

Vorlesungsskript

Computergestützte Mathematik

Kurt Keilhofer und Martin Kerscher

14. Januar 2020

Inhaltsverzeichnis

1. MATLAB	4
1.1. Grundlagen, Rechnen mit skalaren Größen und Matrizen	4
1.1.1. Starten, Bedienung, Beenden und Hilfe	4
1.1.2. Einfaches Rechnen mit skalaren Größen	4
1.1.3. Rechnen mit Vektoren und Matrizen	6
1.2. Programmieren mit MATLAB	8
1.2.1. Skriptdateien und Funktionsdateien	8
1.2.2. Schleifen und bedingte Anweisungen	10
1.2.3. Anonyme Funktionen und Funktionen als Eingabeparameter	11
1.3. Grafik	12
1.3.1. Zweidimensionale Grafiken	12
1.3.2. Dreidimensionale Grafiken	13
1.4. Numerische Lineare Algebra	15
1.4.1. Lineare Gleichungssysteme und Matrix-Faktorisierungen	15
1.4.2. Norm und Kondition	16
1.4.3. Eigenwertaufgaben	16
1.5. Warum ist MATLAB schnell?	16
1.5.1. LAPACK und BLAS	16
1.5.2. Matrixmultiplikation	17
1.5.3. Dünnbesetzte Matrizen	19
2. Maple	20
2.1. Grundlagen, Rechnen mit Zahlen und Ausdrücken	20
2.1.1. Starten, Bedienung, Beenden und Hilfe	20
2.1.2. Maple als Taschenrechner	21
2.1.3. Rechnen mit Ausdrücken, Lösen von Gleichungen	22
2.1.4. Summen und Produkte	23
2.1.5. Komplexe Zahlen	24
2.2. Datenstrukturen, Funktionen und Grafik	24
2.2.1. Sequenzen, Listen und Mengen	24
2.2.2. Funktionen	25
2.2.3. Funktionsgraphen	26
2.2.4. Weitere Grafikbefehle	27
2.3. Analysis	28
2.3.1. Grenzwerte und Reihen	28
2.3.2. Differentiation	29

Inhaltsverzeichnis

2.3.3.	Taylor-Entwicklung	29
2.3.4.	Integration	30
2.3.5.	Vektoranalysis	31
2.4.	Lineare Algebra	32
2.4.1.	Rechnen mit Vektoren und Matrizen	32
2.4.2.	Lineare Gleichungssysteme, Basen, Kern und Bild	33
2.4.3.	Determinanten, Eigenwerte und Normalformen	34
2.5.	Ausblick: Weitere Teilgebiete der Mathematik	35
2.5.1.	Elementare Zahlentheorie	36
2.5.2.	Algebra	37
2.5.3.	Geometrie	38
2.5.4.	Differentialgleichungen	39
2.5.5.	Wahrscheinlichkeitstheorie	40
3.	R	41
3.1.	Grundlagen	41
3.1.1.	Starten, Bedienung, Beenden und Hilfe	41
3.1.2.	Rechnen	41
3.1.3.	Vektoren und Matrizen	42
3.1.4.	Zufallsgrößen, Verteilungen und einfache Plots	44
3.2.	Programmieren und Grafik	45
3.2.1.	Funktionen, Verzweigungen und Schleifen	45
3.2.2.	Grafik	47
3.3.	Deskriptive Statistik	48
3.3.1.	Listen und Data Frames	48
3.3.2.	Mittelwert etc	49
3.3.3.	Histogramme etc.	49
3.4.	Tests und Modelle	50
3.4.1.	Vergleich zweier Experimente	50
3.4.2.	Modelle und Regression	52
3.5.	Sonstiges	53
3.5.1.	Packages	53
A.	Syntaxvergleich: MATLAB – Maple – R	54

1. MATLAB

1.1. Grundlagen, Rechnen mit skalaren Größen und Matrizen

MATLAB ist eine Programmierumgebung für technische und wissenschaftliche Anwendungen. Der Name ist eine Abkürzung für *MATrix LABoratory*. Die Stärken von MATLAB liegen im numerischen Rechnen mit Matrizen, der Simulation sowie in der Visualisierung.

1.1.1. Starten, Bedienung, Beenden und Hilfe

MATLAB wird durch Eingabe von `matlab` an der Linux-Konsole gestartet und über das Menü `File` → `Exit` beendet. MATLAB-Programmdateien haben die Endung `.m`.

Zur Eingabe: Groß/Kleinschreibung ist signifikant. Interaktive Eingaben an der Eingabeaufforderung `>>` werden mit `Enter` abgeschlossen und ausgeführt. Wird eine Zeile mit einem Strichpunkt abgeschlossen, so erfolgt keine Ausgabe. Die Zeile wird aber ganz normal ausgeführt.

Es empfiehlt sich zu Beginn einer neuen Aufgabe

```
>> clear all
```

einzugeben, um nicht benötigte Variablen zu löschen und MATLAB in einen fest definierten Startzustand zu versetzen.

Das umfangreiche Hilfesystem lässt sich über das Menü `Help` starten. Wenn man Hilfe zu einem Befehl benötigt, dessen Namen man schon kennt, ist der einfachste Weg, an der Eingabeaufforderung `help Befehlsname` einzugeben. Dadurch bekommt man eine Kurzbeschreibung des Befehls angezeigt. Eine ausführlichere Hilfeseite mit Beispielen und Querverweisen liefert `doc Befehlsname`

1.1.2. Einfaches Rechnen mit skalaren Größen

Die Rechenoperatoren sind `+`, `-`, `*`, `/` für die vier Grundrechenarten und `^` für das Potenzieren. MATLAB hält sich an die üblichen Punkt-vor-Strich- und Klammerregeln. Beachten Sie, dass das Dezimalzeichen, wie in allen Programmiersprachen, der Punkt ist.

1. MATLAB

```
>> 1 + 2
>> 3.4*(5.2 - 0.1)
>> 2^10
>> 2^(-10)
```

Der Backslash-Operator `\` berechnet bei `a\b` den Ausdruck `inv(a)*b`, wobei `inv(a)` das Inverse zu `a` ist. Für skalare Größen ist das nur eine Spielerei, es ist aber wichtig bei Matrizen.

```
>> 2/7
>> 7/2
>> 2\7
>> 7\2
```

MATLAB rechnet numerisch mit Gleitpunktarithmetik. Die Konstante `eps` liefert die relative Maschinengenauigkeit¹.

```
>> eps
>> (1 + eps) - 1
>> (1 + eps/2) - 1
```

Mit dem Zuweisungsoperator `=` können Variablen Werte zugewiesen werden; mit `clear` können die Zuweisungen wieder gelöscht werden. MATLAB verwendet `pi` für die Kreiszahl π . MATLAB kennt die Funktionen `abs` (Betragsfunktion), `sqrt` (Quadratwurzel), `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan` und noch viele andere.

```
>> pi
>> cos(pi)
>> atan(1)
>> a = sqrt(2)
>> a^2
>> abs(-0.1)
>> b = log(4711)
>> exp(b)
```

MATLAB gibt standardmäßig nur fünf Dezimalstellen aus (rechnet aber natürlich intern genauer). Das kann durch `format long` geändert werden (rückgängig mit `format short`).

```
>> c = 1.2345678
>> c
>> format long
>> c
>> format short
>> c
```

¹Hierbei handelt es sich um die kleinste positive Maschinenzahl, die bei Addition zu 1 einen von 1 verschiedenen Wert ergibt. Sie entspricht also dem relativen Abstand zweier Maschinenzahlen.

1. MATLAB

Die imaginäre Einheit wird mit `i` bezeichnet. Absolutbetrag, Realteil und Imaginärteil komplexer Zahlen bekommt man mit `abs`, `real`, `imag`.

```
>> i^2
>> z = (3 + 4*i) / (1 - 2*i)
>> abs(z)
>> real(z)
>> imag(z)
```

1.1.3. Rechnen mit Vektoren und Matrizen

Der grundlegende Datentyp in MATLAB ist die Matrix. Vektoren sind einspaltige oder einzeilige Matrizen. Man kann Matrizen explizit in eckigen Klammern angeben; Spalten werden durch Kommas oder Leerzeichen, Zeilen durch Strichpunkte getrennt. Auf diese Weise kann man Matrizen auch horizontal oder vertikal aneinander setzen.

```
>> clear all
>> x = [1, 2, 3]
>> y = [1.1; 2.2; 3.3]
>> A = [1, 4, 2; 3, 7, -1; 1, 1, pi]
>> Ay = [A, y]
```

Spezielle Matrizen liefern die Funktionen `zeros` (Nullmatrix), `eye` (Einheitsmatrix) und `ones` (Matrix mit lauter Einsen). Der Befehl `rand` erzeugt zufällige Matrizen mit auf $]0, 1[$ gleichverteilten Einträgen. Matrizen mit standardnormalverteilten Einträgen bekommt man mit `randn`.

```
>> zeros(2,3)
>> eye(3)
>> ones(3,2)
>> B = rand(3,3)
>> randn(1,10)
```

Mit `von:bis` bzw. `von:schritt:bis` können arithmetische Folgen als Zeilenvektoren erzeugt werden. Das ist vor allem für Indexbereiche wichtig.

```
>> 1:10
>> 0:0.1:1
>> 5:-2:-4
```

Einzelne Elemente einer Matrix `A` werden mit `A(Zeile,Spalte)` angesprochen. Dabei können auch Indexbereiche angegeben werden. Ein einzelner Doppelpunkt steht dabei für die ganze Zeile bzw. Spalte. Solche Teilmatrizen dürfen auch links vom Zuweisungsoperator stehen.

1. MATLAB

```
>> A(1,1)
>> A(3,2)
>> A(1,:)
>> A(:,3)
>> A(1:2, 2:3)
>> A(1,2) = -A(2,1)
>> A
>> A(1:2, 1:2) = zeros(2,2)
>> A
```

Die Anzahl der Einträge von (Zeilen- oder Spalten-)Vektoren wird mit `length` bestimmt, die Größe von Matrizen mit `size`.

```
>> length(y)
>> size(ones(4,7))
>> [m,n] = size(Ay)
>> size(Ay,1)
```

Matrizen gleicher Größe können mit `+` addiert und mit `-` subtrahiert werden. Die Multiplikation mit Skalaren wird mit `*` geschrieben. Matrizen können mit `*` (im Sinne der Linearen Algebra) miteinander multipliziert werden, wenn die Dimensionen „passen“. Quadratische Matrizen können mit `^` potenziert werden. Die (konjugiert-)transponierte Matrix wird durch Nachstellen von `'` bezeichnet.

```
>> y + x'
>> A*y
>> 2*A + 3*B
>> A^2
```

Matrizen gleicher Größe können mit `.*`, `./` und `.^` **komponentenweise** multipliziert, dividiert und potenziert werden. Dabei darf jeweils ein Operator auch ein Skalar sein.

```
>> v = 1:10
>> v .^ 2
>> w = 1 ./ v
>> x ./ x
>> A .* B
```

Man kann elementare Funktionen wie `abs`, `sqrt`, `sin` usw. auf Matrizen anwenden. Sie wirken dann komponentenweise.

```
>> sin(A)
>> cos(pi*(1:5))
```

Mit `diag` erhält man die Diagonale einer Matrix und kann Diagonalmatrizen erzeugen. Der Befehl `inv` invertiert eine quadratische Matrix, `det` berechnet ihre Determinante.

1. MATLAB

```
>> A
>> diag(A)
>> diag([1, 2, 3])
>> B
>> inv(B)
>> det(B)
```

1.2. Programmieren mit MATLAB

1.2.1. Skriptdateien und Funktionsdateien

Außer der interaktiven Eingabe von MATLAB-Anweisungen gibt es noch zwei andere Möglichkeiten mit MATLAB zu arbeiten:

Skriptdateien

Man kann MATLAB-Anweisungen in eine Datei mit der Endung `.m` schreiben. Dafür kann man den eingebauten Editor verwenden, der mit

```
>> edit
```

gestartet wird. Gibt man dann den Dateinamen (ohne `.m`) an der Eingabeaufforderung `>> ein`, so werden die Befehle Zeile für Zeile ausgeführt – ganz so, als ob man sie interaktiv eingegeben hätte.

Funktionsdateien

Funktionsdateien werden ebenfalls in Dateien mit der Endung `.m` geschrieben und beginnen mit dem Wort **function**. Eine Funktion hat einen Funktionsnamen, der mit dem Dateinamen (ohne `.m`) übereinstimmen muss: Lautet der Name der Funktion also beispielsweise **func**, muss die Datei, in der sie definiert wird, unbedingt den Namen **func.m** haben.

Funktionen können Eingabeparameter besitzen, die in runden Klammern nach dem Funktionsnamen angegeben werden, sowie Ausgabeparameter, die vor dem Funktionsnamen angegeben werden. Eine Funktion kann mehrere Ausgabeparameter haben; diese werden dann in eckige Klammern gesetzt. Funktionen können aus anderen Funktionsdateien, Skriptdateien oder interaktiv durch Angabe des Namens zusammen mit einer passenden Übergabeparameterliste aufgerufen werden.

1. MATLAB

Nützliche Sprachelemente für Skript- und Funktionsdateien

MATLAB-Dateien können nur ausgeführt werden, wenn sie sich im aktuellen Verzeichnis (oder im Suchpfad) befinden. Das aktuelle Verzeichnis kann mit `pwd` abgefragt und mit `cd` geändert werden.

Kommentarzeilen in Skript- und Funktionsdateien beginnen mit `%`.

Unerwünschte Ausgaben können durch Nachstellen von Strichpunkten unterdrückt werden. In Funktionsdateien sind normalerweise alle Ausgaben unerwünscht.

Mit `disp` können Texte ausgegeben werden. Formatierte Ausgabe erhält man mit `sprintf` (die Syntax entspricht der C-Funktion `printf`).

Eine Funktion kann mit `return` vorzeitig verlassen werden. Der Befehl `error` bricht eine Funktion mit einer Fehlermeldung ab.

Mit der Tastenkombination `Control + c` kann die Ausführung manuell abgebrochen werden (z. B. bei einer Endlosschleife).

Mit dem `diary`-Befehl kann eine interaktive MATLAB-Sitzung in einer Datei protokolliert werden.

Beispiel für das Arbeiten mit Programmdateien

Funktionsdatei: kugel.m

```
% Funktion zur Berechnung des Umfangs U, der Oberflaeche F
% und des Volumens V einer Kugel mit Radius r
function [U, F, V] = kugel(r)
    U = 2*pi*r;
    F = 4*pi*r^2;
    V = (4/3)*pi*r^3;
end
```

Skriptdatei: kugel_test.m

```
% Beispiel einer Skriptdatei zum Aufruf der Funktion kugel
disp('Kugel mit Radius 0.5:')
format long
[U, F, V] = kugel(0.5)
```

Interaktive Eingabe

```
>> [U1, F1, V1] = kugel(1.0)
>> U = kugel(2.7)
>> kugel_test
```

1.2.2. Schleifen und bedingte Anweisungen

Eine **for-Schleife** hat die Form

```
for Laufvariable = von:schritt:bis
    Anweisungen
end
```

Die Anweisungen werden für alle Werte der Laufvariablen aus *von:schritt:bis* ausgeführt. Anstelle von *von:schritt:bis* kann auch ein beliebiger Vektor angegeben werden.

Funktionsdatei: fib1.m

```
% Berechnung eines Zeilenvektors aus den ersten n > 1 Fibonacci-Zahlen;
% die Folge der Fibonacci-Zahlen ist rekursiv definiert durch
% f(1) = 1, f(2) = 1, f(k) = f(k-1) + f(k-2), k > 2.
function f = fib1(n)
    f = ones(1, n); % Speicherplatz mit Einsen vorbelegen --> Effizienz
    for j = 3:n
        f(j) = f(j-1) + f(j-2);
    end
end
```

Eine **while-Schleife** hat die Form

```
while Bedingung
    Anweisungen
end
```

Die Anweisungen werden so lange ausgeführt, wie *Bedingung* wahr ist. Eine Bedingung wird normalerweise aus den Vergleichen == (gleich!), <, >, <=, >=, ~= (ungleich!) und den logischen Verknüpfungen && (und), || (oder) und ~ (nicht) aufgebaut.

Funktionsdatei: fib2.m

```
% Berechnung der kleinsten Fibonacci-Zahl >= m
function f = fib2(m)
    f_alt = 1;
    f = 1;
    while f < m
        f_neu = f + f_alt;
        f_alt = f;
        f = f_neu;
    end
end
```

Eine **bedingte Anweisung** hat z. B. die Form

```
if Bedingung
    Anweisungen1
else
    Anweisungen2
end
```

1. MATLAB

Ist *Bedingung* wahr, so werden nur die *Anweisungen1* ausgeführt, anderenfalls nur die *Anweisungen2*. Der *else*-Teil kann natürlich entfallen, wenn es nicht benötigt wird.

Funktionsdatei: pasch.m

```
% pasch(N) zaehlt, wie oft bei einer Simulation N-maligen Wuerfelns
% zwei gleiche Augenzahlen direkt aufeinander folgen.
function anzahl = pasch(N)
    wuerfe = floor(6*rand(1,N)) + 1; % simuliert N-maliges Wuerfeln
                                   % (floor rundet auf ganze Zahlen ab)

    anzahl = 0;
    for j = 1:N-1
        if wuerfe(j+1) == wuerfe(j)
            anzahl = anzahl + 1;
        end % if
    end % for

end % function
```

Hinweis: In MATLAB sollten aus Effizienzgründen Schleifen möglichst vermieden und durch Matrixoperationen ersetzt werden.

Nützlich sind dabei z. B. auch die Befehle `sum`, `prod`, `min` und `max`, die die Einträge eines Vektors aufsummieren, aufmultiplizieren bzw. Minimum und Maximum der Einträge berechnen. Wendet man sie auf Matrizen an, wirken sie spaltenweise.

1.2.3. Anonyme Funktionen und Funktionen als Eingabeparameter

Für einfache MATLAB-Funktionen muss nicht unbedingt jedesmal eine neue Funktionsdatei angelegt werden. Stattdessen kann eine MATLAB-Funktion auch mit einer sog. anonymen Funktion definiert werden:

```
>> f = @(x) x .* exp(-x)
>> f(0.1)
>> f(0:0.1:1)
```

Man kann MATLAB-Funktionen schreiben, die andere MATLAB-Funktionen als Eingabeparameter besitzen. Innerhalb solcher Funktionen kann die übergebene Funktion ganz normal aufgerufen werden, also z. B. $y = f(x)$ oder $[u,v] = f(x,y)$. Beim Aufruf einer solchen Funktion kann dann der Name einer beliebigen MATLAB-Funktion eingesetzt werden. Vorsicht: Vor die Namen von Funktionsdateien und eingebauten Funktionen muss ein `@`-Zeichen gesetzt werden (sog. *function handle*, z. B. `@sin`), Namen von anonymen Funktionen müssen dagegen ohne `@` angegeben werden!

Funktionsdatei: midpoint.m

```
function v = midpoint(f, a, b)
    v = f(0.5*(a+b));
end
```

1. MATLAB

```
>> f = @(x) x.^2 + x
>> midpoint(f, 0, 1)           % anonyme Funktion ohne @
>> midpoint(@sin, 0, 1)        % eingebaute Funktion mit @
>> midpoint(@(x) sin(x) + 2, 0, 1) % anonyme Funktion direkt eingesetzt
```

Ein wichtiges Beispiel einer eingebauten Funktion mit einer Funktion als Eingabeparameter ist `fplot` zur grafischen Darstellung von Funktionen.

```
>> fplot(@sin, [0, 2*pi])
>> f = @(x) 1+(x-1).^2
>> fplot(f, [0, 2, 0, 3])
```

Weitere solche Beispiele sind `fzero` zur Berechnung von Nullstellennäherungen in einer Dimension, `fminbnd` zur näherungsweise Minimierung eindimensionaler Funktionen, oder `quad` zur numerischen Berechnung von Integralen.

```
>> fzero(@sin, [3,4])
>> fminbnd(@(x) x.^2 - x, 0, 1)
>> f = @(x) 1./x
>> quad(f, 1, 2)
```

1.3. Grafik

1.3.1. Zweidimensionale Grafiken

Zweidimensionale Linien- und Punktgrafiken können mit dem `plot`-Befehl erzeugt werden: Sind x und y zwei n -dimensionale Vektoren, dann zeichnet `plot(x,y)` die Punkte (x_i, y_i) für $i = 1, \dots, n$ in ein Koordinatensystem und verbindet sie standardmäßig mit Linien.

```
>> plot([0,1,2,1,0], [1,0,1,2,1])
>> x = 0:0.01:1;
>> plot(x, x.^2)
```

Als dritter Eingabeparameter des `plot`-Befehls können in einfachen Anführungszeichen Kürzel für Farbe, Punktsymbol und Linienstil angegeben werden. Eine Liste der möglichen Angaben bekommt man mit `help plot`.

```
>> plot(1:10, 1:10, 'ro')
>> plot(x, sin(x), 'k.-')
```

Es können mehrere dieser Eingabetripel in einem `plot`-Befehl verwendet werden. Es wird dann alles in ein Koordinatensystem gezeichnet. Eine andere Möglichkeit mehrere Grafiken in ein Koordinatensystem zu zeichnen, bietet der `hold on`-Befehl. Er bewirkt, dass

1. MATLAB

nachfolgende Plot-Befehle die ursprüngliche Grafik nicht löschen, sondern zusätzlich mit eingezeichnet werden. Als dritte Möglichkeit kann man die y -Koordinaten zu einer Matrix zusammenfassen: Es werden dann alle Zeilen in ein gemeinsames Koordinatensystem geplottet.

Bei erstellten Grafiken kann nachträglich mit `axis` der Koordinatenausschnitt angepasst werden. Außerdem können mit `title`, `legend`, `xlabel`, `ylabel` und `text` Beschriftungen angebracht werden.

Skriptdatei: sincosplot.m

```
x = -5:0.05:5;
y = sin(x);
y1 = cos(x);
plot(x,y,'k',x,y1,'b--'), hold on
plot([-3*pi/2:pi/2], [1, -1, 1, -1], 'ro')
axis([-5, 5, -2, 2])
xlabel('x')
title('Die Sinusfunktion und ihre Ableitung')
legend('f(x) = sin(x)', 'f''(x) = cos(x)')
```

Logarithmische Maßstäbe auf den Koordinatenachsen erhält man mit `semilogx` (nur auf der x -Achse), `semilogy` (nur auf der y -Achse) und `loglog` (auf beiden Achsen) anstelle von `plot`. Es gibt auch noch zahlreiche weitere MATLAB-Befehle zum Erstellen zweidimensionaler Grafiken, beispielsweise `bar` (Säulendiagramme), `hist` (Histogramme), `pie` (Tortendiagramme), `stairs` (Treppenfunktionen), `quiver` (Vektorfelder), `contour` (Höhenlinien) usw. Auf den jeweiligen Hilfe-Seiten kann man sich über die Syntax dieser Grafikbefehle informieren. Dort findet man auch jeweils Anwendungsbeispiele.

Die Grafik wird in einem eigenen Fenster erzeugt. Mit dem `figure`-Befehl können auch mehrere Grafikfenster angelegt werden. Die Menüleiste des Grafikfensters bietet die Möglichkeit, die Grafik nachträglich zu verändern, auszudrucken, abzuspeichern. Beim Abspeichern können verschiedene Grafikformate gewählt werden, z.B. das EPS-, das PDF- oder das PNG-Format.

1.3.2. Dreidimensionale Grafiken

Mit dem `plot3`-Befehl kann man – ganz analog zum `plot`-Befehl – dreidimensionale Linien- und Punktgrafiken zeichnen: Sind x , y und z drei n -dimensionale Vektoren, dann zeichnet `plot3(x,y,z)` die Punkte (x_i, y_i, z_i) für $i = 1, \dots, n$ in ein Koordinatensystem und verbindet sie standardmäßig mit Linien.

Skriptdatei: kurve3d.m

```
% Beispielskript zum Plotten einer dreidimensionalen Kurve
t = 0:pi/100:2*pi;
x = cos(t);
y = sin(t);
z = cos(5*t);
```

1. MATLAB

```
plot3(x,y,z,'r')
grid on
```

Um den Graph einer Funktion zweier Variablen über einem Rechteck zu zeichnen, erzeugt man zunächst mit dem `meshgrid`-Befehl ein rechteckiges Punktegitter `[X,Y]`. Dabei stehen in der Matrix `X` die x -Koordinaten der Punkte und in `Y` die y -Koordinaten. Betrachte hierzu

```
>> [X,Y] = meshgrid(1:3, 4:7)
```

Dies liefert die beiden Matrizen `X,Y`, die die folgenden Koordinatenpaare in Matrixform beschreiben

$$\begin{pmatrix} (x_{11}, y_{11}) & (x_{12}, y_{12}) & (x_{13}, y_{13}) \\ (x_{21}, y_{21}) & (x_{22}, y_{22}) & (x_{23}, y_{23}) \\ (x_{31}, y_{31}) & (x_{32}, y_{32}) & (x_{33}, y_{33}) \\ (x_{41}, y_{41}) & (x_{42}, y_{42}) & (x_{43}, y_{43}) \end{pmatrix} = \begin{pmatrix} (1, 4) & (2, 4) & (3, 4) \\ (1, 5) & (2, 5) & (3, 5) \\ (1, 6) & (2, 6) & (3, 6) \\ (1, 7) & (2, 7) & (3, 7) \end{pmatrix}.$$

Daraus können dann komponentenweise die z -Koordinaten berechnet werden: z. B. `Z = 1./(1+X.^2+Y.^2)`. Der Graph wird dann mit `surf(X,Y,Z)` (Darstellung als Fläche) oder `mesh(X,Y,Z)` (Darstellung als Gitternetz) gezeichnet.

Skriptdatei: graph3d.m

```
% Beispielskript zum Zeichnen des Graphen einer Funktion von 2 Variablen
[X,Y] = meshgrid(-1:0.1:1, -2:0.1:2);
Z = sin(pi*X).*sin(pi*Y);
surf(X,Y,Z)
```

Die Grafik kann nach Anklicken von **Rotate 3D** (Button in der Toolbar) mit der Maus gedreht werden. Die Blickrichtung kann auch mit dem `view`-Befehl geändert werden.

Das verwendete Farbschema kann mit dem `colormap`-Befehl geändert werden. Vordefinierte Paletten sind z. B. `gray`, `hot`, `cool`, `spring`, `summer` usw. Der Befehl `colorbar` blendet die Farbskala ein.

Flächen können auch „glatt“ dargestellt werden (`shading interp`).

Dreidimensionale Grafiken können beleuchtet werden. Der einfachste Befehl hierfür ist `camlight`. Der dafür verwendete Algorithmus kann mit `lighting` eingestellt werden: der beste (und langsamste) ist `lighting phong`.

Skriptdatei: kugelplot.m

```
[X,Y,Z] = sphere;
surf(X,Y,Z)
axis equal
colormap cool
shading interp
camlight
lighting phong
```

1.4. Numerische Lineare Algebra

1.4.1. Lineare Gleichungssysteme und Matrix-Faktorisierungen

Der wichtigste Befehl zur Lösung linearer Gleichungssysteme ist der `\`-Operator: Ist A eine reguläre $n \times n$ -Matrix und b ein Spaltenvektor der Länge n , dann wird die Lösung des linearen Gleichungssystems $A \cdot x = b$ durch $x = A \backslash b$ berechnet².

```
>> clear all
>> A = [1,2,3; 4,5,6; 7,8,0]
>> b = [1;0;0]
>> x = A \ b
>> A * x
```

Hat man mehrere rechte Seiten, so kann man diese spaltenweise zu einer Matrix B zusammenfassen und die Lösungen mit $A \backslash B$ berechnen.

```
>> A \ eye(3)
>> inv(A)
```

Will man ein lineares Gleichungssystem mit einem bestimmten Verfahren lösen, so kann man die Befehle `lu` (LR-Zerlegung), `chol` (Cholesky-Zerlegung) und `qr` (QR-Zerlegung) verwenden. Die LR-Zerlegung wird normalerweise mit $[L,R] = \text{lu}(A)$ berechnet. Hier ist R eine obere Dreiecksmatrix und L eine permutierte untere Dreiecksmatrix mit $A = L \cdot R$. Die Lösung von $A \cdot x = b$ kann nun durch $z = L \backslash b$ und $x = R \backslash z$ berechnet werden. Alternativ kann man durch den Aufruf $[L,R,P] = \text{lu}(A)$ auch zusätzlich die Permutationsmatrix explizit berechnen.

```
>> A = rand(5)
>> [L,R] = lu(A)
>> L*R - A
```

Der `chol`-Befehl liefert beim Aufruf $R = \text{chol}(B)$ eine **obere (!)** Dreiecksmatrix R mit $B = R' \cdot R$

```
>> B = A*A'      % somit ist B symmetrisch
>> R = chol(B)
>> R'*R - B
```

Der `qr`-Befehl liefert bei Aufruf $[Q,R] = \text{qr}(A)$ eine QR-Zerlegung einer $m \times n$ -Matrix A in eine orthogonale $m \times m$ -Matrix Q und eine obere Dreiecksmatrix R . Mit $[Q,R] = \text{qr}(A,0)$ werden im Fall $m > n$ nur die ersten n Spalten von Q und die ersten n Zeilen von R berechnet.

²Siehe auch die Hilfe zu `mldivide`.

1. MATLAB

```
>> C = [ones(4,1), [1;2;3;4]]
>> [Q,R] = qr(C)
>> Q*R - C
```

1.4.2. Norm und Kondition

Mit dem `norm`-Befehl können diverse Normen von Vektoren und Matrizen berechnet werden. Voreingestellt ist die euklidische Vektornorm bzw. die von ihr induzierte Matrixnorm (Spektralnorm). Die Kondition von Matrizen wird mit `cond` berechnet. Die dabei verwendete Norm kann wie beim `norm`-Befehl gewählt werden.

```
>> norm(ones(3,1)) % euklidische Norm
>> A = [1,2;3,4]
>> [norm(A,1), norm(A,2), norm(A,inf), norm(A, 'fro')]
>> [cond(A,1), cond(A,2), cond(A,inf), cond(A, 'fro')]
>> cond(A)
>> norm(A)*norm(inv(A))
```

1.4.3. Eigenwertaufgaben

Der `eig`-Befehl liefert beim Aufruf `e = eig(A)` die Eigenwerte der Matrix `A` als Spaltenvektor. Mit `[V,D] = eig(A)` erhält man eine Ähnlichkeitstransformation auf Diagonalgestalt: $A*V = V*D$, d. h. die Spalten von `V` sind die Eigenvektoren von `A`.

```
>> A = hilb(4) % die 4x4 Hilbert-Matrix H(i,j) = 1/(i+j-1)
>> eig(A)
>> [V,D] = eig(A)
>> V \ A * V
```

Die Singulärwertzerlegung $C = U*S*V'$ einer Matrix `C` mit orthogonalen Matrizen `U`, `V` und einer Diagonalmatrix `S` erhält man mit `[U,S,V] = svd(C)`.

```
>> C = [ones(4,1), (1:4)']
>> [U,S,V] = svd(C)
>> U*S*V'
```

1.5. Warum ist MATLAB schnell?

1.5.1. LAPACK und BLAS

MATLAB kann bei vielen Rechnungen in der Linearen Algebra durchaus mit komplizierten Programmen mithalten. Der Grund ist die konsequente Verwendung von LA-

1. MATLAB

PACK³, welches wiederum auf BLAS⁴ Routinen aufsetzt. LAPACK (Linear Algebra PACKage) ist eine in Fortran geschriebene Softwarebibliothek, die effiziente Routinen zur Lösung linearer Gleichungssysteme, linearer Ausgleichsprobleme und von Eigenwertproblemen beinhaltet. BLAS (Basic Linear Algebra Subprograms) ist eine Softwarebibliothek, die elementare Operationen der linearen Algebra wie Vektor- und Matrixmultiplikationen implementiert. In der Hilfe zu den MATLAB-Funktionen werden die verwendeten LAPACK-, bzw. BLAS-Routinen erwähnt, siehe z.B.

```
>> doc mtimes
>> doc eig
```

1.5.2. Matrixmultiplikation

Seien A und B zwei Matrizen, der Einfachheit halber aus $\mathbb{R}^{n \times n}$. Das Matrixprodukt $C = AB$ ist komponentenweise gegeben durch

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

für alle $i, j = 1, \dots, n$. In MATLAB schreiben wir einfach

```
>> C=A*B
```

Eine mögliche Implementierung leitet sich direkt aus obiger Definition ab (alle Komponenten der Matrix C seien bereits auf 0 gesetzt):

```
% Form ijk
for i=1:1:n
    for j=1:1:n
        for k=1:1:n
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
        end
    end
end
end
```

Dieser Algorithmus benötigt $O(n^3)$ Fließkommaoperationen unabhängig von der Reihenfolge der ijk Schleifen. Viel wesentlicher für die Geschwindigkeit der Routinen ist wie die Matrizen abgespeichert werden und in welcher Reihenfolge auf die Komponenten zugegriffen wird. Die Matrix A

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

³<http://www.netlib.org/lapack/>

⁴<http://www.netlib.org/blas/>

1. MATLAB

wird in MATLAB (und LAPACK) als ein zusammenhängender Speicherbereich wie ein langer Vektor abgelegt

$$(a_{11}, a_{21}, \dots, a_{n1}, \quad a_{12}, a_{22}, \dots, a_{n2}, \quad \dots, \quad a_{1n}, a_{2n}, \dots, a_{nn}).$$

Es werden also die Spaltenvektoren aneinandergesetzt. Bei großen Matrizen ist wegen dem hierarchischen Speicherlayout derzeitiger Prozessoren der aufeinanderfolgende Zugriff in einem Spaltenvektor schneller als der aufeinanderfolgende Zugriff innerhalb eines Zeilenvektors. Hier muss gegebenenfalls der Cache neu gefüllt werden.

Im obigen Algorithmus der Form `ijk` wird für jeden Eintrag c_{ij} in der innersten Schleife ein Skalarprodukt zwischen dem Zeilenvektor $A(i, :)$ und dem Spaltenvektor $B(:, j)$ berechnet. Die Form `ijk` ist besonders ineffizient, da das Ergebnis *zeilenweise* in C für jedes ij gespeichert wird und zur Berechnung des Skalarprodukts einmal *zeilenweise* und einmal spaltenweise auf A und B zugegriffen wird. Das Permutieren der Indizes `ijk` liefert fünf weitere Möglichkeiten diese Matrixmultiplikation zu Implementieren. Die Form `jik` erlaubt einen spaltenweisen Zugriff auf C . Das Problem des zeilenweisen Zugriffs beim Skalarprodukt bleibt und weiterhin wird für jedes ij das Ergebnis sofort nach C geschrieben werden.

Die Form `jki` entspricht der BLAS Routine `DGEMM` und wird in den meisten Fällen von MATLAB, verwendet.

```
% Form jki
for j=1:1:n
    for k=1:1:n
        for i=1:1:n
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
        end
    end
end
end
```

Es wird der k -te Spaltenvektor $A(:, k)$ mit $B(k, j)$ multipliziert und zum Spaltenvektor $C(:, j)$ addiert. In modernen Prozessoren können alle Spaltenvektoren $A(:, k)$ in den Cache oder in Vektorregister geladen, und jeweils mit den Zahlen $B(k, j)$ multipliziert auf-addiert werden. Erst zum Schluss ist das Speichern im Spaltenvektor $C(:, j)$ notwendig. Diese Operation wird „generalized SAXPY“ oder auch GAXPY genannt. Ein weiterer Vorteil der Form `jki` bzw. der GAXPY Operation ist, dass mit ihr effizient die endliche Größe des Caches oder der Vektorregister berücksichtigt werden kann (das sogenannte blocking). Auch wird das Pipelining von Fließkomma- und Vektoroperationen unterstützt.

Am Beispiel der Matrixmultiplikation habe wir gesehen, dass eine direkte Implementierung sehr ineffektiv sein kann. Erst eine Umsortierung unter Berücksichtigung des Speicherzugriffs führt zu effizienten Verfahren. Für viele Verfahren der Numerischen Linearen Algebra gibt es solche speziellen Umformungen der Lehrbuch-Algorithmien (z.B.

auch für die LU-Zerlegung). Diese speziellen Algorithmen sind in der LAPACK Bibliothek implementiert und werden meist von MATLAB genutzt.

1.5.3. Dünnbesetzte Matrizen

In vielen Anwendungen treten sehr große Matrizen auf, bei denen aber nur wenige Einträge von null verschieden sind, sog. *dünnbesetzte* (engl. *sparse*) Matrizen. Beim Rechnen mit dünnbesetzten Matrizen ist es offensichtlich günstig, Datenstrukturen und Algorithmen zu verwenden, die diese Besetzungsstruktur ausnutzen.

In MATLAB können dünnbesetzte Matrizen z. B. mit den Befehlen `sparse` und `spdiags` erzeugt werden. Dabei erzeugt `S = sparse(i,j,s,m,n)` eine dünnbesetzte $m \times n$ -Matrix mit $S(i(k), j(k)) = s(k)$. Zum Erzeugen der dünnbesetzten $n \times n$ -Einheitsmatrix gibt es den Befehl `speye(n)`. Der Befehl `spdiags(C,d,m,n)` erzeugt eine dünnbesetzte $m \times n$ -Bandmatrix, indem die Spalten von C in die durch die Komponenten von d bezeichneten Diagonalen eingetragen werden. Der Befehl `full` wandelt eine dünnbesetzte Matrix in eine gewöhnliche Matrix um. Mit `spy` kann man sich die Besetzungsstruktur grafisch darstellen lassen

```
>> S = sparse([1,2,3,4], [2,1,4,3], [9,9,9,9], 4, 4)
>> full(S)
>> spy(S)
>> D = [-ones(10,1), 2*ones(10,1), -ones(10,1)]
>> B = spdiags(D, [-1,0,1], 10, 10)
>> full(B)
>> spy(B)
```

Mit dünnbesetzten Matrizen kann in MATLAB genauso wie mit vollbesetzten Matrizen gearbeitet werden. Die Auswahl der entsprechenden für dünnbesetzte Matrizen geeigneten numerischen Verfahren trifft MATLAB automatisch. Da es für große dünnbesetzte Matrizen aber im Allgemeinen zu aufwändig ist, alle Eigenwerte und Eigenvektoren zu berechnen, gibt es für diesen Fall den Befehl `eigs`: Mit `eigs(S,k)` berechnet man z. B. die k betragsgrößten Eigenwerte der dünnbesetzten Matrix S .

```
>> S = bucky % duennbesetzte Beispielmatrix der Groesse 60x60
>> spy(S)
>> spy(S^2)
>> S \ rand(60,1)
>> eigs(S,4)
```

2. Maple

2.1. Grundlagen, Rechnen mit Zahlen und Ausdrücken

Das Programm Maple ist ein Beispiel für ein sogenanntes Computeralgebrasystem (CAS). Es kann nicht nur numerisch rechnen wie ein Taschenrechner oder MATLAB, sondern auch symbolisch. Das heißt, es kann z. B. mathematische Ausdrücke umformen und vereinfachen, Gleichungen lösen, Funktionen differenzieren und integrieren usw.

2.1.1. Starten, Bedienung, Beenden und Hilfe

Die Maple-Benutzeroberfläche wird durch Eingabe von `xmaple` gestartet.¹ Mit den Standardeinstellungen ist die Oberfläche aber eher zur Erstellung mathematischer Dokumente geeignet und nicht so sehr zum Erlernen der Sprache und zur Durchführung von Berechnungen. Wir verwenden in diesem Kurs daher die Einstellungen `Tools → Options... → Display → Input display: Maple Notation` und `Tools → Options... → Interface → Default format for new worksheets: Worksheet`.

Zur Eingabe: Groß/Kleinschreibung ist signifikant. Interaktive Eingaben an der Eingabeaufforderung `>` werden normalerweise mit einem Strichpunkt und Enter abgeschlossen und ausgeführt. In neueren Maple-Versionen kann der Strichpunkt weggelassen werden. Schließt man stattdessen die Eingabe mit einem Doppelpunkt ab, dann wird keine Ausgabe angezeigt, aber die Zeile wird dennoch ganz normal ausgeführt. Das ist praktisch, wenn man lange, komplizierte Zwischenergebnisse nicht sehen möchte. Das Kommentarzeichen ist `#`.

Es empfiehlt sich ein `restart` zu Beginn einer neuen Aufgabe

```
> restart;  
> 2+1;
```

Nicht benötigte Variablen werden gelöscht und Maple in einen fest definierten Startzustand versetzt.

Das umfangreiche Hilfesystem lässt sich über das Menü `Help` starten. Wenn man Hilfe zu einem Befehl benötigt, dessen Namen man schon kennt, ist der einfachste Weg, an der Eingabeaufforderung `?Befehlsname` einzugeben.

¹Die Konsolenversion wird mit `maple` gestartet.

2. Maple

```
> ?ifactor  
> ifactor(11111111);
```

2.1.2. Maple als Taschenrechner

Die Rechenoperatoren sind $+$, $-$, $*$, $/$ für die vier Grundrechenarten und $^$ für das Potenzieren. Maple hält sich an die üblichen Punkt-vor-Strich- und Klammerregeln. Beachten Sie, dass das Dezimalzeichen, wie in allen Programmiersprachen, der Punkt ist.

```
> 1 + 2;  
> 123456789 * 987654321;  
> 2^1000;  
> 1.2 + 2.3*3.4;
```

Maple rechnet symbolisch, wenn man nicht ausdrücklich eine numerische Auswertung verlangt. Das heißt, Ausdrücke wie $\frac{12}{30}$ oder $\sqrt{72}$ werden zwar vereinfacht, aber nicht als Dezimalzahl „ausgerechnet“.

```
> 12/30;  
> sqrt(72);
```

Mit dem Zuweisungsoperator `:=` können für Werte und Ausdrücke Variablennamen vergeben werden.

```
> a := sqrt(2);  
> a^5;
```

Man erhält Näherungen in Dezimalschreibweise mit dem Befehl `evalf`. Die Genauigkeit kann beim `evalf`-Befehl in eckigen Klammern mitangegeben werden.

```
> b := 4/14;  
> evalf(b);  
> evalf[30](b);
```

Maple kennt die Funktionen `abs` (Betragsfunktion), `sqrt` (Quadratwurzel), `exp`, `ln`, `sin`, `cos`, `tan`, `cot`, `arcsin`, `arccos`, `arctan`, `arccot` und noch viele andere. Eine Liste der bekannten Funktionen erhält man mit `?inifncs`. Fakultäten werden mit dem Ausrufezeichen eingegeben, Binomialkoeffizienten mit dem Befehl `binomial`. Die Kreiszahl π heißt `Pi`, e^x wird als `exp(x)` eingegeben².

²Achtung: `pi` bezeichnet eine Variable mit Namen `pi`, nicht die Kreiszahl `Pi`.

2. Maple

```
> ln(exp(5));
> 1000!;
> Lotto := binomial(49,6);
> evalf[100](Pi);
> cos(9999*Pi);
> arctan(1);
```

2.1.3. Rechnen mit Ausdrücken, Lösen von Gleichungen

Maple kann mathematische Ausdrücke (Terme) manipulieren, in denen Variablennamen vorkommen. Solche Ausdrücke können mit `simplify` vereinfacht, mit `expand` ausmultipliziert und mit `factor` faktorisiert werden.

```
> restart;
> f := (1-1/x^2)/(x*(x^2-1));
> simplify(f);
> expand((x+y)^6);
> g := x^4 - 2*a*x^2 - 3*a^2;
> factor(g);
```

Ausdrücke mit trigonometrischen Funktionen und Exponentialfunktionen werden durch `expand` mit Hilfe der Additionstheoreme auseinandergezogen. Umformungen in der umgekehrten Richtung bekommt man mit `combine`.

```
> expand(exp(2*x+y) + cos(3*x));
> combine(2*sin(x)^2 + cos(x)*sin(x));
> combine(2^x*(2^y)^2);
```

Für speziellere Termumformungen gibt es den Befehl `convert`, mit dem durch Angabe einer passenden Option³ sehr viele allgemeine Umformungen mathematischer Objekte durchgeführt werden können. Einige Beispiele:

```
> convert(sinh(x) + sin(x), exp);
> convert(tan(x) + cot(x), sincos);
> convert(binomial(2*n, n), factorial);
> convert(0.125*x + 0.3333333333, rational);
```

Mit dem Befehl `eval` kann man einen Ausdruck auswerten, wenn z. B. Variablen auf neue Werte oder Ausdrücke gesetzt werden. Bei mehreren einzusetzenden Gleichungen werden diese in eckigen oder geschweiften Klammern angegeben.

```
> restart;
> g := a*x^3 + x;
> eval(g, a=2);
> eval(g, {a=2, x=t+1});
```

³siehe `?convert` und spezieller z.B. `?convert[rational]`

2. Maple

Gleichungen und Ungleichungen können symbolisch mit dem Befehl `solve` gelöst werden. Das erste Argument ist die zu lösende (Un-)Gleichung, das zweite die Variable, nach der aufgelöst werden soll. Bei polynomialen Gleichungen werden dabei standardmäßig auch komplexe Nullstellen berechnet. Die Relationssymbole sind `=` (gleich!), `<>` (ungleich!), `<`, `>`, `<=`, `>=`.

```
> solve(x^5 - x = 0, x);
> solve(abs(x+1) = 3 + abs(x)/2, x);
> solve(x^2 <= 6 - x);
> solve(x^2 < 6 - x);
```

Bei Gleichungssystemen werden sowohl die Gleichungen als auch die Variablen als Mengen in Mengenklammern angegeben.

```
> restart;
> solve({2*x+y=3, x-5*y=2}, {x,y});
> solve({x*y*z = 8, x + y + z = 6, x^2 + y^2 + z^2 = 12}, {x,y,z});
```

2.1.4. Summen und Produkte

Summen werden mit dem Befehl `sum` eingegeben. Dabei wird $\sum_{k=b}^c a$ als `sum(a, k=b..c)` geschrieben. Die zwei(!) Punkte bedeuten, dass k von b bis c läuft. Diese Schreibweise sollte man sich gut merken, da sie auch bei vielen anderen Maple-Befehlen verwendet wird.

```
> restart;
> sum(1/n, n=1..10);
> sum(x^k, k=0..5);
```

Maple kann oft auch Summen mit allgemeinen Summationsgrenzen berechnen.

```
> s := sum(j, j=1..n);
> simplify(s);
> sum(cos(k*x), k=0..N);
```

Analog werden Produkte mit `product` eingegeben.

```
> product((x-i), i=-3..3);
> p := product(1-1/k^2, k=2..n);
> simplify(p);
```

2.1.5. Komplexe Zahlen

Die imaginäre Einheit i wird in Maple mit einem großen I eingegeben. Ansonsten kann mit komplexen Zahlen genauso wie mit reellen Zahlen gerechnet werden. Es gibt Funktionen für Realteil, Imaginärteil, Absolutbetrag und die konjugierte komplexe Zahl.

```
> restart;
> I^2;
> z := 3-4*I;
> z^2;
> Re(z); Im(z); abs(z); argument(z); conjugate(z);
```

Manchmal ist der Befehl `evalc` nützlich. Er stellt sein Argument z in der Form $\text{Re}(z) + \text{Im}(z)*I$ dar, wobei automatisch alle weiteren Variablen als reell angenommen werden.

```
> z := a + I*b;
> evalc(z^2*exp(z));
```

2.2. Datenstrukturen, Funktionen und Grafik

2.2.1. Sequenzen, Listen und Mengen

Maple benutzt ein komplexes System von Datentypen.⁴ Neben den algebraischen Ausdrücken (Typ *algebraic*) sind für uns vor allem die zusammengesetzten Typen Sequenz (*expression sequence*), Liste (*list*) und Menge (*set*) sowie die Typen *Vector* und *Matrix* für die Lineare Algebra wichtig.

Eine Sequenz besteht aus durch Kommas getrennten Ausdrücken. Einige Maple-Befehle (z.B. `solve`) geben Sequenzen als Ergebnis zurück. Man kann mit Indizes in eckigen Klammern auf die einzelnen Teilausdrücke zugreifen.

```
> restart;
> S := 5, x+y, 7.4, "Hallo";
> S[2];
> S[4];
> Nullst := solve(x*(x^2-9)=0, x);
> Nullst[3];
```

Mit dem Befehl `seq` kann man Sequenzen aus allgemeinen Ausdrücken erzeugen. Hier z.B. die ersten 5 Quadratzahlen. Sequenzen sind assoziativ. An eine Maple-Funktion werden die Parameter immer als eine Sequenz übergeben.

```
> seq(n^2, n=1..5);
> seq(n^2, n=1..5), 4;
```

⁴Eine Übersicht über die mehr als 200(!) vordefinierten Datentypen gibt die Hilfeseite `?type`.

2. Maple

Mit Listen kann man die Zusammengehörigkeit besser ausdrücken. Listen sind Sequenzen, die in eckige Klammern eingeschlossen sind. Sie sind nicht mehr assoziativ, d.h. es sind Listen von Listen möglich.

```
> L1 := [m,a,p,l,e];
> L2 := [seq(1/n, n=1..5)];
> L2[1];
> L:=[L1, L2];
> L[2];
```

Mit `op` können die eckigen Klammern von Listen entfernt werden. Das kann beispielsweise dazu verwendet werden, um Elemente an Listen anzufügen oder um Listen aneinander zu hängen. Die Länge einer Liste bekommt man mit `nops`. Mit `op` kann auch auf die einzelnen Ausdrücke in einer Liste zugegriffen werden.

```
> op(L1);
> L3 := [op(L2), 1/6];
> L4 := [op(L1), op(L3)];
> nops(L4);
> op(1,L4);
> L[1][2];
```

Allgemein extrahiert `op` die Operanden einer Funktion als Sequenz.

```
> op(sin(x));
> op(x^2);
> op(x^2 * exp(x));
```

Eine endliche Menge ist eine Sequenz in geschweiften Klammern, wobei die Anordnung beliebig ist und keine doppelten Einträge vorkommen. Für $\cap$ schreibt man `intersect`, für $\cup$ `union`, und die Mengendifferenz $\setminus$ heißt `minus`.

```
> restart;
> M1 := {1,3,7,8};
> M2 := {2,3,4,7,4};
> M1 intersect M2;
> M1 union M2;
> M1 minus M2;
```

2.2.2. Funktionen

Mathematische Funktionen können in Maple außer durch Ausdrücke mit Variablen auch durch Maple-Funktionen als Zuordnungen eingegeben werden. Der Funktionsoperator ist der Zuordnungspfeil $\rightarrow$, also ein Minuszeichen direkt gefolgt von einem Größerzeichen. Der Hauptvorteil ist dabei, dass man einfach Werte einsetzen kann, ohne den Befehl `eval` verwenden zu müssen.

2. Maple

```
> restart;
> f := x -> (x-3)^5;
> f(2);
> f(t+1);
> f(f(x));
```

Zu beachten ist dabei, dass man bei Befehlen, die Ausdrücke erwarten, nun $f(x)$ statt f übergeben muss.

```
> expand(f(x));
```

Bei Funktionen mehrerer Variablen müssen die Variablennamen geklammert werden.

```
> g := (x,y) -> x^2 + x*y + y^2;
> g(2,3);
> h := g(x+y,x-y);
> expand(h);
```

Mit dem `map`-Befehl ist es möglich, eine Maple-Funktion nacheinander auf alle Elemente einer Liste anzuwenden.

```
> map(x -> x^2, [1,3,7,14,23]);
```

2.2.3. Funktionsgraphen

Graphen von Funktionen einer Variablen werden mit `plot` gezeichnet, wobei die Funktion als Ausdruck mit einer Variablen übergeben werden kann. Es muss mindestens der zu plottende Bereich auf der Abszisse angegeben werden, aber es können noch zahlreiche weitere Einstellungen mitübergeben werden, siehe `?plot,options`.

```
> restart;
> plot(x^2*sin(x), x=-4..4);
> plot(sin(t), t=0..2*Pi, labels=[t,x], title="Sinus",
      color=blue, thickness=2, axes=boxed);
```

Man kann mehrere Funktionen in ein gemeinsames Koordinatensystem plotten, indem man eine Liste aus Funktionen übergibt.

```
> plot([sin(x),cos(x)], x=0..2*Pi, color=[red,blue]);
```

Der Graph von Funktionen zweier Variablen kann mit `plot3d` als Fläche in einem dreidimensionalen Koordinatensystem dargestellt werden.

```
> f := x^3 + 3*x*y + y^4;
> plot3d(f, x=-2..2, y=-2..2);
```

2.2.4. Weitere Grafikbefehle

Mit `plot` können auch zweidimensionale parametrische Kurven grafisch dargestellt werden.

```
> restart;
> plot([cos(t), sin(2*t), t=0..2*Pi]);
```

Weitere Grafikbefehle befinden sich in dem Package `plots`, welches vor der Verwendung der Befehle mit `with(plots);` nachgeladen werden muss. Wir werden noch sehen, dass sich auch viele andere Befehle in Packages befinden. Man muss also immer wissen, welches Package man für welchen Befehl einbinden muss.

Zum Beispiel können Zahlenlisten mit `listplot` grafisch dargestellt werden.

```
> with(plots);
> L := [seq(2 + (-1)^n/n, n=1..30)];
> listplot(L);
> listplot(L, style=point);
```

Hier eine Auswahl weiterer Grafikbefehle, Erläuterungen finden Sie in der jeweiligen Hilfeseite:

- Dreidimensionale parametrische Kurven können mit `spacecurve` gezeichnet werden.
- Mit `implicitplot` und `implicitplot3d` können durch implizite Gleichungen gegebene Kurven bzw. Flächen geplottet werden.
- Höhenlinienbilder von Funktionen zweier Variablen können mit `contourplot` erzeugt werden.
- Punkte können mit `pointplot` geplottet werden.

Mit `animate` lassen sich animierte Grafiken erzeugen. Die Syntax ist etwas gewöhnungsbedürftig: Das erste Argument ist der zu animierende Plot-Befehl, danach werden die Argumente dieses Befehls als Liste übergeben. Das dritte Argument ist der Bereich des (Zeit-)Parameters, der variiert wird. Mit der Option `frames` kann schließlich noch die Anzahl der zu erstellenden Einzelbilder angegeben werden. Nach Anklicken der fertigen Animation erscheint ein Menü und eine Werkzeugleiste zur Ablaufsteuerung.

```
> plot( sin(x), x=0..Pi );
> animate(plot, [sin(t*x), x=0..Pi], t=1..2);
```

Alle Maple-Grafiken können durch Anklicken mit Hilfe des Menüs weiter bearbeitet werden und in verschiedene Grafikformate – z.B. in das pdf-Format – exportiert werden (Menü `Export`).

2.3. Analysis

2.3.1. Grenzwerte und Reihen

Grenzwerte von Folgen und Funktionen werden mit `limit` berechnet. Dazu wird $\lim_{x \rightarrow a} f(x)$ als `limit(f(x), x=a)` geschrieben, wobei ∞ als `infinity` angegeben wird.

```
> restart;
> limit(1/n, n=infinity);
> limit((1+x/n)^n, n=infinity);
> f := sin(x)/x;
> limit(f, x=0);
```

Links- oder rechtsseitige Limites können durch Angabe der Optionen `left` bzw. `right` bestimmt werden.

```
> g := x/abs(x);
> limit(g, x=0);
> limit(g, x=0, left);
> limit(g, x=0, right);
```

In diesem Zusammenhang soll erwähnt werden, dass man abschnittsweise definierte Funktionen mit `piecewise` angeben kann. Als Beispiel soll die Funktion

$$f(x) = \begin{cases} 1+x & \text{für } x \leq 0 \\ e^{-x} & \text{für } 0 < x \leq 2 \\ \sin(x) & \text{sonst} \end{cases}$$

betrachtet werden. Man beachte genau die Syntax: Die Bedingung wird jeweils *vor* dem zugehörigen Funktionsausdruck angegeben. Bei zweiseitigen Bedingungen wie $0 < x \leq 2$ wird nur `x <= 2` geschrieben und im sonst-Fall wird die Bedingung einfach weggelassen.

```
> f := piecewise(x <= 0, 1+x, x <= 2, exp(-x), sin(x));
> plot(f, x=-2..5);
```

Solche Funktionen können mit `iscont` auf Intervallen auf Stetigkeit geprüft werden.

```
> iscont(f, x=-1..1);
> iscont(f, x=-infinity..infinity);
```

Werte von unendlichen Reihen können mit `sum` berechnet werden (obere Grenze `infinity`).

```
> sum(1/n^2, n=1..infinity);
> sum(1/n, n=1..infinity);
> s := sum(2^n/binomial(3*n, n), n=0..infinity);
> evalf(s);
```

2. Maple

Das funktioniert auch für Potenzreihen und manche anderen Funktionenreihen.

```
> sum(x^n/n!, n=0..infinity);  
> sum(cos(n*x)/2^n, n=0..infinity);
```

Man beachte allerdings, dass Funktionenreihen, die nicht überall konvergieren, nur dann (formal) ausgewertet werden, wenn man die Option `formal=true` angibt. Mit der Option `parametric` erhält man eine komplette Fallunterscheidung für den Grenzwert.

```
> sum(x^n/n, n=1..infinity, formal=true);  
> sum(x^n/n, n=1..infinity, parametric);
```

2.3.2. Differentiation

Ausdrücke können mit `diff` differenziert werden. Dabei sind die Variablennamen, nach denen abgeleitet werden soll, durch Kommas getrennt anzugeben. Auf diese Weise können natürlich auch partielle Ableitungen berechnet werden.

```
> restart;  
> f := x^2*sin(x);  
> diff(f, x);  
> diff(f, x, x);  
> g := x*sin(x+y);  
> diff(g, x);  
> diff(g, y);
```

Um höhere Ableitungen einzugeben, gibt es die Abkürzung `x$n` für die n -malige Ableitung nach der Variablen x .

```
> diff(x^2*exp(x), x$10);
```

2.3.3. Taylor-Entwicklung

Mit dem Befehl `taylor` können Funktionen einer Variablen um eine gegebene Entwicklungsstelle bis zu einer gegebenen Ordnung in Taylor-Polynome entwickelt werden. Vorsicht: Die Ordnung ist dabei die Ordnung des Restgliedes, die Maple in O -Notation angibt; der Polynomgrad ist also um 1 kleiner.

```
> taylor(exp(x), x=0, 5);
```

Wenn mit dem Ergebnis des `taylor`-Befehls weitergerechnet werden soll, muss es zuerst durch Abschneiden des Restgliedes in ein Polynom umgewandelt werden. Das leistet der `convert`-Befehl mit der Option `polynom`.

2. Maple

```
> f := x*sin(x);
> Tf := taylor(f, x=1, 4);
> p := convert(Tf, polynom);
> plot([f,p], x=-1..4);
```

Funktionen mehrerer Variablen können mit `mtaylor` in Taylor-Polynome entwickelt werden. Hierbei gibt Maple allerdings kein Restglied an. Die angegebene Ordnung bezieht sich aber ebenfalls auf die Restgliedordnung.

```
> g := (x+y)/(x-y);
> mtaylor(g, [x=2,y=1], 3);
```

Mit dem `series`-Befehl ist auch die Berechnung von allgemeineren Reihenentwicklungen mit gebrochenen oder negativen Potenzen möglich.

```
> series(cot(x), x=0, 4);
> sterling := simplify(series(log(n!), n=infinity, 2));
```

2.3.4. Integration

Das bestimmte Integral $\int_a^b f(x) dx$ wird als `int(f(x), x=a..b)` eingegeben und (falls möglich) berechnet. Auch uneigentliche Integrale sind Maple bekannt.

```
> restart;
> int(x^2, x=0..1);
> f := x^3*exp(-x^2);
> int(f, x=-1..2);
> int(1/(1+x^2), x=0..infinity);
```

Bei Eingabe von `int(f(x), x)` ohne Integrationsgrenzen bestimmt Maple irgendeine Stammfunktion von $f(x)$.

```
> int(x^2, x);
> int(f, x);
```

Kann Maple ein Integral nicht symbolisch auswerten, gibt es den Integralausdruck unverändert zurück. Dieser kann wie üblich mit `evalf` numerisch berechnet werden.

```
> g := exp(cos(sin(x)));
> int(g, x);
> c := int(g, x=0..1);
> evalf[20](c);
```

Das geht schneller, wenn man Maple sagt, dass es eine symbolische Berechnung gar nicht erst versuchen soll. Dazu verwendet man `Int` statt `int`.

```
> evalf[20](Int(g, x=0..1));
```

2.3.5. Vektoranalysis

Die folgenden Befehle zur mehrdimensionalen Differential- und Integralrechnung müssen mit `with(VectorCalculus)` geladen werden.

```
> restart;
> with(VectorCalculus);
```

Mit `Gradient` und `Hessian` werden der Gradient bzw. die Hesse-Matrix einer skalaren Funktion mehrerer Variablen berechnet. `Laplacian` wendet den Laplace-Operator auf die Funktion an. Das Ergebnis von `Gradient` ist ein Vektorfeld in Basisdarstellung (kann mit `BasisFormat(false)` geändert werden.)

```
> f := x^2 + 2*x*y^3;
> Gradient(f, [x,y]);
> Hessian(f, [x,y]);
> Laplacian(f, [x,y]);
```

Vektorwertige Funktionen erzeugt man üblicherweise mit `VectorField`. Dazu muss das zu verwendende Koordinatensystem angegeben werden. Das kann global mit `SetCoordinates` erfolgen.

```
> SetCoordinates('cartesian' [x,y,z]);
> F := VectorField(<2*(x+y^3), 6*x*y^2, z>);
> F[1];
```

Divergenz und Rotation von Vektorfeldern werden mit `Divergence` bzw. `Curl` berechnet, die Jacobi-Matrix mit `Jacobian`.

```
> Divergence(F);
> Curl(F);
> Jacobian(F);
```

Mit `ScalarPotential` wird das skalare Potential (Stammfunktion) eines Vektorfeldes bestimmt, falls ein solches existiert.

```
> ScalarPotential(F);
```

Für die mehrdimensionale Integralrechnung gibt es beispielsweise die Befehle `LineInt` zur Berechnung von Kurvenintegralen von Vektorfeldern, `SurfaceInt` für Flächenintegrale und `Flux` zur Berechnung des Flusses von Vektorfeldern durch Flächen. Beispiele entnehmen Sie bitte der Hilfe.

2.4. Lineare Algebra

Die Befehle in diesem Abschnitt befinden sich im Package `LinearAlgebra`, das zuerst mit `with(LinearAlgebra)` geladen werden muss.

```
> restart;
> with(LinearAlgebra);
```

2.4.1. Rechnen mit Vektoren und Matrizen

Vektoren werden in spitzen Klammern eingegeben. Bei Spaltenvektoren werden die Einträge durch Kommas getrennt, bei Zeilenvektoren durch den vertikalen Strich `|`. Alternativ kann man Spaltenvektoren mit dem Befehl `Vector` eingeben.

```
> v := <1,2>;
> w := <3.4|5.6|7.8>;
> e := Vector([1,0,0]);
```

Bei Matrizen werden die Zeilen- bzw. Spalten ebenfalls in spitze Klammern gesetzt, wobei die Matrix wahlweise als Zeilenvektor aus Spaltenvektoren oder als Spaltenvektor aus Zeilenvektoren eingegeben werden kann. **Merkregel:** Strich heißt neue Spalte, Komma heißt neue Zeile. Auch hier gibt es als Alternative den Befehl `Matrix`.

```
> A := < <1|2>, <3|4> >;
> B := < <1,1,0> | <0,3,4> | <-1,2,t> >;
> C := Matrix([[0,1,0],[1,2,3]]);
> C[2,3];
> Row(C,1);
```

Im `LinearAlgebra`-Package findet man auch Befehle, die spezielle Vektoren und Matrizen erzeugen: Z. B. `IdentityMatrix` für die Einheitsmatrix, `ZeroMatrix` und `ZeroVector` für Nullmatrizen bzw. Nullvektoren, `DiagonalMatrix` für Diagonalmatrizen und `UnitVector` für die Einheitsvektoren.

```
> I3 := IdentityMatrix(3);
> Z := ZeroMatrix(3,2);
> Di := DiagonalMatrix(v);
> e2 := UnitVector(2,3);
```

Außerdem kann man mit dem `Matrix`-Befehl Matrizen erzeugen, deren Einträge sich durch eine Formel angeben lassen. Dazu können als drittes Argument Konstanten oder Funktionen von Zeilen- und Spaltenindex verwendet werden.

```
> H := Matrix(5,5, (i,j) -> 1/(i+j-1));
```

2. Maple

Wenn man Prozeduren mit Vektoren und Matrizen schreibt, braucht man gelegentlich die Anzahl ihrer Zeilen und Spalten: Die Zeilenzahl einer Matrix A bekommt man mit `RowDimension(A)`, ihre Spaltenzahl mit `ColumnDimension(A)`; die Anzahl der Komponenten eines Zeilen- oder Spaltenvektors v erhält man mit `Dimension(v)`.

```
> RowDimension(Z);
```

Man kann Vektoren bzw. Matrizen gleicher Größe mit $+$ addieren und mit $-$ subtrahieren. Die Multiplikation mit Skalaren wird mit $*$ angegeben. Vorsicht: Die Matrixmultiplikation im Sinne der Linearen Algebra wird mit dem Punkt $.$ geschrieben! Außerdem können quadratische Matrizen mit $^{\wedge}$ potenziert werden.

```
> B + 2*I3;
> A . v;
> A . C;
> C . A;
> A^20;
```

Den transponierten Vektor bzw. die transponierte Matrix erhält man mit dem Befehl `Transpose`.

```
> C;
> Transpose(C);
> Transpose(v) . v;
```

Skalarprodukte können auch mit `DotProduct` berechnet werden. Verschiedene Vektor- und Matrixnormen erhält man mit `Norm`. (Vorsicht: Die Standardeinstellung ist die ∞ -Norm; die euklidische Norm bekommt man mit `Norm(v,2)`.) Der Befehl `VectorAngle` berechnet den Winkel zwischen Vektoren.

```
> DotProduct(v,v);
> Norm(v,2);
> phi := VectorAngle(v, A.v);
> evalf(convert(phi,degrees))
```

2.4.2. Lineare Gleichungssysteme, Basen, Kern und Bild

Die Lösung $\mathbf{x} \in \mathbb{R}^n$ eines linearen Gleichungssystems $A\mathbf{x} = \mathbf{b}$ in Matrixform mit $A \in \mathbb{R}^{m \times n}$ und rechter Seite $\mathbf{b} \in \mathbb{R}^m$ kann mit `LinearSolve(A,b)` berechnet werden.

```
> x := LinearSolve(A, v);
> A . x;
```

Zusätzlich kann man für den Fall nicht eindeutiger Lösbarkeit noch angeben, wie der Grundname der verwendeten freien Parameter lauten soll. Das geschieht durch Angabe der Option `free='s'`, wenn der Name s sein soll.

2. Maple

```
> LinearSolve(C, v);  
> LinearSolve(C, v, free='s');
```

Die inverse Matrix A^{-1} kann mit `MatrixInverse(A)` oder einfach mit A^{-1} berechnet werden.

```
> A^(-1) . v;  
> MatrixInverse(B);
```

Der Rang einer Matrix, also die maximale Zahl linear unabhängiger Zeilen bzw. Spalten wird mit `Rank` bestimmt.

```
> B4 := eval(B, t=4);  
> Rank(B4);
```

Der Befehl `NullSpace` berechnet eine Basis des Kerns einer Matrix (genauer: der durch die Matrix gegebenen linearen Abbildung). Mit `ColumnSpace` kann eine Basis des Spaltenraums (also des Bildes) und mit `RowSpace` eine Basis des Zeilenraums bestimmt werden.

```
> C;  
> NullSpace(C);  
> ColumnSpace(C);  
> RowSpace(C);
```

Allgemein können mit `Basis`, `IntersectionBasis` und `SumBasis` Basen von Vektorräumen bzw. von Durchschnitten und Summen von Vektorräumen berechnen. Die Vektorräume sind dabei durch Listen aus erzeugenden Vektoren angegeben.

```
> v1 := <1,0,0>; v2 := <1,1,0>; v3 := <1,1,1>; v4 := <2,2,-1>;  
> Basis([v2,v3,v4]);  
> IntersectionBasis([[v1,v2],[v3,v4]]);
```

Mit `GramSchmidt` bestimmt man zu gegebenen Vektoren eine Orthogonalbasis des aufgespannten Vektorraums. Bei Angabe der Option `normalized` werden die berechneten Basisvektoren zu einer Orthonormalbasis normalisiert.

```
> GramSchmidt([v1,v3,v4], normalized);
```

2.4.3. Determinanten, Eigenwerte und Normalformen

Die Determinante einer quadratischen Matrix A kann mit `Determinant(A)` berechnet werden.

```
> restart;  
> with(LinearAlgebra):  
> A := <<1|2>, <4|3>>;  
> Determinant(A);
```

2. Maple

Das charakteristische Polynom einer quadratischen Matrix, also das Polynom $\det(tI - A)$, kann mit dem Befehl `CharacteristicPolynomial` berechnet werden. Das erste Argument ist die Matrix, das zweite der zu verwendende Variablenname.

```
> CharacteristicPolynomial(A, t);
```

Die Eigenwerte von A können mit `Eigenvalues(A)` berechnet werden; sie werden in einem Spaltenvektor zurückgegeben.

```
> Eigenvalues(A);
```

Mit dem Befehl `Eigenvectors` werden zusätzlich zu den Eigenwerten auch zugehörige Eigenvektoren bestimmt. Sie werden als Spalten einer Matrix zurückgegeben.

```
> E := Eigenvectors(A);  
> E[2];  
> E[2]^(-1) . A . E[2];
```

Will man die Spalten einer Matrix extrahieren, kann man den Befehl `Column` verwenden; z. B. ist `Column(A,2)` die zweite Spalte von A . Damit kann man einzelne Eigenvektoren aus dem Ergebnis des Befehls `Eigenvectors` herausholen. Analog dazu werden mit dem Befehl `Row` die Zeilen einer Matrix angesprochen.

```
> v2 := Column(E[2], 2);  
> A . v2;  
> 5 * v2;
```

Eine vor allem für theoretische Überlegungen wichtige Transformation von Matrizen auf Dreiecksgestalt ist die Jordan-Normalform. Dieses beweis- und rechentechnisch ziemlich aufwändige Verfahren erledigt der Befehl `JordanForm` (mit der Option `output='Q'` erhält man eine Matrix, deren Spalten eine Jordan-Basis bilden). Das in diesem Zusammenhang wichtige Minimalpolynom einer Matrix berechnet man analog zum charakteristischen Polynom mit `MinimalPolynomial`.

```
> B := Matrix([[2,2,-2,1],[-1,5,-3,2],[-1,2,-1,3],[-1,2,-4,6]]);  
> J := JordanForm(B);  
> m := MinimalPolynomial(B, t);  
> factor(m);
```

2.5. Ausblick: Weitere Teilgebiete der Mathematik

Maple versucht alle Teilgebiete der Mathematik, in denen es eine Art von Kalkül gibt, abzudecken. Dabei befinden sich meist alle spezielleren Befehle in Packages, die mit `with` nachgeladen werden müssen. In diesem Abschnitt sollen nun noch einige weitere Gebiete kurz vorgestellt werden.

2.5.1. Elementare Zahlentheorie

Bei natürlichen Zahlen kann man mit `isprime` überprüfen, ob sie prim oder zusammengesetzt sind. Die Zerlegung in Primfaktoren erhält man mit `ifactor`. Man beachte, dass es für große Zahlen algorithmisch wesentlich einfacher ist nachzuweisen, dass sie zusammengesetzt sind, als tatsächlich ihre Primfaktorzerlegung zu bestimmen.

```
> restart;
> isprime(13);
> n := 2^32+1;
> isprime(n);
> ifactor(n);
> n := 2^1024+1;
> isprime(n);
> ifactor(n); # Viel zu aufwendig: Abbrechen mit Stop
```

Die n -te Primzahl kann durch `ithprime(n)` bestimmt werden. Mit `nextprime` und `prevprime` findet man zu einer vorgegebenen Zahl die nächstgrößere bzw. nächstkleinere Primzahl.

```
> ithprime(10000);
> nextprime(10^6);
> prevprime(10^6);
```

Mit dem modulo-Operator `mod` kann mit Restklassen ganzer Zahlen gerechnet werden. Dabei ist jedoch zu beachten, dass bei Potenzen anstelle von `^` der *inerte Operator* `&^` verwendet wird, damit die Exponentiation nicht vor der Restklassenberechnung ausgeführt wird.

```
> 123*345+678 mod 100;
> 10! mod 11;
> 5^(-1) mod 101; # multiplikatives Inverses
> 9 ^ (12!) mod 13; # zu aufwendig: Abbrechen mit Stop
> 9 &^ (12!) mod 13;
```

Der größte gemeinsame Teiler ganzer Zahlen wird mit `igcd` (*integer greatest common divisor*) berechnet. Mit dem *erweiterten euklidischen Algorithmus* lässt sich der ggT g zweier Zahlen a, b in der Form $g = sa + tb$ darstellen. Diese Koeffizienten berechnet `igcdex`.

```
> igcd(123,456);
> igcdex(123,456,'s','t');
> s,t;
> s*123+t*456;
```

Im Package `numtheory` findet man zahlreiche zahlentheoretische Funktionen, u. a. `divisors` (Menge aller Teiler), `tau` (Teileranzahl) und `sigma` (Teilersumme).

2. Maple

```
> with(numtheory);  
> divisors(60);  
> tau(60);  
> sigma(60);
```

Die für viele zahlentheoretische und algebraische Probleme wichtige *eulersche φ -Funktion* heißt einfach `phi`.⁵

```
> phi(100);
```

2.5.2. Algebra

Die meisten Algorithmen eines Computeralgebrasystems wie Maple basieren auf Methoden der höheren Algebra. Dementsprechend stehen die meisten Algebra-Befehle stets zur Verfügung, ohne dass ein Package geladen werden muss.

Beim Rechnen mit Polynomen kann der Grad mit `degree`, die Koeffizienten mit `coeffs` und einzelnen Koeffizienten mit `coef` bestimmt werden.

```
> restart;  
> p := expand(product(x-i,i=1..5));  
> degree(p);  
> coeffs(p);  
> coef(p,x^3);
```

Der Quotient und der Rest bei Polynomdivisionen wird mit `quo` bzw. `rem` berechnet. Dabei muss die Polynomvariable explizit angegeben werden.

```
> q := x^2-4;  
> s := quo(p,q,x);  
> r := rem(p,q,x);  
> expand(s*q+r - p);
```

Der (erweiterte) euklidische Algorithmus lässt sich auch auf Polynome anwenden.⁶ Die Befehle hierzu heißen `gcd` bzw. `gcdex`.

```
> gcd(p,q);  
> gcdex(p,q,x,'s','t');  
> g := s*p+t*q;  
> expand(g);
```

⁵ $\varphi(n)$ ist die Anzahl der zu $n \in \mathbb{N}$ teilerfremden Zahlen. Das Beispiel zeigt also, dass genau 40 Zahlen aus $\{1, 2, \dots, 100\}$ zu 100 teilerfremd sind.

⁶Er lässt sich allgemein in Ringen durchführen, in denen eine Division mit Rest existiert, den sog. *euklidischen Ringen*.

2. Maple

Mit **factor** lassen sich bekanntlich Polynome über den rationalen Zahlen $\mathbb{Q}$ in irreduzible Faktoren zerlegen. Mit Hilfe des inerten Befehls **Factor** und **mod** lassen sich auch Faktorisierungen über den Körpern $\mathbb{Z}/p\mathbb{Z}$ mit einer Primzahl p bestimmen. Das Polynom im folgenden Beispiel ist also irreduzibel über $\mathbb{Q}$ und über $\mathbb{Z}/7\mathbb{Z}$, aber reduzibel über $\mathbb{Z}/13\mathbb{Z}$, es hat dort sogar ein Nullstelle:

```
> p := x^5+x+3;
> factor(p);
> Factor(p) mod 7;
> Factor(p) mod 13;
```

Zur höheren Algebra sei an dieser Stelle nur erwähnt, dass es für das Rechnen in endlichen Gruppen das Gruppentheorie-Package **group** und für das Rechnen in endlichen Körpern das Package **GF** (*Galois Field*) gibt. Ein weiteres wichtiges Ziel der Algebra-Vorlesungen ist die Untersuchung von Polynomen durch ihre sog. Galoisgruppe. Die theoretisch und rechentechnisch anspruchsvolle Bestimmung dieser Gruppe ist in Maple sehr einfach mit dem Befehl **galois** zu bewerkstelligen. Z. B. zeigt das folgende Ergebnis, dass die Galoisgruppe des Polynoms $x^4 - 2$ die sog. *Diedergruppe* D_4 mit 8 Elementen ist.

```
> galois(x^4-2);
```

2.5.3. Geometrie

Mit den Packages **geometry** und **geom3d** lassen sich in Maple recht einfach Berechnungen an zwei- bzw. dreidimensionalen geometrischen Figuren durchführen. Die meisten wichtigen geometrischen Objekte und ihre Eigenschaften sind dort vordefiniert. Mit **draw** können die Figuren gezeichnet werden.

An dieser Stelle soll je ein zwei- und ein dreidimensionales Beispiel als erster Eindruck genügen. Im ersten Beispiel wird aus drei Geraden g_1, g_2, g_3 ein Dreieck T und dessen Umkreis c konstruiert. Anschließend werden die Koordinaten des Mittelpunktes M von c berechnet und die ganze Situation grafisch dargestellt.

```
> restart;
> with(geometry):
> line(g1, x=1, [x,y]);
> line(g2, x+3*y=10, [x,y]);
> line(g3, [point(A,0,0), point(B,2,1)]);
> triangle(T, [g1,g2,g3]);
> circumcircle(c,T, 'centername'=M);
> coordinates(M);
> draw([g1,g2,g3,T,c(color=blue),M(color=green)], view=[-1..5,-1..5]);
```

Im folgenden dreidimensionalen Beispiel werden eine Ebene E , eine Kugel s und eine Gerade g konstruiert. Anschließend werden die Schnittpunkte von g mit E und s berechnet und schließlich wird wieder alles gezeichnet.

2. Maple

```
> restart;
> with(geom3d):
> plane(E,x+2*y+5*z=1,[x,y,z]);
> sphere(s,[point(M,0,0,3),2]);
> line(g,[point(A,0,-1,0),point(B,0,1,6)]);
> intersection(p1,g,E);
> coordinates(p1);
> intersection(plist,g,s);
> map(coordinates,plist);
> draw([s,E,g(color=red)],axes=boxed,view=[-5..5,-5..5,-3..8]);
```

2.5.4. Differentialgleichungen

Der grundlegende Befehl zum Lösen gewöhnlicher Differentialgleichungen ist **dsolve**. Analog zum **solve**-Befehl ist das erste Argument die zu lösende Differentialgleichung und das zweite die gesuchte Funktion. In diesem Fall versucht Maple, die allgemeine Lösung zu berechnen. Ebenfalls wie bei **solve** kann man in geschweiften Klammern eine Menge von Gleichungen angeben. Auf diese Weise kann man Anfangs- und Randbedingungen spezifizieren sowie Differentialgleichungssysteme angeben.

```
> restart;
> dgl := diff(y(t),t) = -2*t*(y(t))^2;
> dsolve(dgl,y(t));
> dsolve({dgl,y(0)=2},y(t));
> dgl1 := diff(x(t),t) = x(t)-y(t);
> dgl2 := diff(y(t),t) = x(t)+y(t);
> dsolve({dgl1,dgl2},{x(t),y(t)});
```

Im Package **DEtools** findet man diverse Befehle zur Manipulation und Lösung spezieller gewöhnlicher Differentialgleichungen. Zur Veranschaulichung gibt es dort auch **DEplot**. Damit lassen sich Richtungsfelder zusammen mit darin eingebetteten speziellen Lösungskurven grafisch darstellen.

```
> with(DEtools):
> DEplot(dgl,y(t),t=0..3,[y(0)=1],[y(0)=2]));
```

Maple kann mit **pdsolve** auch einige Typen partieller Differentialgleichungen lösen. Im Package **PDEtools** gibt es außerdem zahlreiche weitere Befehle zur Analyse und Manipulation partieller Differentialgleichungen.

```
> gl := diff(u(x,y),x)+diff(u(x,y),y)=x*y*u(x,y)^2;
> pdsolve(gl, u(x,y));
```

2.5.5. Wahrscheinlichkeitstheorie

Das Package `Statistics` bietet neben zahlreichen Befehlen zur deskriptiven Statistik und zur Durchführung statistischer Tests auch die Möglichkeit, symbolisch mit Zufallsvariablen zu rechnen. Zufallsvariable werden mit `RandomVariable` durch Angabe ihrer Verteilung (`Distribution`) erzeugt. Hierzu sind die wichtigsten Verteilungen bereits vordefiniert, z. B. `Normal`, `Exponential`, `Uniform`, `Binomial`, `Geometric` usw. Von Zufallsvariablen können wichtige Kenngrößen wie Erwartungswerte (`Mean`), Varianzen (`Variance`), Dichten (`PDF`), Verteilungsfunktionen (`CDF`) usw. bestimmt werden. Mit `Probability` lassen sich Wahrscheinlichkeiten von Ereignissen, die mit Zufallsvariablen formuliert sind, berechnen.

```
> restart;
> with(Statistics):
> X := RandomVariable(Exponential(lambda));
> Mean(X);
> Variance(X);
> Probability(X>1);
> PDF(X,t);
> F := CDF(exp(-X),t); # Verteilungsfunktion von exp(-X)
> with(plots):
> animate(plot, [F, t=0..2], lambda=0.1..3);
```

Zufallsvariable können auch durch Angabe ihrer Dichte oder ihrer Verteilungsfunktion definiert werden. Als Beispiel soll für eine Zufallsvariable X mit Lebesgue-Dichte $f(t) = \frac{1}{2} \exp(-|t-1|)$ der Erwartungswert von $|X|$ sowie die sog. *charakteristische Funktion* $\varphi(u) := E(e^{iuX})$ berechnet werden ⁷.

```
> f := t->exp(-abs(t-1))/2;
> int(f(t), t=-infinity..infinity); # Zur Kontrolle: f ist Dichte
> X:=RandomVariable(PDF=f);
> Mean(abs(X));
> simplify(CharacteristicFunction(X,u));
```

In einem abschließenden Beispiel soll nun für zwei unabhängige standardnormalverteilte Zufallsvariablen X und Y die Dichte von $\frac{X}{Y}$ und die Wahrscheinlichkeit, dass der Punkt (X, Y) im Einheitskreis liegt, berechnet werden.

```
> X := RandomVariable(Normal(0,1));
> Y := RandomVariable(Normal(0,1));
> PDF(X/Y,t);
> P1 := Probability(X^2+Y^2<1);
> evalf(P1);
```

⁷Die charakteristische Funktion ist eine Art Fouriertransformierte und spielt für die Untersuchung von Zufallsvariablen in der Wahrscheinlichkeitstheorie eine wichtige Rolle.

3. R

3.1. Grundlagen

R ist eine auf vielen Plattformen verfügbare interaktive Plattform zur statistischen Analyse und zur grafischen Darstellung von Daten.

3.1.1. Starten, Bedienung, Beenden und Hilfe

R hat keine grafische Benutzeroberfläche und wird durch Eingabe von `R` an der Konsole gestartet und mit `q()` beendet. Im Folgenden arbeiten wir ohne gespeicherte Workspaces; bitte beantworten Sie daher die Frage `Save workspace image?` beim Beenden immer mit `n`.

Zur Eingabe: Groß/Kleinschreibung ist wesentlich. Die interaktive Eingaben an der Eingabeaufforderung `>` werden mit Enter abgeschlossen und dann ausgeführt. Mit den Cursor-Tasten können Sie Befehle aus der Command-Line-History holen und bearbeiten.

Hilfe zu einem `befehl` erhalten Sie mit `?befehl` oder `help(befehl)`. Sie können mit `help.search("Stichwort")` in der Hilfe suchen. Die online-Dokumentation starten Sie mit `help.start()` im Webbrowser.

Mit dem `savehistory`-Befehl können die eigenen Eingaben am Ende einer interaktiven R-Sitzung abgespeichert werden, z. B.

```
> savehistory("uebung1.r")
> q()
Save workspace image? [y/n/c]: n
```

3.1.2. Rechnen

Die Rechenoperatoren sind `+`, `-`, `*`, `/` für die vier Grundrechenarten und `^` für das Potenzieren. R hält sich an die üblichen Punkt-vor-Strich- und Klammerregeln. Das Dezimalzeichen ist der Punkt.

```
> 1+2
> 3.4*(5.2-0.1)
> 2^10
> 2^(-10)
```

3. R

```
> 2/3
```

Numerische Rechnungen werden mit doppelter Genauigkeit (**double**) durchgeführt (wie MATLAB). R verwendet **pi** für die Kreiszahl π . Die gängigen Funktionen werden über die Abkürzungen **log**, **asin** etc. aufgerufen (vergleichbar mit MATLAB, nicht wie in Maple).

```
> pi
> cos(pi)
> abs(-2)
> sqrt(2)
> factorial(3)
> choose(5,3)
```

Die Zuweisung eines Wertes an eine Variable erfolgt mit dem **<-** Operator.

```
> a <- 3
> a
> b <- a+4
> b
```

Ein Text wird in R durch Anführungszeichen **"Text"** gekennzeichnet und kann ebenfalls Variablen zugewiesen werden. Die bereits verwendeten Variablen (Objekte) können mit

```
> ls()
```

angezeigt werden. Ein Objekt wird mit **rm(objekt)** gelöscht, alle Objekte mit **rm(list=ls())**. Mit **options(digits=12)** können Sie die Ausgabegenauigkeit global beeinflussen, explizit mit **print(1/7, digits=9)**

3.1.3. Vektoren und Matrizen

Mit dem Befehl **c("a", "b")** wird ein Vektor mit den beiden Einträgen **"a"** und **"b"** erzeugt. **c(...)** steht für *concatenate*. Auf die Komponenten eines Vektors kann mit eckigen Klammern zugegriffen werden, entweder direkt mit Indizes oder mit logischen Ausdrücken.

```
> l <- c(7, 6, 5, 8, 3, 7)
> l
> length(l)
> l[2]
> l[2] <- 5
> l[c(2,4)]
```

3. R

Mit Vektoren können Bedingungen formuliert werden. Das Ergebnis sind sog. logische Vektoren mit Einträgen `TRUE` oder `FALSE`. Logische Vektoren können als Indizes verwendet werden und selektieren dann aus einem (anderen) Vektor die zugehörigen Elemente.

```
> 1>6
> 1[1>6]
```

Sequentielle Vektoren werden mit `:` erzeugt; auch rückwärts zählen ist möglich. Mit `seq()` kann flexibler vorgegangen werden, `rep()` repliziert einen Ausdruck (Details siehe `?seq` und `?rep`).

```
> 1:4
> 4:1
> seq(from=0, to=1, by=0.1)
> rep(2, times=5)
```

`+`, `-`, `*`, `/` wirken alle komponentenweise auf Vektoren (und auch Matrizen im Gegensatz zu Matlab). Mit `sum()` und `prod()` werden die Komponenten aufaddiert oder miteinander multipliziert. Es gibt auch mehrere spezielle Funktionen: `cumsum()` liefert die Partialsummen als Vektor, `diff()` liefert die sukzessiven Differenzen, also fast die Umkehrung von `cumsum()`. Summation eines logischen Vektors mit `sum` liefert die Anzahl der `TRUE`-Einträge.

```
> x <- 1:5
> 2*x + sin(x^2)
> v <- c(3,1,4,5,2,3,6,3)
> sum(v)
> w <- cumsum(v)
> diff(w)
> sum(w <= 15)
```

Matrizen sind Vektoren mit dem Dimensionsattribut `dim`. Sie können mit `matrix()` erzeugt werden.

```
> 1:12
> A <- matrix(1:12, nrow=3, ncol=4)
> A
> dim(A)
```

Der komponentenweise Zugriff erfolgt mittels eckiger Klammern. Es können auch so Spalten- und Zeilenvektoren selektiert werden. Mit `t()` berechnen Sie die Transponierte.

```
> A[2,2]
> A[1,]
> A[,2]
> t(A)
```

3. R

Die Funktion `c()` hängt Vektoren und Matrizen als Vektor aneinander, ohne Rücksicht auf deren Dimension. Die Funktionen `rbind()` (row bind) und `cbind()` (column bind) fügen passende Vektoren und Matrizen zusammen.

```
> A
> y <- 1:3
> y
> c(A,y)
> cbind(A,y)
> rbind(A,A)
```

3.1.4. Zufallsgrößen, Verteilungen und einfache Plots

R stellt statistische Tabellen bereit. Speziell die Verteilungsfunktionen, Dichten oder Wahrscheinlichkeiten der gängigen Verteilungen sind einfach erreichbar, z. B. die Dichte der Normalverteilung `dnorm` mit

```
> x <- seq(-4,4,0.1)
> y <- dnorm(x)
```

Mit der Funktion `plot()` können Sie Vektoren darstellen

```
> plot(x,y, type="l")
```

Die Komponenten der Vektoren `x` und `y` werden hier als Koordinaten von Punkten aufgefasst (siehe auch `?plot`).

Die Funktion `pnorm()` liefert die Verteilungsfunktion, `qnorm()` die Quantilfunktion¹ und `dnorm(x)` die Dichte der Standard-Normalverteilung. Auch für andere Wahrscheinlichkeitsverteilungen gibt es diese Funktionen mit vorangestelltem `p`, `d`, `q`, wie bei der Binomial- `binom`, χ^2 - `chisq`, Exponential- `exp`, Poisson- `pois`, Cauchy `cauchy` und der Gleichverteilung `unif`, um nur einige zu nennen.

```
> x <- seq(-0.2,1.2,0.01)
> plot(x,dunif(x), type="l")
> plot(x,punif(x), type="l")
```

Mit vorangestelltem `r` erhält man (Pseudo)-Zufallszahlen der gewünschten Verteilung. Beispielweise mit

```
> no <- rnorm(n=100, sd=4, mean=10)
> plot(no)
```

¹Sei F die Verteilungsfunktion, dann ist die Quantilfunktion als $F^{-1}(p) = \inf\{x \in \mathbb{R} | F(x) \geq p\}$ definiert.

3. R

werden 100 normalverteilte Zufallszahlen mit Mittelwert 10 und Standardabweichung 4 erzeugt (siehe `?rnorm`). In `rnorm(n=100, sd=4, mean=10)` übergeben wir die Parameter der Verteilung als *named arguments*. Wir erreichen das gleiche mit `rnorm(100, 10, 4)`, jetzt ist jedoch die Reihenfolge wesentlich.

3.2. Programmieren und Grafik

R selbst bringt unter Linux keinen Editor mit. Wir benötigen einen Editor, wie z.B. **kwrite**, um Skript-Dateien bearbeiten zu können. Ein Skript wie in der Datei: *skript.r*

```
# eine einfache Skriptdatei
x <- seq(0, 2*pi, 0.1)
y <- sin(x)
plot(x,y)
```

können Sie mit `source` laden und ausführen:

```
> source("skript.r")
```

3.2.1. Funktionen, Verzweigungen und Schleifen

R implementiert die Sprache S, daher ist R flexibel erweiterbar. Am einfachsten geschieht dies über Funktionen.

```
mysq <- function(x) {
  y <- x^2
  y + 1
}
```

```
> mysq(2)
> mysq(1:10)
```

Der letzte Wert innerhalb von `{}` ist der Rückgabewert, in unserem Fall x^2+1 selbst. Das Beenden einer Funktion inklusive Rückgabe kann auch explizit mit `return()` erfolgen. Verzweigungen erhalten Sie mit `if`

```
myf <- function(x) {
  if (x < 0) {
    return(x^2)
  } else {
    return(x)
  }
}
```

```
> myf(-3)
> myf(3)
```

3. R

Die Funktion `myf` hat jedoch den Nachteil, dass keine Vektoren für `x` eingesetzt werden können. Eine bessere Implementierung verwendet daher den `ifelse`-Befehl

```
myf2 <- function(x) {  
  ifelse(x < 0, x^2, x)  
}
```

```
> myf2(-3:3)
```

Die üblichen Relationen werden mit `==`, `<=`, `>=`, `!=` bezeichnet. Mit dem logischen Und `&&` und Oder `||` können Sie Ausdrücke verknüpfen. Es gibt `for`-Schleifen

```
for (i in 1:5) {  
  print(i)  
}
```

und `while`-Schleifen.

```
j <- 0  
while (j <= 5) {  
  print(j)  
  j <- j+1  
}
```

Wie in MATLAB führt die Verwendung expliziter Schleifen in R meist zu wenig effizientem Code. Oft liefert `sapply` das Gewünschte

```
> x <- 1:10  
> sapply(x, myf)
```

Wenn Sie eine Funktion n -mal auswerten wollen, können Sie `replicate` verwenden.

Funktionsdefinitionen und Skript können gemischt werden: *Datei: beispiel.r*

```
# Skalarprodukt  
sprod <- function(y1, y2) {  
  n1 <- length(y1)  
  n2 <- length(y2)  
  if (n1 != n2) { return("Fehler") }  
  s <- 0  
  for (i in 1:n1) {  
    s <- s + y1[i]*y2[i]  
  }  
  s  
}  
  
# eine Rechnung  
y1 <- 1:3  
y2 <- 4:6  
sprod(y1,y2)
```

Ausführen mit

```
> source("beispiel.r")
```

Weitere Informationen zum Programmieren finden Sie in den Handbüchern².

3.2.2. Grafik

Wir haben bereits einfache Anwendungen der `plot` Funktion kennengelernt. Es gibt eine Vielzahl von Gestaltungsmöglichkeiten, die entweder als Optionen an das Plot Kommando, oder explizit mit `par()` gesetzt werden (siehe `?plot`).

```
> x <- seq(0,2*pi,0.1); s <- sin(x)
> plot(x,s, col="blue", type="l", xlab="x", ylab="y", main="sin")
```

Text kann im Plot mit `text()` oder am Rand mit `mtext()` hinzugefügt werden. `plot` ist eine „high-level“ Grafikfunktion. Grafiken können auch durch das Hinzufügen von Punkten und Linien erweitert werden

```
> c <- cos(x)
> points(x,c, pch=23)
> s5 <- sin(5*x)
> lines(x,s5)
```

3D Plots von $(x, y) \mapsto f(x, y)$ erhalten Sie mit `persp()`. Die x und y -Werte werden als Vektoren angegeben, die z -Werte als Matrix. Für die x -Werte 1, 2 und die y -Werte 3, 4, 5 wird

$$\begin{pmatrix} f(1,3) & f(1,4) & f(1,5) \\ f(2,3) & f(2,4) & f(2,5) \end{pmatrix}$$

benötigt. Zum Erzeugen solch einer Matrix eignet sich `outer`, als verallgemeinertes äußeres Produkt. Siehe hierzu das Beispiel in *Datei: 3dplot.r*

```
x <- seq(-2*pi, 2*pi, length.out = 50)
y <- x
sinsin <- function(x,y) { sin(x)*sin(y) }
z <- outer(x, y, sinsin )
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
title(main = "Wobbel")
```

Ein Plot wird als pdf-Datei abgespeichert, indem zuerst eine Datei als `pdf()` Gerät geöffnet wird, dann wird das `plot()` Kommando abgesetzt und mit `dev.off()` wird die Datei geschrieben und geschlossen.

²Siehe auch John M. Chambers: Programming with Data: A Guide to the S Language

```

> pdf("bild.pdf")
> plot(x,dnorm(x))
> plot(x,dt(x,2))
> plot(x,dexp(x))
> dev.off()

```

3.3. Deskriptive Statistik

3.3.1. Listen und Data Frames

Eine Liste ist eine geordnete Ansammlung von Objekten. Sie wird mit `list()` erzeugt. Die einzelnen Objekte können mit `[[ ]]` oder ihrem Namen angesprochen werden.

```

> rm(list=ls())
> A <- matrix(1:12, nrow=4,ncol=3)
> y <- c(1, 2, 3, 4)
> L <- list(mat=A, vec=y, name="hallo")
> L
> L[1]
> L[[1]]
> L[[1]][2,3]
> L$mat
> L$name

```

Data Frames sind spezielle Listen der Klasse `data.frame`, die im wesentlichen nur Vektoren mit gleicher Dimension beinhalten können. Mit `attach()` können die einzelnen Vektoren besser zugänglich gemacht werden, mit `detach()` stellen Sie wieder den Normalzustand her.

```

> x <- 1:3
> y <- 4:6
> zweiv <- data.frame(vec1=x, vec2=y)
> zweiv$vec1
> zweiv$vec1[2]
> zweiv[[1]]
> attach(zweiv)
> vec2
> detach(zweiv)
> vec2

```

Mit `fix(zweiv)` wird ein (spartanischer) Editor³ für Data Frames aufgerufen (siehe auch `edit()`).

³Falls Sie unter Unix die `EDITOR` Shell-Variable gesetzt haben, wird dieser Editor verwendet.

3. R

Es gibt mehrere Funktionen, um Daten aus Dateien einzulesen. Die Datei `noten.txt` enthält Spaltenweise die Noten und Übungspunkte einer Anfängervorlesung. Die erste Zeile liefert die Bezeichnung.

```
Note Punkte
1.0 404
1.0 416
1.0 432
...
```

Der Inhalt dieser Datei kann in einen Data Frame eingelesen werden.

```
> vorlesung <- read.table("noten.txt", header=TRUE)
> names(vorlesung)
> vorlesung
```

Weitere Funktionen zum Lesen sind `read.csv()` und `read.delim()`.

3.3.2. Mittelwert etc

Eine wesentliche Aufgabe eines Statistikpakets ist die Berechnung gängiger Kennzahlen aus der beschreibenden Statistik. Für unser Notenbeispiel erhalten wir

```
> attach(vorlesung)
> mean(Note)
> mean(Punkte)
> sd(Punkte)
> median(Note)
> summary(vorlesung)
```

Das Minimum und Maximum erhalten Sie auch direkt mit `min()` bzw. `max()`. Hier noch die Varianz, die Standardabweichung und die Korrelationskoeffizienten.

```
> var(Note)
> sd(Note)
> var(Punkte)
> cor(Note, Punkte)
```

3.3.3. Histogramme etc.

Mit einem Histogramm können wir die Häufigkeitsverteilung der einzelnen Variablen visualisieren.

```
> names(vorlesung)
> hist(Note)
> hist(Punkte, xlab="Punkte in den Uebungen", breaks=20, col="blue")
```

3. R

Mit einem Scatterplot können wir das qualitative Vorurteil über den Zusammenhang zwischen Übungspunkten und Noten veranschaulichen. Hierzu später mehr.

```
> plot(vorlesung)
> plot(Punkte, Note, pch="*")
```

R bringt einige Beispieldatensätze mit, die Sie auch direkt verwenden können. Mit `help` und `names` erhalten Sie Informationen zu diesen Daten.

```
> data()
> faithful
> names(faithful)
> help(faithful)
```

Dieser Datensatz beinhaltet in `eruptions` die Dauer einer Eruption und `waiting` die Zeiten zwischen den Eruptionen von Old Faithful (http://de.wikipedia.org/wiki/Old_Faithful). Zuerst betrachten wir die Kennzahlen und das Histogramm.

```
> summary(faithful)
> attach(faithful)
> hist(waiting)
```

Sind wir an der Wahrscheinlichkeitsdichte der Wartezeit interessiert, so können wir mit einem Modell starten und dessen Parameter aus den Daten schätzen (mehr dazu später). Sie können auch Kern-Dichteschätzer ⁴ verwenden, um die Wahrscheinlichkeitsdichte direkt zu schätzen.

```
> hist( waiting, seq(40, 100, 2), probability=TRUE )
> lines( density(waiting, bw=2) )
```

3.4. Tests und Modelle

Oft ist nicht nur eine Aufbereitung der Daten gewünscht, sondern auch ein Vergleich mehrerer Beobachtungen oder ein Vergleich mit Modellen.

3.4.1. Vergleich zweier Experimente

Wir erzeugen uns zwei Datensätze A und B als Realisierungen normalverteilter Zufallszahlen mit Standardabweichung 1, aber verschiedenen Mittelwerten.

```
> rm(list=ls())

> A <- rnorm(50, mean=0, sd=1)
> B <- rnorm(50, mean=0.5, sd=1)
```

⁴Der Kern-Dichteschätzer der Bandbreite h ist durch $f_h(x) := \frac{1}{n} \sum_{i=1}^n K_h(x - x_i)$ für symmetrische K mit $\int_{\mathbb{R}} K_h(x) dx = 1$ gegeben.

3. R

Boxplot Ein grafischer Vergleich der beiden Datensätze ist mit einem Scatterplot oder einem Boxplot möglich.

```
> plot(A)
> points(B, pch="+")
> boxplot(A, B)
```

Der Strich in der Mitte markiert den Median, das Rechteck in der Mitte markiert den Bereich von Quartil 1 bis Quartil 3 (also 50% der Daten). Die äußeren Markierungen werden als Whisker bezeichnet und sind ein robustes Maß für Ausreißer (Details siehe <http://de.wikipedia.org/wiki/Boxplot>). Obiger Boxplot deutet auf einen Unterschied der beiden Datensätze hin.

Test Mit einem statistischen Test kann untersucht werden, ob sich diese beiden Datensätze *A* und *B* *signifikant* unterscheiden. Mit dem *t-Test* untersuchen wir, inwieweit die Verteilungen von *A* und *B* den gleichen Mittelwert haben. Die wesentlichen Annahmen für die Anwendung des t-Tests sind, dass die Daten in *A* und *B* unabhängige Realisierungen von näherungsweise normalverteilten Zufallsvariablen sind.

```
> t.test(A, B)
```

Welch Two Sample t-test

```
data: A and B
t = -2.242, df = 91.875, p-value = 0.02737
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.93984710 -0.05687677
sample estimates:
 mean of x mean of y
0.07425259 0.57261452
```

Der *p*-value, auch die Überschreitungswahrscheinlichkeit genannt, ist die Wahrscheinlichkeit, dass ein zufälliger Versuch bei gültiger Nullhypothese mindestens so extrem ausgeht wie der beobachtete Versuch. Die Nullhypothese wird abgelehnt, wenn $p \leq \alpha$, für ein vorgegebenes Signifikanzniveau α . Oft wird $\alpha = 0.05$ gewählt. In unserem Fall können wir also die Nullhypothese „*Beide Stichproben haben den gleichen Mittelwert*“ mit dem Signifikanzniveau $\alpha = 0.05$ ablehnen.

Der t-Test ist sicher einer der einfachsten Tests mit sehr restriktiven Voraussetzungen. R implementiert eine Vielzahl weiterer Tests, siehe `help.search("test")`.

3.4.2. Modelle und Regression

Wir laden wieder unsere Noten und Punkte in einen Data Frame.

```
> vorlesung <- read.table("noten.txt", header=TRUE)
> attach(vorlesung)
> plot(Punkte, Note, pch="*")
```

In diesem Scatterplot erkennen wir einen (naheliegenden) Trend. Als Modell nehmen wir einen linearen Zusammenhang zwischen den Noten $n = (n_1, \dots)$ und den erreichten Übungspunkten $u = (u_1, \dots)$ an:

$$n_i = a + bu_i + \epsilon_i.$$

Wobei die Fehler ϵ_i unabhängige und identisch normalverteilte Zufallszahlen seien. Unter diesen (durchaus restriktiven) Voraussetzungen können wir die Parameter a und b mit Methoden der Ausgleichsrechnung bestimmen. In R beschreibt die *Modell-Formel*

$$n \sim u$$

genau obigen Zusammenhang. D.h. der konstante Term a wird nicht angegeben. Solange die Parameter (a , b , etc.) linear eingehen, kann auch u nichtlinear in die Modell-Formel eingehen. Für unser Beispiel verwenden wir das Modell `Note ~ Punkte` und führen den Fit mit der Funktion `lm` (lineares Modell) durch.

```
> thefit <- lm(Note ~ Punkte)
> thefit
```

Um diese Ausgleichsgerade $n = a + bu$ in den Scatterplot einzutragen, extrahieren wir zuerst die Parameter und fügen dann einen Linienzug ein.

```
> attributes(thesfit)
> a <- thesfit$coefficients[1]
> b <- thesfit$coefficients[2]
> u <- 0:500
> n <- a + b*u
> plot(Punkte, Note, pch="*")
> lines(u,n)
```

Dies geht auch kürzer.

```
> plot(Punkte, Note, pch="*")
> abline(coef(thesfit))
```

Die Funktion `coef()` extrahiert die Modellparameter und `abline()` fügt dem Plot eine Gerade bei gegebenem Achsenabschnitt und Steigung hinzu.

3.5. Sonstiges

3.5.1. Packages

Es gibt sehr viele Pakete, die die statistischen und auch die Grafikfunktionalitäten von R erweitern (siehe <http://cran.r-project.org/>). Pakete werden mit `library(paketname)` geladen, mit `library()` listen Sie die installierten Pakete auf. Ist Ihr Wunschkpaket noch nicht installiert, so können Sie es mit `install.packages()` nachinstallieren.

Ein Beispiel: Plots mit Fehlerbalken können mit Hilfe der Funktion `plotCI()` aus dem Paket `gplots` erzeugt werden. Zuerst installieren wir das Paket mit

```
> install.packages("gplots")
```

Wenn Sie die Fragen mit `y` beantwortet und einen Mirror ausgewählt haben, wird das Paket in Ihrem Heimatverzeichnis unter `~/R/` installiert. Vor der Benutzung müssen Sie das Paket nun laden.

```
> library(gplots)
> N <- 10
> x <- 1:N
> y <- x + rnorm(N)
> delta <- rep(2, N)
> plotCI(x=x, y=y, uiw=delta)
```

A. Syntaxvergleich: MATLAB – Maple – R

	MATLAB	Maple	R
Start (Linux-Konsole)	matlab	xmaple	R
Schnellhilfe	help <i>Befehl</i>	? <i>Befehl</i>	? <i>Befehl</i>
Beenden	quit	quit	q()
Kommentar	% Kommentar	# Kommentar	# Kommentar
Zuweisung	a = 5	a := 5	a <- 5
Gleichheit	==	=	==
Ungleichheit	~=	<>	!=
Fakultät	factorial(n)	n!	factorial(n)
Binomialkoeffizient	nchoosek(n,k)	binomial(n,k)	choose(n,k)
Natürlicher Logarithmus	log	ln	log
Arcussinus	asin	arcsin	asin
Kreiszahl π	pi	Pi	pi
komplexe Zahlen	2+3*i	2+3*I	2+3i
Real-, Imaginärteil	real, imag	Re, Im	Re, Im
(Spalten-)Vektor eingeben	[1;2;3]	<1,2,3>	c(1,2,3)
Matrix eingeben	[1,2;3,4]	<<1 2>, <3 4>>	rbind(c(1,2),c(3,4))
Matrizelemente	A(i,j)	A[i,j]	A[i,j]
<i>i</i> -te Zeile	A(i,:)	Row(A,i)	A[i,]
<i>j</i> -te Spalte	A(:,j)	Column(A,j)	A[,j]
$m \times n$ -Nullmatrix	zeros(m,n)	ZeroMatrix(m,n)	matrix(0,m,n)
$n \times n$ -Einheitsmatrix	eye(n)	IdentityMatrix(n)	diag(n)
$m \times n$ -Matrix aus Einsen	ones(m,n)	Matrix(m,n,1)	matrix(1,m,n)
Transponierte	A'	Transpose(A)	t(A)
Matrixmultiplikation	A*B	A.B	A %*% B
Elementw. Multiplikation	A.*B	A~B (ab Vers. 13)	A*B
$Ax = b$ lösen	A\b	LinearSolve(A,b)	solve(A,b)
Einfache Funktion	f= @(x) x.^2	f:=x->x^2	f<-function(x) x^2
Funktion plotten	fplot(f,[0,1])	plot(f(x), x=0..1)	plot(f,0,1)
for-Schleife $i = 1, \dots, n$	for i=1:n end	for i from 1 to n do end do	for (i in 1:n){ } }
if – then – else	if i==n else end	if i=n then else end if	if (i==n){ } }else{ } }