Visualisierung von Quantentrajektorien beim unendlich tiefen Potentialtopf und beim Doppelspaltversuch

.

Visualization of Quantum Trajectories for the Infinite Potential Well and the Double-Slit Experiment

Bachelorarbeit

an der Fakultät für Physik
Ludwig-Maximilians-Universität München

vorgelegt von
Marlon Jan Metzger

Betreuer: Prof. Dr. Detlef Dürr, Prof. Dr. Jochen Weller

München, Dezember 2013

Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurden Videos zum quantenmechanischen unendlich tiefen Potentialtopf und zum Doppelspaltversuch unter Verwendung der Programme MATLAB und POV-Ray erstellt. In den Videos sind Teilchen und ihre zurückgelegten Trajektorien dargestellt, wie sie von der Bohmschen Mechanik beschrieben werden. Sie können online unter `https://www.youtube.com/user/BohmMechAnimations` betrachtet werden. Diese schriftliche Ausarbeitung hat nun primär zwei Ziele:

1. Die Erläuterung der physikalischen Hintergründe der Bohmschen Mechanik so wie sie nötig sind, um zu verstehen was in den Videos dargestellt wurde. Ein Verständnis dieser Grundlagen stellt gleichzeitig die Basis für 2. dar.

2. Erläuterung der technischen Grundlagen wie solche Videos erstellt werden können.

Der Leser soll in die Lage versetzt werden, den vorhandenen MATLAB- und POV-Ray-Code zur Erstellung neuer Videos und Bilder zu verwenden, zu modifizieren und gegebenfalls ein ähnliches Vorhaben zu realisieren.

Inhaltsverzeichnis

Kapitel 1

Einführung

Ziel dieser Arbeit war die Erstellung von Computeranimationen zu zwei quantenmechanischen Szenarios, in denen die Bohmschen Trajektorien für repräsentative Teilchen, nach Möglichkeit zusammen mit der Wellenfunktion, zeitaufgelöst dargestellt werden sollen. Für die darzustellenden physikalischen Situationen wurden zwei Themen ausgewählt, mit denen jeder Student oder sogar Schüler zu Beginn seiner Beschäftigung mit der Quantenmechanik konfrontiert wird:

- **Der eindimensionale unendlich tiefe Potentialtopf:**
  Dabei stand die Idee im Vordergrund, die komplexe Natur der Quantenmechanik hervorzuheben, indem ausgenutzt wird, dass in einer dreidimensionalen Animation noch zwei Dimensionen zur Verfügung stehen, um Real- und Imaginärteil der Wellenfunktion sichtbar zu machen.

- **Der Doppelspaltversuch mit Materieteilchen:**
  Der Doppelspaltversuch wurde ausgewählt, weil an ihm die Mysterien des Welle-Teilchen-Dualismus so deutlich werden, und die Bohmsche Mechanik eine so einfache Antwort auf die aufgeworfenen Fragen bietet.

Die Bohmschen Trajektorien wurden mit MATLAB numerisch berechnet und mit Hilfe des Computergrafikprogramms POV-Ray in 3D-Animationen dargestellt.

1.1 Mögliche Verwendung der Videos und des Quellcodes

Die entstandenen Videos können wohl in erster Linie für didaktische Zwecke Verwendung finden. Sowohl der Doppelspaltversuch als auch der unendlich tiefe Potentialtopf sind Themen, mit denen jeder Lernende der Quantenmechanik in Kontakt kommt. Den meisten Menschen hilft es beim Lernen, sich mathematische und physikalische Sachverhalte zu visualisieren.

Eine große Motivation für diese Arbeit ist auch gewesen, einen Beitrag gegen die Unwissenheit über die Existenz und den Wert der Bohmschen Mechanik zu leisten. Den meisten Wissenschaftlern und Studenten ist die Bohmsche Mechanik weitgehend oder völlig unbekannt, obwohl sich damit die bekannten Phänomene der Quantenmechanik beschreiben lassen, ohne auf Widersprüche zu gelangen, welche von anderen Interpretationen der Quantenmechanik ausgehen.

Noch immer herrscht der Mythos vor, dass sich das beim Doppelspaltexperiment beobachtbare Interferenzmuster *prinzipiell nicht* durch eine kausale Theorie über die Bewegung der Teilchen, die man auf dem Detektionsschirm empfängt, erklären lässt. Die Bohmsche Mechanik liefert jedoch eine ebensolche Erklärung. Verwunderten Studenten und Schülern wird diese Erklärung meist verschwiegen — sei es aufgrund der Unkenntnis oder der Abneigung der Lehrenden Person gegenüber der Theorie.

Die erstellten Videos können hoffentlich helfen, die Bohmsche Mechanik bekannter zu machen, indem sie eine visuelle Ergänzung zu den bestehenden, hauptsächlich in schriftlicher Form vorliegenden, Informationsmaterialien darstellen.

1.2 Bisherige Visualisierungen zur Bohmschen Mechanik

Dieses Projekt ist nicht das erste Vorhaben seiner Art. Hier sei insbesondere auf gelungene Darstellungen von

- Klaus von Bloh (`http://www.youtube.com/user/kvb100b`)
- sowie von Dr. Gebhard Grübl

hingewiesen. Des weiteren existieren zahlreiche Abbildungen von numerisch berechneten Teilchenbahnen hinter dem Doppelspalt.

Kapitel 2

Was ist Bohmsche Mechanik?

Die Bohmsche Mechanik ist eine Theorie über das Verhalten von Teilchen und Wellenfunktionen. Ihre experimentell überprüfbaren Implikationen sind deckungsgleich mit denen der Quantenmechanik.

Das Verhalten der Wellenfunktion eines Systems mit N Teilchen wird beschrieben durch die **Schrödingergleichung**:

$$\boxed{\mathrm{i}\hbar\frac{\partial\Psi(\mathbf{q},t)}{\partial t} = \left(\sum_{i=1}^{N} -\frac{\hbar^2}{2m_i}\frac{\partial^2}{\partial\mathbf{q}_i^2} + V(\mathbf{q})\right)\Psi(\mathbf{q},t), \qquad \mathbf{q} = (\mathbf{q}_1, ..., \mathbf{q}_N).} \tag{2.1}$$

Das Verhalten der Teilchen wird beschrieben durch die sogenannte **Führungsgleichung der Bohmschen Mechanik**:

$$\boxed{\frac{\mathrm{d}\mathbf{Q}_i}{\mathrm{d}t} = \frac{\hbar}{m_i}\,\mathrm{Im}\left(\frac{\nabla_i\Psi}{\Psi}(\mathbf{Q},t)\right), \qquad i = 1,\ldots,N.} \tag{2.2}$$

Es sind folgende Charakteristiken der Theorie festzustellen:

- Die Bohmsche Mechanik ist eine deterministische Theorie.
- Die Wellenfunktion Ψ ist eine Funktion auf dem Konfigurationsraum. Für ein N-Teilchen System hängt Ψ von $3N$ Ortsvariablen und der Zeit t ab.
- Die Wellenfunktion wirkt auf die Teilchen, aber die Teilchen nicht auf die Wellenfunktion.
- Die Führungsgleichung ist eine Bewegungsgleichung *erster* Ordnung. Das bedeutet, dass für einen gegebenen Anfangsort eines Teilchens und eine gegebene Wellenfunktion, die zukünftige Bahn des Teilchens feststeht. Im Gegensatz zur Newtonschen Mechanik muss kein Wert für die Anfangsgeschwindigkeit gegeben sein.

2.1 Wie ist Bohmsche Mechanik mit der Quantenmechanik vereinbar?

Wie ist nun eine deterministische Theorie mit der herkömmlichen Quantenmechanik vereinbar, deren vielfache experimentelle Bestätigung angeblich bewiesen hat, dass die Zufälligkeit eine fundamentale intrinsische Eigenschaft der Natur ist?

Eine Möglichkeit diese Vereinbarkeit zu gewährleisten könnte darin bestehen, zur Schrödingergleichung und der Führungsgleichung ein drittes Postulat hinzuzufügen. Dieses Postulat müsste lauten:

> Die Teilchenorte $\mathbf{Q}_i$ zu einem Zeitpunkt t_0 sind zufällig gemäß einer Wahrscheinlichkeitsdichte $|\Psi(\mathbf{q},t_0)|^2$ im Raum verteilt.

Der Inhalt dieses Postulats ist als **Quantengleichgewichtshypothese** bekannt. Die Führungsgleichung hat (bereits ohne die Quantengleichgewichtshypothese) eine entscheidende Eigenschaft: Teilchen, die zum Zeitpunkt t_0 gemäß $|\Psi(\mathbf{q}, t_0)|^2$ verteilt sind, werden zu einem beliebigen späteren Zeitpunkt t_1 gemäß $|\Psi(\mathbf{q}, t_1)|^2$ verteilt sein. Es reicht also aus, die Teilchenverteilung für t_0 zu postulieren, damit die experimentell überprüfbaren Implikationen der Bohmschen Mechanik mit der Bornschen Wahrscheinlichkeitsinterpretation der Wellenfunktion vereinbar sind.

Nach [2] muss die Quantengleichgewichtshypothese jedoch gar nicht angenommen oder postuliert werden, sondern *ergibt sich* aus der Schrödingergleichung und der Führungsgleichung.

Damit ist die Bohmsche Mechanik nicht nur mit der Quantenmechanik vereinbar, sondern ihr gesamter Formalismus geht aus der Bohmschen Mechanik hervor. Die Vertreter der Bohmschen Mechanik, betrachten die Theorie daher meist nicht als eine konkurrierende Theorie, sondern vielmehr als eine Grundlage der Quantenmechanik, die ansonsten unvollständig ist.

2.2 Kritik an der Bohmschen Mechanik

Es hat einige Versuche gegeben, zu beweisen, dass jede Theorie mit sogenannten verborgenen Variablen, die *mehr* über den Zustand eines Systems aussagt als die herkömmliche Quantenmechanik, *inkonsistent* mit den Voraussagen der QM ist. Diese sogenannten No-Hidden-Variables-Theoreme basieren jedoch alle auf Hypothesen, die auf die Bohmsche Mechanik nicht zutreffen. Für weitere Information dazu verweise ich auf Abschnitt 4.9 des einführenden Buchs zur Bohmschen Mechanik von Oliver Passon [5].

Häufige Kritik an der Bohmschen Mechanik bezieht sich jedoch nicht auf die Anwesenheit von Mängeln der Theorie, sondern behauptet eine Abwesenheit von Vorteilen. Es wird dabei meistens argumentiert, dass die Bohmsche Mechanik in ihren experimentell überprüfbaren Implikationen mit der orthodoxen Quantenmechanik vollkommen übereinstimmt, und daher unnötig oder zumindest nicht allzu interessant sei, weil sie keine neuen überprüfbaren Erkenntnisse mit sich bringe.

Andere Kritiker gehen einen Schritt weiter und bemängeln, dass die Bohmsche Mechanik Aussagen macht, die über die prinzipielle experimentelle Überprüfbarkeit hinausgehen, und daher nicht im Bereich der Wissenschaft sondern der Religion anzusiedeln sei. Häufig hört und liest man in diesem Zusammenhang Aussagen, die den folgenden (ausgedachten) Sätzen ähneln:

> „Alleiniges Ziel einer naturwissenschaftlichen Theorie ist es, den Ausgang von Experimenten vorherzusagen. Alles weitere ist Religion und hat in der Wissenschaft nichts zu suchen."

Kritik an der Kritik

Doch kann eine Theorie so etwas überhaupt? Den Ausgang von Experimenten vorhersagen? Läuft die Theorie herum, nimmt den Sachverhalt genau unter die Lupe und macht eine überlegte Vorhersage? Nein. Es sind Menschen und nicht Theorien, die Vorhersagen machen. Theorien sind dafür da, Menschen zu *befähigen*, zutreffende Vorhersagen zu machen.

Qualitätskriterium sollte also nicht nur sein, ob es grundsätzlich möglich wäre mit Hilfe einer Theorie korrekte vorhersagen zu treffen, sondern auch wie leicht es Menschen fällt sich die Theorie zu Nutzen zu machen, sie zu *verstehen*.

Aufgrund gravierender konzeptueller Schwierigkeiten scheitern jedoch viele Menschen beim Versuch sich die Theorie der Quantenmechanik anzueignen oder geben auf, weil sie an ihrem eigenen oder am Verstand der physikalischen Gemeinschaft zu zweifeln beginnen. In diesen Fällen erfüllt die Theorie ihre Funktion also im Lichte der erklärten Betrachtungsweise *nicht*.

Hier soll kein Urteil darüber gefällt werden, ob es Lernenden mit Hilfe der Bohmschen Mechanik leichter fällt, zu einer Form des Verständnisses zu gelangen, mit dem sie in der Lage sind den Ausgang von Experimenten vorherzusagen. Möglicherweise liegt aber in genau diesem Punkt der entscheidende zusätzliche Wert und Nutzen, den die Bohmsche Mechanik hat.

Kapitel 3

Einführung in POV-Ray

POV-Ray ist ein freies Computergrafikprogramm, das es dem Nutzer ermöglicht, mit Hilfe einer sogenannten Szenenbeschreibungssprache dreidimensionale Bilder von geometrischen Objekten berechnen zu lassen. Es ist mit ausreichender Rechenleistung und Zeit möglich, damit eindrucksvolle fotorealistische Bilder zu erstellen:

Abbildung 3.1: Mit POV-Ray berechnetes Bild *Glasses* von Gilles Tran. (`http://en.wikipedia.org/wiki/File:Glasses_800_edit.png`)

Die Tatsache, dass die zu berechnenden Bilder mit Code beschrieben werden, hat für die Zwecke dieser Bachelorarbeit einige Vorteile. POV-Rays Szenenbescheibungssprache ist nämlich eine Turing-vollständige Programmiersprache, in der sprachliche Konstrukte wie Verzweigungen, Schleifen und Makros zur Verfügung stehen. In den nächsten Abschnitten möchte ich kurz einige wichtige Elemente der Szenenbeschreibungssprache ansprechen und erläutern, soweit es für unsere Zwecke nötig ist, um Bilder und schließlich Animationen zu erstellen. Für eine umfangreiche Dokumentation von POV-Ray sowie für den Download des Programms verweise ich auf die Webseite `http://www.povray.org/`.

Für diese Bachelorarbeit wurde POV-Ray in der Version 3.6.1 für Linux verwendet.

3.1 Bilder mit POV-Ray

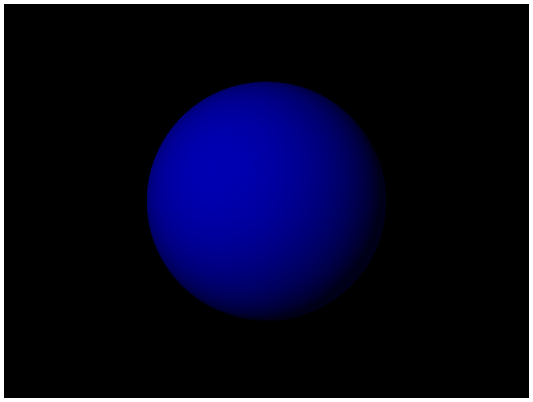
Um ein Bild von einer blauen Kugel mit Radius 5 am Koordinatenursprung zu erzeugen, schreiben wir folgende Befehle in eine `.pov`-Datei:

```
camera {
  angle 45
  location <15, 10, 20>
  look_at <0,0,0>
}

light_source {
  <40,30, 20>
  color <1,1,1> //rgb-vector for
    white
}

sphere {
  <0,0,0>, 5 //center and radius
  pigment {color <0,0,1>} //blue
}
```

blaueKugel.pov



blaueKugel.png

Wie man im Code erkennt, wurde neben der Kugel auch eine Kamera und eine Lichtquelle platziert. Ohne Kamera gibt es eine Fehlermeldung, ohne Lichtquelle ist das entstehende Bild einfach schwarz. Damit POV-Ray jetzt beginnt die `.pov`-Datei mit der Szenenbeschreibung auszuwerten und eine Bilddatei mit der Auflösung 800×600 erstellt, müssen wir in einer Linux-Umgebung den Befehl `povray` mit den folgenden Parametern in der Kommandozeile ausführen:

```
povray blaueKugel.pov +w800 +h600
```

Standardmäßig gibt POV-Ray nun eine `.png`-Datei aus.

3.1.1 Variablen belegen mit #declare

Mit dem Schlüsselwort `#declare` ist möglich in der Szenenbeschreibungssprache Variablen zu deklarieren und diese mit geometrischen Objekten, Zahlen oder Vektoren zu belegen. Es empfiehlt sich dabei Großbuchstaben für die Bezeichner zu verwenden, um zu vermeiden, dass es zu Kollisionen mit Schlüsselwörtern der Szenenbeschreibungssprache kommt. Die letzte Szene kann ebenfalls durch die folgende `.pov`-Datei beschrieben werden:

```
camera {
  angle 45
  location <15, 10, 20>
  look_at <0,0,0>
}

light_source {
  <40,30, 20>
  color <1,1,1>
}

#declare MyRadius = 5;
#declare MyBlue = <0,0,1>;
#declare MySphere = sphere {
  <0,0,0>, MyRadius
  pigment {color MyBlue}
}

object {MySphere}
```

blaueKugelDeclare.pov

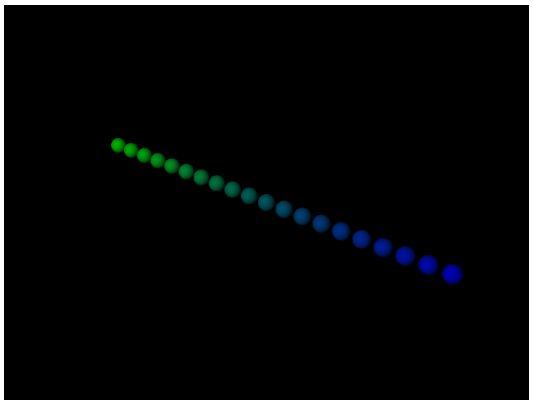
Vektoren und Zahlen können einfach durch die entsprechende Variable ersetzt werden. Um jedoch ein in einer Variablen gespeichertes geometrisches Objekt in der Szene zu platzieren, muss wie in Zeile 19 das Schlüsselwort `object` vorangestellt werden.

3.1.2 Schleifen mit #while

In diesem Abschnitt wird einer der Vorteile einer textbasierten Szenenbeschreibungssprache deutlich. Wir wollen nun nicht nur eine Kugel in der Szene platzieren, sondern 20. Außerdem sollen diese 20 Kugeln nicht alle blau sein, sondern jede Kugel eine etwas andere Farbe haben. Für solche Aufgaben sind Schleifen bestens geeignet:

```
1  camera {
2    angle 45
3    location <20, 15, 25>
4    look_at <0,0,10>
5  }
6
7  light_source {
8    <40,30, 20>
9    color <1,1,1>
10 }
11
12 #declare Index = 0;
13 #while (Index < 20)
14   sphere {
15     <0,0,Index>, 0.4
16     pigment {
17       color <0,1-Index/19,Index
       /19>
18     }
19   }
20   #declare Index = Index + 1;
21 #end
```

loopDemonstration.pov



loopDemonstration.png

3.1.3 Verzweigungen mit #if

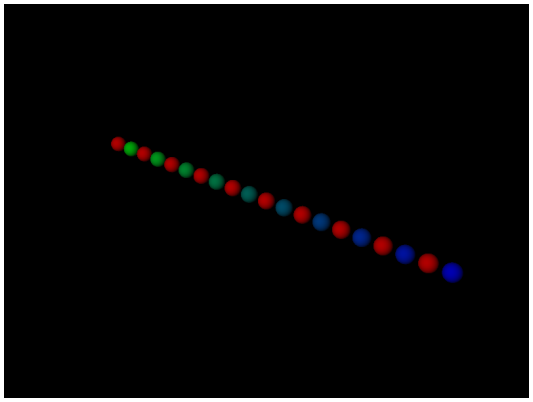
Das nächste wichtige Sprachelement von POV-Ray sind Verzweigungen. Diese werden mit dem Schlüsselwort `#if` eingeleitet. Wir würden nun gerne eine Szene beschreiben, die sich nur im Folgenden von der letzten Szene unterscheidet: Jede zweite Kugel soll nun rot sein. Dazu fügen wir in die `#while`-Schleife eine bedingte Anweisung ein:

```
camera {
  angle 45
  location <20, 15, 25>
  look_at <0,0,10>
}

light_source {
  <40,30, 20>
  color <1,1,1>
}

#declare Index = 0;
#while (Index < 20)
  sphere {
    <0,0,Index>, 0.4
    pigment {
      #if (mod(Index,2) = 0)
        color <1,0,0> //red
      #else
        color <0,1-Index/19,Index/19>
      #end
    }
  }
  #declare Index = Index + 1;
#end
```

ifDemonstration.pov



ifDemonstration.png

Alle Anweisungen im `#if`-Zweig werden nur ausgeführt, wenn die zugehörige Bedingung erfüllt ist. Bei Verwendung eines `#else`-Zweiges werden alle darin enthaltenen Anweisungen ausgeführt, wenn die `#if`-Bedingung nicht erfüllt ist.

3.1.4 Makros mit #macro

Wir haben bereits gesehen wie wir mit `#declare` Zahlen, Vektoren und Objekte speichern können. So ähnlich können wir mit dem Schlüsselwort `#macro` ganze Folgen von Anweisungen zusammenfassen, um diese an anderer Stelle aufzurufen. In POV-Ray können Makros parametrisiert werden, sodass offene Parameter erst beim Aufruf festgelegt werden müssen. Makros können also nützlich sein, wenn wir eine Folge von Anweisungen immer wieder aufrufen müssen. Nicht zuletzt können sie helfen den Code modular aufzubauen, sodass er übersichtlicher und leichter zu ändern ist.

3.1.5 Include-Dateien

Schnell werden `.pov`-Dateien für komplexere Szenen groß und unübersichtlich. Es ist möglich Teile des Codes für eine Szene in andere Dateien, sogenannte Include-Dateien, auszulagern und bei Bedarf auf deren Inhalt zuzugreifen. Diese Dateien haben die Endung `.inc` und können mit dem Schlüsselwort `#include` in eine `.pov`-Datei eingebunden werden. In die Include-Dateien schreibt man mit der gleichen Syntax, jedoch können aus ihnen nicht direkt Bilddateien erstellt werden. Bilddateien müssen weiterhin aus den `.pov`-Dateien berechnet werden.

Glücklicherweise beinhaltet die normale POV-Ray-Installation bereits einige nützliche Include-Dateien. Eine der wichtigsten davon ist die Datei `colors.inc`. Bindet man sie ein, so kann man für Farben englische Wörter verwenden und muss dafür nicht mehr `<r,g,b>`-Vektoren angeben. Unsere allererste Szene lässt sich damit folgendermaßen beschreiben:

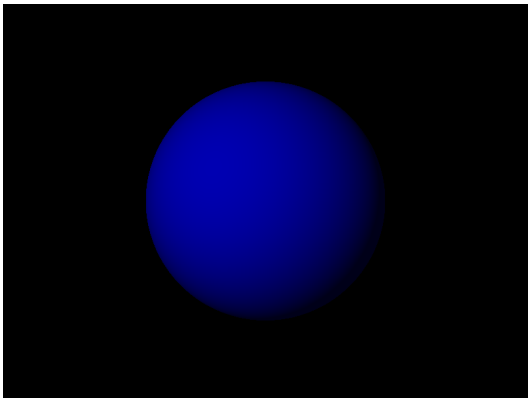
```
#include "colors.inc"

camera {
  angle 45
  location <15, 10, 20>
  look_at <0,0,0>
}

light_source {
  <40,30, 20>
  color White
}

sphere {
  <0,0,0>, 5
  pigment {color Blue}
}
```

blaueKugelInclude.pov



blaueKugelInclude.png

3.2 Animationen mit POV-Ray

Ziel dieser Bachelorarbeit war es Videos zu erstellen. Daher möchte ich nun kurz auf die wichtigsten Aspekte eingehen, was die Erstellung von Animationen mit POV-Ray betrifft. Ein Film entsteht bekanntlich, wenn man einzelne, in der Regel zusammenhängende Bilder schnell genug nacheinander zeigt. Dazu müssen in unserem Fall all diese Bilder mit POV-Ray einzeln berechnet werden. Alles was also bereits über die Erzeugung von Bilddateien geschrieben worden ist, behält auch für Animationen seine Gültigkeit. Bilder, aus denen später Videodateien zusammengefügt werden sollen, nennen wir von nun an *Frames*.

3.2.1 Der globale Parameter Zeit

Während in einem Bild die Zeit still steht, spielt sie in einer Animation eine ganz entscheidende Rolle. Wir können sie uns vorstellen wie einen globalen Parameter `clock`, deren Wert wir vor der Erzeugung eines jeden Bildes neu festlegen müssen. Alles was sich im Ablauf der Animation verändern soll muss in der Szenenbeschreibung von dem Parameter `clock` abhängig sein.

Das wollen wir am Beispiel einer wachsenden Kugel konkretisieren. Damit die Kugel anwächst, muss ihr Radius in irgendeiner Art mit dem linear wachsenden Parameter `clock` größer werden. Für das erste Frame (also für `clock=0`) könnte unsere `.pov`-Datei so aussehen:

```
#declare clock = 0.0;
#declare Radius = 1.0 + 5*clock;
sphere {
  <0,0,0>, Radius
  pigment {color Green}
}
```

Zwischen den einzelnen Frames soll nun jeweils eine Zeitspanne von 0.1 Einheiten vergehen. Dann würde der Code für das zweite Frame lauten:

```
#declare clock = 0.1;
#declare Radius = 1.0 + 5*clock;
sphere {
  <0,0,0>, Radius
  pigment {color Green}
}
```

Bis auf den globalen Parameter `clock` hat sich in der Szenenbeschreibung gar nichts geändert. Diese Beobachtung zusammen mit dem Fakt, dass der globale Parameter auch in Kontrollstrukturen wie `#if`-Verzweigungen eingesetzt werden kann, legt nahe, dass jeder beliebige

Szenen*verlauf* durch eine parametrisierte Szenenbeschreibung ausgedrückt werden kann. Es wäre also toll, wenn man automatisiert eine Vielzahl von Frames mit nur einem Befehl und einer parametrisierten `.pov`-Datei erzeugen könnte! Dazu kommen wir im nächsten Abschnitt.

3.2.2 INI-Dateien

Wenn Bilder mit Hilfe eines Kommandozeilenbefehls erstellt werden, können dem Befehl Optionen wie die Auflösung oder der Dateiname des entstehenden Bildes angefügt werden. INI-Dateien bieten die Möglichkeit diese Optionen zu speichern. Es muss dann der `povray`-Befehl auf die INI-Datei angewendet werden, die einen Verweis auf die auszuwertende `.pov` -Datei enthält.

Gleichzeitig liefern INI-Dateien den Schlüssel zu unserem Vorhaben Animationen zu erstellen. Mit den INI-Dateien ist es möglich, eine ganze Folge von Bilddateien erzeugen zu lassen und dabei für jedes Frame einen neuen Wert für den globalen Zeitparameter `clock` zu setzen.

In der INI-Datei wird unter anderem festgelegt wie viele Frames für das Video insgesamt berechnet werden sollen. Diese Zahl hängt von zwei Fragen ab:

- Wie hoch soll die Bildfrequenz sein?
- Wie lang soll das entstehende Video sein?

Die Gesamtanzahl an Frames ist dann das Produkt aus der Bildfrequenz und der Länge des Videos. Genau genommen muss man noch eins addieren, wenn man möchte, dass exakt die gewünschte Zeit zwischen dem ersten und dem letzten Frame vergeht. Für die Bildfrequenz sollte mindestens ein Wert von etwa 20 Hz gewählt werden, damit das Video den meisten Menschen ruckelfrei erscheint.

Wir kommen zurück zum Beispiel der anwachsenden Kugel. Um die Frames für ein nur 5 Sekunden dauerndes Video mit einer Bildfrequenz von 20 Hz zu erstellen, schreiben wir die folgende `.ini`-Datei:

```
1  +W800                                  ← Setzt Breite auf 800 Pixel.
2  +H600                                  ← Setzt Höhe auf 600 Pixel.
3
4  Input_File_Name=growingSphere.pov      ← Legt zu verwendende .pov-Datei fest.
5  Output_File_Name=growingSphere.png     ← Legt die Namen der auszugebenden .png-Dateien
6                                           fest. Sie erhalten bei Erzeugung noch eine Num-
7  Initial_Frame=1                          merierung.
8  Final_Frame=101                        ← Das erstes Frame bekommt die Nummer 1.
9                                         ← Die Gesamtzahl der Frames soll 20 × 5 + 1 = 101
10 Initial_Clock=0                          sein.
11 Final_Clock=1                          ← Der Zeitparameter clock soll beim ersten Frame
12                                          0 sein und beim letzten Frame den Wert 1 erreichen.
13 Subset_Start_Frame=1                   ← Die Subset-Optionen, gibt es, damit man
14 Subset_End_Frame=101                     für Testzwecke eine Teilmenge der Frames berechnen
                                            lassen kann.
```

Wie im Code-Listing zu sehen ist, beginnt der Parameter `clock` beim ersten Frame mit dem Wert 0 und endet beim letzten Frame mit dem Wert 1. Zwischen aufeinander folgenden Frames wird `clock` um jeweils den gleichen Wert, in unserem Fall um $\frac{1}{101-1} = 0.01$, erhöht.

In der Szenenbeschreibung muss und sollte keine Variable `clock` deklariert werden. Stattdessen kann `clock` einfach in jeder `.pov`- und `.inc`-Datei so verwendet werden, als *hätte* man eine solche Variable deklariert und auf den richtigen Wert gesetzt. Für die Frames der Animation mit der wachsenden Kugel haben wir die folgende Szenenbeschreibung in der Datei `growingSphere.pov`:

```
1  #include "colors.inc"
2
3  camera {
4    angle 45
5    location <15, 10, 20>
6    look_at <0,0,0>
7  }
```

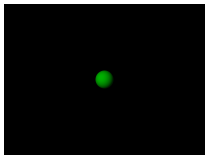
```
8
9  light_source {
10   <40,30, 20>
11   color White
12 }
13
14 #declare Radius = 1.0 + 5*clock;
15 sphere {
16   <0,0,0>, Radius
17   pigment {color Green}
18 }
```

growingSphere.pov

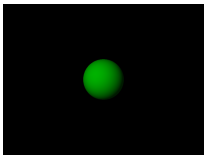
Um nun die Berechnung aller Frames zu starten, wenden wir den Kommandozeilenbefehl `povray` auf die *INI-Datei* an, und *nicht* auf die `.pov`-Datei:

```
1 povray growingGreenSphere.ini
```

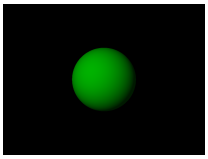
Jetzt werden alle 101 Frames berechnet und erscheinen nacheinander in dem Verzeichnis, in dem sich auch `growingSphere.pov` und `growingSphere.ini` befinden. Hier sind exemplarisch fünf der erzeugten Bilddateien abgebildet:



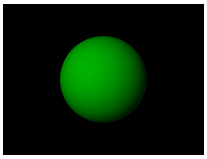
growing-Sphere001.png



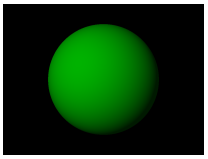
growing-Sphere026.png



growing-Sphere051.png



growing-Sphere076.png



growing-Sphere101.png

3.2.3 Frames zu Videodatei zusammenfügen

Aus den `.png`-Dateien soll nun eine `.mp4`-Datei erzeugt werden. Dazu eignet sich das Kommandozeilenprogramm `ffmpeg`. Mit dem folgenden Befehl werden die 101 Frames aus unserem Beispiel zu einer `.mp4`-Datei zusammengefügt:

```
1 ffmpeg -r 20 -qscale 1 -i growingSphere%03d.png growingSphere.mp4
```

Erklärung der Optionen:

`-r 20`	← setzt Bildfrequenz auf 20 Hz
`-qscale 1`	← 1 für höchste Qualität
`-i growingSphere%03.png`	← legt zu verarbeitende Bilddateien fest
`growingSphere.mp4`	← bestimmt Namen und Dateiformat der Ausgabe

Im gleichen Ordner erscheint nun die Videodatei `growingSphere.mp4`.

Kapitel 4

Einführung in MATLAB

Neben dem Grafikprogramm POV-Ray wurde in dieser Bachelorarbeit auch das Programm MATLAB verwendet. MATLAB ist ein Programm, das darauf optimiert ist numerische Berechnungen speziell unter Zuhilfenahme von Matrizenkalkül durchzuführen. Es enthält viele eingebaute Funktionen und Möglichkeiten für eine einfache Grafikausgabe. Das begünstigt eine zeitsparende Entwicklung erheblich.

MATLAB wurde in dieser Arbeit insbesondere verwendet, um Quantentrajektorien numerisch zu berechnen und diese in eine für POV-Ray lesbare `.pov`-Datei zu exportieren. In diesem Abschnitt wird daher kurz diejenigen Grundlagen von MATLAB eingegangen, die für diese Arbeit relevant sind. Darüber hinaus verweise ich auf die offizielle Webseite von Mathworks, dem Hersteller von MATLAB:

`http://www.mathworks.de/products/matlab/`

Für diese Bachelorarbeit wurde MATLAB in der Version *R2011a Student Version* verwendet.

4.1 Eigene Funktionen in MATLAB

In aller Regel muss in MATLAB jede Funktion in eine eigene Datei geschrieben werden. Eine solche Datei bekommt die Endung `.m`. Als Beispiel implementieren wir eine Funktion, welche die Energie E eines Photons der Kreisfrequenz ω nach der Formel $E(\omega) = \hbar\omega$ berechnet. Das Argument der Funktion ist ω, im Code `omega`, und der Rückgabewert die Energie E, im Code `energy`. Der Wert des reduzierten planckschen Wirkungsquantums $\hbar$, im Code `hbar`, wird im Rumpf der Funktion in der Einheit Joule*Sekunde festgelegt:

```
function energy = energyOfPhoton(omega)
  hbar =  1.05457*e-34;                  % comment: e-34=10^(-34)
  energy = hbar*omega;
end
```

Ein großer Vorteil von MATLAB besteht nun darin, dass diese Funktion direkt in einer Kommandozeile isoliert getestet und benutzt werden kann, ohne dass sie erst in einem größeren Programm eingebaut werden muss. Die MATLAB-Kommandozeile sei hier durch `>>` symbolisiert. Aufgerufen wird die Funktion für das Argument `omega=300` mit:

```
>> energyOfPhoton(300)
```

MATLAB antwortet dann mit:

```
ans =

   3.1637e-32
```

4.2 Globale Variablen

Normalerweise geht in eine MATLAB-Funktion nur das hinein, was man ihr als Funktionsargument übergibt. Alles andere muss im Code der Funktion selbst festgelegt werden, und es gibt von außen kein Zugriff darauf. Mit dem Schlüsselwort `global` ist es in MATLAB jedoch möglich, globale Variablen zu deklarieren und an anderer Stelle (z.B. in einer Funktion) anzugeben, dass diese Variablen dort gültig sein sollen. Das ist von Vorteil, wenn es einen Parameter gibt, der in sehr vielen Funktionen vorkommt, den man aber nicht bei jedem Funktionsaufruf als Argument eingeben möchte, weil er die meiste Zeit gleich bleibt. In dieser Arbeit war das für einige grundsätzliche Parameter der physikalischen Ausgangssituation gegeben. Ich möchte den Einsatz von `global` am Beispiel Photonenenergie demonstrieren. Statt wie zuvor beschrieben den Wert der Variablen `hbar` im Rumpf der Funktion festzulegen, deklarieren wir `hbar` in einer Datei `globalParameters.m` als *globale* Variable und legen dort den Wert fest:

```
global hbar;
hbar = 1.05457*e-34;
```

In der neuen Variante der Funktion `energyOfPhoton` legen wir, ebenfalls mit dem Schlüsselwort `global`, fest, dass die globale Variable `hbar` an dieser Stelle Gültigkeit besitzt und verwendet werden kann:

```
function energy = energyOfPhoton(omega)
  global hbar;
  energy = hbar*omega;
end
```

So kann für alle Funktionen verfahren werden, die für `hbar` einen Wert benötigen. Ändert man dann den Wert in der Datei `globalParameters.m`, so bewirkt das auf alle Funktionen eine Änderung, in denen `global hbar` verwendet wird.

Kapitel 5

Numerisches Handwerkszeug

In diesem Abschnitt sollen einfache numerische Verfahren erklärt werden, die sowohl bei den Animationen zum unendlich tiefen Potentialtopf als auch zum Doppelspaltversuch eingesetzt wurden.

5.1 Numerische Integration der Führungsgleichung

In diesem Abschnitt geht es darum, wie die Bohmsche Trajektorie eines Teilchens numerisch berechnet werden kann. Dafür erinnern wir uns daran, dass das Verhalten von Teilchen in der Bohmschen Mechanik durch die sogenannte Führungsgleichung aus Kapitel 2 beschrieben wird:

$$\frac{\mathrm{d}\mathbf{Q}_i}{\mathrm{d}t} = \frac{\hbar}{m_i} \operatorname{Im}\left(\frac{\nabla_i \Psi}{\Psi}(\mathbf{Q}, t)\right) \tag{5.1}$$

Diese Gleichung beschreibt, wie ein Teilchen von der Wellenfunktion bewegt wird. Um das Prinzip des Verfahrens zu erklären, beschränken wir uns auf nur eine räumliche Dimension und nur ein vorhandenes Teilchen, wodurch sich die Gleichung zu

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{\hbar}{m} \operatorname{Im}\left(\frac{1}{\Psi(x,t)} \frac{\partial \Psi}{\partial x}\right) \tag{5.2}$$

vereinfacht. Im infinitesimal kleinen Zeitschritt $\mathrm{d}t$ bewegt sich das Teilchen um $\mathrm{d}x = \frac{\mathrm{d}x}{\mathrm{d}t}\mathrm{d}t$ in die x-Richtung. Um auszurechnen, wie weit sich das Teilchen in einer bestimmten Zeitspanne fortbewegt hat, muss das Integral

$$x(t_{\text{final}}) - x(t_{\text{start}}) = \int_{t_{\text{start}}}^{t_{\text{final}}} \frac{\mathrm{d}x}{\mathrm{d}t}\,\mathrm{d}t = \int_{t_{\text{start}}}^{t_{\text{final}}} \frac{\hbar}{m} \operatorname{Im}\left(\frac{1}{\Psi(x,t)} \frac{\partial \Psi}{\partial x}\right)\,\mathrm{d}t \tag{5.3}$$

gelöst werden. Das Integral entspricht einer Aufsummierung der infinitesimalen Änderungen im Ort des Teilchens. Wir können diese Aufsummierung numerisch durchführen, indem wir uns mit *endlich* kleinen Zeitschritten und Ortsänderungen zufrieden geben:

$$x(t_{\text{final}}) - x(t_{\text{start}}) \approx \sum \dot{x}(t)\Delta t \tag{5.4}$$

Diese Näherung wird umso genauer, je kleiner der Zeitschritt Δt gewählt wird.

Implementierung

Für das folgende Codebeispiel soll angenommen werden, dass für die Wellenfunktion $\Psi(x,t)$ bereits eine Funktion `waveFunction(x,t)` vorliegt und verwendet werden kann. Auch die Ableitung $\frac{\partial \Psi}{\partial x}$ soll bereits in Form der Funktion `waveFunctionDerivativeX(x,t)` implementiert sein und zur Verfügung stehen. In MATLAB-Syntax sieht dann eine Funktion, die für einen gegebenen Anfangsort `xStart` den Ort `x` zurückgibt, den das Teilchen nach dem Zeitabschnitt `timeIntervall` erreicht hat, so aus:

```
1 function x = position(xStart,timeIntervall,dt)
2 global hbar;
3 global mass;
```

```
t = 0;
x = xStart;
while (t <= timeIntervall)
  dx = hbar/mass * imag(1/waveFunction(x,t)*waveFunctionDerivativeX(x,
    t));
  x = x + dx;
  t = t + dt;
end
```

Die Funktion verwendet die globalen Variablen `hbar` und `mass`, deren Wert in einer anderen Datei festgelegt sein müssen. Vor dem ersten Durchlauf die while-Schleife werden `t` und `x` mit den Startwerten initialisiert. In der Schleife wird zuerst die kleine Ortsänderung `dx` berechnet. Hierbei wird die von MATLAB bereitgestellte Funktion `imag` verwendet, um den Imaginärteil einer komplexen Zahl zu erhalten. Daraufhin werden die Werte des Ortes und der Zeit aktualisiert. Das passiert so oft bis `t` den Wert des Funktionsarguments `timeIntervall` überschreitet. Rückgabewert der Funktion ist dann der letzte Wert von `x`, also der aktuelle Ort des Teilchens.

Für den Zeitschritt `dt` muss ein ausreichend kleiner Wert gewählt werden, damit die numerische Abweichung von der exakten Teilchenposition nicht zu groß ist. Je kleiner er gewählt wird, desto länger dauert jedoch die Berechnung, weil die Schleife häufiger durchlaufen werden muss. Es muss also ein guter Kompromiss aus Präzision und Rechenzeit gefunden werden. Wie dieser Wert im Einzelfall zu wählen ist, findet man leicht durch einige Tests heraus.

Diese Funktion `position(xStart,timeIntervall,dt)` gibt nun natürlich nicht direkt eine Trajektorie im Sinne einer parametrisierten Bahn zurück, sondern lediglich einen Punkt, wo sich das Teilchen unter den angenommenen Umständen befinden würde. Der Code ließe sich jedoch leicht modifizieren, sodass er etwas produziert was wir eher als als Trajektorie akzeptieren können. Fügt man in die Schleife etwa eine Anweisung hinzu, die bewirkt, dass nach jeder Iteration ein Paar (x_i, t_i) in eine Datei exportiert wird, so kann die resultierende Menge von Paaren als Trajektorie bezeichnet werden, sofern aufeinander folgenden Paare genügend nah (räumlich und zeitlich) beieinander liegen. Dieser Datei kann schließlich entnommen werden, wo sich das Teilchen zu den Zeitpunkten t_i befindet.

Numerische Differentiation

Im letzten Codebeispiel wurde angenommen, dass für Ableitung $\frac{\partial \Psi}{\partial x}$ bereits eine implementierte Funktion `waveFunctionDerivativeX(x,t)` vorliegt und in der Schleife aufgerufen werden kann. Doch wie implementiert man diese Funktion, wenn sie noch nicht vorliegt?

In einfachen Fällen ist es möglich $\frac{\partial \Psi}{\partial x}$ analytisch zu berechnen. Dann kann man mit Bleistift und Papier die Ableitung berechnen und muss diese nur noch in MATLAB-Syntax eintippen. So wurde es bei dieser Bachelorarbeit für den einfachen Fall des unendlich tiefen Potentialtopfes gemacht, beim dem lediglich eine Summe von Sinusfunktionen abzuleiten sind.

In anderen Fällen ist eine analytische Berechnung der Ableitung nicht möglich oder sehr aufwendig. Dann kann auch die Ableitung $\frac{\partial \Psi}{\partial x}$ an einer *bestimmten* Stelle (x_0, t_0) numerisch berechnet werden. Dazu erinnere man sich an die Definition der Ableitung als Grenzwert eines Differenzenquotienten:

$$\left.\frac{\partial \Psi(x,t_0)}{\partial x}\right|_{x_0} = \lim_{\Delta x \to 0} \frac{\Psi(x_0+\Delta x, t_0) - \Psi(x_0,t_0)}{\Delta x} \tag{5.5}$$

Auch hier schaffen wir den Sprung von der Analysis zur Numerik, indem wir uns mit *endlich* kleinen Werten für Δx zufrieden geben. Dieser Wert für Δx soll an die Funktion als Parameter `dx` übergeben werden. In MATLAB-Syntax kann die Funktion `waveFunctionDerivativeX (x,t,dx)` dann auf die folgende Weise implementiert werden:

```
function dPsiBydx = waveFunctionDerivativeX(x,t,dx)
    dPsi = (waveFunction(x+dx,t) - waveFunction(x,t));
    dPsiBydx = dpsi/dx;
end
```

Die Funktion gibt schlichtweg den Wert des Differenzenquotienten zurück. Damit dieser Wert brauchbar ist, muss für `dx` ein ausreichend kleiner Wert gewählt werden.

5.2 Anfangsverteilung der Teilchen

Nach der Quantengleichgesichtshypothese sind die Teilchen in einem Bohmschen Universum gemäß der Wahrscheinlichkeitsdichte $\rho = |\Psi|^2$ verteilt. Doch was bedeutet das wirklich? Es bedeutet, dass sich die tatsächlich gemessene Teilchendichte mit der Anzahl der Messungen *wahrscheinlich* der Wahrscheinlichkeitsdichte annähert. Es ist jedoch möglich, dass die Teilchen in einer endlichen Sequenz von Messungen völlig anders verteilt sind. Etwa so wie es durchaus vorkommen kann, dass man mit einem Spielwürfel fünf mal hintereinander eine 1 würfelt und die Verteilung der Ergebnisse von der Gleichverteilung abweicht.

Möchte man eine Bohmsche Trajektorie in einer Animation darstellen, kann man sich die Startposition des Teilchens also aussuchen. Auch die unwahrscheinlichste Startkonfiguration ist physikalisch *möglich*. Einzige zwingende Bedingung ist, dass sich das Teilchen nicht genau bei den Nullstellen der Wahrscheinlichkeitsdichte befindet. Um die Eigenschaften der Bohmschen Trajektorien besser sichtbar zu machen, sollen in einigen Animationen die Trajektorien für mehrere mögliche Anfangspositionen gezeigt werden. Es stellt sich dann die Frage wie diese Positionen gewählt werden sollen. Dafür gibt es grundsätzlich (mindestens) zwei Möglichkeiten:

- Eine zufällige Anordnung
- Eine regelmäßige Anordnung

Was damit gemeint ist, soll an Hand der einfachsten denkbaren Wahrscheinlichkeitsdichte erläutert werden, nämlich an der Gleichverteilung über einen Bereich der Länge L:

$$\rho(x) = \begin{cases} 1/L & \text{für } x \in [0, L] \\ 0 & \text{sonst.} \end{cases} \tag{5.6}$$

Es sollen n repräsentative Teilchen angeordnet werden.

Zufällige Anordnung

Um eine zufällige Anordnung numerisch umzusetzen, würde man einen Pseudozufallszahlengenerator verwenden und von ihm verlangen, dass er neun Zahlen zwischen 0 und L ausgibt. Der Zufallszahlengenerator müsste im Modus der Gleichverteilung verwendet werden, sodass sich die Verteilung der ausgegebenen Werte einer Gleichverteilung annähert.

Regelmäßige Anordnung

Mit einer *regelmäßigen* Anordnung der n Teilchen ist folgendes gemeint:

> Je zwei benachbarte Teilchen sollen so an den Orten x_i und x_{i+1} positioniert werden, dass die kumulative Wahrscheinlichkeit $\int_{x_i}^{x_{i+1}} \rho(x)\,\mathrm{d}x$ zwischen den beiden Positionen $\frac{1}{n+1}$ beträgt.

Zwischen je zwei Teilchen soll also die gleiche Fläche unter dem Graphen der Funktion ρ liegen. Außerdem soll zwischen den Rändern des Bereiches und den äußersten Teilchen ebenfalls diese kumulierte Wahrscheinlichkeit P_0 liegen. Die n Teilchen befinden sich dann an den Positionen $(x_1, x_2, \ldots, x_n) = (\frac{L}{n+1}, \frac{2L}{n+1}, \ldots, \frac{nL}{n+1})$ und unterteilen den gesamten Bereich in zehn Abschnitte. Jeder dieser Abschnitte enthält die kumulierte Wahrscheinlichkeit $P_0 = 1/n + 1$.

Diese Art der regelmäßigen Anordnung der Anfangspositionen kann auf beliebige Wahrscheinlichkeitsdichten angewendet werden. Im allgemeinen Fall muss integriert werden, um aus der Wahrscheinlichkeitsdichte Wahrscheinlichkeiten zu erhalten. Der Algorithmus für die Platzierung der Teilchen lässt sich dann etwas vereinfacht wie folgt formulieren:

> Integriere $\rho(x)$ numerisch. Immer wenn das Zwischenergebnis ein natürliches Vielfaches der Wahrscheinlichkeit $\frac{1}{n*1}$ überschreitet, platziere ein Teilchen am momentanen Wert von x.

In MATLAB-Syntax kann eine Funktion `xStartValues(n,dx)`, die für eine gegebene Wahrscheinlichkeitsdichte `probDensity(x)` ein Array mit `n` Startpositionen zurückgibt, wie folgt implementiert werden:

```
function xArray = xStartValues(n,dx)
  x = 0;
  prob = 0;
  xArray = zeros(1,n); % initialize array with n zeros
  index = 1; % index of next particle to be placed
    while (index <= n)
        prob = prob + probDensity(x)*dx;
        if (prob > index/(n+1))
            xArray(index) = x;
            index = index + 1;
        end
    end
end
```

Über den Parameter `dx` wird angegeben wie groß die Schritte für die numerische Integration sein sollen. Hier besteht wieder der in den letzten Abschnitten erklärte Kompromiss zwischen Präzision und Rechenzeit.

Es kann kritisiert werden, dass diese Anordnung völlig künstlich ist und möglicherweise beim Betrachter den falschen Eindruck erweckt, dass diese Anordnung aus einem physikalischen Gesetz folge, was nicht der Fall ist. Für einige der Animationen wurde eine derartige Anordnung der Startpositionen jedoch gewählt, weil damit wichtige Eigenschaften der Trajektorien besonders deutlich sichtbar werden.

Allgemeine zufällige Anordnung

Sollen die Teilchen nicht in dieser regelmäßigen Anordnung verteilt werden, sondern zufällig gemäß einer allgemeinen Wahrscheinlichkeitsdichte werden die beiden Methoden miteinander kombiniert. Zu erst werden n Zufallszahlen im Modus der Gleichverteilung ausgegeben. Sie teilen den gesamten Abschnitt in $n + 1$ Teile. Diese $n + 1$ Teile legen nun die Größe der Wahrscheinlichkeitsabschnitte fest, nach deren Überschreitung bei der numerischen Integration je ein Teilchen platziert werden soll. So entsteht eine tatsächliche Teilchendichte, die sich mit steigender Teilchenzahl (wahrscheinlich) der gewählten Wahrscheinlichkeitsdichte annähert, aber nicht die Regelmäßigkeit aufweist, die bei der zuvor erklärten Methode auftritt.

Kapitel 6

Videos zum unendlich tiefen Potentialtopf

6.1 Die Physik des unendlich tiefen Potentialtopfs

Die folgende Behandlung des quantenmechanischen unendlich tiefen Potentialtopfs orientiert sich an Kapitel 2.2 des Quantenmechanik Lehrbuchs von D. J. Griffiths [4].

Wir betrachten einen eindimensionalen potentialfreien Bereich der Länge L, welcher an beiden Enden von einer unendlich großen Potentialschwelle begrenzt wird. Wir haben also das eindimensionale Potential

$$V(x) = \begin{cases} 0 & \text{für } x \in [0, L] \\ \infty & \text{sonst.} \end{cases} \tag{6.1}$$

Die Frage ist nun, welche Wellenfunktionen für dieses Potential möglich sind. Erlaubte Wellenfunktionen $\Psi(x,t)$ müssen der Schrödingergleichung

$$\mathrm{i}\hbar\frac{\partial}{\partial t}\Psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,\,t) + V(x)\Psi(x,\,t) \tag{6.2}$$

genügen.

Wie aus der Quantenmechanik bekannt ist, lässt sich die allgemeine Lösung für die Schrödingergleichung zusammensetzen als eine Summe von stationären Lösungen $\psi_k(x)$, die jeweils mit einem zeitabhängigen Phasenfaktor multipliziert werden. Um diese stationären Lösungen zu finden, betrachten wir die zeitunabhängige Schrödingergleichung. Im Inneren des Potentialtopfes, wo $V(x) = 0$ ist, hat sie die Form

$$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi_k}{\mathrm{d}x^2} = E_k\psi_k(x) \tag{6.3}$$

Wir identifizieren den Index k mit der Wellenzahl $k = \frac{2\pi}{\lambda}$ und stellen mit Hilfe der De-Broglie-Gleichung $\lambda = \frac{h}{p}$ einen Zusammenhang zwischen dem Index k und der zugehörigen Energie E_k her:

$$E_k = \frac{p^2}{2m} = \left(\frac{h}{\lambda}\right)^2\frac{1}{2m} = \frac{h^2k^2}{(2\pi)^2}\frac{1}{2m} = \frac{\hbar^2k^2}{2m} \tag{6.4}$$

Nun können wir diesen Zusammenhang in Gleichung 6.3 einsetzen und erhalten

$$\frac{\mathrm{d}^2\psi_k}{\mathrm{d}x^2} = -k^2\psi_k(x) \tag{6.5}$$

Lösungen sind also Funktionen die nach zweifacher Ableitung bis auf den negativen Vorfaktor $-k^2$ wieder ihre ursprüngliche Form haben. Wir setzen daher an mit

$$\psi_k(x) = A\sin(kx) + B\cos(kx), \tag{6.6}$$

wobei A und B beliebige komplexe konstanten sind. Ähnlich wie bei einem schwingenden Seil, welches an den beiden Randpunkten $x = 0$ und $x = L$ befestigt ist, sind hier jedoch nur Wellenfunktionen möglich, die an den Randpunkten zu jedem Zeitpunkt Nullstellen

haben. Für die Wellenfunktion kann man das so begründen, dass $\Psi(x,t)$ überall stetig in x sein muss, und aufgrund des unendlich hohen Potentials außerhalb des Topfes null ist. Es muss also gelten $\Psi(0,t) = \Psi(L,t) = 0$. Diese Randbedingungen gelten natürlich auch für die stationären Lösungen, denn sie sind Spezialfälle der allgemeinen Lösung. Aus $\psi_k(0) = B\cos(0) = 0$ folgt $B = 0$. Übrig bleibt $\psi_k(x) = A\sin(kx)$. Damit die Wellenfunktion am rechten Rand des Potentialtopfes die Bedingung $\psi_k(L) = 0$ erfüllt, muss das Argument kL ein ganzzahliges Vielfaches von π sein. Das heißt

$$k = \frac{n\pi}{L} \text{ mit } n \in \{1, 2, 3, \dots\}. \tag{6.7}$$

Wir können den Index k also für eine einfachere Handhabung durch n ersetzen und die Lösungen lauten damit:

$$\boxed{\psi_n(x) = A\sin(k_n x) = A\sin\left(n\pi\frac{x}{L}\right).} \tag{6.8}$$

Für die Normierungskonstante ergibt sich durch die Bedingung

$$1 = \int_0^L |\psi_n(x)|^2 \,\mathrm{d}x \tag{6.9}$$

der Wert $A = \sqrt{2/L}$.

Auch die Eigenenergien E_n können jetzt in Abhängigkeit von n ausgedrückt werden:

$$E_n = \frac{\hbar^2 k_n^2}{2m} = \frac{n^2\pi^2\hbar^2}{2mL^2} \tag{6.10}$$

Zeitabhängigkeit

Wie bei einer stehenden Welle auf einem an zwei festen Enden gespannten Seil muss die Länge des Wellenträgers also ein Vielfaches der halben Wellenlänge sind. Der Unterschied zum schwingenden Seil besteht jedoch in der Bewegung, welche nun den von den stationären Lösungen ausgeführt wird. Während das Seil im einfachsten Fall in einer Ebene mit einer reellen Auslenkung schwingt, haben unsere stationären Lösungen komplexe Werte: Denn um auch der zeitabhängigen Schrödingergleichung zu genügen, bekommt jede der stationären Lösungen noch einen komplexen Faktor $\mathrm{e}^{-\mathrm{i}E_n t/\hbar}$. Dieser Faktor führt dazu, dass sowohl eine reelle als auch eine imaginäre Schwingung ausgeführt wird. Stellen wir uns Real- und Imaginärteil auf zwei zur x-Achse orthogonalen Achsen vor, so können wir uns die Bewegung als eine Drehung von ψ_n in der komplexen Ebene um die x-Achse verbildlichen.

Damit wir eine allgemeine zeitabhängige Lösung erhalten, müssen Linearkombinationen der stationären Lösungen mitsamt ihren komplexen Phasenfaktoren gebildet werden:

$$\boxed{\Psi(x,t) = \sum_{n=1}^{\infty} c_n \Psi_n(x,t) = \sum_{n=1}^{\infty} c_n \psi_n(x,t)\,\mathrm{e}^{-\mathrm{i}E_n t/\hbar} \text{ mit } c_n \in \mathbb{C}} \tag{6.11}$$

Durch die Wahl der Koeffizienten c_n, kann nun jede beliebige stetige Funktion zusammengesetzt werden, welche die verlangten Nullstellen an den Enden des Potentialtopfes besitzt.

6.2 Die Idee zur Visualisierung

Da es sich um eine eindimensionale physikalische Problemstellung handelt, stehen in einer dreidimensionalen Grafik noch zwei Dimensionen zu Verfügung, um zu jeder Position x im Potentialtopf, Real- und Imaginärteil der Wellenfunktion darzustellen. Sowohl die stationären Wellenfunktionen als auch deren Überlagerungen sind stetige Funktionen in x und sind darstellbar als Raumkurven, die aus den Punkten

$$\{(x,y,z) | x \in [0,L], y = \mathrm{Re}(\Psi(x,t), z = \mathrm{Im}(\Psi(x,t))\} \tag{6.12}$$

bestehen.

Rote Eigenfunktionen

Jede stationäre Lösung

$$\Psi_n(x,t) = \psi_n(x)\mathrm{e}^{-\mathrm{i}E_n t/\hbar} = \sqrt{\frac{2}{L}}\sin\left(n\pi\frac{x}{L}\right)\mathrm{e}^{-\mathrm{i}E_n t/\hbar} \tag{6.13}$$

bildet dann eine sinusförmige Linie, die zum Zeitpunkt $t = 0$ in der x-y-Ebene liegt. Mit laufender Zeit t dreht sich diese Linie mit der Winkelgeschwindigkeit $\omega_n = E_n/\hbar$ um die x-Achse *ohne ihre Form zu verändern*. Diese stationären Wellenfunktionen sollen in roter Farbe dargestellt werden.

Grüne Wellenfunktion

Zusätzlich zu den roten Eigenfunktionen soll in grüner Farbe die resultierende Wellenfunktion abgebildet werden. Im Gegensatz zu den Eigenfunktionen verändert sich die resultierende Wellenfunktion in ihrer Form. Das liegt daran, dass jede der Eigenfunktionen sich mit einer anderen Winkelgeschwindigkeit dreht.

Blaue Wahrscheinlichkeitsverteilung

Das Betragsquadrat $|\Psi(x,t)|^2$ der grünen resultierenden Wellenfunktion soll als letzte Kurve in blau hinzukommen. Sie ist rein reell und liegt daher zu jedem Zeitpunkt in der x-y-Ebene.

Schwarze Teilchen

Um darzustellen, wie ein Teilchen gemäß der Führungsgleichung der Bohmschen Mechanik bewegt wird, sollen einige schwarze Kugeln im Potentialtopf, d.h. auf der x-Achse zwischen 0 und L, platziert werden wie Perlen auf den Stäben eines Abakus.

Die Schrödingergleichung wurde natürlich nur für *ein* Teilchen gelöst. Doch damit man die Eigenschaften der Bohmschen Trajektorien klarer sieht, sollen mehrere *mögliche* Trajektorien für verschiedene mögliche Anfangspositionen dargestellt werden. Dafür werden die Teilchen zum Zeitpunkt $t = 0$ repräsentativ gemäß der blauen Wahrscheinlichkeitsdichte verteilt. Damit ist gemeint, dass die Dichte der platzierten Kugeln $|\Psi(x,t)|^2$ entspricht. Die Bewegung eines solchen Teilchens soll dann so verlaufen wie es die Bohmsche Mechanik für die ausgewählte Wellenfunktion vorhersagt.

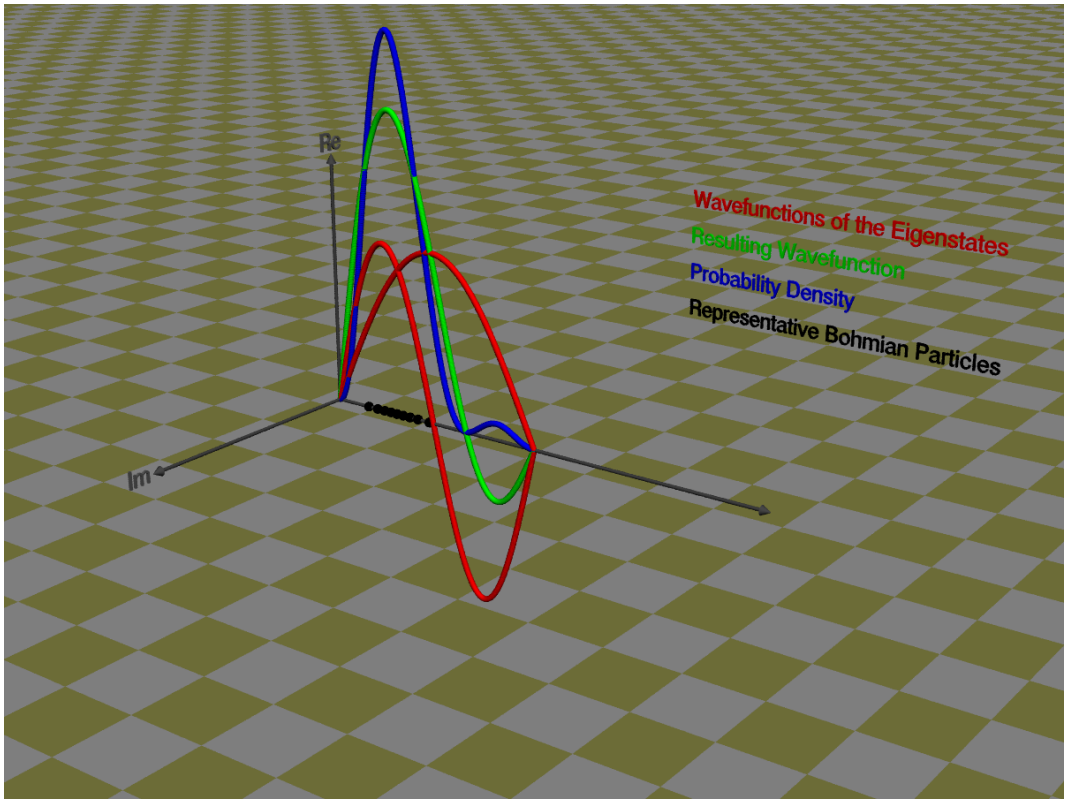
6.3 Frames der erstellten Animation

Die im letzten Abschnitt beschriebene Idee, wie man die Physik des Potentialtopfs visualisieren könnte, wurde umgesetzt für die Wellenfunktion

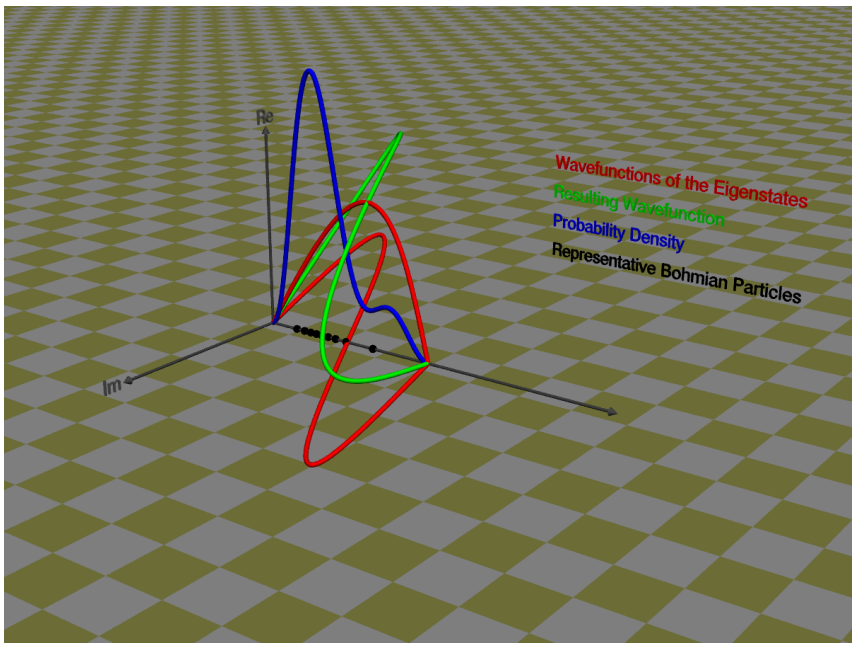
$$\Psi(x,t) = \frac{1}{\sqrt{2}}\Psi_1(x,t)\frac{1}{\sqrt{2}}\Psi_2(x,t). \tag{6.14}$$

Sie ist eine Überlagerung der ersten beiden stationären Lösungen. Außerdem sind neun Teilchen dargestellt, um *mögliche* Teilchentrajektorien zu repräsentieren.

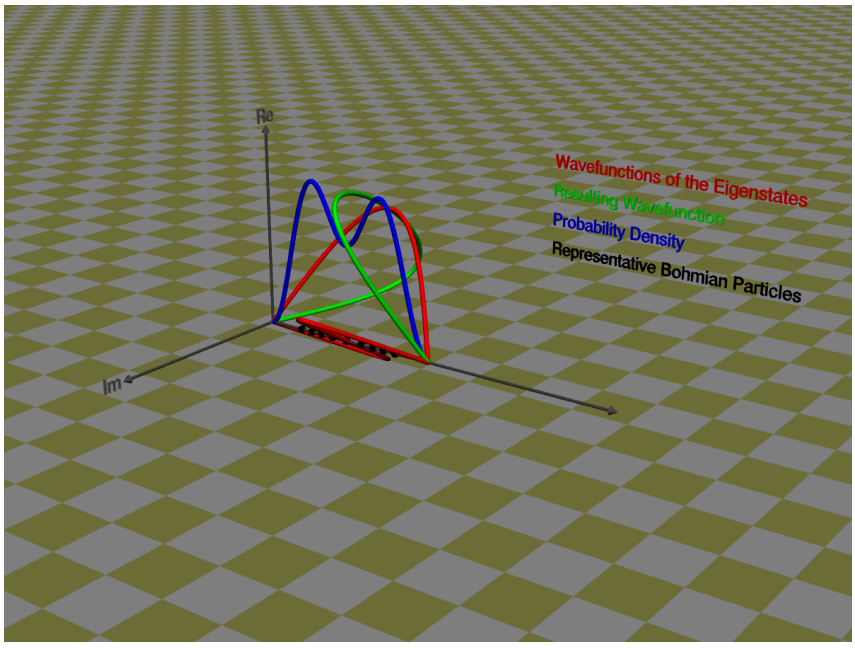
Auf dem Video ist in 400 Frames die Zeitspanne dargestellt, den die erste stationäre Lösung für eine komplette Umdrehung um die x-Achse benötigt. In dieser Zeit dreht sich die zweite stationäre Lösung bereits vier mal um die x-Achse. Das heißt es gibt sieben Momente (inklusive einen zum Start des Videos und zum Schluss), in denen die beiden Raumkurven für die stationären Lösungen in einer gemeinsamen Ebene liegen. Diese sieben Momente teilen die gesamte Zeitspanne in sechs Abschnitte, von denen hier der erste mit Hilfe von fünf ausgewählten Frames dargestellt und erläutert werden soll. Er entspricht der halben Periode des physikalischen Vorgangs.



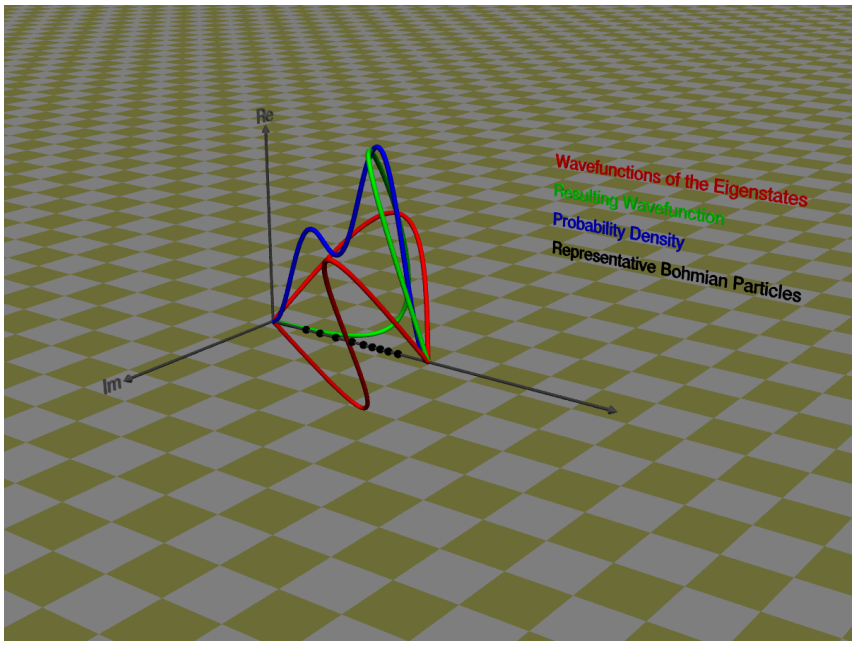
Frame 001: Zum Zeitpunkt $t = 0$ liegen alle Raumkurven in einer Ebene.



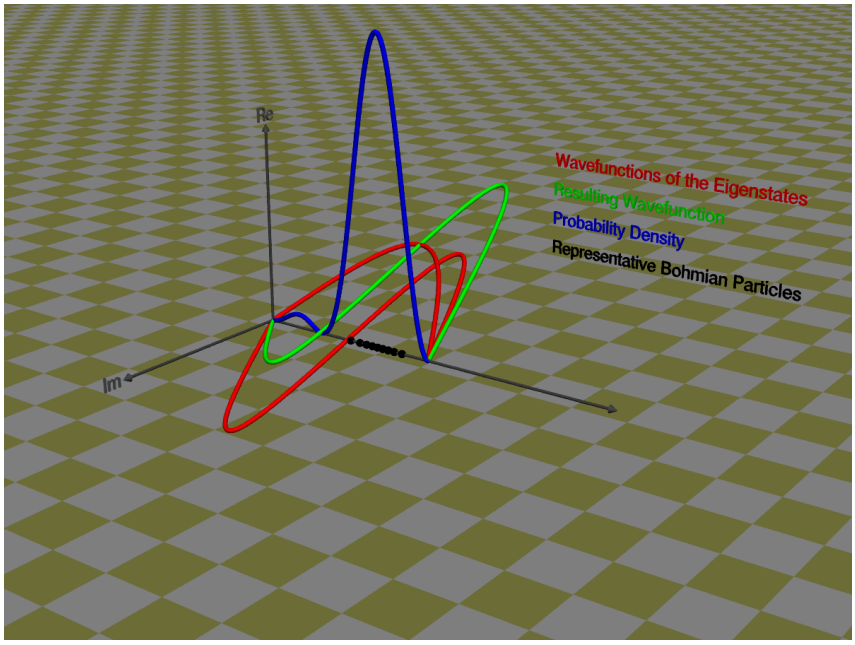
Frame 017: Dadurch, dass sich die roten Raumkurven für die stationären Lösungen verschieden schnell drehen, entsteht Bewegung. Die Form der grünen Kurve für $\Psi(x,t)$ hat deutlich verändert. Die blaue Wahrscheinlichkeitsdichte hat jetzt in der rechten Hälfte höhere (und noch ansteigende) Werte und alle Teilchen bewegen sich jetzt nach rechts.



Frame 034: Im gezeigten Moment liegen die roten Raumkurven in zwei zueinander orthogonalen Ebenen. Deswegen ist die blaue Kurve für $|\Psi|^2$ jetzt symmetrisch um $x = L/2$ Auch die Positionen der Teilchen weisen diese Symmetrie für die verwendeten Startpositionen auf. Sie bewegen sich immer noch nach rechts.



Frame 051: Inzwischen ist das rechte lokale Maximum der blauen Kurve größer als das linke und es befinden sich mehr Teilchen in der rechten Hälfte des Potentialtopfes.



Frame 068: Nach einer gewissen Zeit liegen die Raumkurven der stationären Lösungen wieder in einer gemeinsamen Ebene. Die blaue Kurve für die Wahrscheinlichkeitsdichte hat wieder ihre ursprüngliche Form, jedoch wurde sie gespiegelt an der Ebene $x = L/2$. Zu diesem Zeitpunkt haben die Teilchen die Geschwindigkeit null und werden sich im nächsten Moment nach links bewegen.

6.4 Beobachtungen in der Animation

Folgende bekannte Eigenschaften der Bohmschen Trajektorien können bei Betrachtung der Videos bestätigt werden:

- Die Trajektorien schneiden sich nicht. Das muss so sein, weil die Führungsgleichung eine Bewegungsgleichung *erster* Ordnung ist.
- Wenn die repräsentativen Teilchen am Anfang gemäß $|\Psi(x,t)|^2$ verteilt sind, dann bleiben sie es auch über den gesamten Verlauf der Zeit. Das heißt, wenn ein Teilchen zu Beginn die Fläche unter der blauen Wahrscheinlichkeitsdichte in zwei Teile P_1 links und P_2 rechts unterteilt, so bleiben P_1 und P_2 über die Zeit erhalten.
- Die Teilchen bewegen sich sehr schnell unter aufsteigenden oder herabsinkenden Minima der Wahrscheinlichkeitsdichte hinweg.
- Die Teilchengeschwindigkeit ist proportional zum Gradienten der Phase der Wellenfunktion. Das wird bei genauerer Betrachtung der grünen Wellenfunktion deutlich. Um den Gradienten der Phase zu *sehen*, stelle man sich vor wie ein Zeiger auf den Wert der Wellenfunktion sich in der komplexen Ebene dreht, während man zu einem festen Zeitpunkt t_0 die x-Achse entlang läuft. Der Zeiger setzt an der x-Achse an einem Wert x_0 an und zeigt rechtwinklig davon weg zum Punkt der Wellenfunktion $(x_0, \mathrm{Re}(\Psi(x_0,t_0), \mathrm{Im}(\Psi(x_0,t_0))$. Jetzt stelle man sich vor wie der Zeiger sich unter Verschiebung des gedachten Ansatzpunktes x_0 dreht. Dort wo er sich in dieser Vorstellung am schnellsten dreht, ist der Gradient der Phase am größten und die Teilchen haben dort die größte Geschwindigkeit.

  Durch diese Betrachtung wird deutlich, warum alle Teilchen zu den Zeitpunkten still stehen, zu denen die roten Raumkurven von Ψ_1 und Ψ_2 in einer gemeinsamen Ebene liegen. In diesem Momenten liegen auch alle Punkte der grünen Kurve Ψ in der selben Ebene. Der Gradient der Phase ist dann überall null außer an dem Punkt, wo Ψ die x-Achse schneidet: dort ist der Gradient nicht definiert. Das bringt aber keine Probleme mit sich, denn an dieser Stelle darf sich ohnehin kein Teilchen befinden.

6.5 Implementierung

Für die Implementierung wurde sowohl MATLAB als auch POV-Ray verwendet. In den folgenden Tabellen sind die Dateien aufgelistet, die für die Erstellung des Potentialtopf-Videos geschrieben wurden. Auf der rechten Seite steht jeweils eine kurze Erklärung zum Inhalt der jeweiligen Datei. Für den Fall, dass Interesse an den technischen Details der Implementierung besteht, verweise ich auf den Quellcode, der sich im Anhang befindet.

6.5.1 Überblick MATLAB Quellcode

Datei	Erklärung
`potentialWell.m`	Diese Datei enthält die Festlegung einiger grundlegender Parameter für die physikalische Ausgangssituation sowie für die Animation. Es wird ein Vektor `c = [sqrt(0.5), sqrt(0.5), 0, 0, 0]` für die Koeffizienten c_n deklariert. Außerdem die globalen Variablen: `hbar=1` für den Wert von $\hbar$, `leng=1` für die Länge L des Potentialtopfes, `mass=1` für die Masse m des Teilchens und Parameter für das Video, wie die Anzahl der Frames
`energy.m`	Enthält Definition der Funktion `energy(n)`, die die Energie E_n des n-ten Eigenzustands zurückgibt
`eigenStateWave.m`	Enthält Definition der Funktion `eigenStateWave(n,x,t)`, die den komplexen Wert$\Psi_n(x,t)$ zurückgibt
`wave.m`	Enthält Funktion, die den komplexen Wert der resultierenden Wellenfunktion $\Psi(x,t) = \sum c_n \Psi_n(x,t)$ zurückgibt
`waveStarWave.m`	Funktion, die das Betragsquadrat $\|\Psi(x,t)\|^2$ der Wellenfunktion zurückgibt
`probability.m`	Funktion `probability(c,t,lowerX,upperX, numberOfIntergrationSteps)`, die das Integral $\int \|\Psi(x,t)\|^2 \mathrm{d}x$ von `lowerX` bis `upperX` zurückgibt
`waveDerivativeX.m`	Funktion `waveDerivativeX(c,x,t)`, die den Wert der Ableitung $\frac{\mathrm{d}\Psi}{\mathrm{d}x}$ an der Stelle `x` zum Zeitpunkt `t` zurückgibt
`qDot.m`	Funktion `qDot(c,x,t)`, die für den Ort x und den Zeitpunkt t den Wert des Geschwindigkeitsfeldes zurückgibt, das sich aus der Führungsgleichung der Bohmschen Mechanik ergibt
`q.m`	Funktion `q(c,qStart,dt,endTime)` gibt die zum Zeitpunkt `endTime` zurückgelegte Strecke des am Ort `qStart` zum Zeitpunkt $t = 0$ gestarteten Teilchens zurück. Als Zeitschritt für die numerische Integration wird dabei der Wert des Arguments `dt` verwendet.
`plotQ.m`	Funktion `plotQ(c,qStart,dt,N,endTime,color)` erzeugt einen Plot der Trajektorie eines Teilchens, welches zu $t = 0$ am Ort `qStart` startet. Bei jedem `N`-ten Durchlauf der Integrationsschleife wird ein Punkt (x,t) in das Diagramm gesetzt. Mit dem Parameter `dt` wird der Zeitschritt für die numerische Integration festgelegt.
`initialPositions.m`	Funktion `initialPositions(c,numberOfParticles, numberOfIntergrationSteps)` gibt einen Vektor mit Anfangspositionen für die Teilchen gemäß der Verteilung $\|\Psi(x,t=0)\|^2$ zurück
`printTrajectoryToFile.m`	Funktion `printTrajectoryToFile(c,qStart, stepsBetweenFrames,fileName,particleID)` erzeugt eine für POV-Ray lesbare Datei mit einem Array für die Teilchenpositionen. Jeder Eintrag korrespondiert zu einem Frame. Das Argument `stepsBetweenFrames` legt fest wie viele Integrationsschritte zwischen Arrayeinträgen durchgeführt werden sollen.

6.5.2 Überblick POV-Ray Quellcode

Datei	Erklärung
`potentialWell.ini`	INI-Datei
`potentialWell.pov`	Szenenbeschreibung
`energy.inc`	Enthält ein Makro `Energy(n)`, das die Energie E_n des n-ten stationären Zustands zurückgibt
`eigenFunctions.inc`	Enthält ein Makro `EigenFunctions(c,time)`, das die roten Raumkurven für die stationären Wellenfunktionen zeichnet
`resultingWave.inc`	Enthält ein Makro `ResultingWave(c,time)`, das die grüne Raumkurve der resultierenden Wellenfunktion zeichnet
`probDensity.inc`	Enthält ein Makro `ProbDensCurve(c,time)`, das die blaue Raumkurve für das Betragsquadrat der Wellenfunktion zeichnet
`drawTrajectory.inc`	Enthält neun Makros, die jeweils ein Teilchen im Bild platzieren

`trajectoryValuesFromMatlab1.inc`
`trajectoryValuesFromMatlab2.inc`
⋮
`trajectoryValuesFromMatlab9.inc`
	Diese neun Dateien stehen für jeweils ein Teilchen, und wurden mit dem Matlab Code erzeugt. Jede Datei enthält ein langes Array, in dem die berechneten x-Werte für die Position des zugehörigen Teilchens zu den zum Arrayindex korrespondierenden Zeitpunkten abgelegt sind. Die Arrays haben jeweils 400 Einträge, ein Eintrag für jedes Frame. Die Dateien werden aufgrund ihrer Zeilenanzahl und der Tatsache, dass sie wieder erzeugt werden können, nicht im Anhang angefügt.

Kapitel 7

Videos zum Doppelspaltversuch

7.1 Das Doppelspaltexperiment mit Materieteilchen

Im Jahr 1803 führte Thomas Young das erste Doppelspaltexperiment überhaupt mit sichtbarem Licht durch, um zu zeigen, dass es sich bei Licht um eine Wellenerscheinung handelt und nicht etwa aus Teilchen besteht. Tatsächlich konnte er Interferenzeffekte beobachten, was seine These erheblich unterstützte.

Führt man das Doppelspaltexperiment mit Materieteilchen, beispielsweise Elektronen, durch, so können ebenfalls Interferenzphänomene beobachtet werden. Das erstaunliche dabei ist, dass diese Interferenz sogar dann auftritt, wenn jedes Teilchen *einzeln* auf die Reise durch den Spalt geschickt wird. Einzeln soll dabei heißen, dass ein Teilchen erst auf den Spalt geschossen wird, wenn das vorhergehende Teilchen bereits auf dem Detektionsschirm angekommen ist.

In Abbildung 7.1 ist abgebildet was bei einem solchen Experiment auf dem Detektionsschirm zu beobachten ist. Die Punkte treffen zunächst scheinbar zufällig irgendwo auf den Schirm. Nach und nach wird in den Auftreffpunkten ein Muster erkennbar, das mit steigender Anzahl von registrierten Teilchen immer deutlicher zu sehen ist. Die Dichte der Auftreffpunkte scheint sich einer bestimmten Verteilung anzunähern. Diese Verteilung sieht stark nach einem Interferenzmuster aus, wie es schon Thomas Young bei Licht beobachten konnte.

Während man bei einem Teilchenstrahl noch auf die Idee kommen mag, dass das Muster durch eine Wechselwirkung der Teilchen im Strahl untereinander zustande kommt, ist das im Fall von Teilchen, die in einem großen zeitlichen Abstand nacheinander den Spalt passieren, eher auszuschließen. Womit interferiert das Teilchen also, wenn es allein unterwegs ist? Mit sich selbst?

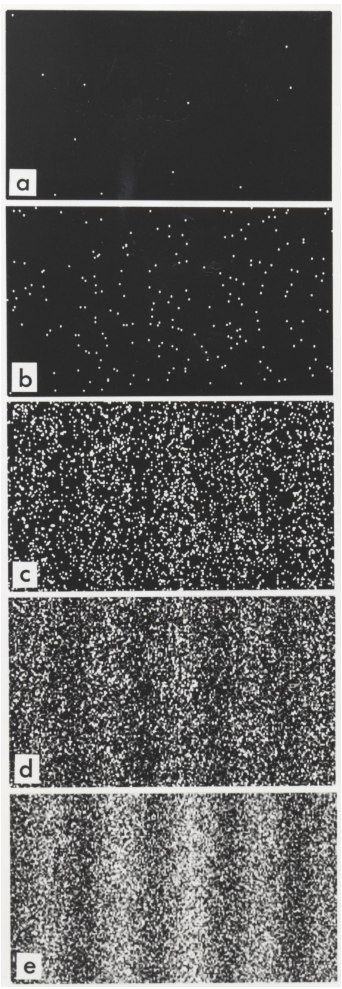


Abbildung 7.1: Auftreffpunkte von Elektronen bei einem Doppelspaltexperiment von A. Tonomura, veröffentlicht 1989 [6]

Erklärung durch die Bohmsche Mechanik

Fest steht, dass das was beim Doppelspaltversuch durch die Anordnung geschossen wird, sowohl Eigenschaften einer Welle als auch eines Teilchens hat. Das entstehende Interferenzmuster lässt sich durch Welleneigenschaften erklären. Auf dem Detektionsschirm werden jedoch einzelne, abzählbare, genau lokalisierte Punkte sichtbar.

Bei dem Versuch diese beiden Eigenschaften miteinander widerspruchsfrei zu vereinbaren, werden verschiedene Wege gegangen. Die Bohmschen Mechanik geht einen besonders einfachen und klaren Weg um die Wellen- und die Teilcheneigenschaft dessen was durch den Spalt geschossen wird zu

beschreiben: Es ist ein Teilchen, das von einer Welle begleitet wird. Damit lösen sich die Mysterien um den bekannten Welle-Teilchen-Dualismus auf:

- Das Teilchen ist ein Teilchen, welches sich zu jedem Zeitpunkt an einem *genauen* Ort befindet.
- Die Welleneigenschaften, die zum Interferenzmuster führen, sind der Wellenfunktion zuzuschreiben, von der das Teilchen geführt wird.

Während das Teilchen immer nur einen der beiden Spalte (es hat ja nur einen Ort zu einem Zeitpunkt) passiert, läuft die Wellenfunktion durch beide Spalte. Hinter den Spalten überlagern sich die beiden Teile, die durch den Doppelspalt getrennt worden sind, und interferieren miteinander. Der scheinbare Zufall der Auftreffpunkte am Schirm lässt sich dadurch erklären, dass sich die auf die Spalte zulaufenden Elektronen in ihren anfänglichen Position in Relation zum jeweiligen Wellenpaket bei jedem Schuss unterscheiden. Die Dichte Auftreffpunkte nähert sich mit steigender Teilchenzahl dem Interferenzmuster der Wellenfunktion an, weil die Startpositionen der Teilchen gemäß des Betragsquadrates der Wellenfunktion in ihrem Wellenpaket verteilt sind. Die Führungsgleichung hat die Eigenschaft, das die derartige Verteilung auch zum Zeitpunkt der jeweiligen Registrierung am Schirm noch erhalten ist.

7.2 Modellierung des Doppelspaltversuchs

Um die genaue Evolution der Wellenfunktion beim Auftreffen auf einen Doppelspalt darzustellen, müsste die Schrödingergleichung für die Anordnung numerisch gelöst werden. Um einer Wahrscheinlichkeitsverteilung der Auftreffpunkte an einem weit genug hinter den Spalten befindlichen Detektionsschirm, wie sie im Experiment beobachtet wird, sehr nahe zu kommen, reicht es jedoch schon aus, zwei Gaußsche Wellenpakete anzunehmen, die von den Spalten ausgehen und sich in Richtung Schirm bewegen. Auf dem Weg zum Schirm werden die Wahrscheinlichkeitsverteilungen breiter und die *komplexen* Wahrscheinlichkeitsamplituden der beiden Gaußpakete interferieren miteinander. An den Punkten, wo beide Pakete die gleiche Phase haben, zeigen die Wahrscheinlichkeitsamplituden in die gleiche Richtung und es kommt zu maximal konstruktiver Interferenz. Konstruktive Interferenz resultiert in einer hohen Wahrscheinlichkeitsdichte. An anderen Punkten kommt es zu maximal destruktiver Interferenz, weil die Wahrscheinlichkeitsamplituden dort um 180° phasenverschoben aufeinandertreffen. Dazwischen liegen Bereiche, in denen die Phasenverschiebung zwischen 0° und 180° liegt.

7.2.1 Gaußsche Wellenpakete

In diesem Abschnitt geht es um die genaue Formulierung der Gaußschen Wellenpakete, sodass die dann für eine Visualisierung des Vorgangs verwendet werden können. Die folgende Herleitung für den Ausdruck eines Gaußschen Wellenpakets ist stark an Kapitel 9 des Quantenmechanik Lehrbuchs von T. Fließbach [3] angelehnt.

Aus der Quantenmechanik ist bekannt, dass sich die Wellenfunktion für ein freies Teilchen [1] durch Überlagerung von elementaren Lösungen der Schrödingergleichung darstellen lässt. Durch Einsetzen in die Schrödingergleichung lässt sich verifizieren, dass

$$\psi_{\mathrm{k}}(x,t) = a\mathrm{e}^{\mathrm{i}(kx-\omega(k)t)} \tag{7.1}$$

mit der konstanten Amplitude a, einer beliebigen reellen Wellenzahl k und der davon abhängigen Frequenz $\omega(k) = \frac{\hbar k^2}{2m}$ eine Lösung ist. Diese Lösung entspräche jedoch einem Teilchen mit exakt bekanntem Impuls und maximal unscharfem Ort und ist nicht normierbar, denn

$$|\psi(x,t)|^2 = |a|^2 = const. \tag{7.2}$$

Um normierbare Lösungen zu erhalten, müssen wir solche Elementarlösungen verschiedener Wellenlängen und Amplituden miteinander addieren:

$$\Psi(x,t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(k)\mathrm{e}^{\mathrm{i}(kx-\omega(k)t}\mathrm{d}k. \tag{7.3}$$

[1] Im Sinne der Bohmschen Mechanik heißt das lediglich, dass kein Potential vorhanden ist. Das Teilchen ist nicht völlig frei, weil es stets von der Wellenfunktion geführt wird.

So erhalten wir für eine sinnvoll gewählte Amplitudenfunktion $A(k)$ eine normierbare Lösung der eindimensionalen freien Schrödingergleichung. Solch eine Lösung nennt man ein Wellenpaket. Doch was ist nun eine sinnvolle Amplitudenfunktion? Für den Zeitpunkt $t = 0$ vereinfacht sich Gleichung 7.3 zu

$$\Psi(x,0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(k) \mathrm{e}^{\mathrm{i}kx} \mathrm{d}k. \tag{7.4}$$

Hierin erkennen wir, dass $A(k)$ die Fouriertransformierte von $\Psi(x,0)$ ist. Um für ein gewünschtes $\Psi(x,0)$ die benötigte Amplitudenfunktion $A(k)$ zu finden, haben wir also

$$A(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \Psi(x,0) \mathrm{e}^{-\mathrm{i}kx} \mathrm{d}x. \tag{7.5}$$

In unserem Fall wollen wir für die Aufenthaltswahrscheinlichkeit $|\Psi(x,0)|^2$ des Teilchens zu Beginn der Doppelspaltsimulation eine Gaußsche Normalverteilung erhalten. Es wird sich gleich herausstellen, dass man solch eine Wellenfunktion bekommt, wenn man auch für die Amplitudenfunktion $A(k)$ eine Gaußfunktion wählt. Motivation für diesen Ansatz sind zwei bekannte Eigenschaften der Gaußfunktion:

- Die Fouriertransformierte einer zentrierten Gaußfunktion ist selbst eine Gaußfunktion.
- Das Quadrat einer Gaußfunktion ist selbst eine Gaußfunktion.

Wählen wir für die Verteilung $A(k)$ der Wellenzahlen, aus denen das Paket besteht, eine um k=0 zentrierte Gaußfunktion, so erhalten wir für $\Psi(x,0)$ ebenfalls eine Gaußfunktion. Wegen Punkt 2 ist dann auch die Aufenthaltswahrscheinlichkeit $|\Psi(x,0)|^2$ eine Gaußfunktion. Mit der Konstanten C in der symmetrischen Amplitudenfunktion

$$A_{sym}(k) = C\mathrm{e}^{-\alpha k^2} \tag{7.6}$$

erhalten wir durch inverse Fouriertransformation

$$\Psi_{sym}(x,0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A_{sym}(k) \mathrm{e}^{\mathrm{i}kx} \mathrm{d}k = \frac{C}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathrm{e}^{-\alpha k^2} \mathrm{e}^{\mathrm{i}kx} \mathrm{d}k = \frac{C}{\sqrt{2\alpha}} \mathrm{e}^{-\frac{x^2}{4\alpha}}. \tag{7.7}$$

Die Annahme, dass $A(k)$ symmetrisch um $k = 0$ ist, entspricht jedoch wie wir sehen werden einem Wellenpaket dessen Schwerpunkt (x-Wert des Maximums der Wahrscheinlichkeitsdichte) zeitlich still steht, und ist daher eine starke Einschränkung der Allgemeinheit.

Um ein allgemeineres Wellenpaket zu erhalten, dessen Schwerpunkt sich mit der Zeit in x-Richtung bewegt, muss mit einer Amplitudenverteilung $A(k)$ angesetzt werden, die nicht symmetrisch um $k = 0$ ist, sondern um einen positiven Wert k_0 verschoben ist. Diese allgemeinere Amplitudenfunktion lautet

$$A(k) = C\mathrm{e}^{-\alpha(k-k_0)^2}. \tag{7.8}$$

Eine Regel zur Fouriertransformation, die beispielsweise [1] zu entnehmen ist und dort als Dämpfungssatz bekannt ist, besagt, dass in diesem Fall

$$\mathcal{F}\{\mathrm{e}^{\mathrm{i}k_0 x} \Psi_{sym}(x,0)\} = A(k - k_0) \tag{7.9}$$

Das resultierende Wellenpaket muss also nur den zusätzlichen Faktor $\mathrm{e}^{\mathrm{i}k_0 x}$ bekommen und sieht ansonsten aus wie $\Psi_{sym}(x,0)$:

$$\Psi(x,0) = \mathrm{e}^{\mathrm{i}k_0 x}\, \Psi_{sym}(x,0) = \frac{C}{\sqrt{2\alpha}} \mathrm{e}^{\mathrm{i}k_0 x} \mathrm{e}^{-\frac{x^2}{4\alpha}} \tag{7.10}$$

Diese Funktion ist aufgrund des hinzugefügten komplexen Faktors[2] keine Gaußfunktion. Ihr Betragsquadrat ist es jedoch! Deswegen und weil die Verteilung der Wellenzahlen $A(k)$ eine Gaußfunktion ist, wird auch eine solche komplexwertige Wellenfunktion als Gaußsches Wellenpaket bezeichnet.

[2]Genau dieser komplexe Faktor ist dafür verantwortlich, dass später so interessante Interferenzmuster zustande kommen werden.

Zeitabhängigkeit

Wie verändert sich solch ein Wellenpaket nun mit der Zeit? Dafür kehren wir zu Gleichung 7.3 zurück:

$$\Psi(x,t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(k) \mathrm{e}^{\mathrm{i}(kx-\omega(k)t} \mathrm{d}k. \tag{7.11}$$

Der Zusammenhang zwischen der Kreisfrequenz ω einer elementaren Lösung und der zugehörigen Wellenzahl k ist durch die Dispersionsrelation für die Schrödingergleichung gegeben:

$$\omega(k) = \frac{\hbar k^2}{2m} \tag{7.12}$$

Taylorreihe von $\omega(k)$ um k_0:

$$\omega(k) = \omega(k_0) + \left.\frac{\mathrm{d}\omega}{\mathrm{d}k}\right|_{k_0} (k-k_0) + \frac{1}{2} \left.\frac{\mathrm{d}^2\omega}{\mathrm{d}k^2}\right|_{k_0} (k-k_0)^2 = \underbrace{\omega(k_0)}_{=:\,\omega_0} + \underbrace{\frac{\hbar k_0}{m}}_{=:\,v_g}(k-k_0) + \underbrace{\frac{\hbar}{2m}}_{=:\,\beta}(k-k_0)^2 \tag{7.13}$$

Die neuen Bezeichner unter den geschweiften Klammern, dienen vorerst nur der einfacheren Schreibweise. Mit der Entwicklung von $\omega(k)$ und der Amplitudenfunktion $A(k)$ für das sich bewegende Wellenpaket lautet das zu lösende Integral:

$$\Psi(x,t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(k) \mathrm{e}^{\mathrm{i}(kx-\omega(k)t} \mathrm{d}k = \frac{C}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \mathrm{e}^{-\alpha(k-k_0)^2} \mathrm{e}^{\mathrm{i}kx} \mathrm{e}^{-\mathrm{i}(\omega_0 + v_g(k-k_0) + \beta(k-k_0)^2)t} \mathrm{d}k. \tag{7.14}$$

Fließbach, Quantenmechanik [3, p. 65] entnehmen wir die Lösung dieses Integrals:

$$\boxed{\Psi(x,t) = \frac{C}{\sqrt{2}} \frac{\mathrm{e}^{i(k_0 x - \omega_0 t)}}{\sqrt{\alpha + \mathrm{i}\beta \mathrm{t}}} \exp\left(-\frac{(x-v_g t)^2}{4(\alpha + \mathrm{i}\beta t)}\right)} \tag{7.15}$$

Für die Aufenthaltswahrscheinlichkeit des Teilchens ergibt sich damit die Gaußfunktion

$$\boxed{|\Psi(x,t)|^2 = \frac{|C|^2}{2\sqrt{\alpha^2 + \beta^2 t^2}} \exp\left(-\frac{\alpha(x-v_g t)^2}{2(\alpha^2+\beta^2 t^2)}\right).} \tag{7.16}$$

Durch einen Vergleich mit der Standardform einer Gaußschen Normalverteilung

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{7.17}$$

können wir Teile der erhaltenen Wahrscheinlichkeitsverteilung mit der Standardabweichung σ und dem Erwartungswert μ identifizieren. Durch Betrachtung der quadrierten Klammer lässt sich μ mit $v_g t$ identifizieren. Aus

$$\frac{\alpha}{2(\alpha^2+\beta^2 t^2)} = \frac{1}{2\sigma^2} \tag{7.18}$$

folgt, dass $\sigma = \sigma(t) = \sqrt{\frac{\alpha^2+\beta^2 t^2}{\alpha}}$ ist. Die Gaußkurve für $|\Psi(x,t)|^2$ wird also breiter mit der Zeit t und verschiebt sich mit der Geschwindigkeit v_g. Man nennt v_g deshalb (in Abgrenzung zur Phasengeschwindigkeit) die *Gruppengeschwindigkeit* des Wellenpakets.

De-Broglie-Wellenlänge des Wellenpakets

Lässt sich diesem Wellenpaket nun eine Wellenlänge zuordnen? Das Paket besteht ist eine Überlagerung von elementaren Wellen mit verschiedenen Wellenzahlen k. Am stärkster Vertreten in der Überlagerung ist eine Welle mit der Wellenzahl k_0, denn hier hat die Verteilung $A(k)$ ein Maximum. Dieser Wellenzahl k_0 entspricht die Wellenlänge $\lambda_0 = \frac{2\pi}{k_0}$. Louis de Broglie hat 1924, (bevor die Schrödingergleichung veröffentlicht war) vorgeschlagen, jedem Materieteilchen die Wellenlänge $\lambda_{dB} = \frac{h}{p}$ zuzuordnen. Nehmen wir für den Impuls des Teilchens $p = v_g m$ an, so ergibt sich mit $v_g = \frac{\hbar k_0}{m}$ nach de Broglie die Wellenlänge:

$$\lambda_{dB} = \frac{h}{p} = \frac{2\pi\hbar}{v_g m} = \frac{2\pi}{k_0} = \lambda_0 \tag{7.19}$$

7.2.2 Zweidimensionaler Fall

Im zweidimensionalen Fall lautet die Schrödingergleichung für ein freies Teilchen

$$-\frac{\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\Psi(x,y,t)=\mathrm{i}\hbar\,\frac{\partial\Psi(x,y,t)}{\partial t}. \tag{7.20}$$

Die elementaren Lösungen lauten dafür

$$\psi(x,y,t)=a\mathrm{e}^{i(k_x x+k_y y-\omega(k_x,k_y)t)}. \tag{7.21}$$

Die allgemeine Lösung ist wieder eine Überlagerung der elementaren Lösungen. Diesmal hängt die Amplitudenfunktion von k_x und k_y ab und es muss über alle Kombinationen der beiden Wellenzahlen integriert werden:

$$\Psi(x,y,t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}A(k_x,k_y)\mathrm{e}^{i(k_x x+k_y y-\omega(k_x,k_y)t)}\mathrm{d}k_x\mathrm{d}k_y \tag{7.22}$$

Die Amplitudenfunktion $A(k_x,k_y)$ soll nun ein Produkt aus zwei Gaußfunktionen sein und kann deshalb separiert werden in zwei Faktoren, die jeweils nur von einer der beiden Wellenzahlen abhängen

$$A(k_x,k_y)=\mathrm{e}^{\alpha_x(k_x-k_{x0})^2}\;\mathrm{e}^{\alpha_y(k_y-k_{y0})^2}=:A_x(k_x)\;A_y(k_y) \tag{7.23}$$

In der Dispersionsrelation für den zweidimensionalen Fall hängt auch ω von beiden Wellenzahlen ab und kann in zwei Summanden separiert werden:

$$\omega(k_x,k_y)=\frac{\hbar}{2m}(k_x^2+k_y^2)=\frac{\hbar k_x^2}{2m}+\frac{\hbar k_y^2}{2m}=:\omega_x(k_x)+\omega_y(k_y) \tag{7.24}$$

Damit lässt sich die komplette Wellenfunktion in ein Produkt zerlegen:

$$\Psi(x,y,t)=\underbrace{\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}A_x(k_x)\mathrm{e}^{i(k_x x-\omega_x(k_x)t)}\mathrm{d}k_x}_{=:\Psi_x(x,t)}\;\underbrace{\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}A_y(k_y)\mathrm{e}^{i(k_y y-\omega_y(k_y)t)}\mathrm{d}k_y}_{=:\Psi_y(y,t)} \tag{7.25}$$

Auf die eindimensionalen Wellenpakete $\Psi_x(x,t)$ und $\Psi_y(y,t)$ lässt sich alles aus dem letzten Abschnitt übertragen, insbesondere die resultierende Gleichung 7.15.

7.2.3 Räumliche Anordnung und Parameter der Wellenpakete

In unserem Modell vom Doppelspaltversuch gibt es zwei Gaußsche Wellenpakete, die jeweils in ihrem Spalt zentriert starten und sich von dort aus von dort aus parallel fortbewegen.

Jetzt soll die im letzten Abschnitt erklärte Möglichkeit, die Wellenfunktion in ein Produkt $\Psi(x,y,t)=\Psi_x(x,t)\;\Psi_y(y,t)$ zu separieren, genutzt werden. Und erinnern uns an Gleichung 7.15 für das eindimensionale Gaußsche Wellenpaket:

$$\Psi(x,t)=\frac{C}{\sqrt{\pi}}\frac{\mathrm{e}^{i(k_0x-\omega_0t)}}{\sqrt{\alpha+\mathrm{i}\beta\mathrm{t}}}\exp\left(-\frac{(x-v_gt)^2}{4(\alpha+\mathrm{i}\beta t)}\right) \tag{7.26}$$

Alle Normierungsfaktoren sollen in der weiteren Darstellung ignoriert werden, weil sie für die später resultierenden Trajektorien keine Rolle spielen.

Wir wählen das Koordinatensystem so, dass sich die Pakete ausschließlich in die x-Richtung bewegen. Der y-Teil $\Psi_y(y,t)$ hat dann die Gruppengeschwindigkeit $v_{gy}=\frac{\hbar k_{0y}}{m}=0$. Der y-Teil der Wellenfunktion besteht wohlgemerkt aus *zwei* Gaußschen Wellenpaketen, deren Schwerpunkte um den Spaltabstand D voneinander versetzt sind.

$$\Psi_y(y,t)=\frac{1}{\sqrt{\alpha_y+\mathrm{i}\beta_\mathrm{y}\mathrm{t}}}\exp\left(-\frac{(y-D/2)^2}{4(\alpha_y+\mathrm{i}\beta_y t)}\right)+\frac{1}{\sqrt{\alpha_y+\mathrm{i}\beta_\mathrm{y}\mathrm{t}}}\exp\left(-\frac{(y+D/2)^2}{4(\alpha_y+\mathrm{i}\beta_y t)}\right) \tag{7.27}$$

Der x-Teil $\Psi_x(x,t)$ soll eine positive Gruppengeschwindigkeit $v_{gx}=\frac{\hbar k_{0x}}{m}>0$ haben und bei $x=0$ zentriert starten. Weiterhin soll er nur ein Maximum im Betragsquadrat besitzen:

$$\Psi_x(x,t)=\frac{1}{\sqrt{\alpha_x+\mathrm{i}\beta_\mathrm{x}\mathrm{t}}}\exp\left(-\frac{(x-v_{gx}t)^2}{4(\alpha_x+\mathrm{i}\beta_x t)}\right) \tag{7.28}$$

Die Wellenfunktion für das Doppelspaltmodell ist dann gegeben durch

$$\Psi(x, y, t) = \Psi_x(x, t)\ \Psi_y(y, t). \tag{7.29}$$

Nun sind die zahlreichen offenen Parameter dieser Gleichungen festzulegen. Dabei sind folgende Fragen vor dem Hintergrund von Gleichung 7.16 zu stellen:

- Wie weit sollen die Spalte voneinander entfernt sein? Je größer der Spaltabstand D ist, desto länger dauert es bis die beiden Pakete ineinander fließen und desto enger liegen Interferenzmaxima aneinander.
- Wie lang sollen die Pakete zu Beginn sein? Genauer: Welche Standardabweichung $\sigma_x = \alpha_x^2$ soll die Verteilung $|\Psi_x(x, t = 0)|^2$ haben?
- Wie breit sollen die Pakete bei $t = 0$ in y-Richtung sein? Genauer: Welche Standardabweichung $\sigma_y = \alpha_y^2$ sollen die beiden Summanden von $|\Psi_y(y, t = 0)|^2$ haben?
- Mit welcher Gruppengeschwindigkeit v_{gx} sollen sich die Pakete in x-Richtung bewegen? Auch diese Größe hat Einfluss auf den Abstand der Interferenzmaxima voneinander. Das ist dadurch zu erklären, dass $\lambda_0 = \frac{2\pi\hbar}{v_g m}$ gilt.
- Welche Masse m soll das Teilchen haben? Davon hängt unter anderem ab wie schnell sich die Pakete verbreitern (denn $\beta = \frac{\hbar}{2m}$) und weit die Interferenzmaxima voneinander entfernt sein werden.

In einer Reihe von Tests haben sich folgende Werte als zufriedenstellend für die erwünschte Darstellung erwiesen:

Größe	Bezeichner in MATLAB Code	Wert
D	`slitDistance`	4
σ_y	`sigmaY`	0.2
σ_x	`sigmaX`	1
$v_{G,x}$	`groupVelocityX`	5
$v_{G,y}$	`groupVelocityY`	0
m	`mass`	2

Darüber hinaus wurde für das Modell das reduzierte Plancksche Wirkungsquantum $\hbar$ auf den Wert 1 gesetzt. In der folgenden Abbildung ist die Wahrscheinlichkeitsverteilung $|\Psi(x, y, 0)|^2$ für die gewählten Parameter (zusammen mit einer Wand mit zwei Spalten) dargestellt:

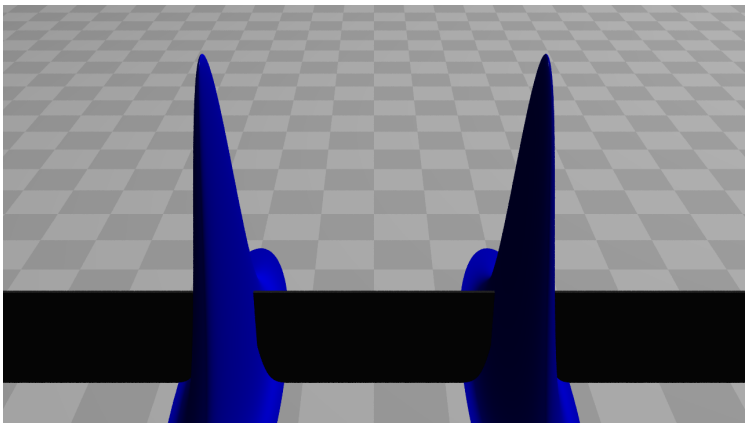


Abbildung 7.2: $|\Psi(x, y, 0)|^2$ zum Zeitpunkt $t = 0$. Dort, wo der Funktionswert unter einem bestimmten Schwellenwert liegt, ist sie nicht dargestellt, sodass auch der Untergrund zu sehen ist. Die Breite eines Kästchens des Schachbettmusters beträgt eine Längeneinheit.

7.3 Idee zur Visualisierung mit POV-Ray

Es sollen nun Videos erzeugt werden, in denen der physikalische Ablauf der modellierten Situation visualisiert wird. Es ist dabei nicht so leicht, wie es etwa beim unendlich tiefen Potentialtopf war, die komplexe Wellenfunktion darzustellen. Während der physikalische Raum

in dem sich das Geschehen abspielt, beim Potentialtopf eindimensional war, ist er beim Doppelspaltmodell zweidimensional. Uns stehen also im Darstellungsraum nicht mehr zwei weitere Dimensionen zur Verfügung, um Real- und Imaginärteil der Wellenfunktion so einfach und deutlich darzustellen. Daher wurde entschieden vorerst nur das Betragsquadrat der Wellenfunktion in den Videos abzubilden. Zusätzlich zu dieser Repräsentation der Wellenfunktion, sollen die möglichen Bohmschen Trajektorien eines Teilchens das von der Wellenfunktion geführt wird, zeitaufgelöst dargestellt werden.

7.3.1 Visualisierung der Trajektorien

Die Teilchen sollen durch schwarze Kugeln dargestellt werden. Es ist wichtig, nicht zu vergessen, dass die Gaußschen Wellenpakete in unserem Modell nur zu *einem* Teilchen gehören. Trotzdem kann es sinnvoll sein, viele *mögliche* Trajektorien *repräsentativ* in einem Video gemeinsam darzustellen. Dadurch wird klarer welche Charakteristiken die möglichen Trajektorien gemeinsam haben. Um diese Charakteristiken noch deutlicher zu machen, sollen die schwarzen Kugeln dünne schwarze Linien hinter sich lassen. So wird auch in einem jedem einzelnen Frame zu sehen sein, welche Route die Teilchen genommen haben.

Die Teilchen sollen sich zum Zeitpunkt $t = 0$ alle auf der y-Achse befinden, und dort in ihren y-Ordinaten regelmäßig gemäß $|\Psi(x = 0, y, t = 0)|^2$ verteilt sein. Diese Anordnung der Teilchen macht charakeristische Eigenschaften der Trajektorien besonders deutlich ist jedoch *keine* physikalische Notwendigkeit. Es wäre völlig legitim sich hier anders zu entscheiden! Hier besteht eine der zahlreichen Möglichkeiten den geschriebenen Code zu modifizieren und weitere interessante Videos zu erstellen. Dazu mehr in Abschnitt 8.

Um jedoch das Missverständnis zu vermeiden (oder um es *auszuräumen*, falls es beim Betrachter schon besteht), dass nur dann ein Interferenzmuster entstünde, wenn viele Teilchen gleichzeitig die Spalte passieren und *miteinander* interferieren, soll auch ein Video erstellt werden, in dem nur ein einziges Teilchen die Anordnung durchläuft.

7.3.2 Visualisierung der Wahrscheinlichkeitsdichte

Die Wahrscheinlichkeitsdichte wurde auf zwei verschiedene Arten in den verschiedenen Videos dargestellt:

- **Als Fläche:**
  Die Wahrscheinlichkeitsdichte $|\Psi(x, y, t)|^2$ kann als Fläche dargestellt werden, die über der x-y-Ebene schwebt. Präzise formuliert: Es soll die Fläche gezeichnet werden, die aus den Punkten $\{(x, y, z)|(x, y) \in \mathbb{R}^2,\ z = |\Psi(x, y, t)|^2\}$ besteht. Das kann in POV-Ray mit einem sogenannten `isosurface`-Objekt realisiert werden.

- **Als Raumkurve:**
  Die Wahrscheinlichkeitsdichte wird als schwebende Linie dargestellt, die sich immer über den Teilchen befindet. In diesem Fall wird die Wahrscheinlichkeitsdichte nur für die Punkte dargestellt, die sich auf der mit den Teilchen mitlaufenden Geraden $\{(x, y)|x(t) = v_{gx}t\}$ befinden. Diese Methode hat den Nachteil, dass verglichen mit der Flächendarstellung Information verloren geht. Der Vorteil liegt jedoch darin, dass die Teilchen und die dünnen Linien, die sie hinter sich lassen, nicht verdeckt sind. (Eine andere Möglichkeit wäre, $|\Psi|^2$ durch eine leicht transparente Fläche darzustellen.

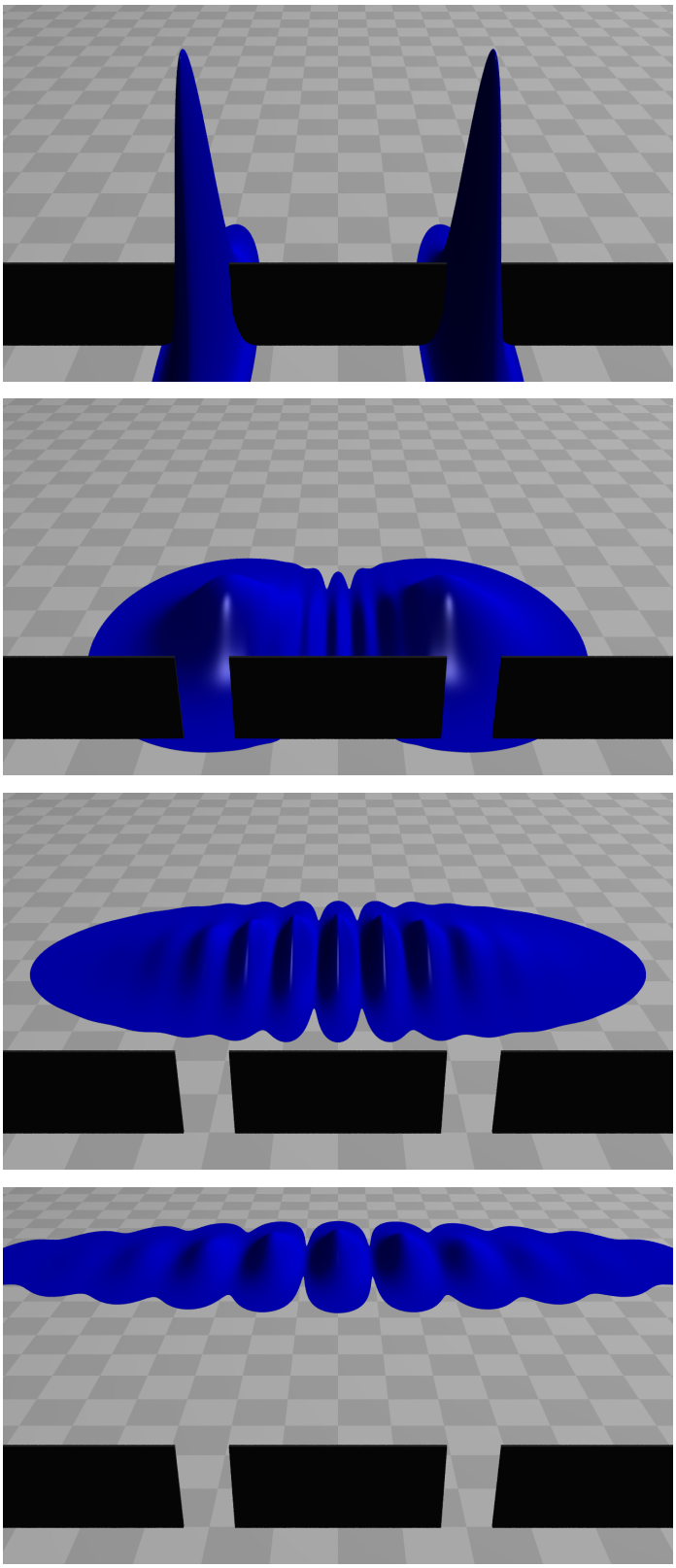
7.3.3 Zufällige Auftreffpunkte

Es soll ein Weiteres Video erstellt werden, in dem nach und nach immer mehr Auftreffpunkt auf einem Detektionsschirm erscheinen. Wie in Abbildung 7.1 zum Experiment von Tonomura soll sich die Dichte der Auftreffpunkte mit der Zeit der Wahrscheinlichkeitsverteilung $|\Psi|^2$ annähern. In diesem Video werden keine Bohmschen Trajektorien abgebildet.

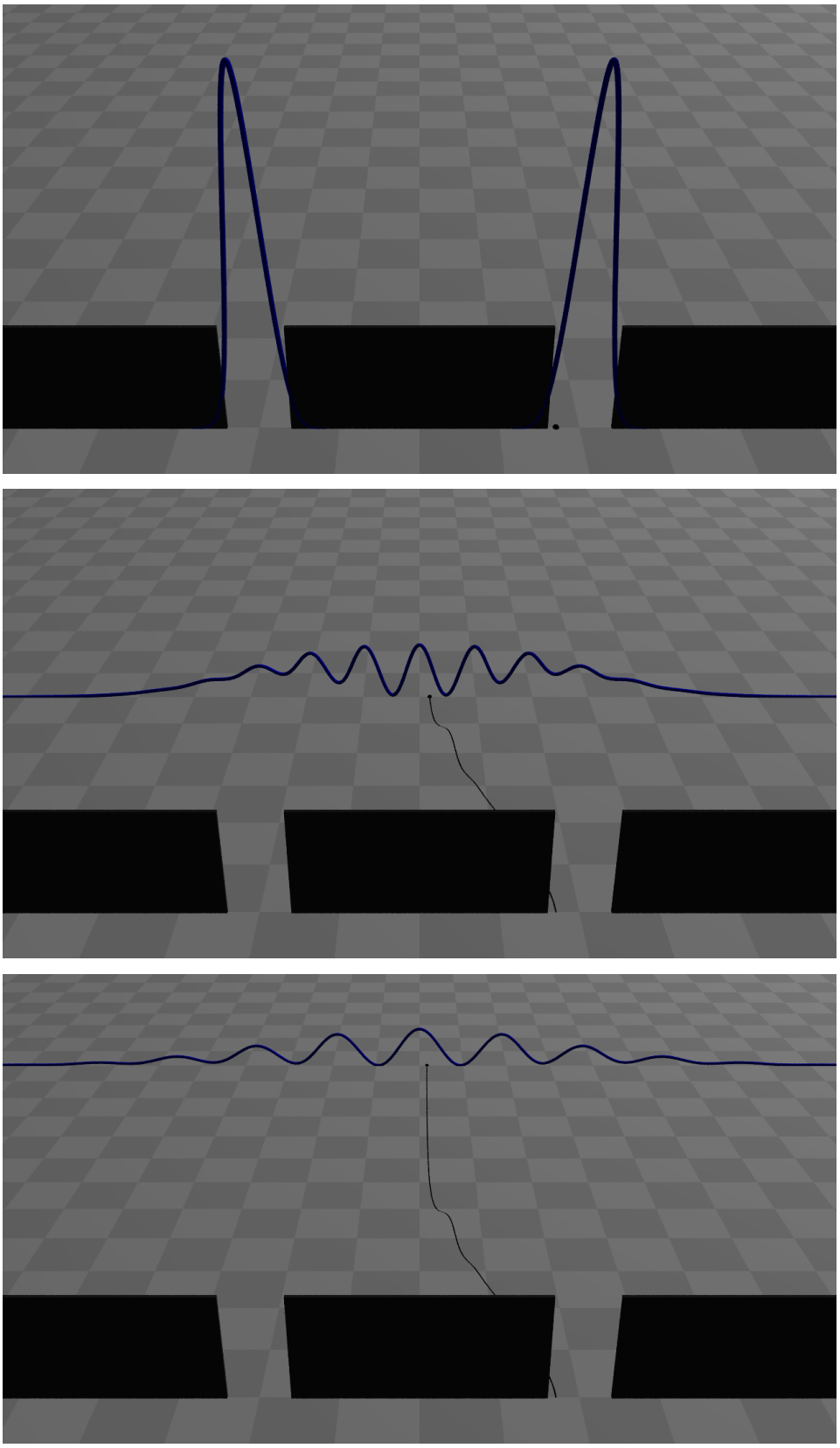
7.4 Frames der erstellten Animationen

Auf den nächsten Seiten sind einzelne Frames der erstellten Animationen abgebildet.

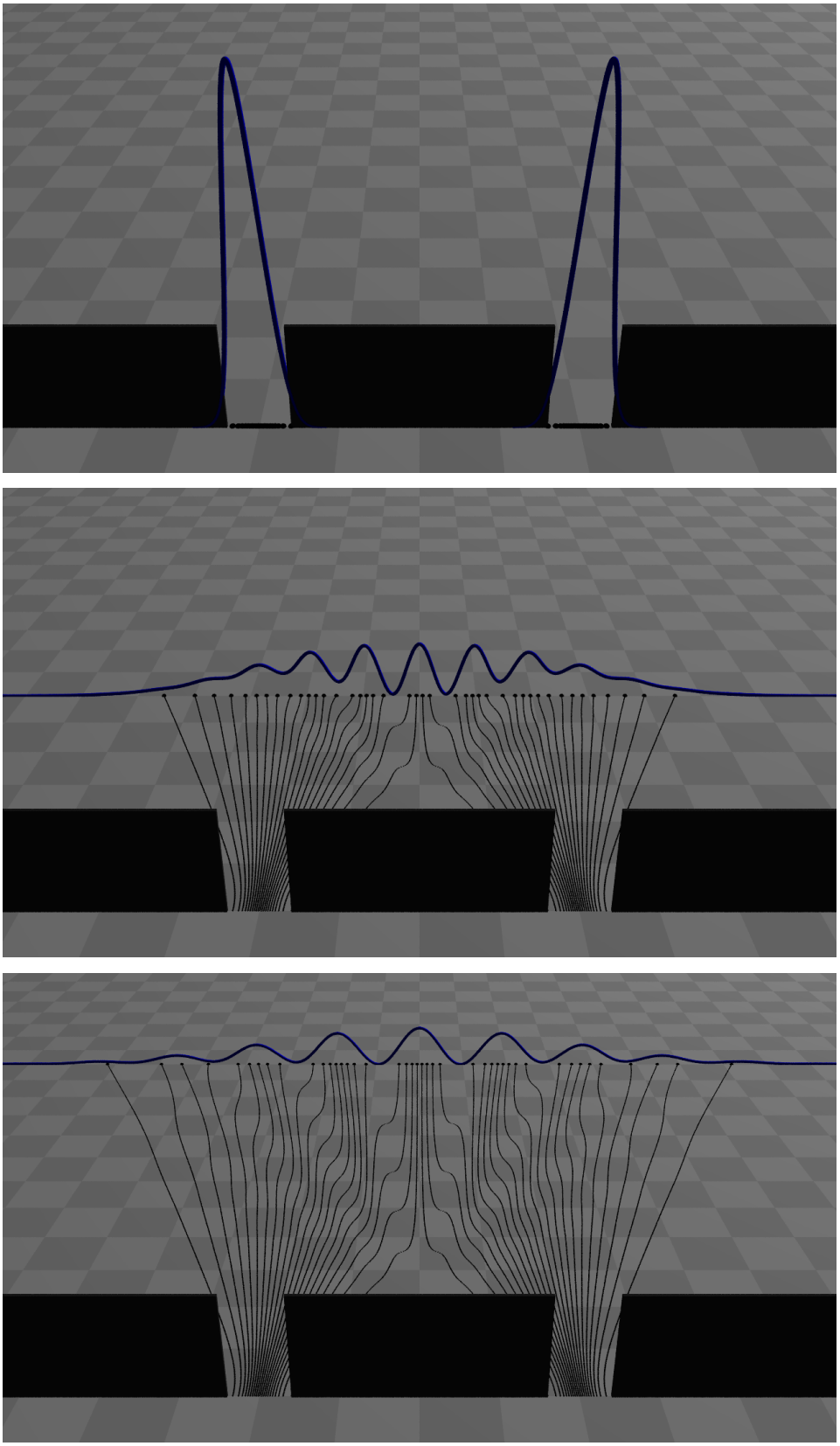
7.4.1 Wahrscheinlichkeitsverteilung als Fläche



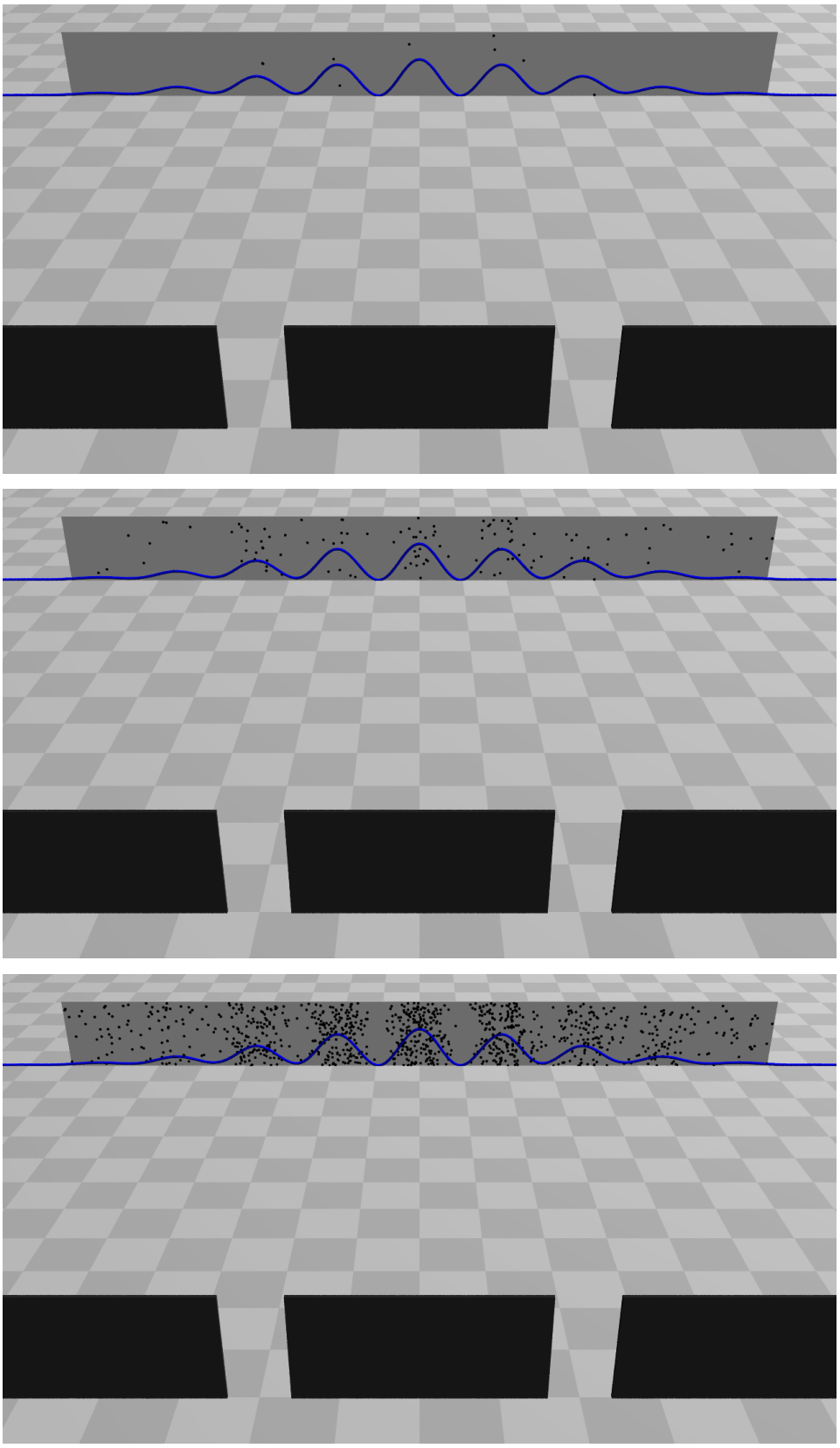
7.4.2 Wahrscheinlichkeitsverteilung als Linie und ein Teilchen



7.4.3 Wahrscheinlichkeitsverteilung als Linie und viele mögliche Trajektorien



7.4.4 Auftreffpunkte auf einem Detektionsschirm



7.5 Was ist in den Videos zu beobachten?

Die beiden spitzen Gaußverteilungen werden schnell flacher und breiter. Die (nicht unmittelbar sichtbaren) Wahrscheinlichkeitsamplituden im Überlappungsbereich werden größer und so entstehen durch Interferenz Täler und Hügel in der blauen Fläche für die Wahrscheinlichkeitsverteilung. Auf der Symmetrieachse befindet sich jetzt stets ein lokaler Hügel, weil die Wahrscheinlichkeitsamplituden der beiden Teilwellenfunktionen dort in gleicher Phase aufeinandertreffen und somit konstruktiv interferieren. Durch die fortschreitende Verbreiterung der Teilwellenfunktionen bildet sich der Hügel auf der Symmetrieachse zu einem globalen Maximum heraus.

Wie schon beim Potentialtopf bestätigen sich bekannte Eigenschaften der Bohmschen Trajektorien:

- Die Trajektorien schneiden sich nicht.
- Die Symmetrieachse wird von den Teilchen nicht überschritten. Dass das zwangsläufig nicht passieren kann, zeigt die folgende Überlegung: Würde ein Teilchen A, das am Punkt (x_0, y_0) startet von links die Symmetrieachse überschreiten, so würde ein hypothetisches Teilchen B, das am Punkt $(x_0, -y_0)$ startet zum gleichen Zeitpunkt die Symmetrieachse überschreiten. Das ist unmöglich weil die Führungsgleichung eine Bewegungsgleichung erster Ordnung ist!

  Die Frage welchen Spalt ein Teilchen passiert hat, ist mit damit in der Bohmschen Mechanik ganz einfach zu beantworten: Trifft das Teilchen links der Symmetrieachse des Aufbaus auf den Detektionsschirm, so ist es durch den linken Spalt geflogen, ansonsten durch den rechten.
- Wenn die repräsentativen Teilchen am Anfang gemäß $|\Psi(t)|^2$ verteilt sind, dann bleiben sie es auch über den gesamten Verlauf der Zeit. Mann kann sich vorstellen, dass sich unter der Fläche für $|\Psi(t)|^2$ eine inkompressible Flüssigkeit befindet. Gemäß der Änderung der Form der Fläche für $|\Psi(t)|^2$ wird die Flüssigkeit an manchen Stellen hin zu Symmetrieachse, an anderen Stellen von der Symmetrieachse weg verdrängt. Die Teilchen werden von dieser Flüssigkeit mitgezogen, sodass sich die anfängliche Teilchendichte über die Zeit in gleicher Weise verändert wie $|\Psi(t)|^2$. Diese Verbildlichung erinnert an eine Kontinuitätsgleichung der Form

  $$\frac{\partial}{\partial t}\rho + \nabla \cdot \mathbf{j} = 0. \tag{7.30}$$

  In der Tat gibt es in der Quantenmechanik solch eine Kontinuitätsgleichung für die Wahrscheinlichkeitsdichte $\rho = |\Psi|^2$. Aus der Schrödingergleichung kann dann für die Wahrscheinlichkeits*strom*dichte $\mathbf{j}$ der Ausdruck

  $$\mathbf{j} = \frac{\mathrm{i}\hbar}{2m}[\psi\nabla\psi^* - \psi^*\nabla\psi] = \frac{\hbar}{m}\,\mathrm{Im}\,(\psi^*\nabla\psi) \tag{7.31}$$

  hergeleitet werden. Damit kann die Führungsgleichung der Bohmschen Mechanik in die ausdrucksstarke Form

  $$\frac{\mathrm{d}\mathbf{Q}}{\mathrm{d}t} = \frac{\mathbf{j}}{\rho} \tag{7.32}$$

  gebracht werden.
- Unter den Minima der blauen Raumkurve bewegen sich die Teilchen sehr schnell hindurch. Dort ist der Quotient $\mathbf{j}/\rho$ besonders groß.

7.6 Implementierung des Doppelspaltmodells

Für die Implementierung wurde, wie schon beim unendlich tiefen Potentialtopf, MATLAB für die Berechnung der Trajektorien verwendet und POV-Ray für die visuelle Darstellung des zeitlichen Ablaufs. Mit den folgenden Tabellen soll ein Überblick über den geschriebenen Quellcode vermittelt werden.

7.6.1 Überblick MATLAB Quellcode

Datei	Erklärung
`doubleSlit.m`	In dieser Datei sind sämtliche globale Variablen abgelegt. Einerseits wird hier die physikalische Ausgangssituation mit den Werten aus Tabelle7.2.3 festgelegt. Zum anderen werden Parameter festgelegt, die das Video und die auszugebenden Dateien für die Teilchentrajektorien betreffen, wie die Gesamtzahl der Frames des Videos und die physikalische Zeitspanne, über die der Ablauf dargestellt werden soll.
`alpha.m`	Enthält eine Funktion `alpha(sigmaStart)`, die für den gewünschten Wert der Standardabweichung σ des Betragsquadrats eines Gaußschen Wellenpakets zum Zeitpunkt $t = 0$ den entsprechenden Wert für α zurückgibt
`waveNumber.m`	Enthält eine Funktion `waveNumber(groupVelocity)`, die für eine Gruppengeschwindigkeit v_g die im Paket am stärksten vorhandene Wellenzahl $k_0 = \frac{v_g m}{\hbar}$ zurückgibt.
`omega.m`	Enthält eine Funktion `omega(waveNumber)`, die für die Wellenzahl k_0 den über die Dispersionsrelation bestimmten Wert $\omega_0 = \frac{\hbar k_0^2}{2m}$ zurückgibt
`gaussPacket.m`	Enthält eine Funktion `gaussPacket(position,t,initialPosition, sigmaStart,groupVelocity)`, die den komplexen Wert eines eindimensionales Gaußschen Wellenpakets zu den angegebenen Parametern gemäß Gleichung 7.15 zurückgibt
`psiX.m`	Enthält eine Funktion `psiX(x,t)`, die einer Implementierung von Gleichung 7.28 entspricht und den komplexen Wert $\Psi_x(x,t)$ des x-Faktors der Wellenfunktion $\Psi(x,y,t) = \Psi_x(x,t)\,\Psi_y(y,t)$ zurückgibt. Dabei wird intern die Funktion `gaussPacket(x,t,0,sigmaX,groupVelocityX )` mit den Parametern σ_x und v_{gx} aus Tabelle 7.2.3 aufgerufen. Diese Parameter müssen jedoch nicht beim Funktionsaufruf von `PsiX(x,t)` eingegeben werden, weil sie als globale Variablen zur Verfügung stehen.
`psiY.m`	Enthält eine Funktion `psiY(y,t)`, die einer Implementierung von Gleichung 7.27 entspricht und den komplexen Wert des y-Faktors $\Psi_y(y,t)$ der Wellenfunktion $\Psi(x,y,t) = \Psi_x(x,t)\ \Psi_y(y,t)$ zurückgibt.
`waveFunction.m`	Enthält eine Funktion `waveFunction(x,y,t)`, den komplexen Wert der Wellenfunktion $\Psi(x,y,t) = \Psi_x(x,t)\ \Psi_y(y,t)$ zurückgibt.
`waveFunctionDerivativeX.m`	Enthält eine Funktion `waveFunctionDerivativeX(x,y,t,dx)`, die den numerisch berechneten Wert der Ableitung $\frac{\mathrm{d}\Psi(x,y_0,t_0)}{\mathrm{d}x}\|_{x_0}$ an der Stelle (x_0,y_0,t_0) =`(x,y,t)` zurückgibt. Der Parameter `dx` entspricht dem Nenner im Differenzenquotienten und sollte sehr klein gewählt werden.
`waveFunctionDerivativeY.m`	Enthält eine Funktion `waveFunctionDerivativeY(x,y,t,dy)`, die den numerisch berechneten Wert der Ableitung $\frac{\mathrm{d}\Psi(x_0,y,t_0)}{\mathrm{d}y}\|_{y_0}$ an der Stelle (x_0,y_0,t_0) =`(x,y,t)` zurückgibt. Der Parameter `dy` entspricht dem Nenner im Differenzenquotienten und sollte sehr klein gewählt werden.
`probDensity.m`	Enthält eine Funktion `probDensity(x,y,t)`, die das Betragsquadrat $\|\Psi(x,y,t)\|^2$ der Wellenfunktion zurückgibt.
`yStartArrayNormalDist.m`	Enthält eine Funktion `yStartArrayNormalDist(numberOfParticles, centeredAt,yIntegrationStep)`, die ein Array mit den y-Werten für die startenden Teilchen zurückgibt. Diese Anfangswerte sind gemäß $\|\Psi(x=0,y,t=0)\|^2$ verteilt.
`printTrajectoryToFile.m`	Enthält eine Funktion `printTrajectoryToFile(xStart,yStart, fileName,particleID)`, die eine für POV-Ray lesbare Datei mit dem Namen `fileName` erzeugt, welche ein Array für die Teilchenpositionen des Teilchens mit der Nummer `particleID` erzeugt. Jeder Eintrag korrespondiert zu einem Frame.
`printHitsOnScreenToFile.m`	Diese Datei wurde für die Erzeugung eines Videos geschrieben, in dem keine Trajektorien zu sehen sind, sondern ein Detektionsschirm, auf dem nach und nach schwarze Punkte gemäß der Verteilung $\|\Psi(x = x_{sc}, y, t = \frac{x_{sc}}{v_{gx}})\|^2$ erscheinen.

7.6.2 Überblick POV-Ray Quellcode

Szenenbeschreibungsdateien und INI-Dateien

Die folgenden `.pov`-Dateien enthalten die Szenenbeschreibungen für die erstellten Animationen. Damit POV-Ray die Frames rendert, muss der `povray` Befehl auf die zugehörige `.ini`-Datei angewendet werden. Vorher muss der entsprechende MATLAB Code ausgeführt und die ausgegebenen Dateien für die Trajektorien in das richtige Verzeichnis verschoben werden.

Datei	Erklärung
`doubleSlitWaveOnly.pov`	Enthält die Szenenbeschreibung für die Animationen, in denen die Wahrscheinlichkeitsdichte $\|\Psi\|^2$ als Fläche dargestellt ist.
`doubleSlitWaveOnly.ini`	Zugehörige INI-Datei, in der festgelegt wird, wie viele Frames berechnet werden sollen und welche Auflösung sie haben sollen.
`doubleSlitLineOneParticle.pov`	Szenenbeschreibung für die Animation, in der $\|\Psi\|^2$ als Raumkurve und nur eine mögliche Teilchentrajektorie dargestellt ist.
`doubleSlitLineOneParticle.ini`	Zugehörige INI-Datei.
`doubleSlitTrajectoryLines.pov`	Szenenbeschreibung für die Animation, in der $\|\Psi\|^2$ als Raumkurve und mit 42 möglichen Teilchentrajektorien dargestellt ist.
`doubleSlitTrajectoryLines.ini`	Zugehörige INI-Datei.
`doubleSlitHitsOnScreen.pov`	Szenenbeschreibung für die Animation, in der ein Detektionsschirm abgebildet ist, auf dem nach und nach (ähnlich wie in Tonomuras Versuch) immer mehr Teilchen registriert werden.
`doubleSlitHitsOnScreen.ini`	Zugehörige INI-Datei.

Include-Dateien

Die folgenden `.inc`-Dateien enthalten Makros und geometrische Objekte, die in den verschiedenen Szenenbeschreibungsdateien verwendet werden.

Datei	Erklärung
`cameraPositions.inc`	Enthält verschiedene Kameras, die sich in ihren Positionen und Öffnungswinkeln voneinander unterscheiden.
`probDensity.inc`	Enthält die analytische Funktion der Wahrscheinlichkeitsdichte $\|\Psi(x,y,t)\|^2$ und Objekte die für deren Visualisierung genutzt werden können: Ein `isosurface`-Objekt für $\|\Psi(x,y,t)\|^2$ als Fläche im Raum sowie ein `union`-Objekt, das aus vielen kleinen Kugeln zusammengesetzt ist, um den Linienplot für $\|\Psi(v_{gx}t,y,t)\|^2$ darzustellen.
`drawTrajectories.inc`	Enthält für jedes Teilchen ein Makro, welches für das betreffende Teilchen eine schwarze Kugel an dem Ort platziert, wo es sich laut der von MATLAB ausgegebenen Datei, zum entsprechenden Zeitpunkt zu finden ist., die
`trajectoryLines.inc`	Enthält ein Makro, dass dünne Linien für die von Teilchen durchlaufenen Bahnen hinterlässt.
`trajectoryLineOneParticle.inc`	Enthält ein Makro, das hinter nur *einem* ausgewählten Teilchen eine Linie hinterlässt.
`probDensityAtFinalTime.inc`	Diese Datei wurde nur für die Videos geschrieben, bei denen nach und nach schwarze Punkte auf einem Detektionsschirm erscheinen. Sie enthält fast das gleiche wie die Datei `probDensity.inc`, wurde jedoch so verändert, dass die Objekte zur Visualisierung der Wahrscheinlichkeitsdichte am Detektionsschirm still stehen.

Kapitel 8

Möglichkeiten die Arbeit fortzusetzen

8.1 Beim Potentialtopf

Ohne allzu großen Aufwand könnten Videos für andere Kombinationen der Eigenfunktionen erstellt werden. Dazu wären nur kleine Änderungen am Code nötig. Es ist jedoch zu bedenken, dass die Winkelgeschwindigkeit, mit der sich die Raumkurven für die stationären Lösungen um die x-Achse drehen quadratisch mit n anwächst.

8.2 Beim Doppelspaltmodell

Andere Startpositionen

Die gewählten Anordnungen der Teilchen für den Start der Simulationen sind *keine* physikalische Notwendigkeit. Es wäre völlig legitim sich hier anders zu entscheiden! Hier besteht eine der zahlreichen Möglichkeiten den geschriebenen Code zu modifizieren und weitere interessante Videos zu erstellen. Das gilt sowohl für die Videos zum Potentialtopf als auch zum Doppelspaltversuch. Insbesondere beim Doppelspalt besteht eine interessante Möglichkeit die Startpositionen zu ändern und damit neue Trajektorien zu berechnen und darzustellen:

Man könnte Teilchen auch vor und hinter den Spalten platzieren und zusammen mit einer transparenten Fläche für die Wahrscheinlichkeitsdichte abbilden. Dabei könnte auf sehr intuitive Art und Weise gezeigt werden, dass die Teilchen auch in ihrer x-Komponente verschiedene Geschwindigkeiten haben können und gemäß der Verbreiterung der Wahrscheinlichkeitsverteilung auch in x-Richtung ihren Abstand zueinander vergrößern.

Visualisierung der komplexen Wellenfunktion

Bisher wurde nur das Betragsquadrat der Wellenfunktion dargestellt, wodurch nicht besonders deutlich wird, wie das Interferenzmuster zustande kommt. Ähnlich wie beim Potentialtopf, könnten Real- und Imaginärteil der Wellenfunktionen des linken und rechten Spaltes auf der Linie, die mit den Teilchen mit wandert, einzeln dargestellt werden. So könnte auch hier die komplexe Natur der Quantenmechanik verdeutlicht werden und anschaulich dargestellt werden, wie es genau zu den Interferenzmaxima und -minima kommt.

Kapitel 9

Fazit bezüglich der verwendeten Software und Methoden

9.1 POV-Ray

POV-Ray lieferte eine hohe grafische Qualität und nahezu unendliche Freiheiten bezüglich der Gestaltung. Diese Freiheiten werden aber erkauft durch ein erhebliches Maß an zusätzlicher Arbeit verglichen mit MATLAB, wenn es darum geht, einfache Plots für Testzwecke zu erstellen.

Als erheblicher Nachteil von POV-Ray hat sich während der Arbeit herausgestellt, dass die Syntax keine native Unterstützung für komplexe Zahlen bietet. So mussten Rechnungen für Real- und Imaginärteil jeweils einzeln durchgeführt werden. Teilweise mussten für die Rechnungen sogar andere Programme herangezogen werden. Mit am stärksten zeigte sich dieser Nachteil bei der Erstellung des Flächenplots für die Wahrscheinlichkeitsdichte beim Doppelspaltversuch. Hier wurde Mathematica zu verwendet, um für die Wahrscheinlichkeitsdichte einen vereinfachten Ausdruck zu erhalten, der keine imaginären Zahlen mehr enthielt.

9.2 MATLAB

MATLAB hat sich als durchweg sehr komfortabel und effizient erwiesen:

- Durch ein Vielzahl von eingebauten Funktionen reduziert sich die Menge des selbst zu schreibenden Codes.
- Durch die immer zu Verfügung stehende Kommandozeile lassen sich Teile des Codes einfach und schnell testen. Dadurch bekommt man sofortiges Feedback.
- Grafische Ausgaben sind schnell und einfach zu erstellen.
- Die grafische Oberfläche erleichtert den Arbeitsablauf in einigen Punkten.

9.3 Mathematica

Mathematica hat sich als rettendes Hilfsmittel erwiesen, als es darum ging beim Doppelspaltversuch die Wahrscheinlichkeitsdichte mit POV-Ray darzustellen. Mit Mathematica gelang es, einen vereinfachten Ausdruck ohne imaginäre Zahlen zu erhalten, wie er für POV-Ray benötigt wurde. Von Hand auf Papier wäre das eine seitenlange Rechnung gewesen. Wer Programmiersprachen wie C, MATLAB oder Python gewöhnt ist, wird ein wenig Zeit brauchen, um sich an Mathematica zu gewöhnen. Danach erweist es sich als mächtiges Werkzeug mit integrierten Möglichkeiten zur grafischen Darstellung.

9.4 ffmpeg

Das Kommandozeilenprogramm hat seinen Zweck erfüllt, indem es problemlos die von POV-Ray erzeugten Bilddateien zu einem mp4-Video zusammenfügte. Hinweis: In Zukunft soll

dieses Paket durch `avconv` abgelöst werden.

9.5 Schlussfolgerung

Es entstand einige zusätzliche Arbeit dadurch, das verschiedene Programme für die Erstellung der Animationen verwendet wurden. So mussten beispielsweise mit MATLAB Dateien erstellt werden, die von POV-Ray gelesen werden konnten. Eine andere Schwierigkeit: wenn man in POV-Ray globale Parameter änderte, musste man sie im MATLAB Code ebenfalls von Hand ändern und immer darauf achten, dass sie zusammenpassten. Nicht zuletzt mussten die meisten Funktionen schlicht doppelt implementiert werden.

Implementiert man alles in einer Programmiersprache, so erübrigen sich einige Arbeitsvorgänge, andere lassen sich besser automatisieren.

Ist der Anspruch an grafische Darstellungsmöglichkeiten nicht zu hoch, so ist es sicher einfacher und effizienter, mit einer Programmiersprache zu arbeiten, die in erster Linie für numerische Berechnungen ausgelegt ist. Für andere Bereiche der Physik, wo komplexe Zahlen eine kleinere Rolle spielen, könnte sogar alles mitsamt der nötigen Numerik in POV-Ray implementiert werden. POV-Ray umfasst eine Turing-vollständige Szenenbeschreibungssprache und kann daher grundsätzlich auch verwendet werden, um numerische Berechnungen auszuführen.

Anhang A

Code zum unendlich tiefen Potentialtopf

A.1 MATLAB Code zum Potentialtopf

Listing A.1: potentialWell.m

```
% coefficients
c = [sqrt(0.5), sqrt(0.5), 0, 0, 0];

% global parameters
global hbar;
hbar = 1;
global leng;
leng = 1;
global mass;
mass = 1;

% parameters for video
global periodOfGroundState;
periodOfGroundState = 4/pi;
global numberOfFrames;
numberOfFrames = 400;

wave(c,0.5,1)
```

Listing A.2: energy.m

```
function f = energy(n)
    global hbar;
    global leng;
    global mass;
    f = n*n*pi*pi*hbar*hbar/(2*mass*leng*leng);
end
```

Listing A.3: eigenStateWave.m

```
function f = eigenStateWave(n,x,t)
    global hbar;
    global leng;
    f = sqrt(2/leng)*sin(n*pi*x/leng)*exp(-1i*energy(n)*t/ hbar);
end
```

Listing A.4: wave.m

```
function f = wave(c,x,t)
    global hbar;
    global leng;
    global mass;
    f = 0;
    for n = 1:length(c)
        f = f + c(n)*eigenStateWave(n,x,t);
    end
end
```

Listing A.5: waveStarWave.m

```
function f = waveStarWave(c,x,t)
  f = conj(wave(c,x,t)) .* wave(c,x,t);
end
```

Listing A.6: probability.m

```
function p = probability(c,t,lowerX,upperX,numberOfIntergrationSteps)
    dx = (upperX - lowerX)/numberOfIntergrationSteps;
    x = lowerX;
    p = 0;
    while x < upperX
        p = p + waveStarWave(c,x,t)*dx;
        x = x + dx;
    end
end
```

Listing A.7: waveDerivative.m

```
function f = waveDerivativeX(c,x,t)
    global hbar;
    global leng;
    global mass;
    f = 0;
    for n = 1:length(c)
        f = f + c(n)*eigenStateWaveDerivativeX(n,x,t);
    end

end


function y = eigenStateWaveDerivativeX(n,x,t)
    global hbar;
    global leng;
    y = sqrt(2/leng)*(n*pi/leng)*cos(n*pi*x/leng)*exp(-1i*energy(n)*t/ hbar);
end
```

Listing A.8: qDot.m

```
function dqbydt = qDot(c,x,t)
  global hbar;
  global mass;
  dqbydt = (hbar/mass) * imag(conj(wave(c,x,t)).*waveDerivativeX(c,x,t)./waveStarWave(c,x
    ,t));
end
```

Listing A.9: q.m

```
function position = q(c,qStart,dt,endTime)
  position = qStart;
  t = 0;
  while t <= endTime
    position = position + qDot(c,position,t) * dt;
    t = t + dt;
  end
  position
end
```

Listing A.10: plotQ.m

```
function plotQ(c,qStart,dt,N,endTime,color) %plots every N-th of all calculated particle
    positions
  position = qStart;
  t = 0;
  count = 0;
  hold on
  while t <= endTime
    position = position + qDot(c,position,t) * dt;
    if mod(count,N) == 0
      plot(t,position,color)
    end
    t = t + dt;
    count = count + 1;
  end
end
```

Listing A.11: initialPositions.m

```
function positionArray = initialPositions(c,numberOfParticles,numberOfIntergrationSteps)
    x = 0;
    prob = 0;
    global leng;
    dx = leng/numberOfIntergrationSteps;
    count = 1;
    while count <= numberOfParticles
        prob = prob + waveStarWave(c,x,0)*dx;
        if (prob > count/(numberOfParticles+1))
            positionArray(count) = x;
            count = count+1;
        end
```

```
    end
    positionArray;
    prob;
end
```

Listing A.12: printTrajectoryToFile.m

```
function printTrajectoryToFile(c,qStart,stepsBetweenFrames,fileName,particleID)
  % prints values for particle position into file for use in povray
  % time difference between Frames: periodOfGroundState/numberOfFrames
  % function plotQ(c,qStart,dt,N,endTime,color)
  global periodOfGroundState;
  global numberOfFrames;
  dt = periodOfGroundState/(numberOfFrames * stepsBetweenFrames); % dt for numerical
     integration

  fileID = fopen(fileName, 'w');
    fprintf(fileID, '#declare TrajectoryValuesFromMatlab%d = array[%d]\n', particleID,
     numberOfFrames);
    fprintf(fileID, '{\n');

    position = qStart;
  t = 0;
  count = 0; % counts integration loop iterations, not frames
    countFrame = 1; % counts frames
  while countFrame <= numberOfFrames
    position = position + qDot(c,position,t) * dt;
    if mod(count,stepsBetweenFrames) == 0
            fprintf(fileID, '%f', position);
            if (countFrame ~= (numberOfFrames))
                fprintf(fileID,',\n');
            end
            countFrame = countFrame + 1;
    end
    t = t + dt;
    count = count + 1;
    end

    fprintf(fileID,'\n}\n');
    fprintf(fileID,'// dt == %f\n', dt);
    fprintf(fileID,'// count == %f\n', count);
    fprintf(fileID,'// countFrame == %f\n', countFrame);
    fprintf(fileID,'// numberOfFrames == %f\n', numberOfFrames);

end
```

A.2 POV-Ray Code zum Potentialtopf

Listing A.13: potentialWell.pov

```
#include "colors.inc"
#include "resultingWave.inc"
#include "eigenFunctions.inc"
#include "probDensity.inc"
#include "energy.inc"
#include "trajectoryValuesFromMatlab1.inc"
#include "trajectoryValuesFromMatlab2.inc"
#include "trajectoryValuesFromMatlab3.inc"
#include "trajectoryValuesFromMatlab4.inc"
#include "trajectoryValuesFromMatlab5.inc"
#include "trajectoryValuesFromMatlab6.inc"
#include "trajectoryValuesFromMatlab7.inc"
#include "trajectoryValuesFromMatlab8.inc"
#include "trajectoryValuesFromMatlab9.inc"
#include "drawTrajectory.inc"

//Constants:::::::::::::::::::::::::::::::::::::::::::::::::
#declare Hbar = 1;
#declare Leng = 1;
#declare L = 1;
#declare Mass = 1;
#declare Pi = 3.14159;
#declare PeriodOfGroundState = 4/Pi;
//"PeriodOfGroundState" is a multiple of all other eigenstates' periods
#declare DrawStep = 0.001;

//Time settings for animation:::::::::::::::::::::::::::::
#declare Start = 0;
#declare End = PeriodOfGroundState;
#declare Time = (End - Start)* clock;
// "Time" is actual physical time, same units as in Matlab
// "clock" goes from 0 to 1 (as defined in INI-file)
#declare NumberOfFrames = 400;
// "NumberOfFrames" needs to be set according to "Final_Frame" in INI-file
#declare TimePerFrame = PeriodOfGroundState/NumberOfFrames;
```

```
#declare FrameNumber = clock * NumberOfFrames;

//Camera, light and background:::::::::::::::::::::::::::::
background {White}

camera {
  angle 33
  location <4.8, 2.5 , -5.2>
  look_at <1,0.2,0>
  rotate <0,0,0>
}

light_source {
  <40,50,-40>
  color White
  parallel
  shadowless
}

plane { y, -2
  pigment { checker White, Goldenrod scale 0.5 }
}

//Coordinate System with gray axes:::::::::::::::::::::::::
#declare ThicknessAxes = 0.007;
#declare LengthAxes = 1;
#declare ThicknessCone = 0.02;
#declare LengthCone= 0.04;
#declare ColorAxes = Gray40;

cylinder { <0,0,0>, <2,0,0>, ThicknessAxes
  pigment {ColorAxes}
}
cone {<2,0,0>,ThicknessCone <2+LengthCone,0,0>, 0
  pigment {ColorAxes}
}
cylinder { <0,0,0>, <0,LengthAxes,0>, ThicknessAxes
  pigment {ColorAxes}
}
cone { <0,LengthAxes,0>, ThicknessCone <0,LengthAxes + LengthCone,0>, 0
  pigment {ColorAxes}
}
cylinder { <0,0,0>, <0,0,-1>, ThicknessAxes
  pigment {ColorAxes}
}
cone { <0,0,-LengthAxes>, ThicknessCone <0,0,-(LengthAxes + LengthCone)>, 0
  pigment {ColorAxes}
}

//Coefficients (mix of eigenfunctions that we want)::::::::
#declare c = array [6] {0,sqrt(0.5),sqrt(0.5),0,0,0};
//NOTE: first coefficient must be chosen zero!!!
//Fist eigenfunction has index n==1!
//Array indices start with a zero in Povray!
//Notation was chosen such that c_1 == c[1].

//Wavefunctions of eigenstates:::::::::::::::::::::::::::::
EigenFunctions(c,Time)

//Resulting wavefunction:::::::::::::::::::::::::::::::::::
ResultingWave(c,Time)

//Probability density:::::::::::::::::::::::::::::::::::::
ProbDensCurve(c,Time)

//Particles:
DrawParticleFromMatlabValues1(FrameNumber)
DrawParticleFromMatlabValues2(FrameNumber)
DrawParticleFromMatlabValues3(FrameNumber)
DrawParticleFromMatlabValues4(FrameNumber)
DrawParticleFromMatlabValues5(FrameNumber)
DrawParticleFromMatlabValues6(FrameNumber)
DrawParticleFromMatlabValues7(FrameNumber)
DrawParticleFromMatlabValues8(FrameNumber)
DrawParticleFromMatlabValues9(FrameNumber)

//Descriptive text on the side:::::::::::::::::::::::::::::
global_settings {charset utf8}
text{
   ttf "cyrvetic.ttf",
   "Re",0.1, 0
   texture{
     pigment{color ColorAxes}
     finish{ambient 0.15
            diffuse 0.85}
          }
   scale 0.1
   rotate <0,-90,0>
   translate<0,1.05,-0.05>
    }

```

```
text{
   ttf "cyrvetic.ttf",
   "Im",0.1, 0
   texture{
     pigment{color ColorAxes}
     finish{ambient 0.15
            diffuse 0.85}
          }
   scale 0.1
   rotate <0,-90,0>
   translate<0,-.03,-1.15>
}

#declare LegendShift = 1.6;
#declare LegendScale = 0.06;
union {
  text{
     ttf "cyrvetic.ttf",
     "Wavefunctions of the Eigenstates",0.1, 0
     texture{
     pigment{color Red}
     finish{ambient 0.15
            diffuse 0.85}
          }
     scale LegendScale
     rotate <0,0,0>
     translate<LegendShift,1.2,0>
  }

  text{
     ttf "cyrvetic.ttf",
     "Resulting Wavefunction",0.1, 0
     texture{
     pigment{color Green}
     finish{ambient 0.15
            diffuse 0.85}
          }
     scale LegendScale
     rotate <0,0,0>
     translate<LegendShift,1.1,0>
  }

  text{
     ttf "cyrvetic.ttf",
     "Probability Density",0.1, 0
     texture{
     pigment{color Blue}
     finish{ambient 0.15
            diffuse 0.85}
          }
     scale LegendScale
     rotate <0,0,0>
     translate<LegendShift,1.0,0>
  }

  text{
     ttf "cyrvetic.ttf",
     "Representative Bohmian Particles",0.1, 0
     texture{
     pigment{color Black}
     finish{ambient 0.15
            diffuse 0.85}
          }
     scale LegendScale
     rotate <0,0,0>
     translate<LegendShift,0.9,0>
  }

  scale 1.3 //scales the whole union
  translate <-0.4,-0.5,0>
}
```

Listing A.14: potentialWell.ini

```
+w1200
+h900

Input_File_Name=potentialWell.pov


Antialias=On

Antialias_Threshold=0.1
Antialias_Depth=2



Initial_Frame=1
Final_Frame=400
Initial_Clock=0
```

```
Final_Clock=1

Subset_Start_Frame=1
Subset_End_Frame=400

Cyclic_Animation=on
Pause_when_Done=off
```

Listing A.15: energy.inc

```
#macro Energy(n)
  n*n*Pi*Pi*Hbar*Hbar/(2*Mass*Leng*Leng)
#end
```

Listing A.16: eigenFunctions.inc

```
#declare Ball = sphere{<0,0,0>,0.012
  texture{
    pigment{color Red}
        finish {ambient 0.15
          diffuse 0.85
          phong 1
        }
  }
}


#macro EigenFunctions(c,time)
  #local n = 1;
  #while (n < 6)
    #if (c[n] != 0)
      #local i = 0;
      #while (i < 1)
        object {Ball
          translate <i*L, c[n] * sin(n * Pi * i), 0>
          rotate <Energy(n)*time*180/Pi,0,0>
        }
        #local i = i + DrawStep;
      #end
    #end
    #local n = n + 1;
  #end
#end
```

Listing A.17: resultingWave.inc

```
#macro RealPartWave(c,i,time)
  #local n = 1;
  #local Re = 0;
  #while (n < 6)
    #local Re = Re + c[n]*sin(n*Pi*i) * cos(Energy(n)*time/Hbar); // ACHTUNG: sin( mit
     RAD)
    #local n=n+1;                          //obwohl rotate mit degrees
  #end
  Re
#end

#macro ImaginaryPartWave(c,i,time)
  #local n = 1;
  #declare Im = 0;
  #while (n < 6)
    #declare Im = Im + c[n]*sin(n*Pi*i) * sin(Energy(n)*time/Hbar);
    #local n=n+1;
  #end
  Im
#end

#macro ResultingWave(c,time)
    #local i = 0;
    #while (i < 1)
      object {Ball
        translate <i*L, RealPartWave(c,i,time), ImaginaryPartWave(c,i,time)>
        pigment {color Green}
      }
      #local i = i + DrawStep;
    #end
#end
```

Listing A.18: probDensity.inc

```
#macro ProbDensPoints(c,i,time)
  RealPartWave(c,i,time)*RealPartWave(c,i,time) + ImaginaryPartWave(c,i,time)*
     ImaginaryPartWave(c,i,time)
#end

#macro ProbDensCurve(c,time)
```

```
  #local i = 0;
  #while (i < 1)
    object {Ball
      translate <i*L, ProbDensPoints(c,i,time), 0>
      pigment {color Blue}
    }
    #local i = i + DrawStep;
  #end
#end
```

Listing A.19: drawTrajectory.inc

```
#declare Particle = sphere {
  <0,0,0>,0.02
  texture{pigment{color Black}
  finish {ambient 0.15 diffuse 0.85 phong 1}
    }
}

#macro DrawParticleFromMatlabValues1(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab1[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues2(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab2[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues3(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab3[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues4(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab4[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues5(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab5[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues6(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab6[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues7(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab7[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues8(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab8[framenumber],0,0>}
#end

#macro DrawParticleFromMatlabValues9(framenumber) //draws the particle at every instant
    of time
  object {Particle translate <TrajectoryValuesFromMatlab9[framenumber],0,0>}
#end
```

Anhang B

Code zum Doppelspaltmodell

B.1 MATLAB Code zum Doppelspaltmodell

Listing B.1: doubleSlit.m

```
% Bachelorarbeit Marlon Metzger, LMU Muenchen, Oktober 2013

% 2D simulation of the Bohmian trajectories of particles originating from
% the slits and guided by two identical gaussian wavepackets

% two slits on the y-axis, screen perpendicular on x-axis


% Global constants:
global hbar;
hbar = 1;
global mass;
mass = 2;
global beta;
beta = hbar / (2*mass);
%groupVelocity;
global groupVelocityX;
groupVelocityX = 5;
global groupVelocityY;
groupVelocityY= 0;
global slitDistance;
slitDistance = 4;
global sigmaX;
sigmaX = 1; % width of packet at t==0;
global sigmaY;
sigmaY = 1/5;

% Parameters depending on animation:
global numberOfFrames;
numberOfFrames = 481;
global totalTime;
totalTime = 2.4;
global integrationStepsBetweenFrames;
integrationStepsBetweenFrames = 3;
%global integrationTimeStep;
%integrationTimeStep = 1/((numberOfFrames-1)*10);
global qStepForGradient;
qStepForGradient = 0.000001;

% Slit with positive y:
particlesLeft = 21;
yStartLeft = yStartArrayNormalDist(particlesLeft, slitDistance/2, 0.001);
for particleID = 1:length(yStartLeft)
    fileName = sprintf('trajectory%d.inc', particleID);
    printTrajectoryToFile(0,yStartLeft(particleID),fileName,particleID)
end
% Right slit:
particlesRight = 21;
yStartRight = yStartArrayNormalDist(particlesRight, -slitDistance/2, 0.001);
for particleID = particlesLeft+1:(particlesLeft+particlesRight)
    fileName = sprintf('trajectory%d.inc', particleID);
    printTrajectoryToFile(0,yStartRight(particleID-particlesLeft),fileName,particleID)
end

% printTrajectoryToFile(0,-1,'trajectory1.inc',11);
% printTrajectoryToFile(0,'trajectory1.inc',12);
% printTrajectoryToFile(0,0,'trajectory1.inc',13);
% printTrajectoryToFile(0,0,'trajectory1.inc',14);
% printTrajectoryToFile(0,0,'trajectory1.inc',15);
% printTrajectoryToFile(0,0,'trajectory1.inc',16);
% printTrajectoryToFile(0,0,'trajectory1.inc',17);
% printTrajectoryToFile(0,0,'trajectory1.inc',18);
```

```
% printTrajectoryToFile(0,0,'trajectory1.inc',19);
```

Listing B.2: alpha.m

```
% for the names of the variables see Torsten Fliessbach, Quantenmechanik, pages 63 to 67
% alpha is only needed for a simpler expression in the gaussian wave packet


function alpha = alpha(sigmaStart)
    alpha = sigmaStart.^2;
end
```

Listing B.3: waveNumber.m

```
% Amplitude funtion has maximum at the wavenummber k = k_0.
% This function allows to express k_0 in terms of the group velocity.
% The formula for the de Broglie wavelength relates the two quantities.

function waveNumber = waveNumber(groupVelocity)
    global hbar;
    global mass;
    waveNumber = groupVelocity * mass / hbar;
end
```

Listing B.4: omega.m

```
% qm dispersion relation

function omega = omega(waveNumber) % waveNumber k
    global hbar;
    global mass;
    omega = hbar * waveNumber^2 / (2*mass);
end
```

Listing B.5: gaussPacket.m

```
% Gaussian wave packet with an initial standard deviation of sigmaStart

function gaussPacket = gaussPacket(position, t, initialPosition, sigmaStart,
    groupVelocity)
    global beta;
    gaussPacket = 1/(sqrt(alpha(sigmaStart) + 1i.*beta.*t)) ...
        .* exp( 1i.*waveNumber(groupVelocity).*(position - initialPosition) - 1i.*omega(
    waveNumber(groupVelocity)).*t ) ...
        .* exp( - ((position-initialPosition-groupVelocity.*t).^2 ./ (4.*(alpha(
    sigmaStart) + 1i.*beta.*t))));
end
```

Listing B.6: psiX.m

```
function psiX = psiX(x,t)
    global sigmaX;
    global groupVelocityX;
    psiX = gaussPacket(x,t,0,sigmaX,groupVelocityX);
    % psiX = gaussPacket(x,t,0,sigmaStart,groupVelocity);
end
```

Listing B.7: psiY.m

```
function psiY = psiY(y,t)
    global slitDistance;
    global sigmaY;
    global groupVelocityY;
    psiY =  gaussPacket(y,t, slitDistance/2, sigmaY, groupVelocityY) ...
                    + gaussPacket(y,t,-slitDistance/2, sigmaY, groupVelocityY);
end
```

Listing B.8: waveFunction.m

```
function waveFunction = waveFunction(x,y,t)
    waveFunction = psiX(x,t).*psiY(y,t);
end
```

Listing B.9: waveFunctionDerivativeX.m

```
% dq/dt = hbar/mass * imag( grad(psi(x,y,t)) /psi(x,y,t))

function dpsiBydx = waveFunctionDerivativeX(x,y,t,dx)
    dpsi = (waveFunction(x+0.5*dx,y,t) - waveFunction(x-0.5*dx,y,t));
    dpsiBydx =dpsi/dx;
end
```

Listing B.10: waveFunctionDerivativeY.m

```
function dpsiBydy = waveFunctionDerivativeY(x,y,t,dy)
    dpsi = (waveFunction(x,y+0.5*dy,t) - waveFunction(x,y-0.5*dy,t));
    dpsiBydy =dpsi/dy;
end
```

Listing B.11: probDensity.m

```
function probDensity = probDensity(x,y,t)
    probDensity = (abs(psiX(x,t).*psiY(y,t))).^2;
end
```

Listing B.12: yStartArrayNormalDist.m

```
function yStartArray = yStartArrayNormalDist(numberOfParticles,centeredAt,
    yIntegrationStep)
    global sigmaY;
    y = 0;
    yStartArray = zeros(1,numberOfParticles);
    indexOfNextParticleToPlace = 1; %
    % For every two particles next to each other there must be the same probability
    between them.
    % Therefore, towards the center of the two slits, from y==0 to the first particle the
    % probability step must be only half of the other steps!
    % Recall that we have another Gaussian packet at the negative y side.
    probValues = [0.5/(numberOfParticles+0.5):1/(numberOfParticles+0.5):1]*0.5 + 0.5; %
    +0.5 for the probability on negative y side
    %probStep = 1/(numberOfParticles+0.5);
    while (indexOfNextParticleToPlace <= numberOfParticles)
        y = y+sign(centeredAt)*yIntegrationStep;
        if (accumulatedProbability(sign(centeredAt)*y,2*centeredAt,sigmaY) > probValues(
    indexOfNextParticleToPlace))
            yStartArray(indexOfNextParticleToPlace) = y;
            indexOfNextParticleToPlace = indexOfNextParticleToPlace + 1;
        end
    end
end

% Take in account both Gaussian Packets in case they are already overlapping
% at the start:
function acc = accumulatedProbability(y,slitDistance,sigmaY)
    acc = 0.5 * (normcdf(y,-slitDistance/2,sigmaY)+normcdf(y,slitDistance/2,sigmaY));
end
```

Listing B.13: printTrajectoryToFile.m

```
function printTrajectoryToFile(xStart,yStart,fileName,particleID)
  % prints vectors for particle position into file for use in povray
    global numberOfFrames;
    global integrationStepsBetweenFrames;
    global totalTime;
    global qStepForGradient;

  fileID = fopen(fileName, 'w');
    fprintf(fileID, '#declare TrajectoryValuesFromMatlab%d = array[%d]\n', particleID,
    numberOfFrames);
    fprintf(fileID, '{\n');


    [xArray,yArray] = qArray(xStart,yStart,qStepForGradient,qStepForGradient,
    integrationStepsBetweenFrames,numberOfFrames,totalTime);
    if (length(xArray) ~= numberOfFrames)
        sprintf('WARNING: length(xArray) ~= numberOfFrames')
        sprintf('length(xArray)==%d', length(xArray))
        sprintf('numberOfFrames==%d', numberOfFrames)
    end
    for index = 1:length(xArray)
        fprintf(fileID, '<%f,%f,0>',xArray(index),yArray(index)); % will later have to
    lift up particles from floor
        if (index ~= length(xArray))
            fprintf(fileID,',\n');
        end
    end

    fprintf(fileID,'\n}\n');
    fprintf(fileID,'// integrationStepsBetweenFrames == %f\n',
    integrationStepsBetweenFrames);
    fprintf(fileID,'// numberOfFrames == %f\n', numberOfFrames);
    fprintf(fileID,'// numberOfOutputtedFrames == %f\n', length(xArray));

end
```

Listing B.14: printHitsOnScreenToFile.m

```
% This function is for a video, where single particles appear randomly
```

```
% ditributed on a screen according to probDensity at the time, when the
% wave packets hit the screen

% first output: file containing povray array, each entry is a
% vector <screenPosition, y of particle i, z of particle i>

% second output: array with a random permutation of all integers between 0 and
% numberOfParticles-1. This is for povray to show the particles in random
% order.

function y = printHitsOnScreenToFile(screenPosition,screenWidth,screenHeight,
    numberOfParticles,yIntegrationStep,fileNameEnding)
    global groupVelocityX;
    time = screenPosition/groupVelocityX;
    yMin = -0.5*screenWidth;
    yMax = 0.5*screenWidth;

    %probDensity(x,y,t)
    % find out area under probDensity for our domain
    area = 0;
    y = yMin;
    while y < yMax
        area = area + probDensity(screenPosition, y, time) * yIntegrationStep;
        y = y + yIntegrationStep;
    end
    % create array with random numbers
    randomArrayUniformDist = area * rand(1,numberOfParticles);
    sortedRandomArrayUniformDist = sort(randomArrayUniformDist);


    %open file
    fileName = strcat('hitsOnScreenArray',fileNameEnding,'.inc');
  fileID = fopen(fileName, 'w');
    fprintf(fileID, '#declare HitsOnScreenArray');
    fprintf(fileID, fileNameEnding);
    fprintf(fileID,'= array[%d]\n', numberOfParticles);
    fprintf(fileID, '{\n');

    % intitialize variables before loop
    cumulatedProbability = 0;
    y = yMin;
    checkCount = 0; % only for debugging
    index = 1;
    while (y < yMax && index <= numberOfParticles)
        %integriere immer bis zum naechsten Eintrag im Array, platziere Teilchen
        cumulatedProbability = cumulatedProbability + probDensity(screenPosition, y, time
    ) * yIntegrationStep;
        y = y + yIntegrationStep;
        if (cumulatedProbability > sortedRandomArrayUniformDist(index))
            %print particle to file with random z value
            randomHeight = screenHeight*rand(1);
            fprintf(fileID, '<%f,%f,%f>', screenPosition, y, randomHeight);

            if (index < numberOfParticles)
                fprintf(fileID,',\n');
            end
            checkCount = checkCount + 1;
            index = index + 1;
        end
    end
    fprintf(fileID, '\n} \n');
    %append some values on the bottom of the file (for debugging):
    fprintf(fileID, '//area under probDensity = %f \n', area);
    fprintf(fileID, '//cumulatedProbability = %f \n', cumulatedProbability);
    fprintf(fileID, '//numberOfParticles hitting screen = %d \n', numberOfParticles);
    fprintf(fileID, '//checkCount = %d \n', checkCount);


    % create array with randomly permuted numbers, so that povray can show particles in
    random order
    randomPermutationArray = randperm(numberOfParticles) - 1; % -1 as array indices start
     at 0 in povray
    fprintf(fileID, '\n');
    fprintf(fileID, '#declare RandomPermutationArray');
    fprintf(fileID, fileNameEnding);
    fprintf(fileID,'= array[%d]\n', numberOfParticles);
    fprintf(fileID, '{\n');
    for i = 1:length(randomPermutationArray)
         fprintf(fileID, '%d',randomPermutationArray(i));
         if (i < numberOfParticles)
             fprintf(fileID, ',\n');
         end
    end
    fprintf(fileID, '} \n');
end
```

B.2 POV-Ray Code zu Doppelspaltmodell

Listing B.15: doubleSlitWaveOnly.pov

```
// Bachelorarbeit Marlon Metzger, LMU Muenchen, October 2013

// In this scene there are no particles, just the isosurface for the probability density

// Povray uses a left-handed coordinate system!!!
// Slits are situated on the y-axis at z==0

#include "colors.inc"
#include "complexNumbers.inc"
#include "gauss.inc"
#include "littleFunctions.inc"
#include "cameraPositions.inc"


//Constants:
#declare Hbar = 1;
#declare Mass = 2;
#declare Pi = 3.14159;

#declare Beta = Hbar/(2*Mass);
#declare GroupVelocityX = 5;
#declare GroupVelocityY = 0;
#declare SlitDistance = 4;
#declare SigmaX = 1;
#declare SigmaY = 0.2;

//Time settings for animation:
#declare Start = 0;
#declare End = 2.4; //TODO needs to be set according to "totalTime" in Matlab
#declare Time = (End - Start)* clock; // Time is actual physical time, same unit as in
    Matlab, clock goes from 0 to 1
#declare NumberOfFrames = 481; //TODO needs to be set according to "Final_Frame" in topf.
    ini file
//#declare TimePerFrame = PeriodOfGroundState/NumberOfFrames;
#declare FrameNumber = clock * NumberOfFrames;

#include "probDensity.inc"


// Camera, light and background:
background {White}

camera {CameraCenteredMediumWIDE} // for an aspect ratio of 16:9


light_source { <40,50,40>
        color White
        //parallel
        shadowless
}

light_source { <-0,-0,40>
        color White
        parallel
        fade_distance 20
        fade_power 2
        //shadowless
}


plane { z, 0
  pigment { checker White, Gray90 scale 1 }
}

// Coordinate axes (left-handed):
#declare ThicknessAxes = 0.007;
#declare LengthAxes = 1;
#declare ThicknessCone = 0.02;
#declare LengthCone= 0.04;
#declare ColorAxes = Gray40;

//cylinder { <0,0,0>, <2,0,0>, ThicknessAxes pigment {ColorAxes} }
//cone {<2,0,0>,ThicknessCone <2+LengthCone,0,0>, 0 pigment {ColorAxes} }
//cylinder { <0,0,0>, <0,LengthAxes,0>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,LengthAxes,0>, ThicknessCone <0,LengthAxes + LengthCone,0>, 0 pigment {
    ColorAxes} }
//cylinder { <0,0,0>, <0,0,1>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,0,LengthAxes>, ThicknessCone <0,0,(LengthAxes + LengthCone)>, 0 pigment {
    ColorAxes} }

// Slit:
#declare SlitSize = 4*SigmaY;
#declare WallThickness = 0.05;
#declare WallHeight = 3;
#declare WallLength = 12;
```

```
#declare WallColor = Gray20;

difference {
  box {<.5*WallThickness,.5*WallLength,.5*WallHeight>
    < -0.5*WallThickness, -.5*WallLength, -.5*WallHeight>
  }

  box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
    translate <0,.5*SlitDistance,0>
  }

  box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
    translate <0,-.5*SlitDistance,0>
  }
  pigment{ color WallColor}
  finish {phong 0.5}
}

// ProbDensitySurface:
object {ProbDensitySurface}
```

Listing B.16: doubleSlitWaveOnly.ini

```
+w1280
+h720

Input_File_Name=doubleSlitWaveOnly.pov
Output_File_Name=doubleSlitWaveOnly.png

Antialias=On
Antialias_Threshold=0.1
Antialias_Depth=2

Initial_Frame=1
Final_Frame=481; always set this according to "numberOfFrames" in Matlab code
Initial_Clock=0
Final_Clock=1

Subset_Start_Frame=1
Subset_End_Frame=481

Cyclic_Animation=off
Pause_when_Done=off
```

Listing B.17: doubleSlitLineOneParticle.pov

```
// Bachelorarbeit Marlon Metzger, LMU Muenchen, October 2013

// In this scene there is just one particle leaving a thin line behind itself

// >>>>>>>>>>>>>>> choose particle at the end of the file <<<<<<<<<<<<<<<

// Povray uses a left-handed coordinate system!!!
// Slits are situated on the y-axis at z==0

#include "colors.inc"
#include "complexNumbers.inc"
#include "gauss.inc"
#include "littleFunctions.inc"
#include "cameraPositions.inc"

#include "drawTrajectories.inc"

#include "Trajectories/trajectory1.inc"
#include "Trajectories/trajectory2.inc"
#include "Trajectories/trajectory3.inc"
#include "Trajectories/trajectory4.inc"
#include "Trajectories/trajectory5.inc"
#include "Trajectories/trajectory6.inc"
#include "Trajectories/trajectory7.inc"
#include "Trajectories/trajectory8.inc"
#include "Trajectories/trajectory9.inc"
#include "Trajectories/trajectory10.inc"
#include "Trajectories/trajectory11.inc"
#include "Trajectories/trajectory12.inc"
#include "Trajectories/trajectory13.inc"
#include "Trajectories/trajectory14.inc"
#include "Trajectories/trajectory15.inc"
#include "Trajectories/trajectory16.inc"
#include "Trajectories/trajectory17.inc"
#include "Trajectories/trajectory18.inc"
#include "Trajectories/trajectory19.inc"
#include "Trajectories/trajectory20.inc"
#include "Trajectories/trajectory21.inc"

#include "Trajectories/trajectory22.inc"
#include "Trajectories/trajectory23.inc"
#include "Trajectories/trajectory24.inc"
#include "Trajectories/trajectory25.inc"
```

```
#include "Trajectories/trajectory26.inc"
#include "Trajectories/trajectory27.inc"
#include "Trajectories/trajectory28.inc"
#include "Trajectories/trajectory29.inc"
#include "Trajectories/trajectory30.inc"
#include "Trajectories/trajectory31.inc"
#include "Trajectories/trajectory32.inc"
#include "Trajectories/trajectory33.inc"
#include "Trajectories/trajectory34.inc"
#include "Trajectories/trajectory35.inc"
#include "Trajectories/trajectory36.inc"
#include "Trajectories/trajectory37.inc"
#include "Trajectories/trajectory38.inc"
#include "Trajectories/trajectory39.inc"
#include "Trajectories/trajectory40.inc"
#include "Trajectories/trajectory41.inc"
#include "Trajectories/trajectory42.inc"


//Constants:
#declare Hbar = 1;
#declare Mass = 2;
#declare Pi = 3.14159;

#declare Beta = Hbar/(2*Mass);
#declare GroupVelocityX = 5;
#declare GroupVelocityY = 0;
#declare SlitDistance = 4;
#declare SigmaX = 1;
#declare SigmaY = 0.2;

//Time settings for animation:
#declare Start = 0;
#declare End = 2.4; //TODO needs to be set according to "totalTime" in Matlab
#declare Time = (End - Start)* clock; // Time is actual physical time, same unit as in
    Matlab, clock goes from 0 to 1
#declare NumberOfFrames = 481; //TODO needs to be set according to "Final_Frame" in topf.
    ini file
//#declare TimePerFrame = PeriodOfGroundState/NumberOfFrames;
#declare FrameNumber = clock * (NumberOfFrames-1) + 1;
// FrameNumber starts at 1, finishes at NumberOfFrames

#include "trajectoryLineOneParticle.inc"
#include "trajectoryLines.inc"
#include "probDensity.inc"

// Camera, light and background:
background {White}

//camera {CameraCenteredMedium}
camera {CameraCenteredMediumWIDE} //16:9 aspect ratio

light_source {
  <40,50,40>
  color White
  //parallel
  shadowless
}

plane { z, 0
  pigment { checker White, Gray90 scale 1 }
}

// Coordinate axes (left-handed):
#declare ThicknessAxes = 0.007;
#declare LengthAxes = 1;
#declare ThicknessCone = 0.02;
#declare LengthCone= 0.04;
#declare ColorAxes = Gray40;

//cylinder { <0,0,0>, <2,0,0>, ThicknessAxes pigment {ColorAxes} }
//cone {<2,0,0>,ThicknessCone <2+LengthCone,0,0>, 0 pigment {ColorAxes} }
//cylinder { <0,0,0>, <0,LengthAxes,0>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,LengthAxes,0>, ThicknessCone <0,LengthAxes + LengthCone,0>, 0 pigment {
    ColorAxes} }
//cylinder { <0,0,0>, <0,0,1>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,0,LengthAxes>, ThicknessCone <0,0,(LengthAxes + LengthCone)>, 0 pigment {
    ColorAxes} }

// Slit:
#declare SlitSize = 4*SigmaY;
#declare WallThickness = 0.05;
#declare WallHeight = 3;
#declare WallLength = 12;
#declare WallColor = Gray20;

difference {
  box {<.5*WallThickness,.5*WallLength,.5*WallHeight>
    < -0.5*WallThickness, -.5*WallLength, -.5*WallHeight>
  }

```

```
  box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
    translate <0,.5*SlitDistance,0>
  }

  box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
    translate <0,-.5*SlitDistance,0>
  }
  pigment{ color WallColor}
  finish {phong 0.5}
}



// ProbDensityLine:
object {ProbDensityLine}

// Particle:
#declare ParticleRadius = 0.04;
DrawParticleFromMatlabValues2(FrameNumber,ParticleRadius) //choose particle here

// Trajectoy lines:
#declare RadiusTrajectoryLines = 0.01;
object {TrajectoryLineOneParticle(RadiusTrajectoryLines)}
```

Listing B.18: doubleSlitLineOneParticle.ini

```
+w1280
+h720

Input_File_Name=doubleSlitLineOneParticle.pov
Output_File_Name=doubleSlitLineOneParticle.png

Antialias=On
Antialias_Threshold=0.1
Antialias_Depth=2

Initial_Frame=1
Final_Frame=481; always set this according to "numberOfFrames" in Matlab code
Initial_Clock=0
Final_Clock=1

Subset_Start_Frame=1
Subset_End_Frame=481

Cyclic_Animation=off
Pause_when_Done=off
```

Listing B.19: doubleSlitTrajectoryLines.pov

```
// Bachelorarbeit Marlon Metzger, LMU Muenchen, October 2013

// In this scene the particles leave thin lines behind them

// Povray uses a left-handed coordinate system!!!
// Slits are situated on the y-axis at z==0

#include "colors.inc"
#include "complexNumbers.inc"
#include "gauss.inc"
#include "littleFunctions.inc"
#include "cameraPositions.inc"

#include "drawTrajectories.inc"

#include "Trajectories/trajectory1.inc"
#include "Trajectories/trajectory2.inc"
#include "Trajectories/trajectory3.inc"
#include "Trajectories/trajectory4.inc"
#include "Trajectories/trajectory5.inc"
#include "Trajectories/trajectory6.inc"
#include "Trajectories/trajectory7.inc"
#include "Trajectories/trajectory8.inc"
#include "Trajectories/trajectory9.inc"
#include "Trajectories/trajectory10.inc"
#include "Trajectories/trajectory11.inc"
#include "Trajectories/trajectory12.inc"
#include "Trajectories/trajectory13.inc"
#include "Trajectories/trajectory14.inc"
#include "Trajectories/trajectory15.inc"
#include "Trajectories/trajectory16.inc"
#include "Trajectories/trajectory17.inc"
#include "Trajectories/trajectory18.inc"
#include "Trajectories/trajectory19.inc"
#include "Trajectories/trajectory20.inc"
#include "Trajectories/trajectory21.inc"

#include "Trajectories/trajectory22.inc"
#include "Trajectories/trajectory23.inc"
#include "Trajectories/trajectory24.inc"
```

```
#include "Trajectories/trajectory25.inc"
#include "Trajectories/trajectory26.inc"
#include "Trajectories/trajectory27.inc"
#include "Trajectories/trajectory28.inc"
#include "Trajectories/trajectory29.inc"
#include "Trajectories/trajectory30.inc"
#include "Trajectories/trajectory31.inc"
#include "Trajectories/trajectory32.inc"
#include "Trajectories/trajectory33.inc"
#include "Trajectories/trajectory34.inc"
#include "Trajectories/trajectory35.inc"
#include "Trajectories/trajectory36.inc"
#include "Trajectories/trajectory37.inc"
#include "Trajectories/trajectory38.inc"
#include "Trajectories/trajectory39.inc"
#include "Trajectories/trajectory40.inc"
#include "Trajectories/trajectory41.inc"
#include "Trajectories/trajectory42.inc"


//Constants:
#declare Hbar = 1;
#declare Mass = 2;
#declare Pi = 3.14159;

#declare Beta = Hbar/(2*Mass);
#declare GroupVelocityX = 5;
#declare GroupVelocityY = 0;
#declare SlitDistance = 4;
#declare SigmaX = 1;
#declare SigmaY = 0.2;

//Time settings for animation:
#declare Start = 0;
#declare End = 2.4; //TODO needs to be set according to "totalTime" in Matlab
#declare Time = (End - Start)* clock; // Time is actual physical time, same unit as in
    Matlab, clock goes from 0 to 1
#declare NumberOfFrames = 481; //TODO needs to be set according to "Final_Frame" in topf.
    ini file
//#declare TimePerFrame = PeriodOfGroundState/NumberOfFrames;
#declare FrameNumber = clock * (NumberOfFrames-1) + 1;
// FrameNumber starts at 1, finishes at NumberOfFrames

#include "trajectoryLines.inc"
#include "probDensity.inc"



// Camera, light and background:
background {White}

//camera {CameraCenteredMedium}
camera {CameraCenteredMediumWIDE} //for a 16:9 aspect ratio

light_source {
  <40,50,40>
  color White
  //parallel
  shadowless
}

plane { z, 0
  pigment { checker White, Gray90 scale 1 }
}

// Coordinate axes (left-handed):
#declare ThicknessAxes = 0.007;
#declare LengthAxes = 1;
#declare ThicknessCone = 0.02;
#declare LengthCone= 0.04;
#declare ColorAxes = Gray40;

//cylinder { <0,0,0>, <2,0,0>, ThicknessAxes pigment {ColorAxes} }
//cone {<2,0,0>,ThicknessCone <2+LengthCone,0,0>, 0 pigment {ColorAxes} }
//cylinder { <0,0,0>, <0,LengthAxes,0>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,LengthAxes,0>, ThicknessCone <0,LengthAxes + LengthCone,0>, 0 pigment {
    ColorAxes} }
//cylinder { <0,0,0>, <0,0,1>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,0,LengthAxes>, ThicknessCone <0,0,(LengthAxes + LengthCone)>, 0 pigment {
    ColorAxes} }


// Slit:
#declare SlitSize = 4*SigmaY;
#declare WallThickness = 0.05;
#declare WallHeight = 3;
#declare WallLength = 12;
#declare WallColor = Gray20;

difference {
  box {<.5*WallThickness,.5*WallLength,.5*WallHeight>
```

```
    < -0.5*WallThickness, -.5*WallLength, -.5*WallHeight>
  }

  box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
    translate <0,.5*SlitDistance,0>
  }

  box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
    translate <0,-.5*SlitDistance,0>
  }
  pigment{ color WallColor}
  finish {phong 0.5}
}


// ProbDensityLine:
object {ProbDensityLine}

// Particles positive y side:
#declare ParticleRadius = 0.04;
DrawParticleFromMatlabValues1(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues2(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues3(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues4(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues5(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues6(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues7(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues8(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues9(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues10(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues11(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues12(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues13(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues14(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues15(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues16(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues17(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues18(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues19(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues20(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues21(FrameNumber,ParticleRadius)
// Particles negative y side:
DrawParticleFromMatlabValues22(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues23(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues24(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues25(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues26(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues27(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues28(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues29(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues30(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues31(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues32(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues33(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues34(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues35(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues36(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues37(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues38(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues39(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues40(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues41(FrameNumber,ParticleRadius)
DrawParticleFromMatlabValues42(FrameNumber,ParticleRadius)

// Trajectoy lines:
#declare RadiusTrajectoryLines = 0.01;
object {TrajectoryLines(RadiusTrajectoryLines)}
```

Listing B.20: doubleSlitTrajectoryLines.ini

```
+w1280
+h720

Input_File_Name=doubleSlitTrajectoryLines.pov
Output_File_Name=doubleSlitTrajectoryLines.png

Antialias=On
Antialias_Threshold=0.1
Antialias_Depth=2

Initial_Frame=1
Final_Frame=481; always set this according to "numberOfFrames" in Matlab code
Initial_Clock=0
Final_Clock=1

Subset_Start_Frame=1
Subset_End_Frame=481

Cyclic_Animation=off
Pause_when_Done=off
```

Listing B.21: doubleSlitHitsOnScreen.pov

```
// Bachelorarbeit Marlon Metzger, LMU Muenchen, October 2013

// In this scene there is no surface plot for the probability density
// Povray uses a left-handed coordinate system!!!
// Slits are situated on the y-axis at z==0

#include "colors.inc"
#include "gauss.inc"
#include "littleFunctions.inc"
#include "cameraPositions.inc"
#include "hitsOnScreenArray1.inc"

//Constants:
#declare Hbar = 1;
#declare Mass = 2;
#declare Pi = 3.14159;

#declare Beta = Hbar/(2*Mass);
#declare GroupVelocityX = 5;
#declare GroupVelocityY = 0;
#declare SlitDistance = 4;
#declare SigmaX = 1;
#declare SigmaY = 0.2;

//Time settings for animation:
#declare Start = 0;
#declare End = 2.4; //TODO needs to be set according to "totalTime" in Matlab
#declare Time = (End - Start)* clock; // Time is actual physical time, same unit as in
    Matlab, clock goes from 0 to 1
#declare NumberOfFrames = 481; //TODO needs to be set according to "Final_Frame" in topf.
    ini file
//#declare TimePerFrame = PeriodOfGroundState/NumberOfFrames;
#declare FrameNumber = clock * (NumberOfFrames-1) + 1;

#declare TimeForProbDensity = 2.4; // so that probability density curve stays at x==12
#include "probDensityAtFinalTime.inc" // for the blue function to stay at the screen


// Camera, light and background:
background {White}

//camera {CameraCenteredMedium}
camera {CameraCenteredMediumWIDE}

light_source {
  <40,50,40>
  color White
  //parallel
  shadowless
}

light_source {
  <-40,50,40>
  color White
  parallel
  shadowless
}

plane { z, 0
  pigment { checker White, Gray90 scale 1 }
}

// Coordinate axes (left-handed):
#declare ThicknessAxes = 0.007;
#declare LengthAxes = 1;
#declare ThicknessCone = 0.02;
#declare LengthCone= 0.04;
#declare ColorAxes = Gray40;

//cylinder { <0,0,0>, <2,0,0>, ThicknessAxes pigment {ColorAxes} }
//cone {<2,0,0>,ThicknessCone <2+LengthCone,0,0>, 0 pigment {ColorAxes} }
//cylinder { <0,0,0>, <0,LengthAxes,0>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,LengthAxes,0>, ThicknessCone <0,LengthAxes + LengthCone,0>, 0 pigment {
    ColorAxes} }
//cylinder { <0,0,0>, <0,0,1>, ThicknessAxes pigment {ColorAxes} }
//cone { <0,0,LengthAxes>, ThicknessCone <0,0,(LengthAxes + LengthCone)>, 0 pigment {
    ColorAxes} }

// Slit:
#declare SlitSize = 4*SigmaY;
#declare WallThickness = 0.05;
#declare WallHeight = 3;
#declare WallLength = 12;
#declare WallColor = Gray20;

difference {
  box {<.5*WallThickness,.5*WallLength,.5*WallHeight>
    < -0.5*WallThickness, -.5*WallLength, -.5*WallHeight>
  }
```

```
86
87    box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
88      translate <0,.5*SlitDistance,0>
89    }
90
91    box {<WallThickness,SlitSize/2,WallHeight><-WallThickness,-SlitSize/2,-WallHeight>
92      translate <0,-.5*SlitDistance,0>
93    }
94    pigment{ color WallColor}
95    finish {phong 0.5}
96  }
97
98  // Screen:
99  #declare ScreenWidth = 16;
100 #declare ScreenHeight = WallHeight;
101 #declare ScreenThickness = WallThickness;
102 #declare ScreenColor = White;
103
104 box {<.5*ScreenThickness,.5*ScreenWidth,.5*ScreenHeight>
105   < -0.5*ScreenThickness, -.5*ScreenWidth, -.5*ScreenHeight>
106   translate <12+0.5*ScreenThickness,0,0> //so that particles are not hidden inside the
        screen
107   //TODO must be set according to x values in HitsOnScreenArray1
108   pigment{ color ScreenColor}
109   finish {phong 0.5}
110 }
111
112 // ProbDensityLine:
113 //object {ProbDensityLine
114   //scale < 1, 1, pow((clock-0.1),3)/(0.9*0.9*0.9) >
115   //translate <0,0,-0.001>
116 //}
117 //stays at x = TimeForProbdensity*GroupVelocityX,
118 //because include file was exchanged with "probDensityAtFinalTime.inc" (at lines 37,38)
119
120
121 // Particles hitting the screen:
122 #declare ParticleOnScreenRadius = 0.03;
123 #declare ParticleOnScreen = sphere {
124   <0,0,0>, ParticleOnScreenRadius // used to be 0.025
125   texture{pigment{color Black}
126   finish {ambient 0.15 diffuse 0.85 phong 1}
127     }
128 }
129
130 #declare numberOfParticlesHittingScreen = 1000; //TODO needs to be set according to size
        of HitsOnScreenArray1
131 #declare ParticleIndex = 0;
132
133
134 #macro LetParticlesHitScreen(FrameNumber)
135   #local ParticleIndex = 0;
136   #while (ParticleIndex < (pow((clock-0.1),3) * numberOfParticlesHittingScreen
        /(0.9*0.9*0.9))) //particles apearing faster and faster, but slowly at the beginning
137     #local ArrayIndex = RandomPermutationArray1[ParticleIndex];
138     object {ParticleOnScreen translate HitsOnScreenArray1[ArrayIndex]}
139     #local ParticleIndex = ParticleIndex +1;
140   #end
141 #end
142
143 LetParticlesHitScreen(FrameNumber)
```

Listing B.22: doubleSlitHitsOnScreen.ini

```
1  +w1280
2  +h720
3
4  Input_File_Name=doubleSlitHitsOnScreen.pov
5  Output_File_Name=doubleSlitHitsOnScreen.png
6
7  Antialias=On
8
9  Antialias_Threshold=0.1
10 Antialias_Depth=2
11
12
13
14 Initial_Frame=1
15 Final_Frame=241; always set this according to "numberOfFrames" in Matlab code
16 Initial_Clock=0
17 Final_Clock=1
18
19 Subset_Start_Frame=1
20 Subset_End_Frame=241
21
22 Cyclic_Animation=off
23 Pause_when_Done=off
```

Listing B.23: cameraPositions.inc

```
// Camera positions

#declare Camera1 = camera {
  angle 45
  location <-7, 2, 6.3>
  sky <0,0,1>
  look_at <5,0,0>
  rotate <0,0,0>
}


#declare Camera2 = camera {
  angle 50
  location <5, 11,12>
  sky <0,0,1>
  look_at <5,0,0>
  rotate <0,0,0>
}

#declare CameraCentered = camera {
  angle 45
  location <-7, 0, 6.3>
  sky <0,0,1>
  look_at <5,0,0>
  rotate <0,0,0>
}

#declare CameraCenteredFar = camera {
  angle 30
  location <-12, 0, 10>
  sky <0,0,1>
  look_at <5,0,0>
  rotate <0,0,0>
}

#declare CameraCenteredMedium = camera {
  angle 37
  location <-10, 0, 8>
  sky <0,0,1>
  look_at <5,0,0>
  rotate <0,0,0>
}

#declare CameraCenteredMediumWIDE = camera {
  angle 45
  location <-10, 0, 8>
  sky <0,0,1>
  look_at <5,0,0>
  right <16/9,0,0>
  rotate <0,0,0>
}
```

Listing B.24: probDensity.inc

```
//Resulting probability density (resulting from the interference of the Gaussian
    wavepackets)
//Expression for "ProbDensity" has been found using Mathematica

//Constants:
#declare AlphaX = SigmaX*SigmaX; // SigmaX is Sigma of the wavepacket at t=0
#declare AlphaY = SigmaY*SigmaY;
#declare Beta = Hbar/(2*Mass); // for both x- and y-direction
#declare Delta = SlitDistance/2;
#declare V = GroupVelocityX;
#declare T = Time; //just abbreviated

//Numerator:
//(split into small bits, as the whole expression is so big)
#declare Num1 = function(x) {exp( - (-V*T+x)*(-V*T+x)*AlphaX / (2*(AlphaX*AlphaX+T*T*Beta
    *Beta)) )}
#declare Num2 = function(y) {exp( - AlphaY*(y-Delta)*(y-Delta) / (2*(AlphaY*AlphaY+T*T*
    Beta*Beta)) )}
#declare Num3 = function(y) {exp( - AlphaY*(y+Delta)*(y+Delta) / (2*(AlphaY*AlphaY+T*T*
    Beta*Beta)) )}
#declare Num4 = function(y) {exp(  AlphaY*(y-Delta)*(y-Delta) / (2*(AlphaY*AlphaY+T*T*
    Beta*Beta)) )}
#declare Num5 = function(y) {exp(  AlphaY*(y+Delta)*(y+Delta) / (2*(AlphaY*AlphaY+T*T*
    Beta*Beta)) )}
#declare Num6 = function(y) {2*exp( (y*y+Delta*Delta)*AlphaY / (2*(AlphaY*AlphaY+T*T*Beta
    *Beta)) )}
#declare NumCos = function(y) {cos( T*y*Beta*Delta / (AlphaY*AlphaY+T*T*Beta*Beta) )}

//Denominator:
#declare Denominator = sqrt( (AlphaX*AlphaX+T*T*Beta*Beta)*(AlphaY*AlphaY+T*T*Beta*Beta)
    );

//Whole expression for probability density:
#declare ProbDensity = function(x,y) {Num1(x)*Num2(y)*Num3(y)* ( Num4(y)+Num5(y)+Num6(y)*
```

```
     NumCos(y) ) / Denominator }


//Size factor for probability density plots:
#declare SizeFactor = 0.19; // used for the isosurface as well as for the line plot


#declare ProbDensitySurfaceTransparent = isosurface {
    function {SizeFactor*ProbDensity(x,y)-0.00001-z }
    contained_by { box { <-2,-10,.005> <16,10,20> } }
    open
    max_gradient 8
    pigment {color Blue transmit 0.3}
    finish {phong 1}
}

//Isosurface object for surface plot of the probability density:
#declare ProbDensitySurface = isosurface {
    function {SizeFactor*ProbDensity(x,y)-0.00001-z }
    contained_by { box { <-5+10*clock,-10,.005> <6+10*clock,10,15-13*clock> } } //
     container moves for faster rendering
    open
    max_gradient 4-2.5*clock
    // max_gradient depending on "clock" for faster rendering
    // should work fine for EndTime==2.4 and SizeFactor==0.19
    pigment {color Blue}
    finish {phong 1}
}


//Line plot for the probability density:
#declare PlotThickness = 0.06;
#declare PlotDot = sphere{<0,0,0>, PlotThickness/2
  texture{pigment{color Blue}
    finish {ambient 0.15 diffuse 0.85 phong 1}
    }
}

#declare PlotYStep = 0.01*PlotThickness/2;
#declare YPlot = SlitDistance/2; // can't start at 0, as then loop doesn't even start
#declare PlotThreshold = 0.005; // minimum value for density to be plotted
#declare ProbDensityLineForPositiveY = union {
  // from center of slit outwards:
  #while (ProbDensity(GroupVelocityX*Time,YPlot) > PlotThreshold)
    object { PlotDot translate <GroupVelocityX*Time,YPlot,SizeFactor*ProbDensity(
     GroupVelocityX*Time,YPlot)>}
    #declare YPlot = YPlot + PlotYStep;
  #end
  // from center of slit to center:
  #declare YPlot = SlitDistance/2;
  #while ((ProbDensity(GroupVelocityX*Time,YPlot) > PlotThreshold) & YPlot>0)
  //second condition (YPlot>0) necessary as soon as packets begin to overlap
    object { PlotDot translate <GroupVelocityX*Time,YPlot,SizeFactor*ProbDensity(
     GroupVelocityX*Time,YPlot)>}
    #declare YPlot = YPlot - PlotYStep; // note the minus
  #end
}

#declare ProbDensityLine = union {
  object {ProbDensityLineForPositiveY} //this is for one slit
  object {ProbDensityLineForPositiveY scale <1,-1,1>} //reflects the object to the
     negative y side
}
```

Listing B.25: drawTrajectories.inc

```
#declare Particle = sphere {
  <0,0,0>,1 // used to be 0.025
  texture{pigment{color Black}
  finish {ambient 0.15 diffuse 0.85 phong 1}
    }
}

#declare HeightOfParticles = 0;

#macro DrawParticleFromMatlabValues1(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab1[framenumber
     -1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues2(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab2[framenumber
     -1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues3(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab3[framenumber
     -1]
```

```
22      translate <0,0,HeightOfParticles>}
23  #end
24
25  #macro DrawParticleFromMatlabValues4(framenumber,ParticleRadius)
26    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab4[framenumber
        -1]
27      translate <0,0,HeightOfParticles>}
28  #end
29
30  #macro DrawParticleFromMatlabValues5(framenumber,ParticleRadius)
31    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab5[framenumber
        -1]
32      translate <0,0,HeightOfParticles>}
33  #end
34
35  #macro DrawParticleFromMatlabValues6(framenumber,ParticleRadius)
36    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab6[framenumber
        -1]
37      translate <0,0,HeightOfParticles>}
38  #end
39
40  #macro DrawParticleFromMatlabValues7(framenumber,ParticleRadius)
41    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab7[framenumber
        -1]
42      translate <0,0,HeightOfParticles>}
43  #end
44
45  #macro DrawParticleFromMatlabValues8(framenumber,ParticleRadius)
46    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab8[framenumber
        -1]
47      translate <0,0,HeightOfParticles>}
48  #end
49
50  #macro DrawParticleFromMatlabValues9(framenumber,ParticleRadius)
51    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab9[framenumber
        -1]
52      translate <0,0,HeightOfParticles>}
53  #end
54
55  #macro DrawParticleFromMatlabValues10(framenumber,ParticleRadius)
56    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab10[
        framenumber-1]
57      translate <0,0,HeightOfParticles>}
58  #end
59
60  #macro DrawParticleFromMatlabValues11(framenumber,ParticleRadius)
61    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab11[
        framenumber-1]
62      translate <0,0,HeightOfParticles>}
63  #end
64
65  #macro DrawParticleFromMatlabValues12(framenumber,ParticleRadius)
66    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab12[
        framenumber-1]
67      translate <0,0,HeightOfParticles>}
68  #end
69
70  #macro DrawParticleFromMatlabValues13(framenumber,ParticleRadius)
71    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab13[
        framenumber-1]
72      translate <0,0,HeightOfParticles>}
73  #end
74
75  #macro DrawParticleFromMatlabValues14(framenumber,ParticleRadius)
76    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab14[
        framenumber-1]
77      translate <0,0,HeightOfParticles>}
78  #end
79
80  #macro DrawParticleFromMatlabValues15(framenumber,ParticleRadius)
81    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab15[
        framenumber-1]
82      translate <0,0,HeightOfParticles>}
83  #end
84
85  #macro DrawParticleFromMatlabValues16(framenumber,ParticleRadius)
86    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab16[
        framenumber-1]
87      translate <0,0,HeightOfParticles>}
88  #end
89
90  #macro DrawParticleFromMatlabValues17(framenumber,ParticleRadius)
91    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab17[
        framenumber-1]
92      translate <0,0,HeightOfParticles>}
93  #end
94
95  #macro DrawParticleFromMatlabValues18(framenumber,ParticleRadius)
96    object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab18[
        framenumber-1]
97      translate <0,0,HeightOfParticles>}
```

```
#end

#macro DrawParticleFromMatlabValues19(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab19[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues20(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab20[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues21(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab21[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues22(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab22[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues23(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab23[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues24(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab24[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues25(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab25[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues26(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab26[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues27(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab27[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues28(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab28[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues29(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab29[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues30(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab30[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues31(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab31[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues32(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab32[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end

#macro DrawParticleFromMatlabValues33(framenumber,ParticleRadius)
  object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab33[
     framenumber-1]
    translate <0,0,HeightOfParticles>}
#end
```

```
174
175 #macro DrawParticleFromMatlabValues34(framenumber,ParticleRadius)
176   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab34[
        framenumber-1]
177     translate <0,0,HeightOfParticles>}
178 #end
179
180 #macro DrawParticleFromMatlabValues35(framenumber,ParticleRadius)
181   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab35[
        framenumber-1]
182     translate <0,0,HeightOfParticles>}
183 #end
184
185 #macro DrawParticleFromMatlabValues36(framenumber,ParticleRadius)
186   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab36[
        framenumber-1]
187     translate <0,0,HeightOfParticles>}
188 #end
189
190 #macro DrawParticleFromMatlabValues37(framenumber,ParticleRadius)
191   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab37[
        framenumber-1]
192     translate <0,0,HeightOfParticles>}
193 #end
194
195 #macro DrawParticleFromMatlabValues38(framenumber,ParticleRadius)
196   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab38[
        framenumber-1]
197     translate <0,0,HeightOfParticles>}
198 #end
199
200 #macro DrawParticleFromMatlabValues39(framenumber,ParticleRadius)
201   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab39[
        framenumber-1]
202     translate <0,0,HeightOfParticles>}
203 #end
204
205 #macro DrawParticleFromMatlabValues40(framenumber,ParticleRadius)
206   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab40[
        framenumber-1]
207     translate <0,0,HeightOfParticles>}
208 #end
209
210 #macro DrawParticleFromMatlabValues41(framenumber,ParticleRadius)
211   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab41[
        framenumber-1]
212     translate <0,0,HeightOfParticles>}
213 #end
214
215 #macro DrawParticleFromMatlabValues42(framenumber,ParticleRadius)
216   object {Particle scale ParticleRadius translate TrajectoryValuesFromMatlab42[
        framenumber-1]
217     translate <0,0,HeightOfParticles>}
218 #end
```

Listing B.26: trajectoryLines.inc

```
1  // This file contains functions leaving thin lines behind the particles. This makes the
       hole path visible that each particle has taken.
2
3  #macro TrajectoryLines(LineRadius)
4    union {
5      #local number = 1;
6      #while (number < FrameNumber)
7        DrawParticleFromMatlabValues1(number,LineRadius)
8        DrawParticleFromMatlabValues2(number,LineRadius)
9        DrawParticleFromMatlabValues3(number,LineRadius)
10       DrawParticleFromMatlabValues4(number,LineRadius)
11       DrawParticleFromMatlabValues5(number,LineRadius)
12       DrawParticleFromMatlabValues6(number,LineRadius)
13       DrawParticleFromMatlabValues7(number,LineRadius)
14       DrawParticleFromMatlabValues8(number,LineRadius)
15       DrawParticleFromMatlabValues9(number,LineRadius)
16       DrawParticleFromMatlabValues10(number,LineRadius)
17       DrawParticleFromMatlabValues11(number,LineRadius)
18       DrawParticleFromMatlabValues12(number,LineRadius)
19       DrawParticleFromMatlabValues13(number,LineRadius)
20       DrawParticleFromMatlabValues14(number,LineRadius)
21       DrawParticleFromMatlabValues15(number,LineRadius)
22       DrawParticleFromMatlabValues16(number,LineRadius)
23       DrawParticleFromMatlabValues17(number,LineRadius)
24       DrawParticleFromMatlabValues18(number,LineRadius)
25       DrawParticleFromMatlabValues19(number,LineRadius)
26       DrawParticleFromMatlabValues20(number,LineRadius)
27       DrawParticleFromMatlabValues21(number,LineRadius)
28       DrawParticleFromMatlabValues22(number,LineRadius)
29       DrawParticleFromMatlabValues23(number,LineRadius)
30       DrawParticleFromMatlabValues24(number,LineRadius)
31       DrawParticleFromMatlabValues25(number,LineRadius)
32       DrawParticleFromMatlabValues26(number,LineRadius)
```

```
      DrawParticleFromMatlabValues27(number,LineRadius)
      DrawParticleFromMatlabValues28(number,LineRadius)
      DrawParticleFromMatlabValues29(number,LineRadius)
      DrawParticleFromMatlabValues30(number,LineRadius)
      DrawParticleFromMatlabValues31(number,LineRadius)
      DrawParticleFromMatlabValues32(number,LineRadius)
      DrawParticleFromMatlabValues33(number,LineRadius)
      DrawParticleFromMatlabValues34(number,LineRadius)
      DrawParticleFromMatlabValues35(number,LineRadius)
      DrawParticleFromMatlabValues36(number,LineRadius)
      DrawParticleFromMatlabValues37(number,LineRadius)
      DrawParticleFromMatlabValues38(number,LineRadius)
      DrawParticleFromMatlabValues39(number,LineRadius)
      DrawParticleFromMatlabValues40(number,LineRadius)
      DrawParticleFromMatlabValues41(number,LineRadius)
      DrawParticleFromMatlabValues42(number,LineRadius)
      #local number = number +1;
    #end
  }
#end
```

Listing B.27: trajectoryLineOneParticle.inc

```
#macro TrajectoryLineOneParticle(LineRadius)
  union {
    #local number = 1;
    #while (number < FrameNumber)
      DrawParticleFromMatlabValues2(number,LineRadius) //choose particle here
      #local number = number +1;
    #end
  }
#end
```

Listing B.28: probDensityAtFinalTime.inc

```
// TODO NOTE: This is basically a copy of probDensity.inc, but time parameter T is set to
      a different value in this version.
// This version is for the "HitsOnScreen" video.


// Resulting probability density (resulting from the interference of the
// Gaussian wavepackets)

// Constants:
#declare AlphaX = SigmaX*SigmaX; // SigmaX is Sigma of |psiX|^2 at t=0
#declare AlphaY = SigmaY*SigmaY;
#declare Beta = Hbar/(2*Mass); // for both x- and y-direction
#declare Delta = SlitDistance/2;
#declare V = GroupVelocityX;
#declare T = TimeForProbDensity; //TODO NOTE: this is the important change

// Numerator:
#declare Num1 = function(x) {exp( - (-V*T+x)*(-V*T+x)*AlphaX / (2*(AlphaX*AlphaX+T*T*Beta
     *Beta)) )}
#declare Num2 = function(y) {exp( - AlphaY*(y-Delta)*(y-Delta) / (2*(AlphaY*AlphaY+T*T*
     Beta*Beta)) )}
#declare Num3 = function(y) {exp( - AlphaY*(y+Delta)*(y+Delta) / (2*(AlphaY*AlphaY+T*T*
     Beta*Beta)) )}
//#declare Num2 = function(y) {exp( - AlphaY*(y*y*Delta*Delta) / (AlphaY*AlphaY+T*T*Beta*
     Beta) )}
#declare Num4 = function(y) {exp(  AlphaY*(y-Delta)*(y-Delta) / (2*(AlphaY*AlphaY+T*T*
     Beta*Beta)) )}
#declare Num5 = function(y) {exp(  AlphaY*(y+Delta)*(y+Delta) / (2*(AlphaY*AlphaY+T*T*
     Beta*Beta)) )}
#declare Num6 = function(y) {2*exp( (y*y+Delta*Delta)*AlphaY / (2*(AlphaY*AlphaY+T*T*Beta
     *Beta)) )}
#declare NumCos = function(y) {cos( T*y*Beta*Delta / (AlphaY*AlphaY+T*T*Beta*Beta) )}

//Denominator:
#declare Denominator = sqrt( (AlphaX*AlphaX+T*T*Beta*Beta)*(AlphaY*AlphaY+T*T*Beta*Beta)
     );

//Whole expression for probability density:
#declare ProbDensity = function(x,y) {Num1(x)*Num2(y)*Num3(y)* ( Num4(y)+Num5(y)+Num6(y)*
     NumCos(y) ) / Denominator }


#declare SizeFactor = 0.19; // used in isosurface as well as in the line plot


#declare ProbDensitySurfaceTransparent = isosurface {
    function {SizeFactor*ProbDensity(x,y)-0.00001-z }
    contained_by { box { <-2,-10,.005> <16,10,20> } }
    open
    max_gradient 8
    pigment {color Blue transmit 0.3}
    finish {phong 1}
}

```

```
#declare ProbDensitySurface = isosurface {
    function {SizeFactor*ProbDensity(x,y)-0.00001-z }
    contained_by { box { <-5+10*clock,-10,.005> <6+10*clock,10,15-13*clock> } } //
     container moves for faster rendering
    open
    max_gradient 4-2.5*clock //for faster redering; should work fine for EndTime==2.4 and
      SizeFactor==0.19
    pigment {color Blue}
    finish {phong 1}
}


// LinePlot:
#declare PlotThickness = 0.06;
#declare PlotDot = sphere{<0,0,0>, PlotThickness/2
              texture{pigment{color Blue}
                  finish {ambient 0.15 diffuse 0.85 phong 1}
                }
}

#declare PlotYStep = 0.01*PlotThickness/2;
#declare YPlot = SlitDistance/2; // can't start at 0, as then loop doesn't even start
#declare PlotThreshold = 0.005; // minimum value for density to be plotted
#declare ProbDensityLineForPositiveY = union {
  // from center of slit outwards:
  #while (ProbDensity(GroupVelocityX*T,YPlot) > PlotThreshold)
    object { PlotDot translate <GroupVelocityX*T,YPlot,SizeFactor*ProbDensity(
     GroupVelocityX*T,YPlot)>}
    #declare YPlot = YPlot + PlotYStep;
  #end
  // from center of slit to center:
  #declare YPlot = SlitDistance/2;
  #while ((ProbDensity(GroupVelocityX*T,YPlot) > PlotThreshold) & YPlot>0)
  //second condition (YPlot>0) necessary as soon as packets begin to overlap
    object { PlotDot translate <GroupVelocityX*T,YPlot,SizeFactor*ProbDensity(
     GroupVelocityX*T,YPlot)>}
    #declare YPlot = YPlot - PlotYStep; // note the minus
  #end
}

#declare ProbDensityLine = union {
  object {ProbDensityLineForPositiveY} //this is for one slit
  object {ProbDensityLineForPositiveY scale <1,-1,1>} //reflects the object to the
     negative y side
}
```

B.3 Mathematica Notebook

Mathematica wurde genutzt, um einen vereinfachten analytischen Ausdruck für das Betragsquadrat der Wellenfunktion zu erhalten. Dafür wurde dieses Notebook geschrieben, an deren Ende die resultierende Formel zu finden ist:

```
ClearAll["Global`*"];
psiX = Exp[i (k x - ω t)] / Sqrt[α + i β t] Exp[-(x - vgroup t)^2 / (4 (α + i β t))];
psiY = 1 / Sqrt[αy + i β t] Exp[-(y - δ)^2 / (4 (αy + i β t))] + 1 / Sqrt[αy + i β t] Exp[-(y + δ)^2 / (4 (αy + i β t))];
Assumptions → {x, k, α, β, vgroup, ω , t , y, δ, αy} ∈ Reals;
psi = psiX * psiY;
psiAbsSquared = (Abs[psi])^2;
fullSim = FullSimplify[psiAbsSquared]
```

$$e^{-2\,\mathrm{Im}[k\,x-t\,\omega]-\frac{1}{2}\mathrm{Re}\left[\frac{(-t\,\mathrm{vgroup}+x)^2}{\alpha+i\,t\,\beta}\right]}\,\mathrm{Abs}\left[\frac{\left(e^{-\frac{(y-\delta)^2}{4\,(\alpha y+i\,t\,\beta)}}+e^{-\frac{(y+\delta)^2}{4\,(\alpha y+i\,t\,\beta)}}\right)^2}{(\alpha+i\,t\,\beta)\;(\alpha y+i\,t\,\beta)}\right]$$

```
compExpanded = ComplexExpand[fullSim]
```

$$\frac{e^{-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{2\,(\alpha^2+t^2\,\beta^2)}-\frac{\alpha y\,(y-\delta)^2}{2\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Cos}\left[\frac{t\,\beta\,(y-\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]^2}{\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}}+$$

$$\frac{2\,e^{-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{2\,(\alpha^2+t^2\,\beta^2)}-\frac{\alpha y\,(y-\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}-\frac{\alpha y\,(y+\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Cos}\left[\frac{t\,\beta\,(y-\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]\mathrm{Cos}\left[\frac{t\,\beta\,(y+\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]}{\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}}+$$

$$\frac{e^{-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{2\,(\alpha^2+t^2\,\beta^2)}-\frac{\alpha y\,(y+\delta)^2}{2\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Cos}\left[\frac{t\,\beta\,(y+\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]^2}{\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}}+\frac{e^{-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{2\,(\alpha^2+t^2\,\beta^2)}-\frac{\alpha y\,(y-\delta)^2}{2\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Sin}\left[\frac{t\,\beta\,(y-\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]^2}{\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}}+$$

$$\frac{2\,e^{-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{2\,(\alpha^2+t^2\,\beta^2)}-\frac{\alpha y\,(y-\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}-\frac{\alpha y\,(y+\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Sin}\left[\frac{t\,\beta\,(y-\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]\mathrm{Sin}\left[\frac{t\,\beta\,(y+\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]}{\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}}+$$

$$\frac{e^{-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{2\,(\alpha^2+t^2\,\beta^2)}-\frac{\alpha y\,(y+\delta)^2}{2\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Sin}\left[\frac{t\,\beta\,(y+\delta)^2}{4\,(\alpha y^2+t^2\,\beta^2)}\right]^2}{\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}}$$

```
simpExpanded = Simplify[compExpanded]
```

$$\left(e^{\frac{1}{2}\left(-\frac{(-t\,\mathrm{vgroup}+x)^2\,\alpha}{\alpha^2+t^2\,\beta^2}-\frac{\alpha y\,(y-\delta)^2}{\alpha y^2+t^2\,\beta^2}-\frac{\alpha y\,(y+\delta)^2}{\alpha y^2+t^2\,\beta^2}\right)}\left(e^{\frac{\alpha y\,(y-\delta)^2}{2\,(\alpha y^2+t^2\,\beta^2)}}+e^{\frac{\alpha y\,(y+\delta)^2}{2\,(\alpha y^2+t^2\,\beta^2)}}+2\,e^{\frac{\alpha y\,(y^2+\delta^2)}{2\,(\alpha y^2+t^2\,\beta^2)}}\,\mathrm{Cos}\left[\frac{t\,y\,\beta\,\delta}{\alpha y^2+t^2\,\beta^2}\right]\right)\right)\Big/\left(\sqrt{\alpha^2+t^2\,\beta^2}\;\sqrt{\alpha y^2+t^2\,\beta^2}\right)$$

Printed by Wolfram Mathematica Student Edition

Literaturverzeichnis

[1] Bronstein, Semendjajew, et al. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 7. Ausgabe, 2008.

[2] Detlef Dürr and Stefan Teufel. *Bohmian Mechanics, The Physics and Mathematics of Quantum Theory*. Springer, 2009.

[3] Torsten Fließbach. *Quantenmechanik, Lehrbuch zur Theoretischen Physik III*. Spektrum Akademischer Verlag, 5. Ausgabe, 2008.

[4] David J. Griffiths. *Quantenmechanik, Lehr- und Übungsbuch*. Pearson Education, 2. Ausgabe, 2012. Übersetzung von Carsten Heinisch.

[5] Oliver Passon. *Bohmsche Mechanik, Eine elementare Einführung in die deterministische Interpretation der Quantenmechanik*. Verlag Harri Deutsch, 2. Ausgabe, 2010.

[6] A. Tonomura et al. Demonstration of single-electron buildup of an interference pattern. *American Journal of Physics*, 57, February 1989.

Danksagung

Vielen Dank an Nicola Vona für all die hilfreichen Erklärungen, Tipps und das regelmäßige Feedback! Vielen Dank auch an Herrn Prof. Dr. Detlef Dürr, der mich darüber in Kenntnis gesetzt hat, dass es die Bohmsche Mechanik überhaupt gibt, und für die Möglichkeit an diesem schönen Thema zu arbeiten. Des weiteren bedanke ich mich herzlich mich bei Herrn Prof. Dr. Jochen Weller, der so freundlich war und sich bereit erklärt hat, in einigen Tagen die Abschlussprüfung als Vertreter der Fakultät für Physik zu übernehmen.

Erklärung

Hiermit erkläre ich, dass ich die vorgelegte Arbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 5.12.2013

_____

Marlon Metzger