Applikationssoftware und Programmierung

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

5. März 2007

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

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Allgemeines

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

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

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

Die Studierenden sollen daher von Anfang an die Möglichkeit erhalten, das für sie bereitgestellte System in vielen Bereichen ihres Studiums zu verwenden. Außerdem sollen sie auf eine Fülle aufbauender Lehrveranstaltungen bestmöglich vorbereitet sein.

Folgende Lehrveranstaltungen sind stark mit der Benutzung von Computern verbunden:

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

Die Lehrveranstaltung ist eine Chance, die Möglichkeiten, Hilfen aber auch Grenzen kennenzulernen die Computer in der heutigen Zeit im Bereich der Physikausbildung und der For-

schung 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 weitere wichtige Nachrichten folgen sie bitte diesem Link auf dem Wiki des Instituts.

1.2.3 Übung

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

Informationen über Termine , Betreuer und Vorlesungen entnehmen sie bitte den Wiki-Seiten.

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

Die Übungen werden mit dem neu entwickelten Programm MATLABTUTOR abgewickelt (siehe dazu 1.7).

1.2.4 Unterlagen und Dokumentation

Dieses Dokument wird laufend aktualisiert und soll jederzeit über das Wiki des Instituts für Theoretische Physik 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 MATLAB Dokumentation ist verfügbar unter folgender Referenz. Darüber hinaus gibt es auch noch eine einführende Startseite mit weiteren Verzweigungen.

Eine Liste verfügbarer Bücher im pdf-Format direkt vom Erzeuger von Matlab findet man unter PDF-Links . Online Hilfe findet sich unter HTML-Links .

1.2.5 Prüfungen

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

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

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

Für einen Abschluss der Lehrveranstaltung bieten wir folgende Möglichkeiten an:

- Aktive Teilnahme an den Übungen und Abgabe aller Beispiele. Teilnahme an einem Prüfungstermin am Computer, Lösung von Problemen unter Zuhilfenahme aller Unterlagen. Prüfungsgespräch mit dem Vortragenden direkt nach Abgabe.

- Durchführung von Projektarbeiten im Umfeld der Lehrveranstaltung. Das Thema kann bzw. sollte bevorzugt aus Ihrem Interessensgebiet oder aus einer anderen Lehrveranstaltung stammen und mit hier besprochenen Methoden behandelt werden. Ergebnisse können dann auch auf unserer WEB-Seite präsentiert werden. In diesem Fall muss nicht unbedingt an den Übungen teilgenommen werden. Eine vorherige Absprache ist aber unbedingt erforderlich. Diese Möglichkeit richtet sich vor allem an Hörer, die bereits gute Kenntnisse in MATLAB haben.

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

1.2.6 Sprache

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

1.3 Computerzugang für Studierende

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

Im Bereich der Physik finden Sie Informationen unter itp.tugraz.at und itp.tugraz.at/wiki/.

1.3.1 Computer für Studierende im Bereich Physik

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

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

Die Computer sind mit den Betriebssystemen LINUX und der gesamten relevanten Software ausgestattet. Vorgesehen ist die Verwendung sowohl für Übungen als auch für die gesamte studentische Arbeit an Computern. Durch die Verwendung des Betriebssystems LINUX ist auch eine Verwendung von Programmen von außerhalb (siehe externer Zugang) möglich. Auf den Rechnern stehen probeweise auch WINDOWS Programme 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 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 Wiki-Seite des Instituts.

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

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

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

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

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

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

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

1.4 Kommunikation

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

Der Betreuer unserer Computeranlage ist Andreas Hirczy, erreichbar unter **hirczy@itp.tu-graz.ac.at**.

Die gesamte Liste aller Betreuer gibt Auskunft über die Betreuungssituation in den unterschiedlichen Gruppen .

1.5 Dokumente

Dieses Dokument kann sowohl im Ganzen als auch in Form von einzelnen Kapiteln jederzeit herunter geladen werden. Ausserdem liegen auch die Vorlesungsunterlagen auf.

Ein lokaler Spiegel einiger Unterlagen wird von uns über das Wiki angeboten.

Die Erstellung dieses Dokuments und auch der Vortragsunterlagen 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

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

erfolgen.

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

1.6.1 Software Version

Im Computerraum Physik ist derzeit die neueste Version MATLAB 7.1 installiert:

```
MATLAB Version 7.1.0.183 (R14) Service Pack 3
```

1.7 COMPUTINGTUTOR - MATLABTUTOR

1.7.1 Kurzfassung

Bei diesem Produkt handelt es sich um eine Lehr- und Lernsoftware (COMPUTINGTUTOR), die für das Lehren und Lernen von Programmiersprachen geeignet ist. In der gegenwärtigen Ausformung wird die Programmiersprache MATLAB unterstützt (MATLABTUTOR), eine Anpassung an andere Programmiersprachen ist aber technisch bereits vorgesehen. MATLAB wurde als erste Umsetzung deshalb gewählt, da diese Sprache große Verbreitung sowohl in der universitären Lehre und Forschung als auch in der Industrie hat. Über die Bereitstellung der Derzeit ist die verwendete Sprache Deutsch, die Internationalisierung insbesondere für Englisch ist aber technisch bereits vorgesehen. Wichtig für die Vermarktung scheint uns das Schulungspotential in der Wirtschaft zu sein, da MATLAB weltweit in der Industrie große Verbreitung hat.

- Die Software bietet zusätzlich zu Lehr- und Lernsoftwarestandards wie beispielsweise Dokument-, Beispiel- und Benutzerverwaltung zwei wesentliche Erweiterungen: Es kön-

nen vom Lehrenden komplexe Tests zur Überprüfung jener Aufgaben erstellt werden, die vom Studierenden gelöst werden müssen. Diese Tests können alle möglichen Datentypen und Ausgabemöglichkeiten (Graphik, Schirm, File) umfassen und können mit beliebiger Genauigkeit durchgeführt werden. Dadurch ermöglichen sie es, auch sehr komplizierte Aufgabenstellungen automatisiert zu überprüfen. Ausserdem können diese Tests auch bei Prüfungen verwendet werden und stellen dabei durch die vielfältigen Möglichkeiten im Vergleich zu "Multiple Choice Tests" eine große Erweiterung des Einsatzgebietes dar.

Die Studierenden bekommen somit eine sofortige Rückmeldung über den Erfolg ihrer Bemühungen, daher ist die Plattform auch für das Selbststudium hervorragend geeignet. Es wird dadurch aber auch verhindert, dass der Lehrende Zeit mit formaler Überprüfung verbringen muss, welche besser für die Kommunikation mit den Studierenden verwendet werden kann.

- Die Benutzer können die Aufgaben an beliebigen Computern mit Netzzugang erledigen und finden immer die gleiche Umgebung vor. Dies ist vor allem in Hinblick auf die legale Nutzung von teurer Software (hier MATLAB) durch Studenten oder Schulungsteilnehmer wichtig. In diesem Punkt ist unsere Lösung Server-basierend und erlaubt die effiziente Nutzung der Lizenzen, die in einer Institution für die Lehre bereitstehen.

Das Gesamtpaket ermöglicht daher den Wechsel zwischen Präsenzphasen im Lehrsaal und der Arbeit zu Hause ohne Unterschiede in der Programmoberfläche oder den Inhalten und ohne lizenzrechtliche Schwierigkeiten für Studierende bzw. Schulungsteilnehmer.

1.7.2 Beschreibung

Ziel des Projektes ist die Bereitstellung von Lehr- und Lernsoftware für Programmiersprachen zur Ausbildung von Studenten, Schülern, aber auch für externe Kursangebote. Das Grundkonzept eignet sich für alle Programmier- und Skriptsprachen. Bedingt durch den speziellen Bedarf an Lehrveranstaltungen im Studium "Technische Physik" an der TU Graz wurde für die konkrete Umsetzung vorerst die Programmiersprache MATLAB ausgesucht.

1.7.2.1 Aus der Sicht des Lernenden

Aus der Sicht des Lernenden handelt es sich um eine Software die auf beliebigen Plattformen (Windows, Unix, Mac, ...) funktioniert. Unter der Voraussetzung, dass ein Zugang zum Internet vorhanden ist, soll der Lernende keinen Unterschied zwischen der Arbeit im Computerraum der Universität oder Schule und seinem Arbeitsplatz zu Hause feststellen. Konkret heisst das, dass der Studierende an jedem Platz Zugang zu den gleichen Informationen, Daten, eigenen Beiträgen, aber auch zu lizenzpflichtiger und für Studenten oft zu teurer Software (in diesem Fall MATLAB) hat.

Das Programm MATLABTUTOR bietet dem Studierenden:

- eine Entwicklungsumgebung für MATLAB natürlich mit perfekter Sprachunterstützung und Links zu Sprachdefinition, Beispielen und Anmerkungen des Lehrenden;
- die Möglichkeit Ideen auszuprobieren und deren Ergebnisse zu analysieren oder graphisch darzustellen;

- die Möglichkeit den gesamten Kursinhalt in einer Baumstruktur zu überblicken und über den Status der einzelnen Übungen und der zugehörigen Beispiele (getestet, abgegeben, korrekt, Anmerkung eines Tutors, usw.) immer informiert zu sein;
- ein Umfeld, in dem er die Beschreibung der Aufgabe, verschiedene Hinweise und Ergänzungen dazu, den Editor für die Fertigstellung der Aufgabe und Resultate der automatisierten Tests im Blickfeld hat;
- ein automatisiertes Testsystem für Übungsbeispiele, welches mit vordefinierten Tests sofort die Richtigkeit der Lösung bestätigt oder für einzelne Variablen oder Graphikelemente Fehler und Hinweise liefert;
- ein automatisiertes Abgabesystem für Übungsbeispiele, das mit Kommunikationsmöglichkeiten zwischen Lernendem und Lehrendem versehen ist.

Der Zugang zu allen Unterlagen an jedem beliebigen Arbeitsplatz ermöglicht das effiziente Arbeiten sowohl während der Präsenzzeit in der Übungseinheit als auch zu Hause. Es besteht keine Notwendigkeit teure Software zu kaufen und auch keine Verführung diese illegal zu benutzen. Die automatisierten Tests und die automatisierte Abgabe ermöglicht die Konzentration auf die Aufgabe mit sofortigem Feedback über die formale Richtigkeit.

Damit eignet sich die Software einerseits für klassische Übungen mit Präsenzzeit und Aufgabenstellungen für zu Hause und andererseits für weitgehendes Selbststudium mit geringerer Interaktionszeit zwischen Lehrendem und Studierendem. Dazu ist vor allem die sofortige Rückmeldung mit Hilfe der automatisierten Tests und die problemlose Verwendung ausserhalb der Lehrsäle wichtig.

1.7.2.2 Aus der Sicht des Lehrenden

In Ergänzung zu den Möglichkeiten der Studierenden haben die Lehrenden natürlich die Möglichkeit

- Lehrveranstaltungen, Übungseinheiten und Beispiele mit zugehörigen Referenzlösungen und Tests zu erstellen und dabei einfache Musterbeispiele und Voragen zu verwenden;
- Übungen aus einem reichen Fundus von bereits bestehenden Beispielen zusammenzustellen und diese als freiwillige Einführung, Pflichtbeispiel oder anspruchsvolle Ergänzung zu charakterisieren bzw. ihren Schwierigkeitsgrad zu definieren;
- Referenzlösungen, wenn gewünscht, nach einer gewissen Zeit den Studierenden zugänglich zu machen;
- Studierende und Gruppen zu verwalten und mit ihnen auf elektronischem Weg zu kommunizieren;
- Abgaben und Tests von Studierenden zu korrigieren und mit Anmerkungen zu versehen;
- Statistiken von Studierenden mit Beispielabgaben, Einhaltung von Terminen und Fehlstunden zu erstellen um somit die Bewertung zu erleichtern;
- Prüfungen zu erstellen und abzuwickeln, wobei die entsprechenden Beispiele nur zeitlich eingeschränkt während der Prüfungszeit zur Verfügung stehen.

Der große Vorteil des Programms MATLABTUTOR besteht darin, dass durch automatisierte Tests die formale Richtigkeit der Abgabe festgestellt wird und Fehler sofort sichtbar sind. Daher geht für diese Überprüfung keine Zeit verloren und der Lehrende kann sich auf Gespräche mit den Übungsteilnehmern konzentrieren und ihnen Hilfestellungen geben, wie formal richtige Beispiele effizienter, besser oder schöner erstellt werden können. Damit ergibt sich eine Verbesserung des Lernerfolgs.

Durch die Tests können auch sehr komplexe Ergebnisse überprüft werden. Dies umfasst alle MATLAB Datentypen, alle denkbaren Graphikelemente, Ausgaben auf Schirm oder in Files, verbotene oder unbedingt notwendige Befehle, Stilfragen wie zu erstellenden Hilfetext oder Syntaxregeln (z.B. Zeilenabschluss). In den Tests können komplexe Analysen durchgeführt werden und die Ergebnisse von Referenzlösungen und Lösungen von Studierenden gegenüber gestellt werden.

Die einzelnen Aufgaben können beliebig kompliziert gestaltet werden. Sie können voneinander abhängen, d.h., ein Beispiel erfordert die vorherige Fertigstellung eines anderen Beispiels. Es können Daten oder Programme beigestellt werden, die für die Lösung des Problems notwendig sind, z.B. Files mit Daten für Datenanalyse. Bei den Aufgaben und Tests können auch Zufallszahlen verwendet werden, wobei das System sicherstellt, dass die Vergleichbarkeit von Referenz- und Studentenlösung trotzdem gewährleistet ist. Damit können variable Inputparameter für Funktionen generiert werden, die sicherstellen, dass die Lösung nicht nur mit sehr partikulären Inputs funktioniert.

Da die Komplexität der durchzuführenden Tests nur durch die im Hintergrund arbeitende Programmiersprache beschränkt wird, bietet sich dem Lehrenden die Möglichkeit die Anwen-

dungsmöglichkeiten der Software selbst zu erweitern bzw. auf seine Bedürfnisse anzupassen. Somit wäre auch eine Erweiterung in Richtung anderer formal beurteilbarer Aufgabenstellungen denkbar, z.B. automatisierte Überprüfung von Formelwissen

1.7.3 Konkretisierung

Die Umsetzung auf Anwenderseite basiert auf JAVA, ECLIPSE und der unterstützten Programmiersprache (hier MATLAB). Im Hintergrund werden Technologien wie Tomcat-Server, Datenbank (MYSQL), Secure Shell, das Filesystem AFS und Kerberos Authentifizierung verwendet. Unterstützte Dokumenttypen für die es derzeit Viewer und Editoren gibt, sind LATEX, PDF, HTML und MATLAB. Die Sprache ist derzeit Deutsch, aber eine Internationalisierung ist technisch vorgesehen.

Im SS 2006 wurde ein Alpha-Test von MATLABTUTOR im Rahmen der Lehrveranstaltung "Applikationssoftware und Programmierung" mit zirka 70 Studenten in 5 Gruppen durchgeführt. Ausserdem wurden alle Prüfungstermine für diese Studenten mit MATLABTUTOR abgewickelt. Trotz technischer Schwierigkeiten, die natürlich aufgetreten sind, war die Erfahrung durchwegs positiv. Auf Grund dieser Erfahrungen wurden Teile verbessert, ergänzt bzw. umgestellt und der Beta-Test findet im SS 2007 im Rahmen der gleichen Lehrveranstaltung statt. Danach soll es für andere Vortragende an der TU Graz und anderen Universitäten angeboten werden. Welches Lizenzmodell hier gewählt werden soll, ist derzeit unklar.

Derzeit wird über eine mögliche Verbindung zum "Teach Center" der TU Graz und über die Verwendung des Kurses im Projekt "Life Long Learning" der TU Graz nachgedacht. Eine voll-

ständige Integration in das ”Teach Center” scheint derzeit schwierig, da es sich beim MATLAB-TUTOR um eine ”Rich Client Application” handelt, die nicht einfach in einem Browser laufen kann. Angestrebt wird aber eine Darstellung aller Inhalte und Beschreibungen im ”Teach Center”, wobei aber die Bearbeitung der Beispiele und die Abgabe in der ”Rich Client Application” durchgeführt werden müssen.

Die Anbindung des Kurses an das Projekt ”Life Long Learning” ist deshalb von großem Interesse, da MATLAB in der Wirtschaft und Industrie verwendet wird und dort Schulungsbedarf besteht.

Eine Ausweitung auf andere Programmiersprachen ist im Prinzip vorgesehen und technisch möglich. Das Team konzentriert sich aber derzeit ausschließlich auf MATLAB, da diese Programmiersprache im Studium der ?Technischen Physik? verwendet wird in dessen Rahmen die Software entwickelt wird. Hier wären sicher Kooperationen mit anderen Lehrenden bzw. mit Experten für die jeweiligen Programmiersprachen notwendig.

Kapitel 2

Basis Syntax in MATLAB

2.1 Variablen und Zuweisung von Werten

Neben der Möglichkeit MATLAB als eine Art überdimensionalen Taschenrechner zu benutzen

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

kann man Ergebnisse auch benannten Größen (Variablen) zuweisen. Das Zeichen für Zuweisung (assignment) ist das = Zeichen. Wird kein = verwendet, wird die Rechnung durchgeführt und das Ergebnis auf der Variablen `ans` (answer) gespeichert.

Gespeicherte Größen können in der Folge für weitere Berechnungen herangezogen werden.

```
a = 3 * (5 + 8)
a = (a - 1) / 4
b = sin(0.5)
res_1 = (b + 1) / b - 1;      % Was ist der Unterschied
res_2 = (b + 1) / (b - 1);    % zwischen den zwei Zeilen?
```

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

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

Der Strichpunkt am Ende einer Programmanweisung regelt nur die Ausgabe am Schirm und hat nichts mit der Berechnung an sich zu tun.

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

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

So berechnet der Befehl

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

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

```
x   = [-1:0.1:1];
y_1 = sinh(x); y_2 = cosh(x);
plot(x,y_1,x,y_2)
```

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

Bei Fehlern in MATLAB-Skripts oder MATLAB-Funktionen werden zusätzlich Zeilen- und Spaltennummern angegeben.

FEHLER	FEHLERMELDUNG
`a = 3+`	`Error: Expression or statement is incom-` `plete or incorrect.`

`a = (3+4`	`Error: Expression or statement`
	`is incorrect-possibly unbalanced`
	`(, \{, or \[.`

`a = sin()`	`Error: Error using ==> sin` `Not enough input arguments.`
`sin(1,2,3)`	`Error using ==> sin` `Too many input arguments.`
`sin(1]`	`Error: Unbalanced or misused parentheses` `or brackets.`
`a = six(1)`	`Undefined command/function ”six”.`
`s = 'Winfried`	`Error: A MATLAB string constant is not` `terminated properly.`
`3a = sin(1)`	`Error: Unexpected MATLAB expression.`

`3*a = sin(1)`	`Error: The expression to the left of the`
	`equals sign is not a valid target for an`
	`assignment.`

`a = b = 5`	`Error: The expression to the left of the`
	`equals sign is not a valid target for an`
	`assignment.`

`a = 5 .+ 2`	`Error: Unexpected MATLAB operator.`

Gültige Variablennamen in MATLAB müssen mit einem Buchstaben beginnen und dürfen nur Buchstaben, Zahlen und als einziges Sonderzeichen den Untersstrich _ verwenden. Groß- und Kleinbuchstaben werden zumindest unter LINUX unterschieden.

Die Verwendung von Umlauten ist unter WINDOWS möglich, sollte aber vermieden werden, da solche Programme unter LINUX dann nicht funktionieren. Äusserst ungeschickt ist es auch Namen von MATLAB-Funktionen und vordefinierten Konstanten 2.2 zu verwenden.

GÜLTIG	UNGÜLTIG	UNGESCHICKT
`a a3 a_3`	`3a 3_a a*`	`Maß ö3 ö_3`
`Sin SIN MAX`	`Sin( Sin() Max+`	`sin max abs`
`k l m`	`"k" k! k#`	`i j pi`
`Resultat_1`	`Resultat[1]`	`Lösung`

Werden MATLAB-Befehle als Namen für Variablen verwendet, schafft man ein Problem dadurch, weil diese neue Bedeutung in der Hirachie höher liegt als die ursprüngliche Bedeutung, d.h., dass die neue Bedeutung Vorrang hat. Schreibt man z.B. `sin=5`, verliert `sin` die Bedeutung als Sinus-Funktion und liefert immer den Wert `5`.

Manchmal ist die Gefahr, dass dies unendeckt bleibt sehr groß:
`sin(1)` liefert dann einfach `5` statt `0.8415`
Manchmal treten Fehler auf:

```
sin(2)
 Error: Index exceeds matrix dimensions.
sin(pi)
 Error: Subscript indices must either be real
 positive integers or logicals.
```

Behoben werden kann das Problem nur durch das Löschen der Variablen (`clear`):
`clear sin` oder `clear('sin')`

Gefährlich ist auch folgender Fall: Durch den Befehl `i=1` verliert die Variable `i` ihre Bedeutung als imaginäre Konstante und `3+2*i` liefert den falschen Wert `5`.

Diejenigen Fehler, die eine Beendigung des Programms zur Folge haben und die damit eine Error-Meldung liefern, sind grundsätzlich leichter zu finden und zu beheben als Fehler, die den Programmablauf nicht stoppen, sondern nur zur Folge haben, dass falsch weiter gerechnet wird.

2.2 Mathematische Konstanten

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

KONSTANTE	BEDEUTUNG	WERT
`eps`	Relative Genauigkeit von Fließkommarechnungen.	$2.2204 \cdot 10^{-16}$
`i`, `j`	Imaginäre Einheit.	$\sqrt{-1}$
`inf`, `Inf`	Unendlich.	∞
`nan`, `NaN`	Not A Number - Keine Zahl im herkömmlichen Sinn. Entsteht z.B. durch $\frac{0}{0}$, $\frac{\infty}{\infty}$, $0 \cdot \infty$ und durch jede Operation mit `nan`.	nan
`pi`	Verhältniss zwischen Umfang des Kreises und seines Durchmessers.	3.14159
`realmax`	Größte positive Fließkommazahl.	$1.79769 \cdot 10^{+308}$
`realmin`	Kleinste positive Fließkommazahl.	$2.22507 \cdot 10^{-308}$
`intmax`	Größte ganze Zahl (`int32`).	2147483647
`intmin`	Kleinste ganze Zahl (`int32`).	-2147483648

Die vorliegende Liste ist beschränkt auf die am häufigsten verwendeten numerischen Datentypen in MATLAB, `double` und `int32` (32 Bit Integer mit Vorzeichen).

Für Fließkommazahlen gibt es neben dem Datentyp `double` (8 Byte) auch den Datentyp `single` (4 Byte). Will man die Fließkommazahlen im Datentyp `single`, muss man schreiben `eps('single'),inf('single'),nan('single'),` bzw. `realmax('single')` und `realmin('single')`.

Für `pi` und andere Zahlen muss man die Befehlsform von `single`, nämlich `single(pi)` verwenden. Die numerischen Datentypen sind in 8 zusammengefasst. Die ganzzahligen Datentypen gibt es von `int8` (1 Byte) bis `int64` (8 Byte). Die richtige Verwendung hier wäre also z.B., `intmax('int8')`.

2.3 Wichtige Befehle

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

BEFEHL	BEDEUTUNG
`quit`	Beendet MATLAB.
`exit`	Beendet MATLAB.
`clc`	Löscht MATLAB-Kommandofenster.
`home`	Löscht MATLAB-Schirm und positioniert den Cursor links oben (man kann aber den Balken verwenden um alte Inhalte zu sehen.
`diary`	Schreibt Befehle und Ergebnisse in einem File mit.
`save`	Speichert Inhalte des Workspaces in einem File.
`format`	Beeinflusst Outputformat.
`system`	Führt Befehl im zugrundeliegenden Betriebssystem aus.
`clear`	Löscht Variable im Workspace (z.B.: `clear all`).
`close`	Schließt Graphikfenster (z.B.: `close all`).
`who`, `whos`	Listed Variablen im Workspace auf.
`exist`	Überprüft, ob ein Name bereits existiert.

2.4 Möglichkeiten für Hilfe in MATLAB

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

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

2.5 Spezielle Zeichen - Special Characters

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

2.5.1 Klammern

2.5.1.1 Runde Klammern - Parenthesis

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

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

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

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

2.5.1.2 Eckige Klammern - Brackets

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

KLAMMER	BEDEUTUNG	BEISPIEL

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

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

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

2.5.1.3 Geschwungene Klammern - Curly Braces

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

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

2.5.2 Punkt - Dot

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

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

2.5.3 Komma und Strichpunkt - Comma and Semicolon

Komma und Strichpunkt fungieren im Wesentlichen als Trennzeichen:

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

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

```
a = 3,4
```

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

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

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

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

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

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

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

2.5.4 Doppelpunkt - Colon

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

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

2.5.5 Hochkomma - Quotation Mark

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

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

Details zu diesen Beispielen findet man im Abschnitt 10. Die hier verwendeten Befehle `num2str` und `disp` dienen zur Umwandlung von Zahlen in Zeichenketten und zur Darstellung von Ergebnissen im MATLAB-Kommandofenster.

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

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

Diese Anwendung ist ident mit den Befehlen `ctranspose`, bzw. mit `transpose`. Details zum Bearbeiten von Matrizen findet man im Abschnitt 6.

2.5.6 Prozent und Rufzeichen - Percent and Exclamation Point

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

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

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

In weiterer Folge kann man dann Programmzeilen kommentieren,

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

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

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

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

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

Einige wichtige Systembefehle sind aber auch direkt in MATLAB vorhanden:

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

2.5.7 Operatoren

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

TYP	VERWEIS	ZEICHEN
Arithmetisch - Matrizen	6	`+ - * / \ ^`
Arithmetisch - Elemente	4	`+ - .* ./ .\ .^`
Transponieren	6	`' .'`
Vergleich	4.2	`== ~= < <= > >=`
Logisch	4.3	`~ & |`

2.6 Schlüsselwörter - Keywords

In MATLAB sind eine Reihe von Schlüsselwörtern definiert, die im Wesentlichen zu Steuerkonstrukten 7 gehören. In alphabetischer Reihenfolge sind dies:

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

2.7 MATLAB-Skripts und MATLAB-Funktionen

MATLAB kennt zwei Typen von Programmeinheiten, Skripts und Funktionen, die im Detail im Abschnitt 9 besprochen werden.

MATLAB-Skripts sind Programme, die im MATLAB-Arbeitsbereich (Workspace) ablaufen und keine Übergabeparameter kennen. Ihnen sind alle definierten Variablen im Workspace (`who`) bekannt, zwei unterschiedliche Skripts können also wechselseitig Variablen benutzen oder überschreiben. Das kann einerseits praktisch sein, birgt aber andererseits auch große Gefahren unerwünschter Beeinflussung. Will man sicher sein, dass Skripts in einem "reinen" Workspace ablaufen, muss man den Befehl `clear all` verwenden.

MATLAB-Funktionen 9.1 hingegen werden in einem eigenen Arbeitsbereich abgearbeitet. Hier gibt es also keine unerwünschten Querverbindungen. Ihre Kommunikation mit Skripts (oder anderen Funktionen) erfolgt durch sogenannte Übergabeparameter,

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

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

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

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

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

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

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

2.7.1 Einfache Beispiele

Zur Einführung werden hier zwei einfache Beispiele gezeigt, wobei man weiterführende Beispiele im Abschnitt 9.6 findet.

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

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

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

Das entsprechende Skript zum Aufruf der Funktion könnte so aussehen:

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

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

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

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

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

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

Je nach Wert der der Variablen `typ` kann die Funktion nun zwei verschiedene Aufgaben erledigen. Eine genaue Beschreibung von Steuerstrukturen finden Sie im Abschnitt 7, Details zum Befehl `switch` finden Sie im Abschnitt 7.2.2.

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

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

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

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

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

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

Anmerkung: Will man die Eingabe des Typs noch weiter erleichtern (Groß- oder Kleinschreibung, nur Anfangsbuchstabe(n), kann man die `switch`-Konstruktion verbessern. Man kann z.B. alle Zeichenketten mit dem Befehl `lower` in Kleinbuchstaben verwandeln. Details zur Verwendung von Zeichenketten findet man im Abschnitt 10. Die Konstruktion

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

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

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

2.8 Einfache MATLAB-Skripts

Zur Vorbereitung auf die Übung gibt es hier noch zwei einfache Beispiele, wobei sich das eine eher mit Skalaren und das andere eher mit Vektoren beschäftigt.

Kapitel 3

Arrays

3.1 Konzept

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

So kann man den Inhalt verstehen als,

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

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

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

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

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

3.2 Eigenschaften von Arrays

Wichtige Eigenschaften von Arrays sind neben ihrem Inhalt,

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

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

3.3 Hilfe für Arrays

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

MATLAB HELP: `ndims`

```
Number of dimensions.

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

In Ergänzung dazu lautet die Hilfe für `size`:

MATLAB HELP: `size`

```
Size of matrix.

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

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

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

---

bzw. für `length`:

MATLAB HELP: `length`

```
Length of vector.

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

3.4 Erzeugung von Matrizen

Arrays bzw. Matrizen können auf vielfältige Weise erzeugt werden:

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

3.4.1 Explizite Eingabe

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

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

kann auf folgende Weise durchgeführt werden:

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

wobei hier eine Zuweisung der Werte auf eine Variable mit dem Namen `A` erfolgt.

Man muss dabei folgende Regeln beachten:

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

Tabelle 3.2: Doppelpunkt Notation zur Erzeugung von Vektoren

Operator	Alternative	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`

3.4.2 Doppelpunkt Notation

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

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

Leere Arrays: Symbolisiert durch `[]`.

Logisches UND: Verwendetes Symbol `&`.

MATLAB Beispiel

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

Einige gültige und ungültige Beispiele für die Doppelpunkt Notation.

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

X = 1:2:5
    1     3     5

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

X = 5:-2:1
    5     3     1

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

Tabelle 3.3: MATLAB Befehle zum Erzeugen von Matrizen

`zeros(m)`	Erzeugt eine m×m Nullmatrix
`zeros(m,n)`	Erzeugt eine m×n Nullmatrix
`ones(m)`	Erzeugt eine m×m Matrix mit lauter Einsen
`ones(m,n)`	Erzeugt eine m×n Matrix mit lauter Einsen
`nan(m)`	Erzeugt eine m×m Matrix mit dem Wert NaN
`nan(m,n)`	Erzeugt eine m×n Matrix mit dem Wert NaN
`inf(m)`	Erzeugt eine m×m Matrix mit dem Wert Unendlich
`inf(m,n)`	Erzeugt eine m×n Matrix mit dem Wert Unendlich
`eye(m)`	Erzeugt eine m×m Einheitsmatrix
`eye(m,n)`	Erzeugt eine m×n Einheitsmatrix

3.4.3 Interne Befehle zur Erzeugen von Matrizen

Es gibt eine Reihe von Befehlen zur einfachen Erzeugung von Matrizen, die in 3.3, 3.4 und 3.5 zusammengefasst sind.

3.4.3.1 Einsen und Ähnliches

Am Beispiel von `ones` sein hier auf einige Besonderheiten hingewiesen. Es führt immer wieder zu Verwirrung, dass `ones(m)` eine m×m-Matrix und nicht einen Spaltenvektor (m×1-Matrix) erzeugt. Will man einen Spaltenvektor mit m-Einsen muss man also `ones(m,1)` schreiben.

Der Befehl `ones(m,n)` liefert das gleiche Ergebnis wie der Befehl `ones([m,n])`, der Input kann also aus einem Vektor von ganzen Zahlen bestehen. Es ist also in MATLAB-Manier eine einfache Schreibweise für die händische Eingabe möglich, `ones(m,n)`, aber auch eine, die für das Programmieren praktischer ist, `ones([m,n])`. Die zweite Schreibweise ermöglicht nämlich, dass man eine Matrix exakt gleicher Größe wie eine bereits bestehende erzeugt:

```
x = rand(3,5);      % (3 x 5)-Matrix aus Zufallszahlen
y = ones(size(x)); % gleich grosse Matrix mit Einsen
```

Das Ergebnis von `size(x)`, nämlich der Vektor `[3,5]`, ist dabei der Input für den Befehl `ones`.

Natürlich sind die Befehle nicht auf zweidimensionale Matrizen beschränkt, sondern es sind auch höher-dimesionale Matrizen möglich. Z.B., liefert `ones(m,n,p,q)` bzw. `ones([m,n,p,q])` eine 4-dimensionale Matrix der Größe `[m,n,p,q]`.

Ist eine oder mehrere der Größenangaben Null oder negativ, dann erhält man ein leere Matrix, also:

```
x = ones(3,4,0);    % leere Matrix
```

```
size(x);            % liefert [3,4,0]
y = ones(0);        % gleichbedeutend mit y = [];
size(y);            % liefert [0,0]
```

Am ersten Beispiel sieht man, dass also eine leere Matrix durchaus eine Größe haben kann, die in manchen Dimensionen von Null verschieden ist. Die Tatsache, dass Null bzw. negativer Input keinen Fehler liefert, ist für das Programmieren durchaus hilfreich. Will man z.B. erreichen, dass ein Zeilenvektor durch Anhängen von Nullen eine Mindestlänge erreicht, kann man das in MATLAB so machen:

```
L = 5;                              % gewuenschte Mindestlaenge
x = 1:3                             % Vektor [1,2,3]
x = [x,zeros(1,L-size(x,2))]; % (5-3=2) Nullen angehaengt
x                                   % liefert [1,2,3,0,0]
y = 1:6                             % Vektor [1,2,3,4,5,6]
y = [y,zeros(1,L-size(y,2))]; % (5-6=-1) Nullen angehaengt
y                                   % liefert [1,2,3,4,5,6]
```

Als zusätzlichen Input kann man den gewünschten Datentyp übergeben, so liefert `ones([m,n],'int3` das Ergebnis nicht im Standard-Datentyp `double` sondern im Integer-Datentyp `'int32'`. Leider funktioniert das nicht mit dem Datentyp `logical`. In diesem Fall muss man

```
L = logical(ones(3,4))
```

Tabelle 3.4: MATLAB Befehle zum Erzeugen von Matrizen

`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)

schreiben. Bei `nan` und `inf` kann man nur die Datentypen `double` und `single` angeben. Näheres über Datentypen findet man in 8.

3.4.3.2 Gleicher Abstand

Die Befehle `linspace` bzw. `logspace` liefern also

```
x_lin = linspace(0,4,5);   % [0,1,2,3,4]
x_log = logspace(0,4,5);   % [1,10,100,1000,10000]
log10(x_log);              % [0,1,2,3,4]
```

daher hat im Falle von `logspace` der Zehnerlogarithmus den äquidistanten Abstand hier zwischen 10^0 und 10^4. Soll das Gleiche für eine andere Basis des Logarithmus erreicht werden, muss man sich mit `linspace` behelfen. Dies soll hier am Beispiel der Basis 2 erläutert werden:

```
x_log2 = 2.^linspace(0,4,5); % [1,2,4,8,16]
log2(x_log2)                 % [0,1,2,3,4]
```

3.4.3.3 Zufallszahlen

Der Befehl `rand` liefert Matrizen mit gleichverteilten Zufallszahlen im offenen Interval $(0, 1) = \{x \in \mathbb{R} \mid 0 < x < 1\}$. Will man gleichverteilte Zufallszahlen im offenen Intervall (a, b) erzeugen, kann man diese mit

```
m = 100; a = 5; b = 10;
r = a + (b - a) * rand(1,m);
```

erreichen und bekommt hier einen Zeilenvektor mit 100 Zufallszahlen. Benötigt man ganzzahlige Zufallszahlen im abgeschlossenen Intervall $(1, n) = \{x \in \mathbb{N} \mid 1 \leq x < n\}$ kann man das durch den Befehl

```
n = 10;
r = ceil( n * rand(1,m) );
```

erreichen. Sollen die ganzzahligen Zufallszahlen im abgeschlossenen Intervall $(a, b) = \{x \in \mathbb{N} \mid a \leq x < b\}$ liegen erreicht man das durch die entsprechende Verschiebung

```
a = 2; b = 10;
r = ceil( (b -a + 1) * rand(1,m) ) + a - 1;
```

Der Befehl `randn` liefert Matrizen mit normalverteilten Zufallszahlen. Der Mittelwert der Verteilung liegt bei Null und die Standardabweichung ist Eins. Benötigt man einen anderen Mittelwert x_m und eine andere Standardabweichung σ kann man das Ergebnis leicht umformen

```
x_m = 2; sigma = 0.5;
r = x_m + sigma * randn(1,m);
```

3.4.3.4 Diagonalen

Der Befehl `diag` kennt zwei Arten der Verwendung. Erstens kann man aus einem Vektor eine Matrix erzeugen in deren (Neben-)Diagonale der Vektor steht. Gegeben sei ein Vektor v

$$v = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} .$$

Mit dem Befehl `D=diag(v)` erzeugt man eine Matrix mit dem Vektor v in der Diagonale

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} .$$

Tabelle 3.5: 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).
`blkdiag(a,b,...)`	Erzeugt eine blockdiagonale Matrix. a,b, . . . sind Matrizen.
`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).
`repmat(a,m,n)`	Erzeugt aus einer Matrix a eine neue Matrix durch Replikation in Zeilenrichtung (m-mal) und Spaltenrichtung (n-mal).

Wählt man die 2-te Nebendiagonale, `D=diag(v,2)`, bekommt man

$$D = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} ,$$

wählt man hingegen eine negative Zahl für die Bezeichnung der Nebendiagonale, `D=diag(v,-2)`, dann erhält man

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{bmatrix} .$$

Im zweiten Mode kann man aus einer Matrix eine bestimmte Diagonale extrahieren. Mit D aus dem letzten Beispiel bekommt man mit dem Befehl `s=diag(D,-2)` den Spaltenvektor

$$s = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} .$$

Mit `blkdiag` kann man Blöcke entlang einer Diagonale anordnen. So liefert der Befehl `bblk-diag(ones(1),2*ones(2),3*ones(3))` die Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 & 3 \\ 0 & 0 & 0 & 3 & 3 & 3 \\ 0 & 0 & 0 & 3 & 3 & 3 \end{bmatrix}.$$

3.4.3.5 Dreiecke

Mit den Befehlen `triu` (upper) bzw. `tril` (lower) kann man aus einer Matrix obere bzw. untere Dreiecksmatrizen erzeugen. Dies sei hier ausgehend von

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

demonstriert. Mit `U=triu(D)` bekommt man

$$U = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 0 & 6 & 10 & 14 \\ 0 & 0 & 11 & 15 \\ 0 & 0 & 0 & 16 \end{bmatrix}$$

und mit `L=tril(D)` bekommt man

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 6 & 0 & 0 \\ 3 & 7 & 11 & 0 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

als Ergebnis.

Mit der Angabe eines zweiten Parameters verschiebt man wie bei `diag` die Grenze zwischen dem Bereich wo die Werte erhalten sind und dem Bereich mit Nullen. So liefert `U=triu(D,1)`

$$U = \begin{bmatrix} 0 & 5 & 9 & 13 \\ 0 & 0 & 10 & 14 \\ 0 & 0 & 0 & 15 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

bzw. `L=tril(D,1)`

$$L = \begin{bmatrix} 1 & 5 & 0 & 0 \\ 2 & 6 & 10 & 0 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

und mit negativen Zahlen verschiebt sich die Grenze in den unteren Bereich.

3.4.3.6 Vervielfältigung

Mit dem Befehl `repmat` kann man den Inhalt von Matrizen in bestimmte Richtungen vervielfältigen. Am leichtesten sieht man das Prinzip, wenn man von einem Vektor ausgeht. Der Zeilenvektor z sei

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} .$$

Der Befehl `repmat(z,2,1)` bzw. `repmat(z,[2,1])` liefert

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} ,$$

wohingegen `repmat(z,[1,2])`

$$\begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \end{bmatrix}$$

liefert. Schlussendlich ergibt `repmat(z,[2,2])`

$$\begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \end{bmatrix} .$$

3.4.3.7 Netz von Zahlen

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

sich folgende Matrizen:

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

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

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

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

3.4.4 Lesen und Schreiben von Daten

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

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

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

3.5 Veränderung und Auswertung von Matrizen

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

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

Der numerische Inhalt von Matrizen muss nicht nur aus reellen Zahlen bestehen, sondern kann auch komplexe Werte enthalten. Dafür ist keine spezielle Deklaration notwendig, MAT-

LAB führt diese automatisch beim ersten Auftreten von komplexen Elementen in einer Matrix durch.

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

MATLAB Beispiel

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

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

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

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

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

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

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

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

MATLAB Beispiel

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

`TRANSPOSE` is the non-conjugate transpose.

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

`CTRANSPOSE` is the complex conjugate transpose.

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

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

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

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

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

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

MATLAB Beispiel

Summation und kummulative Summation in Matrizen.

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

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

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

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

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

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

Y=sum(X)
     3     5     7

Y=sum(X,2)
     3
    12

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

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

MATLAB Beispiel

Multiplikation und kummulative Multiplikation in Matrizen.

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

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

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

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

Y=prod(X)
     0     4    10

Y=prod(X,2)
     0
    60

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

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

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

1. Ist eine Richtung vorgegeben, `sum(X,2)`, erfolgt die Operation in Richtung dieser Dimension.

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

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

Summe der Diagonalelemente der Matrix X:

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

Summe der Elemente in der zweiten Diagonale der Matrix X:

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

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

3.6 Zugriff auf Teile von Matrizen, Indizierung

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

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

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

In der Folge wird nun auf die verschiedenen Möglichkeiten der Indizierung eingegangen. In Tabelle 3.6 werden die einzelnen Regeln erläutert, und in 3.7 die Zuweisung von Werten gezeigt, und in 3.8 der Zugriff auf bestimmte Regionen gezeigt.

Tabelle 3.6: 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 Umrechnung zwischen dem linearen Index und mehrfachen Indices erfolgt mit den Befehlen `ind2sub` und `sub2ind`:

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

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

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

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

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

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

Tabelle 3.7: Zuweisung von Werten an bestimmten Positionen eines Arrays

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

Tabelle 3.8: Zugriff auf bestimmte Positionen eines Arrays

`X` 1 2 3 4 5 6 7 8 9 10 11 12	`X(3,2)` 10	`X(:,2)` 2 6 10
`X(2,:)` 5 6 7 8	`X(:,:)` 1 2 3 4 5 6 7 8 9 10 11 12	`X(:)` 1 5 ⋮ 8 12
`X(:,1:2:4)` 1 3 5 7 9 11	`X(1:2:3,:)` 1 2 3 4 9 10 11 12	`X(1:2:3,1:2:4)` 1 3 9 11
`X(7:10)` 3 7 11 4	`X(1:2,3)` 3 7	`X(2,1:3)` 5 6 7

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

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

3.6.1 Logische Indizierung

In Ergänzung zur normalen Indizierung erlaubt MATLAB auch die sogenannte logische Indizierung mit Arrays die nur die Werte 1 (entspricht TRUE) bzw. 0 (entspricht FALSE) enthalten. Dadurch ist auch der Zugriff auf völlig ungeordnete Bereiche möglich (Tab. 3.9).

Wichtig dabei ist Folgendes:

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

Tabelle 3.9: Zugriff mit Hilfe logischer Indizierung

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

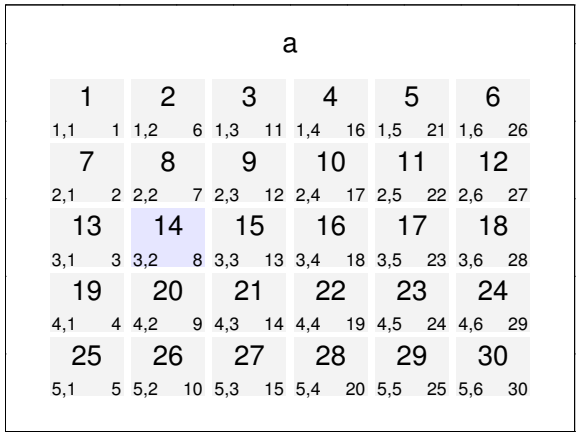
- Bei jeder Zuweisung muss entweder die Anzahl der Werte gleich sein wie die Anzahl der ausgewählten Positionen, oder ein Skalar wird auf eine beliebige Anzahl von Positionen zugewiesen.
- Ist man nur an den Positionen interessiert, kann man mit `I = find(L)` die linearen Indices, bzw. mit `[m,n] = find(L)`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

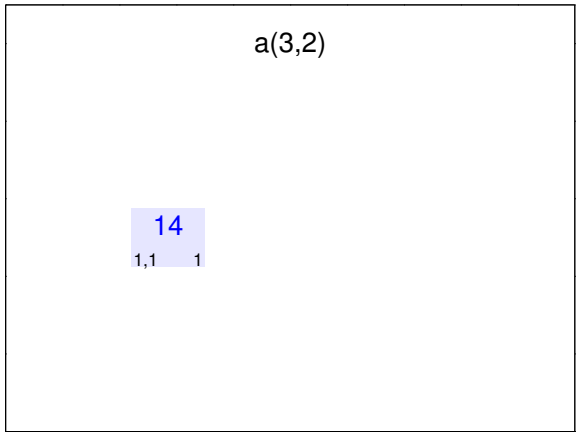
3.6.2 Beispiele zur Indizierung

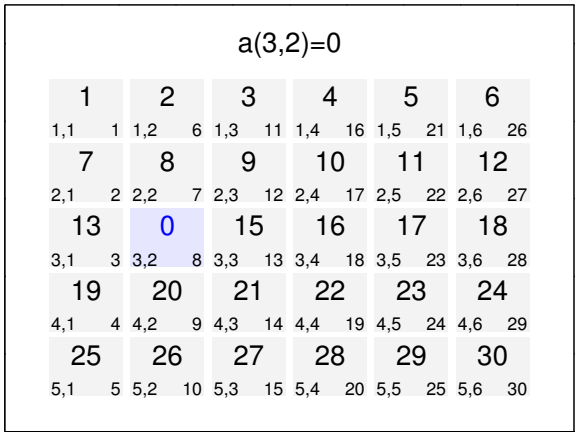
Die vorliegenden Beispiele demonstrieren die Indizierung in MATLAB an Hand von 2-dimensionalen Matrizen. Jedes Element enthält dabei in der unteren linken Ecke den 2-D Index und in der rechten unteren Ecke den linearen Index. Erfolgt eine Zuweisung, bleibt die Form der Matrix erhalten, erfolgt jedoch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

3.6.2.1 Zweidimensionale Indizierung

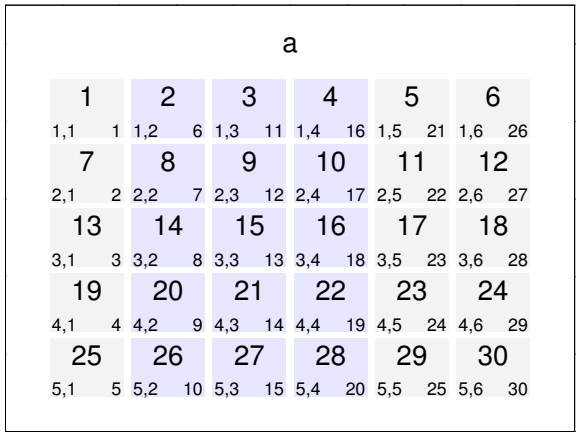
Zugriff auf Einzelelemente.

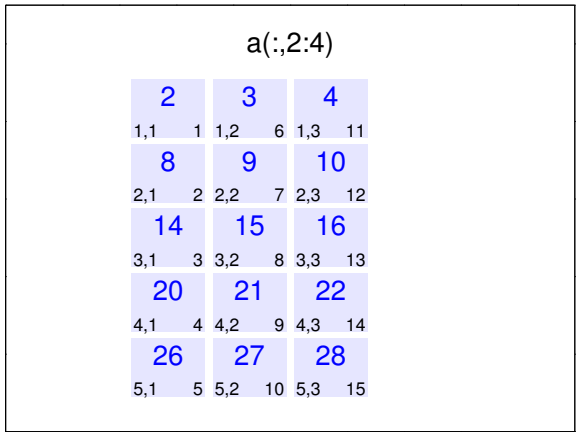


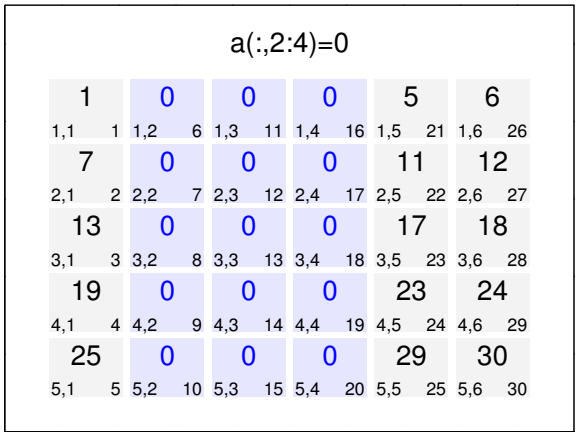




Zugriff auf alle Zeilen in mehreren Spalten.







Zugriff auf Spalten und Zeilen unter Verwendung des Keywortes `end`.

a

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

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

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

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

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

Zugriff auf Spalten und Zeilen unter Verwendung des Keywortes `end`.

a

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

a(1:2:end,1:end−1:end)

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

a(1:2:end,1:end−1:end)=[1,1;2,2;3,3]

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

Zugriff auf die gesamte Matrix.

a

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

a(:,:)

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

a(:,:)=0

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Zugriff auf Spalten und Zeilen mit Vektoren.

a

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

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

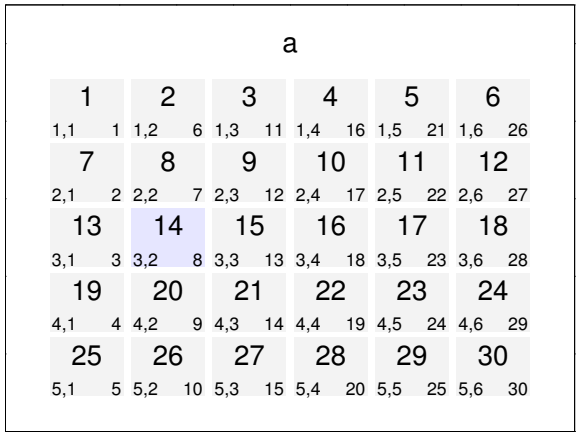
2	3	6
8	9	12
26	27	30

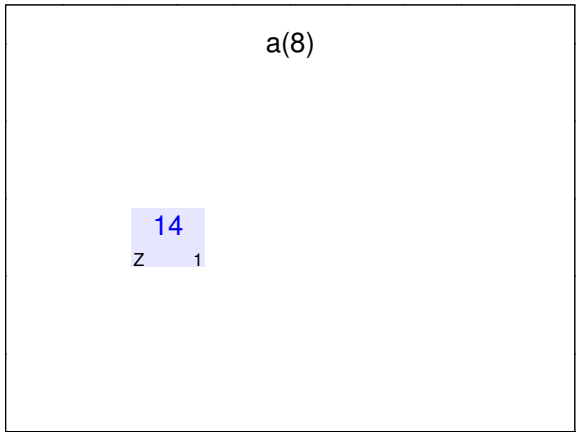
a([1,2,5],[2,3,6])=0

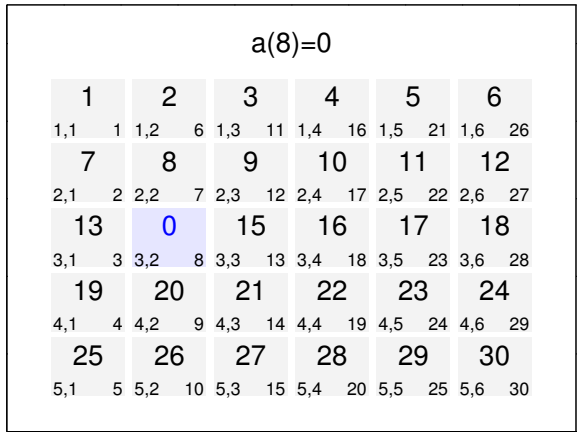
1	0	0	4	5	0
7	0	0	10	11	0
13	14	15	16	17	18
19	20	21	22	23	24
25	0	0	28	29	0

3.6.2.2 Lineare Indizierung

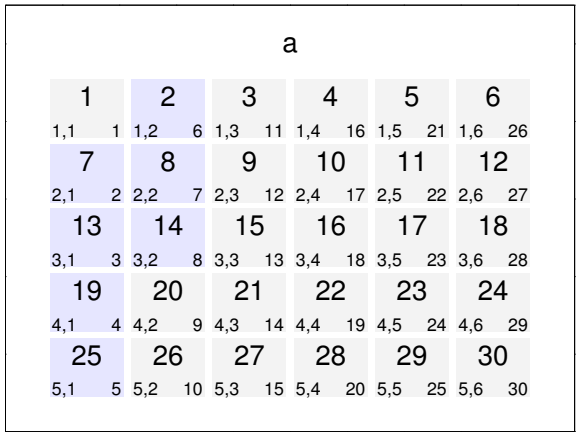
Zugriff auf Einzelelemente.

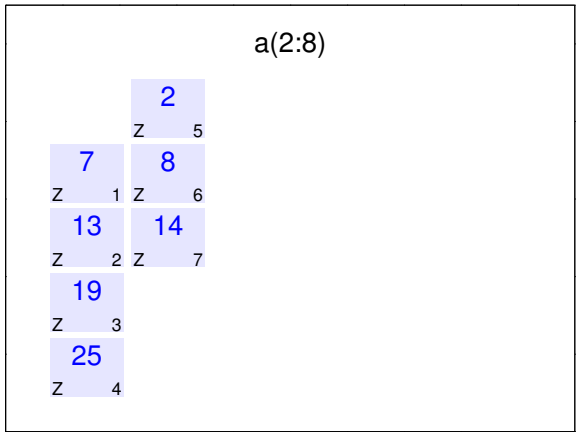


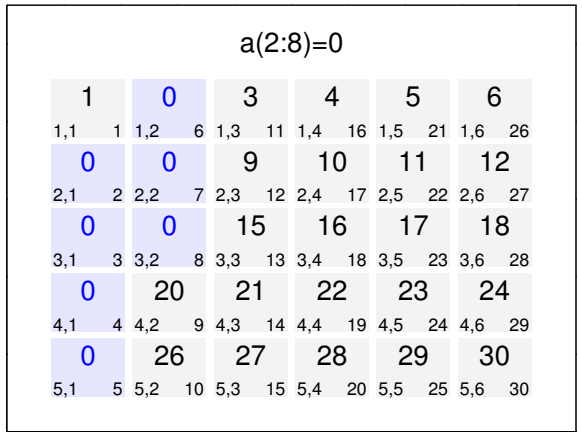




Zugriff auf zusammenhängende Bereiche.







Zugriff auf nichtzusammenhängende Bereiche.

a

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

a(4:3:end−2)

		4			
		Z 5			
	8			11	
	Z 2			Z 7	
		15			18
		Z 4			Z 9
19			22		
Z 1			Z 6		
	26			29	
	Z 3			Z 8	

a(4:3:end−2)=[−1,−2,−3,−4,−5,−6,−7,−8,−9]

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

Zugriff mit Hilfe eines beliebigen Vektors.

a

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

a([1,2,7,8,9,15,30])

1					
Z 1					
7	8				
Z 2	Z 3				
	14				
	Z 4				
	20				
	Z 5				
		27			30
		Z 6			Z 7

a([1,2,7,8,9,15,30])=NaN

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

Zugriff auf gesamte Matrix.

a

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

a(:)

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

a(:)=5

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

3.6.2.3 Logische Indizierung

Zugriff auf Teile der Matrix.

a

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

a<=9

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

a(a<=9)

1	2	3	4	5	6
S 1	S 3	S 5	S 7	S 8	S 9
7	8	9			
S 2	S 4	S 6			

Veränderung von Teilen der Matrix.

a

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

a<=9

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

a(a<=9)=0

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

Zugriff auf Teile der Matrix.

a

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

a>9 & a<25

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

a(a>9 & a<25)

			10	11	12
			S 7	S 10	S 13
13	14	15	16	17	18
S 1	S 3	S 5	S 8	S 11	S 14
19	20	21	22	23	24
S 2	S 4	S 6	S 9	S 12	S 15

Veränderung von Teilen der Matrix.

a

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

a>9 & a<25

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

a(a>9 & a<25)=0

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

Zugriff auf Teile der Matrix.

a

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

a>9 & a<25

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

a(a>9 & a<25)

			10	11	12
			S 7	S 10	S 13
13	14	15	16	17	18
S 1	S 3	S 5	S 8	S 11	S 14
19	20	21	22	23	24
S 2	S 4	S 6	S 9	S 12	S 15

Veränderung von Teilen der Matrix.

a

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

a>9 & a<25

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

a(a>9 & a<25)=0

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

Zugriff auf Teile der Matrix.

a

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

a<4 | a>28

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

a(a<4 | a>28)

1	2	3			
S 1	S 2	S 3			
				29	30
				S 4	S 5

Veränderung von Teilen der Matrix.

a

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

a<4 | a>28

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

a(a<4 | a>28)=[−1,−2,−3,−4,−5]

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

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

a

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

mod(a,3)==0

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

a(mod(a,3)==0)

3	6
S 1	S 6
9	12
S 2	S 7
15	18
S 3	S 8
21	24
S 4	S 9
27	30
S 5	S 10

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

a

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

mod(a,3)~=0

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

a(mod(a,3)~=0)=0

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

3.7 Zusammenfügen von Matrizen

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

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

3.8 Initialisieren, Löschen und Erweitern

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

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

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

3.9 Umformen von Matrizen

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

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

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

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

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

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

Kapitel 4

Operatoren

4.1 Arithmetische Operatoren

4.1.1 Arithmetische Operatoren für Skalare

Die vordefinierten Operatoren auf skalaren double-Ausdrücken sind in Tabelle 4.1 zusammengefaßt. Diese Operatoren sind eigentlich Matrixoperatoren, deren genaue Behandlung in 6 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 $\star$ und / erhalten.

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

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

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

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

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

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

4.1.2 Arithmetische Operatoren für Arrays

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

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

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

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

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

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

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

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

OPERATOR	OPERATION	BEFEHL	BEDEUTUNG	PRIORITÄT
`.^`	`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

In 4.3 kommen in einigen wenigen Fällen die Array-Operatoren mit Punkten und die Matrix-Operatoren gleichwertig vor, da sie zum selben Ergebnis führen. Eine genaue Behandlung der Matrix-Operatoren im Sinne der linearen Algebra erfolgt in 6.

Tabelle 4.4: Vergleichsoperatoren

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

4.2 Vergleichsoperatoren

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

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

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

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

ans =

     1     1     1
     1     1     0
     0     0     0
```

Tabelle 4.5: Logische Operatoren

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

4.3 Logische Operatoren

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

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

Die Priorität der logischen Operatoren ist folgendermaßen geregelt:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Beispiel mit `any` und `all`:

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

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

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

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

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

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

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

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

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

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

Kapitel 5

Mathematik

5.1 Einfache mathematische Funktionen

5.1.1 Arithmetische Operatoren

Mit den in 4 definierten arithmetischen Operatoren können alle Grundrechnungsarten durchgeführt werden. Man muss die in 4 definierte Priorität der Operatoren beachten und unter Umständen mit runden Klammern die Reihenfolge der Ausführung beeinflussen.

Fälle wo die Reihenfolge oft nicht richtig im numerischen Programm spezifiziert wird, sind in folgender Tabelle angeführt:

MATH	MATLAB
$y = \frac{x}{a+b}$	`y = x ./ (a + b)`
$y = \frac{x}{a} + b$	`y = x ./ a + b`
$y = x^{a+b}$	`y = x .^ (a + b)`
$y = x^{a} + b$	`y = x .^ a + b`

Man beachte auch die auf den ersten Blick ungewöhnliche Form der Operatoren `.*` (Multiplikation), `./` (Division) und `.^` (Potenz). Der Punkt vor dem Operator teilt MATLAB mit, dass es die Operation elementweise durchführen muss. Damit funktionieren all diese Operationen mit Arrays gleicher Größe oder mit einer Kombination von Arrays und Skalaren. Behandelt man also nicht Probleme der linearen Algebra (z.B. Matrizenmultiplikation) sollte man immer die Punkt-Operatoren für Multiplikation, Division und Potenzierung verwenden.

5.1.2 Mathematische Funktionen und Konstanten

Die Argumente für mathematische Funktionen, siehe z.B. 5.1.5, müssen anders als in der Mathematik immer innerhalb runder Klammern geschrieben werden:

MATH	MATLAB
$y = \sin x$	`y = sin(x)`
$y = \sin \pi x$	`y = sin(pi .* x)`
$y = \sin(a + b)$	`y = sin(a + b)`
$y = e^{-ix}$	`y = exp(-i.*x)`
$y = \sqrt{1 + cosx}$	`y = sqrt(1 + cos(x))`

An den Beispielen sieht man, dass MATLAB die Zahl π (`pi`) aber nicht die Eulersche Zahl e (`exp(1)`, siehe dazu 5.1.3) kennt. MATLAB kennt auch die imaginäre Konstante $i = \sqrt{-1}$ und praktisch alle (sinnvollen) Operationen können sowohl mit rellen als auch mit komplexen Zahlen durchgeführt werden (siehe dazu 5.1.4).

In der letzten Zeile der Tabelle sieht man einen zusammengesetzten Ausdruck. Die Ausführung in MATLAB muss man sich von Innen nach Aussen vorstellen, d.h., das Ergebnis von `cos(x)` wird mit `1` addiert und danach wird daraus die `Wurzel` gezogen. Die Befehle können beliebig verschachtelt sein. Wichtig ist die korrekte Setzung der Klammern und natürlich die Vorschrift, dass die Ergebnisse innerer Operationen formal richtigen Input für die äusseren Operationen liefern müssen.

5.1.3 Exponential Funktion, Logarithmus

Die Exponentialfunktion $\exp(x)$, die in mathematische Schreibweise oft als e^x geschrieben wird, wird in MATLAB durch die Funktion `exp` berechnet. Die Eulersche Zahl e ist in MATLAB nicht definiert. Braucht man sie, dann muss man sie durch `e=exp(1)` selbst definieren. Für weitere Rechnungen braucht man sie aber nicht wirklich, da die Berechnung von e^x in MATLAB besser mit `exp(x)` und nicht mit mit einer `Potenz` von e (`exp(1).^x`) erfolgt.

Die Funktion `expm1` berechnet $\exp(x) - 1$. Dies ist deshalb wichtig, da für kleine Argumente x der Befehl `exp(x)-1` Null liefert, während `expm1(x)` noch korrekte Ergebnisse liefert, da gilt $\{\forall x \to 0 \mid \exp(x) - 1 \propto x\}$.

Die Umkehrfunktion zur Exponentialfunktion $y = e^x$ ist die Logarithmusfunktion $x = \ln y$. Vertauscht man nun x und y so erhält man $y = \ln x$. Dies bezeichnet man als den natürlichen Logarithmus von x. In MATLAB wird der natürliche Logarithmus durch die Funktion `log(x)` berechnet.

Aus ähnlichen Gründen wie bei `expm1` berechnet die Funktion `logp1` die Funktion $\ln(1 + x)$. Für kleine Argumente x liefert der Befehl `log(1+x)` Null, während `log1p(x)` noch korrekte Werte liefert, da gilt $\{\forall x \to 0 \mid \ln(x + 1) \propto x\}$.

Der Befehl `log` funktioniert für negative Zahlen (komplexes Ergebnis) und für komplexe Zahlen. Interessiert man sich nur für reelle Ergebnisse für positive Zahlen, kann man den Befehl `reallog` verwenden. Dieser liefert für negativen oder komplexen Input eine Fehlermitteilung.

Der natürliche Logarithmus $\ln$ wird auch als Logarithmus zur Basis e bezeichnet

$$\ln x = \log_e x \ .$$

Der Logarithmus zur Basis 10, $\log_{10} x = \log x$, wird in MATLAB mit `log10(x)` berechnet und der Logarithmus zur Basis 2, $\log_2(x)$ wird mit `log2(x)` berechnet. Für Logarithmen zu einer beliebigen Basis a, $\log_a$, muss man sich der Formel

$$\log_a x = \frac{\ln x}{\ln a} ,$$

also in MATLAB z.B. für $\log_3 10$:

```
log_a = @(x,a) log(x) ./ log(a);
y = log_a(10,3);
```

Die allgemeine Exponentialfunktion (Potenz)

$$a^x = e^{x \ln a}$$

wird im Normalfall natürlich mit dem Operator `.^` für das `Potenzieren` berechnet. Es stehen aber einige spezielle Befehle zur Verfügung. Der Operator funktioniert natürlich mit komplexer Basis und/oder Hochzahl. Interessiert man sich nur für reelle Ergebnisse für reellen Input kann man den Befehl `realpow(x,y)` für die Berechnung von x^y verwenden.

Die Quadratwurzel, $\sqrt{x} = x^{1/2}$, wird mit `sqrt(x)` berechnet. Interessiert man sich nur für reelle Wurzeln von nicht-negativen Zahlen, kann man den Befehl `realsqrt(x)` verwenden. Zur Berechnung der reellen n-ten Wurzel soll man den Befehl `nthroot(x,n)` verwenden. Am Beispiel von $\sqrt[3]{-3} = (-3)^{1/3}$ sei der Unterschied zwischen `nthroot(-3,3)` und der Potenzfunktion erläutert:

```
nthroot(-3,3) % liefert -1.4422
(-3).^(1/3)   % liefert  0.7211 + 1.2490i
```

Beide Ergebnisse sin korrekt, aber nur mit `nthroot(-3,3)` erhält man die reelle Wurzel. Klarerweise müssen bei negativen x die Potenzen n ungerade ganze Zahlen sein. Sonst bricht die Funktion mit einer Fehlermitteilung ab.

Für die Basis 2 kann man zur Berechnung von 2^x den Befehl `pow2(x)`, bzw. für $a \cdot 2^x$ den Befehl `pow2(a,x)`. Eine solche Spezial-Funktion ist typischerweise schneller als der normale Befehl.

Der Befehl `p=nextpow2(A)` liefert jene Hochzahl p die sicherstellt, dass gilt $2^p \geq A$. Z.B. ist für alle Werte von A zwischen 513 und 1024 $p = 10$.

5.1.4 Komplexe Zahlen

Als imaginäre Einnheit $i = \sqrt{-1}$ ist in MATLAB `i` aber auch `j` vorgesehen. Man sollte diese beiden Zahlen daher nicht als Variable verwenden:

```
z = 3 + 5*i; % komplexe Zahl

i = 5;
z = 3 + 5*i; % liefert 28

clear('i');
z = 3 + 5*i; % komplexe Zahl
```

Komplexe Zahlen können also durch Befehle vom Typ `z = x + i*y` oder mit `z = complex(x,y)` erzeugt werden.

Folgende Befehle sind für komplexe Zahlen $z = x + iy$ hilfreich:

BEZEICHNUNG	MATH	MATLAB
Realteil	$\mathrm{Re}\ z = x$	`real(z)`
Imaginärteil	$\mathrm{Im}\ z = y$	`imag(z)`
Absolutbetrag	$\lvert z\rvert = \sqrt{x^2 + y^2}$	`abs(z)`
Phasenwinkel	$\phi = \arctan(y, x)$	`angle(z)`
Signum	$\mathrm{sign}\ z = z/\lvert z\rvert$	`sign(z)`
Konjugiert komplex	$\overline{z} = x - iy$	`conj(z)`
Konjugiert komplex transponiert	$\overline{z}^T$	`ctranspose(z)`

Der Befehl `sign(x)` liefert für reelle Werte von x den Wert 1 für $x > 0$, -1 für $x < 0$ und 0 für $x = 0$, für komplexe Werte von x liegen die Ergebnisse am Einheitskreis.

Der Befehl `unwrap(p)` korrigiert Phasensprünge in einem Vektor von Phasenwinkeln p die größer als $\pm\pi$ sind. Dabei werden zu p ganzzahlige Vielfache von 2π addiert oder subtrahiert, damit alle Phasensprünge kleiner als π werden.

Die Überprüfung, ob ein Array (eine Zahl) real oder komplex ist, kann mit `isreal(z)` erfolgen. Dies liefert `1`, wenn `z` reell ist, also keinen Imaginärteil besitzt, ansonsten ist das Resultat `0`. Dieses Ergebnis bekommt man auch, wenn der Imaginärteil vorhanden, aber exakt Null ist. Ob eine Zahl keinen Imaginärteil oder einen Imaginärteil, der exakt Null ist, hat, kann man mit

```
~any(imag(z(:)))
```

feststellen. Im Gegensatz zu `isreal(z)` ist hier das Ergebnis `1`, wenn der Imaginärteil vorhanden aber Null ist.

5.1.5 Trigonometrische Funktionen

In 5.1 sind die trigonometrischen Funktionen zusammengefasst. Diese Funktionen sind in MATLAB für verschiedene Argumente implementiert, nämlich `sin(x)` für das Argument `x` in Radiant, bzw., `sind(x)` für das Argument `x` in Grad (Degree). Ausserdem gibt es jeweils die Hyperbelfunktion, also z.B. den Sinus-Hyperbolicus `sinh`.

Die Funktion `Sekans` ist definiert als $1/\cos(x)$ und die Funktion `Kosekans` ist definiert durch $1/\sin(x)$. Sinngemäß gilt das Gleiche für die entsprechenden Hyperbelfunktionen `Sekans-Hyperbolicus` und `Kosekans-Hyperbolicus`.

Die Umkehrfunktionen (Arkus-Funktionen) liefern den Wert in Radiant (`asin`), bzw., in Grad (`asind`). Die Umkehrfunktionen für die Hyperbelfunktionen heissen korrekterweise nicht Arkus-Sinus-Hyperbolicus sondern Area-Sinus-Hyperbolicus. Dieser Umstand wurde in der Tabelle 5.1 aus praktischen Gründen nicht berücksichtigt.

Alle trigonometrischen Funktionen arbeiten natürlich sowohl mit reellen als auch mit komplexen Argumenten.

In Ergänzung zu `atan` gibt es die sehr praktische Funktion `atan2`. Die Funktion `atan(x)` hat ein Argument und liefert das Ergebnis im Intervall $[-\pi/2, \pi/2]$. Die Funktion `atan2(y,x)` hat zwei Argumente und liefert das Ergebnis im Intervall $[-\pi, \pi]$, womit man auch die richtige Zuordnung zum Quadranten erhält. Der Zusammenhang zwischen Zylinderkoordinaten und kartesischen Koordinaten ist $r = \sqrt{x^2 + y^2}$ und $\phi = \arctan(y/x)$. Mit der normalen Funktion `atan` kann man nicht unterscheiden zwichen den Punkten $(x, y) = (1, 1)$ und $(-1, -1)$, da y/x hier immer gleich 1 ist. Diese Ununterscheidbarkeit gilt für alle um den Ursprung gespiegelten Punkte. Das Problem wird bei der getrennten Übergabe von x und y an `atan2(y,x)` gelöst, da mit dieser Information die Zuordung zum richtigen Quadranten und damit die Berechnung des richtigen Winkels leicht ist.

Die Funktion `c=hypot(a,b)` berechnet die Wurzel aus der Summme der Quadrate

$$c = \sqrt{a^2 + b^2} \, ,$$

wobei sie so geschrieben ist, dass sie in einem viel weiteren Bereich korrekte Ergebnisse liefert als die einfache Umsetzung,

```
hyp = @(a,b) sqrt(a.^2 + b.^2)
```

wobei das Problem durch Überschreiten der größten möglichen Zahl `realmax`, z.B., bei der Operation a^2 besteht, obwohl das Ergebnis noch darstellbar wäre. Die Funktion `hypot` umgeht dieses Problem für große Zahlen.

Tabelle 5.1: Trigonometrische Funktionen

BEZEICHNUNG	RADIANT	GRAD	HYPERBOLISCH
Sinus	`sin`	`sind`	`sinh`
Kosinus	`cos`	`cosd`	`cosh`
Tangens	`tan`	`tand`	`tanh`
Kotangens	`cot`	`cotd`	`coth`
Sekans	`sec`	`secd`	`sech`
Kosekans	`csc`	`cscd`	`csch`
UMKEHRFUNKTIONEN			
Arkus-Sinus	`asin`	`asind`	`asinh`
Arkus-Kosinus	`acos`	`acosd`	`acosh`
Arkus-Tangens	`atan`	`atand`	`atanh`
Arkus-Kotangens	`acot`	`acotd`	`acoth`
Arkus-Sekans	`asec`	`asecd`	`asech`
Arkus-Kosekans	`acsc`	`acscd`	`acsch`

Für komplexe Zahlen a und b berechnet `hypot` das reelle Ergebnis

$$c = \sqrt{|a|^2 + |b|^2} \ ,$$

wobei $|x|$ der Absolutbetrag von x ist (`abs(x)`).

5.1.6 Diskrete Mathematik

5.1.6.1 Primzahlen

Eine Primzahl ist eine natürliche Zahl mit genau zwei natürlichen Teilern, nämlich 1 und sich selbst. Primzahlen sind also 2, 3, 5, 7, 11, usw.. Die fundamentale Bedeutung der Primzahlen für viele Bereiche der Mathematik beruht auf den folgenden drei Konsequenzen aus dieser Definition:

- Primzahlen lassen sich nicht als Produkt zweier natürlicher Zahlen, die beide größer als eins sind, darstellen.
- Lemma von Euklid: Ist ein Produkt zweier natürlicher Zahlen durch eine Primzahl teilbar, so ist bereits einer der Faktoren durch sie teilbar.
- Eindeutigkeit der Primfaktorzerlegung: Jede natürliche Zahl lässt sich als Produkt von Primzahlen schreiben. Diese Produktdarstellung ist bis auf die Reihenfolge der Faktoren eindeutig.

Jede dieser Eigenschaften könnte auch zur Definition der Primzahlen verwendet werden.

Eine natürliche Zahl größer als 1 heißt prim, wenn sie eine Primzahl ist, andernfalls heißt sie zusammengesetzt. Die Zahlen 0 und 1 sind weder prim noch zusammengesetzt.

In MATLABerzeugt der Befehl `p = primes(n)` einen Zeilenvektor p mit allen Primzahlen, die kleiner gleich n sind. Der Befehl `f = factor(n)` liefert die Faktorisierung (Zerlegung in Primzahlen) der natürlichen Zahl n im Zeilenvektor f. Mit dem Befehl `L = isprime(M)` kann man feststellen, ob die Elemente eines Arrays M Primzahlen sind. Das Ergebnis ist ein logisches Feld L der gleichen Größe wie M mit Eines (prim) oder Nullen(nicht prim).

5.1.6.2 Gemeinsame Teiler und Vielfache

Der Befehl `G = gcd(A,B)` erzeugt den "Größten gemeinsamen Teiler" G der entsprechenden Elemente der ganzzahligen Arrays A und B. Nach mathematischer Konvention liefert `gcd(0,0)` den Wert 0 und alle anderen Inputgrößen liefern positive ganze Zahlen für G.

Beim Aufruf `[G,C,D] = gcd(A,B)` liefert der Befehl neben G auch die beiden Arrays C und D, die folgende Gleichung erfüllen

$$A_i C_i + B_i D_i = G_i \ ,$$

wobei der Index i dafür steht, dass dies für alle entsprechenden Elemente in den Arrays gilt.

Der Befehl `L = lcm(A,B)` erzeugt das "Kleinste gemeinsame Vielfache" L der entsprechenden Elemente der ganzzahligen Arrays A und B.

5.1.6.3 Fakultät

Der MATLAB-Befehl `fac=factorial(n)` berechnet n-Fakultät für ganze positive Zahlen inklusive Null, $n \in N_0^+$,

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \ ,$$

wobei in Ergänzung auch gilt $0! = 1$. Für $n \in N^+$ würde natürlich auch

```
fac = prod([1:n])
```

funktionieren, 0-Fakultät wäre hier aber falsch (`prod`). Mit dem Befehl

```
fac1 = cumprod([1:n])
fac0 = [1,cumprod([1:n])]
```

erhält man alle Werte von $1!$ (bzw. $0!$) bis n-Fakultät (`cumprod`). Ist der Input von `factorial` ein Feld `N`, bekommt man ein gleich großes Feld mit der Fakultät jedes Elements von `N`.

Auf Grund der Tatsache, dass der doppeltgenaue Datentyp (`double`) ungefähr 15 Stellen zur Verfügung hat, sind nur Werte bis $21!$ exakt. Darüber hinaus stimmt die Größenordnung und die ersten 15 Stellen.

5.1.6.4 Binomialkoeffizienten

Der MATLAB-Befehl `nchoosek(n,k)` berechnet den Binomialkoeffizienten

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

wobei n und k nichtnegative ganze Zahlen sein müssen. In der Kombinatorik bezeichnet der Binomialkoeffizient die Anzahl von möglichen Kombinationen von n Dingen, wobei immer k Dinge ausgewählt werden. Für einen Zeilenvektor v berechnet `nchoosek(n,v)` eine Matrix, wobei in den Zeilen die möglichen Kombinationen der Elemente von v stehen. Die Befehle

```
v = [1:4]
C = nchoosek(v,3)
```

liefern also

$$v = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \quad , \quad C = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 1 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix} \quad .$$

Der Befehl `nchoosek(4,3)` würde natürlich die Anzahl der möglichen Kombinationen, also die Anzahl der Zeilen in `C`, liefern (`size(C,1)`).

5.1.6.5 Permutationen

Der MATLAB-Befehl `P = perms(v)` berechnet alle möglichen Permutationen der Elemente des Vektors `v`. Die Befehle

```
v = [1:4]
P = perms(v)
```

liefern damit

$$v = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \quad , \quad C = \begin{bmatrix} 3 & 2 & 1 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix} \quad ,$$

wobei der Befehl nur praktikabel ist, wenn die Anzahl der Elemente in `v` kleiner als 15 ist. Will man nur die Anzahl der Permutationen wissen, kann man die natürlich einfacher mit `factorial(numel(v))` berechnen. Der Befehl `numel` liefert dabei die Anzahl der Elemente im Vektor `v`.

5.1.6.6 Näherung durch rationale Zahlen

Obwohl alle Fließkommazahlen durch die beschränkte Anzahl von Stellen rationale Zahlen sind, ist es trotzdem manchmal sinnvoll sie durch einfache rationale Zahlen anzunähern, deren Zähler und Nenner jeweils kleine ganze Zahlen sind. Dazu dient der Befehl `rat` So liefert `[n,d]=rat(pi)` die Werte $n = 355$ (nominator, Zähler) und $d = 113$ (denominator, Nenner), also die rationale Zahl $n/d = 355/113$. Diese Berechnung erfolgt mit dem Defaultwert für die relative Genauigkeit von $1 \cdot 10^{-6}$. Will man eine andere Genauigkeit, muss man diese spezifizieren, also z.B. `[n,d]=rat(pi,1.e-8)` und erhält damit $n/d = 104348/33215$. Die weithin bekannte Repräsentation von $\pi \approx 22/7$ erhält man durch `[n,d]=rat(pi,1.e-2)`.

Verwendet man den Befehl `rat` ohne Ausgabeparameter, dann wird der Wert in Form eines Kettenbruchs (”continued fraction”) ausgegeben

$$\pi \approx 3 + \cfrac{1}{7 + \cfrac{1}{16 - \cfrac{1}{294}}} \approx 3 + \cfrac{1}{7 + \cfrac{1}{16}} \approx 3 + \frac{1}{7} \, .$$

Natürlich funktioniert der Befehl auch mit Feldern `X`, `[N,D]=rat(X)` und man erhält als Ergebnis gleich große Felder `N` und `D` mit Zähler bzw. Nenner für die Näherung der jeweiligen Elemente in `X`.

5.2 Platzhalter

Dies ist ein Platzhalter für ein geplantes Kapitel.

In der Zwischenzeit beschränkt sich der Inhalt auf einen Link auf ein MATLAB-Dokument über mathematische Fragen .

Kapitel 6

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & \dots & a_{mn} \end{bmatrix} = [a_{jk}] \,. \tag{6.1}$$

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,

$$\mathbf{a} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = [a_j] \,. \tag{6.2}$$

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,

$$\mathbf{b} = \begin{bmatrix} b_{11} \; b_{12} \; b_{13} \end{bmatrix} = \begin{bmatrix} b_1 \; b_2 \; b_3 \end{bmatrix} = [b_j] \; . \tag{6.3}$$

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,

$$s = \mathbf{s} = s_{11} = s_1 = [s_j] \; . \tag{6.4}$$

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

6.1 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,

$$\mathbf{A}^T = [a_{kj}] \; . \tag{6.5}$$

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

Symmetrische Matrix: Für eine symmetrische Matrix gilt

$$\mathbf{A}^T = \mathbf{A} \; . \tag{6.6}$$

Schiefsymmetrische Matrix: Für eine schiefsymmetrische Matrix gilt

$$\mathbf{A}^T = -\mathbf{A} \ . \tag{6.7}$$

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

$$\mathbf{A} = \mathbf{S} + \mathbf{U} \ , \tag{6.8}$$

$$\mathbf{S} = \frac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right) \ , \tag{6.9}$$

$$\mathbf{U} = \frac{1}{2}\left(\mathbf{A} - \mathbf{A}^T\right) \ . \tag{6.10}$$

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

6.2 Addition und Subtraktion von Matrizen

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

$$\mathbf{C} = \mathbf{A} \pm \mathbf{B} \implies [c_{jk}] = [a_{jk}] \pm [b_{jk}] \ , \tag{6.11}$$

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,

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \begin{bmatrix} 5\ 7\ 9 \end{bmatrix} \ , \tag{6.12}$$

$$\mathbf{D} = \mathbf{A} + 1 = \begin{bmatrix} 2\ 3\ 4 \end{bmatrix} \ . \tag{6.13}$$

Fehler: Die Rechnung

$$\mathbf{C} = \mathbf{A} + \mathbf{B}^T = \begin{bmatrix} 1\ 2\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \text{Error} \tag{6.14}$$

ergibt natürlich eine Fehlermitteilung.

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

6.4 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,

$$c_{jk} = \sum_{l=1}^{n} a_{jl} b_{lk} \ . \tag{6.15}$$

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 6.15).

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

$$\mathbf{AB} \neq \mathbf{BA} \ . \tag{6.16}$$

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:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 14 & 32 \\ 32 & 77 \end{bmatrix} \ . \tag{6.17}$$

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

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 17 & 22 & 27 \\ 22 & 29 & 36 \\ 27 & 36 & 45 \end{bmatrix} . \tag{6.18}$$

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

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \text{Error} . \tag{6.19}$$

Inneres Produkt zweier Vektoren:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 32 . \tag{6.20}$$

Äußeres Produkt zweier Vektoren:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix} . \tag{6.21}$$

Fehler:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \text{Error} . \tag{6.22}$$

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

$$\mathbf{C}^T = \mathbf{B}^T\mathbf{A}^T \tag{6.23}$$

$$\mathbf{C} = (\mathbf{B}^T\mathbf{A}^T)^T \tag{6.24}$$

$$(c\mathbf{A})^T = c\mathbf{A}^T \tag{6.25}$$

6.5 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

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_{l=1}^{n} a_l b_l \ . \tag{6.26}$$

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.

6.6 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**

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} . \tag{6.27}$$

Für die Einheitsmatrix gilt

$$\mathbf{AI} = \mathbf{IA} = \mathbf{A} . \tag{6.28}$$

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

Die **inverse Matrix** ist definiert durch

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} . \tag{6.29}$$

Eine Matrix für die

$$\mathbf{A}^T = \mathbf{A}^{-1} \tag{6.30}$$

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

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

6.7 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

$$\mathbf{A}\mathbf{x} = \mathbf{b} \; . \tag{6.31}$$

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \text{, und } \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} \text{ und } \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \tag{6.32}$$

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:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array} \tag{6.33}$$

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 6.33 ist ein Vektor $\mathbf{x}$ der Länge n, der alle m Gleichungen erfüllt. Ist das System 6.33 homogen, hat es zumindest die **triviale** Lösung

$$x_1 = 0 \ , \ x_2 = 0 \ , \ \dots \ , \ x_n = 0 \ . \tag{6.34}$$

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 6.31 von Links mit $\mathbf{A}^{-1}$ multiplizieren

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \implies \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{A}\backslash\mathbf{b}. \tag{6.35}$$

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

$$\mathbf{y}\mathbf{A} = \mathbf{c} \ , \tag{6.36}$$

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

$$\mathbf{y}\mathbf{A}\mathbf{A}^{-1} = \mathbf{c}\mathbf{A}^{-1} \implies \mathbf{y} = \mathbf{c}\mathbf{A}^{-1} = \mathbf{c}/\mathbf{A} \ . \tag{6.37}$$

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 6.37 umgeformt werden in

$$\mathbf{y} = \left((\mathbf{A}^{-1})^T\mathbf{c}^T\right)^T = \left(\mathbf{A}^T\backslash\mathbf{c}^T\right)^T \ , \tag{6.38}$$

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

Für die Skalare s und r würde gelten

$$\begin{array}{rcccl} s/r & = & sr^{-1} & = & s/r \\ r\backslash s & = & r^{-1}s & = & s/r, \end{array} \tag{6.39}$$

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 6.33 umschreiben als:

$$\begin{array}{ccccccccc}
a_{11}x_{11} & + & a_{12}x_{21} & + & \dots & + & a_{1n}x_{n1} & = & b_{11} \\
a_{21}x_{11} & + & a_{22}x_{21} & + & \dots & + & a_{2n}x_{n1} & = & b_{21} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{11} & + & a_{m2}x_{21} & + & \dots & + & a_{mn}x_{n1} & = & b_{m1} \\
\hline
a_{11}x_{12} & + & a_{12}x_{22} & + & \dots & + & a_{1n}x_{n2} & = & b_{12} \\
a_{21}x_{12} & + & a_{22}x_{22} & + & \dots & + & a_{2n}x_{n2} & = & b_{22} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{12} & + & a_{m2}x_{22} & + & \dots & + & a_{mn}x_{n2} & = & b_{m2} \\
\hline
\vdots & + & \vdots & + & \vdots & + & \vdots & = & \vdots \\
\hline
a_{11}x_{1p} & + & a_{12}x_{2p} & + & \dots & + & a_{1n}x_{np} & = & b_{1p} \\
a_{21}x_{1p} & + & a_{22}x_{2p} & + & \dots & + & a_{2n}x_{np} & = & b_{2p} \\
\vdots & + & \vdots & + & \ddots & + & \vdots & = & \vdots \\
a_{m1}x_{1p} & + & a_{m2}x_{2p} & + & \dots & + & a_{mn}x_{np} & = & b_{mp}
\end{array} \tag{6.40}$$

Dieses Gleichungssystem kann formal geschrieben werden als

$$\mathbf{AX} = \mathbf{B} \ , \tag{6.41}$$

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 7

Steuerkonstrukte

7.1 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.

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

     1
```

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 = 12;
r = mod(a,b)

     0
```

7.2 Auswahl

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

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

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

7.2.1 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.

7.2.2 Auswahlanweisung

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

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

Bei der Verwendung ist Folgendes zu beachten:

- Während beim `IF-Block` mehrere Bedingungen ausgewertet werden können, richtet sich die Abarbeitung einer `Auswahlanweisung` nach dem Wert eines einzigen Ausdrucks, dem sogenannten Schalter. Dieser Schalter kann ein Skalar oder eine Zeichenkette sein. Er muss **keine logische Variable** sein.

- Die Abarbeitung erfolgt mit Hilfe der `case`-Anweisung. Die Ausführung wird durch einen Vergleich zwischen Schalter und Selektor geregelt. Stimmen die beiden überein, wird der zugehörige Auswahlblock ausgeführt und danach an das Ende der `Auswahlanweisung` gesprungen. Es wird also maximal ein Auswahlblock ausgeführt.

- Die Auswahl erfolgt naturgemäß wieder mit Skalaren oder Zeichenketten. Sollen für einen `case` mehrere Möglichkeiten erlaubt sein, kann man für den Selektor eine durch Beistriche getrennte Liste angeben. Solche Listen werden mit Hilfe geschwungener Klammern geschrieben, z.B. `{1,2,3}` oder `{'a','b','c'}`.

- Während sich die Bedingungen eines `IF-Blocks` überschneiden können, müssen die Alternativen einer `Auswahlanweisung` eindeutig sein.

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

MATLAB Beispiel

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

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

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

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

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

MATLAB Beispiel

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

Als Schalter dient dafür einfach der String `str`.

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

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

MATLAB Beispiel

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

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

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

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

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

7.3 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:

$$a_1 = 0\ ,\ a_2 = 1\ ,\ a_k = \frac{a_{k-2} + a_{k-1}}{2} \qquad \forall k \geq 3\ . \tag{7.1}$$

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.

7.3.1 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.

7.3.2 Die bedingte Schleife

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

```
while Bedingung
   Anweisungsblock
end
```

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

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

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

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

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

MATLAB Beispiel

Hier wird die Rekursionsformel

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

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

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

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

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

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

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

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

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

Kapitel 8

Datentypen - Klassen

Dieses Kapitel ist erst im Entstehen.

Information findet man derzeit unter der MATLAB-Hilfe für Datentypen .

In MATLAB gibt es wie praktisch in jeder Programmiersprache verschiedene Datentypen bzw. Klassen von Daten. Es gibt mehrere numerische Datentypen 8.1, einen logischen Datentyp 8.2, einen Datentyp für Zeichen bzw. Zeichenketten 8.3, zwei Klassen für Behälter 8.4 die wiederum alle anderen Typen beinhalten können, und zwei Klassen für Funktionen 8.5.

8.1 Numerische Datentypen

Die numerischen Datentypen unterscheiden sich durch den Speicherplatz, den sie verbauchen, durch die größte bzw. die kleinste darstellbare Zahl und durch die Möglichkeit negative Zahlen darzustellen oder nicht.

KLASSE	B	KLEINSTE ZAHL	GRÖSSTE ZAHL
double	8	±2.2251e-308	±1.7977e+308
single	4	±1.1755e-38	±3.4028e+38
int64	8	-9223372036854775808	9223372036854775807
uint64	8	0	9223372036854775807
int32	4	-2147483648	2147483647
uint32	4	0	2147483647
int16	2	-32768	32767
uint16	2	0	32767
int8	1	-128	127
uint8	1	0	127

Zu dieser Tabelle sind einige Bemerkungen notwendig.

Speicherplatz: Der Eintrag B steht für den verbrauchten Speicherplatz für eine Zahl in BYTES, wobei ein BYTE 8 BITS entspricht. Ein BIT ist die kleinste Speicherplatzeinheit, die die beiden Werte `0` oder `1` speichern kann. Um den Speicherplatz eines Feldes zu berechnen, muss man diese Zahl mit der `Anzahl`numel der Elemente im Feld multiplizieren.

Handelt es sich ausserdem um eine `Komplexe`-Zahl mit `Real`- und `Imagiär`-Teil muss man den Wert nochmals mit zwei multiplizieren. Der Befehl `whos` gibt Auskunft über den Speicherbedarf der Variablen.

Darstellung: Die Darstellung `2.2251e-308` bei sogenannten Fließkommazahlen steht für den Wert 2.225110^{-308}, dass heisst die Zahl hinter dem `e` stellt den sogenannten Exponenten dar.

Anzahl der Stellen:

Unsignierte Ganze Zahlen:

Kleinste darstellbare Zahl:

Größte darstellbare Zahl:

Genauigkeit:

Besondere Zahlen:

8.1.1 Fließkomma Datentypen

8.1.1.1 Der Datentyp `double`

Der hauptsächlich in MATLAB verwendete Datentyp ist der Typ `double`. Verwendet man keine Angabe eines Datentyps wird automatisch `double` gewählt.

INPUT	OUTPUT	ERLÄUTERUNG
`x = [1:5]`	`[1 2 3 4 5]`	
`class(x)`	`'double'`	Klasse
`isa(x,'double')`	`1`	Klassenüberprüfung
`isnumeric(x)`	`1`	numerisch - wahr
`isfloat(x)`	`1`	Fließkomma - wahr
`isinteger(x)`	`0`	ganzzahlig - falsch

Obwohl es also den Anschein hat, als ob die Variable `x` ganzzahlig ist, gehört sie zum Datentyp `double`. Will man sicherstellen, dass die Zahlen nur ganzzahlig, dass heisst mit geringerem Speicherbedarf verwendet bzw. gespeichert werden, muss man einen ganzzahligen Datentyp 8.1.2 verwenden. Man sollte aber bedenken, dass praktisch alle mathematischen Operationen den Datentyp `double` benötigen.

Darüber hinaus liefern auch Befehle zum automatischen Erzeugen von Feldern (`zeros`, `ones`, ...) ohne Angabe einer Klasse den Datentyp `double`.

8.1.1.2 Der Datentyp `single`

8.1.2 Ganzzahlige Datentypen

8.2 Der logische Datentyp `logical`

8.3 Der Zeichen-Datentyp `char`

8.4 Klassen für Behälter

8.4.1 Der Zellen-Klasse `cell`

8.4.2 Der Struktur-Klasse `struct`

8.5 Klassen für Funktionen

8.5.1 Die Klasse `inline`

8.5.2 Die Klasse `functionhandle`

Kapitel 9

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.

9.1 FUNCTION-Unterprogramme

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.1.4 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 9.1.5 beschrieben werden.

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

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

9.1.5 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.

9.1.6 Optionale Parameter und Rückgabewerte

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

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

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

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

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

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

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

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

Eine mögliche Realisierung einer solchen Überprüfung kann folgendermaßen aussehen:

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

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

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

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

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

9.2 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 ==> ....
```

9.3 Anonyme Funktionen - Function Handle

Ein `function_handle` stellt eine Referenz auf eine Funktion dar. Referenz bedeutet vereinfacht dargestellt, dass eine (MATLAB- oder eine selbst geschriebene) Funktion über einen alternativen Namen angesprochen werden kann. Einen großer Vorteil ist, dass ein `function_handle` an eine Funktion als Parameter übergeben werden kann. Dies soll hier aber nicht weiter erläutert werden (siehe MATLAB-Dokumentation unter `function_handle`)

Es ist auch möglich einen `function_handle` derart zu definieren, dass die "Funktion" nur über diesen ansprechbar ist. Dies nennt man dann einen Handle auf eine anonyme Funktion. Dies ist sehr praktisch, da man damit einfache Funktionen (ähnlich zu `inline`-Funktionen) direkt in einem Skript definieren kann. Ein Vorteil von `function_handle` gegenüber inline-Funktionen ist, dass sie einfacher miteinander kombiniert und modifiziert werden können.

```
% Definition eines Handles auf eine anonyme Funktion
```

```
mod_fun = @(A,x) A*(x.^2+x.^3) ;
% Definition eines weiteren Handles, der sich vom ersten
% durch die Reihenfolge der Parameter unterscheidet.
fun_han = @(x,A) mod_fun(A,x) ;
% Berechnung
A = 5 ;
x = linspace(-1,1,50) ;
y1 = mod_fun(A,x) ;
y2 = fun_han(x,A) ;
```

Man sieht hier, dass es möglich ist den `function_handle` auch für weitere Definitionen zu verwenden. Hier wurde z.B. bei der Definition von `fun_han` die Reihenfolge der Eingabeparameter vertauscht (Die Reihenfolge hinter dem `@` ist maßgeblich).

Im folgenden Beispiel geht es um den Umgang mir vorher bekannten Variablen, die nicht übergeben werden:

```
% Definition
mod_fun = @(A,x) A*(x.^2+x.^3) ;
A = 5 ;
fun_han = @(x) -1*mod_fun(A,x) ;
% Berechnung
x  = linspace(-1,1,50) ;
y1 = mod_fun(A,x) ;
y2 = fun_han(x) ;
```

Hier unterscheidet sich `fun_han` durch zwei Eigenschaften von `mod_fun`:

- Es wurde `mod_fun` mit -1 multipliziert.
- Es wurde `fun_han` so erzeugt, dass `A` keinen Inputparamter mehr darstellt. In diesem Fall muss die Variable `A` existieren, bevor der Handle definiert wird. Der Wert den `A` zu diesem Zeitpunkt hat wird dann fix in die Funktion integriert. Ändern Sie `A` nach der Definition und rufen `fun_han` nochmal auf, ändert sich das Ergebnis nicht.

```
A = 5 ;
fun_han = @(x) A*sin(x) ;
fun_han(pi/2) ; % -> 5
A = 2 ;
fun_han(pi/2) ; % -> IMMER NOCH 5 !!!
% aber
fun_han = @(x) A*sin(x) ;
fun_han(pi/2) ; % -> 2
```

Will man einen `function_handle` an eine Funktion übergeben, so kann man den `function_handle` direkt beim Aufruf der Funktion definieren. Hier berechnet man z.B. das Integral $\int_0^{10} dx(x^2 + x^3)$

```
A = 5 ;
flaeche = quadl(@(x) A*(x.^2+x.^3) ,0,10) ;
```

Mehr zur Übergabe von Unterprogrammen finden man in 9.4.

Schlussendlich geht es noch um eine Verwendung ohne Übergabeparameter

```
A = 5; x = 1:3 ;
fun_t = @() A*x ;
```

Hier gibt es keine Übergabeparameter, da `A` und `x` schon vorher definiert sind. Sowohl bei der Definition als auch bei der Ausführung muss das Klammerpaar `()` verwendet werden.

```
y = fun_t()   % -> liefert [5 10 15]
f = fun_t     % -> @() A*x
```

wobei die erste Zeile das gewünschte Ergebniss der Rechnung liefert und die zweite Zeile eine Kopie der Funktion erzeugt, die danach wie die ursprüngliche Funktion verwendet werden kann, d.h., `f()` liefert dann das Gleiche wie `fun_t()`.

Hier seien noch die beiden `inline`-Funktionen aus 9.2 einer Lösung mit `function_handle` gegenübergestellt:

```
f1 = inline('x.^n .* exp(-x.^2)', 'x', 'n');
h1 = @(x,n) x.^n .* exp(-x.^2);
f2 = inline('m*exp(-n*(x.^2 + y.^2))', 'x', 'y', 'm', 'n');
h2 = @(x,y,m,n) m*exp(-n*(x.^2 + y.^2));
```

9.4 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 9.2 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.

$$A_1 = \int_a^b dx\; x^n \exp(-x^2) \tag{9.1}$$

$$A_2 = \int_a^b \int_c^d dxdy\; m \exp\left(-n(x^2+y^2)\right) \tag{9.2}$$

```
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)
```

Nach der Variante mit `inline`-Funktionen und Unterprogrammen in Files, gibt es natürlich auch die Möglichkeit anonyme Funktionen, wie sie in 9.3 behandelt werden, zu verwenden,

```
A1 = quadl(h1,  a, b, TOL, ANZEIGE, n)
A1 = quadl(h1,  a, b, [],  [],       n)

A2 = dblquad(h2,  a, b, c, d, TOL, @quadl,  m, n)
A2 = dblquad(h2,  a, b, c, d, [],  [],      m, n)
```

wobei hier die beiden Funktionen `h1` und `h2` in 9.3 als

```
h1 = @(x,n) x.^n .* exp(-x.^2);
h2 = @(x,y,m,n) m*exp(-n*(x.^2 + y.^2));
```

definiert wurden. Eine sehr bequeme Möglichkeit ist auch die Variante mit Variablen, die bereits vor der Definition der Funktionen definiert werden,

```
m = 2; n = 3;
h1m = @(x) x.^n .* exp(-x.^2);
h2m = @(x,y) m*exp(-n*(x.^2 + y.^2));
```

wobei hier die beiden Funktionen formal nicht mehr von `m` und `n` abhängen (es wird in der Funktion bereits `2` bzw `3` verwendet. Dann kann man für die Integrale

```
A1 = quadl(h1m, a, b)
A2 = dblquad(h2m, a, b, c, d)
```

schreiben. Ändert man jetzt aber im Programmablauf `m` und `n`, dann muss man auch `h1m` und `h2m` neu zuweisen. Daher muss man darauf vor allem in Schleifen Bedacht nehmen.

9.5 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.

9.6 Beispiele

Einfaches Beispiel (tfunc1.m):

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

% Der erste Kommentarblock wird bei Aufruf des Befehls
%            help tfunc1
% angezeigt

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

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

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

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

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

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

% Die Variable nargin enthaelt nach dem Aufruf der Funktion
% tfunc2a die Anzahl der übergebenen Input-Variablen. Die
% Variable nargout enthält die Anzahl der Output-Variablen.
%
% z.B.:  [r1,r2] = tfunc2a([1:10],3,4) => nargin=3, nargout=2
%         r1     = tfunc2a([1:10],3)   => nargin=2, nargout=1
```

```
%                    tfunc2a                => nargin=0, nargout=0
%
% Diese Variablen kann man nun zum Steuern des Verhaltens der
% Funktion, zum Setzen von Defaultwerten und zur Entscheidung,
% welche Output-Variablen berechnet werden sollen, verwenden.

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

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

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

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

```
if isempty(b), b = 2; end
f1 = a * x.^2 .* sin(x);
if nargout>1, f2 = f1 + b * x; end
```

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

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

(tfunc2c.m):

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

```
%   x  double array
%   a  double array, optional, default a=[1,2]
%   b  double array, optional, default b=[2,4]
% Output:                       Summation ueber alle k
%   f1 double array f1 = a(k) * x.^2 .* sin(a(k)*x)
%   f2 double array f2 = a(k) * x.^2 .* sin(a(k)*x) + b(k) * x

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

Beispiel mit Fallunterscheidung (tfunc3.m):

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

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

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

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

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

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

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

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

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

  if nargin<1, error('Aufruf: [f1,f2]=tfunc3b(f,x,a,b)'); end
  if ~ischar(varargin{1}) % Erstes Element kein String
    % Rekursiver Aufruf von tfunc3b
```

```
  [f1,f2] = tfunc3b('sin',varargin{:});
  % Übergabe von 'sin' und aller Parameter vom ersten Aufruf.
  return % Beendet ersten Aufruf der Funktion
end

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

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

% Die String-Variable f wird nun einerseits als Schalter,
% aber auch zur Konstruktion der Funktion verwendet.
switch f
 case {'sin', 'cos', 'tan', 'cot'}
  % Zusammensetzen einer Zeichenkette [s1,s2,s3]
  e_string = ['a * x.^2 .* ',f,'(x)'];
  disp(['Berechnung mit: ',e_string]);
```

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

9.7 Beispiele - Umwandlungsroutinen

Interessierte Leser finden hier zwei selbst geschriebene Umwandlungsroutinen zwischen einem `function_handle` und einer `inline`Funktion.

Umwandlung von `function_handle` auf `inline` (`func2inline.m`):

```
function fi = func2inline(fh)
% Converts funftion_handle fh into
% inline-function fi.

if nargin < 1 || ~isa(fh,'function_handle')
    error('No function_handle');
end

sfh = func2str(fh);      % function string
pos = strfind(sfh,'@'); % locate arguments
if isempty(pos), error('No arguments'); end

% arguments between ( and )
sfh = sfh(pos+1:end);
arg_left  = strfind(sfh,'(');
arg_right = strfind(sfh,')');
```

```
args = strtrim(sfh(arg_left+1:arg_right-1));
% formula
form = strtrim(sfh(arg_right+1:end));

% list of arguments in cell array
arglist = {};
if isempty(args), error('No arguments'); end
pos = strfind(args,',');
i1 = 1;
for k = 1:length(pos)
    i2 = pos(k)-1;
    arglist{k} = args(i1:i2);
    i1 = i2 + 2;
end
if isempty(k), k = 0; end
arglist{k+1} = args(i1:end);

% inline-function
fi = inline(form,arglist{:});
```

Umwandlung von `inline` auf `function_handle` (`inline2func.m`):

```
function fh = inline2func(fi)
% Converts inline-function fi into
% funftion_handle fh.

if nargin < 1 || ~isa(fi,'inline')
    error('No inline-function');
end

args = argnames(fi); % Cell with arguments
form = formula(fi);  % formula
% make comma-seperated list of arguments
arglist = '';
for k = 1:length(args)
    arglist = [arglist,args{k},','];
end
arglist = arglist(1:end-1);
% create function handle
fh = eval(['@(',arglist,') ',form]);
```

Kapitel 10

Zeichenketten

10.1 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 11

Strukturen und Zellen

Dies ist ein Platzhalter für ein geplantes Kapitel.

In der Zwischenzeit beschränkt sich der Inhalt auf einen Link auf ein MATLAB-Dokument über Datenstrukturen.

11.1 Strukturen

Info über Strukturen

11.2 Zellen

Info über Zellen

Kapitel 12

Polynome

12.1 Grundlagen

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

$$p_1 x^{n-1} + p_2 x^{n-2} + p_3 x^{n-3} + \cdots + p_n \ , \tag{12.1}$$

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,

$$y_i = p_1 x_i^{n-1} + p_2 x_i^{n-2} + p_3 x_i^{n-3} + \cdots + p_n \ , \tag{12.2}$$

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

$$y_i = x_i^3 + 2x_i^2 + x_i + 3 \ , \tag{12.3}$$

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,

$$y_i = ((p_1 x + p_2)\, x + p_3)\, x + p_4 \ , \tag{12.4}$$

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.

12.2 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

$$\det\left(\mathbf{A} - \lambda\mathbf{I}\right) \ , \tag{12.5}$$

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

Dies soll am Beispiel von

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \quad , \quad \mathbf{c} = \det \begin{bmatrix} 2-\lambda & 1 \\ 2 & 3-\lambda \end{bmatrix} \ , \tag{12.6}$$

demonstriert werden, wobei sich hier als Lösung

$$\mathbf{c} = (2-\lambda)(3-\lambda) - 2 = \lambda^2 - 5\lambda + 4 \tag{12.7}$$

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

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 1 & & 1 & & & & \\ & & & 1 & & 2 & & 1 & & & \\ & & 1 & & 3 & & 3 & & 1 & & \\ & 1 & & 4 & & 6 & & 4 & & 1 & \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\ 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1 \end{array} \tag{12.8}$$

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

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{bmatrix} \tag{12.9}$$

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

$$p(x) = x^4 - 29x^3 + 72x^2 - 29x + 1 \; . \tag{12.10}$$

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)`

$$\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \tag{12.11}$$

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

12.3 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))`).

12.4 Differentiation und Integration von Polynomen

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

```
k = polyder(p)
```

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

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

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

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

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

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

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

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

12.5 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.

$$\begin{aligned} v(x) &= & x^3 + 2x^2 + 3x + 4 \\ u(x) &= & x + 2 \\ q(x) = v(x)/u(x) &= & x^2 + 3 \\ r(x) &= & -2 \end{aligned} \tag{12.12}$$

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.

12.6 Fitten mit Polynomen

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

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

$$q = \sum_{j=1}^{m} \left(y_j - p(x_j)\right)^2 \stackrel{!}{=} \text{Min} \ , \tag{12.13}$$

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 13

Input und Output

Dies ist ein Platzhalter für ein geplantes Kapitel.

In der Zwischenzeit beschränkt sich der Inhalt auf einen Link auf ein MATLAB-Dokument über Ein- und Ausgabe .

Kapitel 14

Anwendungen

14.1 Kurvenanpassung - Fitten

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

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

$$q = \sum_{j=1}^{m} (y_j - f(x_j, a))^2 \stackrel{!}{=} \text{Min} \ . \tag{14.1}$$

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

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

$$f(x, a) = \sum_{i=0}^{n} a_i x^i \ , \tag{14.2}$$

$$f(x, a) = a_0 + a_1 x + a_2 x^2 \ , \tag{14.3}$$

$$f(x, a) = \sum_{i=0}^{n} a_i f_i(x) \ , \tag{14.4}$$

$$f(x, a) = a_0 + a_1 \sin x + a_2 \cos x \ . \tag{14.5}$$

Manche Modellfunktionen können z.B. durch Logarithmierung linearisiert werden

$$f(x,a) = a_0 \exp(-a_1 x) \qquad \Rightarrow \hat{f}(x,a) = \ln a_0 - a_1 x \ , \tag{14.6}$$

$$f(x,a) = a_0 x^{a_1} \qquad \Rightarrow \hat{f}(x,a) = \ln a_0 + a_1 \ln x \ . \tag{14.7}$$

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

$$f(x,a) = a_0 \sin(a_1 x + a_2) \ , \tag{14.8}$$

$$f(x,a) = a_0 \exp\left(\frac{-(x-a_1)^2}{a_2}\right) \ , \tag{14.9}$$

$$f(x,a) = \frac{a_0}{a_1 + a_2 x} \ . \tag{14.10}$$

14.1.1 Auswahl der Modellfunktion

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

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

Ist die Wahl auf ein lineares Modell in Bezug auf die Parameter gefallen, kann man den Anweisungen in Abschnitt 14.1.2 folgen, sonst den Anweisungen im Abschnitt 14.1.4.

14.1.2 Lineares Fitten

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

$$\sum_{i=0}^{m-1} a_i x_j^i = y_j \ , \ j = 1 \ldots m \ , \tag{14.11}$$

welches in Matrixform als

$$X_{ji} a_i = y_j \tag{14.12}$$

geschrieben werden kann. In den m Spalten der Matrix $X_{ji} = x_j^i$ stehen somit die Spaltenvektoren x^{m-1}, x^{m-2} $\ldots$ x^0. Dieses bestimmte Gleichungssystem kann nun nach den Regeln der linearen Algebra (siehe Abschnitt 6.7) gelöst werden. In MATLAB lautet die Lösung mit Hilfe der Matrix-Links-Division `a=X\y`, wobei y ein Spaltenvektor sein muss.

Im Falle des Fittens hat man es normalerweise mit einer größeren Anzahl von Datenpunkten zu tun und möchte in den seltensten Fällen Modellfunktionen mit der gleichen Anzahl von Parametern verwenden. Damit hat man es mit einem überbestimmten Gleichungssystem zu tun, dass nicht mehr exakt gelöst werden kann. In MATLAB kann man für die Lösung eines solchen überbestimmten Gleichungssystems aber die gleiche Syntax verwenden. Sobald ein Gleichungssystem überbestimmt ist, löst MATLAB bei Verwendung des Befehls `a=X\y` das Gleichungssystem im Sinne des "Least-Squares"-Verfahrens entsprechend der Gleichung 14.1.

Die Unterscheidung zwischen linearen und nichtlinearen Modellfunktionen ist deswegen so wichtig, weil in der numerischen Umsetzung wesentliche Unterschiede bestehen. Im Fall von linearen Funktionen kann das Minimierungsproblem in Gleichung 14.1 immer in ein exakt lösbares lineares Gleichungssystem überführen. Wie man sich leicht überzeugen kann, erhält man für $k = 0 \ldots n$ aus $\frac{\partial q}{\partial a_k} = 0$ das lineare Gleichungssystem

$$X_{ki} a_i = b_k \ , \tag{14.13}$$

$$X_{ki} = \sum_{j=1}^{m} f_i(x_j) f_k(x_j) \ , \tag{14.14}$$

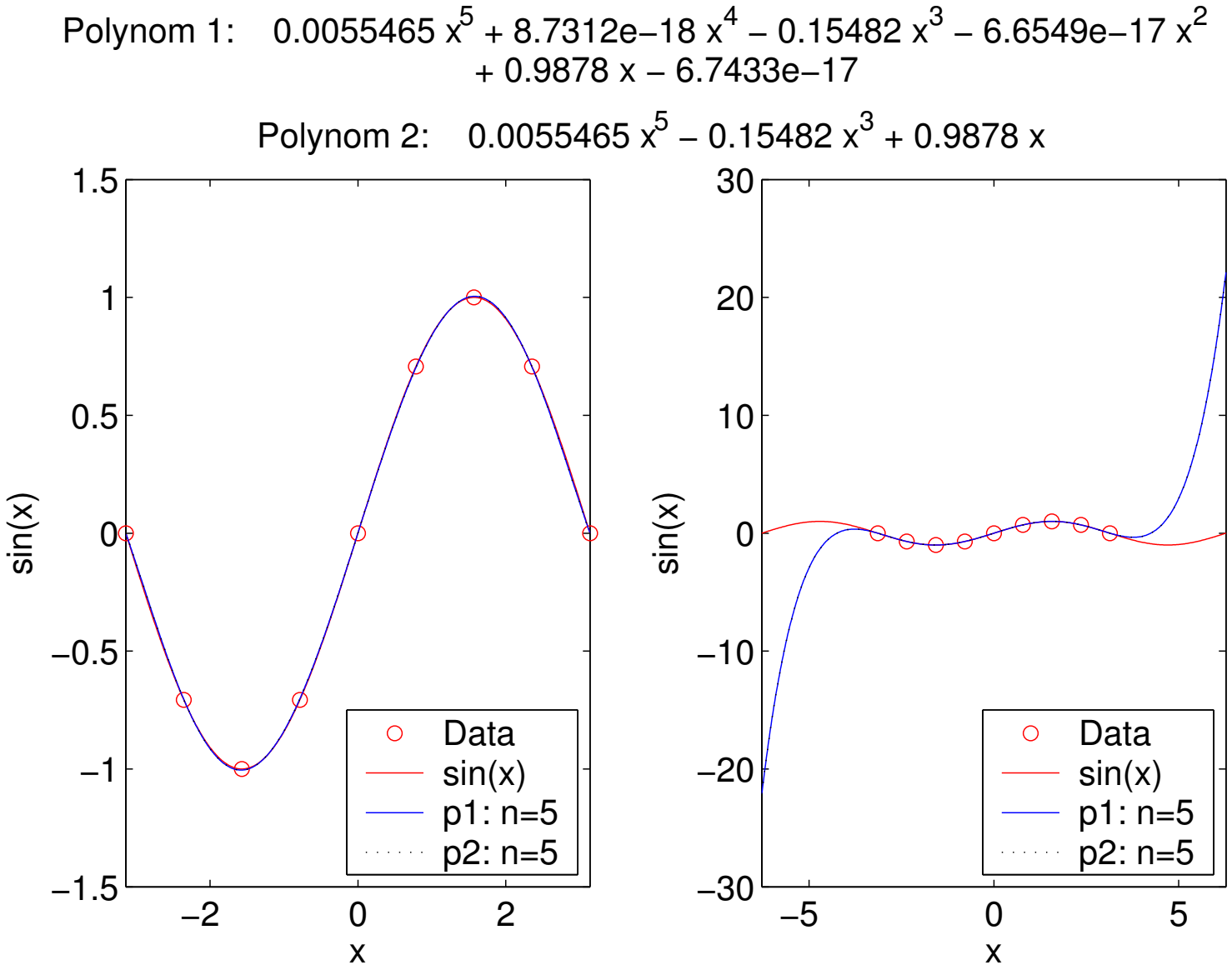
$$b_k = \sum_{j=1}^{m} y_j f_k(x_j) \ . \tag{14.15}$$

Diesen Vorteil hat man bei einem nichtlinearen Zusammenhang natürlich nicht. In diesem Fall ist man dann auf näherungsweise Verfahren zur Minimierung von Gleichung 14.1 angewiesen.

14.1.2.1 Polynom-Fit

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

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



Diese Darstellung demonstriert drei interessante Aspekte:

- Der Fit mit einem Polynom 5-ten Grades ist innerhalb des Datenbereichs sehr gut.

- Außerhalb des Datenbereichs bricht die gute Übereinstimmung sehr rasch zusammen, da Polynome natürlich nicht die beste Modellfunktion für Schwingungen sind. Der Grund dafür ist, dass alle Polynome für $x \to \pm\infty$ nach $\pm\infty$ "explodieren".

- Wie auf Grund der Reihenentwicklung für den Sinus nicht anders zu erwarten, sind die Koeffizienten für gerade Potenzen von x nahezu Null.

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

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

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

14.1.2.2 Allgemeiner linearer Fit

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

$$f(x, a) = a_1 f_1(x) + a_2 f_2(x) \ , \tag{14.16}$$
$$f_1(x) = \exp\left(-0.2\ x\right) \ , \tag{14.17}$$
$$f_2(x) = \sin(4x) \exp\left(-0.4\ x\right) \ , \tag{14.18}$$

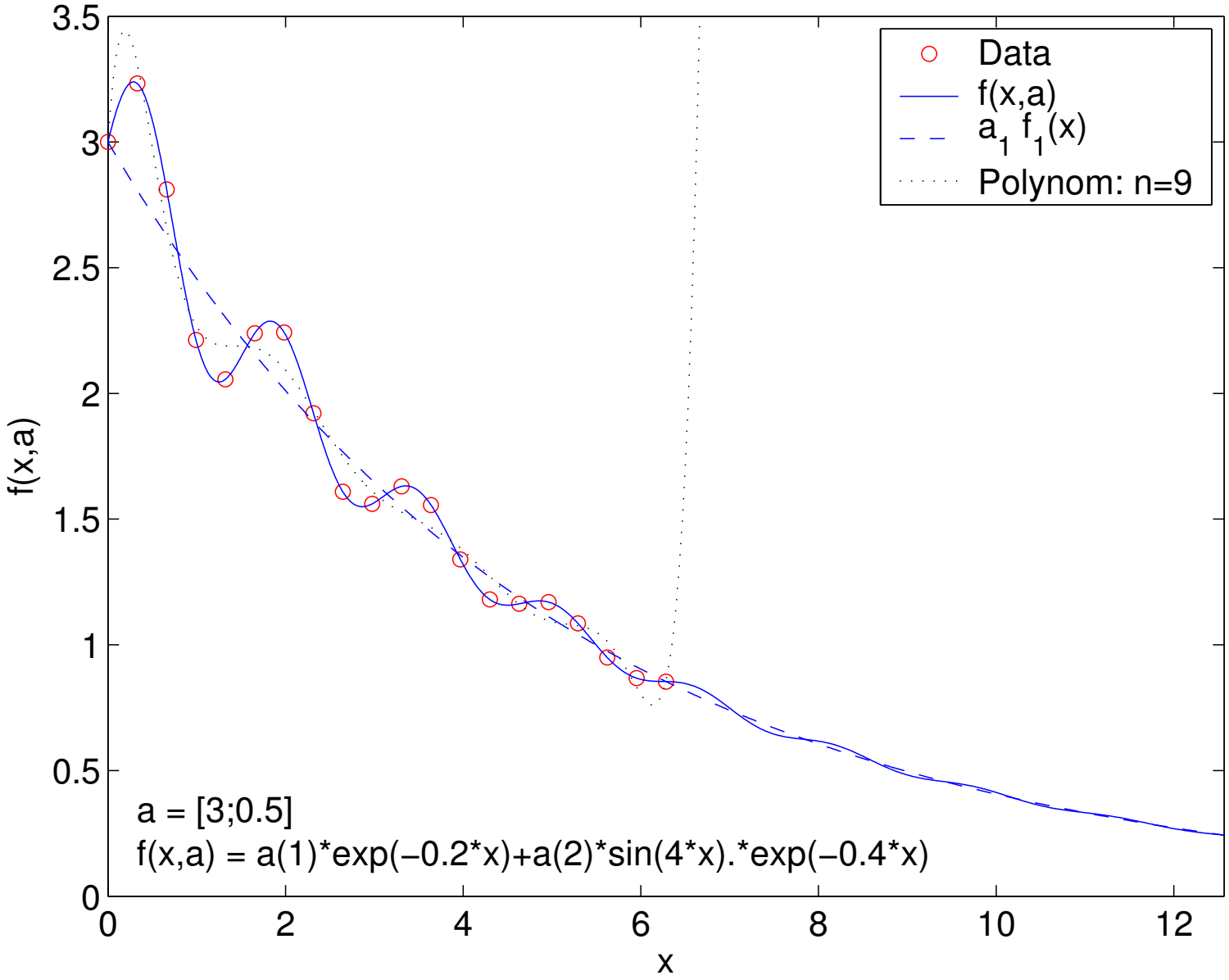
gegeben ist, an die Datenpunkte in den Vektoren `xd` und `yd` angepasst werden. Zuerst kann man in diesem Fall die Funktionen als `inline`-Funktionen definieren:

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

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

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

Mit diesen Daten kann man nun die Funktionen darstellen.



Hier fällt nun auf, dass

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

14.1.3 Exponentieller Fit

Beim radioaktiven Zerfall folgt die Intensität der Strahlung folgendem Gesetz

$$I(t) = I_0 \exp\left(-\frac{t}{\tau}\right) \ , \tag{14.19}$$

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

$$\hat{I}(t) = a_1 t + a_2 \ , \tag{14.20}$$

$$\hat{I} = \ln I \ , \tag{14.21}$$

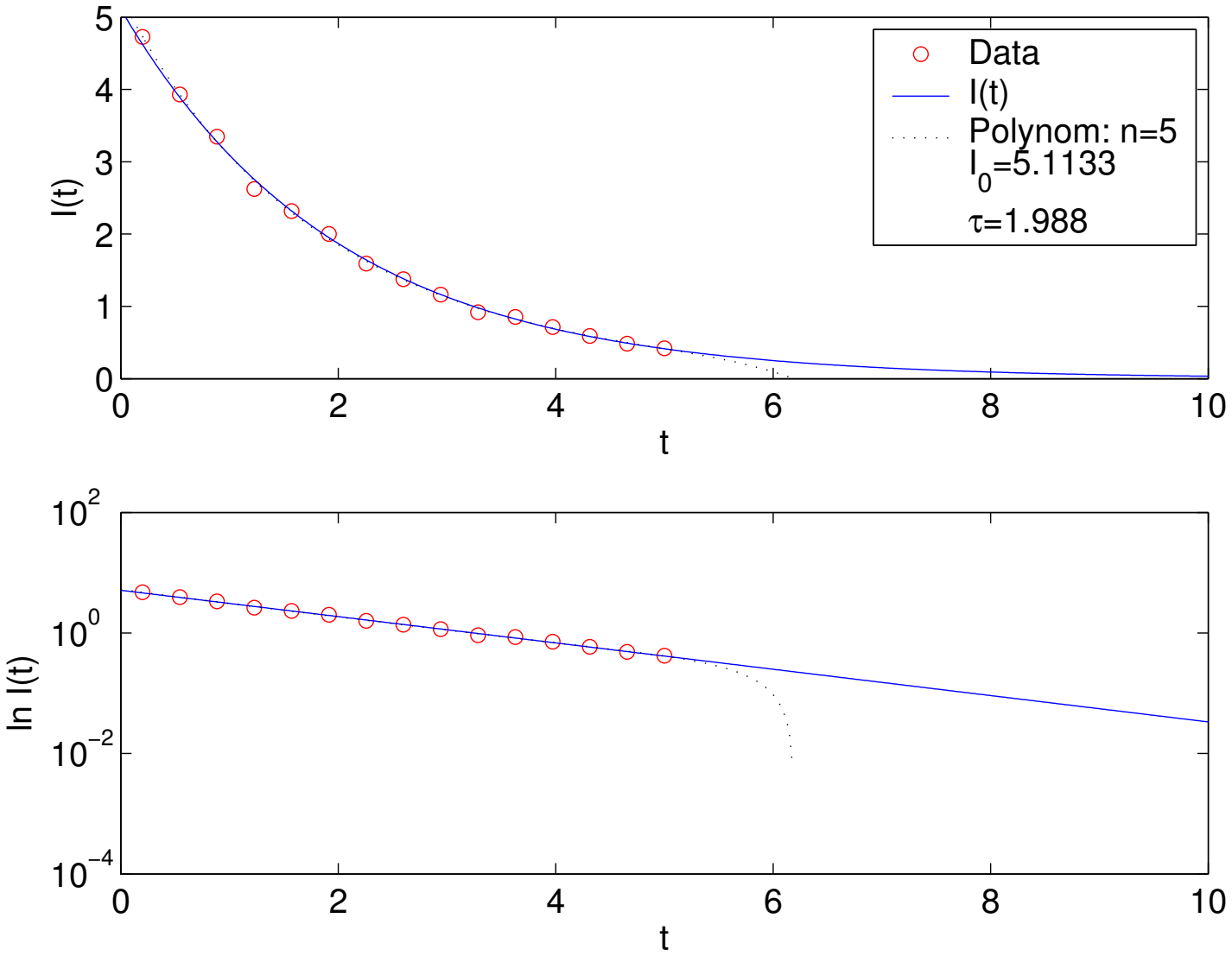
$$a_1 = -\frac{1}{\tau} \ , \tag{14.22}$$

$$a_2 = \ln I_0 \ , \tag{14.23}$$

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

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

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



Der zusätzlich berechnete Polynomfit erweist sich natürlich wieder als untauglich.

14.1.4 Nichtlineares Fitten

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

$$y(t) = A \sin(\omega t + \phi) \ . \tag{14.24}$$

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

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

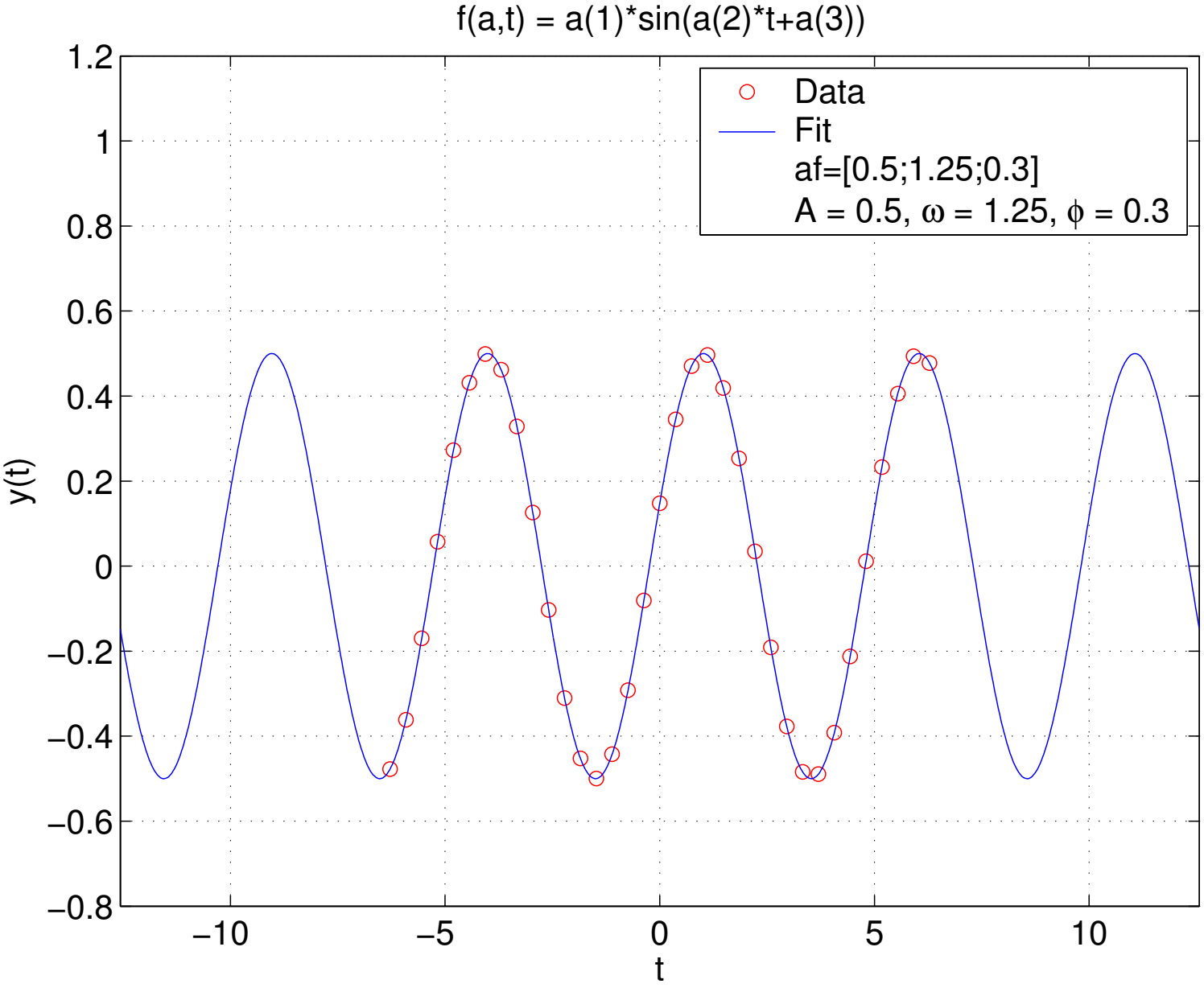
Wichtig dabei ist, dass alle Parameter (hier A, ω, ϕ) in einen Vektor zusammengefasst werden (hier `a`), und dass dieser Vektor an erster Stelle in der Übergabeliste steht.

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

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

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

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


f(a,t) = a(1)*sin(a(2)*t+a(3))
Data
Fit
af=[0.5;1.25;0.3]
A = 0.5, ω = 1.25, ϕ = 0.3
y(t)
t
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−10
−5
0
5
10

Warum braucht man nun einen Startwert. Im Unterschied zum linearen Fitten, wo das zugehörige Gleichungssystem immer eine eindeutige Lösung besitzt, die direkt ermittelt werden kann, benötigt man hier ein iteratives Verfahren. Dabei wird ausgehend von einem Startwert das Minimum einer Funktion gesucht, in dem man sich Schritt für Schritt dem Minimum nähert. Dazu gibt es viele Möglichkeiten (z.B.: Gauss-Newton). Entscheidend ist also der Unterschied, das es kein direktes Verfahren gibt um die Lösung zu finden. Nichtlineare Probleme haben dazu auch häufig mehrere (oft viele) ”lokale” Minima, interessiert ist man aber am so genannten ”globalen” Minimum, welches die ”bestmögliche” Lösung des Problems darstellt. Daher ist eine gute Wahl des Startwertes meist eine essentielle Vorleistung für eine gute Lösung. Meist findet man diese durch ”clevere” Betrachtung der Daten.

Als weiteres Beispiel soll hier die Funktion aus Gleichung 14.16 verallgemeinert werden, indem alle Parameter veränderlich gemacht werden,

$$f(x,a) = a_1 f_1(x) + a_2 f_2(x) \ , \tag{14.25}$$
$$f_1(x) = \exp\left(-a_3 x\right) \ , \tag{14.26}$$
$$f_2(x) = \sin(a_4 x)\exp\left(-a_5 x\right) \ , \tag{14.27}$$

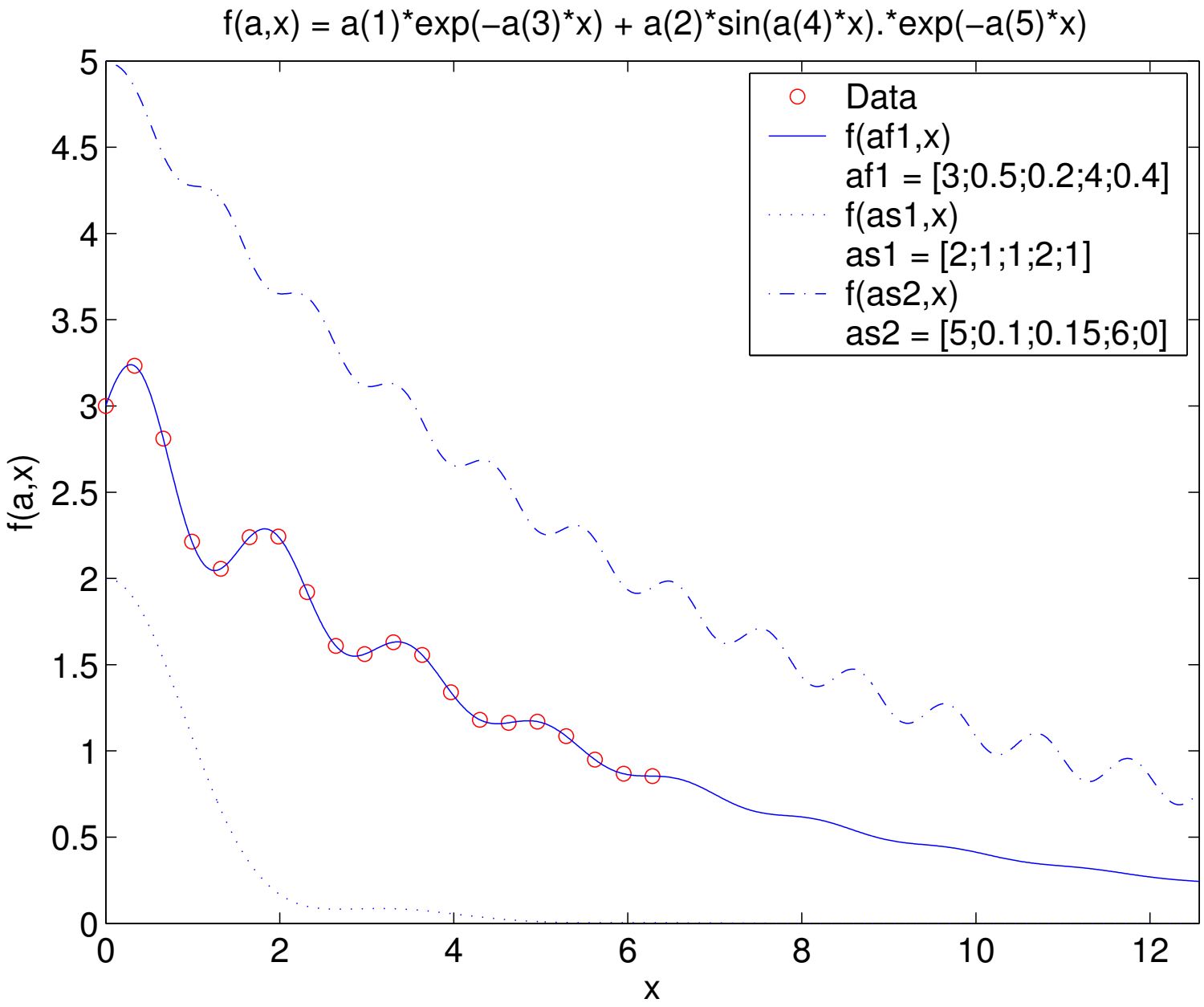
Dies ist nun ein nichtlineares Problem in a mit folgender Umsetzung in MATLAB:

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

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

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

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


f(a,x) = a(1)*exp(−a(3)*x) + a(2)*sin(a(4)*x).*exp(−a(5)*x)
Data
f(af1,x)
af1 = [3;0.5;0.2;4;0.4]
f(as1,x)
as1 = [2;1;1;2;1]
f(as2,x)
as2 = [5;0.1;0.15;6;0]
f(a,x)
x
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
2
4
6
8
10
12

Die Standard MATLAB-Funktion für das suchen von Minima ist die Routine `fminsearch`. Um diese beim vorliegenden Problem verwenden zu können, muss man eine Funktion programmieren, die den Skalarwert q der Gleichung 14.1 zurück gibt. Diese Funktion muss also die Summe der Abstandsquadrate an allen Datenpunkten berechnen und benötigt dafür den Parametervektor `a` und die Daten `xd` und `yd`. Am Beispiel der letzten Funktion kann das so aussehen:

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

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

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

und liefert die "besten" Parameter. An Stelle von `[]` kann man Optionen übergeben (siehe Hilfe zu `fminsearch`). Die Datenvektoren `xd` und `yd` werden von `fminsearch` an die Funktion `lqfunc` weitergegeben. Dieses Beispiel zeigt somit nochmals, dass eigentlich die Funktion 14.1 minimiert wird, und dass dies dann zur besten Annäherung der Modellfunktion an die Daten führt.

14.2 Interpolation

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

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

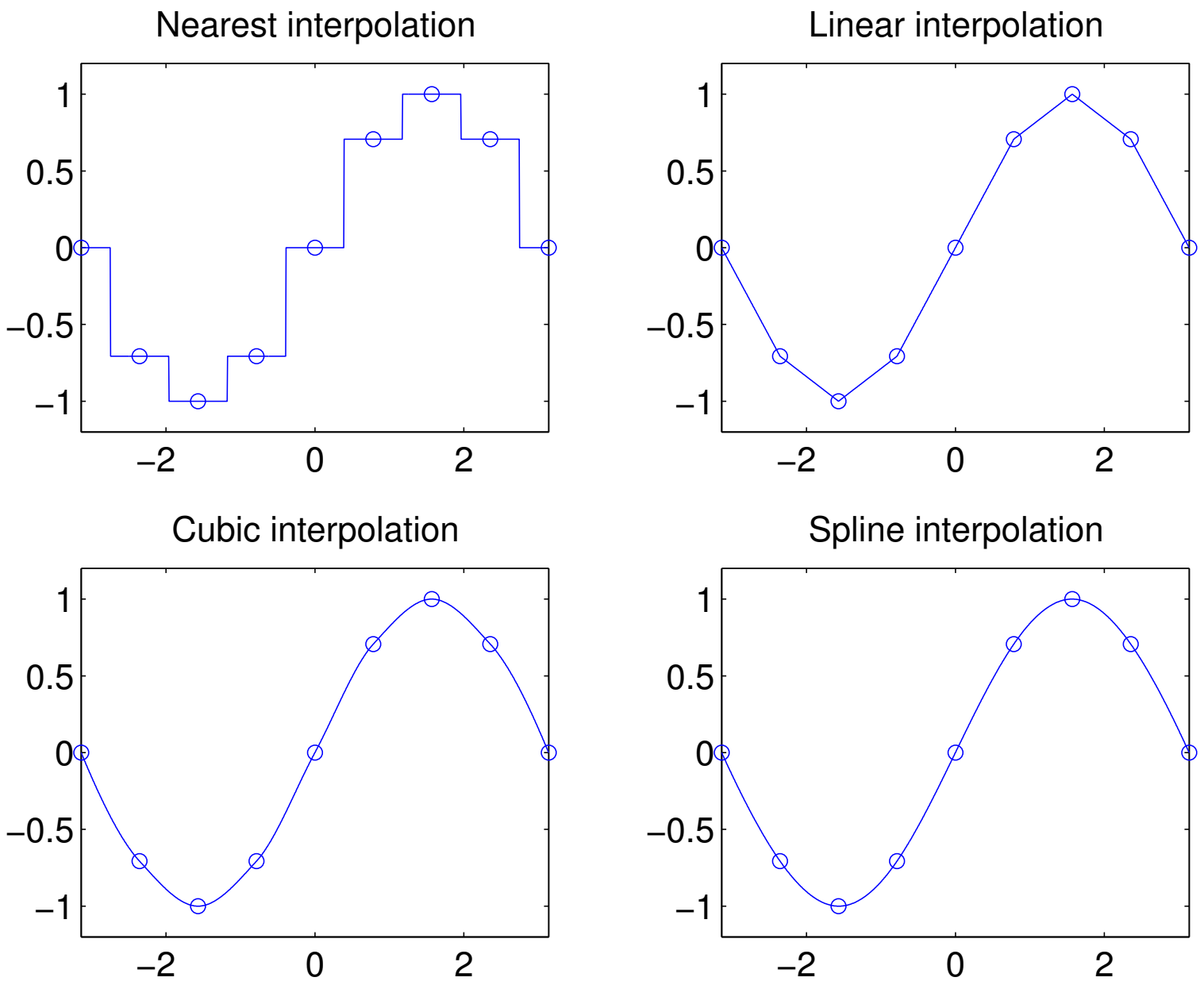
```
y = interp1(xd,yd,x,method)
```

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

Dies wird hier am Beispiel der Sinus-Funktion erläutert:

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

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

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

Die Ergebnisse für die einzelnen Methoden sehen folgendermaßen aus:



Folgende Methoden stehen zur Verfügung:

nearest Nächste Nachbar Interpolation

linear Lineare Interpolation

cubic Interpolation mit Polynomen dritten Grades

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

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

Kapitel 15

Graphische Ausgabe

15.1 MATLAB Dokumentation zur Erstellung von Graphiken

Zur Ergänzung des Kapitels werden hier Links auf MATLAB-Dokumente zu den Themen Erstellen von 2-D Graphiken und Erstellen von von 3-D Graphiken präsentiert.

15.2 Grundlagen

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

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

15.2.1 Graphikobjekte

15.2.1.1 Objekthierarchie

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

Die Eltern-Kind-Beziehung ist in Tabelle 15.1 durch die Rechtsverschiebung symbolisiert. So ist z.B. jede `line` ein Kind einer `axes`, diese ein Kind einer `figure`, und diese wiederum ein Kind des `root`. Auf allen Ebenen können nun Eigenschaften definiert und abgefragt werden.

15.2.1.2 Zugriff auf Objekte - Handles

Um nun Graphikobjekte eindeutig identifizieren zu können, braucht man einen Datentyp, der als Zeiger auf ein Graphikobjekt dient (Handle). Ein solcher Handle ist somit ein eindeutiger ”Identifier” für ein Graphikobjekt. Handles können Variablen zugewiesen werden und stehen damit im jeweiligen Programm zur weiteren Verfügung. Im Wesentlichen kann bei jedem MATLAB-Graphikbefehl eine Zuweisung erfolgen. Anstelle von `plot(x,y)` kann man schreiben `ph=plot(x,y)`, wobei nun in der Variablen `ph` der Handle für die entsprechende Linie gespeichert ist.

Tabelle 15.1: Hierarchie von Graphikobjekten in MATLAB.

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

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

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

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

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

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

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

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

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

```
set(ah, 'Color','green')
```

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

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

15.2.1.3 Spezielle Handles

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

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

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

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

kann man die Einstellung wieder auf MATLAB-Defaultwerte zurücksetzen.

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

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

15.3 Beispiele

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

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

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

15.3.1 Zweidimensionale Plots

Es gibt eine Reihe von Befehlen zur Darstellung zweidimensionaler Graphiken.

Tabelle 15.2: MATLAB Befehle zum Erzeugen einfacher zweidimensionaler Graphiken

`fplot('fun',[`x_{min}`,`x_{max}`])`	15.3.1.1	Zeichnet 'fun' im Bereich von x_{min} bis x_{max}
`plot(x,y)`	15.3.1.2	Zeichnet y als Funktion von x
`ezplot('fun',[`x_{min}`,`x_{max}`])`	15.3.1.3	erstellt u.a. implizite Funktionen, automatische Achsenbeschriftung
`comet(x,y,p)`	15.3.1.4	Zeichnet 2D Funktion in Form eines animierten 'Kometen'
`semilogx(x,y)`	15.3.1.5	Zeichnet 2D Funktion mit (10er-) logarithmischer x-Achse
`semilogy(x,y)`	15.3.1.6	Zeichnet 2D Funktion mit (10er-) logarithmischer y-Achse
`loglog(x,y)`	15.3.1.7	Zeichnet 2D Funktion mit (10er-) logarithmischer x- und y-Achse
`plotyy(`x_1`,`y_1`,`x_2`,`y_2`,'`f_1`','`f_2`')`	15.3.1.8	Erstellt 2 Graphen mit den Plotbefehlen f_1 und f_2 mit getrennten y - Achsen
`polar(phi,r)`	15.3.1.9	Zeichnet die Funktion r(phi) in Polarkoordinaten.

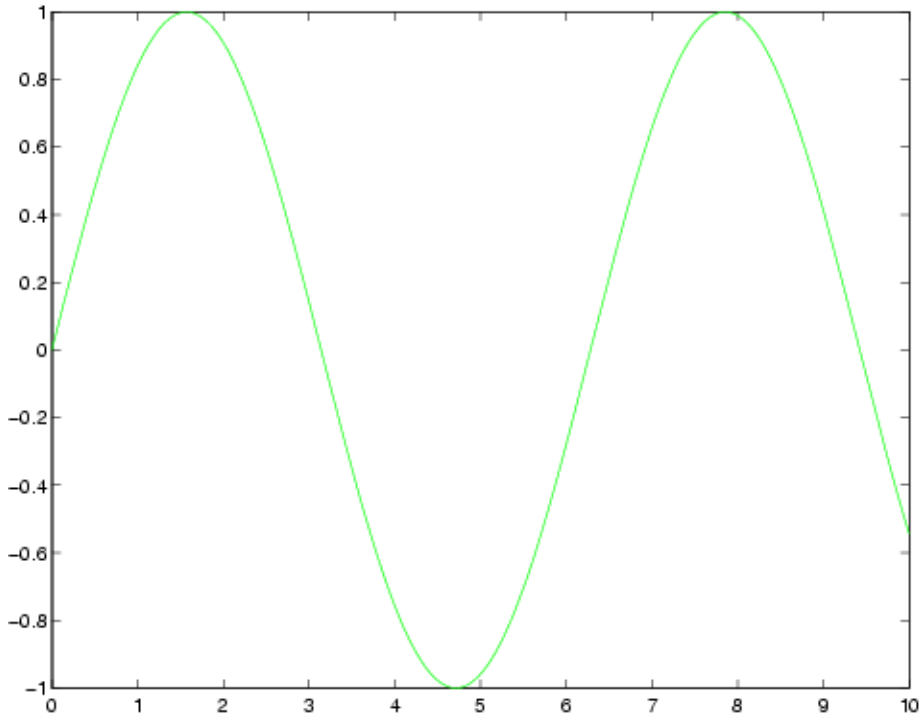
15.3.1.1 Fplot

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`.

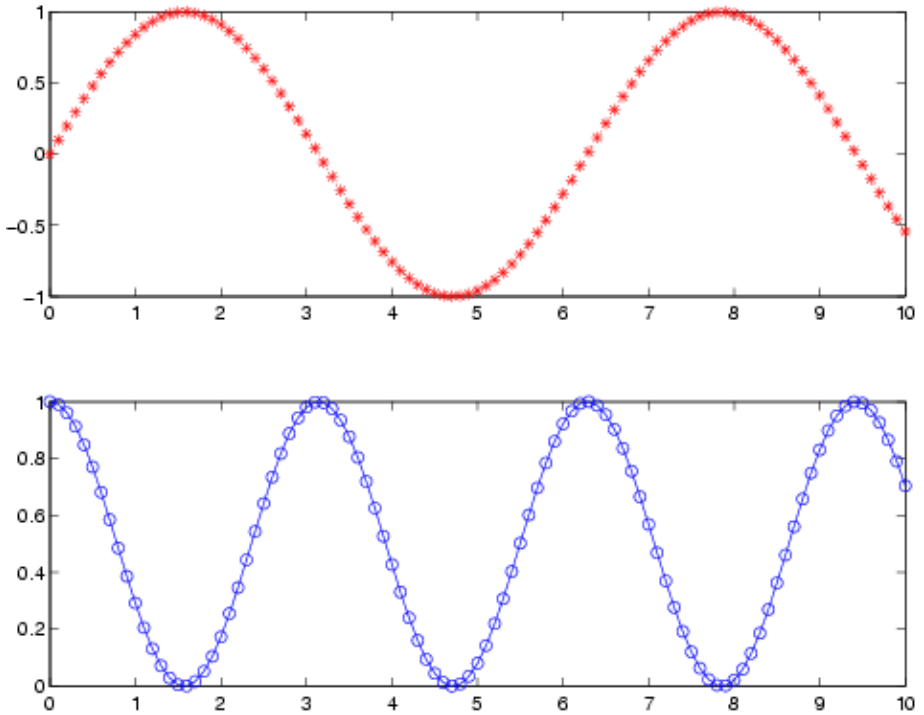
15.3.1.2 Plot

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`

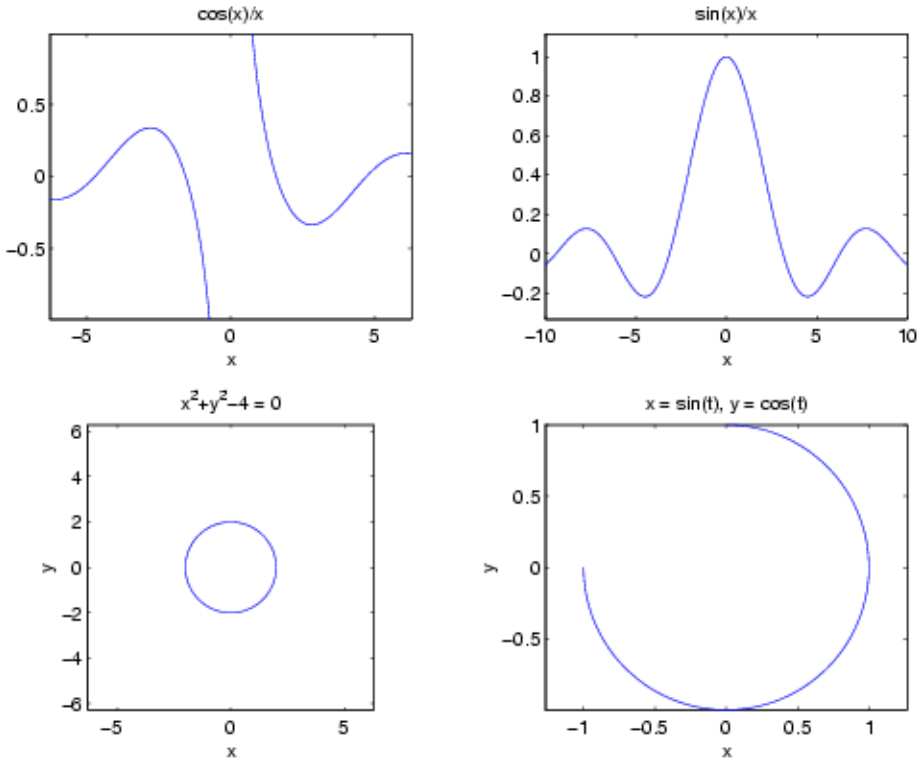
15.3.1.3 Ezplot

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])
```



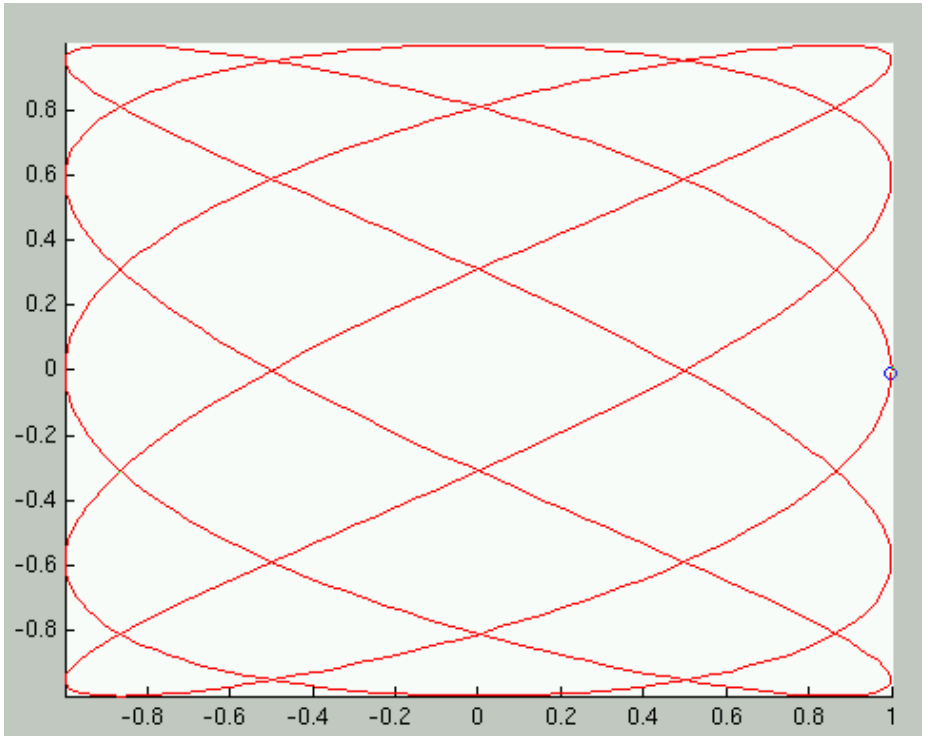
15.3.1.4 Comet

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.

15.3.1.5 Semilogx

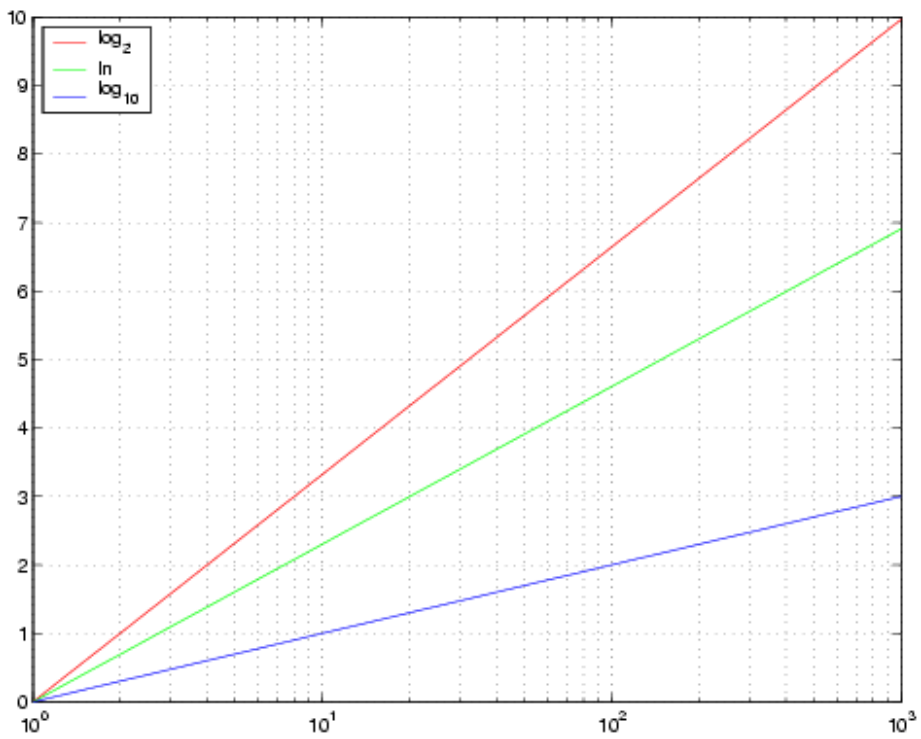
Erstellt eine 2 dimensionale Funktion mit logarithmischer x - Achse.

---

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.

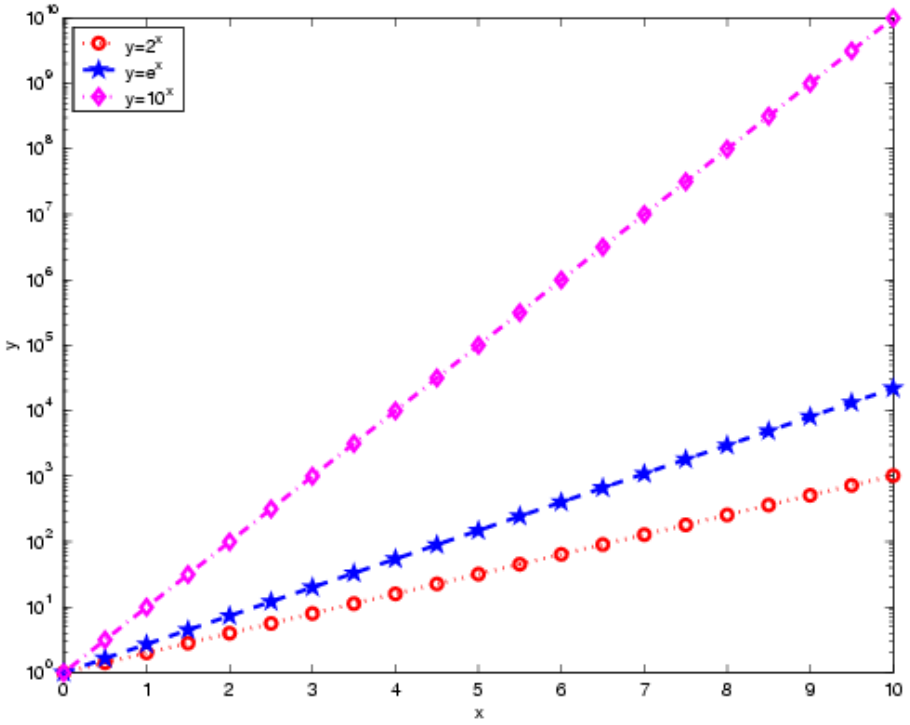
15.3.1.6 Semilogy

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.

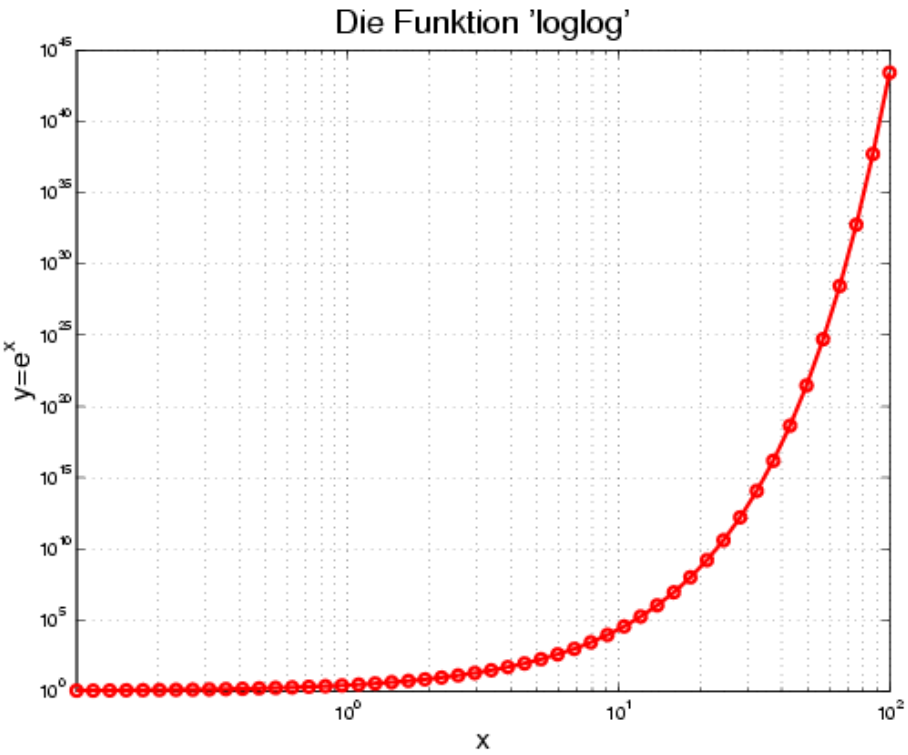
15.3.1.7 Loglog

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.

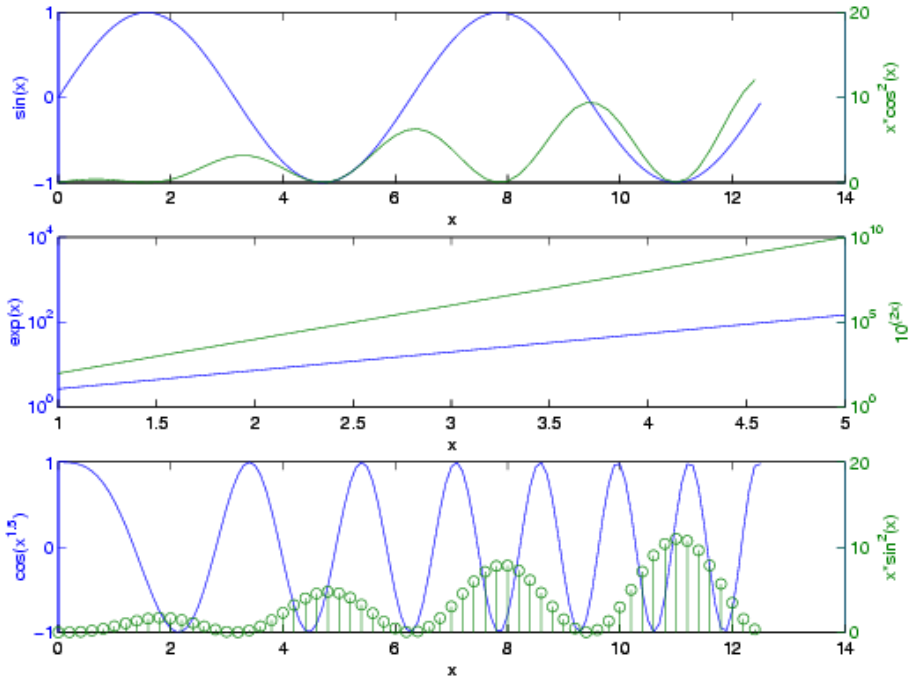
15.3.1.8 Plotyy

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`.

15.3.1.9 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
```

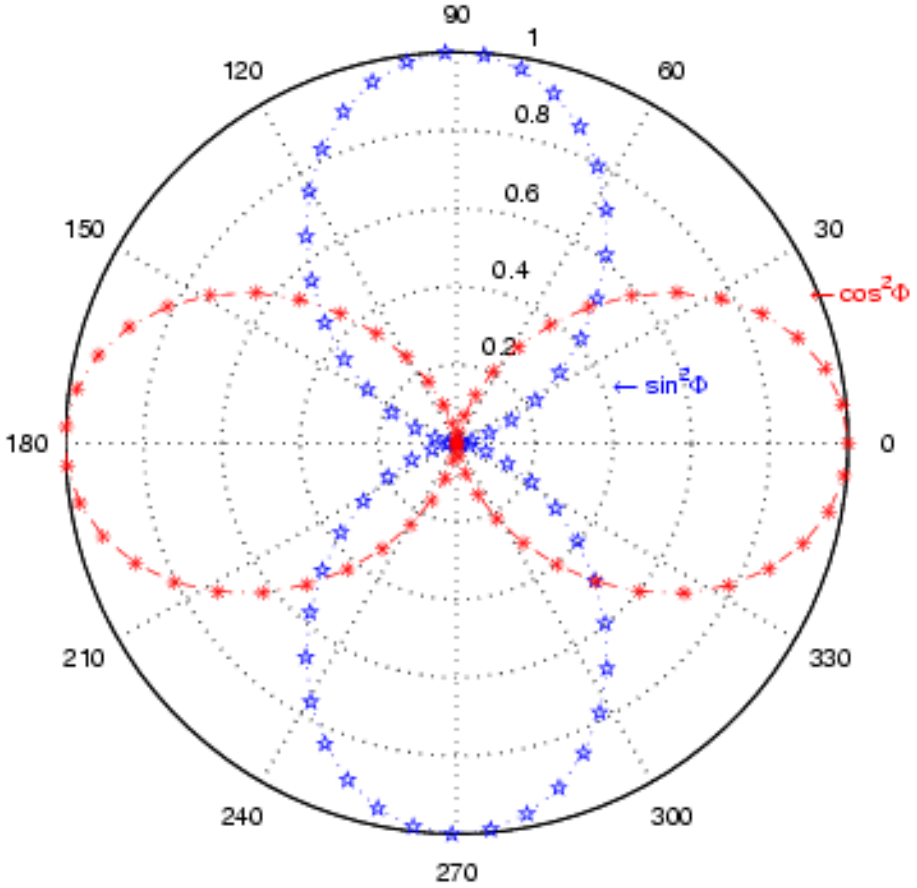


Tabelle 15.3: MATLAB Befehle zum Erzeugen von Balken- und Kreisdiagrammen

`hist(y,x)`	15.3.1.10	Erstellt ein Histogramm der Werte in y über jenen von x
`bar(x,y,'width','style')`	15.3.1.11	Stellt die Datenpaare [x,y] als vertikale Balken dar
`barh(x,y,'width','style')`	15.3.1.12	Stellt die Datenpaare [x,y] als horizontale Balken dar
`pie(x,'explode')`	15.3.1.13	Zeichnet ein 2D Kreisdiagramm der Daten von x

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.

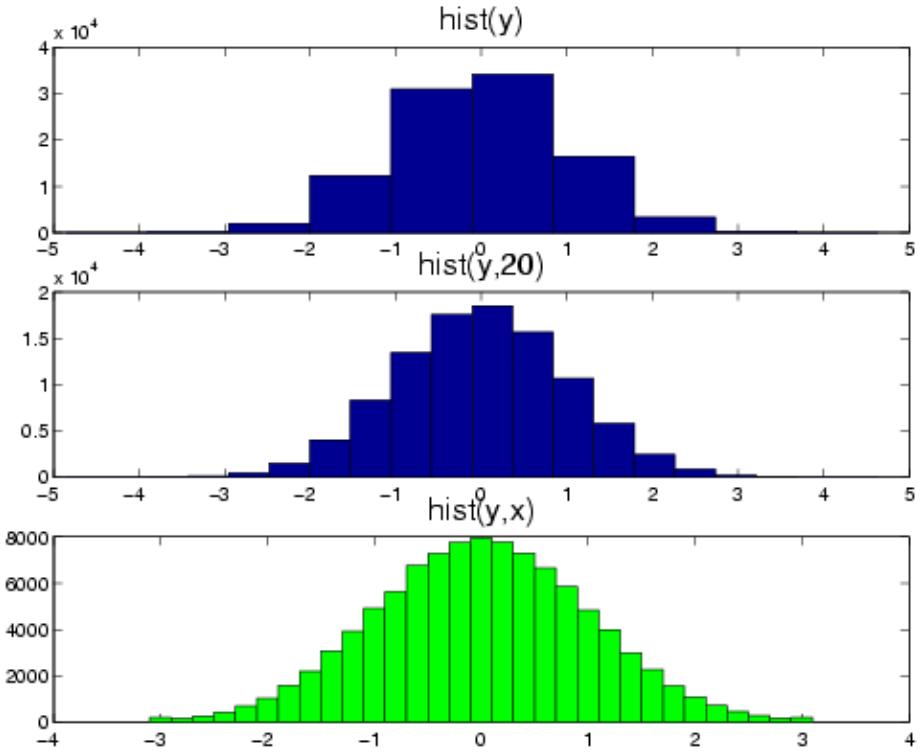
15.3.1.10 Histogramm

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.

15.3.1.11 Bar

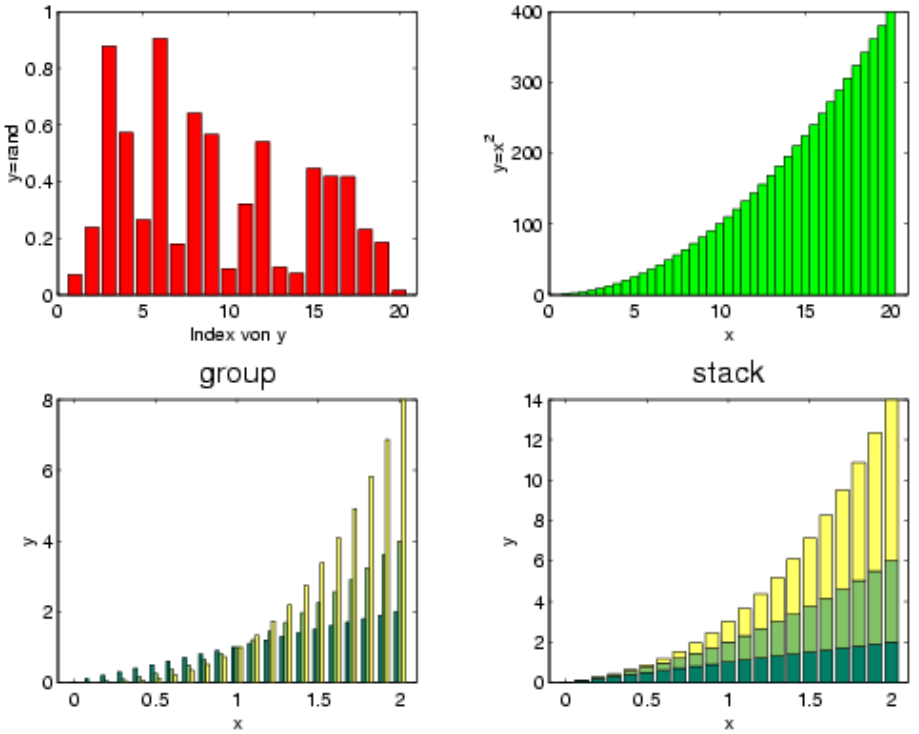
Erstellt an den Positionen von x vertikale Balken der Höhe y mit der relativen Balkenbreite 'width'. Die Balkengruppierung wird mit der Option 'style' gesteuert.

`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.

15.3.1.12 Barh

Die Datenpaare (x,y) werden in Form von horizontalen Balken des Stiles 'style' mit der relativen Breite 'width' veranschaulicht.

---

barh graph_barh.m

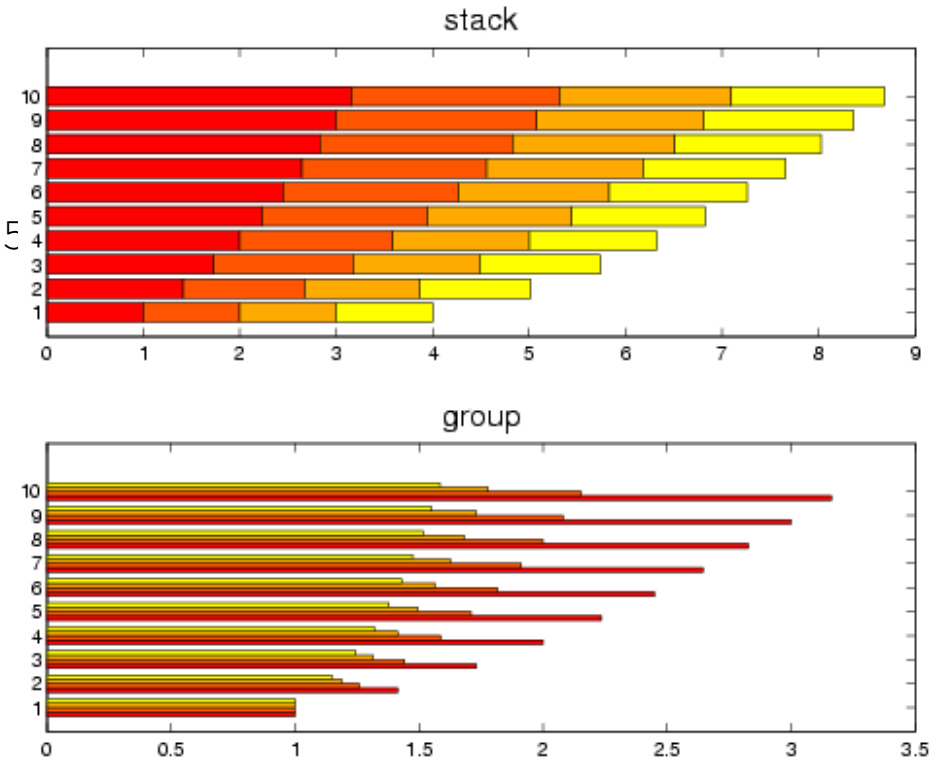
Wie im Beispiel 15.3.1.11 kann y eine Matrix sein.

```
x=(1:1:10)';
y=[x.^(1/2),x.^(1/3),x.^(1/4),x.^(1/5
```

```
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.

15.3.1.13 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)
```

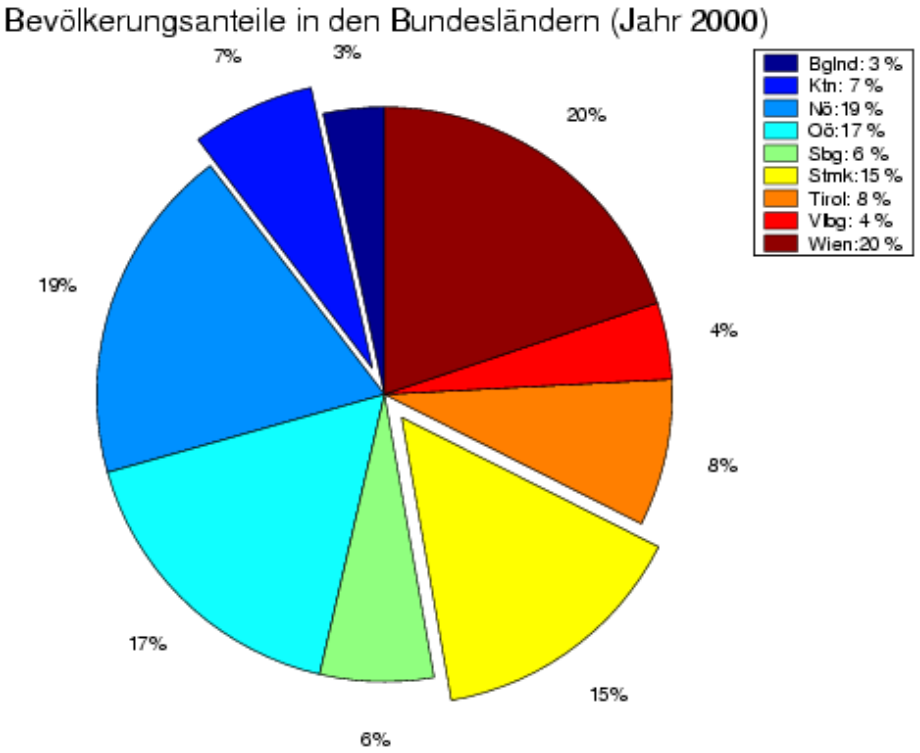


Tabelle 15.4: MATLAB Befehle zum Erzeugen von speziellen zweidimensionalen Graphiken

`stem(x,y)`	15.3.1.14	Zeichnet y=f(x) und verbindet Punkte mit x-Achse
`stairs(x,y)`	15.3.1.15	Erstellt Funktion y=f(x) in Form eines Stufendiagramms
`errorbar(x,y,e)`	15.3.1.16	Zeichnet y als Funktion von x samt Fehlerbalken der Länge e
`compass(x,y)`	15.3.1.17	Zeichnet y=f(x) und verbindet die Punkte durch Vektorpfeile mit dem Ursprung
`feather(u,v)`	15.3.1.18	Zeichnet die relativen Koordinaten u und v und verbindet die Punkte mit den jeweiligen Koordinatenursprüngen entlang der Abszisse
`scatter(x,y,r,c)`	15.3.1.19	Zeichnet Punkte an den Stellen (x,y) der Größe r sowie der Farbe c
`pcolor(x,y,c)`	15.3.1.20	Erstellt einen 'Pseudocolorplot' der Elemente c an den von den Punkten (x,y) definierten Positionen
`area(x,y)`	15.3.1.21	Füllt den Bereich zwischen y=f(x) und der Abszisse mit einer Farbe
`fill(x,y,c)`	15.3.1.22	Malt die durch (x,y) definierten Polygone mit der Farbe c aus
`contour(x,y,z)`	15.3.1.23	Zeichnet durch z=f(x,y) definierte Konturlinien
`contourf(x,y,z)`	15.3.1.24	Zeichnet durch z=f(x,y) definierte Konturlinien und füllt die Flächen dazwischen aus
`quiver(x,y,u,v)`	15.3.1.25	Erstellt von den Punkten (x,y) ausgehende Vek-

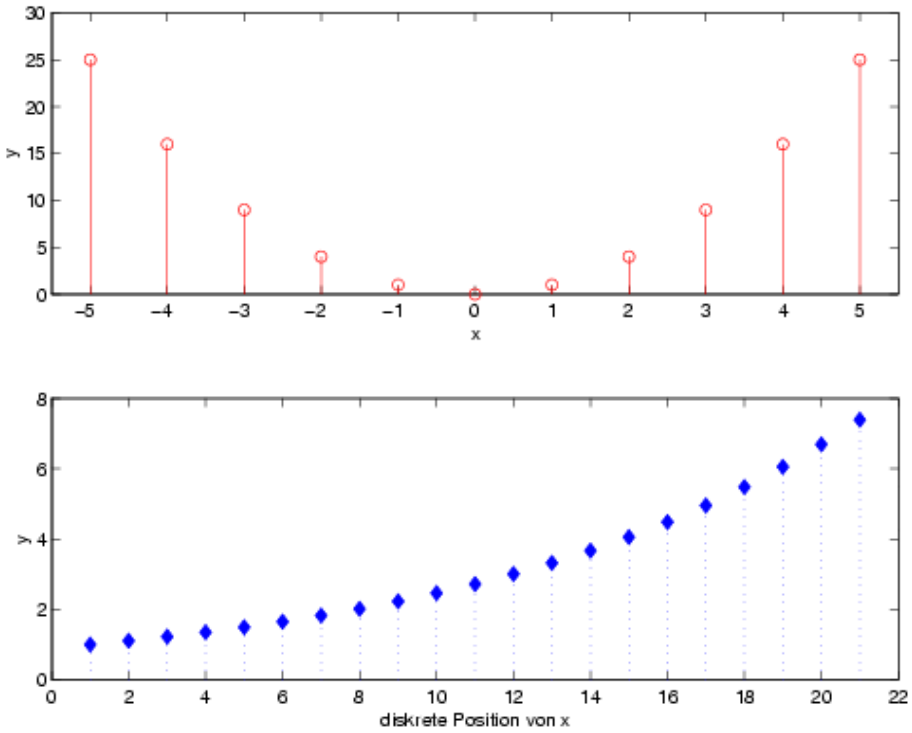
15.3.1.14 Stem

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])
```



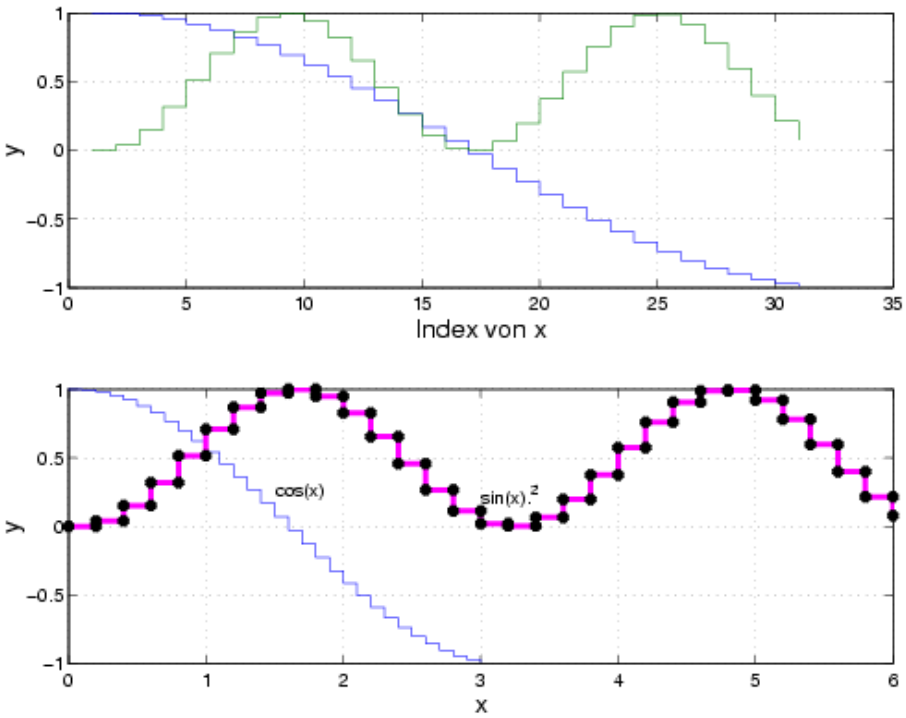
Im ersten Subplot werden die Achsengrenzen durch `axis([xmin,xmax, ymin, ymax])` geregelt, im zweiten Subplot mit dem Befehl `xlim`, wobei der Wertebereich der y-Achse unberührt bleibt.

15.3.1.15 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.

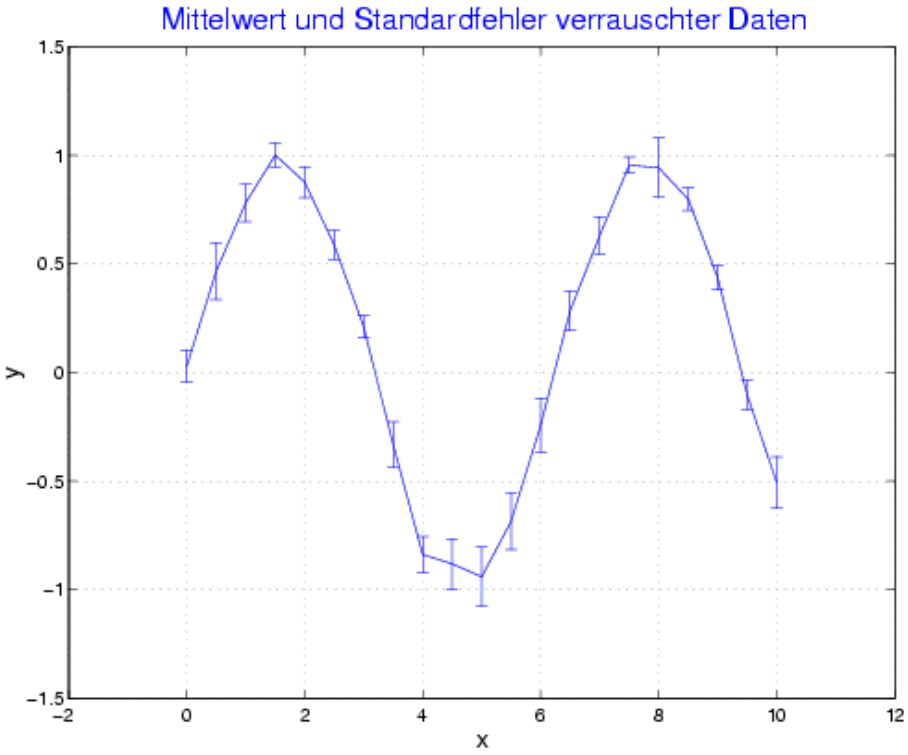
15.3.1.16 Errorbar

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));
```



15.3.1.17 Compass

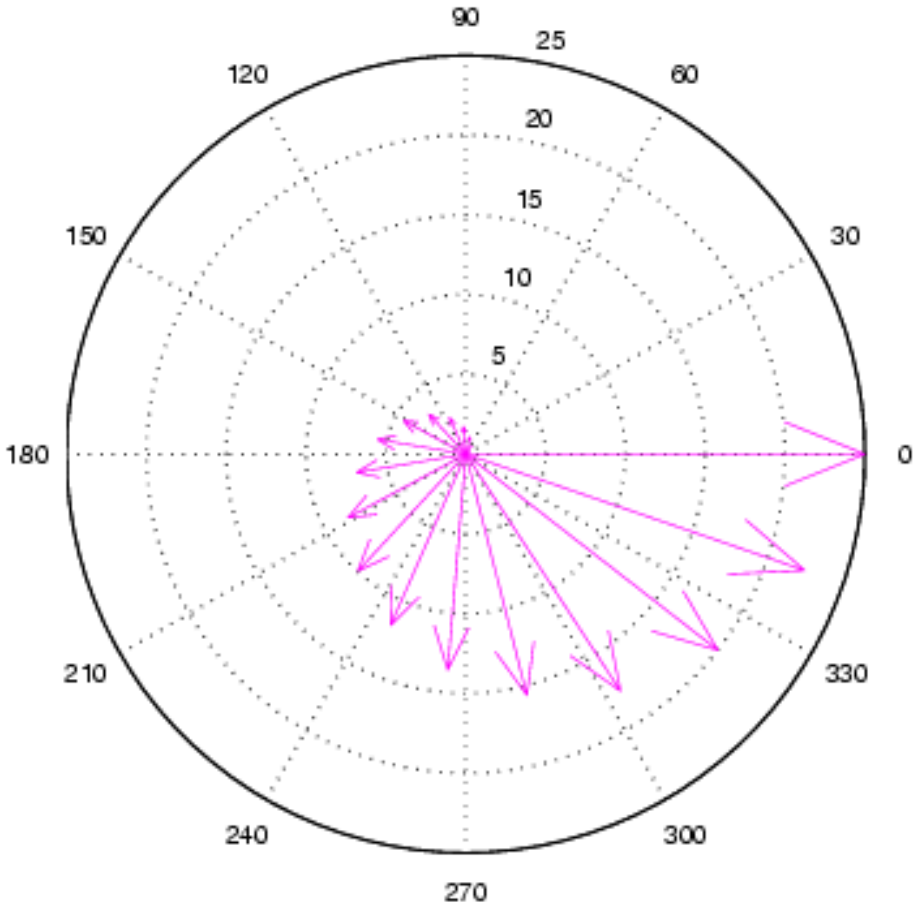
Zeichnet y als Funktion von x und verbindet die Punkte mit dem Koordinatenursprung durch Vektorpfeile.

---

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.

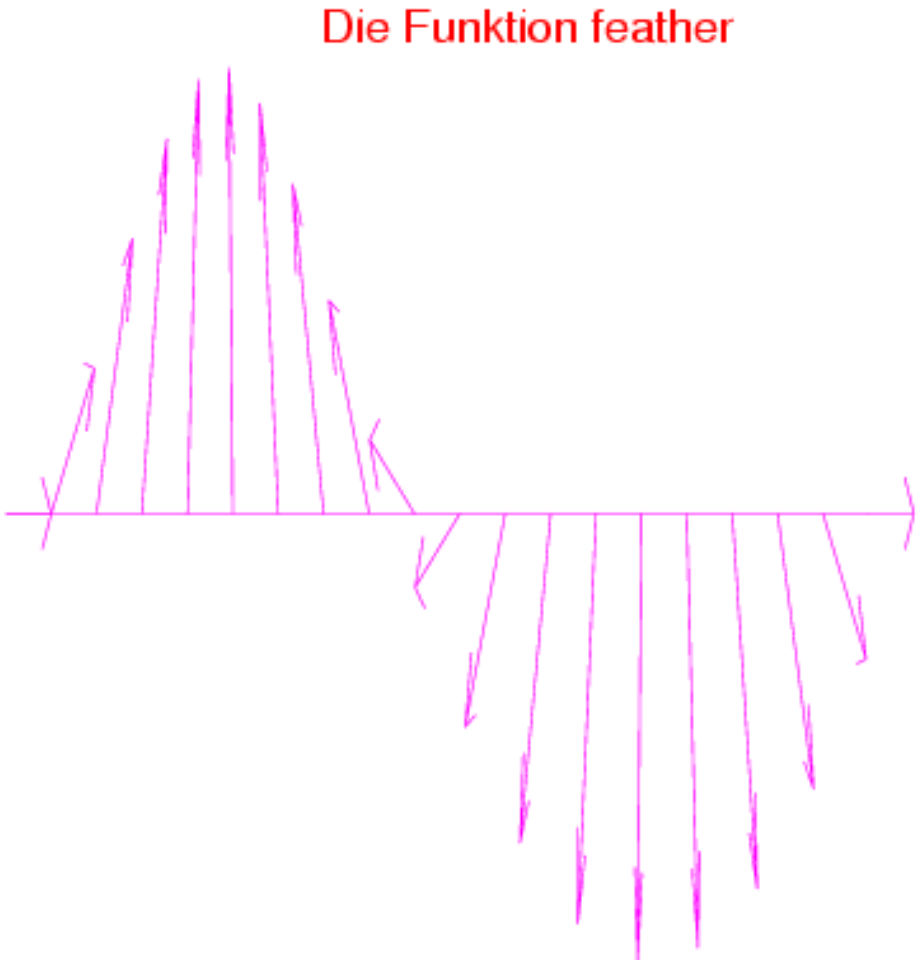
15.3.1.18 Feather

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.

15.3.1.19 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.

15.3.1.20 Pseudocolor

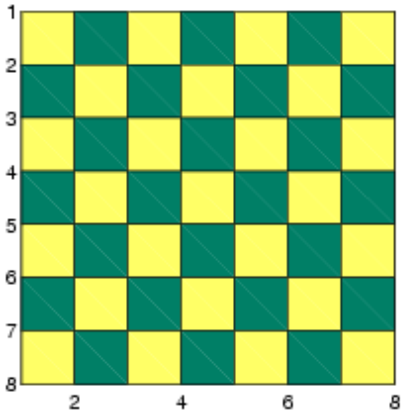
Erstellt einen 'Pseudocolorplot' der Elemente c an den von den Punkten (x,y) definierten Positionen. Wird nur die Farben c angegeben, so werden die Farbe auf einer Matrix der Größe size(c) abgebildet.

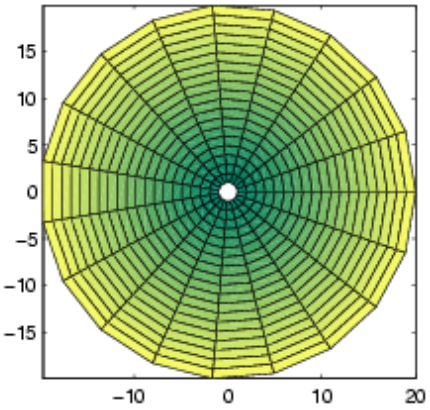
---

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.

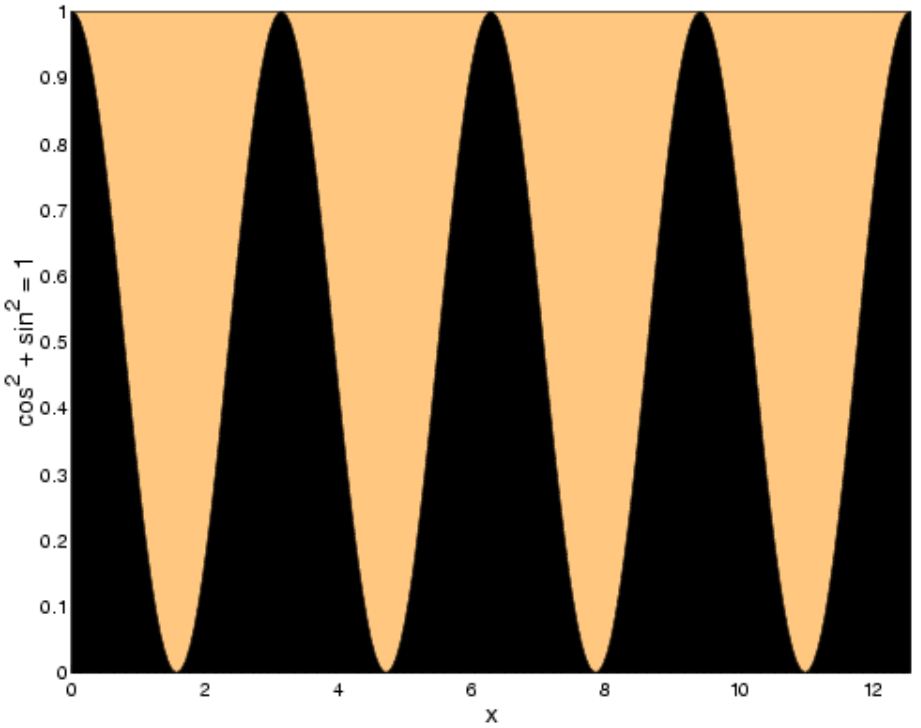
15.3.1.21 Area

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
```



---

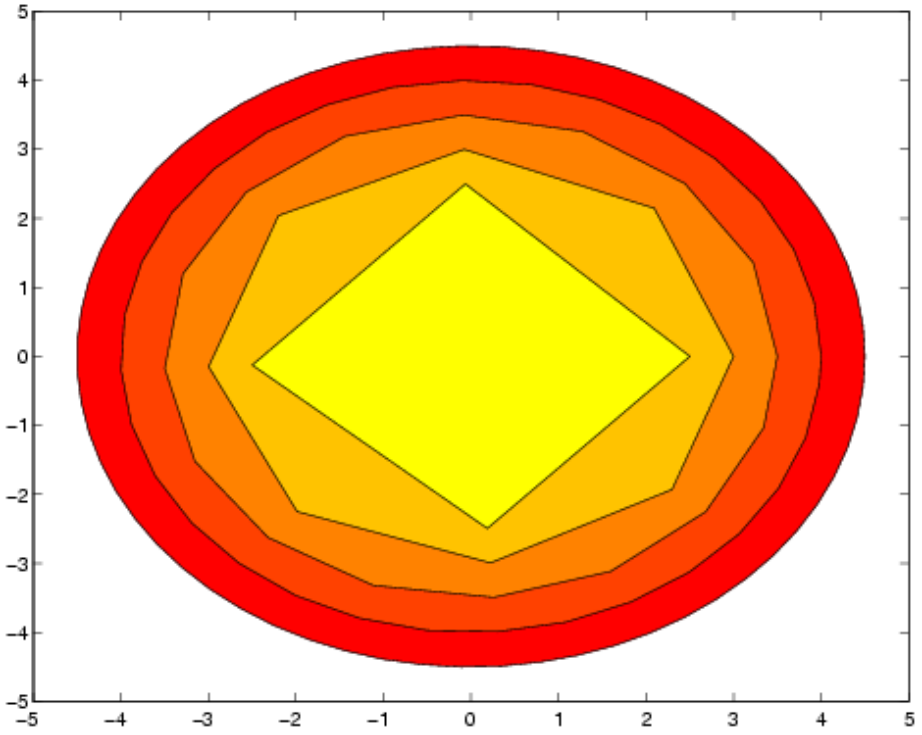
axis tight wählt die Achsengrenzen derart, dass sie nur den Bereich der Graphik abdecken.

15.3.1.22 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
```

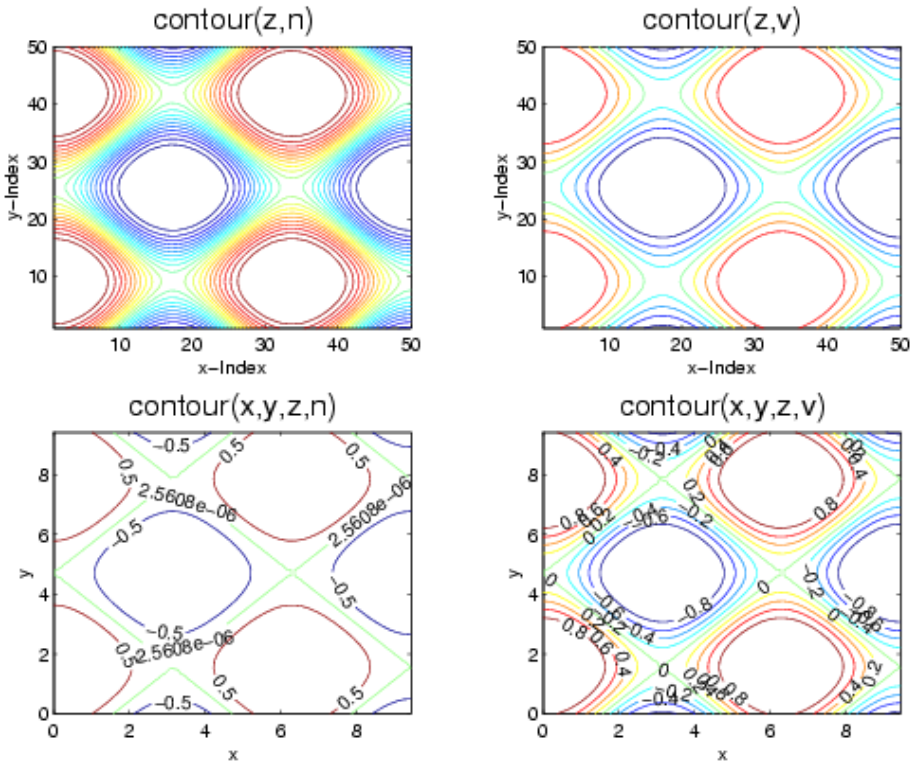
15.3.1.23 Contour

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);
```



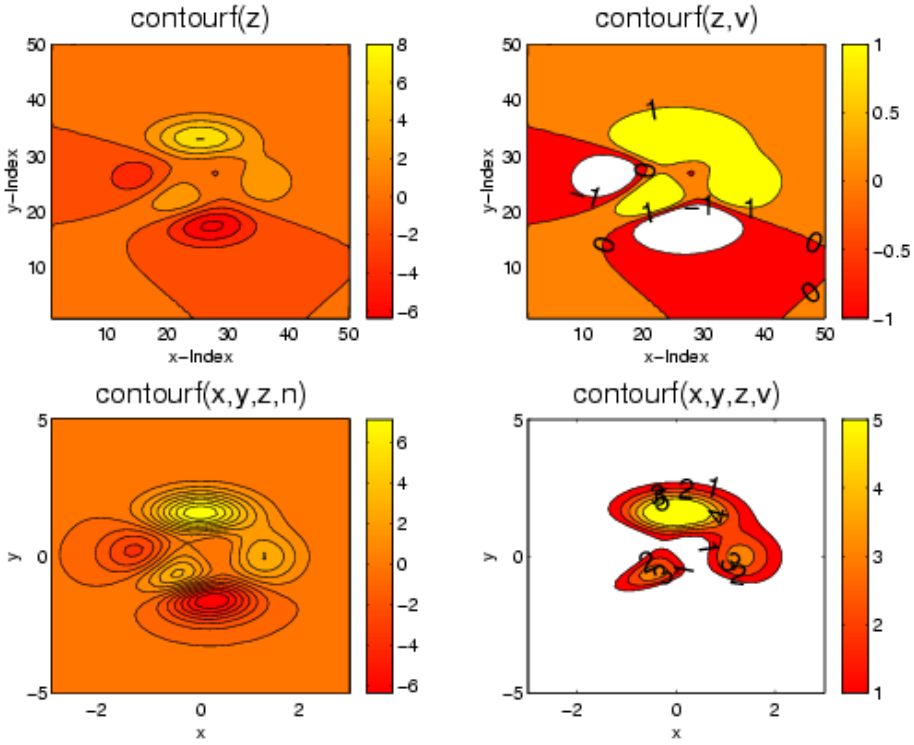
Mit `clabel` werden die Konturlinien mit den entsprechenden z-Werten beschriftet.

15.3.1.24 Contourf

Ähnliche Wirkung wie `contour` in 15.3.1.23, 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.

15.3.1.25 Quiver

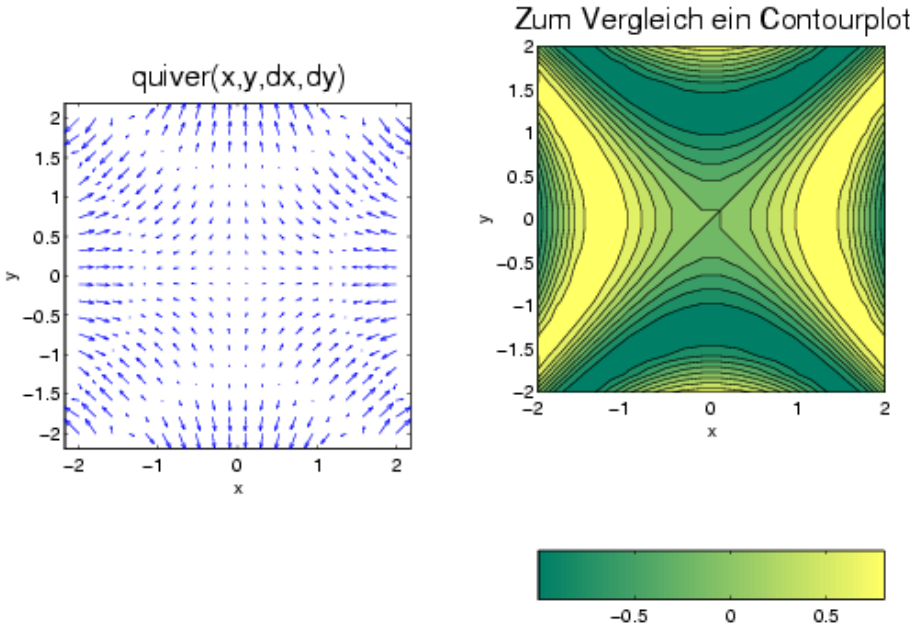
Erstellt von den Punkten (x,y) ausgehende Vektoren mit den Komponenten (u,v).

---

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')
```



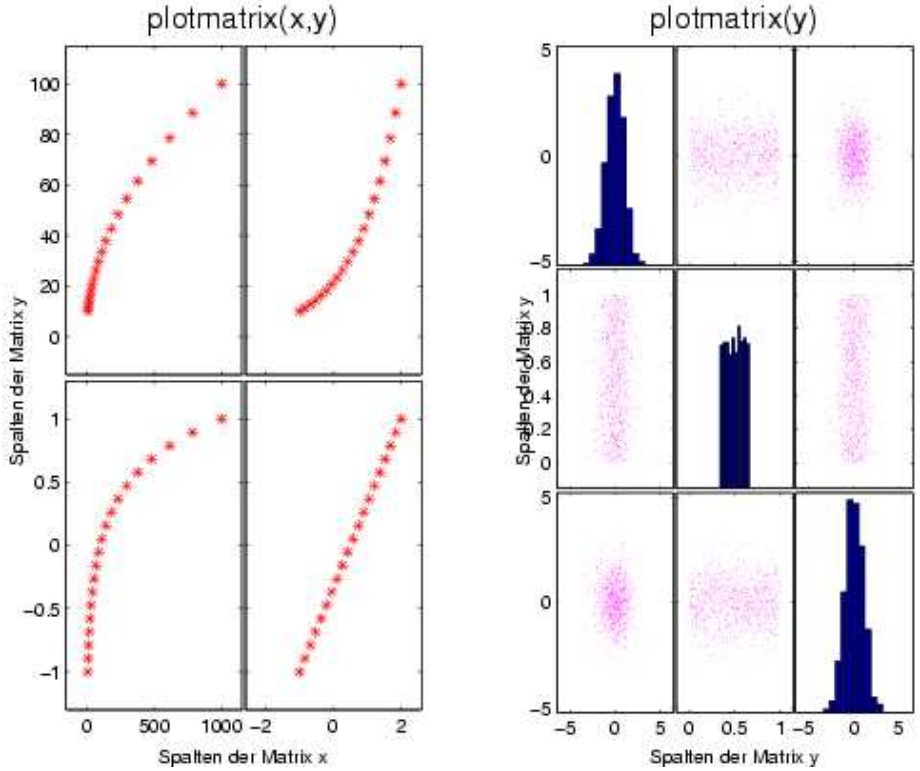
Mit Hilfe von `gradient` erhält man die x- und y- Komponenten des numerischen Gradienten.

15.3.1.26 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.

Tabelle 15.5: MATLAB Befehle zum Erzeugen einfacher dreidimensionaler Graphiken

`plot3(x,y,z)`	15.3.2.1	3D Daten werden durch Angabe von x, y und z dargestellt
`ezplot3(x(t),y(t),z(t))`	15.3.2.2	Erstellt parametrischen 3D Plot durch Angabe der Funktionen als Strings und des Wertebereichs für t
`comet3(x,y,z,p)`	15.3.2.3	Zeichnet 3D Funktion in Form eines animierten 'Kometen'
`fill3(x,y,z,c)`	15.3.2.4	Malt die durch (x,y,z) definierten 3D-Polygone mit der Farbe c aus

15.3.2 Dreidimensionale Plots

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

15.3.2.1 Plot3

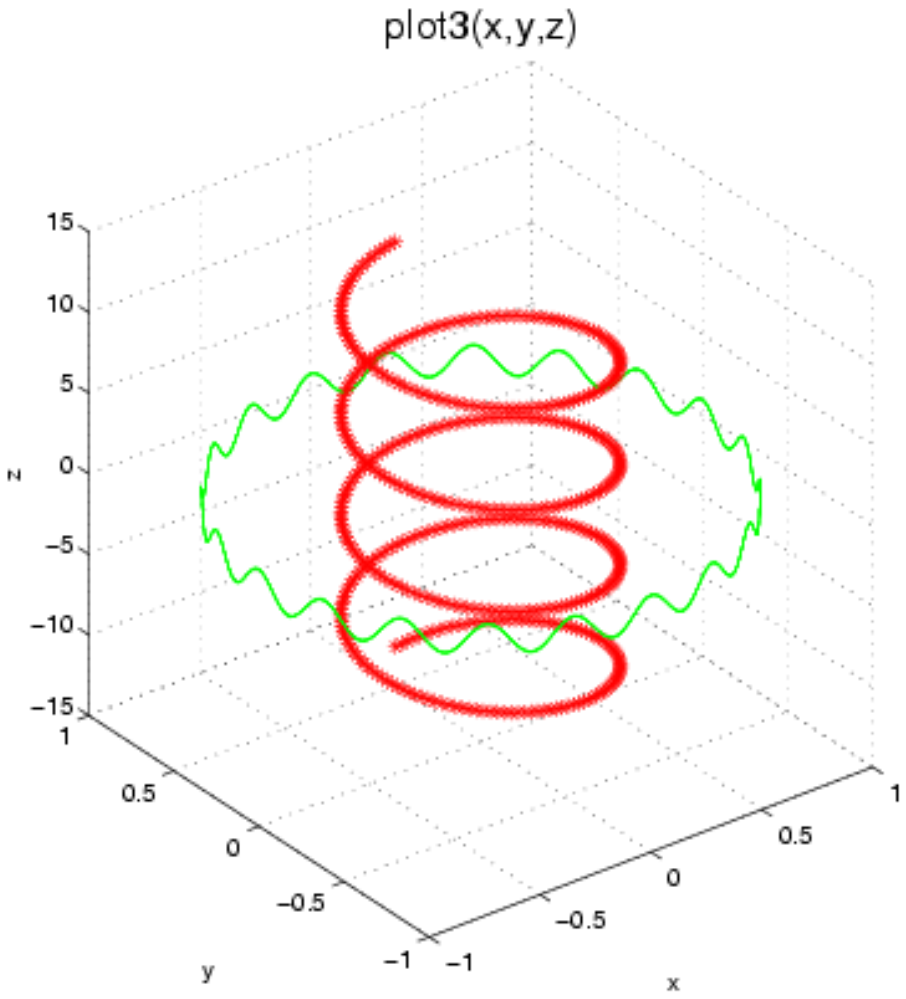
Zeichnet die Daten (x,y,z) in einem 3D-Koordinatensystem ein und verbindet sie gegebenenfalls durch Linien.

---

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
```



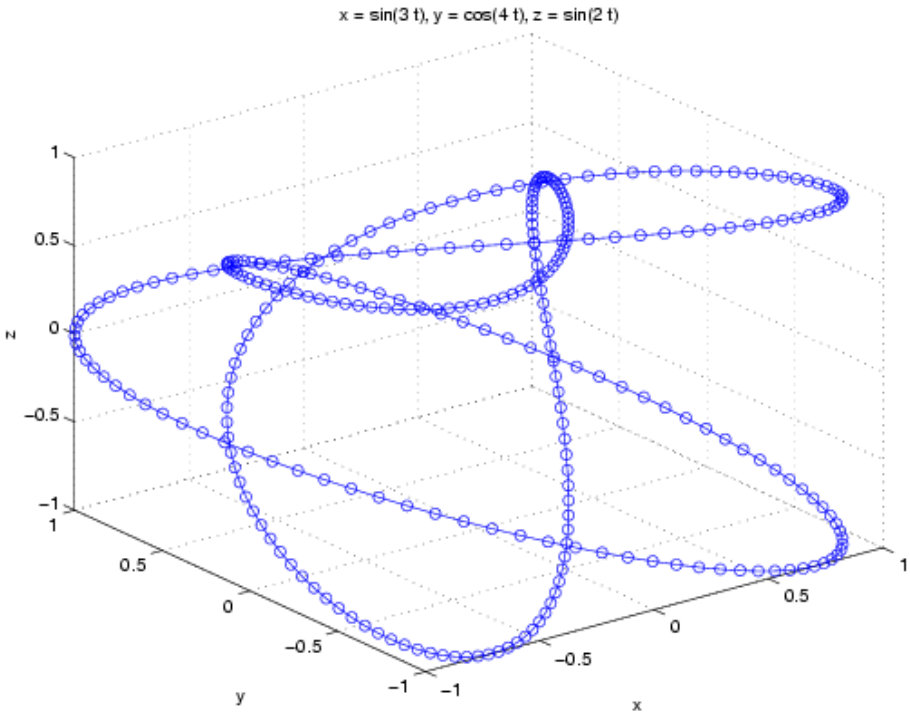
Der Befehl `rotate3d` ermöglicht eine Drehung des Achsensystems mit Hilfe der Maus.

15.3.2.2 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.

15.3.2.3 Comet3

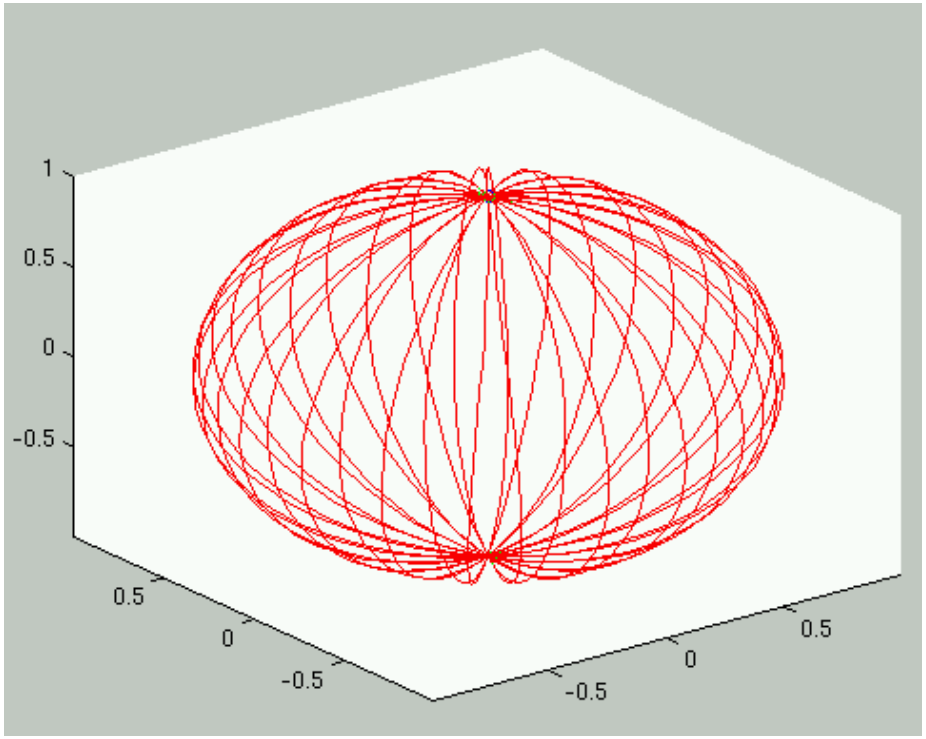
Erstellt eine 3 dimensionale Funktion in Form eines sich bewegenden 'Kometen', dessen Schweif bzw. Spur den Graphen darstellt.

---

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.

15.3.2.4 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)
```

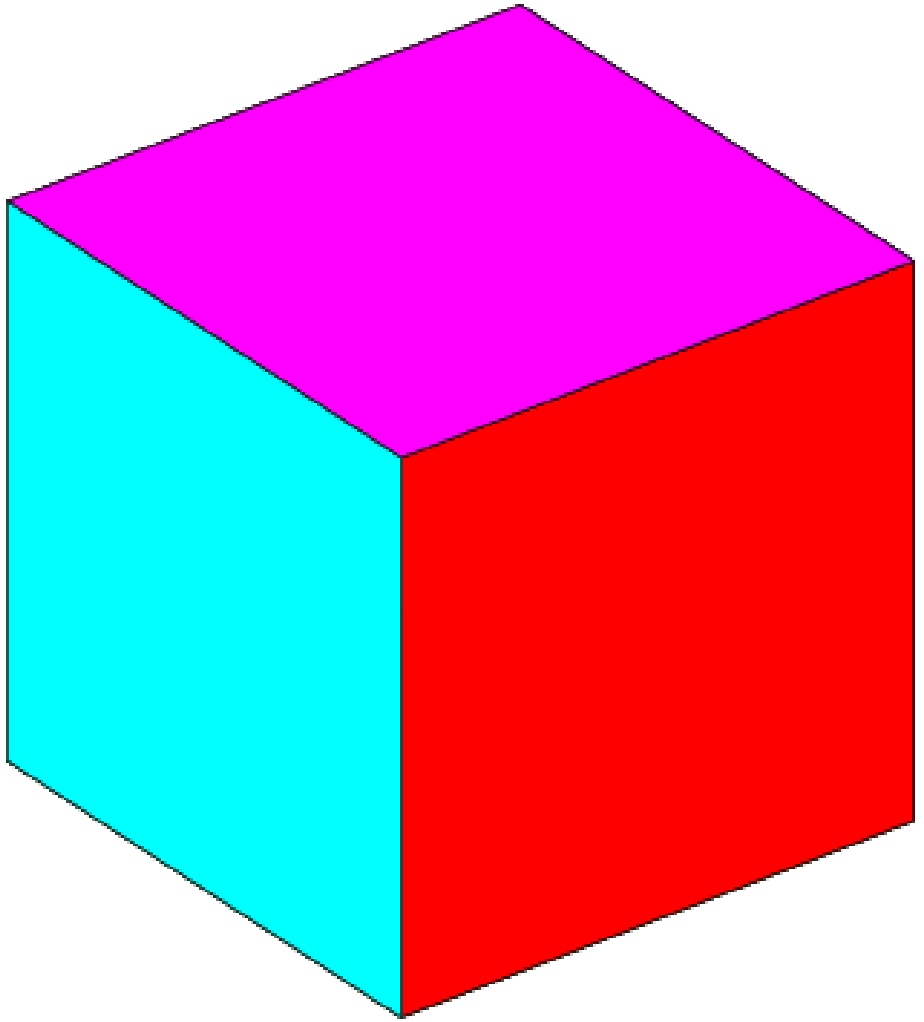


Tabelle 15.6: MATLAB Befehle zum Erzeugen von 3D-Balken- und Kreisdiagrammen

`bar3(x,y,w,'style')`	15.3.2.5	Stellt die 2D Daten als vertikale 3D Balken dar
`bar3h(x,y,w,'style')`	15.3.2.6	Stellt die 2D Daten als horizontale 3D Balken dar
`pie3(x,'explode')`	15.3.2.7	Zeichnet ein 3D Kreisdiagramm von x

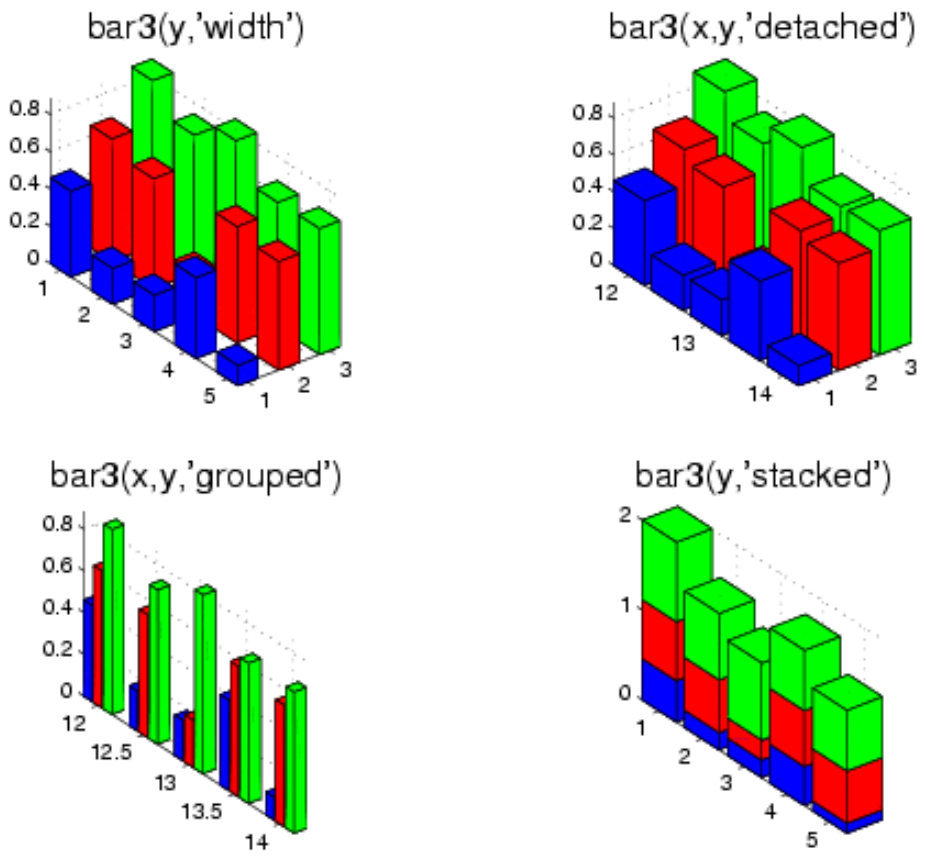
15.3.2.5 Bar3

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')
```



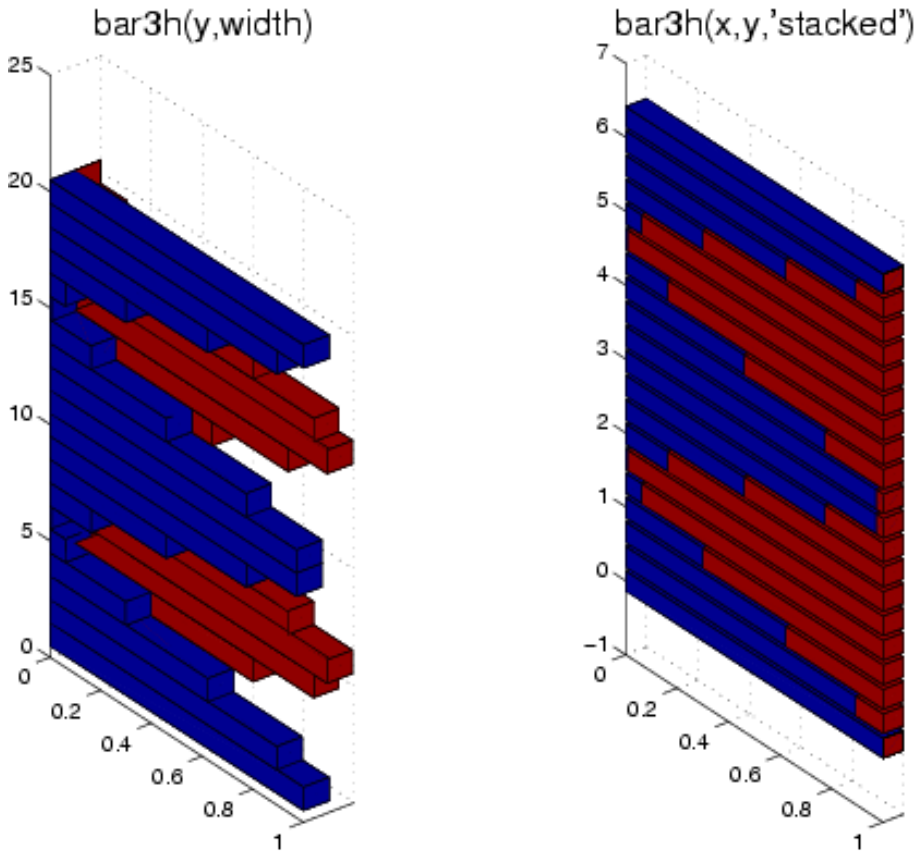
Man beachte die unterschiedliche Darstellung der Säulendiagramme bei der Verwendung der Stile 'detached', 'grouped' und 'stacked'.

15.3.2.6 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.

15.3.2.7 Pie3

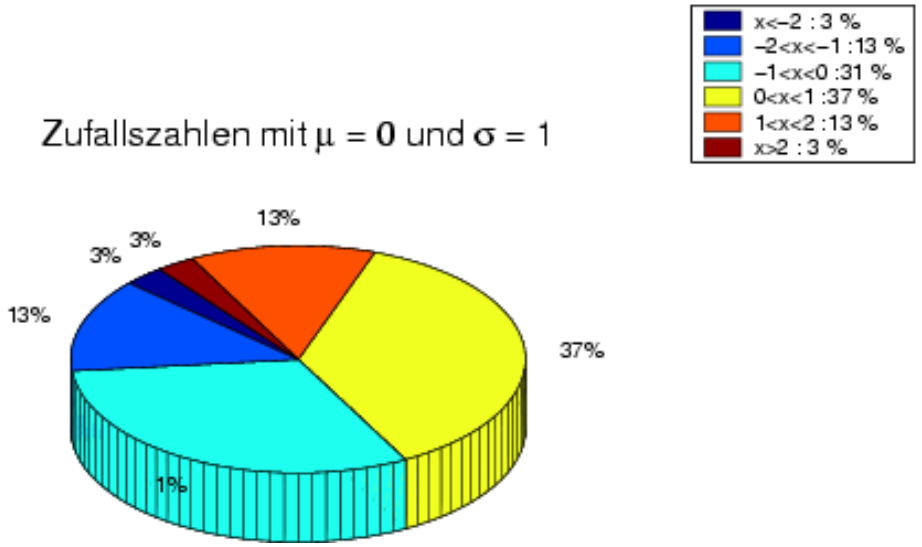
Die Daten des Vektors x werden als 3D-Kreisdiagramme dargestellt, wobei die Segmente optional mit Hilfe des Vektors 'explode' hervorgehoben werden können.

---

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 15.7: MATLAB Befehle zum Erstellen von 3D - Oberflächen

`contour3(x,y,z)`	15.3.2.8	Zeichnet durch z=f(x,y) definierte 3D-Konturlinien
`mesh(x,y,z)`	15.3.2.9	Stellt die Matrix z=f(x,y) in Form eines 'Drahtgitters' dar
`ezmesh('f(x,y)')`	15.3.2.10	'Easy to use' Variante von mesh, f(x,y) wird als String eingegeben
`meshc(x,y,z)`	15.3.2.11	Zeichnet ein 3D-Drahtgitter und einen 2D-Contourplot der Funktion z=f(x,y)
`meshz(x,y,z)`	15.3.2.12	Zeichnet ein 3D-Drahtgitter der Funktion z=f(x,y) mit zusätzlichen seitlichen Referenzlinien
`trimesh(tri,x,y,z)`	15.3.2.13	Zeichnet ein aus Dreiecken bestehendes 3D-Drahtgitter der Funktion z=f(x,y)
`surf(x,y,z)`	15.3.2.14	Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y)
`ezsurf('f(x,y)')`	15.3.2.15	'Easy to use' Variante von surf, f(x,y) wird als String eingegeben
`surfc(x,y,z)`	15.3.2.16	Zeichnet eine 3D-Oberflächengraphik und einen 2D-Contourplot der Funktion z=f(x,y)
`ezsurfc(x,y,z)`	15.3.2.17	'Easy to use' Variante von surfc, f(x,y) wird als String eingegeben
`surfl(x,y,z)`	15.3.2.18	Erstellt eine 3D-Oberflächengraphik der Funktion z=f(x,y) mit wählbarer Beleuchtung
`trisurf(tri,x,y,z)`	15.3.2.19	Zeichnet eine 3D-Oberfläche der Funktion

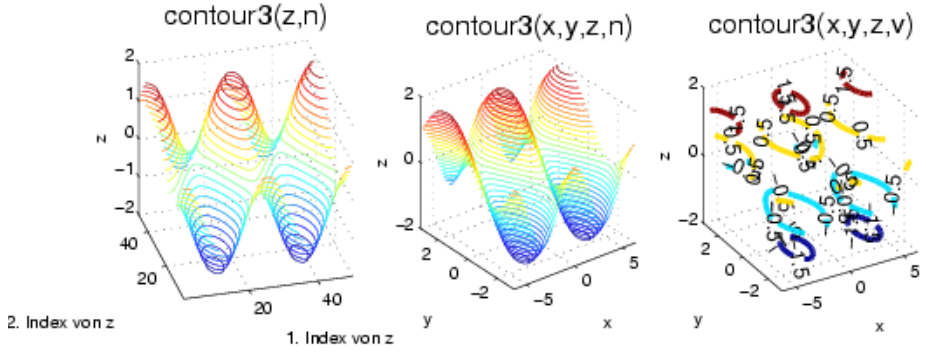
15.3.2.8 Contour3

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);
```



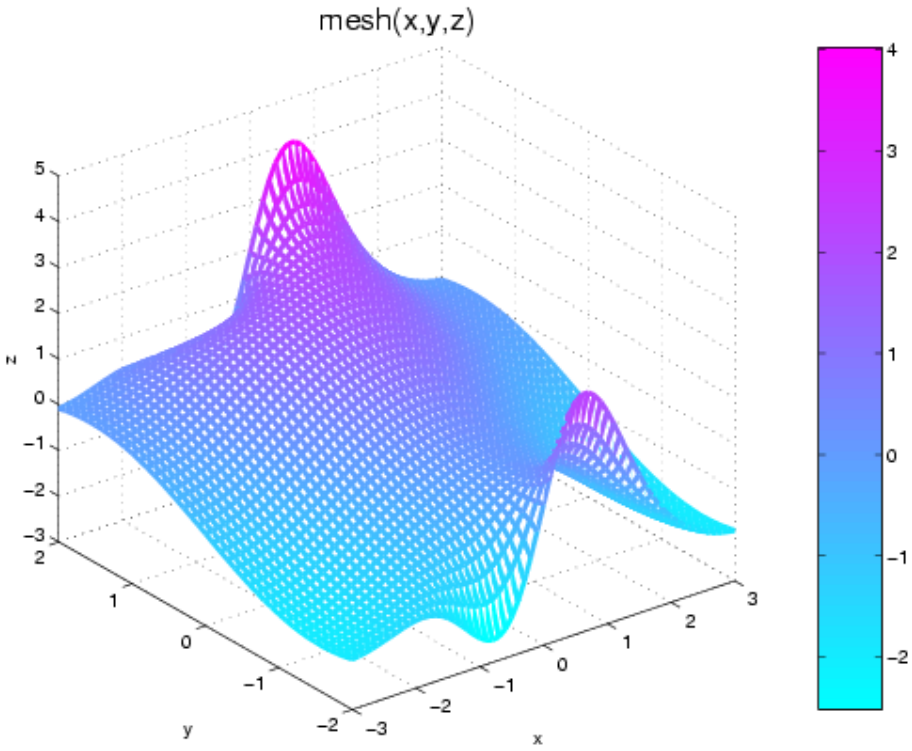
Mit `clabel` werden die Konturlinien mit den entsprechenden z-Werten beschriftet.

15.3.2.9 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.

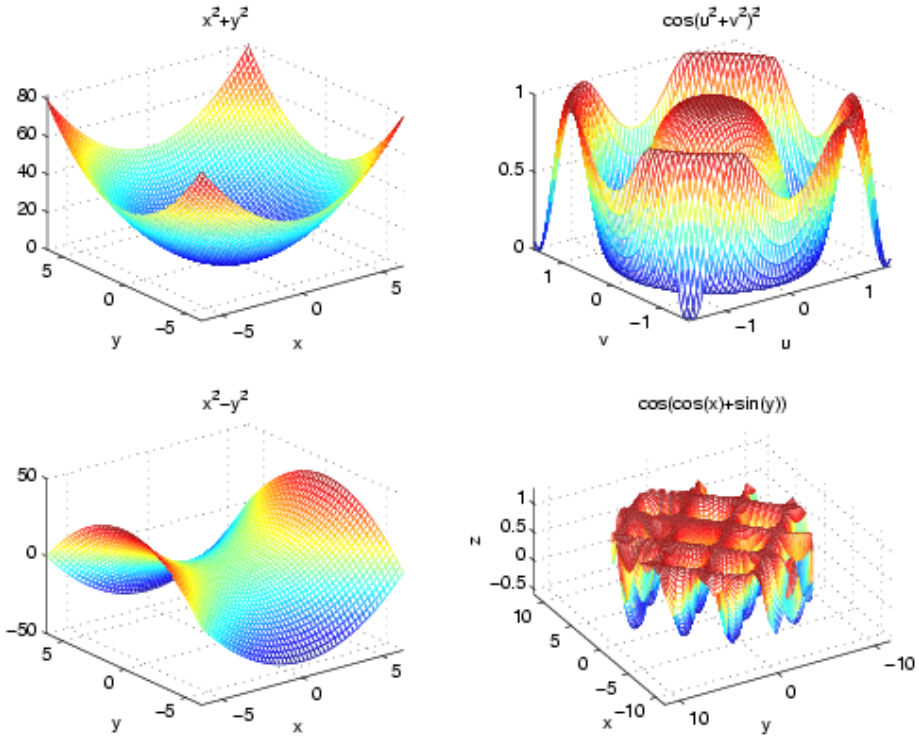
15.3.2.10 Ezmesh

'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')
```



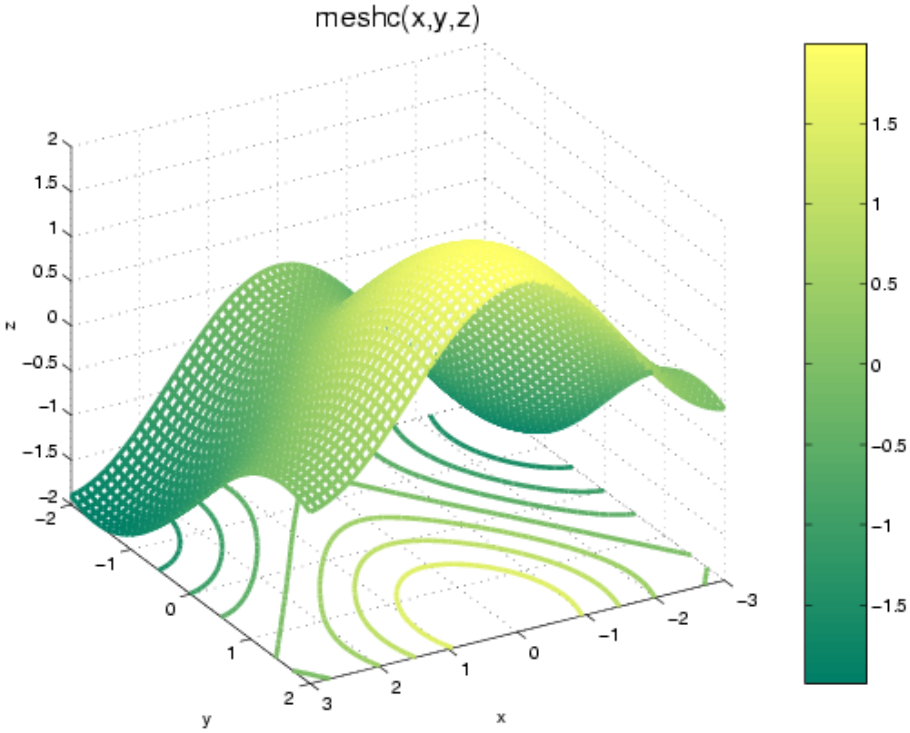
Neben der Funktion f(x,y) können optional die Grenzen von x und y, die Anzahl der Gitterelemente oder der Ausdruck 'circ' (zeichnet Graphik über kreisförmigen Definitionsgebiet) angegeben werden.

15.3.2.11 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);
```



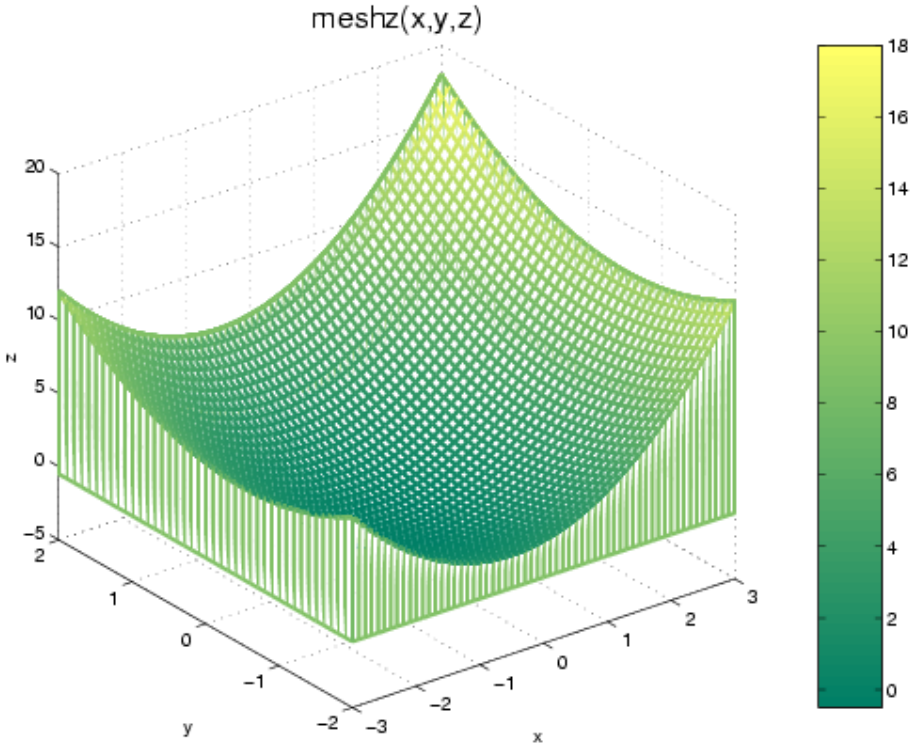
Die Dicke der Konturlinien kann nur gemeinsam mit jenen des Drahtgitters verändert werden.

15.3.2.12 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
```



15.3.2.13 Trimesh

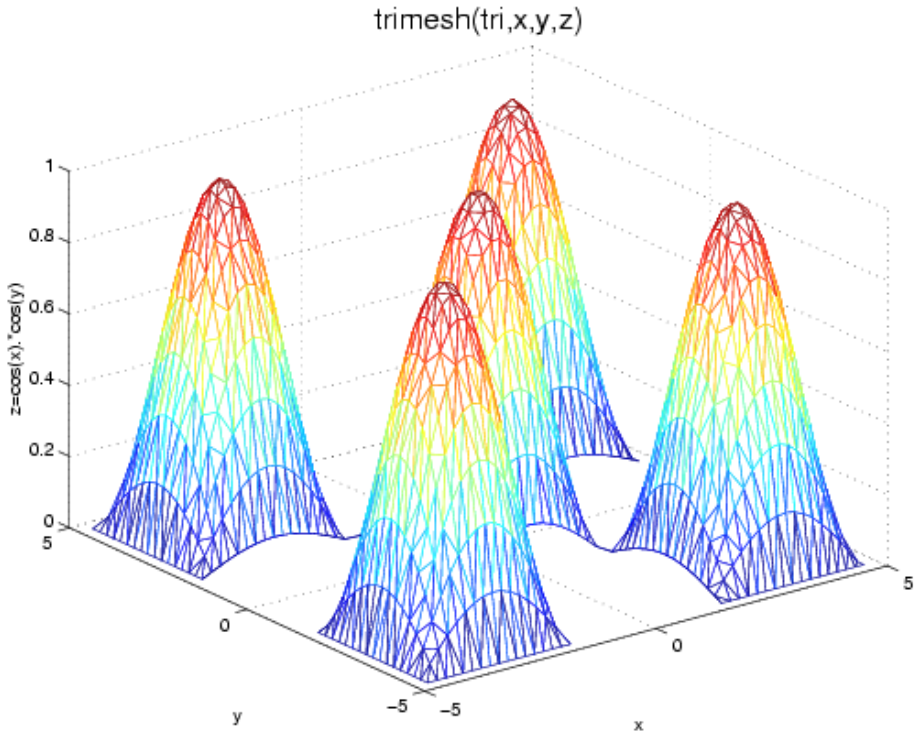
Zeichnet ein aus Dreiecken bestehendes 3D-Drahtgitter der Funktion z=f(x,y).

---

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.

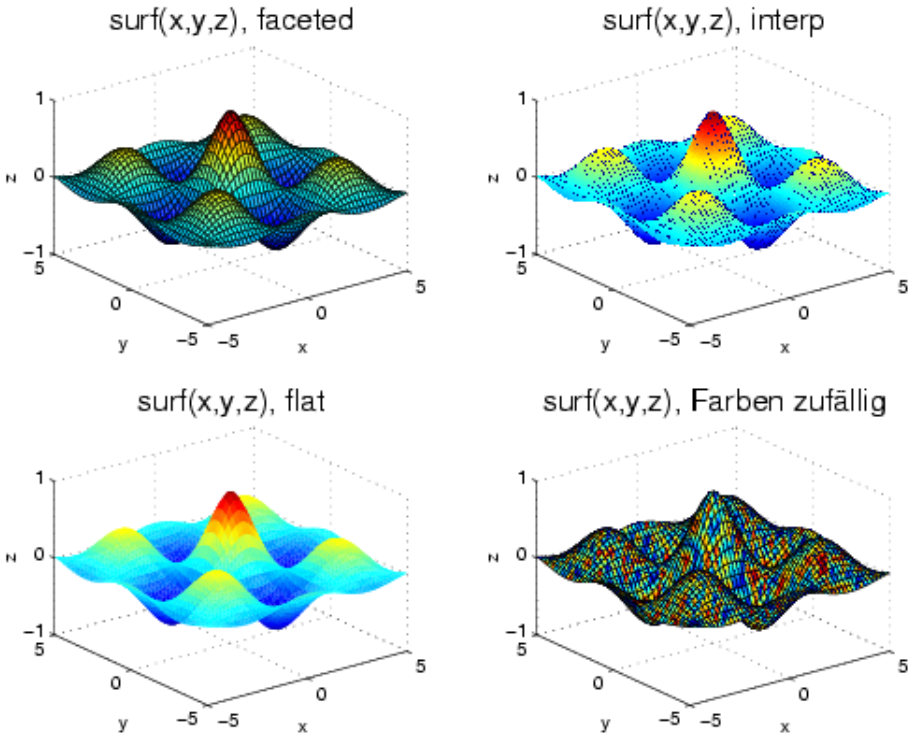
15.3.2.14 Surf

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')
```



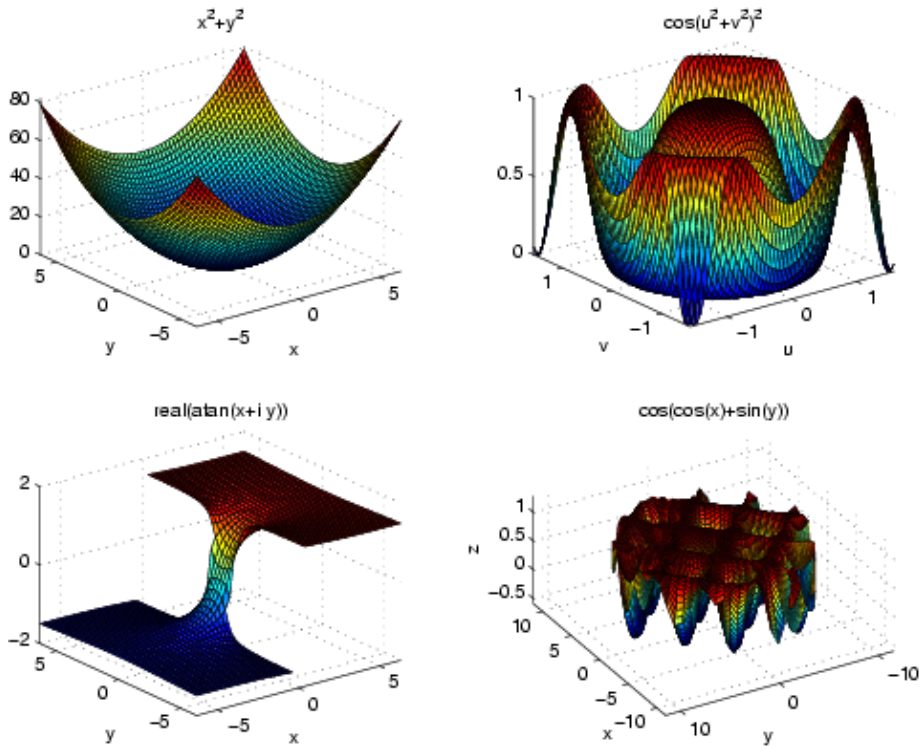
Werden im Aufruf von `surf` die x- und y- Matrizen weggelassen, so werden auf den x- und y-Achsen die beiden Indizes der Matrix z aufgetragen.

15.3.2.15 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 Achsensy-

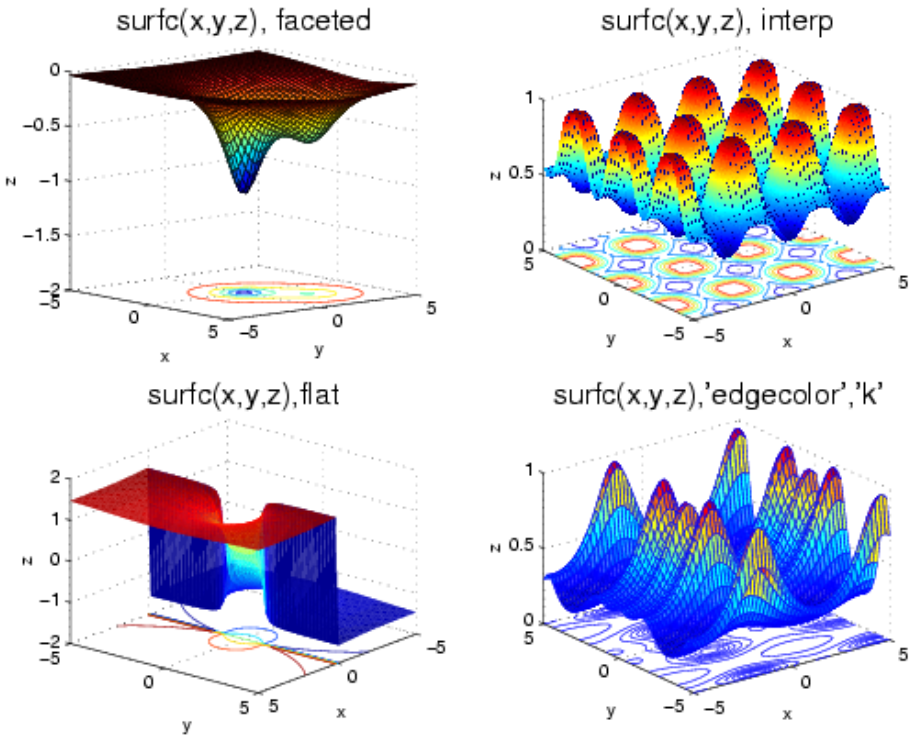
stem ein. Die erste Komponente ist der Azimuthwinkel in Grad (Rotation der x,y Ebene), die zweite Komponente ist der Kippwinkel aus der horizontalen Lage der x,y Ebene.

15.3.2.16 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')
```

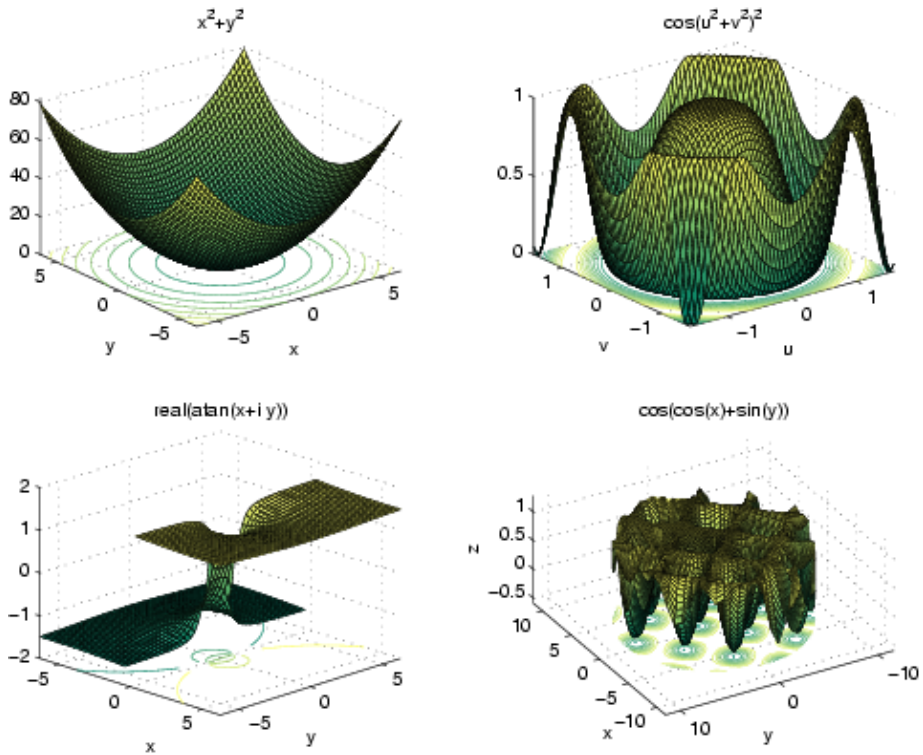


15.3.2.17 Ezsurfc

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')
```

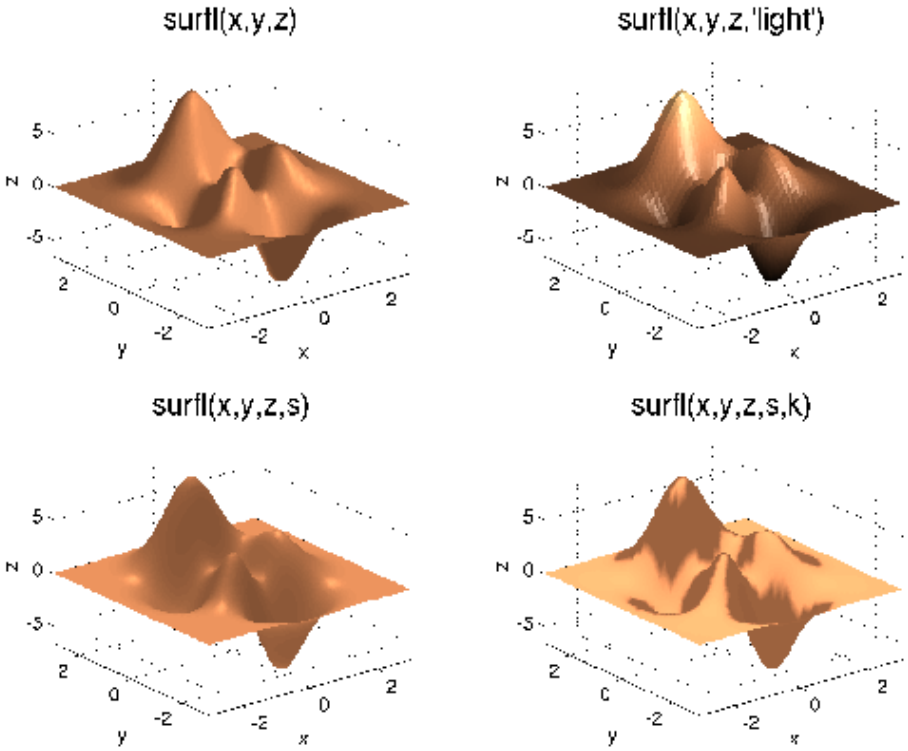


15.3.2.18 Surfl

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)
```



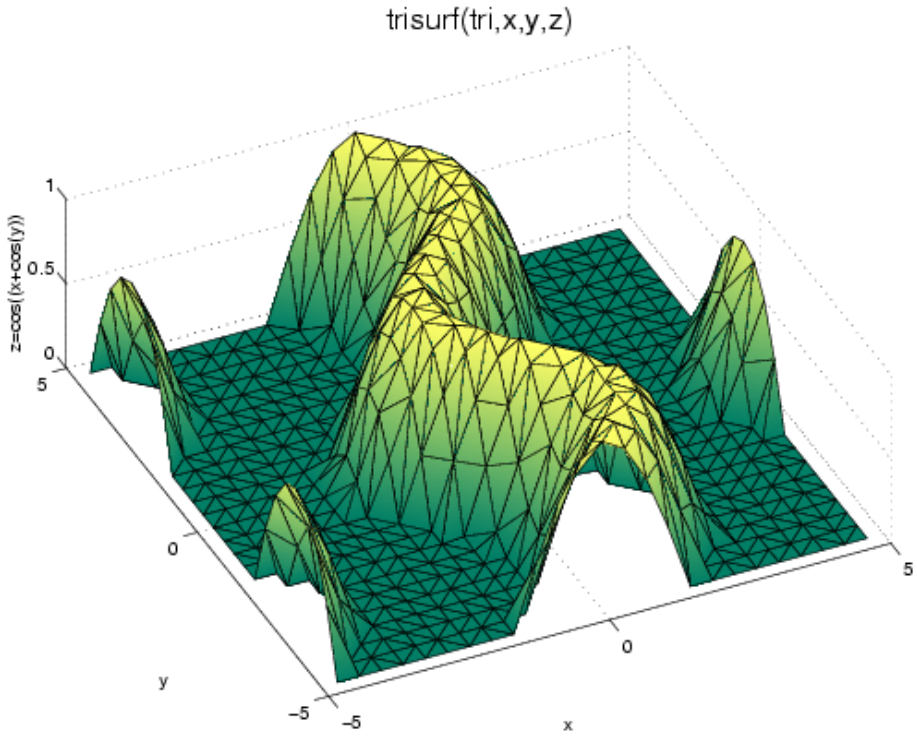
Der Vektor s beinhaltet die x-, y- und z- Komponenten der Einfallsrichtung des Lichts und k die relativen Intensitäten des Umgebungslichtes, der diffusen Reflexion, der spiegelnden Reflexion sowie des spiegelnden Glanzes.

15.3.2.19 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')
```

15.3.2.20 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','k')
set(gcf,'color','k');
```

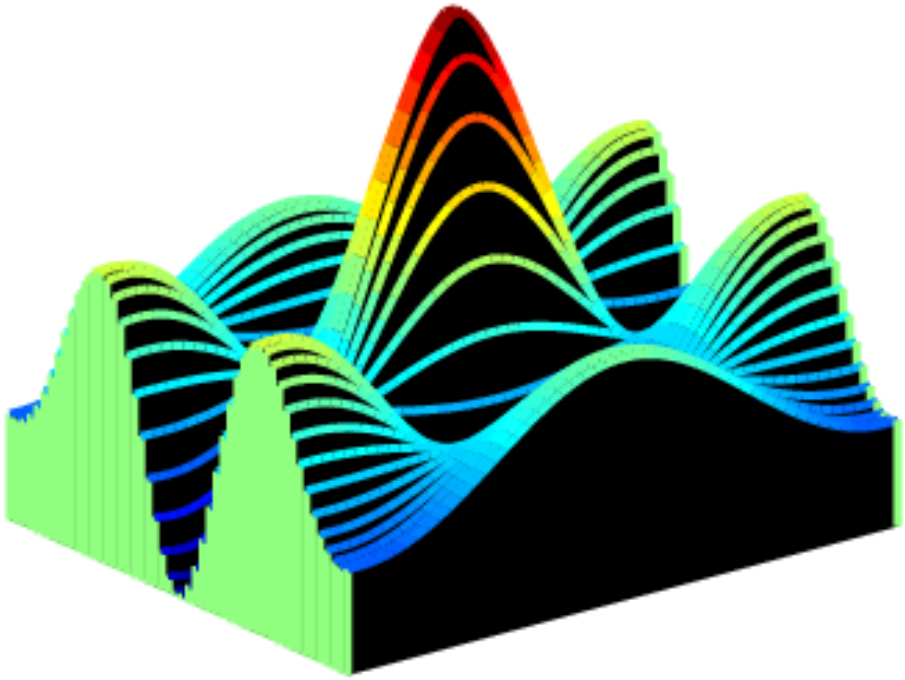


Tabelle 15.8: MATLAB Befehle zum Erstellen von 3D - volumetrischen Graphiken

`quiver3(x,y,z,u,v,w)`	15.3.2.21	Zeichnet an den Punkten (x,y,z) Vektorpfeile mit den Komponenten (u,v,w)
`slice(x,y,z,d,sx,sy,sz)`	15.3.2.22	Veranschaulicht die volumetrische Funktion d=f(x,y,z) durch senkrecht durch die Achsen gelegte Schnittflächen

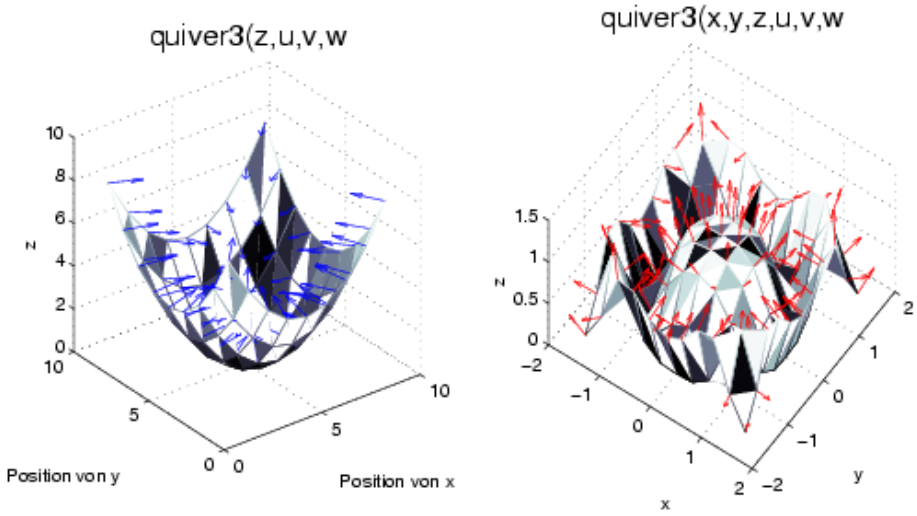
15.3.2.21 Quiver3

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.

15.3.2.22 Slice

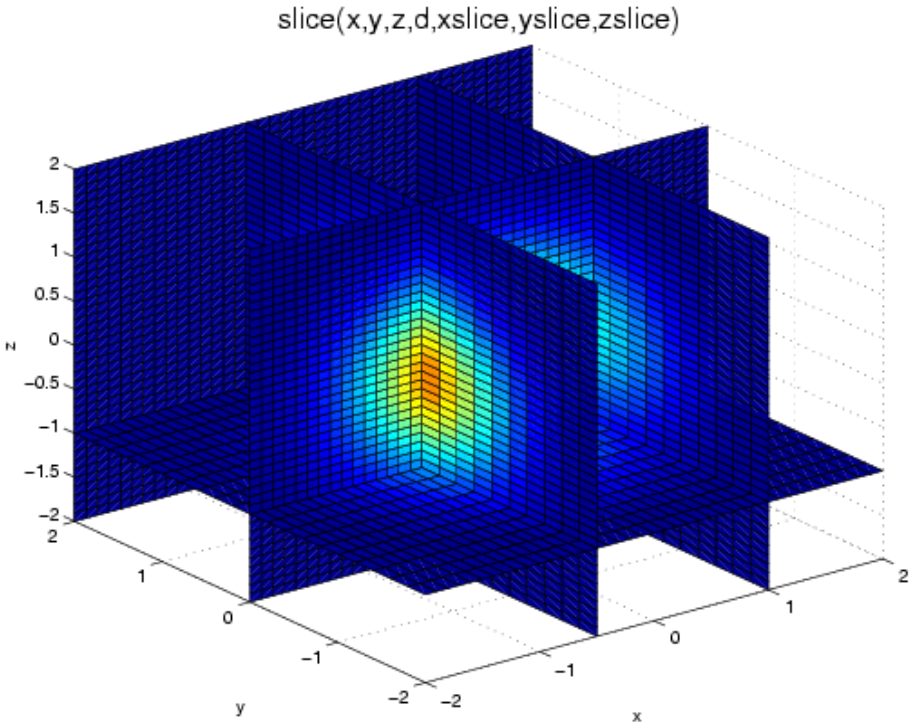
Veranschaulicht die volumetrische Funktion d=f(x,y,z) durch senkrecht durch die Achsen gelegte Schnittflächen. Dabei wird die x-Achse an den Stellen des Vektors xslice geschnitten, analog für die beiden anderen Achsen.

---

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,zslice)
```



Mit meshgrid lassen sich auch die x-, y- und z- Koordinaten dreidimensionaler Gitter berechnen.

Tabelle 15.9: Weitere spezielle 3D Graphik-Befehle

`stem3(x,y,z)`	15.3.2.23	Zeichnet 3D Funktion und verbindet Datenpunkte mit der Ebene z=0
`sphere(n)`	15.3.2.24	Erstellt eine durch n^2 Flächen angenäherte Kugel
`cylinder(r,n)`	15.3.2.25	Erstellt einen durch ein n-seitiges Prisma angenäherten Zylinder mit Radius r
`scatter3(x,y,z,r,c)`	15.3.2.26	Zeichnet Daten an den Positionen (x,y,z) der Größe r sowie der Farbe c
`ribbon(y,z,w)`	15.3.2.27	Zeichnet die Spalten von z über jenen von y als 3D Bänder der Breite w

15.3.2.23 Stem3

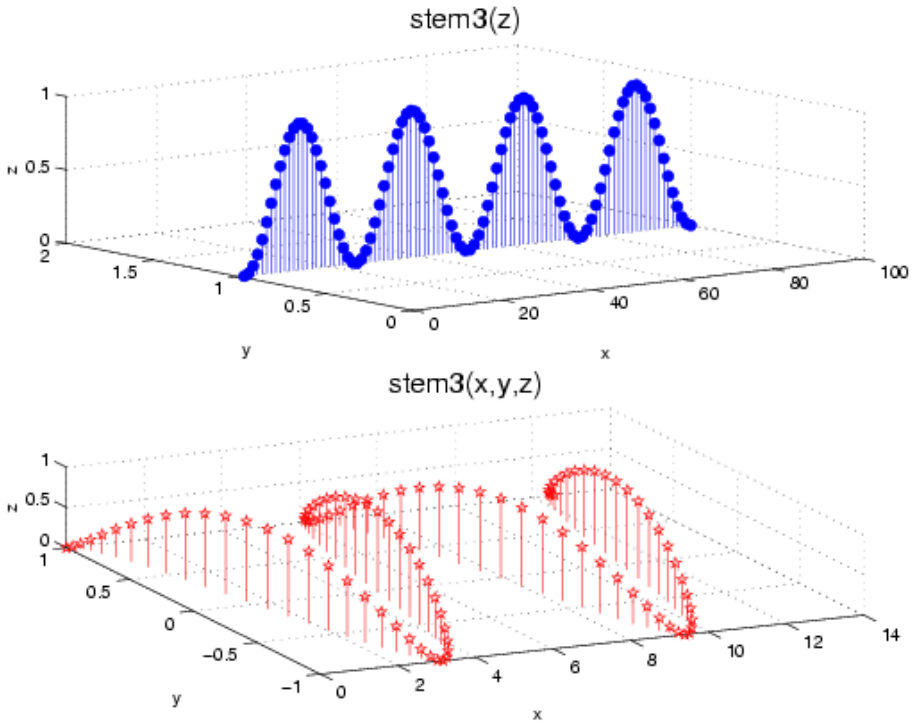
Zeichnet dreidimensionale Daten und verbindet Datenpunkte mit der Ebene z=0.

---

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.

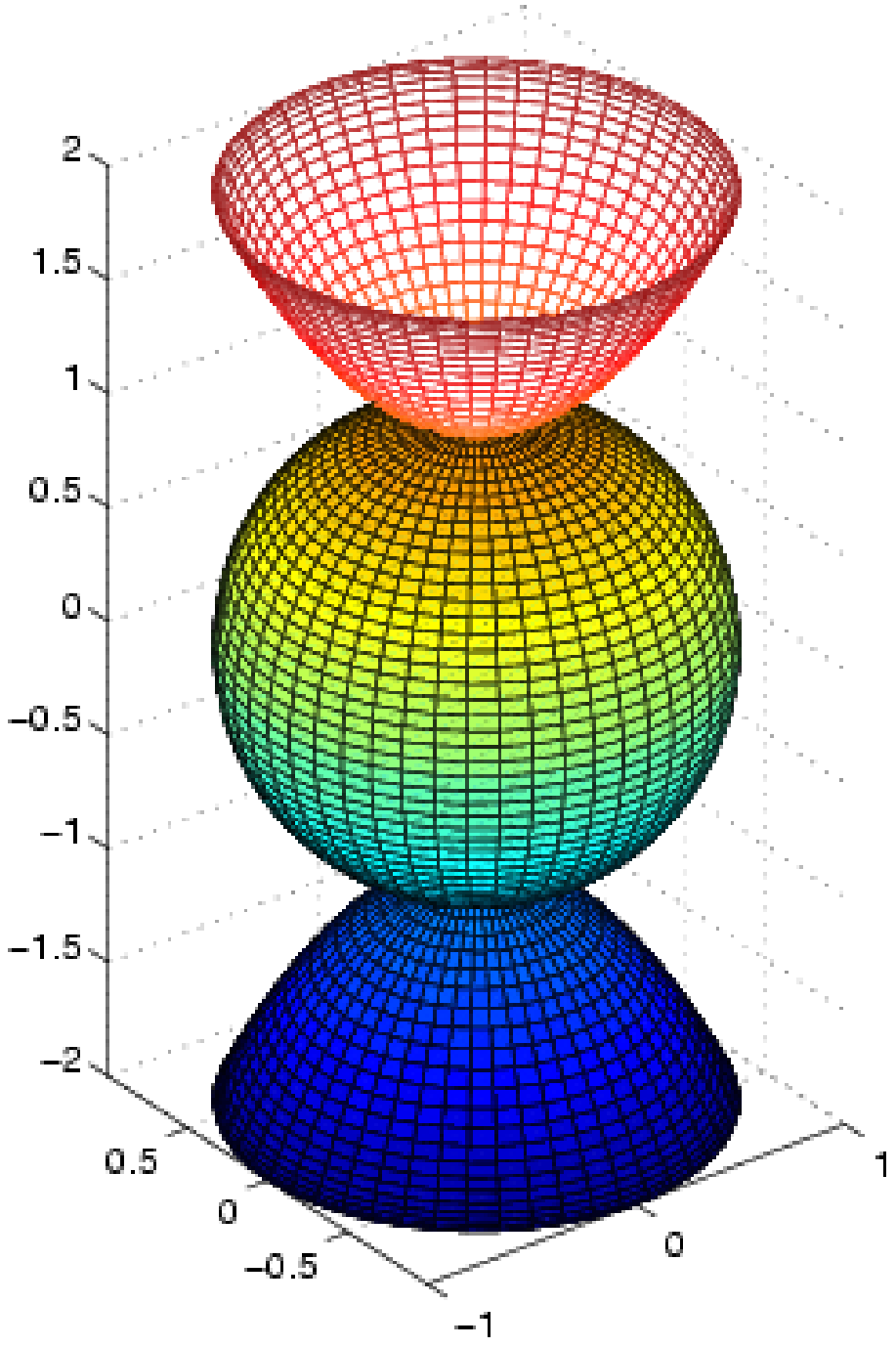
15.3.2.24 Kugel

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.

15.3.2.25 Zylinder

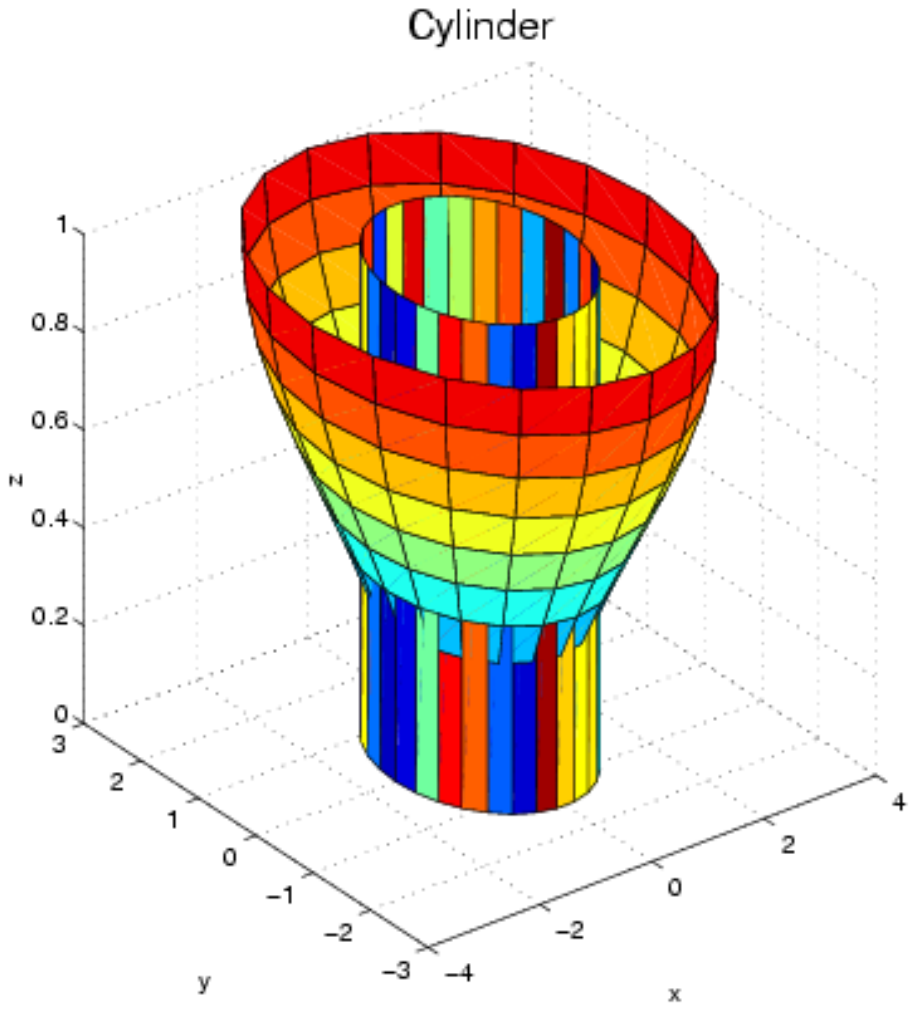
Erstellt Zylinder (bzw. n-seitige Prismen) und allgemeine um die z-Achse symmetrische Körper der Höhe 1 mit der Profilkurve r(h).

---

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

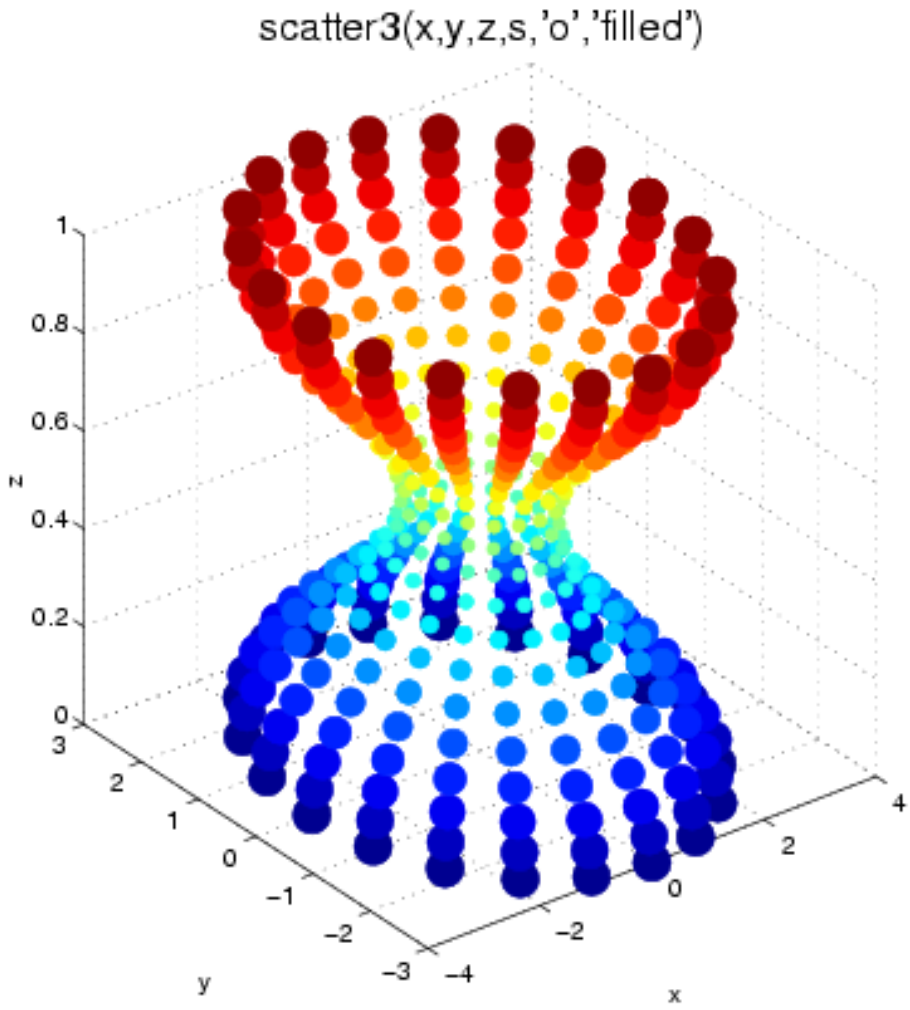
15.3.2.26 Scatter3

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')
```

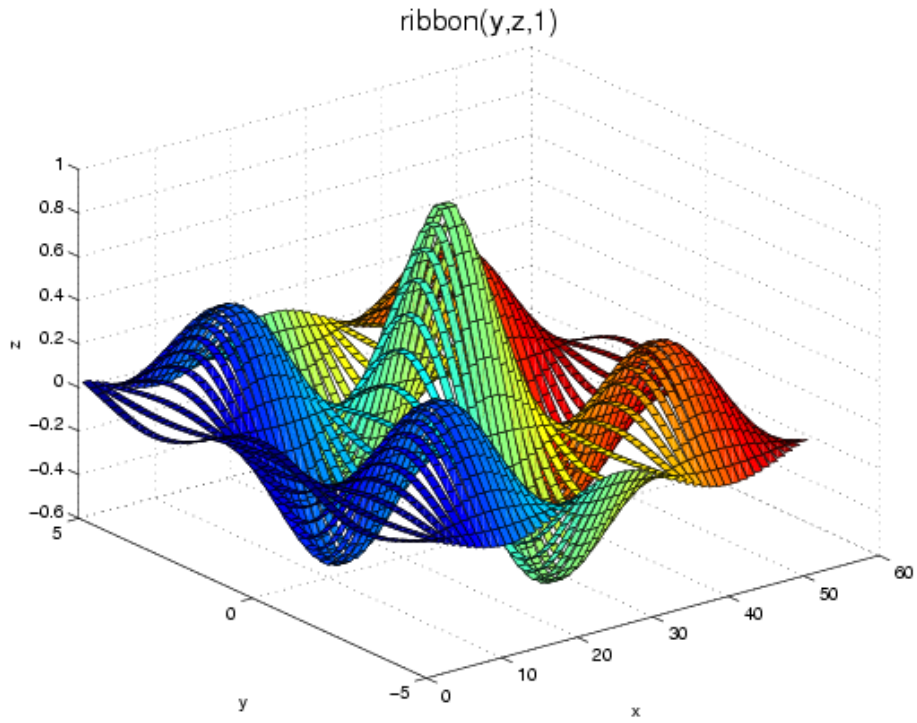


15.3.2.27 Ribbon

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 16

Übungsbeispiele

Übungen werden nur mehr über das Programm MatlabTutor angeboten. Ein Abbild des Datenbankinhaltes findet man unter folgender Webseite für Übungen.

Kapitel 17

Nachlese - Was soll ich können?

17.1 Basis Syntax in MATLAB

17.1.1 Fragen

1. Was unterscheidet ein MATLAB-Skript und eine MATLAB-Funktion?
2. Welche erste Zeile muss eine MATLAB-Funktion enthalten?
3. Wie starte ich den Editor und die Online-Hilfe?
4. Wie bekomme ich direkt im MATLAB-Command-Fenster Hilfe zum Befehl `input`?
5. Wie funktioniert unter Linux ”copy and paste”?

6. Unter der Voraussetzung, dass ich ein Programm im File `test.m` gespeichert habe, wie kann ich es dann in MATLAB ausführen?

7. Was ist bei den folgenden Befehlen **falsch**? Voraussetzung ist, dass die skalaren Variablen `x`, `a`, `b`, `c` , `d` bereits definiert sind.

```
y    = 3x + a
y    = 5 + sin x
y    = a exp(-(b^2 - c^2)*x^2
y(x) = a*sin(x)
y    = b * sin[x] + c * cos[a*x]
y    = a * sqrt( {b^2 + c^2}*x )
y    = b * arcsin(a*x)
y    = a * (x + b * [x + c * {x + d}])
```

8. Was bewirkt der Unterschied in den folgenden Zeilen?

```
y = x^2
y = x^2,
y = x^2;
y = x^2 % Quadrat
```

9. Was ist in der Programmzeile für folgende mathematische Funktion **falsch**?

$$y(x) = \frac{x^2}{x + a}$$

```
y = x^2 / x + a
```

10. Mit welchem Befehl kann man den Benutzer eines Programms auffordern einen Wert einzugeben? Z.B.: "Geben Sie a ein: ". Der eingegebene Wert soll dabei der Variablen `a` zugewiesen werden.

11. Mit welchem Befehl kann ich eine Zahl oder eine Zeichenkette am Schirm ausgeben?

12. Wie kann ich mehrere Zeichenketten (`s1, s2, s3`) aneinanderfügen?

13. Wie kann ich eine Zahl (Datentyp: double) in eine Zeichenkette gleichen Inhalts (Datentyp: char) umwandeln?

14. Was ist **falsch** an folgenden Zeilen, wenn `s1, s2` Zeichenketten und `x` eine Zahl ist?

```
disp(s1,s2)
disp([s1,s2)]
disp([s1,x,s2])
disp([s1;s2])
```

15. Was ist der Unterschied zwischen den beiden Zeilen?

```
y = [1,2,3]
y = [1;2;3]
```

16. Wie kann ich in einem MATLAB-File Kommentare einfügen, die bei Verwendung des Befehls `help` sichtbar sind?

17.1.2 Antworten

1. Was unterscheidet ein MATLAB-Skript und eine MATLAB-Funktion?

   Ein MATLAB-Skript ist eine Aneinanderreihung von Befehlen (Hauptprogramm). Eine MATLAB-Funktion wird mit Ein- und Ausgabeparametern gestartet und braucht eine Deklarationszeile (siehe 2).

2. Welche erste Zeile muss eine MATLAB-Funktion enthalten?

   `function out = func1(in1,in2,in3)` oder
   `function [out1,out2] = func1(in1,in2,in3)`

3. Wie starte ich den Editor und die Online-Hilfe?

   Mit den Befehlen `edit` und `helpbrowser`.

4. Wie bekomme ich direkt im MATLAB-Command-Fenster Hilfe zum Befehl `input`?

   Mit dem Befehl `help input`. Der Befehl `lookfor input` listet alle Befehle in deren help-Text `input` vorkommt.

5. Wie funktioniert unter Linux "copy and paste"?

   Einfärben mit der linken Maustaste (copy) und einfügen mit der mittleren Maustaste (paste).

6. Unter der Voraussetzung, dass ich ein Programm im File `test.m` gespeichert habe, wie kann ich es dann in MATLAB ausführen?

Durch Eingabe des Befehls `test`.

7. Was ist bei den folgenden Befehlen **falsch**? Voraussetzung ist, dass die skalaren Variablen `x`, `a`, `b`, `c` , `c` bereits definiert sind.

```
y = 3*x + a
y = 5 + sin(x)
y = a*exp(-(b^2 - c^2)*x^2)
y = a*sin(x)      Argument x in y(x) entfernt
y = b * sin(x) + c * cos(a*x)
y = a * sqrt( (b^2 + c^2)*x )
y = b * asin(a*x)
y = a * (x + b * (x + c * (x + d)))
```

8. Was bewirkt der Unterschied in den folgenden Zeilen?

```
y = x^2       Ausgabe am Schirm
y = x^2,      Ausgabe am Schirm
y = x^2;      Keine Ausgabe am Schirm
y = x^2 % Quadrat      Ausgabe am Schirm ohne Kommentar
```

9. Was ist in der Programmzeile für folgende mathematische Funktion **falsch**?

$$y(x) = \frac{x^2}{x + a}$$

```
y = x^2 / (x + a)
```

10. Mit welchem Befehl kann man den Benutzer eines Programms auffordern einen Wert einzugeben? Z.B.: "Geben Sie a ein: ". Der eingegebene Wert soll dabei der Variablen `a` zugewiesen werden.

    ```
    a = input('Geben Sie a ein: ');
    ```

11. Mit welchem Befehl kann ich eine Zahl oder eine Zeichenkette am Schirm ausgeben?

    ```
    disp
    ```

12. Wie kann ich mehrere Zeichenketten (`s1, s2, s3`) aneinanderfügen?

    ```
    s = [s1,s2,s3]
    ```

13. Wie kann ich eine Zahl (Datentyp: double) in eine Zeichenkette gleichen Inhalts (Datentyp: char) umwandeln?

    ```
    s=num2str(d)
    ```
    `s=num2str(d,n)` mit `n` Anzahl der Digits

14. Was ist **falsch** an folgenden Zeilen, wenn `s1, s2` Zeichenketten und `x` eine Zahl ist?

    ```
    disp([s1,s2])
    disp([s1,s2])
    disp([s1,num2str(x),s2])
    disp([s1,s2])
    ```

15. Was ist der Unterschied zwischen den beiden Zeilen?

    ```
    y = [1,2,3]
    y = [1;2;3]
    ```

    Zeilenvektor
    Spaltenvektor

16. Wie kann ich in einem MATLAB-File Kommentare einfügen, die bei Verwendung des Befehls `help` sichtbar sind?

    Durch Einfügen von zusammenhängenden Kommentarzeilen am Anfang des Files (MATLAB-Skript) bzw. nach der Deklarationszeile (MATLAB-Funktion). Die erste Leer- oder Kommandozeile beendet diesen Block. Weiter Kommentare werden bei Verwendung von `help` nicht angezeigt.

17.2 Reguläre Polyeder, Kegelschnitte

17.2.1 Fragen

1. Was sind richtige und falsche Namen von Variablen?
   `a12 1a a-3 a_12 a(3) _bb maxi a.b`

2. Wie kann man feststellen, welche Variablen im MATLAB-Arbeitsbereich bereits definiert sind?

3. Wie kann man feststellen, ob ein Name bereits als Variable oder Funktion existiert?

4. Warum sollte man `i`, `j` oder z.B. `max` nicht als Variablennamen verwenden?
5. Wie erzeugt man einen Vektor mit 20 Zahlen, die equidistant zwischen `0` und `2` verteilt sind.
6. Gegeben ist eine Zeichenkette `st='Sinus'`. Welche Ausgabe erzeugen die Befehle `lower(st)`, `upper(st)`, bzw. `lower(st(1))`?
7. Was ist eine Zeichenkette bzw. warum kann man mit einem Index darauf zugreifen?
8. Wie muss man `function [x1,x2]=test(a,b)` aufrufen, damit die Ergebnisse für `a=1` und `b=2` den Variablen `m1` und `n1` zugewiesen werden?
9. Sind nach diesem Aufruf die Variablen `x1` und `x2` im MATLAB-Workspace bekannt?
10. Warum macht nach obiger Deklaration der Befehl `a=input('a')` keinen Sinn?
11. Wie muss ich obige Funktion aufrufen, wenn ich für `a` und `b` Vektoren übergeben will?
12. Wie kann man Variablen löschen?

13. Welche Befehle sind richtig und welche falsch (warum)?

`[1,2,3]*[2,3,4]`		
`[1,2,3]/5`		
`[1,2,3]^2`		
`[1,2;3,4]^2`		
`[1,2,3]*[1;2;3]`		
`1/[1,2,3]`		
`[1,2,3].^(1/2)`		
`[1,2,3].*[1,2,3,4]`		
`1./[1,2,3]`		

17.2.2 Antworten

1. Was sind richtige und falsche Namen von Variablen?
   `a12` `1a` `a-3` `a_12` `a(3)` `_bb` `maxi` `a.b`
   Korrekte Variablennamen müssen mit einem Buchstaben beginnen und dürfen ausser _ keine Sonderzeichen enthalten.

2. Wie kann man feststellen, welche Variablen im MATLAB-Arbeitsbereich bereits definiert sind?
   Mit den Befehlen `who` bzw. `whos`.

3. Wie kann man feststellen, ob ein Name bereits als Variable oder Funktion existiert?
   Mit dem Befehl `exist`.

4. Warum sollte man `i`, `j` oder z.B. `max` nicht als Variablennamen verwenden?
   Da sie intern in MATLAB verwendete Variablen bzw. Funktionen sind.

5. Wie erzeugt man einen Vektor mit 20 Zahlen, die equidistant zwischen `0` und `2` verteilt sind.
   `v=linspace(0,2,20)`

6. Gegeben ist eine Zeichenkette `st='Sinus'`. Welche Ausgabe erzeugen die Befehle `lower(st)`, `upper(st)`, bzw. `lower(st(1))`?
   Liefert `sinus`, `SINUS`, `s`.

7. Was ist eine Zeichenkette bzw. warum kann man mit einem Index darauf zugreifen?
   Eine Zeichenkette ist ein Array (Vektor) von Zeichen.

8. Wie muss man `function [x1,x2]=test(a,b)` aufrufen, damit die Ergebnisse für `a=1` und `b=2` den Variablen `m1` und `n1` zugewiesen werden?
   `[m1,n1]=test(1,2)`

9. Sind nach diesem Aufruf die Variablen `x1` und `x2` im MATLAB-Workspace bekannt?
   Nein! Funktionen arbeiten in einem eigenen Workspace.

10. Warum macht nach obiger Deklaration der Befehl `a=input('a')` keinen Sinn?
    `a` ist nach dem Aufruf bereits bekannt und muss nicht abgefragt werden.

11. Wie muss ich obige Funktion aufrufen, wenn ich für `a` und `b` Vektoren übergeben will?
    `[m1,n1]=test([1,2,3],[2,2,2])`

12. Wie kann man Variablen löschen?
    Mit dem Befehl `clear`.

13. Welche Befehle sind richtig und welche falsch (warum)?

`[1,2,3]*[2,3,4]`	falsch	Matrizenmultiplikation
`[1,2,3]/5`	richtig	jedes Element
`[1,2,3]^2`	falsch	Matrizenmultiplikation
`[1,2;3,4]^2`	richtig	quadratische Matrix
`[1,2,3]*[1;2;3]`	richtig	Matrizenmultiplikation = 14
`1/[1,2,3]`	falsch	Division durch Vektor
`[1,2,3].^(1/2)`	richtig	elementweise
`[1,2,3].*[1,2,3,4]`	falsch	unterschiedliche Länge
`1./[1,2,3]`	richtig	elementweise

Kapitel 18

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!

18.1 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 19

Anhang

19.1 Der Editor EMACS

Als eindeutig bester Editor für MATLAB hat sich der Editor EMACS erwiesen. Der bereitgestellte MATLAB-Mode bietet mit Hilfe von EMACSLINK eine Verbindung zwischen MATLAB und EMACS, erlaubt Syntax-Highlighting und Kommando-Ergänzung. Im Rahmen der Ausbildung bietet EMACS darüber hinaus weitere Vorteile, da dieser Editor ähnliche Unterstützung für andere Sprachen, wie z.B. C, C++, FORTRAN, PYTHON, oder LATEX bietet.

Die beste Verwendung von EMACS mit der Programmiersprache MATLAB ist der Aufruf aus MATLAB. Vorausgesetzt EMACSLINK ist in den Preferenzen eingestellt, startet man den Editor mit folgendem Befehl.

```
edit         % opens file untitled.m
edit file    % opens file or file.m (if file does not exist)
edit file.m
```

In Kontrast zum eingebauten MATLAB-Editor startet EMACS im Buffer `*scratch*` falls der gewünschte File nicht existiert. Man wird dann beim ersten Speichern aufgefordert den Filenamen anzugeben.

Ungeübte Benutzer können EMACS mit Hilfe der Maus und mit Menüeinträgen bedienen. Die wirkliche Stärke des Editors zeigt sich aber, wenn man zumindest die wichtigsten Tastenkombinationen verwendet. Diese werden in der Folge vorgestellt.

19.1.1 Buffer, Frame und Window

EMACS stellt für jeden geöffneten File einen Buffer bereit, der den gesamten Inhalt dieses Files beinhaltet. Außerdem können Buffer noch für andere Aufgaben verwendet werden:

`*scratch*`**:** Ein Bereich für Notizen, die nicht automatisch zum Speichern vorgesehen sind.

`*messages*`**:** Nachrichten von EMACS.

andere: Z.B. für die Ausgabe von Programmen, die aus EMACS aufgerufen werden.

Die Darstellung der Buffer erfolgt in sogenannten Frames (eigene Fenster am Bildschirm), die unter Umständen in mehrere Windows (Bereiche in einem Frame) aufgeteilt sein können. Der Inhalt eines Buffers kann nun in einem Window dargestellt werden, wobei der Inhalt eines Buffers auch in mehreren Windows vorhanden sein kann. Dies hat den Vorteil, dass man verschiedene Bereiche eines Files mehr oder weniger gleichzeitig bearbeiten kann. Man kann damit z.B. recht einfach Teile vom Anfang eines langen Files in einem weit entfernten Bereich einfügen.

Im unteren Bereich des Frames gibt es noch den sogenannten Minibuffer in dem EMACS-Kommandos ausgeführt werden. Dorthin springt z.B. der Editor, wenn ein File geöffnete wird.

19.1.2 Tastenkombinationen

Bei der Beschreibung von Tastenkombinationen werden folgende Abkürzungen verwendet:

`C-x`**:** Control-Key (`Strg` oder `Ctrl`) gleichzeitig gedrückt mit einer weiteren Taste (hier `x`).

`C-x s`**:** Control-Key gleichzeitig gedrückt mit einer weiteren Taste (hier `x`); Nach dem Loslassen drücken einer weiteren Taste (hier `s`).

`M-x`**:** Meta-Key (`Alt`) gleichzeitig gedrückt mit einer weiteren Taste.

`C-M-x`**:** Control- und Meta-Key gleichzeitig gedrückt mit einer weiteren Taste.

`ESC`**:** Escape-Key (`Esc`) gleichzeitig gedrückt mit einer weiteren Taste.

`RET`**:** Return- oder Eingabetaste.

`SPC`**:** Space oder Leerzeichen.

`TAB`**:** Tabulator.

`DEL`**:** Delete- oder Entferne-Taste.

`BSP`**:** Backspace- oder Lösch-Taste.

19.1.2.1 Files, Buffers und Windows

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
`C-z`	Suspend	`C-x C-c`	Beendet
`C-x C-s`	Speichert File	`C-x s`	Speichert alle Files
`C-x C-w`	Speichert File als	`C-x C-v`	Ersetzt File
`C-x C-f`	Öffnet File	`C-x i`	Einfügen eines Files
`C-x b`	Anderer Buffer	`C-x k`	Killt (entfernt) Buffer
`C-x C-b`	Liste aller Buffer		

Eine Eigenart von EMACS ist die Tatsache, dass man für das Öffnen eines neuen, nicht existierenden Files ebenfalls `C-x C-f` für das Anlegen verwendet. Man wird dann im Minibuffer zum Eingeben des Filenamens aufgefordert.

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-x 2	Split Window	C-x 5 2	Neuer Frame
C-x 0	Entfernt Window	C-x 5 0	Entfernt Frame
C-x 1	Entfernt andere Windows	C-x 3	Split Window seitlich
C-M-v	Scrollt anderes Fenster	C-x ^	Macht Window größer
C-x {	Macht Window schmäler	C-x }	Macht Window breiter
C-x o	Cursor in anderes Window	C-x 5 o	Cursor in anderen Frame
C-x 4 b	Auswahl Buffer in anderem Window	C-x 5 b	Auswahl Buffer in anderem Frame
C-x 4 C-o	Buffer in anderem Window	C-x 5 C-o	Buffer in anderem Frame
C-x 4 f	Öffne File in anderem Window	C-x 5 f	Öffne File in anderem Frame

19.1.2.2 Navigation durch den Buffer

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-b	Buchstabe rückwärts	C-f	Buchstabe vorwärts
M-b	Wort rückwärts	M-f	Wort vorwärts
C-p	Zeile rückwärts	C-n	Zeile vorwärts
C-a	Zeilenanfang	C-e	Zeilenende
M-a	Satz rückwärts	M-e	Satz vorwärts
M-{	Paragraph rückwärts	M-}	Paragraph vorwärts
M-x [	Seite rückwärts	M-e]	Seite vorwärts
C-M-a	Funktion rückwärts	C-M-e	Funktion vorwärts
M-<	Bufferanfang	M->	Bufferende
M-v	Scroll Schirm rückwärts	C-v	Scroll Schirm vorwärts
C-x <	Scroll Schirm links	C-x >	Scroll Schirm rechts
C-u C-l	Zeile in Schirmmitte		

19.1.2.3 Markieren, Kopieren und Löschen

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-SPC	Setze Markierung	C-x C-x	Tausche Markierung und Cursor
M-h	Markiere Paragraph	C-x C-p	Markiere Seite
C-M-h	Markiere Funktion	C-x h	Markiere Buffer
BSP	Entferne Zeichen rückwärts	DEL	Entferne Zeichen vorwärts
M-DEL	Lösche Wort rückwärts	M-d	Lösche Wort vorwärts
M-0 C-k	Lösche Zeile rückwärts	C-k	Lösche Zeile vorwärts
C-x DEL	Lösche Satz rückwärts	M-k	Lösche Satz vorwärts
C-w	Lösche Region	M-w	Kopiere Region
C-y	Einfügen	M-y	Tauscht letzte Einfügung gegen vorherige

Als Region bezeichnet man dabei einen markierten Bereich. Diese kann sowohl mit Hilfe der Tastatur, als auch mit Hilfe der Maus (linke Taste) markiert werden. Das Einfügen erfolgt ebenfalls mit Hilfe einer Tastenkombination C-y, bzw. mit Hilfe der mittleren Maustaste.

19.1.2.4 Formatierung, Änderung, Tausch

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
TAB	Einrückung Zeile	C-M-\	Einrückung Region
C-x TAB	Einrückung Region erzwungen		
C-o	Neue Zeile	C-M-o	Restliche Zeile nach unten
C-x C-o	Entferne leere Zeilen	M-^	Verbinde Zeile mit voriger
M-\	Entferne leere Zeichen	M-SPC	Exakt ein Leerzeichen
M-q	Fülle Paragraph	C-x f	Setzte Füll Spalte
M-u	Großbuchstaben Wort (upper)	M-l	Kleinbuchstaben Wort (lower)
M-c	Erster Buchstaben groß (capitalize)		
C-x C-u	Großbuchstaben Region (upper)	C-x C-l	Kleinbuchstaben Region (lower)
C-t	Austausch Zeichen	M-t	Austausch Worte
C-x C-t	Austausch Zeilen		

19.1.2.5 Suchen und Ersetzen

TASTE	BEDEUTUNG	TASTE	BEDEUTUNG
C-s	Suche vorwärts	C-r	Suche rückwärts
M-p	Wähle vorherigen Suchstring	M-n	Wähle nächsten Suchstring
RET	Beende Suche	C-g	Abbruch der Suche
M-%	Suche und Ersetzen (interactiv)		

Im Suchen und Ersetzen Modus M-% sind die folgenden Antworten möglich: ? Hilfe; SPC ersetze und geh weiter; , ersetze und bleibe am selben Platz; . ersetze und beende; DEL ersetze nicht und geh weiter; ! ersetze alle Weiteren; ^ zurück zum vorherigen Suchresultat; RET beende den Ersetzungsmodus; E editiere die Ersetzungszeichen.

Kapitel 20

Literatur

Es gibt eine Reihe sehr guter Bücher von zu MATLAB, die im Wesentlichen eine detailierte Dokumentation der Sprache und der Umgebung beinhalten. Sie liegen alle in englischer Sprache im PDF-Format vor. Eine Liste mit derartigen Dokumentationen gibt es auf unserem Wiki .

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. 20

[2] C. Überhuber and S. Katzenbeisser. *MATLAB 6 Eine Einführung*. Springer, 2000. 20

[3] D. Hanselman and B. Littlefield. *Mastering MATLAB 5, A Comprehensive Tutorial and Reference*. Prentice Hall, 1998. 20

[4] P. Marchand. *Graphics and GUIs with MATLAB*. ORC, second edition, 1999. 20

[5] G. Lindfield and J. Penny. *Numerical Methods Using MATLAB*. Ellis Horwood, 1995. 20

[6] H. Benker. *Mathematik mit MATLAB, Eine Einführung für Ingenieure und Naturwissenschaflter*. Springer, 2000. 20

[7] W. Gander and J. Hřebíček. *Solving Problems in Scientific Computing Using Maple and MATLAB*. Springer, third edition, 1997. 20