sich die dielektrische Suszeptibilität χ meist mit ausreichender Genauigkeit durch Resonanzstellen (elektronische Anregungszustände) bei einer festen Frequenz ω_0 im nahen Ultraviolett beschreiben:

$$\chi = \frac{K}{\omega_0^2 - \omega^2}\,. \tag{2.4}$$

Darin ist K eine materialspezifische Konstante, in der unter anderem die Dichte der relevanten Elektronen und die Übergangswahrscheinlichkeit für deren Anregung enthalten ist. Der Brechungsindex ergibt sich daraus zu

$$n^2 = \varepsilon = 1 + \chi\,. \tag{2.5}$$

Für technische Anwendungen im sichtbaren Spektralbereich ist es üblich, n in Abhängigkeit von der Wellenlänge λ anzugeben:

$$n^2 = A + \frac{A_0}{\lambda^2 - \lambda_0^2}\,. \tag{2.6}$$

Die drei Parameter A, A_0 und λ_0 sind auf geeignete Weise zu bestimmen.

Zur Verbesserung der Beschreibung können weitere Resonanzterme bei anderen Wellenlängen hinzugefügt werden. Wie viele man benötigt, hängt von der erforderlichen Genauigkeit ab, aber auch vom Wellenlängenbereich, für den die Beschreibung gültig sein soll.

Die Berechnung der Fitfunktion mit Gnuplot:

```
14  n(x)=sqrt(A+A0/(x*x-L0*L0))
15  A=2.5;  A0=40000;  L0=400
16  fit [400:1100] n(x) "n_a.dat" using (1e9*$1):2:3 via A,A0,L0
17  plot "n_a.dat" using (1e9*$1):2:3 with errorbars, n(x) lt 1 lw 3
```

Die Funktion wird definiert, Anfangswerte für die Parameter werden festgelegt[21]. Die dritte Zeile führt den Fit durch, die Funktion `n(x)` wird an die Daten in der Datei `n_a.dat` im Bereich 400 – 1100 nm angepasst. Die hier verwendete `using`-Modifikation sorgt dafür, dass die in der Datendatei in Metern angegebenen Wellenlängen in handlichere Einheiten (nm) umgerechnet werden. Mit `via` werden die zu variierenden Parameter in der Funktion `n(x)` definiert. Der Verlauf und das Ergebnis des Fits werden im Textfenster von Gnuplot angezeigt, eine Zusammenfassung wird in einer Datei `fit.log` abgespeichert[22]. Die letzte Zeile schließlich zeichnet die Daten mit Fehlerbalken zusammen mit der Anpassungsfunktion. Das Ergebnis zeigt Abbildung 31, zusätzlich ist hier die Fit-Funktion mit den Startwerten der Parameter eingezeichnet (gestrichelte Kurve).

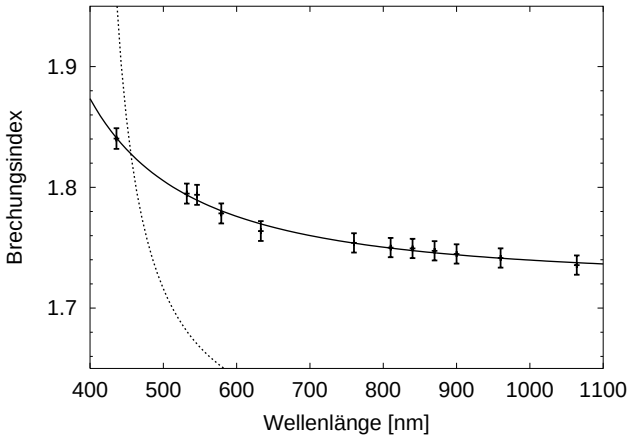


Abbildung 31: Brechungsindex n_a von Benzophenon. Eingezeichnet sind die Messwerte mit ihren Fehlergrenzen, die Ausgangsfunktion (gestrichelt) und die Ergebnisfunktion (durchgezogen) des Fits.

Das Layout der Abbildung (Achsenbeschriftungen etc.) wird erzeugt mit:

```
4  set nokey
5  set xrange [400:1100];  set yrange [1.65:1.95]
6  set ytics 1.6, 0.1, 1.9;  set mytics 5
7  set encoding iso_8859_1
8  set xlabel "Wellenlänge [nm]"
9  set ylabel "Brechungsindex"
```

[21] Der Fit-Algorithmus sucht ein Minimum für die Summe der quadratischen Abweichungen und variiert dabei die Parameter ausgehend von den Anfangswerten. Sinnvolle Anfangswerte sind für den Fit-Algorithmus wichtig, damit das absolute Minimum gefunden wird und der Fit nicht in einem Nebenminimum mit ungünstigen Parametern landet.

[22] Neue Fit-Ergebnisse werden jeweils ans Ende der Log-Datei angefügt; sie wächst stetig, sollte daher ab und zu gelöscht oder gekürzt werden.

Abbildung 31 zeigt, dass die gewählte Fit-Funktion den Verlauf der Messwerte sehr gut beschreibt. Die Hinzunahme weiterer Resonanzterme, d. h. weiterer Fitparameter zur Fit-Funktion würde das Ergebnis höchstens unwesentlich verbessern. Eine stückweise Anpassung über jeweils kleinere Bereiche würde den Verlauf eher ungenauer beschreiben und vor allem der physikalischen Begründung der Dispersion nicht gerecht werden.

Dass ein kompletter Messdatensatz durch eine vergleichsweise einfache Funktion physikalisch beschrieben werden kann, ist eher die Ausnahme. Häufig können aber Teilbereiche durch Standardfunktionen angefittet werden. Auf diese Weise lassen sich aus den meist verrauschten Messdaten mathematisch einigermaßen exakt die charakteristischen Größen extrahieren. So sind in allen Arten von Spektren – optischen, Gamma-, Elektronenverlust-, Elektronenspinresonanz- usw. – typische Kurvenformen enthalten, aus denen physikalische Größen wie Energie, Intensität, Dämpfung bestimmt werden können. Wenn man diese Größen nicht nur mit dem 'Augenmaß' des Experimentators bestimmen will, passt man Theoriekurven an, die die gesuchten Größen als Anpassparameter enthalten.

Das folgende Beispiel dazu ist der Kernphysik entnommen: Ein Gammaspektrum von Cobalt-60, wie es üblicherweise mit Praktikumsmitteln (Szintillationszähler, Vielkanalanalysator) gemessen wird. Das Teilbild links oben in Abbildung 32 zeigt das gemessene Originalspektrum. Es enthält drei für den Strahler typische Charakteristika, die beiden Gamma-Linien bei etwa 1170 und 1330 keV und deren Summenlinie bei 2500 keV. Die Linien zeigen die für Szintillationszähler typische Verbreiterung, man vermutet Gauß-Kurven, da die Verbreiterung rein statistischer Natur ist. Jede der Linien müsste sich also durch eine Funktion des Typs

$$f(E) = a \cdot \exp\left(-\frac{(E-E_0)^2}{b^2} \cdot \ln 2\right) \tag{2.7}$$

beschreiben lassen. Darin ist a die Peak-Intensität, E_0 die energetische Lage und b die halbe Halbwertsbreite der Linie.

Man sollte nun nicht versuchen, die gesamten Daten durch einen globalen Fit mit einer Summe aus mehreren solcher Linien anzupassen, man hätte zu viele Parameter und würde vermutlich in irgendeinem Nebenminimum landen. Zweckmäßigerweise passt man die verschiedenen Linien einzeln durch Gauß-Funktionen an und beschränkt dabei den Fit-Bereich auf die Linie und die direkte Umgebung. Insgesamt erledigt das ein Gnuplot-Skript etwa der Form:

```
 4  set nokey;  set noytics
 5  set xrange [0:2800];  set yrange [0:5500]
 6  plot "co60.txt" pt 6
 7  pause -1 "Startfunktionen"
 8  f1(x)=a1*exp((e1-x)*(x-e1)*log(2)/b1/b1)
 9  f2(x)=a2*exp((e2-x)*(x-e2)*log(2)/b2/b2)
10  f3(x)=a3*exp((e3-x)*(x-e3)*log(2)/b3/b3)
11  a1=3000;  e1=1100;  b1=30
```

```
12  a2=3000;  e2=1300;  b2=30
13  a3=1000;  e3=2400;  b3=30
14  plot "co60.txt" pt 6, f1(x) lw 4, f2(x) lw 4, f3(x) lw 4
15  pause -1 "Fit und Endergebnis"
16  fit [1100:1250] f1(x) "co60.txt" via a1,e1,b1
17  fit [1250:1400] f2(x) "co60.txt" via a2,e2,b2
18  fit [2300:2700] f3(x) "co60.txt" via a3,e3,b3
19  plot "co60.txt" pt 6, f1(x) lw 4, f2(x) lw 4, f3(x) lw 4
```

Zunächst werden die Messdaten vorgestellt, dann werden drei Gauß-Kurven definiert, die Anfangsparameter werden so gewählt, dass die Gauß-Kurven schon in der Nähe der Linie liegen, die sie beschreiben sollen. Schließlich werden die drei Fits durchgeführt, beschränkt auf die Umgebung der jeweiligen Linie. Die Plots des Skripts sind in Abbildung 32 dokumentiert: Links oben die Daten, rechts oben die Ausgangssituation der Fits, rechts unten das Endergebnis.

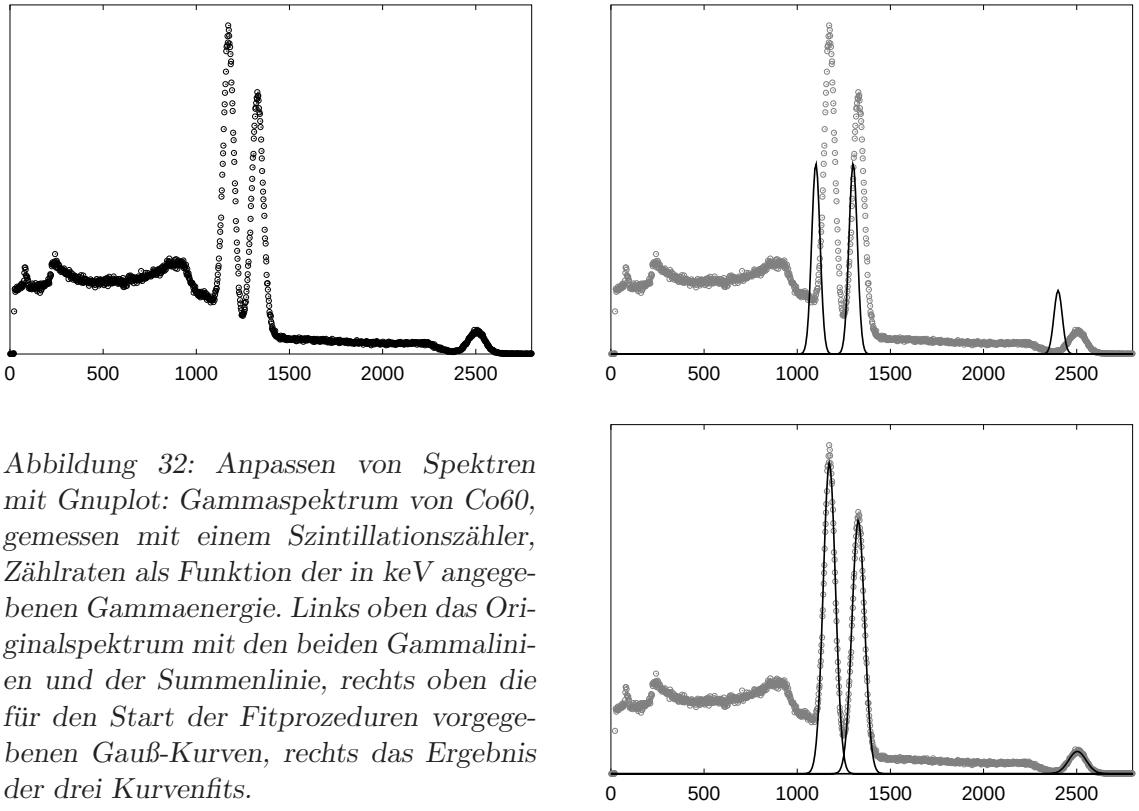


Abbildung 32: Anpassen von Spektren mit Gnuplot: Gammaspektrum von Co60, gemessen mit einem Szintillationszähler, Zählraten als Funktion der in keV angegebenen Gammaenergie. Links oben das Originalspektrum mit den beiden Gammalinien und der Summenlinie, rechts oben die für den Start der Fitprozeduren vorgegebenen Gauß-Kurven, rechts das Ergebnis der drei Kurvenfits.

Aus den gefitteten Anpassparametern lassen sich nun Linienlage E_i, Halbwertsbreite b_i und Gesamtintensität der jeweiligen Linie $I_i \propto a_i \cdot b_i$ bestimmen.

2.7 Polarkoordinaten in 2D

Will man die Winkelabhängigkeit einer Funktion oder eines Messwertes darstellen, bietet einem Gnuplot die Möglichkeit, mit Polarkoordinaten zu arbeiten. Dies wird mit `set polar` angeordnet, Funktionen können dann in Abhängigkeit von einer Winkelkoordinate `t` angegeben werden[23].

Das folgende Gnuplot-Skript verdeutlicht die Verwendung der Polardarstellung, winkelabhängig gemessene Lichtstreuintensitäten sollen veranschaulicht werden.

```
 4  set polar
 5  set angles degrees
 6  set noborder
 7  set grid polar 30
 8  set format xy ""
 9  set ticscale 0
10  set size square
11  set xrange [ -5000 : 5000 ]
12  set yrange [ -5000 : 5000 ]
13  set nokey
14  set arrow 1 from -4000, 0, 0 to -250, 0, 0  lt 1 lw 10
15  set arrow 2 from 0, -300, 0 to 0, 300, 0  nohead lt 1 lw 10
16  plot "powder0.dat" w p pt 1,\
17    "powder1.dat" using 1:(0.8*$2) w l lw 4,\
18    "powder0.dat" using (360-$1):2 w p pt 1, \
19    "powder1.dat" using (360-$1):(0.8*$2) w l lw 4
```

Es wird auf Polarkoordinaten umgeschaltet, als Winkelmaß werden 'Grad' eingestellt. Statt dem gewohnten Rahmen (`border`) wird ein polares Gitter mit 30-Grad-Unterteilung eingestellt. Die Beschriftung der x- und y-Ticks wird indirekt über die Formatangabe unterdrückt, das Zeichnen der Ticks über die Größenangabe (`ticscale`). Die *Size*- und *Range*-Angaben sorgen für die gewünschten Skalen und deren Darstellung in x- und y-Richtung. Mit `arrow` können Linien und Pfeile gezeichnet werden, `plot` zeichnet schließlich 4 Kurven, die beiden Datensätze sind jeweils für einen Winkelbereich von 0 bis 180 Grad gemessen und werden zur besseren Veranschaulichung an der x-Achse gespiegelt. Das Ergebnis des Skripts zeigt Abbildung 33.

2.8 3D-Darstellung

Physikalische Größen und physikalische oder mathematische Funktionen sind oft von mehr als einem Parameter bzw. von mehr als einer unabhängigen Variablen abhängig. In vielen Fällen wird zur graphischen Darstellung ein Satz zweidimensionaler Kurven

[23] Dies ist der Voreinstellung für die unabhängige Variable (`dummy`). Man kann sie jederzeit z. B. mit `set dummy phi` ändern. Die aktuelle kann mit `show dummy` erfragt werden.

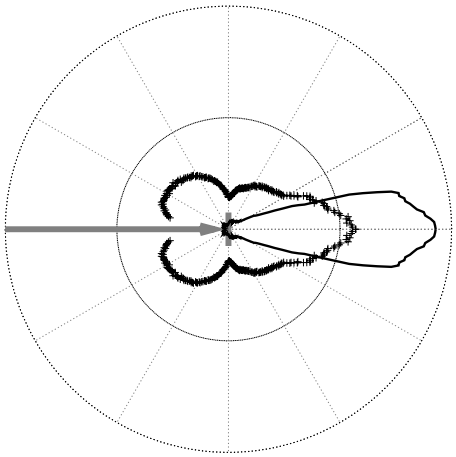


Abbildung 33: Winkelverteilung von frequenzverdoppeltem Licht aus einer Kristallitpulverprobe. Punkte: ohne Immersionsflüssigkeit, durchgezogene Linie: in Immersionsflüssigkeit. Die experimentelle Anordnung ist schematisiert eingezeichnet (grau), das Laserlicht wird in Pfeilrichtung eingestrahlt.

$y_{i,k,\ldots} = f_{i,k,\ldots}(x)$ ausreichen, die durch Angabe der Parameter $i, k, \ldots$ unterschieden werden. So kann es beispielsweise bei temperaturabhängigen optischen Spektren hinreichend sein, charakteristische Einzelspektren bei einigen typischen Temperaturen zu messen und auch so darzustellen.

Oft jedoch kann eine geeignete Darstellung über mehreren Parametern Zusammenhänge klarer machen und zu neuen 'Einsichten' führen. So könnte im Beispiel der temperaturabhängigen Spektren eine flächenartige Modellierung der Intensität über den beiden Parametern Energie und Temperatur die Temperaturabhängigkeit von Linienlagen und Linienintensitäten sehr anschaulich direkt zeigen.

Mit Gnuplot können solche Flächen als Netze im Raum dargestellt werden, die dann unter einem beliebigen Betrachtungswinkel auf die Zeichenebene projiziert werden (reine Parallelprojektion ohne die Möglichkeit einer perspektivischen Verzerrung). Darüber hinaus können Höhenlinien (*contour lines*) gezeichnet werden, entweder auf der Oberfläche selbst oder auf einer ebenen Basisfläche. Für das Zeichnen der Flächen und Höhenlinien ist die Funktion `splot` zuständig, der Betrachtungswinkel und die Vergrößerung (insgesamt und in z-Richtung) werden mit `set view` eingestellt. Ein- und ausgeschaltet wird die Darstellung der Höhenlinien und des Oberflächennetzes durch `set (no)contour` und `set (no)surface` . Die Befehle sind ausführlich in der Online-Hilfe zu Gnuplot und in der Dokumentation beschrieben.

Besonders naheliegend sind flächenartige 3D-Darstellungen, wenn ortsabhängige Größen dargestellt werden sollen, die von zwei oder drei Ortskoordinaten abhängen. Abbildung 34 zeigt als Beispiel dafür die Ortsabhängigkeit der Kristallzusammensetzung bei einem Kristall (Lithiumniobat), der auf spezielle Weise gezüchtet wurde[24]. Die Zusammensetzung wurde durch nichtkollineare Frequenzverdopplung bestimmt, einer neuen Messtechnik, die

[24] Lithiumniobat wird üblicherweise mit einem Li_2O-Gehalt von etwa 48.5 mol% hergestellt. An diesem Punkt des Phasendiagramms haben Schmelze und Kristall die gleiche Zusammensetzung. Der untersuchte Kristall wurde aus einer Schmelze mit definiertem Li-Überschuss gezüchtet, um einen höheren Li-Gehalt im Kristall zu erreichen.

in Osnabrück im Sonderforschungsbereich *Oxidische Kristalle für elektro- und magneto-optische Anwendungen* entwickelt wurde.

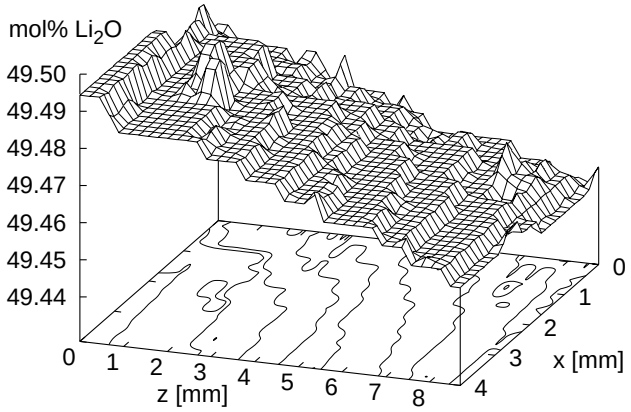


Abbildung 34: Änderung der Zusammensetzung eines Lithiumniobat-Kristalls in Ziehrichtung (z) und senkrecht dazu (x).

Man erkennt, dass der Li-Gehalt im Kristall örtlich variiert, hauptsächlich in z-Richtung, der Richtung des Kristallwachstums. Die Höhenlinien, die auf der Basisebene eingezeichnet sind, entsprechen in ihrer Form der Phasengrenze zwischen flüssiger und fester Phase während des Kristallwachstums.

Die Abbildung wird durch das folgende Skript realisiert:

```
4  X(x)=x/1000
5  Y(y)=(y-1500)/2000
6  Li(z)=(z/199000+2.9)*49.5/3
7  set dgrid3d 40,30,8
8  set nokey
9  set hidden3d;  set view 60,110,1,1.2
10 set ticslevel 0.2
11 set contour base
12 set cntrparam bspline;  set cntrparam levels auto 10
13 set xrange [ 0 : 4 ];  set yrange [ 0 : 8.7 ]
14 set zrange [ 49.44 : 49.50 ]
15 set xtics 0,1;  set ytics 0,1
16 set ztics 49.44,0.01;  set format z "%5.2f"
17 set xlabel "x [mm]" -2.5,-1;  set ylabel "z [mm]" -3,0
18 set zlabel "mol% Li_2O" -1,-1
19 splot "130695.dat" using (X($2)):(Y($3)):(Li($6)) w l
```

Der Plot wird mit der `splot`-Anweisung aus einer Datendatei generiert, alle Daten werden mit geeigneten, vorab definierten Funktionen `X()`, `Y()`, `Li()` umgerechnet (`using`). Die Rohdaten für x, y und z stehen in der zweiten, dritten und sechsten Spalte der Datei. Durch die Anweisung für `dgrid3d` wird eine Umrechnung und Interpolation der Daten auf ein reguläres Gitternetz angeordnet. Das ist notwendig, damit Gnuplot Höhenlinien berechnen kann, außerdem wird dadurch das Netz der dargestellten Fläche festgelegt. Die Höhenlinien werden auf der Basisebene gezeichnet, mit `ticslevel` wird der Abstand der Netzfläche

zur Basisebene bestimmt. Die Art der Höhenlinien wird durch `cntrparam` eingestellt. Die übrigen `set`-Anweisungen legen unter anderem die Eigenschaften und Beschriftungen der drei Achsen fest, die Zahlenangaben bei den `label`-Anweisungen verschieben die Achsenbeschriftungen um die jeweiligen Vielfachen von Zeichenbreite und -höhe.

Dreidimensionale Darstellungen sind naturgemäß auch besonders dafür geeignet, geometrische Körper oder Flächen zu visualisieren. In Kombination mit einer mathematischen Beschreibung können so physikalische Zusammenhänge effizient veranschaulicht werden. Dazu ein einfaches Beispiel aus der Mechanik: eine Kugel rollt, exzentrisch losgelassen, eine geneigte Dachrinne (*Halfpipe*) entlang. In Richtung der Rinne wird sie sich gleichmäßig beschleunigt bewegen, quer dazu etwa nach Art eines Fadenpendels, näherungsweise harmonisch. Die beiden Bewegungen überlagern sich ungestört, wenn man Reibung außer acht lässt. Die Geometrie des Problems ist in Abbildung 35 modelliert, mit eingezeichnet ist der mit den üblichen Näherungen gerechnete Weg der Kugel, eine Schwingung um die Talsohle mit zunehmender Periodenlänge und konstanter Amplitude.

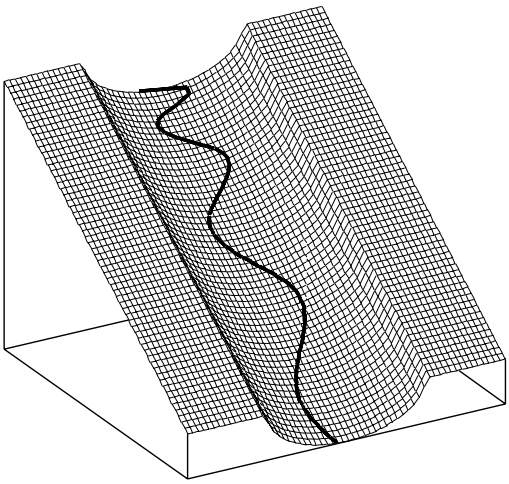


Abbildung 35: Bahn einer reibungsfrei rollenden Kugel in einer leicht geneigten 'Halfpipe'. Der Startpunkt liegt exzentrisch oberhalb der Talsohle.

Das zugehörige Gnuplot-Skript:

```
 4  ax=1;  ay=0.5;  my=0.5;  x0=3;  r=10
 5  Z(x,y)=r-((abs(x)<r)?sqrt(r*r-x*x):0)-my*y
 6  X(v)=x0*cos(ax*T(v))
 7  T(v)=sqrt(2*v/ay)
 8  set parametric
 9  set urange [-20:20];  set vrange [0:100]
10  set isosamples 60,60
11  set hidden3d;  set view 50,150,0.7,1.5
12  set noxtics;  set noytics;  set noztics
13  set nokey;  set ticslevel 0
14  splot u,v,Z(u,v) lw 0.3, X(v),v,Z(X(v),v)+0.2 lw 5
```

`Z` beschreibt die Geometrie der *Halfpipe*, in X-Richtung ein Halbkreis, in Y-Richtung konstante Neigung. `T`, das aus `Y=v` berechnet wird, und `X` parametrisieren den Weg der Kugel.

In Y-Richtung eine beschleunigte Bewegung, für `T` daher die Wurzelabhängigkeit, in X-Richtung die harmonische Schwingung. Zur Darstellung wird die X-Y-Bewegung auf die Fläche projiziert, wegen der Sichtbarkeit ein wenig angehoben (`0.2`).

Durch eine geeignete Parametrisierung lassen sich auch komplexere Flächen darstellen, beispielsweise die typischen hantelförmigen Winkelverteilungen der Elektronendichte von atomaren Wellenfunktionen. So können p-Funktionen durch Winkelabhängigkeiten des Typs

$$f_p(\theta) = \sin^2(\theta)\,, \tag{2.8}$$

d-Funktionen durch

$$f_d(\theta, \varphi) = \sin^2(2\theta)\cos^4(\varphi) \tag{2.9}$$

beschrieben werden. Abbildung 36 visualisiert die bekannten hantel- bzw. keulenförmigen Dichteverteilungen.

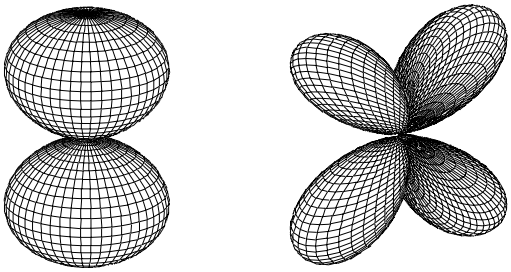


Abbildung 36: Winkelverteilung der Elektronendichte bei atomaren p- und d-Funktionen.

Auch hier das zugehörige Skript, zunächst für die p-Funktion:

```
 4  set parametric
 5  set angles degrees
 6  set noborder;  set nokey
 7  set hidden3d;  set view 60, 30, 1, 1
 8  set noxtics;  set noytics;  set noztics
 9  set urange [ 0 : 360 ];  set vrange [ -90 : 90 ]
10  set xrange [ -1 : 1 ];  set yrange [ -1 : 1 ]
11  set zrange [ -1 : 1 ];  set size square
12  set isosamples 50, 50
13  r(u,v)=sin(v)*sin(v)
14  X(u,v)=r(u,v)*cos(v)*sin(u)
15  Y(u,v)=r(u,v)*cos(v)*cos(u)
16  Z(u,v)=r(u,v)*sin(v)
17  splot X(u,v),Y(u,v),Z(u,v) lw 0.3
```

Für die d-Funktion sind zwei Zeilen zu ändern:

```
12  set isosamples 100, 100
13  r(u,v)=sin(2*v)*sin(2*v)*cos(u)**4
```

2.9 Sphärische und Zylinder-Koordinaten

Für manche Anwendungen – dann, wenn Daten vornehmlich von einem oder zwei Winkeln im Raum abhängen – kann eine räumliche Darstellung (Projektion) auf eine Kugel- oder Zylinderoberfläche sinnvoll sein. Gnuplot erleichtert dies dadurch, dass es sich auf die entsprechenden Abbildungsarten (`set mapping polar`, `set mapping cylindrical`) umschalten lässt. Daten aus Dateien werden dann als Polar-, Azimutwinkel und Radius, bzw. Polarwinkel, Höhe und Radius interpretiert und von Gnuplot in kartesische Koordinaten umgerechnet. Die Radiusangabe kann entfallen, dann wird ein Radius von 1 angenommen. Koordinaten aus Funktionen werden allerdings nicht in entsprechender Weise umgerechnet, diese werden weiterhin als kartesische interpretiert, mithin müssen auch alle drei (x,y,z) als Funktion der Winkelparameter angegeben werden.

Naheliegend ist das Arbeiten mit Winkelkoordinaten bei der Darstellung globaler geographischer Zusammenhänge. Gnuplot liefert dafür ein kurzes Demo-Skript mit, das auch einen groben Datensatz für die Umrisse der Erdteile enthält. Dieser wurde verwendet, um die Abbildung 37 zu erstellen.

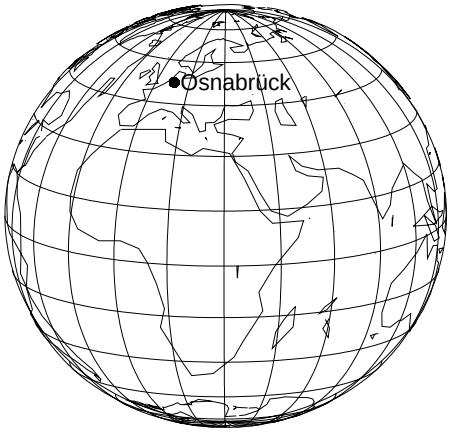


Abbildung 37: 'Osnabrücker Globus'.

Das Skript:

```
 4  X(u,v)=cos(v)*cos(u)
 5  Y(u,v)=cos(v)*sin(u)
 6  Z(u,v)=sin(v)
 7  set nokey;  set noborder
 8  set noxtics;  set noytics;  set noztics
 9  set encoding iso_8859_1
10  set angles degrees
11  set view 70,120,0.7,1.5
12  set parametric
13  set samples 100
14  set isosamples 13
15  set urange [-60:120];  set vrange [-90:90]
16  set label " Osnabrück" at X(8.2,52),Y(8.2,52),Z(8.2,52)
```

```
17  set mapping spherical
18  splot X(u,v),Y(u,v),Z(u,v)  lw 0.3,\
19   'world.dat' with lines,\
20   X(8.2,52),Y(8.2,52),Z(8.2,52) with points 7
```

Es werden zunächst die Abbildungsfunktionen von Winkelkoordinaten auf kartesische definiert, diese werden – trotz sphärischem Mapping – für alle funktional definierten Koordinaten in `splot` benötigt. Da bei der Art der Daten kein `hidden3d` möglich ist, wird `urange` passend zu `view` eingestellt, damit keine Linien von der Rückseite stören. Die `encoding`-Einstellung macht Osnabr*ü*ck möglich.

Zum Vergleich die Code-Änderungen bei Verzicht auf sphärisches Mapping, auch auf die aus der Datei `world.dat` gelesenen Winkeldaten müssen dann die Abbildungsfunktionen angewendet werden:

```
17  set mapping cartesian
18  splot X(u,v),Y(u,v),Z(u,v) lw 0.3,\
19   'world.dat' using (X($1,$2)):(Y($1,$2)):(Z($1,$2)) with lines,\
20   X(8.2,52),Y(8.2,52),Z(8.2,52) with points 7
```

Eine Anwendung aus der Physik, bei der eine sphärische Darstellung Sinn macht, ist der Verlauf der Phasenanpassungswinkel[25] bei der optischen Frequenzverdopplung in optisch zweiachsigen Kristallen. Für jede Wellenlänge der eingestrahlten Grundwelle ist in solchen Kristallen phasenangepasste Verdopplung für einen ganzen Satz von Polar- und Azimutwinkeln möglich, zumindest innerhalb eines gewissen Bereichs von Wellenlängen. Die Ortskurven für solche Winkelpaare auf einem Kugeloktanten[26] sind in Abbildung 38 dargestellt.

Gegenüber dem ‘Osnabrücker Globus’ ändert sich das Skript nur wenig, es wird ein 5°-Netz eingestellt und der Winkelbereich wird auf einen Oktanten reduziert:

```
14  set isosamples 19
15  set urange [0:90];  set vrange [0:90]
16  splot X(u,v),Y(u,v),Z(u,v)  lw 0.3, 'hobd.dat' w l lw 2
```

[25] Phasenanpassung heißt, dass sich Grundwelle und erzeugte frequenzverdoppelte Welle mit gleicher Geschwindigkeit durch den Kristall bewegen, d. h. gleiche Brechungsindizes erfahren. Da sich jedoch in jedem Kristall der Brechungsindex mit der Wellenlänge ändert (vgl. Abbildung 31), ist dies im allgemeinen nicht möglich. Die Wirkung dieser *Dispersion* lässt sich nur in doppelbrechenden Kristallen unterdrücken. Für jede Ausbreitungsrichtung gibt es in solchen Kristallen genau zwei mögliche Polarisationsrichtungen mit zwei unterschiedlichen Brechungsindizes, die von der Ausbreitungsrichtung abhängen. Man nutzt für Grundwelle und verdoppelte Welle die beiden unterschiedlichen Brechungsindizes, indem man die beiden Wellen unterschiedlich polarisiert. Die Ausbreitungsrichtung wird dann so gewählt, dass für eine Polarisation der Brechungsindex bei der Wellenlänge der Grundwelle dem für die andere Polarisation bei der Wellenlänge der verdoppelten Welle entspricht. Die Winkel dieser Ausbreitungsrichtung im Kristall nennt man Phasenanpassungswinkel.

[26] Wegen der Symmetrie (*mmm*) des Brechungsindexverlaufs ist die ausreichend.

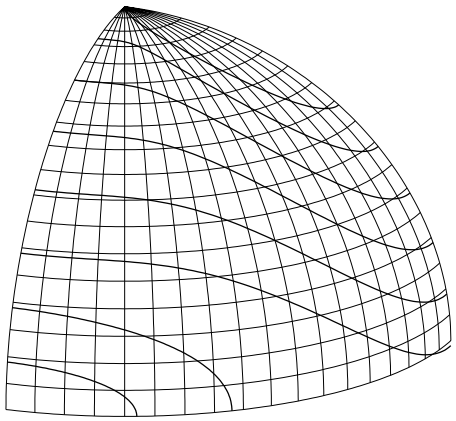


Abbildung 38: Ortskurven der Phasenanpassungswinkel für optische Frequenzverdopplung in optisch zweiachsigen Kristallen. Jede Kurve entspricht einer bestimmten festen Wellenlänge (kurzwelliger Bereich: links unten, langwelliger Bereich: rechts oben).

2.10 Vektorfelder

Eine neue, erst in der aktuellen Version implementierte Fähigkeit von Gnuplot ist die Darstellung von Vektorfeldern. In einem zweidimensionalen Plot können Vektorpfeile an Datenpunkten eingezeichnet werden, um die ortsabhängige Richtung und Größe eines Feldes zu veranschaulichen. Erforderlich sind dazu 4 Größen in der Datendatei oder in der `using`-Anweisung des `plot`-Befehls, x, y, dx und dy. Vektorpfeile werden dann mit der Option `with vector` gezeichnet. Zur Veranschaulichung ein einfaches Beispiel aus der Elektrostatik, das Feld zweier benachbarter gleich großer elektrischer Ladungen, einmal mit unterschiedlichem, einmal mit gleichem Vorzeichen. Die dreidimensionale Visualisierung des Potenzialverlaufs (Potenzial als Funktion von zwei Ortskoordinaten) zeigt Abbildung 39. Deutlich erkennbar ist die $1/r$-Abhängigkeit in der Nähe der beiden Ladungen.

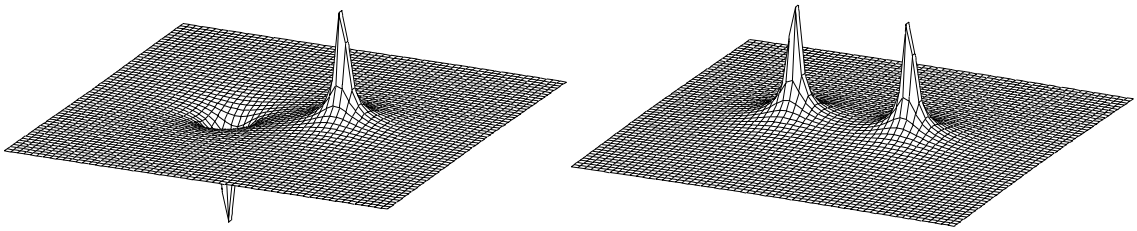


Abbildung 39: Elektrostatisches Potenzial in der Umgebung von zwei Punktladungen, links unterschiedliches, rechts gleiches Vorzeichen der Ladungen.

Eine für Physiker näherliegende Information sind die Höhenlinien (Äquipotenziallinien) und die Feldlinien in einem solchen Potenzialgebirge. Erstere waren in Gnuplot schon seit längerem mit der `contour`-Option realisierbar, jetzt können auch Feldlinien, zumindest als Feld von Vektorpfeilen, visualisiert werden (Abbildung 40).

Im zugrunde liegenden Skript werden zunächst die Funktionen definiert, die das physikalische Problem beschreiben, Potenziale und Felder werden aus Ladungen und Entfernungen berechnet:

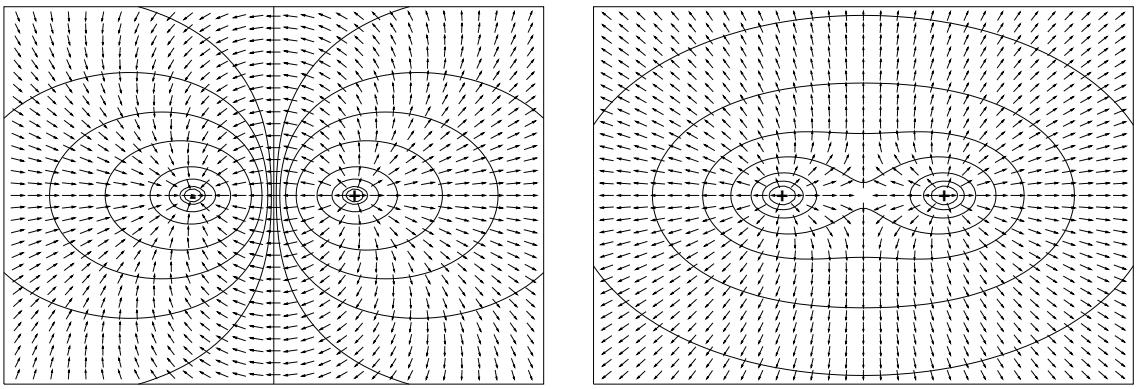


Abbildung 40: Äquipotenziallinien und Feldlinienrichtungen des elektrischen Feldes zweier gleich großer elektrischer Ladungen, links mit unterschiedlichem Vorzeichen, rechts mit gleichem Vorzeichen. Die Vektorpfeile sind auf eine einheitliche Länge normiert, zeigen also nur die Richtung, nicht die Stärke des Feldes an.

```
16  r(x,y)=sqrt(x*x+y*y)
17  v1(x,y)=  q1/(r((x-x0),y))
18  v2(x,y)=  q2/(r((x+x0),y))
19  vtot(x,y)=v1(x,y)+v2(x,y)
20  e1x(x,y)= q1*(x-x0)/r(x-x0,y)**3
21  e1y(x,y)= q1*(y)/r(x-x0,y)**3
22  e2x(x,y)= q2*(x+x0)/r(x+x0,y)**3
23  e2y(x,y)= q2*(y)/r(x+x0,y)**3
24  etotx(x,y)=e1x(x,y)+e2x(x,y)
25  etoty(x,y)=e1y(x,y)+e2y(x,y)
26  enorm(x,y)=sqrt(etotx(x,y)*etotx(x,y)+etoty(x,y)*etoty(x,y))
27  dx(x,y)=coef*etotx(x,y)/enorm(x,y)
28  dy(x,y)=coef*etoty(x,y)/enorm(x,y)
```

Dann die konkreten Parameter, hier für die Ladungen mit unterschiedlichem Vorzeichen:

```
30  coef=.5;  x0=3.;  q1=1;  q2=-1
31  xmin=-10.;  xmax=10.;  ymin=-10.;  ymax=10.
```

Die Höhenlinien werden mit der folgenden Sequenz gezeichnet, anschließend für die spätere Weiterverwendung als Tabelle in einer Datei gespeichert:

```
39  set nosurface
40  set contour base
41  set cntrparam order 4
42  set cntrparam linear
43  set cntrparam levels discrete -3,-2,-1,-0.5,-0.2,-0.1,-0.05,-0.02,\
44                                 0, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 3
```

```
45 set cntrparam points 5
46 splot vtot(x,y) w l
47 pause -1 "Now create a file with equipotential lines <CR> to continue"
48 set term table;  set out "equipo2.dat";  replot
```

Ebenfalls in einer Datei die Potenzialfläche (mit `reset` wird die Voreinstellung `surface`, `nocontour` wiederhergestellt):

```
54 reset
55 set isosam 31,31;  set samples 31
56 set term table;  set out "field2xy.dat"
57 set xr [xmin+0.5:xmax-0.5];  set yr [ymin+0.5:ymax-0.5]
58 splot vtot(x,y) w l
```

Schließlich werden mit `plot` alle gesammelten Daten als Potenziallinien und als Vektorpfeile in eine Graphik gezeichnet:

```
64 set label "-" at -x0,0 center
65 set label "+" at  x0,0 center
66 set nokey;  set noxtics;  set noytics
67 plot "field2xy.dat" using ($1-coef*dx($1,$2)):($2-coef*dy($1,$2)):\
68    (2*coef*dx($1,$2)):(2*coef*dy($1,$2)) with vector,\
69    "equipo2.dat" with lines
```

Im obigen Beispiel wird mehrfach davon Gebrauch gemacht, dass man als Ausgabegerät auch das Pseudogerät `table` definieren kann. Damit können die aktuell gezeichneten Kurvenpunkte in einer Datei (`set out ...`) abgelegt werden. Das ist beispielsweise dann notwendig, wenn Kurven (Höhenlinien o. ä.) mit einer `splot`-Anweisung generiert werden, später aber in einer `plot`-Anweisung mit eingebunden werden sollen. Darüber hinaus kann man Gnuplot auf diese Weise sehr einfach zur Umrechnung von Daten verwenden.

3 MATLAB

In erster Linie für Anwendungen in der Numerik konzipiert bietet MATLAB[27] auch eine Fülle von Visualisierungsfunktionen, die zur Erstellung von Graphiken eingesetzt werden können. MATLAB – der Name steht als Abkürzung für *Matrix Laboratory* – war ursprünglich gedacht als einfache Bedienumgebung (*Frontend*) der Mathematik-Pakete *LINPACK* [8] und *EISPACK* [9], die für das Rechnen mit Matrizen entwickelt worden waren. Inzwischen sind diese Pakete weitgehend durch *LAPACK* [10] ersetzt, das jetzt auch die Basis von MATLAB bildet. Heute ist MATLAB – aktuell in der Version 6 (Release 12) – ein sehr umfangreiches und komplexes Programmsystem, das kommerziell ständig weiterentwickelt wird. Für viele Bereiche, insbesondere auch für industrielle Anwendungen, wurden spezialisierte Zusatzpakete (*Toolboxes*) entwickelt, so für Signalverarbeitung und -analyse, Regelung, Simulation, nichtlineare Optimierung.

Im Lieferumfang von MATLAB ist eine größere Anzahl von Handbüchern enthalten, zwei davon sind auf Graphik spezialisiert: *Using MATLAB Graphics* [11] und *MATLAB Function Reference (Volume 2: Graphics)* [12]. Der Inhalt dieser Handbücher ist auch als HTML-Hilfesystem unter MATLAB verfügbar(*Help Desk*), außerdem gibt es eine sehr umfangreiche Online-Hilfe zu allen Funktionen. Bei spezielleren Problemen kann man die Homepage der Herstellerfirma [13] nach geeigneten Lösungen durchsuchen.

Zur schnellen Darstellung von mathematischen Funktionen enthält MATLAB inzwischen eine ganze Reihe von *eazy* Plotbefehlen, die alle mit `ez...` beginnen. Die zu plottende Funktion wird dabei als Text übergeben. Mit ihnen kann man auf einfache Weise zwei- oder dreidimensionale Funktionen veranschaulichen und sich so einen raschen Überblick über einen Teil der Graphik-Möglichkeiten in MATLAB verschaffen[28]. Daten – numerische oder experimentelle – zu plotten ist damit leider nicht möglich.

3.1 Einlesen von Daten

MATLAB kann eine ganze Anzahl von Dateiformaten direkt lesen, daneben sind verschiedene Funktionen implementiert, mit denen die Ein- und Ausgabe sehr flexibel gesteuert werden kann (eine einigermaßen komplette Liste der Ein-/Ausgabe-Funktionen liefert `help iofun`). Reicht dies alles nicht aus, so bleibt noch die Möglichkeit, eigene Funktionen in C/C++, Fortran oder Java[29] zu implementieren.

3.1.1 SAVE und LOAD

Die Standardfunktionen zum Schreiben und Lesen von Daten sind `save` und `load`. Ohne Argumente speichert `save` alle Variablen der aktuellen MATLAB-Sitzung binär in

[27] MATLAB ist ein eingetragenes Warenzeichen von *The MathWorks Inc.*
[28] Beispiele in der Online-Hilfe, z. B. `help ezsurf`.
[29] In Release 11 inoffiziell, ab Release 12 offiziell unterstützt.

der Datei `matlab.mat`, mit `load` wird diese Datei wieder gelesen, die Variablen werden wiederhergestellt. Durch Argumente kann man bei `save` den Namen der Datei, die zu speichernden Variablen, die Formatierung und die Datengenauigkeit vorgeben. So wird MATLAB mit

```
save  <fname>  X Y Z  -ASCII  -DOUBLE
```

angewiesen, die Variablen (Matrizen) `X`, `Y` und `Z` in der Datei `<fname>` in Textformat mit hoher Genauigkeit (16 Nachkommastellen) zu speichern.

In ähnliche Weise lassen sich auch bei `load` nähere Spezifikationen angeben. Durch Leerzeichen getrennte Zahlen aus einer Textdatei `test.dat` können so mit

```
Z = load('test.dat')
```

in die Matrix `Z` gelesen werden. Deren Spalten- und Zeilenzahl entspricht der Anordnung der Daten in der Textdatei.

3.1.2 DLMREAD, DLMWRITE und TEXTREAD

Die Funktionen `dlmread` und `dlmwrite` können Dateien lesen und schreiben, in denen numerische Variable durch ein bestimmtes Trennzeichen (*delimiter*) voneinander getrennt sind. Dies ist insbesondere interessant für den Datenaustausch mit Tabellenkalkulations- oder Datenbankprogrammen. Voreingestellt ist ein Komma als Trenner. Durch weitere Argumente lässt sich bei diesen Funktionen der Bereich einschränken, um beispielsweise in Tabellen enthaltenen Text zu überlesen (Spalten- und Zeilenüberschriften). So wird mit

```
Z = dlmread('data.txt', '\t', 2, 1)
```

der Inhalt der Datei `data.txt` in die Matrix `Z` gelesen, Trenner ist das Tabulatorzeichen, begonnen wird bei der dritten Zeile und zweiten Spalte, d. h. Titel sowie Zeilen- und Spaltenüberschriften werden überlesen.

Enthält die zu lesende Datei Text, der mit eingelesen werden soll, kann dazu die Funktion `textread` verwendet werden, die sehr flexible, C-ähnliche Formatangaben ermöglicht.

```
[name,x,y,a] = textread('data.txt', '%s %f %f %d')
```

liest die Datei `data.txt` und interpretiert die erste Spalte als Zeichenkette, die zweite und dritte als reelle, die vierte als Ganzzahl. Die Spalten werden in den Matrizen `name`, `x`, `y` und `a` abgelegt.

```
linearray = textread('test.c','%s','delimiter','\n','whitespace','')
```

legt die ganze Datei `test.c` zeilenweise in `linearray` ab.

3.1.3 Spezielle Dateiformate

Für einige Dateiformate sind in MATLAB eigene Schreib- und Lesefunktionen implementiert:

WK1READ und WK1WRITE lesen und schreiben Lotus 1,2,3 kompatible Dateien.

XLSREAD dient dazu, Excel-Dateien zu lesen. Als zweiter Parameter (der erste ist der Dateiname) kann der Name des Tabellenblatts angegeben werden, ansonsten wird das erste gelesen. Als Ergebnis liefert XLSREAD zwei Felder, ein rein numerisches und eines mit den Textinhalten des Tabellenblatts.

IMREAD und IMWRITE lesen und schreiben Bilddateien in den Formaten JPEG[30], TIFF[31], BMP[32], PNG[33], HDF[34], PCX[35] und XWD[36].

AUREAD und AUWRITE lesen und schreiben Sun Audio Dateien.

WAVREAD und WAVWRITE lesen und schreiben Windows Wave Audio Dateien.

AVIREAD, AVIFILE und MOVIE2AVI sind Funktionen, die zusammen mit weiteren wie ADDFRAME die Erstellung und Bearbeitung von Video-Sequenzen in MATLAB ermöglichen.

Näheres dazu in der Online-Hilfe zu den genannten Funktionen.

3.1.4 Internet, Dateisystem, XML

Mit URLREAD können Textdateien aus dem Internet gelesen werden, als Parameter wird die *URL* angegeben, `http://...`, `ftp://...` oder auch lokal `file://...`.

URLWRITE schreibt nicht etwa auf eine Datei im Internet, sondern kopiert den Inhalt einer Internet-Datei in eine lokale (zweiter Parameter ist der Dateiname der lokalen Datei). Damit lassen sich auch binäre Dateien aus dem Netz verarbeiten.

Zur Manipulation von Dateinamen verfügt MATLAB über eine ganze Anzahl von Funktionen:

FILEPARTS zelegt einen Dateinamen in seine Bestandteile.

FULLFILE setzt ihn wieder daraus zusammen.

[30] Joint Photographic Experts Group.
[31] Tagged Image File Format.
[32] Windows Bitmap.
[33] Portable Network Graphics.
[34] Hierarchical Data Format.
[35] Windows Paintbrush.
[36] X Window Dump.

FILESEP liefert den fürs Dateisystem zuständigen Separator (‘\’ in Windows).

TEMPDIR und TEMPNAME ermöglichen das Arbeiten mit vorübergehend benötigten temporären Dateien. Mit DELETE lassen sich die wieder löschen, wenn sie nicht mehr benötigt werden.

Schließlich kann MATLAB seit kurzem auch mit XML-Dateien umgehen. Dafür sind die Funktionen XMLREAD (Lesen), XMLWRITE (Schreiben) und XSLT (Übersetzen eines XML-Dokuments mittels einer XSL-Vorschrift in eine andere Form) zuständig.

3.1.5 Systemnahe Funktionen für Text- und Binärdateien

Neben den bisher beschriebenen *High-Level*-Routinen sind in MATLAB C-ähnliche systemnahe *Low-Level*-Funktionen zum Lesen und Schreiben von Dateien definiert, mit denen man die Ein- und Ausgabe sehr flexibel erledigen kann. Für den Dateizugriff sind das:

fopen: Öffnet eine Datei zum Lesen oder Schreiben, gibt einen File-ID zurück.

ftell: Informiert über die aktuelle Position in der Datei.

fseek: Positioniert neu.

fclose: Schließt die Datei.

Zum Lesen und Schreiben von Textdateien benutzt man:

fscanf: Liest formatierte Daten aus einer Textdatei.

fprintf: Schreibt formatierte Daten in eine Textdatei.

fgetl: Liest eine Zeile aus einer Textdatei.

Zum binären Lesen und Schreiben sind noch zusätzlich definiert:

fread: Liest Binärdaten.

fwrite: Schreibt Binärdaten.

Genauere Informationen zu diesen Funktionen finden sich in der Online-Hilfe.

3.1.6 Bibliotheksfunktionen zum Datenaustausch

Neben den bisher beschriebenen Funktionen zum Lesen und Schreiben von Dateien stellt MATLAB verschiedene Software-Schnittstellen zur Verfügung, die von externen Programmen verwendet werden können. Dabei sind drei unterschiedliche Hierarchie-Szenarien möglich:

MATLAB und externes Programm auf gleicher Ebene: Der Datenaustausch erfolgt über symmetrische Konzepte. Die beiden Programme tauschen Daten durch MAT-Dateien oder per DDE bzw. ActiveX aus.

MATLAB ruft externe Routinen auf: Die externen Funktionen können in C/C++ oder Fortran programmiert werden (*MEX*-Funktionen), von MATLAB aus werden die kompilierten Funktionen wie MATLAB-Funktionen aufgerufen. Die Schnittstelle zur Parameterübergabe ist standardisiert, darüber hinaus können Daten durch Bibliotheksfunktionen ausgetauscht werden. Neben dieser schon betagten Standardschnittstelle *MEX* für kompilierte Objekte bietet MATLAB seit kurzem auch die Möglichkeit, Java-Objekte innerhalb und außerhalb von MATLAB zu verwenden.

Das externe Programm ruft MATLAB auf: Soll das externe Programm Priorität haben, lässt man MATLAB unter dessen Kontrolle ablaufen. Das Programm startet eine MATLAB-*Engine*, kann Daten damit austauschen, Anweisungen an MATLAB schicken, schließlich MATLAB wieder schließen.

Für die einzelnen Bereiche sind jeweils spezielle Funktionsbibliotheken und Definitionsdateien zuständig, die jeweiligen Funktionen und Datentypen sind durch Präfixe gekennzeichnet:

mat ist das Präfix von Funktionen, die das Lesen und Schreiben von MAT-Dateien betreffen, die zugehörige Definitionsdateien sind `mat.h` und `libmat.def`.

mex das für Funktionen, die in MEX-Programmen verwendet werden können (`mex.h`, `libmex.def`),

eng das für Funktionen, die mit der MATLAB-*Engine* zusammenarbeiten (`engine.h`, `libeng.def`).

mx schließlich kennzeichnet alle Funktionen und Definitionen, die zu den MATLAB-Arrays gehören (`matrix.h`, `libmx.def`).

Sollen solche Funktionen in C/C++-Programmen verwendet werden, wird wie üblich die Header-Datei mit `#include` eingebunden, dabei ist darauf zu achten, dass der Include-Pfad entsprechend ergänzt wird. Aus den DEF-Dateien werden LIB-Dateien generiert[37], die ihrerseits auf die zuständigen DLLs hinweisen (Windows).

[37] Die zuständige Anweisung: `lib /def:"d:/programme/matlabR12/extern/include/libeng.def" /MACHINE:IX86 /OUT:_libeng.lib` .

3.1.7 MAT-Dateien

MATLAB verwendet ein eigenes spezielles Format, um Daten zu speichern – MAT-Dateien. Damit solche Dateien auch von Fremdprogrammen auf einfache Weise erstellt und gelesen werden können, stellt MATLAB in der Datei `libmat.dll` die dafür benötigten Funktionen bereit. Die folgenden Programmfragmente skizzieren das Schreiben und Lesen von MAT-Dateien durch ein C/C++-Programm.

Das Schreiben einer MAT-Datei kann etwa durch die folgende Sequenz erledigt werden:

```
#include "mat.h"
MATFile *pMat = matOpen("mattest.mat", "w");
mxArray *pA = mxCreateDoubleMatrix(3, 3, mxREAL);
double *pD = mxGetPr(pA);
mxSetName(pA, "VarName");
...                    // fill array
matPutArray(pMat, pA);
mxDestroyArray(pA);
matClose(pMat);
```

Eine so – oder aber auch von MATLAB – erstellte Datei wird vom externen Programm dann wieder gelesen mit:

```
#include "mat.h"
MATFile *pMat = matOpen("mattest.mat", "r");
mxArray *pA = matGetArray(pMat, "VarName");
matClose(pMat);
double *pD = mxGetPr(pA);
if (mxGetNumberOfDimensions(pA) != 2) { DimensionERROR(); }
...                    // get data
mxDestroyArray(pA);
```

Die Namen von Funktionen und Typen sind ausreichend deskriptiv, so dass sie hier nicht näher diskutiert werden müssen. Man beachte die Unterscheidung zwischen `mat...`- und `mx...`-Namen für Anweisungen bzw. Typen, die einerseits MAT-Dateien, andererseits die Matrix-Darstellung im Speicher betreffen.

3.1.8 MEX-Funktionen

Externe C/C++- oder Fortran-Funktionen können von MATLAB wie eingebaute Funktionen aufgerufen werden, wenn sie gewisse Formalitäten einhalten. Solche externen Funktionen (MEX-Funktionen) müssen als dynamisch ladbare Objekte kompiliert sein – unter Windows als DLLs – und eine Schnittstelle mit festgelegtem Funktionsnamen – `mexFunction` – exportieren.

MEX-Funktionen können unter anderem dazu verwendet werden, um

- vorhandene Programme oder Bibliotheken zu integrieren, ohne sie zu m-Dateien umschreiben zu müssen,
- Geschwindigkeitsprobleme zu lösen (unvermeidbare `for`-Schleifen o. ä.),
- Datenerfassungs- oder Steuerungshardware anzusprechen,
- Benutzer-Schnittstellen zu realisieren.

Bevor man MEX-Funktionen verwendet, sollte man auf jeden Fall prüfen, ob das gewünschte Ergebnis nicht auch mit MATLAB alleine zu erreichen ist. Schließlich wurde MATLAB dazu entwickelt, umfangreiche Programmierung in 'niedrigeren' Programmiersprachen zu vermeiden.

Die einfachste Art, MEX-Funktionen zu kompilieren, ist es, die MATLAB-Anweisung `mex` zu verwenden. Beim ersten Mal muss dabei `mex -setup` verwendet werden, um die Grundeinstellungen (Kompiler etc.) festzulegen. Danach ist `mex` funktionsfähig. Beim Aufruf wird das Perl-Skript `mex.pl` im MATLAB-Verzeichnis aufgerufen, das alles Nötige erledigt. Die Quelldatei für die MEX-Funktion wird mit einem Editor (z. B. dem MATLAB-Editor) im Arbeitsverzeichnis erstellt.

MATLAB kommuniziert mit einer MEX-Datei über die Interface-Funktion `mexFunction()`, die mithin in jeder MEX-Datei vorhanden sein muß. Diese Funktion hat die Signatur

```
void mexFunction ( int nlhs, mxArray *plhs[],
                   int nrhs, const mxArray *prhs[] ),
```

die 4 Parameter beschreiben die Eingabe- und Ausgabeobjekte der MEX-Funktion (*l*eft *h*and *s*ide, *r*ight *h*and *s*ide). Die Felder `prhs` und `plhs` enthalten Zeiger auf die Objekte, `nrhs` und `nlhs` sind deren Anzahl. Bei einer MEX-Funktion `func.dll`, die aus MATLAB mit

```
[u, v] = func (a);
```

aufgerufen wird, ist `nlhs = 2`, `plhs[0]` zeigt auf `u`, `plhs[1]` auf `v`, `nrhs = 1`, `prhs[0]` zeigt auf `a`.

Der C++-Quelltext einer MEX-DLL, die Daten von einem Oszilloskop übernimmt und an MATLAB übergibt, enthält etwa die folgenden Zeilen. Die MEX-Funktion wird mit einem Parameter (Oszilloskop-Kanal) oder ohne (Kanal = 1) aufgerufen und gibt ein eindimensionales Messdatenfeld zurück.

```
#include "mex.h"
#include "scope.h"
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
```

```
    const int XSIZE = 2500;
    scope osc;
    osc.setchannel((nrhs==0) ? 1 : (int)*mxGetPr(prhs[0]));
    plhs[0] = mxCreateDoubleMatrix(XSIZE, 1, mxREAL);
    double * z = mxGetPr(plhs[0]);
    osc.data(z, XSIZE);
} .
```

Die an die MEX-Funktion übergebenen Daten brauchen nur gelesen zu werden, `mxGetPr` generiert einen Zeiger auf den Realteil. Für die Rückgabedaten dagegen muss von der MEX-Funktion zunächst der benötigte Speicher alloziert werden (`mxCreateDoubleMatrix`).

Als Beispiel werden drei Kanäle eines Oszilloskops ausgelesen, an die jeweils passive 10:1-Tastköpfe angeschlossen sind. Die typische Ersatzschaltung eines solchen Tastkopfs ist in Abbildung 41 dargestellt. Der Widerstand mit 9 MΩ und die Kapazität von 15 pF sind

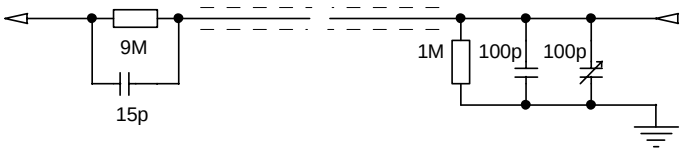


Abbildung 41: Ersatzschaltbild eines 10:1-Tastkopfs.

direkt in der Tastkopfspitze eingebaut, 1 MΩ und die feste Kapazität von 100 pF sind Eingangswiderstand und -kapazität des Oszilloskops sowie die Kapazität des Kabels. Da diese Kapazitäten variieren können, ist ein zusätzlicher variabler Kondensator zur Kompensation parallelgeschaltet. Er dient dazu, ein reelles frequenzunabhängiges Teilerverhältnis zu realisieren – im Beispiel müsste 35 pF eingestellt werden. Diese Einstellung kann mit einem Rechtecksignal überprüft werden. Abbildung 42 zeigt typische Formen dieses Rechtecksignals für drei unterschiedliche Einstellungen der variablen Kapazität (zu groß, zu klein, richtig).

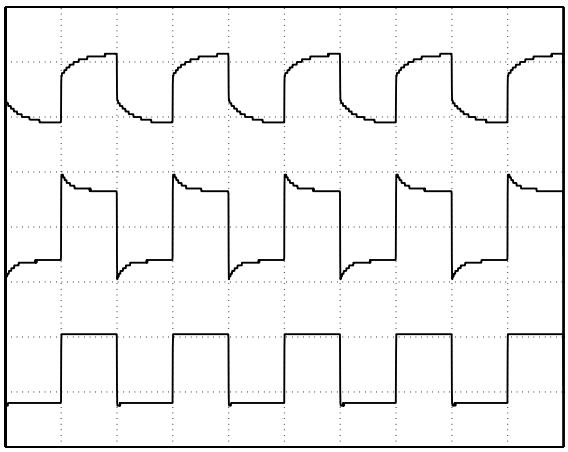


Abbildung 42: Mit Tastköpfen gemessenes Rechtecksignal. Oben: Einstellkapazität zu groß, Mitte: Kapazität zu klein, unten: richtige Einstellung.

Die Datenerfassung und Signaldarstellung kann durch die folgenden Anweisungen erledigt werden:

```
probesignal = [func(1) func(2) func(3)];
plot(probesignal,'xgrid','on','ygrid','on',...)
```

Die MEX-Funktion `func` wird dreimal aufgerufen, um die drei Erfassungskanäle des Oszilloskops zu übertragen. Der Einfachheit halber werden alle Daten in einer Feldvariablen `probesignal` abgelegt, die dann auch zum Plotten verwendet wird.

3.1.9 MATLAB als 'Engine'

Wenn das externe Programm die Hauptrolle spielen sollen, MATLAB nur Hilfsfunktionen übernehmen soll oder in einer zeitlichen Abfolge zunächst mit dem externen Programm, dann mit MATLAB gearbeitet werden soll, kann es sinnvoll sein, die Prioritäten umzukehren, das heißt, MATLAB vom externen Programm aus aufzurufen. Dazu bietet MATLAB die Möglichkeit, sich als *Engine* aufrufen zu lassen. Das nachstehende Beispiel soll die Vorgehensweise illustrieren.

In einem Programm, das Pixeldaten aus einer Bilddatenquelle (Scanner, Kamera) über die Twain-Schnittstelle übernimmt, soll es auch möglich sein, diese Daten direkt an MATLAB zu übergeben und MATLAB zur Weiterverarbeitung aufzurufen. Die Bilddaten werden innerhalb des Programms als *Device Independent Bitmap* (DIB) gehalten, aus dieser Struktur müssen sie in ein dreidimensionales MATLAB-Array umkopiert und dann an MATLAB übergeben werden. Die Funktion `Transfer2Matlab`, die das erledigt, könnte etwa so aussehen:

```
#include "engine.h"
...
void Transfer2Matlab(HGLOBAL _hDIB)
{
  BITMAPINFOHEADER *pBIH = NULL;
  unsigned char * pBM;

  UINT row, col, plane;
  UINT width, height, planes=3;
  UINT linesize;

  Engine * en;
  mxArray * pA;
  unsigned char * pD;
  int dims[3];

  if(!_hDIB)  return;

  if (pBIH = (BITMAPINFOHEADER*)GlobalLock(_hDIB))
  {
```

```
    if (!(pBIH->biBitCount==24))  return;

    dims[0] = height = pBIH->biHeight;
    dims[1] = width = pBIH->biWidth;
    dims[2] = planes;
    linesize = ((width*pBIH->biBitCount+31)/32)*4;
    pBM = (unsigned char *)pBIH + sizeof(BITMAPINFOHEADER)
        + pBIH->biClrUsed * sizeof(RGBQUAD);

    pA = mxCreateNumericArray(3, dims, mxUINT8_CLASS, mxREAL);
    mxSetName(pA, "bild");
    pD = (unsigned char *) mxGetPr(pA);

    for (row = 0; row<height; row++)
      for (col=0; col<width; col++)
        for (plane=0; plane<planes; plane++)
          *(pD+plane*height*width+col*height+row) =
            *(pBM+(height-1-row)*linesize+col*planes+planes-1-plane);

    en = engOpen(NULL);
    engPutArray(en, pA);
    mxDestroyArray(pA);
    engEvalString(en, "image(bild);");
  }
  GlobalUnlock(_hDIB);
}
```

Als Variablen werden drei Gruppen benötigt:

- Bitmap-spezifische,
- solche, die für Größenangaben und als Laufvariable gebraucht werden,
- MATLAB-spezifische.

Im Ausführungsteil werden aus der am Beginn der DIB stehenden `BITMAPINFOHEADER`-Struktur die benötigten Informationen wie Höhe und Breite des Bildes entnommen, dann wird ein dreidimensionales MATLAB-Array mit `mxCreateNumericArray` in der richtigen Größe (`dims`) angelegt, einen Zeiger auf den Beginn des eigentlichen Datenfeldes liefert `mxGetPr`. Schließlich werden die Daten umkopiert und an die mit `engOpen` gestartete MATLAB-*Engine* übergeben. MATLAB wird nicht beendet (dafür wäre `engClose` zuständig), somit kann sofort in MATLAB weitergearbeitet werden. Die etwas unübersichtliche Zeiger-Berechnung zum Umkopieren der Daten kann man sich an Abbildung 43 klarmachen.

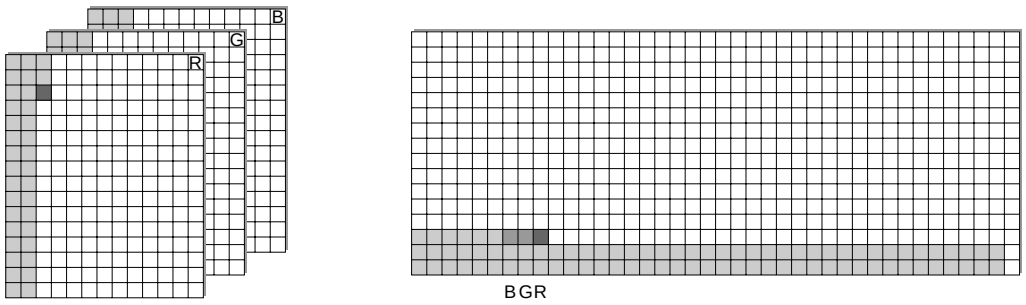


*Abbildung 43: Byte-Anordnung in einem MATLAB-Bild-Array (links) und in einer 'Device Independent Bitmap' (rechts), jeweils für 24-Bit-Farbdarstellung (True Color). Die drei Farben (R, G, B) sind in MATLAB auf 3 Sub-Arrays verteilt, in der Bitmap dagegen für jedes einzelne Pixel zusammenhängend angeordnet (Byte-Anordnung R-G-B, R höchstwertiges Byte). Im Beispiel ist eine Bildgröße vom 13*16 Pixel dargestellt, in MATLAB ist das ein 16*13*3-Feld. Die Zeilenlänge einer 'DIB' beträgt immer ein ganzzahliges Vielfaches von 4, daher die zusätzliche 40. Spalte ganz rechts und damit die Feldgröße 40*16.*

3.2 Graphik-Objekte, Handles

Zur formalen Beschreibung von Graphiken verwendet MATLAB eine hierarchische, baumartige Objektstruktur. Einen Überblick gibt Abbildung 44. Die Basis dieser Hierarchie ist

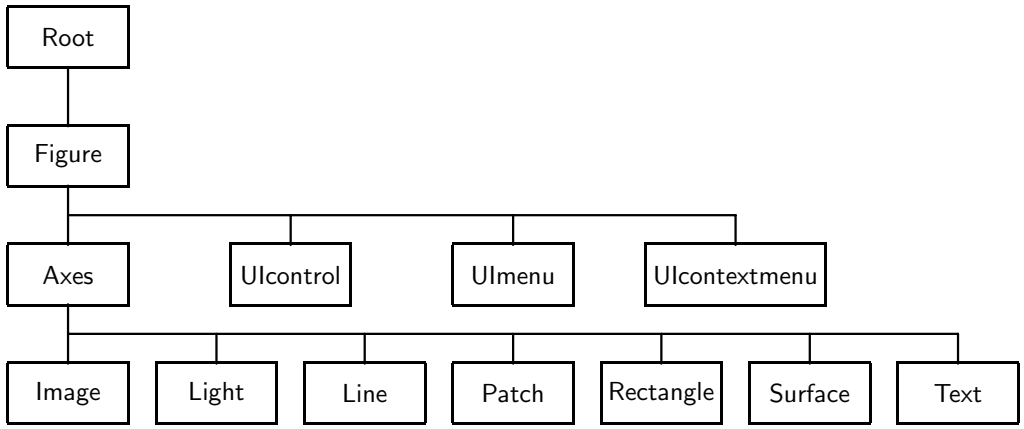


Abbildung 44: Hierarchie der Graphik-Objekte in MATLAB.

das *Root*-Objekt, der Bildschirm. Dieser kann ein oder mehrere *Figure*-Objekte enthalten, eigenständige Graphik-Fenster. Die *Figure*-Objekte wiederum enthalten neben *User Interfaces* (Bedienelemente) einen oder mehrere Koordinatenrahmen (*Axes*-Objekte), in denen die eigentlichen Graphik-Objekte platziert sind. Diese Objekte werden durch Graphik-Anweisungen wie `plot` oder `image` erstellt, die übergeordneten Objekte werden dabei automatisch miterstellt. Nachfolgende Anweisungen zur Realisierung von neuen Objekten

löschen in der Regel die vorhandenen, die kann durch `hold on` verhindert werden. Ebenso kann man durch zusätzliche Graphik-Fenster (Anweisung `figure`) oder zusätzliche Koordinatenrahmen (Anweisung `axes`) dafür sorgen, dass mehrere Objekte gleichzeitig auf dem Bildschirm dargestellt werden.

Die Eigenschaften oder Attribute (*Properties*) all dieser Objekte sind auf sinnvolle Standardwerte voreingestellt, andere Werte können direkt bei der Erstellung oder auch nachträglich vergeben werden. Direkt bei der Graphik-Erstellung werden die Eigenschaften durch die Parameterliste der Graphik-Anweisung definiert. Dort kann eine beliebige Anzahl von Paaren aus *PropertyNames* und *PropertyValues* angegeben werden. So kann beispielsweise bei der `plot`-Anweisung die Linienstärke, Symbolgröße und -farbe eingestellt werden:

```
plot(x, y, '-rs', 'Linewidth', 2, 'Markersize', 8, ...
     'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'g')
```

zeichnet die Punkte (x, y) verbunden durch eine rote durchgezogene Linie und markiert mit quadratischen Symbolen. Linienbreite, Symbolgröße, Symbolrand- und -flächenfarbe sind gegenüber der Voreinstellung geändert.

Um nachträglich auf Graphik-Objekte zugreifen zu können, verwendet MATLAB spezielle Variable, die *Handles*[38] (wörtliche Übersetzung: 'Griff' oder 'Stiel'). In den beiden obersten Hierarchieebenen sind das ganze Zahlen – *Root* hat die Handle 0, die *Figure*-Fenster die Handles 1, 2, 3, ... – bei den übrigen Objekten reelle Zahlen. Den Wert der Handles stellt MATLAB bei jeder Graphik-Anweisung als Rückgabewert bereit. Zugehörige Eigenschaften lassen sich dann mit `set` festlegen oder mit `get` erfragen. Obiges Beispiel könnte man also auch so formulieren:

```
hLine = plot(x, y, '-rs');
set(hLine, 'Linewidth', 2);
set(hLine, 'Markersize', 10);
...
```

Mit `w = get(hLine, 'Linewidth')` könnte anschließend die Linienbreite erfragt werden.

Eine Liste aktuell festgelegten Eigenschaften erhält man mit `get(<handle>)`, die möglichen Eigenschaftswerte werden durch `set(<handle>)` angezeigt.

Ist die Handle eines Objekts nicht bekannt, kann sie mit `findobj` konkretisiert werden. Diese Funktion liefert einen Vektor mit den Handles aller Objekte, die den Abfragebedingungen genügen.

`h=findobj('Color','r')` liefert die Handles aller roten Objekte,

`h=findobj(gcf, 'Type', 'line')` die Handles aller Linienobjekte im aktuellen Graphik-Fenster.

[38] 'Handle Graphics' ist ein eingetragenes Warenzeichen von *The MathWorks Inc.*

`h=findobj(<handle>, 'PropName', PropValue)` beschränkt die Suche auf das Objekt `<handle>` und die darin enthaltenen Objekte.

Die Handle des aktuellen Graphikfensters erhält man durch `gcf` (*G*et *C*urrent *F*igure), die des aktuellen Koordinatensystems durch `gca` (*G*et *C*urrent *A*xis), die des aktuellen Objekts durch `gco` (*G*et *C*urrent *O*bject).

Die Eigenschaften der Objekte im Graphik-Fenster können in den neueren MATLAB-Versionen sehr komfortabel mit dem *Property Editor* verändert werden (Abbildung 45 zeigt das für das *Axes*-Objekt zuständige Editor-Fenster). Diese 'Point-and-Click'-Oberfläche ermöglicht einen sehr einfachen interaktiven Zugang zu den Objekt-Eigenschaften.

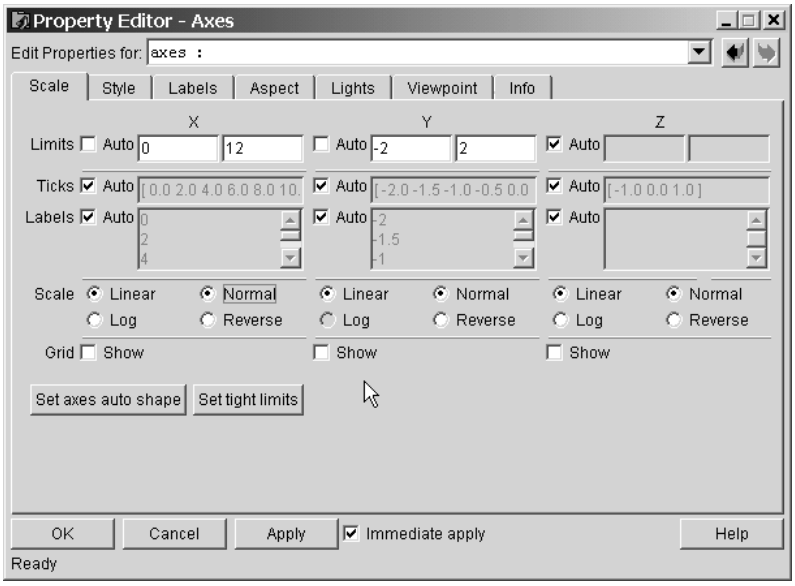


Abbildung 45: Die Property-Editor-*Oberfläche für das* Axes-*Objekt. Eigenschaftsgruppen (Scale, Style,. . .) sind über die einzelnen Karteikarten zugänglich, über das Listenfeld ganz oben erreicht man die weiteren Objekte in der Hierarchie.*

Ein gewisses Problem ist allerdings die Reproduzierbarkeit. Die fertiggestellte Graphik kann zwar binär als `FIG`-Datei gespeichert und daraus auch wieder erstellt werden, jedoch nicht als Textdatei. Dadurch wird es z. B. schwierig, gleichartige Graphiken aus unterschiedlichen Datensätzen zu realisieren. Man kann sich allenfalls damit behelfen, dass man die aktuellen Einstellungen mit der `diary`-Anweisung über eine Folge von `get`-Kommandos protokolliert, und diese Einstellungen dann für weitere Graphiken verwendet (`set`). Wenn Reproduzierbarkeit ein wichtiger Aspekt ist, sollte man besser versuchen, die komplette Graphik über eine `m`-Datei zu erstellen.

3.3 2D-Plots, Beschriftung

MATLAB bietet eine Vielzahl von Funktionen für die Erstellung und Bearbeitung von Graphiken, einen Überblick gibt die Liste *Plotting and Data Visualization* im Kapitel *Functions by Category* des Reference Manuals [14]. Mit den dort aufgeführten Funktionsnamen sollte man die Online-Hilfe für weitere Informationen zu Rate ziehen. Im Rahmen des Skripts können die Möglichkeiten mit einigen Beispielen nur angedeutet werden.

3.3.1 Linienplots

Wie alle MATLAB-Funktionen sind auch die Plot-Befehle der MATLAB-Graphik vektor- bzw. matrixorientiert. Der Basisbefehl `plot` erwartet mithin als Daten geeignete Felder,

```
h = plot(x,y,'k-')
```

generiert einen Linienplot aus den Werten in den Feldern `x` und `y` und liefert als Rückgabewert eine *Handle*-Variable, die auf das Objekt verweist. Die Angabe über den Linientyp (`k` = blac*K*, `-` = durchgehend) kann weggelassen werden, ausführlicher sein (Eigenschafts-Werte-Paare[39] wie z. B. `'Color',0.8*[1 1 1]` für hellgrau) oder anschließend mit

```
set(h,'Color',0.8*[1 1 1],'Linewidth',1.5,...)
```

nachgeliefert werden. Die Felder `x` und `y` können ein- oder zweidimensional sein – auch unterschiedlich eins ein-, das andere zweidimensional, die Feldgrößen müssen zueinander passen.

Wird nur ein Parameter angegeben, wird er als `y`-Wertevorrat über fortlaufenden `x`-Werten interpretiert. Sind diese Daten allerdings komplex, wird Imaginär- über Realteil aufgetragen. Einen Überblick über die von der Parameterangabe abhängige Interpretation von Real- und Imaginärteil bei komplexen Daten gibt Abbildung 46. Die Datenfelder dafür wurden erstellt mit:

```
x=linspace(0,10*pi,200);  y=exp((i-0.1)*x); .
```

Sollen mehrere Linien geplottet werden, können die entsprechenden Parameterpaare bzw. -triple zusammen in eine Plot-Anweisung geschrieben werden, es wird dann eine entsprechende Anzahl von Handles zurückgeliefert.

```
[h1 h2 ...] = plot(x1,y1,'g:',x2,y2,'or',...);
```

plottet das erste Datenfeld als grüne gepunktete Linie, das zweite mit roten Kreissymbolen. Übersichtlicher wird es, wenn man mit einzelnen Anweisungen arbeitet, dabei mit `hold` dafür sorgt, dass der Vorgängerplot nicht gelöscht wird:

```
h1 = plot(x1,y1,'g:');
hold on;
```

[39] Wie schon im Abschnitt *Objektorientierte Graphik* erläutert, erhält man eine Liste der aktuellen Werte mit `get(h)`, eine Liste der möglichen mit `set(h)`.

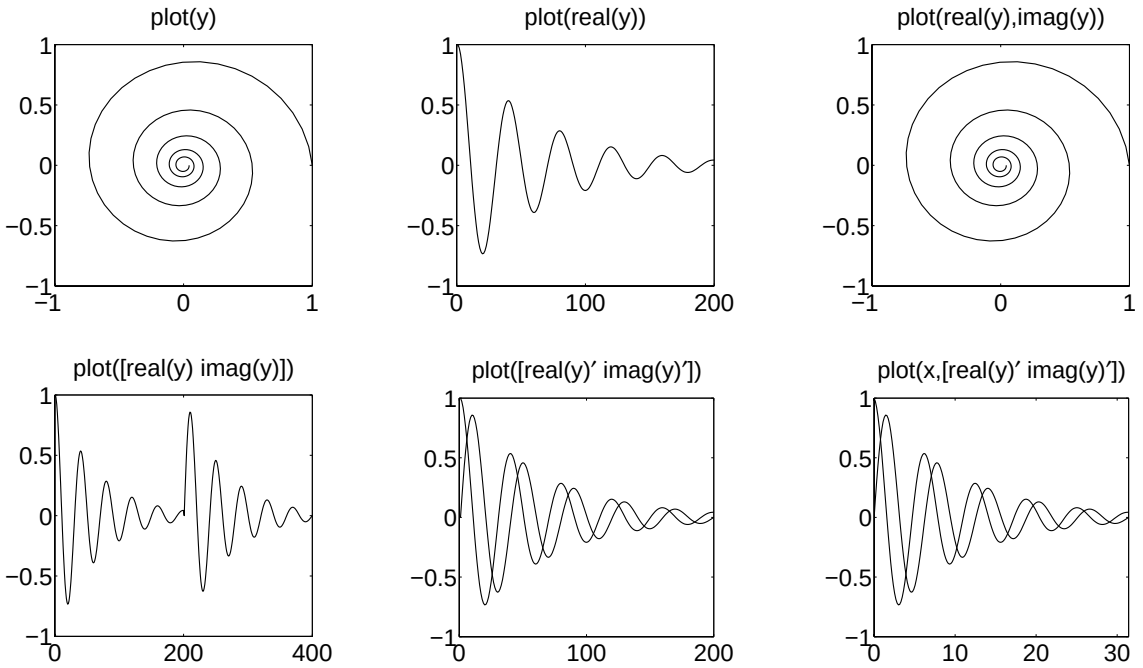


Abbildung 46: Unterschiedliche Interpretation von Real- und Imaginärteil beim Plotten von komplexen Daten.

```
h2 = plot(x2,y2,'or');
h3 = ...
hold off;
```

oder – ausführlich selbstdokumentierend:

```
h1 = plot(x1,y1);
hold on;
h2 = plot(x2,y2);
set(h1,'Color','g','Linestyle',':');
set(h2,'Color','r','Marker','o');
set([h1 h2],'Linewidth',2);
hold off; .
```

Ist (mindestens) eins der Felder zweidimensional, wird für jede seiner Spalten eine Linie gezeichnet. Als Beispiel dafür die zur Fouriersynthese einer Rechteckfunktion notwendigen Sinuskomponenten. Die Fourierreihe für eine solche Rechteckfunktion mit der Frequenz f lautet:

$$U_R(t) = \sum_{n=1,3,5,\ldots}^{\infty} \frac{1}{n} \sin(2\pi n f t)\,. \tag{3.1}$$

Eine Konkretisierung in MATLAB könnte etwa so aussehen:

```
basefreq=1000;
T=2/basefreq;
oddnums=[1:2:99];
npoints=400;
time=linspace(0,T*(1-1/npoints),npoints);
[f,t]=meshgrid(oddnums*basefreq,time);
coeff=f.^(-1)*basefreq;
z=-j*coeff.*exp(j*2*pi*f.*t); .
```

`z` ist ein zweidimensionales Amplitudenfeld über den Variablen `f` und `t`, Frequenz und Zeit, der Realteil entspricht den Sinus-Komponenten aus Gleichung 3.1. Mit

```
plot(time,real(z(:,1:n)))
```

werden die ersten n Komponenten zusammen über der gleichen Zeitachse geplottet, das Ergebnis zeigt Abbildung 47 links. Beim `plot`-Befehl ist die x-Achse nicht besonders ausgezeichnet, die beiden Parameter (ein- und zweidimensionales Feld) können auch vertauscht werden (Abbildung 47 rechts). Der allgemeine Fall sind zwei zweidimensionale Felder mit jeweils N Spalten und M Reihen als Parameter, dann werden N unabhängige Kurven gezeichnet.

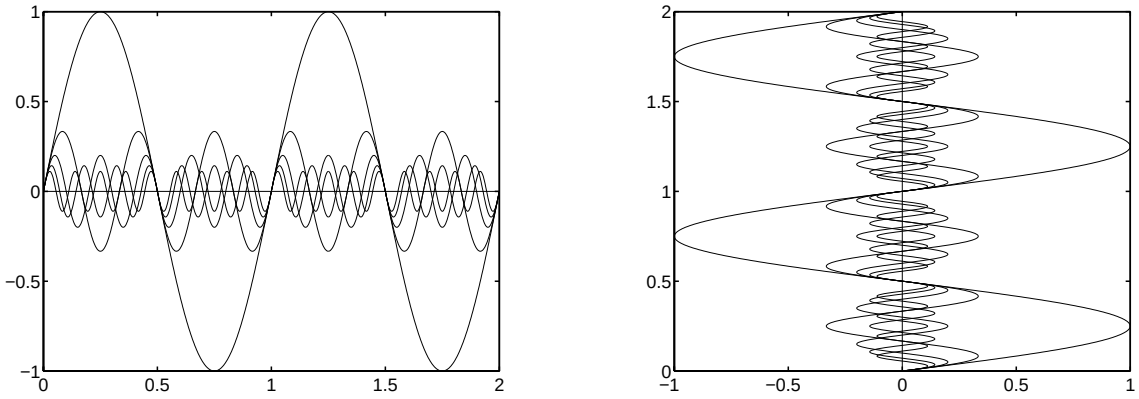


Abbildung 47: Die ersten 5 Fourierkomponenten zur Synthese einer Rechteckfunktion, links Zeitachse horizontal, rechts Zeitachse vertikal.

Dass mit den so definierten Fourierkomponenten tatsächlich ein Rechtecksignal synthetisiert werden kann, lässt sich durch Summation nachweisen. Das linke und rechte Teilbild von Abbildung 48 zeigen die Summation über 5 und über 50 Komponenten, realisiert wurde das mit

```
plot(time,real(sum(z(:,1:5),2)))   bzw.   plot(time,real(sum(z,2))) .
```

Es fällt auf, dass schon wenige Fourierkomponenten ausreichen, um einen richtigen Gesamteindruck zu erreichen, dass jedoch selbst mit einer größeren Anzahl das ‘Dach’ nicht völlig glatt wird. Insbesondere werden die Singularitäten (Überschwinger) an den Kanten

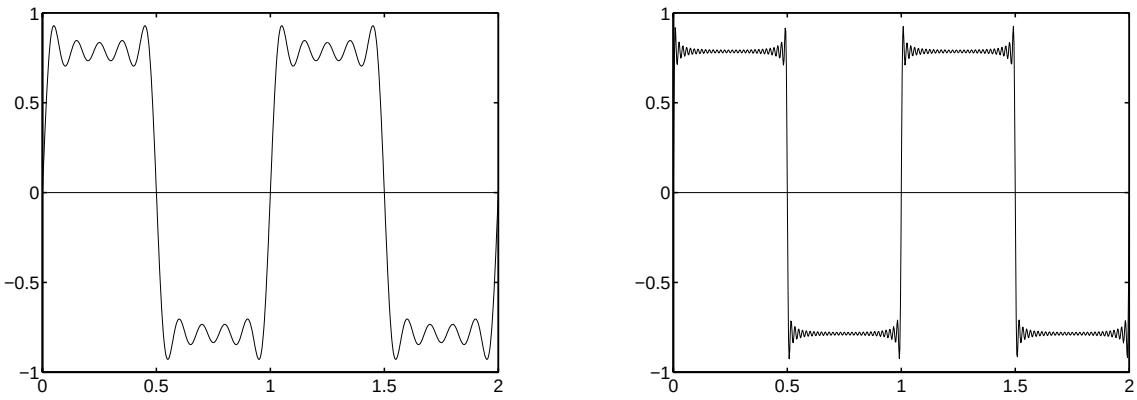


Abbildung 48: Fouriersynthese einer Rechteckfunktion aus Sinusfunktionen. Links: 5 Fourierkomponenten, rechts: 50 Komponenten.

der Rechteckfunktion mit steigender Komponentenzahl nicht geringer, nur nadelförmiger[40]. Weiterhin erkennt man, dass die Amplitude der ersten Fourierkomponente (1.0) merklich größer ist als die Amplitude des Rechtecksignals ($\pi/4$).

3.3.2 Bar, Stem, Pie

Balken- und Tortendiagramme sind zwar eine Domäne von Tabellenkalkulationsprogrammen, die auf diese Art Graphiken spezialisiert sind, können aber auch von MATLAB erstellt werden. Dies kann dann sinnvoll sein, wenn die Daten dafür innerhalb von MATLAB generiert oder verarbeitet werden, wenn spezielle Möglichkeiten der Graphik-Programmierung genutzt werden sollen oder wenn solche Diagramme in andere Graphiken integriert werden. Einige Beispiele sollen die Vorgehensweise illustrieren.

In Abbildung 49 sind die Amplituden der Fourierkomponenten der schon bekannten Rechteckfunktion dargestellt, links als Balkendiagramm, rechts als 'Stangen'-Diagramm (*stem*). Deutlich der $1/x$-Verlauf, im rechten Diagramm sind zudem die verschwindenden geradzahligen Komponenten gut erkennbar. Realisierung mit:

```
n=1:59;   coeff=n.^(-1).*mod(n,2);   bar(coeff);   bzw.
n=1:59;   coeff=n.^(-1).*mod(n,2);   stem(coeff); .
```

Die Kombination unterschiedlicher Diagrammtypen zeigt Abbildung 50. Im linken Teilbild wurden ein Balkendiagramm und ein Linienplot übereinandergezeichnet, rechts Balkendiagramm und *Stem*-Plot.

Die verantwortlichen MATLAB-Skripte demonstrieren das Lesen der Daten, die Verwendung des `hold`-Kommandos zur Überlagerung unterschiedlicher Diagramme und die Fest-

[40] Bei der technischen Anwendung eines so synthetisierten Rechtecksignals machen diese Spitzen in der Regel keine Probleme, da die Frequenzbandbreite realer technischer Geräte immer begrenzt ist.

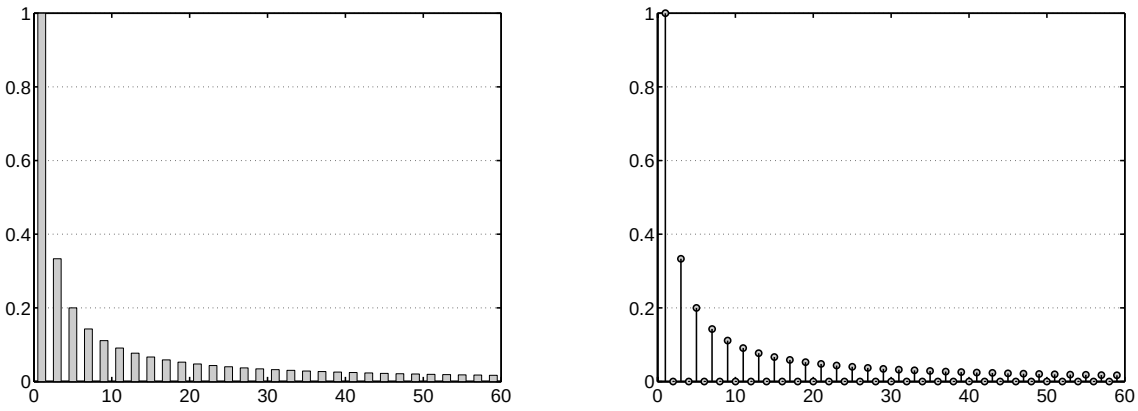


Abbildung 49: Amplituden der für die Fouriersynthese einer Rechteckfunktion benötigten Sinusfunktionen.

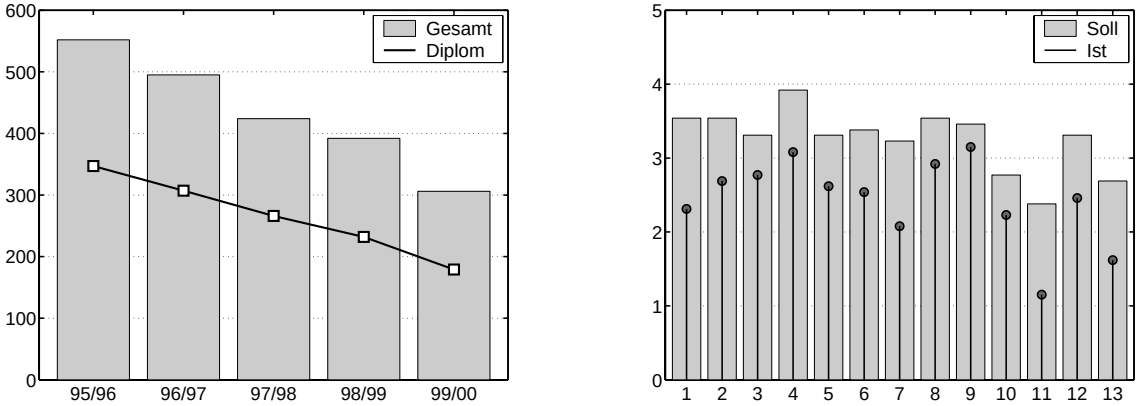


Abbildung 50: Beispiele für die Überlagerung verschiedener Graphiktypen in MATLAB: Links Bar und Plot, rechts Bar und Stem.

legung der Objekteigenschaften mit `set`-Anweisungen.

Linkes Teilbild:

```
[year total dip]=textread('studis.txt','%s%f%*f%f%*f',...
    'delimiter','\t','headerlines',1);
h=bar(total);
hold on;
set(h,'FaceColor',0.8*[1 1 1],'EdgeColor',[0 0 0]);
hl=plot(dip);
hm=plot(dip,'sk');        % 'sk' = Squares in blacK
set([hl,hm],'Linewidth',2,'Color',[0 0 0]);
set(hm,'MarkerSize',12,'MarkerEdgeColor',[0 0 0]);
set(hm,'MarkerFaceColor',[1 1 1]);
```

```
set(gca,'FontUnits','normalized','FontSize',0.05);
set(gca,'XTickLabel',year);
set(gca,'Linewidth',1.5,'YGrid','on');
hleg=legend([h(1),hl],'Gesamt','Diplom');
set(hleg,'FontSize',18);
hold off; .
```

Rechtes Teilbild:

```
[soll ist]=textread('sollist.txt','%*s%f%f','delimiter','\t');
h=bar(soll);
hold on;
set(h,'FaceColor',0.8*[1 1 1],'EdgeColor',[0 0 0]);
hs=stem(ist);
set(hs,'Linewidth',1.5,'Color',[0 0 0],'Markersize',8);
set(hs,'MarkerFaceColor',0.4*[1 1 1],'MarkerEdgeColor',[0 0 0]);
set(gca,'FontUnits','normalized','FontSize',0.05);
set(gca,'Linewidth',1.5,'YGrid','on');
hleg=legend([h(1),hs(2)],'Soll','Ist');
hold off; .
```

Bei der Erstellung der Legenden-Beschriftung ist darauf zu achten, dass die richtigen Handles verwendet werden, `h` beispielsweise ist ein Handle-Feld für alle Balken, `h(1)` verweist auf den ersten.

Neben den in den Beispielen benutzten Diagrammtypen kennt MATLAB Diagramme mit horizontalen und dreidimensionalen Balken (`barh`, `bar3`, `bar3h`), dreidimensionalen Stangen (`stem3`), Histogramme (`hist`), Stufenplots (`stairs`).

Für Tortengraphiken sind die beiden Funktionen `pie` und `pie3` zuständig. Die Rückgabewerte der beiden Funktionen sind Felder mit Handles für die Flächen und die Beschriftungen. Dadurch lassen sich die Diagramme mit den Mitteln der Handle-Graphik in allen Einzelheiten konfektionieren. Einfache Beispiele sind in Abbildung 51 zusammengestellt.

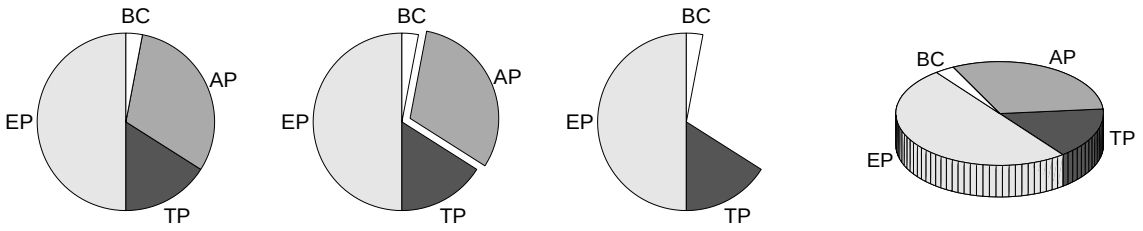


Abbildung 51: Tortengraphiken in MATLAB, dreimal Pie, einmal Pie3.

In den zugehörigen MATLAB-Skripten werden zunächst die Daten bereitgestellt:

```
x = [50 16 31 3];            % data
ex = [0 0 1 0];              % which one to explode
t = {'EP' 'TP' 'AP' 'BC'};   % text .
```

Das erste Diagramm wird dann mit

```
h=pie(x,t);
colormap gray;
k=reshape(h,2,size(x,2));
set(h(1),'facecolor',[1 1 1]*0.9);
set(k(2,:),'fontunits','normalized','fontsize',0.1);
```

erstellt, das zweite mit

```
h=pie(x,ex,t);
...,
```

das dritte mit

```
h=pie(x,t);
...
set(k(:,3),'visible','off');
```

das vierte schließlich mit

```
h=pie3(x,t);
colormap gray;
k=reshape(h,4,size(x,2));
set(k(1:3,1),'facecolor',[1 1 1]*0.9);
set(k(4,:),'fontunits','normalized','fontsize',0.1); .
```

Mit dem `reshape` des Handle-Feldes in eine zweidimensionale Matrix wird jeweils dafür gesorgt, dass zusammengehörige Handles einfacher zu benutzen sind.

3.3.3 Logarithmische Skalen

Logarithmische Skalierungen werden immer dann verwendet, wenn Koordinaten – Variablen oder physikalische Eigenschaften – um mehrere Größenordnungen variieren, in allen Bereichen aber eine Darstellung mit guter relativer Auflösung erwünscht ist. Aber auch dann, wenn theoretische Beschreibungen das nahe legen, sind logarithmische Skalen nützlich zur vereinfachten qualitativen visuellen Auswertung. Einfach logarithmische bei

exponentiellem oder logarithmischem Verlauf, doppelt logarithmische bei potenzartigem Zusammenhang.

Typische Bereiche für logarithmische Skalen sind die Darstellungen frequenzabhängiger Eigenschaften in der Mechanik, Akustik oder Elektronik. Ein Beispiel ist die frequenzabhängige Charakteristik des aus den Abbildungen 41 und 42 bekannten Tastkopfs für ein Oszilloskop.

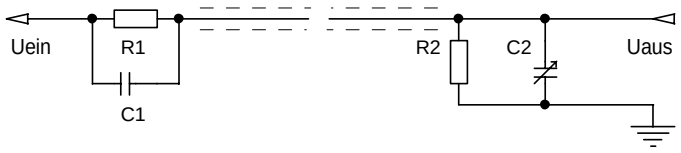


Abbildung 52: Ersatzschaltbild eines Oszilloskop-Tastkopfs.

Die komplexe Durchlasscharakteristik der verallgemeinerten Schaltung (Abbildung 52) ist gegeben durch

$$T = \frac{U_{\text{aus}}}{U_{\text{ein}}} = \frac{R_2 \parallel C_2}{R_1 \parallel C_1 + R_2 \parallel C_2}\,. \tag{3.2}$$

Darin sind R_1 und C_1 Widerstand und Kapazität in der Tastkopfspitze, R_2 und C_2 die der Restschaltung. Die weitere Berechnung kann man MATLAB überlassen:

```
RC=inline('(1/R+2*pi*j*F*C).^(-1)','R','C','F');
f=logspace(1,5,100);
rc1=RC(9e6,[1 1 1]*15e-12,f');
rc2=RC(1e6,[100 135 200]*1e-12,f');
t=rc2./(rc1+rc2); .
```

Die Funktion `RC` berechnet den komplexen frequenzabhängigen Widerstand einer Parallelschaltung aus `R` und `C`. Bei der Berechnung von `rc1` und `rc2` im Frequenzbereich $10 \ldots 10^5$ Hz werden dann die konkreten Werte aus Abbildung 41 eingesetzt, der Einfachheit halber gleich 3 verschiedene Werte für die Kompensationskapazität (zu klein, richtig, zu groß). Die Ergebnisse – die drei Durchlassfunktionen – sind in Abbildung 53 dargestellt, links durch

```
plot(f,abs(t),'Color','k','Linew',1.5);
```

in linearer Skalierung, rechts durch

```
semilogx(f,abs(t),'Color','k','Linew',1.5);
```

mit logarithmischer Frequenzskala. Die Darstellungen sprechen für sich. Aus dem Verlauf wird insbesondere auch klar, warum die Kompensationseinstellung der Tastköpfe bei allen Oszilloskopen mit Rechtecksignalen von ungefähr 1 kHz vorgenommen wird.

Der Funktion `semilogx` entsprechend gibt es `semilogy` und `loglog` für die anderen Möglichkeiten, logarithmische Skalen in zwei Dimensionen zu verwenden. Darüber hinaus

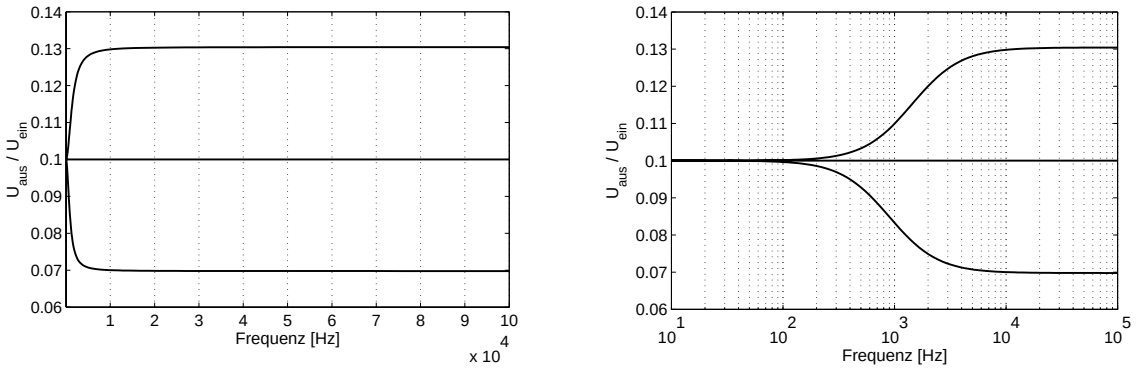


Abbildung 53: Durchlasscharakteristiken eines Oszilloskop-Tastkopfs (Teilerverhältnis 10:1), links lineare, rechts logarithmische Frequenzskala. Obere Kurve: zu kleine Kompensationskapazität, mittlere Kurve: richtige Einstellung, untere Kurve: zu große Kapazität.

bietet MATLAB auch die Möglichkeit, die Skaleneigenschaften von Achsen direkt einzustellen, mit

```
set(gca,'ZScale','log');
```

lässt sich gegebenenfalls auch die dritte Dimension logarithmisch skalieren.

3.3.4 Unterschiedliche Y-Skalierung

Sollen unterschiedliche physikalische Größen im gleichen Diagramm dargestellt werden, ist es meist hilfreich, mit mehreren y-Achsen zu arbeiten. Der einfachste und übersichtlichste Fall sind unterschiedliche Achsen an der linken und rechten Seite des Diagramms. Dafür stellt MATLAB eine eigene Funktion bereit, `plotyy`, der als Parameter neben den Daten auch die Art der Plots übergeben werden kann.

```
plotyy(x1,y1,x2,y2,'loglog','semilogx');
```

würde in einem Diagramm `x1,y1` doppelt logarithmisch und `x2,y2` mit einer (gleichen) logarithmischen x-Achse und einer linearen y-Achse plotten.

Diagramme dieser Art sind in der Elektronik gebräuchlich, der Frequenzgang (Betrag und Phase in Abhängigkeit von der Frequenz) charakteristischer Größen von Bauelementen (Impedanz) und von elektronischen Schaltungen (Verstärkung) wird üblicherweise so dargestellt (*Bode*-Diagramm).

Als Beispiel berechnen wir den Frequenzgang eines einfachen Transistorverstärkers mit einem bipolaren Transistor in Emitterschaltung. Eine Schaltungsmöglichkeit dafür ist im linken Teilbild von Abbildung 54 angegeben. Für die Berechnung sind einige Formalisierungen sinnvoll:

- Es interessiert nur das Wechselspannungsverhalten, alle Gleichspannungen werden

weggelassen.

- Der Transistor wird als Stromverstärker mit konstanter Verstärkung des Basisstroms und konstantem Eingangswiderstand idealisiert.
- Der Frequenzgang bei hohen Frequenzen wird durch die Kollektor-Basis-Kapazität des Transistors bestimmt (Miller-Effekt).

Die so formalisierte Schaltung zeigt das rechte Teilbild von Abbildung 54, Punkt `I` entspricht der Basis des Transistors, Punkt `II` dem Kollektor. G_i sind die den Widerständen und Kapazitäten entsprechenden Leitwerte, G_1 besteht aus der Serienschaltung von Koppelkondensator C_b und Innenwiderstand der Quelle R_b, G_2 ist der reziproke Eingangswiderstand des Transistors R_be, G_3 die Kollektor-Basis-Kapazität C_cb, G_4 der Kollektorwiderstand R_c. Alle anderen Widerstände und Kapazitäten werden bei der Rechnung vernachlässigt.

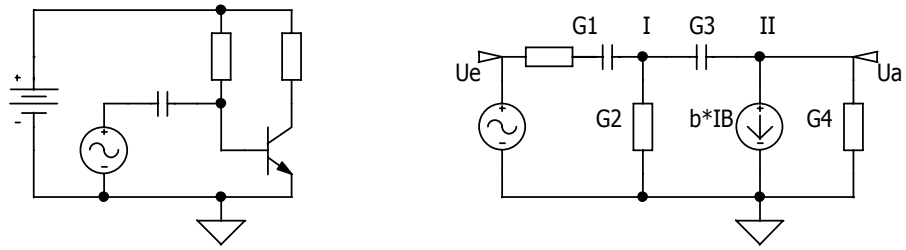


Abbildung 54: Transistorverstärker in Emitterschaltung, links real, rechts formalisiert.

Zur Berechnung werden die Stromsummen an den Punkten `I` und `II` gebildet:

$$\text{(I):}\quad 0 = (U_\mathrm{e} - U_\mathrm{I})G_1 - U_\mathrm{I}G_2 + (U_\mathrm{a} - U_\mathrm{I})G_3 \tag{3.3}$$

$$\text{(II):}\quad 0 = (U_1 - U_\mathrm{a})G_3 - bU_1G_2 - U_\mathrm{a}G_4\,. \tag{3.4}$$

Daraus ergibt sich die komplexe Verstärkung v:

$$v = \frac{U_\mathrm{a}}{U_\mathrm{e}} = \frac{-G_1(G_2b - G_3)}{G_1G_3 + G_1G_4 + G_2G_3(1+b) + G_2G_4 + G_3G_4}\,. \tag{3.5}$$

In der MATLAB-Implementierung werden zunächst alle physikalischen Größen, d. h. alle Widerstände und Kapazitäten der Schaltung festgelegt und daraus die Leitwerte berechnet. Der Koppelkondensator zwischen Wechselspannungsquelle und Basis wird relativ klein gewählt (1 μF), um seinen Einfluss auf den Frequenzgang zu demonstrieren. Für die Kollektor-Basis-Situation werden gleichzeitig zwei unterschiedliche Fälle definiert – reine Miller-Kapazität `Ccb` und zusätzliche Spannungsgegenkopplung mit einem Widerstand `Rg` zwischen Kollektor und Basis (parallel zu `Ccb`):

```
Ccb = 1e-11;                % 10 pF
```

```
Cb = 1e-6;                  % 1 uF
Rb = 1000;                  % 1 kOhm
Rbe = 1000;                 % 1 kOhm
Rc = 1000;                  % 1 kOhm
Rg = 10000;                 % 10 kOhm
b = 300;
f = logspace(1,7.48,100)';  % 10 Hz ... 30 MHz
G1 = i*2*pi*f*Cb./(1+Rb*i*2*pi*f*Cb)*[1  1];
G2 = 1/Rbe;
G3 = [i*2*pi*f*Ccb  i*2*pi*f*Ccb+1/Rg];
G4 = 1/Rc; .
```

In der Formel für `G1` wird die Größe des Datenfeldes durch Multiplikation mit `[1 1]` an die Größe von `G3` angepasst. `G2` und `G4` sind Skalare, daher unproblematisch.

Die Verstärkung wird nach Gleichung 3.5 aufgeschrieben, zu achten ist auf elementweise Multiplikation:

```
v = -G1.*(G2*b-G3)./(G1.*G3+G1.*G4+G2.*G3*(1+b)+G2.*G4+G3.*G4); .
```

Die Funktion `unwrap` sorgt dafür, dass keine Sprünge in der Darstellung der Phase auftreten, dann die Plotanweisung:

```
ph = 180/pi*unwrap(angle(v));
[ax,ht,hp]=plotyy(f,abs(v),f,ph,'loglog','semilogx'); .
```

Das Ergebnis zeigt Abbildung 55. Wie erwartet ist die Verstärkung bei mittleren Frequenzen konstant und nimmt bei hohen Frequenzen proportional zu $1/f$ ab, in der doppelt logarithmischen Auftragung[41] linear mit der Steigung -1. Die Abnahme zu niedrigen Frequenzen ist durch den für übliche Anwendungen (Audiobereich) zu klein gewählten Koppelkondensator bedingt. Eine Gegenkopplung durch einen zusätzlichen Widerstand zwischen Kollektor und Basis verringert die Verstärkung, erhöht aber die Bandbreite zu hohen Frequenzen beträchtlich.

Die Konfektionierung des Plots wurde erledigt durch

```
set(ax,'xlim',[10 3e7]);
set(ax(1),'ylim',[0.5 3e2]);
set(ax(2),'ylim',[-270 -90]);
set(ax(2),'ytick',[-270 -225 -180 -135 -90]);
set(ax(2),'xaxislocation','top','xticklabel',[]);
set(ht,'linestyle','-','linewidth',1);
set(ht(2),'linestyle','-.');
set(hp,'linestyle','--','linewidth',1);
set(hp(2),'linestyle',':');
```

[41] In der Technik ist es oft üblich, statt der Verstärkung deren Logarithmus an die Achse zu schreiben, die Maßgröße ist *deziBel* (dB), das 20fache des Zehnerlogarithmus des Spannungsverhältnisses.

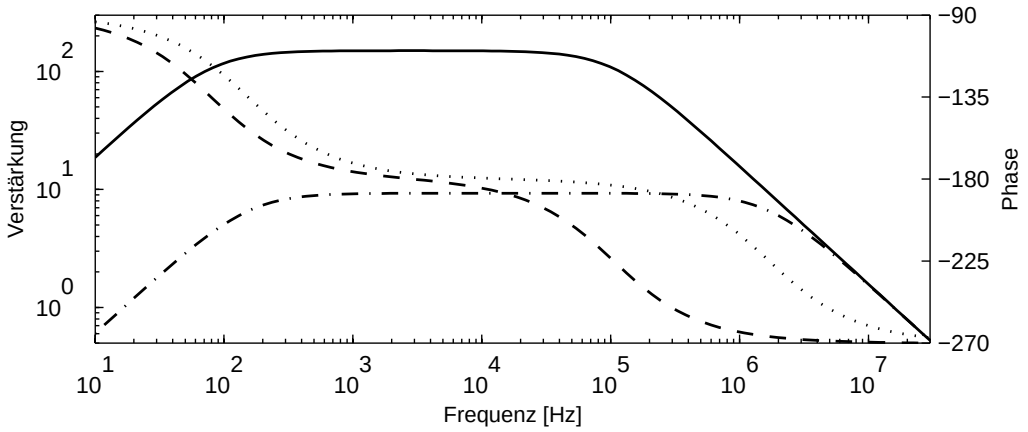


*Abbildung 55: Frequenzgang eines Transistorverstärkers in Emitterschaltung. Durchgezogene Linie: Betrag der Verstärkung ohne Gegenkopplung, strichpunktiert: mit Spannungsgegenkopplung (*10 kΩ *zwischen Kollektor und Basis), gestrichelt und gepunktet sind die zugehörigen Phasengänge eingezeichnet.*

```
set([ht hp],'color','k');
tlabel=get(ax(1),'ylabel');
plabel=get(ax(2),'ylabel');
set(tlabel,'string','Verstärkung');
set(plabel,'string','Phase');
hxl=xlabel('Frequenz [Hz]');
set([ax hxl tlabel plabel],'fontunits','normalized','fontsize',0.07);
set(ax,'position',[0.15 0.18 0.7 0.55]);
set(gcf,'paperposition',[1 3 6 3]); .
```

3.3.5 Fehlerbalken

Bei der Bewertung von Messdaten spielt die kritische Fehlerdiskussion eine wichtige Rolle. Messwerte sind grundsätzlich mit Fehlern behaftet, eine sinnvolle Angabe der Genauigkeit hilft dem Leser, die Messwerte (und Folgerungen daraus) richtig einzustufen.

Ist die Anzahl der Messdaten nicht zu groß, macht es Sinn, den abgeschätzten Fehler zu jedem einzelnen Messpunkt als *Fehlerbalken* einzuzeichnen. In MATLAB ist dafür die Funktion `errorbar` zuständig.

Wir sehen uns das an einem einfachen Beispiel an, der Abschwächung von Gamma-Strahlung durch eine Bleiabschirmung. Gamma-Strahlung wechselwirkt mit Materie über drei Prozesse – Photoeffekt, Compton-Effekt und Paarbildung. In allen wird die ganze oder ein wesentlicher Teil der Energie eines Quants in einem einzigen Wechselwirkungsprozess an Elektronen übertragen. Man erwartet mithin eine Abschwächung durch die

Bleiabschirmung, die exponentiell von der Dicke abhängt[42].

Die Messdaten sind als Zählraten in Abhängigkeit von der Bleidicke (in mm) angegeben, zusätzlich ist zu den einzelnen Messwerten eine Fehlerangabe gemacht (offensichtlich die Wurzel aus der Zählrate als Maß für den statistischen Fehler). Nebenstehend der Anfang der Datentabelle.

Dicke	Rate	Fehler
0	96	9.8
2	64	8.0
4	63	7.9
6	51	7.1
⋮	⋮	⋮

Die Daten seien in den drei Variablenvektoren `x`, `y` und `e` abgelegt, die Anweisung

```
he = errorbar(x, y, e);
```

liefert dann die Graphik in Abbildung 56 links. `he` ist ein Feld mit zwei Handles, die erste für die Fehlerbalken, die zweite für den Linienplot. Linienstärke und Farbe wurden mit

```
set(he,'Color','k','Linewidth',2);
```

eingestellt. Eine Variante ist im rechten Teilbild dargestellt, logarithmische y-Skala und Marker an den Messwerten. Man erreicht das durch

```
set(gca,'YScale','log','YLim',[5,130],'XLim',[-0.5,20.5]);
set(he(2),'Marker','o','Markerfacecolor','w');
```

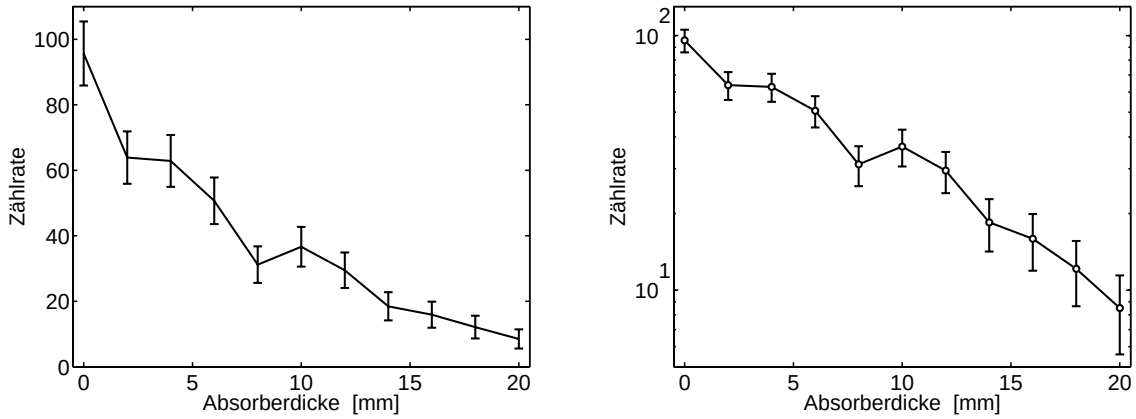


Abbildung 56: Abschwächung von Gamma-Strahlung durch eine Bleiabschirmung, Linienplot mit Fehlerbalken an den Messpunkten. Links lineare, rechts logarithmische y-Skala.

Nicht besonders aussagekräftig ist der von `errorbar` mitgelieferte Linienplot. Man kann die Linien entfernen mit

```
set(he(2),'Linestyle','none');
```

Stattdessen sollte man – wenn physikalisch begründbar – eine geeignete Funktion an die

[42] Bei schnellen geladenen Teilchen (Elektronen, Ionen, Alpha-Teilchen), die über Vielfachstreuprozesse abgebremst werden, würde man statt des exponentiellen Verlaufs relativ feste Reichweiten in Materie erwarten.

Messwerte anfitten und mit einzeichnen. Das ist in Abbildung 57 – links in linearer, rechts in logarithmischer Auftragung – gemacht. Damit die Messwerte nirgendwo von der Fitfunktion überdeckt werden, werden zuerst die Fitfunktion, dann die Messwerte mit Fehlerbalken gezeichnet.

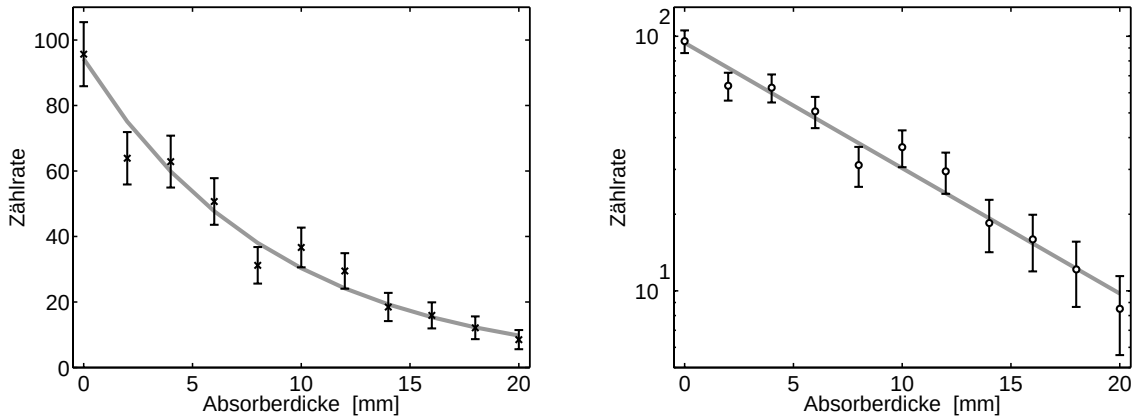


Abbildung 57: Abschwächung von Gamma-Strahlung durch eine Bleiabschirmung, Messpunkte mit Fehlerbalken und Fitfunktion. Links lineare, rechts logarithmische Auftragung.

Für den (eher seltenen) Fall, dass die Fehlerbalken für positive und negative Abweichungen unterschiedlich lang sein sollen, arbeitet `errorbar` auch mit 4 Parametern, der dritte gibt den positiven, der vierte den negativen Fehler an.

3.3.6 Beschriftung

Graphik-Titel und Achsenbeschriftungen werden in MATLAB mit den Funktionen `title`, `xlabel`, `ylabel`, `zlabel` angeordnet, die Tickbeschriftungen können über die Eigenschaften `XTickLabel`, `YTickLabel`, `ZTickLabel` des jeweiligen *Axis*-Objekts verändert werden. Daneben gibt es die allgemeine Funktion `text`, mit der Text an eine beliebige Stelle der Graphik gesetzt werden kann. Beschreibungen (Legenden) schließlich können mit `legend` an der Graphik platziert werden.

Alle Texte (außer den Tickbeschriftungen) sind mit den Möglichkeiten der Handle-Graphik in ihren Objekteigenschaften sehr variabel konfigurierbar. So sind Position, Richtung, Schriftart, -größe, -farbe, Text, Ausrichtung usw. beliebig einzustellen. Darüber hinaus können Zeichen – wie von TEX gewohnt – mit '^' hoch- und mit '_' tiefgestellt werden, und es kann eine große Zahl von TEX-Makros für Sonderzeichen und Symbole verwendet werden. Einen Überblick gibt Tabelle 2.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	ϕ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	←
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	→
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	μ	\Upsilon	Υ	\circ	°
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∋	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	≈	\aleph	ℵ
\Im	ℑ	\Re	ℜ	\wp	℘
\otimes	⊗	\oplus	⊕	\oslash	∅
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in	∈	\o	o
\rfloor	⌋	\lceil	⌈	\nabla	∇
\lfloor	⌊	\cdot	·	\ldots	…
\perp	⊥	\neg	¬	\prime	′
\wedge	∧	\times	×	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	⟨	\rangle	⟩		

Tabelle 2: TEX-Symbole, die im MATLAB-Graphiksystem verwendet werden können.

3.4 Simulations- und Anpassungsrechnungen

Ein wenig von der reinen Graphik abschweifend wollen wir uns in diesem Kapitel mit der Simulation und Anpassung von Daten in MATLAB beschäftigen. Dazu je ein Beispiel an den uns schon bekannten physikalischen Objekten, Transistorverstärker und Tastkopf.

3.4.1 Simulation: Transistorverstärker

MATLAB ist von seinem Konzept her besonders für solche Simulationsprobleme geeignet, die sich auf Operationen mit reellen oder komplexen Matrizen abbilden lassen. Für diesen Bereich gibt es mit SIMULINK[43] einen umfangreichen Zusatz zu MATLAB, mit dem Modelle von dynamischen Systemen graphisch erstellt werden können und mit dem das Verhalten der so modellierten Systeme berechnet werden kann. Es ist mit SIMULINK möglich, auch sehr komplexe Systeme aus der Mechanik, Elektronik o. ä. zügig zu formulieren und zu berechnen.

Ein einfaches lineares Simulationsbeispiel, das direkt in MATLAB realisiert wird, kann die Möglichkeiten allenfalls andeuten. Wir berechnen, wie Rechtecksignale, hochfrequent und niederfrequent, mit dem Transistorverstärker aus Abschnitt 3.3.4 verstärkt und verzerrt werden. Die Eingangsvariable für die Simulation ist die Fourierdarstellung des Rechtecksignals, die Simulationsfunktion der Frequenzgang des Verstärkers. Durch Multiplikation erhält man die Fourierdarstellung des Ausgangssignals.

Zunächst die Formulierung des Rechtecksignals als Fourierreihe; um das Hochfrequenzverhalten zu zeigen, wird eine Frequenz von 50 kHz verwendet:

```
basefreq = 50000;
T = 5/basefreq;
oddnums = [1:2:99];
npoints = 1000;
time = linspace(0,T*(1-1/npoints),npoints);
[f,t] = meshgrid(oddnums*basefreq,time);
coeff = f.^(-1)*basefreq;
z = -j*coeff.*exp(2*j*pi*f.*t); .
```

Der Frequenzgang des Verstärkers, berechnet für das Frequenzfeld der Fourierreihe[44]:

```
v = '-G1.*(G2*b-G3)./(G1.*G3+G1.*G4+G2.*G3*(1+b)+G2.*G4+G3.*G4)';
G1 = i*2*pi*f*Cb./(1+Rb*i*2*pi*f*Cb);
G2 = 1/Rbe;
```

[43] SIMULINK ist ein eingetragenes Warenzeichen von *The MathWorks Inc.*

[44] Der formalen Einfachheit halber wird bei der Implementierung das von `meshgrid` produzierte 2D-Feld `f` benutzt, das aus 1000 gleichen Zeilen besteht. Eine Zeile des Feldes würde ausreichen. Bei umfangreicheren Simulationen sollte man geeignet 'verschlanken'.

```
G3 = i*2*pi*f*Ccb;
G4 = 1/Rc;
v1 = eval(v);
G3 = i*2*pi*f*Ccb+1/Rg;
v2 = eval(v); .
```

Die Verstärkung `v` wird als Textstring definiert, kann auf diese Weise mit `eval` mehrfach verwendet werden (ohne und mit Gegenkopplungswiderstand in `G3`), ohne dass der Text dupliziert werden muss.

Die Multiplikation des Eingangssignals `z` mit den beiden Verstärkungsfunktionen `v1` und `v2` und die Summation der Fourierreihen ergibt die Ausgangssignale `zo1` und `zo2`. Geplottet werden sie zusammen mit dem Eingangssignal:

```
zi = real(sum(z,2));
zo1 = real(sum(v1.*z,2));
zo2 = real(sum(v2.*z,2));
hz = plot(time,[100*zi+250 zo1 10*zo2-250]); .
```

Die Konfektionierung des Plots erfolgt mit:

```
set(hz,'color','k','linewidth',1.5);
set(gca,'xtick',linspace(0,T,10));
set(gca,'xticklabel',[],'yticklabel',[]);
set(gca,'xgrid','on','ygrid','on'); .
```

Die Simulationsergebnisse zeigt Abbildung 58, links für das 50-kHz-Signal ohne und mit Gegenkopplung, rechts für ein 20-Hz-Signal bei unterschiedlichen Koppelkondensatoren.

3.4.2 Fit: Tastkopfdaten

Oft ist es in physikalischen Experimenten nicht möglich, Parameter direkt zu messen, sondern nur ihre Auswirkungen auf andere, direkt messbare Größen. Anpassungsrechnungen, mit denen man durch Variation der Parameter den theoretisch erwarteten Verlauf an die Messergebnisse anpasst, liefern dann als Ergebnis die gewünschten Parameter. Für solche Anpassungsrechnungen gibt es in MATLAB die Funktion `fminsearch` zur Minimalwertsuche. Ausgehend von Anfangswerten wird ein Parametersatz so lange variiert, bis das Minimum einer vom Anwender definierten Funktion gefunden ist, als Strategie wird das Simplex-Verfahren verwendet.

Am Beispiel der Messdaten von einem Oszilloskop-Tastkopf (Abschnitt 3.1.8) soll die Vorgehensweise erläutert werden. Der Frequenzgang eines Tastkopfs wurde in Abschnitt 3.3.3 berechnet. Unterwerfen wir die Sinusreihe für ein Rechtecksignal diesem Frequenzgang,

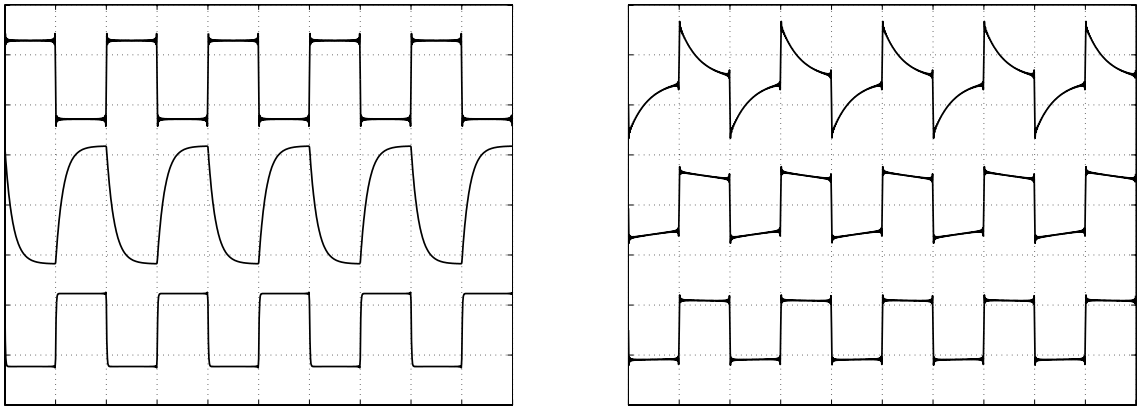


Abbildung 58: Verstärkung von Rechtecksignalen in dem Transistorverstärker aus Abbildung 54. Links das Hochfrequenzverhalten bei 50 kHz, obere Kurve: Eingangssignal, mittlere Kurve: ohne Gegenkopplung, Verstärkung etwa 100fach, untere Kurve: mit Gegenkopplung, Verstärkung 10fach. Rechts das Niederfrequenzverhalten bei 20 Hz mit unterschiedlichen Koppelkondensatoren, von oben nach unten: 5 µF, 50 µF, 500 µF.

können wir die gemessenen Signale simulieren. Um die realen Tastkopfdaten zu bestimmen, wird das simulierte Signal an das gemessene angepasst. Drei Parameter werden dazu variiert, die Kapazitätswerte der beiden Kondensatoren (Schaltung in Abbildung 52) und die Signalamplitude.

Zunächst werden die gemessenen Daten, die in einer Datei abgelegt sind, und die Rechteckfunktion vorbereitet:

```
basefreq = 1000;
period = 1/basefreq;
T = 2*period;
ndata = 1000;
startdata = 1250;
npoints = 400;
all = load('tastkopf.dat');
y = all(startdata:startdata+ndata-1,3);
exptime = linspace(0,T*(1-1/ndata),ndata);
time = linspace(0,T*(1-1/npoints),npoints);
y = interp1(exptime,y,time)';
y = y-mean(y);
oddnums = [1:2:99];
[f,t] = meshgrid(oddnums*basefreq,time);
coeff = f.^(-1)*basefreq;
z = -j*coeff.*exp(2*j*pi*f.*t); .
```

Für die Anpassung werden 2 Perioden des Rechtecksignals von 1 kHz verwendet, das ent-

spricht 1000 Punkten aus dem gemessenen Datensatz. Der Vektor `y` enthält diese Messdaten, `exptime` die zugehörigen Zeiten. Gerechnet werden soll dann in einem Feld mit nur noch 400 Punkten, um die Rechenzeit zu verkürzen. Auf diese Feldgröße wird `y` durch Interpolation reduziert, schließlich noch symmetrisiert (`y-mean(y)`). Die restlichen 4 Zeilen definieren die Sinusreihe für das synthetische Rechtecksignal.

Für die Anpassungsrechnung wird ein einigermaßen günstiger Satz von Anfangswerten `c0` gewählt, dann die Berechnung durchgeführt und das Ergebnis zusammen mit den Messdaten geplottet:

```
global c f y z Z
c0 = [10e-12,150e-12,150];
c = fminsearch(@devi,c0);
plot(time,y,time,real(sum(Z,2)));
```

Durch `global` werden Variable global sichtbar gemacht. An der Stelle, wo sie verwendet werden sollen, muss eine dazu korrespondierende `global`-Anweisung stehen[45].

Die zu minimierende Funktion `devi` ist noch zu definieren:

```
function d = devi(c)
global y z Z
Z = z.*trfun(c);
d = sum(abs(y-real(sum(Z,2))));
```

Schließlich noch die ausgelagerte Funktion für den Frequenzgang:

```
function tr = trfun(c)
global f
RC = inline('(1/R+2*pi*j*F*C).^(-1)','R','C','F');
r1 = RC(9e6,c(1),f);
r2 = RC(1e6,c(2),f);
tr = c(3)*r2./(r1+r2);
```

Das Ergebnis der Anpassungsrechnung ist 14.9 pF und 104 pF für die beiden Kapazitäten, es entspricht ziemlich genau unseren Erwartungen[46]. Die graphische Darstellung in Abbildung 59 zeigt die gute Übereinstimmung zwischen Messkurve und Fit.

[45] Die Anweisung `global` sollte verwendet werden, wenn Variable an unterschiedlichen Stellen (Funktionen) zugänglich sein sollen. Um Variable, d. h. ihren Wert, zwischen verschiedenen Funktionsaufrufen im Speicher zu behalten, benutzt man die Anweisung `persistent`.

[46] Überlegen Sie, was sich ergäbe, wenn man versuchte, die Daten für den richtig eingestellten Tastkopf anzupassen?

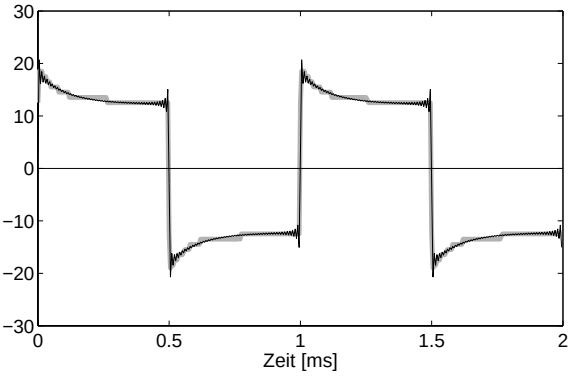


Abbildung 59: Rechtecksignal an einem ungünstig eingestellten Tastkopf (zu kleine Kompensationskapazität). Graue Kurve: Messdaten, schwarz: Ergebnis der Anpassungsrechnung.

3.5 3D-Darstellungen

Der Begriff '3D-Darstellung' ist im Umfeld der physikalischen Graphikanwendungen nicht ganz scharf definiert. Die Dreidimensionalität kann rein räumlich, konstruktiv gemeint sein, man kann darunter aber auch verstehen, dass Funktionen in Abhängigkeit von mehreren (meist zwei) Variablen dargestellt werden. In beiden Bereichen hat MATLAB einiges zu bieten, wir werden uns zunächst dem zweiten Bereich zuwenden und uns mit der Volumendarstellung dann im nächsten Kapitel beschäftigen.

3.5.1 Linienplots

Das dreidimensionale Analogon zu `plot` ist `plot3`. Damit werden Linien und Punkte im Dreidimensionalen gezeichnet. Die Erweiterung ist sinngemäß, d. h. als Parameter werden jeweils 3 ein- oder zweidimensionale Felder einheitlicher Größe erwartet, gegebenenfalls gefolgt von näheren Spezifikationen der Linien- oder Symboleigenschaften.

Mit `view` kann der Blickwinkel auf den Plot verändert werden, als Parameter sind alternativ 2 Winkel (Drehung um z, Kippung gegen xy), 3 Koordinaten (Betrachtungspunkt) oder eine Transformationsmatrix (4×4) anzugeben[47].

Die schon hinlänglich bekannten Fourierkomponenten einer Rechteckfunktion

```
[x,y] = meshgrid(1:2:25,linspace(0,1,200));
z = 1./x.*sin(2*pi*y.*x);
```

lassen sich in einem 3D-Plot etwas übersichtlicher darstellen als zweidimensional. Der Plot in Abbildung 60 links wurde erstellt mit:

[47] Bei 2D-Plots kann der Blickwinkel selbstverständlich auch mit `view` eingestellt werden, da MATLAB bei Graphiken immer von 3 Dimensionen ausgeht. Man kann dies nutzen, um 2D-Plots in der Ebene zu drehen – mit `view(270,90)` beispielsweise um 270 Grad – oder auf den Kopf zu stellen – mit `view(0,-90)`. Voreingestellt ist `view(0,90)` bei 2D- und `view(-37.5,30)` bei 3D-Plots, diese Voreinstellungen können verkürzt durch `view(2)` oder `view(3)` angeordnet werden.

```
xf = [25 1 1 25 25];
yf = [0 0 1 1 0];
zf = zeros(size(yf));
plot3(x,y,z,xf,yf,zf,'Color',[0 0 0],'LineW',1);
set(gca,'XTick',[],'YTick',[],'ZTick',[]);
axis tight;  box on;  view(70,40);  .
```

Die 2D-Felder `x`, `y` und `z` beschreiben die Sinusfunktionen, `xf`, `yf` und `zf` zeichnet einen zusätzlichen rechteckigen Rahmen bei `z=0`.

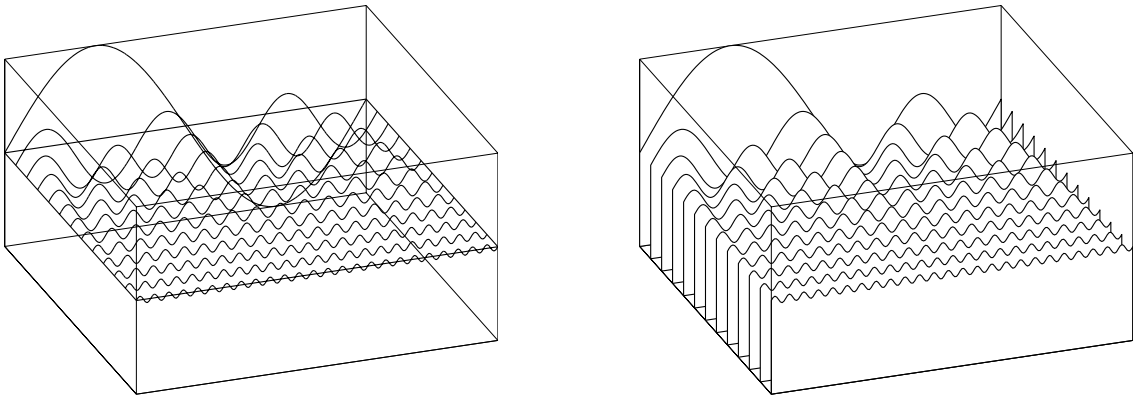


Abbildung 60: Sinuskomponenten einer Rechteckfunktion, links mit `plot3`, rechts mit `waterfall`.

Mit `plot3` hat man keine Möglichkeit, hinter anderen liegende Linien zu verdecken, wird das gewünscht, kann man bei geordneten, gitterartigen Datenfeldern stattdessen `waterfall` verwenden:

```
h = waterfall(x',y',z');
set(h,'EdgeColor','k','LineW',1); .
```

Die Wirkung verdeutlicht das rechte Teilbild von Abbildung 60.

Als weiteres mathematisch einfach zu beschreibendes geometrisches Objekt verwenden wir eine Spiralfeder, die parametrisiert z. B. durch `[sin(t),cos(t),t]` definiert werden kann. Ein wenig komplizierter wird es, wenn die Lage im Raum verändert werden soll. Eine solche Feder mit beliebigem Anfangs- und Endpunkt und Radius sowie kurzen Endstücken lässt sich dann als MATLAB-Funktion so formulieren:

```
function hc = spring(a, b, e, r0)
if (nargin<3) return; end;
if (nargin<4) r0=e/2; end;
PHI = 16*pi;        % 8 turns
```

```
d = b-a;
l = sqrt(sum(d.*d));
L = l-2*e;
r = sqrt(r0*r0-L*L/PHI/PHI);
angle = linspace(0,PHI,400);
x = [0 0 r*sin(angle) 0 0]+a(1);
y = [0 0 r*cos(angle) 0 0]+a(2);
z = [0 e linspace(e,l-e,400) l-e l]+a(3);
hc = plot3(x,y,z,'-k','Linewidth',1);
phirot = 180/pi*acos(d(3)/l);
ax = [-d(2) d(1) 0];
if (sum(ax.*ax)==0)  ax = [1 0 0];  end;
rotate(hc,ax,phirot,a);
```

Anfangs- und Endpunkt `a` und `b` sind Vektoren, Länge der Enden `e` und Radius der ungedehnten Feder `r0` skalar. Die Feder wird mit einem durch die Dehnung verringerten Radius `r` in der richtigen Länge zunächst in z-Richtung gezeichnet, dann um ihren Anfangspunkt `a` in die richtige Lage gedreht (`rotate`).

Eine Graphik, bei der 'Federkräfte' eine Rolle spielen, die schematische Darstellung typischer Schwingungsformen einer linearen Kette, wird erstellt durch:

```
function chains
L = 100,  N = 21,  odd = 1:2:N;
x = linspace(0,L,N);
y = x*0;
z = 4*sin(x*2*pi/L);
chain(x,y,z);
z(odd) = -0.5*z(odd)
chain(x,y,z+10);
axis equal,  axis off;
hold off;
view(-60,5); .
```

Die Auslenkung `z` wird einmal akustisch, einmal optisch formuliert, damit dann jeweils eine transversal ausgelenkte lineare Kette gezeichnet:

```
function chain(x,y,z)
N = max(size(x));
od = 1:2:N,  ev = 2:2:N;
r = [x;y;z];
for i = 1:N-1
  h=spring(r(:,i),r(:,i+1),1,0.3);
  set(h,'LineW',0.2);
```

```
  hold on;
end;
ho = plot3(x(od),y(od),z(od),'ko');
he = plot3(x(ev),y(ev),z(ev),'ko');
set(ho,'MarkerSize',8,'MarkerFaceColor','k');
set(he,'MarkerSize',7,'MarkerFaceColor','k'); .
```

Die graphische Darstellung in Abbildung 61 macht deutlich, dass optische Schwingungsmoden generell bei deutlich höheren Frequenzen als akustische liegen.

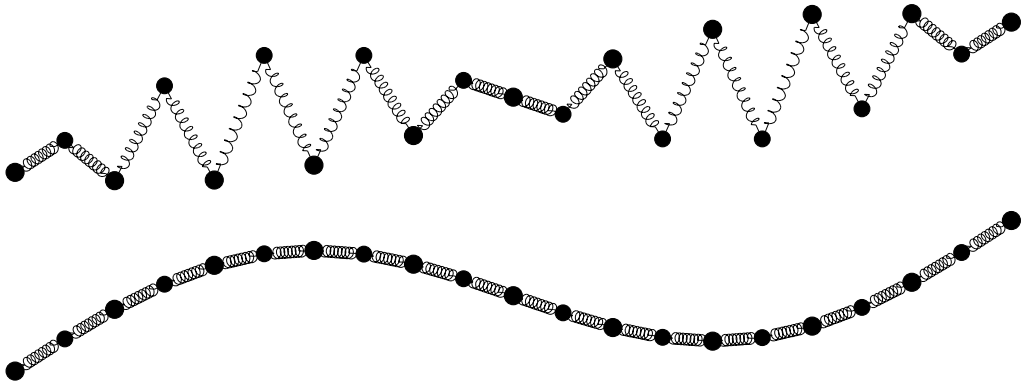


Abbildung 61: Zwei zum gleichen k *gehörende Schwingungsmoden einer linearen Kette, oben transversal optisch, unten transversal akustisch.*

Hat man so ein Objekt einmal zur Verfügung, kann man auch weitere Dinge damit relativ zügig formulieren, das folgende Skript beispielsweise visualisiert eine altbekannte Kniffelei aus der Elektrik (Abbildung 62):

```
u = 4;   e = 1.5;   r0 = 0.25;
x(1,:) = u/2*[1 1 1];
C4 = [1 0 0; 0 0 1; 0 -1 0];      % 90 deg rotation around x
for i=2:4, x(i,:)=x(i-1,:)*C4; end;
x(5:8,:) = -x(1:4,:);             % inversion
for i = 1:7,
  for k = (i+1):8,
    d = x(i,:)-x(k,:);
    if (sqrt(dot(d,d))<(u+0.1))   % edge ?
      spring(x(i,:),x(k,:),e,r0);
      hold on;
    end;
  end;
end;
axis equal, axis off;
```

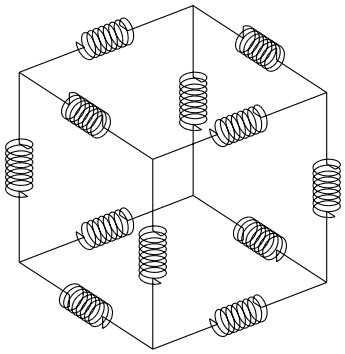


Abbildung 62: Ein Würfel aus 12 gleich großen Widerständen: Berechnen Sie den Gesamtwiderstand zwischen den zwei Endpunkten einer Kante, einer Flächendiagonale und einer Raumdiagonale.

3.5.2 Gewöhnliche Differentialgleichungen

Dreidimensionale Bahnkurven ergeben sich oft als Lösungen von gewöhnlichen Differentialgleichungen in der Physik. Bekannt und einfach ist die beschleunigte Bewegung in der Mechanik, die durch

$$\ddot{r} = a(r) \tag{3.6}$$

beschrieben wird (r und a sind Vektoren). Manchmal ist Gleichung 3.6 analytisch lösbar (freier Fall, schiefer Wurf), im allgemeinen (beliebiges ortsabhängiges Beschleunigungs- bzw. Kraftfeld) jedoch nur numerisch.

Zur numerischen Lösung solcher Anfangswertprobleme[48] der Differentialrechnung stellt MATLAB Funktionen mit unterschiedlichen Verfahren zur Verfügung. Eines der bekanntesten und wohl auch am häufigsten verwendete ist das Runge-Kutta-Verfahren [15], das in den Funktionen `ode45` in vierter und `ode23` in zweiter Ordnung implementiert ist (*ODE* = *O*rdinary *D*ifferential *E*quation). Das Verfahren ist für (Systeme von) Diffentialgleichungen erster Ordnung entwickelt, Differentialgleichungen höherer Ordnung lassen sich dazu umformen, Gleichung 3.6 beispielsweise zu

$$\begin{aligned} \dot{r} &= v \\ \dot{v} &= a(r)\,. \end{aligned} \tag{3.7}$$

Als Anwendungsbeispiel berechnen und zeichnen wir die Bahnkurven von geladenen Teilchen (z. B. Elektronen) im Magnetfeld. Visualisiert werden soll die elektronenoptische Abbildung durch eine 'lange' magnetische Linse, die formal durch ein konstantes Magnetfeld (lange Spule) definiert ist. Elektronen einheitlicher Energie, die von einem Punkt der Objektebene ausgehen und sich ungefähr in Feldrichtung bewegen, treffen auch wieder an einem Punkt der Bildebene zusammen. Für diesen Fall des konstanten Feldes ist das Problem sehr einfach analytisch zu lösen (*Gerthsen*), es werden zwei Bewegungen – parallel und senkrecht zum Feld – überlagert. Die numerische Lösung ist jedoch allgemeiner zu gebrauchen, da dort problemlos auch ein ortsabhängiges Magnetfeld eingesetzt werden kann (Fernsehmonitor).

[48] Die Lösungsfunktion y und ihre Ableitung y' sind bei einem speziellen Wert der unabhängigen Variablen vorgegeben ($x = x_0$ oder $t = t_0$). Gegensatz: Randwertproblem.

Die Elektronen starten am Punkt $r = [0, 0, 0]$ mit der Geschwindigkeit v in die Richtung (φ, Θ). Das Magnetfeld ist konstant und in x-Richtung, φ ist der Winkel zum Magnetfeld, Θ zur x-y-Ebene. Alle Größen werden in 'handlichen' Einheiten angenommen. Im MATLAB-Skript werden zunächst die Anfangswerte bereitgestellt:

```
function [t,u] = elpath(phi,theta)
tspan = 0:0.01:1;
v = 1;
vx = v*cos(phi);
vy = v*sin(phi)*cos(theta);
vz = v*sin(phi)*sin(theta);
u0 = [0 0 0 vx vy vz]; .
```

Die Lösungsfunktion `u` enthält die Orts- und Geschwindigkeitswerte der Bahnkurve, Anfangswert ist `u0`.

Zur Lösung wird `ode45` aufgerufen mit[49]:

```
[t,u] = ode45(@lorentz,tspan,u0);
```

Der Parameter `tspan` gibt an, in welchem Bereich die Lösungsfunktion berechnet werden soll. Wird wie im Beispiel ein Vektor angegeben, so wird – unabhängig von der Stützstellendichte der Berechnung – `u` für genau diese Werte retourniert.

Die von `ode45` benötigte Ableitungsvorschrift `lorentz` liefert v und $v \times B$:

```
function dudt = lorentz(t,ut)
B = [2*pi;0;0];
dudt = [ut(4:6);cross(ut(4:6),B)]; .
```

Statt des konstanten Magnetfeldes könnte an dieser Stelle ein ortsabhängiges eingebaut werden (`ut(1:3)` enthält die aktuellen Ortskoordinaten).

Mit der beschriebenen Funktion `elpath` wird eine einzelne Bahnkurve berechnet, das folgende MATLAB-Skript `elens` ruft diese Funktion mehrfach auf (verschiedene φ und Θ) und zeichnet Bahnkurven, die von zwei verschiedenen Objektpunkten auf der z-Achse bei $z = 0$ und $z = L$ ausgehen:

```
function h = elens
L=0.2;
dL=0.2;
La=0.04;
h=[];
for theta=0:90:270
  for phi=[0 4 8 12 25 35]
    [t,u]=elpath(pi*phi/180,pi*theta/180);
```

[49] Im Beispiel werden keine zusätzlichen Optionen für die numerische Integration festgelegt, sondern die Standardeinstellungen verwendet. Es wäre beispielsweise möglich, die gewünschte Genauigkeit vorzugeben (mehr dazu in den Manuals oder der Online-Hilfe).

```
    if (mod(theta,270)>0) u(:,3)=u(:,3)+L; end;
    hp=plot3(u(:,1),u(:,2),u(:,3),'Color','k');
    hold on;
    h=[h;hp];
    if (phi>20) set(hp,'LineWidth',1); end;
  end;
end; .
```

Schließlich werden noch zwei Pfeile für Objekt und Bild eingezeichnet sowie Koordinatensystem und Betrachtungsrichtung eingestellt:

```
za=[0 L L-La L-La L];
ya=[0 0 -La/2 La/2 0];
xa=zeros(size(za));
ha=plot3(xa,ya,za,xa+1,ya,za);
h=[h;ha];
set(ha,'Color',0.6*[1 1 1],'LineWidth',3);
set(gca,'XTick',[],'YTick',[],'ZTick',[]);
hold off;
axis equal;
box on;
axis([0 1 -dL dL -dL L+dL]);
view(-30,20); .
```

Das Resultat des Skripts zeigt Abbildung 63.

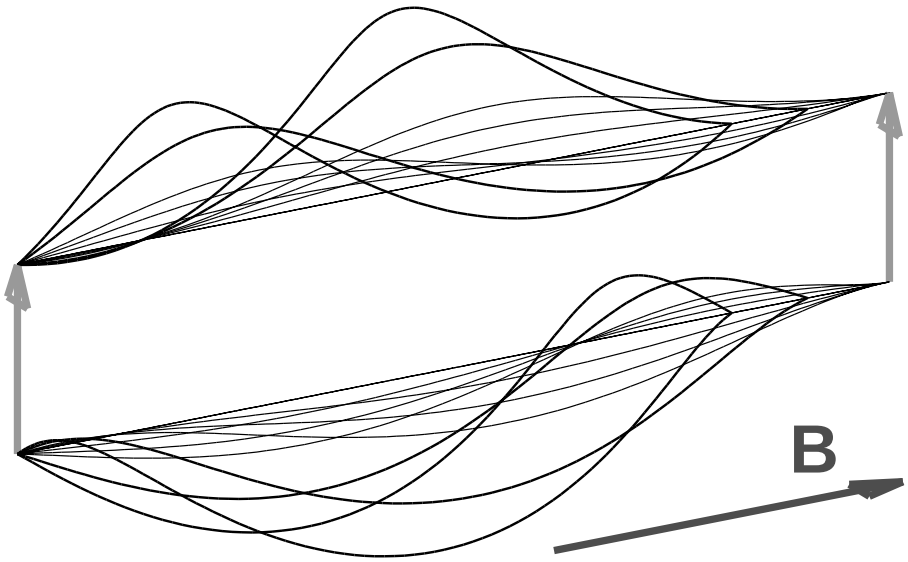


Abbildung 63: Abbildung durch eine 'lange' magnetische Linse. Achsennahe Bahnkurven ergeben ein scharfes Bild, achsenferne führen zu Abbildungsfehlern (Unschärfe).

Durch Wahl einfacher Betrachtungsrichtungen kann man die Bahnkurven auf ausgezeichnete Ebenen projezieren (Abbildung 64). So erhält man mit `view(0,0)` die Projektion auf eine Ebene senkrecht zur Magnetfeldrichtung, das rechte Teilbild der Abbildung 64 zeigt die erwarteten Kreise.

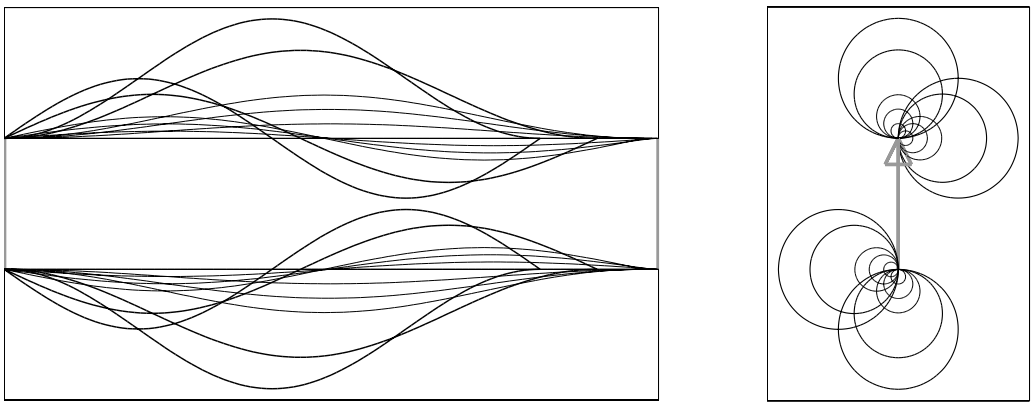


Abbildung 64: Projektion der Bahnkurven auf eine Ebene parallel (links) und senkrecht zum Magnetfeld (rechts), realisiert mit `view(90,0)` *bzw.* `view(0,0)`.

Aus der Exaktheit der Kreise kann man schließen, dass die numerische Lösung einigermaßen genau ist, eine bessere Aussage über die Genauigkeit liefert ein Vergleich zwischen numerischer und analytischer Lösung. Deren Differenz ist in Abbildung 65 geplottet. Die damit festgestellte relative Genauigkeit von etwa 10^{-4} entspricht der Voreinstellung. Für höhere Genauigkeiten (bei dann auch größerer Rechenzeit) sind die entsprechenden Optionen zu ändern.

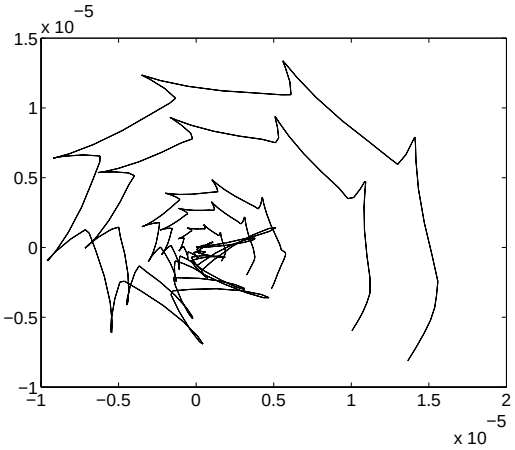


Abbildung 65: Abweichungen der numerischen Lösungen von den exakten (Absolutwerte). Die zugehörigen Funktionswerte liegen zwischen 0 und 0.2, die relativen Abweichungen betragen mithin etwa 10^{-4}.

3.5.3 Potenziale und Felder, partielle Diffentialgleichungen

Äquipotenziallinien (Konturlinien, Höhenlinien) und Feldlinien oder zumindest Feldlinienrichtungen (Vektorpfeile) sind wichtige Visualisierungsmöglichkeiten für Funktionen, die von zwei oder drei Parametern – meist Ortskoordinaten – abhängen. So werden in der Meteorologie Hoch- und Tiefdruckgebiete durch Isobaren veranschaulicht, in der Geographie kennzeichnen Höhenlinienverläufe Berge und Täler. In der Physik werden solche Darstellungen überall dort verwendet, wo man es mit Potenzialen und Feldern im physikalischen Sinne zu tun hat (Gravitation, Wärmeleitung, Elektrostatik, Magnetostatik,...).

Zur Realisierung solcher Graphiken bietet MATLAB unter anderem die Funktion `contour`, mit der ein Konturplot erstellt wird. Mit `gradient` kann die lokale Ableitung in einem Datenfeld näherungsweise berechnet werden, und mit `quiver` wird ein Feld von Vektorpfeilen gezeichnet. Die Anwendung der Funktionen soll wieder an physikalischen Beispielen deutlich gemacht werden, dem elektrostatischen Potenzial und dem elektrischen Feld um verschiedene Elektrodenanordnungen.

Die numerische Modellierung des Potenzialverlaufs wird ebenfalls in MATLAB implementiert. Elektrostatische Potenzialberechnungen basieren auf der Poisson-Gleichung

$$\Delta\varphi = -\rho/(\varepsilon\varepsilon_0)\,, \tag{3.8}$$

deren rechte Seite zu Null wird, wenn keine Ladungen berücksichtigt werden müssen (Laplace-Gleichung). Die elektrisch leitenden Elektroden sind Äquipotenzialflächen, deren Potenzial durch dort angelegte Spannungen auf feste Werte eingestellt ist – ein typisches *Randwertproblem* der Differentialrechnung.

Auf ein diskretes äquidistantes Koordinatennetz umgesetzt, bedeutet $\Delta\varphi = 0$, dass φ im Bereich außerhalb der Elektroden an jeder Stelle gleich dem Mittelwert aus den Umgebungswerten ist[50]. In zwei Dimensionen erhalten wir für φ somit ein lineares Gleichungssystem vom Typ

$$4 \cdot \varphi_{j,k} - \varphi_{j-1,k} - \varphi_{j+1,k} - \varphi_{j,k-1} - \varphi_{j,k+1} = 0\ , \tag{3.9}$$

in drei Dimensionen entsprechend

$$6 \cdot \varphi_{j,k,l} - \varphi_{j-1,k,l} - \varphi_{j+1,k,l} - \varphi_{j,k-1,l} - \varphi_{j,k+1,l} - \varphi_{j,k,l-1} - \varphi_{j,k,l+1} = 0\ . \tag{3.10}$$

An den Elektroden ist φ festgelegt, es kommen dadurch weitere (inhomogene) lineare Gleichungen zum Gleichungssystem.

Wird φ über einem Koordinatennetz der Größe $J{\cdot}K$ (zweidimensional) bzw. $J{\cdot}K{\cdot}L$ diskretisiert, landen wir mithin bei einem inhomogenen linearen Gleichungssystem mit $J{\cdot}K$ bzw. $J{\cdot}K{\cdot}L$ Gleichungen. Wir wollen uns zwei Lösungsverfahren dafür ansehen.

[50] Bei nicht äquidistantem Netz sind die Umgebungswerte mit den reziproken Abständen zu gewichten.

3.5.4 Lösung durch Iterationsverfahren

Ein nahe liegendes, einfaches Lösungsverfahren für das skizzierte *Randwertproblem* bei partiellen Diffentialgleichungen[51] besteht darin, im Potenzialfeld jeden Wert durch den Mittelwert seiner Umgebung zu ersetzen. Im zweidimensionalen Fall mithin

$$\varphi_{j,k}^{(n+1)} = 0.25 \left(\varphi_{j-1,k}^{(n)} + \varphi_{j+1,k}^{(n)} + \varphi_{j,k-1}^{(n)} + \varphi_{j,k+1}^{(n)} \right) . \tag{3.11}$$

Man macht das so oft, bis keine Änderung mehr zu erkennen ist bzw. bis die Änderungen an allen Stellen kleiner sind als ein festgelegter Grenzwert. Wichtig ist, dass die Randwerte, die man festhalten will, nach jedem Durchgang restauriert werden. In unserem Fall sind das die Potenziale der Elektroden. Die Rechenmethode ist äußerst einfach zu implementieren, hat aber leider den Nachteil, dass sie sehr langsam konvergiert. Somit hat sie nur noch geringe praktische Bedeutung[52]. MATLAB vereinfacht die Implementierung zusätzlich dadurch, dass es eine Funktion gibt (`filter2`), die die benötigte Mittelwertersetzung erledigt.

Wir wollen mit dem Verfahren Potenzial und Feld um eine elektrisch geladene Spitze modellieren (genau genommen ist es eine Schneide, da wir nur zweidimensional rechnen). Dafür verwenden wir ein quadratisches Potenzialfeld mit zwei Elektroden, einem spitzen Dreieck (Schneide) in der Mitte mit festgehaltenem Potenzial 1 und dem gesamten Rand mit Potenzial 0 als zweiter Elektrode[53].

Zunächst wird die Elektrodengeometrie festgelegt, als zweidimensionale Matrix, in der die Potenzialwerte der Elektroden eingetragen sind. Das Gebiet außerhalb der Elektroden wird auf einen speziellen, wiedererkennbaren Wert gesetzt. In MATLAB bietet sich dafür der Spezialwert *Not-a-Number*, `NaN`, an, der mit `isnan` leicht wiedererkannt werden kann. Die Implementierung der Geometrie:

```
Size = 40;
[x,y]=meshgrid(1:Size-1);
G = ones(size(x))*NaN;
G((4*(x-Size/2)+y-4*Size/5)<0) = 1;
G(:,1:Size/2) = fliplr(G(:,Size/2:Size-1));
G(1:Size/5,:) = NaN;
G([1,end],:) = 0;
G(:,[1,end]) = 0;
```

Das Dreieck wird über eine Geradengleichung

```
4*(x-Size/2)+y-4*Size/5 = 0
```

[51] Differentialgleichungen dieser Struktur beschreiben beispielsweise auch den Verlauf von anderen 'Potenzial'-Größen wie Temperatur oder Druck.

[52] Da die Methode naturgemäß nur die lokale Umgebung berücksichtigt, nimmt die Rechenzeit überproportional mit der Zahl der Stützstellen zu.

[53] Da man nur mit endlichen Geometrien arbeiten kann, muss man sich für eine sinnvolle Festlegung des Randes entscheiden. Statt das Potenzial dort auf einen festen Wert zu setzen, könnte man auch die Ableitung oder die zweite Ableitung festlegen.

und anschließende Spiegelung `fliplr` definiert.

Bei der Iterationsrechnung wird als Anfangswert für das Potenzial die Geometriematrix eingesetzt, Werte außerhalb der Elektroden (`NaN`) werden mit 0 vorbesetzt.

```
phi = G;
phi(isnan(phi)) = 0;
tri = phi;
ff = [0 1 0; 1 0 1; 0 1 0]/4;
for i=1:n,
    phi = filter2(ff,phi);
    phi(~isnan(G)) = G(~isnan(G));
end;
```

Die Anfangsbedingung wird in `tri` gespeichert (zur späteren Verwendung in der Graphik). Als Filterfunktion `ff` wird eine 3*3-Matrix definiert, die die Iterationsvorschrift Gl. 3.11 umsetzt. Die Mittelung wird dann `n`-mal ausgeführt, die Randbedingungen werden jeweils restauriert.

Aus dem Potenzial wird mit `gradient` das Feld berechnet und mit `quiver` in Form von Vektorpfeilen gezeichnet:

```
[ex,ey] = gradient(phi);
quiver(ex,ey,1.2,'k');
axis equal;  axis off;  hold on;
[c,h] = contourf(tri,[0 0.9]);
set(h(2),'FaceColor',0.2*[1 1 1]);
set(h(1),'FaceAlpha',0,'LineWidth',1.5);
contour(phi,8); .
```

Mit `contourf` wird die Dreieckselektrode (gespeichertes Anfangspotenzial `tri`) als gefüllte Fläche gezeichnet, die Randelektrode als Linie, `'FaceAlpha',0,` macht die zwischenliegende Fläche durchsichtig. Die Äquipotenziallinien werden schließlich mit `contour` eingezeichnet. Das Ergebnis für `n` = 10000 zeigt Abbildung 66. Bei dieser großen Zahl von Iterationen ist die maximale Abweichung im Potenzialfeld vom richtigen Ergebnis etwa 10^{-14}, bei 1000 Iterationen wäre dies etwa 10^{-4}, 10^{-2} wäre bei 533 Iterationen erreicht.

3.5.5 Direkte Lösung des linearen Gleichungssystems

Das beschriebene Iterationsverfahren lässt sich auch in anderen Programmiersprachen wie C oder C++ ohne zusätzliche Hilfsmittel relativ einfach implementieren. Für die direkte Auflösung des linearen Gleichungsystems benötigt man dagegen zusätzliche Bibliotheksfunktionen, die das leisten. MATLAB stellt zur Lösung die Funktion `mldivide` zur Verfügung (kurz als '\' zu schreiben). Ein durch eine Matrixgleichung definiertes lineares Gleichungssystem $A{\cdot}X = B$ hat in MATLAB die Lösung $X = A\backslash B$, A ist darin die Koeffizientenmatrix, X und B sind Spaltenvektoren.

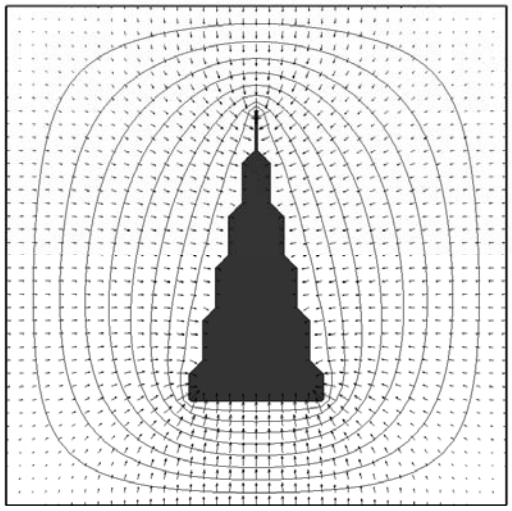


Abbildung 66: Potenzial- und Feldverlauf in der Umgebung einer elektrisch geladenen Schneide.

Wir erproben das Verfahren an einem Beispiel aus den *Laborversuchen zur Physik*, dem *Elektrolytischen Trog*. Im Versuch werden Potenzialverläufe experimentell bestimmt. Eine der dabei verwendeten Anordnungen, ein Dreielektrodensystem (Triode aus Kathode, Gitter und Anode), wollen wir modellieren.

Die Elektrodengeometrie wird als Funktion formuliert, die eine Matrix `G` liefert:

```
function G = geotriode(L)
  if nargin==0, L=6; end;
  xm = 8*L-1;
  ym = 6*L-1;
  y1 = L;
  y2 = 2*L;
  y3 = 4*L;
  x1 = 2*L;
  x2 = 3*L;
  x3 = 5*L;
  x4 = 6*L;
  G = ones(ym,xm)*NaN;
  G([1,end],:) = 0;
  G(:,[1,end]) = 0;
  G(y1,x1:x4) = 0;
  G(y2,[x1:x2,x3:x4]) = -4;
  G(y3,x1:x4) = 6;
```

Aus dem Skalierungsfaktor `L`, der die Matrixgröße `(ym,xm)` definiert, werden handliche Koordinaten `xi, yi` berechnet. Die Geometrie-Matrix wird mit `NaN` vorbesetzt, dann werden die Potenziale der Elektroden fest gelegt – Rand und Kathode auf 0, Gitter auf -4, Anode auf 6. Das entspricht den beim *Elektrolytischen Trog* anliegenden Spannungswerten.

Bei der Lösung des linearen Gleichungssystems wird ausgenutzt, dass die Koeffizientenmatrix äußerst dünn besetzt ist und dass MATLAB die meisten seiner Funktionen auch für solche *Sparse Matrices* implementiert hat. Dünn besetzte Matrizen werden über die Indizes und die zugehörigen Werte der von Null verschiedenen Matrixelemente definiert, hier durch die drei Vektoren `i, j, s`.

```
G = geotriode;
[m,n] = size(G);
i = (1:m*n)';
j = i;
s = ones(size(i));
[u,v] = find(isnan(G));
index = (v-1)*m+u;
for k = [-1,1,-m,m],
    i = [i;index];
    j = [j;index+k];
    s = [s;-0.25*ones(size(index))];
end;
A = sparse(i,j,s);
G(isnan(G)) = 0;
y = A\G(:);
z = reshape(y,m,n);
```

Zunächst werden die Diagonalelemente der Matrix (Indizes `j = i`) mit 1 besetzt, dann im Innenbereich (`isnan`) alle Nachbarelemente (`[-1,1,-m,m]`) mit -0.25. Aus den drei Vektoren `i, j, s` wird die *Sparse Matrix* `A` gebildet und schließlich wird das Gleichungssystem $A \cdot y = G$ gelöst, nachdem die `NaN` jeweils durch 0 ersetzt wurden. Der Lösungsvektor `y` muss zum Schluss noch in eine Matrix der richtigen Dimension umgeformt werden (`reshape`). Das Ergebnis der Rechnung ist in Abbildung 67 dargestellt.

3.5.6 Gitternetzdarstellung

Funktionen oder Daten, die von zwei Parametern abhängen, können durch teppichartige Flächen im Raum visualisiert werden. MATLAB bietet dafür eine größere Zahl von Funktionen an, einfache für die Darstellung einzelner Flächenelemente wie `patch`, `fill` oder `fill3`, allgemeine Basisfunktionen wie `surface` und spezialisierte wie `surf`, `mesh` und `waterfall`.

An zwei schon bekannten Beispielen, den experimentellen Daten zur Kristallzusammensetzung von Lithiumniobat und der Interferenz von Wellen sollen zunächst einige Möglichkeiten der Gitternetzdarstellung in MATLAB aufgezeigt werden.

MESH Die zuständige Funktion `mesh` erwartet die Daten als zweidimensionale Felder `X`, `Y`, `Z` je gleicher Größe. Sinnvollerweise sollten die Daten darin in zwei der drei Raumrich-

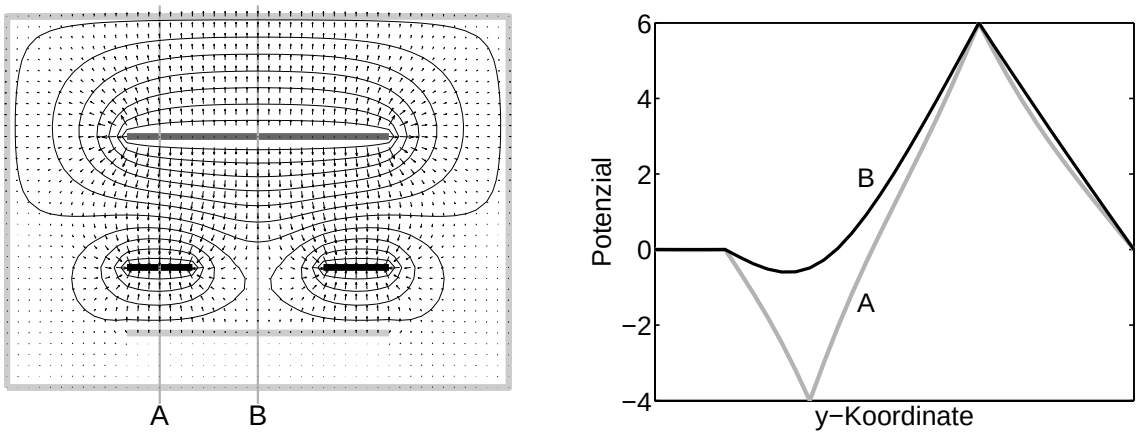


Abbildung 67: Potenzial- und Feldverlauf in einer Dreielektrodenanordnung – Triode aus Kathode, Gitter und Anode (linkes Bild). Hellgrau Rand und Kathode (Potenzial=0), schwarz das Gitter (Potenzial=-4), dunkelgrau die Anode (Potenzial=6). Rechts der Potenzialverlauf entlang der im linken Bild eingezeichneten Linien A und B – der so genannte Durchgriff des Anodenpotenzials auf den Bereich zwischen Kathode und Gitter verringert die Potenzialbarriere für Elektronen.

tungen geordnet sein, da `mesh` jeweils benachbarte Datentripel durch Geraden verbindet. `X` und `Y` können auch (geordnete) Vektoren sein – dann wird ein Rechteckgitter angenommen – oder ganz fehlen – dann wird ein äquidistantes ganzzahliges Gitter verwendet. Als weiterer Parameter ist ein Feld mit Farbwerten möglich, das die Farbe der Linienstücke festlegt, fehlt dies, wird `Z` dafür benutzt.

GRIDDATA Wenn die Daten ungeordnet vorliegen, können sie mithilfe der Funktion `griddata` auf das gewünschte Gitter umgerechnet werden. Für die Lithiumniobat-Daten leistet dies (sowie die Umrechnung auf die gewünschten physikalischen Größen):

```
function [xi,yi,zi] = lnread
[x,y,z] = textread('130695.dat','%*f%f%f%*f%*f%f%*f',...
                'headerlines',2);
x = x/1000;
y = (y-1500)/2000;
z = (z/199000+2.9)*49.5/3;
xr = linspace(min(x),max(x),30);
yr = linspace(min(y),max(y),50);
[xi,yi,zi] = griddata(x,y,z,xr,yr','cubic'); .
```

Die mit `[x,y,z]=lnread` gelesenen und umgerechneten Daten werden mit

```
h = mesh(x,y,z);
```

```
set(h,'EdgeColor','k')
```

als Gitternetz dargestellt, die Farbe der Linien wird einheitlich auf schwarz gesetzt. In einer Farbdarstellung könnte man durch die Farbe den Funktionswert zusätzlich kennzeichnen oder aber die Variation einer weiteren Größe über dem Parameterraum visualisieren.

WATERFALL Eine Variante der Gitternetzdarstellung – Netzlinien nur in einer der beiden Richtungen des Basisgitters – lässt sich mit `waterfall` erstellen. Dadurch werden Netzlinien nur für die Zeilen, nicht für die Spalten der Datenfelder gezeichnet. Durch Angabe der gespiegelten Matrizen lässt sich das vertauschen.

Die Wirkung der verschiedenen Darstellungsarten – Gitter, Zeilen, Spalten – veranschaulicht Abbildung 68.

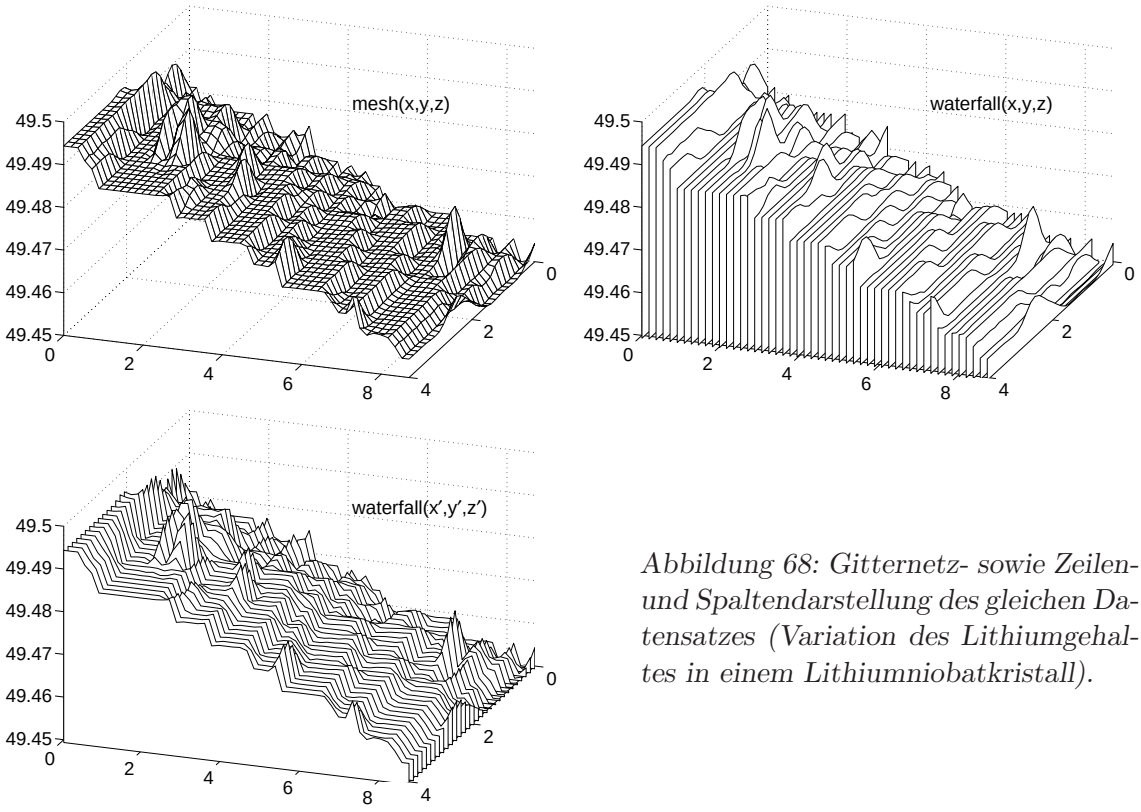


Abbildung 68: Gitternetz- sowie Zeilen- und Spaltendarstellung des gleichen Datensatzes (Variation des Lithiumgehaltes in einem Lithiumniobatkristall).

Die Wirkung von Farbe bzw. Graustufen und unterschiedlichen Linienbreiten bei der Darstellung mit Gitternetzen wird bei 'gleichmäßigeren' Daten deutlich. Abbildung 69 vergleicht die Standardgraustufeneinstellung mit schwarz, Abbildung 70 unterschiedliche Linienbreiten.

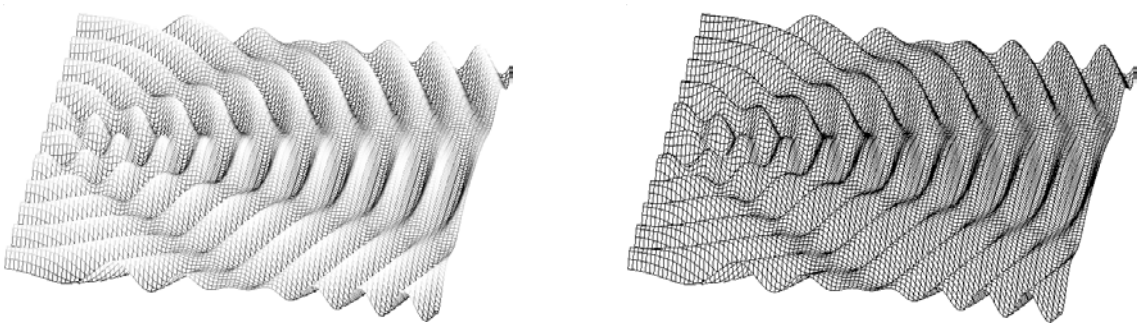


Abbildung 69: Gefärbtes und einfarbiges Gitternetz: links die Standardeinstellung mit Graustufen, der Farbwert ist proportional zum Funktionswert, rechts einheitlich schwarz.

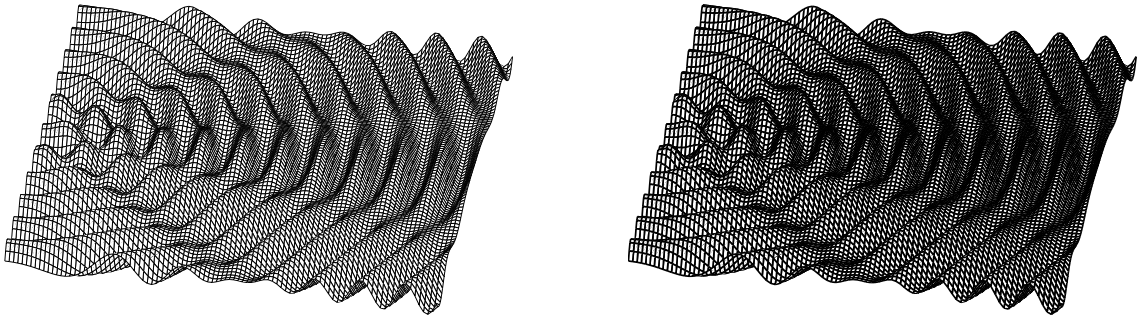


Abbildung 70: Wirkung unterschiedlicher Linienbreiten, links 0.1, rechts 2 (die Voreinstellung ist 0.5).

3.5.7 Flächendarstellung

SURF Gitternetze mit gefüllten Flächen erstellt man über die Anweisung `surf`, die weitgehend wie `mesh` funktioniert.

COLORMAP Die für die Flächen verwendeten Farben werden mit `colormap` festgelegt. *Colormaps* sind Tabellen mit RGB-Farbwerten. Aus den als Parameter zu `surf` angegebenen Funktionswerten (z) oder Farben wird der Tabellenindex berechnet und damit die Farbe ausgewählt. MATLAB verfügt über eine ganze Reihe von vordefinierten Standardfarbtabellen. Bevor man eigene Farbtabellen definiert, sollte man damit experimentieren. Für eine Graustufendarstellung eignet sich `gray`, bei farbigen Bildern hat man mehr Auswahl: hohen Farbkontrast bieten `hsv` und `jet`, Helligkeitskontrast mit geringer Einfärbung in nur einer Farbe `copper`, `bone` und `pink`, dazwischen liegen verschiedene Farbsätze, die mit unterschiedlichem Kontrast über Teilbereiche des Farbspektrums verlaufen wie `cool`, `hot`, `summer`, `winter` usw. Für Spezialanwendungen können `flag`, `vga`, `colorcube` und `prism` nützlich sein.

SHADING Darüber hinaus kann der Farbübergang in den mit `surf` definierten viereckigen Flächenelementen mit `shading` eingestellt werden:

`shading faceted,` die Voreinstellung, färbt die Einzelflächen in jeweils einheitlicher Farbe und zeichnet zusätzlich ein schwarzes Gitternetz.

`shading flat` färbt nur die Einzelflächen in je einheitlicher Farbe, entspricht also `faceted` ohne Gitternetz.

`shading interp` interpoliert die Farben der Einzelflächen linear zwischen den Eckpunktwerten. Bei dieser Art des Farbverlaufs ist allerdings Vorsicht geboten, wenn eine PostScript-Ausgabe erstellt werden soll. Die meisten PostScript-Interpreter werden durch den MATLAB-Code überfordert. Im allgemeinen ist es besser, mit höherer Stützstellendichte zu arbeiten und die Funktionswerte zu interpolieren und damit einen interpolierten Farbverlauf zu simulieren.

Mit dem Aufruf von `shading` legt man bestimmte Eigenschaften für alle graphischen Objekte im aktuellen Fenster fest. So setzt beispielsweise `shading interp` unter anderem die Eigenschaft `EdgeAlpha` für alle Objekte auf 0. Oft ist es besser, diese Objekteigenschaften individuell für Einzelobjekte oder Objektgruppen direkt festzulegen, da dadurch Objekte unterschiedlich konfiguriert werden können, außerdem auch gewisse Nebenwirkungen verhindert werden.

Die Wirkung einer `surf`-Darstellung der schon bekannten Daten mit unterschiedlichem `shading` zeigt Abbildung 71. Als `colormap` wurde `gray` eingestellt.

Die Graphiken werden erstellt z. B. mit

```
[x,y,z] = lnread;
h = surf(x,y,z);
colormap gray;
shading flat; ...
```

Auch hier wird die Wirkung bei der Darstellung von regelmäßigen Objekten noch etwas deutlicher, Abbildung 72 zeigt das Wellenfeld mit unterschiedlichen `shading`-Einstellung. Zum Vergleich eine Darstellung mit einheitlicher Farbe und zusätzlicher Beleuchtung (Teilbild rechts unten), realisiert mit

```
h = surf(z);
set (h,'FaceColor',0.6*[1 1 1]);
lightangle(130,50);
lighting gouraud; ...
```

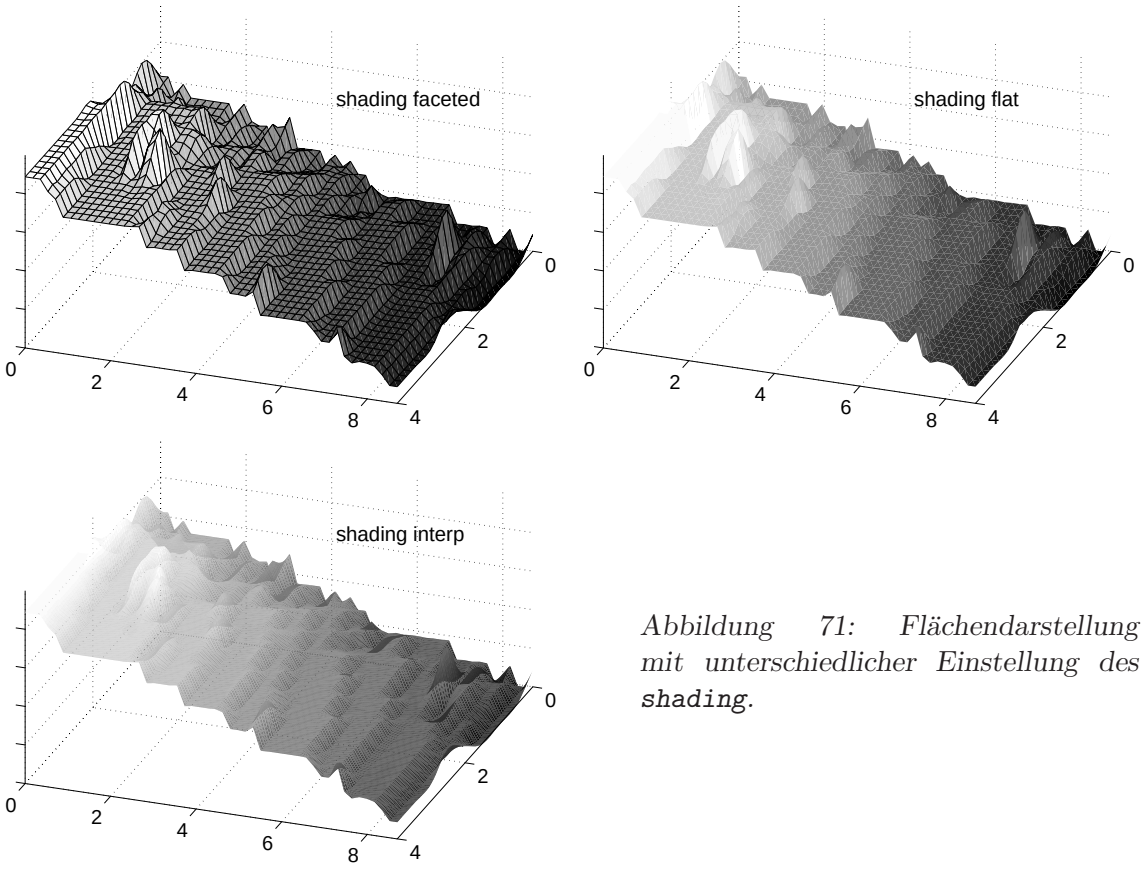


Abbildung 71: Flächendarstellung mit unterschiedlicher Einstellung des `shading`.

PCOLOR Durch Wahl der entsprechenden Betrachtungsrichtung – `view(0,90)` – kann der Eindruck einer ebenen quasizweidimensionalen Fläche erzielt werden, für diese ebene Darstellung gibt es die spezielle Funktion `pcolor`. Auch dafür sind alle Flächeneigenschaften einstellbar. In Abbildung 73 ist unterschiedliches `shading` eingestellt, in Abbildung 74 sind die Gitternetzlinien bei `shading faceted` durch `set(h,'EdgeColor',...)` unterschiedlich eingefärbt.

BRIGHTEN Die Gesamthelligkeit der Flächengraphik lässt sich mit `brighten` verändern, positive Werte hellen auf, negative machen das Bild dunkler. Abbildung 75 zeigt die Wirkung der Werte -0.6, -0.2, 0.2 und 0.6.

CONTOUR Eine weitere Darstellungsart für die Analyse von 2-parametrigen Funktionen oder Daten sind Kontur- oder Höhenlinien, d. h. Linien, die jeweils gleiche Funktionswerte miteinander verbinden. MATLAB bietet dafür die Funktion `contour` für einfache und `contourf` für gefüllte Konturflächen. Die Zahl der Konturlinien oder die gewünschten Funktionswerte, an denen Konturlinien verlaufen sollen, können als Parameter angegeben werden. Abbildung 76 zeigt die Lithiumniobatdaten in diesen Darstellungsarten.

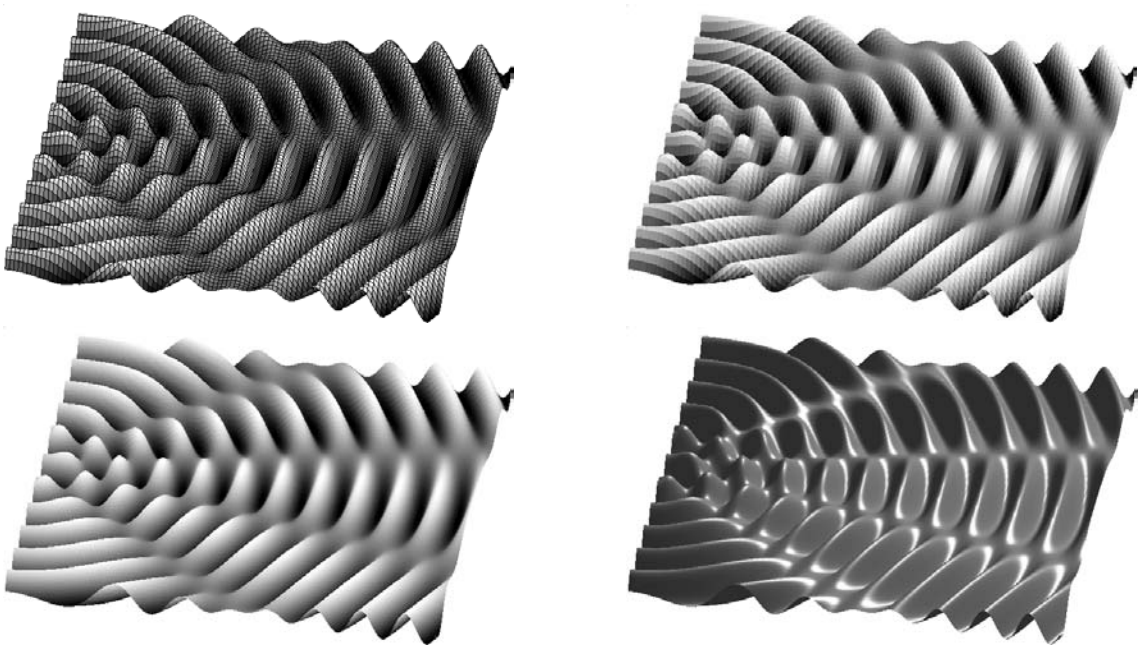


Abbildung 72: Wirkung unterschiedlicher `shading`*-Einstellungen bei einem regelmäßigen Objekt. Rechts unten mit zusätzlicher Beleuchtung und einheitlicher Farbe.*

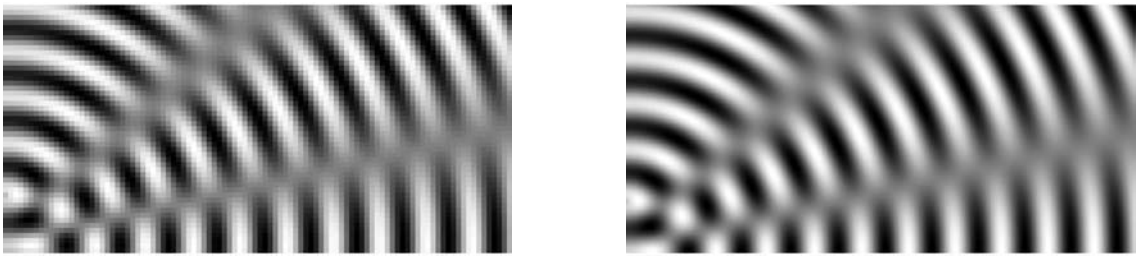


Abbildung 73: `Pcolor`*-Darstellung mit unterschiedlichem* `shading`*: links* `flat`*, rechts* `interp`*.*

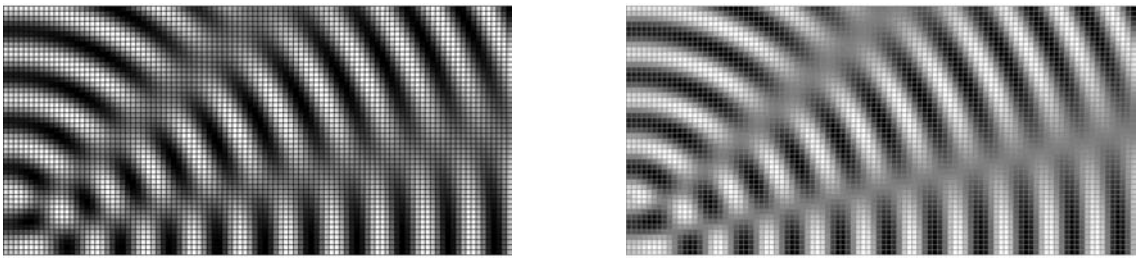


Abbildung 74: `Pcolor`*-Darstellung mit unterschiedlichem Gitter:* `shading faceted` *links in Schwarz, rechts in Grau.*

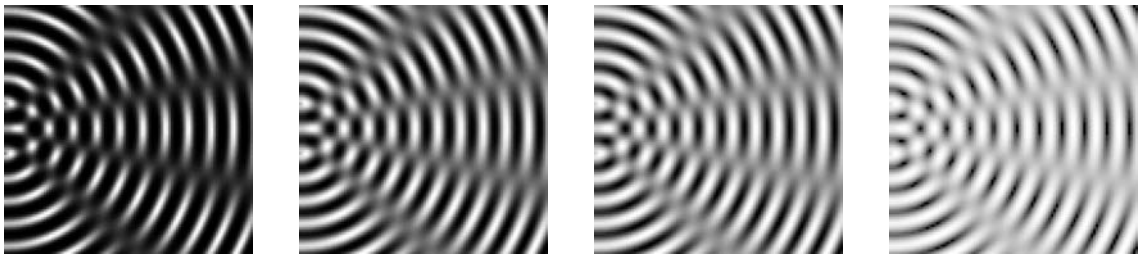


Abbildung 75: Helligkeitsveränderung durch `brighten`*, von links nach rechts:* `brighten(-0.6), (-0.2), (0.2), (0.6)` .

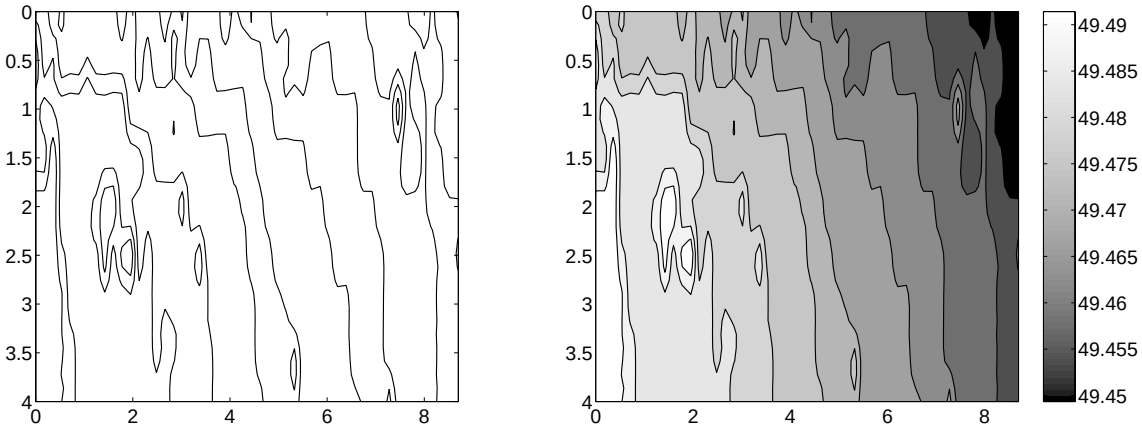


Abbildung 76: Lithium-Gehalt in einem Lithiumniobatkristall, links mit `contour`*, rechts mit* `contourf` *dargestellt.*

COLORBAR, CLABEL Zur Funktions- bzw. Wertezuordnung kann ein Farbmaßstab angebracht werden, das erledigt die Anweisung `colorbar` für einen vertikalen Balken wie im Beispiel oder `colorbar('horiz')` für einen horizontal angeordneten. Sollen die zugehörigen Funktionswerte direkt an die Höhenlinien angeschrieben werden, kann das mit `clabel` veranlasst werden.

CONTOUR3 Auch an 3D-Gebirgen lassen sich Konturlinien anbringen, `contour3` ist die dafür zuständige Funktion. Abbildung 77 zeigt zwei Beispiele, links Konturlinien an einer 3D-Fläche in Graustufendarstellung, rechts an der gleichen, jetzt einfarbig grauen, aber zusätzlich beleuchteten Fläche.

Das Skriptfragment für das rechte Teilbild:

```
[c,h] = contour3(x,y,z,10);
hold on;
hs = surf(x,y,z);
set(hs,'FaceColor',0.9*[1 1 1]);
```

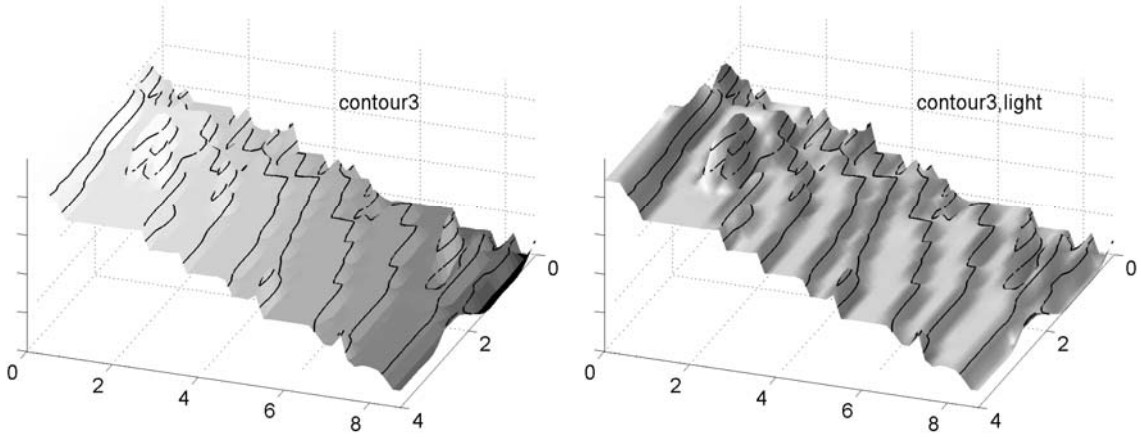


Abbildung 77: Kombination von `surf` *mit* `contour3`*, links in Grautonabstufung, rechts einfarbig mit Beleuchtung.*

```
set(h,'EdgeColor','k','LineW',1.5);
hold off;
light;  lighting gouraud; .
```

Für das linke Teilbild werden die `FaceColor-` und `light`-Zeilen weggelassen.

CONTOURC Das von MATLAB berechnete Wertefeld für die Konturlinien, das beim Aufruf der verschiedenen Konturzeichenfunktionen einer der Rückgabeparameter ist, kann mit `contourc` auch direkt angefordert werden, ohne dass eine Graphik generiert wird. Damit hat man die Möglichkeit, Konturlinien weiterzuverarbeiten, beispielsweise nach Wunsch zu interpolieren, oder die Werte für andere Zwecke zu nutzen.

3.5.8 Rastertunnelmikroskop: HOPG

Als weitere Anwendungsbeispiele sollen experimentelle Daten aus dem Fortgeschrittenen-Praktikum Physik dienen, Messungen mit dem Rastertunnelmikroskop (RTM). Zunächst an Graphit (HOPG) gewonnene Messdaten mit atomarer Auflösung. Die Rohdaten (als TIFF-Datei vorliegend) sind sehr verrauscht, Abbildung 78 zeigt einen Ausschnitt.

FILTER2 Mit dem Praktikumsgerät ist keine bessere als atomare Auflösung möglich, man kann die Daten mithin mit einer vergleichsweise breiten Filterfunktion filtern, um das Signal-Rausch-Verhältnis zu verbessern. Wir wählen als Filter ein domartiges Rotationsparaboloid der Form

$$f(x,y) = 1 - (x^2 + y^2)/r_f^2 \tag{3.12}$$

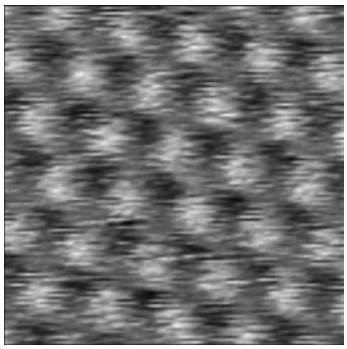


Abbildung 78: Messdaten vom Rastertunnelmikroskop: Oberfläche von Graphit (HOPG-Probe) in atomarer Auflösung.

mit dem Basisradius $r_f = 10$. Nachstehendes Skript-Fragment definiert die Filtermatrix, liest die Daten mit `imread` und glättet sie mit `filter2`:

```
rf = 10;
[xf,yf] = meshgrid(-rf:rf);
ff = rf*rf-xf.*xf-yf.*yf;
ff = ff.*(ff>0);
ff = ff/sum(sum(ff));
[z8,cmap] = imread('atomar14.tif');
z = filter2(ff,double(z8)); .
```

Das Ergebnis ist durchaus erfreulich (Abbildung 79). Die graphische Darstellung erledigt – wie auch schon beim Rohdatenbild (dort ohne Filter) –

```
h = pcolor(z(60:190,60:190));
axis image;  colormap gray;  shading interp; .
```

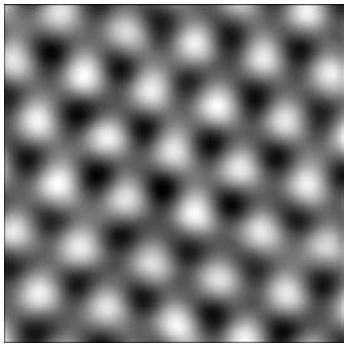


Abbildung 79: Mit `filter2` geglättete Messdaten vom Rastertunnelmikroskop (Originaldaten in Abbildung 78).

SURF, COLORMAP, CONTOUR3 Die Flächendarstellung mit `pcolor` lässt sich quantitativ geometrisch auswerten, einen eher qualitativen Eindruck der Höhengeometrie vermitteln Schrägansichten mit `surf`. Abbildung 80 demonstriert dies an einem Datenausschnitt dargestellt mit `surf` in 3 Varianten – mit voreingestellter und umgekehrter

Colormap sowie zusätzlichen 3D-Konturlinien. Die Umkehrung der *Colormap* (Spiegelung der Matrix) wird mit

```
colormap(flipud(colormap(gray)));
```

erreicht.

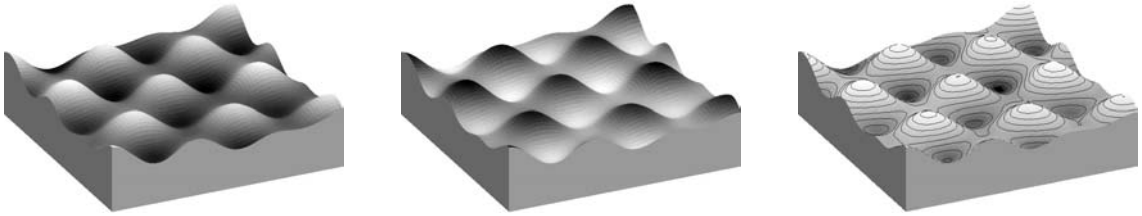


Abbildung 80: RTM-Daten (HOPG), dargestellt mit `surf`*. Links:* `colormap gray`*, Mitte: gespiegelte Colormap, rechts: mit 3D-Höhenlinien.*

LIGHTING Einen materialbetonenden Effekt erreicht man durch einheitliche Färbung und zusätzliche Beleuchtung. Abbildung 81 verdeutlicht dies. Als Farbe ist mit

```
set(h,'FaceColor',0.6*[1 1 1]);
```

einheitliches Grau eingestellt, die Beleuchtungsart wird durch verschiedene Einstellungen von `lighting` variiert (`flat`, `gouraud`, `phong`).

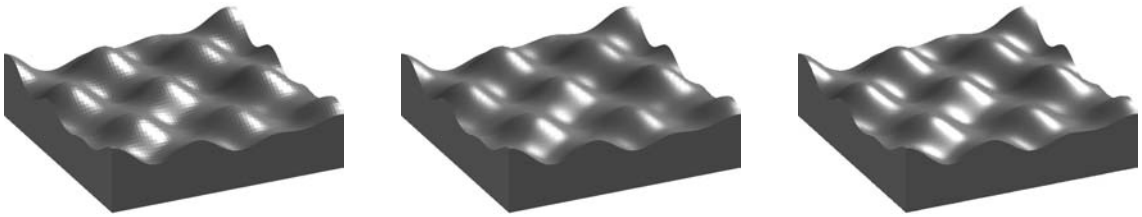


Abbildung 81: `Surf`*-Darstellung mit Beleuchtung. Links:* `lighting flat`*, Mitte:* `lighting gouraud`*, rechts:* `lighting phong`*.*

3.5.9 Rastertunnelmikroskop: CD

Experimentell erheblich einfacher zugänglich sind gröbere Strukturen wie die Oberfläche einer Compact Disk. Die Rohdaten einer Messung zeigt das linke Teilbild von Abbildung 82. Das Rauschen lässt sich auch in diesem Fall durch eine geeignete Filterung verringern. Als Filterfunktion wird wieder ein Rotationsparaboloid verwendet, diesmal mit einem Basisradius von 4. Die geglätteten Daten zeigt das rechte Teilbild der Abbildung.

Sollen 3D-Daten geometrisch ausgewertet werden, sind ebene Projektionen auf eine ausgezeichnete Fläche oft günstiger, Abbildung 83 zeigt eine Auswahl verschiedener Möglich-

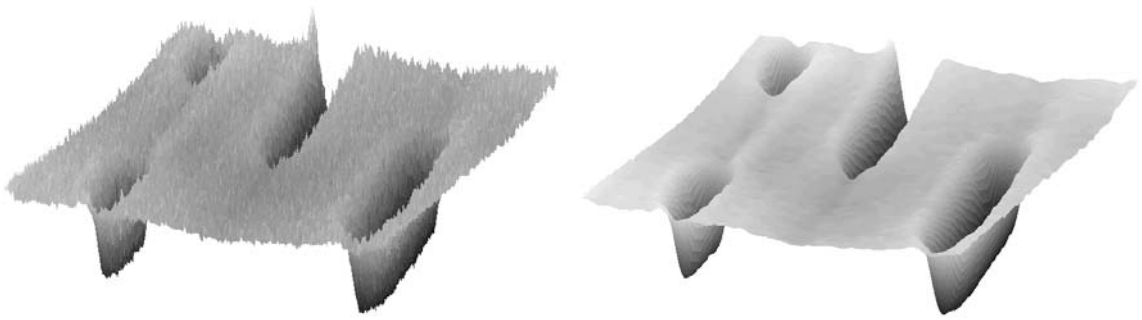


Abbildung 82: Rastertunnelmikroskopaufnahme einer Compact Disk. Links die Rohdaten, rechts nach Filterung.

keiten. Links eine `pcolor`-Darstellung mit `colormap gray` – die Verteilung der z-Werte führt zu relativ harten Hell-Dunkel-Kontrasten.

COLORMAP(CONTRAST) Will man bessere Informationen über Zwischenwerte erhalten, kann man mit

```
colormap(contrast(z));
```

eine gleichmäßigere Verteilung der Helligkeitswerte erreichen (Histogramm-Egalisierung). Das Ergebnis – mit `brighten` etwas aufgehellt – zeigt das mittlere Teilbild der Abbildung. Rechts zum Vergleich die Kontur-Darstellung der Messwerte.

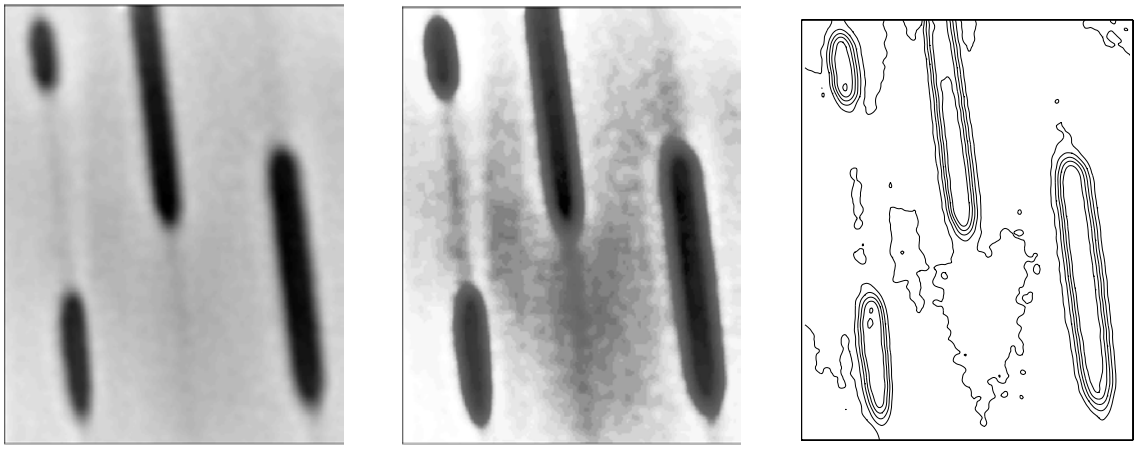


Abbildung 83: Ebene Darstellungen der CD-Daten. Links: `pcolor` mit `colormap gray`, Mitte: `colormap(contrast(z))`, rechts: `contour(z,6)`.

LIGHT Teilinformationen aus Messdaten können durch verschiedene Maßnahmen gezielt hervorgehoben werden. Beispiele zeigen die nächsten Abbildungen. So kann die nach

der Filterung noch vorhandene Oberflächenrauhigkeit durch Beleuchtung hervorgehoben werden. Das linke Teilbild von Abbildung 84 zeigt die Wirkung von zwei Lichtquellen, die mit

```
light('Position',[3 0 1]);
light('Position',[-1 -1 -1]);
lighting gouraud;
```

an zwei verschiedene Positionen gesetzt wurden.

NaN Durchsicht zu tieferliegenden Bereichen kann man dem Betrachter durch geeignete 'Ausschnitte' verschaffen – die z-Werte in diesen Bereichen werden auf `NaN` (Not a Number) gesetzt und dadurch nicht dargestellt (Abbildung 84 rechts).

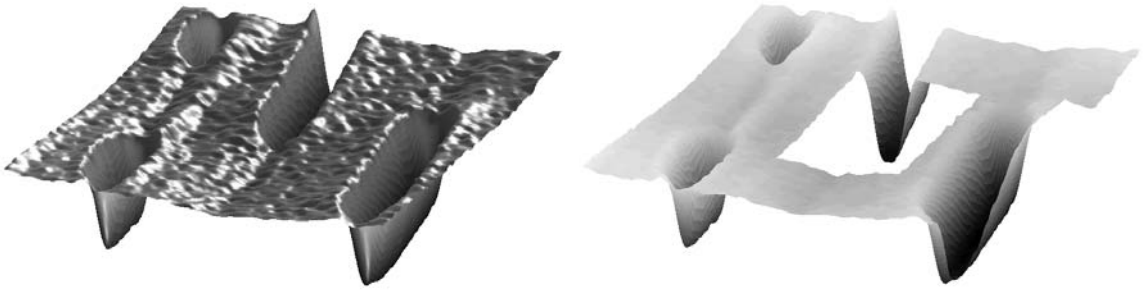


Abbildung 84: Wirkung von 'Light' und 'NaN'. Links: Beleuchtung mit zwei Lichtquellen, rechts: Einblick durch Ausschnitte, die z-Werte wurden dort auf NaN gesetzt.

ALPHADATA, TRANSPARENCY Eine weitere Möglichkeit, in verdeckte Bereiche zu sehen, ist es, die Flächendarstellungen teiltransparent zu machen. Voreingestellt sind alle Flächenobjekte undurchsichtig, die Transparenz ist aber sehr detailliert konfigurierbar. Als Objekteigenschaft eingestellt wird *Alpha*, physikalisch gesehen die Absorption des Objekts. Die Werte liegen im Bereich [0,1], 0 bedeutet totale Transparenz, das Objekt ist nicht sichtbar, 1 komplette Absorption, die Voreinstellung. Bei einem Flächenobjekt legt `FaceAlpha` die Absorption aller Teilflächen, `EdgeAlpha` die der Kanten fest. Darüber hinaus kann durch `AlphaData` ein ortsabhängiger Absoptionsverlauf eingestellt werden, als *Data* muss ein Feld übergeben werden, das in seiner Größe den Flächendaten (z) entspricht. In diesem Fall ist dann `FaceAlpha` auf `flat` oder `interp` zu setzen. Abbildung 85 zeigt Beispiele. Im Teilbild links oben wurde ein festes `FaceAlpha` von 0.7 eingestellt. Bei den beiden anderen Bildern wurde die Absorption ortsabhängig gewählt, rechts umgekehrt proportional zur Höhe z mit

```
set(h,'AlphaData',-z,'FaceAlpha','flat'); ,
```

im unteren Bild von vorne nach hinten – mit wachsendem x und y – gleichmäßig zunehmend durch

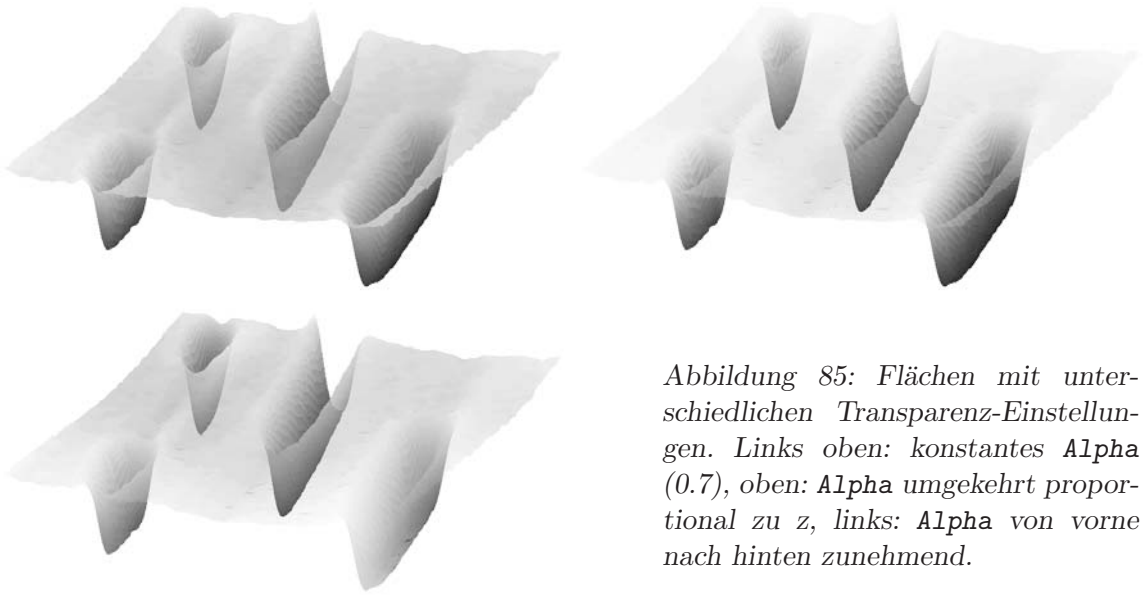


Abbildung 85: Flächen mit unterschiedlichen Transparenz-Einstellungen. Links oben: konstantes `Alpha` *(0.7), oben:* `Alpha` *umgekehrt proportional zu z, links:* `Alpha` *von vorne nach hinten zunehmend.*

```
[x,y]=meshgrid(1:size(z,2),1:size(z,1));
set(h,'AlphaData',x+y,'FaceAlpha','flat'); .
```

Wie aus den Beispielen hervorgeht, brauchen die `AlphaData` nicht auf den Bereich [0,1] normiert zu sein, das erledigt MATLAB. Gegebenenfalls kann hier eingreifen und die Grenzen des Alpha-Wertebereichs mit

```
set(gca,'ALim',[amin,amax]);
```

gezielt anders – auf die individuellen Werte `amin` und `amax` – festlegen.

3.6 Volumenmodellierung

Bei Messdaten oder Funktionen, die von zwei Parametern abhängen, ist eine Visualisierung durch teppichartige Flächendarstellungen naheliegend. MATLAB ist natürlich nicht darauf beschränkt. Es lassen sich grundsätzlich beliebige geometrische Oberflächen formulieren und damit z. B. auch Volumenobjekte modellieren. Am einfachsten ist das, wenn sich die Oberflächen über Gittern aus kartesischen oder polaren Koordinaten als Sammlung von Vierecken definieren lassen. Dann sind die vertrauten Funktionen `surf`, `mesh` usw. zuständig. Für Dreiecksmengen gibt es `trisurf` und `trimesh`, beliebige Polygone[54] im Raum lassen sich mit `patch` oder `fill3` zeichnen.

[54] Man ist dabei nicht auf ebene Polygone beschränkt, allerdings ist das Ergebnis bei gefüllten nichtebenen Polygonen nicht unbedingt das gewünschte.

3.6.1 MATLAB-Basisobjekte

MATLAB stellt drei einfache Volumenobjekte als Funktionen zur Verfügung, Kugel, Zylinder und Ellipsoid. Die zugehörigen Koordinatensätze werden mit

```
[x,y,z] = sphere; ,
[x,y,z] = cylinder; ,
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr);
```

erstellt. Abbildung 86 zeigt ihre Volumendarstellung mit `mesh(x,y,z)`.

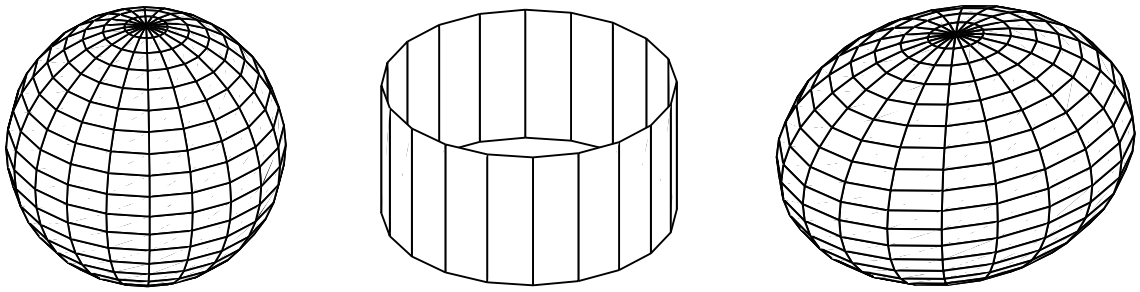


Abbildung 86: Geometrische Basisobjekte in MATLAB: `Sphere`*,* `Cylinder` *und* `Ellipsoid`*.*

Die Grundformen lassen sich durch Bearbeitung der Koordinatensätze modifizieren, so kann man beispielsweise aus `sphere` durch Multiplikation mit `xr`, `yr`, `zr` und Addition von `xc`, `yc`, `zc` nachträglich ein Ellipsoid machen.

Das `cylinder`-Objekt ist nicht auf Zylinder im engeren Sinne beschränkt, sondern kann beliebige Rotationskörper erstellen. Dazu wird die Kontur des gewünschten Objekts als Parametervektor angegeben. Die angegebenen Radien sind zunächst äquidistant auf der Rotationsachse (z-Bereich [0,1]) verteilt, nachträglich können die z-Werte angepasst werden. Abbildung 87 zeigt drei einfache Beispiele.

Der Kegel wird mit

```
[x,y,z] = cylinder([0.5,0]); ,
```

der Kegelstumpf mit

```
[x,y,z] = cylinder([1,0.6,0]);
z([2,3],:) = 1; ,
```

der Ring mit

```
[x,y,z] = cylinder([1,1,0.6,0.6,1]);
z([2,3],:) = 0.5;
z([4,5],:) = 0;
```

formuliert.

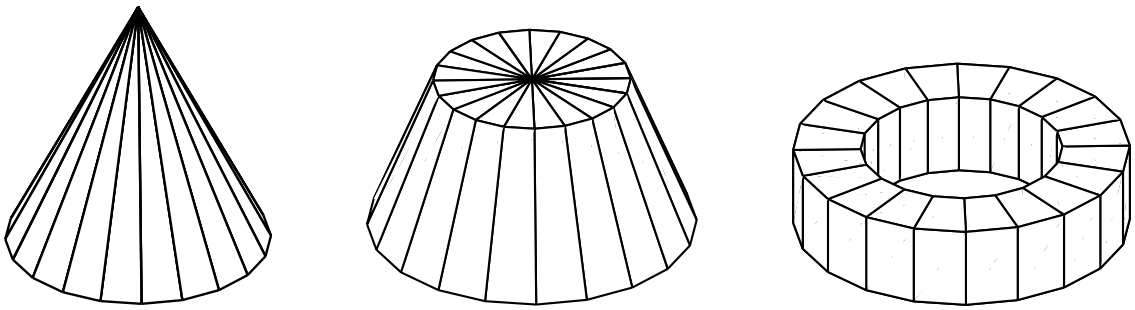


Abbildung 87: Variationen des `Cylinder`-Objekts: Kegel, Kegelstumpf, Ring.

ROTATE MATLABs Volumenobjekte sind am Koordinatensystem ausgerichtet, z ist die Rotationsachse, der Koordinatennullpunkt ist bei `sphere` der Mittelpunkt, bei `cylinder` das Zentrum der Basisfläche. Verschiebungen und Größenänderungen erreicht man dadurch, dass man die Koordinatensätze geeignet multipliziert oder Konstanten dazu addiert. Änderungen in der Orientierung – besonders beim `Cylinder`-Objekt interessant – lässt man am einfachsten von der Funktion `rotate` erledigen, die Graphikobjekte um beliebige Winkel im Raum drehen kann.

3.6.2 Weitere Formen

Eigene Volumenobjekte kann man auf der Basis von Polygonflächen, Dreiecken oder über einem regulären Gitter formulieren. Ein Aufbau aus Polygonen ist die vielseitigste, aber auch aufwendigste der drei Möglichkeiten; wenn anwendbar, führen reguläre Gitter meist am schnellsten und einfachsten zum fertigen Produkt.

FILL3, PATCH Ein Würfel soll als Beispiel für die Verwendung von Polygonflächen dienen. Sechs Flächen sind zu erstellen. Deren Koordinaten können beispielsweise durch die kubischen Symmetrieoperationen C4, C3 und Inversion generiert werden, das erledigt

```
r(1,:) = [1 1 1];
C4 = [0 1 0; -1 0 0; 0 0 1];
for n=2:4, r(n,:) = r(n-1,:)*C4; end;
C3 = [0 1 0; 0 0 1; 1 0 0];
for k=2:3, for n=1:4,
    r(n,:,k) = r(n,:,k-1)*C3;
end; end;
r(:,:,4:6) = -r(:,:,1:3); .
```

Gezeichnet wird dann mit der `fill3`-Anweisung:

```
h = fill3(squeeze(r(:,1,:)),squeeze(r(:,2,:)),...
```

```
squeeze(r(:,3,:)),squeeze(r(:,3,:))); .
```

Durch `squeeze` wird die singuläre Dimension aus den Datenfeldern entfernt, somit erhält `fill3` die benötigten zweidimensionalen Felder `x`, `y`, `z`, `c`, in denen jeweils eine Spalte für ein Polygon zuständig ist.

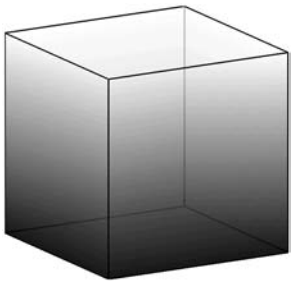


Abbildung 88: Aus Polygonflächen zusammengesetzter Würfel. Die Farbe ist proportional zu z, zusätzlich wurde leichte Transparenz eingestellt.

TRISURF, TRIMESH Ein einfaches Beispiel für Dreiecke ist ein Oktaeder, er wird für die Funktion `trisurf` definiert durch

```
r = [eye(3);-eye(3)];
t = [1 2 3; 1 3 5; 1 5 6; 1 6 2;...
     4 2 3; 4 3 5; 4 5 6; 4 6 2];
```

und dann gezeichnet mit

```
h = trisurf(t,r(:,1),r(:,2),r(:,3)); .
```

Das Feld `r` enthält die Koordinaten der 6 Eckpunkte, `t` ist die Liste der zu zeichnenden Dreiecke[55].

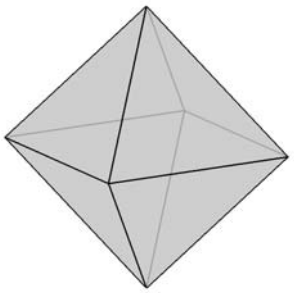


Abbildung 89: Oktaeder, mit `trisurf` aus Dreiecken aufgebaut.

SURF, MESH Raumflächen, die durch einen oder mehrere monotone Parameter bschrieben werden können, lassen sich über reguläre Gitter der entsprechenden Dimension definieren. Als Beispiel ein Torus, er wird durch 2 Laufparameter beschrieben, den Rotationswinkel φ und den Winkel θ, der die kreisförmige Querschnittsfläche charakterisiert, sowie

[55] Ein Oktaeder kann auch einfach als `Cylinder`-Objekt formuliert werden, `cylinder([0,0.5,0],4)` erledigt das.

durch 2 konstante Parameter, Rotationsradius R_0 und Radius der Querschnittsfläche r_0. Die Formulierung in MATLAB[56]:

```
phi = linspace(0,2*pi,N+1);
theta = linspace(0,2*pi,N+1);
[p,t] = meshgrid(phi,theta);
r = R0-r0.*cos(t);
x = r.*cos(p);
y = r.*sin(p);
z = sin(t).*r0; .
```

Abbildung 90 zeigt das Ergebnis als `mesh`.

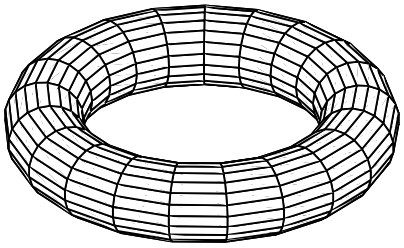


Abbildung 90: Torus mit $R_0 = 4$ und $r_0 = 1$ in `Mesh`-Darstellung.

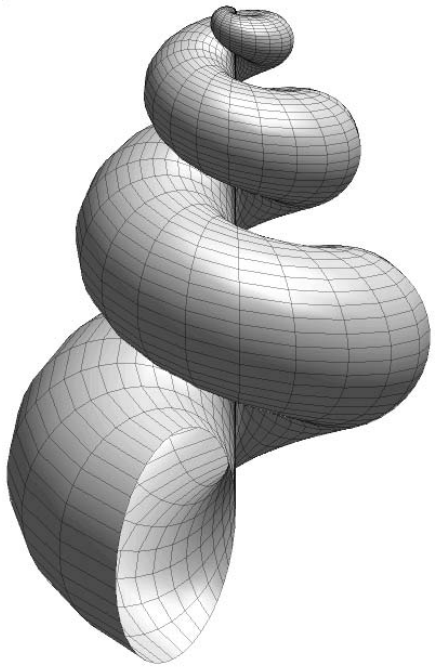
Aus den vorgefertigten Basisobjekten lassen sich dann leicht komplexere, zusammengesetzte Graphiken erstellen, ein Beispiel ist 'Black and White' (Abbildung 91):

```
[x,y,z] = torus(4,1.5,40);
t1 = surf(x,y,z);
hold on;
t2 = surf(x+4,y,z);
rotate(t2,[1 0 0],-90,[4 0 0]);
...
```

Abbildung 91: 'Black and White'.

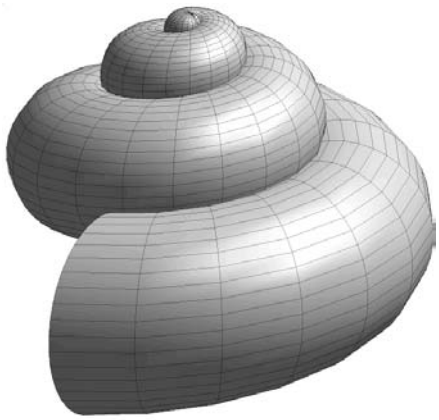
[56] Als rotationssymmetrischer Körper könnte ein Torus auch über ein – nachträglich zu modifizierendes – `Cylinder`-Objekt definiert werden.

Auch nicht rotationssymmetrische Raumflächen können oft durch relativ kurze Skripte realisiert werden. Mathematisch aber auch biologisch interessannte Beispiele dafür sind Konchoide (Schneckenhausformen), die als Torus mit winkelabhängigen Radien beschrieben werden können. In den Abbildungen 92 bis 94 sind drei solche typischen Konchoidgeometrien dargestellt (alle sind Vorbildern nachempfunden).



```
phi = linspace(0,7*pi,70);
theta = linspace(0,2*pi,40);
[p,t] = meshgrid(phi,theta);
R0 = p;  r0 = p;
r = R0-r0.*cos(t);
x = r.*cos(p);
y = r.*sin(p);
z = sin(t).*r0-r0.*r0/5;
h = surf(x,y,z);
set(h,'FaceColor',0.8*[1 1 1]);
set(h,'EdgeAlpha',0.3);
camlight, lighting gouraud;
axis equal, axis off;
```

Abbildung 92: Konchoide 1; oben zugehöriger MATLAB-Code.

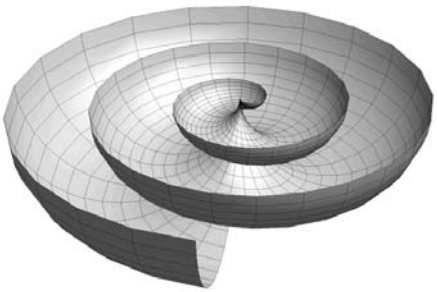


```
R0 = p.*p/20;  r0 = p/1.5;
...
y = -r.*sin(p);
z = sin(t).*r0-r0.*r0/6;
```

Abbildung 93: Konchoide 2; Code-Änderungen gegenüber Abbildung 92.

3.6.3 Material und Beleuchtung

Das Aussehen eines Volumenkörpers bzw. seiner Oberfläche wird durch verschiedene Objekteigenschaften sehr detailliert festgelegt. So werden Farbe und Transparenz der Flächen



```
R0 = p;  r0 = p;
...
z = sin(t).*r0;
z = z+0./(z<=eps);
```

Abbildung 94: Konchoide 3; Code-Änderungen gegenüber Abbildung 92.

und Kanten durch `FaceColor`, `FaceAlpha`, `CData`, `EdgeColor`, `EdgeAlpha` usw. bestimmt. Weitere Aspekte der Material- oder Oberflächenbeschaffenheit sind Beleuchtung und Reflektivität. Lichtart, -farbe und -position sind beliebig einstellbar, es können auch mehrere Beleuchtungsquellen gleichzeitig in einer Szene verwendet werden. Gesteuert wird das durch `light`, `lighting`, `lightangle` und die Objekteigenschaften von `light`.

Die Lichtreflexion von Oberflächen wird durch `AmbientStrength`, `DiffuseStrength` und `SpecularStrength` bestimmt, Eigenschaften, die individuell für Objekte konfigurierbar sind. Einen Überblick gibt Abbildung 95.

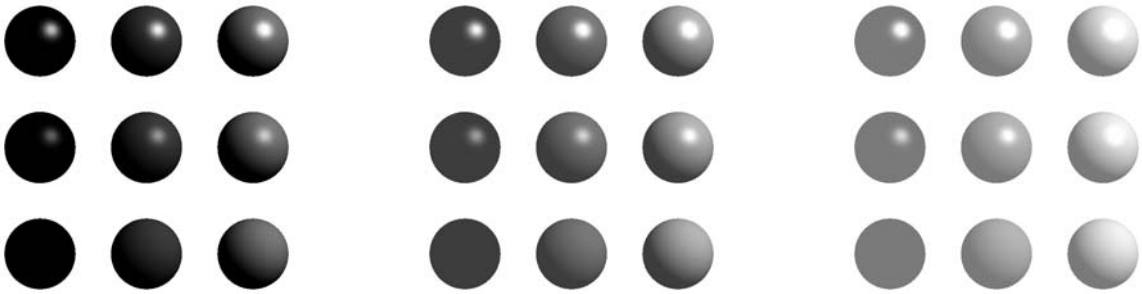


Abbildung 95: Reflexionseigenschaften. Linke Neunergruppe: `AmbientStrength`=0, Mitte: 0.3, rechts: 0.6. In den einzelnen Gruppen nimmt `DiffuseStrength` von links nach rechts zu (0, 0.3, 0.6), `SpecularStrength` von unten nach oben (0, 0.4, 0.8). Das beleuchtete Objekt ist jeweils die gleiche hellgraue Kugel.

Darüber hinaus kann mit dem `SpecularExponent` die räumliche Ausdehnung der direkten Reflexe eingestellt werden (typischer Wert: 10), `SpecularColorReflectance` schließlich stellt die Farbe des Reflexes ein (0: Objektfarbe ... 1: Lichtfarbe).

Statt für Objekte einzeln können mit `material` die Reflexionseigenschaften für alle Objekte im aktuellen Koordinatensystem auf einheitliche Werte gesetzt werden (⇒ `help material`).

3.6.4 Kristallstrukturen

Eine in der Festkörperphysik wichtige Anwendung für geometrische Anordnungen aus einfachen Volumenobjekten ist die Visualisierung von Kristallstrukturen. Für viele Strukturen reichen 2 Grundmotive aus, Atome und Verbindungslinien. Zur einfacheren formalen Handhabung definieren wir dafür zwei Basisfunktionen, `Atom`:

```
function h = Atom(pos,r,color)
if nargin<3,  color = 0.8*[1 1 1];  end;
if nargin<2,  r = 1;  end;
if nargin<1,  pos = [0 0 0];  end;
[x,y,z] = sphere;
h = surf(pos(1)+r*x,pos(2)+r*y,pos(3)+r*z);
set(h,'EdgeAlpha',0,'FaceColor',color);
```

und `Connect`:

```
function h = Connect(a,b,r,color)
if nargin<4,  color = 0.9*[1 1 1];  end;
if nargin<3,  r = 0.1;  end;
if nargin<2,  return;  end;
d = b-a;
l = sqrt(sum(d.*d));
if l==0,  return;  end;
[x,y,z] = cylinder([r r]);
z = l*z;
h=surf(a(1)+x,a(2)+y,a(3)+z);
set(h,'EdgeAlpha',0,'FaceColor',color);
phirot = 180/pi*acos(d(3)/l);
ax = [-d(2) d(1) 0];
if (sum(ax.*ax)==0),  ax = [1 0 0];  end;
rotate(h,ax,phirot,a); .
```

Eine kubische Perowskitstruktur ABO_3, in der Materialien wie Bariumtitanat ($BaTiO_3$), Strontiumtitanat ($SrTiO_3$) oder Kaliumtantalat ($KTaO_3$) kristallisieren, wird aus diesen Grundfunktionen realisiert durch das Fragment

```
A = [0 0 0; 2 0 0; 0 2 0; 0 0 2;...
     2 2 2; 0 2 2; 2 0 2; 2 2 0];
B = [1 1 1];
O = [1 1 0; 1 1 2; 0 1 1; 2 1 1; 1 0 1; 1 2 1];
nA = size(A,1);
nB = size(B,1);
```

```
nO = size(O,1);
figure; hold on;
for n=1:nA, Atom(A(n,:),0.25,0.6*[1 1 1]); end;
for n=1:nB, Atom(B(n,:),0.15,0.3*[1 1 1]); end;
for n=1:nO, Atom(O(n,:),0.2,0.9*[1 1 1]); end;
D=inline('sqrt(dot(a-b,a-b))');
for n=1:nA-1,  for m=n:nA,
  if D(A(n,:),A(m,:))<2.1,
      Connect(A(n,:),A(m,:),0.015);
  end;
end;  end;
for n=1:nO-1,  for m=n:nO,
  if D(O(n,:),O(m,:))<1.5,
      Connect(O(n,:),O(m,:),0.025);
  end;
end; end; .
```

`A`, `B` und `O` definieren die Koordinaten der einzelnen Atomsorten, diese werden in unterschiedlicher Größe und Farbe[57] gezeichnet, schließlich werden der A-Kubus und der O-Oktaeder durch Verbindungszylinder markiert (Abbildung 96).

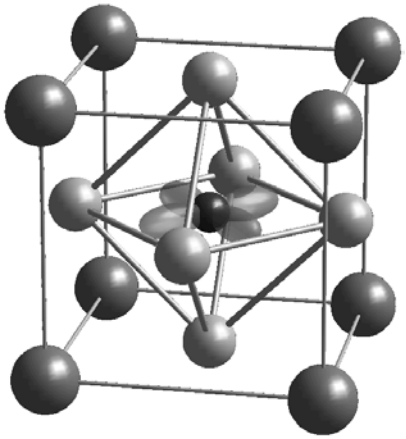


Abbildung 96: Perowskit-Struktur ABO_3. Die A-Atome sitzen an den Würfelecken, ein B-Atom im Zentrum, die O-Atome auf den Flächenmitten.

CAMERA Bei 'Szenen' ist es naheliegend, die Projektion durch Kameraeigenschaften zu definieren. MATLAB bietet dazu eine Reihe von Funktionen, die mit `cam...` beginnen. Darüber lassen sich Abbildungseigenschaften wie Bildausschnitt oder Perspektive besser steuern als mit `view` und `axis`. Die Perowskitstruktur wurde mit

```
axis equal, axis off, light, lighting gouraud;
```

[57] Bei farbigen Darstellungen lassen sich einzelne Atomsorten durch ihre Einfärbung noch besser unterscheiden als in der im Skript durchgängig verwendeten Graustufendarstellung.

```
camtarget([0.95 1 0.95]);
campos([5 -20 8]);
camva(7.5);
camproj('perspective');
```

in Szene gesetzt, `camtarget` legt den Bildmittelpunkt fest, `campos` die Kameraposition, `camva` den gewünschten Winkelausschnitt, `camproj` die Projektionsart.

STRUKTURDATEN Die Atomkoordinaten einfacher Kristallstrukturen wie Perowskit können meist direkt angegeben werden. Wird die Struktur komplizierter, ist dies in der Regel nicht mehr sinnvoll machbar. Strukturdaten werden dann entweder aus den Koordinaten der Basisatome des Gitters durch die Symmetrieoperationen der Raumgruppe berechnet[58] oder aus Tabellen entnommen. Solche Tabellen mit Strukturdaten gibt es in unterschiedlichen Formaten aus unterschiedlichen Quellen[59].

Ein einfach zu lesendes und zu erstellendes Format ist das von .MOL-Dateien, die im wesentlichen zwei Listen – eine für die Atome, eine für die Verbindungen – enthalten. Eine Funktion `molread` zum Lesen von .MOL-Dateien kann in MATLAB etwa so realisiert werden:

```
function [r,element,lnk] = molread(fname);
fid = fopen(fname);
for n=1:4, line=fgetl(fid); end;
npos = sscanf(line(1:3),'%d',1);
nlnk = sscanf(line(4:6),'%d',1);
for n=1:npos,
  line = fgetl(fid);
  r(n,:) = sscanf(line,'%f',3)';
  element(n,:) = sscanf(line,'%*f%*f%*f%*[ ]%2c',1)';
end;
for n=1:nlnk,
  line = fgetl(fid);
  lnk(n,1) = sscanf(line(1:3),'%d',1);
  lnk(n,2) = sscanf(line(4:6),'%d',1);
end;
fclose(fid); .
```

Die Datei wird damit zeilenweise gelesen und interpretiert. Zunächst bis zur 4. Dateizeile, dort stehen die Längen der beiden Listen – Zahl der Atompositionen `npos`, Zahl

[58] Die Symmetrieoperationen werden durch 4x4-Matrizen definiert, die Rotation und Translation enthalten, Koordinaten als 4x1-Vektoren mit den 3 Ortskoordinaten und einer 1 an der 4. Position. Ausführung der Symmetrieoperation durch Matrixmultiplikation.

[59] Beipieldaten für Moleküle oder Kristalle können von spezialisierten Strukturdarstellungsprogrammen wie RasMol (in Linux-Distributionen), WebLab (`www.msi.com`) 'entliehen' werden.

der Verbindungen `nlnk`. Es wird dann die Tabelle der Atompositionen, danach die der Verbindungslinien gelesen (eine Zeile pro Datensatz).

Mit der beschriebenen Lesefunktion vereinfacht sich die Kristalldarstellung in MATLAB, für eine Lithiumniobatstruktur, die in einer Datei `ln2.mol` definiert ist, könnte das so aussehen:

```
[r,element,lnk] = molread('ln2.mol');
figure; hold on;
for n=1:size(r,1),
  if element(n,1)=='O', ra=0.8; c=0.9; end;
  if element(n,1)=='N', ra=0.7; c=0.5; end;
  if element(n,1)=='L', ra=0.6; c=0.2; end;
  atom(r(n,:),ra,c*[1 1 1]);
end;
for n=1:size(lnk,1),
  connect(r(lnk(n,1),:),r(lnk(n,2),:),0.1);
end; .
```

Abbildung 97 zeigt das Ergebnis.

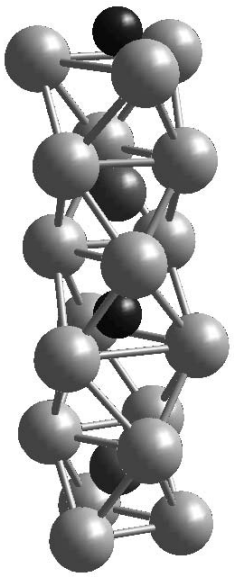


Abbildung 97: Lithiumniobatstruktur (ferroelektrische Phase).

STEREO Stereoskopische Bilder sind gerade für Kristallstrukturdarstellungen sehr hilfreich, da sie räumliche Zusammenhänge und Zuordnungen deutlicher aufzeigen. Die einfachste Technik für Stereobilder ist es, zwei getrennte Bilder mit einer für das jeweilige Auge eingestellten Blickrichtung zu produzieren, in MATLAB zum Beispiel durch zwei Kamerapositionen im Augenabstand. Für die Betrachtung gibt es dann mehrere Möglichkeiten: spezielle Betrachtungsoptiken, entspanntes Schielen (das Bild für das linke Auge liegt

links), und forciertes Schielen (das Bild für das linke Auge liegt rechts). Das erste Beispiel ist für entspanntes Schielen gedacht, die Stereoansicht eines C_{60}-Moleküls (Buckyball).

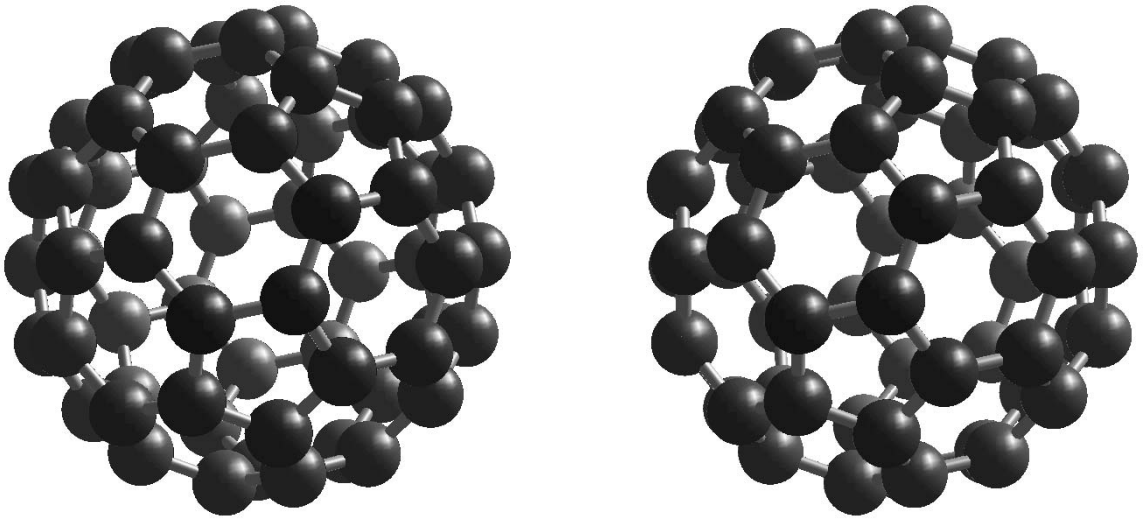


Abbildung 98: Stereoansicht eines C_{60}-Moleküls (mit entspannten Augen zu betrachten).

Das zweite Beispiel, für 'forciertes' Schielen, ist die schon bekannte Perowskitstruktur.

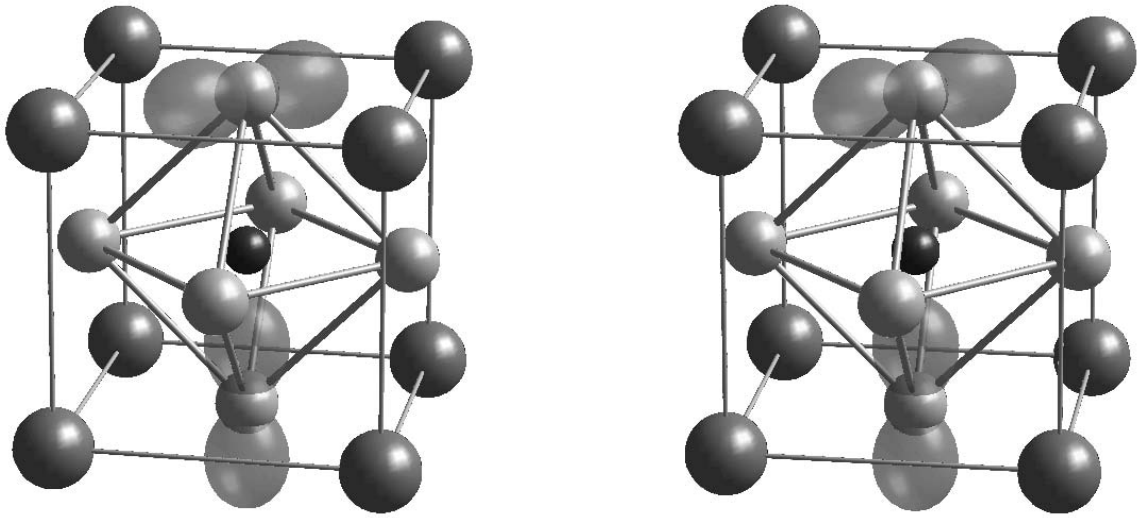


Abbildung 99: Stereoansicht einer Perowskitstruktur (mit 'gekreuzten' Augen bzw. Blickrichtungen zu betrachten).

3.6.5 Texturen, Muster

MATLAB kann Flächen mit Mustern oder Bildern versehen, die als zweidimensionale Felder vorgegeben sind. Dabei brauchen die Feldgrößen zwischen Bild und Oberfläche nicht überein zu stimmen, MATLAB interpoliert geeignet. Auf diese Weise hat man beispielsweise die Möglichkeit, geographische Bilder realistisch auf Kugelflächen zu verarbeiten[60] oder Volumenobjekte mit gemusterten Oberflächen herzustellen.

Drei einfache Beispiele sollen die Vorgehensweise illustrieren. Beim 'Golfball' wird das Bild eines Punktmusters auf eine Kugel projiziert (Abbildung 100). Das (rechteckige) Bild des Punktmusters wird mit der Funktion `golftex` ebenfalls von MATLAB berechnet.

```
zz = golftex;
[x,y,z] = sphere;
h = surf(x,y,z);
set(h,'CData',zz,'FaceColor','texturemap');
set(h,'EdgeColor','none');
```

Abbildung 100: 'Golfball'. Rechts das zugehörige Code-Fragment.

Im Beispiel 'Limited Horizon' wird ein GIF-Bild als Muster verwendet, das Bild und die zugehörige Farbtabelle werden mit `imread` gelesen. Wegen der bei Bildern von der üblichen Koordinatenanordnung abweichenden Orientierung wird die Bildmatrix mit `flipud` auf den Kopf gestellt.

Im Beispiel 'Convex and Concave' werden Bilder auf Halbzylinder aufgebracht; der nicht benötigte Bereich wird mit `NaN` weggeschnitten. Das Bild muss dennoch der ganzen Zylinderfläche entsprechen, daher die Ergänzung der Bildmatrix mit `ones`.

3.6.6 Skalardaten in 3D

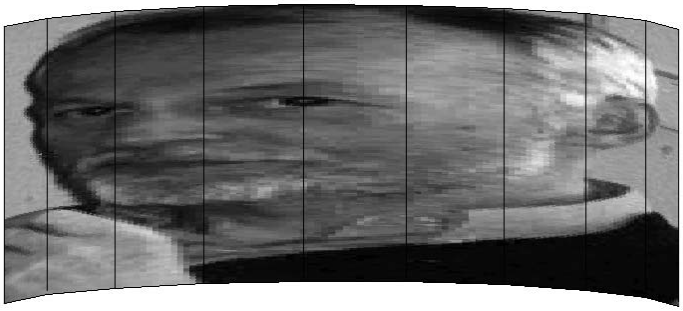
Funktionen oder Daten, die von drei oder mehr Parametern (z. B. Ortskoordinaten) abhängen, sind grundsätzlich nicht mehr als Raumflächen darzustellen. Man kann sie durch geeignete Isoflächen charakterisieren oder ihre Werteverteilung auf sinnvoll gewählten Schnittflächen durch Farbe oder Konturen darstellen.

SLICE Zur Darstellung von Volumendaten mithilfe von Schnittflächen bietet MATLAB die Funktion `slice`, mit der die Volumendaten $v(x, y, z)$ durch Ebenen senkrecht zu den Koordinatenachsen, durch beliebig im Raum liegende Ebenen oder durch allgemei-

[60] MATLAB enthält zwei Datensätze für geographische Darstellungen, `topo` und `earth`, mit denen man Beispiele formulieren kann.


```
[c,map] = imread('westerb.gif');
[x,y,z] = sphere;
h = surf(x,y,z);
set(h,'CData',flipud(c));
set(h,'FaceColor','texturemap');
colormap(map);
```

Abbildung 101: 'Limited Horizon'.



```
[c,map] = imread('klausb.gif');
[x,y,z] = cylinder([1],21);
z(:,11:21) = NaN;
h = surf(x,y,0.8*z);
c = [c(1:200,:),ones(200,150)];
set(h,'CData',flipud(c));
set(h,'FaceColor','texturemap');
colormap(map);
```

Abbildung 102: 'Convex and Concave'.

ne Flächen im Raum geschnitten werden können. Die Schnittflächen mit den jeweils zugehörigen Daten (z. B. $v(x_0, y, z)$ bei einem Schnitt senkrecht zur x-Achse) werden gezeichnet. Voreingestellt ist dabei `shading faceted`, Gitternetzdarstellung mit eingefärbten Flächenelementen.

CONTOURSLICE Isolinien in den Schnittebenen werden mit `contourslice` angeordnet. In diesem Fall können – wie bei der `contour`-Funktion – Anzahl oder Funktionswerte als zusätzliches Argument angegeben werden. Ein Beispiel für die Verwendung der beiden Funktionen `slice` und `contourslice` zeigt Abbildung 103, dargestellt wird die Funktion (Dichteverteilung von p_x-Elektronen)

$$v = x^2 \exp(-x^2 - y^2 - z^2)\,. \tag{3.13}$$

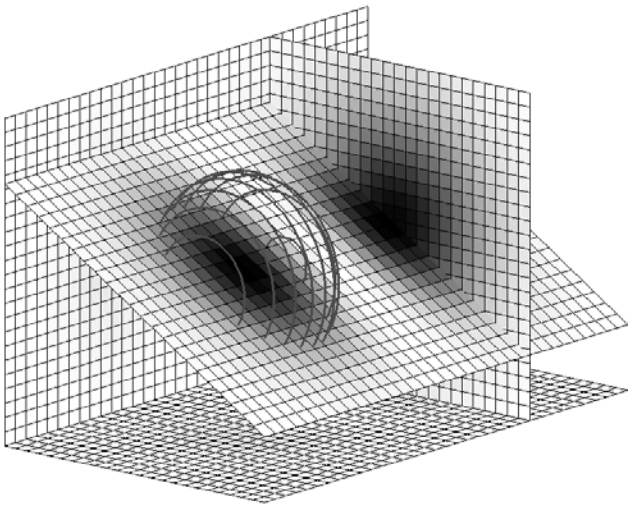


Abbildung 103: `Slice` und `Contourslice`: Dichteverteilung von p_x-Elektronen.

Im zugehörigen MATLAB-Skript wird zunächst v berechnet

```
xv = linspace(-2.5,2.5,35);
yv = linspace(-1.5,1.5,25);
[x,y,z] = meshgrid(xv,yv,yv);
v = x.*x.*exp(-x.*x-y.*y-z.*z); ,
```

dann werden Schnittebenen senkrecht zu den Koordinatenachsen eingezogen ($x = 1.15$, $y = 1.5$, $z = -1.5$)

```
hs = slice(x,y,z,v,[1.15],[1.5],[-1.5]); .
```

Es wird eine schiefliegende Schnittebene `xp`, `yp`, `zp` als Parameter für das nächste `slice` definiert

```
[xp,yp] = meshgrid(xv,yv,yv);
zp = 0.6*yp;
hold on;
hp = slice(x,y,z,v,xp,yp,zp); .
```

Die Schnittebenen für die Isolinien liegen äquidistant in dem durch `linspace(...)` aufgespannten Bereich senkrecht zur x-Achse, es wird jeweils eine Isolinie pro Schnittebene

beim Funktionswert 0.15 gezeichnet (der entsprechende Parameter muss ein Vektor sein, daher die Doppelangabe):

```
hc = contourslice(x,y,z,v,linspace(-2,0,15),[],[],[0.15 0.15]); .
```

Mit

```
set (hc,'EdgeColor',0.3*[1 1 1],'LineW',2);
colormap(flipud(colormap(gray)));
```

werden die verwendeten Grautöne eingestellt.

ISOSURFACE Isoflächen – Raumflächen, auf denen der Funktionswert v eine konstante Größe hat – werden mit `isosurface` gezeichnet. Abbildung 104 zeigt eine Kombination aus Schnittebenen und Isoflächen. Zunächst werden Schnittebenen durch Bereiche mit großem Funktionswert definiert und dann zur Seite oder nach unten verschoben:

```
hs = slice(x,y,z,v,[1],[1],[0]);
set(hs(1),'XData',get(hs(1),'XData')+1.4);
set(hs(2),'YData',get(hs(2),'YData')+1.4);
set(hs(3),'ZData',get(hs(3),'ZData')-2); .
```

Dadurch wird ein projektionsartiger Effekt erzielt.

Die Isoflächen werden mit

```
p = patch(isosurface(x,y,z,v,0.04));
```

für den Funktionswert 0.04 gezeichnet.

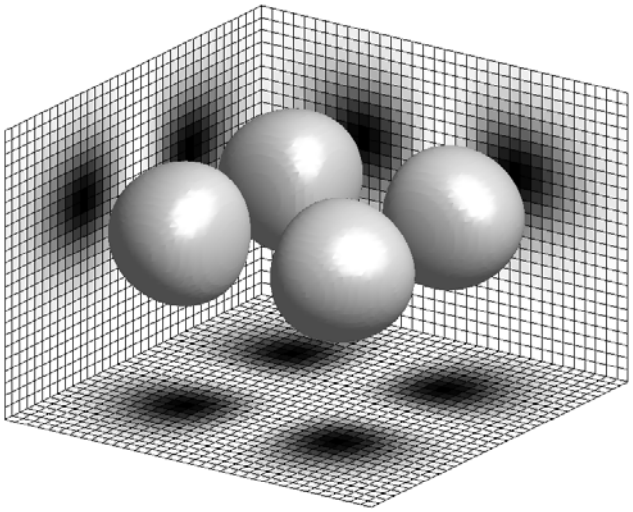


Abbildung 104: `Isosurface` *und* `Slice` *kombiniert: Dichteverteilung von d_{xy}-Elektronen.*

3.6.7 Vektordaten in 3D

Durch dreidimensionale Vektordaten werden Felder aller Art, aber auch Strömungen, Windgeschwindigkeiten u. ä. beschrieben. MATLAB enthält mehrere Funktionen, die aus den Vektordaten Stromlinien oder Feldlinien berechnen und zeichnen (⇒ `help streamline`), sowie Funktionen, die Richtungspfeile oder -kegel im Raum zeichnen.

CONEPLOT Die Richtungskegel für die Strömungsvisualisierung in Abbildung 105 werden im wesentlichen durch die Anweisung

```
hcone = coneplot(x,y,z,vx,vy,vz,cx,cy,cz,8);
```

realisiert. Darin sind als Parameter das zugrunde liegende Koordinatengitter `x`, `y`, `z`, das zugehörige Vektorfeld `vx`, `vy`, `vz` und die Koordinaten `cx`, `cy`, `cz` für die gewünschten Richtungskegel anzugeben, sowie optional ein Vergrößerungsfaktor. Das Aussehen der gezeichneten Kegelflächen wird mit

```
set(hcone,'Edgecolor','none','FaceColor',0.2*[1 1 1]);
```

eingestellt.

Ähnlich wie `coneplot` arbeitet `quiver3`, das Richtungspfeile im Dreidimensionalen zeichnet.

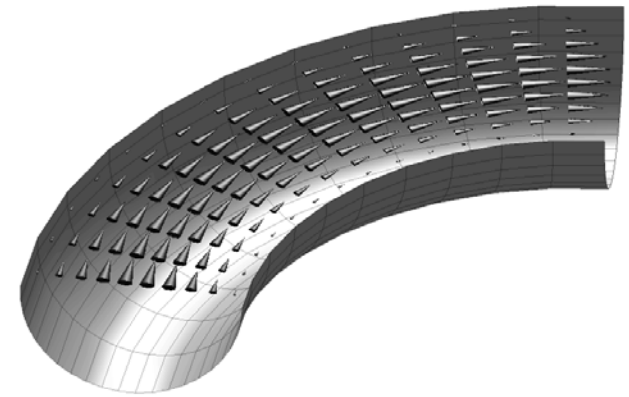


Abbildung 105: `Coneplot`: Visualisierung von Strömungen oder Feldern.

3.7 Animation, Filme

Die Zeit als zusätzlicher Parameter bei graphischen Darstellungen wird naheliegenderweise dann eingesetzt, wenn man zeitabhängige Inhalte darstellen will, Bewegungen, zeitliche Entwicklungen o. ä. Darüber hinaus kann man irgendeinen der Funktionsparameter auf die Zeit abbilden, d. h. zeitlich verändern, und dadurch komplexe Zusammenhänge in eine Folge von einfacheren Einzelgraphiken zerlegen. Ein Beispiel ist das zeitliche 'Durchfahren' von Funktionen oder Daten im Raum entlang einer der Ortskoordinaten oder auch auf

einer beliebigen Ortskurve, um durch unterschiedliche Teilansichten einen Überblick über komplexe Daten zu vermitteln.

3.7.1 Bildfolgen

Für eine Abfolge von Bildern erstellt man die komplette Graphik oder Teile davon innerhalb einer Schleife. So lässt sich im Beispiel der Abbildung 103 eine zeitlich durchlaufende Schnittebene mit

```
hold on;
for x0=-2:0.05:2,
    hs = slice(x,y,z,v,[x0],[],[]);
    drawnow;
    delete(hs);
end;
```

realisieren. Aus Geschwindigkeitsgründen wird jeweils nur diese eine Ebene, nicht das ganze Bild, gezeichnet. Der restliche Bildinhalt wird vor der Schleife mit `hold on` eingefroren, der in der Scheife gezeichnete Teil wird mit `delete` jedes Mal gelöscht, um nicht zu akkumulieren. Mit `drawnow` erreicht man, dass MATLAB genau an dieser Stelle graphisch tätig wird und das nicht bis zum Skriptende aufschiebt.

3.7.2 Betrachtergesteuerte Inhalte

Statt eine Bildfolge mit festem zeitlichen Ablauf vorzugeben, kann man es auch dem Betrachter übertragen, Werte für variable Parameter einzustellen. Damit ist es möglich, interaktiv die graphische Darstellung zu beeinflussen und die Auswirkung von Parameteränderungen zu studieren.

UICONTROL Zur interaktiven Änderung von Parametern werden Bedienelemente – *User Interface Controls* – ins MATLAB-Bildfenster integriert. Solche Bedienelemente werden mit `uicontrol` angefordert, verschiedene Arten – Listen, Texteingabe, Schieber etc. – sind verfügbar. Ein Schieber (*Slider*) für obiges Beispiel würde mit

```
global x y z v hu
hu = uicontrol('Style','Slider');
set(hu,'Position',[20 20 200 30]);
set(hu,'Min',-2,'Max',2,'Value',-1);
set(hu,'CallBack','pxuicb');
```

ins Bild eingebaut. Position im Bild (hier in Pixeln), Minimal-, Maximal- und aktueller Wert werden definiert. Außerdem das Wichtigste, eine *Callback*-Funktion, die als Reaktion

auf eine Bedienung aufgerufen wird. Diese hier zeichnet die variable Schnittebene am über den Schieber eingestellten Ort ein:

```
function pxuicb
global x y z v hu
persistent hp
x0 = get(hu,'Value');
if exist('hp'), delete(hp); end;
hp = slice(x,y,z,v,[x0],[],[]); .
```

Noch kürzer wird's, wenn – wie im Beispielskript `perotate.m` – nur der Betrachtungswinkel über zwei Schieber verändert wird, die dieselbe *Callback*-Funktion verwenden:

```
function perocb
global haz hel
view(get(haz,'Value'),get(hel,'Value')); .
```

Die *Callback*-Funktion muss in einer eigenen m-Datei gespeichert sein, da das Hauptskript zum Zeitpunkt des Aufrufs in aller Regel beendet ist.

3.7.3 Filme

Filme sind gespeicherte Bildfolgen. MATLAB verwendet für die Speicherung ein eigenes spezielles Format, hat aber seit Release 12 auch eine Funktion für die Übersetzung ins AVI-Format. Dieses kann dann mit den üblichen Wiedergabeprogrammen der Betriebssysteme abgespielt werden. Diese Wiedergabeprogramme bieten erweiterte Möglichkeiten, beipielsweise Endloswiedergabe (bei periodischen Vorgängen) oder benutzergesteuerte Einzelbilddarstellung (der Benutzer kann den zeitlich veränderlichen Parameter interaktiv einstellen).

GETFRAME, MOVIE Einzelne Filmbilder werden mit

```
M(i) = getframe;
```

unter dem Laufindex `i` im Film `M` abgelegt. In der Regel wird man die Einzelbilder innerhalb einer Programmschleife generieren und speichern, `i` wird in der Schleife inkrementiert. Der so entstandene Film wird in MATLAB mit

```
movie(M,n)
```

abgespielt, der optionale Parameter `n` gibt die Anzahl der Wiederholungen an.

MOVIE2AVI Mit der Funktion `movie2avi` wird das MATLAB-eigene Movie-Format in das verbreitete Standardformat AVI umgesetzt, als Parameter können u. a. die Filmgeschwindigkeit (`'FPS'`) und die Komprimierungsqualität (`'Quality'`, `0...100`) angegeben

werden. Umfangreiche Filme lassen sich bisweilen MATLAB-intern nicht mehr gut handhaben. Als Alternative bietet sich dann an, die Einzelbilder als Folge von Pixeldateien zu speichern und mit einem spezialisierten Programm zusammen zu binden. Innerhalb einer Schleife mit der Laufvariablen `k` werden die Einzelbilder etwa mit

```
fname = sprintf('bild%03d.bmp',k);
print('-dbmp','-r80', fname);
```

als BMP-Dateien mit laufender Nummerierung ausgegeben.

Die folgenden Abbildungen sind Einzelbilder aus den Filmbeispielen.

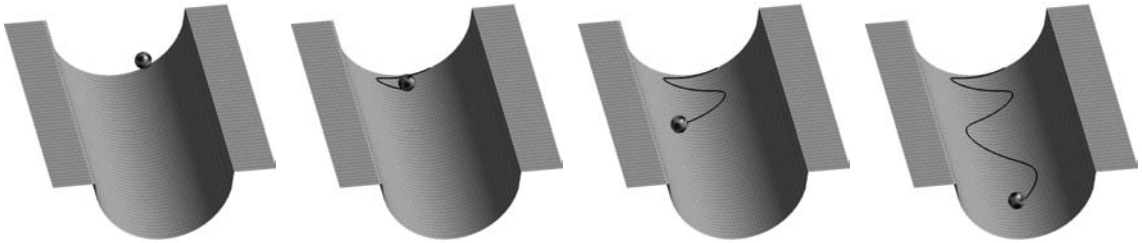


Abbildung 106: 'Halfpipe.avi', rollende Kugel. Die 4 Teilbilder sind zeitlich äquidistant.

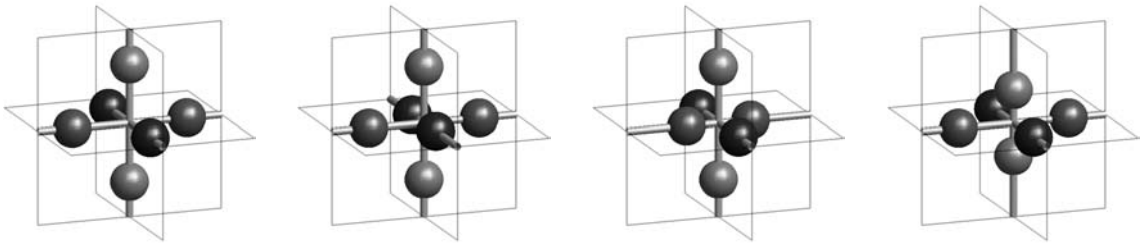


Abbildung 107: 'Modes.avi', Schwingungsmoden: S, P_x, P_y, P_z.

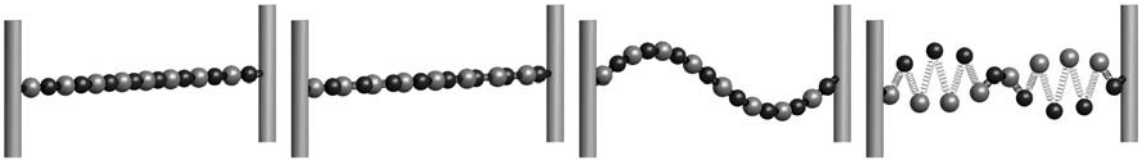


Abbildung 108: 'ChainLA.avi', 'ChainLO.avi', 'ChainTA.avi', 'ChainTO.avi', Schwingungsformen der linearen Kette.

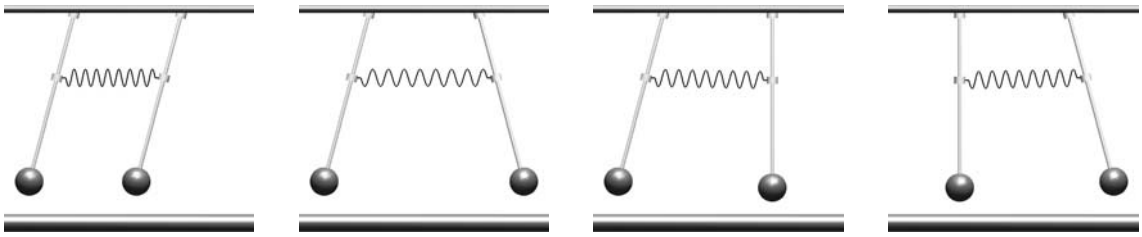


Abbildung 109: 'CouplAsy.avi', 'CouplSym.avi', 'CouplBeat.avi', gekoppelte Pendel.

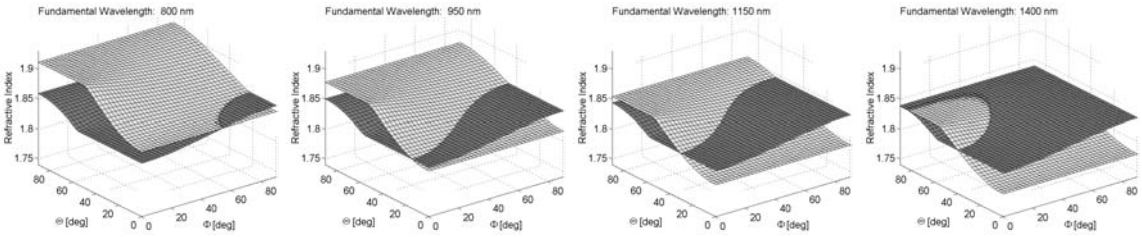


Abbildung 110: 'PMA.avi', Phasenanpassungsbedingung bei der Frequenzverdopplung in einem optisch zweiachsigen Kristall (Bleiformiat).

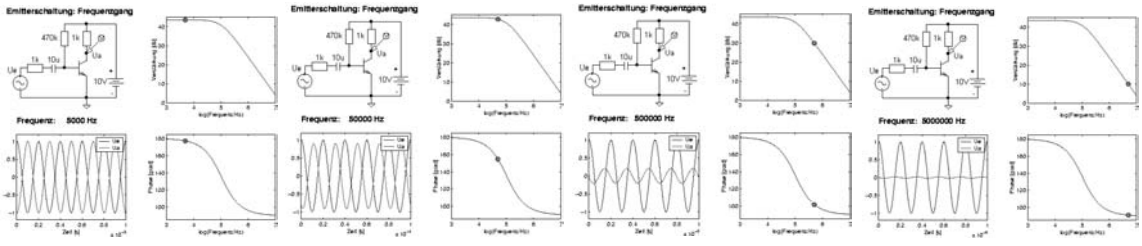


Abbildung 111: 'Ampl.avi', Frequenzgang eines Verstärkers.

Literatur und Links

Die Literaturhinweise sind – insbesondere die zitierten Lehrbücher betreffend – exemplarisch, willkürlich und zufällig, eine Vollständigkeit ist weder möglich noch angestrebt.

[1] Adobe Systems Inc. *PostScript Language Tutorial and Cookbook*. Addison-Wesley Pubishing Company, 1986. ISBN 0-201-10179-3.

[2] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley Pubishing Company, 1990. ISBN 0-201-37922-8.

[3] http://www.geocities.com/SiliconValley/5682/postscript.html.

[4] ftp://ftp.irisa.fr/pub/gnuplot/.

[5] http://www.ifor.math.ethz.ch/~finschi/Papers/LevMar.html.

[6] http://www.ucc.ie/gnuplot/gnuplot-faq.html.

[7] http://www.physik.uni-osnabrueck.de/kbetzler/gw/gpintrorz.pdf.

[8] http://www.netlib.org/linpack.

[9] http://www.netlib.org/eispack.

[10] http://www.netlib.org/lapack.

[11] Bei einer Standard-Installation:
<matlabroot>/help/pdf_doc/matlab/graphg.pdf,
ab Release 12 auf der *Documentation CD*.

[12] Bei einer Standard-Installation:
<matlabroot>/help/pdf_doc/matlab/ref/refbook2.pdf,
ab Release 12 auf der *Documentation CD*.

[13] http://www.mathworks.com/support.

[14] Bei einer Standard-Installation:
<matlabroot>/help/pdf_doc/matlab/ref/refbook.pdf,
ab Release 12 auf der *Documentation CD*.

[15] Näheres zum Runge-Kutta-Verfahren in Skripten (Hertel), Büchern oder Links zur Numerik (z. B. www.nr.com).