

Galaxia

Ein n -Körper Simulator in C++

Informatik für Physiker

MAXIMILIAN F. CASPAR (1521165)
JOHANNES N. ESSER (1528193)

22. JULI 2016

Inhaltsverzeichnis

1	Einführung	4
2	Theorie	4
2.1	Gravitationsphysik	4
2.2	Das n -Körper-Problem	4
2.3	Algorithmen	4
2.3.1	Lösung durch ein klassisches Euler-Verfahren	4
2.3.2	Barnes-Hut-Algorithmus	5
3	Galaxia	6
3.1	Kompilierung des Programms	6
3.1.1	Voraussetzungen	6
3.1.2	Kompilierung	6
3.2	Menüführung	6
3.3	Sterne	7
3.4	Interaktion von Sternen mit Parallelisierung	7
3.5	Erzeugung von Galaxien	8
3.6	Dateien	10
3.7	Plots	11
4	Ergebnisse	13
4.1	Die Milchstraße	14
4.2	Die Kollision von Milchstraße und Andromeda	14
4.3	Chaostheorie	15
5	Anhang	16
5.1	main.cpp	16
5.2	stars.h	17
5.3	gravity.h	18
5.4	plot.h	20
5.5	fileio.h	22
5.6	scenarios.h	24

1 Einführung

Aufgabe des Einführungskurses Informatik für Physiker war es, ein kleines Programmierprojekt zu erstellen, welches ein konkretes, nicht analytisch lösbares Problem numerisch simuliert. Wir haben uns für das Verhalten von Massen mit gegenseitiger gravitativer Wechselwirkung entschieden. Das Programmierprojekt enthält 6 selbst geschriebene Quelldateien, eine Dokumentation und ein Makefile. Wir haben uns für eine Implementierung in 2 Dimensionen entschieden, ein Code in 3D ist analog möglich.

2 Theorie

2.1 Gravitationsphysik

2.2 Das n -Körper-Problem

Für dieses Protokoll werden wir uns auf den Fall der klassischen Gravitation nach Newton beschränken, moderne astrophysikalische Rechnungen beziehen Faktoren wie allgemeine Relativitätstheorie und dunkle Materie in die Simulation mit ein. Bekanntermaßen folgt die Bewegung zweier massebehafteter Körper im Raum der Bewegungsgleichung

$$m_1 \cdot \ddot{\vec{s}}_1 = -m_2 \cdot \ddot{\vec{s}}_2 = G \frac{m_1 \cdot m_2}{|\vec{s}_2 - \vec{s}_1|^2} \frac{\vec{s}_2 - \vec{s}_1}{|\vec{s}_2 - \vec{s}_1|}$$

Für eine Menge von n Körpern (wobei $n \geq 3$) existiert für die Bewegung der Körper keine analytische Lösung. Wir werden deswegen im Folgenden beschreiben, wie durch numerische Verfahren die Lösung angenähert werden kann.

2.3 Algorithmen

An dieser Stelle wollen wir kurz zwei häufig verwendete Algorithmen für n -Körper-Probleme vorstellen.

2.3.1 Lösung durch ein klassisches Euler-Verfahren

Die klassische Berechnung eines Simulationsschrittes läuft so ab:

1. Berechne die Kräfte:
 - Wähle einen Stern.
 - Berechne die Kraft, die alle anderen Sterne auf diesen Stern auswirken und addiere diese auf.
 - Wähle einen anderen Stern. Fahre fort, bis die Kraft für alle Sterne berechnet wurde.
2. Bewege alle Sterne entsprechend ihrer aktuellen Geschwindigkeit mit $\vec{s}_{i+1} \approx \vec{s}_i + \vec{v}_i \cdot \Delta t$.
3. Approximiere die neue Geschwindigkeit aller Sterne nach $\vec{v}_{i+1} \approx \vec{v}_i + \frac{\vec{F}_i}{m_s} \cdot \Delta t$

4. Setze die Kräfte auf die einzelnen Sterne auf $\vec{0}$.

Der Nachteil dieses Verfahrens ist, dass jede Bewegung eines Sterns als abschnittsweise lineare Funktion berechnet wird. Kommen sich zwei Massen sehr nahe, tritt eine sehr hohe Beschleunigung auf, was zu einer hohen momentanen Geschwindigkeit führt. So kann es passieren, dass sich ein Stern plötzlich mit hoher Geschwindigkeit aus dem System entfernt, während die Energieerhaltung verletzt wird. Eine sehr fein gewählte Schrittweite löst dieses Problem, erhöht aber den Rechenaufwand massiv.

Jeder Stern interagiert mit $n - 1$ anderen Sternen, somit ist die gesamte Anzahl an Interaktionen $n \cdot (n - 1) = n^2 - n$. Daher ist der Aufwand der Berechnungen von der Ordnung $\mathcal{O}(n^2)$.

2.3.2 Barnes-Hut-Algorithmus

Der Barnes-Hut-Algorithmus reduziert den Rechenaufwand des n -Körper-Problems dadurch, dass der Raum in einzelne Quadranten unterteilt wird, die wiederum auf Unterquadranten referenzieren. Wir wählen den Faktor θ_0 als Abstandskonstante, dann führen wir einen Simulationsschritt aus:

1. Unterteile den Raum in einen Baum mit jeweils vier Knoten. Speichere Masse und Schwerpunkt jedes Knotens.
2. Durchlaufe den Baum rekursiv, für jeden Stern interagiere mit allen Knoten, die $\theta = \frac{d}{r} \leq \theta_0$ erfüllen. Dabei ist d der Durchmesser des Knotens und r sein Abstand zum aktuellen Stern.

Da hier viele Sterne zu Gruppen zusammengefasst werden, reduziert sich der Rechenaufwand beim Barnes-Hut-Algorithmus auf $\mathcal{O}(n \log n)$ Schritte. Allerdings muss der Baum nach jedem Schritt neu berechnet werden, außerdem ist die Implementierung recht aufwändig. Der Algorithmus wird daher üblicherweise nur für sehr hohe Anzahlen von Sternen verwendet, auch da bei größer werdendem θ_0 die Präzision der Simulation stark eingeschränkt ist.

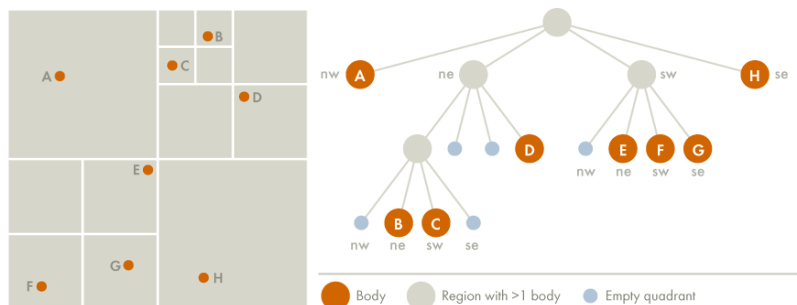


Abbildung 1: Ein Barnes-Hut-Baum (arborjs.org)

3 Galaxia

3.1 Kompilierung des Programms

3.1.1 Voraussetzungen

Folgende Pakete müssen unter Linux installiert sein: *cmake*, *gnuplot*, *doxygen* und *ffmpeg*. Diese können mit dem Befehl `sudo apt-get install gnuplot ffmpeg doxygen cmake` erfüllt werden. Des weiteren muss ein Ordner *boost* angelegt werden, in diesem muss eine aktuelle Version der Boost Bibliothek sein. Diese kann unter <http://www.boost.org/users/download/> bezogen werden.

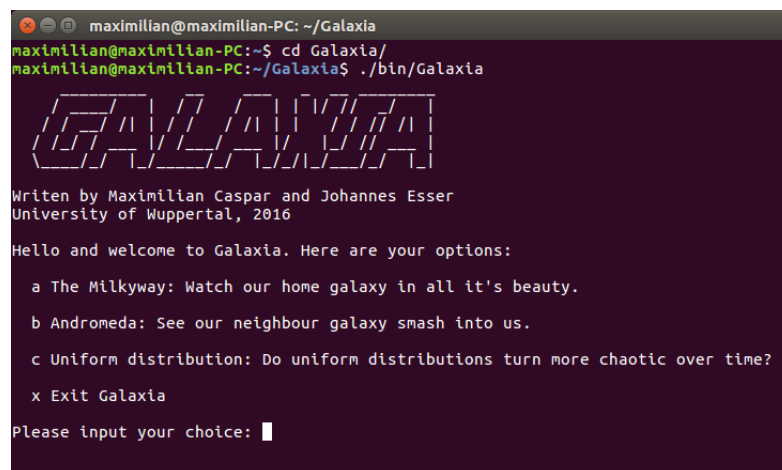
3.1.2 Kompilierung

Folgende Schritte sind zur Kompilierung von Galaxia erforderlich:

1. Benutze `cd boost` um in den boost-Ordner zu gelangen.
2. Führe `./bootstrap.sh` aus um boost zu konfigurieren.
3. Kompiliere boost mit `./b2`. Hol dir einen Kaffee, das könnte dauern!
4. Benutze `cd ..` um zurück in den Galaxia-Ordner zu wechseln.
5. Führe `cmake .` aus um Galaxia zu konfigurieren.
6. Der Befehl `make` kompiliert das Programm.
7. Galaxia kann nun mit `./bin/galaxia` ausgeführt werden.

3.2 Menüführung

Die Menüführung unseres Programms ist simpel gehalten. Man kann zwischen drei Standardszenarien wählen oder das Programm beenden. Die Auswertung der Usereingabe wird über eine *switch* Umgebung ausgewertet, die in eine *while*-Schleife eingebettet ist.



```
maximilian@maximilian-PC: ~/Galaxia
maximilian@maximilian-PC:~$ cd Galaxia/
maximilian@maximilian-PC:~/Galaxia$ ./bin/Galaxia

  GALAXIA

Written by Maximilian Caspar and Johannes Esser
University of Wuppertal, 2016

Hello and welcome to Galaxia. Here are your options:

  a The Milkyway: Watch our home galaxy in all it's beauty.
  b Andromeda: See our neighbour galaxy smash into us.
  c Uniform distribution: Do uniform distributions turn more chaotic over time?
  x Exit Galaxia

Please input your choice: █
```

Abbildung 2: Das Hauptmenü von Galaxia

3.3 Sterne

In der Datei *stars.h* ist das Objekt *Star* definiert. In diesem Objekt sind die Masse, die aktuelle Geschwindigkeit, der Ort und die gerade wirkende Kraft gespeichert. Auch ist die Methode zur Bewegung eines Sterns auf Basis seiner aktuellen Kraft enthalten.

```

1 void move(double dt)
2 {
3     xpos = xpos + xvel*dt;
4     ypos = ypos + yvel*dt;
5     xvel = xvel + (xfor / mass)*dt;
6     yvel = yvel + (yfor / mass)*dt;
7     xfor = 0.0;
8     yfor = 0.0;
9 }
```

Abbildung 3: Die Bewegung des Sterns nach dem Eulerverfahren ist in der *Star*-Klasse implementiert.

3.4 Interaktion von Sternen mit Parallelisierung

Die Sterne interagieren miteinander über das Eulerverfahren. Dabei wird die Bewegung wie oben gezeigt berechnet. Bevor die *move*-Routine aufgerufen werden kann, muss zunächst erst die Kraft auf jeden einzelnen Stern berechnet werden.

```

1 void accelerate(std::vector<Star> & cluster, double dt)
2 {
3     unsigned int n = cluster.size();
4
5     #pragma omp parallel for
6     for(unsigned int i=0; i<n-1;i++)
7     {
8         for(unsigned int j=i+1;j<n;j++)
9         {
10            double xvec = cluster[i].xpos - cluster[j].xpos;
11            double yvec = cluster[i].ypos - cluster[j].ypos;
12            double r = sqrt(xvec*xvec + yvec*yvec);
13            double con = G*cluster[i].mass*cluster[j].mass;
14            cluster[i].xfor -= con*xvec*pow(r,-3);
15            cluster[i].yfor -= con*yvec*pow(r,-3);
16            cluster[j].xfor += con*xvec*pow(r,-3);
17            cluster[j].yfor += con*yvec*pow(r,-3);
18        }
19    }
20
21    #pragma omp parallel for
22    for(unsigned int i = 0; i < n ;i++)
23    {
24        cluster[i].move(dt);
25    }
26 }
```

Abbildung 4: Wechselwirkung aller Sterne in einem Simulationsschritt

Wir schachteln zunächst zwei *for*-Schleifen ineinander, wobei die Schleifen mit *openMP* parallelisiert wurden. In den Schleifen werden jeweils zwei Sterne ausgewählt, der Vektor zwischen ihnen wird bestimmt und mit dem Gravitationsgesetz wird dann die Kraft auf beide Sterne geschrieben. Dabei sind die Indizes der Schleifen so gewählt, dass zwei Sterne jeweils nur einmal miteinander wechselwirken. Da jede Wechselwirkung unabhängig von allen anderen abläuft, können beliebig viele Wechselwirkungen parallel berechnet werden.

3.5 Erzeugung von Galaxien

In Galaxia werden die Anfangsverteilungen von Galaxien mit gaußschen Normalverteilungen ($\mu = 0$, $\sigma = \frac{r}{2}$) approximiert. Dabei muss allerdings darauf geachtet werden, dass Sterne nicht zu nah am Galaxiezentrum erzeugt werden. Einer der Nachteile des Euler-Verfahrens ist nämlich, dass Sterne in der Nähe von großen Massen zu stark beschleunigt werden und dann „durch die Masse hindurch“ fliegen. Um dies zu vermeiden wird ein Schwellwert von $0.05 \cdot r$ eingeführt, ein Mindestabstand, den ein Stern am Anfang vom Galaxiezentrum haben sollte. Dies vermeidet die meisten Rechenfehler recht zuverlässig.

```

1  void make_galaxy(std::vector<Star> & Elements, const double &
    radius, const unsigned int & nos, const double & sx, const
    double & sy, const double & vx, const double & vy, const
    unsigned int & mco)
2  {
3
4      unsigned seed = std::chrono::system_clock::now().time_since_epoch
        ().count();
5      std::default_random_engine generator (seed);
6      std::normal_distribution<double> distribution(0.0, radius/2);
7
8      for (unsigned int i = 0; i < nos; i++)
9      {
10         double x = 0;
11         double y = 0;
12         double rad = 0;
13         while(rad < 0.05*radius)
14         {
15             x = distribution(generator);
16             y = distribution(generator);
17             rad = sqrt(x*x + y*y);
18         }
19         double alpha = get_angle(x, y);
20         double v = orbit_velocity(rad, mco);
21         Star temp(x + sx, y + sy, v*sin(alpha) + vx, -v*cos(alpha) + vy
            , MASS_OF_SUN);
22         Elements.push_back(temp);
23     }
24     Star center(sx, sy, vx, vy, MASS_OF_SUN*mco);
25     Elements.push_back(center);
26 }
```

Abbildung 5: Erzeugung der Galaxien mit einer gaußschen Normalverteilung. Alle Sterne werden auf die Masse der Sonne gesetzt, das Verhältniss zwischen Sternmasse und Zentrumsmasse heißt *mco*

Wie im Code (Zeile 19) zu erkennen wird zur Bestimmung des Geschwindigkeitsvektors ein Winkel benötigt. Diesen bestimmen wir nicht mit der Funktion *tan2*, da diese den falschen Winkelbereich liefert. Stattdessen haben wir eine eigene, auf *atan2* basierende Routine geschrieben.

```

1 double get_angle(const double & x, const double & y)
2 {
3     double at = atan2(y,x);
4     if(at >=0.)
5     {
6         return at;
7     }
8     else
9     {
10        return 6.28318530718 + at;
11    }
12 }
```

Abbildung 6: Die angepasste Funktion zur Winkelbestimmung

Die *atan2* Funktion aus C++ liefert einen Winkel im Bereich $(-\pi, \pi]$ während wir das Intervall $[0, 2\pi)$ benötigen.

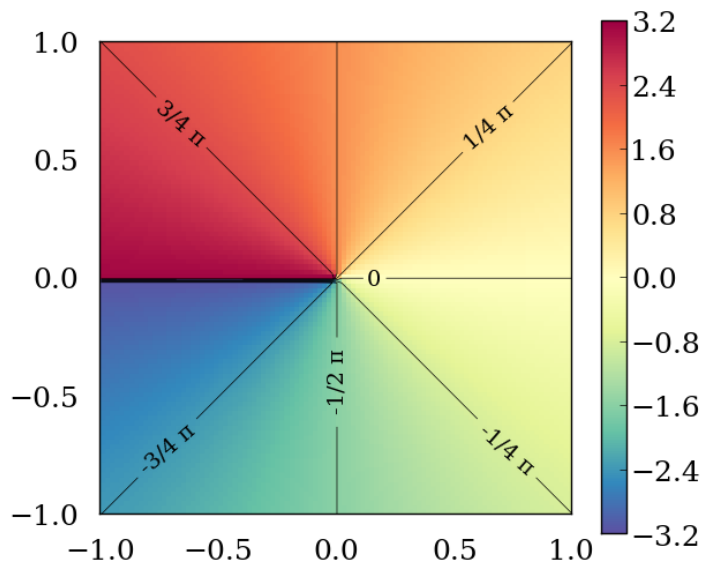


Abbildung 7: Die normale *atan2* Funktion aus C++ (<http://upload.cppreference.com/mwiki/images/9/91/math-atan2.png>)

Ein weiteres Problem war die Anfangsverteilung der Geschwindigkeiten. Wir nähern die Orbitgeschwindigkeit folgendermassen: Sei M_{\odot} die Masse des Galaxiezentrums, m die Masse des aktuellen Sterns mit $m \ll M_{\odot}$. Dann gilt für

den Orbit $F_{Zentrifugal} = F_{Gravitation}$, also $m \frac{v^2}{r} = G \cdot \frac{m \cdot M_{\odot}}{r^2}$, daher

$$v = \sqrt{\frac{G \cdot M_{\odot}}{r}}$$

```

1  double orbit_velocity(const double & radius, const unsigned int &
    mco)
2  {
3      return sqrt(G*MASS_OF_SUN*mco / radius);
4  }
```

Abbildung 8: Unsere Implementierung der Formel, dabei ist *mco* das Massenverhältnis zwischen Stern und Zentrum

3.6 Dateien

Boost erlaubt es, komplette abstrakte Datentypen zu laden und abzuspeichern. Dafür haben wir eine eigene Klasse erzeugt, mit dem Namen *gFile*. Diese Klasse händelt den Input/Output von Vektoren, die mit Sternen gefüllt sind.

```

1  gFile(string in_name, bool write_file)
2  {
3      name = in_name;
4      try
5      {
6          if(exists_test(name) && write_file)
7          {
8              string inp;
9              cout << "File " << name << " already exists, overwrite[y/n]:
              ";
10             cin >> inp;
11             if(inp.at(0) == 'n' || inp.at(0) == 'N')
12             {
13                 throw "File output error\n";
14             }
15         }
16     }
17     catch(string err)
18     {
19         cout << err;
20     }
21 }
```

Abbildung 9: Der Konstruktor der Klasse *gFile*. Es wird ein Dateiname gesetzt und überprüft, ob die Datei bereits existiert. Falls ja wird abgefragt, ob die Datei überschrieben werden soll.

Es sind zwei Methoden angelegt, die nun das Lesen und Schreiben erlauben, *save* und *load*.

```

1  vector<Star> load()                1  void save(vector<Star>& out_stars)
2  {                                  2  {
3      ifstream file{name};          3      std::ofstream file{name};
4      text_iarchive ia{file};        4      text_oarchive oa{file};
5      vector<Star> stars;            5      oa << out_stars;
6      ia >> stars;                   6  }
7      return stars;
8  }

```

Abbildung 10: Die Methoden *save* und *load* definieren einen Filestream und schieben dann den Vektor mit Sternen in die Datei / laden den Vektor wieder.

Zum Speichern der Sterne muss die Klasse *Star* noch mitteilen, welche Werte gespeichert werden sollen, dies passiert in einer eigenen Methode.

```

1  template<class Archive>
2  void serialize(Archive & ar, const unsigned int version)
3  {
4      ar & xpos;
5      ar & ypos;
6      ar & xvel;
7      ar & yvel;
8      ar & mass;
9  }

```

Abbildung 11: Die Serialisierung von *Star*

3.7 Plots

Wir werden in diesem Protokoll nicht auf die Details der Interaktion zwischen Gnuplot und C++ eingehen, diese sind teilweise aus dem Interface von Jeremy Conlin übernommen, dieses ist unter <https://code.google.com/archive/p/gnuplot-cpp/> zu finden. Wir wollen aber kurz unsere Plotroutine besprechen.

```

1 void export_plot(const std::vector<Star> & cluster, const double
    scale)
2 {
3     vector<double> x,y;
4     init_xy(x,y, cluster.size());
5     export_xy(cluster,x,y);
6
7     nop++;
8
9     string name = filename(nop);
10    cout << "File saved: " << name.c_str() << " ";
11
12
13    Gnuplot g1("points");
14    g1.savetopng(name.c_str());
15    g1.set_style("points").set_samples(300).set_xrange(- scale,
        scale).set_yrange(- scale, scale).plot_xy(x,y);
16    g1.reset_all();
17    if(nop%1000 == 0)
18    {
19        sleep(5);
20        g1.remove_tmpfiles();
21        g1.cmd("exit");
22    }
23 }

```

Abbildung 12: Die Plotroutine. Gut zu erkennen ist die Vereinfachung der Kommunikation mit Gnuplot durch das Interface.

Der Vektor *cluster* enthält Sterne und muss daher noch für Gnuplot konvertiert werden. Gnuplot akzeptiert nur Vektoren mit doubles, die Konvertierung ist simpel:

```

1 void export_xy(const std::vector<Star> & cluster, std::vector
    <double> & x, std::vector<double> & y)
2 {
3     unsigned int n = cluster.size();
4     for(unsigned int i=0; i<n; i++)
5     {
6         x[i] = cluster[i].xpos;
7         y[i] = cluster[i].ypos;
8     }
9 }

```

Abbildung 13: Speicherung der Position der Sterne in den Vektoren *x* und *y*.

Des Weiteren bietet Galaxia die Möglichkeit, viele Plotdateien zu einem Video zusammenzufassen. Dazu benutzen wir das Paket *ffmpeg* und benutzen es im Batch-Modus. Der entsprechende Befehl hierzu lautet *ffmpeg -i out/p%05d.png out/movie.mp4*. Über die *system* Schnittstelle lässt sich diese Befehl ausführen.

```

1 void make_video(bool delete_plotfiles)
2 {
3     try
4     {
5         cout << endl;
6         Gnuplot g1;
7         sleep(3);
8         g1.remove_tmpfiles();
9         cout << "Deleted temporary files" << endl;
10        if (system("ffmpeg -hide_banner -loglevel panic -i out/p%05d.
11                png out/movie.mp4 -vcodec mpeg4 -vb 128k -r 18") == 0)
12        {
13            cout << "Successfully created video, you can find it at out/
14                movie.mp4" << endl;
15            if(delete_plotfiles)
16            {
17                for(unsigned int i=1; i<=nop;i++)
18                {
19                    string name = filename(i) + ".png";
20                    if(remove(name.c_str()) != 0)
21                    {
22                        throw ("Unspecified file deletion error");
23                    }
24                    cout << "Deleted plot files" << endl;
25                    nop = 0;
26                }
27            }
28            else
29            {
30                throw ("Failed to create output video output");
31            }
32        }
33        catch(string er)
34        {
35            cout << er << endl;
36        }
37    }
38 }

```

Abbildung 14: Die gesamte Routine zur Videokonvertierung.

4 Ergebnisse

Wir haben drei Szenarien in Galaxia integriert, hier wollen wir kurz die Ergebnisse diskutieren.

4.1 Die Milchstraße

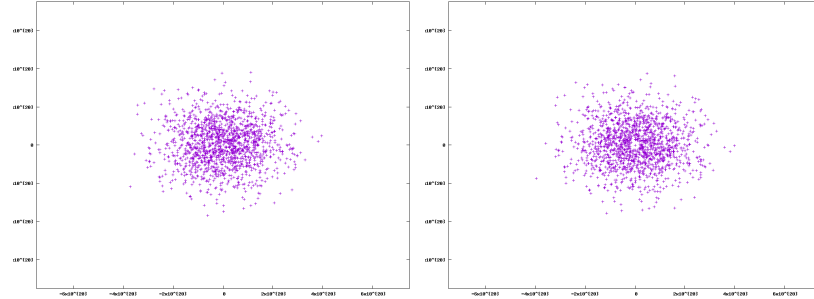


Abbildung 15: Die Entwicklung der Milchstraße in Galaxia.

In Galaxia ist eine Anfangsverteilung, die in Geschwindigkeits und Masseverteilung in etwa unserer Galaxie entspricht, eingebaut. Die Verteilung zeigt sich als zeitlich stabil, zufällige Anfangsverteilungen von Sternen sind also hinreichend als Näherung für reale Galaxien.

4.2 Die Kollision von Milchstraße und Andromeda

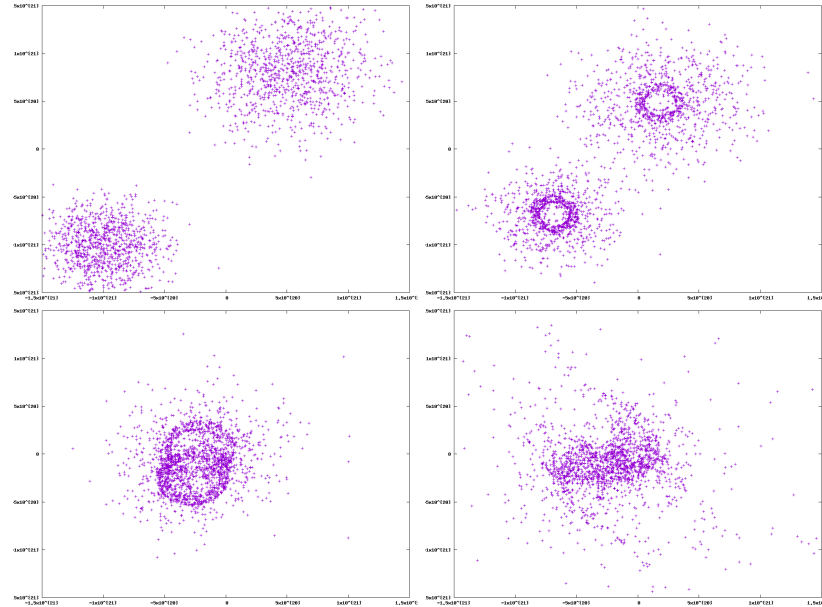


Abbildung 16: Die Kollision von Milchstraße und Andromeda in Galaxia.

Die in mehreren milliarden Jahren erwartete Kollision der Andromedagalaxie mit unserer Milchstraße lässt sich gut in Galaxia darstellen. Allerdings führt die große Schrittweite unserer Simulation dazu, dass die Sterne beim Orbit in der Galaxie stärker vom Galaxiezentrum weg streben, je näher sie sich am Zentrum befinden. Dies könnte durch eine feinere Simulationszeit behoben werden.

Allerdings ist die Entwicklung der Verteilung recht realistisch, es entsteht ein großer, zufällig verteilter Blob, während viele Sterne einfach in die Weite des Universums geschleudert werden.

4.3 Chaostheorie

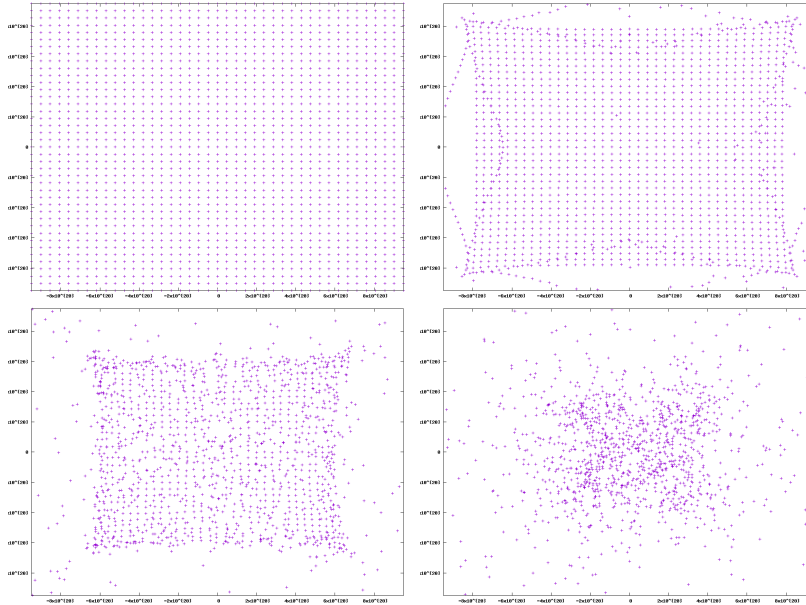


Abbildung 17: Die Entwicklung einer homogenen Verteilung in Galaxia.

Wir können auch eine homogene Verteilung von Sternen in Galaxia laden. Der zweite Hauptsatz der Thermodynamik legt fest, dass in einem abgeschlossenem System die Entropie immer nur zunehmen kann. Das bedeutet, übersetzt auf unser Szenario, dass die Symmetrie unserer homogenen Verteilung aufgrund der gegenseitigen Interaktion im Laufe der Zeit immer weiter abnehmen sollte.¹ Und genau das ist auch in unseren Ergebnissen zu beobachten: nach einigen Schritten ist die Symmetrie noch gut erhalten, während wir am Ende mit einer zufälligen Verteilung von Punkten dastehen.

¹Uns ist durchaus bewusst, dass die Definition der Entropie als „Maß für Unordnung“ in der Fachwelt nicht ganz unstrittig ist, aber sie scheint für die Interpretation unserer Ergebnisse angemessen.

5 Anhang

An dieser Stelle kann der gesamte Quellcode eingesehen werden, für Erklärungen verweisen wir auf die Dokumentation des Projektes. Diese ist, wie das Projekt selbst, unter github.com/M-Caspar/Galaxia zu finden.

5.1 main.cpp

```

1  #include "scenarios.h"
2
3  #include <iomanip>
4
5  using std::cout;
6  using std::endl;
7  using std::cin;
8
9  void intro_art()
10 {
11     cout << "          -----" << endl
12         << "      /  -----/  |  /  /  /  |  |  /  /  /  |" << endl
13         << "    /  /  /  /  /  |  /  /  /  /  |  |  /  /  /  /  |" << endl
14         << "  /  /  /  /  /  |  /  /  /  /  |  |  /  /  /  /  |" << endl
15         << "  \  \  \  \  \  |  \  \  \  \  |  |  \  \  \  \  |" << endl
16         << endl
17         << "Written by Maximilian Caspar and Johannes Esser" << endl
18         << "University of Wuppertal, 2016" << endl << endl;
19 }
20
21 void make_menu()
22 {
23     cout << "Hello and welcome to Galaxia. Here are your options:"
24         << endl << endl;
25     cout << "  a The Milkyway: Watch our home galaxy in all it\'s
26         beauty." << endl << endl;
27     cout << "  b Andromeda: See our neighbour galaxy smash into us
28         ." << endl << endl;
29     cout << "  c Uniform distribution: Do uniform distributions
30         turn more chaotic over time?" << endl << endl;
31     cout << "  x Exit Galaxia" << endl << endl;
32 }
33
34 int main(int argc, char const *argv[])
35 {
36     std::cout << std::fixed << std::setprecision (3);
37     intro_art();
38     make_menu();
39     cout << "Please input your choice: ";
40     char choice;
41
42     do
43     {
44         cin >> choice;
45         switch(choice)
46         {
47             case 'a':
48                 cout << "Do you want to create a video (this will delete
49                     plot files)? (y/n) ";
50                 cin >> choice;
51                 if(choice == 'y')
52                 {

```



```

48         scenario_milkyway(true);
49     }
50     else
51     {
52         scenario_milkyway(false);
53     }
54     break;
55
56     case 'b':
57     cout << "Do you want to create a video (this will delete
58         plot files)? (y/n) ";
59     cin >> choice;
60     if(choice == 'y')
61     {
62         scenario_andromeda(true);
63     }
64     else
65     {
66         scenario_andromeda(false);
67     }
68     break;
69
70     case 'c':
71     cout << "Do you want to create a video (this will delete
72         plot files)? (y/n) ";
73     cin >> choice;
74     if(choice == 'y')
75     {
76         scenario_square(true);
77     }
78     else
79     {
80         scenario_square(false);
81     }
82     break;
83
84     case 'x':
85     break;
86
87     default :
88     choice = 'u';
89     cout << "You picked an invalid option, please choose again
90         : ";
91     break;
92 }
93 while(choice == 'u');
94
95 cout << "Live long and prosper!" << endl;
96
97 return 0;
98 }

```

5.2 stars.h

```

1  #pragma once
2
3  #include <vector>
4  #include <boost/archive/text_oarchive.hpp>
5  #include <boost/archive/text_iarchive.hpp>
6  #include <boost/serialization/vector.hpp>
7  using namespace boost::archive;
8

```

```

9  class Star
10 {
11 public:
12
13     Star(double sx,double sy, double vx, double vy, double m)
14     {
15         mass = m;
16         xpos = sx;
17         ypos = sy;
18         xvel = vx;
19         yvel = vy;
20         xfor = 0.0;
21         yfor = 0.0;
22     };
23
24     Star(){};
25
26     void move(double dt)
27     {
28         xpos = xpos + xvel*dt;
29         ypos = ypos + yvel*dt;
30         xvel = xvel + (xfor / mass)*dt;
31         yvel = yvel + (yfor / mass)*dt;
32         xfor = 0.0;
33         yfor = 0.0;
34     }
35     double xpos,
36         ypos;
37     double xvel,
38         yvel;
39     double xfor,
40         yfor;
41     double mass;
42     friend class boost::serialization::access;
43
44     template<class Archive>
45     void serialize(Archive & ar, const unsigned int version)
46     {
47         ar & xpos;
48         ar & ypos;
49         ar & xvel;
50         ar & yvel;
51         ar & mass;
52     }
53 };

```

5.3 gravity.h

```

1  #pragma once
2  #include "stars.h"
3  #include <cmath>
4  #include <random>
5  #include <chrono>
6  #include <omp.h>
7  #include <iostream>
8
9  const double G = 6.674E-11;
10 const double MASS_OF_SUN = 1.98892E30;
11 const double LIGTHYEAR = 9.46073E15;
12
13 double get_angle(const double & x, const double & y)
14 {

```

```

15     double at = atan2(y,x);
16     if(at >=0.)
17     {
18         return at;
19     }
20     else
21     {
22         return 6.28318530718 + at;
23     }
24 }
25
26 double orbit_velocity(const double & radius)
27 {
28     double x = radius/( LIGTHYEAR*10000);
29     return 1100*x/pow(1+x*x,0.75);
30 }
31 }
32
33 double orbit_velocity(const double & radius, const unsigned int
    & mco)
34 {
35     return sqrt(G*MASS_OF_SUN*mco / radius);
36 }
37
38 void make_galaxy(std::vector<Star> & Elements, const double &
    radius, const unsigned int & nos, const double & sx, const
    double & sy, const double & vx, const double & vy, const
    unsigned int & mco)
39 {
40
41     unsigned seed = std::chrono::system_clock::now().
        time_since_epoch().count();
42     std::default_random_engine generator (seed);
43     std::normal_distribution<double> distribution(0.0, radius/2);
44
45     for (unsigned int i = 0; i < nos; i++)
46     {
47         double x = 0;
48         double y = 0;
49         double rad = 0;
50         while(rad < 0.05*radius)
51         {
52             x = distribution(generator);
53             y = distribution(generator);
54             rad = sqrt(x*x + y*y);
55         }
56         double alpha = get_angle(x , y);
57         double v = orbit_velocity(rad, mco);
58         Star temp(x + sx ,y + sy,v*sin(alpha) + vx,-v*cos(alpha) +
            vy,MASS_OF_SUN);
59         Elements.push_back(temp);
60     }
61     Star center(sx, sy, vx, vy, MASS_OF_SUN*mco);
62     Elements.push_back(center);
63 }
64
65 void accelerate(std::vector<Star> & cluster, double dt)
66 {
67     unsigned int n = cluster.size();
68
69     #pragma omp parallel for
70     for(unsigned int i=0; i<n-1;i++)

```

```

71  {
72
73      for(unsigned int j=i+1;j<n;j++)
74      {
75          double xvec = cluster[i].xpos - cluster[j].xpos;
76          double yvec = cluster[i].ypos - cluster[j].ypos;
77          double r = sqrt(xvec*xvec + yvec*yvec);
78          double con = G*cluster[i].mass*cluster[j].mass;
79          cluster[i].xfor -= con*xvec*pow(r,-3);
80          cluster[i].yfor -= con*yvec*pow(r,-3);
81          cluster[j].xfor += con*xvec*pow(r,-3);
82          cluster[j].yfor += con*yvec*pow(r,-3);
83      }
84  }
85  #pragma omp parallel for
86  for(unsigned int i = 0; i < n ;i++)
87  {
88      cluster[i].move(dt);
89  }
90  }

```

5.4 plot.h

```

1  #pragma once
2
3  #include "stars.h"
4  #include "gnuplot_i.hpp"
5
6  #include <stdio.h>
7  #include <iostream>
8  #include <string>
9  #include <ctime>
10 #include <assert.h>
11
12 using std::cout;
13 using std::endl;
14 using std::string;
15 using std::vector;
16 using std::cin;
17
18 unsigned int nop = 0;
19
20 unsigned int number_of_digits (unsigned int i)
21 {
22     return i > 0 ? (int) log10 ((double) i) + 1 : 1;
23 }
24
25 string filename(unsigned int n)
26 {
27     string name = "out/p";
28
29     for(unsigned int i = 1; i<= 5; i++)
30     {
31         if(i <= 5 - number_of_digits(n))
32         {
33             name += "0";
34         }
35         else
36         {
37             break;
38         }
39     }

```

```

40     name += std::to_string(n);
41     return name;
42 }
43
44 void make_video(bool delete_plotfiles)
45 {
46     try
47     {
48         cout << endl;
49         Gnuplot g1;
50         sleep(3);
51         g1.remove_tmpfiles();
52         cout << "Deleted temporary files" << endl;
53         if (system("ffmpeg -hide_banner -loglevel panic -i out/p%05d
                    .png out/movie.mp4 -vcodec mpeg4 -vb 128k -r 18") == 0)
54         {
55             cout << "Successfully created video, you can find it at
                    out/movie.mp4" << endl;
56             if(delete_plotfiles)
57             {
58                 for(unsigned int i=1; i<=nop;i++)
59                 {
60                     string name = filename(i) + ".png";
61                     if(remove(name.c_str()) != 0)
62                     {
63                         throw ("Unspecified file deletion error");
64                     }
65                 }
66                 cout << "Deleted plot files" << endl;
67                 nop = 0;
68             }
69         }
70         else
71         {
72             throw ("Failed to create output video output");
73         }
74     }
75     catch(string er)
76     {
77         cout << er << endl;
78     }
79 }
80
81
82 void wait_for_key ()
83 {
84     #if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) ||
        defined(__TOS_WIN__)
85         cout << endl << "Press any key to continue..." << endl;
86
87         FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
88         _getch();
89     #elif defined(unix) || defined(__unix) || defined(__unix__) ||
        defined(__APPLE__)
90         cout << endl << "Press ENTER to continue..." << endl;
91
92         cin.clear();
93         cin.ignore(cin.rdbuf()->in_avail());
94         cin.get();
95     #endif
96     return;
97 }

```

```

98
99 void export_xy(const std::vector<Star> & cluster, std::vector<
    double> & x, std::vector<double> & y)
100 {
101     unsigned int n = cluster.size();
102     for(unsigned int i=0; i<n; i++)
103     {
104         x[i] = cluster[i].xpos;
105         y[i] = cluster[i].ypos;
106     }
107 }
108
109 void init_xy(std::vector<double> & x, std::vector<double> & y,
    unsigned int n)
110 {
111     x.reserve(n);
112     y.reserve(n);
113     for (int i = 0; i < n; ++i)
114     {
115         x.push_back(0);
116         y.push_back(0);
117     }
118 }
119
120 void export_plot(const std::vector<Star> & cluster, const double
    scale)
121 {
122     vector<double> x,y;
123     init_xy(x,y, cluster.size());
124     export_xy(cluster,x,y);
125
126     nop++;
127
128     string name = filename(nop);
129     cout << "File saved: " << name.c_str() << " ";
130
131
132     Gnuplot g1("points");
133     g1.savetopng(name.c_str());
134     g1.set_style("points").set_samples(300).set_xrange(- scale,
        scale).set_yrange(- scale, scale).plot_xy(x,y);
135     g1.reset_all();
136     if(nop%1000 == 0)
137     {
138         sleep(5);
139         g1.remove_tmpfiles();
140         g1.cmd("exit");
141     }
142 }

```

5.5 fileio.h

```

1  #pragma once
2
3  #include "stars.h"
4
5  #include <iostream>
6  #include <fstream>
7  #include <string>
8  #include <vector>
9  #include <boost/archive/text_oarchive.hpp>
10 #include <boost/archive/text_iarchive.hpp>

```

```

11 #include <boost/serialization/vector.hpp>
12
13 using namespace std;
14 using namespace boost::archive;
15
16 class gFile
17 {
18 public:
19
20     gFile(string in_name, bool write_file)
21     {
22         name = in_name;
23         try
24         {
25             if(exists_test(name) && write_file)
26             {
27                 string inp;
28                 cout << "File " << name << " already exists, overwrite[y
29                     /n]: ";
30                 cin >> inp;
31                 if(inp.at(0) == 'n' || inp.at(0) == 'N')
32                 {
33                     throw "File output error\n";
34                 }
35             }
36         }
37         catch(string err)
38         {
39             cout << err;
40         }
41     }
42
43     vector<Star> load()
44     {
45         ifstream file{name};
46         text_iarchive ia{file};
47         vector<Star> stars;
48         ia >> stars;
49         return stars;
50     }
51
52     void save(vector<Star>& out_stars)
53     {
54         std::ofstream file{name};
55         text_oarchive oa{file};
56
57         oa << out_stars;
58     }
59
60 private:
61
62     string name;
63
64     inline bool exists_test (const string& in_name)
65     {
66         ifstream f(in_name.c_str());
67         return f.good();
68     }
69
70 };
71

```

5.6 scenarios.h

```

1  #pragma once
2
3  #include "stars.h"
4  #include "plot.h"
5  #include "gravity.h"
6  #include "fileio.h"
7
8  using std::cout;
9  using std::endl;
10 using std::vector;
11
12 void scenario_andromeda(bool video)
13 {
14     cout << std::endl;
15     vector<Star> elements;
16     gFile g("examples/scenarios/Collision1.galaxia", false);
17     elements = g.load();
18     const double time_interval = 1E15;
19
20     for(unsigned int i = 1; i<=1000;i++)
21     {
22         double ctime1 = omp_get_wtime();
23         cout << "Simulation step " << i << "\t";
24         usleep(4);
25         accelerate(elements, time_interval);
26         export_plot(elements, 1.5E21);
27         double ctime2 = omp_get_wtime();
28         cout << "Time: " << ctime2 - ctime1 << " s" << endl;
29     }
30 }
31
32 if(video)
33 {
34     make_video(true);
35 }
36 else
37 {
38     Gnuplot g1;
39     sleep(3);
40     g1.remove_tmpfiles();
41     cout << "Deleted temporary files" << endl;
42 }
43
44 }
45
46 void scenario_milkyway(bool video)
47 {
48     cout << endl;
49     vector<Star> elements;
50     gFile g("examples/scenarios/Milkyway.galaxia", false);
51     elements = g.load();
52     const double time_interval = 6E13;
53
54     for(unsigned int i = 1; i<=1000;i++)
55     {
56         double ctime1 = omp_get_wtime();
57         cout << "Simulation step " << i << "\t";
58         usleep(4);
59         accelerate(elements, time_interval);
60         export_plot(elements, 7.5E20);

```



```

61     double ctime2 = omp_get_wtime();
62     cout << "Time: " << ctime2 - ctime1 << " s" << endl;
63
64 }
65
66 if(video)
67 {
68     make_video(true);
69 }
70 else
71 {
72     Gnuplot g1;
73     sleep(3);
74     g1.remove_tmpfiles();
75     cout << "Deleted temporary files" << endl;
76 }
77
78 }
79
80 void scenario_square(bool video)
81 {
82     cout << endl;
83     vector<Star> elements;
84     gFile g("examples/scenarios/Square.galaxia", false);
85     elements = g.load();
86     const double time_interval = 1.2E13;
87
88     for(unsigned int i = 1; i<=1000;i++)
89     {
90         double ctime1 = omp_get_wtime();
91         cout << "Simulation step " << i << "\t";
92         usleep(4);
93         accelerate(elements, time_interval);
94         export_plot(elements, 9.5E20);
95         double ctime2 = omp_get_wtime();
96         cout << "Time: " << ctime2 - ctime1 << " s" << endl;
97     }
98 }
99
100 if(video)
101 {
102     make_video(true);
103 }
104 else
105 {
106     Gnuplot g1;
107     sleep(3);
108     g1.remove_tmpfiles();
109     cout << "Deleted temporary files" << endl;
110 }
111 }

```

