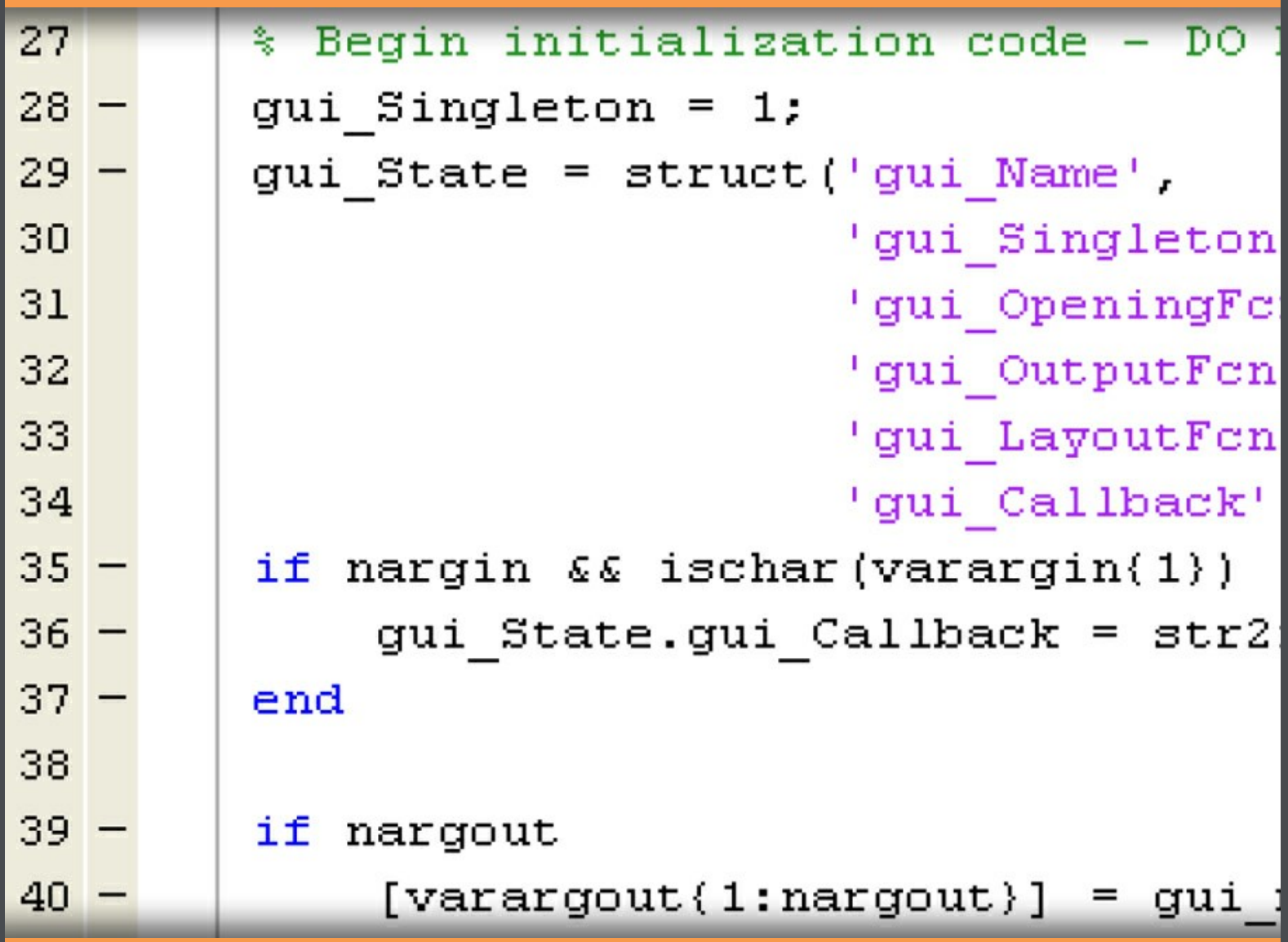
Einführungskurs Matlab & Simulink

Simon Ströbel



Simon Ströbel

Einführungskurs Matlab & Simulink

Einführungskurs Matlab & Simulink

ISBN 978-87-7681-810-4

Inhalt

Vorwort

Dieses Buch richtet sich im Wesentlichen an Studenten technischer und ingenieurwissenschaftlicher Studiengänge im Grundstudium, die nach einer Einführung in die Softwarepakete Matlab und Simulink suchen. Zum Verständnis des Buches sind daher lediglich Kenntnisse der Mathematik, Physik, Elektrotechnik, etc. notwendig, wie sie üblicherweise in den ersten beiden Semestern eines technischen Ingenieurstudiums an Universitäten oder Fachhochschulen vermittelt werden. Nach Durcharbeiten des Buches sollte der Leser in der Lage sein, Matlab effektiv in Studium und auch später im Berufsleben einzusetzen. Beispielsweise zur Verarbeitung, Analyse und Visualisierung von Messergebnissen aus Praktika, Projekt- und Diplomarbeiten oder zur numerischen Simulation dynamischer Prozesse und Systeme. Das Buch erhebt dabei jedoch keinen Anspruch auf Vollständigkeit, sondern stellt lediglich die aus Sicht des Autors wichtigsten Konzepte und Funktionalitäten der Programmpakete in komprimierter Form dar. Der Leser soll insoweit mit der Programmierumgebung und der Programmiersprache Matlab vertraut gemacht werden, dass er sich die für Ihn wichtigen weiterführenden Funktionalitäten durch die Dokumentation selbst aneignen kann.

Das Buch ist in sieben Kapitel gegliedert. In Kapitel 1 "Einführung" werden einige historische Fakten zu Matlab und die Einordnung von Matlab als "Werkzeug" im ingenieurwissenschaftlichen Kontext besprochen. In Kapitel 2 "Grundlagen" werden die fundamentalen Konzepte wie Variablendeklaration, Umgang mit Vektoren und Matrizen, Funktionsaufrufe und Datenvisualisierung behandelt. In Kapitel 3 "Matlab als Programmiersprache" wird vorgestellt, wie der Programmfluss in Matlab gesteuert und wie eigene Funktionen erstellt werden können. In Kapitel 4 "Ausgewählte mathematische Anwendungen" wird die umfangreiche mathematische Funktionsbibliothek auszugsweise vorgestellt, mit Hilfe derer auch komplexe Sachverhalte zielstrebig und intuitiv gelöst werden können. Der Fokus liegt dabei auf der Nutzung von Matlab zur Lösung mathematischer formulierter Problemstellungen selbst und nicht auf den mathematischen Hintergründen oder den in den jeweiligen Matlabfunktionen implementierten Algorithmen. Die eng verwandten Themengebiete in Kapitel 5 "Differentialgleichungen" und Kapitel 6 "Simulink" werden durch eine etwas ausführlichere Einführung ergänzt. Im letzten Kapitel 7 "Graphical User Interfaces" wird vorgestellt, wie Matlabprogramme mit grafischer Benutzerschnittstelle erzeugt werden können.

Alle behandelten Themengebiete werden jeweils anhand kleiner Beispiele erläutert. Für einen maximalen Lernerfolg sollten dabei alle im Buch aufgeführten Beispiele vom Leser praktisch nachvollzogen werden. Ergänzend zu diesem Buch ist der Titel *Grundkurs Matlab & Simulink: Aufgaben und Lösungen* verfügbar. Hier findet der Leser eine Sammlung von ausgewählten Aufgaben samt ausführlicher und kommentierter Musterlösungen. Es wird empfohlen, die jeweiligen Aufgaben parallel zum Studium des Buches zu bearbeiten bzw. die vorgestellten Musterlösungen nachzuvollziehen, um einen maximalen Lernerfolg zu erzielen.

Fragen, Kritik und Anregungen zum Text jedweder Art können Sie jederzeit gerne an simon.stroebel@hs-furtwangen.de senden.

1 Einführung

Matlab wurde Ende der 1970er Jahre von Cleve Moler an der Universität von New Mexico entwickelt. Zunächst wollte er damit lediglich seinen Studenten einen komfortablen Zugang zu den in der Programmiersprache Fortran geschriebenen Bibliotheken zur Linearen Algebra *Linpack* und *Eispack* ermöglichen, so dass diese ohne große Programmierkenntnisse anzuwenden waren. Zusammen mit Jack Little und Steve Bangert gründete er 1984 in Natick (Massachusetts, USA) die Firma The Mathworks, die Matlab zu einem kommerziellen Produkt weiterentwickelte und seither vertreibt.

Matlab hat sich in dieser Zeit zu einem "Allround Engineering Tool" weiterentwickelt. Matlab ist heutzutage eine eigenständige Programmiersprache, die auf die Lösung mathematischer, natur- und ingenieurwissenschaftlicher Problemstellungen zugeschnitten ist. Hierzu stellt Matlab eine umfangreiche und ausgereifte Funktionsbibliothek bereit. Matlab ist damit eine sogenannte *fourth generation language* (Programmiersprache der vierten Generation, kurz 4GL) deren Hauptmerkmal es ist, möglichst schnell und effizient Programme für einen bestimmten Anwendungszweck erstellen zu können. Historisch ist Matlab für matrizenbasierte numerische Berechnungen optimiert, woher sich auch das Akronym Matlab ableitet: *Mat*rix *Lab*oratory.

Simulink ist eine Erweiterung von Matlab und erlaubt im Wesentlichen die grafische Programmierung von differenzialgleichungsbasierten Modellen anhand von Blockschaltbildern, wie sie beispielsweise in der Systemtheorie, der Regelungstechnik und der Signalverarbeitung weit verbreitet sind.

Der Funktionsumfang von Matlab und Simulink kann durch Zusatzpakete, sogenannte Toolboxes (für Matlab) und Blocksets (für Simulink) erweitert werden, wobei diese in der Regel Funktionen und Blöcke für eine bestimmte wissenschaftliche Disziplin bereitstellen. Die folgende Abbildung zeigt einen Überblick über die Einsatzgebiete von Matlab und dessen Erweiterungen:

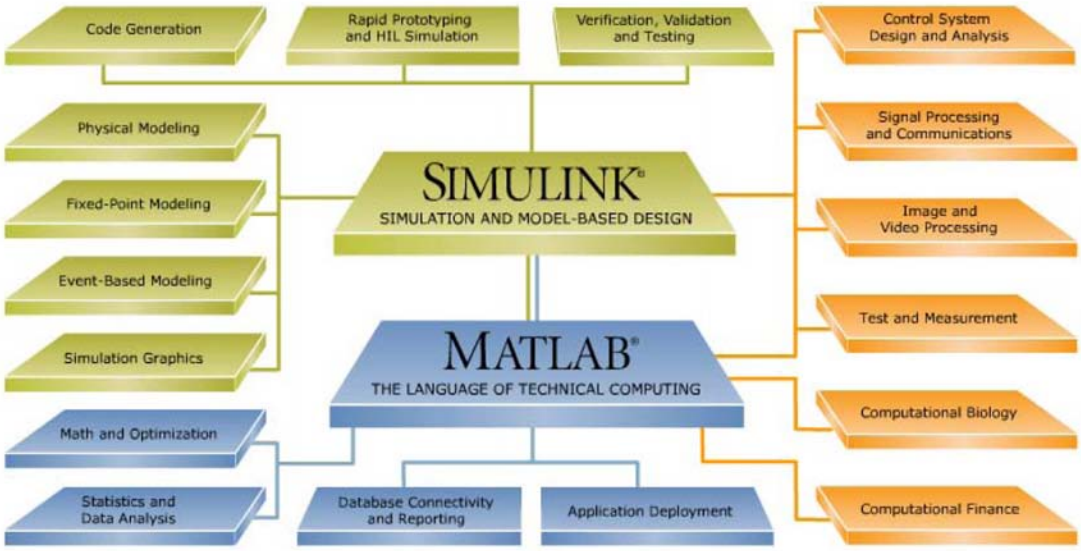


Abbildung 1: Matlab Produktfamilie (Quelle: www.mathworks.com)

Hervorzuheben ist, dass in den vergangenen Jahren seitens The Mathworks umfangreiche Anstrengungen unternommen wurden, um in Matlab oder Simulink erstellte Programme in andere Programmiersprachen konvertieren zu können oder direkt lauffähige Executables für bestimmte Plattformen erstellen zu können, beispielsweise durch die in Matlab R2011a neu eingeführten Produkte *Matlab Coder* und *Simulink Coder*.

Vor allem aufgrund der weiten Verbreitung im akademischen Umfeld besteht eine große User Community, welche zum Beispiel in der Matlab Central von The Mathworks selbst organisiert und unterstützt wird.

Matlab ist ein mächtiges Werkzeug zur Lösung mathematischer, naturwissenschaftlicher und technischer Problemstellungen. Jedoch sollte sich jeder angehende Matlab User über folgenden Sachverhalt von vornherein im Klaren sein: Matlab oder vergleichbare Softwareprogramme nehmen dem Ingenieur im Wesentlichen die *Rechenarbeit* zur Lösung einer mathematisch formulierten Problemstellung ab. Oftmals liegt die Problematik jedoch nicht darin ein Programm zu erstellen, welches eine interessierende Problemstellung löst, sondern überhaupt erst eine entsprechende mathematische Beschreibung dieser Problemstellung aufzustellen. Dieser mehr oder minder aufwendige *Modellbildungsprozess* steht eigentlich immer vor einer Berechnungs- oder Simulationsaufgabe jedweder Fachdisziplin. Gerade hier liegt jedoch oftmals die Kunst, die komplexen Sachverhalte realer naturwissenschaftlicher Phänomene und technischer Systeme insoweit zu vereinfachen und zu abstrahieren, dass das interessierende Problem in eine mit vertretbarem Aufwand lösbare mathematische Aufgabenstellung überführt werden kann.

Für den professionellen Einsatz leistungsstarker Engineering Software wie Matlab ist deshalb beides notwendig:

1. Eine fundierte Ausbildung und Erfahrung in der Fachdisziplin, in welcher der Ingenieur oder Wissenschaftler tätig ist,
2. Das Beherrschen der zur Problemlösung eingesetzten Software oder Programmiersprache selbst, in diesem Fall Matlab.

Dieses Buch legt den Schwerpunkt auf den zweiten Aspekt, das Kennenlernen von Matlab ohne einen spezielleren Bezug zu einer bestimmten Fachrichtung. Die Beispiele sind daher in der Regel recht allgemein gehalten, so dass keine bestimmten Fach- oder Programmierkenntnisse (aus höheren Semestern) notwendig sind. Aus diesem Grund wurde auch auf die Behandlung speziellerer mathematischer Anwendungen wie Integraltransformationen oder diskreter Operationen (z.B. FFT) verzichtet, da die nicht unerheblichen mathematischen Grundlagen hierzu oftmals erst in höheren Semestern vermittelt werden.

2 Grundlagen

2.1 Benutzeroberfläche

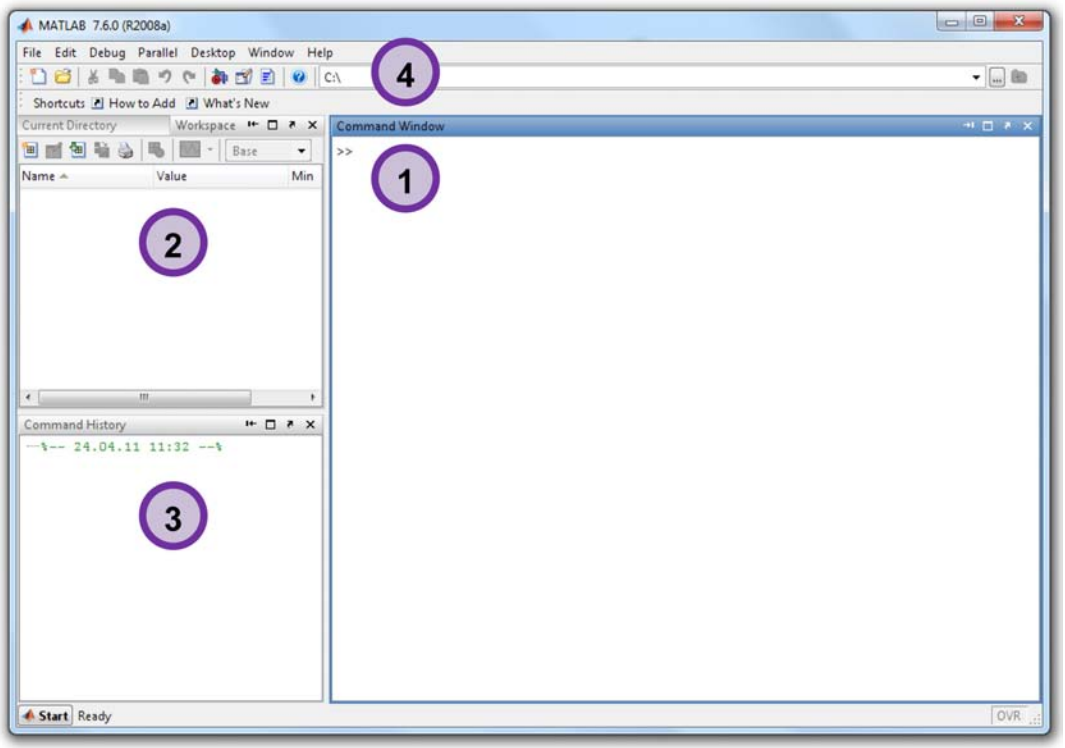


Abbildung 2: Matlab Startbildschirm

1. *Command Window*
   Das *Command Window* dient zur direkten Eingabe von Variablen, Rechenoperationen und zum Aufruf von Funktionen. Ebenso können hier die Ergebnisse der jeweiligen Berechnungen ausgegeben werden. Über das Command Window hat man damit im Wesentlichen einen sehr leistungsstarken "Taschenrechner".
2. *Workspace / Current Directory*
   Im *Workspace* werden alle vom Benutzer definierten Variablen gespeichert. Der Workspace ist sozusagen der Speicherbereich des Rechners, auf den Matlab zugreifen kann. In dem Fenster werden alle angelegten Variablen mit den entsprechenden Namen, Datentyp und Inhalt aufgelistet. Wechselt man auf den Reiter *Current Directory*, so werden alle Dateien und Ordner angezeigt, die sich im gegenwärtigen Arbeitsverzeichnis befinden.
3. *Command History*
   Im Fenster *Command History* werden alle zuletzt ausgeführten Befehle chronologisch gespeichert und aufgelistet. Durch Doppelklick auf die Befehle im Command History können diese einfach erneut ausgeführt werden.
4. *Current Directory*
   Unter *Current Directory* kann das aktuelle Arbeitsverzeichnis festgelegt werden.

2.2 Variablen und Wertzuweisung

2.2.1 Numerische Variablen

Zunächst werden Operationen mit konkreten Zahlenwerten behandelt, also mit *numerischen* Daten. Es soll eine einfache Additionsaufgabe gelöst werden. Geben Sie im Command Window hinter der Eingabeaufforderung `>>` (engl.: prompt) folgende Anweisung ein und drücken Sie anschließend Return/Enter:

```
>> 5+3
```

Im Command Window sehen Sie das Ergebnis der Additionsaufgabe:

```
ans = 8
```

Das Ergebnis wurde dabei automatisch der Variablen `ans` zugewiesen. `ans` ist die Standardausgabevariable und wird immer für die Speicherung des Ergebnisses der letzten Operation verwendet, wenn Sie dieses keiner anderen Variablen zuweisen. Sie sehen, dass im Workspace Window nun die Variable `ans` auftaucht und dieser unter *Value* der Wert 8 zugewiesen wurde. Sie können das Ergebnis der Berechnung einer anderen Variablen `a` zuweisen, indem Sie folgende Syntax verwenden:

```
>> a = 5+3
```

Das Ergebnis der Operation ist nun in der Variablen `a` gespeichert. Das Gleichheitszeichen `=` ist der sogenannte *Zuweisungsoperator*: Alles was rechts vom Gleichheitszeichen steht wird der links vom Gleichheitszeichen stehenden Variablen zugewiesen. Wie Sie vielleicht bemerkt haben ist es nicht notwendig der Variablen `a` explizit einen bestimmten Datentyp zuzuweisen, wie es beispielsweise in der Programmiersprache C zwingend erforderlich ist. Standardmäßig wird für alle numerischen Variablen der Datentyp *long* (double) verwendet, also eine 64-Bit Gleitkommazahl. Matlab kennt auch andere Datentypen, für die meisten Anwendungen genügt es jedoch mit dem Datentyp double zu arbeiten. Unter Matlab können beispielsweise auch logische Operationen mit dem double Datentyp realisiert werden, dazu später mehr.

Ebenso können Sie die Berechnungen natürlich auch mit Variablen selbst durchführen, die Sie zuvor definiert haben. Geben Sie beispielsweise folgendes im Command Window ein und schließen Sie jede Zeile mit Enter ab:

```
>> Zahl_1 = 2.5;
>> Zahl_2 = 4;
>> Ergebnis = Zahl_1*Zahl_2;
```

Der Dezimalpunkt wird in Matlab durch einem Punkt ausgedrückt, das deutsche Komma ist nicht zulässig. Sie sehen, dass alle drei Variablen im Workspace angelegt und die entsprechenden Werte zugewiesen werden. Das Semikolon am Ende einer Zeile unterdrückt dabei die Ausgabe im Command Window. Ändern Sie nun beispielsweise den Wert der Variablen `Zahl_1`, dann ändert sich die Variable `Ergebnis` nicht automatisch mit, Sie müssten hierzu die Zuweisung an `Ergebnis` erneut ausführen.

Bei der Vergabe der Variablennamen sind Sie im Wesentlichen frei. Variablennamen dürfen jeden beliebigen alphanumerischen Wert und Unterstriche enthalten, müssen jedoch mit einem Buchstaben beginnen und es dürfen keine Sonder- oder Leerzeichen im Variablennamen vorkommen. Darüber hinaus gibt es noch einige reservierte Schlüsselwörter, die Sie ebenfalls nicht als Variablennamen verwenden dürfen. Eine Liste dieser Schlüsselwörter erhält man über den Befehl `iskeyword`. Matlab ist *case sensitive*, es wird also zwischen Groß- und Kleinschreibung unterschieden. Des weiteren sollten Sie Variablen möglichst nicht mit den Buchstaben *i* und *j* bezeichnen, da diese für die Darstellung komplexer Zahlen verwendet werden:

```
>> Z = 4 + i*2;
```

oder

```
>> Z = 4 + 2j;
```

generiert eine komplexe Zahl mit 4 als Real- und 2 als Imaginärteil. Auch komplexe Zahlen werden im double Format gespeichert.

2.2.2 Variablen als Funktionsargumente

Real- und Imaginärteil, Betrag und Phasenwinkel sowie die konjugiert komplexe zu einer Zahl `Z` kann man mit folgenden rechts vom Zuweisungsoperator stehenden *Funktionen* ermitteln:

```
>> R = real(Z)
>> I = imag(Z)
>> M = abs(Z)
>> phi = angle(Z)
>> c = conj(Z)
```

Wie bereits angesprochen, bietet Matlab eine umfangreiche Bibliothek an mathematischen Funktionen. Funktionen werden in Matlab stets nach folgender Syntax aufgerufen:

```
>> val = function_name(arg_1,arg_2,…,arg_n)
```

`val` ist der Name der Variablen, der das Ergebnis des Funktionsaufrufes zugewiesen werden soll. `function_name` der Name der aufzurufenden Funktion und `(arg_1,arg_2,…,arg_n)` ist die *Argumentenliste* (Parameterliste), die die aufzurufende Funktion erwartet. Erwartet eine Funktion mehrere Argumente werden dies durch Kommas getrennt. Die Argumente werden immer in runde Klammern (engl.: parentheses) geschrieben. Beispiel Sinusfunktion:

```
>>  a = sin(pi/4)
```

liefert als Ergebnis:

```
a = 0.7071
```

Und die inverse Funktion (Arkussinus):

```
>> asin(a)
ans = 0.7854
```

Funktionen werden ausführlich in Kapitel 3.4 behandelt. Wie Sie sehen, ist die Kreiszahl π ebenso wie die imaginäre Einheit *i* bzw. *j* als Konstante in Matlab vordefiniert. Matlab arbeitet prinzipiell *numerisch*, liefert also immer nur einen Zahlenwert zurück, der das korrekte Ergebnis lediglich annähert:

$$\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \approx 0.7071... \qquad \frac{\pi}{4} \approx 0.7854...$$

Versuchen Sie sich an dieser Stelle den prinzipiellen Unterschied zwischen *symbolischer* und *numerischer Mathematik* klar zu machen.

Im Command Window gibt Matlab bei der Darstellung von Variablen lediglich vier Nachkommastellen an, das Ergebnis wird aber natürlich als 64-Bit Variable abgespeichert und steht für weitere Berechnungen auch in dieser Genauigkeit zur Verfügung. Die Anzahl der im Command Window dargestellten Nachkommastellen kann mit dem Befehl `format long` auf 15 erhöht werden. Mit `format short` werden wieder die standardmäßigen vier Nachkommastellen ausgegeben.

Matlab sucht zuerst im aktuellen Arbeitsverzeichnis nach der Funktion, die aufgerufen wird. Wird diese dort nicht gefunden, so wird in allen Verzeichnissen gesucht, die sie unter *File → Set Path* finden. Hier sind auch die Pfadangaben aller zusätzlich installierten Toolboxen gespeichert. Die verwendete Matlab Version mit allen installierten Toolboxen können Sie auch ermitteln, indem Sie im Command Window den Befehl `ver` eingeben. Alle im Workspace abgelegten Variablen können Sie mit den Befehlen `who` und `whos` ermitteln und auflisten.

Zehnerpotenzen wie beispielsweise 10^6 oder 10^{-6} können folgendermaßen eingegeben werden:

```
>> kleine_zahl = 1e-6;
>> grosse_zahl = 1e6;
>> eins = kleine_zahl*grosse_zahl

eins = 1
```

2.2.3 Löschen von Variablen

Eine nicht mehr benötigte Variable `x` kann mit `clear x` aus dem Workspace gelöscht werden. Der Befehl `clear` alleine löscht alle Variablen aus dem Workspace. Das Command Window kann mit dem Befehl `clc` bereinigt werden, alle Variablen bleiben hierbei jedoch im Workspace erhalten.

2.2.4 Zeichenketten

Neben Zahlen können Sie in Matlab auch *Zeichenketten* (*strings*) definieren. Hierzu wird die Zeichenkette in einfache Hochkommas gesetzt:

```
>> s = 'ich bin ein String'
```

Pro Zeichen (*character*) werden 2 Byte Speicherplatz belegt, hier also 18 x 2 Byte = 36 Byte. Zeichenketten sind in auch im numerischen Rechenprogramm Matlab von Bedeutung, da beispielsweise Messdaten oftmals als ASCII codierte Textdateien vorliegen, die in Matlab analysiert und weiter verarbeitet werden sollen (ASCII = American Standard Code for Information Interchange). Sie können strings auch in einen numerischen Wert wandeln. Zum Beispiel:

```
>> str = 'hallo!';
>> num = double(str)

num = 104     97    108    108    111     33
```

Was bedeuten diese sechs Zahlen? Die Variable `str` ist ein *Feld* (*Array*) bestehend aus sechs ASCII codierten Textzeichen. Der Befehl `double(str)` wandelt jedes Zeichen in den entsprechenden ASCII Wert und speichert diese im numerischen Feld `num`. Auch Ziffern werden in einem string als ASCII Code codiert:

```
>> str = '1234';
>> num = double(str)

num = 49     50     51     52
```

`num` ist ein Array mit vier *Elementen*. Die einzelnen Elemente sind haben die Werte 49, 50, 51, 52, also die ASCII Werte für die Ziffern 1, 2, 3 und 4. Die ASCII Tabelle ist beispielsweise auf Wikipedia zu finden.

Will man hingegen die als Zeichenkette vorliegende Zahl 1234 (Eintausendzweihundertvierunddreißig) in eine entsprechende numerische Zahlenvariable umwandeln, muss der Befehl `str2double()` oder `str2num()` angewendet werden:

```
>> num = str2double(str)
num = 1234
```

Nun ist in `num` wirklich die Zahl 1234 als numerischer Wert gespeichert.

2.3 Vektoren und Matrizen

Eine *Matrix* ist ein rechteckiges *Feld* von (im Allgemeinen komplexen) Zahlen. Die Anzahl der Elemente in jeder Spalte und in jeder Zeile muss daher zwingend identisch sein. Matrizen bestehen also immer aus m Zeilen und n Spalten, beziehungsweise man sagt eine Matrix habe die Dimension $m \times n$. *Skalare* können dabei als 1x1-Matrizen und *Vektoren* als Matrizen mit nur einer Zeile (Zeilenvektor) oder Spalte (Spaltenvektor) aufgefasst werden. Jedes Matrixelement a_{ij} einer Matrix A ist durch den Zeilenindex i und den Spaltenindex j eindeutig definiert:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Der versierte Umgang mit Vektoren und Matrizen ist in Matlab das "täglich Brot" und sollte daher von Anfang an fundiert geübt und verinnerlicht werden. Deshalb werden die hierzu wichtigsten Methoden und Funktionen ausführlich erklärt.

2.3.1 Generierung von Vektoren und Matrizen

Matlab ist ein Akronym für *Matrix Laboratory* und ist demnach für den Umgang mit Vektoren und Matrizen optimiert. Es ist in Matlab sehr einfach Vektoren und Matrizen zu erzeugen. Dies geschieht indem man die einzelnen Elemente in eckige Klammern (engl.: bracktes) schreibt:

```
>> z = [1, 2, 3, 4, 5, 6]
```
oder einfach
```
z = [1 2 3 4 5 6]
```

Dies erzeugt einen *Zeilenvektor* mit sechs Elementen 1, 2, …, 6. Der Vektor hat also die Dimension 1x6. Ersetzt man die Kommas bzw. Leerzeichen durch ein Semikolon so wird ein entsprechender Spaltenvektor der Dimension 6x1 erzeugt.

```
>> s = [1;2;3;4;5;6]
```

Die Summe der einzelnen Vektorelemente kann beispielsweise einfach mit dem Befehl `sum()` berechnet werden:

```
>> summe = sum(z)
ans =  21
```

Entsprechend erhält man das Produkt der einzelnen Elemente mit dem Befehl `prod()`. Ein Zeilenvektor kann mittels eines angehängten Hochkommas in einen Spaltenvektor umgewandelt werden um umgekehrt.

Matrizen werden entsprechend eingegeben:

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

Dies erzeugt eine Matrix der Größe 3x3, also mit je drei Zeilen und Spalten. Die Eingabe kann mit drei aufeinander folgenden Punkten auch auf mehrere Zeilen verteilt werden:

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

A =
      1      2      3
      4      5      6
      7      8      9
```

Die Dimension eines Vektors oder einer Matrix kann einfach mit dem Befehl `size()` ermittelt werden.

```
>> [m,n] = size(A)

m = 3
n = 3
```

Hierdurch wird in `m` die Anzahl der Zeilen und in `n` die Anzahl der Spalten von `A` gespeichert, die Funktion `size()` liefert in diesem Fall also zwei Rückgabewerte. Werden von einer Funktion zwei oder mehr Rückgabewerte erwartet, dann müssen diese durch Kommas getrennt in eckige Klammern geschrieben werden. Wird `size()` mit nur einem Rückgabewert aufgerufen, dann ist das Ergebnis ein Zeilenvektor der Dimension 1x2. Die Anzahl der Zeilen von `A` steht dann im ersten Element, die Anzahl der Spalten im zweiten.

```
>> dim = size(A)
dim = 3      3
```

Zusätzlich gibt es den Befehl `length()`. Auf eine Matrix angewendet gibt dieser lediglich die Anzahl der Spalten an. Auf einen Zeilenvektor der Länge n (1xn Matrix) angewendet gibt er die Anzahl der Elemente des Vektors an, ebenso bei einem Spaltenvektor der Länge m (mx1 Matrix). Der Befehl `numel()` gibt sowohl bei einer Matrix als auch bei einem Vektor die gesamte Anzahl der Elemente an.

Mit `A'` erhält man die Transponierte einer Matrix, also die Matrix `A` an der Hauptdiagonale gespiegelt:

```
>> A'

ans = 1     4     7
      2     5     8
      3     6     9
```

Wird der Befehl `sum()` auf eine Matrix angewendet, so wird die Summe einer jeden Spalte der Matrix berechnet.

```
>> sum(A)
ans = 12    15    18
```

Sollen hingegen alle Elemente der Matrix aufaddiert werden, so kann der Befehl `sum()` einfach zweimal (verschachtelt) angewendet werden.

```
>> sum(sum(A))
ans = 45
```

Durch den inneren `sum()` Befehl werden die drei Spaltensummen berechnet, welche anschließend in einem Zeilenvektor stehen. Der äußere `sum()` Befehl berechnet die Summe dieses Zeilenvektors, sodass das Ergebnis die Summe aller Elemente in `A` ist.

2.3.2 Elementzugriff

Auf die einzelnen Elemente von Vektoren kann folgendermaßen zugegriffen werden:

```
>> el1 = z(2);
```

beziehungsweise

```
>> el2 = s(2);
```

Dies liefert jeweils das zweite Element des zuvor definierten Zeilenvektors `z` oder des Spaltenvektors `s`, hier also den Wert 2.

```
>> el3 = A(2,3);
```

Dies liefert das Element der Matrix `A`, welches in der zweiten Zeile und der dritten Spalte steht.

Anmerkung: Im Unterschied zur Programmiersprache C beginnt die Indizierung von Vektoren und Matrizen (Arrays) in Matlab generell mit dem Wert 1, nicht mit dem Wert 0. Dies ist eine häufige Fehlerquelle für Neulinge in Matlab. Die Indizes müssen natürliche Zahlen sein, können aber vom Datentyp double (Gleitkommazahl) sein.

Wichtig in Matlab ist der *Doppelpunkt Operator* (engl.: colon operator). Er spricht *alle Elemente* einer Spalte oder Zeile an:

```
>> z2 =  A(1,:)
```

`z2` ist ein Zeilenvektor mit allen Elementen der ersten Zeile von `A`.

```
>> s2 = A(:,3)
```

`s2` ist ein Spaltenvektor mit allen Elementen der dritten Spalte von `A`.

```
>> B = A(:,:)
```

`B` ist identisch mit der Matrix `A`.

```
>> C = A(1:2,1:2)

C =   1      2
      4      5
```

`C` Ist eine *Untermatrix* von `A`, bestehend aus den ersten beiden Spalten und den ersten beiden Zeilen von `A`. Der Doppelpunkt Operator erfüllt dann quasi eine *von-bis* Funktion.

Äquidistant geteilte Vektoren können sehr mit dem Doppelpunkt Operator generiert werden:

```
>> v1 = [0:100];
```

Dies generiert einen Zeilenvektor mit den Elementen 0,1,2,…100 also mit insgesamt 101 Elementen.

```
>> v10 = [0:10:100];
```

Generiert einen Zeilenvektor mit der Schrittweite 10, also mit den Elementen 0, 10, 20, … 100. In Worten bedeutet die Anweisung: Erstelle einen Vektor beginnend beim Wert 0 mit der Schrittweite 10 bis zum Wert 100. Äquivalent gibt es hierzu den Befehl `linspace()`, der folgendermaßen benutzt wird:

```
>> v1 = linspace(min,max,anzahl)
```

Erzeugt einen Zeilenvektor mit `anzahl` Elementen. Dabei hat das erste Element den Wert `min` und das letzte den Wert `max`. Alle Elemente haben den gleichen Abstand voneinander, der Vektor ist also äquidistant. Entsprechend gibt es den Befehl `logspace()`, durch den man logarithmisch geteilte Zeilenvektoren erhält. Hierbei sind die Argumente `min` und `max` die *Exponenten* zur Basis 10 des kleinsten und größten Vektorelements.

```
>> logspace(0,3,4)
ans =   1      10      100      1000
```

2.3.3 Standardmatrizen

Nützliche Befehle zur einfachen Erzeugung von Vektoren und Matrizen der Dimension *mxn* sind beispielsweise:

Funktion	Bedeutung
zeros(m,n)	Matrix mit ausschließlich 0 Elementen
ones(m,n)	Matrix mit ausschließlich 1 Elementen
eye(m,n)	Matrix mit 1 Elemente in der Hauptdiagonale, ansonsten 0 Elementen (Einheitsmatrix)
rand(m,n)	Matrix mit *gleichverteilten Zufallszahlen* zwischen 0 und 1
randn(m,n)	Matrix mit *normalverteilten Zufallszahlen* mit Mittelwert 0 und Standardabweichung 1

Tabelle 1: Standardmatrizen

Mit dem Befehl

```
>> A = [];
```

wird eine leere 0x0 Matrix erzeugt, diese enthält *keine* Elemente.

2.3.4 Matrixmanipulationen

Vektoren und Matrizen können einfach zusammengefügt (engl.: to concatenate) werden:

```
>> a = [1 2 3];
>> b = [10 20 30];
>> c = [a b]

c = 1      2      3     10     20     30
```

Es ist hierbei darauf zu achten, dass die Zusammensetzung "sinnvoll" ist, die Dimensionen der Vektoren oder Matrizen also zueinander passen. So führt beispielsweise der Versuch einen Zeilenvektor mit einem Spaltenvektor zu verbinden zu einer Fehlermeldung:

```
>> a = [1 2 3];
>> b = [10;20;30];
>> c = [a b]

??? Error using ==> horzcat
CAT arguments dimensions are not consistent.
```

Wie man an der Fehlermeldung sieht, benutzt Matlab intern die Funktion `horzcat()` beim Versuch die beiden Vektoren zusammenzusetzen. Diese Funktion kann auch direkt vom Command Window aus aufgerufen werden und hat genau den gleichen Effekt wie das erste Beispiel:

```
>> a = [1 2 3];
>> b = [10 20 30];
>> c = horzcat(a,b)

c = 1      2      3     10     20     30
```

Analog gibt es die Funktion `vertcat()` zum Zusammensetzen von Spaltenvektoren. Ebenso können echte Matrizen mit passender Dimension zusammengesetzt werden:

```
>> M1 = [1 1 1;1 1 1];
>> M2 = [2 2 2;2 2 2];
>> HOR = [M1 M2]

HOR = 1      1      1      2      2      2
      1      1      1      2      2      2

>> VERT = [M1;M2]

VERT = 1      1      1
       1      1      1
       2      2      2
       2      2      2
```

Weitere hilfreiche Funktionen zur Matrix-Manipulation sind die `flip__()` Funktionen, mit denen Matrizen und Vektoren gespiegelt werden können:

```
>> fliplr(HOR)

ans = 2      2      2      1      1      1
      2      2      2      1      1      1
```

Führt eine links/rechts Spiegelung durch.

```
>> flipud(VERT)

ans = 2      2      2
      2      2      2
      1      1      1
      1      1      1
```

Führt eine oben/unten Spiegelung durch. Ergänzend sei noch die Funktion `rot90()` erwähnt, mit der Matrizen um 90° gegen den Uhrzeigersinn gedreht werden können. Die Zeilen- und Spaltendimensionen tauschen somit bei jeder Anwendung von `rot90()` ihren Wert.

```
>> rot90(M1)

ans = 1      1
      1      1
      1      1
```

Wenn man nicht weiß, ob in einer Variablen ein Zeilen- oder Spaltenvektor steht, kann man durch Anwendung des Doppelpunkt Operators sicherstellen, dass ein Spaltenvektor vorliegt.

```
>> zeilenvektor = [1 1 1];
>> spaltenvektor = [2;2;2];
>> zeilenvektor(:)

ans = 1
      1
      1

>> spaltenvektor(:)

ans = 2
      2
      2
```

Durch anschließendes Transponieren mit einem einfachen Hochkomma hinter dem Variablennamen, kann man sodann sicherstellen, dass ein Zeilenvektor vorliegt.

```
>> zeilenvektor'
ans =   1       1       1

>> spaltenvektor'
ans =   2       2       2
```

Vektoren und Matrizen können problemlos mit einem Skalar multipliziert werden. Hierbei wird jedes Element des Vektor/der Matrix mit dem Skalar multipliziert.

```
>> skalar = 2;
>> matrix = [1 2;3 4];
>> skalar*matrix

ans =   2       4
        6       8
```

2.4 Matlab Hilfe und Dokumentation

Bisher haben wir schon einige *Funktionen* kennengelernt, beispielsweise die `sum()` oder die `size()` Funktion. Es dürfte jedoch unmöglich sein, alle von Matlab zur Verfügung gestellten Funktionen und Befehle und insbesondere die oftmals verschiedenen Möglichkeiten eines Funktionsaufrufs auswendig zu lernen. Matlab bietet hierzu eine sehr umfangreiche und professionelle Dokumentation.

Die einfachste und schnellste Möglichkeit zu einer Matlab Funktion eine Kurzbeschreibung zu erhalten ist der `help` Befehl. Hierzu muss einfach hinter das Schlüsselwort `help` der Name der nachzuschlagenden Funktion geschrieben werden, woraufhin eine Kurzbeschreibung im Command Window erscheint. Durch:

```
>> help sin
```

Erhält man am Command Window die Hilfe zur Sinusfunktion Funktion `sin()`:

```
>> help sin
 SIN     Sine of argument in radians.
    SIN(X) is the sine of the elements of X.
    See also asin, sind.
    Overloaded methods:
        distributed/sin
        sym/sin
    Reference page in Help browser
        doc sin
```

`sin(x)` berechnet also die Sinuswerte der Elemente von `x`, wobei `x` in Radiant gegeben sein muss. Durch den Hinweis, dass `sin()` auf die Elemente von `x` wirkt, ist klar, dass `sin()` nicht nur auf Skalare, sondern auch auf Vektoren und Matrizen angewendet werden ist.

Der `help` Befehl verweist unter *See Also* oftmals direkt auf ähnliche oder verwandte Funktionen. So können Sie durch `help sind` erfahren, dass es eine zweite Sinusfunktion `sind()` gibt, welche die Argumente `X` in *Grad* erwartet.

Darüber hinaus sehen sie einen Link *doc sin*. Folgen Sie diesem Link, so öffnet sich automatisch der *Matlab Help Browser* mit dem entsprechenden Kapitel. Den Help Browser können Sie auch direkt öffnen, indem Sie im Command Window `doc sin` eingeben.

```
>> doc sin
```

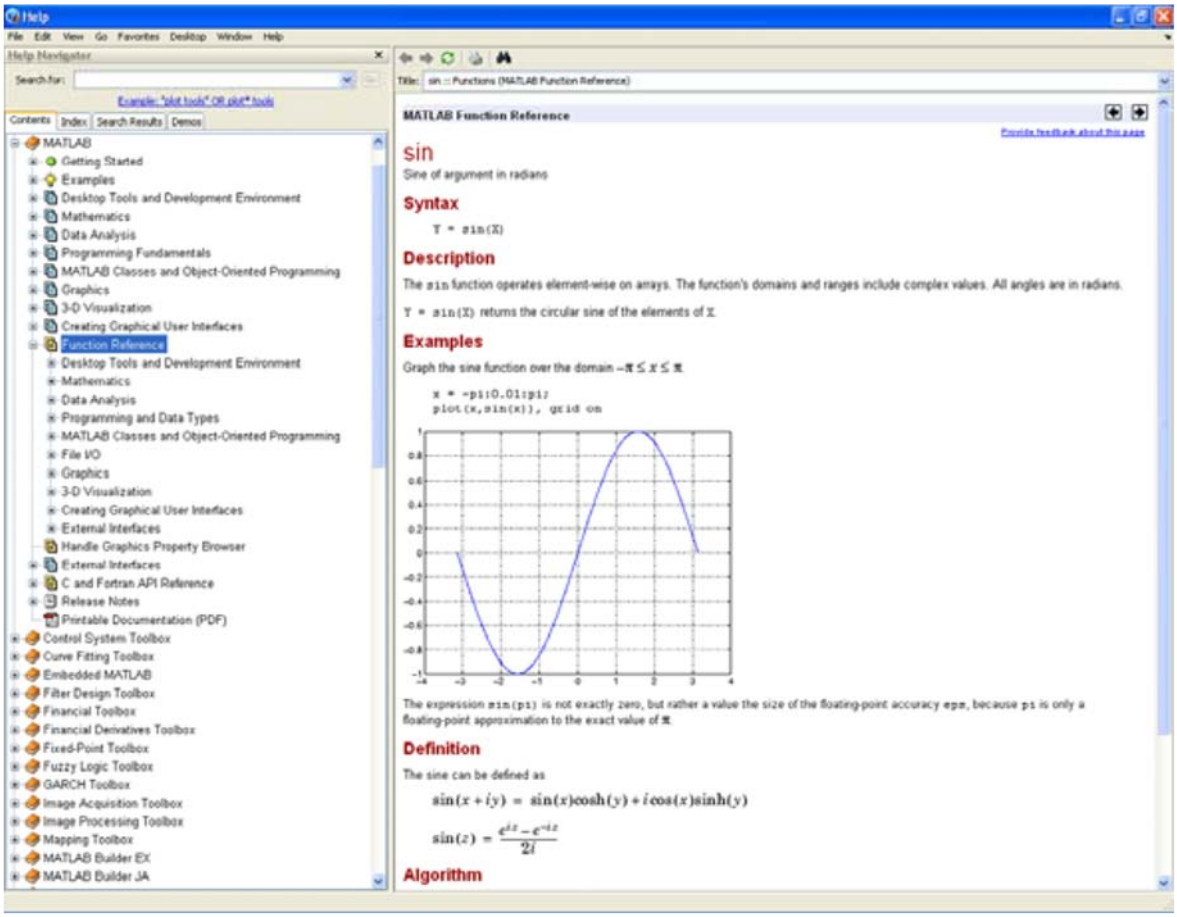


Abbildung 3: Matlab Help Browser

Die Hilfe ist sehr ausführlich, manchmal auch mit Beispielen zum Aufruf und zur Anwendung der jeweiligen Funktion versehen. Die Hilfe ist auch über den Reiter *Help → Product Help* oder über die Taste *F1* erreichbar. Die Hilfe zu jedem Matlab Produkt (jeder Toolbox) ist auch als PDF verfügbar und kann von der Mathworks Homepage heruntergeladen werden: http://www.mathworks.de/help/.

Die Hilfe ist ein sehr wichtiges und mächtiges Instrument, machen Sie sich damit vertraut. Die Hilfe und die Dokumentationen bieten eine Vielzahl an Tutorials, Beispielen und Funktionsreferenzen (alphabetisch oder nach Kategorien sortiert). Da Matlab an sich ein sehr umfangreiches Werkzeug ist, ist konsequenterweise auch die Dokumentation sehr umfangreich.

2.5 Operatoren

2.5.1 Arithmetische Operatoren

Mit Numerischen Variablen können verschiedene mathematische Operationen durchgeführt werden. Es stehen folgende elementaren Rechenoperationen zur Verfügung:

Symbol	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division (Rechtsinverse)
\	Kehrwert (Linksinverse)
^	Potenz
'	Transponierte

Tabelle 2: Arithmetische Operatoren

Zu beachten ist hierbei die "umgekehrte" Division mit der Linksinversen \. Vergleichen Sie

```
>> 3/4
ans =   0.7500
>> 3\4
ans = 1.3333
```

Anmerkung: Mit der Linksinversen lassen sich sehr leicht auch Lineare Gleichungssysteme lösen, hierzu mehr im Kapitel 4.4.

Alle Operatoren können auch auf Vektoren und Matrizen angewendet werden. Bei der Addition und Subtraktion werden einfach die einzelnen Elemente der Vektoren und Matrizen miteinander addiert/subtrahiert. Die Dimension der einzelnen Summanden muss hierzu aber übereinstimmen:

```
>> a = [-2 -4 -6]+[2 4 6]
a = 0      0      0
>> a = [-2 -4 -6]+[2;4;6]

??? Error using ==> plus
Matrix dimensions must agree.
```

Aus der Fehlermeldung wird ersichtlich, dass der Aufruf der Funktion `plus()` aufgrund nicht zueinander passender Dimensionen ungültig war. Für jeden Operator gibt es also auch eine entsprechende Funktion, z.B. für "+" die Funktion `plus()`. Bei der Multiplikation von Matrizen ist darauf zu achten, dass hierbei standardmäßig nicht elementweise multipliziert wird, sondern nach der allgemeinen Regel der *Matrizenmultiplikation*:

$$C = A \cdot B \qquad \text{mit} \qquad c_{ij} = \sum_{k} a_{ik} \cdot b_{kj}$$

Beispiel:

```
>> A = [1 2;-2 4]

A =  1     2
    -2     4

>> B = [2 -3;6 2]

B =  2    -3
     6     2

>> C = A*B

C = 14     1
    20    14
```

Diese Operation ist nur zulässig, wenn die inneren Matrixdimensionen übereinstimmen. Die Anzahl der Spalten der Matrix `A` muss also gleich der Anzahl Zeilen der Matrix `B` sein. Die Operation

```
>> A = [1 2;-2 4];
>> B = [2 -2];
>> C = A*B

??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

resultiert somit in einem Fehler, da `A` zwei Spalten `B` aber nur eine Zeile hat. Hingegen kann

```
>> A = [1 2;-2 4];
>> B = [2;-2];
>> C = A*B

C = -2
    -12
```

problemlos berechnet werden, da hier die Anzahl der Spaltenanzahl von `A` und der Zeilenanzahl von `B` übereinstimmt, somit ist auch klar, dass bei der Matrizenmultiplikation gilt:

$$A \cdot B \neq B \cdot A$$

Oftmals ist es auch erforderlich, die einzelnen Elemente zweier Matrizen mit gleichen Indizes miteinander zu multiplizieren, also eine *elementweise Multiplikation* durchzuführen.

$$C = A \cdot B \quad \text{mit} \quad c_{ij} = a_{ij} \cdot b_{ij}$$

Diese Operation lässt sich mit einem vor das Multiplikationszeichen gestellten *Punkt* realisieren:

```
>> A = [1 2;-2 4];
>> B = [2 -3;6 2];
>> C = A.*B

C =   2    -6
    -12     8
```

Hierzu muss folglich die Dimension von `A` und `B` übereinstimmen. Der Punkt Operator ist immer vor einem Rechenoperator zu verwenden, wenn die Berechnung elementweise erfolgen soll.

Potenzen können einfach mit dem ^ Operator berechnet werden. Dabei gilt ebenfalls wieder, dass die Potenzfunktion auf eine Matrix als solche und nicht auf die einzelnen Element wirkt. Will man Elementweise potenzieren, so muss man wieder den Punkt vor den Operator stellen.

Beispiel:

```
>> A = [1 2 2;3 2 2;1 2 1];
>> A^2

ans =
      9     10      8
     11     14     12
      8      8      7

>> A.^2

ans =
      1      4      4
      9      4      4
      1      4      1
```

Wie bereits gesehen, können die Operatoren können auch durch entsprechende Funktionen ersetzt werden:

Symbol	Funktion
+	plus()
-	minus()
*	mtimes()
/	mrdivide()
\	mldivide()
^	mpower()
.*	times()
./	rdivide()
.\	ldivide()
.^	power()

Tabelle 3: Arithmetische Operatoren und Funktionen

2.5.2 Vergleichsoperatoren

Matlab erlaubt die folgenden Vergleichsoperationen:

Symbol	Funktion	Bedeutung
==	eq()	Gleich
~=	ne()	Ungleich
>	gt()	Größer
<	lt()	Kleiner
>=	ge()	Größer gleich
<=	le()	Kleiner gleich

Tabelle 4: Vergleichsoperatoren

Ist eine Bedingung wahr, so liefert eine Vergleichsoperation eine logische 1 (Boolscher Wahrheitswert *true/wahr*) zurück ansonsten eine logische 0 (bzw. Wahrheitswert *false/falsch*). Der Datentyp des Vergleichsergebnis ist also eine *Boolesche Variable*, kein double Wert. Boolesche Werte sind in Matlab vom Datentyp *logical*. Vergleiche können mit Skalaren, Vektoren und Matrizen durchgeführt werden, wobei die jeweiligen Vektoren und Matrizen, dieselbe Dimension besitzen müssen und der Vergleich elementweise erfolgt.

Beispiele:

```
>> 1>2
ans = 0
>> ne([0 0 1 1],[0 1 0 1])
ans = 0     1     1     0
```

Eine nützliche Eigenschaft von *logical* Variablen ist, dass diese ebenfalls zur Indizierung von Vektoren und Matrizen herangezogen werden können:

```
>> A = [1 2 3;4 5 6; 7 8 9]
A =  1      2      3
     4      5      6
     7      8      9
>> B = [9 8 7;6 5 4; 3 2 1]
B =  9      8      7
     6      5      4
     3      2      1
>> logical_AgtB = A>B
logical_AgtB =
     0      0      0
     0      0      1
     1      1      1
```

Indidzierung der Matrix `A` mit der Matrix `logical_AgtB`:

```
>> A(logical_AgtB)
ans =
     7
     8
     6
     9
```

Ein weiterer nützlicher Befehl ist in diesem Zusammenhang die `find()` Funktion. Mit ihr kann eine Variable auf einen logischen Ausdruck hin untersucht werden. Die Funktion liefert die *Indizes* (nicht die Werte!) der Variablen zurück, für die dieser Ausdruck wahr ist:

```
>> x = [-2*pi:pi/4:2*pi];
>> y = sin(x);
>> I = find(y>0.5)
I = 2      3      4     10     11     12
```

Die entsprechenden Werte kann man aber einfach erhalten, indem man den Vektor `I` nun zur Indizierung des Vektors `y` verwendet.

```
>> y(I)
ans = 0.7071     1.0000     0.7071     0.7071     1.0000     0.7071
```

Der allgemeine Aufruf des Befehls `find()` erfolgt mit drei Argumenten:

```
I = find(logOp,k,'first')
```

beziehungsweise

```
I = find(logOp,k,'last')
```

`logOp` ist hierbei die logische Operation auf die entsprechende Variable angewandt. Mit dem Argument `k` kann zusätzlich angegeben werden, dass nur die `k` ersten bzw. letzten Indizes zurückgegeben werden, die der logischen Operation entsprechen. Beispiel:

```
>> v = [0:10:100]
v = 0     10     20     30     40     50     60     70     80     90    100

>> I = find(v>40,2,'last')
I = 10     11
```

2.5.3 Logische Operatoren

Matlab erlaubt die folgenden Vergleichsoperationen:

Symbol	Funktion	Bedeutung
&	and()	logisches Und
&&		logisches Und (short circuit)
|	or()	logisches Oder
||		logisches Oder (short circuit)
~	not()	logisches Nicht
	xor()	logisches Exklusiv Oder
	any()	wahr, wenn ein Element eines Vektors ungleich null ist
	all()	wahr, wenn alle Elemente eines Vektors ungleich null sind

Tabelle 5: Logische Operatoren

Numerische Variablen werden als logisch 1 gewertet, wenn Sie einen Wert ungleich 0 besitzen. Beispiele:

```
>> 1&2
ans = 1

>> -1&2.5
ans = 1

>> 0&pi
ans =  0

>> A = [0 0 1 1];
>> B = [0 1 0 1];
>> C = xor(A,B)
C =  0     1     1     0
```

Bei den *short circuit* Befehlen wird die zweite Bedingung lediglich dann ausgewertet, wenn dies aus logischen Gründen notwendig ist, ansonsten nicht. Dies kann die Ausführungsgeschwindigkeit optimieren: Wenn der zweite Ausdruck `B` zunächst wieder eine komplexe Verkettung logischer Ausdrücke ist, dann muss beispielsweise bei einer *UND* Verknüpfung dieser nur dann ausgewertet werden, wenn der Ausdruck `A` logisch 1 ist, ansonsten ist die Verknüpfung *A UND B* unabhängig von `B` logisch 0. Beispiel:

```
>>   a = 1;
>>   b = 2;
>>   c = 3;
>>   d = 4;
>>   x = 0;
>>   L = (x) && (a<b & b<c & d>=c | b<=2)
```

Machen Sie sich klar was den Ausdrücken `A` und `B` entspricht.

2.5.4 Logische bitweise Operatoren

Die *logischen bitweisen Operatoren* unterstützen nur ganze positive Zahlen `A` und `B`. Bei bitweisen logischen Operatoren wird die ganze Zahl zunächst in ihre binäre Darstellung gewandelt, dann die entsprechende logische Operation ausgeführt und die Zahl anschließend wieder zurück ins Dezimalformat konvertiert.

Funktion	Bedeutung
bitand(A,B)	bitweises Und
bitor(A,B)	bitweises Oder
bitxor(A,B)	bitweises Exklusiv Oder
bitshift(A,k)	Schiebt die Bits von A um k Stellen nach links (k>0) oder rechts (k<0)
bitset(A,n)	Setzt das das n-te Bit in A
bitget(A,n)	Liefert den Wert des n-ten Bits in A zurück

Tabelle 6: Logische bitweise Operatoren

Beispiel:

Ausdruck	Dezimal	Binär
A	6	0110
B	11	1011
bitand(A,B)	2	0010

Tabelle 7: bitweise logische UND-Verknüpfung

```
>> A = 6;
>> B = 11;
>> bitand(A,B)

ans = 2
```

Das Ergebnis ist also ein numerischer Wert, kein logischer wie bei den herkömmlichen Vergleichsoperationen.

2.5.5 Rundungsoperatoren

Matlab stellt verschiedene Rundungsfunktionen zur Verfügung, mit denen eine Dezimalzahl zu einer bestimmten Ganzzahl (Integer) gerundet werden kann.

Funktion	Bedeutung
round(x)	Rundet x zur nächsten Ganzzahl
floor(x)	Rundet x zur nächst kleineren Ganzzahl
ceil(x)	Rundet x zur nächst größeren Ganzzahl
fix(x)	Rundet x zur nächsten bei 0 gelegenen Ganzzahl

Tabelle 8: Logische bitweise Operatoren

Beispiel:

```
>> x = [-1.3 -0.7 -0.5 -0.3 0.3 0.5 0.7 1.3];
>> round(x)
ans =   -1     -1     -1      0      0      1      1      1
>> floor(x)
ans =   -2     -1     -1     -1      0      0      0      1
>> ceil(x)
ans =   -1      0      0      0      1      1      1      2
>> fix(x)
ans =   -1      0      0      0      0      0      0      1
```

2.6 Datenvisualisierung

Matlab bietet eine Vielzahl an Möglichkeiten, Ergebnisse und Daten aufbereitet auszugeben und zu visualisieren. Für die folgenden Beispiele erzeugen wir zunächst drei Datenvektoren `t`, `x` und `y`. Dabei beinhaltet `t` Werte auf der Zeitsachse und `x` und `y` repräsentieren Funktionen in Abhängigkeit der Zeit.

$$x_{(t)} = 1$$

$$y_{(t)} = 5 \cdot \sin\left(2t + \frac{\pi}{8}\right) \cdot e^{-\frac{t}{2}} + 2$$

```
>> t = linspace(0,10,1001);
>> x = ones(1,length(t));
>> y = 5*sin(2*t+pi/8).*exp(-.5*t)+2;
```

2.6.1 Textausgabe

Am Bildschirm bzw. im Command Window soll nun eine Ausgabe erfolgen, so dass in jeder Zeile die Werte von `t`, `x`, und `y` eines bestimmten Zeitpunkts stehen, danach ein Zeilenumbruch erfolgt und die Werte des nächsten Zeitpunkts folgen. Dies lässt sich mit dem Befehl `fprintf()` folgendermaßen realisieren:

```
>> fprintf('t = %4.2f    x = %1.0f    y = %6.4f\n',[t;x;y])
```

Führt man obigen Code aus, so erhält man im Command Window die folgende formatierte Ausgabe:

```
t = 0.00    x = 1    y = 3.9134
t = 0.01    x = 1    y = 3.9954
t = 0.02    x = 1    y = 4.0758
t = 0.03    x = 1    y = 4.1544
t = 0.04    x = 1    y = 4.2314
t = 0.05    x = 1    y = 4.3066
t = 0.06    x = 1    y = 4.3802
t = 0.07    x = 1    y = 4.4520
t = 0.08    x = 1    y = 4.5220
...
```

Das Erste Argument von `fprintf()` ist eine Zeichnekette (violett in einfachen Hochkommas), der den auszugebenden Text enthält. Durch das `%`-Zeichen innerhalb der Zeichenkette wird angekündigt, dass der folgende Ausdruck durch eine Variable ersetzt werden soll. Im ersten Fall soll der Wert von `t` mit einer maximalen Breite von vier Ziffern und zwei Nachkommastellen ausgegeben werden, im zweiten Fall der Wert von `x` mit nur einer Ziffer und keiner Nachkommastelle und im dritten Fall der Wert von `y` mit maximal sechs Ziffern und vier Nachkommastellen. Durch `\n` wird ein Zeilenumbruch erzeugt. Nach Abschluss der Zeichenkette folgt eine Matrix mit so vielen Zeilen wie `%`-Zeichen in der Zeichenkette vorhanden sind.

Diese numerische Ausgabe kann man anstatt auf dem Bildschirm auszugeben auch sehr leicht in eine Textdatei umleiten. Hierzu muss lediglich mit `fopen()` eine entsprechende Datei geöffnet werden, die Daten mit `fprintf()` in die Datei geschrieben und die Datei mit `fclose()` wieder geschlossen werden.

```
>> fid = fopen('funktionen.txt','w');
>> fprintf(fid,'t = %4.2f %1.0f %6.4f\n',[t;x;y]);
>> fclose(fid);
```

Das erste Argument im Befehl `fopen()` ist der Dateiname, in der die Daten geschrieben werden sollen. Das zweite Argument `'w'` (für *write*) bewirkt, dass Daten in die Datei geschrieben werden können.

2.6.2 2D Linienplots

Eine grafische Ausgabe der Vektoren `t`, `x` und `y` lässt sich beispielsweise mit dem `plot()` Befehl realisieren. Dies ist auch gleichzeitig einer der wichtigsten und am universellsten einsetzbaren Grafikbefehle. Die grafische Ausgabe kann anschließend über verschiedene weitere Befehle formatiert und den jeweiligen Layoutanforderungen angepasst werden. So kann über die Befehlskette

```
>> figure(1)
>>      hold on
>>      plot(t,x,'b','LineWidth',2)
>>      plot(t,y,'r','LineWidth',2)

>>      axis([0 10 -4 8])
>>      set(gca,'YTick',[-4:2:8])
>>      grid on

>>      title('Plot von zwei Funktionen','FontSize',14,'FontWeight','bold')
>>      xlabel('t / s','FontSize',14)
>>      ylabel('Funktionswert','FontSize',14)
>>      text(0.5,6.4,'Maximum von y',...
>>           'FontName','Verdana',...
>>           'FontAngle','Italic',...
>>           'BackgroundColor',[.8 .8 .8],...
>>           'EdgeColor',[1 0 0],...
>>           'FontSize',13)
>>      h_legend = legend('x_{(t)}','y_{(t)}');
>>      set(h_legend,'FontSize',14)
```

folgende grafische Ausgabe erzeugt werden:

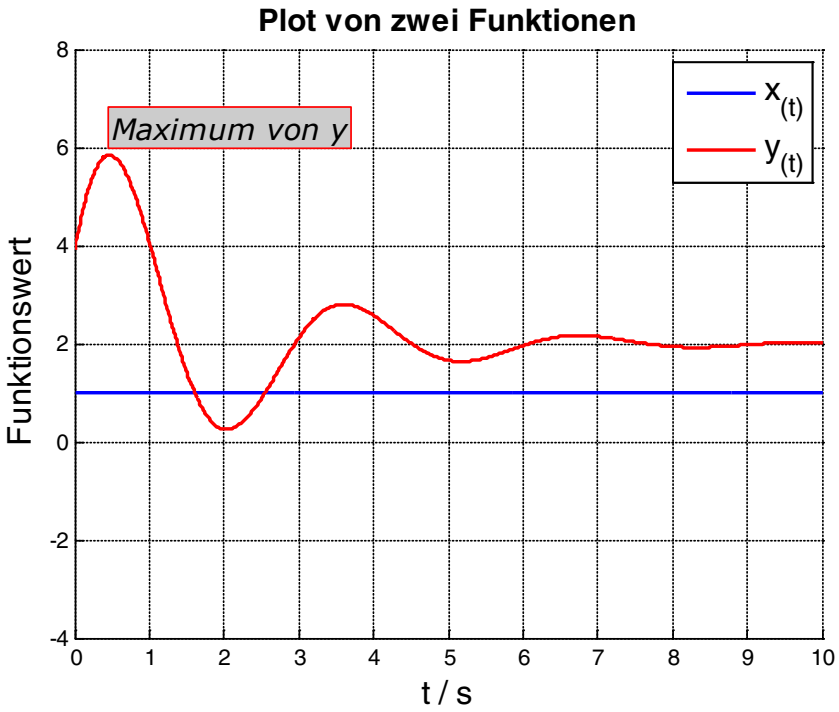


Abbildung 4: Beispiel zur plot() Funktion

Erklärung der einzelnen Codezeilen:

1. `Figure(1)`
   Erzeugt ein neues Fenster *Figure 1* zur Aufnahme eines Koordinatensystems. Besteht das Fenster mit der als Argument angegebenen Nummer bereits, dann wird dieses Fenster aktiv und alle nachfolgenden Befehle beziehen sich auf dieses Fenster und dessen Inhalt. Ein Figure ist quasi ein übergeordneter Container für alle grafischen Elemente.

2. `hold on`
   Der Befehl ist notwendig, wenn mehr als ein einziger Plot in ein Koordinatensystem (axis) gezeichnet werden soll. Ansonsten wird beim Aufruf des zweiten `plot()` Befehls der erste Plot gelöscht. Der Befehl kann mit `hold off` wieder aufgehoben werden.

3. `plot(t,x,'b','LineWidth',2)`
   Zeichnet den Vektor `x` (vertikale Achse) in Abhängigkeit von `t` (horizontale Achse) in der Farbe blau (`'b'`) und der Strichstärke 2. Wird nichts weiter angegeben, so werden alle Punkte des Vektors `x` als durchgezogene Linie dargestellt. Fügt man hinter den Farbbuchstaben `b` beispielsweise noch ein `o` oder `x`, so wird der Plot nicht als durchgezogenen Linie, sondern als einzelne Kreise oder X-Markierungen gezeichnet. Über `help plot` erhalten Sie Informationen darüber, welche weiteren Farben und Formatierungsoptionen ausgewählt werden können. Wichtig: Die beiden Vektoren `t` und `x` müssen stets die gleich Länge haben: Jedem Element von `t` wird das entsprechende Element von `x` zugeordnet und dieses Punktepaar im Koordinatensystem dargestellt.

4. `axis([0 10 -4 8])`
   Legt die minimalen und maximalen Grenzen der horizontalen und der vertikalen Achse fest. Das Argument ist in diesem Fall ein Vektor mit vier Elementen [x_{min} x_{max} y_{min} y_{max}].

5. `set(gca,'YTick',[-4:2:8])`
   Mit dem `set()` Befehl können die verschiedenen *Eigenschaften* (engl.: properties) beispielsweise des Figures oder des Koordinatensystems nachträglich geändert werden. Mit dem ersten Argument `gca` (get current axis) werden beispielsweise die Eigenschaften des aktuellen Koordinatensystems angesprochen. `gca` ist eine sogenannte *Handles Structure*, die alle Informationen über das entsprechende Objekt verwaltet, hierzu später mehr. Mit `YTick` wird die Gitternetzeinteilung der vertikalen Achse geändert. Minimaler Wert ist -4, maximaler Wert 8. Dazwischen soll die Achse im Abstand von 2 Einheiten unterteilt werden. Die `set()` Funktion wird immer nach folgender Syntax aufgerufen: `set(handle,'property1',value1,'property2',value2,…)`

6. `grid on`
   Fügt Gitternetzlinien hinzu. Mit `grid off` werden die Gitternetzlinien wieder entfernt.

7. `title('Plot von zwei Funktionen','FontSize',14,'FontWeight','bold')`
   fügt der Grafik eine Titelzeile mit der Schriftgröße 14 und im Fettdruck hinzu.

8. `xlabel('t [s]','FontSize',14) und ylabel('Funktionswert','FontSize',14)`
   beschriftet die horizontale und die vertikale Achse in der Schriftgröße 14.

9. `text(0.5,6.4,'Maximum von y',… )`
   fügt ein Textfeld an den Koordinaten x = 0.5 und y = 6.2 ein. Darüber hinaus wird der Text in der Schriftart Verdana (`FontName`), in kursiv (`FontAngle`), mit grauem Hintergrund (`BackgroundColor`), mit einer roten Umrandung (`EdgeColor`) und in der Schriftgröße 13 formatiert.

10. `h_legend = legend('x_{(t)}','y_{(t)}')`
    Erzeugt eine Legende. Dem ersten `plot()` Befehl wird die im ersten Argument stehende Zeichenkette zugeordnet, dem zweiten `plot()` Befehl die zweite Zeichenkette, etc. Durch den Unterstrich in den beiden Zeichenketten wird eingeleitet, dass die nachfolgenden Zeichen tiefer gestellt ausgegeben werden. Soll mehr als ein Zeichen tiefer gestellt werden, müssen diese in *geschweiften Klammern* stehen. Dies sind sogenannte TEX-Befehle, des Textsatzsystems TEX und können zur Formatierung der Beschriftung eingesetzt werden. Matlab erkennt in Zeichenketten einige grundlegenden TEX-Befehle, zum Beispiel die griechischen Buchstaben α und Ω über das TEX-Kommando `\alpha` bzw. `\Omega`.
    Darüber hinaus wird die zur Legende gehörende Handles Structure in der Variablen `h_legend` gespeichert.
11. `set(h_legend,'FontSize',14)`
    Verändert die Schriftgröße der Legende durch Zugriff über die zur Legende gehörende Handles Structure `h_legend`.

Jedes Grafische Objelt in einem figure hat ein spezifisches handle und prinzipiell können alle Eigenschaften dieser Grafikobjekte über die entsprechende Handles Structure und der `set()` Funktion geändert werden. Nur für die gebräuchlichsten Formatierungsoptionen gibt es Befehle wie z.B. `xlabel()` oder `axis()`. Alle Eigenschaften eines Grafik Objektes können analog mit der `get()` Funktion aufgelistet werden. So erhält man durch

```
>> get(gcf)
>> get(gca)
>> get(h_legend)
```

Einen Überblick über alle Eigenschaften des aktuellen Figures (`gcf`), der aktuellen Achse (`gca`) oder des zuvor angelegten Legende Objektes (`h_legend`). Figures können mit dem Befehl `close(num_fig)` geschlossen werden, wobei `num_fig` die Nummer des zu schließenden Figures ist. Alle geöffneten Figures können auch einfach mit `close all` geschlossen werden.

Ein Figure kann auch mehrere Koordinatensysteme enthalten. Die Anzahl und Lage der Achsen kann mit dem Befehl `subplot()` festgelegt werden. Mit der Befehlsfolge

```
>> figure(2)
>> set(gcf,'position',[50 50 800 300])

>>      subplot(121)
>>          plot(t,x,'b','LineWidth',2)

>>          grid on
>>      subplot(122)

>>          plot(t,y,'r','LineWidth',2)
>>          grid on
```

erhält man die folgende grafische Ausgabe:

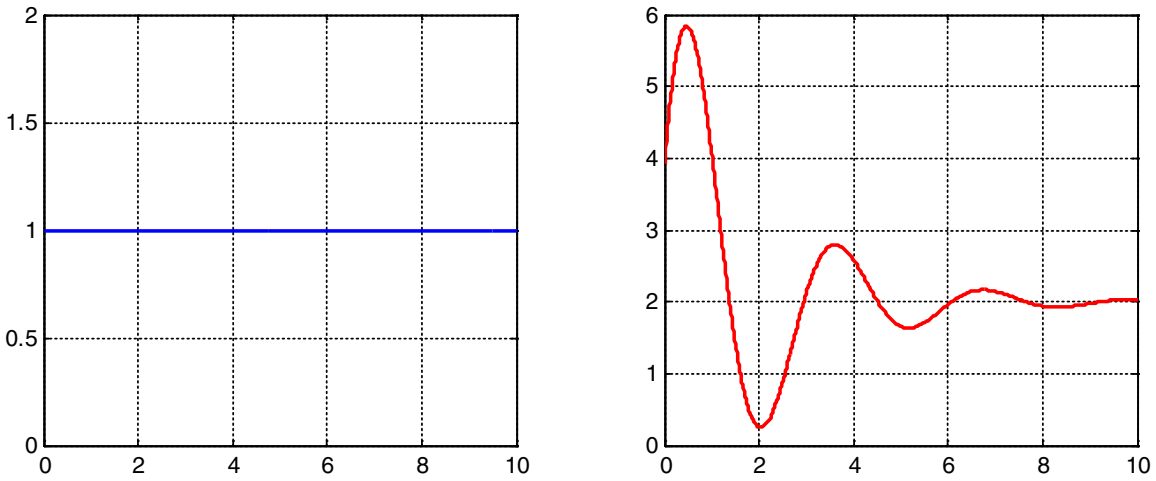


Abbildung 5: Subplots

Erklärung:

1. `set(gcf,'position',[50 50 1200 700])`
   Über die `position` Eigenschaft wird die Größe des Figures festgelegt. Das erste Element gibt die x-Koordinate der linken unteren Ecke des Figures auf dem Bildschirm an, das zweite deren y-Koordinate. Das dritte und vierte Element legen die Breite und die Höhe des Figures fest. Alle Angaben sind hierbei Bildschirmpixel.

2. `subplot(121)`
   mit `subplot()` kann die Anzahl der Achsen in einem Figure festgelegt werden. Subplots müssen in einem Figure immer matrizenförmig erstellt werden. Die erste 1 bedeutet das Figure beinhaltet *eine Zeile* mit Koordinatensystemen, die 2 bedeutet es gibt *zwei Spalten* mit Koordinatensystemen und die zweite 1 bedeutet, dass mit den folgenden Befehlen das erste Koordinatensystem angesprochen wird. Die Nummerierung der Achsen erfolgt dabei von links nach rechts und von oben nach unten.

Anmerkung: Neben dem einfachen `plot()` Befehl gibt es für logarithmisch skalierte Plots analog die Befehle `semilogx()` (x-Achse logarithmisch skaliert), `semilogy()` (y-Achse logarithmisch skaliert) und `loglog()` (Beide Achsen logarithmisch) skaliert. Für die Darstellung von Diagrammen in einem Polarkoordinatensystem gibt es den Befehl `polar()`.

Darüber hinaus gibt es folgende Alternativen zu `plot()`:

- `area()`
- `stairs()`
- `bar()`
- `stem()`

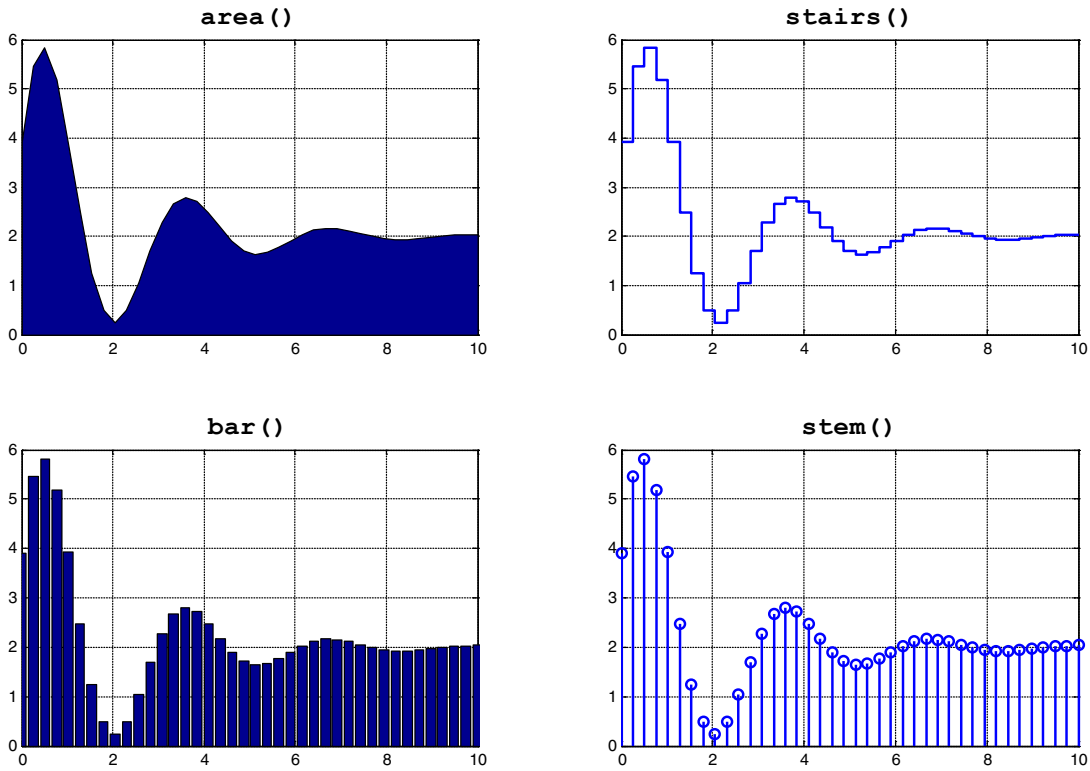


Abbildung 6: 2D Diagrammvarianten

2.6.3 3D Linienplots

Das Kommando `plot3()` ist das 3-dimensionale Pendant zu `plot()` und dient zur Erstellung von Liniengrafiken im 3-Dimensionalen. `plot(x,y,z)` unterscheidet sich von `plot(x,y)` nur durch das Auftreten der dritten Koordinate. Beispiel: $x_{(t)}$ und $y_{(t)}$ seien Funktionen in Abhängigkeit der Zeit t.

$$x_{(t)} = \left(1+t^2\right)\cdot \sin\left(t\right)$$
$$x_{(t)} = \left(1+t^2\right)\cdot \cos\left(t\right)$$

Es soll grafisch dargestellt werden, welchen Wert die Funktion $x_{(t)}$ und die Funktion $y_{(t)}$ in Abhängigkeit von t im Zeitraum von 0 bis 100 Sekunden haben. Hierzu wird $x_{(t)}$ auf der x-Achse, $y_{(t)}$ auf der y-Achse und der Zeitfortschritt t selbst auf der z-Achse (vertikale Achse) eines 3-dimensionalen Koordinatensystems aufgetragen.

```
>> t = 0:0.01:100;
>> x = (1+t.^2).*sin(t);
>> y = (1+t.^2).*cos(t);
```

```
>> plot3(x,y,t,'r','LineWidth',2)
>> grid on
>> title('plot3()','FontSize',20)
>> xlabel('x_{(t)}','FontSize',20)
>> ylabel('y_{(t)}','FontSize',20)
>> zlabel('t','FontSize',20)
```

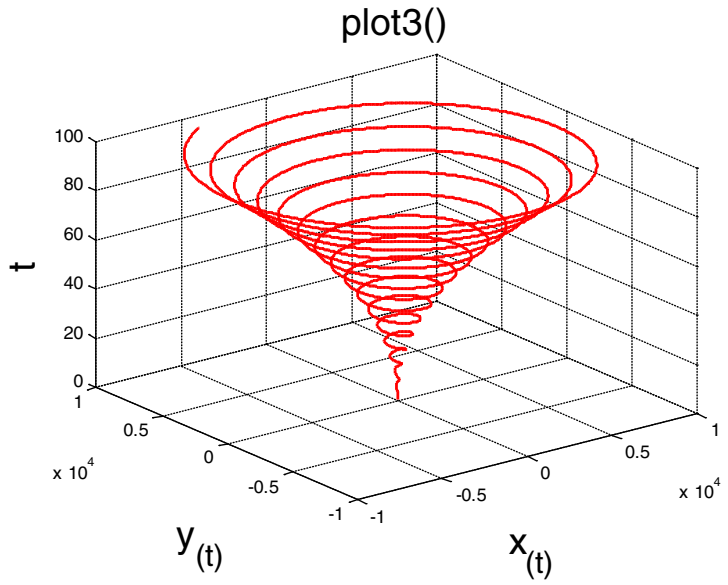


Abbildung 7: Beispiel zur plot3() Funktion

Bei 3D-Plots ist der *Rotate 3D* Button (→ Würfel mit kreisförmigem Pfeil) im *Figure Window* ein nützliches Hilfsmittel. Wird dieser aktiviert so kann man bei gedrückter Maustaste das Diagramm stufenlos in allen drei Raumkoordinaten rotieren. Einen genau definierten Betrachtungswinkel kann über den `view()` Befehl eingestellt werden. Mit `view(30,-30)` wird der Plot beispielsweise von "schräg unten" dargestellt.

2.6.4 3D Flächenplots

Neben Linienplots sind auch dreidimensionale *Flächenplots* möglich. Im Gegensatz zu Linienplots ist bei Flächenplots jeder x-Koordinate und jeder y-Koordinate genau eine z-Koordinate zugeordnet. Beispielhaft soll folgender funktionale Zusammenhang als Fläche im Raum dargestellt werden:

$$z_{(x,y)} = x^2 - y^2 \qquad \text{mit} \begin{array}{l} -10 \leq x \leq 10 \\ -10 \leq y \leq 10 \end{array}$$

Zur Umsetzung von Flächenplots in Matlab stellt man sich diesen Sachverhalt am Besten matrizenbasiert vor: In einer Matrix `X` sind die Werte der Variablen *x* und in einer Matrix `Y` die Werte der Variablen *y folgendermaßen* angeordnet:

X =

-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10
-10	-8	-6	-4	-2	0	2	4	6	8	10

Y =

-10	-10	-10	-10	-10	-10	-10	-10	-10	-10	-10
-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8
-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6
-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
0	0	0	0	0	0	0	0	0	0	0
2	2	2	2	2	2	2	2	2	2	2
4	4	4	4	4	4	4	4	4	4	4
6	6	6	6	6	6	6	6	6	6	6
8	8	8	8	8	8	8	8	8	8	8
10	10	10	10	10	10	10	10	10	10	10

- In `X` stehen also m identische Zeilenvektoren mit den Werten der Variablen x in jeder Zeile.
- In `Y` stehen hingegen n identische Spaltenvektoren mit Werten der Variablen y in jeder Spalte.

Zwei derartig aufgebaute Matrizen lassen sich einfach mit dem `meshgrid()` Befehl erstellen:

```
>> [X,Y] = meshgrid([x_start:x_ink:x_end],[y_start:y_ink:y_ende]);
```

`meshgrid()` erstellt also die beiden Matrizen `X` und `Y` mit zwei Vektoren als Argumente. Dies sind eben der Zeilenvektor der Matrix `X` und der Spaltenvektor der Matrix `Y`. Stellt man sich nun die Matrizen `X` und `Y` überlagert vor so kann man jedem Element einer dritten Matrix `Z` den entsprechenden Funktionswert zuordnen:

$$Z_{ij} = X_{ij}^2 - Y_{ij}^2$$

Wichtig ist also, dass die drei Matrizen `X`, `Y` und `Z` die gleiche Dimension mxn haben. In Matlab lässt sich dies folgenermaßen realisieren:

```
>> Z = X.^2-Y.^2;
```

Die Funktion lässt sich nun beispielsweise mit dem `surf()` Befehl (engl.: surface = Fläche) grafisch darstellen:

```
>> surf(X,Y,Z)
>> grid on
>> title('surf()','FontSize',20)
>> xlabel('x','FontSize',20)
>> ylabel('y','FontSize',20)
>> zlabel('z','FontSize',20)
```

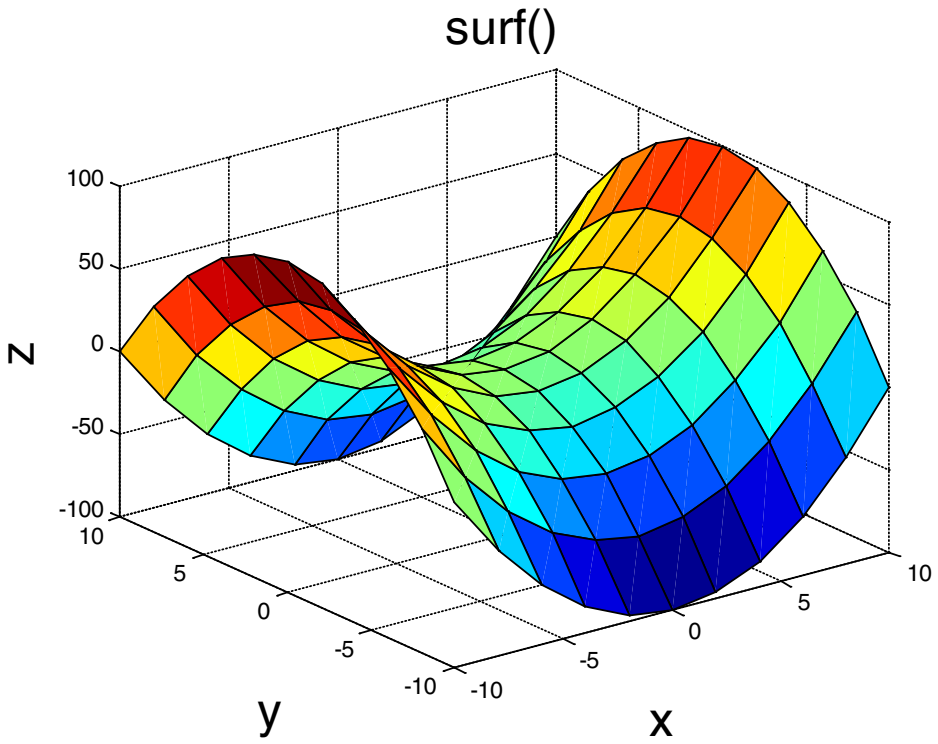


Abbildung 8: Beispiel zur surf() Funktion

- Mit der Funktion `mesh()` kann man anstatt einer ausgefüllten farbigen Fläche eine farbige *Gitterstruktur* mit weißer Fläche erzeugen. Hierbei kann mit `hidden off` eingestellt werden, dass darunterliegende Objekte nicht verdeckt werden, die Fläche der Gitterstruktur also durchsichtig wird.
- Mit `contour(X,Y,Z,n)` oder `contour3(X,Y,Z,n)` wird ein 2- bzw. 3-dimensionaler Plot erstellt, der `n` Linien gleicher Höhe (*Äquipotentiallinien)* auf der z-Achse zeichnet.
- Über die `alpha(x)` Funktion kann die Durchsichtigkeit des Flächenplots eingestellt werden. `x` muss eine Zahl zwischen 0 und 1 sein.
- Es gibt verschiedene Farbschemas für Flächenplots, die über den Befehl `colormap(farbschema)` eingestellt werden können. Voreingestellt ist das Farbschema `jet`. Der Befehl `help jet` verweist beispielsweise auf alternative Farbschemas
- Mit den Eigenschaften `shading interp` erhält man eine interpolierte Flächenstruktur, mit `lighting flat` wird eine Lichtquelle simuliert und mit `colorbar` wird ein Balken erzeugt, der die numerischen Werte zum gewählten Farbschema angibt.

Weitere Möglichkeiten zur Darstellung und zur Formatierung von dreidimensionalen Plots finden Sie in der Dokumentation. Beispiel:

```
>> figure(1)
>>    set(1,'position',[100 100 1200 800])
>>    colormap autumn;
>> subplot(221)
>>      mesh(X,Y,Z);
>>      hidden off
>> subplot(222)
>>      surf(X,Y,Z)
>>      shading  interp
>>      lighting flat
>>      colorbar
>> subplot(223)
>>      contour(X,Y,Z,20)
>> subplot(224)
>>      contour3(X,Y,Z,20)
```

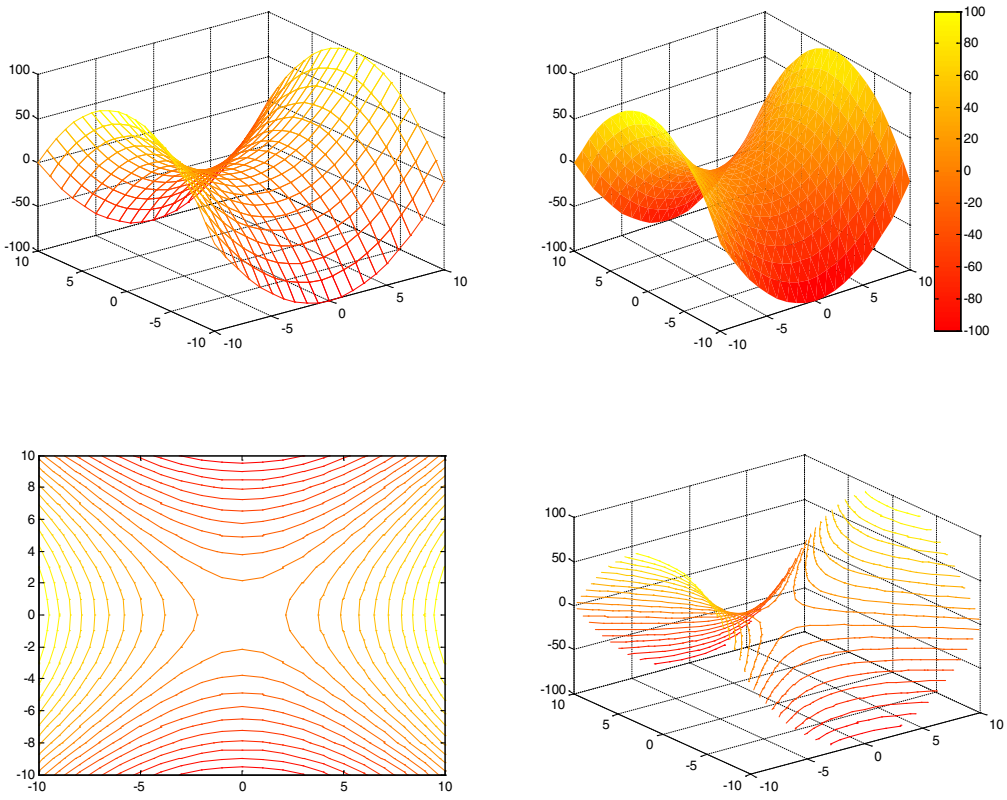


Abbildung 9: Varianten von Flächenplots

2.6.5 Animationen

Matlab bietet eine Vielzahl an weiteren Visualisierungsmöglichkeiten, beispielsweise zur Darstellung von *Vektorfeldern* und von *Volumenobjekten*, siehe Dokumentation. Darüber hinaus ist es auch möglich *Animationen* zu erzeugen und diese quasi als Film abzuspielen. Hierbei ist zunächst jedes Bild der Animation einzeln zu erstellen und in einer entsprechenden Variablen `frames` mit der Funktion `getframe()` abzuspeichern. Diese Aneinanderreihung von Bildern kann anschließend mit dem Befehl `movie()` zusammenhängend als Film abgespielt werden. Das Verfahren soll anhand eines Beispiels verdeutlicht werden. Die folgende Funktion beschreibt eine Fläche in Abhängigkeit von den Raumkoordinaten x und y sowie der Zeit t.

$$z_{(t,x,y)} = \frac{1}{\sqrt{1+4t^2}} \cdot e^{\left(-\frac{(x-3t)^2}{1+4t^2}\right)} \cdot e^{\left(-\frac{(y-3t)^2}{1+4t^2}\right)}$$

Die Fläche $z_{(x,y)}$ kann beispielsweise problemlos mit dem `surf()` Befehl zu einem bestimmten Zeitpunkt t dargestellt werden. Will man nun zusätzlich darstellen, wie sich diese über der Zeit verändert, so kann man einfach die Fläche zu den verschiedenen interessierenden Zeitpunkten grafisch darstellen, diese Grafiken aneinanderreihen und anschließend als Animation abspielen. Dieses Vorgehen ist in nachfolgender Funktion *animation.m* realisiert.

```
function [] = animation()
% --- 1. X-Y Ebene festlegen ---
    x = [-10:0.2:40];
    y = [-10:0.2:40];
    [X,Y] = meshgrid(x,y);
% --- Mathematische Funktion z(t,x,y) als Anonymous Function ---
    z = @(x,y,t)(1/sqrt(1+4*t^2)...
                    .*exp(-(x-3*t).^2/(1+4*t^2))...
                    .*exp(-(y-3*t).^2/(1+4*t^2)));
% --- Zeitschritte festlegen, an denen die Funktion ausgewertet wird ---
    t = 0:0.1:10;
 for(k=1:1:length(t))
    % --- Funktion zum Zeitpunkt t(k) auswerten ---
         Z = z(X,Y,t(k));
    % --- Grafik zum Zeitpunkt t(k) erstellen ---
         surf(X,Y,Z)
         axis([-10 40 -10 40 0 0.75])
         shading interp
         xlabel('x','fontSize',16)
         ylabel('y','fontSize',16)
         zlabel('z','fontSize',16)
    % --- Grafik in Variable "frames" abspeichern --
         frames(k) = getframe;
end
% --- Film n mal abspielen ---
    n = 2;
    movie(frames,n-1)
```

animation.m

3 Matlab als Programmiersprache

3.1 Editor

Bisher wurden alle Eingaben und Berechnungen über das Command Window vorgenommen. Matlab wurde dabei sozusagen lediglich als (sehr mächtiger) "Taschenrechner" missbraucht. Jeder Befehl wurde einzeln im Command Window eingetippt und nacheinander ausgeführt. Besser ist es jedoch, alle benötigten Anweisungen zunächst in einer Datei zu speichern. Dieser *Quellcode* kann dann quasi auf einen Schlag abgearbeitet werden. Somit kann dieses *Programm* auch einfach korrigiert, erweitert und zu einem späteren Zeitpunkt ausgeführt werden. Zur Erstellung, Speicherung und zum Debuggen von Quelltextdateien stellt Matlab einen *Editor* bereit. Der Editor kann über den Menübefehl *File* → *new* → *M-File* oder durch das unter dem Menü *File* liegende Symbol geöffnet *new M-File* werden. Hier können nun wie im Command Window alle Befehle eingegeben werden. Jede Anweisung sollte dabei in einer eigenen Zeile stehen. Nachdem alle entsprechenden Anweisungen eingegeben wurden, muss dieses Quelltextdokument über *File* → *Save as* als Datei gespeichert werden.

3.2 Skripte

Geben Sie im Editor folgendes ein:

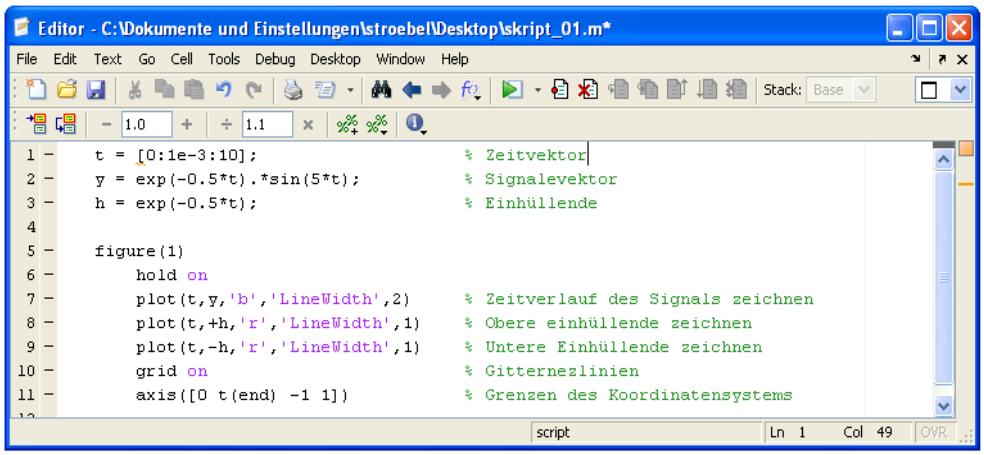


Abbildung 10: Matlab Skript

Speichern Sie die Datei nun beispielsweise unter dem Namen `script_01.m`. Gehen Sie zunächst wieder zum Command Window zurück und klicken Sie auf das Fenster *Current Directory*. Sie sehen, dass eine Datei namens `script_01.m` in ihrem gegenwärtigen Arbeitsverzeichnis erzeugt wurde. Diese enthält nun ihr Programm. Bei diesem sogenannten M-File (Dateiendung .m) handelt es sich um eine reine ASCII Textdatei, der Quelltext wurde also beim speichern nicht in Maschinensprache übersetzt und kann somit mit jedem beliebigen Texteditor angesehen und bearbeitet werden. Dateien, die einfach Aneinanderreihungen von Matlab Code beinhalten, werden auch als *Skripte* (engl.: scripts) bezeichnet. Die Befehle werden bei der Ausführung des Skripts einfach der Reihe nach abgearbeitet. Sie können das Skript ausführen indem Sie im Editor auf den grünen Pfeil in der Toolbar klicken. Es öffnet sich daraufhin ein neues Figure Window mit folgendem Inhalt:

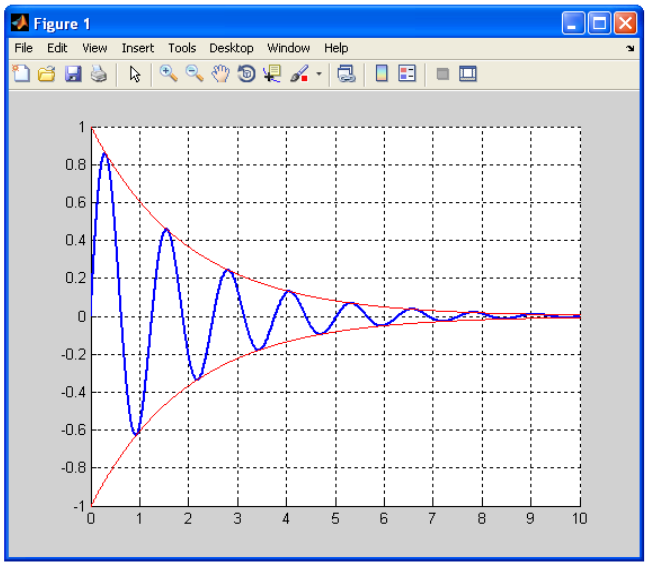


Abbildung 11: Gedämpfte Schwingung

Anmerkung: Im Gegensatz zu anderen Programmiersprachen wie zum Beispiel C, wird der Quelltext in Matlab nicht *kompiliert*, sondern *interpretiert*. Das heißt, dass das M-File erst zur Laufzeit, also bei der Programmausführung, ausgewertet (interpretiert) wird. Programmierfehler, beispielsweise die Nutzung einer nicht definierten Variablen, können daher von einem Compiler vorab nicht bemerkt werden und wirken sich daher erst zur Laufzeit aus. In Matlab erfolgt dann ein Programmabbruch mit einer entsprechenden Fehlermeldung im Command Window.

Die grün eingefärbten Textelemente im Skript sind *Kommentare*, mit denen Sie ihr Programm dokumentieren können. Kommentare werden in Matlab mit einem Prozentzeichen `%` eingeleitet. Alles was hinter dem Prozentzeichen in der gleichen Zeile folgt, wird bei der Ausführung ignoriert. Der Matlab Editor bietet die Funktionalität des *Sytnax Highlighting*, durch die bestimmte Schlüsselworte und Kontrollstrukturen automatisch in einer entsprechenden Farbe dargestellt werden.

Schließen Sie das Fenster *Figure 1* und kehren Sie zum Command Window zurück. Geben Sie hier den Befehl `script_01` ein und drücken Sie Enter. Sie sehen, dass sich das Fenster *Figure 1* wieder öffnet, ihr Skript also erneut aufgerufen wurde. Sie können ein Skript somit ebenfalls mit dem entsprechenden Dateinamen als Befehl aufrufen.

3.2.1 Debug Modus

Der Editor bietet unter den Menüpunkten *Edit*, *Text* und *Debug* eine Reihe an hilfreichen Tools, um die Erstellung und das Debuggen von Programmen zu unterstützen.

Wir beschränken uns hier auf das Setzen von *Breakpoints*. Befindet sich der Cursor im Editor in einer bestimmten Zeile, so kann mit der Taste *F12* ein Breakpoint gesetzt und auch wieder entfernt werden. Wird das Programm mit einem gesetzten Breakpoint ausgeführt, dann hält die Programmausführung in der Zeile des Breakpoints an. Matlab befindet sich sodann im *Debug Modus*. Alle Variableninhalte können dann beispielsweise über das Command Window analysiert werden, bevor das Programm fortgesetzt wird. Auf diese Weise können große Programme Stück für Stück kontrolliert und eventuelle Fehler im Quellcode ausgemacht werden.

Hat die Programmausführung an einem bestimmten Breakpoint angehalten, so kann mit dem Icon/Menübefehl *Continue* (Taste *F5*) fortgefahren werden, oder der Debug Modus mit dem Icon/Menübefehl *Exit Debug Mode* (Tasten *Umschalt+F5*) verlassen werden.

3.3 Flusskontrolle

Auch durch das Einbetten der Anweisungen in Skripte, arbeiten unsere "Programme" bisher lediglich einen Befehl nach dem anderen ab, eben in jener Reihenfolge wie Sie im Code aufeinanderfolgen. Der Sinn von Programmiersprachen ist es aber, den Ablauf eines Programms steuern und kontrollieren zu können. Matlab bietet die Möglichkeit, den sequentiellen Ablauf eines Programms durch Verzweigungen und Schleifen zu beeinflussen.

Die wichtigsten Mechanismen zur Flusskontrolle sind:

1. *if-else* Entscheidung
2. *switch-case* Fallunterscheidung
3. *for* Schleife
4. *while* Schleife
5. *try-catch* Ausnahmebehandlung

Erst durch diese Kontrollmechanismen wird Matlab zu einer vollwertigen Programmiersprache, mit der eigene komplexe Algorithmen und Programme erstellt werden können.

3.3.1 if-else Entscheidung

Die Anweisung `if` wertet einen *logischen Ausdruck* aus und verzweigt zu einer Gruppe von Anweisungen, sofern der Ausdruck logisch wahr ist. Ist der Ausdruck logisch falsch, kann mit dem Schlüsselwort `else` eine alternative Gruppe von Anweisungen ausgeführt werden, oder mit dem Schlüsselwort `elseif` eine erneute logische Abfrage durchgeführt werden. Der if-else Block wird mit dem Schlüsselwort `end` abgeschlossen. Die Syntax zur Eingabe zeigt nachfolgendes Beispiel.

Anmerkung: Es bietet sich an alle nachfolgenden Beispiele nicht im Command Window nachzuvollziehen, sondern entsprechende Skripte zu erstellen und diese anschließend auszuführen.

```
A = ones(2,5);
[m n] = size(A);

if(m==1 & n==1)
     typOfA = 'Skalar';
elseif(m==1 | n==1)
     typOfA = 'Vektor';
else
     typOfA = 'Matrix';
end
```

Die Klammern nach der `if` und `elseif` Abfrage sind optional. Besteht die Anweisungsgruppe nach der Abfrage wie hier jeweils aus nur einer Zeile, dann kann die Anweisung auch direkt hinter der logischen Abfrage in einer Zeile folgen.

```
A = ones(1,1);
[m n] = size(A);

if m==1 && n==1 typOfA = 'Skalar';
elseif m==1 || n==1  typOfA = 'Vektor';
else typOfA = 'Matrix';
end
```

Aus Gründen der Übersichtlichkeit und einer einfach nachvollziehbaren Programmstruktur, sollte der Code jedoch immer durch sinnvolle Klammerausdrücke, Zeilenumbrüche und Einrückungen lesbar gestaltet werden.

3.3.2 switch-case Fallunterscheidung

Mit der *switch-case* Anweisung kann je nach Wert einer Variablen eine bestimmte Gruppe von Anweisungen ausgeführt werden. Ein switch-case Block beginnt immer mit dem Schlüsselwort `switch`, gefolgt von der zu analysierenden Variablen. Jeder zu unterscheidende Fall wird dabei mit dem Schlüsselwort `case` eingeleitet. Nahezu jede switch-case Anweisung könnte auch über eine verschachtelte if-else Struktur realisiert werden, jedoch ist eine swicth-case Abfrage oftmals einfacher und entspricht der logischen Struktur eines Problems besser, wodurch auch die

Lesbarkeit des Programmcodes vereinfacht wird. Mit dem Schlüsselwort `otherwise` kann eine Anweisungsgruppe ausgeführt werden, wenn die Variable keinen der abgedeckten Fälle annimmt. Beispiel "Würfelspiel":

```
wuerfel = ceil(7*rand(1,1));
 switch wuerfel
     case 1
          s = 'gewürfelte Zahl: 1';
     case 2
          s = 'gewürfelte Zahl: 2';
     case 3
          s = 'gewürfelte Zahl: 3';
     case 4
          s = 'gewürfelte Zahl: 4';
     case 5
          s = 'gewürfelte Zahl: 5';
     case 6
          s = 'gewürfelte Zahl: 6';
     otherwise
          s = strcat('Sie haben eine "',num2str(wuerfel),...
          '" gewürfelt? Betrüger! Sie spielen wohl mit keinem 6-seitigen Würfel!');
 end
msgbox(s)
```

Erklärung: Zunächst wird eine zufällige natürliche Zahl zwischen *1* und *7* generiert, und in der Variablen `wuerfel` gespeichert. Anschließend wird über den switch-case Block die Zufallszahl ermittelt und eine entsprechende Zeichenkette in der Variablen `s` gespeichert. Diese Zeichenkette wird über eine sogenannte *message box* mit dem Befehl `msgbox()` am Bildschirm ausgegeben. Beispielsweise könnte erscheinen:

Abbildung 12: Message Box

3.3.3 for Schleife

Schleifen dienen dazu, eine Gruppe von Anweisungen mehrfach hintereinander auszuführen. Alle Anweisungen, die innerhalb eines Schleifenblocks stehen werden dabei wiederholt ausgeführt. Bei der *for* Schleife ist dabei vor Durchlaufen der Schleife bekannt, wie viele Male deren Inhalt ausgeführt wird. Die Anzahl an Durchläufen wird dabei über die *Inkrementanweisung* festgelegt. Beispiel:

```
start         = 0;
schrittweite  = 2;
ende          = 10;
inkrementanweisung = [start:schrittweite:ende];

durchlauefe = 0;

for(k = inkrementanweisung)
     durchlauefe = durchlauefe+1;
end
```

Die *Inkrementvariable* `k` wird dabei von Matlab automatisch bei jedem Schleifendurchlauf um `schrittweite` erhöht, bis diese den Wert `ende` erreicht. Werden diese Codezeilen ausgeführt, so nehmen die Variablen `k` und `durchlauefe` nach Abarbeiten der for Schleife die folgenden Werte an (Machen Sie sich dabei die Syntax und das Ergebnis der for Schleife klar):

```
k = 10
durchlauefe = 6
```

Mit dem Schlüsselwort `break` kann man eine Schleife vorzeitig abbrechen, wenn eine bestimmte Bedingung erfüllt ist.

```
for(k=1:1:100)
     x = k^2;
     if(x>270)
          break
     end
end
```

Die Variablen `k` und `x` nehmen nach Abarbeiten der Schleife die folgenden Werte an:

```
k = 17
x = 289
```

Die Schleife *terminiert* also vorzeitig. Sie wird nicht 100 mal abgearbeitet, wie es zunächst durch die Inkrementanweisung `1:1:100` festgelegt wurde, sondern nur 17 Mal, da nach 17 Schleifendurchläufen die Variable `x` erstmals einen Wert größer 270 aufweist.

3.3.4 while Schleife

Im Gegensatz zur for Schleife wird die *while* Schleife so lange ausgeführt, wie ein *logischer Ausdruck* wahr ist. Bei der while Schleife ist somit prinzipiell vorab nicht bekannt, wie oft diese durchlaufen wird, da der auszuwertende logische Ausdruck seinen Zustand bei einem beliebigen Schleifendurchlauf von *true* auf *false* ändern kann.

```
a = 4;
b = 100;
k = 0;
while(a<b)
     b = b/2;
     k = k+1;
end
```

Nach Durchlaufen der Schleife nehmen die Variablen die folgenden Werte an:

```
a = 4
b = 3.1250
k = 5
```

Die Schleife wird also fünfmal durchlaufen bevor die Variable `a` größer oder gleich `b` ist, also das Abbruchkriterium erreicht wird. Obwohl dies in diesem Fall einfach nachzuvollziehen ist, ist diese Tatsache im Programmcode selbst nicht explizit festgehalten, sonder ergibt sich quasi erst zur Laufzeit.

Der Programmierer hat dabei darauf zu achten, dass das gewählte Abbruchkriterium auch prinzipiell erreicht werden kann, ansonsten bleibt das Programm in der Schleife hängen. Es dürfte einfach nachzuvollziehen sein, dass die Schleife in obigem Beispiel früher oder später terminiert, wohingegen schon die kleine Veränderung

```
a = -4;
b = 100;
k = 0;
while(a<b)
      b = b/2;
      k = k+1;
end
```

zur Folge hat, dass das Abbruchkriterium `a<b` niemals logisch falsch werden kann, wodurch auch die Schleife niemals verlassen wird, das Programm hat sich umgangssprachlich *aufgehängt*. Sollte Ihr Programm einmal in eine solche *Endlosschleife* geraten sein, können Sie in Matlab dessen Ausführung mit *Strg+C* "gewaltsam" beenden.

Kann man vorab nicht genau festlegen, ob und wann ein gewünschtes Abbruchkriterium erreicht wird, so empfiehlt es sich der logischen Abfrage einen "Sicherheitsausdruck" hinzuzufügen. Beispielsweise endet die folgende Schleife mit Sicherheit spätestens nach 101 Durchläufen:

```
a = -4;
b = 100;
k = 0;
while(a<b && k<100)
     b = b/2;
     k = k+1;
end
```

Anmerkung: Bei Programmen, die gewisse Aufgaben in *Echtzeit* (innerhalb einer vorgegebenen Zeitspanne) abarbeiten müssen, sollte man mit der Verwendung von Schleifen und insbesondere von while Schleifen vorsichtig sein, da oftmals vorab nicht genau bestimmt werden kann wie viel Rechenzeit für die Abarbeitung der Schleife benötigt wird. Programme, die Aufgaben in Echtzeit ausführen müssen, sind beispielsweise häufig in der Steuerungs- und Regelungstechnik anzutreffen.

3.3.5 try-catch Ausnahmebehandlung

Die *try-catch Anweisung* dient zur *Ausnahmebehandlung* (engl.: exception handling), also zum Abfangen von Ausdrücken und Anweisungen, die zur Programmlaufzeit zu einem Fehler und somit zu einem Programmabsturz führen würden. Die fragwürdige Anweisung wird dabei zunächst *überprüft* (try) und wenn kein Fehler resultiert auch ausgeführt, andernfalls kann eine alternative Anweisung ausgeführt werden. Die in einem Fehler resultierende Anweisung wird also *abgefangen* (catch).

Beispiel: Die Multiplikation zweier Matrizen, deren innere Dimensionen nicht übereinstimmen, kann nicht ausgeführt werden. Eine entsprechende Anweisung führt zu einem Fehler und somit einem Programmabbruch.

```
>> A = [1 2;3 4];
>> B = [1 2];
>> A*B

??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Weiß man vorab nicht, ob die Dimensionen der Variablen `A` und `B` zueinander passen, so kann dies mit einem try-catch Block überprüft und gegebenenfalls abgefangen werden.

```
A = [1,2;3,4];
B = [1 2];

try A*B;
     C = A*B;
     error = 0;
catch
     C = [];
     error = 1;
end
```

Werden obige Anweisungen mit den Matrizen `A` und `B` ausgeführt, so kann die Matrixmultiplikation nicht durchgeführt werden, es resultiert jedoch keine Fehlermeldung und kein Programmabsturz, da dies durch die `try` Anweisung erkannt und daher lediglich die Anweisung im `catch` Block ausgeführt wird. Ob die Multiplikation in dem Beispiel erfolgreich war, könnte einfach anhand des Wertes der Variablen `error` ermittelt werden.

3.4 Funktionen

In Matlab können eigene *Funktionen* (engl.: functions) zur Erweiterung des bereits vorhandenen Befehlsumfangs erstellt werden. Funktionen werden sogenannte *Argumente* (Parameter, Eingabewerte) übergeben, mit welchen die Funktion einen oder mehrere *Algorithmen* ausführt und die *Funktionswerte* (Rückgabewerte) berechnet. Die Funktionswerte werden an den Funktionsaufrufer zurückgeliefert. Im Gegensatz zu einem *statischen* (unveränderlichen) Skript, sind Funktionen dynamische Elemente, da Funktionsaufrufe mit unterschiedlichen Eingabewerten auch unterschiedlich reagieren bzw. unterschiedliche Rückgabewerte liefern können.

Abbildung 13: Prinzip von Funktionen

Ein Aufrufer einer Funktion kann diese daher als Black-Box auffassen, was intern in der Funktion passiert interessiert ihn in der Regel nicht. Beispielsweise gehen wir davon aus, dass der von der Sinusfunktion `sin()` zurückgegebene Zahlenwert korrekt ist:

```
>> sin(pi/4)

ans = 0.7071
```

Wie die `sin()` Funktion schlussendlich zu diesem Ergebnis kommt, welcher konkrete Algorithmus also in der Funktion implementiert wurde, ist für das Ergebnis irrelevant.

Neben dem von Matlab "von Haus aus" zur Verfügung gestellten Funktionsvorrat, können auch eigene Funktionen erstellt werden, die eine spezielle Aufgabe erfüllen. Funktionen sind unter anderem das wichtigste Mittel, um Programme sinnvoll zu strukturieren und um wiederverwendbare und leicht wartbare Programme zu erstellen.

Funktionen werden in Matlab wie Skripte als M-Files mit der Dateiendung .m gespeichert. Die Deklaration eines *Function Files* erfolgt durch das Schlüsselwort `function` in der *ersten Zeile* des M-Files. Die allgemeine Syntax dieser ersten Zeile (*Funktionskopf*) in einer Funktionsdatei ist:

```
function [y1,y2,y3,…,yn] = funcname(x1,x2,x3,…,xm)
```

Dabei bezeichnen:

- `x1, x2, x3, … xm` die *Funktionsargumente*
- `y1, y2, y3, … yn` die *Rückgabewerte*
- `funcname` den *Funktionsnamen*

Nach der Deklarationszeile folgt der Programmcode, in dem definiert wird, was die Funktion leisten soll bzw. welcher Algorithmus abgearbeitet wird. Soll die Funktion einen oder mehrere Rückgabewerte liefern, so müssen insbesondere diese Rückgabewerte im Programmcode definiert werden. Beispielsweise gibt es in Matlab standardmäßig keinen Befehl für die *Sigmoidfunktion*:

$$y_{(x)} = \frac{1}{1+e^{a \cdot x}}$$

Diese kann aber ganz leicht dem Befehlsumfang durch eine eigene Funktion hinzugefügt werden. Hierzu müssen nur die folgenden Anweisungen in der Datei *sigmoid.m* gespeichert werden:

```
function [y] = sigmoid(a,x)
    y = 1./(1+exp(a*x));
```

Der *Dateiname* sollte immer identisch mit dem *Funktionsnamen* sein. Beachten Sie den Punkt vor dem Bruchstrich bei der Berechnung der Sigmoidfunktion. Dieser ist notwendig, wenn Sie für `x` als Argument Vektoren und Matrizen zulassen wollen. Die Funktion kann nun vom Command Window aus aufgerufen werden:

```
>> a = 1;
>> x = [-10:0.001:10];
>> y = sigmoid(a,x);
```

Der Funktionsverlauf $y_{(x)}$ im Intervall [-10, 10] kann nun auch einfach grafisch dargestellt werden:

```
>> plot(x,y)
```

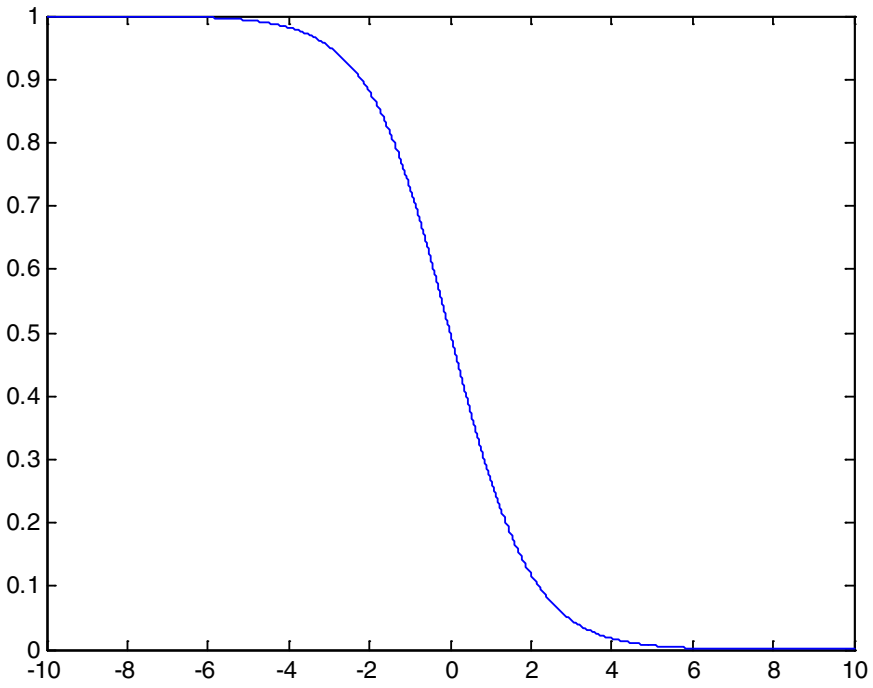


Abbildung 14: Sigmoid Funktion

Hat eine Funktion mehr als einen Rückgabewert, dann müssen diese beim Funktionsaufruf in eckige Klammern gesetzt werden.

Beispiel: Definition einer Funktion `komplex()` mit vier Rückgabewerten. Die Funktion liefert Realteil, Imaginärteil, Betrag und Phasenwinkel einer komplexen Zahl `Z`, welche der Funktion als Argument übergeben wird. `komplex()` nutzt dabei intern wiederum die von Matlab zur Verfügung gestellten Funktionen `real()`, `imag()`, `abs()` und `angle()`.

```
function [R,I,M,P] = komplex(Z)
    R = real(Z);
    I = imag(Z);
    M = abs(Z);
    P = angle(Z);
```

komplex.m

Der Funktionsaufruf kann nun vom Command Window aus erfolgen:

```
>> [real,imaginaer,betrag,phase] = komplex(3+5j)

real = 3
imaginaer = 5
betrag = 5.8310
phase = 1.0304
```

Beachten Sie, welche Variablen nun im Workspace liegen: Dies sind `real`, `imaginaer`, `betrag` und `phase`. Die im Function File *komplex.m* definierten Variablen `R`, `I`, `M`, `P`, und `Z` sind außerhalb der Funktion, also im Workspace, nicht sichtbar. Auf diese Variablen kann nur die Funktion selbst zugreifen. Andererseits kennt die Funktion `komplex` auch nur diejenigen Variablen, die *innerhalb* von ihr definiert wurden. Sie hat keinen Zugriff auf die im Workspace liegenden Variablen. Soll die Funktion mit im Workspace liegenden Variablen arbeiten, so müssen ihr diese über die Argumentliste explizit übergeben werden.

Anmerkung: Sollen die im Workspace liegende Variable auch für Funktionen sichtbar sein, ohne dass diese über die Argumentliste übergeben werden sollen, dann können diese Variablen beispielsweise sowohl im Workspace als auch in der Funktion selbst mit dem Schlüsselwort `global` als *globale Variable* deklariert werden.

3.4.1 Subfunctions

In einem Function File können neben der *Hauptfunktion* (engl.: mainfunction), die den gleichen Namen wie das entsprechende M-File trägt, auch beliebig viele *Unterfunktionen* (engl.: subfunctions) deklariert werden. Diese können einfach nach dem Ende der Hauptfunktion an diese angehängt werden. Schlüsselwort für Unterfunktionen ist ebenfalls wieder `function`. Unterfunktionen können nur von der Hauptfunktion aus aufgerufen werden. Beispiel:

```
function [A,V] = hohlzylinder(h,R,r)
% --- Hauptfunktion ----------------------
    A = (kreisflaeche(R)-kreisflaeche(r)); % Berechnung der Ringfläche
    V = h*A;                               % Berechnung des Volumens
function [Flaeche] = kreisflaeche(radius)
% --- Unterfunktion ----------------------
    Flaeche = pi*radius^2;
```

hohlzylinder.m

Die Variablen `h`, `R` und `r` sollen hierbei die Höhe sowie den Außen- und den Innenradius eines Hohlzylinders beschreiben. Die Funktion `hohlzylinder` berechnet unter Zuhilfenahme der Unterfunktion `kreisflaeche`, die Ringfläche `A` und das Volumen `V` des Hohlzylinders.

```
>> h = 2.13;
>> R = 4.772;
>> r = R/2;
>> [A,V] = hohlzylinder(h,R,r)

A = 53.6552
V = 114.2856
```

3.4.2 Nested Functions

Funktionen beginnen immer mit dem Schlüsselwort `function`. *Unterfunktionen* können dabei wie gesehen sequentiell in eine Matlab Funktionsdatei eingefügt werden. Eine Unterfunktion beginnt wieder mit dem Schlüsselwort `function`, an dieser Stelle endet gleichsam die darüberstehende *Hauptfunktion*. Unterfunktionen kennen jedoch nicht die Variablen aus der Hauptfunktion, diese müssten ihr gegebenenfalls explizit als Argumente übergeben werden. Das folgende Beispiel soll dies verdeutlichen:

```
function y = hauptfunktion(t)
% ---------------------------
A    = 5;     % Amplitude
w    = 3;     % Winkelgeschwindigkeit
phi  = pi/2;  % Phase

y = unterfunktion(t);

function s = unterfunktion(a)
% ---------------------------
     s = A*sin(w*a+phi);
```

hauptfunktion.m

Folgender Funktionsaufruf führt zu einer Fehlermeldung, da die Variablen `A`, `w`, und `phi` in `unterfunktion()` nicht bekannt sind:

```
>> y = hauptfunktion(pi)

??? Undefined function or variable 'A'.

Error in ==> hauptfunktion>unterfunktion at 11
              s = A*sin(w*a+phi);

Error in ==> hauptfunktion at 7
             y = unterfunktion(t);
```

Dieses Problem lässt sich mit sogenannten *Nested Functions* lösen (*engl.: nested = verschachtelt/eingebettet*). Die Unterfunktionen werden dabei *innerhalb der Hauptfunktion* definiert. Hierbei müssen dann aber sowohl Haupt- als auch Unterfunktion(en) mit dem Schlüsselwort `end` abgeschlossen werden. Die Unterfunktion kennt damit alle Variablen, die bis zum Beginn des Unterfunktionskopfes definiert wurden.

```
function y = hauptfunktion(t)
% --------------------------
A    = 5;     % Amplitude
w    = 3;     % Winkelgeschwindigkeit
phi  = pi/2;  % Phase

     function s = unterfunktion(a)
     % --------------------------
          s = A*sin(w*a+phi);
     end

y = unterfunktion(t);
end
```

hauptfunktion.m

Durch diese Funktionsdefinition ist der Aufruf nun erfolgreich:

```
>> y = hauptfunktion(pi)

y = -5
```

3.4.3 Function Handles

Ein *Function Handle* ist ein Matlab *Datentyp*, der alle Informationen enthält um eine bestehende Funktion aufzurufen. Ein Function Handle wird mittels

```
>> fhand = @funcname
```

erzeugt. Dabei steht `funcname` für den Namen der Funktion. Beispiel:

```
function [z] = myfun(x,y)
     z = x.^2-y.^2;
```

myfun.m

Zur Funktion *myfun* wird nun ein *Function Handle* erzeugt:

```
>> fhand = @myfun;
```

Im Workspace liegt nun die Variable `fhand` vom Datentyp *Function Handle*. Der Funktionsaufruf kann nun auch über dieses Function Handle erfolgen:

```
>> Z = fhand(5,4)
Z = 9
```

Vorteile:

- Eine Funktion kann in Form eines Function Handles einer anderen Funktion als Argument übergeben werden. In Matlab wird dieses Prinzip *Function Functions* genannt.
- Funktionsaufrufe erfolgen *effizienter*, insbesondere wenn diese wiederholt auftreten.
- *Namenskonflikte* zwischen gleichlautenden Funktionen können vermieden werden.
- Funktionsaufrufe können erfolgen, auch wenn die Funktion nicht im *aktuellen Verzeichnis* oder im *Matlab Pfad* liegt, da im Function Handle auch Pfadinformationen etc. abgespeichert werden.
- Function Handles können wie jeder andere Datentyp in Dateien *gespeichert* werden.

Auch wenn die Bedeutung von Function Handles an dieser Stelle vielleicht noch nicht ganz klar wird, so sei doch darauf hingewiesen, dass die Nutzung von Function Handles einen hohen Stellenwert in der (professionellen) Matlab Programmierung einnimmt.

3.4.4 Inline Functions

Funktionen mussten bisher immer in eigenen Dateien als *M-File* abgespeichert werden. Mit *Inline Functions* können (kleinere) Funktionen interaktiv programmiert werden, ohne dafür eine eigene Datei anlegen zu müssen. Das Schlüsselwort ist `inline` und die allgemeine Syntax:

```
funcname = inline('ausdruck','arg1','arg2',...)
```

Beispiel:

```
>> myfun_inline = inline('x.^2-y.^2','x','y')

myfun_inline =
      Inline function:
      myfun_inline(x,y) = x.^2-y.^2
```

Dies generiert das Objekt `myfun_inline`, vom Datentyp *Inline*, siehe Workspace. Die so definierte Funktion kann wie eine gewöhnliche Matlab Funktion verwendet werden.

```
>> Z = myfun_inline(5,4)
Z = 9
```

3.4.5 Anonymous Functions

Mit Matlab 7.0 wurde das Konzept der *anonymen Funktionen* eingeführt. Die Syntax ist:

```
fhandle = @(argliste) ausdruck
```

Das Rückgabeargument ist eine Function Handle. Anonyme Funktionen sind eine Alternative zu den *Inline Functions*. Hinter der Einführung der anonymen Funktionen verbirgt sich das Konzept, *Function Handles* deutlicher in den Vordergrund zu stellen. Beispiel:

```
>> myfun_anonym = @(x,y)(x.^2-y.^2)

myfun_anonym =
     @(x,y)(x.^2-y.^2)
```

Dies erzeugt das Function Handle `myfun_anonym`. Der Aufruf der Funktion erfolgt wieder über das *Function Handle*:

```
>> myfun_anonym(5,4)
ans = 9
```

3.4.6 Persistente Variablen

Der Wert von in Funktionen definierten Variablen wird nach Abarbeiten der Funktion nicht gespeichert, der Wert ist bei einem erneuten Funktionsaufruf nicht mehr bekannt. In Funktionen definierte Variablen sind also *lokale Variablen*. Der Wert von einer als *persistent* definierten Variablen ist dagegen auch bei einem erneuten Aufruf der Funktion noch bekannt. Persistente Variablen sind trotzdem lokal, das heißt sie sind nur der entsprechenden Funktion bekannt. Das Schlüsselwort `persistent` muss dabei vor jeglicher Benutzung in der Funktion auftreten. Deswegen können persistente Variablen keine Rückgabewerte von Funktionen sein. Beim ersten Aufruf der Funktion werden persistente Variablen als leere Matrix `[]` angelegt. Im folgenden Beispiel wird in der persistenten Variablen `n` gespeichert, wie viele Male die Funktion *wieoft.m* bereits aufgerufen wurde. Das Ergebnis wird in `aufrufe` zurückgegeben.

```
function [aufrufe] = wieoft()

    persistent n;       % Persistente Variable deklarieren
                        % ohne jedoch einen konkreten Wert zuzuweisen
    if(isempty(n))      % Prüfung, ob n "leer" ist (n = [])
        n = 0;          % Falls ja den Wert 0 zuweisen.
    end
    n = n+1;           % Bei jedem Funktionsaufruf n inkrementieren

    aufrufe = n;        % -> persistente Var. können keine Rückgabewerte sein
```

wieoft.m

3.4.7 Globale Variablen

Sollen mehrere Funktionen auf dieselben Variablen zugreifen können, so genügt es diese mit `global` als globale Variablen zu definieren. Dies gilt auch für die Kommunikation einer Funktion mit dem Workspace. Globale Variablen sollten jedoch aufgrund der höheren Programmiersicherheit (Namenskonflikte, Überschreiben von Variablen, Zugriffsrechte, …) sparsam eingesetzt, beziehungsweise nach Möglichkeit vermieden werden.

3.4.8 Datentypkontrolle

Ein guter Programmierstil erfordert es meist, die einer Funktion übergebenen Argumente auf *Plausibilität* zu prüfen. Neben applikationsspezifischen Prüfungen, beispielsweise dass ein Argument einen bestimmten Wertebereich nicht über- oder unterschreiten darf, ist auch die Kontrolle des übergebenen *Datentyps* an sich oftmals notwendig. Wenn beispielsweise eine Zeichenkette als Argument übergeben wird, die Funktion aber einen numerischen Wert erwartet, so führen anschließende Berechnungen in der Regel zu falschen Ergebnissen oder gar zu einem Fehler und damit zu einem Programmabsturz. Matlab bietet einige nützliche Funktionen um eine *Datentypkontrolle* vorzunehmen. Die Funktionen liefern eine logische 1 falls die Überprüfung erfolgreich verläuft, ansonsten eine logische 0.

Funktion	Beschreibung
isnumeric(x)	Prüft, ob x ein numerischer Wert ist (z.B. double, uint8)
isfloat(x)	Prüft, ob x eine Gleitkommazahl ist (z.B. double)
isinteger(x)	Prüft, ob x eine Ganzzahl ist (z.B. uint8)
islogical(x)	Prüft, ob x eine boolesche Variable (logical) ist
ischar(x)	Prüft, ob x ein Zeichen oder eine Zeichenkette ist
isstruct(x)	Prüft, ob x ein Zeichen oder eine Struktur-Variable ist
iscell(x)	Prüft, ob x ein Zeichen oder eine Zell-Variable ist
isa(x,class)	Prüft, ob x ein Objekt vom Datentyp class ist.
class(x)	Liefert die Klasse (Datentyp) des Objektes x zurück
isempty(x)	Prüft, ob x ein leeres Array ist (x = [])
isinf(x)	Prüft, ob x den Wert unendlich (inf) hat
isfinite(x)	Prüft, ob x den Wert unendlich (inf) hat
isnan(x)	Prüft, ob x den Wert Not a Number (NaN) hat (z.B. 0/0)

Tabelle 9: Datentypkontrolle

Beispiel: Die Funktion *multiplication* prüft ob beide Argumente `A` und `B` numerisch sind. Falls ja wird in `C` das Produkt beider Zahlen gespeichert, ansonsten wird `C` der Wert `NaN` (*Not a Number*) zugewiesen.

```
function [C] = multiplication(A,B)
    if(isnumeric(A) & isnumeric(B))
        C = A*B;
    else
        C = NaN;
    end
```

multiplication.m

Datentypkontrollen und/oder weitere Mechanismen zur Sicherstellung eines korrekten Funktionsaufrufes sollten zumindest dann berücksichtigt werden, wenn Außenstehende mit einer von Ihnen erstellten Funktion arbeiten oder Sie diese im industriellen Umfeld professionell einsetzen wollen.

3.5 Datentypen

3.5.1 Strukturvariablen

Bisher wurden alle Daten in Vektoren oder Matrizen gespeichert. Der Zugriff auf einzelne Elemente erfolgte dabei mittels der runden Klammer und der Indizierung des entsprechenden Elementes. Beispiel:

```
>> A = [17.5, 45+4j; -2.008, pi]

A = 17.5000              45.0000 + 4.0000i
    -2.0080               3.1416
```

Auf das Element mit dem Index $i = 2$ und $j = 2$ konnte folgendermaßen zugegriffen werden:

```
>> el22 = A(2,2)
el22 = 3.1416
```

Dabei kann in den verschiedenen Elementen jeweils immer nur *derselbe Datentyp* stehen, hier also ein numerischer Wert vom Datentyp double. Ebenso kann man in einer Matrix auch einfach verschiedene Zeichen (engl.: character) ablegen. Der Befehl

```
>> S = ['a','b';'c','d']

S = ab
    cd
```

erzeugt ebenfalls eine 2x2-Matrix, wobei auf jedes Element (Zeichen) mittels runder Klammern zugegriffen werden kann. Es ist jedoch nicht möglich in einer Variablen numerische Daten und Zeichenketten gemischt zu speichern, da die Variable als Ganzes einen bestimmten Datentyp haben muss. Problematisch wird es auch schon, wenn die Zeichenketten in den verschiedenen Elementen unterschiedlich lang sind:

```
>> S = ['ich','bin';'eine','Matrix']

??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

Zur Speicherung unterschiedlicher Datentypen in einer Variablen gibt es in Matlab die sogenannten *Containervariablen*. Dies sind die sogenannten *Strukturvariablen* (*structs*) und die *Zellvariablen* (*cells*). Strukturvariablen werden einfach generiert, indem man hinter den Variablennamen einen Punkt setzt und darauffolgen den entsprechenden *Feldnamen* angibt. Beispiel:

```
Person1.Name = 'Mueller';
Person1.Vorname = 'Karlchen';
Person1.Geburtstag = [14,07,1981];
Person1.verheiratet = false;
```

Dies generiert eine Strukturvariable mit vier Feldern `Name`, `Vorname`, `Geburtstag` und `verheiratet`. Jedes Feld ist quasi eine eigene Variable, die aber innerhalb der Variablen `Person1` strukturiert/organisiert sind. `Name` und `Vorname` sind dabei Zeichenketten (Datentyp string), `Geburtstag` ist ein numerisches Array der Größe 1x3 (Datentyp double) und `verheiratet` eine boolesche Variable (Datentyp logical). Alle Felder sind in der gleichen Strukturvariablen `Person1` gespeichert. Auf die einzelnen Felder kann nun wieder mit dem *Punktoperator* zugegriffen werden.

- Zugriff auf ein Strukturfeld:

  ```
  >> sein_Vorname = Person1.Vorname
  sein_Vorname = Karlchen
  ```

- Manipulation eines Strukturfeldes:

  ```
  >> Person1.verheiratet = true

  Person1 =  Name: 'Mueller'
             Vorname: 'Karlchen'
          Geburtstag: [14 7 1981]
         verheiratet: 1
  ```

Anmerkung: Wie der Name schon sagt, dienen Strukturvariablen dazu, logisch zusammengehörende Daten gemeinsam zu speichern und zu verwalten. Strukturvariablen haben eine besondere Bedeutung bei der sogenannten *Objekt Orientierten Programmierung* (OOP), z.B. in den Programmiersprachen C++, C# oder Java. Auch Matlab bietet die Möglichkeit der Objekt Orientierten Programmierung. Auf dieses spezielle Themengebiet wird im Rahmen dieses Einführungskurses jedoch nicht näher eingegangen.

Auch mit dem `struct()` Befehl können Strukturvariablen erzeugt werden:

```
>> rechteck = struct('a',2,'b',4,'A',[])

rechteck =   a: 2
             b: 4
             A: []
```

Dies erzeugt eine Strukturvariable `rechteck` mit den Feldern `a` und `b` für die Seitenlängen, und dem zunächst unbekannten Flächeninhalt `A` eines Rechtecks. Dieser kann einfach mit den Feldern `a` und `b` berechnet werden:

```
>> rechteck.A = rechteck.a*rechteck.b

rechteck = a: 2
           b: 4
           A: 8
```

Natürlich können einer Struktur auch nachträglich noch Felder hinzugefügt werden, z.B. für den Rechteckumfang das Feld `U`:

```
>> rechteck.U = 2*(rechteck.a+rechteck.b)

rechteck = a: 2
           b: 4
           A: 8
           U: 12
```

Oder Felder mit dem Befehl `rmfield()` entfernt werden:

```
>> rmfield(rechteck,'A')

ans = a: 2
      b: 4
      U: 12
```

Die Variable `rechteck` ist eine Strukturvariable der Dimension 1x1, siehe Workspace. Es können aber auch *Strukturvektoren* und *Strukturmatrizen* definiert werden. Beispiel:

```
produkt(1) = struct('Artikel','Motor','Kosten',1279.67,'Preis',2199.00)
produkt(2) = struct('Artikel','Getriebe','Kosten',567.72,'Preis',999.00)
produkt(3) = struct('Artikel','Steuerung','Kosten',977.12,'Preis',1199.00)
```

erzeugt einen 1x3 Strukturvektor, in welchem die einzelnen Produkte quasi geordnet wie in einer Tabelle nebeneinander stehen:

produkt(1)		produkt(2)		produkt(3)	
Feld	**Wert**	**Feld**	**Wert**	**Feld**	**Wert**
Artikel	Motor	Artikel	Getriebe	Artikel	Steuerung
Kosten	1279.67	Kosten	567.72	Kosten	999.00
Preis	2199.00	Preis	999.00	Preis	1199.00

Tabelle 10: Strukturvektor produkt

Es können nun beispielweise die Gesamtkosten der drei Produkte leicht über die `sum()` Funktion ermittelt werden:

```
>> GKosten = sum([produkt.Kosten])
GKosten = 2.8245e+003
```

3.5.2 Zellvariablen

Zellvariablen, *Zellarrays* oder einfach *cells* besitzen den Aufbau einer gewöhnlichen Matrix, können jedoch ähnlich wie Strukturen in den verschiedenen Elementen unterschiedliche Datentypen unterschiedlicher Größe aufnehmen. Zellvariablen werden im Gegensatz zu herkömmlichen Matrizen oder Arrays über *geschweifte Klammern* (engl.: curly brackets) definiert und die einzelnen Zellelemente werden ebenfalls über geschweifte Klammern angesprochen. Beispiel:

```
>> C{1,1} = 'ich';
>> C{1,2} = 'bin';
>> C{2,1} = 'eine';
>> C{2,2} = 'Zellvariable';
>> C

C = 'ich'       'bin'
    'eine'      'Zellvariable'

>> Zellelement = C{2,2}

Zellelement = Zellvariable
```

Die Struktur einer Zellvariablen kann zur Veranschaulichung mit dem `cellplot()` Befehl visualisiert werden.

```
>> Z = {1,[true,false;true,true];['Zeichenkette'],[0:1:10]'};
>> cellplot(Z)
```

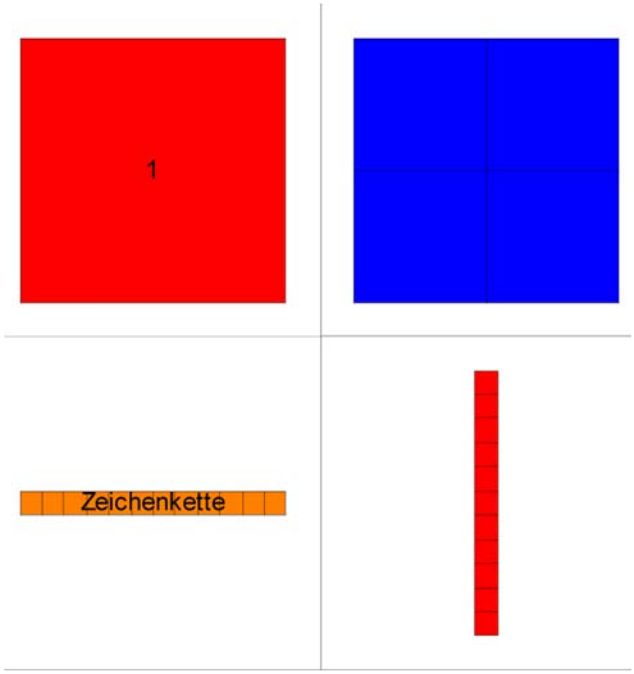


Abbildung 15: Darstellung von Zellstrukturen mittels cellplot()

Wichtig: Die verschiedenen Zellelemente können aus unterschiedlichen Datentypen bestehen, wie in obigem Beispiel die Zellvariable `z`. Struktur- und Zellvariablen können Funktionen problemlos als Argumente übergeben werden oder von diesen als Rückgabewerte zurückgeliefert werden. Struktur- und Zellvariablen können in den jeweils anderen Datentyp mit dem Befehl `struct2cell()` bzw. `cell2struct()` gewandelt werden.

3.5.3 Elementare numerische Datentypen

Bisher wurden alle numerischen Daten (Zahlen) im double Format gespeichert. Matlab speichert automatisch alle Zahlen als double, sofern nicht explizit ein anderer Datentyp angegeben wird. Die Nutzung anderer Datentypen dient unter Matlab hauptsächlich dazu Speicherplatz zu sparen. Dies ist für viele Programme und Anwendungen jedoch nicht notwendig, da moderne Rechner mit "ausreichend" Speicherplatz ausgerüstet sind. Der Vollständigkeit halber aber sei erwähnt, dass Matlab die folgenden elementaren numerischen Datentypen bereitstellt.

Datentyp	Beschreibung
single()	4 Byte (32 Bit) Gleitkommazahl
double()	8 Byte (64 Bit) Gleitkommazahl
int8()	8 Bit Integer (Ganzzahl) mit Vorzeichen
int16()	16 Bit Integer mit Vorzeichen
int32()	32 Bit Integer mit Vorzeichen
int64()	64 Bit Integer mit Vorzeichen
uint8()	8 Bit Integer ohne Vorzeichen
uint16()	16 Bit Integer ohne Vorzeichen
uint32()	32 Bit Integer ohne Vorzeichen
uint64()	64 Bit Integer ohne Vorzeichen
logical()	Boolsche Variable (1 Bit)

Tabelle 11: Elementare numerische Datentypen

Beispiel:

```
>> x = uint8(200)
x = 200
```

Hierbei wird die Zahl 200 in der Variablen x gespeichert, wodurch aber nur 1 Byte (= 8 Bit) Speicherplatz belegt wird, anstatt der 8 Byte, die für den double Datentyp benötigt werden. Allerdings können in diesem Datentyp natürlich auch nur ganze Zahlen ohne Vorzeichen (*unsigned integer*) im Bereich 0…255 gespeichert werden. Ansonsten wird der kleinstmögliche oder der größtmögliche Wert zurückgegeben bzw. gerundet.

```
>> x = uint8(278)
x = 255
>> x = uint8(-2)
x = 0
>> x = uint8(17.4)
x = 17
```

3.5.4 Zahlensysteme

Mit folgenden Befehlen kann eine Zahl in verschiedenen Zahlensystemen dargestellt werden:

Funktion	Bedeutung
dec2bin()	Dezimaldarstellung in Binärdarstellung umwandeln
dec2hex()	Dezimaldarstellung in Hexadezimaldarstellung umwandeln
bin2dec()	Binärdarstellung in Dezimaldarstellung umwandeln
hex2dec()	Hexadezimaldarstellung in Dezimaldarstellung umwandeln
dec2base()	Dezimaldarstellung in Darstellung mit beliebiger Basis umwandeln

Tabelle 12: Zahlensysteme

Beispiel:

```
>> H = dec2hex(2^16-1)
H = FFFF
>> B = dec2bin(9)
B = 1001
```

Das Ergebnis liegt in der Variablen `H` bzw. `B` anschließend als *Zeichenkette* vor.

3.5.5 Umwandlung von Zeichenketten

Oftmals liegen Messdaten als ASCII codierte Zeichenketten in Textdateien vor. Ein wichtiger Befehl ist daher die Umwandlung von Zeichenketten in numerische Datentypen und umgekehrt:

```
>> num = str2num('23.4')
num = 23.4000
>> str = num2str(23.4)
str = 23.4
```

3.6 File Handling

In Matlab gibt es viele Möglichkeiten Daten aus verschiedensten Dateiformaten einzulesen (*read*) und zu speichern (*write*). Beispielhaft sollen die drei folgenden Dateiformate behandelt werden:

1. *.mat Datei*
   MAT-Dateien sind das Standardformat in Matlab, um Daten (effizient) zu speichern.

2. *.txt Datei*
   ASCII codierte Textdateien

3. *.xls Datei*
   Microsoft Excel Sheets

3.6.1 MAT Dateien

In MAT-Dateien können jegliche in Matlab definierte Variable gespeichert und zu einem späteren Zeitpunkt wieder geladen werden. MAT-Files sind binäre Dateien, die Daten können in der Regel von anderen Programmen nicht verarbeitet oder mittels eines Texteditors betrachtet werden. Allerdings können die Daten effizient gespeichert werden. Über den Befehl `save filename` werden alle Variablen im Workspace in der Datei *filename.mat* gespeichert. Mittels `save filename A B C` werden lediglich die Variablen `A`, `B` und `C` in der Datei *filename.mat* gespeichert. Beispiel:

```
>> M = rand(10,5);
>> save MAT_file
```

Hierdurch wird die Zufallsmatrix `M` in der Datei *MAT_file.mat* gespeichert.

Über den Befehl `load filename` werden alle in der Datei *filename.mat* gespeicherten Variablen geladen und im Workspace angelegt. Mit dem Befehl:

```
>> S = load('filename.mat','A','B','C')
```

werden lediglich die Variablen `A`, `B`, und `C` aus der Datei *filename.mat* geladen und als Felder in der Strukturvariablen `S` angelegt. Es ist zu beachten, dass alle Argumente als Zeichenketten, also in einfachen Hochkommas, übergeben werden müssen.

3.6.2 ASCII Dateien

ASCII formatierte Textdateien sind ein beliebtes Format um beispielsweise Messdatenreihen abzuspeichern, da die Daten praktisch von jedem beliebigen Texteditor, Textverarbeitungs- oder Tabellenkalkulationsprogramm gelesen werden können. Messdatenreihen zeichnen sich vielmals dadurch aus, dass zu einem Zeitpunkt ein oder mehrere zusammengehörende Messwerte anfallen, beispielsweise von verschiedenen Sensoren. Diese werden in ASCII Files durch ein *Trennzeichen* (engl.: *delimiter*) getrennt in eine Zeile geschrieben, worauf ein Zeilenumbruch folgt und in die nächste Zeile die Messwerte des nächsten Messzeitpunkts geschrieben werden. Somit wird automatisch eine Matrixstruktur der Dimension $m \times n$ erzeugt, wobei die Spaltenanzahl n für die Anzahl der verschiedenen Messsignale und die Zeilenanzahl m für die Anzahl der Messzeitpunkte steht. Natürlich können aber auch beliebige andere Daten, die in Matrizenform gespeichert werden können, in ASCII Files abgelegt werden. Mit den Befehlen `dlmread()` und `dlmwrite()` können ASCII Files gelesen und geschrieben werden. Die Syntax für das Schreiben von ASCII Dateien ist:

```
>> dlmwrite('filename',M,'delimiter').
```

- `filename`
  Ein String, der den Dateinamen mit Endung (z.B.: .txt) angibt.
- `M`
  Die zu speichernde Variable/Matrix.
- `delimiter`
  Das Zeichen das zur Trennung der Werte verwendet wird. Voreingestellt ist das Komma, daher ist das Argument `delimiter` optional. Wird das Komma als Trennzeichen verwednet, so spricht man dann auch von CSV-Dateien (comma separated values) und es wird die Dateiendung .csv verwendet. Gebräuchlich sind beispielsweise auch das Semikolon (`;`) oder der Tabulator (`\t`) als Trennzeichen. Die Angabe des delimiters erfolgt als Zeichenkette in einfachen Hochkommas.

Optional kann zusätzlich das Argument `'-append'` übergeben werden. Dann wird eine bestehende Datei nicht überschrieben, sondern die Matrix `M` an den bestehenden Datensatz angehängt. Beispiel:

```
>> M = rand(10,5);
>> dlmwrite('ASCII_file.txt',M,',')
```

Sehen Sie sich das so entstanden ASCII File mittels eines Texteditors/Textverarbeitungsprogramms an oder versuchen Sie das File in einem Tabellenkalkulationsprogramm zu öffnen. Die Syntax für das Lesen von ASCII Dateien ist:

```
>> M = dlmread('filename','delimiter')
```

In der Matrix `M` steht dann der Inhalt des ASCII Files.

3.6.3 Excel Dateien

Analog zum Lesen und Schreiben von ASCII Dateien, gibt es auch die Möglichkeit Microsoft Excel Dateien zu lesen und in diese zu schreiben. Die Syntax zum Schreiben von Excel Dateien lautet:

```
>> [status, message] = xlswrite('filename',M,sheet,'range')
```

Die Argumente für den Dateinamen `filename` und die zu speichernde Variable `M` sind zwingend, die anderen optional. `sheet` kann entweder den Namen des entsprechenden Tabellenblattes als Zeichenkette, oder den Index des Tabellenblattes als Zahl enthalten. Mit `range` kann der Bereich des Tabellenblattes angegeben werden, der beschrieben werden soll. Im Rückgabewert `success` steht eine 1, wenn das speichern erfolgreich war, ansonsten eine 0. In `message` stehen eventuelle Warn- und Fehlermeldungen. Beispiel:

```
>> M = rand(10,5);
>> [succes,message] =  xlswrite('EXCEL_file.xls',M,2,'A3:E12');
```

Sehen Sie sich die dadurch angelegte Datei *EXCEL_file.xls* mittels Microsoft Excel an.

Die Syntax zum Lesen von Excel-Dateien ist:

```
>> [numeric,txt,raw] = xlsread('filename',sheet,'range').
```

In Excel kann man verschiedene Zellen unterschiedlich formatieren, beispielsweise als Zahlen (Numerischer Wert) oder als Text (Zeichenketten). Daher werden von der Funktion `xlsread()` mehrere Variablen zurückgegeben: In `numeric` werden alle als Zahlen formatierten Zellen, in `txt` alle als Text formatierten Zellen und in `raw` alle unformatierten Zelleninhalte abgelegt. Achtung: Seit Microsoft Office 2007 wird in Excel das neue Format *.xlsx* unterstützt. Daher gibt es in neueren Matlab Versionen auch die Befehle `xlsxread()` und `xslxwrite()`.

3.6.4 Import Wizard

Matlab bietet mit dem sogenannten *Import Wizard* eine komfortable grafische Schnittstelle zum Importieren von Daten. Sie können über *File → Import* Data die gewünschten Daten importieren. Wenn Sie die drei Dateien aus den vorherigen Beispielen angelegt haben, können Sie diese nun in einem Auswahldialog sehen. Wählen Sie nun beispielsweise durch Doppelklick die *Datei MAT_file.mat* aus, so öffnet sich automatisch der Import Wizard (Hier können Sie je nach Dateiformat verschieden Einstellungen vornehmen). Drücken Sie auf *Finish*, so werden die entsprechenden Variablen im Workspace angelegt.

3.7 Auswertung von Zeichenketten

Mit Hilfe des Befehls `eval()` können in Form von Zeichenketten definierte mathematische Ausdrücke numerisch ausgewertet werden. Beispiel:

```
>> eval('x=5')
x = 5
```

Hierdurch wird im Workspace die Variable `x` mit dem Wert 5 angelegt. Das folgende Beispiel berechnet zu jedem Element im Vektor `a` das mit 2 multiplizierte Quadrat.

```
>> a = [0 1 2 3 4 5];
>> b = eval('2*a.^2')

b = 0      2      8     18     32     50
```

Was ist nun der Vorteil, eine Anweisung als Zeichenkette über `eval()` auszuwerten, gegenüber der direkten numerischen Zuweisung wie in folgendem Beispiel?

```
>> b = 2*a.^2
b =  0      2      8     18     32     50
```

Ein großer Vorteil ist, dass Zeichenketten an Funktionen als Argumente übergeben werden können, numerische Zuweisungen jedoch nicht. `eval()` kann beispielsweise auch geschickt dazu verwendet werden, um Variablen mit unterschiedlcihen Namen zu erstellen:

```
for(k=1:1:5)
     str = strcat('var_',num2str(k),' = ','k^2');
     eval(str);
end
```

Mit dem Befehl `strcat()` können mehrere Zeichenketten aneinandergereiht werden. Mit der obigen Schleife und der `eval()` Funktion werden also fünf unterschiedliche Variablen *automatisiert* erzeugt:

```
var_1 =  1
var_2 =  4
var_3 =  9
var_4 = 16
var_5 = 25
```

4 Ausgewählte mathematische Anwendungen

In den vorangegangenen Kapiteln wurden die wichtigsten Methoden und Programmierkonzepte von Matlab vorgestellt. Matlab stellt darüber hinaus eine Fülle an spezifischen Funktionen bereit, die teilweise recht komplexe Algorithmen implementieren, für den Anwender jedoch als simple Funktionsaufrufe zu verwenden sind. Gerade diese umfangreiche und ausgereifte Funktionsbibliothek macht Matlab zu einem hochprofessionellen Engineering Werkzeug, mit welchem auch komplexe mathematische Problemstellungen zielstrebig, intuitiv und effizient gelöst werden können. Durch diese Funktionsbibliothek eignet sich Matlab zur Lösung mathematischer Problemstellungen oftmals besser als native Hochsprachen wie beispielsweise C/C++, in denen die benötigten Algorithmen erst aufwändig von Hand programmiert werden müssen, sofern keine entsprechende Bibliothek verfügbar ist. Im folgenden Kapitel "Ausgewählte mathematische Anwendungen" soll ein kleiner Teil dieser Funktionsbibliothek anhand ausgewählter Beispiele und Disziplinen vorgestellt werden.

4.1 Polynome

Polynome werden in Matlab durch einen Zeilenvektor repräsentiert, wobei die Vektorelemente die Koeffizienten des Polynoms in absteigender Reihenfolge darstellen. Das Polynom

$$p_{(x)} = x^4 + 7x^3 - 5x + 3$$

wird beispielsweise durch den Zeilenvektor `p = [1 7 0 -5 3]` repräsentiert. Ein Polynom n-ter Ordnung besitzt n reelle oder komplexe, einfache oder mehrfache Nullstellen. Die Funktion `roots()` erlaubt die numerische Berechnung dieser Nullstellen. Das Ergebnis wird als Spaltenvektor zurück gegeben:

```
>> p = [1 7 0 -5 3];
>> r = roots(p)

r =
  -6.8853
  -1.1411
   0.5132 + 0.3441i
   0.5132 - 0.3441i
```

Mit der Funktion `poly()` lassen sich diese wieder zum Polynomvektor wandeln:

```
>> p = poly(r)
p = 1.0000    7.0000    0.0000   -5.0000    3.0000
```

Die Auswertung des Polynoms `p` an der Stelle `x` erlaubt die Funktion `polyval(p,x)`:

```
>> y = polyval(p,-1)
y = 2.0000
```

Somit lassen sich Polynome sehr einfach mit dem `plot()` Befehl zeichnen:

```
>> p = [1 7 0 -5 3];
>> x = [-7:0.001:3];
>> plot(x,polyval(p,x))
>> grid on
```

Liefert das Diagramm:

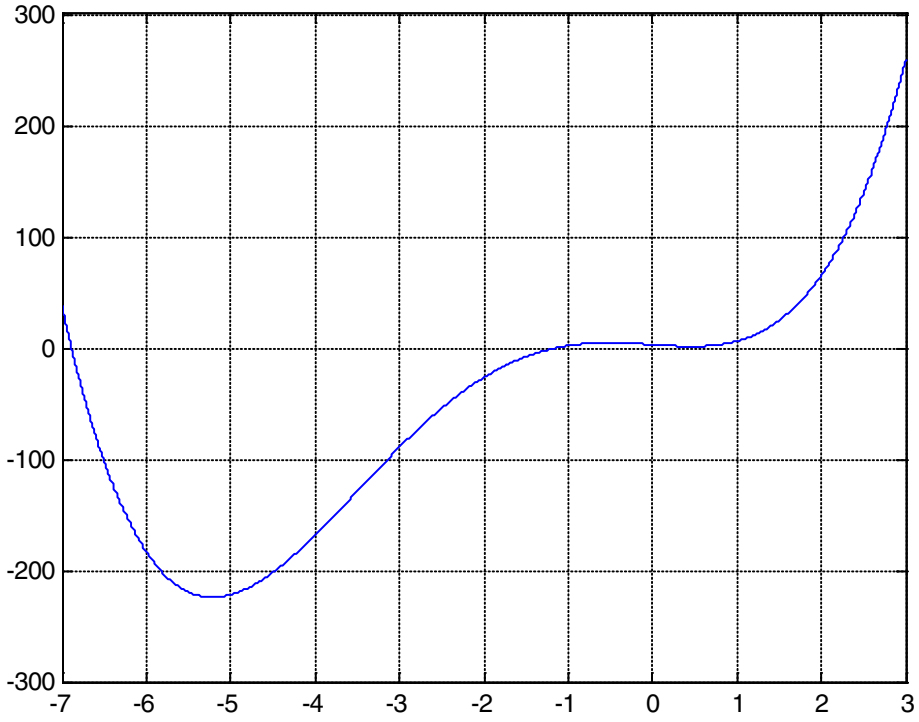


Abbildung 16: Polynomfunktion

4.1.1 Symbolische Ableitung und Integration

Symbolische Differentiation und Integration von Polynomen erlauben die Kommandos `polyder(p)` und `polyint(p,c)`, wobei `p` das Polynom und `c` die Integrationskonstante beschreibt.

```
>> p = [1 7 0 -5 3];
>> p_der = polyder(p)
p_der = 4     21      0     -5

>> p_int = polyint(p,5)
p_int = 1.7500     0    -2.5000     3.0000     5.0000
```

Als Ergebnis werden die Koeffizienten der jeweiligen Funktion zurückgegeben.

4.1.2 Polynomaddition und Polynommultiplikation

Zwei Polynome $p_{1(x)}$ und $p_{2(x)}$ können einfach miteinander addiert werden, indem die entsprechenden Polynomvektoren miteinander addiert werden. Hierbei ist darauf zu achten, dass die beiden Vektoren die gleiche Länge haben, gegebenenfalls müssen diese entsprechend mit Nullen aufgefüllt werden. Beispiel:

$$p_{1(x)} = 5x^5 + 3x^2 - 8x + 2$$
$$p_{2(x)} = 4x^3 - 6x^2 - 5$$

Erzeugen der beiden Polynomvektoren:

```
p1 = [5 0 0 3 -8 2];
p2 = [4 -6 0 -5];
```

Die direkte Addition dieser beiden Vektoren führt zu einem Fehler:

```
>> p1+p2

??? Error using ==> plus
Matrix dimensions must agree.
```

Man muss diese zuerst auf die gleiche Länge bringen:

```
>> h = zeros(1,6);
>> h(3:6) = p2;
>> p2 = h;
>> p1+p2

ans = 5      0      4     -3     -8     -3
```

Mit der Funktion `conv()` (engl. convolution = Faltung) kann man Polynome miteinander multiplizieren und mit dem Befehl `deconv()` dividieren. Beispiel:

$$p_{1(x)} = 2x + 3$$
$$p_{2(x)} = 4x + 5$$
$$p_{(x)} = p_{1(x)} \cdot p_{2(x)} = (2x+3)\cdot(4x+5) = 8x^2 + 22x + 15$$

In Matlab erfolgt die Multiplikation also folgendermaßen:

```
>> p1 = [2 3];
>> p2 = [4 5];
>> p = conv(p1,p2)

p = 8     22     15
```

Und die Polynomdivision:

```
>> [q,r] = deconv(p1,p2)
q = 0.5000
r = 0     0.5000
```

Dabei repräsentiert der Vektor `q` das Ergebnispolynom und `r` das Restpolynom.

4.1.3 Partialbruchzerlegung

Eine Partialbruchzerlegung einer gebrochen rationale Funktion der Form

$$f_{(x)} = \frac{f_{Z(x)}}{f_{N(x)}} = \frac{b_m \cdot x^m + b_{m-1} \cdot x^{m-1} + \cdots + b_1 \cdot x + b_0}{a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x + a_0}$$

mit Zählerpolynom $f_{z(x)}$ und Nennerpolynom $f_{N(x)}$ kann mit dem Befehls `resiude()` durchgeführt werden, beziehungsweise konvertiert dieser Befehl zwischen einer Polynom-Quotienten-Darstellung und einer Pol-Residuum-Darstellung. Beispiel: Die gebrochen rationale Funktion zweiter Ordnung

$$f_{(x)} = \frac{f_{Z(x)}}{f_{N(x)}} = \frac{x}{x^2 - 1}$$

kann über Polynomdivision als Summe zweier gebrochen rationaler Funktionen erster Ordnung dargestellt werden:

$$f_{(x)} = \frac{f_{Z(x)}}{f_{N(x)}} = \frac{r_1}{x - p_1} + \frac{r_2}{x - p_2} + k_{(x)}$$

Dabei sind r_1 und r_2 die sogenannten Residuen und p_1 und p_2 die Polstellen. k ist der direkte Term, der im Allgemeinen bei der Partialbruchzerlegung entstehen kann und gegebenenfalls berücksichtigt werden muss. Die Werte für r, p und k können in Matlab folgendermaßen ermittelt werden:

```
>> fz = [1 0];
>> fn = [1 0 -1];
>> [r,p,k] = residue(fz,fn)

r = 0.5000
    0.5000

p =  -1
      1

k = []
```

Die Lösung lautet also:

$$\frac{x}{x^2 - 1} = \frac{0.5}{x + 1} + \frac{0.5}{x - 1}$$

Umgekehrt lässt sich diese Pol-Residuum-Darstellung wieder in die Polynom-Quotienten-Darstellung mit dem Befehl `residue()` wandeln:

```
>> [FZ,FN] = residue(r,p,k)
FZ = 1      0
FN = 1      0     -1
```

4.1.4 Polynominterpolation und Polynomapproximation

In der numerischen Mathematik versteht man unter *Polynominterpolation* die Suche nach einem Polynom, welches *exakt* durch m vorgegebene Punkte verläuft. Dieses Polynom wird *Interpolationspolynom* genannt und man sagt, es interpoliere die gegebenen Punkte. Sind m Punkte gegeben, so kann ein Polynom der Ordnung $n = m-1$ gefunden werden, welches *exakt* durch die vorgegebenen Punkte verläuft.

Anmerkung: Ein Polynom n-ter Ordnung hat $m = n+1$ Koeffizienten, also ebenso viele Freiheitsgrade wie vorgegebene Punkte.

Ist die Anzahl der gegebenen Punkte größer als die Ordnung des Polynoms, dann verläuft das Polynom im Allgemeinen nicht durch die gegebenen Punkte. Es kann jedoch mittels entsprechender numerischer Verfahren ein Polynom gefunden werden, welches diese Punkte bestmöglich annähert. In diesem Fall spricht man von *Polynomapproximation*. Das folgende Beispiel visualisiert Polynominterpolation und Polynomapproximation anhand der gleichen fünf Datenpunkte.

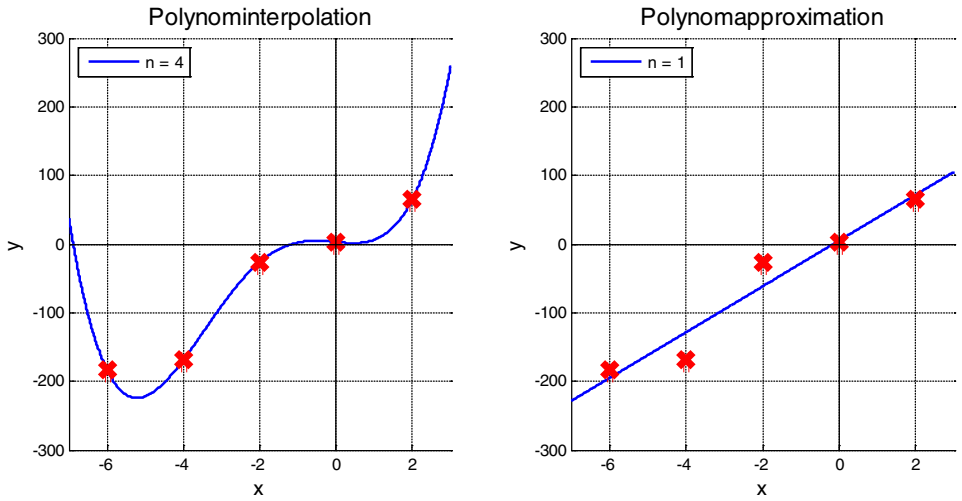


Abbildung 17: Polynominterpolation und Polynomapproximation

Es ist hierbei darauf zu achten, dass die Anzahl der Nullstellen eines Polynoms immer gleich dessen Ordnung ist. Polynome höherer Ordnung neigen also prinzipiell mehr zum *Schwingen*, siehe obige Abbildung. Die einzelnen Punkte einer Messreihe können durch Polynome höherer Ordnung somit zwar besser angenähert werden, jedoch muss darauf geachtet werden, dass hierdurch in der Regel nur schlecht auf eine naturwissenschaftliche oder technische Gesetzmäßigkeit geschlossen werden kann, die gewählte *Modellfunktion* also nicht *generalisiert*.

Zur Polynominterpolation bzw. Polynomapproximation stellt Matlab den Befehl `polyfit()` zur Verfügung. Der Aufruf erfolgt mit drei Argumenten:

```
p = polyfit(x,y,n)
```

`x` und `y` sind die Vektoren gleicher Länge, in denen die Koordinaten der zu interpolierenden/approximierenden Punkte gespeichert sind. `n` ist die Ordnung des Polynoms. `p` ist das Interpolations- bzw. Approximationspolynom und hat *n+1* Elemente.

Beispiel zur Polynomapproximation:
Es soll eine Ausgleichsgerade zu folgender Messreihe gefunden werden.

x	0	1	2	3	4	5	6	7	8	9	10
y	1	5	11	17	19	23	29	32	35	42	45

Tabelle 13: Messreihe

Eine (Ausgleichs)Gerade ist ein Polynom erster Ordnung:

$$p_{(x)} = a_1 \cdot x + a_0$$

Die Polynomapproximation erfolgt mit `polyfit()`:

```
>> x = [0 1 2 3 4 5 6 7 8 9 10];
>> y = [1 5 11 17 19 23 29 32 35 42 45]*1e-3;
>> p = polyfit(x,y,1)

p = 0.0044     0.0017
```

Für die Koeffizienten der Ausgleichgeraden gilt somit:

$$a_1 = 0.0044$$
$$a_0 = 0.0017$$

4.2 Numerische Differentiation und Integration

Im letzten Kapitel haben wir ein einfaches und effektives Verfahren zur Integration und Differentiation von Polynomen kennengelernt. Polynome sind analytische, also formelmäßige, Ausdrücke. Im Folgenden sollen Verfahren vorgestellt werden, wie numerische Vektoren differenziert und integriert werden können. Numerische Integration und Differentiation ist wichtig, da Messdaten oftmals lediglich in Form zahlenbehafteter Datenreihen verfügbar sind.

4.2.1 Bestimmte Numerische Integration

Matlab bietet mehrere Funktionen zur numerischen Integration. Es wird hier lediglich das Verfahren mittels Trapezintegration und dem Befehl `trapz()` vorgestellt. Bei der Trapezintegration werden zwei benachbarte Punkte $y_{(x1)}$ und $y_{(x2)}$ linear miteinander verbunden, so dass der hierdurch aufgespannte Flächeninhalt durch die Fläche eines Trapezes gegeben ist. Es ist zu beachten, dass dies lediglich eine Näherung an das Integral unter der Originalfunktion ist, man beachte hierzu die rote Differenzfläche zwischen der Originalfunktion und der grünen Trapeznäherung.

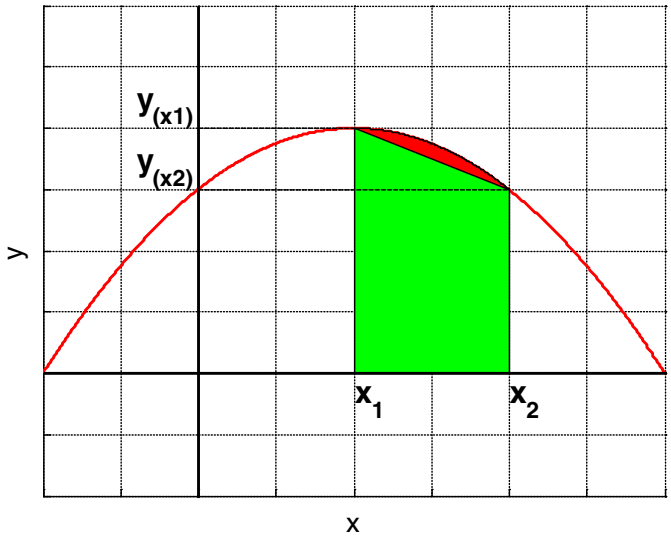


Abbildung 18: Zur Trapezintegration

Da die rote Originalfunktion ja unbekannt und nur die Datenpunkte $y_{(x1)}$ und $y_{(x2)}$ bekannt sind, ist die Annahme eines linearen Verlaufes zulässig, bzw. die einfachste und naheliegendste. `trapz()` integriert prinzipiell längs eines Vektors `y`. Die allgemeine Syntax ist:

```
A = trapz(x,y,dim)
```

`x` und `dim` sind optionale Argumente. `trapz(y)` integriert längs des Vektors `y` mit Schrittweite 1, während mit `trapz(x,y)` die Integrationsschrittweite durch `X` bestimmt ist. `x` muss dabei nicht zwingend äquidistant sein. Ist `y` ein Array so wird spaltenweise ausgewertet, mit `dim` kann allerdings auch die Dimension festgelegt werden, entlang derer integriert wird (`dim = 1` → zeilenweise Integration, `dim = 2` → spaltenweise Integration).

Beispiel: Es soll der Flächeninhalt A der Funktion $y_{(x)} = x^2$ in den Grenzen $x_1 = 0$ und $x_2 = 3$ ermittelt werden. Zunächst die analytische Berechnung "von Hand":

$$A = \int_{x=0}^{x=3} x^2\,dx = \left[\frac{1}{3}x^3\right]_{x=0}^{x=3} = \frac{1}{3}3^3 - \frac{1}{3}0^3 = 9$$

Zur Berechnung in Matlab wird die Funktion $y_{(x)} = x^2$ zunächst als diskreter Zahlenvektor in den Grenzen 0 bis 3 mit 1000 Elementen angelegt:

```
>> x = linspace(0,3,1000);
>> y = x.^2;
```

Die Funktion $y_{(x)}$ liegt nun also in Form von 1000 numerischen Zahlenwerten vor. Die Flächenberechnung erfolgt über:

```
>> A = trapz(x,y)
A = 9.0000
```

Dies ist *näherungsweise* der numerisch bestimmte Flächeninhalt unter der Funktion $y_{(x)}$. Wenn die Ergebnisvariable mit mehr Nachkommastellen dargestellt wird, sieht man, dass bei der numerischen Integration aufgrund des diskreten numerischen Verfahrens der Trapezintegration Fehler gemacht werden:

```
>> format long
>> A

A = 9.000004509013518
```

Untersuchen Sie wie sich der Näherungsfehler bei der Trapezintegration verhält, wenn der Vektor `x` anstatt 1000 nur 10 bzw. 10000 Elemente hat, die Schrittweite also größer oder kleiner gewählt wird.

4.2.2 Unbestimmte Numerische Integration

Soll hingegen nicht ein konkreter Flächeninhalt, sondern das *unbestimmte Integral*, also die *Stammfunktion* $Y_{(x)}$, als Vektor berechnet werden, so ist hierzu der Befehl `cumtrapz()` (kummulative Trapezintegration) zu verwenden.

```
>> Y = cumtrapz(x,y)
```

`x` und `y` definieren dabei die Funktion $y_{(x)}$ als numerische Vektoren und müssen daher die gleiche Länge haben. Die Rückgabevariable `Y` ist dann ebenfalls wieder ein Vektor der gleichen Länge.

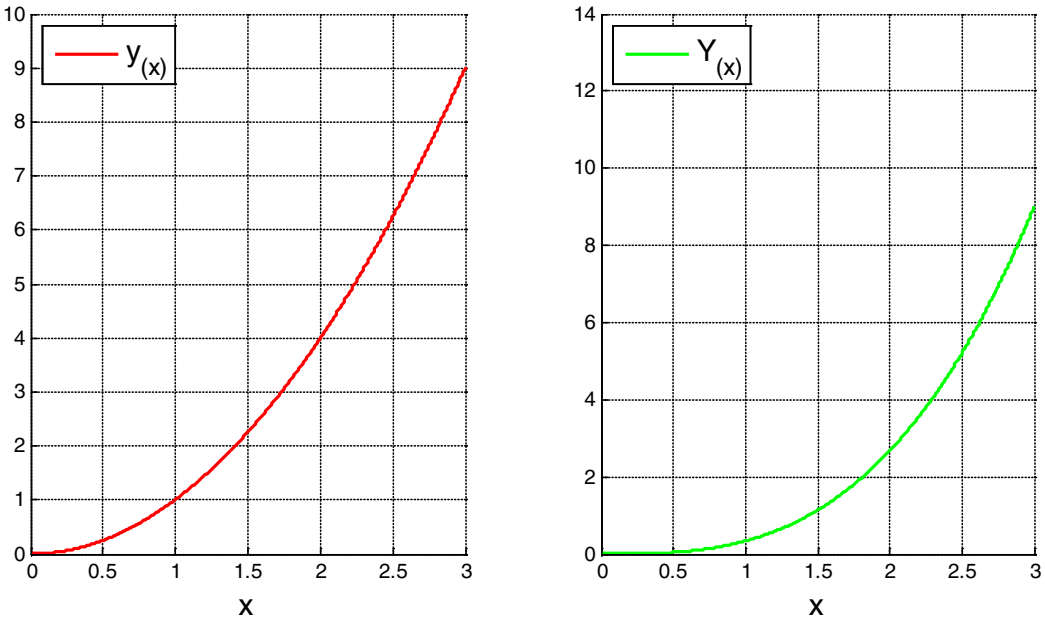


Abbildung 19: Numerische Integration

Anmerkung: `cumtrapz()` berücksichtigt keine Integrationskonstante, diese muss gegebenenfalls nachträglich der Ergebnisvariablen `Y` additiv hinzugefügt werden.

4.2.3 Numerische Differentiation

Zu numerischen Differentiation bietet Matlab den Befehl `gradient()`. Betrachten wir nochmals obige Beispielfunktion $y_{(x)} = x^2$ dann gilt:

$$y_{(x)} = x^2$$
$$y'_{(x)} = 2x$$

Die numerische Berechnung in Matlab erfolgt über:

```
>> x = linspace(0,3,1000);
>> y = x.^2;
>> dy = gradient(y,x)
```

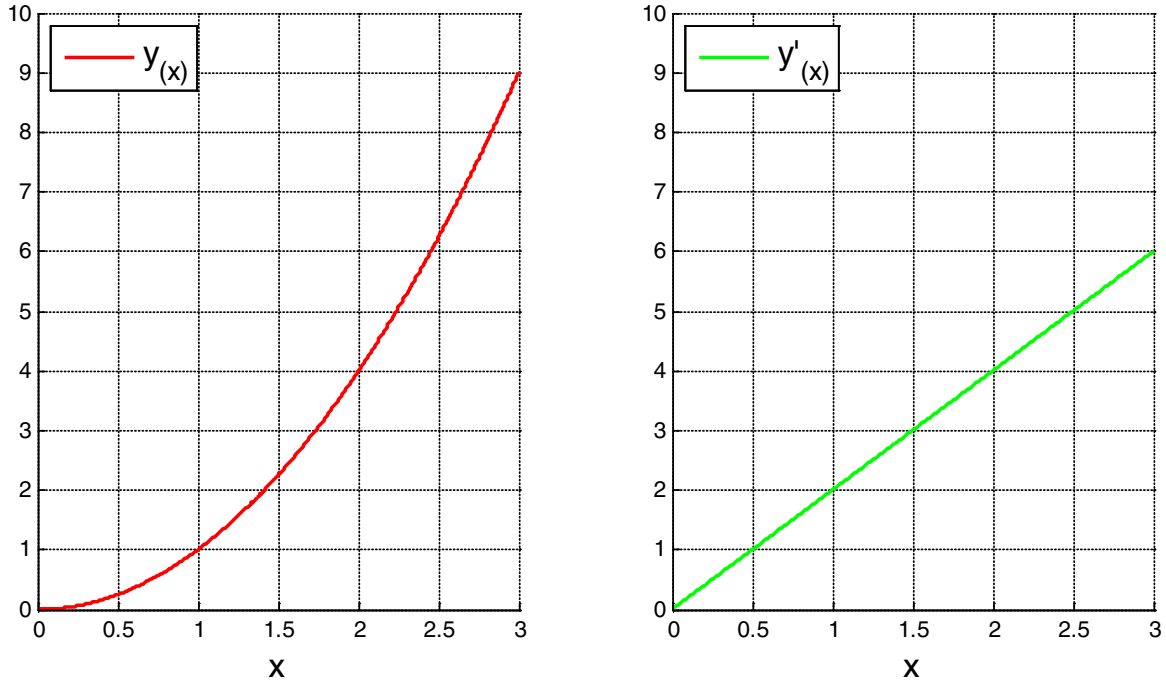


Abbildung 20: Numerische Differentiation

4.3 Datenanalyse von Messergebnissen und Statistik

Eine Messreihe enthält für gewöhnlich eine große aber endliche Anzahl an Stichproben oder Einzelmessungen. Eine zusammenhängende Messreihe kann in der Regel als ein numerischer Datenvektor dargestellt und somit in Matlab weiterverarbeitet werden. Um aus einer Messreihe relevante Informationen extrahieren und angeben zu können, muss diese oftmals mit Methoden der Statistik aufbereitet und analysiert werden. Einige (wenige) wichtige Funktionen zur Datenanalyse und Statistik sollen im folgenden vorgestellt kurz werden.

4.3.1 Sortieren einer Messreihe

Das auf- oder absteigende Sortieren von Elementen eines Vektors oder einer Matrix ist oftmals die Grundlage für eine nachfolgende Datenanalyse. Matlab stellt hierzu die Sortierroutine `sort()` bereit. Die allgemeine Syntax von `sort()` ist:

```
[y,ind] = sort(x,dim,mode)
```

`x` ist die unsortierte und `y` die sortierte Variable. `dim` und `mode` sind optionale Argumente. `dim` bestimmt die Richtung (Dimension) entlang der sortiert werden soll, bei einer Matrix also entweder in Spaltenrichtung (`dim=1`) oder in Zeilenrichtung (`dim=2`). Mit `mode` kann ausgewählt werden, ob in aufsteigender (`mode='ascend'`) oder in absteigender Richtung (`mode='descend'`) sortiert werden soll. Der optionale Rückgabewert `ind` gibt zusätzlich den Index der sortierten Elemente aus der unsortierten Variablen `X` an. Beispiel:

```
>> x = [3 7 6;4 3 1;6 8 6];
>> [y,ind] = sort(x,1,'ascend')

y =
      3      3      1
      4      7      6
      6      8      6

ind =
      1      2      2
      2      1      1
      3      3      3
```

4.3.2 Minimal- und Maximalwerte

Die Befehle `min()` und `max()` dienen der Berechnung minimaler bzw. maximaler Werte eines Datenvektors.

```
>> x = [10 35 12 -1 -1 8 15];
>> max_x = max(x)
max_x = 35
```

Ist das Argument `x` hingegen eine Matrix, so werden die Spaltenmaxima berechnet und in einem Zeilenvektor abgelegt:

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

x =

      1      2      3
      4      5      6
      7      8      9

>> x_max = max(x)

x_max =   7      8      9
```

Der Befehl `max(A,B)`, also mit zwei Funktionsargumenten vergleicht die beiden Matrizen `A` und `B` elementweise und liefert den jeweils größeren Wert zurück. `A` und `B` müssen dabei die gleiche Dimension haben.

```
>> A = [1 2 3;4 5 6;7 8 9];
>> B = [3 2 1;6 5 4;9 8 7];
>> C = max(A,B)

C =

      3      2      3
      6      5      6
      9      8      9
```

Der Befehl `mode(x)` liefert das häufigste in `x` enthaltene Element zurück:

```
>> x = [4 2 4 8 6 9 7 7 4];
>> x_mode = mode(x)

X_mode = 4
```

4.3.3 Mittelwerte

In der Statistik gibt es verschiedene Typen von Mittelwerten. Zwei wichtige Vertreter sind hierbei der *arithmetische Mittelwert* und der *Median Mittelwert*. Für den arithmetischen Mittelwert gilt:

$$\overline{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$$

Der arithmetische Mittelwert eines Vektors `x` wird mit dem Befehl `mean(x)` berechnet werden. Wie man anhand obiger Formel leicht nachvollziehen kann, gilt für den arithmetischen Mittelwert des folgenden Vektors:

```
>> x = [1 1 2 5 5 7 7];
>> mean(x)

ans = 4
```

Ist das Argument eine Matrix, wird der Mittelwert spaltenweise berechnet und als Zeilenvektor ausgegeben.

Der *Median Mittelwert* (Zentralwert) wird folgendermaßen berechnet: Zuerst werden die Elemente des zu untersuchenden Vektors aufsteigend sortiert (dies ist bei dem Beispielvektor schon der Fall). Der Median ist nun jenes konkrete Element des sortierten Vektors, welches in dessen "Mitte" steht. Im Beispiel des aus sieben Elementen bestehenden Vektors `x` wäre dies also das Element mit dem Index **4** und dem Zahlenwert **5**.

```
>> median(x)

ans  = 5
```

Diese Vorgehensweise ist so natürlich nur definiert, wenn der Vektor aus einer ungeraden Anzahl an Elementen besteht. Ansonsten ist der Median das arithmetische Mittel aus den beiden "benachbarten" Elementen in der Mitte des Vektors. Der Vorteil des Median Mittelwerts ist, dass dieser unempfindlich gegen über "Ausreißern" ist.

4.3.4 Streuungsmaße

In der *Stochastik* (Oberbegriff für Wahrscheinlichkeitstheorie und Statistik) ist die *Varianz* s^2 einer statistischen Verteilung ein Streuungsmaß, d. h. ein Maß für die Abweichung von einem *Erwartungswert*. Die *Stichprobenvarianz* s^2 oder *empirische Varianz* stellt einen auf Stichproben basierenden Schätzwert für die Varianz dar. Die Stichprobenvarianz einer Messreihe bzw. eines Datenvektors x mit der Länge n und dem arithmetischen Mittelwert $\overline{x}$ ist folgendermaßen definiert:

$$s^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} \left(x_i - \overline{x} \right)^2$$

Sie ist ein Maß für die Zuverlässigkeit der Messwerte einer Messreihe. Die Stichprobenvarianz eines Vektors kann in Matlab einfach mit dem Befehl `var()` berechnet werden.

```
>> var([1 1 2 5 5 7 7])

ans =  7
```

Die *empirische Standardabweichung* oder *mittlerer Fehler der Einzelmessung* oder auch kurz *Streuung* ist die Quadratwurzel aus der Stichprobenvarianz und ist wiederum ein Schätzwert für die *Standardabweichung* σ:

$$s = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} \left(x_i - \overline{x} \right)^2}$$

Die Streuung kann in Matlab mit dem Befehl `std()` berechnet werden:

```
>> std([1 1 2 5 5 7 7])

ans = 2.6458
```

Die Standardabweichung hat dabei die gleiche Dimension (physikalische Einheit) wie die Variable x.

4.3.5 Normal- und gleichverteilte Zufallsgrößen

Zufallszahlen können beispielsweise dazu verwendet werden, um ein bestimmtes Verhalten zu simulieren, zum Beispiel Messrauschen oder unvorhersagbare statistische Prozesse. Sie sind daher oftmals ein wichtiges Hilfsmittel beim Test und der Evaluierung von Algorithmen und Programmen. Die wichtigsten Vertreter sind die normalverteilten und die gleichverteilten Zufallszahlen.

Normal- oder *Gaußverteilte* Zufallsgrößen sind durch Angabe der Parameter *Mittelwert* μ und *Standardabweichung* σ vollständig über die folgende *Dichtefunktion* charakterisiert:

$$f_{(x)} = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Eine Normalverteilung ist dadurch gekennzeichnet, dass die Zufallswerte in der Nähe des Mittelwertes μ häufiger auftreten als weiter entfernt vom Mittelwert. In Matlab kann man eine aus n normalverteilten Elementen bestehende Variable `f` mit Mittelwert `mu` und Standardabweichung `sigma` folgendermaßen als Spaltenvektor generieren:

```
>> f = sigma*randn(n,1)+mu;
```

Neben der Normalverteilung ist die Gleichverteilung von großer Bedeutung. Im Unterschied zur Normalverteilung ist hier die Dichtefunktion konstant, das heißt in einem abgeschlossenen Intervall $x = [x_{min}\ x_{max}]$ ist die Wahrscheinlichkeit für das Auftreten einer Zufallszahl an jeder Position x gleich.

Ein Spaltenvektor mit `n` gleichverteilte Zufallszahlen zwischen 0 und 1 kann in Matlab mit dem `rand()` Befehl generiert werden.

```
>> f = rand(n,1);
```

4.3.6 Grafische Darstellung von Häufigkeiten

Ein *Histogramm* ist eine grafische Darstellung einer *Häufigkeitstabelle*. In einer Häufigkeitstabelle wird angegeben, wie oft ein bestimmtes Merkmal auftritt, beispielsweise die Häufigkeit einer bestimmten Zufallszahl, siehe Beispiel unten. In Matlab wird mit `hist(daten,merkmal)` ein Histogramm des Datenvektors `daten` geplottet. Ist `daten` ein mxn-Array, so wird das Array spaltenweise ausgewertet und m Histogramme geplottet. Die Variable `merkmal` ist optional. Ist `merkmal` eine ganze Zahl, so wird der gesamte Wertebereich von `daten` in `merkmal` äquidistante Intervalle eingeteilt und die Elemente von `daten` in die entsprechenden Intervalle gelegt. Ist `merkmal` hingegen ein Vektor, so werden die Intervallgrenzen entsprechend dem vorgegebenen Vektor gewählt. Somit können auch unterschiedliche Intervallbreiten zur Histogrammzählung genutzt werden.

Beispiel: Es werden zwei auf Ganzzahlen gerundete Zufallsvektoren `x1` und `x2` mit je einer Million Elementen generiert. `x1` ist normalverteilt mit σ = 5 und μ = 3, `x2` ist im Intervall [-10,+10] gleichverteilt. Mit einem Histogramm soll die Häufigkeit dargestellt werden, wie oft eine der Zufallszahlen im Intervall zwischen zwei ganzen Zahlen in den Grenzen von -20 bis +20 liegt.

```
>> n  = 1e6;
>> x1 = round(5*randn(n,1)+3);
>> x2 = floor(21*rand(n,1))-10;
>> xn = [-20:1:20];

>> Figure(1)
>> subplot(121)
```

```
>> hist(x1,xn);
>> subplot(122)
>> hist(x2,xn);
```

Anmerkung: Machen sie sich klar, weshalb die Rundung auf ganze Zahlen im gleichverteilten Zufallsvektor mit dem `floor()` und nicht mit dem `round()` Befehl realisiert wurde. Prüfen sie gegebenenfalls, wie das Ergebnis mit dem `round()` Befehl ausfällt. Folgende Abbildung zeigt das Ergebnis:

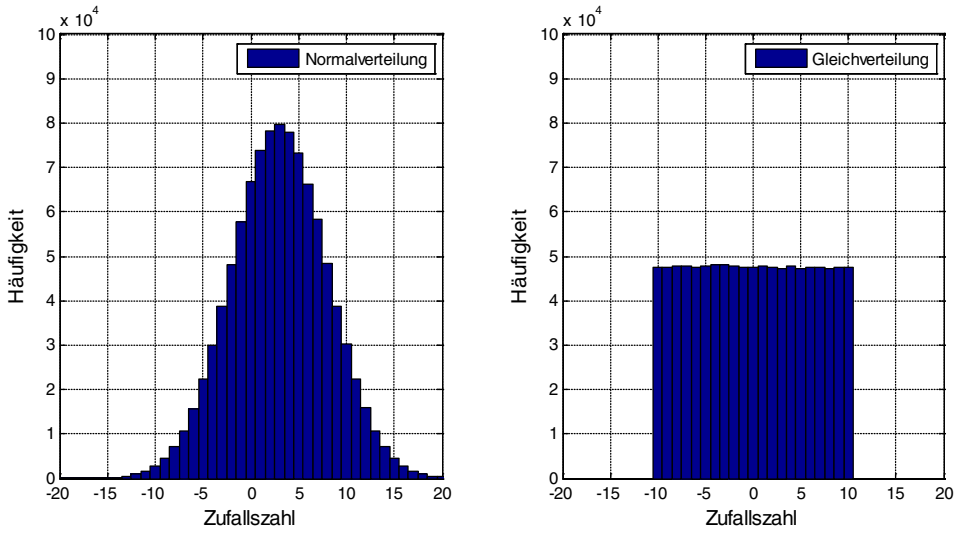


Abbildung 21: Histogramme

Eine andere Möglichkeit Häufigkeiten bzw. Anteile einer Verteilung darzustellen sind beispielsweise *Kuchendiagramme* mittels `pie()`. Beispiel: Von 83 Studenten haben bei einer Klausur die Studenten folgende Noten erhalten:

- 5 Studenten: sehr gut
- 23 Studenten: gut
- 21 Studenten: befriedigend
- 14 Studenten: ausreichend
- 20 Studenten: mangelhaft

Die Notenverteilung soll anschaulich anhand eines Kuchendiagramms dargestellt werden:

```
>> data    = [5 23 21 14 20];
>> explode = [0  0  0  0  1];
>> label   = {'sehr gut','gut','befriedigend','ausreichend','mangelhaft'};
>> pie(data,explode,label)
```

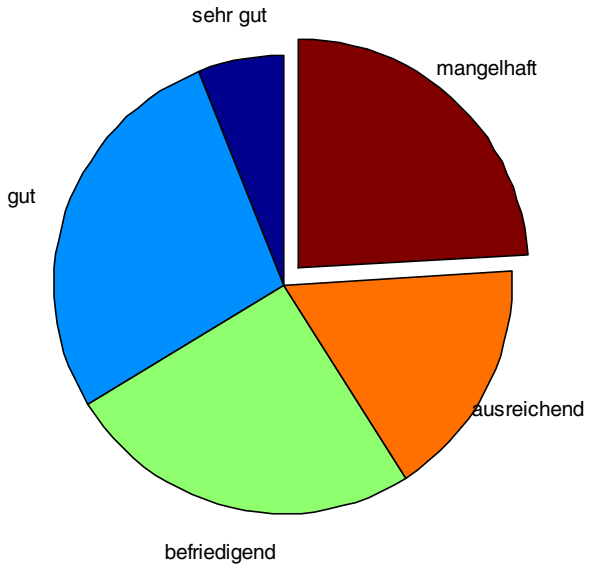


Abbildung 22: Kuchendiagramm

Die Argumente `explode` und `label` sind optional. Beide müssen die gleiche Länge wie der Datenvektor `data` besitzen. Mit `explode` kann man Kuchenstücke abtrennen und hervorheben, je nachdem ob das jeweilige Element den Wert 0 oder 1 (bzw. *false* oder *true*) trägt. In `label` erfolgt die Beschriftung der einzelnen Segmente. `label` ist eine Zelle mit Zeichenketten. Fehlt der Vektor `label`, dann wird als Beschriftung automatisch eine Prozentangabe an die jeweiligen Kuchenstücke gesetzt.

4.4 Numerische Lösung von Gleichungen und Gleichungssystemen

4.4.1 Lineare Gleichungssysteme

Lineare Gleichungssysteme treten in unzähligen technischen und naturwissenschaftlichen Anwendungen auf. Es wurden daher große Anstrengungen unternommen, um schnelle und numerisch stabile Algorithmen zur Lösung solcher Systeme zu entwickeln. Die Frage, ob ein System überhaupt lösbar ist, ist unter anderem Gegenstand der *Lineare Algebra* und soll hier nicht weiter vertieft werden

Anmerkung: Lineare Gleichungssysteme, die aus realen technischen/physikalischen Systemen resultieren, zum Beispiel aus elektrotechnischen Netzwerken oder mechanischen Kräftesystemen, sind in der Regel immer lösbar.

Der \ Operator (bzw. die äquivalente Funktion `mldivide()`) kann in Matlab zur Lösung Linearer Gleichungssysteme verwendet werden. Die Dokumentation gibt unter anderem auch Aufschluss darüber, welch Algorithmen in dem Operator \ zur Lösung unterschiedlich gearteter Gleichungssystem implementiert wurden.

Wir betrachten ein gewöhnliches Lineares Gleichungssystem: $A \cdot \vec{x} = \vec{b}$

Beziehungsweise:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

Die Matrix A ist die sogenannte *Koeffizientenmatrix* und der *Spaltenvektor* x der Lösungsvektor des Linearen Gleichungssystems. Beispiel:

$$\begin{bmatrix} 1 & 2 & 2 \\ 3 & 2 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$$

Der Lösungsvektor x des obigen *lösbaren* Gleichungssystems kann einfach mit der Linksinversen berechnet werden:

```
>> A = [1 2 2;3 2 2;1 2 1];
>> b = [1;2;2];
>> x = A\b

x = 0.5000
    1.2500
   -1.0000
```

Kontrolle:

```
>> b_kontrolle = A*x

b_kontrolle =

      1
      2
      2
```

Anmerkung: Das gleiche Ergebnis erhält man, wenn man die *Inverse* von `A` mit `b` multipliziert. Die Inverse erhält man mit dem Befehl `inv(A)`.

```
>> x = inv(A)*b

x = 0.5000
     1.2500
    -1.0000
```

4.4.2 Nichtlineare Gleichungen

Die Nullstellen *nichtlinearer skalarer Funktionen* können numerisch mit dem Befehl `fzero()` ermittelt werden. Hierzu betrachten wir folgende nichtlineare Funktion:

$$y_{(x)} = \frac{\sin(\pi \cdot x)}{\pi \cdot x}$$

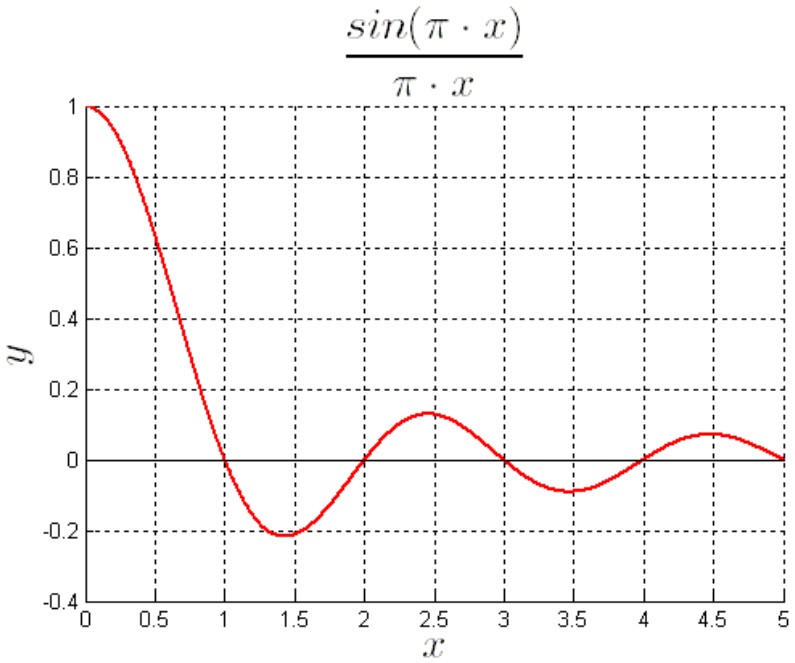


Abbildung 23: Nichtlineare Gleichung

Aus der Abbildung ist ersichtlich, dass $y_{(x)}$ Nullstellen bei positiven ganzzahligen Werten $x = 1, 2, 3, \ldots$ hat. Zunächst erstellen wir in Matlab ein Function Handle `fun` zu dieser Funktion:

```
>> fun = @(x)(sin(pi*x)./(pi*x);
```

Mittels `fzero()` kann nun eine Nullstelle der Funktion ermittelt werden, die innerhalb eines bestimmten Intervalls $[x_1, x_2]$ liegt.

```
>> x1 = 0.5;
>> x2 = 1.5;
>> x0 = fzero(fun,[x1 x2]);

x0 = 1
```

Somit wurde also die Nullstelle $x_0 = 1$ der Funktion bestimmt. Wichtig hierbei ist, dass innerhalb des Intervalls ein *Vorzeichenwechsel* stattfindet.

Das zweite Argument kann auch lediglich ein Skalar (also kein Intervall) sein. Dann wird die nächstgelegene Nullstelle von diesem Punkt aus gesucht.

```
>> x0 = fzero(fun,2.76)

x0 = 3.0000
```

Anmerkung: Die Beschriftung der Titelzeile und der Achsen in Abbildung 23 erfolgte über LATEX Befehle. Hierzu muss die Interpretation von LATEX Befehlen aktiviert sein. Beispiel:

```
>> title('$$\frac{sin(\pi\cdot x)}{\pi\cdot x}$$','interpreter','latex')
```

4.4.3 Nichtlineare Gleichungssysteme

Matlab bietet im Standardumfang keine Funktion zur Lösung von nichtlinearen Gleichungssystemen. Eine entsprechende Funktion beinhaltet jedoch die *Optimization Toolbox* mit dem Befehl `fsolve()`. Dieser kann im Wesentlichen analog zum Befehl `fzero()` verwendet werden und löst nichtlineare Gleichungssysteme iterativ. Als Beispiel betrachten wir das folgende nichtlineare Gleichungssystem:

$$e^{-x_1} = 2x_1 - x_1^2$$
$$e^{-x_2} = -x_1 - 4x_2$$

Dieses Gleichungssystem ist mittels `fsolve()` lösbar, wenn es *implizit* als *Nullstellenproblem* formuliert wird:

$$f_{(x_1,x_2)} = 0$$

Also:

$$f_{1(x_1,x_2)} = 0 = 2x_1 - x_1^2 - e^{-x_1}$$
$$f_{2(x_1,x_2)} = 0 = -x_1 - 4x_2 - e^{-x_2}$$

Dieses System implementieren wir zunächst als *function* in einem entsprechenden File:

```
function [f] = nonlinear_system(x)

    f(1) = 2*x(1) - x(2)^2 - exp(-x(1));
    f(2) = -x(1)  + 4*x(2) - exp(-x(2));
```

nonlinear_system.m

`fsolve()` benötigt zur Lösung des Gleichungssystem ein Function Handle zur Funktion `nonlinear_system` und einen entsprechenden *Startwert* x_{Start}, von dem die numerische Suche nach den Nullstellen ausgeht. Ein geeigneter Startwert muss gegebenenfalls durch Schätzung oder andere Verfahren bestimmte werden, hierauf soll hier jedoch nicht näher eingegangen werden. Da unser Gleichungssystem zwei Variablen x_1 und x_2 aufweist, muss der Startwert x_{Start} folglich ein Vektor mit zwei Elementen sein.

```
>> nonlin = @nonlinear_system;
>> xstart(1) = -5
>> xstart(2) = -5;
>> x0 = fsolve(nonlin,xstart)
```

`fsolve()` ermittelt daraufhin die beiden Nullstellen $x_0 = [x_{01}\ x_{02}]$ des Systems und bestätigt mit einer entsprechenden Meldung, dass diese gefunden wurden:

```
Optimization terminated: first-order optimality is less than options.TolFun.
```

```
x0 = 0.3816     0.2837
```

Beide Funktionen $f_{1(x1,x2)}$ und $f_{2(x1,x2)}$ nehmen also für $x_{01} = 0.3816$ und $x_{02} = 0.2837$ den Wert 0 an. Wir kontrollieren zunächst das Ergebnis:

```
>> kontrolle = nonlin(x0)

kontrolle =

  1.0e-014 *

   -0.8660    -0.0444
```

Das Ergebnis ist definitiv ungleich 0, jedoch ist es mit einem Exponenten von 10^{-14} aus numerischer Sicht relativ klein. Die von `fsolve()` ausgegebene Meldung sagt, dass die *first order optimality* kleiner als der Wert `options.TolFun` ist. Was bedeutet dies? Dem Befehl `fsolve()` (wie auch `fzero()`) kann über eine Strukturvariable `options` mitgeteilt werden, wie die Nullstellensuche erfolgen soll, das heißt welche Algorithmen zur Nullstellensuche benutzt werden sollen, etc. Das Feld `TolFun` gibt Auskunft darüber, wann das *Abbruchkriterium* für die iterative Nullstellensuche erreicht wird, also wenn das Ergebnis "genau genug" ist. Die `options` Struktur zum Befehl `fsolve()` mit den standardmäßig eingestellten Werten erhält man über:

```
>> options = optimset('fsolve');
```

Betrachten wir das Feld `TolFun`, so steht dort der Wert:

```
>> options.TolFun

ans = 1.0000e-006
```

Wenn also das Ergebnis kleiner al 10^{-6} ist, wird die Nullstellensuche abgebrochen und das Ergebnis ausgegeben. Wir lösen das Gleichungssystem noch einmal, teilen `fsolve()` über die `options` Struktur jedoch mit, dass das Ergebnis jeder Iteration im Command Window dargestellt werden soll. Hierzu setzen wir das Feld `Display` in der `options` Struktur auf den Wert `'iter'`.

```
>> options=optimset('Display','iter');
>> x0 = fsolve(nonlin,xstart,options)
```

Daraufhin erfolgt am Command Window eine ausführliche Auflistung des Iterationszyklus

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	3	60344.2		2.74e+004	1
1	6	17448.9	1	7.61e+003	1
2	9	5342.47	1	2.27e+003	1
3	12	1705.87	1	732	1
4	15	541.896	1	249	1
5	18	155.515	1	82.8	1
6	21	33.3357	1	21.4	1
7	24	3.98853	1	6.84	1
8	27	0.0444906	0.609205	0.575	2.5
9	30	6.9606e-006	0.082245	0.00696	2.5
10	33	1.79697e-013	0.00106129	1.11e-006	2.5
11	36	7.51883e-029	1.70826e-007	2.28e-014	2.5

```
Optimization terminated: first-order optimality is less than options.TolFun.

x0 = 0.3816     0.2837
```

Über diese Auflistung wird deutlich, dass nach elf Iterationen das Ergebnis kleiner als der in `options.TolFun` eingestellte Wert von 10^{-6} ist, der Iterationsalgorithmus abbricht und das Ergebnis in der Variablen `x` ausgibt.

Eine genauere Behandlung der in `fsolve()` implementierten Algorithmen und der Steuerung des Lösungsverhaltens über die `options` Struktur soll an dieser Stelle nicht erfolgen. Es sei hierzu auf die Dokumentation verwiesen. Das Gleichungssystem mit der Lösung x_0 wird in Abbildung 24 visualisiert.

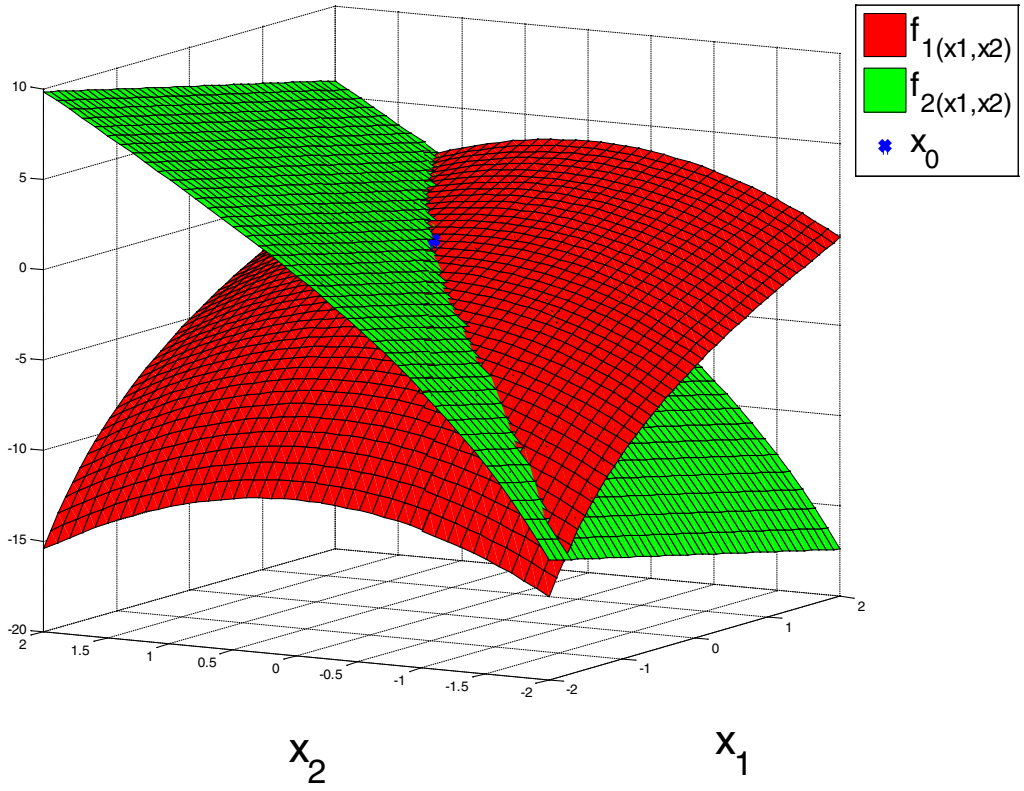


Abbildung 24: Lösung des nichtlinearen Gleichungssystems

4.5 Symbolisches Rechnen mit der "Symbolic Math Toolbox"

Beim symbolischen Rechnen werden die analytischen und algebraischen Methoden, die man vom Rechnen mit Papier und Bleistift her kennt, auf Computern abgebildet. Um mit Matlab symbolisch rechnen zu können, muss die *Symbolic Math Toolbox* installiert sein. Symbolische Rechnungen basieren auf Variablen, denen nicht unbedingt Zahlenwerte zugewiesen sind. Vorteile des Symbolischen Rechnens sind unter anderem:

- Arithmetische Operationen können exakt und nicht nur näherungsweise durchgeführt werden.
- Man kann Polynome oder rationale Ausdrücke symbolisch addieren, subtrahieren und dividieren.
- Ausdrücke können beispielsweise differenziert werden und man erhält die gleichen Ergebnisse, die bisher nur mit Bleistift und Papier zu erzielen waren.
- Ausdrücke können unbestimmt integriert werden.
- Es lassen sich auch kleinere lineare Gleichungssysteme ohne Rundungsfehler lösen.

Diese Möglichkeiten erleichtern das ermüdende und fehlerbedrohte Manipulieren komplizierter Ausdrücke, das häufig numerischen Berechnungen vorangeht. Seit Matlab 7.7 (R2008b) basiert die Symbolic Math Toolbox auf dem von der Universität Paderborn entwickelten Programmsystem *MuPAD* (und nicht mehr auf *Maple*).

4.5.1 Symbolische Variablen und Zahlen

Symbolische Variable müssen in Matlab vor der Verwendung mit dem Schlüsselwort `sym` (eine Variable) oder `syms` (mehrere Variablen) als solche deklariert werden:

```
>> a = sym('a');
>> syms('x','y','z');
```

Beziehungsweise:

```
>> syms x y z;
```

legt im Workspace die symbolischen Variablen `a, x, y` und `z` an. Wie von den numerischen Datentypen gewohnt können natürlich auch symbolische Arrays und Matrizen angelegt werden.

Symbolischen Variablen können wie herkömmlichen numerischen Variablen auch Zahlenwerte zugeordnet werden. Der Unterschied ist, dass eine Zahl nicht durch eine 64-Bit Gleitkommazahl angenähert wird, sondern dass diese den exakten Zahlenwert repräsentiert. Wir vergleichen hierzu symbolische und numerische Zahlenwerte und generieren hierzu zwei entsprechende Variable:

```
>> symbolic = sym('sqrt(2)');
>> numeric  = sqrt(2);
```

Anschließend quadrieren wir die Variablen:

```
>> symbolic = symbolic^2;
>> numeric  = numeric^2;
```

Und machen einen Vergleich, ob die Variablen nun auch wirklich der Zahl 2 entsprechen:

```
>> check = [symbolic==2, numeric==2]

check =  1      0
```

Der Symbolische Ausdruck symbolic entspricht also exakt dem Zahlenwert 2, genauso wie es die Rechnung mit Papier und Bleistift ergeben würde:

$$\text{symbolic} = \left(\sqrt{2}\right)^2 = 2$$

Wohingegen `numeric` lediglich einen Näherungswert an die Zahl 2 darstellt, der über das numerische quadrieren der Quadratwurzel von 2 angenähert wurde.

4.5.2 Symbolische Funktionen

Ebenso wie symbolische Variable können auch symbolische Funktionen definiert werden. Beispielsweise kann man die Funktion

$$z_{(x,y)} = \left(4x^2 + \sqrt{xy}\right) \cdot e^{-2x}$$

folgendermaßen symbolisch definieren:

```
>> syms x y
>> z = (4*x^2 + sqrt(x*y))*exp(-2*x)

z = (4*x^2+(x*y)^(1/2))*exp(-2*x)
```

Im Command Window kann man sich diesen recht unübersichtlichen Zeilenausdruck mit Hilfe des Befehls `pretty()` in einer etwas aufbereiteten Darstellung anzeigen lassen:

```
>> pretty(z)

(4 x^2  + (x y)^1/2) exp(-2 x)
```

Eine symbolische Variable kann man nun beispielsweise ganz leicht mit der Funktion `subs()` durch einen konkreten Zahlenwert ersetzen:

```
>> a = subs(z,x,2)
a = (16+2^(1/2)*y^(1/2))*exp(-4)
```

Das erste Argument beschreibt dabei die symbolische Funktion `z`, das zweite die in `z` zu ersetzende Variable `x` und das dritte den Wert durch den die Variable `x` ersetzt werden soll. Selbstverständlich kann man auch mehrere Variable auf einmal ersetzen:

```
>> a = subs(z,[x y],[2 8])
a = 0.3663
```

Werden alle Variablen in einer symbolischen Funktion durch konkrete Zahlenwerte ersetzt, dann ist das Ergebnis wieder vom Datentyp double.

4.5.3 Differenzieren und Integrieren

Die Mächtigkeit der symbolischen Berechnung wird sofort deutlich, wenn man beispielsweise die Funktion `z` mit dem Befehl `diff()` differenzieren möchte:

```
>> dz_dx = diff(z,x)
```

Dies leitet die Funktion `z` einmal nach der Variablen `x` ab:

```
dz_dx = (8*x+1/2/(x*y)^(1/2)*y)*exp(-2*x)-2*(4*x^2+(x*y)^(1/2))*exp(-2*x)
>> pretty(dz_dx)

          /              y    \
         |8 x + 1/2 ----------| exp(-2 x) - 2 (4 x^2  + (x y)^1/2) exp(-2 x)
         |                  1/2|
          \            (x y)  /
```

Ebenso kann man nach jeder beliebigen anderen Variablen ableiten. Durch Angabe eines dritten Arguments `n`, erhält man die `n`-fache Ableitung:

```
>> d2z_dy = diff(z,y,2)
d2z_dy = -1/4/(x*y)^(3/2)*x^2*exp(-2*x)
>> pretty(d2z_dy)

                                    x^2  exp(-2 x)
                          - 1/4 ------------
                                   (x y)^3/2
```

Analog gibt es zur symbolischen Integration den Befehl `int()`. Es soll beispielsweise das *unbestimmte Integral* (die Stammfunktion) zur folgenden Funktion ermittelt werden:

$$y_{(x)} = \frac{1}{4+x^2}$$

Laut mathematischer Formelsammlung gilt:

$$Y_{(x)} = \int \frac{dx}{4+x^2} = \frac{1}{2}\arctan\left(\frac{x}{2}\right)$$

In Matlab erhält man dieses Integral über die Befehlsfolge:

```
>> syms x
>> y = 1/(4+x^2);
>> Y = int(y,x)

Y = 1/2*atan(1/2*x)
```

Das *bestimmte Integral* der Funktion (die Fläche unter der Funktion) in den Integrationsgrenzen -10 bis +10 erhält man, indem man bei der Integration zwei zusätzliche Argumente für die untere und die obere Integrationsgrenze mit angibt:

```
>> A = int(y,x,-10,10)
A = atan(5)

>> A_num = double(A)
A_num = 1.3734
```

4.5.4 Plots von symbolischen Funktionen

Symbolische Funktionen können sehr leicht mit dem `ezplot()` Befehl (easy plot) gezeichnet werden. Die Darstellung der oben definierten Funktion $y_{(x)}$ und ihrer Stammfunktion $Y_{(x)}$ im Bereich von $x = -10$ bis $x = +10$ erhält man beispielsweise über:

```
>> figure(1)
>>      subplot(121)
>>           ezplot(y,[-10,10])
>>           grid on
>>      subplot(122)
>>           ezplot(Y,[-10,10])
>>           grid on
```

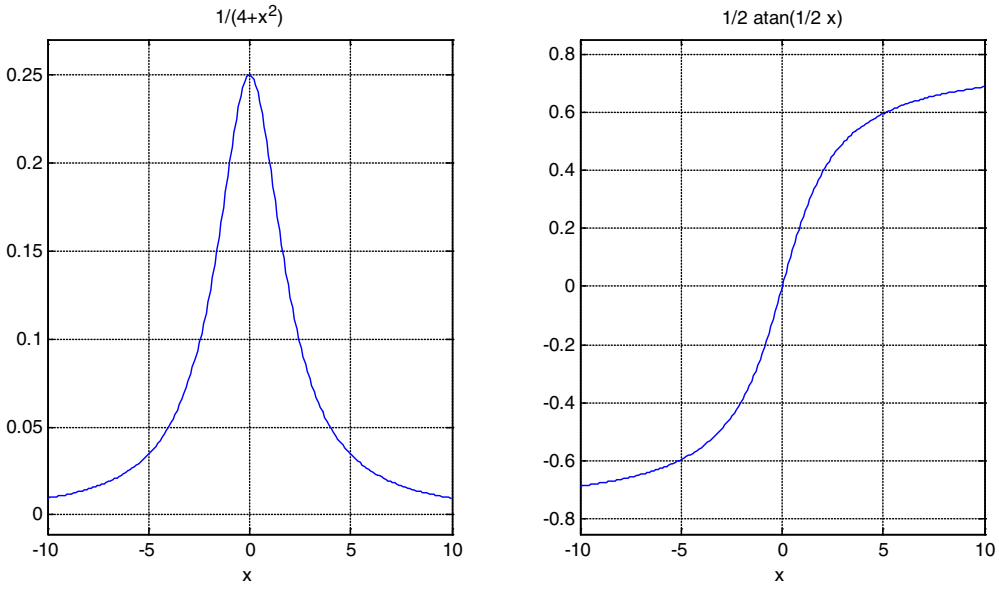


Abbildung 25: Beispiel zu ezplot()

Analog gibt es auch Befehle zur 3D-Darstellung von symbolischen Funktionen, beispielsweise den Befehl `ezsurf()`:

```
>> syms x y
>> z = atan(x)^2*y^3;
>> ezsurf(z,[-pi pi],[-5 5])
```

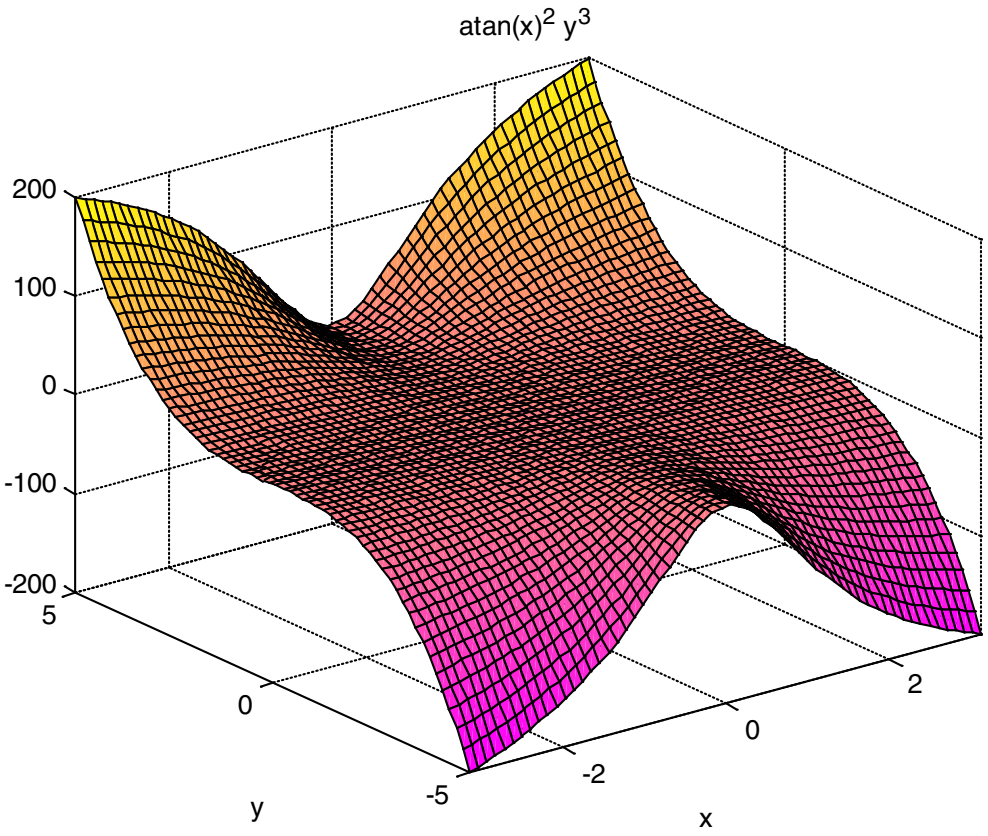


Abbildung 26: Beispiel zu ezsurf()

4.5.5 Vereinfachungen und Zusammenfassungen

Die Symbolic Math Toolbox hat leistungsstarke Algorithmen um symbolische Ausdrücke umzuformen bzw. zu vereinfachen. Der Befehl `symplify()` führt verschiedene analytische Umformtechniken durch und wählt am Schluss die einfachst mögliche Darstellung aus. Bekanntlich gilt:

$$\sin(x)^2 + \cos(x)^2 = 1$$

Dies bestätigt Matlab auch durch den `symplify()` Befehl:

```
>> syms x y
>> y = sin(x)^2+cos(x)^2;
>> y_simple = simplify(y)

y_simple = 1
```

Das *Ausmultiplizieren* von Klammerausdrücken ist mittels `expand()` möglich. Beispiel:

$$y_{(x)} = (x+1)^3 = x^3 + 3x^2 + 3x + 1$$

```
>> syms x y
>> y = (x+1)^3
>> y_ex = expand(y)

y_ex = 1+3*x+3*x^2+x^3
```

Und das umgekehrte *Faktorisieren* mit dem Befehl `factor()`:

```
>> y_fact = factor(y_ex)

y_fact = (1+x)^3
```

4.5.6 Abstrakte Funktionsterme

Will man einen allgemeinen (abstrakten) Funktionsausdruck $f_{(x)}$ erzeugen, das heißt einen Term, der irgendein spezieller Funktionsterm sein kann, so ist das durch die Anweisung

```
>> f = sym('f(x)');
```

möglich. Hierdurch wird das symbolische Objekt `f` erzeugt, das wie $f_{(x)}$ agiert, das heißt dem Objekt `f` ist bekannt, dass eine unabhängige Variable x existiert. Dass dem so ist zeigen die folgenden Zeilen:

```
>> syms f x t
>> f = sym('f(x)');
>> f_dx = diff(f,x);
>> f_dt = diff(f,t);
```

Zuerst werden die drei symbolischen Variablen `f`, `x` und `t` angelegt und anschließend das Objekt `f` als abstrakte Funktion in Abhängigkeit der Variablen `x` definiert. In der dritten Zeile wird $f_{(x)}$ nach `x` abgeleitet und in der vierten Zeile nach `t`. Es gilt hierbei:

$$\frac{\partial f_{(x)}}{\partial x} = f'_{(x)} \qquad \frac{\partial f_{(x)}}{\partial t} = 0$$

Sehen wir uns die entsprechenden Variablen an, die nun im Workspace liegen:

```
>> f_dx
 f_dx = diff(f(x),x)
>> f_dt
 f_dt = 0
```

In der Matlab Terminologie entspricht $f'_{(x)}$ dem Ausdruck `diff(f(x),x)`.

4.5.7 Grenzwerte

Mit dem `limit()` Befehl können *Grenzwerte* von Folgen oder Funktionen berechnet werden. Beispiel: Die Folge

$$a_n = \frac{(2n+1)}{4n} \qquad n \geq 1$$

hat für $n \to \infty$ den Grenzwert 0.5. Der folgende Matlab Code bestätigt dies:

```
>> syms a n
>> a = (2*n+1)/(4*n);
>> limit(a,inf)

ans = 1/2
```

4.5.8 Algebraische Gleichungen

Algebraische Gleichungen können mithilfe des Befehls `solve()` gelöst werden. Kann für die Lösung der Gleichung keine exakte symbolische Lösung gefunden werden, so wird ein Näherungswert angegeben. Beispiel:

```
>> solve('x^2-x = 0')
 ans =
  0
  1
```

Dies liefert die beiden Nullstellen der Gleichung in einem Spaltenvektor. Dies ist identisch mit der vereinfachten Schreibweise:

```
>> solve('x^2-x')
```

Beide Nullstellen können exakt symbolisch berechnet werden. Wohingegen die Gleichung

```
>> solve('cos(x) = x')

ans = .739085133215160641655312087673 87
```

zwar eine Lösung hat, für diese aber kein exakter symbolischer Wert gefunden werden kann.

Diese Gleichung kann auch implizit als *Nullstellenproblem* formuliert und entsprechend gelöst werden:

```
>> solve('cos(x)-x')

ans = .739085133215160641655312087673 87
```

4.5.9 Endliche und unendliche Reihen

Endliche Reihen lassen sich mit der Funktion `symsum()` berechnen. Für die folgende endliche Reihe gilt:

$$\sum_{k=1}^{10} 2^k = 2+4+8+\ldots+1024 = 2046$$

was man durch folgende Zeilen bestätigen kann:

```
>> syms k
>> symsum(2^k,1,10)
ans = 2046
```

Es wird also die Summe der Reihe 2^k (erstes Argument) von 1 (zweites Argument) bis 10 (drittes Argument) berechnet. Ob die Summe einer *unendlichen Reihe* existiert und welchen Wert Sie gegebenenfalls hat, lässt sich ebenfalls mit `symsum()` beantworten. Es lässt sich beweisen, dass gilt:

$$\sum_{k=1}^{\infty} \frac{1}{4k^2-1} = \frac{1}{2}$$

Überprüfung mit Matlab:

```
>> syms k
>> Reihe = 1/(4*k^2-1);
>> Konvergenz = symsum(Reihe,1,inf)
Konvergenz = 1/2
```

4.6 Überblick über wichtige mathematische Funktionen

Die folgenden Übersichten sollen einen *Überblick* über die (wichtigsten) mathematischen Standardfunktionen verschaffen. Eine genaue Besprechung der jeweiligen Funktionen soll hier nicht erfolgen.

4.6.1 Trigonometrische Funktionen

Funktion	Bedeutung
cos()	Kosinus
cot()	Kotangens
csc()	Kosekans
sec()	Sekans
sin()	Sinus
tan()	Tangens

Funktion	Bedeutung
acos()	Arcus Kosinus
acot()	Arcus Kotangens
acsc()	Inverser Kosekans
asec()	Inverser Sekans
asin()	Arcus Sinus
atan()	Arcus Tangens
atan2()	4-Quadr. Arcus Tangens

Tabelle 14: Trigonometrische Funktionen

Alle trigonometrischen Funktionen, mit Ausnahme von `atan2()`, akzeptieren als Argument Arrays, die *elementweise ausgewertet* werden. Die Berechnung wird dabei in rad vorgenommen. Für die Berechnung in Grad dienen die mit einem "d" am Ende des Namens ergänzten Funktionen, z.B. `sind()`, `cosd()`, etc.

4.6.2 Hyperbolische Funktionen

Funktion	Bedeutung
cosh()	Kosinus Hyperbolicus
coth()	Kotangens Hyperbolicus
csch()	Kosekans Hyperbolicus
sech()	Sekans Hyperbolicus
sinh()	Sinus Hyperbolicus
tanh()	Tangens Hyperbolicus

Funktion	Bedeutung
acosh()	Areakosinus
acoth()	Areakotangens
acsch()	Area Kosekans Hyperbolicus
asech()	Area Sekans Hyperbolicus
asinh()	Areasinus
atanh()	Areatangens

Tabelle 15: Hyperbolische Funktionen

4.6.3 Exponential- und Logarithmusfunktionen

Funktion	Bedeutung
exp()	Exponentialfunktion
log()	Natürlicher Logarithmus
log10()	Logarithmus zur Basis 10
log2()	Logarithmus zur Basis 2

Tabelle 16: Exponential - und Logarithmusfunktionen

4.6.4 Potenz - und Wurzelfunktionen

Funktion	Bedeutung
nextpow2()	Berechnet zu einer gegebenen Zahl die kleinste Potenz von 2, die gleich oder größer dem Betrag der gegebenen Zahl ist
pow2()	Berechnet für ein Array A zu jedem Element A_{ij} den Wert 2^{Aij}
realpow()	Dient der Elementweisen Potenzierung zweier Matrizen
sqrt()	Quadratwurzel
nthroot()	Berechnet die n-te reelle Wurzel aus x

Tabelle 17: Potenz- und Wurzelfunktionen

Beispiele:

```
>> x = nextpow2(32)
x = 5

>> x = pow2([1 2;3 4])
x = 2      4
     8     16

>> x = realpow([1 2 3 4],[1 2 3 4])
x = 1      4     27    256

>> x = nthroot(27,3)
x = 3
```

4.6.5 Funktionen zur linearen Algebra

Funktion	Bedeutung
'	(Hochkomma) Transponierte einer Matrix
det()	Determinante einer Matrix
inv()	Inverse einer Matrix
eig()	Eigenwerte einer Matrix
rank()	Rang einer Matrix
norm()	Vektor- und Matrixnorm
diag()	Diagonalteil einer Matrix

Tabelle 18: Funktionen zur linearen Algebra

5 Differentialgleichungen

Differentialgleichungen spielen eine herausragende Rolle in Naturwissenschaft und Technik. Eine Vielzahl von Naturgesetzen, Phänomenen, Prozessen und Systemen kann mit Hilfe von Differentialgleichungen und darauf aufbauenden mathematischen Modellen beschrieben und analysiert werden. Wegen ihrer Bedeutung soll an dieser Stelle eine etwas ausführlichere Einleitung in die Thematik gegeben werden.

5.1 Definition

Eine Differentialgleichung liegt vor, wenn für eine gesuchte *Funktion* $y_{(x)}$ eine Gleichung aufgestellt werden kann, in der neben der gesuchten Lösungsfunktion $y_{(x)}$ selbst auch noch eine oder mehrere Ableitungen (Differentiale) $y'_{(x)}$, $y''_{(x)}$, ... dieser Funktion auftreten. x ist dabei eine von der Lösungsfunktion unabhängige Variable oder ein Vektor unabhängiger Variablen.

Differentialgleichungen können einzeln oder als Differenzialgleichungssysteme auftreten. Die Ordnung einer Differentialgleichung ist dabei gleich der höchsten in der DGL auftretenden Ableitung.

Es gibt verschiedene Klassen von Differentialgleichungen. Die für ingenieurwissenschaftliche Betrachtungen bedeutendste Unterscheidung teilt diese in die *Gewöhnlichen Differentialgleichungen* und die *Partiellen Differentialgleichungen*. Gewöhnliche Differentialgleichungen hängen nur von *einer einzigen* unabhängigen Variablen t ab, wohingegen partielle Differentialgleichungen von *mehr als einer* unabhängigen Variablen, also beispielsweise von t, x, y, z, …, abhängen.

Obwohl in diesem Buch nur die Lösung von Gewöhnlichen Differentialgleichung mit Matlab behandelt wird, werden beide Typen aufgrund ihrer jeweiligen Bedeutung für die Natur- und Ingenieurwissenschaften kurz anhand eines klassischen Vertreters vorgestellt.

5.1.1 Gewöhnliche Differentialgleichungen: Lotka-Volterra-Modell

Das Lotka-Volterra-Modell zur Populationsdynamik von Lebewesen ist durch nachfolgendes *nichtlineares Differentialgleichungssystem* gegeben. Es ist ein einfaches Modell, das beschreibt wie eine Beutetierpopulation N_1 und eine Räubertierpopulation N_2 zusammenwirken und sich gegenseitig beeinflussen:

$$\dot{N}_{1(t)} = \alpha_1 \cdot N_{1(t)} - \beta_1 \cdot N_{1(t)} \cdot N_{2(t)}$$
$$\dot{N}_{2(t)} = \alpha_2 \cdot N_{1(t)} \cdot N_{2(t)} - \beta_2 \cdot N_{2(t)}$$

Dabei sind:

- $N_{1(t)}$: Population der Beutetier zum Zeitpunkt t
- $N_{2(t)}$: Population der Räubertiere zum Zeitpunkt t
- α_1: Reproduktionsrate der Beutetiere bei unbegrenztem Nahrungsangebot
- α_2: Reproduktionsrate der Räubertiere pro Beutetier
- β_1: Sterberate der Beutetiere pro Räubertier
- β_2: Sterberate der Räubertiere, wenn kein Beutetiere vorhanden

Die einzige unabhängige Variable ist in diesem Fall die Zeit t. Die interessierenden Größen sind die Tierpopulationen $N_{1(t)}$ und $N_{2(t)}$. Diese beiden zeitabhängigen Größen bilden die gesuchte Lösungsfunktion $y_{(t)}$. Die Lösungsfunktion ist in diesem Fall also ein *Lösungsvektor*:

$$y_{(t)} = \begin{bmatrix} N_{1(t)} \\ N_{2(t)} \end{bmatrix}$$

α_1, α_2, β_1 und β_2, sind die sogenannten *Systemparameter*. Durch diese *Zahlenwerte* werden die quantitativen Eigenschaften des Systems festgelegt. Da nur eine unabhängige Variable vorliegt, handelt es sich um ein System gewöhnlicher Differentialgleichungen. Dieses System beschreibt die *Änderungsraten* der beiden Populationen, wenn zu einem bestimmten Zeitpunkt t die Beutetierpopulation $N_{1(t)}$ Tiere und die Räubertierpopulation $N_{2(t)}$ Tiere beträgt. Es beschreibt damit also gewissermaßen die Entwicklung beider Tierpopulationen in die Zukunft.

Die Struktur des Differentialgleichungssystem, also die Anzahl der auftretenden Ableitungen und die Art der Verknüpfung der einzelnen Funktionen untereinander, legt im Wesentlichen die *qualitativen Eigenschaften* der Änderungsraten beider Populationen $N_{1(t)}$ und $N_{2(t)}$ fest. Diese Struktur sagt für sich alleine jedoch noch nichts über den konkreten quantitativen Verlauf der Lösungsfunktionen $N_{1(t)}$ und $N_{2(t)}$ aus. Soll eine konkrete Lösungsfunktion ermittelt werden, dann müssen neben der Systemstruktur auch konkrete Zahlenwert für die Systemparameter und für die *Anfangsbedingungen* $N_{1(t=0)}$ und $N_{2(t=0)}$ bekannt sein (Zwei DGL erster Ordnung → zwei Anfangsbedingungen). Die Systemdynamik "startet" gewissermaßen aus den Anfangsbedingungen heraus. Hieraus folgt, dass sich für jedwede Variation der Systemparameter oder der Anfangsbedingungen eine andere Lösungsfunktion einstellt. Somit handelt es sich bei dem vorliegenden System um ein sogenanntes *Anfangswertproblem*.

Zum Veranschaulichung der abstrakten Gleichungen stelle man sich den Sachverhalt folgendermaßen vor: In einem ausreichend großen und umzäunten Waldstück (= in sich abgeschlossenes System) werden zur gleichen Zeit an willkürlichen Position $N_{1(t=0)} = 100$ Hasen und $N_{2(t=0)} = 4$ Füchse ausgesetzt. Durch die Angabe, dass es sich um Hasen und Füchse handelt, werden im Wesentlichen die Zahlenwerte der Systemparameter α_1, α_2, β_1 und β_2 festgelegt. Würde es sich um andere Tierarten handeln, dann würden die Systemparameter auch andere Zahlenwerte annehmen. Das Lotka-Volterra-Modell beschreibt nun (stark vereinfacht), wie sich diese beiden Populationen in dem abgegrenzten Waldstück entwickeln, also wie sich die Tiere vermehren und sterben bzw. fressen und gefressen werden. Durch eine Festlegung der Anfangsbedingungen und der Systemparameter, die das Verhalten der Tiere charakterisieren, gibt es eine einzige und eindeutige Lösung des Differentialgleichungssystems, also der Populationsentwicklung. Dies sind die beiden Funktionen $N_{1(t)}$ und $N_{2(t)}$.

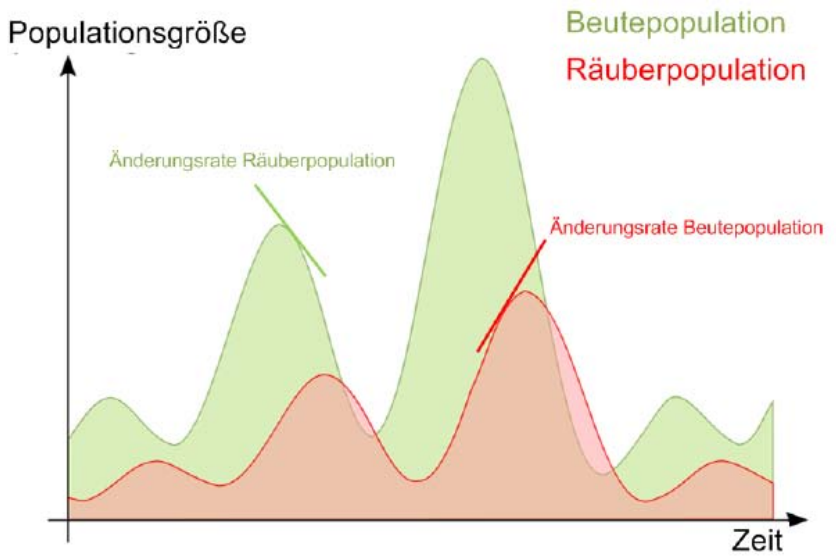


Abbildung 27: Lotka-Volterra-Modell

Versuchen Sie die beiden zeitlichen Verläufe der Beute- und der Räuberpopulation anhand obiger Abbildung zu interpretieren. Beschreiben Sie hierzu den dargestellten Sachverhalt in Worten.

Anmerkungen: Gerade die Ermittlung von Zahlenwerten für die Systemparameter ist oftmals ein schwieriges Unterfangen, was an obigem Beispiel sehr deutlich wird. Durch die vier Systemparameter α_1, α_2, β_1 und β_2 wird im Wesentlichen das *komplette Verhalten* der beiden Tierpopulationen charakterisiert. Auch wenn Biologen oder Ökologen dazu in der Lage wären, hierfür einigermaßen verlässliche Zahlenwerte zu ermitteln, so sollte man sich doch immer deutlich vor Augen halten, dass es sich hierbei nur um "Richtwerte" handeln kann. Natürliche, ökonomische, ökologische oder soziologische Prozesse lassen sich nur schwer durch absolute Zahlen quantifizieren. Bei entsprechenden Modellen und Simulationen, sollte man sich daher immer Gedanken darüber machen, ob die quantitativen Lösungsergebnisse wirklich eine entsprechende Aussagekraft besitzen, oder ob nur der qualitative Verlauf bewertet werden kann. Bei technischen Systemen lassen sich in vielen Fällen die Systemparameter leichter und genauer quantifizieren, aber auch hier ist Vorsicht geboten. Schon augenscheinlich einfache Prozesse wie Gleit- und Haftreibung sind realistisch nur mit entsprechend aufwendigen Modellen modellier- und simulierbar. Diese Thematik ist unter anderem Gegenstand des weiten Gebietes *Modellbildung und Simulation* und soll an dieser Stelle nicht weiter vertieft werden.

5.1.2 Partielle Differentialgleichungen: Maxwell-Gleichungen

Die Maxwell-Gleichungen (nach James Clerk Maxwell) bilden ein spezielles System von vier linearen Partiellen Differentialgleichungen (PDE) erster Ordnung, das von grundlegender Bedeutung für die gesamte Physik und Elektrotechnik ist. Die Gleichungen beschreiben die Erzeugung von elektrischen und magnetischen Feldern durch Ladungen und Ströme und zugleich die Veränderung dieser Felder mit der Zeit. Zusammen mit der Lorentzkraft erklären die Maxwell-Gleichungen alle Phänomene der klassischen Elektrodynamik.

$$\nabla \cdot D_{(t,x,y,z)} = \rho$$
$$\nabla \cdot B_{(t,x,y,z)} = 0$$
$$\nabla \times E_{(t,x,y,z)} = -\dot{B}_{(t,x,y,z)}$$
$$\nabla \times H_{(t,x,y,z)} = -j_{(t,x,y,z)} + \dot{D}_{(t,x,y,z)}$$

D, B, E, H und j sind dabei veränderliche Größen, sie hängen aber nicht nur von der unabhängigen Variablen t (Zeit), sondern auch von drei Raumkoordinaten x, y und z ab, somit liegen hier ein Gleichungssystem partieller Differentialgleichungen vor.

Anmerkung: Der Nabla Operator ∇ ist ein Differentialoperator der Mathematik, der auf Vektoren wirkt.

Gewöhnliche und partielle Differentialgleichungen sind völlig unterschiedliche Klassen von DGLs. Dementsprechend unterscheiden sich auch die Lösungsalgorithmen vollständig. Matlab bietet eine Toolbox zur Lösung partieller Differentialgleichungen (Partial Differential Equation Toolbox), jedoch gibt es spezielle ausgereifte Programmpakete, die partielle Differentialgleichungen zum Beispiel mittels der Finite Elemente Methode (FEM) lösen. Nachfolgende Abbildung zeigt das Ergebnis der Berechnung des Feldlinienverlaufs einer Helmholtz-Spule mit dem Programm "Comsol Multiphysics".

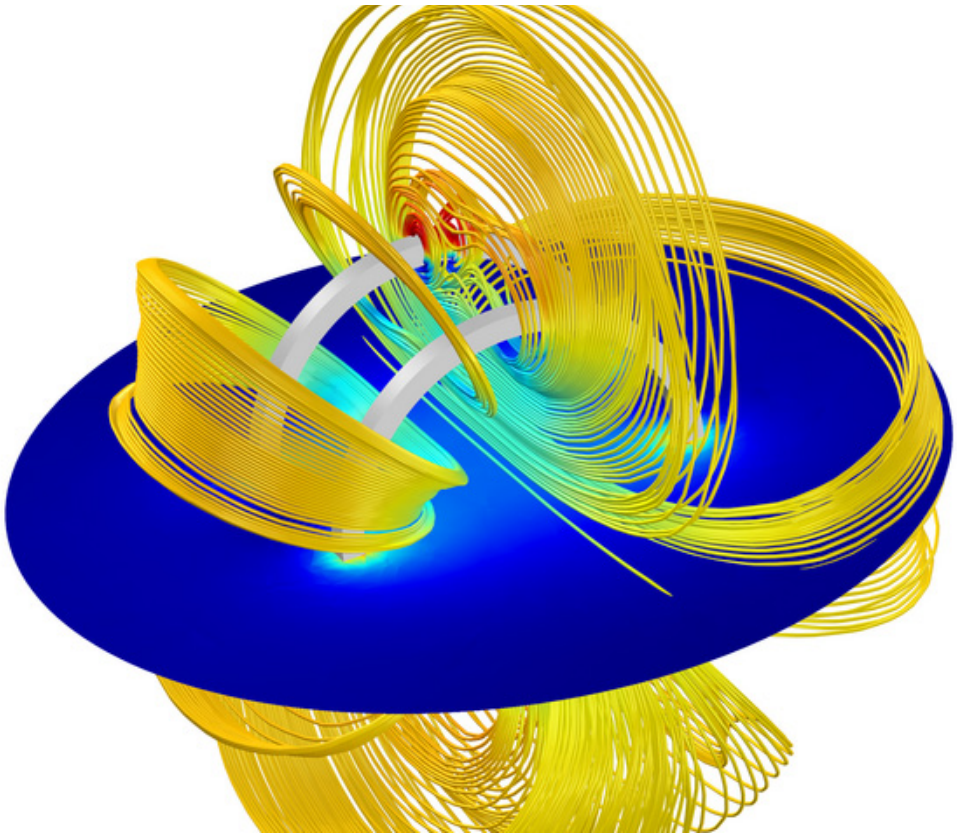


Abbildung 28: PDE Helmholtz Spule (Quelle: www.comsol.com)

5.2 Zur numerischen Lösung von gewöhnlichen Differentialgleichungen

In Mathematik Vorlesungen des Grundstudiums ingenieurwissenschaftlicher Studiengänge werden für Gewöhnlich verschiedene Verfahren zur Lösung von gewöhnlichen Differentialgleichungen und Differentialgleichungssystemen kennengelernt, beispielsweise *Trennung der Variablen* und *Variation der Konstanten*. Dies sind analytische Verfahren, das heißt die Differentialgleichungen werden mit "Papier und Bleistift" gelöst und man erhält für die gesuchte Lösungsfunktion einen formelmäßigen Ausdruck.

Leider existieren nur für relativ wenige Differentialgleichungstypen geschlossene analytische Methoden zur Ermittlung einer Lösungsfunktion. Zum Beispiel für die Klasse der *Linearen Differentialgleichungen mit konstanten Koeffizienten*. Linear bedeutet hier im Wesentlichen, dass die veränderlichen Größen nur in erster Potenz auftreten. Allein schon die Multiplikation zweier veränderlicher Größen wie im Lotka-Volterra-Modell die Populationsgrößen $N_{1(t)}$ und $N_{2(t)}$ führt zu einem nichtlinearen Differentialgleichungssystem. Für nichtlineare DGLs existieren nur für besondere Spezialfälle analytische Lösungsmethoden. Eine formelmäßige Funktion als Lösung der Differentialgleichung lässt sich also in diesen Fällen meist nicht ableiten. Dass eine DGL in Naturwissenschaft oder Technik linear ist, ist jedoch eine seltene Ausnahme.

Die *numerische Mathematik* bietet glücklicherweise eine leistungsfähige Alternative zur analytischen Lösungsmethode. Anstatt einen Formelausdruck für die Lösungsfunktion abzuleiten, kann mittels sogenannter *Solver* (Lösungsalgorithmen) die Differentialgleichung numerisch gelöst werden. Betrachtet werden hierzu zeitabhängige DGLs, die unabhängige Variable ist also die Zeit t, wie im Beispiel des Lotka-Volterra-Modells. Das Prinzip dabei ist, dass die Ableitung der (unbekannten) Lösungsfunktion $y_{(t)}$ zu einem bestimmten Zeitpunkt ja explizit durch die DGL selbst gegeben ist. Kennt man nun den Wert der Lösungsfunktion $y_{(t1)}$ zu einem bestimmten Zeitpunkt t_1, so kann der Wert der Funktion zu einem nachfolgenden Zeitpunkt t_2 (näherungsweise) bestimmt werden, indem man zum aktuellen Wert die Ableitung mit dem Zeitfortschritt Δt gewichtet und zu diesem hinzuaddiert.

$$y_{(t2)} \approx y_{(t1)} + \dot{y}_{(t1)} \cdot \Delta t$$

Dieser Sachverhalt wird durch nachfolgende Abbildung veranschaulicht.

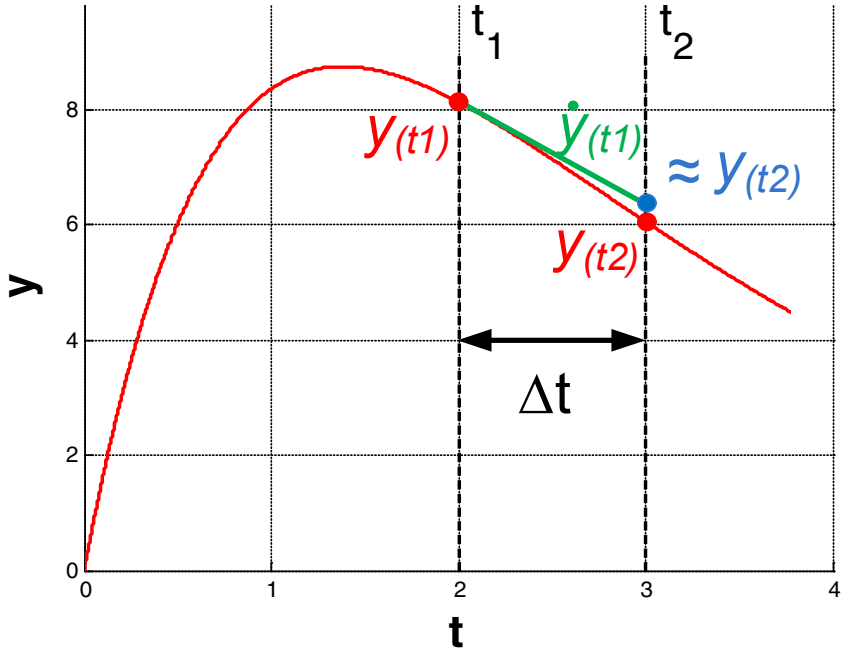


Abbildung 29: Zur numerischen Lösungen gewöhnlicher Differentialgleichungen

Da die Ableitung der Lösungsfunktion $y_{(t)}$ im Allgemeinen über den Zeitraum Δt nicht konstant ist, ist das Ergebnis $y_{(t2)}$ lediglich eine *Näherung* an den korrekten Wert der Lösungsfunktion. Dieser Fehler wird jedoch umso kleiner, je kleiner die Schrittweite Δt gewählt wird.

Auf den Wert der Lösungsfunktion zu einem späteren Zeitpunkt t_3 kann wiederum geschlossen werden, indem man über die Differentialgleichung die Änderungsrate zum Zeitpunkt t_2 (mit Hilfe des zuvor berechneten Werts $y_{(t2)}$) bestimmt und das Verfahren erneut anwendet. Auf diese Weise kann man sich Stück für Stück an der Lösungsfunktion "entlang hangeln", bzw. die Differentialgleichung *rekursiv* lösen.

$$y_{k+1} \approx y_k + \dot{y}_k \cdot \Delta t$$

Das Ergebnis liegt am Ende in Form von zwei numerischen Vektoren t und y vor. Dieses Verfahren ist gleichzeitig der einfachste numerische Solver für Differentialgleichungen, es ist das sogenannte *explizite Euler-Cauchy-Verfahren*.

Da die Anfangs- oder Randbedingungen von gewöhnlichen Differentialgleichungen zur Lösung immer gegeben sein müssen, hat man somit Startwerte, um das Verfahren zum Zeitpunkt t=0 sinnvoll beginnen, bzw. die Änderungsrate zum Zeitpunkt t=0 bestimmen zu können.

Es gibt verschiedene Typen von numerischen Solvern, die aber alle im Wesentlichen das obige Verfahren mehr oder weniger stark variieren. Matlab bietet eine ganze Familie von Solvern, die für unterschiedliche Typen von DGLs besser oder schlechter geeignet sind. Diese bilden auch den Kern des Simulationspaketes Simulink, welches im nachfolgenden Kapitel behandelt wird.

5.3 Lösung von Anfangswertproblemen mit Matlab

Zur Lösung von Anfangswertproblemen mit Matlab sind prinzipiell die folgenden Schritte notwendig:

1. Aufstellen der Differentialgleichung oder des Differentialgleichungssystem in einem Function File.

2. Aufruf des Solvers zur Lösung der Differentialgleichung mit Angabe von
   a) Den Anfangswerten
   b) Dem Start- und Endzeitpunkt der Lösungsfunktion (Integrationszeitraum)
   c) Den Systemparametern
   Optional einer Struktur, um das Lösungsverhalten des Solvers zu steuern.

Zur Lösung der Anfangswertprobleme wird hier der Matlab Solver `ode45()` verwendet. Dies ist ein Lösungsalgorithmus basierend auf dem sogenannten Runge-Kutta Verfahren, benannt nach den Mathematikern Carl Runge und Martin Wilhelm Kutta. Der Aufruf erfolgt nach folgender Syntax:

```
[t,y] = ode45(@DGL,timespan,IC,options,par1,par2,…)
```

- Funktionsargumente
  - `@DGL`: Ein Function Handle. Die entsprechende Funktion implementiert das DGL System.
  - `Timespan`: Das Berechnungsintervall, also Start- und Endzeitpunkt der Lösungsfunktion.
  - `IC`: Ein Vektor mit den Anfangsbedingungen (engl. initial conditions) der DGL
  - `options`: Eine Struktur zur Steuerung des solvers.
  - `par1, par2, …` : Die Systemparameter der Differentialgleichung.

- Rückgabewerte
  - `t`: Ein Spaltenvektor, der die numerischen Werte der unabhängigen Zeitvariablen t enthält.
  - `y`: Eine Matrix, welche die numerischen Werte der Lösungsfunktionen enthält. Die Matrix hat die gleiche Anzahl Zeilen wie der Vektor t und so viele Spalten, wie das Differentialgleichungssystem Lösungsfunktionen bzw. sogenannte *Zustandsvariablen* hat. Im Fall des Lotka-Volterra Modells also zwei Spalten: Eine Für die Funktion $N_{1(t)}$ und eine für die Funktion $N_{2(t)}$.

5.4 Beispiel: Feder-Masse-Dämpfer System

5.4.1 Systembeschreibung

Als Beispiel soll ein mechanisches Feder-Masse-Dämpfer System gemäß nachfolgender Abbildung betrachtet werden.

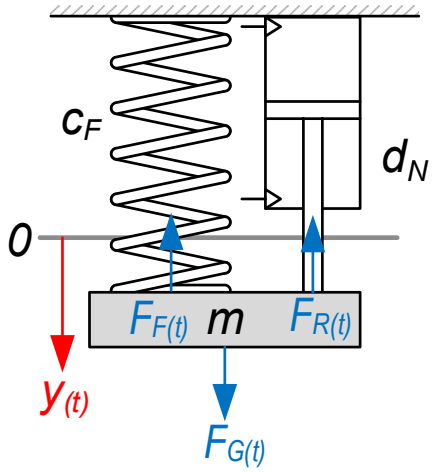


Abbildung 30: Feder-Masse-Dämpfer System (1)

An einer vertikal aufgehängten Feder-Dämpfer Anordnung sei eine Masse m befestigt. Lenkt man diese aus der Gleichgewichtslage aus und lässt sie los, dann kann diese schwingen. Die Auslenkung $y_{(t)}$, die Geschwindigkeit $y'_{(t)}$ und die Beschleunigung $y''_{(t)}$ wirken wie eingezeichnet in positiver Richtung (nach unten). Befindet sich das System nicht in der Ruhelage, so wirkt eine Federkraft entgegengesetzt der Auslenkung $y_{(t)}$. Hat das System eine Geschwindigkeit $y'_{(t)}$, so wirkt zusätzlich eine Reibkraft, ebenfalls entgegengesetzt der Bewegungsrichtung. Die auf die Masse wirkende Gewichtskraft $m \cdot g$ wirke in positiver Richtung. Die Feder ist über die Federkonstante c_F und der Dämpfer über den Reibfaktor d_N charakterisiert. Federkonstante, Reibfaktor und die Masse des schwingenden Körpers sind die konstanten Systemparameter. Es gelten folgende Zusammenhänge für die auf die Masse wirkenden Kräfte:

$$\text{Federkraft: } F_{F(t)} = -c_F \cdot y_{(t)}$$

$$\text{Reibkraft: } F_{R(t)} = -d_N \cdot \dot{y}_{(t)}$$

$$\text{Gewichtskraft: } F_{G(t)} = +m \cdot g$$

Das zweite Newtonsche Gesetz besagt, dass die auf einen Körper mit konstanter Masse wirkende Beschleunigung proportional zur Summe der angreifenden Kräfte und umgekehrt proportional zur Masse des Körpers ist:

$$\text{Beschleunigung: } \ddot{y}_{(t)} = \frac{1}{m} \cdot \sum F$$

Einsetzen der wirkenden Kräfte in die Gleichung:

$$\ddot{y}_{(t)} = \frac{1}{m} \cdot \left(F_{F(t)} + F_{R(t)} + F_{G(t)} \right)$$

Beziehungsweise:

$$\ddot{y}_{(t)} = \frac{1}{m} \cdot \left(-c_F \cdot y_{(t)} - d_N \cdot \dot{y}_{(t)} + m \cdot g \right) \qquad (1)$$

Dies ist die Differentialgleichung die das Bewegungsverhalten der Feder-Masse-Dämpfer Anordnung beschreibt. Diese Gleichung muss zu *jedem* Zeitpunkt t erfüllt sein. Neben der gesuchten Lösungsfunktion $y_{(t)}$ treten auch deren erste und die zweite Ableitung in der Differentialgleichung auf. Somit liegt eine DGL zweiter Ordnung vor. Matlab kann generell nur Differentialgleichungen oder Differentialgleichungssysteme erster Ordnung lösen. Nun ist es glücklicherweise so, dass jede Differentialgleichung n-ter Ordnung in ein System von n Differentialgleichungen erster Ordnung überführt werden kann.

5.4.2 Überführung in ein System 1.Ordnung

Wir führen hierzu ein neue Vektorvariable $z_{(t)}$ ein und sagen, dass deren erstes Element $z_{1(t)}$ gleich der Auslenkung $y_{(t)}$, und das zweite Element $z_{1(t)}$ gleich der Geschwindigkeit $y'_{(t)}$ ist.

$$z_{(t)} = \begin{bmatrix} z_{1(t)} \\ z_{2(t)} \end{bmatrix} = \begin{bmatrix} y_{(t)} \\ \dot{y}_{(t)} \end{bmatrix} \qquad (2)$$

Die Vektorvariable $z_{(t)}$ wird nun einmal nach der Zeit abgeleitet:

$$\dot{z}_{(t)} = \begin{bmatrix} \dot{z}_{1(t)} \\ \dot{z}_{2(t)} \end{bmatrix} = \begin{bmatrix} \dot{y}_{(t)} \\ \ddot{y}_{(t)} \end{bmatrix} \qquad (3)$$

Aus Gleichung (2) wissen wir, dass gilt: $y' = z_2$. Somit können wir schreiben:

$$\dot{z}_{(t)} = \begin{bmatrix} \dot{z}_{1(t)} \\ \dot{z}_{2(t)} \end{bmatrix} = \begin{bmatrix} z_{2(t)} \\ \ddot{y}_{(t)} \end{bmatrix} \qquad (4)$$

Setzen wir Gleichung (1) in Gleichung (4) ein und beachten, dass gilt $y_{(t)} = z_{1(t)}$ und $y_{(t)} = z_{2(t)}$ (Siehe Gleichung (2)), dann können wir das Differentialgleichungssystem folgendermaßen formulieren:

$$\dot{z}_{(t)} = \begin{bmatrix} \dot{z}_{1(t)} \\ \dot{z}_{2(t)} \end{bmatrix} = \begin{bmatrix} z_{2(t)} \\ \frac{1}{m} \cdot \left(-c_F \cdot y_{F(t)} - d_N \cdot \dot{y}_{F(t)} + m \cdot g \right) \end{bmatrix} \qquad (5)$$

Beziehungsweise:

$$\dot{z}_{1(t)} = z_{2(t)}$$
$$\dot{z}_{2(t)} = -\frac{c_F}{m} \cdot z_{1(t)} - \frac{d_N}{m} \cdot z_{2(t)} + g \qquad (6)$$

Dies sind zwei Differentialgleichungen erster Ordnung, welche das Bewegungsverhalten der Masse ebenso beschreiben wie Gleichung (1). Dieses Differentialgleichungssystem oder auch *Vektordifferentialgleichung erster Ordnung* kann von Matlab gelöst werden.

5.4.3 Implementierung der DGL in Matlab

Es soll nun eine Funktion *fdm.m* erstellt werden, welche dieses Differentialgleichungssystem implementiert. Die Systemparameter c_F, d_N und m sollen dieser als Parameter übergeben werden können.

```
function [dy] = fdm(t,y,cF,dN,m)
% ---------------------------------
% Differentialgleichungssystem
% der Feder-Dämpfer-Masse Anordnung
% ---------------------------------
g  = 9.81;                              % [m/s²] Erdbeschleunigung

dy = zeros(2,1);                        % Spaltenvektor generieren

dy(1) = y(2);                           % 1. Gleichung des DGL-Systems
dy(2) = g -cF/m*y(1) -dN/m*y(2);     % 2. Gleichung des DGL-Systems
```

fdm.m

Es ist darauf zu achten, dass das System als *Spaltenvektor* angelegt wird. Dieser wird deshalb zur Sicherheit mit `dy = zeros(2,1)` als solcher initialisiert.

5.4.4 Lösung der DGL in Matlab

In einem Skript *fdm_solve.m* wird nun der Lösungsalgorithmus entsprechend aufgerufen:

```
clear
clc
close all

% --- Systemparameter ---
    cF = 10;              % [N/cm]     Federkonstante
    dN = 1;               % [Ns/cm]    Reibfaktor
    m  = 2;               % [kg]       Masse

% --- Start- und Endzeitpunkt ---
    timespan = [0 30];   % [s]

% --- Anfangswerte ---
    IC = [1 0];          % [cm cm/s]

% --- Aufruf des Lösungsalgorithmus ---
    [t,y] = ode45(@fdm,timespan,IC,[],cF,dN,m);
```

fdm_solve.m

Die Systemparameter werden zu Beginn mit entsprechenden Zahlenwerten belegt. Dabei muss darauf geachtet werden, dass die physikalischen Einheiten zueinander passen.

Im Vektor `timespan` werden die Start- und Endzeitpunkte der Lösungsfunktion angegeben. Üblicherweise wird zum Zeitpunkt $t = 0s$ gestartet. Der Endzeitpunkt sagt aus, wann `ode45()` die Berechnung der Lösungsfunktion abbrechen soll. Die Berechnung endet in diesem Fall nach $t = 30s$. Alle notwendigen Zwischenschritte bzw. Zeitpunkte werden von `ode45()` selbstständig festgelegt. `ode45()` ist ein Solver mit *automatischer Schrittweitensteuerung* und versucht einerseits, die DGL an so wenig wie möglich Zeitpunkten zu lösen, aber andererseits die Schrittweite so klein zu halten, so dass der durch das numerische Näherungsverfahren zwangsläufig resultierende Fehler in einem festgelegten Toleranzband liegt.

Im Vektor `IC` stehen die Anfangswerte für die Position $y_0 = 1cm$ und die Geschwindigkeit $y'_0 = 0\ cm/s$. Der Solver `ode45()` wird mit einem Function Handle der Funktion *fdm.m* aufgerufen. Das vierte Argument ist eine leere Matrix, hier kann wahlweise die `options` Struktur eingefügt werden, mit der bestimmte Eigenschaften des Solvers (z.B. das Toleranzband für die numerische Näherungslösung) festgelegt werden können. Wird als Argument eine leere Matrix übergeben, so werden die Standardeinstellungen benutzt. Der Befehl `odeset` listet die möglichen Eigenschaften auf. Wir wollen uns an dieser Stelle jedoch nicht näher mit diesen Möglichkeiten beschäftigen.

5.4.5 Ergebnis

Wenn die Lösung des Differentialgleichungssystems erfolgreich verlaufen ist, befinden sich im Workspace nun zwei weitere Variablen `t` und `y`. `t` ist ein m-elementiger Spaltenvektor und `y` eine $mx2$-elementige Matrix. `y` enthält die Lösung der Differentialgleichung. In der ersten Spalte steht die Lösung $y_{1(t)}$, die Auslenkung der Masse und in der zweiten Spalte $y_{2(t)}$, die Geschwindigkeit der Masse. Diese Lösungsvektoren können selbstverständlich geplottet und gegebenenfalls mittels Matlab weiter analysiert werden:

```
>> Figure(1)
>> hold on
>> plot(t,y(:,1),'b','LineWidth',2)
>> plot(t,y(:,2),'r','LineWidth',2)
>> plot([t(1) t(end)],[0 0],'k--','LineWidth',2)
>> grid on
>> title('Feder-Dämpfer-Masse System','FontSize',16)
>> xlabel('t [s]','FontSize',16)
>> ylabel('y','FontSize',16)
>> h = legend('Auslenkung y_{(t)}','Geschwindigkeit y´_{(t)}',4);
>> set(h,'FontSize',12)
```

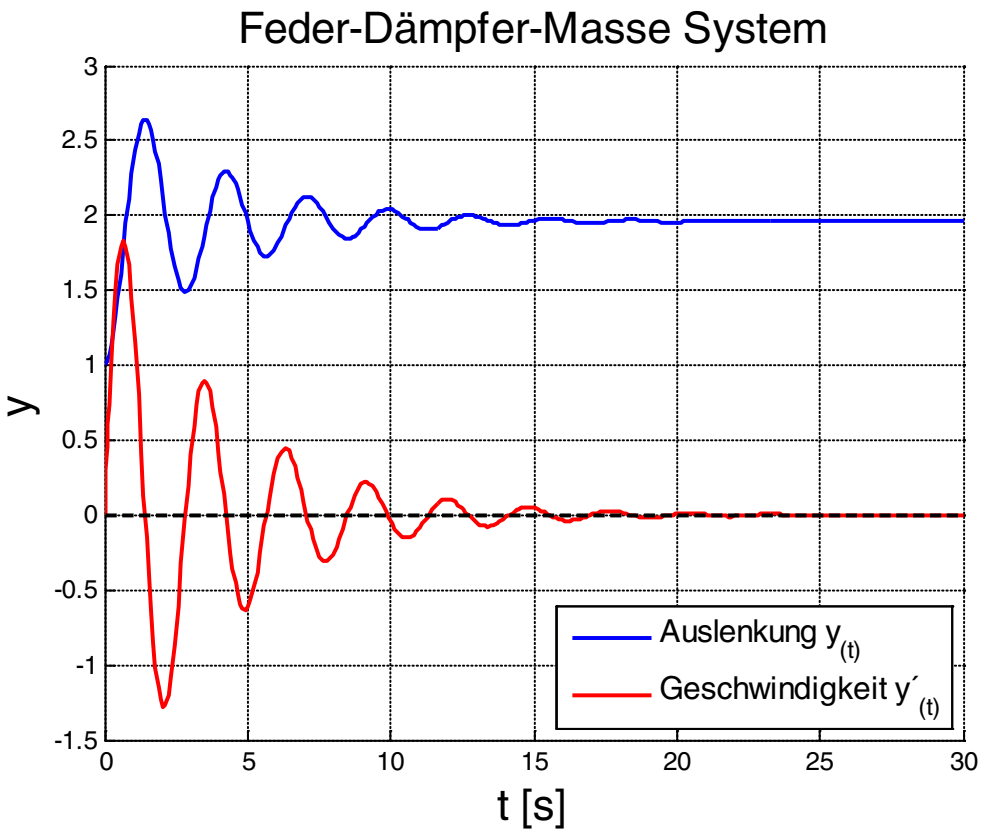


Abbildung 31: Lösung des Feder-Masse-Dämpfer Systems

6 Simulink

6.1 Einführung

Simulink ist eine Erweiterung von Matlab zur grafischen Programmierung und Lösung von auf Differentialgleichungen basierenden Systemen und Prozessen mittels *Blockschaltbildern*. Simulink greift zur Lösung der Systeme auf die gleichen Lösungsalgorithmen zu wie Matlab, also beispielsweise auf den Solver `ode45()`. Der Vorteil gegenüber der (abstrakten) textbasierten Programmierung ist, dass viele technische und naturwissenschaftliche Problemstellungen sehr anschaulich und intuitiv grafisch programmiert werden können. Der Schlüssel dazu ist das Blockschaltbild (Signalflussplan), ein Standardwerkzeug zur Modellierung/Beschreibung von dynamischen Systemen.

6.2 Blockschaltbilder

Blockschaltbilder sind in vielen Ingenieurdisziplinen weit verbreitet. Deshalb ist es wichtig Blockschaltbildern zu verstehen und lesen zu können, bevor effektiv mit Simulink gearbeitet werden kann. Ein Blockschaltbild ist im Wesentlichen nichts anderes als eine grafische Repräsentation von mathematischen Gleichungen, auch von Differentialgleichungen.

Wodurch ist eine Differentialgleichung "charakterisiert", bzw. was sind die wesentlichen "Bestandteile" einer DGL? Wir betrachten hierzu wieder die DGL des Feder-Dämpfer-Masse Systems:

$$\ddot{y}_{(t)} = \frac{1}{m} \cdot \left(-c_F \cdot y_{(t)} - d_N \cdot \dot{y}_{(t)} + m \cdot g \right)$$

Wie wir sehen, setzt sich die DGL aus einer Verknüpfung von unterschiedlichen physikalischen Größen zusammen. Dabei gibt es aus mathematischer Sicht prinzipiell zwei verschieden Arten, nämlich *veränderliche* Größen und *konstante* Größen. Die veränderlichen Größen sind alldiejenigen, die von der Zeit abhängen, die also *(t)* als Index haben. Dies sind also Funktionen in Abhängigkeit der Zeit. In unserem Fall sind dies die Auslenkung $y_{(t)}$, die Geschwindigkeit $y'_{(t)}$ und die Beschleunigung $y''_{(t)}$ der Masse. Diese über der Zeit veränderlichen Größen nennen wir im folgenden *Signale*. Die konstanten Größen c_F, d_N, m und g sind hingegen die bereits erwähnten *Systemparameter*. Im Gegensatz zu den Signalen sind dies mathematisch keine Funktionen sondern lediglich reelle Zahlen.

Signale und Systemparameter sind in der DGL über mathematische Operationen miteinander verknüpft. In vorliegenden Fall sind dies Multiplikation & Division sowie Addition & Subtraktion. Im Blockschaltbild werden Signale stets als *gerichtete Pfeile* dargestellt. Diese Pfeile werden mittels entsprechender Übertragungsblöcke zu anderen Signalen umgesetzt, wobei die Übertragungsblöcke genau den jeweiligen mathematischen Operationen entsprechen. Ein Übertragungsblock hat also immer die Eigenschaft, dass er ein oder mehrere *Eingangsignale* zu einem oder mehreren *Ausgangssignalen* umsetzt. Wir betrachten hierzu folgende einfache mathematische Gleichung:

$$y_{(t)} = 2 \cdot x_{(t)}$$

Das Signal $y_{(t)}$ ist also das Doppelte des Signals $x_{(t)}$. Ein einfaches Blockschaltbild dieses Zusammenhangs könnte nun folgendermaßen aussehen:

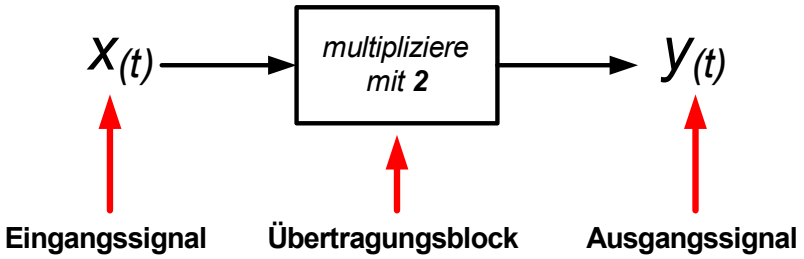


Abbildung 32: Ein- Ausgangsverhalten von Blöcken

Das Blockschaltbild ist folgendermaßen zu lesen: $y_{(t)}$ ist gleich $x_{(t)}$ multipliziert mit der Zahl 2. Man sieht, dass die Signale eine bestimmte *Wirkrichtung* haben müssen, denn durch Inversion der Pfeilrichtung ist die ursprüngliche Gleichung nicht mehr erfüllt:

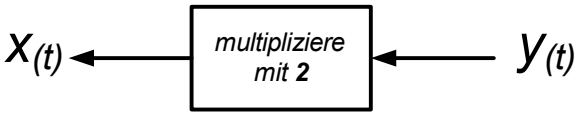


Abbildung 33: Inkorrekte Inversion der Wirkrichtung

Wird die Wirkrichtung der Signale umgedreht, so müsste auch die mathematische Operation "umgedreht" (invertiert) werden. Anstatt "multipliziere mit 2" müsste es dann "dividiere durch 2" heißen.

Übertragungsblöcke werden nun üblicherweise nicht mit Worten, sondern mit entsprechenden Symbolen gekennzeichnet. Im Folgenden werden die Übertragungsblöcke für die wichtigsten mathematischen Operationen im Zusammenhang mit Differentialgleichungen vorgestellt. Diese reichen für unsere Betrachtungen vorerst aus. Es sind dies die Übertragungsblöcke für:

1. die Addition & Subtraktion von Signalen,
2. die Multiplikation & Division von Signalen,
3. die Multiplikation & Division eines Signals mit einer Konstanten und
4. die Integration von Signalen.

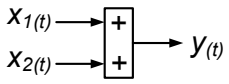
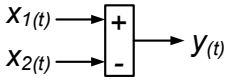
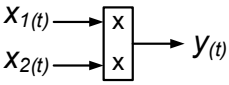
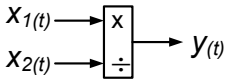
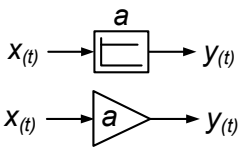
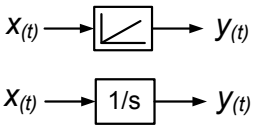
Math. Operation	Gleichung	Blockschaltbild
Addition von Signalen	$y_{(t)} = x_{1(t)} + x_{2(t)}$	
Subtraktion von Signalen	$y_{(t)} = x_{1(t)} - x_{2(t)}$	
Multiplikation von Signalen	$y_{(t)} = x_{1(t)} \cdot x_{2(t)}$	
Division von Signalen	$y_{(t)} = \frac{x_{1(t)}}{x_{2(t)}}$	
Multiplikation eines Signals mit einer Konstanten	$y_{(t)} = a \cdot x_{(t)}$	
Integration eines Signals über der Zeit	$y_{(t)} = \int x_{(t)}\, dt$	

Tabelle 19: Elementare Blockschaltbilder

Beide angegebenen Darstellungsformen der *Multiplikation eines Signals mit einer Konstanten* und der *Integration* sind zulässig und verbreitet. In Simulink wird jeweils die untere Darstellungsform verwendet. Ebenso ist es auch verbreitet, die Additions-/Subtraktionsblöcke und die Multiplikationsblöcke nicht als Quadrate sondern als Kreise darzustellen. Um Missverständnisse zu vermeiden wird bei der Multiplikation ein π Symbol in den Kreis gezeichnet. Anmerkungen: Der Ausdruck *1/s* ist die Laplace-Transformierte der mathematischen Operation *Integration*.

Mit diesen Hintergrundinformationen können wir nun das Blockschaltbild des Feder-Dämpfer-Masse Systems zeichnen.

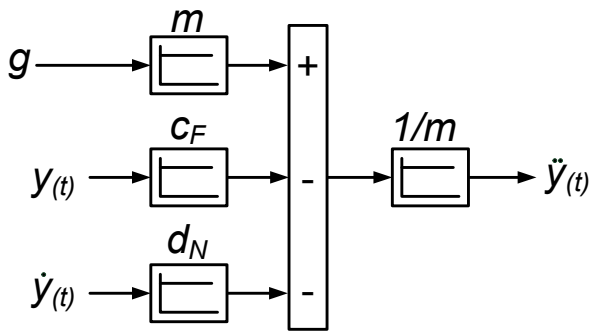


Abbildung 34: Blockschaltbild F-D-M System (1)

Wie wir sehen, wird durch das Blockschaltbild anschaulich dargestellt (erklärt), wie die Beschleunigung $y''_{(t)}$ der Masse von den restlichen Signalen $y_{(t)}$ und $y'_{(t)}$ und g abhängt. Woher kommen nun aber diese Signale, bzw. wo haben die Pfeile ihren "Ursprung"? Wie wir aus der Physik wissen, ist die Geschwindigkeit das Integral der Beschleunigung über der Zeit, und die Auslenkung wiederum das Integral der Geschwindigkeit über der Zeit. Wir erhalten diese beiden Signale also einfach durch zweifache Integration der Beschleunigung. Darüber hinaus ist die Erdbeschleunigung g ja eigentlich gar keine zeitveränderliche Größe, also kein Signal (Pfeil), sondern eine Konstante. Wir definieren daher die Erdbeschleunigung als eine konstante *Signalquelle* (ohne Eingang) und vervollständigen unser Blockschaltbild entsprechend:

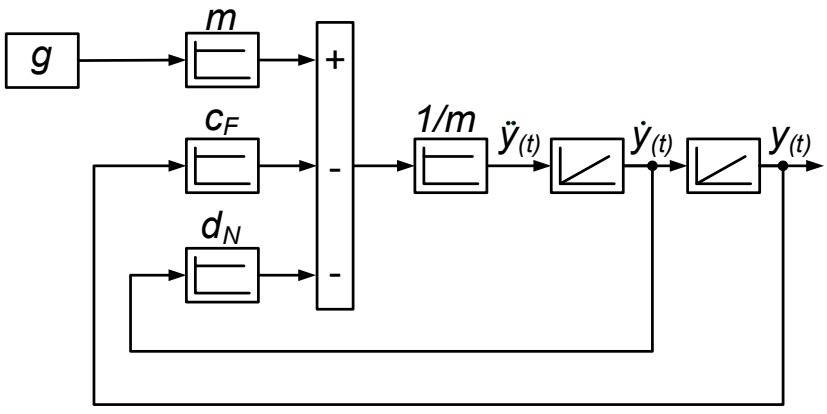


Abbildung 35: Blockschaltbild F-D-M System (2)

Dies ist die vollständige Darstellung der Differentialgleichung als Blockschaltbild. Es erklärt anschaulich wie die verschiedenen Signale zusammenwirken und gegenseitig voneinander abhängen. In einem vollständigen Blockschaltbild wird jedes auf einen Übertragungsblock einwirkende Eingangssignal durch *innere Abhängigkeiten* (Schleifen/Rückkopplungen) erklärt oder ist eine von außen einwirkende Signalquelle. Dieses Blockschaltbild kann direkt in Simulink umgesetzt werden und die Differentialgleichung numerisch gelöst werden.

6.3 Modellierung des Blockschaltbildes in Simulink

Zunächst wollen wir Simulink starten. Hierzu geben wir im Matlab Command Window entweder `simulink` ein, oder starten Simulink über den entsprechenden Button in der Toolbar im Command Window.

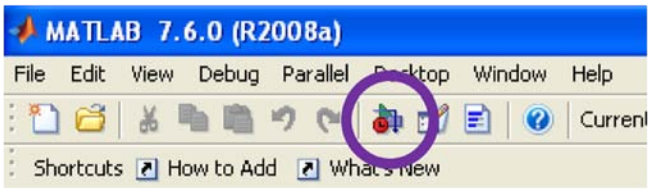


Abbildung 36: Simulink Symbol

Daraufhin öffnet sich der *Simulink Library Browser*. Simulink besitzt eine umfangreiche Bibliothek mit verschiedensten Übertragungsblöcken zur grafischen Programmierung von differentialgleichungsbasierten Systemen. Die verschiedenen verfügbaren Bibliotheken sind rechts im Library Browser in einer Baumstruktur unter *Libraries* angeordnet.

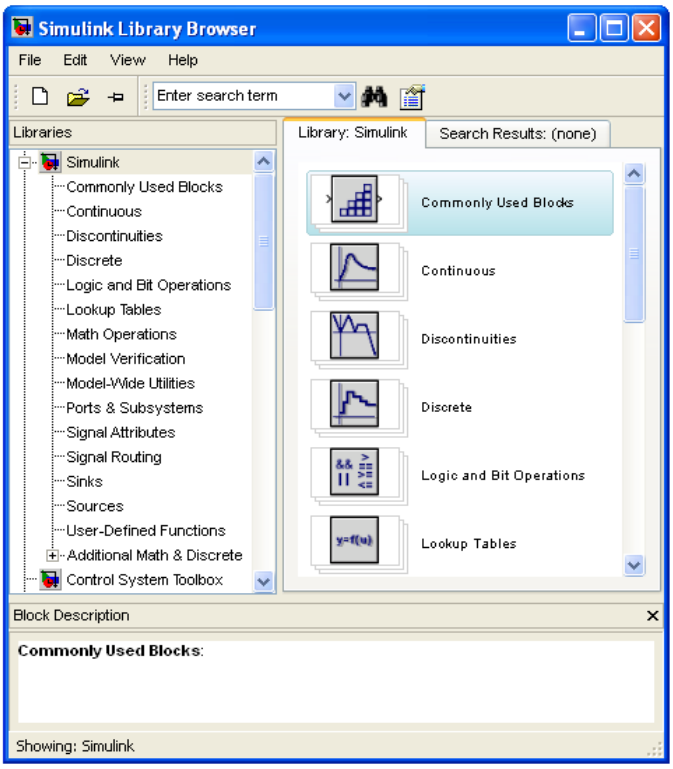


Abbildung 37: Simulink Library Browser

Um das Feder-Dämpfer-Masse System zu programmieren, müssen wir mittels *File* → *New* → *Model* zunächst eine neue Simulink Datei erstellen. Es öffnet sich ein neues leeres Fenster mit dem Titel *untitled*. Dieses speichern wir zunächst unter dem Namen *fdm_model.mdl* ab. Bei Simulink Systemen spricht man oftmals auch von Modellen, woher sich die Dateiendung *.mdl* ableitet. Auf dieses Arbeitsblatt können wir nun per Drag & Drop aus den Bibliotheken die entsprechenden Übertragungsblöcke herüberziehen. Wir benötigen:

1. Drei Blöcke zur Multiplikation eines Signals mit einer Konstanten. Dies ist der Block *Gain* aus der Bibliothek *Math Operations*.
2. Einen Summenblock. Dies ist der Block *Add* oder *Sum* aus der Bibliothek *Math Operations*.
3. Zwei Integrierer. Dies ist der Block *Integrator* aus der Bibliothek *Continuous*
4. Eine konstante Signalquelle. Dies ist der Block *Constant* aus der Bibliothek *Sources*.

Fügen Sie diese Blöcke zunächst auf dem Arbeitsblatt ein und ordnen Sie sie wie folgt an:

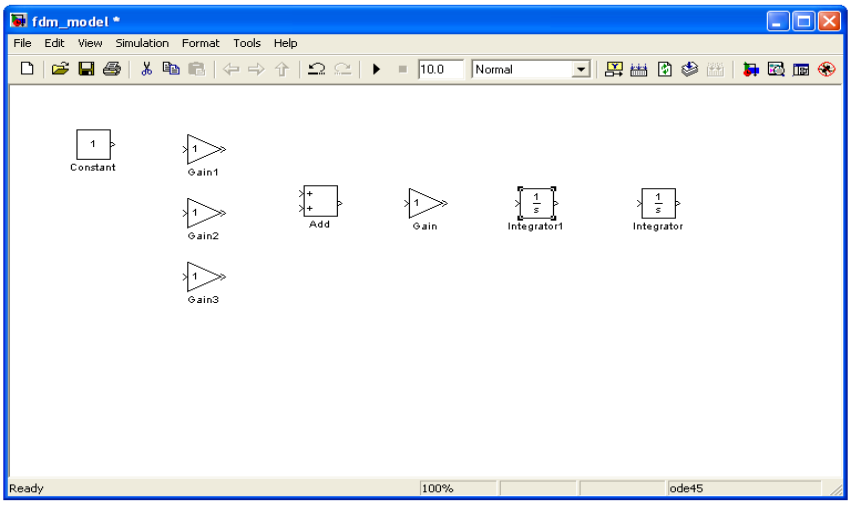


Abbildung 38: Simulink Modell des F-D-M Systems (1)

Öffnen Sie nun den *Add* Block durch Doppelklick und geben Sie im Feld *List of Signs* die Zeichenfolge + - - (Plus Minus Minus) ein. Verbinden Sie die Blöcke wie im Blockschaltbild des Feder-Dämpfer-Masse Systems. Hierzu klicken Sie den Ausgang eines Blockes mit der linken Maustaste an, halten diese gedrückt, führen die Maus zum Eingang des gewünschten Blockes und lassen die Taste los. Eine Abzweigung einer Signalleitung können sie erstellen, indem Sie die Leitung mit der rechten Maustaste anklicken. Fehlerhaft gezogene Leitungen können mit der Taste *Entf* wieder gelöscht werden, wenn diese markiert (angeklickt) sind. Vergrößern Sie den *Add* Block, indem Sie mit der linken Maustaste auf eine Ecke des Blocks klicken und diesen größer ziehen. Ihr Modell sollt dann ungefähr so aussehen:

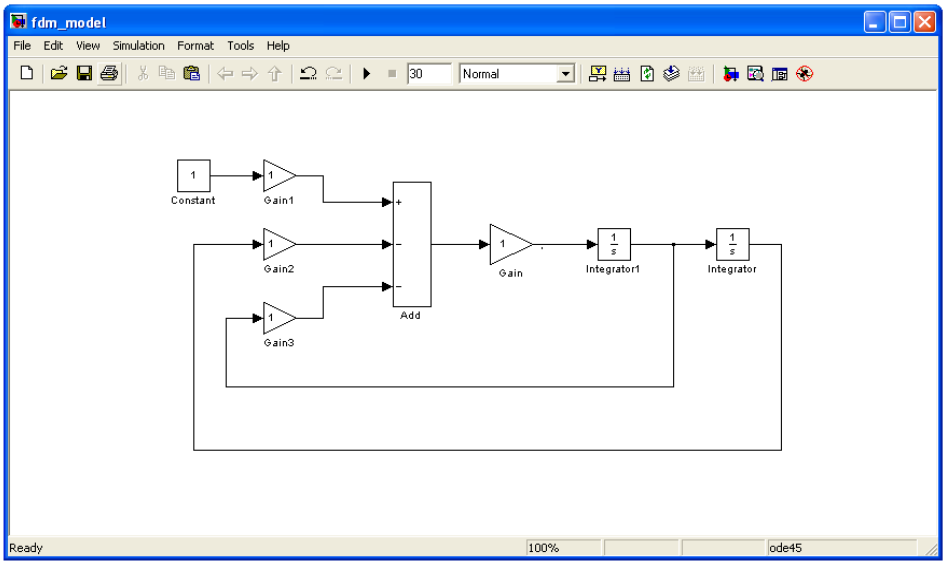


Abbildung 39: Simulink Modell des F-D-M Systems (2)

6.4 Parametrierung des Modells

Bis hierhin haben wir die *Struktur* der Differentialgleichung in Simulink umgesetzt. Im nächsten Schritt soll diese *parametriert* werden, also alle Konstanten mit den gewünschten Zahlenwerten versehen werden. Doppelklicken Sie auf den *Constant* Block und geben Sie im Feld *Constant Value* den Wert 9.81 ein. Verfahren Sie entsprechend mit den vier *Gain* Blöcken. Geben Sie hier jedoch keine konkreten Zahlen ein, sondern die Namen von Variablen: c_F, d_N, m und den Kehrwert $1/m$.

Nun sollte das Modell noch sinnvoll beschriftet werden, so dass auch für Außenstehende sofort das Blockschaltbild und die Bedeutung der Signale und Übertragungsblöcke verständlich wird. Sie können jeden Namen eines Blockes beliebig ändern, allerdings dürfen die Blocknamen innerhalb eines Modells nicht doppelt auftreten. Sie können die Signale beschriften, indem Sie auf diese doppelklicken. Vergessen sie auch nicht hin und wieder ihr Modell zu speichern.

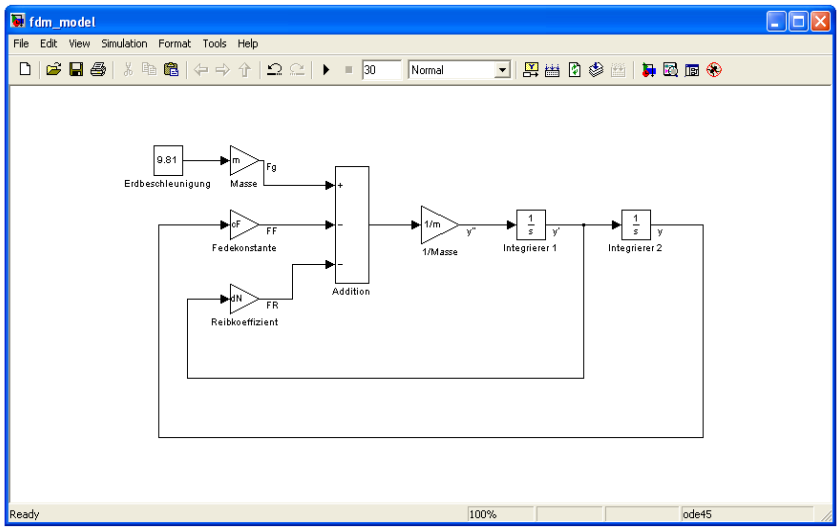


Abbildung 40: Simulink Modell des F-D-M Systems (3)

Erstellen Sie eine Skriptdatei *fdm_param.m* mit folgendem Inhalt:

```
cF = 10;                % [N/cm]     Federkonstante
dN = 1;                 % [Ns/cm]    Reibfaktor
m  = 2;                 % [kg]        Masse
IC = [1 0];             % [cm cm/s] Anfangsbedingungen
```

fdm_param.m

Die Systemparameter c_F, d_N, und m haben wir im Simulink Modell bereits in die entsprechenden *Gain* Blöcke als Variablen eingetragen. Diese Variablen im Simulink Modell werden mit den Zahlenwerten aus der Datei *fdm_param.m* verknüpft, sofern diese im Workspace liegen, sie das Skript also ausgeführt haben.

6.5 Anfangsbedingungen

Wie sieht es mit den *Anfangsbedingungen* aus? Diese haben wir zwar gerade in der Parameterdatei als Variable `IC` angelegt, aber diese noch nirgendwo im Simulink Blockschaltbild als entsprechende Variable *IC* eingetragen. Wo müssen die Anfangsbedingungen in Simulink angegeben werden?

Wir wissen aus Kapitel 5, dass wir zur Lösung des Systems Anfangsbedingungen für die Auslenkung $y_{(t)}$ und die Geschwindigkeit $y'_{(t)}$ benötigen. Wenn wir das Blockschaltbild betrachten, sehen wir, dass dies gerade die *Ausgangssignale der beiden Integrierer* sind.

Anmerkung: Ohne diese bedeutende Tatsache näher zu beleuchten gilt: Die Anfangsbedingungen einer Differentialgleichung sind immer die Ausgangssignale der im Blockschaltbild auftretenden Integrierer zum Zeitpunkt t=0. Es sind daher immer so viele Anfangsbedingungen notwendig, wie Integrierer im Blockschaltbild auftreten (= *Ordnung* des Systems).

Doppelklicken Sie auf den Block, der die Geschwindigkeit zur Auslenkung integriert. Sie sehen das Feld *Initial Condition*, tragen Sie hier den Wert `IC(1)` ein und klicken Sie auf *OK*. Die Anfangsbedingung ist nun das erste Element des zweielementigen Vektors `IC`. Verfahren Sie entsprechend mit dem zweiten Integrierer und der zweiten Anfangsbedingung `IC(2)`.

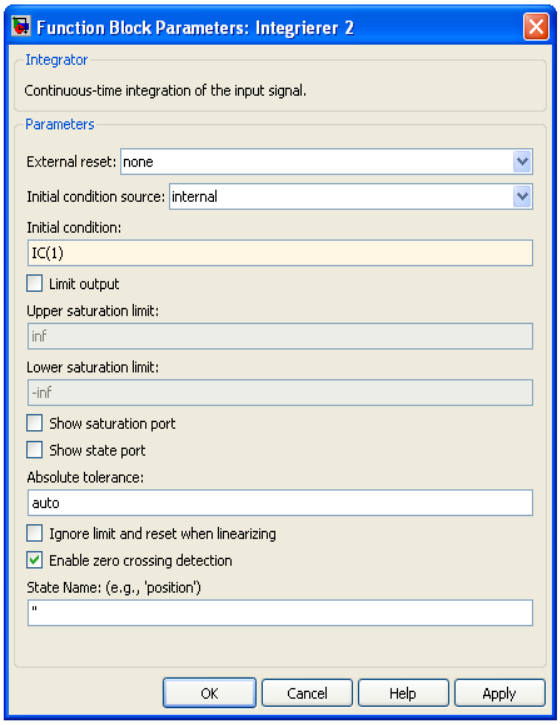


Abbildung 41: Simulink Block "Integrator"

Führen Sie das Parametrierungsskript *fdm_param.m* aus und gehen Sie sicher, dass alle im Skript definierten Variablen auch im Workspace liegen.

6.6 Parametrierung des Solvers

Als nächsten wollen wir den Solver parametrieren. Klicken Sie in der Toolbar auf *Simulation → Configuration Parameters.* Hier können etliche Einstellungen getroffen werden, um den Lösungsalgorithmus zu steuern. Wie wir sehen ist auch hier standradmäßig der Solver `ode45()` eingestellt. Wir wollen uns damit begnügen bei *Stop time* den Wert 30 und bei *Relative Tolerance* den Wert 1e-6 einzutragen.

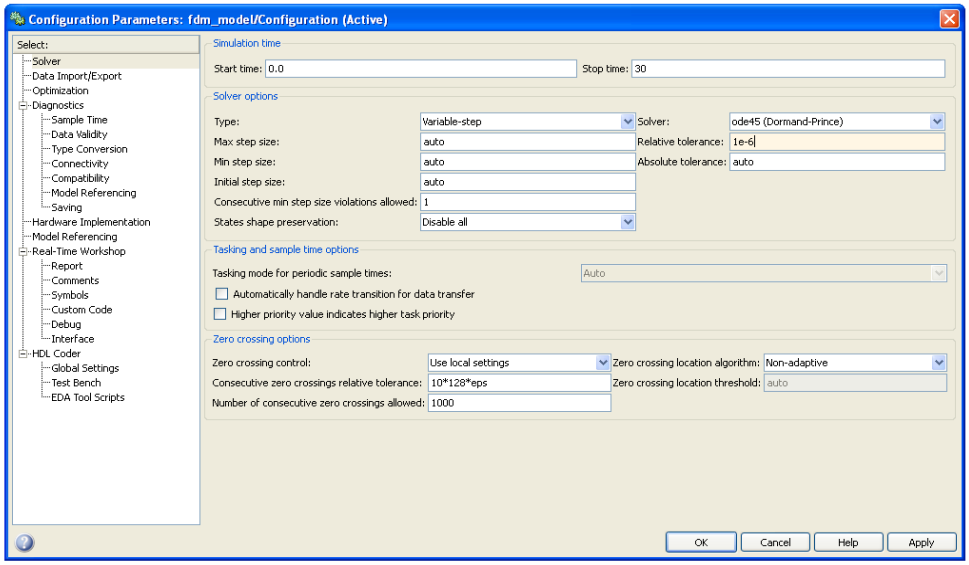


Abbildung 42: Simulink "Configuration Parameters"

Die Angabe der Parameter *Start time* und *Stop time* entspricht bei der Lösung in Matlab dem Inhalt der Variablen `timespan`. Die sonstigen *Configuration Parameters* können in Matlab über die `options` Struktur an `ode45()` übergeben werden.

6.7 Visualisierung und Weiterverarbeitung der Lösung

Nun ist das Modell im Wesentlichen fertig. Wir könnten die Simulation nun bereits ausführen, jedoch können wir das Ergebnis (die Lösung) noch nirgends betrachten bzw. auswerten und analysieren. Deswegen fügen wir noch einen *Scope Block* aus der Bibliothek *Sinks* ein. Doppelklicken Sie darauf und Sie sehen, dass sich eine Art Oszilloskop öffnet. Hier können wir später den zeitlichen Verlauf der Lösungsfunktion betrachten. Klicken Sie zunächst auf den Button *Parameters* und anschließend auf den Reiter *Data History*. Deaktivieren Sie den Checkbutton *Limit data points to last* und schließen Sie das *Scope*.

Anmerkung: Durch letzte Maßnahme wird sichergestellt, dass alle Punkte der durch `ode45()` ermittelten Lösungsfunktion auch im Scope dargestellt werden, ansonsten werden nur die 5000 letzten Punkte des Lösungsvektors gezeichnet. In unserem Fall wird der Lösungsvektor zwar deutlich weniger Elemente aufweisen, jedoch ist dies eine häufige Fehlerquelle bei der Interpretation des Simulationsergebnisses.

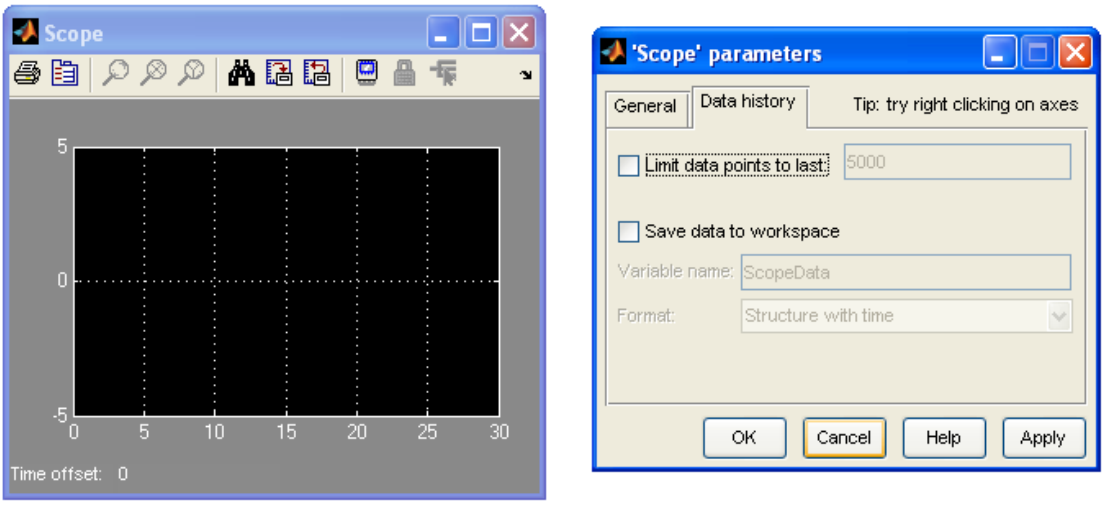


Abbildung 43: Simulink Block "Scope"

Fügen Sie nun noch einen Block *To Workspace* aus der Bibliothek *Sinks* ein. Öffnen Sie diesen und wählen Sie im *Save Format* Menü die Option *Structure with Time*.

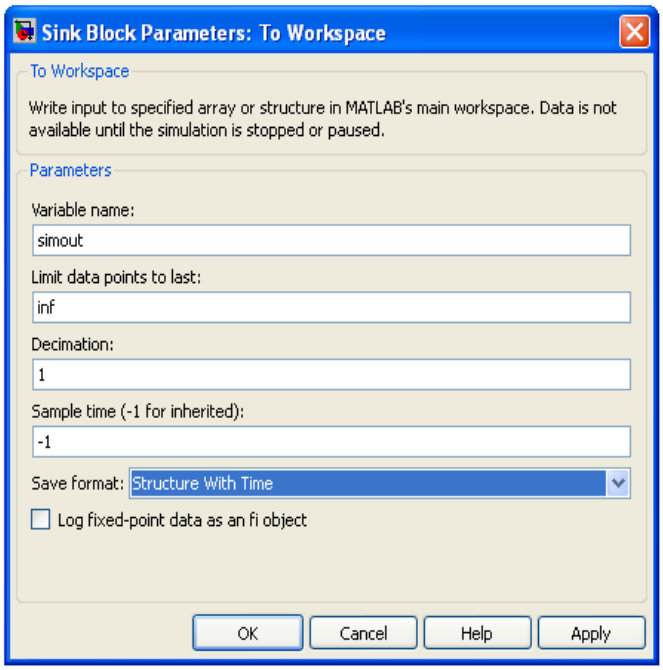


Abbildung 44: Simulink Block "To Workspace"

Fügen Sie schließlich aus der Bibliothek *Signal Routing* noch einen *Mux* Block hinzu, verbinden Sie die Blöcke wie dargestellt und speichern Sie das Modell.

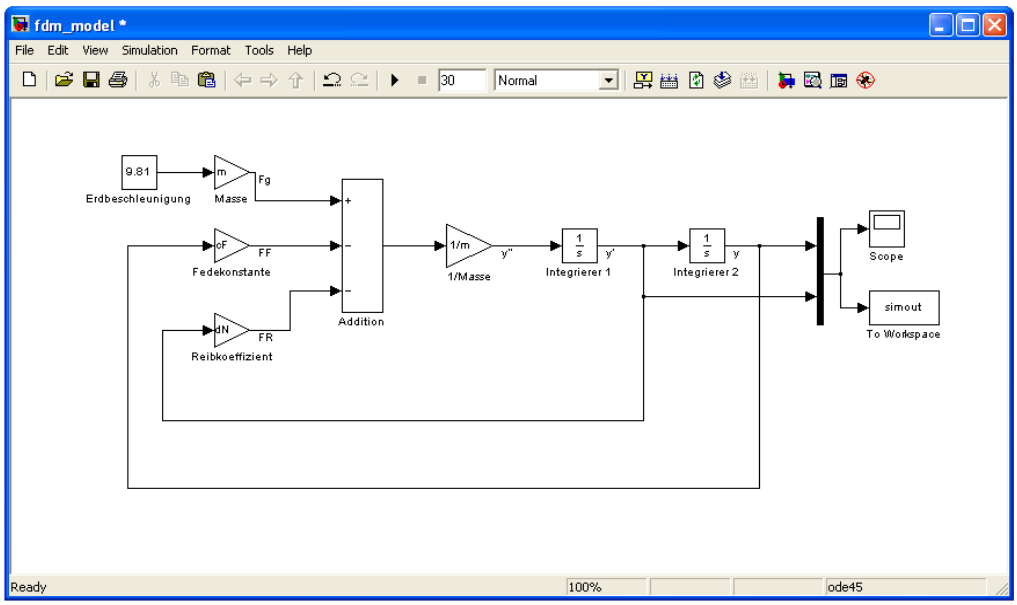


Abbildung 45: Simulink Modell des F-D-M Systems (4)

6.8 Lösung des Modells

Das Modell ist nun vollständig und kann ausgeführt bzw. gelöst werden. Drücken Sie hierzu auf das "Play" Symbol (Start Simulation) in der Toolbar. Öffnen Sie nun den Scope Block und Sie sehen die Lösungsfunktion des Differentialgleichungssystems in Abhängigkeit der Zeit:

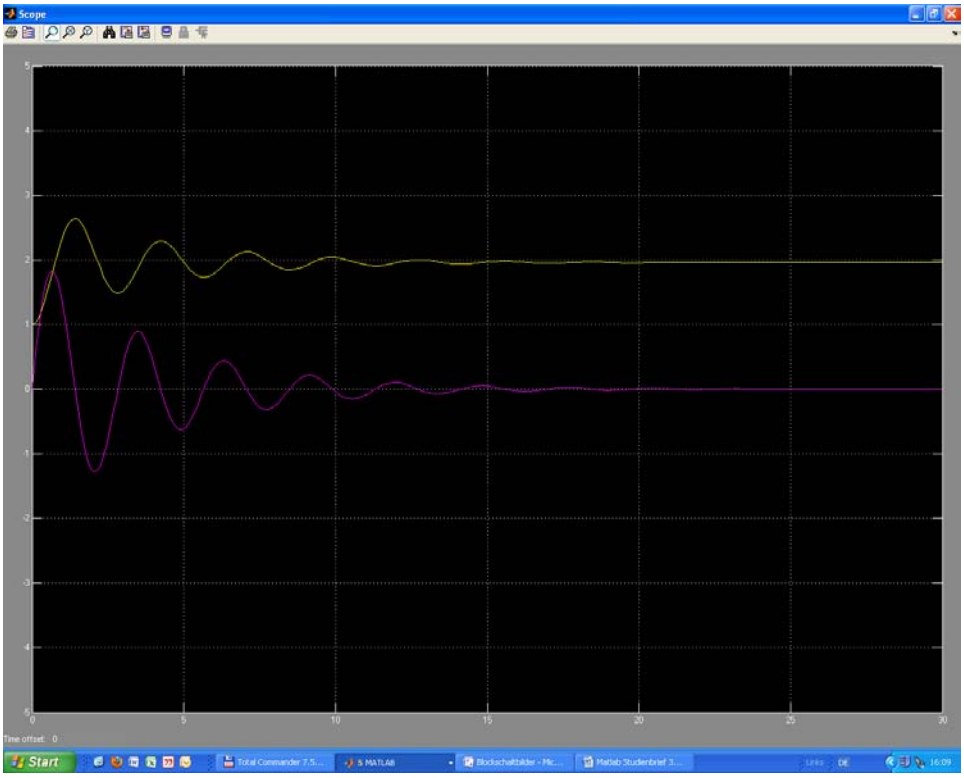


Abbildung 46: Simulationsergebnisse (1)

Gelb ist das Auslenkungssignal und violett das Geschwindigkeitssignal in Abhängigkeit der Zeit dargestellt. Da wir zusätzlich den *To Workspace* Block verwendet haben, liegt die numerische Lösung nun auch im Workspace in der Strukturvariable `simout`:

```
>> simout

simout =

            time: [161x1 double]
            signals: [1x1 struct]
            blockName: 'fdm_model/To Workspace'
```

Im Feld `time` finden Sie den Zeitvektor. In unserem Falle ein *161x1* Spaltenvektor der von 0 bis 30 Sekunden reicht (Die Länge des Vektors kann bei Ihnen u.U. abweichen). Das Feld `signals` ist wiederum selbst eine Struktur mit einem Feld `values`. Dies ist eine *162x2* Matrix. In der ersten Spalte finden Sie den ersten Kanal des *Mux* Blockes, in unserem Fall die Auslenkung $y_{(t)}$ und in der zweiten Spalte den zweiten Kanal, also die Geschwindigkeit $y'_{(t)}$.

```
>> simout.signals

ans =
      values: [161x2 double]
      dimensions: 2
      label: ''
```

6.9 Angeregte Systeme

Bisher haben wir lediglich *Anfangswertprobleme* gelöst. Dies bedeutet, dass das System ab dem Zeitpunkt t=0 sich selbst überlassen wird. Je nachdem welche Anfangsbedingungen gewählt wurden, stellen sich bestimmte Systembewegungen bzw. Lösungsfunktionen ein. Das System ist also in sich abgeschlossen, eine Einwirkung/Beeinflussung von außen findet nicht statt. Viele technische Systeme zeichnen sich aber gerade dadurch aus, dass diese von außen (gewollt) beeinflusst beziehungsweise (ungewollt) gestört werden können. Wir betrachten hierzu noch einmal das Feder-Masse-Dämpfer System und sagen nun, dass dieses zusätzlich von einer veränderlichen äußeren Kraft $F_{ext(t)}$ angeregt werden kann.

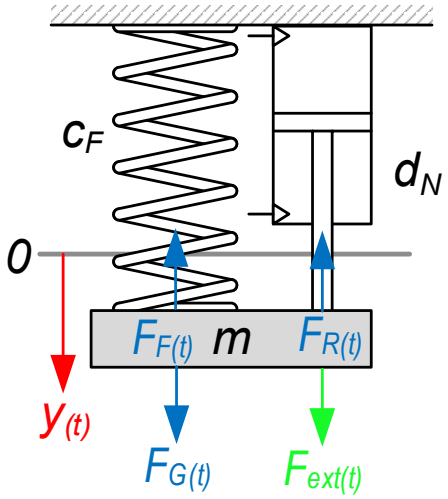


Abbildung 47: Feder-Masse-Dämpfer System (2)

Die Differentialgleichung des Systems wird entsprechend um die von außen angreifende Kraft $F_{ext(t)}$ erweitert:

$$\ddot{y}_{(t)} = \frac{1}{m} \cdot \left(-c_F \cdot y_{(t)} - d_N \cdot \dot{y}_{(t)} + m \cdot g + F_{ext(t)} \right)$$

Wir wollen nun ermitteln, wie das System auf eine solche äußere Anregung reagiert. Die im Systeminneren wirkenden Kräfte, also die blauen Signale, interessieren uns nicht. Wir können das System also etwas abstrakter auffassen:

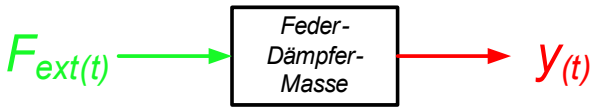


Abbildung 48: Angeregtes Feder-Dämpfer-Masse System

Da wir ja nun wissen, wie die innere Struktur des Feder-Dämpfer-Masse Systems aussieht, können wir dieses gesamte Übertragungsverhalten zu einem neuen Übertragungsblock mit dem Eingangssignal $F_{ext(t)}$ und dem interessierenden Ausgangssignal $y_{(t)}$ zusammenfassen. In Simulink kann man Übertragungsblöcke zu einem sogenannten *Subsystem* zusammenfassen. Hierzu erweitern wir unser Modell zunächst um folgende Blöcke:

1. Zwei *Out1* Blöcke aus der Bibliothek *Sinks*
2. Ein *In1* Block aus der Bibliothek *Sources*

Diese verbinden wir wie folgt in unserem Modell und beschriften diese entsprechend. Die Eingänge des *Mux* Blockes bleiben dabei zunächst offen:

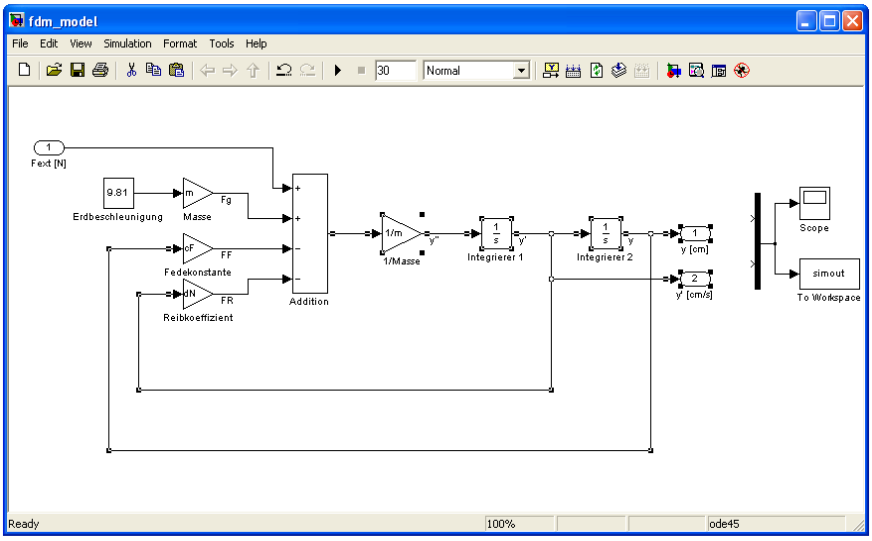


Abbildung 49: Simulink Modell des F-D-M Systems (5)

Markieren Sie nun alle Blöcke bis auf die drei Blöcke rechts (→ Rahmen ziehen mit linker Maustaste). Klicken Sie einen Block mit der rechten Maustaste an und wählen Sie aus dem aufklappenden Menü den Befehl *Create Subsystem*. Ihr Modell wird zu einem Subsystem zusammengepackt. Durch Doppelklick auf das Subsystem können Sie jedoch nach wie vor in dieses hinein blicken und jedwede Änderung vornehmen.

Entfernen Sie nun wieder die *In* und *Out* Blöcke in der oberen Modellebene und verbinden Sie die Ausgänge des Subsystems mit dem *Mux* Block. Fügen Sie einen Block *Step* aus der Bibliothek *Sources* hinzu und verbinden Sie ihn mit dem Eingang des Subsystems. Ihr Modell sollte wie folgt aussehen:

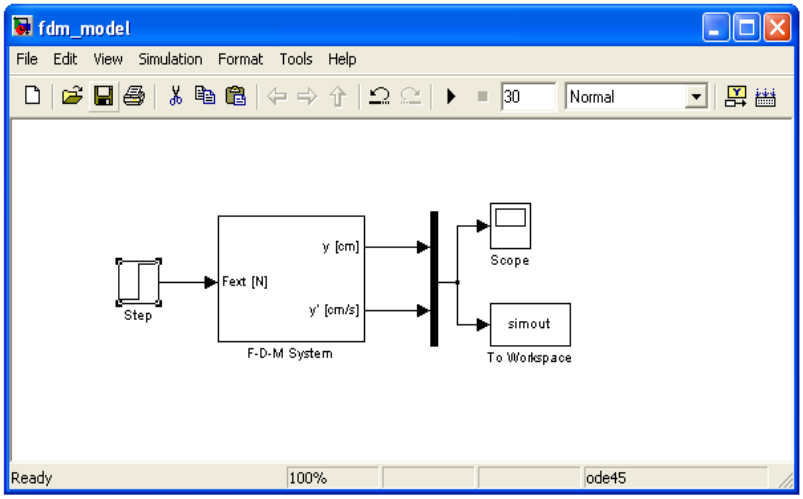


Abbildung 50: Simulink Modell des F-D-M Systems (6)

Das Zusammenfassen von Blöcken zu Subsystemen ist sehr sinnvoll, um ihrem Modell eine logische Struktur zu verleihen und um bei größeren Systemen die Übersicht zu behalten. Man sollte darauf achten, Subsysteme sinnvoll zu definieren, also nur Blöcke zusammenzufassen, die eine funktionale Einheit bilden.

In diesem Zusammenhang wollen wir gleich noch ein sinnvolles Werkzeug kennenlernen, um Systeme in Simulink zu abstrahieren: Das Prinzip der *Maskierung*. Sie können jedem Subsystem eine sogenannte *Maske* verleihen. Eine Maske ist im Wesentlichen eine Schnittstelle von der Welt außerhalb eines Subsystems zum Systeminneren. Über eine Maske können verschiedene Parameter an das System übergeben werden, dessen Inneres bleibt für einen Benutzer jedoch verborgen. Ein maskiertes Subsystem können Sie somit in etwa mit einer Funktion in Matlab vergleichen, die von jedermann mit bestimmten Argumenten aufgerufen werden kann und einen bestimmten Rückgabewert liefert. Im Allgemeinen interessiert sich ein Benutzer jedoch in der Regel nicht, was im Inneren der Funktion passiert, solange diese das korrekte Ergebnis liefert.

Klicken Sie hierzu mit der rechten Maustaste auf das System und wählen Sie *Mask Subsystem*. Es öffnet sich der *Mask Editor*. Klicken Sie zunächst auf den Reiter *Documentation* und schreiben Sie in das Feld *Mask Type* "Feder-Dämpfer-Masse System" und in das Feld *Mask Description* "implementiert die lineare DGL 2. Ordnung eines FDM-Systems".

Gehen Sie nun zum Reiter *Parameters* und klicken Sie viermal auf den Button *Add* (der nach links zeigende Pfeil), füllen Sie die erscheinenden Felder wie folgt aus und klicken Sie anschließend auf *OK*:

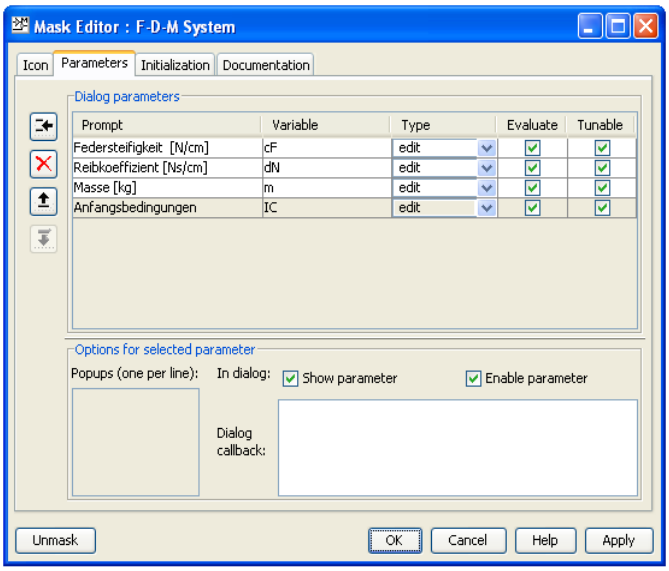


Abbildung 51: Mask Editor

Wenn Sie nun im Simulink Modell auf das Subsystem doppelklicken, wird nicht mehr das darunterliegende Blockschaltbild angezeigt, sondern die Maske geöffnet, über die Sie die entsprechenden Informationen zum Block erhalten und das System parametrieren können.

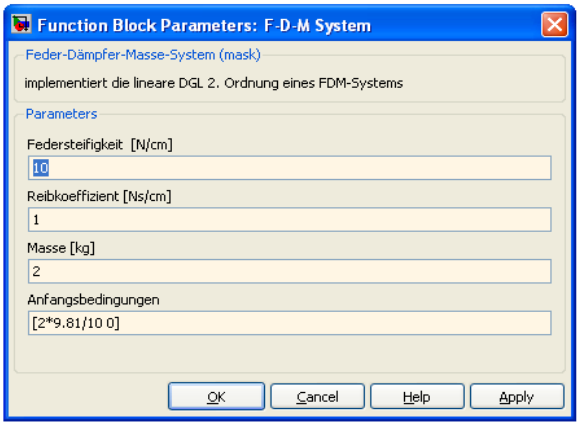


Abbildung 52: Maskiertes F-D-M System

Die Variablen c_F, d_N, m und IC, die wir im Mask Editor eingetragen haben, haben wir bereits bei der Modellierung des Systems in die entsprechenden *Gain* und *Intergator* Blöcke eingegeben. Diese Variablen werden nun mit denjenigen Werten belegt, die in die entsprechenden Felder in der Maske eingetragen werden, unabhängig davon, ob im Workspace Variablen liegen, die denselben Namen tragen. Die Variablen im Subsystem sind durch die Maskierung lokal geworden und nur dem Subsystem selber bekannt. Das gesamte Subsystem wurde sozusagen von der Außenwelt abgekapselt. Lediglich die Maske bietet eine entsprechende Schnittstelle zur Außenwelt.

Anmerkung: Woher kommt der Wert für die Anfangsbedingung "Auslenkung" im entsprechenden Feld in der Eingabemaske? Dies ist genau die Gleichgewichtslage des Systems bei der vorliegenden Parameterkonstellation, also genau die Lage, bei der das System in Ruhe ist (Beschleunigung und Geschwindigkeit = Intgriereingänge = 0) und die Gewichtskraft gerade von der Federkraft kompensiert wird, sofern keine externe Kraft wirkt:

$$\ddot{y}_{(t)} = \frac{1}{m} \cdot \left(-c_F \cdot y_{(t)} - d_N \cdot \dot{y}_{(t)} + m \cdot g + F_{ext(t)} \right)$$

$$0 = \frac{1}{m} \cdot \left(-c_F \cdot y_{stat} - d_N \cdot 0 + m \cdot g + 0 \right)$$

$$y_{stat} = \frac{m \cdot g}{c_F} = \frac{2kg \cdot 9.81 \frac{m}{s^2}}{10 \frac{N}{cm}} = 1.962cm$$

Dies wollen wir zunächst kontrollieren: Das System ist also zu Beginn in Ruhe und im Gleichgewicht. Wenn keine äußere Anregung wirkt, dürfte sich das System also nicht bewegen. Wir deaktivieren die äußere Anregung indem wir den *Step* Block öffnen und bei allen Werten 0 eintragen:

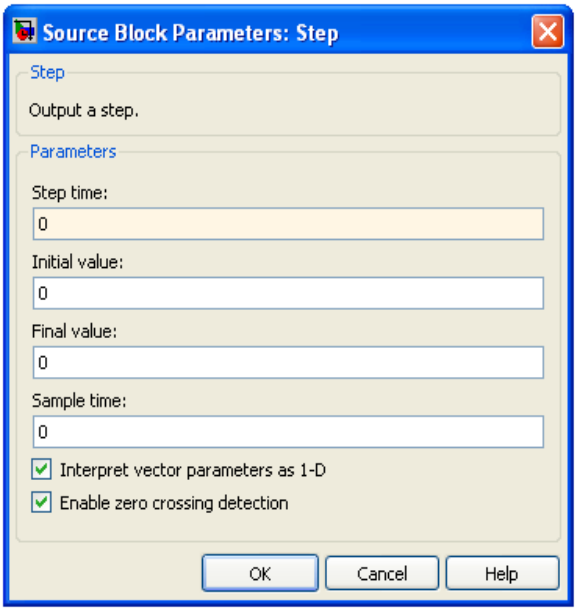


Abbildung 53: Simulink Block "Step"

Danach starten wir die Simulation und öffnen das *Scope*:

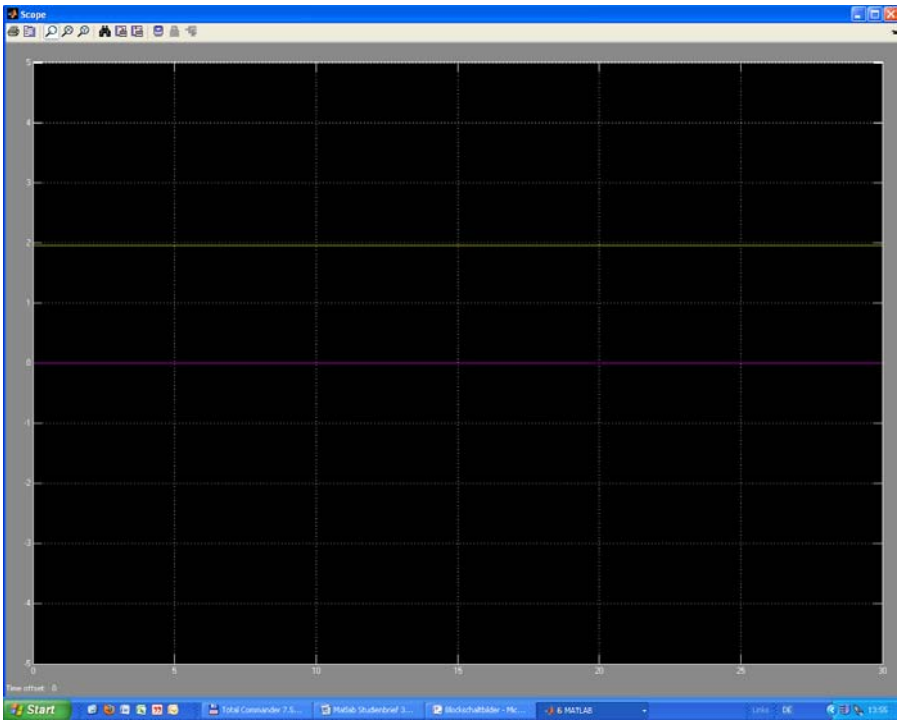


Abbildung 54: Simulationsergebnisse (2)

Beide Signale bleiben konstant und gleich der jeweiligen Anfangsbedingung. Das heißt also die Masse bewegt sich nicht. Dies ist auch logisch: Das System ist zu Beginn in Ruhe und es wirkt keine Anregung. Konsequenz: das System bleibt in dem Zustand indem es sich befindet, also in seiner Gleichgewichtslage.

Dies wollen wir nun ändern: Im *Step Block* tragen wir bei *Step Time* den Wert 5 und bei *Final Value* den Wert 20 ein. Simulieren Sie das System und öffnen sie den *Scope Block* erneut:

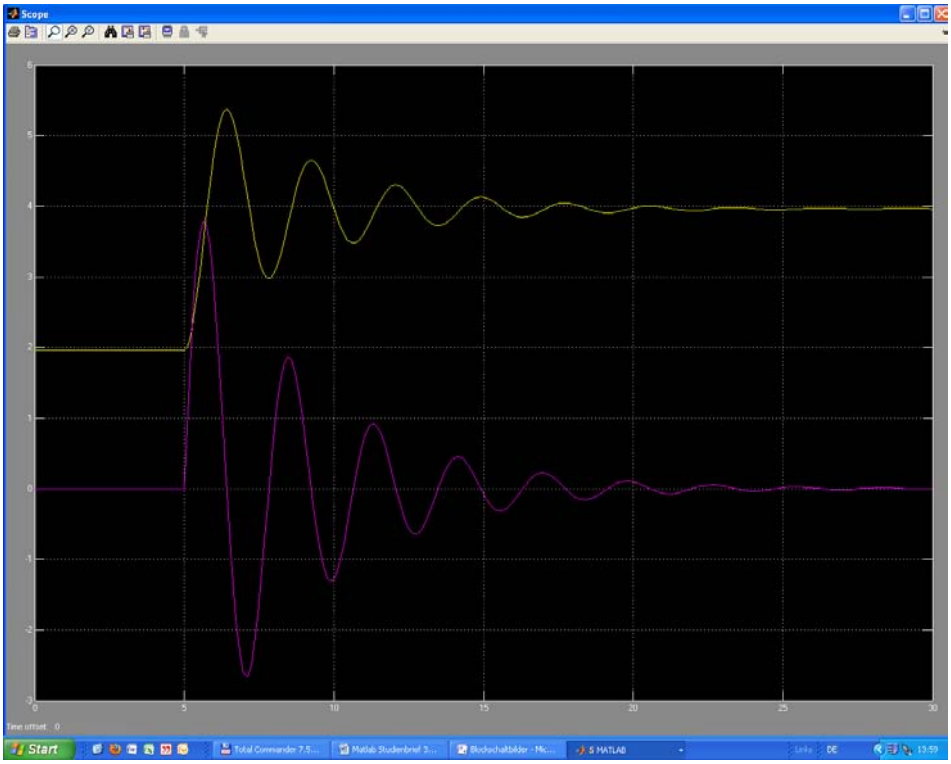


Abbildung 55: Simulationsergebnisse (3)

Das System bleibt zunächst wieder in Ruhe, nach 5 Sekunden wirkt allerdings eingangsseitig die konstante Kraft von $F_{ext(t)} =$ 20 N auf das System ein, wodurch es sich von seiner Gleichgewichtslage wegbewegt und an einer neuen Gleichgewichtslage zur Ruhe kommt. Für die neue Gleichgewichtslage gilt nun:

$$\ddot{y}_{(t)} = \frac{1}{m} \cdot \left(-c_F \cdot y_{(t)} - d_N \cdot \dot{y}_{(t)} + m \cdot g + F_{ext(t)} \right)$$

$$0 = \frac{1}{m} \cdot \left(-c_F \cdot y_{stat} - d_N \cdot 0 + m \cdot g + F_{ext,\,stat} \right)$$

$$y_{stat} = \frac{m \cdot g + F_{stat}}{c_F} = \frac{2kg \cdot 9.81 \frac{m}{s^2} + 20N}{10 \frac{N}{cm}} = 3.962cm$$

Vergleichen Sie diesen theoretischen Wert mit der stationären Endlage des Simulationsmodells. Beide Effekte, die Bewegung aus den Anfangsbedingungen heraus, die ungleich der Gleichgewichtslage ist, sowie die Bewegung aufgrund einer äußeren Anregung können wir nun überlagern. Hierzu geben wir als Anfangsbedingung in der Maske den Vektor `[0 0]` vor (Anfangsposition und Anfangsgeschwindigkeit = 0) und wählen im *Step* Block bei *Step Time* den Wert 10. Wenn wir das System simulieren erhalten wir folgende Lösung:

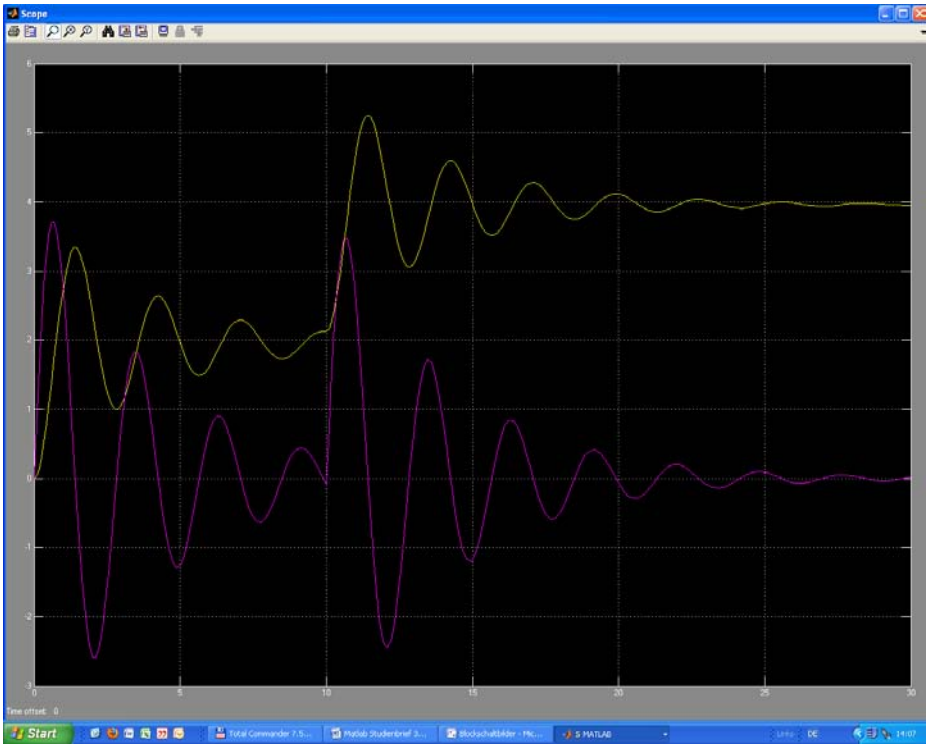


Abbildung 56: Simulationsergebnisse (4)

Sie sehen wie leicht es ist, dynamische Systeme mithilfe von Simulink zu simulieren und entsprechende Änderungen an der Parametrierung oder Modellstruktur vorzunehmen und somit das Systemverhalten entsprechenden Untersuchungen zu unterziehen.

7 Graphical User Interfaces

Grafische Benutzeroberflächen (engl.: Graphical User Interfaces, kurz GUIs) dienen zur einfachen und intuitiven Benutzung eines Computerprogramms. Beispiel: Das standardmäßig in Matlab enthaltene Programm *Taylor Tool* mit grafischer Benutzeroberfläche (Aufruf über den Befehl `taylortool`):

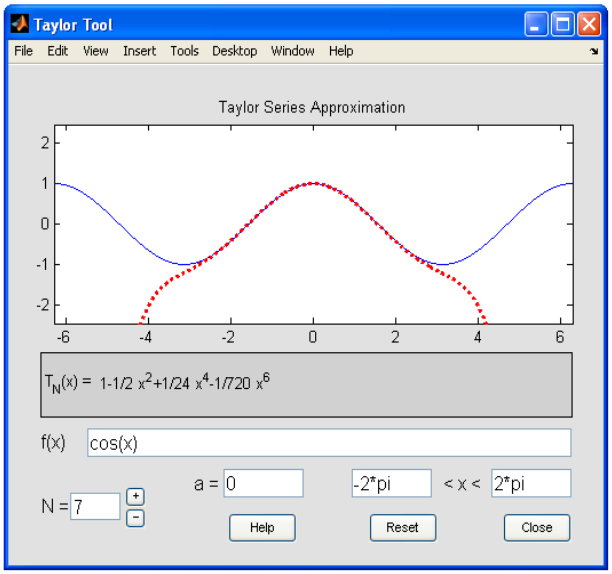


Abbildung 57: Matlab Programm "Talyor Tool"

GUIs habe stets zum Ziel, die Nutzung von Funktionen oder ganzen Programmen zu vereinfachen, so dass ein Anwender ohne Kenntnis des Quellcodes oder interner Programmstrukturen die bereitgestellten Funktionalitäten einfach nutzen kann. GUIs sollten, daher möglichst intuitiv bedienbar sein. Ein Beispiel hierfür ist das Betriebssystem Microsoft Windows, das es de facto jedem Computeranwender erlaubt, per Mausklick auch auf komplexe Funktionalitäten des Betriebssystems zuzugreifen oder Systemeinstellungen zu ändern.

Es ist einiges an Erfahrung und know-how notwendig, um intuitiv zu bedienende, leicht wart- und erweiterbare, stabile und effiziente Benutzeroberflächen zu schreiben. Gute GUIs sind zudem beispielsweise tolerant gegenüber fehlerhaften Benutzereingaben, erkennen und korrigieren diese selbstständig oder geben dem User zumindest Rückmeldung, dass seine Eingaben unbrauchbar sind. Schlechte GUIs verarbeiten jede Benutzereingabe ohne Kontrolle, was bei fehlerhaftem Input zu unerwarteten Ergebnissen oder gar Programmabstürzen führen kann. Im Rahmen dieses Einführungskurses können die theoretischen Grundlagen von GUIs und der damit eng verbundenen objektorientierten Programmierung nicht tiefschürfend behandelt werden. Jedoch lassen sich auch gerade mit Matlab sehr schnell und mit einfachen Mitteln beachtliche Ergebnisse erzielen bzw. mächtige Benutzerschnittstellen erstellen. In diesem Rahmen soll anhand eines kleinen Projektes beispielhaft ein Programm mit grafischer Oberfläche erzeugt werden. Die wesentlichsten Programmiertechniken und Vorgehensweisen werden dabei Schritt für Schritt vorgestellt.

7.1 Grundlegende Konzepte

GUIs arbeiten meist nach dem Prinzip, dass durch eine Eingabe von außen, in der Regel Maus- oder Tastaturbefehle, eine bestimmte Aktion ausgeführt werden soll. Durch die Aktion werden bestimmte Programmabläufe und/oder Berechnungen angestoßen und entsprechende Ergebnisse zurückgemeldet, in der Regel wieder auf der Grafischen Benutzeroberfläche, also auf dem Bildschirm.

7.1.1 GUI Elemente

Matlab stellt im Wesentlichen elf Standardkomponenten zur funktionalen Gestaltung einer grafischen Benutzeroberfläche bereit. Diese sollen im Folgenden kurz vorgestellt werden.

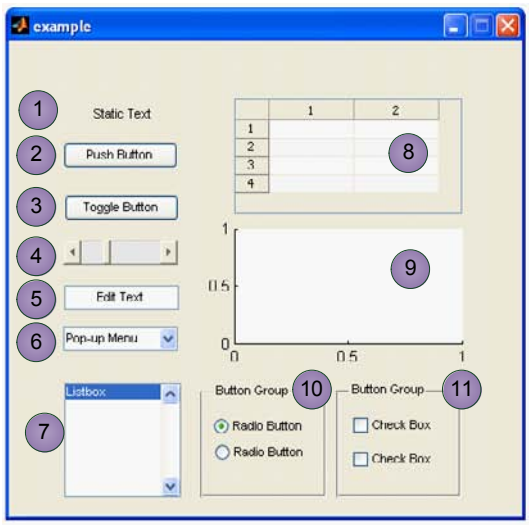


Abbildung 58: GUI Elemente

1. *Static Text*
   Ein *Static Text* Feld, dient lediglich zur Anzeige einer Zeichenkette auf der Oberfläche. Es hat ansonsten keine weitere Funktionalität.
2. *Push Button*
   Beim Drücken eines *Push Buttons* wird einmalig eine Aktion ausgeführt. Der Schalter springt danach wieder in seine Ausgangsposition zurück.
3. *Toggle Button*
   Mit einem *Toggle Button* kann eine An/Aus bzw. Ja/Nein Funktionalität realisiert werden. Nach Drücken des Buttons verbleibt dieser in seiner Position bis dieser erneut gedrückt wird. Je nach Zustand (gedrückt oder nicht gedrückt) hat der *Toggle Button* also einen Werte 0 oder 1, der ausgewertet werden kann und somit der Programmfluss gesteuert werden kann.
4. *Slider*
   Ein *Slider* ist ein Schieberegler, der je nach Stellung einen numerischen Wert zwischen einem minimalen Wert (unterer Anschlag) und einen maximalen Wert (oberer Anschlag) annehmen kann.
5. *Edit Text*
   In ein *Edit Text* Feld kann eine beliebige Zeichenkette eingegeben werden. Diese kann ausgelesen und weiterverarbeitetet werden.
6. *Pop-Up Menu*
   Beim Anklicken eines *Pop-up Menus* klappt eine Liste mit verschiedenen Einträgen auf, von denen einer ausgewählt werden kann.
7. *Listbox*
   Eine *Listbox* stellt verschiedene Auswahlmöglichkeiten anhand einer Liste zur Verfügung, von denen eine oder auch mehrere ausgewählt (aktiv) sein können.
8. *Tabelle*
   In die Zellen einer *Tabelle* können ebenfalls Zeichenketten eingegeben werden. Der Vorteil gegenüber den *Edit Text* Feldern ist, dass alle Zellen simultan ausgelesen werden können und der Inhalt automatisch in einer Zellvariable gespeichert wird. Somit kann sehr einfach auf die verschiedenen Zellen der *Tabelle* zugegriffen werden.
9. *Axes*
   Axes dienen als "Container" zur Darstellung von Plots.
10. *Radio Button*
    Radio Buttons machen nur Sinn, wenn mindestens zwei oder mehrere davon in einer sogenannten *Button Group* zusammengefasst sind. Einerseits kann immer nur ein *Radio Button* aktiv sein, dessen Wert dann 1 beträgt, ansonsten 0, andererseits muss aber auch immer ein *Radio Button* aktiv sein.
11. *Check Box*
    Check Boxes arbeiten ähnlich wie *Radio Buttons*, allerdings können auch mehr als eine *Check Box* gleichzeitig aktiv sein, bzw. auch gar keine.

7.1.2 GUI Dateien und Handles Structure

In Matlab besteht ein GUI im Kern immer aus *zwei* Dateien, einem Function File und einem Figure File, die denselben Namen tragen, jeweils mit der Endung .m beziehungsweise .fig. Das Figure File verwaltet hierbei alle *grafischen Elemente* (Objekte) und deren Eigenschaften (properties) wohingegen in dem M-File die *Funktionalität* der Benutzeroberfläche programmiert wird.

Alle Eigenschaften und alle Daten/Zustände der grafischen Elemente werden in einer *Handles Structure* namens `handles` gespeichert und verwaltet. Das Handles Konzept nimmt bei der Programmierung von GUIs eine zentrale Rolle ein.

7.1.3 Callback Functions

Jedes Grafische Element besitzt eine sogenannte *Callback Function*. Diese Funktion wird aufgerufen, sobald der Benutzer eine Eingabe oder Veränderung an dem jeweiligen grafischen Element vornimmt, also zum Beispiel einen Wert in ein *Edit Text* Feld eingibt oder einen *Push Button* betätigt.

7.1.4 Guide

Die Programmierung des grafischen Anteils eines GUIs, also des Figure Files, kann in Matlab prinzipiell auf zwei Arten erfolgen: Zum einen kann jedes grafische Element textuell über entsprechende Befehle programmiert werden. Hierbei sind dann alle Eigenschaften, wie zum Beispiel Größe, Position, Farbe, Schriftgröße,… von Hand zu programmieren. Diese Methode ist sehr aufwändig und fehleranfällig, gerade für Neulinge in der GUI Programmierung. Deshalb stellt Matlab mit *GUIDE* (Graphical User Interface Design Environment) ein Werkzeug zur Verfügung, mit welchem die grafischen Elemente per Drag & Drop auf einem Arbeitsblatt platziert werden können und somit das Layout des GUIs erstellt werden kann. Alle Eigenschaften der grafischen Elemente können über den sogenannten *Property Inspector* eingestellt werden können.

7.2 Beispielprojekt: Funktionsplotter

Wir wollen nun ein einfaches Programm mit grafischer Benutzeroberfläche erstellen und dabei die wesentlichen Programmierkonzepte für GUIs Schritt für Schritt kennenlernen. Dabei ist es überaus wichtig, dass man sich vor dem Beginn der Programmierung folgendes gründlich überlegt:

1. Festlegung der Funktionalität: Was muss mein Programm können, und (genauso wichtig) was muss es nicht können?
2. Festlegung des Designs: Wie soll meine grafische Benutzeroberfläche aussehen, welche Komponenten benötige ich hierzu und wie sollen diese angeordnet sein, so dass das Programm möglichst einfach und intuitiv zu bedienen ist?

Diese Fragen sollen nun zunächst beantwortet werden. Wir möchten hier im Wesentlichen ein Programm erstellen, das beliebige vom Benutzer zu definierende mathematische Funktionen $y = f_{(x)}$ zeichnet und einige Zusatzinformationen ausgibt.

7.2.1 Festlegung der Funktionalität

Der genaue Umfang der Programmfunktionalität soll wie folgt aussehen:

1. Der Benutzer soll über ein Eingabefeld eine beliebige Funktion $y = f_{(x)}$ als Zeichenkette eingeben können.
2. Diese Zeichenkette wird ausgewertet und die Funktion entsprechend geplottet.
3. Über zwei Eingabefelder kann der Benutzer einen Wert x_{min} und einen Wert x_{max} festlegen. Die Funktion soll genau in diesem Intervall gezeichnet werden.
4. Mit einem Schieberegler kann man einen beliebigen Arbeitspunkt x_A im Intervall zwischen x_{min} und x_{max} festlegen.
5. Der Arbeitspunkt $y_A = f_{(xA)}$ soll ebenfalls in dem Funktionsplot als Markierung angezeigt werden.
6. Der Arbeitspunkt x_A wird in einem Textfeld als Zahlenwert ausgegeben.
7. Über eine Auswahlmöglichkeit kann der Benutzer wählen, ob in einem weiteren Textfeld der Funktionswert $y_A = f_{(xA)}$ oder der Wert der Ableitung $y'_A = f_{(xA)}$ angezeigt wird.
8. Es wird davon ausgegangen, dass alle Benutzereingaben sinnvoll und logisch richtig sind. Weder die Zeichenkette in der die Funktion $y=f_{(x)}$ definiert wird, noch die Eingabefelder für x_{min} und x_{max} werden auf logische oder syntaktische Richtigkeit überprüft.

7.2.2 Festlegung des Designs

Nachdem die Funktionalität festgelegt wurde, kann man sich mit diesen Informationen Gedanken darüber machen, wie die Benutzeroberfläche sinnvollerweise gestaltet werden soll. Hierbei sollte man zunächst dieses Layout in Form einer groben Skizze zu Papier bringen. Die Grafische Benutzeroberfläche sollte anschließend nach dieser Vorlage umgesetzt werden. Ausgehend von der zuvor festgelegten Programmfunktionalität könnte eine mögliche grafische Umsetzung der Bedienoberfläche ungefähr so aussehen:

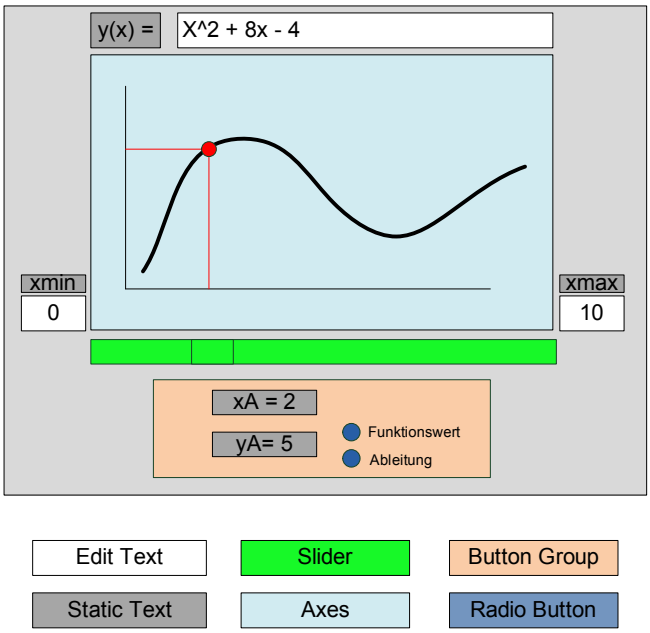


Abbildung 59: Skizze zum GUI Layout

Machen Sie sich den Aufbau dieser "Oberfläche" im Zusammenhang mit den acht zuvor festgelegten Punkten der Programmfunktionalität klar.

7.2.3 Programmierung der grafischen Oberfläche

Es soll nun zunächst die grafische Oberfläche auf Grundlage der Skizze erstellt werden. Öffnen Sie das Design Tool GUIDE, indem Sie `guide` im Command Window eingeben und wählen Sie bei dem sich öffnenden Fenster *Blank GUI (Default)* und drücken Sie *OK*. Es öffnet sich ein Arbeitsblatt mit dem Titel *untitled1.fig*.

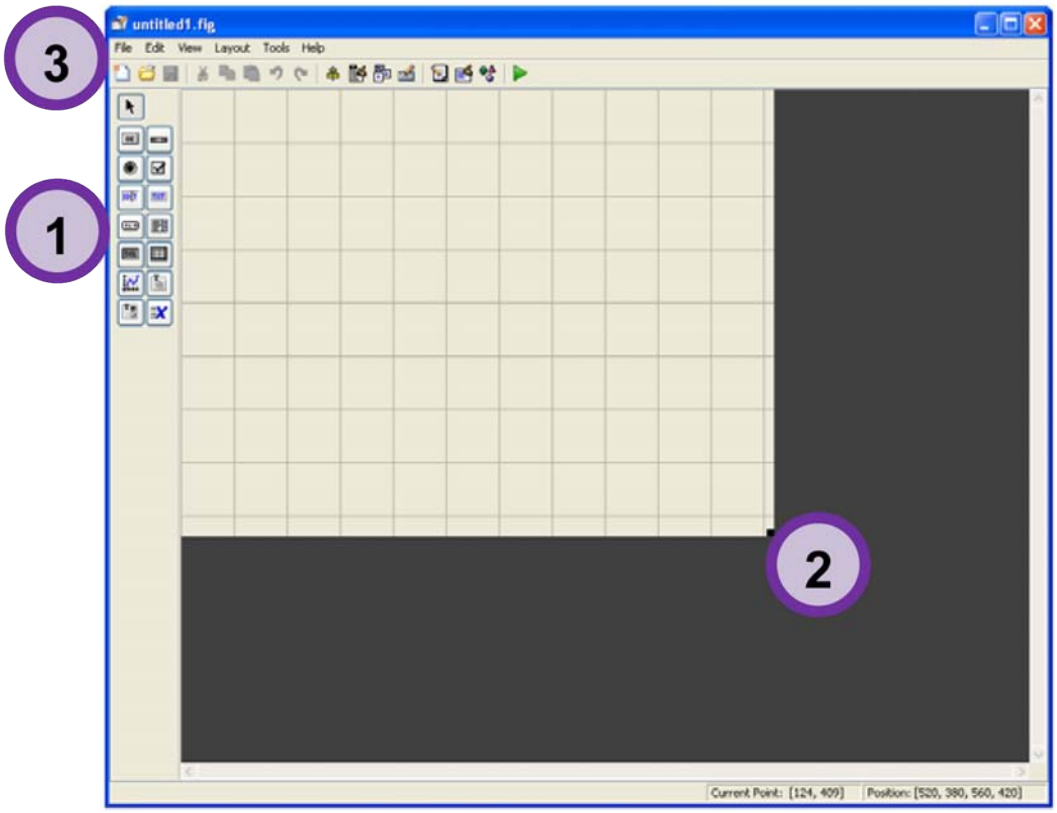


Abbildung 60: GUIDE Layout (1)

Aus der Leiste links (1) können die verschiedenen grafischen Komponenten ausgewählt und auf dem Arbeitsbereich per Drag & Drop platziert werden. Über die rechte untere Ecke des Arbeitsblattes (2) kann die endgültige Größe des Figure Windows festgelegt werden, also die Größe wie das GUI nachher auf dem Bildschirm erscheinen wird. Über das File Menü (3) kann das Figure File abgespeichert werden, was wir zunächst tun wollen. Speichern Sie das File unter dem Namen *funcplot.fig*. Sie sehen, dass automatisch ein zweites File namens *funcplot.m* generiert wird und im Editor geöffnet wird.

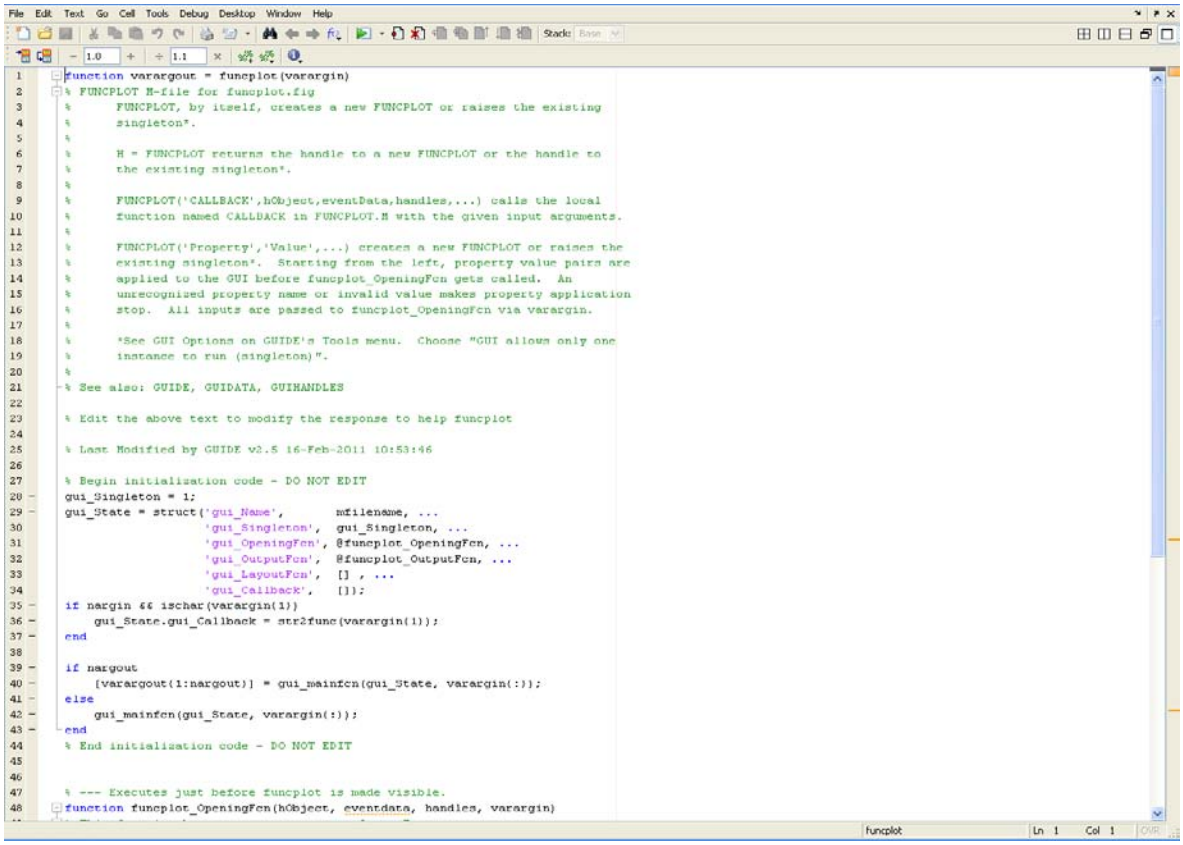


Abbildung 61: GUI Function File

Dieses Function File beinhaltet bereits einige Zeilen von Matlab *automatisch generierten Code*, der notwendig ist, um das Programm als GUI auszuführen. Dieses File wird von Matlab automatisch aktualisiert wenn Sie das Figure File in GUIDE abspeichern und es werden automatisch neue Funktionen hinzugefügt, wenn Sie grafische Elemente in GUIDE hinzufügen. Beide Files, das Figure File und das Function File, sind untrennbar miteinander verknüpft und sind notwendig, um das Programm später auszuführen. Wir wollen uns jedoch zunächst nicht um das Function File kümmern und kehren zu GUIDE zurück.

Fügen Sie auf dem Arbeitsblatt zunächst die notwendigen GUI Komponenten per Drag & Drop ein und ordnen Sie diese wie in nachfolgender Abbildung an. Achten Sie darauf, dass Sie erst die *Button Group* erstellen und anschließend die beiden *Radio Buttons* innerhalb der *Button Group* platzieren.

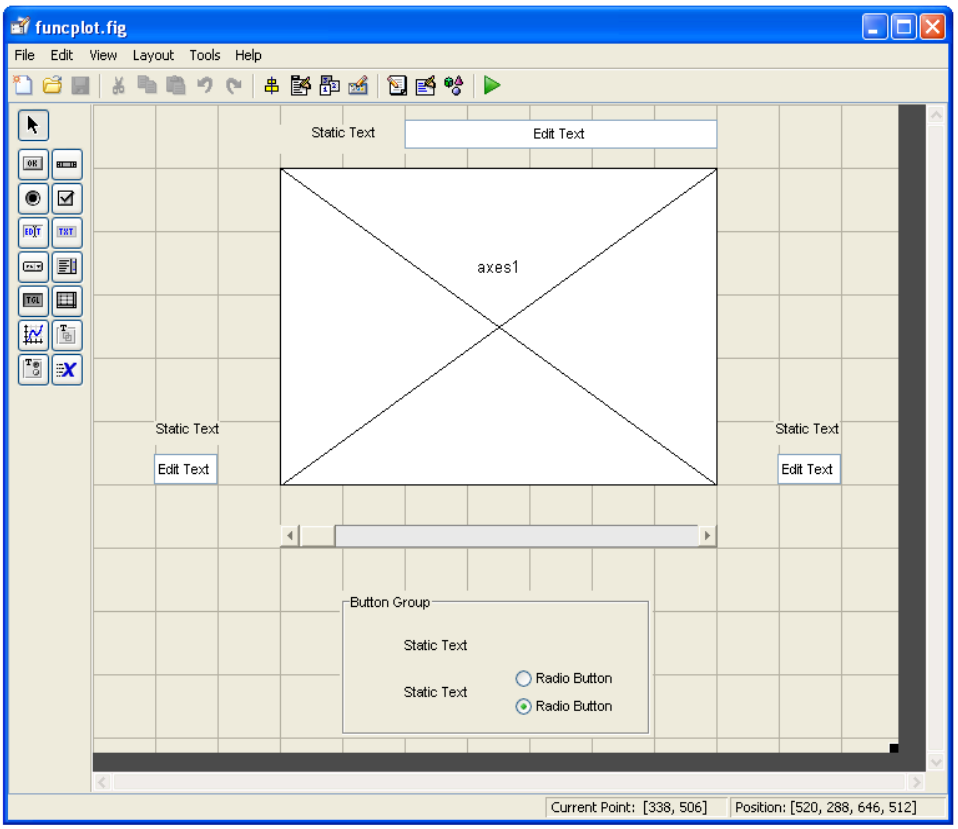


Abbildung 62: GUIDE Layout (2)

Speichern Sie das File ab. Sie können das Programm nun bereits ausführen, indem Sie auf den grünen Pfeil in der Toolbar klicken. Es öffnet sich ein Fenster mit allen grafischen Elementen, die Sie auf dem Arbeitsblatt eingefügt haben.

Natürlich ist das Programm aber bisher noch ohne "Inhalt". Sie können zwar beispielsweise in die *Edit Text* Felder etwas eintragen oder den *Slider* bewegen, aber diese Aktionen haben keine Konsequenzen. Den Programminhalt erstellen wir später. Zunächst wollen wir aber das Figure File in GUIDE weiter bearbeiten.

Doppelklicken Sie auf das obere *Static Text* Feld, es öffnet sich der *Property Inspector*, der alle Eigenschaften des Feldes anzeigt. Setzen Sie hier die Eigenschaft *Font Size* auf den Wert 12, tragen Sie bei der Eigenschaft *String* den Text *y(x) =* ein und schließen sie den *Property Inspector* wieder. Sie sehen, dass sich die Schriftgröße und der Textinhalt des Feldes entsprechend geändert haben.

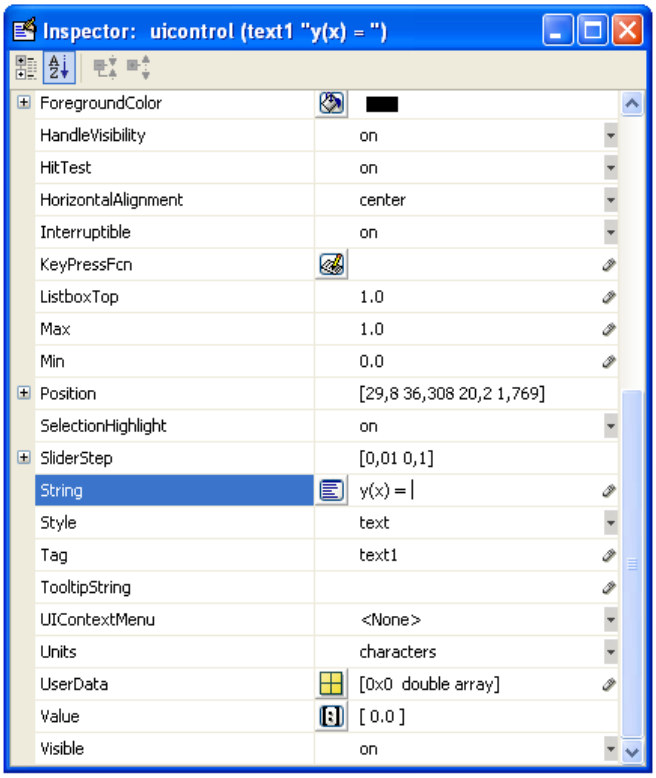


Abbildung 63: Property Inspector

Ändern Sie die Schriftgröße und den Textinhalt aller *Static Text* Felder, aller *Edit Text* Felder, der *Radio Buttons* und der *Button Group* entsprechend. Wählen Sie für das oberste *Edit Text* Feld und die beiden *Static Text* Felder in der *Button Group* zusätzlich für die Eigenschaft *Horizontal Alignment* den Wert *left* aus. Ihr GUI sollte dann ungefähr so aussehen:

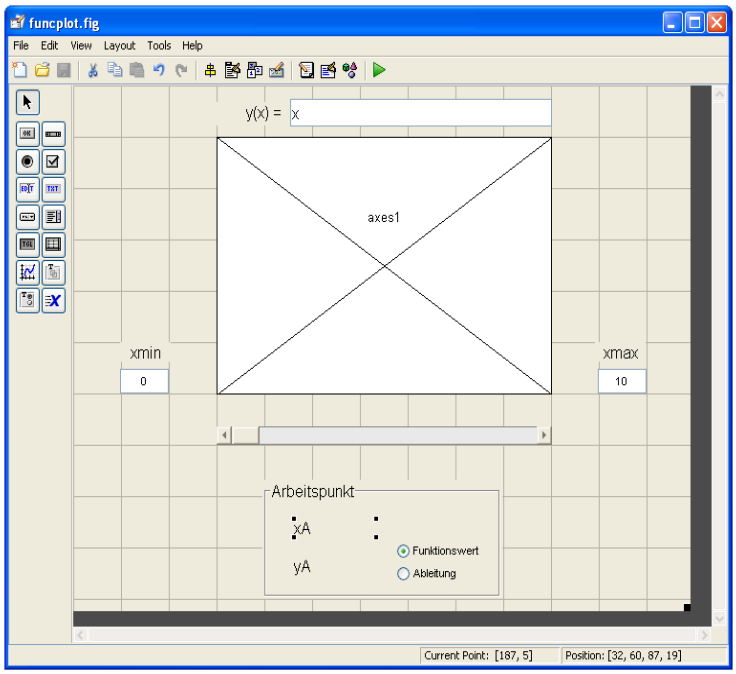


Abbildung 64: GUIDE Layout (3)

Das Design des GUIs entspricht nun im Wesentlichen unserem auf der Skizze festgelegten Layout. Nun könnten wir uns um die Programmierung der eigentlichen Funktionalität unserer Anwendung kümmern.

Zunächst wollen wir aber noch eine wichtige Eigenschaft einiger GUI-Komponenten ändern, nämlich die *Tag* (engl.: Etikett, Schildchen, Aufkleber) Eigenschaft. Doppelklicken Sie auf das oberste *Edit Text* Feld und suchen Sie die Eigenschaft *Tag*. Diese hat die Bezeichnung *editx*, wobei *x* eine Nummer ist. Je nachdem als wievieltes *Edit Text* Feld Sie dieses auf dem Arbeitsblatt platziert haben, kann die Nummer variieren. Jede GUI Komponente kann eindeutig über seine *Tag* Eigenschaft identifiziert werden. Somit muss jede Komponente einen anderen *Tag* Namen aufweisen. Um die anschließende Programmierung des Function Files zu erleichtern, sollte für jede GUI Komponente ein einprägsamer Name verwendet werden. Folgende Konvention zur *Tag* Namenswahl bietet sich beispielsweise an: *komponente_bedeutung*.

Wobei *komponente* für den Typ des grafischen Objekts und *bedeutung* für den Inhalt/Kontext/Aufgabe des Objektes steht. Es ist vorteilhaft, wenn die Tag Namen möglichst kurz gewählt werden. Für das oberste *Edit Text* Feld würd sich also beispielsweise folgender Tag Namen anbieten: *edit_func*

Ändern Sie den *Tag* des Feldes und verfahren Sie ebenso mit den folgenden Komponenten:

GUI Komponente	Tag
Edit Text Feld links neben Plot	*edit_xmin*
Edit Text Feld rechts neben Plot	*edit_xmax*
Slider unter Plot	*slider_x*
Oberes Static Text Feld in der Button Group	*static_x*
Unters Static Text Feld in der Button Group	*static_y*
Oberer Radio Button	*rb_value*
Unterer Radio Button	*rb_diff*

Tabelle 20: Tag N amen für die GUI Elemente

Die Tags der restlichen Felder können ihren *default* Wert behalten. Speichern Sie nun ihr Figure File in GUIDE.

7.2.4 Programmierung der Funktionalität

Öffnen Sie nun das M-File *funcplot.m* im Matlab Editor und scrollen Sie nach unten. Sie sehen, dass für die drei *Edit Text* Felder und den *Slider* automatisch jeweils zwei Funktionen erstellt wurden, die den entsprechenden *Tag* im Funktionsnamen tragen: Für das *Edit Text* Feld mit dem *Tag edit_func* wurden beispielsweise die Funktionen *edit_func_Callback* und *edit_func_CreatFcn* ertsellt:

```
function edit_func_Callback(hObject, eventdata, handles)
% hObject    handle to edit_func (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_func as text
%        str2double(get(hObject,'String')) returns contents of edit_func as a
double

% --- Executes during object creation, after setting all properties.
function edit_func_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_func (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundCo
lor'))
    set(hObject,'BackgroundColor','white');
end
```

Die *Create Function* ist für uns an dieser Stelle nicht von Bedeutung. Diese wird von Matlab aufgerufen, wenn das Programm gestartet und das entsprechende GUI-Objekt erzeugt wird. Von wesentlich größerem Interesse ist für uns die *Callback Function*. Diese wird aufgerufen, wenn der Benutzer an dem jeweiligen Objekt eine Aktion ausführt. Innerhalb der *Callback Functions* muss also die *Programmfunktionalität* implementiert werden. Die *Callback Function* wird mit drei Argumenten aufgerufen:

1. *hObject* ist dabei das zu dem jeweiligen GUI-Objekt gehörende handle, hier also *edit_func*. Das Grafische Objekt kann direkt über dieses handle *hObject* angesprochen werden, z.B. um bestimmte Eigenschaften desselben auszulesen oder zu verändern.

2. *eventdata* ist für uns nicht von Belang.

3. *handles* ist eine Structure, die im Gegensatz zu *hObject* alle Handles des GUI Programms enthält und verwaltet. Dieser Strukturvariablen können wir auch neue Felder hinzufügen und mit ihr funktionsübergreifend in verschiedenen *Callback Functions* oder auch in selbst definierten Funktionen arbeiten.

7.2.5 edit_func_Callback

Zunächst wollen wir die Funktionalität des Edit Text Feldes *edit_func* implementieren: Immer wenn der Benutzer den Inhalt des Feldes ändert, soll der Inhalt ausgelesen und hieraus die entsprechende mathematische Funktion generiert werden. Wir entscheiden uns dafür mit symbolischer Mathematik zu arbeiten, und können die Funktion *edit_func_Callback* somit um folgende Codezeilen erweitern:

```
function edit_func_Callback(hObject, eventdata, handles)
% hObject    handle to edit_func (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_func as text
%        str2double(get(hObject,'String')) returns contents of edit_func as a
double

% --- 1. "edit_func" auslesen ----------------------------------------
    func = get(hObject,'String');

% --- 2. Symbolische Funktion und Ableitung generieren -------------
    y_x  = sym(func);
    yd_x = diff(y_x);

% --- 3. "handles" Structure aktualisieren -------------------------
    handles.data.y_x  = y_x;
    handles.data.yd_x = yd_x;

% --- 4. "handles" Structure abspeichern ---------------------------
    guidata(hObject, handles);
```

Erklärung:

1. Zuerst wird der Inhalt des *Edit Text* Feldes (genauer: dessen Eigenschaft *String*) mit der `get()` Funktion ausgelesen und in der Variablen `func` gespeichert.

2. Anschließend wird auf Grundlage von `func` die symbolische Funktion `y_x` erzeugt. Die Ableitung dieser Funktion wird in der symbolischen Funktion `yd_x` gespeichert.

3. In der `handles` Struktur werden zwei neue Felder angelegt, in welchen die symbolische Funktion und die Ableitung gespeichert werden.

4. Mit dem `guidata` Befehl wird schließlich die `handles` Struktur dauerhaft gespeichert, so dass auch andere Funktionen auf deren aktualisierte Inhalte zugreifen können.

Hinweis: Die *Callback Function* von *Edit Text* Feldern wird immer dann aufgerufen, wenn das Feld angeklickt und der aktuelle oder geänderte Wert im Feld mit *Enter* bestätigt wurde.

7.2.6 edit_xmin_Callback & edit_xmax_Callback

Auf die gleiche Weise lesen wir nun die Inhalte des *edit_xmin* und *edit_xmax* Feldes aus, und speichern diese in der `handles` Struktur. Die Inhalte wollen wir jedoch nicht als Zeichenkette, sondern als numerischen Wert abspeichern, so dass wir nach dem auslesen der *String* Eigenschaft die Funktion `str2num()` anwenden müssen. Gleichzeitig soll die untere bzw. obere Begrenzung des *Sliders* auf den entsprechenden Wert gesetzt werden:

```
function edit_xmin_Callback(hObject, eventdata, handles)
% hObject    handle to edit_xmin (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_xmin as text
%        str2double(get(hObject,'String')) returns contents of edit_xmin as
a double

% --- 1. "edit_xmin" auslesen ----------------------------------------
    xmin = str2num(get(hObject,'String'));

% --- 2. untere Grenze von "slider_x" aktualisieren ----------------
    set(handles.slider_x,'Min',xmin);

% --- 3. "handles" Structure aktualsieren --------------------------
    handles.data.xmin = xmin;

% --- 4. "handles" structure abspeichern ---------------------------
    guidata(hObject, handles);
```

```
function edit_xmax_Callback(hObject, eventdata, handles)
% hObject    handle to edit_xmax (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_xmax as text
%        str2double(get(hObject,'String')) returns contents of edit_xmax as
a double

% --- 1. "edit_xmin" auslesen -------------------------
    xmax = str2num(get(hObject,'String'));

% --- 2. untere Grenze von "slider_x" aktualisieren ---
    set(handles.slider_x,'Max',xmax);

% --- 3. "handles" Structure aktualsieren -------------
    handles.data.xmax = xmax;

% --- 4. "handles" structure abspeichern --------------
    guidata(hObject, handles);
```

Erklärung:

1. Auslesen des Feldinhaltes (Zeichenkette) und Umwandlung in eine double Zahl.
2. Die untere/obere Grenze des *Sliders* (= zulässige Arbeitspunkte x_A) entsprechend setzen. Dies sind die Eigenschaften *Min* bzw. *Max* des *Slider* Objekts.
3. In der `handles` Struktur den Wert von `xmin` bzw. `xmax` aktualisieren.
4. Die `handles` Struktur abspeichern .

7.2.7 slider_x_Callback

Wird der Schieberegler *slider_x* bewegt, wird dessen Callback Function aufgerufen. Die aktuelle Schieberstellung soll dann ausgelesen und als der aktuelle Arbeitspunkt x_A in der `handles` Struktur abgespeichert werden:

```
% --- Executes on slider movement.
function slider_x_Callback(hObject, eventdata, handles)
% hObject    handle to slider_x (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider

% --- 1. Den Arbeitsspunkt (die Stellung) von "slider_x" auslesen ---
    xA = get(hObject,'Value');

% --- 2. "handles" Structure aktualisieren -------------------------
    handles.data.xA = xA;

% --- 3. "handles" Structure abspeichern ---------------------------
    guidata(hObject, handles);
```

Erklärung:

1. Die Eigenschaft *Value* gibt direkt den numerischen Wert des *Sliders* als double zurück.
2. In der `handles` Struktur wird der Wert von x_A aktualisiert.
3. Die `handles` Structure wird abgespeichert.

Als nächstes wollen wir die Funktionalität programmieren, dass bei jeder Bewegung des *Sliders*

a) die mathematische Funktion in den entsprechenden Grenzen gezeichnet,
b) der Arbeitspunkt (x_A/y_A) in den Plot eingezeichnet,
c) der numerische Wert von x_A sowie
d) der numerische Wert von $y_{A(x)}$ oder $y'_{A(x)}$ ausgegeben wird.

Wie wir noch sehen werden, ist es sinnvoll dieses Verhalten nicht direkt in der Funktion *slider_x_Callback* zu implementieren, sondern in einer separaten eigenen Funktion. Diese Funktion nennen wir `ausgabe()` und platzieren diese als Subfunction ganz am Ende des Files *funcplot.m*.

```
function [] = ausgabe(handles)
% --- 1. Handles Structure auslesen ---------------------------------------
    y_x   = handles.data.y_x;
    yd_x  = handles.data.yd_x;
    xA    = handles.data.xA;
    xmin  = handles.data.xmin;
    xmax  = handles.data.xmax;

% --- 2. Funktionswert und Ableitung berechnen ----------------------------
    yA    = subs(y_x,xA);              % Funktionswert am Arbeitspunkt
    ydA   = subs(yd_x,xA);             % Ableitung am Arbeitspunkt

% --- 3. Plot -------------------------------------------------------------
    hold off                           % Achse freigeben für neue Zeichnung
    ezplot(y_x,[xmin,xmax])            % Funktion im Intervall zeichnen
    hold on                            % Achse sperren
    plot(xA,yA,'rx','LineWidth',16)    % Arbeitspunkt einzechnen
    title('')                          % kein Titel im Plot
    grid on                            % Gitternetzlinien ein

% --- 4. Zeichenketten vorbereiten ----------------------------------------
    string_xA = ['xA = ',num2str(xA)];
    string_yA = ['yA = ',num2str(yA)];
    string_ydA= ['y´A = ',num2str(ydA)];

% --- 5. Das Feld "static_x" beschreiben ----------------------------------
    set(handles.static_x,'String',string_xA);

% --- 6. Den Wert des ersten Radiobuttons ermitteln (0 oder 1) ------------
    choice_value = get(handles.rb_value,'Value');

% --- 7. In Abh. von "choice_value" das Feld "static_y" beschr. -----------
    if(choice_value)
        set(handles.static_y,'String',string_yA); % Funktionswert
    else
        set(handles.static_y,'String',string_ydA);% Ableitung
    end
```

Erklärung:

1. Die Funktion `ausgabe()` wird mit dem Argument `handles`, also der Struktur in der unsere Daten hinterlegt sind, aufgerufen. Um den Quellcode übersichtlich zu halten, werden die benötigten Inhalte der `handles` Struktur neuen Variablen mit prägnanten kurzen Namen zugewiesen.
2. Es werden die Funktionswerte `yA` und die Ableitung `ydA` am Arbeitspunkt `xA` berechnet (`xA` entspricht der Sliderstellung).
3. Zuerst wird mit `hold off` die Achse freigegeben, so dass ein eventuell bereits vorhandener Plot in der Achse gelöscht wird. Anschließend wird die symbolische Funktion `y_x` mit dem `ezplot()` Befehl gezeichnet. Die Achse wird mit `hold on` wieder gesperrt, so dass noch der Arbeitspunkt (x_A/y_A) als einzelnes rotes X mit `plot()` gezeichnet werden kann. Zum Schluss wird der automatisch von `ezplot()` erzeugte Titel gelöscht und Gitternetzlinien werden mit `grid on` hinzugeschaltet.
4. Es werden drei Zeichenketten vorbereitet, mit welchen die *Static Text* Felder im *Panel* "Arbeitspunkt" beschrieben werden können.
5. Das *Static Text* Feld *static_x* wird mit dem *x*-Wert des aktuellen Arbeitspunktes beschrieben.
6. Es wird überprüft, ob der erste *Radio Button rb_value* aktiviert ist (Anmerkung: Falls nicht, muss automatisch der zweite *Radio Button rb_diff* aktiviert sein). Falls ja, steht in der Variablen `choice_value` der Wert 1, ansonsten der Wert 0.
7. Das *Static Text* Feld *static_y* wird entsprechend mit einer Zeichenkette beschrieben, je nachdem, ob der Funktionswert oder die Ableitung ausgegeben werden soll.

Die Funktion `ausgabe()` kann jetzt von der *Callback Function* `slider_x_Callback()` mit der `handles` Struktur als Argument aufgerufen werden. Bei jeder Bewegung des Schiebereglers findet dann die Aktualisierung des Plots statt. Wir erweitern daher die *Callback Funktion* um eine Zeile:

```
% --- Executes on slider movement.
function slider_x_Callback(hObject, eventdata, handles)
% hObject    handle to slider_x (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider

% --- 1. Den Arbeitsspunkt (die Stellung) von "slider_x" auslesen ---
    xA = get(hObject,'Value');

% --- 2. "handles" Structure aktualisieren -------------------------
    handles.data.xA = xA;

% --- 3. "handles" Structure abspeichern ---------------------------
    guidata(hObject, handles);

% --- 4. Die Ausgabefunktion "ausgabe" aufrufen ---------------------
    ausgabe(handles)
```

Da wir diese Funktionalität in der separaten Funktion `ausgabe()` und nicht in der *Callback Function* direkt geschrieben haben, kann die Funktion `ausgabe()` nun auch von anderen *Callback Functions* aufgerufen werden kann. Dies ist auch sinnvoll: Wenn ein Benutzer etwa die mathematische Funktion im *Edit Text* Feld *edit_func* oder die untere bzw. obere Grenze ändert, erwartet er, dass die Funktion sofort neu gezeichnet wird und nicht erst nachdem er wieder den *Slider* bewegt hat. Deswegen ergänzen wir die drei *Callback Functions* der *Edit Text* Felder ebenfalls um eine Zeile Code, welche die Funktion `ausgabe()` aufruft:

```
function edit_func_Callback(hObject, eventdata, handles)
% hObject    handle to edit_func (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_func as text
%        str2double(get(hObject,'String')) returns contents of edit_func as
a double

% --- 1. "edit_func" auslesen -------------------------------------
    func = get(hObject,'String');

% --- 2. Symbolische Funktion und Ableitung generieren ------------
    y_x  = sym(func);
    yd_x = diff(y_x);

% --- 3. "handles" Structure aktualisieren ------------------------
    handles.data.y_x  = y_x;
    handles.data.yd_x = yd_x;

% --- 4. "handles" Structure abspeichern --------------------------
    guidata(hObject, handles);

% --- 5. Die Ausgabefunktion "ausgabe" aufrufen --------------------
    ausgabe(handles)
```

```
function edit_xmin_Callback(hObject, eventdata, handles)
% hObject    handle to edit_xmin (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_xmin as text
%        str2double(get(hObject,'String')) returns contents of edit_xmin as
a double

% --- 1. "edit_xmin" auslesen ----------------------------------------
    xmin = str2num(get(hObject,'String'));

% --- 2. untere Grenze von "slider_x" aktualisieren ------------------
    set(handles.slider_x,'Min',xmin);

% --- 3. "handles" Structure aktualsieren ----------------------------
    handles.data.xmin = xmin;

% --- 4. "handles" structure abspeichern -----------------------------
    guidata(hObject, handles);

% --- 5. Die Ausgabefunktion "ausgabe" aufrufen --------------------
    ausgabe(handles)
```

```
function edit_xmax_Callback(hObject, eventdata, handles)
% hObject    handle to edit_xmax (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_xmax as text
%        str2double(get(hObject,'String')) returns contents of edit_xmax as
a double

% --- 1. "edit_xmin" auslesen -------------------------
    xmax = str2num(get(hObject,'String'));

% --- 2. untere Grenze von "slider_x" aktualisieren ---
    set(handles.slider_x,'Max',xmax);

% --- 3. "handles" Structure aktualsieren -------------
    handles.data.xmax = xmax;

% --- 4. "handles" structure abspeichern --------------
    guidata(hObject, handles);

% --- 5. Die Ausgabefunktion "ausgabe" aufrufen --------------------
    ausgabe(handles)
```

7.2.8 Opening Function

Unser Programm ist nun im Wesentlichen fertig. Wir haben das *Design* der Benutzeroberfläche und die *Programmfunktionalität* erfolgreich erstellt. Wenn Sie nun das Programm starten (→ grüner Pfeil in GUIDE) und den *Slider* bewegen, passiert aber erst einmal gar nichts. Im Gegenteil: Sie erhalten im Command Window eine Fehlermeldung:

```
??? Error while evaluating uicontrol Callback
??? Reference to non-existent field 'y_x'.
Error in ==> funcplot>ausgabe at 239
      y_x   = handles.data.y_x;
```

Matlab sagt, dass das Feld `data.y_x` in der `handles` Struktur unbekannt ist. Wir haben aber doch das Feld `data.y_x` in der *Callback Function* des *Edit Text* Feldes *edit_func* definiert:

```
function edit_func_Callback(hObject, eventdata, handles)
% hObject    handle to edit_func (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_func as text
%        str2double(get(hObject,'String')) returns contents of edit_func as
a double

% --- 1. "edit_func" auslesen ----------------------------------------
    func = get(hObject,'String');

% --- 2. Symbolische Funktion und Ableitung generieren ---------------
    y_x  = sym(func);
    yd_x = diff(y_x);

% --- 3. "handles" Structure aktualisieren ---------------------------
    handles.data.y_x  = y_x;
    handles.data.yd_x = yd_x;

% --- 4. "handles" Structure abspeichern -----------------------------
    guidata(hObject, handles);

% --- 5. Die Ausgabefunktion "ausgabe" aufrufen ----------------------
    ausgabe(handles)
```

Das Problem ist folgendes: Wenn Sie direkt nach dem Programmstart den *Slider* bewegen, wurde die *Callback Function* `edit_func_Callback()` aber noch gar nicht ausgeführt, da Sie das *Edit Text* Feld noch nicht geändert haben. Somit ist das Feld `data.y_x` in der `handles` Struktur aber auch noch gar nicht angelegt. Wenn die *Callback Function* des *Slider* nun die Funktion `ausgabe()` aufruft, versucht diese vergeblich auf dieses Feld `data.y_x` zuzugreifen.

Sie müssten also nach Programmstart erst in alle drei *Edit Text* Felder klicken und den Inhalt mit Enter bestätigen, so dass deren jeweilige *Callback Function* ausgeführt und somit alle notwendigen Felder in der `handles` Struktur angelegt werden. Erst danach können Sie den *Slider* und den damit verbundene Aufruf der `ausgabe()` Funktion erfolgreich benutzen. Dieses Verhalten ist natürlich sehr unschön. Deshalb generiert Matlab über GUIDE automatisch die sogenannte *Opening Function*. Diese wird aufgerufen, wenn das Programm gestartet wird. Hier können alle notwendigen *Initialisierungen* beim Programmstart erfolgen.

Wir begnügen uns hier mit einem recht einfachen Inhalt der *Opening Function*: Wir initialisieren einfach alle benötigten Felder in der `handles` Struktur mit den *default* Werten der jeweiligen grafischen Objekte. Dies sind die Werte, die in GUIDE festgelegt wurden. Dabei dürfen wir nicht vergessen auch die Grenzen des *Sliders* gleich entsprechend zu setzen:

```
% --- Executes just before funcplot is made visible.
function funcplot_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to funcplot (see VARARGIN)

% --- 1. Default Werte in der "handles" Structure setzen ------------
    handles.data.y_x  = sym('x');
    handles.data.yd_x = diff(handles.data.y_x);
    handles.data.xA   = 0;
    handles.data.xmin = 0;
    handles.data.xmax = 10;
% --- 2. Die untere und obere Grenze des Sliders entspr. setzen -----
    set(handles.slider_x,'Min',handles.data.xmin);
    set(handles.slider_x,'Max',handles.data.xmax);
% --- 3. Die Ausgabefunktion "ausgabe" aufrufen ---------------------
    ausgabe(handles)

% Choose default command line output for funcplot
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes funcplot wait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

Erklärung:

1. Alle Felder der `handles` Struktur, auf die die Funktion `ausgabe()` zugreift, werden mit sinnvollen Werten belegt, bzw. mit den Werten, welche die entsprechenden *Edit Text* Felder und der *Slider* beim Programmstart haben.
2. Die Grenzen des *Sliders* werden entsprechend dem Inhalt der *Edit Text* Felder *edit_xmin* und *edit_xmax* gesetzt. Alternativ kann dies auch bereits in GUIDE über dem *Property Inspector* gemacht werden.
3. Es wird die Ausgabefunktion `ausgabe()` aufgerufen, so dass beim Programmstart bereits die voreingestellte Funktion gezeichnet wird.

7.2.9 Programmaufruf

Sie können das Programm bekanntlich von GUIDE aus mit dem grünen Pfeil starten. Alternativ können Sie GUIDE jetzt auch schließen und das Programm vom Command Window auch mit dem Programmnamen `funcplot` aufrufen. Das Programm wird gestartet und sollte nun voll funktionsfähig sein.

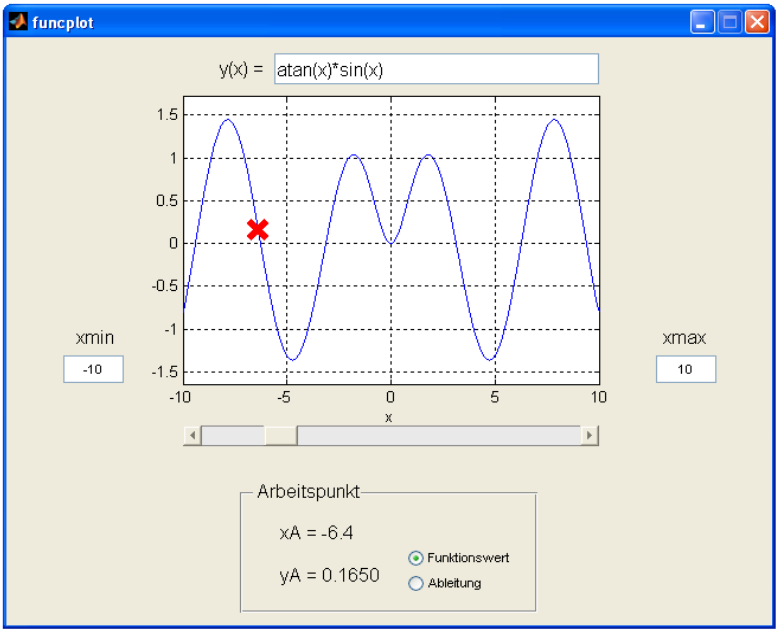


Abbildung 65: Programm "funcplot" mit grafischer Benutzerschnittstelle