Quantum Chaos in Second Harmonic Generation

A Quantum State Diffusion Approach

Stuart Midgley

August 28, 1997

Supervisor
Dr Craig Savage

Thesis submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science (Honours)
at the Australian National University

Declaration of Originality

I declare that the content of this thesis,
except where specified, is my original work.

Stuart Midgley
August 28, 1997

Failure requires no preparation : Sandman

Acknowledgments

I would like to thank the following people

- Craig for providing excellent guidance, support and help in all areas. I found his advise precise, concise and to the point.
- David Singleton for providing endless answers to my endless questions about the super computer and getting the most out of it. Without David I wouldn't have made it past first base.
- Mark Andrews for finding the subtle flaws in my arguments and providing excellent guidance in trying to correct them.
- The ANU Supercomputer Facility for buying me the latest, greatest super computer, allowing my simulations to be performed.
- The Teaching and Learning Technology Support Unit for solving all those nigly little computer problems.
- Gem for providing endless support throughout the year, being an excellent grammar checker, spell checker, style checker the lot.
- Joe Hope for solving a few odd problems and providing great fodder in network computer games.
- Glenn Moy for finding those intricate little mistakes in my thesis.
- Brett, Janie, James, Mike, Jason and James for providing the right amount of distraction to my studies.

Abstract

Recent work on numerically solving open quantum systems has lead to the development of an efficient algorithm, called *Quantum State Diffusion* [Schack 1995].

The quantum state diffusion method was implemented to model *Second Harmonic Generation*, allowing investigation of quantum chaos and the quantum/classical interface. The computer code was run for various regimes where the classical second harmonic generation equations demonstrate limit cycles and chaos, with individual trajectories and Q-functions of reduced average density operators obtained.

Using a recent definition of chaos [Schack 1996 a], which is applicable to quantum systems, second harmonic generation was shown to be hyper-sensitive to perturbations in the regime where the classical system is chaotic.

Contents

List of Figures

Introduction

This thesis is written to investigate *Quantum Chaos in Second Harmonic Generation* and the role the environment plays in obtaining classical chaos from a quantum system. The main emphasis is on the numerical work performed to explore these aims. This work follows directly from previous work carried out by Dr Craig Savage and Ms Xiping Zheng, using a new model developed recently by Dr Rüdiger Schack, Dr Todd Brun and Dr Ian Percival.

Throughout the thesis it is assumed that the reader has some knowledge of quantum optics and open systems. Concepts such as "creation" and "annihilation" operators are not defined, but used extensively. For a good introduction to quantum optics see "Quantum Optics" by D F Walls and G J Milburn [Walls 1995].

The field of quantum chaos and particularly the method of quantum state diffusion is relatively new. The first two chapters give an overview of the literature and the concepts involved, followed by details of the numerical procedure and results. The final section is a listing of the code written to numerically solve the quantum system and is included as a reference. The code included is not the actual code that was used for the numerical simulation, but it is the easiest to understand. The code used for the simulation was highly optimised for the architecture of the VPP300 super computer at the Australian National University.

A brief introduction to the main concepts used in this thesis is given in chapter 1 "Fundamentals of Quantum Optics". A comparison between quantum optics and classical optics is drawn, with the emphasis being on possible methods for making a quantum system classical. Chaos in classical and quantum systems is discussed, with the difficulties in defining chaos for quantum systems being the focus.

Chapter 2, "Second Harmonic Generation", discusses the same topics as chapter 1 in the context of second harmonic generation. Chapter 3, "Solving the Master Equation for Second Harmonic Generation", provides the explicit details about solving the quantum second harmonic generation system, the model being used and the numerical consideration. Chapter 4, "Results of Numerical Simulations", provides the results of the simulations, along with discussion of their meaning. Chapter 5, "Conclusion", summarises the results and suggests areas for further research.

The original work in this thesis:

- the Q-functions of the average reduced density operators for second harmonic generation, Sec. 3.5.2 and Sec. 4.2.3
- hyper-sensitivity of second harmonic generation to perturbations, Sec. 3.5.3 and Sec. 4.2.4
- all computer code produced, App. A

The computer used to perform the simulation was a Fujitsu VPP300 super computer. It has 13 vector processors, each having 512megabytes of memory connect together by a full crossbar capable of transferring data at 570megabytes/second. The peek speed of each processor is 2.2gigaflops, giving a maximum output of 28gigaflops. It has 48gigabytes of RAID disk array with an maximum throughput of 60megabytes/second and is connected to the Australian National Universities massive data storage facility with an almost 3terabytes of storage. The VPP300 at the ANU is rated as the fastest university based computer, outside of Japan (where similar machines are installed).

Chapter 1

Fundamentals of Quantum Optics

This chapter will introduce the topic of quantum optics and outline the major ideas used in this thesis. In Sec. 1.1 a brief introduction to quantum optics and a comparison to classical optics is made, along with a discussion of the methods used to traverse the quantum/classical interface. This is followed by an overview of open systems in Sec. 1.2. Chaos in both the classical and quantum worlds is discussed in Sec. 1.3, with the emphasis on obtaining a definition of chaos which is suitable for both quantum and classical systems.

1.1 Introduction to Quantum Optics

Quantum optics is the study of the manner in which light interacts and behaves. The theory used to describe light and its interactions is that of Quantum Mechanics as opposed to the classical theory of Maxwell. Quantum optics makes the assumption that light is described by a state vector with various operators (and operator equations) describing its evolution and interaction.

This description is similar to the Quantum treatment of particles, with optical devices merely effecting the evolution of the state vector. Like with the quantum theory of matter, the state vector $|\psi\rangle$ of light in a particular mode can be represented in an arbitrary basis according to

$$|\psi\rangle = \sum_{i=0}^{\infty} \alpha_i |b_i\rangle, \tag{1.1}$$

where the $|b_i\rangle$ are the appropriate basis states defining the space the state vector is described in and the α_i are complex coefficients describing the amplitude of the state vector.

For a more comprehensive and general introduction to the field of Quantum Optics see [Walls 1995].

1.1.1 The Classical Approach to Optics

The classical approach to optics is to describe light by a wave obeying Maxwell equations (in source/sink free form)

$$\nabla \cdot \vec{B} = 0 \tag{1.2}$$

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \tag{1.3}$$

$$\nabla \cdot \vec{E} = 0 \tag{1.4}$$

$$\nabla \times \vec{B} = \mu_o \varepsilon_o \frac{\partial \vec{E}}{\partial t}. \tag{1.5}$$

In these equations, Eq. 1.2 to Eq. 1.5, $\vec{B}$ is the magnetic field and $\vec{E}$ is the electric field, with μ_o being the permeability of free space and ε_o being the permittivity of free space. The format used is consistent with the usual formulation of the source and sink free Maxwell equations.

Various optical devices are then implemented by their effect on the refractive index of the medium, the diffraction/refraction they cause, the electric properties of the device, or the magnetic properties of the device.

The actual interaction of light with objects and particles is usually through electro-magnetic interactions such as the dipole interaction. This is a treatment involving the effect that the light wave (an electro-magnetic wave) has upon the electric or magnetic properties of particles. This is also the basis of *light-particle* and *light-particle-light* interactions, such as the absorption of light and the frequency doubling of light.

1.1.2 The Quantum Approach to Optics

As described above, the quantum approach to optics is to assume light is described by a state vector of the form of Eq. 1.1. The choice of basis is arbitrary, though it is usually chosen to have physical significance for the problem. The choice of a time dependent or independent basis is also arbitrary, and signified by the language *Schrödinger* or *interaction* picture respectively.

The interaction of light with particles and objects is performed through the use of *operators* which describe the system being modeled. These operators *operate* on the state vector describing the system.

The equation describing the evolution of a state vector is the *Schrödinger* equation

$$i\hbar\frac{\partial}{\partial t}|\psi\rangle = H|\psi\rangle, \tag{1.6}$$

where H is the Hamiltonian representing the interaction of the system on the state vector, and $\hbar$ is Planck's constant. The Hamiltonian defines the system under consideration.

1.1.3 Transition from Quantum Mechanics to Classical Mechanics

Planck's constant $\hbar$ is a universal constant, fundamentally entrenched in Quantum Mechanics. Its numerical value is $\hbar = 1.0546 \times 10^{-34} Js$. Planck's constant sets the scale of quantum systems in comparison to classical mechanics. Consider the harmonic oscillator. In quantum mechanics, the energy levels are discrete, and separated by energy $\hbar\omega$ (where ω is the frequency of oscillation within the potential). Classically, the harmonic oscillator has a continuous energy spectrum. In this example, Planck's constant is defining the scale between classical mechanics and quantum mechanics.

When the transition from quantum mechanics to classical mechanics is made, it is often done through the setting of $\hbar$ to zero. In the above example, this can be seen to make the quantum energy spectrum continuous, thus (it is hoped) giving rise to classical mechanics. There are other ways of approaching the quantum/classical interface, and in this thesis the method being considered is the limit as the photon number tends to infinity, $n \to \infty$. To understand this consider a classical field with amplitude α. It has the property that

$$\alpha\ \alpha^* = \alpha^*\ \alpha, \tag{1.7}$$

where $\alpha\ \alpha^*$ is the intensity of the field. The quantum equivalent is the relationship

$$aa^\dagger = 1 + a^\dagger a, \tag{1.8}$$

where a is the annihilation operator and $a^\dagger$ is the creation operator. In the classical limit $\langle\psi|a|\psi\rangle$ and $\langle\psi|a^\dagger|\psi\rangle$ become the field amplitudes α and α^*, requiring that the expectation value of Eq. 1.8 becomes Eq. 1.7. For this to occur the state $|\psi\rangle$ requires a large photon number in order to make the 1, in Eq. 1.8, negligible.

It is not the case, however, that all quantum optical systems are classical if they have a large photon number. It is an aim of this thesis to understand this approximation in the context of second harmonic generation.

1.2 Open Systems

An open system is a system that is coupled to an environment. Such systems include:

- the laser, where the light leaves the optical cavity and enters the environment
- the Michelson Interferometer, where one port of the interferometer often has the environment as its input
- Second Harmonic Generation, where light enters the cavity from the environment, and light leaves the cavity into the environment
- any system where measurements occur.

The key feature of open systems is the environment. Often, the dynamics of the closed system are well understood. This is not so with an environment. As the word "environment" suggests, it is, in some sense, large and all encompassing. The major property of the environment is that it is relatively unaffected by the coupling with the system. The dynamics of the system, however, are affected by the coupling with the environment. Take, for example, a glass of water thermally coupled to a lake. If the water in the glass is heated (by some external means), the temperature of the lake remains relatively unchanged. However the lake is capable of removing or adding heat to the glass depending on their relative temperatures. In this way, the lake (environment) greatly effects the temperature of the glass of water.

The mathematical details behind dealing with environments are many and varied. They often involve assuming that the environment produces high frequency continuous random fluctuations to the system, then averaging over these fluctuations removing any explicit dependence on the environment from the equations. The solutions of such systems are often very difficult to obtain, requiring numerical simulation. The effect of the environment on a quantum system is often easily understood as a partial measurement of the system. A measurement of a quantum system forces the system to be in a particular state, thus collapsing the state vector. In this way, the environment tends to force the state of the system to be localised.

1.3 Chaos

The study of *chaos* is the study of systems which exhibit sensitivity to initial conditions over a continuous range of initial conditions. Chaotic systems are a subset of dynamic systems and often display scale invariance, stable points, cycles of varying period and cycles of infinite period (chaos). Classical systems as simple as a damped driven pendulum can exhibit all these features.

A system is said to be *chaotic* if it has a positive real Lyapunov exponent. This is equivalent to saying that the Jacobian, associated with the coupled differential equations

modeling the system, has a positive eigenvalue [Fox 1990]. This means that two bound solutions initially close to each other diverge exponentially for some time.

Chaotic systems, governed by Hamilton's equations, were already discovered at the turn of the century by the mathematician H Poincaré (1892). It was not until 1963 that interest grew in these systems, when E N Lorenz found a simple set of three coupled, first order, nonlinear differential equations that lead to chaotic trajectories. Lorenz is credited (among other things) with discovering one of the first examples of *deterministic chaos* in dissipative systems [Schuster 1988].

1.3.1 Classical Chaos

Classical chaos has been studied for many years. Chaos has been discovered and rediscovered by many different people in many different fields. It appears in everything from predicting the weather to mathematical constructs such as the logistic map. The behavior of these systems has been the centre of great interest and has been studied in great depth.

An example of a simple system which displays chaos is the logistic map. The iterative equation of the logistic map is given by

$$x_{n+1} = r x_n (1 - x_n). \tag{1.9}$$

The subscript n denotes the iteration step of the variable x. For various values of r, this equation can display great structure and ultimately chaos. In Fig. 1.1 the period doubling sequence to chaos for the logistic map Eq. 1.9 is illustrated. As can be seen the number of stable points undergoes a period doubling sequence to chaos (from $r = 1$ to ≈ 3.6). Then from the chaos emerges a sequence of stable points with periodicity 3 (at around $r \approx 3.8$), which then under go a doubling sequence to chaos again. This sort of behavior is typical of chaotic systems.

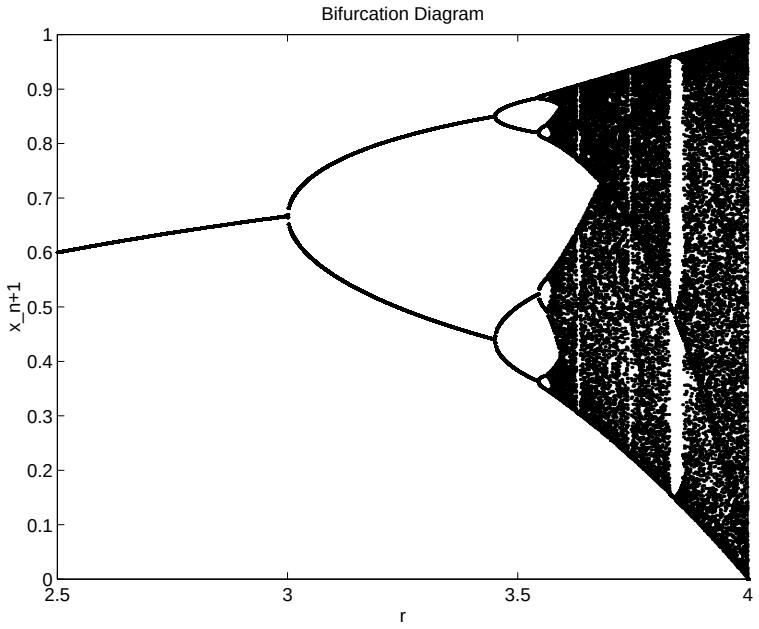

Figure 1.1: The bifurcation diagram for the logistic map. On the x-axis are the values of r, on the y axis are the values of x, which are stable (ie. the iteration of the logistic map tends toward these values).

1.3.2 Quantum Chaos

Quantum Chaos is an ill-defined term. The existence of Heisenberg's uncertainty principle precludes the notion of orbits since position and momentum cannot be measured with infinite precision at each instant [Ford 1992]. Since most definitions of chaos are defined in terms of orbits, they are not valid in describing chaos in quantum systems. One definition of chaos that seems to hold for quantum systems, as well as classical systems, is an information theory definition. The definition used [Schack 1996 a] relies on being able to show that the system is hyper-sensitive to perturbation. The main feature of such a definition is comparing the amount of information needed about the perturbation to keep the change in entropy of the system to some small amount. Schack and Caves [Schack 1996 a] demonstrate that such a definition works for classical systems, is well defined for quantum systems and appears to work for Hamiltonian systems. This thesis will attempt to apply the principles to the open system: second harmonic generation.

Other problems with finding chaos in quantum systems arise from the linearity of quantum mechanics. This can be typified by

$$\begin{aligned} i\hbar\frac{\partial}{\partial t}(|\psi_1\rangle + |\psi_2\rangle) &= H(|\psi_1\rangle + |\psi_2\rangle) && (1.10)\\ &= H|\psi_1\rangle + H|\psi_2\rangle && (1.11) \end{aligned}$$

and

$$\begin{aligned} i\hbar\frac{\partial}{\partial t}(\alpha|\psi\rangle) &= H(\alpha|\psi\rangle) && (1.12)\\ &= \alpha H|\psi\rangle. && (1.13) \end{aligned}$$

The linearity of quantum mechanics is often given as the reason it cannot give rise to chaos (once a suitable definition is obtained). It is important to make the distinction between the linear theory of quantum mechanics and the linear formalism of classical mechanics: *Liouville's theorem* [Goldstein 1965]. Liouville's theorem gives a linear equation of evolution for a phase space density D, and yet classical mechanics is still capable of demonstrating chaos, for example the damped driven pendulum. This formalism does not deal with an individual solution of a system, but rather a density of solutions, and gives an equation which is essentially a conservation of phase space volume rule. This means that the overlap (a measure of distance between the two densities) of two phase space densities remains constant throughout the evolution. Quantum mechanics of closed systems describes the evolution of individual states using a linear equation Eq. 1.6, which describes the unitary evolution of a state vector.

A non-linear equation is required to develop chaos. A linear equation does not show sensitivity to initial conditions for bound solutions. That is, two bound solutions starting close to each other remain close to each other. It is important to note that a chaotic system can show exponential divergence of two bound solutions, initially close to each other, for some time Δt. In this sense, the linearity of closed quantum systems excludes chaos. It is important to note the solutions must be bound. This can be seen via Liouvillian dynamics where conservation of phase space density and overlaps of densities is prevalent, forcing solutions to be bound within some density of phase space, yet as stated earlier, choas exists in classical mechanics.

It is also known that the density operator formalism of quantum mechanics does not admit chaos, as it has no positive real Lyapunov exponents [Haken 1983] (pp. 4.158). The density operator formalism, however, may still show hyper-sensitivity to perturbations, which

has been shown to have a suitable definition in classical Liouville theory as well as quantum mechanics [Schack 1996 a].

1.3.3 From Quantum Optics to Classical Chaos

As outlined above, quantum mechanics generally does not give rise to chaos. If quantum mechanics is assumed to be *the correct theory* then it must be able to explain the chaos that appears in classical mechanics. This leads to the problem of how does a quantum system change as it is taken to the quantum/classical interface? The answer is not easily studied as it is in the regime where many assumptions are no longer strictly valid, as the system is neither quantum mechanical nor classical.

This regime is also difficult for computational experiments to be performed. Since the normal approximations (to evolve a quantum system to a classical system) cannot be made, the system must be treated fully quantum mechanically, while pushing it to the quantum/classical interface. Often when doing this the computer resources are exhausted long before the interface is reached.

Another question arises: if chaos cannot directly arise from the equations of quantum mechanics, why is there classical chaos? One possible solution lies in the difference between classical systems and quantum systems. Classical systems are extended systems and do not merely comprise of a small number of photons or particles. In order to model a classical system using quantum mechanics, a very large number of quantum states need to be combined to get their joint evolution. A very large amount of information is lost through this combining process (usually an averaging), and the resulting evolution is no longer unitary (at the very least time reversibility is lost). Thus the joint evolution no longer obeys the Schrödinger equation, and consequently has different dynamics to the individual quantum states.

Chapter 2

Second Harmonic Generation

This chapter will introduce second harmonic generation, with the focus being on chaos and the quantum/classical interface. The physical realisation of second harmonic generation is given in Sec. 2.1 and its description in both classical and quantum optics is provided. The quantum systems interaction with the environment is then discussed in Sec. 2.2. The last section Sec. 2.3 provides details of how chaos emerges from second harmonic generation and provides a definition of chaos which is applicable to both quantum and classical mechanics.

2.1 Introduction to Second Harmonic Generation

This chapter is written using similar sections to chapter 1, with the emphasis on Second Harmonic Generation.

Second harmonic generation is the conversion of light of frequency $\omega_1 = \omega$ to frequency $\omega_2 = 2\omega$. In the context of this thesis, the system being considered is an optical cavity being driven by light at frequency ω, with a crystal converting photons of frequency ω_1 to ω_2. Inside the cavity, the photons are in one of two cavity modes, Ω_1 and Ω_2, where the detunings $\delta_1 = \Omega_1 - \omega_1$ and $\delta_2 = \Omega_2 - \omega_2$ are adjustable.

A simple schematic of the system is shown in Figure 2.1.

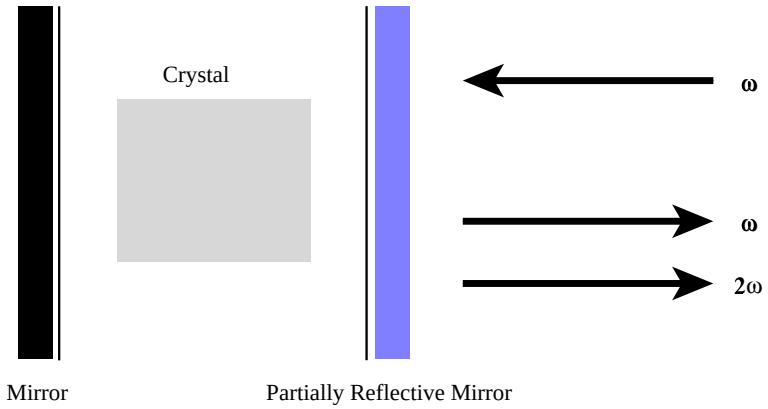

Figure 2.1: Schematic of second harmonic generation

This system can be modeled in both quantum optics and classical optics. This allows the evolution in both models to be compared, thus allow the quantum/classical interface to be examined. Second harmonic generation was chosen as it is simply modeled and demonstrates chaos in the classical regime (whereas other systems such as the simple harmonic oscillator do not).

2.1.1 The Classical Approach to Second Harmonic Generation

The classical model of second harmonic generation can be obtained from Maxwell's equations using the slowly varying envelope and mean-field approximations. The resulting equations are second order coupled differential equations, given by [Savage 1983] [Zheng 1995]

$$\frac{d\alpha_1}{dt} = E - (i\delta_1 + \kappa_1)\alpha_1 + \chi\alpha_1^*\alpha_2 \tag{2.1}$$

$$\frac{d\alpha_2}{dt} = -(i\delta_2 + \kappa_2)\alpha_2 - \frac{1}{2}\chi\alpha_1^2, \tag{2.2}$$

where α_1 and α_2 are the amplitudes of the modes 1 and 2 respectively; E is the amplitude of the external driving of mode 1; χ is the nonlinear coupling strength between the two modes; δ_1 and δ_2 are the detunings of the cavity modes from the driving field; and κ_1 and κ_2 are the loss rates from the two modes of the cavity respectively.

From these equations typical solutions for various regimes resemble the plots in Fig. 2.2. The period doubling can be seen, with the final plot showing what chaos looks like for mode 1 of this system.

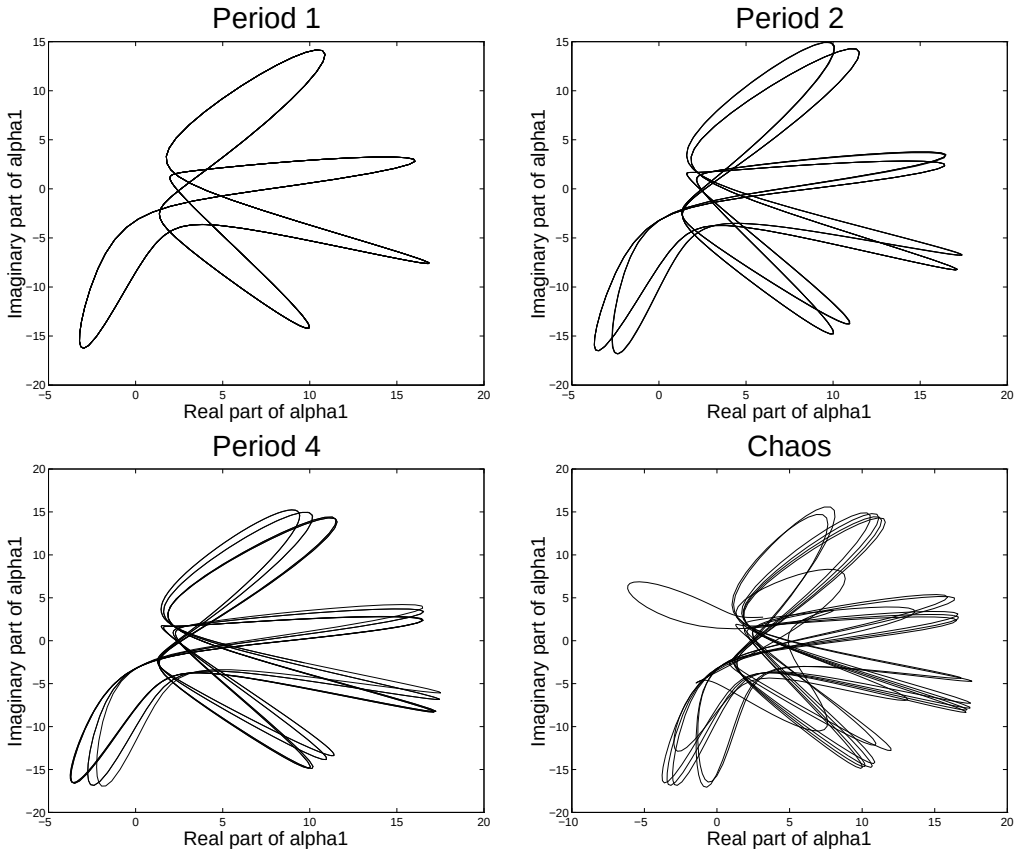

Figure 2.2: These are typical partial (one mode) phase space plots of the classical equations for second harmonic generation. From left to right, top to bottom, the values used are $\chi = 0.5ms^{-2}V^{-1}$, $\delta_1 = -0.4s^{-1}$, $\delta_2 = -0.4s^{-1}$,$\kappa_1 = 0.4s^{-1}$, $\kappa_2 = 0.1s^{-1}$ and $E = 9.7, 9.87, 9.888Vm^{-1}$ and $9.92Vm^{-1}$ respectively.

2.1.2 The Quantum Approach to Second Harmonic Generation

The quantum optics approach to Second Harmonic Generation is to construct a Hamiltonian to represent the system. This Hamiltonian in the interaction picture is given by:

$$H = \hbar\delta_1 a_1^\dagger a_1 + \hbar\delta_2 a_2^\dagger a_2 + i\hbar f(a_1^\dagger - a_1) + i\hbar\frac{\chi}{2}\left({a_1^\dagger}^2 a_2 - a_1^2 a_2^\dagger\right). \tag{2.3}$$

This equation is written in second quantised form, where the a's and $a^\dagger$'s are annihilation and creation operators, respectively. The first two terms of this equation are the energy associated with the detunings of the driving field frequency and its second harmonic with the modes of the cavity. The third term is the input driving field with amplitude f (equal to E in the classical case), and the fourth term is the coupling between the two modes in the cavity with coupling strength χ.

The corresponding master equation, in standard Linblad form, is given by

$$\dot{\rho_S} = -\frac{i}{\hbar}[H, \rho_S] + \left(L_1\rho_S L_1^\dagger - \frac{1}{2}L_1^\dagger L_1\rho_S - \frac{1}{2}\rho_S L_1^\dagger L_1\right) + \left(L_2\rho_S L_2^\dagger - \frac{1}{2}L_2^\dagger L_2\rho_S - \frac{1}{2}\rho_S L_2^\dagger L_2\right), \tag{2.4}$$

with the Hamiltonian H as given in Eq. 2.3, ρ_S the density operator of the system and the Linblad operators L given by

$$\begin{aligned} L_1 &= \sqrt{2\kappa_1}a_1 \\ L_1^\dagger &= \sqrt{2\kappa_1}a_1^\dagger \\ L_2 &= \sqrt{2\kappa_2}a_2 \\ L_2^\dagger &= \sqrt{2\kappa_2}a_2^\dagger. \end{aligned} \tag{2.5}$$

Here the κ's are the damping of the two modes of the system. The density operator ρ_S contains the maximal amount of information about the system and as such its evolution (as described in Eq. 2.4) contains the evolution of the system. This evolution can be simulated via a model programmed on a computer, providing an understanding of the system and its dynamics.

2.1.3 Transition from Quantum Mechanics to Classical Mechanics in Second Harmonic Generation

As discussed in Sec. 1.1.3, one method to undergo the transition from quantum mechanics to classical mechanics is through the setting of $\hbar$ to zero. In the case of second harmonic generation (as described in Sec. 2.1.2) this approximation does not change the solutions to Eq. 2.4 (as the $\hbar$'s cancel). This is the reason alluded to in Sec. 1.1.3, for choosing the limit as the photon number n tends to infinity, $n \to \infty$, to investigate the quantum/classical interface.

The easiest way to do this is to increase the amplitude of the driving field f, in Eq. 2.3, relative to the other parameters. Physically this increases the number of photons entering the cavity Fig. 2.1 while keeping the loss rates and coupling between the modes constant.

By using already known regimes of the classical system [Zheng 1995] [Savage 1983], under the assumption that the quantum system tends toward the classical system, the regions of interest (stable points, limit cycles and chaos) are exploited and studied in the quantum system.

2.2 Open Systems and Second Harmonic Generation

As discussed in Sec. 1.2, the effect of the environment on a quantum optics system can be very profound. In the case of second harmonic generation, there are five ways the environment effects the model being considered:

- Through the input field ($i\hbar f(a_1^\dagger - a_1)$ in Eq. 2.3)
- Via the output field for ω_1 (L_1 in Eq. 2.4)
- Via the output field for ω_2 (L_2 in Eq. 2.4)
- Random vacuum fluctuations in the input/output field ω_1
- Random vacuum fluctuations in the output field ω_2

These combine to cause the state vector to become localised. The evolution of this localised state vector and the role the environment plays in this evolution is not fully understood. By computer simulation of such systems and observing how the computer generated results change as the control parameters are varied, the effects of the environment are studied and a greater understanding is obtained.

The environment is sometimes said to make a system less quantum in nature, forcing it to behave more like a classical system. Through the use of the environment (and computer simulations), interesting classical phenomena such as chaos and hyper-sensitivity to perturbation are investigated.

2.3 Chaos in Second Harmonic Generation

The emergence of chaos in second harmonic generation is well understood in the classical equations [Savage 1983]. The emergence of chaos in the quantum equations (operating in the large photon number regime) has been alluded to [Zheng 1995], but due to the time taken to attain numerical solutions, never studied in great detail.

The emergence of chaos in second harmonic generation is most easily understood in the context of the classical equations Eq. 2.1 and Eq. 2.2. When these equations are operated in the chaotic regime the output field amplitudes, α_1 and α_2, vary in a chaotic way. The same cannot be said for the quantum equations. There is no amplitude of a mode, there is merely a density operator ρ_S which describes the average evolution of the system.

To visualise the chaotic behavior in a quantum system, it is necessary to measure an observable such as photon number $\langle\psi|a^\dagger a|\psi\rangle$ or position in the complex α plane $\langle\psi|a|\psi\rangle$. In the case of position in the complex α plane, it is hypothesised that the evolution followed by the quantum system asymptotes to that of the classical system evolved with the same parameters.

2.3.1 Classical Chaos in Second Harmonic Generation

The classical equations for second harmonic generation as given in Eq. 2.1 and Eq. 2.2 are derived from Maxwell's equations in the slowly varying envelope and mean-field approximations. The parameters/variables α, κ, χ and δ represent the *real* parameters affecting the system. The physics of the system is in how the solution varies as the ratio between the parameters and variables change. For this purpose, the equations are reformulated in terms of dimensionless quantities [Zheng 1995]

$$\frac{d\tilde{\alpha}_1}{d\tau} = \tilde{E} - (i\tilde{\delta}_1 + 1)\tilde{\alpha_1} + \tilde{\alpha}_1^*\tilde{\alpha_2} \tag{2.6}$$

$$\frac{d\tilde{\alpha}_2}{d\tau} = -(i\tilde{\delta}_2 + \tilde{\kappa})\tilde{\alpha_2} - \frac{1}{2}\tilde{\alpha}_1^2, \tag{2.7}$$

where

$$\tilde{\alpha}_i = \frac{\chi}{\kappa_1}\alpha_i \tag{2.8}$$

$$\tilde{E} = E\frac{\chi}{\kappa_1^2} \tag{2.9}$$

$$\tilde{\kappa} = \frac{\kappa_2}{\kappa_1} \tag{2.10}$$

$$\tilde{\delta}_i = \frac{\delta_i}{\kappa_1} \tag{2.11}$$

$$\tau = t\kappa_1. \tag{2.12}$$

These scaled equations depend on the scaled quantities $\tilde{E}$, $\tilde{\delta}_i$ and $\tilde{\kappa}$ only. By varying these parameters the same dynamics can be studied as for the unscaled equations, except that changes in these parameters now directly correspond to changes in the physics of the system.

2.3.2 Quantum Chaos in Second Harmonic Generation

The quantum equations for second harmonic generation, Eq. 2.3 to Eq. 2.5, can be scaled using the same substitutions as in the classical case. As long as the scaled parameters/variables are fixed, varying the ratio χ/κ_1 does not change the classical scaled amplitudes $\tilde{\alpha}_1$ and $\tilde{\alpha}_2$. This ratio is called the scaling parameter S

$$S = \frac{\chi}{\kappa_1}. \tag{2.13}$$

By decreasing S the driving amplitude $E = \tilde{E}\ \kappa_1/S$ is increased, while keeping the classical scaled amplitudes $\tilde{\alpha}_1$ and $\tilde{\alpha}_2$ constant. With the quantum equations formulated using the same scaled variables, decreasing S is equivalent to increasing the number of photons in the system. This allows the quantum/classical interface to be explored and compared to the single classical solution. From the solution of the quantum equations, the density operator ρ_S can be obtained and analysed to see how it changes at various stages toward the classical limit.

The units of S are $ms^{-1}V^{-1}$. They are not included in any values given in this Thesis, as it is how the system changes as S decreases that is of interest, not its value.

It is unclear how the structure of the density operator will change when the quantum equations are operated in the regime where the classical equations show chaos. As alluded to in Sec. 1.3.2 defining chaos for quantum systems is difficult, leading to further problems in understanding chaos in second harmonic generation. If the classical limit cycles can be observed using the quantum system, then running the quantum system in the regime where the classical equations show chaos will hopefully lead to the observation of quantum chaos.

Hyper-sensitivity to Perturbations

Another method for determining whether a quantum system is chaotic is to see whether it is hyper-sensitive to perturbations [Schack 1996 a], which requires only the notions of entropy and information. Since entropy and information can be defined for both a quantum system and a classical system the information theory definition of chaos allows a consistent rule to be a applied.

To define hyper-sensitivity to perturbations, consider a system S coupled to a perturbing environment. If a measurement can be defined on the environment which can only resolve the perturbation to some accuracy δ, then the outcome will be one of r possibilities, each with

probability p_i, where i represents the outcome number. From this measurement the system ends up in one of ρ_i states. The ρ_i states have the property that

$$\sum_{i=1}^{r} p_i \rho_i = \rho_S, \tag{2.14}$$

where ρ_S is the density operator of the system where no measurement is made of the perturbing environment. Associated with each of these states ρ_i is a change in entropy of the system

$$\Delta H_i = -Tr_S(\rho_i \log_2 \rho_i). \tag{2.15}$$

So the average change in entropy of the system is

$$\Delta H = \sum_{i=1}^{r} p_i \Delta H_i. \tag{2.16}$$

Now the average change in information obtained about the environment when making a measurement is defined as

$$\Delta I = -\sum_{i=1}^{r} p_i \log_2 p_i. \tag{2.17}$$

This process is shown graphically in Fig. 2.3. The average change in entropy in this example is then given by $\Delta H = -p_1\ Tr_S(\rho_1 \log_2 \rho_1) - p_2\ Tr_S(\rho_2 \log_2 \rho_2) - p_3\ Tr_S(\rho_3 \log_2 \rho_3) - \cdots - p_r\ Tr_S(\rho_r \log_2 \rho_r)$, which is less than $-Tr_S(\rho_S \log_2 \rho_S)$ due to Eq. 2.14. This demonstrates that by specifying information about the perturbing environment the average change in entropy of the system decreased.

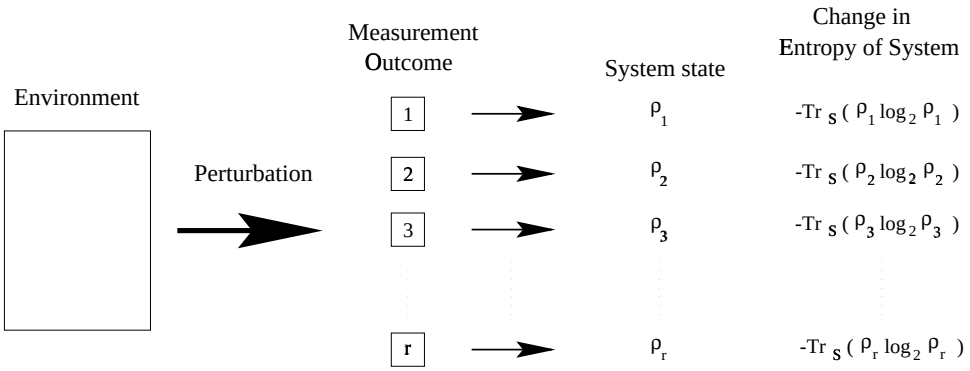

Figure 2.3: Diagrammatic representation of how making a measurement of the environment reduces the average change in entropy of the system

So, to determine whether a system is hyper-sensitive to perturbations, it is necessary to know the minimum average information ΔI_{min} that needs to be known about the environment to reduce the change in system entropy $\Delta H_S = -Tr_S(\rho_S \log_2 \rho_S)$ to some tolerable level ΔH_{Tol}. A system which is not hyper-sensitive (regular system) to perturbations will require very little information about the perturbation to reduce the change in system entropy to the tolerable level. A system which is hyper-sensitive to perturbations (chaotic system) will require roughly a constant amount of information about the perturbation to reduce the change in system entropy to some tolerable amount. For a graphical representation see Fig. 2.4.

The label A refers to the point where ΔH_{Tol} is so small that the maximal information ΔI_{min} needs to be known about the environment. This is effectively following the evolution of the system at every perturbation from the environment. The label B refers to the point

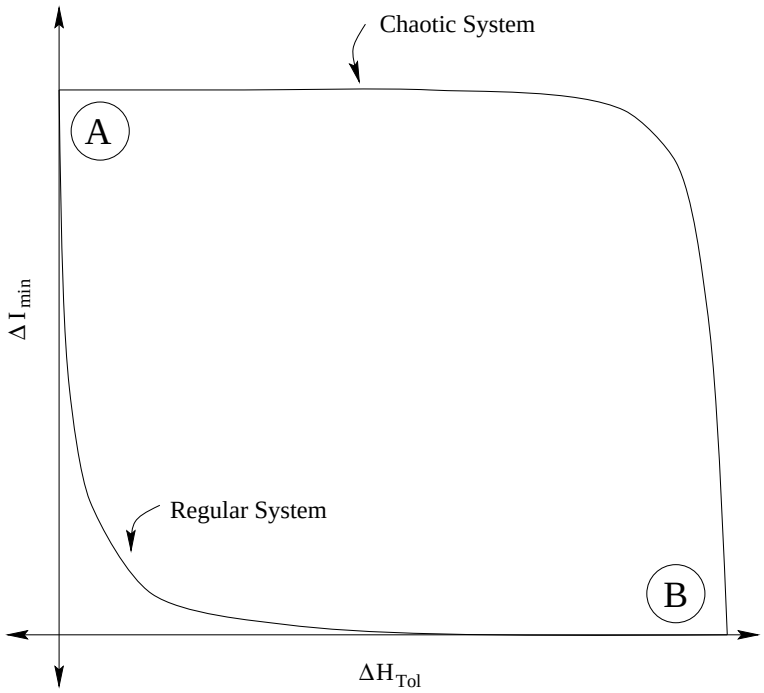

Figure 2.4: Diagrammatic representation of the approximate functional dependence of the minimum information, ΔI_{min}, about a perturbation required to reduce the change in system entropy, ΔH_S, to some tolerable amount, ΔH_{Tol}, for a chaotic and regular system.

where ΔH_{Tol} is so large that no information needs to be known about the environment. This corresponds to the point where the change in entropy of the system due to the perturbing environment is maximal and less than that of ΔH_{Tol}.

To calculate the relationship between ΔH_{Tol} and ΔI_{min} the average change in conditional entropy Eq. 2.15 is approximately equal to the tolerable change desired $\Delta H_{Tol} \approx \Delta H$, with the minimum information approximately equal to the average information known about the environment Eq. 2.17, $\Delta I_{min} \approx \Delta I$. By plotting ΔH against ΔI the approximate functional dependence of ΔH_{Tol} and ΔI_{min} will be shown. As alluded to by Schack *et al* a chaotic system should demonstrate relatively large ΔI_{min} with a regular system demonstrating relatively small ΔI_{min}. That is, a chaotic system (a system that is hyper-sensitive to perturbations) requires the perturbation to be highly specified in order to reduce the system entropy, while a regular system (a system that does not show hyper-sensitivity to perturbations) does not need the perturbation specified to the same degree.

2.3.3 From Quantum Optics to Classical Chaos in Second Harmonic Generation

It is expected that the Q-function, $Q(\alpha) = \langle\alpha|\rho_S|\alpha\rangle$, of the density operator ρ_S will show support on the classical limit cycle as the quantum system is made classical. The width of the distributions is also expected to decrease indicating that the quantum solution is tending toward the classical solution in the approximation as photon number $n \to \infty$. This is in contrast to what might naively be expected, that being a wave packet traveling around the classical limit cycle.

The reason that the solution cannot be a Gaussian traveling around the limit cycles, is due to the long time solution of the master equation Eq. 2.4 being a steady state solution, again due to the master equation having no positive real Lyapunov exponents [Haken 1983] (pp.

4.158). This excludes the possibility of a moving Gaussian, as it is a time dependent solution. This means that the only reasonable solution is as outlined above. Successive measurements of the system will, however, result in different points in the α plane (if $\langle\psi|a|\psi\rangle$ is measured) being observed. Relevant plots of Q-functions, accompanied by their physical significance, are given in Ch. 4.

Chapter 3

Solving the Master Equation for Second Harmonic Generation

The numerical tools for solving the quantum state diffusion equation are developed in this section. An introduction to the quantum master equation and difficulties in numerically solving it are covered in Sec. 3.1. Typical numerical methods used to solve the quantum master equation are dealt with in Sec. 3.2, with their equivalence to the master equation demonstrated. The quantum state diffusion equation for second harmonic generation is introduced in Sec. 3.3. Numerical detail about solving it is provided along with explicit detail about implementing the moving basis. The final two sections Sec. 3.4 and Sec. 3.5 provide a recipe for solving the master equation, via the quantum state diffusion method, with detail provided about the analysis performed on the data obtained.

3.1 The Master Equation

The master equation is an equation which gives the evolution of the reduced density operator ρ_S of the system. The master equation in standard form is obtained by making a Markoff approximation: the reservoir has a constant spectrum across the system bandwidth; and that the coupling of the reservoir is weak and frequency independent across the bandwidth of the system.

The reduced density operator of the system ρ_S is obtained by tracing over the states of the reservoir of the complete density operator ρ, $\rho_S = Tr_{Reservoir}(\rho)$. Usually the complete density operator is not needed as it is the dynamics of the system that are of interest, not that of the reservoir. This is keeping with the discussion given in Sec. 1.2.

The general form of the master equation is

$$\dot{\rho_S} = \frac{i}{\hbar}[\rho_S, H_s] + \mathcal{L}\rho_S, \tag{3.1}$$

where the assumed form for $\mathcal{L}\rho_S$ is

$$\mathcal{L}\rho_S = C\rho_S C^\dagger - \frac{1}{2}C^\dagger C\rho_S - \frac{1}{2}\rho_S C^\dagger C. \tag{3.2}$$

The C's represent an evolution operator in quantised form for the system under consideration. For second harmonic generation, the master equation is given in Eq. 2.4 with the Linblad operators given in Eq. 2.5.

For more detail see [Meystre 1990].

3.1.1 Analytic solution of the Master Equation

A solution of the master equation provides all the information about the evolution of the system to accurately predict its behavior. An analytic solution would provide the greatest understanding of the system, how its evolution is affected by the various parameters and insight into the *physics* of the system. Unfortunately, analytic solutions of the master equation are hard to obtain and have only been done for relatively simple systems such as the harmonic oscillator.

Currently, no known analytic solution of the master equation exists for second harmonic generation. Without such a solution, numerical simulations become relevant and very important as the device used to study second harmonic generation and other systems more complicated than the harmonic oscillator.

3.1.2 Numerical solutions of the Master Equation

The most naive way to numerically solve the master equation is to use a standard numerical integration scheme such as Euler or Runge-Kutta. Unfortunately these prove very expensive on computer memory and time. To see why this is the case, consider the reduced density operator

$$\begin{aligned} \rho_S &= |\psi\rangle\langle\psi| \\ &= \sum_{i=0}^{N}\sum_{j=0}^{N} c_i c_j^* |b_i\rangle\langle b_j|, \end{aligned} \tag{3.3}$$

where N represents an approximation to the size of the basis. This approximation is made as computers cannot deal with an infinite number of basis states. It assumes that for some N the amplitude $c_{i,j}$ are small for all $i, j > N$.

This shows that the density operator has N^2 elements and so solving the master equation Eq. 3.1 is equivalent to solving N^2 coupled differential equations. Also, to add to the problems, it requires at least N^2 complex numbers to be stored.

In the case of second harmonic generation the state vector $|\psi\rangle$ is expressed as (using the number-state basis)

$$|\psi\rangle = \sum_{i=0}^{N_1}\sum_{j=0}^{N_2} c_{i,j}|i\rangle|j\rangle, \tag{3.4}$$

where the sum of i is over the basis states of the first mode of the system and j being over the second mode.

This means that the reduced density operator ρ_S has $N_1^2 * N_2^2$ elements. Now, in the case of [Zheng 1995] the basis sizes for each mode of the cavity were of the order of 500, meaning that approximately $2 * 500^4$ real numbers would have to be stored (corresponding to approximately 930gigabytes of memory) which is beyond the means of computers currently available, let alone the time required to solve such a system.

Thus, while analytic solution are difficult to obtain, directly solving the master equation also leads to unrealisable solutions. One method to overcome this is the numerical method known as *Quantum Trajectories*.

3.2 Quantum Trajectory solutions of the Master Equation

The quantum trajectory method for solving the master equation was developed to overcome the limitations of directly solving the master equation (discussed previously). It involves

the evolution of a single state vector $|\psi\rangle$ as opposed to a density operator ρ_S. By taking an ensemble of state vectors, forming their respective reduced density operator and then averaging the reduced density operators, the solution to the master equation is reconstructed.

The quantum trajectory methods are often likened to what is *physically* happening in the quantum system. Take for example the method of quantum jumps. It evolves the system according to a deterministic equation for some time, then makes a random jump to a predetermined state. In this way, it is often compared to making a measurement of the system. While these sort of analogies are drawn, the quantum trajectory methods are *mathematical* unravellings of the master equation and require no physical interpretation to justify them.

A common numerical method for solving the master equation is that of quantum jumps [Carmichael 1993]. Recently, another method has been developed called *Quantum State Diffusion* and a modification of this called *Quantum State Diffusion with a Moving Basis* [Schack 1995].

3.2.1 Quantum Jumps

The method of quantum jumps is to evolve the system for a time interval (random in length) and make a jump to a predetermined state. The equation for this method, in standard Linblad form, is given by

$$|d\psi(t)\rangle = -\frac{i}{\hbar}H|\psi(t)\rangle dt - \frac{1}{2}\sum_j \left(L_j^\dagger L_j - \langle L_j^\dagger L_j\rangle\right)|\psi\rangle dt + \sum_j \left(\frac{L_j|\psi(t)\rangle}{\sqrt{\langle L_j^\dagger L_j\rangle}} - |\psi\rangle(t)\right) dN_j. \tag{3.5}$$

Here the discrete Poissonian noises dN_j assumes the value 0 or 1. When dN_j is 0 the evolution is continuous and differentiable. When dN_j is 1 there is a *jump* to the state

$$\frac{L_j|\psi(t)\rangle}{\sqrt{\langle L_j^\dagger L_j\rangle}}. \tag{3.6}$$

The dN_j process have mean $M[dN_j] = \langle L_j^\dagger L_j\rangle dt$ with correlations $dN_j dt = 0$ and $dN_j dN_k = \delta_{jk} dN_j$ [Brun 1996] (where M signifies an ensemble average).

3.2.2 Quantum State Diffusion

The method of quantum state diffusion is to evolve the system according to a stochastic differential equation. During this process a random fluctuation is introduced at every time step. This method is often understood in the context of the fluctuations that the environment inputs into the evolution of the system (as opposed to measurements being made, like in the quantum jumps). The equation for this method, in standard Linblad form, is given by

$$|d\psi(t)\rangle = -\frac{i}{\hbar}H|\psi(t)\rangle dt - \frac{1}{2}\sum_j \left(L_j^\dagger L_j - 2\langle L_j^\dagger\rangle L_j + |\langle L_j\rangle|^2\right)|\psi(t)\rangle dt + \sum_j \left(L_j - \langle L_j\rangle\right)|\psi(t)\rangle d\xi_j. \tag{3.7}$$

Here the *noises* $d\xi_j$ are complex-valued Wiener processes of mean $M[d\xi_j] = 0$ with correlations $M[d\xi_j d\xi_k] = 0$ and $M[d\xi_j^* d\xi_k] = \delta_{jk} dt$. The complex Wiener process is normalised to dt for quantum state diffusion. This equation evolves the state vector $|\psi\rangle$ in a continuous non-differentiable way. It is the only continuous unraveling of the master equation which satisfies the same symmetry properties as the master equation [Brun 1996]

Quantum State Diffusion with a Moving Basis

The modification of this process known as Quantum State Diffusion with a Moving Basis uses one further property of open systems to decrease the computational time required to solve these systems: localisation of the state vector. The two quantum trajectory methods discussed both require the basis, which the system is described in terms of, to be large enough to contain the entire evolution of the system. If the basis is of size N, then solving the system is equivalent to solving N coupled differential equations. This is computationally intense, requiring large amounts of computer time [Zheng 1995].

In an attempt to decrease the size of N, the localisation of the state vector becomes important. Using quantum state diffusion it is possible to "move" the basis to follow the evolution of the state vector, and then evolve the state vector on this new shifted basis for a while, then shift the basis again. In this way, the basis size can be decreased (only need to have a basis large enough to cover the state vector, not the entire trajectory) at the expense of increased computational effort. It must be remembered, however, that this method only works for localised state vectors.

Quantum state diffusion with a moving basis is the numerical model that is used to solve the second harmonic generation system in this thesis.

3.2.3 Equivalence to the Master Equation

The ensemble of solutions obtained from the quantum state diffusion equation give the same solution as the master equation. This is demonstrated by taking the average of the reduced density operators of the state vectors obtained by the quantum state diffusion equation

$$\rho_S = M[\rho_{|\psi\rangle}] = M\left[|\psi\rangle\langle\psi|\right], \tag{3.8}$$

Where, again, M signifies an ensemble average. Since the quantum state diffusion equation is a stochastic differential equation Sec. 3.3.2, the second order terms in the derivative must be maintained when computing $d\rho_S$

$$\begin{aligned} d\rho_S &= M\left[|d\psi\rangle\langle\psi| + |\psi\rangle\langle d\psi| + |d\psi\rangle\langle d\psi|\right] && (3.9)\\ &= M\Bigg[-\frac{i}{\hbar}H\rho_{|\psi\rangle}dt - \frac{1}{2}\sum_j \left(L_j^\dagger L_j - 2\langle L_j^\dagger\rangle L_j + |\langle L_j\rangle|^2\right)\rho_{|\psi\rangle}dt + \sum_j \left(L_j - \langle L_j\rangle\right)\rho_{|\psi\rangle}d\xi_j \\ &+ \frac{i}{\hbar}\rho_{|\psi\rangle}Hdt - \frac{1}{2}\rho_{|\psi\rangle}\sum_j \left(L_j^\dagger L_j - 2\langle L_j^\dagger\rangle L_j + |\langle L_j\rangle|^2\right)^\dagger dt + \rho_{|\psi\rangle}\sum_j \left(L_j - \langle L_j\rangle\right)^\dagger d\xi_j^* \\ &+ \sum_{i,j}\left(L_j - \langle L_j\rangle\right)\rho_{|\psi\rangle}\left(L_i - \langle L_i\rangle\right)^\dagger d\xi_j d\xi_i^*\Bigg], && (3.10) \end{aligned}$$

where terms of order $> dt$ are set to zero. Due to the linearity of the L operators and using Eq. 3.8 with the properties of the Wiener process Sec. 3.2.2, equation Eq. 3.10 becomes

$$\begin{aligned} d\rho_S &= -\frac{i}{\hbar}[H,\rho_S]dt - \frac{1}{2}\sum_j \left(L_j^\dagger L_j - 2\langle L_j^\dagger\rangle L_j + |\langle L_j\rangle|^2\right)\rho_S dt - \frac{1}{2}\rho_S\sum_j \left(L_j^\dagger L_j - 2\langle L_j\rangle L_j^\dagger + |\langle L_j\rangle|^2\right)dt \\ &+ \sum_j \left(L_j - \langle L_j\rangle\right)\rho_S\left(L_j^\dagger - \langle L_j^\dagger\rangle\right)dt && (3.11)\\ &= -\frac{i}{\hbar}[H,\rho_S]dt + \sum_j \left(L_j\rho_S L_j^\dagger - \frac{1}{2}L_j^\dagger L_j\rho_S - \frac{1}{2}\rho_S L_j^\dagger L_j\right)dt. && (3.12) \end{aligned}$$

That is

$$\dot{\rho_S} = -\frac{i}{\hbar}[H, \rho_S] + \sum_j \left(L_j \rho_S L_j^\dagger - \frac{1}{2} L_j^\dagger L_j \rho_S - \frac{1}{2} \rho_S L_j^\dagger L_j \right). \tag{3.13}$$

So, the quantum state diffusion equation is equivalent to the master equation, and as such, from the solutions to the quantum state diffusion equation the solution to the master equation can be reconstructed via Eq. 3.8

3.3 Quantum State Diffusion equation for Second Harmonic Generation

The master equation for second harmonic generation is given in Eq. 2.3 to Eq. 2.5. As stated above, the method used to solve the master equation in this thesis is that of quantum state diffusion. The equation for quantum state diffusion in its general form is given in Eq. 3.7. In the context of second harmonic generation, Eq. 2.3 to Eq. 2.5, the quantum state diffusion equation becomes

$$\begin{aligned} |d\psi(t)\rangle &= -iH|\psi(t)\rangle dt - \frac{1}{2}\left(L_1^\dagger L_1 - 2\langle L_1^\dagger \rangle L_1 + |\langle L_1 \rangle|^2 + L_2^\dagger L_2 - 2\langle L_2^\dagger \rangle L_2 + |\langle L_2 \rangle|^2 \right) |\psi(t)\rangle dt \\ &+ \left(L_1 - \langle L_1 \rangle \right) |\psi(t)\rangle d\xi_1 + \left(L_2 - \langle L_2 \rangle \right) |\psi(t)\rangle d\xi_2, \end{aligned} \tag{3.14}$$

with

$$H = \hbar\delta_1 a_1^\dagger a_1 + \hbar\delta_2 a_2^\dagger a_2 + i\hbar f(a_1^\dagger - a_1) + i\hbar\frac{\chi}{2}(a_1^{\dagger\,2} a_2 - a_1^2 a_2^\dagger) \tag{3.15}$$

and

$$\begin{aligned} L_1 &= \sqrt{2\kappa_1} a_1 \\ L_1^\dagger &= \sqrt{2\kappa_1} a_1^\dagger \\ L_2 &= \sqrt{2\kappa_2} a_2 \\ L_2^\dagger &= \sqrt{2\kappa_2} a_2^\dagger. \end{aligned} \tag{3.16}$$

In order to numerically solve this equation, it is necessary to have a representation for the state $|\psi\rangle$. Since second harmonic generation deals with two modes of an harmonic oscillator, it is convenient to use the number-state basis for each oscillator and form the state vector as

$$|\psi\rangle = \sum_{i,j} c_{i,j} |i\rangle |j\rangle, \tag{3.17}$$

where the i is the index for first mode of the system and j for the second mode. Using this formalism the Linblad operators only operate on their respective modes, for example

$$\begin{aligned} L_1|\psi\rangle &= L_1 \sum_{i,j} c_{i,j} |i\rangle |j\rangle \\ &= \sqrt{2\kappa_1} a_1 \sum_{i,j} c_{i,j} |i\rangle |j\rangle \\ &= \sqrt{2\kappa_1} \sum_{i,j} c_{i,j} a_1 |i\rangle |j\rangle \\ &= \sqrt{2\kappa_1} \sum_{i,j} c_{i,j} \sqrt{i} |i-1\rangle |j\rangle. \end{aligned} \tag{3.18}$$

3.3.1 Numerical Considerations

The representation for the state vector given in Eq. 3.17 is a sum over all the modes of two harmonic oscillators. To implement this on a computer, it must be assumed that this infinite sum can be approximated by a finite sum. Since open systems tend to be localised it is assumed that this approximation is valid.

For the simulation the approximate state vector is given by

$$|\psi\rangle = \sum_{i=0}^{N_2}\sum_{j=0}^{N_2} c_{i,j}|i\rangle|j\rangle. \tag{3.19}$$

In making this approximation it is necessary to assume that the Lindblad operators operate on the boundary points such that the terms $a_1^\dagger|N_1\rangle$ and $a_2^\dagger|N_2\rangle$ are set to zero. Precautions must also be taken to ensure that the amplitude $|c_{i,j}|$ is relatively small for $i = N_1$ or $j = N_2$ (this makes sure that the number of basis states is large enough to accurately represent the state vector).

By substituting Eq. 3.19 into Eq. 3.14 the equation for the evolution of the co-efficients $c_{n,m}$ is obtained

$$\begin{aligned} dc_{n,m} &= -i\left(\delta_1 n + \delta_2 m\right)\, c_{n,m}\, dt + f\left(\sqrt{n}\, c_{n-1,m} - \sqrt{n+1}\, c_{n+1,m}\right) \\ &+ \frac{\chi}{2}\left(\sqrt{n(n-1)(m+1)}\, c_{n-2,m+1} - \sqrt{(n+1)(n+2)m}\, c_{n+2,m-1}\right) \\ &- \kappa_1\left(n - 2\langle a_1^\dagger\rangle\sqrt{n+1}\, c_{n+1,m} + |\langle a_1^\dagger\rangle|^2\right) dt - \kappa_2\left(m - 2\langle a_2^\dagger\rangle\sqrt{m+1}\, c_{n,m+1} + |\langle a_2^\dagger\rangle|^2\right) dt \\ &+ \sqrt{2\kappa_1}\left(\sqrt{n+1}\, c_{n+1,m} + \langle a_1^\dagger\rangle\, c_{n,m}\right) d\xi_1 + \sqrt{2\kappa_2}\left(\sqrt{m+1}\, c_{n,m+1} + \langle a_2^\dagger\rangle\, c_{n,m}\right) d\xi_2, \end{aligned} \tag{3.20}$$

where $dc_{n,m}$ represents the infinitesimal increment in the co-efficient $c_{n,m}$.

3.3.2 Numerical Integration of Quantum State Diffusion Equation

In order to find the numerical solution to Eq. 3.14 some sort of integration scheme needs to be implemented. The quantum state diffusion equation is an example of a stochastic differential equation. Since stochastic differential equations obey different calculus to normal differential equations, a brief overview will be given.

Stochastic Differential Equations

The study of many physical systems is often achieved through the use of a mathematical model based on differential equations. These equations express how the variables change with time and how they interact with each other. Often, approximations are made to produce a solvable system of equations, with the results demonstrating good agreement with the physical system. These models typically deal with quantities such as *position, momentum* and *population* and evolve them in a deterministic way. Often these models do not deal with small quantities assumed to be negligible, or occurring infrequently. For example, consider the motion of a simple pendulum. The differential equation, in one dimension, is given by $-kx = m\ddot{x}$ and assumes no friction, damping or driving. It also does not include random fluctuation such as interaction with gas molecules as it swings or the effect of air currents.

In the example of the pendulum, the model can be made more representative, by incorporating damping, driving and friction, quite easily. The inclusion of the other more subtle variables, interaction with the air etc., are often harder to include and effectively involve a continuous random fluctuation. These fluctuations can be ignored for many systems, but they dominate others. A good example of this is *Brownian Motion*, where a grain of pollen,

placed onto the surface of some water, is jiggled by water molecules. The model of this system is quite different to that of the pendulum and includes a *random differential*. The differential equations that govern the evolution of these models are often known as *Stochastic Differential Equations* (SDE).

The following explanation is based on that given in [Gardiner 1985].

Calculus of Stochastic Differential Equations

The stochastic differential equations used in this thesis obey *Itô* calculus, which means that the usual rules of calculus do not apply see [Gardiner 1985] (Sec. 4.3) and [Wiseman 1994] (Ch. 3). Consider the one dimensional stochastic differential equation (where x is assumed to be a function of time $x(t)$)

$$dx = a(x,t)dt + b(x,t)\xi(t)dt. \tag{3.21}$$

The function $\xi(t)$ is a continuous rapidly varying stochastic fluctuation, pertaining to certain properties. It is often taken to be "white noise", obeying

$$\begin{aligned} M[\xi(t)] &= 0 & (3.22)\\ M[\xi(t)\xi(t')] &= \delta(t-t'). & (3.23) \end{aligned}$$

From this the "Wiener process" is obtained: $dW(t) = \xi(t)dt$. The Wiener process has the property

$$dW(t)^2 = dt. \tag{3.24}$$

So Eq. 3.21 becomes

$$dx = a(x,t)dt + b(x,t)dW(t). \tag{3.25}$$

Now consider an arbitrary function of x: $f(x)$. Now expanding the differential $df(x)$ to second order in $dW(t)$ gives

$$\begin{aligned} df(x) &= f(x+dx) - f(x) & (3.26)\\ &= f'(x)dx + \frac{1}{2}f''(x)dx^2 + \cdot. & (3.27) \end{aligned}$$

Now substituting Eq. 3.25

$$df(x) = f'(x)\left\{a(x,t)dt + b(x,t)dW(t)\right\} + \frac{1}{2}f''(x)b(x,t)^2dW(t)^2 + \cdot. \tag{3.28}$$

Keeping with the original statement of keeping only second order in $dW(t)$ and using Eq. 3.24 Itô's formula is obtained

$$df(x) = \left\{a(x,t)f'(x) + \frac{1}{2}b(x,t)^2f''(x)\right\}dt + b(x,t)f'(x,t)dW(t). \tag{3.29}$$

This is clearly different to that obtained using normal calculus where

$$df(x) = f'(x)\left\{a(x,t) + b(x,t)dW(t)\right\}. \tag{3.30}$$

This demonstrates that to have an equation of order dt terms of order $dW^2 = dt$ need to be kept.

Numerical solution of Stochastic Differential Equation

The numerical methods used to solve stochastic differential equations are similar to those for normal differential equations, with account being taken of Itô's equation Eq. 3.29. This introduces a large number of changes into methods of higher order than an Euler one step method. For this reason the integration method used in this thesis is simply an explicit Euler one step method, which has order $\frac{1}{2}$ [Kloeden 1994] (pp. 157). The introduction of higher order methods would be a good area for more research.

It has been suggested in communications with Todd Brun[1] that the deterministic component of the quantum state diffusion equation can be integrated using a Runge-Kutta algorithm, while integrating the stochastic component using an Euler one step algorithm. In making this approximation, the assumption is made that the stochastic component is not varying rapidly, thus allowing the deterministic component to evolve for a small time in a deterministic fashion. It was felt that this was not the case for second harmonic generation and as such was not implemented. This caution was strengthened by comments by Kloeden and Platten [Kloeden 1992] (pp. 386).

The Euler one step method is performed by approximating dt by Δt_i, dW by ΔW_i and dx by $x_{i+1} - xi$, where the subscript i represents the iteration step. Making these substitutions into Eq. 3.25 gives

$$x_{i+1} = x_i + a(x_i, t)\Delta t_i + b(x_i, t)\Delta W_i. \tag{3.31}$$

Graphically the process is shown in Fig. 3.1.

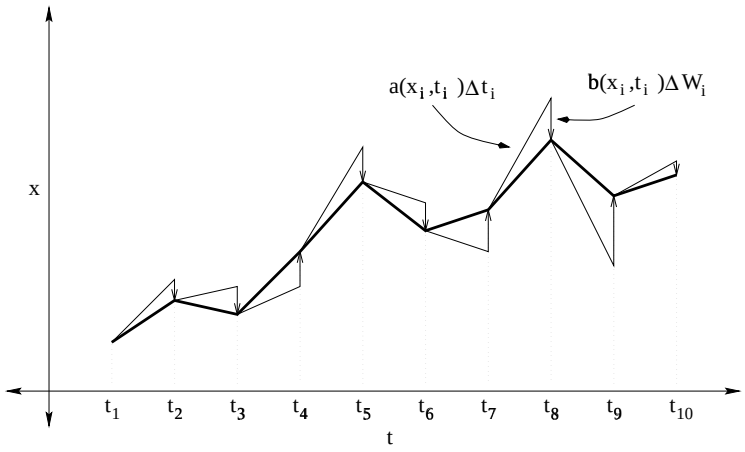

Figure 3.1: Illustration of the Cauchy-Euler procedure for constructing an approximate solution to the stochastic differential equation Eq. 3.25 [Gardiner 1985]. The subscripted numbers on the time represent the iteration step.

The discrete ΔW_i represent a complex Wiener process with properties

$$M[\Delta W_i] = 0 \tag{3.32}$$
$$M[\Delta W_i^2] = \Delta t_i \tag{3.33}$$
$$M[(\Delta W_i^2 - \Delta t_i)^2] = 2\Delta t_i^2. \tag{3.34}$$

As is demonstrated in Eq. 3.31 the integration procedure is quite simple and easily implemented on a computer. The fact that this solution is only of order $\frac{1}{2}$ is its only draw

[1] Department of Physics, Queen Mary and Westfield College University of London, London E1 4NS, England

back. This means that for an increase in the accuracy δ, giving rise to smaller time step and more iterations, the computational time required grows as δ^2 (quite rapidly).

3.3.3 The Moving Basis

The equation for the stochastic evolution of the state vector $|\psi\rangle$ is given in Eq. 3.14. As the stochastic integration proceeds, the localised state vector will evolve and follow a trajectory in the α plane. The trajectory is obtained by taking the expectation value $\alpha_1 = \langle\psi|a_1|\psi\rangle$ for mode 1 and $\alpha_2 = \langle\psi|a_2|\psi\rangle$ for mode 2 at various integration steps. Now the basis is shifted to the position (α_1, α_2) and taken large enough for the state vector to evolve at least one integration step upon this basis. In this way, the basis will follow the trajectory with only a basis large enough to evolve the integration a small number of steps at a time. This has effectively decreased the number of coupled differential equations that need to be solved.

A diagrammatic representation of this scheme is given in Fig. 3.2

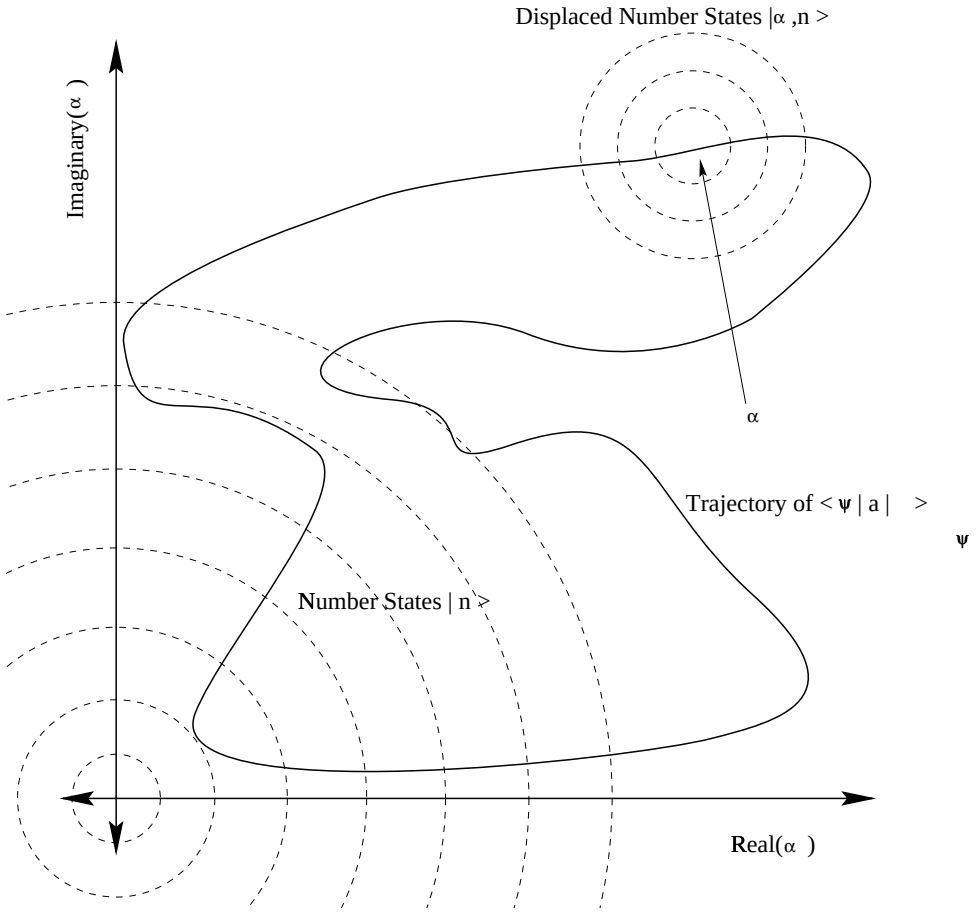

Figure 3.2: A diagrammatic representation of the moving basis scheme. The vertical axis is the real part of $\langle\psi|a|\psi\rangle$ and the horizontal axis the imaginary part.

Displaced Number States

In order to perform the moving basis, a basis is needed that has a parameter that can follow the trajectory. $|\alpha, n\rangle$ is called a displaced number state (a number state $|n\rangle$ displaced by a coherent state $|\alpha\rangle$) [de Oliveira 1990]. To gain a better idea, consider the Q-function of a number state $|n\rangle$

$$
\begin{aligned}
Q(\alpha) &= \frac{1}{\pi}\left|\langle\alpha|n\rangle\right|^2 \\
&= \frac{1}{\pi}\left|e^{-|\alpha|^2/2}\sum_i \frac{(\alpha^*)^i}{\sqrt{i!}}\langle i|n\rangle\right|^2 \\
&= \frac{1}{\pi}\left|e^{-|\alpha|^2/2}\frac{(\alpha^*)^n}{\sqrt{n!}}\right|^2 .
\end{aligned} \tag{3.35}
$$

A plot of this function with $n = 10$ is given in Fig. 3.3

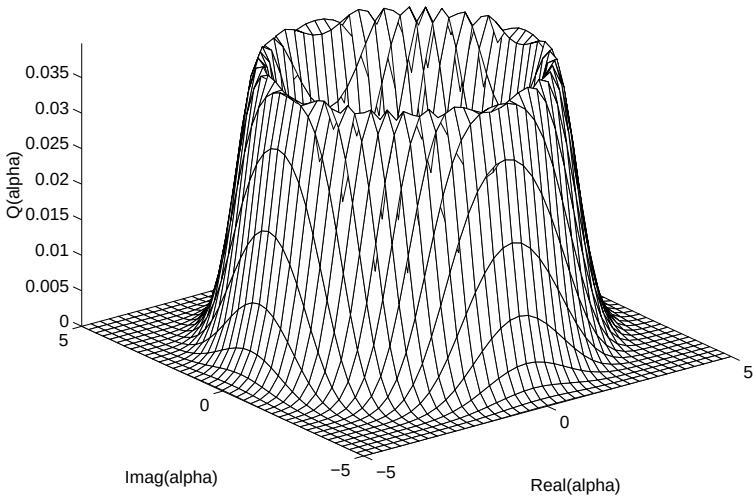

Figure 3.3: Q-function plot for the number state $|10\rangle$

The Q-function of a coherent state $|\alpha_o\rangle$ is

$$
\begin{aligned}
Q(\alpha) &= \frac{1}{\pi}\left|\langle\alpha|\alpha_o\rangle\right|^2 \\
&= \frac{1}{\pi}\left|e^{-|\alpha|^2/2}\sum_i \frac{(\alpha^*)^i}{\sqrt{i!}}e^{-|\alpha_o|^2/2}\sum_j \frac{(\alpha_o^*)^j}{\sqrt{j!}}\langle i|j\rangle\right|^2 \\
&= \frac{1}{\pi}\left|e^{-|\alpha-\alpha_o|^2}\right|^2 .
\end{aligned} \tag{3.36}
$$

A plot of this function with $\alpha_o = 10 + 10i$ is Fig. 3.4

As demonstrated in Eq. 3.35, Eq. 3.36, Fig. 3.3 and Fig. 3.4 the number states are rings about the origin in the complex α plane, while coherent states are a mound centered about a spot on the complex α plane. The displaced number states are rings centered about a coherent state in the complex α plane. By using this as a basis, the basic properties of the number states are preserved, while allowing them to follow the trajectory.

The displaced number states obey the following relationships

$$
|\alpha, n\rangle = D(\alpha)|n\rangle \tag{3.37}
$$

and

$$
a|\alpha, 0\rangle = \alpha|\alpha, 0\rangle, \tag{3.38}
$$

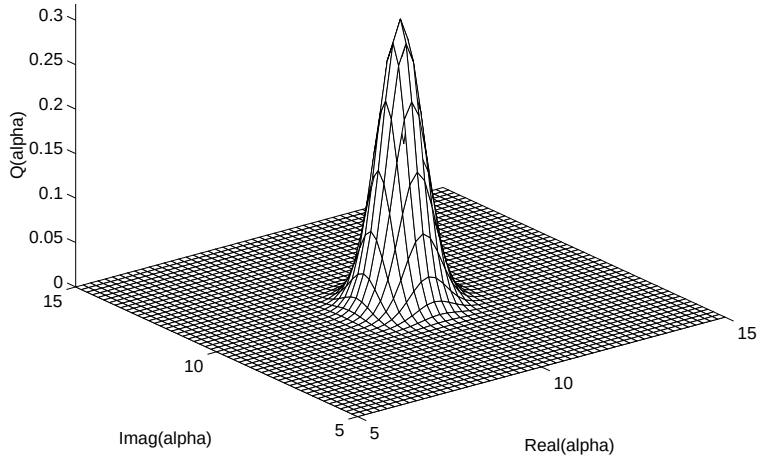

Figure 3.4: Q-function plot for a coherent state $|10+10i\rangle$

where a is the annihilation operator and

$$D(\alpha) = e^{\alpha a^\dagger - \alpha^\star \, a}. \tag{3.39}$$

The displacement operator $D(\alpha)$ can be shown to obey the relationships (using Eq. 3.57)

$$D^\dagger(\alpha) = D(-\alpha) \tag{3.40}$$

$$D^\dagger(\alpha)D(\beta) = e^{\frac{1}{2}\{\alpha^\star\beta - \alpha\beta^\star\}}D(\beta-\alpha). \tag{3.41}$$

The Q-function of the displaced number state $|\alpha_o, n\rangle$ is [de Oliveira 1990]

$$Q(\alpha) = \frac{1}{\pi}e^{-|\alpha-\alpha_o|^2}\frac{|\alpha_o-\alpha|^{2n}}{n!}. \tag{3.42}$$

A plot of this function with $n = 10$ and $\alpha_o = 10 + 10i$ is given in Fig. 3.5

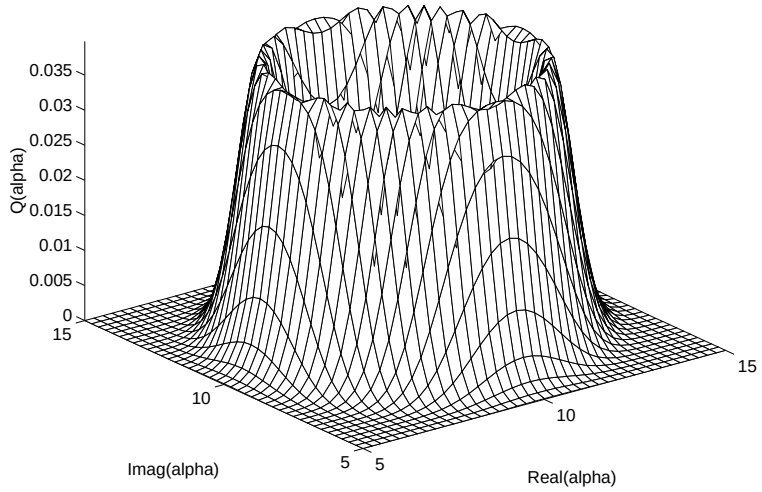

Figure 3.5: Q-function plot for the displaced number state $|10+10i, 10\rangle$

By comparing Fig. 3.3 to Fig. 3.5 the shifting of the number state $|10\rangle$ to the coherent state $|10+10i\rangle$ Fig. 3.4 is clearly shown (see axis of plots) while maintaining the shape of the number state. This demonstrates that the shifted number states are suitable to follow the trajectory.

The operation of the usual creation and annihilation operators a and $a^\dagger$ on the displaced number state $|\alpha, n\rangle$ is quite different from their usual operation on a number state $|n\rangle$. Consider the shifted operators $a(\alpha)$ and $a^\dagger(\alpha)$ defined as

$$a = a(\alpha) + \alpha \tag{3.43}$$
$$a^\dagger = a^\dagger(\alpha) + \alpha^*, \tag{3.44}$$

where the displaced operators operate on displaced number states as

$$a(\alpha)\,|\alpha, n\rangle = \sqrt{n}\,|\alpha, n-1\rangle \tag{3.45}$$
$$a^\dagger(\alpha)\,|\alpha, n\rangle = \sqrt{n+1}\,|\alpha, n+1\rangle. \tag{3.46}$$

This gives the operation of the normal operators on the displaced number states as

$$a\,|\alpha, n\rangle = \sqrt{n}\,|\alpha, n-1\rangle + \alpha\,|\alpha, n\rangle \tag{3.47}$$
$$a^\dagger\,|\alpha, n\rangle = \sqrt{n+1}\,|\alpha, n+1\rangle + \alpha^*\,|\alpha, n\rangle. \tag{3.48}$$

Using the displaced number states and displaced operators, it is possible to have a basis that follows the trajectory of each mode. Using the displaced number states the state vector $|\psi\rangle$ is represented as

$$|\psi\rangle = \sum_{i=0}^{N_1}\sum_{j=0}^{N_2} \tilde{c}_{i,j}|\alpha_1, i\rangle|\alpha_2, j\rangle. \tag{3.49}$$

The α_1 and α_2 can be shifted to follow the evolution of the state vector in mode 1 and mode 2 respectively, by moving the basis 1 to the point given by $\alpha_1 = \langle a_1\rangle$ when needed, and basis 2 to the point $\alpha_2 = \langle a_2\rangle$ when needed.

Since the operation of the normal operates a and $a^\dagger$ on the displaced number states is known Eq. 3.14 does not need to be modified to deal with the new basis.

3.3.4 Changing Basis

To allow the basis to follow the trajectory, it is necessary to be able to transform the current basis into the future basis. In Sec. 3.3.3 a suitable basis was developed, along with suitable operators, which are capable of following the trajectory. This chapter develops the theory allowing a transition from one basis to another to occur.

From Eq. 3.49, where now $\alpha_1 = \alpha$ and $\alpha_2 = \beta$, the basis of representation for each mode can be changed individually. The transformation for the change of basis of mode 1 $|\alpha, i\rangle$ of the state vector is developed with the result for the change of basis in the second mode $|\beta, j\rangle$ given, but not derived. Consider

$$basis\ 1 \rightarrow basis\ 2. \tag{3.50}$$

Now the state of the system $|\psi\rangle$ in each basis is

$$|\psi\rangle = \sum_{i,j} c_{i,j}|b_i\rangle|b_j\rangle \tag{3.51}$$
$$= \sum_{k,j} \tilde{c}_{k,j}|\tilde{b}_k\rangle|b_j\rangle. \tag{3.52}$$

The relationship between the coefficients $c_{i,j}$ and $\tilde{c}_{k,j}$ is then given by

$$\tilde{c}_{k,j} = \sum_i c_{i,j} \langle \tilde{b}_k | b_i \rangle. \tag{3.53}$$

It is now a matter of computing the overlap $\langle \tilde{b}_k | b_i \rangle$, or when the basis states are displaced number states $\langle \alpha, k | \tilde{\alpha}, i \rangle$. Using equation 3.37

$$\langle \alpha, k | \tilde{\alpha}, i \rangle = \langle k | D^\dagger(\alpha) D(\tilde{\alpha}) | i \rangle. \tag{3.54}$$

Substituting Eq. 3.41

$$\langle \alpha, k | \tilde{\alpha}, i \rangle = e^{\frac{1}{2}(\alpha^* \tilde{\alpha} - \alpha \tilde{\alpha}^*)} \langle k | D(\tilde{\alpha} - \alpha) | i \rangle. \tag{3.55}$$

From the definition of $D(\alpha)$, Eq. 3.39

$$\langle \alpha, k | \tilde{\alpha}, i \rangle = e^{\frac{1}{2}(\alpha^* \tilde{\alpha} - \alpha \tilde{\alpha}^*)} \langle k | e^{(\tilde{\alpha}-\alpha)a^\dagger - (\tilde{\alpha}-\alpha)^* a} | i \rangle. \tag{3.56}$$

Using the relationship for operators A and B

$$e^{A+B} = e^A e^B e^{-\frac{1}{2}[A,B]}, \tag{3.57}$$

Eq. 3.56 becomes

$$\langle \alpha, k | \tilde{\alpha}, i \rangle = e^{\frac{1}{2}(\alpha^* \tilde{\alpha} - \alpha \tilde{\alpha}^*)} \langle k | e^{(\tilde{\alpha}-\alpha)a^\dagger} \, e^{-(\tilde{\alpha}-\alpha)^* a} \, e^{\frac{1}{2}(\tilde{\alpha}-\alpha)(\tilde{\alpha}-\alpha)^* [a^\dagger, a]} | i \rangle. \tag{3.58}$$

Now from the relationship $[a, a^\dagger] = 1$ this becomes

$$\langle \alpha, k | \tilde{\alpha}, i \rangle = e^{\frac{1}{2}(\alpha^* \tilde{\alpha} - \alpha \tilde{\alpha}^*) - \frac{1}{2}(\tilde{\alpha}-\alpha)(\tilde{\alpha}-\alpha)^*} \langle k | \, e^{(\tilde{\alpha}-\alpha)a^\dagger} \, e^{-(\tilde{\alpha}-\alpha)^* a} | i \rangle. \tag{3.59}$$

In calculating the product $\langle k | \, e^{(\tilde{\alpha}-\alpha)a^\dagger} \, e^{-(\tilde{\alpha}-\alpha)^* a} | i \rangle$ it is necessary to use

$$a^m | i \rangle = \sqrt{\frac{i!}{(i-m)!}} | i - m \rangle. \tag{3.60}$$

Now consider the right component of Eq. 3.59

$$\begin{aligned} e^{-(\tilde{\alpha}-\alpha)^* a} | i \rangle &= \sum_m \frac{\{-(\tilde{\alpha}-\alpha)^* a\}^m}{m!} | i \rangle \\ &= \sum_{m=0}^{i} \frac{\{-(\tilde{\alpha}-\alpha)^*\}^m}{m!} \sqrt{\frac{i!}{(i-m)!}} | i - m \rangle \end{aligned} \tag{3.61}$$

and similarly the left component of Eq. 3.59 is

$$\langle k | e^{(\tilde{\alpha}-\alpha)a^\dagger} = \sum_{n=0}^{k} \langle k - n | \frac{(\tilde{\alpha}-\alpha)^n}{n!} \sqrt{\frac{k!}{(k-n)!}} \tag{3.62}$$

Now substituting Eq. 3.61 and Eq. 3.62 into Eq. 3.59 the following is obtained

$$\begin{aligned} \langle \alpha, k | \tilde{\alpha}, i \rangle &= C \sum_{n=0}^{k} \langle k - n | \frac{(\tilde{\alpha}-\alpha)^n}{n!} \sqrt{\frac{k!}{(k-n)!}} \sum_{m=0}^{i} \frac{\{-(\tilde{\alpha}-\alpha)^*\}^m}{m!} \sqrt{\frac{i!}{(i-m)!}} | i - m \rangle \\ &= C \sum_{n=0}^{k} \sum_{m=0}^{i} \frac{(\tilde{\alpha}-\alpha)^n}{n!} \sqrt{\frac{k!}{(k-n)!}} \frac{\{-(\tilde{\alpha}-\alpha)^*\}^m}{m!} \sqrt{\frac{i!}{(i-m)!}} \langle k - n | i - m \rangle, \end{aligned} \tag{3.63}$$

where C is a normalisation constant, and given by

$$C = e^{\frac{1}{2}(\alpha^* \tilde{\alpha} - \alpha\tilde{\alpha}^*) - \frac{1}{2}(\tilde{\alpha}-\alpha)(\tilde{\alpha}-\alpha)^*}. \tag{3.64}$$

Now we have the relationship $\langle k-n|i-m\rangle = \delta^{k-n,\,i-m}$ which gives the only non-zero components when $l = k-n = i-m$. Substituting this into Eq. 3.63 the overlap between the two displaced number states $\langle \alpha, k|\tilde{\alpha}, i\rangle$ is given by

$$\langle \alpha, k|\tilde{\alpha}, i\rangle = C \sum_{l=0}^{min(i,k)} \frac{\{-(\tilde{\alpha}-\alpha)^*\}^{i-l}}{(i-l)!} \sqrt{\frac{i!}{l!}} \frac{(\tilde{\alpha}-\alpha)^{k-l}}{(k-l)!} \sqrt{\frac{k!}{l!}}. \tag{3.65}$$

This can be numerically calculated, generating a matrix of values

$$\Delta_{i,k}(\alpha, \tilde{\alpha}) = C \sum_{l=0}^{min(i,k)} \frac{\{-(\tilde{\alpha}-\alpha)^*\}^{i-l}}{(i-l)!} \sqrt{\frac{i!}{l!}} \frac{(\tilde{\alpha}-\alpha)^{k-l}}{(k-l)!} \sqrt{\frac{k!}{l!}}. \tag{3.66}$$

This gives the transformation of coefficients c into $\tilde{c}$, where c is the matrix of $c_{i,j}$ values and $\tilde{c}$ is the matrix of values $\tilde{c}_{k,j}$, as the matrix equation (for changing the basis of the first mode $|\alpha, i\rangle$ of the state vector)

$$\tilde{c} = \Delta(\alpha, \tilde{\alpha}) \times c, \tag{3.67}$$

where $\Delta(\alpha, \tilde{\alpha})$ is the matrix of values $\Delta_{i,k}(\alpha, \tilde{\alpha})$. The matrix equation for changing the basis of the second mode $|\beta, j\rangle$ of the state vector

$$|\psi\rangle = \sum_{i,j} c_{i,j}|b_i\rangle|b_j\rangle \tag{3.68}$$

$$= \sum_{i,k} \tilde{c}_{i,k}|b_i\rangle|\tilde{b}_k\rangle \tag{3.69}$$

is easily shown to given by

$$\tilde{c} = \Delta(\beta, \tilde{\beta}) \times c^T, \tag{3.70}$$

where now c is the matrix of $c_{i,j}$ values, $\tilde{c}$ is the matrix of values $\tilde{c}_{i,k}$ and the c^T is the transpose of c

It must be noted that Eq. 3.67 gives the matrix equation obtaining the co-efficients for a change in basis in the first mode only, while Eq. 3.70 gives the co-efficients for a change in basis in the second mode only. To obtain a general change of basis (a change in both mode 1 and mode 2) it is necessary to change each basis separately.

The two matrix equations Eq. 3.67 and Eq. 3.70 for the coefficients of the state vector Eq. 3.49 allow the basis to follow the evolution of the state vector along its trajectory. With the decrease in basis size that this should bring, the time to compute the integration steps is greatly decreased offsetting the extra computational time required to change the basis.

3.4 Numerical Procedure for solving the Master Equation

The numerical procedure for solving the master equation for second harmonic generation is to solve the quantum state diffusion equation Eq. 3.14 for an ensemble of trajectories. By averaging the reduced density operators Eq. 3.3 the full density operator can be obtained. The solution obtained is the steady state solution of the master equation Eq. 2.4.

To reduce the size of the ensemble needed, advantage can be taken of the random component of the quantum state diffusion equation. After some time, the evolution of the state vector will no longer be correlated to its previous evolution. This means that integrating

for long periods of time is equivalent to integrating many trajectories. This can be used to reduce computation time as the transient behavior does not need to be removed (already on the attractor). The time for this decorrelation to occur is taken to be the inverse damping time $1/\kappa$, where κ is given in Eq. 2.12 and is the ratio of the damping $\kappa = \kappa_1/\kappa_2$ of the two modes of the system. This damping period is a guess at the decorrelation time of the system, made in the absence of a numerical or analytical value.

The general procedure to integrate the quantum state diffusion equation with a moving basis is:

1. Take initial state $|\psi\rangle$ represented in a displaced number state basis, a time period to integrate over and an integration step size Δt
2. Integrate the state $|\psi\rangle$ by the amount Δt
3. Check to see whether the coefficients of the state vector on the boundary of the basis are larger than some specified ε, if they are then change the basis to an appropriate size and position
4. Check to see whether the system has been integrated for the specified period, if it has go to step 6
5. Go to step 2
6. Return the final state vector as the solution to the integration

The size of the basis that the state vector is represented on should be large enough to evolve the state vector a number of steps. A large enough basis should have the coefficients at the boundary of the basis smaller than some predetermined value ε, assuming a normalised state vector.

It must be remembered that the basis changes will be carried out on each mode individually. This means that if the coefficients of a mode grow too large then just the basis of that mode will be changed. That is, a basis change will occur for mode 1 if

$$\sum_{j=0}^{N_2} |c_{N_1,j}| > \varepsilon \tag{3.71}$$

and similarly for mode 2. The value of ε will be chosen according to the accuracy that the simulation will be performed to. When a change of basis occurs, the size of the boundary coefficients are required to be larger than some value $\varepsilon_{lower} < \varepsilon$ to stop the basis from growing too large. Setting this lower size for the boundary coefficients does not reduce the accuracy of the simulation, as the boundary coefficients are allowed to grow up to ε anyway.

3.5 Analysis of Numerical Simulation

Various methods were used to analyse the results of numerically integrating the quantum second harmonic generation quantum equations. The reduced density operator for each mode and Q-function for each mode were constructed for several regimes of operation. The trajectories in the α plane were produced, along with an analysis of how sensitive the system is to perturbations.

The evolution of the trajectory in the α plane were demonstrated by Zheng and Savage [Zheng 1995] and are extended to the classical/quantum interface to a greater extent in Sec. 4.2.2. The Q-function plots have not been investigated to great depth previous to this thesis,

though their expected form is well understood. The sensitivity to perturbations has not been applied to an open quantum system, such as second harmonic generation, with the outcome of such an analysis only speculated about.

3.5.1 Trajectories in the α plane

To compute the trajectory of the system in the α plane the expectation values $\langle a_1 \rangle$ and $\langle a_2 \rangle$ were computed and stored. The computation of these expectation values was done using Eq. 3.43 to Eq. 3.46 on the state vector $|\psi\rangle$ given by Eq. 3.49. To accompany these plots the size of the basis for each mode was also produced.

The number of photons in the system was not computed, though the code could easily be added. An approximate maximum number of photons can be calculated by computing $|\alpha_{max}|^2$. The main interest was how the system changed as S was decreased, making the system more classical, rather than the exact number of photons being simulated in the system.

3.5.2 Q-functions of reduced density operators

To construct the full density operator solution from the quantum state diffusion solutions Sec. 3.2.3 requires that all the state vectors, being averaged over, be in the same basis. The basis chosen must be large enough to cover the entire trajectory of the system through the Hilbert space. Using the results of Zheng and Savage this is at least 500 basis states in each mode of the system. This equates to 500^4 complex numbers being stored (approximately 931 gigabytes of information) to make up the density operator. This is beyond the means of the computer available. To reduce this to a manageable amount reduced density operators of each mode were constructed (requiring 500^2 complex numbers, or approximately 4 megabytes of information, to be stored). The average reduced density operator $\bar{\rho}_{mode1}$ of mode 1 is given by (where all the bases of the state vectors have been moved to $\alpha = 0$)

$$\begin{aligned}
\bar{\rho}_{mode1} = M\left[\rho_{mode1}\right] &= M\left[Tr_{mode2}(|\psi\rangle\langle\psi|)\right] && (3.72)\\
&= M\left[\sum_i \langle i,\beta| \sum_{j,k,l,m} c_{j,k}c^*_{l,m}\,|\beta,k\rangle|0,j\rangle\langle l,0|\langle m,\beta|\beta,i\rangle\right] && (3.73)\\
&= M\left[\sum_{i,j,l} c_{j,i}c^*_{l,i}\,|0,j\rangle\langle l,0|\right] && (3.74)\\
&= \sum_{j,l} \tilde{C}_{j,l}|0,j\rangle\langle l,0| && (3.75)
\end{aligned}$$

and similarly for the reduced density operator of mode 2. $\tilde{C}_{j,l}$ are the co-efficients of the average reduced density operator. From the average reduced density operator $\bar{\rho}_{mode1}$ the Q-function for mode 1 is calculated using

$$\begin{aligned}
Q(\alpha) &= \langle\alpha|\bar{\rho}_{mode1}|\alpha\rangle && (3.76)\\
&= \langle\alpha|\sum_{j,l} \tilde{C}_{j,l}|0,j\rangle\langle l,0|\alpha\rangle && (3.77)\\
&= \sum_{j,l} \tilde{C}_{j,l}\langle\alpha|0,j\rangle\langle l,0|\alpha\rangle. && (3.78)
\end{aligned}$$

Eq. 3.65 is used to calculated the overlaps $\langle\alpha|0,j\rangle$ and $\langle l,0|\alpha\rangle$ giving

$$Q(\alpha) = e^{\frac{1}{2}|\alpha|^2} \sum_{j,l} \tilde{C}_{j,l} \frac{(\alpha^*)^j}{j!} \frac{\alpha^l}{l!}. \tag{3.79}$$

This can be computed for some discrete grid values of α generating a Q-function plot. A similar equation is obtained for mode 2 of the system.

3.5.3 Hyper-sensitivity of Second Harmonic Generation to perturbations

As discussed in Sec. 2.3.2 a measurement of the environment, with r possible outcomes, leads to r possible states of the system each labeled ρ_r. These ρ_r are effectively non-overlapping groupings of the ensemble of state vectors obtained via quantum state diffusion simulation. To simulate this on a computer it is sufficient to obtain a large number of state vectors from the simulation and form non-overlapping groups of them. By changing the size of the groups, computing the average change in information ΔI and and average change in entropy ΔH the approximate function relationship between ΔH_{Tol} and ΔI_{min} is demonstrated.

To determine whether second harmonic generation is hyper-sensitive to perturbations it is necessary to compute Eq. 2.16 and Eq. 2.17 for various groupings of the state vectors. In order to calculate the different groupings, it was suggested by Schack *et al* [Schack 1996 a] that the distance between two normalised state vectors be found using the Hilbert-space angle

$$s\left(|\psi_1\rangle, |\psi_2\rangle\right) = \cos^{-1}\left(|\langle\psi_1|\psi_2\rangle|\right). \tag{3.80}$$

To form a group the following procedure is followed:

1. Take a state vector $|\psi\rangle$ not already in a group
2. Find all state vectors, not in a group, within a distance s of $|\psi\rangle$, this is a group
3. Any state vectors left? If no goto 5
4. goto 1
5. Perform calculations on groups

Unfortunately to compute the overlap $\langle\psi_1|\psi_2\rangle$ for each pair of state vectors would require more computer time than is available. The reason is that both state vectors need to be in the same basis, requiring that all state vectors effectively need to be in the same basis. This necessitates a large basis, increasing computational time drastically. To overcome this, a different distance was used: the distance between bases. Since each state vector is represented using displaced number states $|\alpha, n\rangle|\beta, m\rangle$ (two mode system), the distance between two state vectors $|\psi_1\rangle$ and $|\psi_2\rangle$ was taken to be

$$s\left(|\psi_1\rangle, |\psi_2\rangle\right) = \sqrt{(\alpha_1 - \alpha_2)^2 + (\beta_1 - \beta_2)^2}. \tag{3.81}$$

Since the basis follows the state vectors along the trajectory, this allows a reasonable approximate distance between state vectors to be computed. One major drawback occurs when two state vectors have the same basis but are not equal. The method used does not differentiate between the two vectors. Consequently, groups are possible larger than they should be. Another problem occurs when ΔH_{Tol} is very small. At some level, no matter how much information is known about the perturbing environment, the state vectors cannot be subdivided into any more groups. In some sense the groups obtained are more coarse than desired, but due to the time constraints better results could not be achieved.

With the groups calculated the process of determining $Tr_S(\rho \log_2 \rho)$ in Eq. 2.16 is non trivial. The computation of density operators, as discussed previously, is not viable on

the available computers (not enough memory). To overcome this, consider the non-zero eigenvalues λ_i of ρ. Then $Tr_S(\rho \log_2 \rho) = \sum_i \lambda_i \log_2 \lambda_i$, where the problem is now one of finding eigen values. To compute the eigenvalues of ρ for a group with N state vectors, it is useful to express the density operator as $\rho = (1/N)\ A\ A^\dagger$, where A has the N state vectors as its columns, $A = (|\psi_1\rangle, |\psi_2\rangle, \ldots, |\psi_N\rangle)$. Now the *Singular Value Decomposition* of A can be found, which expresses A as $A = U_1 W_D U_2$, for unitary matrices U_1, U_2 and diagonal matrix W_D. The non-zero eigenvalues of ρ are $\lambda_i = (1/N)\ w_i^2$, where w_i is the i^{th} diagonal element of W_D, allowing the entropy of ρ to be found.

The value of p_r in Eq. 2.16 and Eq. 2.17 is easily computed as the number of state vectors in a group divided by the total number of state vectors used for the computation. Now ΔI and ΔH can be computed and plotted, allowing the approximate functional dependence of ΔH_{Tol} and $[DeltaI_{min}$ to be determined.

Chapter 4

Results of Numerical Simulations

This chapter outline the development of the computer code, the tests performed to ensure its accuracy and the results produced. Discussion and interpretation of the numerical results are included along with their relevance to the aims developed in previous chapters. Detail about the actual programming of the computer code is not included, just general directions taken in producing the code. To see how the actual model was implemented see App. 1 at the end of the Thesis.

4.1 Verification of code

The code was developed from the ground up, using no previously developed code. A package written in c++ was supplied by Schack *et al* [Schack 1996 b], but not used as it was not sufficiently fast enough on the Fujitsu VPP300 super computer available. Consequently all code was written in optimised Fortran90 code.

The code was developed in several sections, with test carried out along the way to ensure numerical stability, accuracy and to gain a feel for the time taken to perform computations. The first component developed was the complex Wiener process, which requires Gaussian random numbers to be produced. A basic integration scheme was developed to integrate a well known stochastic differential equation, the *Ornstein-Uhlenbeck* stochastic differential equation, to test the Weiner process. From this, the integration process was modified to integrate a single mode state vector equation, and tested for the simple harmonic oscillator. The moving basis was then added, followed by the implementation of integrating a two mode system. This was tested for two uncoupled harmonic oscillators, with the final step being to integrate two coupled harmonic oscillators in a configuration to model *Second Harmonic Generation.*

4.1.1 Complex Wiener Process

The complex Wiener process involves producing a complex random number $\mathcal{R} = \frac{a+i\ b}{\sqrt{2}}$, where a and b are real Gaussian random numbers with variance 1 and average 0 and ΔW is defined as $\Delta W = \mathcal{R}\sqrt{\Delta T}$. For the quantum state diffusion process the Wiener process ΔW is normalised to Δt with its real and imaginary parts being normalised to $\Delta t/2$. The method for obtaining real Gaussian random number was taken from "Numerical Recipes" [Press 1995]. Using the built in random number generator on the VPP300, both the Wiener process and Gaussian random numbers were tested to make sure they had the right variance and average.

It was found that using the built in random number generator, the average for both the Wiener process and the Gaussian random numbers correctly asymptotes to the expected

value of 0. The variance, however, could not be reduced below 1 ± 10^{-6} no matter how many random numbers were averaged over.

The solution was to implement a known *good* random number generator. The one used was given in Numerical Recipes [Press 1995]. Using this new routine, the average was seen to asymptote to 0 and the variance of the Wiener process and the Gaussian random numbers a and b tended toward 1. This "better" random number generator was used in all successive calculations.

4.1.2 Ornstein-Uhlenbeck Stochastic Differential Equation

The Ornstein-Uhlenbeck stochastic differential equation [Gardiner 1985] (pp. 106)

$$dx = -kx\ dt + \sqrt{D}\ dW(t) \tag{4.1}$$

can be solved analytically with the substitution

$$y = xe^{kt} \tag{4.2}$$

and remembering to keep second order terms, such that

$$dy = (dx)(de^{kt}) + (dx)e^{kt} + xd(e^{kt}). \tag{4.3}$$

By assuming the initial condition in deterministic or Gaussian distributed and independent of $dW(t)$ for all $t > 0$, the mean and variance of $x(t)$ are

$$\langle x(t) \rangle = \langle x(0) \rangle e^{-kt} \tag{4.4}$$

$$var\{x(t)\} = \left(var\{x(0)\} - \frac{D}{2k} \right) e^{-2kt} + \frac{D}{2k}. \tag{4.5}$$

The Ornstein-Uhlenbeck process Eq. 4.1 was modeled using an Euler-one step method Eq. 3.31. The average trajectory, as given by $\langle x(t) \rangle$, was plotted alongside the analytic solution and found to be in excellent agreement, with greater accuracy obtained with smaller step sizes. The numerical variance was also compared to the analytic variance, and the order of the integration method verified to be $\frac{1}{2}$. This process indicated that the numerical integration was indeed modeling the process correctly, and that the Wiener process was implemented correctly.

4.1.3 Integration of Harmonic Oscillator

The integration process developed to integrate the Ornstien-Uhlenbeck process was modified to integrate a state vector Eq. 1.1. This was a minor change due to the array notation of Fortran90. The differential components (deterministic and stochastic) needed to perform the integration were computed using the operator notation of Fortran90, which were later changed for optimisation. The creation and annihilation operators were constructed, allowing the integration of systems such as the simple harmonic oscillator $H = \hbar\omega(a^\dagger a + \frac{1}{2})$ to be easily performed, with the computer code reflecting directly the process being modeled (again this was changed for optimisation).

The simple harmonic oscillator was modeled (with no stochastic component) and the trajectory, obtained by computing $\langle a \rangle$, compared to the analytic solution. The numerical results were found to be in excellent agreement with the analytic solution, with the order of the integration (without stochastic component) found to be 1. This verified that the operator notation used producing accurate results.

4.1.4 Implementing the Moving Basis

With the code working for a single mode state vector, the moving basis was implemented. This allowed it to be tested, with the results being compared to both the previous numerical and analytic results. The approximate cpu time used to perform various changes of basis were also noted for later approximation of run times. It was found very early in the testing that the change of basis was not operating correctly. It was producing large "kicks" to the trajectory which were found to be a result of rounding error within the computer.

The equation for changing basis Eq. 3.66 involves the alternating sum of similar terms. When this was modeled directly, numerical rounding errors produced the kicks seen. When this was changed and an analytic result used for subtracting neighboring terms, the kicks were removed.

The equation modeled in the computer code is best described by

$$\Delta_{i,k}(\alpha,\tilde{\alpha}) = C \sum_{l=0,\ l\ even}^{min(i,k)} \frac{\{-(\tilde{\alpha}-\alpha)^*\}^{i-l}}{(i-l)!}\sqrt{\frac{i!}{l!}}\,\frac{(\tilde{\alpha}-\alpha)^{k-l}}{(k-l)!}\sqrt{\frac{k!}{l!}}\left(1-\frac{(i-l)(k-l)}{|\tilde{\alpha}-\alpha|^2(l+1)}\right), \tag{4.6}$$

where now the sum over even l is no longer an alternating sum. This improved the accuracy of the calculation, allowing the basis to follow the trajectory accurately.

Another problem encountered with the change of basis algorithm was an overflow error when computing factors of the form, from Eq. 3.65,

$$\frac{d^{k-l}}{(k-l)!}\sqrt{\frac{k!}{l!}}, \tag{4.7}$$

for $d > 2$ and large basis sizes $k \approx 500$. To over come this the basis change was broken into many smaller steps of length $\delta\alpha \approx 1$ and the basis moved along from α to $\tilde{\alpha}$ in steps of $\delta\alpha$. This introduces compounding errors, but these were found to be well below the level of accuracy that the integration was being performed to.

4.1.5 Integration of a Two Mode System

The single mode system with a changing basis was modified to work with a two mode system. All the operators developed were also modified in the appropriate way, to act only on single modes. The change of basis was altered to allow each mode to operate in a different basis to the other mode, allowing it to follow the trajectory of its associated mode.

The system was then set up to model two decoupled simple harmonic oscillators, with no stochastic component. The trajectory followed by each mode was compared to the analytic solution for a simple harmonic oscillator and found to be in excellent agreement.

4.1.6 Integration of the Second Harmonic Generation equations

The parameters for the two mode system were adjusted to model second harmonic generation, with the stochastic term incorporated. Code was added to allow the state of the system, the average density operator and the position in the α plane to be saved to disk at various stages throughout the integration process. This allows the trajectory, Q-function and other diagnostic processes to be calculated, to gain an understanding of the systems dynamics.

The first check that the second harmonic code was working, was to run it in the regimes where results have already been obtained [Zheng 1995]. Zheng *et al* used quantum jumps to develop their trajectories, while this thesis uses quantum state diffusion. Consequently the trajectories obtained were not exactly the same, but good agreement between the two methods was obtained with the general structure being the same. Once the previous results were

reproduced the program was run for various regimes with the density operator (solution to the master equation) being reconstructed, along with the Q-function of the density operator. The last feature of the system that was explored was the hyper-sensitivity to perturbation.

4.2 Results and Interpretations of Numerical Simulation

With the code tested and producing correct results, it was run to study different features of the quantum optical process, second harmonic generation. The first results produced were used to compare the results from quantum state diffusion to those of quantum jumps [Zheng 1995]. The S value Eq. 2.13 was then decreased further, pushing the quantum process toward the quantum/classical interface. The reduced density operator for each mode was then produced, with its Q-function being calculated and compared to the classical cycle. The hyper-sensitivity to perturbations was then tested to determine whether the open quantum system demonstrates chaotic behavior.

In all the computations performed the parameter ε Eq. 3.71 was set to $\varepsilon = 10^{-3}$ with $\varepsilon_{lower} = 10^{-3}\ \varepsilon$.

4.2.1 Quantum State Diffusion compared to Quantum Jumps

The computer program was run for the scaled parameters $\tilde{E} = 31$, $\tilde{\kappa} = 0.25$ and $\tilde{\delta}_1 = \tilde{\delta}_2 = -1$ with the parameters S set to 5, 2.5 and 1.25 to compare the trajectories directly to those obtained by Zheng and Savage [Zheng 1995] and the classical solution (independent of S). The resulting trajectories are shown in Fig. 4.1

The difference between the two classical plots is due to the fact that the integration time Zheng and Savage used to remove any transients from the classical equations was unknown and guessed at.

As can be seen, the quantum state diffusion with a moving basis method produces results very close to those obtained by quantum jumps. As the S parameter is decreased both methods seem to be converging to the classical solution. This is agreement with the theoretical work by Brun *et al* [Brun 1996]. The approximate time required to compute the trajectory for $S = 1.25$ was 2 hours on the Fujitsu VPP300 super computer, in comparison to approximately 100 hours of super computer time used by Zheng and Savage. This demonstrates that Quantum State diffusion uses less computation time while providing the same results.

The size of the basis used in the quantum state diffusion varies along the trajectory, Fig. 4.2 shows how the basis varied along the trajectory for $S = 1.25$ and $\tilde{E} = 31$. They show that the maximum basis size required was less than 70 in each mode, 70^2 in total. The reduced basis size and consequently the reduced computation time demonstrates the usefulness of the quantum state diffusion model.

4.2.2 Toward the Quantum/Classical Interface

The computer program was run for the same parameter values as in Sec. 4.2.1 except that the S parameter was set to 0.75. The trajectories and basis sizes given in Fig. 4.3.

As S is decreased, the random fluctuations in the trajectory become negligible, producing a "smooth" path. The lobes present on the classical trajectory become well defined in the quantum system indicating that the classical solution would emerge for a small enough S.

The decrease in random fluctuations indicates that the random component of the stochastic differential equation Eq. 3.14, $(L_1 - \langle L_1 \rangle)\,|\psi(t)\rangle d\xi_1 + (L_2 - \langle L_2 \rangle)\,|\psi(t)\rangle d\xi_2$, is becoming negligible. This occurs when the state vector approaches a coherent state $|\alpha\rangle$ in each mode.

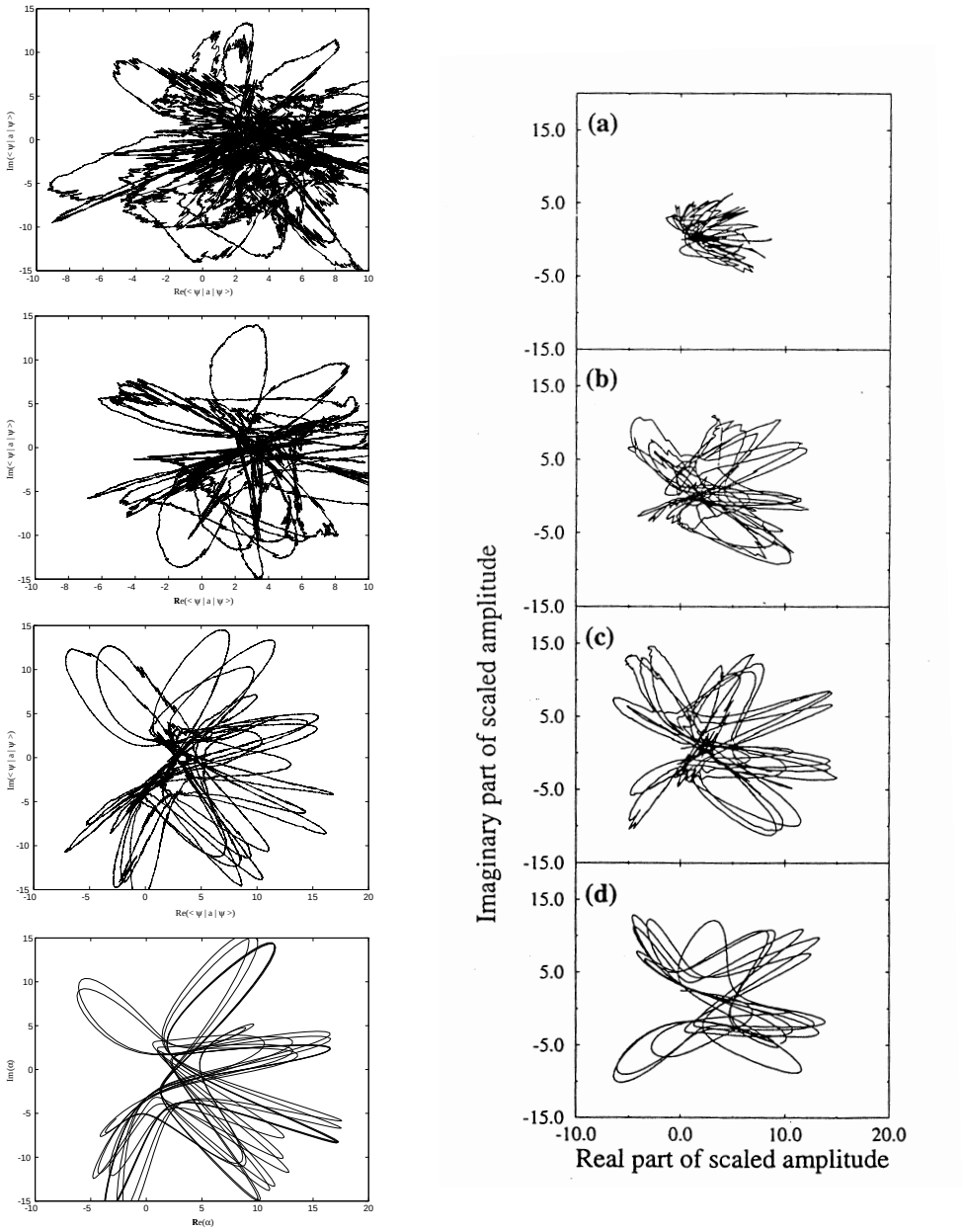

Figure 4.1: Comparison between the quantum jumps method (right) used by Zheng and Savage and the quantum state diffusion process (left). From top to bottom: $S = 5$; $S = 2.5$; $S = 1.25$; and the classical solution

This indicates that as the quantum trajectory is made classical the state vector factorises into a coherent state in each mode;

$$|\psi\rangle = \sum_{i,j}^{N_1,N_2} c_{i,j}|b_i\rangle|b_j\rangle \tag{4.8}$$

$$= \sum_{i,j}^{N_1,N_2} c_{1,i}|b_i\rangle c_{2,j}|b_j\rangle, \tag{4.9}$$

where $\sum_i c_{1,i}|b_i\rangle$ and $\sum_j c_{2,j}|b_j\rangle$ are coherent states.

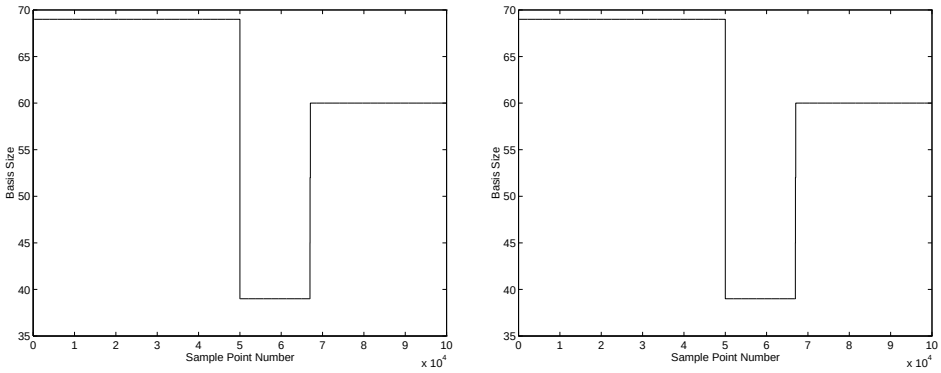

Figure 4.2: Basis size, for $\tilde{E} = 31$ and $S = 1.25$, at each sample point along the trajectory Fig. 4.1

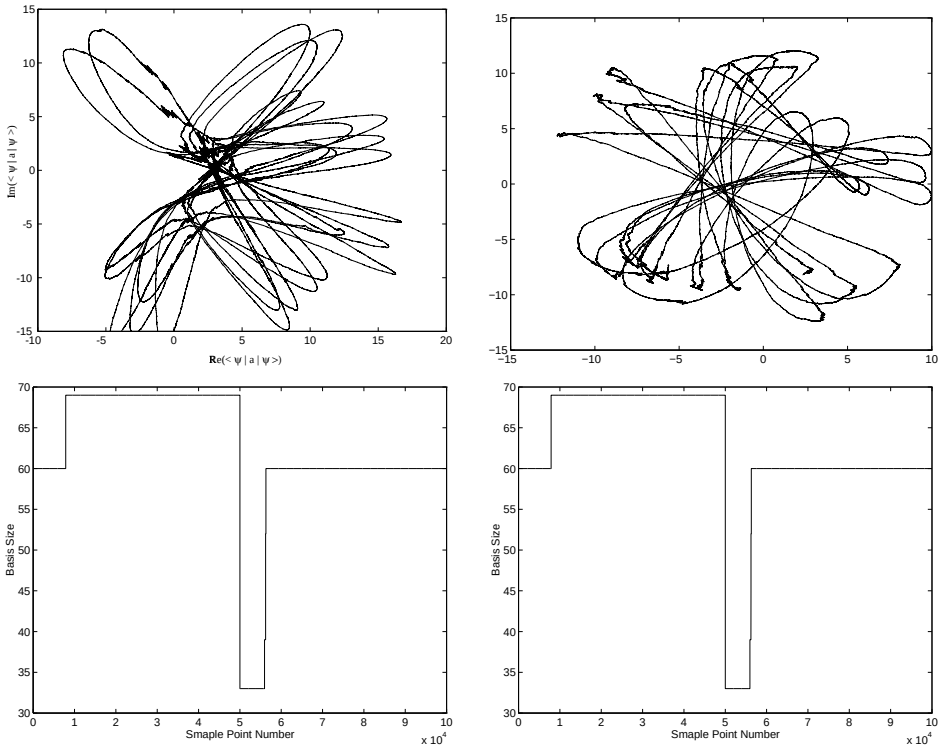

Figure 4.3: The top plots are the trajectory for $\tilde{E} = 31$ and $S = 0.75$ for mode 1 (left) and 2 (right). Below them are the basis sizes for the same trajectories.

4.2.3 Q-function plots of Trajectories

The Q-functions were plotted using the same parameter values as in 4.2.1 except that different driving amplitudes $\tilde{E}$ were used to investigate different regimes. The three regimes investigated using the quantum system were: $\tilde{E} = 1$, classical solution a fixed point; $\tilde{E} = 10$, classical solution a period two limit cycle; and $\tilde{E} = 31$, classical solution chaotic. The fixed point was used as a test case to ensure that the density operator and Q-function code was producing correct results. The period two limit cycle was used to demonstrate how the Q-function has support on the classical limit cycle and that the width decreases with decreasing S. The chaotic case was produced to investigate how the support of the classical trajectory changed when no limit cycle was present.

To generate the average reduced density operators and Q-functions for each mode, it was

necessary to generate a large ensemble of state vectors. To generate statistically *significant* data all of the state vectors need to be generated from individual integrations, ensuring that they are decorrelated. This was not the method used, as to do so would require unrealisable amounts of computer time. Instead the following scheme was used

1. Integrate the system for the decorrelation time (taken to be $1/\tilde{\kappa}$, see Sec. 3.4)
2. Integrate the system for one period of the classical system generating the average reduced density operators $M[\rho_{mode1}]$ and $M[\rho_{mode2}]$
3. Averaged over enough trajectories? If yes go to 5
4. Go to 1
5. Generate Q-function of average reduced density operators

In this way the approximate average density operator is constructed. The data obtained is not statistically significant, but allows a visual picture to be constructed to reasonable accuracy. The plots of Q-functions for various regimes are shown in Fig. 4.4 and Fig. 4.5

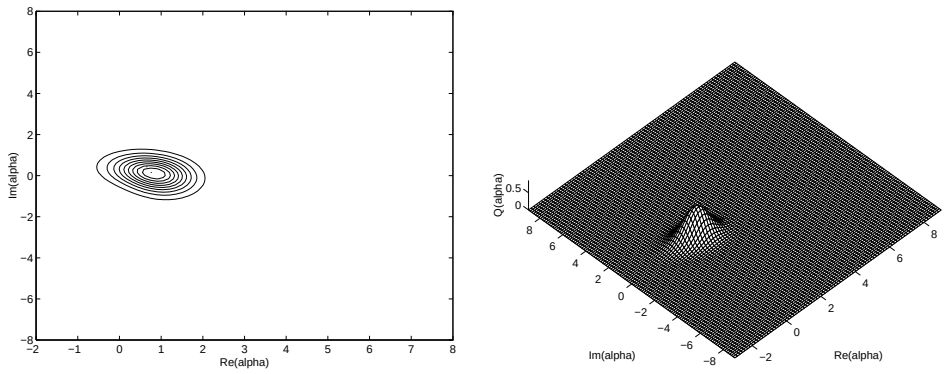

Figure 4.4: Q-function plot for $\tilde{E} = 1$ and $S = 1.25$. The plot was generated using 15 uncorrelated state vectors. The classical fixed point is at (0.645,0.4316).

The fixed point Q-function plot Fig. 4.4 demonstrates that the code is producing accurate results. The peak is well defined about the classical solution indicating that the numerical solution is converging to the correct classical result.

The plots in Fig. 4.5 demonstrate that the Q-function shows support on the classical attractor. It also shows that as the system is made more classical the width of the Q-function decreases, indicating that in the limit as $S \to 0$ the Q-function has support on the classical attractor. As pointed out in Sec. 4.2.2 the state vector seems to be tending to a coherent state $|\alpha\rangle$, in each mode, evolving around the classical path. This suggests that the Q-function's width, of the average reduces density operator, would be the same as the Q-function of the coherent state.

The Q-function plot was obtained for the case where the classical system demonstrates chaos $\tilde{E} = 31$ and a scaling parameter $S = 1.25$. The result are shown in Fig. 4.6. Unfortunately the value of $S = 1.25$ used is not very small, but the computation time, required to make the simulation more classical, was too great.

Fig. 4.6 shows that the Q-function seems to show the general structure of the classical trajectory. Due to the width of the state vector as it evolves along the trajectory and the fact that a chaotic system has very fine structure, the cratering seen for the limit cycles is not as prominent. This is expected, as a chaotic system has no steady state solution. This

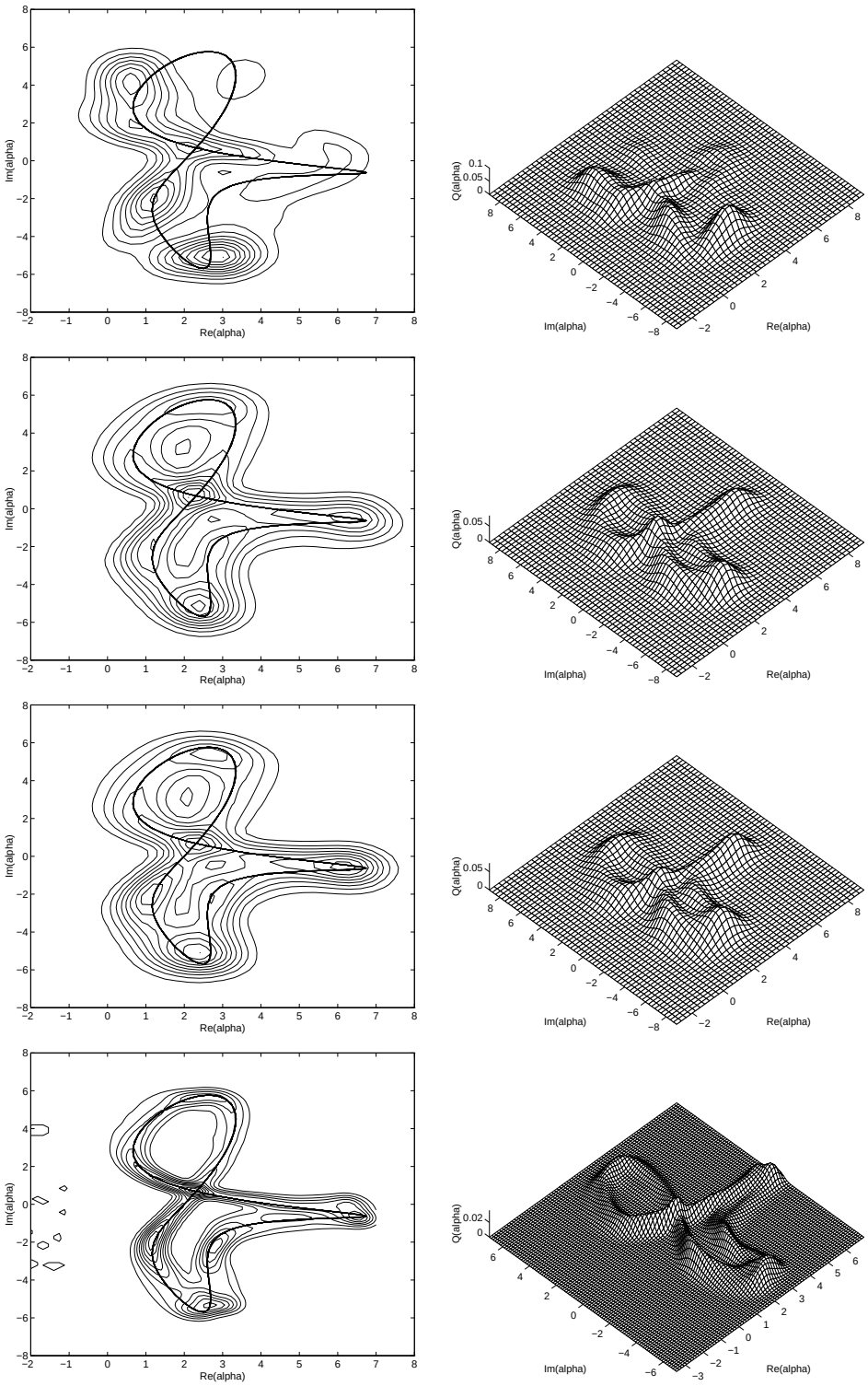

Figure 4.5: Q-function plots for (top to bottom) $\tilde{E}$ = 10, 10, 10, 10 and S = 1.25, 1.25, 0.75, 0.35. The plots are constructed using (top to bottom) : 15 uncorrelated state vectors; 5 lot of 100 correlated state vectors; 5 lots of 100 correlated state vectors; and 5 lots of 100 correlated state vectors. On the contour plots, the solid line is the classical attractor

means, that the path the system follows wanders about never visiting the same point twice. Since the Q-function is an average solution and, in the long time limit, produces a steady

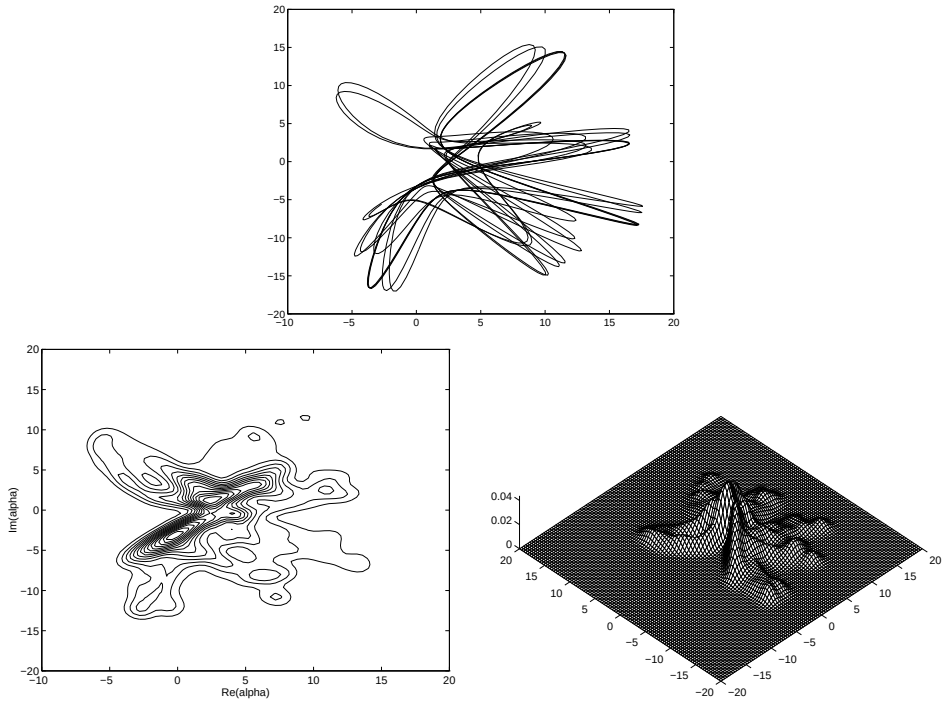

Figure 4.6: Q-function plot for $\tilde{E} = 31$ and $S = 1.25$ with 5 lots of 50 correlated state vectors. The top plot is the classical case, with the bottom two being the result of computing the Q-function for the quantum case

state solution, it will never be capable of showing the fine structure of the classical chaotic trajectory. The result should be a blurred out structure with the general shape of the chaotic path. This feature is seen, to some extent, in Fig. 4.6

4.2.4 Hyper-Sensitivity to Perturbations

The computer code was run to determine whether second harmonic generation was hyper-sensitivite to perturbations. The system was tested in three regimes, these being where the classical solution was: a fixed point; a period 2 cycle; and chaotic. For the simulation 100 uncorrelated state vectors were generated for each regime. The results of the numerical simulation are in Fig. 4.7 with Fig. 4.8 and Fig. 4.9 being constructed using the same data, just fewer state vectors.

From Fig. 4.7 it can be seen that the chaotic case requires a lot more information to be specified than the regular cases (the scale is logarithmic $\log_2$ in information and entropy). The effect of not being able to group the state vectors beyond their basis position is evident in the inability to resolve the fixed point. All state vectors end up in the same basis, consequently they are seen to be undifferentiated. This is clearly incorrect, as the chance of getting two state vectors, let alone 100, the same is negligible due to the random process of the environment.

The period 2 cycle also can not be resolved past $\Delta H_{Tol} \approx 1.5$ as beyond this point all the state vectors are represented using the same basis. The plots do suggest, however, that second harmonic generation, when run in the regime where the classical equations demonstrate chaos, is hyper-sensitive to perturbations. The general features observed by Schack *et al* are seen in Fig. 4.7: the fast drop off of the regular case; and the slower drop off of the chaotic case.

The nature of the model being used could be one reason that the sharp decrease in the regular case is not as prominent as that seen by Schack *et al.* The stochastic process

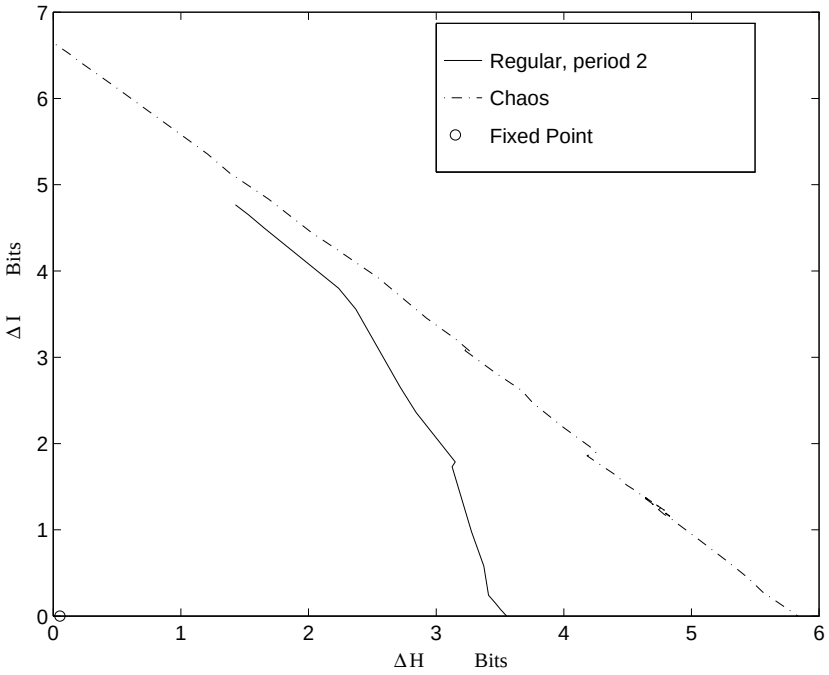

Figure 4.7: Plot of minimum information ΔI_{min} required to decrease the system entropy ΔH_S to some tolerable level ΔH_{Tol} for a fixed point, period 2 cycle and chaos. 100 state vectors were used for each regime.

introduces a continuous random fluctuation throughout the evolution, rather than just at the start of the evolution. This means that the possible outcomes of the system are far greater, requiring the perturbation to be defined to greater detail in order to reduce the change in entropy of the system to some tolerable level.

In the work by Schack *et al* they suggested that even with 4000 state vectors they were seeing effects due to the small sample size. This means that with only 100 state vectors used in the analysis the results are not as conclusive as desired, the fact that the chaotic case is significantly higher is evidence that the system is hyper-sensitive to perturbations. With more state vectors and greater resolving ability in the grouping procedure more accurate results would be obtained.

As the number of state vectors is halved, the information required to specify them decreased by half (groups are much smaller), or by 1 bit on the logarithmic plot. To take this scaling into account, it is necessary to increasing the information of the "25 vector run" by 2 and the "50 vector run" by 1, the result is shown in Fig. 4.10. As can be seen the three regular plots approximately lie on the same curve and the three chaotic plots approximately lie on the same line. The dramatic decrease in the regular case is now more clear. With more state vectors used in the calculation, the plot would be continued down and the difference between the regular case and chaotic case would become greater. This is reason to believe that if more state vectors were obtain, the hyper-sensitivity to perturbation would be more evident.

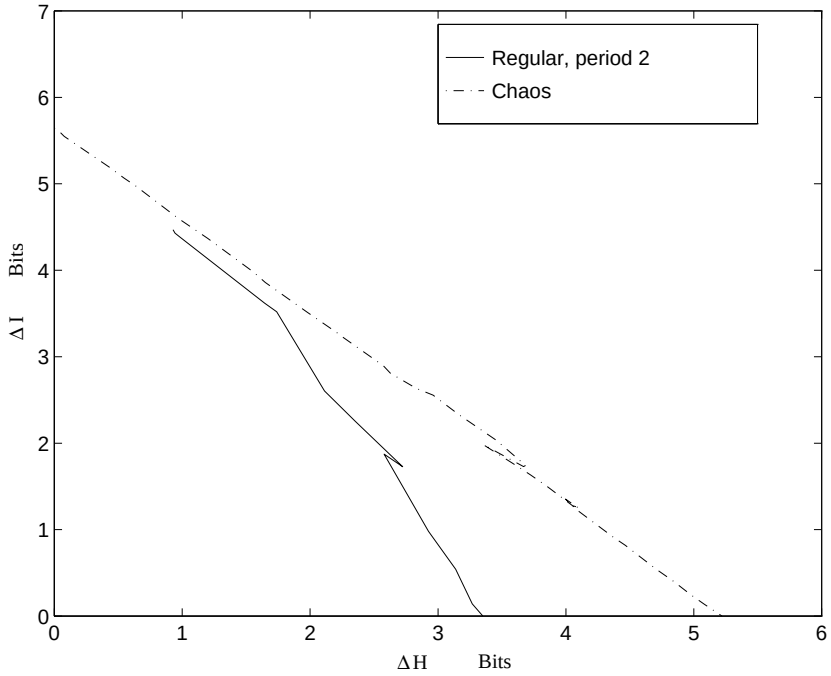

Figure 4.8: Plot of minimum information ΔI_{min} required to decrease the system entropy ΔH_S to some tolerable level ΔH_{Tol} for a fixed point, period 2 cycle and chaos. 50 state vectors were used for each regime.

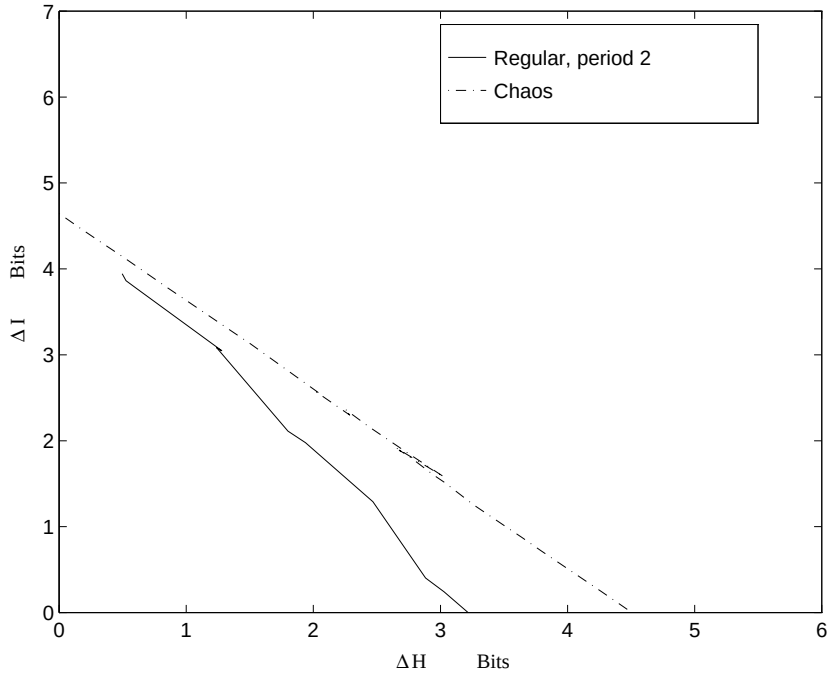

Figure 4.9: Plot of minimum information ΔI_{min} required to decrease the system entropy ΔH_S to some tolerable level ΔH_{Tol} for a fixed point, period 2 cycle and chaos. 25 state vectors were used for each regime.

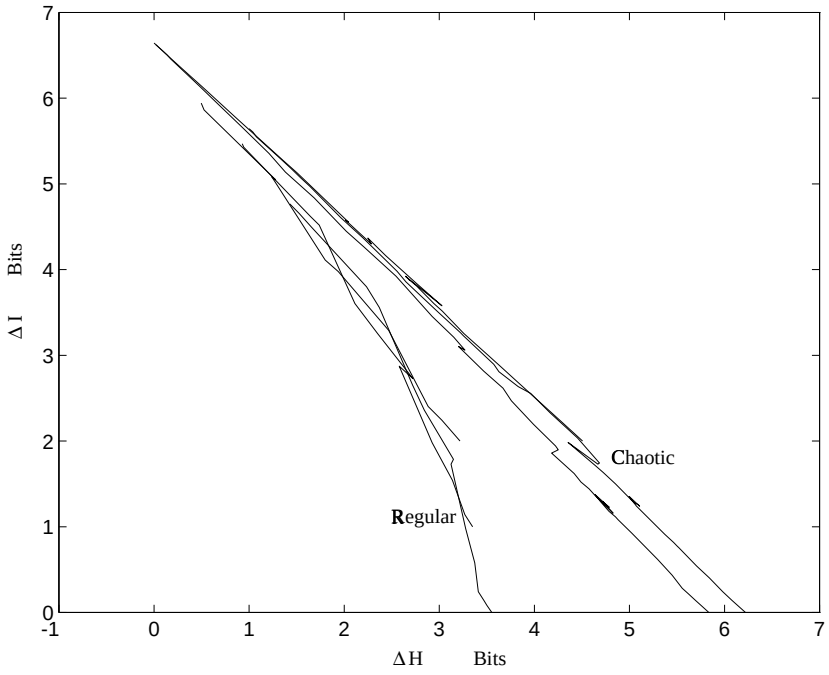

Figure 4.10: Fig. 4.7 to Fig. 4.9 plotted with the reduced information due to reduced state vectors scaled out.

Chapter 5

Conclusion

The objective of this thesis was to determine the existence of chaos in the quantum system second harmonic generation and attempt to characterise it. A major component of the research was writing a computer program which successfully modeled the quantum system using the method known as "quantum state diffusion with a moving basis". Using a recently developed definition of chaos, applicable to quantum systems, the nature of the open quantum system was studied.

The computer code developed allowed the realisation of trajectories to be performed on time scales orders of magnitude less than previous work. The time to produce a trajectory ranged between 5 minutes and 60 hours, depending on the regime, analysis being performed and the quality of results required. With two types of code being supported, one optimised and one easily modified for other systems, the opportunity for further research is enhanced.

It was demonstrated that the individual trajectories of the quantum system in the α plane, approach those of the classical system. As the scaling parameter, S, was decreased, the random fluctuations in the quantum trajectory become negligible indicating that each mode of the system is tending toward a coherent state (a coherent state does not see the random fluctuations $d\xi$ in Eq. 3.14). This is in agreement with the hypothesis of Zheng and Savage [Zheng 1995].

The Q-functions were shown to have support on the classical limit cycles and possess the general form of phase space traversed by the classical equations operating in the chaotic regime. As the scaling parameter was decreased the Q-function became more defined (decreased width). The classical equations describe the evolution of the amplitude of a classical like wave, while the quantum description deals with the evolution of a state vector. It was suspected that as the quantum system is made more classical, the state vector approaches a coherent state traveling around the classical limit cycle. This would limit the minimum width of the Q-function to the width of the coherent state. More investigation into the nature of the Q-functions is needed to verify or disprove this hypothesis.

The decreasing width of the Q-function is only possible if the individual quantum trajectories traverse similar paths in Hilbert space. This agrees with the idea that individual quantum trajectories approach the classical trajectories as the scaling parameter S is decreased. The decreasing spread of the Q-function also indicates that, in some sense, the path of the individual realisations of the quantum system are following those of the classical system. It was not expected that they exactly follow the path of the classical system. As the name "Chaos" suggests, the system is hyper-sensitive to perturbations and initial conditions and as such, the quantum system should not be exactly the same as the classical.

One aspect of chaos, dealing with a quantum system's hyper-sensitivity to perturbations, was researched. While the results were not conclusive, they tend to show that indeed the open quantum approach to second harmonic generation leads to hyper-sensitivity. When the infor-

mation known about the perturbing environment was plotted against the change in entropy of the system, the regular case did not exhibit the sharp decrease expected [Schack 1996 a], but a rather gradual roll off. The small sample of state vectors and limited resolvablity of state vectors may reduce the accuracy, it was felt that the model used was the largest contributing factor for the discrepancy. Quantum state diffusion introduces a random fluctuation into every integration step of the trajectory. The models used by Schack *et al* only introduced a fluctuation at specific periodic times, rather than the approximate continuous spectrum of quantum state diffusion. The continual random process means that the system has greater possible outcomes than the case where a single, or regular, perturbation occurs. The result is an increase in information required to reduce the average change in system entropy to the specified tolerable level.

The possibilities for further research in the field of quantum chaos are enormous. Directly relating to this thesis and second harmonic generation, there are many possible research areas. One area not investigated was whether the environment factorises the state vector into products of the two modes

$$|\psi\rangle = \sum_{i,j}^{N_1,N_2} c_{i,j}|b_i\rangle|b_j\rangle \tag{5.1}$$

$$= \sum_{i,j}^{N_1,N_2} c_{1,i}|b_i\rangle c_{2,j}|b_j\rangle. \tag{5.2}$$

This can be performed by observing the higher order expectation values, eg. $\langle a_1^2\rangle$ and $Re(\langle a_2 a_1^\dagger a_1\rangle + \kappa\langle a_2^\dagger a_2^2\rangle)$ [Zheng 1995]. The effect that this factorising has on the density operator solution to the master equation would also be another area to research.

The evolution of the Q-functions, as the scaling parameter is reduced, would also be a good area to continue research, determining if the width tends to that of a coherent state and other properties of the Q-function and density operators. Generating Q-functions for statistically significant vectors (ensuring they are decorrelated) would be another avenue to pursue.

Along the lines of correlated and uncorrelated state vectors, an approximate decorrelation length was used. Producing a definition of correlation length and then calculating it would be beneficial, allowing the assumption, that a single long trajectory is equivalent to many trajectories, to be verified.

With a definition of chaos, "hyper-sensitivity to perturbations", applicable to quantum systems now defined, investigating quantum chaos can now be performed with some rigor. Applying this definition with better resolvability and more state vectors is required to definitely state "second harmonic generation demonstrates chaos". A study investigating the application of this definition with different grouping schemes would also prove interesting and valuable.

Computer codes which simulate quantum state diffusion, for realisable quantum systems, open the way to modeling more complicated systems. With the computation time greatly reduced, the accuracy of these models can be increased.

Appendix A

Computer Code

This appendix contains one version of the code written. This code is written in Fortran 90 only using standard features and the NAG libraries. The documentation is basic, highlighting what each part of the code performs. Once all the components are compiled, only section A.13 needs to be changed and recompiled to control the output of the integration and the post processing performed.

A.1 Compiling and Running the Code

Before the code can be executed, a certain directory structure needs to be setup. From the directory where the executable will be run a directory "data" needs to be created. This directory is used for storing data from the program. Another directory that is needed is "vfl/data" where large data files are stored. On the VPP300 super computer there is a high speed raid array called "/vfl". The directory "vfl" was a symbolic link to a directory in "/vfl" on the high speed disk array, with a sub directory called "data". These directories can be changed in the code to what ever is desired.

In order to run the code, it is necessary to build the executable. The individual modules need to be compiled in a specific order, this being

- Precision A.5
- Globals A.6
- Random number generator A.7
- Operators A.8
- Matrix multiplication A.9
- Basis change A.10
- Integration A.11
- Post processing A.12
- Integrate second harmonic generation A.13

With all the components built the executable can be run. The final program is designed to be setup for multiple runs. That is, at the end of a run the final state vector and random number seed is saved to disk so that the next run can continue on. This reduces overall computation time, as the transient period does not need to be continually integrated. It also allows very long trajectories to be integrated and partial results to be obtained.

The only input into the program is a small file called "job_no" which has two integers in it. The first is the current job number and the second is the total number of runs being performed. This allows self submitting batch jobs to be performed. All other controls of the program are performed within A.13, through the *params* variable and by commenting out various parts of the code. This program must be recompiled after each change.

A.1.1 Sample Setups of the Code

There were three types of results obtained from the code: trajectories in the α plane; Q-functions of average reduced density operators; and the average information required to reduce the average system entropy to some tolerable level. The integration code A.13 needs to be modified to perform each of these tasks. Calls to the integration scheme "integrate", A.11, the initial state vector, starting integration time, finishing integration time, number of integration steps and the controlling parameters need to be defined. These modifications are made directly to the code A.13, followed by the code being recompiled.

The controlling parameters "params" are:

- "params%realisation" is the trajectory number and is loaded from job_no by the program
- "params%save_waves" is set to true if sate vectors are to be save to disk for post processing (computing the entropy/information)
- "params%wave_saves" is the number of state vectors per trajectory to save to disk
- "params%save_eas" is set to true if the trajectory in the α plane is to be save to disk
- params%ea_saves" is the number of points in the trajectory to save to disk
- "params%compute_D" is set to true to compute, and save to disk, the average reduced density operators, for mode 1 and 2, for post processing (computing the Q-functions)
- "params%D_computes" is the number of reduced density operators to average over in a single trajectory
- "params%S" is the quantum/classical scaling parameter
- "params%d1" is the detuning of the cavity with the external driving field
- "params%d2" is the detuning of the cavity with the second harmonic of the external driving field
- "params%K" is the ratio of the damping for each mode of the cavity
- "params%E" is the amplitude of the external driving field

To produce any results the parameters controlling the physics of a second harmonic generation scheme Ch. 2 need to be defined: params%S; params%d1; params%d2; params%K; and params%E.

To produce a trajectory in the complex α plane "params%save_eas" needs to be set to true and "params%ea_saves" needs to be set to the number of points in the trajectory. The time periods for calls to "integrate" also need to be specified. No post processing is needed for producing the trajectories, so the post processing lines are commented out. A sample setup is provided in A.2.

To produce the Q-function of the average reduced density operators "params%compute_D" needs to set to true and "params$D_computes" needs to be set to the number of reduced density operators to average over per trajectory. Again, the time periods to integrate over need to be specified. During the integration scheme, the average reduced density operator will be saved to disk for post processing. The post processing required is "average_density" which will average the average reduced density operators from a number of trajectories to produce one final average reduced density operator for each mode. Once this is performed "q_function" needs to be called to compute the Q-function of each mode. When calling "average_density" "params" and "ensemble" (total number of trajectories) need to be provided. When calling "q_function" the grid of α values needs to be specified along with "params". A sample setup is provided in A.3.

To produce the entropy/information data, "params%save_waves" needs to be set to true and "params%wave_saves" needs to be set to the number of state vectors to save to disk for post processing. The post processing required to compute the average change in information and average change in entropy of the system is "entropy_information", which requires "params", "ensemble" and the grid defining the distances for forming the groups. A sample setup is provided in A.4.

A.2 Integrate Second Harmonic Generation - Trajectory

```
program shg_program

  !
  !
  !  This program will integrate a system, and perform the necessary
  !  post processing
  !
  !

  use integrate_sde
  use operators
  use rand_num_gen
  use globals
  use basis_change
  use post_process

  !
  !
  !  n  is the initial size of the bases
  !  y, y1  are state vectors for integration
  !  loop  is a loop variable
  !  ensemble  is the number of trajectories being integrated
  !  realisation  is the current trajectory
  !  r  is a complex temporary random number
  !  params  is the parameters that control the integration
  !  damp_period  is the damping period of the system
  !
  !

  integer(idp) :: n=8
  type(wave_fn) :: y, y1
  integer(idp) :: loop, ensemble, realisation
  complex(cdp) :: r
  type(parameters) :: params
  real(fdp) :: damp_period


  !
  !
  !  Loading what job number this run is
  !
  !

  open(unit=1, file='job_no', action='READ')

  read(1,*) realisation,ensemble

  close(1)

  !
  !
  !  Setup the initial random numebr seed
  !
  !
```

```
if (realisation==1) then

   r=random_num('r')
   r=random_num('s')

end if

!
!
!  Setup the square root of integers
!
!

if (first_sqrt) then

   sq_rt=dble(0)

   do loop=0,max_basis+3

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Load in the random number seed
!
!

r=random_num('l')

!
!
!  If this is the first run, setup the initial state vector
!  If it is not the initial run, the saved wave fucntion is loaded
!
!

if (realisation==1) then

   y%in_basis(1)%n=n
   y%in_basis(1)%alpha=cmplx(2.1d0,0.0d0)
   y%in_basis(2)%n=n
   y%in_basis(2)%alpha=cmplx(1.0d0,0.0d0)

   allocate(y%co_eff(0:n,0:n))

   y%co_eff=cmplx(0.0d0,0.0d0)
   y%co_eff(0,0)=cmplx(1.0d0)

   y%co_eff = y%co_eff*normalisation(y)
```

```
else

   open(unit=1, file='data/tmp_basis',action='read',form='unformatted')

   read(1) y%in_basis

   close(1)

   allocate(y%co_eff(0:y%in_basis(1)%n,0:y%in_basis(2)%n))

   open(unit=1, file='data/tmp_wave',action='read',form='unformatted')

   read(1) y%co_eff

   close(1)

end if

!
!
!  The parameters for the run are setup
!
!

params%realisation=realisation
params%save_waves=.false.
params%wave_saves=10
params%save_eas=.false.
params%ea_saves=100000
params%compute_D=.false.
params%D_computes=50
params%S=1.25
params%d1=-1.0d0
params%d2=-1.0d0
params%K=0.25d0
params%E=3.1d1

!
!
!  The damping period is set
!
!

damp_period=1/params%K

!
!  If this is the initial run, the initial transients are removed
!

if (realisation==1) then

   y1=integrate(y,0.0d0,3.0d0,300000,params)

   deallocate(y%co_eff)

   y=y1
```

```
  end if

  !
  !
  !  The damping period is integrated for to make sure that the state is
  !  not correlated to its past.
  !
  !

!!$  y1=integrate(y,0.0d0,damp_period,500000, params)
!!$
!!$  deallocate(y%co_eff)

  !
  !
  !  This is where the various data files are choosen
  !
  !

!!$  params%compute_D=.true.
!!$  params%save_waves=.true.
  params%save_eas=.true.

  !
  !
  !  This is where the integration routine is called
  !
  !

  y=integrate(y1,0.0d0,4.0d0,500000, params)

  !
  !
  !  This is where the state vector final state vectors is stored
  !  to allow it to be loaded at the beginning of the next run
  !
  !

  open(unit=1, file='data/tmp_basis',action='write',form='unformatted')

  write(1) y%in_basis

  close(1)

  open(unit=1, file='data/tmp_wave',action='write',form='unformatted')

  write(1) y%co_eff

  close(1)

  deallocate(y%co_eff, y1%co_eff)

  !
  !
  !  The randomseed is saved
```

```
  !
  !

  r=random_num('s')

  if (realisation==ensemble) then

     !
     !
     !  These are the post processing options
     !
     !  average_density will average the density operators obtained in a
     !  multiple trajectory run
     !
     !  q_function will compute the q-function of the average reduced density
     !  operators
     !
     !  entropy_information will compute the entropy and information for
     !  for different non-overlapping groupings of the state vectors
     !
     !

!!$     call average_density(params, ensemble)
!!$     call q_function(-2.0d1,0.4d0,2.0d1,-2.0d1,0.4d0,2.0d1, params)
!!$     call entropy_information(params, ensemble, 0.0d0, 0.2d0, 1.0d1)

  end if

end program shg_program
```

A.3 Integrate Second Harmonic Generation - Q-function

```
program shg_program

  !
  !
  !  This program will integrate a system, and perform the necessary
  !  post processing
  !
  !

  use integrate_sde
  use operators
  use rand_num_gen
  use globals
  use basis_change
  use post_process

  !
  !
  !  n  is the initial size of the bases
  !  y, y1  are state vectors for integration
  !  loop  is a loop variable
  !  ensemble  is the number of trajectories being integrated
  !  realisation  is the current trajectory
  !  r  is a complex temporary random number
  !  params  is the parameters that control the integration
  !  damp_period  is the damping period of the system
  !
  !

  integer(idp) :: n=8
  type(wave_fn) :: y, y1
  integer(idp) :: loop, ensemble, realisation
  complex(cdp) :: r
  type(parameters) :: params
  real(fdp) :: damp_period


  !
  !
  !  Loading what job number this run is
  !
  !

  open(unit=1, file='job_no', action='READ')

  read(1,*) realisation,ensemble

  close(1)

  !
  !
  !  Setup the initial random numebr seed
  !
  !
```

```
if (realisation==1) then

   r=random_num('r')
   r=random_num('s')

end if

!
!
!  Setup the square root of integers
!
!

if (first_sqrt) then

   sq_rt=dble(0)

   do loop=0,max_basis+3

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Load in the random number seed
!
!

r=random_num('l')

!
!
!  If this is the first run, setup the initial state vector
!  If it is not the initial run, the saved wave fucntion is loaded
!
!

if (realisation==1) then

   y%in_basis(1)%n=n
   y%in_basis(1)%alpha=cmplx(2.1d0,0.0d0)
   y%in_basis(2)%n=n
   y%in_basis(2)%alpha=cmplx(1.0d0,0.0d0)

   allocate(y%co_eff(0:n,0:n))

   y%co_eff=cmplx(0.0d0,0.0d0)
   y%co_eff(0,0)=cmplx(1.0d0)

   y%co_eff = y%co_eff*normalisation(y)
```

```
else

   open(unit=1, file='data/tmp_basis',action='read',form='unformatted')

   read(1) y%in_basis

   close(1)

   allocate(y%co_eff(0:y%in_basis(1)%n,0:y%in_basis(2)%n))

   open(unit=1, file='data/tmp_wave',action='read',form='unformatted')

   read(1) y%co_eff

   close(1)

end if

!
!
!  The parameters for the run are setup
!
!

params%realisation=realisation
params%save_waves=.false.
params%wave_saves=10
params%save_eas=.false.
params%ea_saves=100000
params%compute_D=.false.
params%D_computes=50
params%S=1.25
params%d1=-1.0d0
params%d2=-1.0d0
params%K=0.25d0
params%E=3.1d1

!
!
!  The damping period is set
!
!

damp_period=1/params%K

!
!  If this is the initial run, the initial transients are removed
!

if (realisation==1) then

   y1=integrate(y,0.0d0,3.0d0,300000,params)

   deallocate(y%co_eff)

   y=y1
```

```
  end if

  !
  !
  !  The damping period is integrated for to make sure that the state is
  !  not correlated to its past.
  !
  !

!!$  y1=integrate(y,0.0d0,damp_period,500000, params)
!!$
!!$  deallocate(y%co_eff)

  !
  !
  !  This is where the various data files are choosen
  !
  !

  params%compute_D=.true.
!!$  params%save_waves=.true.
!!$  params%save_eas=.true.

  !
  !
  !  This is where the integration routine is called
  !
  !

  y=integrate(y1,0.0d0,4.0d0,500000, params)

  !
  !
  !  This is where the state vector final state vectors is stored
  !  to allow it to be loaded at the beginning of the next run
  !
  !

  open(unit=1, file='data/tmp_basis',action='write',form='unformatted')

  write(1) y%in_basis

  close(1)

  open(unit=1, file='data/tmp_wave',action='write',form='unformatted')

  write(1) y%co_eff

  close(1)

  deallocate(y%co_eff, y1%co_eff)

  !
  !
  !  The randomseed is saved
```

```
  !
  !

  r=random_num('s')

  if (realisation==ensemble) then

     !
     !
     !  These are the post processing options
     !
     !  average_density will average the density operators obtained in a
     !  multiple trajectory run
     !
     !  q_function will compute the q-function of the average reduced density
     !  operators
     !
     !  entropy_information will compute the entropy and information for
     !  for different non-overlapping groupings of the state vectors
     !
     !

     call average_density(params, ensemble)
     call q_function(-2.0d1,0.4d0,2.0d1,-2.0d1,0.4d0,2.0d1, params)
!!$     call entropy_information(params, ensemble, 0.0d0, 0.2d0, 1.0d1)

  end if

end program shg_program
```

A.4 Integrate Second Harmonic Generation - Entropy/Information

```
program shg_program

  !
  !
  !  This program will integrate a system, and perform the necessary
  !  post processing
  !
  !

  use integrate_sde
  use operators
  use rand_num_gen
  use globals
  use basis_change
  use post_process

  !
  !
  !  n  is the initial size of the bases
  !  y, y1  are state vectors for integration
  !  loop  is a loop variable
  !  ensemble  is the number of trajectories being integrated
  !  realisation  is the current trajectory
  !  r  is a complex temporary random number
  !  params  is the parameters that control the integration
  !  damp_period  is the damping period of the system
  !
  !

  integer(idp) :: n=8
  type(wave_fn) :: y, y1
  integer(idp) :: loop, ensemble, realisation
  complex(cdp) :: r
  type(parameters) :: params
  real(fdp) :: damp_period


  !
  !
  !  Loading what job number this run is
  !
  !

  open(unit=1, file='job_no', action='READ')

  read(1,*) realisation,ensemble

  close(1)

  !
  !
  !  Setup the initial random numebr seed
  !
  !
```

```
if (realisation==1) then

   r=random_num('r')
   r=random_num('s')

end if

!
!
!  Setup the square root of integers
!
!

if (first_sqrt) then

   sq_rt=dble(0)

   do loop=0,max_basis+3

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Load in the random number seed
!
!

r=random_num('l')

!
!
!  If this is the first run, setup the initial state vector
!  If it is not the initial run, the saved wave fucntion is loaded
!
!

if (realisation==1) then

   y%in_basis(1)%n=n
   y%in_basis(1)%alpha=cmplx(2.1d0,0.0d0)
   y%in_basis(2)%n=n
   y%in_basis(2)%alpha=cmplx(1.0d0,0.0d0)

   allocate(y%co_eff(0:n,0:n))

   y%co_eff=cmplx(0.0d0,0.0d0)
   y%co_eff(0,0)=cmplx(1.0d0)

   y%co_eff = y%co_eff*normalisation(y)
```

```
else

   open(unit=1, file='data/tmp_basis',action='read',form='unformatted')

   read(1) y%in_basis

   close(1)

   allocate(y%co_eff(0:y%in_basis(1)%n,0:y%in_basis(2)%n))

   open(unit=1, file='data/tmp_wave',action='read',form='unformatted')

   read(1) y%co_eff

   close(1)

end if

!
!
!  The parameters for the run are setup
!
!

params%realisation=realisation
params%save_waves=.false.
params%wave_saves=10
params%save_eas=.false.
params%ea_saves=100000
params%compute_D=.false.
params%D_computes=50
params%S=1.25
params%d1=-1.0d0
params%d2=-1.0d0
params%K=0.25d0
params%E=3.1d1

!
!
!  The damping period is set
!
!

damp_period=1/params%K

!
!  If this is the initial run, the initial transients are removed
!

if (realisation==1) then

   y1=integrate(y,0.0d0,3.0d0,300000,params)

   deallocate(y%co_eff)

   y=y1
```

```
  end if

  !
  !
  !  The damping period is integrated for to make sure that the state is
  !  not correlated to its past.
  !
  !

  y1=integrate(y,0.0d0,damp_period,500000, params)

  deallocate(y%co_eff)

  !
  !
  !  This is where the various data files are choosen
  !
  !

!!$  params%compute_D=.true.
  params%save_waves=.true.
!!$  params%save_eas=.true.

  !
  !
  !  This is where the integration routine is called
  !
  !

  y=integrate(y1,0.0d0,4.0d0,500000, params)

  !
  !
  !  This is where the state vector final state vectors is stored
  !  to allow it to be loaded at the beginning of the next run
  !
  !

  open(unit=1, file='data/tmp_basis',action='write',form='unformatted')

  write(1) y%in_basis

  close(1)

  open(unit=1, file='data/tmp_wave',action='write',form='unformatted')

  write(1) y%co_eff

  close(1)

  deallocate(y%co_eff, y1%co_eff)

  !
  !
  !  The randomseed is saved
```

```
  !
  !

  r=random_num('s')

  if (realisation==ensemble) then

     !
     !
     !  These are the post processing options
     !
     !  average_density will average the density operators obtained in a
     !  multiple trajectory run
     !
     !  q_function will compute the q-function of the average reduced density
     !  operators
     !
     !  entropy_information will compute the entropy and information for
     !  for different non-overlapping groupings of the state vectors
     !
     !

!!$     call average_density(params, ensemble)
!!$     call q_function(-2.0d1,0.4d0,2.0d1,-2.0d1,0.4d0,2.0d1, params)
     call entropy_information(params, ensemble, 0.0d0, 0.2d0, 1.0d1)

  end if

end program shg_program
```

A.5 Precision

```
module precision

  !
  !
  !  This module will assign fdp (floating point double precision)
  !  isp (integer single precision), idp (integer double precision)
  !  csp (complex single precision), cdp (complex double precision)
  !  fqp (floating point quadratic precision)
  !
  !

  implicit none

  integer, parameter :: fqp=selected_real_kind(20,200)
  integer, parameter :: fdp=selected_real_kind(10,200)
  integer, parameter :: fsp=selected_real_kind(5,50)
  integer, parameter :: isp=selected_int_kind(6)
  integer, parameter :: idp=selected_int_kind(9)
  integer, parameter :: cdp=fdp
  integer, parameter :: csp=fsp

end module precision
```

A.6 Globals

```
module globals

  !
  !
  !  This module contains the global variables and derived types
  !
  !

  use precision

  public



  !
  !
  !  cut_off1 is the maximum size of the state vector at the last basis vector
  !  cut_off2 is the minimum size of the state vector at the last basis vector
  !
  !

  real(fdp), parameter :: cut_off1=1.0d-3, cut_off2=cut_off1*1.0d-3



  !
  !
  !  max_basis is the maximum size of the bases
  !
  !

  integer(idp), parameter :: max_basis=500



  !
  !
  !  pi is the parameter value of pi
  !
  !

  real(fdp), parameter :: pi=3.14159265359



  !
  !
  !  hbar is the parameter value of the physical constant hbar
  !
  !

  real(fdp), parameter :: hbar=1.01d-34
```

```
!
!
!  im  is the complex variable i
!
!

complex(cdp), parameter :: im=(0.0d0,1.0d0)



!
!
!  sq_rt  is a vector of square roots of integers
!  first_sqrt  is set when sq_rt has been setup
!
!

real(fdp), dimension(-3:max_basis+3), save :: sq_rt
logical, save :: first_sqrt=.true.



type basis

   !
   !
   !  This derived type defines a basis
   !
   !  alpha is the displacement
   !  n is the number of basis states being used
   !
   !

   complex(cdp) :: alpha
   integer(isp) :: n

end type basis



type wave_fn

   !
   !
   !  This derived type defines a state vector
   !
   !  co_eff  are the co-efficients of the state vector
   !  in_basis  is the basis that the state vector is represented on
   !
   !

   complex(cdp), dimension(:,:), pointer :: co_eff
   type (basis), dimension(2) :: in_basis

end type wave_fn
```

```
  type parameters

     !
     !
     !  This derived type defines the parameters used for integration.
     !
     !  realisation  is the trajectory number for multiple trajectory runs
     !  wave_saves  is the number of times the state vector is saved to disk
     !  save_waves  is set when wave functions are to be saved to disk
     !  ea_saves  is the number of points <y|a|y> to save to disk
     !  save_eas  is set when a trajectory <y|a|y> is to be save to disk
     !  compute_D  is set when the reduced density operators are to be computed
     !  D_computes  is the number of reduced density operators to average over
     !
     !  The following variables are the scaled variables for defining
     !  second harmonic generation
     !
     !  S  is the quantum/classical scaling parameter
     !  d1, d2  are the detunings of the cavity
     !  K  is the ratio of the dampings of the cavity
     !  E  is the amplitude of the driving field
     !
     !


     integer(idp) :: realisation
     integer(idp) :: wave_saves
     logical :: save_waves
     integer(idp) :: ea_saves
     logical :: save_eas
     logical :: compute_D
     integer(idp) :: D_computes
     real(fdp) :: S
     real(fdp) :: d1 ,d2
     real(fdp) :: K
     real(fdp) :: E

  end type parameters


end module globals
```

A.7 Random Number Generator

```
module rand_num_gen

  !
  !
  !  This module contains a routine to generate random numbers
  !
  !  This is taken from Numerical Recipes for Fortran90
  !
  !  The routine is called "ran" in Numerical Recipes.
  !
  !  It has been modified slightly to make it easily used.  The basic workings
  !  of it have not been changed in any way.
  !
  !

  use precision

  implicit none

  integer(idp), save, private :: idum=-1, ix=-1, iy=-1, c
  real(fdp), save, private :: am


contains



  function random_num(init) result(number)

    !
    !
    !  This function returns a random number.  The optional input init
    !  can be one of three values r, s and l
    !
    !  r  resets the random number generator to its initial seed
    !  s  saves the current variables to disk
    !  l  loads the last saved variables from disk
    !
    !

    character, optional, intent(in) :: init
    real(fdp) :: number
    integer(idp), parameter :: ia=16807, im=2147483647, iq=127773, ir=2836


    if (present(init)) then

       select case (init)

       case('r')

          idum=-1

       case('s')
```

```
            open(unit=1, action='WRITE',file='data/random_save.dat', &
                 form='unformatted')

            write(1) idum, ix, iy, c, am

            close(1)

         case('l')

            open(unit=1, action='READ',file='data/random_save.dat', &
                 form='unformatted')

            read(1) idum, ix, iy, c, am

            close(1)

         case default

         end select

      end if

      if (idum<=0 .or. iy < 0) then

         am=nearest(1.0,-1.0)/im
         iy=ior(ieor(888889999,abs(idum)),1)
         ix=ieor(777755555,abs(idum))
         idum=abs(idum)+1

      end if

      ix=ieor(ix,ishft(ix,13))
      ix=ieor(ix,ishft(ix,-17))
      ix=ieor(ix,ishft(ix,5))
      c=iy/iq
      iy=ia*(iy-c*iq)-ir*c

      if (iy < 0) iy=iy+im

      number=am*ior(iand(im,ieor(ix,iy)),1)

    end function random_num



end module rand_num_gen
```

A.8 Operators

```
module operators

  !
  !
  !  This module contains a lot of routines used by other modules to
  !  evaluate certain operators, normalisation constants and density
  !  operators.
  !
  !

  use precision
  use globals


  interface operator (.af.)

     !
     !
     !  This operator (af) is the annihilation operator for mode 1
     !
     !  expectation_a1 is the function that calculates <y|a|y> for mode 1
     !  oper_a1 is the function that calculates a|y> for mode 1
     !
     !

     module procedure expectation_a1
     module procedure oper_a1
  end interface


  interface operator (.adf.)

     !
     !
     !  This operator (adf) is the creation operator for mode 1
     !
     !  expectation_ad1 is the function that calculates <y|ad|y> for mode 1
     !  oper_ad1 is the function that calculates ad|y> for mode 1
     !
     !

     module procedure expectation_ad1
     module procedure oper_ad1
  end interface


  interface operator (.ah.)

     !
     !
     !  This operator (ah) is the annihilation operator for mode 2
     !
     !  expectation_a2 is the function that calculates <y|a|y> for mode 2
     !  oper_a2 is the function that calculates a|y> for mode 2
     !
```

```
     !

     module procedure expectation_a2
     module procedure oper_a2
  end interface


  interface operator (.adh.)

     !
     !
     !  This operator (adh) is the creation operator for mode 2
     !
     !  expectation_ad2 is the function that calculates <y|ad|y> for mode 2
     !  oper_ad2 is the function that calculates ad|y> for mode 2
     !
     !

     module procedure expectation_ad2
     module procedure oper_ad2
  end interface



contains



  function oper_a1(y) result(tmpy)

    !
    !
    !  This function calculates a|y> for mode 1
    !
    !  y  is the state vector in the operation a|y>
    !  tmpy  is the result of the operation
    !  alpha  is the point that the basis is centered at
    !  loop  is used to setup the global variable sq_rt
    !  n1, n2  are the size of the bases in mode 1 and mode 2
    !  i, j  loop over the basis states of mode 1 and mode 2
    !
    !

    type(wave_fn), intent(in) :: y
    type(wave_fn) :: tmpy
    integer(idp) :: loop, n1, n2, i, j
    complex(cdp) :: alpha


    !
    !
    !  Setup the global variable sq_rt which holds the square root of
    !  integers in a vector.
    !
    !
```

```
  if (first_sqrt) then

     do loop=0,max_basis+1

        sq_rt(loop)=sqrt(dble(loop))

     end do

     first_sqrt=.false.

  end if

  !
  !
  !  Setup the returned variable tmpy
  !
  !

  tmpy%in_basis = y%in_basis

  n1=tmpy%in_basis(1)%n
  n2=tmpy%in_basis(2)%n

  allocate(tmpy%co_eff(0:n1,0:n2))

  tmpy%co_eff=cmplx(0)

  !
  !
  !  Compute the result of a|y>
  !
  !

  alpha = y%in_basis(1)%alpha

  do j=0,n2
     do i=0,n1-1

        tmpy%co_eff(i,j) = sq_rt(i+1)*y%co_eff(i+1,j) + alpha*y%co_eff(i,j)

     end do
  end do

  tmpy%co_eff(n1,:) = alpha*y%co_eff(n1,:)

end function oper_a1



function expectation_a1(y1,y2) result(alpha)

  !
  !
  !  This function calculates <y1|a|y2> for mode 1
  !
  !  y1, y2  are the state vectors in the operation <y1|a|y2>
```

```
  !  alpha  is the results of computing <y1|a|y2>
  !  loop  is used to setup the global variable sq_rt
  !  minn1, minn2  are the minimum size of the bases in mode 1 and mode 2
  !  i, j  loop over the basis states of mode 1 and mode 2
  !
  !

  type(wave_fn), intent(in) :: y1, y2
  complex(cdp) :: alpha
  integer(idp) :: loop, minn1, minn2, i, j


  !
  !
  !  Setup the global variable sq_rt which holds the square root of
  !  integers in a vector.
  !
  !

  if (first_sqrt) then

     do loop=0,max_basis+1

        sq_rt(loop)=sqrt(dble(loop))

     end do

     first_sqrt=.false.

  end if

  minn1=min(y1%in_basis(1)%n,y2%in_basis(1)%n)
  minn2=min(y1%in_basis(2)%n,y2%in_basis(2)%n)

  !
  !
  !  Compute the result of <y1|a|y2>
  !
  !

  alpha=y2%in_basis(1)%alpha

  do j=0,minn2
     do i=0,minn1-1

        alpha=alpha+sq_rt(i+1)*conjg(y1%co_eff(i,j))*y2%co_eff(i+1,j)

     end do
  end do

end function expectation_a1



function oper_ad1(y) result(tmpy)
```

```
!
!
!  This function calculates ad|y> for mode 1
!
!  y  is the state vector in the operation ad|y>
!  tmpy  is the result of the operation
!  calpha  is the conjugate of the point that the basis is centered at
!  loop  is used to setup the global variable sq_rt
!  n1, n2  are the size of the bases in mode 1 and mode 2
!  i, j  loop over the basis states of mode 1 and mode 2
!
!


type(wave_fn), intent(in) :: y
type(wave_fn) :: tmpy
integer(idp) :: loop, n1, n2, i, j
complex(cdp) :: calpha


!
!
!  Setup the global variable sq_rt which holds the square root of
!  integers in a vector.
!
!

if (first_sqrt) then

   do loop=0,max_basis+1

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Setup the returned variable tmpy
!
!

tmpy%in_basis = y%in_basis

n1=tmpy%in_basis(1)%n
n2=tmpy%in_basis(2)%n

allocate(tmpy%co_eff(0:n1,0:n2))

tmpy%co_eff=cmplx(0.0d0)

!
!
```

```
  !  Compute the result of ad|y>
  !
  !

  calpha=conjg(y%in_basis(1)%alpha)

  do j=0,n2
     do i=1,n1

        tmpy%co_eff(i,j)=sq_rt(i)*y%co_eff(i-1,j)+calpha*y%co_eff(i,j)

     end do
  end do

  tmpy%co_eff(0,:) = calpha*y%co_eff(0,:)

end function oper_ad1



function expectation_ad1(y1,y2) result(alpha)

  !
  !
  !  This function calculates <y1|ad|y2> for mode 1
  !
  !  y1, y2  are the state vectors in the operation <y1|ad|y2>
  !  alpha  is the value of <y1|ad|y2>
  !  loop  is used to setup the global variable sq_rt
  !  minn1, minn2  are the minimum size of the bases in mode 1 and mode 2
  !  i, j  loop over the basis states of mode 1 and mode 2
  !
  !


  type(wave_fn), intent(in) :: y1, y2
  complex(cdp) :: alpha
  integer(idp) :: loop, minn1, minn2, i, j


  !
  !
  !  Setup the global variable sq_rt which holds the square root of
  !  integers in a vector.
  !
  !

  if (first_sqrt) then

     do loop=0,max_basis+1

        sq_rt(loop)=sqrt(dble(loop))

     end do

     first_sqrt=.false.
```

```
  end if

  minn1=min(y1%in_basis(1)%n,y2%in_basis(1)%n)
  minn2=min(y1%in_basis(2)%n,y2%in_basis(2)%n)

  !
  !
  !  Compute the result of <y1|ad|y2>
  !
  !

  alpha=conjg(y2%in_basis(1)%alpha)

  do j=0,minn2
     do i=1,minn1

        alpha=alpha+sq_rt(i)*conjg(y1%co_eff(i,j))*y2%co_eff(i-1,j)

     end do
  end do

end function expectation_ad1



function oper_a2(y) result(tmpy)

  !
  !
  !  This function calculates a|y> for mode 2
  !
  !  y  is the state vector in the operation a|y>
  !  tmpy  is the result of the operation
  !  alpha  is the point that the basis is centered at
  !  loop  is used to setup the global variable sq_rt
  !  n1, n2  are the size of the bases in mode 1 and mode 2
  !  i, j  loop over the basis states of mode 1 and mode 2
  !
  !


  type(wave_fn), intent(in) :: y
  type(wave_fn) :: tmpy
  integer(idp) :: loop, n1,n2,i,j
  complex(cdp) :: alpha


  !
  !
  !  Setup the global variable sq_rt which holds the square root of
  !  integers in a vector.
  !
  !

  if (first_sqrt) then
```

```
      do loop=0,max_basis+1

         sq_rt(loop)=sqrt(dble(loop))

      end do

      first_sqrt=.false.

   end if

   !
   !
   !  Setup the returned variable tmpy
   !
   !

   tmpy%in_basis = y%in_basis

   n1=tmpy%in_basis(1)%n
   n2=tmpy%in_basis(2)%n

   allocate(tmpy%co_eff(0:n1,0:n2))

   tmpy%co_eff=cmplx(0)

   !
   !
   !  Compute the result of a|y>
   !
   !

   alpha = y%in_basis(2)%alpha

   do j=0,n2-1
      do i=0,n1

         tmpy%co_eff(i,j) = sq_rt(j+1)*y%co_eff(i,j+1) + alpha*y%co_eff(i,j)

      end do
   end do

   tmpy%co_eff(:,n2) = alpha*y%co_eff(:,n2)

end function oper_a2



function expectation_a2(y1,y2) result(alpha)

   !
   !
   !  This function calculates <y1|a|y2> for mode 2
   !
   !  y1, y2  are the state vectors in the operation <y1|a|y2>
   !  alpha  is the result of <y1|a|y2>
```

```
  !  loop  is used to setup the global variable sq_rt
  !  minn1, minn2  are the minimum size of the bases in mode 1 and mode 2
  !  i, j  loop over the basis states of mode 1 and mode 2
  !
  !


  type(wave_fn), intent(in) :: y1, y2
  complex(cdp) :: alpha
  integer(idp) :: loop, minn1, minn2, i, j


  !
  !
  !  Setup the global variable sq_rt which holds the square root of
  !  integers in a vector.
  !
  !

  if (first_sqrt) then

     do loop=0,max_basis+1

        sq_rt(loop)=sqrt(dble(loop))

     end do

     first_sqrt=.false.

  end if

  minn1=min(y1%in_basis(1)%n,y2%in_basis(1)%n)
  minn2=min(y1%in_basis(2)%n,y2%in_basis(2)%n)

  !
  !
  !  Compute the result of <y1|a|y2>
  !
  !

  alpha=y2%in_basis(2)%alpha

  do j=0,minn2-1
     do i=0,minn1

        alpha=alpha+sq_rt(j+1)*conjg(y1%co_eff(i,j))*y2%co_eff(i,j+1)

     end do
  end do

end function expectation_a2



function oper_ad2(y) result(tmpy)
```

```
!
!
!  This function calculates ad|y> for mode 2
!
!  y  is the state vector in the operation ad|y>
!  tmpy  is the result of the operation
!  calpha  is the conjugate of the point that the basis is centered at
!  loop  is used to setup the global variable sq_rt
!  n1, n2  are the size of the bases in mode 1 and mode 2
!  i, j  loop over the basis states of mode 1 and mode 2
!
!


type(wave_fn), intent(in) :: y
type(wave_fn) :: tmpy
integer(idp) :: loop, n1, n2, i,j
complex(cdp) :: calpha


!
!
!  Setup the global variable sq_rt which holds the square root of
!  integers in a vector.
!
!

if (first_sqrt) then

   do loop=0,max_basis+1

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Setup the returned variable tmpy
!
!

tmpy%in_basis = y%in_basis

n1=tmpy%in_basis(1)%n
n2=tmpy%in_basis(2)%n

allocate(tmpy%co_eff(0:n1,0:n2))

tmpy%co_eff=cmplx(0.0d0)

!
!
```

```
  !  Compute the result of a|y>
  !
  !

  calpha=conjg(y%in_basis(2)%alpha)

  do j=1,n2
     do i=0,n1

        tmpy%co_eff(i,j)=sq_rt(j)*y%co_eff(i,j-1)+calpha*y%co_eff(i,j)

     end do
  end do

  tmpy%co_eff(:,0) = calpha*y%co_eff(:,0)

end function oper_ad2



function expectation_ad2(y1,y2) result(alpha)

  !
  !
  !  This function calculates <y1|ad|y2> for mode 2
  !
  !  y1, y2  are the state vectors in the operation <y1|ad|y2>
  !  alpha  is the results of <y1|ad|y2>
  !  loop  is used to setup the global variable sq_rt
  !  minn1, minn2  are the minimum size of the bases in mode 1 and mode 2
  !  i, j  loop over the basis states of mode 1 and mode 2
  !
  !


  type(wave_fn), intent(in) :: y1, y2
  complex(cdp) :: alpha
  integer(idp) :: loop, minn1, minn2, i, j


  !
  !
  !  Setup the global variable sq_rt which holds the square root of
  !  integers in a vector.
  !
  !

  if (first_sqrt) then

     do loop=0,max_basis+1

        sq_rt(loop)=sqrt(dble(loop))

     end do

     first_sqrt=.false.
```

```
  end if

  minn1=min(y1%in_basis(1)%n,y2%in_basis(1)%n)
  minn2=min(y1%in_basis(2)%n,y2%in_basis(2)%n)

  !
  !
  !  Compute the result of <y1|ad|y2>
  !
  !

  alpha=conjg(y2%in_basis(2)%alpha)

  do j=1,minn2
     do i=0,minn1

        alpha=alpha+sq_rt(j)*conjg(y1%co_eff(i,j))*y2%co_eff(i,j-1)

     end do
  end do

end function expectation_ad2



function normalisation(y) result(norm)

  !
  !
  !  This function will return a constant to normalise a state vector
  !
  !  y  is the state vector that is needed to be normalised
  !  norm  is the constant that will normalise the state vector y
  !  i, j  loop over all the co-efficients of y
  !
  !

  type(wave_fn), intent(in) :: y
  real(fdp) :: norm
  integer(idp) :: i,j


  !
  !
  !  Compute the normalisation constant
  !
  !

  norm=dble(0)

  do j=0,y%in_basis(2)%n
     do i=0,y%in_basis(1)%n

        norm=norm+y%co_eff(i,j)*conjg(y%co_eff(i,j))
```

```
      end do
   end do

   norm=sqrt(norm)

end function normalisation



function density1(y) result(tmpy)

   !
   !
   !  Computes the reduced density operator of mode 1, Tr2(|y><y|)
   !
   !  y  is the co-efficients of the state vector
   !  tmpy  is the returned reduced density operator
   !  i, k, n  are loop variables
   !
   !

   complex(cdp), dimension(0:max_basis, 0:max_basis) :: y, tmpy
   integer(idp) :: i,k,n


   !
   !
   !  Compute the reduced density operator Tr2(|y><y|)
   !
   !

   do k=0,max_basis

      do i=0,max_basis

         tmpy(i,k)=cmplx(0)

      end do

      do n=0,max_basis
         do i=0,max_basis

            tmpy(i,k) = tmpy(i,k) + y(i,n)*conjg(y(k,n))

         end do
      end do
   end do

end function density1



function density2(y) result(tmpy)

   !
   !
```

```
  !  Computes the reduced density operator of mode 2, Tr1(|y><y|)
  !
  !  y  is the co-efficients of the state vector
  !  tmpy  is the returned reduced density operator
  !  j, l, n  are loop variables
  !
  !

  complex(cdp), dimension(0:max_basis, 0:max_basis) :: y, tmpy
  integer(idp) :: j,l,n


  !
  !
  !  Compute the reduced density operator Tr1(|y><y|)
  !
  !

  do l=0,max_basis
     do j=0,max_basis

        tmpy(j,l)=cmplx(0)

        do n=0,max_basis

           tmpy(j,l) = tmpy(j,l) + y(n,j)*conjg(y(n,l))

        end do
     end do
  end do

end function density2



function entropy(A,m,n) result(ent)

  !
  !
  !  This function computes the entropy of the density operator given by
  !  1/N A A^{\dagger} using singular value decomposition of A
  !
  !  This function uses the NAG routine f02xef
  !
  !  A  is the matrix of co-efficients of the state vectors
  !  m,n  are the size of A
  !  ent  is the entropy of the density operator
  !  i  is a loop variable
  !
  !  the other variables are used by the NAG routine - see NAG documentation
  !
  !

  integer(idp), intent(in) :: m, n
  complex(cdp), dimension(m,n), intent(inout) :: A
  integer(idp) :: ifail, i
```

```
    real(fdp), dimension(min(m,n)) :: sv
    real(fdp), dimension(5*min(m,n)+100) :: rwork
    complex(cdp), dimension(2*n+100) :: cwork
    complex(cdp) :: Q,B,P
    real(fdp) :: ent


    !
    !
    !  Call the NAG routine
    !
    !

    ifail=0

    call f02xef(m,n,A,m,0,B,0,.false.,Q,0,sv,.false.,P,0,rwork, cwork,ifail)

    !
    !
    !  Compute the entropy of the density operator
    !
    !

    ent=dble(0)

    sv=sv/sqrt(dble(n))

    do i=1,min(m,n)

       ent=ent - sv(i)*sv(i)*log(sv(i)*sv(i))

    end do

    ent=ent/log(dble(2))

  end function entropy



end module operators
```

A.9 Matrix Multiplication

This routine was provided by Mr David Singleton of the Australian National University Supercomputer Facility. The version that came with the Fortran90 compiler on the VPP300 produced incorrect results for complex numbers. It is included for reference only.

```
module matrix

  interface mat_mul
     module procedure matmul_r4, matmul_r8, matmul_c8, matmul_c16
     module procedure matvec_r4, matvec_r8, matvec_c8, matvec_c16
  end interface

contains

!-------------------------------------------------------------
! REAL SINGLE PRECISION
!-------------------------------------------------------------
  function matmul_r4(a,b) result(mm)
    implicit none
    real, dimension(:,:) :: a,b
    real, dimension(size(a,1),size(b,2)) :: mm

    integer i, j, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix multiply arguments"
       stop
    end if

    mm=0.0
    do j = 1, size(b,2)
       do k = 1, size(a,2)
          do i = 1, size(a,1)
             mm(i,j) = mm(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo

    return
  end function matmul_r4

  function matvec_r4(a,b) result(mm)
    implicit none
    real, dimension(:,:) :: a
    real, dimension(:) :: b
    real, dimension(size(a,1)) :: mm

    integer i, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix vector multiply arguments"
       stop
    end if

    mm=0.0
```

```
    do k = 1, size(a,2)
       do i = 1, size(a,1)
          mm(i) = mm(i) + a(i,k)*b(k)
       enddo
    enddo

    return
  end function matvec_r4

!-------------------------------------------------------------
! REAL DOUBLE PRECISION
!-------------------------------------------------------------
  function matmul_r8(a,b) result(mm)
    implicit none
    real*8, dimension(:,:) :: a,b
    real*8, dimension(size(a,1),size(b,2)) :: mm

    integer i, j, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix multiply arguments"
       stop
    end if

    mm=0.0d0
    do j = 1, size(b,2)
       do k = 1, size(a,2)
          do i = 1, size(a,1)
             mm(i,j) = mm(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo

    return
  end function matmul_r8

  function matvec_r8(a,b) result(mm)
    implicit none
    real*8, dimension(:,:) :: a
    real*8, dimension(:) :: b
    real*8, dimension(size(a,1)) :: mm

    integer i, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix vector multiply arguments"
       stop
    end if

    mm=0.0d0

    do k = 1, size(a,2)
       do i = 1, size(a,1)
          mm(i) = mm(i) + a(i,k)*b(k)
       enddo
    enddo
```

```
    return
  end function matvec_r8

!-------------------------------------------------------------
! COMPLEX SINGLE PRECISION
!-------------------------------------------------------------
  function matmul_c8(a,b) result(mm)
    implicit none
    complex, dimension(:,:) :: a,b
    complex, dimension(size(a,1),size(b,2)) :: mm

    integer i, j, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix multiply arguments"
       stop
    end if

    mm=0.0
    do j = 1, size(b,2)
       do k = 1, size(a,2)
          do i = 1, size(a,1)
             mm(i,j) = mm(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo

    return
  end function matmul_c8

  function matvec_c8(a,b) result(mm)
    implicit none
    complex, dimension(:,:) :: a
    complex, dimension(:) :: b
    complex, dimension(size(a,1)) :: mm

    integer i, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix vector multiply arguments"
       stop
    end if

    mm=0.0

    do k = 1, size(a,2)
       do i = 1, size(a,1)
          mm(i) = mm(i) + a(i,k)*b(k)
       enddo
    enddo

    return
  end function matvec_c8

 !-------------------------------------------------------------
```

```
! COMPLEX DOUBLE PRECISION
!-------------------------------------------------------------
 function matmul_c16(a,b) result(mm)
    implicit none
    complex*16, dimension(:,:) :: a,b
    complex*16, dimension(size(a,1),size(b,2)) :: mm

    integer i, j, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix multiply arguments"
       stop
    end if

    mm=0.0d0
    do j = 1, size(b,2)
       do k = 1, size(a,2)
          do i = 1, size(a,1)
             mm(i,j) = mm(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo

    return
  end function matmul_c16

  function matvec_c16(a,b) result(mm)
    implicit none
    complex*16, dimension(:,:) :: a
    complex*16, dimension(:) :: b
    complex*16, dimension(size(a,1)) :: mm

    integer i, k

    if ( size(a,2) /= size(b,1) ) then
       write(*,*) "Cannot matrix vector multiply arguments"
       stop
    end if

    mm=0.0d0

    do k = 1, size(a,2)
       do i = 1, size(a,1)
          mm(i) = mm(i) + a(i,k)*b(k)
       enddo
    enddo

    return
  end function matvec_c16

end module matrix
```

A.10 Basis Change

```
module basis_change

  !
  !
  !  This module allows a change of basis for either mode 1 or 2 to be
  !  performed
  !
  !

  use precision
  use globals
  use operators
  use matrix

  implicit none

  !
  !
  !  delta  is the matrix which contains the transformation
  !
  !

  complex(cdp), dimension(:,:), allocatable, private :: delta


contains



  subroutine compute_delta(b1,b2)

    !
    !
    !  This computes the transformation matrix delta which transforms the
    !  state vector y1 in basis 1 to y2 in basis 2
    !
    !  b1, b2  are the bases that the transformation is to occur between
    !  facd  is a factor which is computed iteratively
    !  faccd  is a factor which is computed iteratively
    !  i,j,l  loop over mode 1, mode 2 and i,j respectively
    !  minij  is the minimum of i,j
    !  d  is the distance between the two bases
    !  cd  is the conjugate of d
    !  fac  is a factor to improve accuracy in the alternating sum
    !  dcd  is d*cd
    !
    !

    type(basis), intent(in) :: b1, b2
    complex(cdp), dimension(0:b2%n,0:b2%n) :: facd
    complex(cdp), dimension(0:b1%n,0:b1%n) :: faccd
    integer(idp) :: i,j,l,minij
    complex(cdp) :: cd, d, tmp
    real(fdp) :: fac, dcd
```

```
!
!
!  Compute d, cd and dcd
!
!

d=b1%alpha - b2%alpha
cd=conjg(d)
dcd=real(d*cd)

!
!
!  If dcd is 0 then a diagonal matrix is returned
!  otherwise delta is computed
!
!

if (dcd/=cmplx(0)) then

   !
   !
   !  Setup the diagonal elements for facd and faccd
   !
   !

   do i=0,b2%n

      facd(i,i)=cmplx(1)

   end do

   do j=0,b1%n

      faccd(j,j)=cmplx(1)

   end do

   !
   !
   !  Compute facd and faccd
   !
   !

   do i=1,b2%n
      do l=i-1,0,-1

         facd(l,i)=sq_rt(l+1)/dble(i-l)*d*facd(l+1,i)

      end do
   end do

   do j=1,b1%n
      do l=j-1,0,-1

         faccd(l,j)=-sq_rt(l+1)/dble(j-l)*cd*faccd(l+1,j)
```

```
      end do
   end do

   !
   !
   !  Compute delta from fac, facd and faccd
   !
   !

   do j = 0,b1%n
      do i = 0,b2%n

         tmp=cmplx(0)
         minij=min(i,j)

         do l = 0,minij-1,2

            !
            !
            !  fac converts the sum from an alternating sum to a normal
            !  sum to reduce rounding error
            !
            !

            fac=1.0d0 - (i-l)*(j-l)/(dcd*(l+1))

            tmp=tmp + facd(l,i)*faccd(l,j)*fac

         end do

         if ((minij/2)*2 == minij) then

            tmp=tmp + facd(minij,i)*faccd(minij,j)

         end if

         delta(i,j)=tmp

      end do
   end do

else

   !
   !
   !  Setup delta for the case when dcd=0
   !
   !

   delta=cmplx(0)

   do i=0,min(b1%n, b2%n)

      delta(i,i)=cmplx(1)

   end do
```

```
  end if

end subroutine compute_delta



subroutine change_basis1(y1,y2)

  !
  !
  !  This routine changes the basis for mode 1 of state vector y1 to
  !  state vector y2 on the new basis
  !
  !  Since the basis change does not work well for bases more than a
  !  distance 1 away from each other, the distance is broken into small
  !  steps of equal length (approximately 1).
  !
  !  y1  is the state vector to be moved
  !  y2  is the resulting moved state vector
  !  ty  is a temporary state vector
  !  minn1  is the minimum basis size for mode 1
  !  minn2  is the minimum basis size for mode 2
  !  d  is the distance to move the basis
  !  steps  is the number of smaller basis changes to do
  !  step is the distance to move for the smaller basis changes
  !
  !


  type(wave_fn), intent(in) :: y1
  type(wave_fn), intent(inout) :: y2
  type(wave_fn) :: ty
  integer(idp) :: minn1, minn2, loop, steps
  complex(cdp) :: d, step

  !
  !
  !  Compute the distance between the bases
  !
  !

  d=y2%in_basis(1)%alpha-y1%in_basis(1)%alpha

  !
  !
  !  Compute the number of steps to perform
  !
  !

  steps=ceiling(abs(d))

  if (steps==0) then

     steps=1
```

```
end if

!
!
!  Compute the distance for each smaller basis change
!
!

step=d/dble(steps)

!
!
!  Setup the temporary state vector
!
!

ty%in_basis=y1%in_basis

allocate(ty%co_eff(0:ty%in_basis(1)%n, 0:ty%in_basis(2)%n))

ty%co_eff=y1%co_eff

!
!
!  Perform the smaller basis changes
!
!

do loop=1,steps

   y2%in_basis(1)%alpha=ty%in_basis(1)%alpha+step

   if (loop <= 2 ) then

      !
      !
      !  Compute the transformation matix delta.  Since delta is only
      !  dependant on the distance between the bases it only needs to be
      !  computed twice, as the smaller basis changes are all in the
      !  same direction and are the same size.  The first for changing the
      !  basis size and the second for changing between bases of same size
      !
      !

      allocate(delta(0:y2%in_basis(1)%n,0:ty%in_basis(1)%n))

      call compute_delta(ty%in_basis(1),y2%in_basis(1))

      minn1=min(ty%in_basis(1)%n,y2%in_basis(1)%n)
      minn2=min(ty%in_basis(2)%n,y2%in_basis(2)%n)

   end if

   !
   !
   !  Change the basis
```

```
      !
      !

      y2%co_eff=cmplx(0)

      y2%co_eff(:,0:minn2) = mat_mul(delta,ty%co_eff(:,0:minn2))

      y2%co_eff = y2%co_eff*normalisation(y2)

      if (loop < 2) then

         deallocate(delta, ty%co_eff)

         ty%in_basis=y2%in_basis

         allocate(ty%co_eff(0:ty%in_basis(1)%n, 0:ty%in_basis(2)%n))

      end if

      ty%in_basis(1)%alpha=y2%in_basis(1)%alpha
      ty%co_eff=y2%co_eff

   end do

   if (allocated(delta)) then

      deallocate(delta)

   end if

   deallocate(ty%co_eff)

end subroutine change_basis1



subroutine change_basis2(y1,y2)

   !
   !
   !  This routine changes the basis for mode 2 of state vector y1 to
   !  state vector y2 on the new basis
   !
   !  Since the basis change does not work well for bases more than a
   !  distance 1 away from each other, the distance is broken into small
   !  steps of equal length (approximately 1).
   !
   !  y1  is the state vector to be moved
   !  y2  is the resulting moved state vector
   !  ty  is a temporary state vector
   !  minn1  is the minimum basis size for mode 1
   !  minn2  is the minimum basis size for mode 2
   !  d  is the distance to move the basis
   !  steps  is the number of smaller basis changes to do
   !  step  is the distance to move for the smaller basis changes
   !
```

```
!


type(wave_fn), intent(in) :: y1
type(wave_fn), intent(inout) :: y2
type(wave_fn) :: ty
integer(idp) :: minn1, minn2, loop, steps
complex(cdp) :: d, step

!
!
!  Compute the distance between the bases
!
!

d=y2%in_basis(2)%alpha-y1%in_basis(2)%alpha

!
!
!  Compute the number of steps to perform
!
!

steps=ceiling(abs(d))

if (steps==0) then

   steps=1

end if

!
!
!  Compute the distance for each smaller basis change
!
!

step=d/dble(steps)

!
!
!  Setup the temporary state vector
!
!

ty%in_basis=y1%in_basis

allocate(ty%co_eff(0:ty%in_basis(1)%n, 0:ty%in_basis(2)%n))

ty%co_eff=y1%co_eff

!
!
!  Perform the smaller basis changes
!
!
```

```
do loop=1,steps

   y2%in_basis(2)%alpha=ty%in_basis(2)%alpha+step

   if (loop <= 2 ) then

      !
      !
      !  Compute the transformation matix delta.  Since delta is only
      !  dependant on the distance between the bases it only needs to be
      !  computed twice, as the smaller basis changes are all in the
      !  same direction and are the same size.  The first for changing the
      !  basis size and the second for changing between bases of same size
      !
      !

      allocate(delta(0:ty%in_basis(2)%n,0:y2%in_basis(2)%n))

      call compute_delta(y2%in_basis(2),ty%in_basis(2))

      minn1=min(ty%in_basis(1)%n,y2%in_basis(1)%n)
      minn2=min(ty%in_basis(2)%n,y2%in_basis(2)%n)

   end if

   !
   !
   !  Change the basis
   !
   !

   y2%co_eff=cmplx(0)

   y2%co_eff(0:minn1,:) = mat_mul(ty%co_eff(0:minn1,:), conjg(delta))

   y2%co_eff = y2%co_eff*normalisation(y2)

   if (loop < 2) then

      deallocate(delta, ty%co_eff)

      ty%in_basis=y2%in_basis

      allocate(ty%co_eff(0:ty%in_basis(1)%n, 0:ty%in_basis(2)%n))

   end if


   ty%in_basis(2)%alpha=y2%in_basis(2)%alpha
   ty%co_eff=y2%co_eff

end do

if (allocated(delta)) then
```

```
        deallocate(delta)
      end if
      deallocate(ty%co_eff)
    end subroutine change_basis2

end module basis_change
```

A.11 Integration

```
module integrate_sde

  !
  !
  !  This module will provide the routines necessary to
  !  integrate the stochastic differential equation representing
  !  second harmonic generation
  !
  !

  use precision
  use globals
  use rand_num_gen
  use basis_change
  use operators

  implicit none

  !
  !
  !  These variables are used to define the system to integrate
  !
  !  k1, k2  are the damping of the cavity for each mode
  !  delta1, delta2  are the detunings of the cavity for each mode
  !  X  is the nonlineary coupling between the two modes
  !  f  is the amplitude of the external driving field
  !  tau_t  is the scaled time
  !
  !

  real(fdp), private :: k2, k1, delta1, delta2, X, f, tau_t


contains



  function gauss_rand_num() result(dW)

    !
    !
    !  This subroutine produces gaussian random numbers
    !  The random numbers have variance 1, and average 0
    !
    !  This routine was motivated by the routine given in Numerical Recipes
    !
    !  dW  is the returned complex gaussian random number
    !  tmp_ran1, tmp_ran2  are the two random numbers to define dW
    !
    !

    complex(cdp) :: dW
    real(fdp) :: tmp_ran1, tmp_ran2

    !
```

```
  !
  !  Compute random number dW
  !
  !

  tmp_ran1=random_num()
  tmp_ran2=random_num()

  dW=sqrt(-log(tmp_ran1))*exp(-im*2*pi*tmp_ran2)

end function gauss_rand_num



function integrate(initial, taui, tauf, nstep, params) result(y)

  !
  !
  !  This function carries out the integration of the function f,
  !  given its derivative, from ti to tf, with nstep number of integration
  !  steps
  !
  !  nstep  is the number of integration steps
  !  taui  is the starting integration time
  !  tauf  is the finishing integration time
  !  initial  is the initial state vector to integrate
  !  params  are the parameters guiding the integration
  !  eaf, eah  contain the trajectory of the state vector <y|a|y>
  !  y  is the resultant state vector of the integration
  !  det_component  is the deteministic component of the derivative
  !  stoc_component  is the stochastic component of the derivative
  !  save_1, save_2  are temporary state vectors used in converting a
  !                  state vector to a standard basis
  !  D_1, D_2  are the average reduced density operators for mode 1 and 2
  !  dt  is the integration time step
  !  sqrtdt  is the square root of dt used in the Wiener process
  !  ave  is a temporary variable to store co-efficient averages
  !  S  is the quantum/classical scaling variable
  !  tmp, tmp1, tmp2  are temporary real numbers
  !  loop  is used to loop over integers to generate the vector sq_rt
  !  n1, n2  are the size of bases for mode 1 and 2
  !  savewave  is how many steps to wait to save a state vector to disk
  !  snap  is the number of the state vector to be saved
  !  computeea  is the number of steps to wait to save a trajectory point
  !  computeD  is the number of steps to wait to calculate the average
  !            reduced density operators D_1 and D_2
  !  D_count is the number of reduced density operators averaged over
  !  i  is a loop variable
  !  savestring  is the name of the file to write to disk
  !  num  is the file number to save to disk
  !  realisation  is the trajectory number to be saved to disk
  !
  !

  integer(idp), intent(in) :: nstep
  real(fdp), intent(in) :: taui, tauf
```

```
type(wave_fn), intent(in) :: initial
type(parameters), intent(in) :: params
complex(cdp), dimension(params%ea_saves) :: eaf, eah
integer(idp), dimension(params%ea_saves,2) :: n
type(wave_fn) :: y, det_component, stoc_component, save_1, save_2
complex(cdp), dimension(0:max_basis, 0:max_basis) :: D_1, D_2
real(fdp) :: dt, sqrtdt, ave, S, tmp, tmp1, tmp2
integer(idp) :: loop, n1, n2, savewave, snap, computeea, computeD, &
     D_count, i
character*100 :: savestring
character*4 :: num, realisation


!
!
!  Setup the vector of square roots of integers
!
!

if (first_sqrt) then

   do loop=0,max_basis+3

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Setup some variables
!
!

D_1=cmplx(0)
D_2=cmplx(0)
D_count=0

S=params%S

realisation=achar(48+params%realisation/1000)// &
     achar(48+params%realisation/100-(params%realisation/1000)*10)// &
     achar(48+params%realisation/10-(params%realisation/100)*10)// &
     achar(48+params%realisation-(params%realisation/10)*10)

k2=0.1d0
k1=k2/params%K
delta1=params%d1*k1
delta2=params%d2*k1
X=k1*params%S
f=params%E*k1**2/X
tau_t=1/k1
```

```
dt=(tauf-taui)*tau_t/dble(nstep)
sqrtdt=sqrt(dt)

!
!
!  Initialise the integration state vector
!
!

y%in_basis=initial%in_basis

allocate(y%co_eff(0:y%in_basis(1)%n, 0:y%in_basis(2)%n))

y%co_eff=initial%co_eff

!
!
!  Compute savewave, computeea and computeD
!
!

savewave=1
computeea=1
computeD=1

if (params%save_waves) then

   savewave=nstep/params%wave_saves

end if

if (params%save_eas) then

   computeea=nstep/params%ea_saves

end if

if (params%compute_D) then

   computeD=nstep/params%D_computes

end if

!
!
!  The integration loop
!
!

do loop=1,nstep

   !
   !
   !  Compute the deterministic and stochastic derivatives
   !
   !
```

```
det_component = det_deriv(y)
stoc_component = stoc_deriv(y)

!
!
!  Get the size of the basis in each mode
!
!

n1=y%in_basis(1)%n
n2=y%in_basis(2)%n

!
!
!  The discrete integration step is computed here.  The method used is
!  the Euler-Cauchy one step method
!
!

y%co_eff=y%co_eff+dt*det_component%co_eff+sqrtdt*stoc_component%co_eff

!
!
!  Make sure that the state vector is normalised
!
!

y%co_eff = y%co_eff / normalisation(y)

deallocate(det_component%co_eff, stoc_component%co_eff)

!
!
!  Average over the last five basis states of mode 1 to determine
!  whether a basis change needs to be performed
!
!

tmp=1/dble(n2*6)
ave=sum(abs(y%co_eff(n1-5:n1,:)))*tmp

!
!
!  Change basis for mode 1 if needed
!
!

if (ave>cut_off1) then

   call move_basis1(y)

end if

!
!
```

```
!  Get the size of the bases of mode 1 and 2
!
!

n1=y%in_basis(1)%n
n2=y%in_basis(2)%n

!
!
!  Average over the last five basis states of mode 2 to determine
!  whether a basis change needs to be performed
!
!

tmp=1/dble(n1*6)
ave=sum(abs(y%co_eff(:,n2-5:n2)))*tmp

!
!
!  Change basis for mode 2 if needed
!
!

if (ave>cut_off1) then

   call move_basis2(y)

end if

!
!
!  Add another point to the trajectory <y|a|y>
!
!

if ((loop/computeea)*computeea == loop .and. params%save_eas) then

    eaf(loop/computeea)=S*(y.af.y)
    eah(loop/computeea)=S*(y.ah.y)
    n(loop/computeea,1)=y%in_basis(1)%n
    n(loop/computeea,2)=y%in_basis(2)%n

end if

!
!
!  Save the state vector to disk
!
!

if ((loop/savewave)*savewave == loop .and. params%save_waves) then

   snap=loop/savewave

   num=achar(48+snap/1000)//achar(48+snap/100-(snap/1000)*10)// &
        achar(48+snap/10-(snap/100)*10)//achar(48+snap-(snap/10)*10)
```

```
   savestring='data/wave_fn/wave_fn.'//realisation//num

   open(unit=1, file=savestring, form='unformatted')

   write(1) y%co_eff

   close(1)

   savestring='data/basis/basis.'//realisation//num

   open(unit=1, file=savestring, form='unformatted')

   write(1) y%in_basis

   close(1)

end if

!
!
!  Compute the average reduced density operator
!
!

if ((loop/computeD)*computeD == loop .and. params%compute_D) then

   D_count=D_count+1

   !
   !
   !  Change the state vector basis to a standard basis
   !
   !

   allocate(save_1%co_eff(0:max_basis,0:y%in_basis(2)%n))
   allocate(save_2%co_eff(0:max_basis,0:max_basis))

   save_1%in_basis(1)%n=max_basis
   save_1%in_basis(2)%n=y%in_basis(2)%n
   save_1%in_basis(1)%alpha=cmplx(0)
   save_1%in_basis(2)%alpha=y%in_basis(2)%alpha
   save_1%co_eff=cmplx(0)

   call change_basis1(y,save_1)

   save_2%in_basis(1)%n=max_basis
   save_2%in_basis(2)%n=max_basis
   save_2%in_basis(1)%alpha=cmplx(0)
   save_2%in_basis(2)%alpha=cmplx(0)
   save_2%co_eff=cmplx(0)

   call change_basis2(save_1,save_2)

   !
   !
```

```
      !  Compute the running average
      !
      !

      tmp=1/dble(D_count)

      D_1=D_1+tmp*(density1(save_2%co_eff)-D_1)
      D_2=D_2+tmp*(density2(save_2%co_eff)-D_2)

      !
      !
      !  Make sure that the density operator trace is correct
      !
      !

      tmp1=cmplx(0)
      tmp2=cmplx(0)

      do i=0,max_basis

         tmp1=tmp1+D_1(i,i)
         tmp2=tmp2+D_2(i,i)

      end do

      tmp1=1/tmp1
      tmp2=1/tmp2

      D_1=D_1*tmp1
      D_2=D_2*tmp2

      deallocate(save_1%co_eff, save_2%co_eff)

   end if

   !
   !
   !  End of the integration loop
   !
   !

end do

!
!
!  Save the trajectory <y|a|y> to disk
!
!

if (params%save_eas) then

   savestring='data/eaf.'//realisation

   open(unit=1, file=savestring)

   savestring='data/eah.'//realisation
```

```
      open(unit=2, file=savestring)

      savestring='data/n.'//realisation

      open(unit=3, file=savestring)

      write(1,'(f20.12,f20.12)') eaf
      write(2,'(f20.12,f20.12)') eah
      write(3,'(i4,i4)') n

      close(1)
      close(2)
      close(3)

    end if

    !
    !
    !  Save the average reduced density operators to disk
    !
    !

    if (params%compute_D) then

      savestring='vfl/data/D1.'//realisation

      open(unit=1, file=savestring, form='unformatted')

      write(1) D_1

      close(1)

      savestring='vfl/data/D2.'//realisation

      open(unit=1, file=savestring, form='unformatted')

      write(1) D_2

      close(1)

    end if

    !
    !
    !  End of integration
    !
    !

end function integrate



subroutine move_basis1(y)

    !
```

```
!
!  This routine shifts the basis of mode 1 of the state vector to the
!  value <y|a|y> and adjusts the size of the basis
!
!  y  is the state vector to be adjusted
!  tmpy1, tmpy2  are temporary state vectors
!  ave  is a temporary variable to store co-efficient averages
!  tmp  is a temporary real number
!  n1, n2  are the size of the bases for mode 1 and 2
!
!

type(wave_fn), intent(inout) :: y
type(wave_fn) :: tmpy1, tmpy2
real(fdp) :: ave, tmp
integer(idp) :: n1, n2


!
!
!  Setup tmpy1 and tmpy2
!  tmpy2 holds the initial state vector y
!
!

tmpy2%in_basis=y%in_basis

allocate(tmpy2%co_eff(0:tmpy2%in_basis(1)%n, 0:tmpy2%in_basis(2)%n))

tmpy2%co_eff=y%co_eff

tmpy1%in_basis=y%in_basis
tmpy1%in_basis(1)%alpha=y.af.y

allocate(tmpy1%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n))

tmpy1%co_eff=cmplx(0)

!
!
!  Change the basis of tmpy2 (y) to <y|a|y>
!  tmpy1 holds the shifted basis state vector
!
!

call change_basis1(tmpy2,tmpy1)

deallocate(y%co_eff)

!
!
!  Make y equal to the shifted basis state vector
!
!

y=tmpy1
```

```
y%co_eff=y%co_eff*normalisation(y)

!
!
!  Average over the last five basis states of mode 1 to determine
!  whether the basis size needs to be changed
!
!

n1=y%in_basis(1)%n
n2=y%in_basis(2)%n

tmp=1/dble(n2*6)
ave=sum(abs(y%co_eff(n1-5:n1,:)))*tmp

!
!
!  Reduce the size of the basis of mode 1 if it is to large
!
!

do while (ave < cut_off2 .and. n1 > 10)

   tmpy1%in_basis(1)%n=tmpy1%in_basis(1)%n-3

   allocate(tmpy1%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n))

   tmpy1%co_eff=y%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n)

   deallocate(y%co_eff)

   y=tmpy1

   y%co_eff=y%co_eff*normalisation(y)

   n1=y%in_basis(1)%n
   n2=y%in_basis(2)%n

   tmp=1/dble(n2*6)
   ave=sum(abs(y%co_eff(n1-5:n1,:)))*tmp

end do

!
!
!  Increase the size of the basis of mode 1 if it is to small
!
!  Note here that all basis changes are performed from the initial
!  state of the state vector y (tmpy2)
!
!  This was done to reduce the effect of compounding errors in changing
!  the basis
!
!
```

```
  do while(ave > cut_off1)

     tmpy1%in_basis(1)%n=tmpy1%in_basis(1)%n+(n1/10)+3

     allocate(tmpy1%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n))

     tmpy1%co_eff=cmplx(0)

     call change_basis1(tmpy2,tmpy1)

     deallocate(y%co_eff)

     y=tmpy1

     y%co_eff=y%co_eff*normalisation(y)

     n1=y%in_basis(1)%n
     n2=y%in_basis(2)%n

     tmp=1/dble(n2*6)
     ave=sum(abs(y%co_eff(n1-5:n1,:)))*tmp

  end do

  deallocate(tmpy2%co_eff)

  !
  !
  !  Make sure that the basis size is smaller than some maximum
  !
  !

  if (n1 > max_basis .or. n2 > max_basis) then

     write(*,*) 'Basis is larger than the allowed maximum'

  end if

end subroutine move_basis1



subroutine move_basis2(y)

  !
  !
  !  This routine shifts the basis of mode 2 of the state vector to the
  !  value <y|a|y> and adjusts the size of the basis
  !
  !  y  is the state vector to be adjusted
  !  tmpy1, tmpy2  are temporary state vectors
  !  ave  is a temporary variable to store co-efficient averages
  !  tmp  is a temporary real number
  !  n1, n2  are the size of the bases for mode 1 and 2
  !
  !
```

```
type(wave_fn), intent(inout) :: y
type(wave_fn) :: tmpy1, tmpy2
real(fdp) :: ave, tmp
integer(idp) :: n1, n2

!
!
!  Setup tmpy1 and tmpy2
!  tmpy2 holds the initial state vector y
!
!

tmpy2%in_basis=y%in_basis

allocate(tmpy2%co_eff(0:tmpy2%in_basis(1)%n, 0:tmpy2%in_basis(2)%n))

tmpy2%co_eff=y%co_eff

tmpy1%in_basis=y%in_basis
tmpy1%in_basis(2)%alpha=y.ah.y

allocate(tmpy1%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n))

tmpy1%co_eff=cmplx(0)

!
!
!  Change the basis of tmpy2 (y) to <y|a|y>
!  tmpy1 holds the shifted basis state vector
!
!

call change_basis2(tmpy2,tmpy1)

deallocate(y%co_eff)

!
!
!  Make y equal to the shifted basis state vector
!
!

y=tmpy1

y%co_eff=y%co_eff*normalisation(y)

!
!
!  Average over the last five basis states of mode 2 to determine
!  whether the basis size needs to be changed
!
!

n1=y%in_basis(1)%n
```

```
n2=y%in_basis(2)%n

tmp=1/dble(n1*6)
ave=sum(abs(y%co_eff(:,n2-5:n2)))*tmp

!
!
!  Reduce the size of the basis of mode 2 if it is to large
!
!

do while (ave < cut_off2 .and. n2 > 10)

   tmpy1%in_basis(2)%n=tmpy1%in_basis(2)%n-3

   allocate(tmpy1%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n))

   tmpy1%co_eff=y%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n)

   deallocate(y%co_eff)

   y=tmpy1

   y%co_eff=y%co_eff*normalisation(y)

   n1=y%in_basis(1)%n
   n2=y%in_basis(2)%n

   tmp=1/dble(n1*6)
   ave=sum(abs(y%co_eff(:,n2-5:n2)))*tmp

end do

!
!
!  Increase the size of the basis of mode 2 if it is to small
!
!  Note here that all basis changes are performed from the initial
!  state of the state vector y (tmpy2)
!
!  This was done to reduce the effect of compounding errors in changing
!  the basis
!
!

do while(ave > cut_off1)

   tmpy1%in_basis(2)%n=tmpy1%in_basis(2)%n+(n2/10)+3

   allocate(tmpy1%co_eff(0:tmpy1%in_basis(1)%n,0:tmpy1%in_basis(2)%n))

   tmpy1%co_eff=cmplx(0)

   call change_basis2(tmpy2,tmpy1)

   deallocate(y%co_eff)
```

```
      y=tmpy1

      y%co_eff=y%co_eff*normalisation(y)

      n1=y%in_basis(1)%n
      n2=y%in_basis(2)%n

      tmp=1/dble(n1*6)
      ave=sum(abs(y%co_eff(:,n2-5:n2)))*tmp

    end do

    deallocate(tmpy2%co_eff)

    !
    !
    !  Make sure that the basis size is smaller than some maximum
    !
    !

    if (n1 > max_basis .or. n2 > max_basis) then

      write(*,*) 'Basis is larger than the allowed maximum'

    end if

  end subroutine move_basis2



  function det_deriv(y) result(tmpy)

    !
    !
    !  This function returns the deterministic derivatives
    !
    !  y  is the state of the state vector when the derivative is wanted
    !  tmpy  is the deterministic derivative
    !  nfy  is the state vector n|y> for mode 1
    !  nhy  is the state vector n|y> for mode 2
    !  afy  is the state vector a|y> for mode 1
    !  ahy  is the state vector a|y> for mode 2
    !  adfy  is the state vector ad|y> for mode 1
    !  adhy  is the state vector ad|y> for mode 2
    !  adfadfahy  is the state vector ad( ad( a|y> mode 2) mode 1) mode 1
    !  afafadhy  is the state vector a( a( ad|y> mode 2) mode 1) mode 1
    !  eaf  is the expectation value <y|a|y> for mode 1
    !  eah  is the expectation value <y|a|y> for mode 2
    !  eadf  is the expectation value <y|ad|y> for mode 1
    !  eadh  is the expectation value <y|ad|y> for mode 2
    !
    !  To compute the quantities above, creation and annihilation operators
    !  have been defined.  This makes reading the code reasonably easy
    !
    !
```

```
type(wave_fn), intent(in) :: y
type(wave_fn) :: tmpy, nfy, nhy, afy, ahy, adfy, adhy, adfadfahy, afafadhy
complex(cdp) :: eaf, eah, eadf, eadh


!
!
!  Compute the various state vectors needed
!
!

afy=.af.y
ahy=.ah.y
adfy=.adf.y
adhy=.adh.y

nfy=.adf.afy
nhy=.adh.ahy

tmpy=.adf.ahy
adfadfahy=.adf.tmpy

deallocate(tmpy%co_eff)

tmpy=.af.adhy
afafadhy=.af.tmpy

deallocate(tmpy%co_eff)

!
!
!  Compute the various expectation values needed
!
!

eaf=y.af.y
eah=y.ah.y
eadf=y.adf.y
eadh=y.adh.y

tmpy%in_basis=y%in_basis

allocate(tmpy%co_eff(0:tmpy%in_basis(1)%n,0:tmpy%in_basis(2)%n))

!
!
!  Compute the deterministic derivative from all the components
!
!

tmpy%co_eff=-im*(delta1*nfy%co_eff + delta2*nhy%co_eff )&
     + f*(adfy%co_eff - afy%co_eff) &
     + X*0.5d0*(adfadfahy%co_eff - afafadhy%co_eff) &
     + 2*k1*eadf*afy%co_eff - k1*nfy%co_eff &
     - k1*eadf*eaf*y%co_eff &
```

```
       + 2*k2*eadh*ahy%co_eff - k2*nhy%co_eff &
       - k2*eadh*eah*y%co_eff

  deallocate(afy%co_eff, ahy%co_eff, adfy%co_eff, adhy%co_eff, nfy%co_eff, &
       nhy%co_eff, afafadhy%co_eff, adfadfahy%co_eff)

end function det_deriv



function stoc_deriv(y) result(tmpy)

  !
  !
  !  This function returns the deterministic derivatives
  !
  !  y  is the state of the state vector when the derivative is wanted
  !  tmpy  is the deterministic derivative
  !  afy  is the state vector a|y> for mode 1
  !  ahy  is the state vector a|y> for mode 2
  !  eaf  is the expectation value <y|a|y> for mode 1
  !  eah  is the expectation value <y|a|y> for mode 2
  !  sqrt_2k1, sqrt_2k2  are real parameters from the Lindblad operators
  !  rand1, rand2  are the gaussian random numbers for each mode
  !
  !  To compute the quantities above, creation and annihilation operators
  !  have been defined.  This makes reading the code reasonably easy
  !
  !


  type(wave_fn), intent(in) :: y
  type(wave_fn) :: tmpy, afy, ahy
  real(fdp) :: sqrt_2k1, sqrt_2k2
  complex(cdp) :: rand1, rand2, eaf, eah


  !
  !
  !  Compute the various state vectors needed
  !
  !

  afy=.af.y
  ahy=oper_a2(y)

  !
  !
  !  Compute the expectation values needed
  !
  !

  eaf=y.af.y
  eah=y.ah.y

  sqrt_2k1=sqrt(2*k1)
```

```
    sqrt_2k2=sqrt(2*k2)

    !
    !
    !  Get the gaussian random numbers
    !
    !

    rand1=gauss_rand_num()
    rand2=gauss_rand_num()

    tmpy%in_basis=y%in_basis

    allocate(tmpy%co_eff(0:tmpy%in_basis(1)%n,0:tmpy%in_basis(2)%n))

    !
    !
    !  Compute the stochastic derivative from all the components
    !
    !

    tmpy%co_eff=sqrt_2k1*rand1*(afy%co_eff - eaf*y%co_eff) &
         + sqrt_2k2*rand2*(ahy%co_eff - eah*y%co_eff)

    deallocate(afy%co_eff, ahy%co_eff)

  end function stoc_deriv



end module integrate_sde
```

A.12 Post Processing

```
module post_process

  !
  !
  !  This module contains the post-processing routines.
  !
  !

  use precision
  use globals
  use operators
  use basis_change

  implicit none


contains



  subroutine average_density(params, ensemble)

    !
    !
    !  This routine loads in the average reduced density operators from
    !  a number of trajectories, averages them and writes out the new average
    !
    !  params  is the parameters that control the integration
    !  ensemble  is the number of trajectories to average over
    !  i  is a loop variable
    !  D1, D2  are the density operators loaded off disk
    !  AD1, AD2  is the average density
    !  savestring  is the file name to load off disk
    !  tmp  is the temporary variable for averaging
    !  realisation  is the character representation of the trajectory number
    !
    !

    type(parameters), intent(in) :: params
    integer(idp), intent(in) :: ensemble
    integer(idp) :: i
    complex(cdp), dimension(0:max_basis, 0:max_basis) :: D1, D2, AD1, AD2
    character*100 :: savestring
    real(fdp) :: tmp
    character*4 :: realisation


    !
    !
    !  Reset AD1 and AD2
    !
    !

    AD1=cmplx(0)
    AD2=cmplx(0)
```

```
!
!
!  Setup tmp
!
!

tmp=1/dble(ensemble)

!
!
!  Loop over the number of trajectories
!
!

do i=1,ensemble

   !
   !
   !  Load the reduced density operators off disk
   !
   !

   realisation=achar(48+i/1000)// achar(48+i/100-(i/1000)*10)// &
        achar(48+i/10-(i/100)*10)// achar(48+i-(i/10)*10)

   savestring='vfl/data/D1.'//realisation

   open(unit=1, file=savestring, form='unformatted', action='read')

   read(1) D1

   close(1)

   savestring='vfl/data/D2.'//realisation

   open(unit=1, file=savestring, form='unformatted', action='read')

   read(1) D2

   close(1)

   !
   !
   !  Average the reduced density operators
   !
   !

   AD1=AD1+tmp*(D1)
   AD2=AD2+tmp*(D2)

end do

!
!
!  Save the average reduced density operators to disk
```

```
  !
  !

  open(unit=1, file='vfl/data/AD1')

  write(1,'(f20.12,f20.12)') AD1

  close(1)

  open(unit=1, file='vfl/data/AD2')

  write(1,'(f20.12,f20.12)') AD2

  close(1)

end subroutine average_density



subroutine q_function(rmin,rstep,rmax,imin,istep,imax, params)

  !
  !
  !  This subroutine will compute the Q function of the reduced density
  !  operators.
  !
  !  params  is the parameters that control the integration
  !  rmin  is the minimum real part of alpha
  !  rstep  is the step size for the real part of alpha
  !  rmax  is the maximum real part of alpha
  !  imin  is the minimum imaginary part of alpha
  !  istep  is the step size for the imaginary part of alpha
  !  imax  is the maximum imaginary part of alpha
  !  i,j  are loop variables
  !  ralpha, ialpha  loop over the components of alpha
  !  Q1, Q2  are the Q-functions for mode 1 and 2
  !  D1, D2  are the average reduced density operators loaded off disk
  !  alpha  is the value of alpha for calculating the Q-functions
  !  expalpha  is a variable dependant on alpha only
  !  fac  is a factor containing various factors
  !
  !

  type(parameters), intent(in) :: params
  real(fdp), intent(in) :: rmin,rstep,rmax,imin,istep,imax
  integer(idp) :: i,j, ralpha,ialpha
  complex(cdp), dimension(:,:), allocatable :: Q1, Q2
  complex(cdp), dimension(0:max_basis, 0:max_basis) :: D1, D2
  complex(cdp) :: alpha
  real(fdp) :: expalpha
  complex(cdp), dimension(0:max_basis) :: fac


  allocate(Q1(aint(rmin/rstep):aint(rmax/rstep),aint(imin/istep): &
       aint(imax/istep)), Q2(aint(rmin/rstep):aint(rmax/rstep), &
       aint(imin/istep):aint(imax/istep)))
```

```
!
!
!  Load in the average reduced density operators
!
!

open(unit=1, file='vfl/data/AD1', action='read')

read(1,'(f20.12,f20.12)') D1

close(1)

open(unit=1, file='vfl/data/AD2', action='read')

read(1,'(f20.12,f20.12)') D2

close(1)

!
!
!  Loop over the values of ralpha and ialpha
!
!

do ialpha=aint(imin/istep),aint(imax/istep)
   do ralpha=aint(rmin/rstep),aint(rmax/rstep)

      !
      !
      !  Compute alpha from the discrete variables ralpha and ialpha
      !
      !

      alpha=cmplx(ralpha*rstep,ialpha*istep)

      !
      !
      !  Compute the vector fac
      !
      !

      fac(0)=cmplx(1)

      do i=1,max_basis

         fac(i)=alpha/sq_rt(i)*fac(i-1)

      end do

      !
      !
      !  Compute expalpha and reset Q1 and Q2
      !
      !
```

```
        expalpha=exp(-abs(alpha)**2)

        Q1(ralpha,ialpha)=dble(0)
        Q2(ralpha,ialpha)=dble(0)

        !
        !
        !  Compute Q1 and Q2 for the particular value of alpha
        !
        !

        do j=0,max_basis
           do i=0,max_basis

              Q1(ralpha,ialpha)=Q1(ralpha,ialpha) + expalpha*D1(i,j)* &
                   conjg(fac(i))*fac(j)
              Q2(ralpha,ialpha)=Q2(ralpha,ialpha) + expalpha*D2(i,j)* &
                   conjg(fac(i))*fac(j)

           end do
        end do
     end do
  end do

  !
  !
  !  Save the Q-functions to disk
  !
  !

  open(unit=1, file='vfl/data/Q1', action='write')

  write(1,'(f20.12)') real(Q1)

  close(1)

  open(unit=1, file='vfl/data/Q2', action='write')

  write(1,'(f20.12)') real(Q2)

  close(1)

  deallocate(Q1, Q2)

end subroutine q_function



subroutine entropy_information(params, ensemble, dmin, dstep, dmax)

  !
  !
  !  This subroutine calculates the entropy and information of the
  !  state vectors that were saved to disk
  !
  !
```

```
type basis_name

   !
   !
   !  This derived type holds the basis and its character name
   !
   !  b  is the basis
   !  b_name  is the name of the basis saved on disk corresponding to b
   !
   !

   type(basis), dimension(2) :: b
   character*8 :: b_name

end type basis_name

type group

   !
   !
   !  This derived type defines a group
   !
   !  about_basis  is the basis that all state vectors in the group will
   !               be converted to
   !  members  is the name of the members in the group
   !  num_members  is the number of members in the group
   !
   !

   type(basis), dimension(2) :: about_basis
   integer(idp), dimension(:,:), pointer :: members
   integer(idp) :: num_members

end type group

!
!
!  params  is the parameters that control the integration
!  ensemble  is the number of trajectories in the run
!  dmin  is the minimum distance for forming groups
!  dstep  is the size that the distance should be increased
!  dmax  is the maximum distance for forming groups
!  i1, j1, i2, j2, i  are loop variables
!  in_group  is an array keeping track of which state vectors are grouped
!  loadstring  is the name of the file to load off disk
!  num  is the trajectory number to load
!  realisation  is the trajectory the state vector is from
!  tmp_basis  is a temporary basis
!  y  is the current state vector being considered
!  tmp_y1, tmp_y2  are temporary state vectors used for changing basis
!  A  is the matrix containing the state vectors in a group as its columns
!  total_ent  is the average entropy for a grouping scheme
!  total_inf  is the average information for a grouping scheme
!  group_prob  is the probability that a vector is in a group
!  d  is the distance for forming groups
```

```
!  basis_data  contains all the information abouts the bases for grouping
!  groups  contains a the information to define a group
!
!

type(parameters), intent(in) :: params
integer(idp), intent(in) :: ensemble
real(fdp), intent(in) :: dmin, dstep, dmax
integer(idp) :: i1, j1, i2, j2, minn, i
logical*1, dimension(ensemble, params%wave_saves) :: in_group
character*100 :: loadstring
character*4 :: num, realisation
type(basis), dimension(2) :: tmp_basis
type(wave_fn) :: y, tmp_y1, tmp_y2
complex(cdp), dimension(:,:), allocatable :: A
real(fdp) :: total_ent, total_inf, group_prob, d
type(basis_name), dimension(ensemble, params%wave_saves) :: basis_data
type(group) :: groups


d=dmin
allocate(groups%members(ensemble*params%wave_saves,2))

!
!
!  Loop over all bases
!
!

do j1=1,params%wave_saves
   do i1=1,ensemble

      !
      !
      !  Load in the bases and setup basis_data
      !
      !

      realisation=achar(48+i1/1000)//achar(48+i1/100-(i1/1000)*10)// &
           achar(48+i1/10-(i1/100)*10)//achar(48+i1-(i1/10)*10)

      num=achar(48+j1/1000)//achar(48+j1/100-(j1/1000)*10)// &
           achar(48+j1/10-(j1/100)*10)//achar(48+j1-(j1/10)*10)

      basis_data(i1,j1)%b_name=realisation//num

      loadstring='data/basis/basis.'//basis_data(i1,j1)%b_name

      open(unit=1, file=loadstring, form='unformatted', action='read')

      read(1) basis_data(i1,j1)%b

      close(1)

   end do
end do
```

```
!
!
!  Open the file to write the results to
!
!

open(unit=10, file='data/inf_ent.dat',action='write')

!
!
!  Loop over all distances for forming groups
!
!

do while(d <= dmax)

   !
   !
   !  Initialise variables
   !
   !

   in_group=.false.
   minn=ceiling((d+2.0d0)**2)
   total_ent=dble(0)
   total_inf=dble(0)

   !
   !
   !  Loop over all bases
   !
   !

   do i1=1,ensemble
      do j1=1,params%wave_saves

         !
         !
         !  Check to see whether the state vector is in a group
         !
         !

         if (.not. in_group(i1,j1)) then

            !
            !
            !  Reset the group data
            !
            !

            groups%members=0
            groups%about_basis=basis_data(i1,j1)%b
            groups%num_members=0
            groups%about_basis%n=minn
```

```
!
!
!  Loop over all other bases not alread in a group
!
!

do i2=i1,ensemble
   do j2=j1,params%wave_saves

      tmp_basis=basis_data(i2,j2)%b

      if (sqrt(sum(abs(tmp_basis%alpha-groups%about_basis% &
           alpha)**2)) <= d .and. .not. in_group(i2,j2)) then

         !
         !
         !  State vector i2, j2 now in  agroup
         !
         !

         in_group(i2,j2)=.true.

         groups%num_members=groups%num_members+1
         groups%members(groups%num_members,:)=(/ i2, j2 /)

         !
         !
         !  Make sure that the basis that all the groups will
         !  be converted to is large enough
         !
         !

         if (tmp_basis(1)%n > groups%about_basis(1)%n) then

            groups%about_basis(1)%n = tmp_basis(1)%n

         end if

         if (tmp_basis(2)%n > groups%about_basis(2)%n) then

            groups%about_basis(2)%n = tmp_basis(2)%n

         end if
      end if

   end do
end do

!
!
!  Loop over all members in a group and setup A
!
!

allocate(A((groups%about_basis(1)%n+1)*(groups%about_basis(2) &
     %n+1),groups%num_members))
```

```
do i=1,groups%num_members

   y%in_basis=basis_data(groups%members(i,1), groups%members( &
        i,2))%b

   allocate(y%co_eff(0:y%in_basis(1)%n, 0:y%in_basis(2)%n))

   !
   !
   !  Load in the state vector information
   !
   !

   loadstring='data/wave_fn/wave_fn.'//basis_data(groups% &
        members(i,1), groups%members(i,2))%b_name

   open(unit=1, file=loadstring, action='read', &
        form='unformatted')

   read(1) y%co_eff

   close(1)

   !
   !
   !  Change the all state vectors in a group to the same basis
   !
   !

   allocate(tmp_y1%co_eff(0:groups%about_basis(1)%n,0: &
        y%in_basis(2)%n), tmp_y2%co_eff(0:groups%about_basis &
        (1)%n,0:groups%about_basis(2)%n))

   tmp_y1%in_basis(1)=groups%about_basis(1)
   tmp_y1%in_basis(2)=y%in_basis(2)
   tmp_y1%co_eff=cmplx(0)

   call change_basis1(y,tmp_y1)

   tmp_y2%in_basis(1)=groups%about_basis(1)
   tmp_y2%in_basis(2)=groups%about_basis(2)
   tmp_y2%co_eff=cmplx(0)

   call change_basis2(tmp_y1,tmp_y2)

   !
   !
   !  Construct the matrix A of state vectors
   !
   !

   do j2=0,groups%about_basis(2)%n
      do i2=0,groups%about_basis(1)%n

         A(i2+j2*(groups%about_basis(1)%n+1)+1,i) = &
```

```
                        tmp_y2%co_eff(i2,j2)

                     end do
                  end do

                  deallocate(y%co_eff, tmp_y1%co_eff, tmp_y2%co_eff)

               end do

               !
               !
               !  Compute the running average entropy and information for
               !  the groups corresponding to a particular d
               !
               !

               group_prob=groups%num_members/dble(ensemble*params%wave_saves)

               total_ent=total_ent + group_prob*entropy(A,( &
                    groups%about_basis(1)%n+1)*(groups%about_basis(2)%n+1), &
                    groups%num_members)

               total_inf=total_inf - group_prob*log(group_prob)/log(2.0d0)

               deallocate(A)

            end if

         end do
      end do

      !
      !
      !  Write the average entropy and information to disk
      !
      !

      write(10,*) total_ent, total_inf

      d=d+dstep

   end do

   deallocate(groups%members)

   close(10)

 end subroutine entropy_information



end module post_process
```

A.13 Integrate Second Harmonic Generation

```
program shg_program

  !
  !
  !  This program will integrate a system, and perform the necessary
  !  post processing
  !
  !

  use integrate_sde
  use operators
  use rand_num_gen
  use globals
  use basis_change
  use post_process

  !
  !
  !  n  is the initial size of the bases
  !  y, y1  are state vectors for integration
  !  loop  is a loop variable
  !  ensemble  is the number of trajectories being integrated
  !  realisation  is the current trajectory
  !  r  is a complex temporary random number
  !  params  is the parameters that control the integration
  !  damp_period  is the damping period of the system
  !
  !

  integer(idp) :: n=8
  type(wave_fn) :: y, y1
  integer(idp) :: loop, ensemble, realisation
  complex(cdp) :: r
  type(parameters) :: params
  real(fdp) :: damp_period


  !
  !
  !  Loading what job number this run is
  !
  !

  open(unit=1, file='job_no', action='READ')

  read(1,*) realisation,ensemble

  close(1)

  !
  !
  !  Setup the initial random numebr seed
  !
  !
```

```
if (realisation==1) then

   r=random_num('r')
   r=random_num('s')

end if

!
!
!  Setup the square root of integers
!
!

if (first_sqrt) then

   sq_rt=dble(0)

   do loop=0,max_basis+3

      sq_rt(loop)=sqrt(dble(loop))

   end do

   first_sqrt=.false.

end if

!
!
!  Load in the random number seed
!
!

r=random_num('l')

!
!
!  If this is the first run, setup the initial state vector
!  If it is not the initial run, the saved wave fucntion is loaded
!
!

if (realisation==1) then

   y%in_basis(1)%n=n
   y%in_basis(1)%alpha=cmplx(2.1d0,0.0d0)
   y%in_basis(2)%n=n
   y%in_basis(2)%alpha=cmplx(1.0d0,0.0d0)

   allocate(y%co_eff(0:n,0:n))

   y%co_eff=cmplx(0.0d0,0.0d0)
   y%co_eff(0,0)=cmplx(1.0d0)

   y%co_eff = y%co_eff*normalisation(y)
```

```
else

   open(unit=1, file='data/tmp_basis',action='read',form='unformatted')

   read(1) y%in_basis

   close(1)

   allocate(y%co_eff(0:y%in_basis(1)%n,0:y%in_basis(2)%n))

   open(unit=1, file='data/tmp_wave',action='read',form='unformatted')

   read(1) y%co_eff

   close(1)

end if

!
!
!  The parameters for the run are setup
!
!

params%realisation=realisation
params%save_waves=.false.
params%wave_saves=10
params%save_eas=.false.
params%ea_saves=100000
params%compute_D=.false.
params%D_computes=50
params%S=1.25
params%d1=-1.0d0
params%d2=-1.0d0
params%K=0.25d0
params%E=3.1d1

!
!
!  The damping period is set
!
!

damp_period=1/params%K

!
!  If this is the initial run, the initial transients are removed
!

if (realisation==1) then

   y1=integrate(y,0.0d0,3.0d0,300000,params)

   deallocate(y%co_eff)

   y=y1
```

```
end if

!
!
!  The damping period is integrated for to make sure that the state is
!  not correlated to its past.
!
!

y1=integrate(y,0.0d0,damp_period,500000, params)

deallocate(y%co_eff)

!
!
!  This is where the various data files are choosen
!
!

params%compute_D=.true.
params%save_waves=.true.
params%save_eas=.true.

!
!
!  This is where the integration routine is called
!
!

y=integrate(y1,0.0d0,4.0d0,500000, params)

!
!
!  This is where the state vector final state vectors is stored
!  to allow it to be loaded at the beginning of the next run
!
!

open(unit=1, file='data/tmp_basis',action='write',form='unformatted')

write(1) y%in_basis

close(1)

open(unit=1, file='data/tmp_wave',action='write',form='unformatted')

write(1) y%co_eff

close(1)

deallocate(y%co_eff, y1%co_eff)

!
!
!  The randomseed is saved
```

```
  !
  !

  r=random_num('s')

  if (realisation==ensemble) then

     !
     !
     !  These are the post processing options
     !
     !  average_density will average the density operators obtained in a
     !  multiple trajectory run
     !
     !  q_function will compute the q-function of the average reduced density
     !  operators
     !
     !  entropy_information will compute the entropy and information for
     !  for different non-overlapping groupings of the state vectors
     !
     !

     call average_density(params, ensemble)
     call q_function(-2.0d1,0.4d0,2.0d1,-2.0d1,0.4d0,2.0d1, params)
     call entropy_information(params, ensemble, 0.0d0, 0.2d0, 1.0d1)

  end if

end program shg_program
```

Bibliography

[Brun 1996] T A Brun, N Gisin, P F O'Mahony and M Rigo, *From quantum trajectores to classical orbits*, Preprint

[Carmichael 1993] H Carmichael, Lecture Notes in Physics *An Open Systems Approach to Quantum Optics* (Springer-Verlag 1993)

[de Oliveira 1990] F A M de Oliveira, M S Kim, and P L Knight, *Properties of displaced number states*, Phys. Rev. A **41**, 2645 (1990)

[Ford 1992] J Ford and G Mantica, *Does quantum mechanics obey the correspondence principle? Is it complete?*, Am. J. Phys. **60** (12), 1086 (1992)

[Fox 1990] R Fox, *Chaos, molecular fluctuations, and the correspondence limit*, Phys. Rev. A **41**, 2969 (1990)

[Gardiner 1985] C W Gardiner, *Handbook of Stochastic Methods*, (Springer-Verlag 1985)

[Goldstein 1965] Herbert Goldstein, *Classical Mechanics* (Addison-Wesley 1965)

[Haake 1991] Fritz Haake, *Quantum Signatures of Chaos* (Springer-Verlag 1991)

[Haken 1983] H Haken, *Synergetics* (Springer-Verlag 1983)

[Kloeden 1992] P Kloeden and E Platen, *Numerical Solution of Stochastic Differential Equations* (Springer 1992)

[Kloeden 1994] P Kloeden, E Platen and H Schurs, *Numerical Solution of SDE Through Computer Experiments* (Springer-Verlag 1994)

[Meystre 1990] P Meystre and M Sargent III, *Elements of Quantum Optics* (Springer-Verlag 1990)

[Press 1995] W Press, S Teukolsky, W Vettering, B Flannery and M Metcalf, *Numerical Recipes in Fortran* (Cambridge University Press 1995) Second Edition

[Savage 1983] C M Savage and D F Walls, *Optical chaos in second-harmonic generation*, Optica Acta **40**, 557 (1983)

[Schack 1995] R Schack, T Brunn and I percival, *Quantum state diffusion, localization and computation*, J. Phys. A **28**, 5401 (1995)

[Schack 1996 a] R Schack and C Caves, *Information-theoretic characterization of quantum chaos*, Preprint

[Schack 1996 b] R Schack and T Brunn, *A C++ library using quantum trajectories to solve quantum master equations*, Preprint

[Schuster 1988] Heinz Georg Schuster, *Deterministic Chaos* (VCH Verlagsgsesellschaft 1988)

[Walls 1995] D F Walls and G J Milburn, *Quantum Optics* (Springer 1995)

[Wiseman 1994] Howard Mark Wiseman, *Quantum Trajectories and Feedback*, PhD Thesis, University of Queensland, November 1994

[Zheng 1995] Xiping Zheng and C M Savage, *Quantum trajectories and classical attractors in second-harmonic generation*, Phys. Rev. A **51**, 792 (1995)