Rechentechniken: Teilchenkollisions-Simulation über Runge-Kutta

Beispielhafte Verwendung des Programmes rungekut.cpp

Julian Bergmann
julian.bergmann@physik.uni-giessen.de

17. Oktober 2013

Inhaltsverzeichnis:

1 Überblick

Aufgabe dieses Programmes war es die Wechselwirkung mehrerer Teilchen untereinander, speziell die Kollision zweier Teilchenpakete, zu simulieren.

Dazu wird sich des Runge–Kutta–Algorithmus 4. Ordnung bedient, um die sich aus der Kräftewechselwirkungen ergebenen Differentialgleichungen zu lösen.

Im Programm wird zunächst ein Teilchenblock erstellt und ohne äußere Wechselwirkung ruhen gelassen, bis sich ein Grundzustand ausgebildet hat. Dieser kann daraufhin mit einem Startimpuls auf einen weiteren Block geschossen werden. Die einzelnen Teilchenpositionen können anschließend in gnuplot betrachtet oder als animierte .png-Datei exportiert werden.

Zusätzlich sollte hier noch die Betrachtung stabiler Cluster in Abhängigkeit von Startimpuls und Stoßparameter Beachtung finden.

Es folgt zunächst eine Vorstellung des Runge–Kutta–Algorithmus 4. Ordnung. Danach wird eine Beispiel–Einstellung vorgestellt und deren Auswertung bezüglich stabiler Cluster vorgenommen. Zuletzt folgt der kommentierte Quellcode, zugunsten der Lesbarkeit eingerückt und -gefärbt.

2 Runge–Kutta–Algorithmus 4. Ordnung

Um einfache Differentialgleichungen numerisch lösen zu können, wird häufig der Runge–Kutta–Algorithmus 4. Ordnung benutzt. Dieser basiert auf den Ableitungen der Variablen am Startpunkt, 2 mal auf halber Schrittweite $\frac{h}{2}$ und einmal an der vollen Schrittweite h (siehe hierzu Abbildung 1).

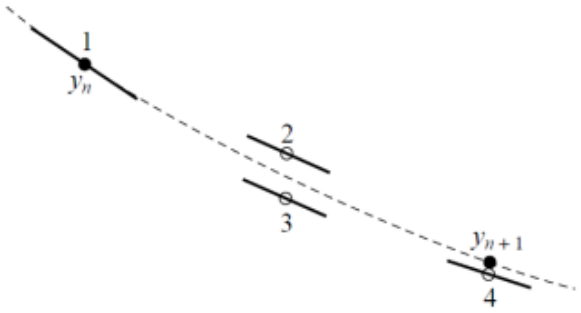

Abbildung 1: Aus Numerical Recipes: Veranschaulichung Runge Kutta 4. Ordnung.

Durch geschickte Addition bleibt lediglich ein Term 4. Ordnung, was zu dem Namen des Algorithmus führte, wodurch auch der Fehler nun lediglich der Größenordnung von h^5

entspricht. Der Algorithmus sieht dabei wie folgt aus:

$$\begin{aligned} k_1 &= h \cdot f(x_n, y_n) \\ k_2 &= h \cdot f(x_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2}) \\ k_3 &= h \cdot f(x_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2}) \\ k_4 &= h \cdot f(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5) \end{aligned} \tag{2.1}$$

Hierbei steht $y(x)$ für die Variable und $f(x_n, y_n)$ für die Ableitung von $y(x)$ nach x.

Der entsprechende Code–Teil, *rk4* Zeile 262f, wurde aus den Numerical Recipes entliehen.

3 Anwendung

3.1 Potential

Wie bereits in der Einführung geschrieben, soll das Programm nun anhand eines bekannten Potentials ein Teilchen–Gitter erzeugen und in den Ruhezustand fallen lassen.

Hierzu wurde als Potential eine Kombination aus 2 Gaußglocken–Kurven benutzt (siehe Abbildung 2):

$$V(r) = a \cdot \exp(-\frac{1}{2}(\frac{r}{\sigma_1})^2) - b \cdot \exp(-\frac{1}{2}(\frac{r}{\sigma_2})^2) \tag{3.2}$$

Die Parameter der Höhen (a und b) bzw. der Breiten der Glockenkurven wurden unter Annahme der Verwendung von Silizium–Atomen (28 u), entsprechend einer Bindungsenergie zwischen 0.5 und 1 eV und einem Potentialminimum von ca. 1 Å, ermittelt:

$$a = 5\ eV, \quad b = 0.27\ eV, \quad \sigma_1 = 1.3\ \text{Å}, \quad \sigma_2 = 3\ \text{Å} \tag{3.3}$$

Damit lässt sich nun die Kraft,die die Teilchen aufeinander auswirken, bestimmen, wobei r für den Teilchenabstand $||\vec{r}_1 - \vec{r}_2||_2$ steht:

$$F(\vec{r}_1, \vec{r}_2) = -\vec{\nabla}V(r) = \left(a \cdot \exp(\frac{1}{2}(\frac{r}{\sigma_1})^2) \cdot \frac{1}{\sigma_1^2} - b \cdot \exp(\frac{1}{2}(\frac{r}{\sigma_2})^2) \cdot \frac{1}{\sigma_2^2}\right) \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{pmatrix} \tag{3.4}$$

Diese Kraft fließt nun in die Newton'sche Bewegungsgleichung mit ein:

$$m\frac{d^2}{dt^2}x = F(\vec{r}_1, \vec{r}_2) \tag{3.5}$$

Da der Runge–Kutta–Algorithmus 4. Ordnung allerdings nur auf DGL erster Ordnung angewendet werden kann, wandeln wir diese DGL 2. Ordnung in 2 DGL 1. Ordnung um:

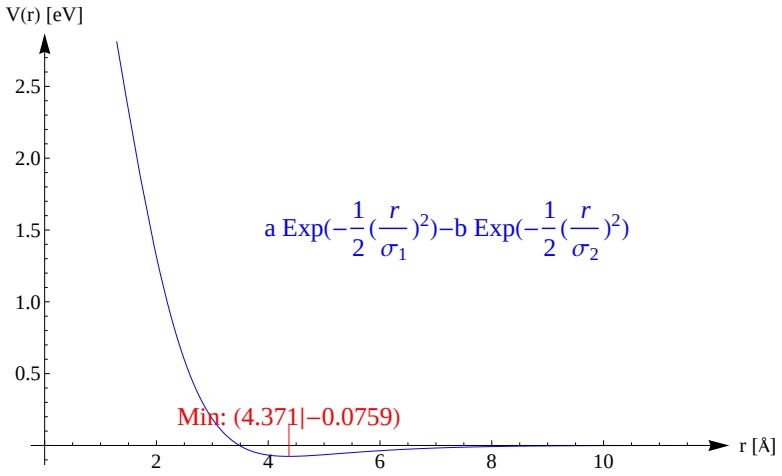

Abbildung 2: Potential entsprechend Gleichung 3.2 mit Parametern 3.3. Das Potential der Form zweier Gaußglocken hat das Minimum bei 4.371 Å

$$\begin{aligned} \frac{d}{dt}x &= \frac{p}{m} \\ \frac{d}{dt}p &= F \end{aligned} \tag{3.6}$$

Diese Potential-/Kraft–Angaben lassen sich im Programmteil *myderiv* Zeile 373 definieren. Zeile 397 beinhaltet dabei den impulsabhängigen Reibungsterm, welcher zur Ermittlung des Grundzustandes benötigt wurde.

3.2 Aufstellung

Innerhalb dieser Anwendung wurden 2 Blöcke mit je $4^3 = 64$ Teilchen gewählt, welche aufeinander geschossen werden sollten. Die Blöcke wurden hierbei mit einem Gitterabstand von je 4.3 Å verteilt.

Nun wurde einer dieser Blöcke mit 4^3 Teilchen aufgestellt (Zeile 430f) und mit einem impulsabhängigen Reibungsterm versehen und gewartet, bis dieser in den Grundzustand fiel. Dabei wurde der Runge–Kutta–Algorithmus 4. Ordnung über einen Zeitraum von 0.5 ns mit einer Schrittweite von 10 fs (=50000 Schritte) laufen gelassen. Der Reibungsterm–Parameter *Gamma* wurde dabei auf 10 ps^{-1} gesetzt.

Nachdem der Grundzustand erreicht wurde, kann dieser in "'endzustand.temp"' abgespeichert werden (Zeile 515f). Die Datei wurde daraufhin in "'grundzustand2.temp"' umbenannt, um wieder eingelesen werden zu können (Zeile 451f).

Um die Durchläufe hier unterbrechen zu können, falls der gewählte Endzeitpunkt noch nicht ausreicht, wurde der Block Zeile 463f geschrieben, der aus "'endzustand.temp"' alle Daten ausließt und als neue Daten auffasst.

Zu Testzwecken können auch direkt 2 Blöcke generiert und aufeinander geschossen werden (Zeile 475f).

Hat man jedoch den Grundzustand ermittelt, wird nun für diesen Block eine Impulsverteilung über den Monte–Carlo–Algorithmus in Abhängigkeit einer Temperatur berechnet (Zeile 490 ⇒ *montecarlo*:231f). Mittels der Maxwell–Boltzmann–Verteilung

$$P(p) = 4\pi(2\pi k_b T m)^{-\frac{3}{2}} p^2 \exp(-\frac{p^2}{2mk_bT}) \tag{3.7}$$

werden nun zufällig ermittelte Werte für θ, φ, Impulsbetragswert und Funktionswert überprüft, bis sie dieser entsprechen. Dabei werden bis zu $2 \cdot 10^6$ Versuche durchgeführt.

In der nächsten Code–Zeile (Zeile 491 ⇒ *Drehimpuls*:55f) wird in das Schwerpunktsystem transformiert, wodurch das Positionieren der Teilchenblöcke später erleichtert wird, und mit dem Drehimpuls die Winkelgeschwindigkeit bestimmt.

Der Schwerpunktimpuls folgt dabei

$$\vec{P}_s = \frac{\sum_i m_i \vec{p}_i}{\sum_i m_i} \tag{3.8}$$

worauf eine Galilei–Transformation in das Schwerpunktsystem folgt. Über den Schwerpunkt lassen sich nun die Teilchenvektoren y und der Grundzustandsvektor r_0 in dessen Relativkoordinatensystem transformieren und der Drehimpuls sowie der Trägheitstensor kann bestimmt werden. Es folgt eine Gauß'sche Elimination zur Ermittlung der Winkelgeschwindigkeit, mithilfe derer das Koordinatensystem entsprechend gedreht werden kann, dass diese in z–Richtung liegt. Dies macht den Vergleich mit dem Grundzustand einfacher, da dieser nun pro Durchlauf lediglich um $|\vec{\omega}|$ gedreht werden muss.

Das Resultierende Teilchenpacket wird in Zeile 494f mit Versatz 20 Å in y-Richtung eingefügt und erhält einen Startimpuls von 0.5 $\frac{\text{ÅeV}}{\text{ns}}$, ebenfalls in y–Richtung. der Block wird daraufhin an der x–z–Ebene ein zweites mal gespiegelt eingefügt. Zusätzlich wird der 1. Block im angefügten Beispiel um 15 Å in x–Richtung verschoben, was als Stoßparameter interpretiert werden kann.

Im Programm folgt der Aufruf des Schnittstelle zum Runge–Kutta–Algorithmus (*rkdumb*, Zeile 512), welche diesen ordnungsgemäß Iterativ entsprechend der Schrittanzahl aufruft und nach jedem Schritt u.a. Energieerhaltung überprüft (Zeile 347f). Diese sollte nach Zeile 362 im Rahmen von 10% Schwankung liegen, andernfalls wird ein Fehler ausgegeben, was durch eine zu groß gewählte Schrittweite verursacht werden kann.

Für jeden Schritt wird dabei der Teilchenvektor (x, y, z, p_x, p_y, p_z) in einem 2–Dimensionalen Array $(\vec{p}, i)$ gespeichert, welche im Anschluss ausgegeben werden kann.
Hierzu wird beispielsweise in Zeile 515f der Teilchenvektor des letzten Schrittes in die Datei "'endzustand.temp"' geschrieben.

Zeile 522f hingegen schreibt den Anfangszustand, also den Teilchenvektor im 0. Schritt in die Datei "'anfangszustand.temp"'

Der Teil Zeile 535f erzeugt Teilchentrajektorien in Gnuplot, sollte allerdings nur bei geringer Teilchen– und Schrittzahl benutzt werden, da Gnuplot nicht mit großen Datenpunktmengen umgehen kann.

Will man das Ergebnis im Gnuplot–Viewer betrachten, sollte man Zeile 551 aus– und Zeile 556 einkommentieren. Der folgende Codeblock erzeugt dann eine gnuplot–Datei, "'bewegung.temp"', welche mit "'wgnuplot bewegung.temp"' aufgerufen werden kann. Hierzu sollte auch Zeile 556, Zeile 570 und Zeile 576 ein– und Zeile 579 auskommentiert werden.

Ändert man in diesem Teil nichts, wird hingegen eine animierte .gif–Datei der Teilchenposition erzeugt, wobei in Zeile 558 geregelt ist, dass nur jedes 500. Bild gespeichert wird.

3.3 Auswertung

Da sinnvolle Algorithmen für die Detektion von Clustern sehr aufwändig sind, wurde sich hier darauf beschränkt, in den grafischen Ausgaben nach stabilen Clustern, also min. 4 Teilchen, die über viele Iterationsschritte nahe beieinander bleiben) zu suchen und zu zählen.

Zunächst wurde hierfür ein frontaler Stoß, also Stoßparameter b=0, mit verschiedenen Anfangsimpulsen ca. 1 ns lang betrachtet (Abbildung 3). Hierbei wird klar, dass die Teil-

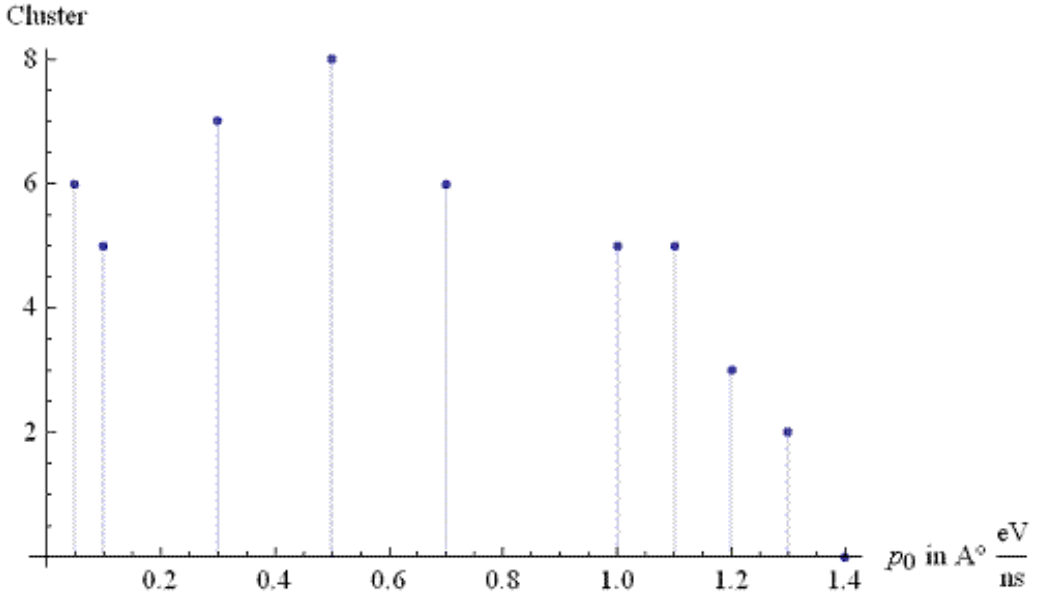

Abbildung 3: temporär stabile Cluster bei verschiedenen Anfangsimpulsen nach etwa 1 ns. Die Impulse beziehen sich hierbei jeweils auf eines der Teilchen. im 2. Teilchenpaket wurden die Vorzeichen in y–Richtung vertauscht.

chenblöcke bei höheren Impulsen aneinander "vorbei" fliegen, also weniger miteinander wechselwirken.

Außerdem wurde der Stoßparameter (Abbidung 4) bei p=0.5 $\frac{\text{ÅeV}}{\text{ns}}$ untersucht. Hierbei wurde ein ähnliches Bild wie beim Start–Impuls p_0 beobachtet, allerdings steigt die Teilchenzahl bei einem Versatz von 15 Angström wieder, da die Teilchen nur peripher kollidieren und somit 2 große Teilchenhaufen mit stabilen Clustern entstehen. Betrachtet man hingegen den Wert bei b=10 A, so sind nur noch 2 kleine Cluster sichtbar, die restlichen Teilchen werden einzeln verstreut.

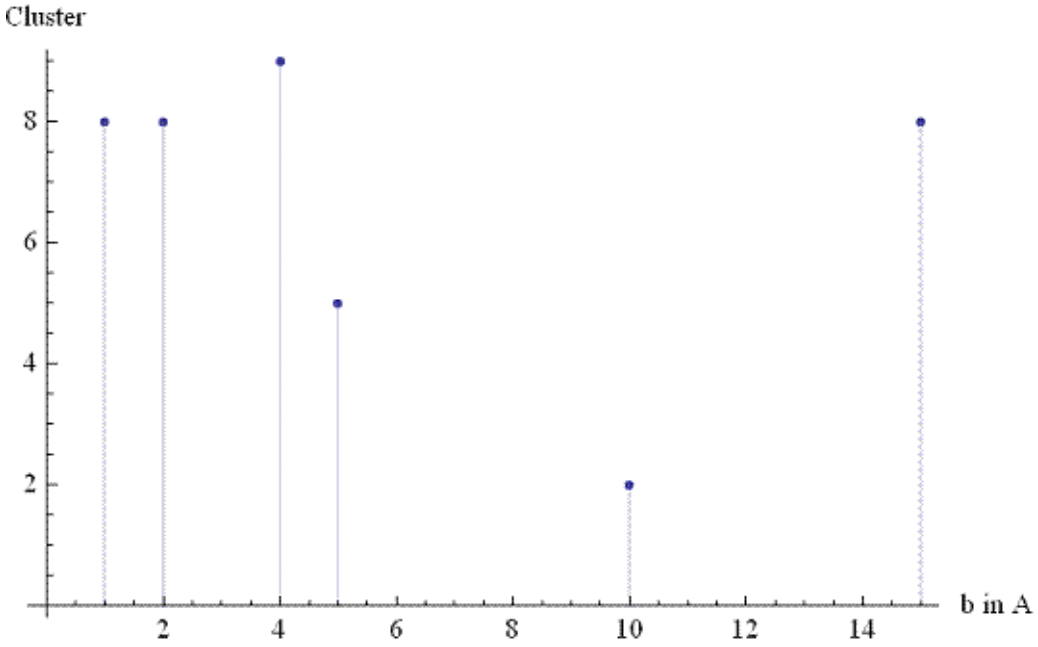

Abbildung 4: Stoßparameter in Angström (realisiert auf x–Achse) und Anzahl stabile Cluster nach etwa 1 ns bei 0.5 $\frac{\text{ÅeV}}{\text{ns}}$.

Die zeitliche Teilchenentwicklung kann man hierbei an folgenden Bildern (Abbildung 5) gut sehen (Grundzustand, 0.14 ns, 0.3 ns, 0.4 ns, 1 ns). Hierbei wurden sie mit b=0 und p_0=0.1 $\frac{\text{ÅeV}}{\text{ns}}$ gestartet.

Vergleichen wir hingegen mehrere unterschiedliche Anfangsimpulse, so lässt sich hier erneut beobachten, dass höhere Impulse zu schneller instabil werdenden Systemen führen. Die Abbildungen (siehe Abbildung 6) wurden hierbei erneut mit b=0 aufgenommen. Impulse in $\frac{\text{ÅeV}}{\text{ns}}$: 0.1, 0.3, 0.5, 1.

Betrachten wir nun noch einmal den Stoß-Parameter. bei festgehaltenen p_0=0.5 $\frac{\text{ÅeV}}{\text{ns}}$ wurde nun der Stoßparameter variiert. In Abbildung 7 sind die Stoßparameter 1, 2, 3, 5, 10 und 15 Angström zu sehen nach je etwa 0.5 ns. Hierbei nehmen die Teilchenhaufen bei höheren Stoßparametern mehr "Drall" auf; die Gebilde drehen sich. Speziell lässt sich das bei b=2, 3 und 10 Angström sehen, wie der Teilchenhaufen länglich wird und zu rotieren beginnt

(Stoßrichtung ist x-Achse, hier je rechts unten beschriftet). Bei b=15 Angström kollidieren die Pakete nur noch peripher: auch nach der Kollision bleiben es 2 Pakete.

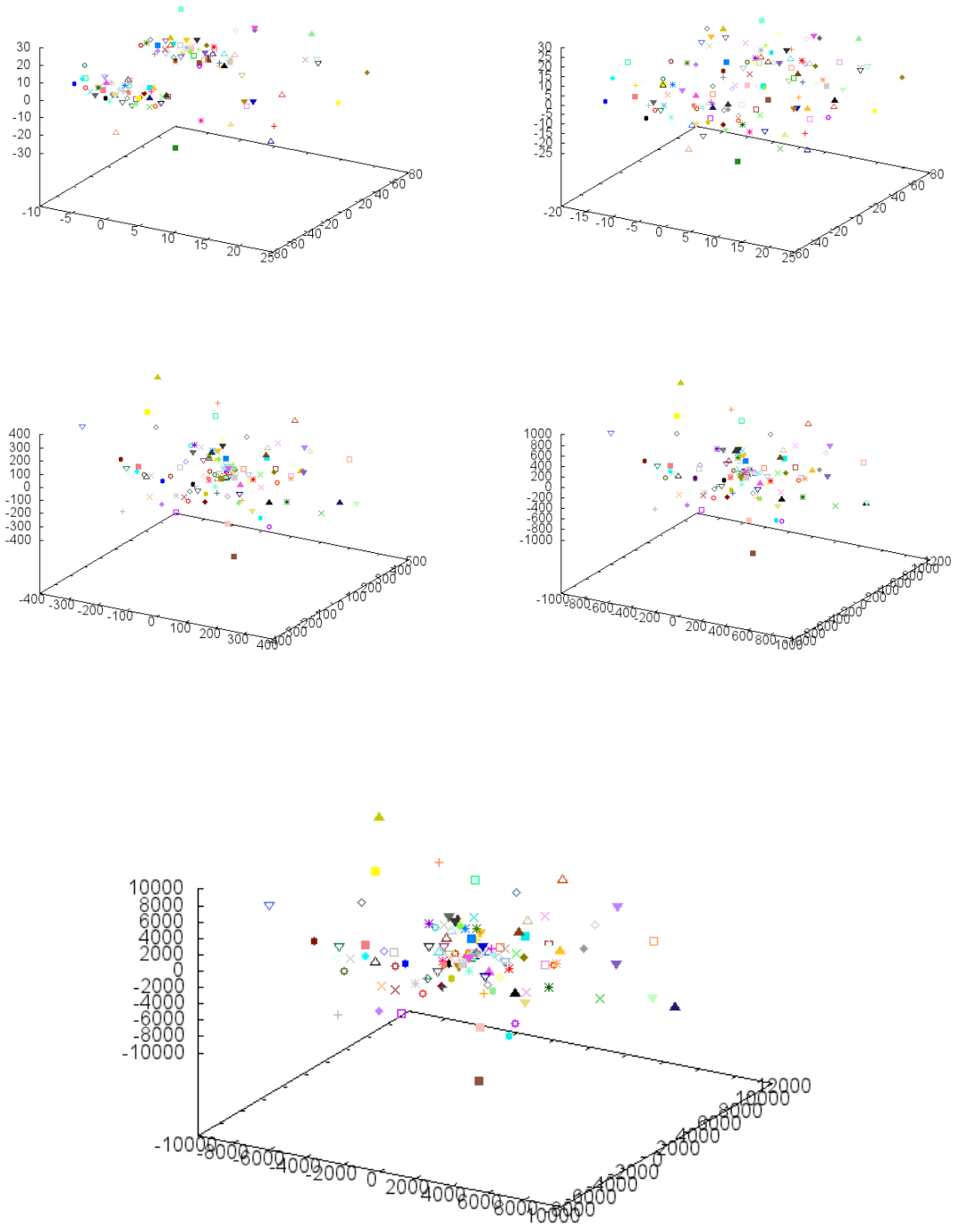

Abbildung 5: Teilchenaufnahmen bei 0 ns, 0.14 ns, 0.3 ns, 0.4 ns und 1 ns mit p_0=0.1 $\frac{\text{ÅeV}}{\text{ns}}$ und b=0

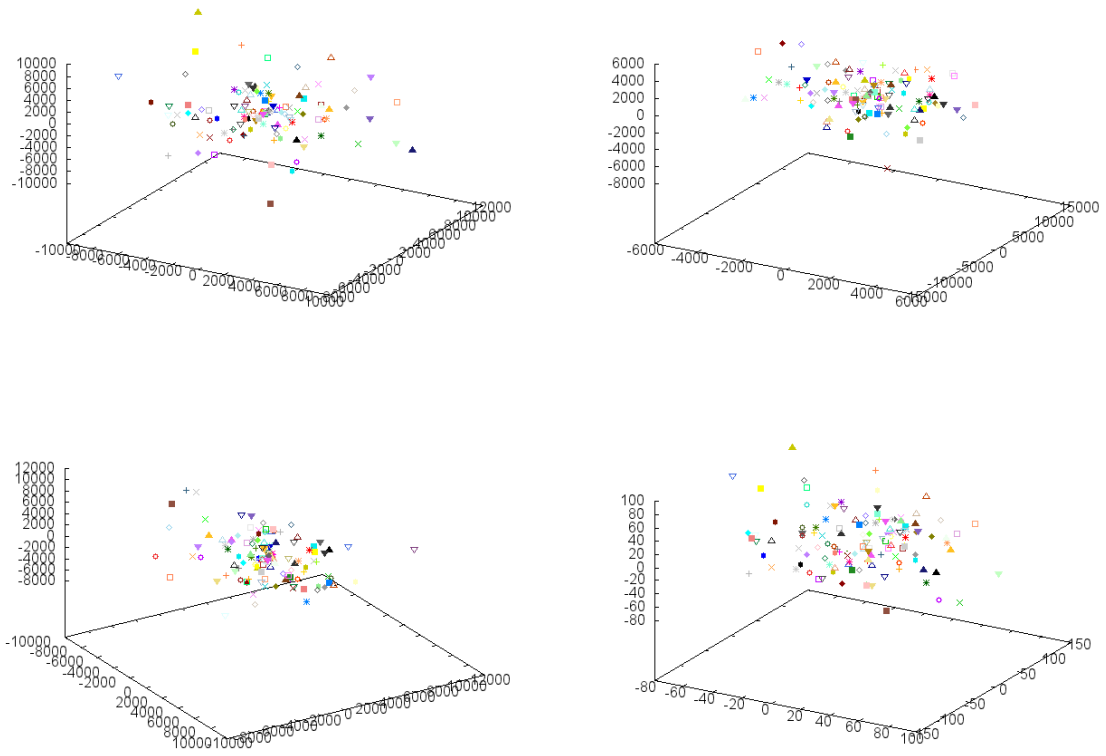

Abbildung 6: Teilchenaufnahmen bei 0.5 ns bei Impulsen in $\frac{\text{ÅeV}}{\text{ns}}$: 0.1, 0.3, 0.5, 1 jeweils b=0

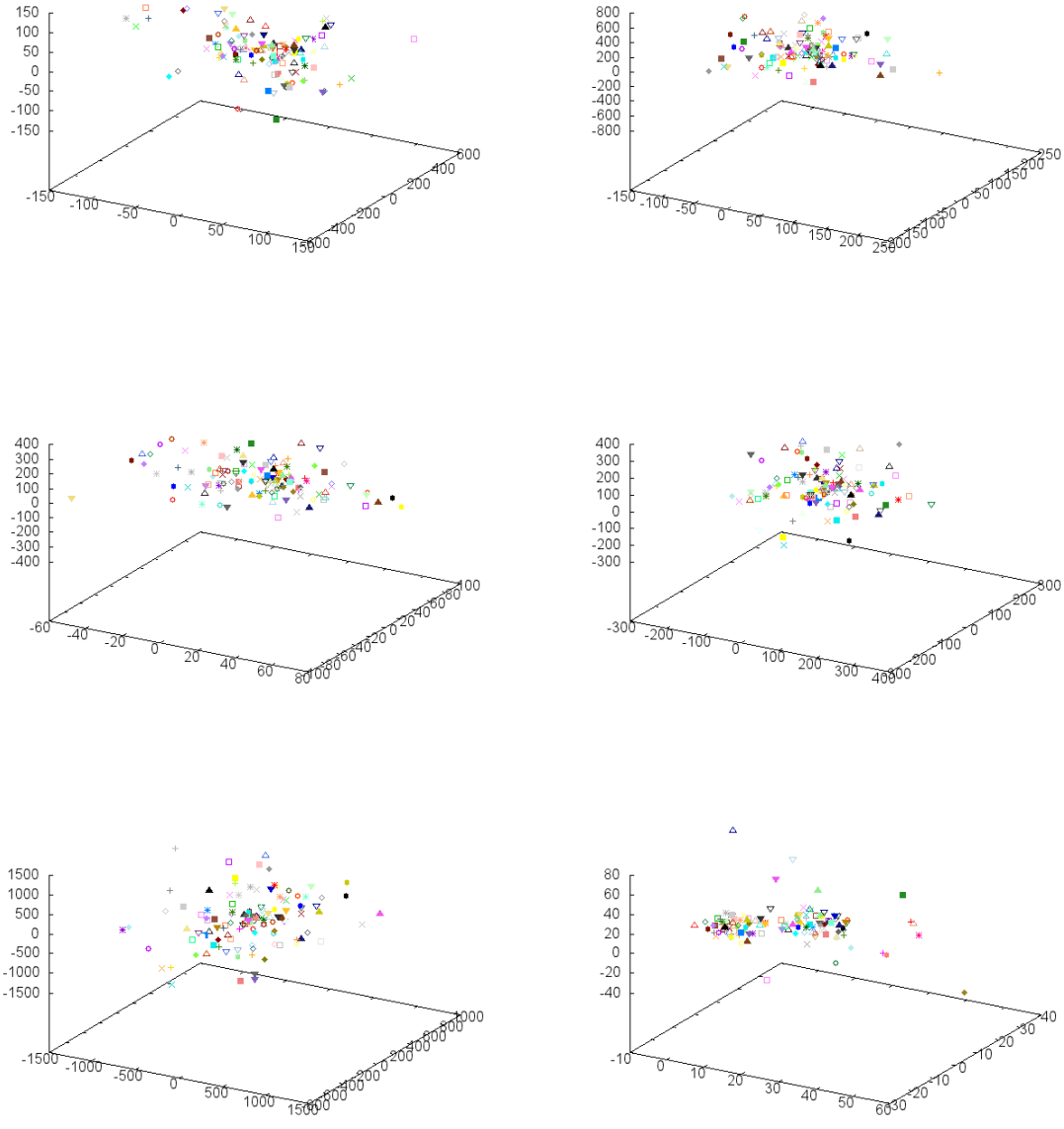

Abbildung 7: Stoßparameter 1, 2, 3, 5, 10 und 15 bei p_0=0.5 $\frac{\text{ÅeV}}{\text{ns}}$ nach etwa 0.5 ns.

4 Quellcode

```
1  #include <iostream>
2  #include <cstdio>
3  #include <time.h>
4
5  #include <fstream>
6  #include <sstream>
7  #include <vector>
8  #include <algorithm>
9
10 #include <stdlib.h>
11 #include <cmath>
12
13 using namespace std;
14
15 /*Dieses Programm simuliert Teilchenkollisionen über Runge Kutta 4. Ordnung.
16 Dabei werden mehrere Teilchen mit definierter Masse zunächst in einem Gitter
       angeordnet und in den Grundzustand fallen gelassen.
17 Die so entstandene Datei kann wieder eingelesen werden, um die Teilchen weiter
       abklingen zu lassen oder
18 um die Teilchen 2 Mal einzufügen und aufeinander zu schießen.
19 Stoßparameter, Anfangsimpulse und Temperatur sind dabei anpassbar.
20 Ausgegeben wird der Endzustand der Teilchen bzw. Wahlweise eine Animation der
       Teilchenbewegung mit aktueller Uhrzeit als Datum.
21 Benötigt wird gnuplot und microsoft visual studio c++ 2010 express o.Ä.
22 */
23 //Neue Blockgröße
24 int anzahl=4*4*4*2; //2 4*4*4-Blöcke aufeinander
25 int kantenlang=4; //Kantenlänge der Blöcke für neue Blöcke. pow(4*4*4,1/3.0) ist
       hier zu ungenau...
26 int   tstart=0; //nur proforma, braucht nicht geändert werden. Erweiterbar: falls
       zeitabhängigkeit in Rungekutta erwünscht!
27 long schritte=10000;//=>0.005ps
28 double tende=0.05,//ns
29    inabst1=4.3, //Rasterabstand Angström Block1 (bei neuen Dateien)
30    temp=300; //Temperatur (K) über Monte-Carlo (bis 10000K). Restliche Parameter
31 //restliche Parameter: Stoßparameter, Anfangsimpuls, Datei einlesen oder neue
       erzeugen siehe in letzter Routine "main"!
32
33 double const masse=0.0028;//28u, alle Teilchen gleiche Masse, 1u =1.036426866d-4 eV
       ps^2/A^2
34
35 //Parameter für Potential
36 const double a=5,
37    b=0.27,
38    sigma1=1.3,
39    sigma2=3.0,
40    gamma=10, //Reibung für Grundzustand
41    kb=0.000086173324, //Boltzmann
42    pi=3.14159265359,
43    c=299792458;
44
45
46 //Zwischenvariabeln für Drehimpuls/Drehtransfos.
```

```
47  vector<double>mitte(3,0),drehimp(3,0),vektor(3,0),winkelg(3,0),l(anzahl*6,0),r0(
        anzahl*6,0),r01(anzahl*6,0),schwerp(3,0),winkelg1(3,0);
48  vector<vector<double>>traeg(3,vector<double>(3,0));
49  vector<vector<double>>temp1(3,vector<double>(4,0)),temp2(3,vector<double>(4,0));
50  double rw,phiw,thetaw;
51
52
53  //Berechnet Drehimpuls und Winkelgeschwindigkeit (über Gauß).
54  //Dadurch lässt sich der Grundzustand so drehen, dass die Abweichung leichter
        ablesbar wird.
55  void Drehimpuls(vector<double> &y){
56     int i,j;
57
58     for(i=0;i<anzahl;i++){ //Schwerpunkt
59       mitte[0] += y[6*i+1]/anzahl;
60       mitte[1] += y[6*i+2]/anzahl;
61       mitte[2] += y[6*i+3]/anzahl;
62     }
63     for(i=0;i<3;i++)
64        std::cout<<"m:"<<mitte[i]<<" ";
65
66     for(i=0;i<anzahl;i++){ //Relativkoordinaten zum Schwerpunkt
67       y[6*i] -=mitte[0];
68       y[6*i+1] -=mitte[1];
69       y[6*i+2] -=mitte[2];
70     }
71     for(i=0;i<anzahl;i++){ //Drehimpuls rel. z. SP.
72       drehimp[0] +=y[6*i+1]*y[6*i+5]-y[6*i+2]*y[6*i+4];
73       drehimp[1] +=y[6*i+2]*y[6*i+3]-y[6*i+0]*y[6*i+5];
74       drehimp[2] +=y[6*i+0]*y[6*i+4]-y[6*i+1]*y[6*i+3];
75     }
76     for(i=0;i<3;i++)
77        std::cout<<"d:"<<drehimp[i]<<" ";
78
79     for(i=0;i<anzahl;i++){ //Trägheitstensor
80        traeg[0][0]+=masse*(pow(y[6*i+1],2)+pow(y[6*i+2],2));
81        traeg[1][1]+=masse*(pow(y[6*i],2)+pow(y[6*i+2],2));
82        traeg[2][2]+=masse*(pow(y[6*i],2)+pow(y[6*i+1],2));
83        traeg[0][1]-=masse*y[6*i]*y[6*i+1];
84        traeg[0][2]-=masse*y[6*i]*y[6*i+2];
85        traeg[1][2]-=masse*y[6*i+1]*y[6*i+2];
86     }
87     traeg[1][0]=traeg[0][1];
88     traeg[2][0]=traeg[0][2];
89     traeg[2][1]=traeg[1][2];
90
91     for(int i=0;i<3;i++){ //Gauß
92        for(int j=0;j<3;j++){
93           temp1[i][j] = traeg[i][j];
94           temp1[i][3] = drehimp[i];
95        }
96     }
97     temp2 = temp1;
98     for(int i=0;i<4;i++){
99        temp1[2][i]-=temp2[2][0]/temp2[1][0] *temp2[1][i];
100       temp1[1][i]-=temp2[1][0]/temp2[0][0] *temp2[0][i];
```

```
101     }
102     temp2 = temp1;
103     for(int i=0;i<4;i++){
104        temp1[2][i]=temp2[2][i] -temp2[2][1]/temp2[1][1] *temp2[1][i];
105     }
106
107     //winkelgeschwindigkeit
108     winkelg[2] = temp1[2][3]/temp1[2][2];
109     winkelg[1] = (temp1[1][3]- winkelg[2]*temp1[1][2])/temp1[1][1];
110     winkelg[0]= (temp1[0][3]- winkelg[2]*temp1[0][2]- winkelg[1]*temp1[0][1])/temp1
            [0][0];
111
112     for(i=0;i<3;i++)
113        std::cout<<"d:"<<drehimp[i]<<" ";
114
115     //Bahndrehimpuls erhalten?
116     for(i=0;i<3;i++){
117        for(j=0;j<3;j++){
118           vektor[i]+=traeg[i][j]*winkelg[j];
119        }}
120 }
121 void winkelanpassung(vector<double> &y){
122     double energie;
123     int i=0;
124     for(i=0;i<anzahl;i++){
125        energie +=(pow(y[6*i+3],2)+pow(y[6*i+4],2)+pow(y[6*i+5],2))/(2.0*masse);
126     }
127     double e0 = 3.0/2.0 *kb*temp; //Schwerpunktsimpuls
128     cout<<(energie/(e0*anzahl));
129     for(i=0;i<anzahl;i++){
130        schwerp[0]+=y[6*i+3]/anzahl;
131        schwerp[1]+=y[6*i+4]/anzahl;
132        schwerp[2]+=y[6*i+5]/anzahl;
133     }
134     for(i=0;i<anzahl;i++){ //Transfo ins SP-impuls-System
135        y[6*i+3]-=schwerp[0];
136        y[6*i+4]-=schwerp[1];
137        y[6*i+5]-=schwerp[2];
138     }
139     for(i=0;i<anzahl;i++){ //Massenmittelpunkt
140        mitte[0]+=y[6*i]/anzahl;
141        mitte[1]+=y[6*i+1]/anzahl;
142        mitte[2]+=y[6*i+2]/anzahl;
143     }
144     for(i=0;i<anzahl;i++){//Relativkoordinaten!
145        r0[6*i]-=mitte[0];
146        r0[6*i+1]-=mitte[1];
147        r0[6*i+2]-=mitte[2];
148     }
149     Drehimpuls(y);
150     //Kugelkoordinaten d. Winkelgeschw.
151     rw=sqrt(pow(winkelg[0],2)+pow(winkelg[1],2)+pow(winkelg[2],2));
152     phiw=atan2(winkelg[1],winkelg[0]);
153     thetaw=acos(winkelg[2]/rw);
154
155     //Drehmatrizen in entspr. Richtung
```

```
156     vector<vector<double>>drehz(3,vector<double>(3,0));
157     drehz[0][0]=cos(-phiw);
158     drehz[0][1]=sin(-phiw);
159     drehz[0][2]=0;
160     drehz[1][0]=-sin(-phiw);
161     drehz[1][1]=cos(-phiw);
162     drehz[1][2]=0;
163     drehz[2][0]=0;
164     drehz[2][1]=0;
165     drehz[2][2]=1;
166     vector<vector<double>>drehy(3,vector<double>(3,0));
167     drehy[0][0]=cos(-thetaw);
168     drehy[0][1]=0;
169     drehy[0][2]=-sin(-thetaw);
170     drehy[1][0]=0;
171     drehy[1][1]=1;
172     drehy[1][2]=0;
173     drehy[2][0]=sin(-thetaw);
174     drehy[2][1]=0;
175     drehy[2][2]=cos(-thetaw);
176
177     for(int i=0;i<3;i++){
178        for(int j=0;j<3;j++){
179           winkelg1[i]+=drehz[i][j]*winkelg[j];
180     }}
181     for(int i=0;i<3;i++)
182        cout<<"winkelg1"<<winkelg1[i]; //Hier Test: x=y=0, z=w !
183
184     winkelg = vector<double>(3,0);
185
186     for(int i=0;i<3;i++){
187        for(int j=0;j<3;j++){
188           winkelg[i]+=drehy[i][j]*winkelg1[j];
189        }
190     }
191
192     for(int i=0;i<3;i++)
193        cout<<"Winkelgeschwindigkeit transformiert"<<winkelg[i]<<endl;
194
195     for(int i=0;i<anzahl;i++){
196        for(int j=0;j<3;j++){
197           for(int k=0;k<3;k++){
198              r01[6*i+j]+=drehz[j][k]*r0[6*i+k];
199           }}}
200     //Nun Transfo von y:
201     r0 = vector<double>(anzahl*6,0);
202     for(int i=0;i<anzahl;i++){
203        for(int j=0;j<3;j++){
204           for(int k=0;k<3;k++){
205              r0[6*i+j]+=drehy[j][k]*r01[6*i+k];
206           }}}
207     r01 = vector<double>(anzahl*6,0);
208     for(int i=0;i<anzahl;i++){
209        for(int j=0;j<3;j++){
210           for(int k=0;k<3;k++){
211              r01[6*i+j]+=drehz[j][k]*y[6*i+k];
```

```
212             r01[6*i+j+3]+=drehz[j][k]*y[6*i+k+3];
213          }}}
214    y = vector<double>(anzahl*6,0);
215    for(int i=0;i<anzahl;i++){
216       for(int j=0;j<3;j++){
217          for(int k=0;k<3;k++){
218             y[6*i+j]+=drehy[j][k]*r01[6*i+k];
219             y[6*i+j+3]+=drehy[j][k]*r01[6*i+k+3];
220          }}}
221    //Kugelkoordinaten des Grundzustandes.
222    //da w in r-Richtung: phi+=rw*(schritte/anzahl) !
223    for(int i=0;i<anzahl;i++){
224       l[3*i] = sqrt(pow(r0[6*i],2) +pow(r0[6*i+1],2) +pow(r0[6*i+2],2));
225       l[3*i+1] = atan2(r0[6*i+1],r0[6*i]);
226       l[3*i+2] = acos(r0[6*i+2]/l[3*i]);
227    }
228 }
229
230 //Maxwell Boltzmann Verteilung mit Monte Carlo Methode
231 void montecarlo(vector<double> &y){
232    //vertmax: Funktion für Maxima der Verteilung (über Mathematica)
233    double vertmax = 181.69723638904276- 49036.1596250/(temp*temp) +
234       12007.293973803038/temp - 0.2326018089098843 *temp +
235       0.00020143492718019792 *temp*temp - 9.889627385975435*pow(10.0,-8)*pow(temp,3)
             +
236       2.8465004602038517*pow(10.0,-11) *pow(temp,4) - 4.898019128141887*pow
             (10.0,-15) *pow(temp,5) +
237       4.950284303906514*pow(10.0,-19) *pow(temp,6) - 2.7071514724946854*pow
             (10.0,-23) *pow(temp,7) +
238       6.176463667836648*pow(10.0,-28) *pow(temp,8);
239    double betragmax = 0.0017642832638790238+ 0.0012577681693344063*sqrt(temp)+0.002;
          //Verteilung hätte hier 10^-5
240    double betrag,test,vert,theta,phi;
241    srand(time(NULL));
242    int j=0;
243    for(int i=0;i<anzahl;i++){
244       for(j=0;j<2000000;j++){
245          vert = ((double)rand()/RAND_MAX)*vertmax; //zufällige maxwell-boltzmann  //
                rand()/RAND_MAX generiert eine Zufallszahl aus dem Intervall (0,1]
246          theta = pi*((double)rand()/RAND_MAX);
247          phi = 2*pi*((double)rand()/RAND_MAX);
248          betrag = betragmax*((double)rand()/RAND_MAX);
249          test = betrag*betrag/(2*masse*kb*temp);
250          if(test<=397 && !(vert>pow(2*kb*temp*masse*pi,-3/2) *4*pi*betrag*betrag*exp
                (-test))){
251             break;
252          }
253       }
254       if(j==2000000){cout<<"Mehr als 2d6 Überschreitungen!";}
255       y[6*i+3] = betrag*sin(theta)*cos(phi);
256       y[6*i+4] = betrag*sin(theta)*sin(phi);
257       y[6*i+5] = betrag*cos(theta);
258    }
259 }
260
```

```
261 //Runge Kutta Algorithmus 4. Dimension übernommen aus numerical recipies, jedoch
        angepasst auf double und vector.
262 void rk4(vector<double> y, vector<double> dydx, double n, double x, double h, vector
        <double> &yout,
263 void (*derivs)(double, vector<double>,  vector<double>&)){ //derives sit
        Gleichungssystem
264     int i;
265     double xh,hh,h6;
266     vector<double> dym(n,1);
267     vector<double> dyt(n,1);
268     vector<double> yt(n,1);
269     hh=h*0.5;
270     h6=h/6.0;
271     xh=x+hh;
272     for (i=0;i<n;i++) yt[i]=y[i]+hh*dydx[i]; //dydx=k1
273     (*derivs)(xh,yt,dyt);                    //dyt=k2
274     for (i=0;i<n;i++) yt[i]=y[i]+hh*dyt[i];
275     (*derivs)(xh,yt,dym);                    //dym=k3
276     for (i=0;i<n;i++) {
277         yt[i]=y[i]+h*dym[i];
278         dym[i] += dyt[i];                //dym=k2+k3
279     }
280     (*derivs)(x+h,yt,dyt);                   //dyt=k4
281     for (i=0;i<n;i++)
282         yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]); //ausgabe= y+ h/6 *(k1+k4+2(k2+k3
            ))
283     dym.clear();
284     dyt.clear();
285     yt.clear();
286
287 }
288
289 //Aufruf/Schnittstelle zu Runge Kutta. Hier wird auch E-Erhaltung überprüft etc.
290 void rkdumb(vector<double> vstart, double nvar, double x1, double x2, double nstep,
        vector<vector<double> > &yy,vector<double> &xx,
291 void (*derivs)(double, vector<double>,vector<double>&))
292 {
293     int i,k;
294     double x,h,betr;
295     vector<double> v(nvar,1); //v ist im jeweiligen Schritt der aktuelle Satz an Orts
            /Impulskoordinaten
296     vector<double> vout(nvar,1);
297     vector<double> dv(nvar,1);
298     for (i=0;i<nvar;i++) { //Load starting values.
299         v[i]=vstart[i];
300         yy[i][0]=v[i]; //y speichert alle Werte jedes Schrittes ab. 0. Schritt sollen
                Startwerte sein.
301     }
302     xx[0]=x1; //Zeit. Wird nicht weiter benutzt, aber zur Erweiterung des Runge Kutta
            mit zeitabhängigen Potential benutzbar
303     x=x1;
304     h=(x2-x1)/nstep;
305     cout<<"\r000%"; //Fortschritt anzeigen.
306     double kngesenergie,pngesenergie; //aktuelle kin und pot energie
307     double gesenergie=0; //anfängliche energie
308     for (k=0;k<nstep;k++) { //Take nstep steps.
```

```
309         (*derivs)(x,v,dv);
310         rk4(v,dv,nvar,x,h,vout,derivs); //Runge Kutta anwenden. derivs definiert
                wirkende Potentiale, v ist Ausgabe
311         x += h;
312         xx[k+1]=x; //Store intermediate steps.
313
314         for (i=0;i<nvar;i++) {
315            v[i]=vout[i];
316            yy[i][k+1]=v[i]; //Lösung aktuellen Schritt abspeichern.
317         }
318         //Korrektur Drehimpuls
319         for(int j=0;j<anzahl;j++)
320            l[3*j+1] += rw*(tende/schritte);
321         for(int j=0;j<anzahl;j++){ //in karth. Koord.
322            r0[6*j+0] = l[3*j]*sin(l[3*j+2])*cos(l[3*j+1]);
323            r0[6*j+1] = l[3*j]*sin(l[3*j+2])*sin(l[3*j+1]);
324            r0[6*j+2] = l[3*j]*cos(l[3*j+2]);
325         }
326
327         //Test ob SP-Impuls=0.
328         schwerp = vector<double>(3,0);
329         for(int j=0;j<anzahl;j++){
330            schwerp[0]+=v[6*j+3];
331            schwerp[1]+=v[6*j+4];
332            schwerp[2]+=v[6*j+5];
333         }
334
335         //Alle 100 Schritte überprüfen...
336         if(k%100==1 && (schwerp[0]+schwerp[1]+schwerp[2])/anzahl<0.000001)cout<<"sp
                weg\n";
337         drehimp = vector<double>(3,0);
338         for(int j=0;j<anzahl;j++){
339            drehimp[0]+=v[6*j+1]*v[6*j+5]-v[6*j+2]*v[6*j+4];
340            drehimp[1]+=v[6*j+2]*v[6*j+3]-v[6*j]*v[6*j+5];
341            drehimp[2]+=v[6*j]*v[6*j+4]-v[6*j+1]*v[6*j+3];
342         }
343         //Test ob Dreh-Impuls=0.
344         //Alle 100 Schritte überprüfen...
345         if(k%100==1&&(drehimp[0]+drehimp[1]+drehimp[2])/anzahl<0.000001)cout<<"dp weg\
                n";
346
347         pngesenergie=0;//potentielle Energie
348         kngesenergie=0;//kinetische Energie
349         for(int i=0;i<anzahl-1;i++){
350               for(int j=i+1;j<anzahl;j++){
351               betr=sqrt(pow(v[i*6]-v[j*6],2)+pow(v[i*6+1]-v[j*6+1],2)+pow(v[i*6+2]-v[j
                      *6+2],2));
352               pngesenergie+=a*exp(-0.5*pow(betr/sigma1,2)) - b*exp(-0.5*pow(betr/
                      sigma2,2));//potentielle Energie
353            }
354            kngesenergie+=(pow(v[6*i+3],2)+pow(v[6*i+4],2)+pow(v[6*i+5],2))/(2.0*masse)
                   ;//kinetische Energie
355         }
356         kngesenergie+=(pow(v[6*(anzahl-1)+3],2)+pow(v[6*(anzahl-1)+4],2)+pow(v[6*(
                anzahl-1)+5],2))/(2.0*masse);//kinetische Energie
357
```

```
358         //Energieerhaltung überprüfen.
359         //Immer ausgeben: kin und pot.
360         if(gesenergie==0)gesenergie=kngesenergie+pngesenergie;
361         if(k%100==1)cout<<"           k"<<kngesenergie<<" - p"<<pngesenergie;
362         if(k%100==1&&abs(kngesenergie+pngesenergie)<abs(gesenergie*0.9))
363            cout<<"\nEnergieerhaltung verletzt: Schrittweite verringern: "<<
                   kngesenergie+pngesenergie<<" - "<<gesenergie;
364         if(k%100==1&&abs(kngesenergie+pngesenergie)>abs(gesenergie*1.1))
365            cout<<"\nEnergieerhaltung verletzt: Schrittweite verringern: "<<
                   kngesenergie+pngesenergie<<" - "<<gesenergie;
366
367         if(k%100==1)cout<<"\r    "<<"\r"<<(k/nstep)*100<<"%"; //Fortschritt anzeigen.
368         }
369      v.clear();vout.clear();dv.clear();
370      cout<<"\r     "<<"\r100%\n";
371   }
372
373   //Potential definieren
374   void myderiv(double x, vector<double> y, vector<double> &dxdy){
375      double betr=0;
376      double temp=0;
377
378      for(int i=0;i<anzahl;i++){
379         dxdy[i*6] = y[6*i+3]/masse; //dx/dt = p/m
380         dxdy[i*6+1] = y[6*i+4]/masse;
381         dxdy[i*6+2] = y[6*i+5]/masse;
382
383         for(int j=i+1;j<anzahl;j++){
384            betr=sqrt(pow(y[i*6]-y[j*6],2)+pow(y[i*6+1]-y[j*6+1],2)+pow(y[i*6+2]-y[j
                   *6+2],2)); //Abstand Teilchen voneinander
385
386            //dp/dt= F
387            temp=a/(sigma1*sigma1)*exp(-0.5/(sigma1*sigma1)*betr*betr)-b/(sigma2*sigma2
                   ) *exp(-0.5/(sigma2*sigma2)*(betr)*(betr));
388
389            dxdy[i*6+3]+=temp*(y[6*i]-y[6*j]);
390            dxdy[i*6+4]+=temp*(y[6*i+1]-y[6*j+1]);
391            dxdy[i*6+5]+=temp*(y[6*i+2]-y[6*j+2]);
392            dxdy[j*6+3]-=temp*(y[6*i]-y[6*j]);
393            dxdy[j*6+4]-=temp*(y[6*i+1]-y[6*j+1]);
394            dxdy[j*6+5]-=temp*(y[6*i+2]-y[6*j+2]);
395         }
396
397         /*dxdy[i*6+3] -= gamma*y[6*i+3]; //Reibung, für Grundzustand
398         dxdy[i*6+4] -= gamma*y[6*i+4];
399         dxdy[i*6+5] -= gamma*y[6*i+5];*/
400      }
401   }
402   //string to double
403   template<class T> T fromString(const std::string& s)
404   {
405        std::istringstream stream (s);
406        T t;
407        stream >> t;
408        return t;
409   }
```

```
410 //aktuelle Uhrzeit für Dateiausgabe von Animationen
411 char* currentDateTime() {
412     time_t     now = time(0);
413     struct tm  tstruct;
414     char       buf[80];
415     tstruct = *localtime(&now);
416     strftime(buf, sizeof(buf), "%Y_%m_%d_%H_%M", &tstruct);
417 
418     return buf;
419 }
420 int main(){
421    cout<<"hello!"<<endl;  //Begrüßung
422    vector<double>t(schritte+1,0); //Zeitschritte, hier nicht benutzt, aber möglich f
       ür Zeitabhängigkeit
423    ifstream readfile;
424    vector<double>y(anzahl*6,0); //die Teilchen!
425    vector<double>y1(anzahl/2*6,0); //doppeltes einlesen vom Grundzustand sodass
       Teilchen gegeneinander geschossen werden können
426    //x1 y1 z1 px1 py1 pz1 , x0 y0 z0 px0 py0 pz0
427 
428    //Dieser Bereich belegt ein Gitter der Kantenlänge "kantenlang" im Abstand von
       inabst1 mit Teilchen.
429    /*
430    for(int i=0;i<anzahl;i++){
431       y[i*6]=inabst1*(i%kantenlang);//+ausabst/2;     //Gitter, inabst1 A° Abstand
           intern, ausabst A° Abstand von anderen Gitter
432       y[i*6+1]=inabst1*((i/kantenlang)%kantenlang);
433       y[i*6+2]=inabst1*((i/(kantenlang*kantenlang))%kantenlang);
434       //px= A° eV/ns =0.1 eV * m/s = 3.3356*10^-10 eV/c
435    }*/
436 
437 
438    //Dieser Bereich lädt die datei grundzustand2.temp.
439    //das Format ist dabei
440    /*
441    x
442    y
443    z
444    px
445    py
446    pz
447    x
448    ...
449    */
450    //geladen wird in y1, sodass die Daten 2 mal mit Abstand und gespiegelt eingefügt
        werden können
451    readfile.open("grundzustand2.temp");
452    std::string line;
453    int z=0;
454    for(int i=0;i<anzahl*6/2;i++){
455       getline(readfile,line);
456       y1[i]=fromString<double>(line);
457    }
458    readfile.close();
459 
460    //Wie vorheriger Bereich, Unterschied: hier werden Daten direkt eingelesen
```

```
461     //Reichen die bisherigen Schritte/Zeit nicht aus, kann hier alles erneut geladen
            und weiter gerechnet werden
462
463     /*readfile.open("endzustand.temp");
464     std::string line;
465     int z=0;
466     for(int i=0;i<anzahl*6;i++){
467        getline(readfile,line);
468        y[i]=fromString<double>(line);
469     }
470     readfile.close();*/
471
472     //Test für das erstellen von 2 Gittern, die man aufeinander schießen könnte
473     //Achtung: keine Grundzustände hier!
474     /*
475     for(int i=0;i<anzahl/2;i++){
476        y1[i*6]=inabst1*(i%kantenlang);//+ausabst/2;    //Gitter, inabst1 A° Abstand
               intern, ausabst A° Abstand von anderen Gitter
477        y1[i*6+1]=inabst1*((i/kantenlang)%kantenlang);
478        y1[i*6+2]=inabst1*((i/(kantenlang*kantenlang))%kantenlang);
479     }
480     for(int i=anzahl/2;i<anzahl;i++){
481        y[i*6]=inabst2*(i%kantenlang)-ausabst/2;     //gitter, inabst2 a° abstand
               intern, ausabst a° abstand von anderen gitter
482        y[i*6+1]=inabst2*((i/kantenlang)%kantenlang);
483        y[i*6+2]=inabst2*((i/(kantenlang*kantenlang))%kantenlang);
484        y[i*6+3]=50;
485     }
486     */
487
488     anzahl/=2; // verechnen eines Blockes dieser wird dann verschoben 2 mal eingefügt
            zur Kollision.
489     r0=y1;
490     montecarlo(y1);//Verteilen der Impulse des Grundzustandes
491     winkelanpassung(y1);//Schwerpunktsystem und Drehimpulse
492
493     //2 mal einfügen von einem Paket
494     for(int i=0;i<anzahl*6;i++){ //Achtung: hier ist anzahl noch halbiert!
495        y[i]=y1[i]; //1. Block einfügen
496        if(i%6==1) //
497           y[i+anzahl*6]=-y1[i];//2. Block an y-Achse spiegeln
498        else
499           y[i+anzahl*6]=y1[i]; //ansonsten 2. Block wie erster Block
500
501        if(i%6==4){y[i]+=0.5;y[i+anzahl*6]-=0.5;}//y-impuls (A° eV/ns ~ 0.1 eV * m/s ~
               3.3356*10^-10 eV/c)
502        if(i%6==1){y[i]-=20;y[i+anzahl*6]+=20;}//y-achse => abstand
503        if(i%6==0){y[i]+=15;}//x-achse => stoßparam
504     }
505     anzahl*=2; //tatsächliche Anzahl wieder herstellen
506
507     vector<vector<double> > yy(anzahl*6,vector<double>(schritte+1,0)); //Ergebnis: y[
            index][schritt]
508     vector<double>dxdy(anzahl*6,0);
509     vector<double>tout(anzahl*6,0);
510
```

```
511     //Runge Kutta-Schnittstelle
512     rkdumb(y,anzahl*6,tstart,tende,schritte,yy,t,myderiv);
513
514     wofstream file;
515     file.open("endzustand.temp"); //Ausgabe-Datei des letzten Schrittes! jede Zeile
            eine Zahl: x y z px py pz x ...
516     for(int i=0;i<anzahl*6;i++){
517        file<<yy[i][schritte-1]<<endl;
518     }
519     file.close();
520
521     //zwecks Testzwecken: Zustand am Anfang wie endzustand in Datei ausgeben.
522     file.open("anfangszustand.temp");
523     for(int i=0;i<anzahl*6;i++){
524        file<<yy[i][0]<<endl;
525     }
526     file.close();
527
528     //Plot-Datei für gnuplot!
529
530     file.open("bewegung.temp");
531
532     //Dieser Bereich erstellt Trajektorien aller Teilchen.
533     //Grundsätzlich sinnvoll, aber nicht flüssig rotierbar und leicht unübersichtlich
534
535     /*file<<"unset key\nsplot '-' with lines";
536     for(int j=0;j<anzahl-1;j++){
537        file<<", '-' with lines";
538     }
539     file<<"\n";
540     for(int i=0;i<anzahl;i++){
541        for(int j=0;j<schritte;j+=25){
542           file<<yy[i*6][j]<<" "<<yy[i*6+1][j]<<" "<<yy[i*6+2][j]<<"\n";
543        }
544        file<<"e\n";
545     }
546     */
547
548     //animierte gif-Datei der Teilchen ausgeben. Ergebnis wird nicht angezeigt!
549     //Achtung: warte bis zum Beenden aufgefordert wird!
550
551     file<<"unset key\nset term gif animate\nset output \"animate_"<<currentDateTime()
            <<".gif\"\n";
552
553     //direkte ausgabe auf dem Bildschirm. schwer zu exportieren.
554     //über den Befehl unten in system() lässt sich bewegung.temp aber erneut aufrufen
            .
555
556     //file<<"unset key\n";
557
558     for(int j=0;j<schritte;j+=schritte/500){
559        file<<"splot '-' w p ls 1";
560        for(int i=0;i<anzahl-1;i++){
561           file<<", '-' w p ls "<<i+2;
562        }
563        file<<"\n";
```

```
564         for(int i=0;i<anzahl;i++){
565             file<<yy[i*6][j]<<" "<<yy[i*6+1][j]<<" "<<yy[i*6+2][j]<<"\n";
566             file<<"e\n";
567         }
568         //für direkte Ausgabe: Zeit in Sekunden zwischen den Plots.
569         //in gif-Animation wird immer 100ms benutzt.
570         //file<<"pause 0.05\n";
571     }
572 
573     file.close();
574 
575     //für direkte Ausgabe
576     //system("wgnuplot bewegung.temp -persistent");
577 
578     //für Animation
579     system("wgnuplot bewegung.temp");
580 
581     system("pause");
582 }
```