Friedrich-Schiller-Universität Jena

seit 1558

Physikalisch-Astronomische Fakultät

WS 2008/2009

# Computational Physics I

## Zusatzskript: Der Einstieg in das Programmieren

Prof. Dr. Thomas Pertsch

von Simon Stützer

Stand: 18. Oktober 2008

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

In der modernen Forschung sind Computer unerlässlich geworden. Indem sie komplizierte Berechnungen oder Simulationen ausführen, verschaffen sie uns neue Erkenntnisse, sodass in der Wissenschaft nahezu kein Weg an ihnen vorbei führt. Ob bei der Auswertung gewaltiger Datenmengen, dem Abarbeiten umfangreicher Algorithmen oder der Visualisierung bestimmter Szenerien - speziell in der Physik spielen Computer eine stetig wachsende Rolle. Damit der Computer jedoch Arbeiten ausführt, gilt es dessen Sprache zu lernen - eine Programmiersprache. Hier gibt es, wie bei echten Sprachen auch, eine große Auswahl an Programmiersprachen. Die heute meist verbreitetsten Sprachen sind Java, C und C++. Allen Programmiersprachen, ja allen Sprachen, ist jedoch eines gemeinsam: man benötigt ein umfangreiches Repertoire an Vokabeln und genaue Kenntnisse der Grammatik, die sich in den gebräuchlichen Programmiersprachen mitunter nur gering unterscheiden. In der Vorlesung Computational Physics I wollen wir die Sprache MATLAB kennen lernen. Diese eignet sich in ihrer Art besonders gut bei numerischen Berechnungen.
Letztlich ist der Weg zum Physiker mit weitreichenden Programmierkenntnissen weit und lässt sich nicht von heute auf morgen beschreiten. Geduld, Fleiß und Übung sind nunmal Vorraussetzungen zum Erlernen einer Sprache. Ist man jedoch gewillt, die nötige Disziplin aufzubringen, eröffnen sich rasch neue Möglichkeiten und die Faszination des Programmierens lässt einen nicht mehr los.

## 1.2 Programmierumgebung

Viele Programmiersprachen setzen eine Programmierumgebung (Editor, Compiler, Bibliotheken etc.) voraus, deren Einrichtung bereits die erste Hürde darstellt. Die zu erlernenden Programmierkenntnisse werden in diesem Skript mit Beispielen in MATLAB versehen. MATLAB ist eine kommerzielle, plattformunabhängige Software des Unternehmens The MathWorks, Inc. Eine lizensierte Version der Software ist jedoch auf allen Rechnern im Computerpool der PAF vorhanden. MATLAB stellt ein komplette Programmierumgebung dar. So liegt die Einstiegsschwelle verglichen mit anderen Programmiersprachen genau genommen bei "Null". Starten wir das Programm mit dem Konsolenbefehl *matlab* oder über das Desktop-Icon so können wir im rechten **Command-Window** sofort mit dem Programmieren beginnen.

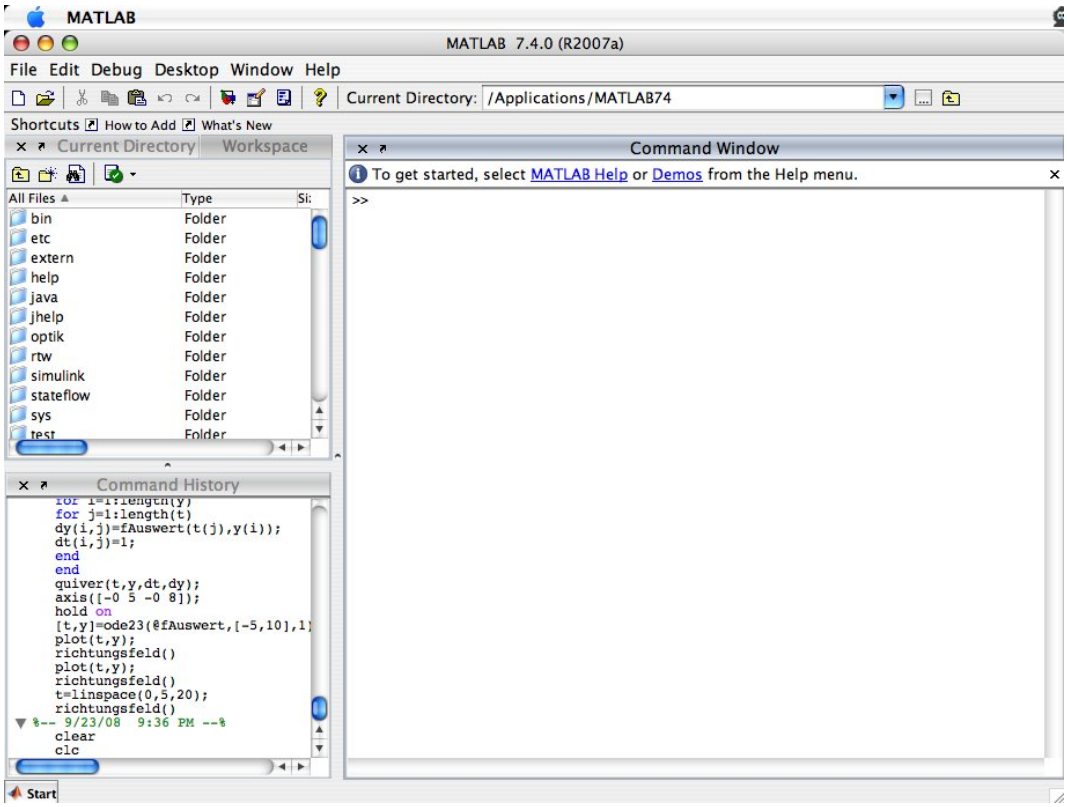



Abbildung 1.1: MATLAB-Fenster in der Standard Anordnung. Links das **Current Directory** und die **Command History**. Rechts das große **Command Window**

## 1.3 Hallo Welt

Im Lernprozess des Programmierens gilt oft *"Learning by Doing!"*. Man sollte sich demnach nicht entmutigen lassen, wenn die Lösung eines Problems oder das Finden eines Fehlers im Programm einige Zeit dauert. Man kann sich jedoch sicher sein, dass viele Zeilen geschriebener und gelesener Programmkode die Kenntnisse einer Programmiersprache rasch wachsen lassen.
Bevor wir jetzt richtig loslegen ist ein `"Hallo-Welt"`-Programm unvermeidbar. In vielen Programmiersprachen nutzt man dies zu Testzwecken der Programmierumgebung, etwa des Compilers. Hier soll es lediglich der letzten Motivation dienen, bevor wir im kommenden Kapitel erste kleine Programme schreiben.

**Kodeschnipsel 1**

```
1 %Programm Hallo-Welt in MATLAB
2
3 disp('Hallo Welt!');
```

Wie man sieht erreicht man eine Bildschirmausgabe in MATLAB mit einem Einzeiler. Die erste Zeile ist lediglich ein Kommentar der vom Programm nicht ausgeführt wird.

Tippen Sie nun noch kurz `why` in das **Command-Window** und schauen sie was passiert.
Viel Spaß!

# Kapitel 2

# Grundlagen

## 2.1 Algorithmen

Zunächst müssen wir uns einen genauen Überblick verschaffen, was es heißt zu programmieren. In der Sprache des Alltags kommt es oft zu mehrdeutige Anweisung zwischen Menschen, die jedoch richtig befolgt werden. Der Mensch hat dabei die Möglichkeit, durch seine Erfahrung, Umgebung, die Betonung der Aufforderung oder schlicht seinen logischen Verstand richtig zu reagieren.
Bei Programmen verhält es sich anders. Für eine korrekte Ausführung eines Programms vom Computer sind drei Dinge unverzichtbar:

- eine konkrete Aufgabenstellung
- eine Algorithmierung des Lösungsweges
- ein fehlerfreie Formulierung des Lösungsweges als Programm.

Die Aufgabe der **konkreten Aufgabenstellung** erfordert dabei eine analytische Herangehensweise an ein bestimmtes Problem, zur **fehlerfreie Formulierung als Programm** bedarf es Kenntnisse der Sprachelemente und der Grammatik einer Programmiersprache. Zusammen bezeichnet man dies als Syntax und wir wollen die Syntax am Beispiel von MATLAB später kennenlernen. Der zentrale Begriff ist jedoch **Algorithmierung** oder **Algorithmus**. Wir wollen darunter eine Art exakte Durchführbetimmung von bekannten, auszuführenden Operationen verstehen, die in ihrer Gesamtheit schließlich zu einem Ziel, der Lösung eines Problems führen. Ein Kochrezept ist in gewisser Hinsicht ein Algorithmus, wenn die einzelnen Schritte (Operationen) nur exakt genug beschrieben sind und weitere genaue Algorithmen für Teilaufgaben wie beispielsweise ”kochen”, ”rühren” oder ”backen” bestehen.

Unter einem Algorithmus versteht man eine genau definiert Abfolgen von Anweisungen die in endlicher Zeit ausführbar sind und zu einer Lösung eines Problems führen. Ein Algorithmus ist dabei **determiniert**, **finit** und **effektiv**.

- Determiniertheit: Ein Algorithmus muss bei gleichen Anfangsbedingungen zu dem selben Ergebnis führen.
- Finitheit: Die Beschreibung eines Algorithmus muss endlich sein. D.h. der Quelltext eines Programms besitzt eine endliche Länge und das Programm benutzt endlich viel Speicher.
- Effektivität: Die Auswirkung jeder Anweisung eines Algorithmus muss eindeutig festgelegt sein. Dabei führt eine guter Programmierstil zur effektiveren Umsetzung von Anweisungen und verkürzt so z.B. die Laufzeit eines Programms.

Eine weiter Eigenschaft von Algorithmen könnte die **Terminierung** sein. Dabei nennt man einen Algorithmus terminierend, wenn er bei seiner Anwendung nach endlich vielen Schritten beendet ist und ein Resultat liefert. Der Begriffe ist nicht mit dem der Finitheit zu verwechseln. Es sind Algorithmen denkbar, etwa zur Steuerung von "nichtabbrechenden"Vorgängen (z.B. in der chemischen Industrie), die durch eine endliche (finit) Beschreibung potentiell endlos laufen (Endlosschleife).

## 2.2 Datentypen

Ein Datentyp legt die Art der Werte fest, die eine Variable annehmen kann. Daraus ergeben sich unterschiedliche Operationen die auf diese Variable angewandt werden können. Jede Variable in einem Programm muß genau einem Datentyp zugeordnet werden. Diese Zuordnung bezeichnen wir als **Datentypdeklaration**. MATLAB unterscheidet skalare Datentypen und strukturierte Datentypen. Letztere setzen sich aus den skalaren Typen zusammen.

Mit einem Datentyp wird also festgelegt, welchen Inhalt ein Behälter, eine Variable haben darf. Der Typ `int` beispielsweise akzeptiert nur ganze Zahlen. Dabei wird in den meisten Programmiersprachen noch zwischen `int`, `longint`, `shortint` usw. differenziert. Die Unterschiede dabei liegen einzig im Speicher der für eine Variable freigegeben wird und haben somit Auswirkungen auf den Wertebereich einer Variablen. Im Gegensatz zu Programmiersprachen wie `C` oder `C++` nimmt MATLAB die Deklaration des Datentyps automatisch vor. Datentypen können jedoch auch in MATLAB gezielt festgelegt werden.

**Kodeschnipsel 2**

```
1 x = int8(-66.325)   % liefert 8-bit Integer: -66
2 y = int8(-666.325)  % liefert 8-bit Integer: -128
3                     % -666.325 liegt nicht im Wertebereich des Datentypes
4 z = int16(-666.325) % liefert 16-bit Intger -666
5
6 class(z)            % liefert ans =int16, also den Datentyp
7
8 l = int32(pi)      %  liefert 3
```

Weitere Datentypen sind beispielsweise `uint8`, `single`, `double` oder `char`. Sie liefern ein vorzeichenlosen 8-Bit Integer-Wert, Fließkommazahlen verschiedener Speichergröße oder sind für alphanumerische Zeichen(ketten) geeignet. Der sogenannte logische Datentyp ist `boolean`. Dieser kann nur die Werte `0` und `1` bzw. `true` und `false` tragen. Später werden wir sehen, dass dieser Typ bei der Steuerung von Kontrollstrukturen eine zentrale Rolle spielt.

### 2.2.1 Vektoren und Arrays

Für die Lösung und Simulation physikalischer Probleme ist ein weiterer Datentyp von großer Bedeutung. Dies ist der abgeleitete Datentyp des Feldes oder Arrays der sich aus mehreren Einträgen eines anderen Datentyps (z.B. integer) zusammensetzt. Ob in der Bildverarbeitung, der numerischen Optik oder Elektrodynamik, die effiziente Umsetzung eines Problems ist oft mit Arrays (Matrizen) zu realisieren. ”Der wichtigste Datentyp (früher: der einzige), den Matlab unterstützt, sind Matrizen mit komplexen Elementen. In ihm sind Skalare, ganze Zahlen, Vektoren, relle Matrizen, usw. als Spezialfälle enthalten.” (vgl. Einfürung in MATLAB, ETH Zürich) Einige Beispiele solle zeigen, wie man in MATLAB Felder erstellt.

**Kodeschnipsel 3**

```
1  a = [1,3,4]   %liefert Zeilenvektor
2  a = [1 3 4]   %liefert Zeilenvektor
3
4  b = [3;4;7]   %liefert Spaltenvektor
5  m= [1 2 3;4 5 6;7 8 9] %liefert Matrix
6
7  a(i)   %liefert i-ten Eintrag des Vektors a z.B. a(2)-> 3, b(3)->7
8  m(i,j) %liefert Eintrag der i-ten Zeile und j-ten Spalte der Matrix m z.B. m(2,3) -> 6
```

Zeile 7 und 8 verdeutlichen nocheinmal, wie man einzelne Elemente einer Matrix anspricht.

## 2.3 Programmierstil

Der Programmierstil den wir hier erlernen wird als **imperatives Programmieren** (oft auch prozedurales Programmieren) bezeichnet. Wesentliche Merkmale sind dabei das Speichern von Werten in Variablen und ein chronologisches Abarbeiten der Befehle eines Programms. Eine andere Programmierstil stellt beispielsweise die **objektorientierte Programmierung** dar.
Im Folgenden geht es um das imperative Programmieren, für Einsteiger sicherlich einfacher zu erlernen, sodass wir noch einige Bemerkungen dazu machen werden. Ein guter Programmierstil sollte folgendes berücksichtigen:

- Der Quellkode (Programmkode) sollte durch Einrücken und Leerzeilen übersichtlich gestaltet werden.
- Abschnitte des Programms sollten im Quellkode kommentiert sein. Kommentare werden dabei vom Programm nicht ausgeführt und dienen dem Programmierer nur zu dessen Verständlichkeit. Kommentare in MATLAB beginnen mit einem Prozentzeichen `%` und enden am Zeilenende.
- Oft benötigte Unterprogramme (Prozeduren) sollten in so-genannten Funktionen (vgl. Kapitel 5 Seite 16) ausgegliedert werden.

# Kapitel 3

# elementare Operationen

## 3.1 arithmetische Operationen

Ein kleiner Überblick über logische Operatoren in MATLAB, sowie in den meisten Programmiersprachen auch, soll folgende Tabelle geben.

| Operator | Operation | Bedeutung | Priorität |
|---|---|---|---|
| `^` | `x^y` | Potenzierung $x^y$ | 4 |
| `-` | `-x` | Negation $-x$ | 3 |
| `+` | `+x` | Vorzeichen $+x$ | 3 |
| `*` | `x*y` | Multiplikation $x \cdot x$ | 2 |
| `/` | `x/y` | Division $\frac{x}{y}$ | 2 |
| `\` | `x \y` | Division $\frac{y}{x}$ | 2 |
| `+` | `x+y` | Addition $x + y$ | 1 |
| `-` | `x-y` | Subtraktion $x - y$ | 1 |

Operationen der höheren Priorität haben dabei Vorrang (z.B. "Punktrechnung vor Strichrechnung"). Bei dem Ausführen von von Operationen mit gleicher Priorität gilt "Linksassoziativität", d.h. die Operatoren werden von links nach rechts abgearbeitet. Dabei ist es durch Klammerung jedoch möglich, die Priorität zu durchbrechen.

### 3.1.1 Zuweisungsoperator

Unbemerkt haben wir weiter oben schon einen weiteren Operator benutzt - der Zuweisungsoperator =. Der Operator = weist einer Variable eindeutig einen Wert zu. Dieser Wert kann auch das Ergebnis eines Ausdrucks z.B. der Rechenoperation `2+5` oder `x+y` sein.

Die Syntax einer **Wertzuweisung** mit Hilfe des "="-Operators hat immer die Form

⟨Variablenname⟩=⟨Ausdruck⟩

**Der Operator ”=” ist grundsätzlich verschieden zu dem im Alltag bekannten ”ist gleich". In vielen Programmiersprachen wird der logische Operator ”ist-gleich"durch ”==” implementiert. Die Wertzuweisung durch ”=” bedeutet hingegen: ”Lege das Ergebnis des Ausdrucks auf einem Speicherplatz ab, der unter der Adresse des Variablennamens zu erreichen ist” Die Größe des Speicherplatzes richtet sich dabei nach dem Datentyp des Ausdrucks.**

Unachtsamkeit bei der Verwendung beider Operatoren haben schon so machen Programmierer kostbare Zeit geraubt. Der ”==” - Operator ist ein erstes Beispiel für einen logischen Operator wie wir sie im folgenden Abschnitt behandeln.

### 3.1.2 Erstes Programm: Umrechnung von Kugelkoordinaten

Wir wollen einen Punkt, der in Kugelkoordinaten $(r, \vartheta, \varphi)$ gegeben ist, in kartesischen Koordinaten darstellen. Mit der Umrechnung

$$x = r \cdot \sin\vartheta \cos\varphi$$
$$y = r \cdot \sin\vartheta \sin\varphi$$
$$z = r \cdot \cos\vartheta$$

haben wir somit eine exakte Problemstellung und können mit dem Programmieren beginnen. Dazu öffnen wir ein neues m.-File z.B. über die Toolbar oder mit der Tastenkombination ”ctrl-n".

**Kodeschnipsel 4**

```
% Programm Kugelkoordinaten (sc.m)
% Bestimmung der kartesischen Koordinaten
% bei gegebenen Kugelkoordinaten

clc % Command-Window loeschen
disp('Umrechnung: Kugelkoordinaten -> Kartesische Koordinaten')
disp('=======================================================')
fprintf('Eingabe: Kugelkoordinaten: \n')
r = input('Radius (in cm) = ');
phi = input('PHI (in Grad) = ');
theta = input('THETA (in Grad) = ');

% Umrechnung in Bogenmass
phi = pi * phi / 180.0;
theta = pi * theta / 180.0;
% Winkelberechnung
sig = sin(theta);
x = sig * cos(phi);
y = sig * sin(phi);
% Kartesische Koordinaten (x,y,z)
x = r * x; y = r * y;
z = r * cos(theta);

fprintf('\nAusgabe: Kartesische Koordinaten: \n')
fprintf('x = %10.5f cm \n', x)
fprintf('y = %10.5f cm \n', y)
fprintf('z = %10.5f cm \n', z)
% Programmende
```

Das m.-File speichert man einfach unter einem Namen z.B. "sc.m" (spherical coordinates). Beinhaltet im MATLAB-Fenster links das '**Current Directory**' die soeben gespeicherte Datei, gibt man nun nur noch "sc" (ohne Dateiendung!) im **Command-Window** ein und das Programm wird ausgeführt.
Es sollen noch kurz einige verwendeten Befehle geklärt werden. Es genügt zu wissen, dass `disp, fprintf` eine Bildschirmausgabe erzeugen wobei `\n` einen Zeilenumbruch bewirkt. Der Befehl `input` erwartet eine Benutzereingabe, womit wir die Zuweisung zu den gegebenen Kugelkoordinaten realisieren. Die Konstante `pi` ist wie auch die Funktionen `sin` und `cos` in MATLAB implementiert und man kann auch zahlreiche weitere Mathematische Funktionen zurückgreifen.

## 3.2 logische Operationen

Logische Operatoren spielen bei der Programmierung eine wichtige Rolle. Sie liefern in sogenannten **logischen Ausdrücken** eine der beiden bool'schen Werte `true` oder `false` und sind somit ein Hauptbestandteil von **Kontrollstrukturen**. Doch dazu mehr im nächsten Kapitel. An dieser Stelle soll zunächst eine Auflistung ausreichen.

### 3.2.1 Vergleichsoperatoren

| Name | **Symbol** | **Bedeutung** |
|---|---|---|
| eq | == | gleich |
| ne | ∼= | ungleich |
| ge | >= | größer gleich |
| gt | > | größer als |
| le | <= | kleiner gleich |
| lt | < | kleiner als |

Dies in MATLAB gültigen Operatoren sind in vielen Programmiersprachen analog. Lediglich der Operator ∼= unterscheidet sich. In `C`, `C++`, `Java` oder auch `PHP` wird das logische "ungleich" durch `!=` implementiert.

### 3.2.2 Verknüpfungsoperatoren

| Name | **Symbol** | **Bedeutung** | **Priorität** |
|---|---|---|---|
| and | && | logisch UND | 1 |
| or | ‖ | logisch ODER | 1 |
| not | ∼ | logisch NICHT | 2 |
| xor | `xor` | exklusives ODER | 1 |

# Kapitel 4

# Kontrollstrukturen

## 4.1 Bedingte Anweisung

Nahezu alle Programmiersprachen erlauben es, das Ausführen bestimmter Programmabschnitte an Bedingungen zu koppeln. Es wird zunächst geprüft ob diese Bedingung wahr ist und erst dann der entsprechende Programmteil ausgeführt. Die einfachste bedingte Anweisung hat dabei die Struktur

**einseitig bedingte Anweisung**

```
if    ⟨Bedingung⟩
      ⟨Anweisung⟩
end
```

**Übersetzung:** "Wenn die *Bedingung* den boole'schen Wert `true` oder `1` zurückgibt, führe die *Anweisung* aus."

Es besteht jedoch auch die Möglichkeit, eine oder gar mehrere Alternative anzugeben für den Fall, dass eine Bedingung nicht eintritt.

**zweiseitige bedingte Anweisung**

```
if      ⟨Bedingung⟩
        ⟨Anweisung 1⟩
else
        ⟨Anweisung 2⟩
end
```

**Übersetzung:** "Wenn die *Bedingung* den boole'schen Wert `true` oder `1` zurückgibt, führe die *Anweisung 1* aus, in allen anderen Fällen führe *Anweisung 2* aus."

**mehrseitig bedingte Anweisung**

```
if        ⟨Bedingung 1⟩
          ⟨Anweisung 1⟩
elseif    ⟨Bedingung 2⟩
          ⟨Anweisung 2⟩
...
elseif    ⟨Bedingung n⟩
          ⟨Anweisung⟩
end
```

**Übersetzung:** "Wenn die *Bedingung 1* den boole'schen Wert `true` oder `1` zurückgibt, führe die *Anweisung 1* aus. Wenn die *Bedingung 2* den boole'schen Wert `true` oder `1` zurückgibt, führe die *Anweisung 2* aus usw. . Beende die Kontrollstruktur jedoch sobald du die Anweisung zur von oben gesehen ersten "wahren" Bedingung ausgeführt hast und setze das Programm nach dem Ende der If-Abfrage fort."

**Hinweise:**

- Sobald von oben beginnend die erste Bedingung erfüllt ist, wird die nachstehende Anweisung ausgeführt und die Kontrollstruktur nicht weiter durchgegangen. Eventuell später auftauchende "wahre" Bedingungen werden samt ihren Anweisungen nicht mehr berücksichtig.
- Es ist auch möglich bei einer mehrseitige bedingte Anweisung am "Ende" der Anweisung eine alternative Anweisung mit `else` anzugeben, die ausgeführt wird, wenn alles vorherigen Bedingungen den Wert `false` liefern.
- Hat man die Syntax einer If-Abfrage verstanden, ist es ohne Probleme möglich diese auch in anderen Programmiersprachen zu lesen. MATLAB hat dabei eine eher knappe, möglicherweise unübersichtliche Syntax. Programmiersprachen wie `C`, `C++`, `Java` oder `PHP` erzielen die gleiche Wirkung durch: `if(`⟨Bedingung⟩`){`⟨Anweisung1⟩`}else{`⟨Anweisung2⟩`}` usw. Dabei ist die Syntax durch die Benutzung von Klammern etwas übersichtlicher.

### 4.1.1 Beispiel: If-Konstruktor

**Kodeschnipsel 5**

```
1  % Programm: minimum.m - das Minimum dreier ganzer Zahlen
2
3  a = input('a = ');
4  b = input('b = ');
5  c = input('c = ');
6
7  if a<b && a<c
8      min = a;
9  elseif b<c
10     min = b;
11 else
12     min = c;
13 end
14
15 fprintf('Minimum = %i',max)
```

**Aufgabe 1**

*Schreiben Sie ein Programm, welches eine Zahl einliest und daraufhin angibt ob die Zahl gerade und ob die Zahl negativ ist. Tipp: geben sie zunächst* `help mod` *im Command-Window ein.*

**Aufgabe 2**

*Die Quadratische Gleichung* $x^2 + a \cdot x + b = 0$ *hat sowohl Lösungen im Reellen als auch im Komplexen. Schreiben sie ein Programm, das bei einzulesenden Konstanten a und b die Lösung berechnet, wenn diese reell ist. Ist die Lösung komplex, soll der Benutzer entscheiden, ob er den Real- und Imaginärteil der Lösung ausgegeben bekommt oder den Betrag der komplexen Zahl.*

**Aufgabe 3**
*Betrachten sie folgenden kurzen Quellkode:*

```
1 a=3; b=2;
2 if a+b=6
3     disp('Die Summe ist 6.');
4 else
5     disp('Die Summe ist ungleich 6.');
6 end
```

*Was wird passieren?*

## 4.2 Fallunterscheidung

Manchmal möchte man eine große Anzahl an Fällen unterscheiden. Die mehrseitig bedingte Anweisung ist eine Möglichkeit um dieses Vorhaben zu realisieren - eine schlechte. Schnell wird der Quelltext unübersichtlich und die Effektivität des Programms leidet. Eine bessere Variante ist der `switch`-Konstruktor, wie er in vielen Programmiersprachen existiert. In MATLAB hat dieser die allgemeine Form

**switch-Konstrukt**

| | |
|---|---|
| `switch` | ⟨Ausdruck⟩ |
| `case` | ⟨Konstante 1⟩ |
| | ⟨Anweisung 1⟩ |
| `case` | ⟨Konstante 2⟩ |
| | ⟨Anweisung 2⟩ |
| ... | |
| `otherwise` | |
| | ⟨Anweisung $n$⟩ |
| `end` | |

**Übersetzung:** Nehme den *Ausdruck* und vergleiche ihn mit *Konstante 1* danach mit *Konstante 2* danach mit *Konstante 3* usw.. Sobald dabei der Vergleich ein `true` oder `1` liefert führe die nachstehende Anweisung aus und beende die Kontrollstruktur. Führt kein Vergleich zu einem "wahr" so führe die *Anweisung n* aus und beende die Kontrollstruktur.

Die einzelnen Fälle ("case") werden der Reihenfolge noch abgearbeitet. Stimmt der Wert des Ausdrucks mit dem einer Konstanten überein, so werden die entsprechenden Anweisungen ausgeführt. Danach ist die switch-Anweisung beendet! Dies ist eine MATLAB-spezifische Eigenschaft. In anderen Programmiersprachen bedarf es der `break`-Anweisung um das `switch`-Konstrukt zu beenden. Der optionale `otherwise`-Fall ist dabei gleichbedeutend dem `else`-Befehl der `If`-Anweisung. Einem `case` lassen sich durch Klammerung auch mehrere Konstanten zuordnen wie im abschließenden Beispiel zu sehen ist.

**Kodeschnipsel 6**

```
1  method = 'Bilinear';
2
3  switch lower(method)
4  case {'linear','bilinear'}
5     disp('Method is linear')
6  case 'cubic'
7     disp('Method is cubic')
```

```
8  case 'nearest'
9     disp('Method is nearest')
10 otherwise
11    disp('Unknown method.')
12 end
13
14 % LOWER  Convert string to lowercase.
15 % B = LOWER(A) converts any uppercase characters in A
16 % to the corresponding lowercase character and
17 % leaves all other characters unchanged.
```

**Aufgabe 4**
*Schreiben Sie ein Programm das die Nummer eines Monats erfasst und die Anzahl seiner Tage ausgibt. Realisieren Sie dies zunächst durch die mehrseitig bedingte Anweisung und im Anschluss mit Hilfe des* `switch`*-Konstruktors.*

## 4.3 Schleifen

Wir wollen nun Kontrollstrukturen kennenlernen, die helfen, einen bestimmten Abschnitt des Codes eines Programms mehrfach zu wiederholen. Die sogenannten **Schleifen** sind beim imperativen Programmieren, wo die Befehle chronologisch abgearbeitet werden, von zentraler Bedeutung. Mit Hilfe von Schleifen kann ein Teil des Quellcodes solange wiederholt werden, bis eine Abbruchbedingung eintritt.

### 4.3.1 while-Schleife

Der allgemeine Aufbau einer `while`-Schleife hat folgende Gestalt

**while-Schleife**

```
while   ⟨Bedingung⟩
        ⟨Anweisungssequenz⟩
end
```

**Übersetzung:** Solange die *Bedingung* erfüllt ist, führe die *Anweisungssequenz* aus. Prüfe vor jedem Durchlauf der Sequenz die *Bedingung*.

Es wird vor jedem Durchlauf der Anweisungssequenz die Bedingung geprüft. Ist das Ergebnis `false`, so ist die Schleife beendet. Liefert die Prüfung dagegen ein `true` so wird die Anweisungssequenz ausgeführt und danach wieder die Bedingung geprüft usw. So ist prinzipiell eine unbegrenzte Anzahl an Iterationsschritten möglich. Zu beachten ist, dass der Schleifenrumpf gegebenen Falls kein einziges Mal ausgeführt wird.
Im nachfolgenden Beispiel dient eine `while` Schleife der iterativen Berechnung des Golden Schnittes. Man erhält den Goldenen Schnitt beispielsweise durch

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\ldots}}}}}$$

**Kodeschnipsel 7**

```
1  % der Goldene Schnitt
2  i=1;
3  n=1;
4  while i<10
5     gold=double(1+n);
6     n=1/(1+n);
7     fprintf('%f\n',gold);
8     i=i+1;
9  end
```

**Aufgabe 5**
*Das Wallissche Produkt dient zur Berechnung der Kreiszahl Pi. Schreiben sie ein Programm, dass Pi bis auf eine von Benutzer einzugebende Genauigkeit berechnet. Dabei sollten sie die Kontrollstrukturen* `If` *und* `while` *und das Produkt*

$$\pi = 2 \cdot \prod_{i=1}^{\infty} \frac{4\,i^2}{4\,i^2 - 1}$$

*implementieren.*

### 4.3.2 for-Schleife

#### Inkrementierung und Dekrementierung

Oft kommt es vor, dass bestimmte Routinen durch eine sogenannte **Laufvariable** gesteuert werden. So ist deren ganzzahlige Erhöung oder Reduzierung schnell realisiert durch die Schritte `i = i+1;` bzw `i = i-1;` wie im nächsten Beispiel zu sehen.

**Kodeschnipsel 8**

```
1 i=1;                          %<Initialisierung>
2 while i<11                    %<Bedingung>
3    fprintf('i ist: %i \n', i); %<Anweisung>
4    i = i+1;                   %<Schritt>
5 end
```

In vielen Programmiersprachen (in MATLAB nicht) existieren für **Inkrementierung** bzw. **Dekrementierung** spezielle Operatoren. `i = i+1;` wird dann durch `++i;` realisiert, `i = i-1;` entspricht dann `--i;`.
Nun wollen wir eine weiter Schleife kennen lernen die zu einer eleganteren Umsetzung des letzten Kodeschnipsels (bestehend aus Initialisierung, Bedingung, Anweisungssequenz und Schritt) führt.

#### Die for-Schleife

Eine `for`-Schleife hat in vielen Programmiersprachen die allgemeine Form

```
for  (⟨Initialisierung⟩⟨Bedingung⟩⟨Schritt⟩)
     {
     ⟨Anweisungssequenz⟩
     }
```

Die **Initialisierung** der Laufvariable wird dabei nur einmal zu Beginn der Schleife ausgeführt. Die **Bedingung** wird vor jedem Durchlauf geprüft, sodass die **Anweisungssequenz** nur ausgeführt wird, wenn die Prüfung `true` ergibt. Ergibt die Prüfung einen von `true` verschiedenen Wert, ist die Schleife beendet. Der **Schritt** am Ende jedes Schleifendurchlaufs wird oft durch die Inkrementierung `++i;` realisiert.
Die for-Schleife in MATLAB unterscheidet sich von der gängigen Form und hat die Gestalt

**for-Schleife**

`for` ⟨Initialisierung⟩:⟨Schritt (optional)⟩:⟨Bedingung⟩)
⟨Anweisungssequenz⟩
end

Der obige Kodeschnipsel kann in MATLAB also etwa geschrieben werden als

**Kodeschnipsel 9**

```
1 for i=1:1:10
2    fprintf('i ist: %i \n', i);
3 end
```

Wird kein **Schritt** angegeben ist die Schrittweite automatisch eins. Um die Wirkung der `for`-Schleife in MATLAB nochmals zu verdeutlichen übersetzen wir den letzten Kodeschnipsel. *Für die Variable i, die zu Beginn eins ist, drucke die Bildschirmausgabe* "i ist: ..." *und erhöhe danach i um eins. Mache dies solange, bis i der Wert 10 zugewiesen wird bzw. solange i kleiner elf ist.*

**Aufgabe 6**
*In MATLAB lässt sich eine* `n`×`n`*-Einheitsmatrix mit dem Befehl* `eye(n)` *erzeugen. Schreiben sie ein Programm, dass eine* `5`×`5`*-Einheitsmatrix erzeugt. Nutzen Sie dazu die* `for`*-Schleife. Lassen Sie von diesem Programm ausgeben, dass ihre Lösung richtig ist indem Sie die erzeugte Matrix mit der durch* `eye(5)` *erzeugten Einheitsmatrix vergleichen. Tipp: Im Abschnitt 2.2.1 haben wir gesehen, wie man die einzelnen Elemente einer Matrix anspricht.*

# Kapitel 5

# Funktionen

Funktionen sind in der imperativen Programmierung ein wichtiges Mittel zur Modularisierung eines Programms. Oft benötigte Algorithmen (Unterprogramme etc. ) könnne als Funktionen zusammengefasst und aus dem eigentlichen Programm ausgegliedert werden, was zu einer besseren "Lesbarkeit" eines Programms führt. Funktionen werden anhand ihres Namens aufgerufen und führen dann an der jeweiligen Stelle im Programm das entsprechende Unterprogramm aus, liefern gegebenenfalls einen Rückgabewert. Man unterscheidet zwischen **rekursiven** und **iterativen** Funktionen.

## 5.1 Deklaration einer Funktion

In MATLAB können Funktion wie normale Programme auch mit dem Editor geschrieben und als m-File gespeichert werden. Der Kopf einer Funktion hat dabei die Form

```
function [<Rueckgabeparameter>] = <Funktionsname>(<Eingabeparameter>)
```

- eine Funktion ist als **Funktionsname**.m abzuspeichern
- **Rückgabe-** und **Eingabeparameter** sind Variablen
- Rückgabe- und Eingabeparameter sind optional, dürfen also auch fehlen. Die Deklaration einer Funktion hat dann die Gestalt

```
function  <Funktionsname>(<Eingabeparameter>)
```

Dem Funktionskopf kann nun ganz normal der Quellcode der Funktion folgen. Dabei können Eingangsparameter verwand und sogar die Funktion selber wieder aufgerufen werden (Rekursion). Ein sehr leichtes Beispiel soll die Funktion zu Berechnungen an einer Kugel sein.

**Kodeschnipsel 10**

```
1  function [ofl,vol] = kugel(r)
2  % Funktion benoetigt Radius r und berechnet daraus Oberflaeche ofl und Volumen vol
3
4  ofl = 4.0 * pi * r * r;
5  vol = ofl / 3.0 * r;
```

Wie wir Funktionen aufrufen und verwenden sehen wir im nächsten Abschnitt.

## 5.2 Aufruf von Funktionen

Funktionen lassen sich anhand ihres Namens aufrufen, dies kann

- aus dem Command-Window heraus
- in einem Programm
- oder in einer Funktion geschehen.

In einem Beispiel nutzen wir die Funktion `kugel(r)` zur Berechnung des Kugelvolumens und verwenden des weitere eine Funktion `draw_sphere(r,ofl,vol)` zum Plot der Kugel. Die Funktion zur graphischen Darstellung ist

**Kodeschnipsel 11**

```
1  function draw_sphere(r,ofl,vol)
2  %Funktion benoetigt Radius, Oberflaeche, Volumen fuer die graphische Darstellung
3
4  %Diskretisierung
5  phi = linspace(0,2*pi,500);
6  theta = linspace(0,pi,500);
7
8  %Umrechnug Kugelkoordinaten -> kartesische Koordinaten
9  X = r*sin(THETA).*cos(PHI);
10 Y = r*sin(THETA).*sin(PHI);
11 Z = r*cos(THETA);
12
13 %graphische Ausgabe
14 figure;
15 [PHI,THETA]=meshgrid(phi,theta);
16 mesh(X,Y,Z);
17 xlabel('x');
18 ylabel('y');
19 zlabel('z');
20 title('3 Dimensionale Darstellung einer Kugel');
21 text(-r/3,r/3,r/3,['Radius: ' num2str(r)]);
22 text(-r/3,r/3,0,['Oberflaeche: ' num2str(ofl)]);
23 text(-r/3,r/3,-r/3,['Volumen: ' num2str(vol)]);
```

Wir sehen, dass die Funktion `draw_sphere(r,ofl,vol)` lediglich für die graphische Darstellung dient und keinen Rückgabewert hat. Die Wirkung der einzelnen verwendeten Befehle wie `meshgrid`, `linspace` usw. sind gut in der MATLAB Hilfe beschrieben. Durch die Eingabe von z.B. `help linspace` im Command-Window gelangt man zu einer Beschreibung.
Das Programm zur Berechnung und Darstellung der Kugel kann dann die folgende Form haben

**Kodeschnipsel 12**

```
1  % Programm: kugelberechnung.m
2  % Berechnet Volumen und Oberflaeche einer Kugel
3  %bei gegebenen Radius und stellt diese graphisch dar
4
5  fprintf('\nOberflaechenberechnung, Volumenberechnung und Darstellung einer Kugel\n')
6  fprintf('*****************************************************************\n\n')
7  fprintf('Bitte geben Sie den Radius der Kugel ein!\n')
8  radius = input('Radius = ');
```

```
9
10 %Berechnung
11 [ofl,vol] = kugel(radius);
12 %Darstellung
13 draw_sphere(radius);
```

Die Ausgliederung verschiedener Unterprogramme in Funktionen führt zu einem deutlich übersichtlicheren Hauptprogramm. Die Modularisierung eines Programms ist in der Praxis vor allem dann wichtig, wenn es sich um umfangreiche Projekte handelt, an denen mitunter mehrere Programmierer beteiligt sind.

### 5.2.1 Bemerkung zu Eingabe- und Rückgabeparametern

Das Arbeiten mit Funktionen bereitet dem Einsteiger hin und wieder Probleme. Wir wollen an dieser Stelle deshalb nochmals einige Beispiele betrachten.
Stellen wir uns vor, wir haben eine Funktion zur Berechnung des Mittelwertes dreier Zahlen geschrieben. Eine mögliche Funktionsdeklaration kann `function m = mittelwert(a,b,c)` sein. Die Philosophie von Funktionen ist nun wie folgt zu verstehen: Die Funktion erwartet drei Parameter und weist diese innerhalb der Funktion den Variablen `a, b, c` zu. Dies bedeutet nicht, dass beim Funktionsaufruf etwa die Variablen `a, b, c` genannte werden müssen die als Parameter übergeben werden!
Analog verhält es sich mit dem Rückgabeparameter. Dieser wird innerhalb der Funktion zwar der Variable `m` zugewiesen, im Programm (also beim Funktionsaufruf) kann man dem Rückgabeparameter jedoch mit einem anderen Variablennamen bezeichnen.
Die obige Funktion zur Mittelwertsberechnung sei also nun geschrieben. In einem Programm verwenden wir die drei Variablen `v1`, `v2`, `v3` für die Geschwindigkeit eines Autos auf drei Wegabschnitten. Mit Hilfe der Funktion `mittelwert` lässt sich nun die Durchschnittsgeschwindigkeit bestimmen. Dabei können wir uns vollkommen von den Parameternamen aus der Funktionsdeklaration lösen. Wir weisen den Mittelwert der Variable `v_mittel` zu. In unserem Programm steht also nun `v_mittel = mittelwert(v1,v2,v3)`.

## 5.3 rekursive Funktionen

Ruft eine Funktion sich wärend sie ausgeführt wird selber wieder auf, so spricht man von einer Rekursion. Rekursionen ermöglichen oft sehr elegante Programme. Einige Probleme lassen sich praktisch nur rekursiv lösen. Das rekursive Arbeiten mit Funktionen ist dennoch schwer vorstellbar. In einem ersten Beispiel sollen deshalb iterative und rekursive Berechnungen gegenübergestellt werden.

#### 1. Beispiel: Euklidischer Algorithmus

Der **Euklidischen Algorithmus** ist eine Vorschrift zum finden des größten gemeinsamen Teilers zweier Zahlen $a$ und $b$.
"*Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.*"
(Aus Euklid, Die Elemente, Herausgegeben von Clemens Thaer)

Euklid berechnete den größten gemeinsamen Teiler, indem er nach einem gemeinsamen „Maß" für die Längen zweier Linien suchte. Dazu zog er wiederholt die kleinere der beiden Längen von der größeren ab.
Die iterative Formulierung kann in MATLAB wie folgt implementiert werden.

**Kodeschnipsel 13**

```
1  function [ggt] = euklid_iter(a,b)
2  if a==0
3      ggt=b;
4  else
5      while b~=0
6          if a>b
7              a=a-b;
8          else
9              b=b-a;
10         end
11     end
12 end
13 ggt=a;
```

Eine Rekursion (durch eine sich selbst wieder aufrufende Funktion) könnte so aufgebaut sein

**Kodeschnipsel 14**

```
1  function [ggt] = euklid_rek(a,b)
2
3  if b==0
4      ggt = a;
5  else
6      if a>b
7          ggt = euklid_rek(a-b,b);
8      else
9          ggt = euklid_rek(a,b-a);
10     end
11 end
```

**Aufgabe 7**

*Schreiben Sie eine Funktion* `fakultaet_iter(n)` *und eine Funktion* `fakultaet_rek(n)`*, welche die Fakultät einer Zahl n iterativ bzw. rekursiv berechnet.*

# Kapitel 6

# Abschließendes Programmierbeispiel: Das Buffonsche Nadelproblem

In diesem Kapitel wollen wir ein abschließendes Programm schreiben. Dabei wollen wir den kompletten Lösungsweg vom Aufstellen des Problems über dessen Analyse bis hin zum fertig implementierten Programm in MATLAB nachvollziehen. Dies soll das Vorgehen in der späteren Praxis als Programmierer noch einmal verdeutlichen.

## 6.1 Das Problem

Das Problem wurde erstmals im 18. Jahrhundert von Georges-Louis Leclerc, Comte de Buffon formuliert: "*Man werfe zufällig eine Nadel der Länge l auf eine Ebene, die mit äquidistanten Linien deren Abstand d beträgt versehen ist. Dabei sei* $l \leq d$. *Wie gross ist die Wahrscheinlichkeit* $P(A)$, *dass die Nadel eine der Linien schneidet?*" Wir werden versuchen, dieses Problem analytisch

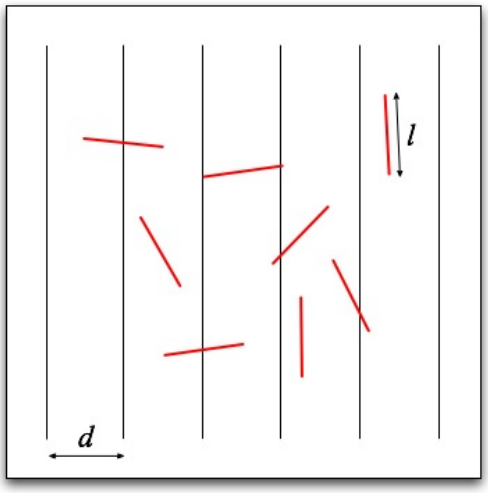
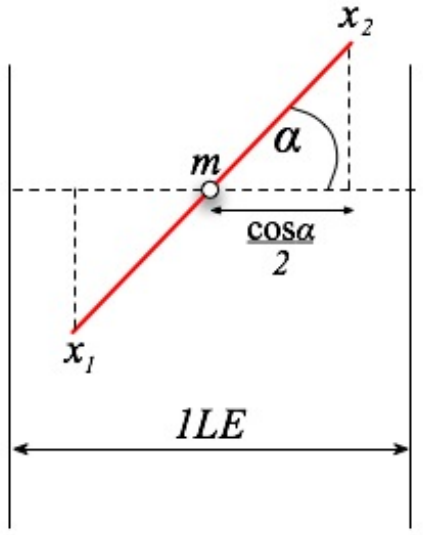



Abbildung 6.1: Buffons Nadelproblem

zu durchdenken und in MATLAB zu implementieren. Dabei werden wir auf eine erstaunliche Approximation der Kreiszahl Pi stoßen die durch unser Programm berechnet werden soll.
Der Einfachheit halber wollen wir dabei einen Abstand und eine Nadel der Länge 1 betrachten ($l = d = 1LE$).

## 6.2 Analyse des Problems

Die folgende Abbildung soll die Problemstellung systematisieren.




Wir wollen mit $x_1$ immer den linken Rand, mit $x_2$ den rechten Rand und mit $m$ den Mittelpunkt der Nadel verstehen. Dabei soll weiterhin gelten, dass $x_1 \leq m \leq x_2$. Es ist der Skizze demnach zu entnehmen, dass

- $\underline{0 \leq m \leq 1}$
- $\underline{0 \leq \alpha \leq \frac{\pi}{2}}$.

Der Zusammenhang der zufälligen Variablen $x$ (der Koordinate auf der $m$ liegt) und $\alpha$ ist gegeben durch

$$x = \cos\alpha.$$

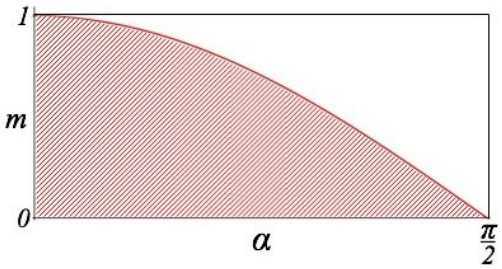
Dabei ist

$$x_1 = m - \frac{\cos\alpha}{2} \quad \text{und} \quad x_2 = m + \frac{\cos\alpha}{2}.$$

Es ist also offensichtlich, dass die Nadel genau dann die Linie schneidet, wenn

$$x_1 \leq 0 \quad \text{oder} \quad x_2 \geq 1.$$

Nun haben wir das Problem soweit systematisiert, dass wir die Frage, mit welcher Wahrscheinlichkeit die Nadel eine Linie schneidet, anschaulich beantworten können. Dazu dient folgende Skizze.




Die Nadel landet beim Werfen **immer** so, dass dem Wertepaar ($\alpha$,$m$) ein Punkt im obigen Koordinatensystem entspricht. Wir machen uns nun folgendes anschaulich klar: Die Fäche unter dem Graphen (der durch $\cos\alpha$ gegeben ist) entspricht genau dem Anteil aller Nadelwürfe, bei dem die Parameter $\alpha$ und $m$ genau so sind, dass die Nadel eine Linie schneidet. So haben wir

$$P(\text{Linie wird geschnitten}) = \frac{\textit{Fläche unter der Kurve}}{\textit{Gesamtfläche}} = \frac{\int_0^{\frac{\pi}{2}} \cos\alpha \, \mathrm{d}\alpha}{1 \cdot \frac{\pi}{2}} = \frac{2}{\pi}.$$

Der Umkehrschluss erlaubt es uns die Kreiszahl $\pi$ zu approximieren. Dazu ermitteln wir die Wahrscheinlichkeit durch die relative Häufigkeit eines Treffers, also dem Quotienten aus der Anzahl der Treffer ($hits$) durch die Anzahl der gemachten Würfe ($n$) und haben schließlich

$$\underline{\underline{\pi = \frac{n}{hits} \cdot 2}}$$

## 6.3 Das Programm zur Berechnung von Pi

Die gründliche und systematische Betrachtung des Problems erlaubt es uns nun eine Funktion zur Berechnung von $\pi$ sehr schnell zu implementieren. Diese erwartet als Parameter lediglich die Anzahl der Würfe und gibt daraufhin eine Nährung von $\pi$ zurück. Dabei wird die MATLAB-Funktion `rand` verwendet um eine Zufallszahl im Intervall $[0, 1]$ zu simulieren.

**Kodeschnipsel 15**

```
1  function piapprox = buffon_pi(wuerfe)
2     hit = 0;
3     for n = 1 : (wuerfe)
4        %Simmuliere Nadelwurf
5         mp = rand(1);                    % Mittelpunkt der Nadel
6         phi = (rand(1) * pi) - pi / 2;   % Winkel der Nadel
7         x2 = mp + cos(phi)/2;            % rechtes
8         x1 = mp - cos(phi)/2;            % linkes
9         if x2 >= 1 | x1 <= 0
10            hit = hit + 1;               % Trefferzaehler
11        end
12    end
13    piapprox = wuerfe / hit * 2;         % Approximation fuer Pi
```

## 6.4 Ein Simulation

Abschließend wollen wir eine mögliche Simulation des Problems lediglich angeben. Die Möglichkeit von MATLAB in der Anwendung bei Simulationen soll dabei demonstriert werden. Wir werden einige MATLAB-Funktionen wie `ceil`, `floor`, `real`, `imag` usw. sowie `plot` zur grafischen Ausgabe. Diese sind alle schnell über die MATLAB-Hilfe zu verstehen.

**Kodeschnipsel 16**

```
function [piapprox] = buffon_plot(wuerfe)
% Buffons Nadelproblem, erwartet eine Anzahl der Wuerfe und
% simuliert dann den Versuch um PI zu approximieren
% Aufruf: z.B: buffon_plot(30);
    close all
    hold on
    axis equal
    % Grafische Darstellung der parallelen Geraden
    for h = 0 : 15
        P1 = [h -0.5];
        P2 = [h 15.5];
        L = [0 1 0.1];
        hold on
        a = (L(1):L(3):L(2));
        x = P1(1) + a*(P2(1) - P1(1));
        y = P1(2) + a*(P2(2) - P1(2));
        plot(x,y,'b');
    end
   % Figure maximieren auf ganzen Bildschirm
   figure(1)
   set(figure(1),'units','normalized','outerposition',[0 0 1 1])
   title('Buffons Nadelproblem','FontSize',20)
```

```
   axis([-1,16,-1,16]);
   hit = 0;
   ps = 0.1;
   % Wuerfe simulieren
   for l = 1 : (wuerfe)
       % Position des Mittelpunkts und Winkel der Nadel in komplexer Ebene
       mp = (rand * 15) + i * (rand * 15);
       phi = (rand * pi) - pi / 2;
       % linker Endpunkt der Nadel und dessen kartesische Koordinaten
       xi1 = 1/2 * exp(i * (phi + pi)) + mp;
       x1 = real(xi1);
       y1 = imag(xi1);
       % rechter Endpunkt der Nadel und dessen kartesische Koordinaten
       xi2 = 1/2 * exp(i * phi) + mp;
       x2 = real(xi2);
       y2 = imag(xi2);
       % zwischen welchen Linien ist die Nadel gelandet?
       l2 = ceil(real(mp));
       l1 = floor(real(mp));
       %Nadeln Plotten
       if x2 >= l2 % Treffer rechte Linie
           hit = hit + 1;
           plot([x2 x1], [y2 y1], ['-','g']);       % gerade geworfen (gr§n)
           plot([l2 l2], [-0.5 15.5], ['-','g']);   % getroffene Linie (gr§n)
           pause(ps)
           plot([x2 x1], [y2 y1], ['-','r']);       % Nadeln rot dargestellt
       elseif x1 <= l1 % Treffer linke Linie
           hit = hit + 1;
           plot([x2 x1], [y2 y1], ['-','g']);       % gerade geworfen (gr§n)
           plot([l1 l1], [-0.5 15.5], ['-','g']);   % getroffene Linie (gr§n)
           pause(ps)
           plot([x2 x1], [y2 y1], ['-','r']);       % Nadeln rot dargestellt
       else %kein Treffer
           plot([x2 x1], [y2 y1], ['-','g']);       % gerade geworfen (gr§n)
           pause(ps)
           plot([x2 x1], [y2 y1], ['-','k']);       % Nadel schwarz darstellen
       end
       % Zur§cksetzen der getroffenen Geraden in blaue Farbe
       plot([l2 l2], [-0.5 15.5], ['-','b']);
       plot([l1 l1], [-0.5 15.5], ['-','b']);
       piapprox = wuerfe / hit * 2;
       %Textfeld neben dem Plot
       text(17,15.1,sprintf('geworfene Nadeln: %i\nTreffer: %i', [l hit]),...
            'HorizontalAlignment','left',...
            'BackgroundColor',[1 1 1],'FontSize',15)
   end
   %Textfeld neben dem Plot mit Pi
   text(17,15.1,sprintf('geworfene Nadeln: %i\nTreffer: %i\nPI=%f', [l hit piapprox]),...
            'HorizontalAlignment','left',...
            'BackgroundColor',[1 1 1],'FontSize',15)
end
```

**Aufgabe 8**

*Das "Ziegenproblem" (Monty-Hall Dilemma) ist wie folgt beschrieben: Hinter drei Türen sind 2 Nieten und ein Hauptgewinn. Auf der Suche nach dem Gewinn wählt der Kandidat ein Tor (öffnet es aber noch nicht). Der Moderator wählt daraufhin immer ein Tor, hinter dem sich eine Niete verbirgt, öffnet es und schlägt*

*dem Kandidaten vor, das Tor zu wechseln. Ist es vorteilhaft für den Kandidaten, das Tor zu wechseln? Simulieren Sie das "Ziegenproblem" in MATLAB und treffen Sie eine Entscheidung auf die Frage.*

## 6.5 Wie geht's weiter?

In der Vorlesung Computational Physics I werden wir numerische Verfahren kennen lernen um mathematische und schließlich physikalische Probleme mit Hilfe des Computers zu bearbeiten. Nachdem zunächst einige Grundlagen der Numerik gelegt werden, gilt es im Anschluss spezielle Probleme in MATLAB zu implementieren. Dabei sind die in diesem Skript erworbenen Programmierkenntnisse minimale Voraussetzung für ein rasche Umsetzung.

# Anhang A

# Lösungsvorschläge

**zu Aufgabe 1**

```
1  disp('Bitte geben Sie eine Zahl ein:');
2  a = input('a = ');
3
4  if a<0 && mod(a,2)==0
5      fprintf('Dies ist eine negative, gerade Zahl.');
6  elseif a<0 && mod(a,2)~=0
7      fprintf('Dies ist eine negative, ungerade Zahl.');
8  elseif a>0 && mod(a,2)==0
9      fprintf('Dies ist eine positive, gerade Zahl.');
10 elseif
11     fprintf('Dies ist eine positive, ungerade Zahl.');
12 end
```

**zu Aufgabe 2**

```
fprintf('\nProgramm zur Loesung der quadratischen Gleichung x^2+a*x+b = 0\n');
fprintf('************************************************************\n\n');

a = input('Koeffizient a = ');
b = input('Koeffizient b = ');
c = -a / 2.0;
d = c * c - b; %Diskriminante
w = sqrt(abs(d));

if d < 0
    re = c;
    im = w;
    fprintf('\nLoesungen ist komplex!\n');
    fprintf('Zur Ausgabe des Real- und Imaginaerteil der Loesung (1)\n');
    fprintf('Zur Ausgabe des Betrags der komplexen Zahl (2)\n');
    wahl = input('bitte waehlen:  ');
    if wahl == 1
        fprintf('Der Realteil ist %f \n',re);
        fprintf('Der Imaginaerteil ist %f \n',im);
    elseif wahl ==2
        betrag = sqrt(re^2+im^2);
        fprintf('Der Betrag der komplexen Zahl ist=%f \n',betrag);
```

```
    else
        fprintf('Keine gueltige Wahl.\n');
    end
else
x1 = c + w;
x2 = c - w;
fprintf('\nLoesungen sind reell!\n');
fprintf('x1=%f \n',x1);
fprintf('x2=%f \n',x2);
end
```

**zu Aufgabe 3**

*Das Programm verursacht einen Fehler in Zeile 2. Die* `if`*-Abfrage erwartet eine Bedingung (logischen Operator). Dies wäre beispielsweise* `a+b == 6`*. Es folgt jedoch der Ausdruck* `a+b=6`*. MATLAB gibt somit einen Fehlermeldung aus. Andere Programmiersprachen z.B.* `C++` *geben bei einem Ausdruck Anstelle der erwarteten Bedingung dennoch* `true` *zurück. Dies führt dazu, dass uns das Programm in diesem Fall mitteilt* `Die Summe ist 6.`*.*
*Das beschriebene Problem ist ein beliebter Anfängerfehler (vgl. Abschnit 3.1.1 Seite 7).*

**zu Aufgabe 4**

```
1  fprintf('\nProgramm zum Ermitteln der Anzahl der Tag eines Monats\n');
2  fprintf('*******************************************************\n\n');
3  fprintf('Nr. des Monats: 1 (Januar) bis 12 (Dezember).\n');
4  monat = input('Monat: ');
5
6  if monat==1 | monat==3 | monat==5 | monat==7 | monat==8 | monat==10 | monat==12
7      tage = 31;
8  elseif monat==2
9      fprintf('Ist es ein Schaltjahr? 1 - Ja / 2 - Nein\n');
10     sj = input('Bitte waehlen: ');
11     if sj==1
12         tage = 29;
13     elseif sj==2
14         tage = 28;
15     else
16         tage = 0;
17         fprintf('Falsche Angaben!!!\n');
18     end
19 elseif monat==4 || monat==6 || monat==9 || monat==11
20     tage = 30;
21 else
22     tage = 0;
23     fprintf('Falsche Angaben!!!\n');
24 end
25
26 if tage~=0
27     fprintf('Der Monat hat %i Tage.\n',tage );
28 end
```

*Alternativ kann die* `If`*-Abfrage auch mit dem* `switch`*-Konstrukt wie folgt realisiert werden*

```
1  nr = input('Bitte geben Sie die Nummer des Monats an: ');
2
```

```
3  switch m
4     case {1,3,5,7,8,10,12}
5         fprintf('\n 31 Tage \n');
6     case 2
7         fprintf('\ n28 Tage \n'); % to DO if-construct f§r Schaltjahr!!!
8     case {4,6,9,11}
9         fprintf('\ n30 Tage \n');
10 end
```

**zu Aufgabe 5**

```
1  n=1;
2  var_pi=1;
3
4  while n<200
5      var_pi=var_pi*4*n^2/(4*n^2-1);
6      n=n+1;
7  end
8
9  var_pi=var_pi*2;
10 fprintf('%f\n',var_pi);
```

**zu Aufgabe 6**

```
1  for i=1:5
2      for j=1:5
3          if i==j
4              m(i,j)=1;
5          else
6              m(i,j)=0;
7         end
8      end
9  end
10
11 if m==eye(5)
12     fprintf('Die Matrix ist richtig erstellt \n');
13    m
14 else
15    fprintf('Die Matrix ist falsch erstellt.');
16 end
```

**zu Aufgabe 7**

*Rekursive Fakultätsberechnung:*

```
1  function fak = fakultaet_rek(n)
2
3  if n <= 1
4   fak = 1;
5  else
6      fak = n * fakultaet_rek(n-1);
7  end
```

*Irerative Fakultätsberechnung:*

```
1  function [fak] = fakultaet_iter(n)
2
3  fak = 1;
4  faktor = 2;
5  while faktor <= n
6    fak = fak * faktor;
7    faktor = faktor + 1;
8  end
```

# Literaturverzeichnis


[1] Ottmar Beucher: *MATLAB und Simulink: Grundlegende Einführung für Studenten und Ingenieure in der Praxis*, Pearson-Studium, 4. Auflage 2008.

[2] Klaus Simon: *Programmieren für Mathematiker und Naturwissenschaftler: Eine pragmatische Einführung mit C++*, vdf Hochschulverlag AG an der ETH Zürich, 2. Auflage 2000.

[3] Werner Vogt: *Algorithmen und Programmierung*, Technische Universität Ilmenau FG Numerische Mathematik und Informationsverarbeitung, Skript im Studienjahr 2007/2008.