Karlsruher Institut für Technologie

Bachelorarbeit

Numerische Berechnung von stationären Zuständen der Bloch-Redfield Mastergleichung

Numerical Calculation of the Stationary Solution of the Bloch-Redfield Master Equation

Marvin Raimund Schulz

Datum der Abgabe
15.12.2015

Betreuung: Dr. Michael Marthaler
Betreuung: Prof. Dr. Gerd Schön

Fakultät für Physik, Institut für Theoretische Festkörperphysik,

Karlsruher Institut für Technologie

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 15.12.2015

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Arbeit beschäftigt sich mit stationären Zuständen der Bloch-Redfield Mastergleichung. Die Gleichung wird dabei als ein lineares homogenes Differentialgleichungssystem erster Ordnung aufgefasst und hat damit die Form

$$\dot{\rho} = L\rho,\ L \in \mathbb{C}^{n\times n},\ \rho \in \mathbb{C}^n. \tag{1.1}$$

Hierbei ist der stationären Zustand oft von besonderem Interesse. Als Vorlage dient eine bestehende Implementation zur Konstruktion von L und zur Lösung von 1.1 in QuTiP. In QuTiP ist es allerdings nicht möglich den stationären Zustand effizient zu berechnen. Um den stationären Zustand mit QuTiP zu berechnen ist es notwendig den Tensor L aufzustellen und dann alle Eigenvektoren von L zu bestimmen. Es zeigt sich, dass der stationäre Zustand einfacher bestimmt werden kann.
In QuTiP wird nicht davon Gebrauch gemacht, dass zur Auswertung der rechten Seite Multiplikationen mit dünnbesetzten Matrizen auftreten. Ein weiteres Problem besteht in der Tatsache, dass die Matrix L für große Systeme groß wird und so Speicherprobleme auftreten können.
Das Ziel ist nun ein Python Programm zu schreiben, welches

- bestehende QuTiP Strukturen unterstützt.
- den stationären Zustand direkt berechnet.
- die Besetzungsstruktur aller Matrizen ausnutzt.
- den Speicherbedarf begrenzt.

Im fertigen Programm wird die Inverse Potenzenmethode mit Shift verwendet um den stationären Zustand zu approximieren. Um die Inverse in jedem Schritt auszuwerten wird das GMRES Verfahren verwendet, welches eine praktikable Speicherbegrenzung ermöglicht.
Im zweiten Kapitel wird die Bloch-Redfield Mastergleichung hergeleitet und ein Ausdruck für den Operator L aus 1.1 formuliert.
Im dritten Kapitel soll die bestehende Implementierung in Qutip erläutert werden und auf die Optimierungsmöglichkeiten eingegangen werden.
Im vierten Kapitel, werden die numerischen Verfahren erläutert, welche im fertigen Programm zum Einsatz kommen. Außerdem wird der vollständige Quellcode erläutert und die Ergebnisse, Laufzeit und der Speicherbedarf mit Qutip verglichen.
Im fünften Kapitel werden anhand von mehreren physikalischen Modellen beispielhafte Einsatzmöglichkeiten des Programms erläutert.

2. Bloch-Redfield Mastergleichung

2.1. Mastergleichungen

Mastergleichungen sind Differentialgleichungen erster Ordnung, welche zu einem gegebenen physikalischen System die Wahrscheinlichkeitsverteilung und deren Zeitentwicklung beschreiben. Allgemein lässt sich diese mit den Übergangswahrscheinlichkeiten zwischen diskreten Zuständen als

$$\frac{dP_k}{dt} = \sum_l \left(T_{kl}P_l - T_{lk}P_k\right) \tag{2.1}$$

schreiben. Hierbei bezeichnet $T \in \mathbb{C}^{n\times n}$ eine Matrix deren Elemente T_{kl} die Übergangswahrscheinlichkeit zwischen dem Zustand k und dem Zustand l enthält. P_i ist die Wahrscheinlichkeit, das sich das System im Zustand i befindet.

Dichteoperator Um quantenmechanische Systeme zu beschreiben wird die Dichtematrix verwendet. Das quantenmechanische System ist durch einen abstrakten Zustandsraum $\mathbb{H}$ und die Linearen Operatoren $L(\mathbb{H})$ auf diesen Raum beschrieben. Dieser Zustandsraum ist ein Hilbertraum. Wenn nun $\{|\Psi_i\rangle\}$ normierte Zustände in diesem Raum sind, dann ist deren Dichteoperator durch

$$\rho = \sum_i p_i \left|\Psi_i\right\rangle \left\langle\Psi_i\right| \tag{2.2}$$

gegeben, wobei p_i die Wahrscheinlichkeit ist, dass sich das System im Zustand $|\Psi_i\rangle$ befindet. Es gilt $\sum_i p_i = 1$. Jede Dichtematrix ist hermitesch, hat Spur eins und ist positiv semidefinit.
Die Allgemeinste Form in der eine Mastergleichung für den Dichteoperator geschrieben werden kann ist die Lindblad-Mastergleichung. Die Hermitizität, die Positivität und die Spur des Dichtoperators ist erhalten. Eine Mastergleichung in Lindbladform hat die allgemeine Form

$$\dot{\rho} = -i[H,\rho] + \sum_{\alpha,\beta=1}^{N^2-1} \gamma_{\alpha\beta}\left(A_\alpha \rho A_\beta^\dagger - \frac{1}{2}\left\{A_\beta^\dagger A_\alpha, \rho\right\}\right). \tag{2.3}$$

Hierbei bezeichnet H den Hamiltonoperator, ρ den Dichteoperator, $\gamma = (\gamma_{\alpha\beta})$ eine positive semidefinite Matrix und A einen Operator.

2.2. Bloch-Redfield Formalismus

Im Bloch-Redfield Formalismus betrachtet man ein offenes Quantensystem als Teil eines größeren geschlossenen Quantensystems. Die beiden Quantensysteme sollen schwach wechselwirken. Das größere Quantensystem wird als Bad bezeichnet und das Kleinere als

System. Der Zustandsraum des kleineren Systems wird im Folgenden mit $\mathbb{S}$ und der des größeren Systems als $\mathbb{B}$ bezeichnet. Ausgangspunkt für die Beschreibung der Systeme ist der Hamilton Operator des gesamten Systems. Dieser kann mithilfe des Kroneckerprodukts für Matrizen $\otimes$ als

$$H = H_S \otimes \mathbb{1} + \mathbb{1} \otimes H_B + H_I \tag{2.4}$$

geschrieben werden. Hierbei bezeichnen $H_S \in L(\mathbb{S})$ den Hamiltonoperator der Umgebung und $H_B \in L(\mathbb{B})$ den Hamiltonoperator des Bades. Außerdem ist ein weiterer Term zur Beschreibung der Wechselwirkung notwendig. $H_I \in L(\mathbb{S} \otimes \mathbb{B})$ bezeichnet den Wechselwirkungshamiltonoperator der auf beide Systeme wirkt. Der Wechselswirkungshamiltonoperator kann durch Operatoren $A_\alpha \in L(\mathbb{S})$ und $B_\alpha \in L(\mathbb{B})$ als

$$H_I = \sum_{\alpha=1}^{N} A_\alpha \otimes B_\alpha \tag{2.5}$$

geschrieben werden. Um das System zu beschreiben wird die Spur über die zusätzlichen Freiheitsgrade des Bades gezogen.

Definition 2.1 (Spur über Subsystem)**.** *Sei $A = C \otimes D$ mit $C \in L(R)$ und $D \in L(V)$, wobei R, V komplexe Hilberträume sind. Dann ist die Spur über V von A durch*

$$\begin{aligned} &tr_V : L(R \otimes V) \to L(R), \\ &tr_V\,(C \otimes D) \mapsto C\,trD \end{aligned}$$

definiert.

Reduzierter Dichteoperator Ausgehend vom Dichteoperator des gesamten Systems kann mit dem Spurbegriff aus 2.1 der reduzierte Dichteoperator definiert werden. Sei hierzu χ ein Dichteoperator aus $L(\mathbb{S} \otimes \mathbb{B})$. Mit

$$\rho = \mathrm{tr}_B \chi \in L(\mathbb{S}) \tag{2.6}$$

bezeichnet man nun den reduzieren Dichteoperator des Systems.
Ziel wird es nun sein eine Mastergleichung für diesen reduzierten Dichteoperator aufzustellen.

2.3. Herleitung der Bloch-Redfield Mastergleichung

Im Folgenden sei $\hbar = 1$. Ausgehend vom Bloch-Redfield Formalismus lässt sich eine Mastergleichung für den reduzierten Dichteoperator aufstellen. Die Zeitentwicklung ist durch die Von-Neumann-Gleichung gegeben. Für das gesamte System lautet diese

$$\frac{\partial \chi}{\partial t} = -i\,[H, \chi]\,. \tag{2.7}$$

Es bezeichnet $\chi \in L(\mathbb{S} \otimes \mathbb{B})$ den Dichteoperator des gesamten Systems (System und Bad). Der dazugehörende Hamiltonoperator H ist entsprechend 2.4 gewählt. Im nächsten Schritt wird 2.7 in das Wechselwirkungsbild transformiert. Da H nicht von der Zeit abhängen soll ist die Transformation des Operators A bezüglich H in das Wechselwirkungsbild durch

$$A_D(t) = e^{iHt} A e^{-iHt} \tag{2.8}$$

gegeben. Durch Transformation bezüglich $H_S \otimes \mathbb{1} + \mathbb{1} \otimes H_B$ folgt

$$\frac{\partial \chi_D}{\partial t} = -i \left[(H_I)_D, \chi_D \right], \tag{2.9}$$

wobei $(H_I)_D$ der Wechselwirkungshamiltonoperator gemäß 2.4 ist. Durch formale Integration von 2.9 findet sich ein Ausdruck für die Dichtematrix zu einem bestimmten Zeitpunkt. Es ist

$$\chi_D(t) = \chi(0) - i \int_0^t \left[(H_I)_D(t'), \chi_D(t') \right] dt'. \tag{2.10}$$

Das Integral ist elementweise zu verstehen. Diese Gleichung kann nun wieder in 2.9 eingesetzt werden. Daraus ergibt sich die Integro-Differentialgleichung

$$\frac{\partial \chi_D(t)}{\partial t} = -i \left[(H_I)_D(t), \chi_D(0) \right] - \int_0^t \left[(H_I)_D(t), \left[(H_I)_D(t'), \chi_D(t') \right] \right] dt'. \tag{2.11}$$

Zusätzlich muss angenommen werden, dass die Wechselwirkung bei $t = 0$ eingeschaltet wird und dass zu diesem Zeitpunkt noch keine Korrelation zwischen den beiden Systemen besteht. Das heißt, zum Zeitpunkt $t = 0$ lässt sich χ in Dichteoperatoren des Systems und des Bades faktorisieren. Es ist daher

$$\chi(0) = \rho_s(0) \otimes R(0), \;\; \rho_s(0) \in L(\mathbb{U}), R(0) \in L(\mathbb{B}). \tag{2.12}$$

Ein Ausdruck für den reduzierten Dichteoperator 2.6 kann durch das Ziehen der Spur über das Bad gewonnen werden. Aus 2.11 wird damit

$$\frac{\partial \rho_D(t)}{\partial t} = -i \mathrm{tr}_B \left[(H_I)_D(t), (\rho_s(0) \otimes R_0)_D \right] - \int_0^t \mathrm{tr}_B \left[(H_I)_D(t), \left[(H_I)_D(t'), \chi_D(t') \right] \right] dt'. \tag{2.13}$$

Diese Gleichung ist nun eine Mastergleichung. Ohne Beschränkung der Allgemeinheit kann

$$-i \mathrm{tr}_B \left[(H_I)_D(t), (\rho_s(0) \otimes R(0))_D \right] = 0 \tag{2.14}$$

gewählt werden [Car, S.6]. Der Fall 2.14 kann erreicht werden, indem der Systemhamiltonoperator um eine geeigente Konstante ergänzt wird, was die Systemdynamik nicht

beeinflusst. Mit 2.14 kann die Mastergleichung einfacher als

$$\frac{\partial \rho_D(t)}{\partial t} = -\int_0^t \mathrm{tr}_B \left[(H_I)_D(t), [(H_I)_D(t'), \chi_D(t')]\right] dt' \tag{2.15}$$

geschrieben werden.

2.4. Born-Markov Näherungen

Bornsche Näherung Nachdem zum Zeitpunkt $t = 0$ keine Kopplung zwischen dem System und dem Bad besteht wird diese zu späteren Zeitpunkten auftreten. Nun soll angenommen werden, dass die Kopplung (welche durch den Wechselwirkungshamilton-operator gegeben ist) sehr schwach ist. Dies führt zu der Bornschen Näherug, dass χ nur wenig von einem unkorrelierten Zustand abweicht. Es gilt die Näherung für $t \geq 0$

$$\chi(t)_D = \rho_D(t) \otimes R(t) + O(H_I).$$

Mit dieser Näherung hängt die Mastergleichung

$$\frac{\partial \rho_D(t)}{\partial t} = -\int_0^t \mathrm{tr}_B \left[(H_I)_D(t), [(H_I)_D(t'), \rho_D(t') \otimes R(t')]\right] dt' \tag{2.16}$$

nicht mehr von dem Dichteoperator χ des gesamten Systems ab, sondern nur noch von dem reduzierten Dichteoperator. Nun ist es zweckmäßig die Kommutatoren auszuführen und die Spur anzuwenden. Es ergeben sich vier Summanden

$$\begin{aligned}
-\frac{\partial \rho_D(t)}{\partial t} &= \int_0^t \mathrm{tr}_B \left[(H_I)_D(t), [(H_I)_D(t'), \rho_D(t') \otimes R(t')]\right] dt' \\
&= \int_0^t \mathrm{tr}_B \Big[(H_I)_D(t)(H_I)_D(t')\{\rho_D(t') \otimes R(t')\} \\
&\qquad + \{\rho_D(t') \otimes R(t')\}(H_I)_D(t)(H_I)_D(t') \\
&\qquad - (H_I)_D(t)\{\rho_D(t') \otimes R(t')\}(H_I)_D(t') \\
&\qquad - (H_I)_D(t')\{\rho_D(t') \otimes R(t')\}(H_I)_D(t) \Big] dt'.
\end{aligned} \tag{2.17}$$

Für H_I soll nun die Definition aus 2.5 eingesetzt werden. Im Wechselwirkungsbild ist H_I durch

$$(H_I)_D(t) = \sum_{\alpha=1}^{N} A_D^\alpha(t) \otimes B_D^\alpha(t) \tag{2.18}$$

gegeben. Hierbei wurden die ursprünglichen Indizes der Übersichtlichkeit halber nach oben verschoben. Eingesetzt in 2.17 ergibt sich

$$
\begin{aligned}
-\frac{\partial \rho_D(t)}{\partial t} = \int_0^t \sum_{\alpha,\beta=1}^{N} \mathrm{tr}_B \Big[& \left(A_D^\alpha(t) \otimes B_D^\alpha(t)\right) \left(A_D^\beta(t') \otimes B_D^\beta(t')\right) \{\rho_D(t') \otimes R(t')\} \\
& + \{\rho_D(t') \otimes R(t')\} \left(A_D^\beta(t') \otimes B_D^\beta(t')\right) \left(A_D^\alpha(t) \otimes B_D^\alpha(t)\right) \\
& - \left(A_D^\alpha(t) \otimes B_D^\alpha(t)\right) \{\rho_D(t') \otimes R(t')\} \left(A_D^\beta(t') \otimes B_D^\beta(t')\right) \\
& - \left(A_D^\beta(t') \otimes B_D^\beta(t')\right) \{\rho_D(t') \otimes R(t')\} \left(A_D^\alpha(t) \otimes B_D^\alpha(t)\right) \Big] dt'.
\end{aligned} \tag{2.19}
$$

Mit der Eigenschaft des Kroneckerprodukts, dass $(A \otimes B)(C \otimes D) = AC \otimes BD$ für $A, C \in L(\mathbb{S})$ und $B, D \in L(\mathbb{B})$ ist

$$
\begin{aligned}
-\frac{\partial \rho_D(t)}{\partial t} = \int_0^t \sum_{\alpha,\beta=1}^{N} \mathrm{tr}_B \Big[& A_D^\alpha(t) A_D^\beta(t') \rho_D(t') \otimes B_D^\alpha(t) B_D^\beta(t') R(t') \\
& + \rho_D(t') A_D^\beta(t') A_D^\alpha(t) \otimes R(t') B_D^\beta(t') B_D^\alpha(t) \\
& - A_D^\alpha(t) \rho_D(t') A_D^\beta(t') \otimes B_D^\alpha(t) R(t') B_D^\beta(t') \\
& - A_D^\beta(t') \rho_D(t') A_D^\alpha(t) \otimes B_D^\beta(t') R(t') B_D^\alpha(t) \Big] dt'.
\end{aligned} \tag{2.20}
$$

Nun kann die Spur über $\mathbb{B}$ gezogen werden. Nutzt man noch die Invarianz der Spur unter zyklischem vertauschen der Argumente aus folgt

$$
\begin{aligned}
-\frac{\partial \rho_D(t)}{\partial t} = \int_0^t \sum_{\alpha,\beta=1}^{N} & \left(A_D^\alpha(t) A_D^\beta(t') \rho_D(t') - A_D^\beta(t') \rho_D(t') A_D^\alpha(t)\right) \left\langle B_D^\alpha(t) B_D^\beta(t') \right\rangle_B \\
& + \left(\rho_D(t') A_D^\beta(t') A_D^\alpha(t) - A_D^\alpha(t) \rho_D(t') A_D^\beta(t')\right) \left\langle B_D^\beta(t') B_D^\alpha(t) \right\rangle_B dt'.
\end{aligned} \tag{2.21}
$$

Hierbei sind

$$
\mathrm{tr}\left[R(t') B_D^\alpha(t) B_D^\beta(t')\right] = \left\langle B_D^\alpha(t) B_D^\beta(t') \right\rangle_B \tag{2.22}
$$

$$
\mathrm{tr}\left[R(t') B_D^\beta(t') B_D^\alpha(t')\right] = \left\langle B_D^\beta(t') B_D^\alpha(t') \right\rangle_B \tag{2.23}
$$

die Korellationsfunktionen zwischen den Badoperatoren.

Markovsche Näherungen Durch das Integral auf der rechten Seite von 2.16 hängt die Gleichung von den vorangegangen Zuständen ab. Die Annahme die zur ersten Markovschen Näherung führt ist, dass die Korrelationfunktionen 2.22 schneller abfallen, als sich der reduzierte Dichteoperator im Wechselwirkungsbild ändern kann [Vog11][S.10].

Idealerweise gilt

$$\left\langle B_D^\beta(t')B_D^\alpha(t')\right\rangle_B \propto \delta(t-t'). \tag{2.24}$$

Daher kann in 2.21 $\rho_D(t')$ durch $\rho_D(t)$ ersetzt werden. Damit hängt die Gleichung nicht mehr von den vorangegangen Zuständen ab.
Geht man etwas näher auf die Korrelationsfunktionen ein findet sich, dass diese nur von $t-t'$ abhängen, denn es ist

$$\begin{aligned}&\mathrm{tr}\left[R(t')B_D^\alpha(t)B_D^\beta(t')\right] \\ &= \mathrm{tr}[e^{iH_Bt'}Re^{-iH_Bt'}e^{iH_Bt}B_D^\alpha e^{-iH_Bt}e^{iH_Bt'}B_D^\beta e^{-iH_Bt'}] \\ &= \mathrm{tr}[Re^{iH_B(t-t')}B_D^\alpha e^{-iH_B(t-t')}B_D^\beta] \\ &= \mathrm{tr}[R(0)B_D^\alpha(t-t')B_D^\beta(0)] = C_{\alpha\beta}(t-t').\end{aligned} \tag{2.25}$$

Hierbei wurde ausgenutzt, dass R mit H_B vertauscht und die zyklische Invarianz der Spur. Für die andere Korrelationsfunktion kann analog vorgegangen werden. Es findet sich außerdem die Symmetrie

$$C_{\alpha\beta}(t-t') = C^\star_{\beta\alpha}(t'-t). \tag{2.26}$$

Mit der Korrelationsfunktion ist 2.21 nun

$$\begin{aligned}-\frac{\partial\rho_D(t)}{\partial t} = \int_0^t \sum_{\alpha,\beta=1}^N &\left(A_D^\alpha(t)A_D^\beta(t')\rho_D(t) - A_D^\beta(t')\rho_D(t)A_D^\alpha(t)\right)C_{\alpha\beta}(t-t') \\ &+ \left(\rho_D(t)A_D^\beta(t')A_D^\alpha(t) - A_D^\alpha(t)\rho_D(t)A_D^\beta(t')\right)C_{\beta\alpha}(t'-t)dt'.\end{aligned} \tag{2.27}$$

Diese Gleichung hat auch im Schrödingerbild zeitabhängige Koeffizienten, da die rechte Seite durch das Integral von der Zeit abhängt. Die zweite Markovsche Näherung führt zu konstanten Koeffizienten. Zuerst wird nun $t-t'=\tau$ substituiert. Hierzu ist es dienlich die Operatoren im Schrödingerbild zu betrachten. Es ist

$$\frac{d\tau}{dt'} = -1,\ \tau(0)=t,\ \tau(t)=0. \tag{2.28}$$

Nach der Substitutionsregel kann 2.27 in

$$\begin{aligned}-\frac{\partial\rho_D(t)}{\partial t} = \int_0^t e^{iH_St}\sum_{\alpha,\beta=1}^N &\left(A^\alpha e^{iH_S\tau}A^\beta e^{-iH_S\tau}\rho(t) - e^{iH_S\tau}A^\beta e^{-iH_S\tau}\rho(t)A^\alpha\right)e^{-iH_St}C_{\alpha\beta}(\tau) \\ &+ e^{iH_St}\left(\rho(t)e^{iH_S\tau}A^\beta e^{-iH_S\tau}A^\alpha - A^\alpha\rho(t)e^{iH_S\tau}A^\beta e^{-iH_S\tau}\right)e^{-iH_St}C_{\beta\alpha}(-\tau)d\tau\end{aligned} \tag{2.29}$$

umgeformt werden, wobei die Grenzen des Integrals vertauscht wurden. Die linke Seite der Gleichung lässt sich ebenfalls im Schrödingerbild als

$$\begin{aligned}\frac{\partial \rho_D(t)}{\partial t} &= \frac{\partial}{\partial t}\left(e^{iH_S t}\rho e^{-iH_S t}\right) \\ &= iHe^{iH_S t}\rho(t)e^{-iH_S t} + e^{iH_S t}\dot{\rho}(t)e^{-iH_S t} + e^{iH_S t}\rho(t)(-iH_S)e^{iH_S t} \\ &= e^{iH_S t}\left(i\left[H_S, \rho(t)\right] + \dot{\rho}(t)\right)e^{-iH_S t} \end{aligned} \tag{2.30}$$

schreiben. Mit 2.29 und 2.30 ist die Mastergleichung für die reduzierte Dichtematrix nun

$$\begin{aligned}\dot{\rho}(t) = -i\left[H, \rho(t)\right] - \int_0^t \Big\{ \sum_{\alpha,\beta=1}^{N} \left(A^\alpha e^{iH_S\tau}A^\beta e^{-iH_S\tau}\rho(t) - e^{iH_S\tau}A^\beta e^{-iH_S\tau}\rho(t)A^\alpha\right) C_{\alpha\beta}(\tau) \\ + \left(\rho(t)e^{iH_S\tau}A^\beta e^{-iH_S\tau}A^\alpha - A^\alpha\rho(t)e^{iH_S\tau}A^\beta e^{-iH_S\tau}\right) C_{\beta\alpha}(-\tau)\Big\} d\tau. \end{aligned} \tag{2.31}$$

Da die Korrelationsfunktionen schnell abfallen im Vergleich zu der Zeitskala in der sich $\rho(t)$ ändert wird angenommen, dass dieses Integral sich nicht wesentlich ändert wenn es auf ∞ ausgedehnt wird. Die Bloch-Redfield Mastergleichung lautet

$$\begin{aligned}\dot{\rho}(t) = -i\left[H_S, \rho(t)\right] - \int_0^\infty \Big\{ \sum_{\alpha,\beta=1}^{N} \left(A^\alpha e^{iH_S\tau}A^\beta e^{-iH_S\tau}\rho(t) - e^{iH_S\tau}A^\beta e^{-iH_S\tau}\rho(t)A^\alpha\right) C_{\alpha\beta}(\tau) \\ + \left(\rho(t)e^{iH_S\tau}A^\beta e^{-iH_S\tau}A^\alpha - A^\alpha\rho(t)e^{iH_S\tau}A^\beta e^{-iH_S\tau}\right) C_{\beta\alpha}(-\tau)\Big\} d\tau. \end{aligned} \tag{2.32}$$

Diese Gleichung hat konstante Koeffizienten auf der rechten Seite und erhält die Spur und die Hermitizität des Dichteoperators. Die Gleichung 2.32 erhält allerdings im Allgemeinen nicht die Positivität des Dichteoperators. Damit diese ebenfalls erhalten ist muss die sogenannte säkulare Näherung durchgeführt werden auf die allerdings verzichtet werden soll.
Die Gleichung 2.32 ist in dieser Form nicht geeignet in einem Computerprogramm implementiert zu werden. Im nächsten Abschnitt werden wir die Gleichung soweit umformen, dass zur Auswertung nur noch Multiplikationen von Matrizen notwendig sein werden. Hierzu werden zwei Operatoren T^+ und T^- eingeführt, welche die für das Integral relevanten Terme zusammnfassen. Es zeigt sich, dass diese durch die Fouriertransformierte von $C_{\alpha\beta}$ leicht berechnen werden könen.

2.5. Bloch-Redfield Tensor

Ausgehend von 2.32 lässt sich ein Tensor L finden sodass

$$\mathrm{vec}(\dot{\rho}) = L\ \mathrm{vec}(\rho). \tag{2.33}$$

Die Funktion vec transformiert eine Matrix in einen Vektor indem sie die Spalten der Matrix untereinander in einen Vektor schreibt.

Es seien

$$T^{+}_{\alpha\beta} := \int_0^\infty e^{iH_S\tau} A^\beta e^{-iH_S\tau} C_{\alpha\beta}(\tau) d\tau, \tag{2.34}$$
$$T^{-}_{\beta\alpha} := \int_0^\infty e^{iH_S\tau} A^\beta e^{-iH_S\tau} C_{\beta\alpha}(-\tau) d\tau$$

die für das Integral relevanten Terme. Damit lässt sich 2.32 als

$$\dot{\rho}(t) = -i\left[H_S, \rho(t)\right] - \sum_{\alpha,\beta=1}^{N} \Big\{ A^\alpha T^{+}_{\alpha\beta}\rho(t) - T^{+}_{\alpha\beta}\rho(t)A^\alpha + \rho(t)T^{-}_{\beta\alpha}A^\alpha - A^\alpha\rho(t)T^{-}_{\beta\alpha} \Big\}. \tag{2.35}$$

schreiben. Mit der Beziehung

$$\mathrm{vec}(AXB^T) = (B \otimes A)\,\mathrm{vec}(X), \tag{2.36}$$

wobei A, X, B komplexe Matrizen sind, kann ein Ausdruck für L gefunden werden. Es findet sich

$$\begin{aligned} L = &-i\left(\mathbb{1} \otimes H_S - H_S^T \otimes \mathbb{1}\right) \\ &- \sum_{\alpha,\beta=1}^{N} \Big(\mathbb{1} \otimes A^\alpha T^{+}_{\alpha\beta} - (A^\alpha)^T \otimes T^{+}_{\alpha\beta} + (T^{-}_{\beta\alpha}A^\alpha)^T \otimes \mathbb{1} + (T^{-}_{\beta\alpha})^T \otimes A^\alpha \Big). \end{aligned} \tag{2.37}$$

Um die Matrizen $T^{-}_{\beta\alpha}$ und $T^{+}_{\alpha\beta}$ zu berechnen muss H_S diagonalisiert werden. Sei

$$VH_SV^T = \mathrm{diag}(\varepsilon_1, \dots \varepsilon_n) \tag{2.38}$$

der Basiswechsel in die Eigenbasis des Hamiltonoperators. Außerdem ist

$$\begin{aligned} Ve^{iH_S\tau}V^T &= \sum_{k=0}^{\infty} V\frac{(iH\tau)^k}{k!}V^T \\ &= \sum_{k=0}^{\infty} \frac{(i\tau)^k VHV^TVH\dots HV^TVHV^T}{k!} \\ &= \sum_{k=0}^{\infty} \frac{(i\mathrm{diag}(\varepsilon_1, \dots \varepsilon_n)\tau)^k}{k!} \\ &= e^{i\mathrm{diag}(\varepsilon_1,\dots,\varepsilon_n)\tau} = \mathrm{diag}(e^{i\varepsilon_1\tau}, \dots, e^{i\varepsilon_n\tau}). \end{aligned} \tag{2.39}$$

Nun ist $T^+_{\alpha\beta}$ in dieser Basis durch

$$VT^+_{\alpha\beta}V^T = \int_0^\infty \mathrm{diag}(e^{-i\varepsilon_1\tau},\dots,e^{-i\varepsilon_n\tau})\underbrace{VA^\beta V^T}_{:=\chi_\beta}\mathrm{diag}(e^{i\varepsilon_1\tau},\dots,e^{i\varepsilon_n\tau})C_{\alpha\beta}(\tau)d\tau \tag{2.40}$$

gegeben. Die Multiplikation einer beliebigen Matrix mit einer diagonalen Matrix lässt sich durch eine elementweise Multiplikation $*$ als

$$\mathrm{diag}(e^{-i\varepsilon_1\tau},\dots,e^{-i\varepsilon_n\tau})\chi_\beta\mathrm{diag}(e^{i\varepsilon_1\tau},\dots,e^{i\varepsilon_n\tau}) = \chi_\beta * G(\tau) \tag{2.41}$$

schreiben, wobei

$$G(\tau) = \begin{pmatrix} 1 & e^{-i(\varepsilon_1-\varepsilon_2)\tau} & \dots \\ e^{-i(\varepsilon_2-\varepsilon_1)\tau} & 1 & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

ist. Es ist also

$$VT^+_{\alpha\beta}V^T = \int_0^\infty \chi_\beta * G(\tau)C_{\alpha\beta}(\tau)d\tau. \tag{2.42}$$

Für jedes Element in $T^+_{\alpha\beta}$ ist ein Integral

$$\left(VT^+_{\alpha\beta}V^T\right)_{ij} \propto \int_0^\infty e^{-i(\varepsilon_i-\varepsilon_j)\tau}C_{\alpha\beta}(\tau)d\tau \tag{2.43}$$

zu lösen. Es sei $\varepsilon_i - \varepsilon_j =: \lambda_{ij}$. Mit der Fouriertransformierten der Korrelationsfunktion

$$s_{\alpha\beta}(\omega) = \int_{-\infty}^\infty e^{i\omega\tau}C_{\alpha\beta}(\tau)d\tau \tag{2.44}$$

kann 2.43 durch die Rücktransformation als

$$\begin{aligned} &\int_0^\infty e^{-i(\varepsilon_i-\varepsilon_j)\tau}C_{\alpha\beta}(\tau)d\tau \\ &= \int_0^\infty e^{-i(\lambda_{ij})\tau}\int_{-\infty}^\infty \frac{1}{2\pi}e^{i\omega\tau}s_{\alpha\beta}(\omega)d\omega d\tau \\ &= \frac{1}{2\pi}\int_{-\infty}^\infty\int_0^\infty e^{-i(\lambda_{ij}-\omega)\tau}s_{\alpha\beta}(\omega)d\tau d\omega \end{aligned} \tag{2.45}$$

geschrieben werden. Um dieses Integral zu berechnen ist eine Erkenntnis aus der Distributionentheorie notwendig. [Car, S.15] Es gilt

$$\int_0^\infty dx\ e^{ikx} = \pi\delta(k) + i\mathcal{P}\frac{1}{k}. \tag{2.46}$$

Wobei $\delta(k)$ die Deltadistribution und $\mathcal{P}$ der Cauchy Hauptwert ist. Mit 2.46 wird 2.45 zu

$$\begin{aligned}&\frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{0}^{\infty} e^{-i(\lambda_{ij}-\omega)\tau}s_{\alpha\beta}(\omega)d\tau d\omega \\ &=\int_{-\infty}^{\infty}\frac{1}{2}s_{\alpha\beta}(\omega)\delta(\omega-\lambda_{ij})+i\frac{\mathcal{P}}{2}s_{\alpha\beta}(\omega)d\omega.\end{aligned} \tag{2.47}$$

Der Imaginärteil führt zu einer Energieverschiebung (Lamb-Shift) und wird hier vernachlässigt. Mit der Definition der Deltadistribution ist dann

$$\int_{0}^{\infty} e^{-i(\lambda_{ij})\tau}C_{\alpha\beta}(\tau)d\tau=\frac{1}{2}s_{\alpha\beta}(\lambda_{ij}). \tag{2.48}$$

Die Elemente von $VT^{+}_{\alpha\beta}V^{T}$ sind nun durch

$$\left(VT^{+}_{\alpha\beta}V^{T}\right)_{ij}=\frac{1}{2}(\chi_{\beta})_{ij}s_{\alpha\beta}(\lambda_{ij}) \tag{2.49}$$

gegeben. Sei nun

$$G_{ij}:=\frac{1}{2}s_{\alpha\beta}(\lambda_{ij}). \tag{2.50}$$

Mit 2.50 ist $T^{+}_{\alpha\beta}$ in der ursprünglichen Basis

$$T^{+}_{\alpha\beta}=V^{T}\big((VA^{\beta}V^{T})*G\big)V. \tag{2.51}$$

Für $T^{-}_{\beta\alpha}$ kann analog vorgegangen werden.
Die Gleichung 2.35 kann nun zur Auswertung der Mastergleichung benutzt werden. Zur Berechnung von $\dot{\rho}$ sind nun nur Multiplikationen von Matrizen notnwendig.

2.6. Stationärer Zustände und säkulare Näherung

Im Folgenden wird auf die säkulare Näherung eingegangen, welche im späteren Programm nicht implementiert wurde. Allerdings kann mit dieser Näherung begründet werden, weshalb die Existenz eines stationären Zustands zu erwarten ist. Ohne die säkulare Näherung wird die Erhaltung der Positivität, der Wahrscheinlichkeit und die Existenz der stationären Lösung nicht gewährleistet.

2.6.1. Entkopplung der Diagonalelemente

Imaginärer Anteil In seiner Eigenbasis ist der Hamiltonoperator diagonal, sodass in dieser Basis $\{|i\rangle\}$ der imaginäre Anteil des Bloch Refield Tensor zu

$$\begin{aligned} \dot{\rho} &= -i[H_S, \rho] \\ \Rightarrow \langle i|\,\dot{\rho}\,|j\rangle &= -i\,\langle i|\,[H_S, \rho]\,|j\rangle \\ &= -i(\varepsilon_i - \varepsilon_j)\,\langle i|\,\rho\,|j\rangle \end{aligned} \tag{2.52}$$

wird. Die Diagonalelemente sind von den nicht Diagonalen entkoppelt. Für die diagonalen $i = j$ gilt

$$\langle i|\,\dot{\rho}\,|i\rangle = 0. \tag{2.53}$$

Ausgehend von 2.35 kann mit den Eigenzuständen des Systemhamiltonoperator $\{|i\rangle\}$ die Matrixmultiplikationen elementweise betrachtet werden. Um die Übersichtlichkeit zu wahren wird im Folgenden der erste Kommutator vernachlässigt. Es ist durch Ergänzen mehrere Identitäten $\sum_{i=1}^{n} |i\rangle\langle i| = \mathbb{1}$,

$$\begin{aligned} \dot{\rho} = & \\ & \sum_{\alpha,\beta=1}^{N} \sum_{a,b,c,d} \Big\{ |a\rangle\langle a|\,A^\alpha\,|b\rangle\langle b|\,T^+_{\alpha\beta}\,|c\rangle\langle c|\,\rho(t)\,|d\rangle\langle d| + \dots \\ & = \int_0^\infty \sum_{\alpha,\beta=1}^{N} \sum_{a,b,c,d} \Big\{ |a\rangle\langle a|\,A^\alpha\,|b\rangle\langle b|\,A^\beta\,|c\rangle\langle c|\,\rho(t)\,|d\rangle\langle d| \\ & + |b\rangle\langle b|\,A^\beta\,|c\rangle\langle c|\,\rho\,|d\rangle\langle d|\,A^\alpha\,|a\rangle\langle a|\,\Big\} e^{i(\varepsilon_b-\varepsilon_c)\tau} C_{\alpha\beta}(\tau) d\tau \\ & + \int_0^\infty \sum_{\alpha,\beta=1}^{N} \sum_{a,b,c,d} \Big\{ |c\rangle\langle c|\,\rho\,|d\rangle\langle d|\,A^\beta\,|a\rangle\langle a|\,A^\alpha\,|b\rangle\langle b| \\ & + |b\rangle\langle b|\,A^\alpha\,|c\rangle\langle c|\,\rho\,|d\rangle\langle d|\,A^\beta\,|a\rangle\langle a|\,\Big\} e^{i(\varepsilon_d-\varepsilon_a)\tau} C_{\beta\alpha}(-\tau) d\tau \\ & = \int_0^\infty \sum_{\alpha,\beta=1}^{N} \sum_{a,b,c,d} \Big\{ |a\rangle\langle a|\,A^\alpha\,|b\rangle\langle b|\,A^\beta\,|c\rangle\langle d| \\ & + |b\rangle\langle b|\,A^\beta\,|c\rangle\langle d|\,A^\alpha\,|a\rangle\langle a|\,\Big\} e^{i(\varepsilon_b-\varepsilon_c)\tau} C_{\alpha\beta}(\tau) d\tau\,\langle c|\,\rho\,|d\rangle \\ & + \int_0^\infty \sum_{\alpha,\beta=1}^{N} \sum_{a,b,c,d} \Big\{ |c\rangle\langle d|\,A^\beta\,|a\rangle\langle a|\,A^\alpha\,|b\rangle\langle b| \\ & + |b\rangle\langle b|\,A^\alpha\,|c\rangle\langle d|\,A^\beta\,|a\rangle\langle a|\,\Big\} e^{i(\varepsilon_d-\varepsilon_a)\tau} C_{\beta\alpha}(-\tau) d\tau\,\langle c|\,\rho\,|d\rangle\,. \end{aligned} \tag{2.54}$$

Ein Element aus $\dot{\rho}$ ist damit

$$\begin{aligned}\langle n|\,\dot{\rho}\,|m\rangle = & \qquad (2.55)\\ &\int_0^\infty \sum_{c,d}\sum_{\alpha,\beta=1}^{N}\sum_{a,b}\Big(\Big\{\delta_{an}\delta_{dm}\,\langle a|\,A^\alpha\,|b\rangle\,\langle b|\,A^\beta\,|c\rangle\\ &+\delta_{bn}\delta_{ma}\,\langle b|\,A^\beta\,|c\rangle\,\langle d|\,A^\alpha\,|a\rangle\,\Big\}e^{i(\varepsilon_b-\varepsilon_c)\tau}C_{\alpha\beta}(\tau)\\ &+\Big\{\delta_{cn}\delta_{bm}\,\langle d|\,A^\beta\,|a\rangle\,\langle a|\,A^\alpha\,|b\rangle\\ &+\delta_{bn}\delta_{am}\,\langle b|\,A^\alpha\,|c\rangle\,\langle d|\,A^\beta\,|a\rangle\,\Big\}e^{i(\varepsilon_d-\varepsilon_a)\tau}C_{\beta\alpha}(-\tau)\Big)\,\langle c|\,\rho\,|d\rangle\,d\tau\\ &=\int_0^\infty \sum_{c,d}\sum_{\alpha,\beta=1}^{N}\Big(\Big\{\sum_k\delta_{dm}\,\langle n|\,A^\alpha\,|k\rangle\,\langle k|\,A^\beta\,|c\rangle\,e^{i(\varepsilon_k-\varepsilon_c)\tau}\\ &+\langle n|\,A^\beta\,|c\rangle\,\langle d|\,A^\alpha\,|m\rangle\,e^{i(\varepsilon_n-\varepsilon_c)\tau}\Big\}C_{\alpha\beta}(\tau)\\ &+\Big\{\sum_k\delta_{cn}\,\langle d|\,A^\beta\,|k\rangle\,\langle k|\,A^\alpha\,|m\rangle\,e^{i(\varepsilon_d-\varepsilon_k)\tau}\\ &+\langle n|\,A^\alpha\,|c\rangle\,\langle d|\,A^\beta\,|m\rangle\,e^{i(\varepsilon_d-\varepsilon_m)\tau}\Big\}C_{\beta\alpha}(-\tau)\Big)\,\langle c|\,\rho\,|d\rangle\,d\tau.\end{aligned}$$

Außerdem werden die Korrelationen zwischen A^α und A^β vernachlässigt, sodass

$$\begin{aligned}\langle n|\,\dot{\rho}\,|m\rangle = & \qquad (2.56)\\ &\int_0^\infty \sum_{c,d}\sum_{\alpha=1}^{N}\Big(\Big\{\sum_k\delta_{dm}\,\langle n|\,A^\alpha\,|k\rangle\,\langle k|\,A^\alpha\,|c\rangle\,e^{i(\varepsilon_k-\varepsilon_c)\tau}\\ &+\langle n|\,A^\alpha\,|c\rangle\,\langle d|\,A^\alpha\,|m\rangle\,e^{i(\varepsilon_n-\varepsilon_c)\tau}\Big\}C(\tau)\\ &+\Big\{\sum_k\delta_{cn}\,\langle d|\,A^\alpha\,|k\rangle\,\langle k|\,A^\alpha\,|m\rangle\,e^{i(\varepsilon_d-\varepsilon_k)\tau}\\ &+\langle n|\,A^\alpha\,|c\rangle\,\langle d|\,A^\alpha\,|m\rangle\,e^{i(\varepsilon_d-\varepsilon_m)\tau}\Big\}C(-\tau)\Big)\,\langle c|\,\rho\,|d\rangle\,d\tau.\end{aligned}$$

Das Integral wurde durch 2.51 gelöst. Damit ist nun

$$\begin{aligned}\langle n|\,\dot{\rho}\,|m\rangle = & \qquad (2.57)\\ &\frac{1}{2}\sum_{c,d}\sum_{\alpha=1}^{N}\Big(\Big\{\sum_k\delta_{dm}\,\langle n|\,A^\alpha\,|k\rangle\,\langle k|\,A^\alpha\,|c\rangle\,s(\lambda_{ck})\\ &-\langle n|\,A^\alpha\,|c\rangle\,\langle d|\,A^\alpha\,|m\rangle\,s(\lambda_{cn})\Big\}\\ &+\Big\{\sum_k\delta_{cn}\,\langle d|\,A^\alpha\,|k\rangle\,\langle k|\,A^\alpha\,|m\rangle\,s(\lambda_{dk})\\ &-\langle n|\,A^\alpha\,|c\rangle\,\langle d|\,A^\alpha\,|m\rangle\,s(\lambda_{dm})\Big\}\Big)\,\langle c|\,\rho\,|d\rangle\end{aligned}$$

die Gleichung, welche in QuTiP [PJ, S.50] implementiert wurde ohne säkulare Näherung. Hiermit sind wir in der Lage eine Mastergleichungen für die Diagonalelemente von $\dot{\rho}$ aufzustellen. Es ist

$$\begin{aligned}\langle i|\,\dot{\rho}\,|i\rangle = \\ &\frac{1}{2}\sum_{c,d}\sum_{\alpha=1}^{N}\Big(\Big\{\sum_{k}\delta_{di}\,\langle i|\,A^{\alpha}\,|k\rangle\,\langle k|\,A^{\alpha}\,|c\rangle\,s(\lambda_{ck}) \\ &-\langle i|\,A^{\alpha}\,|c\rangle\,\langle d|\,A^{\alpha}\,|i\rangle\,s(\lambda_{ci})\Big\} \\ &+\Big\{\sum_{k}\delta_{ci}\,\langle d|\,A^{\alpha}\,|k\rangle\,\langle k|\,A^{\alpha}\,|i\rangle\,s(\lambda_{dk}) \\ &-\langle i|\,A^{\alpha}\,|c\rangle\,\langle d|\,A^{\alpha}\,|i\rangle\,s(\lambda_{di})\Big\}\Big)\,\langle c|\,\rho\,|d\rangle\,.\end{aligned} \tag{2.58}$$

Entscheidend ist, dass diese Gleichung ohne säkulare Näherung nicht nur Diagonalelemente verknüpft. Getrennt nach diagonal und nicht diagonal Elementen ergibt sich

$$\begin{aligned}\langle i|\,\dot{\rho}\,|i\rangle = \\ &\frac{1}{2}\sum_{c=d=r}\sum_{\alpha=1}^{N}\Big(-2|\,\langle i|\,A^{\alpha}\,|r\rangle\,|^{2}s(\lambda_{ri}) \\ &+\sum_{k}\delta_{ri}\Big(\langle i|\,A^{\alpha}\,|k\rangle\,\langle k|\,A^{\alpha}\,|r\rangle+\overline{\langle i|\,A^{\alpha}\,|k\rangle\,\langle k|\,A^{\alpha}\,|r\rangle}\Big)\,s(\lambda_{rk})\Big)\,\langle r|\,\rho\,|r\rangle \\ &+(\textit{nicht diagonal Elemente})\end{aligned} \tag{2.59}$$

für die diagonalen Einträge, wobei ausgenutzt wurde, dass die Systemoperatoren hermitesch sind. Für die nicht diagonalen Einträge gilt nach 2.6.1

$$\begin{aligned}\langle i|\,\dot{\rho}\,|i\rangle = \\ &\frac{1}{2}\sum_{c\neq d}\sum_{\alpha=1}^{N}\Big(\Big\{\sum_{k}\delta_{di}e^{-i(\lambda_{kc}-\lambda_{ki})t}\,\langle i|\,A^{\alpha}\,|k\rangle\,\langle k|\,A^{\alpha}\,|c\rangle\,s(\lambda_{ck}) \\ &-e^{-i(\lambda_{di}-\lambda_{ci})t}\,\langle i|\,A^{\alpha}\,|c\rangle\,\langle d|\,A^{\alpha}\,|i\rangle\,s(\lambda_{ci})\Big\} \\ &+\Big\{\sum_{k}\delta_{ci}e^{-i(\lambda_{kd}-\lambda_{ki})t}\,\langle d|\,A^{\alpha}\,|k\rangle\,\langle k|\,A^{\alpha}\,|i\rangle\,s(\lambda_{dk}) \\ &-e^{-i(\lambda_{di}-\lambda_{ci})t}\,\langle i|\,A^{\alpha}\,|c\rangle\,\langle d|\,A^{\alpha}\,|i\rangle\,s(\lambda_{di})\Big\}\Big)\,\langle c|\,\rho\,|d\rangle \\ &+(\textit{diagonal Elemente})\end{aligned} \tag{2.60}$$

im Wechselwirkungsbild für die Systemoperatoren. Die Säkulare Näherung führt zur Entkopplung der diagonalen und nichtdiagonalen Elemente. Bei der säkularen Näherung

wird angenommen, dass

$$\lambda_{ij} - \lambda_{fg} = \varepsilon_i - \varepsilon_j + \varepsilon_g - \varepsilon_f = 0 \tag{2.61}$$

gilt, womit die Oszillationsterme aus 2.60 verschwinden. Es kann daher angenommen werden, dass

$$e^{-i(\lambda_{ij}-\lambda_{fg})t} \to \delta_{if}\delta_{jg} \tag{2.62}$$

gilt. Diese Näherung ist für t gegen unendlich gerechtfertigt, wenn

$$\varepsilon_i - \varepsilon_j + \varepsilon_g - \varepsilon_f \gg |\mathrm{Re}(L_{ijgf})|, \tag{2.63}$$

gilt, wobei L der Tensor aus 2.37 ist. Die Näherung entspricht einer Mittlung über die Zeit, welche für große Zeiten gerechtfertigt ist. [Tho98, S.13, S.20]. In 2.60 finden sich nun in allen Summanden ein δ_{cd} oder ein äquivalenter Ausdruck. Da in 2.60 explizit über $c \neq d$ summiert wurde verschwindet dieser Term. Das heißt nur 2.59 liefert einen Beitrag zu den Diagonalelementen. Die Diagonalen sind damit von den nicht Diagonalen entkoppelt. Die Mastergleichung in säkularer Näherung erhält nun zusätzlich die Positivität des Dichteoperators. Außerdem kann die Mastergleichung nun in die Lindbladform 2.3 überführt werden [Sch, S.30 ff]. Die Gleichung erhält also insbesondere die Spur und es gilt

$$\sum_i \langle i|\,\dot{\rho}\,|i\rangle = \frac{d}{dt}\sum_i \langle i|\,\rho\,|i\rangle = 0, \tag{2.64}$$

wodurch die Wahrscheinlichkeitserhaltung formuliert wird.

2.6.2. Existenz des stationären Zustands

Da die Diagonalelemente von den nicht Diagonalelementen entkoppeln, kann die Mastergleichung durch das Vertauschen von Zeilen in Blockgestalt

$$\begin{pmatrix} \mathrm{diag}\dot{\rho} \\ \dot{\tilde{\rho}} \end{pmatrix} = \begin{pmatrix} \hat{L} & 0 \\ 0 & \tilde{L} \end{pmatrix} \begin{pmatrix} \mathrm{diag}\rho \\ \tilde{\rho} \end{pmatrix}, \tag{2.65}$$
$$\text{mit } \mathrm{diag}\rho \in \mathbb{C}^n,\ \tilde{\rho} \in \mathbb{C}^{n^2-n},\ \hat{L} \in \mathbb{C}^{n\times n},\ \tilde{L} \in \mathbb{C}^{n^2-n\times n^2-n}$$

gebracht werden. Es findet sich stets eine Matrix $S \in \mathbb{C}^{n^2\times n^2}$ mit $\det S = 1$, sodass für die Matrix aus 2.37

$$L = S \begin{pmatrix} \hat{L} & 0 \\ 0 & \tilde{L} \end{pmatrix} \tag{2.66}$$

gilt. Hierbei ist in 2.65 diagρ ein Vektor, der nur die Diagonalelement von ρ enthält und $\tilde{\rho}$ ein Vektor der nur die anderen Elemente enthält.

Aus 2.64 und der Tatsache, dass die Diagonalelemente von den nicht Diagonalelementen entkoppelt sind ergibt sich, dass eine Mastergleichung für die Diagonalelemente von den anderen linear abhängig ist und somit ist

$$\det(\hat{L}) = 0. \tag{2.67}$$

Es gilt außerdem

$$\det \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} = \det A \det B. \tag{2.68}$$

Daher ergibt sich die Determinante von L zu

$$\det L = \det S \det \hat{L} \det \tilde{L} = 0. \tag{2.69}$$

Da die Determinante von L verschwindet folgt direkt, dass ein Eigenwert von L verschwinden muss, denn 0 ist dann eine Nullstelle des charakteristischen Polynoms. Es ist

$$CP(\lambda) = \det(\lambda \mathbb{1} - L) \tag{2.70}$$

das charakteristische Polynom von L. Nun ist $\lambda = 0$ immer eine Nullstelle, denn

$$CP(0) = \det(0\mathbb{1} - L) = 0 - \det(L) = 0. \tag{2.71}$$

Das heißt es existiert ein Dichteoperator, sodass

$$L\mathrm{vec}(\rho_\infty) = 0 \tag{2.72}$$

gilt. Geeignet normiert ist ρ_∞ der stationär Zustand, für den das System im thermischen Gleichgewicht ist [Mis11, S. 11 ff.]. Über die Eindeutigkeit konnte keine Aussage getroffen werden. Der stationäre Zustand kann entartet sein, wie im Beispiel 5.2 klar wird.
Des weiteren ist nun ersichtlich, dass dieser stationäre Zustand nur auf der Diagonalen besetzt ist. Denn ist

$$\hat{L}\mathrm{diag}\rho_\infty = 0 \tag{2.73}$$

der stationäre Zustand, dann kann dieser auf einen Eigenvektor von L ausgedehnt werden indem er mit dem Wert null auf einen Vektor aus $\mathbb{C}^{n^2 \times n^2}$ erweitert wird. Es ist

$$0 = \begin{pmatrix} \hat{L} & 0 \\ 0 & \tilde{L} \end{pmatrix} \begin{pmatrix} \mathrm{diag}\rho_\infty \\ 0 \end{pmatrix} \tag{2.74}$$

der Zustand der bei geeigneter Normierung dem stationären Zustand entspricht. Da die Kopplung im betrachteten Fall explizit erlaubt wird, kann nicht davon ausgegangen werden, dass nur die Diagonale besetzt ist. In Beispiel 5.1 zeigt sich, dass für kleine Temperaturen diese Entkopplung hier gegeben ist.

3. Implementation in Qutip

3.1. QuTiP

QuTip ist ein Python Paket von P.D. Nation, J.R. Johansson und F.Nori [PNF] .QuTiP steht für Quantum Toolbox in Python. Es baut auf den Python Paketen NumPy und SciPy auf. Quantenobjkete werden in QuTiP als Objekte der Klass Qobj behandelt. Ein Objekt aus Qobj hat die Attribute, Data, Type, Hermitian, Dimensions und Shape. Das Attribut Data beinhaltet ein Array der zugehörigen Matrix oder Vektors des

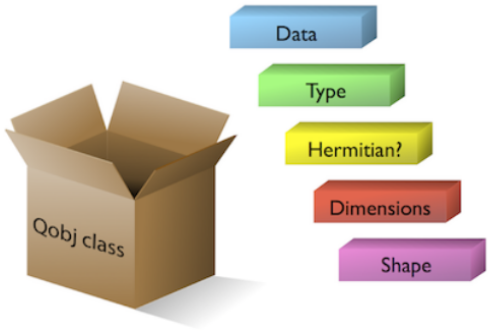

Abbildung 1: Klasse Qobj

Quantenobjekts. Um Speicherplatz zu sparen werden alle Arrays als Sparse Matrizen gespeichert. Standardmäßig wird hier das CSR-Format verwendet. Das Attribut Type beinhaltet die Angabe über welchen Art von Quantenobjekt es sich handelt. Type kann die Werte „ket“, „bra“, „oper“ oder „super“ annehmen. Das Attribut Hermitian gibt Auskunft ob der Operator hermitesch ist oder nicht und entspricht daher entweder „True“ oder „False“. Das Attribut „Shape“ beinhaltet einen Tupel aus zwei Zahlen, welche die Dimension der Matrix oder Vektors angeben. Im Attribut Dimensions wird gespeichert ob das Quantenobjekt aus einem Tensoprodukt hervorgegangen ist. Dies kann zu Problemen führen, denn QuTiP wirft einen Fehler wenn Objekte miteinander multipliziert werden deren „Shape“ Attribute kompatibel, aber deren „Dimension“ nahelegen, dass sie eine andere Struktur besitzen.
Der Konstruktor der Klasse Qobj akzeptiert eine Liste oder ein Array-Like Python Objekt und erzeugt daraus ein entsprechendes Qobj Element. Außerdem stehen eine Reihe vordefinierter Operatoren und Vektoren zur Verfügung. Zum Beispiel kann mit qutip.sigmaz() sofort ein entsprechender Operator erzeugt werden. Darüber hinaus sind die grundlegende mathematische Funktionen für Objekte der Klasse Qobj implementiert worden. Es finden sich nützliche Funktionen wie „qutip.Qobj.transform(ekets)“ für einen Basiswechsel bezüglich einer gegebenen Basis, oder „qutip.Qobj.eigenstates()“ um Eigenwerte und Eigenvektoren eines Operators zu bestimmen. Für eine Vollständige Liste sei auf [PJ] verwiesen.

3.2. Bloch-Redfield Methoden

In QuTiP existiert die Klasse bloch_redfield, welche wiederum drei Methoden beinhaltet.

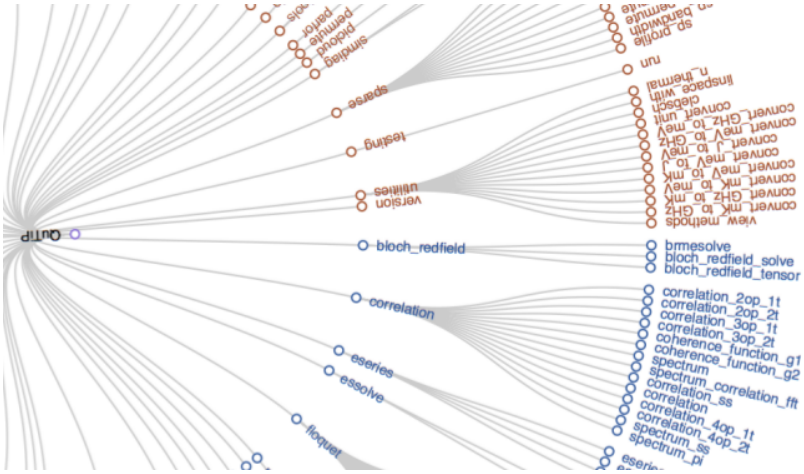

Abbildung 2: Ausschnitt aus dem Baumdiagramm des QuTip Paketes

3.2.1. Methode bloch_redfield_tensor

Die Methode bloch_redfield_tensor berechnet den Tensor wie er in 2.57 aufgestellt wurde. Die Funktion verlangt als Eingabe den Hamiltonoperator des Systems, eine Liste von Systemoperatoren und Spektralfunktionen. Optional kann eine Liste von „collapse operators“ übergeben werden. Außerdem ist es möglich die Standarteinstellung use_secular=True zu ändern.

```
def bloch_redfield_tensor(H,a_ops,spectra_cb,c_ops=None,use_secular=True):
```

Die Funktion gibt den Bloch-Redfield Tensor in der Eigenbasis des Hamilton Operators zurück. Zusätzlich hat die Funktion die Eigenvektoren und Eigenwerte des Hamiltonoperators als Rückgabewert, da diese ohnehin berechnet werden müssen. Außerdem wird in QuTiP angenommen, dass zwischen den Systemoperatoren keine Korrelation besteht, sodass $A^\alpha A^\beta = 0$ ist. Standardmäßig verwendet Qutip die säkulare Näherung. Die Methode bietet die Möglichkeit den Tensor ohne diese Näherung zu berechnen. Der Realteil des Bloch-Redfield Tensor ist in QuTiP durch

$$R_{abcd} = -\frac{\hbar^{-2}}{2}\sum_\alpha \Bigg\{ \delta_{bd}\sum_n \chi^\alpha_{an}\chi^\alpha_{nc}s_\alpha(\omega_{cn}) - \chi^\alpha_{ac}\chi^\alpha_{db}s_\alpha(\omega_{ca}) + \delta_{ac}\sum_n \chi^\alpha_{dn}\chi^\alpha_{nb}s_\alpha(\omega_{dn}) - \chi^\alpha_{ac}\chi^\alpha_{db}s_\alpha(\omega_{db}) \Bigg\}. \tag{3.1}$$

implementiert, wobei hier alle Operatoren in der Eigenbasis des Hamiltonoperators sind. QuTiP macht von der Sparse-Eigenschaft des Systemoperatoren keinen Gebrauch.

Zuerst berechnet bloch_redfield_tensor die Eigenvektoren und Eigenwerte des Hamiltonoperators.

```
evals, ekets = H.eigenstates()
```

Im nächsten Schritt werden die Energiedifferenzen in eine Matrix geschrieben um diese später elementweise zu multiplizieren wie in 2.50 bzw. 2.51.

```
for n in range(N):
        for m in range(N):
            W[m, n] = np.real(evals[m] - evals[n])
```

Sowohl die Systemoperatoren als auch der Hamitlonoperator werden in die Eigenbasis des Hamiltonoperators transformiert.

```
Heb = H.transform(ekets)
for k in range(K):
        A[k, :, :] = a_ops[k].transform(ekets).full()
```

Hierbei werden die zuvor dünnbesetzten als CSR-Matrizen gespeicherte Matrizen durch die Funktion 'full()' als gewöhnliche vollbesetzte Python Arrays übergeben. Zuletzt wird der Bloch-Redfield Tensor gemäß 3.1 berechnet.

```
R = -1.0j * (spre(Heb) - spost(Heb))
R.data = R.data.tolil()
for I in range(N * N):
        a, b = vec2mat_index(N, I)
        for J in range(N * N):
                c, d = vec2mat_index(N, J)

            # unitary part: use spre and spost above, same as this:
            # R.data[I,J] = -1j * W[a,b] * (a == c) * (b == d)

            if use_secular is False or abs(W[a,b]-W[c,d]) < dw_min/10.0:

                # dissipative part:
                for k in range(K):
                    # for each operator coupling
                                        #the system to the environment

                    R.data[I, J] += ((A[k, a, c] * A[k, d, b] / 2) *
                                     (spectra_cb[k](W[c, a]) +
                                      spectra_cb[k](W[d, b])))
                    s1 = s2 = 0
                    for n in range(N):
                        s1 += A[k,a,n] * A[k,n,c] * spectra_cb[k](W[c,n])
                        s2 += A[k,d,n] * A[k,n,b] * spectra_cb[k](W[d,n])

                    R.data[I, J] += - (b==d) * s1 / 2 - (a==c) * s2/2
R.data = R.data.tocsr()
return R, ekets
```

Der vollständige Quellcode ist in (referenz auf Anhange) angehängt.

Säkulare Näherung in Qutip In 2.6 wird erläutert wie durch die säkulare Näherung die diagonalen Elemente von den nicht diagonalen Elemente entkoppelt werden können. Hierzu werden nur Terme berücksichtigt, für die

$$e^{-i(\lambda_{ij}-\lambda_{lk})t} \langle i| A_D(t) |j\rangle \langle k| A_D(t) = \langle i| A_D(t) |j\rangle \langle k| A_D(t) |l\rangle \rightarrow \lambda_{ij} = \lambda_{lk} \tag{3.2}$$

gilt. In Qutip ist die optionale säkulare Näherung dadurch realisiert, dass nur Terme berücksichtigt werden, für die

$$|\lambda_{ab} - \lambda_{cd}| \ll \frac{1}{\tau_{\text{decay}}} \tag{3.3}$$

gilt [PJ, S.51]. Da τ_{decay} nicht bekannt ist werden in Qutip nur Terme berücksichtigt, für die

$$|\lambda_{ab} - \lambda_{cd}| < \frac{\lambda_{\text{min}}}{10}, \tag{3.4}$$

wobei

$$\lambda_{\text{min}} = \min\{\lambda_{ij},\ i, j = 1, 2, \ldots, N | \lambda_{ij} \neq 0\} \tag{3.5}$$

ist.

3.2.2. Methode bloch_redfield_solve

Die Methode bloch_redfield_solve löst die Differentialgleichung 2.33 für vorgegebene Zeitschritte. Die Funktion nimmt als Parameter den Bloch-Redfield Tensor, Vektoren die eine Basis aufspannen, einen Startzustand und eine Liste von Operatoren um deren Erwartungswerte zu den vorgegebenen Zeiten zu berechnen. Die Funktion gibt eine Liste dieser Erwartungswerte zurück.

3.2.3. Methode brmesolve

Diese Methode kombiniert die anderen beiden Methoden aus der Klasse qutip.bloch_redfield. Zuerst ruft die Methode die Methode bloch_redfield_tensor auf, um mit den Ergebnissen die Differentialgleichung 2.33 mithilfe von bloch_redfield_solve für vorgegebene Zeitschritte zu lösen.

4. Numerische Approximation des stationären Zustands

4.1. Das Eigenwertproblem

Im Folgenden wird die Bloch-Redfield Mastergleichung 2.32 in den Näherungen nach Born und Markov 2.4 betrachtet. Es konnte gezeigt werden, dass dieses Problem als ein lineares Differentialgelichungssystem 2.33 geschrieben werden kann.
Vom besonderen Interesse sind die Lösungen von 2.33, für die

$$\mathrm{vec}(\dot{\rho}) = L\mathrm{vec}(\rho) = 0 \tag{4.1}$$

gilt. Dadurch ist ein Eigenwertproblem gegeben. Ziel ist es einen Eigenvektor zum Eigenwert null zu finden. Dies ist bei geeigneter Normierung der stationäre Zustand.

4.1.1. Dimension des Problems

Der Operator L ist durch die Gleichung 2.35 definiert. Hierbei müssen mehrere Tensorprodukte gebildet werden. Ausgehend von den Freiheitsgraden des Systems

$$\dim(H_S \in \mathbb{C}^{n\times n}) = n \tag{4.2}$$

ergibt sich die Dimension des Operators L zu

$$\begin{aligned} \dim(L) = \dim(H_S \otimes \mathbb{1}_{n\times n}) = n^2 \\ \Rightarrow L \in \mathbb{C}^{n^2\times n^2}. \end{aligned} \tag{4.3}$$

Es ist zu erwarten, dass das Problem in $O(n^4)$ liegt.

Beispiel Der Jaynes-Cummings Hamiltonoperator

$$H_{JC} = \frac{1}{2}\Delta E(\sigma_z \otimes N) + g\left(\sigma_x \otimes (a + a^\dagger)\right) + \omega\left(\mathbb{1} \otimes (a^\dagger + a)\right) \tag{4.4}$$

beschreibt beispielsweise ein Atom mit zwei unterschiedlichen Anregungszuständen in einem Laser. Das System soll an mehrere Resonatoren koppeln. Es bezeichnet N den Zähloperator gemäß dem bosonischen Vernichtungsoperator a. ΔE ist der Energieunterschied zwischen angeregtem und nicht angeregtem Zustand, ω die Frequenz der Photonen und g die Kopplungsstärke. Die Größe des Systems hängt nun von der Größe des Resonators ab. Es ist

$$\dim(H_{JC}) = 2n. \tag{4.5}$$

Damit ist die Dimension von L

$$\dim(L) = (2n)^2. \tag{4.6}$$

Für 10 Photonen ist demnach

$$\dim(L) = (2 \cdot 10)^2 = 400 \rightarrow L \in \mathbb{C}^{400\times 400}. \tag{4.7}$$

4.2. Potenzenmethode

Die Potenzenmethode ist ein iteratives Verfahren um den betragsmäßig größten Eigenwert und den entsprechenden Eigenvektor einer Matrix zu bestimmen.
Es seien $\lambda_1 > \lambda_2 \geq \lambda_3 \cdots \geq \lambda_n$ die Eigenwerte eines Operators $A \in L(\mathbb{S})$ und $\{|i\rangle\}$ die zugehörigen Eigenvektoren. Für einen beliebigen Startvektor $|\Psi\rangle \in \mathbb{S}$ konvergiert die Reihe

$$|\Psi_{k+1}\rangle = A\,|\Psi_k\rangle \tag{4.8}$$

gegen $|1\rangle$.

Begründung Falls die Eigenvektoren eine Basis von $\mathbb{S}$ bilden gilt für ein beliebigen Startvektor $|\Psi\rangle \in \mathbb{S}$ die Entwicklung

$$|\Psi_0\rangle = \sum_i^N c_i\,|i\rangle\,. \tag{4.9}$$

Nun kann das kte Reihenglied als

$$\begin{aligned} |\Psi_k\rangle = A^k\,|\Psi_0\rangle = \sum_i^N c_i A^k\,|i\rangle = \sum_i^N c_i \lambda_i^k\,|i\rangle \\ = c_1(\lambda_1)^k \sum_i^N \frac{c_i}{c_1}\left(\frac{\lambda_i}{\lambda_1}\right)^k |i\rangle \end{aligned} \tag{4.10}$$

geschrieben werden. Für große k verschwinden alle Summanden bis auf den Ersten, da $|\frac{\lambda_j}{\lambda_1}| < 1$ für alle $j > 1$ ist. Die Konvergenzgeschwindigkeit hängt maßgeblich von $|\frac{\lambda_2}{\lambda_1}|$ ab. Der Eigenwert kann mit dem Rayleigh Koeffizienten approximiert werden. Es ist

$$r_k := \frac{\langle\Psi_k|A|\Psi_k\rangle}{\langle\Psi_k|\Psi_k\rangle} \tag{4.11}$$

der Rayleigh Koeffizient. In einem Programm kann die Potenzenmethode in wenigen Zeilen realisiert werden.

```
def PowerIteration(A,maxiter):
```

```
y=random
for k =0,1,... maxiter
        z = A*y
        r=y.dag()*z
        y=z/z.norm()
return(z,r)
```

Hierbei konvergiert z gegen den entsprechenden Eigenvektor. Um keinen Overflow zu erhalten wird der Vektor in jedem Schritt normiert.

4.3. Inverse Potenzenmethode mit Shift

Um einen anderen Eigenvektor als den zum betragsmäßig größten Eigenwert zu bestimmen kann die Inverse Potenzenmethode mit Shift verwendet werden. Die Methode funktioniert analog zur Potenzenmethode 4.2. Es wird die Tatsache ausgenutzt, dass für einen Shift an eine Matrix, das Spektrum entsprechend verschoben werden kann.

Lemma 4.1. *(verschobenes Problem) Wenn $|\Psi\rangle \in \mathbb{S}$ ein Eigenvektor von $A \in L(\mathbb{S})$ zum Eigenwert λ ist, so ist $|\Psi\rangle$ ein Eigenvektor von $A - \mu\mathbb{1}$ zum Eigenwert $\lambda - \mu$.*

Des weiteren gilt ein Zusammenhang für das Inverse Problem.

Lemma 4.2. *(inverses Problem) Wenn $|\Psi\rangle \in \mathbb{S}$ ein Eigenvektor von $A \in L(\mathbb{S})$ zum Eigenwert λ ist, so ist $|\Psi\rangle$ ein Eigenvektor von A^{-1} zum Eigenwert $\frac{1}{\lambda}$.*

Beweis von 4.1 Sei $|\Psi\rangle \in \mathbb{S}$ ein Eigenvektor von $A \in L(\mathbb{S})$ zum Eigenwert λ dann gilt

$$\begin{aligned}(A - \mu\mathbb{1})\,|\Psi\rangle &= A\,|\Psi\rangle - \mu\,|\psi\rangle \\ &= (\lambda - \mu)\,|\psi\rangle\end{aligned} \tag{4.12}$$

Beweis von 4.2 Sei $|\Psi\rangle \in \mathbb{S}$ ein Eigenvektor von $A \in L(\mathbb{S})$ zum Eigenwert λ dann gilt

$$\begin{aligned}A\,|\Psi\rangle &= \lambda\,|\Psi\rangle \\ \Leftrightarrow \frac{1}{\lambda}\,|\Psi\rangle &= A^{-1}\,|\Psi\rangle\end{aligned} \tag{4.13}$$

Nach 4.2 konvergiert die Potenzenmethode gegen den betragsmäßig größten Eigenwert und den zugehörigen Eigenvektor von $A \in L(\mathbb{S})$. Um einen anderen Eigenvektor zu bestimmen kann die Potenzenmethode auf den Operator

$$B = (A - \mu\mathbb{1})^{-1} \tag{4.14}$$

angewendet werden. Mit 4.2 und 4.1 haben A und B dieselben Eigenvektoren, allerdings hat sich das Spektrum geändert. Die Eigenwerte von B sind durch

$$\kappa_i = \frac{1}{\lambda_i - \mu} \tag{4.15}$$

gegeben. Der betragsmäßig größte Eigenwert von B ist derjenige, für welchen der Abstand

$$|\lambda_j - \mu| = \delta \tag{4.16}$$

minimal wird. Um den stationären Zustand zu bestimmen wird μ entsprechend klein gewählt um eine vernünftige Approximation an null zu sein. Allerdings muss μ groß genug gewählt werden, sodass es möglich ist eine Inverse zu bestimmen.
Das Programm aus 4.2 kann dementsprechend angepasst werden.

```
def Inverse_PowerIteration(A,maxiter):
y=random
for k =0,1,... maxiter
        solve_for_x: (A-mu*Id)x=y //mit GMRES Verfahren
        y=x/x.norm()
        r=y.dag()*A*y //Rayleigh Koeffizient
return(z,r)
```

Der Aufwand für die inverse Potenzemethode ist im Vergleich zur normalen Potenzemethode größer, da in jedem Schritt die Inverse von $B = A - \mu\mathbb{1}$ berechnet werden muss. Hinzu kommt, dass bei dem Problem 2.33 welches hier betrachtet werden soll der Operator A sehr groß und schwierig zu konstruieren ist. Zur Berechnung der Inversen wird daher das GMRES verfahren verwendet, sodass der verwendete Speicherbedarf variabel ist.

4.4. GMRES

Die Abkürzung GMRES steht für 'generalized minimal residual method'. Das GMRES Verfahren ist ein Krylowraum Verfahren welches iterativ die Lösung eines linearen Gleichungssystem

$$Ax = b,\ A \in \mathbb{C}^{n\times n},\ x, b \in \mathbb{C}^n \tag{4.17}$$

berechnet. Entscheidend ist, dass dieses Verfahren die Matrix zum gegebenen linearen Gleichungssystem nicht benötigt, sondern nur eine Auswertung der Matrix berechenbar sein muss. Das macht diese Methode attraktiv für dünnbesetzte Matrizen.

4.4.1. Krylowraum

Definition 4.3. *(Krylowraum) Wenn A eine Matrix in $\mathbb{C}^{n\times n}$ und q ein Vektor in $\mathbb{C}^n$ ist, dann heißt*

$$\mathcal{K}_m(A, q) = span\{q, Aq, A^2q, \ldots, A^{m-1}q\}$$

der m dimensionale Krylowraum zu A und q.

Ziel ist es eine Basis von $\mathcal{K}_m(A, q)$ zu finden, denn wenn $V_m \in \mathbb{C}^{n\times m}$ die Matrix aus den Basisvektoren des Krylowraums ist, dann lässt sich die Näherung an die Lösung des

linearen Gleichungssystems als

$$x_m = V_m y_m,\ y_m \in \mathbb{C}^n \tag{4.18}$$

schreiben. Für die Basis gilt ein Zusammenhang zu einer Oberen Hessenberg Matrix.

Lemma 4.4. *(Hessenberg Matrix) Ist $V_m \in \mathbb{C}^{n\times m}$ die Matrix aus den Basisvektoren des m dimensionalen Krylowraums zu $A \in \mathbb{C}^{n\times n}$ und $q \in \mathbb{C}^n$ dann existiert eine obere Hessenbergmatrix $H \in \mathbb{C}^{m\times m}$ mit*

$$AV_m = V_{m+1}\tilde{H}_m \tag{4.19}$$

wobei $\tilde{H}_m\mathbb{C}^{m+1\times m}$ die Hessenberg Matrix H ist, welche um eine zusätzliche Zeile zu einer oberen rechten Dreiecksmatrix, mit dem Eintrag $\tilde{H}_{m+1,m}$, ergänzt wird.

Vergleiche hierzu [Hoc, S.132]. Eine Hessenbergmatrix ist eine obere rechte Dreiecksmatrix für die zusätzlich die erste untere Nebendiagonale besetzt ist.
Um die Basis und die entsprechende Hessenbergmatrix aus 4.19 zu bestimmen kann der Arnoldi Algorithmus (modifizierter Gram-Schmidt Prozess) verwendet werden. Der Arnoldi Prozess ermittelt in jeder Iteration ein weiteren Basisvektor von $\mathcal{K}_m(A,q)$, sodass diese eine Orhtonormalbasis bilden. Sind Basis und Hessenbergmatrix bekannt, ist zu klären wie der Koeffizientenvektor y_m aus 4.18 zu wählen ist. Beim GMRES Verfahren wird hierzu die Residuennorm als Abschätzung der Fehlernorm minimiert.

Definition 4.5. *(Fehlernorm) Es sei x_m eine Näherung an $\hat{x} = A^{-1}b$, dann ist für eine gegebene Norm $\|\cdot\|$*

$$f_m = \|\hat{x} - x_m\| \tag{4.20}$$

die Fehlernorm an die Näherung x_m

Definition 4.6. *(Residuum) Es sei x_m eine Näherung an $\hat{x} = A^{-1}b$, dann ist das Residuum durch*

$$r_m := b - Ax_m \tag{4.21}$$

definiert.

Da die exakte Lösung $\hat{x}$ nicht bekannt ist kann f_m nicht berechnet werden. Es ist ersichtlich, dass für

$$\|r_m\| = 0 \tag{4.22}$$

x_m die exakte Lösung von $Ax = b$ ist. Es gelten die Ungleichungen [Hoc, S.135]

$$\frac{1}{\mathcal{K}(A)}\frac{\|r_m\|}{\|r_0\|} \leq \frac{\|f_m\|}{\|f_0\|} \leq \mathcal{K}(A)\frac{\|r_m\|}{\|r_0\|}. \tag{4.23}$$

Es bezeichnet $\mathcal{K}(A)$ die Kondition der Matrix A. Für schlecht Konditionierte Matrizen kann die relative Residuuennorm stark von der relativen Fehlernorm abweichen.
Es ist dennoch das Ziel die Residuennorm zu minimieren, da diese Abschätzung eine praktikable Möglichkeit bietet die Fehlernorm zu approximieren. Effektiv ist dies mit einer QR-Zerlegung der Hessenbergmatrix aus 4.19 möglich [Hoc, S.136]. Das Residuum kann mit 4.18 als

$$r_m = b - AV_m y_m \tag{4.24}$$

geschrieben werden. Mit 4.19 ist dann

$$r_m = b - V_{m+1}\tilde{H}_m y_m. \tag{4.25}$$

Die Residuennorm kann nun als

$$\|r_m\| = \left\|(b - V_{m+1}\tilde{H}_m y_m)\right\| \tag{4.26}$$

geschrieben werden. Da $V_{m+1}^{\dagger}$ eine unitäre Matrix ist ändert diese die Norm nicht und es ist

$$\|r_m\| = \left\|(V_{m+1}^{\dagger} b - \tilde{H}_m y_m)\right\|. \tag{4.27}$$

Da $b = v_1$ der erste Basisvektor ist, kann mit $\beta = \|b\|$ die Residuennorm als

$$\|r_m\| = \left\|(\beta e_1 - \tilde{H}_m y_m)\right\| \tag{4.28}$$

geschrieben werden. Durch die Zerlegung von $\tilde{H}_m = Q\tilde{R}$ in eine unitäre Matrix $Q \in \mathbb{C}^{m+1\times m+1}$ und eine obere Rechte Dreiecksmatrix $R \in \mathbb{C}^{m\times m}$, welche analog zu $\tilde{H}_m$ um eine Zeile (hier aus Nullen) erweitert werden muss, ist dann

$$\|r_m\| = \left\|(Q^{\dagger}\beta e_1 - \tilde{R} y_m)\right\| \tag{4.29}$$

ein lineares Ausgleichsproblem. Es sei

$$z := Q^{\dagger}\beta e_1. \tag{4.30}$$

Es ist nun in 4.29

$$\|r_m\| = \left\| \begin{pmatrix} z' \\ z_m \end{pmatrix} - \begin{pmatrix} R \\ 0\ldots 0 \end{pmatrix} y_m \right\| \tag{4.31}$$

Die Residuuennorm ist minimal, falls

$$z' = R y_m. \tag{4.32}$$

Die Residuuennorm ergibt sich dann zu

$$\|r_m\| = |z_m| \tag{4.33}$$

Um die Residuuennorm zu minimieren ist also ein lineares Ausgleichsproblem mit der QR-Zerlegung zu lösen.

Abbruchkriterium Der Krylowraum kann nun solange um weitere Vektoren erweitert werden bis $|z_m|$ aus 4.33 kleiner als eine gewählte Toleranz ist. Sobald dies erreicht ist, kann die Lösung mit 4.18 berechnet werden.
Für jede Iteration des GMRES Verfahrens ist eine Auswertung der Matrix A und eine Reihe von Skalarprodukten notwendig. Die QR-Zerlegung die in jedem Schritt anfällt kann durch eine Givensrotationen pro Iteration ersetzt werden.
Tatsächlich wird die GMRES Methode so implementiert, dass die Dimension des Krylowraums als Parameter mitgegeben werden kann. Die Neustarts sind eine einfache Methode den Speicherverbrauch zu reduzieren. Allerdings wird durch die Neustarts die Konvergenz zum Teil erheblich verlangsamt.

4.4.2. Speicherbegrenzung durch Neustarts

Falls in 4.4.1 die gewählte Toleranz noch nicht erreicht wurde, es allerdings nicht mehr möglich ist einen weiteren (vollbesetzten) Basisvektor zu speichern, kann die Lösung berechnet werden und die Methode mit der aktuellen Näherung neu gestartet werden.

4.5. Abschließende Implementierung (Methoden)

Die abschließende Implementierung zur Berechnung des stationären Zustands kombiniert die genannten numerischen Verfahren zur Berechnung des Eigenvektors und der Inversen. Das Ergebnis besteht aus einem Python Modul br_ statstate.py mit den Methoden

- br_ statstate.evaluation, um die Bloch-Redfield Mastergleichung auszuwerten.
- br_ statstate.inv_ PowIteration, welche die inverse Iteration implementiert.
- br_ statstate.gmres_ ext_ ev, um die Inverse in jedem Schritt der inversen Iteration zu berechnen.
- br_ statstate.statState, welche die vorangegangen Methoden kombiniert und den stationären Zustand berechnet.

Außerdem beinhaltet br_ statstate eine weitere Methode, welche aus dem Quellcode ausgelagert wurden um die Übersichtlichkeit zu wahren. Dies ist die Methode

- br_ statstate.auswertung welche die Methode evaluation aufruft und die notwendige Umformung zwischen Vektoren und Matrizen berechnet.

Die folgenden Abschnitte gehen explizit auf den Quellcode ein und geben die Laufzeiten der Methoden an.

4.6. Auswertung der Bloch-Redfield Mastergleichung

Die Auswertung der Bloch-Redfield Mastergleichung ist in der Methode br_ statstate.evaluation realisiert. Die Methode verlangt als Eingabe den Systemhamiltonoperator als Objekt der Klasse qutip.qobj, eine Liste von Systemoperatoren, ebenfalls aus der Klasse qutip.qobj, eine Liste von spektralen Dichtefunktionen zu den Systemoperatoren, eine Dichtematrix aus qutip.qobj und einen booleschen Parameter „tidyup“, welcher entscheidet ob nach den Basiswechseln im Programm kleine Elemente vernachlässigt werden sollen. Die Methode gibt die Ableitung der Dichtematrix als Element von qutip.quobj zurück.
Aus 4.4.1 geht hervor, dass die Auswertung wesentlich zur Laufzeit des Programms beitragen wird. Um die Bloch-Redfield Mastergleichung für eine gegebene Dichtematrix effektiv auszuwerten wird im wesentlichen die Gleichung 2.35 verwendet.

$$\dot{\rho}(t) = -i\,[H_S, \rho(t)] - \sum_{\alpha,\beta=1}^{N} \Big\{ A^\alpha T^+_{\alpha\beta}\rho(t) - T^+_{\alpha\beta}\rho(t)A^\alpha + \rho(t)T^-_{\beta\alpha}A^\alpha - A^\alpha\rho(t)T^-_{\beta\alpha} \Big\}.$$

In der abschließenden Implementation werden keine Korrelationen zwischen den Systemoperatoren A^α und A^β angenommen, sodass

$$\dot{\rho}(t) = -i\,[H_S, \rho(t)] - \sum_{\alpha=1}^{N} \Big\{ A^\alpha T^+_{\alpha}\rho(t) - T^+_{\alpha}\rho(t)A^\alpha + \rho(t)T^-_{\alpha}A^\alpha - A^\alpha\rho(t)T^-_{\alpha} \Big\}.$$

Um die sparse Eigenschaften der Systemoperatoren auszunutzen ist es sinnvoll die Gleichung nicht in der Eigenbasis des Hamiltonoperator zu implementieren, denn durch den Basiswechsel kann diese Eigenschaft verloren gehen. Zur Berechnung von T^+_α und T^-_α aus 2.35 ist es allerdings notwendig an einer Stelle in die Eigenbasis des Hamiltonoperators zu wechseln, was anhand von 2.5 ersichtlich ist. Die Operatoren T^+_α und T^-_α können mit 2.51 als

$$T^+_\alpha = V^T\big((VA^\alpha V^T) * G\big)V,$$
$$T^-_\alpha = V^T\big((VA^\alpha V^T) * G^T\big)V$$

aufgestellt werden. Die Matrix G ist durch 2.50 elementweise durch die spektrale Dichte s und die Energieverschiebung λ_{ij} als

$$G_{ij} := \frac{1}{2}s(\lambda_{ij})$$

definiert. Um die Matrix G aufzustellen ist es notwendig über alle Einträge zu iterieren. Dies wurde in A.1 als

```
for n in range(N):
    for m in range(N):
        G[n, m] = ohmic_spectrum[i](evals[m] - evals[n])*0.5
```

realisiert. Der Beitrag der beiden For Schleifen zur Laufzeit ist erheblich, da diese expliziten Rechnung (innerhalb von ohmic_ spectrum()) in Python gegenüber kompilierbaren Sprachen wie C inefizient sind. Es ist daher zu überlegen ob der Code in ein C-Modul ausgelagert werden sollte. Allerdings soll es später möglich sein die spektrale Dichte als Python Funktion zu übergeben, was den Aufwand für ein C-Modul erheblich erhöht. Aufgrund der Auswertungen von ohmic_ spectrum sind zur Erstellung von G keine Anwendung von NumPy Methoden möglich.
Mit der Matrix G kann nun T_α^+ und T_α^- berechnet werden. Um die notwendigen Basiswechsel durchzuführen wurden die Eigenvektoren und Eigenwerte des Hamilton Operators mit einer bestehenden QuTiP Funktion berechnet werden.

```
evals,ekets=H.eigenstates()
```

Der Basiswechsel kann ebenfalls mit einer bestehenden QuTiP Funktion (transform) berechnet werden.

```
1   Erg=1.0j*(Rho.data*H.data-H.data*Rho.data)
2
3       k=len(A)
4       for i in range(k):
5
6           for n in range(N):
7               for m in range(N):
8                   G[n, m] = ohmic_spectrum[i](evals[m] - evals[n])*0.5
9
10          Xi=A[i].transform(ekets)
11          ### elementwise multiplication in Eigen-Basis of H
12          Tplus=Qobj(Xi.data.multiply(G))
13          Tminus=Qobj(Xi.data.multiply(np.transpose(G)))
14
15          ### retrasnformation from eigenbasis of H
16          Tplus=Tplus.transform(ekets,True)
17          Tminus=Tminus.transform(ekets,True)
18
19          if tidyup==True :
20              Tplus=Tplus.tidyup()
21
22          ### multiplication using the sparsity of a
23          Erg=Erg-(A[i].data*Tplus.data*Rho.data)
24          Erg=Erg+(Tplus.data*Rho.data*A[i].data)
25          Erg=Erg-(Rho.data*Tminus.data*A[i].data)
26          Erg=Erg+(A[i].data*Rho.data*Tminus.data)
27
28
29
30          ### create an instance of Cobj and return to eigenbasis of H
31      Erg=Qobj(Erg)
32      Erg=Erg.transform(ekets)
```

Da A bzw. Xi ein Objekt von qutip.qobj ist, kann mit .data die darunter liegende sparse Matrix (scipy.sparse.csr_ matrix) aufgerufen werden 3.1. Für dieses Objekt stehen alle SciPy Funktionen wie die elmentweise Multiplikation scipy.sparse.csr_ matrix.multiply zur Verfügung. Diese Funktionen sind für dünnbesetzte Matrizen optimiert, sodass mit dem üblichen * Operator eine effiziente Multiplikation möglich ist. Für die Matrixmultiplikation ist das CSR Format vorteilhaft [Sci15].

4.6.1. Sparse Eigenschaften der auftretenden Operatoren

Die sparse Eigenschaften der Operatorn kann exemplarisch für das Beispiel 5.1 ermittelt werden. Die Temperatur in der spektralen Dichte ist hier so gewählt, dass

$$\beta = \frac{1}{k_B T} = 0,1 \tag{4.34}$$

gilt. Als Maß für die Besetztheit der Matrizen kann der Begriff der Dichte einer Matrix definiert werden. Die Dichte einer Matrix $A \in C^{n \times m}$ ist durch die Anzahl der von null verschiedenen Elemente s als

$$\rho_A := \frac{s}{n \cdot m} \tag{4.35}$$

definiert. In Abbildung 3 ist die Dichte des Systemoperators A, die Dichte von T_0^+ in der

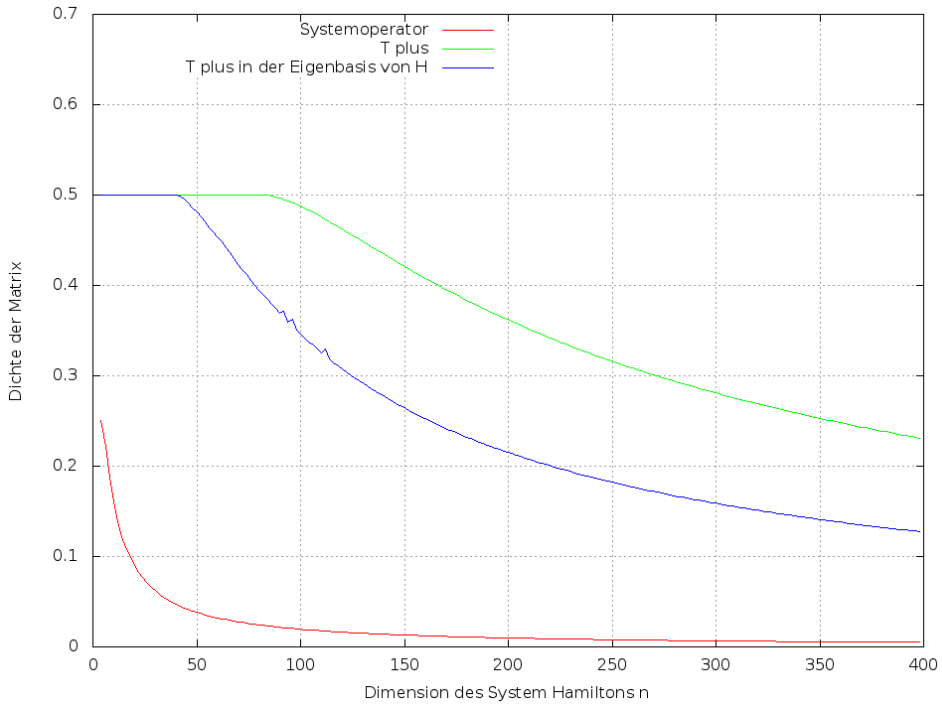

Abbildung 3: Dichte der verschiedenen Operatoren

Eigenbasis des Hamiltonoperators und die Dichte von T_0^+ in der ursprünglichen Basis aufgetragen. Durch den zusätzlichen Basiswechsel ist T_0^+ in der ursprünglichen Basis dichter besetzt als in der Eigenbasis von H.
Abbildung 4 zeigt, dass durch den Befehl

```
if tidyup==True :
        Tplus=Tplus.tidyup()
```

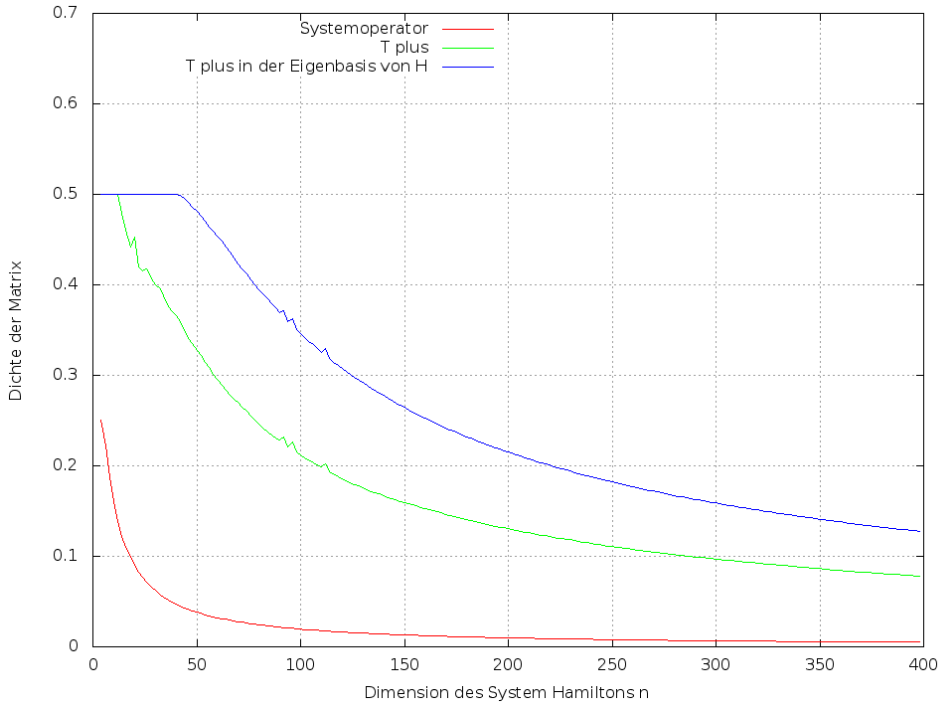

Abbildung 4: Dichte der verschiedenen Operatoren mit tidyup Befehl für T^+

welcher alle Elemente des Operators auf null setzt, die betragsmäßig kleiner als $1.0e-12$ sind, die Dichte des Operators positiv beeinflusst werden kann.

4.6.2. Aufwand und Laufzeit der Auswertung

Zur Auswertung der Bloch-Redfield Gleichung gemäß 2.35 sind bei einem Systemhamiltonoperator der Dimension n und k Systemoperatoren

- eine Bestimmung der Eigenvektoren des Systemhamiltonoperators,
- $3k + 2$ Basiswechsel,
- $n^2 \cdot k$ Auswertungen der ohmschen Dichte,
- $8k$ Matrixmultiplikation mit teils dünnbesetzten Matrizen mit derselben Dimension wie der Systemhamiltonoperator,

notwendig. Im selben Beispiel wie in 4.6.1 kann die Laufzeit der Auswertung in Abhängigkeit von n für ein festes $k = 1$ bestimmt werden. Um die Skalierung der Laufzeit zu bestimmen wird die Dimension des Hamiltonoperators logarithmisch auf der x-Achse

und die Laufzeit logarithmisch auf der y-Achse aufgetragen, denn dann ist

$$t = a \cdot n^x \Rightarrow \frac{t}{t_{max}} = \left(\frac{n}{n_{max}}\right)^x$$
$$\Rightarrow \ln\left(\frac{t}{t_{max}}\right) = x \ln\left(\frac{n}{n_{max}}\right) \tag{4.36}$$

nach einer Normierung mit dem größten Messwert (t_{max}, n_{max}), $t_{max} = a n_{max}^x$, 4.36 eine

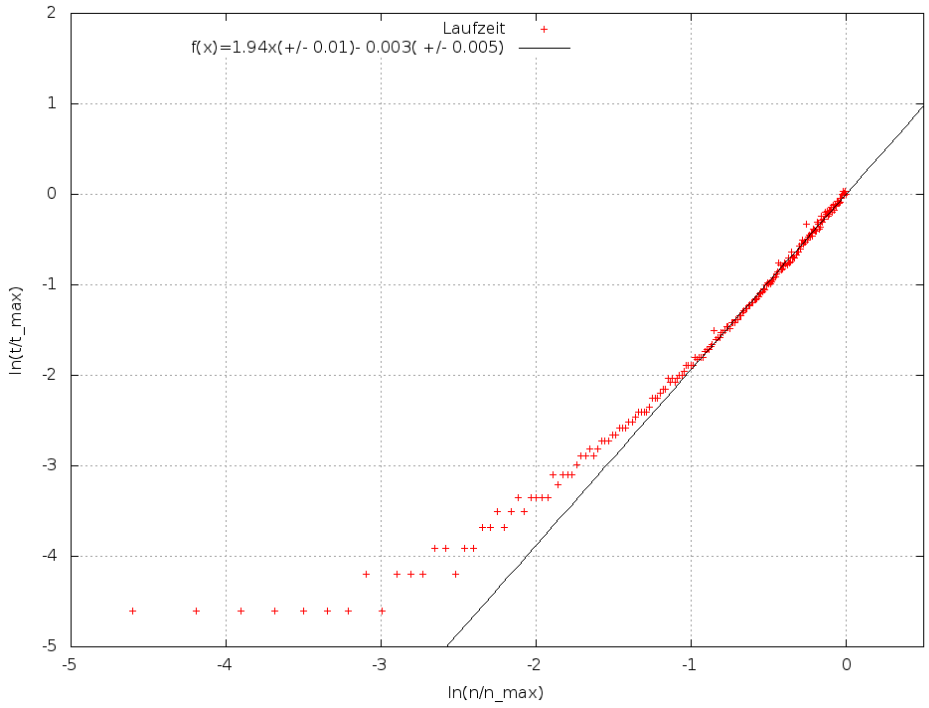

Abbildung 5: Laufzeit der Auswertung

Gerade und die Ordnung kann aus der Steigung direkt abgelesen werden. In Abbildung 5 wurde im linearen Bereich ermittelt, dass sich diese Steigung zu

$$x = 1.94 \ \pm 0.01 \tag{4.37}$$

ergibt. Für festes k liegt die Methode br_ statstate.evaluation also etwa in $O(n^2)$. Dies ist befriedigend, wenn man bedenkt, dass der naive Algorithmus zur Multiplikation zweier vollbesetzter Matrizen in $O(n^3)$ liegt. Die schlechte Auflösung der Zeit für kleine Matrizen in Abbildung 5 ist dadurch zu erklären, dass die Zeitmessung in Python mit time.clock() nur eine beschränkte Auslösbarkeit erlaubt. Da k die Anzahl der Systemoperatoren ist und alle Schritte für jeden Systemoperator (Ausgenommen der Bestimmung der Eigenbasis des Hamiltonoperators) k fach wiederholt werden müssen, skaliert die Methode linear mit k. In Tabelle 1 sind die Laufzeiten auf eine PC mit einem Intel(R) Core(TM) i5-3470 CPU (3.2 GHz) exemplarisch aufgelistet.

Tabelle 1: Laufzeit von br_ statstate.evaluation

Laufzeit in s	**Dimension von H_S**	**Dimension des Bloch-Redfield Tensors L**
0.02	10	100
0.06	50	2500
0.16	100	10000
0.52	200	40000
1.2	300	90000

4.7. GMRES Methode

Das GMRES Verfahren zur Auswertung der Inversen der Bloch-Redfield Gleichung ist in der Methode br_ statstate.gmres_ ext_ ev implementiert. Als Eingabe verlangt die Methode

- den Systemhamiltonoperator als Instanz der Klasse qutip.qobj,
- eine Liste von Systemoperatoren ebenfalls aus qutip.qobj,
- eine Funktion, welche die Bloch-Redfield Mastergleichung löst,
- einen Startoperator aus qutip.qobj als Approximation an die Lösung,
- die Anzahl der Iterationen nach dem die Methode neu gestartet wird,
- einen Shift für die Auswertung,
- einen Parameter der die maximale Anzahl von Neustarts begrenzt,
- den „tidyup“ Wert für br_ statstate.evaluation.

Das Verfahren gibt das Ergebnis der Inversen Bloch-Redfield Gleichung mit einer Verschiebung zurück, das heißt es wird 1.1

$$\dot{\rho} = (L - \mu \mathbb{1})\, \rho,\ L \in \mathbb{C}^{n \times n},\ \rho \in \mathbb{C}^n \tag{4.38}$$

für gegebenes $\dot{\rho}$ nach ρ aufgelöst. Die Methode gibt $\mathrm{vec}(\rho)$ als Liste zurück.
Die Methode berechnet zunächst mithilfe des Arnoldi Algorithmus mit modifiziertem Gram-Schmidt eine Basis von $\mathcal{K}_m((L - \mu \mathbb{1}), r_0)$ wobei r_0 der erste Residuenvektor ist. Der Arnoldi Prozess ist gemäß [Hoc, S. 137] in den Zeilen 13-24 als

```
1       while(k<restart and eps[k]/beta>tol and k<len(x0) ):
2
3           ## if the start vector doesn't lead to an improvement
4           ## throw an error
5           if(eps[0]>1.0e30):
6               print('Unexpected error. With the choosen
7               start vector was no improvement possible')
8               print('returning the startvektor')
9               return x0
10
11
```

```
12          ### Calc the next Krylow
13          omega.append(evaluation(H,S,spectrum,v[k],tidyup)-shift*v[k])
14
15
16          for i in range(k+1):
17              Hess[i,k]=np.vdot(v[i],omega[k])
18              omega[k]=omega[k]-Hess[i,k]*v[i]
19
20          Hess[k+1,k]=np.linalg.norm(omega[k])
21          v.append(omega[k]/Hess[k+1,k])
22
23
24          V[k]=v[k]
25
26          ## Calc QR decomposition with Givensrotations
```

implementiert. In Zeile 13 wird zunächst die Bloch-Redfield Gleichung mit Shift ausgewertet. Der neue Vektor wird dann mit dem Gram-Schmidt Verfahren orthogonalisiert (Zeile 16-18). Außerdem wird die nächste Zeile der Hessenbergmatrix in Zeile 17 berechnet. Der Zusätzliche Eintrag von $\tilde{H}$ wird in Zeile 21 ermittelt.
Wie in Zeile ersichtlich ist berechnet die Methode so lange neue Vektoren bis die Toleranz für das relative Residuum erreicht wurde, oder der maximale Krylowraum bestimmt wurde. Es ist möglich durch den Parameter „restart “möglich die maximale Anzahl der Vektoren zu beschränken um die Methode danach neu zu starten 4.7.1.
In jedem Schritt des Verfahrens muss eine QR-Zerlegung von $\tilde{H}$ bestimmt werden. Hierfür könnten Givensrotationen benutzt werden, die in jedem Schritt eine Rotationsmatrix auf die letzte Spalte der aktuellen Hessenbergmatrix anwenden. Wie die Rotationsmatrix zu wählen ist und der Code implementiert wurde ist in 4.7.3 erläutert. Nach anwenden der Givnesrotation ist es möglich das aktuelle relative Residuum zu bestimmen (vgl. 4.7.3)
Sobald die Toleranz erreicht wurde, oder die Anzahl der Iterationen einen Neustart notwendig machen, wird

```
x=solveQR(Q,R,z)
x=np.dot(np.transpose(Vcalc),x)+x0
```

gesetzt. Gegebenfalls ist es notwendig die Methode mit

```
if(k==restart):
        print("failed to converge in " +str(restart) +" steps. RESTART")
        x=gmres_ext_ev(H,S,spectrum,evaluation,
        b,x,restart,tol,shift,maxiter-1,tidyup)
```

neu zu starten. Der Parameter maxiter dient als Iterationsvariable für die Neustarts. Das erreichen der maximalen Neustarts führt vor Beginn des Arnoldiprozesses zu einem Abbruch durch,

```
if(maxiter==0):
        print('exceeded Maxiter was not able to achieve the tol')
        print("the toleranz is: " +str(beta))
        return x0
```

Ist der Parameter zu groß gewählt kann es passieren, dass Python einen Fehler wirft, da die maximale Rekursionstiefe durch Python beschränkt ist.

4.7.1. Konvergenz der Methode mit Neustarts

Die Methode br_ statstate.gmres_ ext_ ev ist so geschrieben, dass sie einfach für eine andere Auswertung umgeschrieben werden kann.
Um die Funktion zu testen kann eine beliebige Callback Funktion übergeben werden. Die Parameter für die Systemoperatoren, spektralen Dichten und für den Hamilton Operator sind dann bedeutungslos. Um das Verhalten der Funktion zu testen wurde die Matrix FS 760 1 von der Webseite `http://math.nist.gov/MatrixMarket/` verwendet.
In Abbildung 6 ist die schrittweise Approximation für verschiedene Restartlängen ohne Shift aufgetragen. Die Anzahl der notwendigen Iterationen nehmen für kleine Restart-

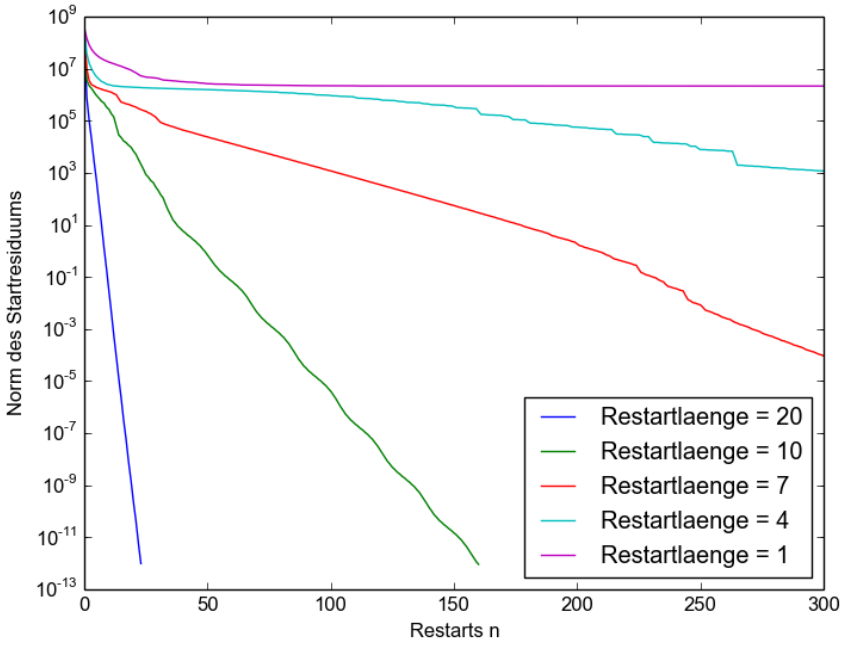

Abbildung 6: Approximation der Inversen durch GMRES

parameter zu. Ist der Parameter zu klein gewählt, kann die Methode stagnieren. Der Restartparameter kann daher nicht beliebig klein gewählt werden, um den Speicherplatz zu minimieren. Ein größerer Parameter führt gewöhnlicherweise zu einer Laufzeitverbesserung.
Allerdings scheint es für die inverse Potenzenmethode nicht notwendig zu sein, dass die Näherung perfekt ist. Im Beispiel wie in 4.6.1 kann dies verdeutlicht werden. Der Systemhamilton wurde hier so gewählt, dass

$$H_S \in \mathbb{C}^{14\times 14} \tag{4.39}$$

gilt, sodass der Bloch-Redfield Tensor

$$L \in \mathbb{C}^{196\times 196} \tag{4.40}$$

ist. Das GMRES Verfahren wurde nun jeweils nach 100 Iterationen neu gestartet, das heißt der Krylowraum ist etwa halb so groß wie die eigentliche Matrix. Es wurden maximal fünf Neustarts erlaubt. Der Shift wurde als $\mu = 1.0e-5$ gewählt. Die Toleranz des GMRES Verfahrens liegt bei $1.0e-12$ und die für die inverse Potenzenmethode bei $1.0e-13$. Zu Beginn konvergiert das GMRES Verfahren nicht, dennoch scheint die kleine Verbesserung der inversen Potenzenmethode zu genügen.
Die Potenzenmethode konvergiert gegen den Eigenvektor $|0\rangle$ daher konvergiert das Star-

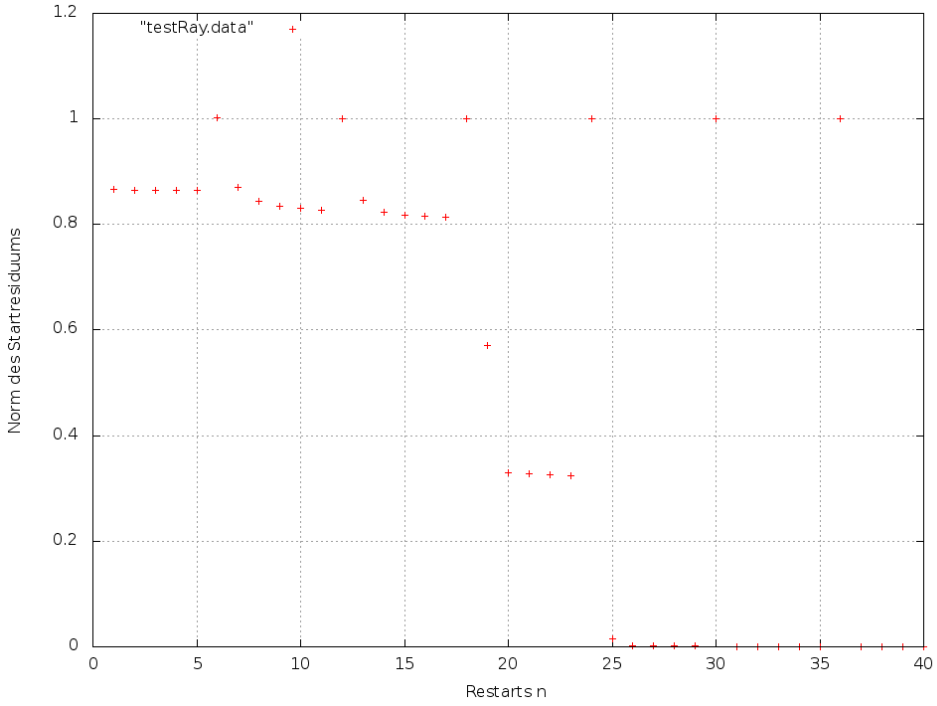

Abbildung 7: Konvergenz von GMRES innerhalb der inversen Potenzenmethode

tresiduum gegen $1-\mu$ denn

$$\begin{aligned}\langle 0|\left(L-\mu\mathbb{1}\right)|0\rangle &= \langle 0|\,L\,|0\rangle - \langle 0|\,\mu\,|0\rangle \\ &= \langle 0|0\rangle - \mu\,\langle 0|0\rangle = 1-\mu \approx 1,\end{aligned} \tag{4.41}$$

da $|0\rangle$ in jedem Schritt normiert wird. Dies ist in Abbildung 7 zu erkennen. Bereits nach einem Schritten scheint die Näherung durch die inverse Potenzenmethode sehr gut zu sein. Nach sieben Schritten erreicht die inverse Potenzenmethode die vorgegebene Toleranz. Es waren daher insgesamt $7\cdot 5\cdot 100 = 3500$ Iterationen der GMRES Methode notwendig. Da für jeden Basisvektor die Bloch-Redfield Mastergleichung ausgewertet werden muss, sind etwa gleich viele Auswertungen notwendig gewesen. Es ist ersichtlich,

dass die maximalen Neustarts der GMRES Methode klein gehalten werden sollten.

4.7.2. Laufzeit der GMRES Methode

Tabelle 2: Aufwand zur Berechnung eines Basisvektors mit dem Arnoldi Verfahren

Berechnung von	**Aufwand**
$\omega[m] = L\omega[m-1]$	Auswertung der Bloch-Redfield Gleichung
$Hess[j,m] = \langle v_i \vert \omega[m] \rangle$	m Skalarprodukte
$v[m] = \omega[m] - \sum_{j=1}^{m-1} Hess[j,m]v[j]$	m SAXPYs
$Hess[m, m+1]$	Ein Skalarprodukt
$v[m]/Hess[m, m+1]$	n Divisionen
Speicheraufwand	m Vektoren

Die Laufzeit der GMRES Methode ist maßgeblich durch die Laufzeit zur Auswertung der Bloch-Redfield Mastergleichung gegeben. Der Aufwand des Arnoldi Verfahrens ist in Tabelle 2 aufgeführt [Hoc, S.134]. Hinzu kommt die QR-Zerlegung die in jeder Iteration mit NumPy berechnet wird. Ist die Konvergenz oder die maximale Iterationslänge erreicht muss zusätzlich mithilfe der QR-Zerlegung durch Rückwärtseinsetzen die Lösung bestimmt werden. Die optimale Laufzeit ist zu erwarten, wenn alle möglichen Basisvektoren bestimmt werden, da Neustarts die Methode verlangsamen 4.7.1.
In 4.6.2 wurde gezeigt, dass die Auswertung wie n^2 skaliert. Das GMRES Verfahren sollte daher schlechter skalieren. Im Beispiel aus 4.6.1 wurde die Laufzeit analog zu 4.6.2 geplottet. Wie in Abbildung 8 erkennbar ist, skaliert die Methode etwa mit $n^{2,3}$ falls

$$H_S \in \mathbb{C}^{n \times n} \tag{4.42}$$

ist. Damit skaliert die Methode etwas schlechter als die Auswertung.

4.7.3. Givensrotationen

Um die QR-Zerlegung einer gegebenen Hessenbergmatrix zu bestimmen sind Givensrotationen, wie sie von Saad und Schultz vorgestellt wurden [SS86], eine effektive Möglichkeit. Die Idee ist es in jeder Spalte das Diagonalelement und die erste Nebendiagonale als einen Vektor aufzufassen, der durch eine Rotationsmatrix so gedreht werden kann, dass das Nebendiagonalelement verschwindet. Es sei $H \in \mathbb{R}$ die Hessenbergmatrix

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1n} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2n} \\ 0 & h_{32} & h_{33} & \cdots & h_{3n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{nn-1} & h_{nn} \end{pmatrix}. \tag{4.43}$$

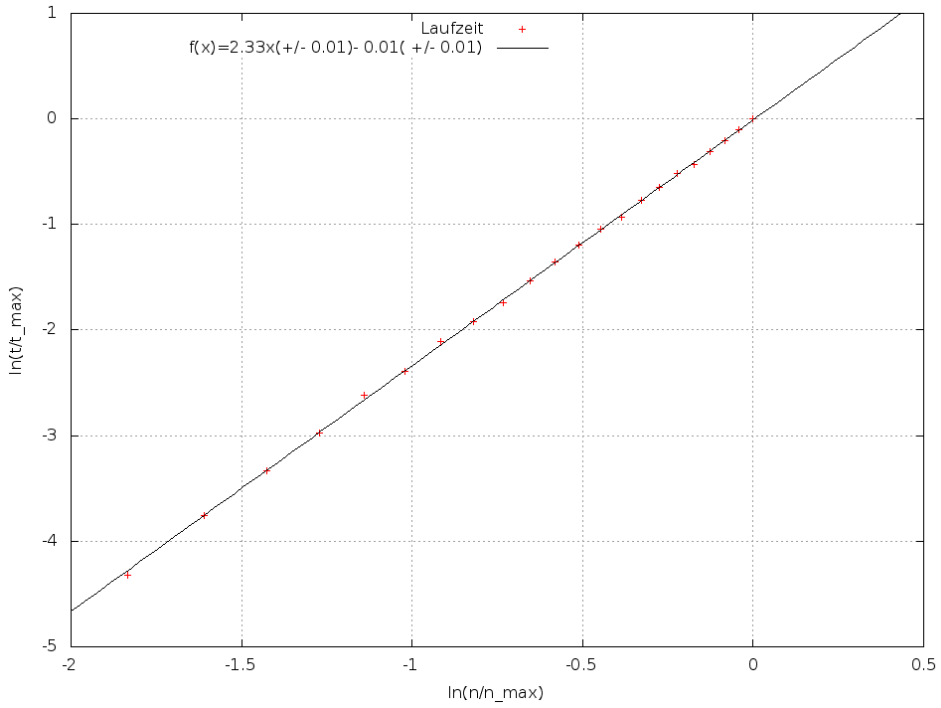

Abbildung 8: Laufzeit der GMRES Methode

Es soll nun in der j-ten Spalte das Elemente $h_{(j+1)j}$ für eine QR-Zerlegung durch Anwenden einer unitären Matrix verschwinden. Es wird daher eine Drehmatrix gesucht, sodass

$$G_j = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} h_{jj} \\ h_{(j+1)j} \end{pmatrix} = \begin{pmatrix} \sqrt{h_{jj}^2 + h_{(j+1)j}^2} = \rho \\ 0 \end{pmatrix} \tag{4.44}$$

ist. Es ist natürlich nicht notwendig den Winkel ϕ explizit zu bestimmen, es genügt die Werte $\cos\phi$ und $\sin\phi$ zu kennen. Man rechnet leicht nach 4.57, dass für

$$cos\phi = \frac{h_{jj}}{\rho}, \tag{4.45}$$

$$sin\phi = \frac{-h_{(j+1)j}}{\rho} \tag{4.46}$$

die Gleichung 4.44 erfüllt ist. Anstatt die Norm ρ des Vektors direkt zu bestimmen ist es numerisch stabiler

$$\tau = |h_{jj}| + |h_{(j+1)j}| \tag{4.47}$$

$$\nu = \rho = \tau \cdot \sqrt{\left(\frac{h_{jj}}{\tau}\right)^2 + \left(\frac{h_{(j+1)j}}{\tau}\right)^2} \tag{4.48}$$

zu verwenden. Da eine Drehmatrix eine unitäre Matrix ist, ändert diese die Norm aus 4.28 nicht. In jedem Schritt des Arnoldi Prozesses kann also eine Givensrotation der Form (für $j \geq 1$)

$$Q_j = \begin{pmatrix} \mathbb{1}_{(\mathbb{j}-\mathbb{1}),(\mathbb{j}-\mathbb{1})} & 0 \cdot \mathbb{1}_{\mathbb{j},\mathbb{2}} \\ 0 \cdot \mathbb{1}_{\mathbb{2},\mathbb{j}} & G_j \end{pmatrix} \tag{4.49}$$

angewendet werden, sodass

$$\|r_{j+1}\| = \|(Q_j z - R_{j+1} y_{j+1})\| \,. \tag{4.50}$$

Die Wirkung auf den Vektor z ist durch

$$\begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} z_j \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\phi z_j \\ \sin\phi z_j \end{pmatrix} \tag{4.51}$$

gegeben. Das neue Residuum kann nun gemäß 4.31 als $\sin\phi z_j$ gewählt werden. Die Givensrotation ändert nur dei j-te und die (j+1)-te Zeile, sodass keine vollständige Matrixmultiplikation ausgeführt werden muss.
Falls $H \in \mathbb{C}^{j+1\times j}$ ist, muss die Rotation angepasst werden. Die Nebendiagonale der Hessenbergmatrix ist immer reell, da diese im Arnoldiprozess als

$$h_{(j+1)j} = \|\omega(j)\|, \quad \omega(j) \in \mathbb{C}^n \tag{4.52}$$

bestimmt wurde. Es kann die unitäre Matrix

$$F_j = \begin{pmatrix} c & \overline{-s} \\ s & c \end{pmatrix} \tag{4.53}$$

verwendet werden. Wählt man nun analog

$$c = \frac{h_{jj}}{\rho} \in \mathbb{C}, \tag{4.54}$$

$$s = \frac{-h_{(j+1)j}}{\rho} \in \mathbb{R}, \tag{4.55}$$

dann ist

$$F_j = \begin{pmatrix} c & \overline{-s} \\ s & c \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = G_j. \tag{4.56}$$

Es ist

$$\begin{pmatrix} \frac{h_{jj}}{\rho} & -\frac{-h_{(j+1)j}}{\rho} \\ \frac{-h_{(j+1)j}}{\rho} & \frac{h_{jj}}{\rho} \end{pmatrix} \begin{pmatrix} h_{jj} \\ h_{(j+1)j} \end{pmatrix} = \begin{pmatrix} \frac{h_{jj}^2+h_{(j+1)j}^2}{\rho} \\ -\frac{h_{jj}h_{(j+1)j}}{\rho} + \frac{h_{jj}h_{(j+1)j}}{\rho} \end{pmatrix} = \begin{pmatrix} \rho \\ 0 \end{pmatrix}. \tag{4.57}$$

In der Methode br_ statstate.gmres_ ext_ ev werden Givnesrotationen verwendet.

```
        ## Use the old Given rotations

        for i in range(k):
            #important use temps to stor the values
            temp1=cos[i]*Hess[i,k]+ -sin[i]*Hess[i+1,k]
            temp2=sin[i]*Hess[i,k]+cos[i]*Hess[i+1,k]
            Hess[i,k]=temp1
            Hess[i+1,k]=temp2

        #calc new Givensrotation  with tau for stabilisation
        tau=np.abs(Hess[k,k])+np.abs(Hess[k+1,k])
        nu=tau*np.sqrt(((Hess[k,k]/tau)**2)+((Hess[k+1,k]/tau)**2))

        cos.append(Hess[k,k]/nu)
        sin.append(-Hess[k+1,k]/nu)

        Hess[k,k]=nu
        Hess[k+1,k]=0
        #print(np.real(Hess))

        #calc approx
        z.append(sin[k]*z[k])
        z[k]=cos[k]*z[k]

        ##the new relative residual
        eps.append(np.abs(z[k+1])/beta)
```

Die Givensrotation die im letzten Schritt berechnet wurde muss zuerst auf die entsprechenden Zeilen angewendet werden, dann kann die neue Givensrotation berechnet werden.

4.8. Inverse Potenzenmethode mit Shift

Die inverse Potenzenmethode ist in br_ statstate in der Methode inv_ PowIteration implementiert. Sie kann in wenigen Zeilen Code implementiert werden. Als Eingabe verlangt die Methode

- den Systemhamiltonoperator als Instanz der Klasse qutip.qobj,
- eine Liste von Systemoperatoren ebenfalls aus qutip.qobj,
- eine Funktion, welche die Bloch-Redfield Mastergleichung löst,
- einen Startoperator aus qutip.qobj als Approximation an die Lösung,
- die Anzahl der Iterationen nach dem die Methode neu gestartet wird,
- einen Shift für die Auswertung,

- einen Parameter der die maximale Anzahl von Neustarts begrenzt,
- den „tidyup “ Wert für br_ statstate.evaluation,
- einen Parameter, der die maximalen Schritte der inversen Iteration begrenzt,
- eine Toleranz bis zur welchen der Eigenwert approximiert werden soll.

Die Methode gibt den Eigenvektor in Form einer Python Liste zurück. Die Umformung zu einer Matrix erfolgt in br_ statstate.statState. Die Funktion ruft in jedem Schritt das GMRES Verfahren auf um das Inverse Problem zu lösen. Da die Potenzenmethode gegen einen Eigenvektor konvergiert, ist es sinnvoll, die aktuelle Näherung x als Startvektor für die GMRES Methode zu übergeben (Z.9).

```
1       ray=[]
2       ray.append(1)
3       k=0
4       x=x0
5
6       while(ray[k]>tol_pow and k<maxiter_pow):
7
8             #Calculate the Inverse
9           x_neu=gmres_ext_ev(H,S,spectrum,auswertung,x,x,
10          restart,tol_gmres,shift,maxiter_gmres,tidyup)
11
12            ## normalize The vector
13          norm=np.linalg.norm(x_neu)
14          x=x_neu/norm
15
16          #rayleigh#
17          xtilde=auswertung(H,S,spectrum,x,tidyup)
18
19                  #print("norm: " +str(np.linalg.norm(xtilde-shift*x)))
20          ray.append(np.abs(np.vdot(x,xtilde)))
21
22          #print(np.abs(np.vdot(x,xtilde)))
23
24          #reshape the vector to a matrix
25          l=len(x)
26          s=np.sqrt(l)
27          xs=np.reshape(x,(s,s),order='F')
28          #print(Qobj(np.abs(xs)).tidyup())
29
30          # make the matrix hermitian (density Operator)
31          x=0.5*(np.conj(np.transpose(xs))+xs)
32          x=np.reshape(x,(l))
33
34          # normalize the vector
35          norm=np.linalg.norm(x)
36          x=x/norm
37
38          k=k+1
39
40      return x
```

In Jedem Schritt muss der Vektor normiert werden (Z.13+14). Der Eigenwert kann mit dem Rayleighkoeffizienten gemäß 4.11 berechnet werden (Z.17 und Z.20). Da die Lösung hermitesch 2.4 sein muss, wird in jedem Schritt der nicht hermitesche Anteil der Lösung verworfen indem

$$\begin{aligned}\rho &= \frac{1}{2}\left(\rho + \rho^\dagger + \rho - \rho^\dagger\right) \\ &= \frac{1}{2}\left(\rho + \rho^\dagger\right) + \frac{1}{2}\left(\rho - \rho^\dagger\right) = r + r'\end{aligned} \tag{4.58}$$

in eine hermitesche Matrix r und eine nicht hermitesche Matrix r' umgeformt wird. Der nicht hermitesche Anteil r' wird verworfen und anschließend wird die Approximation nochmal normiert (Z.25-36).
In Abbildung 7 haben wir bereits gesehen, dass die Konvergenz der Methode sehr schnell sein kann. In Abbildung 9 ist der Rayleighkoeffizient für einige Schritte am Beispiel aus

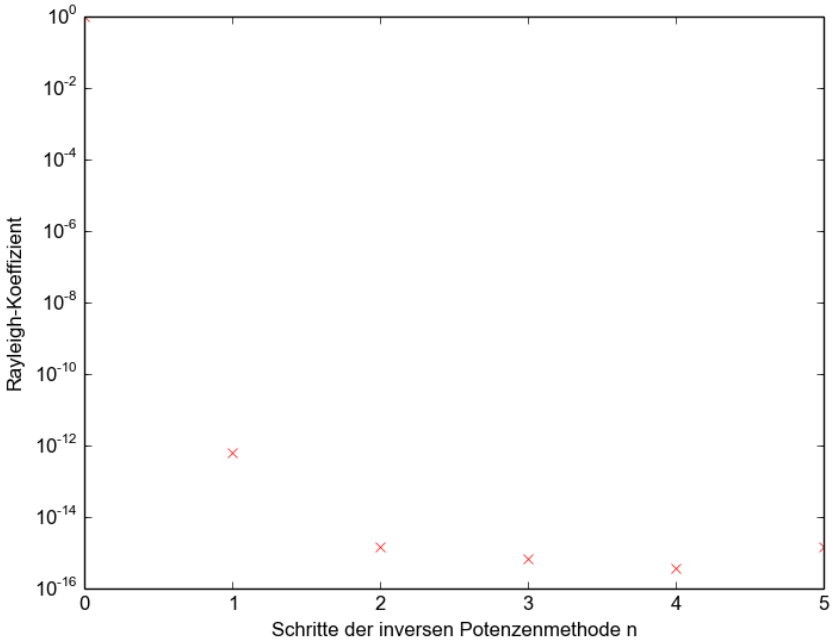

Abbildung 9: Rayleighkoeffizient nach n Schritten der inversen Potenzenmethode

4.6.1 geplottet. Es wurde

$$H_S \in \mathbb{C}^{20\times 20} \tag{4.59}$$

gewählt. Die Toleranz für das GMRES verfahren lag bei $1.0e-12$, kann aber auch größer gewählt werden um Rechenzeit zu sparen.

4.9. Methode br_ statstate.statState

Die Methode br_ statstate.statState ruft die Methoden inv_ PowIteration und gibt das Ergebnis als Instanz von qutip.qobj zurück. Als Eingabe verlangt die Methode

- den Systemhamiltonoperator als Instanz der Klasse qutip.qobj,
- eine Liste von Systemoperatoren ebenfalls aus qutip.qobj,
- eine Funktion, welche die Bloch-Redfield Mastergleichung löst,

- einen Startoperator aus qutip.qobj als Approximation an die Lösung,
- die Anzahl der Iterationen nach dem die Methode neu gestartet wird,
- einen Shift für die Auswertung,
- einen Parameter der die maximale Anzahl von Neustarts begrenzt,
- optional den „tidyup “ Wert für br_ statstate.evaluation (standarmäßig True),
- einen Parameter, der die maximalen Schritte der inversen Iteration begrenzt,
- eine Toleranz bis zur welchen der Eigenwert approximiert werden soll.

Die Methode gibt den stationären Zustand zum gegebenen System als Instanz von qutip.qobj zurück. Der gewählte Startvektor wird zunächst normiert und hermitesch gemacht 4.8.
Das Ergebnis, welches von der inversen Potenzenmethode berechnet wurde wird in eine Matrix umgeformt und zurück gegeben.

```
1  def statState(H,S,ohmic_spectrum,x0,shift,restart,
2                tol_gmres,maxiter_gmres,tol_pow,maxiter_pow,tidyup=True):
3
4      #make start vector hermitian if needed
5      l=len(x0)
6      s=np.sqrt(l)
7      xs=np.reshape(x0,(s,s),order='F')
8      x0=0.5*(np.conj(np.transpose(xs))+xs)
9      x0=np.reshape(x0,(l))
10
11     norm=np.linalg.norm(x0)
12     x0=x0/norm
13
14     x=inv_PowIteration(H,S,ohmic_spectrum,auswertung,x0,shift,restart,
15                        tol_gmres,maxiter_gmres,tol_pow,maxiter_pow,tidyup)
16
17     l=len(x)
18     s=np.sqrt(l)
19     x=np.reshape(x,(s,s),order='F')
20
21     if tidyup==True:
22         x=Qobj(x).tidyup()
23
24     return x
```

4.10. Laufzeit der Implementation und Vergleich mit QuTiP

Mit QuTiP ist es möglich die stationären Zustände zu bestimmen. Hierzu muss zunächt der Bloch-Redfield Tensor bestimmt werden. Mit dem Tensor als Instanz von qutip.qobj können dann alle Eigenvektoren berechnet werden, indem die Funktion qutip.qobj.eignestates() aufgerufen wird. Ein minimales Porgramm kann wie folgt aussehen.

```
from quitp import *
## H,S,ohmic_spectrum Sinvoll initialisiert
ekets,R = bloch_redfield_tensor(H,[S],[ohmic_specturm])
evalsR,eketsR = R.eigenstates()
```

Der Großteil des Aufwands besteht in der Aufstellung des Tensors. Die Laufzeiten beider Programme sind anhand vom Beispiel 5.1 für niedrige und hohe Temperaturen in den Abbildungen 10 und Abbildung 11 geplottet worden, sodass für die niedrigen Temperaturen die Lösung diagonal ist und für die höheren Temperaturen nicht. Die Toleranz für das GMRES Verfahren wurde als $(1.0e-6)$ gewählt, da es nicht notwendig ist das die Inverse genau berechnet wird 4.7.1. Die Toleranz für die inverse Potenzementhode wird mit $(1.0e-16)$ sehr klein gewählt, damit die Approximation entsprechend genau wird. Es wurden keine Neustarts verwendet, da diese erst notwendig werden, wenn die Speichergrenze erreicht ist. Als Verschiebung hat sich der Wert $(1.0e-5)$ als vorteilhaft herausgestellt. Es wurden nur 5 Schritte innerhalb der inversen Potenzenmethode erlaubt, was ausreichend ist um eine Akzeptable Genauigkeit zu erreichen. Dies ist in Abbildung 13 geplottet. In Abbildung 14 ist die Laufzeit für 150 mögliche Schritte und einer Toleranz von $(1.0e-12)$ für die inverse Potenzenmethode geplottet (Es waren jeweils zwischen 2 und 3 Schritten notwendig um die Toleranz in diesem Fall zu erreichen, der Ausreißer kam Zustände, da die Methode hier nur 2 für die umliegende Messungen allerdings 3 Schritte benötigte).

Die Approximation des stationären Zustands scheint besser zu skalieren als die Im-

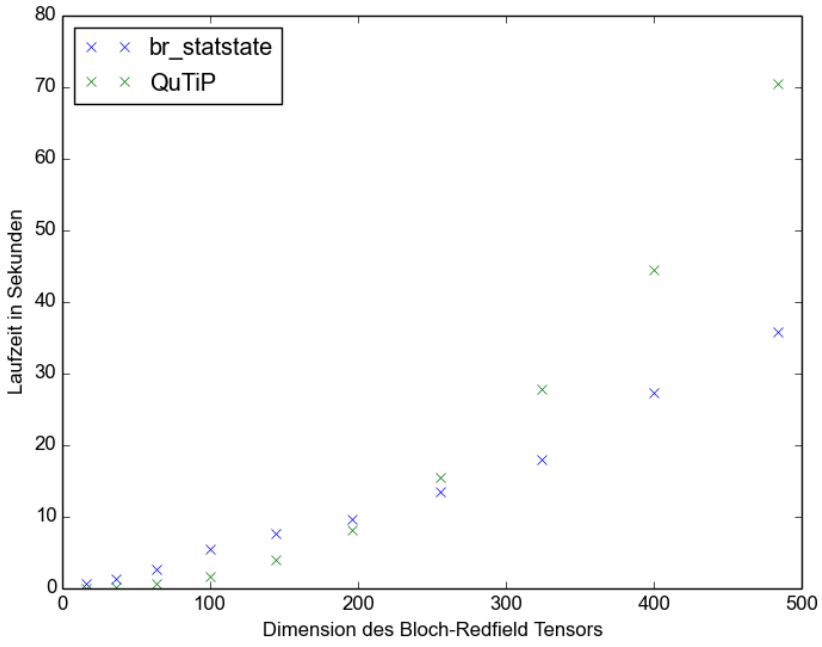

Abbildung 10: Laufzeitvergleich in Abhängigkeit der Dimension des Bloch-Redfield Tensors für hohe Temperaturen

plementation in Qutip. Da QuTiP den naiven Algorithmus zur Matrixmultiplikation verwendet skaliert die Methode wie n^3. Die Implementation in br_ statstate scheint in etwa linear zu skalieren, was zu der Skalierung in Abbildung 8 passt. Durch die indirekte Bestimmung des stationären Zustands ohne den Tensor an sich aufzustellen ist ein

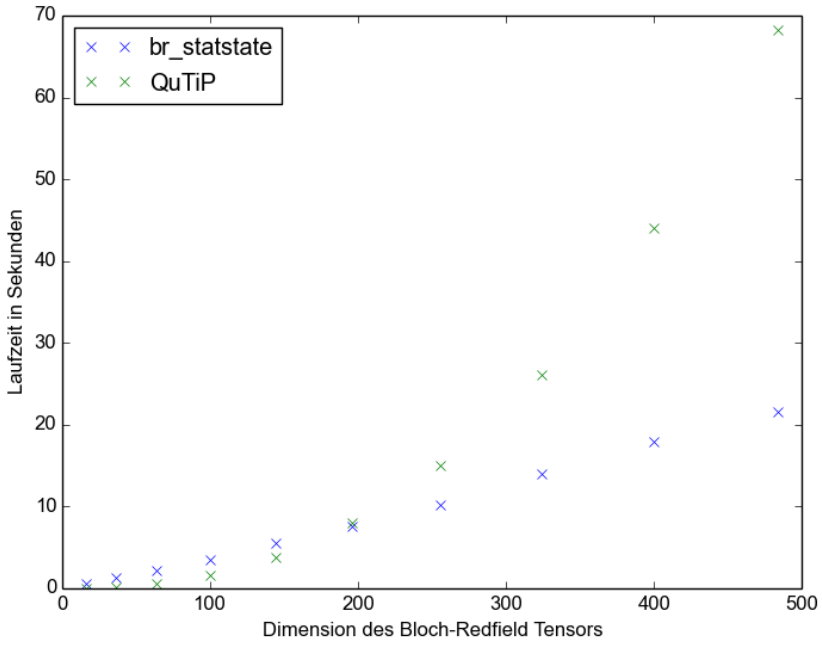

Abbildung 11: Laufzeitvergleich in Abhängigkeit der Dimension des Bloch-Redfield Tensors für niedrige Temperaturen

deutlicher Laufzeitgewinn zu verzeichnen.
Der lineare Zusammenhang bestätigt sich nicht, sobald größere Tensoren betrachtet werden. In Abbildung 12 scheint dieser noch gegeben zu sein. Wird für größere Tensoren nur die Laufzeit von br_ statstate geplottet erkennt man, dass es eine quadratische Komponente mit kleiner Konstante geben muss. Der Fit aus Abbildung 12 ergibt Laufzeiten von

$$t_{QuTiP} = (1.7e-7)n^3 + 0.00027n^2 - 0.022n \tag{4.60}$$

$$t_{brstat} = 0.052n \tag{4.61}$$

4.11. Wahl eines geeigneten Startvektors

In 2.6.2 konnte gezeigt werden, dass in säkularer Näherung der stationäre Zustand existiert und dass dann nur die Diagonalen besetzt sind. In Beispiel 5.1 ist ersichtlich, dass dies ohne säkulare Näherung nicht immer der Fall ist. Dennoch scheint es sinnvoll, den Startzustand diagonaldominant zu wählen. Tatsächlich zeigt sich in dem betrachteten Beispiel hierdurch eine Konvergenzbeschleunigung.
Aus den betrachteten Beispielen scheint es, dass für tiefe Temperaturen die säkulare Näherung direkt gegeben ist.
Der Startzustand sollte darüber hinaus hermitesch gewählt werden, da das Ergebnis ein

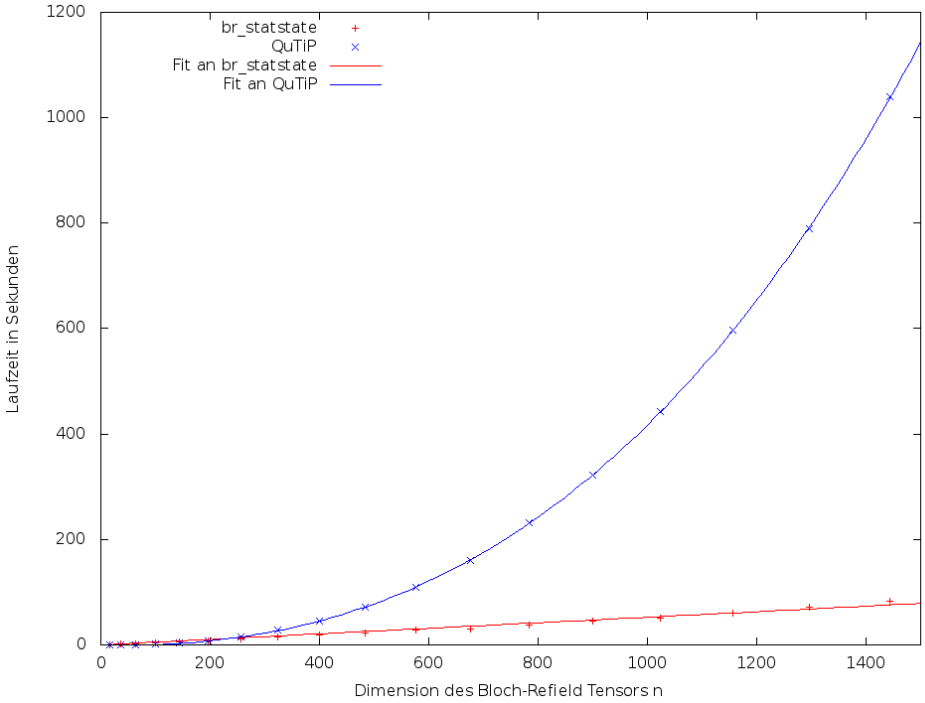

Abbildung 12: Laufzeitvergleich in Abhängigkeit der Dimension des Bloch-Redfield Tensors (niedrige Temperatur)

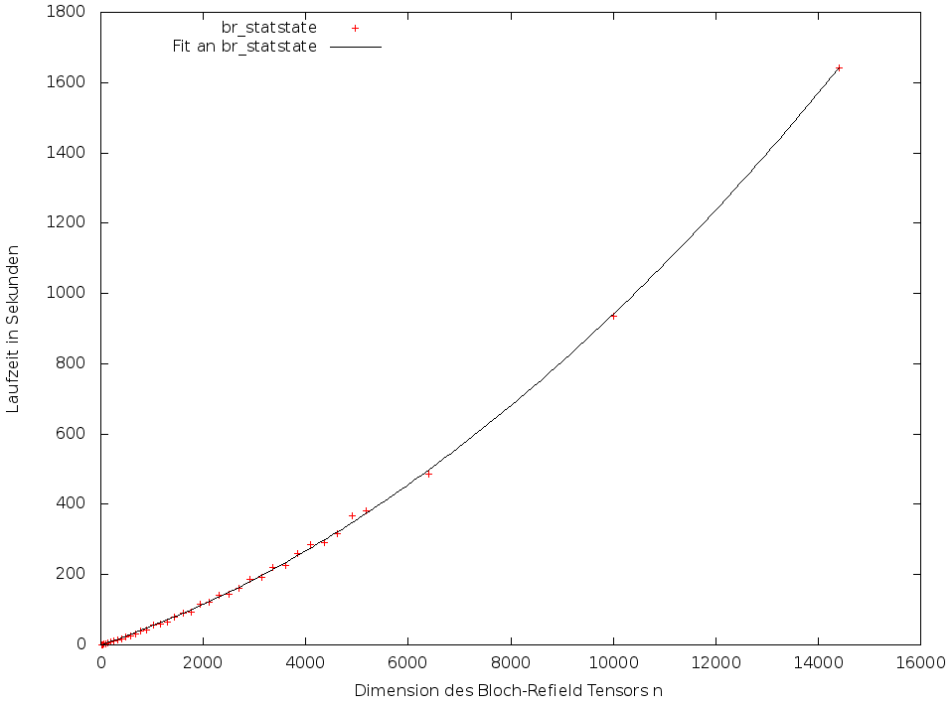

Abbildung 13: Laufzeit in Abhängigkeit der Dimension des Bloch-Redfield Tensors bei fester Anzahl von Schritten der inversen Potenzenmethode (5 Schritte)

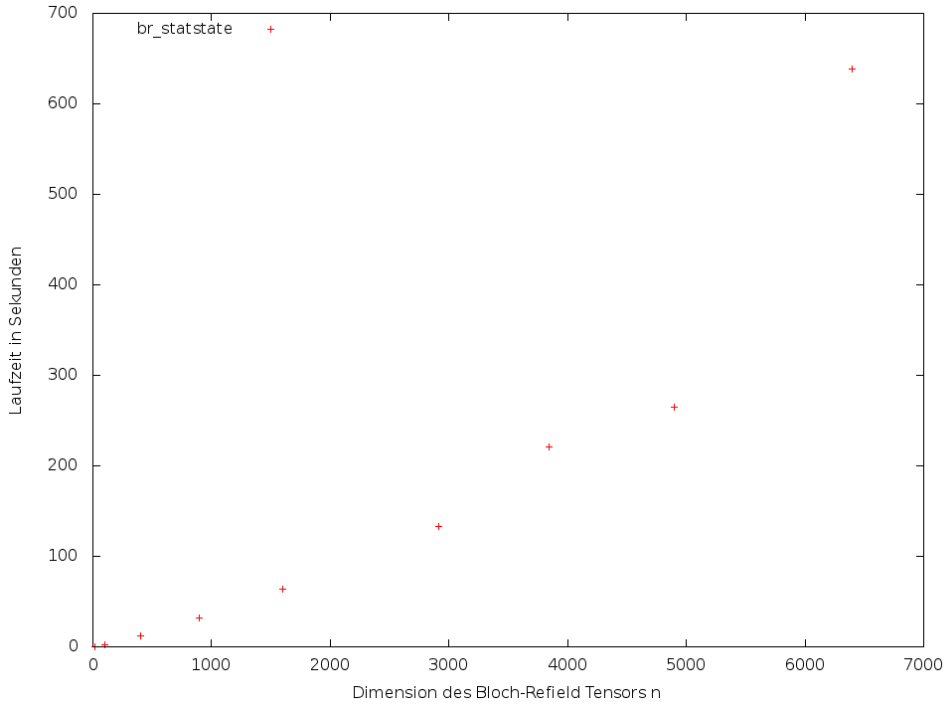

Abbildung 14: Laufzeit in Abhängigkeit der Dimension des Bloch-Redfield Tensors bei variabler Anzahl von Schritten der inversen Potenzenmethode

Dichteoperator ist und dieser daher hermitesch ist.

5. Beispiele anhand von physikalischen Systemen

5.1. Jaynes-Cummings Modell

Beim Jaynes-Cummings Modell wird ein Zweizustandsystem betrachtet, das an einen Resonator koppelt. Der Hamiltonoperator des Systems hat die Form

$$H_{JC} = \frac{1}{2}\Delta E(\sigma_z \otimes N) + g\left(\sigma_x \otimes (a + a^\dagger)\right) + \omega\left(\mathbb{1} \otimes (a^\dagger + a)\right). \tag{5.1}$$

Hierbei bezeichnet ΔE die Energiedifferenz zwischen den beiden Zuständen des Zweizustandssystems, g die Kopplung zwischen Resonator und den zwei Zuständen und ω die Energie des Resonators. σ_z und σ_x sind die entsprechenden Paulimatrizen und a ist der bosonische Vernichtungsoperator. Das Modell beschreibt beispielsweise ein Atom mit einem Grundzustand und einem angeregten Zustand in einem Laser.
Dieses System soll nun an mehrere andere Resonatoren koppeln. Die Kopplung wird durch die spektrale Dichte

$$s(\omega) = \eta \frac{\omega}{1 - e^{-\beta\omega}} \tag{5.2}$$

beschrieben. ω bezeichnet die Energiedifferenz gemäß 2.5, β ist der Kehrwert der thermischen Energie,

$$\beta = \frac{1}{k_B T}. \tag{5.3}$$

Der Vorfaktor η wird vereinfachend als eins angenommen. Als Systemoperator wird der Teilchenzahloperator N verwendet.

Dimension: Für $N \in \mathbb{R}^{n\times n}$ ist $H_S \in \mathbb{C}^{2n\times 2n}$. Der Bloch-Redfield Tensor ist dann $L \in \mathbb{C}^{(2n)^2\times(2n)^2}$.
Es ist nun möglich den stationären Zustand des Systems für eine gegeben Temperatur T und eine gegebene Anzahl von Freiheitsgraden n zu berechnen. Die Wahrscheinlichkeit, das sich das System in einem bestimmten Zustand befindet kann durch dein Diagonalelement des stationären Zustands ausgedrückt werden. Die anderen Elemente beschreiben Übergangswahrscheinlichkeiten zwischen diesen Zuständen.
In Abbildung 15 sind die Diagonalelemente entsprechend der Energie abgebildet. Es wurde $\beta = 0.1$, $\Delta E = 0.6\pi$, $\omega = 2\pi$, $g = 0.5$ und die Anzahl der Freiheitsgrade $2n = 20$ und $2n = 60$ gewählt.
Für höhere Temperaturen werden Zustände mit höherer Energie wahrscheinlicher. Mit den gleichen Werten ergibt sich für $\beta = 0.01$ Abbildung 16. Für $\beta = 0.01$ existieren Elemente im stationären Zustand, die nicht auf der Diagonalen sind und ungleich null sind. Da in Abbildung 16 die Wahrscheinlichkeit der Zustände mit höherer Energie noch signifikant sind, scheint es sinnvoll, es mit mehr Zuständen zu versuchen. Für $2n = 100$ ergibt sich die Abbildung 17. Es ist deutlich zu erkennen wie die Temperatur die Wahr-

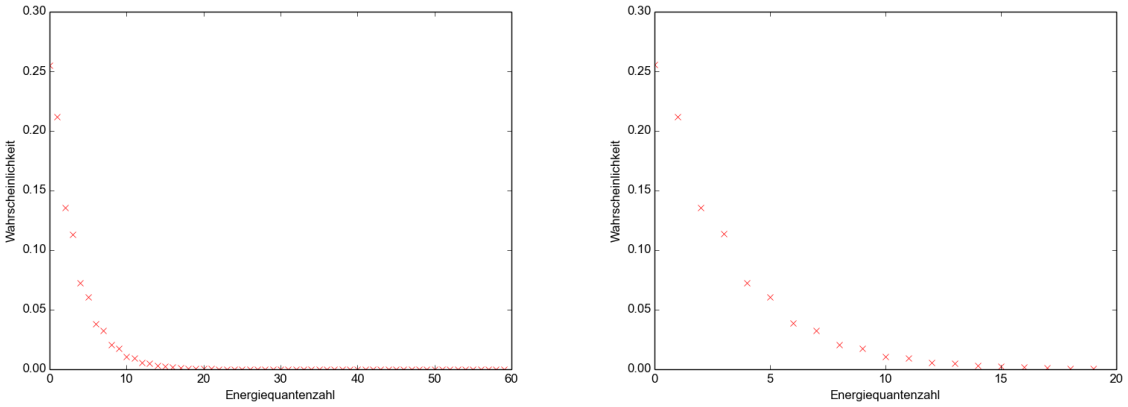

Abbildung 15: Wahrscheinlichkeiten über Energie für Zustände im Jaynes-Cummings Modell

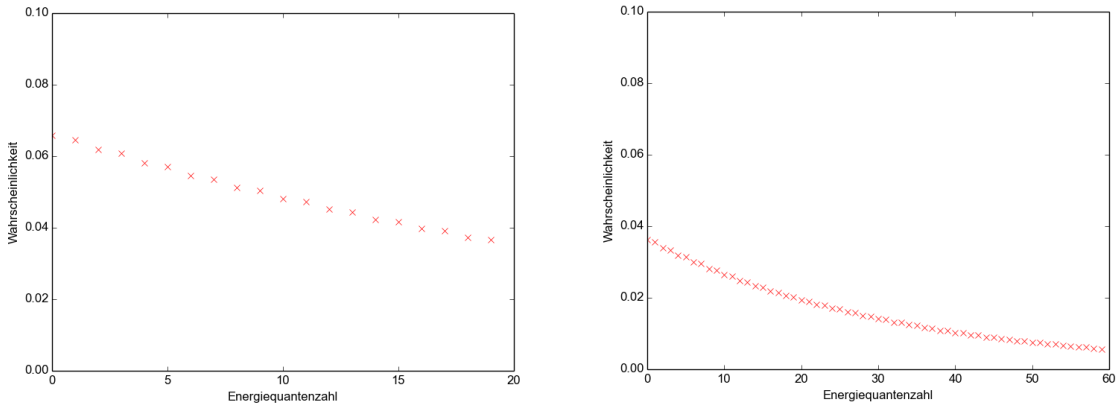

Abbildung 16: Wahrscheinlichkeiten über Energie für Zustände im Jaynes-Cummings Modell

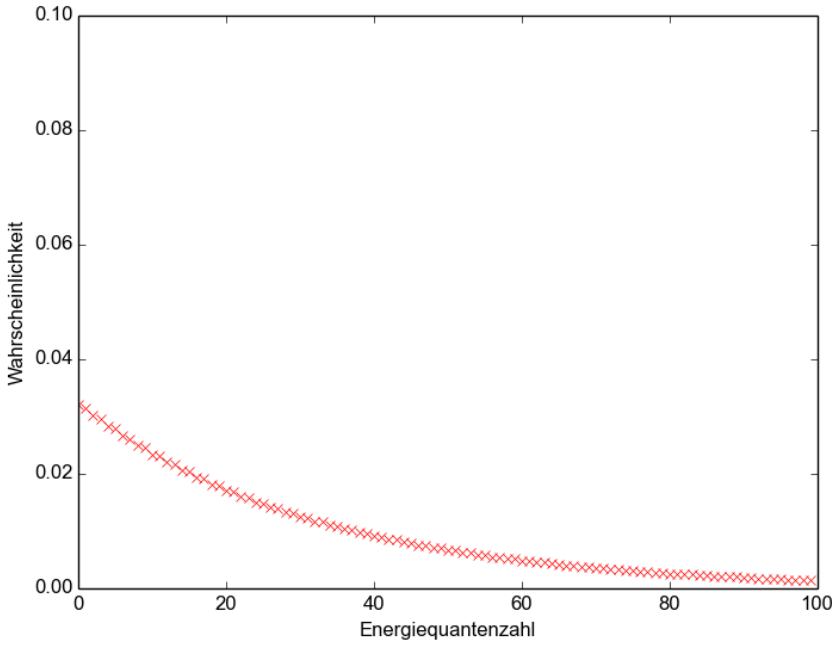

Abbildung 17: Wahrscheinlichkeiten über Energie für Zustände im Jaynes-Cummings Modell

scheinlichkeit beeinflusst. Es sind nun mehr Zustände möglich und die Differenz zwischen der Wahrscheinlichkeit zweier Zustände mit ähnlicher Energie hat abgenommen.

5.2. Gekoppelte Spins

Zwei gekoppelte Spins können mit dem Hamiltonoperator

$$H = \sigma_z \otimes \mathbb{1} + \mathbb{1} \otimes \sigma_z + g(\sigma_x \otimes \sigma_x) \tag{5.4}$$

beschrieben werden, wobei σ_z und σ_x die entsprechenden Paulimatrizen sind und g die Kopplungsstärke ist. Als Systemoperator kann $A = \sigma_z \otimes \mathbb{1}$ gewählt werden. Als spektrale Dichte kann die Funktion wie im Beispiel 5.2 gewählt werden. Die Kopplungskonstante soll klein sein.
In diesem Fall ist der stationäre Zustand entartet. Daher kann die inverse Potenzenmethode nicht gegen einen festen Eigenvektoren konvergieren. Der Rayleigkoeffizient konvergiert dennoch gegeben den Eigenwert null. Der Vektor, welcher als Ergebnis ermittelt wird, ist eine Linearkombination der stationären Zustände. Ist

$$\{|i\rangle : |i\rangle \in \text{Kern}(L), \langle i|j\rangle = 0 \text{ für } i \neq j\} \tag{5.5}$$

die Menge der stationären Zustände von L, dann konvergiert die inverse Potenzenmethode gegen

$$|\Psi\rangle = \sum_{i=1}^{k} c_i |i\rangle \text{ mit } c_i \in \mathbb{C}, \tag{5.6}$$

wobei die c_i unbekannt sind und vom gewählten Startvektor abhängen. Für verschiedene zufällige Anfangszustände für kleine Temperaturen $\beta = 0.1$ und eine kleine Kopplung $g = 0.05$ werden wieder die diagonalen des Ergebnisses geplottet werden. Es ergeben sich je nach c_i aus 5.6 verschiedene Grafiken, zwei Beispiele sind in Abbildung 18 und Abbildung 19 angegeben.
Um die Entartung aufzuheben kann der Systemoperator angepasst werden. Indem dieser

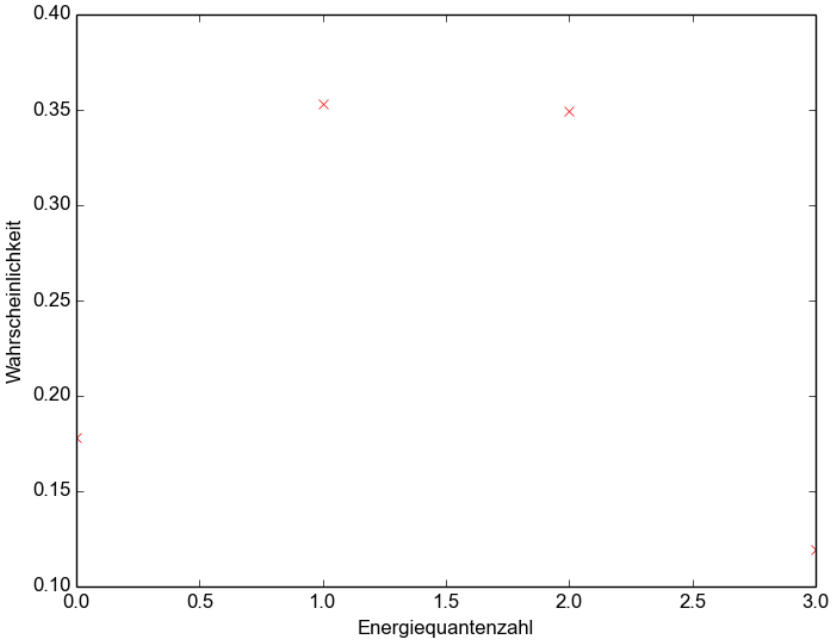

Abbildung 18: Wahrscheinlichkeiten über Energie für die Zustände

als

$$A = (\alpha\sigma_x + \beta\sigma_z) \otimes \mathbb{1},\ \alpha, \beta \in \mathbb{R} \tag{5.7}$$

gewählt werden. Es ergibt sich ein fester stationärer Zustand. Mit denselben Parameter wie im entarteten Fall ergibt sich Abbildung 20.

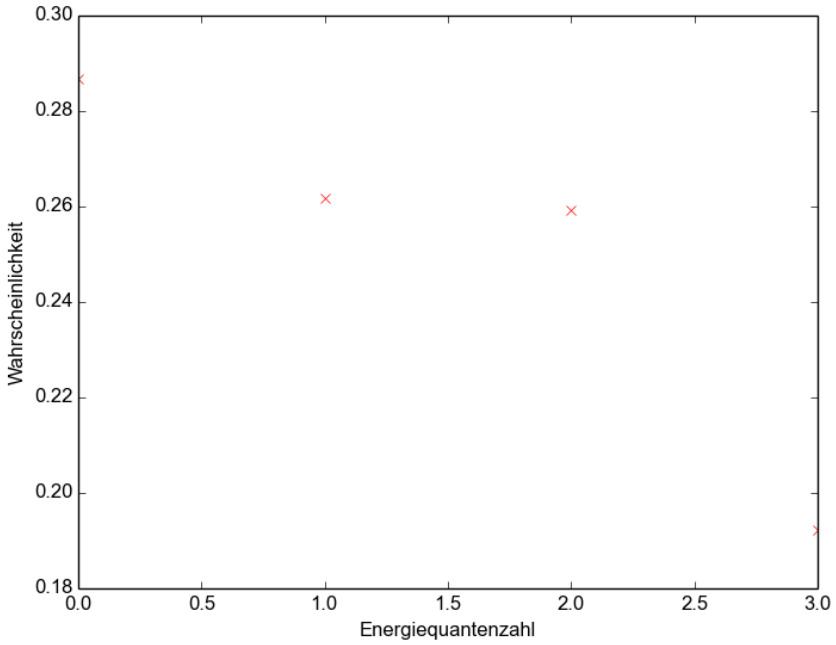

Abbildung 19: Wahrscheinlichkeiten über Energie für die Zustände

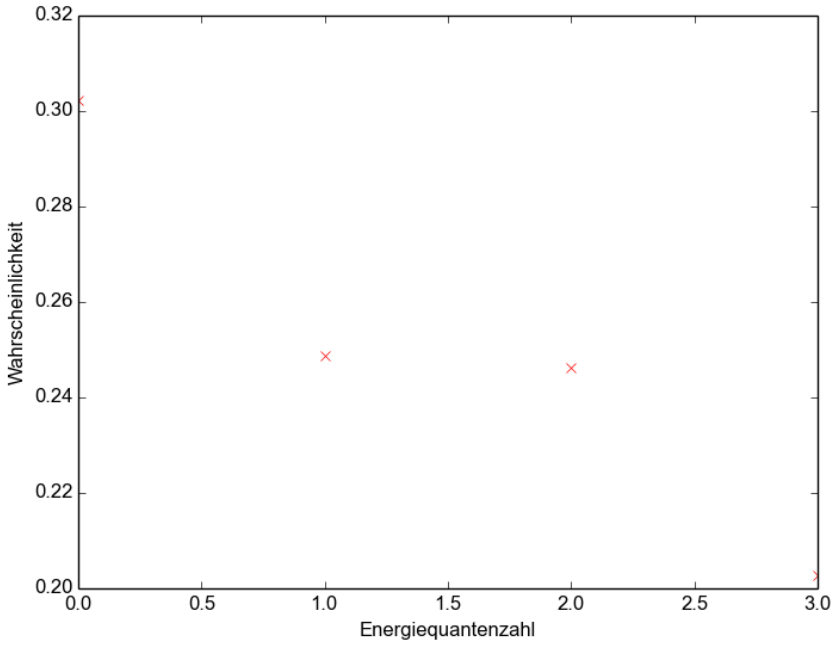

Abbildung 20: Wahrscheinlichkeiten über Energie für die Zustände

6. Ergebnisse

Das Ergebnis dieser Arbeit ist der Quellcode der Methode br_ statstate vgl. A.1 zur Berechnung des stationären Zustands der Bloch-Redfield Mastergleichung 2.15. Die Ziele welche in der Einleitung formuliert sind, konnten weitestgehend erfüllt werden.

Berechnung des stationären Zustands Im Wesentlichen wird zur Berechnung des stationären Zustands die inverse Potenzenmethode mit Shift verwendet. Um die Inverse in jedem Schritt der inversen Potenzenmethode zu bestimmen kommt ein Krylowraumverfahren (GMRES) zum Einsatz, welches eine Speicherbegrenzung ermöglicht.
Im Gegensatz zum bestehenden Programm in QuTiP ist es mit der Methode br_ statstate möglich den stationären Zustand des Systems direkt zu berechnen ohne den Bloch-Redfield Tensor aufstellen zu müssen. Es ist lediglich notwendig einen geeigneten Hamiltonoperator, Systemoperatoren und entsprechende spektrale Dichtefunktionen vorzugeben.
Außerdem ist es möglich die Parameter der notwendigen numerischen Verfahren, welche wesentlichen Einfluss auf die Laufzeit und die Genauigkeit des Ergebnisses haben, frei zu wählen.
Problematisch stellt sich die Berechnung des stationären Zustands heraus, falls dieser nicht eindeutig ist, also wenn Entartung vorliegt. Die Konvergenz des Programms ist auch hier gewährleistet. Das Ergebnis liegt dann allerdings im entsprechenden Eigenraum und ist nicht eindeutig. Für verschiedene Startwerte ergeben sich unterschiedliche Ergebnisse vgl.5.2.

Laufzeitverbesserung Mit dem Programm ist eine Laufzeitverbesserung zu verzeichnen vgl. Abbildung 12. Dies liegt auch an der Tatsache, dass es nicht notwendig ist den Bloch-Redfield Tensor zu bestimmen. Die Laufzeit und die Skalierung des Programms wird im Kapitel 5.2 mit QuTiP verglichen.
Da die Skalierung vom gegebenen physikalischen System abhängt kann keine explizite Laufzeit angegeben werden. Die Laufzeitverbesserung konnte dadurch erreicht werden, dass ausgenutzt wurde, dass für manche Systeme die auftretenden Operatoren dünnbesetzt sind. QuTiP nutzt diesen Umstand nicht aus und verwendet bei der Bestimmung des Bloch-Redfield Tensors den naiven Algorithmus zur Multiplikation von Matrizen. Indem die notwendigen Matrixmultiplikationen durch geeignetere Verfahren ersetzt wurden konnte eine Verbesserung erzielt werden.
Die Besetzungsstruktur der Operatoren hängt von der Basis ab in welcher diese betrachtet werden vgl.4.6.1. Indem die Multiplikationen in der geeigneten Basis ausgeführt wird kann die Besetzungseigenschaft optimal ausgenutzt werden.
Für große Systeme ist die Laufzeitverbesserung zum Teil erheblich vgl. Abbildung 12. Für sehr große Systeme war es mit QuTiP in keiner praktikablen weiße möglich den stationären Zustand zu bestimmen, da die Bestimmung des Bloch-Redfield Tensors sehr viel Zeit in Anspruch nimmt.

Startvektor Zur Berechnung des stationären Zustands wird die inverse Potenzenmethode verwendet. Es zeigt sich, dass die Laufzeit verbessert werden kann indem ein geeigneter Startvektor gewählt wird. Im Kapitel 2.6 wird näher auf die Eigenschaften des stationären Zustands in säkularer Näherung eingegangen. Es zeigt sich, dass dieser nur auf der Diagonalen besetzt ist. Auch wenn die säkulare Näherung nicht verwendet wird scheint es sinnvoll einen Startzustand nahe an einem diagonalen Zustand zu wählen vgl. 4.11.

Speicherbegrenzung Mit der Methode br_ statstate ist eine Speicherbegrenzung möglich. In dem bestehenden Programm von QuTiP wurde der meiste Speicher für den Bloch-Redfield Tensor benötigt. Für große System kann dieser sehr groß werden vgl. 4.1.1. Im Gegensatz zu den anderen Matrizen die auftreten ist der Bloch-Redfield Tensor im Allgemeinen nicht dünnbesetzt.
Hinzu kommt, dass zur Berechnung des Bloch-Redfield Tensors alle notwendigen Matrizen als vollbesetzte Matrizen gespeichert werden mussten. In br_ statstate ist dies nicht mehr notwendig.
Da der Bloch-Refield Tensor in br_ statstate nicht explizit berechnet wird, muss dieser auch nicht gespeichert werden. Der wesentliche Speicheraufwand liegt nun im GMRES Verfahren. Für ein Systemhamiltonoperator $H \in \mathbb{C}^{n\times n}$ ist der Bloch-Redfield Tensor $L \in \mathbb{C}^{n^2\times n^2}$ vgl. 4.1.1. Die Vektoren des Krylowraums sind daher vollbesetzte Vektoren in $\mathbb{C}^{n^2}$. Wird das GMRES Verfahren für den gesamten Krylowraum angewendet, müssen n^2 dieser Vektoren gespeichert werden, was dem gleichen Speicheraufwand entspricht, welcher für L notwendig ist.
Es ist möglich nur eine begrenzte Anzahl dieser Vektoren zu bestimmen und damit den Speicheraufwand zu reduzieren. Hierdurch wird die Konvergenzgeschwindigkeit allerdings zum Teil erheblich verlangsamt vgl. Abbildung 6. Die Laufzeitverbesserung durch br_ statstate wird in diesem Fall relativiert. Es ist möglich, dass das Verfahren nicht konvergiert. In 4.7.1 wird untersucht wie sich die Konvergenzgeschwindigkeit ändert, wenn nicht alle Vektoren bestimmt werden. Solange es nicht notwendig ist, sollten daher alle Vektoren berechnet werden.
Für sehr große Systeme ist es mit hinreichender Rechenzeit möglich den stationären Zustand zu bestimmen. Da QuTiP keine Speicherbegrenzung bietet ist dies hier nicht möglich.

Ausblick Die Laufzeit zur Auswertung der Bloch-Redfield Gleichung konnte erheblich optimiert werden. Die Funktion br_ statstate.evaluation ermöglicht diese Auswertung. Die Laufzeit dieser Funktion ist entscheidend, da für das Krylovraumverfahren viele Auswertungen notwendig sind. Es ist denkbar die Funktion weiter zu optimieren indem die Berechnung der Matrix G aus 2.50 in ein C-Modul ausgelagert wird. Elementare Berechnungen wie diese sind in Python nur sinnvoll, falls diese mit NumPy oder SciPy (welche selbst auf C-Module zurückgreifen) ausgeführt werden können, was hier nicht möglich war. Der C-Compiler optimiert den Code für die entsprechende Architektur, wodurch elementare Rechnungen schneller durchgeführt werden können als in einer rei-

nen Skriptsprache wie Python [Wik15a]. Ein Test am Beispiel 5.1 zeigt den Unterschied der durch dieses C-Modul erreicht werden kann vgl. Abbildung 21. Das entsprechende

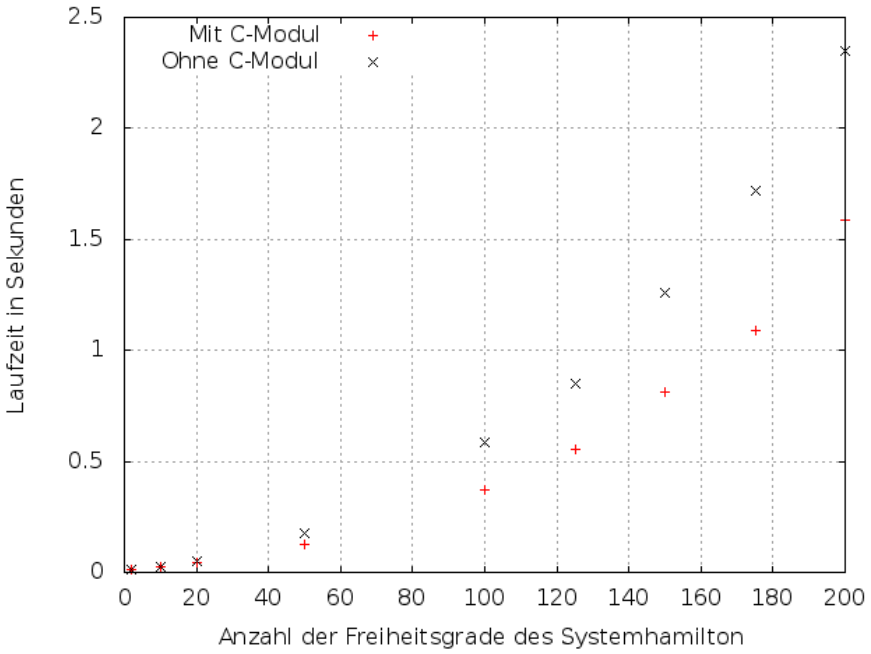

Abbildung 21: Vergleich der Laufzeit zur Auswertung der Bloch-Redfield Mastergleichung mit einem C-Modul

C-Modul ist im Anhang beigefügt. Als Vorlage wurde der Quellcode von Valentin Haenel [Hae15] verwendet.
In der Funktion br_ statstate.evaluation wurde hierauf verzichtet, weil es dann keine praktikable Möglichkeit gegeben hätte die spektrale Dichtefunktion in Python als einfache Recall-Funktion zu übergeben.
Es ist möglich die Funktion br_ statstate.evaluation zu verwenden um die Zeitentwicklung einzelner Zustände zu bestimmen. Die Taylorentwicklung der Dichtematrix $\rho(t)$ ist

$$\rho(t) = \sum_{k=0}^{\infty} \frac{\rho^{(n)}(t_0)}{n!}(t-t_0)^n \tag{6.1}$$

$$= \sum_{k=0}^{\infty} \frac{L^n \rho(t_0)}{n!}(t-t_0)^n = e^{L(t-t_0)}\rho(t_0). \tag{6.2}$$

Die Matrixexponentialfunktion kann nun leicht genähert werden in dem genügend Terme der Taylorreihe berechnet werden. Hierzu sind hauptsächlich Auswertungen der Bloch-Redfield Mastergleichung notwendig, die mit der genannten Funktion effektiv ausführbar sind. Es muss nur ein praktikabler Weg gefunden werden den Zeitschritt $t-t_0$ zu wählen ohne einen Overflow zu riskieren.

A. Vollständiger Quellcode

A.1. Quellcode Python

Die Datei br_ statestate.py.

```
"""
Created in 2015 within the bachelor thesis by Marvin Raimund Schulz
at the Institut fuer Theoretische Festkoerperphysik, KIT
"""

from qutip import Qobj
from qutip.superoperator import operator_to_vector
import numpy as np

def evaluation(H,A,Rho,ohmic_spectrum,tidyup):

    '''
    Description:
    evaluation evluates the bloch-redfield tensor for a given
    density operator.

        Takes:
                - A List of The System Operators of class Qobj
                - The Density Operator of Class Qobj
                - The Hamilton Operator of Class Qobj

                - A List of callback functions, defining the ohmic spectrum

        Returns
                - The multiplication of the Density Operator and the
                bloch-redfield Tensor. (Instance of Qobj)

    note: This function does not suport time depending hamiltonoperators.
    This function could be faster with importet C-Files. Compare
    evaluation_fast

    '''
    #Check if Input is instance of Qobj
    if not isinstance(H, Qobj):
        raise TypeError("H must be an instance of Qobj")

    for a in A:
        if not isinstance(a, Qobj) or not a.isherm:
            raise TypeError("Operators in a_ops must be Hermitian Qobj.")

    #Find eigenvektors for transformation
    evals,ekets=H.eigenstates()
    N = len(evals)


    #Transform the operators to H-Eigenbasis for further Calc

    #Rho in Eigenbasis of the Hamilton, but the calculation will be done in the original Basis
    Rho=Rho.transform(ekets,True)



    G = np.zeros((N,N),dtype=complex)
    Erg=np.zeros((N,N),dtype=complex)

    #The complex Part
    Erg=1.0j*(Rho.data*H.data-H.data*Rho.data)

    k=len(A)
    for i in range(k):

        #Evaluation of the ohmic_spectrum for further calc, slow in Python faster with C-code
        for n in range(N):
            for m in range(N):
                G[n, m] = ohmic_spectrum[i](evals[m] - evals[n])*0.5

        Xi=A[i].transform(ekets)
```

```
68          ### elementwise multiplication in Eigen-Basis of H
69          Tplus=Qobj(Xi.data.multiply(G))
70          Tminus=Qobj(Xi.data.multiply(np.transpose(G)))
71
72          ### retrasnformation from eigenbasis of H
73          Tplus=Tplus.transform(ekets,True)
74          Tminus=Tminus.transform(ekets,True)
75
76          if tidyup==True :
77              Tplus=Tplus.tidyup()
78
79          ### multiplication using the sparsity of a
80          Erg=Erg-(A[i].data*Tplus.data*Rho.data)
81          Erg=Erg+(Tplus.data*Rho.data*A[i].data)
82          Erg=Erg-(Rho.data*Tminus.data*A[i].data)
83          Erg=Erg+(A[i].data*Rho.data*Tminus.data)
84
85
86
87          ### create an instance of Cobj and return to eigenbasis of H
88      Erg=Qobj(Erg)
89      Erg=Erg.transform(ekets)
90
91      return Erg
92
93  def inv_PowIteration(H,S,spectrum,auswertung,x0,shift,restart,
94                       tol_gmres,maxiter_gmres,tol_pow,maxiter_pow,tidyup):
95      '''
96      Description:
97      The inverse power iteration calculates an eigenvector for the given operator by
98      calculating its inverse (see gmres function) and using the normel power iteration.
99      The return value is the eigenvector nearest to the given Shift.
100
101     Takes:
102                 - A callback function for evaluating the operator
103                 - A start vector (Instance of Qobj)
104                 - A shift (any number)
105                 - The dimensiom of the used Krylow-Space in gmres
106                 - A tolerance for the internal gmres (float)
107                 - A maximal iteration for the internal gmres (integer)
108                 - A tolerance for the inverse power Iteration (float)
109                 - A maximal iteration for the internal gmres (integer)
110
111     Returns
112                 - A approximation to the eigenvector neares to the given shift (Instance of Qobj)
113                 - A list including the rayleigh coefficient for each step
114
115     note: The function is going to be faster for larger Krylow spaces. The dimension
116     of the krylow space can be reduced to save memory.
117
118     '''
119
120     ray=[]
121     ray.append(1)
122     k=0
123     x=x0
124
125     while(ray[k]>tol_pow and k<maxiter_pow):
126
127           #Calculate the Inverse
128         x_neu=gmres_ext_ev(H,S,spectrum,auswertung,x,x,restart,tol_gmres,shift,maxiter_gmres,tidyup)
129
130           ## normalize The vector
131         norm=np.linalg.norm(x_neu)
132         x=x_neu/norm
133
134         #rayleigh#
135         xtilde=auswertung(H,S,spectrum,x,tidyup)
136
137                 #print("norm: " +str(np.linalg.norm(xtilde-shift*x)))
138         ray.append(np.abs(np.vdot(x,xtilde)))
139         print(ray)
140
141         #print(np.abs(np.vdot(x,xtilde)))
```

```
142
143         #reshape the vector to a matrix
144         l=len(x)
145         s=np.sqrt(l)
146         xs=np.reshape(x,(s,s),order='F')
147         #print(Qobj(np.abs(xs)).tidyup())
148
149         # make the matrix hermitian (density Operator)
150         x=0.5*(np.conj(np.transpose(xs))+xs)
151         x=np.reshape(x,(l))
152
153         # normalize the vector
154         norm=np.linalg.norm(x)
155         x=x/norm
156
157         k=k+1
158
159     return x
160
161
162 def gmres_ext_ev(H,S,spectrum,evaluation,b,x0,restart,tol,shift,maxiter,tidyup):
163
164     '''
165     Description:
166     gmres_ext_ev solves the equation Ax=b for x, instead of passing A,
167     it's necessary to pass a function, that solves Ax=b for b (given x)
168
169     Input:
170     evaluation: A callback function taking a vector returning the
171     Matrix vektor produkt of this vektor
172
173     b: the right hand Side of the equation
174
175     x0: an initial approximation of x
176
177     restart: The maximum iteration of the inner Loop of gmres (dimension of krylow space)
178
179     tol: the toleranz of the relative residual that is to be achieved
180
181     shift: A shift whicht leads to the equation (A-shift)x=b
182
183     maxiter: the maximal iteration of the outer Loop (recursive calls). Note:
184     the maximal recursive calls is also limited by python itself. Therefore
185     it is possible, that a to big maxiter leads to an error
186
187     Output:
188     x: A vector solving the equation evaluation(x)=b
189
190     '''
191
192     ##Lists for storage of krylow vectors and relative residual
193     v=[]
194     omega=[]
195     eps=[]
196     cos=[]
197     sin=[]
198     z=[]
199     ## the iteration variable of the inner iteration
200     k=0
201
202
203     ## Hessenberg matrix
204     if restart<len(x0):
205         dim=restart
206     else:
207         dim=len(x0)
208
209
210     Hess = np.zeros((dim+1,dim),dtype='complex')
211     ## Matrix of Krylow vectors
212     V=np.zeros((dim,len(b)),dtype='complex')
213
214
215     #Calculate the startresiduum
216     r0=b-(evaluation(H,S,spectrum,x0,tidyup)-shift*x0)
```

```
217
218         #print(b)
219         #print(r0)
220         beta=np.linalg.norm(r0)
221
222         #this is the initial krylow Vector (normalized)
223         v.append(r0/beta)
224
225         #the initial relative residual (beta/beta=1)
226         eps.append(1)
227
228         #write the first Krylow in the Matrix
229         V[0]=v[0]
230
231
232         ## After maxiter recursive calls stop calculation
233         if(maxiter==0):
234             print('exceeded Maxiter was not able to achieve the tol')
235             print("the toleranz is: " +str(beta))
236             return x0
237
238
239         ## If the starting vector is allready a good enough approximation
240         #print("beta: " +str(beta))
241         if(np.abs(beta)<tol):
242             print("starting vector is already solution")
243             return x0
244
245         z.append(beta)
246         ## Start of the inner Loop (calc krylows and Hessenberg and relativ residual)
247         while(k<restart and eps[k]/beta>tol and k<len(x0) ):
248
249             ## if the start vektor doesn't lead to an improvement
250             ## throw an error
251             if(eps[0]>1.0e30):
252                 print('Unexpected error. With the choosen start vector was no improvement possible')
253                 print('returning the startvektor')
254                 return x0
255
256
257             ### Calc the next Krylow
258             omega.append(evaluation(H,S,spectrum,v[k],tidyup)-shift*v[k])
259
260
261             for i in range(k+1):
262                 Hess[i,k]=np.vdot(v[i],omega[k])
263                 omega[k]=omega[k]-Hess[i,k]*v[i]
264
265             Hess[k+1,k]=np.linalg.norm(omega[k])
266             v.append(omega[k]/Hess[k+1,k])
267
268
269             V[k]=v[k]
270
271             ## Use the old Given rotations
272
273             for i in range(k):
274                 #important use temps to stor the values
275                 temp1=cos[i]*Hess[i,k]+ -sin[i]*Hess[i+1,k]
276                 temp2=sin[i]*Hess[i,k]+cos[i]*Hess[i+1,k]
277                 Hess[i,k]=temp1
278                 Hess[i+1,k]=temp2
279
280
281
282             #calc new Givensrotation  with tau for stabilisation
283             tau=np.abs(Hess[k,k])+np.abs(Hess[k+1,k])
284             nu=tau*np.sqrt(((Hess[k,k]/tau)**2)+((Hess[k+1,k]/tau)**2))
285
286             cos.append(Hess[k,k]/nu)
287             sin.append(-Hess[k+1,k]/nu)
288
289             Hess[k,k]=nu
290             Hess[k+1,k]=0
```

```
291             #print(np.real(Hess))
292
293             #calc approx
294             z.append(sin[k]*z[k])
295             z[k]=cos[k]*z[k]
296
297             ##the new relative residual
298             eps.append(np.abs(z[k+1])/beta)
299
300             ## increase the iteration variable
301             #print("relative residual: " +str(eps[k+1]/beta))
302             k=k+1
303
304         y=np.zeros(len(x0),dtype='complex')
305         y[k-1]=z[k-1]/Hess[k-1,k-1]
306
307
308         ##berechne x gemaes Rx=z
309         for t in range(k-1,-1,-1):
310             summe=0
311             #print("t: " +str(t))
312             for s in range(t+1,k):
313                 #print("s: " +str(s))
314                 summe+=Hess[t,s]*y[s]
315             y[t]=(z[t]-summe)/Hess[t,t]
316
317
318         x=np.zeros(len(x0),dtype='complex')
319
320         for s in range(k):
321             x=x+y[s]*v[s]
322
323         x=x+x0
324         # if inner loop stoped by reaching the maximal iteration, call the function with
325         # new better approximation, to reach the tolerance
326         # reduce the maximal iteration of outer iterations
327
328         if(k==restart):
329             print("failed to converge in " +str(restart) +" steps. RESTART")
330             #print("actual approximation: " +str(np.linalg.norm(evaluation(x)-shift*x-b)))
331             x=gmres_ext_ev(H,S,spectrum,evaluation,b,x,restart,tol,shift,maxiter-1,tidyup)
332
333         # if the inner loop stopped after reaching the tol return the solution
334         #print(x)
335         return x
336
337     def auswertung(H,S,spectrum,x,tidyup):
338         l=len(x)
339         l=np.sqrt(l)
340         x=np.reshape(x,(l,l),order='F')
341         x=Qobj(x)
342         x=evaluation(H,S,x,spectrum,tidyup)
343         x=operator_to_vector(x).full()
344         x2=np.zeros(len(x),dtype="complex")
345
346         for i in range(len(x)):
347             x2[i]=x[i][0]
348
349         return x2
350
351     def statState(H,S,ohmic_spectrum,x0,shift,restart,
352                   tol_gmres,maxiter_gmres,tol_pow,maxiter_pow,tidyup=True):
353
354         #make start vector hermitian if needed
355         l=len(x0)
356         s=np.sqrt(l)
357         xs=np.reshape(x0,(s,s),order='F')
358         x0=0.5*(np.conj(np.transpose(xs))+xs)
359         x0=np.reshape(x0,(l))
360
361         norm=np.linalg.norm(x0)
362         x0=x0/norm
363
364         x=inv_PowIteration(H,S,ohmic_spectrum,auswertung,x0,shift,restart,
```

```
365                             tol_gmres,maxiter_gmres,tol_pow,maxiter_pow,tidyup)
366
367
368     l=len(x)
369     s=np.sqrt(l)
370     x=np.reshape(x,(s,s),order='F')
371
372     if tidyup==True:
373         x=Qobj(x).tidyup()
374
375     return x
```

A.2. Quellcode C

C-Modul zur Beschleunigten Auswertung der Bloch-Redfield Mastergleichung

```
1  #include <Python.h>
2  #include <numpy/arrayobject.h>
3  #include <math.h>
4
5  double ohmic_spectrum(double x){
6          if(x==0){
7                  return(10.0);
8          }else{
9                  return(x/(1-exp(-0.1*x)));
10         }
11 }
12
13 static PyObject *matrixG_calcG(PyObject *self, PyObject *args)
14 {
15     PyArrayObject *in_array; //input argument
16     PyObject      *out_array; //output argument
17     NpyIter *in_iter;           //iteration ueber input
18     NpyIter *out_iter;          //iteration ueber output
19     NpyIter_IterNextFunc *in_iternext;
20     NpyIter_IterNextFunc *out_iternext;
21
22     /*  parse single numpy array argument */
23     if (!PyArg_ParseTuple(args, "O!", &PyArray_Type, &in_array))
24         return NULL;
25
26     /*  construct the output array, like the input array */
27     out_array = PyArray_NewLikeArray(in_array, NPY_ANYORDER, NULL, 0);
28     if (out_array == NULL)
29         return NULL;
30
31     /*  create the iterators */
32     in_iter = NpyIter_New(in_array, NPY_ITER_READONLY, NPY_KEEPORDER,
33                           NPY_NO_CASTING, NULL);
34     if (in_iter == NULL)
35         goto fail;
36
37     out_iter = NpyIter_New((PyArrayObject *)out_array, NPY_ITER_READWRITE,
38                         NPY_KEEPORDER, NPY_NO_CASTING, NULL);
39     if (out_iter == NULL) {
40         NpyIter_Deallocate(in_iter);
41         goto fail;
42     }
43
44     in_iternext = NpyIter_GetIterNext(in_iter, NULL);
45     out_iternext = NpyIter_GetIterNext(out_iter, NULL);
46     if (in_iternext == NULL || out_iternext == NULL) {
47         NpyIter_Deallocate(in_iter);
48         NpyIter_Deallocate(out_iter);
49         goto fail;
50     }
51     double ** in_dataptr = (double **) NpyIter_GetDataPtrArray(in_iter);
52     double ** out_dataptr = (double **) NpyIter_GetDataPtrArray(out_iter);
53
54     /*  iterate over the arrays */
55     do {
```

```
56          **out_dataptr = ohmic_spectrum(**in_dataptr)*0.5;
57      } while(in_iternext(in_iter) && out_iternext(out_iter));
58
59      /*  clean up and return the result */
60      NpyIter_Deallocate(in_iter);
61      NpyIter_Deallocate(out_iter);
62      Py_INCREF(out_array);
63      return out_array;
64
65      /*  in case bad things happen */
66      fail:
67          Py_XDECREF(out_array);
68          return NULL;
69
70  }
71
72  static PyMethodDef matrixGMethods[] = {
73    {"calcG", matrixG_calcG, METH_VARARGS,
74      "returns the number."},
75    {NULL, NULL, 0, NULL},
76  };
77
78  void initmatrixG(void)
79  {
80    (void)Py_InitModule("matrixG", matrixGMethods);
81       /* IMPORTANT: this must be called */
82       import_array();
83
84  }
```

Literatur

[Car] H. J. Carmichael. *Statistical Methods in Quantum Optics 1, Master Equations and Fokker-Planck Equations*. Springer Verlag, 2. Auflage 2002.

[CT] Claude Cohen-Tannoudji. *Quantenmechanik 1*. Walter de Gruyter Verlag, 2. Auflage 1999.

[Hae15] Valentin Haenel. Interfacing with C. `http://www.scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html`, 4.10.2015.

[Hoc] Marlis Hochbruck. *Numerische Mathematik I und II*. Vorlesungskript zur Vorlesung Numerische Mathematik II am Karlsruher Institut für Technologie im Sommersemeseter 2015.

[Mis11] Daniel Mischek. *Relaxation eines 2-Niveau-Systems unter dem Einfluss eines Phasenqubits*. 2011.

[PJ] P.D.Nation and J.R. Johansson. *QuTiP: Quantum Toolbox in Python*. Documentation for Qutip, 31 December, 2014.

[PNF] J.R. Johansson P.D. Nation and F.Nori. *QuTiP 2: A Python framework for the dynamics of open quantum systems*. Comp. Phys. Comm. 184, 1234 (2013).

[Sch] Dr. Gernot Schaller. *Non-Equilibrium Master Equations*. Vorlesungsskript Wintersemester 2011/2012, TU Berlin.

[Sci15] SciPy Dokumentation, Artikel zur csr_ matrix. `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.csr_matrix.html`, 15.10.2015.

[SS86] Youcef Saad and Martin H. Schulz. *GMRES: A generalized minimal residual algorithm for solving nonsymmetrix linear systems*. Society for Industrial and Applied Mathematics, 1986.

[Tho98] Michael Thoß. *Modellierung dissipativer Prozesse in Rydbergzuständen großer Moleküle sowie in Elektron-Molekül-Stoßkomplexen*. 1998.

[Vog11] Nicolas Vogt. Three level systems and decoherence. Master's thesis, Karlsruhe Institute of Technology, 2011.

[Wik15a] Wikipedia. Artikel zu Python. `https://de.wikipedia.org/wiki/Dichteoperator`, 12.12.2015.

[Wik15b] Wikipedia. Artikel zum Dichteoperator. `https://de.wikipedia.org/wiki/Python_(Programmiersprache)`, 29.09.2015.

[Wik15c] Wikipedia. Artikel zur Spur über ein Subsystem. `https://en.wikipedia.org/wiki/Partial_trace`, 29.09.2015.