Applikationssoftware und Programmierung

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

14. Februar 2003

[1]Tel.: +43(316)873-8192; Fax.: +43(316)873-8678; e-mail: kernbichler@itp.tu-graz.ac.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:

- 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 http://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 1.3.2). Dieser Raum liegt direkt neben dem Hörsaal P2 im Physikgebäude.

Derzeit sind fünf Gruppen mit jeweils maximal 13 Teilnehmern vorgesehen, da wir von ungefähr 60 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	4. März 2002	CR Physik	Kernbichler
B	Montag	18:00	19:30	4. März 2002	CR Physik	Kernbichler
C	Mittwoch	08:15	09:45	6. März 2002	CR Physik	Prüll
D	Mittwoch	17:00	18:30	6. März 2002	CR Physik	Prüll
E	Mittwoch	15:15	16:45	6. März 2002	CR Physik	Koller

Übungen haben immanenten Prüfungscharakter, das heißt, eine **Teilnahme** an den einzelnen Übungsstunden ist **verpflichtend** (siehe auch 1.2.6). Wenn Sie eine andere Lösung benötigen bzw. verhindert sind, kontaktieren Sie unbedingt den jeweiligen Übungsleiter.

1.2.4 Beginn

Durch die dienstlich bedingte Abwesenheit von Kernbichler/Prüll/Koller in der ersten Vorlesungswoche und von Kernbichler auch in der zweiten Woche haben wir uns zu folgendem Ablauf entschieden.

Datum	Beginn	Titel	Ort	Leiter
25.2.2002	14:15	Accountvergabe	HS P2 / CR Physik	Tutoren
4.3.2002	14:15	Einführung	HS P2	Prüll/Koller
4.3.2002	16:15	Einführungsübung A	CR Physik	Prüll/Koller
4.3.2002	18:00	Einführungsübung B	CR Physik	Prüll/Koller
6.3.2002	8:15	Einführungsübung C	CR Physik	Prüll
6.3.2002	17:00	Einführungsübung D	CR Physik	Prüll
6.3.2002	15:15	Einführungsübung E	CR Physik	Koller

Die Einführung und die zugehörige Übung sollen Sie mit dem Computersystem, dem Betriebssystem LINUX und der Programmumgebung von MATLAB vertraut machen.

Ab der darauffolgenden Woche finden **alle** Lehrveranstaltungen planmäßig statt.

1.2.5 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/matlabhelpdesk.html.
Der Zugang zu vielen Büchern im sogenannten "Portable Document Format - pdf" findet sich ebenfalls auf diesem Server
http://www.itp.tu-graz.ac.at/matlabhelpdesk.html.

1.2.6 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.

Von den Vortragenden und Betreuern 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.

1.2.7 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
- http://fubphpc.tu-graz.ac.at.

1.3.1 Computerzugang an der TU Graz

1.3.1.1 Subzentren

Für die Ausbildung der Studierenden in und mit EDV stellt der Zentrale Informatik Dienst (ZID) in den einzelnen Gebäudekomplexen der TU Graz Computerarbeitsplätze für die Lehre zur Verfügung. Die EDV-Ausbildungsräume sind mit insgesamt mehr als 150 Arbeitsplatzrechnern ausgestattet.

Die EDV-Ausbildungsräume können für Lehrveranstaltungen, Übungen, Seminare usw. genutzt werden. In den übrigen Zeiten stehen die Rechner den Studierenden zur freien Benutzung zur Verfügung.

Öffnungszeiten und Betreuung

Die EDV Ausbildungsräume sind während der Vorlesungszeiten in der Regel von Montag bis Freitag in der Zeit von 8:00 - 21:30 Uhr (Ausnahme Inffeldgasse 25/EG 8:00 - 19:30 Uhr) geöffnet. In den Ferien und vorlesungsfreien Zeiten gelten eingeschränkte Öffnungszeiten. Üblicherweise sind in diesen Zeiten die Subzentren Kopernikusgasse 24/3. Stock und Rechbauerstrasse 12/2. Stock geöffnet. Die aktuellen Öffnungszeiten können auch den Aushängen in den EDV Ausbildungsräumen und den Informationen des Zentralen Informatikdienstes entnommen werden.

Beim Betrieb der EDV Ausbildungsräume werden die Mitarbeiterinnen und Mitarbeiter des ZID durch Betreuerinnen bzw. Betreuer unterstützt, die täglich zu festgelegten Zeiten in den EDV Ausbildungsräumen anwesend sind. Diese sorgen für den laufenden Betrieb, betreuen die Peripherie wie Drucker und Plotter und kümmern sich um Hard- und Softwareprobleme. Eine ihrer wichtigsten Funktionen ist die Unterstützung der Studierenden, die damit eine Ansprechperson für Fragen und Probleme direkt vor Ort vorfinden. In den nicht betreuten Zeiten steht eine zentrale e-mail-Adresse **service@subedvz.tu-graz.ac.at** zur Verfügung, an die Hardware- und Softwareprobleme gemeldet und Fragen gerichtet werden können.

Benutzung der EDV-Ausbildungsräume

Es gelten die Richtlinien für die Benutzung der EDV Benutzerräume (EDV Subzentren).

Die EDV Räumlichkeiten des Zentralen Informatikdienstes sind tagsüber für Angehörige der TU Graz frei zugänglich. Für die Benutzung der Ausbildungsrechner in den EDV Ausbildungsräumen benötigen alle Studierenden einen persönlichen Benutzernamen. Dieser kann von jeder/jedem Studierenden der TU Graz mit dem zugesandten PIN-Code (siehe unten) selbst eingerichtet werden. Damit können die Ausbildungsrechner von allen Studierenden der TU-Graz - und nur von ihnen - benutzt werden.

Die Benutzernamen für Studierende ermöglichen die Verwendung der öffentlich zugänglichen Ausbildungsrechner in den EDV Ausbildungsräumen und der dort installierten Software und sind auch für die Benutzung der Rechner im Rahmen von

EDV Lehrveranstaltungen, Übungen, Seminaren usw. unbedingt nötig. Außerdem wird ein Speicherbereich am Server reserviert, auf dem Studierende eigene Daten ablegen können. Auch Internet-Dienste wie E-Mail, Usenet News, File-Transfer (ftp), remote login auf andere Rechner (telnet), Zugriff auf Informationssysteme (WWW) stehen damit allen Studierenden offen.

1.3.2 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

1.3.3 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
telnet	Terminal Kommunikation	Unverschlüsselt
scp	Secure Copy	Verschlüsselt
ftp	File Transfer	Unverschlüsselt
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 **kernbichler@itp.tu-graz.ac.at** zu erreichen.

Eine Übungsgruppe wird Herr Dipl.-Ing. Alexander Prüll betreuen. Er ist unter der Adresse **pruell@itp.tu-graz.ac.at** zu erreichen.

Außerdem gibt es für die Übungen zwei Tutoren:
Dieter Mayer, **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**, und Christian Pfaffel, erreichbar unter **pfaffel@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.

Kapitel 2

Arrays

2.1 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 2.1: 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
⋮					

2.2 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 2.1 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.

2.3 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.
```

2.4 Erzeugung von Matrizen

Arrays bzw. Matrizen können auf vielfältige Weise erzeugt werden:

- Explizite Eingabe (2.4.1).
- Erzeugung mit Hilfe der Doppelpunkt Notation (2.4.2).
- Erzeugung mit Hilfe eingebauter Funktionen (2.4.3).
- Laden von einem externen File (2.4.4).
- Selbst geschriebene Funktionen (M-files).

2.4.1 Explizite Eingabe

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

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

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.

2.4.2 Doppelpunkt Notation

Die `Doppelpunktnotation` ist eine der mächtigsten Bestandteile von MATLAB. Sie kann einerseits zur Konstruktion von Vektoren (Tab. 2.2), aber auch zum Zugriff auf Teile von Matrizen (Index, 2.6) verwendet werden.

Tabelle 2.2: 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: []
```

2.4.3 Interne Befehle zur Erzeugen von Matrizen

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

Tabelle 2.3: 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 2.4: 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).

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:

$$z = \begin{bmatrix} 1 & 2 & 3 & \dots & n \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & 3 & \dots & n \end{bmatrix}, \; s = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 2 & 2 & 2 & \dots & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m & m & m & \dots & m \end{bmatrix}. \tag{2.1}$$

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:

$$v = \begin{bmatrix} 101 & 102 & 103 & 104 & 105 & \dots \\ 201 & 202 & 203 & 204 & 205 & \dots \\ 301 & 302 & 303 & 304 & 305 & \dots \\ 401 & 402 & 403 & 404 & 405 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{2.2}$$

2.4.4 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.

2.5 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)))
```

Tabelle 2.5: 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

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

2.6 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 2.5 werden die einzelnen Regeln erläutert, und in 2.6 die Zuweisung von Werten gezeigt, und in 2.7 der Zugriff auf bestimmte Regionen gezeigt.

Tabelle 2.6: 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

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,

$$\begin{bmatrix} (1,1) & (1,2) & (1,3) & (1,4) \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & (3,2) & (3,3) & (3,4) \end{bmatrix} \equiv \begin{bmatrix} (1) & (4) & (7) & (10) \\ (2) & (5) & (8) & (11) \\ (3) & (6) & (9) & (12) \end{bmatrix} . \tag{2.3}$$

Da mit Hilfe der Doppelpunkt Notation ja eigentlich Vektoren als Indices erzeugt werden (2.4.2), ist natürlich auch folgende Schreibweise erlaubt:

Tabelle 2.7: 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)`
$\begin{matrix} 3 & 7 & 11 & 4 \end{matrix}$	$\begin{matrix} 3 \\ 7 \end{matrix}$	$\begin{matrix} 5 & 6 & 7 \end{matrix}$

- `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.

2.6.1 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. 2.8).

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.
- 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.

Tabelle 2.8: 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

- 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 3.3 und 3.2.

2.7 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)`
`cat(2,A,B)`
`cat(3,A,B)`
	`vertcat(A,B)`
	`horzcat(A,B)`
		`[A;B]`
		`[A,B]`

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

2.8 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.

2.9 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.

Kapitel 3

Operatoren

3.1 Arithmetische Operatoren

3.1.1 Arithmetische Operatoren für Skalare

Die vordefinierten Operatoren auf skalaren double-Ausdrücken sind in Tabelle 3.1 zusammengefaßt. Diese Operatoren sind eigentlich Matrixoperatoren, deren genaue Behandlung in 4 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 3.1: 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

3.1.2 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 3.2 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 3.1. 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 3.3 findet man die vordefinierten Operatoren zur komponentenweisen Verknüpfung von Feldern und Skalaren.

In 3.3 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 4.

Tabelle 3.2: 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)`	$[a_1b_1\ a_2b_2\ \ldots\ a_nb_n]$	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 3.3: 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)`	$[a_1c\ a_2c\ \ldots\ a_nc]$	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)`	$[a_1c\ a_2c\ \ldots\ a_nc]$	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 3.4: 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	$a \neq b$

3.2 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
```

3.3 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 3.5 zusammengestellt.

Tabelle 3.5: 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.

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, 2.6.1, 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)                     an1 = [ 1     1     1     1 ]
al1 = all(l);                    al1 = [ 0     1     1     0 ]

an2 = any(l,2); al2 = all(l,2);  an2 = [ 1          al2 = [ 0
                                         1                  1
                                         1 ]                0 ]

an = any(l(:));                  an = 1
al = all(l(:));                  al = 0
```

Kapitel 4

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & \dots & a_{mn} \end{bmatrix} = [a_{jk}] \, . \tag{4.1}$$

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,

$$\mathbf{a} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = [a_j] \, . \tag{4.2}$$

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,

$$\mathbf{b} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = [b_j] \ . \tag{4.3}$$

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,

$$s = \mathbf{s} = s_{11} = s_1 = [s_j] \ . \tag{4.4}$$

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

4.1 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,

$$\mathbf{A}^T = [a_{kj}] \ . \tag{4.5}$$

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

Symmetrische Matrix: Für eine symmetrische Matrix gilt

$$\mathbf{A}^T = \mathbf{A} \ . \tag{4.6}$$

Schiefsymmetrische Matrix: Für eine schiefsymmetrische Matrix gilt

$$\mathbf{A}^T = -\mathbf{A} \ . \tag{4.7}$$

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

$$\mathbf{A} = \mathbf{S} + \mathbf{U} \ , \tag{4.8}$$

$$\mathbf{S} = \frac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right) \ , \tag{4.9}$$

$$\mathbf{U} = \frac{1}{2}\left(\mathbf{A} - \mathbf{A}^T\right) \ . \tag{4.10}$$

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

4.2 Addition und Subtraktion von Matrizen

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

$$\mathbf{C} = \mathbf{A} \pm \mathbf{B} \implies [c_{jk}] = [a_{jk}] \pm [b_{jk}] \ , \tag{4.11}$$

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,

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \begin{bmatrix} 5\ 7\ 9 \end{bmatrix} \ , \tag{4.12}$$

$$\mathbf{D} = \mathbf{A} + 1 = \begin{bmatrix} 2\ 3\ 4 \end{bmatrix} \ . \tag{4.13}$$

Fehler: Die Rechnung

$$\mathbf{C} = \mathbf{A} + \mathbf{B}^T = \begin{bmatrix} 1\ 2\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \text{Error} \tag{4.14}$$

ergibt natürlich eine Fehlermitteilung.

4.3 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 $c\mathbf{A} = [ca_{jk}]$.

4.4 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,

$$c_{jk} = \sum_{l=1}^{n} a_{jl} b_{lk} \ . \tag{4.15}$$

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 4.15).

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

$$\mathbf{AB} \neq \mathbf{BA} \; . \tag{4.16}$$

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:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 14 & 32 \\ 32 & 77 \end{bmatrix} \; . \tag{4.17}$$

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

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 17 & 22 & 27 \\ 22 & 29 & 36 \\ 27 & 36 & 45 \end{bmatrix} \; . \tag{4.18}$$

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

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \text{Error} \; . \tag{4.19}$$

Inneres Produkt zweier Vektoren:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 32 \; . \tag{4.20}$$

Äußeres Produkt zweier Vektoren:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix} \; . \tag{4.21}$$

Fehler:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \text{Error} \; . \tag{4.22}$$

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

$$\mathbf{C}^T = \mathbf{B}^T \mathbf{A}^T \tag{4.23}$$

$$\mathbf{C} = (\mathbf{B}^T \mathbf{A}^T)^T \tag{4.24}$$

$$(c\mathbf{A})^T = c\mathbf{A}^T \tag{4.25}$$

4.5 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

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_{l=1}^{n} a_l b_l \ . \tag{4.26}$$

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.

4.6 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**

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \ . \tag{4.27}$$

Für die Einheitsmatrix gilt

$$\mathbf{AI} = \mathbf{IA} = \mathbf{A} \ . \tag{4.28}$$

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

Die **inverse Matrix** ist definiert durch

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \ . \tag{4.29}$$

Eine Matrix für die

$$\mathbf{A}^T = \mathbf{A}^{-1} \tag{4.30}$$

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

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

4.7 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

$$\mathbf{A}\mathbf{x} = \mathbf{b} \; . \tag{4.31}$$

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \text{, und } \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} \text{ und } \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \tag{4.32}$$

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:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array} \tag{4.33}$$

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 4.33 ist ein Vektor $\mathbf{x}$ der Länge n, der alle m Gleichungen erfüllt. Ist das System 4.33 homogen, hat es zumindest die **triviale** Lösung

$$x_1 = 0 \; , \; x_2 = 0 \; , \; \dots \; , \; x_n = 0 \; . \tag{4.34}$$

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 $\mathbf{x}$ zu berechnen, kann man nun formal 4.31 von Links mit $\mathbf{A}^{-1}$ multiplizieren

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \implies \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{A}\backslash\mathbf{b}. \tag{4.35}$$

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

$$\mathbf{y}\mathbf{A} = \mathbf{c}\ , \tag{4.36}$$

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

$$\mathbf{y}\mathbf{A}\mathbf{A}^{-1} = \mathbf{c}\mathbf{A}^{-1} \implies \mathbf{y} = \mathbf{c}\mathbf{A}^{-1} = \mathbf{c}/\mathbf{A}\ . \tag{4.37}$$

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 4.37 umgeformt werden in

$$\mathbf{y} = \left((\mathbf{A}^{-1})^T\mathbf{c}^T\right)^T = \left(\mathbf{A}^T\backslash\mathbf{c}^T\right)^T\ , \tag{4.38}$$

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

Für die Skalare s und r würde gelten

$$\begin{array}{rcccl} s/r & = & sr^{-1} & = & s/r \\ r\backslash s & = & r^{-1}s & = & s/r, \end{array} \tag{4.39}$$

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 4.33 umschreiben als:

$$\begin{array}{ccccccccc}
a_{11}x_{11} & + & a_{12}x_{21} & + & \dots & + & a_{1n}x_{n1} & = & b_{11} \\
a_{21}x_{11} & + & a_{22}x_{21} & + & \dots & + & a_{2n}x_{n1} & = & b_{21} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{11} & + & a_{m2}x_{21} & + & \dots & + & a_{mn}x_{n1} & = & b_{m1} \\
\hline
a_{11}x_{12} & + & a_{12}x_{22} & + & \dots & + & a_{1n}x_{n2} & = & b_{12} \\
a_{21}x_{12} & + & a_{22}x_{22} & + & \dots & + & a_{2n}x_{n2} & = & b_{22} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{12} & + & a_{m2}x_{22} & + & \dots & + & a_{mn}x_{n2} & = & b_{m2} \\
\hline
\vdots & + & \vdots & + & \vdots & + & \vdots & = & \vdots \\
\hline
a_{11}x_{1p} & + & a_{12}x_{2p} & + & \dots & + & a_{1n}x_{np} & = & b_{1p} \\
a_{21}x_{1p} & + & a_{22}x_{2p} & + & \dots & + & a_{2n}x_{np} & = & b_{2p} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{1p} & + & a_{m2}x_{2p} & + & \dots & + & a_{mn}x_{np} & = & b_{mp}
\end{array} \tag{4.40}$$

Dieses Gleichungssystem kann formal geschrieben werden als

$$\mathbf{AX} = \mathbf{B} \, , \tag{4.41}$$

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 5

Steuerkonstrukte

5.1 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
```

5.2 Auswahl

In Programmen ist es häufig notwendig, Entscheidungen zu treffen, welche Strukturblöcke abgearbeitet werden sollen. Man trifft dabei mit Hilfe von logischen Aus-

drücken, 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.

5.2.1 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.

5.2.2 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.

5.3 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:

$$a_1 = 0\ ,\ a_2 = 1\ ,\ a_k = \frac{a_{k-2} + a_{k-1}}{2} \qquad \forall k \geq 3\ . \tag{5.1}$$

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.

5.3.1 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.

5.3.2 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 $k \leq k \leq n$ 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 6

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.

6.1 FUNCTION-Unterprogramme

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 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 6.1.5 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. Der Befehl `isidentical` stammt nicht von MATLAB sondern wird hier bereitgestellt.

`k = ischar(S)`	Zeichenkette
`k = isempty(A)`	Leeres Array
`k = isequal(A,B,...)`	Identische Größe und Inhalt
`k = isidentical(A,B,...)`	Identische Größe
`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

6.1.5 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.

6.1.6 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 angibt. 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
```

6.1.7 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 ==> ....
```

6.1.8 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 6.1.7 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.

$$A_1 = \int_a^b dx\; x^n \exp(-x^2) \tag{6.1}$$

$$A_2 = \int_a^b \int_c^d dxdy\; m \exp\left(-n(x^2+y^2)\right) \tag{6.2}$$

```
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.

6.1.9 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.

Kapitel 7

Polynome

7.1 Grundlagen

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

$$p_1 x^{n-1} + p_2 x^{n-2} + p_3 x^{n-3} + \cdots + p_n \ , \tag{7.1}$$

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,

$$y_i = p_1 x_i^{n-1} + p_2 x_i^{n-2} + p_3 x_i^{n-3} + \cdots + p_n \ , \tag{7.2}$$

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

$$y_i = x_i^3 + 2x_i^2 + x_i + 3 \ , \tag{7.3}$$

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,

$$y_i = ((p_1 x + p_2)\, x + p_3)\, x + p_4 \ , \tag{7.4}$$

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.

7.2 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

$$\det\left(\mathbf{A} - \lambda\mathbf{I}\right) \ , \tag{7.5}$$

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

Dies soll am Beispiel von

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \ , \ \mathbf{c} = \det \begin{bmatrix} 2-\lambda & 1 \\ 2 & 3-\lambda \end{bmatrix} \ , \tag{7.6}$$

demonstriert werden, wobei sich hier als Lösung

$$\mathbf{c} = (2-\lambda)(3-\lambda) - 2 = \lambda^2 - 5\lambda + 4 \tag{7.7}$$

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

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & \\ & & 1 & & 5 & & 10 \\ & & & 1 & & 6 & \end{array} \tag{7.8}$$

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

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{bmatrix} \tag{7.9}$$

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

$$p(x) = x^4 - 29x^3 + 72x^2 - 29x + 1 \; . \tag{7.10}$$

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)`

$$\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \; , \tag{7.11}$$

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

7.3 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))`).

7.4 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(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.

7.5 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.

$$\begin{aligned} v(x) &= & x^3 + 2x^2 + 3x + 4 \\ u(x) &= & x + 2 \\ q(x) = v(x)/u(x) &= & x^2 + 3 \\ r(x) &= & -2 \end{aligned} \tag{7.12}$$

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.

7.6 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 das die Modellfunktion mit $p(x)$ bezeichnet wird, kann die Summe der Abstandsquadrate geschrieben werden als

$$q = \sum_{j=1}^{m} (y_j - p(x_j))^2 \stackrel{!}{=} \text{Min} , \tag{7.13}$$

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 8

Zeichenketten

8.1 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 9

Graphische Ausgabe

9.1 Grundlagen

9.2 Beispiele

9.2.1 Zweidimensionale Plots

Es gibt eine Reihe von Befehlen zur Darstellung zweidimensionaler Graphiken.

Tabelle 9.1: MATLAB Befehle zum Erzeugen einfacher zweidimensionaler Graphiken

`fplot('fun',[`x_{min}`,`x_{max}`])`	9.2.1.1	Zeichnet 'fun' im Bereich von x_{min} bis x_{max}
`plot(x,y)`	9.2.1.2	Zeichnet y als Funktion von x
`ezplot('fun',[`x_{min}`,`x_{max}`])`	9.2.1.3	erstellt u.a. implizite Funktionen, automatische Achsenbeschriftung
`comet(x,y,p)`	9.2.1.4	Zeichnet 2D Funktion in Form eines animierten 'Kometen'
`semilogx(x,y)`	9.2.1.5	Zeichnet 2D Funktion mit (10er-) logarithmischer x-Achse
`semilogy(x,y)`	9.2.1.6	Zeichnet 2D Funktion mit (10er-) logarithmischer y-Achse
`loglog(x,y)`	9.2.1.7	Zeichnet 2D Funktion mit (10er-) logarithmischer x- und y-Achse
`plotyy(`x_1`,`y_1`,`x_2`,`y_2`,'`f_1`','`f_2`')`	9.2.1.8	Erstellt 2 Graphen mit den Plotbefehlen f_1 und f_2 mit getrennten y - Achsen
`polar(phi,r)`	9.2.1.9	Zeichnet die Funktion r(phi) in Polarkoordinaten.

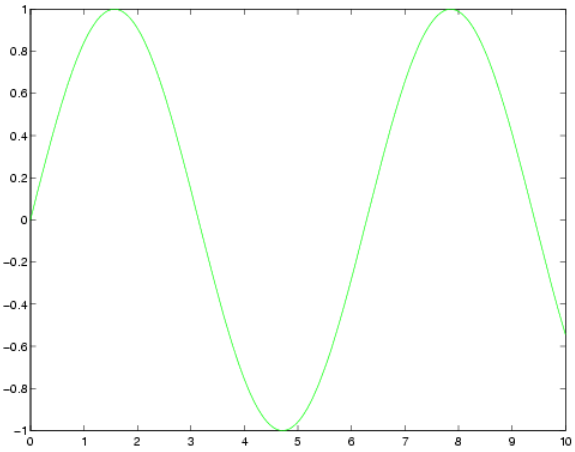
9.2.1.1 Fplot

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`.

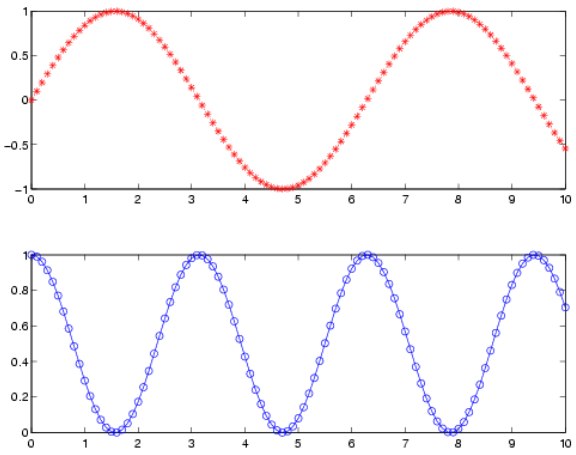
9.2.1.2 Plot

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`

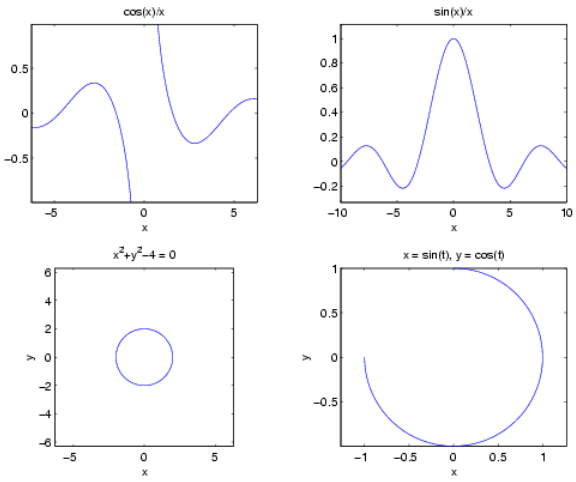
9.2.1.3 Ezplot

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])
```



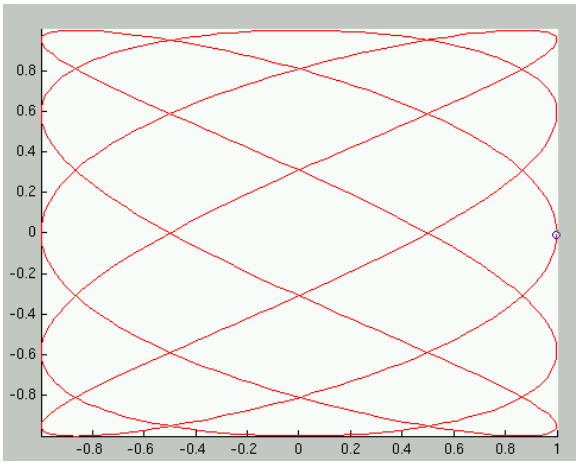
9.2.1.4 Comet

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.

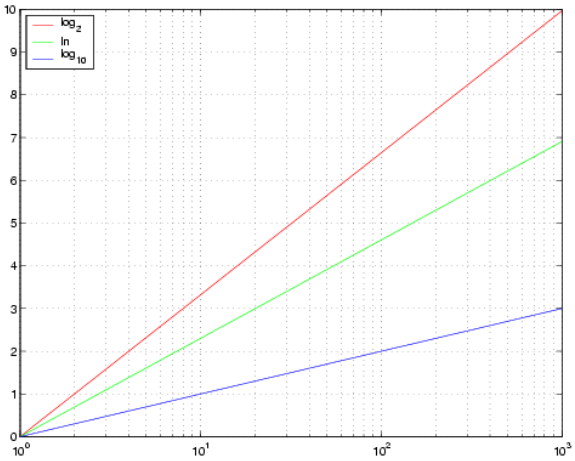
9.2.1.5 Semilogx

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)
```



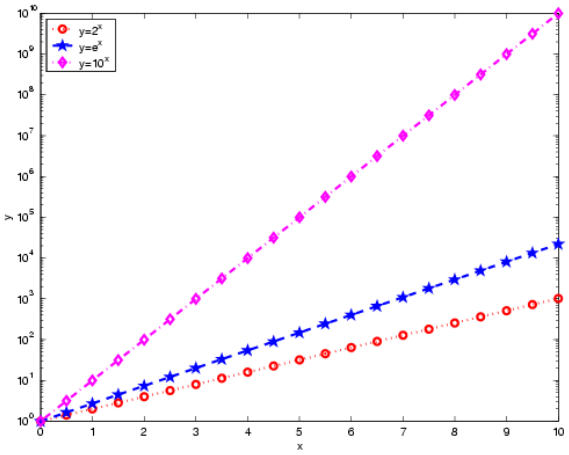
Sollen mehrere Lines in eine Achse gezeichnet werden, so können die Koordinaten und Style Eigenschaften der Lines hintereinandergefügt werden.

9.2.1.6 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.

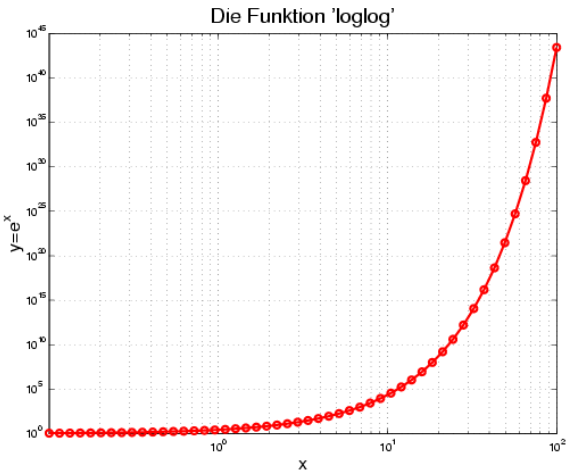
9.2.1.7 Loglog

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.

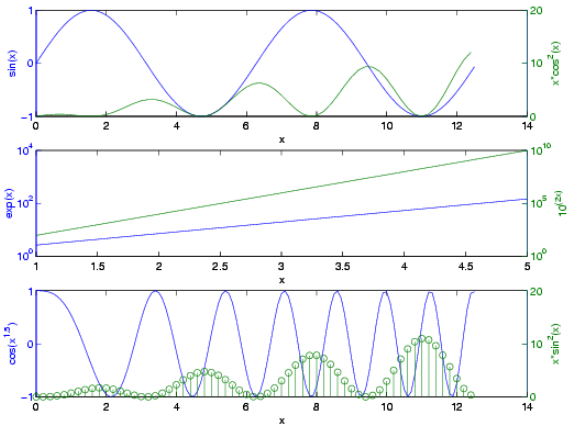
9.2.1.8 Plotyy

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 9.2: MATLAB Befehle zum Erzeugen von Balken- und Kreisdiagrammen

`hist(y,x)`	9.2.1.10	Erstellt ein Histogramm der Werte in y über jenen von x
`bar(x,y,'width','style')`	9.2.1.11	Stellt die Datenpaare [x,y] als vertikale Balken dar
`barh(x,y,'width','style')`	9.2.1.12	Stellt die Datenpaare [x,y] als horizontale Balken dar
`pie(x,'explode')`	9.2.1.13	Zeichnet ein 2D Kreisdiagramm der Daten von x

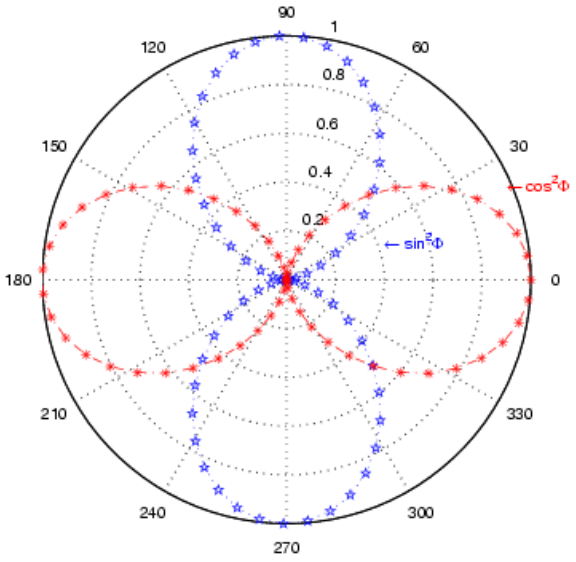
9.2.1.9 Polardiagramm

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.

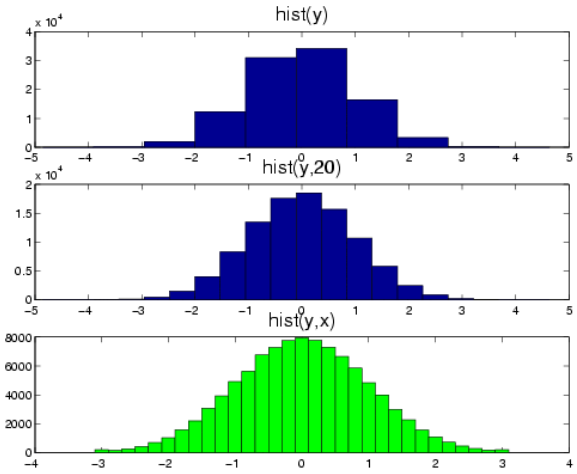
9.2.1.10 Histogramm

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.

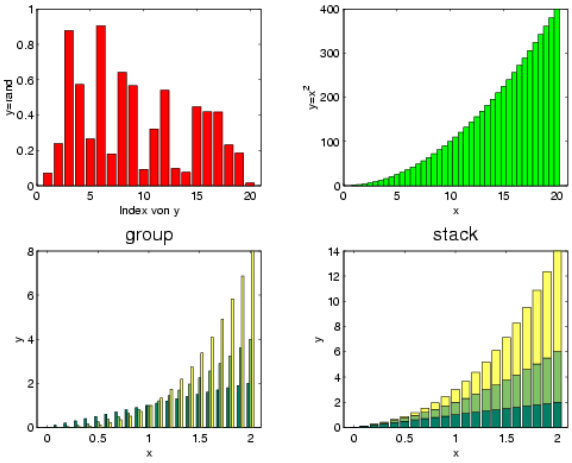
9.2.1.11 Bar

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.

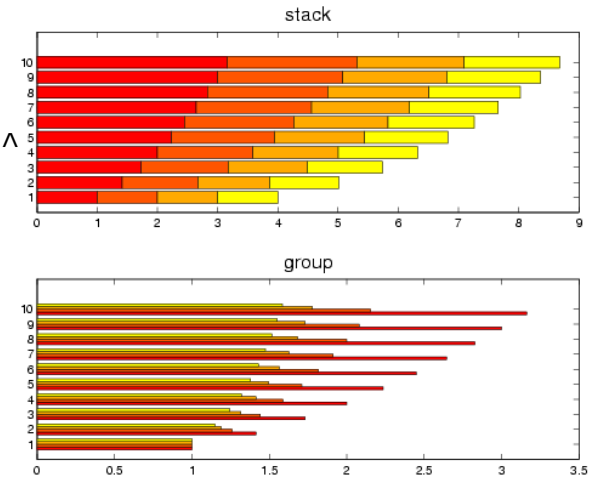
9.2.1.12 Barh

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 9.2.1.11 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.

9.2.1.13 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)
```

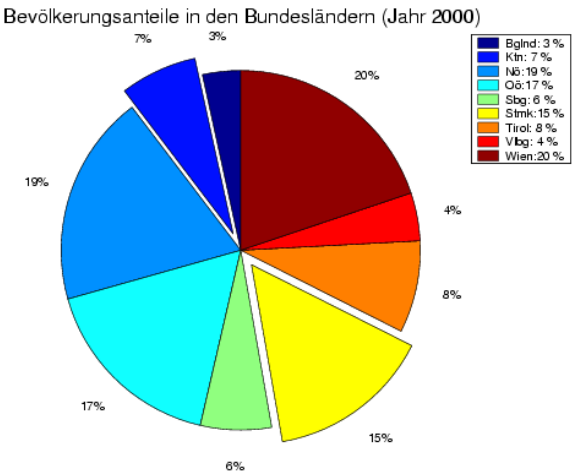


Tabelle 9.3: MATLAB Befehle zum Erzeugen von speziellen zweidimensionalen Graphiken

`stem(x,y)`	9.2.1.14	Zeichnet y=f(x) und verbindet Punkte mit x-Achse
`stairs(x,y)`	9.2.1.15	Erstellt Funktion y=f(x) in Form eines Stufendiagramms
`errorbar(x,y,e)`	9.2.1.16	Zeichnet y als Funktion von x samt Fehlerbalken der Länge e
`compass(x,y)`	9.2.1.17	Zeichnet y=f(x) und verbindet die Punkte durch Vektorpfeile mit dem Ursprung
`feather(u,v)`	9.2.1.18	Zeichnet die relativen Koordinaten u und v und verbindet die Punkte mit den jeweiligen Koordinatenursprüngen entlang der Abszisse
`scatter(x,y,r,c)`	9.2.1.19	Zeichnet Punkte an den Stellen (x,y) der Größe r sowie der Farbe c
`pcolor(x,y,c)`	9.2.1.20	Erstellt einen 'Pseudocolorplot' der Elemente c an den von den Punkten (x,y) definierten Positionen
`area(x,y)`	9.2.1.21	Füllt den Bereich zwischen y=f(x) und der Abszisse mit einer Farbe
`fill(x,y,c)`	9.2.1.22	Malt die durch (x,y) definierten Polygone mit der Farbe c aus
`contour(x,y,z)`	9.2.1.23	Zeichnet durch z=f(x,y) definierte Konturlinien
`contourf(x,y,z)`	9.2.1.24	Zeichnet durch z=f(x,y) definierte Konturlinien und füllt die Flächen dazwischen aus
`quiver(x,y,u,v)`	9.2.1.25	Erstellt von den Punkten (x,y) ausgehende Vektoren mit den Komponenten (u,v)
`plotmatrix(x,y)`	9.2.1.26	Streudiagramm, die Spalten von x werden über jenen von y aufgetragen

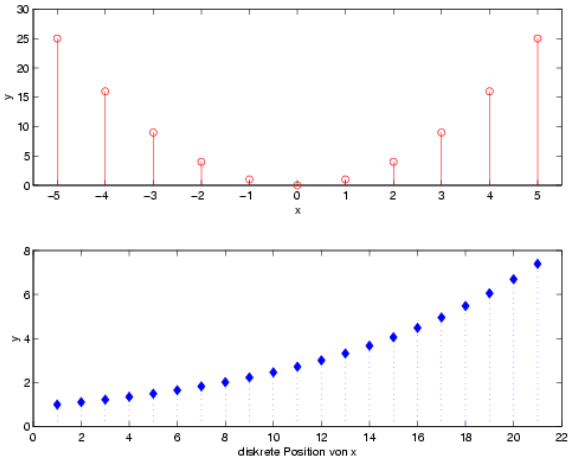
9.2.1.14 Stem

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])
```



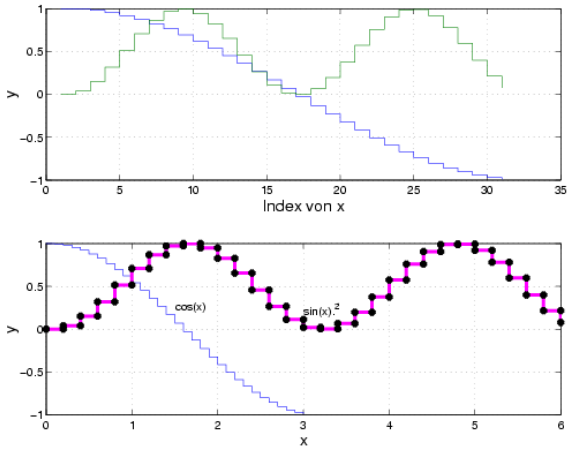
Im ersten Subplot werden die Achsengrenzen durch `axis([xmin,xmax, ymin, ymax])` geregelt, im zweiten Subplot mit dem Befehl `xlim`, wobei der Wertebereich der y-Achse unberührt bleibt.

9.2.1.15 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.

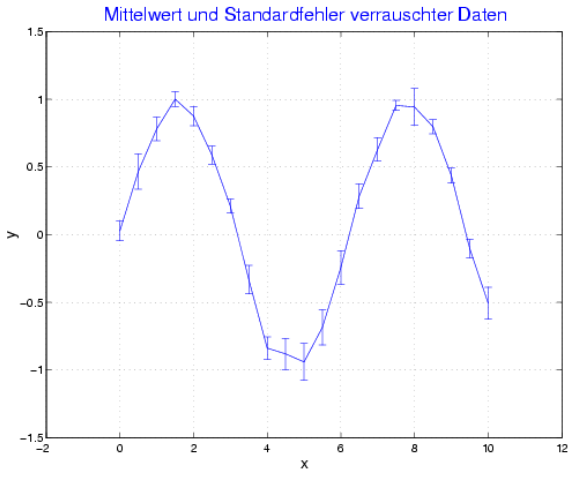
9.2.1.16 Errorbar

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));
```



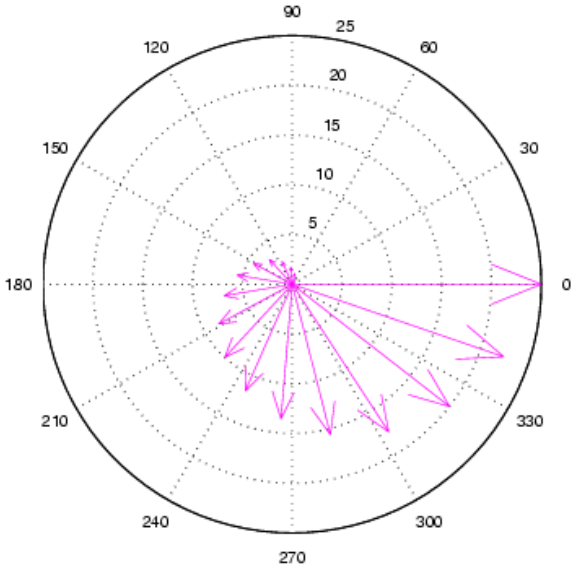
9.2.1.17 Compass

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.

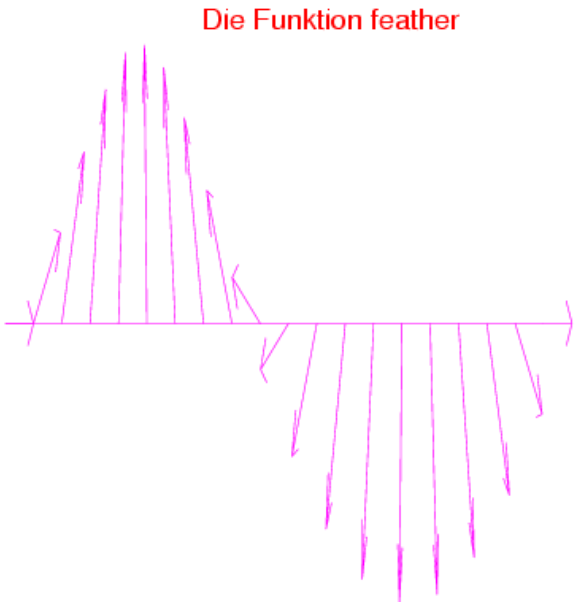
9.2.1.18 Feather

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.

9.2.1.19 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.

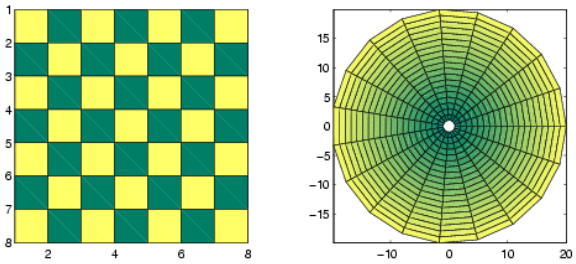
9.2.1.20 Pseudocolor

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)
```



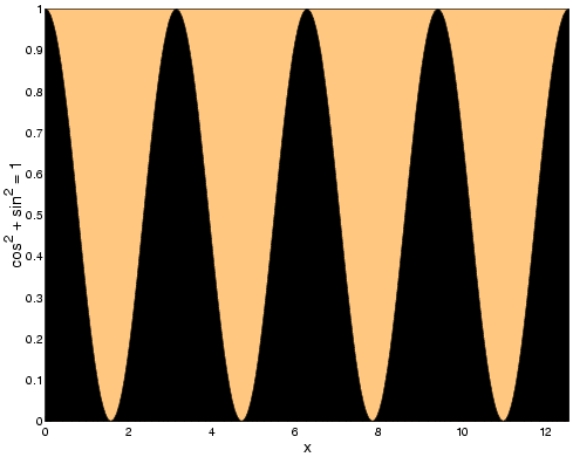
`axis ij` wählt für das Achsensystem den Matrixmodus, wodurch die Indizierung in der linken oberen Ecke der dargestellten Matrix beginnt und jede Zelle die Länge 1 besitzt.

9.2.1.21 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
```



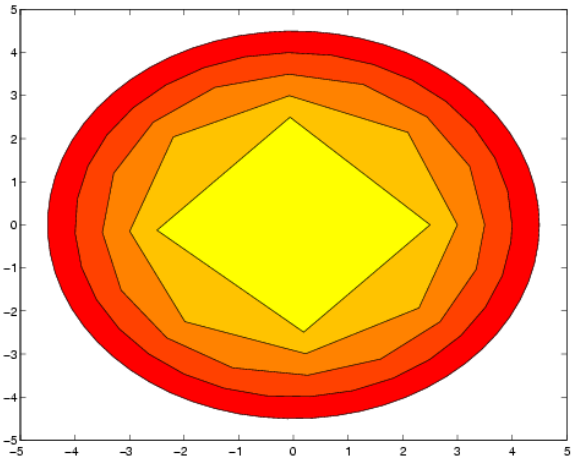
`axis tight` wählt die Achsengrenzen derart, dass sie nur den Bereich der Graphik abdecken.

9.2.1.22 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
```

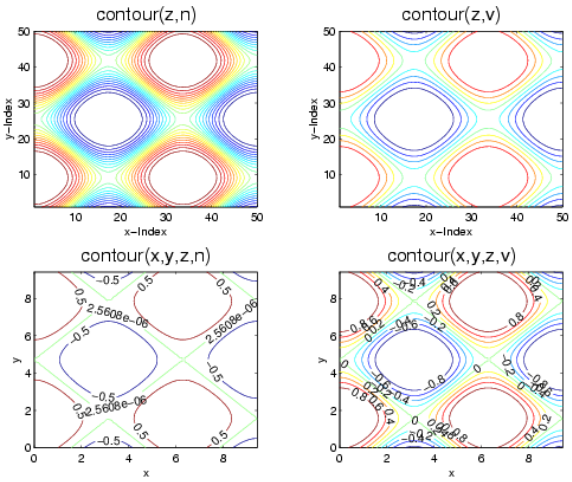
9.2.1.23 Contour

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)
```



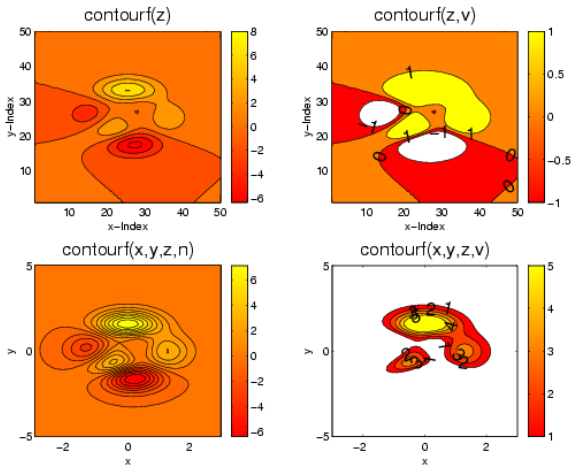
Mit `clabel` werden die Konturlinien mit den entsprechenden z-Werten beschriftet.

9.2.1.24 Contourf

Ähnliche Wirkung wie `contour` in 9.2.1.23, 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
```



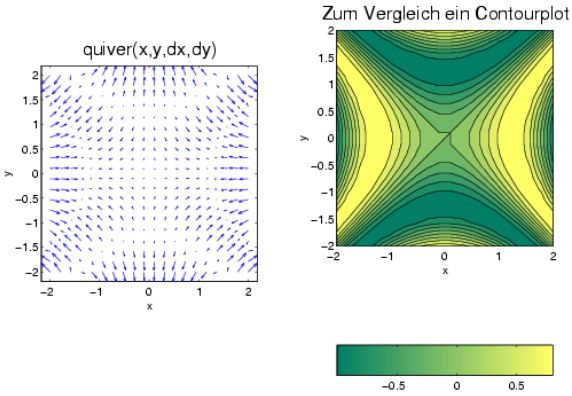
Der Befehl `colorbar` fügt am rechten Rand der Achse eine Farbskala mit einer Zuordnung der Farben zu den z-Werten hinzu.

9.2.1.25 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 9.4: MATLAB Befehle zum Erzeugen einfacher dreidimensionaler Graphiken

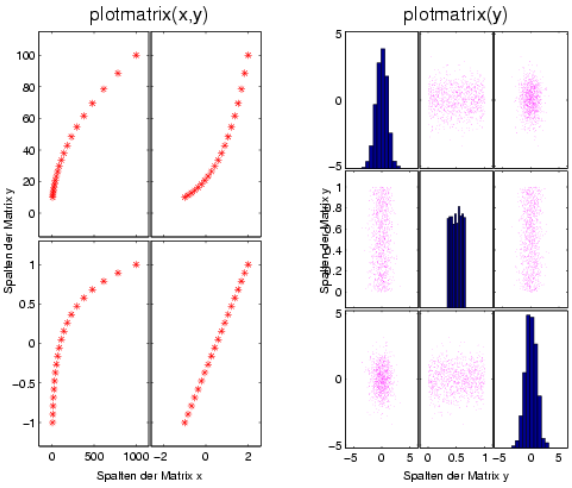
`plot3(x,y,z)`	9.2.2.1	3D Daten werden durch Angabe von x, y und z dargestellt
`ezplot3(x(t),y(t),z(t))`	9.2.2.2	Erstellt parametrischen 3D Plot durch Angabe der Funktionen als Strings und des Wertebereichs für t
`comet3(x,y,z,p)`	9.2.2.3	Zeichnet 3D Funktion in Form eines animierten 'Kometen'
`fill3(x,y,z,c)`	9.2.2.4	Malt die durch (x,y,z) definierten 3D-Polygone mit der Farbe c aus

9.2.1.26 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.

9.2.2 Dreidimensionale Plots

Matlab bietet auch eine Fülle von Befehlen, 3D Graphiken eindrucksvoll darzustellen

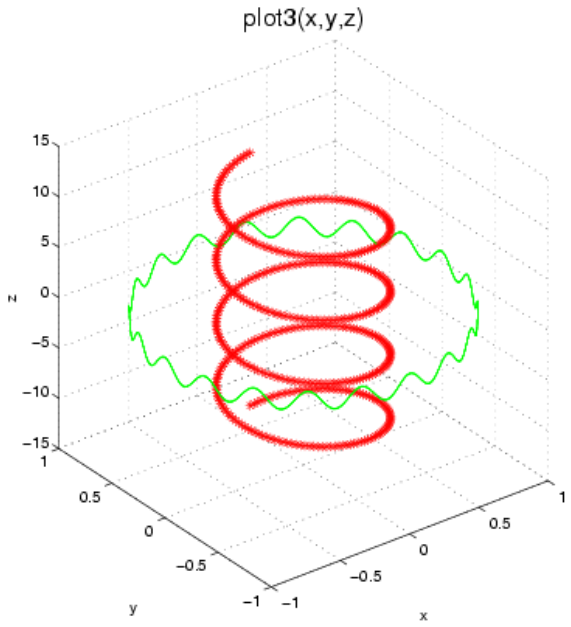
9.2.2.1 Plot3

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
```



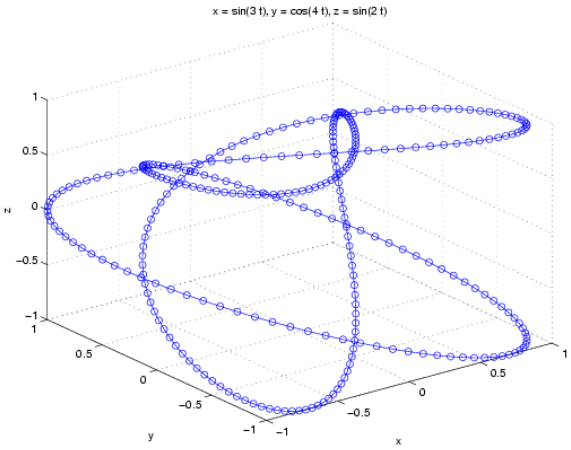
Der Befehl `rotate3d` ermöglicht eine Drehung des Achsensystems mit Hilfe der Maus.

9.2.2.2 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.

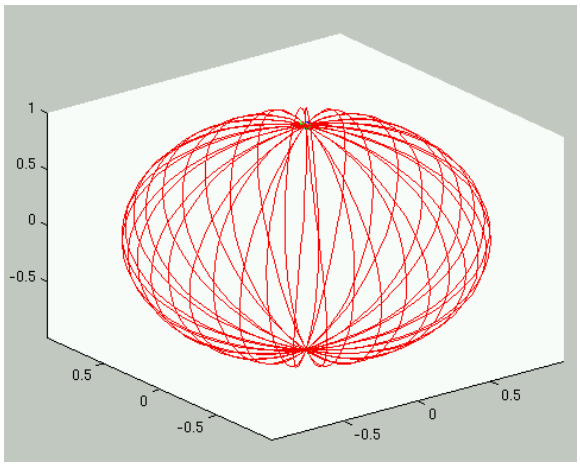
9.2.2.3 Comet3

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 9.5: MATLAB Befehle zum Erzeugen von 3D-Balken- und Kreisdiagrammen

`bar3(x,y,w,'style')`	9.2.2.5	Stellt die 2D Daten als vertikale 3D Balken dar
`bar3h(x,y,w,'style')`	9.2.2.6	Stellt die 2D Daten als horizontale 3D Balken dar
`pie3(x,'explode')`	9.2.2.7	Zeichnet ein 3D Kreisdiagramm von x

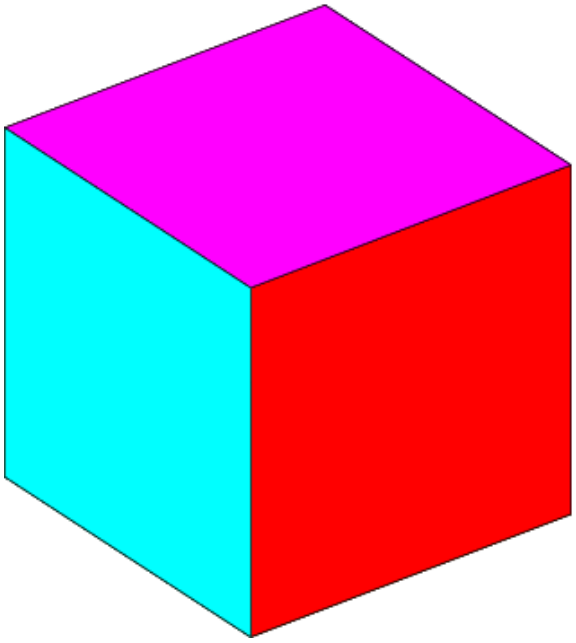
9.2.2.4 Fill3

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)
```



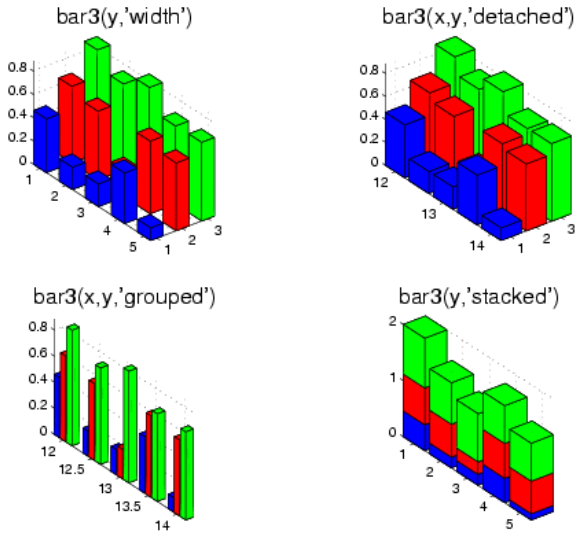
9.2.2.5 Bar3

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')
```



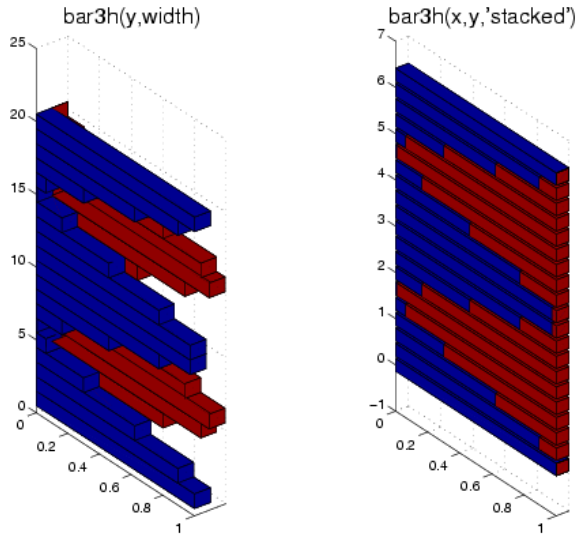
Man beachte die unterschiedliche Darstellung der Säulendiagramme bei der Verwendung der Stile 'detached', 'grouped' und 'stacked'.

9.2.2.6 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.

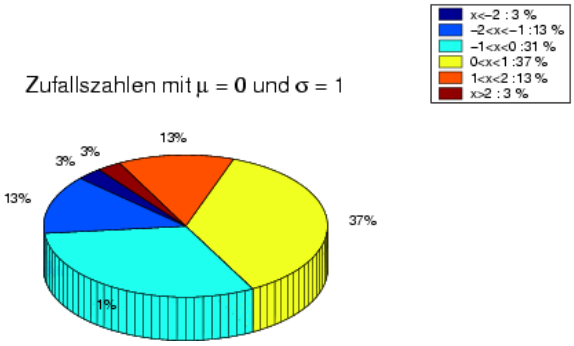
9.2.2.7 Pie3

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 9.6: MATLAB Befehle zum Erstellen von 3D - Oberflächen

`contour3(x,y,z)`	9.2.2.8	Zeichnet durch z=f(x,y) definierte 3D-Konturlinien
`mesh(x,y,z)`	9.2.2.9	Stellt die Matrix z=f(x,y) in Form eines 'Drahtgitters' dar
`ezmesh('f(x,y)')`	9.2.2.10	'Easy to use' Variante von mesh, f(x,y) wird als String eingegeben
`meshc(x,y,z)`	9.2.2.11	Zeichnet ein 3D-Drahtgitter und einen 2D-Contourplot der Funktion z=f(x,y)
`meshz(x,y,z)`	9.2.2.12	Zeichnet ein 3D-Drahtgitter der Funktion z=f(x,y) mit zusätzlichen seitlichen Referenzlinien
`trimesh(tri,x,y,z)`	9.2.2.13	Zeichnet ein aus Dreiecken bestehendes 3D-Drahtgitter der Funktion z=f(x,y)
`surf(x,y,z)`	9.2.2.14	Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y)
`ezsurf('f(x,y)')`	9.2.2.15	'Easy to use' Variante von surf, f(x,y) wird als String eingegeben
`surfc(x,y,z)`	9.2.2.16	Zeichnet eine 3D-Oberflächengraphik und einen 2D-Contourplot der Funktion z=f(x,y)
`ezsurfc(x,y,z)`	9.2.2.17	'Easy to use' Variante von surfc, f(x,y) wird als String eingegeben
`surfl(x,y,z)`	9.2.2.18	Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y) mit wählbarer Beleuchtung
`trisurf(tri,x,y,z)`	9.2.2.19	Zeichnet eine 3D-Oberfläche der Funktion z=f(x,y) aus Dreiecken
`waterfall(x,y,z)`	9.2.2.20	Zeichnet die Reihen der Matrix z=f(x,y) als 3D-Linien entlang der x-Achse

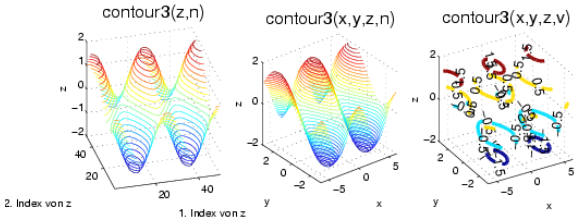
9.2.2.8 Contour3

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);
```



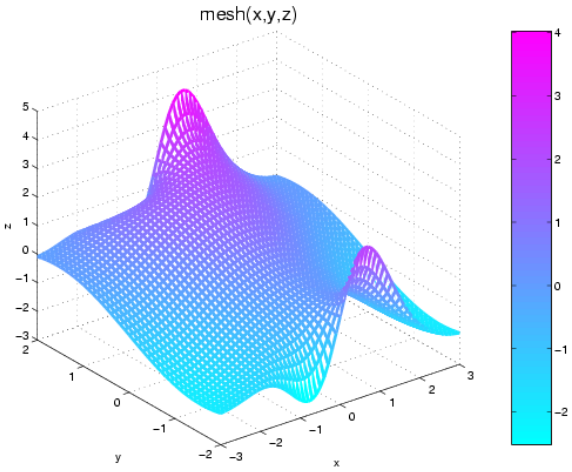
Mit `clabel` werden die Konturlinien mit den entsprechenden z-Werten beschriftet.

9.2.2.9 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
```



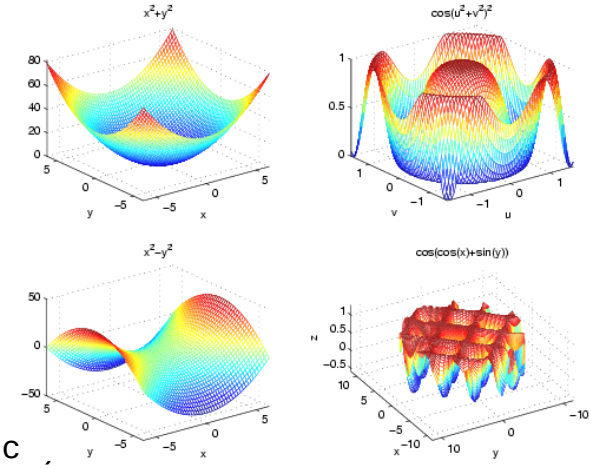
Zur Erinnerung: mit `get(h)` können alle Eigenschaften des mit dem Handle h verknüpften Graphik-Objekts ausgegeben und mit `set(h,'Eigenschaft','Wert')` gesetzt werden.

9.2.2.10 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
```



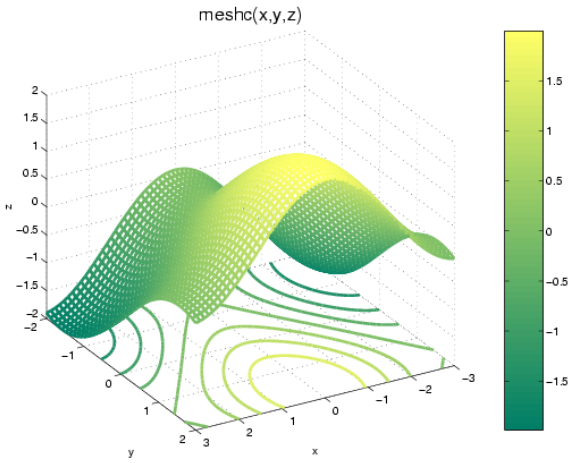
Neben der Funktion f(x,y) können optional die Grenzen von x und y, die Anzahl der Gitterelemente oder der Ausdruck 'circ' (zeichnet Graphik über kreisförmigen Definitionsgebiet) angegeben werden.

9.2.2.11 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);
```



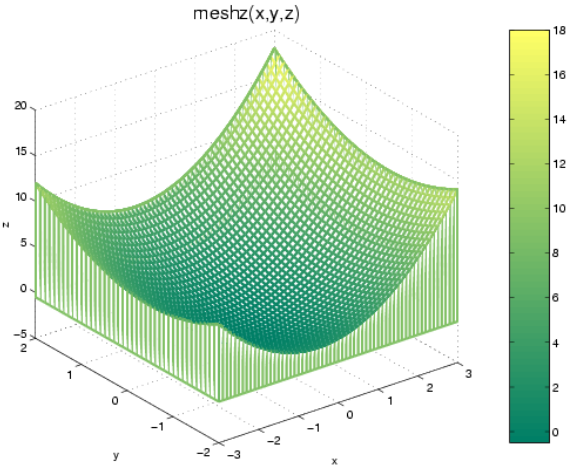
Die Dicke der Konturlinien kann nur gemeinsam mit jenen des Drahtgitters verändert werden.

9.2.2.12 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
```



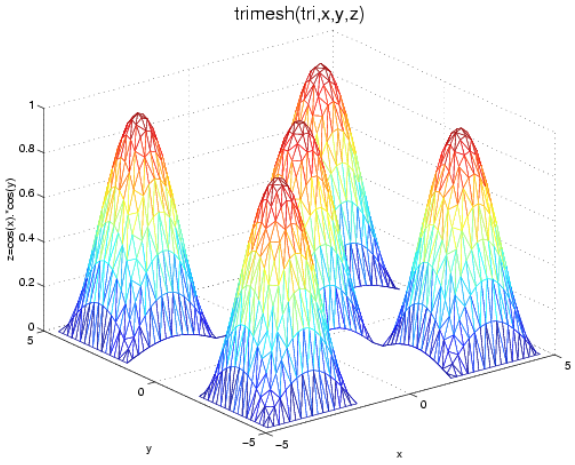
9.2.2.13 Trimesh

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.

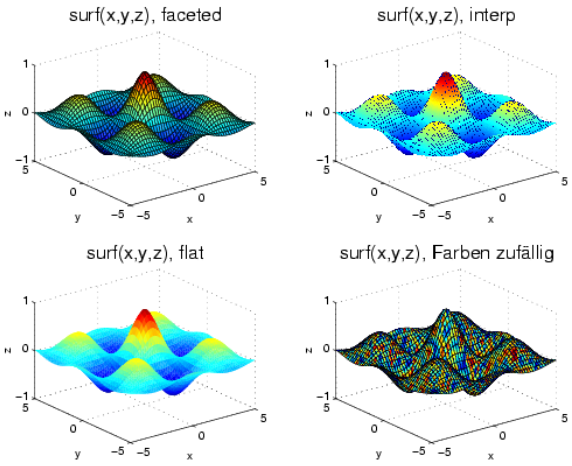
9.2.2.14 Surf

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')
```



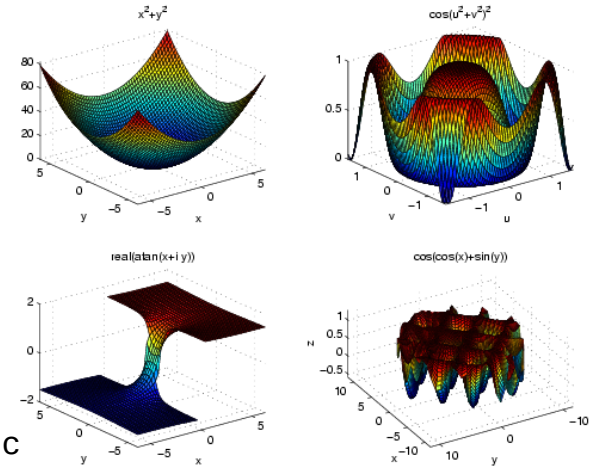
Werden im Aufruf von `surf` die x- und y- Matrizen weggelassen, so werden auf den x- und y- Achsen die beiden Indizes der Matrix z aufgetragen.

9.2.2.15 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)
```



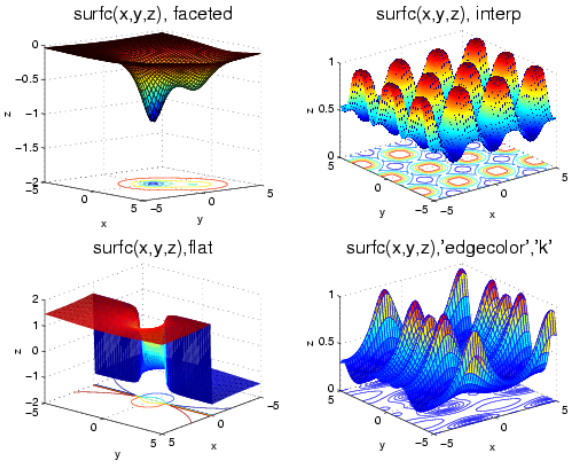
Neben der Funktion f(x,y) können optional die Grenzen von x und y, die Anzahl der Gitterelemente oder der Ausdruck 'circ' (zeichnet Graphik über kreisförmigen Definitionsgebiet) angegeben werden. Mit Hilfe des Befehls `view` stellt man den Blickwinkel auf das Achsensystem ein. Die erste Komponente ist der Azimuthwinkel in Grad (Rotation der x,y Ebene), die zweite Komponente ist der Kippwinkel aus der horizontalen Lage der x,y Ebene.

9.2.2.16 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')
```

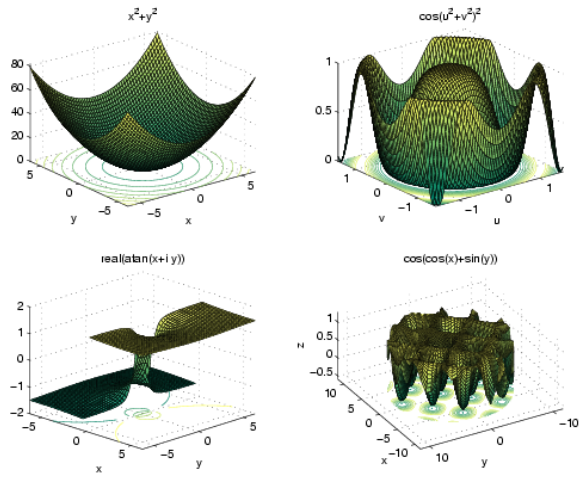


9.2.2.17 Ezsurfc

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')
```

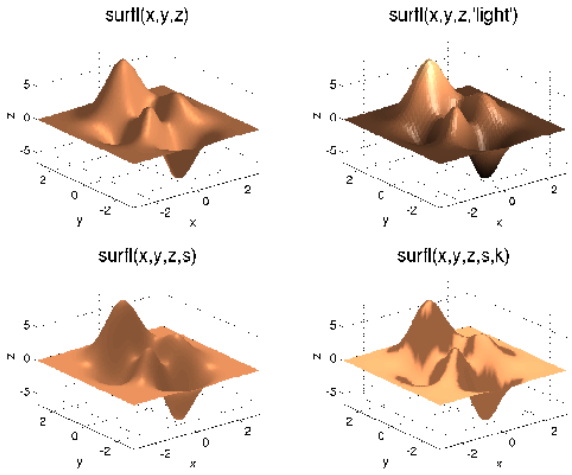


9.2.2.18 Surfl

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.

9.2.2.19 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')
```

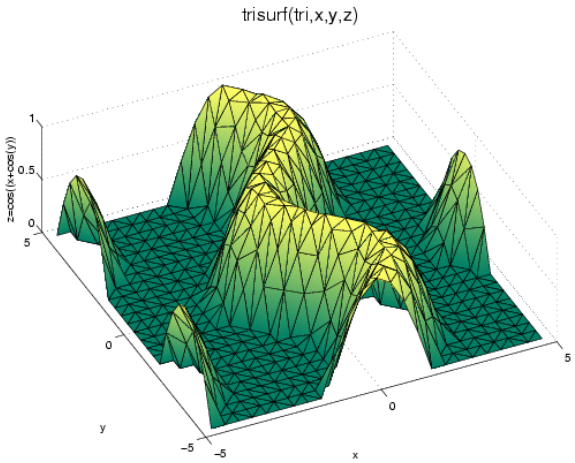


Tabelle 9.7: MATLAB Befehle zum Erstellen von 3D - volumetrischen Graphiken

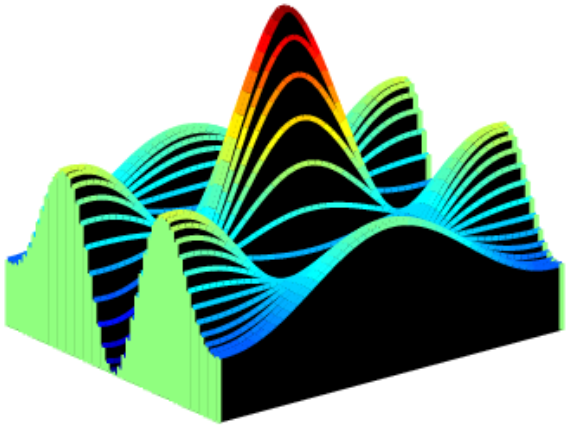
`quiver3(x,y,z,u,v,w)`	9.2.2.21	Zeichnet an den Punkten (x,y,z) Vektorpfeile mit den Komponenten (u,v,w)
`slice(x,y,z,d,sx,sy,sz)`	9.2.2.22	Veranschaulicht die volumetrische Funktion d=f(x,y,z) durch senkrecht durch die Achsen gelegte Schnittflächen

9.2.2.20 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');
```



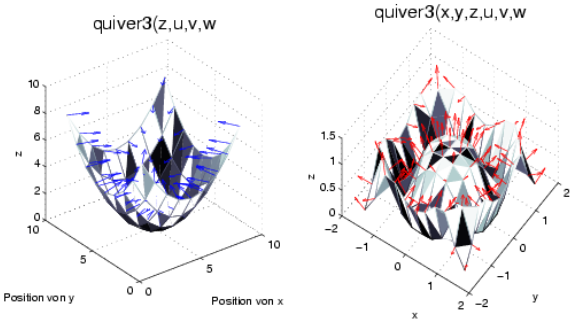
9.2.2.21 Quiver3

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 9.8: Weitere spezielle 3D Graphik-Befehle

`stem3(x,y,z)`	9.2.2.23	Zeichnet 3D Funktion und verbindet Datenpunkte mit der Ebene z=0
`sphere(n)`	9.2.2.24	Erstellt eine durch n^2 Flächen angenäherte Kugel
`cylinder(r,n)`	9.2.2.25	Erstellt einen durch ein n-seitiges Prisma angenäherten Zylinder mit Radius r
`scatter3(x,y,z,r,c)`	9.2.2.26	Zeichnet Daten an den Positionen (x,y,z) der Größe r sowie der Farbe c
`ribbon(y,z,w)`	9.2.2.27	Zeichnet die Spalten von z über jenen von y als 3D Bänder der Breite w

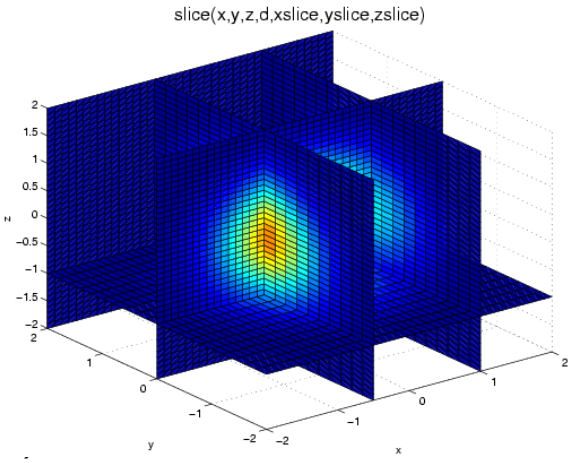
9.2.2.22 Slice

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.

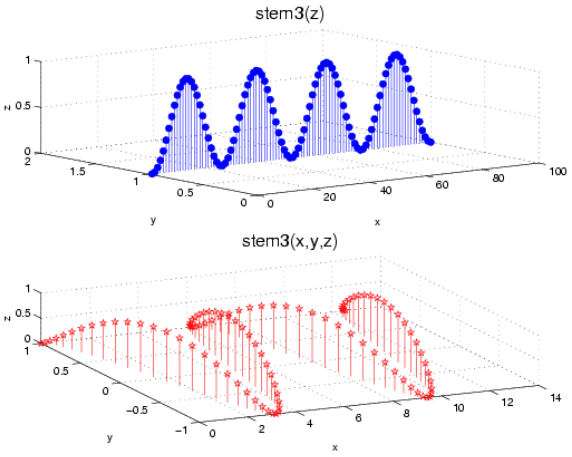
9.2.2.23 Stem3

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.

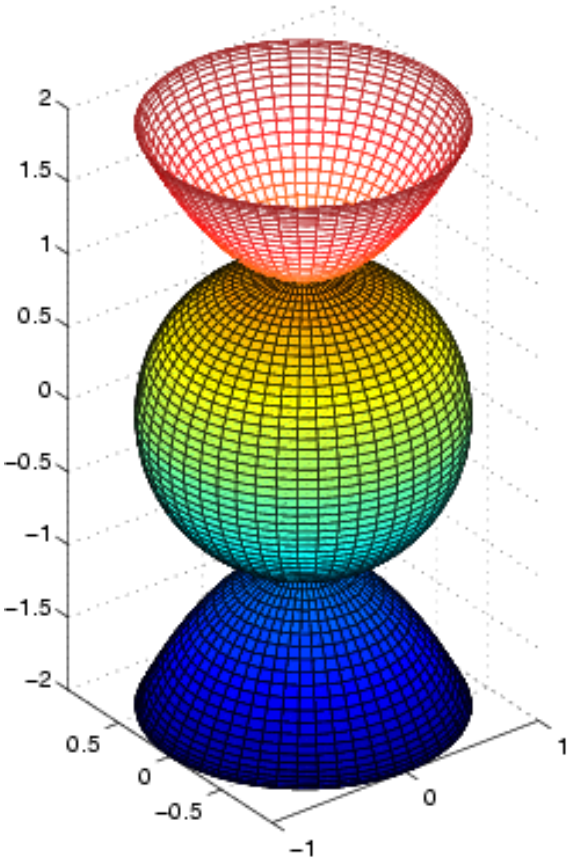
9.2.2.24 Kugel

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.

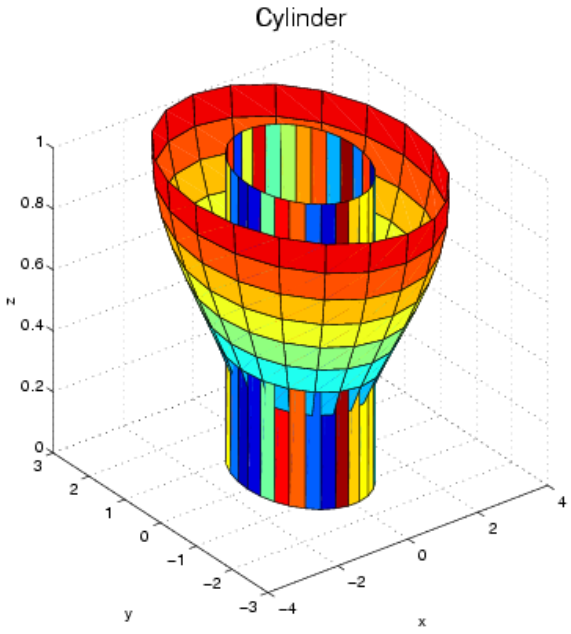
9.2.2.25 Zylinder

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 9.2.2.24 darin, dass auf diese Weise unter anderem Größe, Position und Farbeigenschaften beeinflusst werden können.

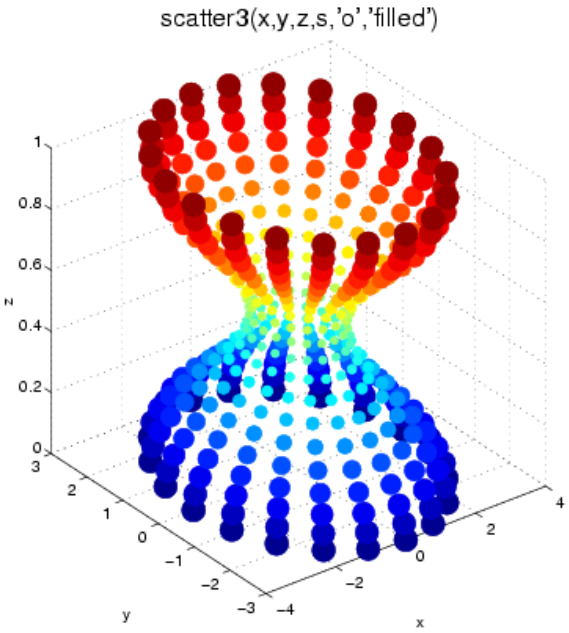
9.2.2.26 Scatter3

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
```

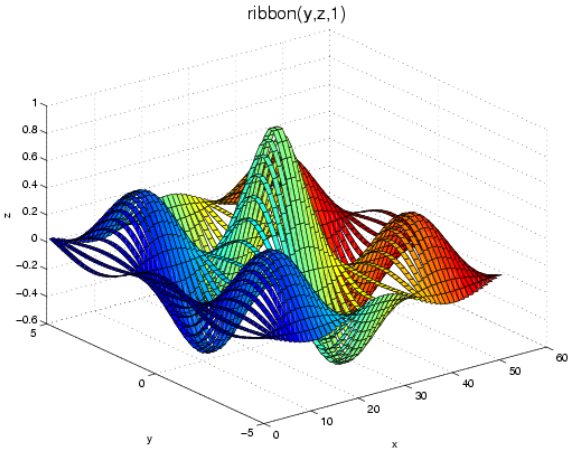


9.2.2.27 Ribbon

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 10

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 ...`
- `uebungsabgabe` allein zeigt Ihnen die Liste der schon abgegebenen Files.
- `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.

10.1 Funktionen, Input, Output

Ziel: Ein erster Einstieg in Matlab unter Linux soll geschafft werden. Editieren, Speichern und Ausführen von MATLAB-skripten. Aufruf einfacher eingebauter Funktionen, Ein- und Ausgabe.

Vorraussetzung: Grundlagen von Linux, Gnome und Matlab wie in der ersten Vorlesungsstunde besprochen.

10.1.1 Eine Formel

Schreiben Sie ein MATLAB-skript `lnexp.m`, das für Sie folgende Dinge tut:

- Lesen Sie drei Zahlen β, ε und μ von der Tastatur ein
- Berechnen Sie die y gemäß der Formel

$$y = \log\left(1 + e^{-\beta(\varepsilon-\mu)}\right) .$$

- Geben Sie das Ergebnis formatiert aus.

Kommentieren Sie Ihr skript (Autor, Datum, Zweck), sodass mit dem Befehl `help lnexp` die Kommentarzeilen erscheinen. Speichern Sie das erstellte skript mit dem Namen `lnexp.m` in ihrem MATLAB-Verzeichnis. Kopieren Sie es von dort auf eine Diskette.

10.1.2 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.

- Prüfen Sie durch Ausrechnen der linken und rechten Seite der unten angeführten Identitäten die Richtigkeit der Formeln nach.

$$
\begin{aligned}
e^{ia} &= \cos a + i\ \sin a \\
\log(a\,b) &= \log a + \log b \\
e^{a+b} &= e^a\, e^b \\
\cos(a+b) &= \cos(a)\cos(b) - \sin(a)\sin(b) \\
\sin(a+b) &= \sin(a)\cos(b) + \cos(a)\sin(b) \\
\sin(a) &= \left(e^{ia} - e^{-ia}\right)/(2i) \\
\cos(a) &= \left(e^{ia} + e^{-ia}\right)/2 \\
\sin(a/2) &= \ldots \\
\cos(a/2) &= \ldots
\end{aligned}
$$

  Geben Sie dazu das Ergebnis der linken und der rechten Seite der Gleichungen formatiert aus.

10.2 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 ein MATLAB Script-File `uebarray.m`, der die Lösungen für die Beispiele enthalten soll.

Voraussetzung: Speichern Sie den File `uebarray.m` in Ihrem MATLAB Directory, welches in der Regel `matlab` heißt. Dies kann direkt aus NETSCAPE mit dem Befehl `save as` erfolgen. Speichern Sie auch den Datenfile `liste.dat`.

Alternativ dazu können Sie mit dem Skript `uebungsdaten` (Übung 1) die Files automatisiert bei Ihnen speichern. Der Aufruf erfolgt einfach in einem Terminalfenster. Dieses Skript kann aber auch direkt aus MATLAB gestartet werden. Dabei muss man wie bei allen externen Programmen ein Rufzeichen voranstellen (`!uebungsdaten`).

Starten Sie MATLAB und geben Sie im Command Fenster den Befehl `uebarray` ein, damit sollte die Vorlage ablaufen, ein Resultat anzeigen, und auf einen Tastendruck warten. Funktioniert das nicht, stellen sie mit der Eingabe `dir` sicher, dass sich der File `uebarray.m` in Ihrem Matlab Directory befindet. Ist das nicht der Fall, dann müssen Sie den File erst mit NETSCAPE oder dem oben angegebenen Skript speichern.

Mit dem Befehl `edit uebarray` öffnet sich ein Editor Fenster mit diesem File. Der Umgang mit dem Editor sollte selbsterklärend sein.

Sie können nun Befehle zum Ausprobieren direkt im MATLAB Command Fenster eingeben, oder im Editor Befehle dazuschreiben, speichern und mit dem Befehl `uebarray` ausfuehren.

Mit dem Befehl `help uebarray` sehen sie wie bei jedem MATLAB Befehl die Hilfe zu diesem Kommando. Dazu wird der erste Block von Kommentaren (gekennzeichnet mit %) verwendet. Füllen Sie dort ihren Namen und das Datum aus.

1. Probieren Sie die Befehle `zeros`, `ones`, `eye`, `linspace`, `logspace` und die Operatoren `+` und `-` aus. Erzeugen Sie z.B. ein 3×4 Array mit lauter Dreiern. Die explizite Eingabe sollte dabei nicht verwendet werden.

2. Erzeugen Sie eine schachbrettartige Anordnung von Einsern und Nullen als Symbole für schwarz und weiß. Die Dimension der Matrix sollte 8×8 sein und sie sollte links oben mit einer Eins beginnen.

   Denken Sie dabei an die Verwendung der Befehle `eye` und `repmat`.

3. Studieren Sie den Befehl `diag` und erzeugen Sie dann mit Hilfe der Befehls `diag` die nachfolgende Matrix. Die Verwendung von `ones` und `eye` ist möglich aber nicht notwendig.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 3 & 1 & 1 \\ 1 & 1 & 1 & 4 & 1 \\ 1 & 1 & 1 & 1 & 5 \end{bmatrix}.$$

   Verwenden Sie dabei `n = 5` und danach nur mehr `n`.

4. Erstellen Sie eine 5×5 Matrix 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.

5. Erzeugen Sie mit dem Befehl `magic` ein $n \times n$ magisches Quadrat `M`, wobei $n = 5$ 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(A,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(A,1)` und `sum(A,2)` unterschiedlich ist.

   Das Zusammenhängen von Arrays erfolgt mit dem Befehl `cat` bzw. mit seiner Kurzform `[A,B]` oder `[A;B]`. Berechnen Sie anschließend die Länge dieses Vektors.

   Stehen im ganzen Vektor die gleichen Zahlen und ist seine Länge $2(n+1)$?

6. Erzeugen Sie folgende Matrix

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

   definieren Sie zuerst `n=4` und verwenden Sie dann die Zahl vier nicht mehr. Erzeugen Sie dabei einen Vektor und verwenden Sie dann die Befehle `reshape` und `transpose`.

Die Erzeugung des Vektors $1 \ldots n^2$ kann mit der `Doppelpunkt` Notation erfolgen. Für das Quadrieren kann man `n*n` oder `n^2` verwenden.

7. Speichern Sie in einer Matrix das mittlere 2×2 Quadrat der Matrix V. Verwenden Sie dabei das Keyword `end` in der Form `end-1`.

   Setzen Sie in der ursprünglichen Matrix die vier Eckpunkte auf Null. Verwenden Sie wieder die `Doppelpunkt` Notation und beachten Sie dabei, dass das Keyword `end` auch in der Schrittweite verwendet werden kann.

8. Lesen sie vom File `liste.dat` alle Werte in die Matrix D. Bilden Sie dann die Summe über alle Werte. Dabei können Sie im Summenbefehl `sum` die Form `D(:)` verwenden.

   Ist das Ergebnis Null?

Kapitel 11

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!

11.1 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 12

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. 12

[2] C. Überhuber and S. Katzenbeisser. *MATLAB 6 Eine Einführung*. Springer, 2000. 12

[3] D. Hanselman and B. Littlefield. *Mastering MATLAB 5, A Comprehensive Tutorial and Reference*. Prentice Hall, 1998. 12

[4] P. Marchand. *Graphics and GUIs with MATLAB*. ORC, second edition, 1999. 12

[5] G. Lindfield and J. Penny. *Numerical Methods Using MATLAB*. Ellis Horwood, 1995. 12

[6] H. Benker. *Mathematik mit MATLAB, Eine Einführung für Ingenieure und Naturwissenschaflter*. Springer, 2000. 12

[7] W. Gander and J. Hřebíček. *Solving Problems in Scientific Computing Using Maple and MATLAB*. Springer, third edition, 1997. 12