# Simulating open quantum systems with high photon numbers in coherent bases

**Masterarbeit in der Studienrichtung Physik**

zur Erlangung des akademischen Grades

Master of Science (MSc.)

eingereicht an der

**Fakultät für Mathematik, Informatik und Physik**
**der Universität Innsbruck**

von

**Sebastian Krämer, BSc.**

Betreuer der Masterarbeit:

Mag.Dr. Helmut Ritsch, Institut für Theoretische Physik

Innsbruck, August 2011

# Preface

This master thesis was written at the University of Innsbruck at the Institute of Theoretical Physics under supervision of Mag.Dr. Helmut Ritsch and with help from Dr. Andras Vukics. I would like to thank them at this point for their great help and support. Many other people also deserve to be mentioned because of their great influence on this work: My fellow students Cornelia Spee and Hannes Pichler for helpful discussions and my coworkers Wolfgang Niedenzu, Tobias Grießer and Matthias Sonnleitner for valuable feedback and good atmosphere in the office. But most of all I would like to thank my parents who made it possible for me to reach this point of my life.

Innsbruck, Juni 2011

# CONTENTS

# Introduction

In quantum physics only very few examples exist that are analytically solvable. Most problems have to be treated by either using approximations or have to be solved numerically. For performing such numerical simulations a computational basis with a limited amount of dimensions has to be chosen. The energy eigenbasis is a good choice when the total energy in the system is low. States corresponding to energies above some threshold can then be neglected. At higher energies this is not longer possible because the amount of states necessary to describe the problem is too large. A good example for this problem is a laser. Inside the optical cavity of a laser a single mode can have millions of photons. The exact quantum state of the field can analytically be described as *coherent state*. A summary of their properties is given in *Coherent states* (chapter 1) but what is important at this point is that the variance of the photon number grows with the amount of photons. This means that also the dimension of the computational basis has to be increased accordingly. Choosing other kinds of bases can circumvent this limitation. Due to the fact that coherent states are complete an obvious idea is to use a basis consisting of coherent states to simulate such problems. How these coherent states have to be chosen from the infinite set of possibilities is discussed in the chapter *Coherent state bases* (chapter 5). The hope is that this will allow for simulations to be done in much lower dimensions but with the same accuracy as compared to number bases. Keeping this goal in mind it is first necessary to gain a deeper understanding of the time evolution of problems in coherent bases. In *Nonorthogonal bases* (chapter 2) properties and notation of nonorthogonal bases are introduced and in chapter *Time evolution in nonorthogonal bases* (chapter 4) the time evolution according to master equations and MCWF-method (which is described in *MCWF method* (chapter 3)) are examined. What the deciding factors are that determine which basis is preferable for what problem is treated in chapter *Complexity of algorithm using adaptive coherent bases* (chapter 8). To be usable for a bigger amount of problems it proves necessary to allow the coherent basis to change over time. This idea is investigated in the chapter *Adaptive bases* (chapter 7). In practice it turns out that achieving a high precision in simulations in coherent bases is difficult. Bases which in principle allow for a good representation of the problem have the tendency to induce a high numerical error in the dynamical simulations. This effect is studied in *Condition numbers of problems in coherent bases* (chapter 6) where also a way to treat this difficulty is described.

Building on the theoretical work done in the first part of the thesis, the next step was to write a solver using these ideas. Instead of doing this from scratch support for nonorthogonal bases, especially for coherent bases, was implemented in C++QED [cppqed]. As stated on its website, *C++QED is a highly flexible framework for simulating open quantum dynamics*, which was created by Dr. Andras Vukics [Vukics-Ritsch]. The code written in scope of this thesis can also be found on the website, partly merged with the main branch and partly in a separate branch. Although this took a great part of the time spent on the thesis this work might not be too interesting for the reader. Therefore only an overview about the design is given in the second part (*Code Design* (Part II)).

Finally, in an additional third part, the C++QED implementation was used to test the theoretical results. Several systems (*Single mode in cavity* (chapter 14), *Pumped lossy mode* (chapter 15) and *Squeezed state* (chapter 16)) were simulated and the numerical outcome was compared to the theoretical predictions.

# Part I

# Fundamentals

CHAPTER

ONE

# COHERENT STATES

## 1.1 Introduction

In view of the throughout usage of coherent states in this thesis a short summary of their definition and properties is given. Many other resources exist which treat this topic extensively and in much greater depth, e.g. [Gerry] and [Walls-Milburn].

## 1.2 Definition

In literature, several different approaches exist to define coherent states. Here, they are defined as eigenstates of the destruction operator:

$$a|\alpha\rangle = \alpha|\alpha\rangle$$

Expressing $|\alpha\rangle$ in the number base results in

$$|\alpha\rangle = e^{-\frac{|\alpha|^2}{2}} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle$$

which can also be written using an exponential function:

$$|\alpha\rangle = e^{\alpha a^\dagger - \alpha^* a}|0\rangle$$

The occurring exponential function is also called *displacement operator*.

## 1.3 Displacement operator

The displacement operator proves to be very handy for many calculations. Premise is of course that some of its properties are known:

1. Definition

$$D(\alpha) = e^{\alpha a^\dagger - \alpha^* a} = e^{-\frac{|\alpha|^2}{2}} e^{\alpha a^\dagger} e^{-\alpha^* a} = e^{\frac{|\alpha|^2}{2}} e^{-\alpha^* a} e^{\alpha a^\dagger}$$

2. Adjoint

$$D^\dagger(\alpha) = D^{-1}(\alpha) = D(-\alpha)$$

3. Composition of displacements

$$D(\alpha)D(\beta) = D(\alpha+\beta)e^{\frac{1}{2}(\alpha^*\beta - \alpha\beta^*)}$$

4. Commutation relations

$$[a, D(\alpha)] = \alpha D(\alpha)$$

5. Similarity transformations

$$D^\dagger(\alpha)aD(\alpha) = a + \alpha$$
$$D(\alpha)aD^\dagger(\alpha) = a - \alpha$$

## 1.4 Scalar product

Using the displacement operator, it can immediately be shown that the scalar product is:

$$\langle\alpha|\beta\rangle = e^{-\frac{1}{2}\left(|\alpha|^2+|\beta|^2\right)+\alpha^*\beta}$$

Also interesting is the norm of the scalar product:

$$|\langle\alpha|\beta\rangle|^2 = e^{-|\alpha-\beta|^2}$$

Only in the limit $|\alpha - \beta| \gg 1$ the corresponding states become more and more orthogonal.

## 1.5 Completeness

For coherent states the following completeness relation can be shown:

$$\frac{1}{\pi}\int |\alpha\rangle\langle\alpha| d^2\alpha = 1$$

Here the integral is taken over the whole complex plane. Any state vector in the same Hilbert space can therefore be represented in terms of coherent states:

$$|\Psi\rangle = \frac{1}{\pi}\int d^2\alpha |\alpha\rangle\langle\alpha|\Psi\rangle$$

In fact the coherent states are overcomplete as can be seen by using this formula for representing a coherent state $|\beta\rangle$:

$$\begin{aligned}|\beta\rangle &= \frac{1}{\pi}\int d^2\alpha |\alpha\rangle\langle\alpha|\beta\rangle \\ &= \frac{1}{\pi}\int d^2\alpha |\alpha\rangle e^{-\frac{1}{2}\left(|\alpha|^2+|\beta|^2\right)+\alpha^*\beta}\end{aligned}$$

Due to this overcompleteness there exist different subsets of the complex plane that by themselves are still complete. An example for these subsets are von Neumann lattices [Bargmann] which take only a discrete set of coherent states $\alpha_{kl} = kd_r + ild_i$ $(k,l) \in \mathcal{Z}$ on a rectangular lattice where the two lattice constants obey certain restrictions.

## 1.6 Phase-space pictures of coherent states

In quantum mechanics the harmonic oscillator is often solved by writing down the Hamiltonian for the operators $x$ and $p$ and then introducing dimensionless creation and annihilation operators. For coherent states this can be done the other way round by defining quadrature operators $X_1$ and $X_2$ which play the role of position and momentum operators respectively:

$$\begin{aligned}X_1 &= \frac{a+a^\dagger}{2} \\ X_2 &= \frac{a-a^\dagger}{2i}\end{aligned}$$

Interesting are their expectation values for a coherent state $|\alpha\rangle$,

$$\begin{aligned}\langle X_1\rangle_\alpha &= \operatorname{Re}\alpha \\ \langle X_2\rangle_\alpha &= \operatorname{Im}\alpha\end{aligned}$$

indicating a correspondence between the complex $\alpha$-plane and phase space. It can be shown that for coherent states the variance in these two operators is completely the same:

$$\begin{aligned}\operatorname{Var}_\alpha(X_1) &= \frac{1}{4} \\ \operatorname{Var}_\alpha(X_2) &= \frac{1}{4}\end{aligned}$$

Calculating the commutator between $X_1$ and $X_2$,

$$[X_1, X_2] = \frac{1}{2i}$$

and comparing it with their variances reveals that coherent states have a minimal uncertainty:

$$\operatorname{Var}_\alpha(X_1)\operatorname{Var}_\alpha(X_2) = \left(\frac{1}{4}\right)^2 = \left(\frac{1}{2i}[X_1, X_2]\right)^2$$

## 1.7 Squeezed states

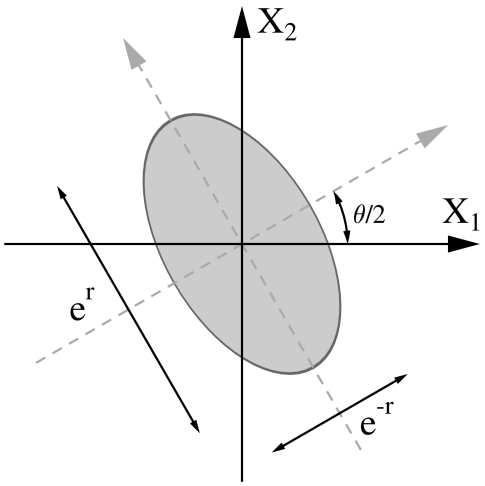
States with minimal uncertainty but unequal variances in the quadrature operators are commonly called (quadrature) squeezed states. These states can easiest be obtained by using the squeezing operator, which is defined as

$$S(\xi) = e^{\frac{1}{2}\left(\xi^* a^2 - \xi (a^\dagger)^2\right)}$$

where $\xi$ is a complex number controlling how much a state is squeezed. Using the polar representation, $\xi = re^{i\theta}$, the following two relations can be obtained which greatly simplify the calculations of expectation values and variances of squeezed states:

$$S^\dagger(\xi) a S(\xi) = a \cosh r - a^\dagger e^{i\theta} \sinh r$$
$$S^\dagger(\xi) a^\dagger S(\xi) = a^\dagger \cosh r - a e^{-i\theta} \sinh r$$

Applying the squeezing operator with $\theta = 0$ onto the vacuum state results in a scaling of the variances of $X_1$ and $X_2$ of $e^{-2r}$ and $e^{2r}$ respectively. Choosing $\theta$ other than zero results in a squeezing along rotated axes, which is shown in following sketch:



CHAPTER

TWO

# NONORTHOGONAL BASES

Nonorthogonal bases are used much less often for numerical calculations than orthogonal ones because the missing orthogonality tends to make calculations more tedious. For example it is not possible to use the usual completeness relations. Formally this can be fixed by introducing a so called *dual basis* but this goes along with additional complexity since all operators and states can be represented in various ways. In the following a notation for nonorthogonal bases is introduced and some important properties are stated.

## 2.1 Notation of nonorthogonal basis and dual basis

For any nonorthogonal basis, which will be written as

$$\underline{b_\bullet} = \{|i\rangle_\bullet\}_i$$

a dual basis

$$\underline{b^\bullet} = \{|i\rangle^\bullet\}_i$$

can be defined by demanding orthogonality between the original basis states and the dual basis states:

$${}^\bullet\langle i|j\rangle_\bullet = \delta^i_j$$

In this case $\delta^i_j$ is the usual kronecker delta. From now on only deltas with both upper and lower indices ($\delta^i_j$ and $\delta^j_i$) are equal to the usual kronecker delta while deltas with only lower or only upper indices will get some different meaning.

## 2.2 Abstract index notation

Normally, the coordinates of a state vector (e.g. $|v\rangle$) in some basis are denoted by a symbol (e.g. $v$) and an element of this vector is denoted by the same symbol plus an index (e.g. $v_i$). In the case of nonorthogonal bases, this notation is ambiguous because one state vector is often represented in both, the original basis and the dual basis. Although related, these coordinates are completely different and have somehow to be distinguished. This problem does not occur

for the elements of the coordinate vectors - the position of the indices signalizes relative to which basis the coordinates are. To get an analogous distinction between the coordinate vectors, dots will be used instead of indices to indicate in which basis they are.

| Vector | Element |
|---|---|
| $v^\bullet$ | $v^i$ |
| $v_\bullet$ | $v_i$ |

This notation can easily be generalized to matrices and will indeed also be used.

## 2.3 Basis transformation

Between different bases of a finite vector space always exists a bijective linear transformation:

$$|j\rangle^\bullet = \sum_i \delta^{ij}|i\rangle_\bullet = \begin{pmatrix} |i_0\rangle_\bullet, & |i_1\rangle_\bullet, & \dots \end{pmatrix} * \begin{pmatrix} \delta^{0j} \\ \delta^{1j} \\ \vdots \end{pmatrix}$$

$$|j\rangle_\bullet = \sum_i \delta_{ij}|i\rangle^\bullet = \begin{pmatrix} |i_0\rangle^\bullet, & |i_1\rangle^\bullet, & \dots \end{pmatrix} * \begin{pmatrix} \delta_{0j} \\ \delta_{1j} \\ \vdots \end{pmatrix}$$

Using matrix notation these relations can shortly be written as:

$$b^\bullet = b_\bullet * \delta^{\bullet\bullet}$$
$$b_\bullet = b^\bullet * \delta_{\bullet\bullet}$$

The transformation matrices can be calculated using the relations:

$$\delta^{ij} = {}^\bullet\langle i|j\rangle^\bullet$$
$$\delta_{ij} = {}_\bullet\langle i|j\rangle_\bullet$$

Of course the two transformation matrices $\delta^{\bullet\bullet}$ and $\delta_{\bullet\bullet}$ are related - one is the inverse of the other!

$$\delta_{\bullet\bullet} * \delta^{\bullet\bullet} = I$$

Another important property is that the transformation matrices are hermitian:

$$\delta^{ij} = \left(\delta^{ij}\right)^\dagger$$
$$\delta_{ij} = \delta^\dagger_{ij}$$

## 2.4 Coordinates

Any state can be expressed in the nonorthogonal basis or in its corresponding dual basis:

$$|\Psi\rangle = \sum_i c^i|i\rangle_\bullet = b_\bullet * c^\bullet$$
$$= \sum_i c_i|i\rangle^\bullet = b^\bullet * c_\bullet$$

The coordinate with respect to one basis state can be calculated by projecting the state vector onto the dual basis state:

$$c^i = {}^\bullet\langle i|\Psi\rangle$$
$$c_i = {}_\bullet\langle i|\Psi\rangle$$

The coordinates transform with the same transformation matrix as the bases:

$$c^\bullet = \delta^{\bullet\bullet} * c_\bullet$$
$$c_\bullet = \delta_{\bullet\bullet} * c^\bullet$$

## 2.5 Completeness

The completeness relation for nonorthogonal bases looks like:

$$\begin{aligned} 1 &= \sum_i |i\rangle_\bullet{}^\bullet\langle i| \\ &= \sum_i |i\rangle^\bullet{}_\bullet\langle i| \end{aligned}$$

This can easily be verified by using the fact that any state can be expressed in any basis and using the orthogonality of dual state vectors:

$$\begin{aligned} \left(\sum_i |i\rangle_\bullet{}^\bullet\langle i|\right)|\Psi\rangle = \left(\sum_i |i\rangle_\bullet{}^\bullet\langle i|\right)\sum_j c^j|j\rangle_\bullet = \sum_{ij} c^j|i\rangle_\bullet{}^\bullet\langle i|j\rangle_\bullet \\ = \sum_i c^i|i\rangle_\bullet = |\Psi\rangle \end{aligned}$$

## 2.6 Norm

Writing down the state in both the nonorthogonal basis and also in the dual basis

$$|\Psi\rangle = \sum_i c^i|i\rangle_\bullet$$
$$\langle\Psi| = \sum_j c^*_j{}^\bullet\langle j|$$

gives a short formula for calculations of norms:

$$\langle\Psi|\Psi\rangle = \sum_{ij} c^i c^*_j{}^\bullet\langle j|i\rangle_\bullet = \sum_i c^i c^*_i = c^\bullet * c^*_\bullet$$

Note that the norm of the dual vectors is not 1.

## 2.7 Operators

Every operator can be represented in four different ways:

$$\begin{aligned} A_{ij} &= {}_{\bullet}\langle i|A|j\rangle_{\bullet} \\ A_i^j &= {}_{\bullet}\langle i|A|j\rangle^{\bullet} \\ A_i^j &= {}^{\bullet}\langle i|A|j\rangle_{\bullet} \\ A^{ij} &= {}^{\bullet}\langle i|A|j\rangle^{\bullet} \end{aligned}$$

Transformations between these representations can be summarized in two rules:

1. To change first (second) index multiply from left (right).
2. To raise (lower) an index use $\delta^{ij}$ ($\delta_{ij}$).

## 2.8 Traces

Taking the trace of some operator using orthogonal bases is done by evaluating the following sum over all basis states:

$$tr\{A\} = \sum_n \langle n|A|n\rangle$$

Let us generalize this for nonorthogonal bases:

$$tr\{A\} = \sum_n \langle n|A|n\rangle = \sum_{nij} \langle n|i\rangle_{\bullet}{}^{\bullet}\langle i|A|j\rangle_{\bullet}{}^{\bullet}\langle j|n\rangle = \sum_{ij} \delta_i^j \, {}^{\bullet}\langle i|A|j\rangle_{\bullet} = \sum_i {}^{\bullet}\langle i|A|i\rangle_{\bullet}$$

So basically there are two ways of calculating traces:

$$\begin{aligned} tr\{A\} &= \sum_i {}^{\bullet}\langle i|A|i\rangle_{\bullet} = \sum_i A_i^i \\ &= \sum_i {}_{\bullet}\langle i|A|i\rangle^{\bullet} = \sum_i A_i^i \end{aligned}$$

## 2.9 Expectation Values

**State Vectors**

Depending on the basis an operator is given in there are slightly different ways of calculating the expectation value for a given state. By inserting the nonorthogonal identity relations following identities can be shown:

$$\begin{aligned} \langle \Psi|A|\Psi\rangle &= (c^{\bullet})^* * A_{\bullet\bullet} * c^{\bullet} \\ &= (c^{\bullet})^* * A_{\bullet}^{\bullet} * c_{\bullet} \\ &= (c_{\bullet})^* * A_{\bullet}^{\bullet} * c^{\bullet} \\ &= (c_{\bullet})^* * A^{\bullet\bullet} * c_{\bullet} \end{aligned}$$

Again there are two rules summarizing these formulas:

1. Multiply the operator matrix from the right (left) side with the (conjugated) coordinates.
2. For the left (right) multiplication use the coordinates dual to the left (right) base of the operator.

**Density operators**

Like in the case of nonorthogonal state vectors there are four different ways to calculate the expectation values:

$$\begin{aligned} tr\{A\} = \sum_i (\rho A)_i^i &= \sum_{ij} \rho^i{}_j A^j_i \\ &= \sum_{ij} \rho^j_i A^i_j \\ &= \sum_{ij} \rho^{ij} A_{ji} \\ &= \sum_{ij} \rho_{ij} A^{ji} \end{aligned}$$

CHAPTER

THREE

# MCWF METHOD

Solving master equations is computational, both in memory cost and also in runtime, a very expensive process. The MCWF-method provides a way out of this problem by working on state vectors instead of density matrices [Dum-Zoller-Ritsch]. The drawback of the method is its stochastic nature which means that it equals only in the mean the result of the corresponding master equation. Therefore it is necessary to repeatedly perform the MCWF-method and finally take the average over the different outcomes. Fortunately it turns out that already a small number of simulations provide a good approximation to the result of the master equation. The connection between the master equation and the MCWF-method can best be seen if the master equation is written in a slightly different form.

## 3.1 Rewriting the Master Equation

The master equation can compactly be written as

$$\dot{\rho} = \frac{1}{i\hbar}[H, \rho] + \mathcal{L}[\rho]$$

where the general Liouvillian $\mathcal{L}$ is given by

$$\mathcal{L}[\rho] = \sum_m (J_m \rho J_m^\dagger - \frac{1}{2}[J_m^\dagger J_m, \rho]_+)$$

Reordering the terms and introducing a non-hermitian Hamiltonian

$$H_{nH} = H - \frac{i\hbar}{2} \sum_m J_m^\dagger J_m$$

yields in

$$\dot{\rho} = \frac{1}{i\hbar} \left( H_{nH}\rho - \rho H_{nH}^\dagger \right) + \sum_m J_m \rho J_m^\dagger$$

In the MCWF-method the first term will be responsible for the non-unitary time evolution, while the second term will lead to quantum jumps at randomly chosen times.

## 3.2 The MCWF method

At this point the MCWF-method will be introduced as simple prescription for how the time evolution has to be done. Why and in what sense this method generates the same result as the corresponding master equation is shown afterwards.

The MCWF-method consists of two different kinds of time evolutions that have to be done for every time step:

1. **Non-unitary evolution**

   This evolution is according to a Schroedinger equation with the above stated non-hermitian Hamiltonian $H_{nH}$. In first order of time this can be written as:

   $$|\dot{\Psi}^{(1)}(t+\delta t)\rangle = e^{-iH_{nH}\delta t/\hbar}|\Psi(t)\rangle = \left(1 - \frac{iH_{nH}\delta t}{\hbar}\right)|\Psi(t)\rangle + \mathcal{O}(\delta t^2)$$

   Since the Hamiltonian is non-hermitian, the norm of the state vector is not preserved. The square of the norm is:

   $$\begin{aligned}\langle\Psi^{(1)}(t+\delta t)|\Psi^{(1)}(t+\delta t)\rangle &= \langle\Psi(t)|\left(1 + \frac{iH_{nH}^\dagger\delta t}{\hbar}\right)\left(1 - \frac{iH_{nH}\delta t}{\hbar}\right)|\Psi(t)\rangle + \mathcal{O}(\delta t^2)\\ &= 1 - \delta t\frac{i}{\hbar}\langle\Psi(t)|H - H^\dagger|\Psi(t)\rangle + \mathcal{O}(\delta t^2)\\ &= 1 - \sum_m \delta t\langle\Psi(t)|J_m^\dagger J_m|\Psi(t)\rangle + \mathcal{O}(\delta t^2)\\ &= 1 - \sum_m \delta p_m + \mathcal{O}(\delta t^2) = 1 - \delta p + \mathcal{O}(\delta t^2)\end{aligned}$$

2. **Quantum jump**

   This part of the time evolution introduces a stochastic element to the calculations. Every jump operator $J_m$ has a certain probability of performing a quantum jump. The probability that this happens in a time interval $\delta t$ depends on the loss of norm caused by the specific jump operator in the non-unitary evolution. Thus these probabilities are the previously calculated $\delta p_m$. In simulations random numbers are used to determine if a quantum jump occurs. If no jump occurs, the state vector is simply renormalized:

   $$|\Psi(t+\delta t)\rangle = \frac{|\Psi^{(1)}(t+\delta t)\rangle}{\sqrt{1-\delta p}} + \mathcal{O}(\delta t^2)$$

   Otherwise a jump is performed by projecting the state vector onto the m-th jump operator and normalizing the result.

   $$|\Psi(t+\delta t)\rangle = \sqrt{\delta t}\frac{J_m|\Psi(t)\rangle}{\sqrt{\delta p_m}} + \mathcal{O}(\delta t^2)$$

## 3.3 Equivalence with the master equation

It still has to be shown that the MCWF-method indeed generates the same result as the master equation in the mean. For sake of simplicity this will only be done in in first order of time.

Let us assume that at some point of time $t$ the system is in some initial state $|\Psi(t)\rangle$ which is equivalent to the density operator $\sigma(t) = |\Psi(t)\rangle\langle\Psi(t)|$. Evolving this state for a short time (short in the sense that the probability for a jump in this time is small) can have according to the MCWF-method different outcomes. Every outcome happens with a certain probability. Repeating this process many times and taking the average results in a mixed state which can be written as

$$\begin{aligned}
\overline{\sigma}(t+\delta t) &= (1-\delta p)\frac{|\Psi^{(1)}(t+\delta t)\rangle}{\sqrt{1-\delta p}}\frac{\langle\Psi^{(1)}(t+\delta t)|}{\sqrt{1-\delta p}} + \sum_m \delta t \delta p_m \frac{J_m|\Psi(t)\rangle}{\sqrt{\delta p_m}}\frac{\langle\Psi(t)|J_m^\dagger}{\sqrt{\delta p_m}} + \mathcal{O}(\delta t^2) \\
&= |\Psi^{(1)}(t+\delta t)\rangle\langle\Psi^{(1)}(t+\delta t)| + \delta t \sum_m J_m|\Psi(t)\rangle\langle\Psi(t)|J_m^\dagger + \mathcal{O}(\delta t^2) \\
&= \left(1 - \frac{iH_{nH}\delta t}{\hbar}\right)\sigma(t)\left(1 + \frac{iH_{nH}^\dagger\delta t}{\hbar}\right) + \delta t\sum_m J_m\sigma(t)J_m^\dagger + \mathcal{O}(\delta t^2) \\
&= \sigma(t) + \delta t\frac{1}{i\hbar}\left(H_{nH}\sigma(t) - \sigma(t)H_{nH}^\dagger\right) + \delta t\sum_m J_m\sigma(t)J_m^\dagger + \mathcal{O}(\delta t^2)
\end{aligned}$$

The master equation can be in first order in time brought into the exact same form:

$$\rho(t+\delta t) = \rho(t) + \delta t\frac{1}{i\hbar}\left(H_{nH}\rho(t) - \rho(t)H_{nH}^\dagger\right) + \delta t\sum_m J_m\rho(t)J_m^\dagger + \mathcal{O}(\delta t^2)$$

This proves the equivalence of the MCWF-method with the master equation in the first order of time. There exist higher order methods [Steinbach] but these are not implemented in C++QED and therefor won't be discussed at this point.

CHAPTER

FOUR

# TIME EVOLUTION IN NONORTHOGONAL BASES

Translating operator equations into equivalent matrix equations is straight forward in the case of orthogonal bases. If nonorthogonal bases are used one has to think more carefully about this translation process. Since the identity relation makes use of both the original basis and also the dual basis it turns out that the matrix equations for non-trivial problems will have the simplest form if the basis and the dual basis are used. A priori it is not clear which different representations are possible and what advantages and disadvantages they might have. However, some basic rules can be obtained which greatly simplify the selection of favourable representations.

1. *Multiplications between operators should always be done between lower and upper indices.*

   This can be most easily seen by the form of the identity operator for nonorthogonal bases. Inserted between two operators it projects the operator on the right side onto the original basis creating a lower index and the operator on the left side onto the dual basis creating an upper index. This rule can be bent by inserting a transformation matrix between the two matrix representations, but this is numerically not optimal.

$$\begin{aligned}(A*B)^i{}_j &= {}^\bullet\langle i|A*B|j\rangle_\bullet \\ &= \sum_k {}^\bullet\langle i|A|k\rangle_\bullet {}^\bullet\langle k|B|j\rangle_\bullet = \sum_k A^i{}_k * B^k{}_j \\ &= \sum_k {}^\bullet\langle i|A|k\rangle^\bullet {}_\bullet\langle k|B|j\rangle_\bullet = \sum_k A^{ik} * B_{kj}\end{aligned}$$

2. *Adjoint operations have to be applied to operators in a pure basis.*

   The reason for this rule is that the matrix representation of the adjoint of an operator only equals the adjoint matrix representation of this operator if it is in a pure basis:

$$\begin{aligned}(A_{ij})^* &= ({}_\bullet\langle i|A|j\rangle_\bullet)^* = \left({}_\bullet\langle j|A^\dagger|i\rangle_\bullet\right)^T = {}_\bullet\langle i|A|j\rangle_\bullet = \left(A^\dagger\right)_{ij} \\ \left(A_i^{\,j}\right)^* &= ({}^\bullet\langle i|A|j\rangle_\bullet)^* = \left({}_\bullet\langle j|A^\dagger|i\rangle^\bullet\right)^T = {}_\bullet\langle i|A|j\rangle^\bullet = \left(A^\dagger\right)^i{}_j\end{aligned}$$

3. *All terms of a sum have to be in the same representation.*

$$(A+B)_{ij} = {}_\bullet\langle i|\,(A+B)\,|j\rangle_\bullet = {}_\bullet\langle i|A|j\rangle_\bullet + {}_\bullet\langle i|B|j\rangle_\bullet = A_{ij} + B_{ij}$$

4. *Density operators and state vectors should only occur in one representation.*

   If this is not the case, in every time step computational expensive basis transformations have to be done.

The two basic time evolution algorithms implemented in C++QED are now analyzed for their suitability for calculations in nonorthogonal bases. Using the above stated rules to derive appropriate matrix equations from the fundamental operator equations assures that the obtained equations are numerically as cheap as possible.

## 4.1 Master Equation

The first algorithm implemented in C++QED directly solves the master equation, albeit not exactly in the ordinarily used form. Instead it utilizes the formulation with non-hermitian Hamiltonians and jump operators which was derived in the previous chapter. With the operator valued *real* function defined as $\mathrm{Re}(A) = \frac{1}{2}(A + A^\dagger)$ the following equation can be obtained:

$$\dot{\rho} = 2\,\mathrm{Re}\left(\frac{1}{i\hbar} H_{nH}\rho\right) + \sum_m J_m \left(J_m\rho\right)^\dagger$$

This operator equation is the starting point for the C++QED implementation. Now the previously stated rules are used to translate this operator equation into a matrix equation. The implicit adjoint operation in the $\mathrm{Re}$ function forces the $H_{nH}\rho$ expression to be in a pure basis. The same argument applies to $(J_m\rho)^\dagger$. Because terms of sums have to be in the same basis, also $J_m(J_m\rho)^\dagger$ and $\dot{\rho}$ have to be in a pure basis. Using the rule that density operators should only occur in one representation it follows that all occurrences of $\rho$ have to be in a pure basis representation. As consequence of these observations the operators $H_{nH}$ and $J_m$ have to be in a mixed basis which is determined by the representation of the density operator. This leaves only the choice in which pure basis the density operator should be represented. As long as the representations of the operators are chosen in a consistent way there is qualitatively no difference between the two possible matrix equations:

$$\dot{\rho}^{ij} = \frac{1}{\hbar} 2\,\mathrm{Re}\left(\frac{1}{i}\left(H_{nH}\right)^i_{\ k}\rho^{kj}\right) + \sum_m \left(J_m\right)^i_{\ k}\left(\left(J_m\right)^k_{\ l}\rho^{lj}\right)^\dagger$$

$$\dot{\rho}_{ij} = \frac{1}{\hbar} 2\,\mathrm{Re}\left(\frac{1}{i}\left(H_{nH}\right)_i^{\ k}\rho_{kj}\right) + \sum_m \left(J_m\right)_i^{\ k}\left(\left(J_m\right)_k^{\ l}\rho_{lj}\right)^\dagger$$

Luckily not only $\rho$, but also all other operators appear in only one representation! This means for this time evolution exactly the same algorithm can be used in the orthogonal and in the nonorthogonal case.

For stability reasons the algorithm implemented in C++QED preforms two further procedures which also have to be examined for the nonorthogonal case:

1. The density matrix is made hermitian.

   This is not a problem since hermitian operators correspond to hermitian matrices if they are represented in a pure basis.

2. The density matrix is renormalized.

   Here for the first time a numerical complication occurs. Since the trace of an operator only equals the trace of its matrix representation if it is taken in a mixed basis, it is necessary to perform a basis transformation before calculating the norm.

## 4.2 MCWF-method

This method was presented in the chapter *MCWF method* (chapter 3). It turned out that the method consists of basically two different kinds of time evolutions which now will be separately examined regarding nonorthogonal bases. For easiness of implementation it would be preferable if operators occurred in exactly the same representation as in the case of the master equation. Only then it is really possible to use the two algorithms interchangeably.

### 4.2.1 Non-unitary evolution

With the previously defined non-hermitian Hamiltonian $H_{nH}$ this time evolution is according to a Schroedinger equation and can be written as:

$$|\dot{\Psi}\rangle = \frac{1}{i\hbar} H_{nH} |\Psi\rangle$$

By requiring again that the state vector should only occur in one representation and by using the rule that multiplications have to be done between upper and lower indices, only two possibilities remain:

$$\dot{\Psi}^i = \frac{1}{i\hbar} \sum_j \left(H_{nH}\right)^i_{\ j} \Psi^j$$

$$\dot{\Psi}_i = \frac{1}{i\hbar} \sum_j \left(H_{nH}\right)_i^{\ j} \Psi_j$$

Here $H_{nH}$ has to be calculated in a mixed form.

### 4.2.2 Quantum jump

Now two different evolutions are possible. When no jump occurs, the state vector simply has to be renormalized. The norm of a state vector in a nonorthogonal basis is given by

$$\| |\Psi\rangle \|^2 = \Psi^\bullet * \Psi^*_\bullet$$

and has to be calculated differently than in the orthogonal case. The second possible time-evolution is described by quantum jumps:

$$|\Psi(t+\delta t)\rangle = \frac{J_m |\Psi(t)\rangle}{\| J_m |\Psi(t)\rangle \|}$$

The probability that a specific quantum jump occurs in a time interval $\delta t$ is in first order given by

$$\delta p_m = \delta t \langle \Psi | J_m^\dagger J_m | \Psi \rangle = \delta t \, \| J_m | \Psi \rangle \|^2$$

All operations occurring in this time evolution have already been treated before. The term containing $\Phi = J_m|\Psi\rangle$ has to be calculated with one of the following two representations:

$$\Phi^i = \sum_j (J_m)^i{}_j \Psi^j$$
$$\Phi_i = \sum_j (J_m)_i{}^j \Psi_j$$

The calculations of norms have again to be treated in a special way.

## 4.3 Conclusion

To perform a time evolution in a nonorthogonal basis the same algorithms as for the orthogonal case can be used, as long as the following points are heeded:

- Density operators are represented in a pure basis.
- All other operators are represented in a mixed basis.
- The norm has to be calculated properly.

CHAPTER

FIVE

# COHERENT STATE BASES

Numerical simulations can only be done if the number of dimensions of a system is small enough. For high dimensional or even infinite dimensional systems a subspace has to be found which has a reasonable amount of basis vectors but still can be used to approximate the actual physical problem sufficiently well. A good example for this are simulations done in an energy eigenbasis. Only states with a photon number smaller than some arbitrarily chosen limit are used. Cutting off all the higher number states doesn't change the outcome of a simulation as long as these states are never occupied. Exactly the same principle applies to bases consisting of coherent states. However, deciding which states should be used to build up a suitable computational subspace gets a little more complicated since the coherent states can be chosen from a continuous set of states. A good coherent basis should be able to represent all coherent states in some region of the $\alpha$ space with an error as low as possible. This property makes the subspace appropriate for calculating problems which are localized inside this area. The question that will be treated in this chapter is how these basis states have to be chosen to guarantee a certain precision.

## 5.1 Quality of a coherent basis

How suitable a basis is to represent states in some area of the phase space can be measured by how good coherent states in that area can be represented. For this it is necessary to find their best possible approximations in that basis and then to calculate their error. This will be done first for a subspace given in a general nonorthogonal basis and then the special case for coherent bases will be treated.

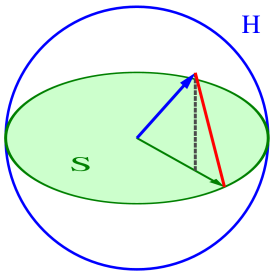
### 5.1.1 Best approximation of arbitrary states in subspaces given in nonorthogonal bases

In the following $|x\rangle$ will denote an arbitrary state in a Hilbert-space $\mathcal{H}$ and $\{|i\rangle_\bullet\}_{i=1}^N$ a nonorthogonal basis of a subspace $S$ of this Hilbert-space. The approximation of this state in the subspace can be written as

$$|\widetilde{x}\rangle = \sum_{i=1}^{N} c^i |i\rangle_\bullet$$

Both, $|x\rangle$ as well as $|\widetilde{x}\rangle$ are assumed to be normalized which imposes some constraints onto the possible values of $c^i$.

The question is now what is the best choice of $c_i$? It seems reasonable to take the state in the subspace with the smallest distance to the approximated state. From geometrical considerations it is then clear that this approximation can be achieved by projecting the state onto the subspace and subsequently normalizing it.




This offers a simple way to calculate the coordinates $c_i$. The state $|x\rangle$ minus its projection onto the subspace is perpendicular to all states in the subspace. This can be expressed mathematically as:

$$|x\rangle - P_{\mathcal{S}}\{x\} \perp \mathcal{S}$$

Since the projection is parallel to $|\widetilde{x}\rangle$ it can also be written as $\mathcal{N} * |\widetilde{x}\rangle$. Keeping in mind that the subspace is not only generated by the basis $\{|i\rangle_\bullet\}_{i=1}^N$, but also by its dual $\{|i\rangle^\bullet\}_{i=1}^N$ it is clear that the following relation holds for all $y$:

$$0 = \sum_{j=1}^{N} (y^j)^* \left({}^\bullet\langle j|x\rangle - \mathcal{N} * {}^\bullet\langle j|\widetilde{x}\rangle\right) = \sum_{j=1}^{N} (y^j)^* \left({}^\bullet\langle j|x\rangle - N c^j\right)$$

This results in a simple and nice formula for calculating the coordinates of the approximated state:

$$c^j = \frac{{}^\bullet\langle j|x\rangle}{\mathcal{N}}$$
$$\mathcal{N} = \| \sum_{i=1}^{N} {}^\bullet\langle i|x\rangle |i\rangle_\bullet \|$$

### 5.1.2 Error of the approximation (general case)

The error with which an arbitrary state vector can be represented in a subspace can now easily be calculated:

$$\mathcal{E}(|x\rangle)^2 = \| |x\rangle - |\widetilde{x}\rangle \|^2 = \| |x\rangle \|^2 - \langle x|\widetilde{x}\rangle - \langle \widetilde{x}|x\rangle + \| |\widetilde{x}\rangle \|^2$$

As a short calculation shows the scalar products between $|x\rangle$ and $|\widetilde{x}\rangle$ are given by the normalization constant $\mathcal{N}$:

$$\langle x|\widetilde{x}\rangle = \sum_{i=1}^{N} c^i \langle x|i\rangle_\bullet = \sum_{i=1}^{N} \frac{{}^\bullet\langle i|x\rangle}{\mathcal{N}} \langle x|i\rangle_\bullet = \frac{\sum_{i=1}^{N} \langle x|i\rangle_\bullet {}^\bullet\langle i|x\rangle}{\mathcal{N}} = \mathcal{N}$$
$$\langle \widetilde{x}|x\rangle = (\langle x|\widetilde{x}\rangle)^* = \mathcal{N}^* = \mathcal{N}$$

Since the states themselves are normalized the total error is:

$$\mathcal{E}(|x\rangle) = \sqrt{2 * (1 - \mathcal{N})}$$

### 5.1.3 Simple upper bound of the error

When calculating the error, one has to calculate $\mathcal{N}$ and therefore the scalar product ${}^{\bullet}\langle i|x\rangle$ for all basis states $|i\rangle^{\bullet}$ of the subspace $\mathcal{S}$. For high dimensional $\mathcal{S}$ this might be a tedious task, symbolically as well as numerically. In this case the best we can hope for is an upper bound for the error.

One method to do this is by looking at a low dimensional subspace $\mathcal{S}'$ of $\mathcal{S}$ and calculating the error of the approximation in this subspace. With the observations that projecting can make the norm of a state vector only smaller,

$$\| P\left(|x\rangle\right) \| \leq \| |x\rangle \|$$

and that projecting first onto $\mathcal{S}$ and then onto $\mathcal{S}'$ is the same as directly projecting onto $\mathcal{S}'$,

$$P_{\mathcal{S}'}\left(P_{\mathcal{S}}\left(|x\rangle\right)\right) = P_{\mathcal{S}'}\left(|x\rangle\right)$$

it is clear that

$$\mathcal{N} \geq \mathcal{N}' \Rightarrow \mathcal{E} \leq \mathcal{E}'$$

and therefore the error in $\mathcal{S}'$ is an upper bound for the error in $\mathcal{S}$.

### 5.1.4 Error of the approximation (specialization for coherent basis)

Now the results of the previous section can be applied to coherent bases. The only thing that has to be done is to specialize the general scalar product to the coherent scalar product:

$$\langle \alpha|\beta\rangle = e^{-\frac{1}{2}\left(|\alpha|^2+|\beta|^2\right)+\alpha^*\beta}$$

The normalization constant is therefore given by:

$$\begin{aligned}\mathcal{N}^2 = \| \sum_{i=1}^{N} {}^{\bullet}\langle\alpha_i|\beta\rangle|\alpha_i\rangle_{\bullet}\|^2 &= \sum_{i,j=1}^{N} {}_{\bullet}\langle\alpha_j|\beta\rangle\delta^{ij}\langle\beta|\alpha_i\rangle_{\bullet} \\ &= \sum_{i,j=1}^{N} \delta^{ij} e^{-\frac{1}{2}\left(|\alpha_i|^2+|\alpha_j|^2+2*|\beta|^2\right)+\alpha_j^*\beta+\alpha_i\beta^*}\end{aligned}$$

The representational error can then be calculated by:

$$\mathcal{E}(|x\rangle) = \sqrt{2 * (1 - \mathcal{N})}$$

### 5.1.5 Invariance of the quality of a basis

An interesting fact is that shifting the coherent basis by a fixed vector in the $\alpha$ space does not change its quality. Of course the region in which coherent states can be well represented changes, but inside that region the representational error is exactly the same. This can be seen by translating all states and the test vector by the same amount and again calculating the resulting error.

$$\begin{aligned}\widetilde{\mathcal{N}}^2 &= \| \sum_{i=1}^{N} {}^{\bullet}\langle \alpha_i + \gamma | \beta + \gamma \rangle | \alpha_i + \gamma \rangle_{\bullet} \|^2 \\ &= \sum_{i,j=1}^{N} \delta^{ij} e^{-\frac{1}{2}\left(|\alpha_i+\gamma|^2+|\alpha_j+\gamma|^2+2*|\beta+\gamma|^2\right)+(\alpha_j+\gamma)^*(\beta+\gamma)+(\alpha_i+\gamma)(\beta+\gamma)^*} \\ &= \sum_{i,j=1}^{N} \delta^{ij} e^{-\frac{1}{2}\left(|\alpha_i|^2+|\alpha_j|^2+2*|\beta|^2\right)+\alpha_j^*\beta+\alpha_i\beta^*} = \mathcal{N}^2\end{aligned}$$

Not only translations leave the quality of the basis unchanged, it also is invariant under rotations:

$$\begin{aligned}\widetilde{\mathcal{N}}^2 &= \| \sum_{i=1}^{N} {}^{\bullet}\langle \alpha_i e^{i\phi} | \beta e^{i\phi} \rangle | \alpha_i e^{i\phi} \rangle_{\bullet} \|^2 \\ &= \sum_{i,j=1}^{N} \delta^{ij} e^{-\frac{1}{2}\left(\left|\alpha_i e^{i\phi}\right|^2+\left|\alpha_j e^{i\phi}\right|^2+2*\left|\beta e^{i\phi}\right|^2\right)+\left(\alpha_j e^{i\phi}\right)^*\beta e^{i\phi}+\alpha_i e^{i\phi}\left(\beta e^{i\phi}\right)^*} \\ &= \sum_{i,j=1}^{N} \delta^{ij} e^{-\frac{1}{2}\left(|\alpha_i|^2+|\alpha_j|^2+2*|\beta|^2\right)+\alpha_j^*\beta+\alpha_i\beta^*} = \mathcal{N}^2\end{aligned}$$

The conclusion is that it doesn't matter where in the $\alpha$ space the simulation is done, the achievable precision only depends on the geometry of the basis and the relative position of the test state.

## 5.2 The optimal coherent basis

Although it is now possible to calculate the representational error for any coherent state in any subspace, the reverse problem, choosing the best subspace which has a small error for coherent states in some region in the complex plane is much harder to solve. This problem is narrowed by the fact that to be useful for adaptive problems, only easily translatable bases (in the sense that removing states from one side and adding them at the other side results again in an acceptable bases) have to be considered. The reason for this is investigated in the chapter *Adaptive bases* (chapter 7), but the consequence is that the coherent basis states have to be chosen from grid points on some virtual lattice in the complex plane. Although this might not give the best choice of basis states for all physical problems, it has the advantage to be very general. Fine tuning the subspace, e.g. choosing more basis vectors in regions of high occupation and fewer in regions of low occupation might lead to a better overall precision but to do so requires good understanding of the systems behavior even before the simulations are done. Besides, it is not

to be expected that the improvements are high enough to justify the additional work. Limiting the possible basis states to points on a lattice greatly reduces the amount of parameters needed to describe the subspace. The question is if some lattice types are superior to others. It turns out that hexagonal grids are especially convenient for the later presented adaptive algorithm and numerical tests show they are suitable to represent coherent states in the area covered by them.

## 5.3 Sample subspaces

To get a better feeling how good coherent states in actual computations can be represented, some sample bases are analyzed in more detail.

### 5.3.1 1 basis state

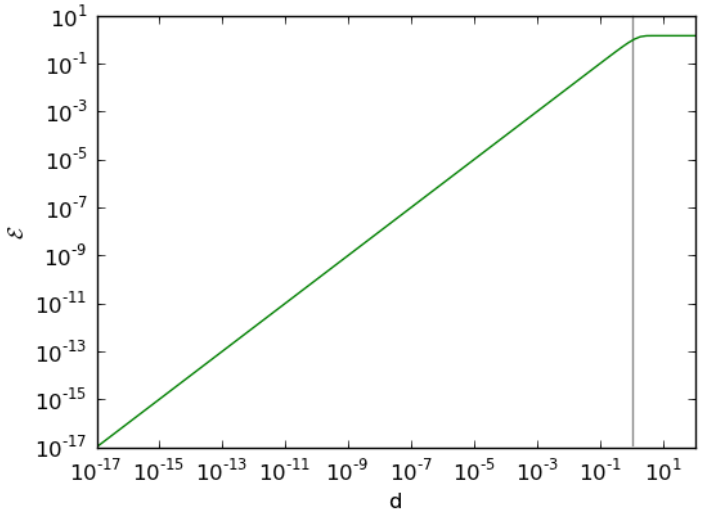
As most simple example, a basis consisting of only one single state vector is examined. How good can other coherent states be represented in this admittedly suboptimal basis? The answer is given by the scalar product for coherent states which reveals that the error decreases exponentially with the distance between test state and basis state in the complex plane. The norm is simply given as

$$\mathcal{N}^2 = e^{-|d|^2}$$

and therefore the error is:

$$\mathcal{E} = \sqrt{2} * \sqrt{1 - e^{-\frac{1}{2}|d|^2}}$$

Plotting this error results in:




### 5.3.2 2 basis states

The next logical step is to take two basis states, which means there are essentially three parameters describing the system. However, the position of the test state is chosen to be exactly

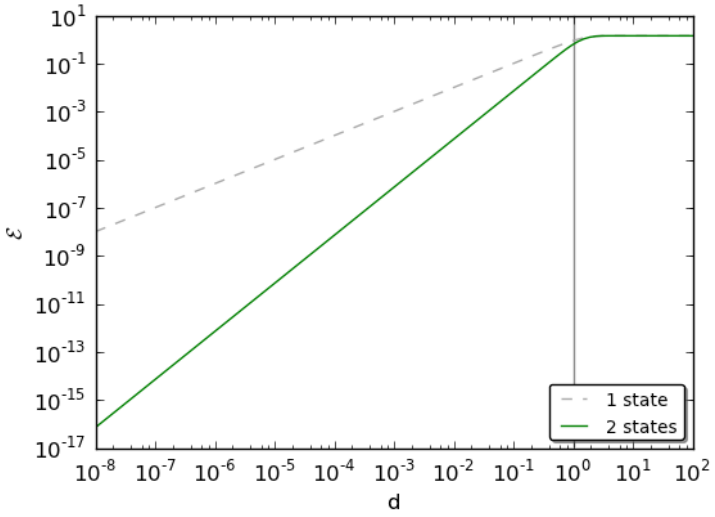
between the two basis states. This leaves only one parameter, the distance between the basis states which is denoted by $d$.


Calculations become more difficult going to higher dimensions, but for this case the norm and the error can still be given as a readily comprehensible formula:

$$\mathcal{N}^2 = \frac{1}{\cosh\left(|d|^2\right)}$$

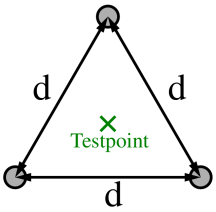
$$\mathcal{E} = \sqrt{2} * \sqrt{1 - \sqrt{\frac{1}{\cosh\left(|d|^2\right)}}} \rightarrow \frac{|d|^2}{4\sqrt{2}}$$

Plotting the error with the 1D error for comparison shows a high improvement going from one to two dimensions:




### 5.3.3 3 basis states

Now equilateral triangles are analyzed, which are of greater importance since they are the building blocks of hexagonal lattices. The test state is chosen in the center of mass, leaving again only one free parameter - the lattice constant $d$.

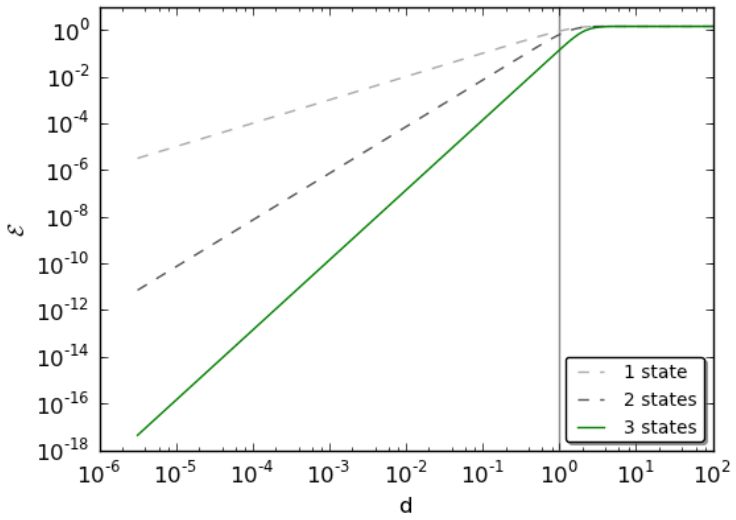



The norm and error can be calculated analytically:

$$\mathcal{N}^2 = \frac{3}{e^{|d|^2} + e^{-\frac{1}{2}\left(1+i\sqrt{3}\right)|d|^2} + e^{-\frac{1}{2}\left(1-i\sqrt{3}\right)|d|^2}}$$

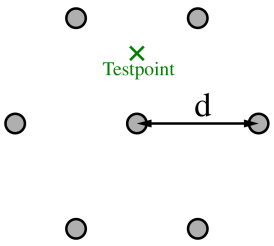
$$\mathcal{E} = \sqrt{2} * \sqrt{1 - \sqrt{\frac{3}{e^{|d|^2} + e^{-\frac{1}{2}\left(1+i\sqrt{3}\right)|d|^2} + e^{-\frac{1}{2}\left(1-i\sqrt{3}\right)|d|^2}}}} \rightarrow \frac{|d|^3}{8\sqrt{6}}$$

Plotting reveals an even bigger improvement over the 1D and 2D cases.

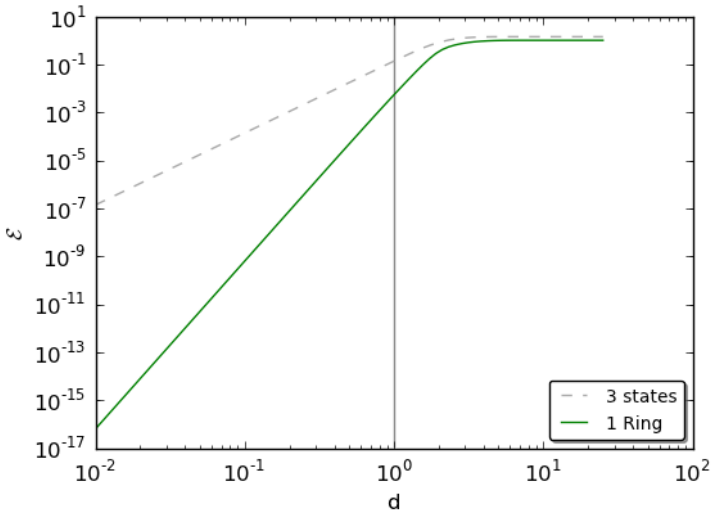



### 5.3.4 1 ring

In the following hexagonal grids will be analyzed, taking more and more basis states. Because of the higher number of dimensions it is easier to do the calculations of the representational error numerically. For this the multiple precision python library [mpmath] was used. First, one central state plus all surrounding states (which form one single ring) are chosen. The geometry and the position of the test state is made clear by this sketch:

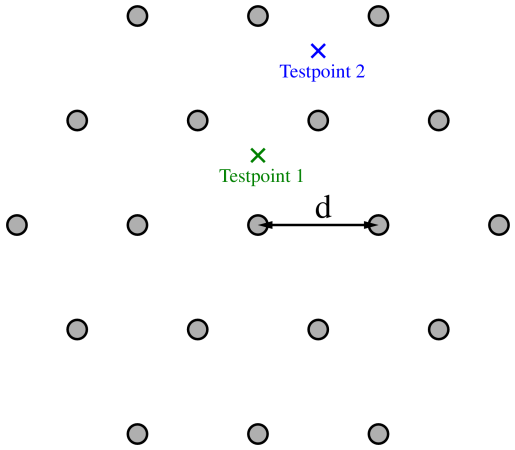



The calculation of the error results in the following plot, again revealing a great improvement compared to the equilateral triangle case.

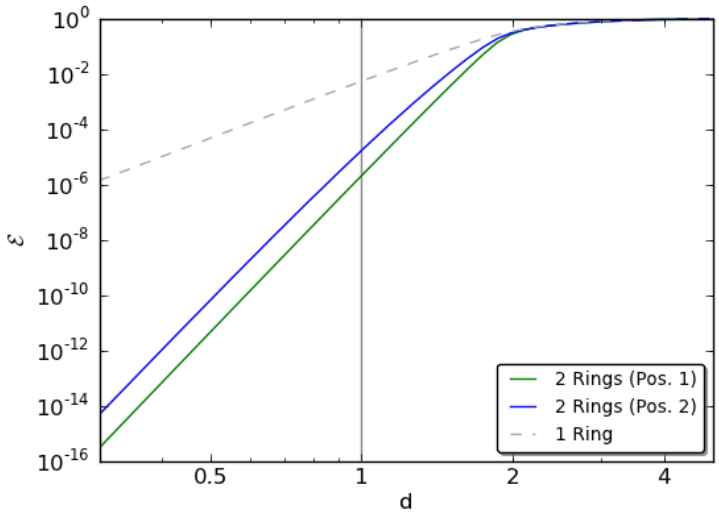



### 5.3.5 2 rings

For the next sample a central state plus two rings are chosen. Test states are taken at two different locations as is illustrated by the following sketch:

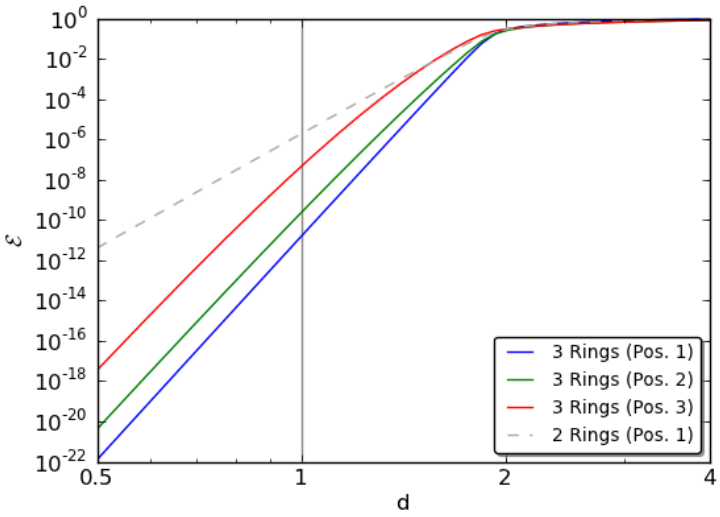



With this basis finally a size is reached that can be used to do actual simulations. The error plot looks like:




### 5.3.6 3 rings

As a last example a central state plus three rings is chosen. The test states are exactly the same as in the two-ring case with one additional test state in the outermost ring.



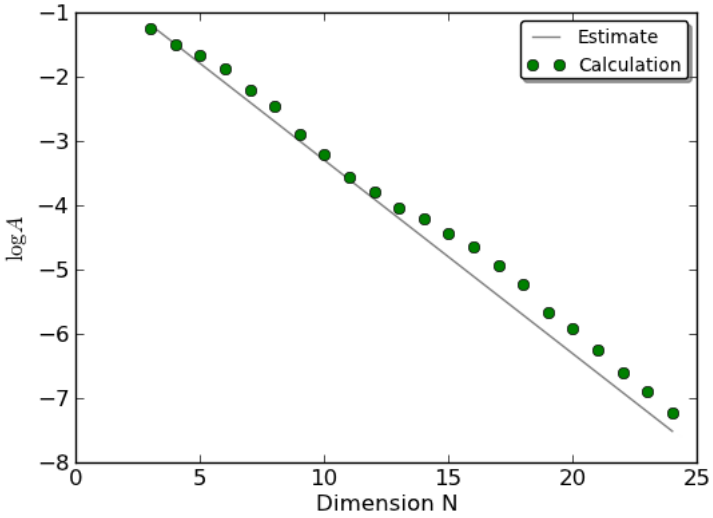

## 5.4 Estimate for the precision

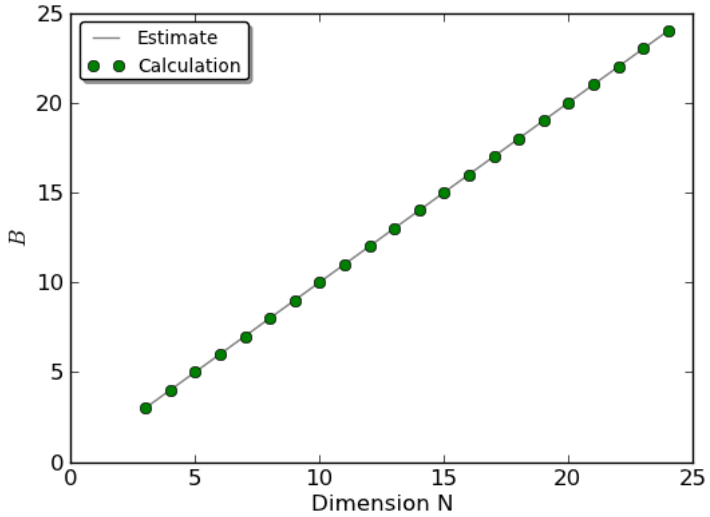
Comparing the different error plots of the previously examined basis samples shows that all of them decrease exponentially as soon as the lattice constant gets below some threshold. This motivates to try following ansatz for the error in case of small lattice constants:

$$\mathcal{E} = A * d^{B}$$

The variables $A$ and $B$ will highly depend on which and how many basis states are chosen but they should be mostly independent of the lattice constant. For further discussion it will be assumed that the basis states are clustered around some test state on the grid points of a hexagonal lattice. With this restriction it is possible to parameterize the geometry of the basis

by only the number of dimensions. Calculating the variables $A$ and $B$ for different dimension numbers reveals a beautiful correlation:




Obviously these two variables are very good estimated by linear functions depending on the dimension number! For $B$ the proportional constant is 1 and therefore it can simply be approximated by $N$:

$$B \approx N$$

For $A$ the correlation can be approximated by:

$$A \approx (0.54)^{N+1}$$

Therefore a good estimate for the representational error of a coherent basis is given by:

$$\mathcal{E} \approx \left(\frac{1}{2}\right)^{N+1} d^N$$

This estimate is very helpful to find an appropriate lattice constant for hexagonal lattices of specific size.

## 5.5 Operator representations in coherent bases

In the former sections it has been shown how coherent bases have to be chosen. It still is necessary to find out the matrix representations of often used operators in such bases. In the chapter *Time evolution in nonorthogonal bases* (chapter 4) it was found that it is preferable to take operators in mixed bases. In later preformed simulations only operators in an up-down representation are used (like $A^i_{\ j}$) so the focus will be on them. The destruction operator is then diagonal:

$$a^i_{\ j} = {}^{\bullet}\langle\alpha_i|a|\alpha_j\rangle_{\bullet} = \alpha_j\delta^i_{\ j}$$
$$(a^n)^i_{\ j} = {}^{\bullet}\langle\alpha_i|a^n|\alpha_j\rangle_{\bullet} = \alpha_j^n\delta^i_{\ j}$$

Whereas the creation operator has not such a nice form since both the transformation matrix as well as the inverse transformation matrix has to be used for its calculation:

$$\left(a^\dagger\right)^i_{\ j} = {}^{\bullet}\langle\alpha_i|a^\dagger|\alpha_j\rangle_{\bullet} = \sum_k {}^{\bullet}\langle\alpha_i|\alpha_k\rangle^{\bullet}{}_{\bullet}\langle\alpha_k|a^\dagger|\alpha_j\rangle_{\bullet} = \sum_k \delta^{ik}\alpha_k^*\delta_{kj}$$
$$\left(\left(a^\dagger\right)^n\right)^i_{\ j} = {}^{\bullet}\langle\alpha_i|\left(a^\dagger\right)^n|\alpha_j\rangle_{\bullet} = \sum_k \delta^{ik}\left(\alpha_k^*\right)^n\delta_{kj}$$

The same applies to general normally ordered operators:

$$\left(a^{\dagger m}a^n\right)^i_{\ j} = \sum_k \delta^{ik}\left(\alpha_k^*\right)^m\left(\alpha_j\right)^n\delta_{kj}$$

CHAPTER

SIX

# CONDITION NUMBERS OF PROBLEMS IN COHERENT BASES

As has been argued before, for nonorthogonal bases the transformation matrix $\delta_{\bullet\bullet}$ and, at least for the calculation of operators, its inverse $\delta^{\bullet\bullet}$ are needed (*Nonorthogonal bases* (chapter 2)). Unfortunately these matrices tend to be ill conditioned in practical examples. It shows that the smaller the phase-space distance between the basis states is, the worse conditioned the problem gets. Thus, the lattice constant has to be chosen very carefully. If chosen too small the condition number $\kappa$ is exploding and makes the calculation of the inverse transformation matrix impossible. On the other hand, if chosen too big, the representation of coherent states in this basis gets imprecise. The consequence of high condition numbers can be best understood by looking at the resulting total precision of simulations. Usually calculations are done in double precision which means the machine precision $eps$ is in the order of magnitude of $10^{-16}$. The total precision can then be calculated as $\kappa * eps$. In the following different basis geometries are analyzed for their condition number. For low dimensional bases this can be done analytically but for more than three basis states this gets difficult and it is easier to do it numerically.

## 6.1 Analytic calculations of condition numbers

The condition number of a normal matrix can be obtained by dividing its biggest eigenvalue by its smallest eigenvalue:

$$\kappa = \left| \frac{\lambda_{max}(A)}{\lambda_{min}(A)} \right|$$

Since transformation matrices are hermitian and because hermitian matrices are normal, this formula can be used to calculate condition numbers.

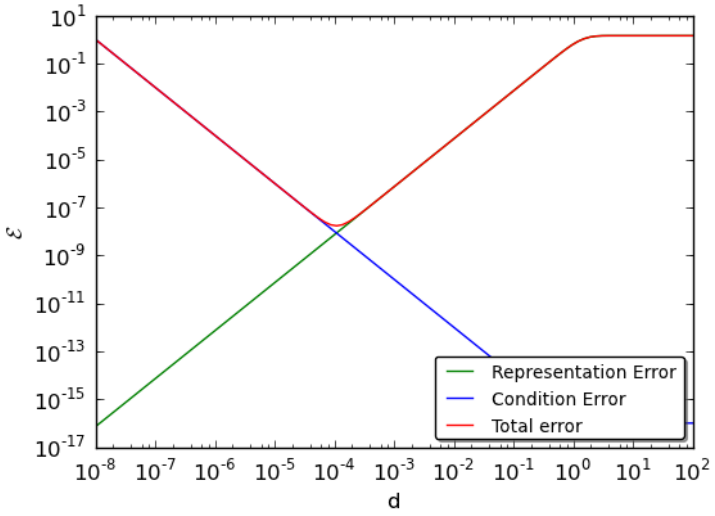
### 6.1.1 2 basis states

The simplest, non-trivial case is a basis consisting of two coherent states. Because of translational and rotational invariance the only parameter is the distance between the two states in the complex plane. Calculating the eigenvalues is straightforward and leads to the condition

number

$$\kappa = \coth\left(|d|^2\right) \to \frac{1}{|d|^2}$$

Comparing the error caused by the condition number and the error caused by the incompleteness of the coherent basis shows nicely the big impact of the lattice constant onto the total error of simulations.



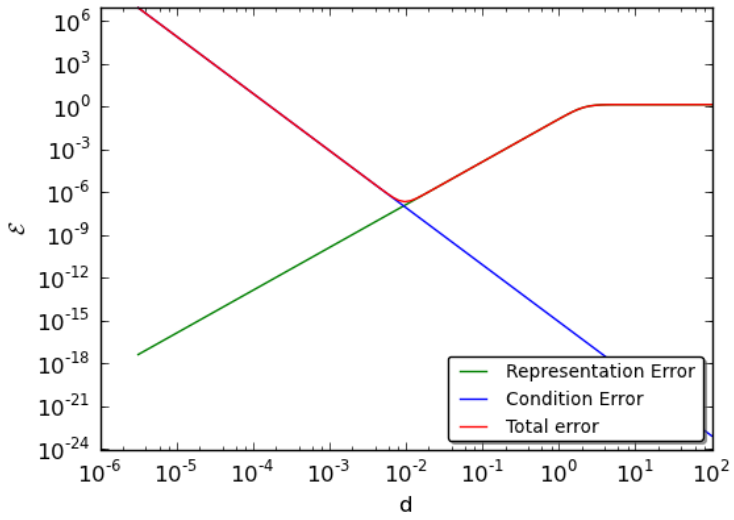

According to this, the best precision that can be achieved when calculating with doubles is around $10^{-8}$ at $d \approx 10^{-4}$.

### 6.1.2 3 basis states

The next complex case, and also the last that will be calculated analytically is a three dimensional basis where the states are placed on the edges of an equilateral triangle. The condition number can be calculated in the same way as before but the resulting expression is quite complex. However, for small distances $d$ the expression simplifies considerably:

$$\kappa \to \frac{2}{|d|^4}$$

The best precision that can be achieved is not as good as in the 2D case and is around $10^{-7}$. Also the distance has to be chosen much higher to reach optimal precision.




## 6.2 Numerical calculation of condition numbers

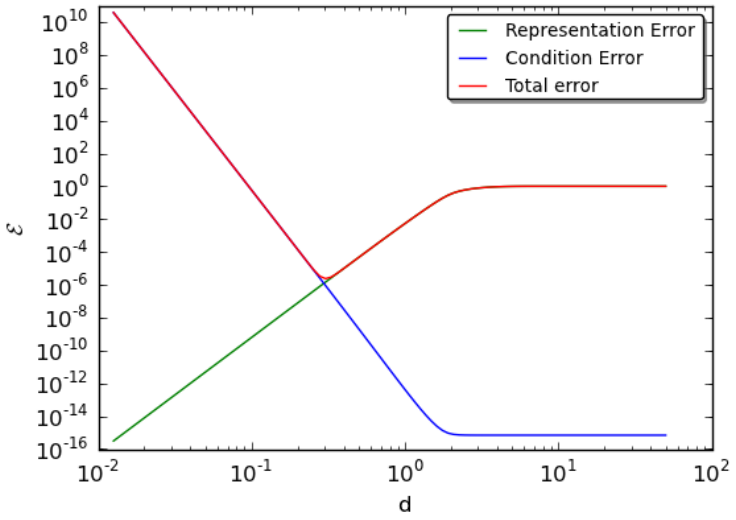
For coherent bases with more than three dimensions it is much easier to calculate the condition number numerically. One possible way to do this is to use directly the definition:

$$\kappa = \|A\| * \|A^{-1}\|$$

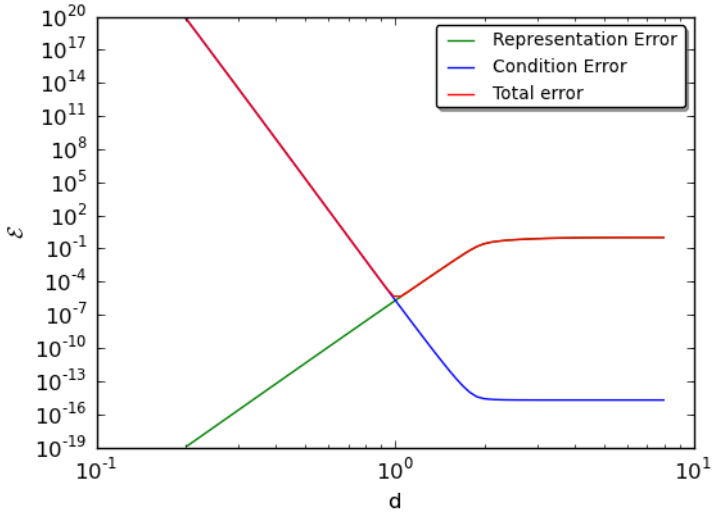
This means both, the transformation matrix and the inverse transformation matrix have to be calculated and then their norms have to be multiplied. Of course, this cannot be done in double precision because if the condition number gets too high it is impossible to calculate the inverse transformation matrix. The brute force solution is to use a high enough precision. These calculations were done with mpmath [mpmath], a python library which makes it very convenient to work with arbitrary precision. Although this approach is rather crude, it is easy and fast to implement and absolutely sufficient as long as the dimension numbers don't get too high.

To get more familiar with the behaviour of the condition number for bases that are used in simulations, examples that were previously studied for their ability to represent coherent states are now examined regarding their condition number.

- One ring:




- Two rings:




Three points can be observed from these graphs, that seem to hold for various bases - at least as long as the basis states are chosen on the grid points of hexagonal lattices:

1. The best precision that can be reached with doubles is around $10^{-5}$ but gets slowly lower for higher dimensions.
2. The higher the dimension of the basis the bigger the lattice constant has to be chosen for optimal precision.
3. The condition number and the lattice constant are approximately connected by $\kappa \propto d^B$ if the lattice constant is chosen small enough.

The first point is rather unpleasant since it limits all double precision calculations to less than 5 digits. And this is only true if optimal basis geometries are used and the problem is very localized in the center. However, for adaptive bases it would be favourable if the geometry was more flexible. All in all this means that double precision simply is not sufficient for our purposes. The calculation of at least parts of the simulation have to be done with higher precision. This is not as bad as it sounds first because the inversion has to be done only when the basis changes. How and when this is done will be described later, important at this point is

that the computational expensive calculation of the inverse matrix has to be done only rarely. This makes it affordable to perform the calculation of the inverse matrix at high precision while using double precision the rest of the time. A way to estimate the necessary precision for every coherent basis in hexagonal geometry will be derived in the next section.

## 6.3 Estimate for the condition number

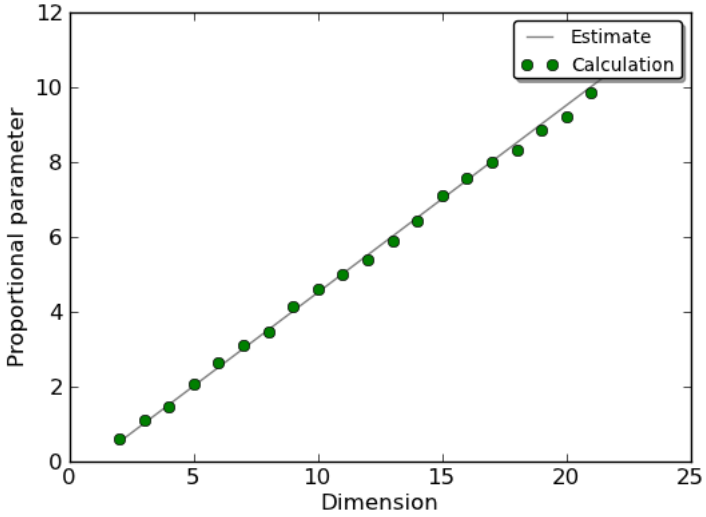
In the previous section the examination of different basis geometries indicated that the condition number and the lattice constant are approximately connected by:

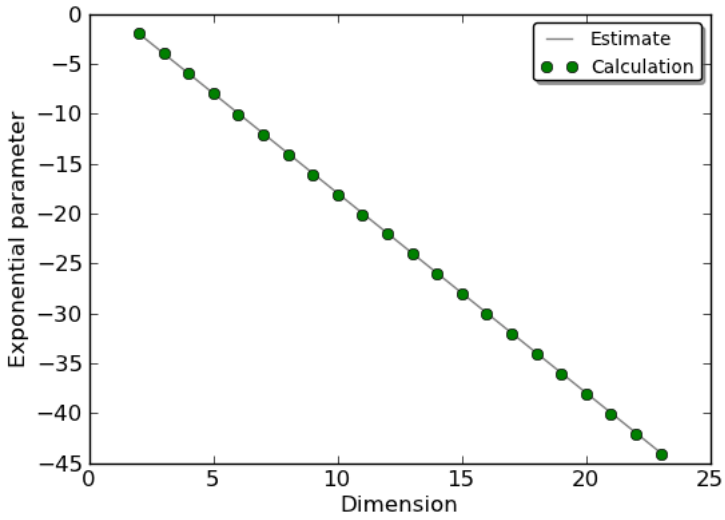
$$\kappa \approx A * d^B$$

The exact values of $A$ and $B$ are dependent on which state vectors are chosen as basis but they are only very weakly dependent on the lattice constant. The aim is now to determine a good estimate for these variables that can be used as upper limit for all hexagonal geometries. It sounds reasonable to assume that the denser the basis states are clustered the bigger the conditional number gets. Therefore it is enough to consider the densest geometry for every dimensional number and find functions for the parameters $A$ and $B$ that are only dependent on the dimension of the used basis. Choosing the densest basis is achieved by starting with one state vector on an arbitrary lattice point and gradually adding more state vectors going along rings around the first basis state. For each dimension number the condition number is calculated for different lattice constants. These values are used to determine optimal parameters $A$ and $B$ in the sense of regression analysis. Doing this reveals very nice correlations between the parameters $A$, $B$ and the number of dimensions:

$$\log_{10} A(N) = \frac{1}{2} * (N-1)$$
$$B(N) = -2 * (N-1)$$

The following plots reveal how good this estimate is:






Finally, plugging the numerically obtained parameters into the formula results in:

$$\kappa \approx 10^{\frac{1}{2}*(N-1)} * d^{-2*(N-1)}$$

This formula can be used in simulations to estimate the precision needed.

## 6.4 Obtaining higher precision

The high condition numbers clearly harm calculations that are done purely in double precision. Fortunately it turns out that it is not necessary to perform all parts of the simulations in higher precision. The inverse transformation matrix, which seems to cause most problems, is only used for calculating the mixed basis representation of operators. This means it is enough to use high precision for these calculations, round the resulting operators to double precision and perform simulations with these operators.

CHAPTER

SEVEN

# ADAPTIVE BASES

In the chapter *Coherent state bases* (chapter 5) it has been shown that coherent bases are suitable for problems that are well localized in the complex $\alpha$ space. However, if the area of localization changes with time, a fixed basis quickly gets useless as the state vector moves into regions that cannot be represented by the initially chosen basis. To still be able to use the same method for such problems, the basis has to follow the movement of the state vector. The computational most favourable solution is to chose a suitable basis for the initial state and hold it fixed for some time. During this period the problem is evolved exactly as it has been described in chapter *Time evolution in nonorthogonal bases* (chapter 4). The necessity of moving the basis is checked after every time step. How the movement is done and how to determine if it is necessary will be treated in this chapter.

## 7.1 Advantages of lattices

Limiting the possible basis states onto points on a lattice has different advantages. As has been seen before, it makes it possible to determine upper bounds for the representation error (*Coherent state bases* (chapter 5)) and lower bounds for the condition number (*Condition numbers of problems in coherent bases* (chapter 6)). Perhaps even more important is the translational invariance of lattices. This allows moving the basis by removing states from one side and adding them on the other side. One advantage of this is that state vectors can be better converted to the new basis if only a few basis states change. Additionally, when operators are calculated in respect to the new basis some previously done calculations can be reused. However, the main reason for using only points on a lattice is that it is easier to determine which basis is optimal for the momentary state of the system. By assigning weight factors to every lattice point, which measure how important the corresponding coherent state is for the correctness of the whole simulation, the best states can be picked to form a basis for the next time step. In the following an adequate weight function will be defined and motivated.

## 7.2 Importance of single basis states

The weight function measures how important a basis state is for the correctness of the result of the whole simulation. For orthogonal bases it is intuitively clear that the less occupied a state is the less important it is for the result. In the nonorthogonal case this statement is not directly

applicable since it is not clear how "occupation" has to be defined. Therefor it is necessary to find some other approach: If a basis state can be taken out of the basis and the remaining basis states are still able to sufficiently represent the current state vector, this basis state simply is not important. So the importance of at least the coherent states that are currently forming the basis can be measured by the error that would be introduced if one of them is removed. Mathematically this can be expressed in the following way. Assuming that the system is in some state $\Psi$ which is represented in the basis:

$$\underline{b_\bullet} = \{|i\rangle_\bullet\}_{i\in B}$$

$|\Psi\rangle$ can be written as:

$$|\Psi\rangle = \sum_{i\in B} {}^\bullet\langle i|\Psi\rangle|i\rangle_\bullet$$

Removing one state vector from this basis, denoted as $|K\rangle_\bullet$, the remaining states form a basis for a subspace:

$$\underline{s_\bullet} = \{|i\rangle_\bullet\}_{i\in S}$$

Although this two sets of state vectors nearly are the same, the corresponding dual bases are not. Between the dual states of $\underline{s_\bullet}$, which will be written as $\underline{s^\bullet} = \{|i\rangle^\times\}_{i\in S}$, and the states of the basis $\underline{b_\bullet}$ the usual orthogonality relation doesn't hold anymore. Instead the following relation takes its place:

$${}^\times\langle s|b\rangle_\bullet = \delta^s_b + {}^\times\langle s|K\rangle_\bullet \delta^K_b$$

The projection of the state vector $|\Psi\rangle$ onto the subspace spanned up by $\underline{s_\bullet}$ can now be easily calculated:

$$\begin{aligned}
\sum_{i\in S} |i\rangle_\bullet {}^\times\langle i|\Psi\rangle &= \sum_{i\in S}\sum_{j\in B} |i\rangle_\bullet {}^\times\langle i|j\rangle_\bullet {}^\bullet\langle j|\Psi\rangle \\
&= \sum_{i\in S}\sum_{j\in B} |i\rangle_\bullet \left(\delta^i_j + {}^\times\langle i|K\rangle_\bullet \delta^K_j\right) \Psi^j \\
&= \sum_{i\in S} \Psi^i |i\rangle_\bullet + \sum_{i\in S} \Psi^K |i\rangle_\bullet {}^\times\langle i|K\rangle_\bullet
\end{aligned}$$

The resulting error is the difference of this projection and the original state:

$$\begin{aligned}
|E\rangle &= \sum_{i\in B} \Psi^i |i\rangle_\bullet - \sum_{i\in S} {}^\times\langle i|\Psi\rangle|i\rangle_\bullet \\
&= \Psi^K |K\rangle_\bullet - \sum_{i\in S} \Psi^K |i\rangle_\bullet {}^\times\langle i|K\rangle_\bullet \\
&= \Psi^K \left( |K\rangle_\bullet - \sum_{i\in S} |i\rangle_\bullet {}^\times\langle i|K\rangle_\bullet \right)
\end{aligned}$$

Calculating the norm leads to:

$$|\langle\mathcal{E}|\mathcal{E}\rangle| = |\Psi^k|^2 \left(1 - \sum_{i\in S} {}_\bullet\langle K|i\rangle_\bullet {}^\times\langle i|K\rangle_\bullet\right) = |\Psi^k|^2 \left(1 - \sum_{i\in S} P_i\{|K\rangle_\bullet\}\right)$$

whith the projection term $P_j\{|i\rangle\}$ defined as:

$$P_i\{|K\rangle\} = \sum_{i \in S} {}_\bullet\langle K|i\rangle_\bullet{}^\times \langle i|K\rangle_\bullet$$

This formula can be interpreted in the following way: The second term inside the parenthesis indicates how good one state can be represented by other basis states while the first factor, $c^i = |\Psi^i|^2$ measures the occupation of that state. Until now only the negative effect that one state can be represented by others was considered. However, this should also work the other way round. The better a state can represent other occupied states the more important it should be. This leads to the following definition of a weighting factor:

$$w(|K\rangle_\bullet) = c^K * \left(1 - \sum_{i \in S} P_i\{|K\rangle_\bullet\}\right) + \sum_{i \in S} c^i * P_i\{|K\rangle_\bullet\}$$

Notable is that this weighting function also assigns weights greater than zero to states that are not included in the current basis which is absolutely necessary for an adaptive algorithm.

## 7.3 The weighting algorithm

The actual implemented algorithm makes some major simplifications which greatly speeds up the calculations of the weights but are only valid for big lattice constants. Only then the projection onto the remaining basis states can be approximated by a sum of single projections onto the neighbor states:

$$\sum_{i \in S} P_i\{|K\rangle\} \approx \sum_{i \in <K>} |{}_\bullet\langle K|i\rangle_\bullet|^2 = \sum_{i \in <K>} e^{-|\alpha_K - \alpha_i|^2}$$

The exponential is constant for this sum and therefore has to be calculated only once. Unfortunately it turns out that for lattice constants used in later simulations this approximation is not always legit. However, numerical experiments show that the algorithm can be improved by calculating the occupation numbers as $c^i = |\Psi_i^* \Psi^i|$ instead of as $c^i = |\Psi^i|^2$. This result could not be proved theoretically but nevertheless works remarkable well.

With this simplifications the following algorithm is suitable for adapting the coherent basis over the time:

1. Iterate over all basis states and do:
   - Calculate the occupation of that state by taking the absolute square of the corresponding coordinate in the state vector.
   - Add this occupation times $D$ to all neighbor weights.
2. Sort the states according to their weights and take the first N as the new basis.

## 7.4 Adjusting state vector and operators to new basis

Of course, if the basis is moved, the state vector and also the operators have to be adapted accordingly. For adapting the state vector a basis transformation has to be performed from the

old basis $\underline{b_\bullet}$ to the new one $\underline{n_\bullet}$. The state vector in respect to the old basis is given as:

$$|\Psi\rangle \simeq \sum_i b^i |b_i\rangle_\bullet$$

Calculating the coordinates in respect to the new basis can be done by projecting this state onto the new dual basis:

$$n^j = {}^\bullet\langle n_j|\Psi\rangle \simeq \sum_i {}^\bullet\langle n_j|b_i\rangle_\bullet \, b_i = \sum_{i,k} {}^\bullet\langle n_j|n_k\rangle^\bullet_\bullet \langle n_k|b_i\rangle_\bullet \, b_i$$

At this point two different transformation matrices occur. First a transformation from the old basis to the new one and then a subsequent transformation to the new dual basis. It is important to note that the first transformation is between two different computational subspaces and therefore has an intrinsic error independent of the numerical precision. How big this error is depends mainly on two factors: How good an old basis state can be represented by the new basis and how occupied the old basis state was. This error can be reduced by choosing a smaller lattice constant and therefore making states better representable in new bases or increasing the dimension of the basis to reduce the occupation number of the outer states. Both approaches result in higher computational cost for simulations and thus have to be seen as trade-off between precision and runtime. The second transformation, from the new basis to the new dual basis, also has to be investigated carefully since it was show in the chapter *Condition numbers of problems in coherent bases* (chapter 6) that this matrix is for commonly used bases very ill conditioned. The high condition number affects not only the calculation of the matrix inverse but also amplifies errors in matrix multiplications. Since the state vector is only known with double precision this could turn out to be devastating for the adaptive algorithm. Fortunately it turns out that although each transformation on its own is ill conditioned their product is not. Thus, calculating both transformations with higher precision and multiplying them with the state vector circumvents the condition problem and the algorithm is saved.

CHAPTER

EIGHT

# COMPLEXITY OF ALGORITHM USING ADAPTIVE COHERENT BASES

In this chapter it will be investigated how the computational effort of calculations in coherent bases compares to calculations done in number bases. The big advantage of coherent bases is that their dimensionality can be much lower for problems which are close to a coherent state. For an average photon number

$$\langle n \rangle = |\alpha|^2$$

the variance in the number basis is

$$(\Delta n)^2 = |\alpha|^2$$

and therefore the amount of basis states, $N_{number}$, needed for calculations scales as $|\alpha|^2$. Contrary, if coherent states are used as computational basis the amount of needed basis states, $N_{coh}$, is independent of the number of photons. Instead the needed dimension is basically determined by how close the state of the system is to a coherent state. The amount of states needed to reach a certain precision was discussed extensively before (*Coherent state bases* (chapter 5)) and highly depends on the specific simulated problem. So for further analysis it is assumed that for simulations in number bases $N_{number}$ states are needed while for the same problem in coherent bases $N_{coh}$ states have to be used. The aim is to estimate which basis is preferable for what type of system.

The MCWF-method performs several matrix multiplications in every time step. In number bases these matrices have in many cases a special form, meaning only some diagonals are not zero and therefore this matrix multiplication scales linearly with the number of dimensions times the number of secondary diagonals which are not zero:

$$Cost_{number} = N_{number} * \alpha$$

For coherent bases these special forms won't occur for non-trivial systems and the cost of the matrix multiplication normally increases with the number of basis states squared. Additionally the cost of adaptive basis changes have to be considered which is approximately $N_{coh}^3$ due to the needed Cholesky decomposition of the transformation matrix. Assuming that the basis is changed after $n$ matrix multiplications the average cost is given by

$$Cost_{coh} = N_{coh}^2 + \frac{N_{coh}^3}{n}$$

Now let us try to answer when to use coherent bases and when to use number bases. In the case that the simulation is perfectly suited for number states and absolutely not suited for coherent states, the operators are diagonal in the number basis which means $\alpha = 1$, and the coherent basis has to be adapted in every step. The relative cost is then:

$$\frac{Cost^{min}_{number}}{Cost^{max}_{coh}} = \frac{N_{number}}{N^3_{coh}}$$

The other extreme is reached when operators in the number states also have a completely dense matrix representation and no adaptive steps are needed at all. Then the relative cost is:

$$\frac{Cost^{max}_{number}}{Cost^{min}_{coh}} = \left(\frac{N_{number}}{N_{coh}}\right)^2$$

These results can be interpreted as having three different regions. For the case $N_{number} < N_{coh}$ always the number basis is superior while if $N^3_{coh} < N_{number}$ always the coherent basis is better. If $N_{coh} < N_{number} < N^3_{coh}$ then it is necessary to look closer and also consider the exact shape of the Hamiltonian in the number basis and maybe try out how many basis changes are necessary in the case of coherent bases.

Until now the impact of adding additional quantum systems to form composed systems was not considered. The main difference lies in the interaction part of the Hamiltonian. For adaptive algorithms at least parts of it have to be recalculated every time the basis changes. How expensive this is depends completely on the exact type of Hamiltonian and therefore has to be considered for every problem separately.

# Part II

# Code Design

The previously introduced theoretical framework for simulations in adaptive coherent bases makes it necessary to extend C++QED in different ways. Following the spirit of C++QED the extensions are done as general and orthogonal as possible. For clarity it is summarized here what is needed:

1. Coherent bases on lattices
2. Adaptivity of coherent bases on lattices
3. High-precision calculations of operators in coherent bases
4. Elements using coherent bases

Further abstractions are made to allow for possible future extensions:

- Adaptive systems are of course also thinkable in other cases than coherent bases. Therefore it makes sense to implement support for adaptive basis in general and then specialize it for coherent bases.
- It might sometimes be beneficial to use coherent bases with basis states not on lattice points. On the other hand there might be cases where bases are needed that somehow are linked to lattices. Implementing these two concepts separately doesn't add much of complexness while at the same time it allows for future extensions.
- State vectors in nonorthogonal bases have some properties which are different to the orthogonal case. To account for this orthogonal state vectors are specialized to nonorthogonal state vectors. The same applies to density operators.

In the next part of this thesis an overview will be given at how this was done.

CHAPTER

NINE

# ADAPTIVE INTERFACE

In the following an interface for adaptive algorithms is presented and discussed. The concept is quite general since many systems can benefit from adaptivity, e.g. not only coherent bases but also number bases. A limitation of the current implementation is that the dimensionality can not change. Although this potentially could be useful it simply would require too many changes in the C++QED code to be feasible. Let us have a closer look what is necessary to perform an adaption of a basis. It turns out that the adapting procedure itself can be split into three different parts:

1. It has to be calculated if the basis has to change and what the new basis will be. If this basis is part of a complex system only the reduced state vector is needed for this step.
2. Only if the basis changes the operators and for nonorthogonal bases also the transformation matrix have to be recalculated. Knowledge of the new basis is enough to perform this step.
3. Finally the state vector has to be adapted. This is done by calculating the transformation matrix from the old to the new basis and applying it onto slices of the whole state vector.

These considerations lead to the requirement of three separate methods which implement the three different parts called `adapt()`, `adaptOperators()` and `adaptSV()`.

What additionally has to be considered is that when the user wants to output the state vector, also information about the basis in which this state vector is given has to be made available. It is clear that it is enough to print this information only when the basis has changed and a state vector should be printed. Taking this into account, the adaptive interface has a `display()` method and an intern variable `was_displayed_`. Every time the basis changes this variable is set to `true` and every time before a state vector is printed the `display()` method is called which checks if the state vector was already printed.

Finally, the interface that all adaptive quantum systems have to implement looks like:

```
template<int RANK>
class Adaptive: public quantumdata::Types<RANK>
{
public:
    Adaptive(): was_displayed_(false) {}

    typedef quantumdata::Types<RANK> Base;
    typedef typename Base::StateVectorLow StateVectorLow;
```

```
    virtual bool adapt(const StateVectorLow&) const = 0;
    virtual void adaptOperators() const = 0;
    virtual void adaptSV(StateVectorLow&) const = 0;

    virtual void display(std::ostream& os, int precision) const = 0;

protected:
    mutable bool was_displayed_;
};
```

One still unsolved problem is the time step management for adaptive bases. In the C++QED framework things happen on basically three different timescales. At smallest level the Schroedinger equation is solved by an ode-solver which needs certain step sizes to reach the desired precision. The next bigger time scale originates from the MCWF-method and is given by the limitation that the probability for a quantum jump has to be small. The last scale is imposed by the user, who has to choose how often expectation values should be calculated. Usually this happens on the lowest time scale. How often adaptive steps have to be done greatly depends on which system is simulated and is more or less independent of the other time scales. This raises the question of how to estimate when the next adaptive step has to happen. At the moment this problem isn't solved for C++QED and the user has to make sure that at no point in the simulation the basis was inadequate for the problem.

CHAPTER

TEN

# LATTICES

In the chapters *Coherent state bases* (chapter 5) and *Adaptive bases* (chapter 7) it was concluded that it makes sense to build up coherent bases by choosing states which lie on the grid points of a hexagonal lattice. This makes it necessary to implement support for lattices in C++QED. Since they also might be useful for other bases their implementation is very abstract but a fully specialized version for 2D hexagonal lattices also exists. General lattices can be defined by declaring one point $\underline{c}$ in a vector space as origin and providing independent vectors, $b = \left\{\underline{b_i}\right\}_i$, spanning up a unity cell. The lattice points are given by:

$$\underline{p} = \underline{c} + \sum_i a_i \underline{b_i} \qquad a_i \in Z$$

Therefore an implementation of a lattice has to somehow store the basis vectors and also the origin. The type of the basis vectors has to be given as template parameter because for 2D lattices it might be convenient to simply use complex numbers while for higher dimensional lattices some kind of vector might be necessary. The dimension of the lattice has to be specified by the second template parameter `RANK`. To obtain the states corresponding to some indices $\{a_i\}_i$, the methods `state()` and `states()` can be used.

Since the basis has to consist of a finite number of basis states there has to be some way to declare which lattice points should build up the basis. This is done with the `select()` method which takes one ore more indices and marks the corresponding points as selected.

Finally the interface for general lattices looks like:

```
template<typename BASIS_TYPE,int RANK>
class Lattice
{
public:
    typedef blitz::TinyVector<BASIS_TYPE,RANK> LatticeBasis;
    typedef blitz::TinyVector<int,RANK> Idx;
    typedef std::list<Idx> IdxList;

    typedef blitz::Array<BASIS_TYPE,1> States;

    Lattice(BASIS_TYPE center, LatticeBasis basis)
        : center_(center), basis_(basis), selected_() {}

    BASIS_TYPE state(const Idx& idx) const;
```

```
    States states(const IdxList& indices) const;
    States states() const {return states(selected());}

    unsigned size() const {return selected().size();}

    virtual IdxList neighbors(const Idx& idx) const = 0;

    void select(const Idx& idx);
    void select(const IdxList& indices);
    IdxList& selected() {return selected_;}
    const IdxList& selected() const {return selected_;}

    bool contains(const Idx& idx) const;
    int index(const Idx& idx) const;
    int replace(const Idx& idx_old, const Idx& idx_new);

protected:
    BASIS_TYPE center_;
    LatticeBasis basis_;
    IdxList selected_;
};
```

The specialization for 2D hexagonal lattices can serve as an example of how to implement an specific type of lattice:

```
class Hexagonal: public Lattice<dcomp,2>
{
public:
    typedef Lattice<dcomp,2> Base;
    typedef Base::LatticeBasis LatticeBasis;
    typedef Base::Idx Idx;
    typedef Base::IdxList IdxList;

    Hexagonal(dcomp center, double d);

    IdxList neighbors(const Idx& idx) const;
    void select_rings(unsigned rings);
    void select_nearest(unsigned N);
};
```

How this class is used is explained in the next chapter.

CHAPTER

ELEVEN

# COHERENT BASIS CLASS

In the chapter *Condition numbers of problems in coherent bases* (chapter 6) it was shown that because of the high condition number of the transformation matrix its inverse and also operators in mixed bases have to be calculated in higher precision. For these calculations the libraries [mpfr] and [mpc] are used, for real and complex arithmetic respectively, and for better integration in C++ the mpfr wrapper [mpfr-cpp] was also included. For the needed linear algebra operations the numerical library [blitz] is used as far as possible. Calculations of the inverses is done via Cholesky decomposition. Numerically it would be favourable to not calculate the inverse directly but to use the decomposition directly for further calculations but the more simple and straightforward code outweighs the small runtime disadvantage. Because the time-evolution is done in double precision it is necessary to store transformation matrices and also their inverses twice - in both double and high precision. This is done in the `CoherentBasis`

```
class CoherentBasis
{
public:
    typedef blitz::Array<dcomp,1> BasisStates;
    typedef TTD_CARRAY(1) Coordinates;

    typedef mp::CMatrix TrafoMatrix;
    typedef mp::MPMatrix MPTrafoMatrix;

    CoherentBasis(unsigned N, unsigned prec);
    CoherentBasis(BasisStates states, unsigned prec);

    template<typename LATTICE>
    CoherentBasis(const LATTICE& lattice, unsigned prec);

    BasisStates& states() {return states_;}
    const BasisStates& states() const {return states_;}

    const TrafoMatrix& trafo() const {return trafo_;}
    const TrafoMatrix& inv_trafo() const {return inv_trafo_;}
    const MPTrafoMatrix& mp_trafo() const {return mp_trafo_;}
    const MPTrafoMatrix& mp_inv_trafo() const {return mp_inv_trafo_;}

    unsigned prec() const {return prec_;}
    unsigned size() const {return states().size();}
```

```
    dcomp inner_product(const dcomp& a, const dcomp& b) const;
    mp::mpcomplex
        inner_product(const mp::mpcomplex& a, const mp::mpcomplex& b) const;

    void calculate_trafo();

    Coordinates coordinates(const dcomp& state) const;

private:
    BasisStates states_;
    TrafoMatrix trafo_;
    TrafoMatrix inv_trafo_;
    MPTrafoMatrix mp_trafo_;
    MPTrafoMatrix mp_inv_trafo_;
    unsigned prec_;
};
```

It also provides some additional functionality which should be clear from the method names. Important to note is that the constructor is taking any 2D lattice as argument. This allows to first create any arbitrary `Lattice`, select the points which should build up the basis and then use that lattice to generate a `CoherentBasis`. Alternatively it is possible to use any arbitrary combination of states instead of lattice points but this might not be used too often. The `calculate_trafo()` method is called automatically the first time the `CoherentBasis` is created but if the basis states are changed, as for example it is the case for adaptive coherent basis, this method has to be called explicitly.

CHAPTER

TWELVE

# ADAPTIVE COHERENT BASIS

Until now only static coherent bases on hexagonal lattices are covered. In the chapter *Adaptive bases* (chapter 7) the idea and algorithm for adaptive bases is described. To make it easy to try out different algorithms the strategy pattern is used for this implementation. A generic class `AdaptiveCoherentBasis` implements the *Adaptive interface* (chapter 9) and additionally provides some useful constructors.

```
class AdaptiveCoherentBasis: public structure::Adaptive<1>
{
public:
    typedef adaptive::AdaptiveCoherentBasisStrategy Strategy;
    typedef lattice::Lattice<dcomp,2> Lattice;
    typedef boost::shared_ptr<Strategy> StrategyPtr;
    typedef CoherentBasis::MPTrafoMatrix MPTrafoMatrix;
    typedef CoherentBasis::BasisStates BasisStates;

    AdaptiveCoherentBasis(CoherentBasis& b, Lattice& l);
    AdaptiveCoherentBasis(CoherentBasis& b, Lattice& l, StrategyPtr s);

    virtual bool adapt(const StateVectorLow& sv) const;
    virtual void adaptOperators() const = 0;
    virtual void adaptSV(StateVectorLow& sv) const;

    void display(std::ostream& os, int precision) const;

private:
    mutable CoherentBasis& basis_;
    mutable Lattice& lattice_;
    StrategyPtr strategy_;
    mutable Strategy::Substitutions subs_;

    mutable MPTrafoMatrix mp_adapt_trafo_;
    mutable BasisStates states_;
};
```

The adaptive algorithm itself is implemented separately and has to be provided as argument to the constructor. All adaptive algorithms have in turn to inherit the `AdaptiveCoherentBasisStrategy` interface which basically defines only one method `adapt()` which takes a coherent basis and the current state vector as arguments and returns a

list of substitutions which map old not longer required basis states to the new more important states.

```
class AdaptiveCoherentBasisStrategy
{
public:
    typedef lattice::Lattice<dcomp,2> Lattice;

    typedef typename Lattice::Idx Idx;
    typedef typename Lattice::IdxList IdxList;

    typedef lattice::details::Hash Hash;
    typedef lattice::details::Equal Equal;

    typedef CoherentBasis::Coordinates Coordinates;


    typedef boost::unordered_map<Idx,Idx,Hash,Equal> Substitutions;

    virtual Substitutions adapt(const Lattice&, const CoherentBasis& basis,
                                const Coordinates&) const = 0;
};
```

The previously described adaptive algorithm is also implemented using this interface but the code will not be shown at this point.

CHAPTER

THIRTEEN

# NONORTHOGONAL STATE VECTOR

In the chapter *Nonorthogonal bases* (chapter 2) the differences between orthogonal and nonorthogonal state vectors were investigated in detail. It turned out that dual vectors are needed to calculate norms as well as expectation values. In order to obtain the dual vectors only the transformation matrix is needed, which means if we take the already existing implementation of orthogonal state vectors and add the transformation matrix all information needed for further calculations is available. Of course some methods have to be adapted and the mixing of nonorthogonal and orthogonal state vectors has to be taken care of. For this reason it makes sense to implement nonorthogonal state vectors as subclass of orthogonal state vectors and analogous nonorthogonal density matrices as subclass of orthogonal density matrices. Another big advantage of this design is that the current time evolution code in C++QED has to be changed only at very few points which lowers the possibility of introducing bugs. The declaration of `NonOrthogonalStateVector` therefore looks like:

```
template<int RANK, typename TRAFO>
class NonOrthogonalStateVector:
    public StateVector<RANK>,
    private linalg::VectorSpace<NonOrthogonalStateVector<RANK, TRAFO> >;
```

In comparison to `StateVector` this class got another template argument `TRAFO` which specifies the type of the transformation. This was done to make the creation of nonorthogonal state vectors as easy and at the same time as general as possible. Currently transformations can be blitz matrices of the right dimensionality and general functions with signature:

```
void(*)(const TTD_CARRAY(RANK)& in, TTD_CARRAY(RANK)& out)
```

Additionally the `Identity` class can be used to represent transformations for orthogonal state vectors. This is essential for composed quantum systems where some subsystems might have different transformations and especially also where systems in nonorthogonal bases are mixed with systems given in orthogonal ones. Making these compositions work in an intuitive way requires a pretty complex design which won't be presented at this point. The code dealing with this problem can be found in the file *quantumdata/Transformation.h* and the corresponding *.cc* and *.impl* files.

One point that deserves explicit mention is when exactly the dual vector has to be calculated. It would be numerically unfavourable to recalculate the dual vector every time the state vector changes since every change would then result in an expensive matrix multiplication which might in the end not be necessary at all. On the other hand it might happen that the same dual

vector is used at different points in the framework and it therefore would be convenient to store at least a reference to the dual vector in the nonorthogonal state vector. The way this problem is solved is that `StateVector` got a method named `update()` which has to be called before the dual vector is used for any calculations.

# Part III

# Examples

CHAPTER

FOURTEEN

# SINGLE MODE IN CAVITY

For testing the validity of the implemented algorithm, an exactly solvable system, a single mode in a cavity, will be solved, numerically and analytically. This makes it possible to calculate exactly the error of the simulation and verify the previously derived error estimates. The system Hamiltonian has the simple form

$$H = \hbar\omega a^\dagger a$$

and as initial state some coherent state $|\alpha_0\rangle$ is taken.

## 14.1 Analytical solution

Formally the solution is given by:

$$|\alpha\left(t\right)\rangle = e^{-\frac{i}{\hbar}Ht}|\alpha_0\rangle$$

Using that the Hamiltonian applied onto a number state,

$$H|n\rangle = \hbar\omega n|n\rangle$$

this problem can be solved analytically by changing into the number basis:

$$\begin{aligned}
|\alpha\left(t\right)\rangle &= e^{-\frac{i}{\hbar}Ht}e^{-\frac{|\alpha_0|^2}{2}}\sum_{n=0}^{\infty}\frac{\left(\alpha_0\right)^n}{\sqrt{n!}}|n\rangle \\
&= e^{-\frac{\left|\alpha_0 e^{-i\omega t}\right|^2}{2}}\sum_{n=0}^{\infty}\frac{\left(\alpha_0 e^{-i\omega t}\right)^n}{\sqrt{n!}}|n\rangle \\
&= |\alpha_0 e^{-i\omega t}\rangle
\end{aligned}$$

## 14.2 Numerical simulation

For the simulation the matrix $\left(a^\dagger a\right)^i_j$ has to be calculated. As described in the chapter *Condition numbers of problems in coherent bases* (chapter 6) this has to be done in a higher precision because of the high condition number of the transformation matrix. The script itself is pretty

simple since all needed parts are already implemented and have only to be composed. So the main work that has still to be done is interpreting the parameters that will be given from the command line.

```
#include "EvolutionHigh.h"

#include "ParsCoherentBasis.h"
#include "CoherentMode.h"

using namespace quantumdata;

typedef TTD_CARRAY(2) Trafo;
typedef NonOrthogonalStateVector<1,Trafo> NOSV;

int main(int argc, char* argv[])
{
    // Parameters
    ParameterTable p;
    ParsEvolution pe(p);
    coherent_basis::ParsHexagonal pcb(p);
    mode::ParsPumpedLossy pcm(p);

    try {update(p,argc,argv,"--");}
    catch (ParsNamedException& pne)
    {
        std::cerr << "Pars named error: " << pne.getName() << std::endl;
    }

    // Create Lattice and select lattice points
    structure::lattice::Hexagonal lattice(pcb.center, pcb.dist);
    if (pcb.dim)
        lattice.select_nearest(pcb.dim);
    else
        lattice.select_rings(pcb.rings);

    // Create Basis using this lattice
    int prec;
    if (pcb.prec)
        prec = pcb.prec;
    else {
        int N = lattice.size();
        prec = 16 + log10((N-1)/2.*pow(pcb.dist, -2*(N-1)))*4;
    }
    structure::CoherentBasis basis(lattice,prec);
    basis.calculate_trafo();
    PumpedLossyCoherentMode mode(basis, lattice, pcm);

    // Initial Statevector
    NOSV psi(basis.coordinates(pcb.initial), basis.trafo(), ByReference());
    psi.update();
    psi.renorm();
```

```
    evolve(psi,mode,pe);
}
```

This program can then be called from the command line with various parameters which determine how the simulation is done. An example would be the following command:

```
>> ./PumpedLossyCoherentMode --T 1 --Dt 0.01 --dc 0 --svdc 1
                             --kappa 0 --deltaC 6.2832
                             --center 3.1 --dist 1 --dim 19
                             --precision 16 --coherent_prec 128
                             --initial 3.2 --o mode.sv
```

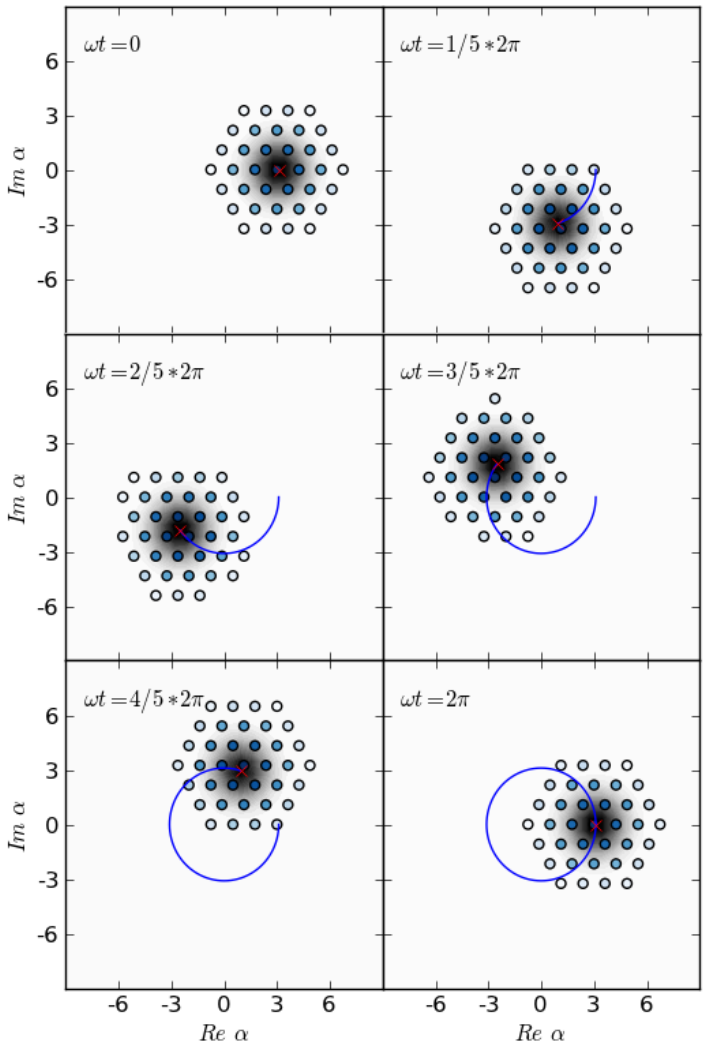
The parameters in the first line specify the integration time and how often state vectors are written to the output file. The second line sets the system parameters while the third line characterizes the used coherent basis. The fourth line specifies the amount of digits used for writing state vectors into the output file and the amount of digits used for calculating ill conditioned operators respectively. Finally, the last line provides the $\alpha$ for the initial coherent state and a filename where the output is written to.

Analyzing the output file can be done with pycppqed [pycppqed], a python library written especially for this task. It can read C++QED output files and convert the data into numpy arrays and also is able to deal with state vectors in coherent bases, which allows visualizing the time evolution in only a few lines.

```
import pycppqed
import mpmath as mp
import numpy as np
from pycppqed import coherent
import pylab

mp.mp.dps = 64

evs, qs = pycppqed.load_cppqed("mode.sv")
svs = qs.statevector

a_re = []
a_im = []

i = 0
for sv in svs.statevectors:
    states = np.array(sv.basis.states, dtype=np.complex128)
    trafo = np.array(sv.basis.trafo.tolist(), dtype=np.complex128)
    n = sv.conj()*np.dot(trafo, sv)
    alpha = np.dot(sv.conj(), np.dot(trafo,sv*states))
    a_re.append(mp.re(alpha))
    a_im.append(mp.im(alpha))
    if i*1./5-1e-5<=sv.time:
        i += 1
        ax = pylab.subplot(3,2,i)
        pylab.scatter(states.real, states.imag, c=np.log(np.abs(n)), cmap=pylab.
        pylab.plot(a_re, a_im)
```

```
        pylab.plot(a_re[-1], a_im[-1], "rx")
pylab.show()
```

The output of this python script produces the following plot, representing the state of the system at different point of times:



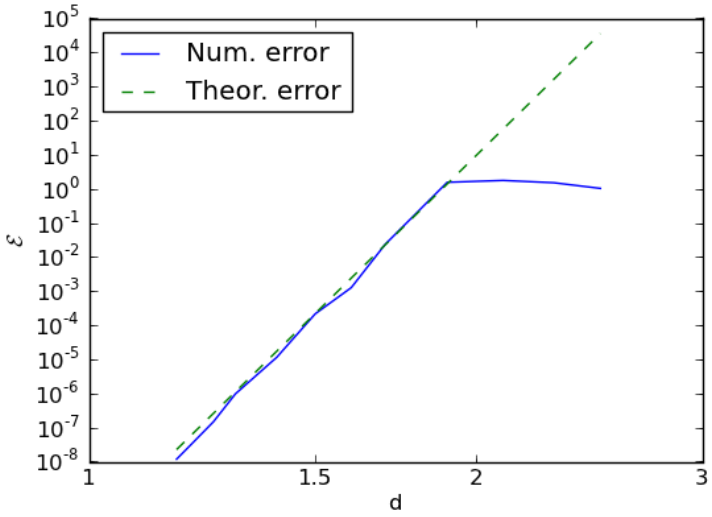

The dots symbolize the coherent basis states in the complex $\alpha$ space while their color at the same time indicates how highly occupied they are. The blue line represents the time evolution of the expectation value of $a$ with the red cross at one end marking the current value. The grey shading below represents the Q-function for the state. Comparing the plots for different point of times illustrates how the basis adapts to the problem. The expectation value is always near the center of the basis and also the occupation is always higher for the inner basis states. Qualitatively the simulation gives the right result - in the following it will be examined how good it is quantitatively.

## 14.3 Comparison of analytical and numerical solution

The error between the numerical and the analytical solution is defined as the norm of the difference between the two states:

$$\mathcal{E} = \| |\alpha\rangle - |\widetilde{\alpha}\rangle \| \approx \| \sum_i |\alpha_i\rangle_\bullet \left(\alpha^i - \widetilde{\alpha}^i\right) \| = \sum_{ik} \left(\alpha^i - \widetilde{\alpha}^i\right) \delta_{ik} \left(\alpha^k - \widetilde{\alpha}^k\right)$$

Comparing the simulation with the analytic solution for different lattice constants gives the following plot:




The dashed line shows the theoretical precision which is calculated in chapter *Coherent state bases* (chapter 5). It fits nearly perfectly with the outcome of the simulation. The reason it stops at a precision of $10^{-}8$ is that the total precision is limited by the time evolution in the MCWF-method.

CHAPTER

FIFTEEN

# PUMPED LOSSY MODE

To proof that indeed systems with high photon numbers can be simulated and at the same time testing the implementation for performing quantum jumps correctly a single mode in a pumped lossy cavity is simulated. The Hamiltonian from chapter *Single mode in cavity* (chapter 14) has to be extended by a driving term:

$$H = \hbar\omega_c a^\dagger a + \hbar\eta \left(a e^{i\omega t} + a^\dagger e^{-i\omega t}\right)$$

Changing into the interaction picture $|\Psi(t)\rangle_I = e^{-i\omega t a^\dagger a}|\Psi(t)\rangle$ makes the Hamiltonian time independent:

$$H = \hbar\delta_c a^\dagger a + \hbar\eta \left(a + a^\dagger\right)$$

Here $\delta_c$ was introduced as $\delta_c = \omega_c - \omega$. Because the cavity is lossy the time evolution has to be calculated using a master equation:

$$\dot{\rho} = -\frac{i}{\hbar}\left[H, \rho\right] + \mathcal{L}[\rho]$$

The Liouvillian is given by:

$$\mathcal{L}[\rho] = \gamma\left(a\rho a^\dagger - \frac{1}{2}a^\dagger a\rho - \frac{1}{2}aa^\dagger\rho\right)$$

For the numerical implementation the non-hermitian Hamiltonian and the jump operator have to be identified:

$$H_{nH} = \hbar(\delta_c - i\frac{\gamma}{2})a^\dagger a + \hbar\eta \left(a + a^\dagger\right)$$
$$J = a$$

## 15.1 Analytical solution

Solving the master equation is far more difficult than solving Schroedinger equations. However, the expectation value of the destruction operator $\langle a\rangle$ can be obtained relatively easily. Starting from the Langevin equation

$$\dot{a}(t) = -\frac{i}{\hbar}\left[a(t), H\right] - \frac{\gamma}{2}a(t)$$

and using the commutator relation

$$[a, H] = \left[a, \hbar\delta_c a^\dagger a + \hbar\eta\left(a + a^\dagger\right)\right] = \hbar\delta_c a + \hbar\eta$$

leads for following equation for the operator $a$:

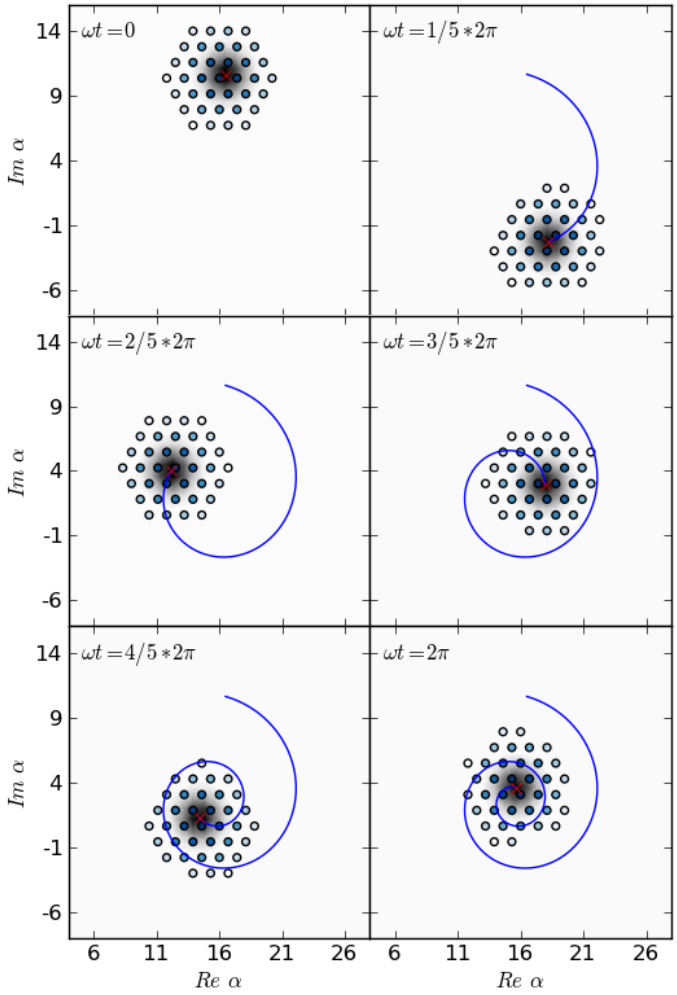
$$\dot{a}(t) = -i\left(\delta_c - i\frac{\gamma}{2}\right)a(t) - i\eta$$

Solving this equation leads to:

$$a(t) = \left(a(0) + \frac{\eta}{\delta_c - i\frac{\gamma}{2}}\right)e^{-i\left(\delta_c - i\frac{\gamma}{2}\right)t} - \frac{\eta}{\delta_c - i\frac{\gamma}{2}}$$

## 15.2 Numerical simulation

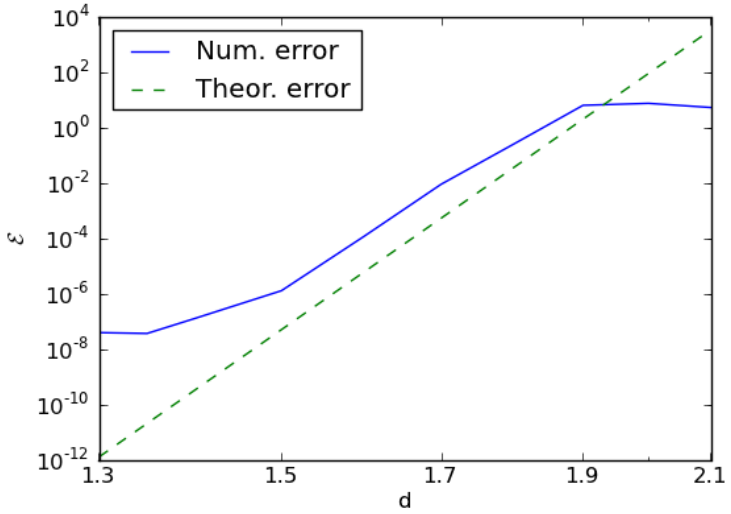
The simulation is actually very similar to the previously treated single mode in a cavity without loss. The only difference is that the parameters for $\eta$ and $\gamma$ and also the initial condition for both the initial state and the initial basis have to be set appropriately. With the coherent state $|16.5 + 10.6i\rangle$ as initial state and $\eta = 100$, $\gamma = 2$, $\delta_c = 2\pi$ the following time evolution can be observed (units were omitted in this case - they only have to be chosen consistently and do not change the qualitative outcome):



The dots symbolize the coherent basis states in the complex $\alpha$ space while their color at the same time indicates how highly occupied they are. The blue line represents the time evolution of the expectation value of $a$ with the red cross at one end marking the current value. The grey shading below represents the Q-function for the state.

## 15.3 Comparison of analytical and numerical solution

Comparing the theoretical and the numerical results for the expectation value of the destruction operator $\langle a \rangle$ for various lattice constants results in:




Although the coincidence with the theoretical prediction is not as good as in the last example (*Single mode in cavity* (chapter 14)) it still is recognizable.

CHAPTER

SIXTEEN

# SQUEEZED STATE

As another example generation of squeezed states will be simulated. Therefore degenerate parametric down-conversion is used which occurs when certain nonlinear mediums are pumped by photons of frequency $\omega_p$. Some of these photons are then converted into two identical photons with half the frequency in the so called signal mode. Using the assumption that the driving field is in a coherent state $|\beta e^{-i\omega_p t}\rangle$ all the time and changing into the interaction picture this process is described by the Hamiltonian

$$H_{int} = i\hbar \left(\eta^* a^2 - \eta a^{\dagger 2}\right)$$

In [Gerry] this is shown in more detail, here the emphasis lies on just simulating this Hamiltonian. For this purpose it will be assumed that also the signal mode is initially in a coherent state allowing us to analytically calculate the expectation values and variances in the quadrature operators.

## 16.1 Analytical solution

The evolution operator for this Hamiltonian is very similar to the squeezing operator:

$$U_{int}(t) = e^{-iH_{int}t/\hbar} = e^{\left(\eta^* a^2 - \eta a^{\dagger 2}\right)t} = S(2\eta t)$$

Calculating the expectation value of $a$ is straightforward using the relations from the chapter *Coherent states* (chapter 1). With the polar representation $\eta = re^{i\theta}$ this results in:

$$\begin{aligned}\langle a\rangle &= \langle\Psi(t)|a|\Psi(t)\rangle = \langle\Psi|U_{int}^{\dagger}(t)aU_{int}(t)|\Psi\rangle \\ &= \langle\Psi|S^{\dagger}(2\eta t)aS(2\eta t)|\Psi\rangle = \langle\Psi|\left(a\cosh 2rt - a^{\dagger}e^{i\theta}\sinh 2rt\right)|\Psi\rangle\end{aligned}$$

Choosing a coherent state $|\alpha\rangle$ as initial state this further simplifies to:

$$\langle a\rangle = \alpha\cosh 2rt - \alpha^* e^{i\theta}\sinh 2rt$$

Expectation values of the quadrature operators are still the real and the imaginary parts of this expression:

$$\begin{aligned}\langle X_1\rangle &= \mathrm{Re}\langle a\rangle \\ \langle X_2\rangle &= \mathrm{Im}\langle a\rangle\end{aligned}$$

The variances of these operators can also be calculated:

$$\mathrm{Var}(X_1) = \frac{1}{4}\left(\cosh 2rt - e^{-i\theta}\sinh 2rt\right)\left(\cosh 2rt - e^{i\theta}\sinh 2rt\right)$$
$$\mathrm{Var}(X_2) = \frac{1}{4}\left(\cosh 2rt + e^{-i\theta}\sinh 2rt\right)\left(\cosh 2rt + e^{i\theta}\sinh 2rt\right)$$

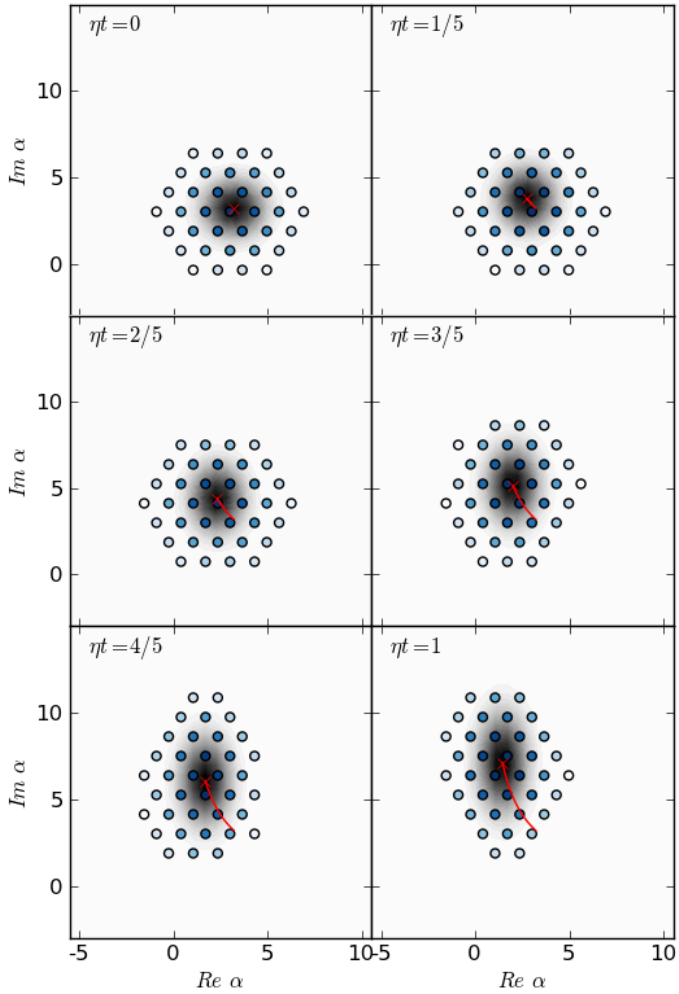
## 16.2 Numerical simulation

The C++QED script for squeezing is very similar to that of the single mode which was presented in the chapter *Single mode in cavity* (chapter 14):

```
#include "EvolutionHigh.h"

#include "ParsCoherentBasis.h"
#include "CoherentSqueezedMode.h"

using namespace quantumdata;

typedef TTD_CARRAY(2) Trafo;
typedef NonOrthogonalStateVector<1,Trafo> NOSV;

int main(int argc, char* argv[])
{
    // Parameters
    ParameterTable p;
    ParsEvolution pe(p);
    coherent_basis::ParsHexagonal pcb(p);
    mode::ParsPumpedLossy pcm(p);

    try {update(p,argc,argv,"--");}
    catch (ParsNamedException& pne)
    {
        std::cerr << "Pars named error: " << pne.getName() << std::endl;
    }
    // Create Lattice and select lattice points
    structure::lattice::Hexagonal lattice(pcb.center, pcb.dist);
    if (pcb.dim)
        lattice.select_nearest(pcb.dim);
    else
        lattice.select_rings(pcb.rings);
    // Create Basis using this lattice
    int prec;
    if (pcb.prec)
        prec = pcb.prec;
    else {
        int N = lattice.size();
        prec = 16 + log10((N-1)/2.*pow(pcb.dist, -2*(N-1)))*4;
    }
    structure::CoherentBasis basis(lattice,prec);
```

```
    basis.calculate_trafo();
    CoherentSqueezedMode mode(basis, lattice, pcm);
    // Initial Statevector
    NOSV psi(basis.coordinates(pcb.initial), basis.trafo(), ByReference());
    psi.update();
    psi.renorm();

    evolve(psi,mode,pe);
}
```

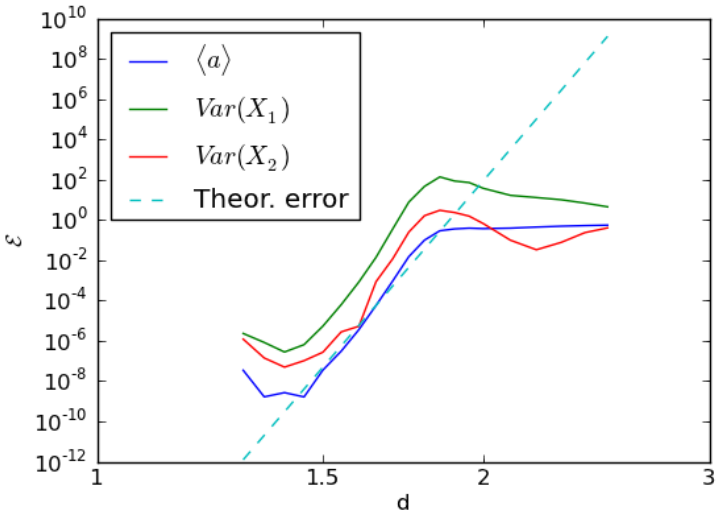
Nearly the only difference is that instead of the element `PumpedLossyCoherentMode` the element `CoherentSqueezedMode` was used which simply implements the squeezing operator. The time evolution can be visualized with the same python script that is used in *Single mode in cavity* (chapter 14):




The dots symbolize the coherent basis states in the complex $\alpha$ space while their color at the same time indicates how highly occupied they are. The blue line represents the time evolution of the expectation value of $a$ with the red cross at one end marking the current value. The grey shading below represents the Q-function for the state.

## 16.3 Comparison of analytical and numerical solution

In this example it is not possible to calculate the exact state vector. However, the expectation value of $a$ and the variances of the quadrature operators can be easily obtained analytically as well as numerically. Therefore their relative errors are used as estimate for the error of the whole simulation:




The theoretical prediction fits nearly perfectly for $\langle a \rangle$ and reasonably well for the variances of the quadrature operators.

# Part IV

# Appendices

CHAPTER

SEVENTEEN

# PYCPPQED USER GUIDE

This guide explains how to use different features of PyCppQED.

**Contents**



## 17.1 Introduction

PyCppQED is a python library that helps working with C++QED - a framework simulating open quantum dynamics written in C++. Since C++ is not the favorite programming language of everyone, PyCppQED extends this framework with useful functionality:

- Import C++QED output files into python.
- Export this data as Matlab `.mat` files.

- Fast and easy visualization/animation of imported data.
- Generating arbitrary initial conditions for C++QED.

## 17.2 Requirements

**Mandatory:**

- **Python:** The python interpreter. At least version 2.3 but not 3.x.
- **NumPy:** A numerical package for python. Provides fast N-dimensional array manipulation.

**Optional:**

- **Matplotlib:** A plotting library. It is needed for any kind of visualization. (For 3D plots at least 0.99 is needed)
- **SciPy:** Library providing scientific algorithms. It is needed for exporting numpy arrays as `.mat` files and importing `.mat` files as numpy arrays. For exporting multidimensional arrays, at least version 0.7 is needed.
- **PyGTK:** GTK bindings for python. It is needed for animations.

## 17.3 Installation

First, the c extensions have to be build:

```
$ python setup.py build
```

This creates a directory like `build/lib.linux-x86_64-2.4/pycppqed/`. Either this package can be moved somewhere and used directly (you may want to add it is location to the `PYTHONPATH`) or alternatively it can be installed:

```
$ python setup.py install --prefix=PATH
```

## 17.4 Overview

PyCppQED has a strict modular design:

- **Python classes representing objects used in QED:**
  - `pycppqed.statevector` implements state vectors.
  - `pycppqed.expvalues` implements classes for working with expectation values.
  - `pycppqed.quantumsystem` implements classes representing quantum systems.

- Functions for generating some useful initial conditions are in `pycppqed.initialconditions`.
- Everything that has to do with reading and writing C++QED files is in the module `pycppqed.io`.
- Plotting stuff is in `pycppqed.visualization` and animation functions are implemented in `pycppqed.animation`.

## 17.5 Usage

PyCppQED can be used either from scripts but also interactively.

### 17.5.1 Scripts

Scripts are simple text files with valid python code that can be executed by invoking the python interpreter with the name of the script as first argument:

```
$ python myscript.py
```

### 17.5.2 Interactive

To use python interactively just invoke the interpreter without arguments:

```
$ python
Python 2.6.2 (r262:71600, Aug 17 2009, 10:52:48)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A good enhancement to the standard python interpreter is IPython:

```
$ ipython
Python 2.4.3 (#1, Jul 27 2009, 17:56:30)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

It provides:

- **Tab completion** which allows easy inspection of modules and objects:

```
In [3]: file.re
file.read       file.readinto   file.readline   file.readlines

In [3]: file.read
```

- **Easy Inspection**:

```
In [3]: file.read?
Type:           method_descriptor
Base Class:     <type 'method_descriptor'>
String Form:    <method 'read' of 'file' objects>
Namespace:      Python builtin
Docstring:
    read([size]) -> read at most size bytes, returned as a string.

    If the size argument is negative or omitted, read until EOF is
    reached. Notice that when in non-blocking mode, less data than what
    was requested may be returned, even if no size parameter was given.
```

- **Many more features ...**

### 17.5.3 Import PyCppQED

To use PyCppQED you have to import it. This is done using an import statement:

```
>>> import pycppqed as qed
```

From now on all commands will assume that PyCppQED is already imported with this statement. Now you are ready to do a lot of fancy stuff with PyCppQED! The next section gives examples how to achieve common tasks.

## 17.6 How to ...

### 17.6.1 Split up a C++QED output file into standard output and state vectors

When a C++QED script is invoked using the *svdc* argument, state vectors are written into the output file between the calculated expectation values. With PyCppQED it is easy to extract the state vectors into own files and getting a standard C++QED output file:

```
>>> qed.io.split_cppqed("ring.dat", "newring.dat")
```

This writes the standard output file to `newring.dat` and the state vectors into separate files named `newring_time.dat.sv` where `time` is substituted by the time when this state vector was reached.

### 17.6.2 Import a C++QED output file

This is done with the function `pycppqed.io.load_cppqed()`:

```
>>> evs, qs = qed.io.load_cppqed("ring.dat")
```

This returns two objects which represent the whole information stored in the C++QED output file:

- A `pycppqed.expvalues.ExpectationValueCollection` instance which holds all expectation values calculated by C++QED.
- A `pycppqed.quantumsystem.QuantumSystemCompound` instance representing the calculated quantum system. This object also stores a `pycppqed.statevector.StateVectorTrajectory` instance which holds all calculated state vectors.

### 17.6.3 Export python data as *.mat* file

If you want to use Matlab or Octave for further processing of the data you can use PyCppQED to convert a C++QED output file into a *.mat* file. First, we have to load the file like in *Import a C++QED output file* (subsection 17.6.2). The obtained objects (or only parts of it, or any other array ...) can be saved with the `scipy.io.savemat()` function:

```
>>> import scipy.io
>>> scipy.io.savemat("out.mat", {"evs":evs, "svs":qs.statevector})
```

This file can be used from Matlab and Octave:

```
>> load('out.mat')
>> whos
  Name        Size                Bytes  Class     Attributes

  evs        15x175              21000  double
  svs         4-D               921600  double    complex

>> size(svs)

ans =

     9    64    10    10


>>> size(evs)

ans =

    15   175
```

**Warning:** Be aware that old versions of scipy (older than 0.7) can't properly export arrays with more than 2 dimensions!

### 17.6.4 Generate arbitrary initial conditions

In the module `pycppqed.initialconditions` are some convenient functions that let you easily create common initial conditions. E.g. to create a gaussian wave packet in the k-space the following command can be used:

```
>>> sv_p = qed.initialconditions.gaussian(x0=1.1, k0=5, sigma=0.3, fin=7)
>>> print sv_p
StateVector(128)
```

Or a coherent mode:

```
>>> sv_m = qed.initialconditions.coherent(alpha=2, N=20)
>>> print sv_m
StateVector(20)
```

To obtain initial conditions for a combined quantum system simply use the ^ operator:

```
>>> sv = sv_a ^ sv_m
>>> print sv
StateVector(128 x 20)
```

It is easy to create any other initial condition you can think of, by simply creating a numpy array with the wanted values and then using the array to build a `pycppqed.statevector.StateVector`:

```
>>> import numpy as np
>>> X = np.linspace(0,10,64) # An array with 64 values between 0 and 10
>>> Y = np.sin(X)
>>> sv = qed.statevector.StateVector(Y, norm=True)
>>> print sv
StateVector(64)
```

### 17.6.5 Export initial conditions as C++QED sv files

Exporting is done with the `pycppqed.io.save_statevector()`:

```
>>> sv = qed.statevector.StateVector((1,2,3), norm=True)
>>> qed.io.save_statevector("mystatevector.sv", sv)
```

The created file then looks like:

```
# 0 1
(0,2)
[ (0.267261241912,0.0) (0.534522483825,0.0) (0.801783725737,0.0) ]
```

### 17.6.6 Calculate standard expectation values

By *standard* expectation values we mean values that are also calculated by C++QED. Automatic calculation for those is implemented in `pycppqed.quantumsystem`. All that has

to be done is to first create a proper quantum system object and then call its `expvalues()` method:

```
>>> sv = qed.initialconditions.gaussian(x0=0.5, k0=3.2, sigma=0.4, fin=7)
>>> qs = qed.Particle(sv)
>>> evs = qs.expvalues()
>>> print evs
ExpectationValueCollection('<k>', 'Var(k)', '<x>', 'Std(x)')
ExpectationValueCollection([ 3.2000+0.j,  1.5625+0.j,  0.5000+0.j, 0.4000+0.j])
```

It is also possible for combined quantum systems:

```
>>> sv_p = qed.initialconditions.gaussian(x0=0.5, k0=3.2, sigma=0.4, fin=7)
>>> sv_m = qed.initialconditions.coherent(alpha=2, N=20)
>>> sv = sv_p ^ sv_m
>>> q = qed.quantumsystem
>>> qs = q.QuantumSystemCompound(sv, q.Particle, q.Mode)
>>> print qs
QuantumSystemCompound(Particle(128), Mode(20))
>>> print evs
ExpectationValueCollection('<k>', 'Var(k)', '<x>', 'Std(x)', '<n>', 'Var(n)',
                           'Re(<a>)', 'Im(<a>)')
>>> print repr(evs)
ExpectationValueCollection([  3.19999997e+00+0.j,   1.56250009e+00+0.j,
    4.99999995e-01+0.j,  4.00000001e-01+0.j,   3.99999979e+00+0.j,
    3.99999747e+00+0.j,   1.99999990e+00+0.j,   6.41996804e-18+0.j])
```

We get a quantum system for free if we load a C++QED output file:

```
>>> evs, qs = qed.io.load_cppqed("ring.dat")
>>> evs_calc = qs.expvalues()
```

### 17.6.7 Calculate arbitrary expectation values

Expectation values for combined systems are calculated in the following way (Assuming the operator only acts on first subsystem):

$$\langle\Psi|\hat{A}(k)|\Psi\rangle = \sum_{k_1k_2}\langle k_1|\hat{A}(k)|k_2\rangle \sum_m \Psi^*_{k_1m}\Psi_{k_2m}$$

That means the expectation value is determined by specifying the quantity $A_{k_1k_2} = \langle k_1|\hat{A}(k)|k_2\rangle$. E.g. let us calculate the expectation value of the destruction operator of a combined system of structure {Particle, Mode}:

```
>>> sv_p = qed.initialconditions.gaussian(x0=0.5, k0=3.2, sigma=0.4, fin=7)
>>> sv_m = qed.initialconditions.coherent(alpha=0.5, N=5)
>>> sv = sv_p ^ sv_m
>>> import numpy as np
>>> a = np.diag(np.sqrt(np.arange(1,5)), 1)
```

```
>>> print a
[[ 0.          1.          0.          0.          0.        ]
 [ 0.          0.          1.41421356  0.          0.        ]
 [ 0.          0.          0.          1.73205081  0.        ]
 [ 0.          0.          0.          0.          2.        ]
 [ 0.          0.          0.          0.          0.        ]]
>>> ev_a = sv.expvalue(a, 1)
>>> print ev_a
(0.499933315175-7.96953264544e-18j)
```

The second argument tells the expvalue method that the operator is only working on the second subsystem. (Python starts counting with 0!)

Let us now consider a slightly more complicated example - a combined system of the form {Particle, Mode, Mode} and let us try to calculate the expectation value for the operator $T = a_1^\dagger a_2 + a_2^\dagger a_1$:

```
>>> sv_p = qed.initialconditions.gaussian(x0=0.5, k0=3.2, sigma=0.4, fin=7)
>>> sv_m1 = qed.initialconditions.coherent(alpha=0.5, N=5)
>>> sv_m2 = qed.initialconditions.coherent(alpha=2, N=20)
>>> sv = sv_p ^ sv_m1 ^ sv_m2
>>> import numpy as np
>>> SV = qed.statevector.StateVector # Define an abbreviation.
>>> m1_a = SV(np.diag(np.sqrt(np.arange(1,5)), 1))
>>> m1_at = SV(np.diag(np.sqrt(np.arange(1,5)), -1))
>>> m2_a = SV(np.diag(np.sqrt(np.arange(1,20)), 1))
>>> m2_at = SV(np.diag(np.sqrt(np.arange(1,20)), -1))
>>> T = (m1_at ^ m2_a) + (m1_a ^ m2_at)
>>> print sv.expvalue(T, (0,1))
(1.99973315754-1.54328182927e-20j)
```

### 17.6.8 Calculate diagonal expectation values

If we want to calculate the expectation value of an diagonal operator, we can use the `pycppqed.statevector.StateVector.diagexpvalue()` method. It takes only the diagonal elements of the operator's matrix representation and has the advantage to be faster and to need less memory than the `pycppqed.statevector.StateVector.expvalue()` method.

A short example:

```
>>> sv = qed.initialconditions.gaussian(x0=0.5, k0=3.2, sigma=0.4, fin=7)
>>> import numpy as np
>>> K = np.arange(-64, 64)
>>> print sv.diagexpvalue(K, 0)
(3.2+0j)
```

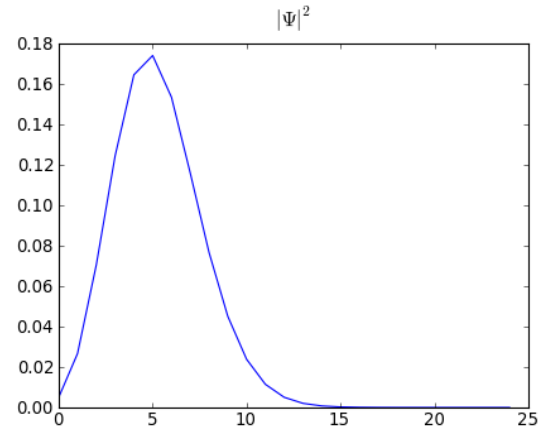
### 17.6.9 Visualize PyCppQED objects

There are basically 4 different types of PyCppQED objects which might be interesting to look at:

- `pycppqed.statevector.StateVector`
- `pycppqed.statevector.StateVectorTrajectory`
- `pycppqed.expvalues.ExpectationValueTrajectory`
- `pycppqed.expvalues.ExpectationValueCollection`

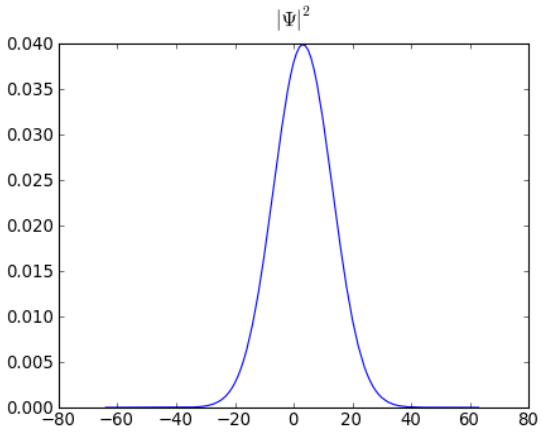
All of them are inheriting from `numpy.ndarray` which means you can easily plot them using Matplotlib or Gnuplot. However, PyCppQED implements some functions to let you take a quick look on these objects. All but the StateVectorTrajectory class have a `plot()` method:

```
>>> sv = qed.initialconditions.coherent(alpha=2.3, N=25)
>>> sv.plot()
```



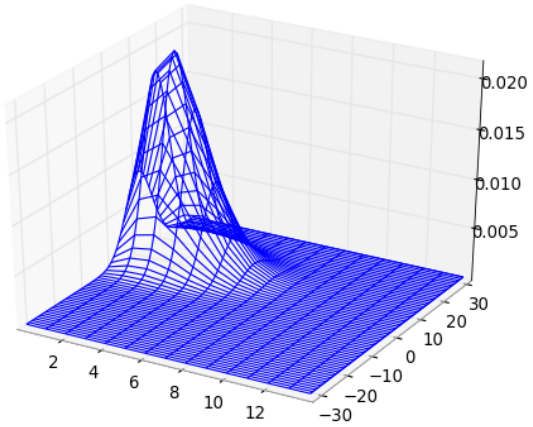

To change the x-axis we can pass an array of x-coordinated to the plot method:

```
>>> import numpy as np
>>> sv = qed.initialconditions.gaussian(x0=0.5, k0=3.2, sigma=0.05, fin=7)
>>> K = np.arange(-64,64)
>>> sv.plot(x=K)
```



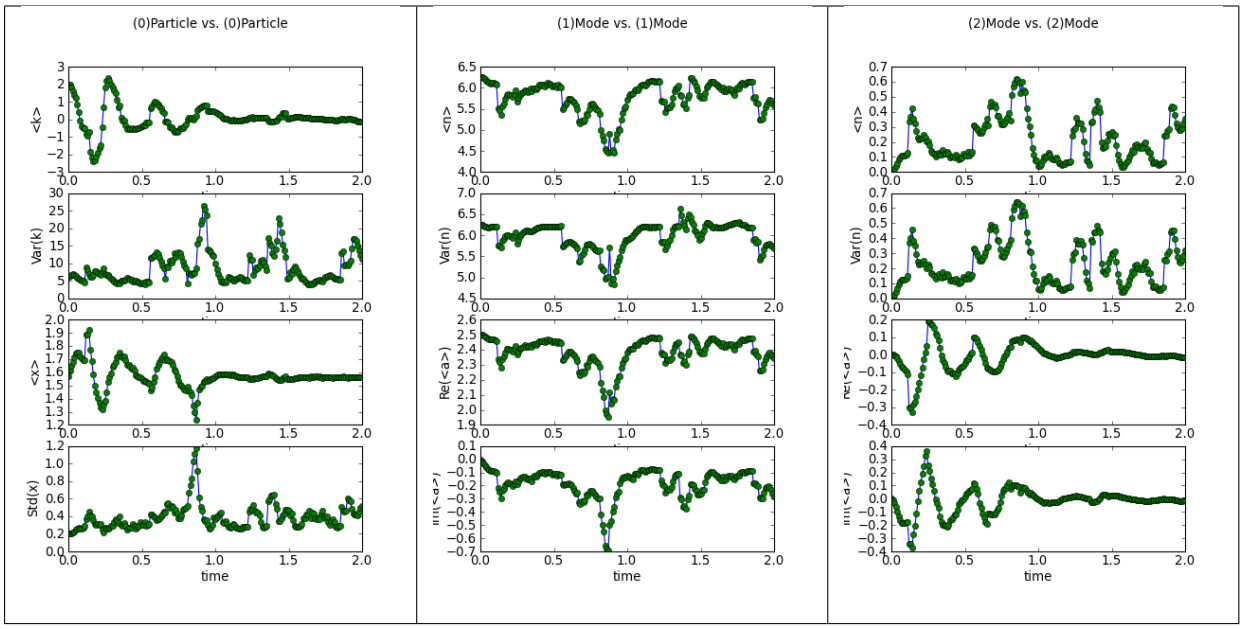
Since Matplotlib version 0.99 it is also possible to draw 3D graphs. This can be used for combined systems:

```
>>> sv_p = qed.initialconditions.gaussian(x0=0.5, k0=10, sigma=0.1, fin=6)
>>> sv_m = qed.initialconditions.coherent(alpha=1.5, N=15)
>>> sv = sv_p ^ sv_m
>>> import numpy as np
>>> K = np.arange(-32,32)
>>> sv.plot(x=K)
```




The expectation value classes work equivalent. Maybe also useful is the function `pycppqed.visualization.compare_expvaluecollections()`. As its name says it is used to compare two sets of expectation values:

```
>>> evs, qs = qed.io.load_cppqed("ring.dat")
>>> evs_calc = qs.expvalues()
>>> qed.visualization.compare_expvalluecollections(evs, evs_calc)
```




The only object that is now left, is the `pycppqed.statevector.StateVectorTrajectory` class. It represents the time evolution of a state vector. For a 1D system there are already three dimensions to plot. This would be possible, but an alternative is to use an animation which

will also work for 2D systems. Animations are implemented as interactive window, but it is also possible to save movies in any format mencoder can write. This functionality is only very basic and it may need changes on the source code to obtain professional looking movies. However, here is the code:

```
>>> import numpy as np
>>> time = np.linspace(-np.pi, np.pi, 2**6)
>>> g = qed.initialconditions.gaussian
>>> svt = [g(sigma=t) ^ g(sigma=0.15-t) for t in time]
>>> svt.animate()
```

# BIBLIOGRAPHY


[cppqed] http://cppqed.sourceforge.net

[pycppqed] http://github.com/bastikr/pycppqed

[mpmath] http://code.google.com/p/mpmath/

[mpfr] http://www.mpfr.org

[mpc] http://www.multiprecision.org

[mpfr-cpp] http://www.holoborodko.com/pavel/mpfr

[blitz] http://www.onumerics.org/blitz

[Dum-Zoller-Ritsch] 18. Dum, P. Zoller and H. Ritsch, "Monte Carlo simulation of the atomic master equation for spontaneous emission", Phys. Rev. A 45, 4879 (1992)

[Vukics-Ritsch] 1. Vukics and H. Ritsch, "C++QED: an object-oriented framework for wave-function simulations of cavity QED systems", The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics (2007)

[Steinbach] Steinbach, J. and Garraway, B. M. and Knight, P. L., "High-order unraveling of master equations for dissipative evolution", Phys. Rev. A 51, 3302 (1995)

[Gerry] Christopher Gerry and Peter Knight, "Introductory Quantum Optics", Cambridge University Press

[Bargmann] 22. Bargmann, P. Butera, L. Girardello, and J. R. Klauder, "Discrete coherent states on the von Neumann lattice", Rep. Math. Phys. 2, 221 (1971)

[Walls-Milburn] 4. Walls and G. Milburn, "Quantum Optics", Springer (1995)