Numerik mit MATLAB

Hans W Borchers
Duale Hochschule, DHBW Mannheim

Skript zum Kurs, WS 2016

Inhaltsverzeichnis

"MATLAB ist eine kommerzielle Software des US-amerikanischen Unternehmens **MathWorks** zur Lösung mathematischer Probleme und zur grafischen Darstellung der Ergebnisse. MATLAB ist vor allem für numerische Berechnungen mithilfe von Matrizen ausgelegt, woher sich auch der Name ableitet: MATrix LABoratory.

Die Software wird in der Industrie und an Hochschulen vor allem für numerische Simulation sowie Datenerfassung, Datenanalyse und -auswertung eingesetzt.

Die akademische Bindung ist in der Entwicklung und im Vertrieb von relativ preisgünstigen Studentenversionen bis heute erhalten geblieben und war möglicherweise auch die Grundlage für den Erfolg der Software."

— *Wikipedia.org/de, 2016*

THIS PAGE INTENTIONALLY LEFT BLANK.

1 Numerische Lineare Algebra

Vektoren und Matritzen sind die grundlegenden Bausteine der Linearen Algebra ebenso wie der Numerik. Beim Lösen numerischer Probleme in der Analysis spielen sie eine wichtige Rolle. Das Aufstellen von langen Vektoren und großen Matrizen (auch spezieller Bauart) ist in Matlab leicht möglich.

1.1 Vektoren

1.1.1 Erzeugung und Zugriff

Vektoren sind in MATLAB – über die geometrische Vorstellung hinausgehend – einfach Folgen reeller (oder komplexer) Zahlen. Ein Vektor wie $x = (1, 2, 3, 4, 5)$ wird dabei mit eckigen Klammern `[` und `]` erzeugt, die Elemente (oder Komponenten) des Vektors werden durch Komma `,` oder auch nur durch ein Leerzeichen getrennt:

```
>> x = [1, 2, 3, 4, 5]
x =
     1     2     3     4     5
```

Genauer ist dies ein *Zeilenvektor* (eine 1x5-Matrix). Um einen *Spaltenvektor* zu erzeugen, werden die Elemente durch ein Semikolon `;` getrennt.

```
>> y = [1; 2; 3; 4; 5]
y =
     1
     2
     3
     4
     5
```

Vektoren einer bestimmten Länge bilden einen *Vektorraum*, d.h. sie können miteinander addiert oder mit einem *Skalar*, einer Zahl, multipliziert werden.

```
>> x + x
ans =
    2     4     6     8    10

>> 10 * x
ans =
    10    20    30    40    50
```

`x` und `y` dagegen gehören trotz gleicher Länge (von 5 Elementen) verschiedenen Vektorräumen an und können nicht addiert werden. Bei dem Versuch gibt MATLAB eine Fehlermeldung aus.

```
>> x + y
Error using  +
Matrix dimensions must agree.
```

Die Meldung "Matrix dimensions must agree." wird immer dann erscheinen, wenn eine Operation auf Vektoren oder Matrizen wegen unterschiedlicher Grössen oder Bauarten nicht möglich ist.

Durch *Transponieren* kann man Zeilen- in Spaltenvektoren umwandeln, und umgekehrt. Transponieren geschieht in MATLAB durch Anhängen des Hochkommas `'` an den Vektor.

```
>>> y'
ans =
     1     2     3     4     5
```

Für grössere Vektoren ist diese Art der Eingabe unpraktisch oder nicht möglich. Um zum Beispiel die Zahlen von 1 bis 20 in einen Vektor zu schreiben, benutzt man den Doppelpunkt (engl. *colon*) zwischen Anfangs- und Endwert.

```
>> x = 1:20
  Columns 1 through 14
 1     2     3     4     5     6     7     8     9    10    11    12    13    14
  Columns 15 through 20
    15    16    17    18    19    20
```

Die Schrittweite zwischen den Elementen ist automatisch 1, man kann aber eine andere Schrittweite zwischen Anfangs- und Endwert angeben, durch einen weiteren Doppelpunkt getrennt. So wird `x = 1:2:20` alle ungeraden Zahlen ausgeben, nur 21 nicht mehr, da dieser Wert schon grösser als der Endwert ist.

Es kann auch eine Liste mit abnehmendem Zahlenverlauf erstellt werden. Dazu wird vor die Schrittweite ein Minus geschrieben. `10:-2:1` erzeugt den Vektor $(10, 8, 6, 4, 2)$.

Eine Folge `0:0.1:pi` bricht schon bei 3.1 ab. Für viele Anwendungen, zum Beispiel beim Plotten, möchte man aber, dass der Endwert wirklich erreicht wird. Das ermöglicht die `linspace` Funktion.

```
>> x = linspace(0, pi, 10)
x =
  Columns 1 through 8
         0    0.3491    0.6981    1.0472    1.3963    1.7453    2.0944    2.4435
  Columns 9 through 10
    2.7925    3.1416
```

Der Aufruf `linspace(a, b, n)` erzeugt einen Vektor von `a` bis `b` in gleichmässigem Abstand mit genau der Länge `n`.

Aufgabe: Wie kann man einen Nullvektor $(0, ..., 0)$ oder einen Vektor $(1, ..., 1)$ der Länge `n` erzeugen?

Die eckigen Klammern `[` und `]` dienen auch dazu, Vektoren aneinander zu hängen, das heisst aus Vektoren $(x_1, ..., x_n)$ und $(y_1, ..., y_m)$ einen neuen Vektor $(x_1, ..., x_n, y_1, ..., y_m)$ zu erzeugen.

```
>> x = [1, 2, 3, 4];
>> [x, x.^2, x.^3]
ans =
     1     2     3     4     1     4     9    16     1     8    27    64
```

Entsprechend kann man zwei Spaltenvektoren `x` und `y` durch Benutzung des Semikolons in eckigen Klammern zusammenfügen: `[x; y]`.

Wie kann man den Wert des `i`-ten Elementes eines Vektors `x` benutzen oder sogar verändern. MATLAB benutzt dazu die runden Klammern `(` und `)`, ähnlich wie beim Aufruf einer Funktion. Mit `x(5)` wird das 5-te Element des Vektors gelesen, mit `x(5) = 10` wird diesem Element der Wert 10 zugewiesen.

WICHTIG: Vektoren und Matrizen sind in MATLAB *1-basiert*, das heisst das erste Element hat den Index 1, das zweite 2, usw. In den meisten Programmiersprachen wird dagegen das erste

Element eines Vektors mit dem Index 0 erreicht (C, VBA, Python, ...), das n-te mit dem Index $n-1$.

Im folgenden Beispiel nutzen wir die Funktion `primes`, welche bei einem Aufruf `primes(n)` alle Primzahlen kleiner oder gleich `n` als Vektor zurückgibt. (Informieren Sie sich auf Wikipedia über Primzahlen.)

```
>> P = Primes(100);
>> P(1)                    % 2 ist die erste Primzahl
```

Um z.B. die 5-te bis 10-te Primzahl zu sehen, werden die Indizes 5:10 mit *colon*-Notation dem Vektor übergeben.

```
>> P(5:10)
ans =
    11    13    17    19    23    29
```

Wenn man nicht weiss, wie lang ein Vektor ist, kann man sehr einfach mit dem Schlüsselwort `end` auf das letzte Element zugreifen, `x(end)`, statt das umständlichere `x(length(x))` benutzen zu müssen.

Aufgabe: Wie viele Primzahlen gibt es bis 100, wie viele bis 10000? Welches ist die grösste Primzahl unterhalb einer Million. Wie gross ist die 1000-te Primzahl?

1.1.2 Logische Vergleiche auf Vektoren

Auch Vergleichsoperatoren wie `>`, `>=`,`<`,`<=`,`==` oder `~=` sind vektorisiert. Ist `x` ein Vektor, wird etwa mit `x >= 5` ein Vektor gleicher Länge wie `x` zurückgegeben, der 1 enthält an den Stellen, an denen x grösser als 5 ist, und 0 an den anderen Stellen.

```
>> x = [0, 1, 2, -3, 4, -5, -6, 7, -8, 9];
>> x >= 0
ans =
     1     1     1     0     1     0     0     1     0     1
```

Die Funktion `find` findet in einem Vektor die Indizes aller Elemente die nicht 0 sind. Da 0 auch der logische Wert für "falsch" ist, lässt sich das gut verbinden mit solchen Vergleichen auf Vektoren.

```
>> inds = find(x >= 0)
inds =
     1     2     3     5     8    10
>> sum(x >= 0)
ans =
     6
```

Da `sum` alle Einsen aufsummiert, wird damit die Anzahl der Elemente bestimmt, welche die Bedingung erfüllen, in diesem Fall `x >= 0`.

1.1.3 Funktionen auf Vektoren

Neben den Vektoroperationen bietet MATLAB verschiedene Funktionen an, die auf dem ganzen Vektor arbeiten. Neben `length(x)`, das die Länge, also Anzahl der Elemente, des Vektors zurückgibt, gibt es `sum`, `prod`, `min`, `max`, und `mean`, in offensichtlichen Bedeutungen. So berechnet

`sum(x)` bzw. `prod(x)` die Summe bzw. das Produkt aller Elemente des Vektors `x`, `min(x)` bzw. `max(x)` das Minimum bzw. Maximum aller Elemente von `x`, und `mean(x)` den Mittelwert.

Aufgabe: Berechne die Summe aller Zahlen von 1 bis 100. Stimmt das Ergebnis mit der Ihnen bekannten Formel überein? Wie gross ist das Produkt aller Zahlen von 1 bis 100?

`sort(x)` gibt den aufsteigend sortierten Vektor zurück (ohne `x` selbst zu verändern). Um den Vektor absteigend zu sortieren, benutzt man das Schlüsselwort 'descend'.

```
>> x = [4, 1, 3, 9, 7, 2, 8, 5, 6];
>> sort(x)
ans =
     1     2     3     4     5     6     7     8     9
>> x
x =
     4     1     3     9     7     2     8     5     6
>> sort(x, 'descend')
ans =
     9     8     7     6     5     4     3     2     1
```

Aufgabe: (Unter Benutzung der MATLAB Dokumentation) Was berechnet die Funktion `median`, was tut die Funktion `unique`? Warum gibt `issorted(x)` den Wert 0 zurück?

In MATLAB sind alle elementaren mathemathischen Funktionen (trigonometrische, Exponential-, Rundungsfunktionen, etc.) *vektorisiert*, das heisst angewandt auf einen Vektor werden sie auf jedes Element des Vektors angewendet und ein Vektor der berechneten Funktionswerte zurückgegeben.

```
>> x = [0, pi/2, pi, 3*pi/2, 2*pi];
>> sin(x)
ans =
     0    1.0000    0.0000   -1.0000   -0.0000
```

Das ist für viele numerische Berechnungen sehr praktisch, muss aber immer beachtet werden, insbesondere beim Schreiben eigener Funktionen.

1.1.4 Elementweise Operationen

In Matlab bezeichnet der Stern `*` *immer* die *Matrix*multiplikation. Solange in einem Ausdruck `x*y` eine der beiden Variablen (oder beide) ein Skalar ist, also eine einfache Zahl, spielt das keine Rolle und `x*y` verhält sich wie eine Matrix multipliziert mit einer Zahl. Wenn aber beides echte Matrizen sind, Zeilen- oder Spaltenanzahl grösser als 1, dann müssen auch die Voraussetzungen für eine Matrizenmultiplikation erfüllt sein. "Inner matrix dimensions must agree" ist sonst eine übliche Fehlermeldung.

```
>> x = [1, 2, 3, 4, 5];
>> x * x
Error using  *
Inner matrix dimensions must agree.
```

Zeilenvektor mal Spaltenvektor erfüllt diese Bedingung und wird daher von MATLAB korrekt ausgeführt.

```
>> x = [1, 2, 3, 4, 5];  % Zeilenvektor
>> y = [1; 2; 3; 4; 5];  % Spaltenvektor
>> x * y
```

```
ans =
    55
```

Dasselbe gilt für Division mit `/` und Potenzierung mit `^`, beide werden als Matrixoperationen aufgefasst. (Insbesondere `x^2` sollte ja mit `x * x` übereinstimmen.)

Dennoch ist die *punktweise*, *elementweise* Multiplikation zweier Vektoren (oder Matrizen) in der numerischen Mathematik oder Datenanalyse ein ständig benötigte Operation. MATLAB hat für diese Operationen eine eigene Schreibweise eingeführt, die elementweise Multiplikation `.*`, Division `./` und Potenzierung mit einer Zahl `.^`. Der Punkt soll dabei andeuten, dass diese Operation eben 'punktweise' ausgeführt werden soll.

```
>> x = [1, 2, 3, 4, 5];
>> x .* x
ans =
     1     4     9    16    25
>> x ./ x
ans =
     1     1     1     1     1
>> x.^0.5
ans =
    1.0000    1.4142    1.7321    2.0000    2.2361
```

Anmerkung: Leider ergibt `x .*x` eine (unberechtigte) Fehlermeldung. Es ist in MATLAB durchaus vorteilhaft, Operatoren immer mit Leerzeichen zu umgeben.

Aufgabe: Berechne $\sum_{n=1}^{N} 1/n^2$ für $N = 1000$ und für $N = 10^6$. Wie gross ist der Fehler in beiden Fällen, wenn $\pi^2/6$ der tatsächliche Grenzwert dieser Reihe ist?

1.1.5 Geometrische Anwendungen

Vektoren mit zwei oder drei Elementen können auch als Vektoren im 2- oder 3-dimensionalen Raum aufgefasst werden. Addition von Vektoren bedeutet dann das Aneinanderfügen zweier solcher geometrischer Vektoren.

Die Norm eines Vektors ist mathematisch definiert als $|(x_1, ..., x_n)| = \sqrt{x_1^2 + ... + x_n^2}$ und wird in MATLAB realisiert mit der Funktion `norm` und berechnet die geometrische Länge des Vektors. Der Vektor `x = [1, 1, 1]` beschreibt die Raumdiagonale im Würfel der Seitenlänge 1, und hat selbst die Länge $\sqrt{3}$:

```
>> x = [1, 2, 3];
>> norm(x)
ans =
    1.7321
```

Die Funktion `dot` berechnet das Skalarprodukt zweier Vektoren (beliebiger Länge), und `cross` das Kreuzprodukt zweier Vektoren im 3-dimensionalen Raum.

```
>> x = [1, 0, 0]; y = [0, 1, 0];
>> dot(x, y)
ans =
     0
>> cross(x, y)
ans =
     0     0     1
```

Aufgabe: Entspricht das Ihren Erwartungen bzw. der Rechte-Hand-Regel für das Kreuzprodukt?

Der Winkel ϕ zwischen zwei Vektoren v und w wird bestimmt durch den Ausdruck $\cos\phi = \frac{v \cdot w}{|v||w|}$. Eine Funktion, welche diesen Winkel berechnet, kann man folgendermassen definieren:

```
>> winkel = @(v, w) acos(dot(v, w) / norm(v) / norm(w));
>> phi = winkel([1, 1, 1], [1, 1, 0]);
>> rad2deg(phi)
ans =
   35.2644
```

Der Winkel zwischen der Raumdiagonale und der (x, y)-Ebene beträgt etwa 35 Grad.

Aufgabe: Die Projektion eines Vektors v auf einen anderen Vektor w wird berechnet zu $vw = \frac{v \cdot w}{|w|^2} w$. Schreibe eine Funktion in MATLAB, die diese Projektion berechnet. Bestimme die Projektion der Flächendiagonale `[1, 1, 0]` auf die Raumdiagonale und umgekehrt die Projektion der Raumdiagonale auf die Flächendiagonale.

1.2 Matrizen

Ein Matrix ist, vereinfacht gesagt, ein rechteckiges Schema von Zahlen. Eine Matrix mit m Zeilen und n Spalten wird auch als $(m \times n)$-Matrix bezeichnet. Ihre grosse Bedeutung haben Matrizen als lineare Abbildungen auf Vektorräumen und als Speicher von Daten in der Datenanalyse.

1.2.1 Erzeugung von Matrizen

Kleinere Matrizen können ähnlich wie Vektoren unter Benutzung der Klammern `[` und `]` eingegeben werden. Beispielsweise eine Matrix wie

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

wird in MATLAB eingetippt mit dem Semikolon `;` als Trennzeichen zwischen den Zeilen und dem Komma `,` (oder nur Leerzeichen) innerhalb der Zeile.

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

Sind bei der Eingabe nicht alle Zeilen und Spalten gleich lang, wird eine Fehlermeldung erscheinen: "Dimensions of matrices being concatenated are not consistent."

Um die Matrix `A` auf einen Spalten(!)vektor (warum?) anzuwenden, wird die Matrizenmultiplikation `*` genutzt. Das Ergebnis ist dann wieder ein Spaltenvektor.

```
>> x = [1; 1; 1]
x =
     1
     1
     1
>> y = A * x
```

```
x =
     6
    15
    24
```

In MATLAB gibt es Standardmatrizen, die öfters auftauchen und somit die Eingabe insbesondere grosser Matrizen vereinfachen. Dies sind die Nullmatrix (aus lauter Nullen bestehend), eine Matrix aus lauter Einsen, oder die Einheitsmatrix mit Einsen in der Diagonale und sonst nur Nullen.

Die Aufrufe dafür sind `zeros(m,n)`, `ones(m,n)`, bzw. `eye(m, n)`, wobei jeweils `m` die Anzahl der Zeilen und `n` die Anzahl der Spalten ist; `eye(n)` ist die quadratische Einheitsmatrix mit `n` Zeilen und Spalten.

`A=zeros(3,5)` bedeutet, dass A eine (3x5)-Matrix mit Nullen ist. Man kann auch einen Nullvektor auf diese Art erstellen. Tippt man zum Beispiel `ones(3,1)` ein, erhält man einen Spaltenvektor aus Einsen, mit `ones(1,3)` einen Zeilenvektor.

`eye(4)` ergibt eine Einheitsmatrix in der angegebenen Größe. Soll die Matrix auf der Diagonalen andere Werte als die Einheitsmatrix haben, kann man `diag([4,6,2,8])` eintippen. Nun sind die Werte auf der Diagonalen 4, 6, 2 und 8. Umgekehrt liefert der `diag` Befehl auch die Diagonalelemente einer bestehenden Matrix: `diag(A)`.

Die Matrix `A` kann auch schneller generiert werden. Dazu wird der Vektor `1:9` mit dem Befehl `reshape` neu formatiert, indem eine neue Zeilen- und Spaltenanzahl vorgegeben wird.

```
>> A = reshape(1:9, 3, 3)
A =
     1     4     7
     2     5     8
     3     6     9
```

Es ist typisch für MATLAB, dass Matrizen spaltenweise aufgefüllt und benutzt werden. Um die korrekte Matrix `A` zu erzeugen, könnte man `A` transponieren, `A = A'`, mit dem Hochkomma als Operatorzeichen.

```
>> A = reshape(1:9, 3, 3)'
A =
     1     2     3
     4     5     6
     7     8     9
```

Eine weitere Möglichkeit, besondere Matrizen zu erzeugen, wird durch den Befehl `repmat` bereitgestellt ("repeat matrix").Dieser Befehl wiederholt eine Matrix so oft in der Zeile und der Spalte, wie angegeben ist.

```
>> B = [1 2; 3 4];
>> repmat(B, 2, 3)
ans =
     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

Matrizen können mithilfe der Klammern `[` und `]` erweitert oder aneinander gefügt werden. Durch `[A, B]` werden `A` und `B` nebeneinander gesetzt (Anzahl ihrer Zeilen muss gleich sein), durch `[A; B]` werden sie untereinander gesetzt (Anzahl ihrer Spalten muss gleich sein).

1.2.2 Zufallszahlen

Manchmal braucht man eine, mehrere oder viele zufällige Zahlen, als Beispieldaten oder zur Simulation von Prozessen. Mit dem Computer können keine wirklichen Zufallszahlen generiert werden. Stattdessen werden sog. *Pseudo-Zufallszahlen* durch teilweise komplizierte math. Prozeduren errechnet.

In MATLAB werden Zufallszahlen mit den Befehlen `rand`, `randn`, oder `randi` erzeugt (engl. *random*). Mit `rand(m, n)` entsteht eine (mxn)-Matrix von Zufallszahlen, also gleichmässig verteilte Gleitkommazahlen zwischen 0 und 1.

```
>> rand(3, 2)
ans =
    0.8147    0.9134
    0.9058    0.6324
    0.1270    0.0975
```

Aufgabe: Generiere 100 gleichverteilte Zufallszahlen im Intervall von -1 bis 1. Ist der Mittelwert nahe bei 0?

Nach jedem MATLAB Neustart werden die gleichen Zufallszahlen erzeugt. Um das zu verhindern, zuvor den Befehl `rng('shuffle')` an den Zufallszahlen-Generator schicken.

Um die Zufallszahlen auch ohne Neustart wiederholbar zu machen, kann man dem Generator eine Art Kennnummer mitgeben, sodass anschliessend immer die gleichen Zufallszahlen benutzt werden. Das ist oft wichtig für die Vergleichbarkeit der Resultate.

Statt Gleitkommazahlen können auch zufällige ganze Zahlen in eine Matrix geschrieben werden. Dies geht über den Befehl `randi`. Zuerst steht, in welchem Bereich $1, ..., n$ die ganzen Zahlen liegen sollen. Danach kommt die Größe der Matrix. `A=randi(10, 3, 2)` gibt somit eine 3x2-Matrix mit Werten von 1 bis 10 aus.

```
>> rng(1001)            % Benutze eine persönliche Kennzahl
>> Z = randi(10, 3, 2)
Z =
     4     5
     3     1
     2     2
```

Wir können auch eine 3x2-Matrix mit zufälligen ganzen Zahlen zwischen -5 und 5 anfordern: `C=randi([-5, 5], 4, 5)`.

Aufgabe: Erzeuge einen Vektor mit 1000 Würfen eines korrekten Würfels (mit sechs Seiten). Wie gross ist der Mittelwert aller Würfe? Bestimme, wie oft eine 1 bzw. eine 6 gewürfelt wurde. Nach wievielen Würfen wurde 10 Mal die 3 gewürfelt?

Interessant ist noch der Befehl `randperm`, der eine Permutation von Zahlen $1, ..., n$, bzw. eine zufällige Auswahl dieser Zahlen erzeugt.

```
>> randperm(10)
ans =
     6     9     2     1    10     7     4     3     5     8
>> randperm(10, 5)
ans =
     6     1     8     7     9
```

Aufgabe: Wähle zufällig 10 Primzahlen aus der Menge der Primzahlen unter 1000 aus. Wie könnte man realisieren, dass auch Mehrfachziehungen einer Zahl möglich sind?

Der Befehl `randn` erzeugt *normalverteilte* Zufallszahlen (s. Gaußsche Normalverteilung) mit Mittelwert 0 und Standardabweichung 1. Nützlich zum Beispiel um die Schwankung von Messwerten um einen Mittelwert zu simulieren.

1.2.3 Index-Yoga

Ähnlich wie für Vektoren geschieht das Auslesen von Elementen aus einer Matrix durch Klammern, allerdings muss hier sowohl die gewünschte Zeile wie Spalte angegeben werden.

```
>> M = magic(4)     % ein magisches Quadrat
M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> M(2, 3)
ans =
    10
```

Entsprechend können auch ganze Untermatrizen oder Zeilen oder Spalten ausgewählt werden. Um etwa die Elemente der 2-ten und 3-ten Zeile und der Spalten 2 bis 4 auszuwählen, schreibt man `M(2:3, 2:4)`.

```
>> M(2:3, 2:4)
ans =
    11    10     8
     7     6    12
```

Man kann auch ganze Zeilen oder Spalten aus Matrizen auswählen. Dies kann man durch Eintippen eines Doppelpunktes `:` erreichen, ohne Anfangs- und Endwert. So bedeutet `m = M(:, 1)`, dass die gesamte erste Spalte ausgewählt wird.

```
>> M(3, :)                 % Zeile 3, auch M(3, 1:end)
ans =
     9     7     6    12
>> M(:, 2)                 % Spalte 2, auch M(1:end, 2)
ans =
     2
    11
     7
    14
```

Wie man sieht, ist das Ergebnis ein Zeilen- oder Spaltenvektor, je nachdem ob man eine Zeile oder Spalte ausgewählt hat.

Neben einzelnen Elementen einer Matrix, z.B. durch `M(2, 3) = 20`, können auch ganze Zeilen, spalten oder Untermatrizen geändert werden. Voraussetzung ist aber, dass die Dimensionen (Zeilen-, Spaltenanzahl) des zu ersetzenden Bereiches und des neuen Bereiches übereinstimmen.

```
>> M(2, 4) = 2 * M(2, 4);   % verdoppele M(2, 4)
>> M(2:3, 2:3) = 0
M =
```

```
    16     2     3    13
     5     0     0    16
     9     0     0    12
     4    14    15     1
```

Ausnahme ist offensichtlich, dass die Zuweisung einer Zahl an einen ganzen Bereich diesen mit nur einen Zahl überschreibt.

Aufgabe: Addiere die erste Zeile zur vierten und dividiere anschliessend die vierte Zeile durch ihr erstes Element.

Durch Angabe nur eines Indexes wird eine Matrix wie ein Vektor addressiert, wieder spaltenweise, daher ergibt `M(5)` den Wert 2, das erste Element der zweiten Spalte. Wird nur `:` eingegeben, erhält man die ganze Matrix als Spaltenvektor.

```
>> M(12:15)
ans =
    15
    13
    16
    12
```

Man vergleiche die Ergebnisse von `length(M)` und `length(M(:)`.

Um eine Zeile (oder Spalte) einer Matrix zu löschen, weist man ihr einen leeren Vektor zu, in MATLAB symbolisiert durch leere Klammern: `[]`.

```
>> M = [1, 2, 3; 4, 5, 6; 7, 8, 9];
>> M(2, :) = []
M =
     1     2     3
     7     8     9
```

Das Einfügen einer Zeile oder Spalte zwischen andere Zeilen oder Spalten ist nicht ganz so einfach und muss in zwei Schritten erfolgen (wie?).

1.2.4 Rechnen mit Matrizen

Matrizen gleicher Dimension können elementweise addiert und subtrahiert oder mit einem Skalar multipliziert werden. Alle (m, n)-Matrizen zusammen bilden also einen Vektorraum der Dimension mn (warum) über den rellen (oder komplexen) Zahlen.

Matrizen der Dimensionen $m \times n$ und $n \times k$ können im Sinne der Matrizenmultiplikation miteinander multipliziert werden und bilden dann eine neue Matrix der Dimension $m \times k$. In MATLAB bezeichnet der Stern `*` immer diese Matrixmultiplikation.

Entsprechend bedeutet `A / B` die Multiplikation der Matrix `A` mit dem Inversen der Matrix `B`, und `A^n` die n-fache Multiplikation der Matrix `A` mit sich selbst. Die Inverse von `A` wird mit `A^-1` oder `inv(A)` berechnet, wobei möglicherweise `inv(A)` vorzuziehen ist.

Und wie für Vektoren gibt es für Matrizen die elementweise Multiplikation `.*`, Division `./` und Potenzierung `.^`. Für `A .* B` und `A ./ B` müssen `A` und `B` gleiche Dimension haben. Im Falle quadratischer Matrizen kann man das direkt miteinander vergleichen.

```
>> M = magic(4);            % siehe oben
>> M * M
ans =
```

```
   345   257   281   273
   257   313   305   281
   281   305   313   257
   273   281   257   345
>> M .* M
ans =
   256     4     9   169
    25   121   100    64
    81    49    36   144
    16   196   225     1
```

1.2.5 Funktionen auf Matrizen

`size(A)` liefert die Anzahl von Zeilen und Spalten der Matrix `A` als einen Vektor mit zwei Elementen; `size(A, 1)` nur die Zeilen- und `size(A, 2)` nur die Spaltenanzahl. Für eine Zahl `a` gibt `size(a)` den Vektor `[1, 1]` zurück, MATLAB betrachtet einzelne Zahlen als (1×1)-Matrizen.

```
>> A = ones(3, 2);
>> size(A)
ans =
     3     2
>> size(1.0)
ans =
     1     1
```

`inv(A)` zum Invertieren einer Matrix `A` haben wir schon kennengelernt. Die Determinante ist eine andere nützliche Funktion und wird aufgerufen mit `det(A)`. `rank(A)` bestimmt den Rang der Matrix, die maximale Zahl linear unabhängiger Zeilen oder Spalten.

```
>> M = magic(4);
>> d = det(M), r = rank(M)
d =
   5.1337e-13
r =
     3
>> inv(M)
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate.
RCOND =  4.625929e-18.
ans =
   1.0e+15 *
   -0.2649   -0.7948    0.7948    0.2649
   -0.7948   -2.3843    2.3843    0.7948
    0.7948    2.3843   -2.3843   -0.7948
    0.2649    0.7948   -0.7948   -0.2649
```

Die von Vektoren bekannten Funktionen `sum`, `prod`, `min`, `max` und `mean` funktionieren auch für Matrizen, allerdings in etwas überraschender Weise.

```
>> M = magic(4);
>> sum(M)
ans =
    34    34    34    34
```

Diese Funktionen berechnen ihre Werte spaltenweise und nicht über die ganze Matrix. `M` ist ein magisches Quadrat, die Summe aller Spalten ist gleich (und muss deshalb 34 sein). Die Summe der Zeilen könnte man erhalten, indem `sum` auf die transponierte Matrix `M'`angewandt wird, aber `sum(M, 2)` funktioniert auch.

```
>> sum(M, 2)
ans =
    34
    34
    34
    34
```

Aufgabe: Ist auch die Summe der Elemente in der Diagonalen gleich 34? Und wie sieht es mit der Nebendiagonalen aus?

Aufgabe: Wie kann man die Summe, das Produkt, etc. aller Elemente der Matrix berechnen?

1.3 Lineare Gleichungssysteme (LGS)

Ein lineares Gleichungssystem in Matrixschreibweise lautet $A\,x = b$. A ist eine $(m \times n)$-Matrix und b ein Vektor der Länge m. Gesucht wird der Vektor x der Länge n, der diese Gleichung erfüllt. Im Allgemeinen wird angenommen, dass $m = n$ gilt und A eine Matrix von maximalen Rang (d.h., invertierbar) ist. Nur dann dann gibt es genau eine Lösung des Gleichungssystems.

Eine bekannte und effiziente Methode, lineare Gleichungssysteme zu lösen, ist das *Gaußsche* (Eliminations-) *Verfahren*, siehe [Papula, "Mathematik für Ingenieure und Naturwissenschaftler", Band 1, Abschnitt I.5]. Dabei wird die Matrix A durch Zeilenoperationen schrittweise in die Einheitsmatrix überführt. Anwendung derselben Schritte auf den Vektor b ergibt dann die Lösung.

Beispiel: Wir wollen das folgende lineare Gleichungssystem lösen:

$$\begin{pmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}$$

Wir erstellen die Matrizen `A` und `b` und daraus die 'erweiterte' Matrix `B`:

$$B = \begin{pmatrix} 1 & 1/2 & 1/3 & 1 \\ 1/2 & 1/3 & 1/4 & -1 \\ 1/3 & 1/4 & 1/5 & 1 \end{pmatrix}$$

```
>> A = [  1, 1/2, 1/3; 1/2, 1/3, 1/4; 1/3, 1/4, 1/5];
>> b = [1; -1; 1];
>> B = [A, b]
B =
    1.0000    0.5000    0.3333    1.0000
    0.5000    0.3333    0.2500   -1.0000
    0.3333    0.2500    0.2000    1.0000
```

Im Gaußschen Verfahren sind nur reine Zeilenoperationen erlaubt. Um zum Beispiel das Element `B(3, 1)` zu Null zu machen, benutzen wir die erste Zeile:

```
>> B(3, :) = B(3, :) - B(3, 1) * B(1, :)
B =
```

```
1.0000    0.5000    0.3333    1.0000
0.5000    0.3333    0.2500   -1.0000
     0    0.0833    0.0889    0.6667
```

Aufgabe: Verwende nur Zeilenoperationen, um die Untermatrix der ersten drei Spalten von `B` in die Einheitsmatrix `eye(3)` umzuwandeln. Was steht dann in der vierten Spalte? (Arbeiten Sie im Editor, wir brauchen diese Folge von Befehlen gleich noch einmal.)

MATLAB stellt das Gaußsche Verfahren in dem Operator \ ('Gegenschrägstrich', engl. *backslash*) zur Verfügung. Durch den Befehl `x = A \ b` wird dem Vektor `x` die Lösung des Gleichungssystems $A\,x = b$ zugewiesen, unter Einsatz eines optimierten Gaußschen Verfahrens.

Im Prinzip könnte man die Gleichungen auch durch `x = inv(A) * b` auflösen, indem die Gleichung $A\,x = b$ von links mit der Inversen von `A` multipliziert wird, allerdings ist das Verfahren mit \ i.A. schneller und vor allem robuster (gegen numerische Ungenauigkeiten).

Aufgabe: Wende die gleichen Schritte und Zeilenumformungen wie in der letzten Aufgabe an auf die Matrix

$$B = \begin{pmatrix} 1 & 1/2 & 1/3 & 1 & 0 & 0 \\ 1/2 & 1/3 & 1/4 & 0 & 1 & 0 \\ 1/3 & 1/4 & 1/5 & 0 & 0 & 1 \end{pmatrix}$$

Welche Matrix steht dann in den letzten drei Spalten der umgewandelten Matrix `B`?

2 Programmierung in MATLAB

2.1 Kontrollstrukturen

Die wichtigen Kontroll- und Steuerelemente von MATLAB sind if-Aweisungen, `if ... else ...end`, for-Schleifen `for ... end` und while-Schleifen `while ... end`.

Das prinzipielle Aussehen einer **if-Anweisung** lautet wie folgt:

```
if x > 0
    y = sqrt(x)
end
```

`if` und `end` umgreifen Bedingung und Anweisung. Falls die Bedingung wahr ist, wird die Anweisung durchgeführt. Mit `elseif` lassen sich in die if-else-Schleife weitere Bedingungen einfügen; es kann mehr als ein `elseif` geben. Die if-Schleife wird mit `else` erweitert: Die Anweisungen nach `else` werden dann ausgeführt, wenn die vorherigen Bedingungen nicht erfüllt sind.

```
if x > 0
    y = sqrt(x);
elseif x == 0
    y = 0;
else
    y = NaN;
end
```

Mit der **for-Schleife** kann eine Abfolge von Befehlen mit 'Laufindex' automatisiert werden. Das prinzipielle Aussehen ist `for` gefolgt von 'variable=ausdruck', dann auszuführende Anweisungen und `end`.

Man beachte, dass am Ende jeder Zuweisung ein Semikolon `;` stehen sollte, damit unerwünschte Zwischenergebnisse nicht im *Command Window* erscheinen. Das ist besonders wichtig in Schleifen, wenn solche Ausgaben immer wieder in der Ausgabe erscheinen würden.

Beispiel: Berechne die ersten 20 Glieder der *rekursiv* definierten Folge a_n mit Startwert $a_1 = 0$ und Rechenvorschrift $a_{n+1} = a_n^2 + 1/4$.

```
>> a = zeros(1, 20);        % a(1) ist schon 0.0
>> for i = 2:20
     a(i) = a(i-1)^2 + 0.25;
   end
>> a
a =
  Columns 1 through 8
         0    0.2500    0.3125    0.3477    0.3709    0.3875    0.4002         0.4102
  Columns 9 through 16
    0.4182    0.4249    0.4305    0.4354    0.4395    0.4432    0.4464         0.4493
  Columns 17 through 20
    0.4519    0.4542    0.4563    0.4582
```

Aufgabe: Gegen welchen Grenzwert konvergiert die Folge (a_n)?

Beispiel: Eine *Hilbert*-Matrix $H = (h_{ij})$ hat folgende Einträge: $h_{ij} = 1/(i + j - 1)$. Man kann eine Hilbert-Matrix der Grösse (5×5) zum Beispiel erzeugen durch zwei ineinander geschachtelte for-Schleifen.

```
>> H = zeros(5, 5)
>> for i = 1:5
     for j = 1:5
       H(i, j) = 1/(i+j-1);
     end
   end
>> H
H =
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

Natürlich ist das nicht auf zwei ineinander geschachtelte for-Schleifen beschränkt. Nebenbei gesagt gibt es die Funktion `hilb(5)`, welche genau diese Matrix zurück gibt.

Das prinzipielle Aussehen einer **while-Schleife** lautet wie folgt: Auf `while` folgt eine Bedingung, die geprüft wird, danach Anweisungen, die ausgeführt werden sollen und `end`. Im Gegensatz zur for-Schleife wird die Anzahl der gewünschten der Iterationen nicht vorgegeben; stattdessen wird iteriert, solange die Bedingung wahr ist.

Beispiel: Bekanntlich divergiert die harmonische Reihe, also $1 + 1/2 + 1/3 + ...$ Es gibt also ein n, so dass $1 + 1/2 + 1/3 + ... + 1/n > 10$. Wir addieren diese kleiner werdenden Brüche solange, bis die Summe größer als 10 wird und geben die Anzahl der Summanden aus.

```
>> s = 0;
>> n = 0;
>> while s <= 10.0
     n = n + 1;
     s = s + 1/n;
   end
>> n
n =
       12367
```

Die Divergenz der harmonischen Reihe ist sehr langsam.

Die Schleifen-Konstrukte `for` und `while` können vorzeitig verlassen werden; dazu gibt es die Schlüsselworte `break` und `continue`. Die `break` Anweisung ist eine Sprunganweisung und dient dazu, aus einer Schleife herauszuspringen. Die Programmausführung wird unmittelbar hinter der Schleife fortgeführt (hinter dem `end`, das zu dieser Schleife gehört).

`continue` ist ebenfalls eine Sprunganweisung und bewirkt, dass zur nächsten Iteration einer for- oder while-Schleife übergegangen wird. Innerhalb der Schleife folgende Anweisungen werden nicht mehr ausgeführt.

2.2 Skripte

Ein Skript enthält eine Folge von MATLAB Anweisungen, die inhaltlich zusammengehören und in einer Datei mit der Dateierweiterung (engl. *extension*) '.m' abgespeichert werden. Die Datei muss im Suchpfad von MATLAB liegen, zum Beispiel im *Current Folder* Fenster sichtbar sein.

Die Anweisungen werden ausgeführt, wenn man den Dateinamen (ohne `.m` Erweiterung) im *Command Window* eingibt. Die Variablen in einem Skript sind *global*, das heisst sie sind im

Command Window verfügbar. Umgekehrt kennt ein Skript alle Variablen alle Variablen, die in MATLAB bereits definiert wurden (und im *Workspace* sichtbar sind).

Beispiel: Wir wollen für alle Zahlen von 1 bis 1000 das sogenannte "(3n+1)-Problem" (auch Collatz-Vermutung) durchrechnen.

> Beginne mit irgendeiner natürlichen Zahl n > 0. Ist n gerade, so nimm als nächstes n/2. Ist n ungerade, so nimm als nächstes 3n + 1. Wiederhole die Vorgehensweise mit der erhaltenen Zahl bis die Zahl 1 erreicht wird.

Beginnen wir mit $n = 17$, dann entsteht die Folge $(17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1)$ der Länge 13 (mit 12 Schritten) und mit höchster Zahl 52. Bei 1 wird abgebrochen, weil sich die Folge sonst mit $(1, 4, 2, 1, ...)$ unendlich wiederholen würde.

Es ist eine mathematische Vermutung, dass für jeden Startwert n die Folge nach endlich vielen Schritten mit 1 endet. Diese Vermutung ist unbewiesen.

Wir wollen ein Skript erstellen, dass mit einer Zahl n beginnt und die entstehende Folge ausrechnet. Dazu erstellen wir im Editor eine Datei "collatz.m":

```
n = 17;
if mod(n, 2) == 0
    n = n / 2;
else
    n = 3*n + 1;
end
n
```

Die MATLAB Funktion `mod(n, m)` bestimmt den Rest bei der ganzzahligen Division von `n` durch `m`. Daher ist `mod(n, 2)` gleich 0, wenn `n` gerade ist und sonst 1. `mod(n, 2) == 0` also wahr genau dann wenn `n` gerade ist. In diesem Fall wird der erste Zweig der *if*-Anweisung ausgeführt (also `n=n/2`), sonst der zweite (`n=3*n+1`).

Führen wir das im *Command Window* aus, erhalten wir das erwartete Ergebnis.

```
>> collatz
n =
    52
```

Das Ziel ist, diese Folge fortzusetzen, bis $n = 1$ erreicht wird. Dazu bietet sich eine `while`-Schleife an. Die Syntax ist wieder ganz einfach.

```
n = 17;
while n > 1
if mod(n, 2) == 0
    n = n / 2;
else
    n = 3*n + 1;
end
end
n
```

Da die `while`-Schleife nachträglich hinzugefügt wurde, stimmt die Einrückung der Befehle nicht. Auswählen aller Zeilen zwischen `while` und (seinem) `end` und Drücken der Taste rückt diese Zeilen ein.

Es soll aber auch die Anzahl der Schritte und die grösste in der Folge auftretende Zahl ermittelt werden, also fügen wir noch zwei Variablen `anz_schritte` und `max_n` ein und berücksichtigen

sie in jedem Schritt: `anz_schritte` wird um 1 erhöht und `max_n` ggf. vergrössert.

```
n = 17;
max_n = n;
anz_schritte = 0;
while n > 1
    anz_schritte = anz_schritte + 1;
    if mod(n, 2) == 0
        n = n / 2;
    else
        n = 3*n + 1;
    end
    if n > max_n
        max_n = n;
    end
end
n, anz_schritte, max_n
```

Beachten Sie, dass wir uns in der `while`-Schleife darauf verlassen, dass die Collatz-Vermutung stimmt und irgendwann 1 erreicht wird, andernfalls laufen wir in eine unendliche Schleife – die man in MATLAB mit >CTRL>-C abbrechen müsste.

```
>> collatz
n =
     1
anz_schritte =
    12
max_n =
    52
```

Wir wollen für alle Zahlen $n = 1$ bis 1000 diese Rechnung durchführen und uns für jedes n das Maximum und die Anzahl der Schritte merken. Dazu bietet sich eine Matrix an mit 1000 Zeilen und 3 Spalten.

Als Kontrollstruktur werden wir eine `for`-Schleife benutzen, wobei mit `for i = 1:N` bis `end` für jedes $i = 1, ..., 1000$ die Anweisungen dazwischen ausgeführt werden. Wir benutzen noch `N = 1000` als Konstante, um den Bereich, in dem wir die Rechnungen ausführen, leichter verändern zu können.

```
N = 1000;
for i = 1:N
    n = i;
    max_n = n;
    anz_schritte = 0;
    while n > 1
        anz_schritte = anz_schritte + 1;
        if mod(n, 2) == 0
            n = n / 2;
        else
            n = 3*n + 1;
        end
        if n > max_n
            max_n = n;
        end
```

```
        end
        R(i, :) = [i, anz_schritte, max_n];
    end
```

Wenn dieses Skript ausgeführt wird, erscheint kein Ergebnis, aber die Matrix `R` ist vorhanden und kann angezeigt werden.

```
>> collatz
>> R(1:10, :)
ans =
     1    0    1
     2    1    2
     3    7   16
     4    2    4
     5    5   16
     6    8   16
     7   16   52
     8    3    8
     9   19   52
    10    6   16
```

Mit `max(R)` werden die Maxima aller drei Spalten angezeigt.

```
>> max(R)
ans =
        1000       178    250504
```

Aufgabe: Für welche Zahlen n ist die Anzahl der Schritte grösser oder gleich 170? Für welche n ist 250504 die grösste erreichte Zahl.

Für mehr Flexibilität in der Wahl von `N` kann man die Zeile `N = 1000` ersetzen durch `if ~exist('N'), N = 1000; end`. Dann kann N in MATLAB gesetzt werden, das Skript erkennt, dass diese Variable einen Wert besitzt und benutzt diesen.

```
>> N = 10000;
>> collatz
>> max(R)
ans =
       10000         261    27114424
```

Aufgabe: Erzeuge einen Plot der Anzahl Schritte für n von 1 bis 10000.

2.3 Benutzerdefinierte Funktionen

In den meisten Anwendungen muss der Benutzer seine eigenen Funktionen aufstellen können, um Probleme zu lösen. Für ganz kleine Funktionen, die in einer Zeile geschrieben werden können, gibt es die Möglichkeit sogenannter "anonymer" Funktionen, mit 'Klammeraffe' `@` (engl. *at-sign*) und Klammern für die Angabe der Argumente.

```
>> fun1 = @(x) exp(-0.05 * x) .* sin(x);
```

Mit `fplot` lässt sich die Funktion einfach plotten und stellt offenbar eine gedämpfte Schwingung dar.

```
>> fplot(@(x) exp(-0.05 * x) .* sin(x))     % oder fplot(fun1)
```

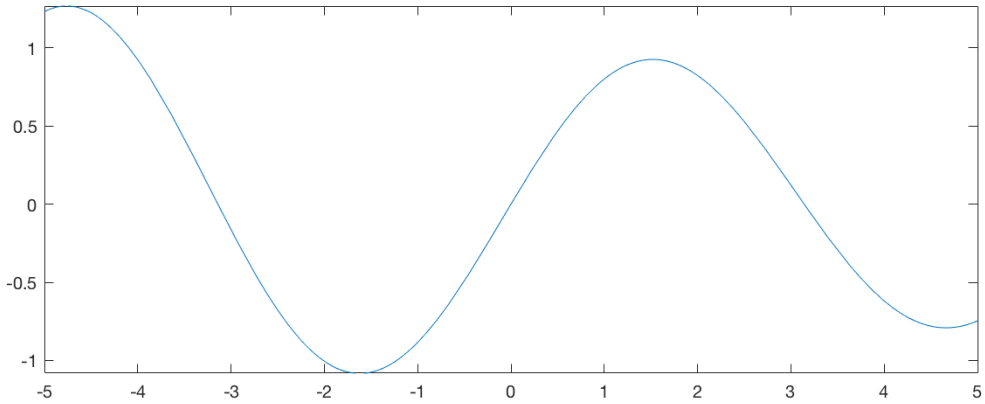

Abbildung 1: Figure: Gedämpfte Schwingung

Der Benutzer kann aber auch eigene, kompliziertere Funktionen schreiben und damit die Funktionalität von MATLAB deutlich zu erweitern. Alle diese Funktionen, und auch die MATLAB-internen Funktionen, sind in Funktionsdateien abgelegt, und zwar pro Funktion *eine* Datei. Der Name der Datei muss mit dem Namen der Funktion übereinstimmen, ergänzt um die Erweiterung '.m'.

Die Syntax zur Definition einer Funktion ist genau vorgeschrieben. ie erste (ausführbare) Zeile muss die Form "function rückgabewert = (argumente)" haben, die ganze Funktion kann (muss aber nicht) mit einem `end` beendet werden. Innerhalb dieses Rahmens können alle MATLAB Befehle benutzt werden.

Die obige Funktion unter dem Namen 'fun1' und geschrieben in einer Funktionsdatei "fun1.m" wird folgendermassen aussehen:

```
function y = fun1(x)
y = exp(-0.05 * x) .* sin(x);
end
```

Wichtig: Vergessen Sie nicht, im Editor die Datei zu speichern! Der Name der Datei muss links im *Current Folder* Fenster zu sehen sein.

Im *Command Window* oder in einer anderen Funktion kann "fun1" aufgerufen wie eine interne MATLAB FUnktion. Da `.*` für die Multiplikation benutzt wird, ist diese Funktion sogar vektorisiert.

```
>> x = linspace(0, 2*pi, 5);
>> y = fun1(x);
>> y
y =
     0    0.9245    0.0000   -0.7901   -0.0000
```

Variablen in Functionsdateien sind lokale Variablen, die in der globalen Umgebung (engl. *scope*) des *Command Windows* nicht bekannt sind. Die Übergabe einzelner Variablen erfolgt über die Parameterliste im Funktionsaufruf. Werte können an die aufrufende Funktion nur über Parameter zurückgegeben werden.

Beim Versuch, diese Funktion mit `fplot` zu plotten, gibt es eine Fehlermeldung:

```
>> fplot(fx23)
Not enough input arguments.
```

```
Error in fx23 (line 2)
x2 = x.^2;
```

Diese Meldung ist unverständlich. Tatsächlich erkennt MATLAB `fx23` nicht als Funktion, ebenso wie es `sin` nicht erkennen wird. Um MATLAB zu zwingen, dies als Funktion zu erkennen, wird der Klammeraffe `@` verwendet.

```
>> fplot(@fx23)
```

Dies wird den gleichen Plot erzeugen wie oben mit `fplot(fun1)`.

Wichtig: `@` wird für alle Funktionen benötigt, die in Dateien definiert werden, u.a. für alle MATLAB-internen Funktionen. Es darf nicht benutzt werden für anonyme Funktionen (die schon mit `@` definiert werden).

Funktionen können Vektoren oder Matrizen zurückgeben, aber auch mehrfache Werte. Die folgende Funktion `fx23` berechnet das Quadrat und die dritte Potenz des Argumentes:

```
function [x2, x3] = fx23(x)
x2 = x.^2;
x3 = x.^3;
end
```

Um aus dieser Funktion beide Werte zu erhalten, muss die Funktion in der Form `[y1, y2] = fx23(x)` aufgerufen werden, sonst wird nur ein Wert zurückgegeben.

```
>> y1 = fx23(1.5)
y1 =
    2.2500
>> [y1, y2] = fx23(1.5)
y1 =
    2.2500
y2 =
    3.3750
```

Aufgabe: Implementiere die folgende Funktion, die offenbar eine if-Anweisung erfordert und nicht ohne weiteres als einzeilige Funktion geschrieben werden kann.

$$\text{myabs}(x) = \begin{cases} 0 & \text{für } x < 0 \\ x & \text{für } x \geq 1 \end{cases}$$

Aufgabe: Ist ihre Version der Funktion vektorisiert? Wie kann man diese Funktion so implementieren, dass sie vektorisiert ist?

Beispiel Ziel ist es eine Funktion `wurzel(a, n)` zu schreiben, die für $x >= 0$ die n-te Wurzel von a berechnet.

Gesucht ist eine Nullstelle der Funktion $f(x) = x^n - a$. Dem Newtonschen Verfahren gemäss wird eine Iteration

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^n - a}{n x_i^{n-1}} = \frac{1}{n}((n-1)x_i + \frac{a}{x_i^{n-1}})$$

mit beliebigem Startwert $x_1 > 0$.

```
function y = wurzel(a, n)
x = 1.0;
y = 1/n * ((n-1)*x + a/x^(n-1));
```

```
    while (abs(y-x) > 1e-15)
        x = y;
        y = 1/n * ((n-1)*x + a/x^(n-1));
    end
    end
```

Da die n-te Wurzel für $a < 0$ nicht existiert, sollte in diesen Fall eine Fehlermeldung ausgegeben werden. Dafür gibt es den Befehl `error`, der mit einer geeigneten Meldung zu versehen ist, innerhalb einer `if`-Abfrage:

```
if a < 0
    error('Argument a muss eine Zahl grösser Null sein.')
end
```

Wenn $a = 0$ oder $a = 1$ ist, sollte keine Berechnung durchgeführt werden, da das Ergebnis bekannt ist. Der Befehl `return` ermöglicht es, die Funktion *sofort* zu verlassen, die folgenden Befehle werden nicht mehr ausgeführt. Den Rückgabewerten müssen aber vorher Werte zugewiesen werden.

```
if a == 0 || a == 1
    y = a;
    return
end
```

Dabei bezeichnet der `||` Operator eine logische *oder*-Verknüpfung, entsprechend würde `&&` eine logische *und*-Verknüpfung bezeichnen.

Die Funktion soll weiterhin auch in der Form `wurzel(a)` aufgerufen werden können, in diesem Fall soll automatisch $n = 2$ gesetzt, also die Quadratwurzel bestimmt werden. Dafür gibt es in MATLAB die Variable `nargin`, die bei jedem Aufruf einer Funktion definiert ist und die Anzahl der mitgegebenen Argumente beinhaltet.

Für `wurzel(a, n)` ist `nargin` 2, für `wurzel(a)` nur 1. Daher setzt der folgende Code einen Fehlwert für `n`. Für einen Aufruf `wurzel()` oder nur `wurzel` wird MATLAB weiterhin eine eigene Fehlermeldung zeigen.

```
if nargin < 2
    n = 2;
end
```

Übrigens gibt es auch eine Variable `nargout`, welche die beim Aufruf angeforderten Rückgabewerte zählt.

Insgesamt sieht die `wurzel`-Funktion jetzt folgendermassen aus:

```
    function y = wurzel(a, n)
    if a < 0
        error('Argument a muss eine Zahl grösser Null sein.')
    end
    if a == 0 || a == 1
        y = a;
        return
    end
    if nargin < 2
        n = 2;
    end
```

```
    %-- Beginn der Berechnung ---
    x = 1.0;
    y = 1/n * ((n-1)*x + a/x^(n-1));
    while (abs(y-x) > 1e-15)
        x = y;
        y = 1/n * ((n-1)*x + a/x^(n-1));
    end
    end
```

Aufgabe: Füge einen weiteren Rückgabewert `e` ein, der die Anzahl der Schritte mitzählt und beim Aufruf `[w, e] = wurzel(a, n)` diese Zahl der benötigten Schritte anzeigt.

```
>> [w, e] = wurzel(1.20/0.05, 60)  % 60-te Wurzel von 24
w = 1.0544
e = 23
```

Beispiel: Die Funktion `bits(a)` soll für eine Zahl a zwischen 0 und 1 die binäre Darstellung (im Zweiersystem) als Zeichenkette zurückgeben, und zwar auf 53 Stellen, so wie sie im Computer abgespeichert werden.

Ist also $0 < a < 1$, dann ergibt sich die Darstellung im Zweiersystem zu

$$a = 0 + \frac{b_1}{2^1} + \frac{b_2}{2^2} + ... + \frac{b_{53}}{2^{53}} + ...$$

wobei b_i gleich 0 oder 1 ist für alle i.

Prüfen Sie nach, dass die folgende Rechnung korrekt ist.

```
function b = bits(a)
if a <= 0 || a >= 1
    error('Argument a muss zwischen 0 und 1 liegen.')
end
b = '0.';
for i = 1:53
    if a >= 1/2^i
        a = a - 1/2^i;
        b = [b, '1'];
    else
        b = [b, '0'];
    end
    if a == 0
        break
    end
end
```

Dabei bedeutet `['Hello ', 'World']` dass zwei Zeichenkette zu einer zusammengefügt, also hintereinander geschrieben werden: 'Hello World'.

```
>> bits(1/2 + 1/8 + 1/16)
ans =
0.1011
>> bits(0.2)
ans =
0.00110011001100110011001100110011001100110011001100110011001100110
```

Aufgabe: Schreibe eine Funktion `bits3`, welche die *ternäre* Darstellung (im Dreiersystem) einer Zahl $0 < a < 1$ erzeugt, mit den Ziffern '0', '1', und '2'.

Aufgabe: Erstellen Sie eine Funktion `agm(a, b)`, welche für zwei Zahlen a und b das **algebraisch-geometrische Mittel* berechnet.

3 Numerische Methoden

3.1 Nullstellenbestimmung

Matlab bietet vorgefertigte Funktionen zum Bestimmen von Nullstellen an, aber es ist nicht schwer, auch eigene Methoden zur Nullstellenbestimmung zu programmieren, zum Beispiel die Bisektionsmethode und das Newtonverfahren.

Die MATLAB Funktion zur Bestimmung der Nullstelle einer Funktion einer Veränderlichen ist `fzero`. Diese Funktion bestimmt genauer gesagt den Ort eines Vorzeichenwechsels einer Funktion. Aus diesem Grund ist es nicht möglich, mit dieser Methode die Nullstelle von Funktionen ohne einen solchen Vorzeichenwechsel, z.B. von $f(x) = x^2$ zu bestimmen.

Zunächst wird f als anonyme Funktion `f` definiert:

```
>> f=@(x) sin(x)+x^2-1;
```

Um einen Überblick zu erhalten, in welchem Bereich die Nullstelle zu suchen ist, ist es immer eine gute Idee, die Funktion zunächst mit `fplot` zu zeichnen. Damit der Nulldurchgang der Funktion besser zu sehen ist, wird noch ein Gitter hinterlegt.

```
>> fplot(f)
>> grid on
```

Man kann sehen, dass die Funktion die x-Achse zweimal zwischen den Werten -2 und 1 schneidet.

Gesucht ist zunächst die Nullstelle zwischen den x-Werten $a = -2$ und $b = 0$. Dabei kann entweder in der Nähe eines x-Wertes, `x_0 = fzero(f,a)`, oder oder in dem Intervall $[a, b]$ zwischen a und b gesucht werden: `x_0 = fzero(f,[a b])`.

```
>> fzero(f, [-2, 0])
ans = -1.4096

>> fzero(f, [0, 1])
ans =  0.6367
```

Zu beachten ist, dass `fzero` immer nur eine Nullstelle findet und dann aufhört. Welche Nullstelle das ist, ist nicht ohne weiteres vorherzusehen. Daher muss eine gesuchte Nullstelle in einem Interval ausreichend eingegrenzt werden.

`fzero` wird keine Funktionen ohne Vorzeichenwechsel lösen. Betrachte dazu das Beispiel $f(x) = x^2$ welche $x = 0$ als NUllstelle hat.

```
>> fzero(@(x) x^2, [-1, 1])
Error using fzero (line 272)
The function values at the interval endpoints must differ in sign.
```

In diesem Beispiel ist f ein Polynom, in MATLAB repräsentiert durch den Vektor `[1, 0, 0]`, daher bietet sich die `roots` Funktion an:

```
>> roots([1, 0, 0])
ans =
      0
      0
```

$x_0 = 0$ ist eine Doppelnullstelle. Leider funktioniert dieser Ansatz nicht für allgemeine Funktionen. Eine andere Möglichkeit wäre, das Newtonsche Verfahren zu implementieren. Für einen geeigneten

Startwert x_1 wird die Folge

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

gegen eine Nullstelle x_0 konvergieren. Allerdings wissen wir noch nicht, wie wir die Ableitung einer Funktion numerisch berechnen sollen.

Hinweis: Eine weitere Möglichkeit wäre, das *Sekantenverfahren* zu implementieren, eine Variante des Newton-Verfahrens, das ohne die Ableitung auskommt.

Aufgabe: Die Kapazität eines eines Diffusionsvorganges werde annähernd mit dem Ausdruck $C(r) = 2r^2 + 3\log(r) + 1$ berechnet. Bestimme ein r so dass die Kapazität gleich 2 ist.

Aufgabe: Die Funktion $f(x) = x\,e^{-x}$ taucht häufig in technischen Anwendungen auf. Bestimme den Punkt x mit $f(x) = 1$, die sogenannte Omega-Konstante. (Definiere die Funktion unter dem Namen 'xexp' in einer Funktionsdatei.)

3.2 Minima und Maxima

Eine ebenso wichtige Aufgabe ist es, Minima und/oder Maxima einer Funktion zu bestimmen, insbesondere in einer Zeit, die so sehr auf Optimierung und Effizienz ausgerichtet ist.

Für Polynome erscheint das einfach. Aus der Analysis (Kurvendiskussion) wissen wir, dass in einem *Optimum* (Minimum bzw. Maximum) x einer differenzierbaren Funktion f gelten muss: $f'(x) = 0$. Das heisst, ein Minimum ist zugleich Nullstelle der Ableitung.

Beispiel: $p(x) = x^3 - x^2 - 4x + 4$. Die Ableitung wird mit `polyder` berechnet und deren Nullstelle mit `roots` bestimmt.

```
>> p = [1, -1, -4, 4];
>> p1 = polyder(p)
p1 =  3    -2    -4         % 3*x^2 - 2*x - 4
>> r1 = roots(p1)
r1 =
    1.5352
   -0.8685
>> polyval(p, r1)
ans =
   -0.8794
    6.0646
```

Das Polynom hat ein Maximum bei $x = -0.8794$ und ein Minimum bei $x = 1.5352$. Diese Optima sind *lokal*, d.h. sie sind minimal oder maximal in einer Umgebung des Punkte. Global gesehen läuft das Polynom natürlicher gegen `Inf` bzw. `-Inf`.

(Genau genommen müsste noch die zweite Ableitung benutzt werden, um zu sehen, was die Minima und Maxima sind.)

Für die Bestimmung des Minimums einer allgemeinen Funktion gibt es in MATLAB den Befehl `fminbnd`. Dabei steht 'bnd' für "bounded", also für das Minimum einer Funktion in einem beschränkten Bereich, einem Intervall. Der Aufruf von `fminbnd` ist ähnlich wie der von `fzero`, nur das Intervall wird nicht durch einen Vektor, sondern als zwei getrennte Argumente übergeben.

Im Beispiel scheiben wir das Polynom als anonyme Funktion, dann ist:

```
>> p = @(x) x.^3 - x.^2 - 4*x + 4;
```

```
>> fminbnd(p, -3, 3)
ans = 1.5352
```

Um gleichzeitig noch den Funktionswert im Minimum zu erhalten, muss `fminbnd` nur mit einem zusätzlichen Ausgabeparameter aufgerufen werden.

```
>> [xmin, fmin] = fminbnd(p, -3, 3)
xmin =  1.5352
fmin = -0.8794
```

Gibt es mehrere Minima in dem angegebenen Bereich, wird ein Minimum gefunden, aber nicht notwendigerweise das mit dem kleinsten Funktionswert.

Um Maxima zu finden, "drehen wir die Funktion um", das heisst multiplizieren sie mit -1, denn ein Minimum der Funktion f ist ein Maximum der Funktion $-f$, und umgekehrt.

```
>> [xmax, fmax] = fminbnd(@(x) -p(x), -3, 3)
xmax = -0.8685
fmax = -6.0646      % besser: fmax = 6.0646
```

Der Wert von `fmax` muss dann wieder mit -1 multipliziert werden!

Aufgabe: Gegeben die gedämpfte Schwingung $s(x) = e^{-0.05x}\sin(x)$. Die Sinus-Funktion hat ihr erstes Maximum in $x = \pi/2$, wo hat die Funktion s ihr erstes Maximum, und um wieviel früher oder später als $\pi/2$?

3.3 Numerische Differentiation

Die Ableitung bzw. der Differenzialquotient eine Funktion f in einem Punkt x_0 wird in der Analysis definiert als der Grenzwert

$$\lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h}$$

Einen solchen Grenzwert kann man numerisch nicht nachbilden. Daher müssen wir uns mit einem geeigneten, wahrscheinlich sehr kleinen h begnügen. Wenn h positiv gewählt wird, berechnen wir hier offenbar den "rechsseitigen", für negatives h den linksseitigen Grenzwert.

Für eine differenzierbare Funktion sollte der rechtsseitig und der linksseitige Grenzwert gleich sein. Daher liegt es nahe, den Mittelwert aus diesen beiden Werten als numerischen Differentialquotienten zu nehmen.

$$\frac{1}{2}\left(\frac{f(x_0+h) - f(x_0)}{h} + \frac{f(x_0) - f(x_0-h)}{h}\right) = \frac{f(x_0+h) - f(x_0-h)}{2h}$$

Der Ausdruck rechts heisst auch "zentraler Differenzquotient" (engl. *central differnce formula*). Die folgende Funktion `deriv(f, x0, h)` , in einer Datei `deriv.m`, berechnet diesen Ausdruck für eine Funktion `f`, einen Punkt `x0` und ein `h` grösser als Null.

```
function df = deriv(f, x0, h)
df = (f(x0 + h) - f(x0 - h)) / (2*h);
end
```

Wie klein soll h gewählt werden? Dazu machen wir einen Test. Wir bestimmen diesen numerischen Differenzialquotienten für die Exponentialfunktion e^x im Punkt $x = 1$ für verschiedne h. Die Ableitung in diesem Punkt kennen wir, sie ist e.

```
>> e = exp(1);
   for i = 1:15
     df = deriv(@exp, 1.0, 10^-i);
     fprintf('%2d  %3.15f  %2.4e\n', i, df, abs(df-e))
   end
 1  2.722814563947418  4.5327e-03
 2  2.718327133382692  4.5305e-05
 3  2.718282281505724  4.5305e-07
 4  2.718281832989611  4.5306e-09
 5  2.718281828517633  5.8587e-11
 6  2.718281828295588  1.6346e-10
 7  2.718281828517632  5.8587e-11
 8  2.718281821856294  6.6028e-09
 9  2.718281821856294  6.6028e-09
10  2.718281155722480  6.7274e-07
11  2.718292257952725  1.0429e-05
12  2.718492098097158  2.1027e-04
13  2.717825964282383  4.5586e-04
14  2.708944180085382  9.3376e-03
15  2.886579864025407  1.6830e-01
```

Warum der absolute Fehler für kleinere $h < 10^{-7}$ wieder grösser wird, klären wir besser mündlich in der Vorlesung. Auf jeden Fall scheint ein Wert um $h = 10^{-6}$ am geeignetsten zu sein – und die Theorie bestätigt das. Daher ändern wir die `deriv` Funktion so dass als dies als 'Fehlwert' gesetzt ist.

```
function df = deriv(f, x0, h)
if nargin < 3, h = 1e-06; end            % h = nthroot(eps, 3)
df = (f(x0 + h) - f(x0 - h)) / (2*h);
end
```

Um zum Beispiel eine Funktion zu schreiben, welche die Ableitung der Funktion $f(x) = \sin(x)\cos(x)$ darstellt, genügt daher:

```
>> f  = @(x) sin(x) .* cos(x);
>> df = @(x) deriv(f, x);
```

Aufgabe: Vergleiche die Funktion `df` mit der symbolischen Ableitung von `f`, am besten an Hand eines Plots und des absoluten Fehlers über das Intervall $[0, 2\pi]$.

Ebenso ist es jetzt möglich, das Newton-Verfahren zu implementieren, ohne symbolisch die Ableitung der Funktion bestimmen zu müssen.

```
function x0 = newton(f, x1)
x2 = x1 - f(x1) / deriv(f, x1);
while abs(x2 - x1) > 1e-15
    x1 = x2;
    x2 = x1 - f(x1) / deriv(f, x1);
end
x0 = x2;
end
```

Aufgabe: Bestimme die Nullstelle der Funktion $f(x) = sin^2(x)$ im Intervall $[2, 4]$ mit Hilfe des Newton-Verfahrens. (`fzero` würde nicht zum Ergebnis führen.)

3.4 Numerische Integration

Das Integral einer Funktion f auf einem Intervall $[a, b]$ ist definiert als Grenzwert der Summen

als Annäherung an eine Bestimmung der Fläche zwischen dem Funktionsgraphen und der x-Achse. Ähnlich wie der Differentialquotient kann dieser Ausdruck numerisch angenähert werden durch einen endlichen Ausdruck.

Eine Möglichkeit ist die Mittelpunktsformel (engl. *midpoint formula*): Das Intervall $[a, b]$ wird in eine endliche Zahl von gleich grossen Teilintervallen einer Länge h zerlegt und die Teilfläche durch das Produkt aus h und dem Funktionswert in der Mitte des Teilintervals abgeschätzt. Für genügend kleines h bzw. eine genügend grosse Anzahl von Teilintervallen sollte das Integral recht gut betimmt werden können.

Die folgende Funktion `midpt` implementiert diesen Ansatz. `n` ist die Anzahl der Teilintervalle.

```
function I = midpt(f, a, b, n)
if nargin < 4, n = 100; end
h = (b - a) / n;
x = a:h:b;
s = 0;
for i = 1:(n-1)
    s = s + f(x(i) + h/2);
end
I = h * s;
end
```

Als Beispiel berechne das Integral der Sinusfunktion von 0 bis π.

```
>> midpt(@sin, 0, pi)
ans = 1.9996            % Fehler: 0.0004
```

Das richtige Ergebnis wäre $cos(0) - cos(\pi) = 2$. Mit 200 Teilintervallen wird das Ergebnis besser.

```
>> midpt(@sin, 0, pi, 200)
ans = 1.9999            % Fehler: 0.0001
```

In Mathematik-Vorlesungen lernt man die *Simpson-Formel* als eine sehr gute Näherung eines Integrals kennen. Dabei wird die zu integrierende Funktion durch eine quadratische Funktion approximiert, nicht durch einen linearen Ausdruck, wie in der Mittelpunktsregel (oder der Trapezregel).

Rechnet man die Summen über alle Teilintervalle zusammen, entsteht der folgende Ausdruck

$$\int_a^b f(x)\,dx \approx \frac{h}{3}(y_1 + 4(y_2 + ... + y_{2n}) + 2(y_3 + ... + y_{2n-1}) + y_{2n+1})$$

Dabei wird das Intervall $[a, b]$ gleichmässig in $2n$ Teilinervalle aufgeteilt, die Funktionswerte in den Stützstellen sind $y_i = f(x_i)$ für $i = 1, ..., 2n + 1$. Die entsprechende MATLAB Funktion kann dann so aussehen (vorausgesetzt, die Funktion f ist vektorisiert):

```
function I = simpson(f, a, b, n)
h = (b - a)/(2*n);
x = a:h:b;
y = f(x);
I = y(1) + y(2*n+1) + 4*(sum(y(2:2:(2*n)))) + 2*(sum(y(3:2:(2*n-1))));
I = I * h / 3;
end
```

Bezogen auf das Beispiel oben verkleinert sich der Fehler bei eine vergleichbaren Anzahl von Teilintervallen deutlich.

```
>> simpson(@sin, 0, pi, 100)
ans = 2.0000            % Fehler: < 10^-9
```

MATLAB verfügt über einen eigenen Befehl zur Integration, `integral`. Diese Funktion benutzt einen *adaptiven* Ansatz. Das heisst, die Breite der einzelnen Teilintervalle variiert: Je stärker die Funktion sich in einem Bereich verändert, desto schmaler werden auch die Teilintervalle.

```
>> integral(@sin, 0, pi)
ans = 2
```

Aufgabe: Berechne das Integral $\int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,\mathrm{d}x$

Die Bogenlänge einer Kurve $y = y(x)$ für $a \le x \le b$ wird berechnet nach der folgenden Formel:

$$s = \int_a^b \sqrt{1 + y'^2(x)}\,\mathrm{d}x$$

Aufgabe: Bestimme die Länge der Sinuskurve zwischen 0 und π, und zwar einmal unter Benutzung der symbolischen Ableitung, einmal unter Benutzung der numerischen Ableitungsfunktion `deriv`.

Aufgabe: Für ein elektrisches Bauteil werde der Energieverbrauch in Abhängigkeit eines Parameters p ermittelt zu $\int_0^\pi \sin(x)\,\cos(p\,x)\mathrm{d}x$. Für welches p ist der Verbrauch am geringsten? Kann das in $\pi/2$ sein (Vermutung der Ingenieure)?

Die Koordinaten (x_S, y_S) des Schwerpunktes einer Fläche, die durch einen Funktionsgraphen $x \to f(x)$, die x-Achse und die senkrechten Linien in $x = a$ und $x = b$ begrenzt wird, berechnen sich zu

$$x_S = \frac{1}{A}\int_a^b x\,f(x)\,\mathrm{d}x, \quad y_S = \frac{1}{A}\int_a^b \frac{f(x)^2}{2}\,\mathrm{d}x$$

Dabei ist A die Gesamtfläche (dieses Flächenstücks), also $A = \int_a^b f(x)\,\mathrm{d}x$.

Wir schreiben eine Funktion `schwerpt(f, a, b)`, welche die Koordinaten des Schwerpunktes berechnet und als einen Vektor `[xS, yS]` und zurückgibt. Innerhalb dieser Funktion braucht es zwei weitere Funktionen, $x \to x\,f(x)$ und $x \to f^2(x)$, die integriert werden müssen. Mit anonymen Funktionen lässt sich das in MATLAB sehr elegant programmieren.

```
function xy = schwerpt(f, a, b)
A = integral(f, a, b);
xs = integral(@(t) t .* f(t), a, b) / A;
ys = integral(@(t) f(t).^2, a, b) / A / 2;
xy = [xs; ys];
end
```

Beispiel: Wo liegt der Schwerpunkt der Fläche, die zwischen 0 und π von der Sinuskurve und der x-Achse begrenzt wird?

```
>> schwerpt(@sin, 0, pi)
ans =
    1.5708
    0.3927
```

Die erste Zahl ist offenbar $\pi/2$, also die Mitte des Intervalls, was aus Symmetriegründen richtig sein muss. Kennen Sie die zweite Zahl?

Entsprechend gibt es auch einen Befehl `integral2`, der das Integral einer zweidimensionalen Funktion, also eine Funktion auf einem Bereich von R^2, berechnet. Der allgemeine Aufruf ist `integral2(fun,xmin,xmax,ymin,ymax)`, wobei die untere und obere Grenze von `y` auch noch von `x` abhängen darf.

Beispiel: Berechne das Integral der Funktion $f(x,y) = \sqrt{1-x^2-y^2}$ auf dem Einheitskreis $x^2+y^2 \leq 1$ im ersten Qudranten ($x \geq 0, y \geq 0$). Wir definieren eine Funktion $y_{max} = \sqrt{1-x^2}$, so dass für festes x von 0 bis $y_{max}(x)$ integriert wird.

```
>> ymax = @(x) sqrt(1 - x.^2);
>> f = @(x, y) sqrt(1 - x.^2 - y.^2);
>> v = integral2(f, 0, 1, 0, ymax)
v =
    0.5236
>> (4/3)*pi - 8*v
ans =
    -6.8301e-13
```

Das Ergebnis sollte ein Achtel des Volumens $\frac{4}{3}\pi r^3$ einer Kugel vom Radius $r = 1$ sein.

Aufgabe: Berechne das Integral $\int_G log(x^2+y^2)\,\mathrm{d}x\,\mathrm{d}y$ auf dem Gebiet G der Ebene, das durch die Kreise $x^2+y^2 = 9$ und $x^2+y^2 = 25$ begrenzt wird.

4 Differentialgleichungen

Das Lösen von Differentialgleichungen und Anfangswertproblemen ist eine der zentralen Aufgabenstellungen der Ingenieurmathematik.

4.1 Einführung DGL

Eine (gewöhnliche) *Differentialgleichung* (DGL) erster Ordnung ist ein Ausdruck der Form

$$y' = \frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y), \quad y(a) = y_0$$

Die rechte Seite ist hierbei eine Funktion f in zwei Variablen x und y. Links steht explizit die Ableitung von y nach x. Dies ist keine explizite Gleichung, sondern eher eine Problembeschreibung:

Gesucht ist eine Funktion $y = y(x)$, welche in einem Intervall $[a, b]$ definiert ist und die *Anfangsbedingung* $y(a) = y_0$ erfüllt.

Erste Ordnung bedeutet hierbei, dass die erste Ableitung die höchste vorkommende Ableitung ist. Und 'gewöhnlich' bedeutet, dass nur Funktionen einer Veränderlichen betrachtet werden.

Beispiel: Eine DGL der Form $y' = f(x)$ (f nur von x abhängig) hat die offensichtliche Lösung $y(x) = \int_a^x f(t)\mathrm{d}t = F(t)|_a^x$ wenn die *Stammfunktion* F von f bekannt ist.

In den Mathematik-Vorlesungen werden verschiedene Methoden zur expliziten Lösung von Differenzialgleichungen vorgestellt. Der einfachste und wichtigste Fall ist vielleicht $y' = k\,y$. Für $k < 0$ beschreibt dies *Zerfallsprozesse*, für $k > 0$ *Wachstumsprozesse*, zum Beispiel Zinseszins-Aufgaben.

Aufgabe:

Oft ist es schwierig, die theoretische Lösung einer Differenzialgleichung zu finden. In vielen Fällen gibt es auch keine explizite Lösung, ähnlich wie für Integrale nicht immer eine explizite Stammfunktion gibt.

Wir sind deshalb an numerischen Verfahren zur Lösung von Differenzialgleichungen interessiert. Zum besseren Verständnis behandeln wir das *Richtungsfeld*, es dient der zeichnerischen Bestimmung einer Näherungslösung einer Differentialgleichung.

Die in der Differenzialgleichung auftretende Funktion f ist auf einem Teil der x-y-Ebene gegeben und definiert dort in jedem Punkt die Steigung einer möglichen Lösung durch diesen Punkt. Die für diesen Kurs erstellte Funktion `richtungsfeld` zeichnet diese Steigungen als Pfeile in einem vorgegebenen rechteckigen Bereich.

```
>> df = @(x, y) sqrt(x.^2 + y.^2)
>> ode45(df, [0, 3], 0)
>> hold on
>> richtungsfeld(df, [0, 3], [0, 10], 20)
>> hold off
```

Das Richtungsfeld veranschaulicht die Ableitungen der Differenzialgleichung $y' = \sqrt{x^2 + y^2}$ und enthält auch den Funktionsgraphen einer Lösung, mit dem Anfangswert $y(0) = 0$.

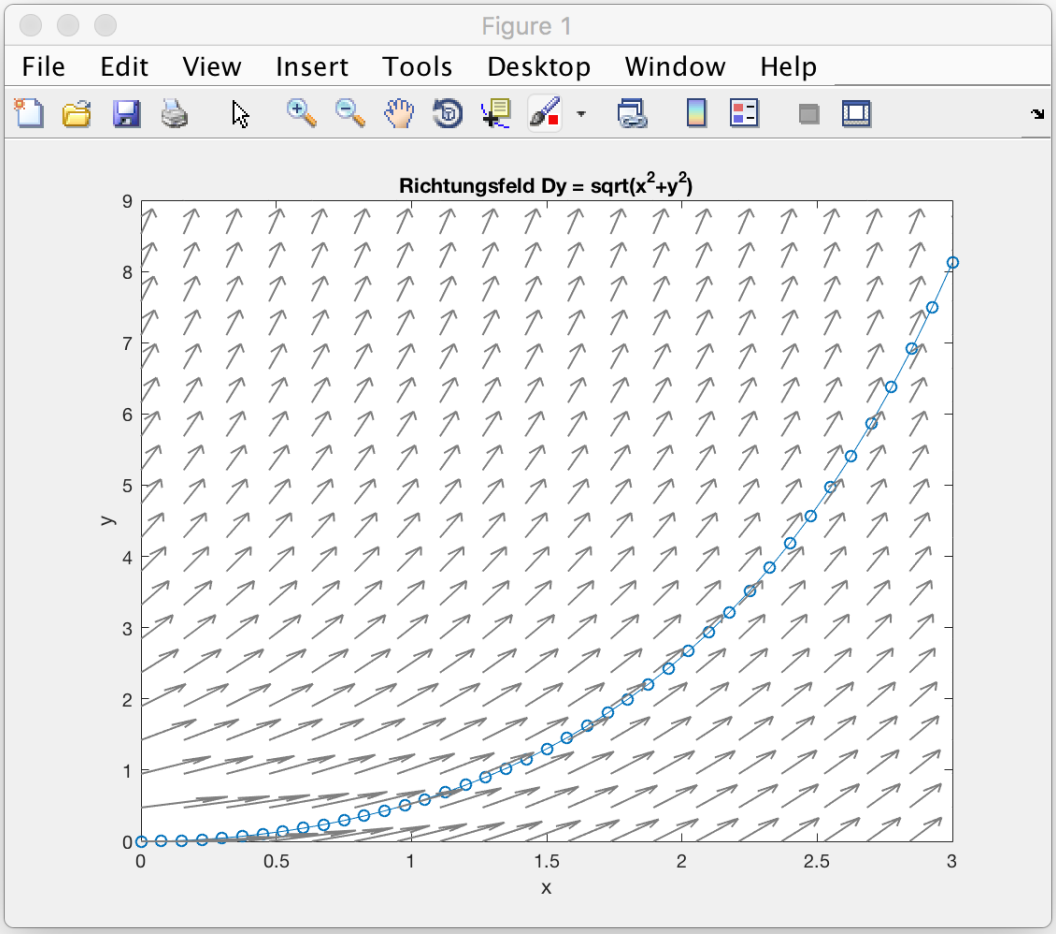

Abbildung 2: Richtungsfeld einer DGL

4.2 Eulers Polygonzug-Verfahren

Ein Anfangswertproblem ist eine Differentialgleichung plus eine Anfangsbedingungen. Gesucht ist die Funktion y in Abhängigkeit von x. Unter gewissen Bedingungen an die Funktion f existiert eine Lösung und ist eindeutig.

Die Idee des Eulerschen Verfahrens ist einfach. Gegeben ein Lösungskurve in einem Punkt x, dann gilt offenbar für eine kleine Schrittweite h

$$y(x+h) \approx y(x) + h \cdot y'(x) = y(x) + h \cdot f(x, y(x))$$

Zusammen mit der Anfangsbedingung liefert und das für äquidistante Punkte Näherungswerte y_k an den Stellen x_k gemäß dieser Vorschrift. Dieses als *explizites Euler-Verfahren* bekannte Vorgehen ist in der folgenden Funktion implementiert.

```
function [x, y] = euler(df, a, b, y0, n)
% Euler Verfahren mit Heun Korrektur
h = (b-a)/n;
x = a:h:b;
y = zeros(1, n+1);
y(1) = y0 ;
for k = 2:n+1
    y(k) = y(k-1) + h * f(x(k-1), y(k-1));
    y(k) = y(k-1) + h * (f(x(k-1), y(k-1)) + f(x(k), y(k))) / 2;
end
end
```

Dabei ist `df` eine Funktion von `x` und `y`, welche die Differenzialgleichung $y' = f(x, y)$ repräsentiert, `[a, b]` das Intervall, in dem eine Lösung gesucht wird, `y0` der Anfangswert in `a`, und `n` die Anzahl der Schritte bzw. `(b-a)/n` die Schrittweite.

Hier ist schon die Euler-Heun Korrektur mit eingearbeitet: Die Steigung im Punkt x_{k-1} wird noch mit der Steigung im neu berechneten Punkt x_k gemittelt, um die Abweichung von der realen Lösung noch kleiner zu halten.

Statt einer analytischen erhalten wir eine numerische Lösung, nämlich in `x` und `y` die x- und y-Koordinaten von Punkten auf der Lösungskurve. Mit `plot(x, y, 'o-')` können die Kurve und die berechneten Punkte geplottet werden.

Aufgabe: Löse die Differenzialgleichung $y' = x + y$ im Intervall $[0, 1]$ mit Anfangsbedingung $y(0) = 0$ mit dem Euler-Verfahren und vergleiche dies mit der analytischen Lösung $y(x) = e^x - x - 1$ graphisch sowie die Endwerte in $b = 1$. Wie gross ist die Abweichung, mit und ohne Heun-Korrektur?

4.3 DGL Löser in MATLAB

Es gibt in MATLAB für das Lösen von Differenzialgleichungen und Anfangswertproblemen vorgefertigte Lösungsverfahren. Diese nennt man ODE-Solver (ODE für engl. *ordinary differential equation*) oder -Löser. Es gibt sogar eine eher verwirrende Vielfalt von solchen Solvern, angepasst an verschiedene Gleichungstypen.

Der wohl am meisten verwendete ODE-Solver ist `ode45`. Dieser Löser implementiert eine Variante des *Runge-Kutta-Verfahrens* und ist für die häufig auftretenden Differentialgleichungen gut geeignet. Die '4' und '5' stehen für Approximationen 4-ter Ordnung an die Funktion und Fehlerterme 5-ter Ordnung. (So gibt es zum Beispiel auch einen Löser `ode23`.)

Der Aufruf des Befehls `ode45` erfolgt ähnlich wie der unserer oben erstellten Funktion `euler`. Tatsächlich könnte man `euler` auch als einen Solver bezeichnen. Nur dass `ode45` durch ein bessere Annäherung an die Funktion und durch angepasste (nicht gleichmässige) Schrittweiten eine weitaus höhere Genauigkeit erreichen kann.

Die rechte Seite der Differentialgleichung $y' = f(x, y)$ muss zunächst als MATLAB Funktion definiert werden, als *anonyme Funktion* (mit Klammeraffe `@`) oder in einer m-Datei. Diese Funktion muss als erstes angegeben werden, als zweites der Zeitraum oder Bereich, in dem `y(x)` berechnet werden soll – als Intervall! Dahinter der Startwert im Anfangspunkt des Intervalls. Diese drei Angaben sind immer notwendig.

Lösen wir als Beispiel das Anfangswertproblem $y' = y \cdot sin(x)$ im Bereich von 0 bis 3π mit der Anfangsbedingung $y(0) = 1$. f ist also die Funktion $f(x, y) = y \cdot sin(x)$ und daher

```
>> f = @(x, y) sin(x) .* y;
>> ode45(f, [0, 3*pi], 1.0)
```

MATLAB erzeugt einen Plot, der den Verlauf der Funktion `y` in dem geforderten Bereich anzeigt. Wir sehen, dass die Lösung offenbar periodisch ist, und wie der `ode45` Solver die Punkte, in denen die Funktion berechnet wird, ungleichmässig im Intervall verteilt.

Bei dieser Art des Aufrufs werden keine numerischen Ergebnisse zurück geliefert. Um tatsächlich die berechneten x- und y-Werte zu erhalten, müssen diese im Aufruf angefordert werden.

```
>> [x, y] = ode45(f, [0, 3*pi], 1.0);
```

Jetzt erscheint kein Plot der Lösung, stattdessen stehen die berechneten Punkte der Lösung in den Vektoren `x` und `y` zur Verfügung, etwa um einen eigenen Plot mit anderen Optionen zu erzeugen, getrichelt und in Rot.

```
>> plot(x, y, 'r--')
>> grid on
```

In diesem Graphen könnte man analytischen Verlauf der Lösung ($y(x) = e^{1-\cos(x)}$) dazu plotten, um die Genauigkeit zu vergleichen.

Sehr oft ist der Anwender am Funktionswert am rechten Rand des Bereiches, im Punkt $x = b$, interessiert. Dieser Wert steht im letzten y-Wert, den man mit `y(end)` erhält.

```
>> y(end)
ans =
    7.3930
```

Wichtig ist, dass `f` immer von zwei Variablen abhängen muss, als erstes der unabhängigen, als zweites der abhängigen Variablen, auch wenn die Funktion der Formulierung nach nur von einer Variablen abzuhängen scheint.

Beispiel: Ein Gefäss mit Wasser von 95 Grad kühle in einem Raum mit Raumtemperatur 20 Grad mit einem Abkühlungskoeffizienten -0.05 über einen Zeitraum von 20 Minuten ab. Wie gross ist seine Temperatur am Ende?

Nach dem Newtonschen Abkühlungsgesetz gilt $T' = -0.075 \cdot (T - 20)$. Mit der Temperatur T und der Zeit t als unabhängiger Variable ist daher (die Variablen können ja beliebig benannt werden):

```
>> dT = @(t, T) -0.075 * (T - 20);
>> [t, T] = ode45(f, [0, 20], 95)
>> plot(t, T, 'r', 'LineWidth', 2)
>> grid on
```

```
>> T(end)
ans =
   36.7348
```

Manchmal ist das Ergebnis des ODE Solvers nicht korrekt oder exakt genug. Aus diesem Grund gibt es die Möglichkeit die Optionen des Solvers zu verändern.

Dazu wird hinter die Anfangswerte eine Variable `opts` eingefügt, mit dem Befehl `odeset` können die Parameter verändert werden. In der Hilfe zu `odeset` ist aufgelistet, welche Möglichkeiten es gibt. Wir erhöhen die absolute und relative Tolerenz:

```
>> opts = odeset('RelTol', 1e-7, 'AbsTol', 1e-7);
```

Im obigen Beispiel $y' = y \cdot sin(x)$ war die Genauigkeit nicht ausreichend. Mit der nun gesetzten Toleranz sieht der Funktionsgraph schon glatter aus.

```
>> f = @(x, y) sin(x) .* y;
>> [x, y] = ode45(f, [0, 3*pi], 1.0, opts);
>> plot(x, y, 'ro-')
>> y(end)
ans =
    7.3891
```

Auch der Enpunkt ist nun deutlich genauer berechnet.

4.4 Anwendungsbeispiele

Aufgabe: Die Ausbreitung einer ansteckenden Krankheit kann modelliert werden durch eine Differenzialgleichung $P' = kP(Pges - P)$, $Pges$ die Gesamtzahl der betroffenen Personen, k ein Ausbreitungsfaktor und $P = P(t)$ die Zahl der Kranken zum Zeitpunkt t.

Bestimme für $Pges = 250\,000$ von Personen einer Gemeinde, $k = 0.00003$ und 250 die Anzahl der Erkrankten zum Zeitpunkt 0. Zeichne den Verlauf der Zahl der Erkrankten über einen Zeitraum von 60 Tagen.

Aufgabe: Für die *Konzentration* eines Stoffes in einem chemischen Reaktors gelte

$$\frac{\mathrm{d}C}{\mathrm{d}t} = \frac{-k_1\, C}{1 + k_2 C}$$

Bestimme die Konzentration über einem geeigneten Zeitraum mit folgenden Reaktionskonstanten, wobei $C(0)$ die Konzentration dieses Stoffes an Beginn der Reaktion ist: $k_1 = 1.0$, $k_2 = 0.3$, $C(0) = 0.8$

Aufgabe: Nach dem *Torricellischen Gesetz* folgt die Höhe y des Wasserstandes in einem auslaufenden Wassertank def folgenden Gleichung

$$y' = -k\,\sqrt{y}\,A(y)$$

Dabei sei $A(y)$ Querschnittsfläche des Tanks in Höhe y, a die Fläche des Auslaufs, $k = a \cdot \sqrt{2g}$, $g = 9.81$ m/s^2 die Erdbeschleunigung.

Schätze die Zeit, die eine Badewanne braucht um leer zu laufen. Wir nehmen $A(y)$ konstant zu 45×125 cm^2 und $a = 6\pi \times 0.4^2$. Achten Sie auf die physikalischen Einheiten.

Aufgabe: Beim freien Fall ohne Luftwiderstand ist die Beschleunigung g gerade die Änderung der Geschwindigkeit v, also $v' = -g$ (nach unten gerichtet). Der Luftwiderstand bremst den

Gegenstand, nach einem oft verwendeten Modell wirkt dieser Widerstand proportional zu $|v|^p$, $1 \leq p \leq 2$: $p \approx 1.1$ bei laminarer Strömung, $p \approx 2$ bei turbulenter Strömung.

Benutze das folgende Modell für die Bestimmung der Geschwindigkeit eines Menschen, zum Beispiel bevor der Fallschirm sich öffnet.

$$v' = -g + \frac{k}{m}|v|^{1.1}$$

Dabei sei $k \approx 128$ eine Konstante, $m = 80$ kg das Gewicht. Wie gross ist die konstante Grenzgeschwindigkeit, wie schnell wird sie erreicht?

4.5 Systeme von Differenzialgleichungen

Mit den ODE Solvern können auch Systeme aus mehreren Differentialgleichungen gelöst werden. Ein solches System aus zwei gekoppelten Gleichungen der Form

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2) \\ y_2' &= f_2(x, y_1, y_2) \end{aligned}$$

wird in MATLAB repräsentiert durch eine Funktion `f`, welche die berechneten Werte für f_1 und f_2 als *Spaltenvektor* zurück gibt. Die Anfangsbedingungen $y_1(a)$ und $y_2(a)$ werden ebenfalls als Spaltenvektoren formuliert.

Beispiel: Lotka-Volterra Gleichungen

Beim Räuber-Beute-Modell werden zwei voneinander abhängige Populationen betrachtet, die Räuber r(t) und die Beute b(t), deren Koexistenz durch das folgende System von Differenzialgleichungen beschrieben wird.

$$\begin{aligned} db(t) &= a_1\, b(t) - a_2\, b(t)\, r(t) \\ dr(t) &= a_3\, r(t) - a_4\, b(t)\, r(t) \end{aligned}$$

Der Ausdruck $a_1\ b(t)$ bzw. $a_3\, r(t)$ beschreibt die Vermehrung der Beute bzw. wie die Zahl der Räuber sich vermindert, wenn keine Beute vorhanden ist. Der zweite Term beschreibt die Wahrscheinlichkeit, dass Räuber und Beute aufeinandertreffen und die Zahl der Beutetiere verringert bzw. das Überleben der Räuber möglich macht.

Als Beispiel setzen wir $a_1 = 0.1$, $a_2 = 0.01$, $a_3 = -0.05$, und $a_4 = 0.001$. Das System sieht also folgendermassen aus:

$$\begin{aligned} db(t) &= 0.1\, b(t) - 0.01\, b(t)\, r(t) \\ dr(t) &= -0.05\, r(t) + 0.001\, b(t)\, r(t) \end{aligned}$$

Damit wir das System mit `ode45` lösen können, legen wir `b` und `r` in den Vektor `y`, der dann so aussieht: `y = [b; r]`. Die rechte Seite des Systems wird in folgender Funktion abgespeichert und berechnet einen Spaltenvektor mit zwei Elementen.

```
>> f = @(t,y) [0.1  * y(1) - 0.01  * y(2) * y(1)
              -0.05 * y(2) + 0.001 * y(1) * y(2)];
>> y0 = [50; 15];
```

(Interpretation: `y(1)` ist `b`, `y(2)` ist `r`, denn `y = [b; r]`.)

Als Anfangswerte legen wir `y0 = [50; 15]` fest, das bedeutet, zu Beginn gibt es 50 'Einheiten' Beute und 15 'Einheiten' Räuber. Angenommen, die Zeiteinheit sind Monate, dann sollen die interagierenden Populationen 2 Jahre beobachtet werden.

```
>> ode45(f, [0, 240], y0)
```

Als Ergebnis sehen wir zwei Kurven, weil `y` aus zwei Komponenten besteht. Die erste, blaue Kurve stellt offenbar den Verlauf der Beute dar, die zweite, rote den der Räuber. Das kann man auch feststellen, indem man sich die Werte für `x` (als Zeit `t`) und `y` ausgeben lässt.

```
>> [t, y] = ode45(f, [0, 240], y0)
>> size(y)
ans =
    73     2
>> y(end, :)
ans =
   63.8465    6.5562
```

`t` ist ein Spaltenvektor und `y` eine Matrix mit 73 Zeilen und 2 Spalten. Zu jedem Zeitpunkt gehört eine Zeile mit den berechneten Werten für Beute und Räuber. Man kann sich wieder einen eigenen Plot anzeigen lassen.

```
>> plot(t, y(:, 1))         % Verlauf der Beute (blau)
>> hold on
>> plot(t, y(:, 2), 'r')    % Verlauf der Raeuber (rot)
>> legend('Population Beute', 'Population Raeuber')
>> grid on, hold off
```

Man kann natürlich auch die Anzahl der Räuber über die der Beute plotten. Das wollen wir in einem zweiten Fenster tun.

```
>> figure(2)
>> plot(y(:,1), y(:,2))
```

Bemerkung: Wir sehen, dass es sich um eine zyklischen Verlauf handelt. Allerdings ist die Lösung dieses Systems von Differenzialgleichungen nicht analytisch, d.h. lässt sich nicht durch uns bekannte elementare Funktionen ausdrücken.

Aufgabe: Spielen Sie mit Anfangswerten und Koeffizienten, um andere Verläufe zu visualisieren. Schaffen Sie es, diese Parameter so einzustellen, dass die Räuber die Beute komplett auffressen?

Aufgabe: In einem Reaktor befinden sich drei chemische Substanzen mit den Anfangskonzentrationen 0.6, 0.2 und 0.2. Der Chemie-Ingenieur sagt Ihnen, dass die Konzentrationen der Stoffe in Gegenwart der anderen Stoffe sich nach folgenden Gesetzen verändern:

$$\begin{aligned} c_1' &= -k_1 c_1 + k_3 c_3 \\ c_2' &= k_1 c_1 - k_2 c_2 \\ c_3' &= k_2 y_2 - k_3 c_3 \end{aligned}$$

Die Konstanten gibt er an zu $k_1 = 0.3$, $k_2 = 0.2$, und $k_3 = 0.5$. Bestimmen Sie die Konzentrationen der drei Substanzen nach eine Reaktionszeit von 10 Sekunden.

4.6 Differenzialgleichungen zweiter Ordnung

Differenzialgleichungen zweiter Ordnung haben die Form

$$y'' = f(x, y, y')$$

Es kommt also eine Ableitung y'' 2. Ordnung vor, welche selbst wieder von x und y, aber auch von der ersten Ableitung y' abhängen kann.

In mechanischen und physikalischen Anwendungen treten solche Gleichungen zweiter Ordnung sehr häufig auf. Das liegt an den Newtonschen Gesetzen, die eine Beziehung zwischen der Kraft und der Beschleunigung – also der zweiten Ableitung des Ortsvektors – herstellen.

Die in MATLAB implementierten ODE Solver lösen nur Differenzialgleichungen erster Ordnung. Daher muss eine DGL zweiter Ordnung umgewandelt werden in ein System von Gleichungen erster Ordnung. Das geschieht durch Einführung zweier Funktionen $y_1 = y$ und $y_2 = y'$, die erste Ableitung wird also als eigenständige Funktion behandelt. Die Gleichung $y'' = f(x, y, y')$ geht dabei über in ein System von Gleichungen folgender Art:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= f(x, y_1, y_2) \end{aligned}$$

Beispiel: Mathematisches Pendel

Das mathematische Pendel ist ein idealisiertes physikalisches Pendel, wo eine punktförmig gedachte Masse an einem masselosen Faden hängt bzw. unter dem Einfluss der Schwerkraft schwingt.

Löst man das Kräfteparallelogramm am Pendel auf, ergibt sich als Bewegungsgleichung für die Auslenkung z des Pendel:

$$z''(t) = -\frac{g}{L} sin(z(t))$$

$g = 9.81$ m/s^2 ist die Erdbeschleunigung, L die Länge des Pendels.

Diese Gleichung lässt sich analytisch resp. symbolisch nicht auflösen, daher wird stattdessen oft die Vereinfachung $sin(\phi) \approx \phi$ für kleine Auslenkungen benutzt, um das Pendelproblem zu lösen.

Numerisch dagegen sollte das kein Problem sein. Benutzen wir $z_1 = z$ für die Auslenkung und $z_2 = z'$ für die Ableitung der Auslenkung, also die Winkelgeschwindigkeit, dann gilt:

$$\begin{aligned} z_1' &= z_2 \\ z_2' &= -\tfrac{g}{L} sin(z_1) \end{aligned}$$

und die entsprechende MATLAB Funktion lautet für ein Pendel von 1 m Länge:

```
>> g = 9.81; L = 1;
>> dz = @(t, z) [z(2); -g/L * sin(z(1))]
```

Wir starten das Pendel zum Zeitpunkt 0 mit maximaler Auslenkung von 90 Grad und lassen einfach los, d.h. Anfangsgeschwindigkeit 0 und damit Startvektor `[pi/2, 0]`, über einen Zeitraum von 10 sec:

```
>> ode45(dz, [0, 10], [pi/2, 0])
```

Die Graphik zeigt uns, dass das Pendel in dieser Zeit etwas mehr als vier Mal hin- und herschwingt.

Aufgabe: Vergleiche bei gleichen Anfangsbedingungen mit einem Pendel der Länge $L = 0.5$ m.

Aufgabe: Führe die gleiche Rechnung aus für das angenäherte Pendel mit $sin(z) \approx z$, also der Differenzialgleichung $z'' = -\frac{g}{L} z$. Zeichne den Verlauf der Auslenkung beider Pendel in einen Plot. Welches Pendel, das exakte oder das angenäherte Pendel, schwingt schneller?

Beispiel: Gedämpftes Federpendel

Ein Federpendel befolgt 'per definitionem' eine Differenzialgleichung $z'' = -\frac{k}{m} z$, z die Auslenkung. Die Lösung wäre offensichtlich eine Schwingung in Form einer Sinusfunktion.

Nehmen wir an, das Federpendel wird in seiner Bewegung gedämpft, zum Beispiel durch aerodynamischen oder mechanischen Widerstand. Die Dämpfung ist i.A. proportional aber der Geschwindigkeit entgegen gerichtet, um einen Betrag $2\delta\, z'$.

Die Differentialgleichung, die wir lösen müssen, lautet dann:

$$z'' + 2\delta\, z' + \frac{k}{m}\, z = 0$$

Mit $k = 5$ N/m, $m = 0.1$ kg, $z(0) = 0.1$ m und $z'(0) = 0$ m/s und einem Dämpfungsfaktor $\delta = 1$ lauten unsere Gleichungen dann

```
>> k = 5; m = 0.1; delta = 1;
>> dz = [z(2); -2*delta*z(2) + k/m*z(1)]
```

und die Lösungskurve wird erzeugt mit

```
>> ode45(dz, [0, 60], [0.1, 0])
```

Der Grenzfall liegt bei $w_0 = \sqrt{k/m}$, für $\delta >\approx \omega_0$ wird keine Schwingung mehr einsetzen.

Aufgabe: Variiere δ, um verschiedene Lösungen anzeigen zu lassen.

4.7 Symbolische Lösung von Differenzialgleichungen

In MATLAB kann man eine gewöhnliche Differenzialgleichung symbolisch (d.h. analytisch) mit der Funktion `dsolve` lösen, falls sie eine analytische Lösung hat. Als Beispiel lösen wir die Differenzialgleichung $y' = x + y$.

Dem Befehl `dsolve` muss man die Differentialgleichung als Zeichenkette, eingeschlossen in Hochkomma `'`, übergeben. Die Ableitung y' wird dabei mit grossem 'D' als `Dy` geschrieben. Dahinter wird noch die unabhängige Variable `x` ebenfalls als Zeichenkette angegeben.

```
>> y = dsolve('Dy = x+y', 'x')
y =
C1*exp(x) - x - 1
```

Die Lösung wird symbolisch ausgegeben. `C1` steht für eine freie Konstante, die je nach Anfangsbedingung angepasst werden kann. Natürlich kann an `dsolve`auch eine Anfangsbedingungen übergeben werden, zusätzlich in der ersten Zeichenkette und nur getrennt durch ein Komma.

```
>> y = dsolve('Dy = x+y, y(0) = 0', 'x')
y =
exp(x) - x - 1
```

So erhält man eine eindeutige Lösung. Um die Lösung von 0 bis 2 noch in einem Plot darzustellen, kann man zum Beispiel `ezplot` verwenden.

```
>> ezplot(y, [0, 2])
>> grid on
```

Um einen numerischen Wert an einer bestimmten Stelle, etwa $x = 0.5$, zu erfahren, substituiert man 0.5 für `x` in der Lösung `y` und verwendet `double`, um daraus eine reelle Zahl zu erzeugen.

```
>> double(subs(y, 'x', 0.5))
ans =
    0.1487
```

Statt einer einzelnen Zahl könnte der Ausdruck auch auf einem ganzen Vektor ausgerechnet werden.

Aufgabe:

`dsolve` kann im Gegensatz zu `ode45` auch Differentialgleichungen zweiter Ordnung lösen – vorausgesetzt es gibt eine analytische Lösung. Die Formulierung dazu kann man unter `doc` nachlesen, hier nur einige Beispiele.

```
>> dsolve('D2y = - y', 'x')
ans =
    C1*cos(x) + C2*sin(x)

dsolve('Db = 0.1*b - 0.01*b*r, Dr = -0.05*r + 0.001*b*r', 't')
Warning: Explicit solution could not be found.
> In dsolve (line 201)
ans =
[ empty sym ]
```

Die Lotka-Volterra Gleichung kann nicht analytisch gelöst werden.

5 Exkurs: Symbolisches Rechnen mit MATLAB

MATLAB hauptsächlich für die Verarbeitung von numerischen Daten gedacht, es kann in MATLAB auch symbolisch gerechnet werden, zum Beispiel exaktes Differenzieren oder Integrieren. Voraussetzung ist, dass die *Symbolic Toolbox* im Suchpfad von MATLAB vorhanden ist.

Normalerweise sucht MATLAB beim Verarbeiten einer Variablen, etwa `x`, nach einem gespeicherten numerischen Wert für diese Variable. Wir müssen MATLAB explizit sagen, dass `x` eine symbolische Variable ist. Das geschieht mit dem Befehl `syms x` bzw. `syms x y ...`.

```
>> syms x
>> y = cos(x)
```

Im Workspace Window erscheinen `x` und `y` jetzt als '1x1 sym', also als symbolische Variablen.

Dabei ist es ein Unterschied, ob wir `y = cos(x)` oder `z(x) = cos(x)` schreiben. Das erste erzeugt y als einen symbolischen Ausdruck, das zweite erzeugt z als symbolische Funktion. Beide erscheinen nun auch im Workspace, `z` als '1x1 symfun'.

Man kann für x wieder einen numerischen Wert einsetzen und sich den Kosinus ausgeben lassen, das geschieht mit `subs(y, x, pi)` bzw. `z(pi)`, durch "Substitution" erhält man `y` und `z` an der Stelle `x = pi`.

```
>> z(pi/2)
ans =
    0
```

Wenn ein Ausdruck symbolisch nicht bekannt ist, z. B. `z(1.0)`, dann wird er numerisch nur ausgewertet, wenn darauf der Befehl `double` angewendet wird.

```
>> z1 = z(1.0)
z1 =
    cos(1)
>> double(z1)
ans =
    0.5403
```

Wie bereits erwähnt, kann man das symbolische Rechnen für Differenzieren und Integrieren nutzen. Zum Differenzieren gibt es den Befehl `diff`,

```
>> diff(cos(x), x)      % oder diff(y, x)
ans =
-sin(x)
```

wobei das zweite Argument die Variable nennt, nach der differenziert werden soll. Man kann auch noch die Höhe der Ableitung als weiteres Argument angeben, die dritte Ableitung nach `x` also bestimmt mit

```
>> diff(cos(x), x, 3)
ans =
sin(x)
```

Analog dazu wird zum Integrieren von `log(x)` der Befehl `int` benutzt, unter Angabe der Integrationsvariablen.

```
>> int(log(x), x)
ans =
x*(log(x) - 1)
```

Die Symbolic Toolbox liefert also nur die Stammfunktion, die freie Integrationskonstante müssen wir uns dazudenken. Ein bestimmtes Integral erhält man durch die zusätzliche Angabe der Integrationsgrenzen.

```
>> int(sin(x)^2, x, 0, pi)
ans =
pi/2
```

(Hinweis: Siehe die obige Berechnung des Schwerpunktes: Die unbekannte Zahl 0.3927 ist offenbar $\pi/8$.)

Auch Stammfunktionen komplexerer Integrale lassen sich mit `int` herleiten, zum Beispiel $\int \frac{log(x)}{(2x+5)^3}\,\mathrm{d}x$. Und mit `pretty` kann eine etwas besser lesbare Anzeige des Ergebnisses erzielt werden.

```
>> int(log(x) / (2*x + 5)^3)
ans =
log(x)/100 - log(x + 5/2)/100 + (x/10 - log(x)/4 + 1/4)/(2*x + 5)^2
>> pretty(ans)
            /     5 \    x   log(x)   1
         log| x + - |   -- - ------ + -
log(x)      \     2 /   10      4     4
------ - ------------ + ---------------
  100         100                    2
                            (2 x + 5)
```

Auch unendliche Integrationsbereiche oder Funktionen mit Polstellen am Rand sind möglich. Soll `1/x^2` von 1 bis Unendlich integriert werden, tippt man: `int(1/x^2, 1, Inf)` mit der Antwort `ans = 1`.

Aufgabe: Wie gross ist $\int_{-\infty}^{\infty} e^{-x^2}\,\mathrm{d}x$?

Ebenso einfach ist es, sich die *Taylorentwicklung* (das Taylorpolynom) einer (hinreichend oft differenzierbaren) Funktion f in einem Punkt a anzeigen zu lassen. Als Beispiel das Taylorpolynom der Funktion $\log(1+x)$ im Punkt $x = 0$.

```
>> taylor(log(1+x), x, 0)
ans =
x^5/5 - x^4/4 + x^3/3 - x^2/2 + x
```

Bis zur fünften Ordnung ist der 'Default'. Um auch höhere Glieder des Taylorpolynoms zu erhalten, müsste man eingeben: `taylor(log(1+x), x, 0, 'Order', 8)`

```
>> T = taylor(log(1+x), x, 0, 'Order', 8);
>> pretty(T)
 7    6    5    4    3    2
x    x    x    x    x    x
-- - -- + -- - -- + -- - -- + x
 7    6    5    4    3    2
```

Aufgabe: Plotten Sie die Funktion $\log(1+x)$ in einer Umgebung von $x = 0$ und zusätzlich in rot das Taylorpolynom.

Tatsächlich können mit `ezplot` auch symbolische Funktionen und Ausdrücke geplottet werden.

```
>> g = 1/(5 + 4*cos(x));
>> ezplot(T, [0, 6*pi])
```

`coeffs(T)` liefert die Koeffizienten des Polynoms als Vektor, allerdings in umgekehrter Reihenfolge, als es von MATLAB erwartet wird. (Man kann die Reihenfolge der Elemente eines Vektors 'umdrehen' mit 'x(end:-1:1).)

Symbolische Ausdrücke können vereinfacht werden mit dem `simplify` Befehl.

```
>> simplify((x^2-5*x+6)/(x-2))
ans =
x - 3
```

Der `factor` Befehl bestimmt die Faktoren eines Polynoms.

```
>> factor(x^2-5*x+6)
ans =
[ x - 2, x - 3]
```

Umgekehrt kann ein Produkt auch ausmultipliziert werden, mit `expand`.

```
>> expand((x+2)*(x-7))
ans =
x^2 - 5*x - 14
```

Symbolische Bestimmung von Nullestellen ist möglich mit dem `solve` Befehl.

```
>> solve(x^2 - 5*x - 14, x)
ans =
    -2
    7
```

Die symbolische Toolbox kennt viel mehr Funktion als ein Anwender, daher kann es auch Überraschungen beim Lösen von Gleichungen geben. Ein Beispiel von früher: $x * e^x = 1$.

```
>> omega = solve(x * exp(x) - 1, x)
omega =
lambertw(0, 1)
```

`lambertw` ist die Umkehrfunktion von $x \to x\,e^x$, daher ist das Ergebnis richtig, obwohl vielleicht nicht gleich verständlich. Den genauen Wert erhält man wieder mit `double`.

```
>> format long
>> double(omega)
ans =
   0.567143290409784
```

Intern rechnet die symbolische Toolbox mit sehr viel höherer Genauigkeit. Um sich mehr Dezimalstellen eines Wertes anzeigen zu lassen, benutze den `vpa` Befehl (*variable-precision arithmetic*). Das zweite Argument gibt die Anzahl der gewünschten Ziffern an:

```
>> vpa(omega, 32)
ans =
0.56714329040978387299996866221036

>> vpa(pi, 64)
ans =
3.14159265358979323846264338327950288419716939937510582097494459 2
```

Auch Grenzwerte von Funktionen oder Folgen können exakt bestimmt werden, etwa $\lim_{x\to 0} \frac{sin(x)}{x}$

```
>> limit(sin(x)/x, x, 0)
```

```
ans =
1
```

Für Folgen wie die der n-ten Wurzel von n: $\lim_{n\to\infty} n^{1/n}$ muss wieder zuerst n als symbolische Variable deklariert werden.

```
>> syms n
>>limit(n^(1/n), n, Inf)
ans =
1
```

Der exakte, symbolische Wert einer unendlichen Reihe wie $\sum_{n=1}^{\infty} \frac{1}{n^2}$ wird mit dem Befehl `symsum` berechnet. Sowohl Anfangs- wie Endwert für n müssen angegeben werden.

```
>> a = symsum(1/n^2, n, 1, Inf)
a =
pi^2/6
```

Einen numerischen Wert für `a` erhält man mit `vpa`; ohne Angabe der Anzahl von Ziffern werden 50 Ziffern angezeigt.

```
>> vpa(a)
ans =
1.6449340668482264364724151666460251892189499012068
```

Beachten Sie, dass `vpa(a)` weiterhin vom Typ 'sym' ist. Um diesen Wert in weiteren rein numerischen Berechnungen verwenden zu können, muss er vorher mit `double` in eine Gleitkommazahl umgewandelt werden. Diese Zahl hat dann aber nur noch unsere übliche Genauigkeit von 15–16 Stellen.

Hinweis: Die wichtige Anwendung der symbolischen Toolbox auf Probleme mit Differenzialgleichungen werden wir später noch kennenlernen.

Die symbolische Toolbox basiert auf **MuPAD**, einem Computer-Algebra System (engl. *Computer-Algebra-System*, CAS), das ursprünglich an der Universität Paderborn entwickelt und dann von MathWorks aufgekauft wurde.

Die ursprüngliche graphische Oberfläche von MuPAD kann man noch sehen, wenn man in MATLAB den Befehl `mupad` eintippt. Im Menu *Command Bar* auf der rechten Seite können vorgefertigte Befehle aufgerufen und dann ausgefüllt werden. Auf der Eingabefläche werden die Ergebnisse in mathematischer Notation angezeigt.

Zu MuPAD gibt es ganze Bücher, zum Beispiel das "MuPAD Tutorial", Springer-Verlag, 2004.

Aufgabe: Generiere in MuPAD die bekannte symbolische Lösung der quadratischen Gleichung $x^2 + p\,x + q = 0$.