

Programmierkurs MATLAB

Felix Fontein

Version Januar/Februar 2013, Alexander Veit

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel des Kurses	1
1.2	Wie funktioniert ein Computer?	1
1.3	Was ist eine Programmiersprache?	1
1.4	Algorithmen	2
1.5	Komplexität	2
1.6	MATLAB	3
1.7	Zum Ablauf des Kurses	3
2	Bedienung von MATLAB und erste Ausdrücke	4
2.1	Einleitung	4
2.2	Die Kommandozeile	4
2.3	Kurzüberblick über die graphische Benutzeroberfläche (GUI)	5
2.4	Die Online-Hilfe	5
2.5	Ausdrücke und Variablen	6
2.6	Matrizen	10
2.7	Operationen auf Matrizen	13
2.8	Exkurs: Fließkommazahlen	14
2.9	Native Datentypen	15
2.10	Der Workspace Browser	17
3	Skripte und Plots	19
3.1	Skripte in MATLAB	19
3.2	Zweidimensionale Plots	19
3.3	Dreidimensionale Plots	22
4	Kontrollkonstrukte und Schleifen	25
4.1	Kontrollkonstrukte	25
4.2	Schleifen	26
4.3	Benutzen des Debuggers	29
5	Funktionen und Skripte	32
5.1	Inline-Funktionen	32
5.2	Funktionen in <i>m</i> -Dateien	32
5.3	Gültigkeitsbereich von Variablen	34
5.4	Function Handles	35
6	Weitere Datenstrukturen	36
6.1	Mehrdimensionale Arrays	36
6.2	Cell Arrays	36
6.3	Structs	37
7	Konzepte	38
7.1	Rekursion	38
7.2	Backtracking	39
7.3	Dynamisches Programmieren	40
7.3.1	Fibonacci-Folge	40
7.3.2	Binomialkoeffizienten	41

7.3.3	Längste gemeinsame Teilsequenz	42
7.4	Vektorisierung	42
8	Toolboxen	43
8.1	Symbolic Math Toolbox	43

1 Einleitung

1.1 Ziel des Kurses

Neben einer kurzen Wiederholung allgemeiner Konzepte, die bereits in der Vorlesung “Einführung in die Programmierung” behandelt worden sind, soll das Arbeiten und Programmieren mit MATLAB erlernt werden. Das Ziel ist, dass die Teilnehmer nach diesem Kurs dazu in der Lage sind,

- selbstständig Algorithmen in MATLAB zu implementieren;
- die Arbeitsweise von MATLAB zu verstehen und Erweiterungen (Toolboxen) nutzen zu können sowie auch selber MATLAB erweitern zu können;
- MATLAB-Programme von anderen nachvollziehen zu können und zu verstehen.

1.2 Wie funktioniert ein Computer?

Ein Computer kann in vier Teile aufgeteilt werden:

1. CPU: verarbeitet Befehle;
2. Arbeitsspeicher: enthält Daten und Befehle;
3. Bus: transportiert Daten zwischen CPU, Speicher und der Peripherie;
4. Peripherie: alles, was an den Computer angeschlossen wird, wie etwa Geräte zur Ein- und Ausgabe oder Datenspeicherung.

Die Befehle, die von der CPU ausgeführt werden, liegen im Arbeitsspeicher in einem bestimmten Format vor, dem sogenannten Maschinencode. Dieser ist für Menschen nicht sehr leserlich.

1.3 Was ist eine Programmiersprache?

Eine Programmiersprache ist eine formale Sprache, die Programme beschreibt, jedoch im Gegensatz zu Maschinencode meist lesbar ist. Ein Programm in einer Programmiersprache muss erst von einem sogenannten Compiler (Übersetzer) in Maschinencode übersetzt werden, bevor das Programm auf dem Computer ausgeführt werden kann.

Es gibt verschiedene Abstraktionsebenen mit zugehörigen Programmiersprachen:

- Assembler ist etwa eine Programmiersprache, die ziemlich direkt Maschinencode entspricht, jedoch den Vorteil hat auch für Menschen (gut) lesbar zu sein.
- Low-Level-Sprachen wie C, in der es Befehle gibt, die vom Compiler in mehrere Maschinenbefehle übersetzt werden. Etwa wird aus der Anweisung

```
if (a == b) fprintf("sind gleich");,
```

die im Falle der Gleichheit zweier Variablen einen Text ausgibt, ein ganzer Satz von Maschinenbefehlen.

- High-Level-Sprachen wie Java oder MATLAB, die völlig von der unterliegenden CPU abstrahieren und erlauben, mit komplexen Objekten (etwa Matrizen) zu arbeiten, ohne sich um die Details (Speicherverwaltung, Operationen, ...) zu kümmern.

MATLAB hat einen sehr hohen Abstraktionsgrad.

Weiterhin gibt es verschiedene Programmierparadigmen, die wichtigsten sind *imperative Programmierung*, *funktionale Programmierung*, *logische Programmierung* sowie *objektorientierte Programmierung*. Imperative Programmierung entspricht der Arbeitsweise einer CPU: imperative Programme sind eine Ansammlung von Befehlen, die in einer gewissen Reihenfolge ausgeführt werden. Funktionale und logische Programmierung dagegen sind anders; dies soll hier weiter nicht diskutiert werden, bis auf den Hinweis, dass sich diese bei gewissen Problemen besser eignen als etwa imperative Programmierung. Das vierte angesprochene Paradigma, *objektorientierte Programmierung*, stellt im Gegensatz zur imperativen Programmierung nicht den Befehle in den Mittelpunkt, sondern *Objekte*. Die Programmiersprache Java, die in der Vorlesung “Einführung in die Programmierung” behandelt wird, erlaubt objektorientierte und imperative Programmierung.

In MATLAB kann sowohl imperativ wie auch objektorientiert programmiert werden. Wir werden objektorientierte Programmierung in diesem Kurs jedoch nicht behandeln.

1.4 Algorithmen

Ein Algorithmus ist im wesentlichen eine Sammlung von Befehlen, die Eingabedaten in einem vorher spezifizierten Format verarbeitet und Ausgabedaten in einem vorher spezifizierten Format zurückliefert. Ein sehr bekannter Algorithmus aus der linearen Algebra ist der Gaußsche Algorithmus: als Eingabe erwartet er eine $n \times m$ -Matrix mit Einträgen aus einem Körper, und er liefert eine $n \times m$ -Matrix mit Einträgen in ebendiesem Körper zurück, welche sich in Zeilenstufenform befindet. Wie dies erreicht wird, ist (zumindest in informeller Form) in jedem Skript zur Linearen Algebra zu finden: dort steht dann, welche Zeilenumformung wann ausgeführt werden muss, um schliesslich eine Matrix in Zeilenstufenform zu erhalten.

Es gibt verschiedene Art und Weisen, Algorithmen zu spezifizieren. Oft geschieht dies in Textform: etwa als Programm in einer Programmiersprache, oder als “Pseudo-Code”, also Text, der Ähnlichkeit mit bekannten Programmiersprachen aufweist. Andere Formen von Algorithmen sind etwa Flussablaufdiagramme. Siehe auch Abbildung 1.

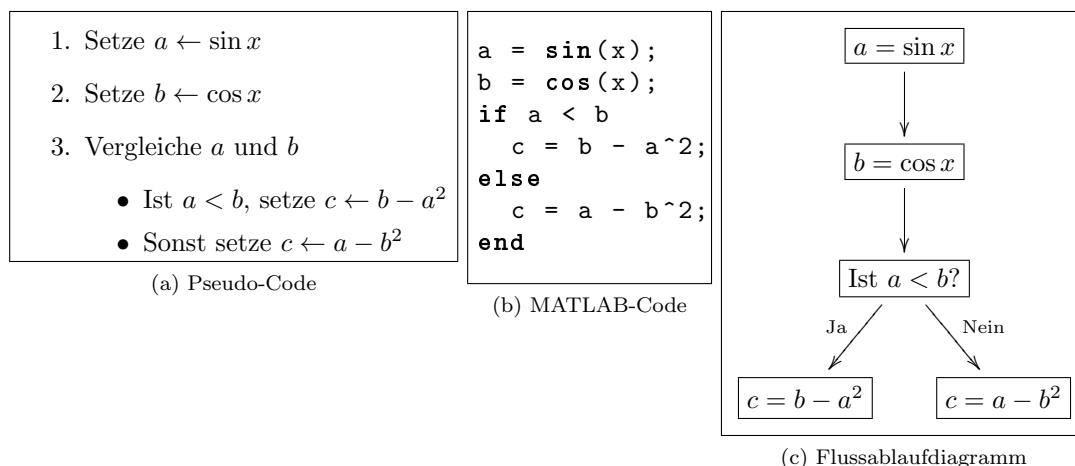


Abbildung 1: Der gleiche Algorithmus in verschiedenen Beschreibungen

1.5 Komplexität

Wenn man einen Algorithmus hat, interessiert einen oft, wie effizient dieser ist. Eine Möglichkeit, dies zu messen, ist die benötigte Laufzeit und den Speicherverbrauch in Relation zur Grösse der Eingabedaten anzugeben.

Man ist häufig nur an der *asymptotischen Laufzeit* interessiert: wenn die Laufzeit zweier Algorithmen sich nur um einen konstanten Faktor unterscheidet, haben sie dieselbe asymptotische Laufzeit. Um dies zu beschreiben, hat sich die *Landau-Notation*, auch *Gross-O-Notation*, eingebürgert. Sind $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$ Funktionen, so sagt man, dass f *von der Ordnung g ist*, in Zeichen $f = O(g)$, falls gilt

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

(In Worten: der Quotient f/g ist für $n \rightarrow \infty$ beschränkt.) Dies ist äquivalent dazu, dass es Konstanten $c > 0$ und $n_0 \in \mathbb{N}$ gibt mit $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

Wenn man etwa eine $n \times m$ -Matrix $A = (a_{ij})_{ij}$ mit einer $m \times \ell$ -Matrix $B = (b_{jk})_{jk}$ multipliziert, so muss man für jedes Paar (i, k) die Summe $c_{ik} = \sum_{j=1}^m a_{ij} b_{jk}$ berechnen. Es gibt $n \cdot k$ solche Paare. Zur Berechnung eines Eintrages c_{ik} werden nun $m - 1$ Additionen und m Multiplikationen benötigt, womit man insgesamt

$$nk(m - 1) \text{ Additionen und } nkm \text{ Multiplikationen}$$

benötigt. Insgesamt braucht man also $nk(2m - 1)$ arithmetische Operationen; und wenn man die Landau-Notation verwendet:

$$O(nkm) \text{ Operationen.}$$

Dies ist ein wenig besser lesbar als $nk(2m - 1)$; bei komplizierteren Algorithmen wären solche exakten Ausdrücke noch wesentlich komplizierter und der Vereinfachungseffekt der Landau-Notation ist um so stärker.

Hat man etwa eine Laufzeit, die von zwei Parametern n, m abhängt, und die durch den folgenden Ausdruck abgeschätzt wird:

$$35n^3m^2 + 75n^2m \cdot \log n \cdot (\log m)^5 + \frac{13n^5}{n^2 + 11} - \frac{1}{3}m^2 \cdot \log \log n$$

Mit der Landau-Notation lässt sich dies durch $O(n^3m^2)$ ausdrücken. Hat man für das gleiche Problem ein weiteres Programm mit der asymptotischen Laufzeit $O(n^2m^2)$, so kann man sofort sehen, dass das zweite Programm für grosse Werte von n schneller ist. Es kann jedoch sein, dass das erste Programm für kleine Werte von n schneller ist: die Landau-Notation sagt nur etwas über ausreichend grosse Argumente aus.

Es gibt etwa zum Multiplizieren von zwei $n \times n$ -Matrizen einen Algorithmus von Coppersmith und Winograd, der $O(n^{2.3727})$ Operationen benötigt – im Vergleich zu den $O(n^3)$ Operationen für den “naiven” Algorithmus. Wenn n sehr gross ist, ist dieser Algorithmus somit wesentlich schneller. Praktische Relevanz hat dieser Algorithmus jedoch nicht, da sein Geschwindigkeitsvorteil erst bei Matrizen sichtbar wird, die aufgrund ihrer Grösse von heutigen Computern nicht mehr verarbeitet werden können.

1.6 MATLAB

MATLAB steht für “MATrix LABoratory”. Es ist entstanden als eine Sammlung von (FORTRAN-) Programmen zur Manipulation von Matrizen. Es wurde später schliesslich zu einer eigenen Programmiersprache, mit der besonders einfach – und ohne sich um die konkrete Umsetzung der Befehle etwa in FORTRAN kümmern zu müssen – mit Matrizen gearbeitet werden kann.

Allgemein ist MATLAB gut geeignet für *numerische Anwendungen*, sowohl in der Forschung wie auch in der Industrie. MATLAB zu beherrschen lohnt sich also nicht nur für die Numerik-Vorlesung, sondern je nach angestrebten Werdegang für viele weitere Dinge.

1.7 Zum Ablauf des Kurses

Der Kurs besteht aus zehn Tagen mit je drei Stunden. Diese Stunden teilen sich auf in Vorlesungsanteile, in denen Konzepte eingeführt oder Anweisungen vorgestellt werden, und in Übungsteile, in denen die Teilnehmer die zuvor kennengelernten Konzepte und Anweisungen benutzen sollen.

Zum Abschluss des Kurses wird es Projekte geben. Jedem Teilnehmer wird ein Projektthema zugeordnet, welches dieser alleine bearbeiten soll. Das Bearbeiten kann an den letzten Kurstagen geschehen, oder in den ersten zwei Vorlesungswochen. Um die Kreditpunkte für den Kurs zu erhalten, müssen die Teilnehmer dem Kursleiter ihr bearbeitetes Projekt präsentieren und Fragen dazu beantworten können. Termine können direkt mit dem Kursleiter ausgemacht werden.

Die Projekte werden Ende der ersten Kurswoche zugewiesen.

2 Bedienung von MATLAB und erste Ausdrücke

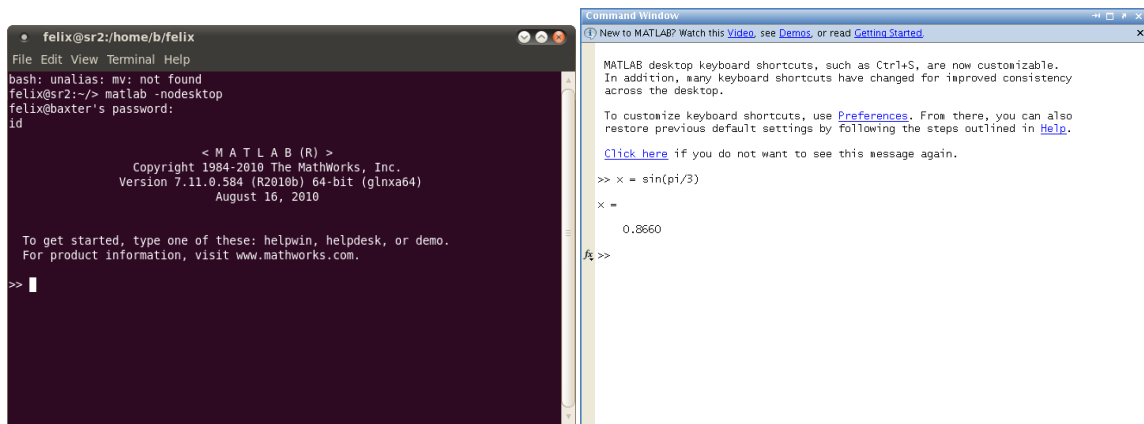
2.1 Einleitung

- MATLAB besteht aus zwei Teilen:
 - einer Benutzeroberfläche samt Editor (dort kann man Programme schreiben) und Debugger (Hilfsmittel zum Fehler finden);
 - einer Kommandozeile, in der etwas ausprobiert werden kann, im Editor geschriebene Funktionen aufgerufen werden können, oder von der aus Skripte (Programme) gestartet werden können.
- Die Kommandozeile ist auch ohne Benutzeroberfläche nutzbar; wenn man sich etwa per SSH einloggt und MATLAB startet, bekommt man nur diese.
- Alternativ zu MATLAB kann auch Octave verwendet werden; dies ist ein Open-Source-Programm, welches grosse Teile von MATLAB selber nachbildet. Für fast alles¹ in diesem Kurs reicht Octave völlig aus.
 - Octave kann man etwa unter

<http://www.gnu.org/software/octave/>

finden.
 - Octave selber hat nur eine Kommandozeile.
 - Es gibt jedoch separate GUIs, wie etwa qt octave.
 - Octave/qt octave bieten sich an, wenn man Zuhause programmieren will, ohne sich per SGD oder SSH auf den Uni-Rechnern einzuloggen (oder weil alle Uni-Lizenzen belegt sind).

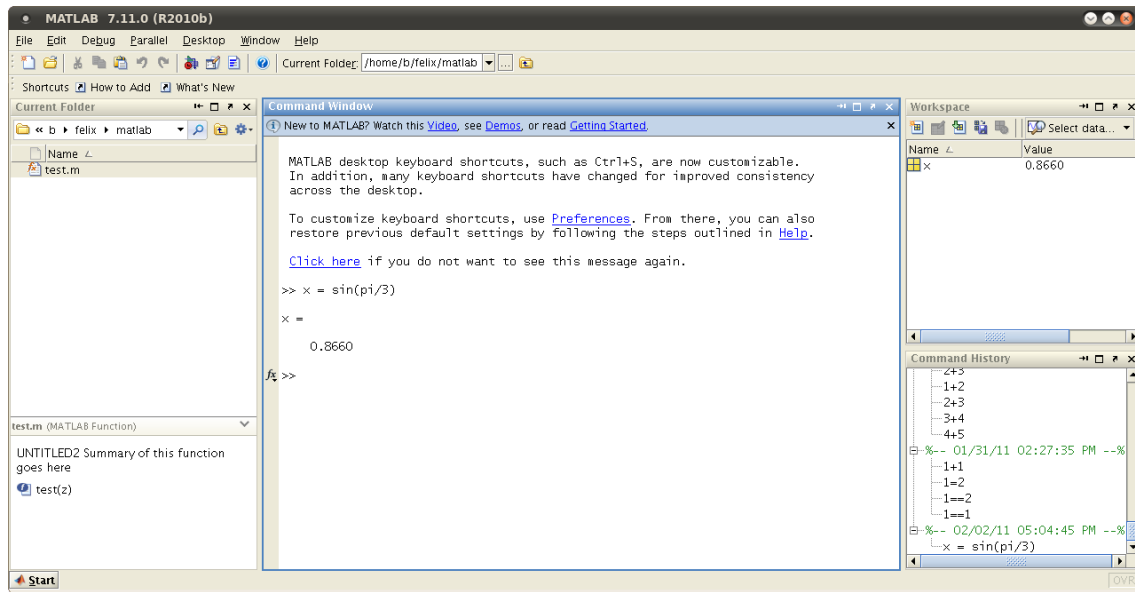
2.2 Die Kommandozeile



- Man kann Ausdrücke (etwa $1 + 1$, $\pi^2/6$), Anweisungen (etwa `fprintf('test\n')`) und Definitionen eingeben
- Diese werden sofort ausgeführt, bzw. bei grösseren Konstrukten (Verzweigungen, Funktionsdefinitionen, Schleifen, ...) wird solange gewartet, bis das Konstrukt vollständig eingegeben ist, bevor es ausgeführt wird
- Eine Berechnung kann abgebrochen werden, falls sie zulange dauert: einfach **Ctrl+C** drücken (**Strg+C** auf deutschen Tastaturen).

¹Der einzige Teil des Kurses, der sich in Octave (noch) nicht machen lässt, ist der Abschnitt über Toolboxes, und insbesondere die symbolische Toolbox.

2.3 Kurzüberblick über die graphische Benutzeroberfläche (GUI)



- Ganz oben findet sich das Menü (File, Edit, Debug, ...) und die Toolbar, in der sich Icons für neue Skripte, zum Öffnen von Skripten, Bearbeiten von Text, ... befinden.
- Darunter befinden sich "Shortcuts", mit denen man direkt vorher festgelegte Sequenzen von MATLAB-Befehlen ausführen kann, sozusagen Makros. Dies können auch Aufrufe der Online-Hilfe sein. Der erste Shortcut "How to add..." beschreibt, wie man Shortcuts hinzufügen kann.
- Darunter ganz links befindet sich die Dateiauswahl. Hier können .m-Dateien (dazu später mehr) ausgewählt werden, per Doppelklick im Editor geöffnet werden, und in der Schnellan-sicht darunter wird angezeigt, welche Funktion(en) sich in der Datei befinden.
- Mittig befindet sich das "Command Window", die Kommandozeile, in der direkt MATLAB-Befehle eingegeben werden können. Befehle müssen hinter dem aktuellen Prompt >> eingegeben werden. Der Eingabecursor befindet sich hinter dem aktuellen Prompt.
- In der Spalte rechts befinden sich oben der Workspace Browser (dazu später mehr).
- Rechts unten findet sich die "Command History", in der alle in der Kommandozeile ausgeführten MATLAB-Befehle gesammelt werden. Diese lassen sich etwa per Doppelklick nochmal ausführen. Die Befehle sind jeweils nach der Sitzung sortiert.

2.4 Die Online-Hilfe

MATLAB hat eine mächtige und hilfreiche Online-Hilfe. Diese kann mit dem Befehl `doc` in der Kommandozeile aufgerufen werden. Eine Nur-Text-Variante, die direkt in der Kommandozeile angezeigt wird, ist der `help`-Befehl. Indem man nur `doc` eingibt, erscheint das Hilfe-Fenster. Wenn man Hilfe zu einem bestimmten Befehl haben möchte, gibt man in der Kommandozeile `help Befehl` oder `doc Befehl` ein.

Beispiele:

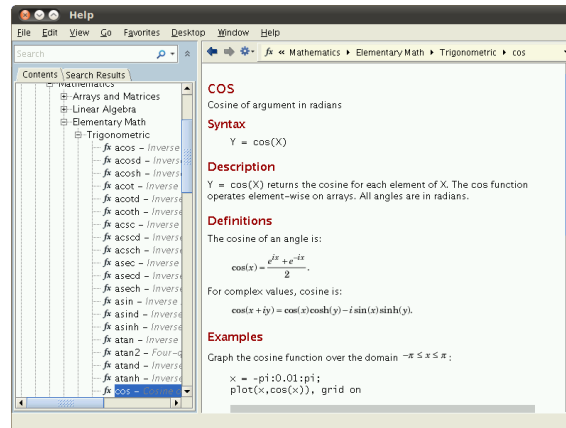
```
>> help sin
SIN      Sine of argument in radians.
        SIN(X) is the sine of the elements of X.

See also asin, sind.

Overloaded methods:
        codistributed/sin

Reference page in Help browser
        doc sin
>> doc cos
```

Beim doc-Befehl erscheint in etwa folgendes Fenster:



2.5 Ausdrücke und Variablen

Ausdrücke sind im Wesentlichen Formeln, die MATLAB versteht. Es ist wichtig, dass man den Aufbau von Ausdrücken kennt und versteht, sowie auch deren Interpretation durch MATLAB. In vielerlei Beziehung verhält sich MATLAB, wie man es erwartet.

Ein weiteres Konzept sind Variablen. Diese speichern einen Wert. Man kann ihnen einen Wert zuweisen und den Wert verwenden. Der Wert einer Variable ist derjenige, der als letztes in ihr gespeichert wurde.

Wir wollen zuerst die Grundstruktur von Ausdrücken anschauen. Was ein Ausdruck ist kann man einerseits – da es ein sehr vertrautes Konzept ist – anhand von Beispielen “definieren”. Ein einfacher Ausdruck ist $3 * (2 + 1)$: dieser hat den Wert 9. Andererseits kann man auch formal definieren, was ein Ausdruck sein soll. Dies macht man am einfachsten rekursiv:

- Sind A und B Ausdrücke, so auch
 - (A) (Klammerung),
 - ~A (logische Negation),
 - -A (Negation),
 - A^B (Exponentiation),
 - A * B (Multiplikation), A / B (Division) und A \ B (“Linksdivison”, ist gleich B / A),
 - A + B (Addition) und A - B (Subtraktion),
 - ==, ~=, <, <=, > und >= (Test auf Gleichheit, Ungleichheit, Kleiner, Kleiner gleich, ...),
 - A & B (logisches Und),
 - A | B (logisches Oder).

Die Operatorpräzedenz ist absteigend aufgelistet (“Punkt-vor-Strich-Rechnung”):

$$1 + 2 \& 3 * 4 + 5 \quad \text{ist gleich} \quad (1 + 2) \& ((3 * 4) + 5);$$

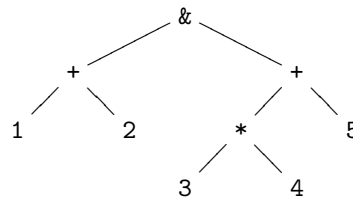
bei Operatoren gleicher Ordnung wird von Links nach Rechts interpretiert ($1 - 2 + 3$ ist gleich $(1 - 2) + 3$).

Die Vergleichsoperatoren und die logischen Operatoren geben den Wert 0 oder 1 zurück. Dabei bedeutet 0 **false** und 1 **true**. Wendet man die logischen Operatoren auf beliebige Zahlen an, so wird 0 als **false** und alles andere als **true** interpretiert.

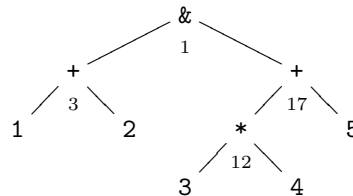
Die Symbole **true** und **false** sind in MATLAB nicht definiert.

Eine gute Möglichkeit, die Struktur eines Ausdruckes zu visualisieren, sind *Bäume*. Ein Knoten des Baumes repräsentiert ein zwischendurch auftretenden Wert bzw. Ausdruck, und die Kinder eines Knotens sind die zugehörigen Teilausdrücke bzw. die Zwischenergebnisse, aus

denen der Wert des aktuellen Knotens berechnet wird. Der Baum zum obigen Ausdruck $1 + 2 \& 3 * 4 + 5$ ist z.B.



Um den Wert eines Ausdrucks zu bestimmen, kann man jeweils die Zwischenergebnisse an die Knoten schreiben:



Aufgabe 2.1: Bestimme für die folgenden Paare $(\text{op}_1, \text{op}_2)$ von Operatoren, ob es ganzzahlige Konstanten $(a_1, a_2, a_3) \in \mathbb{Z}^3$ gibt so, dass $(a_1 \text{ op}_1 a_2) \text{ op}_2 a_3$ einen anderen Wert annimmt als $a_1 \text{ op}_1 (a_2 \text{ op}_2 a_3)$. Wenn es solche Konstanten gibt, gebe solche explizit an und überprüfe, wie MATLAB die Ausdrücke auswertet.

- | | | |
|--|--|--|
| 1. $(\text{op}_1, \text{op}_2) = (-, -);$ | 5. $(\text{op}_1, \text{op}_2) = (+,);$ | 9. $(\text{op}_1, \text{op}_2) = (\wedge, *);$ |
| 2. $(\text{op}_1, \text{op}_2) = (*, +);$ | 6. $(\text{op}_1, \text{op}_2) = (\&,);$ | 10. $(\text{op}_1, \text{op}_2) = (*, ==);$ |
| 3. $(\text{op}_1, \text{op}_2) = (+, \&);$ | 7. $(\text{op}_1, \text{op}_2) = (, \&);$ | 11. $(\text{op}_1, \text{op}_2) = (<, >);$ |
| 4. $(\text{op}_1, \text{op}_2) = (-, +);$ | 8. $(\text{op}_1, \text{op}_2) = (<,);$ | 12. $(\text{op}_1, \text{op}_2) = (\&, >=);$ |

Aufgabe 2.2: Füge maximal viele Klammern ein (ohne doppelte Klammern!), ohne die Abfolge der Operationen zu verändern, und bestimme den zugehörigen Baum und damit den Wert des Ausdrucks. Verifiziere dies, indem du den Ausdruck in MATLAB eingibst.

1. $2 + 4 * - 1 / 3 / 4$
2. $2 + 4 == 5 \mid 3 + 5 \wedge - 1 \& 2$
3. $- 3 < 4 < 5$
4. $4 < - 2 < 5$
5. $- 3 < - 2 < - 1$
6. `sin (n + 1) / (n + 2)`

- Sind A_1, \dots, A_n Ausdrücke und ist f eine Funktion, die auf n Parameter anwendbar ist, so ist $f(A_1, \dots, A_n)$ ebenfalls ein Ausdruck.

Beispiele für Funktionen:

- `sin(x)` (Sinus; Argument in Bogenmass);
- `cos(x)` (Kosinus; Argument in Bogenmass);
- `tan(x)` (Tangens; Argument in Bogenmass);
- `exp(x)` (Exponential von x);
- `log(x)` (natürlicher Logarithmus von x);
- `log2(x)`, `log10(x)` (Logarithmus zur Basis 2 bzw. 10 von x);
- `real(x)`, `imag(x)` (Realteil bzw. Imaginärteil von x).

Diese Funktionen verlangen jeweils nur einen Parameter. Wir werden jedoch später sehen, wie wir selber Funktionen schreiben können, die auch mehr als einen Parameter erwarten.

Weitere können mit **help elfun** (elementare Funktionen), **help specfun** (speziellere Funktionen) und **help elmat** (Matrix-Funktionen) aufgelistet werden.

- Weitere Ausdrücke sind (numerische) Konstanten und Variablen:
 - 1 ist eine Konstante (vom Typ **double**; dazu später mehr)
 - -123.45e67 ist ebenfalls eine Konstante vom Typ **double** mit dem Wert $-123.45 \cdot 10^{67} = -1.2345 \cdot 10^{69}$
 - .123 ist eine Konstante mit dem Wert $0.123 = 1.23 \cdot 10^{-1}$; alternativ kann man auch 123e-3 oder 1.23e-1 schreiben
 - 3i ist eine komplexe Zahl mit Realteil 0 und Imaginärteil 3. Anstelle i kann man auch j schreiben.

Aufgabe 2.3:

1. Überlege, wie man eine komplexe Zahl von Polarkoordinaten (Winkel in Bogenmass und Radius) in kartesische Koordinaten (Real- und Imaginärteil) schreiben kann.
2. Berechne das multiplikativ Inverse von $1 + 2i$ mit MATLAB.

- Variablen sind sozusagen mit Namen versehene Speicherplätze. Im Gegensatz zur üblichen Vorgehensweise in der Mathematik, wo man einem Symbol (Name) einmal einen Wert zuweist und diesen normalerweise nicht ändert,² kann man den Wert von Variablen in Programmen beliebig oft ändern, und dies ist sogar erwünscht und führt erst zu der Flexibilität, die zur Lösung vieler Probleme benötigt wird.

- * Namen von Variablen können (wie in Java) aus Buchstaben, Ziffern und Unterstrichen (_) bestehen, das erste Zeichen darf jedoch keine Ziffer sein.
- * Es wird zwischen Gross- und Kleinschreibung unterschieden.
- * Variablen kann man einem Wert zuweisen (**x = 1** – dies ist eine Anweisung, die der Variablen **x** den Wert 1 zuweist) oder als Ausdruck verwenden (**x** – dies ist ein Ausdruck, dessen Wert gleich dem Wert der Variable ist). Beispiele:
 - **x = sin(pi/4)** – speichert den Wert $\sin \frac{\pi}{4}$ in der Variablen **x**;
 - **asin(x)/pi** – berechnet den Arkussinus vom Wert der Variablen **x**, und teilt das Ergebnis durch π .

- Es gibt einige vordefinierte Variablen (deren Wert man jedoch ändern kann!):

- * **pi**: Approximation der Konstanten π ;
- * **i** oder **j**: die komplexe Einheit i , die $i^2 = -1$ erfüllt;
- * **eps**: die relative Fließkommagenauigkeit ε , meist 2^{-52} (dazu unten mehr);
- * **realmin**: die kleinste positive Fließkommazahl (meist 2^{-1022});
- * **realmax**: die grösste positive Fließkommazahl (meist $(2 - \varepsilon)2^{1023}$);
- * **Inf**: das positiv Unendliche, $+\infty$ (dies repräsentiert ein Ergebnis grösser als **realmax**, etwa das Ergebnis von $1/0$ oder von **exp**(2^{1000})); es gibt auch **-Inf**, etwa als Ergebnis von $-1/0$;
- * **NaN**: “Not a Number” (ein nicht wohl-definiertes Ergebnis; etwa das Ergebnis von $0/0$); im Gegensatz zu **Inf** ist **NaN** vorzeichenlos.

Diese Variablen sollten *unter keinen Umständen* umdefiniert werden! Das Umdefinieren kann auch sehr komische Effekte haben:

²Falls die Variable mehr als einen Wert haben soll, verwendet man meist eine Folge: der Startwert ist x_1 , dann erhält man x_2 , dann x_3 , etc. Bei der Übertragung von Algorithmen aus mathematischen Beschreibungen in eine Programmiersprache wie MATLAB muss man dies meist rückgängig machen, d.h. eine Folge x_1, x_2, x_3, \dots ersetzen durch eine Variable für x_n , wobei n sich im Laufe des Programmes vergrössert (und eventuell eine Variable für x_{n-1} , falls dieser Wert zusammen mit x_n verwendet wird, etc.).

```

>> Inf = 1
    Inf = 1
>> 1/0
    ans = Inf
>> ans == Inf
    ans = 0

```

oder

```

>> i = -1
    i = -1
>> 3i
    ans = 0 + 3i
>> 3*i
    ans = -3

```

- Befehle in MATLAB sind (nach bisherigem Stand) von folgenden möglichen Formen:

1. Variable = Ausdruck
2. Ausdruck

Steht hinter einem Befehl ein Semikolon “;”, so wird das Ergebnis des Befehls nicht angezeigt, andernfalls schon. Das Ergebnis ist im ersten Fall gleich dem Wert der Variable, der etwas zugewiesen wird; im zweiten Fall ist es der Wert des Ausdrucks, und dieser wird in der Variablen **ans** gespeichert.

Beispiel:

```

>> 7;
>> ans
    ans = 7
>> x = 8
    x = 8
>> ans + 2
    ans = 9
>> ans + 1
    ans = 10

```

Aufgabe 2.4: Gebe bei folgender Abfolge von Befehlen nach jedem Befehl an, welchen Wert **ans** gerade hat:

```

>> 23;
>> ans^2;
>> x = sqrt(ans);
>> x^2 - ans;
>> x^2 - ans;
>> ans - x^2;

```

Hinweis: mit den bisherigen Mitteln sind die folgenden Aufgaben relativ mühsam zu lösen. Wir werden diese Aufgaben später in ähnlicher Form nochmal bearbeiten, wenn bessere Hilfsmittel zur Verfügung stehen.

Aufgabe 2.5: Verifiziere numerisch, dass $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$ gilt, indem du die Partialsumme $\sum_{k=1}^n \frac{1}{k^2}$ mit MATLAB für $n = 10$ und $n = 15$ auswertest und $\frac{\pi^2}{6}$ vergleichst.

Aufgabe 2.6: Sei $f : \mathbb{R} \rightarrow \mathbb{R}$ eine differenzierbare Funktion. Das *Newton-Verfahren* zum Nullstellenfinden funktioniert so: man wählt einen Startpunkt x_0 , möglichst nahe einer Nullstelle, und definiert iterativ eine Folge $x_{n+1} := x_n - \frac{f(x_n)}{f'(x_n)}$. Wenn die Bedingungen günstig sind, gilt $f(\lim_{n \rightarrow \infty} x_n) = 0$. (Dazu mehr in der Numerik-Vorlesung!)

Führe die Newton-Iteration bis zur Berechnung von x_5 explizit für folgende Funktionen und Startwerte durch:

1. $f(x) = \sin x$, $x_0 = 3$;
2. $f(x) = x^5 - 14x^4 + x^2 - 3x + 5$, $x_0 = 5$.

Versuche, das ganze so zu schreiben, dass du möglichst einfach viele Iterationen durchführen kannst. Falls du dies schaffst, probiere, wie lange du iterieren musst, um bei 1. ein x_n zu finden mit $|f(x_n)| < 10^{-10}$.

Aufgabe 2.7: Eine weitere numerische Methode, eine Nullstelle einer Funktion zu finden, ist das Bisektionsverfahren. Man fängt mit zwei Startwerten x_1 und x_2 an, so dass $f(x_1)f(x_2) < 0$ ist. Man wählt $x_3 = \frac{x_1+x_2}{2}$. Dann gibt es drei Fälle:

- $f(x_1)f(x_3) < 0$: in dem Fall fährt man mit den Startwerten (x_1, x_3) fort;
- $f(x_1)f(x_3) > 0$: in dem Fall fährt man mit den Startwerten (x_3, x_2) fort;
- $f(x_3) = 0$: in dem Fall ist x_3 eine Nullstelle.

Nach dem Prinzip der Intervallschachtelung nähert man sich immer weiter eine Nullstelle von f an. (Der Beweis des Zwischenwertsatz funktioniert über dieses Prinzip.)

Führe dieses Verfahren explizit für $f(x) = \cos x$, $x_1 = 1$ und $x_2 = 2$ durch, bis $|x_2 - x_1| \leq 2^{-5}$ ist.

2.6 Matrizen

MATLAB basiert auf den Grundsatz

“Alles ist eine Matrix”:

- eine Zahl ist eine 1×1 -Matrix;
- ein Zeilenvektor ist eine $1 \times n$ -Matrix, ein Spaltenvektor eine $n \times 1$ -Matrix;
- eine Zeichenkette der Länge n ist eine $1 \times n$ -Matrix (bestehend aus einzelnen Zeichen).

Matrizen können einfach erzeugt werden:

- Sind A_1, \dots, A_n Matrizen mit gleicher Zeilenanzahl, so ist $[A_1, \dots, A_n]$ eine Matrix, in der die Matrizen hintereinander geschrieben werden.
 - Anstelle eines Kommas kann auch ein Leerzeichen benutzt werden.
 - Leerzeichen kann gefährlich sein: $[1 \ -2] = [-1]$ vs. $[1, -2]$.
- Sind A_1, \dots, A_n Matrizen mit gleicher Spaltenanzahl, so ist $[A_1; \dots; A_n]$ eine Matrix, in der die Matrizen untereinander geschrieben werden.
- “,” und “;” können natürlich kombiniert werden, um eine Blockmatrix zusammenzusetzen (solange die Matrizen das passende Format haben).

Beispiele:

- $[1, 2 \ 3]$ erzeugt einen Zeilenvektor $(1, 2, 3)$ (beachte, dass Zahlen 1×1 -Matrizen sind);
- $[1, 2; 3, 4]$ erzeugt die 2×2 -Matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$;
- ist $A = [1, 2; 3, 4]$, so ist $[A \ A \ A; A \ A \ A]$ gleich der Blockmatrix

$$\begin{pmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{pmatrix}.$$

- $\mathbf{a} : \mathbf{b}$ erzeugt einen Zeilenvektor mit den Einträgen $\mathbf{a}, \mathbf{a}+1, \mathbf{a}+2, \dots$; der letzte Eintrag ist $\leq \mathbf{b}$.
- $\mathbf{a} : \mathbf{b} : \mathbf{c}$ erzeugt einen Zeilenvektor mit den Einträgen $\mathbf{a}, \mathbf{a}+\mathbf{b}, \mathbf{a}+2\mathbf{b}, \dots$; der letzte Eintrag ist $\leq \mathbf{c}$ (falls $\mathbf{b} < 0: \geq \mathbf{c}$). Ist $\mathbf{b} = 0$, so wird die leere Matrix erzeugt.

Beispiele:

- $1 : \mathbf{pi}$ erzeugt den Zeilenvektor $(1, 2, 3)$;
- $3 : 0.03 : \mathbf{pi}$ erzeugt den Zeilenvektor

$$(3, 3.03, 3.06, 3.09, 3.12);$$

- `pi : -0.3 : 3` erzeugt den Zeilenvektor

(3.1416, 3.1116, 3.0816, 3.0516, 3.0216).

- Mit `[]` wird die leere Matrix (vom Format 0×0) erzeugt.
- Mit den Funktionen **eye**, **zeros**, **ones**, **rand** und **randn** lassen sich Matrizen erzeugen:
 - wird ein Argument n angegeben, so wird eine $n \times n$ -Matrix erzeugt;
 - (ausser **eye**) werden zwei oder mehr Argumente k_1, \dots, k_n angegeben mit $n \geq 2$, so wird eine $k_1 \times \dots \times k_n$ -Matrix (n -dimensional) erzeugt;
 - dabei ist **eye** eine Einheitsmatrix, **ones** eine Matrix mit allen Einträgen gleich 1, **zeros** eine Matrix mit allen Einträgen gleich 0, **rand** eine Matrix mit Einträgen aus dem Intervall $[0, 1]$, die mit einer Gleichverteilung zufällig gewählt werden, und **randn** eine Matrix mit Einträgen, die mit einer Standardnormalverteilung zufällig gewählt werden.
- Ist \mathbf{v} ein (Zeilen- oder Spalten-)Vektor, so ist **diag**(\mathbf{v}) die Diagonalmatrix mit den Einträgen aus \mathbf{v} auf der Diagonalen.
- Ist \mathbf{A} eine Matrix, so ist \mathbf{A}' die Transponierte von \mathbf{A} , und \mathbf{A}' die komplex konjugierte Transponierte von \mathbf{A} (macht nur einen Unterschied bei Matrizen mit echt komplexen Einträgen).

Aufgabe 2.8: Konstruiere folgende Matrizen mit möglichst kurzen und wenigen MATLAB-Befehlen:

1.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

2.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 3 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 4 & 0 \\ 2 & 2 & 2 & 2 & 0 & 0 & 5 \\ 2 & 2 & 2 & 2 & 6 & 6 & 7 \\ 2 & 2 & 2 & 2 & 6 & 6 & 8 \end{pmatrix}$$

3.

$$\begin{pmatrix} 5 & 5 & 1 & 0 & 0 & 0 & 5 & 5 & 6 & 7 & 8 \\ 6 & 6 & 0 & 2 & 0 & 0 & 6 & 1 & 0 & 3 & 0 \\ 7 & 7 & 0 & 0 & 3 & 0 & 7 & 0 & 2 & 0 & 4 \\ 8 & 8 & 0 & 0 & 0 & 4 & 8 & 5 & 6 & 7 & 8 \end{pmatrix}$$

Hinweis: der Befehl **ones**(...) erzeugt eine Matrix, die nur aus Einsen besteht. Um eine Matrix zu erzeugen, die nur aus Vieren besteht, muss man z.B. `4 * ones(...)` schreiben. (Dazu später mehr.)

Es gibt viele Möglichkeiten, auf Inhalte von Matrizen zuzugreifen:

- Ist \mathbf{A} eine Matrix, so kann man an die (k, ℓ) -te Stelle zugreifen per `A(k, 1)`.
- Ist \mathbf{A} ein (Zeilen- oder Spalten-)Vektor, so liefert `A(k)` den k -ten Eintrag von \mathbf{A} zurück. Ist \mathbf{A} dagegen eine Matrix, so liefert `A(k)` den k -ten Eintrag des Vektors

$$[\mathbf{A}_{11}, \dots, \mathbf{A}_{n,1}, \mathbf{A}_{12}, \dots, \mathbf{A}_{n,2}, \dots, \mathbf{A}_{1,m}, \dots, \mathbf{A}_{n,m}],$$

falls \mathbf{A} eine $n \times m$ -Matrix ist; in diesem Vektor sind die Einträge der Matrix spaltenweise aufgelistet.

- Die Funktion **sum(A)** liefert einen Zeilenvektor zurück, dessen k -ter Eintrag die Summe der Einträge in der k -ten Spalte von **A** ist. Falls **A** ein (Zeilen- oder Spalten-)Vektor ist, ist **sum(A)** einfach die Summe aller Einträge von **A**.
- Die Funktion **diag(A)** liefert die Diagonaleinträge einer Matrix **A** als Spaltenvektor zurück. Mit **diag(A, k)** kann die k -te (Neben-)Diagonale ausgelesen werden. Ist etwa $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, so ist **diag(A)** gleich $[1, 4]$, **diag(A, 1)** ist 2 und **diag(A, -1)** ist 3. Und insbesondere gilt für jeden Spaltenvektor **v**, dass **diag(diag(v, k), k)** wieder **v** ist.

Quizfrage: Wie kann man **v** zurückerhalten, wenn **v** ein Zeilenvektor ist?

- Es kann auch auf Teilmatrizen zugegriffen werden. In **A(k, 1)** kann **k** oder 1 ersetzt werden durch eins der folgenden:
 - durch **:**, dann wird die komplette Zeile bzw. Spalte zurückgeliefert: so ist **A(1, :)** die erste Zeile von **A** und **A(:, 1)** die erste Spalte von **A**;
 - durch einen Zeilen- oder Spaltenvektor **v**. Ist $\mathbf{v} = [v_1, \dots, v_n]$, so liefert **A(v, :)** die Teilmatrix zurück, welche n Zeilen hat und dessen erste Zeile die v_1 -te Zeile von **A** ist, dessen zweite Zeile die v_2 -te Zeile von **A** ist, etc.
 - * Ist **A** eine 3×3 -Matrix mit den Einträgen a_{ij} , so liefert **A([1 3], [1 2])** die Teilmatrix $\begin{pmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{pmatrix}$ zurück.

Mit diesen Zugriffstechniken auf Teilmatrizen und einfachen Zuweisungen lassen sich etwa Zeilen- und Spaltenumformungen durchführen. Um etwa viermal die dritte Spalte von **A** zu der ersten Spalte von **A** zu addieren, schreibt man einfach

$$\mathbf{A}(:, 1) = \mathbf{A}(:, 1) + 4 * \mathbf{A}(:, 3).$$

Und um die zweite Zeile von **A** mit der vierten zu tauschen, schreibt man

$$\mathbf{A}([2 \ 3], :) = \mathbf{A}([3 \ 2], :).$$

Weiterhin kann man etwa die fünfte und siebte Zeile von **A** auf 0 setzen, indem man

$$\mathbf{A}([5 \ 7], :) = \mathbf{zeros}(2, 10)$$

schreibt, falls **A** 10 Spalten hat.

Aufgabe 2.9:

1. Beschreibe, wie mit MATLAB das arithmetische Mittel aller Einträge einer Matrix **A** mit **n** Zeilen und **m** Spalten berechnet werden kann.
2. Untersuche den Mittelwert von Matrizen, die mit **rand** und **randn** erzeugt wurden. Fällt an den Mittelwerten etwas auf, wenn die Matrizen gross sind (also viele Zeilen und Spalten haben)?
3. Überführe die Matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{pmatrix}$$

mit elementaren Zeilenumformungen in MATLAB in Zeilenstufenform. (Versuche dies möglichst geschickt zu machen, ohne streng dem Gaußschen Algorithmus zu folgen.)

4. Erzeuge eine 4×4 -Matrix mit **rand** und führe sie mit elementaren Zeilenumformungen in reduzierte Zeilenstufenform über. Führe dies in MATLAB durch und gebe keine Konstanten explizit ein. (Halte dich hier an den Gaußschen Algorithmus.)

Bonusfrage: wie sieht die reduzierte Zeilenstufenform einer solchen Matrix in wahrscheinlich fast allen Fällen aus?

5. Schlage in der Onlinehilfe die Befehle **rank**, **null**, **rref** und **orth** nach. Wie kann mit diesen ein lineares Gleichungssystem $Ax = b$ gelöst werden, wenn A und b in MATLAB als Matrix bzw. Vektor vorliegen? Probiere dies mit einfachen linearen Gleichungssystemen aus.

Hinweis: 'row echolon form' bedeutet 'Zeilenstufenform', und der 'nullspace' einer Matrix ist ihr Kern. Schliesslich ist 'range space' der Bildraum.

Verschiedene Methoden, lineare Gleichungssysteme numerisch zu lösen, werden in der Numerik-Vorlesung vorgestellt. Das klassische Gauß-Verfahren wird im Allgemeinen nicht verwendet, da es numerisch schlechte Eigenschaften hat.

2.7 Operationen auf Matrizen

Wie oben schon angedeutet, kann man einfach $A + B$ oder $A * B$ schreiben, um zwei Matrizen A und B passenden Formats zu addieren oder multiplizieren. Im wesentlichen kann man alle Operationen bei Ausdrücken auch auf Matrizen anwenden. Es gibt jedoch ein paar Feinheiten und Extras zu beachten:

- Mit $*$ wird die Matrizenmultiplikation bezeichnet; ebenso sind $/$, \wedge und \backslash in Bezug auf die Matrizenmultiplikation zu sehen.

Falls man komponentenweise multiplizieren, dividieren, exponentieren will, so muss man stattdessen $.*$, $./$, $.^$ oder $.\backslash$ schreiben.

- Man kann Matrizen A auch mit Skalaren s multiplizieren: $s * A$, $A * s$ und A / s sind erlaubt.
- Bei den Operationen $+$, $-$ kann auch eins der Argumente ein Skalar sein. In dem Fall wird er als Matrix im passenden Format aufgefasst mit allen Einträgen gleich diesem Skalar.

Beispiel: $[1 \ 2; \ 3 \ 4] + 5$ ergibt die Matrix $\begin{pmatrix} 6 & 7 \\ 8 & 9 \end{pmatrix}$.

Des weiteren können Funktionen auf Matrizen angewandt werden:

- Alle elementaren Funktionen wie **sin** und **sqrt** können auf Matrizen angewendet werden. Diese Anwendung erfolgt jedoch *komponentenweise* (und nicht nach einem Funktionenkalkül); dies entspricht also den Punkt-Operatoren bei Matrizen ($.*$, $./$, $.^$ etc.).

– Ist A eine Matrix mit reellen Einträgen, so ist etwa **sqrt**(A $.^$ 2) dasselbe wie **abs**(A)

Quizfrage: Was macht **sqrt**($A.*A'$ $.'$) für welche Eingaben?

- Dagegen ist im Allgemeinen **sqrt**(A^2) *nicht* gleich **abs**(A), da A^2 das Matrixprodukt $A*A$ ist, während **sqrt** davon komponentenweise die Wurzel zieht.
- Die Addition von Matrizen mit Skalaren lässt sich so auffassen: ist s ein Skalar, so betrachtet man die Funktion $\varphi_s : x \mapsto x + s$. Dann ist $A + s$ gleich A eingesetzt in φ_s .
- Vorsicht ist geboten bei Vergleichsoperatoren wie $==$, $<$, $<=$, $>$ und $>=$, und bei logischen Operatoren wie $\&$ und $|$. Diese arbeiten ebenfalls Komponentenweise und liefern somit eine Matrix zurück! Beispiel:

```
>> A = [1 2; 3 4]; B = [1 2; 4 3];
>> C = [1 2 3; 4 5 6];
>> A == C
??? Error using ==> eq
Matrix dimensions must agree.
>> A == B
ans = 1 1
      0 0
```

Wenn getestet werden soll, ob zwei Matrizen gleich sind, muss man **isequal** verwenden:

```
>> isequal(A, C)
ans = 0
>> isequal(A, B)
ans = 0
>> isequal(A, [1 2; 3 4])
ans = 1
```

- Es gibt jedoch auch Funktionen, die “richtig” mit Matrizen arbeiten; **help elmat** gibt eine Liste von solchen Funktionen.
 - Ein Beispiel ist **expm**, welche das Matrix-Exponential ausrechnet (dies ist definiert als $\exp(A) := \sum_{k=0}^{\infty} \frac{A^k}{k!}$; man zeigt ähnlich wie bei der “gewöhnlichen” Exponentialfunktion, dass dies für alle Matrizen absolut konvergiert). Für Diagonalmatrizen stimmt **expm** mit **exp** überein, für andere Matrizen im Allgemeinen jedoch nicht! (Das Matrixexponential wird z.B. für Differentialgleichungen benötigt.)
 - Die bereits oben erwähnte Funktion **isequal** gehört ebenfalls in diese Kategorie.

Aufgabe 2.10:

1. Erzeuge eine Wertetabelle der Funktion $f(x)$, womit eine 2×41 -Matrix gemeint ist, in deren erster Zeile die Zahlen $-10, -9.5, -9, \dots, +10$ stehen und in der zweiten Zeile die Zahlen $f(-10), f(-9.5), f(-9), \dots, f(10)$.
 - (a) $f(x) = x^5 - 4x^4 + 3x^2 - 1$;
 - (b) $f(x) = e^{\frac{ix}{10}}$;
 - (c) $f(x) = (\sin x)^3 - \cos x$.
2. Was tun die folgenden Ausdrücke?
 - (a) $A' \cdot ' \cdot \wedge A$;
 - (b) $\text{diag}(\text{diag}(A)) + \text{diag}(\text{diag}(A, 1), 1) + \text{diag}(\text{diag}(A, 2), 2)$, falls A eine $n \times n$ -Matrix ist mit $n \geq 4$?

Aufgabe 2.11: Verifiziere numerisch, dass $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$ gilt, indem du mit MATLAB die n -te Partialsumme $\sum_{k=1}^n \frac{1}{k^2}$ ausrechnest. Hierfür solltest du Befehle finden, die bei gesetzter Variable n den Wert der Partialsumme mit Hilfe von Matrix-Operationen bestimmt.

2.8 Exkurs: Fließkommazahlen

Der standardmässig verwendete Datentyp ist **double**: eine Fließkommazahl mit doppelter Genauigkeit. (Das “doppelt” ist historisch zu sehen im Vergleich zu einer Fließkommazahl mit “einfacher” Genauigkeit.)

Es gibt nur endlich viele Fließkommazahlen. Deswegen können nicht beliebig grosse oder kleine Werte angenommen werden. Der Bereich darstellbarer Zahlen ist jedoch recht gross, er geht von $-(2 - 2^{-52})2^{1023}$ bis -2^{-1022} , $-0, 0, 2^{-1022}$ bis $(2 - 2^{-52})2^{1023}$.

Eine Fließkommazahl besteht aus einem Vorzeichen s , einer Mantisse m und einem Exponenten e . Der Wert ist $s \cdot m \cdot 2^e$. Dabei ist $s \in \{-1, 1\}$, $e \in \{-1022, \dots, 1023\}$, und $m \in [1, 2)$ ist von der Form $1 + x \cdot 2^{-52}$, wobei $x \in \{0, 1, \dots, 2^{52} - 1\}$ ist.

Zusätzlich gibt es noch spezielle Zustände, wie etwa $\pm \text{Inf}$, **NaN** und ± 0 (es gilt $0 == -0$ und **NaN** \sim **NaN**, jedoch $1/(+0) = +\text{Inf}$ und $1/(-0) = -\text{Inf}$).

Da es nur endlich viele Fließkommazahlen gibt, muss gerundet werden. Für zwei Fließkommazahlen $x_1 = s_1 \cdot m_1 \cdot 2^{e_1}$ und $x_2 = s_2 \cdot m_2 \cdot 2^{e_2}$ definiert man das Produkt $x_1 \cdot x_2$ etwa durch Runden von $s_1 s_2 \cdot m_1 m_2 \cdot 2^{e_1+e_2}$ auf die nächste Fließkommazahl $x_3 = s_3 \cdot m_3 \cdot 2^{e_3}$: es gilt also $s_3 = s_1 s_2$, $|m_1 m_2 2^{e_1+e_2-e_3} - m_3| \leq 2^{-52}/2 = 2^{-53}$. Insbesondere sind Fließkommazahloperationen im allgemeinen weder assoziativ noch distributiv.

Die Zahl **eps** = $\varepsilon = 2^{-52}$ ist die *Maschinengenauigkeit*: es ist die kleinste Zahl, so dass $1 + \varepsilon$ als Fließkommazahl nicht gleich 1 ist. Insbesondere gilt $1 + \text{eps}/2 == 1$, und deswegen gilt etwa


```
1 + (eps/2 + eps/2) == 1 + eps ~ 1 == (1 + eps/2) + eps/2;
```

d.h. die Addition ist nicht unbedingt assoziativ. Ebenso gilt

```
(1 + eps) * ((1 + eps) - 1) == (1 + eps) * eps
~ eps == (1 + 2 * eps) - (1 + eps)
== (1 + eps) * (1 + eps) - (1 + eps) * 1,
```

da $(1 + \varepsilon)2^{-52}$ eine andere Fließkommazahl ist als $1 \cdot 2^{-52}$; d.h. das Distributivgesetz gilt ebenfalls nicht.

Beim Rechnen mit Fließkommazahlen ist also eine gewisse Vorsicht geboten! Dies wird noch ausführlicher in der Numerik-Vorlesung behandelt, für den Kurs reicht es aus zu wissen, dass man aufpassen muss.

Etwa ist es meist unklug, Fließkommazahlen direkt zu vergleichen; stattdessen sollte man die Differenz bilden und schauen, ob diese klein genug ist (etwa $|a - b| < 2^{-30}$ überprüfen anstelle $a = b$).

Aufgabe 2.12:

1. Stelle folgende Zahlen als Fließkommazahlen dar, soweit dies möglich ist. Falls es nicht exakt möglich ist, begründe warum das der Fall ist.

(a) 15 (b) 0.5 (c) 0.1 (d) $\frac{1}{3}$ (e) $\frac{1}{256}$

2. Versuche, die Rechnungen oben nachzuvollziehen. Warum gelten jeweils Gleichheit bzw. Ungleichheit?
3. Wenn die oben genannten Fließkommazahlen alle möglichen sind, wieviele Fließkommazahlen vom Typ `double` gibt es dann? (Hier soll zwischen `+0` und `-0` unterschieden werden.)

2.9 Native Datentypen

Wie schon erwähnt, gibt es einen Standarddatentypen: `double` (Fließkommazahl). Daneben gibt es jedoch auch noch weitere, wie `logical` oder `char`.

Verschiedene Datentypen haben verschiedene Wertebereiche und Genauigkeiten. Es gibt beispielsweise Ganzzahltypen, die ganze Zahlen in jeweils einem gewissen Intervall exakt darstellen können. Es gibt auch Verbundtypen, mit denen verschiedene Daten in einer Variable gespeichert werden können – zwei davon werden wir später behandeln (`struct` und `cell`).

Neben `double` gibt es die folgenden Typen:

- `logical` (Wahrheitswert: 0 für `false` und 1 für `true`),
- `single` (Fließkommazahl einfacher Genauigkeit),
- `char` (Zeichen; ganzzahliger Wert in $[0, 2^{16} - 1]$),
- `int8`, `int16`, `int32`, `int64` (Ganzzahlen in $[-2^7, 2^7]$, $[-2^{15}, 2^{15}]$, $[-2^{31}, 2^{31}]$ oder $[-2^{63}, 2^{63}]$),
- `uint8`, `uint16`, `uint32`, `uint64` (nicht-negative Ganzzahlen $< 2^8$, $< 2^{16}$, $< 2^{32}$ oder $< 2^{64}$),
- `function_handle` (Funktionenhandle; wird später behandelt),
- `struct` und `cell` (erweiterte Datentypen, Verbundtypen; werden später behandelt).

Den Typ eines Ausdrucks `A` kann man mit `class(A)` herausfinden. Den Typ ändern (casten) kann man mit `typename(Wert)`. Beispiele:

```
>> x = 1;
>> class(x)
ans = double
>> x = char(65)
x = A
>> class(x)
```

```

ans = char
>> int8(1000)
ans = 127

```

Beachte:

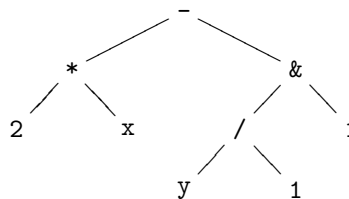
- das Zeichen A hat den ASCII-Code 65;
- ein `int8` kann nur Werte von -128 bis 127 annehmen; da 1000 zu gross ist, wird der grösste zulässige Wert genommen.

Man kann in bestimmten Fällen Werte verschiedenen Typs in Ausdrücken verwenden.

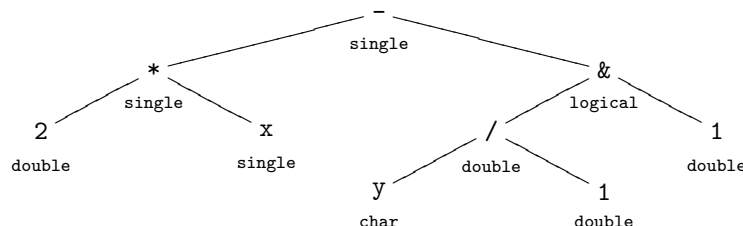
1. man kann einen `intX`-Typ nur mit `intX`-Typen,³ `chars` oder `doubles` verknüpfen, und das Ergebnis ist dann immer vom Typ `intX`;
2. verknüpft man `double` mit `single` – egal in welcher Reihenfolge –, ist das Ergebnis immer vom Typ `single`;
3. verknüpft man `logical` oder `char` mit `single` oder `double`, so ist das Ergebnis ein `single` bzw. `double`;
4. verknüpft man `char` mit `logical`, so ist das Ergebnis ein `double`;
5. verknüpft man `char` mit `intX`, so ist das Ergebnis ein `intX`.

Verknüpft man dagegen ein `intX`-Typ mit einem `intY`-Typ mit $X \neq Y$, so erhält man einen Fehler.

Um den Typ eines Ausdruckes zu bestimmen, bietet es sich an, zuerst einen Baum zu dem Ausdruck zu konstruieren und dann dort Schrittweise den Typ hinzuzuschreiben, ähnlich wie zuvor bei der Bestimmung des Wertes eines Ausdrucks. Um den Wert zu bestimmen ist es sowieso essentiell, die Typen mit zu beachten. Betrachten wir etwa den Ausdruck $2 * x - (y / 3 \& 1)$. Diesem Ausdruck entspricht der folgende Baum:



Hierbei sei `x` eine Variable vom Typ `single` und `y` eine Variable vom Typ `char`. Man erhält dann folgende Typen:



Eine Zeichenkette in MATLAB ist übrigens eine Matrix mit Einträgen vom Typ `char`: der Ausdruck `v = 'Test'` liefert eine 1×4 -Matrix, und es ist etwa `v(2) == 'e'`. Zeichenketten aneinanderhängen kann man mit Matrixoperationen: so ergibt etwa `['ab', 'cd']` die 1×4 -Matrix, die der Zeichenkette `'abcd'` entspricht.

Der folgende Befehl gibt eine Zeichentabelle aus:

³Verknüpft man einen `intX`-Typ mit einem anderen Wert vom Typ `intX`, so verhält sich MATLAB anders als die meisten anderen Programmiersprachen: wäre das Ergebnis einer solcher Operation eine Zahl, die nicht mehr im gültigen Intervall des Datentyps liegt, so wird der kleinste bzw. grösste darstellbare Wert des Types `intX` genommen, je nachdem ob das theoretische Ergebnis unterhalb bzw. oberhalb des kleinsten bzw. grössten darstellbaren Wertes liegt.

```
>> reshape(char(32 : 127), 16, 6)
ans =
! "#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

Quizfrage: Versuche, Anhand dieses Beispiels die Bedeutung des **reshape**-Befehls zu erraten.

Aufgabe 2.13: Überlege, welche Typen die folgenden Ausdrücke haben, und gebe den Wert an. Verifiziere dies mit MATLAB, ob Typ und Wert stimmen. Sei dazu **A** die Matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4i \end{pmatrix}$.

1. `A .* A' - (A.'.^A)'`;
2. `isequal(A, A')`;
3. `sum(sum(A - A')) == 10`;
4. `sum(diag(A == A')) & abs(sum(sum(A)))`;
5. `int8(255) + 1`;
6. `int32(int8(255) + 1) * 5`;
7. `logical(255) + 1`;
8. `int64(double(int64(2)^63) - 1) - int64(2)^63`.

2.10 Der Workspace Browser

Der Workspace ist der momentane Arbeitsbereich von MATLAB: er umfasst alle gerade definierten Variablen mit Informationen zu diesen, sprich dem Format der Matrix und dem Typ der Einträge. Eine Liste aller Variablen im Workspace kann mit **who** ausgegeben werden, und der Workspace kann mit **clear** in den leeren Zustand zurückgesetzt werden.

Beispiel:

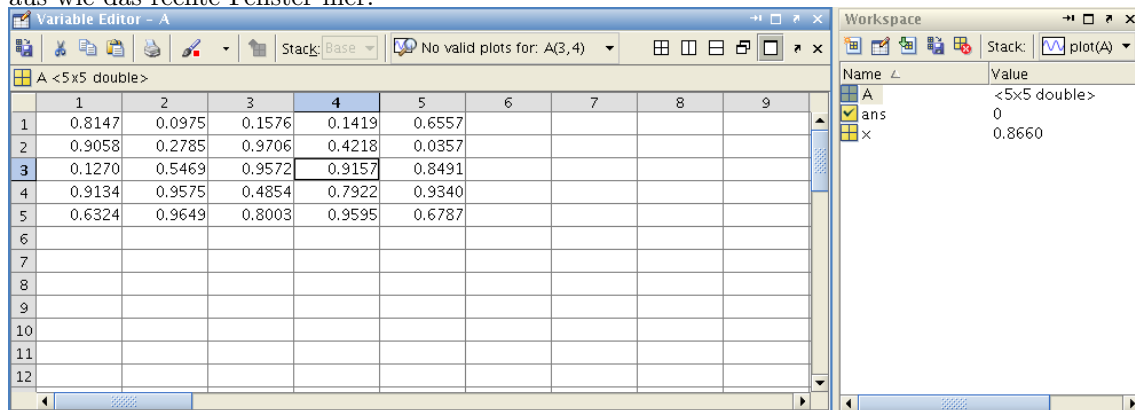
```
>> x = 1;
>> clear
>> y = int16(5);
>> z = 1 | 2;
>> who
Your variables are:
y z
>> class(z)
ans = logical
```

Eine weitere Übersicht bietet der Befehl **whos** an: er zeigt nicht nur die Namen der Variablen im Workspace, sondern auch ihr Format (also Grösse der Matrix), ihren Speicherbedarf (in Bytes) sowie ihren Datentyp:

```
>> A = rand(5, 10);
>> t = 'blah';
>> whos
```

Name	Size	Bytes	Class	Attributes
A	5x10	400	double	
t	1x4	8	char	
y	1x1	2	int16	
z	1x1	1	logical	

Alternativ kann man den Workspace im *Workspace Browser* betrachten; dies ist das Fenster, was sich normalerweise rechts oben vom Kommandozeilenfenster befindet. Es sieht dann etwa so aus wie das rechte Fenster hier:



Neben dem Namen der Variable wird der Wert angezeigt, bzw. das Format inklusive Datentyp bei Matrizen. Wenn man nach rechts scrollt, kann man bei Arrays noch den grössten und kleinsten Eintrag sehen. Klickt man eine Matrix (Zeichenketten gelten hier nicht als Matrix) doppelt an, öffnet sich der *Variable Editor* (links im Bild). Dort kann man eine Matrix inspizieren und die Werte ändern. Bei anderen Variablen (1×1 -Matrizen und Zeichenketten) kann man den Wert editieren, indem man den Wert der Variable markiert und dann nochmal anklickt.

Aufgabe 2.14: Erzeuge die Matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.23456 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9.8765 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 42 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

vom Format 10×18 in MATLAB, indem du zuerst eine Matrix dieser Grösse, bestehend nur aus Nullen erzeugst, und dann mit Hilfe des Variable Editors die Zahlen $\neq 0$ einfügst.

Bisherige Lernziele

- Wissen was ein Ausdruck in MATLAB ist
- Typ und Wert eines Ausdrucks bestimmen können, ohne ihn in MATLAB einzugeben
- (komplexe) Matrizen erstellen können und auf Elemente und Teilmatrizen zugreifen können
- Matrizen manipulieren, u.a. Zeilen- und Spaltenoperationen, Verknüpfen von Matrizen, etc.
- Wissen, wo Probleme bei der Verwendung von Fließkommazahlen liegen


3 Skripte und Plots

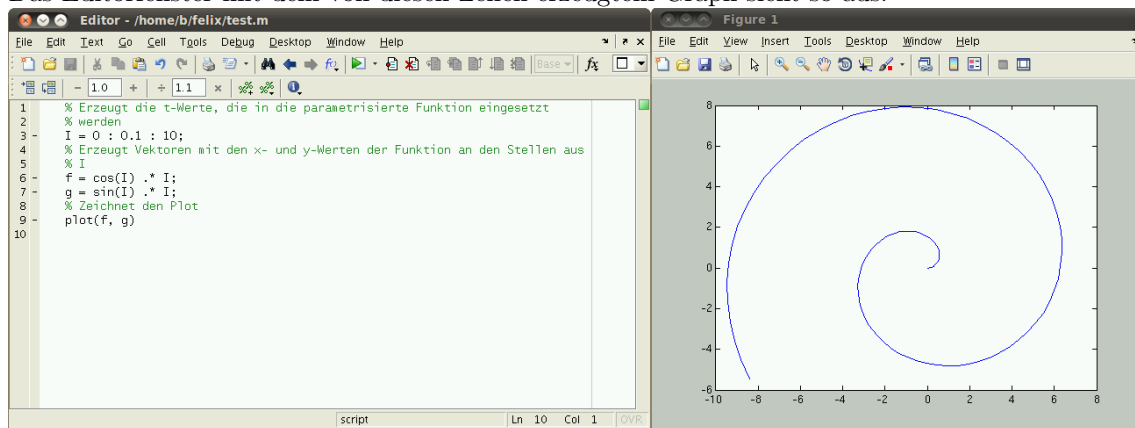
3.1 Skripte in MATLAB

Skripte sind in MATLAB eine Ansammlung von Befehlen, die man auch so in der Kommandozeile eingeben könnte, in einer Datei. Diese können durch Aufruf eines Befehls so ausgeführt werden. Erstellt man etwa eine Datei `test.m` im aktuellen Verzeichnis mit folgendem Inhalt:

```
% Zeichnet einen Graphen
I = 0 : 0.1 : 10;
f = cos(I) .* I;
g = sin(I) .* I;
plot(f, g)
```

und gibt dann in der Kommandozeile `test` ein, so wird ein Plot mit einer Spirale erzeugt. (Warum diese Befehle genau diesen Plot erzeugen, schauen wir uns gleich genauer an.)

Um die Datei zu erzeugen, klickt man in der Toolbar den Neues-Skript-Button  an, der sich ganz links befindet. Dann öffnet sich das Editorfenster, in dem man die obigen Befehle eingeben kann. Dies kann man speichern, etwa unter dem Namen `test.m`, in dem man in der Toolbar des Editor-Fenster auf das Disketten-Symbol klickt, oder im Menü `File` den Befehl `Save` auswählt. Das Editorfenster mit dem von diesen Zeilen erzeugtem Graph sieht so aus:



Der Screenshot zeigt auch, wie man Kommentare schreibt: alles nach einem Prozentzeichen `%` wird ignoriert. Falls in der ersten Zeile in einem Skript ein Kommentar anfängt, wird dieser in der Schnellansicht unten links im MATLAB-Fenster angezeigt, wenn die `.m`-Datei vom Skript in der Dateiauswahl ausgewählt ist. Er wird ebenfalls angezeigt, wenn man in der Kommandozeile `help Skriptname` eingibt.

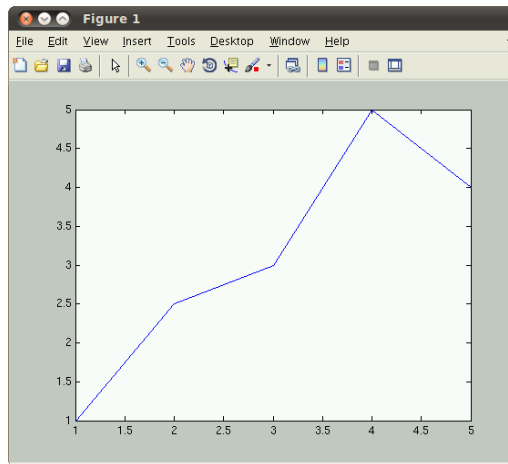
Man kann das Skript auch direkt aus dem Editor ausführen, indem man im Editorfenster in der Toolbar auf das Symbol  klickt.

Mit dem, was wir bisher gemacht haben, benötigt man Skripte noch nicht so wirklich. Jedoch schon bei komplexeren Plots ist es hilfreich, die Befehle in einem Skript zu sammeln. Im nächsten Kapitel werden wir Kontrollstrukturen kennenlernen; diese machen in Skripten mehr Sinn, als wenn man sie direkt an der Kommandozeile eingibt.

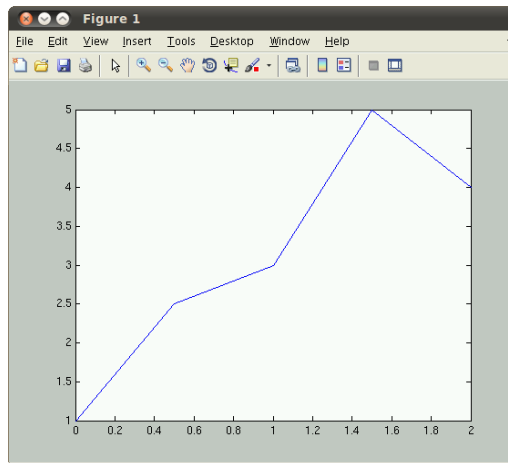
Aufgabe 3.1: Erzeuge das Skript `test` mit dem oben angegebenen Code. Speicher es und führe es aus, einmal aus dem Editor und einmal über die Kommandozeile.

3.2 Zweidimensionale Plots

Der grundlegende Befehl zum Zeichnen von Plots ist der Befehl `plot`. Hat man etwa einen Vektor mit Funktionswerten, sagen wir `v = [1 2 3 5 4]`, so zeichnet `plot(v)` folgendes:



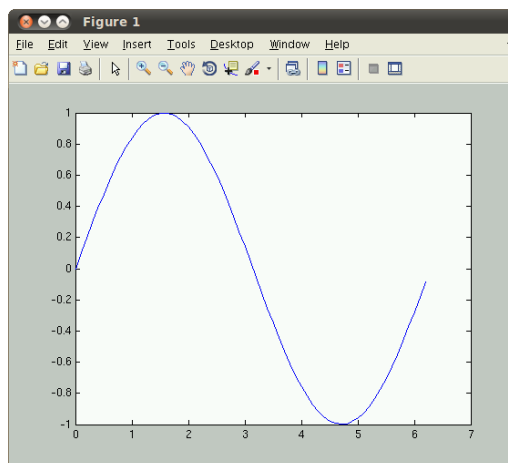
Die durchgezogene Linie befindet sich bei $x = 1$ bei $v(1)$, bei $x = 2$ bei $v(2)$, etc. Wenn man die x -Werte anders setzen will, muss man diese ebenfalls als Vektor angeben: ist $w = [0 \ 0.5 \ 1 \ 1.5 \ 2]$, oder kürzer $w = 0 : 0.5 : 2$, so ergibt `plot(w, v)` folgendes Ergebnis:



Damit lassen sich schon viele Dinge machen. Möchte man etwa den Graphen der Sinus-Funktion von 0 bis 2π plotten, wobei man Zwischenschritte von 0.1 machen möchte, so schreibt man etwa:

```
>> v = 0 : 0.1 : 2*pi;
>> w = sin(v);
>> plot(v, w)
```

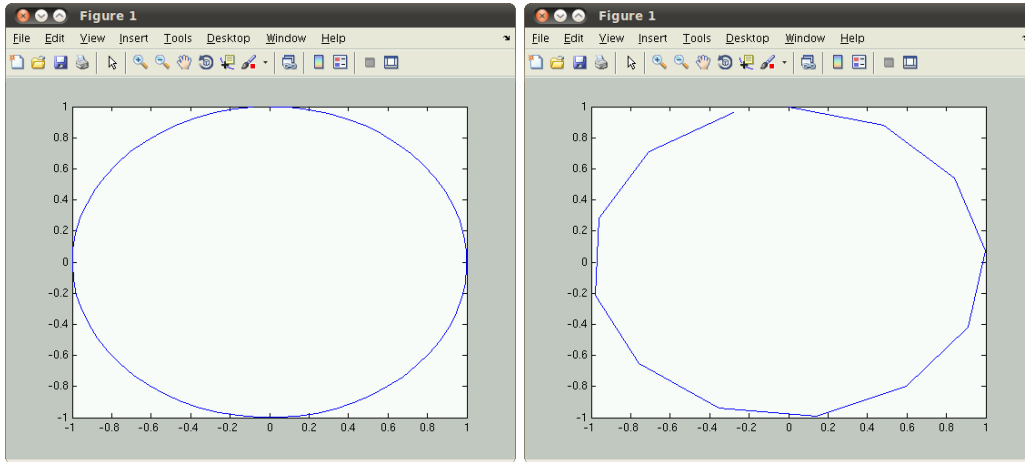
Hierbei macht man sich zunutze, dass die Funktion `sin` angewendet auf eine Matrix komponentenweise ausgeführt wird. Das Ergebnis ist dann:



Mit dem Plot-Befehl lassen sich auch ganz andere Dinge zeichnen, etwa eine parametrisierte Kurve $v(t) = (f(t), g(t))$, $t \in I$. Ist zum Beispiel $f(t) = \sin t$, $g(t) = \cos t$ und $I = [0, 2\pi]$, so kann man sich die Kurve mit

```
>> I = 0 : 0.1 : 2*pi;
>> f = sin(I); g = cos(I);
>> plot(f, g)
```

anzeigen lassen. Das Ergebnis ist unten links dargestellt. Wenn man die Schrittweite größer wählt, erkennt man, dass es sich um einen Streckenzug handelt. So liefert $I = 0 : 0.5 : 2\pi$ mit ansonsten den gleichen Befehlen die rechte hier dargestellte Grafik:



Wenn man genau hinschaut, so sieht man, dass der letzte Streckenzug fehlt. Das Problem ist hier, dass der letzte gezeichnete Punkt bei $t = 6$ liegt und nicht bei $t = 2\pi \approx 6.283$. Dies kann man verhindern, indem man die Schrittweite als Differenz von End- und Startpunkt geteilt durch eine ganze Zahl wählt, etwa wenn man I per $I = 0 : 2\pi/100 : 2\pi$ erzeugt.

Noch einfacher ist es jedoch, anstelle des Doppelpunkt-Operators die MATLAB-Funktion **linspace** zu verwenden. Schreibt man etwa **linspace(0, 2*pi, 100)**, so erzeugt dies einen Zeilenvektor mit 100 Einträgen, der erste davon 0 und der letzte 2π , und die restlichen gleichmässig dazwischen verteilt.

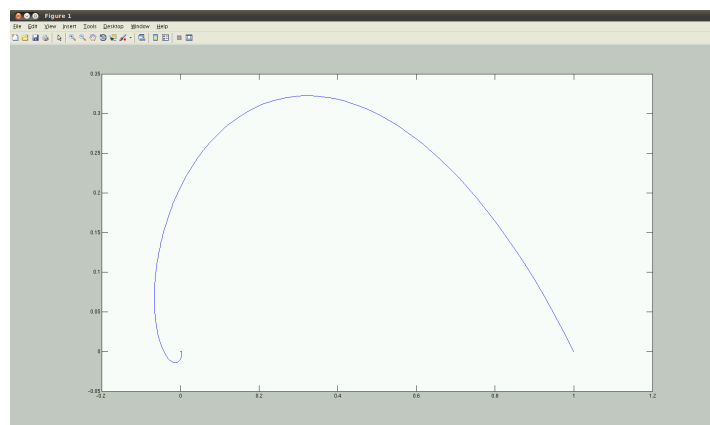
Aufgabe 3.2: Plote die folgenden Funktionen und parametrisierten Kurven:

1. $f(x) = x^2 - 1$;
2. $f(x) = \tan \sin x$;
3. $\gamma(t) = (\sin t \cos t, \cos t - \sin t)$ für $t \in [0, 2\pi]$;
4. $\gamma(t) = (t^2, t^3)$ für $t \in \mathbb{R}$.

Eine weitere Möglichkeit, zweidimensionale Plots zu zeichnen, besteht in der Verwendung eindimensionaler Vektoren mit komplexen Zahlen. Ruft man **plot(v)** auf und enthält v mindestens eine echt komplexe Zahl, so bewirkt **plot(v)** das gleiche wie **plot(real(v), imag(v))**. Damit lässt sich zum Beispiel die Spirale $f : [0, \infty) \rightarrow \mathbb{C}, t \mapsto \exp((i - 1)t)$ per

```
plot(exp((i-1)*(0 : 0.025 : 100)))
```

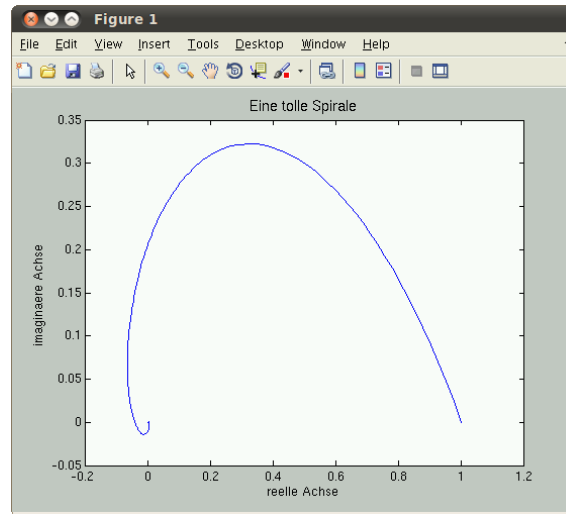
zeichnen:



Sobald ein Plot erzeugt wurde, lassen sich Titel und Beschriftung der Achsen festlegen:

```
>> xlabel('reelle_Achse')
>> ylabel('imaginaere_Achse')
>> title('Eine_tolle_Spirale', 'FontSize', 12)
```

Dies ergibt dann:



Aufgabe 3.3: Gebe **help plot** ein, um den Hilfetext des **plot**-Befehls zu erhalten. Erzeuge damit die folgenden Plots:

1. die Funktion $f : [-1, 1] \rightarrow \mathbb{R}, x \mapsto x^2 - 0.5$ mit roter gestrichelter Linie;
2. die Funktion $f_s : [-1, 1] \rightarrow \mathbb{R}, x \mapsto (x/s)^2$ für die Werte $s \in \{-1, -0.5, 0.5, 1\}$ in verschiedenen Farben im gleichen Plot;
3. die Funktion $f : [0, 1] \rightarrow \mathbb{R}, x \mapsto \sin(2\pi x)$, wobei keine Linie dargestellt werden soll, sondern die Funktionswerte als kleine Kreise an den Stellen $x = \frac{n}{10}, n = 0, 1, \dots, 10$.
4. das gleiche wie bei 2., mit dem Unterschied dass eine Legende vorhanden ist, die den verschiedenen Farben die Werte von s zuordnet. Benutze dafür den Befehl **legend** (informiere dich erst in der Hilfe, wie dieser funktioniert).

Ein praktischer Befehl ist noch der **hold**-Befehl: er erlaubt das Erweitern eines bereits erzeugten Plots um weitere Plots. Dies kann etwa so genutzt werden:

```
>> plot(sin(0 : 0.1 : 10))
>> hold on
>> plot(cos(0 : 0.1 : 10))
>> plot(sin(0 : 0.1 : 10) + cos(0 : 0.1 : 10))
>> hold off
```

Dies zeichnet die drei Funktionen $\sin(x)$, $\cos(x)$ und $\sin(x) + \cos(x)$ in einen Graphen ein.

Ein weiterer Befehl ist der **axis**-Befehl. Er erlaubt, die Grenzen der Achsen selbst festzulegen. Details hierzu finden sich in der Online-Hilfe.

Aufgabe 3.4: Stelle die ersten fünf Newton-Iteration der Funktion $f(x) = x^5 - 4x^2 + 3x - 2$ mit dem Startwert $x_0 = 3$ graphisch da, indem du zu jedem Paar $(x_n, f(x_n))$, $n = 0, \dots, 5$ einen Punkt in den Plot der Funktion einzeichnest zusammen mit einer Tangente an f in diesem Punkt. Verwende Achsenbeschriftungen, den **hold**-Befehl und eine Legende, um die Ausgabe ansprechend und verständlich zu gestalten.

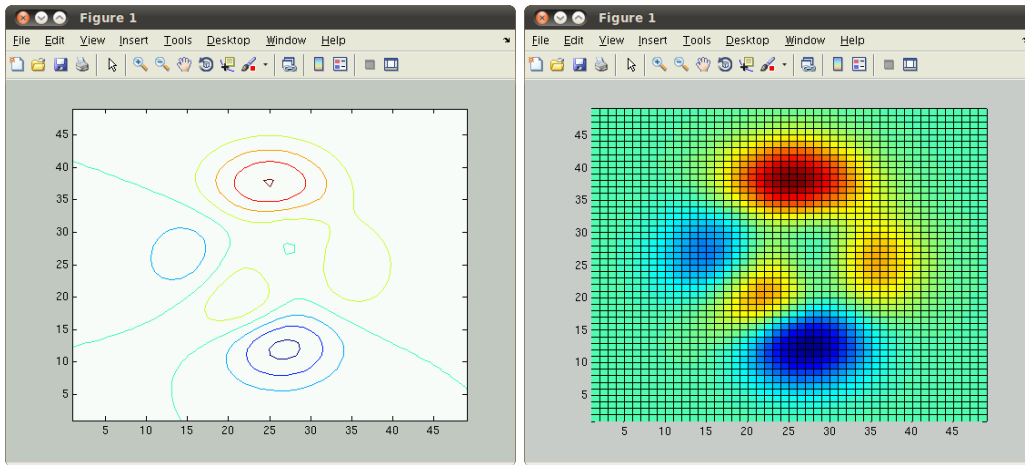
3.3 Dreidimensionale Plots

Es gibt mehrere Wege, zweidimensionale Funktionen $\mathbb{R}^2 \rightarrow \mathbb{R}$ (bzw. dreidimensionale Daten) darzustellen. Eine Möglichkeit ist, diese als Konturen- oder Farbplot darzustellen. Es gibt zum Beispiel

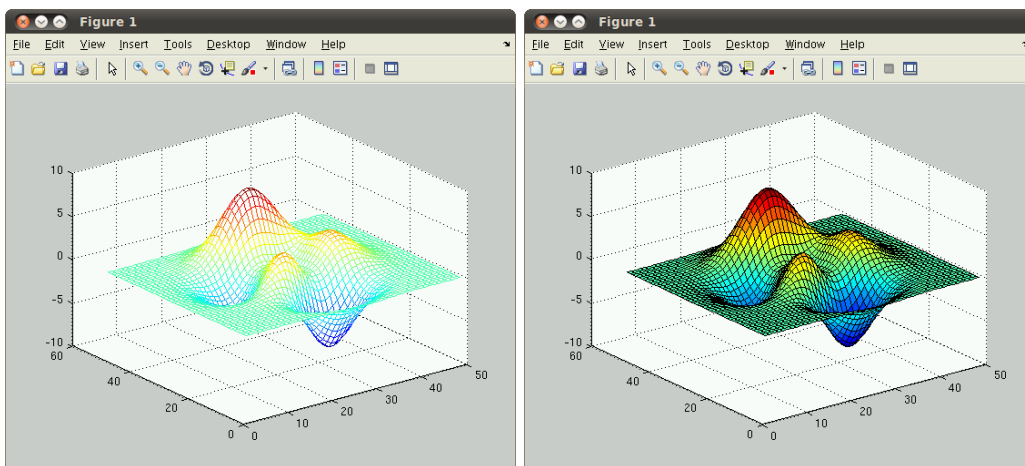
in MATLAB die Funktion **peaks**, welche durch

$$(x, y) \mapsto 3(1 - x)^2 \exp(-x^2 - (y + 1)^2) - 10\left(\frac{1}{5}x - x^3 - y^5\right) \exp(-x^2 - y^2) - \frac{1}{3} \exp(-(x + 1)^2 - y^2)$$

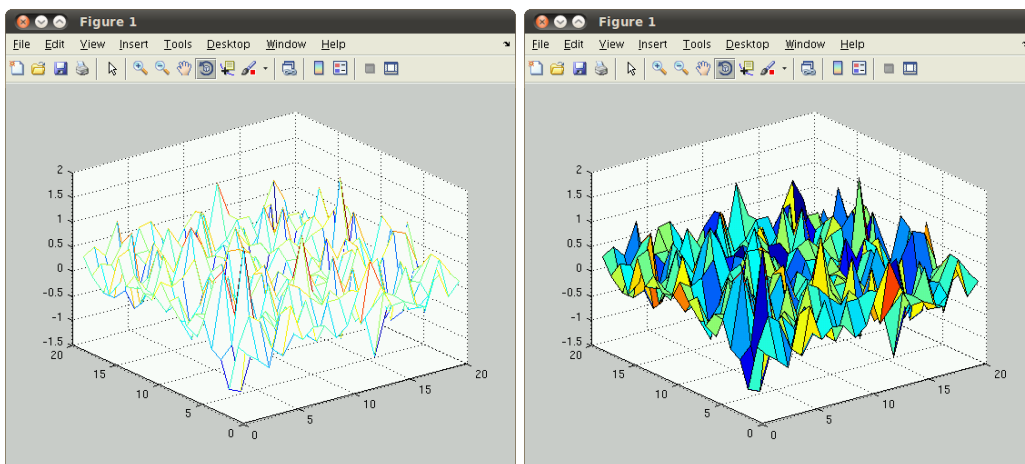
definiert ist. Die Befehle **contour(peaks)** und **pcolor(peaks)** liefern folgende Grafiken:



Eine zweite Möglichkeit, zweidimensionale Funktionen darzustellen, sind 3D-Drahtgitterplots und Flächenplots. Die Befehle **mesh(peaks)** und **surf(peaks)** liefern folgende Grafiken:



Man kann auch genauso wie bei zweidimensionalen Plots rohe Daten dreidimensional darstellen. Hat man etwa eine Matrix wie **A = randn(20, 20) * 0.5**, so erzeugen **mesh(A)** und **surf(A)** folgende Grafiken:



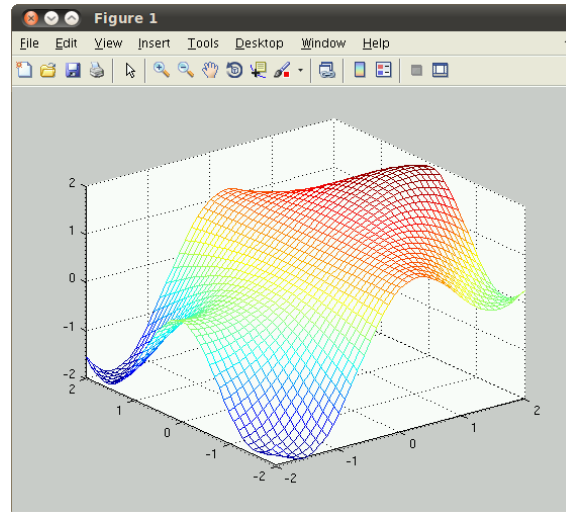
Um Graphen von Funktionen erstellen zu können, ist der **meshgrid**-Befehl praktisch. Hat man einen Vektor **x** der x -Koordinaten (der Länge n) und einen Vektor **y** der y -Koordinaten (der Länge

m), so erzeugt `[X,Y] = meshgrid(x, y)` zwei Matrizen `X` und `Y` vom Format $n \times m$, so dass jede Zeile von `X` den Zeilenvektor `x`, und jede Spalte von `Y` den Spaltenvektor `y` enthält. Zum Beispiel liefert `[X,Y] = meshgrid(0 : 1 : 2, 5 : 1 : 6)` die Matrizen

$$X = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix} \quad \text{und} \quad Y = \begin{pmatrix} 5 & 5 & 5 \\ 6 & 6 & 6 \end{pmatrix}$$

zurück. Gibt man nur einen Vektor an, etwa bei `meshgrid(v)`, so bewirkt dies das gleiche wie `meshgrid(v, v)`.

Wenn man nun die Funktion $(x,y) \mapsto \sin x + \cos xy$ plotten möchte im Bereich $[-2,2] \times [-2,2]$, kann man zuerst `X` und `Y` durch `[X,Y] = meshgrid(-2 : 0.1 : 2)` erzeugen, dann die Funktionswerte-Matrix `Z` durch `Z = sin(X) + cos(X .* Y)`, und schliesslich das ganze mittels dem Befehl `mesh(X, Y, Z)` plotten. Das ergibt dann



Aufgabe 3.5: Stelle folgende Funktionen $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ auf verschiedene Art und Weisen graphisch dar:

1. $f(x,y) = x^2 y^2$
2. $f(x,y) = \frac{\sin \sqrt{x^2+y^2}}{\sqrt{x^2+y^2}};$
3. $f(x,y) = \frac{1}{x^2+y^2};$

die letzten beiden Funktionen haben in $(x,y) = (0,0)$ einen Pol. Wie kann man Probleme beim Erzeugen des Wertefeldes und dem Plotten umgehen?

Bisherige Lernziele

- Wissen wie man Skripte in MATLAB schreibt, ausführt und mit Kommentaren versieht
- Verstehen, wie Plots von eindimensionalen Funktionen und parametrisierten Kurven im \mathbb{R}^2 geplottet werden können
- Verstehen, wie dreidimensionale Plots funktionieren und wie Wertetabellen dafür mit `meshgrid` erstellt werden können

4 Kontrollkonstrukte und Schleifen

Bisher haben wir einfache Befehle kennengelernt: Ausdrücke, Zuweisungen, und Skripte, die Plots erzeugen. Damit war es bisher nicht möglich, sonderlich viel zu machen. In einigen Aufgaben haben wir gesehen, dass Schleifen oder Abfragen zwar teilweise durch Matrixoperationen oder geschickte Ausdrücke ersetzt werden können, jedoch waren wir oft gezwungen, Befehle zu replizieren oder je nach Iteration von Hand auszuwählen, welche Befehle ausgeführt werden oder nicht.

Kontrollkonstrukte erlauben, nach bestimmten Bedingungen gewisse Befehle auszuführen oder nicht auszuführen.

4.1 Kontrollkonstrukte

Das grundlegendste Kontrollkonstrukt ist die **if**-Anweisung. Das folgende Beispiel verdeutlicht, wie sie funktioniert:

```
if <Ausdruck 1>
  <Befehle 1>
elseif <Ausdruck 2>
  <Befehle 2>
else
  <Befehle 3>
end
```

Zuerst wird hier der Ausdruck **<Ausdruck 1>** ausgewertet. Ergibt er 1 (logical, also irgendetwas $\neq 0$), werden **<Befehle 1>** ausgeführt. Falls er 0 (logical) ergibt, wird **<Ausdruck 2>** ausgewertet. Ergibt dieser 1 (logical, also $\neq 0$), so werden **<Befehle 2>** ausgeführt. Andernfalls werden **<Befehle 3>** ausgeführt.

Die **if**-Anweisung ist *modular*: es können beliebig viele **elseif**-Zweige

```
elseif <Ausdruck>
  <Befehle>
```

eingefügt werden, es kann auch kein einziger solcher Zweig verwendet werden. Der **else**-Zweig

```
else
  <Befehle>
```

kann ebenfalls weggelassen werden. Falls er vorkommt, muss er immer zum Schluss kommen (als letzter Block vor dem **end**) und er darf nicht mehrmals vorkommen.

Falsch:

```
if <Ausdruck>
  <Befehle>
else
  <Befehle>
else
  <Befehle>
end
```

Falsch:

```
if <Ausdruck>
  <Befehle>
else
  <Befehle>
elseif <Ausdruck>
  <Befehle>
end
```

Richtig:

```
if <Ausdruck>
  <Befehle>
end

if <Ausdruck>
  <Befehle>
else
  <Befehle>
end
```

Aufgabe 4.1: Schreibe ein Skript, welches davon ausgeht, dass eine Variable **x** eine **double**-Zahl ist. Das Skript soll folgendes ausgeben:

1. ob die Zahl negativ, gleich 0 oder positiv ist;
2. ob die Zahl im Intervall $[0, 1]$ liegt oder nicht;
3. ob die Zahl gleich -10 ist oder nicht.

Spezialfälle wie **±Inf** und **NaN** können ignoriert werden.

Aufgabe 4.2: Schreibe eine **if**-Abfrage für das Bisektionsverfahren aus Aufgabe 2.7. Diese soll davon ausgehen, dass **x1** und **x2** gegeben sind mit $x1 < x2$ und $\cos(x1) * \cos(x2) < 0$ (es ist also wieder $f(x) = \cos x$).

Diese soll **x3** wie in Aufgabe 2.7 wählen und entweder **x1** oder **x2** durch den Wert von **x3** ersetzen.

Schreibe diese **if**-Abfrage in ein Skript namens **bisect**, setze **x1** = 1 und **x2** = 2, und führe das Skript zehnmal aus. Dann gilt $\mathbf{abs}(x2 - x1) = 2^{-10}$, d.h. die Nullstelle von $\cos x$ in $[1, 2]$ (es gibt genau eine) ist dadurch bis auf eine Genauigkeit von $2^{-10} < 10^{-3}$ bestimmt.

Eine weitere Kontrollkonstruktion ist die **switch**-Anweisung. Diese funktioniert so:

```
switch <Ausdruck>
    case <Wert 1>
        <Befehle 1>
    case <Wert 2>
        <Befehle 2>
    case <Wert 3>
        <Befehle 3>
    otherwise
        <Befehle 4>
end
```

Dies bewirkt, dass der Ausdruck **<Ausdruck>** ausgewertet wird. Ist der Ausdruck gleich **<Wert 1>**, so werden **<Befehle 1>** ausgeführt. Ist der Ausdruck gleich **<Wert 2>**, so werden **<Befehle 2>** ausgeführt. Ist der Ausdruck gleich **<Wert 3>**, so werden **<Befehle 3>** ausgeführt. Ansonsten, falls der Ausdruck keinem der drei Werte entspricht, werden **<Befehle 4>** ausgeführt.

Es können beliebig viele **case**-Zweige benutzt werden, und der **otherwise**-Zweig kann weggelassen werden. (In dem Fall passiert nichts, wenn der Ausdruck ungleich allen spezifizierten Werten ist.) Es ist auch möglich, einfach nur

```
switch <Ausdruck>
end
```

zu schreiben. In diesem Fall wird der Ausdruck ausgewertet (und etwa Funktionen, die im Ausdruck vorkommen, aufgerufen) und dann nichts gemacht. Dies ändert den Wert der Variablen **ans** nicht.

Die **switch**-Anweisung macht dann Sinn, wenn ein Ausdruck einmal ausgewertet und mit einer festen Anzahl von Werten verglichen werden soll. Sollen dagegen Ungleichungen geprüft werden, muss man auf **if**-Anweisungen zurückgreifen.

Aufgabe 4.3: Die Funktion **rem(n, m)** liefert den (nichtnegativen) Rest von **n** bei Division durch **m**. Der Rest hat das selbe Vorzeichen wie **n**, und hierbei müssen weder **n** noch **m** ganze Zahlen sein. Weiterhin kann mit dem **disp**-Befehl der Wert einer Variablen oder Zeichenkette ausgegeben werden.

Schreibe damit ein Skript, welches testet, ob die Variable **x** eine ganze Zahl ist, und wenn ja, den Rest bei Division modulo 3 ausgibt. Dies soll über eine **switch**-Anweisung erfolgen. Bei einer Eingabe von 5 soll etwa 2 ausgegeben werden (da $5 = x \cdot 3 + 2$ für ein passendes $x \in \mathbb{Z}$), und bei einer Eingabe von 2.123 soll **Keine ganze Zahl** ausgegeben werden.

4.2 Schleifen

Schleifen erlauben die wiederholte Ausführung von Befehlen. In MATLAB gibt es zwei Arten von Schleifen: **for**-Schleifen und **while**-Schleifen. Während **for**-Schleifen eine vorher festgelegte Anzahl durchlaufen werden, sind **while**-Schleifen in dieser Hinsicht dynamisch, da sie solange ausgeführt werden, bis eine beliebige, vorher festgelegte Bedingung nicht mehr gilt.

Eine **for**-Schleife funktioniert wie folgt:

```
for <Variable> = <Vektor>
    <Befehle>
end
```

Die Variable **<Variable>** nimmt alle Werte der Reihe nach an, die im Vektor **<Vektor>** stehen, und führt für jeden Wert **<Befehle>** aus. Etwa ergibt

```

for n = [1 4 1]
    n
end

```

die Ausgabe

```

n = 1
n = 4
n = 1

```

Es bietet sich an, für die Erzeugung des Vektors den `:-`Operator zu nutzen. Um etwa mit `n` die Zahlen 2, 3, 4, 5, 6 zu durchlaufen, schreibt man

```

for n = 2 : 6
    <Befehle>
end

```

Um die Zahlen 2, 2.5, 3, 3.5, ..., 6 zu durchlaufen, schreibt man stattdessen

```

for n = 2 : 0.5 : 6
    <Befehle>
end

```

In den meisten MATLAB-Programmen wird der `:-`Operator bei `for`-Schleifen benutzt; dies ist jedoch, wie oben demonstriert, nicht notwendig.

Aufgabe 4.4: Erweitere das Skript `bisekt` von oben (Aufgabe 4.2), um die Nullstelle von $\cos x$ in $[1, 2]$ mit einer Genauigkeit von 2^{-50} zu bestimmen. Der bisherige Inhalt vom Skript `bisekt` soll dabei erhalten bleiben, es muss davor und danach etwas eingefügt werden. In jedem Iterationsschritt sollen `x1`, `x2` und `x2 - x1` ausgegeben werden.

Hinweis: verwende zur Ausgabe den `fprintf`-Befehl:

```

fprintf('%0.20d, \u0020%0.20d, \u0020%0.20d\n', x1, x2, x2 - x1)

```

Aufgabe 4.5: In dieser Aufgabe soll π mit Hilfe einer Monte-Carlo-Methode approximiert werden.

Dazu wählt man erst eine Anzahl Iterationen, sagen wir `n`, und setzt einen Zähler `T` auf 0. Dann wird `n`-mal zufällig (gleichverteilt) ein Punkt $(x, y) \in [-1, 1]^2$ gewählt und die euklidische Norm von (x, y) bestimmt (also $\sqrt{x^2 + y^2}$). Ist diese ≤ 1 , so wird der Zähler `T` erhöht.

Wenn `n` gross genug ist, nimmt der Kreis mit Radius 1 im Quadrat $[-1, 1] \times [-1, 1]$ in etwa $\frac{T}{n}$ der Fläche ein, womit

$$\pi \approx 4 \cdot \frac{T}{n}$$

ist. Schreibe ein Skript `montecarlopi`, welches voraussetzt, dass `n` vorher gesetzt wurde, und π mit `n` Monte-Carlo-Iterationsschritten approximiert.

Führe das Skript mehrmals für `n = 1000`, `n = 10000` und `n = 100000` aus. Was kannst du über die Genauigkeit sagen?

Allgemein kann in MATLAB mit `format long` die Ausgabe von Zahlen in der Kommandozeile auf ein längeres Format mit mehr Dezimalziffern eingestellt werden. In der Aufgabe wird die Genauigkeit der Ausgabe im `fprintf`-Befehl auf 20 Nachkommastellen für jede Zahl eingestellt.

Die zweite Möglichkeit, Schleifen in MATLAB zu schreiben, funktioniert mit der `while`-Anweisung. Diese funktioniert so:

```

while <Ausdruck>
    <Befehle>
end

```

Solange der Ausdruck `<Ausdruck>` wahr ist (also $\neq 0$), werden `<Befehle>` ausgeführt. Folgende zwei Programme geben jeweils die Zahlen von 1 bis 10 aus:

```

for k = 1 : 10
    disp(k)
end

```

```

k = 1
while k <= 10
    disp(k)
    k = k + 1
end

```

Wie man sieht, ist eine **for**-Schleife hierfür besser geeignet. Es gibt jedoch auch Fälle, in denen die Anzahl der Iterationen vor der Schleife noch nicht feststehen. Ein Beispiel dafür ist diese Aufgabe:

Aufgabe 4.6: Schreibe ein Skript, welches mit dem Newton-Verfahren (vergleiche Aufgabe 2.6) eine Nullstelle von $f(x) = x^{10} - 5x^6 + 2x^3 - x - 15$ approximiert. Die Iteration soll solange durchgeführt werden, bis $|f(x_n)| \leq 10^{-10}$ gilt. Der Startwert sei $x_0 = 0$.

Aufgabe 4.7: Schreibe ein Programm, welches einen Parameter **fehler** (positiver Wert) annimmt und untersucht, wie gross man **n** mindestens wählen muss, damit

$$\left| \sum_{k=1}^n \frac{1}{k^2} - \frac{\pi^2}{6} \right| \leq \text{fehler}$$

ist.

Damit keine Probleme mit der Maschinengenauigkeit auftreten, gehe wie folgt vor: setze eine Variable **A** auf $\frac{\pi^2}{6}$. Dann ziehe $\frac{1}{1^2}$ ab und schaue ob **A** <= **fehler** ist. Wenn nicht, ziehe $\frac{1}{2^2}$ ab und schaue wieder, ob **A** <= **fehler** ist. Wenn nicht, ziehe $\frac{1}{3^2}$ ab, usw.

Untersuche, wie gross du **n** mindestens wählen muss, damit der Fehler kleiner 10^{-1} , 10^{-2} und 10^{-3} ist.

Im Zusammenhang mit Schleifen gibt es noch zwei Kontrollanweisungen: **continue** und **break**. Wird **continue** in einer Schleife aufgerufen, so wird sofort die nächste Iteration der Schleife eingeleitet (soweit möglich) und die aktuelle Iteration abgebrochen. Wird **break** aufgerufen, so wird die vollständige Schleife abgebrochen.

Der **break**-Befehl ist etwa beim Bisektionsverfahren praktisch: falls **f(x3) == 0** ist, soll die Ausführung der Schleife sofort abgebrochen werden und **x3** als Nullstelle ausgegeben werden.

Der **continue**-Befehl kann etwa benutzt werden, um zufällig Punkte zu erzeugen, die in dem Einheitskreis in der Ebene liegen. Das folgende Skript erzeugt 10 solche Punkte:

```

n = 10;
while n > 0
    x = rand; y = rand;
    if x^2 + y^2 > 1
        continue
    fprintf('%0.20d, \u0020%0.20d\n', x, y)
    n = n - 1;
end

```

Alternativ hätte man auch ohne die **continue**-Anweisung schreiben können:

```

n = 10;
while n > 0
    x = rand; y = rand;
    if x^2 + y^2 <= 1
        fprintf('%0.20d, \u0020%0.20d\n', x, y)
        n = n - 1;
    end
end

```

In diesem Fall sind beide Varianten gleich lang, je nach Schleife kann jedoch die Variante mit oder ohne **continue** besser lesbar und/oder kürzer sein.

4.3 Benutzen des Debuggers

Beim Programmieren treten leicht Fehler auf. Sobald man einen Fehler bemerkt hat, ist es oft nicht einfach, dessen Ursache zu identifizieren oder sie sogar zu beheben.

Ein wichtiges Hilfsmittel bei der Fehleridentifikation ist der *Debugger*. Er erlaubt, die Ausführung eines Programms mitzuverfolgen, bei Bedarf zu unterbrechen, Variablen zu inspizieren und sogar zu verändern.

Wir wollen dies an mehreren Beispielen betrachten. Zuerst folgendes, einfaches Skript:


```
n = 10;
% Erzeuge n Zufallszahlen in [0, 1] und sortiere diese aufsteigend
A = rand(n, 1);
for k = 1 : n
    % finde das kleinste Element in den hinteren n-k+1 Elementen
    kli = k;
    kl = A(k);
    for l = k + 1 : n
        if A(l) < kl
            kli = l;
            kl = A(l);
        end
    end
    % verschiebe das kleinste Element an die k-te Stelle
    for l = k : kli
        A(l + 1) = A(l);
    end
    A(k) = kl;
end
```

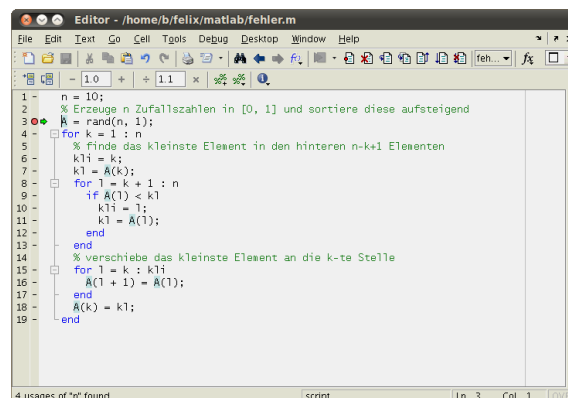
Wenn man es ausführt und A ausgeben lässt, erhält man eine Ausgabe wie

0.182922469414914	0.479922141146060	0.479922141146060
0.479922141146060	0.479922141146060	0.479922141146060
0.479922141146060	0.479922141146060	0.479922141146060
0.479922141146060	0.479922141146060	

(Die Ausgabe erfolgt nur deshalb nicht in einer, sondern in drei Spalten, damit etwas weniger Platz verbraucht wird.)

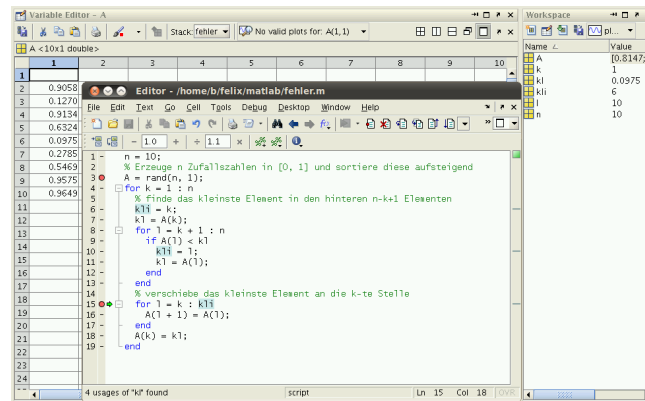
Dies ist tatsächlich aufsteigend. Allerdings ist es sehr, sehr unwahrscheinlich, dass von zehn zufällig in $[0, 1]$ erzeugten Zahlen neun gleich sind. Irgendetwas stimmt also nicht.

Um den Fehler zu finden, wollen wir den Debugger verwenden. Zuerst setzen wir einen *Break-point* in Zeile 3, in der die Zufallszahlen erzeugt werden. Dazu klicken wir im Editor am linken Rand in die Spalte rechts neben der Spalte mit den Zeilennummern. Dort befindet sich vorher ein Strich “-”; dieser besagt, dass sich in dieser Zeile ein Befehl befindet. An der Stelle des Strichs erscheint nun ein roter Punkt. Wenn man das Skript nun ausführt (auf  in der Toolbar klicken), wird die Ausführung an dieser Stelle unterbrochen. Im Editor erscheint neben dem roten Punkt ein grüner Pfeil, der anzeigt, dass als nächstes diese Zeile ausgeführt wird. Das sieht in etwa so aus:



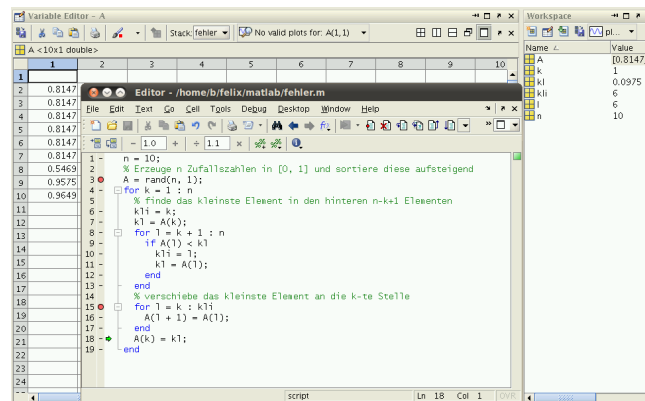
Im Workspace Browser kann man bisher nur die Variable `n` sehen, die den Wert 10 hat. Nun kann man diese eine Zeile ausführen, indem man auf den Step-Button in der Toolbar klickt (drei Buttons neben dem Ausführen-Button), oder indem man F10 drückt.

Nun erscheint im Workspace Browser auch die Variable `A`, und wenn man diese im Variable Editor inspiziert, sieht man, dass diese nicht aufsteigend sortiert sind. Drückt man drei weitere Male F10, so befindet man sich in Zeile 8, am Anfang der inneren Schleife. Die Variable `k` hat den Wert 1, `k1` (bisher kleinster Wert) den Wert `A(1)` und `kli` (Index des bisher kleinsten Wertes) ist ebenfalls 1. Man kann nun so lange F10 drücken, bis man das Ende der inneren Schleife erreicht hat, oder alternativ einen weiteren Breakpoint setzen (in Zeile 15) und das Programm mit F5 (oder dem Continue-Button, drei Buttons rechts vom Step-Button in der Toolbar) bis zum Breakpoint laufen lassen. Das kann dann etwa so aussehen:



Der Cursor befindet sich gerade hinter dem `kli` in Zeile 15; deswegen wird jedes Vorkommen der Variable `kli` im Editor markiert. Weiterhin kann man sich den Wert einer Variable auch anzeigen lassen, wenn man die Maus über die Variable schiebt und dort kurz ruhen lässt. Dann wird Typ, Format und Wert der Variable in einem kleinen Fenster (Tooltip) angezeigt.

Hier wurde nun ermittelt, dass der kleinste Wert an der sechsten Stelle steht (da `kli == 6` ist). Die nächste Schleife sowie die nachfolgende Zuweisung soll diesen Wert nun an die erste Stelle schieben. Wenn man nun mit F10 ein paar Iterationen der Schleife durchlaufen lässt, sieht man im Variable Editor gleich das Problem: der erste Wert im Array überschreibt alle anderen:



Dies liegt offenbar daran, dass die Schleife in der falschen Reihenfolge durchlaufen wird! Damit dies richtig funktioniert, muss mit dem grössten Index angefangen werden und mit dem kleinsten aufgehört werden. Statt `for l = k : kli` sollte man also `for l = kli : -1 : k` schreiben. Man kann dies nun im Editor ändern und alle Breakpoints löschen, indem man sie entweder nochmal anklickt oder indem man in der Toolbar auf den Clear-Breakpoints-Button klickt (der Button links vom Step-Button). Schliesslich muss man den Debug-Mode beenden, indem man etwa auf den Exit-Debug-Mode-Button klickt (rechts vom Continue-Button; oder Shift+F5 drücken). Nun kann man die Änderung auch speichern und das Skript nochmal ausführen.

Das Ergebnis ist jedoch immer noch nicht befriedigend. Man kann zum Beispiel für `A` folgendes erhalten:

0.186872604554379	0.186872604554379	0.381558457093008
0.186872604554379	0.186872604554379	0.438744359656398
0.186872604554379	0.381558457093008	0.438744359656398
0.186872604554379	0.381558457093008	

Aufgabe 4.8: Finde mit Hilfe des Debuggers heraus, was schiefgeht! Das Programm kann auf der Vorlesungshomepage als Text-Datei `fehlersuche-1.txt` heruntergeladen werden.

Bisherige Lernziele

- Wissen, wie das **if**- und das **switch**-Konstrukt funktionieren
- Verstehen von Schleifenkonstrukten mit **for** und **while**
- Wissen, wie damit Iterationen (wie etwa im Newton- oder Bisektionsverfahren) automatisiert werden können
- Wissen, was der Debugger ist und wie man ihn verwenden kann, um den Ablauf eines Programmes mitzuverfolgen

5 Funktionen und Skripte

Eine Funktion ist ein *Unterprogramm*. Es bekommt Argumente übergeben, und liefert Werte zurück. Bekannte Funktionen sind beispielsweise **sin**, **cos**, **exp**, **diag** und **rank**.

5.1 Inline-Funktionen

Die einfachste Möglichkeit, Funktionen zu definieren, sind **inline**-Funktionen. Schreibt man etwa `f = inline('exp(i*x)/10') , 'x')`, so kann die eine Wertetabelle aus Aufgabe 2.10 auch mit `[-10:0.5:10; f(-10:0.5:10)]` erstellen.

Eine **inline**-Funktion kann auch mehr als ein Argument haben: mit der Funktion

```
m = inline('a.*b', 'a', 'b')
```

etwa kann man eine (komponentenweise) Multiplikation `x .* y` als `m(x, y)` schreiben.

Solche **inline**-Funktionen erlauben es, Teilausdrücke mit eigener Bedeutung extra herauszuschreiben. Dies kann der Lesbarkeit eines Programms stark helfen.

Aufgabe 5.1: Schreibe eine **inline**-Funktion mit Argument `n`, welche $\sum_{k=1}^n \frac{1}{k!}$ ausrechnet. (Vergleiche Aufgabe 2.11; arbeite hier mit Matrizen und nicht mit einer Schleife.)

5.2 Funktionen in *m*-Dateien

Der eigentliche Weg, in MATLAB Funktionen zu definieren, geht über *.m*-Dateien. Eine neue Funktion kann erzeugt werden, indem man im File-Menü unter **New** den Punkt **Function** auswählt. Dann öffnet sich das Editor-Fenster und bietet gleich den Rumpf einer Funktion an:

```
function [ output_args ] = untitled1( input_args )
%UNTITLED1 Summary of this function goes here
% Detailed explanation goes here

end
```

Als erstes sollte man das **untitled1** durch den Namen der Funktion ersetzen und die Funktion als **Funktionsname.m** speichern. Weiterhin sollte man in der ersten Kommentar-Zeile das **UNTITLED1** durch den Funktionsnamen (in Grossbuchstaben ersetzen) und den Rest der Zeile in eine Kurzbeschreibung ändern. Diese wird bei **help Funktionsname** oder **doc Funktionsname** als Überschrift angezeigt. Die weiteren Kommentarzeilen werden als eigentlicher Hilfetext angezeigt und sollten eine ausführlichere Beschreibung enthalten. Um zu sehen, wie dies etwa bei der **rank**-Funktion aussieht, kann man in der Konsole **help rank** und **type rank** eingeben. Der **type**-Befehl zeigt den Quellcode einer Funktion an, soweit diese als *.m*-Datei vorhanden ist.⁴

Als nächstes ersetzt man **input_args** durch die Namen der Eingabeargumente, getrennt durch Kommata, und **output_args** durch die Namen der Ausgabeargumente, ebenfalls durch Kommata getrennt. Man kann auch nur eins oder gar kein Eingabe- oder Ausgabeargument angeben:

```
function [ ] = f()
    disp('hello world!')
end
```

Bei einem Aufruf erhält man dann:

```
>> f()
hello world!
```

Man kann hier auch die `()` beim Funktionsaufruf weglassen:

```
>> f
hello world!
```

⁴Wenn man sich den Quellcode des **rank**-Befehls anschaut, sieht man, dass der wichtigste Teil der Aufruf von **svd** (singular value decomposition) ist. Die Funktion **svd** ist im Gegensatz zu **rank** fest eingebaut: gibt man **type svd** ein, so gibt MATLAB die Meldung "**'svd' is a built-in function.**" aus.

Man kann eine Funktion mit weniger als den angegebenen Argumenten aufrufen; ruft man sie mit mehr Argumenten auf, gibt es eine Fehlermeldung. Die Funktion kann die Anzahl der tatsächlich übergebenden Argumente mit **nargin** erfragen. Ebenso kann man mehrere Ausgabeargumente definieren, von denen nicht alle bei einem Funktionsaufruf abgefragt werden müssen. Die Anzahl der tatsächlich vorhandenen Ausgabeargumenten kann durch **nargout** abgefragt werden. Jedem Ausgabeargument muss in der Funktion ein Wert zugewiesen werden, andernfalls treten Fehler auf wenn die Funktion mit genügend Ausgabeargumenten aufgerufen wird.

Betrachte etwa folgende Funktion:

```
function [x, y, z] = test2(a, b, c)
    disp([nargin, nargout]);
    x = 1;
    y = 2;
end
```

Hier wird absichtlich dem dritten Ausgabeargument **z** nichts zugewiesen. Dann erhält man etwa:

```
>> test2(1, 2)
     2     0           % zwei Eingabe- und kein Ausgabeargument
>> test2(1, 2, 3)
     3     0           % drei Eingabe- und kein Ausgabeargument
>> x = test2(1, 2)
     2     1           % zwei Eingabe- und ein Ausgabeargument
>> [x, y] = test2()
     0     2           % kein Eingabe- und zwei Ausgabeargumente
>> [x] = test2
     0     1           % kein Eingabe- und ein Ausgabeargument
>> [x, y, z] = test2(2)
??? Output argument "z" (and maybe others) not assigned
during call to "/home/b/felix/test2.m>test2".
```

Aufgabe 5.2:

1. Schreibe eine Funktion **fakultaet(n)**, welche die Fakultät von **n** mit einer **for**-Schleife berechnet.
2. Schreibe damit eine **inline**-Funktion **binom(n, m)**, welche den Binomialkoeffizienten $\binom{n}{m} = \frac{n!}{m!(n-m)!}$ berechnet.
3. Schreibe eine Funktion **binomialreihe(n)**, welche den Vektor

$$[\text{binom}(n, 0), \text{binom}(n, 1), \dots, \text{binom}(n, n)]$$

zurückgibt. Verifiziere damit die Gleichung $\sum_{m=0}^n \binom{n}{m} = 2^n$ für kleine Werte von n .

Wenn man Binomialkoeffizienten mit dieser Art für grosse Werte von **n** ausrechnet, bekommt man schnell numerische Probleme – dies liegt an der begrenzten Genauigkeit von Fließkommazahlen. Wir werden später sehen, wie man dieses Problem einfach umgehen kann.

Aufgabe 5.3: Schreibe eine Funktion **bisect2**, welche in etwa die Funktionalität vom Skript **bisect** (siehe Aufgabe 4.4) umfasst, und welche folgende Argumente akzeptiert:

1. **x1** und **x2**;
2. ein optionales Argument **res**, welches im Fall dass es nicht angegeben wird als **max(abs(x1), abs(x2)) * eps * 2** gewählt wird;

die Abbruchbedingung für den Algorithmus soll $x_2 - x_1 \leq \text{res}$ sein. Weiterhin soll eine Fehlermeldung ausgegeben werden, falls $x_2 < x_1$ ist bei Aufruf der Funktion oder falls $\cos(x_1) * \cos(x_2) \geq 0$ ist.⁵

Es soll zwei Ausgabeargumente geben: das erste ist eine untere Schranke für die Nullstelle und das zweite eine obere Schranke. Ist also $[x, y] = \text{bisect2}(\dots)$, so gibt es ein $z \in [x, y]$ mit $\cos z = 0$.

Aufgabe 5.4: Schreibe eine Funktion, die eine Matrix **A** und einen Vektor **b** übergeben bekommt und mit Hilfe der Funktion **rref** eine spezielle Lösung des linearen Gleichungssystems $A \cdot x = b$ ausrechnet, oder ausgibt dass es keine solche gibt.

Aufgabe 5.5 (Bonus): Implementiere den Gaußschen Algorithmus zur Berechnung einer reduzierten Zeilenstufenform in MATLAB.

Mögliche Vereinfachung: nimm an, dass die Matrix quadratisches Format hat und invertierbar ist. Dann gibt es in jeder Spalte ein Pivot-Element.

Mögliche Verbesserung: wähle jeweils die Zeile für ein Pivotelement aus, deren Eintrag in der Spalte betragsmäßig am grössten ist. (Im naiven Fall sucht man einfach nach dem ersten Eintrag $\neq 0$; dies ist jedoch oft nicht die beste Wahl.)

Vergleiche die Ausgabe deines Algorithmus mit der von **rref**, etwa für zufällige $n \times m$ -Matrizen, die von **rand** oder **randn** erzeugt werden.

5.3 Gültigkeitsbereich von Variablen

Eine Funktion hat ihren eigenen Workspace: die Funktion kann die Variablen, die im aufrufenden Workspace (also der Workspace der aufrufenden Instanz, ob dies nun eine andere Funktion ist oder die Kommandozeile) definiert sind, nicht sehen, und sie somit auch nicht ändern. Alle Variablen, die in der Funktion benutzt werden – ausgenommen die Eingabe- und Ausgabeargumente –, sind im aufrufenden Workspace nicht sichtbar.

Auch wenn eine Funktion ein zweites mal aufgerufen wird, ist der Workspace wieder leer – die Variablen, die im vorherigen Durchlauf der Funktion benutzt worden sind, sind wieder verschwunden. Eine Funktion kann also auf diese Weise keinen Status speichern.

Allerdings gibt es in MATLAB die Möglichkeit, globale Variablen anzulegen, wie in manchen anderen Programmiersprachen auch. Eine globale Variable kann in jedem Workspace sichtbar gemacht werden, und ändert man den Wert in einem Workspace in dem sie sichtbar ist, wird sie auch in allen anderen Workspaces geändert, in denen sie sichtbar ist. Insbesondere bleiben solche Variablen zwischen zwei Funktionsaufrufen erhalten – soweit sie nicht von anderen Funktionen verändert werden. Globale Variablen können wie folgt deklariert werden:

```
function test3(x)
    global globvar
    if nargin == 1
        globvar = x;
    else
        disp(globvar)
    end
end
```

Die globale Variable kann auch im Workspace der Kommandozeile sichtbar gemacht werden, indem man dort **global globvar** eingibt. Sie erscheint dann auch im Workspace Browser. Das ganze sieht dann zum Beispiel so aus:

```
>> test3
>> test3(4)
>> test3
4
>> test3(5)
5
>> global globvar
>> globvar
globvar = 5
```

Beim ersten Aufruf von **test3** ist **globvar** MATLAB zwar bekannt – es wird keine Fehlermeldung ausgegeben –, jedoch hat die Variable noch keinen Inhalt: der Wert ist die leere Matrix **[]** (vom Format 0×0 , mit Einträgen vom Typ **double**).

Macht man **globvar** im Kommandozeilen-Workspace sichtbar und ruft **whos** auf, so erhält man:

```
>> whos
```

Name	Size	Bytes	Class	Attributes
globvar	1x1	8	double	global

Wenn eine Funktion eine globale Variable verwendet, um einen Status zu speichern, sollte deren Namen möglichst so gewählt sein, dass er nicht zufällig auch von einer anderen Funktion benutzt wird. Heisst die Funktion etwa `Funktionentest`, so kann man die Variable etwa `Funktionentest_g_Variable1` nennen.

Aufgabe 5.6: Eine klassische (jedoch problemanfällige) Methode, Zufallszahlen zu generieren, ist der *lineare Kongruenzgenerator*. Man wählt dazu eine natürliche Zahl m , einen Faktor a , einen Inkrement b und einen Startwert y . Aus dem Zustand y erzeugt man einen neuen Zustand, indem man $a * y + b$ modulo m (siehe `help rem`) als neuen Zustand y wählt. Daraus erhält man eine Zufallszahl in $[0, 1)$, indem man y / m als Zufallszahl nimmt.

Implementiere so einen linearen Kongruenzgenerator mit $m = 2^{32}$, $a = 22695477$ und $b = 1$. Schreibe dazu zwei Funktionen `ZufallInit` und `zufall`; die Funktion `ZufallInit` soll den Status y (wähle einen anderen Namen für die globale Variable!) des Generators auf 1234 setzen, und `zufall` soll einen neuen Status generieren und eine Zufallszahl in $[0, 1)$ wie oben beschrieben zurückliefern.

5.4 Function Handles

Ein *Function Handle* ist ein Verweis auf eine Funktion, der etwa als Argument an eine andere Funktion übergeben werden kann. Diese Funktion kann dann die Funktion aufrufen, auf die das Function Handle verweist.

Ein Function Handle einer Funktion wie `sin` wird über `f = @sin` erzeugt. Hier ist `f` eine normale Variable vom Typ `function_handle`. Möchte man `f` benutzen, schreibt man `feval(f, 1)`; dies führt in diesem Fall `sin(1)` aus und liefert dessen Ergebnis zurück.

Die Funktion `inline`, die in Abschnitt 5.1 beschrieben wurde, liefert ebenfalls `function_handles` zurück. Das folgende Beispiel illustriert, wie `function_handles` eingesetzt werden können:

Aufgabe 5.7: Schreibe eine Funktion `bisect3(f, x1, x2, res)` analog zu der Funktion `bisect2` in Aufgabe 5.3. Der Unterschied zu `bisect2` ist das Argument `f` vom Typ `function_handle`: anstelle die Funktion `cos` fest zu verdrahten, soll `f` verwendet werden.

Teste die Funktion mit `f = @sin`, `x1 = 3` und `x2 = 4`, und mit `f = inline('x*exp(x)-1', 'x')`, `x1 = 0` und `x2 = 1`.

Aufgabe 5.8: Schreibe eine Funktion `newton(f, fp, x0, error)` ähnlich wie in Aufgabe 4.6, welche das Newton-Verfahren auf die Funktion `f` mit Ableitung `fp` anwendet, bis `abs(f(x)) < error` ist.

Teste die Funktion mit `f = @sin`, `fp = @cos`, `x0 = 3` und `error = 10-15`.

6 Weitere Datenstrukturen

Neben (eindimensionalen und zweidimensionalen) Matrizen kennt MATLAB noch weitere Datenstrukturen: mehrdimensionale Arrays (Matrizen), Cell Arrays sowie Structs.

6.1 Mehrdimensionale Arrays

Ein mehrdimensionales Array ist sozusagen eine mehrdimensionale Matrix: ist n eine natürliche Zahl und sind k_1, \dots, k_n natürliche Zahlen, so ist eine n -dimensionale Matrix vom Format $k_1 \times \dots \times k_n$ eine Datenstruktur, die aus $n_1 \dots n_k$ Elementen vom gleichen Typ besteht und die über n -Tupel (a_1, \dots, a_n) mit $1 \leq a_i \leq k_i$ indiziert werden können.

Man kann ein vierdimensionales Array vom Format $2 \times 2 \times 2 \times 2$ etwa per `A = zeros(2, 2, 2, 2)` erzeugen. Dann kann man etwa mit `A(1, 2, 2, 1)` auf ein Element zugreifen. Auch hier kann man `:` oder Arrays verwenden, genauso wie bei (zweidimensionalen) Matrizen, um auf Teilarrays zuzugreifen. So kann man etwa mit `A(1, :, :, 1)` auf ein Teilarray vom Format $1 \times 2 \times 2 \times 1$ zugreifen, also anders gesagt: auf eine 2×2 -Matrix.

Auf mehrdimensionale Arrays können genauso wie auf Matrizen Operatoren angewandt werden, die komponentenweise arbeiten.

6.2 Cell Arrays

Ein *Cell Array* ist eine Art mehrdimensionales Array, dessen Einträge wiederum Matrizen, mehrdimensionale Arrays, Cell Arrays oder beliebige andere Objekte sein können.

Man kann ein n -dimensionales Cell Array vom Format $k_1 \times \dots \times k_n$ durch `A = cell(k1, ..., kn)` erzeugen. (Auch hier gibt es wieder eine Ausnahme: im Fall $n = 1$ muss man `cell(k1, 1)` oder `cell(1, k1)` benutzen, andernfalls erzeugt man ein Cell Array vom Format $k_1 \times k_1$.) Hat man einen n -Index (a_1, \dots, a_n) , so kann man auf den (a_1, \dots, a_n) -Eintrag von `A` durch `A{a1, ..., an}` zugreifen.

Man kann auch direkt ein Cell Array erzeugen, indem man den `{ }`-Operator verwendet, der genauso funktioniert wie der `[]`-Operator bei Matrizen.

Beispiel:

```
>> A = { 1, [2, 3], [4, 5; 6, 7], { 8, 9 } }
A = [1] [1x2 double] [2x2 double] {1x2 cell}
>> A{3}(1, 2)
ans = 5
>> A{4}{2}
ans = 9
>> B = cell(3, 4)
B = [] [] [] []
     [] [] [] []
     [] [] [] []
>> B{2, 3} = { 1, 2, 3 };
>> B{3, 2} = [ 1, 2; 3, 4 ];
>> B{4, 1} = 512
B = [] [] [] []
     [] [] {1x3 cell} []
     [] [2x2 double] [] []
     [512] [] [] []
```

Ein Cell Array kann man etwa nutzen, um mehrere verschiedene Informationen zu einem Objekt zu speichern. Ist etwa `A` eine Matrix, so kann man das Cell Array

```
{ A, sum(A), sum(sum(A))/sum(size(A)) }
```

anlegen: der erste Eintrag ist die Matrix `A` selber, der zweite Eintrag der Vektor der Spaltensummen, und der dritte Eintrag der Mittelwert aller Einträge von `A`.

Aufgabe 6.1: Schreibe eine Funktion, die für gegebenes `N` ein Cell Array des Formats `N` zurückgibt, in dessen n -ten Eintrag ein magisches Quadrat der Grösse n steht.

Hinweis: `help magic`.

Aufgabe 6.2: Ist n eine natürliche Zahl, so kann man auf der Menge $M_n := \{0, 1, \dots, n-1\}$ durch $a \oplus_n b := a + b \bmod n$ eine Addition definieren; hierbei ist $a + b \bmod n$ das gleiche wie `mod(a + b, n)` in MATLAB, also der Rest von $a + b$ nach Division durch n . Das Paar (M_n, \oplus_n) ist eine Gruppe.

Schreibe eine Funktion, welche die Additionstabelle für die Gruppe (M_n, \oplus_n) als $n \times n$ -Matrix zurückgibt: am Eintrag (k, ℓ) soll $(k-1) \oplus_n (\ell-1)$ stehen.

Schreibe eine weitere Funktion, die für gegebenes N ein Cell Array des Formats N zurückgibt, in dessen n -ten Eintrag die Additionstabelle von (M_n, \oplus_n) steht.

6.3 Structs

Ein *Struct* (Abkürzung für *structure*) ist wie `double` oder `logical` ein Datentyp, der als Eintrag einer (mehrdimensionalen) Matrix auftauchen kann. Jeder Eintrag einer solchen Matrix enthält Felder, und jedes solche Feld kann einen anderen Wert und Typ haben.

Zum Beispiel kann man damit eine Adressliste erstellen: dazu nimmt man etwa eine $n \times 1$ -Matrix, so dass jeder Eintrag die Felder `name`, `vorname` und `adresse` enthält:

```
>> adrliste(1).name = 'Fontein';
>> adrliste(1).vorname = 'Felix';
>> adrliste(1).adresse = 'Irgendeine_Strasse_5,_Zuerich';
>> adrliste(2).name = 'Duck';
>> adrliste(2).vorname = 'Donald';
>> adrliste(2).adresse = 'Entenhausen';
>> adrliste(3).name = 'Matlab-Kurs';
>> adrliste(3).adresse = 'H52';
>> adrliste
adrliste = 1x3 struct array with fields:
            name
            vorname
            adresse

>> adrliste(2).vorname
ans = Donald
>> adrliste(3)
ans =         name: 'Matlab-Kurs'
        vorname: []
        adresse: 'H52'
```

Man muss aufpassen, dass man die Feldnamen richtig schreibt. Falls man auf ein nicht-existentes Feld zugreifen will, gibt es zwar eine Fehlermeldung:

```
>> adrliste(3).bla
??? Reference to non-existent field 'bla'.
```

Schreibt man jedoch in ein noch nicht existierendes Feld etwas hinein, so wird dieses Feld angelegt:

```
>> adrliste(1).adresse = 'noch_keine';
>> adrliste
adrliste = 1x3 struct array with fields:
            name
            vorname
            adresse
            adresse
```

Aufgabe 6.3: Schreibe eine Funktion, die als Argument einen Namen `name` und eine Adressliste `adrliste` (wie oben) übergeben bekommt, und sucht, ob es einen Eintrag mit diesem Namen gibt. Jeder Eintrag `adrliste(k)` mit `isequal(adrliste(k).name, name)` soll mit `disp(adrliste(k))` ausgegeben werden.

7 Konzepte

In diesem Abschnitt werden Programmierkonzepte vorgestellt, die häufig in der Praxis vorkommen.

7.1 Rekursion

Eines der wichtigsten Konzepte ist die *Rekursion*. Eine Funktion arbeitet rekursiv, wenn sie sich (indirekt) selber aufruft. Das einfachste Beispiel ist die Berechnung der Fakultät. Die rekursive Definition lautet

$$n! := \begin{cases} 1 & \text{falls } n = 0, \\ n \cdot (n-1)! & \text{falls } n > 0. \end{cases}$$

Dies kann einfach in eine Funktion übertragen werden:

```
function fak = fakultaet(n)
    if n == 0
        fak = 1
    else
        fak = n * fakultaet(n - 1)
    end
end
```

Wichtig ist bei einer Rekursion die *Abbruchbedingung*, damit die Rekursion nicht unendlich oft durchgeführt wird (was in MATLAB zu einem Fehler führt). In diesem Fall ist es die Bedingung `if n == 0`.

Im Fall der Fakultät ist Rekursion nicht die beste Möglichkeit, die Funktion `fakultaet` zu implementieren. Eine `for`-Schleife ist hier effizienter. Es gibt jedoch auch viele Fälle, in denen eine Rekursion einfacher oder besser ist.

Es ist oft der Fall, dass rekursive Algorithmen in iterative Algorithmen umgewandelt werden können. Dies ist teilweise effizienter, jedoch sind solche Algorithmen oft schlechter lesbar. Ein Beispiel ist folgende Aufgabe:

Aufgabe 7.1: Schreibe eine Funktion, welches zu gegebenen `n` alle *Partitionen* von `n` findet. Dabei ist eine Partition ein Tupel (a_1, \dots, a_k) mit $a_1 \geq a_2 \geq \dots \geq a_k \geq 1$ und $\sum_{\ell=1}^k a_\ell = n$.

Gehe dafür wie folgt vor: schreibe eine Funktion `searchPart(Anfang, n, maxa)`, welche alle Partitionen von `n` findet mit Einträgen $\leq \text{maxa}$. Jede gefundene Partition soll ausgegeben werden, wobei der Ausgabe `Anfang` vorgehängt werden soll.

Beispiel: `searchPart([4, 2], 3, 2)` soll alle Partitionen (a_1, \dots, a_k) von 3 finden mit $2 \geq a_1$. Diese sind $(2, 1)$ und $(1, 1, 1)$. Ausgegeben werden soll dann $[4, 2, 2, 1]$ und $[4, 2, 1, 1, 1]$.

Die Funktion `searchPart` ist rekursiv zu implementieren: sie soll alle Möglichkeiten für a_1 durchprobieren und dann `searchPart` mit um a_1 verlängertem `Anfang`, mit um a_1 verringertem `n` und mit a_1 als neuem `maxa` aufgerufen werden. Im Falle von `n == 0` wird einfach `Anfang` mit `disp()` ausgegeben.

Schlussendlich soll die Funktion `searchPart(n)` die Funktion `searchPart(Anfang, n, maxa)` mit passenden Werten für `Anfang` und `maxa` aufrufen.

Diese Aufgabe lässt sich auch durch eine geschickte Iteration lösen. Jedoch ist dies recht mühsam zu implementieren und erfordert eine gewisse Geschicklichkeit im Umgang mit Schleifen, Kontrollstrukturen und Vektoren.

Eine weitere Anwendung ist die binäre Suche. Diese hatten wir bereits in Form des Bisektionsverfahrens (vergl. Aufgaben 2.7, 4.2, 4.4, 5.3 und 5.7):

Aufgabe 7.2: Schreibe eine Funktion `bisectrec(f, x1, x2, res)`, wobei `f` wie in Aufgabe 5.7 beschrieben ein Function Handle und `res` wie in Aufgabe 5.3 beschrieben ein optionales Argument sein soll. Die Funktion `bisectrec` soll rekursiv das Bisektionsverfahren auf die Funktion `f` anwenden.

Bei der binären Suche wird nach etwas gesucht, in diesem Fall nach einer Nullstelle von `f`. Es ist bekannt, dass eine Nullstelle im Intervall $[x_1, x_2]$ liegt. Das Intervall wird in der Hälfte unterteilt, also in zwei gleichgroße Intervalle $[x_1, x_3]$ und $[x_3, x_2]$ aufgeteilt. Dann wird untersucht, ob im ersten Intervall eine Nullstelle liegt (dies ist die Bedingung `f(x1) * f(x3) < 0`); falls ja, wird das

Verfahren auf $[x_1, x_3]$ angewendet, andernfalls auf $[x_3, x_2]$. In jeder Rekursionsstufe wird die Länge des Intervalls halbiert.

Aufgabe 7.3: Sei v ein Vektor der Länge n , der aufsteigend sortiert ist. Schreibe eine Funktion `suche(v, w)`, welche neben dem Vektor v einen Wert w übergeben bekommt. Die Funktion soll feststellen, ob w im Vektor v vorkommt. Dazu wird das Element in der Mitte von v mit w verglichen. Ist w kleiner, so muss w wenn schon in der ersten Hälfte von v vorkommen; es wird dann `suche(vlinks, w)` aufgerufen, wobei `vlinks` die linke Hälfte von v ist. Analog wird vorgegangen, falls w grösser als das Element in der Mitte ist.

Implementiere die Funktion `suche` und überlege dir, wie die Abbruchbedingungen der Rekursion aussehen.

Man kann zeigen, dass die Laufzeit der binären Suche $O(\log n)$ ist. Geht man dagegen der Reihe nach alle Elemente von v durch und vergleicht sie mit w , so ist die Laufzeit gleich $O(n)$, also *wesentlich* langsamer.

Ein ähnlicher Effizienzgewinn lässt sich beim Potenzieren mit natürlichen Zahlen erreichen. Betrachte die Funktion:

```
function p = power(a, b)
% hier soll b eine positive ganze Zahl sein
if b == 0
    p = 1;
else
    p = power(a, b - 1) * a;
end
end
```

Diese Funktion berechnet zwar $\text{power}(a, b) = a^b$, braucht dafür jedoch $O(b)$ Multiplikationen. Dies kann wesentlich verbessert werden durch eine Methode, die als “square and multiply” oder “binary exponentiation” bekannt ist. Beispielsweise kann man a^9 ausrechnen als $((a^2)^2)^2 \cdot a$; dies benötigt vier Multiplikationen, im Gegensatz zu den acht Multiplikationen von $a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$. Für allgemeine a und b kann dies wie folgt implementiert werden:

```
function p = power(a, b)
% hier soll b eine positive ganze Zahl sein
if b == 0
    p = 1;
else
    p = power(a * a, floor(b / 2));
    if rem(b, 2) == 1
        p = p * a;
    end
end
end
```

Man kann hier zeigen, dass $O(\log b)$ Multiplikationen durchgeführt werden.

7.2 Backtracking

Backtracking ist ein Konzept, welches oft eng mit Rekursion verwachsen ist, jedoch nicht sein muss. Ein gutes Beispiel ist die Suche nach einem Ausgang aus einem Labyrinth. Man probiert einen Weg, und wenn man merkt dass es eine Sackgasse ist, kehrt man zur letzten Verzweigung zurück und probiert den anderen Weg. Ist dies ebenfalls eine Sackgasse, so geht man zur letzten Verzweigung zurück wo es noch Wege gibt, in die man nicht gegangen ist, und probiert einen dieser. Dies führt man immerzu fort, bis man einen Ausgang gefunden hat.⁶ Backtracking ist auch eng mit dem Konzept der *Tiefensuche* verbunden, welches bei der Durchsuchung von Bäumen⁷ ein wichtiges Verfahren ist.

⁶Falls das Labyrinth einen Rundweg enthält, kommt man hiermit nicht zum Ziel, es sei denn man markiert den bisher zurückgelegten Weg.

⁷Etwa von *Spielbäumen* bei Spielen: beim Tic-Tac-Toe-Spiel besteht der Spielbaum etwa aus allen Möglichkeiten, wie das Spiel durchgeführt werden kann. An der Wurzel kann erst der erste Spieler alle möglichen Spielzüge auswählen. Danach gibt es Abzweigungen für alle Spielzüge des Gegners, usw. Das Ziel ist es, eine Abzweigung zu finden, die möglichst sicher zu einem Sieg führt; dies kann etwa dadurch erreicht werden, dass eine Tiefensuche durchgeführt wird, die den ganzen Baum durchläuft. Dies funktioniert genauso wie das Backtracking im Labyrinth.

Aufgabe 7.4: Erzeuge eine Matrix `laby` von Format $n \times m$, welche nur 0 und 1 enthält (0 für Weg, 1 für Wand) und deren Rand nur aus 1en besteht. Die Position (2,2) sollte ein Weg sein und es sollte einen Weg dorthin geben. Beispiel:

$$\text{laby} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Schreibe nun eine Funktion `sucheweg(laby, x, y)`, welche einen Weg von der Position (x, y) zu der Position (2,2) per Backtracking sucht. Im Falle dieser Matrix könnte man etwa `sucheweg(laby, 5, 9)` aufrufen.

Der Weg soll mit den Einträgen 2 markiert werden. Voraussetzung für `sucheweg(laby, x, y)` ist, dass an der Position (x, y) in `laby` der Wert 0 steht. Die Abbruchbedingung der Rekursion ist $(x, y) = (2, 2)$: in diesem Fall wird der Wert der Variablen `laby` ausgegeben.

Beispiel: `sucheweg(laby, 5, 9)` soll die Ausgabe

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 & 2 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 2 & 2 & 2 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 2 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

liefern.

7.3 Dynamisches Programmieren

Dynamische Programmierung ist ein Konzept, welches sich für Probleme einer gewissen Struktur eignet. Es eignet sich für Probleme,

1. deren Lösung einfach aus Lösungen etwas kleinerer Teilprobleme zusammengesetzt werden können;
2. diese kleineren Teilprobleme haben eine grosse Überlappung.

Hier soll das Konzept anhand mehrerer Beispiele vorgestellt werden.

7.3.1 Fibonacci-Folge

Die Fibonacci-Folge wird rekursiv wie folgt definiert:

$$a_0 = 1, \quad a_1 = 1, \quad \forall n \geq 2 : a_n = a_{n-1} + a_{n-2}.$$

Dies lässt sich einfach rekursiv implementieren per

```
function f = fibonacci(n)
    if n <= 1
        f = 1;
    else
        f = fibonacci(n - 1) + fibonacci(n - 2);
    end
end
```

Dies ist zwar elegant, aber sehr ineffizient. Man kann zeigen, dass die Laufzeit $O(2^n)$ ist. Warum dies so ist, sieht man leicht:

- Für die Berechnung von `fibonacci(n - 1)` wird ebenfalls `fibonacci(n - 2)` berechnet, womit `fibonacci(n - 2)` bereits zweimal aufgerufen wird.

- Der Aufruf `fibonacci(n - 3)` wird im Aufruf von `fibonacci(n - 2)` sowie im Aufruf von `fibonacci(n - 1)` getätigt, womit `fibonacci(n - 3)` bereits dreimal aufgerufen wird im Laufe der Berechnung von `fibonacci(n - 3)`.
- Genauso überlegt man sich, dass `fibonacci(n - 4)` bereits fünfmal und `fibonacci(n - 5)` sogar achtmal aufgerufen wird.

Wenn man genau hinschaut sieht man, dass `fibonacci(n - k)` genau `fibonacci(k)`-fach aufgerufen wird. Damit wird `fibonacci` zur Berechnung von `fibonacci(n)` genau $\sum_{k=1}^n a_k$ mal aufgerufen, und $\sum_{k=1}^n a_k = O(2^n)$. Die Fibonacci-Folge lässt sich jedoch viel effizienter berechnen:

```
function f = fibonacci(n)
    if n <= 1
        f = 1; % a0 = a1 = 1
    else
        a = 1; % dies ist ak (fuer k = 1)
        b = 1; % dies ist ak-1 (fuer k = 1)
        for k = 2:n
            % hier ist a = ak-1, b = ak-2
            t = a + b;
            b = a;
            a = t;
            % hier ist a = ak, b = ak-1
        end
        f = a; % dies ist nun an
    end
end
```

In diesem Fall ist die Laufzeit $O(n)$ – was wesentlich weniger ist als $O(2^n)$!

Es soll darauf hingewiesen werden, dass das Problem noch schneller gelöst werden kann. Dies basiert auf der Darstellung

$$a_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

von A. de Moivre und J. P. M. Binet.

7.3.2 Binomialkoeffizienten

Wir haben in Aufgabe 5.2 gesehen, dass die Berechnung eines Binomialkoeffizienten $\binom{n}{k}$ über die Formel

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \prod_{\ell=1}^k \frac{n-\ell+1}{\ell}$$

sehr ungünstig ist, sobald n und k gross sind, da Fließkommazahlen nur eine begrenzte Genauigkeit haben.

Dieses Problem lässt sich umgehen, indem man die Additionsformel

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

verwendet. Dies lässt sich wie bei der Fibonacci-Folge sehr einfach rekursiv implementieren, ist jedoch ebenfalls äusserst ineffizient.

Dabei ist es sehr einfach, in $O(n^2)$ Operationen den ganzen Vektor

$$\left(\binom{n}{0}, \binom{n}{1}, \dots, \binom{n}{n}\right)$$

zu berechnen! Dies wird durch folgende, sehr einfache Funktion erledigt:

```
function B = binomialreihe(n)
    B = [1];
    for l = 1 : n
        B = [0, B] + [B, 0]; % beachte, dass  $\binom{\ell-1}{-1} = \binom{\ell-1}{\ell} = 0$  ist
    end
end
```

7.3.3 Längste gemeinsame Teilsequenz

Gegeben seien zwei Vektoren \mathbf{v} und \mathbf{w} . Eine *Teilsequenz* von \mathbf{v} ist ein Vektor \mathbf{x} der Länge k , so dass es eine Folge $i_1 < i_2 < \dots < i_k$ gibt mit $v_j = w_{i_j}$ für $1 \leq j \leq k$. Etwa ist $[1, 5, 6, 9]$ eine Teilsequenz von $[1, 2, 3, 4, 5, 6, 7, 8, 9]$. Eine längste gemeinsame Teilsequenz von \mathbf{v} und \mathbf{w} ist ein Vektor \mathbf{x} der Länge k , der Teilsequenz sowohl von \mathbf{v} und \mathbf{w} gibt, so dass es keine gemeinsame Teilsequenz von \mathbf{v} und \mathbf{w} der Länge $k + 1$ gibt.

Solche Teilsequenzen zu finden ist ein klassisches Problem aus der Informatik, was etwa auch in der Bioinformatik wichtig ist. Eine schnelle Möglichkeit, es zu lösen, ist ein Ansatz mit der dynamischen Programmierung.

Dazu schreibe $\mathbf{v} = [v_1, \dots, v_a]$ und $\mathbf{w} = [w_1, \dots, w_b]$. Jetzt unterscheidet man zwei Fälle:

1. Es gilt $v_a = w_b$. Ist dann \mathbf{x} eine längste gemeinsame Teilsequenz von $[v_1, \dots, v_{a-1}]$ und $[w_1, \dots, w_{b-1}]$, so ist $[\mathbf{x}, v_a]$ eine längste Teilsequenz von \mathbf{v} und \mathbf{w} .
2. Es gilt $v_a \neq w_b$. Sei \mathbf{x} eine längste gemeinsame Teilsequenz von \mathbf{v} und $[w_1, \dots, w_{b-1}]$, und sei \mathbf{y} eine längste gemeinsame Teilsequenz von $[v_1, \dots, v_{a-1}]$ und \mathbf{w} . Sei \mathbf{z} der längere von den beiden Vektoren \mathbf{x} und \mathbf{y} , bzw. irgendeiner falls sie gleichlang sind. Dann ist \mathbf{z} eine längste Teilsequenz von \mathbf{v} und \mathbf{w} .

Wie man sieht, kann man dies rekursiv implementieren: in jedem Schritt wird der Wert $a + b$ kleiner, und sobald $a = 0$ oder $b = 0$ ist, kann man sofort eine Lösung hinschreiben (nämlich der leere Vektor $[\]$).

Aufgabe 7.5: Implementiere den Algorithmus und wende ihn auf die Vektoren $\mathbf{v} = [1, 9, 2, 8, 3, 7, 4, 6, 5]$ und $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ an.

7.4 Vektorisierung

Vektorisieren ist ein Konzept, welches auf Architekturen wichtig ist, in denen Vektoren ähnlich effizient verarbeitet werden können wie Einzeldaten. Dies ist zum Beispiel bei vielen modernen CPUs der Fall. Insbesondere auf modernen Grafikkarten ist dies der Fall, jedoch ebenso bei Grossrechnern und Supercomputern.

Da MATLAB nativ mit Vektoren und Matrizen arbeitet, bringt dieses Konzept in MATLAB ebenfalls Effizienzvorteile. Betrachte etwa die folgenden beiden Programme zur Berechnung von $\sum_{k=1}^n \frac{1}{k^2}$:

```
s = 0;
for k = 1 : n
    s = s + 1 / k^2;
end
```

```
s = sum(1 ./ (1 : n).^2);
```

Um etwa in MATLAB die Zeit zu messen, die ein Befehl oder eine Folge von Befehlen benötigt, verwendet man den **tic**- und den **toc**-Befehl:

```
tic
<Befehle>
toc
```

Wenn man damit die obigen beiden Programme für $n = 100\,000\,000$ vergleicht, sieht man, dass die **for**-Schleife 5.0 Sekunden benötigt, während die Vektor-Version bereits nach 1.5 Sekunden fertig ist.

Eine anderer Fall ist das Ausrechnen eines Skalarproduktes $\langle v, w \rangle = \sum_{i=1}^n v_i \bar{w}_i$. Dies kann man etwa als **sum(v .* conj(w))** schreiben oder auch einfacher als $\mathbf{v} * \mathbf{w}'$, falls \mathbf{v} und \mathbf{w} Zeilenvektoren sind.

Weitere Beispiele und Hinweise findet man auf der Seite

<http://www.mathworks.com/support/tech-notes/1100/1109.html>.

8 Toolboxen

Eine *Toolbox* ist eine Erweiterung von MATLAB die aus einer Sammlung von `.m`-Dateien, aus Dokumentation und teilweise auch aus Binärdateien bestehen.

Sie werden normalerweise in MATLAB eingebunden, indem man die entsprechenden Verzeichnisse der Toolbox in den MATLAB-Pfad übernimmt. Der MATLAB-Pfad kann unter `Set Path...` im **File**-Menü eingesehen und verändert werden.

Alle installierten Toolboxen sind bei der MATLAB-Installation an der Universität Zürich bereits im Pfad vorhanden.

Eine solche Toolbox ist die Symbolic Math Toolbox, die wir in folgendem Unterabschnitt vorstellen wollen.

8.1 Symbolic Math Toolbox

Die *Symbolic Math Toolbox* fügt MATLAB den Datentyp `sym` hinzu. Dieser kann einen symbolischen Ausdruck speichern. Dies erlaubt MATLAB, mit Ausdrücken ähnlich wie in Computer-Algebra-Systemen wie MAPLE, MATHEMATICA oder DERIVE umgehen zu können.

Man kann Unbestimmte mit dem `syms`-Befehl erzeugen. Weiterhin kann man mit “symbolischen Zahlen” arbeiten; dies sind Zahlen, die den Typ `sym` haben:

```
>> syms x
>> x^2 + 1;
    ans = x^2 + 1
>> y = sym(2)
    y = 2
```

Falls eine symbolische Zahl eingegeben werden soll, die zu gross oder zu genau für `double` ist, muss man sie als Text angeben:

```
>> 92845982143190231923.98345928431093812930213091823
    ans = 9.284598214319024e+19
>> sym('92845982143190231923.98345928431093812930213091823')
    ans = 92845982143190231923.98345928431093812930213091823
```

Solche Zahlen werden wesentlich langsamer verarbeitet als etwa ein Wert vom Typ `double`, jedoch hat man auch nicht die Einschränkungen eines `double` an Präzision.

```
>> sin(pi)
    ans = 1.224646799147353e-16
>> y = sym('pi')
    y = pi
>> sin(y)
    ans = 0
```

Man kann auch `doubles` nach `sym` konvertieren; MATLAB versucht dann, diese Zahl als Bruch oder bekannte Konstante (wie π) darzustellen:

```
>> 9432/123981
    ans = 0.076076172961986
>> sym(ans)
    ans = 3144/41327
```

Man kann wie in anderen Computer-Algebra-Systemen Ausdrücke vereinfachen:

```
>> rho = sym('(1+sqrt(5))/2');
>> f = rho^2 - rho - 1
    f = (5^(1/2)/2 + 1/2)^2 - 5^(1/2)/2 - 3/2
>> simplify(f)
    ans = 0
```

Genauso kann man etwas ausmultiplizieren:

```
>> syms x y
>> f = (x + y)^10
    f = (x + y)^10
>> expand(f)
    ans = x^10 + 10*x^9*y + 45*x^8*y^2 + 120*x^7*y^3 + 210*x^6*y^4 ...
```

```

+ 252*x^5*y^5 + 210*x^4*y^6 + 120*x^3*y^7 + 45*x^2*y^8 ...
+ 10*x*y^9 + y^10

```

Man kann auch Polynome oder rationale Funktionen faktorisieren:

```

>> factor((x^10 - 1)/(x - 1)^2)
ans = ((x + 1)*(x^4 + x^3 + x^2 + x + 1)*(x^4 - x^3 + x^2 - ...
      x + 1))/(x - 1)

```

Ersetzen von Variablen durch Werte macht man mit `subs`:

```

>> f = x^3 - x + 1;
>> subs(f, x, 10)
ans = 991

```

Man kann auch Ableiten und Integrieren:

```

>> diff(x^2 + 2 * x + 1)
ans = 2*x + 1
>> diff((x + y)^10, y, 2) % zweifach nach y ableiten
ans = 90*(x + y)^8
>> int(sin(x)^2)
ans = x/2 - sin(2*x)/4
>> int(x, 0, sqrt(pi)/2)
ans = 31859534503965572279823959492121/40564819207303340847894502572032
>> int(x, 0, sqrt(sym(pi)/2))
ans = pi/4

```

Weiterhin kann man Gleichungen lösen; allerdings nur algebraische Gleichungen (Nullstellen von Polynomen) oder Differenzialgleichungen.

```

>> solve(x^2 - 1)
ans = -1
      1
>> [x,y] = solve(x * y - 1, x + y - 10)
x = 2*6^(1/2) + 5
   5 - 2*6^(1/2)
y = 5 - 2*6^(1/2)
   2*6^(1/2) + 5
>> y = dsolve('D2y==4*y', 'y(0)=1') % Anfangswertproblem: y'' = 4y
y = C9/exp(2*t) - exp(2*t)*(C9 - 1) % y(0) = 1
>> diff(y, t, 2)
ans = (4*C9)/exp(2*t) - 4*exp(2*t)*(C9 - 1)

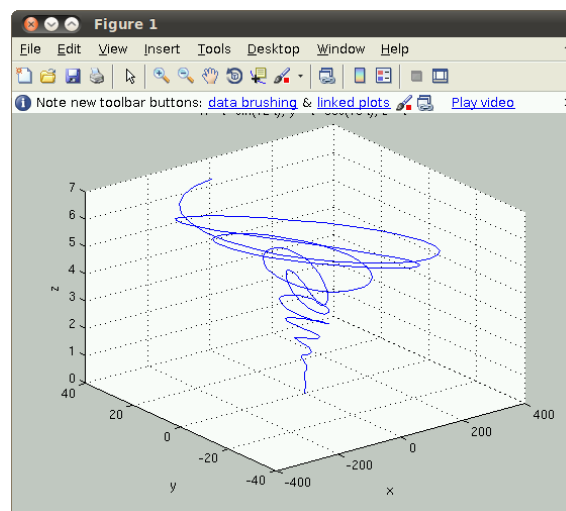
```

Man kann auch symbolische Funktionen plotten:

```

>> syms t
>> ezplot3(t^3 * sin(12*t), t^2 * cos(10*t), t);

```



Viele weitere Informationen zur Toolbox findet sich in der Online-Hilfe.