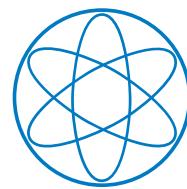


TECHNISCHE UNIVERSITÄT MÜNCHEN



FAKULTÄT FÜR PHYSIK

**Simulation eines Spinpräzessionsexperimentes
für Neutronen**

**Simulation of a spin precession experiment for
neutrons**

Bachelorarbeit

von

Roman Thiele

im Studiengang

Bachelor of Education Physik / Mathematik

Datum der Abgabe: 25. Juli 2011

Themensteller und Betreuer: Prof. Dr. Peter Fierlinger

Inhaltsverzeichnis

1. Einleitung	5
1.1. Motivation	5
1.2. Ziel der Arbeit	6
1.3. Aufbau der Arbeit	6
2. Beschreibung des Experimentes	8
2.1. Physikalische Grundlagen	8
2.2. Ultrakalte Neutronen	11
2.3. Ramseys Methode	12
2.3.1. Separierte oszillierende Felder	12
2.3.2. Verwendung eines Co-Magnetometers	13
2.4. Systematische Fehler	14
2.4.1. Geometrische Phase durch Gradient und $v \times E$ -Effekt	14
2.4.2. Unterschiedlicher Schwerpunkt von Neutronen und Co-Magnetometer im Schwerefeld der Erde	15
2.4.3. Weitere systematische Fehler	16
3. Entwicklung der Simulation	17
3.1. Flugbahnen der Teilchen	17
3.1.1. Anfangsgeschwindigkeit	17
3.1.2. Berechnung der Flugbahn	19
3.1.2.1. Numerische Integration der Bewegungsgleichung	19
3.1.2.2. Analytische Lösung der Bewegungsgleichung	22
3.1.3. Spekulare Reflexion	23
3.1.4. Diffuse Streuung	24
3.2. Magnetfeld	24
3.2.1. Homogenes Magnetfeld	26
3.2.2. Magnetfeld mit Gradient	26

Inhaltsverzeichnis

3.2.3. Dipolfeld	26
3.2.4. Magnetfeld durch Laserstrahl	26
3.2.5. Feldkarten	27
3.3. Lösung der Bloch-Gleichung	28
3.4. Methoden	29
3.4.1. Integration von Differentialgleichungen	29
3.4.2. Finden von Nullstellen	29
3.4.3. Binäre Suche	30
3.4.4. Zufallszahlen	31
3.5. Eingabe und Ausgabe	31
3.5.1. Parameterdatei	31
3.5.2. Felddatei	31
3.5.3. Ausgabedatei	33
4. Testfälle der Simulation	34
4.1. Energieerhaltung	34
4.2. Neutronen im homogenen Magnetfeld	34
4.3. Quecksilber im B-Feld mit Gradient und E-Feld	36
A. Kommentierter Quelltext	38
A.1. Hauptfunktion	39
A.2. Geometrie	42
A.3. Magnetfeld	47
A.4. Flugbahn	50
A.5. Integration der Bloch-Gleichung	59
A.6. Hilfsfunktionen	60
Literaturverzeichnis	73

1. Einleitung

1.1. Motivation

Eine grundlegende Frage der modernen Physik ist die nach der Entstehung des Universums. Auch wenn die Urknall-Theorie durch die exakte Vermessung der kosmischen Hintergrundstrahlung und der Rot-Verschiebung des Lichtes ferner Galaxien gut belegt und weithin anerkannt ist, bleibt dabei ein nicht unerhebliches Problem offen: Nach dem Standardmodell der Teilchenphysik dürfte es im Universum keine Materie mehr geben.

Der Grund dafür ist, dass bei allen bekannten Prozessen, bei denen Materie entsteht gleichzeitig auch Antimaterie entsteht [1]. Ein Beispiel für einen solchen Prozess ist die Paarbildung eines Teilchens der Sorte x aus einem Photon:

$$\gamma \longrightarrow x + \bar{x} \tag{1.1}$$

Dabei entsteht gleichzeitig auch das Antiteilchen \bar{x} . Eine weitere grundlegende Eigenschaft der Physik ist die sogenannte Zeitumkehrsymmetrie. Diese besagt, dass jeder Prozess auch in umgekehrter Richtung ablaufen kann. Im vorherigen Beispiel wäre dies die Annihilation, bei der wieder ein Photon entsteht:

$$x + \bar{x} \longrightarrow \gamma \tag{1.2}$$

Im frühen, dichten Universum muss es ein Gleichgewicht dieser beiden Prozesse gegeben haben. Im Lauf der Ausdehnung des Universums wird dieses Gleichgewicht dann allerdings gestört, da die Photonen (die sich im Gegensatz zu massebehafteten Teilchen mit der Lichtgeschwindigkeit bewegen) sich, sobald sie nicht mehr laufend mit der Materie stoßen, von dieser ablösen können. Insgesamt dürfte bei diesem Prozess nur sehr wenig Materie und gleich wenig Antimaterie zurückbleiben [1]. Dies ist aber nachweislich nicht der Fall, das sichtbare Universum besteht beinahe ausschließlich aus Materie.

1. Einleitung

Eine Möglichkeit, die unterschiedliche Verteilung von Materie und Antimaterie zu erklären ist die CP-Verletzung die, zum Beispiel beim Zerfall des neutralen Kaons, im Standardmodell bereits enthalten ist. Allerdings ist sie dort nicht groß genug, um die Beobachtungen zu erklären.

Deshalb müssen neue Theorien gesucht werden, die das Standardmodell erweitern. Eine Möglichkeit zur Überprüfung solcher Theorien ist die Kollision von Teichen mit hoher Energie (zum Beispiel mit bis zu 14 TeV am CERN) um die Bedingungen kurz nach dem Urknall zu reproduzieren. Eine andere vielversprechende Möglichkeit sind hochpräzise Experimente bei niedrigen Energien, mit denen sogar in Bereiche vorgedrungen werden kann, die am CERN nicht erreicht werden können [1].

In dieser Arbeit wird auf die Suche nach dem elektrischen Dipolmoment des Neutrons („nEDM“) eingegangen. Nach dem Standardmodell sollte dieses kleiner als 10^{-32} ecm sein [2], die aktuelle experimentelle Schranke ist $d < 2.9 \cdot 10^{-26}$ ecm [3, 4]. Wenn man die Genauigkeit der Messung auf 10^{-28} ecm erhöht, kommt man in einen Bereich, in dem von der Theorie der Supersymmetrie bereits ein EDM vorhergesagt wird [1]. Sowohl die Bestätigung als auch die Widerlegung dieser Theorie wäre ein großer Erfolg.

Da die aktuelle Schranke vor allem durch systematische Effekte bedingt ist, ist es für ein verbessertes Experiment vor allem unerlässlich, diese Effekte simulieren zu können. Diese Arbeit legt dafür den Grundstein.

1.2. Ziel der Arbeit

Ziel dieser Arbeit ist, ein Werkzeug zur Simulation eines Spinpräzessionsexperimentes mit Neutronen zu entwickeln, mit dem die verschiedenen systematischen Effekte, die bei einem solchen Experiment zwingend auftreten, berechnet werden können. Mit Hilfe dieser Ergebnisse kann dann die Genauigkeit der Messung verbessert werden.

1.3. Aufbau der Arbeit

Im weiteren Verlauf der Arbeit wird zunächst in Kapitel 2 auf das simulierte Experiment und die relevanten Effekte eingegangen. Das folgende Kapitel 3 beschreibt einige Überlegungen zur Entwicklung der Simulation und deren Realisierung. Anschließend werden

1. Einleitung

in Kapitel 4 einige einfache Probleme mit Hilfe des Programmes betrachtet. Im Anhang ist der Quelltext der Simulation beigegeben.

2. Beschreibung des Experimentes

Bei dem simulierten Experiment handelt es sich um ein Speicherexperiment für ultrakalte Neutronen, ähnlich wie dem von Baker et al. [3], das am ILL¹ die aktuell anerkannte obere Grenze für das elektrische Dipolmoment des Neutrons [4] lieferte. Die vom Forschungsreaktor FRM2 erzeugten Neutronen werden in der direkten Nähe des Reaktorkernes in eine Quelle für ultrakalte Neutronen geleitet und dort auf eine Temperatur von wenigen Millikelvin abgekühlt. Über einen Neutronenleiter, in dem die Neutronen durch ein starkes Magnetfeld polarisiert werden, gelangen die Neutronen in zwei Zylinder, in denen jeweils ein entgegengesetzt gleich großes elektrisches Feld von ca. $10 \frac{\text{kV}}{\text{cm}}$ und ein homogenes Magnetfeld von ca. $1 \mu\text{T}$ vorliegt [5]. Außer den Neutronen ist in den Zellen noch gasförmiges ^{199}Hg enthalten, das als Co-Magnetometer (siehe 2.3.2) verwendet wird. Zwei weitere Zellen ohne elektrisches Feld und ohne Neutronen (aber mit ^{199}Hg als Magnetometergas) dienen als Referenz, um externe Einflüsse auf der Experiment zu untersuchen. Der innere Teil des Experimentes wird in Abbildung 2.1 gezeigt. Im Folgenden wird auf die für die Simulation relevanten physikalischen Grundlagen des Experimentes eingegangen.

2.1. Physikalische Grundlagen

Die Interaktion eines Neutrons mit einem elektromagnetischen Feld wird nach Ref. [2] beschrieben durch den Hamiltonoperator

$$\hat{H} = -(\mu\mathbf{B} + d\mathbf{E}) \cdot \frac{\hat{\mathbf{S}}}{S} . \quad (2.1)$$

Dabei ist μ das bekannte magnetische Moment des Neutrons und d das gesuchte elektrische Dipolmoment. Die Lösung der zeitabhängigen Schrödinger-Gleichung für ein freies

¹Institut Laue-Langevin in Grenoble

2. Beschreibung des Experimentes

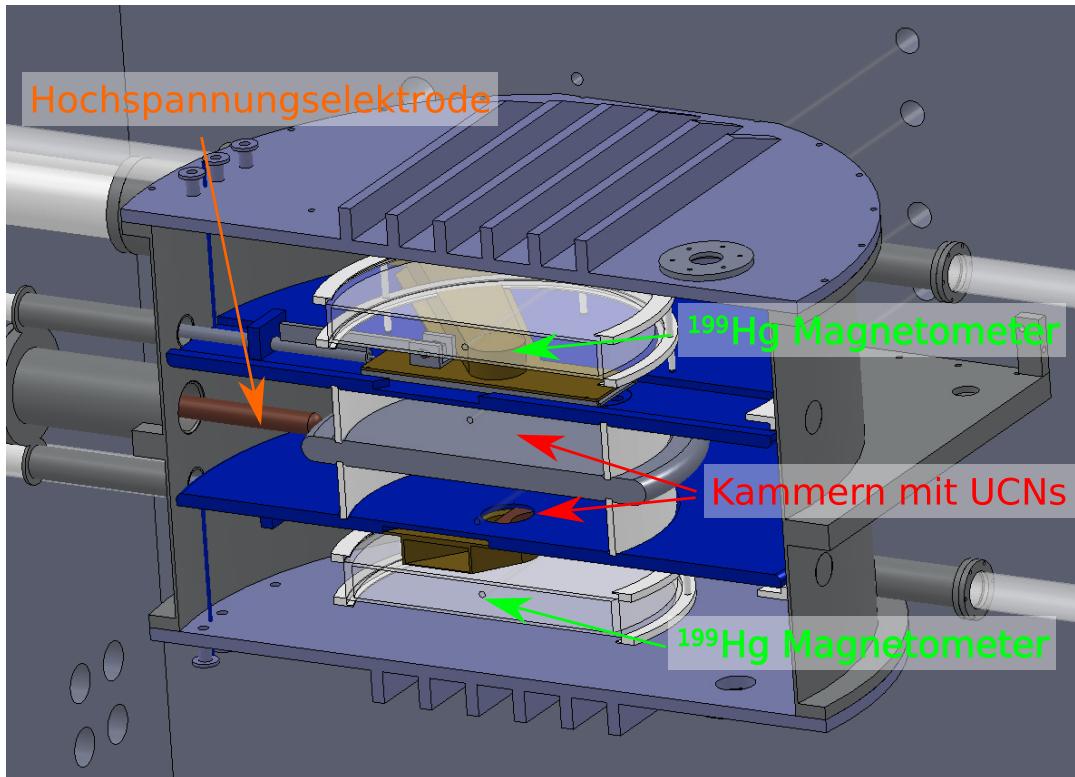


Abbildung 2.1.: Innerer Teil des geplanten nEDM Experiments an der TU München.
In der Zeichnung markiert sind die vier Kammern und die Zuführung
der Hochspannung für das elektrische Feld. Zeichnung von P. Fierlinger,
Beschriftung bearbeitet von R. Thiele.

2. Beschreibung des Experimentes

Teilchen mit $\mathbf{B} = B\hat{\mathbf{e}}_z$ und $\mathbf{E} = E\hat{\mathbf{e}}_z$ (wie zum Beispiel in Ref. [6]) liefert eine Präzession des Spins um die z-Achse. Da es sich dabei um eine stationäre Lösung handelt, kann man die Lösung auch aus der stationären Schrödinger-Gleichung erhalten:

$$\hat{H} |\Psi\rangle = \hbar\omega_L |\Psi\rangle . \quad (2.2)$$

Dabei ergeben sich die Larmor-Frequenzen

$$\omega_L = -\frac{\mu B \pm dE}{\hbar S} . \quad (2.3)$$

Die beiden verschiedenen Vorzeichen von dE ergeben sich durch $\mathbf{B} \uparrow\downarrow \mathbf{E}$ (positiv) und $\mathbf{B} \uparrow\downarrow \mathbf{E}$ (negativ). In einem Experiment kann man also, wenn es ein elektrisches Dipolmoment gibt, eine Verschiebung in der Larmor-Frequenz feststellen, die nur auftritt, wenn ein elektrisches Feld angelegt ist und deren Richtung sich ändert, wenn das Feld umgepolt wird. Deshalb bieten Spinpräzessionsexperimente eine gut nachvollziehbare Möglichkeit, das elektrische Dipolmoment des Neutrons zu messen.

Da in einem realen Spinpräzessionsexperiment die Felder nicht zeitlich konstant und homogen sind, muss eine allgemeine Beschreibung für das Verhalten der Neutronen gefunden werden. Dafür betrachtet man den Erwartungswert \mathbf{P} des Spins und lässt das elektrische Dipolmoment zunächst außen vor. Durch Lösung der zeitabhängigen Schrödinger-Gleichung (wie in Ref. [6] ausgeführt) erhält man die Bloch-Gleichung, die die Präzession des Polarisationsvektors \mathbf{P} um \mathbf{B} beschreibt:

$$\dot{\mathbf{P}} = -\gamma \mathbf{B} \times \mathbf{P} . \quad (2.4)$$

Dabei ist γ das gyromagnetische Verhältnis mit

$$\gamma = \frac{\mu}{\hbar S} \quad \text{und} \quad (2.5)$$

$$\omega_L = -\gamma B . \quad (2.6)$$

Für Neutronen ergibt sich mit $S = \frac{1}{2}$ und $\mu = -1.9130427\mu_N$ (Ref. [2], S. 4) für das gyromagnetische Verhältnis als Frequenz ausgedrückt:

$$\frac{\gamma}{2\pi} = -29.1647 \frac{\text{Hz}}{\mu\text{T}} . \quad (2.7)$$

2.2. Ultrakalte Neutronen

Während bei den ersten Experimenten zum elektrischen Dipolmoment von Neutronen (zum Beispiel von Smith et al. [7]) noch mit Teilchenstrahlen gearbeitet wurde, sind inzwischen Speicherexperimente mit Neutronen üblich, die bessere Ergebnisse liefern. Möglich werden diese Speicherexperimente durch die Verwendung von ultrakalten Neutronen, deren Temperatur unter 5 mK liegt und die entsprechend eine Geschwindigkeit vom ca. $7 \frac{\text{m}}{\text{s}}$ bzw. eine kinetische Energie von unter 250 neV haben. Die de-Broglie-Wellenlänge für solche Teilchen ist dann

$$\lambda_D = \frac{hc}{\sqrt{2mc^2 \cdot E_{kin}}} > 57 \text{ nm} . \quad (2.8)$$

Da übliche Bindungslängen in Festkörpern im Bereich von einigen Ångström liegen, interagiert ein ultrakaltes Neutron (UCN) nicht mit den einzelnen Nukleonen im Festkörper, sondern mit einem effektiven Potential U_F das Fermi erstmals eingeführt hat [2]. Dieses hat für eine ebene Oberfläche die Form eines Stufenpotentials mit einer Höhe im Bereich von einigen 100 neV und kann deshalb UCNs, deren Energie in diesem Bereich liegt, total reflektieren. Deshalb kann für die Bewegung der Neutronen in der Kammer die klassische Newtonsche Dynamik angewendet werden. Außerdem wird es dadurch möglich, Neutronen in einem entsprechend beschaffenen Gefäß zu speichern.

Ein wichtiger Vorteil von Speicherexperimenten ist die deutlich längere Aufenthaltszeit, die Teilchen in der Kammer haben können. Dadurch ist der angesammelte Phasenunterschied (siehe 2.3) durch das EDM größer und kann leichter gemessen werden. Während man in einem Speicherexperiment ohne Probleme 150 s lang messen kann, bräuchte man beim thermischen Neutronen mit 500 K wie in Ref. [7] eine ca. 500 km lange Kammer. Auf kleinerem Raum lassen sich auch die Felder deutlich einfacher und genauer regulieren.

Ein Nachteil von Strahlexperimenten ist der sogenannte $\mathbf{v} \times \mathbf{E}$ -Effekt. Da die Neutronen sich durch das elektrische Feld bewegen, entsteht nach der Relativitätstheorie im Ruhesystem der Neutronen ein Magnetfeld $\mathbf{B}_m \propto \mathbf{v} \times \mathbf{E}$ welches die Messergebnisse beeinflusst. Bei Speicherexperimenten verschwindet dieser Effekt im zeitlichen Mittel, da die Geschwindigkeiten isotrop verteilt sind. Allerdings entstehen in Verbindung mit inhomogenen Feldern sogenannte geometrische Phasen, die im Abschnitt 2.4.1 behandelt werden.

2. Beschreibung des Experimentes

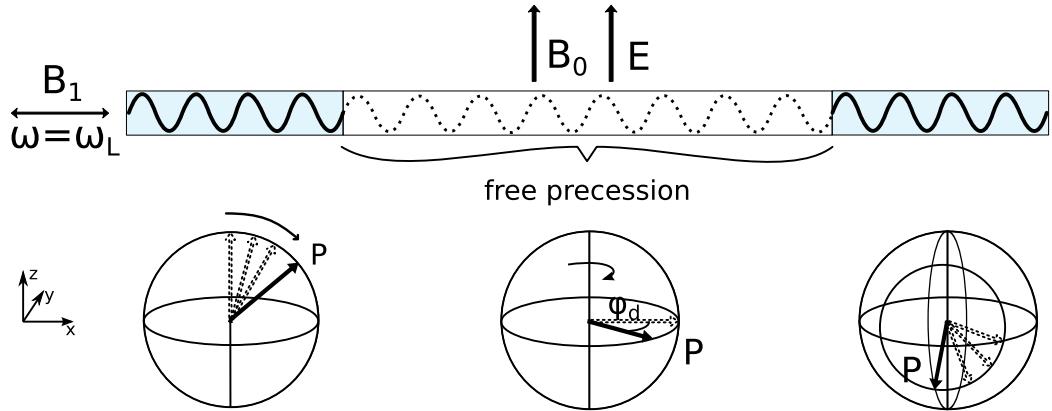


Abbildung 2.2.: Schematische Darstellung von Ramseys Methode. Dabei wird der zeitliche Verlauf des Experimentes von links nach rechts gezeigt, zuerst der erste \mathbf{B}_1 -Puls, dann die freie Präzession und ganz rechts der zweite \mathbf{B}_1 -Puls, bei dem der Polarisationsvektor \mathbf{P} aufgrund eines Phasenunterschiedes nicht exakt auf die z -Achse zurück gestellt wird. Bild von W. Feldmeier [6].

2.3. Ramsey's Methode

2.3.1. Separierte oszillierende Felder

Der grundlegende Aufbau des Experimentes folgt Ramseys Methode separierter oszillierender Felder (Ref. [8] und [9], S. 194ff.) für die er 1989 den Nobelpreis erhielt [10]. Der wesentliche Unterschied zur früheren Methode (für die Rabi 1944 den Nobelpreis erhielt) ist dabei, dass das Feld, das den Übergang zwischen den durch das Magnetfeld aufgespaltenen Zuständen auslöst, nicht während der gesamten Dauer des Experimentes, sondern nur an zwei Punkten angelegt wird. Wie Ramsey bereits 1950 zeigte [8], können auch so sämtliche Informationen über die Resonanzfrequenz des Übergangs ermittelt werden. Während Ramsey für sein Experiment Teilchenstrahlen verwendete und die oszillierenden Felder räumlich trennte, werden die Felder bei modernen Speicherexperimenten zeitlich getrennt, man spricht von Zeit-Interferometrie. Im Folgenden wird anhand von Abbildung 2.2 und auf Grundlage Ramseys ursprünglicher Arbeit die Funktionsweise klassisch über die Präzession des Polarisationsvektors beschrieben.

Zu Beginn des Experimentes werden parallel zum \mathbf{B}_0 -Feld polarisierte ultrakalte Neutronen in eine Kammer gegeben. Dort werden sie einem \mathbf{B}_1 -Puls ausgesetzt, dessen Frequenz die bekannte Larmor-Frequenz für Neutronen (ohne EDM) ist. Bei passender

2. Beschreibung des Experimentes

Dauer dieses Pulses wird dabei der Polarisationsvektor um 90° gedreht sodass er nun in der Ebene senkrecht zum \mathbf{B}_0 -Feld präzidiert. Nun wird das \mathbf{B}_1 -Feld abgeschaltet, der zugehörige Frequenzgenerator läuft aber weiter. In dieser Zeit, in der nur das \mathbf{B}_0 -Feld und das elektrische Feld anliegen, präzidiert der Polarisationsvektor mit der Larmor-Frequenz die allerdings, falls ein EDM vorliegt, wie in 2.1 beschrieben leicht verschoben ist. Deshalb sammelt sich ein leichter Phasenunterschied zwischen dem Frequenzgenerator des \mathbf{B}_1 -Feldes und der Phase des Polarisationsvektors an. Wenn nun nach einiger Zeit (in Speicherexperimenten sind 150 s möglich) ein zweiter \mathbf{B}_1 -Puls identisch zum ersten angelegt wird, führt die angesammelte Phase dazu, dass der Polarisationsvektor nicht genau um 90° weiter gekippt wird, also nach dem Experiment noch immer einen Winkel zum \mathbf{B}_0 -Feld hat. Dieser Winkel kann dann gemessen werden, zum Beispiel wie in Ref. [3] durch eine magnetisierte Folie, die die Neutronen in einen Spin-Eigenzustand ($|\uparrow\rangle$ oder $|\downarrow\rangle$) zwingt und nur einen der Zustände passieren lässt. Durch einen Spinflipper vor dieser Folie können für beide Zustände die Neutronen gezählt werden, über das Verhältnis der Anzahlen wird dann der Winkel bestimmt aus dem dann wiederum die verschobene Larmor-Frequenz bestimmt werden kann.

2.3.2. Verwendung eines Co-Magnetometers

Ein weiterer wichtiger Beitrag Ramseys zur experimentellen Technik ist die Verwendung eines sogenannten Co-Magnetometers [2]. Dabei wird ein weiteres Gas in das Volumen mit den Neutronen gegeben um die unvermeidlichen Fluktuationen des \mathbf{B}_0 -Feldes zu vermessen, die ansonsten die Messung beeinflussen würden. Dabei ist wichtig, dass das Co-Magnetometer kein eigenes EDM hat und vom elektrischen Feld nicht beeinflusst wird. Deshalb wird im geplanten Experiment ^{199}Hg verwendet werden, dessen elektrisches Dipolmoment sehr genau eingeschränkt ist (nach Ref. [2], S. 3 ist $d_{\text{Hg}} < 3.1 \cdot 10^{-29}$ ecm) und auf dessen Grundzustand ein elektrisches Feld keinen Einfluss hat, weil es sich um ein $^1\text{S}_0$ -Atom handelt (Ref. [9], S. 207).

Zur Bestimmung des Magnetfeldes in der Kammer wird das Co-Magnetometer zunächst durch optisches Pumpen polarisiert und mit einem eigenen \mathbf{B}_1 -Puls² zur Präzession gebracht. Durch einen Laserstrahl in der Kammer wird ein effektives Magnetfeld (siehe 2.4.3) in dessen Ausbreitungsrichtung (die in der x-y-Ebene liegt) erzeugt. Dieser gibt eine neue Quantisierungsrichtung für die Polarisation der Hg-Atome entlang der Strahl-

²Die der Puls für ^{199}Hg beeinflusst die Neutronen nicht (und umgekehrt), da die Larmor-Frequenz von Quecksilber und Neutronen sich erheblich unterscheidet.

2. Beschreibung des Experimentes

richtung vor. Wenn Polarisation und Strahlrichtung sich gerade entgegenstehen wird ein Photon, das auf ein Hg-Atom trifft, sicher absorbiert und der Polarisationsvektor wird in Strahlrichtung gedreht. Wenn Strahlrichtung und Polarisation hingegen schon parallel sind, kann kein Photon absorbiert werden. Für alle Winkel zwischen diesen beiden Extremfällen hängt die Wahrscheinlichkeit einer Absorption nach der Quantenmechanik von der Projektion der Polarisation auf die Strahlrichtung ab. Der Strahl wird also entsprechend der Präzession der Hg-Atome amplitudenmoduliert. Aus der Modulationsfrequenz kann dann das Magnetfeld, das die Hg-Atome umgibt, bestimmt werden.

2.4. Systematische Fehler

2.4.1. Geometrische Phase durch Gradient und $\mathbf{v} \times \mathbf{E}$ -Effekt

Systematische Fehler sind bei den aktuellen Experimenten zur Messung von Dipolmomenten zentral, da sie inzwischen die wichtigste Beschränkung der Messgenauigkeit darstellen. Für Teilchen in Fallen spielen dabei vor allem die in Ref. [11] untersuchten geometrischen Phasen eine wichtige Rolle. Als besonders relevant wird dort die Kombination des durch relativistische Effekte aus dem \mathbf{E} -Feld erzeugten Magnetfeldes \mathbf{B}_m und einem Magnetfeld \mathbf{B}_g mit einem Gradienten $g = \frac{\partial B}{\partial z}$ in z-Richtung, der auch eine Komponente in der radialen Ebene hat, erwähnt. Die beiden Felder werden beschrieben durch

$$\mathbf{B}_m = \frac{\mathbf{E} \times \mathbf{v}}{c^2} \quad \text{und} \quad (2.9)$$

$$\mathbf{B}_g = g \cdot \begin{pmatrix} \frac{r}{2} \\ \frac{r}{2} \\ z \end{pmatrix}. \quad (2.10)$$

Für die entstehende geometrische Phase sind dabei vor allem die radialen Komponenten wichtig, da in z-Richtung klar das \mathbf{B}_0 Feld überwiegt. Da \mathbf{B}_g immer radial nach außen weist, \mathbf{B}_m seine Richtung aber je nach Drehsinn des Teilchens ändert, ergibt sich im Mittel über die Teilchen und die Zeit ein Netto-Feld, das die Larmor-Frequenz der Teilchen beeinflusst. Für ein Teilchen, das sich mit der Winkelgeschwindigkeit ω_r in der x-y-Ebene

2. Beschreibung des Experimentes

bewegt, gibt es nach Ref. [11] folgende Verschiebung $\Delta\omega$ der Larmor-Frequenz ω_0 :

$$\Delta\omega = \frac{\gamma^2 \mathbf{B}_{xy}^2}{2(\omega_0 - \omega_r)} = \frac{\gamma^2}{2(\omega_0 - \omega_r)} \cdot (\mathbf{B}_{g,xy}^2 + \mathbf{B}_{m,xy}^2 + 2\mathbf{B}_{g,xy} \cdot \mathbf{B}_{m,xy}) \quad (2.11)$$

Dabei ist \mathbf{B}_{xy} die Projektion von $\mathbf{B}_g + \mathbf{B}_m$ in die x-y-Ebene. Durch diese Änderung (Ramsey-Bloch-Siegert-Shift genannt) der Larmor-Frequenz ergibt sich über Messzeit ein Phasenfehler $\Delta\varphi$ der Polarisation, der mit dem Effekt eines elektrischen Dipolmomentes verwechselt werden könnte. Dabei tragen die drei Terme von B_{xy}^2 unterschiedlich bei:

- Der Term $\mathbf{B}_{g,xy}^2$ hängt nicht von der Anwesenheit oder Richtung des elektrischen Feldes \mathbf{E} ab und lässt sich damit von einem elektrischen Dipolmoment unterscheiden.
- Der Term $\mathbf{B}_{m,xy}^2$ hängt von \mathbf{E}^2 ab. Deshalb bleibt die Verschiebung bei einer Umpolung des elektrischen Feldes gleich. Dadurch lässt er sich von einem EDM unterscheiden, das bei einer Umpolung eine Verschiebung in die andere Richtung bewirkt.
- Der Term $2\mathbf{B}_{g,xy} \cdot \mathbf{B}_{m,xy}$ hängt jedoch genau wie die Verschiebung durch ein elektrisches Dipolmoment linear von \mathbf{E} ab und ist deshalb nicht von einem elektrischen Dipolmoment des Neutrons unterscheidbar. Deshalb ist es wichtig, diesen Beitrag zu simulieren, damit sein Beitrag berücksichtigt werden kann.

2.4.2. Unterschiedlicher Schwerpunkt von Neutronen und Co-Magnetometer im Schwerefeld der Erde

Eine weitere Ursache für systematische Fehler ist bei in z-Richtung inhomogenem Magnetfeld der unterschiedliche Schwerpunkt von Neutronen und Co-Magnetometer-Gas. Letzteres hat eine hohe kinetische Energie und füllt die Kammer deshalb homogen aus, die ultrakalten Neutronen hingegen haben eine sehr geringe kinetische Energie (siehe 2.2) und werden deshalb merklich von der Gravitation beeinflusst. Zum Testen wurde ein Testlauf der Simulation mit Neutronen der Temperatur 5 mK über mehr als 500 simulierte Stunden, bei dem ca. 2 Milliarden Datenpunkte aufgenommen wurden, durchgeführt. Dabei ergab sich eine mittlere z-Koordinate von 5.789 cm bei 12 cm Kammerhöhe. Der Schwerpunkt des Neutronengases ist also um 2.1 mm unter dem Mittelpunkt der Kammer, der den Schwerpunkt des Hg-Co-Magnetometers darstellt.

2.4.3. Weitere systematische Fehler

Neben den geometrischen Phasen treten noch weitere systematische Fehler auf. Durch den Lichtstrahl, mit dem das Co-Magnetometer ausgelesen wird, kann eine sogenannte Light-Shift verursacht werden, die sich nach Ref. [12] wie ein effektives Magnetfeld in Ausbreitungsrichtung des Strahles auswirkt. Deshalb ist ein solches Feld innerhalb eines waagrechten Zylinders (siehe 3.2.4) in der Simulation implementiert. Im geplanten Experiment soll das Auftreten eines Light-Shifts reduziert werden, indem als Lichtquelle nicht eine Hg-Dampflampe wie von Baker et al. [3], sondern ein Laser verwendet wird. Dieser ermöglicht eine sehr geringe spektrale Breite Δ der einfallenden Photonen und kann sehr genau auf die Wellenzahl k des Feinstrukturübergangs von ^{199}Hg eingestellt werden, wodurch nach Ref. [12], Abb. 4 auf S. 328 kein Light-Shift entsteht. Außerdem kann mit einem Laser die Intensität besser reguliert und der Strahldurchmesser reduziert werden.

Eine weitere wichtige Quelle für systematische Effekte sind Dipolfelder innerhalb der Kammer. Diese können durch den Versuchsaufbau selbst bedingt sein. Von Baker et al. [3] wird zum Beispiel ein Dipolfeld genannt, das in der Öffnung entsteht, durch die die Neutronen in das Experiment gelangen. Auch durch einen Leckstrom, der kreisförmig entlang dem Rand der Kammer fließt (Ref. [2], S. 14ff.) kann ein Magnetfeld entstehen, das durch einen Dipol genähert werden kann. Zum andern treten magnetische Dipole durch Materialien auf, die nicht perfekt entmagnetisiert sind. Im Rahmen dieser Arbeit wurde bereits die Möglichkeit implementiert, eine beliebige Anzahl von Dipolen mit beliebiger Richtung im Raum zu verteilen (siehe 3.2.3). Da Dipolfelder nur eine Näherung an die tatsächlich auftretenden Felder sind, werden in Zukunft auch Multipole höherer Ordnung oder Feldkarten (siehe 3.2.5) implementiert werden müssen. Durch Feldkarten ist auch eine Beschleunigung der Simulation bei komplexen Feldern zu erwarten.

3. Entwicklung der Simulation

Die Simulation besteht im Wesentlichen aus drei Teilen. Davon berechnet einer die Flugbahnen der Teilchen (auch „Tracking“ genannt), ein weiterer gibt das Magnetfeld in der Kammer zurück und ein dritter löst ausgehend davon die in (2.4) gegebene Bloch-Gleichung. Im Folgenden wird auf einige wichtige Aspekte dieser Teile eingegangen.

3.1. Flugbahnen der Teilchen

Da das Magnetfeld in der Kammer im Allgemeinen nicht homogen und damit ortsabhängig ist und für den $\mathbf{v} \times \mathbf{E}$ -Effekt (siehe 2.4.1) die Geschwindigkeit der Teilchen bekannt sein muss, ist die Bestimmung der Flugbahnen der Teilchen ein zentraler Bestandteil der Simulation. Flugbahnen, in denen die neue Geschwindigkeit wie in Ref. [6] bei jedem Schritt zufällig neu ermittelt wird, kommen dabei nicht in Betracht, da die ballistischen Flugbahnen einen Einfluss auf den Schwerpunkt der Neutronenwolke haben (siehe 2.4.2) und die spekulare Streuung an den Wänden einen Einfluss auf die Entstehung geometrischer Phasen (siehe 2.4.1) hat.

3.1.1. Anfangsgeschwindigkeit

Für die Simulation kann davon ausgegangen werden, dass die Neutronen ihre Bahn an zufälligen Stellen innerhalb des Zylinders mit zufälligen Geschwindigkeiten beginnen. Die Anfangsposition wird ermittelt, indem drei gleichförmig verteilte Zufallszahlen für die drei Raumkoordinaten erzeugt werden die innerhalb des Quaders liegen, der die zylinderförmige Kammer umschließt. Falls der so ermittelte Punkt außerhalb des Zylinders liegt¹, werden neue Zufallszahlen bestimmt und der Test wiederholt.

¹Die Wahrscheinlichkeit dafür kann man sich anhand eines Kreises mit Radius r und dem umgebenden Quadrat mit Seitenlänge $2 \cdot r$ berechnen. Die Wahrscheinlichkeit, dass ein zufälliger Punkt außerhalb des Kreises ist, beträgt dann $\frac{\pi r^2 - (2r)^2}{(2r)^2} \approx 21,4\%$.

3. Entwicklung der Simulation

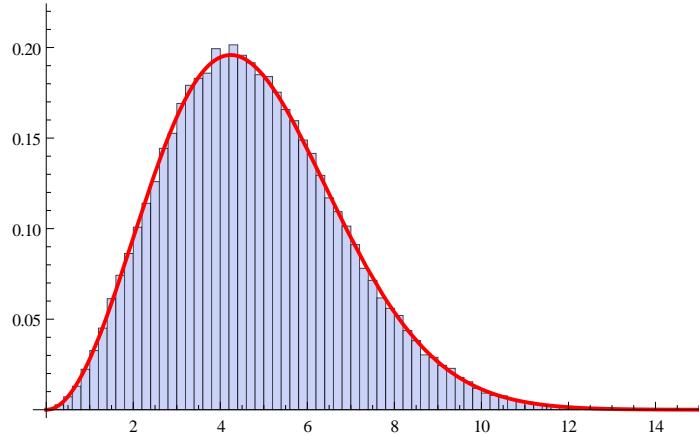


Abbildung 3.1.: Wenn v_x , v_y und v_z normalverteilt sind, folgt $\sqrt{v_x^2 + v_y^2 + v_z^2}$ einer Maxwell-Boltzmann-Verteilung. Für das Bild wurden mit Mathematica 100000 Tripel von normalverteilten Zufallszahlen mit $\sigma = 3$ erzeugt. Die rote Kurve ist ein Fit auf eine Maxwell-Boltzmann-Verteilung, dabei ergibt sich $\sigma = 2.997$.

Bei der Geschwindigkeit kommen zusätzlich physikalische Überlegungen ins Spiel. In der UCN-Quelle werden die Neutronen durch Stöße mit einem sehr kalten Festkörper abgekühlt. Dabei wird das thermische Gleichgewicht erreicht (Ref. [2], S. 9f.), die Geschwindigkeit der Neutronen nach der UCN-Quelle folgt also einer Maxwell-Boltzmann-Verteilung. Beim Transport durch Neutronenleiter können die schnelleren Neutronen noch herausgefiltert werden, indem die in Abschnitt 2.2 beschriebene Totalreflexion nur für die erwünschten langsamen Neutronen ermöglicht wird, zum Beispiel durch gebogene Neutronenleiter (Ref. [2], S. 10f.). Deshalb kann als Spektrum eine bei einer Grenzgeschwindigkeit v_g abgeschnittene Maxwell-Boltzmann-Verteilung angenommen werden.

Um diese Verteilung zu realisieren, werden im Programm die Komponenten v_x , v_y und v_z als normalverteilte Zufallszahlen bestimmt. Die dabei verwendete Standardabweichung ist

$$\sigma = \sqrt{\frac{k_B T}{m}} . \quad (3.1)$$

Der Geschwindigkeitsbetrag $|\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$ ist nach Ref. [13], S. 212 dann nach der Maxwell-Boltzmannschen Geschwindigkeitsverteilung verteilt. Falls $|\mathbf{v}|$ größer als die Grenzgeschwindigkeit v_g ist, werden neue Werte für v_x , v_y und v_z gewürfelt. In Abbildung 3.1 wird zur Untermalerung dieser Aussage ein Histogramm aus Zufallszahlen mit einem entsprechenden Fit gezeigt, in Abbildung 3.2 ist außerdem das Ergebnis eines Testlaufes der Simulation gezeigt.

3. Entwicklung der Simulation

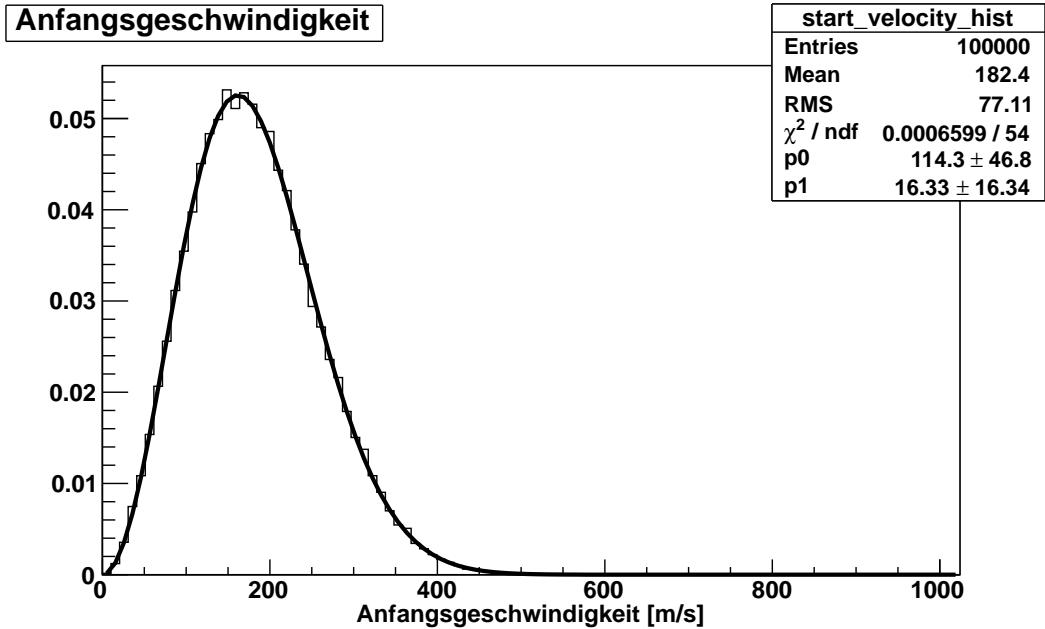


Abbildung 3.2.: Histogramm der Verteilung der Anfangsgeschwindigkeit bei einem Testlauf mit 100000 Hg-Atomen mit $T = 300$ K und $m = 3.18 \cdot 10^{-25}$ kg. Erwartet wird dementsprechend $\sigma = 114.12714 \frac{\text{m}}{\text{s}}$, der Fit ergibt $\sigma_{Fit} = (114.3 \pm 46.8) \frac{\text{m}}{\text{s}}$.

3.1.2. Berechnung der Flugbahn

Für die Berechnung der Flugbahnen der Teilchen in der Kammer wurden zwei verschiedene Ansätze erprobt, die im Folgenden weiter ausgeführt werden.

3.1.2.1. Numerische Integration der Bewegungsgleichung

Die Bewegungsgleichung der Newtonschen Dynamik kann für ein Teilchen das Masse m im Kraftfeld $\mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t)$ geschrieben werden als

$$\ddot{\mathbf{x}} = \frac{\mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t)}{m}. \quad (3.2)$$

Diese Differentialgleichung kann mit $\mathbf{v} = \dot{\mathbf{x}}$ und $\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = \frac{1}{m} \cdot \mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t)$ als System von Differentialgleichungen erster Ordnung geschrieben werden:

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) \end{cases}. \quad (3.3)$$

3. Entwicklung der Simulation

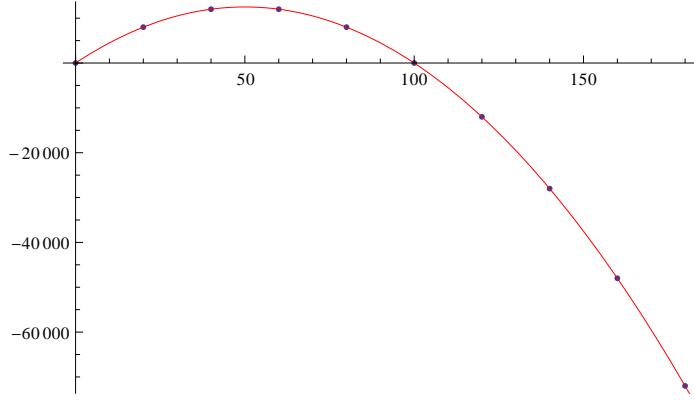


Abbildung 3.3.: Integration von (3.3) mit konstanter Beschleunigung in einer Dimension.
Die blauen Punkte sind die Ergebnisse der Runge-Kutta-Methode vierter Ordnung, die rote Kurve ist ein quadratischer Fit durch die Punkte. Dabei wurden die Koeffizienten des Polynoms im Rahmen der Maschinengenauigkeit exakt richtig ermittelt. Trotz der großen Schrittweite wird die Bewegungsgleichung also exakt gelöst.

Für ein Teilchen im Gravitationsfeld ist zum Beispiel

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = -g\hat{\mathbf{e}}_z . \quad (3.4)$$

In der als **EquationTracker** implementierten Methode zur Konstruktion der Flugbahnen wird das System linearer Differentialgleichungen (3.3) mit Hilfe eines Runge-Kutta-Verfahrens vierter Ordnung mit konstanter Schrittweite (siehe 3.4.1) numerisch gelöst. Da es sich um ein Verfahren vierter Ordnung handelt, ist der Fehler $O(t^5)$ (Ref. [14], S. 907), also ist die Lösung für konstante Beschleunigung wie in (3.4) sogar (bis auf Rundungsfehler) exakt, da die analytische Lösung nur zweiter Ordnung ist. In Abbildung 3.3 wird dies gut sichtbar.

Um auch für die Zeitpunkte zwischen den Runge-Kutta-Schritten eine Position angeben zu können, wurde zunächst ein kubisches Interpolationspolynom (Ref. [14], S. 916) ausprobiert. Da dieses Interpolationspolynom die Genauigkeit nicht verbesserte, sondern durch Rundungsfehler eher verschlechterte und die Simulation verlangsamte, wurde ein quadratisches Interpolationspolynom verwendet. Dieses wurde mit Hilfe von Mathematica ermittelt und lautet für ein Teilchen, das zur Zeit t_0 am Ort x_0 die Geschwindigkeit

3. Entwicklung der Simulation

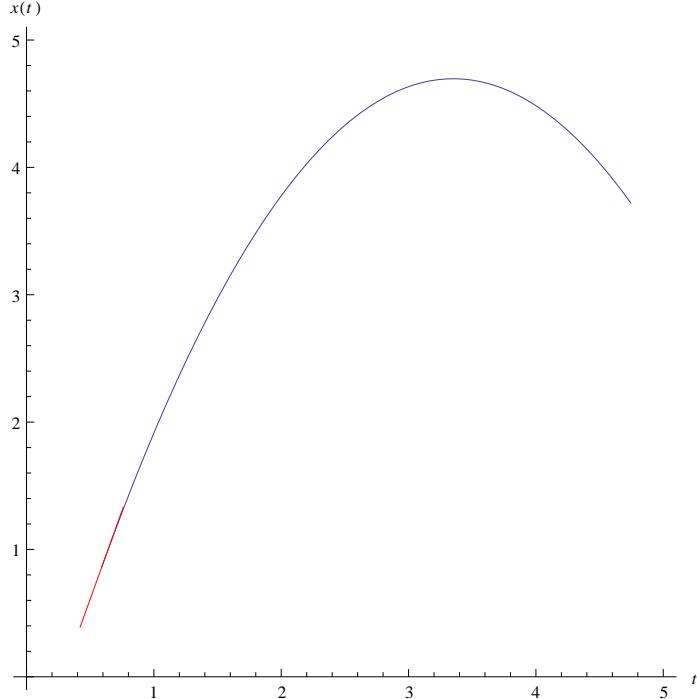


Abbildung 3.4.: Quadratische Interpolation nach (3.5). Die rote Tangente gibt die Anfangsgeschwindigkeit v_0 an, der Anfang und das Ende der Kurve ergeben (t_0, x_0) und (t_1, x_1) .

v_0 hatte und zur Zeit t_1 am Ort x_1 ist:

$$\begin{aligned} x(t) &= a_0 + a_1 t + a_2 t^2 \\ a_0 &= \frac{t_0(t_0(t_1 v_0 + x_1) + t_1(-t_1 v_0 - 2x_0)) + t_1^2 x_0}{t_0(t_0 - 2t_1) + t_1^2} \\ a_1 &= \frac{t_0(-t_0 v_0 + 2x_0 - 2x_1) + t_1^2 v_0}{t_0(t_0 - 2t_1) + t_1^2} \\ a_2 &= \frac{t_0 v_0 - t_1 v_0 - x_0 + x_1}{t_0(t_0 - 2t_1) + t_1^2}. \end{aligned} \tag{3.5}$$

In Abbildung 3.4 ist das Interpolationspolynom für ein Beispiel angegeben.

Wenn sich ein Teilchen nach dem Schritt außerhalb des Gefäßes befindet, muss die Stelle gefunden werden, an der es reflektiert hätte werden müssen. Dafür wird je nachdem, an welcher Seite das Gefäß verlassen wurde, ein Interpolationspolynom für $z(t)$, $z(t) - H$ oder $r^2(t) = x(t)^2 + y(t)^2 - R^2$ aus den Interpolationspolynomen für die einzelnen Komponenten erstellt und mit einem abgewandelten Newton-Verfahren (siehe 3.4.2) die Nullstelle t_0 gesucht. Dann wird ein kürzerer Runge-Kutta-Schritt ausgeführt, der bei t_0

3. Entwicklung der Simulation

endet. Nach dem Schritt befindet sich das Teilchen also in der unmittelbaren Nähe der Wand. Dort wird dann eine spekulare Reflexion oder diffuse Streuung durchgeführt. Für jeden ermittelten Punkt werden Ort, Geschwindigkeit und Zeit in Listen geschrieben auf deren Basis dann für jeden beliebigen Zeitpunkt die Position wie oben beschrieben interpoliert werden kann.

Der Vorteil dieser Methode ist, dass sie sich für beliebige Geometrien verwenden lässt, die unabhängig vom Tracking an sich implementiert werden können. Auch die Bewegungsgleichung kann unabhängig davon modifiziert werden, indem ein anderes $\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t)$ für (3.3) verwendet wird. Diese Flexibilität wird erkauft durch eine deutlich geringere numerische Genauigkeit der Bahnerzeugung, die zwar für die Genauigkeit des Ergebnisses irrelevant ist, aber insbesondere bei hohen Geschwindigkeiten oft dazu führt, dass bei der Suche der Nullstelle keine gefunden werden und die Simulation deshalb abgebrochen werden muss. Durch das Ändern der Einstellungen des Trackers kann die Simulation dann zwar wieder zum Laufen gebracht werden, allerdings bedeutet dies einen erheblichen Mehraufwand für den Anwender. Deshalb wurde zusätzlich ein schneller und einfacher neuer Tracker entwickelt, der aber nur Zylinder unterstützt.

3.1.2.2. Analytische Lösung der Bewegungsgleichung

Wenn man die Gravitation in z-Richtung als einzige wirkende Kraft voraussetzt, lässt sich (3.2) direkt lösen. Für ein Teilchen, das sich zum Zeitpunkt t_0 am Ort $\mathbf{x}_0(t) = (x_0, y_0, z_0)$ mit Geschwindigkeit $\mathbf{v}_0(t) = (v_{x0}, v_{y0}, v_{z0})$ befindet, folgt es der Parabel

$$x(t) = x_0 + v_{x0}(t - t_0) \quad (3.6)$$

$$y(t) = y_0 + v_{y0}(t - t_0) \quad (3.7)$$

$$z(t) = z_0 + v_{z0}(t - t_0) - \frac{g}{2}(t - t_0)^2. \quad (3.8)$$

Auch die Geschwindigkeit lässt sich direkt berechnen:

$$v_x(t) = v_{x0} \quad (3.9)$$

$$v_y(t) = v_{y0} \quad (3.10)$$

$$v_z(t) = v_{z0} - g(t - t_0). \quad (3.11)$$

Wenn man nun fest von einem Zylinder mit Höhe H und Radius R als Geometrie ausgeht, lassen sich Kandidaten für die nächste Kollisionszeit durch Lösen der folgenden

3. Entwicklung der Simulation

quadratischen Gleichungen bestimmen:

$$z(t) = 0 \quad (\text{Boden}) \quad (3.12)$$

$$z(t) - H = 0 \quad (\text{Deckel}) \quad (3.13)$$

$$x(t)^2 + y(t)^2 - R^2 = 0 \quad (\text{Radius}) \quad (3.14)$$

Die Lösung der quadratischen Gleichung kann direkt bestimmt werden und wird der GNU Scientific Library [15] überlassen. Aus den (bis zu sechs) Ergebnissen muss nun noch die richtige Zeit bestimmt werden. Grundsätzlich ist dies die kleinste positive Lösung. Allerdings ergibt sich, wenn das Teilchen an einer Wand startet, auch $t \approx 0$ als Lösung. Deshalb wird bei der Oberfläche, an der die letzte Kollision stattfand, die betragsmäßig kleinere Nullstelle nicht berücksichtigt. Bei jeder Kollision werden Ort, Geschwindigkeit und Zeit in eine Liste geschrieben, auf deren Basis dann nach (3.6) etc. die Geschwindigkeit und der Ort für alle Zeiten bis zur nächsten Kollision berechnet werden können. In Abschnitt 3.4.3 wird beschrieben, wie der passende Listeneintrag gefunden werden kann.

3.1.3. Spekulare Reflexion

Die spekulare Reflexion der Teilchen hat einen Einfluss auf die Entstehung der geometrischen Phasen und muss deshalb simuliert werden. Sie tritt wie in Abschnitt 2.2 beschrieben auf, wenn das Neutron auf eine glatte Oberfläche trifft. Bei rauen Oberflächen und bei Gasteilchen tritt hingegen die in Abschnitt 3.1.4 beschriebene diffuse Streuung auf.

Für den Deckel und Boden des Zylinders muss für die spekulare Reflexion nur die Geschwindigkeit in z-Richtung gespiegelt werden:

$$v'_z = -v_z . \quad (3.15)$$

Für die Spiegelung an der runden Zylinderwand müssen nur die x und y Komponenten betrachtet werden. Deshalb werden mit $\mathbf{v} = (v_x, v_y)$ und $\mathbf{r} = (r_x, r_y)$ im Folgenden die Projektionen von Orts- und Geschwindigkeitsvektor in die x-y-Ebene bezeichnet, die z-Komponente bleibt bei der radialen Reflexion erhalten. Um die Geschwindigkeit \mathbf{v}' (siehe Abbildung 3.5) nach dem Stoß mit der Wand zu erhalten, muss \mathbf{v} an \mathbf{r} gespiegelt

3. Entwicklung der Simulation

werden. Dafür führt man den Einheitsvektor $\hat{\mathbf{n}}$ ein, der senkrecht auf \mathbf{r} steht:

$$\hat{\mathbf{n}} = \frac{1}{\sqrt{r_x^2 + r_y^2}} \cdot \begin{pmatrix} r_x \\ -r_y \end{pmatrix}. \quad (3.16)$$

Mit dessen Hilfe kann man den gespiegelten Vektor \mathbf{v}' berechnen als

$$\mathbf{v}' = -\mathbf{v} + 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}. \quad (3.17)$$

3.1.4. Diffuse Streuung

Wenn die Oberfläche der Kammer nicht eben ist, kann es beim Auftreffen eines Neutrons auch zu einer diffusen Streuung des Neutrons kommen. Bei Atomen (wie dem ^{199}Hg Co-Magnetometer) ist dies aufgrund der deutlich geringeren Kohärenzlänge sogar immer der Fall. Bei der diffusen Streuung bleibt zwar die Energie erhalten, die Richtung nach der Reflexion ist allerdings zufällig. Um dies zu erreichen, wird ein neuer Geschwindigkeitsvektor nach der in Abschnitt 3.1.1 beschriebenen Methode bestimmt und normiert. Teilchen, deren Geschwindigkeitsvektor nach der Reflexion nicht ins Innere der Kammer zeigt sind für die Simulation uninteressant, da sie entweder die Kammer verlassen oder gleich erneut reflektiert werden. Deshalb wird das Skalarprodukt $\hat{\mathbf{n}} \cdot \hat{\mathbf{v}'}$ des Vektors $\hat{\mathbf{n}}$, der senkrecht auf der Fläche steht, an der die diffuse Streuung stattfindet und aus der Kammer heraus zeigt und dem normierten neuen Geschwindigkeitsvektor $\hat{\mathbf{v}'}$ bestimmt. Falls das Skalarprodukt negativ ist², wird ein neuer normierter Geschwindigkeitsvektor bestimmt.

3.2. Magnetfeld

Für eine akkurate Simulation des Experimentes muss eine Reihe von erwünschten und unerwünschten Magnetfeldern berücksichtigt werden, die das Ergebnis der Simulation beeinflussen können. Im Folgenden werde ich auf die bereits in die Simulation eingebauten Felder eingehen und auch die Verwendung von Feldkarten beschreiben, die in der für die Arbeit zur Verfügung stehenden Zeit leider nicht mehr realisiert werden konnten.

²Wie man sich leicht überlegen kann, ist die Wahrscheinlichkeit dafür 50%

3. Entwicklung der Simulation

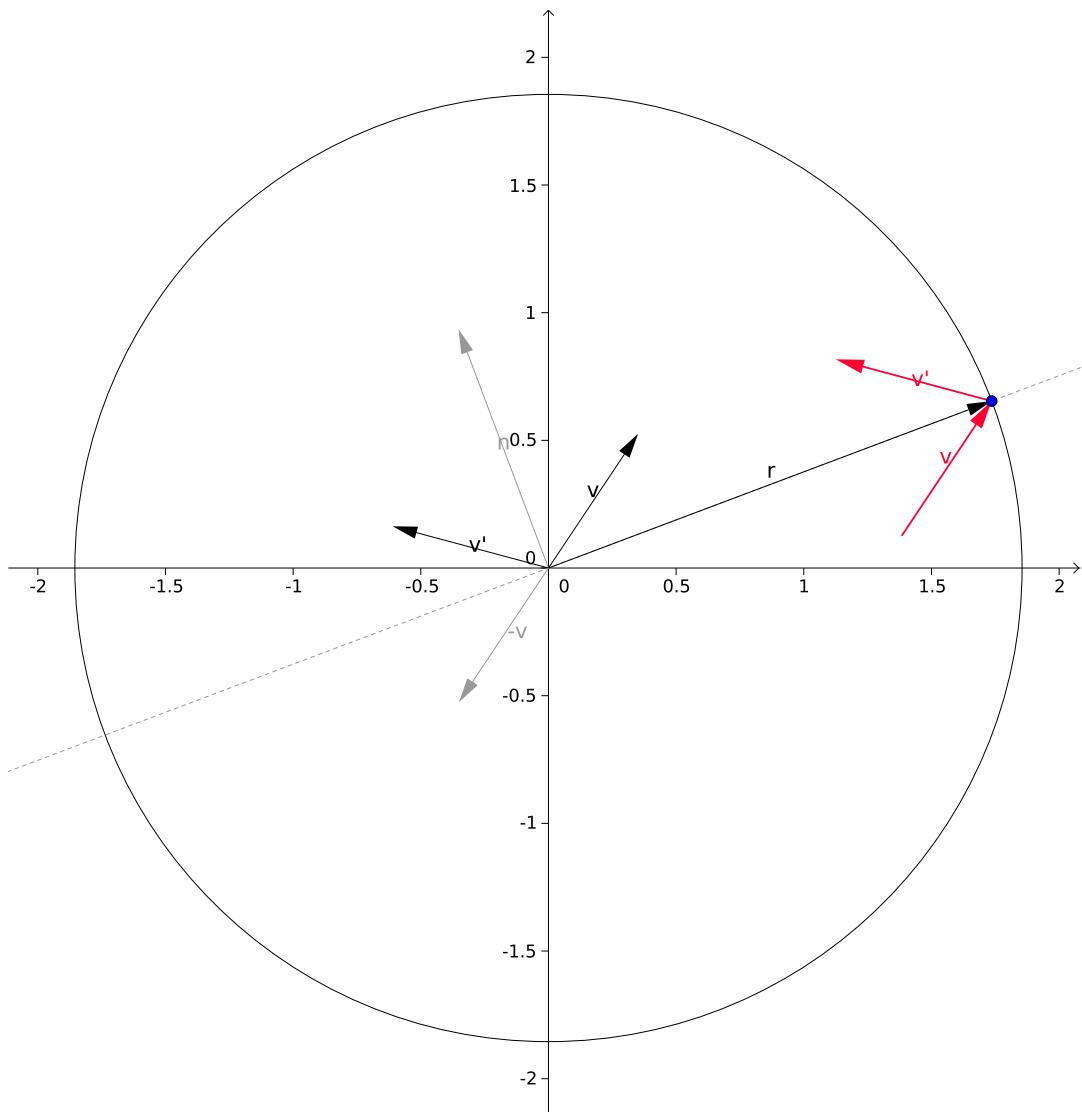


Abbildung 3.5.: Spiegelung der Geschwindigkeit \mathbf{v} am Ortsvektor \mathbf{r} . Die Vektoren sind sowohl am Ursprung als auch am Punkt der Reflexion angetragen.

3. Entwicklung der Simulation

3.2.1. Homogenes Magnetfeld

Das betragsmäßig stärkste und einzige im Experiment erwünschte Magnetfeld ist das homogene \mathbf{B}_0 -Feld, dessen Ausrichtung parallel zur z-Achse ist. In der Simulation besteht die Möglichkeit, eine beliebige Anzahl homogener Felder mit beliebiger Ausrichtung einzustellen.

3.2.2. Magnetfeld mit Gradient

Ein Magnetfeld mit linearem Gradienten in z-Richtung ist eine erste grobe Abschätzung dafür, wie ein inhomogenes Feld die Ergebnisse des Experiments beeinflusst. Es bietet sich für die Simulation an, da die Effekte von Pendlebury et al. [11] (siehe auch 2.4.1) bereits ausführlich untersucht wurden. Für die Simulation wird das dort verwendete Feld, das auch radiale Komponenten hat, verwendet:

$$\mathbf{B}_g = \frac{\partial B_z}{\partial z} \cdot \begin{pmatrix} \frac{r}{2} \\ \frac{r}{2} \\ z \end{pmatrix}. \quad (3.18)$$

3.2.3. Dipolfeld

Nach Ref. [16], S. 246 kann das Magnetfeld eines Dipols im Koordinatenursprung mit dem magnetischen Dipolmoment \mathbf{m} beschrieben werden als

$$\mathbf{B}_d = \frac{\mu_0}{4\pi} \cdot \frac{3(\mathbf{m} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} - \mathbf{m}}{r^3} = \frac{3(\mathbf{m} \cdot \mathbf{r})\mathbf{r} - \mathbf{m}r^2}{r^5} \cdot 10^{-7} \frac{\text{H}}{\text{m}}. \quad (3.19)$$

Für einen Dipol an einem beliebigen Ort \mathbf{r}' ersetzt man \mathbf{r} durch $\mathbf{r} - \mathbf{r}'$.

3.2.4. Magnetfeld durch Laserstrahl

Um den Einfluss der Light-Shift (siehe 2.4.3) untersuchen zu können, wurde die Möglichkeit eines auf einen zylinderförmigen Strahl beschränkten Magnetfeldes geschaffen. Der Strahl läuft dabei immer durch der Mitte der Kammer und parallel zur x-Achse. Um zu prüfen, ob der Punkt $\mathbf{r} = (x, y, z)$ innerhalb des Strahles liegt, wird der quadrierte

3. Entwicklung der Simulation

Abstand r^2 vom Strahlmittelpunkt berechnet:

$$r^2 = y^2 + (z - h)^2 . \quad (3.20)$$

Dabei ist h der Abstand der Mitte des Strahles vom Boden der Kammer. Wenn r^2 kleiner als der quadrierte Radius des Strahles ist, liegt \mathbf{r} innerhalb und ein Magnetfeld wird zurückgegeben. Ansonsten wird kein Magnetfeld zurückgegeben.

3.2.5. Feldkarten

Eine Möglichkeit auch andere, komplexe Felder in die Simulation zu integrieren sind Feldkarten. Diese machen es möglich, das Feld im Vornherein mit einem anderen Programm zu berechnen. Dadurch können zum Beispiel Felder verwendet werden, die mit Hilfe der Finite-Elemente-Methode berechnet wurden oder aus tatsächlichen Messungen stammen. Leider war es im zeitlichen Rahmen der Arbeit nicht mehr möglich, auch diese Funktionalität zu implementieren. Trotzdem werde ich im Folgenden einige Ansätze für Feldkarten aufzeigen die eine Implementierung recht einfach möglich machen sollten.

Die Verwendung einer Karte führt normalerweise zu zwei Problemen. Zum einen müssen Punkte in der Karte gefunden werden, die in der Nähe des gefragten Punktes liegen. Wenn die Feldkarte nach den einzelnen Koordinaten sortiert ist (was sich immer realisieren lässt) können die entsprechenden Punkte wie in 3.4.3 beschrieben durch binäre Suche in logarithmischer Laufzeit gefunden werden. Noch effizienter und sehr leicht zu implementieren ist die Verwendung eines endlichen Gitters. Dafür muss die Feldkarte so beschaffen sein, dass die kartierten Punkte in einem rechtwinkligen Gitter mit Kantenlänge d sind, für die kartierten Punkte \mathbf{a}_{lmn} muss also gelten:

$$\mathbf{a}_{lmn} = dl\hat{\mathbf{e}}_x + dn\hat{\mathbf{e}}_y + dm\hat{\mathbf{e}}_z, \quad l, n, m \in \mathbb{N}_0 \quad (3.21)$$

Dabei muss der Ursprung des Gitters so gewählt werden, dass die Indices l , n und m immer natürliche Zahlen sind, da C keine negativen Indices unterstützt. Um dies zu erreichen, kann das Gitter entsprechend verschoben werden. Der Koordinatenursprung liegt danach am Punkt $\mathbf{a}_{l_0 m_0 n_0}$. Wenn das Gitter die Abmessungen (L, N, M) hat, kann es in einem Array der Größe $L \cdot N \cdot M$ gespeichert werden. Nun kann man die Gitterkoordinaten (l, m, n) in der Nähe des Punktes $\mathbf{r} = (x, y, z)$ durch eine einfache Rechnung

3. Entwicklung der Simulation

bestimmen:

$$l = \left\lfloor \frac{x}{d} \right\rfloor + l_0 \quad (3.22)$$

$$m = \left\lfloor \frac{x}{d} \right\rfloor + m_0 \quad (3.23)$$

$$n = \left\lfloor \frac{x}{d} \right\rfloor + n_0 \quad (3.24)$$

Dabei bezeichnet $\lfloor x \rfloor$ die größte ganze Zahl, die kleiner als x ist. Mit diesen Gitterkoordinaten kann aus $\mathbf{a}_{lmn} = (x_{lmn}, y_{lmn}, z_{lmn})$ und den angrenzenden Gitterpunkten der Wert $\mathbf{B}(\mathbf{r})$ linear interpoliert werden:

$$B_x(\mathbf{r}) \approx B_x(\mathbf{a}_{lmn}) + (x_{lmn} - x) \cdot \frac{B_x(\mathbf{a}_{l+1,m,n}) - B_x(\mathbf{a}_{lmn})}{x_{l+1,m,n} - x_{lmn}} \quad \text{etc.} \quad (3.25)$$

3.3. Lösung der Bloch-Gleichung

Der Code zur Integration der Bloch-Gleichung wurde von Wolfhart Feldmeier [6] übernommen. Es handelt sich dabei um ein eingebettetes Runge-Kutta-Verfahren (Ref. [14], S. 920), das in einer Webnote [17] zum Buch Numerical Recipes veröffentlicht wurde. Um die Polarisierung auch zu Zeitpunkten zwischen zwei Punkten der numerischen Lösung zu bestimmen, wird die Möglichkeit des sogenannten Dense Output genutzt. Dafür werden im verwendeten Code drei weitere Evaluationen durchgeführt wodurch die Lösung in siebter Ordnung interpoliert werden kann (Ref. [14], S. 920).

Am Anfang der Simulation wird der Polarisationsvektor \mathbf{P} so initialisiert, dass er in der y-z-Ebene liegt:

$$\mathbf{P}_0 = \begin{pmatrix} -\sin(\varphi) \\ 0 \\ \cos(\varphi) \end{pmatrix}. \quad (3.26)$$

Der Winkel φ ist dabei der Winkel zwischen der Anfangspolarisation und der positiven z-Achse. Für $0^\circ \leq \varphi \leq 180^\circ$ ist $\text{atan2}(P_{0y}, P_{0x}) = -\pi$. Dementsprechend kann der Endwinkel der Polarisierung in der x-y-Ebene gegenüber der x-Achse φ_e als positive Zahl dargestellt werden:

$$\varphi_E = \text{atan2}(P_{0y}, P_{0x}) + \pi \in [0, 2\pi). \quad (3.27)$$

3.4. Methoden

3.4.1. Integration von Differentialgleichungen

In der Simulation müssen an zwei Stellen gewöhnliche Differentialgleichungen gelöst werden. Diese können immer in folgender Form geschrieben werden:

$$\dot{x}(t) = f(x, t) . \quad (3.28)$$

Zur Integration werden zwei verschiedene Runge-Kutta-Verfahren (Ref. [14], S. 907ff.) verwendet. Diese basieren darauf, dass für die Ermittlung von $x(t + h)$ nicht nur der Wert $f(x, t)$ der Ableitung zum Beginn des Zeitschrittes, sondern auch Werte von $f(x, t)$ zwischen t und $t + h$ verwendet werden. Durch geschickte Kombination dieser Zwischen-schritte kann man den numerischen Fehler so beschränken, dass die führenden Glieder der Taylor-Entwicklung der Lösung exakt ermittelt werden können. Für die Bestimmung der Flugbahn im **EquationTracker** wird das klassische Runge-Kutta-Verfahren vierter Ordnung verwendet (Ref. [14], S. 908):

$$k_1 := hf(x, t) \quad (3.29)$$

$$k_2 := hf\left(x + \frac{k_1}{2}, t + \frac{h}{2}\right) \quad (3.30)$$

$$k_3 := hf\left(x + \frac{k_2}{2}, t + \frac{h}{2}\right) \quad (3.31)$$

$$k_4 := hf(x + k_3, t + h) \quad (3.32)$$

$$x(t + h) = x(t) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \quad (3.33)$$

Dabei wird der Einfachheit halber eine konstante Schrittweite h verwendet. Für die wichtigere Integration der Bloch-Gleichung wird ein komplizierteres Runge-Kutta-Verfahren verwendet, das bereits in 3.3 beschrieben wurde.

3.4.2. Finden von Nullstellen

Zum Finden von Nullstellen im **EquationTracker** wird ein numerisches Verfahren verwendet, um eine Übertragung auf andere Geometrien zu ermöglichen bei denen die Abstands-funktionen vom Rand eine höhere Ordnung als vier³ haben. Bei besagtem Verfah-

³Für Polynome bis zur vierten Ordnung existieren explizite Lösungsformeln.

3. Entwicklung der Simulation

ren handelt es sich um ein leicht modifiziertes Newton-Verfahren, bei dem sichergestellt wird, dass der Schätzwert für die Nullstelle immer innerhalb eines Intervalls bleibt und das die meisten Endlosschleifen, die beim Newton-Verfahren entstehen können (Ref. [14], S. 458) verhindert werden. Dafür wird bei jedem Schritt überprüft, ob der nach dem Newton-Verfahren berechnete Schätzwert x_{n+1} noch innerhalb des gewünschten Intervalls liegt und ob der Betrag des Funktionswertes $|f(x_{n+1})|$ dort geringer ist als der Funktionswert $|f(x_n)|$ der vorherigen Iteration. Falls nicht wird das Intervall stattdessen mit Hilfe eines Bisektionsschrittes verkleinert, das heißt das Vorzeichen des Funktionswertes in der Mitte des Intervalls wird betrachtet und das Intervall so verkleinert, dass sich weiterhin ein Vorzeichenwechsel innerhalb des halbierten Intervalls befindet. In den seltenen Fällen in denen auch dieses Verfahren scheitert, wird nach 1000 Iterationen die Suche abgebrochen und das entsprechende Teilchen für die Simulation verworfen.

3.4.3. Binäre Suche

Um die zu einem bestimmten Wert x nächsten Werte in einer sortierten Liste v und deren Indices l und u zu finden, wird die sogenannte binäre Suche verwendet. Dafür werden l und u zunächst auf den Anfang und das Ende der Liste gesetzt also $l := 0$ und $u := \text{len}(v) - 1$ wobei $\text{len}(v)$ die Länge der Liste bezeichnet. Nun sollen u und l so geändert werden, dass $u - l$ immer kleiner wird und folgende Invariante erhalten bleibt:

$$v[l] \leq x < v[u] \quad (3.34)$$

Dazu wird der Index m in der Mitte von u und l ermittelt:⁴

$$m = \left\lfloor \frac{u + l}{2} \right\rfloor \quad (3.35)$$

Danach wird entweder $l := m$ oder $u := m$ gesetzt sodass die Invariante (3.34) erhalten bleibt. Diese Schritte werden wiederholt bis $u - l = 1$ ist. Da für $u - l > 1$ immer $l < m < u$ ist und bei jedem Schritt entweder l oder u auf m gesetzt werden konvergiert das Verfahren garantiert. Die Konvergenzordnung ist $O(\log n)$, da das Intervall bei jedem Schritt halbiert wird.

⁴Dabei bezeichnet $\lfloor x \rfloor$ die größte ganze Zahl, die kleiner als x ist.

3.4.4. Zufallszahlen

Die für die Simulation benötigten Zufallszahlen werden mit Hilfe des in der GNU Scientific Library [15] implementierten Mersenne-Twisters erzeugt. Der Seed muss als Parameter in `params.txt` übergeben werden, ansonsten wird immer die gleiche Zufallszahlenfolge erzeugt.

3.5. Eingabe und Ausgabe

3.5.1. Parameterdatei

Für jeden Lauf der Simulation muss dem Programm ein Satz von Parametern übergeben werden, zum Beispiel die gewünschte Anzahl von Teilchen oder das gyromagnetische Verhältnis. Diese Parameter werden von der Standardeingabe gelesen und können zum Beispiel in einer Datei übergeben werden. In Tabelle 3.1 sind die verschiedenen Parameter beschrieben. Die Parameterdatei muss einen Parameter pro Zeile enthalten, der Wert wird dabei durch ein Leerzeichen abgetrennt und muss in der in C üblichen Schreibweise angegeben werden. Leere Zeilen und Zeilen, die mit dem Zeichen „#“ beginnen werden ignoriert um Kommentare in der Parameterdatei zu erlauben.

Um die Simulation mit der Parameterdatei `params.txt` im aktuellen Verzeichnis zu starten, kann man folgenden Aufruf verwenden:

```
./cylindric < params.txt
```

3.5.2. Felddatei

In einer weiteren Eingabedatei, die den Namen `fields.dat` im aktuellen Arbeitsverzeichnis haben muss, werden die verschiedenen Magnetfelder definiert. Jedes Magnetfeld entspricht einer Zeile in der Datei, dabei werden wie bei den Parametern Leerzeilen und Zeilen, die mit einem „#“ beginnen, ignoriert. Jede der Zeilen beginnt mit einem Buchstaben, der die Art des Feldes beschreibt und, durch Leerzeichen getrennt, einem oder mehreren Parametern. Alle Parameter haben den Typ „double“ und werden in der in C üblichen Schreibweise akzeptiert. Vektoren (vom Typ `Threevector`) werden als drei einzelne Zahlen übergeben. Die verschiedenen Felder und ihre Parameter werden in Tabelle 3.2 aufgelistet.

3. Entwicklung der Simulation

Name	Einheit	Beschreibung
GyromagneticRatio	$\text{rad} \cdot \text{T}^{-1} \cdot \text{s}^{-1}$	Gyromagnetisches Verhältnis γ der Teilchen. Dabei muss beachtet werden, dass der Wert als Kreisfrequenz angegeben werden muss, für Neutronen also zum Beispiel $-1.83\text{e}8$
NumberOfParticles	1	Anzahl der Teilchen, für die die Simulation durchgeführt werden soll
Lifetime	s	Lebensdauer eines Teilchens
ErrorGoal	rad	Genauigkeitsziel bei der Integration der Boltz-Gleichung, siehe Ref. [6] und Ref. [14], S. 910ff.
Flipangle	°	Winkel der Anfangspolarisation zur z-Achse (siehe 3.3)
Seed	1	Initialisierungswert des Pseudozufallszahlengenerators.
CylinderRadius	m	Radius des Zylinders, in dem das Experiment durchgeführt wird
CylinderHeight	m	Höhe des Zylinders, in dem das Experiment durchgeführt wird
ParticleDataNum	1	Zahl der Teilchen, für die zeitaufgelöste Daten und nicht nur Anfangsgeschwindigkeit und Endpolarisation erfasst werden sollen
SaveTimeDiff	s	Zeitauflösung der gespeicherten Daten während des Durchlaufes
GravitationConstant	$\text{m} \cdot \text{s}^{-2}$	Erdbeschleunigung, in der Regel 9.81
Temperature	K	Temperatur der Teilchen
ParticleMass	kg	Masse der Teilchen, wird zusammen mit Temperature für die Boltzmannverteilung der Anfangsgeschwindigkeiten verwendet
VelocityCutoff	$\text{m} \cdot \text{s}^{-1}$	Höchste Anfangsgeschwindigkeit der Teilchen
Timeout	s	Zeit, die der Durchlauf eines Teilchens höchstens in Anspruch nehmen soll
DiffusionProbability	1	Wahrscheinlichkeit zwischen 0 und 1, mit der bei einem Wandstoß diffuse Streuung statt spekularer Reflexion auftreten soll
MinDiffusionAngle	°	Minimaler Winkel gegenüber der Oberfläche bei Diffusion
TrackerStepSize	s	<i>Schrittweite bei der Erstellung des Tracks</i>
CollisionAccuracy	m	<i>Genauigkeit der Wandstöße</i>

Tabelle 3.1.: Auflistung aller Parameter der Simulation. Die kursiven Parameter werden nur vom `EquationTracker` verwendet.

3. Entwicklung der Simulation

Zeichen	Parameter	Typ	Einheit	Beschreibung
H		Threevector	T	Homogenes Magnetfeld \mathbf{B}_0
E		Threevector	$V \cdot m^{-1}$	Elektrisches Feld für $\mathbf{v} \times \mathbf{E}$ -Effekt
G		double	$T \cdot m^{-1}$	Gradient $\partial_z B_z$ des Gradientenfeldes
D		Threevector	$A \cdot m^2$	Magnetisches Moment \mathbf{m} des Dipols
		Threevector	m	Position des Dipols
L		Threevector	T	Magnetfeld innerhalb des Lichtstrahls
		double	m	Höhe der Mitte des Lichtstahles
		double	m	Radius des Lichtstrahles

Tabelle 3.2.: Auflistung der Typen von Magnetfeldern, die in `fields.dat` definiert werden können mit Parametern. Die Parameter werden in der Reihenfolge aufgelistet, in der sie nach dem Typ des Magnetfeldes erwartet werden. Alle Positionen beziehen sich auf den Koordinatenursprung der beim Zylinder in der Mitte der unteren Begrenzungsfäche liegt.

3.5.3. Ausgabedatei

Als Ausgabe liefert das Programm eine ROOT-Datei im Verzeichnis `output`, in der vier Trees gespeichert sind:

end_polarization Die Polarisation, die ein Teilchen am Ende seiner Lebenszeit hat.

start_values Die Anfangswerte von Geschwindigkeit und Ort jedes Teilchens

particle_data Zeitaufgelöste Daten (Ort, Geschwindigkeit, Polarisation, Feld) der einzelnen Teilchen

random Gleichverteilte Zufallszahlen, die am Anfang der Simulation für jedes Teilchen ermittelt werden und zur Überprüfung der Qualität der Zufallszahlen dienen. Wenn die Zahlen nicht gleichverteilt sind, wird der Zufallszahlengenerator nicht zufällig initialisiert.

Um die Ergebnisse anzusehen, muss die Datei mit ROOT⁵ geöffnet werden:

```
root output/2011-07-08-13-28-39-GMT-pid-9516.root
```

Daraufhin kann in ROOT eine graphische Benutzeroberfläche gestartet werden:

```
TBrowser b;
```

⁵<http://root.cern.ch/>

4. Testfälle der Simulation

4.1. Energieerhaltung

Ein guter Test für die korrekte Funktion des Trackers ist die Überprüfung der Energieerhaltung. Ein Neutron im Gravitationsfeld hat die Gesamtenergie

$$E = \frac{m\mathbf{v}^2}{2} + mgz . \quad (4.1)$$

Für Abbildung 4.1 wurde ein Testlauf mit Neutronen durchgeführt und die Gesamtenergie der einzelnen Teilchen geplottet. Wie man sieht, ist die Energie über die gesamte Laufzeit der Simulation konstant, der Tracker scheint also korrekt zu funktionieren.

4.2. Neutronen im homogenen Magnetfeld

Für einen einfachen Test des Integrator-Codes wurde ein Testlauf mit 1000 Neutronen bei einem Magnetfeld von $1 \mu\text{T}$ und mit einer Lebenszeit von jeweils 150 s durchgeführt. Dabei wurde als `ErrorGoal` 10^{-10} verwendet. Für die Endpolarisation ergibt sich

$$\varphi_E = 4.4289658 \text{ rad} . \quad (4.2)$$

Erwartet wird nach $\varphi_E = \omega_L t$ ein Wert von 4.4289663 rad. Wenn ein Fehler der gleichen Größenordnung bei einem Experiment mit elektrischem Feld auftritt, könnte er für ein EDM gehalten werden. Um dessen maximale Größe zu bestimmen, wird Gleichung (2.3) betrachtet. Im schlimmsten Fall (wenn der Fehler bei unterschiedlicher Richtung des elektrischen Feldes in unterschiedliche Richtungen auftreten sollte) ergibt sich ein falsches EDM von

$$d = \frac{\Delta\omega_L \hbar S}{2E} . \quad (4.3)$$

4. Testfälle der Simulation

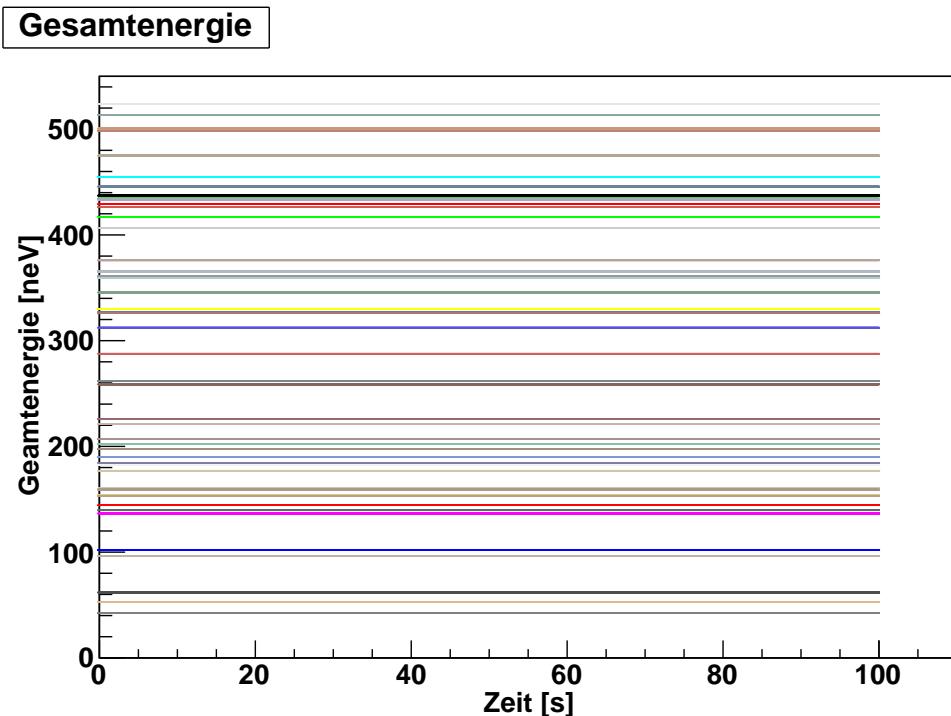


Abbildung 4.1.: Plot der Gesamtenergie einzelner Teilchen im Laufe der Zeit. Die verschiedenen Farben bezeichnen dabei die einzelnen Teilchen. Wie erwartet ist die Energie jeweils konstant. Die Dicke der Linien ist nicht durch Schwankungen, sondern durch die PDF-Ausgabe von ROOT bedingt.

4. Testfälle der Simulation

Bei einer Spannung von 100 kV und einer 12 cm hohen Kammer kann der Fehler der Simulation also höchstens zu einem falschem EDM der Größenordnung $6.6 \cdot 10^{-29}$ ecm führen. Durch eine Simulation ohne elektrisches Feld kann die Größe des Simulationsfehler auch direkt bestimmt und deshalb gut eingeschätzt werden. Außerdem ist die Annahme, dass der numerische Fehler sich bei unterschiedlichem E-Feld unterschiedlich auswirkt pessimistisch, viel wahrscheinlicher ist eine Unabhängigkeit vom E-Feld und damit eine klare Unterscheidbarkeit von einem EDM.

4.3. Quecksilber im B-Feld mit Gradient und E-Feld

Um die Entstehung geometrischer Phasen zu untersuchen wurde als letzter Test noch eine Simulation mit $B_0 = 1 \mu\text{T}$, $E = \pm 833.33 \frac{\text{kV}}{\text{m}}$ und einem Gradienten von $5 \frac{\text{nT}}{\text{m}}$ durchgeführt und mit den Ergebnissen von M. Horras (siehe Abbildung 4.2) verglichen. Aus diesen können folgende Werte für das scheinbare EDM abgelesen werden, dass durch geometrischen Phasen entsteht:

$$d_{Fit} = 4.853 \cdot 10^{-26} \text{ ecm} \quad (4.4)$$

$$d_{Theo} = 6.470 \cdot 10^{-26} \text{ ecm} \quad (4.5)$$

Bei zwei Testläufen mit entgegengesetztem elektrischen Feld ergaben sich folgende Werte für die Phase der Endpolarisation:

$$\varphi_+ = (5.363756429251884 \pm 0.012591687348896) \text{ rad} \quad (4.6)$$

$$\varphi_- = (5.363781183841419 \pm 0.011601669179826) \text{ rad} \quad (4.7)$$

Bei den angegebenen Unsicherheiten handelt es sich um die mit ROOT ermittelte Standardabweichung der Winkelverteilung. Diese ist zwar größer als der simulierte Effekt, allerdings ändert sich die Standardabweichung bei Verhundertfachung der Teilchenzahl so gut wie nicht obwohl sie bei statistischen Fehlern nach $\frac{1}{\sqrt{N}}$ um den Faktor 10 sinken müsste. Bei der Breite handelt es sich also nicht um ein statistisches Artefakt, sondern um eine Linienbreite die vermutlich durch numerische Effekte und das inhomogene Magnetfeld durch den Gradienten entsteht. Deshalb wird die Linienbreite im Folgenden nicht berücksichtigt, sondern mit den Mittelwerten gerechnet.

Aus den beiden gemessenen Polarisationen lässt sich die Verschiebung der Larmor-

4. Testfälle der Simulation

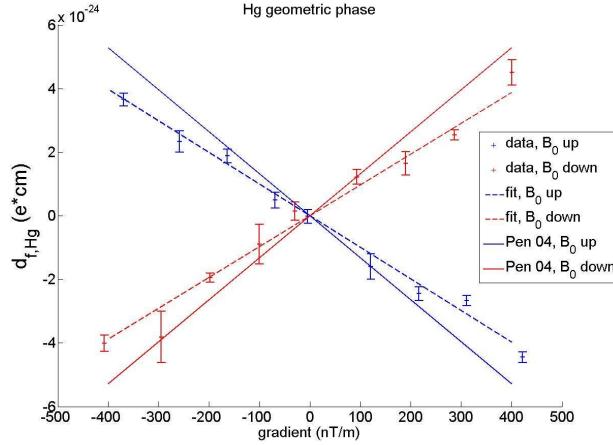


Abbildung 4.2.: Messergebnisse von M. Horras [18] zu den geometrischen Phasen von ^{199}Hg . Die im Diagramm mit „Pen 04“ bezeichneten Theoriewerte basieren auf der Arbeit von Pendlebury et al. [11].

Frequenz bestimmen:

$$\Delta\omega_L = \frac{\Delta\varphi}{T} = 2.475459 \cdot 10^{-6} \frac{\text{rad}}{\text{s}} . \quad (4.8)$$

Mit (4.4) erhält man für das scheinbare EDM:

$$d_{err} = 4.8881493 \cdot 10^{-26} \text{ ecm} . \quad (4.9)$$

Dieser Wert deckt sich sehr gut mit den gemessenen und analytisch abgeschätzten Werten von oben.

A. Kommentierter Quelltext

Im Folgenden ist der Quelltext der Simulation beigefügt. Dieser basiert auf der Arbeit von W. Feldmeier [6], dementsprechend enthalten einige Dateien noch von ihm geschriebenen Quelltext.

Folgende Klassen bzw. Funktionen stammen ursprünglich von W. Feldmeier, wurden aber von mir maßgeblich geändert und kommentiert: `main`, `Basegeometry`, `Bfield`, `Basetracking`, `Random`, `Threevector`, `Parameters`.

Außerdem wurden die Klassen `Derivatives` und `Dopr` zur Integration der Bloch-Gleichung vollständig von W. Feldmeier übernommen, zweitere stammt ursprünglich aus Ref. [17] und kann hier deshalb nicht abgedruckt werden.

Die restlichen Klassen wurden von mir neu entworfen und machen den Kern dieser Arbeit aus.

Aus Rücksicht auf die nicht-deutschsprachigen Mitarbeiter des Lehrstuhles und internationalen Gepflogenheiten folgend ist der Quelltext auf englisch verfasst und kommentiert.

Der Quelltext kann unter folgender URL in maschinenlesbarer Form bezogen werden:

https://fierlinger.wiki.tum.de/nEDM_geom_phase_code

A. Kommentierter Quelltext

A.1. Hauptfunktion

Listing A.1: main.cpp

```

1 #include <iostream>
2 #include <fstream>
3 #include <math.h>
4 #include <stdlib.h>
5 #include <string>
6 #include <vector>
7 #include <sstream>
8 #include <ctime>
9 #include <limits>
10 #include <unistd.h>
11 #include <cassert>
12 #include <cmath>
13
14 #include "TROOT.h"
15 #include "TStyle.h"
16 #include "TFile.h"
17 #include "TTree.h"
18 #include "TH1.h"
19
20 #include "globals.h"
21 #include "superpositionfield.h"
22 #include "random.h"
23 #include "tracking.h"
24 #include "dopr.h"
25 #include "derivatives.h"
26 #include "parameters.h"
27 // #include "gravitationtracker.h"
28 #include "fastcylindertracker.h"
29 #include "cylinder.h"
30 #include "debug.h"
31 #include "exceptions.h"
32 #include "timeout.h"
33
34 using namespace std;
35
36 string generateFileName(void);
37 void setTitles(TH1 &h, const char *x_title, const char *y_title);
38
39 int main(int nargs, char** argv)
40 {
41     initialize_debug();
42
43     double firsttry = 1e-6;
44
45     Parameters theParameters;
46     theParameters.expectDouble("GyromagneticRatio");
47     theParameters.expectInt("NumberOfParticles");
48     theParameters.expectDouble("Lifetime");
49     theParameters.expectDouble("ErrorGoal");
50     theParameters.expectDouble("Flipangle");
51     theParameters.expectInt("Seed");
52     theParameters.expectDouble("CylinderRadius");
53     theParameters.expectDouble("CylinderHeight");
54     theParameters.expectDouble("SaveTimeDiff");
55     // theParameters.expectDouble("TrackerStepSize");
56     theParameters.expectDouble("GravitationConstant");
57     // theParameters.expectDouble("CollisionAccuracy");
58     theParameters.expectInt("Timeout");
59     theParameters.expectInt("ParticleDataNum");
60     theParameters.expectDouble("Temperature");
61     theParameters.expectDouble("ParticleMass");
62     theParameters.expectDouble("VelocityCutoff");
63     theParameters.expectDouble("DiffusionProbability");
64     theParameters.expectDouble("MinDiffusionAngle");
65
66
67     theParameters.readParameters(cin);
68
69     const int N_particles = theParameters.getIntParam("NumberOfParticles");
70     const int save_every = N_particles / theParameters.getIntParam("ParticleDataNum") + 1;
71     const double savetimediff = theParameters.getDoubleParam("SaveTimeDiff");
72     const double lifetime = theParameters.getDoubleParam("Lifetime");
73
74     double approx_b0 = 0; // approximate B0 for expected polarization at end of programm
75
76     // Open output file
77     TFile out(generateFileName().c_str(), "new");
78
79     // Set drawing style
80     gROOT->SetStyle("Plain");
81     gStyle->SetLabelSize(0.025, "xyz");
82
83     // Trees for ROOT
84     double P_end[3]; // Polarization on end of simulation
85     double t_end; // time at end of simulation
86     TTree end_polarization("end_polarization", "Polarization_at_end_of_run");
87     end_polarization.Branch("polarization", P_end, "x/D:y:z");

```

A. Kommentierter Quelltext

```

88     end_polarization.Branch("time", &t_end, "t/D");
89
90     double random_val;
91     TTree random_tree("random", "Random_numbers_from_start_of_each_run");
92     random_tree.Branch("random", &random_val, "random/D");
93
94     float start_velocity[3];
95     float start_position[3];
96     TTree start_tree("start_values", "Place_and_velocity_at_start_of_run");
97     start_tree.Branch("velocity", start_velocity, "x:y:z");
98     start_tree.Branch("position", start_position, "x:y:z");
99
100    UInt_t particle_number; // 32 bit unsigned integer
101    float particle_time;
102    float particle_position[3];
103    float particle_velocity[3];
104    float particle_polarization[3];
105    float particle_field[3];
106
107    TTree particle_tree("particle_data", "Time_resolved_data_for_particles");
108    particle_tree.Branch("number", &particle_number, "n/i");
109    particle_tree.Branch("time", &particle_time, "t");
110    particle_tree.Branch("position", particle_position, "x:y:z");
111    particle_tree.Branch("velocity", particle_velocity, "x:y:z");
112    particle_tree.Branch("polarization", particle_polarization, "x:y:z");
113    particle_tree.Branch("bfield", particle_field, "x:y:z");
114
115    Random seed_generator(theParameters.getIntParam("Seed"));
116
117    #pragma omp parallel
118    {
119        Random *randgen;
120        #pragma omp critical
121        {
122            randgen = new Random(seed_generator.generate_seed());
123        }
124        double T = 0.0;
125        double flipangle = theParameters.getDoubleParam("Flipangle");
126        double P[3] = {-sin(flipangle/180*M_PI), 0.0, cos(flipangle/180*M_PI)};
127        double dPdt[3] = {0.0};
128        double hdid = 0.0;
129
130        Cylinder *c = new Cylinder(theParameters, randgen);
131        //GravitationTracker *tracker = new GravitationTracker(theParameters, randgen, c);
132        FastCylinderTracker *tracker = new FastCylinderTracker(theParameters, randgen, c);
133
134        Bfield *bfield = new SuperpositionField(tracker, std::string(" fields.dat"));
135
136        Derivatives * derivatives = new Derivatives(theParameters, bfield);
137        double errgoal = theParameters.getDoubleParam("ErrorGoal");
138        Dopr *stepper = new Dopr(firsttry, //initial stepsize guess
139                                3,           //dimension of ODE system
140                                P,
141                                dPdt,
142                                T,
143                                derivatives,
144                                errgoal, //relative error tolerance
145                                errgoal, //absolute error tolerance
146                                true,      //dense output?
147                                tracker
148                                );
149
150    #pragma omp for
151    for (int i = 0; i < N_particles; i++)
152    {
153        // save polarization during run because
154        // dense_out cannot be used later
155        float savetime = 0;
156        vector<float> temp_polarization_t;
157        vector<float> temp_polarization_x;
158        vector<float> temp_polarization_y;
159        vector<float> temp_polarization_z;
160
161        if (i % save_every == 0) {
162            temp_polarization_t.reserve(1.1*lifetime/savetimediff);
163            temp_polarization_x.reserve(1.1*lifetime/savetimediff);
164            temp_polarization_y.reserve(1.1*lifetime/savetimediff);
165            temp_polarization_z.reserve(1.1*lifetime/savetimediff);
166        }
167
168        try {
169            Timeout timeout(theParameters.getIntParam("Timeout"));
170
171            #pragma omp critical
172            {
173                cout << "Particle" << i << "/" << N_particles << endl;
174            }
175            #pragma omp critical
176            {
177                random_val = randgen->uniform_double(0, 1);
178                random_tree.Fill();
179            }
180        }
181    }
182
183    // write out final state
184    particle_tree.Fill();
185
186    // clean up
187    delete randgen;
188    delete derivatives;
189    delete stepper;
190    delete bfield;
191    delete c;
192
193    cout << "done" << endl;
194
195    return 0;
196}
```

A. Kommentierter Quelltext

```

179 T = 0.0;
180 int Nsteps = 0;
181 P[0] = -sin(flipangle/180*M_PI);
182 P[1] = 0.0;
183 P[2] = cos(flipangle/180*M_PI);
184 tracker->initialize();
185 approx_b0 += bfield->eval(0)[2];
186 #pragma omp critical
187 {
188     debug << "Got from_initialize: x=" << tracker->fTrackpositions[0].toString() << endl;
189     debug << "Got from_initialize: v=" << tracker->fTrackvelocities[0].toString() << endl;
190
191     for (int i = 0; i < 3; i++) {
192         assert(tracker->fTrackpositions.size() == 1);
193         start_position[i] = tracker->fTrackpositions[0][i];
194         assert(tracker->fTrackvelocities.size() == 1);
195         start_velocity[i] = tracker->fTrackvelocities[0][i];
196     }
197
198     start_tree.Fill();
199 }
200 stepper->reset(firsttry, P, dPdt, T);
201 savetime = 0;
202 debug << "Will run for " << lifetime << "seconds" << endl;
203 while(T <= lifetime)
204 {
205     timeout.check();
206
207     try
208     {
209         stepper->step();
210         Nsteps++;
211         debug << "Step " << Nsteps << " done" << endl;
212     }
213     catch(char const* error)
214     {
215         cerr << "ERROR: " << error << " at t=" << T << endl;
216         exit(1);
217     }
218
219     T = stepper->getT();
220     debug << "Stepper TIME: " << T << endl;
221     hdid = stepper->getHdid();
222     debug << "hdid=" << hdid << endl;
223
224 // save time resolved polarization to temporary storage
225 // until end of run.
226 while ((i % save_every == 0) && savetime <= T) {
227     temp_polarization_t.push_back(savetime);
228     temp_polarization_x.push_back(stepper->dense_out(0, savetime, hdid));
229     temp_polarization_y.push_back(stepper->dense_out(1, savetime, hdid));
230     temp_polarization_z.push_back(stepper->dense_out(2, savetime, hdid));
231     savetime += savetimediff;
232 }
233 } // while
234
235 // write end polarization
236 #pragma omp critical
237 {
238     // fill TTree
239     for (int j = 0; j < 3; j++) {
240         P_end[j] = stepper->dense_out(j,lifetime,hdid);
241     }
242     t_end = lifetime;
243     end_polarization.Fill();
244 }
245
246 // write time resolved data
247 if (i % save_every == 0) {
248     #pragma omp critical
249     {
250         assert(temp_polarization_x.size() == temp_polarization_y.size() && temp_polarization_y.size() ==
251             temp_polarization_z.size());
252         assert(temp_polarization_z.size() == temp_polarization_t.size());
253
254         particle_number = i; // i of particle loop
255         for (unsigned int j = 0; j < temp_polarization_t.size(); j++) {
256             particle_time = temp_polarization_t[j];
257             particle_polarization[0] = temp_polarization_x[j];
258             particle_polarization[1] = temp_polarization_y[j];
259             particle_polarization[2] = temp_polarization_z[j];
260
261             const Threvector pos = tracker->getPosition(particle_time);
262             const Threvector vel = tracker->getVelocity(particle_time);
263             const Threvector field = bfield->eval(particle_time);
264             for (int k = 0; k < 3; k++) {
265                 particle_position[k] = pos[k];
266                 particle_velocity[k] = vel[k];
267                 particle_field[k] = field[k];
268             }
269     }
270 }
```

A. Kommentierter Quelltext

```

269         particle_tree.Fill();
270     }
271     temp_polarization_t.clear();
272     temp_polarization_x.clear();
273     temp_polarization_y.clear();
274     temp_polarization_z.clear();
275   }
276 }
277
278 #pragma omp critical
279 {
280   cout << "Particle" << (i+1) << ":" << Nsteps << " steps successful," << stepper->getStepsnottaken() <<
281   " steps not taken!" << endl;
282 }
283 // try
284 catch (const Exception& e) {
285   #pragma omp critical
286   {
287     cout << "Exception for particle" << i << ":" << e.what() << endl;
288   }
289 }
290 } // for
291
292 delete randgen; randgen = 0;
293 delete stepper; stepper = 0;
294 delete derivatives; derivatives = 0;
295 delete tracker; tracker = 0;
296 delete bfield; bfield = 0;
297 delete c; c = 0;
298
299 }
300
301 cout << "Creating histograms..." << endl;
302
303 // Make some histograms
304 TH1D end_polarization_hist("end_polarization_hist", "Endpolarisation", 100, 0, 2*M_PI);
305 end_polarization.Draw("atan2(polarization.y,polarization.x)+pi>>end_polarization_hist", "", "goff,norm");
306 setTitles(end_polarization_hist, "Polarisation_[rad]", "");
307
308 TH1D start_velocity_hist("start_velocity_hist", "Anfangsgeschwindigkeit", 100, 0, 1);
309 start_velocity_hist.SetBit(TH1::kCanRebin);
310 start_tree.Draw("sqrt(velocity.x^2+velocity.y^2+velocity.z^2)>>start_velocity_hist", "", "goff,norm");
311 setTitles(start_velocity_hist, "Anfangsgeschwindigkeit_[m/s]", "");
312
313 out.Write();
314
315 approx_b0 /= N_particles;
316 cout << endl << "Run complete" << endl;
317 cout << fixed;
318 cout.precision(15);
319 cout << "=====" << endl;
320 cout << "End_polarization: (" << end_polarization_hist.GetMean() << "+-"
321 << end_polarization_hist.GetRMS() << ") "
322 rad" << endl;
323 cout << "Expected: " << fmod(-theParameters.getDoubleParam("GyromagneticRatio")*lifetime*approx_b0,
324 2*M_PI + 2*M_PI, 2*M_PI) << " rad_(approximate_value,_based_on_B0=)" << approx_b0 << ")" << endl;
325
326 return 0;
327 }
328 void setTitles(TH1 &h, const char *x_title, const char *y_title) {
329   h.GetXaxis()->SetTitle(x_title);
330   h.GetXaxis()->CenterTitle();
331   h.GetYaxis()->SetTitle(y_title);
332   h.GetYaxis()->CenterTitle();
333 }
334
335 string generateFileName(void) {
336   const size_t MAX_DATE_LEN = 100;
337   time_t rawtime;
338   char date[MAX_DATE_LEN] = "";
339
340   // get date
341   time(&rawtime);
342   strftime(date, MAX_DATE_LEN, "%Y-%m-%d-%H-%M-%S-%Z-pid-", gmtime(&rawtime));
343
344   // add pid
345   ostringstream o;
346   o << "./output/" << date << getpid() << ".root";
347   return o.str();
348 }
349 }
```

A.2. Geometrie

A. Kommentierter Quelltext

Listing A.2: basegeometry.h

```

1  #ifndef BASEGEOMETRY_H
2  #define BASEGEOMETRY_H
3
4  #include "random.h"
5  #include "threvector.h"
6  #include "polynom.h"
7
8  /**
9   * @class Basegeometry
10  * @attention This class keeps internal state, use one instance for one specific particle only!
11  */
12
13 class Basegeometry
14 {
15     public:
16         Basegeometry(Random *ran) : fRandom(ran) {};
17         virtual ~Basegeometry() {};
18
19         /**
20          * Initialize @p x to be inside the cylinder
21          */
22         virtual void initialize(Threvector &v, Threvector &x) = 0;
23
24         /**
25          * This method must return if a point x is insinde of the cylinder.
26          * @p x The point which should be checked.
27          */
28         virtual bool contains(const Threvector &x) const = 0;
29
30         /**
31          * This method should check if @p x is inside of the cylinder and
32          * set a reflection state. The reflection state is internal to the
33          * geometry class and should state at which surface a particle should
34          * be reflected when reflect is called. For example, for a cylinder
35          * the reflection state would contain if the particle is to be reflected
36          * at the top/bottom surface (z-direction) or the round cylinder wall
37          * (r-direction) or both.
38          *
39          * @param x the point for which the bounds check is to be done.
40          * @returns if the particle is outside of the cylinder and should be reflected by calling reflect.
41          * @see reflect
42          */
43         virtual bool boundsCheck(const Threvector &x) = 0;
44
45         /**
46          * This method reflects a particle based on the reflection state
47          * which has to be set beforehand by boundsCheck. This method
48          * must always be called after a prior call to boundsCheck has
49          * returned <tt>true</tt>. The parameter @p x may be changed between
50          * the two calls, the decision how to reflect must then be based on
51          * the internal reflection state and not on @p x. The reflection state
52          * has to be reset after the reflection.
53          *
54          * @param v The velocity vector which will be reflected.
55          * @param x The current position of the particle.
56          *
57          * @see boundsCheck
58          */
59         virtual void reflect(Threvector &v, const Threvector &x) = 0;
60         virtual void diffuse(Threvector &v, const Threvector &x) = 0;
61
62         virtual double findIntersection(double t0, double t1,
63                                         const Polynom &px, const Polynom &py, const Polynom &pz, double eps) = 0;
64
65     protected:
66         Random *fRandom;
67     };
68
69 #endif

```

Listing A.3: cylinder.h

```

1  #ifndef CYLINDER_H
2  #define CYLINDER_H
3
4  #include "basegeometry.h"
5  #include "threvector.h"
6  #include "parameters.h"
7
8  class FastCylinderTracker;
9
10 class Cylinder : public Basegeometry
11 {
12     public:
13         Cylinder(const Parameters& params, Random *ran);
14
15         bool contains(const Threvector &x) const;
16         bool insideHeight(const Threvector &x) const;
17         bool insideRadius(const Threvector &x) const;
18
19         bool boundsCheck(const Threvector &x);

```

A. Kommentierter Quelltext

```

20 void reflect(Threevector &v, const Threevector &x);
21 void diffuse(Threevector &v, const Threevector &x);
22 void diffuseAtSurface(Threevector &v, Threevector n);
23
24 void initialize(Threevector &v, Threevector &x);
25
26 double findIntersection(double t0, double t1,
27                         const Polynom &px, const Polynom &py, const Polynom &pz, double eps);
28
29 void reflectHeight(Threevector &v);
30 void reflectRadius(Threevector &v, const Threevector &x);
31
32 friend class FastCylinderTracker;
33
34 private:
35     double fRadius; ///radius of cylinder
36     double fRSquared; ///squared radius of cylinder
37     double fHeight; ///height of cylinder
38
39 /**
40 * Reflection state for reflect and boundsCheck.
41 * @see Basegeometry::boundsCheck
42 * @see Basegeometry::reflect
43 */
44 bool fReflectRadius;
45 /**
46 * Reflection state for reflect and boundsCheck.
47 * @see Basegeometry::boundsCheck
48 * @see Basegeometry::reflect
49 */
50 bool fReflectTop;
51 bool fReflectBottom;
52
53 double fVelocitySigma; ///sigma for maxwell distribution of velocity
54 double fCutoffSquare; ///highest possible velocity
55 double fMinDiffusionAngle; ///minimal angle after diffusion versus surface
56 };
57
58 #endif // CYLINDER_H

```

Listing A.4: cylinder.cpp

```

1 #include "cylinder.h"
2 #include "roots.h"
3 #include "debug.h"
4 #include "parameters.h"
5 #include "globals.h"
6 #include <math.h>
7 #include <iostream>
8 #include <cassert>
9
10 /**
11 * @class Cylinder
12 * Provides methods to find out if a given vector lies inside a cylinder and
13 * to reflect it at the cylinders walls.
14 * @attention This class keeps internal state, use one instance for one specific particle only!
15 */
16
17 /**
18 * Create a cylinder.
19 * @param radius radius of the cylinder
20 * @param height height of the cylinder
21 */
22 Cylinder::Cylinder(const Parameters &params, Random *ran)
23 : Basegeometry(ran), fRadius(params.getDoubleParam("CylinderRadius")),
24   fRSquared(fRadius*fRadius), fHeight(params.getDoubleParam("CylinderHeight")),
25   fReflectRadius(false), fReflectTop(false), fReflectBottom(false),
26   fVelocitySigma(sqrt(params.getDoubleParam("Temperature")*boltzmann/params.getDoubleParam("ParticleMass"))),
27   fCutoffSquare(params.getDoubleParam("VelocityCutoff")*params.getDoubleParam("VelocityCutoff")),
28   fMinDiffusionAngle(-sin(params.getDoubleParam("MinDiffusionAngle")/180.*M_PI))
29 {
30     assert(fRSquared == fRadius*fRadius);
31     debug << "New_cylinder," << fRadius << ",h=" << fHeight << ",r^2=" << fRSquared << std::endl;
32 }
33
34 void Cylinder::initialize(Threevector &v, Threevector &x) {
35     // initialize x[0] and x[1] to be inside of radius
36     do {
37         for (int i = 0; i <= 1; i++)
38             x[i] = fRandom->uniform_double(-fRadius, fRadius);
39     } while (!insideRadius(x));
40
41     assert(x[0]*x[0] + x[1]*x[1] < fRadius*fRadius);
42
43     // initialize x[2] to be inside of height
44     x[2] = fRandom->uniform_double(0, fHeight);
45
46     // initialize velocity randomly
47     do {
48         for (int i = 0; i < 3; i++)
49             v[i] = fRandom->gaussian(fVelocitySigma);
50

```

A. Kommentierter Quelltext

```

51     } while (fCutoffSquare > 0 && v.magsquare() > fCutoffSquare);
52
53     debug << "initialize:x=" << x.toString() << std::endl;
54     debug << "initialize:v=" << v.toString() << std::endl;
55 }
56
57 /**
58 * Check if the cylinder contains the point @p x.
59 */
60
61 bool Cylinder::contains(const Threvector &x) const
62 {
63     return insideRadius(x) && insideHeight(x);
64 }
65
66 /**
67 * Check if @p x is inside the height of the cylinder.
68 *
69 * This function does not check if it is inside the radius.
70 * @see Cylinder::contains(double[])
71 */
72 bool Cylinder::insideHeight(const Threvector &x) const
73 {
74     return (x[2] > 0) && (x[2] < fHeight);
75 }
76
77 /**
78 * Check if @p x is inside the radius of the cylinder.
79 *
80 * This function does not check if it is inside the height.
81 * @see Cylinder::contains(double[])
82 */
83
84 bool Cylinder::insideRadius(const Threvector &x) const
85 {
86     assert (fRSquared == fRadius*fRadius);
87     return (x[0]*x[0] + x[1]*x[1]) < fRSquared;
88 }
89
90 /**
91 * Check if particle is inside the cylinder and set reflection
92 * state accordingly. Always call reflect if this method has
93 * returned true.
94 *
95 * @param x The point for which the bounds check is made
96 * @returns if the particle is outside the bounds of the geometry
97 *
98 * @see Basegeometry::boundsCheck
99 * @see reflect
100 */
101 bool Cylinder::boundsCheck(const Threvector &x) {
102     debug << "boundsCheck_for_x=" << x.toString() << std::endl;
103     // Set reflection flag if x is outside of height or radius.
104     fReflectBottom = false;
105     fReflectTop = false;
106
107     if (x[2] < 0) {
108         fReflectBottom = true;
109     }
110     else if (x[2] > fHeight) {
111         fReflectTop = true;
112     }
113     fReflectRadius = !insideRadius(x);
114
115     debug << "fReflectTop=" << fReflectTop << ",fReflectBottom=" << fReflectBottom << ",fReflectRadius=" <<
116     fReflectRadius << std::endl;
117     // If any of the flags is set, return true to signal that reflect must be called.
118     return (fReflectRadius || fReflectTop || fReflectBottom);
119 }
120
121 /**
122 * Reflect v if the previous boundsCheck has returned true.
123 *
124 * @param[in,out] v velocity vector, will be changed if necessary.
125 * @param[in] x position of particle, read-only
126 *
127 * @see Basegeometry::reflect
128 */
129 void Cylinder::reflect (Threvector &v, const Threvector &x)
130 {
131     if (fReflectBottom || fReflectTop) {
132         reflectHeight(v);
133
134         // reset state
135         fReflectBottom = false;
136         fReflectTop = false;
137     }
138
139     if (fReflectRadius) {
140         reflectRadius(v, x);
141
142         // reset state
143         fReflectRadius = false;
144     }
}

```

A. Kommentierter Quelltext

```

145 }
146 /**
147 * Diffusion method for EquationTracker which uses the saved state in
148 * fReflectRadius, fReflectBottom and fReflectTop.
149 */
150 void Cylinder::diffuse(Threevector &v, const Threevector &pos) {
151     /// Diffusion is only implemented at the side wall, in other
152     /// cases, reflection is used instead.
153     if (fReflectTop || fReflectBottom) {
154         reflect(v, pos);
155     }
156     else {
157         assert(fReflectRadius);
158         Threevector n(pos);
159         n[2] = 0; // n points radially out of the cylinder now
160         diffuseAtSurface(v, n);
161         // reset reflection state
162         fReflectRadius = false;
163     }
164 }
165 }
166 /**
167 * Calculate the diffusion of @p v at the surface perpendicular to @p n.
168 *
169 * @param[in,out] v velocity of the particle, will be used and set to new value
170 * @param[in] n normal vector that is perpendicular to the wall and points to the outside
171 */
172 void Cylinder::diffuseAtSurface(Threevector &v, Threevector n) {
173     Threevector v_new; // New velocity
174     n.normalize();
175
176     do {
177         #pragma omp critical
178         {
179             for (int i = 0; i < 3; i++)
180                 v_new[i] = fRandom->gaussian(1);
181         }
182     } while (v_new.normalized() * n > fMinDiffusionAngle);
183     // The scalar product of @p n and the new velocity is checked to find out
184     // if the diffusion goes into the right direction.
185
186     debug << "diffuse_at_n=" << n << ",v_new=" << v_new << std::endl;
187
188     // factor which has to be multiplied to vx, vy, vz to conserve velocity
189     const double v_fact = sqrt(v.magsquare() / v_new.magsquare());
190
191     assert(fabs(v.magsquare() - (v_fact*v_new).magsquare()) < v.magsquare()*0.0001);
192     v = v_fact * v_new;
193 }
194
195 void Cylinder::reflectHeight(Threevector &v) {
196     debug << "Before_reflectHeight:v=" << v.toString() << std::endl;
197     v[2] = -v[2];
198     debug << "After_reflectHeight:v=" << v.toString() << std::endl;
199 }
200
201 void Cylinder::reflectRadius(Threevector &v, const Threevector &x) {
202     debug << "!!!_Reflecting_x-y,r=" << sqrt(x[0]*x[0]+x[1]*x[1]) << std::endl;
203     double n[2]; // unity vector perpendicular to x[1,2]
204     double sp = 0; // scalar product of v and n
205
206     // fill n
207     double x_abs = sqrt(x[0]*x[0] + x[1]*x[1]);
208     if (x_abs == 0)
209         throw "Cylinder::reflectRadius_called_with_x[0]^2+x[1]^2=0";
210     n[0] = x[1] / x_abs;
211     n[1] = -x[0] / x_abs;
212
213     // Calculate sp
214     for (int i = 0; i <= 1; i++) {
215         sp += v[i]*n[i];
216     }
217
218     // set v to reflected value
219     for (int i = 0; i <= 1; i++) {
220         v[i] = -v[i] + 2*n[i]*sp;
221     }
222 }
223
224 double Cylinder::findIntersection(double t0, double t1, const Polynom &px, const Polynom &py, const Polynom &pz, double eps)
225 {
226     assert(eps > 0);
227
228     bool try_again = true;
229     one_more_try:
230     double t_height = INFINITY;
231     double t_radius = INFINITY;
232     debug << "Looking_for_intersection_in[" << t0 << "," << t1 << "]" << std::endl;
233
234     try {
235         if (fReflectRadius) {

```

A. Kommentierter Quelltext

```

237     // Polynomial for radius:  $x^2 + y^2 - r^2 == 0$ 
238     debug << "Constructing  $r^2(t)$ .polynomial from px=" << px.toString() << " and py=" << py.toString() << " with_offset_"
239     << fRSquared << std::endl;
240     Polynom rint(px*px + py*py); // Calculate  $x^2 + y^2$ 
241     rint[0] -= fRSquared; // Apply offset
242     debug << "Looking_for_radius_intersection, r^2(t) = " << rint.toString() << std::endl;
243
244     // Find intersection time
245     t_radius = Roots::safeNewton(rint, rint.derivative(), t0, t1, eps);
246 }
247
248 if (fReflectBottom) {
249     debug << "Looking_for_bottom_intersection, z(t) = " << pz.toString() << std::endl;
250     t_height = Roots::safeNewton(pz, pz.derivative(), t0, t1, eps);
251 }
252
253 if (fReflectTop) {
254     debug << "Looking_for_top_intersection, z(t) = " << pz.toString() << std::endl;
255     Polynom top(pz);
256     top[0] -= fHeight; // Apply offset
257     t_height = Roots::safeNewton(top, top.derivative(), t0, t1, eps);
258 }
259
260 catch (Roots::NoRoot) {
261     if (try_again) {
262         // Try again once with extended Interval
263         try_again = false;
264         const double delta = (t1-t0)/2;
265         assert(delta > 0);
266         t0 -= delta;
267         t1 += delta;
268         debug << "NO_ROOT: Trying again with extended interval [" << t0 << "," << t1 << "]"
269         goto one_more_try;
270     }
271     else {
272         throw;
273     }
274 }
275
276 if (t_radius < t_height)
277     return t_radius;
278 else
279     return t_height;
}

```

A.3. Magnetfeld

Listing A.5: bfield.h

```

1 #ifndef _BFIELD_H
2 #define _BFIELD_H
3
4 #include "parameters.h"
5 #include "basetracking.h"
6 #include "threvector.h"
7
8 class Bfield
9 {
10     public:
11     Bfield(Basetracking *tracker) : fTracker(tracker) {};
12     virtual ~Bfield() {};
13     Threvector operator()(const double time) const { return eval(time); };
14     virtual Threvector eval(const double time) const = 0;
15
16     protected:
17     Basetracking *fTracker;
18 };
19
20 #endif // _BFIELD_H

```

Listing A.6: basicfields.h

```

1 #ifndef _BASICFIELDS_H
2 #define _BASICFIELDS_H
3
4 #include <string>
5 #include <cmath>
6
7 #include "bfield.h"
8 #include "threvector.h"
9 #include "globals.h"
10
11 class DipoleField : public Bfield

```

A. Kommentierter Quelltext

```

12  {
13      public:
14          DipoleField(Basetracking *t, const Threvector m, const Threvector pos) :
15              Bfield(t), m(m), dr(-pos) {};
16          Threvector eval(const double time) const {
17              /// r is set to the vector connecting the position of the dipole x0 with the
18              /// current position of particle x.
19              /// |f| / vec r = |vec x - |vec x_0 |f|
20              Threvector r = fTracker->getPosition(time) + dr;
21              const double r2 = r.magsquare();
22              const double r5 = r2*r2*r.mag();
23              // 1e-7 = mu0 / 4pi
24              return (1e-7/r5) * (3*r*(m*r)-m*r2);
25      };
26      private:
27          const Threvector m, dr;
28  };
29
30
31 class HomogenousMagneticField : public Bfield
32 {
33     public:
34         HomogenousMagneticField(Basetracking *t, const Threvector B) :
35             Bfield(t), B(B) {};
36         Threvector eval(const double time) const {
37             return B;
38         };
39     private:
40         const Threvector B;
41     };
42
43 class RelativisticField : public Bfield
44 {
45     public:
46         RelativisticField (Basetracking *t, const Threvector E) :
47             Bfield(t), E(E) {};
48         Threvector eval(const double time) const {
49             const Threvector v = fTracker->getVelocity(time);
50             return (-1/(speed_of_light*speed_of_light)) * v.cross(E);
51         };
52     private:
53         const Threvector E;
54     };
55
56 class GradientField : public Bfield
57 {
58     public:
59         GradientField(Basetracking *t, const double gradient) :
60             Bfield(t), g(gradient) {};
61         Threvector eval(const double time) const {
62             const Threvector pos = fTracker->getPosition(time);
63             const double r = sqrt(pos[0]*pos[0]+pos[1]*pos[1]);
64             const double radial_part = r*g/2.;
65
66             return Threvector(radial_part, radial_part, g*pos[2]);
67         };
68     private:
69         const double g;
70     };
71
72 /**
73 * LaserField is a homogenous magnetic field inside a tube in
74 * x direction with radius r. Outside the tube, there is no field.
75 */
76 * @param[in] t pointer to tracker instance
77 * @param[in] B field inside the tube
78 * @param[in] height height of center of tube above z = 0
79 * @param[in] radius radius of the tube
80 *
81 * @return B if inside of tube, 0 else
82 */
83 class LaserField : public Bfield
84 {
85     public:
86         LaserField(Basetracking *t, const Threvector B, const double height, const double radius) :
87             Bfield(t), B(B), h(height), r2(radius*radius), ZERO(0,0,0) {};
88         Threvector eval(const double time) const {
89             const Threvector pos = fTracker->getPosition(time);
90             const double y = pos[1];
91             const double z = pos[2] - h;
92
93             if (y*y+z*z <= r2)
94                 return B;
95             else
96                 return ZERO;
97         };
98     private:
99         const Threvector B;
100        const double h, r2;
101        const Threvector ZERO;
102    };
103
104 #endif // _BASICFIELDS_H

```

A. Kommentierter Quelltext

Listing A.7: superpositionfield.h

```

1 #ifndef _SUPERPOSITIONFIELD_H
2 #define _SUPERPOSITIONFIELD_H
3
4 #include <string>
5 #include <vector>
6
7 #include "bfield.h"
8 #include "basetracking.h"
9
10 class SuperpositionField : public Bfield
11 {
12     public:
13         SuperpositionField(Basetracking *t, std::string field_file);
14         virtual ~SuperpositionField();
15         Threemode eval(const double time) const;
16     private:
17         void readFields(std::string filename);
18         std::vector<Bfield*> fFields;
19     };
20
21 #endif // _SUPERPOSITIONFIELD_H

```

Listing A.8: superpositionfield.cpp

```

1 #include <fstream>
2 #include <iostream>
3 #include <cstdlib>
4 #include <string>
5 #include <sstream>
6
7 #include "superpositionfield.h"
8 #include "basicfields.h"
9
10 SuperpositionField::SuperpositionField(Basetracking *t, std::string field_file) : Bfield(t)
11 {
12     readFields(field_file);
13 }
14
15 Threemode SuperpositionField::eval(const double time) const
16 {
17     Threemode ret;
18
19     for (unsigned int i = 0; i < fFields.size(); i++) {
20         ret += fFields[i] -> eval(time);
21     }
22
23     return ret;
24 }
25
26 void SuperpositionField::readFields(std::string fname)
27 {
28     // open file
29     std::ifstream f(fname.c_str());
30
31     // check for success
32     if (!f.is_open() || !f.good()) {
33         std::cerr << "Could not open field file:" << fname << std::endl;
34         exit(1);
35     }
36
37     // read lines
38     while (!f.eof()) {
39         std::string sline;
40         getline(f, sline); // read one line
41         std::stringstream line(sline);
42
43         /// The first token in the line is the type
44         std::string type;
45         line >> type;
46
47         if (type.size() == 0) {
48             /// Empty lines are ignored
49         } else if (type[0] == '#') {
50             /// Comments begin with a # and are only allowed on a
51             /// line on their own.
52         } else if (type == "D") {
53             /// D is a magnetic dipole. The first three parameters are
54             /// the magnetic moment, the next three the position of the
55             /// dipole.
56             Threemode m;
57             Threemode pos;
58
59             line >> m;
60             line >> pos;
61
62             fFields.push_back(new DipoleField(fTracker, m, pos));
63
64     }
65 }
```

A. Kommentierter Quelltext

```

66     std::cout << "Added_magnetic_dipole_with_m=" << m << "A_m^2" << "and_position" << pos << "m" <<
67     std::endl;
68 } else if (type == "E") {
69     /// E is a homogenous electric field that causes relativistic effects.
70     /// It takes three doubles as arguments
71     Threevector E;
72     line >> E;
73
74     fFields.push_back(new RelativisticField(fTracker, E));
75
76     std::cout << "Added_electric_field_with_E=" << E << "V/m" << std::endl;
77 }
78 else if (type == "H") {
79     /// H is an homogenous magnetic field, just like E. It takes the
80     /// vector B as three doubles as argument.
81     Threevector B;
82     line >> B;
83
84     fFields.push_back(new HomogenousMagneticField(fTracker, B));
85
86     std::cout << "Added_homogenous_magnetic_field_B=" << B << "T" << std::endl;
87 }
88 else if (type == "L") {
89     /// L for "laser" is a homogenous magnetic field inside a tube (the laser beam).
90     /// It takes the field as three doubles, the height of the center of the tube and
91     /// the radius of the tube as parameters
92     Threevector B;
93     double h, r;
94     line >> B;
95     line >> h;
96     line >> r;
97
98     fFields.push_back(new LaserField(fTracker, B, h, r));
99
100    std::cout << "Added_laser_field_B=" << B << "T" << "inside_of_cylinder_with_center_at_h=" << h
101    << "m_and_radius_of_" << r << "m" << std::endl;
102 }
103 else if (type == "G") {
104     /// G is a gradient field and takes only the gradient in z direction
105     /// as parameter
106     double gradient;
107     line >> gradient;
108
109     fFields.push_back(new GradientField(fTracker, gradient));
110
111     std::cout << "Added_gradient_field_B=(r*g/r, r*g/2, g*z) with_g=" << gradient << std::endl;
112 }
113 else {
114     /// The program is aborted if an unknown entry is encountered.
115     std::cerr << "Can't_parse_line'" << sline << "'in'" << fname << std::endl;
116     exit(1);
117 }
118 }
119 }
120 SuperpositionField::~SuperpositionField() {
121     for (unsigned int i = 0; i < fFields.size(); i++)
122         delete fFields[i];
123 }
124 }
```

A.4. Flugbahn

Listing A.9: basetracking.h

```

1 #ifndef BASETRACKING_H
2 #define BASETRACKING_H
3
4 #include <vector>
5 #include "random.h"
6 #include "parameters.h"
7 #include "basegeometry.h"
8 #include "threevector.h"
9
10 class Basetracking
11 {
12     public:
13     Basetracking(Random* ran, Basegeometry *geo);
14     virtual ~Basetracking();
15
16     virtual void initialize();
17     virtual Threevector getPosition(double time) = 0;
18     virtual Threevector getVelocity(double time) = 0;
19
20     /**
21      * Pure virtual function
22      * Creates a track with length @p hmax in time
23 }
```

A. Kommentierter Quelltext

```

23     * This method must be provided by a user tracking class
24     */
25
26     virtual void makeTrack(double t_start, double h) = 0;
27     virtual void reset();
28     virtual void stepDone(double time) {};
29
30     std::vector<double> fTracktimes;
31     std::vector<Threemode> fTrackpositions;
32     std::vector<Threemode> fTrackvelocities;
33
34     protected:
35     Random *fRandomgenerator;
36     Basegeometry *fGeometry;
37     int fIndex;
38     double fLasttime;
39     double fstarttime;
40 };
41 #endif

```

Listing A.10: basetracking.cpp

```

1  #include <cassert>
2  #include "basetracking.h"
3
4  /**
5   * @class Basetracking
6   * Abstract base class, derive from it to create an actual user tracking class.
7   * Provides methods to initialize the starting point of a track portion,
8   * to reset the tracking if a timestep was not accepted and to mark the track
9   * as finished and prepare for a new one.
10  */
11
12 /**
13 *
14 * @param ran Pointer to the random generator
15 * @param geo Pointer to the geometry object
16 */
17
18 Basetracking::Basetracking(Random *ran, Basegeometry *geo)
19 : fRandomgenerator(ran), fGeometry(geo), fIndex(0),
20   fLasttime(0), fstarttime(0)
21 {
22 }
23
24 Basetracking::~Basetracking()
25 {
26 }
27 /**
28  * Initializes the track portion. Generates a starting point and sets the
29  * starting time to zero
30 */
31 void Basetracking::initialize ()
32 {
33     // clear anything that may be left over from last run
34     fTracktimes.clear();
35     fTrackvelocities .clear();
36     fTrackpositions .clear();
37
38     fLasttime = fstarttime = 0.0;
39     Threemode pos;
40     Threemode vel;
41     fGeometry->initialize(vel, pos);
42
43     assert (fGeometry->contains(pos));
44
45     fTracktimes.push_back(0.0);
46     fTrackpositions.push_back(pos);
47     fTrackvelocities .push_back(vel);
48
49     assert ( fTrackvelocities [0] == vel);
50     assert (fTrackpositions[0] == pos);
51     assert (fTracktimes[0] == 0);
52 }
53
54 /**
55  * Resets the already constructed track portion to its starting point.
56  * This method should be called if the timestep was not accepted.
57 */
58 void Basetracking::reset ()
59 {
60     fIndex = 0;
61     fLasttime = fstarttime;
62 }

```

Listing A.11: equationtracker.cpp

```

1  #include "equationtracker.h"
2  #include "debug.h"

```

A. Kommentierter Quelltext

```

3 #include "parameters.h"
4 #include "interpolationpolynomial.h"
5
6 #include <memory>
7
8 /**
9 * Construct a new @p EquationTracker Object. Do not use it
10 * without calling initialize(void) first!
11 * @param ran random number generator
12 * @param geo geometry
13 */
14 EquationTracker::EquationTracker(const Parameters &params, Random *ran, Basegeometry *geo) :
15     Basetracking(ran, geo), fStepSize(0.01), fNewtonEps(params.getDoubleParam("CollisionAccuracy")),
16     fTime(0), fTimeout(params.getIntParam("Timeout")), fDiffuseProbability(params.getDoubleParam("DiffusionProbability"))
17 {
18     fStepSize = params.getDoubleParam("TrackerStepSize");
19 }
20
21 /**
22 * Initialize the EquationTracker. Must be called exactly once before any
23 * other method of EquationTracker is called.
24 */
25 void EquationTracker::initialize () {
26     fTimeout.reset();
27     fTime = 0;
28     Basetracking::initialize ();
29     fPos = fTrackpositions.back();
30     fVel = fTrackvelocities.back();
31 }
32
33 /**
34 * Do one step of the Runge-Kutta algorithm.
35 *
36 * @param[in] h size of time step that should be taken
37 * @param[in] t current time
38 * @param[in,out] x current location, will be updated to new location
39 * @param[in,out] v current velocity, will be updated to new velocity
40 */
41 void EquationTracker::rkStep(const double &h, const double &t, Threvector &x, Threvector &v) {
42     Threvector k1x, k2x, k3x, k4x;
43     Threvector k1v, k2v, k3v, k4v;
44
45     debug << "i\tkix\tkiv" << std::endl;
46     derivs(t, x, v, k1x, k1v);
47     k1x *= h;
48     k1v *= h;
49     debug << "1\t" << k1x.toString() << "\t" << k1v.toString() << std::endl;
50     derivs(t + .5*h, x + .5*k1x, v + .5*k1v, k2x, k2v);
51     k2x *= h;
52     k2v *= h;
53     debug << "2\t" << k2x.toString() << "\t" << k2v.toString() << std::endl;
54     derivs(t + .5*h, x + .5*k2x, v + .5*k2v, k3x, k3v);
55     k3x *= h;
56     k3v *= h;
57     debug << "3\t" << k3x.toString() << "\t" << k3v.toString() << std::endl;
58     derivs(t + h, x + k3x, v + k3v, k4x, k4v);
59     k4x *= h;
60     k4v *= h;
61     debug << "4\t" << k4x.toString() << "\t" << k4v.toString() << std::endl;
62
63     x += (1./6.)*k1x + (1./3.)*k2x + (1./3.)*k3x + (1./6.)*k4x;
64     v += (1./6.)*k1v + (1./3.)*k2v + (1./3.)*k3v + (1./6.)*k4v;
65 }
66
67 /**
68 * Generate track up to t_start + h
69 */
70 void EquationTracker::makeTrack(const double t_start, double h) {
71     debug << "makeTrack(" << t_start << ", " << h << ")" << std::endl;
72     // time until which the solution is to be calculated
73     const double goal = t_start + h;
74
75     // Generate track as far as necessary
76     while (fTime < goal) {
77         fTimeout.check();
78
79         debug << "TIME: " << fTime << ", dt=" << fStepSize << std::endl;
80         debug << "Before_step: " << fPos.toString() << ", speed=" << fVel.toString() << std::endl;
81         // Do one Runge-Kutta step
82         rkStep(fStepSize, fTime, fPos, fVel);
83
84         debug << "After_step: " << fPos.toString() << ", speed=" << fVel.toString() << std::endl;
85         // Check if the particle has left the volume during the step
86         if (fGeometry->boundsCheck(fPos)) {
87             // If the particle left the volume during the step, backtrack to the intersection time,
88             // do the reflection and save position and velocity before and after the collision.
89             debug << "COLLISION: " << fPos.toString() << std::endl;
90
91             // Save current vectors and time for interpolation polynomial
92             Threvector pos1(fPos);
93             Threvector vel1(fVel);
94             double t1 = fTime + fStepSize;
95

```

A. Kommentierter Quelltext

```

96     // Reset fPos and fVel to values before rkStep
97     fPos = fTrackpositions.back();
98     fVel = fTrackvelocities.back();
99
100    // Create interpolation polynomials for x, y and z
101    InterpolationPolynomial px(fTime, fPos[0], fVel[0], t1, pos1[0], vel1[0]);
102    InterpolationPolynomial py(fTime, fPos[1], fVel[1], t1, pos1[1], vel1[1]);
103    InterpolationPolynomial pz(fTime, fPos[2], fVel[2], t1, pos1[2], vel1[2]);
104
105    // Find the time when the particle left the volume
106    double t_coll = fGeometry->findIntersection(fTime, t1, px, py, pz, fNewtonEps);
107
108    // Do smaller step and advance time
109    rkStep(t_coll - fTime, fTime, fPos, fVel);
110    fTime = t_coll;
111
112    debug << "AT_COLLISION:" << fPos.toString() << "_speed_" << fVel.toString() << std::endl;
113    // Save velocity and position before collision
114    fTracktimes.push_back(fTime);
115    fTrackpositions.push_back(fPos);
116    fTrackvelocities.push_back(fVel);
117
118    // Randomly decide whether to diffuse or reflect
119    double rand;
120    #pragma omp critical
121    {
122        rand = fRandomgenerator->uniform();
123        assert(rand <= 1 && rand >= 0);
124    }
125
126    if (rand > fDiffuseProbability) {
127        // Reflect
128        fGeometry->reflect(fVel, fPos);
129    }
130    else {
131        // Diffuse scattering
132        fGeometry->diffuse(fVel, fPos);
133    }
134
135    else {
136        // Step was still inside, just advance time
137        fTime += fStepSize;
138    }
139
140    // Save new trackpoint (in case of collision the one after the refelction)
141    fTracktimes.push_back(fTime);
142    fTrackpositions.push_back(fPos);
143    fTrackvelocities.push_back(fVel);
144    debug << "At_end_of_step:" << fPos.toString() << "_speed_" << fVel.toString() << std::endl;
145}
146
147
148 Threevector EquationTracker::getPosition(double time) {
149     assert(fTracktimes.size() == fTrackvelocities.size() && fTrackvelocities.size() == fTrackpositions.size());
150
151     if (fTrackpositions.size() == 1) {
152         return fTrackpositions[0];
153     }
154     assert(fTrackpositions.size() > 1);
155
156     static bool initialized = false;
157     static Polynom px(2), py(2), pz(2);
158     static double tmin = 0.0;
159     static double tmax = 0.0;
160
161     if (!initialized || time < tmin || time > tmax) {
162         unsigned int l = 0;
163         unsigned int u = fTracktimes.size() - 1;
164         assert(u > 0);
165
166         // Find time by bisection
167         while (u - l > 1) {
168             assert(u > l);
169
170             int i = l + (u-l)/2;
171
172             if (fTracktimes[i] > time)
173                 u = i;
174             else // (fTracktimes[i] <=
175                 l = i;
176         }
177
178         // set new limits
179         assert(l < fTracktimes.size() && u < fTracktimes.size());
180         double min = fTracktimes[l];
181         double max = fTracktimes[u];
182         assert(max > min);
183
184         // generate new interpolation polynomial
185         px = InterpolationPolynomial(min, fTrackpositions[l][0], fTrackvelocities[1][0],
186                                     max, fTrackpositions[u][0], fTrackvelocities[u][0]);
187         py = InterpolationPolynomial(min, fTrackpositions[l][1], fTrackvelocities[1][1],

```

A. Kommentierter Quelltext

```

188     max, fTrackpositions[u ][1], fTrackvelocities [u ][1]);
189     pz = InterpolationPolynomial(min, fTrackpositions[l ][2], fTrackvelocities [l ][2],
190     max, fTrackpositions[u ][2], fTrackvelocities [u ][2]);
191
192     initialized = true;
193 }
194
195 return Threemodevector(px(time), py(time), pz(time));
196 }
197
198 /**
199 * This function linearly interpolates the velocity the particle has at the time
200 * given as a parameter.
201 */
202 Threemodevector EquationTracker::getVelocity(double time) {
203     assert( fTrackvelocities .size() == fTrackpositions.size() && fTracktimes.size() == fTrackpositions.size());
204     unsigned int l = 0;
205     unsigned int u = fTracktimes.size() - 1;
206     assert(u >= 0);
207
208     if (u == 0) // only one entry yet
209         return fTrackvelocities [0];
210     assert(u > 0);
211
212     // Find time by bisection
213     while (u - l > 1) {
214         assert(u > l);
215         const unsigned int i = l + (u-l)/2;
216
217         if (fTracktimes[i] > time)
218             u = i;
219         else // (fTracktimes[i] <=)
220             l = i;
221     }
222     assert(u - l == 1);
223     assert(l >= 0 && u < fTracktimes.size());
224
225     // Interpolate
226     const Threemodevector dv = fTrackvelocities[u] - fTrackvelocities [l];
227     const double dt = fTracktimes[u] - fTracktimes[l];
228     assert(dt > 0);
229     const double t = time - fTracktimes[l];
230     assert(t >= 0);
231     assert(t <= fTracktimes[u] - fTracktimes[l]);
232     const Threemodevector v0 = fTrackvelocities[l];
233
234     return (t/dt)*dv + v0;
235 }
236
237 EquationTracker::~EquationTracker()
238 {
239 }
```

Listing A.12: equationtracker.h

```

1 #ifndef _EQUATIONTRACKER_H
2 #define _EQUATIONTRACKER_H
3
4 #include "threemodevector.h"
5 #include "basetracking.h"
6 #include "basegeometry.h"
7 #include "interpolationpolynomial.h"
8 #include "timeout.h"
9
10 /**
11 * @class EquationTracker
12 * Calculate the track of a particle by solving the equation of motion.
13 * This class is an abstract base class, implementations have to overwrite
14 * the derivs method.
15 */
16 class EquationTracker : public Basetracking {
17     public:
18         EquationTracker(const Parameters&, Random *ran, Basegeometry *geo);
19         void makeTrack(double t_start, double h);
20         void initialize();
21         Threemodevector getPosition(double time);
22         Threemodevector getVelocity(double time);
23         virtual ~EquationTracker();
24
25     private:
26         void rkStep(const double &h, const double &t, Threemodevector &x, Threemodevector &v);
27         double fStepSize; ///// stepsize of the Runge-Kutta integration
28
29         const double fNewtonEps; ///// accuracy of roots found with safeNewton
30
31         // State for makeTrack()
32         double fTime;
33         Threemodevector fPos;
34         Threemodevector fVel;
35
36         // Timeout
```

A. Kommentierter Quelltext

```

37     Timeout fTimeout;
38
39     double fDiffuseProbability;
40
41 protected:
42     /**
43      * Provides the derivatives @p xdot and @p vdot of time and velocity.
44      * Override this pure virtual method to provide motional equations.
45      *
46      * @attention The parameters @p xdot and @p vdot are not initialized
47      * to zero, so you have to overwrite all components!
48      */
49     * @param[in] t    current time
50     * @param[in] x    current location
51     * @param[in] v    current velocity
52     * @param[out] xdot \f$ \dot{x} \f$, usually just \f$ \dot{x} \f$ = v\f$ \dot{v} \f$, <b>not initialized to zero</b>
53     * @param[out] vdot \f$ \dot{v} \f$, usually \f$ \dot{v} \f$ = \frac{F}{m} \f$ (Newton), <b>not initialized to zero</b>
54     */
55     virtual void derivs(const double t, const Threvector x, const Threvector v, Threvector &xdot, Threvector &vdot) = 0;
56 };
57
58 #endif // _EQUATIONTRACKER_H

```

Listing A.13: fastcylindertracker.cpp

```

1  #include <cassert>
2  #include <limits>
3  #include <cmath>
4  #include <algorithm>
5  #include <gsl/gsl_poly.h>
6
7  #include "fastcylindertracker.h"
8  #include "exceptions.h"
9
10 FastCylinderTracker::FastCylinderTracker(const Parameters &params, Random* ran, Cylinder *geo) :
11     Basetracking(ran, geo), iGeometry(geo), H(params.getDoubleParam("CylinderHeight")),
12     R(params.getDoubleParam("CylinderRadius")), R2(R*R), g(params.getDoubleParam("GravitationConstant")),
13     fLastCollisionSurface(None), fDiffuseProbability(params.getDoubleParam("DiffusionProbability")) {}
14
15 FastCylinderTracker::~FastCylinderTracker()
16 {
17 }
18
19 void FastCylinderTracker::initialize()
20 {
21     fLastCollisionSurface = None;
22     Basetracking::initialize();
23 }
24
25 Threvector FastCylinderTracker::getPosition(double t)
26 {
27     static Threvector res; // static for caching
28     static double t_cached = std::numeric_limits<double>().quiet_NaN(); // time for which result is cached
29
30     // calculate new result, if the cached one
31     if (t != t_cached) {
32         const unsigned int i = findIndex(t);
33         const Threvector &x = fTrackpositions[i];
34         const Threvector &v = fTrackvelocities[i];
35         const double dt = t - fTracktimes[i];
36         assert(dt >= 0);
37
38         res[0] = x[0] + v[0]*dt;
39         res[1] = x[1] + v[1]*dt;
40         res[2] = x[2] + dt*(-0.5*g*dt + v[2]);
41         t_cached = t;
42     }
43
44     return res;
45 }
46
47 Threvector FastCylinderTracker::getVelocity(double t)
48 {
49     const unsigned int i = findIndex(t);
50     const Threvector &v = fTrackvelocities[i];
51     const double t0 = fTracktimes[i];
52     assert(t >= t0);
53
54     return Threvector(v[0], v[1], v[2] - g*(t-t0));
55 }
56
57 void FastCylinderTracker::makeTrack(double t_start, double h)
58 {
59     // get position and time of last intersection
60     Threvector pos = fTrackpositions.back();
61     Threvector vel = fTrackvelocities.back();
62     double t = fTracktimes.back();
63
64     while (t < t_start + h) {
65         debug << "t=" << t << ",x=" << pos << ",v=" << vel << std::endl;
66         /**

```

A. Kommentierter Quelltext

```

67      * The coefficients used for the polynomials
68      * are described in Roman's thesis.
69      * The polynomials all have the form
70      * |f| ax^2 + bx + c |
71      */
72 double a, b, c;
73
74 // collision times
75 std::vector<std::pair<double, Surface>> collisions;
76 collisions.reserve(6); // allocate enough place for six values
77
78 // DON'T change the order of these!
79 // collision at bottom: z(t) = 0
80 a = -0.5*g;
81 b = g*t + vel[2];
82 c = t*(a*t - vel[2]) + pos[2];
83 addSolutions(collisions, a, b, c, t, Bottom);
84 // collision at top: z(t) - H = 0
85 c -= H;
86 addSolutions(collisions, a, b, c, t, Top);
87
88 // collision at radius: x(t)*x(t)*y(t)*y(t) - R^2 = 0
89 a = vel[0]*vel[0] + vel[1]*vel[1];
90 const double temp = 2.0*(vel[0]*pos[0] + vel[1]*pos[1]);
91 b = -2.*t*a + temp;
92 c = -R*R + t*(t*a - temp) + pos[0]*pos[0] + pos[1]*pos[1];
93 addSolutions(collisions, a, b, c, t, Radius);
94
95 // particle has left the volume
96 if (collisions.size() <= 0)
97     throw LeftVolume();
98
99 debug << "Looking for smallest time in future..." << std::endl;
100
101 // Now find the time of the collision which is smallest time
102 // which is greater than the time of the last collision.
103 // At the same time, the surface of the last collision is
104 // set accordingly.
105 fLastCollisionSurface = None;
106 double t_coll = std::numeric_limits<double>().max();
107 for (unsigned int i = 0; i < collisions.size(); i++) {
108     const double candidate = collisions[i].first;
109     if (candidate > t && candidate < t_coll) {
110         t_coll = candidate;
111         fLastCollisionSurface = collisions[i].second;
112     }
113 }
114 assert(t_coll != std::numeric_limits<double>().max()); // was changed
115 assert(fLastCollisionSurface != None);
116 assert(t_coll > t);
117 debug << "Collision at t=" << t_coll << std::endl;
118
119 // now go to the place of the collision
120 t = t_coll;
121 pos = getPosition(t);
122 vel = getVelocity(t);
123
124 // and reflect or scatter
125 double rand;
126 #pragma omp critical
127 {
128     rand = fRandomgenerator->uniform();
129     assert(rand <= 1 && rand >= 0);
130 }
131 if (rand > fDiffuseProbability) {
132     assert(fLastCollisionSurface != None);
133     if (fLastCollisionSurface == Radius)
134         fGeometry->reflectRadius(vel, pos);
135     else // Top or Bottom
136         fGeometry->reflectHeight(vel);
137 }
138 else {
139     // diffuse scattering
140     Threevector n; // normal vector for diffusion
141     assert(n.magsquare() == 0);
142     switch (fLastCollisionSurface) {
143         case Top:
144             n[2] = 1.0;
145             break;
146         case Bottom:
147             n[2] = -1.0;
148             break;
149         case Radius:
150             n = pos;
151             n[2] = 0;
152             break;
153         case None:
154             assert(false);
155             break;
156     }
157     fGeometry->diffuseAtSurface(vel, n);
158 }
159

```

A. Kommentierter Quelltext

```

160      // save new trackpoint
161      fTracktimes.push_back(t);
162      fTrackpositions.push_back(pos);
163      fTrackvelocities.push_back(vel);
164  }
165 }
166 */
167 /**
168 * Find the solutions of  $ax^2 + bx + c = 0$  and add them to the vector @p v.
169 * The function will take care that only the absolutely greater value is added if
170 * the surface is the same as the surface of last reflection.
171 *
172 * @param [in] current the surface on which the reflection takes place
173 */
174 inline void FastCylinderTracker::addSolutions(std::vector< std::pair< double, Surface> > &v, const double a, const double b, const
175     double c,
176     const double t0, const Surface current) const
177 {
178     double x1, x2; // used to store results
179     debug << "Looking_for_solutions_at_" << current << ", last_was_" << fLastCollisionSurface << std::endl;
180     // use GSL to solve quadratic equation#
181     // nresults will hold the number of results that were obtained
182     int nresults = gsl_poly_solve_quadratic(a, b, c, &x1, &x2);
183     debug << "Got_" << nresults << "roots:" << x1 << "," << x2 << std::endl;
184     if (current == fLastCollisionSurface) {
185         assert(current != None);
186         /// If the current surface is the surface of the last reflection, we
187         /// need to drop the smaller solution which will be equal to zero, but
188         /// only approximately.
189         if (nresults == 2) {
190             // find the value with the absolute value nearest to zero
191             // and make it x2.
192             if (fabs(x1 - t0) < fabs(x2 - t0))
193                 std::swap(x1, x2);
194
195             // add x1 to results
196             v.push_back(std::make_pair(x1, current));
197
198             // check x2
199             assert(fabs(x2 - t0) < 1e-5); // quite lax check
200         }
201         else {
202             // if the equation has less than two solutions, it should
203             // have exactly one which is close to zero.
204             assert(nresults == 1);
205             assert(fabs(x1) < 1e-5); // quite lax check
206         }
207     }
208     else {
209         // If it is another surface, just add all results.
210         if (nresults >= 1)
211             v.push_back(std::make_pair(x1, current));
212         if (nresults == 2)
213             v.push_back(std::make_pair(x2, current));
214     }
215 }
216 */
217
218 unsigned int FastCylinderTracker::findIndex(const double time)
219 {
220     assert(fTrackvelocities.size() == fTrackpositions.size() && fTrackpositions.size() == fTracktimes.size());
221     // cache last result
222     static unsigned int last_index = -1;
223
224 #ifndef NDEBUG
225     for (unsigned int j = 1; j < fTracktimes.size(); j++)
226         assert(fTracktimes[j - 1] < fTracktimes[j]);
227 #endif
228
229     if (fTracktimes.back() <= time) {
230         last_index = fTracktimes.size() - 1;
231         return last_index;
232     }
233     assert(fTracktimes.size() > 1);
234
235     debug << "last_index=" << last_index << "#trackpoints=" << fTracktimes.size() << std::endl;
236     // short circuit if cached result is still valid
237     if (last_index >= 0 && fTracktimes[last_index] <= time && fTracktimes[last_index + 1] > time) {
238         assert(last_index < fTracktimes.size());
239         return last_index;
240     }
241
242     // bounds of interval
243     unsigned int u = fTracktimes.size() - 1;
244     unsigned int l = 0;
245
246     while (u - l != 1) {
247         assert(u > 0 && l >= 0);
248         assert(u < fTracktimes.size() && l < fTracktimes.size());
249         assert(u - l > 1);

```

A. Kommentierter Quelltext

```

250     const unsigned int m = (u + 1)/2;
251     assert(fTracktimes[1] <= fTracktimes[m] && fTracktimes[u] > fTracktimes[m]);
252
253     if (fTracktimes[m] > time) {
254         u = m;
255     }
256     else {
257         l = m;
258     }
259     assert(fTracktimes[1] <= time && fTracktimes[u] > time);
260 }
261
262 debug << "bracketed_t_u=" << time << "between_l_u=" << 1 << ",t[l]=u" << fTracktimes[1]
263 << "and_u=u" << u << ",t[u]=u" << fTracktimes[u] << std::endl;
264 assert(u - 1 == 1);
265 assert(fTracktimes[1] <= time && fTracktimes[u] > time);
266
267 debug << "index_for_t=u" << time << "is_u" << 1 << std::endl;
268
269 last_index = l;
270 assert(l >= 0);
271 assert(l < fTracktimes.size());
272 return l;
273 }
```

Listing A.14: fastcylindertracker.h

```

1 #ifndef _FASTCYLINDERTRACKER_H
2 #define _FASTCYLINDERTRACKER_H
3
4 #include <vector>
5 #include <utility>
6 #include "basetracking.h"
7 #include "cylinder.h"
8
9 class FastCylinderTracker : public Basetracking
10 {
11     public:
12         FastCylinderTracker(const Parameters &params, Random* ran, Cylinder *geo);
13         virtual ~FastCylinderTracker();
14
15         virtual void initialize();
16         Threecvector getPosition(double time);
17         Threecvector getVelocity(double time);
18         void makeTrack(double t_start, double h);
19
20     private:
21         enum Surface {None, Top, Bottom, Radius};
22         Cylinder *fGeometry;
23         const double H; //height of the cylinder
24         const double R; //radius of the cylinder
25         const double R2; //squared radius of the cylinder
26         const double g; //acceleration due to gravitation
27
28         Surface fLastCollisionSurface;
29         const double fDiffuseProbability;
30
31         void addSolutions(std::vector<std::pair<double, Surface> > &v, const double a, const double b, const double c,
32                           const double t0, const Surface current) const;
33
34         /**
35          * Find the index in fTracktimes for @time.
36          * Returns i with <tt>fTracktimes[i] <= time <= fTracktimes[i+1]</tt>.
37          */
38         unsigned int findIndex(const double time);
39     };
40 #endif // _FASTCYLINDERTRACKER_H
```

Listing A.15: gravitationtracker.h

```

1 #include "equationtracker.h"
2 #include "basegeometry.h"
3 #include "parameters.h"
4 #include "debug.h"
5
6 class GravitationTracker : public EquationTracker {
7     public:
8         /**
9          * Construct a new GravitationTracker and set the acceleration to @p g.
10         * @param g acceleration due to gravitation, e.g. g = 9.81
11         */
12         explicit GravitationTracker(const Parameters &params, Random *ran, Basegeometry *geo) :
13             EquationTracker(params, ran, geo), minus_g(-params.getDoubleParam("GravitationConstant"))
14         {
15             debug << "minus_g=" << minus_g << std::endl;
16         }
17
18         /**
19          * This class represents the equation of motion for a particle in the
```

A. Kommentierter Quelltext

```

20      * gravitational field  $\mathbf{F} = -m \mathbf{g}$  transformed to
21      *
22      * The motion of equation
23      *  $\dot{\mathbf{x}} = \mathbf{v}$ ,  $\ddot{\mathbf{x}} = \mathbf{g}$ ,  $\hat{\mathbf{e}}_z \times \mathbf{v}$ 
24      *
25      * is transformed to
26      *
27      *  $\dot{\mathbf{x}} = \mathbf{v}$ 
28      *  $\dot{\mathbf{v}} = \mathbf{g} - \hat{\mathbf{e}}_z \times \mathbf{v}$ 
29      *
30      */
31
32 inline void derivs(const double t, const Threevector x, const Threevector v, Threevector &xdot, Threevector &vdot) {
33     // The derivative @p xdot is the velocity and hence set to @p v.
34     xdot = v;
35
36     // Newton says that @p vdot is equal to the acceleration which is g in z direction here.
37     // The x and y components must explicitly be set to zero because @p xdot and @p vdot are not
38     // zeroed before they are given to derivs.
39     vdot[0] = 0;
40     vdot[1] = 0;
41     vdot[2] = minus_g;
42 }
43
44 private:
45     double minus_g; // Acceleration  $\mathbf{g} = -g \mathbf{e}_z$ 
46
47     GravitationTracker(); // No default constructor
48 };

```

A.5. Integration der Bloch-Gleichung

Listing A.16: derivatives.h

```

1 #ifndef DERIVATIVES_H
2 #define DERIVATIVES_H
3
4 #include "parameters.h"
5
6 class Bfield;
7
8 class Derivatives
9 {
10     public:
11     Derivatives(const Parameters&, Bfield*);
12     void operator()(const double, const double[], double[]);
13     void eval(const double, const double[], double[]);
14     private:
15     Bfield *field;
16     double gyromag;
17 };
18 #endif

```

Listing A.17: derivatives.cpp

```

1 #include "derivatives.h"
2 #include "globals.h"
3 #include <math.h>
4 #include <iostream>
5 #include "bfield.h"
6 #include "threevector.h"
7 #include "parameters.h"
8
9 using namespace std;
10
11 /**
12  * @class Derivatives
13  * The differential equation System for Dopr.
14  *
15  * @see Dopr
16  */
17
18 Derivatives :: Derivatives(const Parameters& theParameters, Bfield* F)
19 : field(F), gyromag(0.0)
20 {
21     gyromag = theParameters.getDoubleParam("GyromagneticRatio");
22 }
23
24 void Derivatives::operator()(const double t, const double y[], double derivs[])
25 {
26     eval(t, y, derivs);
27 }
28 /**
29 */

```

A. Kommentierter Quelltext

```

30  * Save the value of the derivatives into @p derivs[].
31  *
32  * @param[in] time current simulation time
33  * @param[in] y[] current coordinates of the particle
34  * @param[out] derivs[] is set to the derivatives
35  */
36 void Derivatives::eval(const double time, const double y[], double derivs[])
37 {
38     Threevector B = field->eval(time);
39     debug << "In eval: B = " << B.toString() << endl;
40
41     /**
42      * The first 3 components of @p derivs are the polarization vector
43      * P and follow the Boch equation.
44      */
45     derivs[0] = gyromag * (y[1]*B[2] - y[2]*B[1]);
46     derivs[1] = gyromag * (y[2]*B[0] - y[0]*B[2]);
47     derivs[2] = gyromag * (y[0]*B[1] - y[1]*B[0]);
48 }

```

A.6. Hilfsfunktionen

Listing A.18: polynom.h

```

1  #ifndef _POLYNOM_H
2  #define _POLYNOM_H
3
4  #include <vector>
5  #include <string>
6
7  class Polynom {
8     public:
9         Polynom();
10        explicit Polynom(int n);
11        Polynom(double a1, double a0);
12        Polynom(double a2, double a1, double a0);
13        Polynom(double a3, double a2, double a1, double a0);
14        unsigned int degree() const;
15        Polynom derivative() const;
16        std::string toString() const;
17        virtual double operator()(double x) const;
18        double operator[](unsigned int n) const { return coeffs[n]; };
19        Polynom operator-() const;
20        friend bool operator==(const Polynom &l, const Polynom &r);
21        friend bool operator!=(const Polynom &l, const Polynom &r);
22        friend Polynom operator+(const Polynom &l, const Polynom &r);
23        friend Polynom operator-(const Polynom &l, const Polynom &r);
24        friend Polynom operator*(const Polynom &l, const Polynom &r);
25    protected:
26        std::vector<double> coeffs;
27    };
28
29
30 #endif // _POLYNOM_H

```

Listing A.19: polynom.cpp

```

1  #include <iostream>
2  #include <algorithm>
3  #include <cassert>
4  #include "debug.h"
5  #include "polynom.h"
6
7  /**
8   * @class Polynom
9   *
10  * Represents a polynomial.
11  */
12
13 /**
14  * Create a Polynom object ready to take a polynomial of degree @p n without
15  * needing a resize of the internal coefficient vector. The coefficients
16  * will be set to zero.
17  */
18 Polynom::Polynom(int n) :
19     coeffs(n+1, 0.)
20 {
21 }
22
23 /**
24  * Create Polynom <tt>f(x) = a1*x + a0</tt>
25  */
26 Polynom::Polynom(double a1, double a0) :

```

A. Kommentierter Quelltext

```

27     coeffs(2)
28 {
29     coeffs[0] = a0;
30     coeffs[1] = a1;
31 }
32
33 /**
34 * Create Polynom <tt>f(x) = a2*x^2 + a1*x + a0</tt>
35 */
36 Polynom::Polynom(double a2, double a1, double a0) :
37     coeffs(3)
38 {
39     coeffs[0] = a0;
40     coeffs[1] = a1;
41     coeffs[2] = a2;
42 }
43
44 /**
45 * Create Polynom <tt>f(x) = a3*x^3 + a2*x^2 + a1*x + a0</tt>
46 */
47 Polynom::Polynom(double a3, double a2, double a1, double a0) :
48     coeffs(4)
49 {
50     coeffs[0] = a0;
51     coeffs[1] = a1;
52     coeffs[2] = a2;
53     coeffs[3] = a3;
54 }
55
56 /**
57 * Evaluate polynomial at @p x using a horner schema.
58 */
59 double Polynom::operator()(double x) const {
60     debug << "Polynom: " << this->toString() << " called for x = " << x << std::endl;
61     double acc = 0;
62
63     for (int i = degree(); i >= 0; i--) {
64         debug << "i = " << i << ", acc = " << acc << std::endl;
65         acc *= x;
66         debug << "acc *= x -> acc = " << acc << std::endl;
67         acc += coeffs[i];
68         debug << "acc += " << coeffs[i] << " -> acc = " << acc << std::endl;
69     }
70
71     debug << "Polynom: result = " << acc << std::endl;
72     return acc;
73 }
74
75 /**
76 * Return the degree of the polynomial.
77 * @returns the biggest n with <tt>coeffs[n] != 0</tt>.
78 */
79 unsigned int Polynom::degree() const {
80     if (coeffs.size() == 0)
81         return 0;
82     assert(coeffs.size() > 0);
83
84     // Reduce degree if degree is smaller than size of vector
85     unsigned int deg = coeffs.size() - 1;
86
87     while (deg > 0 && coeffs[deg] == 0)
88         deg--;
89
90     assert(deg >= 0);
91
92     return deg;
93 }
94
95 /**
96 * Return the derivative.
97 */
98 Polynom Polynom::derivative() const {
99     const int deg = degree();
100    Polynom d(deg - 1);
101
102    for (int i = 1; i <= deg; i++)
103        d[i-1] = coeffs[i]*i;
104
105    return d;
106 }
107
108 /**
109 * Get or set coefficients of Polynom.
110 * If @p n is greater than the current degree of the
111 * polynomial, it is resized.
112 */
113 double &Polynom::operator[](unsigned int n) {
114     // Enlarge coefficient-array if necessary
115     if (n >= coeffs.size())
116         coeffs.resize(n + 1, 0.);
117
118     return coeffs[n];
119 }
120

```

A. Kommentierter Quelltext

```

121  /**
122   * Compare two Polynom objects for equality.
123   */
124  bool operator==(const Polynom &l, const Polynom &r) {
125      const unsigned int deg = l.degree();
126
127      if (deg != r.degree())
128          return false;
129
130      for (int i = deg; i >= 0; i--)
131          if (l[i] != r[i])
132              return false;
133
134      return true;
135  }
136
137  /**
138   * Compare two Polynom objects for inequality.
139   */
140  bool operator!=(const Polynom &l, const Polynom &r) {
141      return !(l == r);
142  }
143
144  /**
145   * Add two Polynom objects.
146   */
147  Polynom operator+(const Polynom &l, const Polynom &r) {
148      // l should be the polynomials with the lower degree, r the one with the higher (or equal) degree. See below for
149      // how this is made sure.
150
151      // common_deg must be the degree that both polynomials have in common, that is the lower of the two degrees
152      const unsigned int common_deg = l.degree();
153
154      // max_deg must be the greater of the two degrees
155      const unsigned int max_deg = r.degree();
156
157      // Make sure l is the polynomial of lower degree (see above)
158      if (common_deg > max_deg)
159          return r+l; // Change roles of r and l
160
161      debug << "Adding polynomials " << l.toString() << "+ " << r.toString() << ", degrees: " << common_deg << ", " <<
162      max_deg << std::endl;
163
164      assert(common_deg <= max_deg);
165
166      // The resulting polynomial will have the maximum degree of both
167      Polynom t(max_deg);
168
169      for (unsigned int i = 0; i <= max_deg; i++) {
170          if (i <= common_deg) {
171              t[i] = l[i]+r[i]; // Add both polynomials
172          } else {
173              assert(i > l.degree());
174              assert(i <= r.degree());
175              t[i] = r[i]; // Only r[i] has entries of this degree
176          }
177      }
178
179      return t;
180  }
181
182  /**
183   * Subtract two polynomials.
184   */
185  Polynom operator-(const Polynom &l, const Polynom &r) {
186      return l + (-r);
187  }
188
189  /**
190   * Negate a polynomial.
191   */
192  Polynom Polynom::operator-() const {
193      Polynom t(degree());
194      for (unsigned int i = 0; i < degree(); i++)
195          t[i] = -coeffs[i];
196
197      return t;
198  }
199
200  /**
201   * Multiply two polynomials
202   */
203  Polynom operator*(const Polynom &l, const Polynom &r) {
204      // Get degrees of polynomials
205      const int dl = l.degree();
206      const int dr = r.degree();
207
208      // Create temporary new polynomial
209      Polynom res(dl+dr);
210
211      debug << "Created temporary Polynomial: " << res.toString() << std::endl;
212      for (unsigned int i = 0; i < res.coeffs.size(); i++) {
213          debug << "coeffs[" << i << "] = " << res.coeffs[i] << std::endl;
214      }

```

A. Kommentierter Quelltext

```

213     debug << "Degree_of_result_polynomial_is:" << res.degree() << std::endl;
214
215     // Multiply polynomials
216     for (int il = 0; il <= dl; il++)
217         for (int ir = 0; ir <= dr; ir++)
218             res[il+ir] += r[ir] * l[il];
219
220     debug << "Multiplication_returning:" << res.toString() << std::endl;
221     return res;
222 }
223
224 /**
225 * Return a human-readable string representation.
226 */
227 std::string Polynom::toString() const {
228     bool first = true;
229     std::ostringstream o;
230
231 #ifndef NDEBUG
232     o << std::scientific;
233     o.precision(15);
234 #endif
235
236     for (int i = degree(); i >= 0; i--) {
237         const double c = coeffs[i];
238
239         // coefficient
240         if (c == 0 && i != 0) {
241             continue;
242         }
243         else if (first) {
244             if (c == 1) {
245                 // do nothing
246             }
247             else if (c == -1) {
248                 o << "-";
249             }
250             else {
251                 o << c;
252             }
253             first = false;
254         }
255         else if (c == 1 && i != 0) {
256             o << "+";
257         }
258         else if (c < 0) {
259             o << "-<< -c;
260         }
261         else {
262             o << "+<< c;
263         }
264
265         // x^n
266         if (i > 1)
267             o << "x^" << i;
268         else if (i == 1)
269             o << "x";
270         // do nothing for i == 0
271     }
272
273     return o.str();
274 }

```

Listing A.20: interpolationpolynomial.h

```

1  #ifndef _INTERPOLATIONPOLYNOMIAL_H
2  #define _INTERPOLATIONPOLYNOMIAL_H
3
4  #include <cassert>
5  #include "polynom.h"
6  #include "debug.h"
7
8  class InterpolationPolynomial : public Polynom
9  {
10     public:
11         InterpolationPolynomial(double t0, double y0, double d0, double t1, double y1, double d1) :
12             Polynom(2)
13         {
14             const double t1_squared = t1*t1;
15             const double t0_squared = t0*t0;
16             coeffs[0] = (d0*t0*(t0*t1 - t1_squared) + t1_squared*y0 + t0*(-2*t1*y0 + \
17                         t0*y1))/(t0*(t0 - 2*t1) + t1_squared);
18             coeffs[1] = (d0*(-t0_squared + t1_squared) + t0*(2*y0 - 2*y1))/(t0*(t0 - 2*t1) + t1_squared);
19             coeffs[2] = (d0*(t0 - t1) - y0 + y1)/(t0*(t0 - 2*t1) + t1_squared);
20         };
21
22     };
23
24 #endif // _INTERPOLATIONPOLYNOMIAL_H

```

A. Kommentierter Quelltext

Listing A.21: roots.h

```

1 #include "debug.h"
2 #include "exceptions.h"
3 #include <iostream>
4 #include <cmath>
5
6 /**
7 * @namespace Roots
8 * Algorithms for root finding.
9 */
10 namespace Roots {
11     template <class T>
12     double bisectStep(const T &f, double &x1, double &x2);
13     template <class T>
14     double safeNewton(const T &f, const T &d, double x1, double x2, double eps);
15
16     const unsigned int MAX_ITERATIONS = 10000;
17
18     class NoRoot : public Exception {
19         public:
20             NoRoot() : Exception("no_root_found") {};
21     };
22 }
23
24 /**
25 * Do one bisection step for <tt>f(double)</tt>. The root must certainly
26 * be between @p x1 and @p x2, that means their signs need to be different
27 * from each other. At the end of the call, @p x1 or @p x2 will be set to
28 * <math>0.5*(x1+x2)</math>.
29 *
30 * @param f Function to find roots of
31 * @param[in,out] x1 Lower bound of the interval where the root is.
32 * @param[in,out] x2 Upper bound of the interval where the root is.
33 *
34 * @returns The center of the new interval <math>0.5*(x1+x2)</math>
35 */
36 template <class T>
37 double Roots::bisectionStep(const T &f, double &x1, double &x2) {
38     const double y1 = f(x1);
39     const double y2 = f(x2);
40     const double xm = .5*(x1+x2);
41     const double ym = f(xm);
42
43     debug << "bisecting: (" << x1 << "," << y1 << ") -- (" << xm << "," << ym << ") -- (" << x2 << "," << y2 << ")"
44     << std::endl;
45
46     if (x1 > x2)
47         throw "x1 > x2";
48
49     if (y1*ym < 0)
50         x2 = xm; // root between xm and x2
51     else if (y2*ym < 0)
52         x1 = xm; // root between x1 and xm
53     else if (y1 == 0)
54         x2 = x1;
55     else if (y2 == 0)
56         x1 = x2;
57     else if (ym == 0) {
58         x1 = xm;
59         x2 = xm;
60     }
61     else
62         throw NoRoot();
63
64     return .5*(x1+x2);
65 }
66
67 /**
68 * Find the root of function @p f with derivative @p d. The root must already be
69 * bracketed between @p x1 and @p x2 for this to work. This algorithm is quite
70 * robust and will use bisection if the newton algorithm does strange things. It
71 * not evaluate @p f or @p d outside of <tt>[x1,x2]</tt> except for debugging prints.
72 *
73 * @param f The function to find a root of
74 * @param d The derivative of @p f
75 * @param[in] x1 Lower bound of the interval that brackets the root.
76 * @param[in] x2 Upper bound of the interval that brackets the root.
77 * @param[in] eps Accuracy goal, the algorithm will ensure <math>|f(x\_root)| < \epsilon</math>
78 */
79 template <class T>
80 double Roots::safeNewton(const T &f, const T &d, double x1, double x2, double eps) {
81     if (fabs(f(x1)) < eps)
82         return x1;
83
84     debug << "Doing_initial_bisection_step:" << std::endl;
85     double x = bisectStep(f, x1, x2); // Do a bisection step first to check arguments for sanity
86     double y = f(x);
87     double last_y = INFINITY;
88
89     unsigned int iteration = 0;
90
91     while (fabs(y) > eps) {
92         // Protect against infinite loops
93         if (iteration++ > MAX_ITERATIONS)

```

A. Kommentierter Quelltext

```

93         throw EndlessLoop();
94
95     const double xn = x - f(x)/d(x);
96
97     debug << "f(" << x << ")" << y << ", last_was:" << last_y << std::endl;
98
99     // Test if result is ok
100    if (xn < x1 || xn > x2) {
101        // outside of original rage, bisect
102        debug << "Bisecting:" << xn << " out_of_range." << std::endl;
103        x = bisectStep(f, x1, x2);
104    }
105    else if (fabs(f(xn)) >= last_y) {
106        // function value got bigger, maybe oscillation or divergence
107        debug << "Bisecting: |f(" << xn << ")"| >= " << last_y << std::endl;
108        x = bisectStep(f, x1, x2);
109    }
110    else {
111        // newton seems to go into the right direction, go on
112        debug << "Accepting: " << x << " ->" << xn << std::endl;
113        x = xn;
114    }
115
116    last_y = fabs(y);
117    y = f(x);
118}
119
120 debug << "Reached_accuracy_of_ " << eps << " , f(" << x << ")" ) = " << y << std::endl;
121
122 return x;
123 }
```

Listing A.22: random.h

```

1 #ifndef RANDOM_H
2 #define RANDOM_H
3
4 #include <gsl/gsl_rng.h>
5 #include <gsl/gsl_randist.h>
6
7
8 class Random
9 {
10     public:
11     Random(unsigned long int);
12     ~Random();
13     double uniform();
14     int uniform_int(int max);
15     unsigned long int generate_seed();
16     double gaussian(const double);
17     double exponential(const double);
18     double uniform_double(double min, double max);
19     gsl_rng* GetGsl_Rng();
20
21     private:
22     const gsl_rng_type* T;
23     gsl_rng* r;
24 };
#endif
```

Listing A.23: random.cpp

```

1 #include "random.h"
2 #include "debug.h"
3
4 Random::Random(unsigned long int seed)
5 : T(gsl_rng_mt19937), r(gsl_rng_alloc(T))
6 {
7     debug << "seed: " << seed << std::endl;
8     gsl_rng_set(r, seed);
9 }
10
11 Random::~Random()
12 {
13     gsl_rng_free(r);
14 }
15
16 int Random::uniform_int(int max)
17 {
18     return gsl_rng_uniform_int(r, max+1);
19 }
20
21 double Random::uniform_double(double min, double max)
22 {
23     return gsl_ran_flat(r, min, max);
24 }
25
26 unsigned long int Random::generate_seed()
```

A. Kommentierter Quelltext

```

27  {
28      return gsl_rng_get(r);
29  }
30
31 double Random::uniform()
32 {
33     return gsl_rng_uniform(r);
34 }
35
36 double Random::gaussian(const double sigma)
37 {
38     return gsl_ran_gaussian(r, sigma);
39 }
40
41 double Random::exponential(const double mu)
42 {
43     return gsl_ran_exponential(r, mu);
44 }
45
46 gsl_rng* Random::GetGsl_Rng()
47 {
48     return r;
49 }
```

Listing A.24: threevector.h

```

1 #ifndef THREEVECTOR_H
2 #define THREEVECTOR_H
3
4 #include "debug.h"
5 #include <vector>
6 #include <string>
7 #include <iostream>
8
9 class Threevector
10 {
11     public:
12     Threevector(double x, double y, double z);
13     Threevector();
14     Threevector(double *v);
15     Threevector(const Threevector &t);
16     void setX(double x){fVec[0] = x;};
17     void setY(double y){fVec[1] = y;};
18     void setZ(double z){fVec[2] = z;};
19     double getX(){return fVec[0];};
20     double getY(){return fVec[1];};
21     double getZ(){return fVec[2];};
22     double &operator[](const int i) {return fVec[i];};
23     double operator[](const int i) const {return fVec[i];};
24     Threevector &operator*=(const double a) {
25         for (int i = 0; i < 3; i++)
26             fVec[i] *= a;
27         return *this;
28     };
29     Threevector &operator=(const Threevector &v) {
30         for (int i = 0; i < 3; i++)
31             fVec[i] = v.fVec[i];
32         return *this;
33     };
34     Threevector &operator+=(const Threevector &v) {
35         for (int i = 0; i < 3; i++)
36             fVec[i] += v.fVec[i];
37         return *this;
38     };
39     Threevector operator-() const {
40         Threevector ret;
41         for (int i = 0; i < 3; i++)
42             ret[i] = -fVec[i];
43         return ret;
44     };
45
46     friend Threevector operator+(const Threevector &left, const Threevector &right);
47     friend Threevector operator-(const Threevector &left, const Threevector &right);
48     friend double operator*(const Threevector &left, const Threevector &right);
49     friend Threevector operator*(const double &left, const Threevector &right);
50     friend Threevector operator*(const Threevector &left, const double &right) {
51         return right * left;
52     };
53     friend bool operator==(const Threevector &left, const Threevector &right);
54     friend bool operator!=(const Threevector &left, const Threevector &right);
55     double mag() const;
56     double magsquare() const;
57     void normalize();
58     Threevector normalized() {
59         return (1./mag())*(*this);
60     };
61     std::string toString() const;
62     Threevector cross (const Threevector &x) const ;
63 protected:
64     double fVec[3];
```

A. Kommentierter Quelltext

```

65 };
66
67 std::ostream& operator<<(std::ostream &o, const Threemode &v);
68 std::istream& operator>>(std::istream &i, Threemode &v);
69
70 #endif

1 #include "debug.h"
2 #include "threemode.h"
3 #include <cmath>
4 #include <iostream>
5
6 Threemode::Threemode(double x, double y, double z)
7 {
8     fVec[0] = x;
9     fVec[1] = y;
10    fVec[2] = z;
11 }
12
13 Threemode::Threemode()
14 {
15     for(int i=0; i<3; i++)
16     {
17         fVec[i] = 0.0;
18     }
19 }
20
21 Threemode::Threemode(double *v)
22 {
23     for(int i=0; i<3; i++)
24     {
25         fVec[i] = v[i];
26     }
27 }
28
29 Threemode::Threemode(const Threemode &v)
30 {
31     //debug << "Copy constructor called!" << std::endl;
32     for(int i=0; i<3; i++)
33         fVec[i] = v.fVec[i];
34 }
35
36 Threemode operator+(const Threemode &left, const Threemode &right)
37 {
38     Threemode a;
39     for(int i=0; i<3; i++)
40     {
41         a.fVec[i] = left.fVec[i] + right.fVec[i];
42     }
43     return a;
44 }
45
46 Threemode operator-(const Threemode &left, const Threemode &right)
47 {
48     Threemode a;
49     for(int i=0; i<3; i++)
50     {
51         a.fVec[i] = left.fVec[i] - right.fVec[i];
52     }
53     return a;
54 }
55
56 double operator*(const Threemode &left, const Threemode &right)
57 {
58     double a=0.0;
59     for(int i=0; i<3; i++)
60     {
61         a += left.fVec[i]*right.fVec[i];
62     }
63     return a;
64 }
65
66 Threemode operator*(const double &left, const Threemode &right)
67 {
68     Threemode a;
69     for(int i=0; i<3; i++)
70     {
71         a.fVec[i] = left*right.fVec[i];
72     }
73     return a;
74 }
75
76 bool operator==(const Threemode &left, const Threemode &right)
77 {
78     bool a = true;
79     for(int i=0; i<3; i++)
80     {
81         a = a && (left.fVec[i] == right.fVec[i]);
82     }
83     return a;
84 }

```

A. Kommentierter Quelltext

```

83     }
84     return a;
85 }
86
87 bool operator!=(const Threemode &left, const Threemode &right)
88 {
89     return !(left==right);
90 }
91
92 double Threemode::mag() const
93 {
94     double a=0.0;
95     for(int i=0; i<3; i++)
96     {
97         a += fVec[i]*fVec[i];
98     }
99     return sqrt(a);
100}
101
102 double Threemode::magsquare() const
103 {
104     double a=0;
105     for(int i=0; i<3; i++)
106     {
107         a += fVec[i]*fVec[i];
108     }
109     return a;
110}
111
112 void Threemode::normalize()
113 {
114     const double div = mag();
115     for(int i=0; i<3; i++)
116         fVec[i] /= div;
117 }
118
119
120 std::string Threemode::toString() const {
121     std::ostringstream o;
122     o << "(" << fVec[0] << " " << fVec[1] << " " << fVec[2] << ")";
123     return o.str();
124 }
125
126 Threemode Threemode::cross (const Threemode &x) const {
127     Threemode c;
128     c[0]=fVec[1]*x[2]-fVec[2]*x[1];
129     c[1]=fVec[2]*x[0]-fVec[0]*x[2];
130     c[2]=fVec[0]*x[1]-fVec[1]*x[0];
131     return c;
132 }
133
134 std::ostream& operator<<(std::ostream &o, const Threemode &v)
135 {
136     o << v.toString();
137     return o;
138 }
139
140 std::istream& operator>>(std::istream &i, Threemode &v)
141 {
142     for (int j = 0; j < 3; j++)
143         i >> v[j];
144     return i;
145 }
```

Listing A.26: parameters.h

```

1 #ifndef PARAMETERS_H
2 #define PARAMETERS_H
3 #include <map>
4 #include <set>
5 #include <string>
6 #include <iostream>
7 #include <vector>
8
9 class Parameters
10 {
11     public:
12
13     struct SerializedParam {
14         char name[50];
15         char value[25];
16     };
17
18     Parameters();
19     ~Parameters();
20     void add(std::string name, double value);
21     void add(std::string name, int value);
22     void expectInt(std::string name);
23     void expectInt(std::string name, int def);
24     void expectDouble(std::string name);
25     void expectDouble(std::string name, double def);
```

A. Kommentierter Quelltext

```

26     double getDoubleParam(std::string name) const;
27     int getIntParam(std::string name) const;
28     int getSize() const;
29     void readParameters(std::istream &in);
30     std::vector<SerializedParam> serialize();
31
32     private:
33
34     std::map<std::string, double> fDoubles;
35     std::map<std::string, int> fInts;
36     std::set<std::string> fExpectedDoubles;
37     std::set<std::string> fExpectedInts;
38 };
39 #endif

```



```

1 #include <iostream>
2 #include <cstring>
3 #include "parameters.h"
4 #include <utility> // make_pair
5 #include <sstream>
6
7 /**
8 * @class Parameters
9 * Read and save parameters for the simulation
10 */
11 * The method #readParameters of this class will read parameters from
12 * an input stream of the simple format
13 * @verbatim name value @endverbatim
14 * and save them for later use.
15 *
16 * The parameters have to be introduced to the class by calling
17 * #expectInt(std::string) or #expectDouble(std::string) for
18 * it before calling #readParameters.
19 */
20
21 Parameters::Parameters()
22 {
23 }
24
25 Parameters::~Parameters()
26 {
27 }
28
29
30 /**
31 * Set the value for a parameter.
32 * @param name name of the parameter
33 * @param value new value of the parameter
34 */
35
36 void Parameters::add(std::string name, double value)
37 {
38     // don't use insert, it will not replace a default value
39     fDoubles[name] = value;
40 }
41
42 /**
43 * Set the value for a parameter.
44 * @param name name of the parameter
45 * @param value new value of the parameter
46 */
47 void Parameters::add(std::string name, int value)
48 {
49     // don't use insert, it will not replace a default value
50     fInts[name] = value;
51 }
52
53 /**
54 * Tell the parser that we expect a parameter of
55 * type int.
56 * @param name the name of the expected parameter
57 */
58 void Parameters::expectInt(std::string name) {
59     fExpectedInts.insert(name);
60 }
61
62 /**
63 * Tell the parser that we expect a parameter and give
64 * default value.
65 * @see #expectInt(std::string)
66 */
67 void Parameters::expectInt(std::string name, int def) {
68     expectInt(name);
69     add(name, def);
70 }
71
72 /**
73 * Tell the parser that we expect a parameter of
74 * type double.
75 * @param name the name of the expected parameter

```

A. Kommentierter Quelltext

```

76  */
77 void Parameters::expectDouble(std::string name) {
78     fExpectedDoubles.insert(name);
79 }
80
81 /**
82 * Tell the parser that we expect a parameter and give
83 * default value.
84 * @see #expectDouble(std::string)
85 */
86 void Parameters::expectDouble(std::string name, double def) {
87     expectDouble(name);
88     add(name, def);
89 }
90
91 /**
92 * Retrieve value of parameter
93 */
94 double Parameters::getDoubleParam(std::string name) const
95 {
96     std::map<std::string,double>::const_iterator it = fDoubles.find(name);
97     if(it == fDoubles.end())
98     {
99         std::cout << "Parameter " << name << " not found, returning 0.0" << std::endl;
100    }
101   return 0.0;
102 }
103
104
105 /**
106 * Retrieve value of parameter
107 */
108 int Parameters::getIntParam(std::string name) const
109 {
110     std::map<std::string,int>::const_iterator it = fInts.find(name);
111     if(it == fInts.end())
112     {
113         std::cout << "Parameter " << name << " not found, returning 0" << std::endl;
114         return 0;
115     }
116     return it->second;
117 }
118
119 /**
120 * Return total number of parameters in table.
121 */
122 int Parameters::getSize() const
123 {
124     return fDoubles.size() + fInts.size();
125 }
126
127 /**
128 * Read parameters from stream.
129 * @param in stream from which parameters will be read
130 * @see Parameters for description of file format
131 */
132 void Parameters::readParameters(std::istream &in) {
133     std::string name;
134     double d;
135     int i;
136
137     while (!in.eof()) {
138         // read name from stream
139         in >> name;
140
141         if (name[0] == '#') {
142             // ignore comment
143         }
144         else if (fExpectedInts.count(name) > 0) {
145             in >> i; // read integer
146             add(name, i); // add to parameters
147         }
148         else if (fExpectedDoubles.count(name) > 0) {
149             in >> d; // read double
150             add(name, d); // add to parameters
151         }
152         else {
153             std::cout << "Parameter " << name << " not known!" << std::endl;
154         }
155
156         // ignore garbage at end of line
157         in.ignore(4192, '\n');
158     }
159 }
160
161
162 std::vector<Parameters::SerializedParam> Parameters::serialize() {
163     std::vector<Parameters::SerializedParam> out;
164     Parameters::SerializedParam p;
165     std::ostringstream o;
166
167     for (std::map<std::string,int>::iterator i = fInts.begin(); i != fInts.end(); i++) {
168         o.str("");

```

A. Kommentierter Quelltext

```

169     strncpy(p.name, i->first.c_str(), 50);
170     o << i->second;
171     strncpy(p.value, o.str().c_str(), 25);
172     out.push_back(p);
173 }
174
175 for (std::map<std::string, double>::iterator i = fDoubles.begin(); i != fDoubles.end(); i++) {
176     o.str("");
177     strncpy(p.name, i->first.c_str(), 50);
178     o << i->second;
179     strncpy(p.value, o.str().c_str(), 25);
180     out.push_back(p);
181 }
182
183 return out;
184 }
```

Listing A.28: timeout.h

```

1 #ifndef _TIMEOUT_H
2 #define _TIMEOUT_H
3
4 #include <ctime>
5 #include "exceptions.h"
6
7 class TimeoutException : public Exception {
8     public:
9         TimeoutException() : Exception("timeout") {};
10    };
11
12 class Timeout {
13     public:
14         explicit Timeout(time_t seconds) : fTimeout(seconds) { reset(); };
15         void reset() { fStartTime = time(NULL); };
16         void check()
17         {
18             if (time(NULL) - fStartTime > fTimeout)
19                 throw TimeoutException();
20         };
21     private:
22         time_t fTimeout;
23         time_t fStartTime;
24    };
25
26 #endif // _TIMEOUT_H
```

Listing A.29: globals.h

```

1 #ifndef GLOBALS_H
2 #define GLOBALS_H
3
4 static const double hbar = 1.054571682364455e-34; // Planck constant over 2 pi (J s)
5 static const double elementarycharge = 1.60218e-19; // (C)
6 static const double speed_of_light = 299792458; // m/s
7 static const double boltzmann = 1.3806488e-23; // J/K CODATA
8
9 #endif
```

Listing A.30: debug.h

```

1  /**
2   * @file
3   * This file provides debugging facilities for the code. You can use debug as output
4   * stream like cout and cerr, but it will be deactivated, if NDEBUG or NO_DEBUG_PRINTS
5   * is set. It will also add the filename and linenumber of the code that generated the
6   * debug print to the output. Don't forget to add std::endl at the end of any debug
7   * statement.
8   */
9
10 #ifndef _UCN_DEBUG_H
11 #define _UCN_DEBUG_H
12
13 #include <iostream>
14
15 /// Debug prints are automatically deactivated if NDEBUG (which deactivates
16 /// asserts) is set.
17 #ifdef NDEBUG
18 #define NO_DEBUG_PRINTS
19 #endif
20
21 /// By setting NO_DEBUG_PRINTS, you can deactivate debug prints without
22 /// deactivating asserts.
23 #ifndef NO_DEBUG_PRINTS
24 #define debug std::cerr << "(" __FILE__ ":" << __LINE__ << ")"
25 #define initialize_debug() std::cerr << std::fixed; std::cerr.precision(15)
```

A. Kommentierter Quelltext

```
26 #else
27 #define debug if(false) std::cerr
28 #define initialize_debug()
29 #endif
30
31 #endif // _UCN_DEBUG_H
```

Listing A.31: exceptions.h

```
1 #ifndef _EXCEPTIONS_H
2 #define _EXCEPTIONS_H
3
4 class Exception {
5     public:
6         Exception() : fWhat("unknown_exception") {};
7         Exception(const char *what) : fWhat(what) {};
8         virtual const char *what() const { return fWhat; };
9     private:
10        const char *fWhat;
11    };
12
13 class EndlessLoop : public Exception {
14     public:
15         EndlessLoop() : Exception("endless_loop") {};
16    };
17
18 class LeftVolume : public Exception {
19     public:
20         LeftVolume() : Exception("particle_left_volume") {};
21    };
22
23 #endif // _EXCEPTIONS_H
```

Literaturverzeichnis

- [1] P. Fierlinger und S. Paul, „Die Materie-Antimaterie-Asymmetrie im Universum.”
Im Druck, Veröffentlichung in „Sterne und Weltraum”, 2011.
- [2] S. K. Lamoreaux und R. Golub, „Experimental searches for the neutron electric dipole moment,” *Journal of Physics G: Nuclear and Particle Physics* **36** (2009) .
- [3] C. A. Baker *et al.*, „Improved Experimental Limit on the Electric Dipole Moment of the Neutron,” *Physical Review Letters* **97** (2006) .
- [4] K. Nakamura *et al.* (Particle Data Group), „The Review of Particle Physics,” *Journal of Physics G: Nuclear and Particle Physics* **37** (2010) .
<http://pdg.lbl.gov/>.
- [5] P. Fierlinger. Private Gespräche über das nEDM Experiment mit Prof. Peter Fierlinger.
- [6] W. Feldmeier, „Numerical investigation of a new ^{129}Xe EDM experiment,”
Diplomarbeit, Technische Universität München, Dec., 2010.
- [7] J. H. Smith, E. M. Purcell, und N. F. Ramsey, „Experimental Limits on the Electric Dipole Moment of the Neutron,” *Physical Review* **108** (1957) 120ff.
- [8] N. F. Ramsey, „A Molecular Beam Resonance Method with Separated Oscillating Fields,” *Physical Review* **78** (1950) 695ff.
- [9] R. Golub, D. J. Richardson, und S. K. Lamoreaux, *Ultra-Cold Neutrons*. Adam Hilger, Bristol, Philadelphia, New York, 1991.
- [10] Nobelprize.org, „All Nobel Prizes in Physics.”
http://nobelprize.org/nobel_prizes/physics/laureates/.
- [11] J. M. Pendlebury, W. Heil, Y. Sobolev, P. G. Harris, J. D. Richardson, R. J. Baskin, D. D. Doyle, P. Geltenbort, K. Green, M. G. D. van der Grinten, P. S. Iaydjiev, S. N. Ivanov, D. J. R. May, und K. F. Smith, „Geometric-phase-induced

Literaturverzeichnis

- false electric dipole moment signals for particles in traps," *Physical Review A* **70** Nr. 3, (Sept., 2004) .
- [12] C. Cohen-Tannoudji, „Light shifts and multiple quantum transitions,” in *Physics of the one- and two-electron atoms*. North-Holland, 1969.
- [13] W. Demtröder, *Experimentalphysik 1*. Springer-Lehrbuch, 5. Auflage, 2008.
- [14] W. H. Press, S. A. Teukolsky, W. T. Vetterling, und B. P. Flannery, *Numerical Recipes*. Cambridge University Press, 3. Auflage, 2007.
- [15] M. Galassi *et al.*, *GNU Scientific Library Reference Manual*. Network Theory Ltd, 3. Auflage, 2009. <http://www.gnu.org/software/gsl/>.
- [16] D. J. Griffiths, *Introduction to Electrodynamics*. Prentice Hall, New Jersey, 10. Auflage, 1999.
- [17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, und B. P. Flannery, „Numerical Recipes – Webnote No. 20, Rev. 1,” 2007.
<http://www.nr.com/webnotes/nr3web20.pdf>.
- [18] M. Horras, „A measurement of the Hg geometric phase.” Poster auf der Jahrestagung der Schweizer Physikalischen Gesellschaft, 2011.
- [19] R. K. Harris *et al.*, „NMR nomenclature. Nuclear spin properties and conventions for chemical shifts (IUPAC Recommendations 2001),” *Pure and Applied Chemistry* **73** Nr. 11, (2001) 1795–1818.
- [20] J. J. Hudson, D. M. Kara, I. Smallman, B. E. Sauer, M. R. Tarbutt, und E. A. Hinds, „Improved measurement of the shape of the electron,” *Nature* **473** (2011) 493 – 496. <http://dx.doi.org/10.1038/nature10104>.
- [21] E. M. Purcell und N. F. Ramsey, „On the Possibility of Electric Dipole Moments for Elementray Particles and Nuclei,” *Physical Review* **78** (1950) 807.

Danksagungen

An dieser Stelle möchte ich mich noch bei allen bedanken, die zum Entstehen dieser Arbeit beigetragen haben.

Mein herzlicher Dank geht an meinen Betreuer Peter Fierlinger, der mich mit seiner Begeisterung anstecken konnte und immer für Fragen zur Verfügung stand und an Wolfhart Feldmeier, auf dessen Arbeit ich aufbauen konnte, für seine vielfältige Unterstützung und das Korrekturlesen der Arbeit.

Vielen Dank auch an Tobias Lins und Mike Marino, die mir mit Mathematica und BibTeX weitergeholfen haben und an meinen Komilitonen Maxi Huber für seine Hilfe mit L^AT_EX und den Formalia der schriftlichen Hausarbeit.

Beim gesamten Lehrstuhl möchte ich mich für die freundliche Aufnahme und die vielen kleinen Dinge, die ich vergessen habe zu erwähnen, bedanken. Nicht zuletzt möchte ich auch meinen Kommilitonen danken, ohne die das Studium keinen Spaß machen würde und natürlich auch meiner Familie, die mich immer unterstützt hat.

Erklärung zur Hausarbeit gemäß § 29 (Abs. 6) LPO I

Hiermit erkläre ich, dass die vorliegende Hausarbeit von mir selbstständig verfasst wurde, und dass keine anderen als die angegebenen Hilfsmittel benutzt wurden. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen sind, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf etwa in der Arbeit enthaltene Grafiken, Zeichnungen, Kartenskizzen und bildliche Darstellungen.

Garching, 25. Juli 2011