Numerische Verfahren für klassische Spinsysteme

Vergleich verschiedener Algorithmen

Diplomarbeit

Andreas Bröermann

Fachbereich Physik
Universität Osnabrück

6. August 2008

Inhaltsverzeichnis

1 Einleitung **5**

2 Grundlagen **7**

2.1 Begriffe 7
2.2 Integrable Systeme 10
2.2.1 Gleichförmige Kopplung 10
2.2.2 Konstruktionsbäume 11
2.3 Symplektische Integration durch Zerlegung in integrable Systeme 14
2.4 Integration nach Runge-Kutta 18
2.5 Projektor der Hamiltonfunktion 20

3 Realisierung als C-Programm **23**

3.1 Bedienung des Programms 23
3.2 Konzept der Implementation 24
3.3 Die Module des Programms 24
3.4 Implementierung des symplektischen Integrators 26

4 Ergebnisse **28**

4.1 Betrachtete Systeme 28
4.2 Erhaltungsgrößen 29
4.3 Projektoren 33
4.4 Euklidischer Abstand 35
4.4.1 Bewertung der Projektoren 39
4.4.2 Bewertung der Integratoren 40
4.5 Laufzeiten 42

5 Zusammenfassung und Ausblick **47**

A Drehung durch Quaternionen **49**

A.1 Definition und Einführung 49
A.2 Darstellung von Drehungen 51
A.3 Vergleich mit Drehmatrizen 53
A.3.1 Speicherbedarf 53
A.3.2 Konstruktionsaufwand 53

A.3.3 Verkettung zweier Drehungen 54
A.3.4 Drehung eines Vektors 54

B Messung der Laufzeiten **55**

C Implementierte Spinsysteme **56**
C.1 Fliege 56
C.2 Quadrat 57
C.3 Fe_{30} 59

D Quellcode des erstellten Programms **60**

Literatur **82**

Bezeichnungen

s^i_μ	bezeichnet die i-te Komponente des μ-ten Einzelspins.
$\vec{s}_\mu$	bezeichnet den dreikomponentigen Vektor $\left(s^x_\mu, s^y_\mu, s^z_\mu\right)$ des μ-ten Einzelspins.
N	bezeichnet die Anzahl der Spins im betrachteten System.
$\underline{s}^i$	bezeichnet den N-komponentigen Vektor $\left(s^i_1, s^i_2, \ldots, s^i_N\right)$ der i-ten Komponente aller Einzelspins.
$\underline{\vec{s}}$	bezeichnet den $3N$-komponentigen Vektor $\left(s^x_1, s^y_1, s^z_1, s^x_2, \ldots, s^x_N, s^y_N, s^z_N\right)$ aller Komponenten aller Einzelspins.
S^i	bezeichnet die i-te Komponente des Gesamtspins $s^i_1 + s^i_2 + \cdots + s^i_N$.
$\vec{S}$	bezeichnet den drei-komponentigen Vektor des Gesamtspins (S^x, S^y, S^z).
$\vec{\nabla}_\mu$	bezeichnet den drei-komponentigen Gradienten $\left(\frac{\partial}{\partial s^x_\mu}, \frac{\partial}{\partial s^y_\mu}, \frac{\partial}{\partial s^z_\mu}\right)$.
$\underline{\vec{\nabla}}$	bezeichnet den $3N$-komponentigen Gradienten $\left(\vec{\nabla}_1, \vec{\nabla}_2, \ldots, \vec{\nabla}_N\right)$.
$H(\underline{\vec{s}})$	bezeichnet die Hamiltonfunktion.
$J_{\mu\nu}$	bezeichnet den Eintrag in der μ-ten Zeile und der ν-ten Spalte der $N \times N$ großen symmetrischen Kopplungsmatrix J.
ϵ_{ijk}	ist das Levi-Civita-Symbol.
$\mathbb{D}(\vec{\omega}, \Delta t)$	bezeichnet einen Drehoperator. Auf einen dreidimensionalen Vektor $\vec{s}$ angewendet, wird dieser um die Achse $\vec{\omega}$ um den Winkel $\lvert\vec{\omega}\rvert \Delta t$ gedreht.

1 Einleitung

Magnetische Materialien begegnen uns im täglichen Leben meist als Festkörper mit Teilchenzahlen von einigen hundert Trilliarden (10^{23}). Lange Zeit konnten nur solche makroskopischen Systeme mit einer sehr großen Anzahl von magnetischen Momenten betrachtet werden. Seit einigen Jahren ist es nun möglich, gezielt magnetische Moleküle herzustellen. Das ermöglicht es, unser Verständnis vom Magnetismus auch an kleineren Systemen zu testen, in denen nur 2 bis 30 magnetische Momente miteinander wechselwirken.

Diese neuen Moleküle sind derart beschaffen, dass Wechselwirkungen zwischen benachbarten Molekülen vernachlässigt werden können. Daher werden bei der Untersuchung von makroskopischen Proben dieser Moleküle die Eigenschaften des einzelnen Moleküls erfasst. Als dominierende Methode zur Untersuchung der Dynamik dieser Systeme hat sich die Neutronenstreuung etabliert. [1][2][3]

Um theoretische Vorhersagen machen zu können und diese mit den experimentellen Daten zu vergleichen, sind numerische Verfahren unerlässlich.

Neben quantenmechanischen Ansätzen ist auch eine klassische Betrachtungsweise möglich. Dabei werden die Spins einzelner Atome und Ionen als klassische magnetische Momente behandelt. Die Wechselwirkung zwischen den Spins wird durch die Struktur der Moleküle vorgegeben. Die klassische Betrachtung ist vor allem dann sinnvoll, wenn die quantenmechanische Simulation die Leistungsfähigkeit heutiger Computersysteme übersteigt. Für einige quantenmechanische Systeme lässt sich zeigen, dass sich ihre Eigenschaften mit zunehmendem Betrag des Spins denen ihres klassischen Gegenstücks annähern [4].

Weil die Bewegungsgleichungen der klassischen Spinsysteme im Allgemeinen nicht integrabel sind, kommen verschiedene Näherungsverfahren zur Anwendung. Wohl am verbreitetsten sind Verfahren nach Runge-Kutta (wie in [3]), aber es wurden auch symplektische Verfahren vorgeschlagen ([1] und [5]).

Ziel dieser Diplomarbeit ist es, verschiedene Verfahren zu betrachten und ihre Leistungsfähigkeit zu vergleichen. Insbesondere wurde versucht, den symplektischen Integrator effizient zu implementieren. Bisher existierte nur eine Implementation in dem Computer-Algebra-System Mathematica [1]. Im Rahmen dieser Arbeit wurde ein unabhängiges Programm in C geschrieben, mit dem sich verschiedene Spinsysteme nach mehreren Methoden symplektisch integrieren lassen. Eine Besonderheit in dieser Implementation ist, dass die Drehung mit Hilfe von Quaternionen durchgeführt wird.

Darüber hinaus wurden Projektoren implementiert, mit denen sich bekannte Erhaltungsgrößen der Spinsysteme konstant halten lassen. Schließlich wurde auch ein Runge-Kutta-Verfahren realisiert, das auf die gleichen Daten wie das symplektische Verfahren angewendet werden kann. Damit ist es möglich, die Laufzeiten und die Ergebnisse der beiden Verfahren unter gleichen Bedingungen miteinander zu vergleichen.

Im Abschnitt 2 werden die theoretischen Grundlagen der verwendeten Algorithmen vorgestellt. Der dritte Abschnitt erklärt, wie diese Algorithmen in der Programmiersprache C implementiert worden sind. Im Abschnitt 4 werden schließlich die Ergebnisse präsentiert. Nach der Zusammenfassung werden im Anhang unter anderem die Drehung durch Quaternionen erläutert, die Konstruktionsbäume einiger Spinsysteme vorgestellt und der Quellcode des geschriebenen Programms präsentiert.

2 Grundlagen

Im ersten Teil dieses Abschnittes werden einige grundlegende Begriffe definiert. Der Teil 2.2 erklärt den exakten Integrator für Heisenberg-integrable Spinsysteme, die über einen Konstruktionsbaum verfügen. In den folgenden beiden Teilen werden der symplektische Integrator und das Runge-Kutta-Verfahren beschrieben. Im letzten Teil wird der Projektor der Hamiltonfunktion hergeleitet.

2.1 Begriffe

Die in dieser Diplomarbeit betrachteten Systeme sind klassische Spinsysteme, deren Hamiltonfunktion $H(\underline{\vec{s}})$ vom Heisenberg-Typ ist. Das heißt, dass sich $H(\underline{\vec{s}})$ als

$$H(\underline{\vec{s}}) = \frac{1}{2} \sum_{\mu,\nu} J_{\mu\nu} \; \vec{s}_\mu \cdot \vec{s}_\nu = \sum_{\mu<\nu} J_{\mu\nu} \; \vec{s}_\mu \cdot \vec{s}_\nu \tag{2.1}$$

schreiben lässt. Dabei ist die Kopplungsmatrix J symmetrisch und die diagonalen Einträge verschwinden.

$$J_{\mu\nu} = J_{\nu\mu} \;\; , \;\; J_{\mu\mu} = 0 \tag{2.2}$$

Mit dem $3N$-dimensionalen Vektor $\underline{\vec{s}}$ ist der Zustand des Systems zu einem bestimmten Zeitpunkt eindeutig definiert. Eine weitere Eigenschaft der betrachteten Systeme ist, dass die einzelnen Spins konstant den Betrag $|\vec{s}_\mu| = 1$ aufweisen. Damit ist der Raum möglicher Zustände tatsächlich nur $2N$-dimensional. Dieser Zustandsraum ist ein *Phasenraum* $\mathcal{P}$, auf dem sich eine Poisson-Klammer definieren lässt. [1]

Die kanonischen Koordinaten von $\mathcal{P}$ entsprechen den beiden Zylinder-Koordinaten ϕ_μ und z_μ der Einzelspins. Jedem Phasenraumelement $\xi \in \mathcal{P}$ lässt sich genau ein Zustand $\underline{\vec{s}} \in \mathbb{R}^{3N}$ mit $|\vec{s}_\mu| = 1$ für alle μ zuordnen. Zwischen den kartesischen und den kanonischen Koordinaten gilt die Beziehung

$$\vec{s}_\mu = \begin{pmatrix} s_\mu^x \\ s_\mu^y \\ s_\mu^z \end{pmatrix} = \begin{pmatrix} \sqrt{1 - z_\mu^2}\cos\phi_\mu \\ \sqrt{1 - z_\mu^2}\sin\phi_\mu \\ z_\mu \end{pmatrix} \quad . \tag{2.3}$$

Die Poisson-Klammer hat mit den kanonischen Koordinaten $p_\mu = z_\mu$ und $q_\mu = \phi_\mu$ für Funktionen auf dem Phasenraum $f, g : \mathcal{P} \to \mathbb{R}$ die bekannte Form

$$\{f(\xi), g(\xi)\} = \sum_{\mu=1}^{N} \frac{\partial f}{\partial q_\mu}\frac{\partial g}{\partial p_\mu} - \frac{\partial f}{\partial p_\mu}\frac{\partial g}{\partial q_\mu} \quad . \tag{2.4}$$

In unseren Algorithmen soll der Spin in kartesischen Koordinaten dargestellt werden. Eine Umformung der Poisson-Klammer ergibt [6]

$$\{f(\xi), g(\xi)\} = \sum_{\mu=1}^{N} \epsilon_{ijk} \frac{\partial f}{\partial s_\mu^i}\frac{\partial g}{\partial s_\mu^j} s_\mu^k \quad , \tag{2.5}$$

dabei wird über die Indizes i, j und k gemäß der *Einsteinschen Summenkonvention* summiert.

Gemäß dem *Hamilton-Formalismus* lässt sich die zeitliche Ableitung eines Einzelspins schreiben als

$$\frac{d\vec{s}_\lambda}{dt} = \frac{\partial \vec{s}_\lambda}{\partial t} + \{\vec{s}_\lambda, H\} \quad . \tag{2.6}$$

Da die einzelnen Spins nicht explizit von der Zeit abhängen, gilt $\frac{\partial \vec{s}_\lambda}{\partial t} = 0$. Die Berechnung der Poisson-Klammer gemäß (2.5) ergibt ein Kreuzprodukt aus der Ableitung der Hamiltonfunktion nach dem Einzelspin und dem Einzelspin selbst.

$$\dot{\vec{s}}_\lambda = \{\vec{s}_\lambda, H\} = \vec{\nabla}_\lambda H \times \vec{s}_\lambda \quad . \tag{2.7}$$

Diese Differentialgleichung ist zugleich die zu lösende Bewegungsgleichung. Die Ableitung der Hamiltonfunktion ist ein lokales Feld, um das der Spin $\vec{s}_\lambda$ präzediert. Wenn die Hamiltonfunktion vom Heisenberg-Typ ist und kein äußeres Feld vorliegt, dann ist das lokale Feld an der Stelle des Spins $\vec{s}_\lambda$ die durch die Kopplungsmatrix J gewichtete Summe der Nachbarspins.

$$\vec{\nabla}_\lambda H = \sum_{\mu=1}^{N} J_{\mu\lambda}\, \vec{s}_\mu \tag{2.8}$$

Unter einem *Integrator* verstehen wir im Rahmen dieser Arbeit einen Operator

$$F_{\Delta t} : \mathbb{R}^{3N} \to \mathbb{R}^{3N} \quad ,$$

der einem Zustand $\underline{\vec{s}}$ zum Zeitpunkt t_0 den Folgezustand zum Zeitpunkt $t_0 + \Delta t$ zuordnet. Im-

plementiert wird der Integrator als Algorithmus mit dem Ziel, die Bewegungsgleichung möglichst genau zu lösen. Da die meisten Systeme leider nicht integrabel sind, ist das Ergebnis mit einem Fehler behaftet, der von der Ordnung k des verwendeten Algorithmus abhängt.

$$F_{\Delta t}(\underline{\vec{s}}(t_0)) = \underline{\vec{s}}(t_0 + \Delta t) + \mathcal{O}(\Delta t^{k+1}) \tag{2.9}$$

2.2 Integrable Systeme

Für einige Spinsysteme kann die Zeitentwicklung exakt angegeben werden. Im Falle von *Heisenberg-integrablen* Systemen, für die ein *Konstruktionsbaum* angegeben werden kann, ist der Integrator identisch mit einem Produkt von mehreren Drehoperatoren. [1]

Einfachstes Beispiel hierfür ist ein Spin-Dimer, der nur aus zwei Spins besteht, die über die Kopplungskonstante J_{12} verbunden sind. Die Hamiltonfunktion vereinfacht sich hier zu

$$H = J_{12}\vec{s}_1 \cdot \vec{s}_2 \quad . \tag{2.10}$$

Die Bewegungsgleichung für den ersten Spin lautet

$$\dot{\vec{s}_1} = \vec{\nabla}_1 H \times \vec{s}_1 = J_{12}\, \vec{s}_2 \times \vec{s}_1 = J_{12}\, \vec{S} \times \vec{s}_1 \quad . \tag{2.11}$$

Für $\vec{s}_1$ und entsprechend für $\vec{s}_2$ gilt, dass sie sich um den Gesamtspin $\vec{S}$ mit der Winkelgeschwindigkeit $\omega = J_{12}|\vec{S}|$ drehen:

$$\vec{s_\mu}(t_0 + \Delta t) = \mathbb{D}(J_{12}\vec{S}, \Delta t)\vec{s_\mu}(t_0) \tag{2.12}$$

2.2.1 Gleichförmige Kopplung

Werden zwei Heisenberg-integrable Systeme H_A und H_B gleichförmig miteinander gekoppelt, so ist das resultierende System H ebenfalls Heisenberg-integrabel. Gleichförmige Kopplung bedeutet, dass jeder Spin aus H_A mit jedem Spin aus H_B mit der selben Kopplungskonstante c_{AB} koppelt.

Die Mengen M_A und M_B seien so gegeben, dass $\mu \in M_A$ für alle $\vec{s}_\mu$ die zum System H_A gehören, und $\nu \in M_B$ für alle $\vec{s}_\nu$ die zum System H_B gehören. Die Kopplungsmatrizen der beiden Systeme seien J^A und J^B. Eine gleichförmige Kopplung zwischen den Systemen liegt genau dann vor, wenn für die Einträge der Kopplungsmatrix J des Systems H gilt:

$$J_{\mu\nu} = \begin{cases} J^A_{\mu\nu} & \text{wenn } \mu \in M_A \wedge \nu \in M_A \\ J^B_{\mu\nu} & \text{wenn } \mu \in M_B \wedge \nu \in M_B \\ c_{AB} & \text{wenn } \mu \in M_A \wedge \nu \in M_B \text{ oder } \mu \in M_B \wedge \nu \in M_A \end{cases} \tag{2.13}$$

Der Gesamtspin der Teilsysteme H_A und H_B ist die Summe aller Einzelspins aus dem jeweiligen Teilsystem:

$$\vec{S}_A = \sum_{\mu \in A} \vec{s}_\mu \quad \text{bzw.} \quad \vec{S}_B = \sum_{\mu \in B} \vec{s}_\mu \quad . \tag{2.14}$$

Betrachten wir nun die Bewegungsgleichung von $\vec{S}_A$:

$$\begin{aligned}
\dot{\vec{S}}_A &= \sum_{\mu \in A} \dot{\vec{s}}_\mu = \sum_{\mu \in A} \{\vec{s}_\mu, H\} \\
&= \sum_{\mu \in A} \vec{\nabla}_\mu H \times \vec{s}_\mu \\
&= \sum_{\mu \in A} \sum_{\nu=1}^{N} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu \\
&= \sum_{\mu \in A} \sum_{\nu \in A} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu + \sum_{\mu \in A} \sum_{\nu \in B} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu \\
&= \sum_{(\mu<\nu) \in A} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu + \sum_{(\nu<\mu) \in A} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu + c_{AB} \sum_{\mu \in A} \sum_{\nu \in B} \vec{s}_\nu \times \vec{s}_\mu \\
&= \sum_{(\mu<\nu) \in A} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu - \sum_{(\mu<\nu) \in A} J_{\mu\nu}\, \vec{s}_\nu \times \vec{s}_\mu + c_{AB} \sum_{\nu \in B} \vec{s}_\nu \times \vec{S}_A \\
&= c_{AB}\, \vec{S}_B \times \vec{S}_A \\
&= c_{AB}\, \vec{S} \times \vec{S}_A
\end{aligned} \tag{2.15}$$

Die Berechnung von $\dot{\vec{S}}_B$ verläuft entsprechend. Analog zu den Einzelspins des Dimers präzedieren $\vec{S}_A$ und $\vec{S}_B$ offenbar um den Gesamtspin $\vec{S} = \vec{S}_A + \vec{S}_B$ mit der Winkelgeschwindigkeit $\omega = c_{AB}|\vec{S}|$. Die Beträge $|\vec{S}_A|$ und $|\vec{S}_B|$ bleiben dabei erhalten.

$$\vec{S}_A(t_0 + \Delta t) = \mathbb{D}(c_{AB}\vec{S}, \Delta t)\vec{S}_A(t_0) \quad \text{bzw.} \quad \vec{S}_B(t_0 + \Delta t) = \mathbb{D}(c_{AB}\vec{S}, \Delta t)\vec{S}_B(t_0) \tag{2.16}$$

2.2.2 Konstruktionsbäume

Wenn ein Konstruktionsbaum $\mathcal{B}_H$ zu einem Heisenberg-System H angegeben werden kann, beschreibt dieser das System H eindeutig. Umgekehrt können zu einem System oft viele verschiedene Konstruktionsbäume angegeben werden. Um einen exakten Integrator angeben zu können, genügt es, nur einen Konstruktionsbaum zu finden. Allerdings gibt es auch integrierbare Heisenberg-Systeme, für die kein Konstruktionsbaum angegeben werden kann, wie das allgemeine Spin-Dreieck. [1]

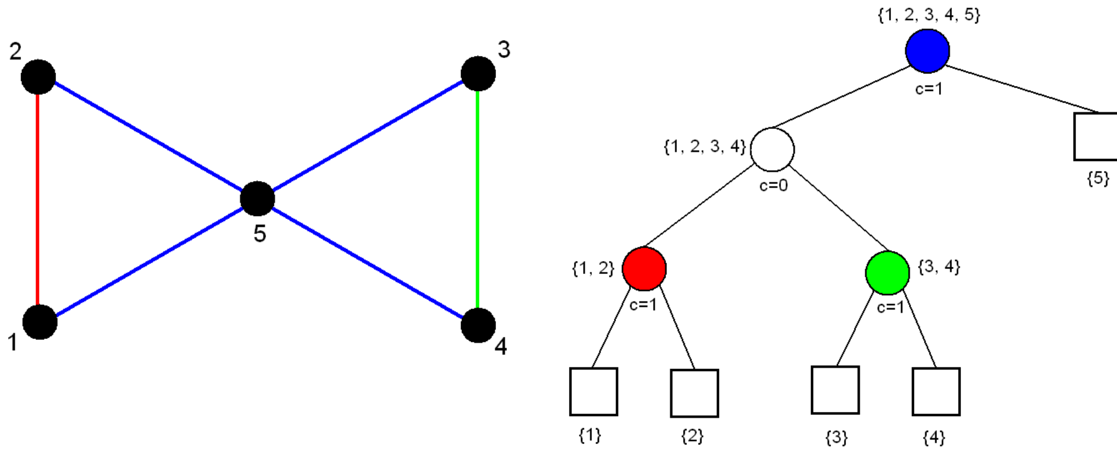


Abbildung 1: Links ein Spinsystem in Form einer „Fliege“, rechts der dazugehörige Konstruktionsbaum. Die Blätter sind als Quadrate dargestellt, die inneren Knoten als Kreise. Die Farbe eines Kreises zeigt an, welche Kanten des Systems durch den Knoten beschrieben werden. Der weiße Knoten entspricht der Kopplung der Stärke Null zwischen den Dimeren $\{1, 2\}$ und $\{3, 4\}$.

Der Konstruktionsbaum ist ein *strikt binärer* Baum. Das heißt, jedes seiner Elemente ist entweder ein *innerer Knoten* mit genau zwei *Söhnen*, oder ein *Blatt*, also ein Knoten ohne Söhne. Bis auf die *Wurzel* des Baumes hat jeder Knoten genau einen *Vater*.

Jedem Blatt von $\mathcal{B}_H$ ist ein Einzelspin des Systems H zugeordnet. Bei einem System aus N Spins hat der zugehörige Baum somit N Blätter und $N - 1$ innere Knoten inklusive der Wurzel. Einem inneren Knoten K_i ist eine Menge M_i von mehreren Einzelspins zugeordnet, und zwar die Vereinigungsmenge der Mengen seiner beiden Söhne. Desweiteren ist jedem inneren Knoten eine Kopplungskonstante c_i zugeordnet.

Haben zwei Knoten K_A und K_B einen gemeinsamen Vater K_{AB} , so bedeutet dies, dass die Spins aus M_A mit den Spins aus M_B gleichförmig mit der Kopplungskonstante c_{AB} koppeln. Jeder Teilbaum von $\mathcal{B}_H$ beschreibt also ein Teilsystem von H. Das System H selbst entsteht durch sukzessive gleichförmige Kopplung von kleineren Spinsystemen.

Nun kann man in $\mathcal{B}_H$ zu jedem Einzelspin $\vec{s}_\mu$ einen *Konstruktionspfad* $\mathcal{P}_\mu$ bilden, der von dem ihm zugeordneten Blatt bis zur Wurzel verläuft. Die Länge L des Pfades gibt die Anzahl der Knoten auf $\mathcal{P}_\mu$ an. Die Knoten auf dem Pfad wollen wir K_i mit i ganzzahlig von 1 bis L nennen, wobei K_L das Blatt und K_1 die Wurzel bezeichne. Dabei sei K_{i-1} stets Vaterknoten von K_i. Mit M_i, der zu K_i gehörigen Menge von Einzelspins, bezeichne $\vec{S}_i$ den Gesamtspin des Teilsystems $\vec{S}_i = \sum_{\nu \in M_i} \vec{s}_\nu$. c_i sei die zum Knoten K_i gehörige Kopplungskonstante.

Mit diesen Bezeichnungen lässt sich die Zeitentwicklung eines Einzelspins durch ein Produkt aus mehreren Drehoperatoren beschreiben. Als Schreibweise für ein Produkt aus mehreren Operatoren definieren wir

$$\prod_{i=a}^{b} \mathbb{A}_i = \begin{cases} \mathbb{A}_a \mathbb{A}_{a+1} \cdots \mathbb{A}_{b-1} \mathbb{A}_b & \text{für } a \leq b \\ \mathbb{I} & \text{für } a > b \end{cases} \quad . \tag{2.17}$$

Dabei ist $\mathbb{I}$ die Identität. Für die Zeitentwicklung eines Einzelspins gilt nun

$$\vec{s}_\mu(t_0 + \Delta t) = \mathbb{D}(c_1 \vec{S}_1(t_0), \Delta t) \left(\prod_{i=2}^{L-1} \mathbb{D}\left((c_i - c_{i-1}) \vec{S}_i(t_0), \Delta t \right) \right) \vec{s}_\mu(t_0) \quad . \tag{2.18}$$

Hierbei ist zu beachten, dass die Größen L, c_i, c_{i-1} und $\vec{S}_i$ Eigenschaften des Pfades $\mathcal{P}_\mu$ sind und damit von μ abhängen. Definieren wir nun die *reduzierte Kopplungskonstante*

$$\tilde{c}_i = \begin{cases} c_i - c_{i-1} & \text{für } i \geq 2 \\ c_i & \text{für } i = 1 \end{cases} \quad , \tag{2.19}$$

so können wir (2.18) komprimiert schreiben als

$$\vec{s}_\mu(t_0 + \Delta t) = \left(\prod_{i=1}^{L-1} \mathbb{D}\left(\tilde{c}_i \vec{S}_i(t_0), \Delta t \right) \right) \vec{s}_\mu(t_0) \quad . \tag{2.20}$$

Da die Pfadlänge für die einzelnen Spins verschieden sein kann, kommen in dem selben Spinsystem Zeitentwicklungen mit verschieden langen Produkten von Drehoperatoren vor. Zum Beispiel lauten die Zeitentwicklungen für den ersten und den fünften Spin des Systems aus Abbildung 1

$$\begin{aligned} \vec{s}_1(t_0 + \Delta t) &= \mathbb{D}\Big(\vec{S}(t_0), \Delta t \Big)\, \mathbb{D}\Big(-\left(\vec{S}(t_0) - \vec{s}_5(t_0) \right), \Delta t \Big)\, \mathbb{D}\Big(\vec{s}_1(t_0) + \vec{s}_2(t_0), \Delta t \Big)\, \vec{s}_1(t_0) \\ \vec{s}_5(t_0 + \Delta t) &= \mathbb{D}\Big(\vec{S}(t_0), \Delta t \Big)\, \vec{s}_5(t_0) \quad . \end{aligned}$$

Die Korrektheit dieses Integrators wurde in [1] durch vollständige Induktion bewiesen.

2.3 Symplektische Integration durch Zerlegung in integrable Systeme

Jedes nicht-Heisenberg-integrable System vom Heisenberg-Typ lässt sich in eine Summe von Heisenberg-integrablen Systemen zerlegen, da jeder Term $J_{\mu\nu}\ \vec{s}_\mu \cdot \vec{s}_\nu$ der Hamiltonfunktion $H(\underline{\vec{s}})$ einen einfachen Dimer beschreibt. Für viele Systeme ist sogar eine Zerlegung in nur zwei Heisenberg-integrable Summanden $H_A + H_B = H$ möglich. Jeder Summand H_A und H_B beschreibt ein System mit den selben Einzelspins wie das Gesamtsystem H, aber mit nur einem Teil der Kopplungen.

Da die Summanden H_A und H_B integrabel sind, kann für diese Teilsysteme eine exakte Zeitentwicklung angegeben werden. Mit dem *symplektischen Integrator* soll die tatsächliche Zeitentwicklung des Gesamtsystems H dadurch genähert werden, dass in mehreren Teilschritten die jeweilige exakte Zeitentwicklung der beiden Teilsysteme durch Drehmatrizen nacheinander ausgeführt wird. Da jeder Teilschritt ein exakter Integrationsschritt ist, ist die Symplektizität garantiert. Aus dem selben Grund bleiben auch die Komponenten des Gesamtspins stets erhalten. Andere Erhaltungsgrößen, wie z. B. die Gesamtenergie, bleiben allerdings nicht exakt erhalten, jedoch zeigt sich, dass sie um den wahren Wert nur schwanken, anstatt sich immer weiter vom Anfangswert zu entfernen.

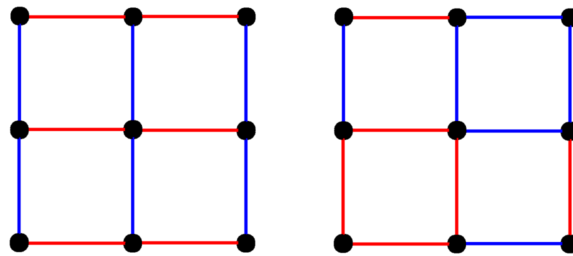


Abbildung 2: Zwei mögliche Zerlegungen des 3×3-Gitters.

In Abbildung 2 sind als Beispiel mögliche Zerlegungen des 3×3-Gitters in zwei Teilsysteme angegeben. Für das Gesamtsystem lässt sich kein Konstruktionsbaum finden. Betrachtet man nur die blau gezeichneten Kopplungen, so liegt im linken Beispiel ein System aus drei ungekoppelten Dreierketten vor und im rechten Beispiel ein System aus einem Spin-Quadrat und zwei Dimeren. Für diese Teilsysteme kann ein Konstruktionsbaum und damit ein exakter Integrator angegeben werden. Dasselbe gilt für die rot gezeichneten Teilsysteme.

Wenn eine geeignete Zerlegung $H_A + H_B = H$ in zwei Summanden vorliegt und für beide Teil-Systeme ein Konstruktionsbaum gefunden werden kann, dann sei A der exakte Integrator des Systems H_A und B der exakte Integrator der Systems H_B. A und B sind gemäß (2.18) durch die

jeweiligen Konstruktionsbäume gegeben. Der symplektische Integrator F ist von der Form [1]

$$F_{\Delta t} = \prod_{i=1}^{m} A_{a_i \Delta t} B_{b_i \Delta t} \quad . \tag{2.21}$$

Das heißt, man erhält den zu berechnenden Zustand $F_{\Delta t}\vec{\underline{s}}$, indem auf den Ausgangszustand $\vec{\underline{s}}$ zunächst im ersten Teilschritt der Integrator B angewendet wird. Die Schrittweite ist dabei $b_i \Delta t$. Auf diesen errechneten Zustand $B_{b_i \Delta t}\vec{\underline{s}}$ wird dann der Integrator A mit der Schrittweite $a_i \Delta t$ angewendet. Nach diesem Muster werden für einen Integrationsschritt also $2m$ Teilschritte berechnet. Der Parameter m und die Koeffizienten a_i und b_i werden durch die verwendete *Zerlegung* festgelegt.

Um die Koeffizienten a_i und b_i zu bestimmen, werden die Integratoren $A_{a_i \Delta t}$ als Exponentialoperatoren der Form $e^{\mathbb{A} a_i \Delta t}$ aufgefasst [7]. In dieser Darstellung lässt sich auch der exakte Integrator des Gesamtsystems H formal schreiben als $e^{(\mathbb{A} + \mathbb{B})\Delta t}$ [1][8]. Ist F von k-ter Ordnung, so gilt

$$e^{(\mathbb{A} + \mathbb{B})\Delta t} = F_{\Delta t} + \mathcal{O}(\Delta t^{k+1}) = \prod_{i=1}^{m} e^{\mathbb{A} a_i \Delta t} e^{\mathbb{B} b_i \Delta t} + \mathcal{O}(\Delta t^{k+1}) \quad . \tag{2.22}$$

Wird nun bei gegebenem m auf beiden Seiten der Gleichung für die Exponentialfunktionen die Reihenentwicklung durchgeführt, so ergibt sich eine Gleichung für jede Ordnung von Δt. Die Koeffizienten a_i und b_i werden so gewählt, dass sie diese Gleichungen bis zu einer bestimmten Ordnung k lösen. Da die Gleichungen für die Ordnung $k+1$ nicht mehr gelöst werden, ist der Fehler von der Ordnung $\mathcal{O}(\Delta t^{k+1})$.

Für den einfachsten Fall mit $m = 1$ gilt auf der linken Seite

$$\begin{aligned} e^{(\mathbb{A} + \mathbb{B})\Delta t} &= 1 + (\mathbb{A} + \mathbb{B})\Delta t + \frac{(\mathbb{A} + \mathbb{B})^2}{2}(\Delta t)^2 + \mathcal{O}(\Delta t^3) \\ &= 1 + (\mathbb{A} + \mathbb{B})\Delta t + \frac{\mathbb{A}^2 + \mathbb{B}^2 + \mathbb{A}\mathbb{B} + \mathbb{B}\mathbb{A}}{2}(\Delta t)^2 + \mathcal{O}(\Delta t^3) \quad , \end{aligned} \tag{2.23}$$

und auf der rechten Seite

$$\begin{aligned} & e^{\mathbb{A} a_1 \Delta t} e^{\mathbb{B} b_1 \Delta t} \\ &= \left(1 + \mathbb{A} a_1 \Delta t + \frac{(\mathbb{A} a_1)^2}{2}(\Delta t)^2\right)\left(1 + \mathbb{B} b_1 \Delta t + \frac{(\mathbb{B} b_1)^2}{2}(\Delta t)^2\right) + \mathcal{O}(\Delta t^3) \\ &= 1 + \mathbb{A} a_1 \Delta t + \frac{(\mathbb{A} a_1)^2}{2}(\Delta t)^2 + \mathbb{B} b_1 \Delta t + \mathbb{A} a_1 \Delta t \mathbb{B} b_1 \Delta t + \frac{(\mathbb{B} b_1)^2}{2}(\Delta t)^2 + \mathcal{O}(\Delta t^3) \\ &= 1 + (a_1 \mathbb{A} + b_1 \mathbb{B})\Delta t + \frac{a_1^2 \mathbb{A}^2 + b_1^2 \mathbb{B}^2 + 2 a_1 b_1 \mathbb{A}\mathbb{B}}{2}(\Delta t)^2 + \mathcal{O}(\Delta t^3) \quad . \end{aligned} \tag{2.24}$$

Für die erste Ordnung von Δt erhalten wir die Gleichung

$$\mathbb{A} + \mathbb{B} = a_1 \mathbb{A} + b_1 \mathbb{B} \quad . \tag{2.25}$$

Soll diese Gleichung für alle $\mathbb{A}$ und $\mathbb{B}$ gelten, so ist die eindeutige Lösung $a_1 = b_1 = 1$. Die Gleichung für die zweite Ordnung von Δt lautet

$$\frac{\mathbb{A}^2 + \mathbb{B}^2 + \mathbb{A}\mathbb{B} + \mathbb{B}\mathbb{A}}{2} = \frac{a_1^2 \mathbb{A}^2 + b_1^2 \mathbb{B}^2 + 2a_1 b_1 \mathbb{A}\mathbb{B}}{2} \quad . \tag{2.26}$$

Da im Allgemeinen $\mathbb{A}\mathbb{B} \neq \mathbb{B}\mathbb{A}$ ist, existiert für diese Gleichung keine allgemeine Lösung.

Der gefundene Integrator $F_{\Delta t} = A_{\Delta t} B_{\Delta t}$ ist somit erster Ordnung. Hierbei handelt es sich um die *Suzuki-Trotter-Zerlegung* erster Ordnung (ST1) [9]. Mit höheren Werten für m lassen sich andere Zerlegungen höherer Ordnung finden.

Die in dieser Arbeit betrachteten Zerlegungen umfassen die Suzuki-Trotter-Zerlegung zweiter und vierter Ordnung (ST2 [10] und ST4 [11]), sowie die gewöhnliche Zerlegung nach Forrest-Ruth (FR4) [12] und eine optimierte Variante der Forrest-Ruth-Zerlegung (OFR4) [13]. Die Koeffizienten dieser Zerlegungen sind in Tabelle 1 angegeben.

Zerlegung	Koeffizienten		Hilfsgrößen
ST2	$a_1 = \frac{1}{2}$	$b_1 = 1$	
	$a_2 = \frac{1}{2}$	$b_2 = 0$	
ST4	$a_1 = \frac{p}{2}$	$b_1 = p$	$p = \frac{1}{4-4^{1/3}}$
	$a_2 = p$	$b_2 = p$	
	$a_3 = \frac{1-3p}{2}$	$b_3 = 1 - 4p$	
	$a_4 = \frac{1-3p}{2}$	$b_4 = p$	
	$a_5 = p$	$b_5 = p$	
	$a_6 = \frac{p}{2}$	$b_6 = 0$	
FR4	$a_1 = \frac{\theta}{2}$	$b_1 = \theta$	$\theta = \frac{1}{2-2^{1/3}}$
	$a_2 = \frac{1-\theta}{2}$	$b_2 = 1 - 2\theta$	
	$a_3 = \frac{1-\theta}{2}$	$b_3 = \theta$	
	$a_4 = \frac{\theta}{2}$	$b_4 = 0$	
OFR4	$a_1 = \xi$	$b_1 = \frac{1-2\lambda}{2}$	$\lambda = -0{,}09156203$
	$a_2 = \chi$	$b_2 = \lambda$	$\xi = 0{,}17208656$
	$a_3 = 1 - 2(\xi + \chi)$	$b_3 = \lambda$	$\chi = -0{,}16162176$
	$a_4 = \chi$	$b_4 = \frac{1-2\lambda}{2}$	
	$a_5 = \xi$	$b_5 = 0$	

Tabelle 1: Die verschiedenen betrachteten Zerlegungen. Daten der Tabelle übernommen aus [5].

2.4 Integration nach Runge-Kutta

Ein Standardverfahren für die Lösung von solchen Anfangswertproblemen wie die betrachteten Spinsysteme ist das klassische Runge-Kutta-Verfahren vierter Ordnung [14]. Ausgangspunkt ist auch hier die zu lösende Differentialgleichung (2.7). Als Hilfsfunktion wird in diesem Algorithmus die zeitliche Ableitung $\dot{\vec{s}}_\lambda$ direkt ausgerechnet.

$$\dot{\vec{s}}_\lambda = \{\vec{s}_\lambda, H\} = \vec{\nabla}_\lambda H \times \vec{s}_\lambda = \sum_{\mu=1}^{N} J_{\lambda\mu}\, \vec{s}_\mu \times \vec{s}_\lambda \tag{2.27}$$

Der Einzelspin $\vec{s}_\lambda$ zum Zeitpunkt $t_0 + \Delta t$ wird auf Grundlage seines gegebenen Wertes zum Zeitpunkt t_0 wie folgt berechnet:

$$\vec{s}_\lambda(t_0 + \Delta t) = \vec{s}_\lambda(t_0) + \frac{\Delta t}{6}\left(\dot{\vec{s}}_0 + 2\dot{\vec{s}}_A + 2\dot{\vec{s}}_B + \dot{\vec{s}}_C\right) \tag{2.28}$$

Die vier Summanden in der rechten Klammer ergeben sich als zeitliche Ableitung gemäß (2.27) aus den Hilfsgrößen

$$\vec{s}_0 = \vec{s}_\lambda(t_0) \tag{2.29a}$$

$$\vec{s}_A = \vec{s}_0 + \frac{1}{2}\Delta t \dot{\vec{s}}_0 \tag{2.29b}$$

$$\vec{s}_B = \vec{s}_0 + \frac{1}{2}\Delta t \dot{\vec{s}}_A \tag{2.29c}$$

$$\vec{s}_C = \vec{s}_0 + \Delta t \dot{\vec{s}}_B \quad . \tag{2.29d}$$

Der Fehler ist bei diesem Verfahren von der Ordnung $\mathcal{O}(\Delta t^5)$. Bei der Diskussion der Ergebnisse in Abschnitt 4 wird dieser Integrator auch abkürzend RK4 genannt.

Wie bei der Betrachtung der Ergebnisse in Abschnitt 4 gezeigt wird, sind die Beträge der Einzelspins und die Hamiltonfunktion mit dem Runge-Kutta-Integrator nicht erhalten sondern weisen eine Drift auf.

Für den Gesamtspin können wir jedoch zeigen, dass dieser auch mit dem RK4-Integrator exakt erhalten bleibt. Dazu betrachten wir die Summen aus den zeitliche Ableitung der Hilfsgrößen $\vec{s}_0$ bis $\vec{s}_C$. Da es sich bei der Hilfsgröße $\vec{s}_0$ lediglich um die Einzelspins zum Zeitpunkt t_0 ist klar, dass hier

die Summe der Ableitungen verschwindet. Das wird durch die Rechnung

$$\begin{aligned}
\sum_{\lambda=1}^{N} \dot{\vec{s}}_0 &= \sum_{\lambda=1}^{N} \dot{\vec{s}}_\lambda(t_0) \\
&= \sum_{\lambda=1}^{N} \sum_{\mu=1}^{N} J_{\lambda\mu}\, \vec{s}_\mu \times \vec{s}_\lambda \\
&= \sum_{\lambda<\mu} J_{\lambda\mu}\, \vec{s}_\mu \times \vec{s}_\lambda + \sum_{\mu<\lambda} J_{\lambda\mu}\, \vec{s}_\mu \times \vec{s}_\lambda \\
&= \sum_{\lambda<\mu} J_{\lambda\mu}\, \vec{s}_\mu \times \vec{s}_\lambda + \sum_{\lambda<\mu} J_{\lambda\mu}\, \vec{s}_\lambda \times \vec{s}_\mu \\
&= 0
\end{aligned} \tag{2.30}$$

bestätigt. Für die entsprechende Summe der Hilfsgröße $\vec{s}_A$ ergibt sich daher

$$\begin{aligned}
\sum_{\lambda=1}^{N} \dot{\vec{s}}_A &= \frac{d}{dt}\left(\sum_{\lambda=1}^{N} \vec{s}_A\right) \\
&= \frac{d}{dt}\left(\sum_{\lambda=1}^{N} \vec{s}_0 + \frac{1}{2}\Delta t \sum_{\lambda=1}^{N} \dot{\vec{s}}_0\right) \\
&= \frac{d}{dt}\left(\sum_{\lambda=1}^{N} \vec{s}_0\right) \\
&= \sum_{\lambda=1}^{N} \dot{\vec{s}}_0 \\
&= 0 \quad .
\end{aligned} \tag{2.31}$$

Auf die gleiche Weise kann gezeigt werden, dass auch die entsprechenden Summen der Hilfsgrößen $\vec{s}_B$ und $\vec{s}_C$ verschwinden. Der Gesamtspin zum Zeitpunkt $t_0 + \Delta t$ ist daher identisch mit dem Gesamtspin zum Zeitpunkt t_0, das heißt

$$\begin{aligned}
\sum_{\lambda=1}^{N} \vec{s}_\lambda(t_0 + \Delta t) &= \sum_{\lambda=1}^{N} \vec{s_\lambda}(t_0) + \frac{\Delta t}{6} \sum_{\lambda=1}^{N} \left(\dot{\vec{s}}_0 + 2\dot{\vec{s}}_A + 2\dot{\vec{s}}_B + \dot{\vec{s}}_C\right) \\
&= \sum_{\lambda=1}^{N} \vec{s_\lambda}(t_0) \quad .
\end{aligned} \tag{2.32}$$

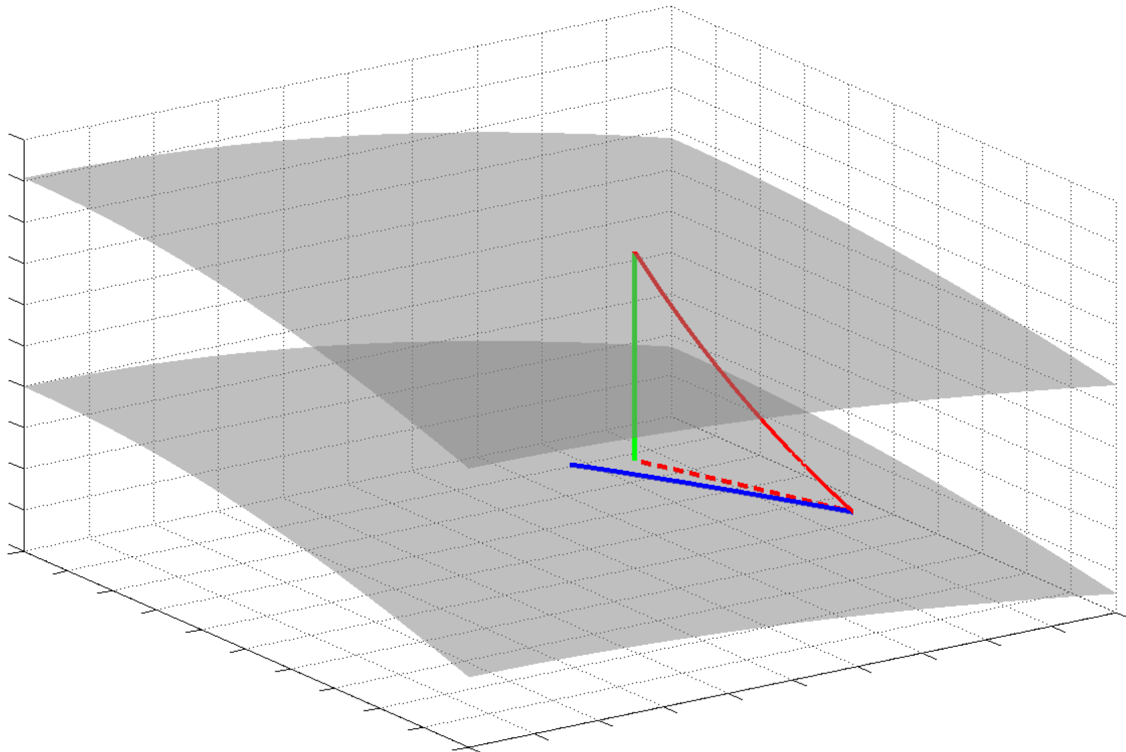


Abbildung 3: Illustration des Projektors P_H. Durch den Fehler des Integrators F entfernt sich die berechnete Trajektorie (rot) von der exakten Lösung (blau). P_H projiziert nach jedem Schritt den errechneten Punkt im Phasenraum wieder auf die Hyperfläche konstanter Hamiltonfunktion (graue Flächen). Die Projektion erfolgt entlang eines Vektors senkrecht zur oberen Hyperfläche (grün). Diese Illustration ist rein anschaulich. Tatsächlich arbeitet P_H in einem $3N$-dimensionalen Zustandsraum mit $3N - 1$-dimensionalen Hyperflächen.

2.5 Projektor der Hamiltonfunktion

Bei vielen numerischen Näherungsverfahren ist eine Abweichung des Funktionswertes der Hamiltonfunktion von seinem Anfangswert zu beobachten. Diese Abweichung reicht von Schwankungen wie bei dem symplektischen Verfahren bis zu einer eindeutigen Drift wie bei dem Runge-Kutta-Verfahren. Da der Wert der Hamiltonfunktion eine Konstante in der Zeitentwicklung sein sollte, scheint es wünschenswert, diese Abweichung zu korrigieren. Es sollte jedoch deutlich unterschieden werden, ob diese Korrektur lediglich eine „kosmetische“ Wirkung hat oder tatsächlich eine bessere Approximation an die exakte Lösung der Bewegungsgleichung erreicht wird.

Alle Zustände mit konstantem Wert der Hamiltonfunktion bilden eine Hyperfläche im Zustands-

raum. Eine naheliegende Methode der Korrektur ist es nun, nach jedem Simulationsschritt den erhaltenen Zustand $\underline{\vec{s}}$ auf die Hyperfläche, die dem Anfangswert der Hamiltonfunktion entspricht, zu projizieren. Der Projektor ist eine Abbildung auf dem Zustandsraum

$$P_H : \mathbb{R}^{3N} \to \mathbb{R}^{3N} \quad .$$

Wird der Projektor nach jedem Integrationsschritt angewendet, so geht nach k Schritten aus dem Anfangszustand $\underline{\vec{s}}_0 = \underline{\vec{s}}(t_0)$ der Zustand $(P_H F_{\Delta t})^k \, \underline{\vec{s}}_0$ hervor. Die Projektion erfolgt entlang des Normalenvektors $\underline{\vec{n}} = \underline{\vec{\nabla}} H$, der sich als Gradient der Hamiltonfunktion $H(\underline{\vec{s}})$ berechnet. Mit dem passenden Vorfaktor λ gilt somit

$$P_H \underline{\vec{s}} = \underline{\vec{s}} + \lambda \underline{\vec{n}} \quad . \tag{2.33}$$

Die Hamiltonfunktion ist durch die Kopplungsmatrix $J_{\mu\nu}$ gegeben.

$$H(\underline{\vec{s}}) = \sum_{\mu<\nu} J_{\mu\nu} \, \vec{s}_\mu \cdot \vec{s}_\nu \tag{2.34}$$

Die Komponenten $\vec{n}_\mu$ des Normalenvektors ergeben sich daher zu

$$\vec{n}_\mu = \vec{\nabla}_\mu H = \sum_\nu J_{\mu\nu} \, \vec{s}_\nu \quad . \tag{2.35}$$

Der exakte Wert der Hamiltonfunktion ist der Wert für den Startzustand $H(\underline{\vec{s}}_0)$. Die Abweichung der Hamiltonfunktion des unprojizierten Zustandes $\underline{\vec{s}}$ wollen wir ΔE nennen.

$$\Delta E \equiv H(\underline{\vec{s}}_0) - H(\underline{\vec{s}}) \tag{2.36}$$

Um den Vorfaktor λ zu bestimmen, betrachten wir nun die Hamiltonfunktion des projizierten Zustandes $P_H \underline{\vec{s}}$:

$$H(P_H \underline{\vec{s}}) = H(\underline{\vec{s}} + \lambda \underline{\vec{n}}) \tag{2.37a}$$

$$= \sum_{\mu<\nu} J_{\mu\nu} \; (\vec{s}_\mu + \lambda \vec{n}_\mu) \cdot (\vec{s}_\nu + \lambda \vec{n}_\nu) \tag{2.37b}$$

$$= \sum_{\mu<\nu} J_{\mu\nu} \, \vec{s}_\mu \cdot \vec{s}_\nu + \lambda \sum_{\mu<\nu} J_{\mu\nu} \; (\vec{s}_\mu \cdot \vec{n}_\nu + \vec{s}_\nu \cdot \vec{n}_\mu) + \lambda^2 \sum_{\mu<\nu} J_{\mu\nu} \, \vec{n}_\mu \cdot \vec{n}_\nu \tag{2.37c}$$

$$= H(\underline{\vec{s}}) + \lambda \sum_{\mu<\nu} J_{\mu\nu} \; (\vec{s}_\mu \cdot \vec{n}_\nu + \vec{s}_\nu \cdot \vec{n}_\mu) + \lambda^2 \sum_{\mu<\nu} J_{\mu\nu} \, \vec{n}_\mu \cdot \vec{n}_\nu \tag{2.37d}$$

Für den idealen Projektor wäre $H(P_H\underline{\vec{s}})$ identisch mit dem exakten Wert $H(\underline{\vec{s}}_0)$. Für kleine Abweichungen dürfen wir in linearer Näherung $\lambda^2 \approx 0$ annehmen. Dann erhalten wir λ durch Lösung der Gleichung

$$H(\underline{\vec{s}}_0) = H(\underline{\vec{s}}) + \lambda \sum_{\mu<\nu} J_{\mu\nu} \ (\vec{s}_\mu \cdot \vec{n}_\nu + \vec{s}_\nu \cdot \vec{n}_\mu) \quad . \tag{2.38}$$

Wir subtrahieren auf beiden Seiten $H(\underline{\vec{s}})$ und erhalten:

$$\Delta E = \lambda \sum_{\mu<\nu} J_{\mu\nu} \ (\vec{s}_\mu \cdot \vec{n}_\nu + \vec{s}_\nu \cdot \vec{n}_\mu) \tag{2.39a}$$

$$= \lambda \sum_{\mu<\nu} J_{\mu\nu} \ \left(\vec{s}_\mu \cdot \sum_\rho J_{\rho\nu} \ \vec{s}_\rho + \vec{s}_\nu \cdot \sum_\rho J_{\rho\mu} \ \vec{s}_\rho \right) \tag{2.39b}$$

$$= \lambda \sum_{\mu<\nu} \sum_\rho J_{\mu\nu} \ (J_{\rho\nu} \ \vec{s}_\mu \cdot \vec{s}_\rho + J_{\rho\mu} \ \vec{s}_\nu \cdot \vec{s}_\rho) \tag{2.39c}$$

$$= \lambda \left(\sum_{\mu<\nu} \sum_\rho J_{\mu\nu} \ J_{\rho\nu} \ \vec{s}_\mu \cdot \vec{s}_\rho + \sum_{\mu<\nu} \sum_\rho J_{\mu\nu} \ J_{\rho\mu} \ \vec{s}_\nu \cdot \vec{s}_\rho \right) \tag{2.39d}$$

$$= \lambda \left(\sum_{\mu<\nu} \sum_\rho J_{\mu\nu} \ J_{\rho\nu} \ \vec{s}_\mu \cdot \vec{s}_\rho + \sum_{\mu>\nu} \sum_\rho J_{\mu\nu} \ J_{\rho\nu} \ \vec{s}_\mu \cdot \vec{s}_\rho \right) \tag{2.39e}$$

$$= \lambda \sum_\mu \sum_\nu \sum_\rho J_{\mu\nu} \ J_{\nu\rho} \ \vec{s}_\mu \cdot \vec{s}_\rho \tag{2.39f}$$

$$= \lambda \sum_\mu \sum_\rho J_{\mu\rho}^2 \ \vec{s}_\mu \cdot \vec{s}_\rho \quad . \tag{2.39g}$$

Dabei haben wir benutzt, dass $J_{\mu\nu} = J_{\nu\mu}$ und $J_{\mu\mu} = 0$ für alle μ, ν. Der Vorfaktor λ lässt sich somit berechnen als

$$\lambda = \frac{\Delta E}{\sum_\mu \sum_\rho J_{\mu\rho}^2 \ \vec{s}_\mu \cdot \vec{s}_\rho} \quad . \tag{2.40}$$

3 Realisierung als C-Programm

Es gibt gegenwärtig auf dem Markt ein breites Angebot an algebraischen und numerischen Software-Paketen wie Mathematica und Matlab. Damit sind die Rechnungen, die wir hier durchführen wollen, sehr komfortabel umzusetzen. Wenn uns aber die Laufzeit unserer Algorithmen interessiert, haben diese Pakete gerade wegen ihrer Mächtigkeit den Nachteil, dass sie den eigentlichen Algorithmus ausbremsen.

Um sicherzustellen, dass dies nicht geschieht, wurde der symplektische Integrator in der maschinennahen Programmiersprache C implementiert. Dadurch wird erreicht, dass der Algorithmus auch weitgehend plattformunabhängig ist. Die Möglichkeiten, die C mit seiner Zeigerarithmetik bietet, kommen uns bei dem Aufbau der Konstruktionsbäume sehr entgegen.

Als Editor wird die freie Software Dev-C++ der Version 4.9.9.2 für Windows genutzt. Um ein lauffähiges Programm zu erstellen, wird der integrierte Compiler dieser Software verwendet. Mit der Verwendung entsprechender Compiler kann das Programm aber auch auf anderen Betriebssystemen genutzt werden.

Alle Variablen, die reelle Zahlen darstellen, sind in diesem Programm vom Typ `double`. Dieser Datentyp belegt 64 Bit, von denen 52 Bit auf die Mantisse entfallen. Das entspricht einer Genauigkeit von etwa 15 Stellen [15].

3.1 Bedienung des Programms

Die Bedienung des Programms erfolgt auf der Kommandozeile. Die möglichen Einstellungen müssen in vier verschiedenen externen Dateien im Textformat angegeben werden. Diese Einstellungen umfassen die Startkonfiguration der Spins, die Konstruktionsbäume der Teilsysteme, die Kopplungsmatrix, die verwendete Zerlegung, die Simulationsdauer und die Anzahl der Zwischenschritte. Beim Aufruf des Programms auf der Kommandozeile wird in einem Parameter ein Verzeichnis angegeben, in dem sich die Dateien mit den Einstellungen befinden. Diese müssen die vorgegebenen Namen `spins.txt`, `tree.txt`, `matrix.txt` und `dec.txt` tragen.

Sind die Einstellungs-Dateien gültig, werden die ausgelesenen Einstellungen angezeigt. Wenn der Benutzer die Korrektheit bestätigt, wird die Berechnung gestartet. Am Ende wird die benötigte Rechenzeit auf dem Bildschirm angegeben.

Die Ergebnisse werden gleichzeitig binär in der Datei `out.bin` und im Textformat in der Datei `out.txt` gespeichert. So ist es leicht möglich, die Daten zur weiteren Auswertung in verschiedene

Programme zu importieren. Um den Import in das Programm Matlab zu erleichtern, werden die Dateien `symplImport.m` und `symplImport2.m` erzeugt. Diese können in Matlab ausgeführt werden und importieren dort die Binärdaten als Matrizen.

3.2 Konzept der Implementation

Die grundlegende Idee, nach der dieses Programm gestaltet ist, besteht darin, sowohl die Drehoperatoren als auch die Einzelspins des betrachteten Systems einheitlich durch Quaternionen darzustellen. Dabei handelt es sich um Vierer-Vektoren, die geeignet sind, sowohl die Dreier-Vektoren der Spins, als auch die Drehoperatoren zu speichern. Die Umrechnungen zwischen den Drehoperatoren und den Spins wird dadurch vereinheitlicht. Bei einem System aus N Spins werden $2N - 1$ Quaternionen benötigt. Die ersten $N - 1$ Quaternionen sind den inneren Knoten des Konstruktionsbaumes zugeordnet und repräsentieren die jeweiligen Drehoperatoren. Die restlichen N Quaternionen speichern die Einzelspins und sind den Blättern zugeordnet.

Die betrachteten symplektischen Verfahren arbeiten mit zwei Konstruktionsbäumen. Grundsätzlich kann aber eine beliebige Anzahl von Bäumen unterstützt werden. Die Anzahl der benötigten Quaternionen wird von der Anzahl der Konstruktionsbäume nicht beeinflusst. Die verschiedenen Bäume nutzen dieselben $2N - 1$ Quaternionen und unterscheiden sich voneinander nur in den Kopplungskonstanten und darin, wie die Quaternionen vernetzt werden. Das ist möglich, weil die Drehoperatoren ohnehin für jeden Teilschritt neu bestimmt werden müssen. Für die Blätter ist es sogar zwingend erforderlich, dass verschiedene Konstruktionsbäume auf dieselben Einzelspins zugreifen, da jeder Teilschritt mit dem Ergebnis des vorherigen Teilschritts weiterrechnen muss, welcher im Allgemeinen einen anderen Baum benutzt hat.

Die Verwendung von Quaternionen zur Darstellung von Drehungen im Allgemeinen wird im Anhang A erläutert. Die spezielle Abfolge von Operationen zur symplektischen Integration ist im Kapitel 3.4 beschrieben. Der Quellcode des Programms kann im Anhang D eingesehen werden.

3.3 Die Module des Programms

Das Programm ist aus mehreren Modulen aufgebaut, die verschiedene Teil-Funktionen realisieren.

Das Hauptmodul `main.c` beschreibt den allgemeinen Ablauf des Programms. Hier werden auch die weiteren benötigten Bibliotheken eingebunden und alle globalen Variablen deklariert.

Die Header-Datei `structures.h` definiert alle grundlegenden benötigten Strukturen. So beschreibt die Struktur `QUAT` eine Quaternion, indem sie vier Gleitkommazahlen vom Typ `double` als Einheit zusammenfasst. Die Struktur für ein Blatt aus dem Konstruktionsbaum `LEAF` besteht aus den beiden Zeigern auf das zugehörige `QUAT` und auf das `QUAT` des Vater-Knotens. Ein innerer Knoten des Konstruktionsbaumes wird durch die Struktur `NODE` beschrieben, die eine Variable vom Typ `double` für die reduzierte Kopplungskonstante $\tilde{c}_i$, einen Zeiger auf die zugehörige Quaternion und zwei Zeiger auf die Quaternionen der beiden Sohn-Knoten enthält. Darüber hinaus definiert `structures.h` verschiedene Operationen auf Basis der Quaternionen, die bei der Simulation benötigt werden, wie beispielsweise Addition und Multiplikation.

Das Modul `treebuilder.h` enthält die Funktion `buildTree()`. Mit dieser Funktion wird die Datei `tree.txt` ausgelesen und anhand der dortigen Angaben der Konstruktionsbaum erstellt, indem die Zeiger der `NODE`s und `LEAF`s auf die passenden `QUAT`s gesetzt werden. Außerdem werden die reduzierte Kopplungskonstanten $\tilde{c}_i$ gesetzt. Der Aufbau der Datei `tree.txt` wird bei den Beispielen im Anhang C erläutert.

Die Header-Datei `matrixbuilder.h` liest die Datei `matrix.txt` ein und erstellt entsprechend eine J-Kopplungsmatrix sowie ihr Quadrat J^2. Diese beiden Matrizen werden benötigt, um eine Projektion auf die konstante Hamiltonfunktion durchführen zu können. Dies wird in Abschnitt 2.5 näher beschrieben. Für die symplektische Integration selbst werden diese Matrizen nicht benötigt.

Die Aufgabe des Moduls `spinreader.h` ist es, aus der Datei `spins.txt` die Anfangskonfiguration der Einzelspins auszulesen und diese in den `QUAT`s der Blätter des Konstruktionsbaumes abzuspeichern. Dabei wird der Realteil der Quaternion auf Null gesetzt und die anderen drei Komponenten auf die Koordinaten des Spins. Anschließend werden die Spins auf die Länge Eins normiert, um sicherzustellen, dass die Bedingung $|\vec{s}_\mu| = 1$ erfüllt ist.

Mit Hilfe des Moduls `decsetter.h` wird die Datei `dec.txt` ausgelesen. Dort ist die verwendete Zerlegung, die Simulationsdauer und die Anzahl der Zwischenschritte angegeben.

Das Modul `calculus.h` schließlich übernimmt die eigentliche Simulation. Der schematische Verlauf eines symplektischen Integrationsschritts wird im nächsten Abschnitt anhand eine Beispiels erläutert. Aber auch die Projektoren, der Runge-Kutta-Integrator und einige Hilfsfunktionen sind in diesem Modul implementiert.

3.4 Implementierung des symplektischen Integrators

Im Folgenden soll beschrieben werden, wie ein symplektischer Integrationsschritt in diesem Programm durchgeführt wird. Voraussetzung für einen korrekten Ablauf ist, dass die Konstruktionsbäume in der Datei `tree.txt` so angegeben sind, dass der Sohn eines inneren Knotens niemals vor seinem Vaterknoten genannt wird. Ist diese Bedingung gegeben, so liegen die von `buildTree()` erzeugten `NODE`s für jeden Baum sortiert in einem Array der Länge $N - 1$ vor. An der Stelle `0` des Arrays befindet sich die Wurzel. Der Index eines `NODE` ist stets höher als der Index seines Vaterknotens.

Jeder symplektische Integrationsschritt besteht gemäß (2.21) aus mehreren Teilschritten von der Form (2.20).

In einem solchen Teilschritt wird das Array der inneren Knoten als Erstes von hohen zu niedrigen Indizes durchlaufen. Bei jedem `NODE` werden die beiden Quaternionen der Söhne ausgelesen, die von Zeigern `son1` und `son2` referenziert werden. Diese beiden Quaternionen werden addiert und des Ergebnis in die Quaternion des aktuellen Knotens gespeichert, auf die der Zeiger `data` verweist.

Handelt es sich bei einem Sohn des aktuellen Knotens um ein Blatt, so enthält die entsprechende Quaternion einen Einzelspin $\vec{s}_\mu$. Ist der Sohn dagegen ein innerer Knoten, so ist durch die Reihenfolge der `NODE`s im Array sichergestellt, dass in der entsprechenden Quaternion die Summe für diesen Teilschritt bereits berechnet worden ist. Sobald der Algorithmus alle inneren Knoten durchlaufen hat, sind in allen Quaternionen der inneren Knoten die Gesamtspins $\vec{S}_i$ der Teilsysteme enthalten. Durch dieses Vorgehen können die Knoten die Zwischenergebnisse ihrer Söhne verwerten.

Nun wird für jeden inneren Knoten der Gesamtspin $\vec{S}_i$ in die Quaternionen-Repräsentation des Drehoperators $\mathbb{D}\left(\tilde{c}_i \vec{S}_i(t_0), \Delta t\right)$ umgerechnet. Danach durchläuft das Programm abermals das Array der inneren Knoten, diesmal jedoch von niedrigen zu hohen Indizes. Jeder innere Knoten multipliziert nun seine Quaternion von links mit der Quaternion seines Sohnes und legt das Ergebnis in die Quaternion dieses Sohnes ab. Diese Multiplikation wird für beide Söhne durchgeführt.

Hat der der Algorithmus das Array durchlaufen, so steht in den Quaternionen der Väter der Blätter nun der vollständige Drehoperator $\prod_{i=1}^{L-1} \mathbb{D}\left(\tilde{c}_i \vec{S}_i(t_0), \Delta t\right)$. Diese Quaternion ist nun auch schon einmal mit den Quaternionen der Blätter multipliziert. Wegen der Art und Weise, wie Drehungen durch Quaternion dargestellt werden, ist es nötig, in einem letzten Schritt die Quaternionen in den Blättern mit den Konjugierten der Quaternionen-Repräsentation der Drehoperatoren zu multiplizieren, um die Drehung abzuschließen.

Nachdem auf diese Weise ein Teilschritt mit dem ersten Konstruktionsbaum berechnet wurde, wird hiervon ausgehend ein Teilschritt mit dem zweiten Konstruktionsbaum berechnet. Je nach der verwendeten Zerlegung folgen weitere solche Teilschritte abwechselnd mit den beiden Konstruktionsbäumen. Am Ende des Integrationsschritts stehen dann in den Quaternionen der Blätter die berechneten Einzelspins nach einer Schrittweite Δt.

4 Ergebnisse

In diesem Abschnitt werden einige Ergebnisse präsentiert, die bei der Anwendung der vorgestellten Algorithmen gewonnen wurden. Als erstes wollen wir das Verhalten einiger Erhaltungsgrößen betrachten. Dabei zeigt sich, dass bei den verschiedenen Integratoren verschieden geartete Abweichungen auftreten.

Die Abweichungen in den Erhaltungsgrößen lassen sich durch geeignete Projektoren beheben. Das Verhalten dieser Projektoren wird daher genauer analysiert.

Zur Bewertung der Projektoren wird der *Euklidische Abstand* zur genähert korrekten Lösung eingeführt. Daraufhin vergleichen wir auch das Verhalten dieses Abstandes für die verschiedenen Integratoren.

Da für die Qualität eines Algorithmus auch dessen Laufzeit von wesentlicher Bedeutung ist, wird abschließend auf die benötigten Rechenzeiten der verschiedenen Integratoren eingegangen.

Die in diesem Abschnitt gezeigten Plots wurden mit Matlab erstellt. Dazu wurden die zuvor errechneten Resultate mit `symplImport2.m` importiert und dann aus diesen Daten die gewünschten Größen errechnet.

4.1 Betrachtete Systeme

Im Laufe der Diplomarbeit wurden mehrere verschiedene Spinsysteme implementiert. Eine Auswahl dieser Systeme befindet sich im Anhang C. In einer frühen Phase der Arbeit diente vor allem das Spin-Quadrat als exakt lösbares und überschaubares Modell mit dem Zweck, die Algorithmen zu testen und erste Ergebnisse zu erhalten.

Die hier nun vorgestellten Ergebnisse beziehen sich auf ein System von 30 Spins, die in Form eines Ikosidodekaeders angeordnet sind. Ein solches System ist bereits erfolgreich in Form des Moleküls $\{Mo_{72}Fe_{30}\}$ synthetisiert worden [17]. Das magnetische Verhalten dieses Moleküls wird vor allem durch die 30 Fe^{3+}-Ionen bestimmt, wobei die nächsten Nachbarn über O–Mo–O-Brücken miteinander koppeln [18]. Wir bezeichnen dieses System auch kurz als Fe_{30}. Als Startkonfiguration wird eine zufällige Ausrichtung der einzelnen Spins gewählt.

Abbildung 4 zeigt einen Graph des Fe_{30}-Systems. Farblich hervorgehoben ist die verwendete Zerlegung in sechs Fliegen und acht Dreiecke.

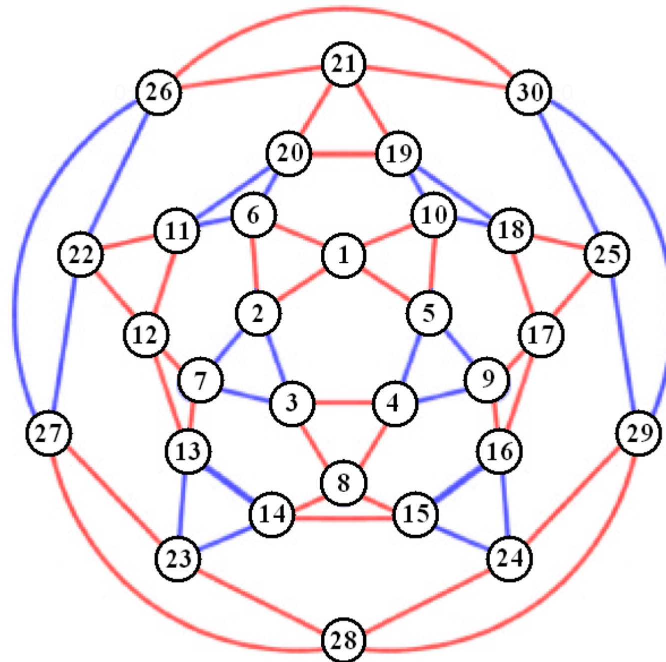


Abbildung 4: Das Fe_{30}-System als ebener Graph, zerlegt in zwei integrable Teilsysteme bestehend aus sechs Fliegen (rot) bzw. acht Dreiecken (blau). Diese Zerlegung wurde übernommen aus [1]. Die Nummerierung der Spins und die Vorlage dieser Grafik stammen aus [16].

4.2 Erhaltungsgrößen

Die Konstanz von Erhaltungsgrößen ist ein notwendiges Kriterium für eine exakte Zeitentwicklung. Auch wenn für einige Probleme kein exakter Integrator gefunden werden kann, ist es wünschenswert, dass die bekannten Erhaltungsgrößen wenigstens annähernd konstant bleiben. Da sich das Verhalten von Erhaltungsgrößen einfach überprüfen lässt, ist dies ein wichtiger Indikator für die Qualität eines Integrators. Oft ist die Betrachtung von Erhaltungsgrößen bei sehr komplizierten Problemen, von denen keine exakte Lösung bekannt ist, eine der wenigen Möglichkeiten, überhaupt Aussagen über die verwendeten Methoden zu machen.

Für klassische Heisenberg-Systeme lassen sich oft mehrere spezifische Erhaltungsgrößen finden. Unabhängig vom betrachteten System sind stets die Hamiltonfunktion $H(\underline{\vec{s}})$ und die Komponenten des Gesamtspins $\vec{S}$ Erhaltungsgrößen. Darüber hinaus sind natürlich auch immer die Beträge der Einzelspins $|\vec{s}_\mu|$ erhalten.

Die Abbildungen 5 bis 7 zeigen den Verlauf der Hamiltonfunktion über einen simulierten Zeitraum von 100 Zeiteinheiten für die verschiedenen Integratoren. Als Schrittweiten Δt wurden 0,1, 0,01

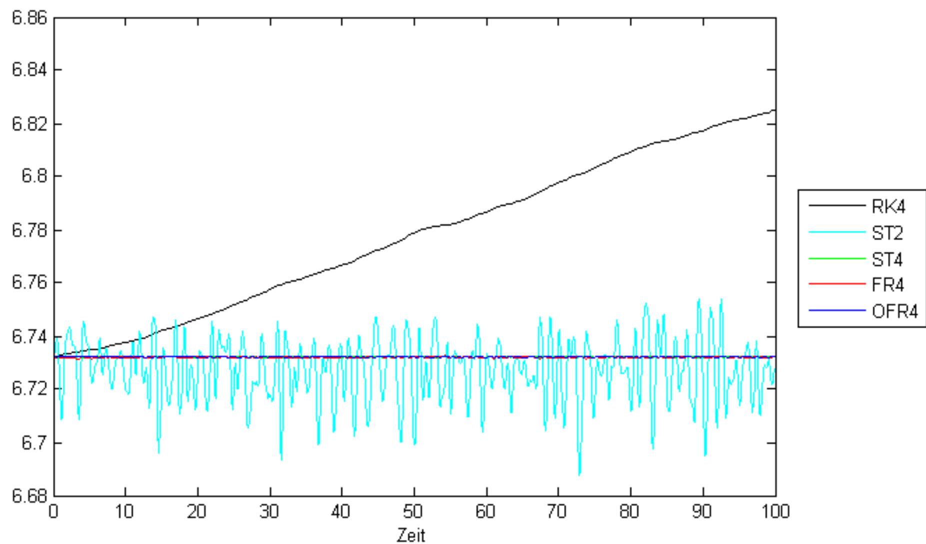


Abbildung 5: Hamiltonfunktion bei 1 000 Schritten in 100 Zeiteinheiten, $\Delta t = 0{,}1$
Der Runge-Kutta-Integrator zeigt eine deutliche Drift, während bei den symplektischen Verfahren lediglich Schwankungen von wesentlich geringerem Ausmaß auftreten.

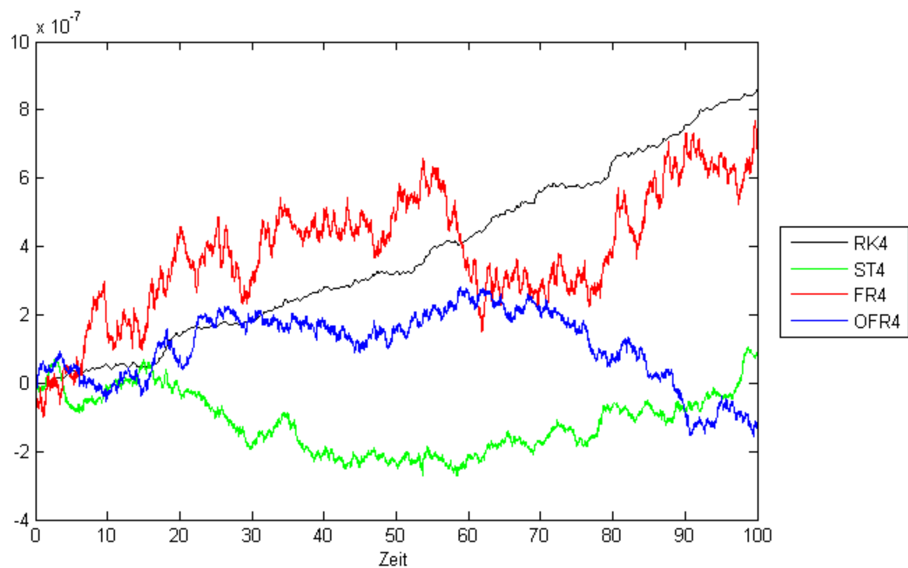


Abbildung 6: Hamiltonfunktion bei 10 000 Schritten in 100 Zeiteinheiten, $\Delta t = 0{,}01$
Die Drift des Integrators RK4 ist immer noch deutlich zu erkennen. In dem betrachteten Zeitraum ist die daraus resultierende Ungenauigkeit aber von der gleichen Größenordnung wie die Schwankungen der symplektischen Integratoren vierter Ordnung.

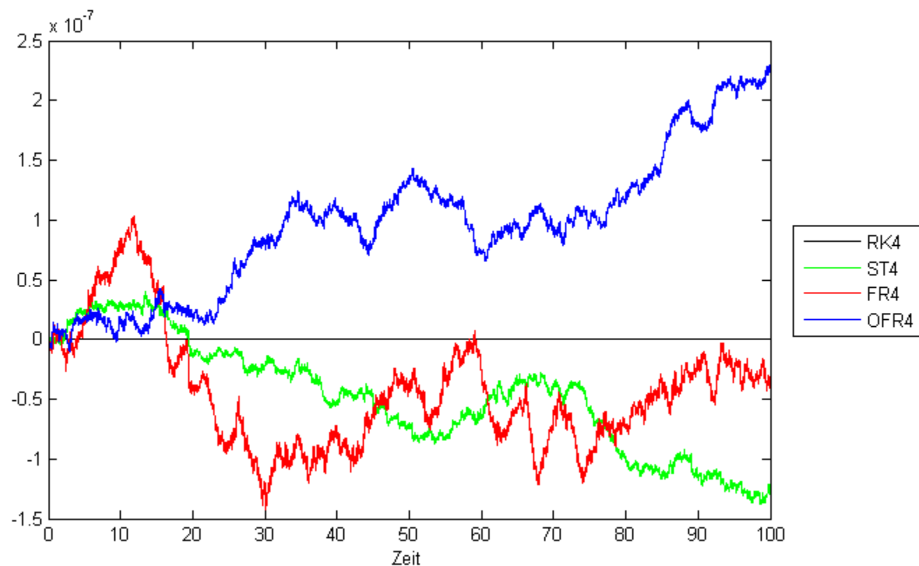


Abbildung 7: Hamiltonfunktion bei 100 000 Schritten in 100 Zeiteinheiten, $\Delta t = 0{,}001$
Die Schwankungen der symplektischen Integratoren haben sich kaum verringert. Die Drift des Verfahrens nach Runge-Kutta dagegen ist nun so gering, dass sie in diesem Plot nicht mehr auszumachen ist.

und 0,001 Zeiteinheiten gewählt. Bei den letzten beiden Diagrammen wurde der symplektische Integrator zweiter Ordnung ST2 weggelassen und der Wert der Hamiltonfunktion zum Startzeitpunkt $H_0 = H(\vec{\underline{s}})|_{t=0}$ subtrahiert, um eine brauchbare Skala zu erhalten.

Während bei den symplektischen Verfahren H um den Anfangswert schwankt, ist für den Runge-Kutta-Integrator eine eindeutige Drift weg von H_0 zu beobachten. Überraschenderweise zeigt sich, dass bei einer Verkleinerung der Schrittweite die Abweichung des Integrators RK4 stärker abnimmt, als die der symplektischen Integratoren. Es scheint so, dass das Runge-Kutta-Verfahren von einer kleineren Schrittweite mehr profitiert.

Da die Teilschritte der symplektischen Integratoren die exakte Zeitentwicklung integrierbarer Systeme beschreiben, ist die Erhaltung der Komponenten des Gesamtspins bis auf numerisch bedingte Ungenauigkeiten wie Rundungsfehler garantiert. Auch für das Runge-Kutta-Verfahren wurde in Abschnitt 2.4 gezeigt, dass der Gesamtspin erhalten bleibt. Beide Vorhersagen werden durch die Ergebnisse der Simulationen bestätigt, wie in Abbildung 8 zu sehen ist.

Auch die Erhaltung der Beträge der Einzelspins ist bei den symplektischen Integratoren garantiert, da lediglich Drehoperatoren auf die Einzelspins angewendet werden. Bei dem Integrator RK4 hin-

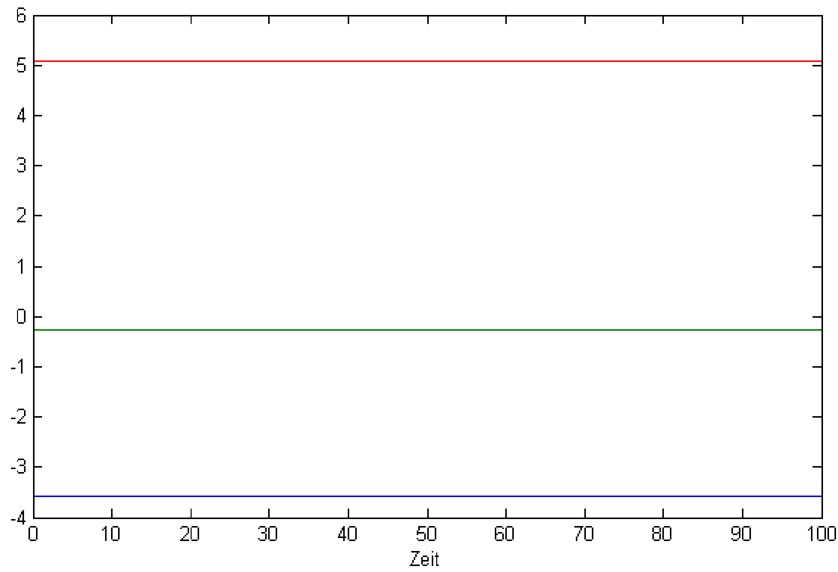


Abbildung 8: Der Gesamtspin bleibt bei allen Integratoren sehr gut erhalten.

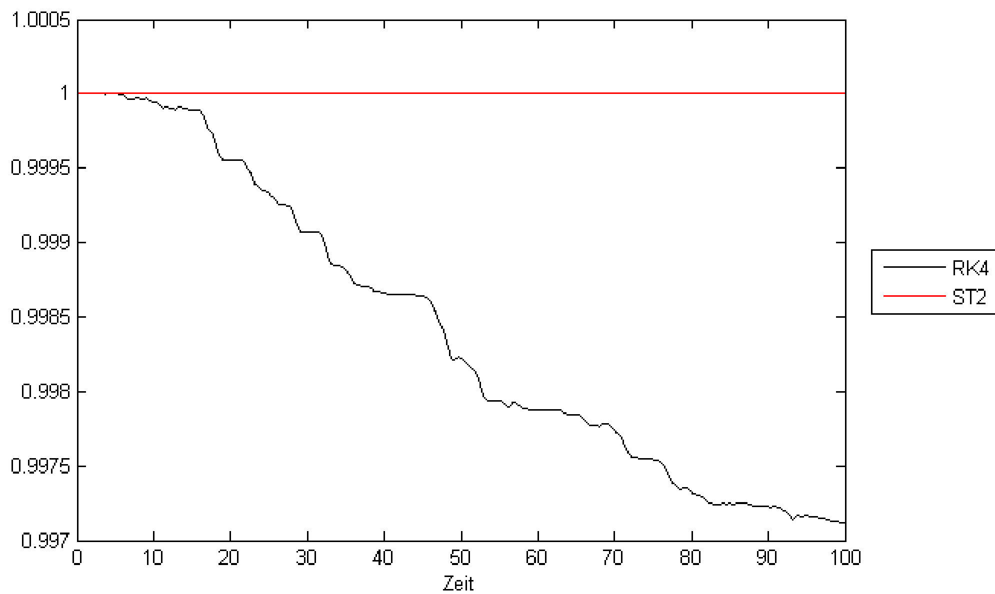


Abbildung 9: Bei den symplektischen Integratoren bleiben die Beträge der Einzelspins $\vec{s}_\mu = 1$ erhalten. Beim Runge-Kutta-Integrator dagegen zeigt sich eine eindeutige Drift.

gegen ist eine deutliche Drift auszumachen. Das Diagramm in Abbildung 9 wurde mit $\Delta t = 0{,}1$ erstellt und zeigt ein stetes Abfallen eines Einzelspins. Bei den symplektischen Integratoren wie dem ST2 kann keine größere Abweichung beobachtet werden.

4.3 Projektoren

Diese Abweichungen bei den Erhaltungsgrößen können durch Projektoren wieder ausgeglichen werden. Im Falle der Hamiltonfunktion verwenden wir den in 2.5 eingeführten Projektor P_H. Zwar ist dieser Projektor nicht exakt, aber im logarithmischen Plot in Abbildung 10 erkennt man, dass die Abweichung $|H - H_0|$ deutlich verringert werden kann. Durch mehrmaliges Anwenden von P_H wird die Abweichung weiter reduziert.

Allerdings führt die Anwendung dieses Projektors dazu, dass bisher konstant gebliebene Erhaltungsgrößen nun schwanken. In Abbildung 11 ist die Gesamtspinkomponente S^x aufgetragen. Während ohne Projektion noch keine Abweichungen auftritt, zeigt sich nun mit Anwendung von P_H jedoch eine Drift bei Runge-Kutta und Schwankungen beim symplektischen Integrator. Qualitativ tauchen hier genau die Arten von Abweichungen auf, die ohne Projektion bei der Hamiltonfunktion zu beobachten waren. Im Diagramm 12 sieht man, dass die Beträge der Einzelspins bei den symplektischen Verfahren nun ebenfalls Abweichungen aufweisen.

Diese durch P_H hervorgerufenen Abweichungen lassen sich durch weitere Projektoren ebenfalls beheben. Die Beträge der Einzelspins hält der Projektor P_A durch einfache Normierung konstant. Die Konstanz des Gesamtspins wird durch den Projektor P_T erzwungen, indem von jedem der N Einzelspins der Vektor $\frac{\vec{S}-\vec{S}_0}{N}$ subtrahiert wird.

Wünschenswert wäre ein einziger Projektor P, der alle diese Größen gleichzeitig konstant hält. Dieser Projektor lässt sich durch mehrmaliges Anwenden der drei vorhandenen Projektoren als

$$P^k = (P_H P_A P_T)^k \tag{4.1}$$

annähern. In den Simulationen erhalten wir für $k = 10$ schon sehr gute Ergebnisse. Bei $k = 100$ ist die Grenze der numerisch möglichen Konstanz bereits erreicht.

Es stellt sich allerdings die Frage, ob die durch Projektion erhaltene Zeitentwicklung wirklich näher an der exakten Zeitentwicklung ist, oder ob es sich nur um eine „kosmetische“ Maßnahme handelt. Um dies zu bewerten, wollen wir nun den *Euklidischen Abstand* einführen.

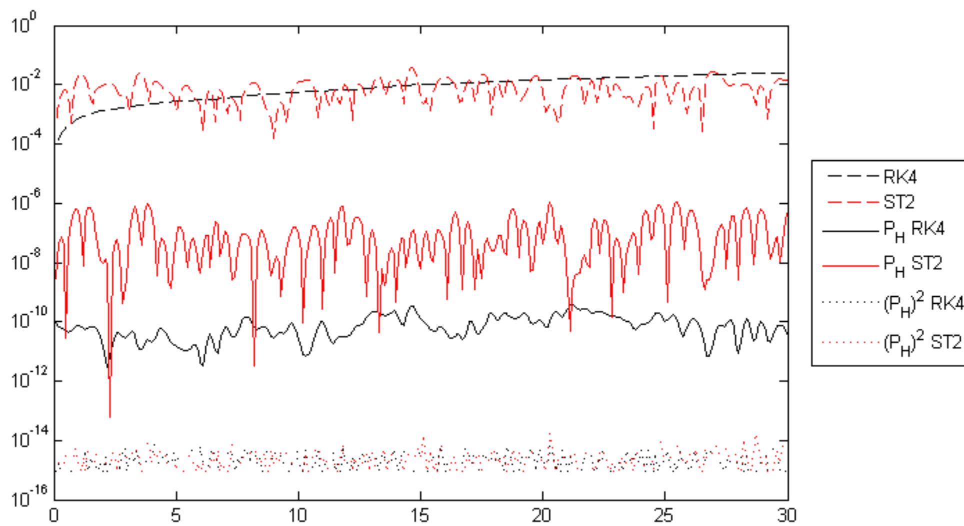


Abbildung 10: Mit angewendetem Projektor P_H sind die Abweichungen der Hamiltonfunktion $|H - H_0|$ von ST2 und RK4 bei $\Delta t = 0{,}1$ um einige Größenordnungen kleiner.

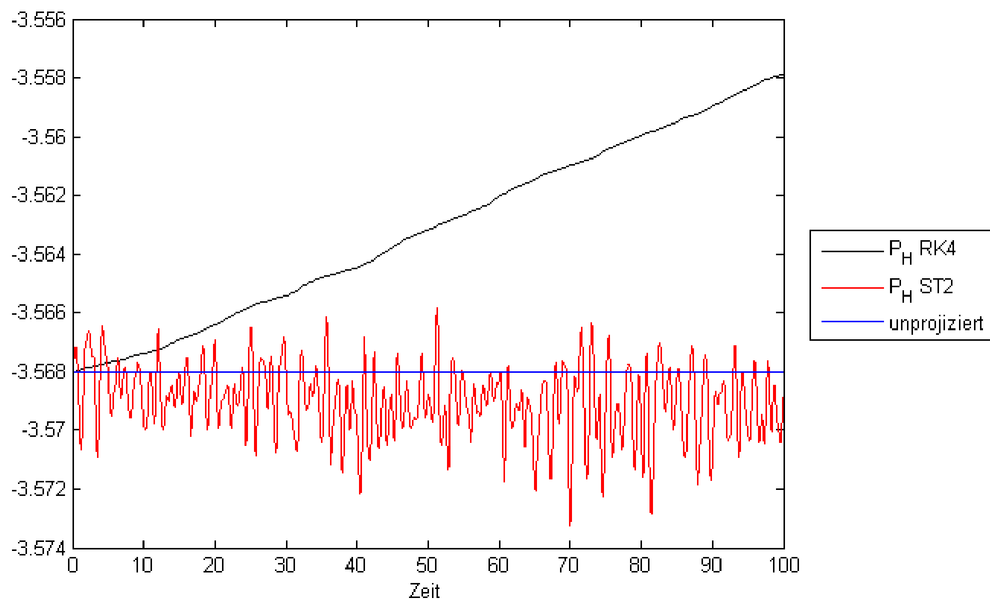


Abbildung 11: Durch die Projektion auf konstante Hamilton-Funktion ist nun der Gesamtspin nicht mehr erhalten. Die Komponente S^x beispielsweise zeigt nun eine Drift bei RK4 und Schwankungen bei ST2. ($\Delta t = 0{,}1$)

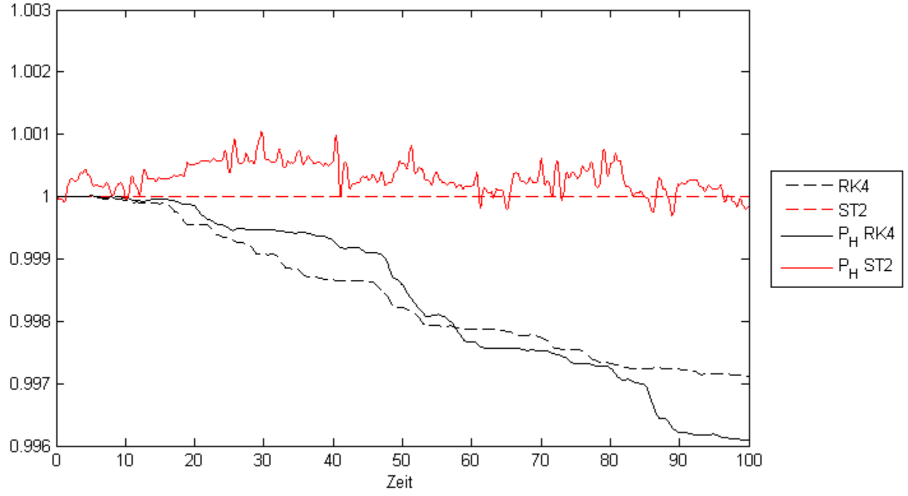


Abbildung 12: Auch die Beträge der Einzelspins sind mit der Anwendung von P_H Schwankungen unterworfen. ($\Delta t = 0{,}1$)

4.4 Euklidischer Abstand

Der Euklidische Abstand $d(\vec{\underline{r}}, \vec{\underline{s}})$ ist die durch die Euklidische Norm festgelegte Metrik auf dem betrachteten Zustandsraum $\mathbb{R}^{3N}$

$$d(\vec{\underline{r}}, \vec{\underline{s}}) = \|\vec{\underline{r}} - \vec{\underline{s}}\| = \sqrt{\sum_{\mu=1}^{N} (\vec{r}_\mu - \vec{s}_\mu)^2} = \sqrt{\sum_{\mu=1}^{N} \sum_{i=1}^{3} (r_\mu^i - s_\mu^i)^2} \ . \tag{4.2}$$

Wenn man Zustände des $2N$-dimensionalen Phasenraumes $\mathcal{P}$ betrachtet, müsste eigentlich ein geodätischer Abstand definiert werden. Allerdings würde dies zu Verfälschungen führen, wenn Zustände auftreten, die außerhalb des Phasenraumes liegen. Dies wurde beim Runge-Kutta-Integrator beobachtet, da dieser die Beträge der Einzelspins nicht erhält. Darüber hinaus ist bei kleinen Abständen der Unterschied zwischen Euklidischem und geodätischem Abstand vernachlässigbar.

Insbesondere wollen wir den Verlauf des Abstandes

$$d'(\vec{\underline{s}}) = d(\vec{\underline{s}}^*, \vec{\underline{s}}) \tag{4.3}$$

mit der Zeit betrachten. Dabei sei $\vec{\underline{s}}$ der durch wiederholte Anwendung des Integrators simulierte Zustand $\vec{\underline{s}} = (F_{\Delta t})^k \ \vec{\underline{s}}|_{t=0}$ und $\vec{\underline{s}}^*$ entspreche möglichst dem exakten Zustand zum jeweiligen Zeitpunkt $\vec{\underline{s}}^* = \vec{\underline{s}}|_{t=k\Delta t}$.

Bei nicht-integrablen Systemen stehen wir nun vor dem Problem, dass die exakten Zustände der Zeitentwicklung eben nicht bekannt sind. Die exakte Zeitentwicklung wird daher durch numerische Integration mit sehr kleinem $\Delta t = 10^{-6}$ genähert. Bei 100 simulierten Zeiteinheiten werden also 10^8 Schritte berechnet. Um die Datenmenge handhaben zu können, wird nur jeder tausendste Schritt gespeichert. Mit dem verwendeten Rechner dauert solch eine Simulation für das Fe_{30}-System etwa zehn Stunden.

Um keinen Integrator indirekt zu bevorzugen, wird diese Simulation sowohl mit dem ST4- als auch mit dem RK4-Integrator durchgeführt. Daher können wir zusätzlich einen Vergleichsabstand

$$d^* = d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}}^*_{\mathrm{RK4}}) \tag{4.4}$$

definieren. Bei der Hamiltonfunktion haben wir gesehen, dass die Abweichungen der beide Integratoren völlig verschiedener Art sind. Deshalb ist es äußerst unwahrscheinlich, dass diese beiden Näherungen nun gleichartige Fehler produzieren. Wir gehen daher davon aus, das sowohl $\underline{\vec{s}}^*_{\mathrm{ST4}}$, als auch $\underline{\vec{s}}^*_{\mathrm{RK4}}$ in guter Näherung der exakten Zeitentwicklung entsprechen, solange d^* sehr klein ist.

Aufgrund der Dreiecksungleichung gilt

$$\begin{aligned} d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}}) - d^* \le d(\underline{\vec{s}}^*_{\mathrm{RK4}}, \underline{\vec{s}}) \le d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}}) + d^* \\ d(\underline{\vec{s}}^*_{\mathrm{RK4}}, \underline{\vec{s}}) - d^* \le d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}}) \le d(\underline{\vec{s}}^*_{\mathrm{RK4}}, \underline{\vec{s}}) + d^* \end{aligned} \tag{4.5}$$

Daraus folgt

$$d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}}) \approx d(\underline{\vec{s}}^*_{\mathrm{RK4}}, \underline{\vec{s}}) \;\text{ für }\; d^* \ll d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}}), d(\underline{\vec{s}}^*_{\mathrm{RK4}}, \underline{\vec{s}}) \;\;. \tag{4.6}$$

Solange die Bedingung von kleinem d^* gegeben ist, betrachten wir wahlweise $d(\underline{\vec{s}}^*_{\mathrm{ST4}}, \underline{\vec{s}})$ oder $d(\underline{\vec{s}}^*_{\mathrm{RK4}}, \underline{\vec{s}})$ als hinreichend gute Näherung zu der uns interessierenden Größe $d'(\underline{\vec{s}})$. In Abbildung 13 sehen wir den Verlauf von $d'(\underline{\vec{s}})$ bei $\Delta t = 0{,}1$ und d^*. Alle Kurven zeigen zunächst einen exponentiellen Anstieg bis zur Sättigung, die bei einem Abstand von etwa 7 erreicht ist. Ab hier ist ein Schwanken innerhalb des Intervalls $[6, 9]$ zu beobachten.

Der maximal mögliche Abstand zweier Punkte auf der Einheitskugel ist 2. Für das Fe_{30}-System ergibt sich entsprechend $d_{\max} = \sqrt{30 \cdot 2^2} = \sqrt{120} \approx 11$. Bei einem Abstand in der Größenordnung von 6 bis 9 haben wir es also nicht mehr mit zwei nahe beieinander liegenden Punkten zu tun. Stattdessen beobachten wir in dem Sättigungsbereich die Schwankungen des Abstandes zwischen zwei völlig unkorrelierten Punkten im Zustandsraum.

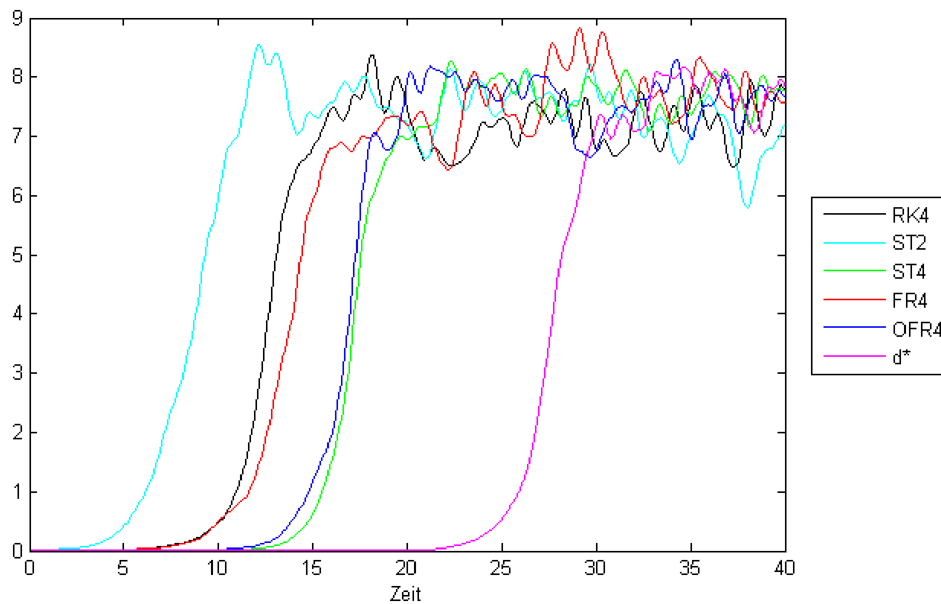


Abbildung 13: Der Euklidische Abstand $d'(\vec{\underline{s}})$ für die verschiedenen Integratoren bei $\Delta t = 0{,}1$ und der Vergleichsabstand d^*.

Der Vergleichsabstand d^* steigt deutlich langsamer als die $d'(\vec{\underline{s}})$. Bis etwa $t = 20$ liegt der Wert unter 10^{-2}, während die $d'(\vec{\underline{s}})$ hier schon Sättigung erreicht haben. Daher betrachten wir bis $t \leq 20$ die Bedingung $d^* \ll d'(\vec{\underline{s}})$ als gegeben. Ein direkter Vergleich der beiden Abstände $d(\vec{\underline{s}}^*_{\mathrm{ST4}}, \vec{\underline{s}})$ und $d(\vec{\underline{s}}^*_{\mathrm{RK4}}, \vec{\underline{s}})$ in Abbildung 14 zeigt sogar erst ab $t = 25$ einen sichtbaren Unterschied.

In Abbildung 15 ist das Diagramm 13 noch einmal logarithmisch dargestellt. Deutlich sieht man, dass die Kurven bis zur Sättigung, abgesehen von kleinen Abweichungen, von der Form $a\,e^{b\,t}$ sind. Auffällig ist dabei, dass sich die Kurven nur in dem Vorfaktor a stark voneinander unterscheiden, während der Koeffizient b bei allen Kurven bei $b \approx 0{,}8$ liegt. Dies äußert sich darin, dass die Kurven in der logarithmischen Darstellung Geraden mit gleicher Steigung bilden.

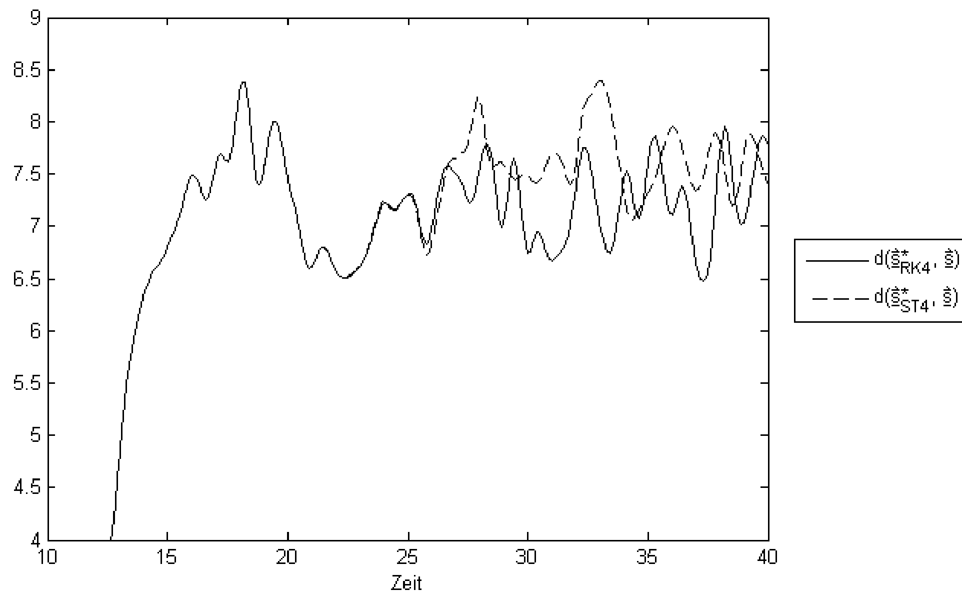


Abbildung 14: Ein deutlicher Unterschied zwischen den beiden Realisationen von $d'(\vec{s})$ ist erst ab etwa $t = 25$ auszumachen.

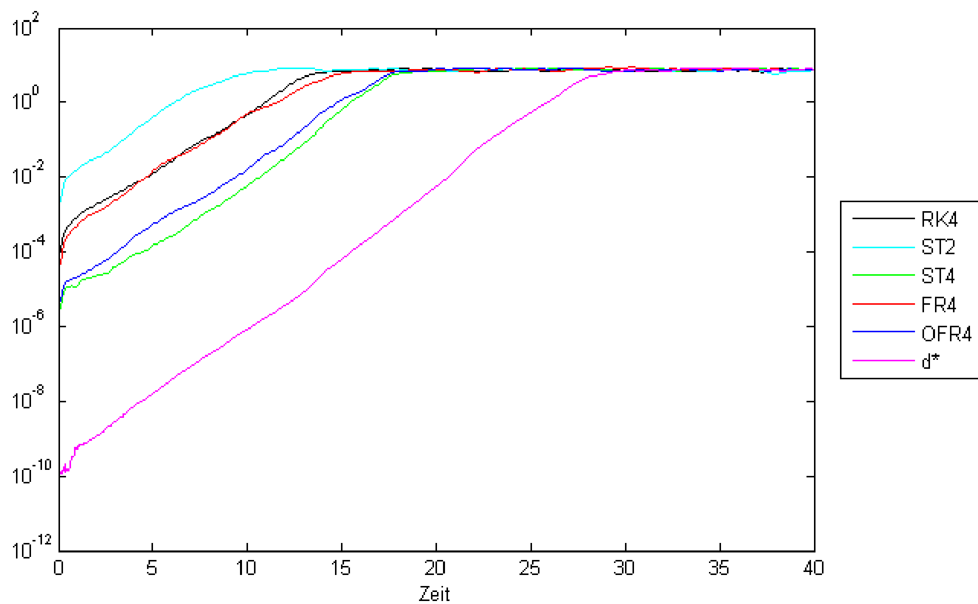


Abbildung 15: Die Abstände $d'(\vec{s})$ bei $\Delta t = 0{,}1$ und d^* in logarithmischer Auftragung.

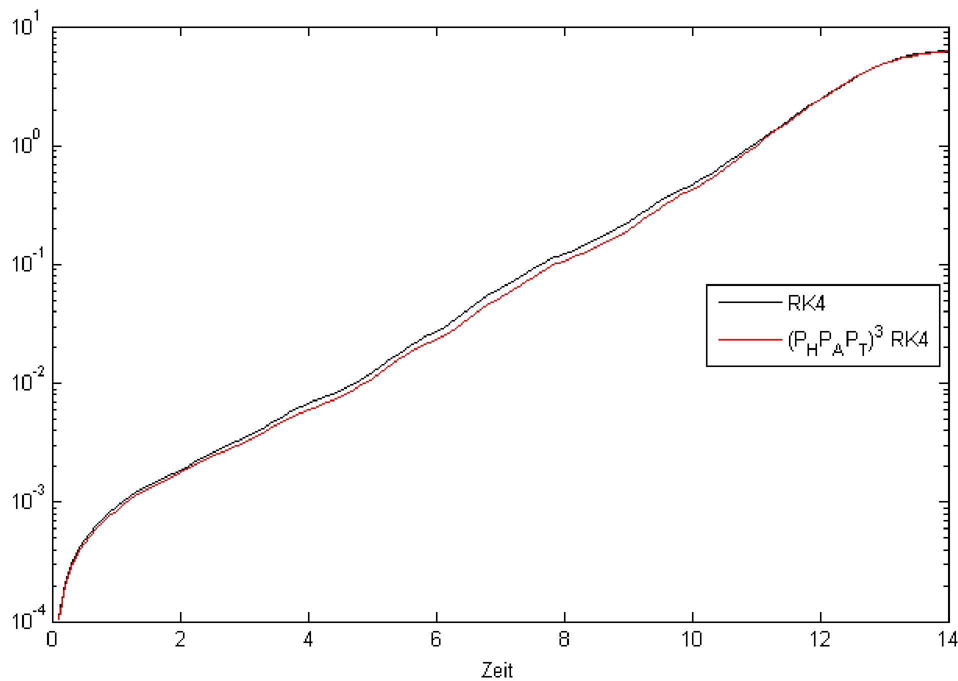


Abbildung 16: Der Euklidische Abstand unter Verwendung des Runge-Kutta-Integrators bei $\Delta t = 0{,}1$ mit und ohne Projektion. Durch die Verwendung des Projektors wird nur eine minimale Verbesserung erzielt.

4.4.1 Bewertung der Projektoren

Für die Projektoren stellt sich nun heraus, dass keine wesentliche Verbesserung erzielt wird. Abbildung 16 zeigt beispielhaft den Verlauf von $d'(\vec{\underline{s}})$ für den RK4-Integrator bei $\Delta t = 0{,}1$ einmal ohne Projektor und einmal mit dreimaliger Projektion auf die einzelnen Erhaltungsgrößen $(P_H P_A P_T)^3$. Noch häufigeres Projizieren $(P_H P_A P_T)^k$ mit $k > 3$ wurde bis $k = 100$ getestet, erzeugte aber keine Kurve, die sich von der hier dargestellten mit $k = 3$ unterscheiden ließ.

Auch mit den anderen Integratoren wird ein ähnliches Verhalten des Projektors beobachtet. In den betrachteten Fällen ist die einzige nennenswerte Wirkung das Projektors, dass die jeweilige Erhaltungsgröße konstant gehalten wird. Die tatsächliche Verbesserung verglichen mit der exakten Zeitentwicklung ist so minimal, dass sie vernachlässigt werden kann. Zieht man noch die durch die zusätzlichen Rechnungen verlängerte Laufzeit in Betracht, so zeigt sich, dass sich der Einsatz des Projektors unter dem Gesichtspunkt des Euklidischen Abstandes zur exakten Zeitentwicklung nicht lohnt. Stattdessen ist die Rechenzeit besser genutzt, wenn man die Schrittweite Δt verkleinert.

4.4.2 Bewertung der Integratoren

Vor diesem Hintergrund erscheint die Bewertung der Integratoren auf Basis des Verhaltens einiger Erhaltungsgrößen, wie sie im Abschnitt 4.2 vorgenommen wurde, als fraglich. Denn ein Integrator mit eingebauten Projektoren würde hier überdurchschnittlich gut bewertet, obwohl die berechneten Spin-Zustände möglicherweise sogar stärker von der exakten Zeitentwicklung abweichen. Daher sollen hier die verschiedenen Integratoren anhand des Verhaltens von $d'(\underline{\vec{s}})$ bewertet werden.

Für $\Delta t = 0{,}1$ können wir diese Bewertung bereits der Abbildung 15 entnehmen. Hier scheinen sich die Integratoren in drei Gruppen einteilen zu lassen. Am schlechtesten schneidet die Suzuki-Trotter-Zerlegung zweiter Ordnung ab, die bereits bei $t = 10$ die Sättigung erreicht. Die Integratoren RK4 und FR4 sind bei $t = 14$ gesättigt. Das beste Ergebnis erzielen die beiden symplektischen Integratoren ST4 und FR4, die die Sättigung bei $t = 17$ erreichen.

Das Diagramm 17 zeigt den Verlauf der $d'(\underline{\vec{s}})$ für $\Delta t = 0{,}01$. Hier scheinen alle vier Integratoren vierter Ordnung von etwa gleicher Qualität zu sein. Das passt zu unserem Befund aus 4.2 bei der Betrachtung der Hamiltonfunktion. Auch hier waren die Abweichungen dieser vier Integratoren bei $\Delta t = 0{,}01$ von der gleichen Größenordnung, während bei $\Delta t = 0{,}1$ der Runge-Kutta-Integrator den symplektischen Verfahren noch deutlich unterlegen war. Wenn sich dieser Befund auch hier fortsetzt, dann müsste $d'(\underline{\vec{s}})$ für Δt kleiner als $0{,}01$ bei dem RK4-Integrator deutlich unter den Werten der symplektischen Integratoren liegen.

Der Kurvenverlauf in Abbildung 18 bei $\Delta t = 0{,}004$ scheint diese Annahme zu bestätigen. Allerdings sollte dieses Diagramm nur mit Vorsicht bewertet werden, da die Bedingung $d^* \ll d'(\underline{\vec{s}})$ hier nicht mehr erfüllt ist. Zwei unterscheidbare Verläufe der beiden Realisationen von $d'(\underline{\vec{s}})$ zeigen sich dennoch nur für das RK4-Verfahren. Die Tatsache, dass in beiden Fällen der Abstand stets unterhalb der Abstände der symplektischen Integratoren bleibt, spricht allerdings dafür, dass diese Zeitentwicklung tatsächlich deutlich näher an der exakten Lösung liegt.

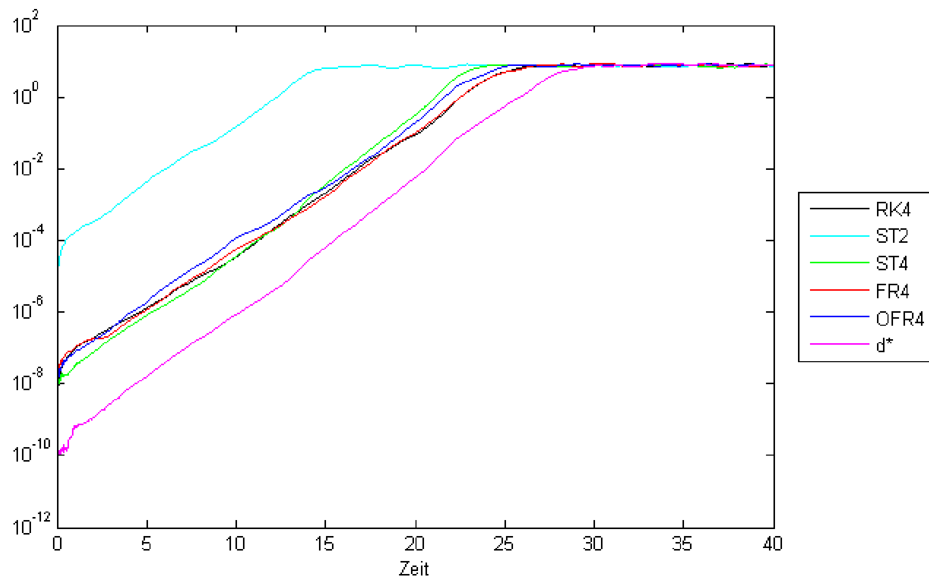


Abbildung 17: Die Abstände $d'(\vec{\underline{s}})$ bei $\Delta t = 0{,}01$ und d^* in logarithmischer Auftragung.

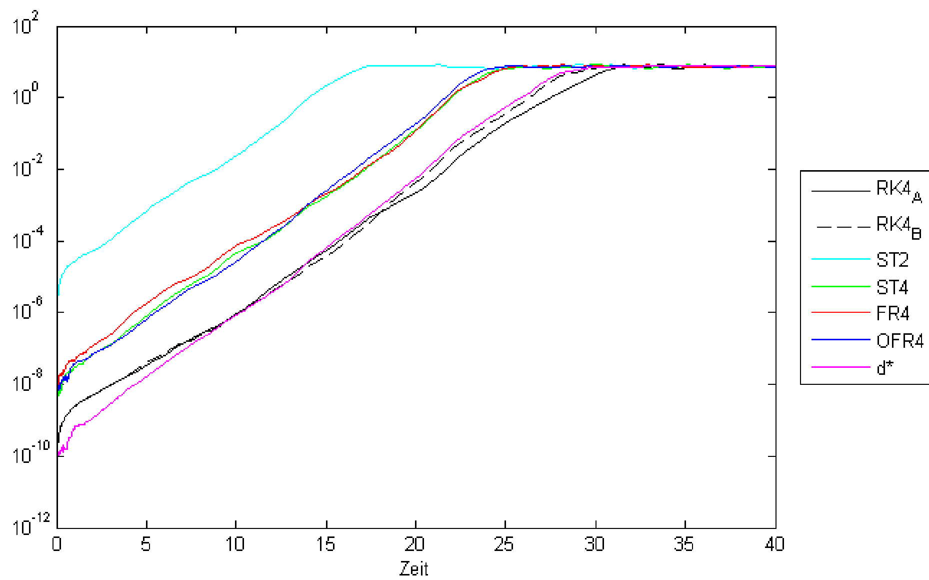


Abbildung 18: Die Abstände $d'(\vec{\underline{s}})$ bei $\Delta t = 0{,}004$ und d^* in logarithmischer Auftragung. Für RK4 bewegt sich der Abstand in der Größenordnung von d^* und zeigt Unterschiede, je nachdem mit welcher Zeitentwicklung verglichen wird. RK4_A bezeichnet den Abstand von $\vec{\underline{s}}^*_{\text{RK4}}$ und RK4_B den Abstand von $\vec{\underline{s}}^*_{\text{ST4}}$.

4.5 Laufzeiten

Bisher haben wir hier nur die verschiedenen Integratoren bei gleicher Schrittweite miteinander verglichen. Allerdings ist es ebenso wichtig, die Laufzeiten der Algorithmen zu beachten. Schließlich kann ein schnellerer Algorithmus in einer gegebenen Rechenzeit mehr Zwischenschritte berechnen als ein langsamer Algorithmus und so die Genauigkeit des Ergebnisses erhöhen.

Zur Messung der Laufzeiten wurde die Funktion `clock()` aus dem Paket `time.h` der C-Standard-Bibliothek verwendet. Der Rückgabewert dieser Funktion repräsentiert die verstrichene Zeit seit dem Start des Prozesses [19]. Dieser Wert wird direkt vor und nach der Simulation aller Schritte gespeichert. Die Differenz der beiden Werte gibt die verstrichene Zeit in Millisekunden an.

Eine detailliertere Betrachtung der einzelnen Anweisungen des Programms, die in diesem Intervall durchgeführt werden, ergab, dass ein großer Teil der Zeit dafür gebraucht wird, die Ergebnisse nach jedem Schritt in die Ausgabedateien zu schreiben. Eine naheliegende Modifikation der Algorithmen ist es daher, bei einer Verkleinerung der Schrittweite auf die Speicherung der zusätzlichen Zwischenergebnisse zu verzichten. Dieses Verfahren wurde im vorherigen Kapitel auch bei der Berechnung der genähert exakten Lösungen verwendet.

Würde die Zeit, die für das Speichern der Zwischenschritte benötigt wird, bei der Bewertung der Laufzeiten berücksichtigt, so wäre beispielsweise der ST2-Integrator gegenüber dem ST4-Integrator stark benachteiligt, da bei ersterem die eigentliche Berechnung der Zeitentwicklung ohne das Speichern einen deutlich geringeren Anteil ausmacht. Deshalb wird für die Messung der Laufzeiten das Programm so modifiziert, dass überhaupt keine Ergebnisse gespeichert werden. Es wird also nur die Zeit gemessen, die allein für Rechenleistung notwendig ist.

Die Laufzeitmessung mit diesem modifizierten Programm wird auf einem gewöhnlichen Personal Computer mit dem Betriebssystem Microsoft Windows XP durchgeführt. Die Taktfrequenz des Prozessors beträgt 2000 MHz. Es wird darauf geachtet, dass während der Messung keine unnötigen Programme auf dem Rechner ausgeführt werden. Allerdings sind dennoch Schwankungen in der gemessenen Zeit zu beobachten, die vermutlich von Aktivitäten des Betriebssystems im Hintergrund herrühren.

Aus diesem Grund führen wir für jeden Integrator mehrere Messungen durch und bilden den Mittelwert. Mit dem ST2-Integrator werden 10^6 Schritte je Messung berechnet, mit den anderen Integratoren 10^5 Schritte je Messung. Das betrachtete System war wieder das Fe_{30}-System.

Die Tabelle 2 fasst die Ergebnisse der Laufzeitmessung zusammen. Die einzelnen gemessenen Zeiten können der Tabelle 4 im Anhang B entnommen werden.

Integrator	Zeit pro Schritt [μs]	Mögliche Schritte in 100 ms
ST2	88,17 $\pm$ 0,18	1134
ST4	317,45 $\pm$ 1,54	315
FR4	202,36 $\pm$ 0,43	494
OFR4	258,90 $\pm$ 0,34	386
RK4	126,38 $\pm$ 8,31	791

Tabelle 2: Ergebnisse aus der Messung der Laufzeiten.

Bei dem Mittelwert der benötigten Zeit pro Schritt ist zusätzlich die Standardabweichung

$$\sigma = \sqrt{\frac{1}{k-1}\sum_{i=1}^{k}(\tau_i - \bar{\tau})^2} \tag{4.7}$$

angegeben. Dabei bezeichnet k die Anzahl der Messungen, τ_i sind die gemessenen Zeiten und $\bar{\tau}$ ist der Mittelwert $\bar{\tau} = \frac{1}{k}\sum_{i=1}^{k}\tau_i$. Warum die Standardabweichung bei dem Runge-Kutta-Integrator wesentlich größer ist als bei den symplektischen Integratoren konnte nicht geklärt werden.

Aus Abschnitt 2.3 wissen wir, dass sich ein Schritt eines symplektischen Integrators aus mehreren Teilschritten zusammensetzt. Teilen wir die gemessenen Zeiten durch die Anzahl der Schritte, so müsste das Resultat konstant die Laufzeit eines einzelnen symplektischen Teilschritts wiedergeben. Diese Vorhersage wird in Tabelle 3 überprüft.

Integrator	Zeit pro Schritt [μs]	Anzahl der Teilschritte	Zeit pro Teilschritt [μs]
ST2	88,17	3	29,39
ST4	317,45	11	28,86
FR4	202,36	7	28,91
OFR4	258,90	9	28,77

Tabelle 3: Kontrolle der Laufzeit eines symplektischen Teilschritts.

Die Vorhersage wird mit kleinen Abweichungen bestätigt. Die größte Abweichung tritt bei dem Suzuki-Trotter-Integrator zweiter Ordnung auf. Dies ist wohl dadurch zu erklären, dass neben den

Teilschritten bei jedem symplektischen Schritt weitere Anweisungen wie das Hochzählen der Laufvariable der `for`-Schleife ausgeführt werden.

Da nun die Laufzeiten der Integratoren bekannt sind, wissen wir, wie viele Schritte bei vorgegebener Rechenzeit simuliert werden können. Mit den Angaben aus Tabelle 2 wird nun für jeden Integrator eine Schrittzahl gewählt, die einer Rechenzeit von 100 ms entspricht und wieder der Euklidische Abstand betrachtet. Die simulierte Gesamtzeit ist 100 Zeiteinheiten. Entsprechend liegt die Schrittweite Δt zwischen $\frac{100}{1134} \approx 0,088$ und $\frac{100}{315} \approx 0,317$. Die errechneten Graphen sind in Abbildung 19 zu sehen.

Die Forrest-Ruth-Zerlegung FR4 schneidet in diesem Vergleich ähnlich schlecht wie die ST2-Zerlegung ab. Die übrigen drei Integratoren vierter Ordnung sind von ungefähr vergleichbarer Qualität. Lediglich der Runge-Kutta-Integrator scheint einen minimal größeren Abstand zur exakten Lösung zu haben.

Das Diagramm 20 wird nach dem gleichen Vorgehen bei 1000 ms vorgegebener Rechenzeit erstellt. Auch hier zeigt der FR4-Integrator ein schlechteres Verhalten als die anderen Integratoren vierter Ordnung, ist aber nun deutlich besser als der ST2-Integrator. Zwischen den Integratoren ST4, OFR4 und RK4 ist kein Qualitätsunterschied auszumachen.

Bei den Fluktuationen am Anfang der Kurven handelt es sich um Artefakte, die auftreten, solange der Abstand $d'(\underline{\vec{s}})$ noch nicht wesentlich größer als 10^{-6} ist. Die Ursache dieser Artefakte liegt in den unterschiedlichen Schrittweiten begründet. Durch die angepassten Schrittweiten werden nun Zustände $\underline{\vec{s}}$ zu Zeitpunkten berechnet, bei denen keine genähert exakten Zustände $\underline{\vec{s}}^*$ vorliegen. Daher werden die genähert exakten Zustände für diese Zeitpunkte interpoliert. Dabei wird eine Interpolation erster Ordnung verwendet. Solange die tatsächlichen Euklidischen Abstände noch unter 10^{-6} liegen, werden die Graphen durch die Ungenauigkeit der Interpolation dominiert. Bei $d'(\underline{\vec{s}}) \geq 10^{-4}$ ist dieser Fehler jedoch vernachlässigbar.

Abbildung 21 zeigt die Euklidischen Abstände bei 1500 ms vorgegebener Rechenzeit. Wesentlicher Unterschied zu der vorhergehenden Abbildung ist, dass der Runge-Kutta-Integrator nun ein besseres Verhalten zeigt als die symplektischen Integratoren. Dies ist ein weiteres Indiz für Vermutung, dass der RK4-Integrator ab einem bestimmten Punkt von einer Verkleinerung der Schrittweite stärker profitiert als die symplektischen Verfahren.

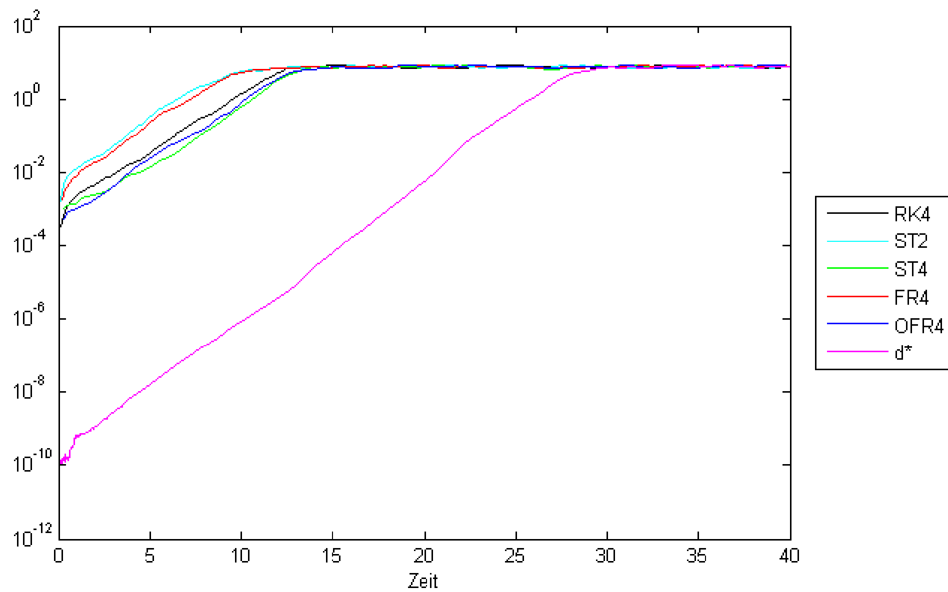


Abbildung 19: Die Abstände $d'(\vec{s})$ bei einer vorgegebenen Rechenzeit von 100 ms und d^* in logarithmischer Auftragung.

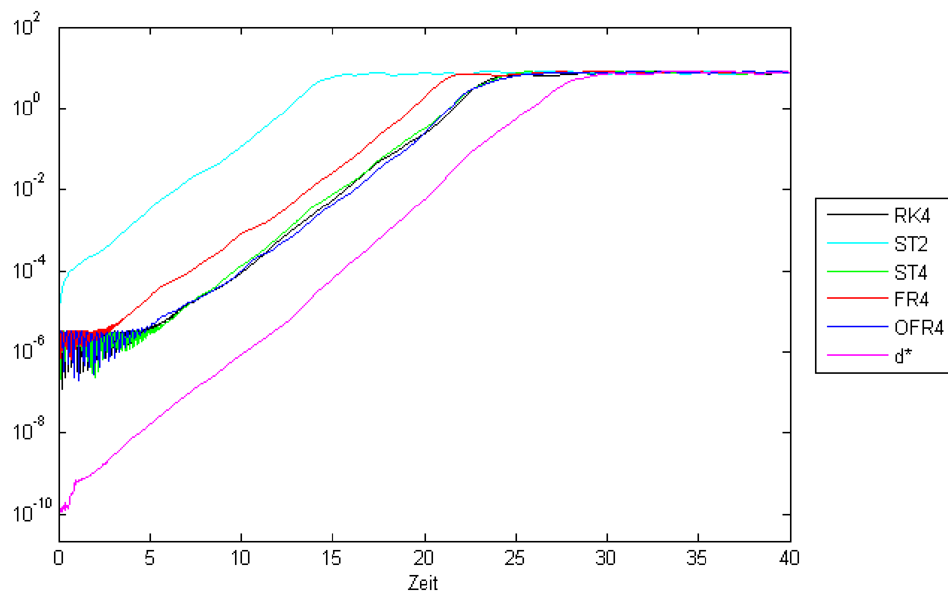


Abbildung 20: Die Abstände $d'(\vec{s})$ bei einer vorgegebenen Rechenzeit von 1000 ms und d^* in logarithmischer Auftragung.

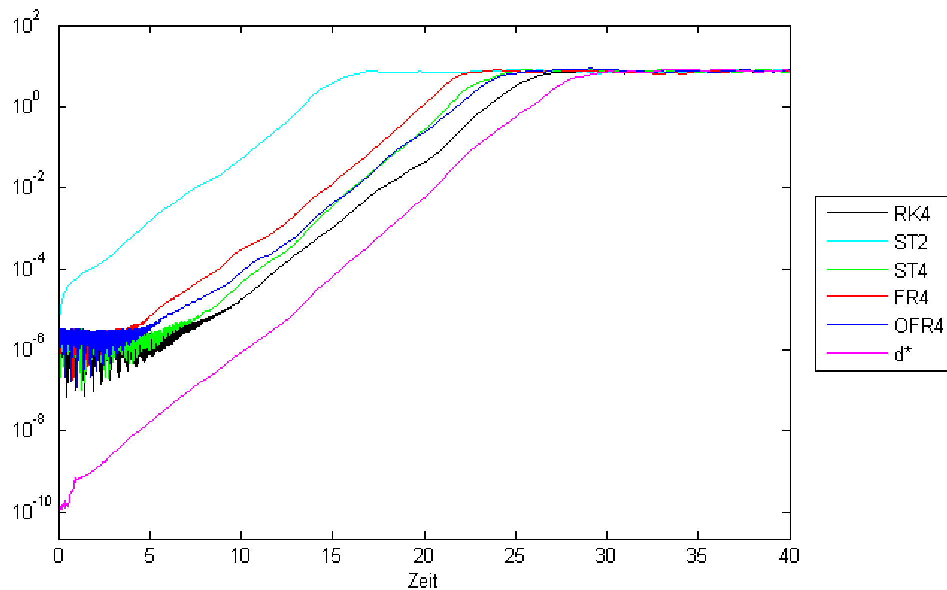


Abbildung 21: Die Abstände $d'(\vec{s})$ bei einer vorgegebenen Rechenzeit von 1500 ms und d^* in logarithmischer Auftragung.

Zusammengefasst ergibt sich aus der Betrachtung der Laufzeiten in Verbindung mit dem Euklidischen Abstand zur genähert exakten Lösung folgende Bewertung:

Die Integratoren ST2 und FR4 schaffen unter den symplektischen Verfahren in einer vorgegebenen Rechenzeit zwar die meisten Schritte, weisen aber dennoch die größten Abweichungen auf. Der ST4-Integrator ist bei gegebener Schrittweite zwar dem OFR4-Integrator überlegen, letzterer kann dies aber durch seine höhere Geschwindigkeit ausgleichen. Bei vorgegebener Rechenzeit haben diese beiden Integratoren in etwa die gleich Qualität.

Der RK4-Integrator kann von allen betrachteten Integratoren vierter Ordnung in einer vorgegeben Rechenzeit die meisten Schritte berechnen. Dabei soll an dieser Stelle nochmals darauf hingewiesen werden, dass der verwendete Algorithmus genauso wie die symplektischen Integratoren selbst implementiert wurde. Es handelt sich also nicht um einen hochgradig optimierten Algorithmus, wie er in professioneller Numerik-Software zur Verfügung steht. Bei eher kurzer vorgegebener Rechenzeit ist der Runge-Kutta-Integrator qualitativ mit den Integratoren ST4 und OFR4 vergleichbar. Wird die Rechenzeit weiter ausgedehnt, so liefert das RK4-Verfahren ab einem bestimmten Punkt bessere Ergebnisse als die anderen betrachteten Integratoren.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurden verschiedene Verfahren zur numerischen Simulation klassischer Spinsysteme verglichen. Diese Verfahren umfassen verschiedene Varianten des symplektischen Integrators, sowie den bekannten Algorithmus nach Runge-Kutta vierter Ordnung. Außerdem wurden Kombinationen dieser Integratoren mit Projektoren für einzelne Erhaltungsgrößen untersucht.

Für die Projektoren konnte gezeigt werden, dass sie ihre Aufgabe, eine Erhaltungsgröße konstant zu halten, erfüllen. Die erhoffte Wirkung, dass sich durch die Anwendung der Projektoren der Euklidische Abstand zur exakten Lösung wesentlich verkleinert, konnte leider nicht bestätigt werden.

Unter den symplektischen Integratoren weisen bei vorgegebener Rechenzeit die Zerlegung nach Suzuki-Trotter vierter Ordnung (ST4) und die optimierte Zerlegung nach Forest-Ruth (OFR4) die besten Ergebnisse auf. Es kann als ein Vorteil gegenüber ST4 angesehen werden, dass der OFR4-Integrator dabei in gleicher Rechenzeit mehr Zwischenergebnisse liefert.

Bei dem Vergleich der symplektischen Verfahren mit dem Runge-Kutta-Integrator (RK4) wurde festgestellt, dass die beiden Verfahren unterschiedlich stark von einer Erhöhung der verfügbaren Rechenzeit profitieren. Bei geringer Rechenzeit und großen Schrittweiten zeigen die symplektischen Verfahren einen geringeren Euklidischen Abstand zur exakten Lösung als RK4. Wenn allerdings genügend Rechenzeit zur Verfügung steht, dann scheint es empfehlenswerter, den Algorithmus nach Runge-Kutta zu verwenden. Ein weiterer Vorteil des RK4 ist, dass es nicht nötig ist, vor der eigentlichen Rechnung nach einer geeigneten Aufteilung des Systems in zwei integrable Systeme zu suchen und die Konstruktionsbäume aufzustellen. Die Ursache für die unterschiedliche Abhängigkeit von der Schrittweite konnte nicht geklärt werden.

Zwar wurde versucht, den symplektischen Algorithmus möglichst optimal zu gestalten, jedoch sind noch weitere Optimierungsmöglichkeiten denkbar, die im Rahmen dieser Arbeit nicht untersucht wurden. Zum Beispiel wird bei der Konstruktion der Drehoperatoren auf die nativen trigonometrische Funktionen `sin` und `cos` der Programmiersprache C zurückgegriffen. Für sehr kleine Schrittweiten Δt wäre denkbar, dass es effizienter ist, eine Reihenentwicklung der trigonometrischen Funktionen geringer Ordnung zu verwenden.

Die Konstruktionsbäume wurden als strikt binäre Bäume implementiert. In einer Verallgemeinerung könnte ein innerer Knoten eine beliebige Anzahl von Söhnen haben. Da es oft vorkommt, dass mehr als zwei Teilsysteme im Konstruktionsbaum gleichförmig koppeln, würde dadurch die benötigte Anzahl von Knoten und die Tiefe des Baumes verringert.

Eine weitere Optimierungsmöglichkeit betrifft die Drehung durch Quaternionen (siehe Anhang A). Wenn die Quaternionen in den direkten Vätern der Blätter feststehen, wird in dieser Implementierung die zweifache Multiplikation $\boldsymbol{q} * \boldsymbol{s} * \overline{\boldsymbol{q}}$ durch zwei vollständige Quaternionenmultiplikationen durchgeführt. Unter Ausnutzung der Tatsache, dass $\overline{\boldsymbol{q}}$ das Konjugierte von $\boldsymbol{q}$ ist, und dass der Realteil von $\boldsymbol{s}$ Null ist, wäre es möglich, beide Operationen zu einer zusammenzufassen. Dabei könnten einige Additionen und Multiplikationen von `floats` eingespart werden und so der einzige Punkt wo die Quaternionen gegenüber den Drehmatrizen im Nachteil sind, abgemildert würde.

A Drehung durch Quaternionen

Im Folgenden soll erläutert werden, was Quaternionen sind, wie Drehungen von dreidimensionalen Vektoren mit Quaternionen dargestellt werden und was die Vorteile gegenüber Drehmatrizen sind. Eine umfassende Darstellung der Anwendung von Quaternionen zur Beschreibung von Drehung findet sich beispielsweise in [20] und [21].

A.1 Definition und Einführung

Wie die komplexen Zahlen, so sind auch die Quaternionen eine Erweiterung der reellen Zahlen. Statt nur einer imaginären Einheit werden drei imaginäre Einheiten, i, j und k eingeführt, die sich unabhängig voneinander addieren. Eine Quaternion $\boldsymbol{q}$ besteht mit einem Realteil und drei Imaginärteilen also aus vier Komponenten

$$\boldsymbol{q} = x_0 + x_1\, i + x_2\, j + x_3\, k \quad , \tag{A.1}$$

wobei x_0, x_1, x_2 und x_3 reelle Zahlen sind. Die Multiplikation zwischen den verschiedenen imaginären Einheiten sowie der Eins sind nach den *Hamilton-Regeln* wie folgt definiert:

$$\begin{array}{llll} 1\cdot 1 = 1 & 1\cdot i = i & 1\cdot j = j & 1\cdot k = k \\ i\cdot 1 = i & i\cdot i = -1 & i\cdot j = k & i\cdot k = -j \\ j\cdot 1 = j & j\cdot i = -k & j\cdot j = -1 & j\cdot k = i \\ k\cdot 1 = k & k\cdot i = j & k\cdot j = -i & k\cdot k = -1 \end{array} \tag{A.2}$$

Häufig werden die Hamilton-Regeln in der kürzeren Formel

$$i^2 = j^2 = k^2 = ijk = -1 \tag{A.3}$$

zusammengefasst. Man beachte, dass die Multiplikation von zwei Quaternionen nicht kommutativ ist. Ansonsten werden aber alle Eigenschaften eines algebraischen Körpers erfüllt. Die Quaternionen bilden somit einen sogenannten *Schiefkörper*.

Wegen der einfacheren Schreibweise wollen wir die Quaternionen im Folgenden als Vierer-Vektoren betrachten. Außerdem wird es sich als nützlich erweisen, die drei imaginären Komponenten zu

einem Dreier-Vektor zusammenzufassen.

$$\boldsymbol{q} = \alpha + x\,i + y\,j + z\,k \equiv \begin{pmatrix} \alpha \\ x \\ y \\ z \end{pmatrix} \equiv \begin{pmatrix} \alpha \\ \vec{v} \end{pmatrix} \tag{A.4}$$

Wir nennen $\vec{v} = \left(\begin{smallmatrix} x \\ y \\ z \end{smallmatrix}\right)$ auch den *Vektorteil* von $\boldsymbol{q}$ und der Realteil α wird entsprechend *Skalarteil* genannt.

Die Addition und äquivalent auch die Subtraktion von zwei Quaternionen erfolgt komponentenweise.

$$\boldsymbol{q_1} + \boldsymbol{q_2} = \begin{pmatrix} \alpha_1 + \alpha_2 \\ x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{pmatrix} = \begin{pmatrix} \alpha_1 + \alpha_2 \\ \vec{v_1} + \vec{v_2} \end{pmatrix} \tag{A.5}$$

Die Multiplikationsvorschrift ergibt sich aus den Hamilton-Regeln.

$$\boldsymbol{q_1} * \boldsymbol{q_2} = \begin{pmatrix} \alpha_1\alpha_2 - x_1x_2 - y_1y_2 - z_1z_2 \\ x_1\alpha_2 + \alpha_1x_2 - z_1y_2 + y_1z_2 \\ y_1\alpha_2 + z_1x_2 + \alpha_1y_2 - x_1z_2 \\ z_1\alpha_2 - y_1x_2 + x_1y_2 + \alpha_1z_2 \end{pmatrix} = \begin{pmatrix} \alpha_1\alpha_2 - \vec{v_1} \cdot \vec{v_2} \\ \alpha_2\vec{v_1} + \alpha_1\vec{v_2} + \vec{v_1} \times \vec{v_2} \end{pmatrix} \tag{A.6}$$

Wir nennen $\overline{\boldsymbol{q}}$ die zu $\boldsymbol{q}$ *konjugierte* Quaternion, wenn $\boldsymbol{q}$ und $\overline{\boldsymbol{q}}$ denselben Skalarteil haben, ihre

Vektorteile aber genau entgegengesetzt sind.

$$\overline{\boldsymbol{q}} = \begin{pmatrix} \alpha \\ -x \\ -y \\ -z \end{pmatrix} = \begin{pmatrix} \alpha \\ -\vec{v} \end{pmatrix} \quad \text{für} \quad \boldsymbol{q} = \begin{pmatrix} \alpha \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \alpha \\ \vec{v} \end{pmatrix} \tag{A.7}$$

Das Konjugierte eines Produkts von zwei Quaternionen ist das umgekehrte Produkt der konjugierten Quaternionen, das heißt $\overline{\boldsymbol{q_1} * \boldsymbol{q_2}} = \overline{\boldsymbol{q_2}} * \overline{\boldsymbol{q_1}}$. Den Beweis liefert die kurze Rechnung

$$\begin{aligned} \overline{\boldsymbol{q_1} * \boldsymbol{q_2}} &= \begin{pmatrix} \alpha_1\alpha_2 - \vec{v_1} \cdot \vec{v_2} \\ -(\alpha_2\vec{v_1} + \alpha_1\vec{v_2} + \vec{v_1} \times \vec{v_2}) \end{pmatrix} \\ &= \begin{pmatrix} \alpha_1\alpha_2 - \vec{v_1} \cdot \vec{v_2} \\ (-\alpha_2\vec{v_1}) + (-\alpha_1\vec{v_2}) + (-\vec{v_2}) \times (-\vec{v_1}) \end{pmatrix} = \overline{\boldsymbol{q_2}} * \overline{\boldsymbol{q_1}} \quad . \end{aligned} \tag{A.8}$$

Multipliziert man eine Quaternion mit ihrem Konjugierten, so erhält man das Quadrat des *Betrages* der Quaternion. Dabei handelt es sich um einen positiven Skalar, das heißt eine reelle Zahl ≥ 0, da im Skalarteil nach der Multiplikation eine Summe von Quadraten steht und der Vektorteil Null beträgt.

$$|\boldsymbol{q}| = \sqrt{\boldsymbol{q} * \overline{\boldsymbol{q}}} = \sqrt{\alpha^2 + x^2 + y^2 + z^2} = \sqrt{\alpha^2 + \vec{v}^{\,2}} \tag{A.9}$$

Als *Einheitsquaternionen* bezeichnen wir alle Quaternionen mit $|\boldsymbol{q}| = 1$. Das Produkt zweier Einheitsquaternionen ist wieder eine Einheitsquaternion. Als Teilmenge des Quaternionen-Schiefkörpers bilden sie damit eine Gruppe bezüglich der Mutliplikation.

$$\begin{aligned} |\boldsymbol{q_1} * \boldsymbol{q_2}|^2 &= (\boldsymbol{q_1} * \boldsymbol{q_2}) * \overline{(\boldsymbol{q_1} * \boldsymbol{q_2})} = \boldsymbol{q_1} * \boldsymbol{q_2} * \overline{\boldsymbol{q_2}} * \overline{\boldsymbol{q_1}} \\ &= \boldsymbol{q_1} * 1 * \overline{\boldsymbol{q_1}} = \boldsymbol{q_1} * \overline{\boldsymbol{q_1}} = 1 \end{aligned} \tag{A.10}$$

A.2 Darstellung von Drehungen

Drehungen von Vektoren im dreidimensionalen Raum können durch Einheitsquaternionen realisiert werden. Der Einheitsvektor $\vec{e}$ mit $\vec{e} \cdot \vec{e} = 1$ repräsentiere die Achse, um die gedreht werden soll, und

ϕ sei der dazugehörige Winkel. Die Quaternion für die Drehung ist dann

$$\boldsymbol{q} = \begin{pmatrix} \cos(\frac{\phi}{2}) \\ \vec{e}\,\sin(\frac{\phi}{2}) \end{pmatrix} \quad . \tag{A.11}$$

Durch Bildung des Betrages sieht man sofort, dass dies eine Einheitsquaternion ist. Weiterhin seien $\vec{s}$ der zu drehende Vektor und $\vec{s\prime}$ der Vektor nach der Drehung. Die zu $\vec{s}$ gehörende Quaternion $\boldsymbol{s}$ enthält $\vec{s}$ als seinen Vektorteil und hat einen Skalarteil von Null. Die Drehung wird dann durch die Rechnung

$$\begin{pmatrix} 0 \\ \vec{s\prime} \end{pmatrix} = \boldsymbol{s\prime} = \boldsymbol{q} * \boldsymbol{s} * \overline{\boldsymbol{q}} \tag{A.12}$$

durchgeführt. Wir wollen $\vec{s\prime}$ für den allgemeinen Fall ausrechnen.

$$\begin{aligned}
\begin{pmatrix} 0 \\ \vec{s\prime} \end{pmatrix} &= \begin{pmatrix} \cos(\frac{\phi}{2}) \\ \vec{e}\,\sin(\frac{\phi}{2}) \end{pmatrix} * \begin{pmatrix} 0 \\ \vec{s} \end{pmatrix} * \begin{pmatrix} \cos(\frac{\phi}{2}) \\ -\vec{e}\,\sin(\frac{\phi}{2}) \end{pmatrix} \\
&= \begin{pmatrix} -\vec{e}\cdot\vec{s}\,\sin(\frac{\phi}{2}) \\ \vec{s}\,\cos(\frac{\phi}{2}) + \vec{e}\times\vec{s}\,\sin(\frac{\phi}{2}) \end{pmatrix} * \begin{pmatrix} \cos(\frac{\phi}{2}) \\ -\vec{e}\,\sin(\frac{\phi}{2}) \end{pmatrix} \\
&= \begin{pmatrix} -\vec{e}\cdot\vec{s}\,\sin(\frac{\phi}{2})\cos(\frac{\phi}{2}) + \vec{e}\cdot\vec{s}\,\sin(\frac{\phi}{2})\cos(\frac{\phi}{2}) + (\vec{e}\times\vec{s})\cdot\vec{e}\,\sin^2(\frac{\phi}{2}) \\ \vec{s}(\cos^2(\frac{\phi}{2}) - \sin^2(\frac{\phi}{2})) + \vec{e}\times\vec{s}\,2\sin(\frac{\phi}{2})\cos(\frac{\phi}{2}) + \vec{e}(\vec{e}\cdot\vec{s})2\sin^2(\frac{\phi}{2}) \end{pmatrix} \\
&= \begin{pmatrix} 0 \\ \vec{s}\,\cos(\phi) + \vec{e}\times\vec{s}\,\sin(\phi) + \vec{e}(\vec{e}\cdot\vec{s})(1-\cos(\phi)) \end{pmatrix}
\end{aligned} \tag{A.13}$$

Das Ergebnis ist identisch mit dem Produkt $\mathbb{D}\vec{s}$ aus der allgemeinen Drehmatrix und dem zu drehenden Vektor. Die allgemeine Drehmatrix ist auf der folgenden Seite in (A.15) zu sehen.

Soll erst die Drehung $\boldsymbol{q_1}$ und danach $\boldsymbol{q_2}$ ausgeführt werden, so ist offensichtlich

$$\boldsymbol{s\prime} = \boldsymbol{q_2} * \boldsymbol{q_1} * \boldsymbol{s} * \overline{\boldsymbol{q_1}} * \overline{\boldsymbol{q_2}} \tag{A.14}$$

zu rechnen. Aufgrund von Gleichung (A.8) können wir im Algorithmus Rechenzeit sparen, indem

wir zunächst $\boldsymbol{q} = \boldsymbol{q_2} * \boldsymbol{q_1}$ berechnen, uns dieses Ergebnis merken und sein konjugiertes $\overline{\boldsymbol{q}} = \overline{\boldsymbol{q_1}} * \overline{\boldsymbol{q_2}}$ für die letzten beiden Faktoren einsetzen.

A.3 Vergleich mit Drehmatrizen

Beim Vergleich zwischen der Drehung durch Quaternionen und der bekannteren Drehung durch Matrizen zeigen sich Unterschiede im Speicherbedarf und beim Aufwand für die verschiedenen Operationen.

A.3.1 Speicherbedarf

In dem Programm werden die vier Komponenten der Quaternionen als Gleitkommazahlen vom Typ `double` gespeichert. Für eine Drehmatrix der Größe 3×3 würde mit neun statt nur vier Einträgen mehr als doppelt so viel Arbeitsspeicher benötigt.

A.3.2 Konstruktionsaufwand

In jedem Integrationsschritt müssen die Drehoperatoren komplett neu konstruiert werden. Sowohl für die Drehmatrizen als auch für die Einheitsquaternionen benötigen wir zunächst die Drehachse als Einheitsvektor $\vec{e}$ und den Drehwinkel ϕ. Die Konstruktion der Quaternion geschieht nach Gleichung (A.11). Eine Drehmatrix wird im allgemeinen Fall durch

$$\mathbb{D} = \begin{pmatrix} \mathfrak{c} + e_1^2\,(1-\mathfrak{c}) & e_1 e_2\,(1-\mathfrak{c}) - e_3\mathfrak{s} & e_1 e_3\,(1-\mathfrak{c}) + e_2\mathfrak{s} \\ e_2 e_1\,(1-\mathfrak{c}) + e_3\mathfrak{s} & \mathfrak{c} + e_2^2\,(1-\mathfrak{c}) & e_2 e_3\,(1-\mathfrak{c}) - e_1\mathfrak{s} \\ e_3 e_1\,(1-\mathfrak{c}) - e_2\mathfrak{s} & e_3 e_2\,(1-\mathfrak{c}) + e_1\mathfrak{s} & \mathfrak{c} + e_3^2\,(1-\mathfrak{c}) \end{pmatrix} \tag{A.15}$$

erzeugt. Unter Ausnutzung von Hilfsvariablen reicht hier, wie auch bei der Quaternion, die einmalige Berechnung von $\mathfrak{c} = \cos\phi$ und $\mathfrak{s} = \sin\phi$ aus. Möglicherweise böte es sich auch an, die Differenz $(1 - \cos\phi)$ nur einmal zu bilden. Dann sind für die Berechnung eines Diagonalelements je zwei Multiplikationen und eine Addition nötig sowie je drei Multiplikationen und eine Addition für die anderen Einträge. In der Summe kommt man so auf 24 Multiplikationen und zehn Additionen. Dem stehen lediglich drei Multiplikationen im Vektorteil sowie die im Vorfeld erfolgte Halbierung des Winkels ϕ bei der Quaternion gegenüber. Der Aufwand ist bei Drehmatrizen also etwa achtmal so groß.

A.3.3 Verkettung zweier Drehungen

Zwei Drehmatrizen werden durch Matrizenmultiplikation miteinander verkettet. Für jedes Element der resultierenden Matrix müssen drei Multiplikationen und zwei Additionen durchgeführt werden. Insgesamt kommen wir so auf 27 Multiplikationen und 18 Additionen. Die Verkettung der Einheitsquaternionen geschieht entsprechend durch Quaternionenmultiplikation. Hier müssen für jedes Element vier Multiplikationen und drei Additionen durchgeführt werden. In der Summe kommen wir also mit nur 16 Multiplikationen und zwölf Additionen aus.

A.3.4 Drehung eines Vektors

An dieser Stelle sind Drehmatrizen schneller als die Einheitsquaternionen. Für die Multiplikation der Matrix mit dem Vektor sind insgesamt neun Multiplikationen und sechs Additionen nötig. Demgegenüber müssen gemäß Gleichung (A.12) zwei Quaternionenmultiplikationen durchgeführt werden, also insgesamt 32 Multiplikationen und 24 Additionen.

Die Langsamkeit der Einheitsquaternionen bei der letztlichen Drehung wird in dem in dieser Arbeit betrachteten Anwendungsbereich von dem Geschwindigkeitsgewinn durch die schnelle Konstruktion der Drehoperatoren ungefähr ausgeglichen. Einheitsquaternionen sind gegenüber den Drehmatrizen vor allem dann im Vorteil, wenn viele Drehungen miteinander verkettet werden müssen.

B Messung der Laufzeiten

Integrator	ST2	ST4	FR4	OFR4	RK4
Schritte	1 000 000	100 000	100 000	100 000	100 000
Zeit (Sekunden)	88,186	31,846	20,189	25,887	12,959
	88,547	31,596	20,249	25,877	11,977
	88,197	31,770	20,319	25,977	11,797
	88,137	31,835	20,209	25,907	13,269
	88,247	31,585	20,289	25,897	13,599
	87,937	31,826	20,238	25,867	13,199
	88,207	31,706	20,238	25,877	13,740
	88,057	31,605	20,179	25,857	12,337
	88,297	32,066	20,239	25,867	11,276
	87,936	31,616	20,209	25,887	12,227
Mittelwert	88,175	31,745	20,236	25,890	12,638
Standardabweichung	0,179	0,154	0,043	0,034	0,831

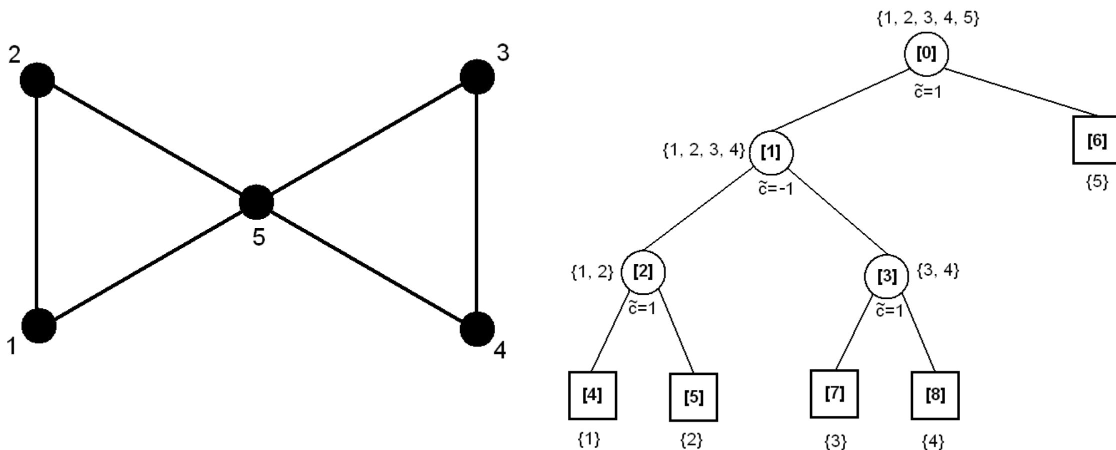
Tabelle 4: Diese Tabelle zeigt die einzelnen Werte, die bei der Bestimmung der Laufzeiten der verschiedenen Integratoren in Kapitel 4.5 gemessen wurden. Außerdem ist für jeden Integrator der errechnete Mittelwert und die Standardabweichung aufgeführt.

C Implementierte Spinsysteme

Neben dem Fe_{30}-System wurden einige weitere Spinsysteme implementiert. Die Beschreibung eines Systems erfolgt anhand seiner Konstruktionsbäume, die in der Datei `tree.txt` festgelegt sind.

C.1 Fliege

Das integrable System der Fliege kann durch den Konstruktionsbaum beschrieben werden, der in Abbildung 1 bei der Einführung der Konstruktionsbäume angegeben wurde. Um diesen Konstruktionsbaum in `tree.txt` zu definieren, werden alle Knoten des Baumes durchnummeriert. Da das System aus $N = 5$ Spins besteht, sind $2N - 1 = 9$ Knoten zu nummerieren. Die Wurzel erhält die Nummer 0 und die Nummern 1 bis $N - 2 = 3$ werden an die anderen inneren Knoten verteilt. Die Nummern $N - 1 = 4$ bis $2N - 2 = 8$ werden an die Blätter des Baumes vergeben. Außerdem ersetzen wir die Kopplungskonstanten c durch die reduzierten Kopplungskonstanten $\tilde{c}$. Somit erhalten wir nun diese Darstellung des Kontruktionsbaumes:



Für dieses System sieht die Datei `tree.txt` folgendermaßen aus:

```
5
1
1 6  1.0
2 3 -1.0
4 5  1.0
7 8  1.0
```

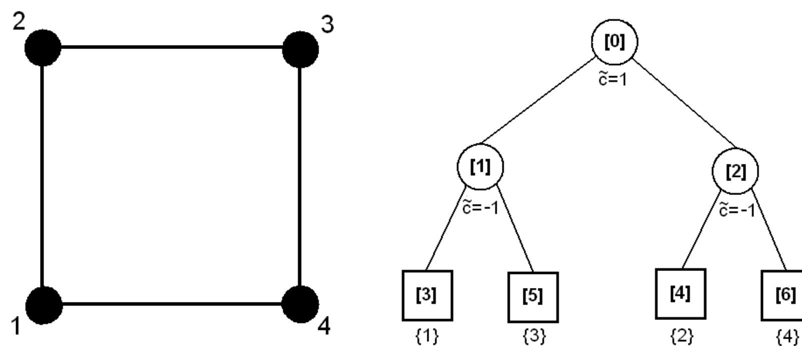
In der ersten Zeile wird die Anzahl der Spins mit 5 angegeben. Die zweite Zeile legt die Anzahl der beschriebenen Bäume auf 1 fest. Die letzten vier Zeilen beschreiben jeweils einen der inneren Knoten 0 bis 3. Angegeben sind die Nummer des linken Sohnes, die Nummer des rechten Sohnes und als letztes die reduzierte Kopplungskonstante.

Wird diese Datei von der Funktion `buildTree()` ausgelesen, so werden die Angaben benutzt, um die Zeiger der `NODE`s und der `LEAF`s den richtigen `QUAT`s zuzuordnen. Die verwendete Nummerierung der Knoten entspricht dabei genau den Indizes in dem Array der `QUAT`s.

Entsprechend diesem Muster wird auch für die Konstruktionsbäume anderer Systeme die Datei `tree.txt` erstellt.

C.2 Quadrat

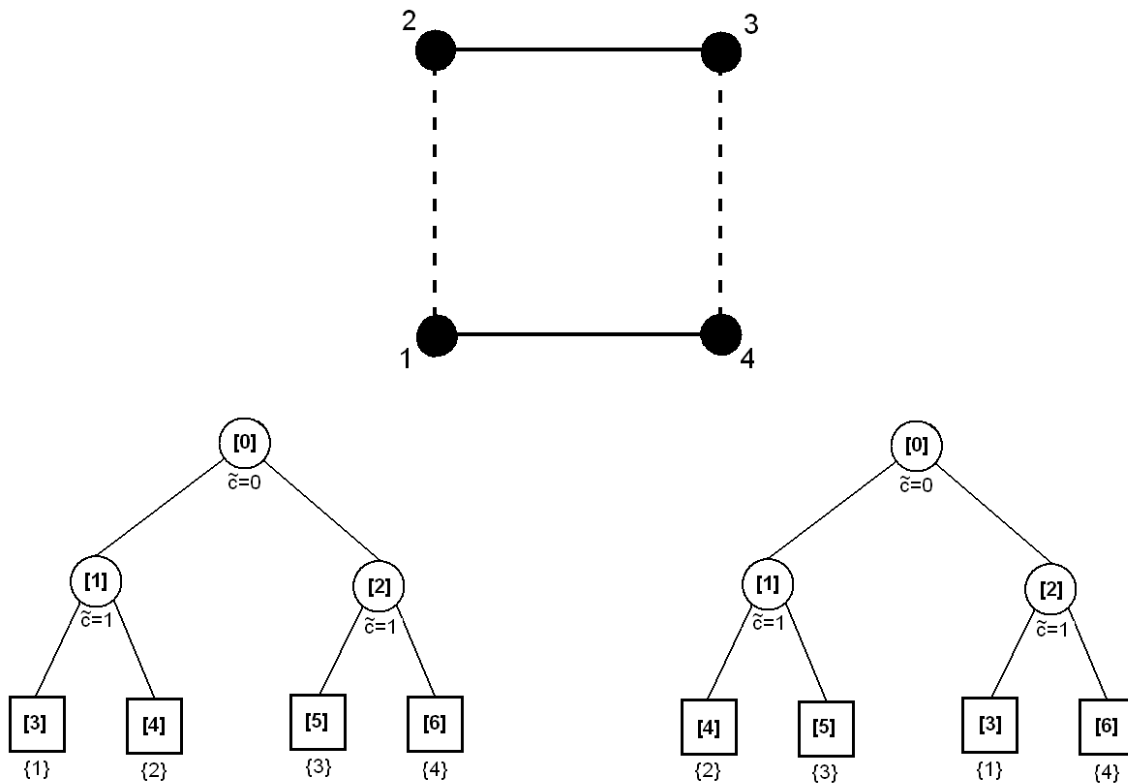
Das integrable Spinquadrat hat diesen Konstruktionsbaum:



Die Datei `tree.txt` sieht damit folgendermaßen aus:

```
4
1
1 2  1.0
3 5 -1.0
4 6 -1.0
```

Betrachten wir das Spinquadrat als aus zwei Paaren von Dimeren zusammengesetzt, so werden in `tree.txt` zwei Konstruktionsbäume definiert. Die Bäume sehen folgendermaßen aus:



Und dies ist die Darstellung der beiden Bäume in der Datei `tree.txt`:

```
4
2

1 2 0.0
3 4 1.0
5 6 1.0

1 2 0.0
4 5 1.0
6 3 1.0
```

Leerzeilen werden von dem Programm ignoriert und dienen nur der Übersichtlichkeit.

C.3 Fe_{30}

Das Fe_{30}-System wird gemäß Abbildung 4 in sechs Fliegen und acht Dreiecke zerlegt. Die zugehörigen Konstruktionsbäume sind bei so großen Systemen sehr umfangreich. Ein solcher Baum setzt sich aus mehreren Teilbäumen zusammen, die den Bäumen der Teilsysteme Fliege bzw. Dreieck entsprechen. Die Datei `tree.txt` sieht für das Fe_{30}-System folgendermaßen aus:

```
30
2

 1  2  0.0
 3  4  0.0
 5  9  0.0
13 17  0.0
21 25  0.0

 6 29  1.0
 7  8 -1.0
30 34  1.0
33 38  1.0

10 36  1.0
11 12 -1.0
31 32  1.0
42 43  1.0
```

```
14 40  1.0
15 16 -1.0
35 41  1.0
39 50  1.0

18 45  1.0
19 20 -1.0
37 44  1.0
46 53  1.0

22 49  1.0
23 24 -1.0
47 48  1.0
54 58  1.0

26 56  1.0
27 28 -1.0
51 55  1.0
52 57  1.0
```

```
 1  6  0.0
 2  3  0.0
 4  5  0.0

29 36  0.0
40 45  0.0
49 56  0.0

 7  8  0.0
 9 10  0.0
11 12  0.0

13 15  0.0
17 19  0.0
21 23  0.0
25 27  0.0

14 30  1.0
31 35  0.0
```

```
16 32  1.0
33 37  0.0

18 34  1.0
39 48  0.0

20 38  1.0
46 47  0.0

22 41  1.0
42 51  0.0

24 43  1.0
44 52  0.0

26 50  1.0
54 55  0.0

28 53  1.0
57 58  0.0
```

D Quellcode des erstellten Programms

Der dargestellte Quellcode zeigt die Konfiguration des Programms zur symplektischen Integration. Um den Runge-Kutta-Integrator zu benutzen, ist in der Datei `main.c` in Zeile 100 das Kommando `calculate` durch `calculateRK4` zu ersetzen.

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  /*define the structs QUAT, NODE, LEAF*/
6  /*and the basic funtions            */
7  #include "structures.h"
8
9  #define FILENAMELENGTH 30
10
11 /*global variables*/
12 int nSpins;            /*the number of spins*/
13 int nTrees;            /*the number of different trees*/
14 int nSteps;            /*the total number of steps to be calculated*/
15 int nSubSteps;         /*number of substeps in each step due to decomposition*/
16 double stepSize;       /*stepSize * nSteps == total time to be simulated*/
17 double *subStepSize;   /*the values of the various substeps*/
18 QUAT  *quats;          /*Array holding all the (2*nSpins-1) QUATS aka spins*/
19 NODE **nodes;          /*One array of nSpins-1 nodes for each tree*/
20 LEAF **leaves;         /*One array of nSpins leaves for each tree*/
21 int i,j,k;             /*Counting variables*/
22 char treeFile[FILENAMELENGTH];      /*String holding filename*/
23 char spinFile[FILENAMELENGTH];      /*String holding filename*/
24 char decompFile[FILENAMELENGTH];    /*String holding filename*/
25 char txtOutFile[FILENAMELENGTH];    /*String holding filename*/
26 char binOutFile[FILENAMELENGTH];    /*String holding filename*/
27 char matlabFile[FILENAMELENGTH];    /*String holding filename*/
28 char matlabFile2[FILENAMELENGTH];   /*String holding filename*/
29 char matrixFile[FILENAMELENGTH];    /*String holding filename*/
30 double exactEnergy;
31 QUAT exactTotalSpin;
32 double **jmatrix, **j2matrix;       /* NxN J and J^2 matrix */
33 clock_t start, end;                 /*Measure the required time*/
34
35 /*These variables are used by the RK4-Integrator only.*/
36 QUAT *spinsTmp, *spinDot_0, *spinDot_A, *spinDot_B, *spinDot_C;
37
38 /*define the function buildTree*/
39 #include "treebuilder.h"
40
41 /*define the function buildJmatrix*/
42 #include "matrixbuilder.h"
```

```
43
44 /*define the function readStartSpins*/
45 #include "spinreader.h"
46
47 /*define the function setDecomp*/
48 #include "decsetter.h"
49
50 /*define the functions calculate, writeMFile and auxiliary funtions*/
51 #include "calculus.h"
52
53 int main(int argc, char *argv[]){
54
55   if(argc<2) {
56     printf("Usage: Symplec treeFile spinFile decompFile outputFile matrixFile\n"
57            "   or: Symplec directory");
58     exit(1);
59   }
60
61   if(argc==6) {
62     strcpy(treeFile,   argv[1]);
63     strcpy(spinFile,   argv[2]);
64     strcpy(decompFile, argv[3]);
65     strcpy(txtOutFile, argv[4]);
66     strcat(strcpy(binOutFile, argv[4]), ".bin");
67     strcat(strcpy(matlabFile, argv[4]), ".m");
68     strcat(strcpy(matlabFile2,argv[4]), "2.m");
69     strcpy(matrixFile, argv[5]);
70   }
71   else {
72     strcat(strcpy(treeFile,   argv[1]), "\\tree.txt");
73     strcat(strcpy(spinFile,   argv[1]), "\\spins.txt");
74     strcat(strcpy(decompFile, argv[1]), "\\dec.txt");
75     strcat(strcpy(txtOutFile, argv[1]), "\\out.txt");
76     strcat(strcpy(binOutFile, argv[1]), "\\out.bin");
77     strcat(strcpy(matlabFile, argv[1]), "\\symplImport.m");
78     strcat(strcpy(matlabFile2,argv[1]), "\\symplImport2.m");
79     strcat(strcpy(matrixFile, argv[1]), "\\matrix.txt");
80   }
81
82   buildTree(treeFile);
83
84   buildJmatrix(matrixFile);
85
86   readStartSpins(spinFile);
87
88   setDecomp(decompFile);
89
90   {
91     char answer;
92     printf("Are the values above correct? Enter 'y' to start calculation! ");
93     scanf("%c",&answer);
94     if (answer != 'y') {
```

```
95          printf("Program aborted!!\n");
96          exit(0);
97        }
98      }
99
100     calculate(txtOutFile, binOutFile);
101
102     printf("The calculation took %f seconds CPU time\n\a",
103                                              (float)(end - start)/CLOCKS_PER_SEC);
104     printf("Results written to %s and %s\n", txtOutFile, binOutFile);
105
106     if(argc==6) {
107       writeMFiles(matlabFile, matlabFile2, binOutFile);
108     }
109     else {
110       writeMFiles(matlabFile, matlabFile2, "out.bin");
111     }
112
113     return 0;
114
115   }
```

structures.h

```
#define QUAT struct quaternion
#define NODE struct treenode
#define LEAF struct treeleaf


QUAT {
  double re;
  double im;
  double jm;
  double km;
};

NODE {
  double coupling;
  QUAT *data;
  QUAT *son1;
  QUAT *son2;
};

LEAF {
  QUAT *data;
  QUAT *father;
};

void addSpins(QUAT *a , QUAT *b, QUAT *c)
{
  /* Adding of real part omitted since it should equal zero anyway */
  c->re = 0.0;
  c->im = a->im + b->im;
  c->jm = a->jm + b->jm;
  c->km = a->km + b->km;
}

void multQuats(QUAT *a , QUAT *b)
{
  /*QUAT b is overwritten by the result!*/

  double real, imag, jmag;
  /*The k-part can be set directly.*/

  real  = a->re * b->re - a->im * b->im - a->jm * b->jm - a->km * b->km;
  imag  = a->im * b->re + a->re * b->im - a->km * b->jm + a->jm * b->km;
  jmag  = a->jm * b->re + a->km * b->im + a->re * b->jm - a->im * b->km;

  b->km = a->km * b->re - a->jm * b->im + a->im * b->jm + a->re * b->km;

  b->re = real;
  b->im = imag;
  b->jm = jmag;
}
```

```
51
52  void multQuatsConj(QUAT *a , QUAT *b)
53  {
54    /*The Conjugate of QUAT b is used to multiply and           */
55    /*QUAT a is overwritten by the result instead of QUAD b!!   */
56
57    double real, imag, jmag;
58    /*The k-part can be set directly.*/
59
60    real  = a->re * b->re + a->im * b->im + a->jm * b->jm + a->km * b->km;
61    imag  = a->im * b->re - a->re * b->im + a->km * b->jm - a->jm * b->km;
62    jmag  = a->jm * b->re - a->km * b->im - a->re * b->jm + a->im * b->km;
63
64    a->km = a->km * b->re + a->jm * b->im - a->im * b->jm - a->re * b->km;
65
66    a->re = real;
67    a->im = imag;
68    a->jm = jmag;
69  }
70
71  void factorizeQuat(QUAT *a, double f)
72  {
73    /*Real part omitted since it should equal zero anyway */
74    a->im *= f;
75    a->jm *= f;
76    a->km *= f;
77  }
78
79  void copyQuat(QUAT *a , QUAT *b) {
80    /* copy a to b */
81    b->re = a->re;
82    b->im = a->im;
83    b->jm = a->jm;
84    b->km = a->km;
85  }
86
87  double scalProd(QUAT *a , QUAT *b) {
88    return a->im * b->im + a->jm * b->jm + a->km * b->km;
89  }
90
91  void vectProd(QUAT *a , QUAT *b) {
92    /*QUAT a is overwritten by the result*/
93    double x,y;
94
95    x = a->jm * b->km - a->km * b->jm;
96    y = a->km * b->im - a->im * b->km;
97
98    a->km = a->im * b->jm - a->jm * b->im;
99
100   a->im = x;
101   a->jm = y;
102 }
```

```
103
104 void addToQuat(QUAT *a , QUAT *b, double factor) {
105   a->im += factor * b->im;
106   a->jm += factor * b->jm;
107   a->km += factor * b->km;
108 }
109
110
111 void setQuatZero(QUAT *a)
112 {
113   a->re = 0.0;
114   a->im = 0.0;
115   a->jm = 0.0;
116   a->km = 0.0;
117 }
118
119 double absOfQuat(QUAT *a)
120 {
121   /*Real part omitted since it should equal zero anyway */
122   return sqrt(pow(a->im,2) + pow(a->jm,2) + pow(a->km,2));
123 }
```

treebuilder.h

```
1  #define MEM_MSG printf("Not enough memory avaible!\n")
2
3  void buildTree(char *fileName){
4
5    FILE *inpFile;
6    char txtLine[10];
7    int i,j,sonNum;
8
9    if ((inpFile = fopen(fileName, "r")) ==NULL){
10     printf("Error opening tree file\n");
11     exit(1);
12   }
13
14   /*First line declares the number of spins in the system*/
15   fscanf(inpFile, "%9s", txtLine);
16   nSpins = atoi(txtLine);
17   printf("Number of spins is set to %i\n", nSpins);
18
19   /*Second line declares the number of trees describing the graph*/
20   fscanf(inpFile, "%9s", txtLine);
21   nTrees = atoi(txtLine);
22   printf("Number of trees is set to %i\n", nTrees);
23
24   quats  = (QUAT *) malloc((nSpins * 2 - 1) * sizeof(QUAT));
25   nodes  = (NODE **)malloc(nTrees * sizeof(NODE *));
26   leaves = (LEAF **)malloc(nTrees * sizeof(LEAF *));
27
28   /*These variables are used by the RK4-Integrator only.*/
29   spinsTmp   = (QUAT *) malloc(nSpins * sizeof(QUAT));
30   spinDot_0  = (QUAT *) malloc(nSpins * sizeof(QUAT));
31   spinDot_A  = (QUAT *) malloc(nSpins * sizeof(QUAT));
32   spinDot_B  = (QUAT *) malloc(nSpins * sizeof(QUAT));
33   spinDot_C  = (QUAT *) malloc(nSpins * sizeof(QUAT));
34
35   if(NULL == leaves || NULL == nodes || NULL == quats) {
36       MEM_MSG;
37       exit(1);
38    }
39
40   for(i=0; i<nTrees; i++) {
41     nodes[i]  = (NODE *)malloc((nSpins - 1) * sizeof(NODE));
42     leaves[i] = (LEAF *)malloc( nSpins       * sizeof(LEAF));
43
44     if(NULL == leaves[i] || NULL == nodes[i]) {
45       MEM_MSG;
46       exit(1);
47     }
48
49     for(j=0; j<nSpins-1; j++) {
50       /*Set data pointer of the node*/
```

```
51         nodes[i]->data = &quats[j];
52 
53         /*Set first son's pointer of the node*/
54         fscanf(inpFile, "%9s", txtLine);
55         sonNum = atoi(txtLine);
56         nodes[i]->son1 = &quats[sonNum];
57         /*If son is a leaf, set the leaf's pointers*/
58         if(sonNum > nSpins-2) {
59           leaves[i][sonNum-nSpins+1].data   = &quats[sonNum];
60           leaves[i][sonNum-nSpins+1].father = &quats[j];
61         }
62 
63         /*Set second son's pointer of the node*/
64         fscanf(inpFile, "%9s", txtLine);
65         sonNum = atoi(txtLine);
66         nodes[i]->son2 = &quats[sonNum];
67         /*If son is a leaf, set the leaf's pointers*/
68         if(sonNum > nSpins-2) {
69           leaves[i][sonNum-nSpins+1].data   = &quats[sonNum];
70           leaves[i][sonNum-nSpins+1].father = &quats[j];
71         }
72 
73         /*set coupling constant of the node*/
74         fscanf(inpFile, "%9s", txtLine);
75         nodes[i]->coupling = atof(txtLine);
76 
77         /*go to next node of the i'th tree*/
78         nodes[i]++;
79         /*This sets the pointers to the end of nodes-arrays*/
80       }
81     }
82     printf("Trees have been constructed\n");
83     fclose(inpFile);
84 }
```

matrixbuilder.h

```
#define MEM_MSG printf("Not enough memory avaible!\n")

void buildJmatrix(char *fileName){

  FILE *inpFile;
  char txtLine[10];
  int i,j;

  if ((inpFile = fopen(fileName, "r")) ==NULL){
    printf("Error opening matrix file\n");
    exit(1);
  }

  /*First line declares the number of spins in the system*/
  fscanf(inpFile, "%9s", txtLine);
  /*nSpins = atoi(txtLine);*/                    /*is already set by treebuilder*/
  printf("Will produce a %i-square J-matrix and J^2-matrix\n", nSpins);

  jmatrix = (double **)malloc(nSpins * sizeof(double *));

  if(NULL == jmatrix) {
      MEM_MSG;
      exit(1);
   }

  for(i=0; i<nSpins; i++) {
    jmatrix[i] = (double *) malloc(nSpins * sizeof(double));

    if(NULL == jmatrix[i]) {
      MEM_MSG;
      exit(1);
    }
    for(j=0; j<nSpins; j++) {
      /*Set the matrix data*/
      fscanf(inpFile, "%9s", txtLine);
      jmatrix[i][j] = atof(txtLine);
    }
  }
  printf("J-matrix has been constructed\n");

  fclose(inpFile);

  /*Now, calculated the J^2 matrix*/

  j2matrix = (double **)malloc(nSpins * sizeof(double *));

  if(NULL == j2matrix) {
      MEM_MSG;
      exit(1);
   }
```

```
51
52    for(i=0; i<nSpins; i++) {
53      j2matrix[i] = (double *) malloc(nSpins * sizeof(double));
54
55      if(NULL == j2matrix[i]) {
56        MEM_MSG;
57        exit(1);
58      }
59
60      for(j=0; j<nSpins; j++) {
61        /*Set the J^2 matrix data*/
62        j2matrix[i][j] = 0.0;
63        for(k=0; k<nSpins; k++) {
64          j2matrix[i][j] += jmatrix[i][k] * jmatrix[k][j];
65        }
66      }
67    }
68    printf("J^2-matrix has been constructed\n");
69  }
```

spinreader.h

```
1
2  void readStartSpins(char *fileName){
3
4    FILE *inpFile;
5    char txtLine[10];
6    int i;
7    QUAT *currentSpin;
8
9    if ((inpFile = fopen(fileName, "r")) ==NULL){
10     printf("Error opening spins file\n");
11     exit(1);
12   }
13
14   for(i=0; i<nSpins; i++) {
15     /* Since the leaves of all trees point to the same QUATs, */
16     /* setting leaves[0][i] is sufficient                      */
17     currentSpin = leaves[0][i].data;
18     currentSpin->re = 0.0;
19     fscanf(inpFile, "%9s", txtLine);
20     currentSpin->im = atof(txtLine);
21     fscanf(inpFile, "%9s", txtLine);
22     currentSpin->jm = atof(txtLine);
23     fscanf(inpFile, "%9s", txtLine);
24     currentSpin->km = atof(txtLine);
25
26    /*Observations showed that normalizing twice minimizes the offset*/
27     factorizeQuat(currentSpin, 1.0/absOfQuat(currentSpin));
28     factorizeQuat(currentSpin, 1.0/absOfQuat(currentSpin));
29
30   }
31
32   printf("Starting spins have been read\n");
33   fclose(inpFile);
34 }
```

decsetter.h

```
1
2  void setDecomp(char *fileName){
3
4    FILE *inpFile;
5    int i;
6    char txtLine[10];
7    double subStepSum = 0.0;
8
9    if ((inpFile = fopen(fileName, "r")) ==NULL){
10     printf("Error opening decomposition file\n");
11     exit(1);
12   }
13
14   /*First line declares the number of timesteps to be calculated*/
15   fscanf(inpFile, "%9s", txtLine);
16   nSteps = atoi(txtLine);
17
18   /*Second line declares the total time to  be simulated*/
19   fscanf(inpFile, "%9s", txtLine);
20   stepSize = atof(txtLine)/nSteps;
21
22   printf("Will calculate %i steps ", nSteps);
23   printf("of %g time.\n", stepSize);
24   printf("Total simulated time is %g\n", stepSize * nSteps);
25
26   /*Third line selects a predifined decomposition*/
27   /*(or this and the remaining lines define an explicite decomposition)*/
28   fscanf(inpFile, "%9s", txtLine);
29   if(!strcmp(txtLine,"ST1")) {
30     nSubSteps = 2;
31     subStepSize = (double *)malloc((nSubSteps) * sizeof(double));
32     /*a1*/ subStepSize[0] = 1.0;
33     /*b1*/ subStepSize[1] = 1.0;
34   }
35   else if(!strcmp(txtLine,"ST2")) {
36     nSubSteps = 3;
37     subStepSize = (double *)malloc((nSubSteps) * sizeof(double));
38     /*a1*/ subStepSize[0] = 0.5;
39     /*b1*/ subStepSize[1] = 1.0;
40     /*a2*/ subStepSize[2] = 0.5;
41     /*b2 = 0*/
42   }
43   else if(!strcmp(txtLine,"ST4")) {
44     nSubSteps = 11;
45     subStepSize = (double *)malloc((nSubSteps) * sizeof(double));
46     double p = 1.0 / (4.0 - pow(4.0, 1.0/3.0));
47     /*a1*/ subStepSize[ 0] =  p * 0.5;
48     /*b1*/ subStepSize[ 1] =  p;
49     /*a2*/ subStepSize[ 2] =  p;
50     /*b2*/ subStepSize[ 3] =  p;
```

```
51        /*a3*/ subStepSize[ 4] = (1.0 - 3.0 * p) * 0.5;
52        /*b3*/ subStepSize[ 5] =  1.0 - 4.0 * p;
53        /*a4*/ subStepSize[ 6] = (1.0 - 3.0 * p) * 0.5;
54        /*b4*/ subStepSize[ 7] =  p;
55        /*a5*/ subStepSize[ 8] =  p;
56        /*b5*/ subStepSize[ 9] =  p;
57        /*a6*/ subStepSize[10] =  p * 0.5;
58        /*b6 = 0*/
59      }
60      else if(!strcmp(txtLine,"FR4")) {
61        nSubSteps = 7;
62        subStepSize = (double *)malloc((nSubSteps) * sizeof(double));
63        double th = 1.0 / (2.0 - pow(2.0, 1.0/3.0));
64        /*a1*/ subStepSize[ 0] =  th * 0.5;
65        /*b1*/ subStepSize[ 1] =  th;
66        /*a2*/ subStepSize[ 2] = (1.0 - th) * 0.5;
67        /*b2*/ subStepSize[ 3] =  1.0 - 2.0 * th;
68        /*a3*/ subStepSize[ 4] = (1.0 - th) * 0.5;
69        /*b3*/ subStepSize[ 5] =  th;
70        /*a4*/ subStepSize[ 6] =  th * 0.5;
71        /*b4 = 0*/
72      }
73      else if(!strcmp(txtLine,"OFR4")) {
74        nSubSteps = 9;
75        subStepSize = (double *)malloc((nSubSteps) * sizeof(double));
76        double xi   =  0.17208656;
77        double chi  = -0.16162176;
78        double lmbd = -0.09156203;
79        /*a1*/ subStepSize[ 0] =  xi;
80        /*b1*/ subStepSize[ 1] = (1 - 2 * lmbd) * 0.5;
81        /*a2*/ subStepSize[ 2] =  chi;
82        /*b2*/ subStepSize[ 3] =  lmbd;
83        /*a3*/ subStepSize[ 4] =  1.0 - 2.0 * (xi + chi);
84        /*b3*/ subStepSize[ 5] =  lmbd;
85        /*a4*/ subStepSize[ 6] =  chi;
86        /*b4*/ subStepSize[ 7] = (1 - 2 * lmbd) * 0.5;
87        /*a5*/ subStepSize[ 8] =  xi;
88        /*b5 = 0*/
89      }
90      else{
91        /*The substeps are given explicitly and this is their quantity */
92        nSubSteps = atoi(txtLine);
93        /*Read nSubSteps values in the subStepSize array*/
94        subStepSize  = (double *)malloc((nSubSteps) * sizeof(double));
95        for(i=0; i<nSubSteps; i++) {
96          fscanf(inpFile, "%9s", txtLine);
97          subStepSize[i] = atof(txtLine);
98        }
99      }
100     /*decomposition is chosen now!*/
101     for(i=0; i<nSubSteps; i++) {
102       subStepSum += subStepSize[i];
```

```
103      }
104      printf("Decomposition values:\n", stepSize);
105      /*Ensure that the sum of  the subStepSize's equals (stepSize * nTrees)*/
106      for(i=0; i<nSubSteps; i++) {
107          subStepSize[i] *= (nTrees*stepSize/subStepSum);
108          printf("%g\t", subStepSize[i]/stepSize);
109      }
110      printf("\n");
111
112
113      printf("Decomposition parameters have been set\n");
114      fclose(inpFile);
115  }
```

calculus.h

```
void writeMFiles(char *mFileName, char *mFileName2, char *binFileName) {
  /* Write matlab input .m-Files */
  FILE *mFile, *mFile2;
  if ((mFile = fopen(mFileName, "w")) ==NULL){
    printf("Error opening file\n");
    exit(1);
  }
  fprintf(mFile, "nSteps = %i;\n", nSteps);
  fprintf(mFile, "time = linspace(0,%f,nSteps+1);\n", nSteps*stepSize);
  for (k=1; k<=nSpins; k++) {
    fprintf(mFile, "s%i = zeros(4, nSteps+1);\n", k);
  }
  fprintf(mFile, "file = fopen('%s','r');\n", binFileName);
  fprintf(mFile, "for i=1:(nSteps+1)\n");
  for (k=1; k<=nSpins; k++) {
    fprintf(mFile, "  s%i(:,i) = fread(file, 4, 'double');\n", k);
  }
  fprintf(mFile, "end\n");
  fprintf(mFile, "fclose(file);\n");
  fclose(mFile);

  if ((mFile2 = fopen(mFileName2, "w")) ==NULL){
    printf("Error opening file\n");
    exit(1);
  }
  fprintf(mFile2, "nSteps = %i;\n", nSteps);
  fprintf(mFile2, "time = linspace(0,%f,nSteps+1);\n", nSteps*stepSize);
  fprintf(mFile2, "nSpins = %i;\n", nSpins);
  fprintf(mFile2, "spindata = zeros(4*nSpins, nSteps+1);\n");
  fprintf(mFile2, "file = fopen('%s','r');\n", binFileName);
  fprintf(mFile2, "for i=1:(nSteps+1)\n");
  fprintf(mFile2, "  spindata(:,i) = fread(file, 4*nSpins, 'double');\n");
  fprintf(mFile2, "end\n");
  fprintf(mFile2, "fclose(file);\n");
  fclose(mFile2);

  printf("mFiles written to %s and %s\n", mFileName, mFileName2);
}

/*used by calcRK4Step*/
void calcDotted(QUAT *spins, QUAT *spinsDot) {
  int i, j;           /*counting integers*/
  /*for every spin...*/
  for(i=0; i<nSpins; i++) {
    setQuatZero(&spinsDot[i]);
    for(j=0; j<nSpins; j++) {
      /*... calc the sum of all neighbour spins...*/
      addToQuat(&spinsDot[i], &spins[j], jmatrix[i][j]);
    }
    /*and take the crossProduct of the original spin*/
```

```
51      vectProd(&spinsDot[i], &spins[i]);
52    }
53  }
54
55  void calcRK4Step(double timeStep) {
56    int i,j;                          /*counting integers*/
57    float absul, phi;                 /*helping variables*/
58
59    /*Calculating spinDot_0*/
60    for(i=0; i<nSpins; i++) {
61       copyQuat(leaves[0]->data, spinsTmp);
62       spinsTmp++;
63       leaves[0]++;
64    }
65    spinsTmp  -= nSpins;
66    leaves[0] -= nSpins;
67    calcDotted(spinsTmp, spinDot_0);
68
69    /*Calculating spinDot_A*/
70    for(i=0; i<nSpins; i++) {
71       copyQuat(leaves[0]->data, spinsTmp);
72       addToQuat(spinsTmp, spinDot_0, 0.5*timeStep);
73       spinsTmp++;
74       leaves[0]++;
75       spinDot_0++;
76    }
77    spinsTmp  -= nSpins;
78    leaves[0] -= nSpins;
79    spinDot_0 -= nSpins;
80    calcDotted(spinsTmp, spinDot_A);
81
82    /*Calculating spinDot_B*/
83    for(i=0; i<nSpins; i++) {
84       copyQuat(leaves[0]->data, spinsTmp);
85       addToQuat(spinsTmp, spinDot_A, 0.5*timeStep);
86       spinsTmp++;
87       leaves[0]++;
88       spinDot_A++;
89    }
90    spinsTmp  -= nSpins;
91    leaves[0] -= nSpins;
92    spinDot_A -= nSpins;
93    calcDotted(spinsTmp, spinDot_B);
94
95    /*Calculating spinDot_C*/
96    for(i=0; i<nSpins; i++) {
97       copyQuat(leaves[0]->data, spinsTmp);
98       addToQuat(spinsTmp, spinDot_B, timeStep);
99       spinsTmp++;
100      leaves[0]++;
101      spinDot_B++;
102   }
```

```
103    spinsTmp   -= nSpins;
104    leaves[0] -= nSpins;
105    spinDot_B -= nSpins;
106    calcDotted(spinsTmp, spinDot_C);
107
108    /*Calculating  new spins*/
109    for(i=0; i<nSpins; i++) {
110       addToQuat(spinDot_A, spinDot_B, 1.0);
111       addToQuat(spinDot_0, spinDot_A, 2.0);
112       addToQuat(spinDot_0, spinDot_C, 1.0);
113       addToQuat(leaves[0]->data, spinDot_0, timeStep/6.0);
114       spinDot_0++;
115       spinDot_A++;
116       spinDot_B++;
117       spinDot_C++;
118       leaves[0]++;
119    }
120    leaves[0] -= nSpins;
121    spinDot_0 -= nSpins;
122    spinDot_A -= nSpins;
123    spinDot_B -= nSpins;
124    spinDot_C -= nSpins;
125 }
126
127 /*one exact step for the subsystem defined by tree*/
128 void calcStep(int tree, double timeStep) {
129    int i;                              /*counting integers*/
130    float absul, phi;                   /*helping variables*/
131
132    /*Building sums of spins in nodes*/
133    for(i=0; i<nSpins-1; i++) {
134      /*Pointers are at end of arrays and we go backward*/
135      nodes[tree]--;
136      addSpins(nodes[tree]->son1, nodes[tree]->son2, nodes[tree]->data);
137    }
138
139    /*Construct Rotor QUADs in nodes*/
140    for(i=0; i<nSpins-1; i++){
141      absul = absOfQuat(nodes[tree]->data);
142      phi = nodes[tree]->coupling * absul * timeStep * 0.5;
143      if(phi==0.0){
144        factorizeQuat(nodes[tree]->data, 0.0);
145        nodes[tree]->data->re = 1.0;
146      }
147      else{
148        factorizeQuat(nodes[tree]->data, sin(phi)/absul);
149        nodes[tree]->data->re = cos(phi);
150      }
151
152      /*Pointers are at beginning of arrays and we go forward*/
153      nodes[tree]++;
154    }
```

```
155
156     /*Rewind pointers to beginning of arrays*/
157     nodes[tree] -= (nSpins-1);
158
159     /*Multiply each Rotor from left to its sons*/
160     for(i=0; i<nSpins-1; i++){
161       multQuats(nodes[tree]->data, nodes[tree]->son1);
162       multQuats(nodes[tree]->data, nodes[tree]->son2);
163       /*Pointers are at beginning of arrays and we go forward*/
164       nodes[tree]++;
165     }
166
167     /*finalize rotation by multipliying the leaves with their conj. fathers*/
168     for(i=0; i<nSpins; i++){
169       multQuatsConj(leaves[tree]->data, leaves[tree]->father);
170       /*Pointers are at beginning of arrays and we go forward*/
171       leaves[tree]++;
172     }
173
174     /*Rewind pointers to beginning of arrays*/
175     leaves[tree] -= nSpins;
176   }
177
178   /*used by projHamilton*/
179   double totalEnergy() {
180     int i, j;          /*counting integers*/
181     double energy = 0;
182     /*for every spin...*/
183     for(i=0; i<nSpins; i++) {
184       for(j=i; j<nSpins; j++) {
185         /*... add the neighbours with a higher number*/
186         energy += jmatrix[i][j] * scalProd(leaves[0][i].data, leaves[0][j].data);
187       }
188     }
189     return energy;
190   }
191
192   /*used by projTotalSpin*/
193   void totalSpin(QUAT *result) {
194     setQuatZero(result);
195     for(i=0; i<nSpins; i++) {
196       /*add all spins*/
197       addToQuat(result, leaves[0][i].data, 1.0);
198     }
199   }
200
201   /* This is the second-next neighbour weight    */
202   /*      \sum_ij J^2_ij s_i \cdot s_j            */
203   /* uses J^2-matrix contructed from J-matrix!!) */
204   /* used by projHamilton                        */
205   double weight() {
206     int i, j;          /*counting integers*/
```

```
207     double weight = 0.0;
208     /*for every spin...*/
209     for(i=0; i<nSpins; i++) {
210       for(j=0; j<nSpins; j++) {
211        /*... add the over-next neighbours with a higher number twice*/
212        weight += j2matrix[i][j] *scalProd(leaves[0][i].data, leaves[0][j].data);
213       }
214     }
215     return weight;
216   }
217
218   void projHamilton(double lambda) {
219       /*Projecting to nearly exact energy*/
220       /* Projecting vector n is taken from J-matrix */
221       /*for every spin...*/
222       for(i=0; i<nSpins; i++) {
223         for(j=0; j<nSpins; j++) {
224           /*... add the sum of all neighbour spins times lambda*/
225           addToQuat(leaves[0][i].data, leaves[0][j].data, jmatrix[i][j]*lambda);
226         }
227       }
228   }
229
230   void projTotalSpin(){
231     QUAT tempSpin;
232     totalSpin(&tempSpin);                              /*  S            */
233     addToQuat(&tempSpin, &exactTotalSpin, -1.0);   /*  S - S_0      */
234     factorizeQuat(&tempSpin, -1.0 / nSpins);         /* (S_0 - S) / N */
235     for(i=0; i<nSpins; i++) {
236       /*project the individual spins*/
237       addToQuat(leaves[0][i].data, &tempSpin, 1.0);
238     }
239   }
240
241   void normalizeSpins() {
242       /* Normalising spins to |s|=1 */
243       for(i=0; i<nSpins; i++) {
244         factorizeQuat(leaves[0][i].data, 1.0/absOfQuat(leaves[0][i].data));
245       }
246   }
247
248   /*Symplectic calculation*/
249   void calculate(char *txtFileName, char *binFileName) {
250
251     FILE *txtFile, *binFile;
252     int stepCount,subCount,tree,i,j;   /*counting integers*/
253
254     if ((txtFile = fopen(txtFileName, "w")) ==NULL){
255       printf("Error opening text output file\n");
256       exit(1);
257     }
258     if ((binFile = fopen(binFileName, "wb")) ==NULL){
```

```
259       printf("Error opening binary output file\n");
260       exit(1);
261     }
262
263     /*Print headline to txtFile*/
264     fprintf(txtFile, "time\t");
265     for(i=0; i<nSpins; i++){
266       fprintf(txtFile, "s_%i_x\t", i);
267       fprintf(txtFile, "s_%i_y\t", i);
268       fprintf(txtFile, "s_%i_z\t", i);
269     }
270     fprintf(txtFile, "\n");
271
272
273     /*Print starting spins at t=0 to txtFile and binFile*/
274     fprintf(txtFile, "%f\t", 0.0);
275     for(i=0; i<nSpins; i++){
276       fwrite(leaves[0]->data, sizeof(QUAT), 1, binFile);
277       fprintf(txtFile, "%f\t", leaves[0]->data->im);
278       fprintf(txtFile, "%f\t", leaves[0]->data->jm);
279       fprintf(txtFile, "%f\t", leaves[0]->data->km);
280       leaves[0]++;
281     }
282     fprintf(txtFile, "\n");
283     leaves[0] -= nSpins;
284
285     /*exact data for projectors*/
286     exactEnergy = totalEnergy();
287     totalSpin(&exactTotalSpin);
288
289     printf("Starting symplectic calculation..\n");
290     start = clock();
291
292     /*Calculation begins here*/
293     for(stepCount=0; stepCount<nSteps; stepCount++) {
294
295       /*One step in chosen decomposition*/
296       for(subCount=0; subCount<nSubSteps; subCount++) {
297         tree = subCount%nTrees;
298         calcStep(tree, subStepSize[subCount]);
299       }
300
301   /*insert desired projectors here*/
302   /*
303   for(i=0; i<10; i++) {
304   projHamilton( ( -totalEnergy() + exactEnergy) / weight() );
305   normalizeSpins();
306   projTotalSpin();
307   }
308   */
309       /*Print calculated spins to txtFile and binFile*/
310       fprintf(txtFile, "%f\t", (stepCount + 1) * stepSize);
```

```
311       for(i=0; i<nSpins; i++){
312         fwrite(leaves[0]->data, sizeof(QUAT), 1, binFile);
313         fprintf(txtFile, "%f\t", leaves[0]->data->im);
314         fprintf(txtFile, "%f\t", leaves[0]->data->jm);
315         fprintf(txtFile, "%f\t", leaves[0]->data->km);
316         leaves[0]++;
317       }
318       fprintf(txtFile, "\n");
319       leaves[0] -= nSpins;
320 
321     }
322 
323     end = clock();
324     printf("Calculation finished.\n");
325 
326     fclose(txtFile);
327     fclose(binFile);
328 
329   }
330 
331   /*Runge-Kutta-calculation*/
332   void calculateRK4(char *txtFileName, char *binFileName) {
333 
334     FILE *txtFile, *binFile;
335     int stepCount, tree, i, j;   /*counting integers*/
336 
337     if ((txtFile = fopen(txtFileName, "w")) ==NULL){
338       printf("Error opening text output file\n");
339       exit(1);
340     }
341     if ((binFile = fopen(binFileName, "wb")) ==NULL){
342       printf("Error opening binary output file\n");
343       exit(1);
344     }
345 
346     /*Print headline to txtFile*/
347     fprintf(txtFile, "time\t");
348     for(i=0; i<nSpins; i++){
349       fprintf(txtFile, "s_%i_x\t", i);
350       fprintf(txtFile, "s_%i_y\t", i);
351       fprintf(txtFile, "s_%i_z\t", i);
352     }
353     fprintf(txtFile, "\n");
354 
355 
356     /*Print starting spins at t=0 to txtFile and binFile*/
357     fprintf(txtFile, "%f\t", 0.0);
358     for(i=0; i<nSpins; i++){
359       fwrite(leaves[0]->data, sizeof(QUAT), 1, binFile);
360       fprintf(txtFile, "%f\t", leaves[0]->data->im);
361       fprintf(txtFile, "%f\t", leaves[0]->data->jm);
362       fprintf(txtFile, "%f\t", leaves[0]->data->km);
```

```
363       leaves[0]++;
364     }
365     fprintf(txtFile, "\n");
366     leaves[0] -= nSpins;
367
368     /*exact data for projectors*/
369     exactEnergy = totalEnergy();
370     totalSpin(&exactTotalSpin);
371
372     printf("Starting Runge-Kutta calculation..\n");
373     start = clock();
374
375     /*Calculation begins here*/
376     for(stepCount=0; stepCount<nSteps; stepCount++) {
377
378       /*One Runge-Kutta step*/
379       calcRK4Step(stepSize);
380
381 /*insert desired projectors here*/
382 /*for(i=0; i<10; i++) {
383 projHamilton( (-totalEnergy() + exactEnergy) / weight() );
384 normalizeSpins();
385 projTotalSpin();
386 }
387
388       /*Print calculated spins to txtFile and binFile*/
389       fprintf(txtFile, "%f\t", (stepCount + 1) * stepSize);
390       for(i=0; i<nSpins; i++){
391         fwrite(leaves[0]->data, sizeof(QUAT), 1, binFile);
392         fprintf(txtFile, "%f\t", leaves[0]->data->im);
393         fprintf(txtFile, "%f\t", leaves[0]->data->jm);
394         fprintf(txtFile, "%f\t", leaves[0]->data->km);
395         leaves[0]++;
396       }
397       fprintf(txtFile, "\n");
398       leaves[0] -= nSpins;
399
400     }
401
402     end = clock();
403     printf("Calculation finished.\n");
404
405     fclose(txtFile);
406     fclose(binFile);
407
408 }
```

Literatur

[1] R. Steinigeweg. *Zur Dynamik von klassischen Heisenberg-Systemen*. Diplomarbeit, Universität Osnabrück, Apr 2005.

[2] D. Mentrup. *Untersuchungen zu kleinen Heisenberg-Spin-Systemen*. Diplomarbeit, Universität Osnabrück, Okt 1999.

[3] C. Schöder. *Numerische Simulation zur Thermodynamik magnetischer Strukturen mittels deterministischer und stochastischer Wärmebadankopplung*. Dissertation, Universität Osnabrück, Nov 1998.

[4] D. Mentrup, H.-J. Schmidt, J. Schnack, M. Luban. *Transition from quantum to classical Heisenberg trimers: thermodynamics and time correlation functions*. Physica A: Statistical Mechanics and its Applications, 278(1-2):214–221, Apr 2000.

[5] R. Steinigeweg, H.-J. Schmidt. *Symplectic integrators for classical spin systems*. Computer Physics Communications, 174:853–861, 2006.

[6] A. Bulgac, D. Kusnezov. *Classical limit for Lie algebras*. Annals of Physics, 199:187–224, April 1990.

[7] R. Abraham, J.E. Marsden, R. Ratiu. *Manifolds, Tensor Analysis, and Applications*. Addison-Wesley, London, 1988.

[8] S.-H. Tsai, M. Krech, D.P. Landau. *Symplectic integration methods in molecular and spin dynamics simulations*. Brazilian Journal of Physics, 34:384 – 391, 06 2004.

[9] H.F. Trotter. *On the product of semi-groups of operators*. Proc. Am. Math. Soc., 10:545, 1959.

[10] G. Strang. *On the construction and comparison of difference schemes*. SIAM J. Numer. Anal., 5:506, 1968.

[11] M. Suzuki. *Fractal decomposition of exponential operators with application to many-body theories and Monte Carlo simulations*. Phys. Lett. A, 146:319, 1990.

[12] E. Forest, R.D. Ruth. *Fourth-order symplectic integration*. Phys. D, 43:105, 1990.

[13] I.P. Omelyan, I.M. Mryglod, R. Folk. *Optimized Forest-Ruth- and Suzuki-like algorithms for integration of motion in many-body systems*. Comput. Phys. Commun., 146:188, 2002.

[14] T.L. Chow. *Mathematical Methods for Physicists: A concise introduction*. Cambridge Univerity Press, Cambridge, 2000.

[15] G. Kempfer. *Online-Skript zur Vorlesung und Übung - Programmieren in C*. `http://public.tfh-berlin.de/~kempfer/skript_c/c.html`, Feb 2008.

[16] K. Jahns. *Monte-Carlo-Simulationen an Fe_{30}-Schichtsystemen*. Bachelorarbeit, Universität Osnabrück, Apr 2006.

[17] A. Müller, M. Luban, C. Schröder, R. Modler, P. Kögerler, M. Axenovich, J. Schnack, P. Canfield, S. Bud'ko, N. Harrison. *Classical and Quantum Magnetism in Giant Keplerate Magnetic Molecules*. Chem. Phys. Chem., 2:517–521, 2001.

[18] M. Axenovich, M. Luban. *Exact ground state properties of the classical Heisenberg model for giant magnetic molecules*. Phys. Rev. B, 63(10):100407, Feb 2001.

[19] A. Stutz, P. Klingebiel. *C Standard-Bibliothek*. `http://www2.hs-fulda.de/~klingebiel/c-stdlib/`, Nov 1999.

[20] E.B. Dam, M. Koch, M. Lillholm. *Quaternions, Interpolation and Animation*. Technical Report DIKU-TR-98/5, University of Copenhagen, Jul 1998.

[21] K. Shoemake. *Quaternions*. `ftp://ftp.cis.upenn.edu/pub/graphics/shoemake/quatut.ps.Z`, 1994.

Danksagung

Diese Diplomarbeit entstand in der Arbeitsgruppe „Makroskopische Systeme und Quantentheorie“ des Fachbereichs Physik an der Universität Osnabrück. An dieser Stelle möchte ich allen danken, die mir bei der Erstellung der vorliegenden Arbeit geholfen haben.

Ganz besonders danke ich Prof. Dr. Heinz-Jürgen Schmidt für die Betreuung meiner Diplomarbeit und für die interessante Themenstellung. Durch viele wertvolle Anregungen und Hinweise auf relevante Literatur hat Prof. Dr. Schmidt wesentlich zum Gelingen dieser Arbeit beigetragen.

Desweiteren danke ich meinen Freunden Katrin Jahns, Timo Kuschel und Carsten Rezny, die meine Arbeit Korrektur gelesen haben und deren Fragen, Hinweise und Kommentare mir sehr geholfen haben. Mein Dank gilt auch Ralf Goschnick für die Diskussion mathematischer Fragestellungen und Peter Hage für weitere wertvolle Hinweise.

Zum Abschluss möchte ich mich bei meiner Familie und meinen Freunden für die moralische Unterstützung bedanken, insbesondere bei meiner Schwester Gudrun und bei meinen Eltern.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel verwendet und zuvor noch keine Diplomprüfung der Fachrichtung Physik abgelegt habe.

Osnabrück, den 6. August 2008

____

Andreas Bröermann