Applikationssoftware und Programmierung

Ass.-Prof.Dipl.-Ing.Dr. Winfried Kernbichler [1]
Institut für Theoretische Physik
Technische Universität Graz
Petersgasse 16, A-8010 Graz, Austria

23. Juni 2004

[1]Tel.: +43(316)873-8182; Fax.: +43(316)873-8678; e-mail: winfried.kernbichler@tugraz.at

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Allgemeines

Die Verwendung von Computern wurde, wie in vielen Bereichen des Lebens, auch in der Physik zu einem zentralen Bestandteil sowohl der Ausbildung als auch der Forschung. Die meisten Forschungsbereiche wären heute ohne die Verwendung von Computern und entsprechender Software gar nicht mehr denkbar. Das gilt sowohl für Experimente, deren Steuerung und Auswertung, als auch für die theoretische Behandlung von Problemen bzw. die numerische Simulation von Experimenten.

Die Lehrveranstaltung **Applikationssoftware und Programmierung** wurde im Studienplan der Studienrichtung **Technische Physik** daher bewußt an den Anfang des Studiums gestellt. Die Studierenden sollen dabei mit folgenden Bereichen konfrontiert werden:

- Verwendung von Computern, wie sie im Bereich der Physik üblich ist.
- Kennenlernen der Computerinfrastruktur für Studierende im Bereich der TU-Graz und speziell im Bereich der Physik.
- Kennenlernen und Verwenden von Programmpaketen (Applikationen), die für das weitere Studium nützlich sind (Auswertung und Darstellung von Messungen; numerische Berechnungen; Visualisierung; Präsentation und Dokumentation)
- Informationsbeschaffung aus dem World Wide Web, aus lokalen Dokumentationen oder von ihren Kollegen.
- Grundzüge des Programmierens.

Die Studierenden sollen daher von Anfang an die Möglichkeit erhalten, das für sie bereitgestellte System in vielen Bereichen ihres Studiums zu verwenden. Außerdem

sollen sie auf eine Fülle aufbauender Lehrveranstaltungen bestmöglich vorbereitet sein.

Folgende Lehrveranstaltungen sind stark mit der Benutzung von Computern verbunden:

- Numerische Methoden in der Physik
- Computersimulationen
- Numerische Behandlung von Vielteilchenproblemen
- Computersimulation und Vielteilchenphysik (1 und 2)
- Computersimulation in der Festkörperphysik
- Physik und Simulation des Strahlungstransports
- Applikationssoftware für Fortgeschrittene
- Computermeßtechnik
- Symbolisches Rechnen
- Programmieren in C
- Programmieren in FORTRAN
- Viele Praktika (Experiment und Theorie)
- Viele Übungen

Die Lehrveranstaltung ist eine Chance, die Möglichkeiten, Hilfen aber auch Grenzen kennenzulernen die Computer in der heutigen Zeit im Bereich der Physikausbildung und der Forschung bieten. Sie dient mehr einer Vermittlung von Fertigkeiten zur Problemlösung als einer Vermittlung von festgeschriebenen Fakten. Damit soll sie zur erfolgreichen Anwendung von Computersoftware während des Physikstudiums hinführen.

Die Lehrveranstaltung beinhaltet nicht:

- Eine allgemeine Einführung in die EDV
- Eine Erklärung der Funktionsweise von Computern
- Konzepte von Betriebssystemen
- Erklärung von Basissoftware (Office Packete, WEB-Browser, ...)

Diese Bereiche werden entweder in ihrer elementaren Form vorausgesetzt oder sind nicht von so großer Wichtigkeit in unserem Umfeld. Fragen dazu an mich oder an Ihre Kollegen sind aber natürlich jederzeit willkommen.

1.2 Organisation der Lehrveranstaltung

Die Lehrveranstaltung gliedert sich in **Vorlesung** und in **Übung** mit praktischen Arbeiten am Computer. Eine getrennte Teilnahme bzw. eine getrennte Prüfung macht keinen Sinn, da jeder Teil für sich genommen etwas isoliert dastehen würde. In der Vorlesung werden sowohl die Grundlagen für die jeweilige Übung vermittelt, als auch die Übung an sich vorgestellt. Damit soll die Bewältigung der Übungsbeispiele erleichtert werden.

1.2.1 Ziel

Die verwendete Programmiersprache ist MATLAB. Das Ziel der Lehrveranstaltung ist die Vermittlung der Grundlagen des Programmierens unter Verwendung von MATLAB. Am Ende der Lehrveranstaltung sollten Sie in der Lage sein, für die Physik relevante Probleme eigenständig zu lösen. Dazu gehören:

- Umsetzen mathematischer Formeln in Computercode
- Auswerten von Messdaten (Ausgleichskurven, Fitten)
- Visualisieren von Ergebnissen
- Erstellen von Programmen
- Verstehen von Programm- und Datenstrukturen
- Verstehen von vektor- und matrixorientierter Programmierung
- Grundlagen von MATLAB und Informationsbeschaffung aus dem englischsprachigen MATLAB-internen Hilfesystem

1.2.2 Anmeldung

Für die Teilnahme an der Vorlesung ist nur das Eintragen in der offiziellen Teilnehmerliste am TUG Online nötig.

Für die Teilnahme an den Übungen sind jedoch folgende Schritte unbedingt notwendig: www.itp.tu-graz.ac.at

- Eintragen in die offizielle Teilnehmerliste (TUG Online).
- Anmeldung für einen Computeraccount auf den Bereichsrechnern für die Studierenden im Bereich "Technische Physik" auf der WEB-Seite des Instituts für Theoretische Physik www.itp.tu-graz.ac.at. Dies ist ein von den Subzentren komplett getrenntes System, bei dem die dortige Kombination aus Benutzername und Passwort nicht funktioniert.

- Im Rahmen der ersten Lehrveranstaltung werden wir daher im Computerraum Physik einige Anmelderechner zur Verfügung stellen. Die Passwörter werden sofort ausgeteilt.

1.2.3 Übung

Die Übungen werden im Computerraum Physik abgehalten (siehe 1.3.1). Dieser Raum liegt direkt neben dem Hörsaal P2 im Physikgebäude.

Derzeit sind drei Gruppen mit jeweils maximal 15 Teilnehmern vorgesehen, da wir von ungefähr 45 Teilnehmern ausgehen. Dabei hat jeder Teilnehmer einen eigenen Computerarbeitsplatz. Bei starker Unter- bzw. Überschreitung dieser Zahl müssen notwendige Maßnahmen diskutiert werden.

Name	Tag	von	bis	Beginn	Ort	Leiter
A	Montag	16:15	17:45	1. März 2004	CR Physik	Kernbichler
B	Montag	18:00	19:30	1. März 2004	CR Physik	Kernbichler
C	Mittwoch	08:15	09:45	3. März 2004	CR Physik	Kernbichler

Übungen haben immanenten Prüfungscharakter, das heißt, eine **Teilnahme** an den einzelnen Übungsstunden ist **verpflichtend** (siehe auch 1.2.5). Wenn Sie eine andere Lösung benötigen bzw. verhindert sind, kontaktieren Sie unbedingt den jeweiligen Übungsleiter.

1.2.4 Unterlagen und Dokumentation

Dieses Dokument wird laufend aktualisiert und soll jederzeit über die WEB-Seite des Instituts für Theoretische Physik www.itp.tu-graz.ac.at abrufbar sein. Ein darüber hinaus gehendes Skriptum wird es wegen der Schnelligkeit der Entwicklung am Computer- und Softwaresektor nicht geben. Wir werden aber eine Reihe von Dokumenten und Hilfssystemen über die oben genannte WEB-Seite anbieten.

Die gesamte MATLAB Dokumentation ist verfügbar unter
http://www.itp.tu-graz.ac.at/matlab/helpdesk.html.
Der Zugang zu vielen Büchern im sogenannten "Portable Document Format - pdf" findet sich ebenfalls auf diesem Server
www.itp.tu-graz.ac.at.

1.2.5 Prüfungen

Da der Schwerpunkt dieser Lehrveranstaltung nicht die Vermittlung von Fakten ist, sondern hier der Zugang zu Problemlösungen mit Hilfe von Computern erleichtert werden soll, ist auch das Prüfen von Fakten nicht das erklärte Ziel. Entscheidend hingegen ist, welche Kompetenz Sie bei der Lösung von Problemen an den Tag legen. Dazu sind jeweils alle Hilfsmittel (Unterlagen, Fragen, Hilfssysteme, Internet, ...) erlaubt. Diese Vorgangsweise soll eher eine Problemlösung während des Studiums oder während der Forschung simulieren.

Es soll hier nochmals darauf hingewiesen werden, dass eine Teilnahme an allen Übungseinheiten notwendig ist, außer es wurden andere Vereinbarungen mit uns getroffen. Bei Verhinderung ersuchen wir um eine Absage z.B. durch eine E-mail an den Übungsleiter.

Die Grundvoraussetzung für die Ablegung der Abschlussprüfung ist die Abgabe **aller Übungsbeispiele** auf elektronischem Weg. Für die Abgabe der Übungsbeispiele ist eine Frist von jeweils 2 Wochen vorgesehen. Genaue Daten dazu werden jeweils vorgegeben. Die Übungsbeispiele werden von einem Tutor korrigiert, eine Rückmeldung wird direkt über ein Computerprogramm erfolgen.

Für einen Abschluss der Lehrveranstaltung bieten wir folgende Möglichkeiten an:

- Aktive Teilnahme an den Übungen und Abgabe aller Beispiele. Teilnahme an einem Prüfungstermin am Computer, Lösung von Problemen unter Zuhilfenahme aller Unterlagen. Prüfungsgespräch mit dem Vortragenden direkt nach Abgabe.
- Durchführung von Projektarbeiten im Umfeld der Lehrveranstaltung. Das Thema kann bzw. sollte bevorzugt aus Ihrem Interessensgebiet oder aus einer anderen Lehrveranstaltung stammen und mit hier besprochenen Methoden behandelt werden. Ergebnisse können dann auch auf unserer WEB-Seite präsentiert werden. In diesem Fall muss nicht unbedingt an den Übungen teilgenommen werden. Eine vorherige Absprache ist aber unbedingt erforderlich. Diese Möglichkeit richtet sich vor allem an Hörer, die bereits gute Kenntnisse in MATLAB haben.

Vom Vortragenden wird angestrebt, dass ein Großteil der Studierenden die Lehrveranstaltung am Ende des Semesters noch vor den Ferien mit einem positiven Zeugnis abschließen kann. Es wird aber nochmals darauf hingewiesen, dass die **aktive Teilnahme** an den Übungen und die selbstständige Lösung der Übungsbeispiele eine Voraussetzung dafür ist. Termine und genauere Anforderungen werden rechtzeitig vor Ende des Semesters bekanntgegeben.

1.2.6 Sprache

Die Vortragssprache ist Deutsch. Viele Dokumentationen und Beschreibungen bzw. das Hilfesystem von viele Programmen ist aber natürlich in Englisch. Dadurch wird die Benutzung beider Sprachen notwendig.

1.3 Computerzugang für Studierende

Da wir für unsere gemeinsame Arbeit Zugang zu Computern (oder, falls eigene Computer vorhanden sind, Zugang zum TU-Netz) brauchen, habe ich in der Folge einige interessante Fakten zusammengestellt. Nähere Informationen dazu finden Sie auf der WEB-Seite des Zentralen Informatik Dienstes unter http://www.zid.tu-graz.ac.at bzw. unter http://www.vc-graz.ac.at.

Im Bereich der Physik finden Sie Informationen unter:

- http://www.itp.tu-graz.ac.at

1.3.1 Computer für Studierende im Bereich Physik

Der Bereich Physik hat im Bereich der studentischen Ausbildung eine Sonderstellung. Für unsere speziellen Bedürfnisse steht ein eigener Computerraum zur Verfügung.

Die Ausstattung besteht aus 15 Workstations für Studierende, an denen auch in Zweiergruppen gearbeitet werden kann. Für den Vortragenden besteht ein eigener Platz mit Projektionsmöglichkeiten direkt vom Rechner aus. Damit sollte eine Gruppengröße von 15 bzw. 30 (in Zweiergruppen) Studierenden möglich sein.

Die Computer sind mit den Betriebssystemen LINUX und der gesamten relevanten Software ausgestattet. Vorgesehen ist die Verwendung sowohl für Übungen als auch für die gesamte studentische Arbeit an Computern. Durch die Verwendung des Betriebssystems LINUX ist auch eine Verwendung von Programmen von außerhalb (siehe externer Zugang) möglich. Bei einigen Rechnern steht auch das Betriebssystem WINDOWS NT als Gastbetriebssystem zur Verfügung (Office, ...). Die Lehrveranstaltungen werden aber zur Gänze unter LINUX abgewickelt

Für dieses Computersystem ist eine getrennte Anmeldung über die WEB-Seite des Instituts für Theoretische Physik **www.itp.tu-graz.ac.at** unbedingt erforderlich. Hier ist im Gegensatz zu den Subzentren keine freie Wahl des Passwortes möglich, das Passwort wird Ihnen nach der Anmeldung ausgehändigt. Weitere Informationen über dieses Computersystem finden sie auf der WEB-Seite http://fubphpc.tu-graz.ac.at.

Anders als in den Subzentren stehen die Rechner rund um die Uhr zur Verfügung. Das einzige Problem dabei ist der Zugang zum Physikgebäude.

Wir haben uns bemüht einige interessante Manuals zusammenzustellen, insbesondere auch eine kurze .Einführung in LINUX

1.3.2 Externer Zugang über ISDN, Modem, Virtual Campus oder Telekabel

Der Zentrale Informatikdienst der TU Graz bietet für die Angehörigen der TU (Mitarbeiter und Studierende) einen externen Zugang in das TUGnet mit Modem Verbindung oder ISDN-Verbindung an. Für die Bewohner diverser Studentenheime gibt es die Möglichkeit via Netzwerk am sogenannten "Virtual Campus" teilzunehmen. Die Firma Telekabel bietet einen verbilligten Zugang für Studierende über Kabel-Modem an.

Unabhängig vom gewählten Internetprovider kann man sich auf unseren LINUX-Rechnern anmelden bzw. Daten von und zu diesen Rechnern transferieren. Gängige Protokolle dafür sind

Protokoll	Beschreibung	Sicherheit
ssh	Secure Shell	Verschlüsselt
scp	Secure Copy	Verschlüsselt
rsync	Synchronisieren	Verschlüsselung möglich

Zu all diesen Protokollen gibt es Clients auf allen Betriebssystemen. Eine wirkliche Hilfe von uns in Installationsfragen kann es aber nicht geben. Unsere Unterstützung beschränkt sich auf die Bereitstellung der Dienste auf Serverseite auf allen Rechnern. Dies sind die Rechner fubphpcxx.tu-graz.ac.at, wobei xx für die Zahlen 01 bis 16 mit Ausnahme von 09 steht.

Um auch Grafik übertragen zu können, braucht man eine X-Windows Server Software. Auch die gibt es für alle Betriebssysteme.

1.4 Kommunikation

Neben der Kommunikation während und nach den Vorlesungen und Übungen, sollte vor allem die Kommunikation über Electronic Mail stattfinden. Ich bin unter meiner Mail-Adresse **winfried.kernbichler@tugraz.at** zu erreichen.

Außerdem gibt es für die Übungen zwei Tutoren:
Franz Rieder, **dieter.mayer@itp.tu-graz.ac.at**, und
Oliver Teschl, **teschl@fubphpc.tu-graz.ac.at**.

Die studentischen Betreuer unserer Computeranlage sind Andreas Hirczy, erreichbar unter **hirczy@itp.tu-graz.ac.at**, Christian Pfaffel, erreichbar unter **pfaffel@itp.tu-graz.ac.at**, und David Camhy, erreichbar unter **camhy@itp.tu-graz.ac.at**,

1.5 Dokumente

Dieses Dokument kann als PDF Datei von unserem Server geladen werden.

Die Erstellung dieses Dokuments und auch der appsoft1_talk.pdf erfolgt mit **pdflatex**, einem Programm zum Erzeugen von PDF-Dateien direkt aus der Typesetting-Sprache **LATEX**.

In Zukunft werden hier noch weitere Referenzen zu interessanten Dokumenten angeboten werden.

1.6 Programmpakete

Der Schwerpunkt unserer Arbeit wird auf dem Programmpaket MATLAB basieren. Der Name steht für MATrix LABoratory und bezieht sich auf eine herausragende Eigenschaft von MATLAB, nämlich die Fähigkeit fast alle Befehle auf Vektoren bzw. Matrizen anwenden zu können.

Das Paket ist gleichzeitig:

- eine Art Taschenrechner auch für Vektoren und Matrizen
- eine Programmiersprache
- ein Compiler
- ein mächtiges Programm zur Visualisierung
- ein Tool zur Erstellung von Graphical User Interfaces (GUI)
- erweiterbar durch Toolboxen zu den verschiedensten Themenbereichen
- ein graphisches Werkzeug zur Simulation von komplexen Abläufen (SIMULINK)
- eine Schnittstelle zu symbolischen Rechenprogrammen (MAPLE)
- eine Schnittstelle zu anderen Programmiersprachen (C, FORTRAN)

In Ergänzung dazu wird auf Seite der symbolischen Programmpakete MAPLE vorgestellt werden. Dabei werden wir uns maximal mit wenigen Grundzügen bschäftigen bzw. die Verbindung zwischen MATLAB und MAPLE kennenlernen.

Numerischen Programme wie MATLAB und symbolische Programme wie MAPLE oder MATHEMATICA unterscheiden sich in folgendem Punkt:

MATLAB Numerische Programme arbeiten mit Zahlenwerten, das heißt, einer Variablen muss ein Wert zugewiesen werden, $x = 1 : 10$ (Vektor der Zahlen 1 bis 10), und dann können Operationen darauf angewandt werden, z.B.: $y = sin(x)$. Resultate liegen daher immer "numerisch" vor und sind mit der inhärenten Ungenauigkeit von numerischen Darstellungen behaftet. Numerische Programme haben daher ihre Bedeutung bei einer großen Anzahl "symbolisch" nicht lösbarer Probleme bzw. bei der Verarbeitung von numerisch vorliegenden Daten (Messdaten, ...).

MAPLE Symbolische Programme hingegen arbeiten mit Variablen, denen keine numerischen Werte zugewiesen sind. Hier liefert z.B. die Eingabe y=int(x^2,x) das Ergebnis $y = x^3/3$. Danach können dann bei Bedarf Werte für x eingesetzt werden. Lösbare Probleme können daher auf exakte Art und Weise gelöst werden.

Der Unterschied sei hier am Beispiel der Differentiation erklärt. In einem symbolischen Rechenprogramm kann die Differentiation exakt ausgeführt werden, falls eine Lösung existiert

$$\frac{d}{dx} \sin x = \cos x \ . \tag{1.1}$$

In der Numerik hingegen liegen Zahlenwerte, z.B. in Form eines Vektors vor

$$\mathbf{xv} = [x_1, x_2, \ldots, x_n] \ , \tag{1.2}$$

wobei n die Anzahl der Elemente im Vektor $\mathbf{xv}$ ist. Mit dem Befehl

$$\mathbf{yv} = \sin(\mathbf{xv}) \ , \mathbf{yv} = [y_1, y_2, \ldots, y_n] \ , \tag{1.3}$$

kann man nun einen Vektor $\mathbf{yv}$ der gleichen Länge n erzeugen. Die Differentiation kann jetzt aber nur näherungsweise mit Hilfe des Differenzenquotienten

$$\frac{d}{dx} \sin x \approx \frac{\Delta (\sin x)}{\Delta x} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \ , \tag{1.4}$$

erfolgen.

Diese Vorgangsweise mag hier unlogisch erscheinen, sie funktioniert aber auch dann, wenn überhaupt kein funktionaler Zusammenhang bekannt ist (z.B.: Messdaten) oder wenn ein Problem nicht exakt lösbar ist. In der Realität ist deshalb eine numerische Behandlung von Problemen häufig notwendig. Man muss sich aber natürlich immer im Klaren sein, dass die Numerik mit Ungenauigkeiten behaftet ist.

1.6.1 Software Version

Im Computerraum Physik ist derzeit die neueste Version MATLAB 6.5 installiert:

```
MATLAB Version 6.5.1.199709 (R13) Service Pack 1
```

Kapitel 2

Basis Syntax in MATLAB

2.1 Variablen und Zuweisung von Werten

Neben der Möglichkeit MATLAB als eine Art überdimensionalen Taschenrechner zu benutzen

```
3*(5 + 8)
3 + sin(pi/3)
```

kann man Ergebnisse auch benannten Größen (Variablen) zuweisen. Das Zeichen für Zuweisung (assignment) ist das = Zeichen. So erzeugte Größen können in der Folge für weitere Berechnungen herangezogen werden.

```
a = 3 * (5 + 8)
a = (a - 1) / 4
b = sin(0.5)
```

Wichtig dabei ist, dass Größen die rechts vom = Zeichen stehen durch vorige Berechnungen bekannt sind und gültige Operatoren (z.B.: `+`, `-`, ...), bzw. gültige Programmnamen (z.B.: `sin`) enthalten.

Wird ein Name wieder auf der linken Seite einer Zuweisung verwendet (`a` in der zweiten Zeile) wird zuerst die rechte Seite mit dem alten Wert von `a` berechnet und dann dieser Wert der Variablen zugewiesen. Die alte Bedeutung geht dabei dann verloren.

Insgesamt muss natürlich auch die Sprachsyntax (Regelwerk, Grammatik) korrekt sein. Insbesonders müssen alle Klammern geschlossen sein und Operatoren und Funktionen richtig verwendet werden.

Ein riesiger Vorteil von MATLAB ist, dass viele Befehle direkt auf ganze Felder bzw. Matrizen angewandt werden können.

So berechnet der Befehl

```
s  = sin( [0.0, 0.1, 0.2, 0.3] )
```

den Sinus aller vier Werte und speichert ihn in der Variablen `s`. Zusammen mit der entsprechenden Syntax für die Konstruktion von Feldern ermöglicht dies eine sehr elegante und auch effiziente Programmierung.

Einige einfache Fehler und entsprechende Fehlermitteilungen von MATLAB, die immer in der Farbe rot im Kommando-Fenster ausgegeben werden, finden sich hier:

FEHLER	FEHLERMELDUNG
`a = 3+`	`Error: Expected a variable, function, or constant, found ”end of line”.`
`a = (3+4`	`Error: ”)” expected, ”end of line” found.`
`a = sin()`	`Error: Expected a variable, function, or constant, found ”)”.`
`sin(1,2,3)`	`Error using ==> sin` `Incorrect number of inputs.`
`a = six(1)`	`Undefined function or variable ”six”.`
`s = 'Winfried`	`Error: Expected a variable, functi-on, or constant, found ”incomplete string”.`
`3a = sin(1)`	`Error: Missing operator, comma, or semicolon.`

Die letzte Fehlermitteilung bezieht sich auf die Verwendung eines falschen Variablennamens. Gültige Variablennamen in MATLAB müssen mit einem Buchstaben beginnen und dürfen nur Buchstaben, Zahlen und als einziges Sonderzeichen den Untersstrich _ verwenden. Groß- und Kleinbuchstaben werden zumindest unter LINUX unterschieden. Die Verwendung von Umlauten ist unter WINDOWS möglich, sollte aber vermieden werden, da solche Programme unter LINUX dann nicht funktionieren. Äusserst ungeschickt ist es auch Namen von MATLAB-Funktionen und vordefinierten Konstanten 2.2 zu verwenden.

GÜLTIG	UNGÜLTIG	UNGESCHICKT
`a  a3  a_3`	`3a  3_a  a*`	`Maß  ö3  ö_3`
`Sin  SIN  MAX`	`Sin(  Sin()  Max+`	`sin  max  abs`
`k  l  m`	`"k"  k!  k#`	`i  j  pi`

Treten Fehler in MATLAB-Skripts oder MATLAB-Funktionen auf, wird zusätzlich zur Fehlermitteilung auch die Position im File angegeben. Dies sollte die Fehlersuche in Programmen wesentlich erleichtern.

2.2 Mathematische Konstanten

In MATLAB sind eine Reihe von mathematischen Konstanten vordefiniert. Ihre Namen und ihre Bedeutung findet sich in der nachfolgenden Tabelle. Sie sollen auf keinen Fall überschrieben werden. Wenn Sie mit komplexen Zahlen rechnen, vermeiden Sie daher unbedingt die Verwendung von `i` und `j` als Zählvariablen in Schleifen.

KONSTANTE	BEDEUTUNG	WERT
`eps`	Relative Genauigkeit von Fließkommarechnungen.	$2.2204 \cdot 10^{-16}$
`i`, `j`	Imaginäre Einheit.	$\sqrt{-1}$
`inf`, `Inf`	Unendlich.	∞
`nan`, `NaN`	Not A Number - Keine Zahl im herkömmlichen Sinn. Entsteht z.B. durch $\frac{0}{0}$, $\frac{\infty}{\infty}$, $0 \cdot \infty$ und durch jede Operation mit `nan`.	nan
`pi`	Verhältniss zwischen Umfang des Kreises und seines Durchmessers.	3.14159
`realmax`	Größte positive Fließkommazahl.	$1.79769 \cdot 10^{+308}$
`realmin`	Kleinste positive Fließkommazahl.	$2.22507 \cdot 10^{-308}$

2.3 Wichtige Befehle

In der folgenden Tabelle sind einige für das Arbeiten mit MATLAB wichtige bzw. paraktische Befehle aufgelistet.

BEFEHL	BEDEUTUNG
`quit`	Beendet MATLAB.
`exit`	Beendet MATLAB.
`clc`	Löscht MATLAB-Kommandofenster.
`diary`	Schreibt Befehle und Ergebnisse in einem File mit.
`format`	Beeinflusst Outputformat.
`system`	Führt Befehl im zugrundeliegenden Betriebssystem aus.
`clear`	Löscht Variable im Workspace (z.B.: `clear all`).
`close`	Schließt Graphikfenster (z.B.: `close all`).
`who`, `whos`	Listed Variablen im Workspace auf.
`exist`	Überprüft, ob ein Name bereits existiert.

2.4 Möglichkeiten für Hilfe in MATLAB

MATLAB bietet eine Reihe von Möglichkeiten Hilfe zu Befehlen bzw. zur Syntax der Programmiersprache zu finden. Hier sind einige wichtige aufgelistet, die sich als äussert praktisch erwiesen haben.

BEFEHL	BEDEUTUNG
`helpbrowser`	Startet den Browser für das ausführliche Helpsystem von MATLAB.
`help`	Zeigt MATLAB-Hilfe im Kommandofenster. `help help`: Hilfe über den Hilfebefehl. `help sin`: Hilfe für die Funktion Sinus.
`lookfor`	Sucht nach speziellen Schlüsselwörtern im Hilfesystem. `lookor hyperbolic`: Listet alle Befehle, wo in der Hilfe das Schlüsselwort vorkommt.

2.5 Spezielle Zeichen - Special Characters

Um eine Programmiersprache wie MATLAB korrekt benutzen zu können, muss man die Bedeutung von speziellen Zeichen kennen.

2.5.1 Klammern

2.5.1.1 Runde Klammern - Parenthesis

Runde Klammern erfüllen in MATLAB im Wesentlichen eine Abgrenzungsfunktion zwischen arithmetischen Ausdrücken (Gliederung), zwischen Feldname und Indices (Indizierung), bzw. zwischen Funktionsnamen und Argumenten. Einige Beispiele sind in folgender Tabelle zusammengefasst.

KLAMMER	BEDEUTUNG	BEISPIEL
`()`	Gliederung arithmetischer Ausdrücke	`3*(a+b)`

	Indizierung von Feldern
	Ein Element (Zeile, Spalte)
	Mehrere Elemente
		`a(3)`
		`a(1,3)`
		`a([1,3,5])`

	Abgrenzung von Argumenten
	Übergabeparameter an Funktionen
		`sin(a)`
		`plot(x,y1,x,y2)`
		`plus(3,4)`

2.5.1.2 Eckige Klammern - Brackets

Mit Hilfe von eckigen Klammern werden in MATLAB Vektoren und Matrizen erzeugt bzw. zusammengefügt. Ausserdem werden sie bei der Rückgabe von Funktionswerten verwendet, wenn es mehrere Ergebnisse gibt. Einige Beispiele sind in folgender Tabelle zusammengefasst.

KLAMMER	BEDEUTUNG	BEISPIEL

`[]`	Erzeugen von Vektoren
	(auch ohne Beistrich)
	Bereich (von - bis)
	Schrittweite
	Leeres Feld
		`a=[1,2,3,4,5]`
		`a=[1 2 3 4 5]`
		`a=[1:5]`
		`b=[1:2:5]`
		`c=[]`

	Erzeugung von Matrizen
	Nebeneinander
	Übereinander
	Zeichenketten
		`[1,2,3;4,5,6]`
		`[a,a]`
		`[a;a]`
		`['ich',' ','bin']`

	Mehrere Ausgabewerte	`[a,b,c]=func(x,y,z)`

2.5.1.3 Geschwungene Klammern - Curly Braces

Geschwungene Klammern werden in MATLAB für die Erzeugung und Indizierung von Zellen verwendet. Zellen sind Felder, die an jeder Stelle beliebige Elemete (Felder, Zeichenketten, Strukturen) und nicht nur Skalare enthalten können.

KLAMMER	BEDEUTUNG	BEISPIEL
`{}`	Erzeugen von Zellen Leere Zelle	`z={[1:3],'string'}` `l={}`
	Zugriff auf Zellelemente Zuweisung	`a=z{1}` `z{3}=[1,2;3,4]`

2.5.2 Punkt - Dot

Punkte haben eine vielfältige Bedeutung in MATLAB, wobei die wichtigste wohl der Dezimalpunkt ist:

ZEICHEN	BEDEUTUNG	BEISPIEL
`.`	Dezimalpunkt auch in Fließkommazahlen $1.5 \cdot 10^{-5}$	`p=3.14` `1.5e-5`
`.`	Zugriff auf Strukturelemente	`s.f` `s.('f')`
`..`	Übergeordnetes Verzeichnis	`cd ..`
`...`	Fortsetzungszeile	`m=[1,2; ...` `3,4]`
`.* ./ .\ .^`	Operator für alle Elemente z.B.: Quadrieren	`[1,2,3].*[4,5,6]` `[1,2,3].^2`
`.'`	Transponieren 2.5.5	M.'

2.5.3 Komma und Strichpunkt - Comma and Semicolon

Komma und Strichpunkt fungieren im Wesentlichen als Trennzeichen:

ZEICHEN	BEDEUTUNG	BEISPIEL
`,`	Trennzeichen - Spalte	`[1,2,3]`
`;`	Trennzeichen - Zeile Spaltenvektor	`[1,2,3;4,5,6]` `[1;2;3]`
`,`	Trennzeichen - Index höhere Dimension	`a(3,4)` `a(m,n,o,p,q)`
`,`	Trennzeichen - Funktion auch bei Output	`plus(3,4)` `[a,b]=func(x,y)`
`,`	Trennzeichen - Kommando mit Ausgabe	`a=[1,2], b=5` `a=3,b=a,c=a`
`;`	Trennzeichen - Kommando ohne Ausgabe	`a=[1,2]; b=5;` `a=3;b=a;c=a;`

Hier gibt es eine interessante Fehlermöglichkeit, nämlich die Verwechslung von `.` (Punkt) mit `,` als Dezimalzeichen. Das MATLAB-Kommando

```
a = 3,4
```

liefert keine Fehlermitteilung, sondern setzt `a=3`, zeigt es wegen des Kommas am Schirm an, setzt die Variable `ans=4` und zeigt sie ebenfalls am Schirm an.

Anmerkung: Die Variable `ans` wird immer für das letzte Resultat verwendet, wenn keine explizite Zuweisung erfolgt.

MATLAB kann in diesem Fall keine Fehlermitteilung anzeigen, da es sich um eine korrekte Eingabe handelt, die "nur" etwas anderes berechnet, als sich der Benutzer vielleicht erwartet.

Fehler, die sich auch öfters ergeben, sind hier mit Fehlermitteilungen angeführt:

FEHLER	FEHLERMELDUNG
`a = [1,2;3,4`	`Error: "]" expected, "End of Input" found.`
`a = [1,2;3]`	`Error using ==> vertcat` `All rows in the bracketed expression must have the same number of columns.`

Die letzte Fehlermitteilung beruht darauf, dass das Feldelement `a(2,2)` fehlt, und ein Feld in allen Positionen besetzt sein muss. Will man markieren, dass an dieser Stelle der Matrix eigentlich kein "richtiger" Wert steht, kann man sich der Zahl `nan` (Not a Number) bedienen. Richtig müsste es also heissen:

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

2.5.4 Doppelpunkt - Colon

Die `Doppelpunktnotation` ist eine der mächtigsten Bestandteile von MATLAB. Sie kann einerseits zur Konstruktion von Vektoren (Tab. 3.2), aber auch zum Zugriff auf Teile von Matrizen (Index, 3.6) verwendet werden. Alle Details dazu findet man im Abschnitt 3.4.2. Hier werden nur elementare Beispiele angeführt:

```
m = [1:5]      % [1,2,3,4,5]
m = [1:2:5]    % [1,3,5]
m = [5:-1:1]   % [5,4,3,2,1]
x = [0:0.1:2]  % [0,0.1,0.2, ...., 1.9,2.0]
```

2.5.5 Hochkomma - Quotation Mark

Das Hochkomma wird zur Definition von Zeichenketten (Strings) verwendet:

```
str1 = 'Winfried'; str2 = 'Kernbichler';
str3 = 'Resultat: ';
str4 = num2str( sin(1) );
disp([str3,str4])
```

Details zu diesen Beispielen findet man im Abschnitt 9. Die hier verwendeten Befehle `num2str` und `disp` dienen zur Umwandlung von Zahlen in Zeichenketten und zur Darstellung von Ergebnissen im MATLAB-Kommandofenster.

Eine weitere Verwendung findet das Hochkomma als Operator für das Transponieren und das komplex konjugierte Transponieren von Matrizen. Wenn `M` eine Matrix ist, kann man den Operator wie folgt anwenden:

```
M1 = M.'      % transpose
M2 = M'       % conjugate complex transpose
```

Diese Anwendung ist ident mit den Befehlen `ctranspose`, bzw. mit `transpose`. Details zum Bearbeiten von Matrizen findet man im Abschnitt 5.

2.5.6 Prozent und Rufzeichen - Percent and Exclamation Point

Mit Hilfe des Prozentzeichens % können Kommentare in MATLAB-Programme eingefügt werden. Macht man das am Anfang des Files, z.B. wie

```
% Program: func
% Aufruf:  [a,b] = fucn(x,y)
% Beschreibung ......
% Input:   x: Beschreibung
%          y: ....
% Output:  a: ....
%          b: ....
% Autor:   Winfried Kernbichler
% Datum:   01-03-2004
```

kann man diese Kommentare als Programmdokumentation verwenden, die mit dem Befehl `help` einfach betrachtet werden kann.

In weiterer Folge kann man dann Programmzeilen kommentieren,

```
% Abstandsberechnung
d = sqrt(x.^2 + y.^2);
[ds,ind] = sort(d); % Sortierung nach Größe
```

wobei dies in eigenen Zeilen oder am Ende von Zeilen gemacht werden kann.

Mit Hilfe des Rufzeichens können Systembefehle an das Betriebssystem übergeben werden, die dann ausserhalb von MATLAB abgearbeitet werden:

```
!cp file1 file2      % Kopieren
!mv file1 file2      % Verschieben
```

Das Rufzeichen ist eine Kurzform des MATLAB-Befehls `system`, der auch die Rückgabe der Ergebnisse auf Variable ermöglicht.

Einige wichtige Systembefehle sind aber auch direkt in MATLAB vorhanden:

BEFEHL	BEDEUTUNG
`dir`	Verzeichnis Listing.
`pwd`	Anzeige des aktuellen Verzeichnisses.
`cd`	Wechsel des Verzeichnisses.
`mkdir`	Anlegen von Verzeichnissen.
`rmdir`	Löschen von Verzeichnissen.
`delete`	Löschen von Files.
`copyfile`	Kopieren von Files.
`movefile`	Verschieben von Files.
`fileattrib`	Setzen von Fileattributen (Rechte).

2.5.7 Operatoren

Eine Reihe von Zeichen sind für Operatoren reserviert, die hier nur kurz angeführt werden sollen. Details findet man in den jeweiligen Verweisen:

TYP	VERWEIS	ZEICHEN
Arithmetisch - Matrizen	5	`+ - * / \ ^`
Arithmetisch - Elemente	4	`+ - .* ./ .\ .^`
Transponieren	5	`' .'`
Vergleich	4.2	`== ~= < <= > >=`
logisch	4.3	`~ & |`

2.6 Schlüsselwörter - Keywords

In MATLAB sind eine Reihe von Schlüsselwörtern definiert, die im Wesentlichen zu Steuerkonstrukten 6 gehören. In alphabetischer Reihenfolge sind dies:

`break`	`case`	`catch`	`continue`	`else`
`elseif`	`end`	`for`	`function`	`global`
`if`	`otherwise`	`persistent`	`return`	`switch`
`try`	`while`			

2.7 MATLAB-Skripts und MATLAB-Funktionen

MATLAB kennt zwei Typen von Programmeinheiten, Skripts und Funktionen, die im Detail im Abschnitt 7 besprochen werden.

MATLAB-Skripts sind Programme, die im MATLAB-Arbeitsbereich (Workspace) ablaufen und keine Übergabeparameter kennen. Ihnen sind alle definierten Variablen

im Workspace (`who`) bekannt, zwei unterschiedliche Skripts können also wechselseitig Variablen benutzen oder überschreiben. Das kann einerseits praktisch sein, birgt aber andererseits auch große Gefahren unerwünschter Beeinflussung. Will man sicher sein, dass Skripts in einem "reinen" Workspace ablaufen, muss man den Befehl `clear all` verwenden.

MATLAB-Funktionen 7.1 hingegen werden in einem eigenen Arbeitsbereich abgearbeitet. Hier gibt es also keine unerwünschten Querverbindungen. Ihre Kommunikation mit Skripts (oder anderen Funktionen) erfolgt durch sogenannte Übergabeparameter,

```
function [out1,out2,out3] = test(in1,in2,in3)
```

wobei die Position innerhalb der Klammern die Zuordnung bestimmt. Ein Aufruf der Funktion `test` in folgender Art,

```
[a,b,c] = test(x,y,z)
```

führt innerhalb der Funktion dazu, dass `in1` den Wert von `x`, `in2` den Wert von `y` und `in3` den Wert von `z` zugewiesen bekommt.

Nach Ablauf aller Programmschritte innerhalb der Funktion `test`, wobei die Werte für `out1`, `out2` und `out3` berechnet werden müssen, bekommt ausserhalb der Funktion die Variablen `a` den Wert von `out1`, `b` den Wert von `out2` und `c` den Wert von `out3`.

Der lokale Arbeitsbereich einer Funktion ist bei jedem Aufruf leer. Nach dem Aufruf sind also nur die Input-Parameter bekannt. In Funktionen können natürlich genauso wie in Skripts alle MATLAB-Befehle und eigene Programme verwendet werden.

Zwei Regeln müssen bei Funktionen unbedingt eingehalten werden. Erstens, die Funktion muss in einem gleichnamigen MATLAB-File abgespeichert werden, d.h., die Funktion `test` muss im File `test.m` gespeichert werden und steht dann unter dem Namen `test` zur Verfügung. Dabei soll man keinesfalls existierende MATLABFunktionsnamen (`exist`) verwenden, da sonst deren Zugänglichkeit blockiert ist. Zweitens, muss die Funktion eine sogenannte Deklarationszeile enthalten, die den Funktionsnamen und die Namen (und somit die Anzahl) der Übergabeparameter enthält. Diese Zeile muss mit `function` beginnen (siehe oben).

2.7.1 Einfache Beispiele

Zur Einführung werden hier zwei einfache Beispiele gezeigt, wobei man weiterführende Beispiele im Abschnitt 7.1.10 findet.

Eine einfache Funktion mit zwei Input- und zwei Output-Parametern könnte so aussehen:

```
function [a,b] = test_fun1(x,y)
% TEST_FUN1 - Test Function
%   Syntax:   [a,b] = test_fun1(x,y)
%   Input:    x,y - Array (same size)
%   Output:   a,b - Array (same size as x and y)
%             a = sqrt(x.^2 + y.^2);
%             b = exp(-a.^2)
a = x.^2 + y.^2;
b = exp(-a);
a = sqrt(a);
```

Sie besteht aus einer Deklarationszeile, einer Reihe von Kommentarzeilen, die mit dem Befehl `help` angezeigt werden können, und drei Programmzeilen zur Berechnung der Output-Parameter. Beachten Sie bitte die Verwendung des Operators `.^`, da es sich bei den Übergabegrößen um Felder handeln kann (elementweise Berechnung).

Das entsprechende Skript zum Aufruf der Funktion könnte so aussehen:

```
% Test-Skript for test_fun1.m
% Winfried Kernbichler 08.03.2004
z_max = 1; z_n   = 101;  % Prepare input
z_1 = linspace(-z_max, z_max, z_n);
[d, e] = test_fun1(z_1, -z_1);  % Call function
figure(1); % Plot output
plot(z_1,d,'r',z_1,e,'b:');
xlabel('z_1'); ylabel('f(z_1,-z_1)');
legend('Distance','Exponent');
title('Result of test_fun1');
```

Im aufrufenden Skript werden typischerweise die Input-Parameter vorbereitet und die Ergebnisse dargestellt. Man trennt damit das "Umfeld" von der eigentlichen Berechnung.

Will man mit dem Benutzer des Programmes kommunizieren, kann man zur Eingabe von `z_max` und `z_n` auch den Befehl `input` eventuell in folgender Form verwenden:

```
z_max = input('Bitte geben Sie z_max ein: ');
```

Manchmal möchte man eine Funktion auch für die Erledigung unterschiedlicher Aufgaben verwenden. Dazu bietet sich die Verwendung der Steuerstruktur `switch` an. Bei dieser Steuerstruktur wird eine Schaltvariable `switch` benutzt. Für verschiedene Werte dieser Schaltvariablen können dann Fälle (`case`) und entsprechende Aktionen programmiert werden.

```
function [a,b] = test_fun2(typ,x,y)
% TEST_FUN2 - Test Function
%   Syntax:   [a,b] = test_fun2(typ,x,y)
%   Input:    typ - String
%             x,y - Array (same size)
%   Output:   a,b - Array (same size as x and y)
%             a = sqrt(x.^2 + y.^2);
%             b = exp(-a.^2)  [typ: 'exp']
%             b = sech(-a.^2) [typ: 'sech']
a = x.^2 + y.^2;
switch typ
 case 'exp'
  b = exp(-a);
 case 'sech'
  b = sech(-a);
 otherwise
  error('Case not defined!')
end
a = sqrt(a);
```

Je nach Wert der der Variablen `typ` kann die Funktion nun zwei verschiedene Aufgaben erledigen. Eine genaue Beschreibung von Steuerstrukturen finden Sie im Abschnitt 6, Details zum Befehl `switch` finden Sie im Abschnitt 6.2.2.

Der Aufruf schaut nun ein wenig anders aus (`typ`):

```
% Test-Skript for test_fun2.m
% Winfried Kernbichler 08.03.2004
z_max = 1; z_n   = 101;  % Prepare input
typ   = 'exp';  % oder 'sech'
z_1 = linspace(-z_max, z_max, z_n);
[d, e] = test_fun2(typ,z_1, -z_1);  % Call function

figure(1); % Plot output
plot(z_1,d,'r',z_1,e,'b:');
xlabel('z_1'); ylabel('f(z_1,-z_1)');
legend('Distance',typ);
title('Result of test_fun2');
```

Anmerkung: Will man mit der Funktion `input` den Wert der Variablen `typ` abfragen, empfiehlt es sich, sie in dieser Form

```
typ = input('Bitte geben Sie den Typ ein: ','s')
```

zu verwenden. Damit kann man einfach `exp` anstelle von `'exp'` eingeben und MATLAB erkennt trotzdem, dass es sich um einen String handelt.

Anmerkung: Will man die Eingabe des Typs noch weiter erleichtern (Groß- oder Kleinschreibung, nur Anfangsbuchstabe(n), kann man die `switch`-Konstruktion verbessern. Man kann z.B. alle Zeichenketten mit dem Befehl `lower` in Kleinbuchstaben verwandeln. Details zur Verwendung von Zeichenketten findet man im Abschnitt 9. Die Konstruktion

```
switch lower(typ)
 case 'exp'
  ...
end
```

würde nun auch mit Werten wie `Exp` oder `EXP` funktionieren. Will man nur den Anfangsbuchstaben der Zeichenkette auswerten, kann man mit Hilfe der Indizierung auf den ersten Buchstaben zugreifen. Dies könnte so aussehen:

```
switch lower(typ(1))
 case 'e'
  ...
end
```

Kapitel 3

Arrays

3.1 Konzept

Eine der großen Stärken von MATLAB liegt im einfachen Umgang mit Matrizen bzw. Arrays (Felder), wobei diese beiden Bezeichnungen praktisch gleichbedeutend verwendet werden. In MATLAB werden beinahe alle Größen als Arrays behandelt. An dieser Stelle beschränken wir uns auf numrische Arrays, deren Inhalt Zahlen sind. Später werden auch andere Typen, wie z.B.: Zeichenketten, Zellen, oder Strukturen besprochen werden. Am einfachsten vorstellen kann man sich also ein Array als eine geordnete Anordnung von Zahlen, deren Bedeutung natürlich unterschiedlich sein kann.

So kann man den Inhalt verstehen als,

- Matrix im Sinne der linearen Algebra,
- Tensor oder Vektor im Sinne der Vektor-Tensor-Rechnung,
- Menge von Zahlen im Sinne der Mengenlehre,
- numerisches Ergebnis einer Berechnung, z.B.: der Funktion $f(x, y) = \sin xy$ für verschiedene (geordnete) Werte von x und y,
- Resultat eines Lesevorgangs (Zeilen und Spalten einer Tabelle).

Anders als die meisten anderen Programmiersprachen kann Matlab die meisten Operationen nicht nur auf einzelne Zahlen, sondern auch auf ganze Arrays anwenden. Man kann also beispielsweise Matrizen miteinander multiplizieren, muss sich aber natürlich bewußt sein, dass dies zumindest auf zwei verschiedene Arten geschehen kann:

- Matrizenmultiplikation im Sinne der linearen Algebra.
- Elementweises Multiplizieren für numerische Berechnungen.

Tabelle 3.1: Eigenschaften von Arrays: Dimension, Größe, Länge, Anzahl

Bezeichnung	Elemente	Dimension `ndims`	Größe `size`	Länge `length`	Anzahl `numel`
Leeres Array	0	2	`[0 0]`	0	0
Skalar	1	2	`[1 1]`	1	1
Zeilenvektor	3	2	`[1 3]`	3	3
Spaltenvektor	3	2	`[3 1]`	3	3
2-dim Matrix	3×4	2	`[3 4]`	4	12
3-dim Matrix	$3 \times 4 \times 2$	3	`[3 4 2]`	4	24
⋮					

3.2 Eigenschaften von Arrays

Wichtige Eigenschaften von Arrays sind neben ihrem Inhalt,

- ihre Dimension, und
- ihre Größe, entspricht der Anzahl der Elemente in jeder Dimension, und
- ihre Länge, entspricht der maximalen Ausdehnung in einer beliebigen Dimension.

In Tabelle 3.1 kann man erkennen, dass auch leere Arrays, Skalare und Vektoren die Dimension 2 haben. Daran sieht man, dass in MATLAB jede Zahl als zumindest 2-dim Array aufgefasst wird.

3.3 Hilfe für Arrays

Eine genaue Erklärung der einzelnen Befehle in MATLAB erhält man durch Aufruf des Befehls `help` also z.B.: `help ndims`. Man kann auch den Links in diesem Dokument folgen, bzw. erhält man mit `doc ndims` die Hilfe in MATLAB in HTML Format.

MATLAB HELP: `ndims`

```
Number of dimensions.

N = NDIMS(X) returns the number of dimensions in the array X.
The number of dimensions in an array is always greater than
or equal to 2.  Trailing singleton dimensions are ignored.
Put simply, it is LENGTH(SIZE(X)).
```

In Ergänzung dazu lautet die Hilfe für `size`:

MATLAB HELP: `size`

```
Size of matrix.

D = SIZE(X), for M-by-N matrix X, returns the two-element
row vector D = [M, N] containing the number of rows and columns
in the matrix.  For N-D arrays, SIZE(X) returns a 1-by-N
vector of dimension lengths.

[M,N] = SIZE(X) returns the number of rows and columns in
separate output variables. [M1,M2,M3,...,MN] = SIZE(X)
returns the length of the first N dimensions of X.

M = SIZE(X,DIM) returns the length of the dimension specified
by the scalar DIM.  For example, SIZE(X,1) returns the number
of rows.
```

bzw. für `length`:

MATLAB HELP: `length`

```
Length of vector.

LENGTH(X) returns the length of vector X.  It is equivalent
to MAX(SIZE(X)) for non-empty arrays and 0 for empty ones.
```

3.4 Erzeugung von Matrizen

Arrays bzw. Matrizen können auf vielfältige Weise erzeugt werden:

- Explizite Eingabe (3.4.1).
- Erzeugung mit Hilfe der Doppelpunkt Notation (3.4.2).
- Erzeugung mit Hilfe eingebauter Funktionen (3.4.3).
- Laden von einem externen File (3.4.4).
- Selbst geschriebene Funktionen (M-files).

3.4.1 Explizite Eingabe

Die explizite Eingabe einer beliebigen Matrix (hier z.B. eines magisches Quadrats),

$$\begin{bmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{bmatrix}$$

kann auf folgende Weise durchgeführt werden:

```
A = [16,3,2,13; 5,10,11,8; 9,6,7,12; 4,15,14,1]
```

wobei hier eine Zuweisung der Werte auf eine Variable mit dem Namen `A` erfolgt.
Man muss dabei folgende Regeln beachten:

- Die einzelnen Einträge innerhalb einer Zeile (row) werden durch Leerzeichen (blanks) oder bevorzugt durch Beistriche (commas) getrennt.
- Der Strichpunkt (semicolon) schließt eine Zeile ab.
- Die gesamte Liste der Einträge wird in eckige Klammern `[]` gestellt.

3.4.2 Doppelpunkt Notation

Die `Doppelpunktnotation` ist eine der mächtigsten Bestandteile von MATLAB. Sie kann einerseits zur Konstruktion von Vektoren (Tab. 3.2), aber auch zum Zugriff auf Teile von Matrizen (Index, 3.6) verwendet werden.

Tabelle 3.2: Doppelpunkt Notation zur Erzeugung von Vektoren

Op.	Alt.	Befehl	Resultat	Bedingung
`J:K`	`J:1:K`	`colon(J,K)`	`[J, J+1, ..., K]`	`K>=J`
`J:K`	`J:1:K`	`colon(J,K)`	`[]`	`K<J`
`J:D:K`		`colon(J,D,K)`	`[J, J+D, ..., J+m*D]`	`K>=J & D>0`
`J:D:K`		`colon(J,D,K)`	`[J, J+D, ..., J+m*D]`	`K<=J & D<0`
`J:D:K`		`colon(J,D,K)`	`[]`	`K<J & D>0`
`J:D:K`		`colon(J,D,K)`	`[]`	`K>J & D<0`
`J:D:K`		`colon(J,D,K)`	`[]`	`D=0`

Definition: `m = fix((K-J)/D)`, Umwandlung in ganze Zahlen durch Abschneiden.

Leere Arrays: Symbolisiert durch `[]`.

Logisches UND: Verwendetes Symbol &.

MATLAB Beispiel

Der Befehl `colon` bzw. der Operator `:`

Einige gültige und ungültige Beispiele für die Doppelpunkt Notation.

```
X = 1:5
    1     2     3     4     5

X = 1:2:5
    1     3     5

X = 1:-2:5
    Empty matrix: []

X = 5:-2:1
    5     3     1

X = 5:2:1
    Empty matrix: []
```

Tabelle 3.3: MATLAB Befehle zum Erzeugen von Matrizen

`zeros(m)`	Erzeugt eine m×m Nullmatrix
`zeros(m,n)`	Erzeugt eine m×n Nullmatrix
`ones(m)`	Erzeugt eine m×m Matrix mit lauter Einsen
`ones(m,n)`	Erzeugt eine m×n Matrix mit lauter Einsen
`eye(m)`	Erzeugt eine m×m Einheitsmatrix
`eye(m,n)`	Erzeugt eine m×n Einheitsmatrix
`linspace(a,b,n)`	Erzeugt Zeilenvektor mit n äquidistanten Werten von a bis b.
`logspace(a,b,n)`	Erzeugt Zeilenvektor mit n Werten von 10^a bis 10^b mit logarithmisch äquidistantem Abstand.
`rand(m)`	Erzeugt eine m×m Zufallsmatrix (gleichverteilt aus [0,1])
`rand(m,n)`	Erzeugt eine m×n Zufallsmatrix (gleichverteilt aus [0,1])
`randn(m)`	Erzeugt eine m×m Zufallsmatrix (normalverteilt)
`randn(m,n)`	Erzeugt eine m×n Zufallsmatrix (normalverteilt)

Tabelle 3.4: Ergänzende MATLAB Befehle zum Erzeugen von Matrizen

`diag(v,k)`	v ... Vektor, k ... Skalar. Erzeugt eine Matrix mit lauter Nullen, außer auf der k-ten Nebendiagonale, die mit den Werten von v gefüllt wird. k = 0 ist die Hauptdiagonale, k>0 darüber, k<0 darunter. Für k=0 kann man auch diag(v) schreiben.
`diag(m,k)`	m ... Matrix. Extrahiert die k-te Nebendiagonale. (k siehe oben).
`triu(m)`	Extrahiert oberes Dreieck aus der Matrix m.
`triu(m,k)`	Extrahiert Dreieck oberhalb der Nebendiagonale k aus der Matrix m. (k siehe oben).
`tril(m)`	Extrahiert unteres Dreieck aus der Matrix m.
`tril(m,k)`	Extrahiert Dreieck unterhalb der Nebendiagonale k aus der Matrix m. (k siehe oben).
`blkdiag(a,b,...)`	Erzeugt eine blockdiagonale Matrix. a,b, ... sind Matrizen.
`repmat(a,m,n)`	Erzeugt aus einer Matrix a eine neue Matrix durch Replikation in Zeilenrichtung (m-mal) und Spaltenrichtung (n-mal).

3.4.3 Interne Befehle zur Erzeugen von Matrizen

Es gibt eine Reihe von Befehlen zur einfachen Erzeugung von Matrizen.

Mit Hilfe des Befehls `[z,s]=meshgrid(v1,v2)` ist es sehr leicht zwei gleich große Matrizen zu erzeugen. Sind die beiden Vektoren `v1` und `v2` z.B. die Vektoren `1:n` und `1:m`, dann ergeben sich folgende Matrizen:

$$z = \begin{bmatrix} 1 & 2 & 3 & \dots & n \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \dots & n \end{bmatrix}, \; s = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 2 & 2 & 2 & \dots & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m & m & m & \dots & m \end{bmatrix}. \tag{3.1}$$

Die Variablen `m` und `n` müssen dabei vorher definiert werden. Analog kann das natürlich mit allen anderen Vektoren ausgeführt werden. Die so erhaltenen Matrizen eignen sich bestens zum Kombinieren.

Mit dem Befehl `v = z + 100*s` erhält man sofort folgende Matrix:

$$v = \begin{bmatrix} 101 & 102 & 103 & 104 & 105 & \dots \\ 201 & 202 & 203 & 204 & 205 & \dots \\ 301 & 302 & 303 & 304 & 305 & \dots \\ 401 & 402 & 403 & 404 & 405 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{3.2}$$

3.4.4 Lesen und Schreiben von Daten

Neben komplexen Befehlen zum Schreiben und Lesen von Daten und dem Umgang mit externen Datenfiles, gibt es zum Lesen geordneter Strukturen den einfachen Befehl `load`. Er funktioniert nur, wenn die Daten in Tabellenform ohne fehlende Einträge oder Kommentarzeilen gespeichert sind.

Die Form des Aufrufs ist `D = load('d.dat')`, wobei hier `'d.dat'` für eine Zeichenkette mit dem Filenamen steht. Das Gegenstück zum Speichern von lesbaren Daten ist `save`. Dieser Befehl wird in folgender Form verwendet: `save('d.dat','D','-ascii')`

Eine detailierte Beschreibung von Schreibe- und Leseroutinen folgt in einem späteren Kapitel.

3.5 Veränderung und Auswertung von Matrizen

Viele Befehle haben als Inputparameter eine Matrix und liefern eine (im Allgemeinen nicht unbedingt gleich große) Matrix zurück. (Zur Erinnerung: Spalten- bzw. Zeilenvektoren werden ebenfalls als Matrizen angesehen).

Beispiele dafür sind das Bilden von Summen oder Produkten, oder das Transponieren und Konjugieren. Im Folgenden wurden dafür einige einfache Beispiele zusammengestellt.

Der numerische Inhalt von Matrizen muss nicht nur aus reellen Zahlen bestehen, sondern kann auch komplexe Werte enthalten. Dafür ist keine spezielle Deklaration notwendig, MATLAB führt diese automatisch beim ersten Auftreten von komplexen Elementen in einer Matrix durch.

Die Variablen `i` oder auch `j` werden als imaginäre Einheit $i = \sqrt{-1}$ verwendet, und sollen daher sonst nicht verwendet werden. MATLAB hat keinen effektiven Schutz vor dem Überschreiben von wichtigen Variablen. Die beiden Befehle `i=1` und `j=1` legen die Fähigkeit von MATLAB lahm, mit komplexen Zahlen zu rechnen.

MATLAB Beispiel

Einige Befehle stehen in MATLAB zur Verfügung, um Matrizen zu kippen bzw. zu drehen. Außerdem gibt es noch `FLIPDIM(X,DIM)`, für Kippen entlang der Dimension DIM.

`FLIPLR` Flip matrix in left/right direction.
`FLIPLR(X)` returns X with row preserved and columns flipped in the left/right direction.

`FLIPUD` Flip matrix in up/down direction.
`FLIPUD(X)` returns X with columns preserved and rows flipped in the up/down direction.

`ROT90` Rotate matrix 90 degrees.
`ROT90(X)` is the 90 degree counterclockwise rotation of matrix X. `ROT90(X,K)` is the K×90 degree rotation of X, K = ±1, ±2, ...

```
X = [1 2 3; 4 5 6]
     1     2     3
     4     5     6

Y=fliplr(X)
     3     2     1
     6     5     4

Y=flipud(X)
     4     5     6
     1     2     3

Y=rot90(X)
     3     6
     2     5
     1     4
```

MATLAB Beispiel

Drei Befehle stehen in MATLAB zur Verfügung, um transponierte, konjugiert komplex transponierte oder konjugiert komplexe Matrizen zu berechnen.

TRANSPOSE is the non-conjugate transpose.

Operator form: X.' is the transpose of X.

CTRANSPOSE is the complex conjugate transpose.

Operator form: X' is the complex conjugate transpose of X.

CONJ is the complex conjugate of X.
For a complex X,
CONJ(X) = REAL(X) - i*IMAG(X).

```
X = [1+i 2+i 3+i; 4+i 5+i 6+i]
      1 + i   2 + i   3 + i
      4 + i   5 + i   6 + i

Y=transpose(X)
      1 + i   4 + i
      2 + i   5 + i
      3 + i   6 + i

Y=ctranspose(X)
      1 - i   4 - i
      2 - i   5 - i
      3 - i   6 - i

Y=conj(X)
      1 - i   2 - i   3 - i
      4 - i   5 - i   6 - i
```

MATLAB Beispiel

Summation und kummulative Summation in Matrizen.

`SUM` Sum of elements.
For vectors, `SUM(X)` is the sum of the elements of X. For matrices, `SUM(X)` is a row vector with the sum over each column. For N-D arrays, `SUM(X)` operates along the first non-singleton dimension.

`SUM(X,DIM)` sums along the dimension DIM.

`CUMSUM` Cumulative sum of elements. For vectors, `CUMSUM(X)` is a vector containing the cumulative sum of the elements of X. For matrices, `CUMSUM(X)` is a matrix the same size as X containing the cumulative sums over each column. For N-D arrays, `CUMSUM(X)` operates along the first non-singleton dimension.

`CUMSUM(X,DIM)` works along the dimension DIM.

The first non-singleton dimension is the first dimension which size is greater than one.

```
X = [0 1 2; 3 4 5]
     0     1     2
     3     4     5

Y=sum(X)
     3     5     7

Y=sum(X,2)
     3
    12

Y=cumsum(X)
     0     1     2
     3     5     7

Y=cumsum(X,2)
     0     1     3
     3     7    12
```

MATLAB Beispiel

Multiplikation und kummulative Multiplikation in Matrizen.

`PROD` Product of elements.
For vectors, `PROD(X)` is the product of the elements of X. For matrices, `PROD(X)` is a row vector with the product over each column. For N-D arrays, `PROD(X)` operates along the first non-singleton dimension.

`PROD(X,DIM)` works along the dimension DIM.

`CUMPROD` Cumulative product of elements. For vectors, `CUMPROD(X)` is a vector containing the cumulative product of the elements of X. For matrices, `CUMPROD(X)` is a matrix the same size as X containing the cumulative product over each column. For N-D arrays, `CUMPROD(X)` operates along the first non-singleton dimension.
`CUMPROD(X,DIM)` works along the dimension DIM.

```
X = [0 1 2; 3 4 5]
     0     1     2
     3     4     5

Y=prod(X)
     0     4    10

Y=prod(X,2)
     0
    60

Y=cumprod(X)
     0     1     2
     0     4    10

Y=cumprod(X,2)
     0     0     0
     3    12    60
```

Alle Befehle in Matlab, bei denen die Richtung innerhalb der Matrix von Bedeutung ist, wie z.B. der Befehl `sum`, folgen folgenden Regeln:

1. Ist eine Richtung vorgegeben, `sum(X,2)`, erfolgt die Operation in Richtung dieser Dimension.
2. Ist keine Richtung vorgegeben, erfolgt die Summation in Richtung der ersten Dimension, die ungleich eins ist (non-singleton dimension). Das heißt, dass sowohl in einem Spaltenvektor (`size(X)` z.B. `[3 1]`), als auch in einem Zeilenvektor (`size(X)` z.B. `[1 3]`) über alle Elemente summiert wird.

Befehle können in MATLAB beliebig geschachtelt werden, solange die Syntax für jeden einzelnen Befehl korrekt ist. So kann man z.B. die Summe über die Diagonale bzw. die zweite Diagonale (links unten bis rechts oben) einer Matrix mit folgenden Befehlen berechnen:

Summe der Diagonalelemente der Matrix X:

```
S_D = sum(diag(X))
```

Summe der Elemente in der zweiten Diagonale der Matrix X:

```
S_ND = sum(diag(fliplr(X)))
```

Die große Vielzahl von verfügbaren Befehlen und die Möglichkeit der Schachtelung führt dazu, dass sehr mächtige Programme in sehr kompakter Form geschrieben werden können.

3.6 Zugriff auf Teile von Matrizen, Indizierung

Sehr häufig ist es wichtig, auf bestimmte Teile einer Matrix in Abhängigkeit von ihrer Position in der Matrix zuzugreifen. Dazu braucht man die sogenannte Indizierung, die hier am Beispiel einer 2-dim Matrix erläutert werden soll. Bei höher dimensionalen Matrizen ist das Konzept analog anzuwenden.

In MATLAB bezieht sich der Befehl `A(i,j)` auf das Element a_{ij} der Matrix A. Diese Bezeichnung ist praktisch in allen Programmiersprachen üblich. MATLAB bietet jedoch einen viel weitergehenden Aspekt der Indizierung, der es auf einfache Weise erlaubt auf bestimmte Regionen innerhalb einer Matrix zuzugreifen. Diese Eigenschaft macht die Matrix Manipulation einfacher als in vielen anderen Programmiersprachen. Außerdem bietet es eine einfache Möglichkeit die "vektorisierte" Natur von Berechnungen in MATLAB zu benutzen.

Die meisten Programme werden dadurch viel lesbarer und übersichtlicher, da man sich eine große Anzahl von Schleifen (und damit auch eine große Anzahl von Fehlerquellen) sparen kann.

In der Folge wird nun auf die verschiedenen Möglichkeiten der Indizierung eingegangen. In Tabelle 3.5 werden die einzelnen Regeln erläutert, und in 3.6 die Zuweisung von Werten gezeigt, und in 3.7 der Zugriff auf bestimmte Regionen gezeigt.

Die Umrechnung zwischen dem linearen Index und mehrfachen Indices erfolgt mit den Befehlen `ind2sub` und `sub2ind`:

Mehrfacher Index von linearem Index: `[JI,MI] = ind2sub(size(X),I)`

Linearer Index von mehrfachem Index: `[I] = sub2ind(size(X),JI,MI)`

In beiden Befehlen muss natürlich die Größe, `size(X)`, angegeben werden, da nur mit diesem Wissen der Zusammenhang zwischen den Indices eineindeutig ist. Wie bei dem Befehl `sum` folgt der lineare Index zuerst der ersten, dann der zweiten, dann der nächsten Dimension. Der Zusammenhang sollte aus folgender Darstellung klar werden,

$$\begin{bmatrix} (1,1) & (1,2) & (1,3) & (1,4) \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & (3,2) & (3,3) & (3,4) \end{bmatrix} \equiv \begin{bmatrix} (1) & (4) & (7) & (10) \\ (2) & (5) & (8) & (11) \\ (3) & (6) & (9) & (12) \end{bmatrix} . \tag{3.3}$$

Tabelle 3.5: Indizierung von Arrays

Index	Alternative	Zeilen	Spalten	Resultat
INDIZIERUNG MIT ZWEI INDICES				
`X(J,M)`		`J`	`M`	Skalar
`X(J,:)`	`X(J,1:end)`	`J`	`ALLE`	Zeilenvektor
`X(:,M)`	`X(1:end,M)`	`ALLE`	`M`	Spaltenvektor
`X(:,:)`	`X(1:end,1:end)`	`ALLE`	`ALLE`	2-D Array
`X(J:K,M)`		`J:K`	`M`	Spaltenvektor
`X(J:D:K,M)`		`J:D:K`	`M`	Spaltenvektor
`X(J:K,M:N)`		`J:K`	`M:N`	2-D Array
INDIZIERUNG MIT EINEM INDEX (LINEAR)				
`X(:)`		`ALLE`	`ALLE`	Spaltenvektor
`X(I)`		`JI`	`MI`	Skalar
`X(I:H)`		`JI:JH`	`MI:MH`	Zeilenvektor

Tabelle 3.6: Zuweisung von Werten an bestimmten Positionen eines Arrays

`X`	`X(3,2)=1`	`X(:,2)=1`
0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 1 0 0	0 1 0 0 0 1 0 0 0 1 0 0
`X(2,:)=1`	`X(:,:)=1`	`X(:)=1`
0 0 0 0 1 1 1 1 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1
`X(:,1:2:4)=1`	`X(1:2:3,:)=1`	`X(1:2:3,1:2:4)=1`
1 0 1 0 1 0 1 0 1 0 1 0	1 1 1 1 0 0 0 0 1 1 1 1	1 0 1 0 0 0 0 0 1 0 1 0
`X(7:10)=1`	`X(1:2,3)=1`	`X(2,1:3)=1`
0 0 1 1 0 0 1 0 0 0 1 0	0 0 1 0 0 0 1 0 0 0 0 0	0 0 0 0 1 1 1 0 0 0 0 0

Tabelle 3.7: Zugriff auf bestimmte Positionen eines Arrays

`X`	`X(3,2)`	`X(:,2)`
$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$	10	$\begin{matrix} 2 \\ 6 \\ 10 \end{matrix}$
`X(2,:)`	`X(:,:)`	`X(:)`
$\begin{matrix} 5 & 6 & 7 & 8 \end{matrix}$	$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$	$\begin{matrix} 1 \\ 5 \\ \vdots \\ 8 \\ 12 \end{matrix}$
`X(:,1:2:4)`	`X(1:2:3,:)`	`X(1:2:3,1:2:4)`
$\begin{matrix} 1 & 3 \\ 5 & 7 \\ 9 & 11 \end{matrix}$	$\begin{matrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{matrix}$	$\begin{matrix} 1 & 3 \\ 9 & 11 \end{matrix}$
`X(7:10)`	`X(1:2,3)`	`X(2,1:3)`
$\begin{matrix} 3 & 7 & 11 & 4 \end{matrix}$	$\begin{matrix} 3 \\ 7 \end{matrix}$	$\begin{matrix} 5 & 6 & 7 \end{matrix}$

Da mit Hilfe der Doppelpunkt Notation ja eigentlich Vektoren als Indices erzeugt werden (3.4.2), ist natürlich auch folgende Schreibweise erlaubt:

- `X([1 2],[2 3])` äquivalent zu `X(1:2,2:3)`
- `X([1 3],[2 4])` äquivalent zu `X(1:2:3,2:2:4)`

Eine wichtige Rolle spielt auch das Keyword `end`, das im richtigen Kontext die entsprechende Größe angibt. Damit ist es nicht notwendig bei der Indizierung die Größe der Arrays zu kennen:

- `X(1:2:end,3)` für die dritte Spalte jeder 2.ten Zeile.
- `X(2:end-1,2:end-1)` für die 2.te bis vorletzte Zeile bzw. Spalte.

3.6.1 Logische Indizierung

In Ergänzung zur normalen Indizierung erlaubt MATLAB auch die sogenannte logische Indizierung mit Arrays die nur die Werte 1 (entspricht TRUE) bzw. 0 (entspricht FALSE) enthalten. Dadurch ist auch der Zugriff auf völlig ungeordnete Bereiche möglich (Tab. 3.8).

Wichtig dabei ist Folgendes:

- Das Array `L` muss die gleiche Größe wie das Array `X` haben.
- Das Array `L` muss ein logisches Array sein, das entstanden ist durch
  - logische Operationen (`and`, `or`, `xor`, `not`),
  - Vergleichsoperationen (z.B.: `<`),
  - durch Verwendung des Befehls `logical(Y)`, wodurch ein numerisches Array in ein logisches umgewandelt wird.
- Ein logisches Array darf nicht nur die Werte 0 und 1 beinhalten, MATLAB folgt der Konvention, dass alle Zahlen die ungleich 0 sind als TRUE gelten.
- Wegen der möglicherweise ungeordneten Anordung der Zielelemente in der Matrix, geht die Form verloren. Das Ergebnis liegt immer in Form eines Spaltenvektors vor, außer beide Matrizen sind ein Zeilenvektor, dann bleibt ein Zeilenvektor erhalten.
- Der Verlust der Form spielt natürlich bei einer Zuweisung von Werten auf diese Positionen keine Rolle, die Form der Matrix bleibt dabei erhalten.

Tabelle 3.8: Zugriff mit Hilfe logischer Indizierung

`X`	`L`	`X(L)`
1 2 3 4 5 6 7 8 9 10 11 12	0 0 1 0 1 0 0 0 0 0 0 1	5 3 12
`X`	`L`	`X(L)=0`
1 2 3 4 5 6 7 8 9 10 11 12	0 0 1 0 1 0 0 0 0 0 0 1	1 2 0 4 0 6 7 8 9 10 11 0
`X`	`L`	`X(L)`
1 2 3 4 5 6 7 8 9 10 11 12	1 0 0 0 0 1 0 0 0 0 1 0	1 6 11
`X`	`L`	`X(L)=0`
1 2 3 4 5 6 7 8 9 10 11 12	1 0 0 0 0 1 0 0 0 0 1 0	0 2 3 4 5 0 7 8 9 10 0 12

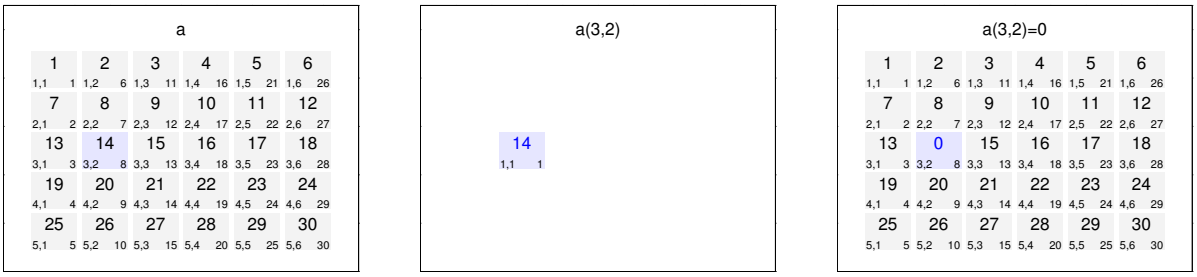
- Bei jeder Zuweisung muss entweder die Anzahl der Werte gleich sein wie die Anzahl der ausgewählten Positionen, oder ein Skalar wird auf eine beliebige Anzahl von Positionen zugewiesen.
- Ist man nur an den Positionionen interessiert, kann man mit `I = find(L)` die linearen Indices, bzw. mit `[m,n] = find(L)` die 2-dim Indices erhalten.
- Details über Vergleichsoperatoren und logische Operatoren finden sich in den Abschnitten 4.3 und 4.2.

3.6.2 Beispiele zur Indizierung

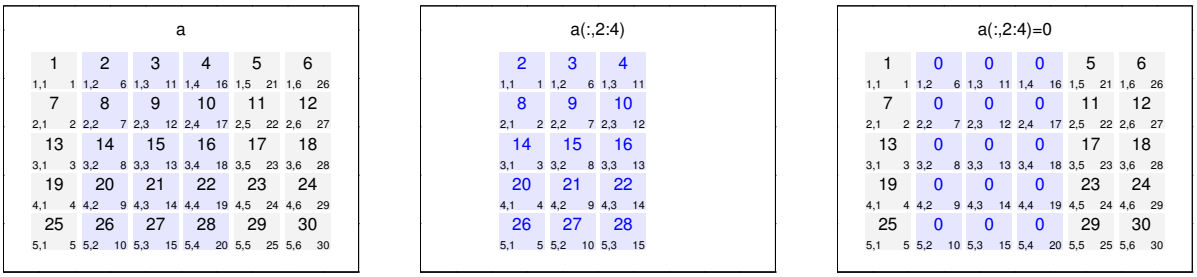
Die vorliegenden Beispiele demonstrieren die Indizierung in MATLAB an Hand von 2-dimensionalen Matrizen. Jedes Element enthält dabei in der unteren linken Ecke den 2-D Index und in der rechten unteren Ecke den linearen Index. Erfolgt eine Zuweisung, bleibt die Form der Matrix erhalten, erfolgt jedoch keine Zuweisung werden die entsprechenden Elemente ausgeblendet. Ändert sich dabei die Form in einen Zeilen- oder Spaltenvektor, wird in der linken unteren Ecke `Z` oder `S` ausgegeben. Der lineare Index in der rechten unteren Ecke gibt dabei die Position im Vektor an und die Form der Darstellung hat keine Bedeutung mehr.

3.6.2.1 Zweidimensionale Indizierung

Zugriff auf Einzelelemente.



Zugriff auf alle Zeilen in mehreren Spalten.



Zugriff auf Spalten und Zeilen unter Verwendung des Keywortes end.

a

1	2	3	4	5	6
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	8	9	10	11	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	26	27	28	29	30
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

a(1:2,1:2:end)

1	3	5
1,1 1	1,2 3	1,3 5
7	9	11
2,1 2	2,2 4	2,3 6

a(1:2,1:2:end)=[50,51,52;53,54,55]

50	2	51	4	52	6
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
53	8	54	10	55	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	26	27	28	29	30
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

Zugriff auf Spalten und Zeilen unter Verwendung des Keywortes end.

a

1	2	3	4	5	6
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	8	9	10	11	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	26	27	28	29	30
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

a(1:2:end,1:end−1:end)

1		6
1,1 1		1,2 4
13		18
2,1 2		2,2 5
25		30
3,1 3		3,2 6

a(1:2:end,1:end−1:end)=[1,1;2,2;3,3]

1	2	3	4	5	1
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	8	9	10	11	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
2	14	15	16	17	2
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
3	26	27	28	29	3
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

Zugriff auf die gesamte Matrix.

a

1	2	3	4	5	6
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	8	9	10	11	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	26	27	28	29	30
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

a(:,:)

1	2	3	4	5	6
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	8	9	10	11	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	26	27	28	29	30
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

a(:,:)=0

0	0	0	0	0	0
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
0	0	0	0	0	0
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
0	0	0	0	0	0
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
0	0	0	0	0	0
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
0	0	0	0	0	0
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

Zugriff auf Spalten und Zeilen mit Vektoren.

a

1	2	3	4	5	6
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	8	9	10	11	12
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	26	27	28	29	30
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

a([1,2,5],[2,3,6])

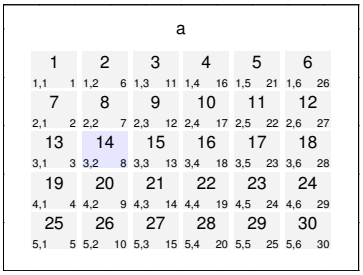
2	3	6
1,1 1	1,2 4	1,3 7
8	9	12
2,1 2	2,2 5	2,3 8
26	27	30
3,1 3	3,2 6	3,3 9

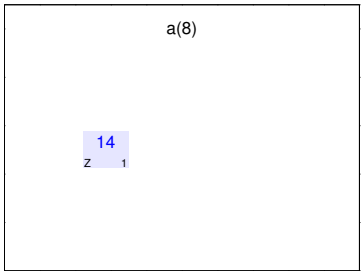
a([1,2,5],[2,3,6])=0

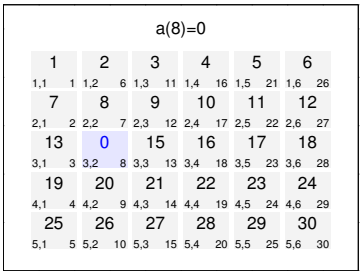
1	0	0	4	5	0
1,1 1	1,2 6	1,3 11	1,4 16	1,5 21	1,6 26
7	0	0	10	11	0
2,1 2	2,2 7	2,3 12	2,4 17	2,5 22	2,6 27
13	14	15	16	17	18
3,1 3	3,2 8	3,3 13	3,4 18	3,5 23	3,6 28
19	20	21	22	23	24
4,1 4	4,2 9	4,3 14	4,4 19	4,5 24	4,6 29
25	0	0	28	29	0
5,1 5	5,2 10	5,3 15	5,4 20	5,5 25	5,6 30

3.6.2.2 Lineare Indizierung

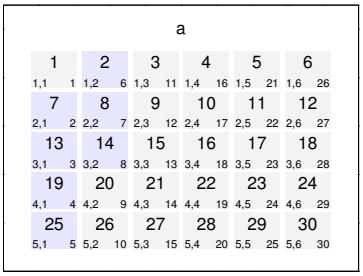
Zugriff auf Einzelelemente.

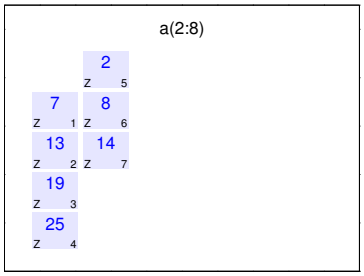


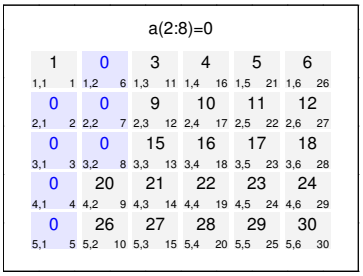




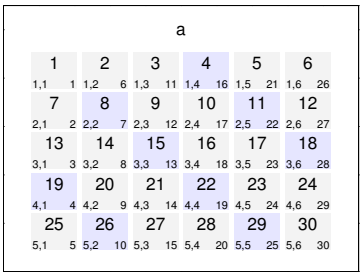
Zugriff auf zusammenhängende Bereiche.

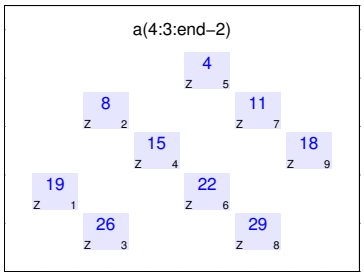


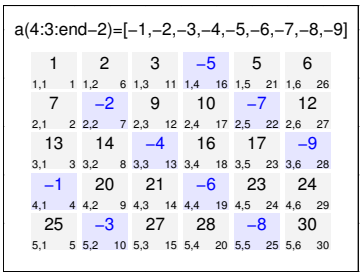




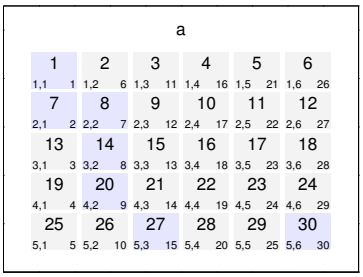
Zugriff auf nichtzusammenhängende Bereiche.

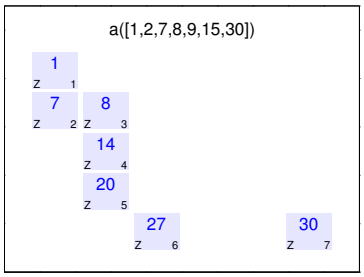


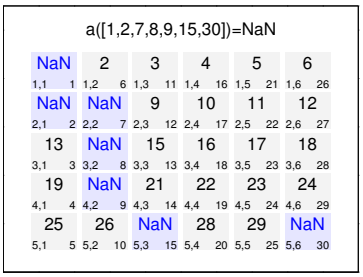




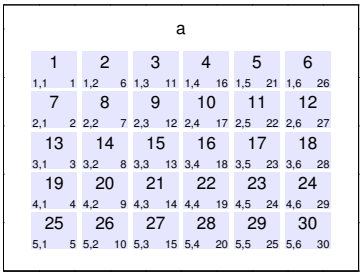
Zugriff mit Hilfe eines beliebigen Vektors.

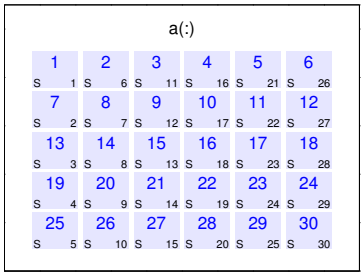


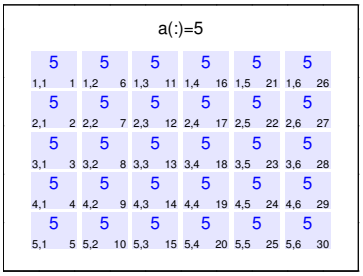




Zugriff auf gesamte Matrix.

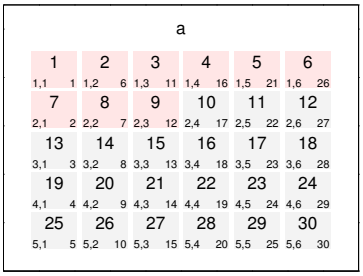


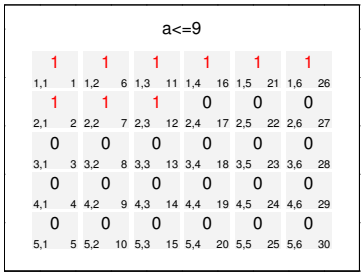


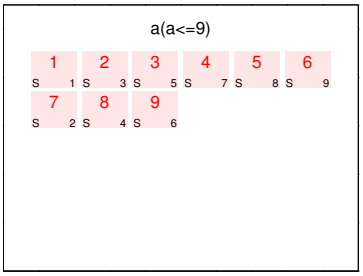


3.6.2.3 Logische Indizierung

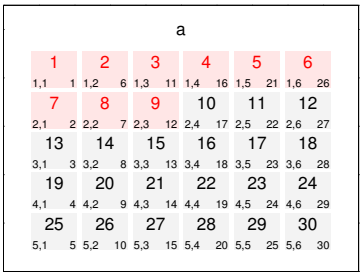
Zugriff auf Teile der Matrix.

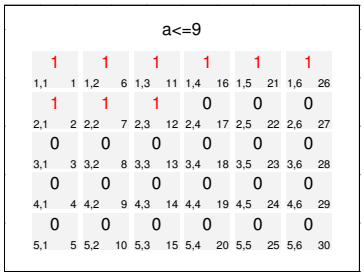


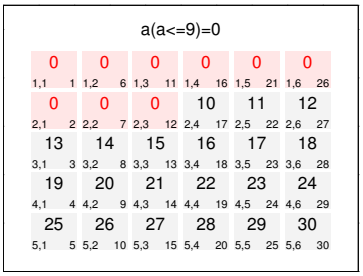




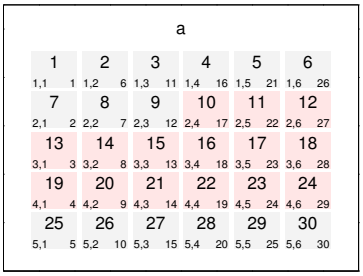
Veränderung von Teilen der Matrix.

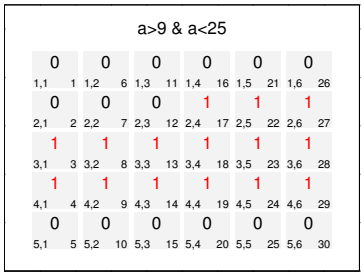


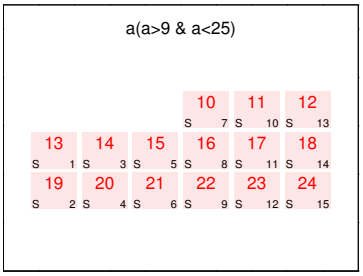




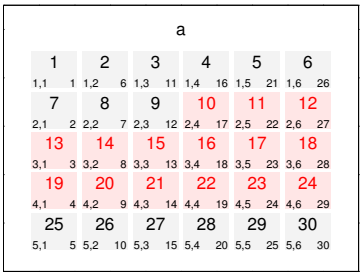
Zugriff auf Teile der Matrix.

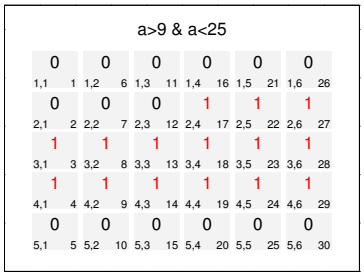


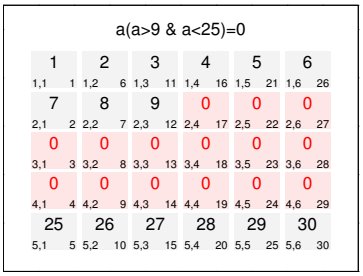




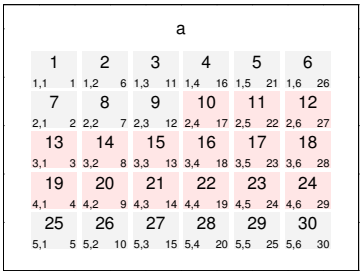
Veränderung von Teilen der Matrix.

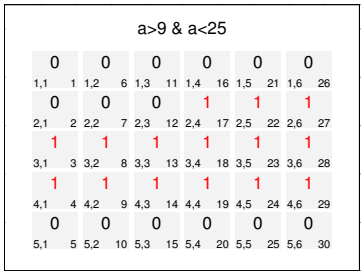


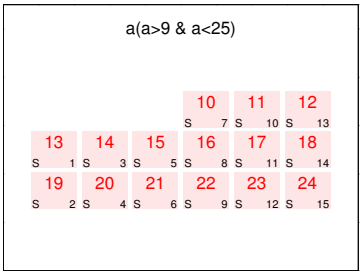




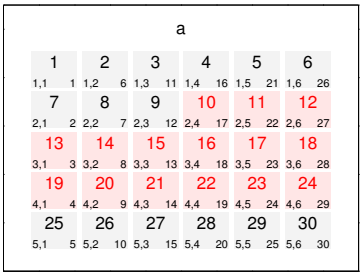
Zugriff auf Teile der Matrix.

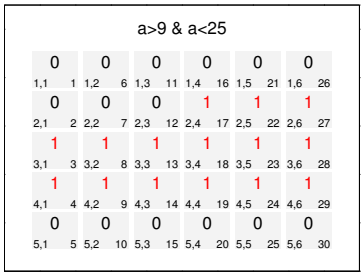


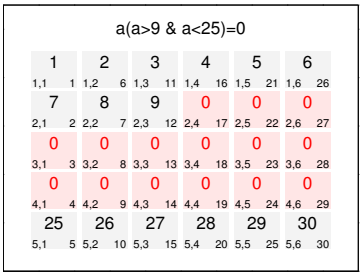




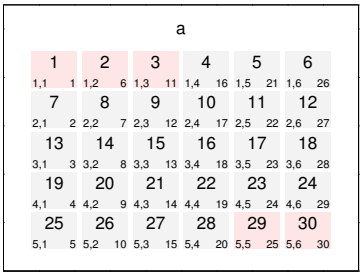
Veränderung von Teilen der Matrix.

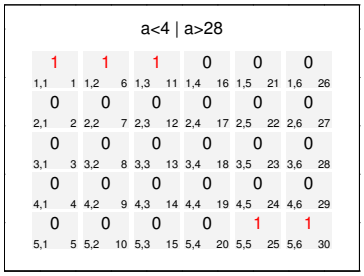


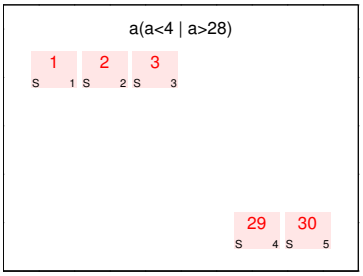




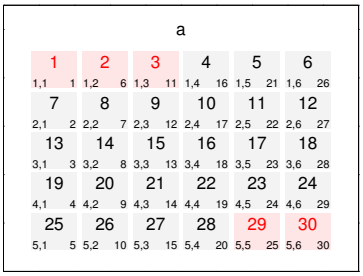
Zugriff auf Teile der Matrix.

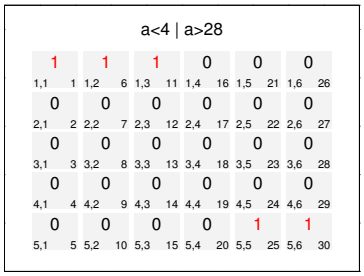


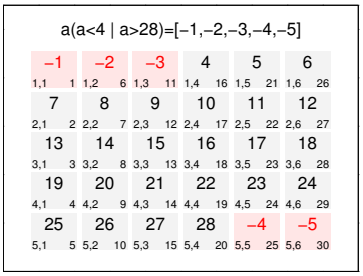




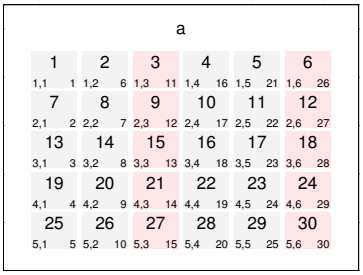
Veränderung von Teilen der Matrix.

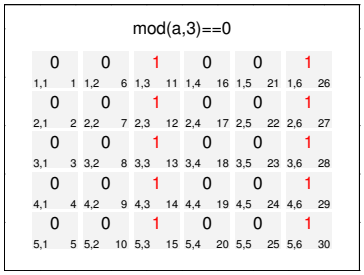


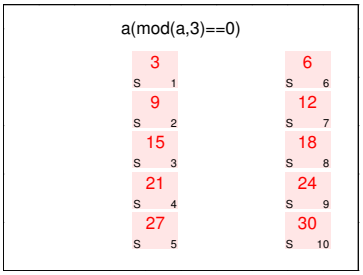




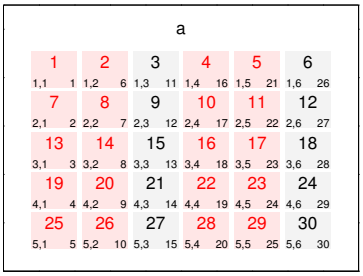
Zugriff auf jene Teile der Matrix, die durch drei teilbar sind.

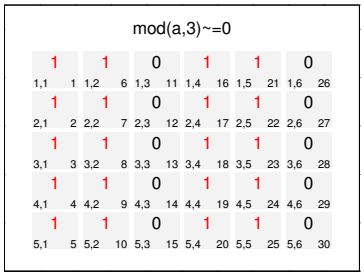


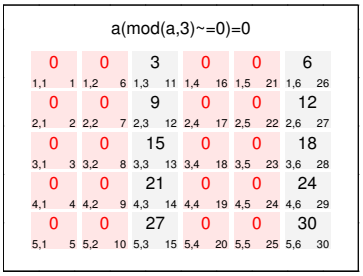




Veränderung von Teilen der Matrix, die nicht durch drei teilbar sind.







3.7 Zusammenfügen von Matrizen

Für das Zusammenfügen von Matrizen zu einer Einheit stehen die Befehle `cat`, `vertcat` (untereinander) und `horzcat` (nebeneinander) zur Verfügung. Der Befehl `cat(DIM,A,B)` fügt die beiden Matrizen entlang der Dimenssion `DIM` zusammen. Alle anderen Dimension müssen natürlich übereinstimmen.

BEFEHL	ALTERNATIVE	KURZFORM
`cat(1,A,B)`	`vertcat(A,B)`	`[A;B]`
`cat(2,A,B)`	`horzcat(A,B)`	`[A,B]`
`cat(3,A,B)`		
`cat(1,A,B,C,...)`	`vertcat(A,B,C,...)`	`[A;B;C;...]`
`cat(2,A,B,C,...)`	`horzcat(A,B,C,...)`	`[A,B,C,...]`

3.8 Initialisieren, Löschen und Erweitern

Eine Initialisierung bzw. Deklaration von Matrizen in MATLAB ist nicht unbedingt notwendig. Bei Matrizen kann jederzeit ihr Inhalt, ihre Größe oder ihr Typ verändert werden. Trotzdem ist es meist sinnvoll, Matrizen mit Typ und Größe zu initialisieren, wie sie später benötigt werden.

Vor allem bei großen Matrizen und bei sogenannten dynamischen Matrizen, dass sind solche, deren Inhalt sich in Schleifen dauert ändert, ist dies ein wichtiger Schritt. Beim Initialisieren wird ein kontinuierlicher Bereich im Computerspeicher angelegt (alloziert), auf den rasch zugegriffen werden kann. Ändert sich der Typ oder die Größe muss neu alloziert werden, was jedesmal Zeit kostet.

Zum Initialisieren bietet sich der Befehl `zeros(m,n)` an. Benötigt man eine Matrix, die gleich groß wie eine bestehende Matrix `X` sein soll, kann man den Befehl auch so `zeros(size(X))` schreiben.

3.9 Umformen von Matrizen

Zum Umformen von Matrizen steht im Wesentlichen der Befehl `reshape` zur Verfügung.

Der Befehl `Y=reshape(X,SIZ)` liefert ein Array mit den gleichen Werten aber der Größe `SIZ`. Natürlich muss `prod(SIZ)` mit `prod(SIZE(X))` übereinstimmen (gleiche Anzahl von Elementen), sonst meldet MATLAB einen Fehler.

Der Befehl `reshape` kann auf zwei verschiedene Weisen geschrieben werden:

- `reshape(X, M, N, P, ...)`

- `reshape(X, [M N P ...])`

Die zweite Form eignet sich bestens um einen Vektor einzusetzen, der automatisch z.B. mit `size` erhalten wurde.

Das Löschen von Zeilen oder Spalten kann man erreichen, indem man ganzen Zeilen oder Spalten den Wert des leeren Arrays `[]` zuweist. Z.b. löscht der Befehl `a(end-1:end,:)=[]` die letzten beiden Zeilen der Matrix `a`.

Kapitel 4

Operatoren

4.1 Arithmetische Operatoren

4.1.1 Arithmetische Operatoren für Skalare

Die vordefinierten Operatoren auf skalaren double-Ausdrücken sind in Tabelle 4.1 zusammengefaßt. Diese Operatoren sind eigentlich Matrixoperatoren, deren genaue Behandlung in 5 folgt. Der Grund dafür liegt darin, dass skalare Größen auch als Matrizen mit nur einem Element aufgefasst weren können. Damit bleibt hier die übliche Notation mit `*` und `/` erhalten.

Operatoren haben Prioritäten, die die Abarbeitung bestimmen. Operationen mit höherer Priorität werden zuerst ausgeführt.

Die Reihenfolge der Auswertung eines Ausdrucks kann durch Klammerung beeinflußt werden. In Klammern eingeschlossene (Teil-) Ausdrücke haben die höchste Priorität, d.h., sie werden auf jeden Fall zuerst ausgewertet. Bei verschachtelten Klammern werden die Ausdrücke im jeweils innersten Klammerpaar zuerst berechnet. Zur Klammerung verwendet MATLAB die sogenannten runden Klammern `()`.

Kommen in einem Ausdruck mehrere aufeinanderfolgende Verknüpfungen durch Operatoren mit gleicher Priorität vor, so werden sie von links nach rechts abgearbeitet, sofern nicht Klammern vorhanden sind, die etwas anderes vorschreiben; dies ist vor allem dann zu beachten, wenn nicht-assoziative Operatoren gleicher Priorität hintereinander folgen.

Operatoren können nur auf bereits definierte Variablen angewandt werden. Sie liegen immer in Form von Operatoren (`+`) oder in Form von Befehlen (`plus`) vor.

Tabelle 4.1: Skalare Operationen; a und b sind skalare Variablen

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	MATH	PRIORITÄT
^	a^b	mpower(a,b)	Exponentiation	a^b	4
+	+a	uplus(a)	Unitäres Plus	$+a$	3
-	-a	uminus(a)	Negation	$-a$	3
*	a*b	mtimes(a,b)	Multiplikation	ab	2
/	a/b	mrdivide(a,b)	Division	a/b	2
\	a\b	mldivide(a,b)	Linksdivision	b/a	2
+	a+b	plus(a,b)	Addition	$a+b$	1
-	a-b	minus(a,b)	Subtraktion	$a-b$	1

4.1.2 Arithmetische Operatoren für Arrays

Eine herausragende Eigenschaft von MATLAB ist die einfache Möglichkeit der Verarbeitung ganzer Felder durch eine einzige Anweisung. Ähnlich wie in modernen Programmiersprachen Operatoren überladen werden können, lassen sich die meisten Operatoren und vordefinierten Funktionen in MATLAB ohne Notationsunterschied auf (ein- oder mehrdimensionale) Felder anwenden. Tabelle 4.2 enthält die vordefinierten Operatoren für Arrays am Beispiel von Zeilenvektoren. Die Anwendung auf mehrdimensionale Felder erfolgt analog.

Die hier vorgestellten Operatoren, die mit einem Punkt beginnen, werden komponentenweise auf Felder übertragen. Andere Operatoren haben unter Umständen bei Feldern eine andere Bedeutung.

Bei Anwendung auf Skalare haben sie natürlich die gleiche Bedeutung wie die Operatoren in 4.1. In diesem Fall ist also das Resultat von z.B. * und .* das selbe, da Skalare Matrizen mit einem Element sind. Bei + und - erübrigt sich eine Unterscheidung der Bedeutung überhaupt, was zur Folge hat, dass es keine .+ und .- Operatoren gibt.

Durch den Einsatz von Vektoroperatoren kann auf die Verwendung von Schleifen (wie sie etwa in C oder FORTRAN notwendig wären) sehr oft verzichtet werden, was die Lesbarkeit von MATLAB-Programmen fördert.

In MATLAB werden die gleichen Operatoren verwendet, um Vektoren oder allgemein Arrays mit Skalarausdrücken zu verknüpfen. In Tabelle 4.3 findet man die vordefinierten Operatoren zur komponentenweisen Verknüpfung von Feldern und Skalaren.

In 4.3 kommen in einigen wenigen Fällen die Array-Operatoren mit Punkten und die Matrix-Operatoren gleichwertig vor, da sie zum selben Ergebnis führen. Eine genaue Behandlung der Matrix-Operatoren im Sinne der linearen Algebra erfolgt in 5.

Tabelle 4.2: Array-Array Operationen; a und b sind Felder der gleichen Größe, in diesem Beispiel Zeilenvektoren der Länge n.

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	PRIO.
.^	a.^b	power(a,b)	$[a_1^{b_1}\ a_2^{b_2}\ \ldots\ a_n^{b_n}]$	4
.*	a.*b	times(a,b)	$[a_1 b_1\ a_2 b_2\ \ldots\ a_n b_n]$	2
./	a./b	rdivide(a,b)	$[a_1/b_1\ a_2/b_2\ \ldots\ a_n/b_n]$	2
.\	a.\b	ldivide(a,b)	$[b_1/a_1\ b_2/a_2\ \ldots\ b_n/a_n]$	2
+	a+b	plus(a,b)	$[a_1+b_1\ a_2+b_2\ \ldots\ a_n+b_n]$	1
-	a-b	minus(a,b)	$[a_1-b_1\ a_2-b_2\ \ldots\ a_n-b_n]$	1

Tabelle 4.3: Skalar-Array Operationen; a ist in diesem Beispiel ein Zeilenvektor der Länge n und c ist ein Skalar.

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	PRIO.
.^	a.^c	power(a,c)	$[a_1^c\ a_2^c\ \ldots\ a_n^c]$	4
.^	c.^a	power(c,a)	$[c^{a_1}\ c^{a_2}\ \ldots\ c^{a_n}]$	4
.*	a.*c	times(a,c)	$[a_1 c\ a_2 c\ \ldots\ a_n c]$	2
./	a./c	rdivide(a,c)	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2
./	c./a	rdivide(c,a)	$[c/a_1\ c/a_2\ \ldots\ c/a_n]$	2
.\	a.\c	ldivide(a,c)	$[c/a_1\ c/a_2\ \ldots\ c/a_n]$	2
.\	c.\a	ldivide(c,a)	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2
+	a+c	plus(a,c)	$[a_1+c\ a_2+c\ \ldots\ a_n+c]$	1
-	a-c	minus(a,c)	$[a_1-c\ a_2-c\ \ldots\ a_n-c]$	1
*	a*c	mtimes(a,c)	$[a_1 c\ a_2 c\ \ldots\ a_n c]$	2
/	a/c	mrdivide(a,c)	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2
\	c\a	mldivide(c,a)	$[a_1/c\ a_2/c\ \ldots\ a_n/c]$	2

Tabelle 4.4: Vergleichsoperatoren

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	MATH
<	a<b	lt(a,b)	kleiner als	$a < b$
<=	a<=b	le(a,b)	kleiner oder gleich	$a \leq b$
>	a>b	gt(a,b)	größer als	$a > b$
>=	a>=b	ge(a,b)	größer oder gleich	$a \geq b$
==	a==b	eq(a,b)	gleich	$a = b$
~=	a~=b	ne(a,b)	ungleich	$a \neq b$

4.2 Vergleichsoperatoren

Vergleichsoperatoren sind <, <=, >, >=, ==, and ~=. Mit ihnen wird ein Element-für-Element Vergleich zwischen zwei Feldern durchgeführt. Beide Felder müssen gleich groß sein. Als Antwort erhält man ein Feld gleicher Größe, mit dem jeweiligen Element auf logisch TRUE (1) gesetzt, wenn der Vergleich richtig ist, oder auf logisch FALSE (0) gesetzt wenn der Vergleich falsch ist.

Die Operatoren <, <=, > und >= verwenden nur den Realteil ihrer Operanden, wohingegen die Operatoren == und ~= den Real- und den Imaginärteil verwenden.

Wenn einer der Operanden ein Skalar ist und der andere eine Matrix, dann wird der Skalar auf die Größe der Matrix expandiert. Die beiden folgenden Beispiele geben daher das gleiche Resultat.

```
X = 5; X >= [1 2 3; 4 5 6; 7 8 10]
X = 5*ones(3,3); X >= [1 2 3; 4 5 6; 7 8 10]

ans =
     1     1     1
     1     1     0
     0     0     0
```

4.3 Logische Operatoren

Die Symbole &, |, and ~ stehen für die logischen Operatoren `and`, `or`, and `not`. Sind die Operanden Felder, wirken alle Befehle elementweise. Der Wert `0` representiert das logische FALSE (F), und alles was nicht Null ist, representiert das logische TRUE (T). Die Funktion `xor(A,B)` implementiert das "exklusive oder". Die Wahrheitstabellen für diese Funktionen sind in 4.5 zusammengestellt.

Tabelle 4.5: Logische Operatoren

INPUT		and	or	xor	not
A	B	A&B	A|B	xor(A,B)	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Wenn einer der Operanden ein Skalar ist und der andere eine Matrix, dann wird der Skalar auf die Größe des Feldes expandiert. Die logischen Operatoren verhalten sich dabei gleich wie die Vergleichsoperatoren. Das Ergebnis der Operation ist wieder ein Feld der gleichen Größe.

Die Priorität der logischen Operatoren ist folgendermaßen geregelt:

- **not** hat die höchste Priorität.
- **and** und **or** haben die gleiche Priorität und werden von links nach rechts abgearbeitet.

Die "Links vor Rechts" Ausführungspriorität in MATLAB macht `a|b&c` zum Gleichen wie `(a|b)&c`. In den meisten Programmiersprachen ist `a|b&c` jedoch das Gleiche wie `a|(b&c)`. Dort hat `&` eine höhere Priorität als `|`. Es ist daher in jedem Fall gut, mit Klammern die notwendige Abfolge zu regeln.

Eine Besonderheit stellen die beiden logischen Operatoren `&&` und `||`, die man als logische Operatoren mit short circuit bezeichnet. Dabei wird z.B. beim Befehl

```
x = (b ~= 0) && (a/b > 18.5)
```

der zweite Teil mit der Divison nicht mehr ausgeführt, wenn `b` gleich `0` ist. Dies ist nämlich nicht mehr notwendig, da das Ergebnis unabhängig vom zweiten Teil das Resultat `FALSE` liefern muss. Man spart sich damit in diesem Fall die Warnung, dass durch Null dividiert wird.

Besonders praktisch ist dieses Feature, wenn eine Variable zum Zeitpunkt der Berechnung nicht existiert. Wenn also `a` nicht als Variable existiert (`exist`), liefert der Befehl

```
~exist('a','var') | isempty(a)
```

die Fehlermitteilung `Undefined function or variable 'a'`, da der zweite Befehl (`isempty`) nicht ausgeführt werden kann. Verwendet man hingegen

```
~exist('a','var') || isempty(a)
```

kann die Zeile auch in diesem Fall ausgewertet werden und liefert den Wert `1` (`TRUE`), wenn `a` nicht existiert oder ein leeres Feld ist.

Die Verwendung von && und || stellt also sicher, dass jene Teile des logischen Konstrukts nicht mehr ausgeführt werden, wenn sie das Ergebnis nicht mehr beeinflussen können.

Bei der Verwendung von Vergleichsoperatoren und logischen Operatoren in Steuerkonstrukten, wie z.B. `if-Strukturen`, ist zur Entscheidung natürlich nur ein skalarer logischer Wert möglich. Einen solchen kann man aus logischen Arrays durch die Befehle:

`any(M)` **oder** `any(M,DIM)`**:** Ist `TRUE`, wenn ein Element ungleich Null ist.

`all(M)` **oder** `all(M,DIM)`**:** Ist `TRUE`, wenn alle Elemente ungleich Null sind.

> Wenn die Befehle `any(M)` und `all(M)` auf Felder angewandt werden, verhalten sie sich analog zu anderen Befehlen (wie z.B. `sum(M)`) und führen die Operation entlang der ersten von Eins verschiedenen Dimension aus. Das Ergebnis ist dann in der Regel kein Skalar.

Die Ergebnisse von Vergleichsoperationen und logischen Operarationen können für die logische Indizierung, 3.6.1, verwendet werden.

Ist man nur an den Positionen interessiert, kann man mit `I = find(L)` die linearen Indices, bzw. mit `[m,n] = find(L)` die 2-dim Indices erhalten, für die die Bedingung in `L` erfüllt ist.

Beispiel mit `find`, `ind2sub` und `sub2ind`:

```
m = reshape([1:12],3,4);        m = [ 1     4     7    10
                                      2     5     8    11
                                      3     6     9    12 ]

l = m>3 & m<8;                  l = [ 0     1     1     0
                                      0     1     0     0
                                      0     1     0     0 ]

i = find(l);                     i = [ 4;    5;    6;    7 ]

[si,sj] = find(l);              si = [ 1;    2;    3;    1 ]
                                sj = [ 2;    2;    2;    3 ]

Umrechnung: [si,sj] = ind2sub(size(m),i);
                  i = sub2ind(size(m),si,sj);
```

Beispiel mit `any` und `all`:

```
m = reshape([1:12],3,4);
```

```
m = [ 1     4     7    10
      2     5     8    11
      3     6     9    12 ]
```

```
l = m>=2 & m<=11;
```

```
l = [ 0     1     1     1
      1     1     1     1
      1     1     1     0 ]
```

```
an1 = any(l)
al1 = all(l);
```

```
an1 = [ 1     1     1     1 ]
al1 = [ 0     1     1     0 ]
```

```
an2 = any(l,2); al2 = all(l,2);
```

```
an2 = [ 1          al2 = [ 0
        1                  1
        1 ]                0 ]
```

```
an = any(l(:));
al = all(l(:));
```

```
an = 1
al = 0
```

Kapitel 5

Operatoren für Matrizen - Lineare Algebra

Als Matrizen bezeichnet man eine rechteckige Anordnung von Zahlen (oder Variablen). Im Unterschied zu den Feldern (Arrays), die exakt das gleiche Aussehen haben, werden hier Matrizen als Konstrukte der linearen Algebra aufgefasst, für die natürlich andere Regeln in Bezug auf für Multiplikation und Division gelten.

Eine Matrix $\mathbf{A}$ kann geschrieben werden als

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & \dots & a_{mn} \end{bmatrix} = [a_{jk}] \, . \tag{5.1}$$

Dies ist eine $m \times n$ Matrix mit m Zeilen (rows) and n Spalten (columns). Die einzelnen Elemente werden in der Mathematik mit Hilfe von Indizes a_{jk} bezeichnet, in MATLAB lautet die Schreibweise `A(j,k)`. Die Matrix $\mathbf{A}$ ist 2-dimensional, es gibt jedoch keine Beschränkung in der Anzahl der Dimensionen. Die beiden MATLAB Befehle `ndims` und `size` geben die jeweilige Dimension der Matrix und die Größe in jeder Dimension. Einige Befehle und die zugrundeliegenden Konzepte (z.B.: Transponieren) sind aber nur für 2-dim Matrizen definiert.

Spezielle zweidimensionale Matrizen sind:

Spaltenvektor, column vector: Matrix mit nur einer Spalte,

$$\mathbf{a} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = [a_j] \, . \tag{5.2}$$

Die Eingabe in MATLAB erfolgt mit `a=[1;2;3]` oder `a=[1,2,3].'`. Die Anzahl der Dimensionen ist 2, der Befehl `size` liefert `[3 1]`.

Zeilenvektor, row vector: Matrix mit nur einer Zeile,

$$\mathbf{b} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = [b_j] \ . \tag{5.3}$$

Die Eingabe in MATLAB erfolgt mit `b=[1,2,3]`. Die Anzahl der Dimensionen ist 2, der Befehl `size` liefert `[1 3]`.

Skalar, scalar: Matrize reduziert auf eine einzige Zahl,

$$s = \mathbf{s} = s_{11} = s_1 = [s_j] \ . \tag{5.4}$$

Die Anzahl der Dimensionen ist auch hier 2, der Befehl `size` liefert `[1 1]`.

5.1 Transponieren einer Matrix

Es erweist sich als praktisch, die Transponierte einer Matrix zu definieren. Die transponierte Matrix $\mathbf{A}^T$ einer $m \times n$ Matrix $\mathbf{A} = [a_{jk}]$ ist eine $n \times m$ Matrix, wobei die Zeilen in Spalten und die Spalten in Zeilen verwandelt werden,

$$\mathbf{A}^T = [a_{kj}] \ . \tag{5.5}$$

Mit Hilfe der transponierten Matrix können zwei Klassen von reellen quadratischen Matrizen definiert werden:

Symmetrische Matrix: Für eine symmetrische Matrix gilt

$$\mathbf{A}^T = \mathbf{A} \ . \tag{5.6}$$

Schiefsymmetrische Matrix: Für eine schiefsymmetrische Matrix gilt

$$\mathbf{A}^T = -\mathbf{A} \ . \tag{5.7}$$

Zerlegung: Jede quadratische Matrix $(n \times n)$ lässt sich in eine Summe aus einer symmetrischen und einer schiefsymmetrischen Matrix zerlegen,

$$\mathbf{A} = \mathbf{S} + \mathbf{U} \ , \tag{5.8}$$

$$\mathbf{S} = \frac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right) \ , \tag{5.9}$$

$$\mathbf{U} = \frac{1}{2}\left(\mathbf{A} - \mathbf{A}^T\right) \ . \tag{5.10}$$

In MATLAB steht zum Transponieren der Operator `.'` oder der Befehl `transpose` zur Verfügung.

5.2 Addition und Subtraktion von Matrizen

Diese beiden Operationen existieren nur "elementweise", d.h. es gibt keinen Unterschied zwischen Matrix- und Array-Operationen

$$\mathbf{C} = \mathbf{A} \pm \mathbf{B} \implies [c_{jk}] = [a_{jk}] \pm [b_{jk}] \ , \tag{5.11}$$

und daher ist die Defintion von `.+` und `.-` Operatoren nicht notwendig.

Voraussetzung: Gleiche Dimension und gleiche Größe von $\mathbf{A}$ und $\mathbf{B}$.

Ausnahme: A oder B ist ein Skalar, dann wird die skalare Größe zu allen Elementen addiert (oder von allen subtrahiert).

Beispiele: Mit $\mathbf{A} = [1\ 2\ 3]$ und $\mathbf{B} = [4\ 5\ 6]$ ergibt sich,

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \begin{bmatrix} 5\ 7\ 9 \end{bmatrix} \ , \tag{5.12}$$

$$\mathbf{D} = \mathbf{A} + 1 = \begin{bmatrix} 2\ 3\ 4 \end{bmatrix} \ . \tag{5.13}$$

Fehler: Die Rechnung

$$\mathbf{C} = \mathbf{A} + \mathbf{B}^T = \begin{bmatrix} 1\ 2\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \text{Error} \tag{5.14}$$

ergibt natürlich eine Fehlermitteilung.

5.3 Skalar Multiplikation

Das Produkt einer $m \times n$ Matrix $\mathbf{A} = [a_{jk}]$ und eines Skalars c (Zahl c) wird geschrieben als $c\mathbf{A}$ und ergibt die $m \times n$ Matrix $c\mathbf{A} = [ca_{jk}]$.

5.4 Matrix Multiplikation

Dies ist eine Multiplikation im Sinne der linearen Algebra. Das Produkt $\mathbf{C} = \mathbf{AB}$ einer $m \times n$ Matrix $\mathbf{A} = [a_{jk}]$ und einer $r \times p$ Matrix $\mathbf{B} = [b_{jk}]$ ist nur dann definiert, wenn gilt $n = r$. Die Multiplikation ergibt eine $m \times p$ Matrix $\mathbf{C} = [c_{jk}]$, deren Elemente gegeben sind als,

$$c_{jk} = \sum_{l=1}^{n} a_{jl} b_{lk} \ . \tag{5.15}$$

In MATLAB steht für die Matrizenmultiplikation der Befehl `C=mtimes(A,B)` oder die Operatorform `C=A*B` zur Verfügung, wobei nach den oben genannten Regeln die "inneren" Dimensionen übereinstimmen müssen, d.h., die Anzahl der Spalten von A muss mit der Anzahl der Zeilen von B übereinstimmen (Index l in 5.15).

Im Unterschied zur Multiplikation von Skalaren ist die Multiplikation von Matrizen nicht kommutativ, im Allgemeinen gilt daher

$$\mathbf{AB} \neq \mathbf{BA} \; . \tag{5.16}$$

Außerdem folgt aus $\mathbf{AB} = 0$ nicht notwendigerweise $\mathbf{A} = 0$ oder $\mathbf{B} = 0$ oder $\mathbf{BA} = 0$.

Beispiele: Multiplikation einer (2×3)-Matrix mit einer (3×2)-Matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 14 & 32 \\ 32 & 77 \end{bmatrix} \; . \tag{5.17}$$

Multiplikation einer (3×2)-Matrix mit einer (2×3)-Matrix:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 17 & 22 & 27 \\ 22 & 29 & 36 \\ 27 & 36 & 45 \end{bmatrix} \; . \tag{5.18}$$

Multiplikation einer (3×2)-Matrix mit einer (3×2)-Matrix:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \text{Error} \; . \tag{5.19}$$

Inneres Produkt zweier Vektoren:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 32 \; . \tag{5.20}$$

Äußeres Produkt zweier Vektoren:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix} \; . \tag{5.21}$$

Fehler:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \text{Error} \; . \tag{5.22}$$

Für das Transponieren von Produkten, $\mathbf{C} = \mathbf{AB}$, kann sich ganz leicht davon überzeugen, dass gilt:

$$\mathbf{C}^T = \mathbf{B}^T \mathbf{A}^T \tag{5.23}$$

$$\mathbf{C} = (\mathbf{B}^T \mathbf{A}^T)^T \tag{5.24}$$

$$(c\mathbf{A})^T = c\mathbf{A}^T \tag{5.25}$$

5.5 Inneres Produkt zweier Vektoren

Wenn $\mathbf{a}$ und $\mathbf{b}$ Spaltenvektoren der Länge n sind, dann ist $\mathbf{a}^T$ ein Zeilenvektor und das Produkt $\mathbf{a}^T\mathbf{b}$ ergibt die 1×1 Matrix (bzw. die Zahl), welche **inneres Produkt** von $\mathbf{a}$ und $\mathbf{b}$ genannt wird. Das innere Produkt wird mit $\mathbf{a} \cdot \mathbf{b}$ bezeichnet

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_{l=1}^{n} a_l b_l \ . \tag{5.26}$$

In MATLAB existiert dafür der Befehl `dot(a,b)`. Er berechnet das innere Produkt zweier Vektoren, und kümmert sich nicht um deren "Ausrichtung" als Zeilen- oder Spaltenvektor.

Das innere Produkt hat interessante Anwendungen in der Mechanik und der Geometrie.

5.6 Spezielle Matrizen

Für die Beschreibung der Matrix Division beschränken wir uns vorerst auf quadratische Matrizen ($n \times n$). Dafür benötigen wir zuerst die Definition der **Einheitsmatrix**

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \ . \tag{5.27}$$

Für die Einheitsmatrix gilt

$$\mathbf{AI} = \mathbf{IA} = \mathbf{A} \ . \tag{5.28}$$

In MATLAB steht für die Erzeugung der Befehl `eye` (zB. `eye(3)`) zur Verfügung.

Die **inverse Matrix** ist definiert durch

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \ . \tag{5.29}$$

Eine Matrix für die

$$\mathbf{A}^T = \mathbf{A}^{-1} \tag{5.30}$$

gilt, wird als **orthogonale** Matrix bezeichnet.

In MATLAB gibt es zur Berechnung der inversen Matrix den Befehl `inv(A)`.

5.7 Matrix Division - Lineare Gleichungssysteme

Die Division von Matrizen kann man sich am besten mit Hilfe von linearen Gleichungssystemen vorstellen. Solche Gleichungssysteme können sehr elegant mit Hilfe von Matrizen formuliert werden. Bei bekanntem $\mathbf{A}$ und $\mathbf{b}$ kann eine Gleichung für $\mathbf{x}$ folgendermaßen geschrieben werden

$$\mathbf{A}\mathbf{x} = \mathbf{b} \; . \tag{5.31}$$

Im Allgemeinen ist die Koeffizientenmatrix $\mathbf{A} = [a_{jk}]$ die $m \times n$ Matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \text{, und } \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} \text{ und } \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \tag{5.32}$$

sind Spaltenvektoren. Man sieht, dass die Anzahl der Elemente von $\mathbf{x}$ gleich n und von $\mathbf{b}$ gleich m ist. Dadurch wird ein lineares System von m Gleichungen in n Unbekannten beschrieben:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array} \tag{5.33}$$

Die a_{jk} sind gegebene Zahlen, die **Koeffizienten** des Systems genannt werden. die b_i sind ebenfalls gegebene Zahlen. Wenn alle b_i gleich Null sind, handelt es sich um ein **homogenes System**, wenn hingegen zumindest ein b_i ungleich Null ist, handelt es sich um ein **inhomogenes System**.

Die Lösung von 5.33 ist ein Vektor $\mathbf{x}$ der Länge n, der alle m Gleichungen erfüllt. Ist das System 5.33 homogen, hat es zumindest die **triviale** Lösung

$$x_1 = 0 \; , \; x_2 = 0 \; , \; \dots \; , \; x_n = 0 \; . \tag{5.34}$$

Ein System heißt

überbestimmt, wenn es mehr Gleichungen als Unbekannte hat ($m > n$);

bestimmt, wenn es gleich viel Gleichungen wie Unbekannte hat ($m = n$);

unterbestimmt, wenn es weniger Gleichungen als Unbekannte hat ($m < n$).

Ein unterbestimmtes System hat immer eine Lösungsschar, ein bestimmtes Gleichungssystem hat mindestens eine Lösung, und ein überbestimmtes System hat nur eventuell eine Lösung.

Um x zu berechnen, kann man nun formal 5.31 von Links mit $\mathbf{A}^{-1}$ multiplizieren

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \implies \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{A}\backslash\mathbf{b}. \tag{5.35}$$

Dafür steht in MATLAB der Befehl `x=A\b` bereit. Das Zeichen \ steht für die sogenannte "Matrix-Linksdivision". Es wird hier in der Numerik verwendet, in der mathematischen Beschreibung der linearen Algebra aber nicht.

Handelt es sich um ein **bestimmtes** Gleichungsystem, löst MATLAB das System mit Hilfe des Gaußschen Eliminierungsverfahrens.

Handelt es sich um eine **über-** oder **unterbestimmtes** Gleichungssystem, findet MATLAB die Lösung mit Hilfe des "Least Squares"-Verfahrens, dass später besprochen wird.

Lautet das lineare Gleichungssystem hingegen

$$\mathbf{y}\mathbf{A} = \mathbf{c}\ , \tag{5.36}$$

wobei $\mathbf{y}$ und $\mathbf{c}$ jetzt Zeilenvektoren der Länge m bzw. n sind, muss man von rechts mit $\mathbf{A}^{-1}$ multiplizieren und erhält

$$\mathbf{y}\mathbf{A}\mathbf{A}^{-1} = \mathbf{c}\mathbf{A}^{-1} \implies \mathbf{y} = \mathbf{c}\mathbf{A}^{-1} = \mathbf{c}/\mathbf{A}\ . \tag{5.37}$$

Dafür steht in MATLAB der Befehl `y=c/A` bereit. Das Zeichen `/` steht dafür für "Matrix-Rechtsdivision". Mit Hilfe der Regeln für das Transponieren, kann 5.37 umgeformt werden in

$$\mathbf{y} = \left((\mathbf{A}^{-1})^T\mathbf{c}^T\right)^T = \left(\mathbf{A}^T\backslash\mathbf{c}^T\right)^T\ , \tag{5.38}$$

wobei dies die Form ist, die MATLAB intern verwendet (`y=(A.'\c.').'`).

Für die Skalare s und r würde gelten

$$\begin{array}{rcccl} s/r & = & sr^{-1} & = & s/r \\ r\backslash s & = & r^{-1}s & = & s/r, \end{array} \tag{5.39}$$

was in beiden Fällen das Gleiche ist, da die Multiplikation von Skalaren kommutativ ist. Dies gilt jedoch nicht für die Matrizenmultiplikation.

MATLAB hat darüber hinaus den Vorteil, dass lineare Gleichungssysteme simultan für verschiedene inhomogene Vektoren $\mathbf{b}$, die in einer Matrix $\mathbf{B}$ zusammengefasst

sind, gelöst werden können. Man kann 5.33 umschreiben als:

$$\begin{array}{ccccccccc}
a_{11}x_{11} & + & a_{12}x_{21} & + & \dots & + & a_{1n}x_{n1} & = & b_{11} \\
a_{21}x_{11} & + & a_{22}x_{21} & + & \dots & + & a_{2n}x_{n1} & = & b_{21} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{11} & + & a_{m2}x_{21} & + & \dots & + & a_{mn}x_{n1} & = & b_{m1} \\
\hline
a_{11}x_{12} & + & a_{12}x_{22} & + & \dots & + & a_{1n}x_{n2} & = & b_{12} \\
a_{21}x_{12} & + & a_{22}x_{22} & + & \dots & + & a_{2n}x_{n2} & = & b_{22} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{12} & + & a_{m2}x_{22} & + & \dots & + & a_{mn}x_{n2} & = & b_{m2} \\
\hline
\vdots & + & \vdots & + & \vdots & + & \vdots & = & \vdots \\
\hline
a_{11}x_{1p} & + & a_{12}x_{2p} & + & \dots & + & a_{1n}x_{np} & = & b_{1p} \\
a_{21}x_{1p} & + & a_{22}x_{2p} & + & \dots & + & a_{2n}x_{np} & = & b_{2p} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{1p} & + & a_{m2}x_{2p} & + & \dots & + & a_{mn}x_{np} & = & b_{mp}
\end{array} \tag{5.40}$$

Dieses Gleichungssystem kann formal geschrieben werden als

$$\mathbf{AX} = \mathbf{B} \, , \tag{5.41}$$

wobei die $m \times p$ Inhomogenitätsmatrix $\mathbf{B}$ aus p nebeneinander angeordneten Spaltenvektoren $\mathbf{b}$ besteht. Auf die genau gleiche Weise liegen die Lösungen in den p Spaltenvektoren der $n \times p$ Matrix $\mathbf{X}$. Die Lösung erfolgt in MATLAB mit dem analogen Befehl `X=A\B`.

Kapitel 6

Steuerkonstrukte

6.1 Sequenz

Die einfachste Steuerstruktur ist die Aneinanderreihung von Programmteilen. Diese Programmteile sind Teile des gesamten Algorithmus und werden Teilalgorithmen oder auch Strukturblöcke genannt. Durch die Aneinanderreihung wird die zeitliche Abarbeitung von Strukturblöcken $S_1, S_2, \ldots, S_n$ in der Reihenfolge der Niederschrift festgelegt.

MATLAB Beispiel

Die zeitliche Abfolge wird durch Aneinanderreihung von Befehlen erreicht. Bei allen Zuweisungen (=) muss sichergestellt sein, dass alle Variablen auf der rechten Seite bereits bekannt sind.

Ändert man den Wert einer Variablen, ändern sich nicht automatisch damit vorher berechnete Größen. Man muss, z.B. die Modulo-Division `mod` nach der Änderung von `a` wieder ausführen, damit sich auch `r` ändert.

```
a = 10; b = 3;
r = mod(a,b)

     1

a = 12;
r = mod(a,b)

     0
```

6.2 Auswahl

In Programmen ist es häufig notwendig, Entscheidungen zu treffen, welche Strukturblöcke abgearbeitet werden sollen. Man trifft dabei mit Hilfe von logischen Ausdrücken, sogenannten Bedingungen, Entscheidungen, die den Ablauf eines Programms direkt beeinflussen.

Dafür gibt es in jeder Programmiersprache sogenannte Steueranweisungen, die die einzelnen Anweisungsblöcke einrahmen. Diese definieren den Beginn und das Ende der Steueranweisung und regeln den Ablauf innerhalb des Steuerkonstrukts.

In MATLAB gibt es für Entscheidungen zwei Strukturen, den sogenannten `IF-Block` oder die `Auswahlanweisung`, die in der Folge an konkreten Beispielen besprochen werden.

6.2.1 IF-Block

Die einfachste Form des `IF-Blocks` ist die einseitig bedingte Anweisung, die die bedingte Ausführung eines Anweisungsblocks erlaubt.

```
if Bedingung
   Anweisungsblock
end
```

Diese Form wird häufig auch in einer einzeiligen Version geschrieben:

```
if Bedingung, Anweisung; end
```

Diese einfachste Form kann durch beliebig viele `elseif Anweisungen` und maximal eine `else Anweisungen` erweitert werden. In ihrer vollständigen Form hat der `IF-Block` daher die folgende Form:

```
if Bedingung 1
   Anweisungsblock 1
elseif Bedingung 2
   Anweisungsblock 2
...
else
   Anweisungsblock n
end
```

MATLAB Beispiel

Das Programmfragment erkennt, ob eine Zahl x durch 2 und 3, bzw. nur durch 2 oder nur durch 3 oder gar nicht durch 2 und 3 teilbar ist.

Mit dem Befehl `mod(a,b)` wird die Modulodivision a/b durchgeführt. Wenn diese Null ergibt ist die Zahl a durch b teilbar.

Es wird immer nur ein Anweisungsblock ausgeführt, obwohl die Bedingungen bei mehreren erfüllt sein können.

```
if mod(x,2)==0 & mod(x,3)==0
    disp('Durch 2 und 3 teilbar!')
elseif mod(x,2)==0
    disp('Nur durch 2 teilbar!')
elseif mod(x,3)==0
    disp('Nur durch 3 teilbar!')
else
    disp('Nicht teilbar!')
end
```

Folgende wichtige Regeln gelten für `IF`-Blöcke:

- Die Bedingungen müssen logische Ausdrücke sein, mit deren Hilfe die Entscheidung getroffen wird. Es können keine logischen Felder, wie sie bei der logischen Indizierung verwendet werden, direkt die Steuerung übernehmen, da diese mehrdeutig sein können.
- Muss man logische Felder verwenden, kann man die Befehle `all(L(:))` oder `any(L(:))` anwenden, die `TRUE` ergeben, wenn alle Elemente bzw. zumindest ein Element des Feldes `TRUE` sind.
- Die direkte Anwendung der logischen Indizierung kann sehr häufig `IF`-Blöcke bei der Manipulation von Feldern ersetzen.
- Die Bedingungen werden nacheinander überprüft und es wird der erste Anweisungsblock ausgeführt, bei dem die Bedingung erfüllt ist. Danach wird das Programm am Ende des `IF`-Blocks fortgesetzt.
- Es ist erlaubt, dass sich Bedingungen überlappen. Es wird jedoch nur ein Anweisungsblock ausgeführt.
- Falls keine Bedingung erfüllt ist, wird bei Vorhandensein einer `else`-Anweisung der dort spezifizierte Block ausgeführt. Falls auch keine `else`-Anweisung vorhanden ist, wird kein Befehl ausgeführt.
- Will man für den weiteren Programmablauf sicherstellen, dass eine Variable innerhalb eines `IF`-Blocks zugewiesen wird, muss man entweder alle möglichen Fälle mit den Bedingungen abdecken, oder unbedingt die `else`-Anweisung verwenden.

- Mehrere `IF`-Blöcke können ineinander geschachtelt werden, wobei jeder mit einem `end` abgeschlossen werden muss.
- Zur besseren Lesbarkeit von Programmen sollte man die Anweisungsblöcke je nach Zugehörigkeit zu Steuerkonstrukten einrücken. Damit wird die Struktur von Programmen viel leichter ersichtlich.

6.2.2 Auswahlanweisung

Die `Auswahlanweisung` ist dem `IF`-Block sehr ähnlich und ermöglicht die Ausführung maximal eines Blocks von mehreren möglichen Anweisungsblöcken. Eine `Auswahlanweisung` hat folgende Form:

```
switch Schalter
case Selektor 1
   Anweisungsblock 1
case Selektor 2
   Anweisungsblock 2
...
otherwise
   Anweisungsblock 2
end
```

Bei der Verwendung ist Folgendes zu beachten:

- Während beim `IF-Block` mehrere Bedingungen ausgewertet werden können, richtet sich die Abarbeitung einer `Auswahlanweisung` nach dem Wert eines einzigen Ausdrucks, dem sogenannten Schalter. Dieser Schalter kann ein Skalar oder eine Zeichenkette sein. Er muss **keine logische Variable** sein.
- Die Abarbeitung erfolgt mit Hilfe der `case`-Anweisung. Die Ausführung wird durch einen Vergleich zwischen Schalter und Selektor geregelt. Stimmen die beiden überein, wird der zugehörige Auswahlblock ausgeführt und danach an das Ende der `Auswahlanweisung` gesprungen. Es wird also maximal ein Auswahlblock ausgeführt.
- Die Auswahl erfolgt naturgemäß wieder mit Skalaren oder Zeichenketten. Sollen für einen `case` mehrere Möglichkeiten erlaubt sein, kann man für den Selektor eine durch Beistriche getrennte Liste angeben. Solche Listen werden mit Hilfe geschwungener Klammern geschrieben, z.B. `{1,2,3}` oder `{'a','b','c'}`.
- Während sich die Bedingungen eines `IF-Blocks` überschneiden können, müssen die Alternativen einer `Auswahlanweisung` eindeutig sein.

- Wenn keine der von den Selektoren abgedeckten Bedingungen zutrifft, wird der Anweisungsblock einer eventuell vorhandenen `otherwise`-Anweisung ausgeführt.

MATLAB Beispiel

Dieser Programmteil zeigt an, ob die Zahl x kleiner, gleich oder größer Null ist.

Als Schalter dient dafür die Signum-Funktion `sign`.

Im zweiten Teil wird wird an Stelle des dritten Falles einfach `otherwise` verwendet, da es sonst keine Möglichkeiten mehr gibt.

```
switch sign(x)
case -1
    disp('x<0')
case 1
    disp('x>0')
case 0
    disp('x==0')
end

switch sign(x)
case -1
    disp('x<0')
case 1
    disp('x>0')
otherwise
    disp('x==0')
end
```

MATLAB Beispiel

Dieser Programmteil zeigt an, ob ein String `str` gleich einem bestimmten Buchstaben ist.

Als Schalter dient dafür einfach der String `str`.

Im dritten Fall wird eine Liste zum Vergleich herangezogen. Listen werden in MATLAB mit dem Klammerpaar `{}` umschlossen.

```
switch str
case 'a'
    disp('Fall a')
case 'b'
    disp('Fall b')
case {'c','d','e'}
    disp('Fall c, d, oder e')
otherwise
    disp('Nicht bekannt!')
end
```

MATLAB Beispiel

Dieses kleine Unterprogramm berechnet die Tage pro Monat in einem beliebigen Jahr unter Berücksichtigung der Schaltjahre ab der Kalenderreform im Jahr 1582.

An diesem Beispiel sieht man die Verschachtelung von `switch-case`- und `if`-Strukturen.

Wenn in der Zeile Platz ist, können die Keywörter und die Befehle in einer Zeile stehen. Als Trennzeichen wird dann der Beistrich verwendet.

```
function [tage] = tagepromonat(monat,jahr)
sj = 0;
switch jahr > 1582
case 1
    if mod(jahr,  4)==0, sj=1; end
    if mod(jahr,100)==0, sj=0; end
    if mod(jahr,400)==0, sj=1; end
end

switch monat
case {4,6,9,11},         tage = 30;
case {1,3,5,7,8,10,12}, tage = 31;
case 2
    switch sj
    case 0, tage = 28;
    case 1, tage = 29;
    end
otherwise
    tage = 0; error('Falscher Monat');
end
```

6.3 Wiederholung

Ein wichtiges Konstruktionsmittel in Programmiersprachen ist die Wiederholung. Sie erlaubt die wiederholte Ausführung einer Anweisungsfolge, ohne dass man gezwungen ist, die entsprechenden Anweisungen mehrmals zu schreiben. Die Anzahl der Wiederholungen wird dabei durch einen Schleifenkopf bestimmt.

In MATLAB gibt es zwei Schleifentypen, die Zählschleife `for` und die bedingte Schleife `while`. Beide Schleifentypen können darüberhinaus mit dem Befehl `break` abgebrochen werden.

Schleifenkonstrukte haben eine große Bedeutung bei allen Iterationen, wo aus bekannten Werten neue erzeugt werden. Als Beispiel soll folgende Rekursionsformel dienen:

$$a_1 = 0\ ,\ a_2 = 1\ ,\ a_k = \frac{a_{k-2} + a_{k-1}}{2} \qquad \forall k \geq 3\ . \tag{6.1}$$

In vielen anderen Fällen, hat sich durch Matrix- und Arrayoperatoren, durch die Doppelpunktnotation und durch eine Fülle von MATLAB-Befehlen die Notwendigkeit für Schleifenkonstrukte verringert. Als wichtige Regel gilt, dass allen Befehlen, die direkt auf Felder angewandt werden, gegenüber einer expliziten Programmierung mit Schleifen Vorrang zu geben ist. Bei allen internen Befehlen kann die zeitliche Optimierung durch MATLAB viel besser durchgeführt werden. Außerdem werden bei Vermeidung von Schleifen die Programme weit einfacher, kürzer und übersichtlicher. Daher sollte man sich immer die Frage stellen, ob man eine Schleife wirklich braucht oder ob man die Aufgabe nicht besser anders erledigen kann.

6.3.1 Zählschleife

In der Zählschleife wird explizit angegeben, wie oft ein Anweisungsblock ausgeführt werden soll.

```
for Schleifenindex = Feld
   Anweisungsblock
end
```

Der Schleifenindex nimmt dabei nacheinander alle Werte der Elemente eines beliebigen Feldes "Feld" an. Der Ablauf erfolgt dabei entsprechend den Regeln der linearen Feldindizierung, zuerst entlang der ersten Dimension, dann der zweiten, usw.. Für jeden Wert des Schleifenindex wird der Anweisungsblock einmal ausgeführt und danach die Schleife beendet.

Durch eine Kombination mit einer `IF-Entscheidung` kann unter Verwendung des Befehls `break` die Schleife jederzeit beendet werden.

```
for Schleifenindex = Feld
   Anweisungsblock 1
   if Bedingung, break; end
   Anweisungsblock 2
end
```

Natürlich können auch Schleifen ineinander geschachtelt werden, wobei zu jedem `for` ein `end` gehören muss. Bei geschachtelten Schleifen beendet der Befehl `break` die jeweils innere Schleife.

Der Schleifenindex hat am Ende der Abarbeitung den jeweils letzten Wert. Man kann daher überprüfen, ob es zur Ausführung des `break-Befehls` gekommen ist.

6.3.2 Die bedingte Schleife

Bei der bedingten Schleife (`while`) hängt die Anzahl der Durchläufe von einer logischen Bedingung im Schleifenkopf ab. Die Bedingung wird bei jedem Schleifendurchlauf am Beginn ausgewertet. Der Anweisungsblock wird nur dann ausgeführt, wenn die Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, wird die Schleife beendet.

```
while Bedingung
   Anweisungsblock
end
```

Ist die Bedingung schon am Anfang nicht erfüllt, wird der Anweisungsblock nie ausgeführt. Natürlich kann auch eine solche Schleife an jeder beliebigen Stelle durch eine `break`-Anweisung unterbrochen werden.

```
while Bedingung
   Anweisungsblock 1
   if Abbruchbedingung, break; end
   Anweisungsblock 2
end
```

In manchen Programmiersprachen gibt es eine sogenannte nichtabweisende Schleife (UNTIL-Schleife), die zumindest einmal durchlaufen wird. Eine solche gibt es in MATLAB nicht, man kann sie jedoch mit Hilfe einer "Endlosschleife" und eine `break`-Anweisung realisieren.

```
while 1                % immer wahr
   Anweisungsblock
   if Bedingung, break; end
end
```

Am Beispiel der "Endlosschleife" sollte auch klar werden, welche Gefahr in Schleifen steckt, wenn die Abbruchbedingungen nicht gut durchdacht sind. In solchen Fällen ist es möglich, dass Schleifen von selbst nicht mehr verlassen werden. Dies kann dann nur durch einen externen Abbruch des Programms erfolgen. Solche Fehler sollten daher vermieden werden.

MATLAB Beispiel

Hier wird die Rekursionsformel

$$a_1 = 0\ ,\ a_2 = 1\ ,\ a_k = \frac{a_{k-2} + a_{k-1}}{2} \qquad \forall k \geq 3$$

für $1 \leq k \leq n$ ausgewertet und gezeigt, wie man bei einem Limit $|a_k - a_{k-1}| < \epsilon$ die Berechnung abbricht.

Die Realisierung erfolgt einmal mit einer `for`-Schleife ohne weitere Einschränkung, einmal mit einer `for`-Schleife mit zusätzlicher Verwendung des `break`-Befehls und einmal mit einer Kombination aus `while`-Schleife und `break`-Befehl.

Wann immer es möglich ist, empfiehlt es sich, Felder vor dem Gebrauch in ihrer maximalen Größe anzulegen, damit sie nicht innerhalb der Schleife immer dynamisch vergrößert werden müssen.

Mit dem Befehl `c(k:end)=[]` wird der nicht benötigte Rest des Feldes gelöscht.

```
n = 100; limit = 1.e-6;

a = zeros(1,n); a(2) = 1;
for k = 3:n
    a(k) = (a(k-1) + a(k-2)) / 2;
end

b = zeros(1,n); b(2) = 1;
for k = 3:n
    b(k) = (b(k-1) + b(k-2)) / 2;
    if abs(b(k)-b(k-1)) < limit
        break;
    end
end
b(k+1:end) = [];

c = zeros(1,n); c(2) = 1;
k = 2;
while abs(c(k)-c(k-1)) >= limit
    k = k + 1;
    if k > n, break; end
    c(k) = (c(k-1) + c(k-2)) / 2;
end
c(k:end) = [];
```

Kapitel 7

Programmeinheiten

MATLAB kennt zwei Typen von Programmeinheiten, Skripts und Funktionen, die sich in ihrem Verhalten wesentlich unterscheiden. Beiden gemeinsam ist, dass sie in Files "filename.m" gespeichert sein müssen. Die Extension muss immer ".m" sein. Liegt ein solcher File im MATLAB-Pfad, so kann er innerhalb von MATLAB mit seinem Namen ohne die Extension ".m" ausgeführt werden.

Für eigene Skripts und Funktionen sollten keine Namen verwendet werden, die in MATLAB selbst Verwendung finden, da ansonsten MATLAB-Routinen "lahmgelegt" werden können. Falls man sich nicht sicher ist, ob ein Name bereits existiert, kann man den Befehl `exist('name')` verwenden. Falls `exist` den Wert 0 retourniert, kann man ihn beruhigt verwenden, falls der Wert 5 retourniert wird, handelt es sich um eine interne MATLAB-Routine.

Sowohl Skripts als auch Funktionen helfen, wiederkehrende Aufgaben zu erledigen und dienen daher einer besseren Strukturierung und der Arbeitserleichterung.

Skripts: Sie enthalten eine Abfolge von MATLAB-Befehlen und werden ohne Übergabeparameter aufgerufen. Die Befehle laufen in gleicher Weise hintereinader ab, wie wenn man sie Schritt für Schritt eingeben würde.

Skripts können alle bereits im Workspace definierten Variablen verwenden und verändern.

Nach ihrem Ablauf sind alle dort definierten Variablen bekannt. Werden mehrere Skripts exekutiert, kann es zu unliebsamen Überschneidungen kommen, wenn z.B. ungewollt in mehreren Skripts die gleichen Variablennamen verwendet werden.

Dadurch, dass die Variablen nicht in einem eigenen Workspace gekapselt sind, eignen sich Skripts nicht wirklich für eine schöne modulare Trennung von Programmen in selbständige Teile.

Funktionen: Im Unterschied zu Skripts enthalten Funktionen eine Deklarationszeile, die sie klar als Funktion kennzeichnet.

Ihre Deklaration enthält normalerweise auch sogenannte Übergabeparameter, die in Eingabe- und Ausgabeparameter gegliedert sind.

Funktionen laufen in einem lokalen Workspace ab, der zum jeweiligen Funktionsaufruf gehört. Dadurch findet eine totale Kapselung der Variablen statt und es kann zu keinen Überschneidungen mit anderen Programmen kommen, solange auf die Deklaration und die Verwendung globaler Variablen verzichtet wird.

Die einzige Verbindung zwischen den Variablen innerhalb einer Funktion und dem Workspace einer aufrufenden Funktion (bzw. dem MATLAB-Workspace) sind die Ein- und Ausgabeparameter. Die Variablen innerhalb einer Funktion existieren nur temporär während der Funktionsausführung.

Durch diese Art der Kapselung ist es auch möglich, dass Funktionen sich selbst aufrufen. Dies nennt man Rekursion.

7.1 FUNCTION-Unterprogramme

7.1.1 Deklaration

Die Deklaration eines FUNCTION-Unterprogramms ist mit der Anweisung `function` auf folgende Arten möglich:

```
function name
function name(Eingangsparameter)
function Ausgangsparameter = name
function Ausgangsparameter = name(Eingangsparameter)
```

Gibt es mehrere Eingangsparameter sind diese durch Beistriche zu trennen. Gibt es mehrere Ausgangsparameter, ist die Liste der Parameter durch Beistriche zu trennen und mit eckigen Klammern zu umschließen.

```
[aus_1, aus_2, ..., aus_n]
```

Ein Typ der Parameter muss, wie schon bei den Skripts, nicht explizit definiert werden, dieser ergibt sich durch die Zuweisungen innerhalb der Funktion.

Die Deklarationszeile sollte unmittelbar von einer oder von mehreren Kommentarzeilen gefolgt werden, die mit dem Prozentzeichen % beginnen. Diese werden beim Programmablauf ignoriert stehen aber als Hilfetext bei Aufruf von `help name` jederzeit zur Verfügung. Typischerweise sollen sie einem Benutzer mitteilen, was das jeweilige Programm macht.

Danach sollte eine Überprüfung der Eingabeparameter auf ihre Zulässigkeit bzw. auf ihre Anzahl erfolgen. Die Anzahl beim Aufruf muss nämlich nicht mit der Anzahl in der Deklaration übereinstimmen.

Danach folgen alle ausführbaren Anweisungen und die Zuweisung von Werten auf die Ausgangsparameter. Die Anzahl der Ausgangsparameter beim Aufruf muss ebenfalls nicht mit der Anzahl in der Deklaration übereinstimmen. Es muss aber sichergestellt werden, dass alle beim Aufruf geforderten Ausgangsparameter übergeben werden.

7.1.2 Resultat einer Funktion

Das Resultat einer Funktion ist - sofern es existiert - durch den Wert der Ausgangsparameter der Funktion gegeben. Diese können durch gewöhnliche Wertzuweisungen definiert werden; ihr Typ wird implizit über die Wertzuweisung bestimmt.

Normalerweise endet der Ablauf einer Funktion mit der Exekution der letzten Zeile. Es kann aber auch innerhalb der Funktion der Befehl `return` verwendet werden. Auch dies führt zu einer sofortigen Beendigung der Funktion. Dies kann z.B. bei Erfüllung einer Bedingung der Fall sein

```
if Bedingung, return; end
```

Übergeben wird jener Wert der Ausgangsparameter, der zum Zeitpunkt der Beendigung gegeben ist. Werden die Eingangsparameter verändert, hat das keinen Einfluss auf den Wert dieser Variablen im rufenden Programm.

7.1.3 Aufruf einer Funktion

Der Aufruf einer Funktion erfolgt gleich wie der Aufruf eines MATLAB-Befehls:

```
[aus_1, aus_2, ..., aus_n] = name(in_1, in_2, ..., in_m)
```

Viele der von MATLAB bereitgestellten Befehle liegen in Form von Funktionen vor. Sie können daher nicht nur exekutiert sondern auch im Editor angeschaut werden. Dies ist manchmal äußerst nützlich, da man dadurch herausfinden kann, wie MATLAB gewisse Probleme löst.

7.1.4 Überprüfung von Eingabeparametern

Für den Einsatz von Funktionen ist es sinnvoll, dass innerhalb von Funktionen die Gültigkeit der Eingabeparameter überprüft wird. Dies umfasst typischerweise die Überprüfung von

- der Dimension und Größe von Feldern,
- des Typs von Variablen, und
- des erlaubten Wertebereichs.

Damit soll ein Benutzer davor gewarnt werden, dass eine Funktion überhaupt nicht funktioniert oder für diese Parameter nur fehlerhaft rechnen kann. Dies sollte sinnvoll mit Fehlermitteilungen und Warnungen kombiniert werden, wie sie in 7.1.5 beschrieben werden.

In Ergänzung zu den bekannten logischen Abfragen, gibt es eine Reihe von MATLAB-Befehlen zur Überprüfung des Typs bzw. der Gleichheit oder des Inhalts. Sie geben für `k=1`, wenn die Bedingung erfüllt ist, bzw. `k=0`, wenn die Bedingung nicht erfüllt ist. Das Gleiche gilt für `TF`, außer dass hier ein logisches Feld zurückgegeben wird. Hier sind einige Beispiele angeführt. Eine gesamte Auflistung aller Befehle dieser Art findet man in der MATLAB-Hilfe für `is`.

`k = ischar(S)`	Zeichenkette
`k = isempty(A)`	Leeres Array
`k = isequal(A,B,...)`	Identische Größe und Inhalt
`k = islogical(A)`	Logischer Ausdruck
`k = isnumeric(A)`	Zahlenwert
`k = isreal(A)`	Reelle Werte
`TF = isinf(A)`	Unendlich
`TF = isfinite(A)`	Endliche Zahl
`TF = isnan(A)`	Not A Number
`TF = isprime(A)`	Primzahl

7.1.5 Fehler und Warnungen

Die MATLAB-Funktion `error` zeigt eine Nachricht im Kommandofenster an und übergibt die Kontrolle der interaktiven Umgebung. Damit kann man z.B. einen ungültigen Funktionsaufruf anzeigen.

```
if Bedingung, error('Nachricht'); end
```

Analog dazu gibt es den Befehl `warning`. Dieser gibt ebenfalls die Meldung aus, unterbricht aber nicht den Programmablauf. Falls Warnungen nicht erwünscht bzw. doch wieder erwünscht sind, kann man mit `warning off` bzw. `warning on` aus- bzw. einschalten, ob man gewarnt werden will.

7.1.6 Optionale Parameter und Rückgabewerte

MATLAB unterstützt die Möglichkeit, Formalparameter eines Unterprogramms optional zu verwenden. Das heißt, die Anzahl der Aktualparameter kann kleiner sein, als die Anzahl der Formalparameter.

Formalparameter sind jene Parameter, die in der Deklaration der Funktion spezifiziert werden.

Aktualparameter sind jene Parameter, die beim Aufruf der Funktion spezifiziert werden.

Bei einem Aufruf eines FUNCTION-Unterprogramms werden die Aktualparameter von links nach rechts mit Formalparametern assoziiert. Werden beim Aufruf einer Funktion weniger Aktualparameter angegeben, so bleiben alle weiteren Formalparameter ohne Wert, sie sind also undefiniert.

Werden solche undefinierten Variablen verwendet, beendet MATLAB die Abarbeitung des Programms mit einer Fehlermeldung. Der Programmierer hat zwei Möglichkeiten mit dieser Situation umzugehen:

- Sicherstellen, dass nicht übergebene Parameter nicht verwendet werden.
- Vergabe von Defaultwerten für nicht übergebene Parameter am Anfang des Programms.

Zu diesem Zweck hat MATLAB die beiden Variablen `nargin` und `nargout`, die nach dem Aufruf einer Funktion die Anzahl der aktuellen Eingabe-, bzw. Ausgabeparameter angeben. Mit Hilfe von `nargin` kann ganz leicht die Vergabe von Defaultwerten geregelt werden.

Ist die Anzahl der aktuellen Ausgabeparameter kleiner als die der Formalparameter, kann man sich das Berechnen der nicht gewünschten Ergebnisse sparen. Dies macht vor allem bei umfangreichen Rechnungen mit großem Zeitaufwand Sinn und kann helfen sehr viel Rechenzeit einzusparen.

Eine mögliche Realisierung einer solchen Überprüfung kann folgendermaßen aussehen:

```
function [o1,o2]=name(a,b,c,d)
% Hilfetext
if nargin<1, a=1; end
if nargin<2, b=2; end
if nargin<3, c=3; end
if nargin<4, d=4; end

if nargout>0, o1 = a+b; end
if nargout>1, o2 = c+d; end
```

Eine zusätzlich Möglichkeit bietet auch die Verwendung der Funktion `isempty`. Damit kann überprüft werden, ob ein Übergabeparameter als leeres Feld `[]` übergeben wird. Damit könnte obiges Beispiel so aussehen:

```
function [o1,o2]=name(a,b,c,d)
% Hilfetext
if nargin<1, a=1; end, if isempty(a), a=1; end
if nargin<2, b=2; end, if isempty(b), b=2; end
...
```

Nun würde auch ein Aufruf `[x,y]=name([],4,5,6)` den Defaultwert für `a` setzen.

7.1.7 Inline-Funktionen

Einfache Funktionen, die in einer Befehlszeile Platz finden, können auch mit Hilfe der Funktion `inline` definiert werden.

```
f1 = inline('x.^n .* exp(-x.^2)', 'x', 'n');
f2 = inline('m*exp(-n*(x.^2 + y.^2))', 'x', 'y', 'm', 'n');
```

Dabei muss als erster String die Funktion angegeben, der dann von Strings für die Inputparameter gefolgt wird. Die Reihenfolge der Strings für die Inputparameter entscheidet über die Reihenfolge beim Aufruf. Es ist in manchen Fällen auch möglich, keine Inputparameter zu übergeben. Von einer Verwendung dieser Eigenschaft wird jedoch abgeraten.

Wichtig ist auch hier, dass die Funktionen mit den richtigen Operatoren geschrieben werden, sodass eine Verwendung auch für Vektoren und Arrays möglich ist.

Bei der Definition der `inline`-Funktion wird keine Überprüfung der Syntax der Funktion und auch keine Überprüfung der Übergabeparameter durchgeführt. Daher werden in dieser Phase keine Fehler erkannt, die dann erst bei der Verwendung auftreten. Typische Fehlermitteilungen sind dann

```
Error using ==> inlineeval
Error in inline expression ==> ....
```

7.1.8 Unterprogramme als Parameter

Bisher wurden Datenobjekte als Parameter eines Unterprogramms betrachtet. Man kann jedoch auch Unterprogramme als Parameter an weitere Unterprogramme übergeben. In diesem Fall wird dem aufgerufenen Unterprogramm der Name des Unterprogramms als String oder als Funktionenhandle übergeben. Dies funktioniert natürlich auch mit `inline`-Funktionen, in diesem Fall muss diese Funktion direkt übergeben werden.

Als Beispiel sollen hier die Funktionen `quad`, `quadl` und `dblquad` verwendet werden, wobei zuerst die `inline`-Funktionen aus 7.1.7 Verwendung finden.

Die beiden Unterprogramme `quad` und `quadl` unterscheiden sich durch die verwendete numerische Methode, `quad` verwendet eine adaptive Simpson Methode und `quadl` verwendet eine adaptive Lobatto Methode.

Berechnet sollen z.B. folgende Integrale werden.

$$A_1 = \int_a^b dx\; x^n \exp(-x^2) \tag{7.1}$$

$$A_2 = \int_a^b \int_c^d dxdy\; m \exp\left(-n(x^2+y^2)\right) \tag{7.2}$$

```
A1 = quadl(f1, a, b, TOL, ANZEIGE, n)
A1 = quadl(f1, a, b, [],  [],      n)

A2 = dblquad(f2, a, b, c, d, TOL, METHODE, m, n)
A2 = dblquad(f2, a, b, c, d, [],  [],      m, n)
```

Die Reihenfolge der Inputparameter für `f1` bzw. `f2` ist ganz wichtig, es müssen immer jene Variablen vorne stehen über die integriert wird. Erst danach kann einen beliebige Anzahl von anderen Größen folgen, die durch die Integrationsroutine nur durchgeschleust werden, d.h. diese zusätzlichen Größen haben mit der Integrationsroutine nichts zu tun, werden aber für die Auswertung des Integranden benötigt.

Die optionalen Werte für `TOL`, `ANZEIGE` und `METHODE` müssen nicht übergeben werden. Falls man, wie in unseren Beispielen, weitere Parameter für die zu integrierende Funktion braucht, müssen für `TOL`, `ANZEIGE` und `METHODE` entweder Werte oder leere Arrays `[]` übergeben werden. Die Defaultwerte sind

```
TOL = 1.e-6, ANZEIGE = 0, METHODE='quad'
```

Größere Werte für `TOL` resultieren in einer geringeren Anzahl von Funktionsaufrufen und daher einer kürzeren Rechenzeit, verschlechtern aber natürlich die Genaugkeit des Ergebnisses.

Setzt man `ANZEIGE = 1`, bekommt man eine Statistik der Auswertung am Schirm ausgegeben. Bei der `METHODE` hat man die Wahl zwischen `'quad'` und `'quadl'`, wobei dies den MATLAB-Funktionen `quad` und `quadl` entspricht. In Prinzip könnte man auch eine eigene Integrationsroutine zur Verfügung stellen, die den gleichen Konventionen wie `quad` folgen muss.

Liegen die Funktionen nicht als `inline`-Funktionen sondern als Funktionen in den Files `ff1.m` bzw. `ff2.m` vor, so hat man zwei Möglichkeiten, (i) Angabe des Namens als String `'ff1'` oder als Funktionshandle `@ff1`. Damit kann man obige Befehle z.B. als

```
A1 = quadl('ff1', a, b, TOL, ANZEIGE, n)
A1 = quadl(@ff1,  a, b, TOL, ANZEIGE, n)

A2 = dblquad('ff2', a, b, c, d, TOL, 'quadl', m, n)
A2 = dblquad(@ff2,  a, b, c, d, TOL, @quadl,  m, n)
```

schreiben. Die Funktionen müssen der Konvention für die Erstellung von Unterprogrammen folgen und müssen mit dem Befehl `feval` auswertbar sein.

```
feval('ff1', x, n)        feval(@ff1, x, n)
feval('ff2', x, y, m, n)  feval(@ff2, x, y, m, n)
```

Wichtig ist vor allem auch, dass sie für beliebige Vektoren `x` und `y` auszuwerten sind.

7.1.9 Globale Variablen

In MATLAB ist es möglich, ein Set von Variablen für eine Reihe von Funktionen global zugänglich zu machen, ohne diese Variablen durch die Inputliste zu übergeben. Dafür steht der Befehl `global var1 var2` zur Verfügung. Er muss in jeder Programmeinheit ausgeführt werden, wo diese Variablen zur Verfügung stehen sollen, d.h. die Variablen sind nicht automatisch überall verfügbar.

Das `global`-Statement soll vor allen ausführbaren Anweisungen in einem Skript oder einem `function`-Unterprogramm angeführt werden. Da diese Variablennamen in weiten Bereichen ihre Gültigkeit haben können, empfiehlt es sich längere und unverwechselbare Namen zu verwenden, damit sie sich nicht mit lokalen Variablennamen decken.

Mit dem Befehl `global` gibt es also neben den Input- und Outputlisten eine weitere Möglichkeit Informationen zwischen Skripts und Funktionen, bzw. zwischen Funktionen untereinander auszutauschen. Dies ist vor allem dann interessant, wenn Funktionen als Parameter übergeben werden und man sich beim Aufruf der "Zwischenfunktion" (z.B. `quadl`) keine Gedanken über die weiteren Parameter, die eventuell im Unterprogramm noch gebraucht werden, machen will.

Das Skript

```
global flag
k = 3;
flag = 'a';
A_a  = quadl(ifunc,0,1,[],[],k);
flag = 'b';
A_b  = quadl(ifunc,0,1,[],[],k);
clear global flag
```

berechnet mit der Funktion `ifunc`

```
function [y] = ifunc(x,k)
global flag
switch flag
case 'a', y = sin(k*x);
case 'b', y = cos(k*x);
otherwise, error('flag existiert nicht');
end
```

die Integrale über zwei verschiedene mathematische Funktionen.

Am Ende solcher Berechnungen sollte man in der übergeordneten Einheit diese Variablen wieder löschen. Das geschieht mit `clear global var1 var2`, damit verschwinden die Variablen aus allen lokalen Speicherbereichen und sind nirgendwo mehr zugänglich.

7.1.10 Beispiele

Einfaches Beispiel (`tfunc1.m`):

```
function f = tfunc1(x)
% Einfachste Funktion mit einer Input-Variablen und einer
% Output-Variablen.
%
% Berechnet die Funktion f(x) = x^2 * sin(x)
%
% Aufruf: f = tfunc1(x)
% Input:  x   double array x
% Output: f   double array x.^2 .* sin(x)

% Der erste Kommentarblock wird bei Aufruf des Befehls
```

```
%            help tfunc1
% angezeigt

% Hier beginnt nun die Berechnung
  f = x.^2 .* sin(x);
```

Beispiel mit mehreren Input- und Outputvariablen (tfunc2.m):

```
function [f1,f2] = tfunc2(x,a,b)
% Funktion mit drei Input-Variablen und zwei Output-Variablen.
%
% Berechnet die Funktionen f1(x) = a * x^2 * sin(x)
%                          f2(x) = a * x^2 * sin(x) + b * x
%
% Aufruf: [f1,f2] = tfunc2(x,a,b)
% Input:  x   double array
%         a   double scalar
%         b   double scalar
% Output: f1  double array   f1 = a * x.^2 .* sin(x)
%         f2  double array   f2 = a * x.^2 .* sin(x) + b * x

  f1 = a * x.^2 .* sin(x);
  f2 = f1 + b * x; % f1 verwendet um Rechenzeit zu sparen
```

Beispiel mit mehreren Input- und Outputvariablen, wobei Defaultwerte für einige Inputvariablen gesetzt werden. (tfunc2a.m):

```
function [f1,f2] = tfunc2a(x,a,b)
% Funktion mit drei Input-Variablen und zwei Output-Variablen.
% Setzen von Default-Werten.
%
% Berechnet die Funktionen f1(x) = a * x^2 * sin(x)
%                          f2(x) = a * x^2 * sin(x) + b * x
%
% Aufruf: [f1,f2] = tfunc2a(x,a,b)
% Input:  x   double array
%         a   double scalar, optional, default a=1
%         b   double scalar, optional, default b=2
% Output: f1  double array   f1 = a * x.^2 .* sin(x)
%         f2  double array   f2 = a * x.^2 .* sin(x) + b * x

% Die Variable nargin enthaelt nach dem Aufruf der Funktion tfunc2a
% die Anzahl der übergebenen Input-Variablen. Die Variable nargout
% enthält die Anzahl der Output-Variablen.
%
% z.B.:  [r1,r2] = tfunc2a([1:10],3,4) => nargin=3, nargout=2
%         r1     = tfunc2a([1:10],3)   => nargin=2, nargout=1
%                  tfunc2a             => nargin=0, nargout=0
%
% Diese Variablen kann man nun zum Steuern der Verhaltens der Funktion,
% zum Setzen von Defaultwerten und zur Entscheidung, welche
% Output-Variablen berechnet werden sollen, verwenden.

% Der Befehl isempty(a) überprüft ob die Variable a eine leeres Array []
% ist. Damit kann man auch den Defaultwert von a verwenden, obwohl man b
% eingibt:
% z.B.:  [r1,r2] = tfunc2a([1:10],[],4)
  if nargin<1, error('Aufruf mit [f1,f2] = tfunc2a(x,a,b)'); end
  if nargin<2,   a = 1; end, if isempty(a), a = 1; end
  if nargin<3,   b = 2; end, if isempty(b), b = 2; end
  f1 = a * x.^2 .* sin(x);
  if nargout>1, f2 = f1 + b * x; end
```

Beispiel mit mehreren Input- und Outputvariablen, wobei auch globale Variable verwendet werden. (tfunc2b.m):

```
function [f1,f2] = tfunc2b(x)
% Funktion mit drei Input-Variablen und zwei Output-Variablen. Verwendung
% von globalen Variablen für die Variablen a und b.
%
% Berechnet die Funktionen f1(x) = a * x^2 * sin(x)
%                          f2(x) = a * x^2 * sin(x) + b * x
%
% Aufruf: [f1,f2] = tfunc2b(x)
% Global: a   double scalar, optional, default a=1
%         b   double scalar, optional, default b=2
% Input:  x   double array
% Output: f1  double array   f1 = a * x.^2 .* sin(x)
%         f2  double array   f2 = a * x.^2 .* sin(x) + b * x

% Definition von globalen Variablen. Wurden diese vorher noch nicht
% definiert, existieren sie nach der Anweisung global als leere Arrays.
  global a b
  if nargin<1, error('Aufruf mit [f1,f2] = tfunc2a(x,a,b)'); end
  if isempty(a), a = 1; end
  if isempty(b), b = 2; end
  f1 = a * x.^2 .* sin(x);
  if nargout>1, f2 = f1 + b * x; end
```

Beispiel mit mehreren Input- und Outputvariablen mit Summation zur Berechnung von:

$$f_1(x) = \sum_{k=1}^{n} a_k x^2 \sin(a(k)x), \qquad f_2(x) = \sum_{k=1}^{n} a_k x^2 \sin(a(k)x) + b_k x$$

(tfunc2c.m):

```
function [f1,f2] = tfunc2c(x,a,b)
% Funktion mit drei Input-Variablen und zwei Output-Variablen.
% Setzen von Default-Werten.
%
% Bei diesem Beispiel können die Variablen a und b Vektoren sein.
%
% Berechnet die Funktionen f1(x) = a(1) * x^2 * sin(x)
%                                + a(2) * x^2 * sin(x)
%                                + ...
%                          f2(x) = a(1) * x^2 * sin(x) + b(1) * x
%                                + a(2) * x^2 * sin(x) + b(2) * x
%                                + ...
%
% Aufruf: [f1,f2] = tfunc2c(x,a,b)
% Input:  x   double array
%         a   double array, optional, default a=[1,2]
%         b   double array, optional, default b=[2,4]
% Output:                    Summation ueber alle k
%         f1  double array   f1 = a(k) * x.^2 .* sin(a(k)*x)
%         f2  double array   f2 = a(k) * x.^2 .* sin(a(k)*x) + b(k) * x

  if nargin<1, error('Aufruf mit [f1,f2] = tfunc2c(x,a,b)'); end
  if nargin<2,   a = [1,2]; end, if isempty(a), a = [1,2]; end
  if nargin<3,   b = [2,4]; end, if isempty(b), b = [2,4]; end
  % Output-Groessen werden mit 0 initialisiert
  f1 = zeros(size(x)); f2 = f1;
  % Summation über alle f1(k) und f2(k)
  for k = 1:length(a)
    h1 = a(k) * x.^2 .* sin(a(k)*x);
    h2 = h1 + b(k) * x;
    f1 = f1 + h1;
    f2 = f2 + h2;
  end
```

Beispiel mit Fallunterscheidung (tfunc3.m):

```
function [f1,f2] = tfunc3(f,x,a,b)
% Funktion mit vier Input-Variablen und zwei Output-Variablen.
% Die Variable f dient dabei zur Fallunterscheidung.
%
% Berechnet die Funktionen f1(x) = a * x^2 * f(x)
%                          f2(x) = a * x^2 * f(x) + b * x
%
% Aufruf: [f1,f2] = tfunc3(f,x,a,b)
% Input:  f   char   array,  {'sin', 'cos', 'tan', 'cot'}, default 'sin'
%         x   double array
%         a   double scalar, optional, default a=1
%         b   double scalar, optional, default b=2
% Output: f1  double array   f1 = a * x.^2 .* f(x)
%         f2  double array   f2 = a * x.^2 .* f(x) + b * x

  if nargin<2, error('Aufruf mit [f1,f2] = tfunc3(f,x,a,b)'); end
  if isempty(f), f = 'sin'; end;
  if nargin<3, a = 1; end; if isempty(a), a = 1; end;
  if nargin<4, b = 2; end; if isempty(b), b = 2; end;

  % Die Fallunterscheidung wird mit einer switch-case-Konstruktion
  % durchgeführt, wobei die die String-Variable f als Schalter dient.
  switch f
   case 'sin'
    f1 = a * x.^2 .* sin(x);
   case 'cos'
    f1 = a * x.^2 .* cos(x);
   case 'tan'
    f1 = a * x.^2 .* tan(x);
   case 'cot'
    f1 = a * x.^2 .* cot(x);
   otherwise
    error(['Fall ',f,' existiert nicht!']);
  end
  if nargout>1, f2 = f1 + b * x; end
```

Beispiel mit Fallunterscheidung und automatischer Konstruktion von Funktionsaufrufen (tfunc3a.m):

```
function [f1,f2] = tfunc3a(f,x,a,b)
% Funktion mit vier Input-Variablen und zwei Output-Variablen.
% Die Variable f dient dabei zur Fallunterscheidung.
%
% Berechnet die Funktionen f1(x) = a * x^2 * f(x)
%                          f2(x) = a * x^2 * f(x) + b * x
%
% Aufruf: [f1,f2] = tfunc3a(f,x,a,b)
% Input:  f   char   array,  {'sin', 'cos', 'tan', 'cot'}, default 'sin'
%         x   double array
%         a   double scalar, optional, default a=1
%         b   double scalar, optional, default b=2
% Output: f1  double array   f1 = a * x.^2 .* f(x)
%         f2  double array   f2 = a * x.^2 .* f(x) + b * x

  if nargin<2, error('Aufruf mit [f1,f2] = tfunc3a(f,x,a,b)'); end
  if isempty(f), f = 'sin'; end;
  if nargin<3, a = 1; end; if isempty(a), a = 1; end;
  if nargin<4, b = 2; end; if isempty(b), b = 2; end;

  % Die String-Variable f wird nun einerseits als Schalter, aber auch zur
  % Konstruktion der Funktion verwendet.
  switch f
   case {'sin', 'cos', 'tan', 'cot'}
    % Zusammensetzen einer Zeichenkette [s1,s2,s3]
    e_string = ['a * x.^2 .* ',f,'(x)'];
    disp(['Berechnung mit: ',e_string]);
    % Mit eval kann man den Inhalt einer Zeichenkette als Kommando
    % ausführen.
    % z.B.: f = eval('3*x+2'); equivalent mit f = 3*x+2;
    f1 = eval(e_string);
   otherwise
    error(['Fall ',f,' nicht erlaubt!']);
  end
  if nargout>1, f2 = f1 + b * x; end
```

Beispiel mit Fallunterscheidung, automatischer Konstruktion von Funktionsaufrufen und rekursivem Aufruf. (tfunc3b.m):

```
function [f1,f2] = tfunc3b(varargin)
% Funktion mit vier Input-Variablen und zwei Output-Variablen.
% Die Variable varargin ist eine Zelle, die alle übergebenen
% Input-Variablen enthält.
%
% Berechnet die Funktionen f1(x) = a * x^2 * f(x)
%                          f2(x) = a * x^2 * f(x) + b * x
%
% Aufruf: [f1,f2] = tfunc3b(f,x,a,b)
% Input:  f   char   array,  {'sin', 'cos', 'tan', 'cot'}, default 'sin'
%         x   double array
%         a   double scalar, optional, default a=1
%         b   double scalar, optional, default b=2
% Output: f1  double array   f1 = a * x.^2 .* f(x)
%         f2  double array   f2 = a * x.^2 .* f(x) + b * x

  if nargin<1, error('Aufruf mit [f1,f2] = tfunc3b(f,x,a,b)'); end
  if ~ischar(varargin{1}) % Erstes Element der Zelle kein String
    [f1,f2] = tfunc3b('sin',varargin{:}); % Rekursiver Aufruf von tfunc3b
    % Übergabe von 'sin' und aller Parameter vom ersten Aufruf.
    return % Beendet ersten Aufruf der Funktion
  end

  f = varargin{1}; % Erster Übergabeparameter
  if nargin<2,
    error('Aufruf mit [f1,f2] = tfunc3b(f,x,a,b)');
  else
    x = varargin{2}; % Zweiter Übergabeparameter
  end

  if nargin<3, a = 1; else, a = varargin{3}; end
  if isempty(a), a = 1; end
  if nargin<4, b = 2; else, b = varargin{4}; end
  if isempty(b), b = 2; end

  % Die String-Variable f wird nun einerseits als Schalter, aber auch zur
  % Konstruktion der Funktion verwendet.
  switch f
   case {'sin', 'cos', 'tan', 'cot'}
    % Zusammensetzen einer Zeichenkette [s1,s2,s3]
```

```
  e_string = ['a * x.^2 .* ',f,'(x)'];
  disp(['Berechnung mit: ',e_string]);
  % Mit eval kann man den Inhalt einer Zeichenkette als Kommando
  % ausführen.
  % z.B.: f = eval('3*x+2'); equivalent mit f = 3*x+2;
  f1 = eval(e_string);
 otherwise
  error(['Fall ',f,' nicht erlaubt!']);
end
if nargout>1, f2 = f1 + b * x; end
```

Kapitel 8

Polynome

8.1 Grundlagen

In MATLAB werden Polynome durch ihren Koeffizientenvektor repräsentiert, d.h. der Vektor `p=[p1,p2,...,pn]` stellt das Polynom,

$$p_1 x^{n-1} + p_2 x^{n-2} + p_3 x^{n-3} + \cdots + p_n \ , \tag{8.1}$$

dar. Für ein Polynom vom Grad $n-1$ braucht man daher einen Vektor der Länge n. Für die Auswertung ein solchen Polynoms für verschiedene Werte von x,

$$y_i = p_1 x_i^{n-1} + p_2 x_i^{n-2} + p_3 x_i^{n-3} + \cdots + p_n \ , \tag{8.2}$$

stellt MATLAB die Funktion `y=polyval(p,x)` zur Verfügung. Die Variable x kann dabei ein Skalar, ein Vektor, bzw. eine Matrix sein, y hat dann immmer die gleiche Größe wie x.

Will man also z.B. das Polynom

$$y_i = x_i^3 + 2x_i^2 + x_i + 3 \ , \tag{8.3}$$

darstellen, kann man Folgendes tun:

```
p = [1,2,1,3];
x = linspace(-2,2,30); y = polyval(p,x);
plot(x,y,'b');
```

Die Auswertung erfolgt natürlich mit dem Horner-Schema, das hier am Beispiel eines Polynomes dritten Grades demonstriert wird,

$$y_i = ((p_1 x + p_2)\, x + p_3)\, x + p_4 \ , \tag{8.4}$$

wobei die Anzahl der Multiplikationen pro x-Wert von $m(m+1)/2$ auf m reduziert wird, die Anzahl der Additionen bleibt mit m gleich. Daraus folgt, dass das Horner-Schema viel effizienter ist.

Es gibt auch eine Auswertung für $n \times n$ Matrizen, `polyvalm`, wobei alle Multiplikationen als Matrixmultiplikationen aufgefasst werden.

8.2 Nullstellen und charakteristische Polynome

Die Nullstellen eines Polynoms können mit dem Befehl `r=roots(p)` gefunden werden. Umgekehrt erzeugt der Befehl `p=poly(r)` das Polynom `p`, wenn `r` ein Vektor von Nullstellen ist. Für Vektoren sind `roots` und `poly` inverse Funktionen bis auf Skalierungsfaktoren, Reihenfolge der Nullstellen und Rundungsfehler.

Der Aufruf von `c=poly(A)`, wobei $\mathbf{A}$ eine $n \times n$ Matrix sein muss, liefert das charakteristische Polynom c der Matrix $\mathbf{A}$. Das charakteristische Polynom ergibt sich aus folgender Determinante

$$\det\left(\mathbf{A} - \lambda\mathbf{I}\right) \ , \tag{8.5}$$

wobei λ die Eigenwerte der Matrix $\mathbf{A}$, bzw. die Nullstellen des charakteristischen Polynoms c sind.

Dies soll am Beispiel von

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \ , \ \mathbf{c} = \det \begin{bmatrix} 2-\lambda & 1 \\ 2 & 3-\lambda \end{bmatrix} \ , \tag{8.6}$$

demonstriert werden, wobei sich hier als Lösung

$$\mathbf{c} = (2-\lambda)(3-\lambda) - 2 = \lambda^2 - 5\lambda + 4 \tag{8.7}$$

ergibt. Die Darstellung in MATLAB ergibt `c=[1,-5,4]` und die Nullstellen des charakteristischen Polynoms liegen bei $\lambda_{1,2} = \{1, 4\}$.

Das Pascal'sche Dreieck

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & \\ 1 & & 5 & & 10 & & 10 \\ & 1 & & 6 & & 15 & \end{array} \tag{8.8}$$

kann auch in Form der Pascal'schen Matrix, hier für $n = 4$ mit z.B. dem MATLAB-Befehl `P=pascal(4)`

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{bmatrix} \tag{8.9}$$

dargestellt werden. Das charakteristische Polynom kann nun mit dem Befehl `p=poly(P)` erzeugt werden und ergibt `[1,-29,72,-29,1]`, was folgendem Polynom entspricht

$$p(x) = x^4 - 29x^3 + 72x^2 - 29x + 1 \; . \tag{8.10}$$

Pascal'sche Matrizen haben die kuriose Eigenschaft, dass der Vektor der Koeffizienten des charakteristischen Poylnoms "palindromic" ist, d.h. er ergibt das Selbe von vorne und von hinten gelesen.

Evaluiert man das charakteristische Polynom nun im Sinne der Matrixmultiplikation, so erhält man mit `R=polyvalm(p,P)`

$$\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \; , \tag{8.11}$$

d.h. die Nullmatrix. Dies ist eine Folge des Cayley-Hamilton Theorems, dass besagt das eine Matrix ihre eigene charakteristische Gleichung erfüllt.

8.3 Addition von Polynomen

MATLAB stellt keinen Befehl für die Addition von Polynomen bereit. Eine solche Routine muss man sich als kleine Übungsaufgabe selbst erstellen. Da es sich bei der Addition von Polynomen um die Addition von Vektoren unterschiedlicher Länge handelt, kann nicht einfach der Befehl `plus` verwendet werden. Man muss also vorher den kürzeren der beiden Vektoren am Beginn mit Nullen auffüllen, um die gleiche Länge bei der Verwendung von `plus` zu gewährleisten.

Die Ergänzung mit Nullen kann man durch Zusammenhängen von Vektoren erreichen. Der Befehl `zeros(1,l)` erzeugt einen Zeilenvektor der der Länge l für $l > 0$ und ein leeres Array für $l \leq 0$. Dies kann man sich in diesem Fall zu Nutze machen.

Ausserdem eignet sich dieses Beispiel bestens für die Verwendung variabler Inputlisten. Dies hat den Vorteil, dass man dann beliebig viele Polynome mit einem Aufruf addieren kann. Dafür sind die Variablen `nargin` und `varargin` bestens geeignet:

```
function p = polyadd(varargin)
p = [];
```

```
for k = 1:nargin
   p1 = varargin{k}; p1 = p1(:).';  % k-tes Polynom, Zeilenvektor
   l  = length(p); l1 = length(p1); % Längen
   p  = ...
end
```

Schön ist natürlich auch, wenn man alle führenden Nullen beseitigt, so dass maximal eine überbleibt, wenn das Ergebnis das Polynom `p=[0]` ist. Dazu kann man sich des Befehls `find` bedienen und nach dem ersten Element von p suchen, das ungleich Null ist (`min(find(p~=0))`).

8.4 Differentiation und Integration von Polynomen

Für die Differentiation von Polynomen steht der Befehl `polyder` zur Verfügung. Er kann in verschiedenen Formen verwended werden:

```
k = polyder(p)
k = polyder(a,b)
[z,n] = polyder(b,a)
```

Im zweiten Fall wird die Ableitung des Produkts der Polynome a und b berechnet und im dritten Fall erhält man den Zähler z und den Nenner n der Ableitung des Polynomquotienten b/a. Will man also z.B. die Extremwerte eines Polynoms in sortierter Reihenfolge bestimmen, kann man die x- und y-Werte der Extremwerte folgendermaßen bestimmen:

```
p   = [1,1,-2,4];
e_x = sort( roots( polyder(p) ) );
e_y = polyval( p, e_x );
```

Der Befehl `sort(x)` führt dabei die Sortierung nach der Größe von x durch.

Die Integration von Polynomen erfolgt mit dem Befehl `polyint(p)` oder `polyint(p,k)`, wobei im zweiten Fall das Skalar k als Konstante der Integration verwendet wird. Ohne Angabe von k wird dafür der Wert Null verwendet.

Will man also das Integral $\int_1^2 p(x)dx$ ausführen, kann man Folgendes machen

```
p    = [1,1,1];            u = 1; o = 2;
pint = polyint(p);
r = diff(polyval(pint,[u,o]));
```

Der Befehl `diff` führt bei einem Vektor v der Länge n die Differenzberechnung $v_{i+1} - v_i$ durch, wodurch sich ein Vektor der Länge $n-1$ ergibt.

8.5 Konvolution und Dekonvolution von Polynomen

Der Befehl `w=conv(u,v)` führt die Konvoluation der Polynome u und v aus. Algebraisch ist das das Gleiche wie die Multiplikation der Polynome, deren Koeffizienten in u und v gegeben sind. Die Multiplikation $(x+1)(x-1)$ wird also in MATLAB durchgeführt mit

```
u = [1,1]; v=[1,-1];
w=conv(u,v)                    w = [1,0,-1]
```

womit sich das Polynom $x^2 - 1$ ergibt.

Als Dekonvolution bezeichnet man die Division von Polynomen. Der MATLAB-Befehl `[q,r] = deconv(v,u)` dekonvolviert den Vektor u aus dem Vektor v, d.h. es wird der Quotient v/u gebildet und in q retourniert. Ein eventuelles Restpolynom findet sich in r wieder. Die Division von v durch u ergibt z.B.

$$\begin{aligned} v(x) &= & x^3 + 2x^2 + 3x + 4 \\ u(x) &= & x + 2 \\ q(x) = v(x)/u(x) &= & x^2 + 3 \\ r(x) &= & -2 \end{aligned} \tag{8.12}$$

was in MATLAB in folgender Form durchgeführt werden kann:

```
v = [1,2,3,4]; u = [1,2];
[q,r] = deconv(v,u);

    q = [1,0,3]    r = [0,0,0,-2]

v = conv(q,u)+r
```

Hier wurde in der letzten Zeile die Umkehroperation für Division von Polynomen gezeigt.

8.6 Fitten mit Polynomen

Im Allgemeinen bezeichnet man das Ermitteln von Funktionen, die am Besten einen gegebenen Verlauf von daten entsprechen, als Fitten der Daten. Dabei gibt man eine Modellfunktion vor, die im einfachsten Fn denall eine Gerade oder ein Polynom der Ordnung k ist. Unter der Annahme, dass die Daten in den gleich langen Vektoren x und y vorliegen, wird im sogenannten "Least Squares" Verfahren die Summe der Abstandsquadrate minimiert.

Bei Vorliegen von m Datenpunkten und unter der Annahme dass die Modellfunktion mit $p(x)$ bezeichnet wird, kann die Summe der Abstandsquadrate geschrieben werden als

$$q = \sum_{j=1}^{m} (y_j - p(x_j))^2 \stackrel{!}{=} \text{Min} , \tag{8.13}$$

wofür ein minimaler Wert gesucht wird.

Wenn man sich auf Polynome als Modellfunktionen beschränkt, kann man für diese Aufgabe den MATLAB-Befehl `p=polyfit(x,y,n)` verwenden, der in p die Koeffizienten des "besten" Polynoms vom Grad n, d.h. einen Vektor der Länge $n+1$, retourniert. Dieses Polynom kann dann mit `polyval` im interessanten Bereich ausgewertet werden. Diese Vorgangsweise macht natürlich nur Sinn, wenn die Modellfunktion zu den Daten "passt".

Kapitel 9

Zeichenketten

9.1 Grundlagen

Der MATLAB-Datentyp `char` dient zur Speicherung von ASCII-Zeichen. Dies kann auch in ein- bzw. mehrdimensionalen Feldern geschehen. Es besteht jedoch die Einschränkung, dass alle Zeilen die gleiche Anzahl von Zeichen enthalten müssen, ähnlich wie alle Zeilen einer Matrix die gleiche Anzahl von Elementen beinhalten müssen.

MATLAB speichert Zeichenketten, auch Strings genannt, als Felder von ASCII-Werten. Diese liegen z.B. zwischen `48` und `57` für die Zahlen 0 bis 9, zwischen `65` und `90` für die Großbuchstaben und zwischen `97` und `122` für die Kleinbuchstaben. Neben anderen Sonderzeichen hat der horizontale Tabulatur `HT` den Wert `9`, der Zeilenumbruch `LF` den Wert `10`, und das Leerzeichen `SP` den Wert `32`.

Die ASCII-Werte `A` und die Zeichenketten `S` können mit den Befehlen `S=char(A)` bzw. `A=double(S)` ineinander umgewandelt werden.

Die Erzeugung von Strings erfolgt mit `s1='sin'` bzw. mit `s2=char('(x)')`. Ein horizontales Aneinanderfügen erfolgt mit `[s1,s2]` bzw. mit `strcat(s1,s2)`, eine Anordnung in mehreren Zeilen kann in Prinzip wie bei Vektoren mit `[s1;s2]` erfolgen, wenn beide Strings gleich lang sind. Besser ist jedoch die Verwendung von `char(s1,s2)` oder `strvcat(s1,s2`, da hier zu kurze Zeilen am Zeilenende durch Leerzeichen aufgefüllt werden.

Ist das Auffüllen mit Leerzeichen nicht erwünscht, muss auf Objekte des Typs `cell` zurückgreifen.

```
z = {'Erste Zeile', ...
     'Zweite Zeile'}
```

Mit `z{1}` bzw. `z{2}` kann man dann wieder auf die einzelnen Elemente der Zelle zugreifen. Auch hier gibt es Funktionen zur Umwandlung, `s=char(z)` von der Zelle zum String, bzw. `z=cellstr(s)`.

Eine Zusammenstellung interessanter Umwandlungsroutinen für Strings:

`s = num2str(d)`	Zahlen in Strings
`s = num2str(d,n)`	Zahlen in Strings; n Stellen
`s = int2str(d)`	Integer in Strings (runden)
`s = mat2str(m)`	Matrizen in Strings mit []
`s = mat2str(m,n)`	Matrizen in Strings; n Stellen
`d = str2num(s)`	String in Zahlen
	Leeres Array `[]` falls keine Zahl
`d = str2double(s)`	String in eine double-Zahl
	`NaN` falls nicht möglich
`sm = str2mat2(s)`	String in Stringmatrix
	Leerzeichen erzeugt neue Zeile
`z = cellstr(s)`	Strings in Zellen
`s = char(z)`	Zellen in Stringmatrix
`A = double(s)`	Strings in ASCII Werte
`s = char(A)`	ASCII Werte in Strings

Darüber hinaus gibt es eine Reihe von Befehlen, die Strings umwandeln, in Strings suchen, Strings vergleichen usw.

`s=blanks(n)`	Erzeugt String der Länge n mit Leerzeichen
`s=deblank(s)`	Entfernt Leerzeichen am Beginn
`s=lower(s)`	Umwandlung in Kleinbuchstaben
`s=upper(s)`	Umwandlung in Großbuchstaben
`l=ischar(s)`	`TRUE` wenn String
`l=isletter(s)`	`TRUE` wenn Buchstabe
`l=strcmp(s1,s2)`	`TRUE` wenn gleich
`l=strcmpi(s1,s2)`	`TRUE` wenn gleich
	ignoriert Groß/Kleinschreibung
`l=strncmp(s1,s2,n)`	`TRUE` wenn die ersten n gleich
`l=strncmpi(s1,s2,n)`	`TRUE` wenn die ersten n gleich
	ignoriert Groß/Kleinschreibung
`i=strfind(s1,s2)`	Positionen von s2 im String s1
`i=findstr(s1,s2)`	Positionen des kürzeren im längeren
`i=strmatch(s,sm)`	Zeilen in Stringmatrix oder Zelle,
	die mit String s beginnen
`i=strmatch(s,sm,'exact')`	Zeilen in Stringmatrix oder Zelle,
	die mit String s exakt übereinstimmen

Kapitel 10

Anwendungen

10.1 Kurvenanpassung - Fitten

Kurvenanpassung, oder auch Fitten genannt, ist eine Technik mit der man versucht, eine gegebene mathematische Modellfunktion bestmöglich an Datenpunkte anzupassen. Der einfachste Fall ist wohl die Bestimmung einer Ausgleichsgeraden, wo die Koeffizienten k und d des Polynoms ersten Grades $f(x, k, d) = kx + d$ so bestimmt werden, dass die Summe der Abstandsquadrate von den Datenpunkten den kleinstmöglichen Wert annimmt.

Diese Minimierung der Summe der Abstandsquadrate bezeichnet man als "Least Squares"-Verfahren. Bei Vorliegen von m Datenpunkten (x_j, x_j) und unter der Annahme, dass die Modellfunktion mit $f(x, a)$ bezeichnet wird, kann die Summe der Abstandsquadrate geschrieben werden als

$$q = \sum_{j=1}^{m} (y_j - f(x_j, a))^2 \stackrel{!}{=} \text{Min} . \tag{10.1}$$

Diese Summe q der Abstandsquadrate soll minimiert werden, d.h. man sucht nach den besten Parametern a der Funktion $f(x, a)$, wobei a am Beispiel der Ausgleichsgeraden der Vektor $[k, d]$ ist.

Im Wesentlichen sind zwei unterschiedliche Klassen von Modellfunktionen zu unterscheiden, solche die lineare Funktionen der Parameter a_i sind und solche die nichtlineare Funktionen der Parameter a_i sind. Lineare Funktionen sind Polynome bzw. Funktionen, die man im weitesten Sinne als verallgemeinerte Polynome bezeichnen

könnte

$$f(x,a) = \sum_{i=0}^{n} a_i x^i \ , \tag{10.2}$$

$$f(x,a) = a_0 + a_1 x + a_2 x^2 \ , \tag{10.3}$$

$$f(x,a) = \sum_{i=0}^{n} a_i f_i(x) \ , \tag{10.4}$$

$$f(x,a) = a_0 + a_1 \sin x + a_2 \cos x \ . \tag{10.5}$$

Manche Modellfunktionen können z.B. durch Logarithmierung linearisiert werden

$$f(x,a) = a_0 \exp\left(-a_1 x\right) \qquad \Rightarrow \hat{f}(x,a) = \ln a_0 - a_1 x \ , \tag{10.6}$$

$$f(x,a) = a_0 x^{a_1} \qquad \Rightarrow \hat{f}(x,a) = \ln a_0 + a_1 \ln x \ . \tag{10.7}$$

Typische nichtlineare Funktionen sind solche, wo die Parameter z.B. als Argument von trigonometrischen Funktionen (Frequenz, Phase) vorkommen oder andere Funktionen mit komplexeren funktionalen Zusammenhang

$$f(x,a) = a_0 \sin\left(a_1 x + a_2\right) \ , \tag{10.8}$$

$$f(x,a) = a_0 \exp\left(\frac{-(x-a_1)^2}{a_2}\right) \ , \tag{10.9}$$

$$f(x,a) = \frac{a_0}{a_1 + a_2 x} \ . \tag{10.10}$$

10.1.1 Auswahl der Modellfunktion

Ein wichtiger Schritt bei der Kurvenanpassung ist die Auswahl der Modellfunktion. Wann immer es möglich ist, lässt man sich von einem zugrunde liegenden physikalischen Modell bei der Auswahl leiten:

- Schwingung - Kombination trigonometrischer Funktionen
- Dämpfung, Abklingverhalten - Exponentialfunktion
- Theoretisches Modell

Ist die Wahl auf ein lineares Modell in Bezug auf die Parameter gefallen, kann man den Anweisungen in Abschnitt 10.1.2 folgen, sonst den Anweisungen im Abschnitt 10.1.4.

10.1.2 Lineares Fitten

Um lineares Fitten zu verstehen, sollte man sich vergegenwärtigen, wie man ein Polynom exakt durch m-Datenpunkte $(x_j, y_j$ legen kann. Man benötigt dazu im Normalfall ein Polynom $(m-1)$-ten Grades mit m Koeffizienten. Die Koeffizienten müssen dabei folgendes bestimmtes Gleichungssystem erfüllen

$$\sum_{i=0}^{m-1} a_i x_j^i = y_j \ , \ j = 1 \dots m \ , \tag{10.11}$$

welches in Matrixform als

$$X_{ji} a_i = y_j \tag{10.12}$$

geschrieben werden kann. In den m Spalten der Matrix $X_{ji} = x_j^i$ stehen somit die Spaltenvektoren x^{m-1}, x^{m-2} $\dots$ x^0. Dieses bestimmte Gleichungssystem kann nun nach den Regeln der linearen Algebra (siehe Abschnitt 5.7) gelöst werden. In MATLAB lautet die Lösung mit Hilfe der Matrix-Links-Division `a=X\y`, wobei y ein Spaltenvektor sein muss.

Im Falle des Fittens hat man es normalerweise mit einer größeren Anzahl von Datenpunkten zu tun und möchte in den seltensten Fällen Modellfunktionen mit der gleichen Anzahl von Parametern verwenden. Damit hat man es mit einem überbestimmten Gleichungssystem zu tun, dass nicht mehr exakt gelöst werden kann. In MATLAB kann man für die Lösung eines solchen überbestimmten Gleichungssystems aber die gleiche Syntax verwenden. Sobald ein Gleichungssystem überbestimmt ist, löst MATLAB bei Verwendung des Befehls `a=X\y` das Gleichungssystem im Sinne des "Least-Squares"-Verfahrens entsprechend der Gleichung 10.1.

Die Unterscheidung zwischen linearen und nichtlinearen Modellfunktionen ist deswegen so wichtig, weil in der numerischen Umsetzung wesentliche Unterschiede bestehen. Im Fall von linearen Funktionen kann das Minimierungsproblem in Gleichung 10.1 immer in ein exakt lösbares lineares Gleichungssystem überführen. Wie man sich leicht überzeugen kann, erhält man für $k = 0 \dots n$ aus $\frac{\partial q}{\partial a_k} = 0$ das lineare Gleichungssystem

$$X_{ki} a_i = b_k \ , \tag{10.13}$$

$$X_{ki} = \sum_{j=1}^{m} f_i(x_j) f_k(x_j) \ , \tag{10.14}$$

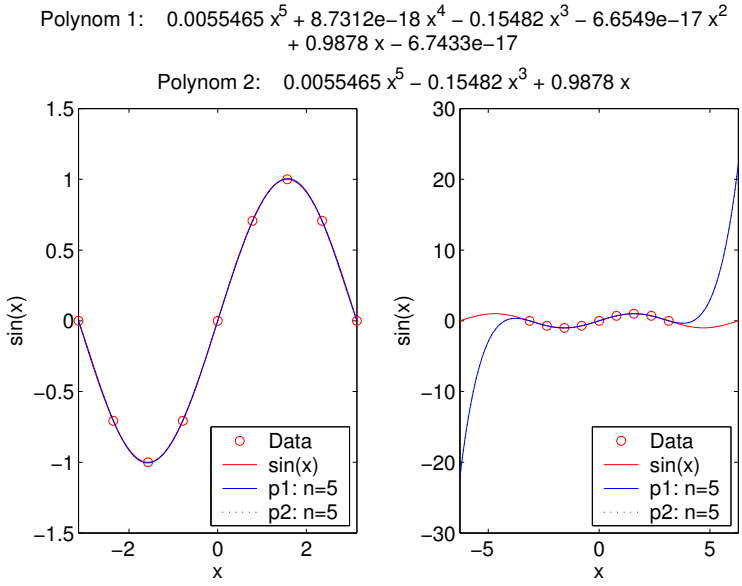
$$b_k = \sum_{j=1}^{m} y_j f_k(x_j) \ . \tag{10.15}$$

Diesen Vorteil hat man bei einem nichtlinearen Zusammenhang natürlich nicht. In diesem Fall ist man dann auf näherungsweise Verfahren zur Minimierung von Gleichung 10.1 angewiesen.

10.1.2.1 Polynom-Fit

Hat man sich für ein Polynom vom Grad n entschieden, kann man die MATLAB-Funktion `polyfit` verwenden. Die Verwendung soll hier am Beispiel eines Polynomfittes der Sinusfunktion gezeigt werden.

```
xd = linspace(-pi,pi,9); yd = sin(xd); grad = 5;
p1 = polyfit(xd,yd,grad);
x  = linspace(-pi,pi,500); y1 = polyval(p1,x);
subplot(1,2,1);plot(xd,yd,'ro',x,sin(x),'r-',x,y1,'b-');
x  = linspace(-2*pi,2*pi,500); y1 = polyval(p1,x);
subplot(1,2,2);plot(xd,yd,'ro',x,sin(x),'r-',x,y1,'b-');
```



Diese Darstellung demonstriert drei interessante Aspekte:

- Der Fit mit einem Polynom 5-ten Grades ist innerhalb des Datenbereichs sehr gut.
- Außerhalb des Datenbereichs bricht die gute Übereinstimmung sehr rasch zusammen, da Polynome natürlich nicht die beste Modellfunktion für Schwingungen sind. Der Grund dafür ist, dass alle Polynome für $x \to \pm\infty$ nach $\pm\infty$ "explodieren".
- Wie auf Grund der Reihenentwicklung für den Sinus nicht anders zu erwarten, sind die Koeffizienten für gerade Potenzen von x nahezu Null.

Will man an diesem Beispiel die Koeffizienten für die geraden Potenzen von vorne herein auf Null setzen, kann man `polyfit` nicht verwenden und muss sich auf das Lösen von überbestimmten Gleichungssystemen besinnen.

```
X = [xd(:).^5,xd(:).^3,xd(:)];
b = yd(:);
a = X \ b;
p2 = zeros(1,grad+1); p2(1:2:end) = a;
```

Hier werden nach Lösen des Gleichungssystems mit `a = X\b` die drei erhaltenen Koeffizienten an den richtigen Stellen im Polynom `p2` gespeichert.

10.1.2.2 Allgemeiner linearer Fit

Im allgemeinen Fall muss es sich nun nicht um Polynome handeln. Im folgenden Beispiel soll die Funktion $f(x, a)$, die als

$$f(x, a) = a_1 f_1(x) + a_2 f_2(x) \ , \tag{10.16}$$
$$f_1(x) = \exp(-0.2\ x) \ , \tag{10.17}$$
$$f_2(x) = \sin(4x) \exp(-0.4\ x) \ , \tag{10.18}$$

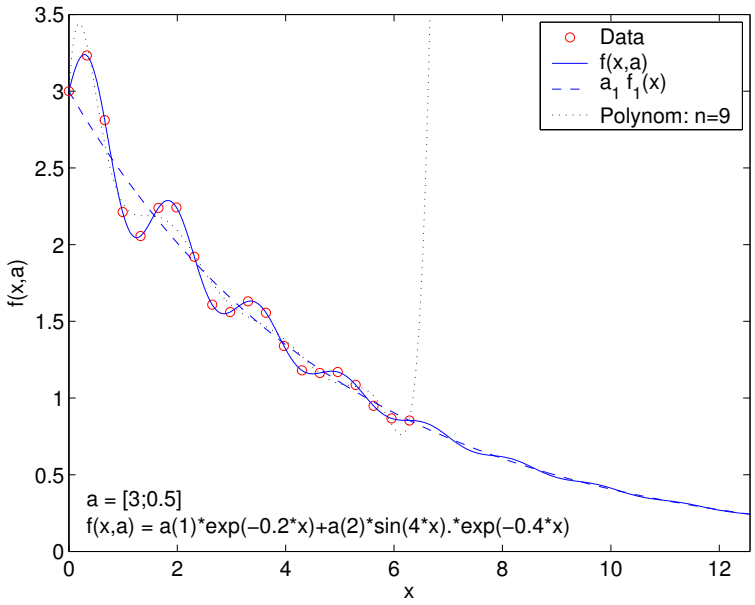
gegeben ist, an die Datenpunkte in den Vektoren `xd` und `yd` angepasst werden. Zuerst kann man in diesem Fall die Funktionen als `inline`-Funktionen definieren:

```
f1c = 'exp(-0.2*x)';
f2c = 'sin(4*x).*exp(-0.4*x)';
fc  = ['a(1)*',f1c,'+a(2)*',f2c];
f1  = inline(f1c,'x');
f2  = inline(f2c,'x');
f   = inline(fc,'x','a');
```

Um das lineare Gleichungssystem nun lösen zu können, muss man die Koeffizientenmatrix `X` und den Inhomegenitätsvektor `b` definieren und dann die Lösung für die Koeffizienten `a` bestimmen:

```
X  = [f1(xd(:)),f2(xd(:))];
b  = yd(:);
a  = X\b;
```

Mit diesen Daten kann man nun die Funktionen darstellen.



Hier fällt nun auf, dass

- bei passender Modellfunktion die Kurve auch außerhalb des Datenbereichs sich "vernünftig" verhält, und dass
- der zusätzlich eingezeichnete Polynomfit innerhalb der Daten nicht sehr gut liegt, und dass
- er natürlich außerhalb der Daten das gewohnte Verhalten zeigt.

10.1.3 Exponentieller Fit

Beim radioaktiven Zerfall folgt die Intensität der Strahlung folgendem Gesetz

$$I(t) = I_0 \exp\left(-\frac{t}{\tau}\right) \ , \tag{10.19}$$

wobei I_0 die Anfangsintensität und τ die Halbwertszeit ist. Logarithmiert man die Gleichung, kommt man zu folgender Darstellung

$$\hat{I}(t) = a_1 t + a_2 \ , \tag{10.20}$$

$$\hat{I} = \ln I \ , \tag{10.21}$$

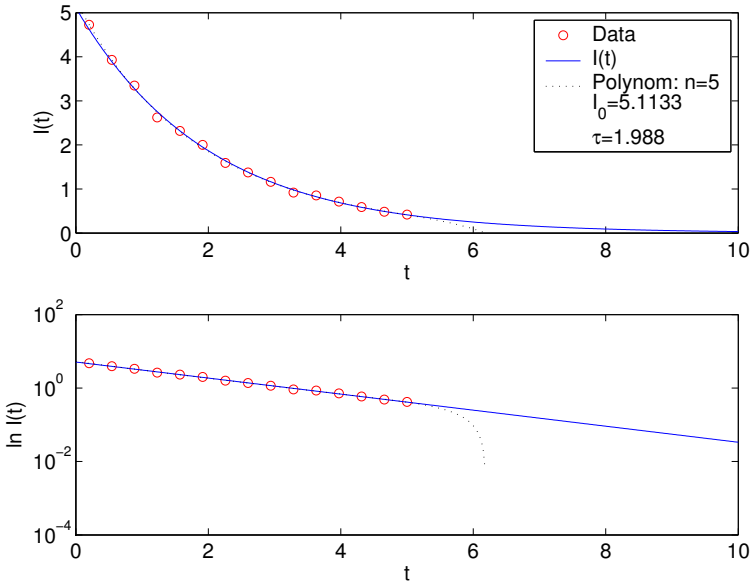
$$a_1 = -\frac{1}{\tau} \ , \tag{10.22}$$

$$a_2 = \ln I_0 \ , \tag{10.23}$$

Vorausgesetzt die Zeit- und die Intensitätswerte sind in den Variablen `t` und `I` gespeichert, kann man nun wieder das Gleichungssystem aufbauen und lösen. Hier sieht man auch, dass wenn man ein konstantes Glied bestimmen will, die Matrix `X` einfach eine Reihe mit Einsen enthalten muss:

```
X = [t(:),ones(size(t(:)))];
b = log(I(:));
a = X\b;
tau = -1 / a(1);
I0  = exp(a(2));
```

Nach Durchführen des Fits (Lösen des Gleichungssystems) kann man natürlich dann wieder die interessierenden Größen τ und I_0 berechnen.



Der zusätzlich berechnete Polynomfit erweist sich natürlich wieder als untauglich.

10.1.4 Nichtlineares Fitten

Im Falle einer Modellfunktion, die eine nichtlineare Funktion in den Modellparametern ist, muss man nun zu anderen Methoden greifen. Als einfachstes Beispiel soll hier eine Schwingung mit Amplitude A, Frequenz ω und Phase ϕ verwendet werden, die durch folgenden Zusammenhang gegeben ist

$$y(t) = A \sin(\omega t + \phi) \ . \tag{10.24}$$

Für eine Umsetzung des Problems in MATLAB muss man nun eine MATLAB-Funktion oder eine `inline`-Funktion schreiben, die den funktionalen Zusammenhänge wiedergibt:

```
fc = 'a(1)*sin(a(2)*t+a(3))';
f  = inline(fc,'a','t');
```

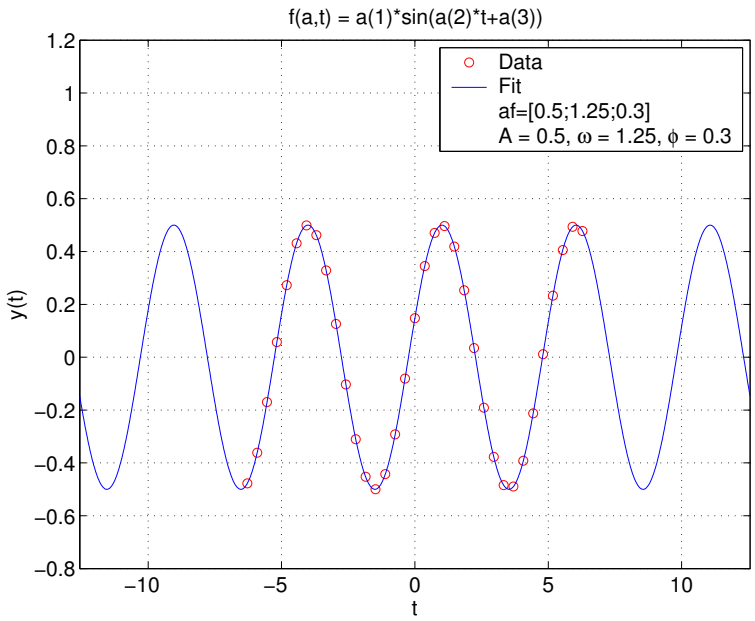
Wichtig dabei ist, dass alle Parameter (hier A, ω, ϕ) in einen Vektor zusammengefasst werden (hier `a`), und dass dieser Vektor an erster Stelle in der Übergabeliste steht.

Der Funktionsaufruf `y=f(a,t)` muss also für jeden Parametervektor `a` und jeden Zeitvektor `t` die Auslenkung `y` liefern, wobei `y` die gleiche Größe wie `t` haben muss.

In diesem Beispiel liegen die Datenpunkte wieder als Vektoren `td` und `yd` vor. Im Unterschied zum nichtlinearen Fitten braucht man nun aber auch einen Startwert für die Parameter um MATLAB mitzuteilen, wo man ungefähr die Lösung erwartet. Danach kann man mit dem MATLAB-Programm nlinfit den Fit durchführen. Diese Routine stammt aus dem MATLAB-Statistik Toolbox, ist also beim Grundpaket nicht installiert.

```
as = [0.8,1,0.2];
af = nlinfit(td,yd,f,as)
t = linspace(-4*pi,4*pi,1000);
y = f(af,t);
```

Damit kann man sich dann mit dem Zeitvektor `t` die Modellfunktion berechnen und zusammen mit den Daten darstellen.



Warum braucht man nun einen Startwert. Im Unterschied zum linearen Fitten, wo das zugehörige Gleichungssystem immer eine eindeutige Lösung besitzt, die direkt ermittelt werden kann, benötigt man hier ein iteratives Verfahren. Dabei wird ausgehend von einem Startwert das Minimum einer Funktion gesucht, in dem man

sich Schritt für Schritt dem Minimum nähert. Dazu gibt es viele Möglichkeiten (z.B.: Gauss-Newton). Entscheidend ist also der Unterschied, das es kein direktes Verfahren gibt um die Lösung zu finden. Nichtlineare Probleme haben dazu auch häufig mehrere (oft viele) "lokale" Minima, interessiert ist man aber am so genannten "globalen" Minimum, welches die "bestmögliche" Lösung des Problems darstellt. Daher ist eine gute Wahl des Startwertes meist eine essentielle Vorleistung für eine gute Lösung. Meist findet man diese durch "clevere" Betrachtung der Daten.

Als weiteres Beispiel soll hier die Funktion aus Gleichung 10.16 verallgemeinert werden, indem alle Parameter veränderlich gemacht werden,

$$f(x,a) = a_1 f_1(x) + a_2 f_2(x) \ , \tag{10.25}$$

$$f_1(x) = \exp\left(-a_3 x\right) \ , \tag{10.26}$$

$$f_2(x) = \sin(a_4 x)\exp\left(-a_5 x\right) \ , \tag{10.27}$$

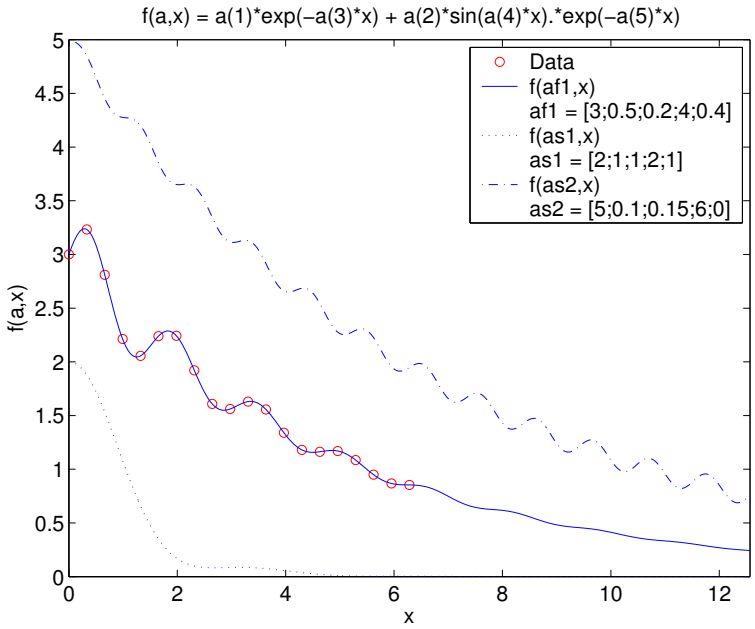
Dies ist nun ein nichtlineares Problem in a mit folgender Umsetzung in MATLAB:

```
fc  = 'a(1)*exp(-a(3)*x) + a(2)*sin(a(4)*x).*exp(-a(5)*x)';
fa  = inline(fc,'a','x');
```

Unter der Voraussetzung, dass die Daten wieder in `xd` und `yd` zur Verfügung stehen, kann man die Lösung folgendermaßen finden

```
as1 = [2,1,1,2,1];
af1 = nlinfit(xd,yd,fa,as1)
```

Die Darstellung der Daten, des Ergebnisses und der Resultate von zwei Startwerten kann man in folgender Graphik sehen:



Die Standard MATLAB-Funktion für das suchen von Minima ist die Routine `fminsearch`. Um diese beim vorliegenden Problem verwenden zu können, muss man eine Funktion programmieren, die den Skalarwert q der Gleichung 10.1 zurück gibt. Diese Funktion muss also die Summe der Abstandsquadrate an allen Datenpunkten berechnen und benötigt dafür den Parametervektor `a` und die Daten `xd` und `yd`. Am Beispiel der letzten Funktion kann das so aussehen:

```
function q = lqfunc(a,xd,yd)
y = a(1)*exp(-a(3)*xd) + a(2)*sin(a(4)*xd).*exp(-a(5)*xd);
q = sum((y-yd).^2);
```

Wichtig ist also der Punkt, dass hier jeweils nur ein skalarer Wert, nämlich der Wert der zu minimierenden Funktion, zurückgegeben wird. Diese Funktion wird nun an `fminsearch` zusammen mit Startwerten übergeben

```
af3 = fminsearch(@lqfunc,as1,[],xd,yd)
```

und liefert die "besten" Parameter. An Stelle von `[]` kann man Optionen übergeben (siehe Hilfe zu `fminsearch`). Die Datenvektoren `xd` und `yd` werden von `fminsearch` an die Funktion `lqfunc` weitergegeben. Dieses Beispiel zeigt somit nochmals, dass eigentlich die Funktion 10.1 minimiert wird, und dass dies dann zur besten Annäherung der Modellfunktion an die Daten führt.

10.2 Interpolation

Im Unterschied zur Kurvenanpassung (Fitten einer Modellfunktion) verwendet man bei Interpolieren "lokal" an die Datenpunkte angepasste Funktionen, wobei sichergestellt wird, dass diese Funktionen exakt die Datenpunkte reproduzieren. Ziel des Verfahrens ist es, zwischen den diskreten Datenpunkten (z.B. Messwerten) einen vernünftigen Verlauf zu finden (z.B. zum Plotten). In den meisten Fällen beschränkt man sich dabei auf Polynome bis maximal dritten Grades, die aber nur "lokal" um den jeweiligen Datenpunkt verwendet werden.

MATLAB bietet für eindimensionale Probleme die Routine `interp1` an, die folgenden Aufruf benötigt

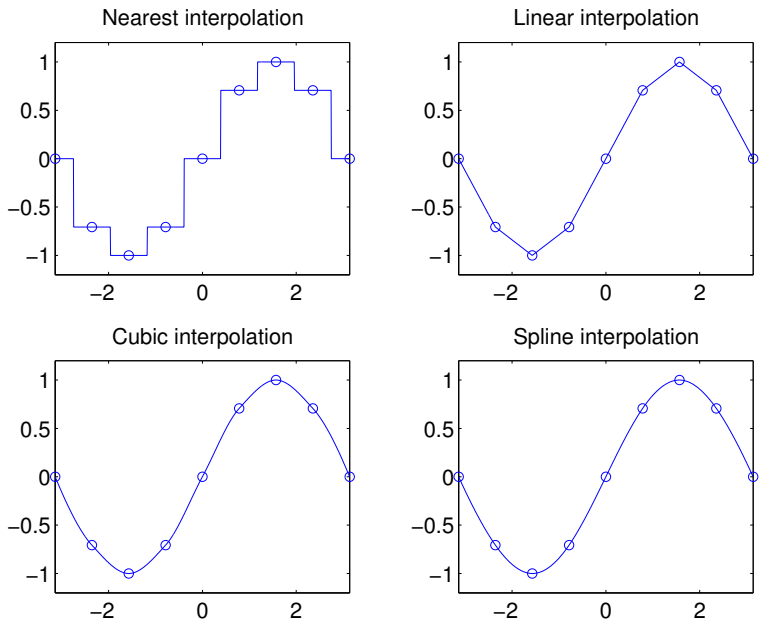
```
y = interp1(xd,yd,x,method)
```

wobei in `xd` und `yd` wieder die Datenpunkte liegen, `x` ein Vektor mit meist dichter liegenden x-Werten ist und `method` eine Stringvariable mit der gewünschten Methode ist.

Dies wird hier am Beispiel der Sinus-Funktion erläutert:

```
xd = linspace(-pi,pi,9);
yd = sin(xd);

x  = linspace(-pi,pi,1000);
y  = sin(x);

y1 = interp1(xd,yd,x,'nearest');
y2 = interp1(xd,yd,x,'linear');
y3 = interp1(xd,yd,x,'cubic');
y4 = interp1(xd,yd,x,'spline');
```

Die Ergebnisse für die einzelnen Methoden sehen folgendermaßen aus:



Folgende Methoden stehen zur Verfügung:

nearest Nächste Nachbar Interpolation

linear Lineare Interpolation

cubic Interpolation mit Polynomen dritten Grades

spline Die Spline-Technik verwendet Polynome dritten Grades, wobei sichergestellt wird, dass sich sowohl die Werte als auch die ersten Ableitungen bei den Datenpunkten ein glattes Verhalten zeigen.

Solche Interpolationen stehen natürlich auch in höheren Dimensionen zur Verfügung. Siehe dazu die Hilfe zu den Funktionen `interp2` und `interp3`.

Kapitel 11

Graphische Ausgabe

11.1 Grundlagen

MATLAB beinhaltet hervoragende Werkzeuge zur Visualisierung von numerischen Ergebnissen. Dies reicht von einfachen Befehlen bis zur detailierten Gestaltungsmöglichkeit praktisch aller Eigenschaften einer Graphik.

Die Art der Befehle gliedert sich in zwei Kategorien, sogenannte "High Level"- Befehle, die komplexe Aufgaben erfüllen und "Low Level"-Befehle zur Manipulation von Graphikobjekten.

11.1.1 Graphikobjekte

11.1.1.1 Objekthierarchie

Die Hierarchie von Graphikobjekten folgt einer Eltern-Kind-Beziehung (parent-child-relationship).

Die Eltern-Kind-Beziehung ist in Tabelle 11.1 durch die Rechtsverschiebung symbolisiert. So ist z.B. jede `line` ein Kind einer `axes`, diese ein Kind einer `figure`, und diese wiederum ein Kind des `root`. Auf allen Ebenen können nun Eigenschaften definiert und abgefragt werden.

11.1.1.2 Zugriff auf Objekte - Handles

Um nun Graphikobjekte eindeutig identifizieren zu können, braucht man einen Datentyp, der als Zeiger auf ein Graphikobjekt dient (Handle). Ein solcher Handle ist somit ein eindeutiger "Identifier" für ein Graphikobjekt. Handles können Variablen

Tabelle 11.1: Hierarchie von Graphikobjekten in MATLAB.

`root`				Graphiksystem
	`figure`			Zeichnung
		`axes`		Achsensystem
			`line`	Linie
			`patch`	Polygonzug
			`text`	Text
			`image`	Bild
			`surface`	Fläche
			`light`	Licht
			`rectangle`	Rechteck
		`uicontrol`		Benutzerinterface
		`uimenu`		Menüeinträge
		`uicontextmenu`		Kontextmenü

zugewiesen werden und stehen damit im jeweiligen Programm zur weiteren Verfügung. Im Wesentlichen kann bei jedem MATLAB-Graphikbefehl eine Zuweisung erfolgen. Anstelle von `plot(x,y)` kann man schreiben `ph=plot(x,y)`, wobei nun in der Variablen `ph` der Handle für die entsprechende Linie gespeichert ist.

Am Beispiel von Linien soll hier demonstriert werden, wie man zu Handles kommt.

```
x  = linspace(0,pi,100);
y1 = sin(x); y2 = cos(x);

fh    = figure;
ah    = axes;
lh(1) = line(x,y1, 'Color','red',  'LineStyle','-');
lh(2) = line(x,y2, 'Color','blue', 'LineStyle',':');
```

Die Variablen `fh`, `fh` und `lh` enthalten nun die Handles. Man sieht hier, dass es in MATLAB natürlich möglich ist, Arrays von Handles zu speichern.

Um nun alle oder nur bestimmte Eigenschaften eines Objektes abfragen zu können, benötigt man den Befehl `get`.

```
get(fh)
get(fh,'Position')
```

Die erste Form liefert dabei alle Eigenschaften und deren Werte und die zweite Form liefert nur den Wert der Eigenschaft `'Position'`.

Als Gegenstück ermöglicht der Befehl `set` das Verändern von Eigenschaften.

```
set(ah, 'Color','green')
set(fh, 'Units','normalized', 'Position',[0.1,0.1,0.8,0.8])
set(lh, 'LineWidth',10)
set(lh(2), 'Color','black')
```

Wie bei allen "Low Level" Graphikbefehlen (z.B.: `line`) kann man also Wertepaare angeben, die jeweils aus einer `'Eigenschaft'` und dem zugehörigen `'Wert'` bestehen. Die `'Eigenschaft'` ist dabei immer eine Zeichenkette aus einer vordefinierten Liste von Eigenschaften, der zugehörige `'Wert'` kann je nach `'Eigenschaft'` von unterschiedlichem Datentyp sein.

11.1.1.3 Spezielle Handles

Da das Graphiksystem automatisch gestarted wird, gibt es nach dem MATLAB-Programmstart den Handle auf `root`. Er hat immer den Wert `0`. Dieser ist besonders interessant, wenn man Defaulteinstellungen für Graphikobjekte einstellen will. So kann man z.B. mit

```
set(0,'DefaultFigureColor','b')
```

bevor man eine Figure öffnet das Defaultverhalten aller weiteren Figuren verändern. Sinngemäß gilt das natürlich für alle Graphikobjekte. Mit

```
set(0,'DefaultFigureColor','remove')
```

kann man die Einstellung wieder auf MATLAB-Defaultwerte zurücksetzen.

Hat man z.B. mehrere Figuren und/oder mehrere Achsensysteme kann man mit speziellen Handles auf die derzeit aktiven zugreifen:

`gcf`	Handle für aktive Figure	get current figure
`gca`	Handle für aktive Achse	get current axes
`gco`	Handle für aktives Objekt	get current object

11.2 Beispiele

Die Fülle der möglichen MATLAB-Befehle zur Darstellung von Graphiken übersteigt die Möglichkeiten des Skriptums. Hier finden Sie daher einige Beispiele aus deren Verhalten man die Wirkungsweise der MATLAB-Befehle erkennen kann (`htplot2d.m`, `htplot2da.m`, `htplot2ds.m` unter Verwendung der Hilfsroutine zum Setzen von Defaultwerten `setmyfig.m`).

Tabelle 11.2: MATLAB Befehle zum Erzeugen einfacher zweidimensionaler Graphiken

`fplot('fun',[`x_{min}`,`x_{max}`])`	11.2.1.1	Zeichnet 'fun' im Bereich von x_{min} bis x_{max}
`plot(x,y)`	11.2.1.2	Zeichnet y als Funktion von x
`ezplot('fun',[`x_{min}`,`x_{max}`])`	11.2.1.3	erstellt u.a. implizite Funktionen, automatische Achsenbeschriftung
`comet(x,y,p)`	11.2.1.4	Zeichnet 2D Funktion in Form eines animierten 'Kometen'
`semilogx(x,y)`	11.2.1.5	Zeichnet 2D Funktion mit (10er-) logarithmischer x-Achse
`semilogy(x,y)`	11.2.1.6	Zeichnet 2D Funktion mit (10er-) logarithmischer y-Achse
`loglog(x,y)`	11.2.1.7	Zeichnet 2D Funktion mit (10er-) logarithmischer x- und y-Achse
`plotyy(`x_1`,`y_1`,`x_2`,`y_2`,'`f_1`','`f_2`')`	11.2.1.8	Erstellt 2 Graphen mit den Plotbefehlen f_1 und f_2 mit getrennten y - Achsen
`polar(phi,r)`	11.2.1.9	Zeichnet die Funktion r(phi) in Polarkoordinaten.

Ansonsten kann hier nur auf die MATLAB-Hilfe verwiesen werden. Eine lange Liste von HTML-Seiten finden man unter diesem Link auf Graphikhilfe.

Einen guten Überblick bekommt man auch im `helpdesk` unter den Punkten `Functions by Catagory`, `Graphics`, `3-D Visualization` and `Handle Graphics Object Property Browser`

11.2.1 Zweidimensionale Plots

Es gibt eine Reihe von Befehlen zur Darstellung zweidimensionaler Graphiken.

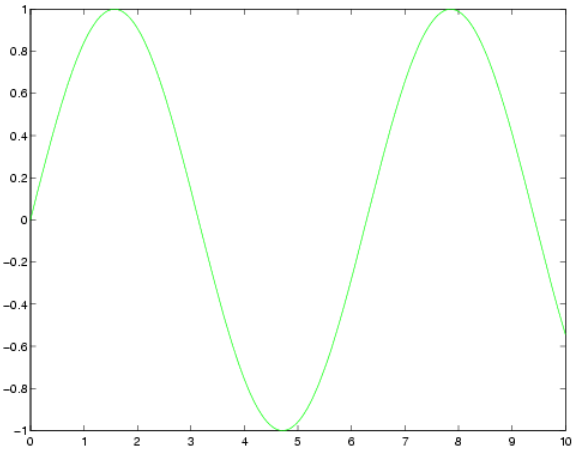
11.2.1.1 Fplot

Einfachste Möglichkeit, eine Funktion (in String - Schreibweise) innerhalb eines Intervalls zu plotten.

`fplot` `graph_fplot.m`

Plot einer grünen Sinuskurve im Bereich von x = 0 bis 10

```
fplot('sin',[0,10],'g')
```



Als weitere Farbkürzel neben 'g' (grün) sind 'k' (schwarz), 'm' (violett), 'r' (rot), 'c' (türkis), 'b' (blau), 'w' (weiß) und 'y' (gelb) erlaubt, siehe auch `linespec`.

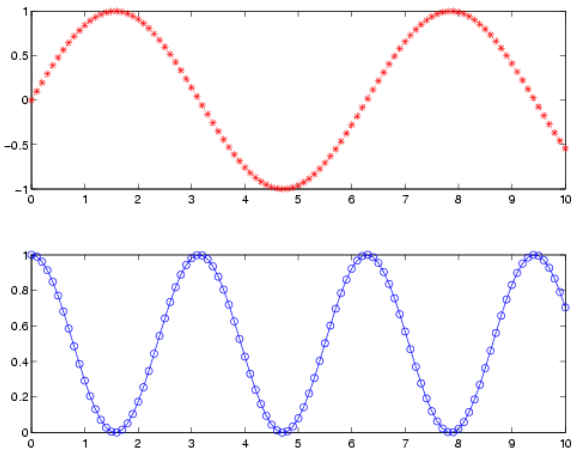
11.2.1.2 Plot

Einfacher 2D Plot, zeichnet die Funktion y = f(x) bei Vorgabe des Vektors x

`plot` `graph_plot.m`

Mit Hilfe von `subplot` werden 2 Achsen geschaffen, die Zeichen zwischen den ' ' in plot symbolisieren Farbe, 'Marker Style' und 'Line Style'.

```
x=0:0.1:10;
y1=sin(x);
y2=cos(x).^2;

figure
subplot(2,1,1)
plot(x,y1,'r*:')

subplot(2,1,2)
plot(x,y2,'bo-')
```



Eine vollständige Auflistung der verfügbaren Symbole der erwähnten 'Styles' finden sich in der Hilfe von `linespec`

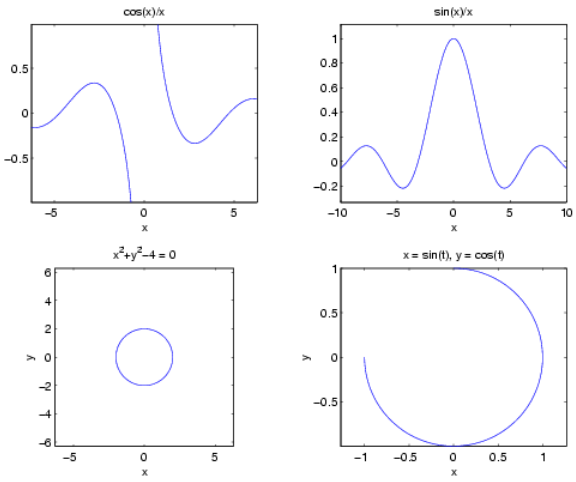
11.2.1.3 Ezplot

Erstellt 2 dimensionale, unter anderem auch implizite Funktionen mit automatischer Achsenbeschriftung und wenn erwünscht, mit automatischen Intervallgrenzen.

`ezplot` `graph_ezplot.m`

Der Befehl `axis square` stellt jede Achse mit derselben Länge dar und verhindert, dass Kreise als Ellipsen wirken.

```
subplot(2,2,1)
ezplot('cos(x)/x')

subplot(2,2,2)
ezplot('sin(x)/x',[-10,10])

subplot(2,2,3)
ezplot('x^2+y^2-4')
axis square

subplot(2,2,4)
ezplot('sin','cos',[0,1.5*pi])
```



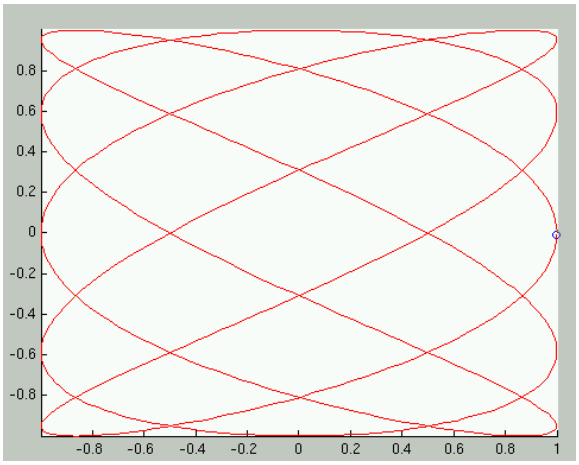
11.2.1.4 Comet

Erstellt eine 2 dimensionale Funktion in Form eines sich bewegenden 'Kometen', dessen Schweif bzw. Spur den Graphen darstellt.

`comet` `graph_comet.m`

Der letzte Parameter in `comet` gibt die Schweiflänge relativ zur Gesamtlänge des Graphen an.

```
t=0:0.01:2*pi;
x=cos(5*t);
y=sin(3*t);

comet(x,y,0.2)
```



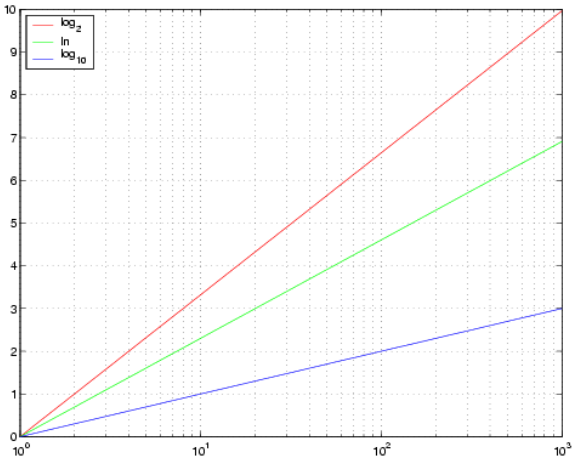
Achtung, die Erstellung des Graphen erfolgt im `erasemode none`, wird das Graphikfenster vergrößert, verschwindet der Graph, er kann daher auch nicht gedruckt werden.

11.2.1.5 Semilogx

Erstellt eine 2 dimensionale Funktion mit logarithmischer x - Achse.

`semilogx` `graph_semilogx.m`

Der Befehl `legend` fügt dem Plot an einer wählbaren Position eine Legende der Lines hinzu, `grid` fügt der Graphik Gitterlinien hinzu.




```
x=logspace(0,3,30);
y1=log2(x);
y2=log(x);
y3=log10(x);

semilogx(x,y1,'r',...
         x,y2,'g',...
         x,y3,'b')

grid on
legend('log_2','ln','log_{10}',2)
```

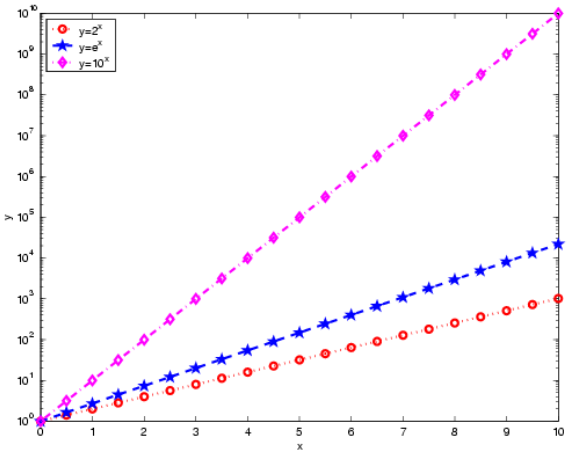
Sollen mehrere Lines in eine Achse gezeichnet werden, so können die Koordinaten und Style Eigenschaften der Lines hintereinandergefügt werden.

11.2.1.6 Semilogy

Erstellt eine 2 dimensionale Funktion mit logarithmischer y - Achse.

`semilogy` `graph_semilogy.m`

Die Befehle `xlabel` und `ylabel` ermöglichen die Beschriftung der x - und der y - Achse.




```
x=0:0.5:10;
y1=2.^x;
y2=exp(x);
y3=10.^x;

semilogy(x,y1,'r:o',...
         x,y2,'b--p',...
         x,y3,'m-.d',...
 'linewidth',2)

xlabel('x')
ylabel('y')
legend('y=2^x','y=e^x','y=10^x',2)
```

Die Dicke der Linien läßt sich mit der Line - Eigenschaft `linewidth` verändern, im Beispiel beträgt sie 2 Punkte.

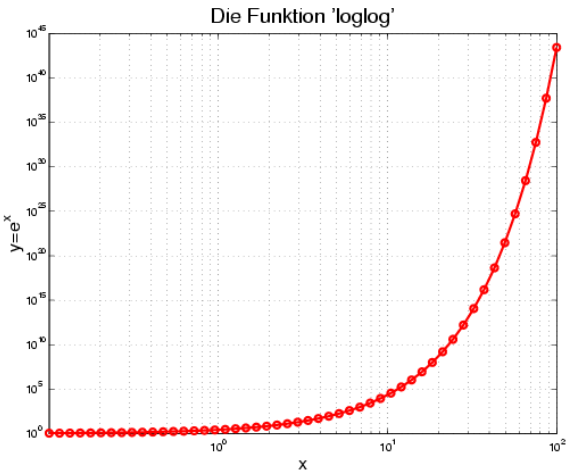
11.2.1.7 Loglog

Erstellt eine 2 dimensionale Funktion mit logarithmischer x - und y - Achse.

`loglog` `graph_loglog.m`

Um die Achse mit einer Überschrift zu versehen, kann der Befehl `titel` verwendet werden.

```
x=logspace(-1,2);
y=exp(x);

loglog(x,y,'ro-','linewidth',2)

xlabel('x','fontsize',16)
ylabel('y=e^x','fontsize',16)
title('Funktion ''loglog''',...
      'fontsize',18)
```



Die Größe der Schrift wird mit `fontsize` gesteuert, dies ist jedoch nur eine von vielen Texteigenschaften. Werden in einem String ' ' - Symbole verwendet, so muss man, wie im Beispiel der Überschrift, zwei statt nur eines der ' ' Symbole verwenden.

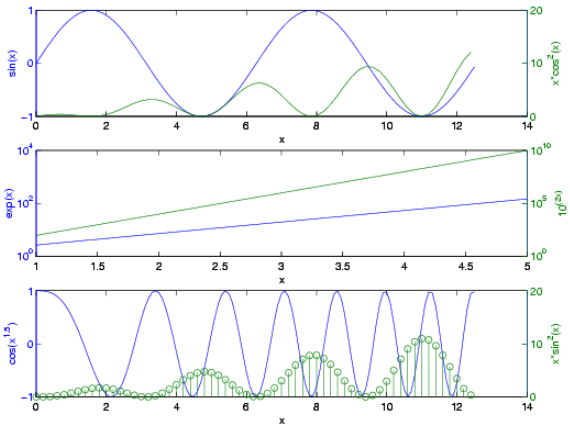
11.2.1.8 Plotyy

Erstellt zwei durch x1 und y1 bzw. x2 und y2 definierte Graphen mit eigenen y-Achsen. Es ist erlaubt, beide Funktionen mit unterschiedlichen Plot-Befehlen darzustellen.

`plotyy` `graph_plotyy.m`

Die linke y-Achse gehört zur ersten, die rechte hingegen zur zweiten Funktion. Stellvertretend für die 3 Subplots sei hier nur der 3. angeführt.

```
subplot(3,1,3)
x1=0:0.1:4*pi;
y1=cos((x1.^1.5));
x2=0:.2:4*pi;
y2=x2.*sin(x2).^2;

[AX,H1,H2]=plotyy(x1,y1,x2,y2,...
                  'plot','stem');

set(get(AX(1),'xlabel'),...
    'String','x')
set(get(AX(2),'xlabel'),...
    'String','x')
set(get(AX(1),'ylabel'),...
   'String','cos(x^{1.5})')
set(get(AX(2),'ylabel'),...
   'String','x*sin^2(x)')
```



In diesem Beispiel tritt erstmals das sehr wichtige 'Graphik-Handle' Konzept auf. Ein Graphik-Handle ist ein Code, der die gesamte Information von Achsen, Figures und anderen Graphik-Objekten beinhaltet. Mit dem Befehl `get` können alle Eigenschaften des Objekts abgefragt und mit `set` gesetzt werden. In diesem Beispiel etwa werden die 'String' Eigenschaften von x- und ylabel gesetzt. AX beinhaltet die Handles beider Achsen, H1 und H2 sind die Handles der beiden 'Line' Objekte. So bekommt man beispielsweise mit get(H1) die gesamte Information über den blau gezeichneten Graphen, mit set(H1,'linewidth',4) verändert man die Liniendicke auf 4 Punkte.

Für die Darstellungsarten der Funktionen sind folgende Varianten erlaubt: `plot`, `semilogx`, `semilogy`, `loglog` sowie `stem`.

Tabelle 11.3: MATLAB Befehle zum Erzeugen von Balken- und Kreisdiagrammen

`hist(y,x)`	11.2.1.10	Erstellt ein Histogramm der Werte in y über jenen von x
`bar(x,y,'width','style')`	11.2.1.11	Stellt die Datenpaare [x,y] als vertikale Balken dar
`barh(x,y,'width','style')`	11.2.1.12	Stellt die Datenpaare [x,y] als horizontale Balken dar
`pie(x,'explode')`	11.2.1.13	Zeichnet ein 2D Kreisdiagramm der Daten von x

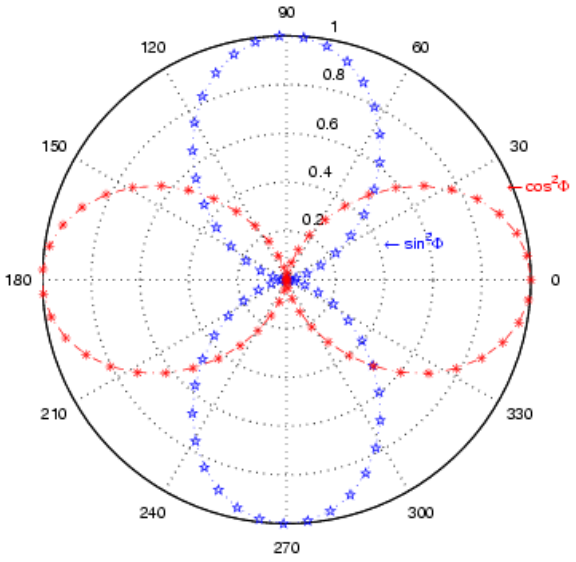
11.2.1.9 Polardiagramm

Zeichnet die Funktion r=f(phi) im Polardiagramm.

`polar` `graph_polar.m`

Text in der Spalte

```
phi=0:0.1:2*pi;
r1=sin(phi).^2;
r2=cos(phi).^2;

polar(phi,r1,'b:p')
hold on
polar(phi,r2,'r-.*')

text(phi(5),r1(5),...
  '\leftarrow sin^2\Phi',...
  'color','blue')
text(phi(10),r2(10),...
  '\leftarrow cos^2\Phi',...
  'color','red')
hold off
```



Nach dem Befehl `hold on` werden alle weiteren Graphiken in das aktuelle Achsensystem gezeichnet, ohne die vorigen Graphiken zu löschen, erst mit `hold off` werden alten Graphiken durch neue ersetzt.

`text(x,y,'string')` gestattet die Positionierung eines Texts 'string' bei den Koordinaten (x,y) im Achsensystem.

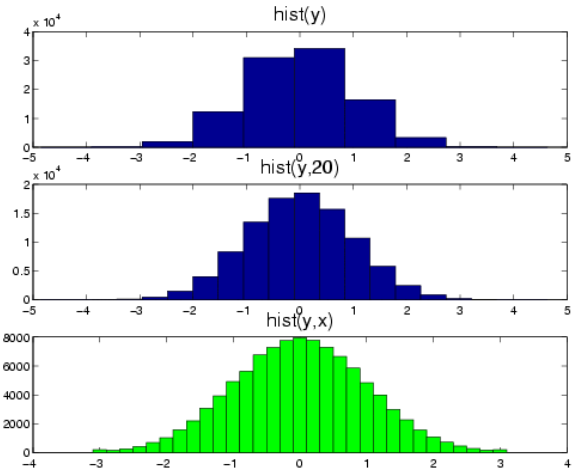
11.2.1.10 Histogramm

Die Daten von y werden in Form von Histogrammen dargestellt.

`hist` `graph_hist.m`

Die unterschiedlichen Aufrufe des Histogramm - Befehls anhand eines Beispiels normalverteilter Daten:

```
y=randn(1,100000);
subplot(3,1,1)
hist(y)
title('hist(y)','fontsize',16);

subplot(3,1,2)
hist(y,20)
title('hist(y,20)','fontsize',16)

subplot(3,1,3)
x=-3:0.2:3;
hist(y,x)
title('hist(y,x)','fontsize',16);

h = findobj(gca,'Type','patch');
set(h,'facecolor','g')
```



Die letzten beiden Zeilen färben die Balken des Histogramms grün ein, dabei wird mit `findobj` nach allen Graphik-Objekten der mit `gca` abgefragten aktuellen Achsen gesucht, die vom Typ `patch` sind. Der resultierende Handle wird von `set` zum Verändern der Patch-Eigenschaft herangezogen.

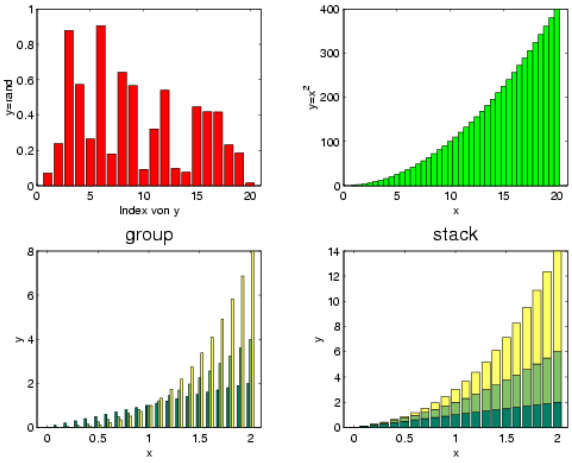
11.2.1.11 Bar

Erstellt an den Positionen von x vertikale Balken der Höhe y mit der relativen Balkenbreite 'width'. Die Balkengruppierung wird mit der Option 'style' gesteuert.

`bar` `graph_bar.m`

y kann sowohl ein Vektor, als auch eine n * m Matrix sein, wobei n=length(x) und m die Anzahl der dargestellten Datensätze entspricht.

```
subplot(2,2,1)
y=rand(20,1);
bar(y,'r')

subplot(2,2,2)
x=1:0.5:20;
y=x.^2;
bar(x,y,1,'g')

subplot(2,2,3)
x=[0:0.1:2]';
y=[x,x.^2,x.^3];
colormap summer
bar(x,y,1,'group')

subplot(2,2,4)
bar(x,y,'stack')
```



Der Style 'grouped' positioniert die Balken der m Datensätze nebeneinander, mit 'stack' werden sie übereinander angeordnet. Mir `colormap` lassen sich sowohl vordefinierte, als auch selbst entworfene Farbskalen für die Darstellung der Graphiken verwenden.

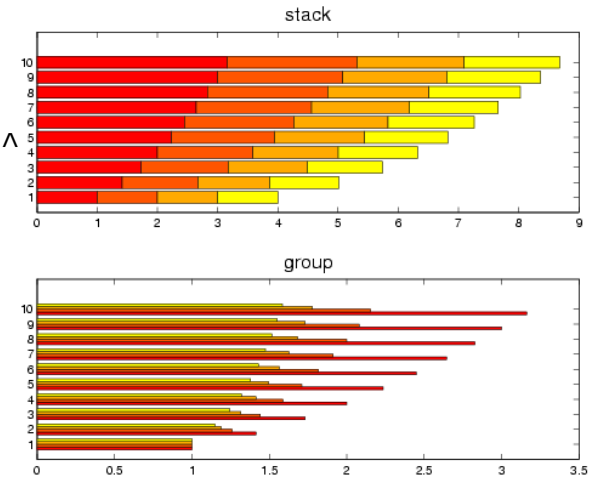
11.2.1.12 Barh

Die Datenpaare (x,y) werden in Form von horizontalen Balken des Stiles 'style' mit der relativen Breite 'width' veranschaulicht.

`barh` `graph_barh.m`

Wie im Beispiel 11.2.1.11 kann y eine Matrix sein.

```
x=(1:1:10)';
y=[x.^(1/2),x.^(1/3),x.^(1/4),x.^

subplot(2,1,1)
barh(x,y,'stack')

subplot(2,1,2)
barh(x,y,1,'group')

colormap autumn
set(gcf,'color','w')
```



Für die Darstellungsmöglichkeiten gruppierter Daten kann man zwischen 'grouped' und 'stack' wählen.

Der Befehl `gcf` ermittelt den Handle der aktuellen Figure, im Beispiel wird er benutzt, um die Farbe des Fensters auf weiß zu setzen.

11.2.1.13 Pie

Erstellt aus den Daten von x ein 2D Kreisdiagramm.

`pie` `graph_pie.m`

Wird der aus 0 und 1 bestehende Vektor 'explode' angegeben, so werden jene Segmente hervorgehoben, die in explode (muß dieselbe Länge wie x haben) den Wert 1 aufweisen.

```
einwohner=[278,562.7,1545.3,...
           1380.5,518.6,1202.3,..
           672.2,350.3,1611.4];
explode=[0,1,0,0,0,1,0,0,0];

pie(einwohner,explode)
```

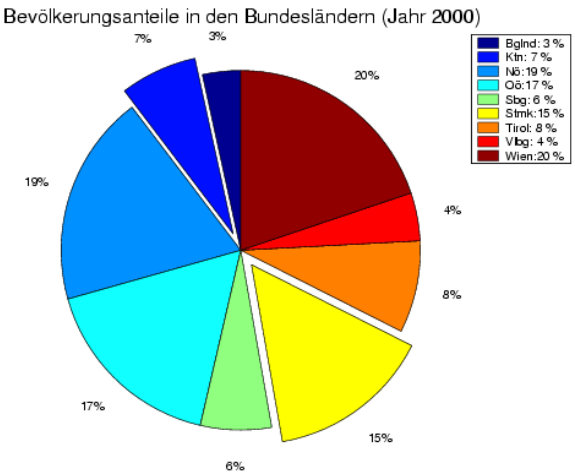


Tabelle 11.4: MATLAB Befehle zum Erzeugen von speziellen zweidimensionalen Graphiken

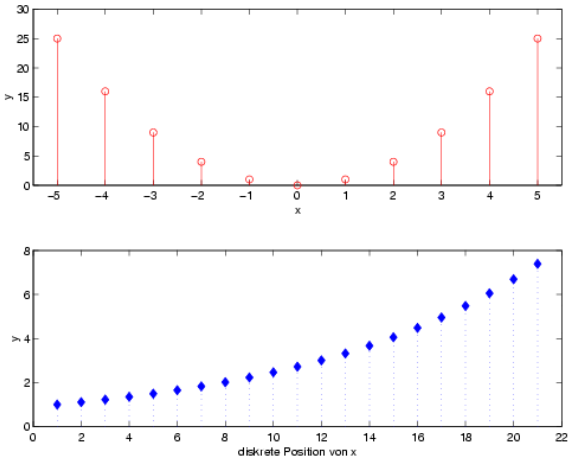
`stem(x,y)`	11.2.1.14	Zeichnet y=f(x) und verbindet Punkte mit x-Achse
`stairs(x,y)`	11.2.1.15	Erstellt Funktion y=f(x) in Form eines Stufendiagramms
`errorbar(x,y,e)`	11.2.1.16	Zeichnet y als Funktion von x samt Fehlerbalken der Länge e
`compass(x,y)`	11.2.1.17	Zeichnet y=f(x) und verbindet die Punkte durch Vektorpfeile mit dem Ursprung
`feather(u,v)`	11.2.1.18	Zeichnet die relativen Koordinaten u und v und verbindet die Punkte mit den jeweiligen Koordinatenursprüngen entlang der Abszisse
`scatter(x,y,r,c)`	11.2.1.19	Zeichnet Punkte an den Stellen (x,y) der Größe r sowie der Farbe c
`pcolor(x,y,c)`	11.2.1.20	Erstellt einen 'Pseudocolorplot' der Elemente c an den von den Punkten (x,y) definierten Positionen
`area(x,y)`	11.2.1.21	Füllt den Bereich zwischen y=f(x) und der Abszisse mit einer Farbe
`fill(x,y,c)`	11.2.1.22	Malt die durch (x,y) definierten Polygone mit der Farbe c aus
`contour(x,y,z)`	11.2.1.23	Zeichnet durch z=f(x,y) definierte Konturlinien
`contourf(x,y,z)`	11.2.1.24	Zeichnet durch z=f(x,y) definierte Konturlinien und füllt die Flächen dazwischen aus
`quiver(x,y,u,v)`	11.2.1.25	Erstellt von den Punkten (x,y) ausgehende Vektoren mit den Komponenten (u,v)
`plotmatrix(x,y)`	11.2.1.26	Streudiagramm, die Spalten von x werden über jenen von y aufgetragen

11.2.1.14 Stem

Zeichnet y als eine Funktion von x und verbindet zusätzlich die Punkte (x,y) durch senkrechte Linien mit der Abszisse.

`stem` `graph_stem.m`

Mit der Option 'filled' werden die Datenpunkte ausgefüllt.




```
subplot(2,1,1)
x=-5:5;
y=x.^2;
stem(x,y,'r')
axis([-5.5,5.5,0,30])

subplot(2,1,2)
x=0:0.1:2;
stem(exp(x),'fill','b:d')
xlim([0,length(x)+1])
```

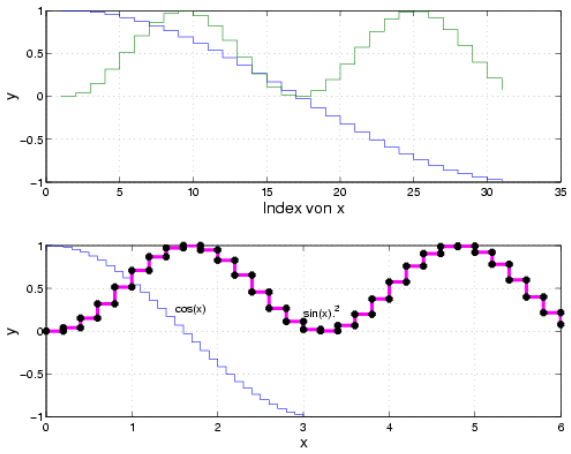
Im ersten Subplot werden die Achsengrenzen durch `axis([xmin,xmax, ymin, ymax])` geregelt, im zweiten Subplot mit dem Befehl `xlim`, wobei der Wertebereich der y-Achse unberührt bleibt.

11.2.1.15 Stairs

Erstellt ein 2D Stufendiagramm von y als Funktion von x

`stairs` `graph_stairs.m`

```
subplot(2,1,1)
x1=[0:0.1:3]';x2=[0:0.2:6]';
y1=cos(x1);y2=sin(x2).^2;
y=[y1,y2];
stairs(y)

subplot(2,1,2)
x=[x1,x2];
handle=stairs(x,y);

set(handle(2),'linewidth',3,...
      'color','m','marker','*',..
      'markeredgecolor','k')
```



Mit Hilfe des Handle-Konzepts werden Liniendicke, Malfarbe, Datensymbole sowie die Umrandung dieser Datensymbole verändert.

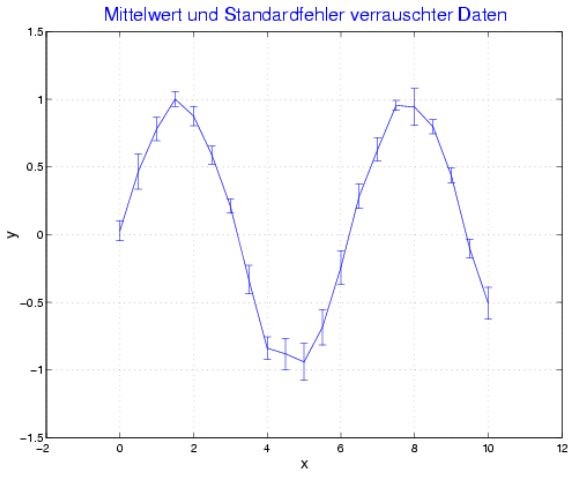
11.2.1.16 Errorbar

Zeichnet y als Funktion von x und fügt Fehlerbalken hinzu, die nach unten und oben durchaus unterschiedlicher Länge sein können.

`errorbar` `graph_errorbar.m`

Mit Errorbar lassen sich elegant Mittelwerte und Standardabweichungen abbilden.

```
x=0:0.5:10;
y=repmat(sin(x),[5,1]);
zufalls_fehler=randn(size(y))/10;
y = y + zufalls_fehler;

errorbar(x,mean(y),std(y));
```



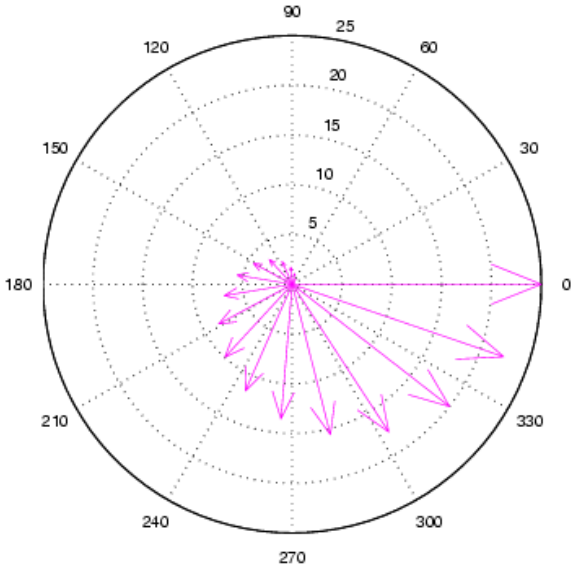
11.2.1.17 Compass

Zeichnet y als Funktion von x und verbindet die Punkte mit dem Koordinatenursprung durch Vektorpfeile.

`compass` `graph_compass.m`

Bei den Daten (x,y) handelt es sich um kartesische Koordinaten.

```
phi=linspace(0,2*pi,20);
r=linspace(0,5,20);
[x,y]=pol2cart(phi,r);

compass(r.*x,r.*y,'m')
```



Mit `[x,y]=pol2cart(phi,r)` lassen sich die Polarkoordinaten (phi,r) in die kartesischen Koordinaten (x,y) umwandeln.

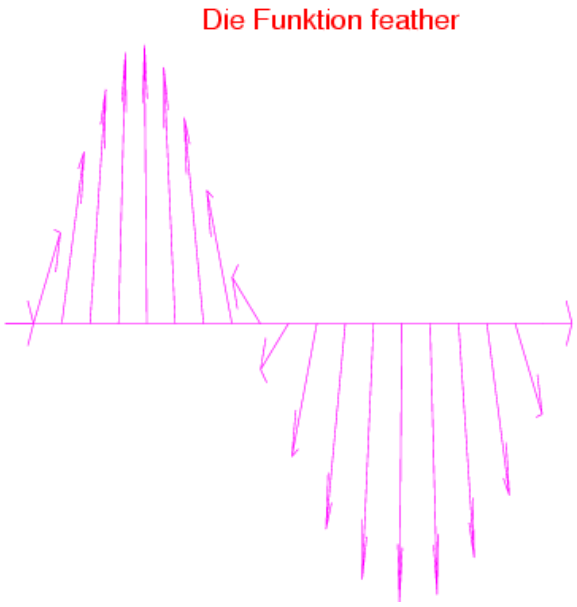
11.2.1.18 Feather

Zeichnet die Punkte (u, v) relativ zu äquidistanten, auf der Abszisse liegenden Koordinatenursprüngen und verbindet sie mit Vektorpfeilen. Statt der reellen Werte (u,v) können auch komplexe Werte (z) verwendeten werden, wobei auf der Abszisse die Real- und auf der Ordinate die Imaginärteile aufgetragen werden.

`feather` `graph_feather.m`

Normalerweise ist i auch in Matlab die imaginäre Einheit, das Symbol 'i' wird jedoch häufig als Laufindex verwendet und verliert dadurch den Wert $sqrt(-1)$.

```
phi=linspace(0,2*pi,20);
i=sqrt(-1);
z=exp(i*phi);

feather(z,'m')

axis off
```



Mit `axis off` werden die Achsenbeschriftungen sowie -ticks entfernt.

11.2.1.19 scatter

Zeichnet Daten durch Angabe der Positionen (x,y). Die Größe r sowie die Farbe c ist für alle Punkte getrennt einstellbar. Zusätzlich kann die Form der Datenpunkte ausgewählt und bei Bedarf durch die Option 'filled' gefüllt werden.

scatter — graph_scatter.m

```
t=linspace(0,pi,50);
x=repmat(cos(t),[7,1]);
y=repmat(sin(t),[7,1]);
r=[6:12]';
r=repmat(r,[1,50]);
farbe=[1:7]';
farbe=repmat(farbe,[1,50]);

xx=reshape(r.*x,[],1);
yy=reshape(r.*y,[],1);
rr=5*reshape(r,[],1);
farbe=reshape(farbe,[],1);

scatter(xx,yy,rr,farbe,'o','filled')
axis equal off
```

`axis equal` paßt das Achsensystem einem Quadrat an, sodass Kreise wirklich kreisförmig und nicht elliptisch aussehen.

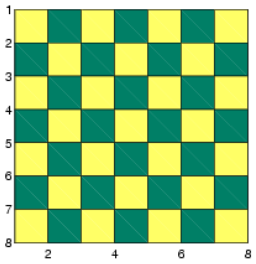
11.2.1.20 Pseudocolor

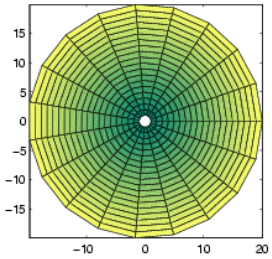
Erstellt einen 'Pseudocolorplot' der Elemente c an den von den Punkten (x,y) definierten Positionen. Wird nur die Farben c angegeben, so werden die Farbe auf einer Matrix der Größe size(c) abgebildet.

`pcolor` `graph_pcolor.m`

Der Befehl `eye(2)` erzeugt eine 2*2 Diagonalmatrix, mit `repmat` wird diese Diagonalmatrix zu einem Schachbrettmuster aneinanderkopiert.

```
x=eye(2);
X=repmat(x,[4,4]);
pcolor(X)
colormap summer; axis ij square

t=linspace(0,2*pi,20);
x=cos(t); y=sin(t); r=[1:20]';
X=repmat(x,[20,1]);
Y=repmat(y,[20,1]);
R=repmat(r,[1,20]);
axis square; pcolor(R.*X,R.*Y,R)
```





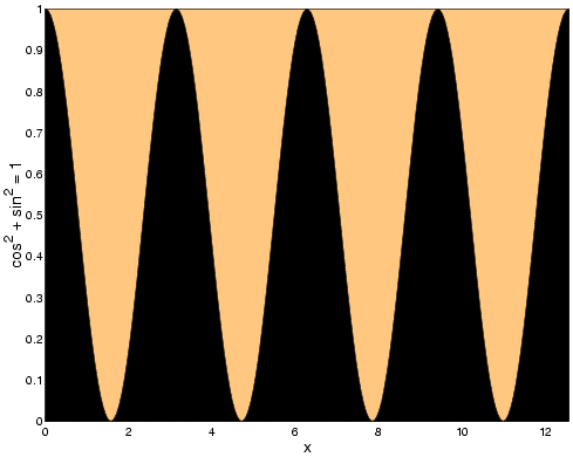
`axis ij` wählt für das Achsensystem den Matrixmodus, wodurch die Indizierung in der linken oberen Ecke der dargestellten Matrix beginnt und jede Zelle die Länge 1 besitzt.

11.2.1.21 Area

Füllt den Bereich zwischen 2 Graphen (wenn y eine Matrix ist) bzw. zwischen einem Graphen und der Abszisse (wenn y ein Vektor ist) mit Farben aus.

`area` `graph_area.m`

```
t=linspace(0,4*pi,200);
y1=cos(t).^2;
y2=sin(t).^2;
y=[y1;y2]';

area(t,y)

axis tight
colormap copper
```



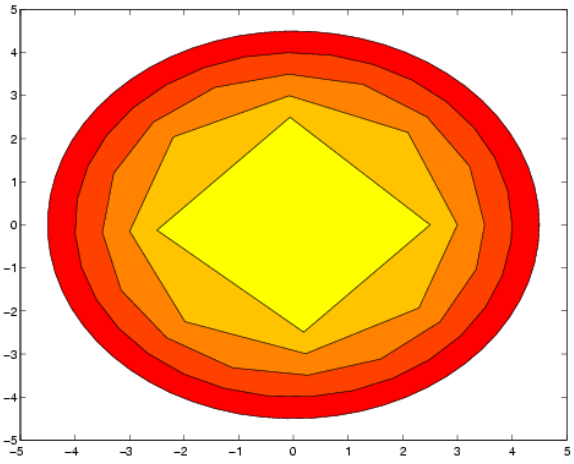
`axis tight` wählt die Achsengrenzen derart, dass sie nur den Bereich der Graphik abdecken.

11.2.1.22 Fill

Malt die durch die Punkte (x,y) definierten Polygone mit der Farbe c aus.

`fill` `graph_fill.m`

Von dem Kreis (eigentlich 64-Eck) werden in einer Schleife jeder, jeder 2., 4., 8. und 16. Punkt herausgegriffen und durch Linien zu einem Polygon verbunden und mit der i. Farbe der aktuellen `colormap` ausgemalt.




```
t=linspace(0,2*pi,64);
x=cos(t);
y=sin(t);

for i=1:5
    r=5-i/2;
    index=2^(i-1);
    fill(r*x(1:index:end),...
         r*y(1:index:end),i)
    hold on
end
```

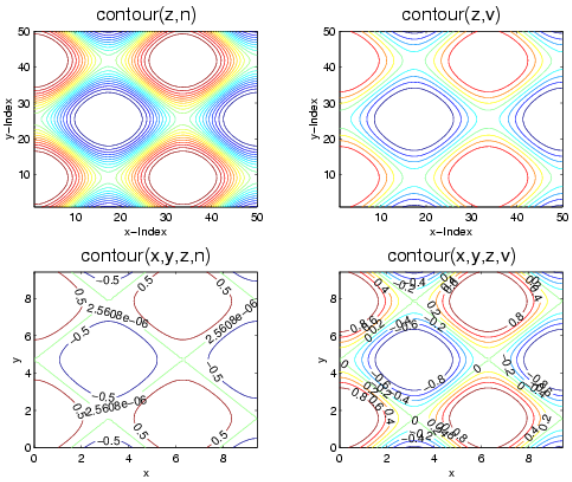
11.2.1.23 Contour

Zeichnet z als Funktion von x und y in Form von Konturlinien (Höhenlinien), die je nach Aufruf von `contour` äquidistant sind oder bei bestimmten Werten von z liegen.

`contour` `graph_contour.m`

Der sehr wichtige und vorallem bei 3D Plots unabkömmliche Befehl `meshgrid` erzeugt eine Matrix für die x- sowie eine für die y- Komponente des Gitters, über dem z definiert ist

```
x=linspace(0,3*pi,50);
y=linspace(0,3*pi,50);
[xx,yy]=meshgrid(x,y);
z=(sin(cos(xx)+sin(yy)));
v=linspace(min(min(z)),...
       max(max(z)),10);

subplot(2,2,1)
contour(z,20)

subplot(2,2,2)
contour(z,v)

subplot(2,2,3)
[c,h]=contour(xx,yy,z,3);
clabel(c,h)

subplot(2,2,4)
v=[-1:0.2:1];
[c,h]=contour(xx,yy,z,v);
clabel(c,h)
```



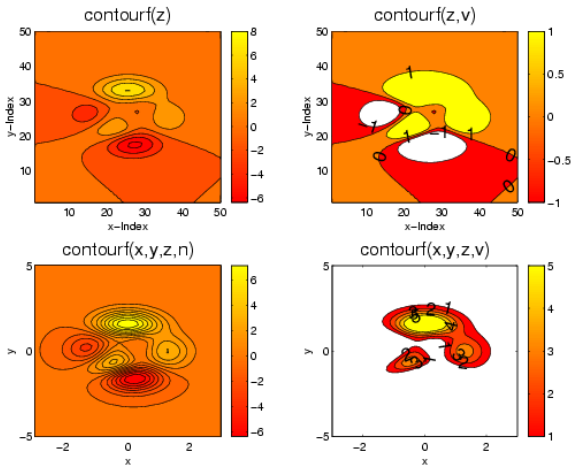
Mit `clabel` werden die Konturlinien mit den entsprechenden z-Werten beschriftet.

11.2.1.24 Contourf

Ähnliche Wirkung wie `contour` in 11.2.1.23, allerdings werden die Flächen zwischen den Konturlinien ausgemalt.

`contourf` — `graph_contourf.m`

```
x=linspace(-3,3,50);
y=linspace(-5,5,50);
[xx,yy]=meshgrid(x,y);

subplot(2,2,1)
zz=peaks(xx,yy);
contourf(zz);

subplot(2,2,2);
v=[-1,0,1];
[c,h]=contourf(zz,v);
clabel(c,h,'fontsize',16)

subplot(2,2,3)
contourf(xx,yy,zz,15)

subplot(2,2,4)
v=[1,2,3,4,5];
[c,h]=contourf(xx,yy,zz,v);
clabel(c,h,'fontsize',16)

colorbar
```



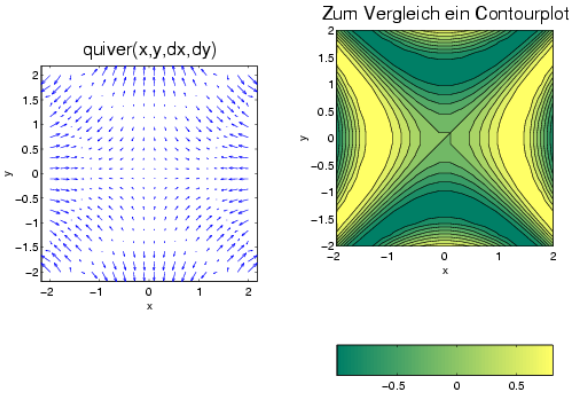
Der Befehl `colorbar` fügt am rechten Rand der Achse eine Farbskala mit einer Zuordnung der Farben zu den z-Werten hinzu.

11.2.1.25 Quiver

Erstellt von den Punkten (x,y) ausgehende Vektoren mit den Komponenten (u,v).

`quiver` `graph_quiver.m`

Die linke Abbildung wurde mit dem Befehl `quiver` erzeugt, rechts davon befindet sich zum besseren Verständnis seiner Funktionsweise ein Contourplot




```
x=linspace(-2,2,20);
y=linspace(-2,2,20);
[xx,yy]=meshgrid(x,y);

zz=sin(xx.^2-yy.^2);
[dx,dy]= gradient(zz);

subplot(1,2,1)
quiver(xx,yy,dx,dy)

subplot(1,2,2)
contourf(xx,yy,zz)
colorbar('horiz')
```

Mit Hilfe von `gradient` erhält man die x- und y- Komponenten des numerischen Gradienten.

Tabelle 11.5: MATLAB Befehle zum Erzeugen einfacher dreidimensionaler Graphiken

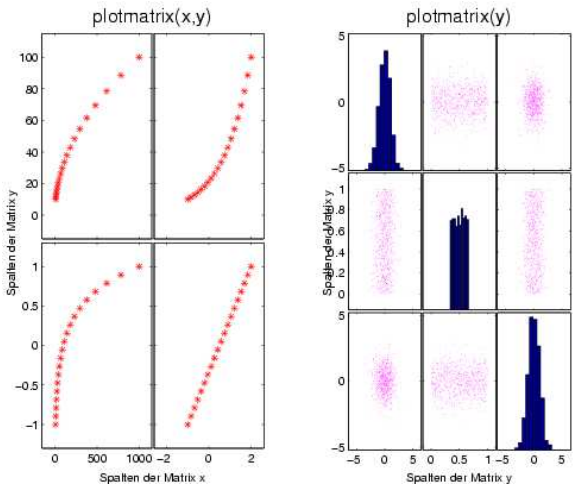
`plot3(x,y,z)`	11.2.2.1	3D Daten werden durch Angabe von x, y und z dargestellt
`ezplot3(x(t),y(t),z(t))`	11.2.2.2	Erstellt parametrischen 3D Plot durch Angabe der Funktionen als Strings und des Wertebereichs für t
`comet3(x,y,z,p)`	11.2.2.3	Zeichnet 3D Funktion in Form eines animierten 'Kometen'
`fill3(x,y,z,c)`	11.2.2.4	Malt die durch (x,y,z) definierten 3D-Polygone mit der Farbe c aus

11.2.1.26 Plotmatrix

Erstellung eines Streudiagramms, die Spalten der Matrix x werden über jenen der Matrix y aufgetragen.

`plotmatrix` `graph_plotmatrix.m`

```
subplot(1,2,1)
x1=logspace(1,3,20)';
x2=linspace(-1,2,20)';
y1=logspace(1,2,20)';
y2=linspace(-1,1,20)';
x=[x1,x2];
y=[y1,y2];

plotmatrix(x,y,'r*')
subplot(1,2,2)
y = randn(1000,3);
y(:,2)=rand(1000,1);
plotmatrix(y,'m.')
```



Wird nur eine Matrix übergeben, dann werden in den Diagonalen der Subplots Histogramme der betreffenden Spalten eingezeichnet.

11.2.2 Dreidimensionale Plots

Matlab bietet auch eine Fülle von Befehlen, 3D Graphiken eindrucksvoll darzustellen

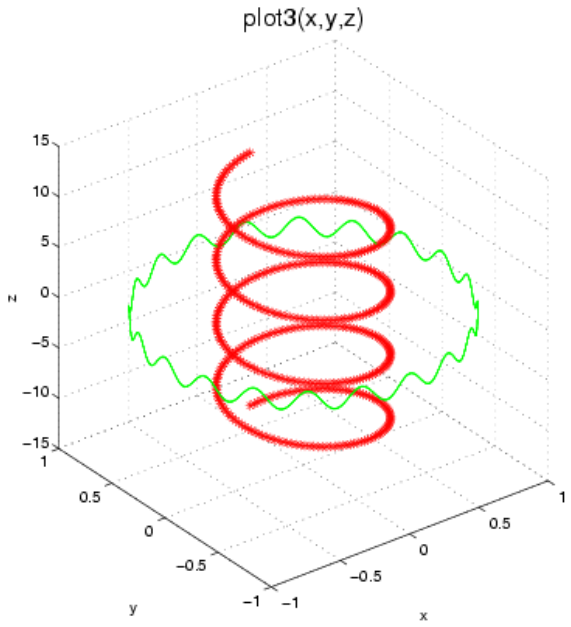
11.2.2.1 Plot3

Zeichnet die Daten (x,y,z) in einem 3D-Koordinatensystem ein und verbindet sie gegebenenfalls durch Linien.

`plot3` — `graph_plot3.m`

Informationen zu den möglichen Farben und Stilen der 3D-Linien findet man unter `linespec`

```
t=linspace(-4*pi,4*pi,500);
x1=0.5*sin(t);
y1=0.5*cos(t);
z1=t;
x2=cos(t);
y2=sin(t);
z2=cos(20*t);

plot3(x1,y1,z1,'r*-',x2,y2,z2,'g'

rotate3d
```



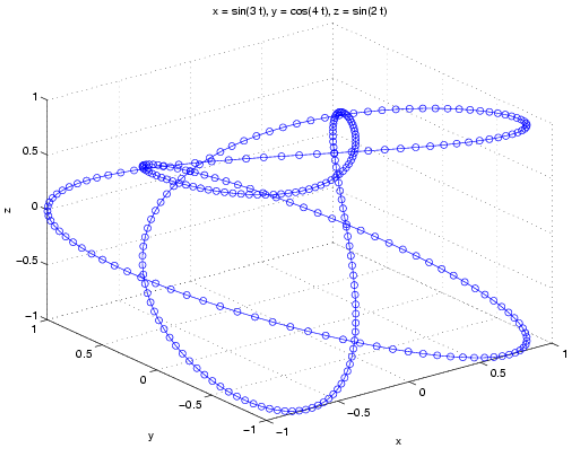
Der Befehl `rotate3d` ermöglicht eine Drehung des Achsensystems mit Hilfe der Maus.

11.2.2.2 Ezplot3

Die 'Easy to Plot' Version von `plot3` zeichnet die durch x(t), y(t) und z(t) definierte parametrische 3D-Kurve, wobei x, y und z von t abhängige Funktionen sind.

`ezplot3` — `graph_ezplot3.m`

```
h=ezplot3('sin(3*t)','cos(4*t)',.
          'sin(2*t)',[0,2*pi]);

set(h, 'marker','o')
rotate3d
```



Die Grenzen von t sind, wenn nicht anders festgelegt, 0 und 2 pi, die Achsenbeschriftung erfolgt automatisch.

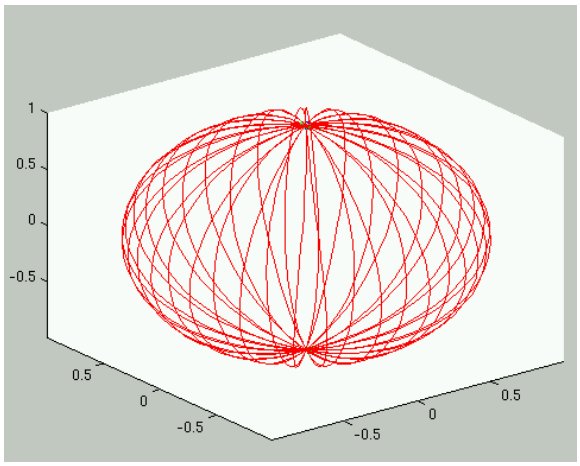
11.2.2.3 Comet3

Erstellt eine 3 dimensionale Funktion in Form eines sich bewegenden 'Kometen', dessen Schweif bzw. Spur den Graphen darstellt.

`comet3` `graph_comet3.m`

Optional kann in `comet3` die Schweiflänge relativ zur Gesamtlänge des Graphen angegeben werden.

```
t=linspace(0,2*pi,1000);

x=cos(t).*sin(20*t);
y=sin(t).*sin(20*t);
z=cos(20*t);

comet3(x,y,z);
```



Achtung, die Erstellung des Graphen erfolgt im `erasemode none`, wird das Graphikfenster vergrößert, verschwindet der Graph, er kann daher auch nicht gedruckt werden.

Tabelle 11.6: MATLAB Befehle zum Erzeugen von 3D-Balken- und Kreisdiagrammen

`bar3(x,y,w,'style')`	11.2.2.5	Stellt die 2D Daten als vertikale 3D Balken dar
`bar3h(x,y,w,'style')`	11.2.2.6	Stellt die 2D Daten als horizontale 3D Balken dar
`pie3(x,'explode')`	11.2.2.7	Zeichnet ein 3D Kreisdiagramm von x

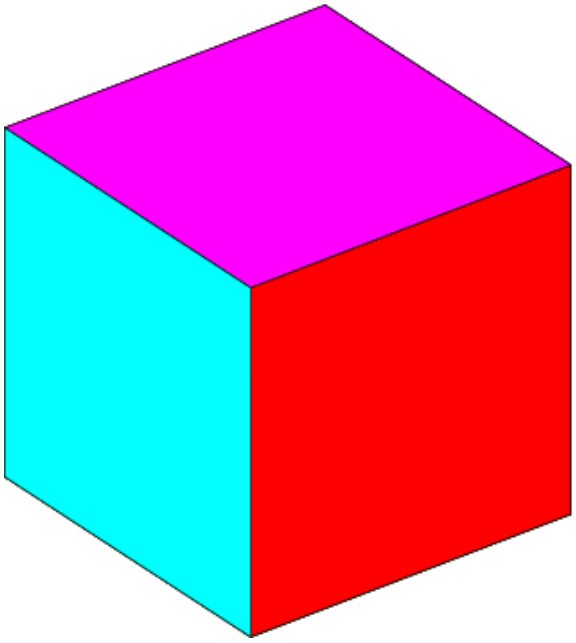
11.2.2.4 Fill3

Zeichnet dreidimensionale Polygone durch Angabe der Eckpunkte sowie der Füllfarben. Die Punkte werden in Form von Vektoren für die x-, y- und z- Komponenten angegeben, die Farbe c als Index in der aktuellen `colormap`.

`fill3` `graph_fill3.m`

Definition der 6 Flächen eines Würfels:

```
x=[0,1,1,0;0,1,1,0;1,1,1,1;...
   0,1,1,0;0,1,1,0;0,0,0,0]';
y=[0,0,0,0;0,0,1,1;0,1,1,0;...
   1,1,1,1;0,0,1,1;0,1,1,0]';
z=[0,0,1,1;0,0,0,0;0,0,1,1;...
   0,0,1,1;1,1,1,1;0,0,1,1]';

colormap([1,0,0;0,1,0;0,0,1;...
          1,1,0;1,0,1;0,1,1]);

fill3(x,y,z,1:6)
```



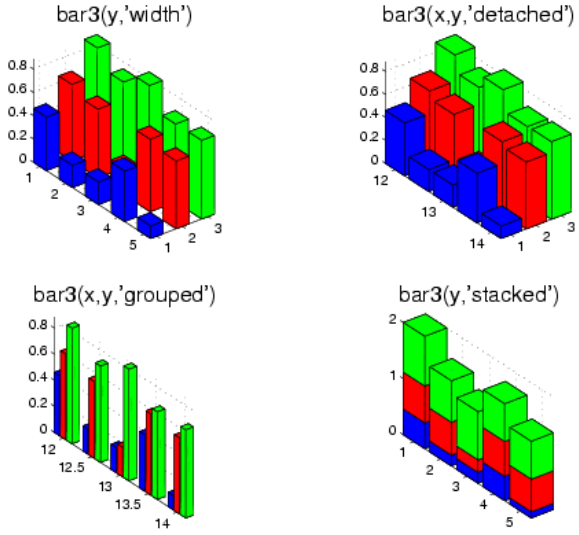
11.2.2.5 Bar3

Daten von y werden entlang der Abszisse als vertikale Säulen der Breite w dargestellt.

`bar3` `graph_bar3.m`

Wird der Vektor x angegeben, so werden die Säulen an den Positionen von x aufgetragen, sonst bei den Werten von 1 bis length(n)

```
y=sort(rand(3,5))';
x=linspace(12,14,size(y,1));
colormap([0,0,1;1,0,0;0,1,0]);

subplot(2,2,1)
bar3(y,0.5)

subplot(2,2,2)
bar3(x,y,'detached')

subplot(2,2,3)
bar3(x,y,'grouped')

subplot(2,2,4)
bar3(y,'stacked')
```



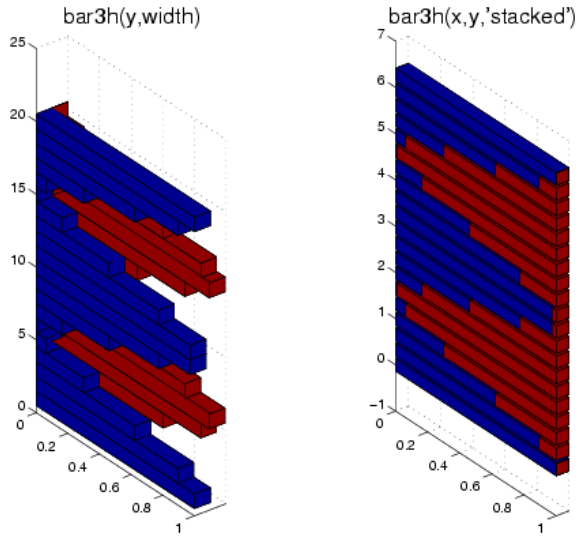
Man beachte die unterschiedliche Darstellung der Säulendiagramme bei der Verwendung der Stile 'detached', 'grouped' und 'stacked'.

11.2.2.6 Bar3h

Daten von y werden als horizontale Säulen der Breite w gezeichnet.

`bar3h` `graph_bar3h.m`

```
x=linspace(0,2*pi,20)';
y=[cos(x).^2,sin(x).^2];

subplot(1,2,1)
bar3h(y,1);

subplot(1,2,2);
bar3h(x,y,'stacked');
```



Hier gilt dasselbe wie bei `bar3` mit dem Unterschied, dass hier Ordinate und Abszisse vertauscht sind.

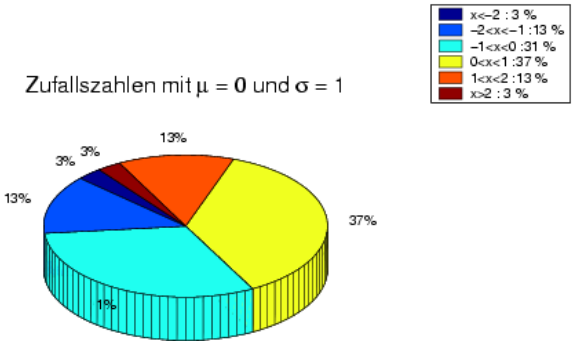
11.2.2.7 Pie3

Die Daten des Vektors x werden als 3D-Kreisdiagramme dargestellt, wobei die Segmente optional mit Hilfe des Vektors 'explode' hervorgehoben werden können.

`pie3` `graph_pie3.m`

Anteile normalverteilter Daten innerhalb bestimmter Intervalle (siehe Legende)

```
x=randn(1000,1);
y1=length(x(find(x<-2)));
y2=length(x(find(d<-1 & x>-2)));
y3=length(x(find(x<0 & x>-1)));
y4=length(x(find(x<1 & x>0)));
y5=length(x(find(x<2 & x>1)));
y6=length(x(find(x>2)));
y=[y1,y2,y3,y4,y5,y6];

h=pie3(y);
```



Der Vektor 'explode' muß die selbe Länge wie x aufweisen, Einträge des Wertes 1 führen zur Betonung des entsprechenden Segments.

Tabelle 11.7: MATLAB Befehle zum Erstellen von 3D - Oberflächen

`contour3(x,y,z)`	11.2.2.8	Zeichnet durch z=f(x,y) definierte 3D-Konturlinien
`mesh(x,y,z)`	11.2.2.9	Stellt die Matrix z=f(x,y) in Form eines 'Drahtgitters' dar
`ezmesh('f(x,y)')`	11.2.2.10	'Easy to use' Variante von mesh, f(x,y) wird als String eingegeben
`meshc(x,y,z)`	11.2.2.11	Zeichnet ein 3D-Drahtgitter und einen 2D-Contourplot der Funktion z=f(x,y)
`meshz(x,y,z)`	11.2.2.12	Zeichnet ein 3D-Drahtgitter der Funktion z=f(x,y) mit zusätzlichen seitlichen Referenzlinien
`trimesh(tri,x,y,z)`	11.2.2.13	Zeichnet ein aus Dreiecken bestehendes 3D-Drahtgitter der Funktion z=f(x,y)
`surf(x,y,z)`	11.2.2.14	Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y)
`ezsurf('f(x,y)')`	11.2.2.15	'Easy to use' Variante von surf, f(x,y) wird als String eingegeben
`surfc(x,y,z)`	11.2.2.16	Zeichnet eine 3D-Oberflächengraphik und einen 2D-Contourplot der Funktion z=f(x,y)
`ezsurfc(x,y,z)`	11.2.2.17	'Easy to use' Variante von surfc, f(x,y) wird als String eingegeben
`surfl(x,y,z)`	11.2.2.18	Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y) mit wählbarer Beleuchtung
`trisurf(tri,x,y,z)`	11.2.2.19	Zeichnet eine 3D-Oberfläche der Funktion z=f(x,y) aus Dreiecken
`waterfall(x,y,z)`	11.2.2.20	Zeichnet die Reihen der Matrix z=f(x,y) als 3D-Linien entlang der x-Achse

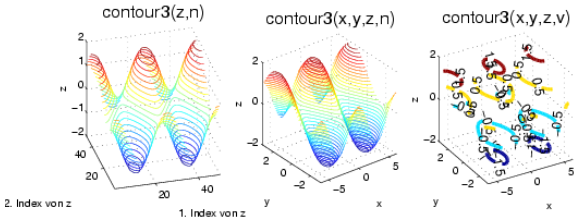
11.2.2.8 Contour3

Zeichnet z als Funktion von x und y in Form von 3D-Konturlinien (Höhenlinien), die je nach Aufruf von `contour3` äquidistant sind oder bei bestimmten Werten von z liegen.

`contour3` `graph_contour3.m`

Text in Spalten

```
x=linspace(-2*pi,2*pi,50);
y=linspace(-pi,pi,50);
[xx,yy]=meshgrid(x,y);
zz=cos(xx)+sin(yy);

subplot(2,2,1)
contour3(zz,20);

subplot(2,2,2)
contour3(xx,yy,zz,30);

subplot(2,2,4)
v=[-1.5,-0.5,0.5,1.5];
[c,h]=contour3(xx,yy,zz,v);
clabel(c,h,'fontsize',12);
```



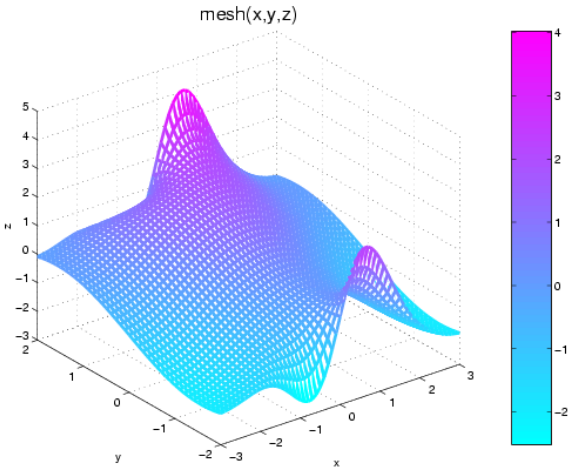
Mit `clabel` werden die Konturlinien mit den entsprechenden z-Werten beschriftet.

11.2.2.9 Mesh

Zeichnet die Funktion z=f(x,y) in Form eines Drahtgittermodells.

`mesh` `graph_mesh.m`

```
[x,y]=meshgrid(-3:0.1:3,-2:0.1:2)
z1=x.*exp(-x.^2+y.^2);
z2=10+cos(x)+sin(y);
z=z1./z2;

h=mesh(x,y,z);

set(h,'linewidth',2.5);
colormap cool
colorbar
```



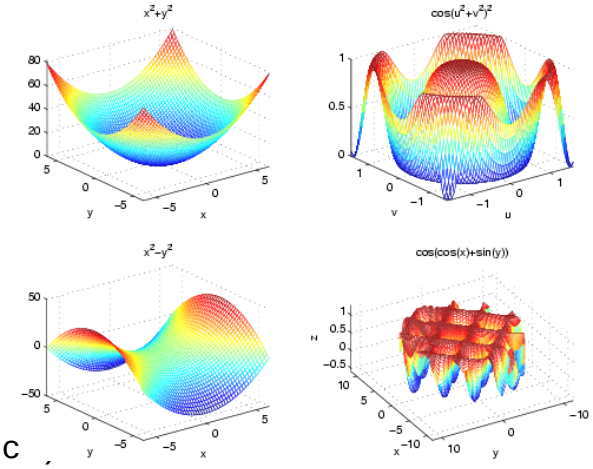
Zur Erinnerung: mit `get(h)` können alle Eigenschaften des mit dem Handle h verknüpften Graphik-Objekts ausgegeben und mit `set(h,'Eigenschaft','Wert')` gesetzt werden.

11.2.2.10 Ezmesh

'Easy to use' Variante von mesh, die als String eingegebene Funktion f(x,y) wird als Drahtgittermodell gezeichnet, Achsenbeschriftung und Titel werden automatisch hinzugefügt.

`ezmesh` `graph_ezmesh.m`

```
subplot(2,2,1)
ezmesh('x^2+y^2')

subplot(2,2,2)
ezmesh('cos(u^2+v^2)^2',...
       [-pi/2,pi/2])

subplot(2,2,3)
ezmesh('x^2-y^2',50)

subplot(2,2,4)
ezmesh('cos(cos(x)+sin(y))','circ
```



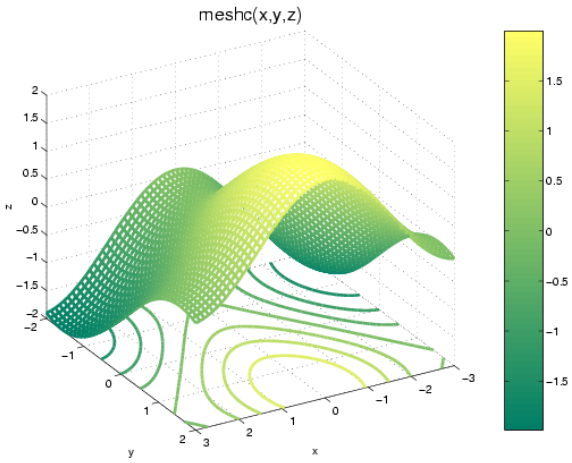
Neben der Funktion f(x,y) können optional die Grenzen von x und y, die Anzahl der Gitterelemente oder der Ausdruck 'circ' (zeichnet Graphik über kreisförmigen Definitionsgebiet) angegeben werden.

11.2.2.11 Meshc

Die Funktion z=f(x,y) wird als 'Drahtgittermodell' inklusive 2D-Konturlinien in der Ebene z = 0 gezeichnet.

`meshc` `graph_meshc.m`

```
[x,y]=meshgrid(-3:0.1:3,-2:0.1:2)
z1=;x.*exp(-x.^2-y.^2)
z2=10+cos(x)+sin(y);
z=z1./z2;

h=meshc(x,y,z);

set(h,'linewidth',2.5);
```



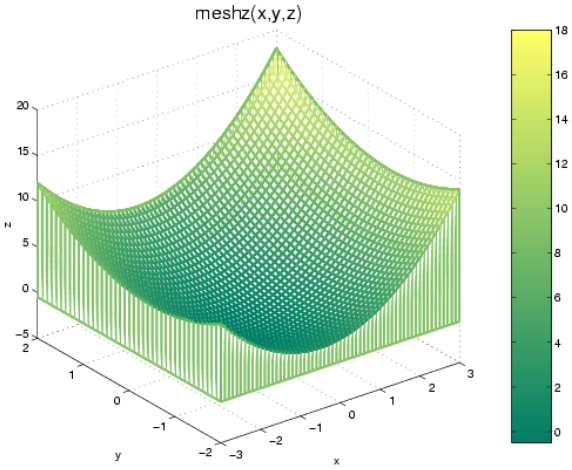
Die Dicke der Konturlinien kann nur gemeinsam mit jenen des Drahtgitters verändert werden.

11.2.2.12 Meshz

Zeichnet die Funktion z=f(x,y) als 'Drahtgittermodell', wobei die Ränder des Gitters mit der durch z=0 definierten Ebene verbunden sind.

`meshz` `graph_meshz.m`

```
[x,y]=meshgrid(-3:0.1:3,-2:0.1:2)
z=x+y+x.^2+y.^2;

h=meshz(x,y,z);

colorbar
set(h,'linewidth',2.0);
colormap summer
```



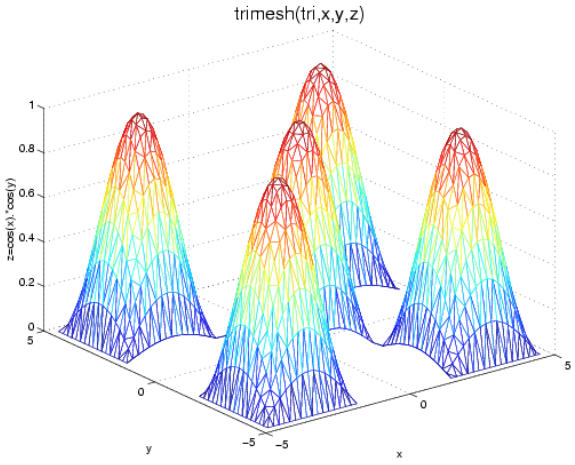
11.2.2.13 Trimesh

Zeichnet ein aus Dreiecken bestehendes 3D-Drahtgitter der Funktion z=f(x,y).

`trimesh` `graph_trimesh.m`

Die Koordinaten (tri) der Dreiecke werden mit der `delaunay` Triangulation aus den (x,y) Daten gewonnen.

```
t=linspace(-1.5*pi,1.5*pi,50);
[x,y]=meshgrid(t,t);
z=cos(x).*cos(y);
z(z<0)=nan;
tri = delaunay(x,y);

trimesh(tri,x,y,z)
```



Elemente der Matrix z mit dem Eintrag `nan` werden nicht gezeichnet.

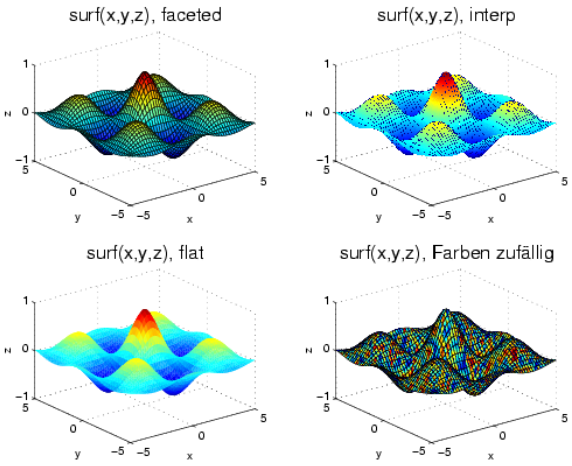
11.2.2.14 Surf

Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y) mit dem in `shading` spezifizierten Schattiermodus.

`surf` `graph_surf.m`

Die Farbgebung im 4. Subplot erfolgt zufällig.

```
x=linspace(-5,5,50);
y=linspace(-5,5,50);
[xx,yy]=meshgrid(x,y);
z1=cos(xx).*cos(yy);
z2=exp(-0.2*sqrt(xx.^2+yy.^2));
zz=z1.*z2;

subplot(2,2,1)
surf(xx,yy,zz);
shading faceted

subplot(2,2,2)
surf(xx,yy,zz);
shading interp

subplot(2,2,3)
surf(xx,yy,zz);
shading flat

subplot(2,2,4)
h=surf(xx,yy,zz);
shading interp
set(h,'cdata',rand(size(zz)),'edgecolor','k')
```



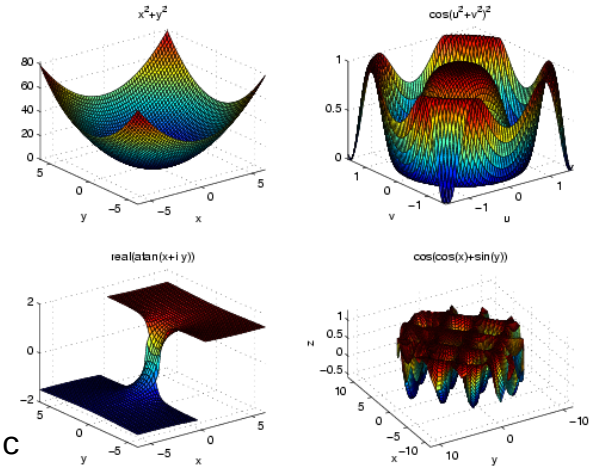
Werden im Aufruf von `surf` die x- und y- Matrizen weggelassen, so werden auf den x- und y- Achsen die beiden Indizes der Matrix z aufgetragen.

11.2.2.15 Ezsurf

Die 'Easy to use' Variante von `surf` mit automatischer Achsenbeschriftung und Überschrift.

`ezsurf` — `graph_ezsurf.m`

```
subplot(2,2,1)
ezsurf('x^2+y^2')

subplot(2,2,2)
ezsurf('cos(u^2+v^2)^2',...
       [-pi/2,pi/2])

subplot(2,2,3)
i=sqrt(-1);
ezsurf('real(atan(x+i*y))',50)

subplot(2,2,4)
ezsurf('cos(cos(x)+sin(y))','circ
view(-120,50)
```



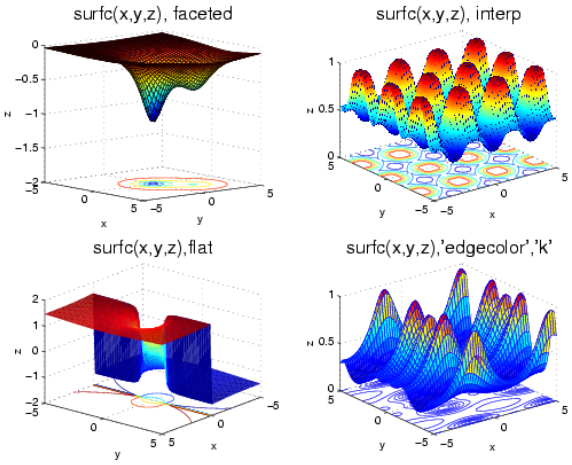
Neben der Funktion f(x,y) können optional die Grenzen von x und y, die Anzahl der Gitterelemente oder der Ausdruck 'circ' (zeichnet Graphik über kreisförmigen Definitionsgebiet) angegeben werden. Mit Hilfe des Befehls `view` stellt man den Blickwinkel auf das Achsensystem ein. Die erste Komponente ist der Azimuthwinkel in Grad (Rotation der x,y Ebene), die zweite Komponente ist der Kippwinkel aus der horizontalen Lage der x,y Ebene.

11.2.2.16 Surfc

Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y) mit dem in `shading` spezifizierten Schattiermodus und fügt 2D-Konturlinien in der Ebene z = 0 hinzu.

`surfc` `graph_surfc.m`

```
x=linspace(-5,5,50);
[xx,yy]=meshgrid(x,x);;

subplot(2,2,1)
zz=-1./(xx.^2+yy.^2+1)-1./...
      ((xx-2).^2+(yy-2).^2+2);
surfc(xx,yy,zz)
shading faceted

subplot(2,2,2)
zz=1./(cos(xx).^4+sin(yy).^4+1);
surfc(xx,yy,zz)
shading interp

subplot(2,2,3)
zz=real(atan(xx+sqrt(-1)*yy));
surfc(xx,yy,zz);
shading flat

subplot(2,2,4)
zz=1./(sin(xx)+2+abs(yy).*cos(yy).^2);
h=surfc(xx,yy,zz);
set(h,'edgecolor','b')
```

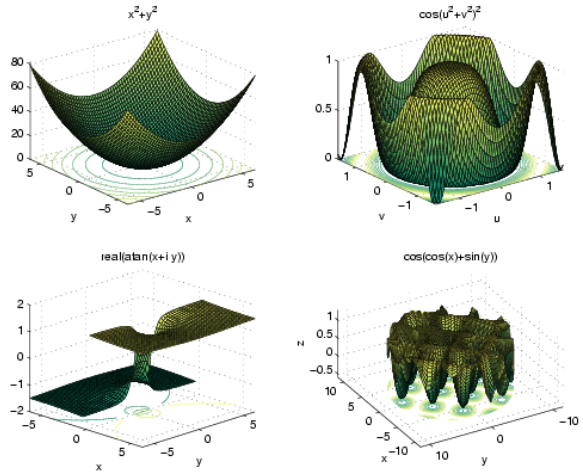


11.2.2.17 Ezsurfc

Die 'Easy to use' Variante von `surfc` mit automatischer Achsenbeschriftung und Überschrift.

`ezsurfc` `graph_ezsurfc.m`

```
subplot(2,2,1)
ezsurfc('x^2+y^2')

subplot(2,2,2)
ezsurfc('cos(u^2+v^2)^2',...
        [-pi/2,pi/2])

subplot(2,2,3)
i=sqrt(-1);
ezsurfc('real(atan(x+i*y))',50)
view(45,25)

subplot(2,2,4)
ezsurfc('cos(cos(x)+sin(y))','circ')
```

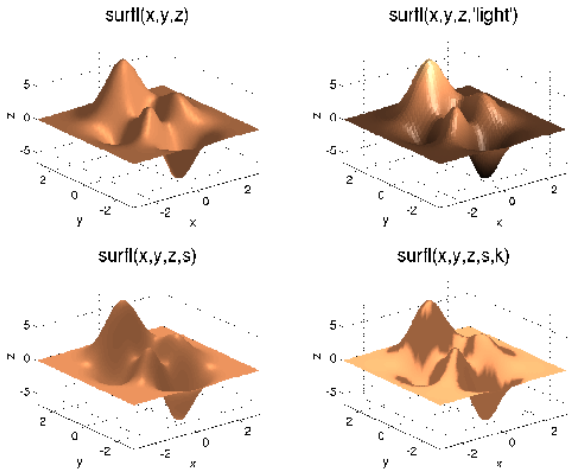


11.2.2.18 Surfl

Erstellt beleuchtete 3D Oberflächenplots einer Funktion z=f(x,y).

`surfl` `graph_surfl.m`

```
[x,y] = meshgrid(-3:1/8:3);
z = peaks(x,y);

subplot(2,2,1)
surfl(x,y,z);

subplot(2,2,2)
surfl(x,y,z,'light')

subplot(2,2,3)
s=[0,90];
surfl(x,y,z,s)

subplot(2,2,4)
s=[0,90];
k=[1,0.1,1,0.1];
surfl(x,y,z,s,k)
```



Der Vektor s beinhaltet die x-, y- und z- Komponenten der Einfallsrichtung des Lichts und k die relativen Intensitäten des Umgebungslichtes, der diffusen Reflexion, der spiegelnden Reflexion sowie des spiegelnden Glanzes.

11.2.2.19 Trisurf

Zeichnet eine aus Dreiecken bestehende Oberflächengraphik der Funktion z=f(x,y).

`trisurf` `graph_trisurf.m`

Die Koordinaten der Dreiecke werden mittels `delaunay` aus den x- und y-Werten des Gitters gewonnen.

```
t=linspace(-1.5*pi,1.5*pi,25);
[x,y]=meshgrid(t,t);
z=cos(x+cos(y));
z(z<0)=0;
tri = delaunay(x,y);

h=trisurf(tri,x,y,z);

shading interp
set(h,'edgecolor','k')
```

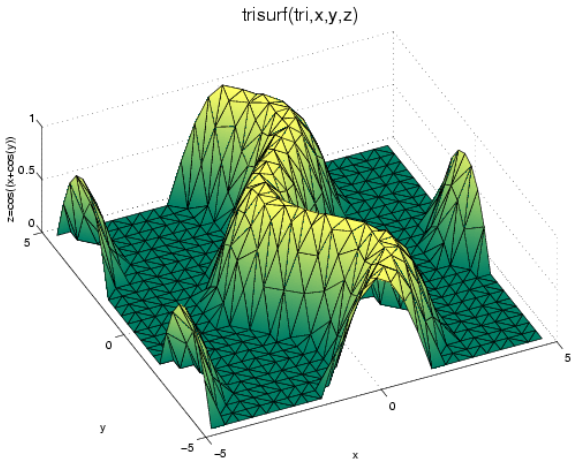


Tabelle 11.8: MATLAB Befehle zum Erstellen von 3D - volumetrischen Graphiken

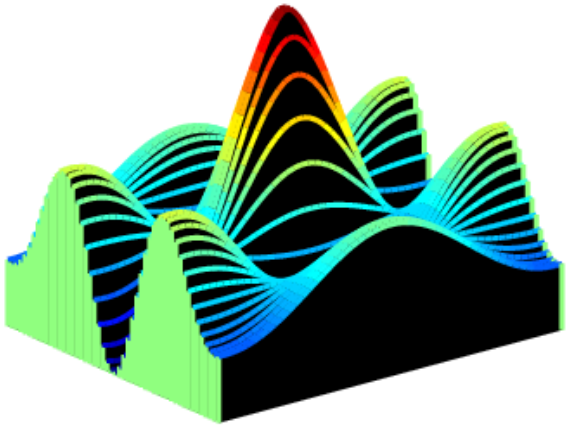
`quiver3(x,y,z,u,v,w)`	11.2.2.21	Zeichnet an den Punkten (x,y,z) Vektorpfeile mit den Komponenten (u,v,w)
`slice(x,y,z,d,sx,sy,sz)`	11.2.2.22	Veranschaulicht die volumetrische Funktion d=f(x,y,z) durch senkrecht durch die Achsen gelegte Schnittflächen

11.2.2.20 Waterfall

Zeichnet die Reihen der Matrix z=f(x,y) als 3D-Linien entlang der x-Achse

`waterfall` `graph_waterfall.m`

```
x=linspace(-pi,pi,50);
y=linspace(-2*pi,2*pi,50);
[xx,yy]=meshgrid(x,y);
z1=cos(xx).*cos(yy);
z2=exp(-(sqrt(xx.^2+yy.^2))./4;
zz=z1.*z2;

h=waterfall(xx,yy,zz);

set(h,'linewidth',3,'facecolor','
set(gcf,'color','k');
```



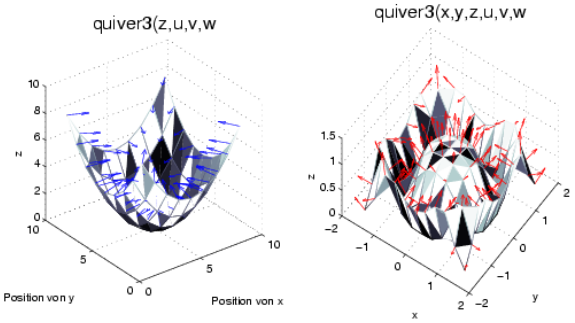
11.2.2.21 Quiver3

Zeichnet an den Punkten (x,y,z) Vektorpfeile mit den Komponenten (u,v,w).

`quiver3` — `graph_quiver3.m`

Es ist sinnvoll, diesen Graphikbefehl gemeinsam mit `mesh` oder `surf` zu verwenden.

```
subplot(1,2,1)
[x,y]=meshgrid(-2:0.5:2,-2:0.5:2);
z=x.^2+y.^2;
[u,v,w] = surfnorm(x,y,z);

quiver3(z,u,v,w)

hold on
mesh(z)

subplot(1,2,2)
[x,y]=meshgrid(-pi/2:pi/10:pi/2);
z=cos(x.^2+y.^2).^2;
[u,v,w] = surfnorm(x,y,z);

quiver3(x,y,z,u,v,w,'r')

hold on
mesh(x,y,z)
```



Die Komponenten der Normalvektoren auf die Oberfläche z=f(x,y) werden mit dem Befehl `[u,v,w]=surfnorm(x,y,z)` berechnet.

Tabelle 11.9: Weitere spezielle 3D Graphik-Befehle

`stem3(x,y,z)`	11.2.2.23	Zeichnet 3D Funktion und verbindet Datenpunkte mit der Ebene z=0
`sphere(n)`	11.2.2.24	Erstellt eine durch n^2 Flächen angenäherte Kugel
`cylinder(r,n)`	11.2.2.25	Erstellt einen durch ein n-seitiges Prisma angenäherten Zylinder mit Radius r
`scatter3(x,y,z,r,c)`	11.2.2.26	Zeichnet Daten an den Positionen (x,y,z) der Größe r sowie der Farbe c
`ribbon(y,z,w)`	11.2.2.27	Zeichnet die Spalten von z über jenen von y als 3D Bänder der Breite w

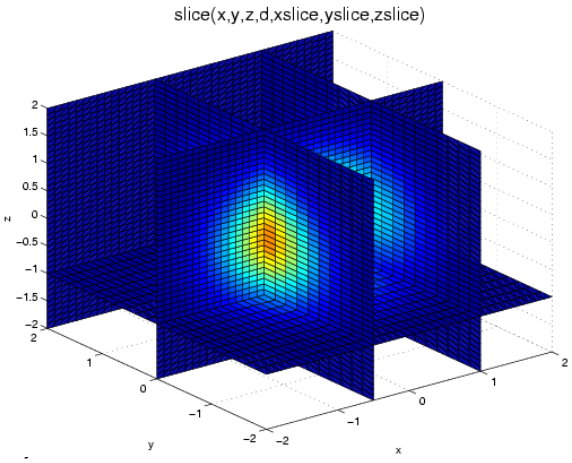
11.2.2.22 Slice

Veranschaulicht die volumetrische Funktion d=f(x,y,z) durch senkrecht durch die Achsen gelegte Schnittflächen. Dabei wird die x-Achse an den Stellen des Vektors xslice geschnitten, analog für die beiden anderen Achsen.

`slice` `graph_slice.m`

Wie zu den Achsen geneigte Schnittflächen erstellt werden, findet man in in der Hilfe von `slice`

```
[x,y,z] = meshgrid(-2:.1:2,...
          -2:.2:2,-2:.1:2);
d=exp(-x.^2-y.^2-z.^2);
xslice = [-0.5,1];
yslice = [0,2];
zslice = [-1];

slice(x,y,z,d,xslice,yslice,zslic
```



Mit `meshgrid` lassen sich auch die x-, y- und z- Koordinaten dreidimensionaler Gitter berechnen.

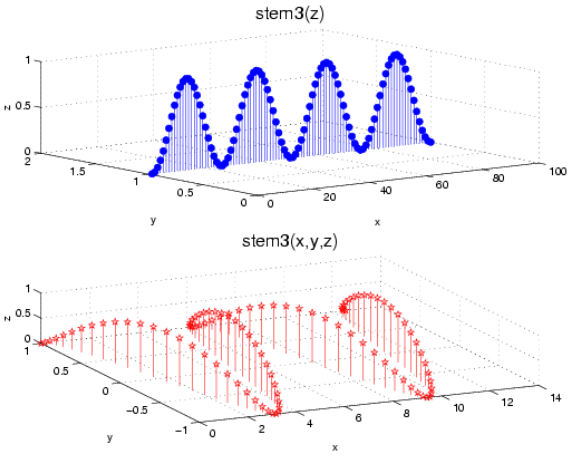
11.2.2.23 Stem3

Zeichnet dreidimensionale Daten und verbindet Datenpunkte mit der Ebene z=0.

`stem3` `graph_stem3.m`

Die in `linespec` definierten Datensymbole können mit der Option 'filled' ausgefüllt werden.

```
t=linspace(0,4*pi,100);
x=t;
y=cos(t);
z=sin(t).^2;

subplot(2,1,1)
stem3(z,'filled')

subplot(2,1,2);
stem3(x,y,z,'rp')

view(-25,60)
```



Wird `stem3` nur der Vektor z übergeben, dann wird z über x=1 bis size(z,1) und y=1 bis size(z,2) aufgetragen.

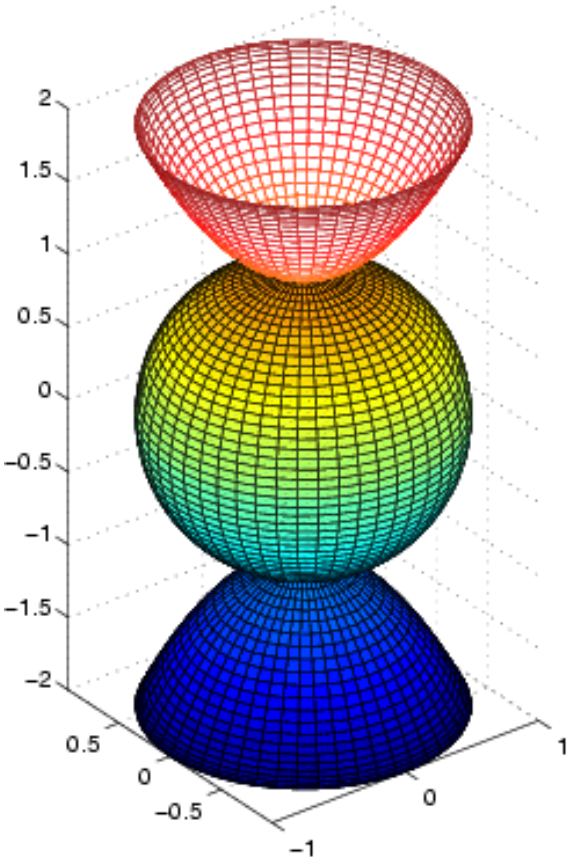
11.2.2.24 Kugel

Erstellt eine durch n*n Segmenten angenäherte Kugel mit dem Radius 1.

`sphere` `graph_sphere.m`

Einheitskugel mit vertikal angrenzenden paraboloid-ähnlichen Objekten.

```
sphere(50)
[x,y,z]=sphere(50);

hold on
mesh(x,y,-z.^2+2)

surf(x,y,z.^2-2)
axis equal
```



Wird der Befehl in Form von [x,y,z]=`sphere(n)` verwendet, so können wie im Beispiel mit `surf(x,y,z)` oder `mesh(x,y,z)` ebenfalls Kugeln und kugelähnliche Objekte gezeichnet werden. Der Vorteil liegt darin, dass auf diese Weise Eigenschaften wie Größe, Position und Farben beeinflusst werden können.

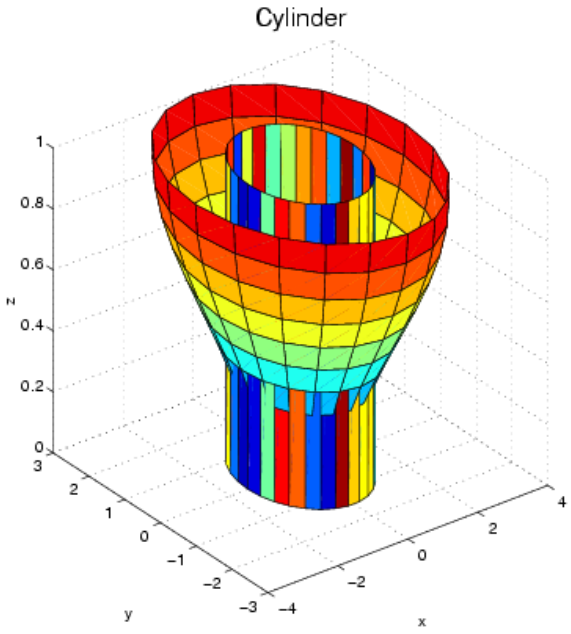
11.2.2.25 Zylinder

Erstellt Zylinder (bzw. n-seitige Prismen) und allgemeine um die z-Achse symmetrische Körper der Höhe 1 mit der Profilkurve r(h).

`cylinder` `graph_cylinder.m`

Zylinder und Rotationskörper mit der Profilkurve r(t)=2+cos(t)

```
t = pi:pi/10:2*pi;
[X1,Y1,Z1] = cylinder(2+cos(t));
[X2,Y2,Z2] = cylinder(1.5,30);

h1=surf(X1,Y1,Z1);
hold on
h2=surf(X2,Y2,Z2);
c=rand(size(get(h2,'cdata')));

set(h2,'cdata',c)
axis square
```



Wird der Befehl in Form von [x,y,z]=`cylinder(r,n)` verwendet, so können wie im Beispiel mit `surf(x,y,z)` oder `mesh(x,y,z)` ebenfalls Rotationskörper gezeichnet werden. Der Vorteil liegt wie im Beispiel 11.2.2.24 darin, dass auf diese Weise unter anderem Größe, Position und Farbeigenschaften beeinflusst werden können.

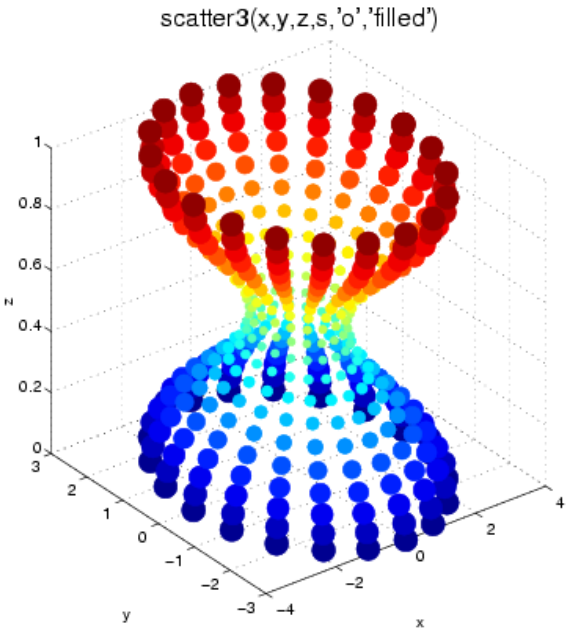
11.2.2.26 Scatter3

Zeichnet Daten an den Positionen (x,y,z) der Größe r sowie der Farbe c, wobei im Gegensatz zu `plot3` die Attribute Größe und Farbe für jeden Punkt getrennt eingestellt werden können. Allen Punkten gemeinsam ist das Datensymbol (siehe `linespec`) sowie die Option 'filled', wodurch Datensymbole ausgemalt werden.

`scatter3` `graph_scatter3.m`

Mit Hilfe des Graphikbefehls `cylinder` erhaltene Koordinaten des Rotationskörpers der Profilkurve r(t)=2+cos(t). Farbe und Punktgröße hängen von den Koordinaten ab.

```
t = 0:pi/10:2*pi;
[x,y,z] = cylinder(2+cos(t));

vx=reshape(x,[],1);
vy=reshape(y,[],1);
vz=reshape(z,[],1);
r=25*((vx.^2)+(vy).^2)
c=vz;;

scatter3(vx,vy,vz,r,c,'o','filled
```

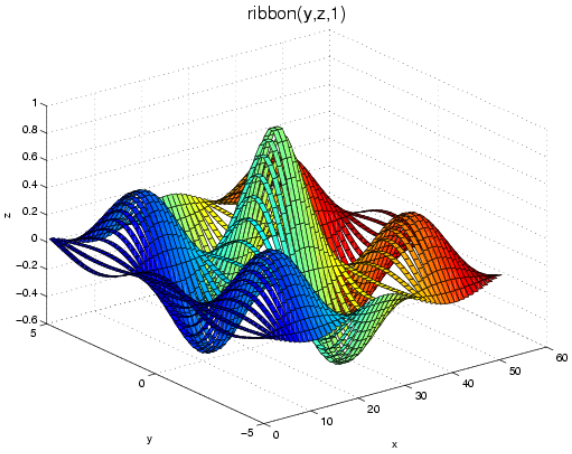


11.2.2.27 Ribbon

Zeichnet die Spalten von z über jenen von y als 3D Bänder der Breite w

ribbon graph_ribbon.m

```
x=linspace(-5,5,50);
y=linspace(-5,5,50);
[xx,yy]=meshgrid(x,y);
z1=cos(xx).*cos(yy);
z2=exp(-0.2*sqrt(xx.^2+yy.^2));
zz=z1.*z2;

ribbon(yy,zz,1)
```



Kapitel 12

Übungsbeispiele

Achtung: Der Inhalt der Übungen kann sich im Laufe des Semesters ändern. Überprüfen sie immer vor Beginn der Übung den jeweiligen Inhalt.

Die Programme zu den einzelnen Übungen werden mit Hilfe eines bereitgestellten Scripts abgegeben. Sie werden direkt an einem für sie vorbestimmten Platz gespeichert, wo sie beurteilt werden können.

Der Name des Scripts ist:

- `uebungsabgabe file1 file2 ...`
- `uebungsstatus` zeigt Ihnen den jeweiligen Status (abgegeben, korrigiert, mangelhaft, ...) an.

In MATLAB können Sie die Skripts mit vorangestellten Rufzeichen aufrufen. Falls Sie einen Fehler vermuten, können Sie Ihr Beispiel auch ändern und wieder übermitteln.

12.1 Mathematische Funktionen

Ziel: Ein erster Einstieg in MATLAB unter Linux soll geschafft werden. Editieren, Speichern und Ausführen von MATLAB-Skripten. Aufruf einfacher eingebauter Funktionen, Einführung in die Sprachsyntax, einfache graphische Ausgabe.

Vorraussetzung: Grundlagen von Linux und MATLAB.

Abgabe: Die Abgabe erfolgt mit Hilfe des Skripts

```
uebungsabgabe  "filename"       % im Terminal
!uebungsabgabe "filename"       % in Matlab
```

12.1.1 Der Beginn

Das Programm MATLAB kann in einem Terminal mit dem Befehl `matlab` gestartet werden. Besser ist die Verwendung des Befehls `matlab &`, da damit MATLAB im Hintergrund läuft und das Terminal zur Verwendung frei ist. Vergisst man auf das Zeichen `&`, kann man das Programm durch Eingabe im Terminal von `Strg z` (drücken von `z` bei gleichzeitig gedrückter Control-Taste `Strg` oder `Ctrl`) und dannach `bg` (background) in den Hintergrund schieben.

Im Normalfall sollte es auch möglich sein, MATLAB von einem Icon am Desktop aus zu starten. Nach dem Starten erhält man ein MATLAB-Fenster in dem man erste Befehle eingeben kann. Probieren Sie ein paar Dinge nach Lust und Laune aus.

```
3 + 4
a = 3
b = 5 * a
b = 5 * c               % Fehler, warum wohl?
s = sin(a)
s = sin(pi)             % pi ist bekannt!
s = sin(pi/2)
x = [0:4]               % Erzeugen von Vektoren
x = [0:0.5:4]           % Schrittweite ungleich 1
y = sin(x);             % Was macht der Strichpunkt?
disp(y)                 % Anzeige
plot(x,y)               % Was bekommt man da?
u = [0:5]; v = [10:15];
w = u + v
w = u * v               % Fehler, warum wohl?
w = u .* v              % Richtiger Operator .* (oder ./ .^)
```

Nach Eingabe des Befehls `helpbrowser` öffnet sich ein Fenster in dem die gesamte MATLAB-Hilfe in einem Webbrowser zugänglich ist. Hilfe bekommt man z.B. auch mit dem Befehl `help sin`, falls man den Befehlsnamen kennt, oder mit `lookfor hyperbolic`, falls man den Namen nicht kennt. Die gesamte Hilfe ist aber in englischer Sprache verfasst.

12.1.2 Editor

Mit dem Befehl `edit` öffnet sich ein Editor Fenster, in dem Sie Files bearbeiten können. Als Standard-Editor sollte bei uns `emacs` eingestellt sein.

Schreiben Sie im Editor ein MATLAB-Skript `uebung1.m`, das für Sie die im folgenden beschriebenen Dinge tun soll.

Ein solches MATLAB-Skript sollte mit ein paar Kommentarzeilen beginnen (Name, Übung, Datum). Damit MATLAB weiss, dass es sich nicht um Programmanweisungen handelt, müssen solche Zeilen mit dem Prozentzeichen `%` beginnen. Füllen Sie also diese Zeilen aus, speichern den File unter dem Namen `uebung1.m` und geben Sie im MATLAB-Fenster `uebung1` ein. Eigentlich sollte gar nichts passieren, da das Programm keine ausführbaren Zeilen enthält.

Probieren Sie jetzt auch mal `help uebung1`. Was fällt Ihnen auf?

Anmerkung: Das Speichern erfolgt über die Menüeinträge oder mit praktischen Kurzbefehlen. Im `emacs` speichert `C-x C-s`, wobei `C` für das gleichzeitige Drücken der `Strg`-Taste steht. Im MATLAB-Mode ist auch `C-c C-s` sehr praktisch. Dieser Befehl speichert das Skript und führt es in MATLAB aus.

Schreiben Sie bei dieser ersten Übung auch folgende Zeilen in das MATLAB-Skript

```
clear all           % löscht alle Variablen
close all           % schließt Graphik-Fenster
clc                 % löscht den Inhalt des Command-Fensters
```

Nun kann man mit dem eigentlichen Programmieren beginnen. Weisen Sie den Variablen `xstart`, `xend`, `xdelta` die Werte -1, 1, bzw. 0.05 zu. Erzeugen Sie damit die Vektoren x (von `xstart` bis `xend` mit der Schrittweite `xdelta`) und den Vektor $x_2 = 2x$. In praktisch allen Programmiersprachen schreibt man für Größen mit Indices (wie hier x_2 den Variablenname mit dem Zeichen Unterstrich _ (`x_2`).

12.1.3 Erste Berechnungen

Führen Sie erste Berechnungen aus und geben Sie auf die korrekte Verwendung der Operatoren acht. Denken Sie auch die Verwendung des Strichpunktes am Zeilenende.

$$
\begin{aligned}
y_1 &= \sin \pi x \\
y_2 &= y_1 + x \\
y_3 &= y_1 x \\
y_4 &= y_1^2
\end{aligned}
$$

Am besten kann man die Ergebnisse mit Hilfe einer graphischen Ausgabe überprüfen. Für den Beginn gibt es hier einmal die Befehlsfolge für eine einfache graphische Ausgabe.

```
figure(1);
plot(x,y_1,x,y_2,x,y_3,x,y_4);
xlabel('x');
ylabel('y');
legend('y_1','y_2','y_3','y_4');
title('Functions with Sinus');
```

Die Befehle sollten eigentlich in ihrer einfachen Form selbsterklärend sein. Die vier Kurven $y_k(x)$ zeichnet dabei der Befehl `plot`.

Anmerkung: Zeichen zwischen einfachen Hochkommas `'` sind Zeichenketten und stellen keine Variablennamen dar (mehr dazu im Laufe der Lehrveranstaltung).

12.1.4 Hyperbelfunktionen

Berechnen Sie in gleicher Weise die Variablen `hsin`, `hcos`, `htan`, `hsec` mit den Hyperbelfunktion $\sinh x_2$, $\cosh x_2$, $\tanh x_2$ und $\operatorname{sech} x_2$.

Anmerkung: Wenn man eine Varibale z.B. `sinh` nennt, würde ab diesem Zeitpunkt die Funktion `sinh` nicht mehr funktionieren, da sie mit eigener Bedeutung überschrieben ist. Daher Vorsicht bei der Wahl der Variablennamen (`clear sinh` stellt die alte Funktionalität wieder her.

Machen Sie einen ähnlichen Plot wie oben in einem zweiten Graphikfenster, `figure(2)`.

Anmerkung: In Linux können Sie sehr einfach Teile von Bildschirminhalten an anderer Stelle duplizieren. Markieren Sie den zu kopierenden Text mit der linken Maustaste, setzen Sie den Cursor auf die Stelle, wo eingefügt werden soll und drücken Sie die mittlere Maustaste.

12.1.5 Gaussfunktion und Secans Hyperbolicus

Geben Sie den Variablen `x_0` und `s` die Werte 0 und 0.3. Berechnen Sie damit die Funktionen

$$g = \frac{1}{s\sqrt{2\pi}} \exp\left(-\frac{(x-x_0)^2}{2s^2}\right)$$
$$h = \frac{1}{\pi s} \operatorname{sech}\left(-\frac{x-x_0}{s}\right)$$

und stellen Sie sie in einem dritten Graphikfenster dar (`sqrt`, `exp`, `pi`).

12.1.6 Mathematische Identitäten

Schreiben Sie ein MATLAB-Skript `idents.m` und speichern Sie es in Ihrem MATLAB-Verzeichnis ab. Das Skript soll folgende Dinge tun:

- Lesen Sie zwei Zahlen a und b von der Tastatur ein (`input`)
- Prüfen Sie durch Ausrechnen der linken und rechten Seite der unten angeführten Identitäten die Richtigkeit der Formeln nach. Die Konstante $i = \sqrt{-1}$ steht in MATLAB zur Verfügung, sofern nicht vorher durch eine Anweisung `i=...` der Wert überschrieben wurde. Verwenden Sie die Variablennamen `vl1`, `vr1`, usw. für die Berechnungen.

1	e^{ia}	$\cos a + i\ \sin a$
2	$\log(a\,b)$	$\log a + \log b$
3	e^{a+b}	$e^a\, e^b$
4	$\cos(a+b)$	$\cos(a)\cos(b) - \sin(a)\sin(b)$
5	$\sin(a+b)$	$\sin(a)\cos(b) + \cos(a)\sin(b)$
6	$\sin(a)$	$\left(e^{ia} - e^{-ia}\right)/(2i)$
7	$\cos(a)$	$\left(e^{ia} + e^{-ia}\right)/2$
8	$\sin(a/2)$	$\sqrt{\frac{1-\cos(a)}{2}}$
9	$\cos(a/2)$	$\sqrt{\frac{1+\cos(a)}{2}}$

Geben Sie dazu das Ergebnis der linken und der rechten Seite der Gleichungen formatiert aus. Eine Einfache Möglichkeit dazu ist

```
disp([1,vl1,vr1])
disp([2,vl2,vr2])
```

Anmerkung: Mit Hilfe der eckigen Klammer kann man Zahlen zu Vektoren verbinden, die dann gemeinsam mit `disp` dargestellt werden können (mehr dazu im Laufe der Lehrveranstaltung).

12.2 Reguläre Polyeder, Kegelschnitte

Ziel: Umgang mit Funktionen.

Vorraussetzung: Grundlagen von Funktionen in MATLAB, wie sie im Kapitel 2 beschrieben werden. Wichtig ist vorallem das Verständnis des Unterschieds zwischen MATLAB-Skripts und MATLAB-Funktionen 2.7.

12.2.1 Reguläre Polyeder

Schreiben Sie eine MATLAB-Funktion `regpol.m`, die folgende Aufgaben erfüllt. Für eine vorgegebene Kantenlänge a soll das Volumen V, die Oberfläche F, der Radius der umbeschriebenen Kugel R und der Radius der einbeschriebenen Kugel r wahlweise für einen der 5 regulären konvexen Polyeder berechnet werden:

1. Tetraeder

$$V = \frac{a^3}{12}\sqrt{2} \qquad F = a^2\sqrt{3}$$
$$R = \frac{a}{4}\sqrt{6} \qquad r = \frac{a}{12}\sqrt{6}$$

2. Würfel

$$V = a^3 \qquad F = 6a^2$$
$$R = \frac{a}{2}\sqrt{3} \qquad r = \frac{a}{2}$$

3. Oktaeder

$$V = \frac{a^3}{3}\sqrt{2} \qquad F = 2a^2\sqrt{3}$$
$$R = \frac{a}{2}\sqrt{2} \qquad r = \frac{a}{6}\sqrt{6}$$

4. Dodekaeder

$$V = \frac{a^3}{4}\left(15 + 7\sqrt{5}\right) \qquad F = 3a^2\sqrt{5\left(5 + 2\sqrt{5}\right)}$$
$$R = \frac{a}{4}\left(1 + \sqrt{5}\right)\sqrt{3} \qquad r = \frac{a}{4}\sqrt{\frac{50 + 22\sqrt{5}}{5}}$$

5. Ikosaeder

$$V = \frac{5a^3}{12}\left(3+\sqrt{5}\right) \qquad F = 5a^2\sqrt{3}$$

$$R = \frac{a}{4}\sqrt{2\left(5+\sqrt{5}\right)} \qquad r = \frac{a}{2}\sqrt{\frac{7+3\sqrt{5}}{6}}$$

Der Aufruf der Funktion soll folgendermaßen erfolgen:

```
[V,F,R,r] = regpol(typ,a)
```

Das Programm soll für einen Vektor von a-Werten funktionieren und gleichlange Vektoren mit den Resultaten für V, F, R und r zurückgeben. Mit der String-Variablen `typ` soll der Typ des regulären Polyeders übergeben werden. Im Programm soll zur Unterscheidung der Fälle die `switch`-Konstruktion verwendet werden:

```
switch lower(typ(1))
   case 't'
      ...
   case 'w'
      ...
   ...
end
```

Überlegen Sie, welche Zeichenkette man als `typ` eingeben kann, was `typ(1)` bewirkt, und was der Befehl `lower` dabei bewirkt.

12.2.2 Polargleichung von Kegelschnitten

Schreiben Sie eine MATLAB-Funktion `kegelpol`, welche die Polargleichung von den Kegelschnitten Ellipse, Hyperbel und Parabel in Hauptachsenlage auswertet. Der Aufruf sollte mit

```
r = kegelpol(typ,phi,a,b)
```

erfolgen, wobei `typ` der Typ des Kegelschnitts ist, `phi` ein Vektor von ϕ-Werten, und `a` und `b` Konstanten zur Beschreibung des Kegelschnittes sind. Diese Werte werden der Funktion vom MATLAB-Skript `skegelpol` (siehe unten) übergeben. Der Vektor `phi` ist also ein Übergabeparameter und wird nicht in der Funktion selbst erzeugt. Vergessen Sie nicht die Verwendung der richtigen Operatoren.

Die Formeln für die drei Kegelschnitte lauten:

1. Ellipse

$$
\begin{aligned}
r^2 &= \frac{b^2}{1 - \epsilon^2 \cos^2 \phi} \\
e &= \sqrt{a^2 - b^2} \\
\epsilon &= \frac{e}{a} \\
a &> b
\end{aligned}
$$

2. Hyperbel

$$
\begin{aligned}
r^2 &= \frac{b^2}{\epsilon^2 \cos^2 \phi - 1} \\
e &= \sqrt{a^2 + b^2} \\
\epsilon &= \frac{e}{a}
\end{aligned}
$$

3. Parabel

$$
r = \frac{a}{1 - \cos \phi}
$$

Eine Überprüfung aller drei Fälle soll dabei im MATLAB-Skript `skegelpol` erfolgen, in dem die Werte für `a`, `b` und `r_0` festgelegt werden. Den ϕ-Vektor können Sie mit dem Befehl `linspace` erzeugen:

```
phi = linspace(phi_min,phi_max,n_points)
```

Die Winkel für Ellipse, Hyperbel und Parabel reichen dabei von ϕ_{min} bis zum Maximalwert ϕ_{max},

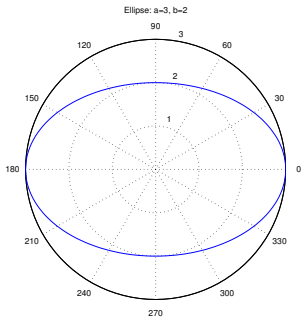
$$
\begin{aligned}
\phi_{min,max} &= 0, \quad 2\pi \\
\phi_{min,max} &= \mp \arccos \left(\left(\frac{1 + b^2/r_0^2}{\epsilon^2} \right)^{1/2} \right) \\
\phi_{min,max} &= \pi \mp \left| \arccos \left(\frac{a}{r_0} - 1 \right) \right|
\end{aligned}
$$

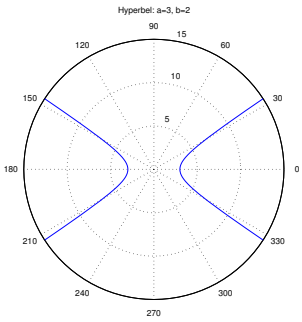
wobei für Hyperbel und Parabel ein größter gewünschter Radius $r_0 > a$ definiert werden muss (warum?). Falls Sie in irgendeinem Fall komplexe Zahlen erhalten, haben Sie mit Sicherheit eine Formel falsch programmiert.

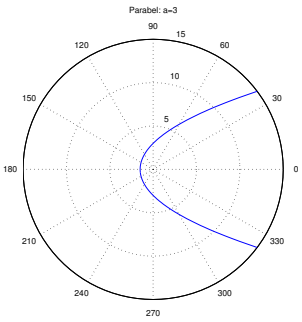
Um zu sehen, ob Sie wirklich eine Ellipse, Hyperbel, bzw. Parabel berechnen, können Sie dabei folgende Plotbefehle ausführen:

```
figure(1) % Ellipse
polar(phi,r);

figure(2) % Hyperbel
polar(phi,r); hold on; polar(pi-phi,r); hold off;

figure(3) % Parabel
polar(phi,r);
```

Überlegen Sie, warum man in einem Fall zwei Plotbefehle braucht, um den gesamten Kegelschnitt zu zeichnen. Der Befehl `hold` ermöglicht, dass man zu einer bereits bestehenden Kurve weitere hinzufügt. Man schaltet diesen Modus mit `hold on` ein und mit `hold off` aus. Überlegen Sie auch, wie groß man den Wert für `n_points` setzen muss, damit man vernünftige Kurven erhält.







12.3 Spiralen, Schwebung, Gaußverteilung

Ziel: Umgang mit Funktionen. Setzen von Defaultwerten für Inputparameter. Erstellen einfacher Plots.

Vorraussetzung: Grundlagen von Funktionen in MATLAB.

Erstellen Sie ein Skript `sfunk.m`, das alle folgenden Funktionen testet und einfache Kurven der Funktionen erstellt. Erklärungen zu den Plotbefehlen finden Sie in dem Abschnitt 11.2.1 bzw. in der MATLAB-Hilfe für `plot`.

Defaultwerte kann man setzen, indem man die Anzahl der Input-Variablen überprüft. Dafür steht nach dem Aufruf der Funktion die Variable `nargin` zur Verfügung. Darüberhinaus bietet sich auch die Funktion `isempty` an. Erklärungen dafür finden Sie in dem Abschnitt 7.1.6.

12.3.1 Spiralen

Schreiben Sie eine MATLAB-Funktion `spirale1`, die mit dem Aufruf

```
[x,y] = spirale1(t,nu,a1,a2)
```

folgende Funktion berechnet

$$x(t) = a_1 e^{-a_2 t} \cos(2\pi\nu t)$$
$$y(t) = a_1 e^{-a_2 t} \sin(2\pi\nu t) \ ,$$

wobei t ein Zeitvektor ist. Der Defaultwert für die Frequenz ν soll 5 sein und für a_1 und a_2 sollen als Defaultwert 1 gesetzt werden. Erzeugen Sie im zugehörigen Testskript `sfunk` einen vernünftigen Zeitvektor und plotten Sie y als Funktion von x, bzw. x und y als Funktion von t.

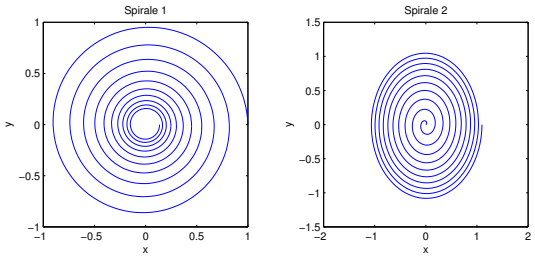
Schreiben Sie eine MATLAB-Funktion `spirale2`, die mit dem Aufruf

```
[x,y] = spirale2(t,nu,a1,a2)
```

folgende Funktion berechnet

$$x(t) = a_1 \log(1 + a_2 t) \cos(2\pi\nu t)$$
$$y(t) = a_1 \log(1 + a_2 t) \sin(2\pi\nu t) \ ,$$

wobei t ein Zeitvektor ist. Der Defaultwert für die Frequenz ν soll 5 sein und für a_1 und a_2 sollen als Defaultwert 1 gesetzt werden. Erzeugen Sie im Skript einen vernünftigen Zeitvektor und plotten Sie y als Funktion von x, bzw. x und y als Funktion von t.



12.3.2 Schwebung

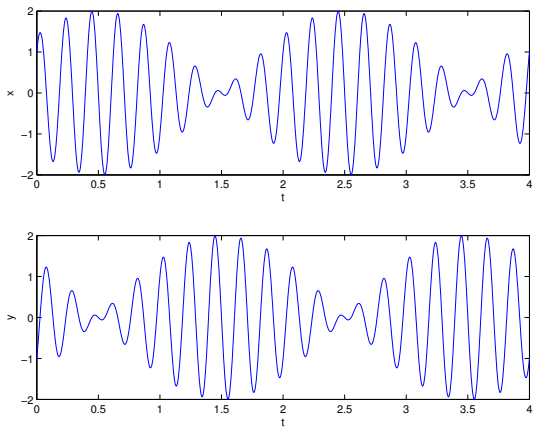
Schreiben Sie eine MATLAB-Funktion `schwebung`, die mit dem Aufruf

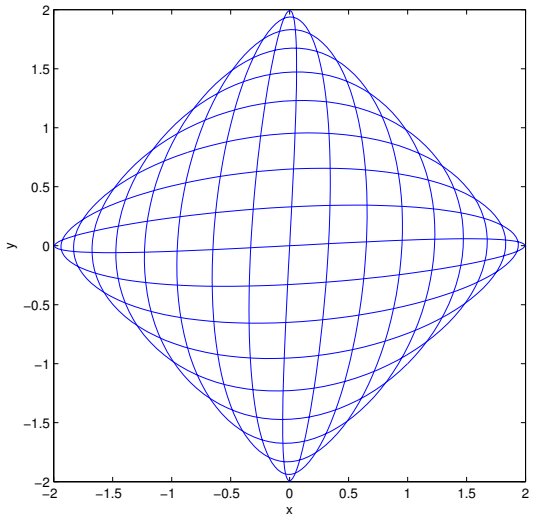
```
[x,y] = schwebung(t,nu1,nu2,a1)
```

folgende Funktion berechnet

$$x(t) = a_1 \left(\sin(2\pi\nu_1 t) + \cos(2\pi\nu_2 t)\right)$$
$$y(t) = a_1 \left(\sin(2\pi\nu_1 t) - \cos(2\pi\nu_2 t)\right) \ ,$$

wobei t ein Zeitvektor ist. Die Defaultwerte für die Frequenz ν_1 und ν_2 sollen 5 und 4.5 sein. Der Defaultwert für für a_1 soll 1 sein. Erzeugen Sie im Skript einen vernünftigen Zeitvektor und plotten Sie x und y als Funktion von t, bzw. y als Funktion von x. Der Zeitvektor sollte so gewählt werden, dass man das Phänomen der Schwebung erkennen kann. Die Darstellung y als Funktion von x nennt man eine Lissajous-Figur. Die Kurve schließt sich zum ersten Mal, wenn $t_0 = 0$ und $t_{end} = 1/\left|\nu_1 - \nu_2\right|$ ist.





12.3.3 Gaußverteilung

In der Wahrscheinlichkeitsrechnung und auch in der Physik hat die Normalverteilung bzw. Gaußverteilung eine große Bedeutung. Sie ist definiert durch

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{(x-x_0)^2}{2\sigma^2}\right\} ,$$

wobei x_0 und σ die Parameter der Verteilung sind. In x_0 liegt sowohl das Maximum als auch das Symmetriezentrum, und σ ist der Abstand von diesem Zentrum zu den Wendepunkten. Wie für jede Wahrscheinlichkeitsverteilung gilt

$$\int_{-\infty}^{\infty} g(x)dx = 1 .$$

In der Physik verwendet man häufig auch eine Summation über mehrere Gaußfunktionen mit jeweils unterschiedlichen Parametern. Dies hat den Sinn, dass man gleichzeitig mehrere Maxima darstellen kann. Allgemein kann man die Funktion dann definieren als

$$g(x) = \sum_{k=1}^{n} \frac{1}{n\sqrt{2\pi}\sigma_k} \exp\left\{-\frac{(x-x_{0k})^2}{2\sigma_k^2}\right\} ,$$

wobei x_{0k} und σ_k die gleiche Bedeutung wie vorher haben. Die Summation erfolgt über alle n Werte der Parameter x_{0k} und σ_k.

Schreiben Sie eine MATLAB-Funktion `gauss1d`, die mit folgendem Aufruf

```
g(x) = gauss1d(x,x0,sigma)
```

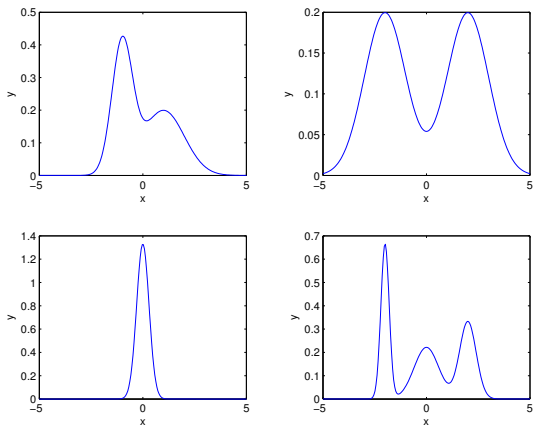
die Gaußverteilung als Funktion des Vektors x berechnet. Setzen Sie die Defaultwerte $x_0 = [-1, 1]$ und $\sigma = [0.5, 1]$.

- Initialisieren Sie g als Feld mit lauter Nullen in der Größe von x.
- Um sicherzustellen, dass die Vektoren für x_0 und σ gleich lang sind, kann man deren Länge (`length`) vergleichen und bei unterschiedlicher Länge einen Fehler (`error`) melden.
- Bilden Sie die Summe über alle Werte ($k = 1 \ldots n$) der Parameter mit Hilfe einer einzigen `for`-Schleife. Vergessen Sie dabei nicht, dass x ein Array sein kann.

  Hinweis: Eine `for`-Schleife hilft bei der Summation über die einzelnen Beiträge zur Summe 12.3.3. Dabei ist es praktisch, wenn man noch vor der Schleife ein Feld erzeugt, das gleich groß ist, wie die Inputvariable `x`, das aber lauter Nullen enthält. Der sinnvollste Befehl dafür ist:
  ```
  g = zeros(size(x))
  ```
  Dann kann man in der Schleife

```
g = g + ...
```
schreiben und bei jedem Durchlauf werden die Werte für einen Peak addiert. Nach dem Ende der Schleife muss man durch die Anzahl der Peaks `n`, also durch die Länge des Vektors `x0` (oder `sigma`) dividieren. Damit ist dann die Normierung auch bei mehreren Peaks sichergestellt.

- Probieren Sie die Funktion mit Hilfe des Skripts `sfunk` aus und machen Sie einen Plot
  ```
  plot(x,g)
  ```
  Wählen Sie den x-Vektor in einem vernünftigen Bereich, damit die Funktion $g(x)$ sinnvoll dargestellt wird (`linspace`). Beschriften Sie den Plot mit Achsenbeschriftung und Titel.



12.4 Felder

Ziel: Ziel der Übung ist die Verwendung grundlegender Befehle zum Erstellen und Verändern von Arrays und das Üben der Doppelpunkt Notation für den Zugriff auf Arrays.

Anzufertigen ist eine MATLAB-Funktion `arrtest`, die die Lösungen für alle Unterbeispiele enthalten soll. Diese werden beim Aufruf durch die Übergabe einer Stringvariablen `sw` ausgewählt.

Voraussetzung: Die Eigenschaften von Arrays und von Funktionen, die Arrays erzeugen bzw. verändern werden im Kapitel 3 erklärt. Die Basiseigenschaften werden dort in Tabelle3.1 zusammengefasst, während Befehle zum Erzeugen von Matrizen in den Tabellen 3.3 und 3.4 dargestellt sind. Befehle zum Verändern von Matrizen findet man im Abschnitt 3.5.

Der Zugriff auf Teile von Arrays wird wir im Abschnitt 3.6 erläutert, wobei man im Abschnitt 3.6.2 ausführliche Beispiele findet.

Basiswissen über dieses Kapitel des Skriptums ist unbedingte Voraussetzung für die Übung. Probieren Sie daher u.a. die Befehle `zeros`, `ones`, `eye`, `diag` aus, und versuchen Sie mit der `Doppelpunkt` Notation auf Teile der Matrizen zuzugreifen.

Der Aufruf der Funktion sollte mit `r = arrtest(sw,m,n,o)` erfolgen, wobei `sw` als Schalter für die einzelnen Subaufgaben dient, `m`, `n` und `o` erfüllen je nach Unterbeispiel unterschiedliche Aufgaben und haben auch jeweils andere Defaultwerte.

Um das Setzen der Defaultwerte in den Unterbeispielen zu erleichtern, sollen am Beginn des Programms nicht übergebene Parameter (`nargin`) auf den Wert `[]` (leeres Array) gesetzt werden. Dann kann man innerhalb der `switch`-Kosntruktion mit Hilfe des Befehls `isempty` die eigentlichen Defaultwerte setzen.

1. Erzeugen Sie z.B. ein $m \times n$-Array (Default 3×4) mit lauter gleichen Zahlen (Default `o=3`). Für den Schalterwert sollte `'const'` verwendet werden.

2. Mit dem Schalterwert `'chess'` soll eine schachbrettartige Anordnung von Einsern und Nullen als Symbole für schwarz und weiß erzeugt werden. Die Größe der $m \times n$-Matrix sollte 8×5 sein und sie sollte links oben mit einer Eins beginnen.

   Denken Sie dabei an die Verwendung der Befehle `eye` und `repmat`. Praktisch ist auch der Befehl `ceil` und das Löschen von ganzen Zeilen oder Spalten mit Hilfe der Zuweisung eines leeren Arrays `[]`.

3. Studieren Sie den Befehl `diag` und erzeugen Sie dann für den Schalterwert `'diag'` mit Hilfe der Befehls `diag` folgende Matrix mit `m=8` Zeilen und `n=7` Spalten

$$r = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 2 & 0 & 0 \end{bmatrix} .$$

Die Einheitszelle sollte dabei allgemein von `1` bis `o` reichen,

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 2 & 0 & \dots & 0 \\ 0 & 0 & 3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & o \end{bmatrix} ,$$

wobei als Defaultwert für `o=3` verwendet werden soll.

4. Erstellen Sie für den Schalterwert `'diagplus'` eine $m \times m$-Matrix mit einem Defaultwert `m = 5`.

$$r = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 3 & 1 & 1 \\ 1 & 1 & 1 & 4 & 1 \\ 1 & 1 & 1 & 1 & 5 \end{bmatrix} .$$

Die Werte in der Diagonale sollten immer von `1` bis `m` laufen. Die restlichen Werte der Matrix sollten den Wert von `n` haben, wobei der Defaultwert `1` ist.

5. Erstellen Sie für den Schalterwert `'randsym'` eine $m \times m$-Matrix (Default 5) mit Zufallszahlen im Intervall $[0,\ 1]$, die zusätzlich symmetrisch ist. Als symmetrische Matrix bezeichnet man eine Matrix für die $a_{ij} = a_{ji}$. Verwenden Sie dazu die Befehle `rand`, `triu` und `transpose`.

   Die Lösung dieses Problems ist ein wenig schwieriger und erfolgt in zwei Stufen. Die erste Stufe erzeugt mit `rand` und `triu` eine Matrix, bei der nur der Teil über und inklusive der Hauptdiagonale besetzt ist. In einer zweiten Stufe wird dann die zweite Form des Befehls `triu(A,k)` verwendet, das Ergebnis transponiert und zum ersten Teil addiert.

   Wie immer gibt es natürlich auch hier andere Lösungen.

6. Erzeugen Sie für die Schalterstellung `'magic'` mit dem Befehl `magic` ein $m \times m$ magisches Quadrat `M`, wobei $m = 5$ ist. Beweisen Sie durch Anwendung der

Befehle `sum`, `diag` und `fliplr`, dass alle Summen über die Reihen, die Spalten und die beiden Diagonalen gleich groß sind.

Verwenden Sie dazu die verschiedenen Formen von `sum(M,k)` und die Befehle `diag` und `fliplr`.

Verbinden Sie die vier Summenvektoren zu einem Zeilenvektor und bedenken Sie dabei, dass die Ausrichtung der resultierenden Vektoren von `sum(M,1)` und `sum(M,2)` unterschiedlich ist.

Das Zusammenhängen von Arrays erfolgt mit dem Befehl `cat` bzw. mit seiner Kurzform `[A,B]` oder `[A;B]`.

Geben Sie den Vektor mit allen $2(n+1)$-Summenwerten als Resultat `r` zurück.

7. Erzeugen Sie für die Schalterstellung `'numberst'` folgende $m \times n$-Matrix (Default 4×6)

$$r = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 19 & \dots & \dots & \dots & mn-1 & mn \end{bmatrix},$$

   Erzeugen Sie dabei einen Vektor und verwenden Sie dann die Befehle `reshape` und `transpose`. Die Erzeugung des Vektors soll mit der `Doppelpunkt` Notation erfolgen.

8. Erzeugen Sie mit `'star'` folgende Matrix der Größe $2m \times 2m$, wobei als Defaultwert `m=3` zu verwenden ist,

$$r = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 2 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 2 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

9. Erzeugen Sie mit der `'rstar'` folgende Matrix der Größe $2m-1 \times 2m-1$, wobei als Defaultwert `m=3` zu verwenden ist,

$$r = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

10. Erzeugen Sie mit der Schalterstellung `'setnan'` folgende Matrix der Größe $m \times n$ (Defaultwert 5×6),

$$r = \begin{bmatrix} nan & 1 & 1 & 1 & 1 & nan \\ 1 & nan & nan & nan & nan & 1 \\ 1 & nan & nan & nan & nan & 1 \\ 1 & nan & nan & nan & nan & 1 \\ nan & 1 & 1 & 1 & 1 & nan \end{bmatrix} .$$

    Beginnen Sie mit einer Matrix mit lauter Einsen und setzen Sie dann mit Hilfe der `Doppelpunkt`-Notation in zwei Befehlszeilen einmal die Eckpunkte und dann den Mittelbereich auf `nan`. Dabei ist immer das Keyword `end` zu verwenden. Der Wert `nan` symbolisiert in MATLAB "Not a Number". Mit `nan` kann man immer weiterrechnen, wobei die Rechenregel ganz einfach ist. Jede Operation mit `nan` gibt wieder `nan`.

Hier noch Anregungen für Studenten mit etwas fortgeschrittenem Interesse:

1. Man kann ganz einfach einen Schalterwert `'test'` erstellen, wo die Funktion `arrtest` sich immmer wieder für alle Fälle aufruft. Dafür erzeugt man eine Zelle
   `testcases = {'const','chess','diag', .... };`
   und ruft in einer `for`-Schleife von `1` bis `length(testcases)`
   `arrtest(testcases{k},m,n,o)`
   und gibt z.B. in diesem Fall die Resultate mit `disp` aus.

2. Auf ähnliche Weise kann man auch einen Fall `'in'` erstellen, der den Benutzer nach den Werten für `sw`, `m`, `n` und `o` fragt (`input`) und dann damit wiederum `arrtest` aufruft.

In solchen Fällen kann auch eine Ausgabe am Schirm aus einer Funktion, bzw. die zuätzliche Eingabe von Werten mit `input` interessant sein. Viel Spaß beim Ausprobieren.

12.5 Logische Indizierung

Ziel: Ziel der Übung ist die Verwendung der logischen Indizierung in MATLAB, wie sie im Abschnitt 3.6.1 besprochen wird.

Anzufertigen sind die MATLAB-Funktionen `quadgl`, `quadgleval`, `rangetest` und ein MATLAB-Skript `ueblogical`. Die Funktionen enthalten die Lösungen für alle Unterbeispiele und das Skript ist eine einfache Testroutine für die Funktionen.

Voraussetzung: Der Zugriff auf Teile von Arrays wird wir im Abschnitt 3.6 erläutert, wobei man im Abschnitt 3.6.2 ausführliche Beispiele findet. Teile davon waren Bestandteil der letzten Übung. Für die heutige Übung benötigt man zusätzlich Kenntnisse über die logische Indizierung aus dem Abschnitt 3.6.1. Auch dazu gibt es Beispiele in 3.6.2.3.

Die vorbereiteten Programme für die Übung können mit dem Befehl `!uebungsdaten` geladen werden. Die Übungsnummer ist 5. Die Programme, die zur Verfügung stehen sind:

- `issamesize` zum Größenvergleich von Arrays
- `showmat` zur graphischen Darstellung von Arrays
- `showmata` zur graphischen Darstellung von Arrays in einem Bild

Benötigt wird auch Basiswissen über `Logische Operatoren`.

1. Eine quadratische Gleichung der Form

$$ax^2 + bx + c = 0 \qquad a, b, c \in \mathbb{R} \tag{12.1}$$

hat folgende Lösungen

$$\begin{aligned} x_{1,2} &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} && a \neq 0 \\ x_1 &= -c/b && a = 0 \wedge b \neq 0 \\ x_1 &= 0 \quad \text{triviale Lösung} && a = 0 \wedge b = 0 \wedge c = 0 \end{aligned} \tag{12.2}$$

Der Ausdruck

$$D = b^2 - 4ac \tag{12.3}$$

wird dabei als Diskriminante bezeichnet. Im Falle $a \neq 0$ entscheidet sein Wert, ob es zwei reelle Lösungen, eine reelle Doppellösung, oder zwei konjugiert komplexe Lösungen gibt. Für den Fall $a = 0 \wedge b = 0 \wedge c \neq 0$ gibt es keine Lösung.

Zu erstellen ist nun ein Unterprogramm
`[x1,x2]=quadgl(a,b,c)`

welches 12.1 löst, wobei die Koeffizienten a, b und c beliebige Felder mit reellen Zahlen sein können. D.h. die Gleichung soll simultan für mehrere Werte der Koeffizienten gelöst werden, wobei die Ergebnisse in den Feldern $x1$ und $x2$ gespeichert werden sollen. Zu verwenden sind dabei die Lösungen aus 12.2, wobei dort wo keine Lösungen existieren, der Wert `NaN` (entspricht: Not A Number) verwendet werden soll. Dort wo beliebige Lösungen erlaubt sind, soll für x_1 die triviale Lösung 0 verwendet werden.

Dazu notwendig ist Folgendes:

- Zu einem guten Programm gehört die Überprüfung der Eingabeparameter:

  Die Anzahl der Eingabeparameter (7.1.6) sollte hier immer drei sein (keine Defaultwerte), ansonsten sollte eine Fehlermitteilung geschrieben werden (`error`).

  Außerdem sollte mit Hilfe der Befehle `issamesize` (wurde bereitgestellt), `isnumeric` und `isreal` überprüft werden, ob die Koeffizientenfelder gleich groß sind, ob sie Zahlen sind und, ob sie reelle Zahlen enthalten. Hilfe zu den Befehlen bekommt man z.B. mit `help issamesize` oder auch im Abschnitt 7.1.4. Falls die Bedingungen nicht erfüllt sind, sollte wieder eine Fehlermitteilung geschrieben werden.

- Die Lösung des Problems ist mit Hilfe der logischen Indizierung relativ einfach. Die beste Strategie ist für `x1`, `x2` und `D` Felder der Größe von `a` anzulegen und mit dem Wert `nan` zu belegen.

  Danach kann man drei logische Felder entsprechend der drei logischen Bedingungen in 12.2 erzeugen und diese dann auf beiden Seiten der Zuweisungen verwenden. Damit ist sichergestellt, dass die jeweiligen Berechnungen nur durchgeführt werden, wenn die entsprechende Bedingung erfüllt ist.

  Zwei Dinge sollte man sich überlegen: Warum ist es besser die Diskriminante nur einmal zu berechnen und dann in der Formel für $x_{1,2}$ zu verwenden? Warum muss man in MATLAB keine Unterscheidung treffen, ob der Wert $b^2 - 4ac$ größer oder kleiner Null ist?

2. Schreiben Sie eine kleine MATLAB-Funktion
   `[r1,r2] = quadgleval(a,b,c,x1,x2)`
   welche die rechte Seite von 12.1 ausrechnet. Überlegen Sie wieder welche Operatoren Sie verwenden müssen. Mit `nan` kann man normal rechnen, wobei jede arithmetische Operation mit `nan` wieder `nan` ergibt.

3. Zum Testen der Funktionen benötigen Sie ein MATLAB-Skript
   `ueblogical`
   wobei mit `input` die Variablen `n` (Größe der Matrizen, Default `15`), `rmin` (Minimale Zufallszahl, Default `2`), `rmax` (Maximale Zufallszahl, Default `5`), `delay`

(Verzögerungszeit, Default 2), und `graph` (Graphikoutput, Default `'y'`) zugewiesen werden. Für das Setzen der Defaultwerte empfiehlt sich die Verwendung von `isempty`.

Erzeugen Sie nun die drei $n \times n$-Felder `a`, `b` und `c` mit gleichverteilten Zufallszahlen zwischen r_{min} und r_{max}. Der Befehl `rand(n)` liefert dabei gleichverteilte Zufallszahlen zwischen 0 und 1.

Setzen Sie danach die mittlere Spalte von `a` und die mittlere Zeile von `b` auf den Wert Null.

Rufen Sie `quadgl` und danach mit den Resultaten `quadgleval` auf. Die Resultate können Sie graphisch darstellen:

```
showmata(real(x1),real(x2)) % Realteile
pause(delay)                % Verzögerung
showmata(imag(x1),imag(x2)) % Imaginärteile
pause(delay)
showmata(abs(r1),abs(r2))   % Absolutwerte der rechten Seite
pause(delay)
```

Die Befehle `real`, `imag` und `abs` berechnen den Realteil, den Imaginärteil und den Absolutwert.

Sie sollten nun die Resultate in einer Farbcodierung sehen, wobei weisse Bereiche `nan` symbolisieren. Wegen numerischer Ungenauigkeiten ergeben sich für die Werte auf der rechten Seite nicht nur Null, sondern auch kleine Werte im Bereich 10^{-15}.

Zählen Sie nun für die beiden Felder `ar1=abs(r1)` und `ar2` die Anzahl der Werte mit exakter Null, mit `nan` (`isnan`). Dabei summiert man immer über alle Werte (`Doppelpunkt`) in den entsprechenden logischen Arrays.

Außerdem soll man in einer `for`-Schleife jeweils alle Werte $10^k \leq |r_{1,2}| < 10^{k+1}$ zählen, wobei k von -20 bis -10 laufen soll.

Geben Sie all diese Summen formatiert aus und überlegen Sie, ob sie stimmen können. Es sollten eine gewisse Anzahl von Nullen, von `nan` und sonst eher kleine Werte vorkommen. Kommen zu große Werte vor, ist sicher etwas falsch.

4. Schreiben Sie eine Funktion
   `[m,count] = rangetest(m,b,e)`
   wobei `m` eine beliebige Matrix, `b` ein Vektor der Länge 2 und `e` ein Vektor der Länge 3 ist.

   Für die Output-Matrix `m` soll Folgendes gelten

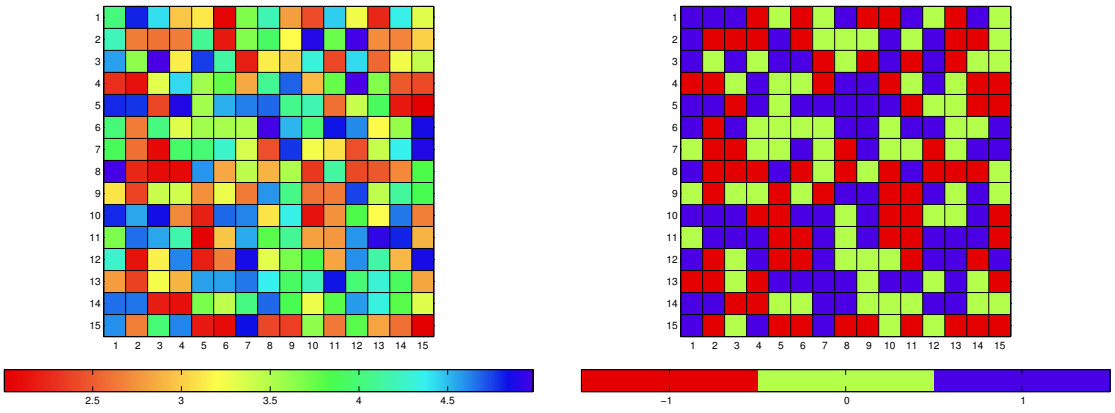
$$m_{i,j} = e_1 \qquad m_{i,j} < b_1 \tag{12.4}$$
$$m_{i,j} = e_2 \qquad b_1 \leq m_{i,j} \leq b_2 \tag{12.5}$$
$$m_{i,j} = e_3 \qquad m_{i,j} > b_2\ . \tag{12.6}$$

Der Vektor `count` (Länge 3) soll die Anzahl der jeweiligen Ersetzungen enthalten.

Sie können auf alle Überprüfungen und auf das Setzen von Defaultwerten verzichten, wichtig ist nur die Funktionalität der MATLAB-Funktion. Die gesamte Aufgabe soll wieder ohne `if` und `for` unter Verwendung der logischen Indizierung funktionieren.

5. Ergänzen Sie das MATLAB-Skript
   `ueblogical`
   um folgende Teile. Erzeugen Sie eine weitere $n \times n$-Matrix `m` mit gleichverteilten Zufallszahlen zwischen r_{min} und r_{max}. Erzeugen Sie einen Vektor `b` mit zwei Einträgen, der den Bereich zwischen r_{min} und r_{max} in drei gleiche Teile teilt. Als Vektor `e` verwenden Sie `[-1,0,1]`.

   Stellen Sie die Matrix vor und nach der Ersetzung graphisch dar. Die Bilder sollten dabei ungefähr so aussehen.



   Geben Sie die Werte von `count` formatiert aus und überprüfen Sie, ob die Gesamtzahl der Ersetzungen gleich der Anzahl der Elemente in `m` ist.

12.6 Berechnung von Reihen und Konvergenzverhalten

Ziel: Ziel der Übung ist die Berechnung von endlichen und unendlichen Reihen in MATLAB.

Anzufertigen sind die MATLAB-Funktionen `reihe` und `eulernum`. Die Funktion `reihe` löst die Reihenentwicklung einiger Funktionen, und die Funktion `eulernum` berechnet die Eulerschen Zahlen, die für eine spezielle Reihe notwendig sind.

Darüber hinaus benötigt man ein MATLAB-Skript `uebreihe`, welches die Funktionen testet und die Ergebnisse in einfachen Plots darstellt. Exemplarische Plots sind nach der Beschreibung der Übung eingefügt.

Voraussetzung: Die Voraussetzung für die Übung sind Kenntnisse über die grundlegenden Befehle in MATLAB und über die Erstellung von Funktionen und deren Verwendung. Außerdem braucht man einfache Plotbefehle, wie `plot`, `legend`, `xlabel`, `ylabel`, `title`.

Als einfaches Beispiel für eine unendliche Reihe wird hier die Reihendarstellung der Funktion $\sin(x)$ verwendet. Diese ist gegeben durch

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \tag{12.7}$$

$$= \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \,, \tag{12.8}$$

welche für alle $|x| < \infty$ konvergiert.

In allen numerischen Programmiersprachen kann man solche unendlichen Reihen natürlich nur näherungsweise auswerten, da die Summation von unendlich vielen Termen unendlich lange dauern würde. Daher gibt man sich mit der endlichen Teilsumme

$$\sin(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \,, \tag{12.9}$$

zufrieden. Abhängig von der gewählten Reihe und von den Werten für x, konvergieren sie langsamer, schneller oder eben gar nicht zum gewünschten Wert.

Summen dieser Art kann man in MATLAB sehr gut ohne die Verwendung von `for`-Schleifen programmieren. Nützlich sind dabei folgende Informationen:

- Die Berechnung von $k!$ (Fakultät) kann mit Hilfe der Γ-Funktion durchgeführt werden

  $$\Gamma(n+1) = n! \qquad (n = 0, 1, 2, \dots) \,, \tag{12.10}$$

  wobei dafür in MATLAB die Funktion `gamma(k)` zur Verfügung steht. Diese funktioniert natürlich wieder für Arrays von k-Werten. Für große $k > 171$ liefert

$\Gamma(k)$ den Wert unendlich, da das Ergebnis jenseits der Darstellungsgrenze für den Datentyp `double` liegt.

- Äußerst praktisch für die Berechnung solcher Reihen ist der Befehl `meshgrid`. Damit kann man aus einem Vektor `x` mit allen x-Werten und einem Vektor `k` mit allen k-Werten zwei gleich große Matrizen

```
x = [-2:2]; k = [0:5];
[xx,kk] = meshgrid(x,k);
```

erzeugen

$$xx = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix} \qquad kk = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{bmatrix} \quad . \tag{12.11}$$

Mit dem Befehl `A=xx.^(2*kk+1)` kann man nun z.B. x^{2k+1} für alle Werte von x und k gleichzeitig ausrechnen. Die Summe über alle k-Werte kann man dann mit Hilfe des `sum(A,1)`-Befehles erhalten.

- Sehr einfach kann die gesamte Reihenauswertung auch mit Hilfe der Matrizenmultiplikation durchgeführt werden. Aus der linearen Algebra ist bekannt, dass für einen $(1 \times n)$-Vektor $\mathbf{f}$ (Zeilenvektor) und eine $(n \times m)$-Matrix $\mathbf{A}$ gilt

$$\{\mathbf{fA}\}_l = \sum_k f_k A_{kl} \; . \tag{12.12}$$

Erzeugt man nun für alle Werte von k den Vektor $\mathbf{f} = (-1)^k/(2k+1)!$, kann man mit $\mathbf{fA}$ (in MATLAB `f*A`) für alle Werte von x gleichzeitig den jeweiligen Wert der Teilsumme erhalten.

Der Aufruf der MATLAB-Funktion `reihe` soll folgendermaßen aussehen

```
[r,u,a] = reihe(typ,x,n)
```

wobei `typ` eine Schaltvariable (`switch`), `x` ein Array von x-Werten und `n` der Maximalwert für die k-Werte ist. Ausgegeben werden sollen die Teilsummen `r`, die Werte der Funktion `u` (hier z.B. $\sin(x)$) und eine eventuelle analytische Lösung `a` für die endlichen Teilsummen der Reihe.

Alle Outputwerte müssen die gleiche Größe wie x haben. Dazu merkt man sich am Anfang des Programms die Größe von x (`size`), macht aus dem Array x einen Vektor (`colon`). Am Ende des Programms kann man dann mit `reshape` sicherstellen, dass die richtige Größe zurückgegeben wird.

1. Programmieren Sie entsprechend der Angabe die Reihe für die Sinusfunktion (`typ='sin'`). Für a geben Sie lauter `NaN` zurück.

   Stellen Sie mit Hilfe des Skripts `uebreihe` einen Vergleich zwischen der Teilsumme der Reihe und der Funktion für verschiedene Werte von n dar.

2. Die Teilsumme der Geometrischen Reihe ist gegeben durch

$$r = \sum_{k=1}^{n} x^{k-1} = \begin{cases} (1-x^n)/(1-x) & : \quad x \neq 1 \\ n & : \quad x = 1 \end{cases} . \tag{12.13}$$

   Die unendliche geometrische Reihe konvergiert für alle $|x| < 1$

$$u = \sum_{k=1}^{\infty} x^{k-1} = \begin{cases} 1/(1-x) & : \quad |x| < 1 \\ \infty & : \quad x \geq 1 \\ NaN & : \quad x \leq -1 \end{cases} . \tag{12.14}$$

   Berechnen Sie für den `typ 'geom'` den Wert `r` der geometrischen Reihe 12.13, in `a` den analytischen Wert aus 12.13 und in `u` den Wert der unendlichen Reihe 12.14. Die Fallunterscheidungen können Sie mit Hilfe der logischen Indizierung durchführen.

   Probieren Sie in `uebreihe` das Konvergenzverhalten aus und stellen Sie die Ergebnisse für `r`, `a` und `u` gemeinsam als Funktion von `x` dar.

3. Eine der Hyperbelfunktionen, $\mathrm{sech}(x) = 1/\cosh(x)$, kann durch folgende asymptotische Reihe dargestellt werden

$$\mathrm{sech}(x) = 2\sum_{k=0}^{\infty}(-1)^k e^{-(2k+1)|x|} , \tag{12.15}$$

   die nun für große Werte von x rasch konvergiert und um Null Schwierigkeiten macht . Als Typ verwenden Sie `'sechasymp'`.

   Geben Sie in `r` die Werte der Teilsumme, in `u` die Werte der Funktion `sech` und in `a` einen Vektor mit `NaN` zurück.

4. Die Taylorreihenentwicklung für sech ist gegeben durch

$$\mathrm{sech}(x) = 1 - \frac{x^2}{2} + \frac{5x^4}{24} - \frac{61x^6}{720} + \ldots \tag{12.16}$$

$$= \sum_{k=0}^{\infty} \frac{\mathrm{E}_k}{(2k)!} x^{2k} , \tag{12.17}$$

   welche für alle $|x| < \pi/2$ konvergiert (`typ='sech'`). Außerhalb des Konvergenzintervalls geben Sie `NaN` zurück.

Die Eulerschen Zahlen E_k sind durch folgende Reihe definiert

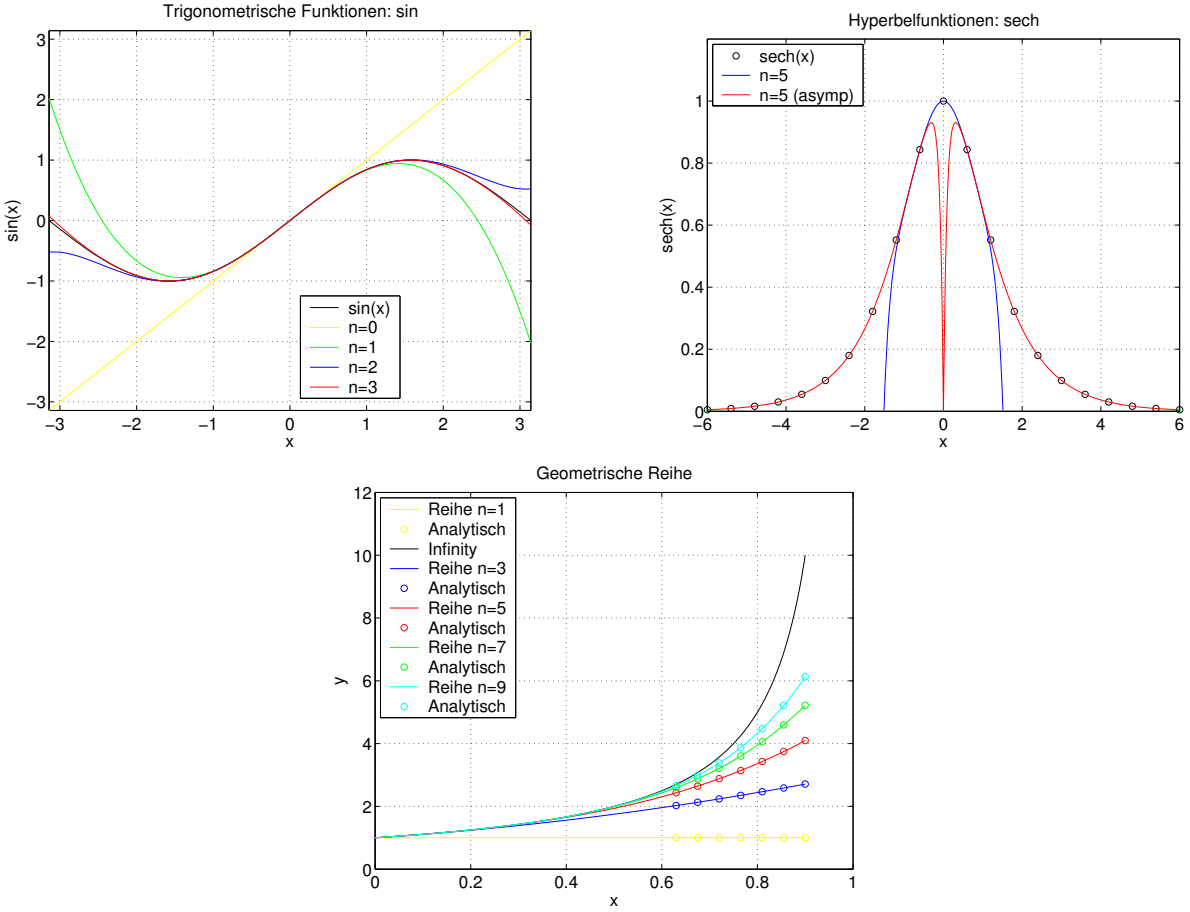
$$\mathrm{E}_k = \frac{(-1)^k 2^{2k+2}(2k)!}{\pi^{2k+1}} \sum_{l=1}^{\infty} \frac{(-1)^{l-1}}{(2l-1)^{2k+1}} \ , \tag{12.18}$$

die im Programm

```
e = eulernum(k,lmax)
```

berechnet werden sollen. Hier wird nun über l summiert und der **k**-Vektor spielt die Rolle, die vorher x gespielt hat, d.h. für jeden Wert von k bekommt man einen Wert von E_k. Als Defaultwert für `lmax` kann man 10 verwenden. Da alle Eulerzahlen ganzzahlig sind, empfiehlt es sich, am Ende der Funktion den Befehl `round` zu verwenden. Damit werden leichte numerische Ungenauigkeiten ausgeglichen, und man muss `lmax` nicht größer wählen. Zur Kontrolle hier die Werte von E_0 bis E_6, die sich als 1, -1, 5, -61, 1385, -50521, 2702765 ergeben. Diese sollten Sie beim Aufruf von `eulernum([0:6])` erhalten.

Vergleichen Sie im Skript `uebreihe` die Ergebnisse der beiden Reihenentwicklungen und stellen Sie sie graphisch dar.



12.7 Kurvendiskussion

Ziel: Der Umgang mit Parameterlisten und Funtionen soll geübt werden. Bei dieser Übung liegt der Schwerpunkt auf der Verwendung von Polynomen.

Vorraussetzung: Grafisches Darstellen von Funktionen und die Kenntnis der Funktionen zum Umgang mit Polynomen sind von Nöten (`polyval`, `polyint`, `polyder`, `roots`). Diese und andere Befehle für die Bearbeitung von Polynomen werden im Kapitel 8 besprochen.

Einige Programme für das Plotten von Funktionen können mit dem Befehl `uebungsdaten` heruntergeladen werden. Sie zeigen in verschiedenen Stufen, wie man Kurven in MATLAB darstellen kann (`htplot2d`, `htplot2da`, `htplot2s`). An Hand von `htpolyplot` bekommt man ein paar Hinweise für die Erstellung des Plots für diese Übung.

Außerdem bewährt es sich bei der numerischen Integration von Funktionen (`quadl`) sogenannte `inline`-Funktionen zu verwenden. Dies sind Funktionen die im Skript definiert werden können und daher kein eigenenes MATLAB-File brauchen. Eine Besprechung dieses Funktionentyps finden Sie im Unterkapitel 7.1.7. Die Verwendung solcher Funktionen z.B. beim numerischen Integrieren ist im Unterkapitel 7.1.8 beschrieben.

Abgabe: Abzugeben sind die Funktionen `polyadd.m`, `polytang.m`, `kreis.m` und das Skript `polydiskussion.m`.

1. **Addition von Polynomen:**
   Schreiben Sie eine Funktion `pout = polyadd(p1,p2)`, die zwei Polynome mit Koeffizientenvektoren `p1` und `p2` addiert und den resultierenden Koeffizientenvektor `pout` zurückgibt. Warum kann man Polynome eigentlich nicht einfach mit dem Operator `+` addieren?

2. **Tangente an ein Polynom:**
   Schreiben Sie eine Funktion `pout = polytang(p,x0)`, die die Tangente an das Polynom mit Koeffizientenvektor `p` im Punkt `x0` berechnet und als Koeffizientenvektor `pout` zurückgibt (Tangente = Gerade = Polynom ersten Grades). Zur Erinnerung: Die Tangentengleichung für eine Tangente an das Polynom $y(x)$ an der Stelle x_0 lautet

   $$y_t(x) = y'_0\, x + (y_0 - y'_0\, x_0) \quad ,$$

   wobei $y_0 = y(x_0)$ und $y'_0 = dy/dx|_{x=x_0}$ sind.

3. **Kurvendiskussion:**
   Gegeben sind zwei Polynome

   $$y_1 = 0.1\, x^3 - 0.7\, x^2 + 1.4\, x - 0.8$$
   $$y_2 = -0.2\, x^2 + 1.2\, x - 1.35 \quad .$$

Legen Sie ein Script `polydiskussion.m` an, das folgende Punkte ausführt:

(a) Berechnen Sie die Schnittpunkte der beiden Polynome. (Hinweis: $y_1 = y_2 \iff y_1 - y_2 = 0$).

(b) Zeichnen Sie die Polynome im relevanten Bereich (Schnittpunkte deutlich innerhalb der Zeichnung). Markieren Sie in der Abbildung die Schnittpunkte.

(c) Berechnen Sie Nullstellen, Maxima, Minima, Wendepunkte der Polynome und zeichnen Sie diese ein. Nullstellen findet man mit `roots`. Extrema findet man bei den Nullstellen der ersten Ableitung. Um ein Maximum handelt es sich, wenn gleichzeitig die zweite Ableitung an dieser Stelle kleiner Null ist. Wendepunkte liegen an der Stelle, wo die zweite Ableitung der Funktion Null ist.

(d) Legen Sie eine Tangente durch den Wendepunkt des ersten Polynoms. Verwenden Sie dafür Ihre Funktion `polytang` und plotten Sie eine kurze Gerade im Bereich um den Wendepunkt.

(e) Berechnen Sie die Fläche zwischen den beiden Polynomen vom zweiten bis zum dritten Schnittpunkt:

$$A = \left| \int_{x_0}^{x_1} (y_1(x) - y_2(x))\, dx \right| \quad .$$

($A \approx 1$). Diese Integration wird nicht mit `quadl` durchgeführt, da man Polynome mit Hilfe der Funktion `polyint` integrieren kann. In das damit erhaltene Polynom setzt man dann die obere und untere Grenze ein und erhält so den Wert des bestimmten Integrals.

(f) Berechnen Sie den Umfang der eben bestimmten Fläche. Die Bogenlänge einer Kurve ist durch

$$s = \int_{x_0}^{x_1} \sqrt{1 + (y'(x))^2}\, dx$$

gegeben. ($s \approx 5.5$). Diese Aufgabe kann natürlich nicht mit `polyint` erledigt werden, da hier kein Polynom integriert wird. Dazu braucht man also wieder eine `inline`-Funktion und die MATLAB-Routine `quadl`.

(g) Legende, Beschriftung der Grafik.

(h) Bestimmen Sie die Krümmung

$$\kappa = \left| \frac{y''(x)}{[1 + (y'(x))^2]^{3/2}} \right|$$

und den Schmiegekreisradius

$$r = \frac{1}{\kappa}$$

im Maximum des zweiten Polynoms ($\kappa \approx 0.4$). Die Koordinaten des Schmiegekreismittelpunkts sind durch

$$m_x = x - \frac{y'(1+y'^2)}{y''}$$
$$m_y = y + \frac{1+y'^2}{y''}$$

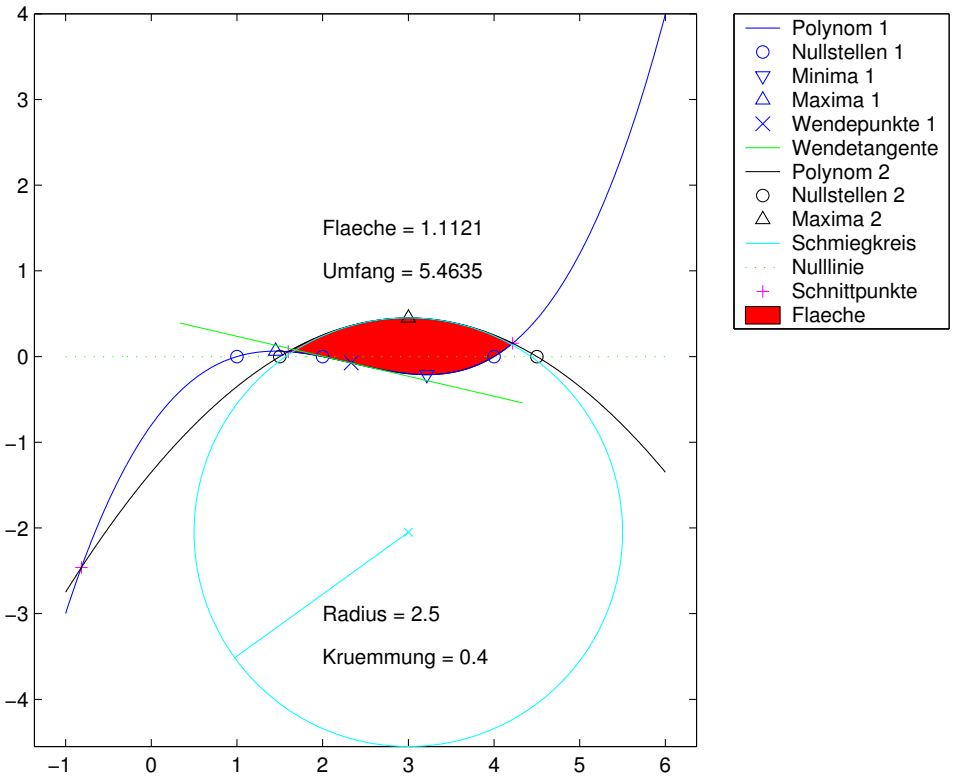
gegeben.
Schreiben Sie dafür ein Unterprogramm `kreis.m`, welches die x- und y-Werte für einen Kreis berechnet, der durch die Parameterdarstellung

$$x = m_x + r\cos(\phi)$$
$$y = m_y + r\sin(\phi)$$

beschrieben wird. Der Aufruf für das Programm sollte folgendermaßen lauten:

```
[x,y] = kreis(r,m,phi)
Defaultwerte: r = 1
m = [0,0]
phi = 0 ... 2*pi, 100 Punkte
```

Zeichnen Sie den Schmiegekreis samt Mittelpunkt und Radius ein.



Exemplarische Darstellung der Ergebnisse.

12.8 Darstellung von Daten

Ziel: Das Ziel der Übung ist es, den Umgang mit Daten und deren Visualisierung zu erlernen.

Voraussetzung: Speichern Sie die files `kerrtab-i.dat` und `kerrtab-pu.dat` in Ihrem MATLAB-Directory ab (Skript: uebungsdaten).

1. **Animation:**
   Erstellung eines Skripts `anim.m`, welches die Bewegung von Punkten entlang Kurven in einer Animation darstellt. Ein Punkt bewegt sich dabei entlang einer elliptischen Bahn, die unter Umständen im Laufe der Zeit eine Verkleinerung der Achsen erfährt. Diese Bahn wird beschrieben durch

   $$x_1(t) = a_1 cos(2\pi\nu_1 t)e^{-\kappa_1 t} \ , \tag{12.19}$$
   $$y_1(t) = b_1 sin(2\pi\nu_1 t)e^{-\kappa_1 t} \ . \tag{12.20}$$

   Dieser Punkt wird von einem zweiten Punkt umrundet, wobei seine Bahn durch

   $$x_2(t) = x_1(t) + a_2 cos(2\pi\nu_2 t)e^{-\kappa_2 t} \ , \tag{12.21}$$
   $$y_2(t) = y_1(t) + b_2 sin(2\pi\nu_2 t)e^{-\kappa_2 t} \ , \tag{12.22}$$

   beschrieben wird. Die Zeit t soll dabei von 0 in Zeitschritten von Δt bis t_{end} laufen. Verwenden Sie dabei als Defaultwert eine sinnvolle Kombination von Frequenzen ν, Achsen a und b, Dämpfungsfaktoren κ und Zeitwerten. Der erste Punkt soll dabei ein paar Umläufe machen, während in der zweite Punkt pro Umlauf den ersten mehrfach umkreist.

   Die Gestaltung der Graphik und die Auswahl der Defaultwerte ist Ihnen überlassen. Wählen Sie sie so, dass man eine interessante Animation sieht.

   Verwenden Sie dabei zur Erstellung der Graphik die Befehle `figure`, `axes`, `line`. Diese (alle) Graphikbefehle haben als Rückgabewert einen sogenannten Graphik-Handle, der einen weiteren Zugriff auf diese Graphikobjekte ermöglicht:

   ```
   fh = figure;
   clf reset;    % loescht alle Children, Eigenschaften
   set(fh,'Color',[1,1,0.8]);
   set(fh,.........)
   ```

   Damit kann man alle Eigenschaften von Graphikobjekten verändern. Dazu kann man sich im `helpbrowser` die Seite "Handle Graphics Property Browser" ansehen, die für alle Graphikobjekte Eigenschaften und mögliche Werte angibt. Eine Einführung zu diesem Konzept gibt es im Abschnitt 11.1. Verwenden Sie dieses Konzept um einige Einstellungen für `figure` und `axes` vorzunehmen.

Das gleiche Konzept wird zur Animation von Graphiken verwendet. Dabei tauscht man in einer Schleife (`while`) z.B. die Daten einer Linie (eines Punktes) aus:

```
lh = line(x,y,'LineStyle','none');
set(lh,'Marker','o')
% MarkerSize, MarkerEdgeColor, MarkerFaceColor, ....
t = t_anf;
while t <= t_end
  t = t + delta_t;
  x = ....
  y = ....
  set(lh,'XData',x,'YData',y);
  drawnow;
  pause( ........... )
end
```

Die Animation erfolgt also durch Austauschen der x- und y-Werte. Der Befehl `drawnow` stellt sicher, dass zu diesem Zeitpunkt ein Update der graphischen Ausgabe (Schirm) stattfindet. MATLAB merkt sich ansonsten Graphikbefehle in einer Art Pipeline und optimiert die Umsetzung.

Damit sollte es Ihnen möglich sein, dass sich die Punkte über die Graphik bewegen. Wichtig dabei ist, dass man mit den Eigenschaften `XLim` bzw. `YLim` die Achsenlimits gleich am Anfang in entsprechender Größe einstellt, damit sich diese nicht bei der Bewegung mitändern. In manchen Fällen, will man das Achsensystem so gestalten, das es sich über die gesamte Graphik ausdehnt und unsichtbar bleibt. Dies erreicht man mit Hilfe der Eigenschaften `Position` und `Visible`.

Ein wichtiger Punkt ist auch die oft notwendige Verzögerung der Graphikausgabe mit dem Befehl `pause`, da sonst oft die Animationen zu schnell abläuft. Dabei kann man mit den Befehlen `clock` und `etime` die gegenwärtige Zeit bzw. die verstrichene Zeit berechnen und damit ein vernünftiges Argument für `pause` berechnen:

```
t_0 = clock;
......
t_elapsed = etime(clock,t_0);
```

Wenn dieser Teil funktioniert, ergänzen Sie die Ausgabe um "Schweife" beschränkter Länge, die jeder Punkt nach sich zieht. Diese stellen quasi eine Spur dar. Dafür führt man weitere zwei Linien ein, die nun nicht mehr nur aus einem Punkt sondern aus einer Reihe von Punkten bestehen. Dies kann man dadurch erreichen, dass man an die entsprechenden Vektoren bei jedem Zeitschritt vorne

den neuen Punkt anfügt. Wenn die Vektoren die gewünschte Länge überschreiten, schneidet man den hinteren Bereich ab und animiert dann unter Verwendung von `XData` und `YData` diese Kurven.

2. **Kerrzelle:**
   Erstellung eines Skripts `kerr.m`.

   (a) Die Datei `kerrtab-i.dat` enthält eine Matrix mit folgendem Inhalt:

   - 1. Spalte: Winkel in Grad
   - 2. Spalte: Intensität von linear polarisiertem Licht
   - 3. Spalte: Intensität von zirkular polarisiertem Licht
   - 4. Spalte: Intensität von elliptisch polarisiertem Licht

   Die Daten (`load`) stammen von folgender Messung: Polarisiertes Licht wird durch einen Analysator geschickt. (Ein Analysator lässt nur eine Schwingungsrichtung von Licht durch.) Wird der Analysator gedreht, ändert sich die Intensität gemäß der oben gegebenen Matrix.
   Erzeugen Sie eine Grafik, in der alle drei Polarplots zu sehen sind. Beschriften Sie die Achsen und die Grafik entsprechend. Erzeugen Sie eine Legende.

   (b) Bringt man isotrope Dielektrika in ein homogenes elektrisches Feld, so erhalten diese optische Eigenschaften einachsiger Kristalle, das heißt, sie werden doppelbrechend. Für den Phasenunterschied $\Delta\varphi$ von ordentlichem und außerordentlichem Strahl folgt nach dem *empirischen Gesetz von Kerr*:

   $$\Delta\varphi = const \cdot U^2 \tag{12.23}$$

   $const$ . . . Konstante, die aber von der Wellenlänge des Lichtes abhängt
   U . . . Spannung an einem Kondensator, der das elektrische Feld erzeugt

   Die Datei `kerrtab-pu.dat` enthält folgende Daten:

   - 1. Spalte: U^2
   - 2. Spalte: $\Delta\varphi$ für grünes Licht
   - 3. Spalte: Fehler für grünes Licht
   - 4. Spalte: $\Delta\varphi$ für gelbes Licht
   - 5. Spalte: Fehler für gelbes Licht
   - 6. Spalte: $\Delta\varphi$ für blaus Licht
   - 7. Spalte: Fehler für blaus Licht

Erzeugen Sie eine Grafik, die $\Delta\varphi$ als Funktion von U^2 für die einzelnen Farben darstellt (alle drei in einer Grafik). Erforderlich: Beschriftung der Achsen und der Grafik; Fehlerbalken; Ausgleichsgeraden durch den Ursprung.
Achtung: Der Befehl polyfit legt die Ausgleichsgerade nicht unbedingt durch den Ursprung. Wenn man das erreichen will, darf man für die Gerade nur die Formel $y = kx$ verwenden. Die Steigung k ergibt sich dann aus

$$k = \frac{\sum_{I=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2},$$

wobei N die Anzahl der Datenpunkte ist.

12.9 Lineare Gleichungssysteme

Ziel: In dieser Übungseinheit soll der Umgang mit linearen Gleichungssystemen gelernt werden. Anzufertigen sind die MATLAB Script-Files `ueblinear.m` und `kegelgl.m`.

Voraussetzung: Grundlegendes Wissen über lineare Algebra, wie sie im Kapitel 5 besprochen werden.

12.9.1 Netzwerk

Gegeben ist ein Netzwerk mit ohmschen Widerständen und Spannungsquellen (Abb. 12.1). Die sich einstellenden Ströme kann man bequem über die Kirchhoffschen Regeln be-

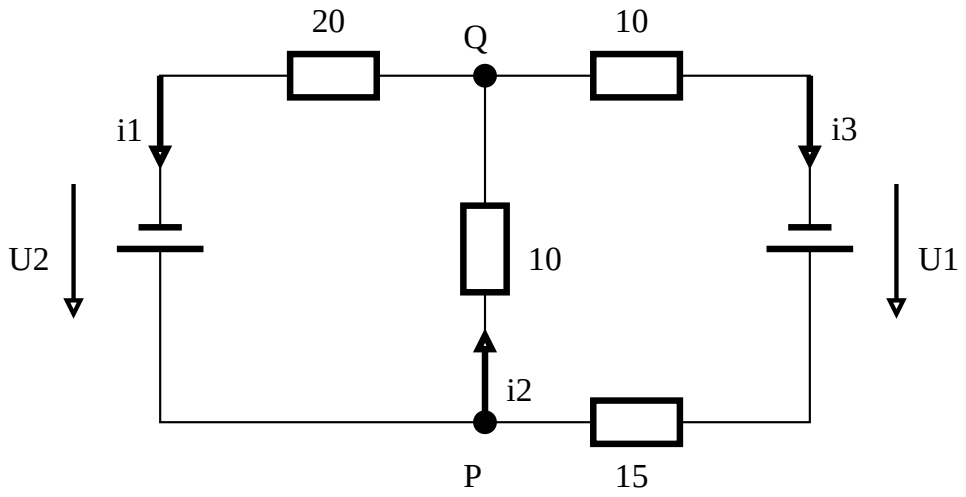


Abbildung 12.1: *Netzwerk mit 4 ohmschen Widerständen und 2 Spannungsquellen.*

stimmen:

a) Knotenregel: An jedem Knoten ist de Summe der einfließenden Ströme gleich der Summe der ausfließenden Ströme. In unserem Beispiel heißt das:

Knoten P:

$$i_1 - i_2 + i_3 = 0 \tag{12.24}$$

Knoten Q:

$$-i_1 + i_2 - i_3 = 0 \tag{12.25}$$

In diesem Fall sind das äquivalente Gleichungen, wovon nur eine benötigt wird.

b) Schleifenregel: In jeder Schleife ist die Summe der Spannungsabfälle gleich der Summe der Spannungsquellen.

Rechte Schleife:

$$10\,i_2 + (10 + 15)\,i_3 = U_1 \tag{12.26}$$

Linke Schleife:

$$20\,i_1 + 10\,i_2 = U_2 \tag{12.27}$$

Wir können dieses lineare Gleichungssystem in Matrixform schreiben.

$$\begin{bmatrix} 1 & -1 & 1 \\ 0 & 10 & 25 \\ 20 & 10 & 0 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 0 \\ U_1 \\ U_2 \end{bmatrix} \quad . \tag{12.28}$$

Lösen Sie diese Gleichungssystem für

a) $U_1 = 90$, $U_2 = 80$

b) $U_1 = 125$, $U_2 = 90$

c) Kann man dieses Gleichungssystem auch für mehrere Vektoren

$$\begin{bmatrix} 0 \\ U_1 \\ U_2 \end{bmatrix} \tag{12.29}$$

zugleich lösen? Lösen Sie das System für die Vektoren aus a), b) und für den Vektor mit $U_1 = 150$, $U_2 = 70$ mit einem Befehl.

Überlegen Sie, wie Sie dafür die Spaltenvektoren in einer Matrix anordnen müssen.

12.9.2 Vertauschung

Gegeben sind eine Matrix $\mathbf{A}$ und ein Vektor $\mathbf{b}$:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 4 & 0 \\ -2 & 3 & 1 \end{bmatrix} \quad , \qquad \mathbf{b} = \begin{bmatrix} 3 \\ 15 \\ 11 \end{bmatrix} \quad . \tag{12.30}$$

Lösen Sie

a) $\mathbf{A}\mathbf{x} = \mathbf{b}$

b) $\mathbf{x}\mathbf{A} = \mathbf{b}^T$

Sind die Ergebnisse gleich?

12.9.3 Diagonalmatrix

Die Matrix M ist eine $2n \times 2n$ Matrix, deren Hauptdiagonale aus Einsen besteht und deren beiden Nebendiagonalen mit 0.5 gefüllt sind

$$\mathbf{M} = \begin{bmatrix} 1 & 0.5 & 0 & 0 & \dots \\ 0.5 & 1 & 0.5 & 0 & \dots \\ 0 & 0.5 & 1 & 0.5 & \dots \\ 0 & 0 & 0.5 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} . \tag{12.31}$$

Der Vektor $\mathbf{v}$ ist durch

$$\mathbf{v} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 2 \\ 0 \\ 3 \\ \vdots \\ 0 \\ n \end{bmatrix} \tag{12.32}$$

gegeben. Erstellen Sie die Matrix M mit Hilfe der Befehle `eye` und `diag`, und den Vektor $\mathbf{v}$ mit Hilfe des Befehls `zeros` und der `Doppelpunk Notation`.

a) Berechnen Sie die Lösung der Gleichung $\mathbf{My} = \mathbf{v}$ für $n = 5$.

b) Machen Sie die Probe $\mathbf{My} - \mathbf{v}$. Kommt tatsächlich der Nullvektor heraus? Überlegen Sie sich, woher die Abweichungen herrühren!

Im letzten Punkt haben wir gesehen, dass durch die endliche Genauigkeit der Zahlen während der Berechnung Rundungsfehler entstehen.
Daher macht eine Probe in der Form

```
if all( (M*y - v) == 0 )
   disp('Probe bestanden.');
else
   disp('Probe nicht bestanden.');
end
```

keinen Sinn!

Die einzige Möglichkeit, die man hat, ist auf beinahe Gleichheit zu testen. Dazu bietet sich der *absolute* oder der *relative Fehler* an. Wir wollen hier den absoluten Fehler verwenden.

```
error_limit = 1.0e-8;
```

```
if all( abs(M*y - v) < error_limit )
   disp('Probe bestanden.');
else
   disp('Probe nicht bestanden.');
end
```

c) Versuchen Sie diese Programmzeilen zu verstehen!!! Führen Sie auf diese Weise die Probe durch.

12.9.4 Kegelschnitt

Als verallgemeinerte quadratische Formen bezeichnet man Funktionen folgender Form:

$$z(x, y) = s_1 x^2 + 2s_2 xy + s_3 y^2 + s_4 x + s_5 y + s_6 \ . \tag{12.33}$$

Schnitte einer solchen Fläche mit einer Ebene (z.B., $z(x, y) = 0$) bezeichnet man als Kegelschnitte (Ellipse, Kreis, Hyperbel, Parabel) bzw. als entartete Kegelschnitte (Gerade, Geradenpaar, Punkt).

Bei bekannten Datenpunkten x_d und y_d kann man $z(x_d, y_d) = 0$ als lineares Gleichungssystem für s_i auffassen

$$s_1 x_d^2 + 2s_2 x_d y_d + s_3 y_d^2 + s_4 x_d + s_5 y_d + s_6 = 0 \ . \tag{12.34}$$

In dieser Form ist es ein homogenes Gleichungssystem, das immer die triviale Lösung $s_i = 0$ liefern würde. Man kann sich aber helfen und einen der Koeffizienten $s_k = 1$ setzen und den entsprechenden Term auf die rechte Seite bringen. Wählt man z.B. den dritten Term $s_3 = 1$ lautet die das inhomogene Gleichungssystem

$$s_1 x_d^2 + 2s_2 x_d y_d + s_4 x_d + s_5 y_d + s_6 = -y_d^2 \ , \tag{12.35}$$

welches nun die Form $\mathbf{Ds} = \mathbf{b}$ hat und mit MATLAB gelöst werden kann. Da es nun fünf unbestimmte Koeffizienten gibt, genügen fünf Datenpunkte (x_d, y_d), um den Kegelschnitt genau zu bestimmen.

Gehen Sie nun im Skript `kegelgl.m` folgendermaßen vor:

- Erzeugen Sie zwei (5×1)-Vektoren `xd` und `yd` mit gleichverteilten Zufallszahlen zwischen -0.5 und 0.5 (`rand`).
- Erzeugen Sie mit Hilfe dieser Vektoren die Hilfsmatrix

  $$M = [x_d^2, 2x_d y_d, y_d^2, x_d, y_d, 1] \ ,$$

  wobei x_d, y_d und 1 jeweils Spaltenvektoren gleicher Länge sein müssen.

- Wählen Sie den Index einer Spalte aus, die auf die rechte Seite des Gleichungssystems gebracht werden soll und erzeugen Sie damit die Matrix $\mathbf{D}$ und den Inhomogenitätsvektor $\mathbf{b}$. Diese Auswahl kann man am leichtesten mit Hilfe eines logischen Vektors (z.B.: $[0, 0, 1, 0, 0, 0]$) und seiner Negation treffen (logische Indizierung).

- Lösen Sie das entsprechende Gleichungssytem $\mathbf{Ds} = \mathbf{b}$ und fügen im so erhaltenen $\mathbf{s}$ an der richtigen Stelle 1 ein (logische Indizierung). Damit ist das Problem gelöst und das Ergebnis muss nur noch visualisiert werden.

- Erzeugen Sie dafür Vektoren x und y zwischen -1 und 1 mit circa 100 Punkten. Alle Kombinationen von x- und y-Werten kann man mit dem Befehl `meshgrid` erzeugen und erhält damit die Matrizen $\mathbf{X}$ und $\mathbf{Y}$. Durch Auswertung von Gleichung 12.33 für diese Matrizen erhält man $\mathbf{Z}(\mathbf{X}, \mathbf{Y})$.

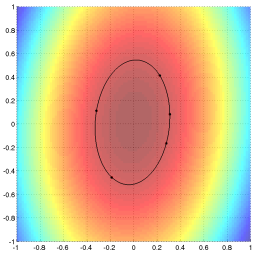
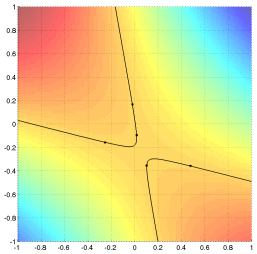
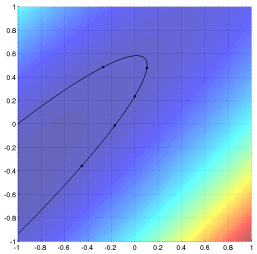
- Diese Fläche kann man mit dem Befehl `surf` graphisch Darstellen. Die Farbe repräsentiert den Wert der Funktion und man kann von Oben auf die Fläche schauen (`view(2)`). Verwendet man eine Handle für diese "Surface", kann man die `EdgeColor` auf `none` setzen um die störenden schwarzen Linien zu beseitigen.

- Nun kann man in der gleichen Zeichnung die Datenpunkte einzeichen (`line`) und den erhaltenen Kegelschnitt als Höhenschichtlinie zeichnen. Dafür gibt es in MATLAB den Befehl `contour(X,Y,Z)`. In dieser Form wählt MATLAB die z-Werte für die Höhenschichtlinien automatisch aus. Man kann aber auch einen weiteren Vektor `v` mit gewünschten Höhenwerten übergeben:

  ```
  v = [0,0]; contour(X,Y,Z,v);
  ```

  Will man nur einen Höhenwert muss man ihn seltsamerweise zweimal übergeben. Vergessen Sie auch nicht die Verwendung von `hold`.

- Damit sollten nun die Höhe von z in Farbe dargestellt sein, die Datenpunkte und der Kegelschnitt als Höhenschichtlinie eingezeichnet sein. Der Kegelschnitt sollte genau durch die fünf Datenpunkte gehen und entlang einer gleichen Farbe verlaufen.

12.10 Lineares Fitten

Ziel: Es soll das Fitten von Polynomen durch eine gegebene Datenmenge erlernt werden.

Voraussetzung: Speichern Sie die Datei `polydat.dat` in Ihrem MATLAB-Directory ab (Skript: uebungsdaten, Übung 10). Damit bekommen Sie auch eine Reihe von Datenfiles für Kegelschnitte, die alle mit `kschnitt` beginnen. Die Befehle `polyval` und `polyfit` erleichtern den Umgang mit Polynomen. Das Lösen von überbestimmten Gleichungssystemen kann mit dem \-Operator erfolgen. Das Kapitel 10 beinhaltet eine detailierte Beschreibung der Probleme.

Skriptnamen: `fitlin.m`, `skegel.m`, `kegelfit.m`

12.10.1 Fitten von Polynomen

1. Die Datei `polydat.dat` enthält eine Messkurve. Die erste Spalte enthält die x-Werte, die zweite die y-Werte und die dritte den Fehler Δy. Lesen Sie diese Daten ein.

2. Bestimmen Sie die besten Polynome nullter, erster, zweiter und dritter Ordnung, die durch diese Daten passen.

3. Zeichnen Sie die Datenpunkte mit Fehlerbalken (`errorbar`) und die gefitteten Polynome (Beschriftung, Legende, ...).

12.10.2 Fitten von Polynomen mit teilweise vorgegebenen Koeffizienten

1. Schreiben Sie ein Programm, das durch die Daten `polydat.dat` ein Polynom 3. Ordnung legt, bei dem jedoch zwei Koeffizienten bekannt sind:

   - Koeffizient der 3. Ordnung (also der Koeffizient vor x^3): $a_3 = 1$ (Achtung der Koeffizient zur höchsten vorkommenden Potenz von x steht in der MATLAB-Polynomdarstellung immer an erster Stelle im Koeffizientenvektor)
   - Koeffizient der 1. Ordnung (also der Koeffizient vor x): $a_1 = -3$

   Dafür müssen Sie nun selbst ein lineares Gleichungssystem erstellen und dieses mit Hilfe des \-Operators lösen. Dies liefert Ihnen dann die gewünschten Koeffizienten. Überlegen Sie sich gut, wie die rechte Seite dieses Gleichungssystems auszusehen hat.

2. Zeichnen Sie die Datenpunkte mit Fehlerbalken und das gefittete Polynom (Beschriftung, Legende, ...).

12.10.3 Erzeugen von verrauschten Daten zu Testzwecken

1. Erzeugen Sie für ein beliebiges Polynom verrauschte Testdaten. Wählen Sie dazu einen Vektor mit x-Werten und berechnen den entsprechenden Vektor mit y-Werten. Verrauschen Sie die Daten, indem Sie einen normalverteilten Zufallsvektor mit der Varianz σ addieren (`sig*randn(size(x))`).
2. Stellen Sie das Polynom und die verrauschten Daten mit Fehlerbalken ($\Delta y = \sigma$) grafisch dar.
3. Fitten Sie ein Polynom entsprechender Ordnung (mit oder ohne teilweise vorgegebene Koeffizienten) und stellen Sie es in derselben Figur grafisch dar.

12.10.4 Fitten von Kegelschnitten

Wie schon in Übung 12.9.4 besprochen, kann man Kegelschnitte durch verallgemeinerte quadratische Formen darstellen

$$z(x, y) = s_1 x^2 + 2s_2 xy + s_3 y^2 + s_4 x + s_5 y + s_6 = 0 \; . \tag{12.36}$$

Liegen nun n Datenpunkte für x und y in den Spaltenvektren x_d und y_d vor, kann man einen der Koeffizienten s_i mit einem vorgegebenen Wert belegen (z.B.: $s_6 = 1$) und den entsprechenden Term auf die rechte Seite des Gleichungssystems bringen

$$s_1 x_d^2 + 2s_2 x_d y_d + s_3 y_d^2 + s_4 x_d + s_5 y_d = -s_6 \; , \tag{12.37}$$

wobei dieses Gleichungssystem nun nicht mehr homogen ist. Ist $n = 5$ bekommt man als Lösung des Gleichungssystems einen Kegelschnitt auf dem alle Datenpunkte liegen. Ist $n > 5$ hat man es mit einem überbestimmten linearen Gleichungssystem zu tun, das von MATLAB im "Least Squares" Verfahren gelöst wird. Damit wird jener Kegelschnitt gefunden für den die Summe der Abstandsquadrate

$$S = \sum_{d=1}^{n} z(x_d, y_d)^2 \tag{12.38}$$

ein Minimum ist.

Probleme mit der Lösung des Gleichungssystems bekommt man, wenn ein "ungeeigneter" Term auf die rechte Seite verschoben wird. Ungeeignet sind Terme, deren Koeffizienten eigentlich Null ergeben würden, wie z.B. bei der Parabel $y = x^2$ der oben gewählte Trem s_6. Daher lohnt es sich, alle 6 Terme nacheinander auf die rechte Seite zu bringen, das jeweilige Gleichungssystem zu lösen und die Summe der Fehlerquadrate zu berechnen. Damit kann man dann jene Lösung mit der geringsten Summe der Fehlerquadrate als besten Fit verwenden.

1. Erzeugen Sie jetzt ein MATLAB-Skript `skegel.m`, das die Namen aller zugehörigen Datenfiles einliest. Dazu gibt es die Möglichkeit diese Namen in eine MATLAB-Struktur einzulesen:

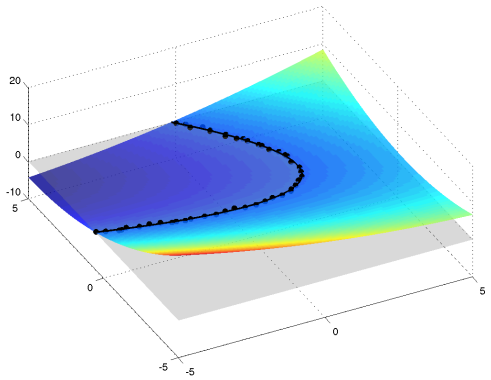
```
files = dir('kschnitt*.dat');  % Struktur
anz   = length(files);         % Anzahl
k = 5; file = files(k).name;   % Name des 5. Files
```

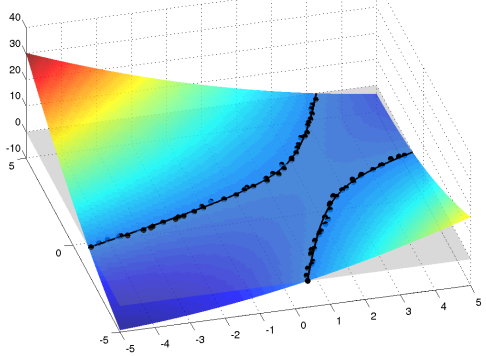
   Der Benutzer soll dann eine Zahl und damit einen Filenamen auswählen, den File einlesen und die Datenvektoren `xd` und `yd` belegen.

2. Diese Daten werden der MATLAB-Funktion
   `[s,err2] = kegelfit(xd,yd)`
   übergeben, die dann den besten Fit für die Koeffizienten s_i und die zugehörige Summe der Fehlerquadrate `err2` zurückgibt.

   Folgen Sie bei der Erstellung der Funktion der oben beschriebenen Strategie alle 6 Terme abwechselnd auf die rechte Seite zu bringen. Für die Aufteilung in Koeffizientenmatrix und Inhomogenitätsvektor eignet sich die logische Indizierung hervorragend. Damit kann man das Problem leicht in einer `for`-Schleife erledigen.

3. Erstellen Sie danach im Skript eine Darstellung der Fläche $z(x, y)$ für x- und y-Werte ungefähr im Bereich der Daten, zeichnen Sie ein Konturlinie bei $z = 0$ (entspricht Kegelschnitt) ein und plotten die zugehörigen Datenpunkte. Diese sollten, wenn alles richtig ist, in der nahen Umgebung des Kegelschnittes liegen.





12.11 Nichtlineares Fitten

Ziel: Das Ziel der Übung ist das Erlernen der Kurvenanpassung mit Hilfe nichtlinearer Modelfunktionen. Außerdem soll die Auswertung solcher Daten (Nullstellen, Minima, Fläche) erlernt werden. Abzugeben sind die beiden Skriptfiles `expfit.m` und `sinfit.m`.

Voraussetzung: Mitschrift der Vorlesungsstunde über die Verwendung von Inline-Funktionen und über Funktionen zum Minimieren, Nullstellensuchen und Integrieren. Laden Sie die Files `expfun1.dat` und `sinfun1.dat` in Ihr MATLAB-Directory.

1. **Exponentialfunktion:** Beim radioaktiven Zerfall ergibt sich bei einer Zerfallskette von Isotop 1 zu Isotop 2 (Halbwertszeit τ_1) und dem weiteren Zerfall von Isotop 2 (Halbwertszeit τ_2) folgender Typ von Gleichung für die Anzahl der Teilchensorte 2

$$y(t) = a_1 \left[1 - \exp(-a_2 t)\right] \exp(-a_3 t) \ . \tag{12.39}$$

   Im File `expfun1.dat` finden Sie in jeder Zeile Messwerte für t und $y(t)$. Fitten Sie dazu die obige Modellfunktion und stellen Sie sie zusammen mit den Datenwerten dar. Verwenden Sie bei der Darstellung eine größere Endzeit, Sie sollten dabei einen schönen exponentiellen Abfall der Kurve sehen.

   Für die Definition obiger Gleichung in MATLAB verwendet man den MATLAB-Befehl `inline`. Damit können einfache Funktionen direkt in einer Zeile eingegeben werden, ohne dass man ein Skriptfile schreiben muss. Zur nichtlinearen Anpassung kann man die Funktion nlinfit verwenden. Im Gegensatz zum linearen Fitten, wo sich aus dem Lösen des linearen Gleichungssystems immer eine exakte Lösung ergibt, braucht man hier Anfangswerte für die Parameter a, die nicht zu weit von der Lösung entfernt sein sollen. In diesem Fall liegen sie in der Nähe von `0.8, 4, 0.5`. Bedenken Sie auch, dass bei nlinfit die Inline-Funktion für y als $y(a, t)$ (zuerst der Paramtervektor a, dann der Zeitvektor t) aufgerufen wird.

   Der Zusammenhang zwischen den Parametern a und den physikalischen Größen beim Zerfall lautet:

$$\tau_1 = \ln 2/a_3 \ \ , \ \ \tau_2 = \ln 2/(a_2 + a_3) \ \ , \ \ y_{01} = a_1 a_2/a_3 \ , \tag{12.40}$$

   wobei y_{01} die Anzahl der Teilchen 1 zum Zeitpunkt $t = 0$ ist.

   Außerdem soll das Maximum der Funktion $y_{max} = y(t_{max})$ gefunden und im Plot mit einem Marker dargestellt werden. Dazu verwendet man die Funktion `fminsearch`, wobei man als Startwert wieder einen Zeitwert in der Nähe des Maximums angeben muss. Vom Programm wird nun aber bei der Suche nach dem Maximum jeweils t bei fixen Werten von a verändert. Daher muss man eine modifizierte Inline-Funktion mit vertauschten Parametern schreiben. Ein

zweiter Punkt zum Aufpassen liegt darin, dass wir hier ein Maximum suchen, `fminsearch` aber, wie schon der Name sagt, ein Minimum finden will. Daher muss diese Inline-Funktion $-y(t, a)$ enthalten.

Sie können Ihr numerisches Ergebnis mit dem analytischen vergleichen (ein bisschen Analysis schadet nicht)

$$t_{max} = -\ln(a_3/(a_2+a_3))/a_2 \quad , \quad y_{max} = \frac{a_1 a_2 \exp(a_3 \ln(a_3/(a_2+a_3))/a_2)}{a_2+a_3} \; . \tag{12.41}$$

Auch das Integral für die Fläche unter der gesamten Kurve

$$\int_0^\infty y(t)dt = \frac{a_1 a_2}{(a_2+a_3)a_3} \; , \tag{12.42}$$

soll mit dem numerischen Resultat verglichen werden. Dafür kann man die Routine `quadl` verwenden. Der Aufruf der Funktion geht auch hier mit $y(t, a)$. Aufpassen muss man bei der oberen Grenze, da man bei numerischen Rechnungen nicht ∞ einsetzen kann. Man darf daher die obere Grenze weder zu klein noch zu groß wählen.

2. **Sinusfunktion** Fitten Sie die Daten in `sinfun1.dat` mit Hilfe der nichtlinearen Modelfunktion

$$y(a, x) = a_1 x \sin^2(a_2 x) \; , \tag{12.43}$$

und stellen Sie sie zusammen mit den Daten im Intervall $[0, 4\pi]$ dar. Die Startwerte liegen im Bereich von `1.8, 1.1`.

Finden Sie die ersten vier Minima außer jenem bei Null (die Minima entsprechen hier auch den Nullstellen) und die ersten vier Maxima und markieren Sie sie im Plot (Tipp: Zwei neue Inline-Funktionen $y(x, a)$ und $-y(x, a)$).

Berechnen Sie das Integral der Funktion zwischen 0 und dem vierten Minimum. Der Wert sollte ungefähr 60 ergeben.

3. FREIWILLIG
MATLAB kann nicht nur numerisch sondern auch symbolisch rechnen. Die symbolischen Rechnungen zum ersten Teil der Übung kann man sich hier anschauen und auch ausprobieren.

```
% Definition der Konstanten als symbolische Groessen

a1 = sym('a1','positive');

a2 = sym('a2','positive');

a3 = sym('a3','positive');
```

```
% Definition der Zeit (hier xx) und der Grenzen

xx = sym('xx','positive');

syms gl gu

% Funktion

yy = a1*(1-exp(-a2*xx)).*exp(-a3*xx)

dyy = diff(yy,xx) % Ableitung nach xx

% Maximum (Nullstelle der ersten Ableitung)

maxx = solve(dyy,xx) % Loese dyy=0 nach xx auf

maxy = factor(subs(yy,xx,maxx)) % Setze fuer xx ein

% Integral

iyy = int(yy,xx,gl,gu)

% Unter Grenze 0; Obere Grenze Limes gegen unendlich

diyy = limit(subs(iyy,gl,0),gu,inf)
```

Hier wird zuerst mit symbolischen Größen gearbeitet und erst durch `subs` wird z.B. für die unter Grenze 0 zugewiesen. Die Ergebnisse kann man mit den obigen Formeln vergleichen.

Die symbolischen Rechnungen für den zweiten Teil der Übung finden Sie hier.

```
% Definition der symbolischen Groessen

a1 = sym('a1','positive');

a2 = sym('a2','positive');

xx = sym('xx','positive');

syms gl gu

% Funktion und Ableitung
```

```
ee = a1*xx*sin(a2*xx)^2

dee = diff(ee,xx)

% Unbestimmtes und bestimmtes Integral

iiee = int(ee,xx)

iee  = int(ee,xx,gl,gu)

% Einsetzen der Werte, wobei davon ausgegangen wird, dass die

% Parameter a im Vektor af und dass die Minima im Vektor xmin

% stehen.

fl = subs(iee,{a1,a2,gl,gu},{af(1),af(2),0,xmin(4)});
```

Mit `solve` findet man hier nur das triviale Minimum bei 0, dies hilft einem also hier nicht weiter. Nach Anschauen der Funktion erkennt man aber, dass die Minima natürlich bei $x_{min} = k\pi/a_2$ liegen müssen, wobei k eine ganze Zahl ist. Für die Maxima gibt es keine analytische Lösung.

12.12 Fourierreihen

Ziel: In dieser Übungseinheit soll nochmals die Erstellung von Unterprogrammen und der Umgang mit diesen erlernt werden. Darüber hinaus benötigt man `inline`-Funktionen und die Integrationsroutine `quadl`. Abzugeben sind die Unterprogramme `fouriercoeff.m`, `fourierexpansion.m` und das Skript `fouriertest.m`.

Voraussetzung: Voraussetzung für die Übung ist das Kapitel über Programmeinheiten 7.

Eine 2π-periodische Funktion $f(x)$ kann mit Hilfe einer Fourierreihe approximiert werden. Diese Reihe konvergiert unter gewissen Bedingungen gegen die Funktion $f(x)$ und spielt in der Physik eine wichtige Rolle (Wellenlehre, Wärmeleitung, Quantenmechanik, lineare Differentialgleichungen).

Die n-te Teilsumme der Fourierreihe einer Funktion $f(x)$ lautet

$$s_n(x) = m + \sum_{k=1}^{n} \Big(a_k \cos(kx) + b_k \sin(kx) \Big) \; . \tag{12.44}$$

Die Information über die Funktion $f(x)$ ist in den Koeffizienten $m, \vec{a}$ und $\vec{b}$ enthalten. Diese erhält man aus der Funktion $f(x)$ durch Berechnung von Integralen. Der Koeffizient m ist durch den Mittelwert

$$m = \frac{1}{2\pi} \int_{-\pi}^{\pi} \mathrm{d}x \, f(x)$$

der Funktion gegeben. Die Koeffizienten a_k und b_k ergeben sich aus den Formeln

$$\begin{aligned} a_k &= \frac{1}{\pi} \int_{-\pi}^{\pi} \mathrm{d}x \, \big(f(x) \cos(kx) \big) \\ b_k &= \frac{1}{\pi} \int_{-\pi}^{\pi} \mathrm{d}x \, \big(f(x) \sin(kx) \big) \end{aligned} \qquad \text{mit} \quad k = 1, \ldots, n \; .$$

1. Schreiben Sie eine MATLAB-Funktion `[m,a,b]=fouriercoeff(fun,n)`, die die Fourierkoeffizienten der Funktion `fun` berechnet und in den Variablen `m,a,b` zurückgibt. Der Funktion `fouriercoeff` soll die zu entwickelnde Funktion als `inline`-Funktion im Argument `fun` übergeben werden. Weiters braucht `fouriercoeff` noch die Ordnung n, bis zu der die Fourierkoeffizienten berechnet werden sollen.

   Die Integrale können mit `quadl` ausgeführt werden. Für die sin/cos-Integrale müssen `inline`-Funktionen `funccos` und `funcsin` definiert werden, die die Integranden bilden.

   Tipp: Bei `inline`-Funktionen kann man mit den Befehlen `formula` und `argnames` die Funktion und ihre Argumente als String bzw. Zelle erhalten

```
f1   = inline('exp((x-x0).^2)','x','x0')
form = formula(f1)    ->   'exp((x-x0).^2)'
args = argnames(f1)   ->   {'x','x0'}
```

und bekommt damit die benötigten Informationen. Bei der Definition der `inline`-Funktionen verwendet man als erstes Argument eine Zeichenkette, die jeweils die zu integrierende Funktion enthält. Für die sin/cos-Integrale muss diese Zeichenkette sinnvoll aus `form`, `args` und sin/cos zusammengesetzt werden. Teilbereiche aus Zellen bekommt man mit `args{1}` oder `args{2:end}`. Das Zusammensetzen von Zeichenketten erfolgt wie bei Vektoren mit `[]`, z.B. liefert `['sin','(x)']` den String `'sin(x)'`. Bedenken Sie auch, dass der Integrand neben x eine weitere Inputvariable k benötigt. Diese muss über `quadl` an die Routine für den Integranden "durchgeschleust" werden.

Probieren Sie am Anfang des Skripts `fouriertest` `[m,a,b]=fouriercoeff(f1,4)` mit der `inline`-Funktion für `f1=sin(x)` aus. Funktioniert es und ist das Ergebnis sinnvoll? Warum bekommt man für die Koeffizienten, die eigentlich 0 sein müssten, sehr kleine Werte? Wie kann man das in `fouriercoeff` beheben?

Probieren Sie ihr Programm mit den `inline`-Funktionen für $\sin^2(x)$ oder $\cos^2(x)$ aus. Wie schaut es jetzt mit den Fourier-Koeffizienten aus?

2. Schreiben Sie eine Funktion `s=fourierexpansion(x,m,a,b)`, die aus den Koeffizienten m, a, b die Werte der Summe (12.44) an den Stellen x ausrechnet und in s zurückgibt.

3. Zeigen Sie im MATLAB-Skript `fouriertest.m` die Entwicklung von zwei der im Bild 12.2 gezeichneten Funktionen in eine Fourierreihe und stellen das Ergebnis für verschiedene Ordnungen n dar. Dazu müssen Sie die (blau) gezeichneten Funktionen als `inline`-Funktionen programmieren. Denken Sie dabei an die Verwendung von Absolutbetrag und Signum-Funktion.

4. Entwickeln Sie weiters im Skript `fouriertest.m` die 2π-periodischen Funktionen

$$
\begin{aligned}
f(x) &= e^{-|\hat{x}|} && \text{(Exponentieller Abfall)} \\
f(x) &= \sqrt{1-(\hat{x}/\pi)^2} && \text{(Halbellipse)} \\
f(x) &= \frac{1/5}{\hat{x}^2+1/5} && \text{(Lorentzfunktion)} \\
f(x) &= e^{-4\hat{x}^2} && \text{(Gaußglocke)}
\end{aligned}
$$

mit $\hat{x} = (x+\pi) \bmod 2\pi - \pi$ in Fourierreihen und stellen Sie die Ergebnisse für verschiedene Ordnungen n und $x \in [-3\pi, 3\pi]$ grafisch dar. Dafür müssen Sie wieder `inline`-Funktionen schreiben. Für die graphische Ausgabe obiger Funktionen muss man diese periodisch fortsetzen. Das gelingt am Besten mit Hilfe der Modulofunktion `mod` und obiger Formel für $\hat{x}$. Die Entwicklung der

periodischen Halbellipsen und der Lorentzfunktion ist in Abbildung 12.3 zu sehen.

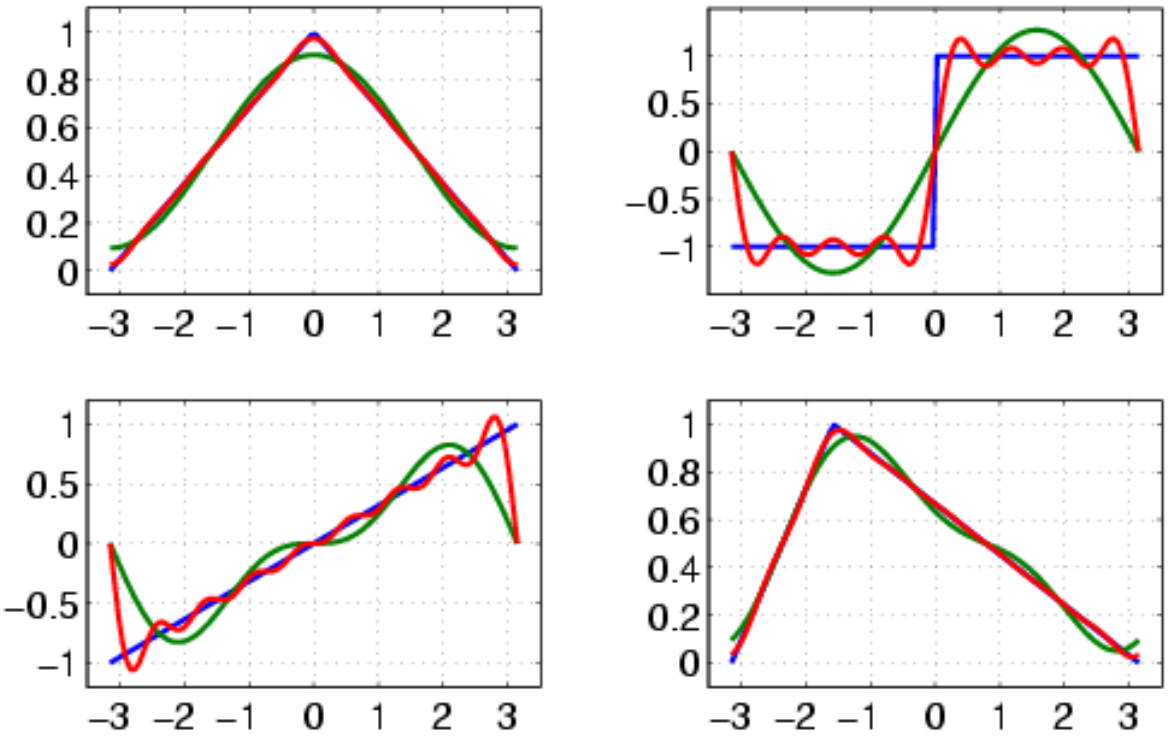


Abbildung 12.2: Verschiedene 2π-periodische Funktionen (blau) und deren Fourierentwicklung 2. Ordnung (grün) und 8. Ordnung (rot)

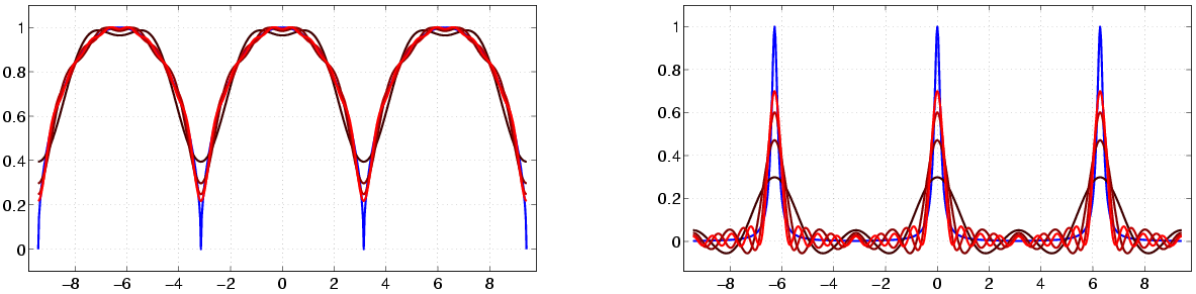


Abbildung 12.3: Periodische Halbellipsen (blau, oben) und Lorentz-Peaks (blau, unten) und deren Fourier-Approximationen der Ordnungen 2 (dunkelrot), 4, 6 und 8 (hellrot).

In der Folge finden sich einige Übungsbeispiele, die momentan nicht verwendet werden.

12.13 Funktionen

Ziel: In dieser Übungseinheit soll die Erstellung von Unterprogrammen und der Umgang mit diesen erlernt werden. Es werden drei Skriptfiles zur Verfügung gestellt:

`scschleife.m`, `scpeaks3.m`, `sczykloide.m`

Diese Skripts dienen zum Testen der zu erstellenden Funktionen und stellen die Ergebnisse außerdem graphisch dar. Abzugeben sind fünf MATLAB-Funktionen und ein kleines Skript:

`schleife.m, peaks3.m, peaks3a.m, epizykloide.m, hypozykloide.m,`
`scinttest.m`

Die einzelnen Funktionen sind relativ kurz und sollen das Wissen um Funktionen und Operatoren festigen.

Voraussetzung: Die vorbereiteten Programme für die Übung können mit dem Befehl

```
!uebungsdaten
```

geladen werden. Die Übungsnummer ist 5.

Voraussetzung für die Übung sind vor allem die Kapitel 7 und 4 und die entsprechenden Diskussionen in der Vorlesung.

1. Die folgende Funktion und ihre Ableitung soll in Form einer MATLAB-Funktion `schleife.m` programmiert werden:

$$y_{1,2} = \pm\frac{x}{a}\sqrt{a^4 - x^2} \qquad |x| \le a^2 \tag{12.45}$$

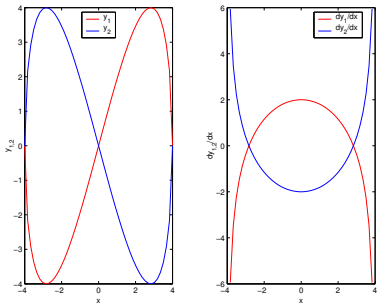
$$\frac{dy_{1,2}}{dx} = \pm\frac{a^4 - 2x^2}{a\sqrt{a^4 - x^2}} \qquad |x| \le a^2$$

Der Aufruf für das Programm erfolgt mit

```
[y1,dy1] = schleife(x,a),
```

wobei außerhalb des Gültigkeitsbereichs von 12.45 `NaN` zurückgegeben werden soll. Die Variable `x` ist ein Vektor und `a` ist ein Skalar. Die Outputgrößen sollen Felder der gleichen Größe wie `x` sein. Die Werte für $y_2 = -y_1$ und $dy_2/dx = -dy_1/dx$ brauchen nicht zurückgegeben zu werden.

Die mögliche Fehlermeldung bei der Divison durch 0 an der Bereichsgrenze kann mit Hilfe der Befehle `warning('off')` und `warning('on')` vor und nach der betreffenden Zeile im Programm verhindert werden.



Ergebnis von `schleife` für $a = 2$.

2. Die folgende Funktion soll in Form einer MATLAB-Funktion `peaks3.m` programmiert werden:

$$\begin{aligned} z(x,y,a) &= 3a_1(1-\hat{x})^2 \exp\left(-\hat{x}^2-(\hat{y}+1)^2\right) \\ &- 10a_2\left(\hat{x}/5-\hat{x}^3-\hat{y}^5\right)\exp\left(-\hat{x}^2-\hat{y}^2\right) \\ &- \frac{a_3}{3}\exp\left(-(\hat{x}+1)^2-\hat{y}^2\right)\ , \\ \hat{x}(x) &= (x+\pi)\bmod(2\pi)-\pi\ , \\ \hat{y}(y) &= (y+\pi)\bmod(2\pi)-\pi\ . \end{aligned} \tag{12.46}$$

Diese Funktion ist periodisch, das heißt

$$z(x+2m\pi, y+2n\pi, a) = z(x,y,a) \qquad m,n\in\mathbb{Z}\ , \tag{12.47}$$

wobei das Periodizitätsintervall $[-\pi,\pi]$ ist.

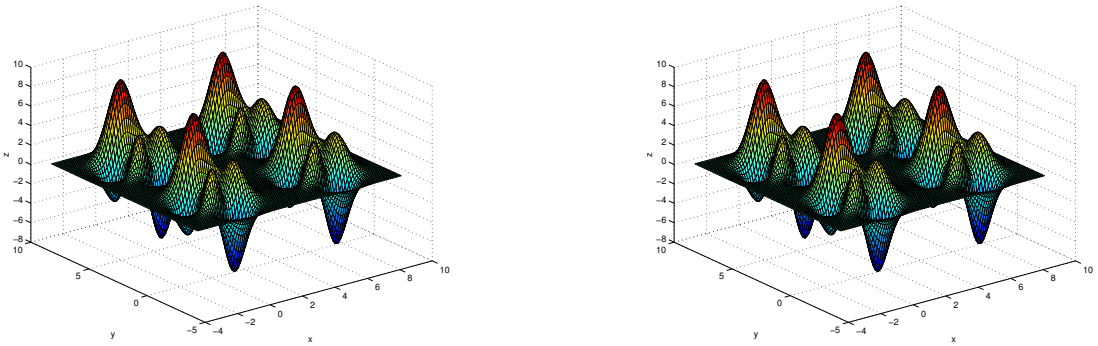
Der Aufruf für das Programm erfolgt mit

`[xx,yy,zz] = peaks3(x,y,a)`, bzw. mit
`[zz] = peaks3a(xx,yy,a)`.

Die Variable `a` enthält die Parameter a_i in Form eines Vektors. Die Variablen `x` und `y` sind auch Vektoren, wobei die Funktion für alle Kombinationen von `x` und `y` berechnet werden soll. Dazu wird zuerst der Befehl `meshgrid` verwendet, mit dem zwei gleich große Matrizen `xx` und `yy` erzeugt werden (siehe dazu auch 3.4.3 und insbesonders 3.1).

Mit diesen Matrizen für `xx` und `yy` sollte nun die zweite Funktion gerufen werden. Bevor man nun die Formel für `zz` auswerten kann, muß man noch mit Hilfe des Befehls `mod` aus `xx` und `yy` die neuen Variablen `xh` und `yh` $(\hat{x},\hat{y})$ berechnen. Dadurch wird die Periodizität 12.47 sichergestellt. Diese Matrizen enthalten nun alle Werte um damit die Formel 12.46 auszuwerten. Überlegen Sie, welche Operatoren Sie verwenden müssen.

Falls keine Werte für `x` und/oder `y` übergeben werden, sollten als Defaultwert 49 Punkte im Intervall $[-\pi, \pi]$ verwendet werden (`linspace`). Falls vom Dreiervektor `a` keine oder zuwenige Werte übergeben werden, sollten die fehlenden Werte auf 1 gesetzt werden.



Ergebnis von `peaks3` für $a = [1, 1, 1]$ im Intervall $[-\pi, \pi]$ und mit periodischer Fortsetzung.

3. Zykloiden (Rollkurven) sind Kurven, die dadurch entstehen, dass ein Kreis **K** auf einer Leitkurve **L** abrollt ohne zu gleiten. Bezeichnet M einen Punkt auf der Peripherie des Kreises **K**, so beschreibt M bei dieser Bewegung eine Zykloide. Ist C der Mittelpunkt von **K** und M′ ein Punkt auf der Verbindungsstrecke CM, so beschreibt M′ eine verkürzte Zykloide. Liegt M″ außerhalb des Kreises auf der Verlängerung von CM, so beschreibt M″ eine verlängerte Zykloide.

   Ist die Leitkurve **L** ebenfalls ein Kreis, so spricht man von Epizykloiden, wenn der Kreis **K** auf der Außenseite abrollt, und von Hypozykloiden, wenn der Kreis **K** auf der Innenseite abrollt.

   Ist a der Radius des rollenden Kreises **K**, b der Radius der Leitkurve **L** und λ der Verlängerungs- bzw. Verkürzungsfaktor, so gilt für die Parameterdarstellung der Epizykloide,

$$\begin{aligned} x &= (a+b)\cos\phi - \lambda a \cos\left(\frac{(a+b)\phi}{a}\right) \\ y &= (a+b)\sin\phi - \lambda a \sin\left(\frac{(a+b)\phi}{a}\right) , \end{aligned} \tag{12.48}$$

   mit

$$a = \frac{qb}{p} > 0, \qquad b > 0, \qquad p, q \in \mathbb{N}, \qquad 0 \le \phi \le 2q\pi \; . \tag{12.49}$$

   Für die Hypozykloide gilt,

$$\begin{aligned} x &= (b-a)\cos\phi - \lambda a \cos\left(\frac{(b-a)\phi}{a}\right) \\ y &= (b-a)\sin\phi - \lambda a \sin\left(\frac{(b-a)\phi}{a}\right) , \end{aligned} \tag{12.50}$$

mit

$$a = \frac{qb}{p}, \qquad b > a > 0, \qquad p, q \in \mathbb{N}, \qquad 0 \leq \phi \leq 2q\pi \; . \tag{12.51}$$

Zu programmieren sind nun zwei Funktionen

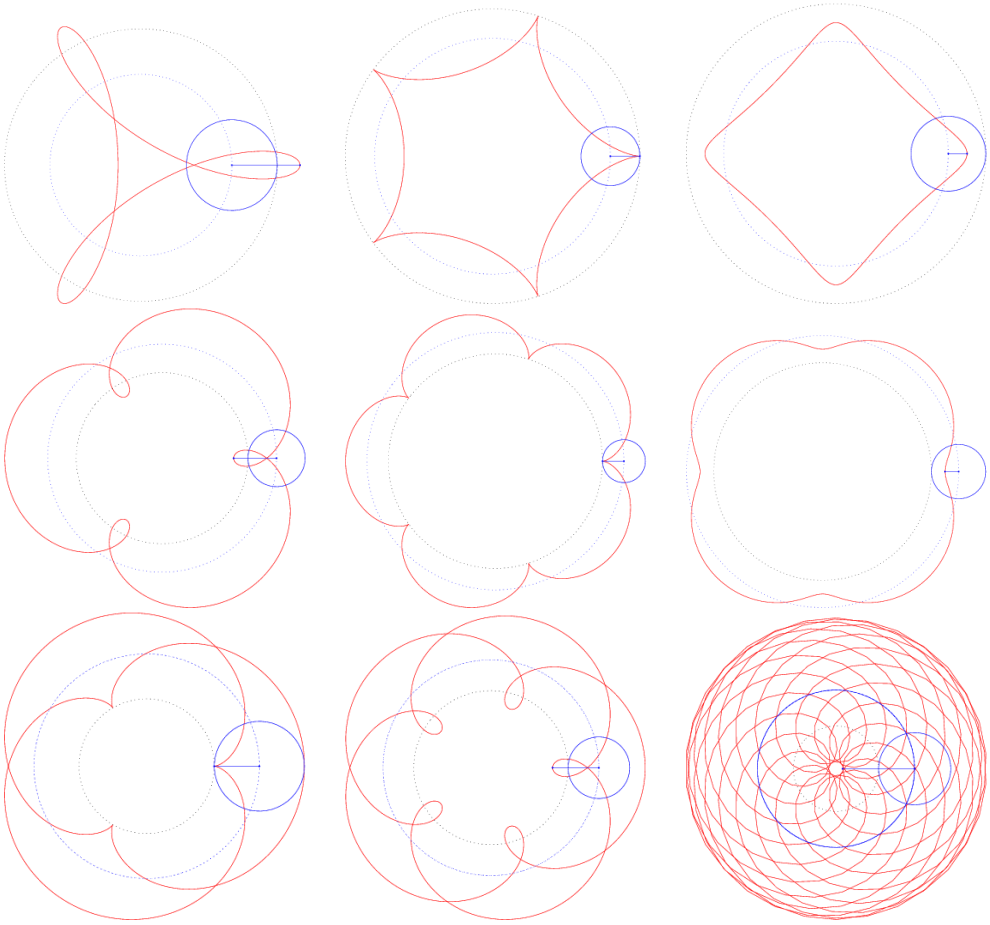
`[x,y] = epizykloide (b,p,q,l,n)` und
`[x,y] = hypozykloide(b,p,q,l,n),`

wobei hier `l` für λ steht. Mit `n` wird die Länge der Vektoren `x` und `y` angegeben.

Die Minimalversion der Funktionen muß nun ϕ (`linspace`), a, und damit dann x und y berechnen.

Weiters sollte man noch die Anzahl der Inputvariablen überprüfen und Defaultwerte setzen, $b = 1$, $p = 3$, $q = 1$, $\lambda = 1$ und $n = 500$. Zusätzlich sollte man die Zulässigkeit der Werte für die Inputvariablen überprüfen (`nargin`) und gegebenfalls Fehlermitteilungen (`error`) schreiben. Man kann auch durch Anwendung des Befehls `floor` sicherstellen, dass p und q als ganzzahlige Werte verwendet werden. Damit stellt man sicher, dass sich die Kurve jeweils am Ende schließt.

Einige typische Beispiele können Sie hier sehen:



4. Schreiben Sie ein kleines Skript `scinttest.m` das die Integrale

$$A_1(a) = \int_0^a dx\ y_1(x, a) \tag{12.52}$$

mit $y_1(x, a)$ aus 12.45 und

$$A_2(a) = \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} dxdy\ z(x, y, a) \tag{12.53}$$

mit $z(x, y, a)$ aus 12.46 berechnet.

Verwenden Sie dazu die MATLAB-Funktionen `quadl` und `dblquad`. Bedienen Sie sich der MATLAB-Hilfe bzw. des Kapitels 7.1.8 um herauszufinden, wie diese Integrationsroutinen funktionieren. Überlegen Sie auch, ob Sie im zweiten Fall `peaks3` oder `peaks3a` verwenden müssen.

Geben Sie die berechneten Größen von Fläche und Volumen mit `disp` formatiert aus.

12.14 Gaußfunktion

Ziel: Bei der vorliegenden Übung wird nochmals die Erstellung von Funktionen geübt. Darüber hinaus wird hier auch Wert auf die graphische Darstellung in zwei und drei Dimensionen gelegt. Abzugeben sind die Unterprogramme `gauss.m`, `gauss1d.m`, `gauss2d.m` und `gausstest.m`.

Voraussetzung: Voraussetzung für die Übung ist das Kapitel über Programmeinheiten 7 und über Graphik 11. Als kleine Einführung zum Plotten können die Files `htplot2d.m` und `htplot2da.m` der verwendet werden.

In der Wahrscheinlichkeitsrechnung und auch in der Physik hat die Normalverteilung bzw. Gaußverteilung eine große Bedeutung. Sie ist definiert durch

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{(x-x_0)^2}{2\sigma^2}\right\} , \tag{12.54}$$

wobei x_0 und σ die Parameter der Verteilung sind. In x_0 liegt sowohl das Maximum als auch das Symmetriezentrum, und σ ist der Abstand von diesem Zentrum zu den Wendepunkten. Wie für jede Wahrscheinlichkeitsverteilung gilt

$$\int_{-\infty}^{\infty} g(x)dx = 1 . \tag{12.55}$$

In der Physik verwendet man häufig auch eine Summation über mehrere Gaußfunktionen mit jeweils unterschiedlichen Parametern. Dies hat den Sinn, dass man gleichzeitig mehrere Maxima darstellen kann. Allgemein kann man die Funktion dann definieren als

$$g(x) = \sum_{k=1}^{n} \frac{f_k}{\sqrt{2\pi}\sigma_k} \exp\left\{-\frac{(x-x_{0k})^2}{2\sigma_k^2}\right\} + g_0 , \tag{12.56}$$

wobei x_{0k} und σ_k die gleiche Bedeutung wie vorher haben. Der Parameter f_k ist ein Multiplikator, um Peaks unterschiedlicher Höhe darstellen zu können. Die Summation erfolgt über alle n Werte der Parameter x_{0k}, σ_k und f_k. Zusätzlich kann die ganze Funktion noch um den Wert g_0 vertikal verschoben sein. Für $g_0 = 0$ gilt natürlich

$$\int_{-\infty}^{\infty} g(x)dx = \sum_{k=1}^{n} f_k . \tag{12.57}$$

In zwei Dimensionen kann die Gaußverteilung folgendermaßen definiert werden

$$g(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left\{-\frac{(x-x_0)^2}{2\sigma_x^2} - \frac{(y-y_0)^2}{2\sigma_y^2}\right\} , \tag{12.58}$$

wobei nun x_0, y_0, σ_x und σ_y die Parameter der Verteilung sind. Auch hier gilt die Normierungsbedingung

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)dxdy = 1 \; . \tag{12.59}$$

Die Summation über mehrere Peaks kann natürlich auch hier erfolgen,

$$g(x,y) = \sum_{k=1}^{n} \frac{f_k}{2\pi\sigma_{xk}\sigma_{yk}} \exp\left\{-\frac{(x-x_{0k})^2}{2\sigma_{xk}^2} - \frac{(y-y_{0k})^2}{2\sigma_{yk}^2}\right\} + g_0 \; , \tag{12.60}$$

wobei die Parameter die analoge Bedeutung wie in 12.56 haben. Für $g_0 = 0$ gilt natürlich

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)dxdy = \sum_{k=1}^{n} f_k \; . \tag{12.61}$$

1. Schreiben Sie nun eine Funktion `gauss1d.m`, welche die Formel 12.56 für beliebige numerische Arrays von x auswertet. Die Ausgabegröße sollte ein Feld g sein, das die gleiche Größe und Dimension wie x hat. Die Parameter x_{0k}, σ_k und f_k können ebenfalls in numerischen Arrays gespeichert sein, welche die gleiche Größe haben müssen. Der Parameter g_0 muss natürlich ein Skalar sein.

   - Die Deklaration der Funktion sollte so aussehen.
     ```
     function [g] = gauss1d(x,x0,s,f,g0)
     ```
     `x0` enthält alle x_{0k}-Werte, `s` alle σ_k-Werte, und `f` alle f_k-Werte.
   - Setzen Sie die Defaultwerte $x_0 = [-1, 1]$, $\sigma = [0.5, 1]$, $f = [1, 0.5]$ und $g_0 = 0$ (`nargin`).
   - Überprüfen Sie die Eingabegrößen.
   - Initialisieren Sie g als Feld mit lauter Nullen in der Größe von x.
   - Bilden Sie die Summe über alle Werte ($k = 1 \ldots n$) der Parameter mit Hilfe einer einzigen `for`-Schleife. Vergessen Sie dabei nicht, dass x ein Array sein kann.
   - Probieren Sie die Funktion mit Hilfe des Skripts `gausstest.m` aus und machen Sie einen Plot
     ```
     plot(x,g)
     ```
     Wählen Sie den x-Vektor in einem vernünftigen Bereich, damit die Funktion $g(x)$ sinnvoll dargestellt wird (`linspace`). Beschriften Sie den Plot mit Achsenbeschriftung und Titel.

2. Schreiben Sie nun eine Funktion `gauss2d.m`, welche die Formel 12.60 auswertet. Die Inputgrößen x and y können entweder gleich große Arrays sein oder eine der beiden ist ein Array und die andere ein Skalar. Im ersten Fall (gleich groß) wird die Funktion für alle Punkte (x_i, y_i) berechnet, im zweiten Fall (ein

Skalar) für alle Punkte (x_i, y_1) oder (x_1, y_i). Die Ausgabegröße sollte ein Feld g sein, das die gleiche Größe und Dimension wie das Array x oder y hat. (Aufpassen, da x oder y auch ein Skalar sein können!) Die Parameter x_{0k}, σ_{xk}, y_{0k}, σ_{yk} und f_k können ebenfalls in numerischen Arrays gespeichert sein, welche die gleiche Größe haben müssen. Der Parameter g_0 muss natürlich wieder ein Skalar sein.

- Die Deklaration der Funktion sollte so aussehen.
  `function [g] = gauss2d(x,y,x0,sx,y0,sy,f,g0)`
- Setzen Sie die Defaultwerte für zwei Peaks Ihrer Wahl.
- Überprüfen Sie die Eingabegrößen.
- Initialisieren Sie g als Feld mit lauter Nullen in der Größe von x oder y.
- Bilden Sie nun die Summe in 12.60 mit einer `for`-Schleife über alle Werte $k = 1 \ldots n$.
- Probieren Sie die Funktion mit Hilfe des Skripts `gausstest.m` aus und machen Sie eine dreidimensionale Darstellung
  `surf(xx,yy,g)` oder
  `mesh(xx,yy,g)`.
  Wählen Sie den x-Vektor und den y-Vektor in einem vernünftigen Bereich und bilden Sie mit `meshgrid` Arrays für die Darstellung der Funktion.
- Beschriften Sie den Plot und probieren Sie sinnvolle Darstellungen in Subplots einer Figure. Zum Beispiel sollte man auch Konturplots ausprobieren.

3. Eine numerische Integration kann in MATLAB mit Hilfe der Funktion `quadl` durchgeführt werden. Ein Aufruf zum Auswerten des 1-D Integrals kann so aussehen,
   `flaeche = quadl('gauss',x1,x2,tol,trace,parameter)`,
   wobei hier `gauss` die Funktion ist, über die von `x1` bis `x2` integriert wird. Für die Genauigkeit `tol` und für den Wert `trace` können zum Testen leere Felder `[]` eingegeben werden. Danach folgen die Parameter der Gaußfunktion `x0`, `s`, `f` und `y0`.

   - Schreiben Sie am Ende von `gausstest.m` den Code für die Integration und geben Sie das Ergebnis formatiert aus. Überprüfen Sie es mit der analytischen Lösung 12.57. Was passiert, wenn g_0 von Null verschieden ist?
   - Probieren Sie es mit verschiedenen Parametern aus und überprüfen Sie 12.57. Bedenken Sie dabei, dass die Integrationsgrenzen weit genug entfernt von den Maxima sind, damit 12.57 näherungsweise erfüllt sein kann. Natürlich dürfen sie aber nicht $\pm\infty$ sein.
   - Machen Sie das Gleiche für die 2-dimensionale Gaußfunktion.

4. - Schreiben Sie die Funktion `gauss.m`, die auf zwei Arten aufgerufen werden kann,

`[g] = gauss(a,x)` oder
`[g] = gauss(a,x,y)`.

- Aus ihr werden je nach Anzahl der Inputparameter (2 oder 3) die 1-D oder 2-D Funktionen `gauss1d` oder `gauss2d` aufgerufen. Der Vektor `a` soll dabei alle Parameter enthalten, also
  $[x_{0k}, \sigma_k, f_k, g_0]$ in 2-D, oder
  $[x_{0k}, \sigma_{xk}, y_{0k}, \sigma_{yk}, f_k, g_0]$ in 3-D.
- Diese Vektoren haben die Länge $3n + 1$ bzw. $5n + 1$, wobei n die Anzahl der Peaks ist. Vor dem Aufruf von `gauss1d` oder `gauss2d` müssen Sie auf die dort benötigten Inputparameter aufgespalten werden. Bestimmen Sie daher zuerst die Länge von `a`, daraus dann `n`, und damit sollte die Aufspaltung klar sein. Sollte die Länge von `a` nicht stimmen, soll eine `error`-Meldung ausgegeben werden.
- Testen Sie die neue Routine im Skript `gausstest.m` mit zwei möglichst sinnvollen Aufrufen.

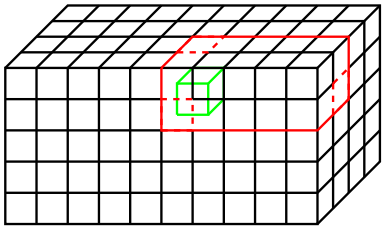
Geben Sie `gauss.m`, `gauss1d.m`, `gauss2d.m` und das Skript `gausstest.m` ab. Verwenden Sie dabei wieder `!uebungsabgabe`.

12.15 Ein industrielles Stapelproblem

Ziel: Der Umgang mit Funktionen soll wiederholt werden.

Vorraussetzung: Programierung von Funktionen, Hantieren mit Feldern. Der Befehl `hist` steht zur Verfügung, um Histogramme zu berechnen. Der Befehl `rand` erzeugt gleichverteilte Zufallszahlen zwischen 0 und 1.

1. **Ein Kistenstapelproblem:**
   Kisten (Quader) gleicher Größe sind so gestapelt, dass ein Quader entsteht (Größe: $m \times n \times l$ Kisten).

   In der Abbildung ist $m = 10$, $n = 4$, $l = 5$. Gegeben sind die Koordinaten (m_0, n_0, l_0) einer Kiste (in unserem Beispiel $m_0 = 6$, $n_0 = 2$, $l_0 = 4$). Gefragt ist nun die Mindestanzahl anz der Kisten, die ausgehend von einer Ecke entfernt werden müssen, um zur gegebenen Kiste zu gelangen (im Beispiel ist das der rot umrandete Quader der Größe $n_r \times m_r \times l_r = 5 \times 2 \times 2 = 20$).
   Schreiben Sie eine Funktion:

   [anz, gr] = remcube(dim,pos)

   Eingabeparameter:

`dim = [m, n, l]`:	ein Vektor mit den Dimensionen des Stapels
`pos = [m0, n0, l0]`:	ein Vektor mit den Koordinaten der zu entfernenden Kiste

   Ausgabeparameter:

`anz`:	Anzahl der zu entfernenden Kisten
`gr = [mr, nr, lr]`:	ein Vektor mit den Dimensionen des zu entfernenden Stapels

2. **Histogramm:**
   Schreiben Sie ein Programm, das eine Anzahl N von Kistenpositionen pos erzeugt (mit dem Befehl `rand`). Mittels des Programms `remcub` von oben können Sie zu jeder Position die Anzahl der Kisten berechnen, die man mindestens (von einem Eck ausgehend) entfernen muss, um zur gegebenen Position vorzudringen. Speichern Sie diese Zahlen in einem Vektor anz. Stellen Sie diesen Vektor mit Hilfe des `hist`-Befehls in Form eines Histogramms dar.

anz = randcube(dim,N,hanz)

Eingabeparameter:

`dim = [m, n, l]`:	ein Vektor mit den Dimensionen des Stapels
`N`:	die Anzahl der zufälligen Positionen
`hanz`:	Anzahl der Bins für die Histogrammfunktion

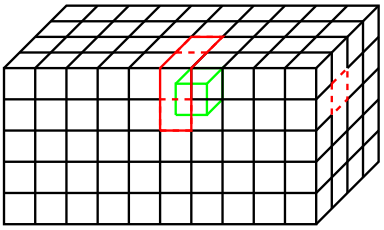
Ausgabeparameter:

`anz`:	Vektor mit der jeweiligen Anzahl zu entfernender Kisten

3. **Variationen: (freiwillig)**

   Sie können das Problem noch um folgende Fragestellungen erweitern:

   (a) **Schichten statt Quader**
   Erweitern Sie Ihre Funktionen derart, dass sie, gesteuert über einen zusätzlichen Eingabeparameter how, folgendes bewerkstelligen:
   Anstatt ausgehend von einer Ecke einen ganzen Quader abzutragen, kann man nur eine Schicht herausheben. Bestimmen Sie die Dimensionen der kleinst möglichen Schicht und die Anzahl der darin enthaltenen Kisten.

   Im Beispiel ist das die rot umrandete Schicht mit den Dimensionen $n_r \times m_r \times l_r = 1 \times 2 \times 2 = 4$.

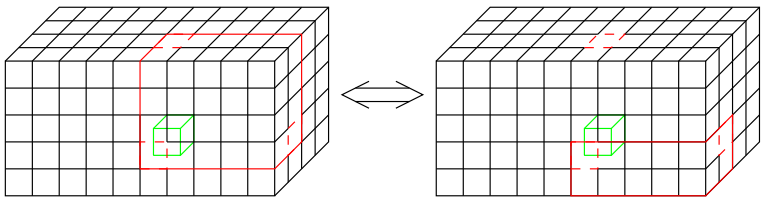
   [anz, gr] = remcube(dim,pos,how)

   zusätzlicher Eingabeparameter:

`how`:	`'c'`	für Quader herausheben (cube)
	`'s'`	für Schicht herausheben (slice)

   Sie sollten auch die Funktion **randcube** um den Parameter how erweitern.

   (b) **Quader bzw. Schichten auch nach unten entfernen**
   Erweitern Sie Ihre Funktionen derart, dass sie, gesteuert über einen zusätzlichen Eingabeparameter dir, folgendes bewerkstelligen:
   Anstatt die Quader bzw. Schichten immer nur nach oben entfernen zu dürfen, kann man auch erlauben, sie nach unten zu entfernen.



Im Beispiel wird anstatt, wie im linken Bild, der obere Quader, der untere Quader (rechtes Bild) entfernt, da er kleiner ist.

[anz, gr] = remcube(dim,pos,how,dir)

zusätzlicher Eingabeparameter:

`dir`:	`'u'`	für nur nach oben (up)
	`'a'`	für alle Richtungen erlaubt (all)

Sie sollten auch die Funktion **randcube** um den Parameter dir erweitern.

Kapitel 13

Nachlese - Was soll ich können?

13.1 Basis Syntax in MATLAB

13.1.1 Fragen

1. Was unterscheidet ein MATLAB-Skript und eine MATLAB-Funktion?
2. Welche erste Zeile muss eine MATLAB-Funktion enthalten?
3. Wie starte ich den Editor und die Online-Hilfe?
4. Wie bekomme ich direkt im MATLAB-Command-Fenster Hilfe zum Befehl `input`?
5. Wie funktioniert unter Linux "copy and paste"?
6. Unter der Voraussetzung, dass ich ein Programm im File `test.m` gespeichert habe, wie kann ich es dann in MATLAB ausführen?
7. Was ist bei den folgenden Befehlen **falsch**? Voraussetzung ist, dass die skalaren Variablen `x`, `a`, `b`, `c` , `d` bereits definiert sind.

```
y    = 3x + a
y    = 5 + sin x
y    = a exp(-(b^2 - c^2)*x^2
y(x) = a*sin(x)
y    = b * sin[x] + c * cos[a*x]
y    = a * sqrt( {b^2 + c^2}*x )
y    = b * arcsin(a*x)
y    = a * (x + b * [x + c * {x + d}])
```

8. Was bewirkt der Unterschied in den folgenden Zeilen?

```
y = x^2
y = x^2,
y = x^2;
y = x^2 % Quadrat
```

9. Was ist in der Programmzeile für folgende mathematische Funktion **falsch**?

$$y(x) = \frac{x^2}{x + a}$$

```
y = x^2 / x + a
```

10. Mit welchem Befehl kann man den Benutzer eines Programms auffordern einen Wert einzugeben? Z.B.: "Geben Sie a ein: ". Der eingegebene Wert soll dabei der Variablen `a` zugewiesen werden.

11. Mit welchem Befehl kann ich eine Zahl oder eine Zeichenkette am Schirm ausgeben?

12. Wie kann ich mehrere Zeichenketten (`s1, s2, s3`) aneinanderfügen?

13. Wie kann ich eine Zahl (Datentyp: double) in eine Zeichenkette gleichen Inhalts (Datentyp: char) umwandeln?

14. Was ist **falsch** an folgenden Zeilen, wenn `s1, s2` Zeichenketten und `x` eine Zahl ist?

```
disp(s1,s2)
disp([s1,s2)]
disp([s1,x,s2])
disp([s1;s2])
```

15. Was ist der Unterschied zwischen den beiden Zeilen?

```
y = [1,2,3]
y = [1;2;3]
```

16. Wie kann ich in einem MATLAB-File Kommentare einfügen, die bei Verwendung des Befehls `help` sichtbar sind?

13.1.2 Antworten

1. Was unterscheidet ein MATLAB-Skript und eine MATLAB-Funktion?

   Ein MATLAB-Skript ist eine Aneinanderreihung von Befehlen (Hauptprogramm). Eine MATLAB-Funktion wird mit Ein- und Ausgabeparametern gestartet und braucht eine Deklarationszeile (siehe 2).

2. Welche erste Zeile muss eine MATLAB-Funktion enthalten?

   `function out = func1(in1,in2,in3)` oder
   `function [out1,out2] = func1(in1,in2,in3)`

3. Wie starte ich den Editor und die Online-Hilfe?

   Mit den Befehlen `edit` und `helpdesk`.

4. Wie bekomme ich direkt im MATLAB-Command-Fenster Hilfe zum Befehl `input`?

   Mit dem Befehl `help input`. Der Befehl `lookfor input` listet alle Befehle in deren help-Text `input` vorkommt.

5. Wie funktioniert unter Linux "copy and paste"?

   Einfärben mit der linken Maustaste (copy) und einfügen mit der mittleren Maustaste (paste).

6. Unter der Voraussetzung, dass ich ein Programm im File `test.m` gespeichert habe, wie kann ich es dann in MATLAB ausführen?

   Durch Eingabe des Befehls `test`.

7. Was ist bei den folgenden Befehlen **falsch**? Voraussetzung ist, dass die skalaren Variablen `x`, `a`, `b`, `c` , `c` bereits definiert sind.

```
y = 3*x + a
y = 5 + sin(x)
y = a*exp(-(b^2 - c^2)*x^2)
y = a*sin(x)     Argument x in y(x) entfernt
y = b * sin(x) + c * cos(a*x)
y = a * sqrt( (b^2 + c^2)*x )
y = b * asin(a*x)
y = a * (x + b * (x + c * (x + d)))
```

8. Was bewirkt der Unterschied in den folgenden Zeilen?

```
y = x^2      Ausgabe am Schirm
y = x^2,     Ausgabe am Schirm
y = x^2;     Keine Ausgabe am Schirm
```

```
y = x^2 % Quadrat
```
Ausgabe am Schirm ohne Kommentar

9. Was ist in der Programmzeile für folgende mathematische Funktion **falsch**?

$$y(x) = \frac{x^2}{x + a}$$

```
y = x^2 /(x + a)
```

10. Mit welchem Befehl kann man den Benutzer eines Programms auffordern einen Wert einzugeben? Z.B.: "Geben Sie a ein: ". Der eingegebene Wert soll dabei der Variablen `a` zugewiesen werden.

```
a = input('Geben Sie a ein: ');
```

11. Mit welchem Befehl kann ich eine Zahl oder eine Zeichenkette am Schirm ausgeben?

```
disp
```

12. Wie kann ich mehrere Zeichenketten (`s1, s2, s3`) aneinanderfügen?

```
s = [s1,s2,s3]
```

13. Wie kann ich eine Zahl (Datentyp: double) in eine Zeichenkette gleichen Inhalts (Datentyp: char) umwandeln?

```
s=num2str(d)
```
`s=num2str(d,n)` mit `n` Anzahl der Digits

14. Was ist **falsch** an folgenden Zeilen, wenn `s1, s2` Zeichenketten und `x` eine Zahl ist?

```
disp([s1,s2])
disp([s1,s2])
disp([s1,num2str(x),s2])
disp([s1,s2])
```

15. Was ist der Unterschied zwischen den beiden Zeilen?

```
y = [1,2,3]
```
Zeilenvektor
```
y = [1;2;3]
```
Spaltenvektor

16. Wie kann ich in einem MATLAB-File Kommentare einfügen, die bei Verwendung des Befehls `help` sichtbar sind?

Durch Einfügen von zusammenhängenden Kommentarzeilen am Anfang des Files (MATLAB-Skript) bzw. nach der Deklarationszeile (MATLAB-Funktion). Die erste Leer- oder Kommandozeile beendet diesen Block. Weiter Kommentare werden bei Verwendung von `help` nicht angezeigt.

13.2 Reguläre Polyeder, Kegelschnitte

13.2.1 Fragen

1. Was sind richtige und falsche Namen von Variablen?
   `a12 1a a-3 a_12 a(3) _bb maxi a.b`
2. Wie kann man feststellen, welche Variablen im MATLAB-Arbeitsbereich bereits definiert sind?
3. Wie kann man feststellen, ob ein Name bereits als Variable oder Funktion existiert?
4. Warum sollte man `i`, `j` oder z.B. `max` nicht als Variablennamen verwenden?
5. Wie erzeugt man einen Vektor mit 20 Zahlen, die equidistant zwischen 0 und 2 verteilt sind.
6. Gegeben ist eine Zeichenkette `st='Sinus'`. Welche Ausgabe erzeugen die Befehle `lower(st)`, `upper(st)`, bzw. `lower(st(1))`?
7. Was ist eine Zeichenkette bzw. warum kann man mit einem Index darauf zugreifen?
8. Wie muss man `function [x1,x2]=test(a,b)` aufrufen, damit die Ergebnisse für `a=1` und `b=2` den Variablen `m1` und `n1` zugewiesen werden?
9. Sind nach diesem Aufruf die Variablen `x1` und `x2` im MATLAB-Workspace bekannt?
10. Warum macht nach obiger Deklaration der Befehl `a=input('a')` keinen Sinn?
11. Wie muss ich obige Funktion aufrufen, wenn ich für `a` und `b` Vektoren übergeben will?
12. Wie kann man Variablen löschen?
13. Welche Befehle sind richtig und welche falsch (warum)?

`[1,2,3]*[2,3,4]`		
`[1,2,3]/5`		
`[1,2,3]^2`		
`[1,2;3,4]^2`		
`[1,2,3]*[1;2;3]`		
`1/[1,2,3]`		
`[1,2,3].^(1/2)`		
`[1,2,3].*[1,2,3,4]`		
`1./[1,2,3]`		

13.2.2 Antworten

1. Was sind richtige und falsche Namen von Variablen?
   `a12` `1a` `a-3` `a_12` `a(3)` `_bb` `maxi` `a.b`
   Korrekte Variablennamen müssen mit einem Buchstaben beginnen und dürfen ausser _ keine Sonderzeichen enthalten.

2. Wie kann man feststellen, welche Variablen im MATLAB-Arbeitsbereich bereits definiert sind?
   Mit den Befehlen `who` bzw. `whos`.

3. Wie kann man feststellen, ob ein Name bereits als Variable oder Funktion existiert?
   Mit dem Befehl `exist`.

4. Warum sollte man `i`, `j` oder z.B. `max` nicht als Variablennamen verwenden?
   Da sie intern in MATLAB verwendete Variablen bzw. Funktionen sind.

5. Wie erzeugt man einen Vektor mit 20 Zahlen, die equidistant zwischen 0 und 2 verteilt sind.
   `v=linspace(0,2,20)`

6. Gegeben ist eine Zeichenkette `st='Sinus'`. Welche Ausgabe erzeugen die Befehle `lower(st)`, `upper(st)`, bzw. `lower(st(1))`?
   Liefert `sinus`, `SINUS`, `s`.

7. Was ist eine Zeichenkette bzw. warum kann man mit einem Index darauf zugreifen?
   Eine Zeichenkette ist ein Array (Vektor) von Zeichen.

8. Wie muss man `function [x1,x2]=test(a,b)` aufrufen, damit die Ergebnisse für `a=1` und `b=2` den Variablen `m1` und `n1` zugewiesen werden?
   `[m1,n1]=test(1,2)`

9. Sind nach diesem Aufruf die Variablen `x1` und `x2` im MATLAB-Workspace bekannt?
   Nein! Funktionen arbeiten in einem eigenen Workspace.

10. Warum macht nach obiger Deklaration der Befehl `a=input('a')` keinen Sinn?
    `a` ist nach dem Aufruf bereits bekannt und muss nicht abgefragt werden.

11. Wie muss ich obige Funktion aufrufen, wenn ich für `a` und `b` Vektoren übergeben will?
    `[m1,n1]=test([1,2,3],[2,2,2])`

12. Wie kann man Variablen löschen?
    Mit dem Befehl `clear`.

13. Welche Befehle sind richtig und welche falsch (warum)?

`[1,2,3]*[2,3,4]`	falsch	Matrizenmultiplikation
`[1,2,3]/5`	richtig	jedes Element
`[1,2,3]^2`	falsch	Matrizenmultiplikation
`[1,2;3,4]^2`	richtig	quadratische Matrix
`[1,2,3]*[1;2;3]`	richtig	Matrizenmultiplikation = 14
`1/[1,2,3]`	falsch	Division durch Vektor
`[1,2,3].^(1/2)`	richtig	elementweise
`[1,2,3].*[1,2,3,4]`	falsch	unterschiedliche Länge
`1./[1,2,3]`	richtig	elementweise

Kapitel 14

Voraussetzungen zum positiven Abschluss der Lehrveranstaltung Applikationssoftware und Programmierung

In den folgenden Zeilen ist kurz zusammengestellt, was Sie können müssen, um die Applikationssoftware positiv abzuschließen. Beachten Sie bitte, dass Sie in der Lage sein sollten, bei der Abschlussübung die gestellten Aufgaben, die unten angeführten Problemekreise umfassen, *selbständig* zu lösen. Ihr Betreuer wird Ihnen während der Prüfung natürlich nicht helfen können. Machen Sie sich deshalb mit der *Verwendung der Matlab-Online-Hilfe* vertraut! Weiters können Sie die von Ihnen erarbeiteten Übungsbeispiele, das Skriptum sowie sonstige Matlab-Bücher während der Prüfung verwenden.

Unbedingt notwendig ist die rechtzeitige Abgabe aller Übungsbeispiele. Rechtzeitig bedeutet, dass die Abgabe so erfolgen soll, dass die Beispiele noch korrigiert werden können!

14.1 Notwendige Grundlagen von Matlab

- Verwendung der Matlab-Online-Hilfe
- Umgang mit Vektoren und Feldern
  - Indizierung: Zeilen, Spalten
  - Doppelpunktnotation
  - logische Indizierung

  - Datentypen in Matlab
  - Bestimmung der Dimensionen von Feldern (size, length)
- Einlesen und Abspeichern von Daten mit den Befehlen load und save.
- Übersetzen mathemetischer Ausdrücke und Formeln in korrekte Matlab Befehle
- Ausgabe von Nachrichten und Ergebnissen im Textfenster, Einlesen von Daten von der Tastatur
- Vektorisierung
  - Verwendung der Punkt-Notation,
  - Arithmetische Operatoren, Vergleichsoperatoren
- Lösen linearer Gleichungssysteme; Transponieren einer Matrix, Verwendung des \-Operator
- Unterschied: Elementweise Operationen – Matrixoperationen im Sinne der Linearen Algebra
- Verwendung von Matlab-eigenen Routinen, wie sum, quadl, polyval, polyfit
- Steuerelemente zur Kontrolle des Programmflusses: if, for, while, case
- Erstellen einfacher Funktionen in eigenen Matlab-Dateien
  - Verwendung von Eingabe– und Ausgabeparametern
  - Vektorisierung der Funktionsberechnung; d.h. anstatt nur einen Funktionswert $f(x)$ für ein bestimmtes x zurückzuliefern, müssen Ihre Funktionen auch mit ganzen Vektoren $\vec{x}$ von Argumenten zurecht kommen.
  - Steuerung der Auswertung durch logische Felder
  - Abfrage der korrekten Parameterübergabe (Anzahl, Typ)
  - Ausgabe von Fehlermeldungen bzw. Verwendung von Default– Parametern
- Verwendung von Inline-Funktionen mit Parametern
- Lineares und Nicht-lineares Fitten von Funktionen
- Graphisches Darstellen von Daten und Funktionen
  - Darstellung von Datenpunkten mit verschiedenen Symbolen
  - Darstellung von Kurven und Funktionen
  - Achsenbeschriftung, Legende, Überschriften

– Verwendung der Handles zum Zugriff auf Grafik-Objekte (gcf, gca, gco, set, get)
– Verändern der Textgröße der Beschriftungen

Kapitel 15

Anhang

15.1 Der Editor EMACS

Als eindeutig bester Editor für MATLAB hat sich der Editor EMACS erwiesen. Der bereitgestellte MATLAB-Mode bietet mit Hilfe von EMACSLINK eine Verbindung zwischen MATLAB und EMACS, erlaubt Syntax-Highlighting und Kommando-Ergänzung. Im Rahmen der Ausbildung bietet EMACS darüber hinaus weitere Vorteile, da dieser Editor ähnliche Unterstützung für andere Sprachen, wie z.B. C, C++, FORTRAN, PYTHON, oder LATEX bietet.

Die beste Verwendung von EMACS mit der Programmiersprache MATLAB ist der Aufruf aus MATLAB. Vorausgesetzt EMACSLINK ist in den Preferenzen eingestellt, startet man den Editor mit folgendem Befehl.

```
edit        % opens file untitled.m
edit file   % opens file or file.m (if file does not exist)
edit file.m
```

In Kontrast zum eingebauten MATLAB-Editor startet EMACS im Buffer `*scratch*` falls der gewünschte File nicht existiert. Man wird dann beim ersten Speichern aufgefordert den Filenamen anzugeben.

Ungeübte Benutzer können EMACS mit Hilfe der Maus und mit Menüeinträgen bedienen. Die wirkliche Stärke des Editors zeigt sich aber, wenn man zumindest die wichtigsten Tastenkombinationen verwendet. Diese werden in der Folge vorgestellt.

15.1.1 Buffer, Frame und Window

EMACS stellt für jeden geöffneten File einen Buffer bereit, der den gesamten Inhalt dieses Files beinhaltet. Außerdem können Buffer noch für andere Aufgaben verwendet werden:

`*scratch*`: Ein Bereich für Notizen, die nicht automatisch zum Speichern vorgesehen sind.

`*messages*`: Nachrichten von EMACS.

andere: Z.B. für die Ausgabe von Programmen, die aus EMACS aufgerufen werden.

Die Darstellung der Buffer erfolgt in sogenannten Frames (eigene Fenster am Bildschirm), die unter Umständen in mehrere Windows (Bereiche in einem Frame) aufgeteilt sein können. Der Inhalt eines Buffers kann nun in einem Window dargestellt werden, wobei der Inhalt eines Buffers auch in mehreren Windows vorhanden sein kann. Dies hat den Vorteil, dass man verschiedene Bereiche eines Files mehr oder weniger gleichzeitig bearbeiten kann. Man kann damit z.B. recht einfach Teile vom Anfang eines langen Files in einem weit entfernten Bereich einfügen.

Im unteren Bereich des Frames gibt es noch den sogenannten Minibuffer in dem EMACS-Kommandos ausgeführt werden. Dorthin springt z.B. der Editor, wenn ein File geöffnete wird.

15.1.2 Tastenkombinationen

Bei der Beschreibung von Tastenkombinationen werden folgende Abkürzungen verwendet:

`C-x`: Control-Key (`Strg` oder `Ctrl`) gleichzeitig gedrückt mit einer weiteren Taste (hier `x`).

`C-x s`: Control-Key gleichzeitig gedrückt mit einer weiteren Taste (hier `x`); Nach dem Loslasen drücken einer weiteren Taste (hier `s`).

`M-x`: Meta-Key (`Alt`) gleichzeitig gedrückt mit einer weiteren Taste.

`C-M-x`: Control- und Meta-Key gleichzeitig gedrückt mit einer weiteren Taste.

`ESC`: Escape-Key (`Esc`) gleichzeitig gedrückt mit einer weiteren Taste.

`RET`: Return- oder Eingabetaste.

`SPC`: Space oder Leerzeichen.

`TAB`: Tabulator.

`DEL`: Delete- oder Entferne-Taste.

`BSP`: Backspace- oder Lösch-Taste.

15.1.2.1 Files, Buffers und Windows

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-z	Suspend	C-x C-c	Beendet
C-x C-s	Speichert File	C-x s	Speichert alle Files
C-x C-w	Speichert File als	C-x C-v	Ersetzt File
C-x C-f	Öffnet File	C-x i	Einfügen eines Files
C-x b	Anderer Buffer	C-x k	Killt (entfernt) Buffer
C-x C-b	Liste aller Buffer		

Eine Eigenart von EMACS ist die Tatsache, dass man für das Öffnen eines neuen, nicht existierenden Files ebenfalls C-x C-f für das Anlegen verwendet. Man wird dann im Minibuffer zum Eingeben des Filenamens aufgefordert.

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-x 2	Split Window	C-x 5 2	Neuer Frame
C-x 0	Entfernt Window	C-x 5 0	Entfernt Frame
C-x 1	Entfernt andere Windows	C-x 3	Split Window seitlich
C-M-v	Scrollt anderes Fenster	C-x ^	Macht Window größer
C-x {	Macht Window schmäler	C-x }	Macht Window breiter
C-x o	Cursor in anderes Window	C-x 5 o	Cursor in anderen Frame
C-x 4 b	Auswahl Buffer in anderem Window	C-x 5 b	Auswahl Buffer in anderem Frame
C-x 4 C-o	Buffer in anderem Window	C-x 5 C-o	Buffer in anderem Frame
C-x 4 f	Öffne File in anderem Window	C-x 5 f	Öffne File in anderem Frame

15.1.2.2 Navigation durch den Buffer

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-b	Buchstabe rückwärts	C-f	Buchstabe vorwärts
M-b	Wort rückwärts	M-f	Wort vorwärts
C-p	Zeile rückwärts	C-n	Zeile vorwärts
C-a	Zeilenanfang	C-e	Zeilenende
M-a	Satz rückwärts	M-e	Satz vorwärts
M-{	Paragraph rückwärts	M-}	Paragraph vorwärts
M-x [	Seite rückwärts	M-e]	Seite vorwärts
C-M-a	Funktion rückwärts	C-M-e	Funktion vorwärts
M-<	Bufferanfang	M->	Bufferende
M-v	Scroll Schirm rückwärts	C-v	Scroll Schirm vorwärts
C-x <	Scroll Schirm links	C-x >	Scroll Schirm rechts
C-u C-l	Zeile in Schirmmitte		

15.1.2.3 Markieren, Kopieren und Löschen

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-SPC	Setze Markierung	C-x C-x	Tausche Markierung und Cursor
M-h	Markiere Paragraph	C-x C-p	Markiere Seite
C-M-h	Markiere Funktion	C-x h	Markiere Buffer
BSP	Entferne Zeichen rückwärts	DEL	Entferne Zeichen vorwärts
M-DEL	Lösche Wort rückwärts	M-d	Lösche Wort vorwärts
M-0 C-k	Lösche Zeile rückwärts	C-k	Lösche Zeile vorwärts
C-x DEL	Lösche Satz rückwärts	M-k	Lösche Satz vorwärts
C-w	Lösche Region	M-w	Kopiere Region
C-y	Einfügen	M-y	Tauscht letzte Einfügung gegen vorherige

Als Region bezeichnet man dabei einen markierten Bereich. Diese kann sowohl mit Hilfe der Tastatur, als auch mit Hilfe der Maus (linke Taste) markiert werden. Das Einfügen erfolgt ebenfalls mit Hilfe einer Tastenkombination C-y, bzw. mit Hilfe der mittleren Maustaste.

15.1.2.4 Formatierung, Änderung, Tausch

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
TAB	Einrückung Zeile	C-M-\	Einrückung Region
C-x TAB	Einrückung Region erzwungen		
C-o	Neue Zeile	C-M-o	Restliche Zeile nach unten
C-x C-o	Entferne leere Zeilen	M-^	Verbinde Zeile mit voriger
M-\	Entferne leere Zeichen	M-SPC	Exakt ein Leerzeichen
M-q	Fülle Paragraph	C-x f	Setzte Füll Spalte
M-u	Großbuchstaben Wort (upper)	M-l	Kleinbuchstaben Wort (lower)
M-c	Erster Buchstaben groß (capitalize)		
C-x C-u	Großbuchstaben Region (upper)	C-x C-l	Kleinbuchstaben Region (lower)
C-t	Austausch Zeichen	M-t	Austausch Worte
C-x C-t	Austausch Zeilen		

15.1.2.5 Suchen und Ersetzen

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-s	Suche vorwärts	C-r	Suche rückwärts
M-p	Wähle vorherigen Suchstring	M-n	Wähle nächsten Suchstring
RET	Beende Suche	C-g	Abbruch der Suche
M-%	Suche und Ersetzen (interactiv)		

Im Suchen und Ersetzen Modus M-% sind die folgenden Antworten möglich: ? Hilfe; SPC ersetze und geh weiter; , ersetze und bleibe am selben Platz; . ersetze und beende; DEL ersetze nicht und geh weiter; ! ersetze alle Weiteren; ^ zurück zum vorherigen Suchresultat; RET beende den Ersetzungsmodus; E editiere die Ersetzungszeichen.

Kapitel 16

Literatur

Es gibt eine Reihe sehr guter Bücher von MATLAB, die im Wesentlichen eine detailierte Dokumentation der Sprache und der Umgebung beinhalten. Sie liegen alle in englischer Sprache im PDF-Format vor. Auch für Anfänger geeignet sind folgende Bücher:

- Getting Started with MATLAB
- Using MATLAB
- Using MATLAB Graphics

Die folgenden Bücher sind erst für fortgeschrittene Benutzer von Interesse:

- Creating Graphical User Interfaces
- MATLAB Functions: Volume 1
- MATLAB Functions: Volume 2
- MATLAB Functions: Volume 3
- External Interfaces/API
- Application Program Interface Reference
- MAT-File Format

Außerdem gibt es eine Reihe von Büchern anderer Autoren. In [1] und [2] geht es vor allem um eine Einführung in MATLAB, wobei in [2] schon auf die MATLAB Version 6 eingegangen wird. Beide Bücher bieten eine Reihe von Beispielen und Lösungen.

In [3] geht es bereits um eine etwas fortgeschrittene Benutzung von MATLAB und in [4] wird speziell auf Graphik und graphische Benutzeroberflächen in MATLAB eingegangen.

In [5] wird speziell auf numerische Methoden eingegangen, die mit MATLAB realisiert werden, [6] behandelt mathematische Fragestellungen in MATLAB vor allem auch mit Hilfe der `symbolischen Toolbox`, [7] löst wissenschaftliche Probleme mit Hilfe von MATLAB und MAPLE.

Literaturverzeichnis

[1] R. Pratap. *Getting Started with MATLAB 5, A Quick Introduction for Scientists and Engineers*. Oxford University Press, 1999. 16

[2] C. Überhuber and S. Katzenbeisser. *MATLAB 6 Eine Einführung*. Springer, 2000. 16

[3] D. Hanselman and B. Littlefield. *Mastering MATLAB 5, A Comprehensive Tutorial and Reference*. Prentice Hall, 1998. 16

[4] P. Marchand. *Graphics and GUIs with MATLAB*. ORC, second edition, 1999. 16

[5] G. Lindfield and J. Penny. *Numerical Methods Using MATLAB*. Ellis Horwood, 1995. 16

[6] H. Benker. *Mathematik mit MATLAB, Eine Einführung für Ingenieure und Naturwissenschaflter*. Springer, 2000. 16

[7] W. Gander and J. Hřebíček. *Solving Problems in Scientific Computing Using Maple and MATLAB*. Springer, third edition, 1997. 16