Manfred Liebmann

Ein effizienter Algorithmus zur numerischen Simulation von zeitabhängigen Problemen aus der Quantenmechanik

Diplomarbeit

Vorgelegt am Institut für Theoretische Physik
Karl-Franzens-Universität
Graz

Betreut von Ao. Univ.-Prof. Dr. Bernd Thaller

Graz, März 1999

Gewidmet Anton und Christine

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Die numerische Behandlung von Problemen aus der Quantenmechanik bringt gegenüber der formalen, analytischen Vorgangsweise für realistische Problemstellungen wesentliche Vorteile. In einer numerischen Formulierung können Problemstellungen realisiert werden, die einer formalen Rechnung aufgrund ihrer Komplexität nicht mehr zugänglich sind. In eine Simulation eines zeitabhängigen Quantensystems können irreguläre Geometrien und komplizierte Potentialverteilungen ohne großen Aufwand integriert werden.

Die Schwierigkeit der numerischen Methode liegt für ein realistisches Problem in der Lösung des zugeordneten Gleichungssystems. Die Berechnung eines Gleichungssystems mit einigen Millionen Variablen ist ein technisch schwieriges Problem. Die Effizienz und Skalierbarkeit des verwendeten Algorithmus ist für die praktische Brauchbarkeit der numerischen Methode von entscheidender Bedeutung.

Unter diesem Gesichtspunkt stellt die vorliegende Arbeit den theoretischen Hintergrund und die praktische Umsetzung eines hoch effizienten Algorithmus zur Approximation der Zeitevolution eines quantenmechanischen Systems vor. Der Algorithmus beruht auf einer vom Autor vorgeschlagenen Verallgemeinerung eines fraktalen Approximationsschemas für Exponentialoperatoren von Suzuki [2]. Dieses neue Schema ermöglicht es, Simulationen mit einigen Millionen Gleichungen in angemessener Zeit zu bearbeiten.

Neben diesen numerischen Aspekten spielt die Visualisierung der Zeitentwicklung eines Zustandsvektors eine bedeutende Rolle. Gerade für große Systeme ist es wichtig, die berechneten Daten in überschaubarer Weise darzustellen. Verschiedene Visualisierungsmethoden und Farbtransformationen für reelle und komplexe Funktionen stehen in diesem Zusammenhang zur Diskussion.

Die Theorie zur fraktalen Approximation von Exponentialoperatoren wird im ersten Kapitel vorgestellt. Sie beinhaltet den Konstruktionssatz für fraktale Approximationen und den Symmetriesatz. Weitere Aussagen zur fraktalen Struktur der Approximationskoeffizienten und Betrachtungen zur Approximationsgüte werden diskutiert.

Die numerischen Algorithmen zur Behandlung der quantenmechanischen Bewegungsgleichungen werden im zweiten Kapitel konstruiert. Die Darstellung umfaßt die Schrödinger-, Pauli- und Dirac-Gleichung in zwei und drei Raumdimensionen. In den Algorithmen finden elektromagnetische Potentiale ebenso Berücksichtigung, wie irreguläre Simulationsgebiete.

Das Thema des dritten Kapitels ist die Visualisierung von Funktionen. Es werden verschiedene Farbabbildungen zur Darstellung von reellen und komplexen Funktionen vorgestellt. Die Problematik der Darstellung von zwei- und dreidimensionalen zeitabhängigen Funktionen wird ebenfalls erläutert.

Das vierte Kapitel umfaßt die technischen Aspekte einer numerischen Simulation. Die Integration der Algorithmen in ein objektorientiertes Programmsystem und die Einbindung des sogenannten QuantumKernel in das Mathematica-System wird diskutiert. Weiters wird das Mathematica-Interface zum Quantum-Kernel im Detail erklärt.

Beispiele zu Problemen aus der Quantenmechanik sind im letzten Kapitel angegeben. Es werden der Aharonov-Bohm-Effekt, die Bewegung in einer Kugelschale, die Streuung an einer magnetischen Scheibe, die Spinpräzession im magnetischen Feld, ein Bindungszustand im konstanten Magnetfeld und die Bewegung in einem kovarianten skalaren Potential behandelt. Alle Simulationen werden anhand ihres Mathematica-Notebooks erklärt und mit den diskutierten Darstellungsmethoden visualisiert.

Der Anhang umfaßt den Source-Code zum Simulationsprogramm, sowie die Implementierung des Mathematica-Package und die Notebooks für die verschiedenen Simulationen.

Danksagung

Mein besonderer Dank gilt meinem Betreuer Dr. Bernd Thaller für die Unterstützung bei der Erstellung der Arbeit, die eingehenden Diskussionen und ganz besonders die große kreative Freiheit in wesentlichen Belangen der Arbeit. Aber auch für sein persönliches Engagement die Arbeit finanziell zu unterstützen.

Mein herzlicher Dank gilt meinen Eltern, Anton und Christine Liebmann, für die großzügige finanzielle und moralische Unterstützung, die diese Arbeit in dem Umfang ermöglicht haben.

Mein Dank gilt auch meiner Schwester Bettina für die formale Korrektur der Diplomarbeit und die zahlreichen Verbesserungsvorschläge.

Kapitel 1

Fraktale Approximation

1.1 Cauchy Probleme

Die Gleichungen der Quantenmechanik sind von einem abstrakten Standpunkt aus gesehen Cauchy-Probleme. Diese lassen sich in der Form

$$\mathrm{i}\frac{\mathrm{d}}{\mathrm{d}t}\psi(t) = H\psi(t) \tag{1.1}$$

$$\psi(0) \in \mathcal{D}(H) \subset \mathfrak{H} \tag{1.2}$$

schreiben.

Der Operator H ist der Hamilton-Operator des quantenmechanischen Systems. Die Wellenfunktion $\psi(0) \in \mathcal{D}(H)$ beschreibt den Anfangszustand des Systems im Hilbertraum $\mathfrak{H}$. Die zeitliche Entwicklung des quantenmechanischen Systems kann für zeitunabhängige Operatoren mit Hilfe des Exponentialoperators ausgedrückt werden.

$$\psi(t) = \mathrm{e}^{-\mathrm{i}tH}\psi(0) \tag{1.3}$$

Die geschlossene Berechnung der Zeitentwicklung eines Zustandes ist nur in sehr speziellen Situationen möglich. Ein Beispiel ist die freie Zeitevolution eines Gaußpaketes in der nichtrelativistischen Quantenmechanik unter der Schrödinger-Gleichung [11].

Praktische Fragestellungen umfassen meist sehr viel kompliziertere Operatoren als den freien Schrödinger-Operator. Die Berücksichtigung von allgemeinen elektromagnetischen Feldern und komplizierten Geometrien führt auf Anfangswertprobleme, die nur mehr durch eine numerische Behandlung zugänglich sind.

1.1.1 Numerische Methoden

Die Struktur der allgemeinen Lösung (1.3) eines quantenmechanischen Problems legt eine numerische Approximation des Exponentialoperators nahe. Es stellen sich dabei zwei Probleme: Erstens die Art der Restriktion des im allgemeinen unendlichdimensionalen Hilbertraumes $\mathfrak{H}$ auf einen endlichdimensionalen Raum und damit verbunden die Restriktion der Operatoren und Wellenfunktionen des Quantensystems. Zweitens die Berechnung der Exponentialfunktion des Hamilton-Operators im restringierten Raum.

Die Einschränkung des Hilbertraumes $\mathfrak{H}$ wird mit Hilfe eines regelmäßigen Gitternetzes realisiert. Das heißt, das für die numerische Simulation betrachtete Gebiet wird in Quadrate oder Würfel zerlegt. Die Ecken bilden dann die Menge der Gitterpunkte. Eine Wellenfunktion im restringierten Hilbertraum ist auf dieser Punktemenge definiert. Die Restriktion von Operatoren erfolgt in ähnlicher Weise: Multiplikationsoperatoren werden, wie die Wellenfunktionen, auf dem Gitter definiert. Differentialoperatoren werden mit Hilfe von Differenzenschemata diskretisiert. Die genaue Vorgangsweise wird anhand der Schrödinger-Gleichung weiter unten diskutiert. Die skizzierte Methode ist in der Literatur unter dem Namen Finite-Differenzen-Methode (FDM) bekannt.

Für die Berechnung der Exponentialfunktion des Hamilton-Operators, der unter Anwendung der FD-Methode zu einem Matrix-Operator wird, stehen verschiedene Methoden zur Verfügung. Die folgenden Paragraphen sollen einen exemplarischen Überblick geben.

Crank-Nicholson-Verfahren Ein gutes Verfahren zur Berechnung der Zeitevolution in der Quantenmechanik ist das Crank-Nicholson-Verfahren. Eine der ersten Anwendungen beschreibt Goldberg [9]. Eine klassische Darstellung des Verfahrens findet sich zum Beispiel im Buch von Ames [7]. Die hier gewählte Darstellung soll, abweichend vom Standard, die Verbindung zum Exponentialoperator unterstreichen. Das CN-Verfahren beruht auf der Cayley-Transformierten des Hamilton-Operators, genauer $-tH/2$.

Definition 1.1. *Sei T ein selbstadjungierter Operator. Dann heißt*

$$C := (\mathrm{i} - T)(\mathrm{i} + T)^{-1} \tag{1.4}$$

Cayley-Transformierte *von T.*

Explizit ist die Cayley-Transformierte und somit die Approximation an den Exponentialoperator durch den Operator

$$Q(t) = (\mathrm{i} + tH/2)(\mathrm{i} - tH/2)^{-1} = (1 - \mathrm{i}tH/2)(1 + \mathrm{i}tH/2)^{-1} \tag{1.5}$$

gegeben. Die wesentliche Eigenschaft der Cayley-Transformierten eines selbstadjungierten Operators ist die Unitarität. Die Cayley-Transformation ermöglicht also die Definition einer *unitären* Approximation an den Evolutionsoperator. Die Unitarität der Approximation ist besonders in der Quantenmechanik von großer Bedeutung, da sie die Konsistenz der Wahrscheinlichkeitsinterpretation der Wellenfunktion sicherstellt. Die Norm der Wellenfunktion bleibt erhalten.

Für die Diskussion der Konvergenzordnung des CN-Verfahrens ist die Definition des Landau-Symbol o nützlich.

Definition 1.2. *Seien f, g Funktionen in einem Banach-Raum. Dann ist das* Landau-Symbol o *definiert durch*

$$f(t) = o(g(t)) \quad (t \to s) :\Leftrightarrow \lim_{t \to s} \frac{|f(t)|}{|g(t)|} = 0. \tag{1.6}$$

Die Konvergenzordnung der Approximation $Q(t)$ kann unter Verwendung der Neumann-Reihe aus der Potenzreihendarstellung abgelesen werden. Explizit

gilt:

$$\mathrm{e}^{-\mathrm{i}tH} - (1 - \mathrm{i}tH/2)(1 + \mathrm{i}tH/2)^{-1} = o(t^2) \quad (t \to 0) \tag{1.7}$$

Die Konvergenz des CN-Verfahrens ist von zweiter Ordnung.

Das CN-Verfahren besitzt jedoch einen Nachteil und das ist der implizite Charakter des Verfahrens. Das heißt, die Anwendung der Cayley-Transformierten auf eine Wellenfunktion und somit ein Zeitschritt erfordert eine Matrixinversion. Bei realistischen Problemen entsteht daraus ein relativ hoher Rechenaufwand. In den nächsten Paragraphen stehen daher explizite Verfahren im Zentrum des Interesses, die im wesentlichen nur aus Matrix-Vektor-Operationen aufgebaut werden können oder die Fourier-Transformation als Hilfsmittel verwenden.

Operator-Splitting-Methode Die Operator-Splitting-Methode ist ein allgemeiner Zugang zur Approximation des Exponentialoperators. Die Grundlage für diese Methode bildet die Trotter-Formel [1]. Explizit lautet die Formel

$$\lim_{n\to\infty} (\mathrm{e}^{A/n}\mathrm{e}^{B/n})^n = \mathrm{e}^{A+B}. \tag{1.8}$$

Der Nutzen der Formel liegt in der Möglichkeit einen komplizierten Exponentialoperator mit Hilfe von einfacheren zu approximieren. Die Konvergenzordnung der Trotter-Formel ist eins. Formal ausgedrückt:

$$\mathrm{e}^{t(A+B)} - \mathrm{e}^{tA}\mathrm{e}^{tB} = o(t) \quad (t \to 0) \tag{1.9}$$

Die Anwendung des Operator-Splitting auf den Schrödinger-Operator $-\Delta + V$ führt auf die Approximation

$$Q(t) = \mathrm{e}^{-\mathrm{i}tV/2}\mathrm{e}^{\mathrm{i}t\Delta}\mathrm{e}^{-\mathrm{i}tV/2}. \tag{1.10}$$

Die gewählte symmetrisierte Form der Zerlegung verbessert die Konvergenzordnung auf zwei. Das Hauptproblem für die praktische Berechnung der Approximation liegt in der Exponentialfunktion des Laplace-Operators, das Potential bereitet als Multiplikationsoperator keine Schwierigkeiten. Eine Variante zur Berechnung ist die Verwendung der Fourier-Transformation.

Definition 1.3. *Die Fourier-Transformation $\mathcal{F}$ im $\mathbb{R}^n$ mit $n \in \mathbb{N}$ ist definiert durch*

$$(\mathcal{F}\psi)(\mathbf{p}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} \mathrm{e}^{-\mathrm{i}\mathbf{p}\mathbf{x}}\psi(\mathbf{x})\mathrm{d}^n\mathbf{x}. \tag{1.11}$$

Die Anwendung der Fourier-Transformation führt den Laplace-Operator in einen Multiplikationsoperator im Impulsraum über. Die Berechnung der Exponentialfunktion ist in diesem Rahmen einfach.

$$\mathrm{e}^{\mathrm{i}t\Delta} = \mathcal{F}^{-1}\mathrm{e}^{-\mathrm{i}tp^2}\mathcal{F} \tag{1.12}$$

Eine frühe Anwendung dieser Spektral-Methode beschreibt Feit [10]. Bei der praktischen, numerischen Realisierung des beschriebenen Verfahrens spielt die Fast-Fourier-Transformation (FFT) die entscheidende Rolle. Der Rechenaufwand der Transformation bei der Problemgröße N liegt bei $N \log N$.

Trotter-Suzuki-Schema Eine von Suzuki [3] vorgeschlagene Verallgemeinerung der Trotter-Formel ermöglicht es, Approximationen in *beliebiger* Ordnung an den Exponentialoperator in systematischer Weise zu konstruieren. Das Trotter-Suzuki-Schema verwendet einen Satz von rekursiv definierten komplexen Zahlen zur Konstruktion einer Approximation an den Exponentialoperator.

$$\lim_{n\to\infty} (\mathrm{e}^{z_1 A/n}\mathrm{e}^{z_1 B/n}\mathrm{e}^{z_2 A/n}\mathrm{e}^{z_2 B/n}\cdots\mathrm{e}^{z_N A/n}\mathrm{e}^{z_N B/n})^n = \mathrm{e}^{A+B} \tag{1.13}$$

Die Zahlen $z_i \in \mathbb{C},\ 1 \leq i \leq N \in \mathbb{N}$ sind in der komplexen Ebene nach einem fraktalen Muster verteilt. Das Schema wird daher auch als *fraktale* Zerlegung [2] des Exponentialoperators bezeichnet. Eine genaue Analyse des Verfahrens wird im Konstruktionssatz für fraktale Approximationen 1.1 angegeben. Die Konvergenzordnung des Verfahrens $m \in \mathbb{N}$ hängt nur von der Wahl der komplexen Koeffizienten ab und ist unabhängig von der Zerlegung des Ausgangsoperators.

$$\mathrm{e}^{t(A+B)} - \mathrm{e}^{z_1 tA}\mathrm{e}^{z_1 tB}\mathrm{e}^{z_2 tA}\mathrm{e}^{z_2 tB}\cdots\mathrm{e}^{z_N tA}\mathrm{e}^{z_N tB} = o(t^m) \quad (t \to 0) \tag{1.14}$$

Die Anzahl N der Koeffizienten nimmt jedoch mit der Konvergenzordnung exponentiell zu. Die Konstruktion einer unitären Approximation kann im Kontext der Quantenmechanik durch eine Zerlegung des Hamilton-Operators in Hermitesche Anteile verwirklicht werden.

Das Trotter-Suzuki-Schema ist in seinen Grundzügen eine Operator-Splitting-Methode. Daher kann die bereits diskutierte Spektral-Methode auch hier angewendet werden. Es ist aber auch möglich, eine Fourier-Transformation zu vermeiden. Durch geschickte Zerlegung des Laplace-Operators kann der zugehörige Exponentialoperator auch direkt berechnet werden. Eine Anwendung dieser Methode auf die Schrödinger-Gleichung beschreibt De Raedt [4].

Verallgemeinertes Trotter-Suzuki-Schema Die in den letzten beiden Paragraphen beschriebenen Verfahren verwenden Approximationen an den Exponentialoperator, die ihrerseits aus einem Produkt von Exponentialoperatoren aufgebaut sind. Eine vom Autor vorgeschlagene Verallgemeinerung ermöglicht es, eine Approximation an den Exponentialoperator mit Hilfe einer analytischen Operatorfunktion $Q(t)$ aufzubauen. Diese Operatorfunktion unterliegt nur der Einschränkung, eine Approximation erster Ordnung an den Exponentialoperator zu sein.

$$\mathrm{e}^{t(A+B)} - Q(t) = o(t) \quad (t \to 0) \tag{1.15}$$

Eine Approximation an den Exponentialoperator kann aus einem Produkt der Operatorfunktion aufgebaut werden.

$$\lim_{n\to\infty} (Q(z_1/n)Q(z_2/n)\cdots Q(z_N/n))^n = \mathrm{e}^{A+B} \tag{1.16}$$

Die Koeffizienten der Approximation $z_1, z_2, \ldots, z_N \in \mathbb{C}$ stammen aus dem von Suzuki vorgeschlagenen Rekursionsschema. Wie im Konstruktionssatz 1.1 bewiesen wird, hängt die Konvergenzordnung $m \in \mathbb{N}$ des Verfahrens nur von der Wahl der komplexen Koeffizienten ab. Sie ist unabhängig von der Wahl der analytischen Operatorfunktion.

$$\mathrm{e}^{t(A+B)} - Q(z_1 t)Q(z_2 t)\cdots Q(z_N t) = o(t^m) \quad (t \to 0) \tag{1.17}$$

Einige Beispiele für die Operatorfunktion $Q(t)$ zur Approximation des Exponentialoperators $\mathrm{e}^{t(A+B)}$ sind:

$$Q(t) = \mathrm{e}^{tA}\mathrm{e}^{tB} \tag{1.18}$$

$$Q(t) = (1 + tA)\mathrm{e}^{tB} \tag{1.19}$$

$$Q(t) = 1 + t(A + B) \tag{1.20}$$

Die Definition in der ersten Zeile führt auf das ursprüngliche Trotter-Suzuki-Schema. Die zweite Definition ist eine Variation, die zum Beispiel für den Schrödinger-Operator $-\Delta + V$ eingesetzt werden kann. Die Definition in der letzten Zeile, eine Approximation durch eine lineare Operatorfunktion, wird später als Basis für die Konzeption der Simulationsalgorithmen dienen. Obwohl die daraus konstruierten Approximationen *nicht* unitär sind, wirkt sich diese Schwäche in der numerischen Rechnung nicht negativ aus. Eine genaue Darstellung dieser Problematik erfolgt später anhand einiger Simulationen zur Schrödinger-, Pauli- und Dirac-Theorie.

Eine triviale, ebenfalls nicht unitäre Approximation des Exponentialoperators ist die Potenzreihenentwicklung.

$$\mathrm{e}^{tA} - \sum_{n=0}^{m} \frac{(tA)^n}{n!} = o(t^m) \quad (t \to 0) \tag{1.21}$$

Obwohl theoretisch mit der Potenzreihe sehr einfach Approximationen in beliebiger Ordnung konstruiert werden können, ist diese Methode für numerische Simulationen nur sehr eingeschränkt einsetzbar. Die fehlende Unitarität der Approximation wirkt sich sehr negativ auf die numerische Stabilität der Methode aus. In der Praxis müssen die Zeitschritte sehr klein gewählt werden, um die numerische Stabilität über einen längeren Zeitraum zu erhalten. Ein empirischer Vergleich der Potenzreihenmethode mit der hier skizzierten fraktalen Methode zeigt, daß die letztere, bei gleichem Rechenaufwand, wesentlich größere Zeitschritte erlaubt, ohne die Stabilität zu beeinträchtigen. Die Zeitschritte können zumindest um einen Faktor 100 größer gewählt werden, als bei der Potenzreihenmethode.

Der Rechenaufwand der fraktalen Zerlegung des Exponentialoperators mit Hilfe einer linearen Operatorfunktion ist proportional zur Problemgröße N. Zur Berechnung einer Approximation vierter Ordnung werden im wesentlichen nur acht Matrix-Vektor-Multiplikationen benötigt. Die aus der Diskretisierung der Schrödinger- oder Dirac-Gleichung stammenden Matrizen sind nur dünn besetzt. Daher ist der Aufwand für eine Matrix-Vektor-Multiplikation proportional zu N. Eine volle Matrix-Vektor Multiplikation erfordert hingegen N^2 Operationen.

1.2 Konstruktionssatz

Der Konstruktionssatz bildet die Grundlage für die fraktale Zerlegung des Exponentialoperators. Die Begriffe *fraktale Zerlegung* und *fraktale Approximation* werden in Folge synonym verwendet und beziehen sich auf das vom Autor formulierte verallgemeinerte Trotter-Suzuki-Schema. Der Konstruktionssatz beschreibt ein rekursives Schema zum Aufbau einer Approximation der Ordnung

$m \in \mathbb{N}$ an den Exponentialoperator. Der Satz wird im Rahmen einer Banach-Algebra formuliert und bewiesen. Grundlegende Eigenschaften von Banach-Räumen und Banach-Algebren können zum Beispiel in [12] nachgelesen werden.

Satz 1.1. *Sei $\mathfrak{B}$ eine Banach-Algebra und $A \in \mathfrak{B}$ ein Operator. Sei $Q_1(t) \subset \mathfrak{B}$ eine analytische Operatorfunktion und Approximation erster Ordnung an den Exponentialoperator* $\exp(tA)$

$$\exp(tA) - Q_1(t) = o(t) \quad (t \to 0) \tag{1.22}$$

und $r \geq 2$ eine feste natürliche Zahl. Dann ist eine analytische Approximation der Ordnung $m \geq 2$ an den Exponentialoperator gegeben durch

$$Q_m(t) = \prod_{j=1}^{r} Q_{m-1}(p_{m,j}t). \tag{1.23}$$

Wobei die komplexen Koeffizienten $p_{m,j}$ mit $1 \leq j \leq r$ die Bedingungsgleichungen

$$\sum_{j=1}^{r} p_{m,j} = 1, \quad \sum_{j=1}^{r} p_{m,j}^{m} = 0 \tag{1.24}$$

erfüllen.

Beweis. Induktion. $Q_1(t)$ ist nach Voraussetzung des Satzes eine analytische Approximation erster Ordnung an den Exponentialoperator. Schluß von $m - 1 \to m$: Nach Induktionsvoraussetzung ist $Q_{m-1}(t)$ eine analytische Approximation der Ordnung $m - 1$.

$$\exp(tA) - Q_{m-1}(t) = o(t^{m-1}) \quad (t \to 0) \tag{1.25}$$

Aufgrund der Analytizität der Approximation gibt es einen Operator R_{m-1} mit

$$\exp(tA) - Q_{m-1}(t) = t^m R_{m-1} + o(t^m) \quad (t \to 0). \tag{1.26}$$

Damit gilt für die Approximation m-ter Ordnung:

$$Q_m(t) = \prod_{j=1}^{r} Q_{m-1}(p_{m,j}t) \tag{1.27}$$

$$= \prod_{j=1}^{r} (\exp(p_{m,j}tA) - (p_{m,j}t)^m R_{m-1}) + o(t^m) \tag{1.28}$$

Ausmultiplizieren des Produktes führt unter Berücksichtigung der Potenzreihendarstellung des Exponentialoperators auf

$$= \prod_{j=1}^{r} \exp(p_{m,j}tA) - \sum_{j=1}^{r} (p_{m,j}t)^m R_{m-1} \prod_{\substack{k=1 \\ k \neq j}}^{r} \exp(p_{m,k}tA) + o(t^m) \tag{1.29}$$

$$= \exp(tA \sum_{j=1}^{r} p_{m,j}) - t^m R_{m-1} \sum_{j=1}^{r} p_{m,j}^{m} + o(t^m). \tag{1.30}$$

Einsetzen der Bedingungsgleichungen (1.24) führt auf das gesuchte Resultat

$$\exp(tA) - Q_m(t) = o(t^m) \quad (t \to 0) \tag{1.31}$$

und $Q_m(t)$ ist als endliches Produkt von analytischen Approximationen analytisch. □

Definition 1.4. *Die im Satz 1.1 definierte Operatorfunktion $Q_m(t)$ heißt fraktale Approximation der Ordnung m an den Exponentialoperator* $\exp(tA)$.

Die natürliche Zahl r im Konstruktionssatz 1.1 bestimmt die Komplexität der Approximation. Es ist sinnvoll r möglichst klein zu halten, da die Anzahl der für eine Approximation m-ter Ordnung notwendigen Koeffizienten gleich r^{m-1} ist. In den folgenden Abschnitten werden Approximationen mit der Komplexität $r = 3$ und $r = 2$ diskutiert.

1.3 Symmetriesatz

Die Komplexität der fraktalen Approximation hängt von der Wahl des Parameters $r \in \mathbb{N}$ im Konstruktionssatz 1.1 ab. Für $r = 2$ sind die Bedingungsgleichungen (1.24) ausreichend, um die Koeffizienten $p_{m,j},\ 1 \le j \le r$ bis auf eine triviale Freiheit eindeutig zu bestimmen. Für die Wahl $r = 3$ besteht die Freiheit eine weitere Bedingungsgleichung für die Koeffizienten anzugeben. Diese Freiheit kann für die Symmetrieforderung

$$p_{m,1} = p_{m,3} \tag{1.32}$$

ausgenutzt werden.

Fraktale Approximationen der Komplexität $r = 3$ mit der zusätzlichen Bedingungsgleichung (1.32) im Konstruktionssatz und einer Operatorfunktion mit der Symmetrie

$$Q_1(t)Q_1(-t) = 1, \quad t \in \mathbb{C} \tag{1.33}$$

erfüllen den Symmetriesatz. Der Symmetriesatz liefert eine Aussage über die verbesserte Konvergenzordnung von ungeraden, fraktalen Approximationen.

Satz 1.2. *Sei $\mathfrak{B}$ eine Banach-Algebra und $A \in \mathfrak{B}$ ein Operator. Sei $Q_1(t) \subset \mathfrak{B}$ eine analytische Operatorfunktion und Approximation erster Ordnung an den Exponentialoperator* $\exp(tA)$ *mit der Symmetrie*

$$Q_1(t)Q_1(-t) = 1, \quad t \in \mathbb{C}. \tag{1.34}$$

und sei $p_{m,1} = p_{m,3}$ eine zusätzliche Bedingungsgleichung an die Koeffizienten im Konstruktionssatz 1.1 der Komplexität $r = 3$. Dann besitzt die fraktale Approximation $Q_m(t)$ der Ordnung $m \in \mathbb{N}$ die Symmetrie

$$Q_m(t)Q_m(-t) = 1, \quad t \in \mathbb{C} \tag{1.35}$$

und die Konvergenzordnung der fraktalen Approximation $Q_{2k-1}(t),\ k \in \mathbb{N}$ ist bereits $2k$.

$$\exp(tA) - Q_{2k-1}(t) = o(t^{2k}) \quad (t \to 0) \tag{1.36}$$

Beweis. Zu zeigen: Die fraktale Approximation $Q_m(t)$ besitzt die Symmetrieeigenschaft

$$Q_m(t)Q_m(-t) = 1, \quad t \in \mathbb{C}. \tag{1.37}$$

Beweis durch Induktion. $Q_1(t)$ ist nach Voraussetzung des Satzes eine symmetrische Approximation. Schluß von $m-1 \to m$: Nach Voraussetzung ist $Q_{m-1}(t)$ symmetrisch, also gilt unter Verwendung des Konstruktionssatzes 1.1 mit der Komplexität $r = 3$

$$\begin{aligned} Q_m(t)Q_m(-t) &= Q_{m-1}(p_{m,1}t)Q_{m-1}(p_{m,2}t)Q_{m-1}(p_{m,3}t) && (1.38)\\ &\quad \times Q_{m-1}(-p_{m,1}t)Q_{m-1}(-p_{m,2}t)Q_{m-1}(-p_{m,3}t). && (1.39) \end{aligned}$$

Das Resultat $Q_m(t)Q_m(-t) = 1$ folgt dann unter Beachtung der Gleichung $p_{m,1} = p_{m,3}$ aus den Relationen

$$\begin{aligned} &Q_{m-1}(p_{m,3}t)Q_{m-1}(-p_{m,1}t) = Q_{m-1}(p_{m,3}t)Q_{m-1}(-p_{m,3}t) = 1 && (1.40)\\ &Q_{m-1}(p_{m,2}t)Q_{m-1}(-p_{m,2}t) = 1 && (1.41)\\ &Q_{m-1}(p_{m,1}t)Q_{m-1}(-p_{m,3}t) = Q_{m-1}(p_{m,1}t)Q_{m-1}(-p_{m,1}t) = 1. && (1.42) \end{aligned}$$

Noch zu zeigen: Die fraktale Approximation $Q_{2k-1}(t),\ k \in \mathbb{N}$ besitzt die Konvergenzordnung $2k$.

$$\exp(tA) - Q_{2k-1}(t) = o(t^{2k}) \quad (t \to 0) \tag{1.43}$$

Nach dem Konstruktionssatz ist Q_{2k-1} eine Approximation der Ordnung $2k-1$ an den Exponentialoperator. Aufgrund der Analytizität der Approximation gibt es einen Operator R_{2k-1} mit

$$\exp(tA) - Q_{2k-1}(t) = t^{2k}R_{2k-1} + o(t^{2k}) \quad (t \to 0). \tag{1.44}$$

Aus der Symmetrieeigenschaft der Approximation folgt

$$\begin{aligned} 1 &= Q_{2k-1}(t)Q_{2k-1}(-t) && (1.45)\\ &= (\exp(tA) - t^{2k}R_{2k-1})(\exp(-tA) - (-t)^{2k}R_{2k-1}) + o(t^{2k}) && (1.46)\\ &= 1 - 2t^{2k}R_{2k-1} + o(t^{2k}). && (1.47) \end{aligned}$$

Äquivalent dazu ist die Gleichung

$$t^{2k}R_{2k-1} = o(t^{2k}). \tag{1.48}$$

Aus der Definition des Landau-Symbols 1.2 folgt unmittelbar $R_{2k-1} = 0$ und damit die Behauptung. □

1.4 Symmetrische Approximationen

Eine interessante Anwendungsmöglichkeit des Konstruktionssatzes 1.1 und des Symmetriesatzes 1.2 bietet das Crank-Nicholson-Verfahren. Aus der Approximation

$$Q_2(t) = (1 - \mathrm{i}tH/2)(1 + \mathrm{i}tH/2)^{-1} \tag{1.49}$$

kann mit Hilfe des Konstruktionssatzes eine Approximation dritter Ordnung gewonnen werden. Hinsichtlich der Anwendung des Symmetriesatzes wird die Komplexität der fraktalen Approximation $r = 3$ gewählt und die zusätzliche Bedingungsgleichung $p_{m,1} = p_{m,3}$ eingeführt. Die Gleichungen für die Koeffizienten unter Berücksichtigung der Bedingungsgleichungen (1.24) lauten:

$$p_{m,1} + p_{m,2} + p_{m,3} = 1 \tag{1.50}$$
$$p^m_{m,1} + p^m_{m,2} + p^m_{m,3} = 0 \tag{1.51}$$
$$p_{m,1} = p_{m,3} \tag{1.52}$$

Für die gesuchte Approximation dritter Ordnung sind nur die Koeffizienten mit $m = 3$ interessant. Diese ergeben sich nach kurzer Rechnung zu

$$\alpha = p_{3,1} = p_{3,3} = \frac{1}{2+w} \tag{1.53}$$
$$\beta = p_{3,2} = \frac{w}{2+w} \tag{1.54}$$
$$w = \sqrt[3]{2}\,(1 + \mathrm{i}\sqrt{3})/2. \tag{1.55}$$

Das verbesserte CN-Verfahren lautet damit:

$$Q_3(t) = Q_2(\alpha t)Q_2(\beta t)Q_2(\alpha t) \tag{1.56}$$

Nachdem aber alle Voraussetzungen für den Symmetriesatz erfüllt sind, gilt $Q_3(t) = Q_4(t)$. Das konstruierte Verfahren ist bis zur vierten Ordnung genau.

$$Q_4(t) = (1 - \mathrm{i}\alpha tH/2)(1 + \mathrm{i}\alpha tH/2)^{-1} \tag{1.57}$$
$$\times\,(1 - \mathrm{i}\beta tH/2)(1 + \mathrm{i}\beta tH/2)^{-1} \tag{1.58}$$
$$\times\,(1 - \mathrm{i}\alpha tH/2)(1 + \mathrm{i}\alpha tH/2)^{-1} \tag{1.59}$$

Der Konstruktionssatz und der Symmetriesatz erlauben es auf einfache Weise, die Konvergenzordnung von bekannten Verfahren zu verbessern. In analoger Weise kann die in der Einleitung beschriebene Spektral-Methode verbessert werden. Der Ausgangspunkt der Methode ist die Approximation

$$Q_2(t) = \mathrm{e}^{-\mathrm{i}tV/2}\mathrm{e}^{\mathrm{i}t\Delta}\mathrm{e}^{-\mathrm{i}tV/2}. \tag{1.60}$$

Die Approximation besitzt die Symmetrieeigenschaft (1.34), also kann wie beim Crank-Nicholson-Verfahren vorgegangen werden.

$$Q_4(t) = \mathrm{e}^{-\mathrm{i}\alpha tV/2}\mathrm{e}^{\mathrm{i}\alpha t\Delta}\mathrm{e}^{-\mathrm{i}(\alpha+\beta)tV/2}\mathrm{e}^{\mathrm{i}\beta t\Delta}\mathrm{e}^{-\mathrm{i}(\beta+\alpha)tV/2}\mathrm{e}^{\mathrm{i}\alpha t\Delta}\mathrm{e}^{-\mathrm{i}\alpha tV/2} \tag{1.61}$$

Die resultierende Spektral-Methode ist eine Approximation vierter Ordnung.

Wie diese beiden Beispiele veranschaulichen, können mit Hilfe der fraktalen Approximation bestehende Verfahren in sehr einfacher Weise verbessert werden. Die Struktur des vorliegenden Verfahrens bleibt im wesentlichen erhalten, was einen Vorteil bei der Implementierung der Methode darstellt.

1.5 Fraktale Strukturen

Die Koeffizienten $p_{m,j},\ 1 \le j \le r$ aus dem Konstruktionssatz 1.1 bestimmen die Struktur der fraktalen Approximation. Für $r = 2$ sind die Koeffizienten bereits

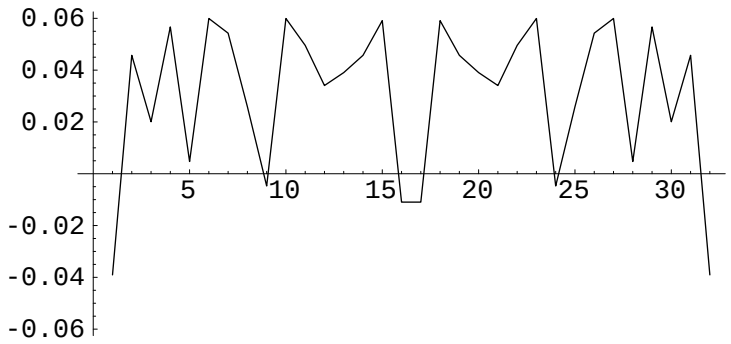

Abbildung 1.1: Realteil von $(z_{6,n})_{n=1}^{32}$

durch die Bedingungsgleichungen (1.24) festgelegt. Das Ziel dieses Abschnittes ist die explizite Berechnung dieser Koeffizienten und die Angabe einer praktischen Formel für die Berechnung einer fraktalen Approximation der Ordnung m.

Im Fall $r = 2$ reduzieren sich die Bedingungsgleichungen (1.24) zu

$$p_{m,1} + p_{m,2} = 1, \quad p_{m,1}^m + p_{m,2}^m = 0, \quad m \geq 2. \tag{1.62}$$

Die Lösungen des Gleichungssystems sind durch

$$p_{m,1} = \frac{1}{1 + \mathrm{e}^{-\mathrm{i}\pi/m}}, \quad p_{m,2} = \frac{1}{1 + \mathrm{e}^{\mathrm{i}\pi/m}}, \tag{1.63}$$

gegeben oder äquivalent dazu

$$p_m = \frac{1}{2} + \frac{\mathrm{i}}{2}\tan(\pi/2m), \quad p_{m,1} = p_m, \quad p_{m,2} = \bar{p}_m. \tag{1.64}$$

Die Koeffizienten der Zerlegung sind komplex und zueinander konjugiert. Dies ermöglicht eine einfache Darstellung der fraktalen Struktur der Zerlegung. Die Anwendung der Rekursionsvorschrift (1.23) mit der Operatorfunktion $Q_1(t)$ führt auf ein Schema der Art

$$Q_2(t) = Q_1(p_2 t)Q_1(\bar{p}_2 t) \tag{1.65}$$

$$Q_3(t) = Q_1(p_3 p_2 t)Q_1(p_3 \bar{p}_2 t)Q_1(\bar{p}_3 p_2 t)Q_1(\bar{p}_3 \bar{p}_2 t) \tag{1.66}$$

$$\ldots$$

$$Q_m(t) = Q_1(p_m \cdots p_3 p_2 t)Q_1(p_m \cdots p_3 \bar{p}_2 t) \cdots Q_1(\bar{p}_m \cdots \bar{p}_3 \bar{p}_2 t). \tag{1.67}$$

Allgemein kann eine fraktale Approximation m-ter Ordnung in folgender Weise dargestellt werden:

$$Q_m(t) = \prod_{n=1}^{2^{m-1}} Q_1(z_{m,n} t) \tag{1.68}$$

$$z_{m,n} = \prod_{k=2}^{m} \frac{\mathrm{e}^{-\mathrm{i}\pi\nu_k/k}}{1 + \mathrm{e}^{-\mathrm{i}\pi/k}}, \quad n = 1 + \sum_{l=2}^{m} \nu_l 2^{l-2} \tag{1.69}$$

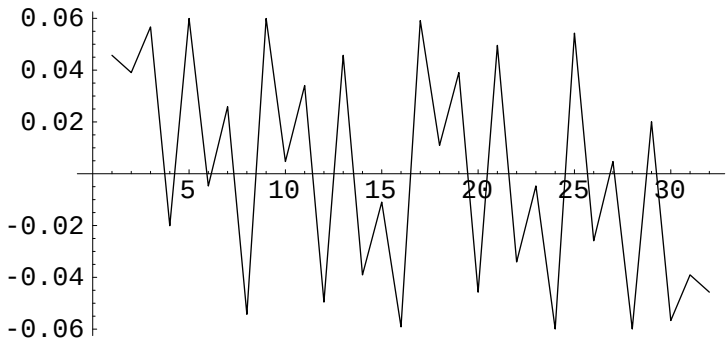

Abbildung 1.2: Imaginärteil von $(z_{6,n})_{n=1}^{32}$

Diese Formel korrespondiert mit der in der Einleitung dargestellten Zerlegung des Exponentialoperators. Die Anzahl der Koeffizienten ist abhängig von der Ordnung der Approximation 2^{m-1}. Der Realteil und der Imaginärteil der Folge $(z_{m,n})_{n=1}^{2^{m-1}}$ besitzen die fraktale Struktur, die dem Verfahren seinen Namen gibt. Beispiele für $m = 6$ sind in den Abbildungen zu sehen.

Eine Aussage über die Beschränktheit der Koeffizienten $z_{m,n}$ wird im folgenden Satz bewiesen.

Satz 1.3. *Sei $m \geq 2$ eine natürliche Zahl und $1 \leq n \leq 2^{m-1}$ und*

$$z_{m,n} = \prod_{k=2}^{m} \frac{\mathrm{e}^{-\mathrm{i}\pi\nu_k/k}}{1+\mathrm{e}^{-\mathrm{i}\pi/k}}, \quad n = 1 + \sum_{l=2}^{m} \nu_l 2^{l-2}. \tag{1.70}$$

Dann gilt

$$|z_{m,n}| = \prod_{k=2}^{m} \frac{1}{2\cos(\pi/2k)} < 3(2^{-m+1}). \tag{1.71}$$

Beweis. Die Gleichung folgt unmittelbar aus der Definition. Es gilt

$$|z_{m,n}|^2 = \prod_{k=2}^{m} \frac{1}{(1+\mathrm{e}^{-\mathrm{i}\pi/k})(1+\mathrm{e}^{\mathrm{i}\pi/k})} \tag{1.72}$$

$$= \prod_{k=2}^{m} \frac{1}{2+2\cos(\pi/k)} = \prod_{k=2}^{m} \frac{1}{4\cos(\pi/2k)^2} \tag{1.73}$$

und damit

$$|z_{m,n}| = \prod_{k=2}^{m} \frac{1}{2\cos(\pi/2k)}. \tag{1.74}$$

Die Koeffizienten $z_{m,n}$ liegen für festes m auf einem Kreis in der komplexen Ebene. Aus der Ungleichung $1 < 1/\cos(\pi/2k) \leq \sqrt{2},\ k \geq 2$ folgt die Beziehung

$$\prod_{k=2}^{m} \frac{1}{2\cos(\pi/2k)} < 2^{-m+1} \prod_{k=2}^{\infty} \frac{1}{\cos(\pi/2k)}. \tag{1.75}$$

Die Abschätzung $1/\cos(\pi x) < \exp(2\pi x^2),\ 0 \leq x \leq 1/4$ liefert das Resultat

$$\prod_{k=2}^{\infty} \frac{1}{\cos(\pi/2k)} < \prod_{k=2}^{\infty} \exp(\frac{\pi}{2k^2}) = \exp(\frac{\pi}{2}\sum_{k=2}^{\infty}\frac{1}{k^2}) \tag{1.76}$$

$$= \exp(\frac{\pi}{2}(\zeta(2)-1)) = \exp(\frac{\pi}{2}(\pi^2/6-1)) < 3. \tag{1.77}$$

Daraus resultiert die gesuchte Ungleichung

$$\prod_{k=2}^{m} \frac{1}{2\cos(\pi/2k)} < 3(2^{-m+1}). \tag{1.78}$$

□

1.6 Gruppeneigenschaft

In der Quantenmechanik erzeugt der Hamilton-Operator die unitäre Gruppe der Zeitevolution. Approximationen an den Exponentialoperator, die aus einer polynomialen Operatorfunktion $Q_1(t)$ aufgebaut sind, verletzen jedoch die Unitarität. Wie bereits in der Einleitung angesprochen wurde, wirkt sich diese Schwäche in der numerischen Rechnung nicht gravierend aus. Eine weitere, für eine numerische Simulation wichtige Eigenschaft ist die Umkehrbarkeit der Zeitevolution. Die fraktalen Approximationen $Q_m(t)$ und $Q_m(-t)$ sind approximativ invers zueinander. Es stellt sich nun die Frage, wie gut die Identität angenähert wird. Der folgende Satz gibt Auskunft über eine verbesserte Konvergenz bei geraden, fraktalen Approximationen.

Satz 1.4. *Sei $\mathfrak{B}$ eine Banach-Algebra und $A \in \mathfrak{B}$ ein Operator. Sei $Q_{2k}(t)$, $k \in \mathbb{N}$ eine fraktale Approximation der Ordnung $2k$ an den Exponentialoperator $\exp(tA)$. Dann wird der Einheitsoperator in der Ordnung $2k+1$ approximiert*

$$Q_{2k}(t)Q_{2k}(-t) = 1 + o(t^{2k+1}) \quad (t \to 0). \tag{1.79}$$

Beweis. Nach dem Konstruktionssatz 1.1 ist $Q_{2k}(t)$ eine analytische Approximation $2k$-ter Ordnung. Also gibt es einen Operator R_{2k} mit

$$\exp(tA) - Q_{2k}(t) = t^{2k+1}R_{2k} + o(t^{2k+1}) \quad (t \to 0). \tag{1.80}$$

Einsetzen und ausmultiplizieren führt direkt auf die Behauptung:

$$Q_{2k}(t)Q_{2k}(-t) \tag{1.81}$$

$$= (\exp(tA) - t^{2k+1}R_{2k})(\exp(-tA) - (-t)^{2k+1}R_{2k}) + o(t^{2k+1}) \tag{1.82}$$

$$= 1 - (1 + (-1)^{2k+1})t^{2k+1}R_{2k} + o(t^{2k+1}) \tag{1.83}$$

$$= 1 + o(t^{2k+1}) \tag{1.84}$$

□

Fraktale Approximationen gerader Ordnung approximieren den Einheitsoperator besser. Diese Eigenschaft der Approximationen konnte auch empirisch bestätigt werden. In der Praxis sind daher gerade, polynomiale Approximationen zu bevorzugen.

1.7 Normabschätzung

Der Abschnitt über die Normabschätzung liefert eine explizite Formel für den Fehler der fraktalen Approximation bei Verwendung der Operatorfunktion

$$Q_1(t) = 1 + tA \tag{1.85}$$

als Ausgangspunkt für die Approximation. Lineare Operatorfunktionen werden später als Basis für die Algorithmen zur Berechnung der Zeitevolution der Schrödinger-, Pauli-, und Dirac-Gleichung dienen. Eine explizite Aussage über den Fehler der Approximation ist daher von besonderem Interesse. Die spezielle Wahl der Operatorfunktion $Q_1(t)$ erlaubt eine scharfe Abschätzung des Approximationsfehlers in Abhängigkeit von der Norm des Operators tA.

Satz 1.5. *Sei $\mathfrak{B}$ eine Banach-Algebra und $A \in \mathfrak{B}$ ein Operator. Sei $Q_m(t)$ eine fraktale Approximation der Ordnung $m \in \mathbb{N}$ an den Exponentialoperator $\exp(tA)$ und $s = 3(2^{-m+1})\|tA\| < 1$. Dann gilt für den Approximationsfehler die Abschätzung*

$$\|\exp(tA) - Q_m(t)\| < (\exp(2^{m-1} \sum_{k=m+1}^{\infty} \frac{s^k}{k}) - 1)(1+s)^{2^{m-1}}. \tag{1.86}$$

Beweis. Der Operator $Q_m(t)$ besitzt die Darstellung

$$Q_m(t) = \prod_{n=1}^{2^{m-1}} (1 + z_{m,n} tA) \tag{1.87}$$

wobei die Koeffizienten $z_{m,n}$ durch Gleichung (1.70) gegeben sind. Für die Norm gilt die Abschätzung

$$\|\exp(tA) - Q_m(t)\| \leq \|\exp(tA) Q_m(t)^{-1} - 1\| \|Q_m(t)\|. \tag{1.88}$$

Unter der Voraussetzung $\|z_{m,n} tA\| < s < 1$ ist der inverse Operator $Q_m(t)^{-1}$ wohldefiniert und es gilt die Identität $Q_m(t)^{-1} = \exp(-\ln Q_m(t))$. Somit gilt:

$$\exp(tA) Q_m(t)^{-1} = \exp(tA - \sum_{n=1}^{2^{m-1}} \ln(1 + z_{m,n} tA)) \tag{1.89}$$

$$= \exp(tA + \sum_{n=1}^{2^{m-1}} \sum_{k=1}^{\infty} \frac{(-z_{m,n} tA)^k}{k}) \tag{1.90}$$

Die Reihendarstellung des Logarithmus ist absolut konvergent, also können die Summen vertauscht werden.

$$= \exp(tA + \sum_{k=1}^{\infty} \frac{(-tA)^k}{k} \sum_{n=1}^{2^{m-1}} z_{m,n}^k) \tag{1.91}$$

$$= \exp(\sum_{k=m+1}^{\infty} \frac{(-tA)^k}{k} \sum_{n=1}^{2^{m-1}} z_{m,n}^k) \tag{1.92}$$

Die letzte Zeile folgt aus den Bedingungsgleichungen (1.24). Die Zahlen $z_{m,n}$ sind aus einem Produkt der Koeffizienten $p_{m,j}$ aufgebaut. Einsetzen der Bedingungsgleichungen führt auf $\sum_{n=1}^{2^{m-1}} z_{m,n}^k = \delta_{1,k}$ für $1 \leq k \leq m$.

Die Norm des Exponenten berechnet sich unter Beachtung der Satzes 1.3 über die Beschränktheit des Betrages der Koeffizienten $z_{m,n}$ zu

$$\left\| \sum_{k=m+1}^{\infty} \frac{(-tA)^k}{k} \sum_{n=1}^{2^{m-1}} z_{m,n}^k \right\| \leq \sum_{k=m+1}^{\infty} \frac{\|tA\|^k}{k} \sum_{n=1}^{2^{m-1}} |z_{m,n}|^k \tag{1.93}$$

$$< 2^{m-1} \sum_{k=m+1}^{\infty} \frac{3^k (2^{-m+1})^k \|tA\|^k}{k} \tag{1.94}$$

$$= 2^{m-1} \sum_{k=m+1}^{\infty} \frac{s^k}{k}. \tag{1.95}$$

Weiters gilt für die Norm der fraktalen Approximation

$$\|Q_m(t)\| = \left\| \prod_{n=1}^{2^{m-1}} (1 + z_{m,n} tA) \right\| \leq \prod_{n=1}^{2^{m-1}} (1 + |z_{m,n}| \|tA\|) \tag{1.96}$$

$$< \prod_{n=1}^{2^{m-1}} (1 + 3(2^{-m+1}) \|tA\|) = (1+s)^{2^{m-1}}. \tag{1.97}$$

Unter Verwendung der Ungleichung

$$\| \exp(T) - 1\| = \left\| \sum_{k=1}^{\infty} \frac{T^k}{k!} \right\| \leq \sum_{l=1}^{\infty} \frac{\|T\|^k}{k!} = \exp(\|T\|) - 1 \tag{1.98}$$

berechnet sich der Approximationsfehler zu

$$\| \exp(tA) - Q_m(t)\| < (\exp(2^{m-1} \sum_{k=m+1}^{\infty} \frac{s^k}{k}) - 1)(1+s)^{2^{m-1}} \tag{1.99}$$

und das entspricht der Behauptung des Satzes. □

Die wesentliche Aussage der Abschätzung ist der Konvergenzradius der Approximation

$$s = 3(2^{-m+1}) \|tA\|, \quad s < 1. \tag{1.100}$$

Die Verbesserung der Approximation um eine Ordnung vergrößert den Konvergenzradius um einen Faktor 2. Dieses theoretische Resultat konnte qualitativ auch in den numerischen Rechnungen beobachtet werden.

Kapitel 2

Numerische Algorithmen

2.1 Konzeption

Die Konzeption der numerischen Algorithmen zur Behandlung von Anfangswertproblemen aus der Quantenmechanik basiert auf der fraktalen Zerlegung des Evolutionsoperators. Das quantenmechanische Anfangswertproblem

$$\mathrm{i}\frac{\mathrm{d}}{\mathrm{d}t}\psi(t) = H\psi(t) \tag{2.1}$$

$$\psi(0) \in \mathcal{D}(H) \subset \mathfrak{H} \tag{2.2}$$

besitzt die allgemeine Lösung

$$\psi(t) = \exp(-\mathrm{i}tH)\psi(0). \tag{2.3}$$

Nach dem Konstruktionssatz 1.1 für fraktale Approximationen kann der Exponentialoperator mit Hilfe einer analytischen, operatorwertigen Funktion in beliebiger Ordnung angenähert werden. Die Freiheit in der Wahl der Operatorfunktion kann zur Konzeption eines effektiven Algorithmus ausgenützt werden. Unter dem Gesichtspunkt des numerischen Rechenaufwandes ist eine lineare Operatorfunktion die beste Wahl.

$$Q_1(t) = 1 - \mathrm{i}tH \tag{2.4}$$

Die Anwendung des Operators auf eine Wellenfunktion erfordert im wesentlichen nur eine Matrix-Vektor-Multiplikation.

Eine noch offene Frage ist die Wahl der Komplexität $r \in \mathbb{N},\ r \geq 2$ der fraktalen Approximation. So benötigt eine fraktale Approximation m-ter Ordnung r^{m-1} Multiplikatoren. Im Vordergrund der Arbeit steht ein effizienter Algorithmus, also fällt die Wahl auf $r = 2$. Die Wahl der Komplexität $r = 3$ ist für symmetrische Approximationen, im Sinne des Symmetriesatzes 1.2, sinnvoll. Die Symmetrieforderung ist für polynomiale Operatorfunktionen nicht erfüllt, daher ist die kleinst mögliche Wahl der Komplexität der fraktalen Zerlegung gerechtfertigt.

Mit diesen Festlegungen einer linearen Operatorfunktion und der kleinst möglichen Komplexität, ist die fraktale Zerlegung eindeutig bestimmt. Eine

Approximation an den Evolutionsoperator $\exp(-\mathrm{i}tH)$ der Ordnung m kann explizit angegeben werden.

$$Q_m(t) = \prod_{n=1}^{2^{m-1}} (1 - \mathrm{i}z_{m,n}tH) \tag{2.5}$$

$$z_{m,n} = \prod_{k=2}^{m} (1 + \mathrm{i}\mathrm{e}^{\mathrm{i}\pi\nu_k} \tan(\pi/2k))/2 \tag{2.6}$$

$$n = 1 + \sum_{l=2}^{m} \nu_l 2^{l-2} \tag{2.7}$$

Die Ableitung der Formel findet sich im Abschnitt 1.5. Das Konvergenzintervall der Approximation ist durch die Bedingung

$$\|tH\| < 2^{m-1}/3 \tag{2.8}$$

festgelegt. Das heißt, fraktale Approximationen höherer Ordnung liefern nicht nur eine bessere Konvergenz, sondern erlauben auch größere Zeitschritte. Eine Diskussion dazu kann im Abschnitt 1.7 nachgelesen werden.

2.1.1 Algorithmus

Die Formulierung des Algorithmus zur Berechnung der fraktalen Approximation wird durch die Formel (2.5) bereits nahegelegt. Eine sprachenunabhängige Formulierung des Algorithmus in Pseudocode soll das Prinzip verdeutlichen. Im wesentlichen besteht der Algorithmus aus drei Teilen. Der Hauptteil, die Funktion *approximation*, berechnet für eine Wellenfunktion ψ einen Zeitschritt der Länge t und legt das Ergebnis in ϕ ab. Der Evolutionsoperator wird dabei in m-ter Ordnung approximiert.

```
function φ := approximation(ψ, t, m)
    for n := 1 to 2^(m-1) do
        z := fractal(m, n);
        φ := kernel(ψ, zt);
        ψ := φ;
    end for
```
(2.9)

Die erste der verbleibenden Funktionen, *fractal*, berechnet den komplexen Koeffizienten $z_{m,n}$ der fraktalen Approximation. Die Ordnung der Approximation

m und der Index n können entsprechend angegeben werden.

$$\begin{aligned}
&\textbf{function } z := \mathit{fractal}(m, n)\\
&\quad z := 1;\\
&\quad n := n - 1;\\
&\quad \textbf{for } k := 2 \textbf{ to } m \textbf{ do}\\
&\qquad p := (1 + \mathrm{i}\tan(\pi/2k))/2;\\
&\qquad \textbf{if } 1 \equiv n \mod 2 \textbf{ then}\\
&\qquad\quad p := \bar{p};\\
&\qquad\quad n := n - 1;\\
&\qquad \textbf{end if}\\
&\qquad z := pz;\\
&\qquad n := n/2;\\
&\quad \textbf{end for}
\end{aligned} \tag{2.10}$$

Die letzte Funktion *kernel* berechnet die Anwendung des Operators $(1 - \mathrm{i}zH)$ auf die Wellenfunktion ψ und gibt das Resultat in ϕ zurück. Die komplexe Zahl z, der komplexe Zeitschritt der Approximation, wird als Parameter übergeben.

$$\begin{aligned}
&\textbf{function } \phi := \mathit{kernel}(\psi, z)\\
&\quad \phi := (1 - \mathrm{i}zH)\psi;
\end{aligned} \tag{2.11}$$

Die Isolation der Funktion *kernel* aus dem Hauptteil des Algorithmus erlaubt eine einfache Implementierung von verschiedenen Hamilton-Operatoren, ohne die Struktur des Hauptteiles zu zerstören.

Die folgenden Abschnitte behandeln die Diskretisierung von verschiedenen Hamilton-Operatoren. Das heißt, die in der Funktion *kernel* noch abstrakt formulierte Anwendung des Hamilton-Operators auf eine Wellenfunktion wird konkretisiert, sodaß der Algorithmus in ein funktionierendes Programm umgesetzt werden kann.

2.2 Schrödinger-Operator

Der Schrödinger-Operator kann, in seiner abstrakten Form, als

$$H_S = \frac{1}{2m}\mathbf{D}^*\mathbf{D} + e(V + \mathrm{i}W), \quad \mathbf{D} = \mathbf{p} - e\mathbf{A} \tag{2.12}$$

angegeben werden. Das Interesse der Arbeit liegt bei zwei- und dreidimensionalen Problemen, da die Berücksichtigung von Magnetfeldern nur in diesem Rahmen möglich ist. Das Plancksche Wirkungsquantum und die Lichtgeschwindigkeit werden, als Konstanten, nicht angegeben. Die Einheiten sind also so zu verstehen, daß $\hbar = c = 1$ gilt. In der üblichen Notation bezeichnet m die Masse des Teilchens, $\mathbf{p}$ den Impulsoperator, $\mathbf{A}$ das Vektorpotential und V das skalare Potential. Das rein imaginäre Potential $\mathrm{i}W$ wird zur Berücksichtigung von nichtlokalen, absorbierenden Randbedingungen eingeführt. Ein Operator mit Stern ist als adjungierter Operator zu verstehen.

Für den Übergang in die Ortsraumdarstellung ist es angebracht, den abstrakten Schrödinger-Operator auszuschreiben.

$$H_S = \frac{1}{2m}\left(\mathbf{p}^2 - e\mathbf{A}\mathbf{p} - e\mathbf{p}\mathbf{A} + e^2\mathbf{A}^2\right) + e(V + \mathrm{i}W) \tag{2.13}$$

Der Zusammenhang $\mathbf{p} = -\mathrm{i}\nabla$ liefert bei Beachtung der Rechenregeln für Operatoren den Schrödinger-Operator in Ortsraumdarstellung.

$$H_S = \frac{1}{2m}\left(-\Delta + 2\mathrm{i}e\mathbf{A}\cdot\nabla + \mathrm{i}e\,\mathrm{div}\mathbf{A} + e^2\mathbf{A}^2\right) + e(V + \mathrm{i}W) \tag{2.14}$$

Die weitere Vorgangsweise erfordert nun eine getrennte Betrachtung der zwei- und dreidimensionalen Gleichung.

2.2.1 2D-Schrödinger-Operator

Die Approximation des Hamilton-Operators im zweidimensionalen Raum wird durch die Restriktion des Hilbertraumes $\mathfrak{H}$ auf ein rechteckiges, regelmäßiges Gitternetz realisiert. Die Maschenweite h und die Anzahl der Gitterpunkte n_1, n_2 bezüglich der Koordinatenachsen legen das Gitter fest. Die Abmessungen l_1, l_2 des Rechtecks, das das Gitter umschließt, sind durch

$$l_1 = h(n_1 - 1), \quad l_2 = h(n_2 - 1) \tag{2.15}$$

festgelegt. Um die Notation einfach zu gestalten, wird die linke untere Ecke des Rechtecks in den Koordinatenursprung gelegt.

Die Berücksichtigung eines irregulären Grundgebietes kann durch die Einbettung des Gebietes in ein Rechteck erreicht werden. In der Rechnung werden dann nur die Gitterpunkte bearbeitet, die im betrachteten Gebiet liegen. Diese Methode ermöglicht zwar nur eine grobe Auflösung des Randes, aber die Implementierung ist einfach und die Definition von irregulären Gebieten kann leicht realisiert werden.

Die Approximation eines Zustandsvektors aus dem Hilbertraum $\mathfrak{H}$ wird mit Hilfe einer Projektion auf das Gitternetz definiert.

$$\psi_{i,j} = \psi(ih, jh), \quad i = 0, \ldots, n_1 - 1,\ j = 0, \ldots, n_2 - 1 \tag{2.16}$$

Wie in der Quantenmechanik üblich, werden Dirichlet-Randbedingungen verwendet und die Wellenfunktion am Rand auf Null gesetzt.

Die Wirkung des Schrödinger-Operators (2.14) auf eine Wellenfunktion im restringierten Raum läßt sich mit Hilfe der Finite-Differenzen-Methode schrittweise berechnen. Der erste Summand im Schrödinger-Operator ist der Laplace-Operator. Der Laplace-Operator wird durch den Fünf-Punkte-Stern approximiert.

$$(-\Delta\,\psi)_{i,j} = \frac{1}{h^2}(4\psi_{i,j} - \psi_{i+1,j} - \psi_{i-1,j} - \psi_{i,j+1} - \psi_{i,j-1}) \tag{2.17}$$

Der Ableitungsoperator im zweiten Summanden kann mit Hilfe des zentralen Differenzenquotienten angenähert werden.

$$(2\mathrm{i}e\mathbf{A}\cdot\nabla\,\psi)_{i,j} = \frac{\mathrm{i}e}{h}(A_{1;i,j}(\psi_{i+1,j} - \psi_{i-1,j}) + A_{2;i,j}(\psi_{i,j+1} - \psi_{i,j-1})) \tag{2.18}$$

Der Divergenzterm des Vektorpotentials wirkt als Multiplikationsoperator. Die Ableitung des Vektorpotentials wird mit dem zentralen Differenzenquotienten berechnet.

$$(\mathrm{i}e\,\mathrm{div}\mathbf{A}\,\psi)_{i,j} = \frac{\mathrm{i}e}{2h}(A_{1;i+1,j} - A_{1;i-1,j} + A_{2;i,j+1} - A_{2;i,j-1})\psi_{i,j} \tag{2.19}$$

Die verbleibenden Terme sind ebenfalls Multiplikationsoperatoren.

$$(e^2\mathbf{A}^2\,\psi)_{i,j} = e^2(A^2_{1;i,j} + A^2_{2;i,j})\psi_{i,j} \tag{2.20}$$

$$(e(V + \mathrm{i}W)\,\psi)_{i,j} = e(V_{i,j} + \mathrm{i}W_{i,j})\psi_{i,j} \tag{2.21}$$

Mit den angegebenen Approximationen kann die Funktion *kernel* aufgebaut werden. Die Umsetzung der Operation

$$\phi = (1 - \mathrm{i}zH_S)\psi \tag{2.22}$$

in eine C++-Methode zeigt der folgende Codeausschnitt.

```
// ------------------------------------------------------------------------
//       Kernel
// ------------------------------------------------------------------------

void    TSchroedinger2D::Kernel(    Float* rePsiP,
                                    Float* imPsiP,
                                    Float* rePhiP,
                                    Float* imPhiP,
                                    Float* v0P,
                                    Float* w0P,
                                    Float* a1P,
                                    Float* a2P,
                                    Int8* domP,
                                    Float reZ,
                                    Float imZ,
                                    Int32 ni,
                                    Int32 nj)
{
    Float   rePsiC, imPsiC, reEtaC, imEtaC;
    Float   rePsiR, imPsiR, rePsiL, imPsiL;
    Float   rePsiU, imPsiU, rePsiD, imPsiD;
    Float   reT, imT;
    Float   v0C, w0C;
    Float   a1C, a1R, a1L;
    Float   a2C, a2U, a2D;
    Float   chh, ceh, deh, cee, e;
    Float   one = 1.0, two = 2.0, four = 4.0;
    Int32   i, mode = 0;


    chh = one / (two * mMass * mUnits * mUnits);
    ceh = mCharge / (two * mMass * mUnits);
    deh = ceh / two;
    cee = mCharge * mCharge / (two * mMass);
    e = mCharge;

    if( mScalarID ) mode |= kScalar;
    if( mVectorID ) mode |= kVector;

    for( i = 0; i < ni*nj; i++ ) {
        if( *domP++ ) {
            rePsiR = *(rePsiP + 1);
            imPsiR = *(imPsiP + 1);
            rePsiL = *(rePsiP +-1);
            imPsiL = *(imPsiP +-1);
            rePsiU = *(rePsiP + ni);
            imPsiU = *(imPsiP + ni);
            rePsiD = *(rePsiP +-ni);
```

```
            imPsiD = *(imPsiP +-ni);
            rePsiC = *rePsiP++;
            imPsiC = *imPsiP++;
            reT = rePsiR + rePsiL + rePsiU + rePsiD;
            imT = imPsiR + imPsiL + imPsiU + imPsiD;
            reEtaC = chh * (four * rePsiC - reT);
            imEtaC = chh * (four * imPsiC - imT);
            if( mode & kVector ) {
                a1R = *(a1P + 1);
                a1L = *(a1P +-1);
                a2U = *(a2P + ni);
                a2D = *(a2P +-ni);
                a1C = *a1P++;
                a2C = *a2P++;
                reEtaC -= ceh * a1C * (imPsiR - imPsiL);
                imEtaC += ceh * a1C * (rePsiR - rePsiL);
                reEtaC -= ceh * a2C * (imPsiU - imPsiD);
                imEtaC += ceh * a2C * (rePsiU - rePsiD);
                reT = cee * (a1C * a1C + a2C * a2C);
                reEtaC += reT * rePsiC;
                imEtaC += reT * imPsiC;
                imT = deh * (a1R - a1L + a2U - a2D);
                reEtaC -= imT * imPsiC;
                imEtaC += imT * rePsiC;
            }
            if( mode & kScalar ) {
                v0C = *v0P++;
                w0C = *w0P++;
                reEtaC += e * v0C * rePsiC;
                imEtaC += e * v0C * imPsiC;
                reEtaC -= e * w0C * imPsiC;
                imEtaC += e * w0C * rePsiC;
            }
            *rePhiP++ = rePsiC + imZ * reEtaC + reZ * imEtaC;
            *imPhiP++ = imPsiC - reZ * reEtaC + imZ * imEtaC;
        }
        else {
            rePsiP++;
            imPsiP++;
            if( mode & kScalar ) { v0P++; w0P++; }
            rePhiP++;
            imPhiP++;
            if( mode & kVector ) { a1P++; a2P++; }
        }
    }

}
```

└

Die Methode ist so aufgebaut, daß eine leichte Identifikation der einzelnen Summanden des Schrödinger-Operator möglich ist.

$$(-\Delta\,\psi)_{i,j} = \frac{1}{h^2}(4\psi_{i,j} - \psi_{i+1,j} - \psi_{i-1,j} - \psi_{i,j+1} - \psi_{i,j-1}) \tag{2.23}$$

┌

```
reT = rePsiR + rePsiL + rePsiU + rePsiD;
imT = imPsiR + imPsiL + imPsiU + imPsiD;
reEtaC = chh * (four * rePsiC - reT);
imEtaC = chh * (four * imPsiC - imT);
```

└

$$(2\mathrm{i}e\mathbf{A}\cdot\nabla\,\psi)_{i,j} = \frac{\mathrm{i}e}{h}(A_{1;i,j}(\psi_{i+1,j} - \psi_{i-1,j}) + A_{2;i,j}(\psi_{i,j+1} - \psi_{i,j-1})) \tag{2.24}$$

┌

```
reEtaC -= ceh * a1C * (imPsiR - imPsiL);
imEtaC += ceh * a1C * (rePsiR - rePsiL);
reEtaC -= ceh * a2C * (imPsiU - imPsiD);
imEtaC += ceh * a2C * (rePsiU - rePsiD);
```

$$(e^2\mathbf{A}^2\,\psi)_{i,j} = e^2(A^2_{1;i,j} + A^2_{2;i,j})\psi_{i,j} \tag{2.25}$$

```
reT = cee * (a1C * a1C + a2C * a2C);
reEtaC += reT * rePsiC;
imEtaC += reT * imPsiC;
```

$$(\mathrm{i}e\,\mathrm{div}\mathbf{A}\,\psi)_{i,j} = \frac{\mathrm{i}e}{2h}(A_{1;i+1,j} - A_{1;i-1,j} + A_{2;i,j+1} - A_{2;i,j-1})\psi_{i,j} \tag{2.26}$$

```
imT = deh * (a1R - a1L + a2U - a2D);
reEtaC -= imT * imPsiC;
imEtaC += imT * rePsiC;
```

$$(e(V + \mathrm{i}W)\,\psi)_{i,j} = e(V_{i,j} + \mathrm{i}W_{i,j})\psi_{i,j} \tag{2.27}$$

```
reEtaC += e * v0C * rePsiC;
imEtaC += e * v0C * imPsiC;
reEtaC -= e * w0C * imPsiC;
imEtaC += e * w0C * rePsiC;
```

$$((1 - \mathrm{i}zH_S)\,\psi)_{i,j} = \phi_{i,j} \tag{2.28}$$

```
*rePhiP++ = rePsiC + imZ * reEtaC + reZ * imEtaC;
*imPhiP++ = imPsiC - reZ * reEtaC + imZ * imEtaC;
```

Die Variablen im Programm sind so bezeichnet, daß ihre Bedeutung klar wird. Die Abkürzungen R, L, U, D und C stehen für right, left, up, down und center. Diese Bezeichnungskonvention ermöglicht einen übersichtlichen Programmcode und vermeidet potentielle Indexfehler.

2.2.2 3D-Schrödinger-Operator

Die Approximation des Schrödinger-Operators im dreidimensionalen Raum folgt weitgehend dem zweidimensionalen Vorbild. Der Hilbertraum $\mathfrak{H}$ wird auf ein räumliches regelmäßiges Gitternetz eingeschränkt. Die Maschenweite ist in alle drei Raumrichtungen gleich h und n_1, n_2, n_3 entsprechen der Anzahl der Gitterpunkte bezüglich der Koordinatenachsen. Die Abmessungen des umschreibenden Quaders sind l_1, l_2, l_3 und analog wie im zweidimensionalen Fall zu berechnen.

$$l_1 = h(n_1 - 1), \quad l_2 = h(n_2 - 1), \quad l_3 = h(n_3 - 1) \tag{2.29}$$

Die Approximation des Zustandsvektors ist durch

$$\psi_{i,j,k} = \psi(ih, jh, kh) \tag{2.30}$$

$$i = 0, \ldots, n_1 - 1,\ j = 0, \ldots, n_2 - 1,\ k = 0, \ldots, n_3 - 1 \tag{2.31}$$

definiert. Die Wirkung des Schrödinger-Operators (2.14) im dreidimensionalen Raum wird mit Hilfe der Finite-Differenzen-Methode auf dem Gitternetz definiert. Die Zerlegung in die einzelnen Summanden folgt dem zweidimensionalen Vorbild.

$$(-\Delta\,\psi)_{i,j,k} = \frac{1}{h^2}(6\psi_{i,j,k} - \psi_{i+1,j,k} - \psi_{i-1,j,k} \tag{2.32}$$

$$- \psi_{i,j+1,k} - \psi_{i,j-1,k} - \psi_{i,j,k+1} - \psi_{i,j-1,k-1}) \tag{2.33}$$

Der Ableitungsoperator wird wieder mit Hilfe des zentralen Differenzenquotienten angenähert.

$$(2\mathrm{i}e\mathbf{A}\cdot\nabla\,\psi)_{i,j,k} = \frac{\mathrm{i}e}{h}(A_{1;i,j,k}(\psi_{i+1,j,k} - \psi_{i-1,j,k}) \tag{2.34}$$

$$+ A_{2;i,j,k}(\psi_{i,j+1,k} - \psi_{i,j-1,k}) + A_{3;i,j,k}(\psi_{i,j,k+1} - \psi_{i,j,k-1})) \tag{2.35}$$

Der Divergenzterm kann analog zum zweidimensionalen Fall angegeben werden.

$$(\mathrm{i}e\,\mathrm{div}\mathbf{A}\,\psi)_{i,j,k} = \frac{\mathrm{i}e}{2h}(A_{1;i+1,j,k} - A_{1;i-1,j,k} \tag{2.36}$$

$$+ A_{2;i,j+1,k} - A_{2;i,j-1,k} + A_{3;i,j,k+1} - A_{3;i,j,k+1})\psi_{i,j,k} \tag{2.37}$$

Die verbleibenden Terme sind ebenfalls Multiplikationsoperatoren und sind einfach zu berechnen.

$$(e^2\mathbf{A}^2\,\psi)_{i,j,k} = e^2(A^2_{1;i,j,k} + A^2_{2;i,j,k} + A^2_{3;i,j,k})\psi_{i,j,k} \tag{2.38}$$

$$(e(V + \mathrm{i}W)\,\psi)_{i,j,k} = e(V_{i,j,k} + \mathrm{i}W_{i,j,k})\psi_{i,j,k} \tag{2.39}$$

Die Implementierung in einer C++-Methode kann im Anhang A.8 nachgelesen werden. Es werden die selben Konventionen zur Bezeichnung von Variablen verwendet wie im zweidimensionalen Fall. Die dritte Raumdimension wird mit den Bezeichnungen F und B, also front und back, berücksichtigt.

2.3 Pauli-Operator

Der Pauli-Operator kann abstrakt in der selben Form wie der Schrödinger-Operator angeschrieben werden.

$$H_P = \frac{1}{2m}D^*D + e(V + \mathrm{i}W) \tag{2.40}$$

Bei der Berücksichtigung von nur zwei Raumdimensionen ist D ein komplexer Operator.

$$D = (p_1 - eA_1) + \mathrm{i}(p_2 - eA_2) \tag{2.41}$$

Die Spineigenschaften eines Teilchens werden vom dreidimensionalen Pauli-Operator erfaßt. Der Operator D ist ein 2×2 Matrixoperator und wird mit Hilfe der Pauli-Matrizen

$$\sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_2 = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix}, \quad \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.42}$$

definiert:

$$D = \sum_{i=1}^{3} \sigma_i (p_i - eA_i) \tag{2.43}$$

Der Übergang in die Ortsraumdarstellung wird durch eine explizite Darstellung des Pauli-Operators erleichtert. Zur Vereinfachung der Schreibweise wird das äußere Produkt verwendet. Für zweidimensionale Vektoren gilt die Definition $\mathbf{a} \wedge \mathbf{b} = a_1 b_2 - a_2 b_1$. Der Pauli-Operator lautet damit:

$$H_P = \frac{1}{2m}(\mathbf{p}^2 - e\mathbf{A}\mathbf{p} - e\mathbf{p}\mathbf{A} + e^2\mathbf{A}^2 \tag{2.44}$$

$$- \mathrm{i}e(\mathbf{p} \wedge \mathbf{A} + \mathbf{A} \wedge \mathbf{p})) + e(V + \mathrm{i}W) \tag{2.45}$$

Die Ersetzung des Impulsoperators $\mathbf{p} = -\mathrm{i}\nabla$ liefert unter Beachtung der üblichen Rechenregeln den Pauli-Operator im Ortsraum.

$$H_P = \frac{1}{2m}(-\Delta + 2\mathrm{i}e\mathbf{A} \cdot \nabla + \mathrm{i}e \,\mathrm{div}\mathbf{A} + e^2\mathbf{A}^2 - e \,\mathrm{rot}\mathbf{A}) + e(V + \mathrm{i}W) \tag{2.46}$$

In zwei Raumdimensionen ist $\mathrm{rot}\mathbf{A} = \partial_1 A_2 - \partial_2 A_1$ definiert.

Der Pauli-Operator im dreidimensionalen Raum kann in analoger Weise in die Ortsraumdarstellung übergeführt werden. Das äußere Produkt im dreidimensionalen Raum entspricht dem üblichen Vektorprodukt. Für den Umgang mit den Pauli-Matrizen ist die Beziehung $(\boldsymbol{\sigma} \cdot \mathbf{a})(\boldsymbol{\sigma} \cdot \mathbf{b}) = \mathbf{1}\,(\mathbf{a} \cdot \mathbf{b}) + \mathrm{i}\boldsymbol{\sigma} \cdot (\mathbf{a} \wedge \mathbf{b})$ nützlich. Damit gilt:

$$H_P = \frac{1}{2m}(\mathbf{1}\,(\mathbf{p}^2 - e\mathbf{A}\mathbf{p} - e\mathbf{p}\mathbf{A} + e^2\mathbf{A}^2) \tag{2.47}$$

$$- \mathrm{i}e\boldsymbol{\sigma} \cdot (\mathbf{p} \wedge \mathbf{A} + \mathbf{A} \wedge \mathbf{p})) + e\mathbf{1}\,(V + \mathrm{i}W) \tag{2.48}$$

Die Übersetzung in den Ortsraum bereitet nun keine Schwierigkeiten mehr.

$$H_P = \frac{1}{2m}(\mathbf{1}\,(-\Delta + 2\mathrm{i}e\mathbf{A} \cdot \nabla + \mathrm{i}e \,\mathrm{div}\mathbf{A} + e^2\mathbf{A}^2) - e\boldsymbol{\sigma} \cdot \mathrm{rot}\mathbf{A}) + e\mathbf{1}\,(V + \mathrm{i}W) \tag{2.49}$$

Im weiteren wird der Pauli-Operator für den zwei- und dreidimensionalen Fall diskretisiert.

2.3.1 2D-Pauli-Operator

Für die Implementierung des Pauli-Operators ist es nützlich, die Verbindung zum Schrödinger-Operator auszunutzen.

$$H_P = H_S - \frac{1}{2m} e \,\mathrm{rot}\mathbf{A} \tag{2.50}$$

Verglichen mit dem Schrödinger-Operator gibt es nur einen zusätzlichen Term, der den Einfluß des Magnetfeldes $B = \partial_1 A_2 - \partial_2 A_1$ berücksichtigt.

Die Diskretisierung des Pauli-Operators verläuft daher wie im Falle des zweidimensionalen Schrödinger-Operators. Der verbleibende Term kann, unter Verwendung der im letzten Abschnitt eingeführten Notation, wie folgt diskretisiert werden.

$$(e\,\mathrm{rot}\mathbf{A}\,\psi)_{i,j} = \frac{e}{2h}(A_{2;i+1,j} - A_{2;i-1,j} - A_{1;i,j+1} + A_{1;i,j-1})\psi_{i,j} \tag{2.51}$$

Im wesentlichen reicht für die Implementierung eine kleine Modifikation der Funktion *kernel* des zweidimensionalen Schrödinger-Operator aus. Die C++-Methode zum Pauli-Operator ist im Anhang A.9 nachzulesen.

2.3.2 3D-Pauli-Operator

Der dreidimensionale Pauli-Operator bringt gegenüber dem Schrödinger-Operator in drei Dimensionen eine wesentliche Neuerung. Der Spin erfordert zweikomponentige, komplexe Wellenfunktionen, die sogenannten Spinoren. Der Hilbertraum des quantenmechanische Systems ist typischerweise

$$\mathfrak{H} = L^2(\mathbb{R}^3) \oplus L^2(\mathbb{R}^3) = L^2(\mathbb{R}^3) \otimes \mathbb{C}^2. \tag{2.52}$$

Ein Spinor hat die Gestalt

$$\psi = \begin{pmatrix}\psi_1\\ \psi_2\end{pmatrix} \in \mathfrak{H}. \tag{2.53}$$

Der Vergleich des Pauli-Operators mit dem Schrödinger-Operator ist auch im dreidimensionalen Fall hilfreich.

$$H_P = \mathbf{1}\,H_S - \frac{1}{2m}e\boldsymbol{\sigma}\cdot\mathrm{rot}\mathbf{A} \tag{2.54}$$

Der Anteil vom Schrödinger-Operator wirkt getrennt auf die beiden Komponenten des Spinors ψ_1, ψ_2. Die Implementierung kann also direkt vom Schrödinger-Operator übernommen werden. Der Spinanteil im Pauli-Operator vermischt die beiden Komponenten des Spinors und erfordert eine gesonderte Betrachtung.

$$(e\boldsymbol{\sigma}\cdot\mathrm{rot}\mathbf{A}\,\psi)_{1;i,j,k} \tag{2.55}$$
$$= \frac{e}{2h}((A_{3;i,j+1,k} - A_{3;i,j-1,k} - A_{2;i,j,k+1} + A_{2;i,j,k-1})\psi_{2;i,j,k} \tag{2.56}$$
$$- \mathrm{i}(A_{1;i,j,k+1} - A_{1;i,j,k-1} - A_{3;i+1,j,k} + A_{3;i-1,j,k})\psi_{2;i,j,k} \tag{2.57}$$
$$+ (A_{2;i+1,j,k} - A_{2;i-1,j,k} - A_{1;i,j+1,k} + A_{1;i,j-1,k})\psi_{1;i,j,k}) \tag{2.58}$$

$$(e\boldsymbol{\sigma}\cdot\mathrm{rot}\mathbf{A}\,\psi)_{2;i,j,k} \tag{2.59}$$
$$= \frac{e}{2h}((A_{3;i,j+1,k} - A_{3;i,j-1,k} - A_{2;i,j,k+1} + A_{2;i,j,k-1})\psi_{1;i,j,k} \tag{2.60}$$
$$+ \mathrm{i}(A_{1;i,j,k+1} - A_{1;i,j,k-1} - A_{3;i+1,j,k} + A_{3;i-1,j,k})\psi_{1;i,j,k} \tag{2.61}$$
$$- (A_{2;i+1,j,k} - A_{2;i-1,j,k} - A_{1;i,j+1,k} + A_{1;i,j-1,k})\psi_{2;i,j,k}) \tag{2.62}$$

Mit den beschriebenen Erweiterungen ist die Umsetzung in eine C++-Methode nicht weiter schwierig. Ein Listing ist im Anhang A.10 abgedruckt.

2.4 Dirac-Operator

Der Dirac-Operator ist der zentrale Operator der relativistischen Quantenmechanik. In seiner Struktur ist er wesentlich vom Schrödinger- oder Pauli-Operator verschieden. Qualitativ liegt der Unterschied in der hyperbolischen Struktur der Dirac-Gleichung. Die nichtrelativistischen Gleichungen sind vom parabolischen Typ.

In zwei Raumdimensionen kann der Dirac-Operator unter Verwendung der Pauli-Matrizen (2.42) als 2×2 Matrixoperator angeschrieben werden.

$$H_D = \sigma_1(p_1 - eA_1) + \sigma_2(p_2 - eA_2) + \sigma_3(m - eA_3) + e\mathbf{1}\,(V + \mathrm{i}W) \tag{2.63}$$

Die Struktur des Dirac-Operators kommt in der supersymmetrischen Darstellung [13] klar zum Vorschein.

$$H_D = \begin{pmatrix} m - eA_3 & D^* \\ D & -m + eA_3 \end{pmatrix} + \begin{pmatrix} e(V + \mathrm{i}W) & 0 \\ 0 & e(V + \mathrm{i}W) \end{pmatrix} \tag{2.64}$$

Der Operator

$$D = (p_1 - eA_1) + \mathrm{i}(p_2 - eA_2) \tag{2.65}$$

ist bereits aus der zweidimensionalen Pauli-Gleichung bekannt.

Das Vektorpotential $\mathbf{A}$ verdient eine gesonderte Betrachtung. Die Erweiterung des Vektorpotentials um die A_3-Komponente ermöglicht die Berücksichtigung eines kovarianten skalaren Potentials. Ein derartiges Potential kann zur Implementierung von reflektierenden, nichtlokalen Randbedingungen eingesetzt werden. Das Potential V kann zur Berücksichtigung von elektrostatischen Potentialen verwendet werden. Wie im nichtrelativistischen Fall definiert das imaginäre Potential $\mathrm{i}W$ ein absorbierendes Potential.

Im dreidimensionalen Raum bestimmen die Dirac-Matrizen $\beta, \boldsymbol{\alpha}$ die Struktur des Dirac-Operators.

$$\beta = \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & -\mathbf{1} \end{pmatrix}, \quad \alpha_i = \begin{pmatrix} \mathbf{0} & \sigma_i \\ \sigma_i & \mathbf{0} \end{pmatrix}, \quad i = 1, 2, 3 \tag{2.66}$$

Der dreidimensionale Dirac-Operator ist damit ein 4×4 Matrixoperator.

$$H_D = \boldsymbol{\alpha} \cdot (\mathbf{p} - e\mathbf{A}) + \beta(m - eA_4) + e\mathbf{1}\,(V + \mathrm{i}W) \tag{2.67}$$

Die supersymmetrische Darstellung besitzt die selbe abstrakte Struktur wie oben.

$$H_D = \begin{pmatrix} \mathbf{1}\,(m - eA_4) & D^* \\ D & -\mathbf{1}\,(m - eA_4) \end{pmatrix} + \begin{pmatrix} e\mathbf{1}\,(V + \mathrm{i}W) & \mathbf{0} \\ \mathbf{0} & e\mathbf{1}\,(V + \mathrm{i}W) \end{pmatrix} \tag{2.68}$$

Der Operator

$$D = \sum_{i=1}^{3} \sigma_i(p_i - eA_i) \tag{2.69}$$

entspricht dem bekannten Operator aus der Pauli-Gleichung.

Wie im zweidimensionalen Fall wird in der erweiterten Definition des Vektorpotentials ein kovariantes skalares Potential A_4 berücksichtigt. Weiters sind noch die Potentiale V und iW inkludiert.

Die Ortsraumdarstellung des Dirac-Operators kann leicht angegeben werden. In der supersymmetrischen Darstellung muß nur der Operator D entsprechend angepaßt werden. Mit der Beziehung $\mathbf{p} = -\mathrm{i}\nabla$ gilt in zwei Raumdimensionen

$$D = (-\mathrm{i}\partial_1 + \partial_2) - e(A_1 + \mathrm{i}A_2) \tag{2.70}$$

und in drei Raumdimensionen

$$D = \begin{pmatrix} -\mathrm{i}\partial_3 & -\mathrm{i}\partial_1 - \partial_2 \\ -\mathrm{i}\partial_1 + \partial_2 & \mathrm{i}\partial_3 \end{pmatrix} - e \begin{pmatrix} A_3 & A_1 - \mathrm{i}A_2 \\ A_1 + \mathrm{i}A_2 & -A_3 \end{pmatrix}. \tag{2.71}$$

2.4.1 2D-Dirac-Operator

Die Diskretisierung des Dirac-Operators kann mit den in den vorangegangenen Abschnitten besprochenen Methoden durchgeführt werden. Eine relativistische Wellenfunktion, bei Berücksichtigung von nur zwei Raumdimensionen, lebt im Hilbertraum

$$\mathfrak{H} = L^2(\mathbb{R}^2) \oplus L^2(\mathbb{R}^2) = L^2(\mathbb{R}^2) \otimes \mathbb{C}^2 \tag{2.72}$$

und ist eine zweikomponentige, komplexe Funktion, also ein Spinor.

$$\psi = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} \in \mathfrak{H} \tag{2.73}$$

Die Betonung der Supersymmetrie des Dirac-Operators (2.64) ist für die Konstruktion des diskreten Operators sehr hilfreich. Der einzige nichttriviale Anteil des Dirac-Operators ist der Operator D, vergleiche Formel (2.70). Die restlichen Anteile sind Multiplikationsoperatoren und daher leicht zu berechnen.

$$(D\,\psi_1)_{i,j} = \frac{1}{2h}(-\mathrm{i}(\psi_{1;i+1,j} - \psi_{1;i-1,j}) + (\psi_{1;i,j+1} - \psi_{1;i,j-1})) \tag{2.74}$$

$$- e(A_{1;i,j} + \mathrm{i}A_{2;i,j})\psi_{1;i,j} \tag{2.75}$$

Die Wirkung des adjungierten Operators

$$D^* = (-\mathrm{i}\partial_1 - \partial_2) - e(A_1 - \mathrm{i}A_2) \tag{2.76}$$

kann analog berechnet werden.

$$(D^*\,\psi_2)_{i,j} = \frac{1}{2h}(-\mathrm{i}(\psi_{2;i+1,j} - \psi_{2;i-1,j}) - (\psi_{2;i,j+1} - \psi_{2;i,j-1})) \tag{2.77}$$

$$- e(A_{1;i,j} - \mathrm{i}A_{2;i,j})\psi_{2;i,j} \tag{2.78}$$

Die konkrete Implementierung des zweidimensionalen Dirac-Operators ist im Anhang A.11 angegeben.

2.4.2 3D-Dirac-Operator

In drei Raumdimensionen wirkt der Dirac-Operator auf vierkomponentige, komplexe Wellenfunktionen, die sogenannten Bispinoren. Der zugehörige Hilbertraum hat die Struktur

$$\mathfrak{H} = L^2(\mathbb{R}^3) \oplus L^2(\mathbb{R}^3) \oplus L^2(\mathbb{R}^3) \oplus L^2(\mathbb{R}^3) = L^2(\mathbb{R}^3) \otimes \mathbb{C}^4 \tag{2.79}$$

und ein Element aus diesem Raum kann als

$$\psi = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \end{pmatrix} \in \mathfrak{H} \tag{2.80}$$

angeschrieben werden.

Wie im zweidimensionalen Fall dient die supersymmetrische Formulierung des Dirac-Operators (2.68) als Ausgangspunkt für die Diskretisierung. Relevant ist wiederum nur der Operator D, vergleiche Gleichung (2.71). Die übrigen Anteile des Dirac-Operators sind Multiplikationsoperatoren.

$$\begin{aligned}
(D\begin{pmatrix}\psi_1\\ \psi_2\end{pmatrix})_{1;i,j,k} = \frac{1}{2h}(&-\mathrm{i}(\psi_{1;i,j,k+1} - \psi_{1;i,j,k-1}) && (2.81)\\
&- \mathrm{i}(\psi_{2;i+1,j,k} - \psi_{2;i-1,j,k}) - (\psi_{2;i,j+1,k} - \psi_{2;i,j-1,k})) && (2.82)\\
&- e(A_{3;i,j,k}\psi_{1;i,j,k} + (A_{1;i,j,k} - \mathrm{i}A_{2;i,j,k})\psi_{2;i,j,k}) && (2.83)
\end{aligned}$$

$$\begin{aligned}
(D\begin{pmatrix}\psi_1\\ \psi_2\end{pmatrix})_{2;i,j,k} = \frac{1}{2h}(&-\mathrm{i}(\psi_{1;i+1,j,k} - \psi_{1;i-1,j,k}) && (2.84)\\
&+ (\psi_{1;i,j+1,k} - \psi_{1;i,j+1,k}) + \mathrm{i}(\psi_{2;i,j,k+1} - \psi_{2;i,j,k-1})) && (2.85)\\
&- e((A_{1;i,j,k} + \mathrm{i}A_{2;i,j,k})\psi_{1;i,j,k} - A_{3;i,j,k}\psi_{2;i,j,k}) && (2.86)
\end{aligned}$$

Im Gegensatz zum zweidimensionalen Fall ist der Operator D selbstadjungiert, das heißt $D^* = D$. Die Wirkung des Operators D^* auf die beiden unteren Komponenten der Wellenfunktion ist also die selbe wie auf die beiden oberen. Eine explizite Darstellung ist daher nicht mehr notwendig. Der gesamte Algorithmus ist im Anhang A.12 als C++-Methode angegeben.

Kapitel 3

Visualisierung

3.1 Farbabbildungen

Die Darstellung einer Funktion $f : \mathcal{D}(f) \subset \mathbb{R} \to \mathbb{R}$ durch ihren Graphen, also der Punktmenge

$$G_f = \{(x, f(x))|x \in \mathcal{D}(f)\} \subset \mathbb{R}^2, \tag{3.1}$$

ist eine gebräuchliche Methode, um einen Eindruck vom Funktionsverlauf zu gewinnen. Eine zweidimensionale Funktion $g : \mathcal{D}(g) \subset \mathbb{R}^2 \to \mathbb{R}$ mit dem Graphen

$$G_g = \{(x, y, g(x, y))|(x, y) \in \mathcal{D}(g)\} \subset \mathbb{R}^3 \tag{3.2}$$

kann durch eine Projektion der im $\mathbb{R}^3$ eingebetteten Fläche auf eine zweidimensionale Ebene dargestellt werden. Im Gegensatz zu zweidimensionalen Graphen kann nicht einfach die Punktmenge G_g abgebildet werden. Der entstehende schwarze Fleck würde kaum etwas aussagen. Es ist daher notwendig, die Fläche mit einem Gitternetz zu überziehen und sie transparent zu gestalten oder sie mit Höhenschichtlinien zu versehen. Neben diesen traditionellen Darstellungsmethoden gibt es auch die Möglichkeit, die Fläche mit Farbabstufungen, entsprechend den Funktionswerten, zu zeichnen. Diese Darstellungsmethode soll, in Hinblick auf die Visualisierung von Wellenfunktionen, nun ausführlicher behandelt werden.

3.1.1 Graustufen-Abbildung

Die Verwendung einer Graustufen-Abbildung eignet sich besonders für positive, reelle Funktionen. Die Wahrscheinlichkeitsdichte $|\psi|^2$ ist ein Beispiel dafür. Die einfachste Methode der Abbildung des Graphen einer zweidimensionalen Funktion $g : \mathcal{D}(g) \subset \mathbb{R}^2 \to \mathbb{R}_+$ ist die Projektion auf den Definitionsbereich $\mathcal{D}(g)$. Die einzelnen Punkte $(x, y) \in \mathcal{D}(g)$ werden, entsprechend ihren Funktionswerten $g(x, y) \in \mathbb{R}_+$, mit einer Grauschattierung versehen. Die Abbildung

$$T_G : \begin{cases} \mathbb{R}_+ \to [0, 1] \\ x \mapsto T_G(x) = \frac{2}{\pi} \arctan(x) \end{cases} \tag{3.3}$$

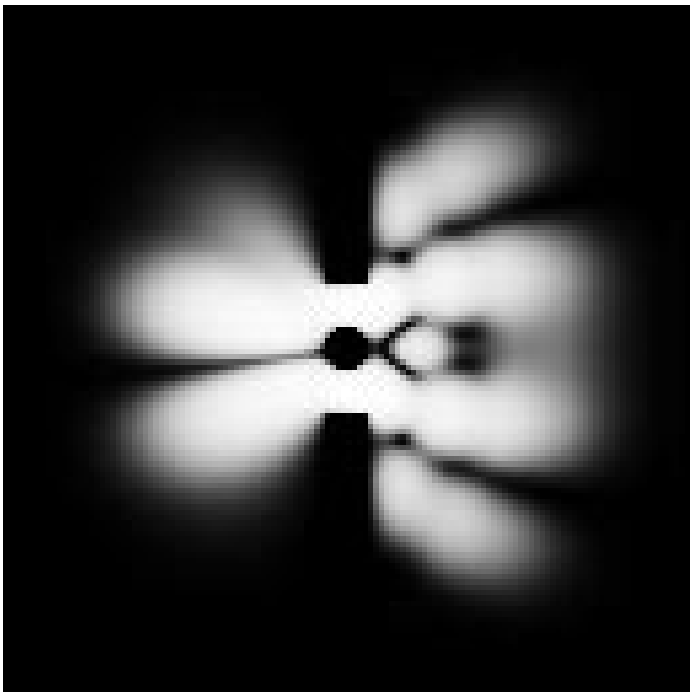

Abbildung 3.1: Zweidimensionale Graustufen-Abbildung

liefert die entsprechende Zuordnung. Jeder positiven, reellen Zahl wird eine Grauschattierung zugeordnet. Die Abbildung T_G besitzt überdies die Eigenschaft, daß die gesamte positive reelle Halbachse auf das Intervall $[0, 1]$ abgebildet wird. Null wird auf schwarz abgebildet und unendlich auf weiß.

Ein Beispiel für die Graustufen-Abbildung zeigt Abbildung 3.1. Zu sehen ist die Wahrscheinlichkeitsdichte $|\psi|^2$ bei einem Doppelspaltexperiment zum Aharonov-Bohm-Effekt. Die selbe Momentaufnahme des Experiments dient später auch als Schaubild für andere Farbtransformationen.

Manchmal ist es auch nützlich, den Graphen nicht nur von oben zu sehen, sondern aus einem beliebigen Blickwinkel. Die Projektion des Graphen auf eine allgemeine Bildebene ist in der Abbildung 3.2 zu sehen.

3.1.2 Schwarz-Weiß-Abbildung

Der Definitionsbereich $\mathcal{D}(g)$ einer zweidimensionalen Funktion g muß nicht notwendigerweise die einfache Gestalt eines Quadrates oder Rechteckes besitzen. So ist zum Beispiel für die Simulation des Aharonov-Bohm-Effektes ein topologisch mehrfach zusammenhängendes Grundgebiet notwendig. Irreguläre Definitionsbereiche werden mit Hilfe einer Funktion $\delta : \mathcal{D}(\delta) \subset \mathbb{R}^2 \to \mathbb{R}$ definiert. Der Definitionsbereich $\mathcal{D}(g) \subset \mathcal{D}(\delta)$ wird durch das Gebiet, auf dem δ positiv ist, festgelegt. Die Schwarz-Weiß-Abbildung T_{BW} ordnet allen positiven Funktions-

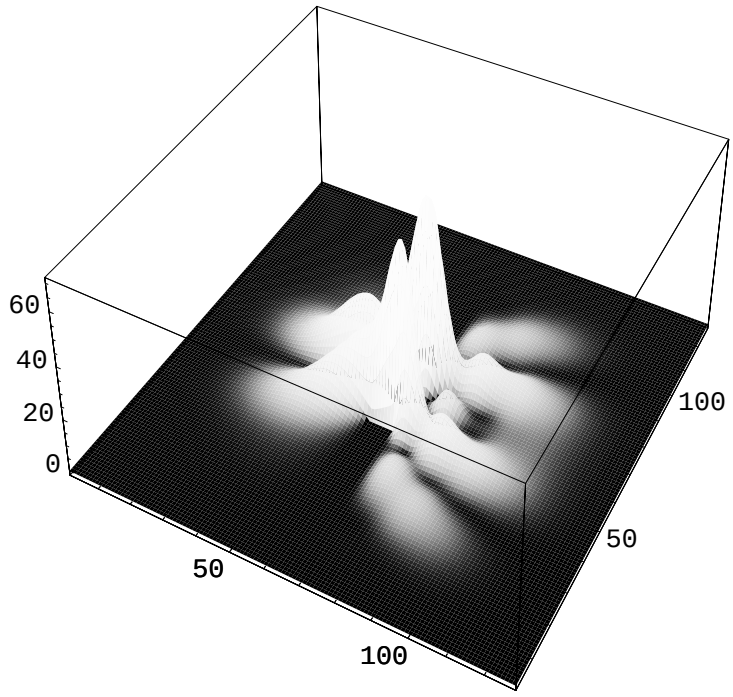

Abbildung 3.2: Dreidimensionale Graustufen-Abbildung

werten schwarz zu und allen anderen weiß.

$$T_{BW} : \begin{cases} \mathbb{R} \to [0,1] \\ x \mapsto T_{BW}(x) = \theta(-x) \end{cases} \tag{3.4}$$

Die Funktion θ ist die Heaviside-Stufenfunktion. Mit dieser Abbildung wird das von δ definierte Grundgebiet $\mathcal{D}(g)$ als schwarzer Bereich dargestellt. Der für die gezeichnete Wellenfunktion ψ verwendete Definitionsbereich ist in Abbildung 5.1 zu sehen.

3.1.3 Rot-Blau-Abbildung

Für die Darstellung des Realteils der Wellenfunktion $\mathrm{Re}\,\psi$ ist die Graustufen-Abbildung nicht geeignet. Eine Erweiterung der Farbtransformation auf die negative reelle Halbachse ist notwendig. Eine elegante Variante ist, die positiven Zahlen rot einzufärben und die negativen Zahlen blau. Der Absolutbetrag bestimmt dann, ähnlich wie bei der Graustufen-Abbildung, die Helligkeit der Farbe. Die Anpassung der Helligkeit einer Ausgangsfarbe $(r, g, b) \in [0,1]^3$, in Abhängigkeit vom Absolutbetrag der Zahl $z \in \mathbb{C}$, erledigt die nachstehende Funktion.

$$l : \begin{cases} \mathbb{C} \times [0,1]^3 \to [0,1]^3 \\ (z, r, g, b) \mapsto l(z, r, g, b) \end{cases} \tag{3.5}$$

Die allgemeine Definition für komplexe Zahlen z wird später gebraucht und soll hier nicht stören. Die Farben werden nach dem RGB-Schema codiert. Ein Punkt im Farbwürfel $(r, g, b) \in [0, 1]^3$ entspricht einer Farbe mit der Intensität r für rot, g für grün und b für blau.

$$l(z, r, g, b) = \begin{cases} s(z)(r, g, b), & s(z) < 1 \\ (2 - s(z))(r, g, b) + (s(z) - 1)(1, 1, 1), & s(z) \geq 1 \end{cases} \tag{3.6}$$

Die Hilfsfunktion

$$s : \begin{cases} \mathbb{C} \to [0, 2] \\ z \mapsto s(z) = \frac{4}{\pi} \arctan |z| \end{cases} \tag{3.7}$$

komplettiert die Definition.

Mit Hilfe der Abbildung l kann die eigentliche Farbtransformation

$$T_{RB} : \begin{cases} \mathbb{R} \to [0, 1]^3 \\ x \mapsto T_{RB}(x) \end{cases} \tag{3.8}$$

mit

$$T_{RB}(x) = \begin{cases} l(x, 0, 0, 1), & x < 0 \\ l(x, 1, 0, 0), & x \geq 0 \end{cases} \tag{3.9}$$

übersichtlich angegeben werden. Die Abbildung erfaßt die gesamte reelle Achse. In der Praxis ist diese Eigenschaft nützlich, da die Ausgangsfunktion nicht skaliert werden muß, um in das vorgegebene Farbintervall zu passen.

Beispiele für diese Farbtransformation sind in den Abbildungen 3.3 und 3.4 zu sehen. Es ist der Realteil der Wellenfunktion $\operatorname{Re} \psi$ beim Doppelspaltexperiment dargestellt.

3.1.4 RGB-Abbildung

Die Abbildung zur grafischen Darstellung der komplexen Wellenfunktion ψ kann als Erweiterung der Rot-Blau-Abbildung verstanden werden. Die wesentliche Idee ist die Codierung der Phase der komplexen Zahl z durch Farben auf dem Farbkreis. Die Helligkeit der Farbe wird mit der Funktion l berechnet. Explizit lautet die Transformation

$$T_{RGB} : \begin{cases} \mathbb{C} \to [0, 1]^3 \\ z \mapsto T_{RGB}(z) \end{cases} \tag{3.10}$$

mit

$$T_{RGB}(z) = \begin{cases} l(z, 0, -\phi(z) - 2, 1), & \phi(z) \in [-3, -2) \\ l(z, 2 + \phi(z), 0, 1), & \phi(z) \in [-2, -1) \\ l(z, 1, 0, -\phi(z)), & \phi(z) \in [-1, 0) \\ l(z, 1, \phi(z), 0), & \phi(z) \in [0, 1) \\ l(z, 2 - \phi(z), 1, 0), & \phi(z) \in [1, 2) \\ l(z, 0, 1, \phi(z) - 2), & \phi(z) \in [2, 3) \end{cases} . \tag{3.11}$$

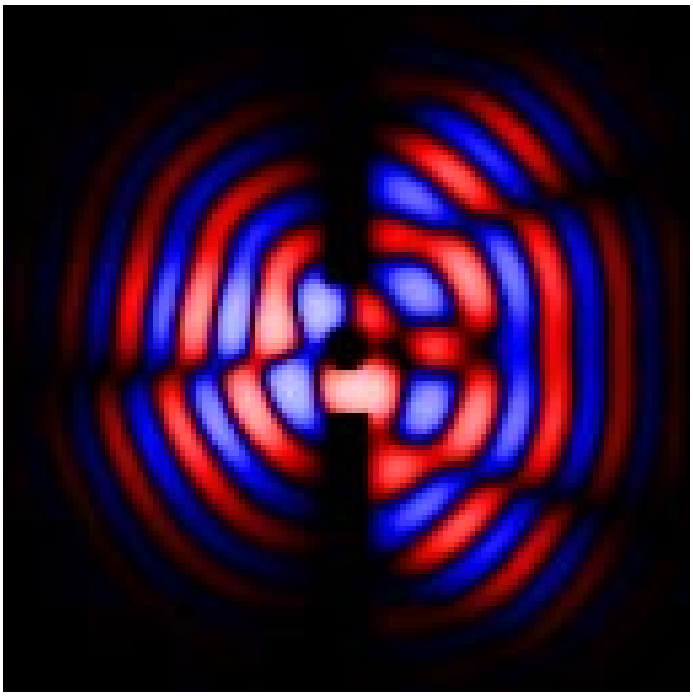

Abbildung 3.3: Zweidimensionale Rot-Blau-Abbildung

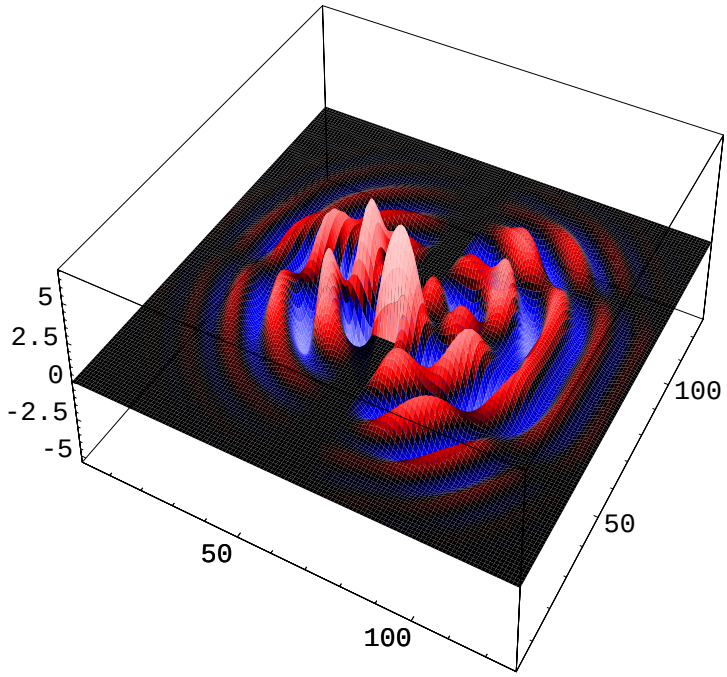

Abbildung 3.4: Dreidimensionale Rot-Blau-Abbildung

Die Hilfsfunktion

$$\phi : \begin{cases} \mathbb{C} \to [-3, 3) \\ z \mapsto \phi(z) = \frac{3}{\pi} \arg(z) \end{cases} \tag{3.12}$$

skaliert die Phase auf das Intervall $[-3, 3)$. Dies erlaubt eine übersichtliche Zuordnung der Farbwerte. Die oben definierten Farbwerte entsprechen der abgerollten Polygonkurve, die den RGB-Farbwürfel umschließt und zwar so, daß die weiße und die schwarze Ecke ausgespart bleiben. Die Polygonkurve besteht also aus den sechs färbigen Kanten des Farbwürfels.

Wie bei den beiden anderen Transformationen wird die gesamte komplexe Ebene erfaßt. Die Oberfläche des Farbwürfels wird eins-zu-eins auf die komplexe Ebene abgebildet. Der Ursprung ist schwarz und der Fernpunkt ∞ weiß. Dies ermöglicht die einfache Identifikation von Nullstellen und Polstellen einer komplexwertigen Funktion. Die Idee zu dieser Farbabbildung stammt von Thaller [15].

Der Algorithmus zur Berechnung der RGB-Abbildung ist unten in Pseudocode angegeben.

function $(r, g, b) := \mathit{colormap}(z)$
  $\phi := 3 \arg(z)/\pi;\ \phi_0 := |\phi|;$
  if $\phi_0 < 1$ **then**
    $r = 1;\ g = \phi_0;\ b = 0;$
  else
    if $\phi_0 < 2$ **then**
      $r = 2 - \phi_0;\ g = 1;\ b = 0;$
    else
      $r = 0;\ g = 1;\ b = \phi_0 - 2;$
    end if
  end if
  if $\phi < 0$ **then**
    $t = g;\ g = b;\ b = t;$
  end if

  $s := 4 \arctan(|z|)/\pi;$
  if $s < 1$ **then**
    $r = rs;\ g = gs;\ b = bs;$
  else
    $t = 2 - s;\ t_0 = s - 1;$
    $r = rt + t_0;\ g = gt + t_0;\ b = bt + t_0;$
  end if (3.13)

Eine Implementierung der Funktion *colormap* findet sich im Anhang A.1. Im selben Listing finden sich auch die Implementierungen der anderen besprochenen Farbabbildungen.

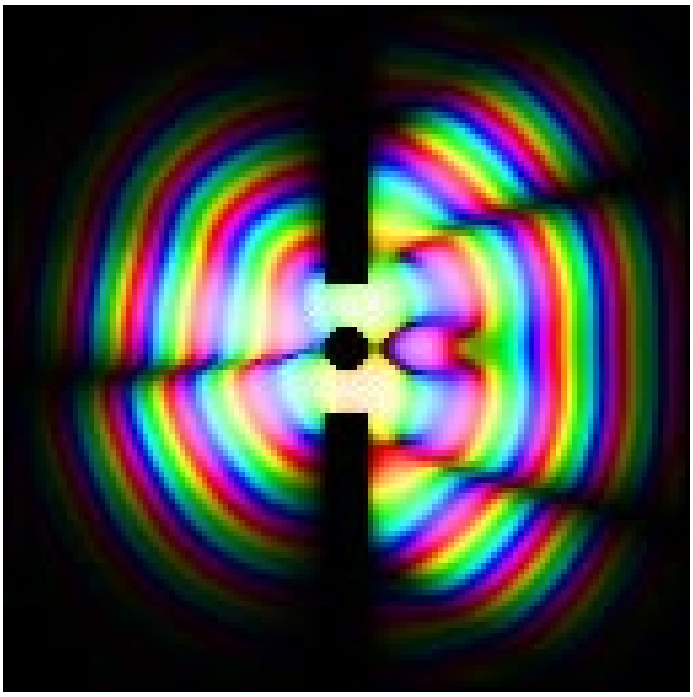

Abbildung 3.5: Zweidimensionale RGB-Abbildung

Die Abbildungen 3.5 und 3.6 zeigen die komplexe Funktion ψ beim Doppelspaltexperiment.

3.2 Animationen

Eine Abbildung des Graphen einer zweidimensionalen Funktion, zum Beispiel der Wellenfunktion, mit den beschriebenen Farbabbildungen erlaubt es, die räumlichen Aspekte der Funktion darzustellen. Im Zusammenhang mit der Zeitevolution der Wellenfunktion, des Zustandsvektors, kann eine Momentaufnahme des Geschehens wiedergegeben werden.

Der zeitliche Ablauf einer Simulation kann für eine eindimensionale Funktion $f : \mathcal{D}(f) \subset \mathbb{R} \times [0, T] \to \mathbb{R}$ in einem Graphen dargestellt werden.

$$G_f = \{(x, t, f(x, t))|(x, t) \in \mathcal{D}(f)\} \subset \mathbb{R}^3 \tag{3.14}$$

Die Berücksichtigung der Zeitkoordinate beansprucht eine zusätzliche Dimension in der Darstellung. Die Abbildung einer zweidimensionalen Funktion $g : \mathcal{D}(g) \subset \mathbb{R}^2 \times [0, T] \to \mathbb{R}$ nach diesem System ist nicht mehr praktikabel, da der Graph der Funktion eine Teilmenge des $\mathbb{R}^4$ ist.

$$G_g = \{(x, y, t, g(x, y, t))|(x, y, t) \in \mathcal{D}(g)\} \subset \mathbb{R}^4 \tag{3.15}$$

Das Problem läßt sich aber einfach durch Berücksichtigung der realen Zeit als Darstellungskoordinate beheben. Im Grunde ist diese Sichtweise auch natürlich.

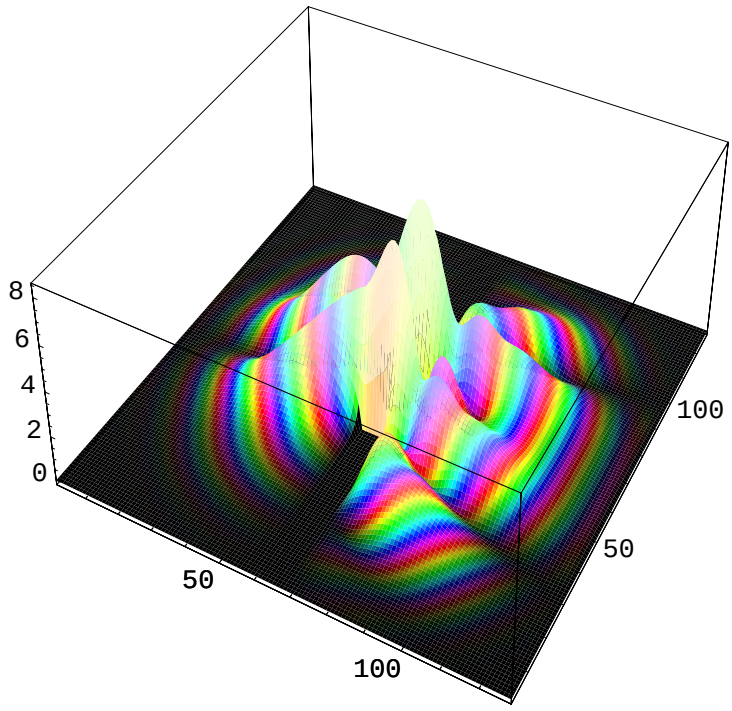

Abbildung 3.6: Dreidimensionale RGB-Abbildung

Der Graph einer zeitabhängigen, zweidimensionalen Funktion kann als Gesamtheit von Momentaufnahmen verstanden werden. Eine solche Animation zur Darstellung des Graphen kann zum Beispiel mit dem Programmpaket Mathematica erzeugt und wiedergegeben werden.

Der Ablauf eines Streuexperimentes kann damit anhand eines Filmes nachvollzogen werden. Im letzten Kapitel werden verschiedene Experimente zur Schrödinger-, Pauli- und Dirac-Theorie beschrieben, die diese Darstellungsmethode verwenden.

3.3 Höherdimensionale Graphen

Bisher wurden nur zweidimensionale Funktionen für die Visualisierung in Betracht gezogen. Die Visualisierung von dreidimensionalen Funktionen und die Berücksichtigung von deren Zeitabhängigkeit wirft neue Probleme auf. Der Graph einer dreidimensionalen Funktion $h : \mathcal{D}(h) \subset \mathbb{R}^3 \to \mathbb{R}$, ohne Berücksichtigung der Zeitabhängigkeit, ist eine Teilmenge des $\mathbb{R}^4$.

$$G_h = \{(x, y, z, h(x, y, z))|(x, y, z) \in \mathcal{D}(h)\} \subset \mathbb{R}^4 \tag{3.16}$$

Eine zum zweidimensionalen Fall analoge Vorgangsweise zur Darstellung der Funktion ist die Projektion des Graphen auf den Definitionsbereich $\mathcal{D}(h)$, wobei jeder Punkt $(x, y, z) \in \mathcal{D}(h)$ entsprechend dem Funktionswert $h(x, y, z)$ eingefärbt wird. Für eine reelle Funktion kann zum Beispiel die Rot-Blau-

Abbildung verwendet werden. Nun bleibt aber noch das Problem, die dreidimensionale Farbwolke geeignet abzubilden. Die wohl einfachste Methode ist, zweidimensionale Schnitte durch die Farbwolke zu legen und diese darzustellen. Aufgrund der Einfachheit dieser Methode wird sie für die Darstellung von Wellenfunktionen, Spinoren und Bispinoren in den Simulationen verwendet. Einen Überblick über weitere Methoden zur Volumen-Visualisierung, wie zum Beispiel dem Ray-Tracing, liefert [14].

Mit der Methode der Schnitte ist die Darstellung einer zeitabhängigen, dreidimensionalen Funktion $h : \mathcal{D}(h) \subset \mathbb{R}^3 \times [0, T] \to \mathbb{R}$ kein Problem mehr. Der Graph

$$G_h = \{(x, y, z, t, h(x, y, z, t))|(x, y, z, t) \in \mathcal{D}(h)\} \subset \mathbb{R}^5, \tag{3.17}$$

eine im $\mathbb{R}^5$ eingebettete vierdimensionale Mannigfaltigkeit, kann als Zusammenfassung von zweidimensionalen, animierten Schnitten dargestellt werden.

Kapitel 4

Implementierung

4.1 Objektorientiertes Design

Der Entwurf eines objektorientierten Programmes umfaßt idealerweise drei Stufen. Zielsetzung der ersten Stufe ist ein klares Verständnis der Problemstellung (Analyse). Danach steht die Identifikation der wesentlichen Konzepte im Vordergrund (Design) und schließlich die Umsetzung der Lösung in das Programm (Programmierung). Die erste Phase des Entwurfes ist für ein Simulationsprogramm, das Probleme aus der Quantenmechanik behandelt, bereits erledigt. In den vorangegangenen Kapiteln wurden die wichtigsten Aspekte einer quantenmechanischen Simulation beschrieben. Sowohl die numerische Seite, als auch die Thematik der Visualisierung. Die zweite Stufe des Entwurfprozesses ist Gegenstand dieses Kapitels, die letzte Stufe wird hingegen nur exemplarisch im Anhang wiedergegeben. Eine umfangreiche Darstellung zum Entwurf von objektorientierten Programmen findet der interessierte Leser bei Booch [16].

Im Rahmen der objektorientierten Programmierung ist in der Design-Phase die *Abstraktion* der Problemstellung in Klassen und Objekte das Ziel der Bemühungen. Welche Klassen und Objekte können im Zusammenhang mit einer numerischen Simulation eines Quantensystems und dessen Visualisierung identifiziert werden? Der nächste Abschnitt gibt Aufschluß darüber.

4.1.1 Klassenhierarchie

Die Klassenhierarchie des Simulationsprogrammes faßt die wesentlichen Konzepte, die für die Beschreibung einer Simulation notwendig sind, zusammen. Die axiomatische Betrachtung der Quantenmechanik ist für die Identifikation der Klassen nützlich. In dieser Sichtweise gibt es einen Hilbertraum, in dem das quantenmechanische Problem eingebettet ist. Ein Zustandsvektor ist ein Element aus diesem Raum. Es gibt Operatoren, die Zustände aufeinander abbilden. In diesem Bild treten die wichtigsten Konzepte, die für eine Simulation modelliert werden müssen, klar hervor. So modelliert die Klasse `TFunction` das Konzept eines Zustandsvektors, einer Wellenfunktion, und die Klasse `TOperator` das Konzept eines Operators. Die Eigenschaften des Hilbertraumes, speziell seine Geometrie, werden mit der Klasse `TDomain` eingefangen.

Das Konzept eines Operators ist sehr allgemein. Jedoch erfordern die Simulationen verschiedene spezielle Hamilton-Operatoren. Die Klasse `TOperator`

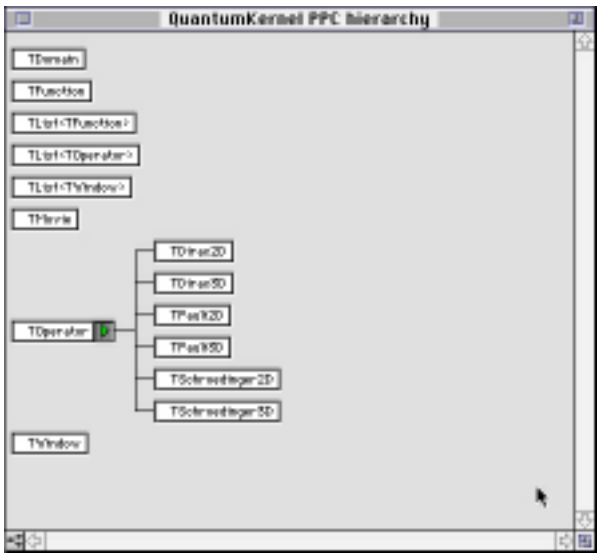

Abbildung 4.1: Hierarchiediagramm

trägt diesem Sachverhalt als abstrakte Klasse Rechnung. Die konkreten Klassen `TSchroedinger2D`, `TSchroedinger3D`, `TPauli2D`, `TPauli3D`, `TDirac2D` und `TDirac3D` verwenden sie als Interface für verschiedene allgemeine Aktionen, wie der Berechnung des Evolutionsoperators und Anwendung des Operators auf eine Wellenfunktion.

Neben diesen primären Klassen für die numerische Simulation eines Quantensystems sind noch Klassen für die Umsetzung der Visualisierung notwendig. Die Darstellung einer Wellenfunktion erfolgt im Kontext einer grafischen Benutzeroberfläche. Die Klasse `TWindow` definiert das Konzept eines Darstellungsfensters. In Verbindung mit der Darstellung der Zeitevolution einer Wellenfunktion ist es notwendig, die zeitliche Entwicklung eines Zustandsvektors in einer Animation festzuhalten. Die Klasse `TMovie` stellt die entsprechende Funktionalität zur Verfügung.

Die beschriebenen Klassen reichen im Grunde aus, um eine Simulation zu beschreiben. Die Integration einer flexiblen Steuerung von mehreren Simulationen erfordert es jedoch, zusätzliche Klassen zur Verwaltung der einzelnen Elemente einer Simulation zu definieren. Die parametrische Klasse `TList` stellt dazu das Konzept einer Liste zur Verfügung. In der Implementierung treten die Klassen `TList<TFunction>`, `TList<TOperator>` und `TList<TWindow>` auf.

Ein Überblick über die gesamte Klassenhierarchie für das Simulationsprogramm ist in Abbildung 4.1 zu sehen. Ein Hierarchiediagramm zeigt die verschiedenen Klassen, mit deren Hilfe ein Problem modelliert werden kann. Es liefert jedoch keine Aussage über das Zusammenspiel der Objekte, die aus den Klassen instanziert werden, um die Funktionalität des Programms zu realisieren. Im nächsten Abschnitt wird auf diese Problematik eingegangen.

4.2 QuantumKernel

Die Funktionalität des Simulationsprogrammes muß, um eine interaktive Bearbeitung einer Simulation zu ermöglichen, dem Benutzer in transparenter Weise zugänglich sein. Das heißt, es ist ein Mechanismus notwendig, mit dessen Hilfe Objekte wie Funktionen und Operatoren dynamisch erzeugt werden können und Aktionen wie die Berechnung der Zeitevolution auf diesen Objekten ausgeführt werden können. Für das betrachtete Simulationsprogramm wird die Kommunikation zwischen Benutzer und Programm mit dem Programmpaket Mathematica realisiert. Mathematica dient als Interface zum Simulationsprogramm, dem QuantumKernel. Der Name QuantumKernel für das Simulationsprogramm wird durch die Analogie zum MathKernel motiviert. Das heißt, vom Standpunkt des Benutzers aus gesehen, verhält sich der QuantumKernel ähnlich wir der MathKernel. Befehle an den QuantumKernel können in gewohnter Mathematica-Notation angegeben und ausgeführt werden. Eine Simulation kann damit als Notebook formuliert und interaktiv bearbeitet werden. Technisch wird die Kommunikation zwischen Mathematica und dem QuantumKernel über das MathLink-Protokoll [20] realisiert. Das Protokoll erlaubt einen Datenaustausch in beide Richtungen.

Für den Aufbau einer Simulation können vom Benutzer verschiedene Objekte im QuantumKernel erzeugt werden. Konkret sind es Objekte vom Typ `TFunction`, `TOperator` und `TWindow`. Intern werden alle diese Objekte in drei Listen verwaltet. Die Listen sind Instanzen der Klassen `TList<TFunction>`, `TList<TOperator>` und `TList<TWindow>`. Die restlichen im letzten Abschnitt beschriebenen Klassen `TDomain` und `TMovie` werden implizit instanziert und sind nicht direkt für den Benutzer zugänglich.

Die Klasse `TFunction` ermöglicht die Definition von Funktionen. Eine Wellenfunktion, ein Spinor, ein Vektorpotential oder ein skalares Potential wird als verschachtelte Liste, je nach Dimension des betrachteten Raumes und Anzahl der Komponenten, an den QuantumKernel übergeben. Als Ergebnis wird eine Referenz auf die Funktion, ein `TFunction`-Objekt im QuantumKernel, an Mathematica zurückgegeben. Der selbe Mechanismus ist auch für die Definition von Operatoren, also `TOperator`-Objekten, implementiert. Ein Operator, zum Beispiel der Schrödinger-Operator, wird mit Hilfe von Referenzen auf Funktionen, `TFunction`-Objekten, definiert. Beliebige skalare Potentiale, Vektorpotentiale und Definitionsbereiche können so zu einem Operator zusammengefaßt werden. Der Rückgabewert ist, wie angedeutet, eine Referenz auf ein `TOperator`-Objekt. Der Mathematica-Befehl zur Berechnung der Zeitevolution nimmt als Eingabeparameter Referenzen auf ein `TFunction`-Objekt, die Wellenfunktion, und auf ein `TOperator`-Objekt, den Hamilton-Operator. Die Visualisierung einer Funktion funktioniert über die Definition eines `TWindow`-Objektes. Wie bei der Berechnung der Zeitevolution wird die Verknüpfung mit dem `TFunction`-Objekt mit Hilfe einer Referenz realisiert. Die `TWindow`-Objekte selbst werden ebenfalls über Referenzen verwaltet. Eine detailierte Auflistung aller zur Verfügung stehenden Mathematica-Befehle zur Steuerung des QuantumKernel erfolgt in Abschnitt 4.3.

4.2.1 Templates

Die technische Ausführung der Integration des objektorientierten Simulationsprogrammes in das Mathematica-System mit Hilfe des Template-Mechanismus ist Gegenstand dieses Abschnittes. Das Mathematica-System stellt zur Integration externer Programme einen Präprozessor zur Verfügung, der aus einer Vorlage, dem Template, C-Source-Code generiert. Der automatisch erzeugte Code implementiert die gesamte Kommunikation mit Mathematica über das MathLink-Protokoll. Eine genaue Beschreibung des Template-Mechanismus ist im Mathematica-Buch [19] angegeben. Ein Beispiel für ein Template zeigt der folgende Code-Ausschnitt.

```
:Begin:
:Function: NewFunction
:Pattern: NewFunction[ arrays__ ]
:Arguments: { { arrays } }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:
```

Ein Template definiert den Befehlssyntax auf der Mathematica-Seite, die Parameterübergabe und den Namen der C-Funktion, die im externen Programm aufgerufen wird. Im Anhang B.4 findet sich eine Auflistung aller Templates, die für die Integration des Simulationsprogrammes, des QuantumKernel, in das Mathematica-System verwendet werden. Jeder auf diese Weise definierte Mathematica-Befehl steht in Verbindung mit einer C-Routine im QuantumKernel. Der QuantumKernel ist als objektorientiertes System konzipiert und in der Sprache C++ [17] formuliert. Der Übergang vom C-Interface in das objektorientierte System ist über Glue-Routinen realisiert. Diese Strukturierung erlaubt eine saubere Trennung des objektorientierten Anteils des QuantumKernel vom, aus technischen Gründen notwendigen, klassischen prozeduralen Anteil des Programms. Die notwendigen Glue-Routinen sind im Anhang B.2 zusammengefaßt.

4.3 Mathematica-Interface

Die Befehle zur Steuerung des QuantumKernel können grob in drei Kategorien eingeteilt werden. Die Kategorien beziehen sich auf die Objekte auf denen die Befehle operieren. Also `TFunction`, `TOperator` und `TWindow`. In dieser Reihenfolge sind die Befehle, mitsamt einer kurzen Beschreibung, aufgelistet.

4.3.1 TFunction

NewFunction

`NewFunction[ arrays__ ]`

`NewFunction` erzeugt ein `TFunction`-Objekt aus `arrays`, einer oder mehreren Listen von reellen Zahlen. Eine Referenz auf das `TFunction`-Objekt wird als `FunctionObjekt` an Mathematica zurückgegeben.

DisposeFunction

```
DisposeFunction[ function_ ]
```

`DisposeFunction` löscht die numerischen Daten des `TFunction`-Objektes, das durch die Referenz `function`, ein Ausdruck vom Typ `FunctionObject`, identifiziert wird.

FunctionInfo

```
FunctionInfo[ function_ ]
```

`FunctionInfo` liefert Informationen zum `TFunction`-Objekt, das durch die Referenz `function` identifiziert wird.

ValueArray

```
ValueArray[ function_ ]
```

`ValueArray` übergibt die numerischen Daten des `TFunction`-Objektes, identifiziert durch die Referenz `function`, als Liste von reellen Zahlen an Mathematica.

ColorArray

```
ColorArray[ function_ ]
```

`ColorArray` übergibt die numerischen Daten des `TFunction`-Objektes, identifiziert durch die Referenz `function`, als Liste von `RGBColor`-Farbwerten an Mathematica. Erlaubt sind komplexe Funktionen, diese werden mit der RGB-Abbildung transformiert.

GrayArray

```
GrayArray[ function_ ]
```

`GrayArray` übergibt die numerischen Daten des `TFunction`-Objektes, identifiziert durch die Referenz `function`, als Liste von `GrayLevel`-Werten an Mathematica. Erlaubt sind komplexe Funktionen, deren Absolutbetrag zum Quadrat wird mit der Graustufen-Abbildung transformiert.

RedBlueArray

```
RedBlueArray[ function_ ]
```

`RedBlueArray` übergibt die numerischen Daten des `TFunction`-Objektes, identifiziert durch die Referenz `function`, als Liste von `RGBColor`-Farbwerten an Mathematica. Erlaubt sind reelle Funktionen, diese werden mit der Rot-Blau-Abbildung transformiert.

BlackWhiteArray

```
BlackWhiteArray[ function_ ]
```

`BlackWhiteArray` übergibt die numerischen Daten des `TFunction`-Objektes, identifiziert durch die Referenz `function`, als Liste von `GrayLevel`-Werten an Mathematica. Erlaubt sind reelle Funktionen, diese werden mit der Schwarz-Weiß-Abbildung transformiert.

AbsArray

```
AbsArray[ function_ ]
```

`AbsArray` übergibt die numerischen Daten des `TFunction`-Objektes, identifiziert durch die Referenz `function`, als Liste von Absolutbeträgen an Mathematica. Erlaubt sind komplexe Funktionen.

Info

```
Info[ ]
```

`Info` liefert Informationen zum Zustand des QuantumKernel. Informationen zu allen `TFunction`-, `TOperator`- und `TWindow`-Objekten werden aufgelistet.

4.3.2 TOperator

Schroedinger2D

```
Schroedinger2D[ scalar_:None, vector_:None, domain_:None,
                mass_:1, charge_:1, units_:1 ]
```

`Schroedinger2D` erzeugt ein `TSchroedinger2D`-Objekt aus `TFunction`-Objekten, identifiziert durch `scalar`, `vector` und `domain`. Erlaubt sind ein komplexes, skalares Potential, ein zweikomponentiges Vektorpotential und ein reeller Definitionsbereich, positive Werte legen das Simulationsgebiet fest. Die reellen Zahlen `mass`, `charge` und `units` definieren die Masse, die Ladung und die Maschenweite des Gitters. Eine Referenz auf das `TOperator`-Objekt wird als `OperatorObject` an Mathematica zurückgegeben.

Schroedinger3D

```
Schroedinger3D[ scalar_:None, vector_:None, domain_:None,
                mass_:1, charge_:1, units_:1 ]
```

`Schroedinger3D` erzeugt ein `TSchroedinger3D`-Objekt aus `TFunction`-Objekten, identifiziert durch `scalar`, `vector` und `domain`. Erlaubt sind ein komplexes, skalares Potential, ein dreikomponentiges Vektorpotential und ein reeller Definitionsbereich, positive Werte legen das Simulationsgebiet fest. Die reellen Zahlen `mass`, `charge` und `units` definieren die Masse, die Ladung und die Maschenweite des Gitters. Eine Referenz auf das `TOperator`-Objekt wird als `OperatorObject` an Mathematica zurückgegeben.

Pauli2D

```
Pauli2D[ scalar_:None, vector_:None, domain_:None,
         mass_:1, charge_:1, units_:1 ]
```

`Pauli2D` erzeugt ein `TPauli2D`-Objekt aus `TFunction`-Objekten, identifiziert durch `scalar`, `vector` und `domain`. Erlaubt sind ein komplexes, skalares Potential, ein zweikomponentiges Vektorpotential und ein reeller Definitionsbereich, positive Werte legen das Simulationsgebiet fest. Die reellen Zahlen `mass`, `charge` und `units` definieren die Masse, die Ladung und die Maschenweite des Gitters. Eine Referenz auf das `TOperator`-Objekt wird als `OperatorObject` an Mathematica zurückgegeben.

Pauli3D

```
Pauli3D[ scalar_:None, vector_:None, domain_:None,
         mass_:1, charge_:1, units_:1 ]
```

`Pauli3D` erzeugt ein `TPauli3D`-Objekt aus `TFunction`-Objekten, identifiziert durch `scalar`, `vector` und `domain`. Erlaubt sind ein komplexes, skalares Potential, ein dreikomponentiges Vektorpotential und ein reeller Definitionsbereich, positive Werte legen das Simulationsgebiet fest. Die reellen Zahlen `mass`, `charge` und `units` definieren die Masse, die Ladung und die Maschenweite des Gitters. Eine Referenz auf das `TOperator`-Objekt wird als `OperatorObject` an Mathematica zurückgegeben.

Dirac2D

```
Dirac2D[ scalar_:None, vector_:None, domain_:None,
         mass_:1, charge_:1, units_:1 ]
```

`Dirac2D` erzeugt ein `TDirac2D`-Objekt aus `TFunction`-Objekten, identifiziert durch `scalar`, `vector` und `domain`. Erlaubt sind ein komplexes, skalares Potential, ein dreikomponentiges Vektorpotential und ein reeller Definitionsbereich, positive Werte legen das Simulationsgebiet fest. Die reellen Zahlen `mass`, `charge` und `units` definieren die Masse, die Ladung und die Maschenweite des Gitters. Eine Referenz auf das `TOperator`-Objekt wird als `OperatorObject` an Mathematica zurückgegeben.

Dirac3D

```
Dirac3D[ scalar_:None, vector_:None, domain_:None,
         mass_:1, charge_:1, units_:1 ]
```

`Dirac3D` erzeugt ein `TDirac3D`-Objekt aus `TFunction`-Objekten, identifiziert durch `scalar`, `vector` und `domain`. Erlaubt sind ein komplexes, skalares Potential, ein vierkomponentiges Vektorpotential und ein reeller Definitionsbereich, positive Werte legen das Simulationsgebiet fest. Die reellen Zahlen `mass`, `charge` und `units` definieren die Masse, die Ladung und die Maschenweite des Gitters. Eine Referenz auf das `TOperator`-Objekt wird als `OperatorObject` an Mathematica zurückgegeben.

DisposeOperator

```
DisposeOperator[ operator_ ]
```

`DisposeOperator` löscht die Daten des `TOperator`-Objektes, das durch die Referenz `operator`, ein Ausdruck vom Typ `OperatorObject`, identifiziert wird.

OperatorInfo

```
OperatorInfo[ operator_ ]
```

`OperatorInfo` liefert Informationen zum `TOperator`-Objekt, das durch die Referenz `operator` identifiziert wird.

TimeEvolution

```
TimeEvolution[ operator_, function_, timestep_,
               fractal_:4, steps_:1 ]
```

`TimeEvolution` berechnet die Zeitevolution mit dem `TOperator`-Objekt, identifiziert durch die Referenz `operator`, und dem `TFunction`-Objekt, identifiziert durch die Referenz `function`. Erlaubt sind beliebige Hamilton-Operatoren mit kompatiblen Wellenfunktionen. Die reelle Zahl `timestep` ist die Länge eines Zeitschrittes. Die natürlichen Zahlen `fractal` und `steps` definieren die Ordnung der fraktalen Approximation und die Anzahl der Zeitschritte.

4.3.3 TWindow

ShowWindow

```
ShowWindow[ function_, mode_:0, slice_:0 ]
```

`ShowWindow` erzeugt ein `TWindow`-Objekt und öffnet ein Darstellungsfenster für das `TFunction`-Objekt, identifiziert durch `function`. Erlaubt sind zwei- und dreidimensionale Funktionen. Die natürlichen Zahlen `mode` und `slice` definieren den Darstellungsmodus (vgl. Tabelle 4.1) und für dreidimensionale Funktionen die z-Koordinate der Schnittebene. Eine Referenz auf das `TWindow`-Objekt wird als `WindowObject` an Mathematica zurückgegeben.

HideWindow

```
HideWindow[ window_ ]
```

`HideWindow` schließt das Darstellungsfenster und löscht das `TWindow`-Objekt, das durch die Referenz `window`, ein Ausdruck vom Typ `WindowObject`, identifiziert wird.

WindowInfo

```
WindowInfo[ window_ ]
```

`WindowInfo` liefert Informationen zum `TWindow`-Objekt, das durch `window` identifiziert wird.

mode	$V(x) \in \mathbb{R}$	$\psi(x) \in \mathbb{C}$	$\mathbf{A}(x) \in \mathbb{R}^3$	$\psi(x) \in \mathbb{C}^2$	$\psi(x) \in \mathbb{C}^4$
0	$T_{RB} \circ V$	$T_{RGB} \circ \psi$	$T_{RB} \circ A_1$	$T_{RGB} \circ \psi_1$	$T_{RGB} \circ \psi_1$
1	$T_{BW} \circ V$	$T_G \circ \lvert\psi\rvert^2$	$T_{RB} \circ A_2$	$T_{RGB} \circ \psi_2$	$T_{RGB} \circ \psi_2$
2		$T_{RB} \circ \mathrm{Re}\,\psi$	$T_{RB} \circ A_3$	$T_{RB} \circ \psi^* \sigma_1 \psi$	$T_{RGB} \circ \psi_3$
3		$T_{RB} \circ \mathrm{Im}\,\psi$		$T_{RB} \circ \psi^* \sigma_2 \psi$	$T_{RGB} \circ \psi_4$
4				$T_{RB} \circ \psi^* \sigma_3 \psi$	$T_{RB} \circ \psi^* \alpha_1 \psi$
5				$T_G \circ \lvert\psi\rvert^2$	$T_{RB} \circ \psi^* \alpha_2 \psi$
6					$T_{RB} \circ \psi^* \alpha_3 \psi$
7					$T_{RB} \circ \psi^* \beta \psi$
8					$T_G \circ \lvert\psi\rvert^2$

Tabelle 4.1: Darstellungsmodus

BeginMovie

```
BeginMovie[ window_ ]
```

`BeginMovie` startet die Aufzeichnung einer Animation im Darstellungsfenster des `TWindow`-Objektes, identifiziert durch `window`.

EndMovie

```
EndMovie[ window_ ]
```

`EndMovie` beendet die Aufzeichnung der Animation im Darstellungsfenster des `TWindow`-Objektes, identifiziert durch `window`.

4.4 Integration

Die Integration des QuantumKernel in das Mathematica-System wird durch die Definition eines Mathematica-Package vervollständigt. Das Package enthält Befehle zur Initialisierung des QuantumKernel und Definitionen für grafische Hilfsfunktionen. Das Package ist im Anhang C.1 abgedruckt. Im FrontEnd des Mathematica-Systems kann der QuantumKernel mit dem Befehl

```
<<QuantumMechanics‘QuantumKernel‘
```

initialisiert und gestartet werden. Die globale Variable `QuantumLink` speichert das `LinkObject`, das Mathematica mit dem QuantumKernel verbindet. Mit dem Befehl

```
Uninstall[QuantumLink]
```

kann die MathLink-Verbindung zum QuantumKernel beendet werden. Beispiele für Anwendungen des Mathematica-Package `QuantumMechanics` folgen im nächsten Kapitel.

Kapitel 5

Simulationen

5.1 Interaktive Simulation

An dieser Stelle stehen alle wesentlichen Hilfsmittel zur Erstellung einer interaktiven Simulation eines quantenmechanischen Systems mit dem Mathematica-System zur Verfügung. Die im Anschluß an diesen Abschnitt vorgestellten Simulationen sollen einen groben Überblick über die Möglichkeiten und die Leistungsfähigkeit des QuantumKernel geben. Die nichtrelativistische Quantenmechanik wird von der Schrödinger- und Pauli-Gleichung abgedeckt. Die relativistische Quantenmechanik von der Dirac-Gleichung. Die einzelnen Simulationen umfassen den Aharonov-Bohm-Effekt, die freie Bewegung eines Teilchens in einer Kugelschale, die Streuung eines Elektrons an einer magnetischen Scheibe, die Spinpräzession eines Elektrons im konstanten Magnetfeld, einen Bindungszustand eines relativistischen Elektrons im konstanten Magnetfeld und die Evolution des Bispinorfeldes in einem kovarianten skalaren Potential. Die Mathematica-Notebooks zu den Simulationen sind im Anhang D abgedruckt.

Die Möglichkeit einer interaktiven Bearbeitung einer Simulation mit dem Mathematica-System ist sehr vorteilhaft. Es kann die gesamte Funktionalität des Mathematica-Systems für die Simulationen eingesetzt werden. Komplizierte Definitionen für Anfangszustände oder Potentiale können elegant formuliert und bearbeitet werden. Die Weiterverarbeitung von Simulationsdaten aus dem QuantumKernel ist ebenfalls via Mathematica kein Problem.

5.1.1 Konventionen

Die Notebooks zu den einzelnen Simulationen folgen einigen praktischen Konventionen. Alle zweidimensionalen Simulationen werden auf einem Einheitsquadrat gerechnet. Alle dreidimensionalen Simulationen in einem Einheitswürfel. In diesem Sinne wird die Größe des Simulationsgitters von der Maschenweite h des Gitternetzes bestimmt. Allgemein gilt der Zusammenhang:

$$n_1 = l_1/h + 1, \quad n_2 = l_2/h + 1, \quad n_3 = l_3/h + 1 \tag{5.1}$$

Die Abmessungen der Simulationsbox $l_1 = l_2 = l_3 = 1$ führen auf die Notwendigkeit einer reziprok ganzzahligen Maschenweite. Wie bei der Implementierung

der Algorithmen festgelegt, ist die Maschenweite unabhängig von der Raumrichtung. In den vorliegenden Simulationen wird die Maschenweite im zweidimensionalen Fall auf $h = 1/127 \approx 0.007874$ festgelegt und im dreidimensionalen Fall auf $h = 1/63 \approx 0.015873$. Damit wird das Einheitsquadrat von einem Gitternetz mit $128 \times 128 = 16384$ Gitterpunkten überzogen. Der Einheitswürfel wird mit $64 \times 64 \times 64 = 262144$ Gitterpunkten approximiert.

Die Konvention in einem Einheitsquadrat oder Einheitswürfel zu arbeiten bietet den Vorteil, daß simulationsspezifische Koordinaten sich immer auf Einheitslängen beziehen. Mit wie vielen Gitterpunkten die Simulationsbox aufgelöst wird, spielt nur eine untergeordnete Rolle. Die Skalierung der Gittergröße ist damit unabhängig vom Simulationsaufbau möglich. So kann sehr einfach durch Anpassung der Maschenweite der Rechenaufwand für eine Simulation vergrößert oder verkleinert werden.

5.2 Schrödinger-Gleichung

Die Simulationsbeispiele zur Schrödinger-Gleichung umfassen den Aharonov-Bohm-Effekt und die Bewegung eines freien Teilchens in einer Kugelschale. Der Aharonov-Bohm-Effekt wird im Rahmen der zweidimensionalen Theorie formuliert und dient als Vorlage für alle weiteren Simulationen. Das zugehörige Mathematica-Notebook wird im Detail besprochen. Als Beispiel für die dreidimensionale Theorie wird die Bewegung in einer Kugelschale herangezogen.

5.2.1 Aharonov-Bohm-Effekt

In der klassischen Elektrodynamik wird die Wirkung eines elektromagnetischen Feldes auf ein geladenes Teilchen *lokal* durch die Feldstärken $\mathbf{E}, \mathbf{B}$ vermittelt. Die elektromagnetischen Potentiale $\phi, \mathbf{A}$ dienen als mathematische Hilfsgrößen in der Formulierung der Theorie.

$$\mathbf{E}(\mathbf{x}, t) = -\nabla\phi(\mathbf{x}, t) - \frac{\partial}{\partial t}\mathbf{A}(\mathbf{x}, t) \tag{5.2}$$

$$\mathbf{B}(\mathbf{x}, t) = \nabla \times \mathbf{A}(\mathbf{x}, t) \tag{5.3}$$

Den Potentialen wird jedoch kein physikalischer Gehalt zugemessen, da sie nicht eindeutig bestimmt sind. Eine Eichtransformation der Gestalt

$$\phi(\mathbf{x}, t) \to \phi(\mathbf{x}, t) + \frac{\partial}{\partial t} g(\mathbf{x}, t) \tag{5.4}$$

$$\mathbf{A}(\mathbf{x}, t) \to \mathbf{A}(\mathbf{x}, t) - \nabla g(\mathbf{x}, t) \tag{5.5}$$

mit der Eichfunktion g hat keine Auswirkungen auf die elektromagnetischen Feldstärken und somit die Bewegung des Teilchens.

Die Unabhängigkeit eines physikalischen Systems von der Eichung findet sich auch in der Quantenmechanik wieder. Eine Eichtransformation eines quantenmechanischen Systems umfaßt aber nicht nur die elektromagnetischen Potentiale, sondern auch die Wellenfunktion.

$$\psi(\mathbf{x}, t) \to \psi(\mathbf{x}, t)\, \mathrm{e}^{-\mathrm{i}eg(\mathbf{x},t)} \tag{5.6}$$

Abbildung 5.1: Simulationsgebiet zum Aharonov-Bohm-Effekt

Aussagen über physikalische Eigenschaften des Systems werden durch diese lokale Phasentransformation nicht verändert. Am einfachsten ist dies an der Wahrscheinlichkeitsdichte $|\psi|^2$ zu sehen.

Trotz dieser Korrespondenz zwischen der klassischen und der quantenmechanischen Theorie gibt es einen Effekt, bei dem die elektromagnetischen Potentiale auf ein geladenes Teilchen wirken, obwohl im betrachteten Raumgebiet kein elektromagnetisches Feld vorhanden ist. Damit ist die Wirkung des elektromagnetischen Feldes in der Quantenmechanik nicht mehr lokal. Im Extremfall bewegt sich das geladene Teilchen immer in einem Raumgebiet, in dem kein elektromagnetisches Feld vorhanden ist und trotzdem gibt es einen Einfluß auf die Bewegung des Teilchens, der von einem elektromagnetischen Feld in einem Raumbereich herrührt, der dem Teilchen nicht zugänglich ist. Dieser Effekt wurde von Aharonov und Bohm [6] aus theoretischen Überlegungen hergeleitet. Wesentlich für diesen Effekt ist die mehrfach zusammenhängende Topologie des Gebietes, in dem sich das Teilchen bewegt.

Die angesprochene Fernwirkung des elektromagnetischen Feldes oder die physikalische Wirkung der elektromagnetischen Potentiale wird nun im Rahmen eines Doppelspaltexperimentes behandelt. Das Besondere an diesem Doppelspalt ist der zentrale Streukörper. Dieser wird als abgeschirmte Spule realisiert. Im Inneren der Spule herrscht ein konstantes magnetisches Feld. Im Außenraum hingegen gibt es kein Feld. Bei einem Interferenzexperiment zeigt sich, daß das Interferenzmuster vom magnetischen Fluß durch die Spule abhängt, obwohl sich das Streuteilchen immer im feldfreien Raum bewegt.

Das Mathematica-Notebook zur Simulation dieses Experimentes zeigt das nachstehende Listing.

```
TDSE2D: AHARONOV-BOHM EFFECT

DOMAIN

Domain2D[h_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    If[ (x1 >= 15/32 && x1 <= 17/32 &&
        (x2 >= 19/32 || x2 <= 13/32)) ||
        (x1-1/2)^2 + (x2-1/2)^2 <= 1/32^2, -1, 1]]}

h = 1/127; n1 = 1/h+1; n2 = 1/h+1;
Di = Array[Domain2D[h], {n2, n1}, 0];
De = NewFunction[Di];
```

```
Dw = ShowWindow[De, 1];

HideWindow[Dw];

BeginMovie[Dw];
EndMovie[Dw];

VECTOR POTENTIAL

Vector2D[h_,Alpha_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    I Alpha / ((x1-1/2) - I(x2-1/2))]}

Alpha = 1/3;
Ai = Array[Vector2D[h, Alpha], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai]];

Aw = ShowWindow[Ae, 0];

HideWindow[Aw];

BeginMovie[Aw];
EndMovie[Aw];

WAVEFUNCTION

GaugeGauss2D[h_,e_,Alpha_,k1_,k2_,y1_,y2_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-y1)^2 + (x2-y2)^2)/2]*Exp[I(k1 x1 + k2 x2)]*
    Exp[I e Alpha Arg[(x1-1/2) + I(x2-1/2)]]]}

e = 1; k1 = -16 Pi; k2 = 0 Pi; y1 = 3/4; y2 = 1/2; a = 10; b = 200;
Psii = Array[GaugeGauss2D[h, e, Alpha, k1, k2, y1, y2, a, b], {n2, n1}, 0];
Psie = NewFunction[Re[Psii], Im[Psii]];

Psiw0 = ShowWindow[Psie, 0];
Psiw1 = ShowWindow[Psie, 1];

HideWindow[Psiw0];
HideWindow[Psiw1];

HAMILTON OPERATOR

scalar = None; vector = Ae; domain = De;
mass = 1.; charge = 1.e; units = 1.h;
He = Schroedinger2D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];

EndMovie[Psiw0];
EndMovie[Psiw1];

opts2D = {AspectRatio->n2/n1};
opts3D = {AspectRatio->n2/n1, Mesh->False, PlotRange->All};

RenderScalar2DBlackWhite[De, opts2D];

RenderScalar3DBlackWhite[De, opts3D];

RenderComplex2DGray[Psie, opts2D];

RenderComplex3DGray[Psie, opts3D];

RenderScalar2DRedBlue[NewFunction[Part[ValueArray[Psie],1]], opts2D];

RenderScalar3DRedBlue[NewFunction[Part[ValueArray[Psie],1]], opts3D];
```

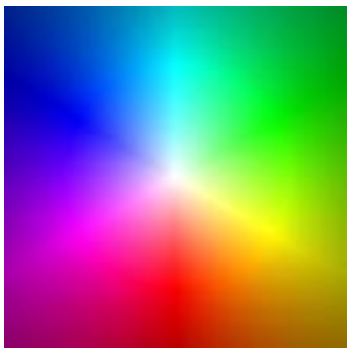

Abbildung 5.2: Vektorpotential einer abgeschirmten Spule

```
RenderComplex2DColor[Psie, opts2D];

RenderComplex3DColor[Psie, opts3D];
```

⌞

Der erste Schritt in der Simulation ist die Definition des Hilbertraumes, also des Bereiches des Einheitsquadrates, der dem Teilchen zugänglich ist. Der Doppelspalt wird mit dieser Definition realisiert. Die Blöcke und die zentrale Kreisscheibe werden aus dem Einheitsquadrat herausgestanzt. Das verbleibende Gebiet ist nun topologisch mehrfach zusammenhängend.
⌜

```
DOMAIN

Domain2D[h_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    If[ (x1 >= 15/32 && x1 <= 17/32 &&
        (x2 >= 19/32 || x2 <= 13/32)) ||
        (x1-1/2)^2 + (x2-1/2)^2 <= 1/32^2, -1, 1]]}

h = 1/127; n1 = 1/h+1; n2 = 1/h+1;
Di = Array[Domain2D[h], {n2, n1}, 0];
De = NewFunction[Di];

Dw = ShowWindow[De, 1];

HideWindow[Dw];

BeginMovie[Dw];
EndMovie[Dw];
```

⌞

Die Teilmenge des Simulationsgebietes, auf dem die Funktion `Domain2D` positiv ist, definiert das Grundgebiet, also den für das Teilchen zugänglichen Bereich. Mit `Domain2D` wird ein zweidimensionales Feld erzeugt und an den QuantumKernel mit der Funktion `NewFunction` übergeben. Die verbleibenden Funktionen dienen der Visualisierung des Grundgebietes. Eine Darstellung des Definitionsbereiches mit der Schwarz-Weiß-Abbildung ist in Abbildung 5.1 zu sehen.

Der nächste Schritt ist die Definition des Vektorpotentials der Spule im Zentrum des Simulationsgebietes. Das Vektorpotential einer Spule mit Radius R

und Flußparameter α lautet:

$$\mathbf{A}(\mathbf{x}) = \begin{cases} \frac{\alpha}{R^2}(-x_2, x_1), & |\mathbf{x}| \leq R \\ \frac{\alpha}{x_1^2+x_2^2}(-x_2, x_1), & |\mathbf{x}| > R \end{cases} \tag{5.7}$$

Der Flußparameter α ist proportional zum magnetischen Fluß Φ durch die Spule.

$$2\pi\alpha = \Phi = BR^2\pi \tag{5.8}$$

Der genaue Potentialverlauf im Inneren der Spule ist für die Simulation ohne Bedeutung, da das Streuteilchen nicht in diesen Bereich eindringen kann. Aus diesem Grund wird in der Simulation nur das Potential im Außenraum berücksichtigt.

```
VECTOR POTENTIAL

Vector2D[h_,Alpha_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    I Alpha / ((x1-1/2) - I(x2-1/2))]}

Alpha = 1/3;
Ai = Array[Vector2D[h, Alpha], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai]];

Aw = ShowWindow[Ae, 0];

HideWindow[Aw];

BeginMovie[Aw];
EndMovie[Aw];
```

Die Funktion `Vector2D` beschreibt das Vektorpotential als komplexe Funktion. Real- und Imaginärteil entsprechen aber der Definition des Potentials von oben. Eine Darstellung des Vektorpotentials ist in Abbildung 5.2 zu sehen. In diesem Bild ist das Vektorpotential mit der RGB-Farbtransformation abgebildet. Die Farben entsprechen den Richtungen der Potentialvektoren. Die Feldlinien bilden Kreise.

Die Definition der Wellenfunktion wird im nächsten Schritt vorgenommen. Die Idee zur Definition des Anfangszustandes ist ein Teilchen, das sich von links auf den Doppelspalt zu bewegt. Die Wahrscheinlichkeitsdichte wird somit durch eine lokalisierte Gaußverteilung realisiert. Die Definition der komplexen Phase des Anfangszustandes erfordert eine kurze Überlegung. Wie aus der klassischen Mechanik bekannt gilt für ein Teilchen im elektromagnetischen Feld die Beziehung:

$$m\mathbf{v} = \mathbf{p} - e\mathbf{A} \tag{5.9}$$

Das heißt, die Geschwindigkeit des Teilchens ist nicht nur vom kanonischen Impuls abhängig, sondern auch vom Vektorpotential. Auch in der Quantenmechanik muß diesem Sachverhalt Rechnung getragen werden.

In einem einfach zusammenhängenden Gebiet, in dem kein Magnetfeld existiert, gilt $\nabla \times \mathbf{A} = 0$. Damit gibt es eine Funktion g mit $\nabla g = \mathbf{A}$. Mit dieser Eichfunktion kann eine Eichtransformation auf $\mathbf{A} = 0$ durchgeführt werden. Die Transformation eines Anfangszustandes ψ_0 aus der freien Theorie ist damit einfach durch $\psi = \psi_0 \mathrm{e}^{\mathrm{i}eg}$ gegeben. Zu beachten ist, daß in diesem Fall die Eichfunktion $-g$ ist.

Im vorliegenden Fall gibt es im Außenraum der Spule kein Magnetfeld. Jedoch ist das Gebiet mehrfach zusammenhängend. Das hat zur Folge, daß die Funktion $\psi = \psi_0 \mathrm{e}^{\mathrm{i}eg}$ im allgemeinen nicht mehr eindeutig ist und damit auch kein zulässiger Anfangszustand für die Schrödinger-Gleichung. Die Eichfunktion für die Spule ist:

$$g(\mathbf{x}) = \alpha \arg(x_1 + \mathrm{i}x_2) \tag{5.10}$$

Die arg-Funktion, als komplexe Funktion aufgefaßt, ist auf einer unendlich blättrigen Riemannschen Fläche definiert. Der Übergang zwischen den Blättern ist typischerweise entlang der negativen reellen Halbachse festgelegt. Damit stehen zwei Möglichkeiten offen, die Mehrdeutigkeit in der Wellenfunktion $\psi = \psi_0 \mathrm{e}^{\mathrm{i}eg}$ zu beseitigen. Für den Fall, daß der Flußparameter α eine ganze Zahl ist, wird die Mehrdeutigkeit durch die Periodizität der komplexen Exponentialfunktion kompensiert. Ist aber α keine ganze Zahl, dann muß die Wellenfunktion ψ_0 auf der negativen x_1-Halbachse eine Knotenlinie besitzen, also dort verschwinden, um eine eindeutige Definition sicher zu stellen.

Mit diesen Betrachtungen kann ein Gaußpaket aus der freien Theorie in das Eichfeld übernommen werden.

$$\psi_0(\mathbf{x}) = a\mathrm{e}^{-b((x_1-y_1)^2+(x_2-y_2)^2)/2}\mathrm{e}^{\mathrm{i}(k_1x_1+k_2x_2)} \tag{5.11}$$

$$\psi(\mathbf{x}) = \psi_0(\mathbf{x})\mathrm{e}^{\mathrm{i}e\alpha \arg(x_1+\mathrm{i}x_2)} \tag{5.12}$$

Genau genommen müßte das Gaußpaket bei einem bestimmten Radius abgeschnitten werden, um auf der negativen x_1-Halbachse exakt zu verschwinden. Für die praktische Rechnung ist dies allerdings nicht notwendig, da die Wellenfunktion im Bereich der negativen x_1-Halbachse hinreichend klein ist und die Numerik nicht stört.

Die aus diesen Überlegungen resultierende Wellenfunktion wird im Notebook in der Funktion `GaugeGauss2D` codiert.

```
WAVEFUNCTION

GaugeGauss2D[h_,e_,Alpha_,k1_,k2_,y1_,y2_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-y1)^2 + (x2-y2)^2)/2]*Exp[I(k1 x1 + k2 x2)]*
    Exp[I e Alpha Arg[(x1-1/2) + I(x2-1/2)]]]}

e = 1; k1 = -16 Pi; k2 = 0 Pi; y1 = 3/4; y2 = 1/2; a = 10; b = 200;
Psii = Array[GaugeGauss2D[h, e, Alpha, k1, k2, y1, y2, a, b], {n2, n1}, 0];
Psie = NewFunction[Re[Psii], Im[Psii]];

Psiw0 = ShowWindow[Psie, 0];
Psiw1 = ShowWindow[Psie, 1];

HideWindow[Psiw0];
HideWindow[Psiw1];
```

Die Erzeugung eines zweidimensionalen Feldes und die Übergabe der numerischen Daten an den QuantumKernel erfolgt in gewohnter Weise. Die Visualisierung der Wellenfunktion mit der Graustufen-Abbildung für $|\psi|^2$ und der RGB-Abbildung für ψ wird mit der Funktion `ShowWindow` realisiert. Die verschiedenen Darstellungsmodi von `ShowWindow` sind in der Tabelle 4.1 zusammengefaßt.

Mit diesen Vorbereitungen kann der Schrödinger-Operator definiert werden. Die Mathematica-Schreibweise lehnt sich dabei in natürlicher Weise an die Definition des Hamilton-Operators eines quantenmechanischen Systems an.

```
HAMILTON OPERATOR

scalar = None; vector = Ae; domain = De;
mass = 1.; charge = 1.e; units = 1.h;
He = Schroedinger2D[scalar, vector, domain, mass, charge, units];
```

Das Vektorpotential und der Definitionsbereich werden als Referenzen an die Funktion `Schroedinger2D` übergeben. Das skalare Potential wird mit der Referenz `None` aus der Definition des Schrödinger-Operators ausgespart. Weitere Argumente sind die Masse, die Ladung und die Maschenweite.

Die Definition des Hamilton-Operators vervollständigt den Simulationsaufbau. Die Berechnung der Zeitentwicklung der Wellenfunktion wird mit der Funktion `TimeEvolution` durchgeführt.

```
TIME EVOLUTION

t = 4.h^2; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];
```

Neben dem Schrödinger-Operator und der Wellenfunktion gehen als Argumente die Länge eines Zeitschrittes, die Ordnung des fraktalen Algorithmus und die Anzahl der Zeitschritte für den Simulationslauf ein. Die Skalierung des Zeitschrittes mit h^2 ist aufgrund der parabolischen Struktur des Schrödinger-Operators notwendig. Der Vorfaktor 4 ist eine empirische Stabilitätsgrenze für einen fraktalen Algorithmus sechster Ordnung. Die gesamte Simulation umfaßt 32 Zeitschritte. Dies entspricht einem Zeitintervall von $T \approx 0.007936$ Zeiteinheiten.

Der verbleibende Teil des Notebooks dient der Visualisierung der Zeitentwicklung der Wellenfunktion als Computer-Animation. Die Befehle `BeginMovie` und `EndMovie` werden zu Beginn und am Ende einer Filmaufzeichnung aufgerufen. Die übrigen Funktionen nutzen die Funktionalität von Mathematica zur Darstellung der Wellenfunktion als Computer-Grafik. Einige Beispiele sind im Abschnitt 3.1 abgedruckt.

```
VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];

EndMovie[Psiw0];
EndMovie[Psiw1];

opts2D = {AspectRatio->n2/n1};
opts3D = {AspectRatio->n2/n1, Mesh->False, PlotRange->All};

RenderScalar2DBlackWhite[De, opts2D];

RenderScalar3DBlackWhite[De, opts3D];

RenderComplex2DGray[Psie, opts2D];

RenderComplex3DGray[Psie, opts3D];
```

```
RenderScalar2DRedBlue[NewFunction[Part[ValueArray[Psie],1]], opts2D];

RenderScalar3DRedBlue[NewFunction[Part[ValueArray[Psie],1]], opts3D];

RenderComplex2DColor[Psie, opts2D];

RenderComplex3DColor[Psie, opts3D];
```

Die Abbildungen zum Aharonov-Bohm-Effekt zeigen Streuexperimente mit verschiedenen Flußparametern. Die Grafiken in Abbildung 5.3 stellen zwei Wellenfunktionen nach der Zeit $T \approx 0.007936$, codiert mit der RGB- und Graustufen-Abbildung, dar. Der Ausgangszustand ist in beiden Fällen ein Gaußpaket, das von links auf den Doppelspalt zu läuft. Die linke Spalte zeigt die Simulation für den Flußparameter $\alpha = 1/2$ und die rechte für $\alpha = 2$. Diese beiden Simulationen zeigen zwei Extrema des Aharonov-Bohm-Effektes. Der halbzahlige Flußparameter verursacht eine maximale Verschiebung des Interferenzmusters gegenüber der Streuung ohne Magnetfeld im Inneren der Spule. Allgemein ist die Verschiebung des Interferenzmusters durch den Winkel

$$\varphi = 2\pi e\alpha = e\Phi \tag{5.13}$$

gegeben. Der Verschiebungswinkel ist proportional zum magnetischen Fluß Φ durch die Spule. In den Simulationen wird für die elektrischen Ladung die Definition $e = 1$ verwendet. Damit entspricht ein halbzahliger Flußparameter dem Winkel π und ein ganzzahliger dem Winkel 2π, also keiner Verschiebung.

Die kontinuierliche Änderung des Interferenzmusters in Abhängigkeit vom Flußparameter wird in Abbildung 5.4 demonstriert. Eine Serie von drittelzahligen Flußparametern $\alpha = 0, 1/3, 2/3, 1$ ist dargestellt. Die erste Zeile zeigt die komplexen Wellenfunktionen und die zweite Zeile die zugehörigen Wahrscheinlichkeitsdichten.

Der zeitliche Ablauf eines Streuexperimentes mit $\alpha = 1/3$ ist in Abbildung 5.5 dargestellt. Die Zeitachse verläuft von oben nach unten. Die linke Spalte zeigt die komplexe Wellenfunktion, die rechte Spalte die Wahrscheinlichkeitsdichte.

Ein typischer Simulationslauf zum Aharonov-Bohm-Effekt benötigt circa 30 Sekunden Rechenzeit auf einem PowerMacintosh mit 80 MHz PowerPC 601 Prozessor. In dieser Zeit ist auch die Visualisierung berücksichtigt. Das bearbeitete Gleichungssystem umfaßt dabei 32768 reelle Variable.

5.2.2 Freie Bewegung in einer Kugelschale

Die dreidimensionale Schrödinger-Gleichung steht nun in Hinblick auf die Integration eines irregulären dreidimensionalen Grundgebietes zur Diskussion. Das angesprochene Grundgebiet umfaßt zwei konzentrische Kugeln. Der Außenraum der größeren Kugel und der Innenraum der kleineren Kugel ist aus dem Definitionsbereich ausgenommen. Im Mathematica-Notebook lautet die Definition:

```
DOMAIN

Domain2D[h_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    If[ (x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2 <= 1/16^2 ||
```

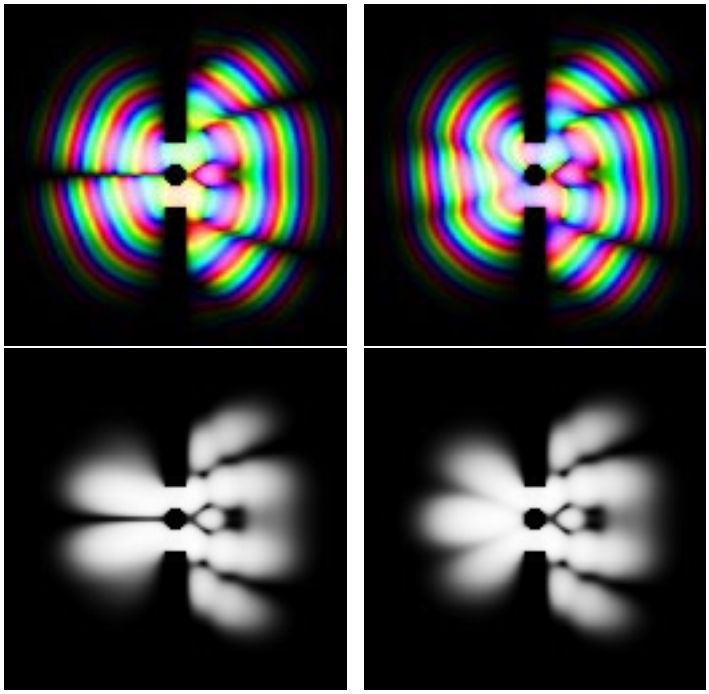

Abbildung 5.3: Aharonov-Bohm-Effekt: $\alpha = 1/2$ und $\alpha = 2$

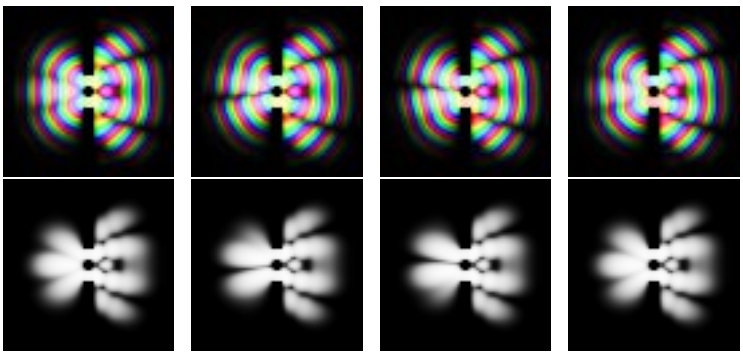

Abbildung 5.4: Aharonov-Bohm-Effekt: $\alpha = 0, 1/3, 2/3, 1$

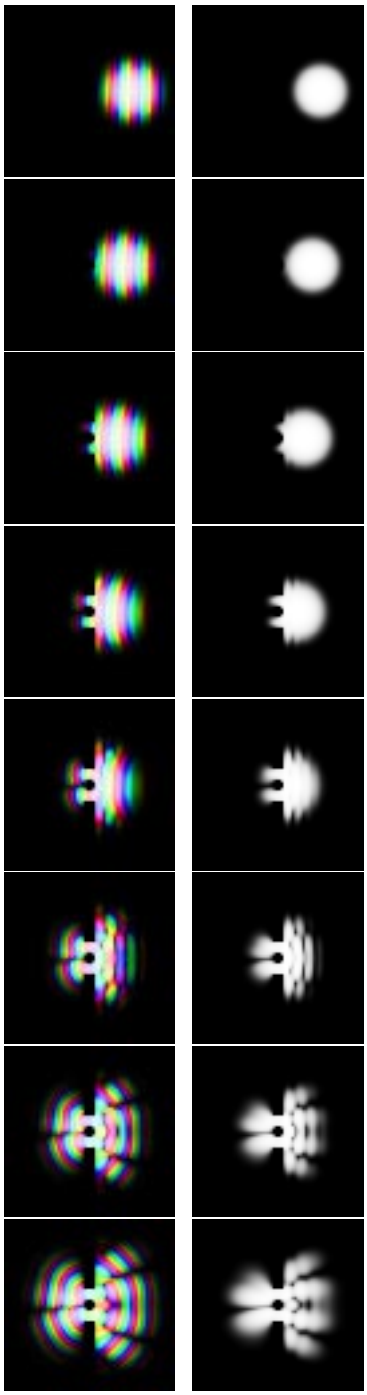

Abbildung 5.5: Aharonov-Bohm-Effekt: $\alpha = 1/3$

```
        (x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2 >= 1/2^2, -1, 1]]}

h = 1/63; n1 = 1/h+1; n2 = 1/h+1; n3 = 1/h+1;
Di = Array[Domain2D[h], {n3, n2, n1}, 0];
De = NewFunction[Di];
```

└

Der Anfangszustand des quantenmechanischen Problems wird als ruhende, dreidimensionale Gaußverteilung gewählt. Das Gaußpaket ist um den Punkt $(3/4, 1/2, 1/2)$ zentriert.
┌

```
WAVEFUNCTION

Gauss3D[h_,a_,b_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    a*Exp[-b((x1-3/4)^2 + (x2-1/2)^2 + (x3-1/2)^2)/2]]}

a = 10; b = 200;
Psii = N[Array[Gauss3D[h, a, b], {n3, n2, n1}, 0]];
Psie = NewFunction[Psii, 0 Psii];

slices = {31, 39, 47, 55};
Psiw0 = ShowWindow[Psie, 0, #]& /@ slices;
Psiw1 = ShowWindow[Psie, 1, #]& /@ slices;

HideWindow[#]& /@ Psiw0;
HideWindow[#]& /@ Psiw1;
```

└

Die Darstellung der dreidimensionalen Wellenfunktion wird mit vier Schnitten durch die Simulationsbox realisiert. Die Schnittebenen verlaufen orthogonal zur x_3-Achse mit den Koordinaten $x_3 = 31/63, 39/63, 47/63, 55/63 \approx 0.4921, 0.6190, 0.7460, 0.8730$.
┌

```
HAMILTON OPERATOR

scalar = None; vector = None; domain = De;
mass = 1.; charge = 1.; units = 1.h;
He = Schroedinger3D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2 2/3; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];
```

└

Die Definition des Hamilton-Operators des Quantensystems umfaßt nur den Definitionsbereich und die üblichen Werte für die Teilchenmasse, die Ladung und die Maschenweite. Die Zeitevolution des Zustandsvektors wird mit einem fraktalen Algorithmus sechster Ordnung approximiert. Im Unterschied zum zweidimensionalen Fall ist die empirische Stabilitätsgrenze 4 mit dem Faktor 2/3 skaliert. Die Norm des Hamilton-Operators vergrößert sich in typischer Weise im Verhältnis zur Dimension des betrachteten Systems. Der Skalierungsfaktor spiegelt dieses Verhältnis zum zweidimensionalen Operator wieder.
┌

```
VISUALIZATION

BeginMovie[#]& /@ Psiw0;
BeginMovie[#]& /@ Psiw1;

EndMovie[#]& /@ Psiw0;
EndMovie[#]& /@ Psiw1;
```

Die Visualisierung der Zeitevolution der Wellenfunktion mit den vier Schnitten ist in den Abbildungen 5.6 und 5.7 zu sehen. Die Schnitte sind in ansteigender Reihenfolge von links nach rechts gezeichnet. Die linke Spalte zeigt also den Schnitt durch die Mitte des Simulationsgebietes. Die Zeitachse verläuft wie gewohnt von oben nach unten. Die erste Serie zeigt die komplexe Wellenfunktion codiert mit der RGB-Abbildung, die zweite die Wahrscheinlichkeitsdichte codiert mit der Graustufen-Abbildung.

Die Rechenzeit für die Simulation beträgt inklusive Visualisierung vier Minuten auf einem 80 MHz PowerPC 601 Prozessor. Das primäre Gleichungssystem umfaßt dabei 524288 reelle Variable. Die Einschränkung des Definitionsbereiches auf die konzentrischen Kugeln verringert allerdings die Größe des effektiv zu berechnenden Gleichungssystems.

5.3 Pauli-Gleichung

Die Simulationen zur Pauli-Gleichung umfassen die Streuung an einer magnetischen Scheibe in zwei Raumdimensionen und die Spinpräzession in einem konstanten Magnetfeld in drei Raumdimensionen.

5.3.1 Streuung an einer magnetischen Scheibe

Das experimentelle Setup umfaßt eine zentrale Spule, die im Inneren ein konstantes Magnetfeld erzeugt. Die Spule ist so gebaut, daß ein geladenes Teilchen ungehindert passieren kann. Im zweidimensionalen Raum entspricht diese Versuchsanordnung einer magnetischen Scheibe im Zentrum des Simulationsgebietes. Das Vektorpotential der Spule wird aus der Definition (5.7) vom Aharonov-Bohm-Effekt übernommen. In Abbildung 5.8 ist das Vektorpotential mit der RGB-Abbildung dargestellt.

```
VECTOR POTENTIAL

Vector2D[h_,Alpha_,R_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    If[ (x1-1/2)^2 + (x2-1/2)^2 <= R^2,
        I Alpha / R^2 ((x1-1/2) + I(x2-1/2)),
        I Alpha / ((x1-1/2) - I(x2-1/2))]]}

h = 1/127; Alpha = 2; R = 1/16; n1 = 1/h+1; n2 = 1/h+1;
Ai = Array[Vector2D[h, Alpha, R], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai]];

Aw = ShowWindow[Ae, 0];

HideWindow[Aw];

BeginMovie[Aw];
EndMovie[Aw];
```

Der Anfangszustand des Streuexperimentes ist ein Gaußpaket, zentriert um den Punkt $(3/4, 1/2)$, mit einem Wellenzahlvektor $(-16, 0)\pi$. Dies entspricht einer Wellenlänge $\lambda = 1/16$ Längeneinheiten. Das Simulationsgebiet wird mit 128 Gitterpunkten in einer Raumrichtung approximiert, damit wird ein Wellenzug mit 8 Gitterpunkten aufgelöst.

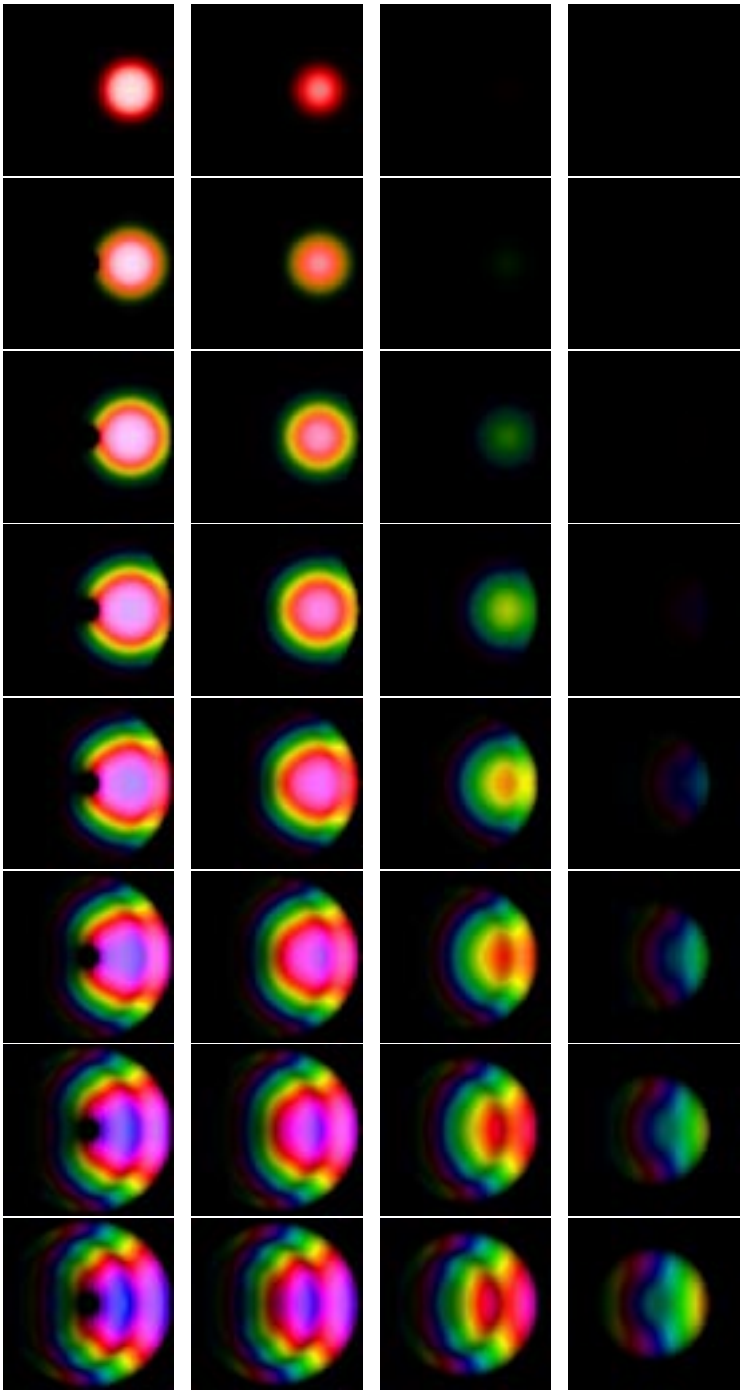

Abbildung 5.6: Freie Bewegung in einer Kugelschale: $T_{RGB} \circ \psi$

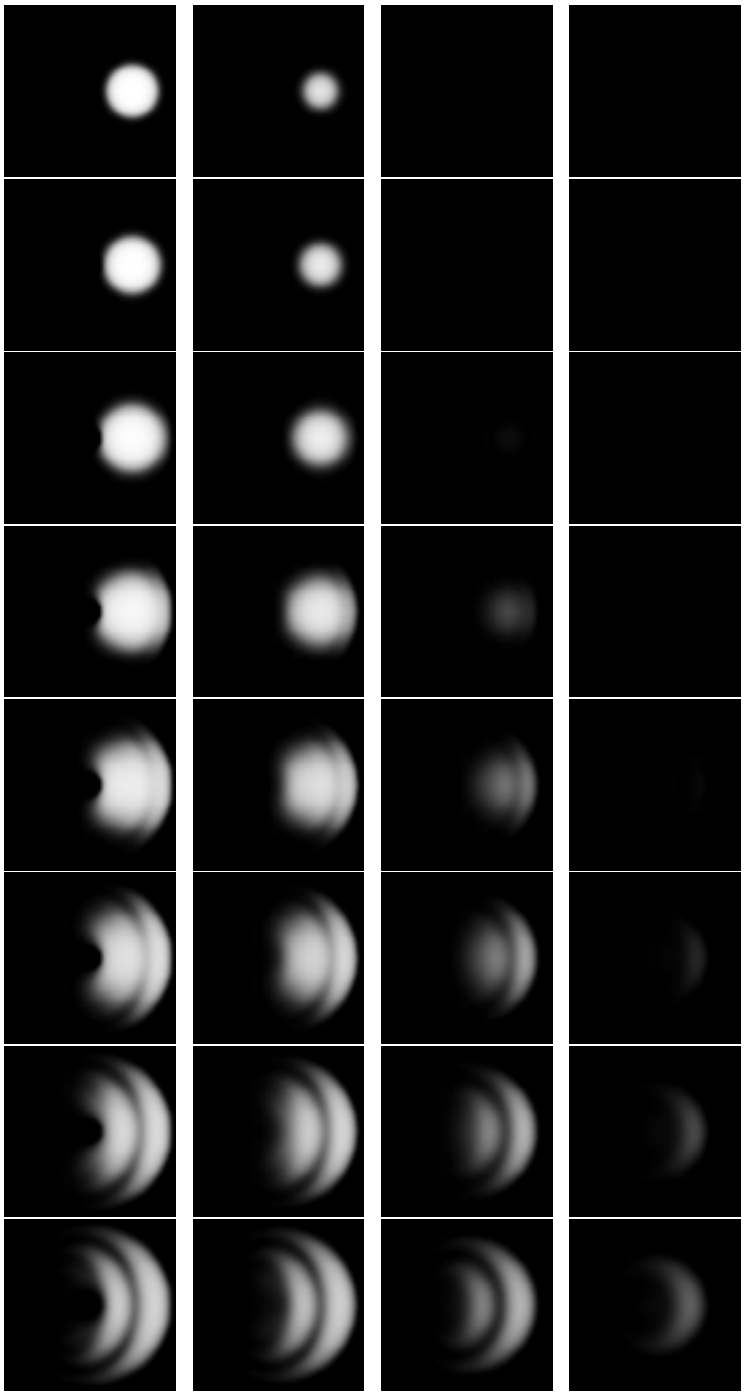

Abbildung 5.7: Freie Bewegung in einer Kugelschale: $T_G \circ |\psi|^2$

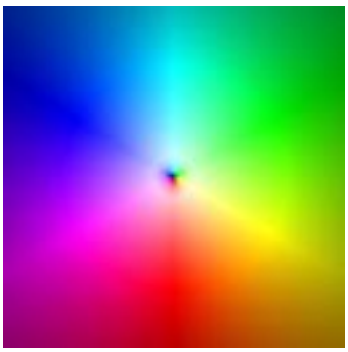

Abbildung 5.8: Vektorpotential einer magnetischen Scheibe

Die Berücksichtigung des Eichpotentials im Außenraum der Spule erfolgt mit Hilfe der Eichfunktion aus Definition (5.10).

```
WAVEFUNCTION

GaugeGauss2D[h_,e_,Alpha_,k1_,k2_,y1_,y2_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-y1)^2 + (x2-y2)^2)/2]*Exp[I(k1 x1 + k2 x2)]*
    Exp[I e Alpha Arg[(x1-1/2) + I(x2-1/2)]]]}

e = 1; k1 = -16 Pi; k2 = 0 Pi; y1 = 3/4; y2 = 1/2; a = 10; b = 200;
Psii = Array[GaugeGauss2D[h, e, Alpha, k1, k2, y1, y2, a, b], {n2, n1}, 0];
Psie = NewFunction[Re[Psii], Im[Psii]];

Psiw0 = ShowWindow[Psie, 0];
Psiw1 = ShowWindow[Psie, 1];

HideWindow[Psiw0];
HideWindow[Psiw1];
```

Der restliche Teil des Simulations-Notebooks zur Definition des Hamilton-Operators, Berechnung der Zeitevolution und Visualisierung folgt dem gewohnten Schema.

```
HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.e; units = 1.h;
He = Pauli2D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];

EndMovie[Psiw0];
EndMovie[Psiw1];
```

Die Ergebnisse von verschiedenen Streuexperimenten sind in einzelnen Bilderserien zusammengefaßt. Die Geschwindigkeit, Lokalisierung und Ausdehnung des Ausgangszustandes werden für alle Simulationen beibehalten. Variiert

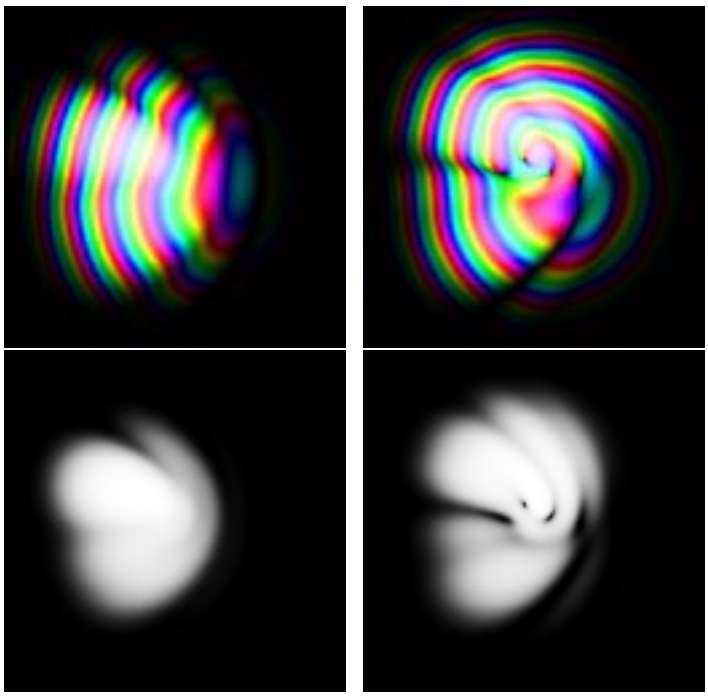

Abbildung 5.9: Streuzustand: $\alpha = 1/2$ und $\alpha = 2$

wird nur der Flußparameter der Spule $\alpha = \Phi/2\pi$, also der magnetische Fluß Φ durch die Spule. Die erste Serie zeigt zwei Experimente mit dem Flußparameter $\alpha = 1/2$ und $\alpha = 2$, vergleiche Abbildung 5.9. Die zweite Serie in Abbildung 5.10 zeigt Experimente mit den Flußparametern $\alpha = 0, 1/3, 2/3, 1$. In der ersten Zeile ist jeweils die komplexe Wellenfunktion gezeichnet und in der zweiten Zeile die dazugehörige Aufenthaltswahrscheinlichkeit. Die verbleibende Serie von Bildern in Abbildung 5.11 beschreibt den zeitlichen Ablauf des Streuexperimentes mit dem Flußparameter $\alpha = 2$. Die Zeitachse verläuft von oben nach unten. Die linke Spalte stellt die komplexe Wellenfunktion dar, die rechte Spalte die Wahrscheinlichkeitsdichte.

Die Rechenzeit für die Simulation entspricht ungefähr der Zeit für die Simulation des Aharonov-Bohm-Effektes, also 30 Sekunden.

5.3.2 Spinpräzession im konstanten Magnetfeld

Die dreidimensionale Pauli-Gleichung bringt das Konzept des Teilchenspins in die Theorie. Die Wellenfunktionen sind Spinoren, also zweikomponentige komplexe Vektoren. Mit der vorgestellten Simulation soll das Verhalten der Spindichte in einem konstanten Magnetfeld untersucht werden. Wie aus der Theorie [11] bekannt, ist im konstanten Magnetfeld der Spinfreiheitsgrad der Wellenfunktion vom Ortsfreiheitsgrad entkoppelt. Im vorliegenden Fall führt die Spindichte eine harmonische Bewegung aus. Der Spindichtevektor dreht sich mit konstanter Winkelgeschwindigkeit um die x_3-Achse. Die vektorielle Spin-

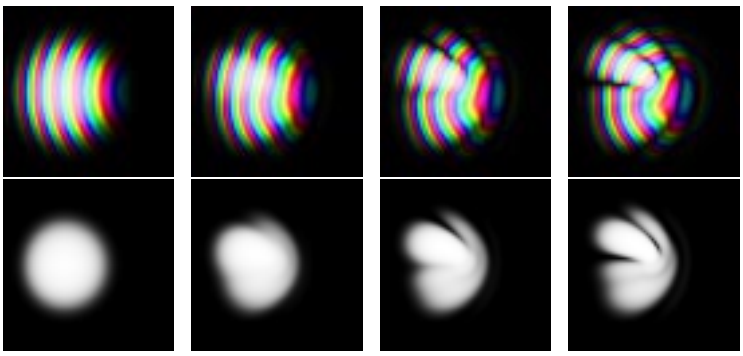

Abbildung 5.10: Streuzustand: $\alpha = 0, 1/3, 2/3, 1$

dichte $\mathbf{s}$ genügt der Definition

$$\mathbf{s} = (\psi^* \sigma_1 \psi, \psi^* \sigma_2 \psi, \psi^* \sigma_3 \psi). \tag{5.14}$$

Die Matrizen σ_1, σ_2 und σ_3 sind die Pauli-Matrizen.

Ein konstantes homogenes Magnetfeld in x_3-Richtung wird im dreidimensionalen Raum durch das Vektorpotential

$$\mathbf{A}(\mathbf{x}) = \frac{B}{2}(-x_2, x_1, 0) \tag{5.15}$$

dargestellt.

```
VECTOR POTENTIAL

Vector3D[h_,B_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    I B/2 ((x1-3/4) + I(x2-1/2))]}

h = 1/63; B = 32 Pi; n1 = 1/h+1; n2 = 1/h+1; n3 = 1/h+1;
Ai = Array[Vector3D[h, B], {n3, n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai], 0 Ai];
```

Der Anfangszustand des Spinorfeldes wird so gewählt, daß der Vektor der Spindichte orthogonal zum Magnetfeld steht. Das Magnetfeld zeigt in x_3-Richtung, die Spindichte in x_1-Richtung. Die räumliche Lokalisierung erfolgt mit einer Gaußverteilung, zentriert um den Punkt$(3/4, 1/2, 1/2)$. Die Problematik der Eichung wird durch die Zentrierung des Vektorpotentials im selben Punkt gelöst. Dadurch wird erreicht, daß das Teilchen im Mittel in Ruhe ist.

```
WAVEFUNCTION

Gauss3D[h_,a_,b_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    a*Exp[-b((x1-3/4)^2 + (x2-1/2)^2 + (x3-1/2)^2)/2]]}

a = 10; b = 200;
Psii = N[Array[Gauss3D[h, a, b], {n3, n2, n1}, 0]];
Psie = NewFunction[Psii, 0 Psii, Psii, 0 Psii];

slices = {31, 39, 47, 55};
Psiw0 = ShowWindow[Psie, 0, #]& /@ slices;
```

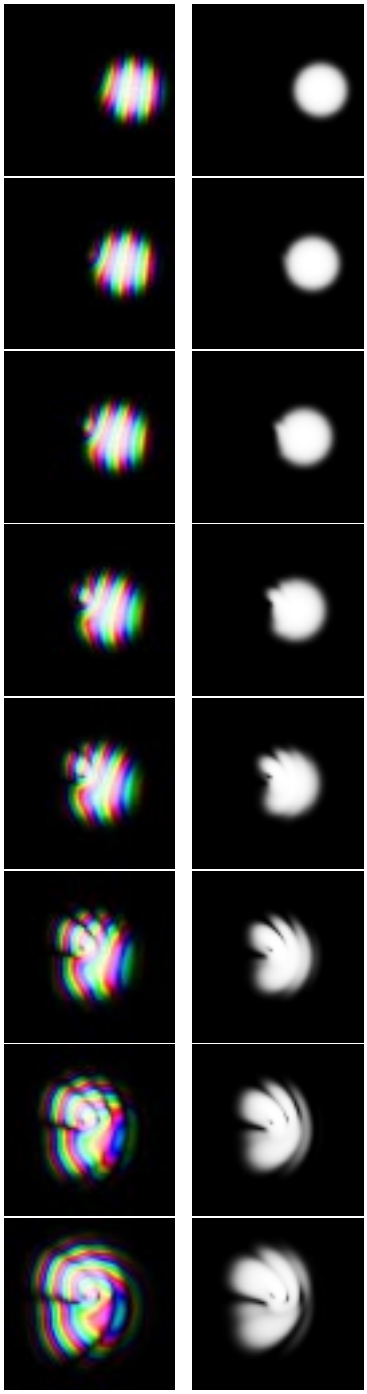

Abbildung 5.11: Streuzustand: $\alpha = 2$

```
Psiw1 = ShowWindow[Psie, 1, #]& /@ slices;
Psiw2 = ShowWindow[Psie, 2, #]& /@ slices;
Psiw3 = ShowWindow[Psie, 3, #]& /@ slices;
Psiw4 = ShowWindow[Psie, 4, #]& /@ slices;
Psiw5 = ShowWindow[Psie, 5, #]& /@ slices;

HideWindow[#]& /@ Psiw0;
HideWindow[#]& /@ Psiw1;
HideWindow[#]& /@ Psiw2;
HideWindow[#]& /@ Psiw3;
HideWindow[#]& /@ Psiw4;
HideWindow[#]& /@ Psiw5;
```

Die Schnitte zur Visualisierung der Wellenfunktion werden wie bei der dreidimensionalen Schrödinger-Gleichung gelegt. Die x_3-Koordinaten der Schnittebenen sind $0.4921, 0.6190, 0.7460, 0.8730$.

Der Hamilton-Operator und die Zeitevolution sind in gewohnter Weise definiert.

```
HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.; units = 1.h;
He = Pauli3D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2 2/3; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[#]& /@ Psiw0;
BeginMovie[#]& /@ Psiw1;
BeginMovie[#]& /@ Psiw2;
BeginMovie[#]& /@ Psiw3;
BeginMovie[#]& /@ Psiw4;
BeginMovie[#]& /@ Psiw5;

EndMovie[#]& /@ Psiw0;
EndMovie[#]& /@ Psiw1;
EndMovie[#]& /@ Psiw2;
EndMovie[#]& /@ Psiw3;
EndMovie[#]& /@ Psiw4;
EndMovie[#]& /@ Psiw5;
```

In den Abbildungen 5.12, 5.13, 5.14, 5.15, 5.16 und 5.17 ist das zeitliche Verhalten des Spinsystems dargestellt. Die aufgezeichneten Animationen umfassen die zwei komplexen Komponenten des Spinorfeldes, die drei Komponenten der Spindichte und die Aufenthaltswahrscheinlichkeitsdichte. Die Komponenten der Spindichte werden als reelle Funktionen mit der Rot-Blau-Abbildung gezeichnet. Die Bilderserien zeigen horizontal die vier Schnittebenen und vertikal die Zeitentwicklung. Der Simulationsablauf umfaßt circa ein Drittel einer Umlaufperiode des Spindichtevektors. Die Entkopplung der räumlichen Bewegung von der harmonischen Bewegung der Spindichte ist deutlich zu erkennen. Die Bilder der Spindichte zeigen eine homogene Farbverteilung unabhängig von der Verteilung der räumlichen Wahrscheinlichkeitsdichte.

Die Simulation umfaßt ein System von über einer Million Gleichungen, exakt 1048576. Die komplizierte Struktur des Hamilton-Operators führt dazu, daß die vom Algorithmus aktuell benötigten Daten trotz einer effizienten Verarbeitung nicht mehr in den Cache-Speicher des Rechners passen. Dies führt

zu einem erheblichen Leistungsverlust im Vergleich zur Schrödinger-Gleichung. Die benötigte Rechenzeit für die Simulation liegt daher bei circa zwei Stunden.

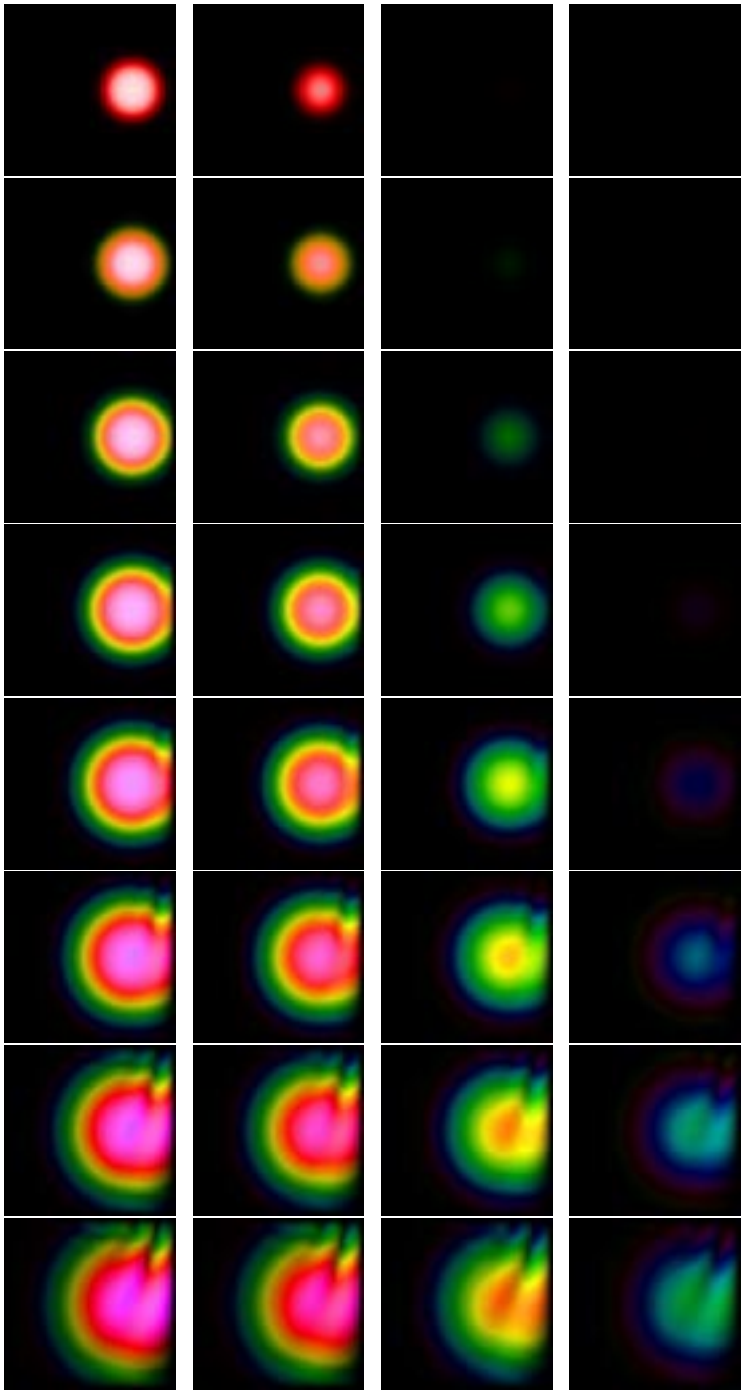

Abbildung 5.12: Spinpräzession: $T_{RGB} \circ \psi_1$

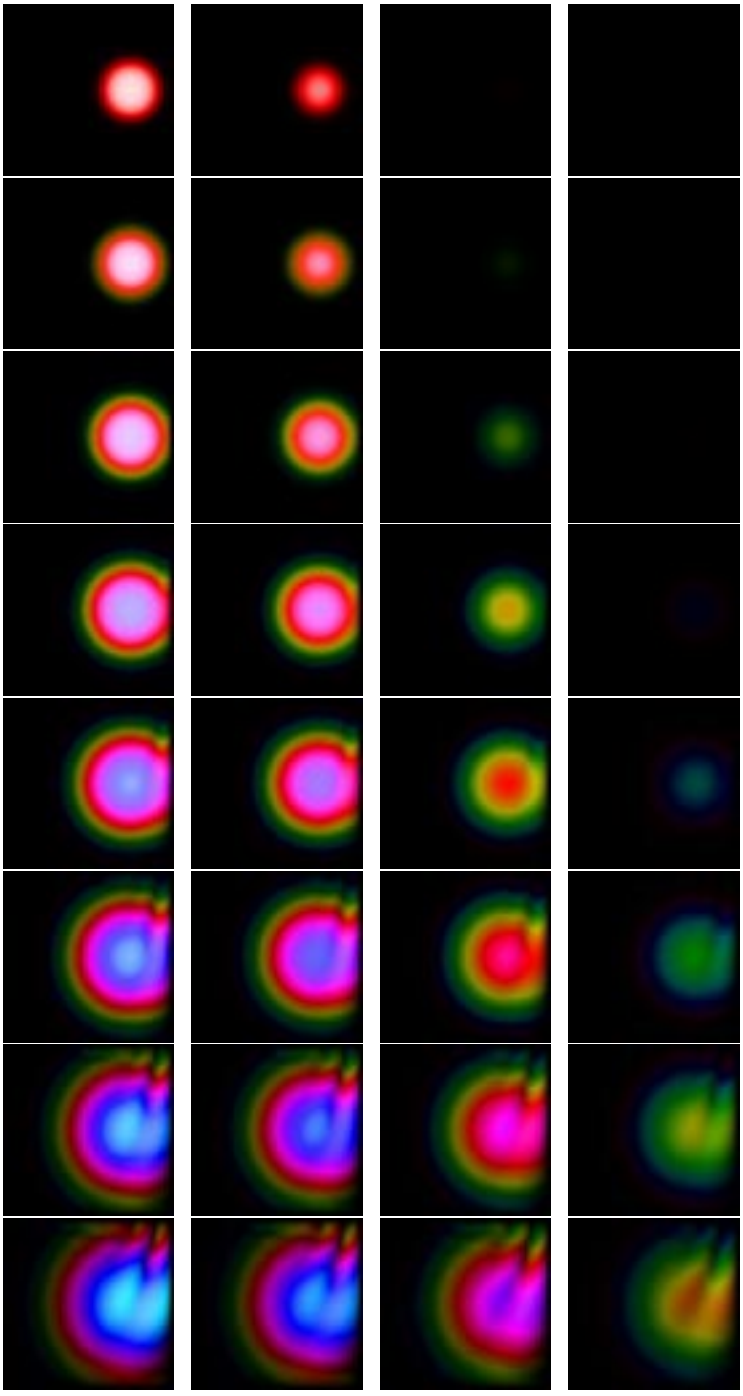

Abbildung 5.13: Spinpräzession: $T_{RGB} \circ \psi_2$

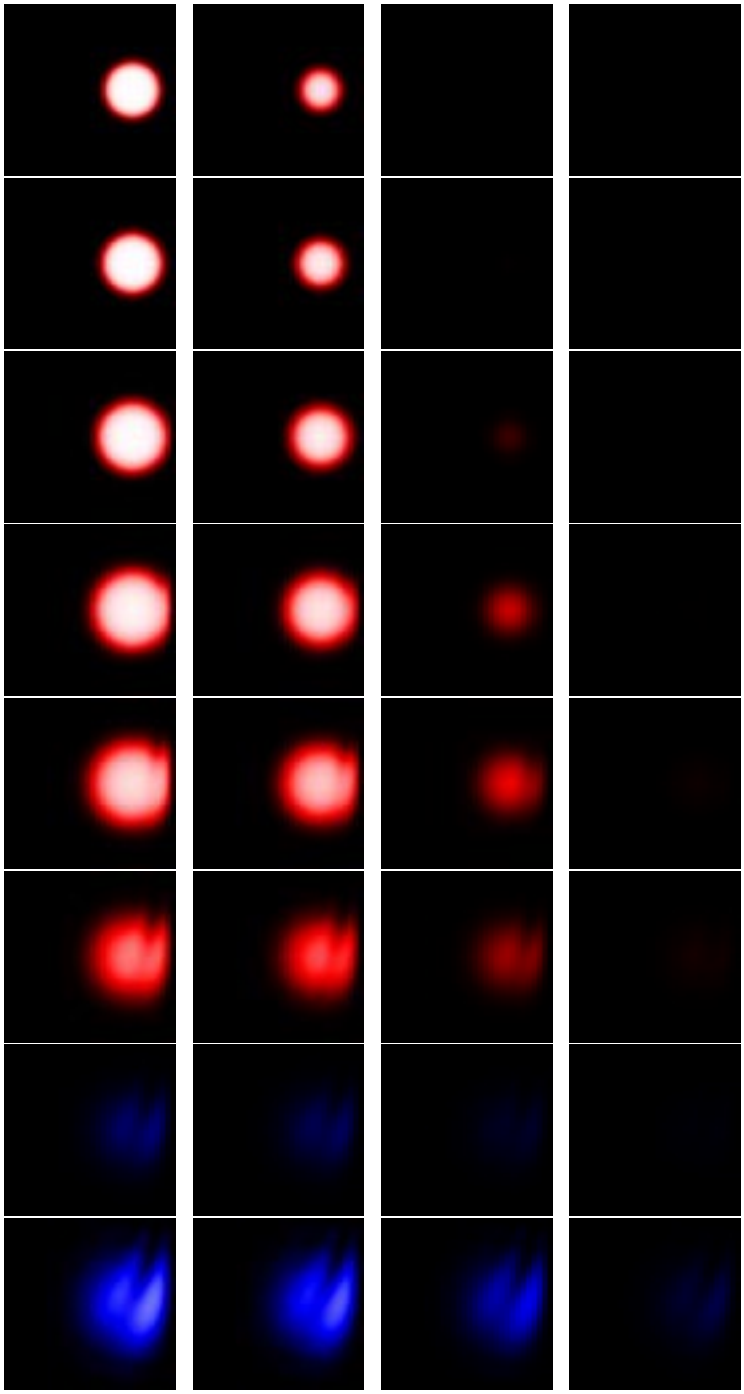

Abbildung 5.14: Spinpräzession: $T_{RB} \circ \psi^* \sigma_1 \psi$

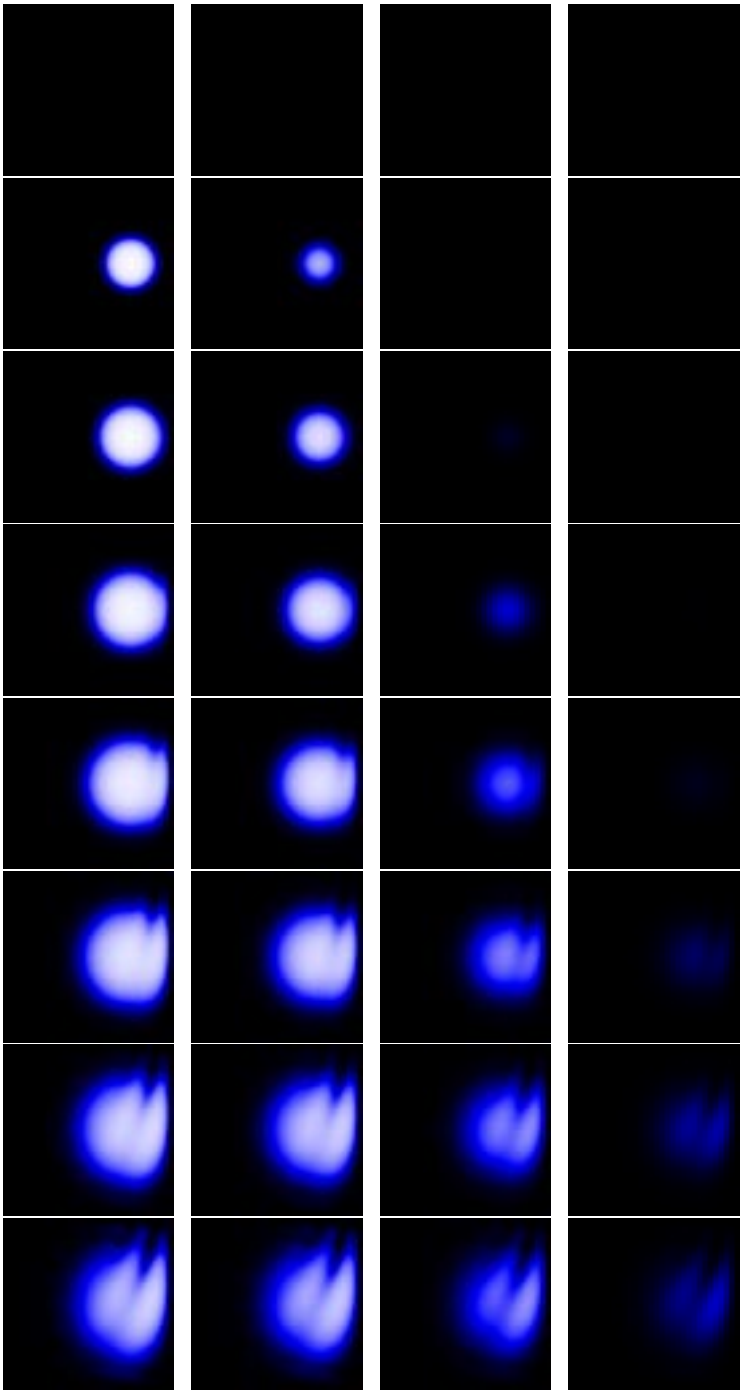

Abbildung 5.15: Spinpräzession: $T_{RB} \circ \psi^* \sigma_2 \psi$

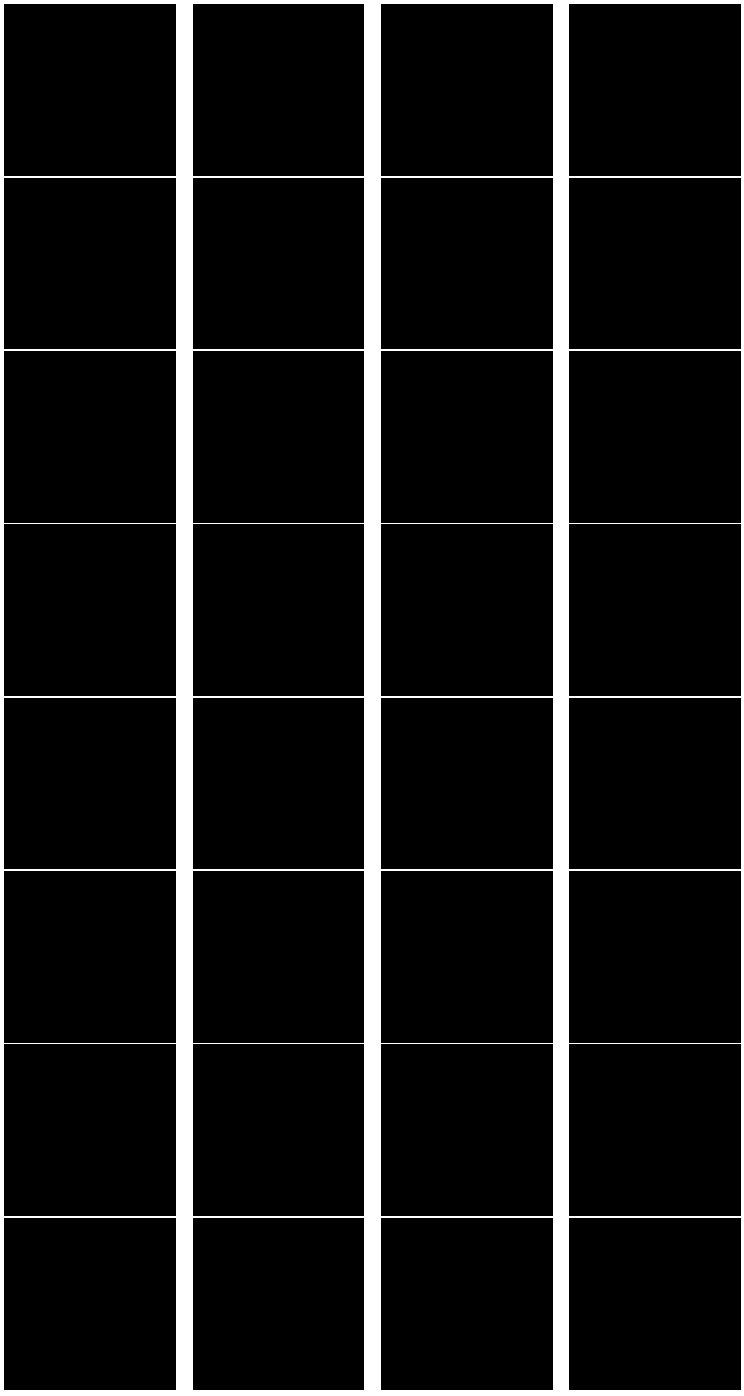

Abbildung 5.16: Spinpräzession: $T_{RB} \circ \psi^* \sigma_3 \psi$

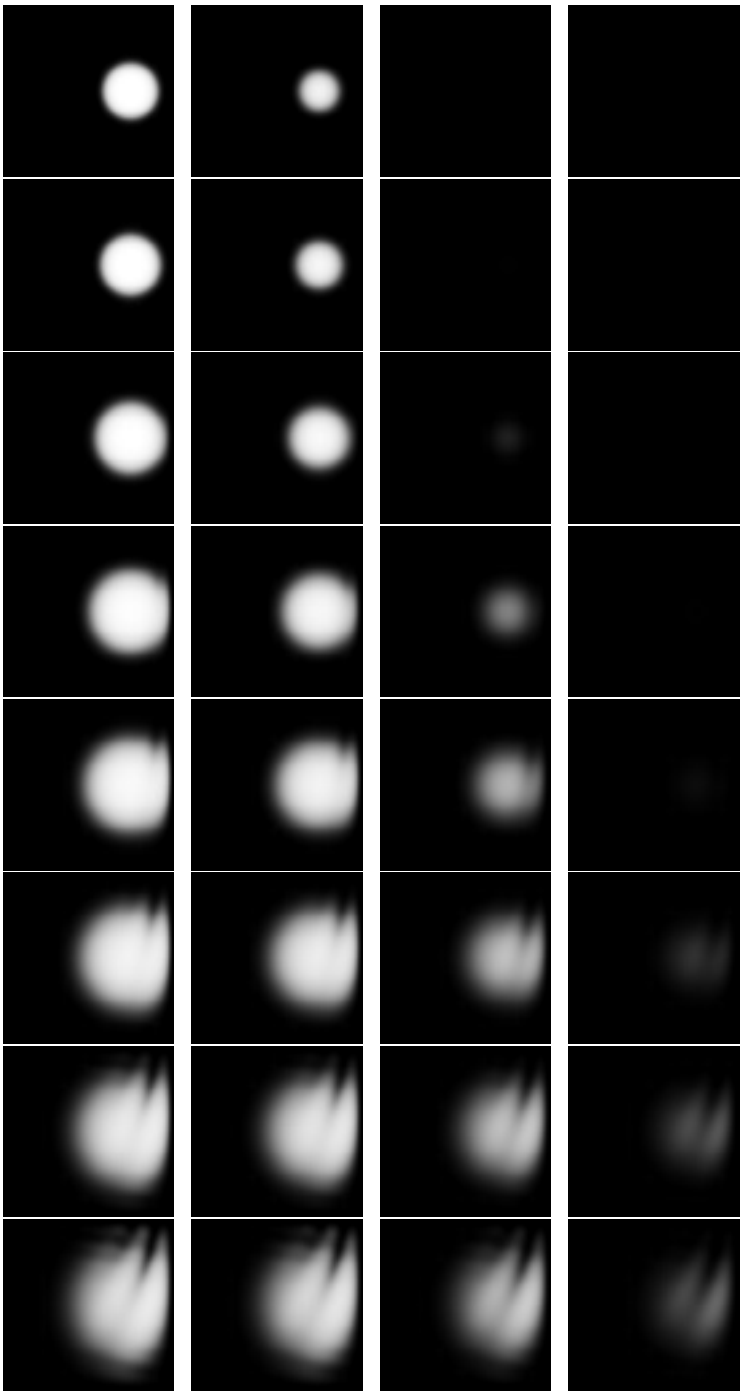

Abbildung 5.17: Spinpräzession: $T_G \circ |\psi|^2$

5.4 Dirac-Gleichung

Die Simulationen zur Dirac-Gleichung umfassen einen Bindungszustand im konstanten Magnetfeld in zwei Raumdimensionen und einen lokalisierten Zustand in einem kovarianten skalaren Potential in drei Raumdimensionen. Im Gegensatz zur Schrödinger- und Pauli-Gleichung bereiten die Dirichlet-Randbedingungen, die im Algorithmus in natürlicher Weise integriert sind, erhebliche Probleme. In vernünftiger Weise lassen sich daher nur lokalisierte Zustände simulieren, die die Ränder des Simulationsgebietes nicht berühren.

5.4.1 Bindungszustand im konstanten Magnetfeld

Die zweidimensionale Simulation zur Dirac-Gleichung beschreibt einen Bindungszustand im konstanten Magnetfeld mit der Feldstärke B. In der üblichen Definition lautet das Vektorpotential:

$$\mathbf{A}(\mathbf{x}) = \frac{B}{2}(-x_2, x_1) \tag{5.16}$$

Entsprechend ist das Vektorpotential im Mathematica-Notebook formuliert.

```
VECTOR POTENTIAL

Vector2D[h_,B_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    I B/2 ((x1-1/2) + I(x2-1/2))]}

h = 1/127; B = 256; n1 = 1/h+1; n2 = 1/h+1;
Ai = Array[Vector2D[h, B], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai], 0 Ai];
```

Die dritte Komponente in der Definition des Vektorpotentials beschreibt ein kovariantes skalares Potential. Ein Beispiel für ein derartiges Potential wird im nächsten Abschnitt anhand der dreidimensionalen Dirac-Gleichung vorgestellt.

Die Wellenfunktion für die Simulation ist ein reelles Spinorfeld. Die beiden komplexen Komponenten des Spinors sind als zentrierte, reelle Gaußverteilungen definiert.

```
WAVEFUNCTION

Gauss2D[h_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-1/2)^2 + (x2-1/2)^2)/2]]}

a = 10; b = 200;
Psii = Array[Gauss2D[h, a, b], {n2, n1}, 0];
Psie = NewFunction[Psii, 0 Psii, Psii, 0 Psii];

Psiw0 = ShowWindow[Psie, 0];
Psiw1 = ShowWindow[Psie, 1];
Psiw2 = ShowWindow[Psie, 2];
Psiw3 = ShowWindow[Psie, 3];
Psiw4 = ShowWindow[Psie, 4];
Psiw5 = ShowWindow[Psie, 5];

HideWindow[Psiw0];
HideWindow[Psiw1];
HideWindow[Psiw2];
HideWindow[Psiw3];
HideWindow[Psiw4];
HideWindow[Psiw5];
```

Trotz der einfachen Struktur des Anfangszustandes ist die Zeitevolution der Spinorfunktion nicht trivial. Der Spindichtevektor $\mathbf{s} = (\psi^*\sigma_1\psi, \psi^*\sigma_2\psi, \psi^*\sigma_3\psi)$ aus der Pauli-Theorie besitzt in der Dirac-Theorie die Interpretation einer Geschwindigkeitsdichte, die der Zitterbewegung unterliegt [13]. Die zeitabhängige vektorielle Dichte $\mathbf{s}(t)$ korrespondiert mit dem relativistischen Geschwindigkeitsvektor $(v_1(t), v_2(t), \sqrt{1 - \mathbf{v}(t)^2})$ aus der klassischen Mechanik.

```
HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.; units = 1.h;
He = Dirac2D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 1.h; fractal = 4; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];
BeginMovie[Psiw2];
BeginMovie[Psiw3];
BeginMovie[Psiw4];
BeginMovie[Psiw5];

EndMovie[Psiw0];
EndMovie[Psiw1];
EndMovie[Psiw2];
EndMovie[Psiw3];
EndMovie[Psiw4];
EndMovie[Psiw5];
```

Die Struktur des Dirac-Operators unterscheidet sich durch die Linearität in den Impulsoperatoren wesentlich von der des Schrödinger- oder Pauli-Operators. Diese Linearität führt auf eine lineare Abhängigkeit der Stabilitätsgrenze des Algorithmus von der Maschenweite h des Simulationsgitters. In der nichtrelativistischen Theorie war diese Abhängigkeit quadratisch. Aufgrund dieser linearen Abhängigkeit ist für die Dirac-Theorie ein fraktaler Algorithmus vierter Ordnung ausreichend, da die Zeitschritte entsprechend größer gewählt werden können.

Die Wellenfunktion, die Wahrscheinlichkeitsdichte und die Geschwindigkeitsdichte nach 32 Zeitschritten, also zur Zeit $T \approx 0.25197$ sind in Abbildung 5.18 dargestellt. Die Visualisierung der Zeitevolution umfaßt die beiden komplexen Komponenten des Spinorfeldes und die Wahrscheinlichkeitsdichte in Abbildung 5.19 und die Komponenten der Geschwindigkeitsdichte in Abbildung 5.20. Die Zeitachse in den Darstellungen verläuft von oben nach unten. Ein Simulationslauf zur Berechnung der Animationen benötigt circa 30 Sekunden. Das bearbeitete Gleichungssystem umfaßt 65536 reelle Variable.

5.4.2 Lokalisierter Zustand im kovarianten Potential

Das letzte Simulationsbeispiel befaßt sich mit der dreidimensionalen Dirac-Gleichung in einem kovarianten skalaren Potential von parabolischer Gestalt.

$$A_4(\mathbf{x}) = c(x_1^2 + x_2^2 + x_3^2) \tag{5.17}$$

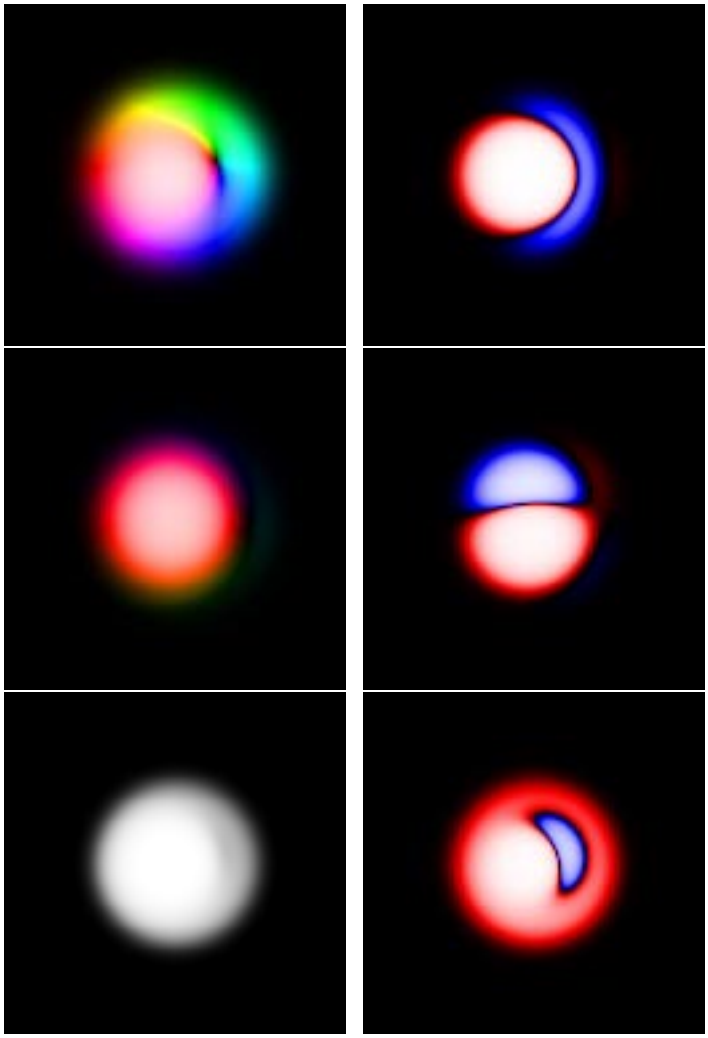

Abbildung 5.18: Bindungszustand im konstanten Magnetfeld

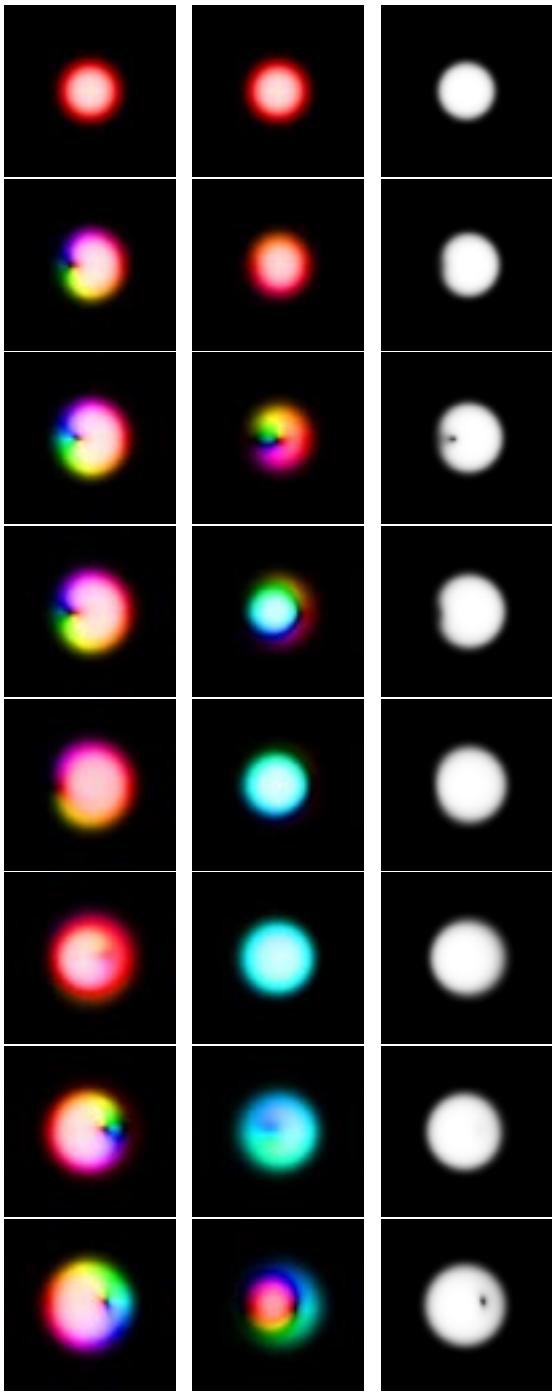

Abbildung 5.19: Bindungszustand: $T_{RGB} \circ \psi_1, T_{RGB} \circ \psi_2, T_G \circ |\psi|^2$

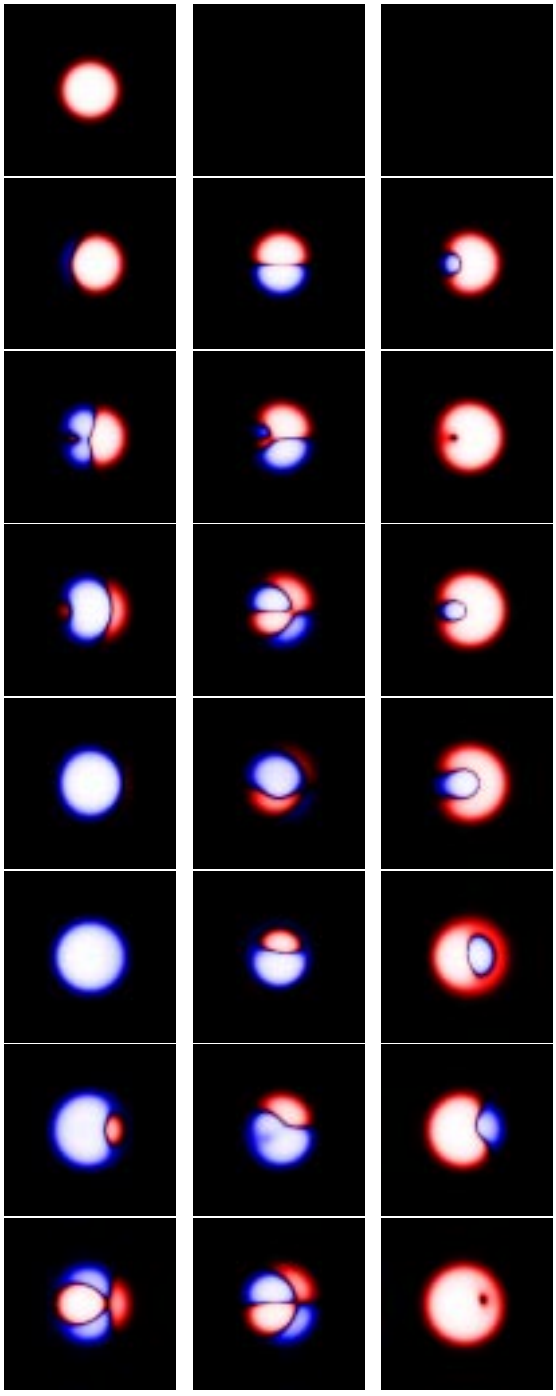

Abbildung 5.20: Bindungszustand: $T_{RB} \circ \psi^* \sigma_1 \psi, T_{RB} \circ \psi^* \sigma_2 \psi, T_{RB} \circ \psi^* \sigma_3 \psi$

Das Potential wird als vierte Komponente des Vektorpotentials in den Algorithmus integriert.

```
VECTOR POTENTIAL

Vector3D[h_,c_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    c ((x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2)]}

h = 1/63; c = 200; n1 = 1/h+1; n2 = 1/h+1; n3 = 1/h+1;
Ai = Array[Vector3D[h, c], {n3, n2, n1}, 0];
Ae = NewFunction[0 Ai, 0 Ai, 0 Ai, Ai];
```

Die Wellenfunktion zur Simulation wird als reelles Bispinorfeld mit zentrierten Gaußverteilungen in jeder Komponente gewählt.

```
WAVEFUNCTION

Gauss3D[h_,a_,b_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    a*Exp[-b((x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2)/2]]}

a = 10; b = 200;
Psii = N[Array[Gauss3D[h, a, b], {n3, n2, n1}, 0]];
Psie = NewFunction[Psii, 0 Psii, Psii, 0 Psii,
                   Psii, 0 Psii, Psii, 0 Psii];

slices = {31, 39, 47, 55};
Psiw0 = ShowWindow[Psie, 0, #]& /@ slices;
Psiw1 = ShowWindow[Psie, 1, #]& /@ slices;
Psiw2 = ShowWindow[Psie, 2, #]& /@ slices;
Psiw3 = ShowWindow[Psie, 3, #]& /@ slices;
Psiw4 = ShowWindow[Psie, 4, #]& /@ slices;
Psiw5 = ShowWindow[Psie, 5, #]& /@ slices;
Psiw6 = ShowWindow[Psie, 6, #]& /@ slices;
Psiw7 = ShowWindow[Psie, 7, #]& /@ slices;
Psiw8 = ShowWindow[Psie, 8, #]& /@ slices;

HideWindow[#]& /@ Psiw0;
HideWindow[#]& /@ Psiw1;
HideWindow[#]& /@ Psiw2;
HideWindow[#]& /@ Psiw3;
HideWindow[#]& /@ Psiw4;
HideWindow[#]& /@ Psiw5;
HideWindow[#]& /@ Psiw6;
HideWindow[#]& /@ Psiw7;
HideWindow[#]& /@ Psiw8;
```

Die Visualisierung umfaßt die bekannten vier Schnitte durch den Simulationswürfel. Gezeichnet werden die vier Komponenten der Wellenfunktion, die Geschwindigkeitsdichte und die Wahrscheinlichkeitsdichte.

```
HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.; units = 1.h;
He = Dirac3D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 1.h 2/3; fractal = 4; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION
```

```
BeginMovie[#]& /@ Psiw0;
BeginMovie[#]& /@ Psiw1;
BeginMovie[#]& /@ Psiw2;
BeginMovie[#]& /@ Psiw3;
BeginMovie[#]& /@ Psiw4;
BeginMovie[#]& /@ Psiw5;
BeginMovie[#]& /@ Psiw6;
BeginMovie[#]& /@ Psiw7;
BeginMovie[#]& /@ Psiw8;

EndMovie[#]& /@ Psiw0;
EndMovie[#]& /@ Psiw1;
EndMovie[#]& /@ Psiw2;
EndMovie[#]& /@ Psiw3;
EndMovie[#]& /@ Psiw4;
EndMovie[#]& /@ Psiw5;
EndMovie[#]& /@ Psiw6;
EndMovie[#]& /@ Psiw7;
EndMovie[#]& /@ Psiw8;
```

⌙

Der Hamilton-Operator und die Zeitevolution sind in der üblichen Art und Weise definiert. Die für die Stabilität des Algorithmus zulässige Größe eines Zeitschrittes hängt linear von der Maschenweite h ab und ist mit dem Faktor $2/3$ für dreidimensionale Systeme skaliert. Der Vorfaktor 1 ist die empirische Stabilitätsgrenze für den fraktalen Algorithmus vierter Ordnung. Für einen Algorithmus sechster Ordnung ist dieser Vorfaktor 4.

Insgesamt umfaßt die Simulation mehr als zwei Millionen Gleichungen, was sich in einer Rechenzeit von etwa einer Stunde niederschlägt. Die im Vergleich zur Pauli-Gleichung geringe Rechenzeit hängt mit der einfachen Struktur der Dirac-Gleichung und der kleineren Ordnung der fraktalen Approximation zusammen.

Die Visualisierung umfaßt neben den Komponenten des Bispinorfeldes und der Wahrscheinlichkeitsdichte die Geschwindigkeitsdichte. Die Observable der klassischen Geschwindigkeit genügt der Heisenbergschen Bewegungsgleichung

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = \mathrm{i}[H, \mathbf{x}(t)] = \boldsymbol{\alpha}(t). \tag{5.18}$$

Die Operatoren $\mathbf{x}(t)$ und $\boldsymbol{\alpha}(t)$ sind als Heisenberg-Operatoren zu verstehen. Die Beziehung zwischen einem Operator im Schrödinger-Bild und einem im Heisenberg-Bild ist durch die Transformation $\boldsymbol{\alpha}(t) = \mathrm{e}^{\mathrm{i}Ht}\boldsymbol{\alpha}\mathrm{e}^{-\mathrm{i}Ht}$ gegeben. Die Geschwindigkeitsdichte des Spinorfeldes lautet damit:

$$\mathbf{v}(t) = \psi^*(0)\boldsymbol{\alpha}(t)\psi(0) = \psi^*(t)\boldsymbol{\alpha}\psi(t) \tag{5.19}$$

In einer Analogiebetrachtung zur klassischen relativistischen Mechanik kann der Heisenberg-Operator $\beta(t)$ mit dem relativistischen Ausdruck $\sqrt{1-\mathbf{v}(t)^2}$ in Zusammenhang gebracht werden [13]. Damit besteht eine Korrespondenz zwischen dem relativistischen Geschwindigkeitsvektor $(\mathbf{v}(t), \sqrt{1-\mathbf{v}(t)^2})$ und dem Vektor $(\boldsymbol{\alpha}(t), \beta(t))$ von Heisenberg-Operatoren. Die Geschwindigkeitsoperatoren unterliegen wie im zweidimensionalen Fall der Zitterbewegung.

Die Bilderserien zur Simulation umfassen in horizontaler Richtung die vier Schnitte durch die Simulationsbox und in vertikaler Richtung die Zeitentwicklung. In der angegebenen Reihenfolge sind die vier Komponenten der komplexen Wellenfunktion, die vier Komponenten der Geschwindigkeitsdichte und die Wahrscheinlichkeitsdichte in den Abbildungen 5.21, 5.22, 5.23, 5.24, 5.25, 5.26, 5.27, 5.28 und 5.29 dargestellt.

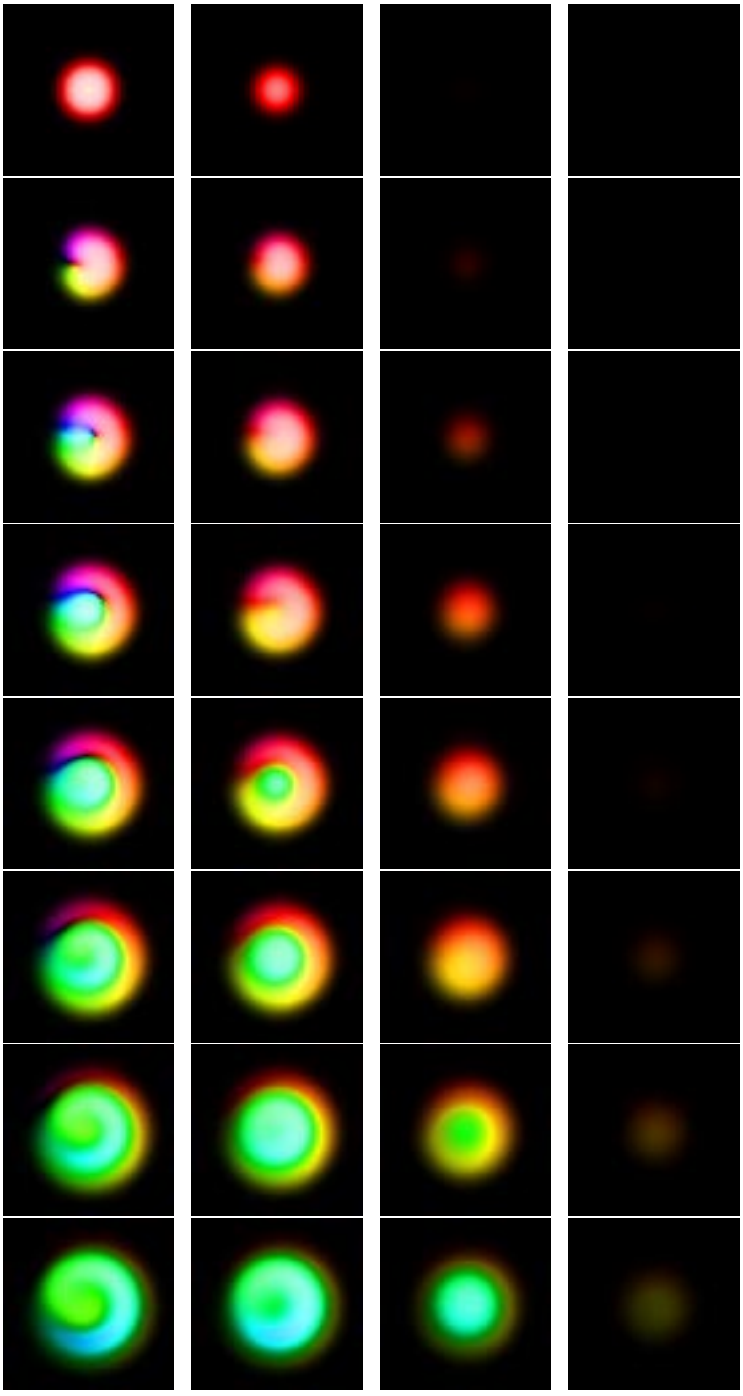

Abbildung 5.21: Lokalisierter Zustand: $T_{RGB} \circ \psi_1$

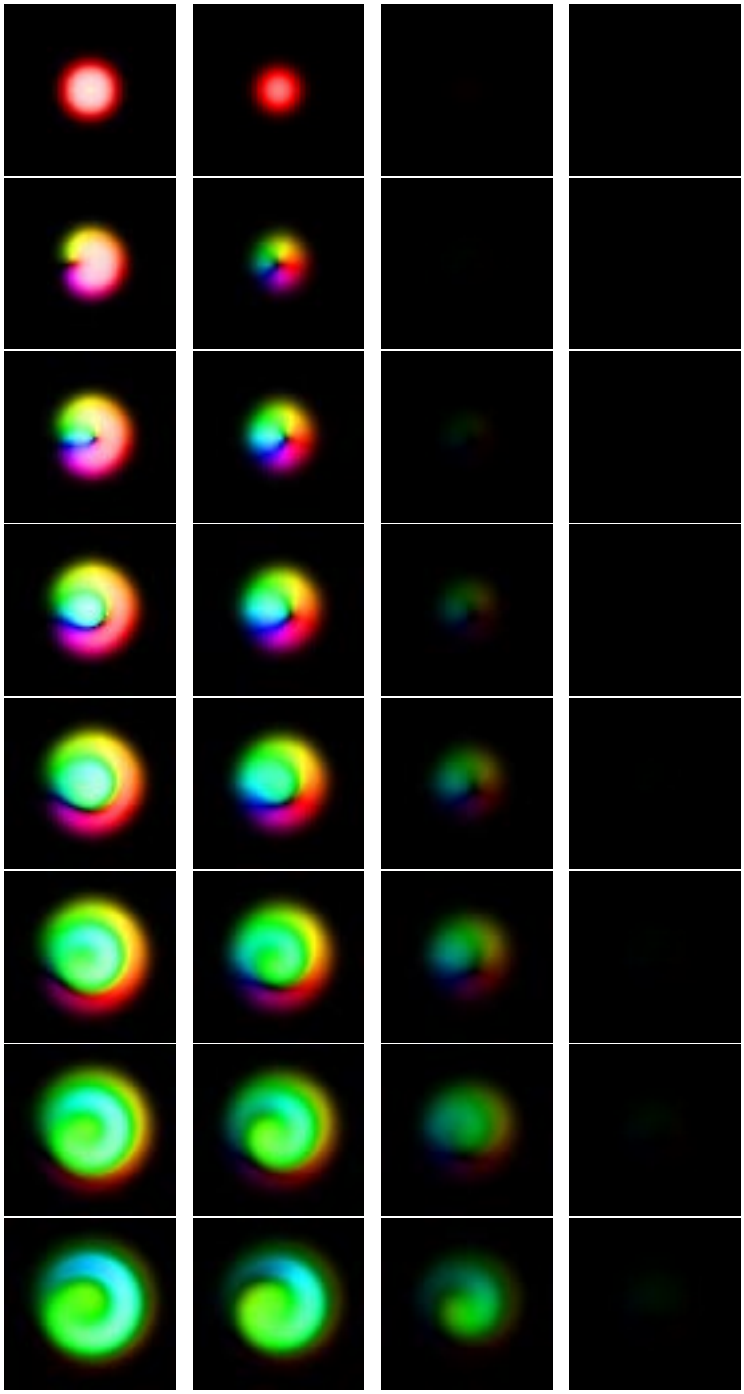

Abbildung 5.22: Lokalisierter Zustand: $T_{RGB} \circ \psi_2$

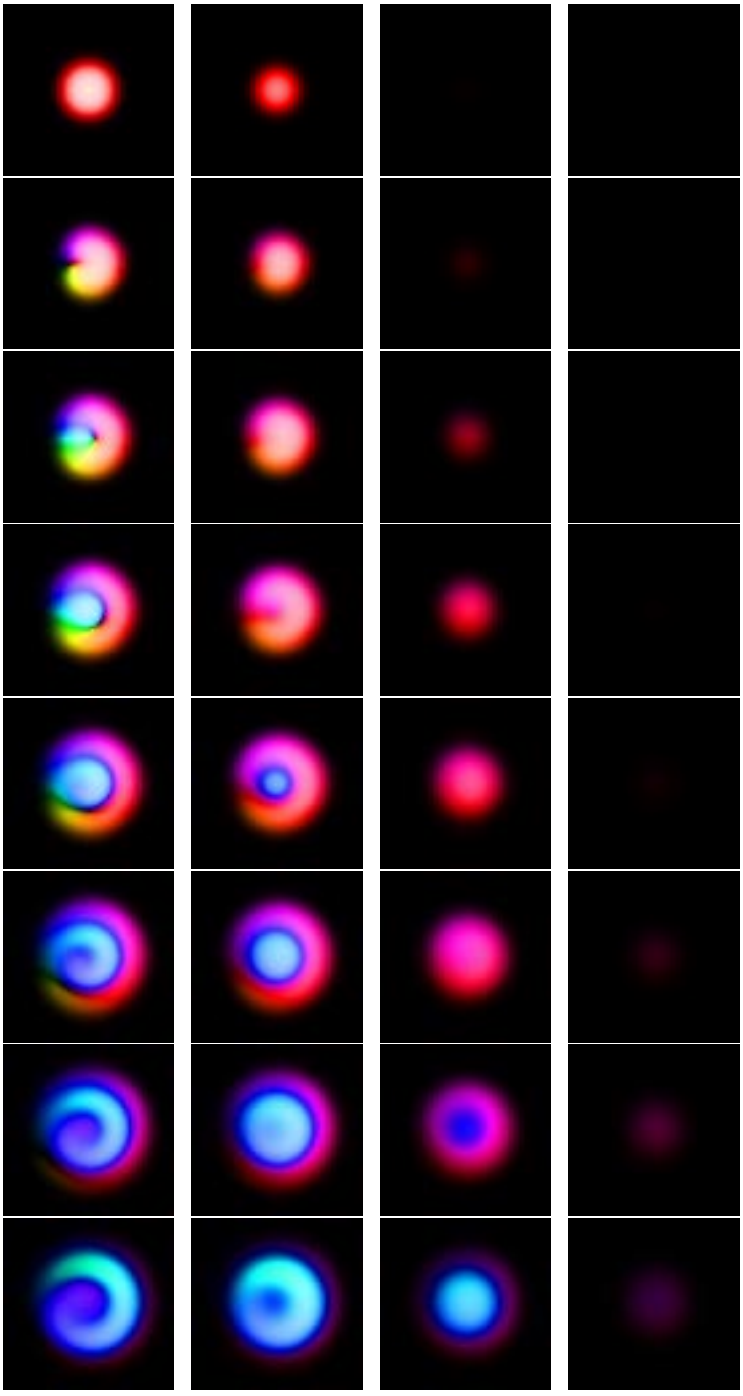

Abbildung 5.23: Lokalisierter Zustand: $T_{RGB} \circ \psi_3$

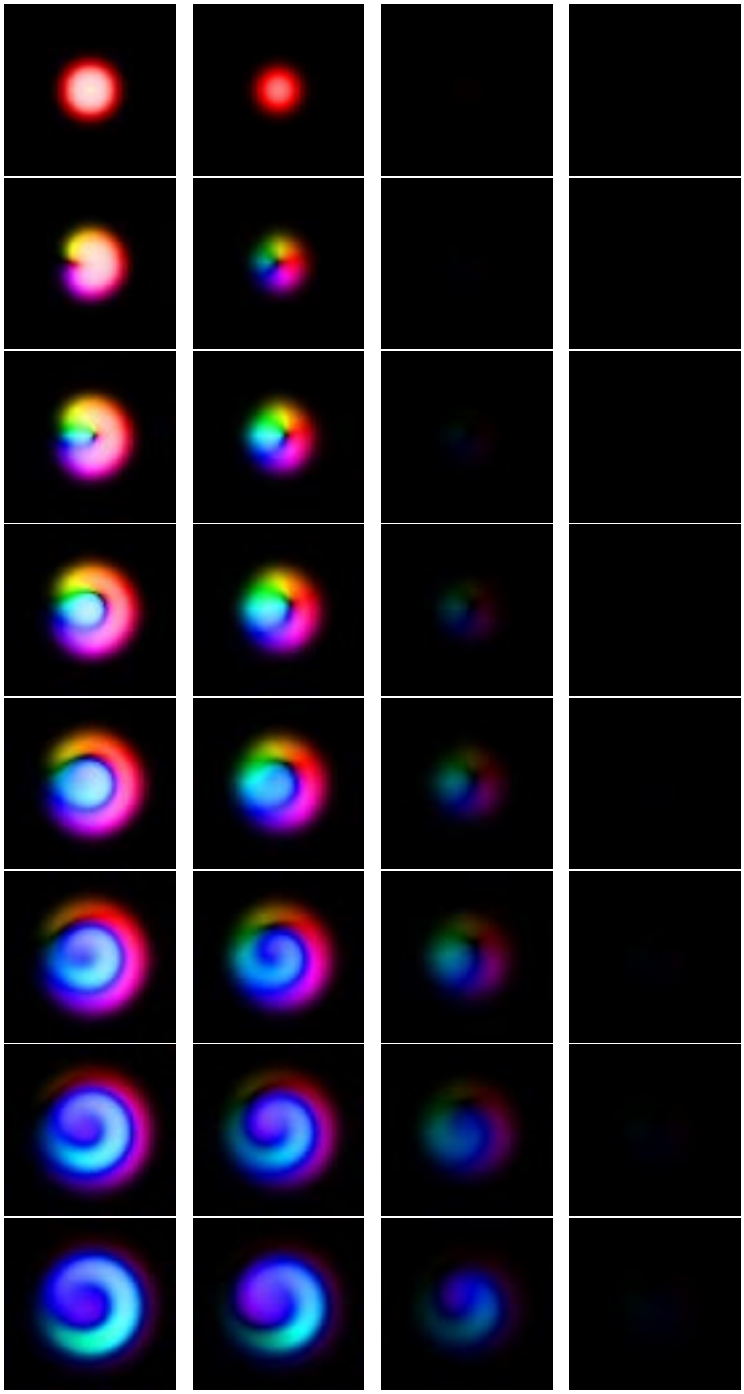

Abbildung 5.24: Lokalisierter Zustand: $T_{RGB} \circ \psi_4$

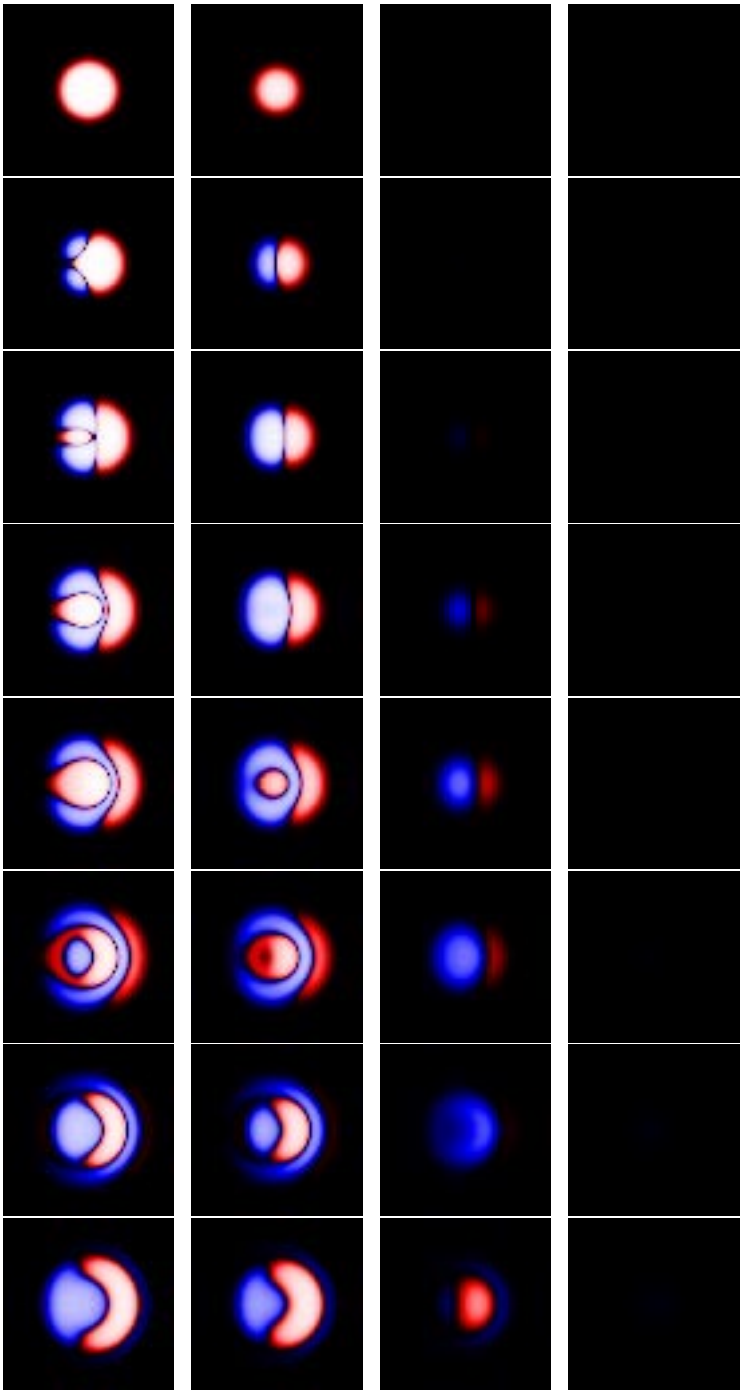

Abbildung 5.25: Lokalisierter Zustand: $T_{RB} \circ \psi^* \alpha_1 \psi$

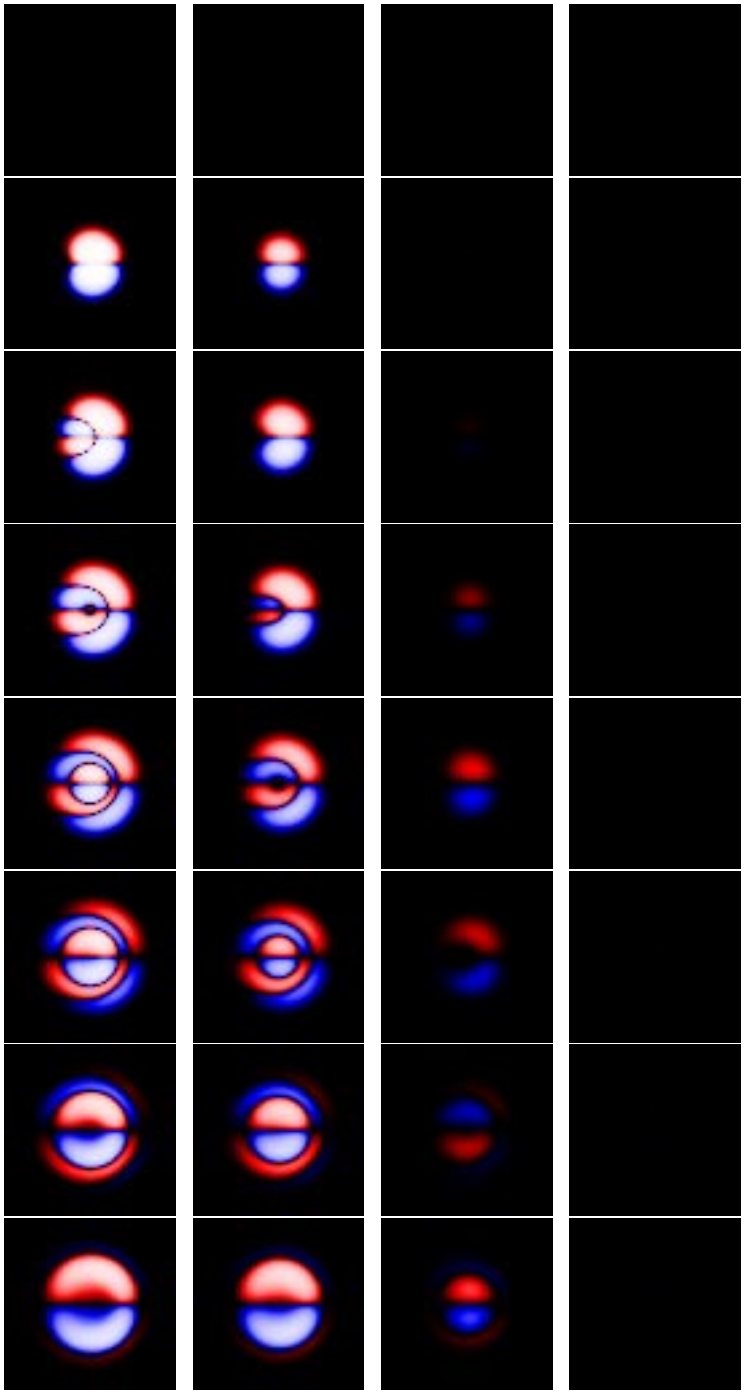

Abbildung 5.26: Lokalisierter Zustand: $T_{RB} \circ \psi^* \alpha_2 \psi$

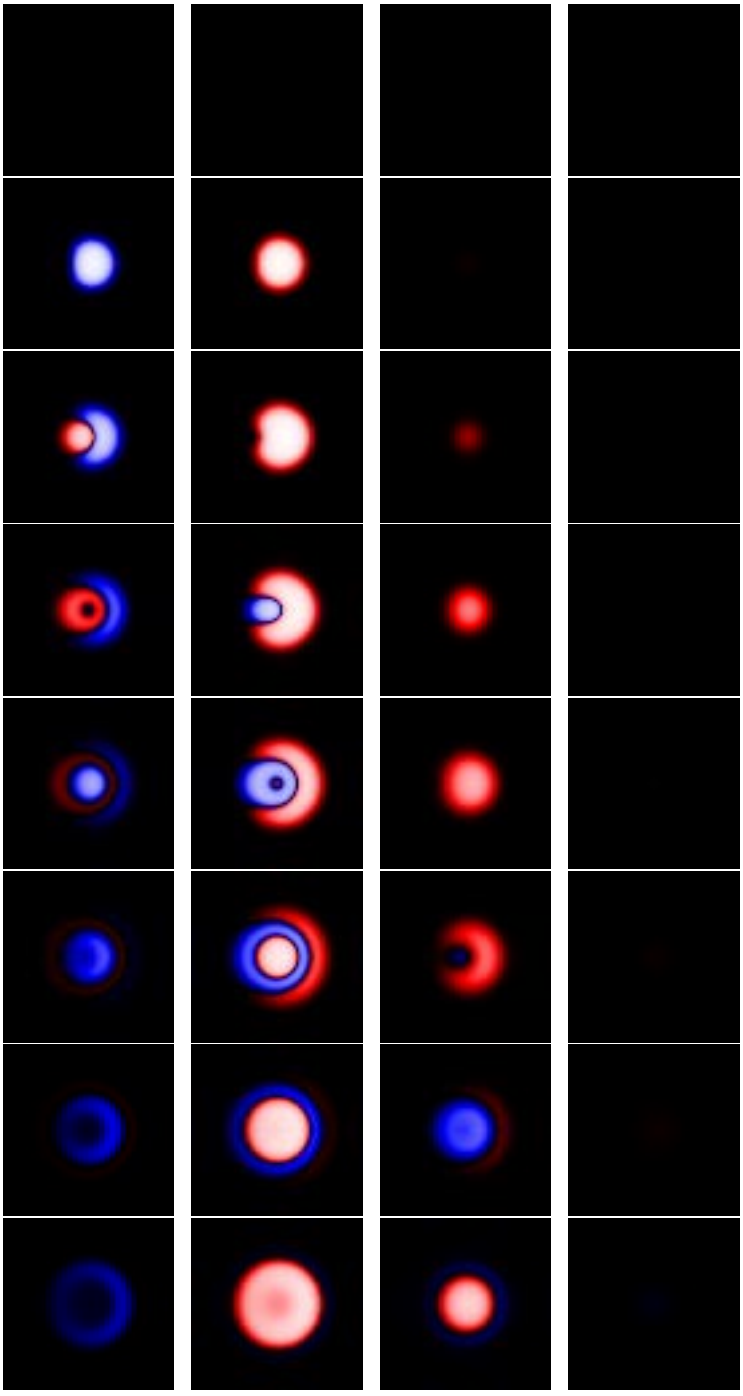

Abbildung 5.27: Lokalisierter Zustand: $T_{RB} \circ \psi^* \alpha_3 \psi$

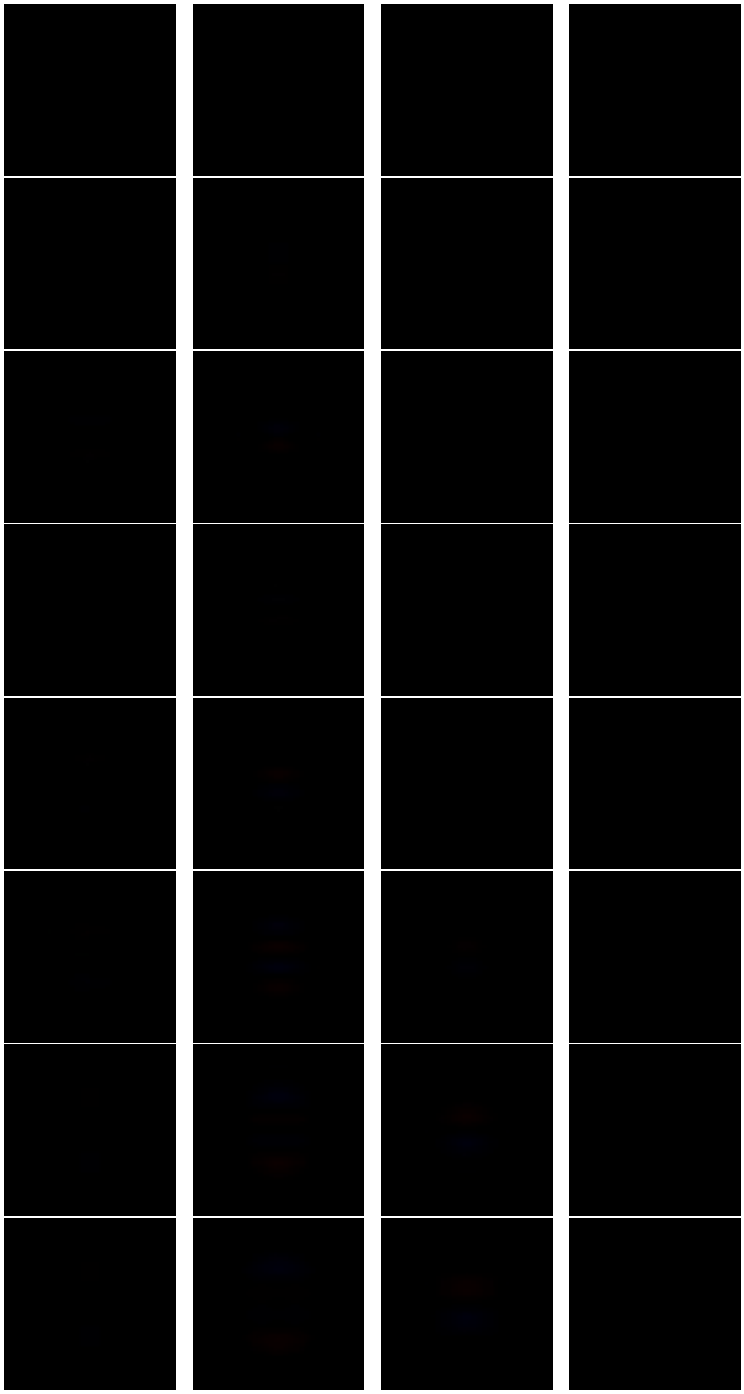

Abbildung 5.28: Lokalisierter Zustand: $T_{RB} \circ \psi^* \beta \psi$

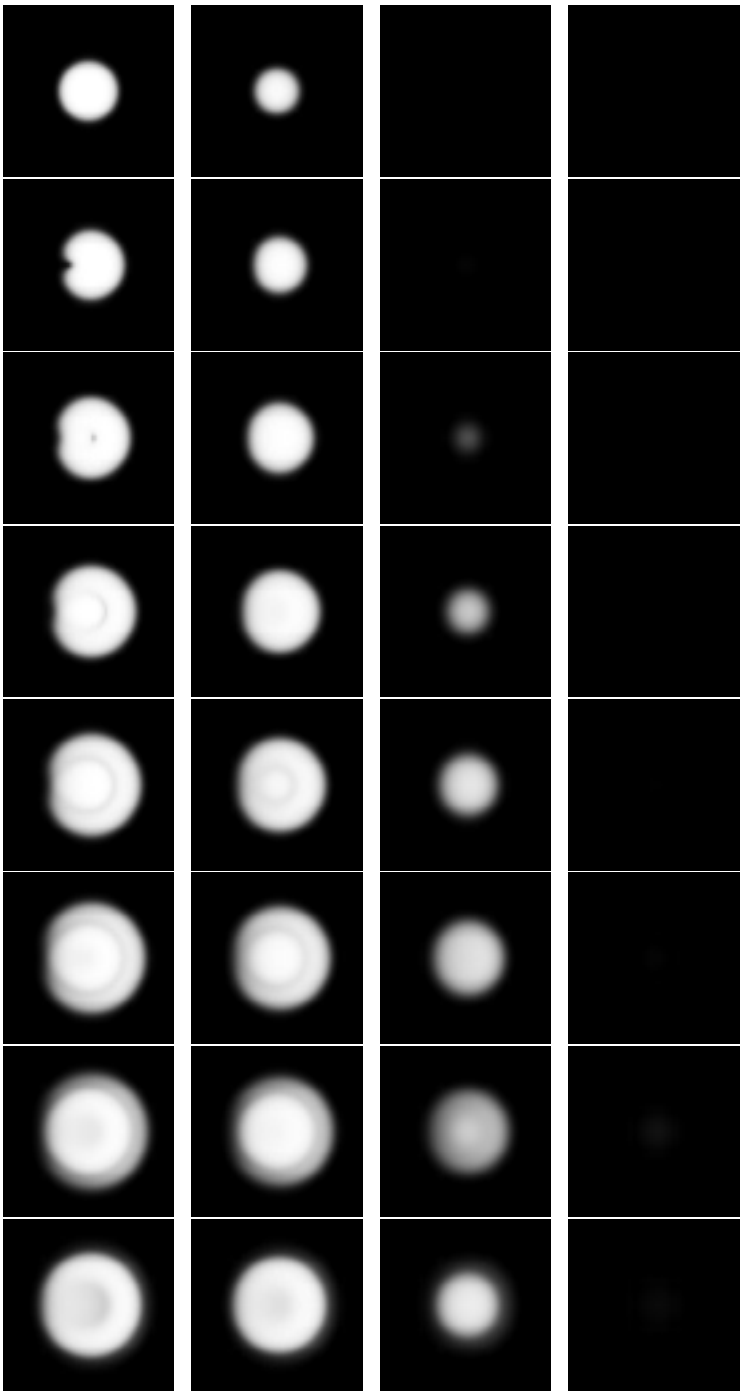

Abbildung 5.29: Lokalisierter Zustand: $T_G \circ |\psi|^2$

Anhang A

C++-Source-Code

A.1 TFunction Class

A.1.1 TFunction.h

```
// ============================================================================
//  TFunction.h         (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TFunction
#define _H_TFunction
#pragma once

#include "TypeDefinition.h"
#include "TList.h"


class   TFunction
{
    public:
                TFunction( void );
                ~TFunction( void );
        Int32   GetArray( void );
        Int32   PutInfo( void );
        Int32   PutArray( void );
        Int32   PutColor( void );
        Int32   PutGray( void );
        Int32   PutRedBlue( void );
        Int32   PutBlackWhite( void );
        Int32   PutAbs( void );
        bool    IsFunction2D(   Int32 &ioNi,
                                Int32 &ioNj );
        bool    IsFunction3D(   Int32 &ioNi,
                                Int32 &ioNj,
                                Int32 &ioNk );
        bool    IsFunctionR( Float* &outSP );
        bool    IsFunctionR2(   Float* &outS1P,
                                Float* &outS2P );
        bool    IsFunctionR3(   Float* &outS1P,
                                Float* &outS2P,
                                Float* &outS3P );
        bool    IsFunctionR4(   Float* &outS1P,
                                Float* &outS2P,
                                Float* &outS3P,
                                Float* &outS4P );
        bool    IsFunctionC(    Float* &outS1P,
                                Float* &outS2P );
        bool    IsFunctionC2(   Float* &outS1P,
                                Float* &outS2P,
                                Float* &outS3P,
```

```
                                Float* &outS4P );
        bool    IsFunctionC4(   Float* &outS1P,
                                Float* &outS2P,
                                Float* &outS3P,
                                Float* &outS4P,
                                Float* &outS5P,
                                Float* &outS6P,
                                Float* &outS7P,
                                Float* &outS8P );
        Int32   UpdateWindow( void );
        Int32   Copy( TFunction* inFunction );
        void    SwapArrayPointers( TFunction* inFunction );

        Int32   mID;

    private:
        Float*  mArrayP;
        Int32*  mCountP;
        Int8**  mHeadH;
        Int32   mDepth;
};

extern TList<TFunction> *gFunctionList;

#endif
```

A.1.2 TFunction.cp

```
// ============================================================================
//  TFunction.cp        (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Class for functions

#include "MathLinkUtilities.h"
#include "TWindow.h"
#include "TFunction.h"
#include "TOperator.h"
#include <string.h>
#include <stdio.h>
#include <math.h>



// ----------------------------------------------------------------------------
//       TFunction
// ----------------------------------------------------------------------------
//  Constructor

TFunction::TFunction( void )
{
    mArrayP = nil;
    mCountP = nil;
    mHeadH = nil;
    mDepth = 0;
}



// ----------------------------------------------------------------------------
//       ~TFunction
// ----------------------------------------------------------------------------
//  Destructor

TFunction::~TFunction( void )
{
    delete [] mArrayP;
    delete [] mCountP;
```

```
}


// ------------------------------------------------------------------------
//       GetArray
// ------------------------------------------------------------------------
//  Read float array from MathLink

Int32   TFunction::GetArray( void )
{
    return MLGetRealArray2(stdlink, mArrayP, mCountP, mDepth);
}



// ------------------------------------------------------------------------
//       PutInfo
// ------------------------------------------------------------------------
//  Return list of array dimensions

Int32   TFunction::PutInfo( void )
{
    Int32   i;

    MLPutFunction(stdlink, "List", mDepth);
    for( i = 0; i < mDepth; i++ )
        MLPutLongInteger(stdlink, mCountP[mDepth-1-i]);

    return eOK;
}



// ------------------------------------------------------------------------
//       PutArray
// ------------------------------------------------------------------------
//  Return float array to MathLink

Int32   TFunction::PutArray( void )
{
    return MLPutDoubleArray(stdlink, mArrayP, mCountP, mHeadH, mDepth);
}



// ------------------------------------------------------------------------
//       PutColor
// ------------------------------------------------------------------------
//  Return RGB color array

Int32   TFunction::PutColor( void )
{
    Int32   i, size;
    Float   *realP, *imagP, *colorP;
    Int8    *headH[3] = {"List", "List", "RGBColor"};
    Int32   countP[3] = {0, 0, 3};
    Float   re, im, r, s, s0, phi, phi0, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k3pi = 3.0 / pi, k4pi = 4.0 / pi;


    if( !IsFunction2D(countP[1], countP[0]) )
        return MLErrorReport(stdlink, "two-dimensional function expected");
    if( !IsFunctionC(realP, imagP) )
        return MLErrorReport(stdlink, "complex function expected");

    size = countP[0] * countP[1];
    colorP = new Float[3 * size];
    if( !colorP )
        return MLErrorReport(stdlink, "out of memory");
    if( eError == MLCheckMemoryReserve(stdlink) ) return eError;

    for( i = 0; i < size; i++ ) {
```

```
        re = *realP++;
        im = *imagP++;

        r = hypot( re, im );
        s = k4pi * atan( r );
        phi = k3pi * atan2( im, re );
        phi0 = fabs( phi );

        if( phi0 < one ) { red = one; green = phi0; blue = zero; }
        else {
            if( phi0 < two ) { red = two - phi0; green = one; blue = zero; }
            else { red = zero; green = one; blue = phi0 - two; }
        }

        if( phi < zero ) { temp = green; green = blue; blue = temp; }

        if( s < one ) { red *= s; green *= s; blue *= s; }
        else {
            s = two - s; s0 = one - s;
            red = red * s + s0; green = green * s + s0; blue = blue * s + s0;
        }

        if( !isfinite(red) ) red = zero;
        if( !isfinite(green) ) green = zero;
        if( !isfinite(blue) ) blue = zero;
        *colorP++ = red;
        *colorP++ = green;
        *colorP++ = blue;
    }
    colorP -= 3*size;


    MLPutDoubleArray(stdlink, colorP, countP, headH, 3);
    delete [] colorP;

    return eOK;
}



// ------------------------------------------------------------------------
//       PutGray
// ------------------------------------------------------------------------
//  Return gray array

Int32   TFunction::PutGray( void )
{
    Int32   i, size;
    Float   *realP, *imagP, *grayP;
    Int8    *headH[3] = {"List", "List", "GrayLevel"};
    Int32   countP[3] = {0, 0, 1};
    Float   re, im, r, gray;
    Float   zero = 0.0, one = 1.0;
    Float   k2pi = 2.0 / pi;


    if( !IsFunction2D(countP[1], countP[0]) )
        return MLErrorReport(stdlink, "two-dimensional function expected");
    if( !IsFunctionC(realP, imagP) )
        return MLErrorReport(stdlink, "complex function expected");

    size = countP[0] * countP[1];
    grayP = new Float[size];
    if( !grayP )
        return MLErrorReport(stdlink, "out of memory");
    if( eError == MLCheckMemoryReserve(stdlink) ) return eError;

    for( i = 0; i < size; i++ ) {
        re = *realP++;
        im = *imagP++;
        r = re * re + im * im;
        gray = k2pi * atan( r );

        if( !isfinite(gray) ) gray = zero;
```

```
        *grayP++ = gray;
    }
    grayP -= size;

    MLPutDoubleArray(stdlink, grayP, countP, headH, 3);
    delete [] grayP;

    return eOK;
}



// ------------------------------------------------------------------------
//       PutRedBlue
// ------------------------------------------------------------------------
//  Return red-blue color array

Int32   TFunction::PutRedBlue( void )
{
    Int32   i, size;
    Float   *realP, *colorP;
    Int8    *headH[3] = {"List", "List", "RGBColor"};
    Int32   countP[3] = {0, 0, 3};
    Float   r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0;
    Float   k4pi = 4.0 / pi;


    if( !IsFunction2D(countP[1], countP[0]) )
        return MLErrorReport(stdlink, "two-dimensional function expected");
    if( !IsFunctionR(realP) )
        return MLErrorReport(stdlink, "real function expected");

    size = countP[0] * countP[1];
    colorP = new Float[3 * size];
    if( !colorP )
        return MLErrorReport(stdlink, "out of memory");
    if( eError == MLCheckMemoryReserve(stdlink) ) return eError;

    for( i = 0; i < size; i++ ) {
        r = *realP++;
        s = k4pi * atan( fabs( r ) );

        if( s < one ) { red = s; green = blue = zero; }
        else { red = one; green = blue = s - one; }

        if( r < zero ) { temp = red; red = blue; blue = temp; }

        if( !isfinite(red) ) red = zero;
        if( !isfinite(green) ) green = zero;
        if( !isfinite(blue) ) blue = zero;
        *colorP++ = red;
        *colorP++ = green;
        *colorP++ = blue;
    }
    colorP -= 3*size;

    MLPutDoubleArray(stdlink, colorP, countP, headH, 3);
    delete [] colorP;

    return eOK;
}



// ------------------------------------------------------------------------
//       PutBlackWhite
// ------------------------------------------------------------------------
//  Return black-white array

Int32   TFunction::PutBlackWhite( void )
{
    Int32   i, size;
    Float   *realP, *grayP;
```

```
    Int8    *headH[3] = {"List", "List", "GrayLevel"};
    Int32   countP[3] = {0, 0, 1};
    Float   gray;
    Float   zero = 0.0, one = 1.0;


    if( !IsFunction2D(countP[1], countP[0]) )
        return MLErrorReport(stdlink, "two-dimensional function expected");
    if( !IsFunctionR(realP) )
        return MLErrorReport(stdlink, "real function expected");

    size = countP[0] * countP[1];
    grayP = new Float[size];
    if( !grayP )
        return MLErrorReport(stdlink, "out of memory");
    if( eError == MLCheckMemoryReserve(stdlink) ) return eError;

    for( i = 0; i < size; i++ ) {
        if( *realP++ < zero ) gray = one;
        else gray = zero;

        *grayP++ = gray;
    }
    grayP -= size;

    MLPutDoubleArray(stdlink, grayP, countP, headH, 3);
    delete [] grayP;

    return eOK;
}



// ------------------------------------------------------------------------
//       PutAbs
// ------------------------------------------------------------------------
//  Return abs array

Int32   TFunction::PutAbs( void )
{
    Int32   i, size, countP[2] = {0, 0};
    Float   *realP, *imagP, *absP, re, im, r;
    Float   zero = 0.0;


    if( !IsFunction2D(countP[1], countP[0]) )
        return MLErrorReport(stdlink, "two-dimensional function expected");
    if( !IsFunctionC(realP, imagP) )
        return MLErrorReport(stdlink, "complex function expected");

    size = countP[0] * countP[1];
    absP = new Float[size];
    if( !absP )
        return MLErrorReport(stdlink, "out of memory");
    if( eError == MLCheckMemoryReserve(stdlink) ) return eError;

    for( i = 0; i < size; i++ ) {
        re = *realP++;
        im = *imagP++;
        r = hypot( re, im );

        if( !isfinite(r) ) r = zero;
        *absP++ = r;
    }
    absP -= size;

    MLPutDoubleArray(stdlink, absP, countP, mHeadH, 2);
    delete [] absP;

    return eOK;
}
```

```
// ---------------------------------------------------------------------------
//       IsFunction2D
// ---------------------------------------------------------------------------
//  Check if function is 2D

bool    TFunction::IsFunction2D(    Int32 &ioNi,
                                    Int32 &ioNj )
{
    if( mDepth != 3 ) return false;

    if( ioNi == 0 && ioNj == 0 ) {
        ioNj = mCountP[1];
        ioNi = mCountP[2];
    }
    else if( ioNj != mCountP[1] || ioNi != mCountP[2] ) return false;

    return true;
}



// ---------------------------------------------------------------------------
//       IsFunction3D
// ---------------------------------------------------------------------------
//  Check if function is 3D

bool    TFunction::IsFunction3D(    Int32 &ioNi,
                                    Int32 &ioNj,
                                    Int32 &ioNk )
{
    if( mDepth != 4 ) return false;

    if( ioNi == 0 && ioNj == 0 && ioNk == 0 ) {
        ioNk = mCountP[1];
        ioNj = mCountP[2];
        ioNi = mCountP[3];
    }
    else if( ioNk != mCountP[1] || ioNj != mCountP[2] || ioNi != mCountP[3] ) return false;

    return true;
}



// ---------------------------------------------------------------------------
//       IsFunctionR
// ---------------------------------------------------------------------------
//  Check if function is R-valued

bool    TFunction::IsFunctionR( Float* &outSP )
{
    if( mCountP[0] != 1 ) return false;

    outSP = mArrayP;
    return true;
}



// ---------------------------------------------------------------------------
//       IsFunctionR2
// ---------------------------------------------------------------------------
//  Check if function is R2-valued

bool    TFunction::IsFunctionR2(    Float* &outS1P,
                                    Float* &outS2P )
{
    return IsFunctionC(outS1P, outS2P);
}



// ---------------------------------------------------------------------------
//       IsFunctionR3
```

```
// ----------------------------------------------------------------------------
//  Check if function is R3-valued

bool    TFunction::IsFunctionR3(    Float* &outS1P,
                                    Float* &outS2P,
                                    Float* &outS3P )
{
    Int32   i, offset = 1;


    if( mCountP[0] != 3 ) return false;

    for( i = 1; i < mDepth; i++ ) offset *= mCountP[i];
    outS1P = mArrayP;
    outS2P = mArrayP + offset;
    outS3P = mArrayP + 2*offset;
    return true;

}



// ----------------------------------------------------------------------------
//       IsFunctionR4
// ----------------------------------------------------------------------------
//  Check if function is R4-valued

bool    TFunction::IsFunctionR4(    Float* &outS1P,
                                    Float* &outS2P,
                                    Float* &outS3P,
                                    Float* &outS4P )
{
    return IsFunctionC2(outS1P, outS2P, outS3P, outS4P);
}



// ----------------------------------------------------------------------------
//       IsFunctionC
// ----------------------------------------------------------------------------
//  Check if function is C-valued

bool    TFunction::IsFunctionC( Float* &outS1P,
                                Float* &outS2P )
{
    Int32   i, offset = 1;


    if( mCountP[0] != 2 ) return false;

    for( i = 1; i < mDepth; i++ ) offset *= mCountP[i];
    outS1P = mArrayP;
    outS2P = mArrayP + offset;
    return true;
}



// ----------------------------------------------------------------------------
//       IsFunctionC2
// ----------------------------------------------------------------------------
//  Check if function is C2-valued

bool    TFunction::IsFunctionC2(    Float* &outS1P,
                                    Float* &outS2P,
                                    Float* &outS3P,
                                    Float* &outS4P )
{
    Int32   i, offset = 1;


    if( mCountP[0] != 4 ) return false;

    for( i = 1; i < mDepth; i++ ) offset *= mCountP[i];
```

```
    outS1P = mArrayP;
    outS2P = mArrayP + offset;
    outS3P = mArrayP + 2*offset;
    outS4P = mArrayP + 3*offset;
    return true;
}


// ---------------------------------------------------------------------------
//       IsFunctionC4
// ---------------------------------------------------------------------------
//  Check if function is C4-valued

bool    TFunction::IsFunctionC4(    Float* &outS1P,
                                    Float* &outS2P,
                                    Float* &outS3P,
                                    Float* &outS4P,
                                    Float* &outS5P,
                                    Float* &outS6P,
                                    Float* &outS7P,
                                    Float* &outS8P )
{
    Int32   i, offset = 1;


    if( mCountP[0] != 8 ) return false;

    for( i = 1; i < mDepth; i++ ) offset *= mCountP[i];
    outS1P = mArrayP;
    outS2P = mArrayP + offset;
    outS3P = mArrayP + 2*offset;
    outS4P = mArrayP + 3*offset;
    outS5P = mArrayP + 4*offset;
    outS6P = mArrayP + 5*offset;
    outS7P = mArrayP + 6*offset;
    outS8P = mArrayP + 7*offset;
    return true;
}



// ---------------------------------------------------------------------------
//       UpdateWindow
// ---------------------------------------------------------------------------
//  Update window

Int32   TFunction::UpdateWindow( void )
{
    TWindow*    theWindow;
    Int32   size, ID;

    size = gWindowList->GetSize();
    for( ID = 0; ID < size; ID++ ) {
        theWindow = gWindowList->Fetch(ID);
        if( theWindow ) {
            if( theWindow->HasCorrectID(mID) ) {
                if( eError == theWindow->Draw(this) ) return eError;
            }
        }
    }

    return eOK;
}



// ---------------------------------------------------------------------------
//       Copy
// ---------------------------------------------------------------------------
//  Copy function object

Int32   TFunction::Copy( TFunction* inFunction )
{
```

```
    Int32   i, size;


    size = 1;
    for( i = 0; i < inFunction->mDepth; i++ ) size *= inFunction->mCountP[i];

    delete [] mArrayP;
    mArrayP = new Float[size];
    if( !mArrayP ) {
        return MLErrorReport(stdlink, "out of memory");
    }
    for( i = 0; i < size; i++ ) mArrayP[i] = inFunction->mArrayP[i];

    delete [] mCountP;
    mCountP = new Int32[inFunction->mDepth];
    if( !mCountP ) {
        return MLErrorReport(stdlink, "out of memory");
    }
    for( i = 0; i < inFunction->mDepth; i++ ) mCountP[i] = inFunction->mCountP[i];

    mDepth = inFunction->mDepth;

    return eOK;
}



// ---------------------------------------------------------------------------
//       SwapArrayPointers
// ---------------------------------------------------------------------------
//  Swap array pointers

void    TFunction::SwapArrayPointers( TFunction* inFunction )
{
    Float   *theArrayP;

    theArrayP = inFunction->mArrayP;
    inFunction->mArrayP = mArrayP;
    mArrayP = theArrayP;
}

└
```

A.2 TOperator Class

A.2.1 TOperator.h

```
┌

// ===========================================================================
//  TOperator.h         (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ===========================================================================

#ifndef _H_TOperator
#define _H_TOperator
#pragma once

#include "TypeDefinition.h"
#include "TFunction.h"
#include "TList.h"


class   TOperator
{
    public:
        virtual Int32   TimeEvolution(  TFunction* inFunction,
                                        Float inTimeStep,
                                        Int32 inFractal,
                                        Int32 inSteps ) { };
        virtual Int32   PutInfo( void ) { };
```

```
        Int32   mID;

    protected:
        Int32   GetFractalNumber(   Int32 inFractal,
                                    Int32 inIndex,
                                    Float &outReal,
                                    Float &outImag );

};

extern TList<TOperator> *gOperatorList;

enum
{
    kScalar = 1 << 0,
    kVector = 1 << 1
};

#endif
```

A.2.2 TOperator.cp

```
// ============================================================================
//  TOperator.cp        (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Abstract class for matrix-operators

#include "MathLinkUtilities.h"
#include "TOperator.h"
#include <math.h>



// ----------------------------------------------------------------------------
//       GetFractalNumber
// ----------------------------------------------------------------------------
//  Calculate fractal numbers

Int32   TOperator::GetFractalNumber(    Int32 inFractal,
                                        Int32 inIndex,
                                        Float &outReal,
                                        Float &outImag )
{
    Int32   i;
    Float   reP, imP, reT, imT, twom, temp;
    Float   zero = 0.0, half = 0.5, one = 1.0, two = 2.0;


    if( inFractal >= 32 ) return eError;

    reP = one;
    imP = zero;

    twom = two;
    for( i = 1; i < inFractal; i++ ) {
        twom += two;
        temp = tan( pi / twom );

        if( inIndex & 1 ) temp = -temp;
        inIndex >>= 1;
        reT = reP - imP * temp;
        imT = imP + reP * temp;
        reP = reT * half;
        imP = imT * half;
    }

    outReal = reP;
```

```
    outImag = imP;

    return eOK;
}
```

A.3 TDomain Class

A.3.1 TDomain.h

```
// ============================================================================
//  TDomain.h           (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TDomain
#define _H_TDomain
#pragma once

#include "TypeDefinition.h"


class   TDomain
{
    public:
                TDomain( void );
                ~TDomain( void );
        Int32   InitPlain2D(    Int8* &outDomP,
                                Float* inD0P,
                                Int32 inNi,
                                Int32 inNj );
        Int32   InitPlain3D(    Int8* &outDomP,
                                Float* inD0P,
                                Int32 inNi,
                                Int32 inNj,
                                Int32 inNk );
        void    ClearC( Float* inS1P,
                        Float* inS2P,
                        Int32 inLen );
        void    ClearC2(    Float* inS1P,
                            Float* inS2P,
                            Float* inS3P,
                            Float* inS4P,
                            Int32 inLen );
        void    ClearC4(    Float* inS1P,
                            Float* inS2P,
                            Float* inS3P,
                            Float* inS4P,
                            Float* inS5P,
                            Float* inS6P,
                            Float* inS7P,
                            Float* inS8P,
                            Int32 inLen );

    private:

        Int8*   mDomainP;

};

#endif
```

A.3.2 TDomain.cp

```
// ==========================================================================
//  TDomain.cp          (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================
//
//  Class for operator-domains

#include "MathLinkUtilities.h"
#include "TDomain.h"



// --------------------------------------------------------------------------
//       TDomain
// --------------------------------------------------------------------------
//  Constructor

TDomain::TDomain( void )
{
    mDomainP = nil;
}



// --------------------------------------------------------------------------
//       ~TDomain
// --------------------------------------------------------------------------
//  Destructor

TDomain::~TDomain( void )
{
    delete [] mDomainP;
}



// --------------------------------------------------------------------------
//       InitPlain2D
// --------------------------------------------------------------------------
//  Create 2D domain array, plain format

Int32   TDomain::InitPlain2D(   Int8* &outDomP,
                                Float* inDOP,
                                Int32 inNi,
                                Int32 inNj )
{
    Float   zero = 0.0;
    Int32   i, j;
    Int8    *domP, type;


    domP = new Int8[inNi*inNj];
    if( !domP )
        return MLErrorReport(stdlink, "out of memory");
    outDomP = mDomainP = domP;

    for( j = 0; j < inNj; j++ ) {
        for( i = 0; i < inNi; i++ ) {
            type = 0xff;
            if( !inDOP || *inDOP++ > zero ) {
                if( i == inNi-1 || i == 0 || j == inNj-1 || j == 0 ) type = ~type;
            }
            else type = ~type;
            *domP++ = type;
        }
    }

    return eOK;
}



// --------------------------------------------------------------------------
//       InitPlain3D
// --------------------------------------------------------------------------
```

```
//  Create 3D domain array, plain format

Int32   TDomain::InitPlain3D(   Int8* &outDomP,
                                Float* inD0P,
                                Int32 inNi,
                                Int32 inNj,
                                Int32 inNk )
{
    Float   zero = 0.0;
    Int32   i, j, k;
    Int8    *domP, type;


    domP = new Int8[inNi*inNj*inNk];
    if( !domP )
        return MLErrorReport(stdlink, "out of memory");
    outDomP = mDomainP = domP;

    for( k = 0; k < inNk; k++ ) {
        for( j = 0; j < inNj; j++ ) {
            for( i = 0; i < inNi; i++ ) {
                type = 0xff;
                if( !inD0P || *inD0P++ > zero ) {
                    if( i == inNi-1 || i == 0 ||
                        j == inNj-1 || j == 0 ||
                        k == inNk-1 || k == 0 )
                        type = ~type;
                }
                else type = ~type;
                *domP++ = type;
            }
        }
    }

    return eOK;
}



// ----------------------------------------------------------------------------
//       ClearC
// ----------------------------------------------------------------------------
//  Clear boundary points

void    TDomain::ClearC(    Float* inS1P,
                            Float* inS2P,
                            Int32 inLen )
{
    Float   zero = 0.0;
    Int32   i;
    Int8    *domP;

    domP = mDomainP;
    for( i = 0; i < inLen; i++ ) {
        if( !domP[i] ) inS1P[i] = inS2P[i] = zero;
    }

}



// ----------------------------------------------------------------------------
//       ClearC2
// ----------------------------------------------------------------------------
//  Clear boundary points

void    TDomain::ClearC2(   Float* inS1P,
                            Float* inS2P,
                            Float* inS3P,
                            Float* inS4P,
                            Int32 inLen )
{
    Float   zero = 0.0;
    Int32   i;
```

```
    Int8    *domP;

    domP = mDomainP;
    for( i = 0; i < inLen; i++ ) {
        if( !domP[i] ) inS1P[i] = inS2P[i] = inS3P[i] = inS4P[i] = zero;
    }


}

// ----------------------------------------------------------------------------
//       ClearC4
// ----------------------------------------------------------------------------
//  Clear boundary points

void    TDomain::ClearC4(   Float* inS1P,
                            Float* inS2P,
                            Float* inS3P,
                            Float* inS4P,
                            Float* inS5P,
                            Float* inS6P,
                            Float* inS7P,
                            Float* inS8P,
                            Int32 inLen )
{
    Float   zero = 0.0;
    Int32   i;
    Int8    *domP;

    domP = mDomainP;
    for( i = 0; i < inLen; i++ ) {
        if( !domP[i] )
            inS1P[i] = inS2P[i] = inS3P[i] = inS4P[i] =
            inS5P[i] = inS6P[i] = inS7P[i] = inS8P[i] = zero;
    }

}
```

A.4 TWindow Class

A.4.1 TWindow.h

```
// ============================================================================
//  TWindow.h           (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TWindow
#define _H_TWindow
#pragma once

#include "TypeDefinition.h"
#include "TFunction.h"
#include "TMovie.h"
#include "TList.h"
#include <QDOffscreen.h>


class   TWindow
{
    public:
                TWindow(    Int32 inFunctionID,
                            Int32 inMode,
                            Int32 inSlice );
                ~TWindow( void );
        Int32   PutInfo( void );
        Int32   Create( void );
        Int32   Draw( TFunction* inFunction );
```

```
    void    Show( void );
    void    Update( void );
    void    Zoom( void );
    bool    HasCorrectID( Int32 inID ) { return mFunctionID == inID; };
    bool    IsMovie( void ) { return mMovieP ? true : false; };
    Int32   BeginMovie( void );
    Int32   EndMovie( void );

    Int32   mID;

private:
    Int32   CalcWindowPosition( Rect &outRect );
    void    RealToRB( Float* inRealP );
    void    RealToBW( Float* inRealP );
    void    ComplexToRGB(   Float* inRealP,
                            Float* inImagP );
    void    AbsCToGray( Float* inRealP,
                        Float* inImagP );
    void    Sigma1ToRB( Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P );
    void    Sigma2ToRB( Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P );
    void    Sigma3ToRB( Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P );
    void    AbsC2ToGray(    Float* inReal1P,
                            Float* inImag1P,
                            Float* inReal2P,
                            Float* inImag2P );
    void    Alpha1ToRB( Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P,
                        Float* inReal3P,
                        Float* inImag3P,
                        Float* inReal4P,
                        Float* inImag4P );
    void    Alpha2ToRB( Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P,
                        Float* inReal3P,
                        Float* inImag3P,
                        Float* inReal4P,
                        Float* inImag4P );
    void    Alpha3ToRB( Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P,
                        Float* inReal3P,
                        Float* inImag3P,
                        Float* inReal4P,
                        Float* inImag4P );
    void    BetaToRB(   Float* inReal1P,
                        Float* inImag1P,
                        Float* inReal2P,
                        Float* inImag2P,
                        Float* inReal3P,
                        Float* inImag3P,
                        Float* inReal4P,
                        Float* inImag4P );
    void    AbsC4ToGray(    Float* inReal1P,
                            Float* inImag1P,
                            Float* inReal2P,
                            Float* inImag2P,
                            Float* inReal3P,
                            Float* inImag3P,
                            Float* inReal4P,
                            Float* inImag4P );
```

```
        Int32       mFunctionID;
        Int32       mMode;
        WindowPtr   mWindowP;
        GWorldPtr   mOffGWorldP;
        Int32       mWidth;
        Int32       mHeight;
        Int32       mDepth;
        Int32       mSlice;
        Int32       mZoom;
        TMovie*     mMovieP;

};

extern TList<TWindow> *gWindowList;

#endif
```

A.4.2 TWindow.cp

```
// ==========================================================================
//  TWindow.cp          (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================
//
//  Class for windows

#include "MathLinkUtilities.h"
#include "TWindow.h"
#include <stdio.h>
#include <math.h>



// --------------------------------------------------------------------------
//       TWindow
// --------------------------------------------------------------------------
//  Constructor

TWindow::TWindow(   Int32 inFunctionID,
                    Int32 inMode,
                    Int32 inSlice )
{
    mID = 0;
    mFunctionID = inFunctionID;
    mMode = inMode;
    mWindowP = nil;
    mOffGWorldP = nil;
    mWidth = 0;
    mHeight = 0;
    mDepth = 0;
    mSlice = inSlice;
    mZoom = 2;
    mMovieP = nil;
}



// --------------------------------------------------------------------------
//       ~TWindow
// --------------------------------------------------------------------------
//  Destructor

TWindow::~TWindow( void )
{
    if( mOffGWorldP ) DisposeGWorld(mOffGWorldP);
    if( mWindowP ) DisposeWindow(mWindowP);
}
```

```
// ---------------------------------------------------------------------------
//       PutInfo
// ---------------------------------------------------------------------------
//  PutInfo

Int32   TWindow::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 3);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Function");
    MLPutFunction(stdlink, "FunctionObject", 1);
    MLPutLongInteger(stdlink, mFunctionID);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mode");
    MLPutLongInteger(stdlink, mMode);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Movie");
    if( mMovieP ) MLPutSymbol(stdlink, "True");
    else MLPutSymbol(stdlink, "False");

    return eOK;
}



// ---------------------------------------------------------------------------
//       Create
// ---------------------------------------------------------------------------
//  Create window with function

Int32   TWindow::Create( void )
{
    Rect    theRect;
    TFunction*  theFunction;


    theFunction = gFunctionList->Fetch(mFunctionID);
    if( !theFunction )
        return MLErrorReport(stdlink, "invalid function ID");

    if( !theFunction->IsFunction2D(mWidth, mHeight) &&
        !theFunction->IsFunction3D(mWidth, mHeight, mDepth) )
        return MLErrorReport(stdlink, "2D or 3D function expected");

    if( mSlice && (mSlice < 0 || mSlice >= mDepth) )
        return MLErrorReport(stdlink, "invalid slice");

    CalcWindowPosition( theRect );

    mWindowP = NewCWindow(  nil,
                            &theRect,
                            "\puntitled",
                            false,
                            zoomDocProc,
                            (WindowPtr) -1,
                            false,
                            0);
    if( !mWindowP )
        return MLErrorReport(stdlink, "out of memory");
    SetWRefCon( mWindowP, (Int32)this );

    SetRect(&theRect, 0, 0, mWidth, mHeight);
    NewGWorld(&mOffGWorldP, 32, &theRect, nil, nil, 0);
    if ( !mOffGWorldP )
        return MLErrorReport(stdlink, "out of memory");

    if( eError == Draw(theFunction) ) return eError;
    return eOK;
}
```

```
// ---------------------------------------------------------------------------
//       Draw
// ---------------------------------------------------------------------------
//  Draw window with function

Int32   TWindow::Draw( TFunction* inFunction )
{
    Float   *realP, *imagP;
    Float   *real1P, *imag1P, *real2P, *imag2P;
    Float   *real3P, *imag3P, *real4P, *imag4P;
    Int8    title[256];
    Int32   offset;
    bool    badMode = true;


    if( mMovieP ) {
        if( eError == mMovieP->AddFrame() ) return eError;
        sprintf(title, "%ld%s%ld%s%ld",mMovieP->GetFrame(),"F",mFunctionID," W",mID);
        SetWTitle(mWindowP, c2pstr(title));
    }

    if( inFunction->IsFunctionC4(   real1P, imag1P, real2P, imag2P,
                                    real3P, imag3P, real4P, imag4P) ) {
        offset = mSlice * mWidth * mHeight;
        real1P += offset;
        imag1P += offset;
        real2P += offset;
        imag2P += offset;
        real3P += offset;
        imag3P += offset;
        real4P += offset;
        imag4P += offset;

        badMode = false;
        switch( mMode ) {
        case 0:
            ComplexToRGB(real1P, imag1P);
            break;
        case 1:
            ComplexToRGB(real2P, imag2P);
            break;
        case 2:
            ComplexToRGB(real3P, imag3P);
            break;
        case 3:
            ComplexToRGB(real4P, imag4P);
            break;
        case 4:
            Alpha1ToRB(real1P, imag1P, real2P, imag2P, real3P, imag3P, real4P, imag4P);
            break;
        case 5:
            Alpha2ToRB(real1P, imag1P, real2P, imag2P, real3P, imag3P, real4P, imag4P);
            break;
        case 6:
            Alpha3ToRB(real1P, imag1P, real2P, imag2P, real3P, imag3P, real4P, imag4P);
            break;
        case 7:
            BetaToRB(real1P, imag1P, real2P, imag2P, real3P, imag3P, real4P, imag4P);
            break;
        case 8:
            AbsC4ToGray(real1P, imag1P, real2P, imag2P, real3P, imag3P, real4P, imag4P);
            break;
        default:
            badMode = true;
            break;
        }
    }

    if( inFunction->IsFunctionC2(real1P, imag1P, real2P, imag2P) ) {
        offset = mSlice * mWidth * mHeight;
        real1P += offset;
```

```
    imag1P += offset;
    real2P += offset;
    imag2P += offset;

    badMode = false;
    switch( mMode ) {
    case 0:
        ComplexToRGB(real1P, imag1P);
        break;
    case 1:
        ComplexToRGB(real2P, imag2P);
        break;
    case 2:
        Sigma1ToRB(real1P, imag1P, real2P, imag2P);
        break;
    case 3:
        Sigma2ToRB(real1P, imag1P, real2P, imag2P);
        break;
    case 4:
        Sigma3ToRB(real1P, imag1P, real2P, imag2P);
        break;
    case 5:
        AbsC2ToGray(real1P, imag1P, real2P, imag2P);
        break;
    default:
        badMode = true;
        break;
    }
}

if( inFunction->IsFunctionC(realP, imagP) ) {
    offset = mSlice * mWidth * mHeight;
    realP += offset;
    imagP += offset;

    badMode = false;
    switch( mMode ) {
    case 0:
        ComplexToRGB(realP, imagP);
        break;
    case 1:
        AbsCToGray(realP, imagP);
        break;
    case 2:
        RealToRB( realP );
        break;
    case 3:
        RealToRB( imagP );
        break;
    default:
        badMode = true;
        break;
    }
}

if( inFunction->IsFunctionR3(real1P, real2P, real3P) ) {
    offset = mSlice * mWidth * mHeight;
    real1P += offset;
    real2P += offset;
    real3P += offset;

    badMode = false;
    switch( mMode ) {
    case 0:
        RealToRB( real1P );
        break;
    case 1:
        RealToRB( real2P );
        break;
    case 2:
        RealToRB( real3P );
        break;
    default:
        badMode = true;
```

```
            break;
        }
    }

    if( inFunction->IsFunctionR( realP ) ) {
        offset = mSlice * mWidth * mHeight;
        realP += offset;

        badMode = false;
        switch( mMode ) {
        case 0:
            RealToRB( realP );
            break;
        case 1:
            RealToBW( realP );
            break;
        default:
            badMode = true;
            break;
        }
    }

    if( badMode )
        return MLErrorReport(stdlink, "view mode not supported");

    Update();
    return eOK;
}



// ---------------------------------------------------------------------------
//       Show
// ---------------------------------------------------------------------------
//  Show window

void    TWindow::Show( void )
{
    char    title[256];


    sprintf(title, "%s%ld%s%ld","F",mFunctionID," W",mID);
    SetWTitle(mWindowP, c2pstr(title));

    ShowWindow(mWindowP);
    SelectWindow(mWindowP);
}



// ---------------------------------------------------------------------------
//       Update
// ---------------------------------------------------------------------------
//  Update window

void    TWindow::Update( void )
{
    WindowPtr   savePtr;
    CGrafPtr    savePort;
    GDHandle    saveDev;
    PixMapHandle    offPixMapHandle;
    Rect    sourceRect, destRect;


    GetPort(&savePtr);
    SetPort(mWindowP);

    GetGWorld( &savePort, &saveDev);
    SetGWorld(mOffGWorldP, nil);

    offPixMapHandle = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapHandle);

    SetGWorld(savePort, saveDev);
```

```
    sourceRect = mOffGWorldP->portRect;
    destRect = mWindowP->portRect;

    CopyBits(   &((GrafPtr) mOffGWorldP)->portBits,
                &((GrafPtr) mWindowP)->portBits,
                &sourceRect,
                &destRect,
                srcCopy + ditherCopy,
                nil);

    UnlockPixels(offPixMapHandle);

    SetPort(savePtr);

}



// ---------------------------------------------------------------------------
//       Zoom
// ---------------------------------------------------------------------------
//  Zoom window

void    TWindow::Zoom( void )
{
    WindowPtr       savePtr;


    if( mZoom == 2 ) mZoom = 1;
    else mZoom = 2;

    GetPort(&savePtr);
    SetPort(mWindowP);

    EraseRect(&mWindowP->portRect);
    SizeWindow(mWindowP, mZoom * mWidth, mZoom * mHeight, false);
    Update();

    SetPort(savePtr);
}



// ---------------------------------------------------------------------------
//       BeginMovie
// ---------------------------------------------------------------------------
//  Begin recording

Int32   TWindow::BeginMovie( void )
{
    Int8    title[256];


    if( mMovieP )
        return MLErrorReport(stdlink, "active movie window");

    mMovieP = new TMovie( mOffGWorldP );
    if( mMovieP == nil ) {
        return MLErrorReport(stdlink, "out of memory");
    }
    if( eError == mMovieP->Begin() ) {
        delete mMovieP;
        mMovieP = nil;
        return eError;
    }

    sprintf(title, "%s%ld%s%ld","F",mFunctionID," W",mID);
    SetWTitle(mWindowP, c2pstr(title));

    MLPutSymbol(stdlink, "Null");
    return eOK;
}
```

```
// ---------------------------------------------------------------------------
//       EndMovie
// ---------------------------------------------------------------------------
//  End recording

Int32   TWindow::EndMovie( void )
{
    Int8    title[256];


    if( !mMovieP )
        return MLErrorReport(stdlink, "inactive movie window");

    if( !mMovieP->GetFrame() )
        if( eError == mMovieP->AddFrame() ) return eError;

    if( eError == mMovieP->End() ) {
        delete mMovieP;
        mMovieP = nil;
        return eError;
    }
    delete mMovieP;
    mMovieP = nil;

    sprintf(title, "%s%ld%s%ld","F",mFunctionID," W",mID);
    SetWTitle(mWindowP, c2pstr(title));

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ---------------------------------------------------------------------------
//       CalcWindowPosition
// ---------------------------------------------------------------------------
//  Calculate window position

Int32   TWindow::CalcWindowPosition( Rect &outRect )
{
    Int32   posx = 2, posy = 40;
    static Int32    count = 0;

    posx += 20 * count;
    posy += 20 * count;
    SetRect(&outRect, posx, posy, posx + mZoom * mWidth, posy + mZoom * mHeight);
    if( count++ == 10 ) count = 0;
    return eOK;
}



// ---------------------------------------------------------------------------
//       RealToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::RealToRB( Float* inRealP )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;
```

```
    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            r = *inRealP++;
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}


// ---------------------------------------------------------------------------
//       RealToBW
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::RealToBW( Float* inRealP )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color;
    Float   zero = 0.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;

    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            if( *inRealP++ < zero ) color = 0x00ffffff;
            else color = 0x00000000;

            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}


// ---------------------------------------------------------------------------
//       ComplexToRGB
// ---------------------------------------------------------------------------
//  Color map
```

```
void    TWindow::ComplexToRGB(  Float* inRealP,
                                Float* inImagP )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re, im, r, s, s0, phi, phi0, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k3pi = 3.0 / pi, k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;

    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re = *inRealP++;
            im = *inImagP++;

            r = hypot( re, im );
            s = k4pi * atan( r );
            phi = k3pi * atan2( im, re );
            phi0 = fabs( phi );

            if( phi0 < one ) { red = one; green = phi0; blue = zero; }
            else {
                if( phi0 < two ) { red = two - phi0; green = one; blue = zero; }
                else { red = zero; green = one; blue = phi0 - two; }
            }

            if( phi < zero ) { temp = green; green = blue; blue = temp; }

            if( s < one ) { red *= s; green *= s; blue *= s; }
            else {
                s = two - s; s0 = one - s;
                red = red * s + s0; green = green * s + s0; blue = blue * s + s0;
            }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       AbsCToGray
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::AbsCToGray(    Float* inRealP,
                                Float* inImagP )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
```

```
    Int32   color, colg;
    Float   re, im, r, gray;
    Float   k2pi = 2.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;

    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re = *inRealP++;
            im = *inImagP++;

            r = re * re + im * im;
            gray = k2pi * atan( r );

            colg = gray * k255;

            color = colg;
            colg <<= 8;
            color += colg;
            colg <<= 8;
            color += colg;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       Sigma1ToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::Sigma1ToRB(    Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;

            r = two * (re1 * re2 + im1 * im2);
```

```
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       Sigma2ToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::Sigma2ToRB(    Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;

            r = two * (re1 * im2 - im1 * re2);
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
```

```
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       Sigma3ToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::Sigma3ToRB(    Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;

            r = re1 * re1 + im1 * im1 - re2 * re2 - im2 * im2;
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}
```

```
// ---------------------------------------------------------------------------
//       AbsC2ToGray
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::AbsC2ToGray(   Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, r, gray;
    Float   k2pi = 2.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;

    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;

            r = re1 * re1 + im1 * im1 + re2 * re2 + im2 * im2;
            gray = k2pi * atan( r );

            colg = gray * k255;

            color = colg;
            colg <<= 8;
            color += colg;
            colg <<= 8;
            color += colg;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       Alpha1ToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::Alpha1ToRB(    Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P,
                                Float* inReal3P,
                                Float* inImag3P,
                                Float* inReal4P,
                                Float* inImag4P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, re3, im3, re4, im4;
```

```
    Float   r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;
            re3 = *inReal3P++;
            im3 = *inImag3P++;
            re4 = *inReal4P++;
            im4 = *inImag4P++;

            r = two * (re1 * re4 + im1 * im4 + re2 * re3 + im2 * im3);
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       Alpha2ToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::Alpha2ToRB(    Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P,
                                Float* inReal3P,
                                Float* inImag3P,
                                Float* inReal4P,
                                Float* inImag4P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, re3, im3, re4, im4;
    Float   r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;
```

```
    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;
            re3 = *inReal3P++;
            im3 = *inImag3P++;
            re4 = *inReal4P++;
            im4 = *inImag4P++;

            r = two * (re1 * im4 - im1 * re4 - re2 * im3 + im2 * re3);
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       Alpha3ToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::Alpha3ToRB(    Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P,
                                Float* inReal3P,
                                Float* inImag3P,
                                Float* inReal4P,
                                Float* inImag4P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, re3, im3, re4, im4;
    Float   r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
```

```
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;
            re3 = *inReal3P++;
            im3 = *inImag3P++;
            re4 = *inReal4P++;
            im4 = *inImag4P++;

            r = two * (re1 * re3 + im1 * im3 - re2 * re4 - im2 * im4);
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       BetaToRB
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::BetaToRB(  Float* inReal1P,
                            Float* inImag1P,
                            Float* inReal2P,
                            Float* inImag2P,
                            Float* inReal3P,
                            Float* inImag3P,
                            Float* inReal4P,
                            Float* inImag4P )
{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, re3, im3, re4, im4;
    Float   r, s, red, green, blue, temp;
    Float   zero = 0.0, one = 1.0, two = 2.0;
    Float   k4pi = 4.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
```

```
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;


    pixoff = mHeight * pixels;
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;
            re3 = *inReal3P++;
            im3 = *inImag3P++;
            re4 = *inReal4P++;
            im4 = *inImag4P++;

            r = re1 * re1 + im1 * im1 + re2 * re2 + im2 * im2;
            r -= re3 * re3 + im3 * im3 + re4 * re4 + im4 * im4;
            s = k4pi * atan( fabs( r ) );

            if( s < one ) { red = s; green = blue = zero; }
            else { red = one; green = blue = s - one; }

            if( r < zero ) { temp = red; red = blue; blue = temp; }

            colr = red * k255;
            colg = green * k255;
            colb = blue * k255;

            color = colb;
            colg <<= 8;
            color += colg;
            colr <<= 16;
            color += colr;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}



// ---------------------------------------------------------------------------
//       AbsC4ToGray
// ---------------------------------------------------------------------------
//  Color map

void    TWindow::AbsC4ToGray(   Float* inReal1P,
                                Float* inImag1P,
                                Float* inReal2P,
                                Float* inImag2P,
                                Float* inReal3P,
                                Float* inImag3P,
                                Float* inReal4P,
                                Float* inImag4P )

{
    PixMapHandle    offPixMapH;
    Int32   *pixelP, pixels, pixoff, i, j;
    Int32   color, colr, colg, colb;
    Float   re1, im1, re2, im2, re3, im3, re4, im4, r, gray;
    Float   k2pi = 2.0 / pi, k255 = 255.0;


    offPixMapH = GetGWorldPixMap(mOffGWorldP);
    LockPixels(offPixMapH);

    pixelP = (Int32*)GetPixBaseAddr(offPixMapH);
    pixels = ((*offPixMapH)->rowBytes & 0x7fff) / 4;

    pixoff = mHeight * pixels;
```

```
    for( j = 0; j < mHeight; j++ ) {
        pixoff -= pixels;
        for( i = 0; i < mWidth; i++ ) {
            re1 = *inReal1P++;
            im1 = *inImag1P++;
            re2 = *inReal2P++;
            im2 = *inImag2P++;
            re3 = *inReal3P++;
            im3 = *inImag3P++;
            re4 = *inReal4P++;
            im4 = *inImag4P++;

            r = re1 * re1 + im1 * im1 + re2 * re2 + im2 * im2;
            r += re3 * re3 + im3 * im3 + re4 * re4 + im4 * im4;
            gray = k2pi * atan( r );

            colg = gray * k255;

            color = colg;
            colg <<= 8;
            color += colg;
            colg <<= 8;
            color += colg;
            *(pixelP + pixoff) = color;
            pixoff++;
        }
        pixoff -= mWidth;
    }

    UnlockPixels(offPixMapH);
}
```

A.5 TMovie Class

A.5.1 TMovie.h

```
// ============================================================================
//  TMovie.h            (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TMovie
#define _H_TMovie
#pragma once

#include "TypeDefinition.h"
#include <ImageCompression.h>
#include <Movies.h>


class   TMovie
{
    public:
                TMovie( GWorldPtr inOffGWorldP );
                ~TMovie( void );
        Int32   GetFrame( void ) { return mFrame; };
        Int32   Begin( void );
        Int32   AddFrame( void );
        Int32   End( void );

        Int32   mID;

    private:
        void    NotifyUser( void );

        GWorldPtr               mOffGWorldP;
        SFReply                 mSFReply;
        FSSpec                  mSpec;
```

```
        Int16                   mResRefNum;
        Movie                   mMovie;
        Track                   mTrack;
        Media                   mMedia;
        ImageSequence           mSequenceID;
        ImageDescriptionHandle  mImageDesc;
        Handle                  mCompressedData;
        Int32                   mFrame;
};

extern bool gSwitchedIn;

#endif
```

A.5.2 TMovie.cp

```
// ==========================================================================
//  TMovie.cp           (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================
//
//  Class for movies

#include "MathLinkUtilities.h"
#include "TMovie.h"
#include <Notification.h>



// --------------------------------------------------------------------------
//       TMovie
// --------------------------------------------------------------------------
//  Constructor

TMovie::TMovie( GWorldPtr inOffGWorldP )
{
    mID = 0;
    mOffGWorldP = inOffGWorldP;
    mResRefNum = 0;
    mMovie = nil;
    mTrack = nil;
    mMedia = nil;
    mSequenceID = 0;
    mImageDesc = nil;
    mCompressedData = nil;
    mFrame = 0;
}



// --------------------------------------------------------------------------
//       ~TMovie
// --------------------------------------------------------------------------
//  Destructor

TMovie::~TMovie( void )
{

}



// --------------------------------------------------------------------------
//       Begin
// --------------------------------------------------------------------------
//  Begin movie recording

Int32   TMovie::Begin( void )
{
```

```
OSErr err = noErr;
Point where = {-1, -1};
CodecType   codecKind = 'raw ';
Int32   maxCompressedSize;
DlgHookUPP  MLEventLoopUPP;


NotifyUser();

err = EnterMovies();
if( err != noErr )
    return MLErrorReport(stdlink, "enter movies failed");

SFPutFile(where, "\pEnter movie file name:", "\pMovie", nil, &mSFReply);
if (!mSFReply.good)
    return MLErrorReport(stdlink, "open movie file failed");

FSMakeFSSpec(mSFReply.vRefNum, 0, mSFReply.fName, &mSpec);

err = CreateMovieFile(  &mSpec,
                        'TVOD',
                        smCurrentScript,
                        createMovieFileDeleteCurFile,
                        &mResRefNum,
                        &mMovie);
if( err != noErr )
    return MLErrorReport(stdlink, "create movie file failed");

mTrack = NewMovieTrack( mMovie,
                        FixRatio(mOffGWorldP->portRect.right, 1),
                        FixRatio(mOffGWorldP->portRect.bottom, 1),
                        kNoVolume);
if( GetMoviesError() != noErr )
    return MLErrorReport(stdlink, "create new movie track failed");

mMedia = NewTrackMedia( mTrack,
                        VideoMediaType,
                        600,
                        nil,
                        0);
if( GetMoviesError() != noErr )
    return MLErrorReport(stdlink, "create new track media failed");

err = BeginMediaEdits(mMedia);
if( err != noErr )
    return MLErrorReport(stdlink, "begin media edits failed");

LockPixels(mOffGWorldP->portPixMap);

mImageDesc = (ImageDescriptionHandle)NewHandle(4);
if( MemError() != noErr )
    return MLErrorReport(stdlink, "out of memory");

err = CompressSequenceBegin (   &mSequenceID,
                                mOffGWorldP->portPixMap,
                                nil,
                                &mOffGWorldP->portRect,
                                &mOffGWorldP->portRect,
                                0,
                                codecKind,
                                (CompressorComponent) anyCodec,
                                codecHighQuality,   // spatial quality
                                codecMinQuality,    // temporal quality
                                50,
                                nil,
                                codecFlagUpdatePrevious,
                                mImageDesc);
if( err != noErr )
    return MLErrorReport(stdlink, "compress sequence begin failed");

err = GetMaxCompressionSize(    mOffGWorldP->portPixMap,
                                &mOffGWorldP->portRect,
                                0,
                                codecHighQuality,   // spatial quality
```

```
                                codecKind,
                                (CompressorComponent)anyCodec,
                                &maxCompressedSize);
    if( err != noErr )
        return MLErrorReport(stdlink, "get maximum compression size failed");

    UnlockPixels(mOffGWorldP->portPixMap);

    mCompressedData = NewHandle(maxCompressedSize);
    if( MemError() != noErr )
        return MLErrorReport(stdlink, "out of memory");

    MoveHHi(mCompressedData);
    HLock(mCompressedData);

    return eOK;
}



// ---------------------------------------------------------------------------
//       AddFrame
// ---------------------------------------------------------------------------
//  Add frame to movie

Int32   TMovie::AddFrame( void )
{
    UInt8   similarity = 0;
    TimeValue   sampleTime;
    OSErr err = noErr;
    Ptr compressedDataPtr;
    long    compressedSize;


    LockPixels(mOffGWorldP->portPixMap);
    compressedDataPtr = StripAddress(*mCompressedData);

    err = CompressSequenceFrame(    mSequenceID,
                                    mOffGWorldP->portPixMap,
                                    &mOffGWorldP->portRect,
                                    codecFlagUpdatePrevious,
                                    compressedDataPtr,
                                    &compressedSize,
                                    &similarity,
                                    nil);
    if( err != noErr )
        return MLErrorReport(stdlink, "compress sequence frame failed");

    err = AddMediaSample(   mMedia,
                            mCompressedData,
                            0,
                            compressedSize,
                            (TimeValue)60,
                            (SampleDescriptionHandle)mImageDesc,
                            1,
                            similarity?mediaSampleNotSync:0,
                            &sampleTime);
    if( err != noErr )
        return MLErrorReport(stdlink, "add media sample failed");

    UnlockPixels(mOffGWorldP->portPixMap);
    mFrame++;

    return eOK;
}



// ---------------------------------------------------------------------------
//       End
// ---------------------------------------------------------------------------
//  End movie recording

Int32   TMovie::End( void )
```

```
{
    OSErr err = noErr;
    short resId = 0;


    CDSequenceEnd(mSequenceID);

    DisposeHandle(mCompressedData);
    DisposeHandle((Handle)mImageDesc);

    err = EndMediaEdits(mMedia);
    if( err != noErr )
        return MLErrorReport(stdlink, "end media edits failed");

    err = InsertMediaIntoTrack( mTrack,
                                0,
                                0,
                                GetMediaDuration(mMedia),
                                fixed1);
    if( err != noErr )
        return MLErrorReport(stdlink, "insert media into track failed");

    err = AddMovieResource( mMovie,
                            mResRefNum,
                            &resId,
                            mSFReply.fName);
    if( err != noErr )
        return MLErrorReport(stdlink, "add movie resource failed");

    CloseMovieFile(mResRefNum);
    DisposeMovie(mMovie);

    ExitMovies();

    return eOK;
}



// ----------------------------------------------------------------------------
//       NotifyUser
// ----------------------------------------------------------------------------
//  Notify user

void    TMovie::NotifyUser( void )
{
    NMRec   notRec;
    Handle  iconH;
    Int32   i;


    if( !gSwitchedIn ) {
        iconH = GetResource('SICN', 128);
        notRec.qType = 8;
        notRec.nmMark = 1;
        notRec.nmIcon = iconH;
        notRec.nmSound = (Handle)-1;
        notRec.nmStr = nil;
        notRec.nmResp = nil;
        notRec.nmRefCon = 0;

        NMInstall(&notRec);
        while( !gSwitchedIn ) MLEventLoop();
        for( i = 0; i < 64; i++ ) MLEventLoop();
        NMRemove(&notRec);
        if( iconH ) ReleaseResource(iconH);
    }
}
```

└

A.6 TList Class

A.6.1 TList.h

```
// ==========================================================================
//  TList.h             (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================
//
//  A template class for a list of items

#ifndef _H_TList
#define _H_TList
#pragma once


template <class T>
class   TList
{
    public:
                TList( void );
                ~TList( void );
        bool    New( Int32 inSize );
        Int32   Insert( T* inItem );
        T*      Remove( Int32 inID );
        T*      Fetch( Int32 inID );
        Int32   GetSize( void ) { return mSize; };

    private:
        T**     mItemH;
        Int32   mSize;
        Int32   mID;

};



// ---------------------------------------------------------------------------
//       TList
// ---------------------------------------------------------------------------
//  Constructor

template <class T>
TList<T>::TList( void )
{
    mItemH = nil;
    mSize = 0;
    mID = 0;
}



// ---------------------------------------------------------------------------
//       ~TList
// ---------------------------------------------------------------------------
//  Destructor

template <class T>
TList<T>::~TList( void )
{
    delete [] mItemH;
}



// ---------------------------------------------------------------------------
//       New
// ---------------------------------------------------------------------------
//  Create a new list

template <class T>
bool    TList<T>::New( Int32 inSize )
{
```

```
    Int32   i;


    mItemH = new T*[inSize];
    if( mItemH ) {
        for( i = 0; i < inSize; i++ )
            mItemH[i] = nil;
        mSize = inSize;
        return true;
    }
    return false;
}



// ---------------------------------------------------------------------------
//       Insert
// ---------------------------------------------------------------------------
//  Insert the object pointer into the list

template <class T>
Int32   TList<T>::Insert( T* inItem )
{
    mID++;
    if( (mID >= 0) && (mID < mSize) ) {
        mItemH[mID] = inItem;
        return mID;
    }
    mID = mSize;
    return 0;
}



// ---------------------------------------------------------------------------
//       Remove
// ---------------------------------------------------------------------------
//  Remove the object pointer from the list

template <class T>
T*  TList<T>::Remove( Int32 inID )
{
    T*  item;

    if( (inID >= 0) && (inID < mSize) ) {
        item = mItemH[inID];
        mItemH[inID] = nil;
        return item;
    }
    return nil;
}



// ---------------------------------------------------------------------------
//       Fetch
// ---------------------------------------------------------------------------
//  Fetch the object pointer from the list

template <class T>
T*  TList<T>::Fetch( Int32 inID )
{
    if( (mID >= 0) && (mID < mSize) ) {
        return mItemH[inID];
    }
    return nil;
}


#endif



└
```

A.7 TSchroedinger2D Class

A.7.1 TSchroedinger2D.h

```
// ============================================================================
//  TSchroedinger2D.h   (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TSchroedinger2D
#define _H_TSchroedinger2D
#pragma once

#include "TypeDefinition.h"
#include "TOperator.h"


class   TSchroedinger2D : public TOperator
{
    public:
                TSchroedinger2D(    Int32 inScalarID,
                                    Int32 inVectorID,
                                    Int32 inDomainID,
                                    Float inMass,
                                    Float inCharge,
                                    Float inUnits );
                ~TSchroedinger2D( void );
        Int32   TimeEvolution(  TFunction* inFunction,
                                Float inTimeStep,
                                Int32 inFractal,
                                Int32 inSteps );
        Int32   PutInfo( void );

    private:
        void    Kernel( Float* rePsiP,
                        Float* imPsiP,
                        Float* rePhiP,
                        Float* imPhiP,
                        Float* v0P,
                        Float* w0P,
                        Float* a1P,
                        Float* a2P,
                        Int8* domP,
                        Float re,
                        Float im,
                        Int32 ni,
                        Int32 nj );

        Int32   mScalarID;
        Int32   mVectorID;
        Int32   mDomainID;
        Float   mCharge;
        Float   mMass;
        Float   mUnits;

};

#endif
```

A.7.2 TSchroedinger2D.cp

```
// ============================================================================
//  TSchroedinger2D.cp  (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Class for Schroedinger operators
```

```
#include "MathLinkUtilities.h"
#include "TSchroedinger2D.h"
#include "TDomain.h"



// ------------------------------------------------------------------------
//       TSchroedinger2D
// ------------------------------------------------------------------------
//  Constructor

TSchroedinger2D::TSchroedinger2D(   Int32 inScalarID,
                                    Int32 inVectorID,
                                    Int32 inDomainID,
                                    Float inMass,
                                    Float inCharge,
                                    Float inUnits )
{
    mScalarID = inScalarID;
    mVectorID = inVectorID;
    mDomainID = inDomainID;
    mMass = inMass;
    mCharge = inCharge;
    mUnits = inUnits;
}



// ------------------------------------------------------------------------
//       ~TSchroedinger2D
// ------------------------------------------------------------------------
//  Destructor

TSchroedinger2D::~TSchroedinger2D( void )
{

}



// ------------------------------------------------------------------------
//       PutInfo
// ------------------------------------------------------------------------
//  PutInfo

Int32   TSchroedinger2D::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 7);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Type");
    MLPutSymbol(stdlink, "Schroedinger2D");

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "ScalarPotential");
    if( !mScalarID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mScalarID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "VectorPotential");
    if( !mVectorID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mVectorID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Domain");
    if( !mDomainID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
```

```
        MLPutInteger(stdlink, mDomainID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mass");
    MLPutDouble(stdlink, mMass);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Charge");
    MLPutDouble(stdlink, mCharge);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Units");
    MLPutDouble(stdlink, mUnits);

    return eOK;
}



// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  TimeEvolution

Int32   TSchroedinger2D::TimeEvolution( TFunction* inFunction,
                                        Float inTimeStep,
                                        Int32 inFractal,
                                        Int32 inSteps )
{
    Int32   ni, nj, i, n, s;
    Float   *rePsiP, *imPsiP, *rePhiP, *imPhiP;
    Float   *v0P, *w0P, *a1P, *a2P, *d0P, *tempP, re, im;
    TFunction   *scalarP, *vectorP, *domainP, tempFunction;
    TDomain tempDomain;
    Int8    *domP;


    if( inFractal < 0 || inFractal >= 16 )
        return MLErrorReport(stdlink, "fractal order is out of range");

    ni = nj = 0;
    if( !inFunction->IsFunction2D(ni, nj) )
        return MLErrorReport(stdlink, "two-dimensional wavefunction expected");
    if( !inFunction->IsFunctionC(rePsiP, imPsiP) )
        return MLErrorReport(stdlink, "complex wavefunction expected");

    v0P = w0P = a1P = a2P = d0P = nil;
    if( mScalarID ) {   //  scalar potential
        scalarP = gFunctionList->Fetch(mScalarID);
        if( !scalarP )
            return MLErrorReport(stdlink, "invalid function ID for scalar potential");
        if( !scalarP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "scalar potential is not compatible");
        if( !scalarP->IsFunctionC(v0P, w0P) )
            return MLErrorReport(stdlink, "scalar potential is not C-valued");
    }
    if( mVectorID ) {   //  vector potential
        vectorP = gFunctionList->Fetch(mVectorID);
        if( !vectorP )
            return MLErrorReport(stdlink, "invalid function ID for vector potential");
        if( !vectorP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "vector potential is not compatible");
        if( !vectorP->IsFunctionR2(a1P, a2P) )
            return MLErrorReport(stdlink, "vector potential is not R2-valued");
    }
    if( mDomainID ) {   //  domain function
        domainP = gFunctionList->Fetch(mDomainID);
        if( !domainP )
            return MLErrorReport(stdlink, "invalid function ID for domain function");
        if( !domainP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "domain function is not compatible");
        if( !domainP->IsFunctionR(d0P) )
            return MLErrorReport(stdlink, "domain function is not R-valued");
```

```
    }

    domP = nil;
    if( eError == tempDomain.InitPlain2D(domP, d0P, ni, nj) ) return eError;
    tempDomain.ClearC(rePsiP, imPsiP, ni*nj);

    if( eError == tempFunction.Copy(inFunction) ) return eError;
    tempFunction.IsFunctionC(rePhiP, imPhiP);

    for( s = 0; s < inSteps; s++ ) {
        n = 1 << inFractal >> 1;    //fractal iteration
        for( i = 0; i < n; i++ ) {
            GetFractalNumber( inFractal, i, re, im );
            re *= inTimeStep;
            im *= inTimeStep;
            Kernel( rePsiP, imPsiP, rePhiP, imPhiP,
                    v0P, w0P, a1P, a2P, domP, re, im, ni, nj);

            tempP = rePsiP; rePsiP = rePhiP; rePhiP = tempP;
            tempP = imPsiP; imPsiP = imPhiP; imPhiP = tempP;
        }
        if( n == 1 ) inFunction->SwapArrayPointers(&tempFunction);

        if( eError == inFunction->UpdateWindow() ) return eError;

        MLCallYieldFunction(MLYieldFunction(stdlink), stdlink, (MLYieldParameters)0);
        if(MLAbort) {
            MLPutFunction(stdlink, "Abort", 0);
            return eError;
        }
    }

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ------------------------------------------------------------------------
//       Kernel
// ------------------------------------------------------------------------

void    TSchroedinger2D::Kernel(    Float* rePsiP,
                                    Float* imPsiP,
                                    Float* rePhiP,
                                    Float* imPhiP,
                                    Float* v0P,
                                    Float* w0P,
                                    Float* a1P,
                                    Float* a2P,
                                    Int8* domP,
                                    Float reZ,
                                    Float imZ,
                                    Int32 ni,
                                    Int32 nj)
{
    Float   rePsiC, imPsiC, reEtaC, imEtaC;
    Float   rePsiR, imPsiR, rePsiL, imPsiL;
    Float   rePsiU, imPsiU, rePsiD, imPsiD;
    Float   reT, imT;
    Float   v0C, w0C;
    Float   a1C, a1R, a1L;
    Float   a2C, a2U, a2D;
    Float   chh, ceh, deh, cee, e;
    Float   one = 1.0, two = 2.0, four = 4.0;
    Int32   i, mode = 0;


    chh = one / (two * mMass * mUnits * mUnits);
    ceh = mCharge / (two * mMass * mUnits);
    deh = ceh / two;
    cee = mCharge * mCharge / (two * mMass);
    e = mCharge;
```

```
    if( mScalarID ) mode |= kScalar;
    if( mVectorID ) mode |= kVector;

    for( i = 0; i < ni*nj; i++ ) {
        if( *domP++ ) {
            rePsiR = *(rePsiP + 1);
            imPsiR = *(imPsiP + 1);
            rePsiL = *(rePsiP +-1);
            imPsiL = *(imPsiP +-1);
            rePsiU = *(rePsiP + ni);
            imPsiU = *(imPsiP + ni);
            rePsiD = *(rePsiP +-ni);
            imPsiD = *(imPsiP +-ni);
            rePsiC = *rePsiP++;
            imPsiC = *imPsiP++;
            reT = rePsiR + rePsiL + rePsiU + rePsiD;
            imT = imPsiR + imPsiL + imPsiU + imPsiD;
            reEtaC = chh * (four * rePsiC - reT);
            imEtaC = chh * (four * imPsiC - imT);
            if( mode & kVector ) {
                a1R = *(a1P + 1);
                a1L = *(a1P +-1);
                a2U = *(a2P + ni);
                a2D = *(a2P +-ni);
                a1C = *a1P++;
                a2C = *a2P++;
                reEtaC -= ceh * a1C * (imPsiR - imPsiL);
                imEtaC += ceh * a1C * (rePsiR - rePsiL);
                reEtaC -= ceh * a2C * (imPsiU - imPsiD);
                imEtaC += ceh * a2C * (rePsiU - rePsiD);
                reT = cee * (a1C * a1C + a2C * a2C);
                reEtaC += reT * rePsiC;
                imEtaC += reT * imPsiC;
                imT = deh * (a1R - a1L + a2U - a2D);
                reEtaC -= imT * imPsiC;
                imEtaC += imT * rePsiC;
            }
            if( mode & kScalar ) {
                v0C = *v0P++;
                w0C = *w0P++;
                reEtaC += e * v0C * rePsiC;
                imEtaC += e * v0C * imPsiC;
                reEtaC -= e * w0C * imPsiC;
                imEtaC += e * w0C * rePsiC;
            }
            *rePhiP++ = rePsiC + imZ * reEtaC + reZ * imEtaC;
            *imPhiP++ = imPsiC - reZ * reEtaC + imZ * imEtaC;
        }
        else {
            rePsiP++;
            imPsiP++;
            if( mode & kScalar ) { v0P++; w0P++; }
            rePhiP++;
            imPhiP++;
            if( mode & kVector ) { a1P++; a2P++; }
        }
    }

}
```

A.8 TSchroedinger3D Class

A.8.1 TSchroedinger3D.h

```
// ============================================================================
//  TSchroedinger3D.h   (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
```

```
#ifndef _H_TSchroedinger3D
#define _H_TSchroedinger3D
#pragma once

#include "TypeDefinition.h"
#include "TOperator.h"


class   TSchroedinger3D : public TOperator
{
    public:
                TSchroedinger3D(    Int32 inScalarID,
                                    Int32 inVectorID,
                                    Int32 inDomainID,
                                    Float inMass,
                                    Float inCharge,
                                    Float inUnits );
                ~TSchroedinger3D( void );
        Int32   TimeEvolution(  TFunction* inFunction,
                                Float inTimeStep,
                                Int32 inFractal,
                                Int32 inSteps );
        Int32   PutInfo( void );

    private:
        void    Kernel( Float* rePsiP,
                        Float* imPsiP,
                        Float* rePhiP,
                        Float* imPhiP,
                        Float* v0P,
                        Float* w0P,
                        Float* a1P,
                        Float* a2P,
                        Float* a3P,
                        Int8* domP,
                        Float reZ,
                        Float imZ,
                        Int32 ni,
                        Int32 nj,
                        Int32 nk );

        Int32   mScalarID;
        Int32   mVectorID;
        Int32   mDomainID;
        Float   mCharge;
        Float   mMass;
        Float   mUnits;

};

#endif
```

A.8.2 TSchroedinger3D.cp

```
// ==========================================================================
//  TSchroedinger3D.cp  (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================
//
//  Class for Schroedinger operators

#include "MathLinkUtilities.h"
#include "TSchroedinger3D.h"
#include "TDomain.h"



// --------------------------------------------------------------------------
```

```
//       TSchroedinger3D
// ---------------------------------------------------------------------------
//  Constructor

TSchroedinger3D::TSchroedinger3D(   Int32 inScalarID,
                                    Int32 inVectorID,
                                    Int32 inDomainID,
                                    Float inMass,
                                    Float inCharge,
                                    Float inUnits )
{
    mScalarID = inScalarID;
    mVectorID = inVectorID;
    mDomainID = inDomainID;
    mMass = inMass;
    mCharge = inCharge;
    mUnits = inUnits;
}



// ---------------------------------------------------------------------------
//       ~TSchroedinger3D
// ---------------------------------------------------------------------------
//  Destructor

TSchroedinger3D::~TSchroedinger3D( void )
{

}



// ---------------------------------------------------------------------------
//       PutInfo
// ---------------------------------------------------------------------------
//  PutInfo

Int32   TSchroedinger3D::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 7);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Type");
    MLPutSymbol(stdlink, "Schroedinger3D");

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "ScalarPotential");
    if( !mScalarID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mScalarID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "VectorPotential");
    if( !mVectorID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mVectorID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Domain");
    if( !mDomainID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mDomainID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mass");
    MLPutDouble(stdlink, mMass);

    MLPutFunction(stdlink, "Rule", 2);
```

```
    MLPutSymbol(stdlink, "Charge");
    MLPutDouble(stdlink, mCharge);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Units");
    MLPutDouble(stdlink, mUnits);

    return eOK;
}



// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  TimeEvolution

Int32   TSchroedinger3D::TimeEvolution( TFunction* inFunction,
                                        Float inTimeStep,
                                        Int32 inFractal,
                                        Int32 inSteps )
{
    Int32   ni, nj, nk, i, n, s;
    Float   *rePsiP, *imPsiP, *rePhiP, *imPhiP;
    Float   *v0P, *w0P, *a1P, *a2P, *a3P, *d0P, *tempP, re, im;
    TFunction   *scalarP, *vectorP, *domainP, tempFunction;
    TDomain tempDomain;
    Int8    *domP;


    if( inFractal < 0 || inFractal >= 16 )
        return MLErrorReport(stdlink, "fractal order is out of range");

    ni = nj = nk = 0;
    if( !inFunction->IsFunction3D(ni, nj, nk) )
        return MLErrorReport(stdlink, "three-dimensional wavefunction expected");
    if( !inFunction->IsFunctionC(rePsiP, imPsiP) )
        return MLErrorReport(stdlink, "complex wavefunction expected");

    v0P = w0P = a1P = a2P = a3P = d0P = nil;
    if( mScalarID ) {   //  scalar potential
        scalarP = gFunctionList->Fetch(mScalarID);
        if( !scalarP )
            return MLErrorReport(stdlink, "invalid function ID for scalar potential");
        if( !scalarP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "scalar potential is not compatible");
        if( !scalarP->IsFunctionC(v0P, w0P) )
            return MLErrorReport(stdlink, "scalar potential is not C-valued");
    }
    if( mVectorID ) {   //  vector potential
        vectorP = gFunctionList->Fetch(mVectorID);
        if( !vectorP )
            return MLErrorReport(stdlink, "invalid function ID for vector potential");
        if( !vectorP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "vector potential is not compatible");
        if( !vectorP->IsFunctionR3(a1P, a2P, a3P) )
            return MLErrorReport(stdlink, "vector potential is not R3-valued");
    }
    if( mDomainID ) {   //  domain function
        domainP = gFunctionList->Fetch(mDomainID);
        if( !domainP )
            return MLErrorReport(stdlink, "invalid function ID for domain function");
        if( !domainP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "domain function is not compatible");
        if( !domainP->IsFunctionR(d0P) )
            return MLErrorReport(stdlink, "domain function is not R-valued");
    }

    domP = nil;
    if( eError == tempDomain.InitPlain3D(domP, d0P, ni, nj, nk) ) return eError;
    tempDomain.ClearC(rePsiP, imPsiP, ni*nj*nk);

    if( eError == tempFunction.Copy(inFunction) ) return eError;
    tempFunction.IsFunctionC(rePhiP, imPhiP);
```

```
    for( s = 0; s < inSteps; s++ ) {
        n = 1 << inFractal >> 1;    //fractal iteration
        for( i = 0; i < n; i++ ) {
            GetFractalNumber( inFractal, i, re, im );
            re *= inTimeStep;
            im *= inTimeStep;
            Kernel( rePsiP, imPsiP, rePhiP, imPhiP,
                    v0P, w0P, a1P, a2P, a3P, domP, re, im, ni, nj, nk);

            tempP = rePsiP; rePsiP = rePhiP; rePhiP = tempP;
            tempP = imPsiP; imPsiP = imPhiP; imPhiP = tempP;
        }
        if( n == 1 ) inFunction->SwapArrayPointers(&tempFunction);

        if( eError == inFunction->UpdateWindow() ) return eError;

        MLCallYieldFunction(MLYieldFunction(stdlink), stdlink, (MLYieldParameters)0);
        if(MLAbort) {
            MLPutFunction(stdlink, "Abort", 0);
            return eError;
        }
    }

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ---------------------------------------------------------------------------
//       Kernel
// ---------------------------------------------------------------------------

void    TSchroedinger3D::Kernel(    Float* rePsiP,
                                    Float* imPsiP,
                                    Float* rePhiP,
                                    Float* imPhiP,
                                    Float* v0P,
                                    Float* w0P,
                                    Float* a1P,
                                    Float* a2P,
                                    Float* a3P,
                                    Int8* domP,
                                    Float reZ,
                                    Float imZ,
                                    Int32 ni,
                                    Int32 nj,
                                    Int32 nk )
{
    Float   rePsiC, imPsiC, reEtaC, imEtaC;
    Float   rePsiR, imPsiR, rePsiL, imPsiL;
    Float   rePsiU, imPsiU, rePsiD, imPsiD;
    Float   rePsiF, imPsiF, rePsiB, imPsiB;
    Float   reT, imT;
    Float   v0C, w0C;
    Float   a1C, a1R, a1L;
    Float   a2C, a2U, a2D;
    Float   a3C, a3F, a3B;
    Float   chh, ceh, deh, cee, e;
    Float   one = 1.0, two = 2.0, six = 6.0;
    Int32   i, mode = 0;


    chh = one / (two * mMass * mUnits * mUnits);
    ceh = mCharge / (two * mMass * mUnits);
    deh = ceh / two;
    cee = mCharge * mCharge / (two * mMass);
    e = mCharge;

    if( mScalarID ) mode |= kScalar;
    if( mVectorID ) mode |= kVector;

    for( i = 0; i < ni*nj*nk; i++ ) {
```

```
        if( *domP++ ) {
            rePsiR = *(rePsiP + 1);
            imPsiR = *(imPsiP + 1);
            rePsiL = *(rePsiP +-1);
            imPsiL = *(imPsiP +-1);
            rePsiU = *(rePsiP + ni);
            imPsiU = *(imPsiP + ni);
            rePsiD = *(rePsiP +-ni);
            imPsiD = *(imPsiP +-ni);
            rePsiF = *(rePsiP + ni*nj);
            imPsiF = *(imPsiP + ni*nj);
            rePsiB = *(rePsiP +-ni*nj);
            imPsiB = *(imPsiP +-ni*nj);
            rePsiC = *rePsiP++;
            imPsiC = *imPsiP++;
            reT = rePsiR + rePsiL + rePsiU + rePsiD + rePsiF + rePsiB;
            imT = imPsiR + imPsiL + imPsiU + imPsiD + imPsiF + imPsiB;
            reEtaC = chh * (six * rePsiC - reT);
            imEtaC = chh * (six * imPsiC - imT);
            if( mode & kVector ) {
                a1R = *(a1P + 1);
                a1L = *(a1P +-1);
                a2U = *(a2P + ni);
                a2D = *(a2P +-ni);
                a3F = *(a3P + ni*nj);
                a3B = *(a3P +-ni*nj);
                a1C = *a1P++;
                a2C = *a2P++;
                a3C = *a3P++;
                reEtaC -= ceh * a1C * (imPsiR - imPsiL);
                imEtaC += ceh * a1C * (rePsiR - rePsiL);
                reEtaC -= ceh * a2C * (imPsiU - imPsiD);
                imEtaC += ceh * a2C * (rePsiU - rePsiD);
                reEtaC -= ceh * a3C * (imPsiF - imPsiB);
                imEtaC += ceh * a3C * (rePsiF - rePsiB);
                reT = cee * (a1C * a1C + a2C * a2C + a3C * a3C);
                reEtaC += reT * rePsiC;
                imEtaC += reT * imPsiC;
                imT = deh * (a1R - a1L + a2U - a2D + a3F - a3B);
                reEtaC -= imT * imPsiC;
                imEtaC += imT * rePsiC;
            }
            if( mode & kScalar ) {
                v0C = *v0P++;
                w0C = *w0P++;
                reEtaC += e * v0C * rePsiC;
                imEtaC += e * v0C * imPsiC;
                reEtaC -= e * w0C * imPsiC;
                imEtaC += e * w0C * rePsiC;
            }
            *rePhiP++ = rePsiC + imZ * reEtaC + reZ * imEtaC;
            *imPhiP++ = imPsiC - reZ * reEtaC + imZ * imEtaC;
        }
        else {
            rePsiP++;
            imPsiP++;
            if( mode & kScalar ) { v0P++; *w0P++; }
            rePhiP++;
            imPhiP++;
            if( mode & kVector ) { a1P++; a2P++; a3P++; }
        }
    }

}
```

└

A.9 TPauli2D Class

A.9.1 TPauli2D.h

```
// ============================================================================
//  TPauli2D.h          (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TPauli2D
#define _H_TPauli2D
#pragma once

#include "TypeDefinition.h"
#include "TOperator.h"


class   TPauli2D : public TOperator
{
    public:
                TPauli2D(   Int32 inScalarID,
                            Int32 inVectorID,
                            Int32 inDomainID,
                            Float inMass,
                            Float inCharge,
                            Float inUnits );
                ~TPauli2D( void );
        Int32   TimeEvolution(  TFunction* inFunction,
                                Float inTimeStep,
                                Int32 inFractal,
                                Int32 inSteps );
        Int32   PutInfo( void );

    private:
        void    Kernel( Float* rePsiP,
                        Float* imPsiP,
                        Float* rePhiP,
                        Float* imPhiP,
                        Float* v0P,
                        Float* w0P,
                        Float* a1P,
                        Float* a2P,
                        Int8* domP,
                        Float re,
                        Float im,
                        Int32 ni,
                        Int32 nj );

        Int32   mScalarID;
        Int32   mVectorID;
        Int32   mDomainID;
        Float   mCharge;
        Float   mMass;
        Float   mUnits;

};

#endif
```

A.9.2 TPauli2D.cp

```
// ============================================================================
//  TPauli2D.cp         (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Class for Pauli operators
```

```
#include "MathLinkUtilities.h"
#include "TPauli2D.h"
#include "TDomain.h"



// ---------------------------------------------------------------------------
//       TPauli2D
// ---------------------------------------------------------------------------
//  Constructor

TPauli2D::TPauli2D( Int32 inScalarID,
                    Int32 inVectorID,
                    Int32 inDomainID,
                    Float inMass,
                    Float inCharge,
                    Float inUnits )
{
    mScalarID = inScalarID;
    mVectorID = inVectorID;
    mDomainID = inDomainID;
    mMass = inMass;
    mCharge = inCharge;
    mUnits = inUnits;
}



// ---------------------------------------------------------------------------
//       ~TPauli2D
// ---------------------------------------------------------------------------
//  Destructor

TPauli2D::~TPauli2D( void )
{

}



// ---------------------------------------------------------------------------
//       PutInfo
// ---------------------------------------------------------------------------
//  PutInfo

Int32   TPauli2D::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 7);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Type");
    MLPutSymbol(stdlink, "Pauli2D");

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "ScalarPotential");
    if( !mScalarID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mScalarID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "VectorPotential");
    if( !mVectorID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mVectorID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Domain");
    if( !mDomainID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mDomainID);
```

```
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mass");
    MLPutDouble(stdlink, mMass);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Charge");
    MLPutDouble(stdlink, mCharge);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Units");
    MLPutDouble(stdlink, mUnits);

    return eOK;
}



// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  TimeEvolution

Int32   TPauli2D::TimeEvolution(    TFunction* inFunction,
                                    Float inTimeStep,
                                    Int32 inFractal,
                                    Int32 inSteps )
{
    Int32   ni, nj, i, n, s;
    Float   *rePsiP, *imPsiP, *rePhiP, *imPhiP;
    Float   *v0P, *w0P, *a1P, *a2P, *d0P, *tempP, re, im;
    TFunction   *scalarP, *vectorP, *domainP, tempFunction;
    TDomain tempDomain;
    Int8    *domP;


    if( inFractal < 0 || inFractal >= 16 )
        return MLErrorReport(stdlink, "fractal order is out of range");

    ni = nj = 0;
    if( !inFunction->IsFunction2D(ni, nj) )
        return MLErrorReport(stdlink, "two-dimensional wavefunction expected");
    if( !inFunction->IsFunctionC(rePsiP, imPsiP) )
        return MLErrorReport(stdlink, "complex wavefunction expected");

    v0P = w0P = a1P = a2P = d0P = nil;
    if( mScalarID ) {   //  scalar potential
        scalarP = gFunctionList->Fetch(mScalarID);
        if( !scalarP )
            return MLErrorReport(stdlink, "invalid function ID for scalar potential");
        if( !scalarP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "scalar potential is not compatible");
        if( !scalarP->IsFunctionC(v0P, w0P) )
            return MLErrorReport(stdlink, "scalar potential is not C-valued");
    }
    if( mVectorID ) {   //  vector potential
        vectorP = gFunctionList->Fetch(mVectorID);
        if( !vectorP )
            return MLErrorReport(stdlink, "invalid function ID for vector potential");
        if( !vectorP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "vector potential is not compatible");
        if( !vectorP->IsFunctionR2(a1P, a2P) )
            return MLErrorReport(stdlink, "vector potential is not R2-valued");
    }
    if( mDomainID ) {   //  domain function
        domainP = gFunctionList->Fetch(mDomainID);
        if( !domainP )
            return MLErrorReport(stdlink, "invalid function ID for domain function");
        if( !domainP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "domain function is not compatible");
        if( !domainP->IsFunctionR(d0P) )
            return MLErrorReport(stdlink, "domain function is not R-valued");
    }
```

```
    domP = nil;
    if( eError == tempDomain.InitPlain2D(domP, d0P, ni, nj) ) return eError;
    tempDomain.ClearC(rePsiP, imPsiP, ni*nj);

    if( eError == tempFunction.Copy(inFunction) ) return eError;
    tempFunction.IsFunctionC(rePhiP, imPhiP);

    for( s = 0; s < inSteps; s++ ) {
        n = 1 << inFractal >> 1;    //fractal iteration
        for( i = 0; i < n; i++ ) {
            GetFractalNumber( inFractal, i, re, im );
            re *= inTimeStep;
            im *= inTimeStep;
            Kernel( rePsiP, imPsiP, rePhiP, imPhiP,
                    v0P, w0P, a1P, a2P, domP, re, im, ni, nj);

            tempP = rePsiP; rePsiP = rePhiP; rePhiP = tempP;
            tempP = imPsiP; imPsiP = imPhiP; imPhiP = tempP;
        }
        if( n == 1 ) inFunction->SwapArrayPointers(&tempFunction);

        if( eError == inFunction->UpdateWindow() ) return eError;

        MLCallYieldFunction(MLYieldFunction(stdlink), stdlink, (MLYieldParameters)0);
        if(MLAbort) {
            MLPutFunction(stdlink, "Abort", 0);
            return eError;
        }
    }

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ---------------------------------------------------------------------------
//       Kernel
// ---------------------------------------------------------------------------

void    TPauli2D::Kernel(   Float* rePsiP,
                            Float* imPsiP,
                            Float* rePhiP,
                            Float* imPhiP,
                            Float* v0P,
                            Float* w0P,
                            Float* a1P,
                            Float* a2P,
                            Int8* domP,
                            Float reZ,
                            Float imZ,
                            Int32 ni,
                            Int32 nj)
{
    Float   rePsiC, imPsiC, reEtaC, imEtaC;
    Float   rePsiR, imPsiR, rePsiL, imPsiL;
    Float   rePsiU, imPsiU, rePsiD, imPsiD;
    Float   reT, imT;
    Float   v0C, w0C;
    Float   a1C, a1R, a1L, a1U, a1D;
    Float   a2C, a2R, a2L, a2U, a2D;
    Float   chh, ceh, deh, cee, e;
    Float   one = 1.0, two = 2.0, four = 4.0;
    Int32   i, mode = 0;


    chh = one / (two * mMass * mUnits * mUnits);
    ceh = mCharge / (two * mMass * mUnits);
    deh = ceh / two;
    cee = mCharge * mCharge / (two * mMass);
    e = mCharge;

    if( mScalarID ) mode |= kScalar;
```

```
    if( mVectorID ) mode |= kVector;

    for( i = 0; i < ni*nj; i++ ) {
        if( *domP++ ) {
            rePsiR = *(rePsiP + 1);
            imPsiR = *(imPsiP + 1);
            rePsiL = *(rePsiP +-1);
            imPsiL = *(imPsiP +-1);
            rePsiU = *(rePsiP + ni);
            imPsiU = *(imPsiP + ni);
            rePsiD = *(rePsiP +-ni);
            imPsiD = *(imPsiP +-ni);
            rePsiC = *rePsiP++;
            imPsiC = *imPsiP++;
            reT = rePsiR + rePsiL + rePsiU + rePsiD;
            imT = imPsiR + imPsiL + imPsiU + imPsiD;
            reEtaC = chh * (four * rePsiC - reT);
            imEtaC = chh * (four * imPsiC - imT);
            if( mode & kVector ) {
                a1R = *(a1P + 1);
                a2R = *(a2P + 1);
                a1L = *(a1P +-1);
                a2L = *(a2P +-1);
                a1U = *(a1P + ni);
                a2U = *(a2P + ni);
                a1D = *(a1P +-ni);
                a2D = *(a2P +-ni);
                a1C = *a1P++;
                a2C = *a2P++;
                reEtaC -= ceh * a1C * (imPsiR - imPsiL);
                imEtaC += ceh * a1C * (rePsiR - rePsiL);
                reEtaC -= ceh * a2C * (imPsiU - imPsiD);
                imEtaC += ceh * a2C * (rePsiU - rePsiD);
                reT = cee * (a1C * a1C + a2C * a2C);
                reEtaC += reT * rePsiC;
                imEtaC += reT * imPsiC;
                imT = deh * (a1R - a1L + a2U - a2D);
                reEtaC -= imT * imPsiC;
                imEtaC += imT * rePsiC;
                reT = deh * (a1U - a1D - a2R + a2L);
                reEtaC += reT * rePsiC;
                imEtaC += reT * imPsiC;
            }
            if( mode & kScalar ) {
                v0C = *v0P++;
                w0C = *w0P++;
                reEtaC += e * v0C * rePsiC;
                imEtaC += e * v0C * imPsiC;
                reEtaC -= e * w0C * imPsiC;
                imEtaC += e * w0C * rePsiC;
            }
            *rePhiP++ = rePsiC + imZ * reEtaC + reZ * imEtaC;
            *imPhiP++ = imPsiC - reZ * reEtaC + imZ * imEtaC;
        }
        else {
            rePsiP++;
            imPsiP++;
            if( mode & kScalar ) { v0P++; w0P++; }
            rePhiP++;
            imPhiP++;
            if( mode & kVector ) { a1P++; a2P++; }
        }
    }

}
```

└

A.10 TPauli3D Class

A.10.1 TPauli3D.h

```
// ============================================================================
//  TPauli3D.h          (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TPauli3D
#define _H_TPauli3D
#pragma once

#include "TypeDefinition.h"
#include "TOperator.h"


class   TPauli3D : public TOperator
{
    public:
                TPauli3D(   Int32 inScalarID,
                            Int32 inVectorID,
                            Int32 inDomainID,
                            Float inMass,
                            Float inCharge,
                            Float inUnits );
                ~TPauli3D( void );
        Int32   TimeEvolution(  TFunction* inFunction,
                                Float inTimeStep,
                                Int32 inFractal,
                                Int32 inSteps );
        Int32   PutInfo( void );

    private:
        void    Kernel( Float* rePsi1P,
                        Float* imPsi1P,
                        Float* rePsi2P,
                        Float* imPsi2P,
                        Float* rePhi1P,
                        Float* imPhi1P,
                        Float* rePhi2P,
                        Float* imPhi2P,
                        Float* v0P,
                        Float* w0P,
                        Float* a1P,
                        Float* a2P,
                        Float* a3P,
                        Int8* domP,
                        Float reZ,
                        Float imZ,
                        Int32 ni,
                        Int32 nj,
                        Int32 nk );

        Int32   mScalarID;
        Int32   mVectorID;
        Int32   mDomainID;
        Float   mCharge;
        Float   mMass;
        Float   mUnits;

};

#endif
```

A.10.2 TPauli3D.cp

```
// ============================================================================
//  TPauli3D.cp         (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Class for Pauli operators

#include "MathLinkUtilities.h"
#include "TPauli3D.h"
#include "TDomain.h"



// ----------------------------------------------------------------------------
//       TPauli3D
// ----------------------------------------------------------------------------
//  Constructor

TPauli3D::TPauli3D( Int32 inScalarID,
                    Int32 inVectorID,
                    Int32 inDomainID,
                    Float inMass,
                    Float inCharge,
                    Float inUnits )
{
    mScalarID = inScalarID;
    mVectorID = inVectorID;
    mDomainID = inDomainID;
    mMass = inMass;
    mCharge = inCharge;
    mUnits = inUnits;
}



// ----------------------------------------------------------------------------
//       ~TPauli3D
// ----------------------------------------------------------------------------
//  Destructor

TPauli3D::~TPauli3D( void )
{

}



// ----------------------------------------------------------------------------
//       PutInfo
// ----------------------------------------------------------------------------
//  PutInfo

Int32   TPauli3D::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 7);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Type");
    MLPutSymbol(stdlink, "Pauli3D");

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "ScalarPotential");
    if( !mScalarID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mScalarID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "VectorPotential");
    if( !mVectorID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mVectorID);
    }
```

```
    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Domain");
    if( !mDomainID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mDomainID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mass");
    MLPutDouble(stdlink, mMass);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Charge");
    MLPutDouble(stdlink, mCharge);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Units");
    MLPutDouble(stdlink, mUnits);

    return eOK;
}



// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  TimeEvolution

Int32   TPauli3D::TimeEvolution(    TFunction* inFunction,
                                    Float inTimeStep,
                                    Int32 inFractal,
                                    Int32 inSteps )
{
    Int32   ni, nj, nk, i, n, s;
    Float   *rePsi1P, *imPsi1P, *rePsi2P, *imPsi2P, *rePhi1P, *imPhi1P, *rePhi2P, *imPhi2P;
    Float   *v0P, *w0P, *a1P, *a2P, *a3P, *d0P, *tempP, re, im;
    TFunction   *scalarP, *vectorP, *domainP, tempFunction;
    TDomain tempDomain;
    Int8    *domP;


    if( inFractal < 0 || inFractal >= 16 )
        return MLErrorReport(stdlink, "fractal order is out of range");

    ni = nj = nk = 0;
    if( !inFunction->IsFunction3D(ni, nj, nk) )
        return MLErrorReport(stdlink, "three-dimensional wavefunction expected");
    if( !inFunction->IsFunctionC2(rePsi1P, imPsi1P, rePsi2P, imPsi2P) )
        return MLErrorReport(stdlink, "spinor wavefunction expected");

    v0P = w0P = a1P = a2P = a3P = d0P = nil;
    if( mScalarID ) {   //  scalar potential
        scalarP = gFunctionList->Fetch(mScalarID);
        if( !scalarP )
            return MLErrorReport(stdlink, "invalid function ID for scalar potential");
        if( !scalarP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "scalar potential is not compatible");
        if( !scalarP->IsFunctionC(v0P, w0P) )
            return MLErrorReport(stdlink, "scalar potential is not C-valued");
    }
    if( mVectorID ) {   //  vector potential
        vectorP = gFunctionList->Fetch(mVectorID);
        if( !vectorP )
            return MLErrorReport(stdlink, "invalid function ID for vector potential");
        if( !vectorP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "vector potential is not compatible");
        if( !vectorP->IsFunctionR3(a1P, a2P, a3P) )
            return MLErrorReport(stdlink, "vector potential is not R3-valued");
    }
    if( mDomainID ) {   //  domain function
        domainP = gFunctionList->Fetch(mDomainID);
        if( !domainP )
```

```
            return MLErrorReport(stdlink, "invalid function ID for domain function");
        if( !domainP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "domain function is not compatible");
        if( !domainP->IsFunctionR(d0P) )
            return MLErrorReport(stdlink, "domain function is not R-valued");
    }

    domP = nil;
    if( eError == tempDomain.InitPlain3D(domP, d0P, ni, nj, nk) ) return eError;
    tempDomain.ClearC2(rePsi1P, imPsi1P, rePsi2P, imPsi2P, ni*nj*nk);

    if( eError == tempFunction.Copy(inFunction) ) return eError;
    tempFunction.IsFunctionC2(rePhi1P, imPhi1P, rePhi2P, imPhi2P);

    for( s = 0; s < inSteps; s++ ) {
        n = 1 << inFractal >> 1;    //fractal iteration
        for( i = 0; i < n; i++ ) {
            GetFractalNumber( inFractal, i, re, im );
            re *= inTimeStep;
            im *= inTimeStep;
            Kernel( rePsi1P,imPsi1P, rePsi2P, imPsi2P, rePhi1P, imPhi1P, rePhi2P, imPhi2P,
                    v0P, w0P, a1P, a2P, a3P, domP, re, im, ni, nj, nk );

            tempP = rePsi1P; rePsi1P = rePhi1P; rePhi1P = tempP;
            tempP = imPsi1P; imPsi1P = imPhi1P; imPhi1P = tempP;
            tempP = rePsi2P; rePsi2P = rePhi2P; rePhi2P = tempP;
            tempP = imPsi2P; imPsi2P = imPhi2P; imPhi2P = tempP;
        }
        if( n == 1 ) inFunction->SwapArrayPointers(&tempFunction);

        if( eError == inFunction->UpdateWindow() ) return eError;

        MLCallYieldFunction(MLYieldFunction(stdlink), stdlink, (MLYieldParameters)0);
        if(MLAbort) {
            MLPutFunction(stdlink, "Abort", 0);
            return eError;
        }
    }

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ---------------------------------------------------------------------------
//       Kernel
// ---------------------------------------------------------------------------

void    TPauli3D::Kernel(   Float* rePsi1P,
                            Float* imPsi1P,
                            Float* rePsi2P,
                            Float* imPsi2P,
                            Float* rePhi1P,
                            Float* imPhi1P,
                            Float* rePhi2P,
                            Float* imPhi2P,
                            Float* v0P,
                            Float* w0P,
                            Float* a1P,
                            Float* a2P,
                            Float* a3P,
                            Int8* domP,
                            Float reZ,
                            Float imZ,
                            Int32 ni,
                            Int32 nj,
                            Int32 nk )
{
    Float   rePsi1C, imPsi1C, reEta1C, imEta1C;
    Float   rePsi1R, imPsi1R, rePsi1L, imPsi1L;
    Float   rePsi1U, imPsi1U, rePsi1D, imPsi1D;
    Float   rePsi1F, imPsi1F, rePsi1B, imPsi1B;
    Float   rePsi2C, imPsi2C, reEta2C, imEta2C;
```

```
Float   rePsi2R, imPsi2R, rePsi2L, imPsi2L;
Float   rePsi2U, imPsi2U, rePsi2D, imPsi2D;
Float   rePsi2F, imPsi2F, rePsi2B, imPsi2B;
Float   reT, imT, reS, imS;
Float   v0C, w0C;
Float   a1C, a1R, a1L, a1U, a1D, a1F, a1B;
Float   a2C, a2R, a2L, a2U, a2D, a2F, a2B;
Float   a3C, a3R, a3L, a3U, a3D, a3F, a3B;
Float   chh, ceh, deh, cee, e;
Float   one = 1.0, two = 2.0, six = 6.0;
Int32   i, mode = 0;


chh = one / (two * mMass * mUnits * mUnits);
ceh = mCharge / (two * mMass * mUnits);
deh = ceh / two;
cee = mCharge * mCharge / (two * mMass);
e = mCharge;

if( mScalarID ) mode |= kScalar;
if( mVectorID ) mode |= kVector;

for( i = 0; i < ni*nj*nk; i++ ) {
    if( *domP++ ) {
        rePsi1R = *(rePsi1P + 1);
        imPsi1R = *(imPsi1P + 1);
        rePsi1L = *(rePsi1P +-1);
        imPsi1L = *(imPsi1P +-1);
        rePsi1U = *(rePsi1P + ni);
        imPsi1U = *(imPsi1P + ni);
        rePsi1D = *(rePsi1P +-ni);
        imPsi1D = *(imPsi1P +-ni);
        rePsi1F = *(rePsi1P + ni*nj);
        imPsi1F = *(imPsi1P + ni*nj);
        rePsi1B = *(rePsi1P +-ni*nj);
        imPsi1B = *(imPsi1P +-ni*nj);
        rePsi1C = *rePsi1P++;
        imPsi1C = *imPsi1P++;
        reT = rePsi1R + rePsi1L + rePsi1U + rePsi1D + rePsi1F + rePsi1B;
        imT = imPsi1R + imPsi1L + imPsi1U + imPsi1D + imPsi1F + imPsi1B;
        reEta1C = chh * (six * rePsi1C - reT);
        imEta1C = chh * (six * imPsi1C - imT);
        if( mode & kVector ) {
            a1C = *a1P;
            a2C = *a2P;
            a3C = *a3P;
            reEta1C -= ceh * a1C * (imPsi1R - imPsi1L);
            imEta1C += ceh * a1C * (rePsi1R - rePsi1L);
            reEta1C -= ceh * a2C * (imPsi1U - imPsi1D);
            imEta1C += ceh * a2C * (rePsi1U - rePsi1D);
            reEta1C -= ceh * a3C * (imPsi1F - imPsi1B);
            imEta1C += ceh * a3C * (rePsi1F - rePsi1B);
        }
        rePsi2R = *(rePsi2P + 1);
        imPsi2R = *(imPsi2P + 1);
        rePsi2L = *(rePsi2P +-1);
        imPsi2L = *(imPsi2P +-1);
        rePsi2U = *(rePsi2P + ni);
        imPsi2U = *(imPsi2P + ni);
        rePsi2D = *(rePsi2P +-ni);
        imPsi2D = *(imPsi2P +-ni);
        rePsi2F = *(rePsi2P + ni*nj);
        imPsi2F = *(imPsi2P + ni*nj);
        rePsi2B = *(rePsi2P +-ni*nj);
        imPsi2B = *(imPsi2P +-ni*nj);
        rePsi2C = *rePsi2P++;
        imPsi2C = *imPsi2P++;
        reT = rePsi2R + rePsi2L + rePsi2U + rePsi2D + rePsi2F + rePsi2B;
        imT = imPsi2R + imPsi2L + imPsi2U + imPsi2D + imPsi2F + imPsi2B;
        reEta2C = chh * (six * rePsi2C - reT);
        imEta2C = chh * (six * imPsi2C - imT);
        if( mode & kVector ) {
            a1C = *a1P;
            a2C = *a2P;
```

```
            a3C = *a3P;
            reEta2C -= ceh * a1C * (imPsi2R - imPsi2L);
            imEta2C += ceh * a1C * (rePsi2R - rePsi2L);
            reEta2C -= ceh * a2C * (imPsi2U - imPsi2D);
            imEta2C += ceh * a2C * (rePsi2U - rePsi2D);
            reEta2C -= ceh * a3C * (imPsi2F - imPsi2B);
            imEta2C += ceh * a3C * (rePsi2F - rePsi2B);
        }
        if( mode & kVector ) {
            a1R = *(a1P + 1);
            a2R = *(a2P + 1);
            a3R = *(a3P + 1);
            a1L = *(a1P +-1);
            a2L = *(a2P +-1);
            a3L = *(a3P +-1);
            a1U = *(a1P + ni);
            a2U = *(a2P + ni);
            a3U = *(a3P + ni);
            a1D = *(a1P +-ni);
            a2D = *(a2P +-ni);
            a3D = *(a3P +-ni);
            a1F = *(a1P + ni*nj);
            a2F = *(a2P + ni*nj);
            a3F = *(a3P + ni*nj);
            a1B = *(a1P +-ni*nj);
            a2B = *(a2P +-ni*nj);
            a3B = *(a3P +-ni*nj);
            a1C = *a1P++;
            a2C = *a2P++;
            a3C = *a3P++;
            reT = cee * (a1C * a1C + a2C * a2C + a3C * a3C);
            reEta1C += reT * rePsi1C;
            imEta1C += reT * imPsi1C;
            reEta2C += reT * rePsi2C;
            imEta2C += reT * imPsi2C;
            imT = deh * (a1R - a1L + a2U - a2D + a3F - a3B);
            reEta1C -= imT * imPsi1C;
            imEta1C += imT * rePsi1C;
            reEta2C -= imT * imPsi2C;
            imEta2C += imT * rePsi2C;
            reT = deh * (a1U - a1D - a2R + a2L);
            reEta1C += reT * rePsi1C;
            imEta1C += reT * imPsi1C;
            reEta2C -= reT * rePsi2C;
            imEta2C -= reT * imPsi2C;
            imS = deh * (a3R - a3L - a1F + a1B);
            reEta1C += imS * imPsi2C;
            imEta1C -= imS * rePsi2C;
            reEta2C -= imS * imPsi1C;
            imEta2C += imS * rePsi1C;
            reS = deh * (a2F - a2B - a3U + a3D);
            reEta1C += reS * rePsi2C;
            imEta1C += reS * imPsi2C;
            reEta2C += reS * rePsi1C;
            imEta2C += reS * imPsi1C;
        }
        if( mode & kScalar ) {
            v0C = *v0P++;
            w0C = *w0P++;
            reEta1C += e * v0C * rePsi1C;
            imEta1C += e * v0C * imPsi1C;
            reEta2C += e * v0C * rePsi2C;
            imEta2C += e * v0C * imPsi2C;
            reEta1C -= e * w0C * imPsi1C;
            imEta1C += e * w0C * rePsi1C;
            reEta2C -= e * w0C * imPsi2C;
            imEta2C += e * w0C * rePsi2C;
        }
        *rePhi1P++ = rePsi1C + imZ * reEta1C + reZ * imEta1C;
        *imPhi1P++ = imPsi1C - reZ * reEta1C + imZ * imEta1C;
        *rePhi2P++ = rePsi2C + imZ * reEta2C + reZ * imEta2C;
        *imPhi2P++ = imPsi2C - reZ * reEta2C + imZ * imEta2C;
    }
    else {
```

```
            rePsi1P++;
            imPsi1P++;
            rePsi2P++;
            imPsi2P++;
            if( mode & kScalar ) { v0P++; w0P++; }
            rePhi1P++;
            imPhi1P++;
            rePhi2P++;
            imPhi2P++;
            if( mode & kVector ) { a1P++; a2P++; a3P++; }
        }
    }

}

└
```

A.11 TDirac2D Class

A.11.1 TDirac2D.h

```
┌
// ============================================================================
//  TDirac2D.h          (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TDirac2D
#define _H_TDirac2D
#pragma once

#include "TypeDefinition.h"
#include "TOperator.h"


class   TDirac2D : public TOperator
{
    public:
                TDirac2D(   Int32 inScalarID,
                            Int32 inVectorID,
                            Int32 inDomainID,
                            Float inMass,
                            Float inCharge,
                            Float inUnits );
                ~TDirac2D( void );
        Int32   TimeEvolution(  TFunction* inFunction,
                                Float inTimeStep,
                                Int32 inFractal,
                                Int32 inSteps );
        Int32   PutInfo( void );

    private:
        void    Kernel( Float* rePsi1P,
                        Float* imPsi1P,
                        Float* rePsi2P,
                        Float* imPsi2P,
                        Float* rePhi1P,
                        Float* imPhi1P,
                        Float* rePhi2P,
                        Float* imPhi2P,
                        Float* v0P,
                        Float* w0P,
                        Float* a1P,
                        Float* a2P,
                        Float* a3P,
                        Int8* domP,
                        Float reZ,
                        Float imZ,
                        Int32 ni,
                        Int32 nj );
```

```
        Int32   mScalarID;
        Int32   mVectorID;
        Int32   mDomainID;
        Float   mCharge;
        Float   mMass;
        Float   mUnits;

};

#endif

└
```

A.11.2 TDirac2D.cp

```
┌

// ============================================================================
//  TDirac2D.cp         (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Class for Dirac operators

#include "MathLinkUtilities.h"
#include "TDirac2D.h"
#include "TDomain.h"



// ----------------------------------------------------------------------------
//       TDirac2D
// ----------------------------------------------------------------------------
//  Constructor

TDirac2D::TDirac2D( Int32 inScalarID,
                    Int32 inVectorID,
                    Int32 inDomainID,
                    Float inMass,
                    Float inCharge,
                    Float inUnits )
{
    mScalarID = inScalarID;
    mVectorID = inVectorID;
    mDomainID = inDomainID;
    mMass = inMass;
    mCharge = inCharge;
    mUnits = inUnits;
}



// ----------------------------------------------------------------------------
//       ~TDirac2D
// ----------------------------------------------------------------------------
//  Destructor

TDirac2D::~TDirac2D( void )
{

}



// ----------------------------------------------------------------------------
//       PutInfo
// ----------------------------------------------------------------------------
//  PutInfo

Int32   TDirac2D::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 7);
    MLPutFunction(stdlink, "Rule", 2);
```

```
    MLPutSymbol(stdlink, "Type");
    MLPutSymbol(stdlink, "Dirac2D");

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "ScalarPotential");
    if( !mScalarID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mScalarID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "VectorPotential");
    if( !mVectorID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mVectorID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Domain");
    if( !mDomainID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mDomainID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mass");
    MLPutDouble(stdlink, mMass);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Charge");
    MLPutDouble(stdlink, mCharge);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Units");
    MLPutDouble(stdlink, mUnits);

    return eOK;
}



// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  TimeEvolution

Int32   TDirac2D::TimeEvolution(    TFunction* inFunction,
                                    Float inTimeStep,
                                    Int32 inFractal,
                                    Int32 inSteps )
{
    Int32   ni, nj, i, n, s;
    Float   *rePsi1P, *imPsi1P, *rePsi2P, *imPsi2P, *rePhi1P, *imPhi1P, *rePhi2P, *imPhi2P;
    Float   *v0P, *w0P, *a1P, *a2P, *a3P, *d0P, *tempP, re, im;
    TFunction   *scalarP, *vectorP, *domainP, tempFunction;
    TDomain tempDomain;
    Int8    *domP;


    if( inFractal < 0 || inFractal >= 16 )
        return MLErrorReport(stdlink, "fractal order is out of range");

    ni = nj = 0;
    if( !inFunction->IsFunction2D(ni, nj) )
        return MLErrorReport(stdlink, "two-dimensional wavefunction expected");
    if( !inFunction->IsFunctionC2(rePsi1P, imPsi1P, rePsi2P, imPsi2P) )
        return MLErrorReport(stdlink, "spinor wavefunction expected");

    v0P = w0P = a1P = a2P = a3P = d0P = nil;
    if( mScalarID ) {   //  scalar potential
        scalarP = gFunctionList->Fetch(mScalarID);
```

```
        if( !scalarP )
            return MLErrorReport(stdlink, "invalid function ID for scalar potential");
        if( !scalarP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "scalar potential is not compatible");
        if( !scalarP->IsFunctionC(v0P, w0P) )
            return MLErrorReport(stdlink, "scalar potential is not C-valued");
    }
    if( mVectorID ) {   //  vector potential
        vectorP = gFunctionList->Fetch(mVectorID);
        if( !vectorP )
            return MLErrorReport(stdlink, "invalid function ID for vector potential");
        if( !vectorP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "vector potential is not compatible");
        if( !vectorP->IsFunctionR3(a1P, a2P, a3P) )
            return MLErrorReport(stdlink, "vector potential is not R3-valued");
    }
    if( mDomainID ) {   //  domain function
        domainP = gFunctionList->Fetch(mDomainID);
        if( !domainP )
            return MLErrorReport(stdlink, "invalid function ID for domain function");
        if( !domainP->IsFunction2D(ni, nj) )
            return MLErrorReport(stdlink, "domain function is not compatible");
        if( !domainP->IsFunctionR(d0P) )
            return MLErrorReport(stdlink, "domain function is not R-valued");
    }

    domP = nil;
    if( eError == tempDomain.InitPlain2D(domP, d0P, ni, nj) ) return eError;
    tempDomain.ClearC2(rePsi1P, imPsi1P, rePsi2P, imPsi2P, ni*nj);

    if( eError == tempFunction.Copy(inFunction) ) return eError;
    tempFunction.IsFunctionC2(rePhi1P, imPhi1P, rePhi2P, imPhi2P);

    for( s = 0; s < inSteps; s++ ) {
        n = 1 << inFractal >> 1;    //fractal iteration
        for( i = 0; i < n; i++ ) {
            GetFractalNumber( inFractal, i, re, im );
            re *= inTimeStep;
            im *= inTimeStep;
            Kernel( rePsi1P, imPsi1P, rePsi2P, imPsi2P, rePhi1P, imPhi1P, rePhi2P, imPhi2P,
                    v0P, w0P, a1P, a2P, a3P, domP, re, im, ni, nj );

            tempP = rePsi1P; rePsi1P = rePhi1P; rePhi1P = tempP;
            tempP = imPsi1P; imPsi1P = imPhi1P; imPhi1P = tempP;
            tempP = rePsi2P; rePsi2P = rePhi2P; rePhi2P = tempP;
            tempP = imPsi2P; imPsi2P = imPhi2P; imPhi2P = tempP;
        }
        if( n == 1 ) inFunction->SwapArrayPointers(&tempFunction);

        if( eError == inFunction->UpdateWindow() ) return eError;

        MLCallYieldFunction(MLYieldFunction(stdlink), stdlink, (MLYieldParameters)0);
        if(MLAbort) {
            MLPutFunction(stdlink, "Abort", 0);
            return eError;
        }
    }

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ---------------------------------------------------------------------------
//       Kernel
// ---------------------------------------------------------------------------

void    TDirac2D::Kernel(   Float* rePsi1P,
                            Float* imPsi1P,
                            Float* rePsi2P,
                            Float* imPsi2P,
                            Float* rePhi1P,
                            Float* imPhi1P,
```

```
                          Float* rePhi2P,
                          Float* imPhi2P,
                          Float* v0P,
                          Float* w0P,
                          Float* a1P,
                          Float* a2P,
                          Float* a3P,
                          Int8* domP,
                          Float reZ,
                          Float imZ,
                          Int32 ni,
                          Int32 nj )
{
    Float   rePsi1C, imPsi1C, reEta1C, imEta1C;
    Float   rePsi1R, imPsi1R, rePsi1L, imPsi1L;
    Float   rePsi1U, imPsi1U, rePsi1D, imPsi1D;
    Float   rePsi2C, imPsi2C, reEta2C, imEta2C;
    Float   rePsi2R, imPsi2R, rePsi2L, imPsi2L;
    Float   rePsi2U, imPsi2U, rePsi2D, imPsi2D;
    Float   v0C, w0C, a1C, a2C, a3C;
    Float   ch, m, e;
    Float   one = 1.0, two = 2.0;
    Int32   i, mode = 0;


    ch = one / ( two * mUnits );
    m = mMass;
    e = mCharge;

    if( mScalarID ) mode |= kScalar;
    if( mVectorID ) mode |= kVector;

    for( i = 0; i < ni*nj; i++ ) {
        if( *domP++ ) {
            rePsi1R = *(rePsi1P + 1);
            imPsi1R = *(imPsi1P + 1);
            rePsi1L = *(rePsi1P +-1);
            imPsi1L = *(imPsi1P +-1);
            rePsi1U = *(rePsi1P + ni);
            imPsi1U = *(imPsi1P + ni);
            rePsi1D = *(rePsi1P +-ni);
            imPsi1D = *(imPsi1P +-ni);
            reEta2C = ch * (imPsi1R - imPsi1L + rePsi1U - rePsi1D);
            imEta2C = ch * (rePsi1L - rePsi1R + imPsi1U - imPsi1D);
            rePsi2R = *(rePsi2P + 1);
            imPsi2R = *(imPsi2P + 1);
            rePsi2L = *(rePsi2P +-1);
            imPsi2L = *(imPsi2P +-1);
            rePsi2U = *(rePsi2P + ni);
            imPsi2U = *(imPsi2P + ni);
            rePsi2D = *(rePsi2P +-ni);
            imPsi2D = *(imPsi2P +-ni);
            reEta1C = ch * (imPsi2R - imPsi2L - rePsi2U + rePsi2D);
            imEta1C = ch * (rePsi2L - rePsi2R - imPsi2U + imPsi2D);
            rePsi1C = *rePsi1P++;
            imPsi1C = *imPsi1P++;
            rePsi2C = *rePsi2P++;
            imPsi2C = *imPsi2P++;
            if( mode & kVector ) {
                a1C = *a1P++;
                a2C = *a2P++;
                a3C = *a3P++;
                reEta1C -= e * a1C * rePsi2C;
                imEta1C -= e * a1C * imPsi2C;
                reEta2C -= e * a1C * rePsi1C;
                imEta2C -= e * a1C * imPsi1C;
                reEta1C -= e * a2C * imPsi2C;
                imEta1C += e * a2C * rePsi2C;
                reEta2C += e * a2C * imPsi1C;
                imEta2C -= e * a2C * rePsi1C;
                reEta1C -= e * a3C * rePsi1C;
                imEta1C -= e * a3C * imPsi1C;
                reEta2C += e * a3C * rePsi2C;
                imEta2C += e * a3C * imPsi2C;
```

```
            }
            reEta1C += m * rePsi1C;
            imEta1C += m * imPsi1C;
            reEta2C -= m * rePsi2C;
            imEta2C -= m * imPsi2C;
            if( mode & kScalar ) {
                v0C = *v0P++;
                w0C = *w0P++;
                reEta1C += e * v0C * rePsi1C;
                imEta1C += e * v0C * imPsi1C;
                reEta2C += e * v0C * rePsi2C;
                imEta2C += e * v0C * imPsi2C;
                reEta1C -= e * w0C * imPsi1C;
                imEta1C += e * w0C * rePsi1C;
                reEta2C -= e * w0C * imPsi2C;
                imEta2C += e * w0C * rePsi2C;
            }
            *rePhi1P++ = rePsi1C + imZ * reEta1C + reZ * imEta1C;
            *imPhi1P++ = imPsi1C - reZ * reEta1C + imZ * imEta1C;
            *rePhi2P++ = rePsi2C + imZ * reEta2C + reZ * imEta2C;
            *imPhi2P++ = imPsi2C - reZ * reEta2C + imZ * imEta2C;
        }
        else {
            rePsi1P++;
            imPsi1P++;
            rePsi2P++;
            imPsi2P++;
            if( mode & kScalar ) { v0P++; w0P++; }
            rePhi1P++;
            imPhi1P++;
            rePhi2P++;
            imPhi2P++;
            if( mode & kVector ) { a1P++; a2P++; a3P++; }
        }
    }

}

└
```

A.12 TDirac3D Class

A.12.1 TDirac3D.h

```
┌

// ============================================================================
//  TDirac3D.h          (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_TDirac3D
#define _H_TDirac3D
#pragma once

#include "TypeDefinition.h"
#include "TOperator.h"


class   TDirac3D : public TOperator
{
    public:
                TDirac3D(   Int32 inScalarID,
                            Int32 inVectorID,
                            Int32 inDomainID,
                            Float inMass,
                            Float inCharge,
                            Float inUnits );
                ~TDirac3D( void );
        Int32   TimeEvolution(  TFunction* inFunction,
                                Float inTimeStep,
                                Int32 inFractal,
```

```
                            Int32 inSteps );
        Int32   PutInfo( void );

    private:
        void    Kernel( Float* rePsi1P,
                        Float* imPsi1P,
                        Float* rePsi2P,
                        Float* imPsi2P,
                        Float* rePsi3P,
                        Float* imPsi3P,
                        Float* rePsi4P,
                        Float* imPsi4P,
                        Float* rePhi1P,
                        Float* imPhi1P,
                        Float* rePhi2P,
                        Float* imPhi2P,
                        Float* rePhi3P,
                        Float* imPhi3P,
                        Float* rePhi4P,
                        Float* imPhi4P,
                        Float* v0P,
                        Float* w0P,
                        Float* a1P,
                        Float* a2P,
                        Float* a3P,
                        Float* a4P,
                        Int8* domP,
                        Float reZ,
                        Float imZ,
                        Int32 ni,
                        Int32 nj,
                        Int32 nk );

        Int32   mScalarID;
        Int32   mVectorID;
        Int32   mDomainID;
        Float   mCharge;
        Float   mMass;
        Float   mUnits;

};

#endif
```

A.12.2 TDirac3D.cp

```
// ============================================================================
//  TDirac3D.cp         (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  Class for Dirac operators

#include "MathLinkUtilities.h"
#include "TDirac3D.h"
#include "TDomain.h"



// ----------------------------------------------------------------------------
//       TDirac3D
// ----------------------------------------------------------------------------
//  Constructor

TDirac3D::TDirac3D( Int32 inScalarID,
                    Int32 inVectorID,
                    Int32 inDomainID,
                    Float inMass,
                    Float inCharge,
```

```
                    Float inUnits )
{
    mScalarID = inScalarID;
    mVectorID = inVectorID;
    mDomainID = inDomainID;
    mMass = inMass;
    mCharge = inCharge;
    mUnits = inUnits;
}



// ---------------------------------------------------------------------------
//       ~TDirac3D
// ---------------------------------------------------------------------------
//  Destructor

TDirac3D::~TDirac3D( void )
{

}



// ---------------------------------------------------------------------------
//       PutInfo
// ---------------------------------------------------------------------------
//  PutInfo

Int32   TDirac3D::PutInfo( void )
{
    MLPutFunction(stdlink, "List", 7);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Type");
    MLPutSymbol(stdlink, "Dirac3D");

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "ScalarPotential");
    if( !mScalarID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mScalarID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "VectorPotential");
    if( !mVectorID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mVectorID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Domain");
    if( !mDomainID ) MLPutSymbol(stdlink, "None");
    else {
        MLPutFunction(stdlink, "FunctionObject", 1);
        MLPutInteger(stdlink, mDomainID);
    }

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Mass");
    MLPutDouble(stdlink, mMass);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Charge");
    MLPutDouble(stdlink, mCharge);

    MLPutFunction(stdlink, "Rule", 2);
    MLPutSymbol(stdlink, "Units");
    MLPutDouble(stdlink, mUnits);

    return eOK;
}
```

```
// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  TimeEvolution

Int32   TDirac3D::TimeEvolution(    TFunction* inFunction,
                                    Float inTimeStep,
                                    Int32 inFractal,
                                    Int32 inSteps )
{
    Int32   ni, nj, nk, i, n, s;
    Float   *rePsi1P, *imPsi1P, *rePsi2P, *imPsi2P, *rePsi3P, *imPsi3P, *rePsi4P, *imPsi4P;
    Float   *rePhi1P, *imPhi1P, *rePhi2P, *imPhi2P, *rePhi3P, *imPhi3P, *rePhi4P, *imPhi4P;
    Float   *v0P, *w0P, *a1P, *a2P, *a3P, *a4P, *d0P, *tempP, re, im;
    TFunction   *scalarP, *vectorP, *domainP, tempFunction;
    TDomain tempDomain;
    Int8    *domP;


    if( inFractal < 0 || inFractal >= 16 )
        return MLErrorReport(stdlink, "fractal order is out of range");

    ni = nj = nk = 0;
    if( !inFunction->IsFunction3D(ni, nj, nk) )
        return MLErrorReport(stdlink, "three-dimensional wavefunction expected");
    if( !inFunction->IsFunctionC4(  rePsi1P, imPsi1P, rePsi2P, imPsi2P,
                                    rePsi3P, imPsi3P, rePsi4P, imPsi4P) )
        return MLErrorReport(stdlink, "bispinor wavefunction expected");

    v0P = w0P = a1P = a2P = a3P = a4P = d0P = nil;
    if( mScalarID ) {   //  scalar potential
        scalarP = gFunctionList->Fetch(mScalarID);
        if( !scalarP )
            return MLErrorReport(stdlink, "invalid function ID for scalar potential");
        if( !scalarP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "scalar potential is not compatible");
        if( !scalarP->IsFunctionC(v0P, w0P) )
            return MLErrorReport(stdlink, "scalar potential is not C-valued");
    }
    if( mVectorID ) {   //  vector potential
        vectorP = gFunctionList->Fetch(mVectorID);
        if( !vectorP )
            return MLErrorReport(stdlink, "invalid function ID for vector potential");
        if( !vectorP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "vector potential is not compatible");
        if( !vectorP->IsFunctionR4(a1P, a2P, a3P, a4P) )
            return MLErrorReport(stdlink, "vector potential is not R4-valued");
    }
    if( mDomainID ) {   //  domain function
        domainP = gFunctionList->Fetch(mDomainID);
        if( !domainP )
            return MLErrorReport(stdlink, "invalid function ID for domain function");
        if( !domainP->IsFunction3D(ni, nj, nk) )
            return MLErrorReport(stdlink, "domain function is not compatible");
        if( !domainP->IsFunctionR(d0P) )
            return MLErrorReport(stdlink, "domain function is not R-valued");
    }

    domP = nil;
    if( eError == tempDomain.InitPlain3D(domP, d0P, ni, nj, nk) ) return eError;
    tempDomain.ClearC4( rePsi1P, imPsi1P, rePsi2P, imPsi2P,
                        rePsi3P, imPsi3P, rePsi4P, imPsi4P, ni*nj*nk);

    if( eError == tempFunction.Copy(inFunction) ) return eError;
    tempFunction.IsFunctionC4(  rePhi1P, imPhi1P, rePhi2P, imPhi2P,
                                rePhi3P, imPhi3P, rePhi4P, imPhi4P);

    for( s = 0; s < inSteps; s++ ) {
        n = 1 << inFractal >> 1;    //fractal iteration
        for( i = 0; i < n; i++ ) {
            GetFractalNumber( inFractal, i, re, im );
```

```
            re *= inTimeStep;
            im *= inTimeStep;
            Kernel( rePsi1P, imPsi1P, rePsi2P, imPsi2P, rePsi3P, imPsi3P, rePsi4P, imPsi4P,
                    rePhi1P, imPhi1P, rePhi2P, imPhi2P, rePhi3P, imPhi3P, rePhi4P, imPhi4P,
                    v0P, w0P, a1P, a2P, a3P, a4P, domP, re, im, ni, nj, nk );

            tempP = rePsi1P; rePsi1P = rePhi1P; rePhi1P = tempP;
            tempP = imPsi1P; imPsi1P = imPhi1P; imPhi1P = tempP;
            tempP = rePsi2P; rePsi2P = rePhi2P; rePhi2P = tempP;
            tempP = imPsi2P; imPsi2P = imPhi2P; imPhi2P = tempP;
            tempP = rePsi3P; rePsi3P = rePhi3P; rePhi3P = tempP;
            tempP = imPsi3P; imPsi3P = imPhi3P; imPhi3P = tempP;
            tempP = rePsi4P; rePsi4P = rePhi4P; rePhi4P = tempP;
            tempP = imPsi4P; imPsi4P = imPhi4P; imPhi4P = tempP;

        }
        if( n == 1 ) inFunction->SwapArrayPointers(&tempFunction);

        if( eError == inFunction->UpdateWindow() ) return eError;

        MLCallYieldFunction(MLYieldFunction(stdlink), stdlink, (MLYieldParameters)0);
        if(MLAbort) {
            MLPutFunction(stdlink, "Abort", 0);
            return eError;
        }
    }

    MLPutSymbol(stdlink, "Null");
    return eOK;
}



// ---------------------------------------------------------------------------
//       Kernel
// ---------------------------------------------------------------------------

void    TDirac3D::Kernel(   Float* rePsi1P,
                            Float* imPsi1P,
                            Float* rePsi2P,
                            Float* imPsi2P,
                            Float* rePsi3P,
                            Float* imPsi3P,
                            Float* rePsi4P,
                            Float* imPsi4P,
                            Float* rePhi1P,
                            Float* imPhi1P,
                            Float* rePhi2P,
                            Float* imPhi2P,
                            Float* rePhi3P,
                            Float* imPhi3P,
                            Float* rePhi4P,
                            Float* imPhi4P,
                            Float* v0P,
                            Float* w0P,
                            Float* a1P,
                            Float* a2P,
                            Float* a3P,
                            Float* a4P,
                            Int8* domP,
                            Float reZ,
                            Float imZ,
                            Int32 ni,
                            Int32 nj,
                            Int32 nk )
{
    Float   rePsi1C, imPsi1C, reEta1C, imEta1C;
    Float   rePsi1R, imPsi1R, rePsi1L, imPsi1L;
    Float   rePsi1U, imPsi1U, rePsi1D, imPsi1D;
    Float   rePsi1F, imPsi1F, rePsi1B, imPsi1B;
    Float   rePsi2C, imPsi2C, reEta2C, imEta2C;
    Float   rePsi2R, imPsi2R, rePsi2L, imPsi2L;
    Float   rePsi2U, imPsi2U, rePsi2D, imPsi2D;
    Float   rePsi2F, imPsi2F, rePsi2B, imPsi2B;
```

```
Float   rePsi3C, imPsi3C, reEta3C, imEta3C;
Float   rePsi3R, imPsi3R, rePsi3L, imPsi3L;
Float   rePsi3U, imPsi3U, rePsi3D, imPsi3D;
Float   rePsi3F, imPsi3F, rePsi3B, imPsi3B;
Float   rePsi4C, imPsi4C, reEta4C, imEta4C;
Float   rePsi4R, imPsi4R, rePsi4L, imPsi4L;
Float   rePsi4U, imPsi4U, rePsi4D, imPsi4D;
Float   rePsi4F, imPsi4F, rePsi4B, imPsi4B;
Float   v0C, w0C, a1C, a2C, a3C, a4C;
Float   ch, m, e;
Float   one = 1.0, two = 2.0;
Int32   i, mode = 0;


ch = one / ( two * mUnits );
m = mMass;
e = mCharge;

if( mScalarID ) mode |= kScalar;
if( mVectorID ) mode |= kVector;

for( i = 0; i < ni*nj*nk; i++ ) {
    if( *domP++ ) {
        rePsi1R = *(rePsi1P + 1);
        imPsi1R = *(imPsi1P + 1);
        rePsi1L = *(rePsi1P +-1);
        imPsi1L = *(imPsi1P +-1);
        rePsi1U = *(rePsi1P + ni);
        imPsi1U = *(imPsi1P + ni);
        rePsi1D = *(rePsi1P +-ni);
        imPsi1D = *(imPsi1P +-ni);
        rePsi1F = *(rePsi1P + ni*nj);
        imPsi1F = *(imPsi1P + ni*nj);
        rePsi1B = *(rePsi1P +-ni*nj);
        imPsi1B = *(imPsi1P +-ni*nj);
        rePsi2R = *(rePsi2P + 1);
        imPsi2R = *(imPsi2P + 1);
        rePsi2L = *(rePsi2P +-1);
        imPsi2L = *(imPsi2P +-1);
        rePsi2U = *(rePsi2P + ni);
        imPsi2U = *(imPsi2P + ni);
        rePsi2D = *(rePsi2P +-ni);
        imPsi2D = *(imPsi2P +-ni);
        rePsi2F = *(rePsi2P + ni*nj);
        imPsi2F = *(imPsi2P + ni*nj);
        rePsi2B = *(rePsi2P +-ni*nj);
        imPsi2B = *(imPsi2P +-ni*nj);
        reEta3C = ch * (imPsi2R - imPsi2L - rePsi2U + rePsi2D + imPsi1F - imPsi1B);
        imEta3C = ch * (rePsi2L - rePsi2R - imPsi2U + imPsi2D + rePsi1B - rePsi1F);
        reEta4C = ch * (imPsi1R - imPsi1L + rePsi1U - rePsi1D + imPsi2B - imPsi2F);
        imEta4C = ch * (rePsi1L - rePsi1R + imPsi1U - imPsi1D + rePsi2F - rePsi2B);
        rePsi3R = *(rePsi3P + 1);
        imPsi3R = *(imPsi3P + 1);
        rePsi3L = *(rePsi3P +-1);
        imPsi3L = *(imPsi3P +-1);
        rePsi3U = *(rePsi3P + ni);
        imPsi3U = *(imPsi3P + ni);
        rePsi3D = *(rePsi3P +-ni);
        imPsi3D = *(imPsi3P +-ni);
        rePsi3F = *(rePsi3P + ni*nj);
        imPsi3F = *(imPsi3P + ni*nj);
        rePsi3B = *(rePsi3P +-ni*nj);
        imPsi3B = *(imPsi3P +-ni*nj);
        rePsi4R = *(rePsi4P + 1);
        imPsi4R = *(imPsi4P + 1);
        rePsi4L = *(rePsi4P +-1);
        imPsi4L = *(imPsi4P +-1);
        rePsi4U = *(rePsi4P + ni);
        imPsi4U = *(imPsi4P + ni);
        rePsi4D = *(rePsi4P +-ni);
        imPsi4D = *(imPsi4P +-ni);
        rePsi4F = *(rePsi4P + ni*nj);
        imPsi4F = *(imPsi4P + ni*nj);
        rePsi4B = *(rePsi4P +-ni*nj);
```

```
imPsi4B = *(imPsi4P +-ni*nj);
reEta1C = ch * (imPsi4R - imPsi4L - rePsi4U + rePsi4D + imPsi3F - imPsi3B);
imEta1C = ch * (rePsi4L - rePsi4R - imPsi4U + imPsi4D + rePsi3B - rePsi3F);
reEta2C = ch * (imPsi3R - imPsi3L + rePsi3U - rePsi3D + imPsi4B - imPsi4F);
imEta2C = ch * (rePsi3L - rePsi3R + imPsi3U - imPsi3D + rePsi4F - rePsi4B);
rePsi1C = *rePsi1P++;
imPsi1C = *imPsi1P++;
rePsi2C = *rePsi2P++;
imPsi2C = *imPsi2P++;
rePsi3C = *rePsi3P++;
imPsi3C = *imPsi3P++;
rePsi4C = *rePsi4P++;
imPsi4C = *imPsi4P++;
if( mode & kVector ) {
    a1C = *a1P++;
    a2C = *a2P++;
    a3C = *a3P++;
    a4C = *a4P++;
    reEta1C -= e * a1C * rePsi4C;
    imEta1C -= e * a1C * imPsi4C;
    reEta2C -= e * a1C * rePsi3C;
    imEta2C -= e * a1C * imPsi3C;
    reEta3C -= e * a1C * rePsi2C;
    imEta3C -= e * a1C * imPsi2C;
    reEta4C -= e * a1C * rePsi1C;
    imEta4C -= e * a1C * imPsi1C;
    reEta1C -= e * a2C * imPsi4C;
    imEta1C += e * a2C * rePsi4C;
    reEta2C += e * a2C * imPsi3C;
    imEta2C -= e * a2C * rePsi3C;
    reEta3C -= e * a2C * imPsi2C;
    imEta3C += e * a2C * rePsi2C;
    reEta4C += e * a2C * imPsi1C;
    imEta4C -= e * a2C * rePsi1C;
    reEta1C -= e * a3C * rePsi3C;
    imEta1C -= e * a3C * imPsi3C;
    reEta2C += e * a3C * rePsi4C;
    imEta2C += e * a3C * imPsi4C;
    reEta3C -= e * a3C * rePsi1C;
    imEta3C -= e * a3C * imPsi1C;
    reEta4C += e * a3C * rePsi2C;
    imEta4C += e * a3C * imPsi2C;
    reEta1C -= e * a4C * rePsi1C;
    imEta1C -= e * a4C * imPsi1C;
    reEta2C -= e * a4C * rePsi2C;
    imEta2C -= e * a4C * imPsi2C;
    reEta3C += e * a4C * rePsi3C;
    imEta3C += e * a4C * imPsi3C;
    reEta4C += e * a4C * rePsi4C;
    imEta4C += e * a4C * imPsi4C;
}
reEta1C += m * rePsi1C;
imEta1C += m * imPsi1C;
reEta2C += m * rePsi2C;
imEta2C += m * imPsi2C;
reEta3C -= m * rePsi3C;
imEta3C -= m * imPsi3C;
reEta4C -= m * rePsi4C;
imEta4C -= m * imPsi4C;
if( mode & kScalar ) {
    v0C = *v0P++;
    w0C = *w0P++;
    reEta1C += e * v0C * rePsi1C;
    imEta1C += e * v0C * imPsi1C;
    reEta2C += e * v0C * rePsi2C;
    imEta2C += e * v0C * imPsi2C;
    reEta3C += e * v0C * rePsi3C;
    imEta3C += e * v0C * imPsi3C;
    reEta4C += e * v0C * rePsi4C;
    imEta4C += e * v0C * imPsi4C;
    reEta1C -= e * w0C * imPsi1C;
    imEta1C += e * w0C * rePsi1C;
    reEta2C -= e * w0C * imPsi2C;
    imEta2C += e * w0C * rePsi2C;
```

```
                reEta3C -= e * w0C * imPsi3C;
                imEta3C += e * w0C * rePsi3C;
                reEta4C -= e * w0C * imPsi4C;
                imEta4C += e * w0C * rePsi4C;
            }
            *rePhi1P++ = rePsi1C + imZ * reEta1C + reZ * imEta1C;
            *imPhi1P++ = imPsi1C - reZ * reEta1C + imZ * imEta1C;
            *rePhi2P++ = rePsi2C + imZ * reEta2C + reZ * imEta2C;
            *imPhi2P++ = imPsi2C - reZ * reEta2C + imZ * imEta2C;
            *rePhi3P++ = rePsi3C + imZ * reEta3C + reZ * imEta3C;
            *imPhi3P++ = imPsi3C - reZ * reEta3C + imZ * imEta3C;
            *rePhi4P++ = rePsi4C + imZ * reEta4C + reZ * imEta4C;
            *imPhi4P++ = imPsi4C - reZ * reEta4C + imZ * imEta4C;
        }
        else {
            rePsi1P++;
            imPsi1P++;
            rePsi2P++;
            imPsi2P++;
            rePsi3P++;
            imPsi3P++;
            rePsi4P++;
            imPsi4P++;
            if( mode & kScalar ) { v0P++; w0P++; }
            rePhi1P++;
            imPhi1P++;
            rePhi2P++;
            imPhi2P++;
            rePhi3P++;
            imPhi3P++;
            rePhi4P++;
            imPhi4P++;
            if( mode & kVector ) { a1P++; a2P++; a3P++; a4P++; }
        }
    }

}

└
```

Anhang B

C-Source-Code

B.1 MathLinkUtilities

B.1.1 MathLinkUtilities.h

```
// ==========================================================================
//  MathLinkUtilities.h (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================

#ifndef _H_MathLinkUtilities
#define _H_MathLinkUtilities
#pragma once

#include "mathlink.h"
#include"TypeDefinition.h"

//  Prototypes

Int32   MLErrorReport(  MLINK inLink,
                        Int8* inMessage );
Int32   MLGetRealArray2(    MLINK inLink,
                            Float* &outArrayP,
                            Int32* &outCountP,
                            Int32 &outDepth );
Int32   MLReadList( MLINK inLink,
                    Float* inArrayP,
                    Int32* inCountP,
                    Int32 inDepth,
                    Int32 &ioIteration );
Int32   MLCheckMemoryReserve( MLINK inLink );
Int32   MLGetFunctionObject(    MLINK inLink,
                                Int32 &outID );
Int32   MLGetOperatorObject(    MLINK inLink,
                                Int32 &outID );
Int32   MLGetWindowObject(  MLINK inLink,
                            Int32 &outID );
Int32   MLEventLoop( void );

//  Errorcodes

enum {
    eError = 1, eOK = 0
};

#endif
```

B.1.2 MathLinkUtilities.c

```
// ==========================================================================
//  MathLinkUtilities.c (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ==========================================================================
//
//  MathLink utility functions

#include "MathLinkUtilities.h"
#include "TWindow.h"
#include <string.h>
#include <stdio.h>



// --------------------------------------------------------------------------
//       MLErrorReport
// --------------------------------------------------------------------------
//  Send error message

Int32   MLErrorReport(  MLINK inLink,
                        Int8* inMessage )
{
    char    errMsg[256];


    sprintf(errMsg, "%s\"%.192s\"%s","Message[QuantumKernel::err,",inMessage,"]");
    MLClearError(inLink);
    MLNewPacket(inLink);
    MLEvaluate(inLink, errMsg);
    while( MLNextPacket(inLink) != RETURNPKT ) MLNewPacket(inLink);
    MLNewPacket(inLink);
    MLPutSymbol(inLink, "$Failed");

    return eError;      //we have an error!
}



// --------------------------------------------------------------------------
//       MLGetRealArray2
// --------------------------------------------------------------------------
//  Substitution for MLGetRealArray

Int32   MLGetRealArray2(    MLINK inLink,
                            Float* &outArrayP,
                            Int32* &outCountP,
                            Int32 &outDepth )
{
    Int32   i, len, size, ioIteration;
    const Int32 kIterationLimit = 16;


    outCountP = new Int32[kIterationLimit];
    if( outCountP == nil )
        return MLErrorReport(inLink, "out of memory");

    outDepth = 0;
    while( MLGetType(inLink) == MLTKFUNC ) {
        MLCheckFunction(inLink, "List", &len);
        if( MLError(inLink) || len == 0 )
            return MLErrorReport(inLink, "out of sequence");
        outCountP[outDepth] = len;
        outDepth++;
        if( outDepth >= kIterationLimit )
            return MLErrorReport(inLink, "iteration limit reached");
    }
    if( outDepth == 0 ) return MLErrorReport(inLink, "out of sequence");

    size = 1;
    for( i = 0; i < outDepth; i++ ) size *= outCountP[i];
    outArrayP = new Float[size];
    if( outArrayP == nil )
```

```
        return MLErrorReport(inLink, "out of memory");
    if( eError == MLCheckMemoryReserve(inLink) ) return eError;

    if( eError == MLReadList(inLink, outArrayP, outCountP, outDepth, ioIteration = 0) )
        return eError;

    return eOK;
}



// ---------------------------------------------------------------------------
//       MLReadList
// ---------------------------------------------------------------------------
//  Iterator for MLGetRealArray2

Int32   MLReadList( MLINK inLink,
                    Float* inArrayP,
                    Int32* inCountP,
                    Int32 inDepth,
                    Int32 &ioIteration )
{
    static Float*   sArrayP;
    Int32   i, len;


    if( ioIteration == 0 ) sArrayP = inArrayP;

    if( sArrayP == inArrayP ) len = inCountP[ioIteration];
    else {
        MLCheckFunction(inLink, "List", &len);
        if( MLError(inLink) || len != inCountP[ioIteration] )
            return MLErrorReport(inLink, "out of sequence");
    }

    ioIteration++;
    for( i = 0; i < len; i++) {
        if( ioIteration == inDepth ) MLGetDouble(inLink, sArrayP++);
        else {
            if( eError == MLReadList(inLink, inArrayP, inCountP, inDepth, ioIteration) )
                return eError;
        }
    }
    ioIteration--;

    if( MLError(inLink) )
        return MLErrorReport(inLink, "out of sequence");

    return eOK;
}



// ---------------------------------------------------------------------------
//       MLCheckMemoryReserve
// ---------------------------------------------------------------------------
//  Check memory reserve

Int32   MLCheckMemoryReserve( MLINK inLink )
{
    Int8    *reserveP[32];
    Int32   i, j;
    const Int32 kMemoryReserve = 32768;     //32 Kbytes memory reserve


    for( i = 0; i < 32; i++ ) {                 //32 small memory blocks
        reserveP[i] = new Int8[kMemoryReserve/32];
        if( reserveP[i] == nil ) {
            for( j = 0; j < i; j++ ) delete [] reserveP[j];
            return MLErrorReport(inLink, "low memory situation");
        }
    }
    for( i = 0; i < 32; i++ ) delete [] reserveP[i];
```

```
    reserveP[0] = new Int8[kMemoryReserve];     //one large memory block
    if( reserveP[0] == nil )
        return MLErrorReport(inLink, "low memory situation");
    delete [] reserveP[0];

    return eOK;
}



// ---------------------------------------------------------------------------
//       MLGetFunctionObject
// ---------------------------------------------------------------------------
//  Get function identification

Int32   MLGetFunctionObject(    MLINK inLink,
                                Int32 &outID )
{
    Int32   len, result, ID;
    char*   symbolP;


    switch( MLGetType(inLink) ) {
    case MLTKFUNC:
        MLCheckFunction(inLink, "FunctionObject", &len);
        if( MLError(inLink) || len != 1 )
            return MLErrorReport(inLink, "FunctionObject expected");
        if( !MLGetLongInteger(inLink, &ID) )
            return MLErrorReport(inLink, "integer ID expected");
        outID = ID;
        return eOK;
        break;
    case MLTKSYM:
        MLGetSymbol(inLink, &symbolP);
        result = strcmp( symbolP, "None");
        MLDisownSymbol(inLink, symbolP);
        if( result == 0 ) {
            outID = 0;
            return eOK;
        }
    default:
        return MLErrorReport(inLink, "FunctionObject or None expected");
    }

}



// ---------------------------------------------------------------------------
//       MLGetOperatorObject
// ---------------------------------------------------------------------------
//  Get operator identification

Int32   MLGetOperatorObject(    MLINK inLink,
                                Int32 &outID )
{
    Int32   len, ID;


    MLCheckFunction(inLink, "OperatorObject", &len);
    if( MLError(inLink) || len != 1 )
        return MLErrorReport(inLink, "OperatorObject expected");
    if( !MLGetLongInteger(inLink, &ID) )
        return MLErrorReport(inLink, "integer ID expected");

    outID = ID;
    return eOK;
}



// ---------------------------------------------------------------------------
//       MLGetWindowObject
// ---------------------------------------------------------------------------
```

```
//  Get window identification

Int32   MLGetWindowObject(  MLINK inLink,
                            Int32 &outID )
{
    Int32   len, ID;


    MLCheckFunction(inLink, "WindowObject", &len);
    if( MLError(inLink) || len != 1 )
        return MLErrorReport(inLink, "WindowObject expected");
    if( !MLGetLongInteger(inLink, &ID) )
        return MLErrorReport(inLink, "integer ID expected");

    outID = ID;
    return eOK;
}



// ---------------------------------------------------------------------------
//       MLEventLoop
// ---------------------------------------------------------------------------
//  Substitution for _handle_user_event

bool    gSwitchedIn = false;

extern void do_about_box( void);

#define mApple      1128
#define iAbout      1
#define mFile       1129
#define mEdit       1130

Int32   MLEventLoop( void )
{
    EventRecord event;
    Int8    eventType;
    Int16   menuID, menuItem, windowPart;
    Int32   ticks, menuResult = 0;
    Str255  daName;
    Rect    dragRect;
    WindowPtr   windowP;
    TWindow*    theWindow;


    if( gSwitchedIn ) ticks = 0;
    else ticks = 8;

    if( WaitNextEvent(everyEvent, &event, ticks, nil) ) {
        switch ( event.what ) {
        case updateEvt:
            windowP = (WindowPtr)event.message;
            theWindow = (TWindow*)GetWRefCon(windowP);
            BeginUpdate(windowP);
            theWindow->Update();
            EndUpdate(windowP);
            break ;
        case mouseDown:
            windowPart = FindWindow(event.where, &windowP);
            switch( windowPart ) {
            case inSysWindow:
                SystemClick (&event, windowP);
                break;
            case inMenuBar:
                menuResult = MenuSelect(event.where);
                break;
            case inDrag:
                dragRect = (**GetGrayRgn()).rgnBBox;
                DragWindow( windowP, event.where, &dragRect );
                break ;
            case inContent:
                if( windowP != FrontWindow() )
                    SelectWindow(windowP);
```

```
                break ;
            case inZoomIn:
            case inZoomOut:
                if( TrackBox( windowP, event.where, windowPart ) )
                {
                    theWindow = (TWindow*)GetWRefCon(windowP);
                    theWindow->Zoom();
                }
                break;
            }
        case keyDown:
            if( event.modifiers & cmdKey )
                menuResult = MenuKey((short)event.message & charCodeMask);
            break;
        case osEvt:
            eventType = event.message >> 24;
            if( eventType & suspendResumeMessage ) {
                if(event.message & resumeFlag) {
                    gSwitchedIn = true;
                }
                else {
                    gSwitchedIn = false;
                }
            }
            break;
        case kHighLevelEvent:
            AEProcessAppleEvent(&event);
            break;
        }

        menuID = HiWord(menuResult);
        menuItem = LoWord(menuResult);
        switch ( menuID ) {
        case mFile:
            MLDone = MLAbort = 1;
            break;
        case mApple:
            switch ( menuItem ) {
            case iAbout:
                do_about_box();
                break;
            default:
                GetItem(GetMHandle(mApple), menuItem, daName);
                (void) OpenDeskAcc(daName);
                break;
            }
            HiliteMenu(0);
            break;
        }

    }

    return MLDone;
}



// ---------------------------------------------------------------------------
//       MLDefaultYielder
// ---------------------------------------------------------------------------
//  Substitution for disabled yielder

extern pascal long MLDefaultYielder( MLINK mlp, MLYieldParameters yp)
{
    mlp = mlp; /* suppress unused warning */
    return MLEventLoop();
}
```

└

B.2 MathLinkGlue

B.2.1 MathLinkGlue.h

```
// ============================================================================
//  MathLinkGlue.h      (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================

#ifndef _H_MathLinkGlue
#define _H_MathLinkGlue
#pragma once

#include "TypeDefinition.h"

//  Prototypes

void    NewFunction( void );
void    DisposeFunction( void );
void    FunctionInfo( void );
void    ValueArray( void );
void    ColorArray( void );
void    GrayArray( void );
void    RedBlueArray( void );
void    BlackWhiteArray( void );
void    AbsArray( void );
void    Info( void );

void    Schroedinger2D( void );
void    Schroedinger3D( void );
void    Pauli2D( void );
void    Pauli3D( void );
void    Dirac2D( void );
void    Dirac3D( void );
void    DisposeOperator( void );
void    TimeEvolution( void );
void    OperatorInfo( void );

void    ShowWindow( void );
void    HideWindow( void );
void    WindowInfo( void );
void    BeginMovie( void );
void    EndMovie( void );

#endif
```

B.2.2 MathLinkGlue.c

```
// ============================================================================
//  MathLinkGlue.c      (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ============================================================================
//
//  MathLink glue functions

#include "MathLinkUtilities.h"
#include "MathLinkGlue.h"
#include "TFunction.h"
#include "TOperator.h"
#include "TWindow.h"
#include "TSchroedinger2D.h"
#include "TSchroedinger3D.h"
#include "TPauli2D.h"
#include "TPauli3D.h"
#include "TDirac2D.h"
#include "TDirac3D.h"
```

```
// ---------------------------------------------------------------------------
//       main
// ---------------------------------------------------------------------------
//  Start QuantumKernel

TList<TFunction>    *gFunctionList;
TList<TOperator>    *gOperatorList;
TList<TWindow>      *gWindowList;

void    main( void )
{
    const Int32 kListSize = 1024;


    gFunctionList = new TList<TFunction>();
    if( !gFunctionList ) return;
    if( !gFunctionList->New(kListSize) ) return;

    gOperatorList = new TList<TOperator>();
    if( !gOperatorList ) return;
    if( !gOperatorList->New(kListSize) ) return;

    gWindowList = new TList<TWindow>();
    if( !gWindowList ) return;
    if( !gWindowList->New(kListSize) ) return;

    MLMain( 0, nil );
}



// ---------------------------------------------------------------------------
//       NewFunction
// ---------------------------------------------------------------------------
//  Create function object

void    NewFunction( void )
{
    Int32   ID;
    TFunction   *theFunction;


    theFunction = new TFunction();
    if ( theFunction == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    if( eError == theFunction->GetArray() ) {
        delete theFunction;
        return;
    }

    ID = gFunctionList->Insert(theFunction);
    if( ID == 0 ) {
        delete theFunction;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theFunction->mID = ID;

    MLPutFunction(stdlink, "FunctionObject", 1);
    MLPutLongInteger(stdlink, ID);
}



// ---------------------------------------------------------------------------
//       DisposeFunction
// ---------------------------------------------------------------------------
//  Remove function object

void    DisposeFunction( void )
```

```
{
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Remove(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    delete theFunction;
    MLPutSymbol(stdlink, "Null");
}



// ---------------------------------------------------------------------------
//       FunctionInfo
// ---------------------------------------------------------------------------
//  Return function info

void    FunctionInfo( void )
{
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutInfo();
}



// ---------------------------------------------------------------------------
//       ValueArray
// ---------------------------------------------------------------------------
//  Return float array

void    ValueArray( void )
{
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutArray();
}



// ---------------------------------------------------------------------------
//       ColorArray
// ---------------------------------------------------------------------------
//  Return RGB color array

void    ColorArray( void )
{
```

```
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutColor();
}



// ---------------------------------------------------------------------------
//       GrayArray
// ---------------------------------------------------------------------------
//  Return gray array

void    GrayArray( void )
{
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutGray();
}



// ---------------------------------------------------------------------------
//       RedBlueArray
// ---------------------------------------------------------------------------
//  Return red-blue color array

void    RedBlueArray( void )
{
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutRedBlue();
}



// ---------------------------------------------------------------------------
//       BlackWhiteArray
// ---------------------------------------------------------------------------
//  Return black-white array

void    BlackWhiteArray( void )
{
    TFunction   *theFunction;
    Int32   ID;
```

```
    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutBlackWhite();
}



// ---------------------------------------------------------------------------
//       AbsArray
// ---------------------------------------------------------------------------
//  Return abs array

void    AbsArray( void )
{
    TFunction   *theFunction;
    Int32   ID;


    if( eError == MLGetFunctionObject(stdlink, ID) ) return;

    theFunction = gFunctionList->Fetch(ID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theFunction->PutAbs();
}



// ---------------------------------------------------------------------------
//       Info
// ---------------------------------------------------------------------------
//  Return function, operator and window info

void    Info( void )
{
    Int32   len, size, ID;
    TFunction   *theFunction;
    TOperator   *theOperator;
    TWindow     *theWindow;


    MLPutFunction(stdlink, "List", 3);

    size = gFunctionList->GetSize();
    len = 0;
    for( ID = 0; ID < size; ID++ ) {
        theFunction = gFunctionList->Fetch(ID);
        if( theFunction ) len++;
    }
    MLPutFunction(stdlink, "List", len);
    for( ID = 0; ID < size; ID++ ) {
        theFunction = gFunctionList->Fetch(ID);
        if( theFunction ) {
            MLPutFunction(stdlink, "List", 2);
            MLPutFunction(stdlink, "FunctionObject", 1);
            MLPutLongInteger(stdlink, ID);
            theFunction->PutInfo();
        }
    }

    size = gOperatorList->GetSize();
    len = 0;
    for( ID = 0; ID < size; ID++ ) {
```

```
        theOperator = gOperatorList->Fetch(ID);
        if( theOperator ) len++;
    }
    MLPutFunction(stdlink, "List", len);
    for( ID = 0; ID < size; ID++ ) {
        theOperator = gOperatorList->Fetch(ID);
        if( theOperator ) {
            MLPutFunction(stdlink, "List", 2);
            MLPutFunction(stdlink, "OperatorObject", 1);
            MLPutLongInteger(stdlink, ID);
            theOperator->PutInfo();
        }
    }

    size = gWindowList->GetSize();
    len = 0;
    for( ID = 0; ID < size; ID++ ) {
        theWindow = gWindowList->Fetch(ID);
        if( theWindow ) len++;
    }
    MLPutFunction(stdlink, "List", len);
    for( ID = 0; ID < size; ID++ ) {
        theWindow = gWindowList->Fetch(ID);
        if( theWindow ) {
            MLPutFunction(stdlink, "List", 2);
            MLPutFunction(stdlink, "WindowObject", 1);
            MLPutLongInteger(stdlink, ID);
            theWindow->PutInfo();
        }
    }

}



// ------------------------------------------------------------------------
//       Schroedinger2D
// ------------------------------------------------------------------------
//  Create operator object

void    Schroedinger2D( void )
{
    Int32   scalarID, vectorID, domainID, ID;
    Float   mass, charge, units;
    TOperator   *theOperator;


    if( eError == MLGetFunctionObject(stdlink, scalarID) ) return;
    if( eError == MLGetFunctionObject(stdlink, vectorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, domainID) ) return;

    MLGetDouble(stdlink, &mass);
    MLGetDouble(stdlink, &charge);
    MLGetDouble(stdlink, &units);
    if( MLError(stdlink) ) {
        MLErrorReport(stdlink, "real numbers for mass, charge and units expected");
        return;
    }

    theOperator = new TSchroedinger2D(scalarID, vectorID, domainID, mass, charge, units);
    if( theOperator == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    ID = gOperatorList->Insert(theOperator);
    if( ID == 0 ) {
        delete theOperator;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theOperator->mID = ID;

    MLPutFunction(stdlink, "OperatorObject", 1);
```

```
    MLPutLongInteger(stdlink, ID);
}



// ---------------------------------------------------------------------------
//       Schroedinger3D
// ---------------------------------------------------------------------------
//  Create operator object

void    Schroedinger3D( void )
{
    Int32   scalarID, vectorID, domainID, ID;
    Float   mass, charge, units;
    TOperator   *theOperator;


    if( eError == MLGetFunctionObject(stdlink, scalarID) ) return;
    if( eError == MLGetFunctionObject(stdlink, vectorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, domainID) ) return;

    MLGetDouble(stdlink, &mass);
    MLGetDouble(stdlink, &charge);
    MLGetDouble(stdlink, &units);
    if( MLError(stdlink) ) {
        MLErrorReport(stdlink, "real numbers for mass, charge and units expected");
        return;
    }

    theOperator = new TSchroedinger3D(scalarID, vectorID, domainID, mass, charge, units);
    if( theOperator == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    ID = gOperatorList->Insert(theOperator);
    if( ID == 0 ) {
        delete theOperator;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theOperator->mID = ID;

    MLPutFunction(stdlink, "OperatorObject", 1);
    MLPutLongInteger(stdlink, ID);
}



// ---------------------------------------------------------------------------
//       Pauli2D
// ---------------------------------------------------------------------------
//  Create operator object

void    Pauli2D( void )
{
    Int32   scalarID, vectorID, domainID, ID;
    Float   mass, charge, units;
    TOperator   *theOperator;


    if( eError == MLGetFunctionObject(stdlink, scalarID) ) return;
    if( eError == MLGetFunctionObject(stdlink, vectorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, domainID) ) return;

    MLGetDouble(stdlink, &mass);
    MLGetDouble(stdlink, &charge);
    MLGetDouble(stdlink, &units);
    if( MLError(stdlink) ) {
        MLErrorReport(stdlink, "real numbers for mass, charge and units expected");
        return;
    }

    theOperator = new TPauli2D(scalarID, vectorID, domainID, mass, charge, units);
```

```
    if( theOperator == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    ID = gOperatorList->Insert(theOperator);
    if( ID == 0 ) {
        delete theOperator;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theOperator->mID = ID;

    MLPutFunction(stdlink, "OperatorObject", 1);
    MLPutLongInteger(stdlink, ID);
}



// ------------------------------------------------------------------------
//       Pauli3D
// ------------------------------------------------------------------------
//  Create operator object

void    Pauli3D( void )
{
    Int32   scalarID, vectorID, domainID, ID;
    Float   mass, charge, units;
    TOperator   *theOperator;


    if( eError == MLGetFunctionObject(stdlink, scalarID) ) return;
    if( eError == MLGetFunctionObject(stdlink, vectorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, domainID) ) return;

    MLGetDouble(stdlink, &mass);
    MLGetDouble(stdlink, &charge);
    MLGetDouble(stdlink, &units);
    if( MLError(stdlink) ) {
        MLErrorReport(stdlink, "real numbers for mass, charge and units expected");
        return;
    }

    theOperator = new TPauli3D(scalarID, vectorID, domainID, mass, charge, units);
    if( theOperator == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    ID = gOperatorList->Insert(theOperator);
    if( ID == 0 ) {
        delete theOperator;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theOperator->mID = ID;

    MLPutFunction(stdlink, "OperatorObject", 1);
    MLPutLongInteger(stdlink, ID);
}



// ------------------------------------------------------------------------
//       Dirac2D
// ------------------------------------------------------------------------
//  Create operator object

void    Dirac2D( void )
{
    Int32   scalarID, vectorID, domainID, ID;
    Float   mass, charge, units;
    TOperator   *theOperator;
```

```
    if( eError == MLGetFunctionObject(stdlink, scalarID) ) return;
    if( eError == MLGetFunctionObject(stdlink, vectorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, domainID) ) return;

    MLGetDouble(stdlink, &mass);
    MLGetDouble(stdlink, &charge);
    MLGetDouble(stdlink, &units);

    if( MLError(stdlink) ) {
        MLErrorReport(stdlink, "real numbers for mass, charge and units expected");
        return;
    }

    theOperator = new TDirac2D(scalarID, vectorID, domainID, mass, charge, units);
    if( theOperator == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    ID = gOperatorList->Insert(theOperator);
    if( ID == 0 ) {
        delete theOperator;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theOperator->mID = ID;

    MLPutFunction(stdlink, "OperatorObject", 1);
    MLPutLongInteger(stdlink, ID);
}


// ---------------------------------------------------------------------------
//       Dirac3D
// ---------------------------------------------------------------------------
//  Create operator object

void    Dirac3D( void )
{
    Int32   scalarID, vectorID, domainID, ID;
    Float   mass, charge, units;
    TOperator   *theOperator;


    if( eError == MLGetFunctionObject(stdlink, scalarID) ) return;
    if( eError == MLGetFunctionObject(stdlink, vectorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, domainID) ) return;

    MLGetDouble(stdlink, &mass);
    MLGetDouble(stdlink, &charge);
    MLGetDouble(stdlink, &units);

    if( MLError(stdlink) ) {
        MLErrorReport(stdlink, "real numbers for mass, charge and units expected");
        return;
    }

    theOperator = new TDirac3D(scalarID, vectorID, domainID, mass, charge, units);
    if( theOperator == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    ID = gOperatorList->Insert(theOperator);
    if( ID == 0 ) {
        delete theOperator;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theOperator->mID = ID;

    MLPutFunction(stdlink, "OperatorObject", 1);
```

```
    MLPutLongInteger(stdlink, ID);
}



// ---------------------------------------------------------------------------
//       DisposeOperator
// ---------------------------------------------------------------------------
//  Remove operator object

void    DisposeOperator( void )
{
    TOperator   *theOperator;
    Int32   ID;


    if( eError == MLGetOperatorObject(stdlink, ID) ) return;

    theOperator = gOperatorList->Remove(ID);
    if( !theOperator ) {
        MLErrorReport(stdlink, "invalid operator ID");
        return;
    }

    delete theOperator;
    MLPutSymbol(stdlink, "Null");
}



// ---------------------------------------------------------------------------
//       TimeEvolution
// ---------------------------------------------------------------------------
//  Calculate time evolution

void    TimeEvolution( void )
{
    Int32   operatorID, functionID, fractal, steps;
    Float   timeStep;
    TOperator   *theOperator;
    TFunction   *theFunction;


    if( eError == MLGetOperatorObject(stdlink, operatorID) ) return;
    if( eError == MLGetFunctionObject(stdlink, functionID) ) return;

    if( !MLGetDouble(stdlink, &timeStep) ) {
        MLErrorReport(stdlink, "real number for timestep expected");
        return;
    }
    if( !MLGetLongInteger(stdlink, &fractal) ) {
        MLErrorReport(stdlink, "integer for fractal expected");
        return;
    }
    if( !MLGetLongInteger(stdlink, &steps) ) {
        MLErrorReport(stdlink, "integer for steps expected");
        return;
    }

    theOperator = gOperatorList->Fetch(operatorID);
    if( !theOperator ) {
        MLErrorReport(stdlink, "invalid operator ID");
        return;
    }

    theFunction = gFunctionList->Fetch(functionID);
    if( !theFunction ) {
        MLErrorReport(stdlink, "invalid function ID");
        return;
    }

    theOperator->TimeEvolution(theFunction, timeStep, fractal, steps);
}
```

```
// ---------------------------------------------------------------------------
//       OperatorInfo
// ---------------------------------------------------------------------------
//  Return operator info

void    OperatorInfo( void )
{
    TOperator   *theOperator;
    Int32   ID;


    if( eError == MLGetOperatorObject(stdlink, ID) ) return;

    theOperator = gOperatorList->Fetch(ID);
    if( !theOperator ) {
        MLErrorReport(stdlink, "invalid operator ID");
        return;
    }

    theOperator->PutInfo();
}



// ---------------------------------------------------------------------------
//       ShowWindow
// ---------------------------------------------------------------------------
//  Create window object

void    ShowWindow( void )
{
    Int32   functionID, ID, mode, slice;
    TWindow *theWindow;


    if( eError == MLGetFunctionObject(stdlink, functionID) ) return;

    if( !MLGetLongInteger(stdlink, &mode) ) {
        MLErrorReport(stdlink, "integer for mode expected");
        return;
    }
    if( !MLGetLongInteger(stdlink, &slice) ) {
        MLErrorReport(stdlink, "integer for slice expected");
        return;
    }

    theWindow = new TWindow(functionID, mode, slice);
    if( theWindow == nil ) {
        MLErrorReport(stdlink, "out of memory");
        return;
    }

    if( eError == theWindow->Create() ) {
        delete theWindow;
        return;
    }

    ID = gWindowList->Insert(theWindow);
    if( ID == 0 ) {
        delete theWindow;
        MLErrorReport(stdlink, "ID list is full");
        return;
    }
    theWindow->mID = ID;
    theWindow->Show();

    MLPutFunction(stdlink, "WindowObject", 1);
    MLPutLongInteger(stdlink, ID);
}
```

```
// ---------------------------------------------------------------------------
//       HideWindow
// ---------------------------------------------------------------------------
//  Remove window object

void    HideWindow( void )
{
    TWindow *theWindow;
    Int32   ID;


    if( eError == MLGetWindowObject(stdlink, ID) ) return;

    theWindow = gWindowList->Fetch(ID);
    if( !theWindow ) {
        MLErrorReport(stdlink, "invalid window ID");
        return;
    }
    if( theWindow->IsMovie() == true ) {
        MLErrorReport(stdlink, "turn off movie recording before hiding the window");
        return;
    }

    theWindow = gWindowList->Remove(ID);
    if( !theWindow ) {
        MLErrorReport(stdlink, "invalid window ID");
        return;
    }

    delete theWindow;
    MLPutSymbol(stdlink, "Null");
}



// ---------------------------------------------------------------------------
//       WindowInfo
// ---------------------------------------------------------------------------
//  Return window info

void    WindowInfo( void )
{
    TWindow *theWindow;
    Int32   ID;


    if( eError == MLGetWindowObject(stdlink, ID) ) return;

    theWindow = gWindowList->Fetch(ID);
    if( !theWindow ) {
        MLErrorReport(stdlink, "invalid window ID");
        return;
    }

    theWindow->PutInfo();
}



// ---------------------------------------------------------------------------
//       BeginMovie
// ---------------------------------------------------------------------------
//  Begin movie recording

void    BeginMovie( void )
{
    TWindow *theWindow;
    Int32   ID;


    if( eError == MLGetWindowObject(stdlink, ID) ) return;

    theWindow = gWindowList->Fetch(ID);
    if( !theWindow ) {
```

```
        MLErrorReport(stdlink, "invalid window ID");
        return;
    }

    theWindow->BeginMovie();
}



// ---------------------------------------------------------------------------
//       EndMovie
// ---------------------------------------------------------------------------
//  End movie recording

void    EndMovie( void )
{
    TWindow *theWindow;
    Int32   ID;


    if( eError == MLGetWindowObject(stdlink, ID) ) return;

    theWindow = gWindowList->Fetch(ID);
    if( !theWindow ) {
        MLErrorReport(stdlink, "invalid window ID");
        return;
    }

    theWindow->EndMovie();
}


└
```

B.3 TypeDefinition

B.3.1 TypeDefinition.h

```
┌


// ===========================================================================
//  TypeDefinition.h    (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ===========================================================================

#ifndef _H_TypeDefinition
#define _H_TypeDefinition
#pragma once

//  Float type
#define __USE_FLOAT__

#ifdef __USE_FLOAT__
    #define MLPutDoubleArray MLPutFloatArray
    #define MLGetDouble MLGetFloat
    typedef float       Float;
#else
    typedef double      Float;
#endif

//  Integer types
typedef long        Int32;
typedef short       Int16;
typedef char        Int8;

//  Constants
extern const double pi;

#endif


└
```

B.4 Templates

B.4.1 mathlink

```
// ====================================================================
//  mathlink            (C) 1996-1998 Manfred Liebmann. All rights reserved.
// ====================================================================
//
//  Mathlink templates

:Evaluate: Print["QuantumKernel 1.0.0 PPC, Copyright (C) 1996-98 Manfred Liebmann"];
:Evaluate: QuantumKernel::err = "`1`.";

//  TFunction

:Begin:
:Function: NewFunction
:Pattern: NewFunction[ arrays__ ]
:Arguments: { { arrays } }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: DisposeFunction
:Pattern: DisposeFunction[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: FunctionInfo
:Pattern: FunctionInfo[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: ValueArray
:Pattern: ValueArray[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: ColorArray
:Pattern: ColorArray[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: GrayArray
:Pattern: GrayArray[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: RedBlueArray
:Pattern: RedBlueArray[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
```

```
:Function: BlackWhiteArray
:Pattern: BlackWhiteArray[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: AbsArray
:Pattern: AbsArray[ function_ ]
:Arguments: { function }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: Info
:Pattern: Info[ ]
:Arguments: { }
:ArgumentTypes: { }
:ReturnType: Manual
:End:

//  TOperator

:Begin:
:Function: Schroedinger2D
:Pattern: Schroedinger2D[scalar_:None,vector_:None,domain_:None,mass_:1,charge_:1,units_:1]
:Arguments: { scalar, vector, domain, mass, charge, units }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: Schroedinger3D
:Pattern: Schroedinger3D[scalar_:None,vector_:None,domain_:None,mass_:1,charge_:1,units_:1]
:Arguments: { scalar, vector, domain, mass, charge, units }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: Pauli2D
:Pattern: Pauli2D[scalar_:None,vector_:None,domain_:None,mass_:1,charge_:1,units_:1]
:Arguments: { scalar, vector, domain, mass, charge, units }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: Pauli3D
:Pattern: Pauli3D[scalar_:None,vector_:None,domain_:None,mass_:1,charge_:1,units_:1]
:Arguments: { scalar, vector, domain, mass, charge, units }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: Dirac2D
:Pattern: Dirac2D[scalar_:None,vector_:None,domain_:None,mass_:1,charge_:1,units_:1]
:Arguments: { scalar, vector, domain, mass, charge, units }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: Dirac3D
:Pattern: Dirac3D[scalar_:None,vector_:None,domain_:None,mass_:1,charge_:1,units_:1]
:Arguments: { scalar, vector, domain, mass, charge, units }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
```

```
:Function: DisposeOperator
:Pattern: DisposeOperator[ operator_ ]
:Arguments: { operator }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: OperatorInfo
:Pattern: OperatorInfo[ operator_ ]
:Arguments: { operator }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: TimeEvolution
:Pattern: TimeEvolution[ operator_, function_, timestep_, fractal_:4, steps_:1 ]
:Arguments: { operator, function, timestep, fractal, steps }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

//  TWindow

:Begin:
:Function: ShowWindow
:Pattern: ShowWindow[ function_, mode_:0, slice_:0 ]
:Arguments: { function, mode, slice }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: HideWindow
:Pattern: HideWindow[ window_ ]
:Arguments: { window }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: WindowInfo
:Pattern: WindowInfo[ window_ ]
:Arguments: { window }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: BeginMovie
:Pattern: BeginMovie[ window_ ]
:Arguments: { window }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

:Begin:
:Function: EndMovie
:Pattern: EndMovie[ window_ ]
:Arguments: { window }
:ArgumentTypes: { Manual }
:ReturnType: Manual
:End:

└
```

Anhang C

Mathematica-Package

C.1 QuantumMechanics

C.1.1 init.m

```
(* :Title: QuantumMechanics *)

(* :Author: Manfred Liebmann *)

(* :Summary: Package Initialization *)

(* :Context: QuantumMechanics`Kernel`init` *)

(* :Package Version: 1.0 *)

(* :Copyright: Copyright (C) 1996-98 Manfred Liebmann *)

(* :Source: Manfred Liebmann: Diploma Thesis, University of Graz, Austria, 1999 *)

(* :Mathematica Version: 3.0 *)


BeginPackage["QuantumMechanics`Kernel`init`"];
EndPackage[];

(* QuantumMechanics`QuantumKernel` *)

DeclarePackage["QuantumMechanics`QuantumKernel`",
    {"QuantumLink", "NewFunction", "DisposeFunction", "FunctionInfo",
    "ValueArray", "ColorArray", "GrayArray", "RedBlueArray",
    "BlackWhiteArray", "AbsArray", "Info", "Schroedinger2D",
    "Schroedinger3D", "Pauli2D", "Pauli3D", "Dirac2D", "Dirac3D",
    "DisposeOperator", "OperatorInfo", "TimeEvolution", "ShowWindow",
    "HideWindow", "WindowInfo", "BeginMovie", "EndMovie",
    "RenderComplex2DColor", "RenderComplex2DGray", "RenderComplex3DColor",
    "RenderComplex3DGray", "RenderScalar2DRedBlue", "RenderScalar2DBlackWhite",
    "RenderScalar3DRedBlue", "RenderScalar3DBlackWhite"}];

Null
```

C.1.2 QuantumKernel.m

```
(* :Title: QuantumKernel *)
```

```
(* :Author: Manfred Liebmann *)

(* :Summary: Interactiv Visualization System for Quantum Mechanical Problems. *)

(* :Context: QuantumMechanics`QuantumKernel` *)

(* :Package Version: 1.0 *)

(* :Copyright: Copyright (C) 1996-98 Manfred Liebmann *)

(* :Source: Manfred Liebmann: Diploma Thesis, University of Graz, Austria, 1999 *)

(* :Mathematica Version: 3.0 *)


BeginPackage["QuantumMechanics`QuantumKernel`"];

    (* QuantumKernel *)

    QuantumLink::usage = "QuantumLink";

    (* TFunction Objects *)

    NewFunction::usage =
        "NewFunction[arrays]";
    DisposeFunction::usage =
        "DisposeFunction[function]";
    FunctionInfo::usage =
        "FunctionInfo[function]";
    ValueArray::usage =
        "ValueArray[function]";
    ColorArray::usage =
        "ColorArray[function]";
    GrayArray::usage =
        "GrayArray[function]";
    RedBlueArray::usage =
        "RedBlueArray[function]";
    BlackWhiteArray::usage =
        "BlackWhiteArray[function]";
    AbsArray::usage =
        "AbsArray[function]";
    Info::usage =
        "Info[]"

    (* TOperator Objects*)

    Schroedinger2D::usage =
        "Schroedinger2D[scalar, vector, domain, mass, charge, units]";
    Schroedinger3D::usage =
        "Schroedinger3D[scalar, vector, domain, mass, charge, units]";
    Pauli2D::usage =
        "Pauli2D[scalar, vector, domain, mass, charge, units]";
    Pauli3D::usage =
        "Pauli3D[scalar, vector, domain, mass, charge, units]";
    Dirac2D::usage =
        "Dirac2D[scalar, vector, domain, mass, charge, units]";
    Dirac3D::usage =
        "Dirac3D[scalar, vector, domain, mass, charge, units]";
    DisposeOperator::usage =
        "DisposeOperator[operator]";
    OperatorInfo::usage =
        "OperatorInfo[operator]";
    TimeEvolution::usage =
        "TimeEvolution[operator, function, timestep, fractal, steps]";

    (* TWindow Objects*)

    ShowWindow::usage =
        "ShowWindow[function, mode, slice]";
    HideWindow::usage =
        "HideWindow[window]";
    WindowInfo::usage =
        "WindowInfo[window]";
    BeginMovie::usage =
```

```
        "BeginMovie[window]";
    EndMovie::usage =
        "EndMovie[window]";

    (* Render Functions *)

    RenderComplex2DColor::usage =
        "RenderComplex2DColor[function, options]";
    RenderComplex2DGray::usage =
        "RenderComplex2DGray[function, options]";

    RenderComplex3DColor::usage =
        "RenderComplex3DColor[function, options]";
    RenderComplex3DGray::usage =
        "RenderComplex3DGray[function, options]";

    RenderScalar2DRedBlue::usage =
        "RenderScalar2DRedBlue[function, options]";
    RenderScalar2DBlackWhite::usage =
        "RenderScalar2DBlackWhite[function, options]";

    RenderScalar3DRedBlue::usage =
        "RenderScalar3DRedBlue[function, options]";
    RenderScalar3DBlackWhite::usage =
        "RenderScalar3DBlackWhite[function, options]";

    Begin["`Private`"];

        QuantumLink =
            Install[":AddOns:Applications:QuantumMechanics:QuantumKernel"];

        RenderComplex2DColor[psi_, opts___] :=
            Show[Graphics[RasterArray[ColorArray[psi]]],opts];
        RenderComplex2DGray[psi_, opts___] :=
            Show[Graphics[RasterArray[GrayArray[psi]]],opts];

        RenderComplex3DColor[psi_, opts___] :=
            ListPlot3D[AbsArray[psi],Map[Rest,ColorArray[psi],{0,1}],opts];
        RenderComplex3DGray[psi_, opts___] :=
            ListPlot3D[Power[AbsArray[psi],2],Map[Rest,GrayArray[psi],{0,1}],opts];

        RenderScalar2DRedBlue[v_, opts___] :=
            Show[Graphics[RasterArray[RedBlueArray[v]]],opts];
        RenderScalar2DBlackWhite[v_, opts___] :=
            Show[Graphics[RasterArray[BlackWhiteArray[v]]],opts];

        RenderScalar3DRedBlue[v_, opts___] :=
            ListPlot3D[Part[ValueArray[v],1],Map[Rest,RedBlueArray[v],{0,1}],opts];
        RenderScalar3DBlackWhite[v_, opts___] :=
            ListPlot3D[Part[ValueArray[v],1],Map[Rest,BlackWhiteArray[v],{0,1}],opts];

    End[];

EndPackage[];

⌊
```

C.1.3 QuantumMechanics.nb

```
⌜

QUANTUMMECHANICS

INSTALL QUANTUMKERNEL

<<QuantumMechanics`QuantumKernel`

LINK OBJECT

QuantumLink

UNINSTALL QUANTUMKERNEL
```

```
Uninstall[QuantumLink]

└
```

Anhang D

Mathematica-Notebooks

D.1 Schrödinger-Gleichung

D.1.1 TDSE2D.nb

```
TDSE2D: AHARONOV-BOHM EFFECT

DOMAIN

Domain2D[h_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    If[ (x1 >= 15/32 && x1 <= 17/32 &&
        (x2 >= 19/32 || x2 <= 13/32)) ||
        (x1-1/2)^2 + (x2-1/2)^2 <= 1/32^2, -1, 1]]}

h = 1/127; n1 = 1/h+1; n2 = 1/h+1;
Di = Array[Domain2D[h], {n2, n1}, 0];
De = NewFunction[Di];

Dw = ShowWindow[De, 1];

HideWindow[Dw];

BeginMovie[Dw];
EndMovie[Dw];

VECTOR POTENTIAL

Vector2D[h_,Alpha_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    I Alpha / ((x1-1/2) - I(x2-1/2))]}

Alpha = 1/3;
Ai = Array[Vector2D[h, Alpha], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai]];

Aw = ShowWindow[Ae, 0];

HideWindow[Aw];

BeginMovie[Aw];
EndMovie[Aw];

WAVEFUNCTION

GaugeGauss2D[h_,e_,Alpha_,k1_,k2_,y1_,y2_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-y1)^2 + (x2-y2)^2)/2]*Exp[I(k1 x1 + k2 x2)]*
    Exp[I e Alpha Arg[(x1-1/2) + I(x2-1/2)]]]}
```

```
e = 1; k1 = -16 Pi; k2 = 0 Pi; y1 = 3/4; y2 = 1/2; a = 10; b = 200;
Psii = Array[GaugeGauss2D[h, e, Alpha, k1, k2, y1, y2, a, b], {n2, n1}, 0];
Psie = NewFunction[Re[Psii], Im[Psii]];

Psiw0 = ShowWindow[Psie, 0];
Psiw1 = ShowWindow[Psie, 1];

HideWindow[Psiw0];
HideWindow[Psiw1];

HAMILTON OPERATOR

scalar = None; vector = Ae; domain = De;
mass = 1.; charge = 1.e; units = 1.h;
He = Schroedinger2D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];

EndMovie[Psiw0];
EndMovie[Psiw1];

opts2D = {AspectRatio->n2/n1};
opts3D = {AspectRatio->n2/n1, Mesh->False, PlotRange->All};

RenderScalar2DBlackWhite[De, opts2D];

RenderScalar3DBlackWhite[De, opts3D];

RenderComplex2DGray[Psie, opts2D];

RenderComplex3DGray[Psie, opts3D];

RenderScalar2DRedBlue[NewFunction[Part[ValueArray[Psie],1]], opts2D];

RenderScalar3DRedBlue[NewFunction[Part[ValueArray[Psie],1]], opts3D];

RenderComplex2DColor[Psie, opts2D];

RenderComplex3DColor[Psie, opts3D];
```

└

D.1.2 TDSE3D.nb

┌

```
TDSE3D: BALL

DOMAIN

Domain2D[h_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    If[ (x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2 <= 1/16^2 ||
        (x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2 >= 1/2^2, -1, 1]]}

h = 1/63; n1 = 1/h+1; n2 = 1/h+1; n3 = 1/h+1;
Di = Array[Domain2D[h], {n3, n2, n1}, 0];
De = NewFunction[Di];

WAVEFUNCTION

Gauss3D[h_,a_,b_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    a*Exp[-b((x1-3/4)^2 + (x2-1/2)^2 + (x3-1/2)^2)/2]]}
```

```
a = 10; b = 200;
Psii = N[Array[Gauss3D[h, a, b], {n3, n2, n1}, 0]];
Psie = NewFunction[Psii, 0 Psii];

slices = {31, 39, 47, 55};
Psiw0 = ShowWindow[Psie, 0, #]& /@ slices;
Psiw1 = ShowWindow[Psie, 1, #]& /@ slices;

HideWindow[#]& /@ Psiw0;
HideWindow[#]& /@ Psiw1;

HAMILTON OPERATOR

scalar = None; vector = None; domain = De;
mass = 1.; charge = 1.; units = 1.h;
He = Schroedinger3D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2 2/3; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[#]& /@ Psiw0;
BeginMovie[#]& /@ Psiw1;

EndMovie[#]& /@ Psiw0;
EndMovie[#]& /@ Psiw1;
```

└

D.2 Pauli-Gleichung

D.2.1 TDPE2D.nb

┌

```
TDPE2D: MAGNETIC DISC

VECTOR POTENTIAL

Vector2D[h_,Alpha_,R_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    If[ (x1-1/2)^2 + (x2-1/2)^2 <= R^2,
        I Alpha / R^2 ((x1-1/2) + I(x2-1/2)),
        I Alpha / ((x1-1/2) - I(x2-1/2))]]}

h = 1/127; Alpha = 2; R = 1/16; n1 = 1/h+1; n2 = 1/h+1;
Ai = Array[Vector2D[h, Alpha, R], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai]];

Aw = ShowWindow[Ae, 0];

HideWindow[Aw];

BeginMovie[Aw];
EndMovie[Aw];

WAVEFUNCTION

GaugeGauss2D[h_,e_,Alpha_,k1_,k2_,y1_,y2_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-y1)^2 + (x2-y2)^2)/2]*Exp[I(k1 x1 + k2 x2)]*
    Exp[I e Alpha Arg[(x1-1/2) + I(x2-1/2)]]]}

e = 1; k1 = -16 Pi; k2 = 0 Pi; y1 = 3/4; y2 = 1/2; a = 10; b = 200;
Psii = Array[GaugeGauss2D[h, e, Alpha, k1, k2, y1, y2, a, b], {n2, n1}, 0];
Psie = NewFunction[Re[Psii], Im[Psii]];

Psiw0 = ShowWindow[Psie, 0];
```

```
Psiw1 = ShowWindow[Psie, 1];

HideWindow[Psiw0];
HideWindow[Psiw1];

HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.e; units = 1.h;
He = Pauli2D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 4.h^2; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];

EndMovie[Psiw0];
EndMovie[Psiw1];
```

D.2.2 TDPE3D.nb

```
TDPE3D: SPIN

VECTOR POTENTIAL

Vector3D[h_,B_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    I B/2 ((x1-3/4) + I(x2-1/2))]}

h = 1/63; B = 32 Pi; n1 = 1/h+1; n2 = 1/h+1; n3 = 1/h+1;
Ai = Array[Vector3D[h, B], {n3, n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai], 0 Ai];

WAVEFUNCTION

Gauss3D[h_,a_,b_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    a*Exp[-b((x1-3/4)^2 + (x2-1/2)^2 + (x3-1/2)^2)/2]]}

a = 10; b = 200;
Psii = N[Array[Gauss3D[h, a, b], {n3, n2, n1}, 0]];
Psie = NewFunction[Psii, 0 Psii, Psii, 0 Psii];

slices = {31, 39, 47, 55};
Psiw0 = ShowWindow[Psie, 0, #]& /@ slices;
Psiw1 = ShowWindow[Psie, 1, #]& /@ slices;
Psiw2 = ShowWindow[Psie, 2, #]& /@ slices;
Psiw3 = ShowWindow[Psie, 3, #]& /@ slices;
Psiw4 = ShowWindow[Psie, 4, #]& /@ slices;
Psiw5 = ShowWindow[Psie, 5, #]& /@ slices;

HideWindow[#]& /@ Psiw0;
HideWindow[#]& /@ Psiw1;
HideWindow[#]& /@ Psiw2;
HideWindow[#]& /@ Psiw3;
HideWindow[#]& /@ Psiw4;
HideWindow[#]& /@ Psiw5;

HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.; units = 1.h;
He = Pauli3D[scalar, vector, domain, mass, charge, units];
```

```
TIME EVOLUTION

t = 4.h^2 2/3; fractal = 6; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[#]& /@ Psiw0;
BeginMovie[#]& /@ Psiw1;
BeginMovie[#]& /@ Psiw2;
BeginMovie[#]& /@ Psiw3;
BeginMovie[#]& /@ Psiw4;
BeginMovie[#]& /@ Psiw5;

EndMovie[#]& /@ Psiw0;
EndMovie[#]& /@ Psiw1;
EndMovie[#]& /@ Psiw2;
EndMovie[#]& /@ Psiw3;
EndMovie[#]& /@ Psiw4;
EndMovie[#]& /@ Psiw5;

└
```

D.3 Dirac-Gleichung

D.3.1 TDDE2D.nb

```
┌

TDDE2D: BOUND STATE

VECTOR POTENTIAL

Vector2D[h_,B_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    I B/2 ((x1-1/2) + I(x2-1/2))]}

h = 1/127; B = 256; n1 = 1/h+1; n2 = 1/h+1;
Ai = Array[Vector2D[h, B], {n2, n1}, 0];
Ae = NewFunction[Re[Ai], Im[Ai], 0 Ai];

WAVEFUNCTION

Gauss2D[h_,a_,b_] :=
Compile @@ {{y,x}, With[{x1 = x h, x2 = y h},
    a*Exp[-b((x1-1/2)^2 + (x2-1/2)^2)/2]]}

a = 10; b = 200;
Psii = Array[Gauss2D[h, a, b], {n2, n1}, 0];
Psie = NewFunction[Psii, 0 Psii, Psii, 0 Psii];

Psiw0 = ShowWindow[Psie, 0];
Psiw1 = ShowWindow[Psie, 1];
Psiw2 = ShowWindow[Psie, 2];
Psiw3 = ShowWindow[Psie, 3];
Psiw4 = ShowWindow[Psie, 4];
Psiw5 = ShowWindow[Psie, 5];

HideWindow[Psiw0];
HideWindow[Psiw1];
HideWindow[Psiw2];
HideWindow[Psiw3];
HideWindow[Psiw4];
HideWindow[Psiw5];

HAMILTON OPERATOR

scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.; units = 1.h;
He = Dirac2D[scalar, vector, domain, mass, charge, units];
```

```
TIME EVOLUTION

t = 1.h; fractal = 4; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[Psiw0];
BeginMovie[Psiw1];
BeginMovie[Psiw2];
BeginMovie[Psiw3];
BeginMovie[Psiw4];
BeginMovie[Psiw5];

EndMovie[Psiw0];
EndMovie[Psiw1];
EndMovie[Psiw2];
EndMovie[Psiw3];
EndMovie[Psiw4];
EndMovie[Psiw5];
```

D.3.2 TDDE3D.nb

```
TDDE3D: BOUND STATE 2

VECTOR POTENTIAL

Vector3D[h_,c_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    c ((x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2)]}

h = 1/63; c = 200; n1 = 1/h+1; n2 = 1/h+1; n3 = 1/h+1;
Ai = Array[Vector3D[h, c], {n3, n2, n1}, 0];
Ae = NewFunction[0 Ai, 0 Ai, 0 Ai, Ai];

WAVEFUNCTION

Gauss3D[h_,a_,b_] :=
Compile @@ {{z,y,x}, With[{x1 = x h, x2 = y h, x3 = z h},
    a*Exp[-b((x1-1/2)^2 + (x2-1/2)^2 + (x3-1/2)^2)/2]]}

a = 10; b = 200;
Psii = N[Array[Gauss3D[h, a, b], {n3, n2, n1}, 0]];
Psie = NewFunction[Psii, 0 Psii, Psii, 0 Psii,
                   Psii, 0 Psii, Psii, 0 Psii];

slices = {31, 39, 47, 55};
Psiw0 = ShowWindow[Psie, 0, #]& /@ slices;
Psiw1 = ShowWindow[Psie, 1, #]& /@ slices;
Psiw2 = ShowWindow[Psie, 2, #]& /@ slices;
Psiw3 = ShowWindow[Psie, 3, #]& /@ slices;
Psiw4 = ShowWindow[Psie, 4, #]& /@ slices;
Psiw5 = ShowWindow[Psie, 5, #]& /@ slices;
Psiw6 = ShowWindow[Psie, 6, #]& /@ slices;
Psiw7 = ShowWindow[Psie, 7, #]& /@ slices;
Psiw8 = ShowWindow[Psie, 8, #]& /@ slices;

HideWindow[#]& /@ Psiw0;
HideWindow[#]& /@ Psiw1;
HideWindow[#]& /@ Psiw2;
HideWindow[#]& /@ Psiw3;
HideWindow[#]& /@ Psiw4;
HideWindow[#]& /@ Psiw5;
HideWindow[#]& /@ Psiw6;
HideWindow[#]& /@ Psiw7;
HideWindow[#]& /@ Psiw8;

HAMILTON OPERATOR
```

```
scalar = None; vector = Ae; domain = None;
mass = 1.; charge = 1.; units = 1.h;
He = Dirac3D[scalar, vector, domain, mass, charge, units];

TIME EVOLUTION

t = 1.h 2/3; fractal = 4; steps = 32;
TimeEvolution[He, Psie, t, fractal, steps];

VISUALIZATION

BeginMovie[#]& /@ Psiw0;
BeginMovie[#]& /@ Psiw1;
BeginMovie[#]& /@ Psiw2;
BeginMovie[#]& /@ Psiw3;
BeginMovie[#]& /@ Psiw4;
BeginMovie[#]& /@ Psiw5;
BeginMovie[#]& /@ Psiw6;
BeginMovie[#]& /@ Psiw7;
BeginMovie[#]& /@ Psiw8;

EndMovie[#]& /@ Psiw0;
EndMovie[#]& /@ Psiw1;
EndMovie[#]& /@ Psiw2;
EndMovie[#]& /@ Psiw3;
EndMovie[#]& /@ Psiw4;
EndMovie[#]& /@ Psiw5;
EndMovie[#]& /@ Psiw6;
EndMovie[#]& /@ Psiw7;
EndMovie[#]& /@ Psiw8;
```

D.4 Fraktale Graphen

D.4.1 Fractals.nb

```
FRACTALS

DEFINITION

z[m_Integer] := Join[z[m-1]/(1+Exp[-I Pi/m]), z[m-1]/(1+Exp[I Pi/m])] /; m>1
z[1] := {1.}

PLOTS

opts = {PlotJoined->True, PlotRange->{-1/16, 1/16}, AspectRatio->1/2};

ListPlot[Re[z[6]], opts];

ListPlot[Im[z[6]], opts];
```

Literaturverzeichnis

[1] H.F. Trotter, Proc. Am. Math. Soc. **10**, 545 (1959).

[2] M. Suzuki, J. Math. Phys. **32**, 400 (1991).

[3] M. Suzuki, Phys. Lett. A **146**, 319 (1990).

[4] H. De Readt, K. Michielsen, Comp. Phys. **8**, 600 (1994).

[5] M. Jursa, P. Kasperkovitz, Phys. Rev. A **47**, 3602 (1993).

[6] Y. Aharonov and D. Bohm, Phys. Rev. **115**, 485 (1959).

[7] W.F. Ames, *Numerical Methods for Partial Differential Equations* (Academic Press, New York, 1992).

[8] P. Deuflhard, A. Hohmann, *Numerische Mathematik* (De Gruyter, Berlin, 1993).

[9] A. Goldberg, H.M. Schey, J.L. Schwartz, Am. J. Phys. **35**, 177 (1967).

[10] M.D. Feit, J.A. Fleck, A. Steiger, J. Comp. Phys. **47**, 412 (1982).

[11] L.D. Landau, E.M. Lifschitz, *Quantenmechanik* (Akademie-Verlag, Berlin, 1979).

[12] D. Werner, *Funktionalanalysis* (Springer-Verlag, 1995).

[13] B. Thaller, *The Dirac Equation* (Springer-Verlag, 1992).

[14] R.S. Wolff, Comp. Phys. **6**, 421 (1992).

[15] B. Thaller, *Visualization of Complex Functions*, The Mathematica Journal **7**(2), to appear.

[16] G. Booch, *Object-Oriented Analysis and Design* (Benjamin/Cummings, 1994).

[17] B. Stroustrup, *The C++ Programming Language*, 3rd ed. (Addison-Wesley, 1997).

[18] Apple Computer, *Inside Macintosh*, (Addison-Wesley, 1992).

[19] S. Wolfram, *The Mathematica Book*, 3rd ed. (Wolfram Media/Cambridge University Press, 1996).

[20] Wolfram Research, *MathLink Reference Guide* (Wolfram Research Inc., 1993). Available as MathSource item 0204-398

[21] T. Gayley, *A MathLink Tutorial* (Wolfram Research Inc., 1993). Available as MathSource item 0206-693

[22] D.B. Wagner, The Mathematica Journal **6**(3), 44 (1996).

[23] A.S. Berdnikov, S.B. Turtia, The Mathematica Journal **6**(3), 65 (1996).