Physik mit dem Computer

Andreas Honecker

Institut für Theoretische Physik

Technische Universität Braunschweig

Sommersemester 2002

http://www.tu-bs.de/~honecker/comp

1 Allgemeine Bemerkungen

☞ Die Übungen finden

Freitags 14:00-15:30
in HS 65.2 des Rechenzentrums

statt.

☞ Bitte besorgen Sie sich hierzu einen Account am Rechenzentrum (Y-Nummer), falls noch nicht vorhanden !

☞ Sie lernen nur durch Mitmachen – scheuen Sie sich also nicht vor Fragen !

Skript:

☞ Die neueste Version dieses Skripts finden Sie immer auf der Web-Seite

`http://www.tu-bs.de/~honecker/comp`

☞ Kommentare zum Programmieren werden mit Serifen (Times), zur Physik und Algorithmen ohne Serifen (Helvetica) geschrieben.

A1.1 *Übungsaufgaben werden wie hier durchnummeriert und kursiv geschrieben.*

A1.2* *Aufgaben, die nicht wesentlich sind, werden mit Sternchen* * *markiert (je mehr Sternchen, desto anspruchsvoller bzw. unwichtiger). Alle anderen Aufgaben sollte jeder bearbeiten und (mit Hilfe) lösen.*

2 Freier Fall

A2.1 *Ein Teilchen liege* ($\vec{v}_0 = 0$) *bei* $t = 0$ *in einer Höhe* $z = 20m$. *Geben Sie die Höhe z des frei fallenden Teilchens unter Verwendung der expliziten Lösung (2.3) in den ersten zwei Sekunden* ($0 \leq t \leq 2s$) *im Abstand von* $0,02$ *Sekunden aus !*

A2.2 *Behandeln Sie das Problem aus A2.1, indem Sie die Newton'sche Bewegungsgleichung mit dem Euler-Verfahren lösen ! Vergleichen Sie die numerische Lösung mit der exakten ! Wie klein muß man* Δt *wählen, damit der numerische Wert für* z *bei* $t = 2s$ *um weniger als* $1mm$ *von der exakten Lösung abweicht ?*

A2.3 *Lösen Sie das Problem aus A2.1 noch einmal, aber jetzt mit dem Euler-Richardson-Algorithmus (ansonsten wie in A2.2) ! Wie klein müssen Sie jetzt* Δt *wählen, um bei* $t = 2s$ *eine Abweichung besser als* $1mm$ *zu erhalten ?*
Fällt Ihnen etwas auf ?

A2.4 *Nun betrachten wir den Fall eines Teilchens aus* $z = 1km$ *Höhe unter Einfluß der Reibungskraft (2.10) mit* $\lambda = 0,7s^{-1}$, *d.h. bei* $t = 0$ *gilt* $\vec{v}_0 = 0$. *Lösen Sie die Bewegungsgleichung mit dem Euler-Richardson-Algorithmus für* $\Delta t = 0,1s$ *!*
Bestimmen Sie den Aufschlagzeitpunkt und vergleichen Sie die Falldauer mit dem Ergebnis ohne Reibung !
Wie hoch ist die Geschwindigkeit zum Zeitpunkt des Aufschlags ? Vergleichen Sie dies mit dem nach (2.11) bestimmten Wert von $\vec{v}_{\max}$ *!*

A2.5* *Bestimmen Sie im Fall von A2.4, nach welcher Fallstrecke und -zeit die Geschwindigkeit* $0,98\,\vec{v}_{\max}$ *erreicht, mit einer Genauigkeit von mindestens* $1cm$ *bzw.* $0,05s$ *!*
Können Sie das hierzu erforderliche Δt *auch abschätzen, ohne zu programmieren ?*

2.1 Elementare Mechanik und Euler-Verfahren

Die Newton'sche Bewegungsgleichung lautet bekanntlich

$$m \frac{\mathrm{d}^2}{\mathrm{d}t^2} \vec{r} = \vec{F} \,, \tag{2.1}$$

wobei $\vec{F}$ die Summe aller auf das Teilchen wirkenden Kräfte ist. Die Gravitationskraft kann in der Nähe der Erdoberfläche approximiert werden durch

$$\vec{F} = -m \, g \, \vec{e}_z \,. \tag{2.2}$$

Für diesen Fall ist (2.1) leicht lösbar. Man findet, daß die Bahnkurve im Gravitationsfeld der Erde lautet:

$$\vec{r}(t) = \vec{r}_0 + \vec{v}_0 \, t - \frac{1}{2} \, g \, t^2 \vec{e}_z \,. \tag{2.3}$$

Zwecks Vergleichbarkeit der Ergebnisse verwenden wir bitte alle

$$g = 9{,}80665 \, \frac{\mathsf{m}}{\mathsf{s}^2} \quad ! \tag{2.4}$$

Die Masse m kürzt sich übrigens beim Einsetzen von (2.2) in (2.1) heraus und man erhält die Bewegungsgleichung

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2} \vec{r}(t) = \vec{a} \left(\vec{r}(t), \vec{v}(t), t \right) \tag{2.5}$$

(mit $\vec{a} \left(\vec{r}(t), \vec{v}(t), t \right) = -g \, \vec{e}_z$).

Das Euler-Verfahren ist eine (die wohl einfachste) Methode, Differentialgleichungen vom Typ (2.5) numerisch zu lösen. Dazu betrachtet man zuerst diskrete Zeiten mit Abständen Δt.

$$t_n = t_0 + n \, \Delta t \,. \tag{2.6}$$

Ist $\vec{a}_n = \vec{a}(t_n)$ bekannt, kann man für die Werte $\vec{v}_n = \vec{v}(t_n)$ und $\vec{r}_n = \vec{r}(t_n)$ mit $n > 0$ und bekannten $\vec{v}_0$ und $\vec{r}_0$ die Approximation des Euler-Verfahrens verwenden:

$$\begin{aligned} \vec{v}_{n+1} &= \vec{v}_n + \vec{a}_n \, \Delta t \,, \\ \vec{r}_{n+1} &= \vec{r}_n + \vec{v}_n \, \Delta t \,. \end{aligned} \tag{2.7}$$

Dies ergibt sich einfach, indem man die Ableitung bei t_n durch einen Differenzenquotienten von den Zeiten t_{n+1} und t_n ersetzt und umformt.

Beachte: Die Werte am Ende des Zeitintervals $\vec{v}_{n+1}$, $\vec{r}_{n+1}$ werden ausschließlich durch die Werte an dessen Anfang $\vec{a}_n$, $\vec{v}_n$ und $\vec{r}_n$ bestimmt.

Genau an dieser Stelle kann man ansetzen, um das Verfahren zu verbessern, d.h. bei gleichem Rechenaufwand genauere Ergebnisse zu erhalten. Die Euler-Richardson-Methode (eng verwandt mit dem Runge-Kutta-Verfahren) ist eine solche Vebesserung: Man bestimmt zuerst mit dem Euler-Verfahren Werte für eine mittlere Geschwindigkeit $\vec{v}_{\rm mid}$ und Position $\vec{r}_{\rm mid}$ zu einem mittleren Zeitpunkt $t_{\rm mid} = t + \Delta t/2$. Die neuen Werte $\vec{v}_{n+1}$ und $\vec{r}_{n+1}$ werden dann unter Verwendung von $\vec{a}_{\rm mid} = \vec{F}\left(\vec{r}_{\rm mid}, \vec{v}_{\rm mid}, t_n + \Delta t/2\right)/m$, $\vec{v}_{\rm mid}$ und $\vec{r}_{\rm mid}$ bestimmt. Wir fassen das Euler-Richardson-Verfahren zusammen:

$$
\begin{aligned}
\vec{a}_n &= \vec{F}\left(\vec{r}_n, \vec{v}_n, t_n\right)/m\,, \\
\vec{v}_{\rm mid} &= \vec{v}_n + \vec{a}_n \frac{\Delta t}{2}\,, \\
\vec{r}_{\rm mid} &= \vec{r}_n + \vec{v}_n \frac{\Delta t}{2}\,, \\
\vec{a}_{\rm mid} &= \vec{F}\left(\vec{r}_{\rm mid}, \vec{v}_{\rm mid}, t_n + \frac{\Delta t}{2}\right)/m
\end{aligned} \tag{2.8}
$$

und

$$
\begin{aligned}
\vec{v}_{n+1} &= \vec{v}_n + \vec{a}_{\rm mid}\,\Delta t\,, \\
\vec{r}_{n+1} &= \vec{r}_n + \vec{v}_{\rm mid}\,\Delta t\,.
\end{aligned} \tag{2.9}
$$

Zwar müssen wir jetzt doppelt so viele Operationen pro Zeitschritt ausführen, meistens ist der Euler-Richardson-Algorithmus aber trotzdem deutlich schneller als der einfache Euler-Algorithmus, da wir größere Zeitschritte Δt verwenden können.

Bemerkung: Im Euler-Algorithmus erhält man Abweichungen der Ordnung Δt^2, beim Euler-Richardson-Algorithmus sind die Abweichungen von der Ordnung Δt^3.

Zurück zur Physik. Reibungskräfte sind geschwindigkeitsabhängig. Für Flüssigkeiten und Gase gilt für nicht allzu große Geschwindigkeiten in guter Näherung

$$
\vec{F}_R = -\lambda\, m\, \vec{v}\,, \tag{2.10}
$$

wobei die Konstante λ von der Geometrie und Masse des Körpers abhängt und die Einheit $1/\mathrm{s}$ besitzt.

Die Reibung führt zu einer wichtigen Änderung im Vergleich zum idealen freien Fall. Nun können sich die Erdanziehung und Reibung aufheben ! Dies führt zu einer maximalen Geschwindigkeit $\vec{v}_{\max}$, die sich leicht aus (2.2) und (2.10) herleiten läßt:

$$\vec{v}_{\max} = -\frac{g}{\lambda}\,\vec{e}_z\,. \tag{2.11}$$

2.2 Erste Schritte im Programmieren unter C++

Unser erstes Programm soll `first.cc` heißen:

```
#include <iostream>

int main()
 {
  cout << "Ein allerserstes Programm in C++\n";
 }
```

Bemerkungen:

1. Jedes C/C++-Programm muß eine Funktion `main` enthalten (erkennbar an den Argument-Klammern '`()`'), bei der das Programm startet.
2. In C/C++ schließt jede Anweisung mit einem '`;`' ab.
3. Blöcke (d.h. auch jede Funktion) werden mit geschweiften Klammern '`{}`' zusammengefaßt.
4. Wir wollen hier 'nur' etwas anzeigen (eigentlich ist das bereits ziemlich anspruchsvoll). Dazu verwenden wir die C++-Bibliothek `iostream`:
   (a) `#include <iostream>`
   liest die Definitionen für diese Bibliothek ein.
   (b) '`<<`' leitet alles dahinter auf die Standard-Ausgabe '`cout`'.
   (c) Die Anführungszeichen '`""`' umschließen eine Zeichenkette (String).
   (d) Das '`\n`' sorgt für einen Zeilenvorschub. Eine andere häufig verwendete Kontrollsequenz ist '`\t`', sie erzeugt einen Tabulator.

Dieser Programm-Quelltext muß nun in eine ausführbare Datei übersetzt werden. Dies wird 'Compilieren' genannt und von einem 'Compiler' ausgeführt. Es gibt verschiedene C++-Compiler, die sich in Details (u.a. der Programm-Optimierung) unterscheiden, aber Standardprogramme alle anstandslos verarbeiten sollten. Wir verwenden den freien Gnu-C++-Compiler. Unser Programm-Quelltext `first.cc` wird durch folgenden Compiler-Aufruf übersetzt:

```
g++ -o first first.cc
```

Wir haben nun ein ausführbares Programm `first`. Dies kann man wie folgt ausführen:

```
./first
```

Je nach Definition des Pfades kann man das '`./`' auch weglassen und einfach nur '`first`' eingeben. Hat man alles richtig gemacht, erscheint die folgende Ausgabe:

```
Ein allererstes Programm in C++
```

Weitere Sprachbestandteile von C++ und insbesondere die elementarsten Rechenoperationen illustriert das folgende Programm:

```
// Alles, was in der Zeile nach '//' steht, ist ein Kommentar
// und wird von dem Compiler ignoriert.
#include <iostream>

int main()
 {                              // Bis hierher kennen wir das schon

  cout << 2*3+4 << "\n";        // Eine einfache Berechnung
                                // Das nachgestellte "\n" sorgt
                                // fuer einen Zeilenvorschub

  cout << (2*3+4)/7 << "\n";    // Hier sieht man, dass die
                                // Berechnung mit ganzen Zahlen
                                // (Integers) ausgefuehrt wird

  cout << (2*3+4)/7.0 << "\n";  // Wollen wir hingegen ein Fliess-
                                // komma-Ergebnis, muessen wir das
                                // irgendwie signalisieren - z.B.,
                                // indem wir eine Zahl in Fliess-
                                // kommadarstellung angeben
 }
```

Die Ausgabe dieses Programms sieht wie folgt aus:

```
10
1
1.42857
```

Damit haben nun auch gesehen:

1. Wie man Kommentare mit '//' einfügt.
2. Das Aneinanderfügen mehrerer Ausgabe-Elemente mit '<<'.
3. Die Grundrechenarten '+', '-', '*' und '/' kenngelernt.
4. Den Unterschied beim Rechnen mit ganzen Zahlen (Integers) und Fließkommazahlen.

Nun werden wir noch etwas anspruchsvoller: Wir wollen die Summe

$$S_N = \sum_{k=1}^{N} \frac{1}{k^4} \tag{2.12}$$

für $N = 100$ auswerten. Eine Lösung ist das folgende Programm:

```
#include <iostream>

int main()
 {
  int N=100;                     // Wir definieren eine Integer-
                                 // Variable "N", die den Wert 100
                                 // erhaelt

  double summe=0;                // und eine Fliesskomma-Variable
                                 // "summe", die den Wert 0 erhaelt

  for(int k=1; k<=N; k+=1)       // Nun eine Schleife ueber k von 0
                                 // bis N in Schritten von 1
    {
     double ksq = k*k;           // k^2 berechnen
     summe += 1/(ksq*ksq);       // 1/k^4 zu "summe" addieren
    }

   cout.precision(11);           // Ausgabe auf 11 Stellen
   cout << "Summe = " << summe << "\n";  // das Ergebnis anzeigen
 }
```

Das Ergebnis lautet dann[1]

[1] $N = 100$ ist schon eine ziemlich gute Approximation an $N = \infty$, denn $S_\infty = \frac{\pi^4}{90} \approx 1,082323234$.

```
Summe = 1.0823229053
```

Und wieder haben wir einiges gelernt:

1. Die Definition von Integer-Variablen mit '`int`' sowie die von Fließkomma-Variablen mit '`double`' (aufgrund der numerischen Stabilität verwenden wir grundsätzlich doppelte Genauigkeit).

2. Die Zuweisung von Variablen-Inhalten mit '`=`'.

3. Die Addition eines vorgegebenen Wertes zum Variablen-Inhalt mit Hilfe von '`+=`'.

4. Den Vergleich von zwei Zahlen mit '`<=`' (es gibt auch '`<`', '`>`', '`>=`', gleich '`==`' und ungleich '`!=`').
   Achtung: Der Test auf Gleichheit erfolgt mit '`==`' – '`=`' bewirkt eine Zuweisung !

5. Wie man eine Schleife mittels

   `for(`Initialisierung`;` Abbruchbedingung`;` Inkrementierung`)`

   konstruiert. Die Schleife wird dabei so lange durchlaufen, bis die 'Abbruchbedingung' falsch wird.
   Achtung: In der `for`-Anweisung kommt nach dem '`)`' kein Semikolon '`;`' ! Das würde dann als Schleife ohne Kommando interpretiert (ist auch manchmal sinnvoll).

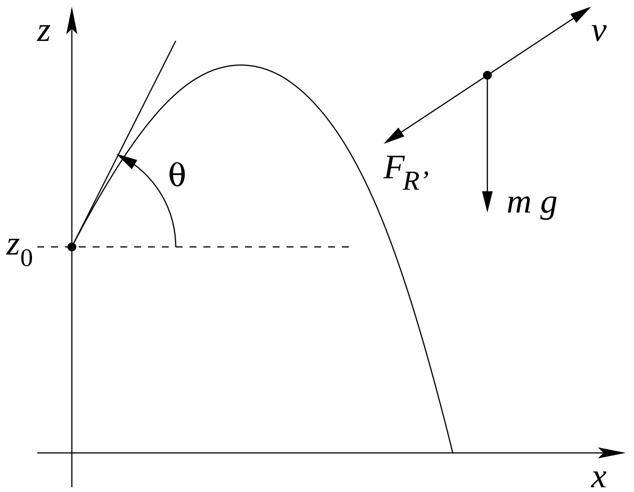
6. Daß man bei der Berechnung von $1/k^4$ aufpassen muß, damit für $k > 1$ nicht 0 herauskommt, d.h. rechtzeitig vom Integer- in den Fließkomma-Datentyp konvertieren muß.
   Eigentlich würde k^2 noch richtig berechnet, wenn wir bis dahin noch Integer-Arithmetik verwenden, aber probieren Sie mal aus, was passiert, wenn Sie statt '`double ksq = k*k;`' schreiben '`int ksq = k*k;`'!).

7. Daß man mit

   `cout.precision(`n`);`

   die Genauigkeit der Ausgabe verändern –und damit auch erhöhen– kann.

3 Schiefer Wurf



A3.1 *Ein Ball werde in einer Höhe von* $z = 1,6m$ *mit einer Anfangsgeschwindigkeit* $|\vec{v}_0| = 24\frac{m}{s}$ *unter einem Winkel* $\theta = 45°$ *abgeworfen. Lösen Sie die Bewegungsgleichung mit dem Euler-Richardson-Verfahren mit* $\Delta t = 0,002s$ *unter Einbeziehung der Erdanziehung (2.2) und phänomenologischen Reibungskraft (3.4) mit* $C = 10^{-2}m^{-1}$ *!*

a. *Der Aufschlagszeitpunkt sei gegeben durch* $z(t) = 0$. *Bestimmen Sie Flugstrecke und -zeit und vergleichen Sie diese mit der exakten Lösung für den Fall ohne Reibung !*

b. *Bestimmen Sie den Ort* $x_{\max}$, $z_{\max}$ *des Maximums der Flugbahn und vergleichen Sie diese Koordinaten mit der exakten Lösung für den Fall ohne Reibung !*

A3.2* *Nehmen wir an, daß der Ball bei* $z(t) = 0$ *elastisch reflektiert wird* ($v_z \to -v_z$). *Nach wievielen solchen Hüpfern erreicht ein wie in A3.1 abgeworfener Ball noch eine Höhe von* $2,5m$, *bevor er wieder auf den Boden auftrifft ?*

A3.3** *Maximieren Sie die Wurfweite (Flugstrecke) durch Variation von* θ *bei ansonsten gleichen Bedingungen wie in A3.1 !*
Um wieviel können Sie die Wurfweite durch diese Optimierung vergrößern ?

3.1 Mehr elementare Mechanik

Für den schiefen Wurf betrachten wir die x-z-Ebene. Die Lösung (2.3) für den Fall ohne Reibung lautet dann

$$\begin{aligned} x(t) &= x_0 + v_{x,0}\, t\,, \\ z(t) &= z_0 + v_{z,0}\, t - \frac{1}{2}\, g\, t^2\,. \end{aligned} \tag{3.1}$$

Hieraus berechnet sich leicht die Zeit t_{fl}, nach der das Teilchen eine Höhe z_1 erreicht:

$$g\, t_{\text{fl}} = v_{z,0} + \sqrt{v_{z,0}^2 + 2\, g\, (z_0 - z_1)}\,. \tag{3.2}$$

Die zugehörige Flugstrecke folgt sofort aus (3.1) zu $x_{\text{fl}} = x_0 + v_{x,0}\, t_{\text{fl}}$.

Für das Maximum der Bahn (3.1) folgt mit $v_z(t_{\max}) = v_{z,0} - g\, t_{\max} = 0$:

$$t_{\max} = \frac{v_{z,0}}{g}\,, \qquad x_{\max} = x_0 + \frac{v_{x,0}\, v_{z,0}}{g}\,, \qquad z_{\max} = z_0 + \frac{v_{z,0}^2}{2g}\,. \tag{3.3}$$

In diesem Kapitel wollen wir anstatt des Reibungsgesetzes (2.10) eine andere phänomenologische Formel verwenden, die für höhere Geschwindigkeiten eine bessere Beschreibung darstellt:

$$\vec{F}_{R'} = -C\, m\, |\vec{v}|\, \vec{v}\,, \tag{3.4}$$

wobei auch die Konstante C von der Geometrie und Masse des Körpers abhängt und nun die Einheit $1/\mathrm{m}$ besitzt.

Eine wichtige Konsequenz der Reibungsraft (3.4) ist eine Kopplung der Bewegungsgleichung für die verschiedenen Koordinaten (über $|\vec{v}|$) – mit der Reibungskraft (2.10) entkoppeln die einzelnen Komponenten der Bewegungsgleichung.

Durch Nullsetzen der Summe von (2.2) und (3.4) erhalten wir auch hier eine maximale Geschwindigkeit $\vec{v}'_{\max}$, die nun durch folgenden Ausdruck gegeben ist:

$$v'_{x,\max} = v'_{y,\max} = 0\,, \qquad v'_{z,\max} = -\sqrt{\frac{g}{C}}\,. \tag{3.5}$$

3.2 Die nächsten Schritte mit C++

Häufig brauchen wir elementare Funktionen, wie z.B. in folgendem Beispielprogramm:

```
#include <iostream>
#include <cmath>       // Zugriff auf mathematische Funktionen

const double Pi = 3.14159265358979323846264;
                       // Pi wird leider nicht von C++ definiert
int main()
 {
  cout.precision(9);

  cout << cos(Pi/4) << "\t"
       << sin(Pi/4) << "\t"
       << 1/sqrt(2) << "\n";

  cout << cos(Pi/3) << "\t" << sin(Pi/3) << "\n";
  cout << 1/2.0     << "\t" << sqrt(3)/2 << "\n";
 }
```

Die Ausgabe dieses Beispiels lautet:

```
0.707106781     0.707106781     0.707106781
0.5     0.866025404
0.5     0.866025404
```

Diese Ergebnisse sind Ausdruck folgender Identitäten

$$\cos 45^\circ = \sin 45^\circ = \frac{1}{\sqrt{2}}\,, \qquad \cos 60^\circ = \frac{1}{2}\,, \qquad \sin 60^\circ = \frac{\sqrt{3}}{2}\,.$$

Bemerkungen:

1. Beim Programmieren können wir nicht die Grad-Schreibweise für trigonometrische Funktionen verwenden, sondern müssen in Radian umwandeln, also z.B. $\cos\frac{\pi}{4}$ statt $\cos 45^\circ$ schreiben.

2. Leider müssen wir π von Hand eingeben. Es empfiehlt sich, diesen Wert als Konstante zu definieren. In C++ wird eine solche Konstante wie eine Variable definiert und initialisiert, nur daß man der Deklaration

ein ‘`const`’ voranstellt. Konstante können nachträglich nicht verändert werden, müssen deswegen also auch bei der Deklaration initialisiert werden. Der Compiler ersetzt beim Übersetzen jedes Auftauchen der Konstante direkt durch deren Wert. Obwohl Konstanten also genauso wirken, als wären sie überall eingesetzt, bieten sie doch mehrere Vorteile:

- Weniger Tipparbeit (zumindest manchmal).
- Die Möglichkeit, den Wert der Konstante einfach an *einer* Stelle zu ändern, auch wenn sie mehrfach im Programm verwendet wird.
- Bessere Lesbarkeit des Programms, wenn ein bedeutungsvoller Name gewählt wird.

Man sollte auch nicht einfach eine Variable sondern wirklich ‘`const`’ für diesen Zweck verwenden, um z.B. das unbeabsichtigte Ändern der Konstante im Programm zu vermeiden.

3. Wenn Sie

```
#include <cmath>
```

weglassen, läuft das Programm vermutlich genauso gut. Sie sollten trotzdem diese Definitionen einlesen, um sicherzustellen, daß jeder Compiler das gewünschte Ergebnis produziert.

Ein weiteres Beispiel ist der folgende Primzahlentest:

```
#include <iostream>

int main()
 {
  for(int num=3; num<100; num+=2)
   {
    bool isprime = true;          // Ein Flag, ob diese Zahl
                                  // eine Primzahl ist
    int teiler = 3;               // Wir testen Teiler
    while((teiler*teiler <= num)  // von 3 bis sqrt(num),
          && isprime)             // hoeren aber auch auf
     {                            // wenn wir einen Teiler haben
```

```
        if(num % teiler == 0)       // % ist Modulo-Operation
         {
          cout << "\t" << teiler << " teilt " << num << "\n";
          isprime = false;          // doch keine Primzahl
         }
        teiler += 2;                // Wir testen weder gerade
        if((teiler % 3) == 0)       // noch durch 3 teilbare
          teiler += 2;              // Teiler (ausser 3)
       }
      if(isprime)                   // am Ende wissen wir, ob
        cout << num << " ist prim\n";
                                    // die Zahl prim war
    }
  }
```

Dieser Test ist zwar keineswegs optimiert, er soll aber auch nur weitere C/C++-Sprachelemente illustrieren:

1. Mit '`if`' kann man Programmteile (Anweisungen) nur dann ausführen lassen, wenn eine Bedingung erfüllt ist:

   `if(`Bedingung`)` Anweisungen;

   Manchmal will man eine Fallunterscheidung machen. Dann mag '`else`' nützlich sein:

   `if(`Bedingung`)` Anweisungen; `else` Andere Anweisungen;

   Dies führt die ersten Anweisungen aus, wenn die Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, werden die Anderen Anweisungen aus dem zweiten Teil ausgeführt.

2. Es kann vorkommen, daß die Initialisierung einer Schleife besser vor ihr erfolgt und auch die Inkrementierung natürlicher in ihr duchgeführt ist. Für diesen Fall bietet die Konstruktion mit '`while`' eine Alternative zu der mit '`for`'.

   `while(`Abbruchbedingung`)` Anweisungen;

führt die Anweisungen so lange aus, wie die Abbruchbedingung wahr ist, ist also äquivalent zu

`for(;` Abbruchbedingung`;)` Anweisungen;

3. `%` ist die modulo-Operation für ganze Zahlen, d.h. 'a `%` b' liefert den Rest, der bei der ganzzahligen Division 'a `/` b' übrig bleibt.

4. Ergebnisse von Vergleichen haben den C++-Datentyp '`bool`'. Ein `bool`-Datentyp kann nur die Werte '`true`' (wahr) und '`false`' (falsch) annehmen[2].

5. Eine logische Und-Verknüpfung wird mit '`&&`' durchgeführt.

   1. Bedingung `&&` 2. Bedingung

   ist nur dann wahr, wenn sowohl die 1. Bedingung als auch die 2. Bedingung wahr sind.
   Eine logische Oder-Verknüpfung wird mit '`||`' durchgeführt.
   '`!`' negiert eine logische Aussage. Also werden z.B. bei

   `if(!` Bedingung`)` Anweisungen;

   die Anweisungen nur dann ausgeführt, wenn die Bedingung *nicht* erfüllt ist.

[2]'`true`' und '`false`' werden in C++ durch bestimmte ganze Zahlen definiert-

3.3 Prozeduren

Einige Beispiele für Prozeduren (auch Funktionen oder Unterroutinen genannt) haben wir bereits kennengelernt: `main()`, `cos()`, `sin()`, `sqrt()`, ... Das folgende Beispielprogramm definiert die Fakultät $n!$ und einen verallgemeinerten Binomialkoeffizienten $\binom{x}{n}$:

```
#include <iostream>

long factorial(int n)
 {                // Die Fakultaet
  if(n < 0)                        // Bei negativem n
    return 0;                      // mit 0 antworten
  long nfac=1;                     // nfac auf 1 initialisieren
  for(int i=2; i<=n; i++)          // Schleife von i=2 bis n
      nfac *= i;                   // Produkte sammeln
  return nfac;                     // Wert zurueckgeben
 }

double binomial(double x, int n)
 {                // verallg. Binomialkoeffizient n aus x
  if(n < 0)                        // Bei negativem n
    return 0;                      // mit 0 antworten
  double res=1;                    // res auf 1 initialisieren
  for(int i=0; i<n; i++)           // Schleife von i=0 bis n-1
     res *= ((x-i)/double(i+1)); // Faktoren sammeln
  return res;                      // Wert zurueckgeben
 }

int main()
 {
   for(int i=0; i<18; i++)             // Die ersten paar i!
     cout << i << "! = " << factorial(i) << "\n";
   cout << binomial(12, 6) << "\n";  // Binomialkoeffizient
   cout << binomial(120, 3) << "\t"  // Symmetrie testen
        << binomial(120, 117) << "\n";
   cout << binomial(12.34, 5) << "\n";
 }
```

Bemerkungen:

1. Es ist nicht nur nützlich, häufig verwendete Programmteile als Prozeduren zu programmieren, sondern steigert (zumindest bei längeren Programmen) die Übersichtlichkeit.

2. Jede Prozedur hat einen Rückgabewert, der bei ihrer Definition als Datentyp vor den Namen gestellt wird. Gibt eine Prozedur keinen Wert zurück, sollte sie als '`void`' deklariert werden.

3. Die Argumente einer Prozedur werden von Klammern '`()`' eingeschlossen und durch Kommas '`,`' getrennt. Der Datentyp jedes Arguments sollte in der Definition der Prozedur deklariert werden.

4. Werte der Argumente werden von einer Prozedur nicht verändert.

5. Der Rückgabewert r wird mit '`return` r`;`' zurückgegeben.

6. Der Inhalt der Prozedur steht in geschweiften Klammern '`{}`'.

7. Der Datentyp '`long`' definiert eine ganze Zahl mit mindestens sovielen Stellen wie '`int`'. Die Fakultät liefert selbst für kleine Argumente schnell Ergebnisse außerhalb des Bereichs ganzer Zahlen. Somit sind möglichst viele Stellen nützlich.

8. 'n`++`' erhöht den Wert von n um 1, nachdem er ggfs. in einer Berechnung ausgelesen wurde. Die Schleife in der Funktion '`factorial()`' könnte also auch wie folgt aussehen:

   ```
   for(int i=2; i<=n; ) nfac *= i++;
   ```

   Allerdings wäre dies deutlich unübersichtlicher. Hingegen liefert

   ```
   for(int i=2; i<=n; ) nfac *= ++i;
   ```

   nicht das gewünschte Ergebnis, denn '`++`n' erhöht den Wert von n um 1 und verwendet *anschließend* diesen Wert in der Berechnung.

9. '`*=`' multipliziert den Wert einer Variablen mit dem Ausdruck auf der rechten Seite.

10. Man kann sich überzeugen, daß der verallgemeinerte Binomialkoeffizient folgende Beziehung zur verallgemeinerten Fakultät $n! = \Gamma(n+1)$ besitzt:
$$\binom{x}{n} = \frac{\Gamma(x+1)}{\Gamma(n+1)\,\Gamma(x-n+1)}\,. \tag{3.6}$$

Als physikalische Anwendung wollen wir vielleicht die Erdanziehung (2.2) alleine bzw. unter Einbeziehung der Reibungskräfte (2.10) und (3.4) als Prozeduren programmieren:

```
#include <cmath>

const double g=9.80665;
const double lambda=0.7;
const double C=0.01;

double b_frei(double v)
 {          // Freier Fall
  return -g;
 }

double b_lin(double v)
 {          // Reibung nach (2.10)
  return -g-lambda*v;
 }

double b_quad(double v)
 {          // Reibung nach (3.4)
  return -g-C*abs(v)*v;     // Die Funktion abs() liefert Betrag
 }
```

3.4 Einfache grafische Ausgabe

Wahrscheinlich brennen Sie längst darauf, Ihre Ergebnisse endlich auf dem Bildschirm darzustellen. Am schnellsten geht das mit einem Plot-Programm. Dazu müssen aber erst die Ergebnisse in Dateien geschrieben werden. Das folgende Programm simuliert eine eindimensionale Bewegung unter Verwendung der Unterroutinen aus Kapitel 3.3 und schreibt die Ergebnisse für Zeit

t und Geschwindigkeit v spaltenweise in eine Ausgabedatei:

```
#include <iostream>
#include <fstream>

const double deltaT = 0.01;

int main()
 {
  ofstream datei("v_frei.dat",
                 ios::out);          // Ausgabedatei oeffnen

  double v=0;                        // Anfangsgeschwindigkeit 0

  for(double t=0; t<10.0001; t += deltaT)
    {
     datei << t << "\t" << v << "\n";
     double a    = b_frei(v);        // Euler-Richardson Schritt
     double vmid = v + a*deltaT/2;
     double amid = b_frei(vmid);
     double vneu = v + amid*deltaT;
     v = vneu;                       // aktuellen Wert uebernehmen
    }
  datei.close();                     // Datei schliessen (optional)
 }
```

Zum Befehl, der die Ausgabedatei öffnet

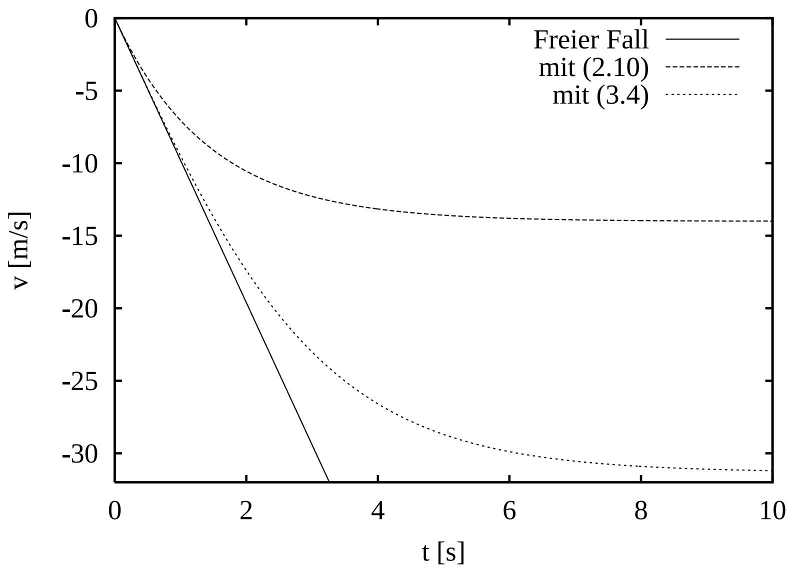
```
ofstream datei("v_frei.dat", ios::out);
```

sei hier nur bemerkt, daß '`v_frei.dat`' der Name der Datei und '`datei`' ein Objekt ist, das hinterher genau wie '`cout`' verwendet werden kann. Dateien werden bei Beendigung des Programmes automatisch geschlossen, im obigen Beispiel könnte also die explizite Schließanweisung '`datei.close();`' weggelassen werden.

Nehmen wir an, daß wir mit obigem Programm neben '`v_frei.dat`' auch Dateien '`v_lin.dat`' und '`v_quad.dat`' für die beiden anderen Kräfte erzeugt haben. Diese Dateien können nun mit einem Plot-Programm dargestellt werden. Wir verwenden als Beispiel `gnuplot`. Die folgenden Eingaben

```
gnuplot> set xlabel "t [s]"
gnuplot> set ylabel "v [m/s]"
gnuplot> set yrange [-32:0]
gnuplot> plot "v_frei.dat" t "Freier Fall" w l
gnuplot> replot "v_lin.dat" t "mit (2.10)" w l
gnuplot> replot "v_quad.dat" t "mit (3.4)" w l
```

führen zu einer Ausgabe, die ungefähr wie folgt aussieht:



Dabei

- setzt '`set xlabel`' die Bezeichnung der x-Achse,
- '`set ylabel`' die Bezeichnung der y-Achse,
- '`set yrange`' den Bereich auf der y-Achse,
- '`plot "v_frei.dat"`' plottet die Datei '`v_frei.dat`' mit einem Titeltext ('`t`') 'Freier Fall' und unter Verwendung von Linien ('`w l`'),
- '`replot "v_lin.dat"`' plottet den zweiten Fall *dazu* und
- '`replot "v_quad.dat"`' plottet den dritten Fall zu den beiden anderen hinzu.

In diesem Beispiel beobachten wir, daß die lineare Reibungskraft (2.10) bereits deutlich früher zu merklichen Abweichungen von der reibungsfreien Bewegung führt als die quadratische (3.4), wie ja auch zu erwarten ist.

4 Kepler-Probleme

A4.1 *Wir betrachten einen Satelliten im Erdumlauf. In diesem Fall ist es bequem, Abstände in Einheiten des Erdradius $R_E = 6,37\,10^6 m$ zu messen und die Zeit in Stunden h. Der Wert der Gravitationskonstante G ist in diesen Erdeinheiten (EU)*

$$\begin{aligned} G &= 6,67\,10^{-11}\frac{m^3}{kg\,s^2}\left(\frac{1\,EU}{6,37\,10^6 m}\right)^3\left(3,6\,10^3\frac{s}{h}\right)^2 \\ &= 3,34\,10^{-24}\frac{EU^3}{kg\,h^2}\,. \end{aligned} \tag{4.1}$$

Die Kraft auf den Satelliten ist proportional zu $G\,M$, wobei M die Erdmasse ist. Diese berechnen wir zu

$$\begin{aligned} G\,M &= 3,34\,10^{-24}\frac{EU^3}{kg\,h^2}\,5,99\,10^{24}kg \\ &= 20,0\frac{EU^3}{h^2}\,. \end{aligned} \tag{4.2}$$

Wir nehmen ferner an, daß die Reibungskraft proportional zum Quadrat der Geschwindigkeit des Satelliten ist (3.4). Um in vernünftiger Zeit einen Effekt zu beobachten, setzen wir für C einen Wert ein, so daß die Reibungskraft ungefähr 1/10 der Gravitationskraft beträgt.
Wählen Sie Anfangsbedingungen, so daß Sie ohne Luftwiderstand eine kreisförmige Umlaufbahn erhalten ! Erlauben Sie mindestens einen Umlauf, bevor Sie den Luftwiderstand 'einschalten' ! Beschreiben Sie qualitativ die Änderung der Umlaufbahn aufgrund der Reibung ! Wie ändern sich Gesamtenergie und Geschwindigkeit des Satelliten aufgrund der Reibung ?

A4.2* *Wir wollen einen vereinfachten Raketenflug in eine Mondumlaufbahn simulieren. Dazu brauchen wir den Mondradius*

$$R_M = 1,738\,10^6 m = 0,27\,EU\,, \tag{4.3}$$

die Masse m des Mondes, bzw. die Kombination

$$G\,m = 0,0123\,G\,M = 0,246\frac{EU^3}{h^2} \tag{4.4}$$

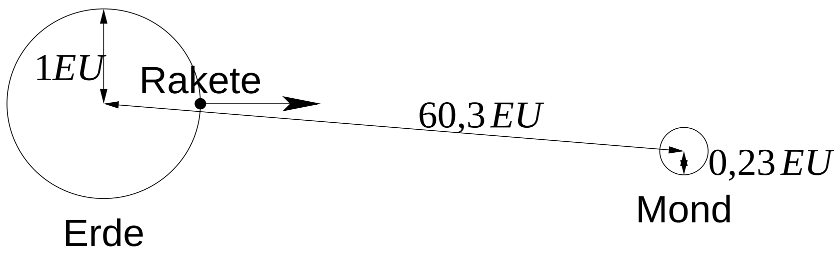
und den (mittleren) Abstand Mond-Erde

$$d_{E-M} = 3,84\,10^8\,m = 60,3\,EU\,. \qquad (4.5)$$

Wir vernachlässigen die Bewegung von Erde und Mond, nehmen also an, daß der Mond an einem festen Punkt im Abstand d_{E-M} von der Erde steht.
Für die Rakete mit Masse m_R nehmen wir an, daß wir einen Schub vom Betrag

$$|\vec{a}| = \frac{|\vec{F}|}{m_R} = 150\frac{m}{s^2} = 150\frac{m}{s^2}\,\frac{1\,EU}{6,37\,10^6 m}\left(3,6\,10^3\frac{s}{h}\right)^2 = 305\frac{EU}{h^2} \qquad (4.6)$$

nach Belieben ein- und ausschalten können[3]*. Ferner nehmen wir vereinfachend an, daß wir die Richtung dieses Schubes in jedem Zeitpunkt frei wählen können.*
Starten Sie die Rakete an einem geeigneten Ort senkrecht zur Erdoberfläche ! Bringen Sie die Rakete durch geeignete Steuerung in eine stabile Mondumlaufbahn, d.h. umkreisen Sie den Mond mindestens 10 mal ohne Schub anzuschalten !
Optimieren Sie Ihre Steuerung so, daß Sie mit einer möglichst geringen Antriebszeit hinkommen[4] *!*



A4.3** *Lösen Sie A4.2, indem Sie bei den Anfangsbedingungen berücksichtigen, daß sich die Erde in 24h einmal um sich selbst dreht (Position von Erde und Mond seien weiterhin fest) !*

[3]Dies entspricht etwa dem maximalen Schub der 1. Stufe einer Saturn V-Rakete.
[4]Eine Saturn V-Rakete hatte eine Gesamtbrenndauer von $0,29h$.

4.1 Planetare Umläufe

Für die Planetenbahnen gelten die bekannten Keplerschen Gesetze:

1. Die Planetenbahnen sind Ellipsen. Die Sonne steht in einem Brennpunkt dieser Ellipsen.

2. Die Geschwindigkeit eines Planeten erhöht sich bei der Annäherung an die Sonne so, daß die Verbindungslinie von der Sonne zum Planten gleiche Flächen in gleichen Zeiten überstreicht.

3. Die Quadrate T^2 der Umlaufzeiten der Planeten verhalten sich wie die dritten Potenzen a^3 der großen Bahnhalbachsen (T^2/a^3 ist konstant).

Die Bewegung der Erde um die Sonne ist ein Bespiel für ein Zweikörperproblem, bei dem die potentielle Energie nur von der relativen Distanz abhängt. Ein solches Problem läßt sich durch Einführung von Schwerpunkt- und Relativkoordinate auf ein effektives Einkörperproblem zurückführen. Sind m und M die Massen der beiden Körper, so tritt hierbei die reduzierte Masse

$$\mu = \frac{M\,m}{m+M} \tag{4.7}$$

auf. Die Masse der Erde $m = 5,99\,10^{24}$kg ist viel kleiner als die Masse der Sonne $M = 1,99\,10^{30}$kg. Man kann deswegen für die meisten praktischen Anwendungen die reduzierte Masse des Sonne-Erde Systems durch die der Erde alleine nähern $\mu = m$ und die Sonne als stationär annehmen.

Ein Körper der Masse M zieht einen anderen Körper der Masse m mit der Gravitationskraft an:

$$\vec{F} = -\frac{G\,M\,m}{r^3}\,\vec{r}\,. \tag{4.8}$$

Hierbei ist r der Vektor, der von M nach m zeigt, und $r = |\vec{r}| = \sqrt{\vec{r}\cdot\vec{r}}$ dessen Länge. Für den Wert der Gravitationskonstanten G verwenden wir

$$G = 6,67\,10^{-11}\frac{\mathsf{m}^3}{\mathsf{kg\,s}^2}\,. \tag{4.9}$$

Das Kraftgesetz (4.8) ist ein Beispiel für eine Zentralkraft, bei der die potentielle Energie V nur von dem Abstand $\vec{r}$ abhängt. Bei Zentralkräften ist der Drehimpuls

$$\vec{L} = \vec{r}\times\vec{p} \tag{4.10}$$

($\vec{p} = m\vec{v}$) erhalten. Man kann deswegen das Koordinatensystem so wählen, daß $\vec{L}$ in z-Richtung zeigt: $\vec{L} = \left|\vec{L}\right| \vec{e}_z$. Die Bewegung verläuft dann in der x-y-Ebene. Wir wollen deswegen die Planetenbewegungen grundsätzlich als zweidimensional ansehen und die z-Koordinate nicht betrachten.

Ist der Körper mit der Masse M fest, so ist die Gesamtenergie

$$E = m \left(\frac{v^2}{2} - \frac{G\,M}{r} \right) . \tag{4.11}$$

Aus dem Kraftgesetz (4.8) folgt mit (2.1) die Bewegungsgleichung

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\,\vec{r} = -\frac{G\,M}{r^3}\,\vec{r}\,. \tag{4.12}$$

Diese Bewegungsgleichung läßt sich bekanntlich geschlossen lösen. Man findet im allgemeinen Kegelschnitte und für $E < 0$ Ellipsen (1. Keplersches Gesetz).

Häufig können die Planetenbahnen gut durch Kreisbahnen approximiert werden. So schwankt z.B. der Abstand Erde-Sonne lediglich um 1-2%. Wir wollen deswegen kurz Kreisbahnen betrachten. Man überzeugt sich leicht, daß in diesem Fall der Betrag der Beschleunigung

$$\left| \frac{\mathrm{d}^2}{\mathrm{d}t^2}\,\vec{r} \right| = \frac{v^2}{r} \tag{4.13}$$

ist. Durch Einsetzen in (4.12) findet man damit

$$v = \sqrt{\frac{G\,M}{r}}\,. \tag{4.14}$$

Für die Umlaufdauer $T = \frac{2\pi\,r}{v}$ folgt also

$$T^2 = \frac{4\pi^2}{G\,M}\,r^3\,. \tag{4.15}$$

Diese Beziehung ist die Aussage des 3. Keplerschen Gesetzes für den speziellen Fall einer Kreisbahn, wobei der Radius r gleich der großen Halbachse a ist.

Die Bewegungsgleichung (4.12) ist ein Spezialfall der gekoppelten Bewegungsgleichungen für N Körper der Massen m_i, die miteinander gravitativ wechselwirken:

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\,\vec{r}_i = -G \sum_{\substack{j=1 \\ j\neq i}}^{N} m_j\,\frac{\vec{r}_i - \vec{r}_j}{\left|\vec{r}_i - \vec{r}_j\right|^3}\,. \tag{4.16}$$

4.2 Einheiten im Sonnensystem

Für eine numerische Lösung empfiehlt es sich grundsätzlich, an das Problem angepaßte Einheiten zu verwenden. Bei einem Zentralkraftproblem sollten diese so gewählt werden, daß der Zahlenwert von $G\,M$ in der Nähe von 1 liegt. Zur Beschreibung des Sonnensystems verwendet man als Längeneinheit die astronomische Einheit (AU), die gleich der großen Halbachse der Erdumlaufbahn (d.h. im wesentlichen deren mitteleren Radius) gesetzt wird:

$$1\,AU = 1,496\,10^{11}\text{m}\,. \tag{4.17}$$

Als Zeiteinheit verwendet man das Jahr, $1\,yr \approx 3,15\,10^7$s. Nun kann man leicht die dem 3. Keplerschen Gesetz entsprechende Beziehung (4.15) nach dem Produkt von Gravitationskonstante und Sonnenmasse auflösen:

$$G\,M = 4\pi^2\,\frac{AU^3}{yr^2}\,. \tag{4.18}$$

4.3 Vektoren

Bei der numerischen Lösung der Bewegungsleichungen (4.12) und (4.16) ist es bequemer, mit Vektoren zu arbeiten, als wie z.B. im Kapitel 3 für jede Komponente eine eigene Variable zu verwenden. Wir verwenden dazu C-Arrays[5]. Zur Illustration diene das folgende Programm, das in drei Dimensionen das Kreuzprodukt (vgl. die Definition des Drehimpulses (4.10)) und das Skalarprodukt implementiert:

```
#include <iostream>

const int dim=3;           // Nur Dimension 3

double skalar(double a[dim], double b[dim])
 {                         // Skalarprodukt der Vektoren a und b
  double summe=0;
  for(int i=0; i<dim; i++)
      summe += a[i]*b[i];
  return summe;
 }

void kreuz(double a[dim], double b[dim], double r[dim])
 {                         // r = a x b
  for(int k=0; k<dim; k++)
   {                       // Schleife ueber Zielkomponenten
    int i = (k+1) % dim;
    int j = (k+2) % dim;
    r[k] = a[i]*b[j] - b[i]*a[j];
   }
 }
```

[5] C++ stellt eine `vector`-Klasse zur Verfügung, die für echte Vektoren vorzuziehen wäre. Wir verwenden sie aber nicht, da wir mit C-Arrays einfacher zu mehr als einem Index übergehen können.

```
int main()
 {
  double e1[dim] = {1, 0, 0};    // 1. Einheitsvektor
  double e2[dim] = {0, 1, 0};    // 2. Einheitsvektor
  double a[dim] = {2, -3, -4};   // a definieren und initialisieren
  double b[dim] = {6, 5, 1};     // b definieren
  double r[dim];                 // r definieren, aber nicht
                                 // initialisieren

  cout << "e1 * e2 = " << skalar(e1, e2) << "\n";
  cout << "e1 * b = " << skalar(e1, b) << "\n";
  cout << "a * b = " << skalar(a, b) << "\n";
  kreuz(e1, e2, r);
  cout << "e1 x e2 = [" << r[0] << ", " << r[1]
                       << ", " << r[2]<<"]\n";
  kreuz(a, b, r);
  cout << "a x b = [" << r[0] << ", " << r[1]
                     << ", " << r[2]<<"]\n";
  cout << "(a x b) * (a x b) = " << skalar(r, r) << "\n";
 }
```

Dieses Programm gibt folgende (nicht sehr tiefschürfende) Ergebnisse aus:

```
e1 * e2 = 0
e1 * b = 6
a * b = -7
e1 x e2 = [0, 0, 1]
a x b = [17, -26, 28]
(a x b) * (a x b) = 1749
```

Damit haben wir primär folgendes für den Umgang mit Vektoren illustriert:

1. Ein Vektor wird über

   Datentyp Name`[`n`]`;

   deklariert. Dabei muß die Dimension n eine ganze Zahl sein, die aber z.B. auch als '`const int`' definiert sein kann.

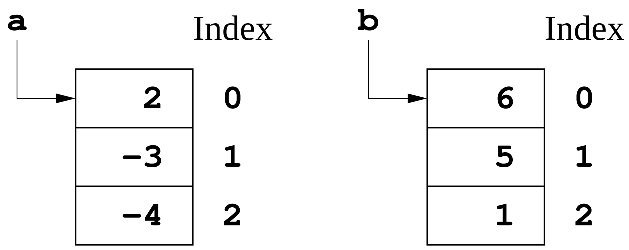
2. In C/C++ fangen die Indizes von Vektoren (und Arrays) bei 0 an, laufen also bis $n-1$, wenn n die Dimension ist.

3. Name[i]
   bezeichnet das $i$te Element eines Vektors mit dem angegebenen Namen. Ein solcher Ausdruck kann genauso wie ein einfacher Variablenname des gleichen Datentyps verwendet werden.

4. Vektoren können bei der Deklaration komplett initialisiert werden. Dazu werden die Werte für die einzelnen Elemente nacheinander durch Kommas '`,`' getrennt in geschweiften Klammern '`{}`' angegeben.
   Achtung: Die Anzahl der bei der Initialisierung angegebenen Werte muß nicht (sollte aber in der Regel) mit der Länge des Vektors übereinstimmen.

5. Der Compiler reserviert bei der Deklaration ausreichend Speicherplatz für alle Elemente, die Dimension ist danach nicht veränderbar. Die Elemente werden in aufsteigender Reihenfolge abgelegt.

6. Der Name des Arrays zeigt auf die Startadresse des Arrays, d.h. auf das erste (mit `0` indizierte) Element. Im obigen Beispielprogramm sieht das für die Vektoren `a` und `b` wie folgt aus:



7. Bei einem Funktionsaufruf wird nur der Name des Arrays übergeben. Die Prozedur erhält damit nur den Zeiger auf den Speicherplatz, die einzelnen Elemente des Arrays werden bei der Übergabe *nicht* kopiert.

8. Bei der Deklaration einer Prozedur müssen den Namen der Arrays eckige Klammern '`[]`' nachgestellt werden, damit diese erkannt werden. Die Dimension eines Vektors kann an dieser Stelle weggelassen werden und wird, falls angegeben, tatsächlich vom Compiler ignoriert. Wir hätten also die Prozedur für das Skalarprodukt auch wie folgt deklarieren können:

```
double skalar(double a[], double b[])
```

9. Da bei einem Funktionsaufruf nur der Zeiger auf den Speicherplatz übergeben wird, kann die Prozedur den *Inhalt* des Arrays verändern – im Gegensatz zu 'normalen' Variablen, die als Kopie übergeben werden und deren Wert an der aufrufenden Stelle also von der Unterroutine nicht direkt geändert werden kann. Dies nutzen wir in der Prozedur '`kreuz`' aus, um den Wert des Kreuzproduktes in dem Vektor '`r`' zurückzugeben[6].
   Achtung: Hat ein Array bei einem Funktionsaufruf noch keinen sinnvollen Inhalt, so muß es doch vor dem Aufruf deklariert werden, damit der nötige Speicher reserviert wird.

[6] Die Rückgabe kompletter Arrays aus Prozeduren mittels '`return`' ist nicht möglich.

4.4 Ein Programm für unser Sonnensystem

Als Beispiel sei hier ein Programm `kepler.cc` angegeben, das die Bewegungsgleichung (4.16) mit dem Euler-Richardson-Algorithmus (2.8,2.9) für $N =$ `nobj` Körper löst, von denen `aktZahl` aktualisiert werden (d.h. `nobj` − `aktZahl` Körper werden festgehalten)[7]. Zuerst kommen einige Definitionen:

```
const double Pi = 3.14159265358979323846264;

const int rdim = 2;             // 2 raeumliche Dimensionen
const int nobj = 3;             // Die Anzahl der Objekte

double Gm[nobj] = {             // Die Produkte G m in AU
          4*Pi*Pi,              // Sonne
          0.000119,             // Erde
          0.037789              // Jupiter
       };

const int aktZahl = 2;          // Zahl der zu aktualisiernden
                                // Koerper
int akt[aktZahl] = {1, 2};      // und deren Indizes

char *name[aktZahl] = {         // die "Datei"-Namen
   "erde.dat", "jupiter.dat" }; // dazu

double r0[nobj] = {0, 1, 5.2};  // Bahnradien von Sonne, Erde, Jupiter

const double deltaT = 0.005;    // Die Schrittweite

const double Tgesamt = 10*11.86;  // 10 Jupiter-Umlaeufe
```

Neu sind hier zwei Kleinigkeiten:

1. Der C/C++-Datentyp '`char`', der für Zeichen steht (eine Zeichenkette ist ein Array von Zeichen).

[7] Eine erweiterte Version des Quellcodes steht auf der Kurs-Homepage zum Download zur Verfügung.
C++-Kenner mögen den C-artigen Stil verzeihen. Wenn man C++ gut beherrscht, geht es sicher eleganter, aber wir wollen schnell zu Ergebnissen kommen.

2. Das '*' vor dem Variablennamen signalisiert einen *Zeiger* auf Daten des angegebenen Typs. Hier verwenden wir es, um ein Array zu definieren, das mehrere Zeichenketten enthält.

Als nächstes kommt die erste Prozedur des eigentlichen Programms. Sie berechnet die Beschleunigung, die der Körper i nach (4.16) durch die anderen erfährt:

```
#include <cmath>

void beschl(int i, double r[nobj][rdim], double b[rdim])
 {
  double rij[rdim];
  double rabs, temp;

  for(int n=0; n<rdim; n++)     // b auf Null
     b[n] = 0;                  // initialisieren

  for(int j=0; j<nobj; j++)     // Schleife ueber alle
    if(j != i)                  // ANDEREN Objekte
      {
        temp = 0;               // temp auf Null initialisieren
        for(int n=0; n<rdim; n++)
          {
            rij[n] = r[i][n] - r[j][n]; // Differenzvektor bestimmen
            temp += (rij[n]*rij[n]);    // Quadrat in temp
          }
        rabs = sqrt(temp);      // Wurzel ziehen
        temp *= rabs;           // davon die 3. Potenz in temp

        for(int n=0; n<rdim; n++)       // Schliesslich zu b addieren
            b[n] -= (Gm[j]*rij[n])/temp;
      }
 }
```

Hier tauchen nun erstmalig Arrays mit mehreren Indizes auf. Diese werden mit *mehreren* eckigen Klammern '[]' angegeben (eine Trennung durch Kommas ist *nicht* zulässig !). Damit die Prozedur korrekt mit den Arrays umgehen kann, *müssen* alle Dimensionen (mit Ausnahme der ersten) bei der Deklaration angegeben werden. Statt 'double r[nobj][rdim]' dürfte man also auch 'double r[][rdim]' verwenden, das 'rdim' darf aber nicht auch noch weggelassen werden.

Die Prozedur beschl() wird von der nächsten verwendet, die einen Euler-Richardson-Schritt mit $\Delta t =$ deltaT nach (2.8) und (2.9) implementiert:

```
void eulerRichardson(double r[nobj][rdim],
                     double v[nobj][rdim], double deltaT)
 {
  double be[aktZahl][rdim], bmid[aktZahl][rdim];
  double rmid[nobj][rdim], vmid[nobj][rdim];

  for(int i=0; i<aktZahl; i++)  // Erst einmal alle
    beschl(akt[i], r, be[i]);   // Beschleunigungen berechnen

  for(int i=0; i<nobj; i++)
   for(int n=0; n<rdim; n++)
    {                           // Sicherheitshalber
     rmid[i][n] = r[i][n];      // r und
     vmid[i][n] = v[i][n];      // v in rmid und vmid
    }                           // kopieren

  for(int i=0; i<aktZahl; i++)  // Dann ein Euler-Schritt
   for(int n=0; n<rdim; n++)    // mit deltaT/2
    {
     rmid[akt[i]][n] += v[akt[i]][n]*deltaT/2;
     vmid[akt[i]][n] += be[i][n]*deltaT/2;
    }

  for(int i=0; i<aktZahl; i++)  // Beschleunigungen noch
    beschl(akt[i], rmid, bmid[i]);     // einmal berechnen

  for(int i=0; i<aktZahl; i++)  // Und schliesslich
   for(int n=0; n<rdim; n++)    // der Schritt ueber deltaT
    {                           // unter Verwendung von
     r[akt[i]][n] += vmid[akt[i]][n]*deltaT;  // vmid
     v[akt[i]][n] += bmid[i][n]*deltaT;       // und bmid
    }
 }
```

Hier sei nur darauf hingewiesen, daß z.B. bei dem ersten Aufruf von `beschl()`, durch das Argument '`be[i]`' ein Zeiger auf ein Array mit einem Index weniger als `be` übergeben wird, also ein Vektor.

Schließlich kommt das Hauptprogramm:

```
#include <iostream>
#include <fstream>

int main()
 {
  double r[nobj][rdim], v[nobj][rdim];
  ofstream ausgabe[aktZahl];

  for(int i=0; i<aktZahl; i++)  // Ausgabedateien oeffnen
    ausgabe[i].open(name[i], ios::out);

  for(int i=0; i<nobj; i++)     // Initialisierung
   {
    r[i][0] = r0[i];            // x-Koordinate von r
    r[i][1] = 0;                // y-Koordinate auf 0
    v[i][0] = 0;                // Anfangsgeschwindigkeit
    if(! i)                     // so, dass es eine Kreis-
      v[i][1] = 0;              // bahn um die Sonne gibt
    else
      v[i][1] = sqrt(Gm[0]/r0[i]);
   }

  for(double t=0; t<=Tgesamt; t += deltaT)
   {
    eulerRichardson(r, v, deltaT);   // Zuerst ein Euler-
                                     // Richardson-Schritt

    for(int i=0; i<aktZahl; i++)     // Dann die x-
      ausgabe[i] << r[akt[i]][0]     // und y-Koordinaten
                 << "\t"             // ausgeben
                 << r[akt[i]][1] << "\n";
   }
```

```
  for(int i=0; i<aktZahl; i++)
     ausgabe[i].close();         // Ausgabedateien schliessen
}
```

In diesem Programmteil wird ein Array von Ausgabedateien angelegt, damit die Bahnkurven der sich bewegenden Körper in separate Ausgabedateien geschrieben werden können – im Beispiel in '`erde.dat`' und '`jupiter.dat`'.

Das Hauptprogramm initialisiert zuerst Ort und Geschwindigkeit unter Verwendung von (4.14) so, daß sich unter Vernachlässigung der Wechselwirkung zwischen Körpern i und j mit $i, j > 0$ Kreisbahnen um den Körper $i = 0$ bei $\vec{r}_0 = 0$ ergeben würden (im Beispiel die Sonne). Dann wird für einige Zeit die Bewegung der ausgewählten Körper simuliert.

Wird die Schrittweite Δt zu klein gewählt, wird ein Körper nach außen weggeschleudert. Dies wird besonders augenfällig, wenn man z.B. anstelle des Euler-Richardson-Algorithmus (2.8,2.9) das einfache Euler-Verfahren (2.7) mit dem gleichen Δt verwendet. Die Tatsache, daß der Körper nach außen wegfliegt, ist anschaulich einsichtig, da die lineare bzw. quadratische Approximation an die Bahnkurve in jedem Fall deren Krümmung unterschätzt.

5 Dreikörperproblem

A5.1 *Wir betrachten das System Erde-Mond und wählen ein Koordinatensystem, dessen Ursprung im Erdmittelpunkt liegt. Die potentielle Energie einer Probemasse m_P ist dann entlang der Erde-Mond Verbindungslinie*

$$V(r) = -\frac{G\,M\,m_P}{r} - \frac{G\,m\,m_P}{d_{E-M} - r}\,. \tag{5.1}$$

a. *Verwenden Sie Java, um $V(r)/m_P$ von der Erd- bis zur Mondoberfläche in Schritten von $0,1\,EU$ auszugeben !*

b. *Bestimmen Sie auf dieser Linie die Position $r_{\max}$ des Maximums von V (d.h. die Potentialschwelle, die es beim Mondflug zu überwinden gilt) !*

Zur Erinnerung:
$R_E = 1\,EU$, $R_M = 0,27\,EU$, $d_{E-M} = 60,3\,EU$,
Erde: $G\,M = 20,0\,EU^3/h^2$, *Mond*[8]: $G\,m = 0,246\,EU^3/h^2$.

A5.2 *Die Informationen von A5.1 sollen nun grafisch dargestellt werden. Verwenden Sie 15 Pixel für $1\,EU$!*

a. *Öffnen Sie ein Fenster der Größe 930×300 mit weißem Hintergrund !*

b. *Malen Sie an dessen linken Rand in der Mitte einen ausgefüllten blauen Kreis, der die Erde maßstabsgerecht darstellt !*

c. *Zeichen Sie an den rechten Rand in der Mitte einen ausgefüllten gelben Kreis, der den Mond maßstabsgerecht darstellt !*

d. *Schreiben Sie jeweils darüber in Schwarz an den oberen Rand des Fensters die Texte '*`Erde`*' bzw. '*`Mond`*'.*

e. *Plotten Sie zwischen den beiden Kreisen grün das Potential (5.1) unter Verwendung einer geeigneten Skala für V !*

f. *Zeichen Sie um das Maximum von V einen roten Kreis mit einem Durchmesser von 7 Pixeln !*

[8] Dieser Zahlenwert war leider in einer früheren Version des Skriptes um den Faktor 10 zu groß.

A5.3** *Programmieren Sie ausgehend von dem Programm aus A5.2 eine Animation des Raketenflugs zum Mond nach A4.2 bzw. A4.3 !*

A5.4 *Simulieren Sie die Bewegung des Dreikörper-Systems Sonne-Erde-Mond und stellen Sie die Bewegung der drei Körper grafisch dar ! Setzen Sie zu Beginn die Sonne mit* $\vec{v}_0 = 0$ *ins Zentrum* $\vec{r}_0 = 0$*, lösen aber anschließend auch ihre Bewegungsgleichung ! Wählen Sie die übrigen Anfangsbedingungen so, daß die Erde eine Kreisbahn um die Sonne beschreibt und der Mond eine Kreisbahn um die Erde ausführt.*

Zahlenwerte (in astronomischen Einheiten):

Abstand Sonne-Erde:	$r_E = 1\,AU$, *(vgl. (4.17))*
Abstand Erde-Mond:	$d_{E-M} = 2,57\,10^{-3}\,AU$
Sonnemasse:	$G\,M = 4\pi^2\,\frac{AU^3}{yr^2}$ *(vgl. (4.18))*
Erdmasse:	$G\,m_E = 1,19\,10^{-4}\,\frac{AU^3}{yr^2}$
Mondmasse:	$G\,m_M = 1,46\,10^{-6}\,\frac{AU^3}{yr^2}$

A5.5* *Erweitern Sie das Programm aus A5.4 so, daß Sie 3 Ansichten haben:*

a. *Das Gesamtsystem Sonne-Erde-Mond,*

b. *das System Erde-Mond im Ruhesystem der Erde (damit man die Bewegung des Mondes genau verfolgen kann) und*

c. *die Sonne, bzw. deren Schwerpunkt (damit man die Bewegung der Sonne verfolgen kann, bzw. den Anteil, der durch die Erde und den Mond verursacht wird).*

Zahlenwerte zur maßstabsgerechten Darstellung:

Sonnenradius:	$R_S = 4,65\,10^{-3}\,AU$
Erdradius:	$R_E = 4,26\,10^{-5}\,AU$
Mondradius:	$R_M = 1,61\,10^{-5}\,AU$

5.1 Ganz grundlegende Bemerkungen zu Java

Java ist C bzw. C++ sehr ähnlich, so daß wir viel von dem bisher Gelernten einfach übertragen können. Leider gibt es im Detail aber auch einige Unterschiede, wie wir bereits in unserem ersten Beispiel sehen werden. Andere Konstruktionen gibt es im Prinzip auch in C++, nur haben wir sie bisher noch nicht kennengelernt.

Zuerst betrachten wir eine Java-Variante des ersten Programms aus Kapitel 2.2. Das Programm soll `first.java` heißen:

```
public class first {
 public static void main(String[] args)
  {
   System.out.println("Ein allererstes Programm in Java");
  }
}
```

Auch in Java muß der Programm-Quelltext compiliert werden. Das Kommando für den Compiler-Aufruf heißt nun:

```
javac first.java
```

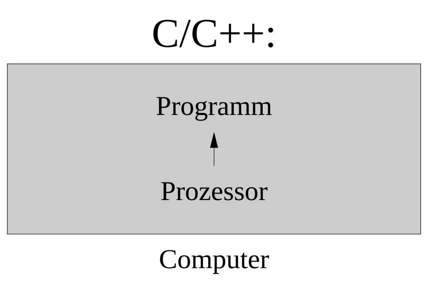
Nun haben wir eine Datei `first.class`. Allgemeiner erzeugt `javac` aus einer Datei 'Name`.java`' eine Datei 'Name`.class`'. Dieses Java-Programm kann man wie folgt ausführen:

```
java first
```

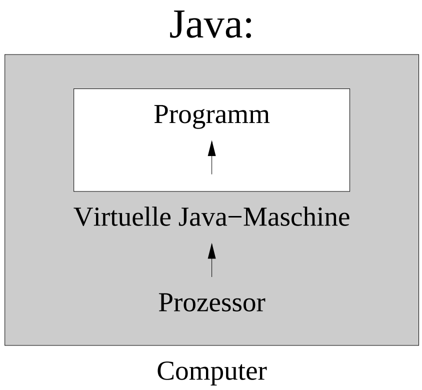
Nun sollte in der Konsole die folgende Ausgabe erscheinen:

```
Ein allererstes Programm in Java
```

Java hat einen fundamentalen Unterschied zu C/C++ und anderen Compilern. So erzeugt C/C++ ein Programm (z.B. `first`), das von dem Prozessor des Computers direkt ausgeführt wird:



Java hingegen erzeugt einen speziellen Bytecode (z.B. `first.class`), zu dessen Ausführung ein weiteres Programm, eine Virtuelle Java-Maschine benötigt wird. Das Kommando `java` ruft die Virtuelle Java-Maschine auf, die dann den Bytecode interpretiert und ausführt:



Diese Konzeption mit einer virtuellen Maschine und einem Bytecode hat verschiedene Vorteile, so z.B.:

1. **Portabilität**
   Ist er einmal auf einem Entwicklungsrechner erzeugt worden, kann der Bytecode auf jede beliebige Plattform kopiert werden (vorausgesetzt es gibt dort eine Virtuelle Java-Maschine) und verhält sich immer gleich. Es gibt für Java auch verschiedene Schnittstellen, die die gleiche Portabilität besitzen.
   So gibt es Pakete, die einen einfachen und Betriebssystemunabhängigen Zugriff auf Grafik ermöglichen. Der einfache Zugriff auf Grafik ist auch der Hauptgrund, warum wir uns hier mit Java beschäftigen – in Kapitel 5.4 wird das Abstract Window Toolkit eingeführt.

2. **Sicherheit**
   Der Java-Bytecode besitzt keinen direkten Zugriff auf den Computer, sondern läuft in einer Art Sandkasten. Die virtuelle Maschine kann dann so eingestellt werden, daß sie den Zugriff auf bestimmte Komponenten des Systems (wie z.B. Dateien) einschränkt oder ganz verbietet.

Diese beiden Punkte sind besonders im Zusammenhang mit Internet und WWW wichtig. Internet und WWW sind somit sicher auch ein wichtiger Grund für die Popularität der verschiedenen Java-Varianten.

Neben den erwähnten Vorteilen hat Java aber auch einen gravierenden Nachteil:

1. **Effizienz**
   Das Interpretieren des Bytecodes ist im Regelfall (deutlich) langsamer als die direkte Ausführung eines vergleichbaren Programmes von dem Prozessor. Auch ist Java nicht für die Ausnutzung aller System-Resourcen (z.B. Speicher) gedacht.

Rechen- oder Speicher-intensive Programme, d.h. Lösungen für numerisch anspruchsvolle Probleme, werden deswegen besser nicht in Java, sondern z.B. in C++ geschrieben. Dies ist auch der Grund, warum wir nicht gleich mit Java, sondern zuerst mit C/C++ angefangen haben.

Zur Illustration der Gemeinsamkeiten und Unterschiede von C/C++ und Java betrachten wir folgende Lösung von A2.2 in Java:

```
public class a2_2 {                          // Die Datei MUSS a2_2.java heissen
 final static double deltaT = 0.0001;   // Diese Zahlen koennen
 final static double g=9.80665;         // vom Programm nicht
 final static double z0=20, v0=0;       // geaendert werden

 public static void main(String[] args) // Das Hauptprogramm MUSS so
  {                                     // deklariert werden
   double v = v0;
   double z = z0;

   System.out.println("t\tz (exakt)\tz (Euler)\tDifferenz");
                                        // Ausgabe einer Kopfzeile

   for(double t=0; t<=2; t += deltaT)   // In Java funktioniert diese
    {                                   // Abbruchbedingung wie gewuenscht
     double zexakt = z0-1/((double) 2)  // Achtung mit Datentyp !
                           *t*t*g;      // Glg. (2.3)
     System.out.println(t + "s\t" + zexakt + "m\t" + z + "\t"
                          + (1000*(z-zexakt)) + "mm");
                                        // Ausgabe der Ergebnisse
     z += v*deltaT;                     // Glg. (2.7)
     v -= g*deltaT;                     // (Euler-Schritt)
```

```
    }
  }
}
```

Java-spezifische Konstruktionen sind:

1. Eine Ausgabe auf die Konsole erfolgt mit

   `System.out.println(Ausgabe);`

   Dies gibt eine *Zeile* aus, erzeugt also automatisch nach der Ausgabe einen Zeilenvorschub (‘`System.out.print(Ausgabe);`’ erzeugt keinen Zeilenvorschub).

2. Verschiedene Ausgabe-Elemente werden mit ‘`+`’ aneinander gehängt.

3. Die explizite Umwandlung eines Datentyps erfolgt mit ‘`(`’Typ‘`)`’ – im Beispiel macht ‘`((double) 2)`’ aus der ganzen Zahl 2 eine Fließkommazahl. Diese Konstruktion unterscheidet sich etwas von der in Anhang B vorgestellten C++-Variante, ist aber mit mit der C-Konvention identisch, kann also auch in C++ verwendet werden.

4. Bei der Deklaration übernimmt ‘`final`’ in etwa die Rolle, die ‘`const`’ in C++ spielt (in Java gibt es letzeres nicht).

Die Bedeutung von ‘`String[] args`’ illustriert folgendes Programm

```
public class argumente {           // Die Datei MUSS argumente.java
                                   // heissen
 public static void
        main(String[] args)        // args ist ein Array von Zeichenketten
  {
   for(int i=0; i<args.length; i++) // args.length enthaelt Anzahl
      System.out.println((i+1) + ". Argument: "
                         + args[i]);  // i-tes Argument ausgeben
  }
}
```

Ein Aufruf nach dem Compilieren z.B. mit

```
java argumente 1 Zwei C
```

sollte selbsterklärend sein.

Dann sind da natürlich noch die Deklarationen '`public`', '`class`' und '`static`', deren Bedeutung in Kapitel 5.3 erklärt wird.

5.2 Arrays in Java

Zur Illustration von Besonderenheiten beim Umgang mit Arrays in Java diene folgendes Fragment des Programmes `kepler.java`[9], das einen Euler-Schritt mit $\Delta t =$ `deltaT` nach (2.7) implementiert:

```
public void euler(double[][] r, double[][] v, double deltaT)
 {
  double[][] be = new double[aktZahl][v[0].length];

  for(int i=0; i<aktZahl; i++)  // Zuerst alle benoetigten
    beschl(akt[i], r, be[i]);   // Beschleunigungen berechnen

  for(int i=0; i<aktZahl; i++)  // Dann die
   for(int n=0; n<v[0].length; n++)
    {
     r[akt[i]][n] += v[akt[i]][n]*deltaT; // Orte und
     v[akt[i]][n] += be[i][n]*deltaT;     // Geschwindigkeiten
    }                           // aktualisieren
 }
```

1. Ein Array wird über

   $$\text{Datentyp}\texttt{[]}\ldots\texttt{[]}\ \text{Name};$$

   deklariert. Die Anzahl der eckigen Klammern gibt die Anzahl der Indizes an.

2. Zur *Erzeugung* eines Arrays wird '`new`' verwendet:

   $$\texttt{new}\ \text{Datentyp}\texttt{[}n_1\texttt{]}\ldots\texttt{[}n_r\texttt{]}$$

[9]Der vollständige Quellcode steht auf der Kurs-Homepage zum Download zur Verfügung.

n_1 bis n_r legen hierbei die Größe fest und müssen ganzzahlige Ausdrücke sein. Deklaration und Erzeugung können zusammengefaßt werden (siehe obiges Beispiel).

3. Die Anzahl der Elemente eines Arrays wird gespeichert und kann mit 'Name`.length`' abgefragt werden. Diese Speicherung der Elementzahl hat zur Folge, daß:

   a. Dimensionen von Arrays bei Prozeduren nicht deklariert werden müssen (und auch nicht dürfen).

   b. Java die Einhaltung der Array-Grenzen zur Laufzeit testen kann und dies auch tut.

Im Übrigen (Indizierung von 0 bis $n-1$, Initialisierung bei Deklaration mit '`{` Werte `}`', ...) verhalten sich Java-Arrays wie die bekannten C-Arrays.

Folgende Hinweise wären noch zu anderen Teilen von `kepler.java` zu ergänzen:

1. Mathematische Funktionen gehören zur Klasse `Math` und werden z.B. mit '`Math.sqrt()`', '`Math.cos()`' und '`Math.sin()`' aufgerufen.

2. Die Klasse `Math` enthält auch die Konstante π als '`Math.PI`'.

5.3 Objektorientierte Programmierung

In C++ kann man objektorientiert programmieren (auch wenn wir dies bisher nicht wirklich getan haben), in Java führt kein Weg daran vorbei. Dies war bereits daran zu erkennen, daß Programme jeweils als Klassen deklariert werden mußten. Ganz offensichtlich wird die Objektorientierung, sobald man Grafik verwendet. Deswegen schieben wir an dieser Stelle eine kurze Diskussion der Objektorientierung ein.

5.3.1 Klassen und Objekte

Eine *Klasse* ist ganz allgemein ein Bauplan für die Erzeugung von konkreten Ausprägungen von Dingen, die allgemeingültig beschrieben werden können. Die Ausprägungen einer Klasse werden als *Objekte* bzw. *Instanzen* bezeichnet. Eine Klasse besteht aus *Attributen* (Variablen) und *Funktionen*, welche den Prozeduren entsprechen.

Hier verwenden wir folgenden allgemeinen Aufbau einer Klasse[10]:

```
[Modifizierer] class Klassenname
    [extends Basisklasse] {
    Attributdeklarationen
    Funktionsdeklarationen
}
```

Die Einheiten in eckigen Klammern sind optional. Die Modifizierer schränken die Verwendung der Klasse ein; die Basisklasse ist die Klasse, von der die neu definierte Klasse bei Vererbung (siehe Kapitel 5.3.3) abgeleitet wird.

Jede Quelldatei sollte nur *eine* Klassendefinition enthalten. Der Name der Datei muß (abgesehen von der Endung '`.java`') mit dem Namen der Klasse identisch sein. Sollten mehrere Klassen in einer Datei definiert sein, so darf nur eine den Modifizierer '`public`' besitzen – diese Klasse bestimmt dann den Dateinamen.

Eine Instanz einer Klasse wird erzeugt, indem eine Variable vom Typ der Klasse deklariert wird (Klassen sind Datentypen !) und anschließend mit Hilfe des Operators '`new`' ein neu angelegtes Objekt dieser Variablen zugewiesen wird:

[10]Zusätzlich zu dem angegebenen Aufbau kann eine Klasse auch Schnittstellen implementieren.

Klasse Variable = `new` Klasse();

Dabei ist 'Klasse();' eine spezielle Funktion (der Konstruktor), mit deren Hilfe Speicherplatz für ein neues Objekt vom Typ 'Klasse' reserviert und initialisiert wird. Die Variable enthält dabei nur einen Zeiger auf den Speicherplatz, in dem das neue Objekt abgelegt wurde.

Auf Attribute oder Funktionen wird mit Hilfe der *Punktnotation* zugegriffen:

Zeiger.Attribut
Zeiger.Funktion()

5.3.2 Funktionen

Operationen auf Objekten werden mit Funktionen ausgeführt. Funktionen können nur innerhalb von Klassen definiert werden. Die Definition von Funktionen ist aber sonst fast mit der aus C/C++ bekannten Struktur identisch.

Bemerkung: Innerhalb von Funktionen kann man Objekte und andere komplexe Datenstrukturen mittles '`new`' erzeugen. Der *Zeiger* auf dieses Objekt kann dann mit '`return`' zurückgegeben werden. Damit kann man z.B. die Kreuzprodukt-Funktion aus Kaptitel 4.3 eleganter implementieren:

```
double[] kreuz(double[] a, double[] b)
 {
  if((a.length != 3) || (b.length != 3))
   {
    System.out.println("Dimensionsfehler in kreuz()");
    double[] r = new double[0]; // Leeren Vektor anlegen
    return r;                   // und zurueckgeben
   }

  double[] r = new double[3];   // Vektor r neu anlegen

  for(int k=0; k<3; k++)
   {                      // Schleife ueber Zielkomponenten
    int i = (k+1) % 3, j = (k+2) % 3;
    r[k] = a[i]*b[j] - b[i]*a[j];
   }

  return r;               // Zeiger zurueckgeben
 }
```

'`this`' ist ein Zeiger, der auf das aktuelle Objekt verweist. Er wird von Funktionen verwendet, um auf Elemente der eigenen Klasse zuzugreifen, die eigene Instanz als Wert zurückzugeben oder sie als Argument beim Aufruf von Funktionen anderer Klassen zu übergeben.

5.3.3 Konstruktoren und Vererbung

Konstruktoren sind spezielle Funktionen, die bei der Erzeugung eines Objektes aufgerufen werden und der Initialisierung dienen. Ein Konstruktor hat den gleichen Namen wie die Klasse und besitzt keinen Rückgabetyp. Der Compiler baut einen Standardkonstruktor ein, sofern kein Konstruktor explizit deklariert ist.

Vererbung ist ein Mechanismus, der es erlaubt, neue Klassen als Erweiterungen bereits vorhandener Klassen zu definieren. Um eine neue Klasse aus einer bestehenden abzuleiten, wird das Schlüsselwort '`extends`' in der `class`-Deklaration verwendet. Die neue Klasse ist eine *Subklasse* der anderen Klasse (*Superklasse*) und übernimmt automatisch alle deren Attribute und Funktionen mit Ausnahme der Konstruktoren. Innerhalb eine Konstruktors der Subklasse kann der Konstruktor der Superklasse mittels '`super()`' als erste Anweisung aufgerufen werden. Fehlt ein solcher Aufruf, setzt der Compiler automatisch ein '`super()`' *ohne Parameter* ein (was natürlich zu Fehlern führen kann).

5.3.4 Modifizierer

Der Modifizierer '`final`' wurde bereits in Kapitel 5.1 erwähnt. Wir benötigen zwei weitere wichtige Modifizierer:

Mit '`static`' können Attribute und Funktionen deklariert werden, deren Nutzung nicht an die Existenz von Objekten gebunden ist. Von als '`static`' deklarierten Variablen existiert während der gesamten Programmlaufzeit nur ein Exemplar (unabhängig von der Anzahl der Instanzen). Als '`static`' deklarierte Funktionen können von außen sowohl über den Klassennamen als auch über einen Instanzennamen aufgerufen werden. Man braucht keine Instanz der Klasse, um sie aufrufen zu können.

Klassen, die mit '`public`' gekennzeichnet sind, können überall benutzt werden. Attribute, die als '`public`' vereinbart werden, können von überall gelesen und geschrieben werden. Funktionen und Konstruktoren, deren Deklaration mit '`public`' beginnt, können von allen Klassen benutzt werden.

Eine beliebige Klasse wird durch Hinzufügen einer Funktion

```
public static void main(String[] args)
```

zu einer Java-Anwendung. `main()` ist als statische Funktion deklariert, da zum Startzeitpunkt noch kein Objekt existiert.

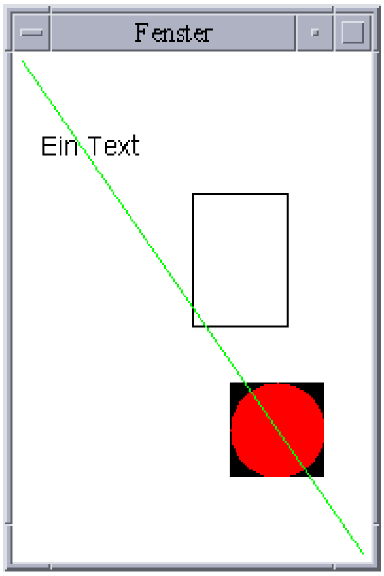
5.4 Grafik in Java

Nun kommen wir endlich zur versprochenen Grafik mit Java. Das folgende Programm `fenster.java` illustriert das Öffnen eines Fensters und einige elementare Zeichen-Funktionen:

```
import java.awt.*;                      // Abstract Window Toolkit (awt)
                                        // einbinden
public class fenster extends Frame {    // Die Datei "fenster.java"
                                        // erweitert die Klasse Frame
public static void main(String[] args)
 {
   fenster frame = new fenster();       // Ein Object "frame" dieser Klasse
 }                                      // erzeugen

public fenster()                        // Der Konstruktor dieser Klasse
 {
   super("Fenster");                    // Ein Objekt der Superklasse "Frame"
                                        // erzeugen
   setSize(200,300);                    // Groesse: 200*300 Pixel
   setLocation(50, 100);                // Position 50,100 auf Bildschirm
   setBackground(Color.white);          // Weissen Hintergrund und
   setForeground(Color.black);          // schwarzen Vordergrund setzen
   setVisible(true);                    // Dieses Fenster auch anzeigen
 }

public void paint(Graphics g)           // Diese Routine wird zum Neuzeichnen
 {                                      // des Fensterinhalts aufgerufen
  g.drawRect(100,100,50,70);            // Rechteck zeichnen
  g.setFont(new Font("SansSerif", Font.PLAIN, 14));  // Schrift waehlen
```

```
    g.drawString("Ein Text", 20, 80);  // Text ausgeben
    g.fillRect(120,200,50,50);         // Ausgefuelltes Rechteck zeichnen
    g.setColor(Color.red);             // Farbe Rot waehlen
    g.fillOval(120,200,50,50);         // Ausgefuellten Kreis zeichnen
    g.setColor(Color.green);           // Farbe Gruen waehlen
    g.drawLine(10,30,190,290);         // Linie von (10,30) nach (190,290)
   }                                   // zeichnen
  }
```

Auf einer SUN (Solaris und CDE) erzeugt dieses Programm ein Fenster, das wie folgt aussieht:



Das Aussehen des Rahmens hängt hier vom Betriebssystem bzw. Windowmanager ab, das Aussehen des Fenster-*Inhalts* wird allerdings von Java systemunabhängig definiert.

Da das Programm keine Funktionalität enthält, um das Fenster ordnungsgemäß zu schließen, muß es vom Benutzer abgebrochen werden, z.B. durch Eingabe von '`Strg-C`' in dem Fenster, aus dem das Programm gestartet wurde.

Weitere Bemerkungen zu `fenster.java`:

1. `import java.awt.*;`

   bindet alle Klassen ('`*`') des Abstract Window Toolkit ein.

2. Ein Fenster ist ein Objekt der Klasse '`Frame`'. Deswegen müssen wir diese Klasse erweitern.

3. Das Hauptprogramm `main` ruft lediglich den Konstruktor unserer neuen Klasse `fenster` auf. Letztere Funktion öffnet dann tatsächlich ein Fenster unter Benutzung folgender Funktionen:

   (a) Der Konstruktor der Klasse `Frame` kann mit einer Zeichenkette ('`String`') als Argument aufgerufen werden, die dann den Titel des Fensters definiert. Im Beispielprogramm handelt es sich um einen Aufruf des Konstruktors der Superklasse. Deswegen kommt '`super()`' zum Einsatz.

   (b) '`setSize(`b`,`h`)`' definiert die Breite b und Höhe h des Fensters.

   (c) '`setLocation(`x`,`y`)`' plaziert die linke obere Ecke des Fensters an Position (x, y) (auf dem Bildschirm).

   (d) '`setVisible(true)`' macht das Fenster sichtbar.
   '`setVisible(false)`' würde es schließen.

   (e) '`setBackground()`' und '`setForeground()`' setzen die Farbe für den Hinter- bzw. Vordergrund des Fensters.
   Die Klasse `Color` enthält einige Konstanten (Attribute) für Farben – das Beispielprogramm verwendet '`white`', '`black`', '`red`' sowie '`green`'.

4. Der Ursprung des Koordinatensystem $(0, 0)$ liegt links oben. Die y-Koordinaten werden nach *unten* positiv gezählt, x-Koordinaten sind wie gewohnt nach rechts positiv.

5. Um etwas in einem Fenster auszugeben, muß eine Funktion `paint` definiert werden. `paint` wird immer dann automatisch aufgerufen, wenn die Grafik aktualisiert werden muß, z.B. weil das Fenster verdeckt war oder, weil das Fenster zum ersten Mal angezeigt wird.
   Der Funktion `paint` wird vom Virtuellen Java-Maschine ein Objekt der Klasse '`Graphics`' übergeben. Wir verwenden hier vor allem die folgenden Zeichen-Funktionen aus dieser Klasse:

   (a) '`setColor()`' setzt die Farbe, die von den folgenden Zeichenoperationen verwendet wird.

   (b) '`drawLine(`x_1`,`y_1`,`x_2`,`y_2`)`' zeichnet eine Linie von (x_1, y_1) nach (x_2, y_2).

(c) '`drawRect(`x`,`y`,`b`,`h`)`' zeichnet ein Rechteck der Breite $b + 1$ und Höhe $h + 1$ und der linken oberen Ecke (x, y).
'`drawOval(`x`,`y`,`b`,`h`)`' zeichnet ein Oval innerhalb des durch die Argumente definierten Rechtecks. Ist $b = h$ (wie in unserem Beispiel), so entsteht ein Kreis.
Ersetzt man den Anfang '`draw`' des Namens dieser Funktionen durch '`fill`', so erhält man ausgefüllte Varianten.

(d) '`Font(`Schriftart`,` Stil`,` Größe`)`' ist der Konstruktor der `Font`-Klasse (Font = Zeichensatz). Mögliche Schriftarten sind z.B. das hier verwendete `"SansSerif"`, `"Serif"`, `"Monospaced"` und `"Dialog"`.
Der Stil ist ein Attribut der Klasse '`Font`' und kann '`PLAIN`' (Normal – wie in diesem Beispiel), '`BOLD`' (Fett) und '`ITALIC`' (Kursiv) sein[11].
Die Größe ist die Schrifthöhe in Pixeln.

(e) '`setFont(`Zeichensatz`)`' wählt den Zeichensatz, der dann bei allen späteren Textausgaben verwenden wird. Das Argument dieser Funktion muß ein `Font`-Objekt sein.

(f) '`drawString(`Text`,`x`,`y`)`' zeichnet den angegebenen Text, wobei (x, y) der linke Rand der Grundlinie ist.

6. **Achtung:** Ein Teil der Zeichenfläche eines Fensters wird von seinem Rahmen verwendet.

Zum Schluß dieses Kapitels stellen wir ein Programm `animation.java` vor, das eine einfache Bewegung darstellt, nämlich einen kleinen Kreis entgegen den Uhrzeigersinn um einen großen Kreis kreisen läßt:

```
import java.awt.*;

public class animation extends Frame {
 static double x=90, y=0;              // Koordinaten des kleinen Kreises

public static void main(String[] args)
 {
   animation frame = new animation();  // Fenster mit Animation erzeugen
 }
```

[11]'`BOLD`' und '`ITALIC`' können mittels Addition '`+`' kombiniert werden.

```
public animation()
 {
   super("Ein Fenster mit Animation");
   setSize(250,250);
   setLocation(50, 100);
   setBackground(Color.white);
   setForeground(Color.black);
   setVisible(true);
                                        // Elegant ist es zwar nicht,
   while(true)                          // aber die Animation kann hierhin
    for(int phi=0; phi<360; phi++)      // Schleife ueber Winkel phi
     {
      x=90*Math.cos(phi*Math.PI/180);   // x- und y- Koordinate aktualisieren
      y=90*Math.sin(phi*Math.PI/180);   // Java kennt Pi !
      repaint((int) (118+x),            // Neu zeichnen -> Animation (!)
              (int) (118-y),            // Nur einen kleinen Bereich neu
              14, 14);                  // zeichnen, sonst flackert's arg
      warte(50);                        // 50 Millisekunden warten
     }
 }

public void paint(Graphics g)
 {
  g.fillOval(100,100,50,50);            // Grossen Kreis in Mitte zeichen
  g.fillOval((int) (120+x),             // und kleinen Kreis darum
             (int) (120-y), 10, 10);
 }

public static void warte(long ms)       // Prozedur zum Warten von
 {                                      // ms Millisekunden
   try                                  // - ohne weitere Erklaerung
    { Thread.sleep(ms); }
   catch(InterruptedException e) {}
 }
}
```

Folgende Elemente dieses Programms sind neu bzw. bedürfen der Erklärung:

1. Die Animation kann in dem Konstruktor laufen[12]. In unserem Beispiel enthält dieser dann eine Endlosschleife, kehrt also nie zurück.

2. '`repaint()`' erzwingt die sofortige Aktualisierung einer Zeichnung. `repaint` ruft die vordefinierte Funktion `update` auf, die die Zeichenfläche löscht und dann ihrerseits die Funktion `paint` aufruft.
   Um Flackern zu reduzieren, kann mit

   $$\texttt{repaint(}x\texttt{,}y\texttt{,}b\texttt{,}h\texttt{)}$$

   das Neuzeichnen auf das über die Argumente definierte Rechteck eingeschränkt werden.
   Probieren Sie ruhig einmal aus, was passiert, wenn Sie im Programm `animation.java` erst die `warte`-Schleife herausnehmen, und in einem zweiten Schritt dann die Argumente von `repaint` löschen !

3. Die letzte Funktion '`warte(`ms`)`' wartet die angegebene Anzahl von Millisekunden. Sie dient zum Steuern der Geschwindigkeit der Animation.
   Hier wird die Funktion '`sleep()`' der `Thread`-Klasse verwendet. Da diese nicht ohne Ausnahmebehandlung verwendet werden darf, kommen hier auch Ausnahmeklassen ('`try`', '`catch`') zum Einsatz. Beides soll hier nicht weiter erklärt werden – verwenden Sie die Funktion `warte()` bei Bedarf einfach in der angegebenen Form !

[12]Eigentlich wäre es besser, diese in einem speziellen `Thread` laufen zu lassen. Dieser Einstieg in den internen Prozeß-Ablauf würde aber hier zu weit führen.

6 Runge-Kutta-Verfahren & Sonnensystem

6.1 Runge-Kutta-Verfahren

Im folgenden werden die in Kapitel 2.1 angegebenen Verfahren etwas genauer diskutiert und erweitert.

6.1.1 Systeme $n$ter Ordnung

Wir betrachten ein System gewöhnlicher Differentialgleichungen $n$ter Ordnung

$$\frac{\mathrm{d}^n u_l}{\mathrm{d}t^n} = f_l\left(t, u_1, \ldots, u_m, \frac{\mathrm{d}\, u_1}{\mathrm{d}t}, \ldots, \frac{\mathrm{d}\, u_m}{\mathrm{d}t}, \ldots \frac{\mathrm{d}^{n-1}\, u_1}{\mathrm{d}t^{n-1}}, \ldots, \frac{\mathrm{d}^{n-1}\, u_m}{\mathrm{d}t^{n-1}}\right) \tag{6.1}$$

für die m Funktionen $u_l(t)$ $(l = 1, \ldots, m)$. Die Bewegungsgleichungen (2.5) und (4.16) sind Beispiele mit $n = 2$.

Eine Differentialgleichungssystem vom Typ (6.1) läßt sich immer auf ein System gewöhnlicher Differentialgleichungen *erster* Ordnung zurückführen. Dazu führt man Variablen

$$z_{k,l}(t) = \frac{\mathrm{d}^k\, u_l(t)}{\mathrm{d}t^{k-1}} \tag{6.2}$$

mit $k = 0, \ldots, n-1$, $k = l, \ldots, m$ ein. Mit diesen Variablen ist das System (6.1) äquivalent zu dem folgenden System 1. Ordnung:

$$\begin{aligned} \frac{\mathrm{d}\, z_{n-1,l}}{\mathrm{d}t} &= f_l\left(t, z_{0,0}, \ldots, z_{0,m}, z_{1,1}, \ldots, z_{1,m}, z_{n-1,1}, \ldots, z_{n-1,m}\right)\,, \\ \frac{\mathrm{d}\, z_{k,l}}{\mathrm{d}t} &= z_{k+1,l}\,, \qquad\qquad \text{für } k = 0, \ldots, n-2\,. \end{aligned} \tag{6.3}$$

Die gesuchten Funktionen erhält man dann nach (6.2) als $u_l(t) = z_{0,l}(t)$. Aufgrund dieser Überlegung kann man sich bei der Konstruktion von numerischen Verfahren auf Gleichungen 1. Ordnung beschränken:

$$\frac{\mathrm{d}\, x_r}{\mathrm{d}t} = f_r\left(t, x_1, \ldots, x_N\right)\,. \tag{6.4}$$

Genaugenommen haben wir dies sowohl beim Euler-Verfahren (2.7) wie auch beim Euler-Richardson-Algorithmus (2.8,2.9) bereits getan, indem wir die Variablen $\vec{v}$ (entsprechend $z_{1,l}$) und $\vec{r}$ (entsprechend $u_l = z_{0,l}$) verwendet haben.

6.1.2 Runge-Kutta-Verfahren zweiter Ordnung

Die einfachste Näherungslösung für eine Gleichung vom Typ (6.4) zu diskreten Zeiten $t_n = t_0 + n\,\Delta t$ ist durch das Euler-Verfahren gegeben

$$x_r(t_n) \quad \to \quad x_{r,n+1} = x_{r,n} + \Delta t\, f_r\left(t_n, x_{1,n}, \ldots, x_{N,n}\right)\,. \tag{6.5}$$

Dies ist offensichtlich eine Approximation, die genau ist bis zur Ordnung Δt. Nach Einsetzen der Variablen-Identifikationen erkennt man tatsächlich schnell die bereits früher angegebene Form (2.7) des Euler-Verfahrens.

Eigentlich gibt es keinen Grund, warum man bei der Berechnung von f_r auf der rechten Seiten von (6.5) ausgerechnet den Anfangspunkt des Intervalls $[t_n, t_{n+1}[$ verwendet. Tatsächlich kann man nicht nur andere Zeiten wählen, sondern durch Linearkombination die Genauigkeit des Verfahrens verbessern. Speziell können wir unter Verwendung von zwei Zeitpunkten den Ansatz machen, daß wir zwei Zeitschritte der Länge $\alpha\,\Delta t$ und $(1-\alpha)\,\Delta t$ nacheinander ausführen und das Ergebnis dann zu einem direkten Zeitschritt über Δt addieren:

$$\begin{aligned}
k_r &= \Delta t\, f_r\left(t_n, x_{1,n}, \ldots, x_{N,n}\right)\,, \\
x_{r,n+1} &= x_{r,n} + \Delta t\,\Big(w_1\, f_r\left(t_n, x_{1,n}, \ldots, x_{N,n}\right) \\
&\qquad + w_2\, f_r\left(t_n + \alpha\,\Delta t, x_{1,n} + \alpha\, k_1, \ldots, x_{N,n} + \alpha\, k_N\right)\Big)\,.
\end{aligned} \tag{6.6}$$

Dieser Ansatz soll nun die Taylor-Entwicklung für $x_r(t)$ bis zur zweiten Ordnung reproduzieren:

$$\begin{aligned}
x_r(t+\Delta t) - x_r(t) &= \Delta t\,\frac{\mathrm{d}\,x_r}{\mathrm{d}t} + \frac{\Delta t^2}{2}\,\frac{\mathrm{d}^2\,x_r}{\mathrm{d}t^2} + \mathcal{O}(\Delta t^3) \\
&= \Delta t\, f_r + \frac{\Delta t^2}{2}\,\frac{\mathrm{d}\,f_r}{\mathrm{d}t} + \mathcal{O}(\Delta t^3)\,.
\end{aligned} \tag{6.7}$$

Hierbei ist

$$\frac{\mathrm{d}\,f_r}{\mathrm{d}t} = \frac{\partial\,f_r}{\partial t} + \sum_s \frac{\partial\,f_r}{\partial x_s}\,\frac{\mathrm{d}\,x_s}{\mathrm{d}t} = \frac{\partial\,f_r}{\partial t} + \sum_s \frac{\partial\,f_r}{\partial x_s}\, f_s\,, \tag{6.8}$$

wobei wir im zweiten Schritt die Differentialgleichung (6.5) eingesetzt haben.

Andererseits lautet die Taylor-Entwicklung von (6.6)

$$\begin{aligned}
x_{r,n+1} - x_{r,n} &= w_1\,\Delta t\, f_r + w_2\,\Delta t\, f_r \\
&\quad + w_2\,\Delta t^2\,\alpha\,\frac{\partial\,f_r}{\partial t} + w_2\,\Delta t^2\,\alpha \sum_s \frac{\partial\,f_r}{\partial x_s}\, f_s \\
&\quad + \mathcal{O}(\Delta t^3)\,,
\end{aligned} \tag{6.9}$$

mit $f_r = f_r\left(t_n, x_{1,n}, \ldots, x_{N,n}\right)$.

Durch Vergleich der Koeffizienten von (6.7) und (6.9) folgt

$$w_1 + w_2 = 1\,, \qquad w_2\,\alpha = \frac{1}{2}\,. \tag{6.10}$$

Diese Bedingungen besitzen eine einparametrige Schar von Lösungen, unter denen keine ausgezeichnet ist. Eine einfache Wahl ist

$$\alpha = \frac{1}{2}\,, \qquad w_2 = 1\,, \qquad w_1 = 0\,. \tag{6.11}$$

Wir erhalten damit das Runge-Kutta-Verfahren 2. Ordnung, das bis zur Ordnung Δt^2 genau ist:

$$\begin{aligned} k_r &= \Delta t\, f_r\left(t_n, x_{1,n}, \ldots, x_{N,n}\right)\,, \\ x_{r,n+1} &= x_{r,n} + \Delta t\, f_r\left(t_n + \frac{\Delta t}{2}, x_{1,n} + \frac{k_1}{2}, \ldots, x_{N,n} + \frac{k_N}{2}\right)\,. \end{aligned} \tag{6.12}$$

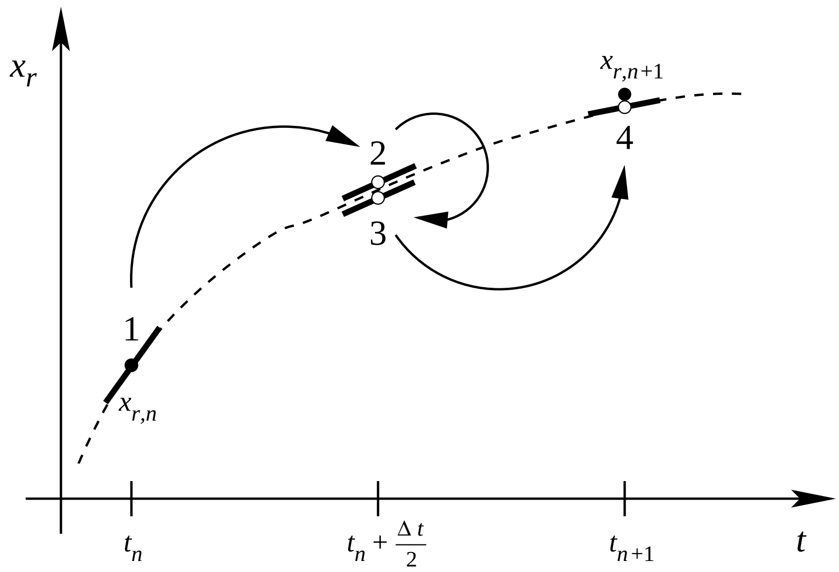
Nach Transformation auf die Variablen $\vec{r}$ und $\vec{v}$ erkennt man den Euler-Richardson-Algorithmus (2.8,2.9) wieder.

6.1.3 Runge-Kutta-Verfahren vierter Ordnung

Unter Verwendung von mehr Zeitschritten kann man ganz ähnlich wie in dem letzten Kapitel auch Runge-Kutta-Verfahren höherer Ordnung konstruieren. Ein häufig verwendetes Verfahren ist das Runge-Kutta-Verfahren vierter Ordnung, das bis zur Ordnung Δt^4 genau ist:

$$\begin{aligned} k_{1,r} &= \Delta t\, f_r\left(t_n, x_{1,n}, \ldots, x_{N,n}\right)\,, \\ k_{2,r} &= \Delta t\, f_r\left(t_n + \frac{\Delta t}{2}, x_{1,n} + \frac{k_{1,1}}{2}, \ldots, x_{N,n} + \frac{k_{1,N}}{2}\right)\,, \\ k_{3,r} &= \Delta t\, f_r\left(t_n + \frac{\Delta t}{2}, x_{1,n} + \frac{k_{2,1}}{2}, \ldots, x_{N,n} + \frac{k_{2,N}}{2}\right)\,, \\ k_{4,r} &= \Delta t\, f_r\left(t_n + \Delta t, x_{1,n} + k_{3,1}, \ldots, x_{N,n} + k_{3,N}\right)\,, \\ x_{r,n+1} &= x_{r,n} + \frac{k_{1,r}}{6} + \frac{k_{2,r}}{3} + \frac{k_{3,r}}{3} + \frac{k_{4,r}}{6}\,. \end{aligned} \tag{6.13}$$

Dieses Verfahren soll hier nicht hergeleitet werden. Stattdessen wollen wir es uns grafisch veranschaulichen:



Das Verfahren (6.13) ist inzwischen als Funktion `rk4()` sowohl in den Programmen `kepler.cc` als auch in `kepler.java` implementiert, die beide von der Kurs-Homepage heruntergeladen werden können.

Denken wir nun z.B. an das System Sonne-Erde-Mond oder alle Planeten in unserem Sonnensystem. In beiden Fällen treten Umlaufdauern auf, die sich um mehr als eine Größenordnung unterscheiden. Vielleicht haben Sie bereits gemerkt, daß Sie hier Δt sehr klein wählen müssen, um eine stabile Lösung zu erhalten. In solchen Fällen lohnt sich meist die Verwendung des Runge-Kutta-Verfahrens vierter Ordnung trotz der größeren Anzahl an Rechenoperationen im Vergleich zum Euler-Richardson-Algorithmus/Runge-Kutta-Verfahren zweiter Ordnung.

6.2 Schrittweitenanpassung und Fehlerkontrolle

Es gibt Probleme, wie z.B. der Flug einer Rakete zum Mond, in der sowohl Phasen mit schnellen Änderungen (Raketenstart, Einschwenken in Mondumlaufbahn, ...) solchen mit einer recht übersichtlichen Zeitentwicklung (z.B. Flugphase der Rakete von der Erde zum Mond) gegenüberstehen. In solchen Fällen ist es sinnvoll, nicht das gesamte Problem mit einer so kleinen Schrittweite Δt zu rechnen, daß zu allen Zeiten die gewünschte Genauigkeit erreicht wird, sondern eine Anpassung der Schrittweite im Verlauf des Verfahrens vorzunehmen.

Am elegantesten ist eine adaptive Steuerung der Schrittweite, die gleichzeitig auch eine Abschätzung des Fehlers erlaubt. Eine Möglichkeit den Fehler zu kontrollieren erhält man, wenn man jeden Schritt doppelt ausführt, und zwar einmal direkt und einmal als zwei halbe Schritte[13]:

$$\begin{aligned} x_{r,n} \quad &\rightarrow \quad x_{r,n+1} = X_1 \,, \\ x_{r,n} \quad &\rightarrow \quad x_r\left(t_n + \frac{\Delta t}{2}\right) \quad \rightarrow \quad x_{r,n+1} = X_2 \,. \end{aligned} \tag{6.14}$$

Für ein Verfahren $m$ter Ordnung gilt nun

$$\begin{aligned} x_r\left(t_n + \Delta t\right) &= X_1 + c\,\Delta t^{m+1} + \mathcal{O}\left(\Delta t^{m+2}\right) \\ &= X_2 + 2\,c\left(\frac{\Delta t}{2}\right)^{m+1} + \mathcal{O}\left(\Delta t^{m+2}\right) \,. \end{aligned} \tag{6.15}$$

Wir haben also

$$X_2 - X_1 \approx c\,\Delta t^{m+1}\left(1 - 2^{-m}\right) \,. \tag{6.16}$$

Wir können nun eine Schrittweite $\tilde{\Delta} t$ so bestimmen, daß der Fehler mit dieser Schrittweite in einem Schritt nach (6.14) einen vorgegebenen Wert ϵ annimmt

$$\left|\tilde{X}_2 - \tilde{X}_1\right| = \epsilon \,. \tag{6.17}$$

Aus (6.16) folgt

$$\begin{aligned} \epsilon &= |c|\;\tilde{\Delta} t^{m+1}\left(1 - 2^{-m}\right) \,, \\ |X_2 - X_1| &= |c|\;\Delta t^{m+1}\left(1 - 2^{-m}\right) \,. \end{aligned} \tag{6.18}$$

[13] Der einfache und doppelte Schritt teilen zumindest den Anfangspunkt. Durch Wiederverwenden solchen Überlapps kann der zusätzliche Rechenaufwand auf weniger als 50% reduziert werden.

Dies kann nach $\tilde{\Delta} t$ aufgelöst werden:

$$\tilde{\Delta} t = \Delta t \sqrt[m+1]{\frac{\epsilon}{|X_2 - X_1|}} \, . \tag{6.19}$$

War der Fehler größer als gewünscht, muß der letzte Schritt mit der Schrittweite (6.19) *wiederholt* werden. Ist der Fehler geringer als gefordert, kann im *nächsten* Schritt die größere Schrittweite (6.19) verwendet werden[14].

Da man c kennt, kann man außerdem eine sogenannte lokale Extrapolation ausführen, die den Fehlerterm der Ordnung $m+1$ korrigiert. Dazu kombiniert man (6.15) so, daß der Term $m+1$ter Ordnung eliminiert wird und verwendet als Schätzwert

$$x_{r,n+1} = X_2 + \frac{X_2 - X_1}{2^m - 1} \, . \tag{6.20}$$

Ob man dies tut oder läßt, ist Geschmackssache. In jedem Fall darf man nicht vergessen, daß man auch bei dieser Korrektur der $m+1$ten Ordnung nur den Fehler der $m$ten Ordnung kontrolliert.

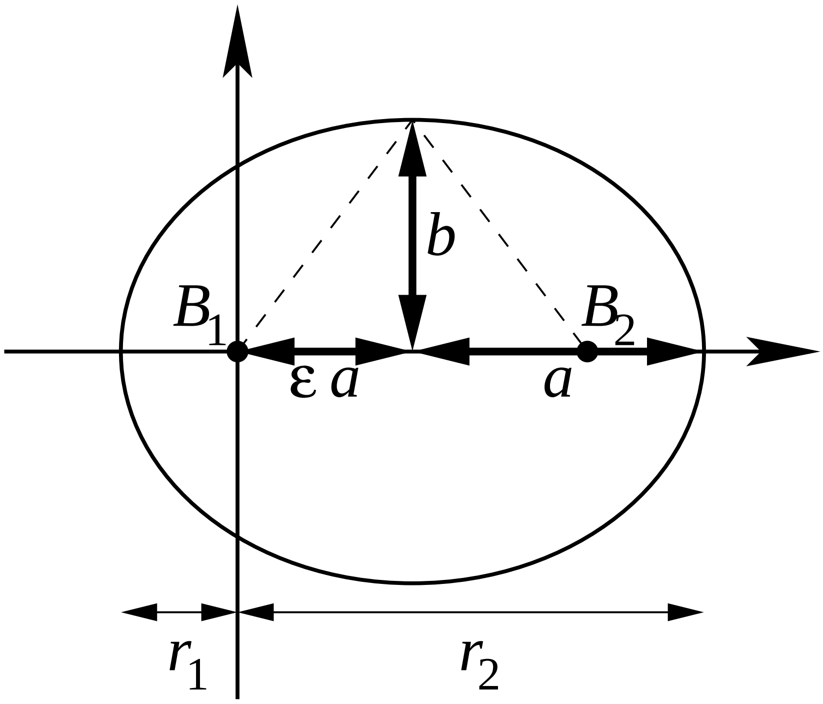
[14] Es empfiehlt sich, die Schrittweite Δt grundsätzlich etwas kleiner zu wählen, als die 'genaue' Formel nahelegt.

6.3 Elliptische Bahnen

Da eine gebundene Bahn in dem Gravitationspotential

$$V(r) = -\frac{G\,M\,m}{r} \tag{6.21}$$

im allgemeinen eine Ellipse ist, und wir im folgenden eine einigermassen realistische Simulation des Sonnensystems durchführen wollen, fassen wir nun kurz elliptische Bahnen zusammen. Dazu betrachten wir zuerst die folgende Skizze:



Hierbei sind B_1 und B_2 die beiden Brennpunkte, a die große Halbachse und b die kleine Halbachse. Eine mögliche Charakterisierung einer Ellipse ist, daß die Summe der Abstände von den beiden Brennpunkten B_1 und B_2 zu einem beliebigen Punkt auf ihr konstant ist. Mit den Bezeichnungen in der Skizze kann man daraus folgern, daß

$$b = a\sqrt{1-\epsilon^2}\,. \tag{6.22}$$

Hierbei ist $\epsilon \geq 0$ die sogenannte Exzentrizität. Für $\epsilon = 0$ findet man $a = b$ und die beiden Brennpunkte fallen zusammen – also eine Kreisbahn. Für eine Ellipse ist $\epsilon < 1$; $\epsilon = 1$ entspricht einer Parabel, $\epsilon > 1$ einer Hyperbel.

Das Kraftzentrum (und damit auch der Koordinatenursprung) befindet sich in dem Brennpunkt, den wir B_1 genannt haben. Die Orte mit den minimalen und maximalen Bahnradien (r_1 bzw. r_2) werden Apsiden genannt. Aus obiger Skizze liest man ab:

$$r_1 = (1 - \epsilon)\, a \qquad r_2 = (1 + \epsilon)\, a\,. \tag{6.23}$$

Dies ist übrigens auch konsistent mit der Definition der großen Halbachse $a = (r_1 + r_2)/2$.

Aus der expliziten Lösung (vgl. die Theoretische Mechanik) der Bewegungsgleichung für das Zentralpotential (6.21) folgt

$$a = \frac{r_1 + r_2}{2} = -\frac{G\, M\, m}{2\, E}\,, \tag{6.24}$$

mit der Gesamtenergie

$$E = \frac{1}{2}\, m\, v^2 + V(r)\,. \tag{6.25}$$

Diese Definition von E kann man nun nach $v(r)$ auflösen. Man findet mit Hilfe von (6.21) und (6.24)

$$v(r) = \sqrt{G\, M\, \left(\frac{2}{r} - \frac{1}{a}\right)}\,. \tag{6.26}$$

Für eine Kreisbahn ist $r = a$. In diesem Fall reduziert sich (6.26) auf die bereits bekannte Beziehung (4.14).

Da man außer dem Betrag der Geschwindigkeit auch ihre Richtung kennen muß, empfiehlt es sich, (6.26) auf Werte von r anzuwenden bei denen $\mathrm{d}\,r/\mathrm{d}t$ verschwindet. Dies gilt insbesondere für die beiden Apsiden r_1 und r_2; in diesen Fällen ist $\vec{v} \perp \vec{r}$.

6.4 Aufgaben

A6.1 *Simulieren Sie das gesamte Planetensystem bestehend aus der Sonne und allen 9 Planeten über mindestens einen Pluto-Umlauf ! Betrachten Sie das Problem als eben (nur zwei Dimensionen), aber wählen Sie die Anfangsbedingung so, daß Sie für jeden Planeten einzeln und bei Vernachlässigung der Sonnen-Mitbewegung eine elliptische Bahn um die Sonne mit der bekannten großen Bahnhalbachse a und Exzentrizität ϵ erhalten würden !*
Fertigen Sie zwei Grafiken an, von denen eine alle Planeten zeigt und die zweite nur die inneren 4 (bis einschließlich Mars) !
Hinweis: *Damit Ihnen das Sonnensystem während der Simulation nicht wegläuft, sollten Sie die Anfangsgeschwindigkeit der Sonne so wählen, daß der Gesamtimpuls des Sonnensystems verschwindet.*

A6.2* *Führen Sie eine volle dreidimensionale Simulation des Sonnensystems durch unter Berücksichtigung der bekannten Neigung der Bahn-Ebenen ! Fertigen Sie mehrere Grafiken an, die die Projektionen der Bahnkurven auf die x-y- und x-z-Ebenen zeigen !*

A6.3* *Berechnen Sie während der Simulation nach A6.1 oder A6.2 den Bahnradius der Sonne $r_S(t) = |\vec{r}_S(t)|$ und schreiben Sie die Zeit t und $r_S(t)$ in eine Datei ! Plotten Sie $r_S(t)$ und interpretieren Sie das Ergebnis !*

6.4.1 Planeten

Planet	**Masse**	**Große Halbachse** a	**Exzentrizität** ϵ	**Bahnneigung**	**Umlaufdauer**
Merkur	$0,055\,m_E$	$0,39\,AU$	$0,206$	$7,0°$	$0,24\,yr$
Venus	$0,815\,m_E$	$0,72\,AU$	$0,007$	$3,4°$	$0,65\,yr$
Erde	m_E ⋆	$1\,AU$	$0,017$	$0°$	$1\,yr$
Mars	$0,107\,m_E$	$1,52\,AU$	$0,093$	$1,85°$	$1,88\,yr$
Jupiter	$318,0\,m_E$	$5,20\,AU$	$0,048$	$1,30°$	$11,86\,yr$
Saturn	$95,2\,m_E$	$9,54\,AU$	$0,056$	$2,49°$	$29,46\,yr$
Uranus	$14,5\,m_E$	$19,18\,AU$	$0,047$	$0,77°$	$84,01\,yr$
Neptun	$17,2\,m_E$	$30,06\,AU$	$0,009$	$1,77°$	$164,79\,yr$
Pluto	$0,0022\,m_E$	$39,44\,AU$	$0,250$	$17,2°$	$247,7\,yr$

⋆ $G\,m_E = 1,19\,10^{-4}\,\frac{AU^3}{yr^2}$

Um Ihnen Tipparbeit zu ersparen, steht der wesentliche Inhalt dieser Tabelle als Datei `planeten` auf der Kurs-Homepage zum Download zur Verfügung.

7 Elektrostatik & Partielle Differentialgleichungen

7.1 Grundgleichungen der Elektrostatik

Die Elektrostatik im Vakuum ist durch die Maxwellgleichungen für das elektrische Feld $\vec{E}$ festgelegt, die in CGS-Einheiten lauten

$$\vec{\nabla} \cdot \vec{E} = 4\,\pi\,\rho\,, \tag{7.1}$$

$$\vec{\nabla} \times \vec{E} = 0\,. \tag{7.2}$$

In einem endlichen Gebiet sind diese Gleichungen natürlich noch durch geeignete Randbedingungen zu ergänzen. Aufgrund von (7.2) existiert ein Potential Φ, so daß

$$\vec{E} = -\vec{\nabla}\,\Phi\,. \tag{7.3}$$

Einsetzen in (7.1) führt auf die Poisson-Gleichung

$$-\Delta\,\Phi = 4\,\pi\,\rho\,, \tag{7.4}$$

mit dem Laplace-Operator

$$\Delta = \vec{\nabla} \cdot \vec{\nabla} = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}\,. \tag{7.5}$$

Im leeren Raum ist $\rho = 0$ und (7.4) reduziert sich auf die Laplace-Gleichung

$$\Delta\,\Phi = 0\,. \tag{7.6}$$

Nun soll beispielhaft eine zweidimensionale Variante ($d = 2$) von (7.6) noch etwas ausführlicher diskutiert werden. Dazu betrachten wir ein Rechteck $x \in [-L_x/2, L_x/2]$, $y \in [-L_y/2, L_y/2]$, auf dessen Kanten wir das Potential Φ vorgeben wollen. In diesem Fall kann der Separationsansatz

$$\Phi(x, y) = X(x)\,Y(y) \tag{7.7}$$

zur Lösung benutzt werden. Mit dem Ansatz (7.7) führt die Laplace-Gleichung (7.6) auf

$$\frac{1}{X(x)}\,\frac{\mathrm{d}^2}{\mathrm{d}x^2} X(x) = -\frac{1}{Y(y)}\,\frac{\mathrm{d}^2}{\mathrm{d}y^2} Y(y) = -k^2\,. \tag{7.8}$$

Da die linke und die rechte Seite Funktionen verschiedener Variablen sind, können diese nur dann gleich sein, wenn sie konstant sind. Diese Konstante haben wir hier bereits auf $-k^2$ gesetzt, wobei zu beachten ist, daß k durchaus auch imaginär sein kann. Nun kann eine Lösung von (7.8) sofort angegeben werden:

$$\begin{aligned} X(x) &= S_x \sin(k\,x) + C_x \cos(k\,x)\,, \\ Y(y) &= S_y \sinh(k\,y) + C_y \cosh(k\,y)\,. \end{aligned} \tag{7.9}$$

Die Lösung für gegebene Randbedingungen erhält man durch eine geeignete Überlagerung von Lösungen vom Typ (7.9). Lauten die Randbedingungen z.B.

$$\Phi\left(\pm\frac{L_x}{2}, y\right) = 0\,, \qquad \Phi\left(x, \pm\frac{L_y}{2}\right) = \Phi_0\,, \tag{7.10}$$

so folgt aus Symmetrie-Gründen $S_x = S_y = 0$. Wegen $\Phi(\pm L_x/2, y) = 0$ gilt nun $\cos(k\,L_x/2) = 0$, also

$$k_n = \frac{(2\,n+1)\,\pi}{L_x} \tag{7.11}$$

mit $n \geq 0$ ganzzahlig. Damit haben wir für die Lösung dieses Randwertproblems

$$\Phi(x,y) = \sum_{n=0}^{\infty} C_n \cos(k_n\,x)\,\cosh(k_n\,y)\,. \tag{7.12}$$

Man kann nun einfach zeigen, daß

$$C_n = (-1)^n \frac{4\,\Phi_0}{(2\,n+1)\,\pi\,\cosh(k_n\,L_y/2)}\,, \tag{7.13}$$

also

$$\begin{aligned} \Phi(x,y) &= \frac{4\,\Phi_0}{\pi} \sum_{n=0}^{\infty} \cos(k_n\,x)\,\frac{(-1)^n\,\cosh(k_n\,y)}{(2\,n+1)\,\cosh(k_n\,L_y/2)} \\ &= \frac{4\,\Phi_0}{L_x} \sum_{n=0}^{\infty} (-1)^n \frac{\cos(k_n\,x)\,\cosh(k_n\,y)}{k_n\,\cosh(k_n\,L_y/2)}\,. \end{aligned} \tag{7.14}$$

7.2 Diskretisierung partieller Differentialgleichungen

Bei (7.4) und (7.6) handelt es sich um partielle Differentialgleichungen, die für eine numerische Behandlung geeignet diskretisiert werden müssen. Eine Möglichkeit, eine partielle Differentialgleichung für eine Funktion $f(\vec{x})$ zu diskretisieren, ist die Betrachtung eines Gitters von Punkten. Eine einfache Wahl ist

$$\vec{x}_{\vec{r}} = \vec{x}_0 + \sum_{i=1}^{d} \Delta x_i \; r_i \; \vec{e}_i \tag{7.15}$$

mit ganzzahligen r_i. Für die Funktionswerte an diesen Punkten schreiben wir

$$f_{\vec{r}} = f\left(\vec{x}_{\vec{r}}\right) \, . \tag{7.16}$$

Wir betrachten nun die partiellen Ableitungen $\frac{\partial^m}{\partial x_i^m} f(\vec{x})$ und tun der Einfachheit halber so, als ob dies die einzige Variable wäre. Für die erste partielle Ableitung an der Stelle $\vec{x}_r$ liegen zwei Definitionen nahe:

$$\begin{aligned} f_{x_i}(\vec{x}_r) = \frac{\partial}{\partial x_i} f(\vec{x}_r) \quad &= \quad \frac{f_r - f_{r-1}}{\Delta x_i} + \mathcal{O}(\Delta x_i) \\ &= \quad \frac{f_{r+1} - f_r}{\Delta x_i} + \mathcal{O}(\Delta x_i) \, . \end{aligned} \tag{7.17}$$

Durch eine geschickte symmetrische Kombination der beiden Möglichkeiten (7.17) kann man die Ordnung verbessern:

$$f_{x_i}(\vec{x}_r) = \frac{f_{r+1} - f_{r-1}}{2\,\Delta x_i} + \mathcal{O}(\Delta x_i^2) \, . \tag{7.18}$$

Analog kann man die zweite partielle Ableitung konstruieren. Eine geschickte Kombination ist

$$f_{x_i x_i}(\vec{x}_r) = \frac{\partial^2}{\partial x_i^2} f(\vec{x}_r) = \frac{f_{r+1} - 2\,f_r + f_{r-1}}{\Delta x_i^2} + \mathcal{O}(\Delta x_i^2) \, . \tag{7.19}$$

Wir müssen nun noch unsere Randbedingungen unterbringen. Bei *Dirichletschen Randbedingungen* ist die Funktion f auf dem Rand des zu untersuchenden Gebiets bekannt. Dem läßt sich leicht Rechnung tragen, indem man für die Ableitungen direkt neben dem Rand in (7.18) oder (7.19) einfach für die entsprechenden $f_{r\pm1}$ die vorgegebenen Werte einsetzt. Bei *Neumannschen Randbedingungen* ist hingegen die Ableitung der Funktion f in Normalenrichtung des Randes auf diesem bekannt. Auch dies läßt sich unterbringen, z.B. indem man $\frac{f_r - f_{r-1}}{\Delta x_i}$ in (7.19) an den entsprechenden Randpunkten durch die vorgegebene Funktion ersetzt.

7.3 Jacobi-Verfahren

Die Diskretisierung (7.19) hat auf einem d-dimensionalen hyperkubischen Gitter eine bemerkenswerte Folgerung für die Laplace-Gleichung $\Delta\,\Phi = 0$:

$$\Phi\,(\vec{x}_{\vec{r}}) = \frac{1}{2\,d}\sum_{i=1}^{d}\left\{\Phi\,(\vec{x}_{\vec{r}} + \Delta x\,\vec{e}_i) + \Phi\,(\vec{x}_{\vec{r}} - \Delta x\,\vec{e}_i)\right\} + \mathcal{O}(\Delta x^4)\,, \qquad (7.20)$$

d.h. das Potential am Platz $\vec{x}_{\vec{r}}$ ist gleich dem Mittelwert über seine Werte an den $2\,d$ Nachbarplätzen !

Diese Beobachtung legt ein einfaches Lösungsverfahren für die Laplace-Gleichung (7.6) nahe, das unter dem Namen Jacobi-Verfahren bekannt ist:

1. Wähle eine beliebige Start-Potential-Konfiguration $\Phi\,(\vec{x}_{\vec{r}})$, die allerdings die geforderten Randbedingungen erfüllt.

2. Berechne

   $$\Phi_{\mathrm{neu}}\,(\vec{x}_{\vec{r}}) = \frac{1}{2\,d}\sum_{i=1}^{d}\left\{\Phi\,(\vec{x}_{\vec{r}} + \Delta x\,\vec{e}_i) + \Phi\,(\vec{x}_{\vec{r}} - \Delta x\,\vec{e}_i)\right\}\,, \qquad (7.21)$$

   bzw. bestimme $\Phi_{\mathrm{neu}}\,(\vec{x}_{\vec{r}})$ für Randplätze $\vec{x}_{\vec{r}}$ aus den Randbedingungen.

3. Berechne

   $$\delta\Phi = \max_{\vec{x}_{\vec{r}}}\left|\Phi_{\mathrm{neu}}\,(\vec{x}_{\vec{r}}) - \Phi\,(\vec{x}_{\vec{r}})\right|\,. \qquad (7.22)$$

4. Ersetze $\Phi\,(\vec{x}_{\vec{r}})$ durch $\Phi_{\mathrm{neu}}\,(\vec{x}_{\vec{r}})$.

5. Fahre bei 2. fort, wenn $\delta\Phi$ eine vorgegebene Schwelle überschreitet. Andernfalls höre auf.

7.4 Lineare Gleichungssysteme

Mit der Diskretisierung (7.19) nimmt z.B. die Poisson-Gleichung (7.4) die Form

$$\sum_{i=1}^{d} \frac{\Phi\left(\vec{x}_{\vec{r}} + \Delta x_i\, \vec{e}_i\right) + \Phi\left(\vec{x}_{\vec{r}} - \Delta x_i\, \vec{e}_i\right) - 2\,\Phi\left(\vec{x}_{\vec{r}}\right)}{\Delta x_i^2} = -4\,\pi\,\rho\left(\vec{x}_{\vec{r}}\right) \qquad (7.23)$$

an. Bringt man alle bekannte Information in (7.23) auf die rechte Seite, so erhält man eine Gleichung der Form

$$A\,\vec{\Phi} = \vec{b}\,. \qquad (7.24)$$

Dies ist ein Spezialfall der allgemeineren Aussage, daß die Diskretisierung einer partiellen Differentialgleichung auf ein lineares Gleichungssystem führt. Diese Beobachtung wollen wir zum Anlaß nehmen, Verfahren zur Lösung linearer Gleichungssysteme zu diskutieren. Normalerweise hat A in (7.24) zusätzliche nützliche Eigenschaften, insbesondere

1. Für vernünftige Differentialgleichungen ist A symmetrisch (bzw. hermitesch)
$$A = A^{\dagger}\,. \qquad (7.25)$$

2. A ist dünn besetzt, d.h. die meisten Matrixelemente a_{rs} von A verschwinden. Im Fall von (7.23) gilt in jeder Spalte s, daß $a_{rs} \neq 0$ nur für höchstens $2\,d + 1$ Werte von r, und zwar unabhängig von der Größe des gewählten Gitters.

3. $-\Delta$ ist positiv definit, d.h.
$$-\int \mathrm{d}^d x\, f(\vec{x})^{\star}\,\Delta\, f(\vec{x}) > 0\,, \qquad (7.26)$$
für geeignete Funktionen $f(\vec{x})$. Gleichung (7.26) läßt sich leicht durch partielle Integration motivieren (zumindest, wenn man sich nicht besonders um die Randterme kümmert):
$$-\int \mathrm{d}^d x\, f(\vec{x})^{\star}\,\Delta\, f(\vec{x}) = \int \mathrm{d}^d x\, \left|\vec{\nabla}\, f(\vec{x})\right|^2 > 0\,. \qquad (7.27)$$

Bevor wir uns nun linearen Gleichungssystemen zuwenden, sei allerdings angemerkt, daß die *direkte* Lösung des Gleichungssystems (7.24) zumindest für die Diskretisierung der Laplace-Gleichung (7.6) auf einem hyperkubischen Gitter keineswegs das effizienteste Verfahren ist, sondern für diesen Fall das Jacobi-Verfahren aus dem letzten Kapitel und dessen Verallgemeinerungen deutlich besser sind.

7.4.1 Gauß-Elimination

Ein lineares Gleichungssystem

$$A\,\vec{x} = \vec{b}\,. \tag{7.28}$$

hat u.a. folgende bekannte Eigenschaften:

1. Das Vertauschen zweier beliebiger *Zeilen* von A und der entsprechenden 'Zeilen' von $\vec{b}$ hat keinen Einfluß auf die Lösung $\vec{x}$. Eine solche Vertauschung entspricht lediglich einer Vertauschung der Reihenfolge beim Aufschreiben der linearen Gleichungen für $\vec{x}$.

2. Die Lösung $\vec{x}$ bleibt genauso unverändert, wenn man eine Zeile von A durch eine Linearkombination ihrer selbst und einer beliebigen anderen Zeile ersetzt, so lange die gleiche Linearkombination in den 'Zeilen' von $\vec{b}$ gebildet wird.

Diese Eigenschaften werden von dem wohl bekanntesten Lösungsverfahren für lineare Gleichungssysteme ausgenutzt, der sogenannten Gauß-Elimination[15].

Sei A eine $n \times n$-Matrix mit Matrixelementen a_{rs}. Dann besteht eine Variante der Gauß-Elimination aus folgenden Schritten:

1. Für alle Spalten $s = 1, \ldots, n$

   (a) (*Pivotisierung*) Suche das Matrixelement mit dem größten $|a_{rs}|$. Vertausche dann die Zeilen r und s von A und $\vec{b}$.

   (b) (*Elimination*) Für alle Zeilen $r = 1, \ldots, n$ mit $r \neq s$ und $a_{rs} \neq 0$ ersetze

$$\begin{aligned} a_{rt} &= a_{rt} - \frac{a_{rs}}{a_{ss}}\, a_{st} \qquad \text{für } t = s, \ldots, n\,, \\ b_r &= b_r - \frac{a_{rs}}{a_{ss}}\, b_s\,. \end{aligned} \tag{7.29}$$

2. Die Lösung von (7.28) ist nun gegeben durch

$$x_r = \frac{b_r}{a_{rr}}\,. \tag{7.30}$$

[15] Löst man ein lineares Gleichungssystem mit Papier und Bleistift, so verwendet man meistens dieses oder zumindest ein ähnliches Verfahren, auch wenn man nicht bewußt den Gauß-Algorithmus anwendet.

Bemerkungen:

1. Das komplette Verfahren benötigt $\mathcal{O}(n^3)$ Operationen.

2. Man kann das Gauß-Verfahren durchaus weiter optimieren. Allerdings ist die obige Variante besonders einfach zu programmieren.

3. Für die Diskretisierung der Poisson-Gleichung (7.23) steht das größte Matrixelement bereits auf der Diagonalen $r = s$. Wir können also in diesem Fall auf die Pivot-Suche verzichten[16].

7.4.2 Conjugate Gradient Verfahren

Die direkte Lösung eines linearen Gleichungssystems (7.28) z.B. mit dem Gauß-Verfahren führt für Matrizen moderater Größe (n einige Tausend) zu folgenden Problemen:

1. Der Speicherbedarf für die Matrix A wächst mit n^2. Zum Beispiel für $n = 10\,000$ braucht man fast 800MByte Speicher um die Matrixelemente als `double`-Zahlen (zu 8 Byte) zu speichern.

2. Für große Matrizen wächst der CPU-Zeit-Bedarf stark (bei der Gauß-Elimination mit n^3).

3. Da die Lösung durch viele aufeinanderfolgende Rechenoperationen bestimmt wird, sammeln sich Rundungsfehler an. Direkte Lösungsverfahren sind deswegen für große Matrizen weder besonders stabil noch genau.

Alle diese Probleme können mit iterativen Verfahren zumindest in speziellen Fällen gleichzeitig gelöst werden. Die endliche Maschinengenauigkeit kann man z.B. so ausnutzen, daß man mit weniger als n^3 Operationen eine Lösung der gewünschten Genauigkeit approximiert. Vermeidet man dabei Transformationen der Matrix, so kann man auch die Speicherung der ganzen Matrix für dünn besetzte Ursprungsmatrizen vermeiden und so den Speicherbedarf erheblich reduzieren.

Das Conjugate Gradient Verfahren (Konjugierte Gradienten-Verfahren) ist ein solches iteratives Verfahren. Es basiert auf dem Gedanken, die Funktion

$$f(\vec{x}) = \frac{1}{2}\,\vec{x} \cdot A\,\vec{x} - \vec{b} \cdot \vec{x} \tag{7.31}$$

[16] Da das Diagonal-Element verschwinden kann, ist die Pivotisierung im allgemeinen zwingend erforderlich, um in (7.29) die Division durch $a_{ss} = 0$ zu vermeiden.

zu minimieren. In einem Extremal- oder Sattelpunkt gilt für symmetrische Matrizen A

$$\vec{0} = \vec{\nabla} f(\vec{x}) = A\,\vec{x} - \vec{b}\,, \tag{7.32}$$

d.h. ein Extremal- oder Sattelpunkt $\vec{x}$ der Funktion (7.31) ist automatisch eine Lösung des linearen Gleichungssystems (7.28).

Beim Conjugate Gradient (CG) Verfahren nimmt man nun an, daß die Matrix A symmetrisch und positiv definit ist (beides gilt für Diskretisierungen des Operators $-\Delta$). Sei $\vec{x}_0$ die Lösung von (7.32), d.h. $A\,\vec{x}_0 - \vec{b} = \vec{0}$. Dann kann man unter Ausnutzung der Symmetrie von A zeigen, daß

$$f(\vec{x}_0 + \vec{x}\,') = \frac{1}{2}\,\vec{x}\,' \cdot A\,\vec{x}\,' + c \tag{7.33}$$

mit

$$c = \frac{1}{2}\,\vec{x}_0 \cdot A\,\vec{x}_0 - \vec{b} \cdot \vec{x}_0\,. \tag{7.34}$$

Da A positiv definit ist, gilt $\vec{x}\,' \cdot A\,\vec{x}\,' > 0$ für $\vec{x}\,' \neq \vec{0}$. Folglich ist $\vec{x}\,' = \vec{0}$ das eindeutige Minimum der Funktion (7.33).

Wir haben somit gezeigt, daß für symmetrische positiv definite Matrizen A die Lösung des Gleichungssystems (7.28) äquivalent ist zur Bestimmung des Minimums der Funktion (7.31).

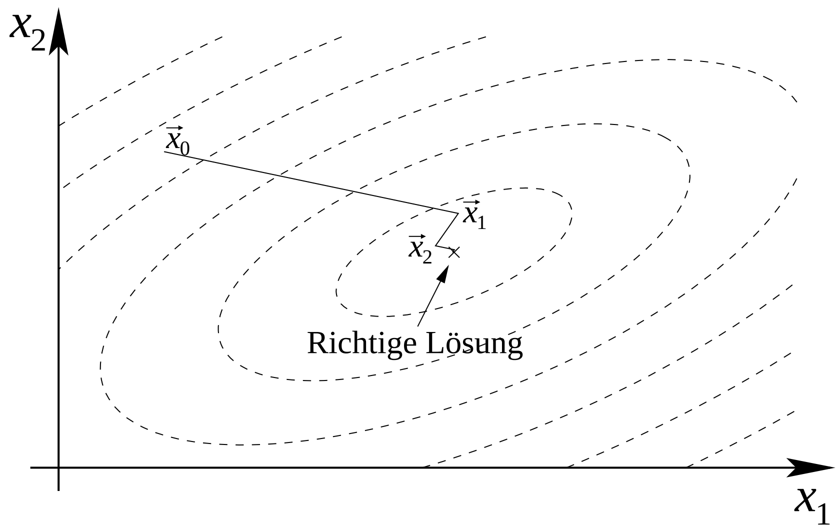
Dieses Minimum kann nun iterativ bestimmt werden. Ein sehr einfaches Verfahren, ist die sogenannte Methode des steilsten Abstiegs, bei der man jeweils die Funktion f entlang ihrer maximalen Steigung nach unten läuft. Die Richtung der maximalen Steigung an einem Punkt $\vec{x}_k$ ist durch den Gradienten

$$\vec{\nabla} f(\vec{x}_k) = A\,\vec{x}_k - \vec{b} = -\vec{r}_k \tag{7.35}$$

gegeben. Man bestimmt nun einen neuen Punkt

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k\,\vec{r}_k \tag{7.36}$$

so, daß $f(\vec{x}_{k+1})$ bzgl. α_k minimal ist. Eine Formel für α_k kann leicht angegeben werden, worauf wir aber hier verzichten wollen. Stattdessen soll das Verfahren des steilsten Abstiegs mit dem folgenden Bild für ein zweidimensionales Beispiel illustriert werden:



Man sieht, daß sich die Vektoren $\vec{x}_k$ allmählich dem Minimum entlang einer Zick-Zack-Bahn annähren, wobei auch bereits in Richtung des Minimums gelaufene Schritte teilweise wieder rückwärts gelaufen werden.

Eine bessere Wahl ist, wenn man nicht einfach immer in Richtung des Gradienten läuft, sondern eine Folge von Vektoren $\vec{p}_k$ wählt, die diese steilste Abstiegsrichtung möglichst gut unter der Nebenbedingung approximieren, daß die Vektoren $\vec{p}_k$ paarweise konjugiert sind. Hierbei bedeutet 'konjugiert', daß die Vektoren bzgl. A paarweise orthogonal sind:

$$\vec{p}_i \cdot A\,\vec{p}_j = 0 \tag{7.37}$$

für $i \neq j$.

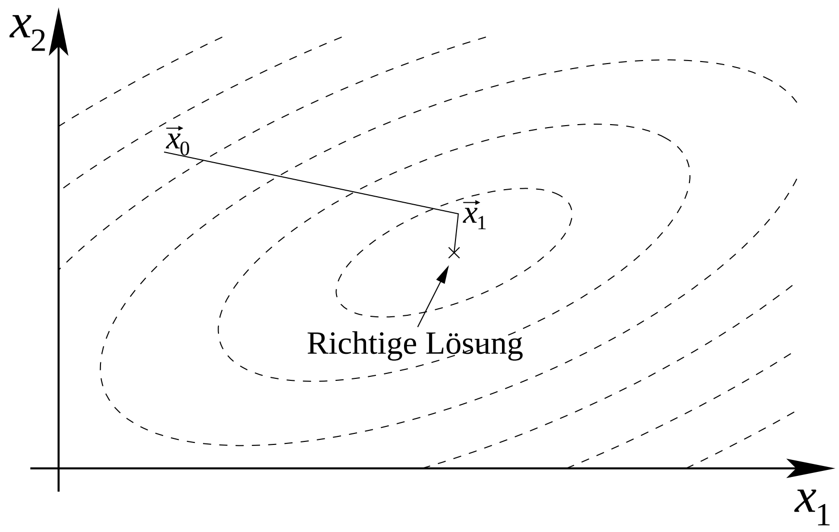
Man kann zeigen, daß die folgende Iterationsvorschrift des CG Verfahrens die gewünschten Eigenschaften hat:

$$\begin{aligned}
\vec{u}_k &= A\,\vec{p}_k\,, \\
\alpha_k &= \frac{\vec{r}_k \cdot \vec{r}_k}{\vec{p}_k \cdot \vec{u}_k}\,, \\
\vec{x}_{k+1} &= \vec{x}_k + \alpha_k\,\vec{p}_k\,, \\
\vec{r}_{k+1} &= \vec{r}_k - \alpha_k\,\vec{u}_k\,, \\
\beta_k &= \frac{\vec{r}_{k+1} \cdot \vec{r}_{k+1}}{\vec{r}_k \cdot \vec{r}_k}\,, \\
\vec{p}_{k+1} &= \vec{r}_{k+1} + \beta_k\,\vec{p}_k\,.
\end{aligned} \tag{7.38}$$

Für einen gegebenen Startvektor $\vec{x}_0$ sind dabei die folgenden Anfangswerte zu wählen:

$$\vec{p}_0 = \vec{r}_0 = \vec{b} - A\,\vec{x}_0\,. \tag{7.39}$$

Aufgrund der Orthogonalitätsrelation (7.37) der $\vec{p}_k$ findet das CG Verfahren theoretisch die exakte Lösung in n Schritten. Zur Veranschaulichung sei das Verfahren an dem obigen zweidimensionalen Beispiel ($n = 2$) mit demselben $\vec{x}_0$ illustriert:



In diesem Fall haben wir die richtige Lösung bereits nach dem 2. Schritt !

Für große Matrizen kommt man sogar meistens mit deutlich weniger als n Schritten aus, um die Lösung mit einer gewünschten Genauigkeit zu approximieren. Man kann zeigen, daß sich das Quadrat der Fehlerfunktion im Schritt $k + 1$ um den folgenden Betrag reduziert:

$$\alpha_k\,\vec{r}_k \cdot \vec{r}_k\,. \tag{7.40}$$

Das CG Verfahren kann somit abgebrochen werden, wenn (7.40) eine vorgegebene Fehlerschwelle unterschreitet.

Bemerkungen:

1. Auch beim CG Verfahren summieren sich die Fehler im Verlauf der Iteration. Dies hat zur Folge, daß die Orthogonalitätsrelation (7.37) umso schlechter erfüllt ist, je größer $|i - j|$ ist. Damit es diese Rundungsfehler vergißt, sollte man das CG Verfahren nach einigen Iterationen j neu

starten, d.h. $\vec{x}_j$ als Startvektor betrachten und mit diesem neue Anfangsbedingungen nach (7.39) berechnen.

2. Die Matrix A wird im CG Verfahren nicht verändert – man braucht lediglich eine Matrix-Vektor Multiplikation. Deswegen kann man spezielle Algorithmen für dünn besetzte Matrizen verwenden, wie sie z.B. bei der Diskretisierung partieller Differentialgleichungen auftreten. Eine Möglichkeit ist, auf die Speicherung der Matrix A zu verzichten und das Produkt $A\,\vec{x}$ für das gegebene Problem als Funktion zu programmieren.

3. Ein gut implementiertes CG Verfahren ist für die Lösung der Diskretisierung einer partiellen Differentialgleichung bei gegebener Genauigkeit mindestens so effizient wie das Jacobi-Verfahren aus Kapitel 7.3.
Das CG Verfahren und seine Verallgemeinerungen können allerdings auch bei zahlreichen anderen Problemen verwendet werden.

7.4.3 Beispiel für Conjugate Gradient Verfahren

Die Poisson-Gleichung

$$-\Delta\,\Phi(x,y) = r(x,y) \tag{7.41}$$

soll für $x \in [-1,1]$, $y \in [-1,1]$ mit Hilfe des CG Verfahrens gelöst werden für

- $r(x,y) = \begin{cases} 1 & \text{für } |x| \le 1/2,\ |y| \le 1/2, \\ 0 & \text{sonst.} \end{cases}$

- $\Phi\,(\pm 1, y) = 0$ (Dirichlet-Randbedingungen in x-Richtung).

- $\frac{\partial\,\Phi(x,\pm 1)}{\partial y} = 0$ (Neumann-Randbedingungen in y-Richtung).

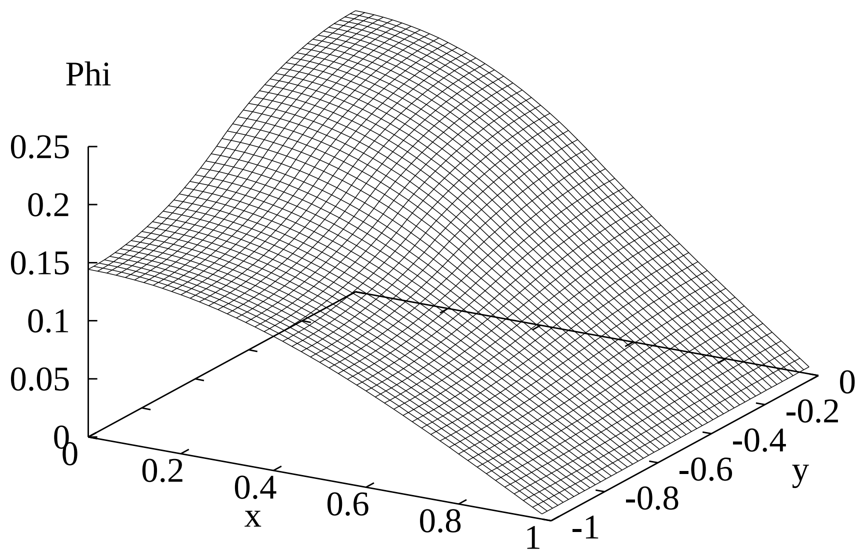
Zur Implementation der Neumannschen Randbedingungen lesen wir aus (7.18) ab, daß $f_{r+1} = f_{r-1}$, wenn r ein Randpunkt ist. Man erhält somit eine Variante von (7.19) am Rand, für die man unter Verwendung von $\frac{\partial}{\partial x_i} f(\vec{x}_r) = 0$ leicht zeigt, daß

$$\frac{\partial^2}{\partial x_i^2} f(\vec{x}_r) = \frac{2\,(f_{r\pm 1} - f_r)}{\Delta x_i^2} + \mathcal{O}(\Delta x_i)\,. \tag{7.42}$$

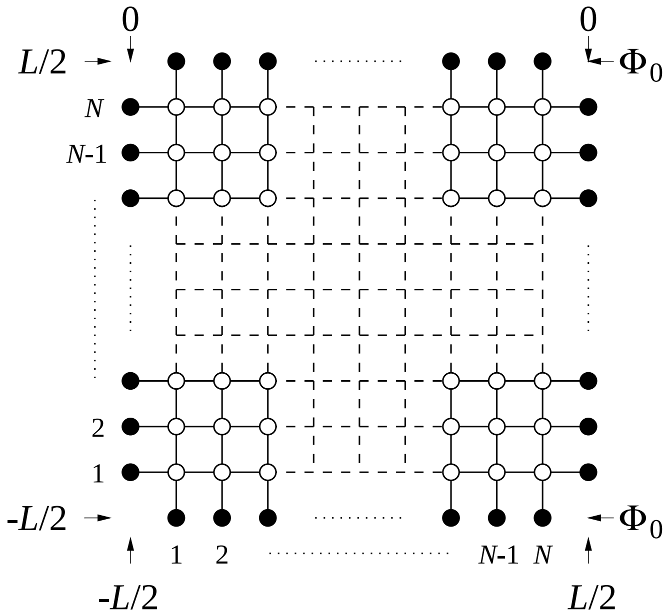
Dieses Problem ist in dem Programm `cg-demo.cc` implementiert, das auf der Kurs-Homepage zum Download zur Verfügung steht. Die Ausgabe-Datei dieses Programms kann z.B. mit `gnuplot` durch Eingabe der folgenden Befehle geplottet werden:

```
gnuplot> set xlabel "x"
gnuplot> set ylabel "y"
gnuplot> set zlabel "Phi"
gnuplot> set xrange [0:1]
gnuplot> set yrange [-1:0]
gnuplot> set ticslevel 0
gnuplot> splot "PhiCGdemo.dat" t "" w l
```

Man erhält damit eine Ausgabe, die ungefähr wie folgt aussieht:



7.5 Aufgaben



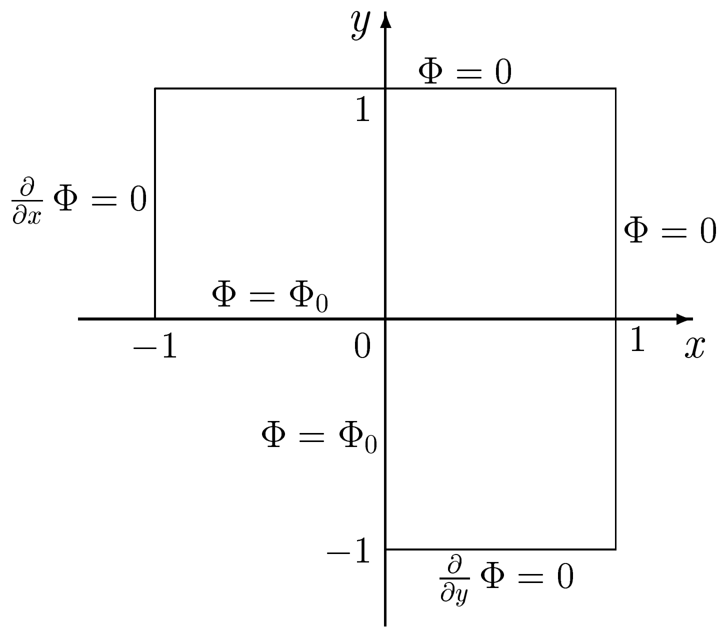
A7.1 *Wir betrachten die Laplace-Gleichung (7.6) auf einem Quadrat der Größe* $L = L_x = L_y = 2$. *Als Randbedingung seien die Dirichletschen Randbedingungen (7.10) gegeben, d.h. der Rand ist bei* $x = \pm L/2 = \pm 1$ *geerdet (*$\Phi = 0$*) und bei* $y = \pm L/2 = \pm 1$ *sei ein Potential* $\Phi(x, \pm 1) = \Phi_0 = 1$ *vorgegeben.*
Berechnen Sie $\Phi(\vec{x}_{\vec{r}})$ *und* $\left|\vec{E}(\vec{x}_{\vec{r}})\right|$ *auf einem* $N+2 \times N+2$*-Gitter, indem Sie die Summanden der exakten Lösung (7.14) mit* $n \leq n_0$ *aufsummieren ! Untersuchen Sie die Fälle* $N = 25$, $n_0 = 100, 500, 2000$ *! Plotten, vergleichen und diskutieren Sie die Ergebnisse für* $\Phi(\vec{x})$ *und* $\left|\vec{E}(\vec{x})\right|$ *!*

A7.2 *Lösen Sie das Randwertproblem aus A7.1 mit dem Jacobi-Verfahren für die* $N \times N$ *inneren Gitterpunkte ! Starten Sie im Inneren mit* $\Phi(\vec{x}_{\vec{r}}) = \Phi_0/2 = 1/2$ *und betrachten Sie die Fälle* $N = 11, 25, 51$ *! Wie oft müssen Sie das Jacobi-Verfahren durchlaufen, um eine Genauigkeit von* $\delta\Phi = 10^{-4}$ *zu erreichen ?*
Berechnen Sie aus Ihrer numerischen Lösung für $\Phi(\vec{x}_{\vec{r}})$ *numerisch* $\left|\vec{E}(\vec{x}_{\vec{r}})\right|$ *unter Verwendung von (7.18) !*
Plotten Sie die Ergebnisse für Φ *und* $\left|\vec{E}\right|$ *und vergleichen Sie sie untereinander sowie mit der exakten Lösung !*

A7.3* *Formulieren Sie das Randwertproblem aus A7.1 als lineares Gleichungssystem für die $n = N^2$ inneren Gitterpunkte und lösen Sie dieses mit Hilfe der Gauß-Elimination ! Betrachten Sie nacheinander die Fälle $N = 11$, 25 und 51 und vergleichen Sie den Rechenaufwand sowie Ihre Ergebnisse für Φ und $\left|\vec{E}\right|$ mit denen aus A7.2 !*

A7.4 *Formulieren Sie genau wie in A7.3 das Randwertproblem aus A7.1 als lineares Gleichungssystem für die $n = N^2$ inneren Gitterpunkte und lösen Sie dieses mit Hilfe des Conjugate Gradient Verfahrens für $A = -\Delta$ und $N = 11$, 25 und 51 ! Nutzen Sie aus, daß die Diskretisierung von $-\Delta$ auf eine dünn besetzte Matrix führt, d.h. vermeiden Sie es, alle Matrixelemente von $A = -\Delta$ zu speichern !*
Vergleichen Sie Ihre Ergebnisse für Φ und $\left|\vec{E}\right|$ mit denen aus A7.2 bzw. A7.3 ! Wie groß ist nun der Rechenaufwand im Vergleich mit dem Jacobi-Verfahren (A7.2) bzw. der Gauß-Elimination (A7.3) ?

A7.5** *Lösen Sie die Laplace-Gleichung $-\Delta\,\Phi(x, y) = 0$ auf dem unten skizzierten L-förmigen Gebiet mit Hilfe des Conjugate Gradient Verfahrens ! Setzen Sie bei den angebebenen Randbedingungen $\Phi_0 = 1$! Berechnen Sie numerisch $\vec{E}$ und plotten Sie die Ergebnisse für Φ und $\left|\vec{E}\right|$!*



Dank

Ich danke dem Rechenzentrum für die Bereitstellung eines Übungsraumes. Ganz besonders danke ich allen Kursteilnehmern für die Begeisterung bei der Teilnahme und insbesondere für Hilfe beim Auffinden von Fehlern in älteren Versionen dieses Skripts.

A Zur Ablage der Übungsaufgaben

Im Verlauf dieses Kurses werden Sie viele kleine Programme schreiben. Deswegen rate ich Ihnen dringend

Halten Sie Ordnung !

Dazu gehören

1. Die Verwendung von Dateinamen, die Ihnen auch nach Wochen, Monaten oder gar Jahren noch etwas sagen. Dabei mag es durchaus von Nutzen sein, daß UNIX Dateinamen praktisch unbeschränkter Länge zuläßt, auch wenn die Faulheit die Verwendung eines kurzen standardisierten Namens nahelegt.

2. Die Verwendung von Unterverzeichnissen für Ihre Projekte, deren Namen ebenfalls bedeutungsvoll sein sollten. Eigentlich gehört jedes Programm in ein eigenes Unterverzeichnis. Bei kürzeren Programmen ist es aber auch zulässig, diese z.B. in einem Verzeichnis pro Kapitel zusammenzufassen. In keinem Fall gehören die Programme in Ihr Hauptverzeichnis.

   Bemerkung: Unter UNIX werden Verzeichnis- und Dateinamen mit '`/`' aneinandergefügt. Ihr Hauptverzeichnis können Sie mit '`~`' erreichen. So bezeichnet

   ```
   ~/programme/kapitel2/first.cc
   ```

   die Datei `first.cc` im Unterverzeichnis `kapitel2` des Verzeichnisses `programme`, das wiederum in Ihrem Hauptverzeichnis liegt.

Im folgenden erwähnen wir kurz einige Kommandos, mit deren Hilfe Sie Ordnung schaffen können (jeweils in eine Konsole –'Shell'– einzutippen):

☞ `cd` Verzeichnis
wählt 'Verzeichnis' als das aktuelle Arbeitsverzeichnis an. Ein leerer 'Verzeichnis'-Text bezeichnet Ihr Hauptverzeichnis, d.h. mit '`cd`' gelangen Sie zurück in Ihr Hauptverzeichnis.

- ☞ `ls` Name
  zeigt alle Dateien mit dem Namen 'Name' an. Ist 'Name' ein Verzeichnis, so wird dessen Inhalt angezeigt. Ein leerer Text 'Name' bezeichnet das aktuelle Verzeichnis, d.h. mit '`ls`' sehen Sie den Inhalt des aktuellen Verzeichnisses.

- ☞ `mkdir` Name
  legt ein Unterverzeichnis des Namens 'Name' im aktuellen Verzeichnis an.

- ☞ `mv -i` Quelle Ziel
  verschiebt eine Datei oder ein Verzeichnis mit Namen 'Quelle'. Sind 'Quelle' und 'Ziel' Dateinamen, so wird die Datei mit Namen 'Quelle' in 'Ziel' umbenannt. Ist 'Ziel' ein Verzeichnis, so wird 'Quelle' *in* das Verzeichnis 'Ziel' verschoben.
  Achtung: Die Option '`-i`' wird empfohlen, damit Sie vor dem Überschreiben bereits existenter Dateinamen gefragt werden und so das unbeabsichtige und unwiderbringliche Löschen von Daten und Programmen vermeiden.

- ☞ `cp -p -i` Quelle Ziel
  kopiert eine Datei oder ein Verzeichnis von 'Quelle' nach 'Ziel'.
  Die Option '`-p`' sorgt dafür, daß Eigenschaften wie z.B. der letzte Änderungszeitpunkt von 'Quelle' mit kopiert werden. Ansonsten ist der `cp`-Befehl dem `mv`-Befehl sehr ähnlich.

- ☞ `rm -i` Datei(en)
  löscht eine (oder mehrere) Datei(en) mit den angegebenen Namen.
  Achtung: Die Option '`-i`' wird dringend empfohlen, damit Sie nicht unbeabsichtigt Daten oder Programme unwiderbringlich löschen.

- ☞ `rm -r -i` Verzeichnis(se)
  muß für das Löschen von Verzeichnissen einschließlich ihres Inhaltes (Dateien und Unterzeichnisse) verwendet werden.

Sogenannte 'Wildcards' sind nützliche Verweise auf mehrere Namen gleichzeitig und werden mit einem '`*`' gebildet.
Beispiel: '`fi*`' bezeichnet sowohl '`first`', '`first.cc`' wie auch '`filename`' (sollten Dateien dieses Namens existieren). Verwenden Sie '`mv`', '`cp`' oder '`rm`'

zusammen mit Wildcards, so lege ich Ihnen die Option '`-i`' besonders ans Herz.

Für weitere Details seien ein UNIX-Handbuch oder die `man`-pages empfohlen.

KDE2 wie auch die meisten anderen grafischen Oberflächen bieten Ihnen Alternativen für diese Ordnungsaufgaben, die aber hier nicht näher erläutert werden sollen.

B Rundungsfehler

Die von C/C++ verwendeten Fließkommazahlen bestehen aus einem Exponenten und einer gewissen Anzahl von Nachkommastellen. Somit werden die meisten Ergebnisse vom Programm automatisch auf die Zahl der verfügbaren Stellen gerundet. Dies führt zu Rundungsfehlern, die sich im Verlauf der Rechnung auch erheblich verstärken können. Betrachten wir z.B. die folgenden Ausdrücke[17]:

```
(2.0 + 5e-16) - 2.0
((2.0 + 5e-16 + 5e-16 + 5e-16) - 1.5e-15) - 2.0
(2.0 + (5e-16 + 5e-16 + 5e-16 - 1.5e-15)) - 2.0
```

Diese Ausdrücke können z.B. zu folgenden Ergebnissen führen[18]:

```
4.44089e-16
-2.22045e-16
0
```

Wir sehen an diesem Beispiel erstens, daß das Fließkomma-Ergebnis von dem exakten abweichen kann. Zweitens sehen wir, daß diese Rundungsfehler auch dazu führen, daß die Fließkomma-Implementation der Grundrechenarten nicht mehr assoziativ ist, sondern es auf die Reihenfolge der Rechenoperationen ankommt !

Verwenden wir ausschließlich Fließkomma-Zahlen vom Datentyp `double`, so müssen wir uns über diese Rundungsfehler normalerweise keine Sorgen machen. Anders sieht es bei Benutzung des weniger genauen Datentyps `float` aus. Die Genauigkeit von `float` im Vergleich zu `double` wird durch folgendes Beispiel illustriert:

```
#include <iostream>

int main()
 {
  double a = 2/double(3);        // Nebenbei lernen wir hier
  float  b = 2/double(3);        // eine alternative Moeglichkeit
                                 // kennen, in C++ die Umwandlung
```

[17] aeb ist die C/C++-Syntax für die Fließkommazahl $a\,10^b$.

[18] Das tatsächliche Ergebnis kann vom Compiler und der Fließkomma-Verarbeitung des Prozessors abhängen !

```
  cout.precision(17);           // von einem Integer- in einen
                                // Fliesskomma-Datentyp zu
  cout << a << "\n";            // erzwingen
  cout << b << "\n";
}
```

Die Ausgabe dieses Beispiels kann die folgende sein (auch hier können Implementationsunterschiede zu kleinen Abweichungen führen):

```
0.66666666666666663
0.66666668653488159
```

Früher war die Rechnung mit `float` deutlich schneller als mit `double`. Da dies auf modernen Rechnern nicht mehr der Fall ist, rate ich dazu, im Regelfall ausschließlich `double` zu verwenden.

Unabhängig von der Genauigkeit führen Rundungsfehler dazu, daß Tests auf Gleichheit bei Fließkommazahlen zu anderen als den erwarteten Ergebnissen führen können, selbst dann, wenn dies anhand einer Bildschirmausgabe nicht erkennbar ist. Der Grund ist, daß kleinste Abweichungen am Rande der numerischen Genauigkeit fast automatisch durch Rundungsfehler enstehen. Der Computer kennt aber nun kein 'fast gleich', so daß er auf 'ungleich' schließt. Am besten ist es, Tests auf exakte Gleichheit bei Fließkommazahlen zu vermeiden.

C `new` und `delete` in C++

Mit Hilfe des Operators '`new`' können in C++ dynamische Datenobjekte angelegt werden, d.h. Objekte, für die Speicher erst zur Programmlaufzeit reserviert wird. Damit ist es auch möglich, mit Objekten variabler Größe zu arbeiten. Die Verwendung von `new` ist in C++ ähnlich wie in Java (vgl. Kapitel 5.1). Im Gegensatz zu Java gibt es in C++ auch einen Operator '`delete`', mit dem man den Speicherplatz freigeben kann, sobald er nicht mehr benötigt wird.

Zur Illustration betrachten wir die folgende Variante des Programmes aus Kapitel 4.3, das das Kreuzprodukt analog zu dem Java-Programm aus Kapitel 5.3.2 eleganter implementiert:

```
#include <iostream>

const int dim=3;          // Nur Dimension 3

double *kreuz(double a[dim], double b[dim])
 {
  double *r = new double[dim];   // Vektor r neu anlegen

  for(int k=0; k<dim; k++)
   {                      // Schleife ueber Zielkomponenten
    int i = (k+1) % dim;
    int j = (k+2) % dim;
    r[k] = a[i]*b[j] - b[i]*a[j];
   }

  return r;               // Zeiger zurueckgeben
 }

int main()
 {
  double e1[dim] = {1, 0, 0};    // 1. Einheitsvektor
  double e2[dim] = {0, 1, 0};    // 2. Einheitsvektor
  double a[dim] = {2, -3, -4};   // a definieren und initialisieren
  double b[dim] = {6, 5, 1};     // b definieren
```

```
  double *r =  kreuz(e1, e2);
  cout << "e1 x e2 = [" << r[0] << ", " << r[1]
                        << ", " << r[2]<<"]\n";
  delete[] r;                    // r wieder freigeben
  r = kreuz(a, b);
  cout << "a x b = [" << r[0] << ", " << r[1]
                      << ", " << r[2]<<"]\n";
  delete[] r;                    // r wieder freigeben
}
```

Erläuterungen:

1. '`new` Datentyp' reserviert ausreichend Speicher für den angegebenen Datentyp. Der Datentyp kann auch ein Vektor sein (siehe oben).

2. `new` gibt einen Zeiger auf den Speicher zurück. Deswegen muß das Ergebnis von `new` an eine Zeiger-Variable zugewiesen werden (erkennbar an dem '`*`' vor dem Variablen-Namen).

3. `new` kann auch in einer Funktion ausgeführt werden. Das Beispiel verwendet es zur Rückgabe eines Vektors. Der Rückgabe-Datentyp dieser Funktion ist dann ein Zeiger.

4. '`delete` Zeiger' gibt den von dem Zeiger belegten Speicher wieder frei. **Achtung:** Zur Freigabe von Vektoren sollte

   `delete[]` Zeiger

   verwendet werden. Andernfalls würde nur ein einziges Element freigegeben; der von dem Vektor belegte Speicher wäre immer noch belegt, ohne jedoch zugänglich zu sein.

D Off-Screen Images in Java

Off-Screen Images sind nützlich, wenn man Flackern ganz vermeiden will, oder aber einmal gezeichnete Elemente auf Dauer in der Ausgabe behalten möchte (Beispiel: Eine Bahnkurve soll insgesamt sichtbar sein). Dazu werden alle oder ein Teil der Zeichenfunktionen in ein Off-Screen Image ausgeführt, d.h. ein Bild, das nicht direkt auf dem Bildschirm erscheint, sondern erst auf Wunsch dorthin kopiert wird. Da die Funktion `update` immer zuerst das Fenster löscht (oder zumindest einen Teil davon), muß sie überschrieben werden. Meist wird man in ihr das Off-Screen Image aktualisieren (z.B. durch Aufruf der Funktion `paint`) und es dann auf den Bildschirm kopieren.

Zur Illustration betrachten wir das folgende Programm `offscreen.java`, einer Variante des Programms `animation.java` aus Kapitel 5.4:

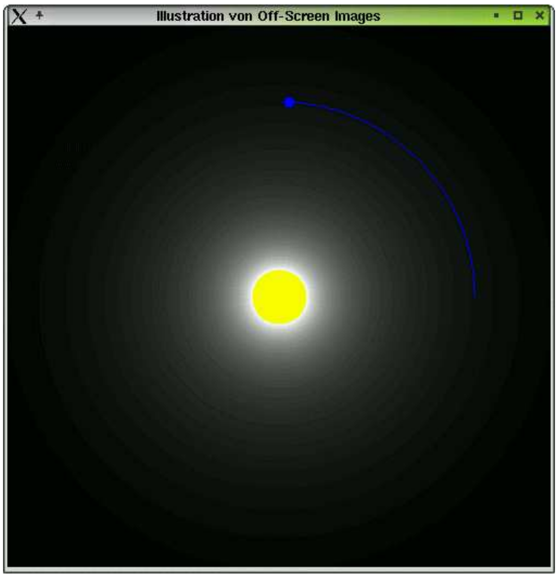
```
import java.awt.*;

public class offscreen extends Frame {
 double x=180, y=0;                    // aktuelle Koordinaten
 double lastx=180, lasty=0;            // und die alten des kleinen Kreises
 Image hintergrund;                    // Hintergrundbild
 Graphics hg;                          // Grafik-Kontext dafuer
 boolean initflag = false;             // Schon initialisiert ?

public static void main(String[] args)
 {
   offscreen frame = new offscreen();  // Fenster mit Animation erzeugen
 }

offscreen()
 {
   super("Illustration von Off-Screen Images");
   setSize(500,500);
   setLocation(50, 100);
   setBackground(Color.white);
   setForeground(Color.black);
   setVisible(true);

   while(true)                         // Endlosschleife fuer Animation
```

```
    for(int phi=0; phi<360; phi++)     // Schleife ueber Winkel phi
     {
      x=180*Math.cos(phi*Math.PI/180); // x- und y-Koordinate
      y=180*Math.sin(phi*Math.PI/180); // aktualisieren
      repaint();                       // Neu zeichnen
      warte(50);                       // 50 Millisekunden warten
     }
 }

public void draw_hintergrund()         // Hintergrundbild zeichnen
 {
   hintergrund = createImage(500,500); // Hintergrundbild erzeugen
   hg = hintergrund.getGraphics();     // Grafik-Kontext dafuer

   for(int r=354; r>=26; r--)          // Bild vom Rand aus mit Kreisen
    {                                  // fuellen
     // Graustufen proportional zu 1/r berechnen
     int c = (int) (7155*(1/((double) r) - 1.0/354.0));
     hg.setColor(new Color(c, c, c));  // Entsprechendes Grau erzeugen
     hg.fillOval(250-r, 250-r, 2*r, 2*r); // und Kreis zeichnen
    }

   hg.setColor(Color.yellow);          // Farbe gelb
   hg.fillOval(225,225,50,50);         // Grossen Kreis in Mitte zeichen

   initflag = true;                    // Flag fuer Initialisierung setzen
 }

public void paint(Graphics g)          // Direkt ins Fenster zeichnen
 {
  if(! initflag)                       // Hintergrund erzeugen, falls
     draw_hintergrund();               // noch nicht geschehen

  g.setColor(Color.blue);              // Mit der Farbe blau
  g.fillOval((int) (245+x),            // den kleinen Kreis zeichnen
             (int) (245-y), 10, 10);
```

```
    // Weg des kleinen Kreises im Hintergrund einzeichnen
    hg.setColor(Color.blue);              // Farbe blau
    // Zurueckgelegten Weg zeichnen
    hg.drawLine((int) (250+lastx), (int) (250-lasty),
                (int) (250+x), (int) (250-y));
    lastx = x;                            // x und y in lastx
    lasty = y;                            // und lasty speichern
   }

  public void update(Graphics g)          // Fenster-Inhalt aktualisieren
   {
    if(! initflag)                        // Hintergrund erzeugen, falls
       draw_hintergrund();                // noch nicht geschehen

    // Hintergrundbild ins aktuelle Fenster kopieren
    g.drawImage(hintergrund, 0, 0, this);
    // und den Rest der Zeichenfunktionen ausfuehren
    paint(g);
   }

  public static void warte(long ms)       // Warte ms Millisekunden
   {
     try { Thread.sleep(ms); }
     catch(InterruptedException e) {}
   }
  }
```

Dieses Programm erzeugt zuerst ein Hintergrundbild, das um einen gelben Kreis in der Mitte radial Graustufen enthält, die proportional zu $1/r$ von weiß (direkt am gelben Kreis) bis zu schwarz am Rand des Fensters abfallen. Dann läßt `offscreen.java` einen kleineren blauen Kreis um den inneren gelben kreisen. Dabei wird die Bahnkurve mit blauen Linien in das Hintergrundbild gezeichnet, die blaue Kreisscheibe hingegen direkt in das Fenster, nachdem der Hintergrund dorthin kopiert worden ist.

Nach einer gewissen Zeit sieht das Bild wie folgt aus (der Rahmen des Fensters wurde in diesem Fall von KDE2 unter Linux erzeugt):



Folgende Elemente des Programmes `offscreen.java` sind noch zu erklären:

1. '`createImage(`b`,`h`)`' erzeugt ein Off-Screen Image (Datentyp '`Image`') der Breite b und Höhe h.

2. '`getGraphics()`' liefert den Graffikkontext (Datentyp '`Graphics`') eines `Image`-Objektes. Mit diesem Grafikkontext können u.a. die in Kapitel 5.4 beschriebenen Zeichenfunktionen in der dort angegebenen Weise ausgeführt werden.

3. '`drawImage(`i`,`x`,`y`,`o`)` zeichnet ein `Image` i mit der linken oberen Ecke an der Position (x, y) in das `Graphics`-Objekt. o hat den Typ `ImageObserver` – meistens kann hier einfach '`this`' eingesetzt werden.

4. '`update(`g`)`' ist eine vordefinierte Funktion, die immer zum Neuzeichnen des Fensterinhalts aufgerufen wird (z.B. durch Aufruf von `repaint`). Hier wird die Funktion `update` überschrieben, um den Ablauf des Fenster-Neuzeichnens umzudefinieren. g ist ein `Graphics`-Objekt.

5. '`Color(`r`,`g`,`b`)`' ist der Konstruktor der Klasse `Color`. r, g und b sind die Rot-, Grün-, bzw. Blau-Anteile der neuen Farbe. Die ganzen Zahlen r, g und b laufen von 0 (dunkel) bis 255 (maximale Intensität). $r = g = b = 255$ liefert Weiß und $r = g = b = 0$ führt zu Schwarz.

6. Der Java-Datentyp '`boolean`' entspricht dem C++-Datentyp '`bool`'.

Literatur

[1] D. Abts, *Grundkurs Java*, Vieweg, Braunschweig (1999).

[2] U. Breymann, *C++, Eine Einführung*, Hanser, München (1999).

[3] D.N. Capper, *Introducing C++ for Scientists, Engineers and Mathematicians*, Springer, London (2001).

[4] H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods*, Addison-Wesley, Reading, Massachusetts (1996).

[5] A. Jennings, *Matrix Computation for Engineers and Scientists*, John Wiley & Sons, Chichester (1977).

[6] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C*, Cambridge University Press (1992).

[7] RRZN, *Die Programmiersprache C. Ein Nachschlagewerk*, 12. Auflage (2001);
RRZN, *C++ für C-Programmierer*, 11. Auflage (2000).

[8] B. Stroustrup, *Die C++-Programmiersprache*, Addison-Wesley, Bonn (1998).

[9] C. Ullenboom, *Java ist auch eine Insel*, Galileo Computing, Bonn (2002)
[`http://www.galileocomputing.de/openbook/javainsel/`].

[10] U. Wolff, B. Bunk, M. Hasenbusch, *Computational Physics I*, Skriptum HU Berlin WS 2001/02
[`http://linde.physik.hu-berlin.de/comphys/comphys.html`].