Numerical Modelling for Microwave-Optical Transduction and Photon Pair Generation using Atomic Ensembles

Maria Nicolae

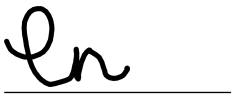
May 17, 2024

Abstract

Quantum networking, the transfer of quantum information across long distances, has great promise for scaling and interoperating quantum technologies, to allow us to solve problems that would not be possible or feasible classically. Many of the quantum systems that would form the nodes of this network have microwave energy scales, but the most feasible long-distance interconnects are fibre optics transmitting single optical photons. Thus, a means of correlating quantum information between microwave and optical systems is a near-requirement of quantum networking. This requires a hybrid microwave-optical quantum system, which can be used for quantum *transduction*, direct conversion of microwave and optical photons, and microwave-optical entangled photon pair generation. In this project, I develop numerical models with which to characterise transduction efficiencies and photon pair generation rates in hybrid systems that use ensembles of atoms with both microwave and optical transitions.

Statement of Originality

I certify that this thesis contains work carried out by myself except where otherwise acknowledged.



Maria Nicolae
2024-05-17

Acknowledgements

First of all, I would like to acknowledge my supervisor, Dr John Bartholomew. I thank him both for guiding and directing me throughout this project and for letting me occasionally deviate from that guidance and direction. I thank him for his patience while I was onboarding and learning the background for this project. I thank him for his hard working reviewing and giving feedback on drafts of this report as well as earlier Honours documents such as the talk and research plan. I also thank him for giving me the opportunities to attend the 2023 EQUS Annual Workshop and 2024 Quantum Australia conferences.

Next, I would like to thank Professor Andrew Doherty for taking time out of his day to help me understand quantum input-output theory, as well as for helping me reason about unitary transformations of Rabi Hamiltonians. I would also like to thank Gargi Tyagi for our discussions on input-output theory.

I would like to thank Dr Sahand Mahmoodian for our discussions about my biphoton generation modelling, and helping to clarifying some subtle points about the underlying physics, as well as future possibilities for the model.

I would like to thank Ben Field and John for giving me spin Hamiltonian code that helped me understand the system so that I could write my own minimal implementation of the spin Hamiltonian of ytterbium.

Finally, I would like to thank Gargi and Alice Jeffery for reading drafts of this report and giving me feedback, and Alice, Tim Newman, Ben, Gargi, John, and Elizabeth Marcellina for giving me feedback on a practice talk.

Statement of Contribution of Student

I programmed my own implementations of the three-level transduction models in References [1] and [2], and used these to replicate some of the plots in those references. After noticing discrepancies between and inconsistencies within those papers regarding phase conventions, I ran one of my model implementations to evaluate input and output phases, finding that only one convention produced physically sensible results.

I developed and implemented in code a model for four-level transduction, building on top of the concepts in the existing three-level transduction models. I implemented numerical methods to mitigate grid aliasing that were based on and built on top of methods in Reference [2]. I then benchmarked this model against results from experiments described in Reference [3], finding the model parameters corresponding to those experiments, using a mixture of theory and trial-and-error manual adjustments. My supervisor also helped refine those parameters.

I developed and implemented both steady-state and dynamical models for biphoton generation in three-level atomic systems, adapting the three-level transduction model in Reference [2] by changing indices of input and output atomic transitions, and modifying the atomic dynamics part of the model to accommodate vacuum interactions that start the generation processes into empty cavities.

Contents

Chapter 1

Introduction

1.1 Quantum Networking

There are a diverse range of quantum technologies presently being researched and developed. These include quantum computing[4], quantum simulation[5], and quantum sensing and metrology[6]. Quantum computing is the algorithmic processing and transformation of data encoded in the joint state space (the *Hilbert space*) of multiple two-level systems (*qubits*), which can offer an exponential speed advantage over classical computers on certain algorithms. Quantum simulation uses an engineered, controllable quantum system, known as a quantum simulator, to implement a Hamiltonian analogous to that of some natural or less controlled quantum system, in order to study its behaviour. Quantum sensing and metrology uses the great sensitivity of quantum systems to their environments to make measurements more precise than classical equipment can.

These systems have distinct applications from each other, but are not inherently interoperable. Additionally, all of these systems have proven challenging to scale up. *Quantum networking* would alleviate both of these issues by allowing these systems to communicate quantum states between each other, rather than mere measurement outcomes as in classical networking. Quantum networking would greatly expand the joint Hilbert spaces of quantum computers and simulators, allowing them to solve larger problems, and allow them to interoperate with quantum sensors. It would also enable the use of quantum cryptography[7, 8] in large, complex networks, which could enable eavesdropping to be detected, through a process analogous to the observer effect. Finally, quantum networking would enable fundamental experiments of quantum physics, such as tests of Bell's inequality, at greater scales than previously possible[9].

1.2 Correlating Microwave and Optical Photons

Many quantum technology platforms, such as superconducting circuits and trapped ions, have energy levels separated by microwave transition frequencies, meaning that they absorb and emit microwave photons. Directly transmitting single microwave photons between quantum devices, however, is impractical. This is because bulky and expensive cooling infrastructure is needed across the entire length of the link, to mitigate thermal microwave noise and loss.

On the other hand, optical photons can much more easily be transmitted across multi-kilometre distances using optical fibres (and further if quantum repeaters[10] are used). Thermal noise is negligible for optical frequencies, even at room temperature, so no cooling is necessary. In order to use optical photons to network microwave-energy quantum systems, we would need to be able to entangle the quantum information in microwave and optical photons. This would require the use of hybrid systems that contain both optical and microwave degrees of freedom. One way to use such a hybrid system for quantum networking is *transduction*, in which a transducer directly converts microwave and optical photons by absorbing one type and emitting the other type, and vice versa. Another approach is entangled microwave-optical photon pair generation (henceforth called *biphoton generation* for short).

Quantum networking using these processes is illustrated in Figure 1.1. To entangle two distant

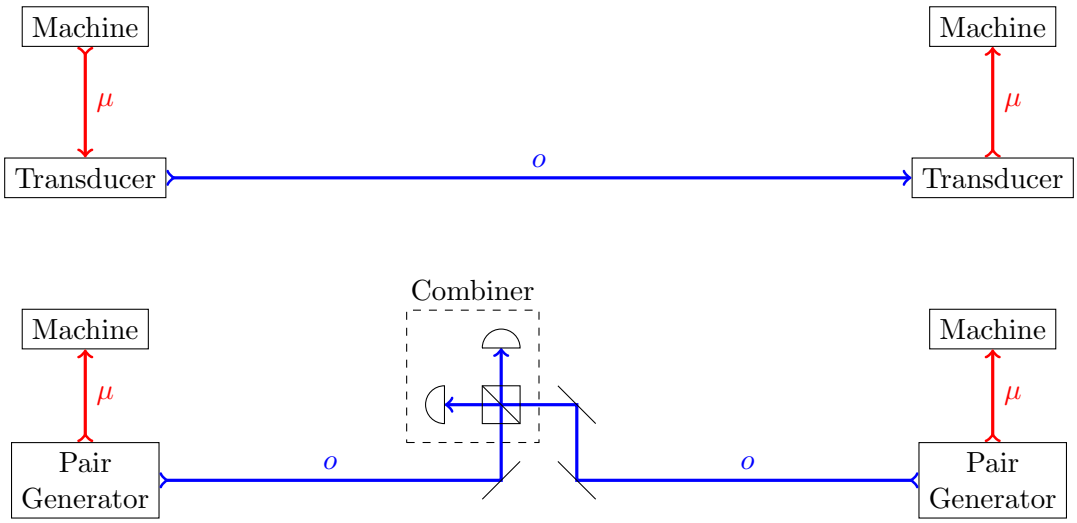


Figure 1.1: Quantum networking of two distant machines using transduction (top) and biphoton generation (bottom). Measurements of the combined optical signals entangle the two machines.

quantum machines, one approach would be for a microwave photon to be emitted from one machine, transduced to an optical photon, and then transduced back to a microwave photon at the other end, to interact with the other machine. Another approach, using biphoton generation, would be to generate entangled pairs on either end, interact the microwave photons with the machines, and interfere and measure the optical photons at some common destination.

1.3 Wave Mixing Processes

In order for microwave and optical photons to interact through some mediating system, that system must have some nonlinearity through which *wave-mixing* processes, in which frequencies mix to produce new frequencies, can occur. The simplest of these processes are second-order *three-wave mixing* processes, in which three frequencies are involved. These processes include *sum frequency generation* (SFG), in which two input frequencies ω_1 and ω_2 mix to produce a third output frequency $\omega_3 = \omega_1 + \omega_2$, *difference frequency generation* (DFG), in which two input frequencies mix to produce an output frequency $\omega_3 = |\omega_1 - \omega_2|$, and *spontaneous parametric downconversion* (SPDC), in which a single input frequency ω_1 produces two output frequencies ω_2 and ω_3 for which $\omega_2 + \omega_3 = \omega_1$.

Transduction can be performed through the mixing of microwave frequencies ω_μ and optical frequencies ω_o with an optical pump ω_p; SFG $\omega_p + \omega_\mu = \omega_o$ for microwave to optical transduction and DFG $|\omega_p - \omega_o| = \omega_\mu$ for optical to microwave transduction. An ω_μ and ω_o photon pair can be generated through SPDC $\omega_p = \omega_o + \omega_\mu$. (Note that $\omega_p < \omega_o$ for transduction but $\omega_p > \omega_o$ for biphoton generation.)

1.4 Hybrid Microwave-Optical Quantum Systems

Many different hybrid systems and processes have been proposed and experimentally studied for use in transduction and biphoton generation. One example of such systems are dielectric media with secord-order nonlinear polarisabilities [11, 12, 13] ($\chi^{(2)} \neq 0$). In these media, incident electromagnetic waves create time-varying polarisation in the material at new frequencies that are then emitted, giving rise to three-wave mixing. Another example is optomechanical systems[14, 15], in which a mechanical resonator simultaneously constitutes a mirror in an optical resonant cavity and a capacitor in a microwave resonator. This results in coupling between the optical, mechanical, and microwave modes. Yet another platform is atoms with both microwave and optical transitions between their energy levels[16, 17], in which both types of photons can interact with atomic transitions. Reviews of the various systems can be found at References [18, 19]. My project focuses on atomic systems in particular.

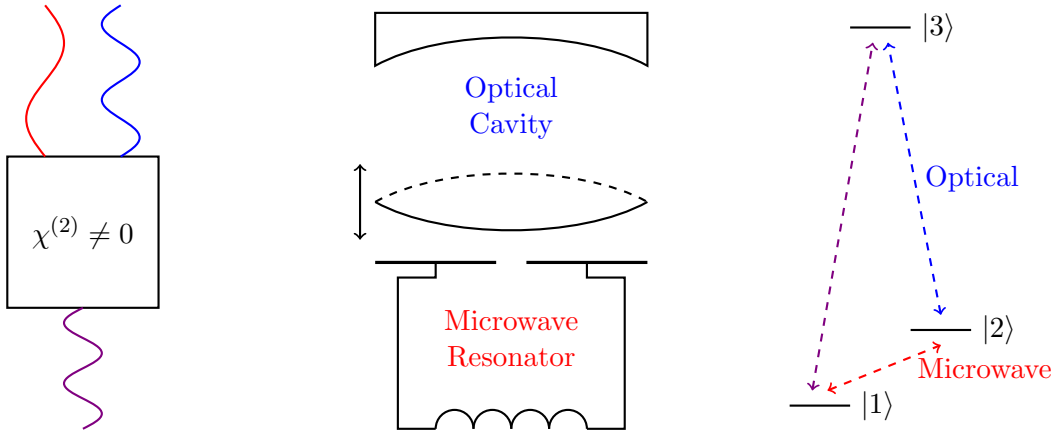


Figure 1.2: Hybrid microwave-optical platforms illustrated, including $\chi^{(2)}$-nonlinear dielectric media (left), optomechanical systems (middle), and atomic energy levels (right).

1.5 Hybrid Atomic Systems

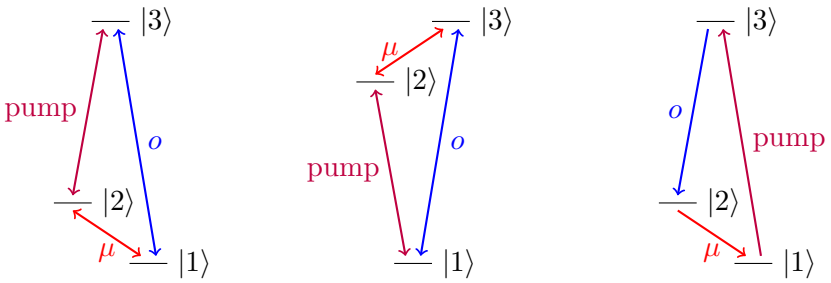


Figure 1.3: Transduction in a Λ-system (left) and V-system (middle), and biphoton generation in a Λ-system (right).

Atoms can be used for transduction by working in a three-level system consisting of two levels separated by a microwave transition and a third level separated from the other two by optical transitions. This can either be a *Λ-system*, in which the $|1\rangle$ and $|2\rangle$ levels are microwave-separated, or a *V-system*, in which $|2\rangle$ and $|3\rangle$ are microwave-separated[1]. To perform transduction in a Λ-system, $|2\rangle \leftrightarrow |3\rangle$ is pumped so that absorption of a microwave photon by the $|1\rangle \rightarrow |2\rangle$ transition results in coherence between the $|1\rangle$ and $|3\rangle$ levels and ultimately the emission of an optical photon from the $|3\rangle \rightarrow |1\rangle$ transition, and vice-versa. In a V-system, $|1\rangle \rightarrow |2\rangle$ is pumped, microwave signals interact with the $|2\rangle \leftrightarrow |3\rangle$ transition, and optical signals interact with $|1\rangle \leftrightarrow |3\rangle$. Biphoton generation can be performed by pumping $|1\rangle \rightarrow |3\rangle$ to obtain continuous output of $|3\rangle \rightarrow |2\rangle$ and $|2\rangle \rightarrow |1\rangle$ photon pairs. These processes are illustrated in Figure 1.3.

Atoms can be used in quantum technology as electromagnetically trapped ions or neutral atoms, or as constituents of crystals, with the latter being the focus of my project. Atoms in crystals are more miniaturisable and therefore scalable than trapped atoms because they do not need trapping infrastructure; confinement is provided by the crystal. Scaling is desirable because the more atoms are used, the stronger their collective interactions with the microwave and optical signals are. However, atomic ensembles in crystals are subject to *inhomogeneous broadening* of the collective spectral line shapes, compared to individual atoms.

Let the absorption spectrum of a single atom around its transition frequency ω_0 (the *homogeneous* lineshape) be $\alpha_S(\omega - \omega_0)$. The absorption spectrum of N atoms of the same species, if all atoms had the same transition frequency, would simply be

$$\alpha_E(\omega) = N\alpha_S(\omega - \omega_0). \tag{1.1}$$

[1] Λ-systems and V-systems are named as such because the two optical transitions form diagrams resembling those glyphs.

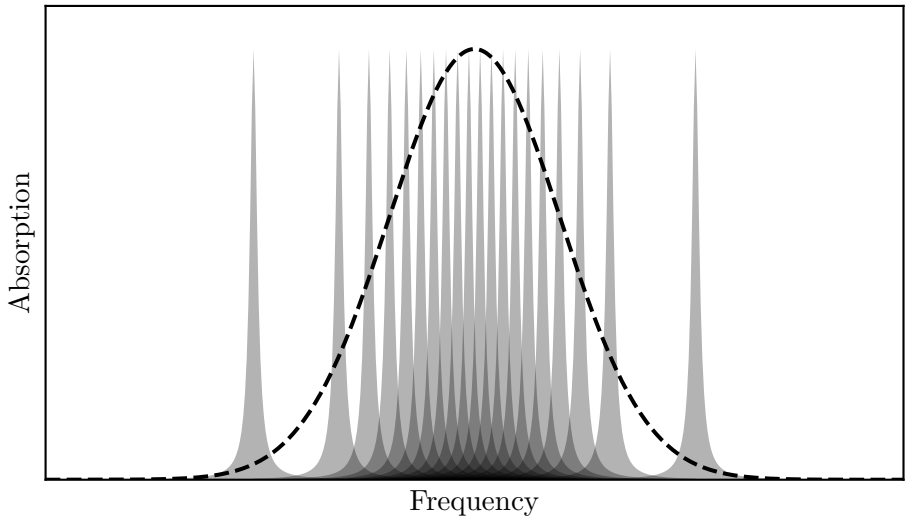


Figure 1.4: Inhomogeneous broadening of an atomic ensemble's absorption spectrum. The individual atoms have absorption spectra (grey filled curves) that are slightly shifted from each other, resulting in an overall ensemble absoption spectrum (dashed line) that is broader than the individual atom spectrum.

However, in a crystal, every atom has its own slightly different transition frequency ω_0. This is because each atom has a slightly different local electromagnetic field due to strain and defects in the crystal, which shift the atomic energy levels via effects such as the Stark and Zeeman effects, by a different amount for each atom. Given that the atomic transition frequencies are distributed with probability density function (PDF) $p(\omega_0)$, the absorption spectrum of an N-atom ensemble (the *inhomogeneous* lineshape) is

$$\alpha_E(\omega) = N \int_0^\infty \alpha_S(\omega - \omega_0) p(\omega_0)\, d\omega_0 = N \alpha_S * p. \tag{1.2}$$

This is at least as wide as the homogeneous line width, and is usually much wider.

1.5.1 Rare Earths

Of all atomic species one could use for hybrid systems, rare earths are a leading candidate. Their states have long coherence times, such as nuclear spin coherence times longer than 1 s in Er^{3+}[20] and up to 6 hours in Eu^{3+}[21], and electronic coherence times of 4 ms in Er^{3+}[22]. Rare earths also have narrow inhomogeneous linewidths[23], MHz for microwave transitions and hundreds of MHz for optical transitions. Both of these properties are the result of the full $5s$ and $5p$ electron shells of rare earths having larger radii than their $4f$ valence shells, which shields the latter from the external environment[24]. Erbium in particular has an optical transition frequency in the infrared telecommunications band which is attenuated least by optical fibres, making it well-suited for long-distance communication and networking applications.

1.6 Background Quantum Theory

This section explains the quantum-mechanical formalisms that are used throughout the remainder of the thesis. This begins with *second quantisation*, the quantum description of light and systems that interact with light (*quantum emitters*), and then describes those interactions as exchanges of energy quanta, and a semiclassical approximation thereof. Then, the theory of inputs and outputs of quantum systems is presented. Finally, a formalism for quantum decoherence is presented; due to this being a stochastic phenomenon, this formalism is expressed in terms of probabilistic mixtures of quantum states.

1.6.1 Second Quantisation

In the formalism of *second quantisation*, quantum systems are analysed as being composed of components (*modes*) which are occupied by energy quanta. These modes have Hamiltonians of the form

$$\hbar\omega_0\hat{n} \tag{1.3}$$

where

$$\hat{n} = \sum_n n\,|n\rangle\langle n| \tag{1.4}$$

is the observable for the number of energy quanta in the mode, and $\hbar\omega_0$ is the energy quantum.

For a *bosonic* mode, the sum in Equation 1.4 is over $n = 0, 1, 2, \ldots$, i.e. an arbitrarily large number of quanta can occupy the mode. An electromagnetic cavity mode is a bosonic mode, with the energy quanta being photons. For a *fermionic* mode, only one quantum can occupy it, and $n = 0, 1$ only. The fermionic $|0\rangle$ and $|1\rangle$ states are sometimes alternatively denoted $|g\rangle$ (*ground*) and $|e\rangle$ (*excited*) respectively. A two-level quantum emitter is a fermionic mode, and the level pairs of multi-level systems like atoms can be modelled as fermionic modes.

Ladder Operators

The *number operator*

$$\hat{n} = \hat{c}^\dagger\hat{c} \tag{1.5}$$

is composed of a *lowering operator* $\hat{c}$ and a *raising* operator[2] $\hat{c}^\dagger$. These ladder operators act on number states by lowering or raising them respectively to adjacent number states. Specifically, the bosonic ladder operators act on number states by

$$\hat{a}\,|n\rangle = \sqrt{n}\,|n-1\rangle\,, \quad \hat{a}^\dagger\,|n\rangle = \sqrt{n+1}\,|n+1\rangle\,, \tag{1.6}$$

and the fermionic ladder operators are

$$\hat{\sigma} = |0\rangle\langle 1|\,, \quad \hat{\sigma}^\dagger = |1\rangle\langle 0|\,; \tag{1.7}$$

$\hat{c}$ in Equation 1.5 is any one of $\hat{a}$ or $\hat{\sigma}$. These operators have commutation relations[3]

$$[\hat{a}, \hat{a}^\dagger] = \hat{\mathbb{1}}, \quad [\hat{\sigma}, \hat{\sigma}^\dagger] = \hat{\mathbb{1}} - 2\hat{\sigma}^\dagger\hat{\sigma}. \tag{1.8}$$

As the notation suggests, ladder operators are Hermitian conjugates of each other, and the lowering operator, by convention, is the one represented without a dagger.

1.6.2 Light-Matter Interactions

Quantum Model

In the language of second quantisation, light-matter interactions are described in terms of an electromagnetic cavity with lowering operator $\hat{a}$ and a two-level quantum emitter with lowering operator $\hat{\sigma}$. Here I consider light-matter interactions through the dipole interaction, which, for the example of an electric dipole, is represented by a Hamiltonian

$$\hat{H} = \hat{H}_{\text{light}} + \hat{H}_{\text{emitter}} - \hat{\mathbf{d}}\cdot\hat{\mathbf{E}}. \tag{1.9}$$

$\hat{\mathbf{d}}$ is the dipole moment operator of the emitter, and can therefore be expressed in terms of $\hat{\sigma}$, and $\hat{\mathbf{E}}$ is the electric field operator, which can be expressed in terms of $\hat{a}$. Making appropriate approximations[4]

[2] Alternatively, *annihilation* and *creation* operator respectively

[3] Here, unlike in Equation 1.3, $\hbar = 1$ is used and $\hbar$ is dropped accordingly. The same will be done in the remainder of this thesis.

[4] The *rotating wave approximation*

and writing the result out in terms of mode operators yields the *Jaynes-Cummings Hamiltonian*[25, 26]

$$\hat{H}_{\mathrm{JC}} = \omega_r \hat{a}^\dagger \hat{a} + \omega_a \hat{\sigma}^\dagger \hat{\sigma} + g(\hat{a}\hat{\sigma}^\dagger + \hat{a}^\dagger \hat{\sigma}). \tag{1.10}$$

Here, ω_r is the resonant frequency of the cavity, ω_a is the transition frequency of the emitter, and g is a constant representing the strength of the interaction. The terms $\hat{a}\hat{\sigma}^\dagger$ and $\hat{a}^\dagger\hat{\sigma}$ that are scaled by g represent the interaction itself as the transfer of energy quanta between the cavity and the emitter. Because this interaction is through the dipole mechanism, the interaction strength is proportional to

$$g \propto \langle g | \, \hat{d}_{\parallel} \, | e \rangle \tag{1.11}$$

the component of the dipole moment matrix element parallel to the light polarisation.

Semiclassical Approximation and Rabi Frequencies

A unitary transformation of Equation 1.10 eliminates the cavity energy term to obtain

$$\hat{H} = \omega_a \hat{\sigma}^\dagger \hat{\sigma} + g\hat{a}e^{-i\omega_r t}\hat{\sigma}^\dagger + g\hat{a}^\dagger e^{i\omega_r t}\hat{\sigma}. \tag{1.12}$$

To form a semiclassical approximation, the cavity operator $\hat{a}$ is replaced with a complex number α that represents the amplitude and phase of a 'classical-like' cavity state[5], scaled so that $|\alpha|^2 = \langle \hat{n} \rangle$ resulting in the semiclassical *mean-field* model in [27], a Hamiltonian which, in the $(|g\rangle, |e\rangle)$ basis of the emitter, is

$$\hat{H} = \begin{bmatrix} 0 & g\alpha^* e^{i\omega_r t} \\ g\alpha e^{-i\omega_r t} & \omega_a \end{bmatrix}. \tag{1.13}$$

This represents the dipole interaction between a quantum emitter and a classical oscillating electromagnetic field in terms of the *Rabi frequency* $\Omega = g\alpha$. More specifically, for the example of an electric dipole,

$$\Omega = \frac{\langle g | \, \hat{\mathbf{d}} \, | e \rangle \cdot \boldsymbol{\mathcal{E}}_0}{\hbar} \tag{1.14}$$

where $\boldsymbol{\mathcal{E}}_0$ is the complex amplitude of the electric field. This model therefore also applies to emitters driven by waveguides or free-space light beams, for some Ω that has no interpretation as a $g\alpha$. A unitary transformation of Equation 1.13 gives a time-independent Hamiltonian

$$\hat{H} = \begin{bmatrix} 0 & \Omega^* \\ \Omega & \omega_a - \omega_r \end{bmatrix}. \tag{1.15}$$

Furthermore, Equation 1.13 can be extended quite simply to driven multi-level systems: for energy levels indexed by k and drives indexed by ℓ,

$$\hat{H} = \sum_k \omega_k \hat{\sigma}_{kk} + \sum_\ell \left(\Omega_\ell e^{i\omega_\ell t} \hat{\sigma}_{i_\ell j_\ell} + \Omega_\ell^* e^{-i\omega_\ell t} \hat{\sigma}_{j_\ell i_\ell} \right) \tag{1.16}$$

where $\hat{\sigma}_{ij} = |i\rangle \langle j|$ are unit matrices and $i_\ell j_\ell$ are the transitions driven by drive ℓ.

1.6.3 The Heisenberg Picture

The *Heisenberg picture* of quantum mechanics is a formulation of quantum mechanics in which operators evolve in time, but state vectors (kets) are static, representing initial conditions. This is in opposition to the *Schrödinger picture*. Operators in the Heisenberg picture obey the *Heisenberg equation*[28]

$$\frac{d\hat{A}}{dt} = -i[\hat{A}, \hat{H}]. \tag{1.17}$$

When forming semiclassical approximations that replace light operators with amplitudes, such as in Subsection 1.6.2, the Heisenberg equation of a light operator becomes the differential equation of the amplitude.

[5] Known as a *coherent state*; see Reference [26].

Langevin Equations and Input-Output Theory

If we have some system, with Hamiltonian $\hat{H}_{\text{sys}}$ that is coupled through an operator $\hat{c}$ to a waveguide or transmission line, *input-output theory*[29] provides a quantum model of such a system, which includes the dynamics of some (not necessarily bosonic) system operator $\hat{a}$ expressed in terms of modified Heisenberg equation, known as a *Langevin equation*,

$$\frac{d\hat{a}}{dt} = -i[\hat{a}, \hat{H}_{\text{sys}}] + [\hat{a}, \hat{c}^\dagger]\left(-\frac{\gamma}{2}\hat{c} + \sqrt{\gamma}\hat{b}_{\text{in}}(t)\right) + \left(\frac{\gamma}{2}\hat{c}^\dagger - \sqrt{\gamma}\hat{b}_{\text{in}}^\dagger(t)\right)[\hat{a}, \hat{c}]. \tag{1.18}$$

Here, $\hat{b}_{\text{in}}(t)$ is a bosonic-like operator representing the input from the waveguide into the system at time t, and γ is the rate of energy loss from $\hat{c}$ into the waveguide. A similar output operator is

$$\hat{b}_{\text{out}}(t) = -\hat{b}_{\text{in}}(t) + \sqrt{\gamma}\hat{c}. \tag{1.19}$$

The operators $\hat{b}_{\text{in}}(t)$ and $\hat{b}_{\text{out}}(t)$ at one time and at another time correspond to separate modes, and should not be interpreted as any sort of time evolution. Loosely speaking, we can think of these operators as representing δ-function pulses that arrive at the system at time t, which can be integrated over to form arbitrary signals.

1.6.4 Density Matrices and the Master Equation

A *density matrix*[30] for a system is an operator which describes the probability distribution of states in that system, distinguishing 'classical' probability from quantum superpositions. For states $|\psi_k\rangle$ with probabilities p_k, the density matrix is

$$\hat{\rho} = \sum_k p_k |\psi_k\rangle \langle\psi_k| . \tag{1.20}$$

In the Schrödinger picture, density matrices evolve according to the *Master equation*[6]

$$\frac{d\hat{\rho}}{dt} = -i[\hat{H}, \hat{\rho}]. \tag{1.21}$$

The expectation value of an operator $\hat{O}$ can be calculated from a density matrix $\hat{\rho}$ as

$$\left\langle \hat{O} \right\rangle = \text{tr}(\hat{\rho}\hat{O}), \tag{1.22}$$

which, for a density matrix of the form in Equation 1.20, is the weighted sum of the expectation value for all $|\psi_k\rangle$ states

$$\left\langle \hat{O} \right\rangle = \sum_k p_k \langle\psi_k| \hat{O} |\psi_k\rangle . \tag{1.23}$$

Density Matrices and Quantum Decoherence

Consider the density matrix of a two-level system. A pure ground state $|0\rangle$ has density matrix

$$\hat{\rho} = |0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \tag{1.24}$$

whereas an even probabilistic mixture of $|0\rangle$ and $|1\rangle$ has density matrix

$$\hat{\rho} = \frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}. \tag{1.25}$$

This demonstrates that the diagonal elements of a density matrix represent probabilities of states. Indeed, $\text{tr}\,\hat{\rho} = 1$. The even superposition $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ has density matrix

$$\hat{\rho} = |\psi\rangle\langle\psi| = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}, \tag{1.26}$$

[6] Alternatively, the *Lindblad equation*

which demonstrates that the off-diagonal elements represent *coherences* between states. That is to say, they distinguish between superpositions and classical probabilities.

Density matrices are a useful formalism for expressing quantum decoherence. The two types of decoherence in quantum systems are *depolarisation* and *dephasing*. Depolarisation is unwanted transitions between states, which includes transitions to lower-energy states (*relaxation*) as well as unwanted excitations to higher-energy states. In a density matrix, this is represented by changes in the diagonal elements. Dephasing is drift in relative phase between states, caused by fluctuations in the energy levels. This is represented by decreases in the magnitudes of the off-diagonal elements. The dynamics of decoherence can be represented by additional terms in the Master equation. For decoherence resulting from the coupling of operators $\hat{A}_k$ to the environment, with rates γ_k, the Master equation becomes

$$\frac{d\hat{\rho}}{dt} = -i[\hat{H}, \hat{\rho}] + \sum_k \frac{\gamma_k}{2} \left(2\hat{A}_k \hat{\rho} \hat{A}_k^\dagger - \hat{A}_k^\dagger \hat{A}_k \hat{\rho} - \hat{\rho} \hat{A}_k^\dagger \hat{A}_k \right). \tag{1.27}$$

In both the original and prior modelling work presented in this thesis, density matrices are used to represent the states of atoms, but not cavities. This is because cavities do not dephase, only depolarise by gaining and losing photons, and so the input-output formalism of Subsection 1.6.3 is better suited.

1.7 Outline of Thesis

This thesis is structured as follows. Chapter 2 mostly presents prior work on both numerical and analytical modelling for atomic ensemble based microwave-optical transduction, that the original work in this thesis builds from. At the end of that chapter, in Section 2.5, it describes original work on analysing phase conventions and relations in that model. Chapter 3 describes original numerical modelling for transduction in four-level atomic systems. This model computes transduction signal strengths, and thereby conversion efficiencies. The chapter also compares model and experimental results. Chapter 4 describes numerical modelling for the pair generation rate resulting from biphoton generation in three-level atomic systems. Finally, Chapter 5 concludes the thesis.

Chapter 2

Prior Work on Transduction Modelling

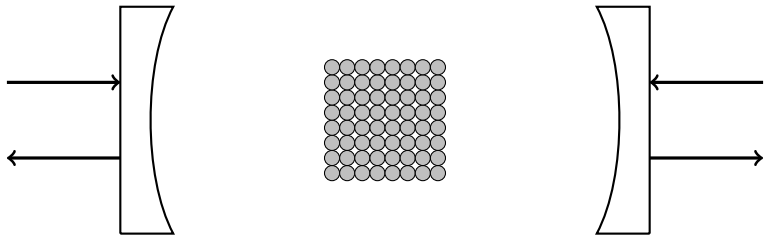


Figure 2.1: An illustration of a crystal in a cavity, with signals entering and exiting the cavity.

In this chapter, I review some existing modelling work[1, 2, 31] for microwave-optical quantum transduction using atomic ensembles in microwave and optical cavities. This prior work focuses on atoms in cavities rather than in free space, because cavity-based systems promise to be more efficient because of stronger light-matter interactions in cavities due to concentrated electromagnetic field. To begin, I present a fully quantum model for three-level atoms in cavities in Section 2.1, and then in later sections, I review the semiclassical models of References [1, 2, 31], and discuss the approximations made in those models. In Section 2.5, I investigate relations between input and output phases in these models, and find that there is a physically sensible phase relation only if a particular modification is made to the model. The phases of optical photons can be used to encode quantum information, and so accurate predictions of phase are important to such applications. Notation used here may differ from that of the original sources.

2.1 Quantum Model

A fully quantum model for such a system is the Jaynes-Cummings-like Hamiltonian[2]

$$\hat{H} = \hat{H}_{\text{cavities}} + \sum_{k=1}^{N} \hat{H}_{\text{atom},k} + \sum_{k=1}^{N} \hat{H}_{\text{int},k} \tag{2.1}$$

where N is the total number of active atoms and k is an index over the atoms. The Hamiltonian of the cavities is

$$\hat{H}_{\text{cavities}} = \delta_{co}\hat{a}^\dagger\hat{a} + \delta_{c\mu}\hat{b}^\dagger\hat{b} \tag{2.2}$$

where $\hat{a}$ and $\hat{b}$ are the lowering operators of the optical and microwave cavity modes respectively, and $\delta_{co} = \omega_{co} - \omega_o$ and $\delta_{c\mu} = \omega_{c\mu} - \omega_\mu$ are the detunings of the optical and microwave signals respectively from the resonant frequencies of the cavities. The other two components of Equation 2.1 depend on whether the three-level system being used is a Λ-system or a V-system. The atom Hamiltonian, with both cases explicitly written out, is

$$\hat{H}_{\text{atom},k} = \begin{cases} \delta_{\mu,k}\hat{\sigma}_{22,k} + \delta_{p,k}\hat{\sigma}_{33,k} & \Lambda\text{-system} \\ \delta_{o,k}\hat{\sigma}_{22,k} + \delta_{p,k}\hat{\sigma}_{33,k} & \text{V-system} \end{cases}. \tag{2.3}$$

$\hat{\sigma}_{ij,k} = |i_k\rangle\langle j_k|$ are atomic unit matrices, $\delta_{o,k} = \omega_{13,k} - \omega_o$ is the detuning of the optical signal from the $|1\rangle \leftrightarrow |3\rangle$ transition frequency, and $\delta_{\mu,k}$ and $\delta_{p,k}$ are the detunings of the microwave and pump signals respectively from their corresponding atomic transition frequencies. These transition frequencies are different for each atom due to inhomogeneous broadening. By conservation of energy, $\delta_{\mu,k} + \delta_{p,k} = \delta_{o,k}$, and so, even though we only directly control the frequency of the two inputs, we always know the frequency of the output. The interaction Hamiltonian is

$$\hat{H}_{\text{int},k} = \Omega_{p,k}\hat{\sigma}_{i_p j_p,k} + g_{o,k}\hat{a}\hat{\sigma}_{31,k} + g_{\mu,k}\hat{b}\hat{\sigma}_{i_\mu j_\mu,k} + \text{h.c.} \tag{2.4}$$

where $\Omega_{p,k}$ is the pump Rabi frequency on atom k, $g_{o,k}$ and $g_{\mu,k}$ are the coupling strengths of atom k to the optical and microwave cavities respectively, and i_p and j_p (i_μ and j_μ) are the lower and upper atomic levels respectively of the transition corresponding to the pump (microwave) signal.

The cavity Langevin equations for this system, which include damping and signal input-output, are

$$\frac{d\hat{a}}{dt} = -i\delta_{co}\hat{a} - i\sum_{k=1}^{N} g_{o,k}^*\hat{\sigma}_{13,k} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\hat{a} + \sqrt{\gamma_{oc}}\hat{a}_{\text{in}}(t) \tag{2.5}$$

$$\frac{d\hat{b}}{dt} = -i\delta_{c\mu}\hat{b} - i\sum_{k=1}^{N} g_{\mu,k}^*\hat{\sigma}_{i_\mu j_\mu,k} - \frac{\gamma_{\mu i}+\gamma_{\mu c}}{2}\hat{b} + \sqrt{\gamma_{\mu c}}\hat{b}_{\text{in}}(t) \tag{2.6}$$

where γ_{oi} and γ_{oc} ($\gamma_{\mu i}$ and $\gamma_{\mu c}$) are the energy loss rates of the optical (microwave) cavity through intrinsic damping and coupling to the input-output channel respectively. The output operators are, in analogy with Equation 1.19,

$$\hat{a}_{\text{out}}(t) = -\hat{a}_{\text{in}}(t) + \sqrt{\gamma_{oc}}\hat{a}(t) \tag{2.7}$$

$$\hat{b}_{\text{out}}(t) = -\hat{b}_{\text{in}}(t) + \sqrt{\gamma_{\mu c}}\hat{b}(t). \tag{2.8}$$

2.2 Adiabatic Elimination of the Atomic Dynamics

The fully quantum model is intractable to solve exactly. One approach to simplifying the model into something tractable is that of Williamson et. al. (2014) [31]. By assuming that the atom-signal detunings are large ($|\delta_{o,k}| \gg |g_{o,k}|$, $|\delta_{\mu,k}| \gg |g_{\mu,k}|$, and $|\delta_{o,k}\delta_{\mu_k}| \gg |\Omega_{p,k}|^2$), we can approximate the indirect interaction of the microwave and optical cavities as a direct interaction, *adiabatically eliminating*[32] the atomic dynamics. This gives an effective interaction Hamiltonian for the system

$$\hat{H}_{\text{eff}} = S\hat{a}^\dagger\hat{b} + S^*\hat{a}\hat{b}^\dagger \tag{2.9}$$

where the effective interaction strength is

$$S = \sum_{k=1}^{N} \frac{\Omega_{p,k} g_{\mu,k} g_{o,k}^*}{\delta_{o,k}\delta_{\mu,k}}. \tag{2.10}$$

Note that this is the same Hamiltonian as for two cavities that share a mirror, with the transduction process being analogous to photons passing through the shared mirror. The cavity Langevin equations are then

$$\frac{d\hat{a}}{dt} = -iS\hat{b} - \frac{\gamma_{oc}}{2}\hat{a} + \sqrt{\gamma_{oc}}\hat{a}_{\text{in}}(t) \tag{2.11}$$

$$\frac{d\hat{b}}{dt} = -iS^*\hat{a} - \frac{\gamma_{\mu c}}{2}\hat{b} + \sqrt{\gamma_{\mu c}}\hat{b}_{\text{in}}(t); \tag{2.12}$$

this model further assumes that all loss in the cavities is through the input-output channels, i.e. $\gamma_{oi} = \gamma_{\mu i} = 0$. In the steady state of the cavities, the conversion efficiency (in both directions) can be found analytically to be

$$\eta = \left| \frac{4iS\sqrt{\gamma_{oc}\gamma_{\mu c}}}{4|S|^2 + \gamma_{oc}\gamma_{\mu c}} \right|^2. \tag{2.13}$$

2.3 Semiclassical Cavity and Atomic Master Equation Steady States

A less simplistic model, one which must be solved numerically rather than analytically, is that of Fernandez-Gonzalvo et. al. (2019) [1]. That paper only explicitly describes Λ-systems, but the generalisation to V-systems is straightforward. In this model, the atom-cavity interaction is replaced with semiclassical (Rabi) drives in the atom Hamiltonian

$$\hat{H}_{\text{atom},k} = \begin{bmatrix} 0 & \Omega^*_{\mu,k} & \Omega^*_{o,k} \\ \Omega_{\mu,k} & \delta_{\mu,k} & \Omega^*_{p,k} \\ \Omega_{o,k} & \Omega_{p,k} & \delta_{p,k} \end{bmatrix} \tag{2.14}$$

where $\Omega_{\mu,k}$ and $\Omega_{o,k}$ are the Rabi frequencies of the driving that results from coupling to the microwave and optical cavities respectively. With α as the semiclassical amplitude of the optical cavity, $\Omega_{o,k} = g_{o,k}\alpha$. However, we do not treat the microwave cavity similarly, and instead assume that its amplitude is large enough that atomic absorption and emission is negligible, and so disregard the dynamical details of $\Omega_{\mu,k}$, setting it to be some constant value.

To handle the atomic dynamics, we use the Master equation

$$\frac{d\hat{\rho}_k}{dt} =: \mathcal{L}_k\hat{\rho}_k = -i[\hat{H}_{\text{atom},k}, \hat{\rho}_k] + \mathcal{L}_{\text{dec},k}\hat{\rho}_k \tag{2.15}$$

with decoherence operator

$$\begin{aligned} \mathcal{L}_{\text{dec},k}\hat{\rho}_k &= \mathcal{L}_{12,k}\hat{\rho}_k + \mathcal{L}_{13,k}\hat{\rho}_k + \mathcal{L}_{23,k}\hat{\rho}_k + \mathcal{L}_{2d,k}\hat{\rho}_k + \mathcal{L}_{3d,k}\hat{\rho}_k \\ \mathcal{L}_{ij,k}\hat{\rho}_k &= \begin{cases} \dfrac{\gamma_{ij}(n_{ij,k}+1)}{2}\left(2\hat{\sigma}_{ij,k}\hat{\rho}_k\hat{\sigma}_{ji,k} - \hat{\rho}_k\hat{\sigma}_{jj,k} - \hat{\sigma}_{jj,k}\hat{\rho}_k\right) & i=1, j=2 \\ + \dfrac{\gamma_{ij}n_{ij,k}}{2}\left(2\hat{\sigma}_{ji,k}\hat{\rho}_k\hat{\sigma}_{ij,k} - \hat{\rho}_k\hat{\sigma}_{ii,k} - \hat{\sigma}_{ii,k}\hat{\rho}_k\right) & \\ \frac{\gamma_{ij}}{2}\left(2\hat{\sigma}_{ij,k}\hat{\rho}_k\hat{\sigma}_{ji,k} - \hat{\rho}_k\hat{\sigma}_{jj,k} - \hat{\sigma}_{jj,k}\hat{\rho}_k\right) & \text{otherwise} \end{cases} \\ \mathcal{L}_{id,k}\hat{\rho}_k &= \frac{\gamma_{id}}{2}\left(2\hat{\sigma}_{ii,k}\hat{\rho}_k\hat{\sigma}_{ii,k} - \hat{\rho}_k\hat{\sigma}_{ii,k} - \hat{\sigma}_{ii,k}\hat{\rho}_k\right). \end{aligned} \tag{2.16}$$

γ_{2d} and γ_{3d} are the dephasing rates of levels $|2\rangle$ and $|3\rangle$ respectively with level $|1\rangle$, and $\gamma_{12,k}$, γ_{13}, and γ_{23} are the relaxation rates via the indicated transitions. $n_{12,k}$ is the mean thermal excitation count at $\omega_{12,k}$, as per the Bose-Einstein distribution, which is approximately zero for all other transition frequencies. For this transition,

$$\gamma_{12,k} = \frac{1}{\tau_{12}}\frac{1}{n_{12}+1} \tag{2.17}$$

where τ_{12} is the relaxation lifetime, whereas the other transitions follow the simpler $\gamma_{ij} = 1/\tau_{ij}$.

α evolves in time according to a semiclassical approximation of Equation 2.5, in which $\hat{a}$ is, of course, replaced with α, and the $\hat{\sigma}_{ij,k}$ operators in the atomic interaction terms are replaced with $\rho_{ij,k}$ to give

$$\frac{d\alpha}{dt} = -i\delta_{co}\alpha - i\sum_{k=1}^{N} g^*_{o,k}\rho_{13,k} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\alpha + \sqrt{\gamma_{oc}}\alpha_{\text{in}}. \tag{2.18}$$

2.3.1 Steady States

Despite this model being much smaller than the fully quantum model, operators having been replaced with complex numbers, it is still intractable to solve the time evolution of, because it requires N density matrices to be stored in memory. Finding steady states, however, is tractable with a few further simplifications. Let $\hat{\rho}_{k,SS}(\alpha)$ be the steady state of the density matrix of atom k given an optical cavity amplitude α, which is found by solving the linear system in Equation 2.15. We can drop the k index by including as function arguments all atom variables to obtain $\hat{\rho}_{SS}(\Omega_{p,k}, g_{o,k}, \alpha, \Omega_{\mu,k}, \delta_{o,k}, \delta_{\mu,k}, \omega_{12,k})$. If we assume that all atoms have the same coupling strengths and Rabi frequencies, $\hat{\rho}_{SS}$ varies only with α and the inhomogeneous shifts of each atom. This allows us to replace the sum in Equation 2.18 with an integral over the inhomogeneous distribution, of the form in Equation 1.2

$$\frac{d\alpha}{dt} = -i\delta_{co}\alpha - iNg^*_o\iint \rho_{13,SS}(\alpha, \delta_{12}, \delta_{23})p(\delta_{12}, \delta_{23})\,d\delta_{12}d\delta_{23} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\alpha + \sqrt{\gamma_{oc}}\alpha_{\text{in}}. \tag{2.19}$$

Here, $\delta_{ij} = \omega_{ij} - \omega'_{ij}$ is the inhomogeneous shift of an atomic transition frequency ω'_{ij} from some 'nominal' transition frequency ω_{ij}, and $p(\delta_{12}, \delta_{23})$ is the PDF of those shifts. A value of α for which Equation 2.19 is zero, i.e. a steady state, can be found using numerical root-finding, in which each iterative step involves evaluating the integral over the inhomogeneous distribution using some numerical quadrature scheme.

2.3.2 Further Development by Barnett and Longdell (2020)

Barnett and Longdell (2020) [2] further developed this model by including the dynamics of both cavities, with a semiclassical microwave cavity amplitude β from which the microwave Rabi frequency $\Omega_\mu = g_\mu \beta$ derives. Additionally, the assumption that all atoms have equal interaction strengths was replaced with the assumption that $N_o \leq N$ atoms have equal g_o and N_μ atoms have equal g_μ, with the remaining atoms not interacting with those cavities at all, due to being outside the mode volume. Put together, this replaces Equation 2.19 with the system

$$\frac{d\alpha}{dt} = -i\delta_{co}\alpha - iN_o g_o^* \iint \rho_{13,SS}(\alpha, \beta, \delta_{12}, \delta_{23}) p(\delta_{12}, \delta_{23})\, d\delta_{12} d\delta_{23} - \frac{\gamma_{oi} + \gamma_{oc}}{2}\alpha + \sqrt{\gamma_{oc}}\alpha_{\text{in}} \tag{2.20}$$

$$\frac{d\beta}{dt} = -i\delta_{c\mu}\beta - iN_\mu g_\mu^* \iint \rho_{12,SS}(\alpha, \beta, \delta_{12}, \delta_{23}) p(\delta_{12}, \delta_{23})\, d\delta_{12} d\delta_{23} - \frac{\gamma_{\mu i} + \gamma_{\mu c}}{2}\beta + \sqrt{\gamma_{\mu c}}\beta_{\text{in}}. \tag{2.21}$$

Numerical Methods

When performing the integral over the inhomogeneous distribution, $\hat{\rho}_{SS}$ will vary rapidly around values for which $\hat{H}_{\text{atom}}(\delta_{12}, \delta_{23})$ has degenerate eigenvalues. Accordingly, to achieve good numerical accuracy in the integral, samples should be concentrated around those parts of the domain. Barnett and Longdell address this by splitting the two-variable integral into an inner and outer integral, and, for each inner integral, performing root finding on the discriminant of the characteristic polynomial of $\hat{H}_{\text{atom}}$, which is zero where the Hamiltonian has degenerate eigenvalues, and partitioning the domain interval of the inner integral about that root point. The integral on each of those subintervals is evaluated using Gauss-Lobatto quadrature[33], which includes the endpoints of the interval in the nodes of integration. This ensures that the points of degenerate eigenvalues are not skipped in the numerical integral.

Real Density Matrix and Master Equation

An atomic density matrix $\hat{\rho}$ has nine complex elements, but because it is Hermitian, only nine real degrees of freedom. These degrees of freedom can be arranged in a real non-symmetric matrix

$$\hat{\rho}_{\text{real}} = \begin{bmatrix} \rho_{11} & \operatorname{Re}\rho_{12} & \operatorname{Re}\rho_{13} \\ \operatorname{Im}\rho_{12} & \rho_{22} & \operatorname{Re}\rho_{23} \\ \operatorname{Im}\rho_{13} & \operatorname{Im}\rho_{23} & \rho_{33} \end{bmatrix}. \tag{2.22}$$

Barnett (2019) [34] showed that this can be expressed as a linear transformation

$$\hat{\rho}_{\text{real}} = \mathcal{C}\hat{\rho}, \tag{2.23}$$

and so the Master equation can be expressed as

$$\begin{aligned} \frac{d\hat{\rho}_{\text{real}}}{dt} &= \mathcal{L}_{\text{real}}\hat{\rho}_{\text{real}} \\ \mathcal{L}_{\text{real}} &= \mathcal{C}\mathcal{L}\mathcal{C}^{-1}. \end{aligned} \tag{2.24}$$

Accordingly, steady states of the Master equation can be found as

$$\hat{\rho}_{SS} = \mathcal{C}^{-1}\hat{\rho}_{\text{real},SS} \tag{2.25}$$

where $\hat{\rho}_{\text{real},SS}$ is the steady state of Equation 2.24.

2.4 Comparisons of Models

Barnett (2019) [34] compared the numerical results of the semiclassical cavity amplitude and atomic master equation model to those of experiments with an Er:YSO (erbium doped in yttrium orthosilicate) crystal and found good agreement. That work also compared the theoretical and numerical results of that model to that of the simpler adiabatic model, and found significant disagreement between the two, at least for some choice of parameters. This demonstrates that the cavity amplitude and atomic master equation model is much more accurate than the adiabatic model.

2.5 Transduction Signal Phase Relations

In these models, the semiclassical approximation is formed by replacing the atomic unit matrices $\hat{\sigma}_{ij,k}$ with density matrix elements $\rho_{ij,k}$. However, the formal expectation values of these atomic unit matrices, using Equation 1.22, are in fact $\langle\hat{\sigma}_{ij,k}\rangle = \mathrm{tr}(\hat{\rho}_k\hat{\sigma}_{ij,k}) = \rho_{ji,k}$, which is the complex conjugate of $\rho_{ij,k}$. The papers acknowledge this, but use $\rho_{ij,k}$ instead. A complex conjugate flips phase and preserves magnitude, and so the choice of index order would have its effect on the output phases.

Using my own implementation of the model in Reference [2], I compute transduction signals for both index orders, to find the output phases. For each index order, I use the three different microwave input powers investigated in the paper[1] $P_\mu = -200\,\mathrm{dBm}, -75\,\mathrm{dBm}, 5\,\mathrm{dBm}$. For each, I perform 60 trials of random pairs of phases for Ω_p and β_{in}. The optical pump strength was kept constant for all evaluations, with $|\Omega_p| = 35\,\mathrm{kHz}$ as in Reference [2] (see Appendix A). All evaluations use zero detunings $\delta_o = \delta_\mu = \delta_{co} = \delta_{c\mu} = 0$. The results (Figure 2.2) showed that $\arg\Omega + \arg\beta_{\mathrm{in}} - \arg\alpha_{\mathrm{out}}$ was independent of phase for the $\rho_{ji,k}$ index order, but not for the $\rho_{ij,k}$ index order, and that this was the only 'conserved' phase sum. Therefore, the effect of this complex conjugation on phase is quite nontrivial, and only $\rho_{ji,k}$ has a physically sensible relationship between input and output phases, in which inputs and outputs have internally consistent and opposite phases. Accordingly, I use $\hat{\sigma}_{ij,k} \to \rho_{ji,k}$ in all of my original modelling.

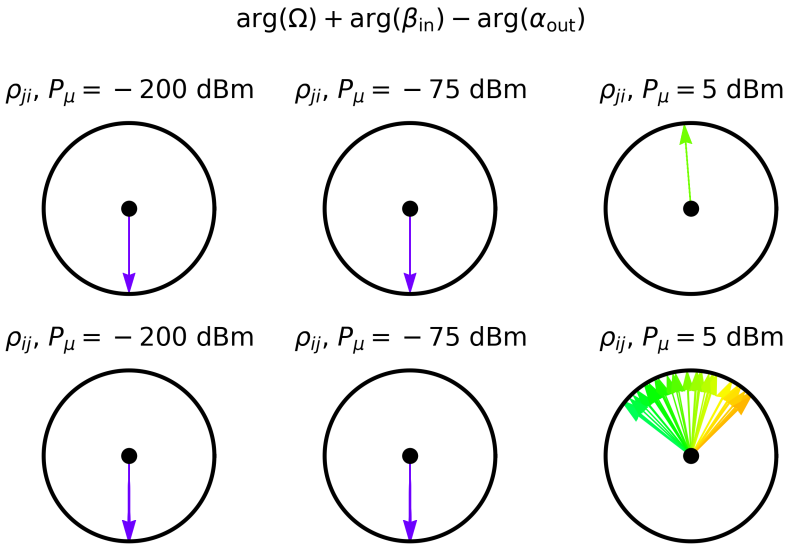


Figure 2.2: Sum of input phases minus output phase, for $\hat{\sigma}_{ij,k} \to \rho_{ji,k}$ (top) and $\hat{\sigma}_{ij,k} \to \rho_{ij,k}$ (bottom) and for different microwave powers (columns). For $\rho_{ji,k}$, this phase sum is consistent for all input phases, though it does vary between power levels. For $\rho_{ij,k}$, on the other hand, it varies, only slightly for low powers but quite substantially at high power.

[1] dBm is a 'unit' of power which is a decibel scale with $0\,\mathrm{dBm} = 1\,\mathrm{mW}$, so that e.g. $30\,\mathrm{dBm} = 1\,\mathrm{W}$.

Chapter 3

Transduction in a Four-Level System in Yb:YVO_4

This chapter describes original work on modelling the output power from transduction in a four-level atomic system, which can then be used to calculate transduction efficiency. This builds on concepts used in the three-level transduction models discussed in Chapter 2. Transduction in a four-level system involves multiple atomic transitions that produce output, which may have different phases from each other and therefore interfere. This can affect transduction efficiencies by orders of magnitude, but a three-level model does not account for it. This is applicable to many atomic platforms because the atomic levels used for transduction are almost always part of electronic multiplets, and so a 'three-level' transduction system will usually have a fourth level near the optical-separated level. Additionally, this chapter focuses specifically on transduction with atoms coupled directly to waveguides, rather than through cavities as in the prior work, which requires a different formalism to model.

To construct this model, I first constructed a four-level Hamiltonian and resultant Master equation for driven atoms, then used input-output theory to derive an expression for emission from the atoms, and finally expressed the overall emitted power from the entire ensemble as an integral over the inhomogeneous distribution. This thesis also presents numerical methods I developed for implementing the model. After describing this model, this chapter presents a comparison of the output powers computed using the model with those measured in experiments is presented, detailing the process of finding appropriate model parameters.

3.1 Target Platform and Benchmark Experimental Data

In constructing a model of four-level transduction. I aim to simulate the experiments described in Bartholomew et. al. 2020[3]. This work used an on-chip device (Figure 3.1) consisting of an optical waveguide constructed of $^{171}Yb^{3+}$:YVO_4 (yttrium orthovanadate doped with ytterbium) crystal, inside a microwave transmission line. For transduction experiments, the device is placed inside a dilution fridge and cooled to $\approx 1\,\mathrm{K}$.

$^{171}Yb^{3+}$, the active species in transduction, has a nuclear spin $I = 1/2$ and electron spin $S = 1/2$. The energy levels of $^{171}Yb^{3+}$ form electronic quadruplets that are non-degenerate (Zeeman-split) in the presence of an external magnetic field. The four levels relevant to the model are the upper two levels of the $^2F_{7/2}$ quadruplet ($|1\rangle$ and $|2\rangle$) and the lower two levels of the $^2F_{5/2}$ quadruplet ($|3\rangle$ and $|4\rangle$), which are separated by microwave transitions within a quadruplet and by near-infrared optical transitions between the quadruplets.

Transduction experiments in Reference [3] consisted of an optical pump at ω_p that was swept between and around the ω_{13} and ω_{23} transition frequencies and a continuous microwave drive at ω_μ that was swept around the ω_{34} transition frequencies, producing an optical output signal at $\omega_o = \omega_p + \omega_\mu$ through the $|4\rangle \to |1\rangle$ and $|4\rangle \to |2\rangle$ relaxations that is measured and recorded. Examples of these recorded transduction signal strengths are shown in Figure 3.2.

This experimental data is used (in Section 3.7) to benchmark the model, specifically a dataset

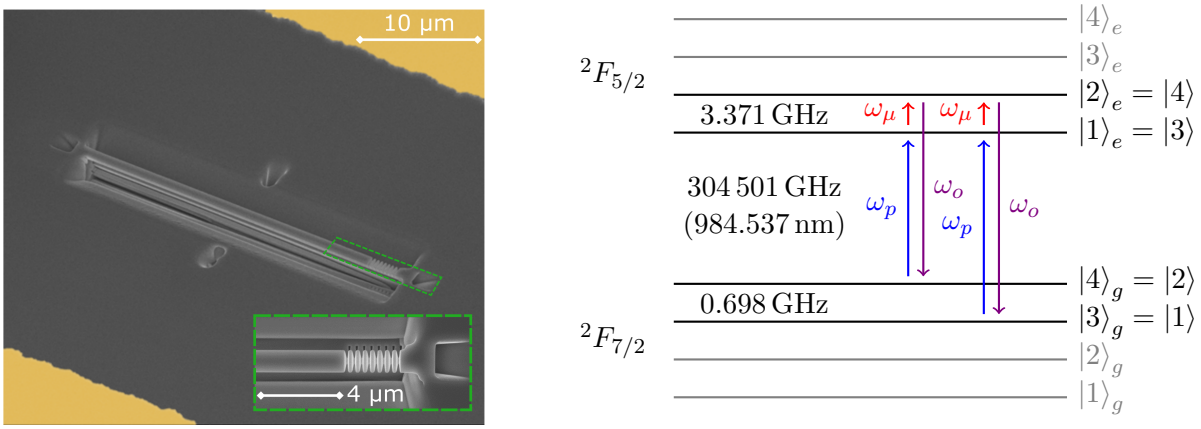


Figure 3.1: **Left** The transduction device in Reference [3], consisting of a suspended optical waveguide constructed of $^{171}Yb^{3+}$:YVO_4, terminating at a Bragg reflector so that signals enter and exit at the same end. Image credit: Reference [3]. **Right** The four-level system in $^{171}Yb^{3+}$:YVO_4 used, annotated with transition frequencies at $B_z = 2.09\,\text{mT}$ and with signal frequencies ω_μ, ω_p, and ω_o of the transduction experiments. Shown in grey are the unused levels of the electronic quadruplets.

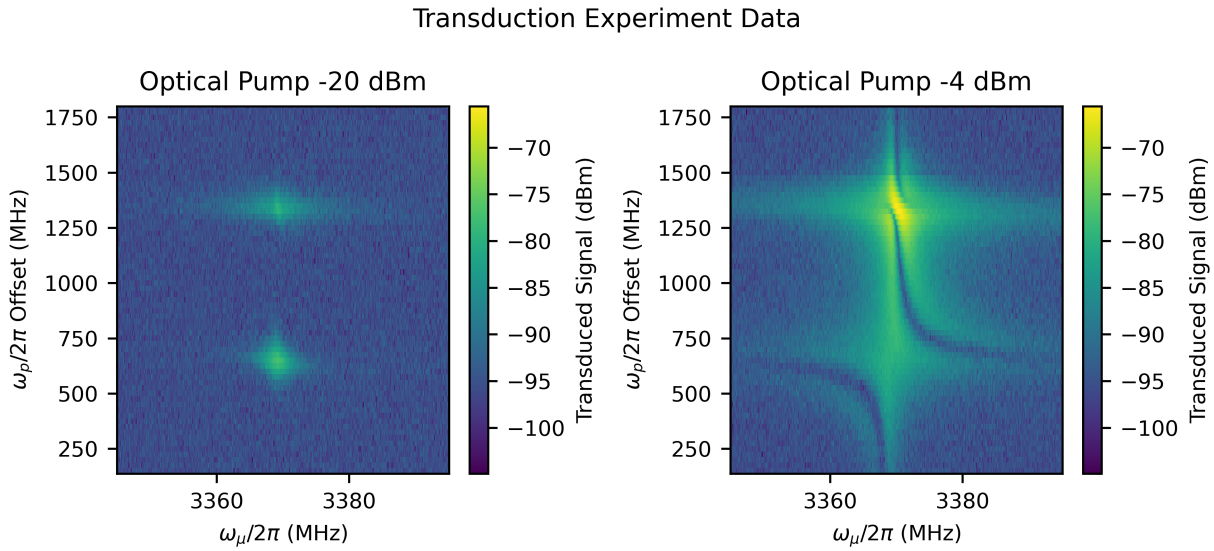


Figure 3.2: Experimental transduction signals measured by sweeping ω_p and ω_μ. Left shows data captured using a weak optical pump in which the transduction signal consists of simple bright spots near the atomic transition frequencies (One for $\omega_p = \omega_{13}$ and one for $\omega_p = \omega_{23}$). Right shows data captured with a strong optical pump, which exhibits nontrivial structure with thin curve-like features. Data credit: Bartholomew et. al. (unpublished).

that overlaps with the data published in Reference [3], but that also includes some unpublished data[1]. This dataset consists of frequency scans for optical pump powers ranging from $-40\,\text{dBm}$ to $-4\,\text{dBm}$ inclusive. Published in Reference [3] are experimental data in a weak optical pump regime in which the transduction signal simply consists of two spots around the two optical transition frequencies, which can be modelled as two separate three-level V-systems that do not significantly interact with each other. However, the unpublished data is in a strong optical pump regime in which the transduction signal contains features that stretch between both optical transitions. Reproducing these features was a goal of my modelling, and this requires a full four-level model.

[1]Given to me by my supervisor who is the lead author of Reference [3]

3.2 Driven Atom Hamiltonian

The effect of the optical pump is represented by Rabi frequencies Ω_{13} and Ω_{23} on those respective transitions, because both are driven by the one pump. Ω_μ is the Rabi frequency of the microwave drive on the $|3\rangle \to |4\rangle$ transition. Additionally, I include Rabi frequencies Ω_{14} and Ω_{24} to represent re-absorption of the emitted light by the atoms, possibly due to some back-reflection or weak cavity-like behaviour in the waveguide. Alternatively, these terms could be used in future modelling work for four-level atoms in cavities. Putting these together using blocks of Equation 1.13, the driven atom Hamiltonian in a static frame is

$$\hat{H}_{\text{static}}(\omega'_{12}, \omega'_{13}, \omega'_{14}) = \begin{bmatrix} 0 & 0 & \Omega_{13}^* e^{i\omega_p t} & \Omega_{14}^* e^{i\omega_o t} \\ 0 & \omega'_{12} & \Omega_{23}^* e^{i\omega_p t} & \Omega_{24}^* e^{i\omega_o t} \\ \Omega_{13} e^{-i\omega_p t} & \Omega_{23} e^{-i\omega_p t} & \omega'_{13} & \Omega_\mu^* e^{i\omega_\mu t} \\ \Omega_{14} e^{-i\omega_o t} & \Omega_{24} e^{-i\omega_o t} & \Omega_\mu e^{-i\omega_\mu t} & \omega'_{14} \end{bmatrix} \tag{3.1}$$

where $\omega'_{ij} = \omega_{ij} - \delta_{ij}$ is, as in Subsection 2.3.1, the $|i\rangle \to |j\rangle$ transition frequency of the atom, which is different for each atom because of inhomogeneous broadening. A unitary transformation to a frame co-rotating with the signals gives a time-independent Hamiltonian

$$\hat{H}(\delta_{12}, \delta'_p, \delta'_\mu) = \begin{bmatrix} 0 & 0 & \Omega_{13}^* & \Omega_{14}^* \\ 0 & \omega'_{12} & \Omega_{23}^* & \Omega_{24}^* \\ \Omega_{13} & \Omega_{23} & \delta'_p & \Omega_\mu^* \\ \Omega_{14} & \Omega_{24} & \Omega_\mu & \delta'_p + \delta'_\mu \end{bmatrix} \tag{3.2}$$

which is expressed in terms of detuning variables

$$\delta'_p = \omega'_{13} - \omega_p = \delta_p - \delta_{13} \tag{3.3}$$

$$\delta'_\mu = \omega'_{34} - \omega_\mu = \delta_\mu - \delta_{34}. \tag{3.4}$$

The Master equation for the atoms is then

$$\frac{d\hat{\rho}}{dt} =: \mathcal{L}(\delta_{12}, \delta'_p, \delta'_\mu)\hat{\rho} = -i[\hat{H}(\delta_{12}, \delta'_p, \delta'_\mu), \hat{\rho}] + \mathcal{L}_{\text{dec}}(\delta_{12}, \delta_{34})\hat{\rho} \tag{3.5}$$

where the decoherence operator

$$\begin{aligned} \mathcal{L}_{\text{dec}}\hat{\rho} &= \mathcal{L}_{12}\hat{\rho} + \mathcal{L}_{13}\hat{\rho} + \mathcal{L}_{14}\hat{\rho} + \mathcal{L}_{23}\hat{\rho} + \mathcal{L}_{24}\hat{\rho} + \mathcal{L}_{2d}\hat{\rho} + \mathcal{L}_{3d}\hat{\rho} + \mathcal{L}_{4d}\hat{\rho} \\ \mathcal{L}_{ij}\hat{\rho} &= \begin{cases} \dfrac{\gamma_{ij}(n'_{ij}+1)}{2}\left(2\hat{\sigma}_{ij}\hat{\rho}\hat{\sigma}_{ji} - \hat{\rho}\hat{\sigma}_{jj} - \hat{\sigma}_{jj}\hat{\rho}\right) & \\ \quad + \dfrac{\gamma_{ij} n'_{ij}}{2}\left(2\hat{\sigma}_{ji}\hat{\rho}\hat{\sigma}_{ij} - \hat{\rho}\hat{\sigma}_{ii} - \hat{\sigma}_{ii}\hat{\rho}\right) & i = 1, j = 2 \text{ or } i = 3, j = 4 \\ \frac{\gamma_{ij}}{2}\left(2\hat{\sigma}_{ij}\hat{\rho}\hat{\sigma}_{ji} - \hat{\rho}\hat{\sigma}_j - \hat{\sigma}_{jj}\hat{\rho}\right) & \text{otherwise} \end{cases} \\ \mathcal{L}_{id}\hat{\rho} &= \frac{\gamma_{id}}{2}\left(2\hat{\sigma}_{ii}\hat{\rho}\hat{\sigma}_{ii} - \hat{\rho}\hat{\sigma}_{ii} - \hat{\sigma}_{ii}\hat{\rho}\right) \end{aligned} \tag{3.6}$$

is analogous to that of Equation 2.16, and depends on the microwave transition shifts via the thermal excitation counts of the microwave transition frequencies n'_{12} and n'_{34}. The steady-state density matrix $\hat{\rho}_{SS}(\delta_{12}, \delta'_p, \delta'_\mu)$ can then be found by solving the linear system of the Master equation. In practice, I do this via the real version of the Master equation, as in Equation 2.24, because an $\mathbb{R}^{4\times4}$ system of equations is faster to solve than a $\mathbb{C}^{4\times4}$ system.

3.3 Atomic Output

Adapting from Equation 1.19, the input-output relation for a transition $|i\rangle \leftrightarrow |j\rangle$ of some atom is, up to some phase convention,

$$\hat{a}_{\text{out},ij} = -\hat{a}_{\text{in},ij} + \sqrt{\gamma_{ij,c}}\hat{\sigma}_{ij} \tag{3.7}$$

where $\gamma_{ij,c} \leq \gamma_{ij}$ is the relaxation rate of the atomic transition through coupling to the waveguide. To form a semiclassical approximation, I replace $\hat{a}_{\text{out},ij} \to \alpha_{\text{out},ij}$, as well as $\hat{a}_{\text{in},ij} \to \alpha_{\text{in},ij} = 0$ because the expectation value of the input is already captured by the Rabi frequencies on the atom Hamiltonian, and $\hat{\sigma}_{ij} \to \rho_{ji,SS}$ where I take the steady state, obtaining

$$\alpha_{\text{out},ij}(\delta_{12}, \delta'_p, \delta'_\mu) = \sqrt{\gamma_{ij,c}}\rho_{SS,ji}. \tag{3.8}$$

The output power from a given transition, then, is

$$P_{\text{atom},ij}(\delta_{12}, \delta'_p, \delta'_\mu) = \hbar\omega_o \left|\alpha_{\text{out},ij}\right|^2 = \hbar\omega_o\gamma_{ij,c} \left|\rho_{SS,ji}\right|^2. \tag{3.9}$$

Recall that there are two transitions producing output in this system, $|1\rangle \leftrightarrow |4\rangle$ and $|2\rangle \leftrightarrow |4\rangle$. The output power from each transition cannot simply be summed to obtain the total atomic output power, because each transition's emission may have different phases, and so they may interfere with each other. Specifically, because this model assumes that the light-matter interactions are through the dipole mechanism, the output has a phase related to that of the matrix element

$$d_{ij} = \langle i| \hat{d}_\parallel |j\rangle \tag{3.10}$$

of the component of the dipole moment operator parallel to the emission polarisation. To capture this, I re-express the atomic relaxation rates in terms of complex numbers C_{ij} for which

$$|C_{ij}|^2 = \gamma_{ij,c} \tag{3.11}$$

$$\arg C_{ij} = \arg d_{ij}. \tag{3.12}$$

Thus, the total atomic output power is

$$P_{\text{atom}}(\delta_{12}, \delta'_p, \delta'_\mu) = \hbar\omega_o \left|C_{14}\rho_{SS,41} + C_{24}\rho_{SS,42}\right|^2 =: \hbar\omega_o\Gamma_{\text{atom}} \tag{3.13}$$

$$\Gamma_{\text{atom}}(\delta_{12}, \delta'_p, \delta'_\mu) = \left|C_{14}\rho_{SS,41} + C_{24}\rho_{SS,42}\right|^2 \tag{3.14}$$

where Γ_{atom} is the photon emission rate from the atom.

3.4 Ensemble Output

The total power $P(\delta_p, \delta_\mu)$ from the ensemble can be found as an integral of the single atom power $P_{\text{atom}}(\delta_{12}, \delta'_p, \delta'_\mu)$ over the inhomogeneous distribution

$$P(\delta_p, \delta_\mu) = N \iiint p(\delta_{12}, \delta_{13}, \delta_{34}) P_{\text{atom}}(\delta_{12}, \delta_p - \delta_{13}, \delta_\mu - \delta_{34})\, d\delta_{12} d\delta_{13} d\delta_{34} \tag{3.15}$$

$$= \hbar\omega_o N \iiint p(\delta_{12}, \delta_{13}, \delta_{34}) \Gamma_{\text{atom}}(\delta_{12}, \delta_p - \delta_{13}, \delta_\mu - \delta_{34})\, d\delta_{12} d\delta_{13} d\delta_{34} \tag{3.16}$$

where $p(\delta_{12}, \delta_{13}, \delta_{34})$ is the PDF of the inhomogeneous distribution. I then make the approximation that the inhomogeneous shifts are much smaller than the transition frequencies $|\delta_{ij}| \ll \omega_{ij}$. Γ_{atom} depends on δ_{12} only via the shifted transition frequency ω'_{12}, and in this approximation $\omega'_{12} \approx \omega_{12}$, and so δ_{12} can be ignored. This is not true of the other shifts δ_{13} and δ_{34} because Γ_{atom} depends on them directly. In this approximation, then,

$$P(\delta_p, \delta_\mu) = \hbar\omega_o N \iint p(\delta_{13}, \delta_{34}) \Gamma_{\text{atom}}(\delta_p - \delta_{13}, \delta_\mu - \delta_{34})\, d\delta_{13} d\delta_{34} \tag{3.17}$$

$$= \hbar(\omega_{14} - \delta_p - \delta_\mu) N (p * \Gamma_{\text{atom}}) \tag{3.18}$$

where $p(\delta_{13}, \delta_{34}) = \int p(\delta_{12}, \delta_{13}, \delta_{34})\, d\delta_{12}$ is a marginal PDF, and ω_o has been re-expressed explicitly in terms of the detuning variables. Thus, numerically evaluating a grid of $P(\delta_p, \delta_\mu)$ is a matter of first evaluating a grid of $\Gamma_{\text{atom}}(\delta'_p, \delta'_\mu)$ and then convolving it with a grid of $p(\delta_{13}, \delta_{34})$, which is much cheaper computationally than evaluating the integral in Equation 3.15 (or even Equation 3.17) using numerical quadrature because convolutions can be evaluated using Fast Fourier Transform. Furthermore, in the experiments that I am modelling, $|\delta_p|, |\delta_\mu| \ll \omega_{14}$, and so the expression for ω_o in Equation 3.18 simplifies to

$$P = \hbar\omega_{14} N (p * \Gamma_{\text{atom}}). \tag{3.19}$$

3.5 Numerical Methods

As can be seen in Figure 3.2, there are thin curve-like features in the transduction signal. When evaluating a grid of Γ_{atom}, these curve features may be subject to grid *aliasing*, in which the relative alignment of the features with the grid points result in unphysical structure in the resulting grid of Γ_{atom} values, which is then blown up in P by convolution. In this section, I describe the methods I use to implement the model in a manner that is robust to grid aliasing while being computationally cheaper than simply using a finer grid.

3.5.1 Grid Aliasing

A mathematical description of grid aliasing is as follows. Letting Δ_{13} and Δ_{34} be the grid spacing of the discretised convolution kernel p, the discretised convolution in Equation 3.19 is

$$P(\delta_p, \delta_\mu) \approx \hbar\omega_{14} N \sum_{(i,j)\in\mathbb{Z}^2} p(i\Delta_{13}, j\Delta_{34}) \Gamma_{\text{atom}}(\delta_p - i\Delta_{13}, \delta_\mu - j\Delta 34) \tag{3.20}$$

$$= \iint \underbrace{p(\delta_{13}, \delta_{34}) \text{Ш}_{\Delta_{13}}(\delta_{13}) \text{Ш}_{\Delta_{34}}(\delta_{34})}_{\text{convolution kernel}} \Gamma_{\text{atom}}(\delta_p - \delta_{13}, \delta_\mu - \delta_{34})\, d\delta_{13} d\delta_{34} \tag{3.21}$$

where $\text{Ш}_T(x)$ is a Dirac comb of period T. This means that discretising the convolution in Equation 3.19 is equivalent to discretising the kernel into a weighted Dirac comb. Convolving with this kernel, unlike the original kernel, does not smooth out curve features, but merely displaces them, which is precisely the aliasing mentioned earlier. However, if our grid of Γ_{atom} contained the integral within the neighbourhood of a grid point instead of just the value at the grid point itself, the discretised convolution would be

$$P(\delta_p, \delta_\mu) \approx \hbar\omega_{14} N \sum_{(i,j)\in\mathbb{Z}^2} p(i\Delta_{13}, j\Delta_{34}) \int_{i-1/2}^{i+1/2} \int_{j-1/2}^{j+1/2} \Gamma_{\text{atom}}(\delta_p - i'\Delta_{13}, \delta_\mu - j'\Delta 34)\, di' dj' \tag{3.22}$$

$$= \iint \underbrace{p([i]\Delta_{13}, [j]\Delta_{34})}_{\text{convolution kernel}} \Gamma_{\text{atom}}(\delta_p - i\Delta_{13}, \delta_\mu - j\Delta_{34})\, didj \tag{3.23}$$

where $[x]$ is the rounding of x to the nearest integer. This convolution kernel is a step function, which is both closer to the original kernel than a weighted Dirac comb, and smooths out curve features because it is finite everywhere. Evaluating this integral within a neighbourhood for every grid point, however, is equivalent to simply evaluating the discrete convolution with a finer grid, which is the trivial way to deal with aliasing. Instead, my approach is to determine which neighbourhoods contain curve features, and replace only those grid points' values with integrals.

3.5.2 Feature Finding

To find which neighbourhoods contain curve features, it suffices to simply identify the points of intersection of these features with the edges of neighbourhoods, so that any neighbourhood lying on such an edge contains a curve feature[2]. This is illustrated in Figure 3.3. For an $n \times m$ grid, this process allows all nm neighbourhoods of grid points to be tested for feature presence by testing only the $(n+1)(m+1)$ lines that form edges between them.

As mentioned in Subsection 2.3.2, Reference [2] identified that curve features occur where the discriminant of the characteristic polynomial of the atom Hamiltonian

$$\Delta(\delta'_p, \delta'_\mu) = \text{Disc}_\lambda \left(\det \left(\hat{H}(\delta'_p, \delta'_\mu) - \lambda \hat{\mathbb{1}} \right) \right) \tag{3.24}$$

[2] Alternatively, curve features could be closed loops contained entirely within that single neighbourhood. I have not observed closed loop features in practice, and even if they do exist, a grid that is coarse enough to contain an entire such loop within a single point's neighbourhood is too coarse to be very useful.

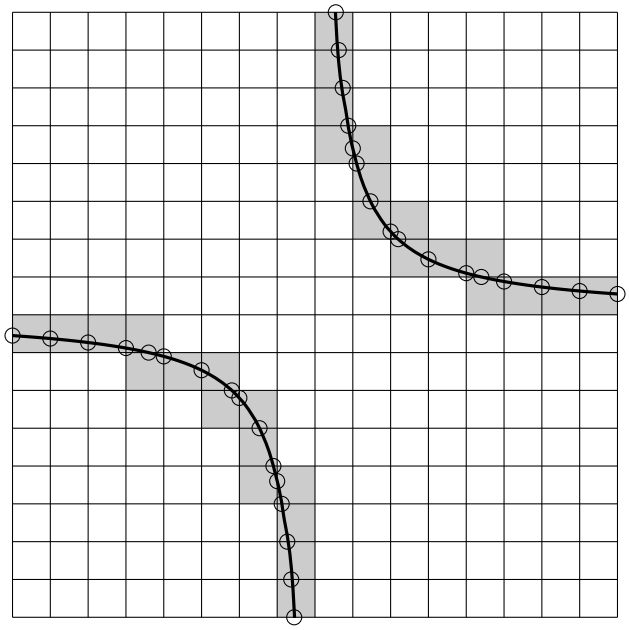


Figure 3.3: An illustration of feature finding. The grid squares are neighbourhoods around grid points, and the thick curves represent the features we want to find. The intersections of the feature curves with the neighbourhood edges are shown by circles, and the neighbourhoods containing these features are highlighted grey.

has its roots. Because the Hamiltonian is Hermitian and therefore its characteristic polynomial has exclusively real roots, this discriminant is uniformly non-negative. This means all roots of the discriminant are 'partial' (single-variable) local minima, and it is more numerically stable to identify partial local minima via roots of the discriminant's partial derivative than to directly find roots of the discriminant itself[34]. Reference [2] used generic numerical root-finding, but I instead use polynomial-specific root-finding that uses the polynomial coefficients of $\Delta(\delta'_p, \delta'_\mu)$ and its partial derivatives. This has the advantage of finding all roots rather than just one.

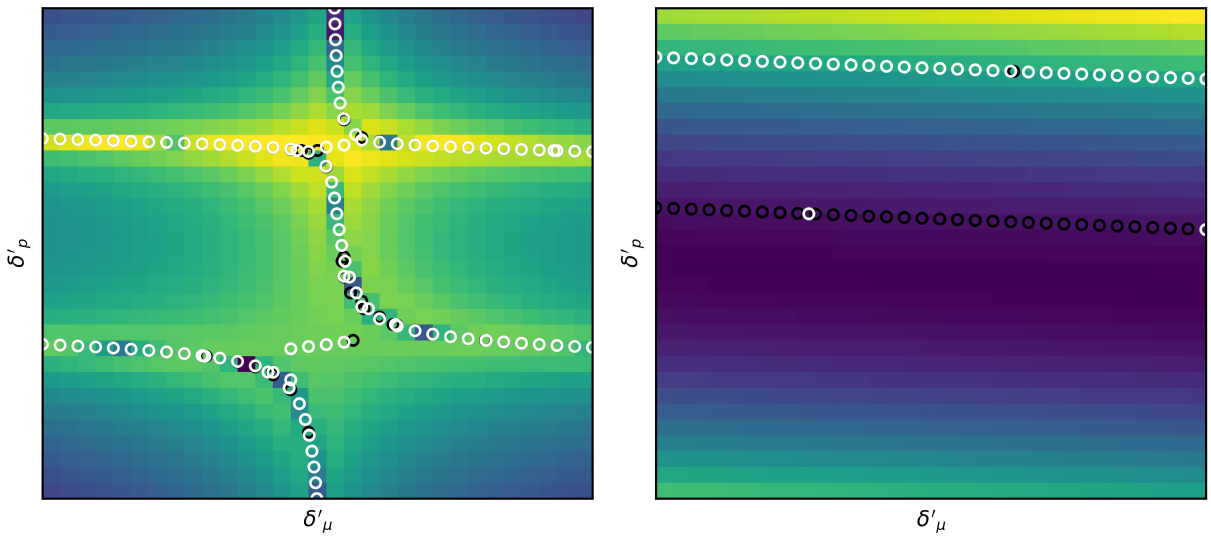


Figure 3.4: Γ_{atom} grids with curve feature intersections identified by along-edge partial local minima (white circles) and across-edge partial local minima (black circles). Left has the same domain as Figure 3.2 and shows aliasing artefacts along curve features, and right is a close-up around a curve feature, showing duplicate identification and misalignment from the feature's centre.

The procedure is as follows.

1. Precompute the bivariate polynomial coefficients of $\Delta(\delta'_p, \delta'_\mu)$, $\frac{\partial\Delta}{\partial\delta'_p}(\delta'_p, \delta'_\mu)$, $\frac{\partial^2\Delta}{\partial\delta'^2_p}(\delta'_p, \delta'_\mu)$, $\frac{\partial\Delta}{\partial\delta'_\mu}(\delta'_p, \delta'_\mu)$, and $\frac{\partial^2\Delta}{\partial\delta'^2_\mu}(\delta'_p, \delta'_\mu)$, because only the detuning variables change over this process.

2. To find all the intersections along some δ'_p edge (constant δ'_μ), evaluate the univariate polynomial coefficients of $\frac{\partial\Delta}{\partial\delta'_p}(\delta'_p)$ for the edge's δ'_μ, and perform root-finding with them to find all critical points of $\Delta(\delta'_p)$.

3. Evaluate Δ at each critical point. Critical points whose values of Δ are smaller than those of their immediately adjacent critical points are local minima and are therefore taken to be curve feature intersections.

4. Evaluate the polynomial coefficients of $\frac{\partial\Delta}{\partial\delta'_\mu}(\delta'_p)$ and use them to find critical points of $\Delta(\delta'_\mu)$.

5. Evaluate the polynomial coefficients of $\frac{\partial^2\Delta}{\partial\delta'^2_\mu}(\delta'_p)$ and use them to perform the second derivative test $\frac{\partial^2\Delta}{\partial\delta'^2_\mu} \geq 0$ for each critical point of $\Delta(\delta'_\mu)$ to identify local minima.

This process is repeated for each δ'_p edge, and then vice-versa (δ'_p and δ'_μ swapped) for all δ'_μ edges (constant δ'_p).

For each of the two edge directions, both partial local minima along the edge and perpendicular to the edge are identified by this procedure. However, there is an asymmetry between these in that the former use direct comparisons between critical points and the latter use a second derivative test; the latter is to avoid having to evaluate extra Δ values outside the edge being tested. Furthermore, the second derivative test is done inclusive of exact equality in order to err on the side of false positives rather than false negatives.

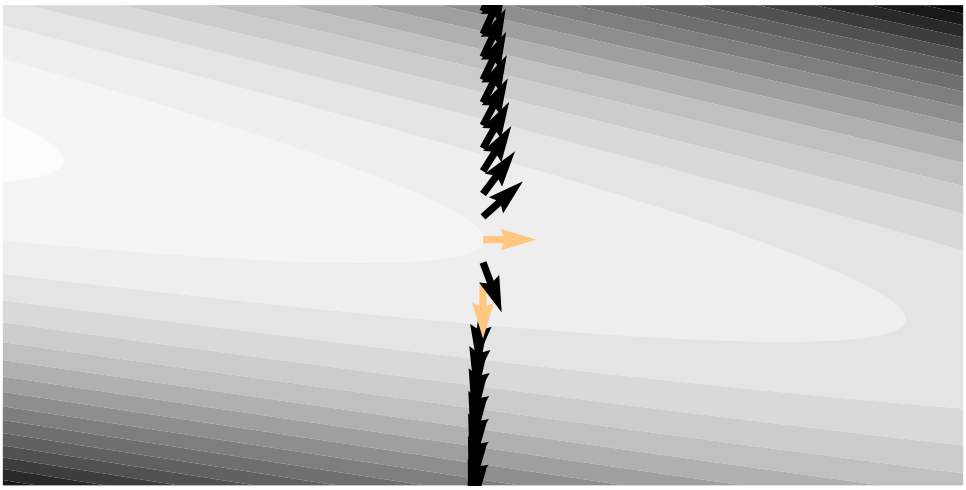


Figure 3.5: Around a curve feature, Δ often takes the shape of a 'trench' which is non-constant at its bottom. When scanning the gradient (arrows) of Δ in such a region, the gradient flips around on either side, but it is not zero at any point, and so it rotates through the parallel and perpendicular (orange arrows), resulting in this curve feature being found twice.

As is shown by Figure 3.4, this procedure successfully identifies curve features that produce aliasing artefacts. Additionally, there are curve feature intersections that are found only by the across-edge test, and not the along-edge test, and so this test is necessary to find all grid point neighbourhoods containing curve features. Checking both parallel and perpendicular partial local minima, however, has the side effect of causing many curve features to be identified twice, but this has no effect beyond slightly increasing computational cost. The reasons for this are explained in Figure 3.5. Additionally, these partial local minima are often very slightly misaligned from the actual curve features, by an amount that is much smaller than most useful grid spacings, and that therefore does not have any significant effect on the final results of this process.

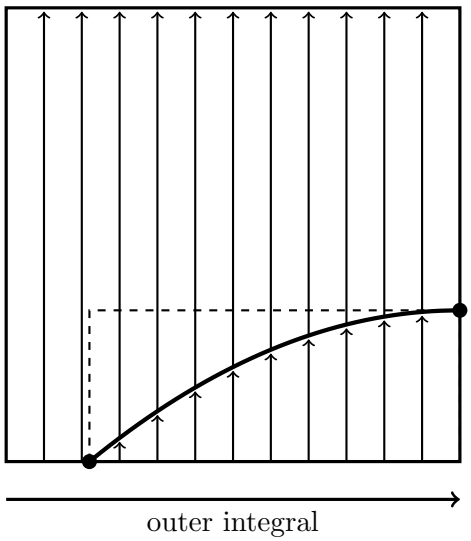


Figure 3.6: The integration within a single grid point's neighbourhood. The curve feature has an axis-aligned bounding box (dashed) computed from its edge intersections (filled circles) that is wider than it is tall proportional to the neighbourhood's dimensions, and so the outer integral is horizontal and the inner integral (which is represented by the vertical arrows) is vertical. The inner integral's domain is split at curve feature intersections (and surrounding points, which are not shown here).

3.5.3 Neighbourhood Integration

Once the points whose neighbourhoods contain curve features are identified, the integral within the neighbourhood must be evaluated using some numerical quadrature scheme. To do this, I use the discriminant once again to perform importance sampling along an inner integral, using Gauss-Lobatto quadrature, as in Reference [2]. For each intersection point, the inner integral's domain is split at intersections with curve features, as well as at addition points surrounding the intersections, with each interval between splits evaluated using a separate instance of Gauss-Lobatto quadrature. If the inner integral is along δ'_p, then for each curve intersection ${\delta'_p}^{(*)}$, the split points are ${\delta'_p}^{(*)}$ itself, ${\delta'_p}^{(*)} \pm \gamma_{ph}$, ${\delta'_p}^{(*)} \pm 3\gamma_{ph}$, and ${\delta'_p}^{(*)} \pm 10\gamma_{ph}$, where $\gamma_{ph} = \gamma_{3d}$ is used as an estimate of the homogeneous linewidth and therefore the thickness of the curve feature. If the inner integral is along δ'_μ, the splits points from each intersection ${\delta'_\mu}^{(*)}$ are the intersection itself as well as ${\delta'_\mu}^{(*)} \pm \gamma_{\mu h}$, ${\delta'_\mu}^{(*)} \pm 3\gamma_{\mu h}$, and ${\delta'_\mu}^{(*)} \pm 10\gamma_{\mu h}$, where $\gamma_{\mu h} = \gamma_{3d} + \gamma_{4d}$. Split points outside the bounds of the integral (the neighbourhood edges) are excluded.

Reference [2] chose the inner and outer integral axes arbitrarily, but the outer integral would ideally be as close to parallel to the curve feature as possible, because that minimises the distance along the curve between sample points. To handle this, I re-use the curve intersection data computed in the previous step to find an axis-aligned bounding box for the curve feature inside the neighbourhood, and let the outer integral axis be the axis along which this bounding box takes up the largest fraction of the neighbourhood's size (Figure 3.6).

3.6 Experimental Parameters for Model

To summarise, this model requires as input the following parameters:

- Microwave transition frequencies ω_{12} and ω_{34}
- Rabi frequencies Ω_{13}, Ω_{23}, Ω_{14}, Ω_{24}, and Ω_μ
- Microwave transition relaxation lifetimes τ_{12} and τ_{34} and operating temperature T
- Optical transition relaxation rates γ_{13}, γ_{23}, γ_{14}, and γ_{24}

- Optical output waveguide couplings C_{14} and C_{24}
- Inhomogeneous PDF $p(\delta_{13}, \delta_{34})$
- Atom count N and optical transition frequency ω_{14}.

N merely scales the output signal power uniformly, and therefore does not need much precision, and can be found through trial and error. Reference [3] quotes an estimate for the operating temperature of $T \approx 1\,\mathrm{K}$, and a microwave Rabi frequency $\Omega_\mu = 2\pi \times 1\,\mathrm{MHz}$.

3.6.1 Inhomogeneous Broadening

Reference [3] specifies the inhomogeneous distribution as Gaussian with standard deviations $\Gamma_{\mathrm{ih},o} \approx 200\,\mathrm{MHz}$ for optical transitions and $\Gamma_{\mathrm{ih},\mu} \approx 130\,\mathrm{kHz}$ for microwave transitions, with a correlation slope of -120 (optical/microwave) between them. Using the formula

$$\text{slope} := \frac{\Delta y}{\Delta x} = \frac{\sigma_{xy}}{\sigma_x^2} \tag{3.25}$$

gives a covariance of $-2.028\,\mathrm{MHz}^2$, and so the inhomogeneous PDF is that of the bivariate normal distribution

$$p\left(\boldsymbol{\delta} = \begin{bmatrix}\delta_{13}\\ \delta_{34}\end{bmatrix}\right) = \frac{1}{\sqrt{2\pi \det \Sigma}} \exp\left(-\frac{1}{2}\boldsymbol{\delta}^T \Sigma^{-1} \boldsymbol{\delta}\right) \tag{3.26}$$

$$\Sigma = \begin{bmatrix}(200\,\mathrm{MHz})^2 & -2.028\,\mathrm{MHz}^2\\ -2.028\,\mathrm{MHz}^2 & (130\,\mathrm{kHz})^2\end{bmatrix}. \tag{3.27}$$

3.6.2 Spin Hamiltonian

To find the remaining parameters, I make use of the spin Hamiltonians[35] of the two electronic quadruplets in this system

$$\hat{H}_{g,e} = \mu_B \mathbf{B}^T g_{g,e} \hat{\mathbf{S}} + \hat{\mathbf{I}}^T A_{g,e} \hat{\mathbf{S}} \tag{3.28}$$

where subscripts g and e are indices indicating the ground ($^2F_{7/2}$) and excited ($^2F_{5/2}$) multiplets respectively, $g_{g,e}$ are Zeeman interaction tensors, and $A_{g,e}$ are hyperfine interaction tensors. $\mathbf{B}$ is the external magnetic field, $\hat{\mathbf{S}} = [\hat{S}_x, \hat{S}_y, \hat{S}_z]^T$ is the electron spin operator and $\hat{\mathbf{I}} = [\hat{I}_x, \hat{I}_y, \hat{I}_z]^T$ is the nuclear spin operator. The symmetry of the crystal site occupied by ytterbium ions[3] means that the $g_{g,e}$ and $A_{g,e}$ tensors have two eigenvalues, one unique (multiplicity 1) and one non-unique (multiplicity 2). Denoting the unique eigenvalues as $g_{\parallel g,e}$ and $A_{\parallel g,e}$ and the non-unique eigenvalues as $g_{\perp g,e}$ and $A_{\perp g,e}$, Equation 3.28 expands into

$$\hat{H}_{g,e} = \mu_B[g_{\perp g,e}(B_x\hat{S}_x + B_y\hat{S}_y) + g_{\parallel g,e}B_z\hat{S}_z] + A_{\perp g,e}(\hat{I}_x\hat{S}_x + \hat{I}_y\hat{S}_y) + A_{\parallel g,e}\hat{I}_z\hat{S}_z \tag{3.29}$$

where it is assumed without generality that z is the unique axis and x and y are the non-unique axes[4]. The magnetic field in the benchmarking dataset is $\mathbf{B} = [0, 0, 2.09\,\mathrm{mT}]^T$, of which the only nonzero component is B_z, but the other components are kept for later calculations regarding magnetic field noise. The interaction tensor elements are shown in Table 3.1.

3.6.3 Transition Frequencies

The eigenvectors of $\hat{H}_g$ are the four levels of the multiplet (Figure 3.1), and so the transition frequencies within a multiplet are simply the difference of the corresponding eigenvalues, which gives us

$$\omega_{12} = \langle 2|\, \hat{H}_g\, |2\rangle - \langle 1|\, \hat{H}_g\, |1\rangle \tag{3.30}$$

$$\omega_{34} = \langle 4|\, \hat{H}_e\, |4\rangle - \langle 3|\, \hat{H}_e\, |3\rangle \tag{3.31}$$

where $\hat{H}_{g,e}$ are evaluated with $B_z = 2.09\,\mathrm{mT}$. At zero magnetic field[3], $\omega_{13} = 2\pi \times 304\,501.0\,\mathrm{GHz}$, and because this is only used to calculate the scale of the output energy (via $\omega_{14} = \omega_{12} + \omega_{23} + \omega_{34}$), this is a good enough approximation of the value at $B_z = 2.09\,\mathrm{mT}$.

[3] In the usual crystallography notation, this is the D_{2d} symmetry group

[4] In the usual crystallography notation, $c \parallel z$

Parameter	Value
$g_{\perp g}$	0.85
$g_{\parallel g}$	-6.08
$g_{\perp e}$	1.7
$g_{\parallel e}$	2.51
$A_{\perp g}$	$2\pi \times 675\,\mathrm{MHz}$
$A_{\parallel g}$	$2\pi \times -4.82\,\mathrm{GHz}$
$A_{\perp e}$	$2\pi \times 3.37\,\mathrm{GHz}$
$A_{\parallel e}$	$2\pi \times 4.86\,\mathrm{GHz}$

Table 3.1: Interaction tensor elements of the spin Hamiltonian in Equation 3.29. $g_{\perp g}$ and $g_{\parallel g}$ are from Reference [36], and the remaining values are from from Reference [35].

3.6.4 Dephasing Rates

If we assume that magnetic field noise, which would be caused by the nuclear spin flips in the yttrium and vanadium of the host crystal, is the dominant source of dephasing in this system, then the dephasing rates are proportional to

$$\gamma_{id} \propto \left|\left| \frac{d\omega_{1i}}{d\mathbf{B}} \right|\right|^2. \tag{3.32}$$

The transition frequency is

$$\omega_{1i} = \langle i| \hat{H}_{g,e} |i\rangle - \langle 1| \hat{H}_g |1\rangle \tag{3.33}$$

where the index on the first $\hat{H}_{g,e}$ is g if $i = 2$ and e if $i \in \{3, 4\}$. Substituting Equation 3.29 and differentiating, we obtain

$$\gamma_{id} \propto \left|\left| g_{g,e} \langle i| \hat{\mathbf{S}} |i\rangle - g_g \langle 1| \hat{\mathbf{S}} |1\rangle \right|\right|^2 \tag{3.34}$$

where the index on the first $g_{g,e}$ is the same index as the one on $\hat{H}_{g,e}$. In practice, $\hat{S}_z$ is the only nonzero component of $\hat{\mathbf{S}}$ when restricting these operators to the $(|1\rangle, |2\rangle, |3\rangle, |4\rangle)$ basis, and so Equation 3.34 becomes

$$\gamma_{id} \propto \left| g_{\parallel g,e} \langle i| \hat{S}_z |i\rangle - g_{\parallel g} \langle 1| \hat{S}_z |1\rangle \right|^2. \tag{3.35}$$

3.6.5 Dipole Moments

The remaining parameters are proportional to dipole moment matrix elements. These include the atomic coupling depolarisation rates

$$\gamma_{ij,c} \propto \mathbf{d}_{ij} = \left|\left| \langle i| \hat{\mathbf{d}} |j\rangle \right|\right|^2 \tag{3.36}$$

as well as, from Equation 1.14,

$$\Omega_{ij} \propto d_{\parallel ij} \tag{3.37}$$

where $d_{\parallel ij}$ is the component of $\mathbf{d}_{ij}$ parallel to the drive polarisation. In this system, both electric and magnetic dipoles are proportional to electron spin

$$\hat{\mathbf{d}} \propto \hat{\mathbf{S}}, \tag{3.38}$$

and so the spin Hamiltonian can once again be used. First of all, Equation 3.37 gives a ratio

$$\frac{\Omega_{13}}{\Omega_{23}} = \frac{\langle 1| \hat{S}_z |3\rangle}{\langle 2| \hat{S}_z |3\rangle} \tag{3.39}$$

between the optical pump Rabi frequencies, where the fact that $\hat{S}_z$ is the only nonzero spin component has once again been used. The remaining optical Rabi frequencies are set to zero ($\Omega_{14} = 0 = \Omega_{24}$) because the transduction efficiencies in the benchmarking dataset are quite low ($< 10^{-5}$). Second,

from Reference [35], $\gamma_{14}(\mathbf{B}=\mathbf{0}) = 1.4\,\text{kHz}$ and $\gamma_{23}(\mathbf{B}=\mathbf{0}) = 1.3\,\text{kHz}$, and optical transitions $|1\rangle \leftrightarrow |3\rangle$ and $|2\rangle \leftrightarrow |4\rangle$ are forbidden at zero magnetic field. In terms of the spin operators, these forbidden transitions are reflected by the fact that

$$\langle 1|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|4\rangle = -\langle 2|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|3\rangle \neq 0 \tag{3.40}$$

$$\langle 1|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|3\rangle = \langle 2|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|4\rangle = 0. \tag{3.41}$$

The fact that dipole-forbidden transitions have depolarisation rates much lower than dipole-permitted transitions shows that these optical depolarisations happen predominantly through dipole emission, and so I set $\gamma_{ij} = \gamma_{ij,c}$. At $B_z = 2.09\,\text{mT}$, $\langle 1|\,\hat{S}_z\,|3\rangle = \langle 2|\,\hat{S}_z\,|4\rangle$ take on a small nonzero value, and $\langle 1|\,\hat{S}_z\,|4\rangle = -\langle 2|\,\hat{S}_z\,|3\rangle$ decrease in magnitude as the levels hybridise. Using Equation 3.36,

$$\begin{aligned}
\gamma_{14} &= \frac{\left|\langle 1|\,\hat{S}_z\,|4\rangle\right|^2}{\left|\langle 1|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|4\rangle\right|^2}\gamma_{14}(\mathbf{B}=\mathbf{0}) \\
\gamma_{23} &= \frac{\left|\langle 2|\,\hat{S}_z\,|3\rangle\right|^2}{\left|\langle 2|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|3\rangle\right|^2}\gamma_{23}(\mathbf{B}=\mathbf{0}) \\
\gamma_{13} &= \frac{\left|\langle 1|\,\hat{S}_z\,|3\rangle\right|^2}{\left|\langle 2|\,\hat{S}_z\,|3\rangle\right|^2}\gamma_{23} \\
\gamma_{24} &= \frac{\left|\langle 2|\,\hat{S}_z\,|4\rangle\right|^2}{\left|\langle 1|\,\hat{S}_z\,|4\rangle\right|^2}\gamma_{23}
\end{aligned} \tag{3.42}$$

and C_{14} and C_{24} are set accordingly by Equations 3.11 and 3.12. From Reference [37], $\tau_{12}(\mathbf{B}=\mathbf{0}) = 54\,\text{ms}$, and so

$$\tau_{12} = \frac{\left|\langle 1|\,\hat{S}_z(\mathbf{B}=\mathbf{0})\,|2\rangle\right|^2}{\left|\langle 1|\,\hat{S}_z\,|2\rangle\right|^2}\tau_{12}(\mathbf{B}=\mathbf{0}). \tag{3.43}$$

In the absence of data on τ_{34}, I set it to 10 ms, to match the order of magnitude of τ_{12}.

3.6.6 Optical Pump Calibration

From Equation 1.14, $\Omega_{23} \propto \sqrt{P}$ where P is the pump power. The equation for this proportionality, appropriately calibrated, is

$$\Omega_{23} = \frac{\Omega_{23,\text{ref}}\sqrt{\eta}}{\sqrt{P_{\text{ref}}}}\sqrt{P} \tag{3.44}$$

where $(\Omega_{23,\text{ref}}, P_{\text{ref}})$ is some known 'reference' Rabi frequency-power pair for calibration, and η is the efficiency with which power from the source makes it into the waveguide. Reference [3] contains a frequency-power pair ($\Omega_{23,\text{ref}} = 2\pi \times 6\,\text{MHz}, P_{\text{ref}} = 2\,\mu\text{W}$), and my supervisor provided data with which to calibrate the efficiency to $\eta = 0.055$ (Appendix B).

3.7 Results

To evaluate the model, I computed power-frequency grids with frequency domain and input parameters corresponding to the experimental data. Because the model does not include the noise in the experimental apparatus, I add simulated noise to the computed power. This consists of random samples from the experimental frequency sweep with $-40\,\text{dBm}$ optical pump power, which contains no discernible transduction signal, only noise. A comparison of the resultant power-frequency grids with experimental data is shown in Figure 3.7.

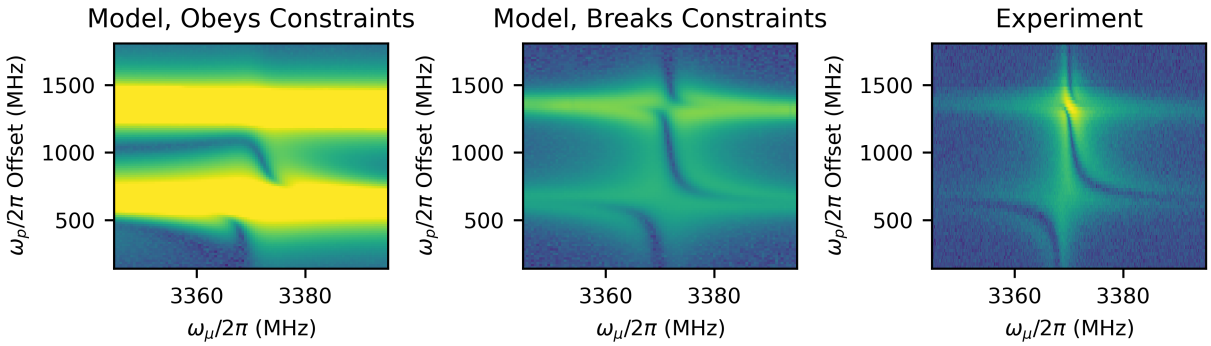


Figure 3.7: Power-frequency grids from the model when obeying the constraints found in Section 3.6 (left) and when disobeying those constraints (middle) as compared with the experimental data (right) shown in Figure 3.2, with optical pump power $-4\,\mathrm{dBm}$. All three plots have the same frequency and colour scales. A simple noise model is used in this comparison.

After the analysis in Section 3.6, the remaining free parameters are N and γ_{2d}, as well as some leeway in adjusting Ω_{23} since the calibration parameters $\Omega_{23,\mathrm{ref}}$ and P_{ref} have only one significant figure of precision. I adjusted these free parameters to find a set of parameters that best replicated the experimental data, which resulted in $N = 1 \times 10^{14}$, $\gamma_{2d} = 10\,\mathrm{kHz}$, and an adjustment to the Ω_{23} calibration of $P_{\mathrm{ref}} = 1.7\,\mu\mathrm{W}$.

This showed only superficial qualitative agreement, and so I next adjusted the parameters while breaking the constraints identified in Section 3.6, finding better agreement with $N = 1 \times 10^{10}$, $\tau_{12}(\mathbf{B} = \mathbf{0}) = 1\,\mu\mathrm{s}$, and a modified *optical hybridisation ratio*

$$\frac{\left|\langle 1|\,\hat{S}_z\,|3\rangle\right|}{\left|\langle 2|\,\hat{S}_z\,|3\rangle\right|} = \frac{\left|\langle 2|\,\hat{S}_z\,|4\rangle\right|}{\left|\langle 1|\,\hat{S}_z\,|4\rangle\right|} \approx 0.12 \to 0.38, \tag{3.45}$$

and all downstream variables τ_{12}, γ_{13}, γ_{23}, γ_{14}, γ_{24}, and Ω_{13} modified accordingly using Equations 3.43, 3.42, and 3.39, preserving the phase of Ω_{13}/Ω_{23}.

In both model grids, we see dark curve features, corresponding to destructive interference between the two output transitions, that span the distance between the two atomic transition frequencies ω_{23} and ω_{13}, which are also seen in the experimental data, and can only be produced by a model that accounts for all four levels and both output transitions simultaneously. However, their locations are much more accurate to the experimental data in the model grid evaluated using the constraint-breaking parameter set. Furthermore, the constraint-obeying grid has a dynamic range much greater than that of the experimental data, with the bright transduction signal oversaturating the colour scale (or else, by a downscaling of N, losing the dark features in the noise.), while the constraint-breaking grid gives much better quantitative agreement.

However, even the better of the two grids is not perfect. The main discrepancy is that the bright transduction signals are much broader than in the experimental data. Additionally, the region immediately surrounding $(\omega_\mu, \omega_p) = (\omega_{34}, \omega_{13})$ is darker than its surroundings in the model grid, whereas the opposite is true in the experimental data. What is not problematic, however, is that the features in model grid are translated along the δ_μ axis (horizontal in the plot) compared to their locations in the experimental data; this shift is by an amount smaller than the uncertainty[5] on $A_{\perp e} = \omega_{34}(\mathbf{B} = \mathbf{0}) = 2\pi \times 3.37\,\mathrm{GHz}$ of $\pm 2\pi \times 5\,\mathrm{MHz}$. Indeed, this model and experimental data could be used, in theory, to find a more precise estimate of that spin Hamiltonian parameter.

Given that the experimental parameters relevant to the model are not known with certainty, especially since parameters ostensibly constrained by theory must be modified to obtain good agreement with experiment, these discrepancies between model and experiment do not necessarily indicate deficiencies in the model. Instead, they might simply be the result of input parameters to the model not

[5]Implied by the number of significant figures

matching those used in the experiments that produced the benchmarking data.

3.7.1 Accounting for Constraint Breaking

In this subsection, I offer some possible explanations for why the true parameters of the benchmark experiment might be the constraint-breaking parameters that give good model-experiment agreement, and why they might be different to those calculated in Section 3.6.

Lifetime τ_{12}

The depolarisation lifetime of a dipole transition depends not only on the dipole moment matrix element of the transition, which is a fundamental constant of the emitter, but also on the per-photon electromagnetic field strength, which depends on the environment surrounding the emitter, i.e. whether it is in free space, a waveguide, or a resonator, and the specific geometries of the latter two. Equivalently, and more commonly, this is expressed in terms of the local density of states (LDOS) around the emitter.

Reference [37], which measured $\tau_{12}(\mathbf{B} = \mathbf{0})$, did so in a device different to that with which the experimental data was produced, and so this could potentially be source of this discrepancy.

Hybridisation Ratio

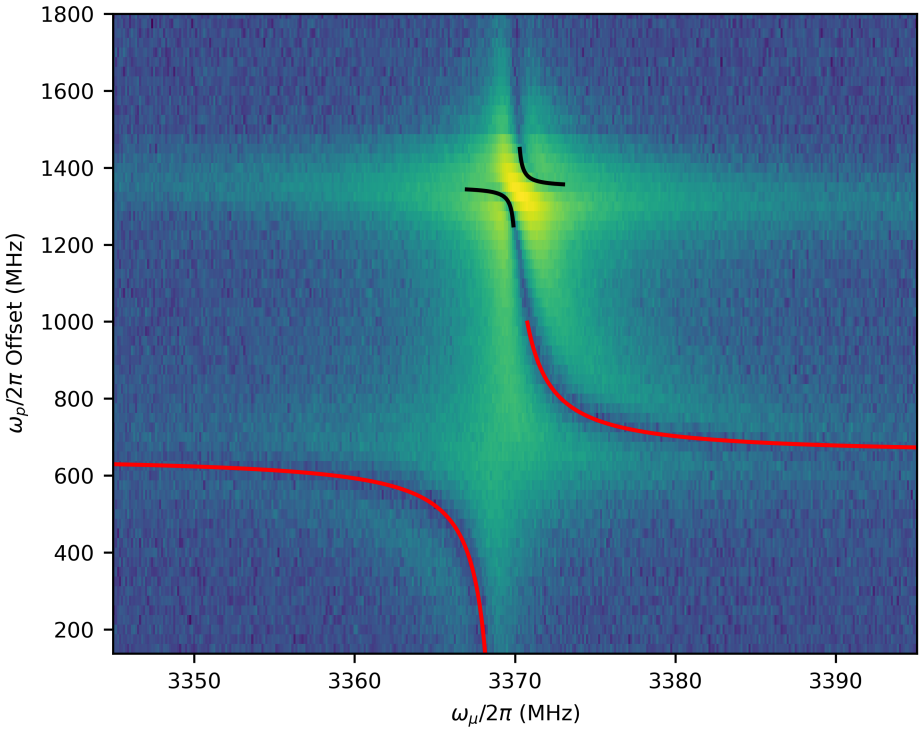


Figure 3.8: Manual axis-aligned hyperbola fits to the dark features surrounding the ω_{13} and ω_{23}transitions. The (red) hyperbola surrounding the ω_{23} (lower) transition has a centre of $(3369.2, 652)$ in the axis coordinates shown and a vertex-to-vertex distance of $2\pi \times 66\,\text{MHz}$. The (black) hyperbola surrounding the ω_{13} (upper) transition has a centre of $(3370.1, 1350)$ and a vertex-to-vertex distance of $2\pi \times 12.6\,\text{MHz}$. The ratio of these vertex-to-vertex distances is 0.19.

The hybridisation ratio of 0.38 was identified using the experimental data. Specifically, the hyperbolic dark features[6] surrounding the transition frequencies are expected from theory to have vertex-to-vertex distances proportional to the Rabi frequencies on the transitions[38]. Measuring these (Figure 3.8)

[6] *Rabi splittings*

yielded a ratio of 0.19. This does not produce a good fit in the model, but its double, 0.38, does; it is unclear why this factor of two is needed.

This leaves the question of why the hybridisation ratio is so large, compared to the value at $B_z = 2.09\,\mathrm{mT}$ of 0.12. One possibility would be that that magnetic field value is wrong, and the magnetic field is in fact much stronger than that. As Figure 3.9 shows, a value of $B_z \approx 8\,\mathrm{mT}$ would produce a hybridisation ratio of 0.38. However, such a strong magnetic field would create Zeeman shifts much larger than those observed in the experimental data, translating every feature by a substantial amount. Therefore, such a magnetic field strength is implausible.

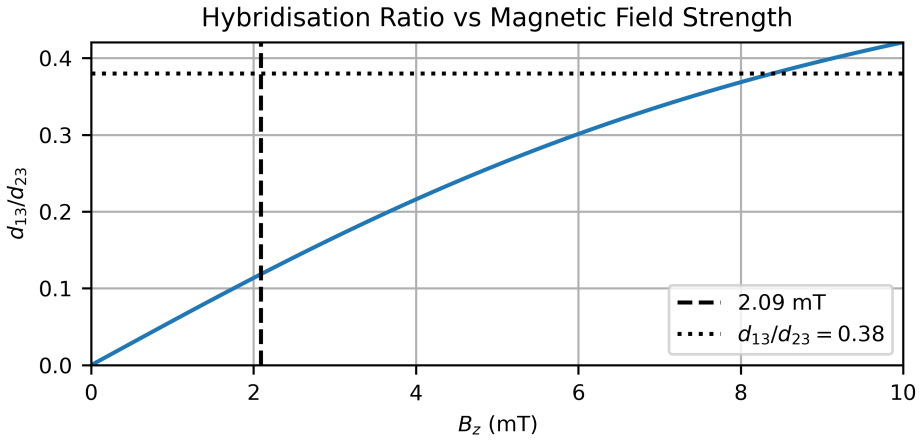


Figure 3.9: A plot of hybridisation ratio vs magnetic field strength B_z. $B_z = 2.09\,\mathrm{mT}$ (dashed vertical line) and hybridisation ratio $||\mathbf{d}_{13}||^2 / ||\mathbf{d}_{23}||^2 = 0.38$ (dotted horizontal line) are superimposed, with the latter intersecting the curve at about $B_z \approx 8\,\mathrm{mT}$.

Chapter 4

Biphoton Generation in 3-Level Systems

This chapter describes original work on modelling the generation rate of photon pairs from biphoton generation in a three-level system, using both dynamical and steady-state models. This happens in the same type of atoms-in-cavity system as in Chapter 2, and this chapter begins by describing how I adapt the three-level transduction model of Reference [2] for biphoton generation. In addition to a steady-state model of the kind in Reference [2], I produce an approximation of the dynamical model which reduces the degrees of freedom to a number tractable to simulate. This chapter then presents results for both the steady-state and dynamical models. Unlike the four-level transduction model of Chapter 3, these results are not benchmarked against any specific experimental data.

4.1 Dynamical Model

Adapting from Equations 2.7, 2.8, 2.15, and 2.18, and using the $\hat{\sigma}_{ij,k} \to \rho_{ji,k}$ semiclassical approximation as per Section 2.5, a semiclassical model for transduction in a three-level system with an ensemble of N atoms is the system of $N+2$ coupled differential equations

$$\frac{d\alpha}{dt} = -i\delta_{co}\alpha - i\sum_{k=1}^{N} g_{o,k}^{*}\rho_{31,k} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\alpha + \sqrt{\gamma_{oc}}\alpha_{\text{in}} \tag{4.1}$$

$$\frac{d\beta}{dt} = -i\delta_{c\mu}\beta - i\sum_{k=1}^{N} g_{\mu,k}^{*}\rho_{j_\mu i_\mu,k} - \frac{\gamma_{\mu i}+\gamma_{\mu c}}{2}\beta + \sqrt{\gamma_{\mu c}}\beta_{\text{in}} \tag{4.2}$$

$$\frac{d\hat{\rho}_k}{dt} = \mathcal{L}_k(\alpha,\beta)\hat{\rho}_k = -i[\hat{H}_{\text{atom},k}(\alpha,\beta),\hat{\rho}_k] + \mathcal{L}_{\text{dec},k}\hat{\rho}_k \tag{4.3}$$

and the semiclassical input-output relations

$$\alpha_{\text{out}} = -\alpha_{\text{in}} + \sqrt{\gamma_{oc}}\alpha \tag{4.4}$$

$$\beta_{\text{out}} = -\beta_{\text{in}} + \sqrt{\gamma_{\mu c}}\beta. \tag{4.5}$$

Here, $|i_\mu\rangle \leftrightarrow |j_\mu\rangle$ is the microwave transition. To adapt this for biphoton generation, I first swap the indices of the optical pump and signal transitions so that $|1\rangle \to |3\rangle$ is being pumped and $|3\rangle \to |2\rangle$ and $|2\rangle \to |1\rangle$ are producing output signals, turning Equation 4.1 into

$$\frac{d\alpha}{dt} = -i\delta_{co}\alpha - i\sum_{k=1}^{N} g_{o,k}^{*}\rho_{j_o i_o,k} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\alpha + \sqrt{\gamma_{oc}}\alpha_{\text{in}} \tag{4.6}$$

where $|j_o\rangle \to |i_o\rangle$ is the optical-emitting transition, and the driven atom Hamiltonian into

$$\hat{H}_{\text{atom},k} = \begin{cases} \begin{bmatrix} 0 & g^*_{\mu,k}\beta^* & \Omega^*_{p,k} \\ g_{\mu,k}\beta & \delta_{\mu,k} & g^*_{o,k}\alpha^* \\ \Omega_{p,k} & g_{o,k}\alpha & \delta_{p,k} \end{bmatrix} & \Lambda\text{-system} \\ \begin{bmatrix} 0 & g^*_{o,k}\alpha^* & \Omega^*_{p,k} \\ g_{o,k}\alpha & \delta_{o,k} & g^*_{\mu,k}\beta^* \\ \Omega_{p,k} & g_{\mu,k}\beta & \delta_{p,k} \end{bmatrix} & \text{V-system} \end{cases}. \tag{4.7}$$

Next, I set the input amplitudes to $\alpha_{\text{in}} = 0 = \beta_{\text{in}}$, because the optical and microwave cavities are used only for output in biphoton generation, not for any input. Note that the input *operators* are not zero because of this, only their expectation values. Furthermore, I set the cavity-signal detunings $\delta_{co} = 0 = \delta_{c\mu}$. This is because there are two output frequencies but only one input (pump) frequency, and so, unlike in transduction, an experimenter cannot arbitrarily control the output frequencies by adjusting the input frequency. Instead, the output frequencies are constrained to be as close to the cavity resonances as possible. Putting these together, the cavity Langevin equations and input-output relations become

$$\frac{d\alpha}{dt} = -i\sum_{k=1}^{N} g^*_{o,k}\rho_{j_o i_o,k} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\alpha \tag{4.8}$$

$$\frac{d\beta}{dt} = -i\sum_{k=1}^{N} g^*_{\mu,k}\rho_{j_\mu i_\mu,k} - \frac{\gamma_{\mu i}+\gamma_{\mu c}}{2}\beta \tag{4.9}$$

$$\alpha_{\text{out}} = \sqrt{\gamma_{oc}}\alpha \tag{4.10}$$

$$\beta_{\text{out}} = \sqrt{\gamma_{\mu c}}\beta. \tag{4.11}$$

4.1.1 Vacuum Rabi Frequency

Suppose that an atom in this system is initially in the ground state so that its density matrix $\hat{\rho} = |1\rangle\langle 1|$, and consider what happens in the model described so far as the optical pump Ω_p is applied. This pump will transfer population from $|1\rangle$ to $|3\rangle$ and generate coherence ($\rho_{13} \neq 0$) between these two levels, so that, in the absence of decoherence, the density matrix becomes

$$\hat{\rho} = \begin{bmatrix} \rho_{11} & 0 & \rho_{13} \\ 0 & 0 & 0 \\ \rho_{31} & 0 & \rho_{33} \end{bmatrix}. \tag{4.12}$$

Now consider the effect of decoherence. Depolarisation will transfer some population into level $|2\rangle$ so that $\rho_{22} \neq 0$, and dephasing will decrease the magnitude of all off-diagonal elements, of which only $\rho_{13} = \rho^*_{31}$ are nonzero. At no point in this process of pumping the atom with losses, then, do ρ_{21} or ρ_{32} become nonzero. This means that there is no emission into the cavities, because those density matrix elements are the ones in the cavity Langevin equations. This is clearly unphysical, and therefore indicates a deficiency in the model described so far.

The problem is that the biphoton generation process is kickstarted, from cavities in vacuum, by interactions between the atoms and fluctuations in the cavity's vacuum field. This model is a mean-field approximation, and therefore does not account for fluctuations in the cavity field. This vacuum interaction can be treated as having an effective Rabi frequency equal to the atom-cavity coupling, $g_{o,k}$ for the optical cavity and $g_{\mu,k}$ for the microwave cavity, known as a *vacuum Rabi frequency*[26]. To capture this in the model, then, I modify the cavity Rabi frequencies from Equation 4.7

$$\Omega_{o,k} = g_{o,k}\alpha \tag{4.13}$$

$$\Omega_{\mu,k} = g_{\mu,k}\beta \tag{4.14}$$

so that, as $\alpha \to 0$, $|\Omega_{o,k}| \to |g_{o,k}|$ (vice-versa for β and $\Omega_{\mu,k}$). Such a modified expression should also be approximately the same as the original for large cavity amplitudes

$$\alpha \gg 1 \implies \Omega_{o,k} \approx g_{o,k}\alpha \tag{4.15}$$

$$\beta \gg 1 \implies \Omega_{\mu,k} \approx g_{\mu,k}\beta \tag{4.16}$$

where stimulated emission, which is a function of the mean field, dominates. If I furthermore require that the phases of the original and modified cavity Rabi frequencies match, then the modified expression should be of the form

$$\Omega_{o,k} = g_{o,k}e^{i\arg\alpha}f(|\alpha|) \tag{4.17}$$

$$\Omega_{\mu,k} = g_{\mu,k}e^{i\arg\beta}f(|\beta|) \tag{4.18}$$

where $f : \mathbb{R}_+ \to \mathbb{R}_+$ is a function for which $f(0) = 1$ and $x \gg 1 \implies f(x) \approx x$. I select $f(x) = \sqrt{x^2+1}$ as such a function, to obtain the modified Rabi frequencies

$$\Omega_{o,k} = g_{o,k}e^{i\arg\alpha}\sqrt{|\alpha|^2+1} \tag{4.19}$$

$$\Omega_{\mu,k} = g_{\mu,k}e^{i\arg\beta}\sqrt{|\beta|^2+1}. \tag{4.20}$$

4.2 Steady States

Taking the approach of Subsection 2.3.1, the steady states of Equations 4.8 and 4.9 can be expressed as a root-finding problem by replacing the dynamical atomic density matrices with steady-state density matrices and replacing the sum over the atoms with an integral over the inhomogeneous distribution, obtaining expressions for the 'residuals' which are zero at steady state

$$\alpha_{\text{res}} = -iN_o g_o^* \iint \rho_{j_o i_o,SS}(\alpha,\beta,\delta_{12},\delta_{23})p(\delta_{12},\delta_{23})\,d\delta_{12}d\delta_{23} - \frac{\gamma_{oi}+\gamma_{oc}}{2}\alpha \tag{4.21}$$

$$\beta_{\text{res}} = -iN_\mu g_\mu^* \iint \rho_{j_\mu i_\mu,SS}(\alpha,\beta,\delta_{12},\delta_{23})p(\delta_{12},\delta_{23})\,d\delta_{12}d\delta_{23} - \frac{\gamma_{\mu i}+\gamma_{\mu c}}{2}\beta. \tag{4.22}$$

Here, the same simplifying assumptions as in Subsection 2.3.1 have been made about the atom-cavity couplings and pump Rabi frequencies, namely that $\Omega_p = \Omega_{p,k}$ is identical across all atoms, that $N_o \leq N$ atoms have the same optical cavity coupling strength $g_o = g_{o,k}$ and the remaining $N - N_o$ atoms do not couple to the optical cavity at all, and that $N_\mu \leq N$ atoms have the same microwave cavity coupling strength $g_\mu = g_{\mu,k}$ with the remaining $N - N_\mu$ atoms not coupling to the microwave cavity at all.

4.3 Super-Atom Dynamics

The $N+2$ coupled differential equations of this model are intractable to solve the dynamics of, because N is very large in realistic systems. To make this tractable, I make the approximation that the N atoms can be partitioned into $n \ll N$ sets with the same atom-cavity couplings and inhomogeneous shifts. Those variables are all that make one atom's dynamics (in terms of density matrices) different from any other, so all atoms within one such set have the same density matrix. Thus, only $n+2$ coupled equations with n density matrices need to be solved. Letting ℓ be an index over these sets and w_ℓ be the number of atoms in set ℓ (of course $\sum_\ell w_\ell = N$), the system of equations becomes

$$\frac{d\alpha}{dt} = -i\sum_{\ell=1}^{n} w_\ell g_{o,\ell}^* \rho_{j_o i_o,\ell} - \frac{\gamma_o}{2}\alpha \tag{4.23}$$

$$\frac{d\beta}{dt} = -i\sum_{\ell=1}^{n} w_\ell g_{\mu,\ell}^* \rho_{j_\mu i_\mu,\ell} - \frac{\gamma_\mu}{2}\beta \tag{4.24}$$

$$\frac{d\hat{\rho}_\ell}{dt} = \mathcal{L}_\ell(\alpha,\beta)\hat{\rho}_\ell \tag{4.25}$$

where $\gamma_o = \gamma_{oi} + \gamma_{oc}$ and $\gamma_\mu = \gamma_{\mu i} + \gamma_{\mu c}$. From these equations, w_ℓ can alternatively be interpreted as scale factors applied to the atom-cavity coupling strengths in the cavity Langevin equations (but not in the atom Master equations) to turn many atoms into a smaller number of 'super-atoms' that interact more strongly with the cavities. Thus, w_ℓ need not necessarily even be integers because, by this interpretation, they are simply weights applied to the super-atoms.

4.3.1 Numerical Methods

A system of ordinary differential equations (ODEs) can be expressed as a single vector ODE $\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x})$, so that numerical methods for ODEs can be applied. The system in Equations 4.23, 4.24, and 4.25 can be expressed in this way with

$$\mathbf{x} = [\operatorname{Re}\alpha, \operatorname{Im}\alpha, \operatorname{Re}\beta, \operatorname{Im}\beta, \rho_{11,1}, \operatorname{Re}\rho_{12,1}, \ldots, \rho_{33,n}]^T \in \mathbb{R}^{9n+4}, \tag{4.26}$$

which contains the four cavity degrees of freedom followed by, for each super-atom, the nine degrees of freedom of its density matrix, as in Equation 2.22. I implemented and used the 4th-order Runge-Kutta method[33] (RK4).

4.4 Results

Both the steady-state and super-atom dynamics models were tested using parameters corresponding to the three-level Λ-system in Er:YSO in Reference [2], which are shown in Table 4.1. For the steady-state model, $N_o = N = N_\mu$ was used, and for the super-atom dynamics model, $n = 1\,000\,000$ super-atoms with equal weight $w_\ell = N/n$ were used. The super-atom simulations were initialised with all super-atoms in the ground state $\hat{\rho}_\ell = |1\rangle_\ell \langle 1|_\ell$ and with small cavity occupancies $\alpha = 1 = \beta$.

Parameter	Value	Parameter	Value
ω_{12}	$2\pi \times 5186\,\text{MHz}$	σ_o	$2\pi \times 419\,\text{MHz}$
τ_{12}	$11\,\text{s}$	σ_μ	$2\pi \times 5\,\text{MHz}$
τ_3	$11\,\text{ms}$	N	1×10^{16}
d_{13}	$1.63 \times 10^{-32}\,\text{C m}$	γ_{oi}	$2\pi \times 7.95\,\text{MHz}$
d_{23}	$1.15 \times 10^{-32}\,\text{C m}$	γ_{oc}	$2\pi \times 1.7\,\text{MHz}$
τ_{13}	$\tau_3 d_{13}^2/(d_{13}^2 + d_{23}^2)$	$\gamma_{\mu i}$	$2\pi \times 650\,\text{kHz}$
τ_{23}	$\tau_3 d_{23}^2/(d_{13}^2 + d_{23}^2)$	$\gamma_{\mu c}$	$2\pi \times 1.5\,\text{MHz}$
T	$4.6\,\text{K}$	g_o	$51.9\,\text{Hz}$
γ_{2d}	$1\,\text{MHz}$	g_μ	$1.04\,\text{Hz}$
γ_{3d}	$1\,\text{MHz}$	Ω_p	$35\,\text{kHz}$

Table 4.1: The parameters common to all runs of both the steady-state and super-atom dynamics models, reproduced from the parameters of the Er:YSO Λ-system in Reference [2] to provide a set of realistic parameters. g_o and g_μ are the same across all atoms that have a nonzero coupling to the respective cavities.

4.4.1 Super-Atom Simulations

Results for the super-atom model are shown in Figure 4.1 for small detunings $\delta_o = -100\,\text{kHz}$ and $\delta_\mu = 1\,\text{MHz}$, and in Figure 4.2 for large detunings $\delta_o = -6.5\sigma_o$ and $\delta_\mu = 8\sigma_\mu$, with two runs each that have identical parameters, including identical inhomogeneous shift samples. The timescales shown, of dozens of microseconds, are much longer than the cavities' sub-microsecond characteristic dynamical timescales (their decay lifetimes), but much shorter than the atomic populations' characteristic dynamical timescales of milliseconds to seconds.

There exists an adiabatic approximation of biphoton generation in a large-detuning regime[12] analogous to the adiabatic model of transduction in Section 2.2. The large detunings were chosen so that they satisfied the requirements of the adiabatic approximation, and the small detunings were

chosen so that they did not. These two sets of detunings therefore corresponded to distinct regimes of behaviour.

Note that these results can largely only be interpreted qualitatively, because the same Rabi frequency can be produced by many different pump powers. Thus, power efficiency, and whether the pump power is constant over time or not, cannot be determined.

Small Detuning

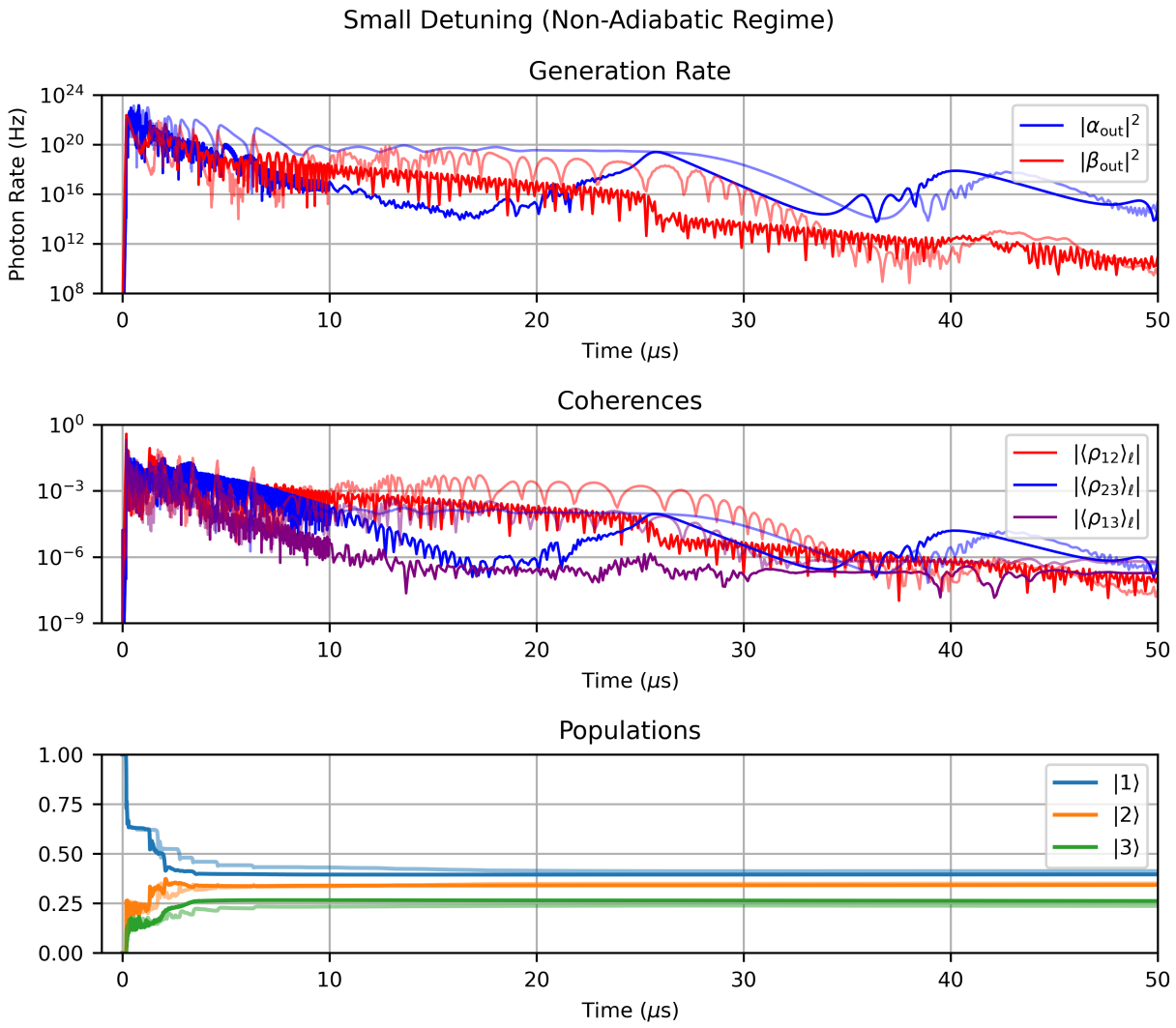


Figure 4.1: Super-atom simulation results for detunings $\delta_o = -100\,\text{kHz}$ and $\delta_\mu = 1\,\text{MHz}$, which are not in the regime where the adiabatic approximation (Section 2.2) holds. Solved using RK4 with time step $\Delta t = 10\,\text{ps}$. Solid and translucent curves are two separate runs with identical parameters (including atom detunings), demonstrating amplification of floating-point errors.

The small-detuning super-atom runs exhibit highly non-convergent behaviour in the cavity dynamics for the entire length of the simulations, which does not visibly suggest that any steady state is being asymptotically approached. Furthermore, the two runs yielded very different dynamics, despite the fact that their parameters, initial conditions, and numerical ODE truncation are all identical, which indicates that floating-point errors[1] were amplified over time. This great divergence of dynam-

[1]Floating-point arithmetic is, in principle, deterministic, so one may wonder why there are different rounding errors in different simulation runs. This is because the super-atom simulations are implemented with the super-atom dynamics running in parallel, and so the sums over super-atom density matrix elements in Equations 4.23 and 4.24 are done in an undefined, variable order. Thus, the nonassociativity of floating point addition leads to slightly different rounding errors each time.

ics from very similar earlier states is characteristic of a chaotic system. Such chaotic behaviour in light-matter systems has also been observed experimentally[39]. Additionally, at some points, there is pulsed, periodic-like behaviour; such behaviour has also been observed in experiments in biphoton generation[40].

Large Detuning

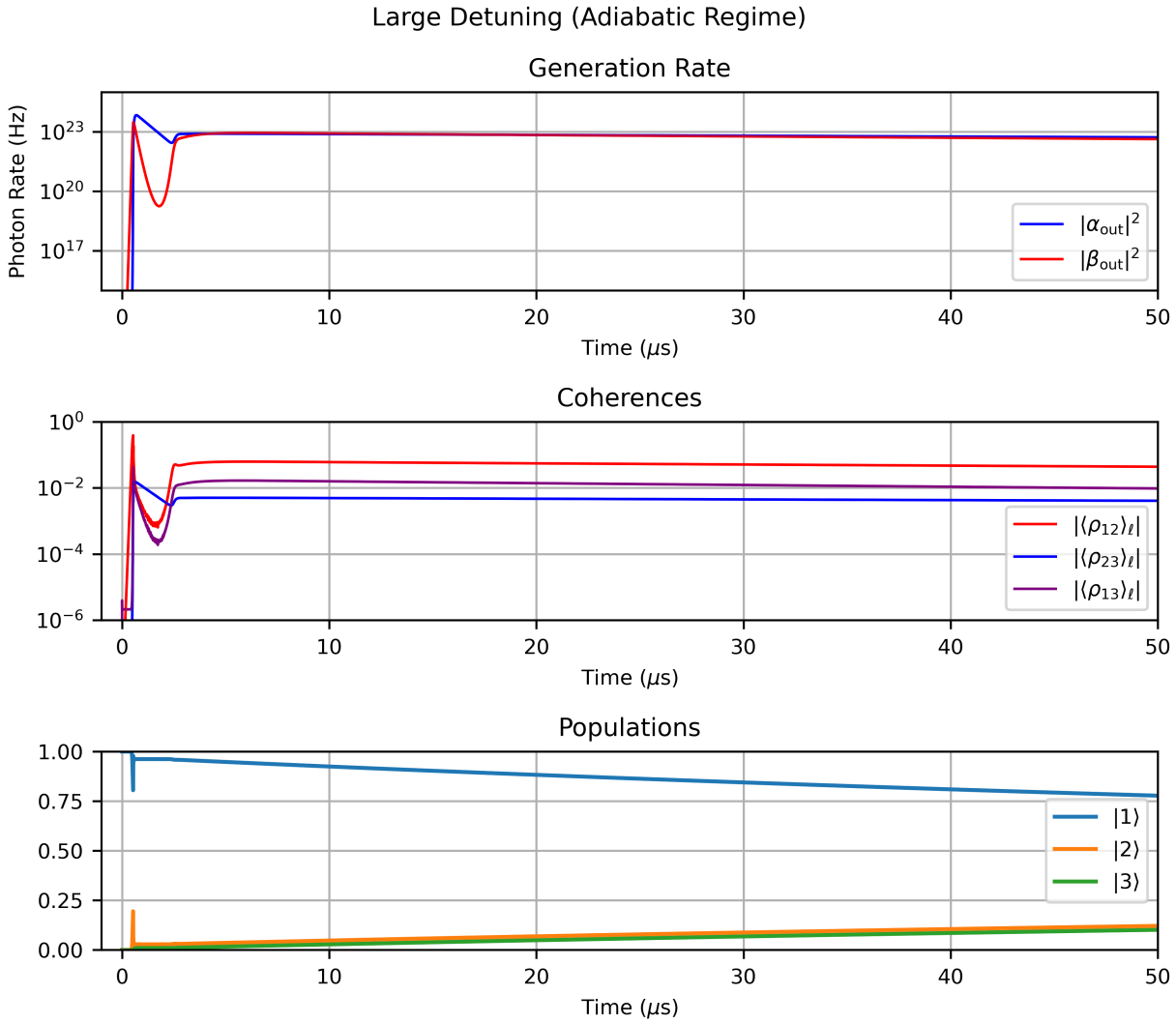


Figure 4.2: Super-atom simulation results for detunings $\delta_o = -6.5\sigma_o$ and $\delta_\mu = 8\sigma_\mu$, which are in the regime where the adiabatic approximation holds. Solved using RK4 with time step $\Delta t = 50\,\text{ps}$. Rendered on the plots but indistinguishable due to overlap are distinct solid and translucent curves corresponding to two separate runs with identical parameters (including atom detunings), demonstrating that floating-point errors do not amplify over time.

By contrast, the large-detuning super-atom runs show quite simple dynamics that are consistent between the two runs, which show that the system is not chaotic for these large detunings. Furthermore, the dynamics are slowing down as time passes, and appear to be asymptotically approaching a steady state. Both of these facts are consistent with the adiabatic approximation holding at these large detunings. However, results[2] from the steady-state model, on the other hand, suggest that this apparent asymptote may not actually be a steady state.

[2]Or lack thereof

Stiffness

All runs of the super-atom model required time steps much smaller than the shortest characteristic timescale $1/\gamma_o \approx 16.5\,\text{ns}$ of the system, otherwise the scales of the system dynamics variables grew unphysically large (e.g. $\operatorname{tr}\hat{\rho}_\ell \gg 1$) until exceeding the floating point limit and polluting every variable with ∞ and NaN. Specifically, $\Delta t = 10\,\text{ps}$ was used for the small detuning and $\Delta t = 50\,\text{ps}$ was used for the large detuning. This result indicates that this ODE problem is stiff.

4.4.2 Steady States

When running the steady-state model, the root-finding for Equations 4.21 and 4.22 does not converge on any nontrivial solution, for both small and large detunings. This is despite the fact that the large-detuning super-atom simulations appear to be converging to a steady state. Indeed, there is no convergence even when using the final cavity amplitudes of those super-atom simulations as an initial guess for the steady state root-finding.

4.5 Implicit Euler Method

Because the ODE problem in Equations 4.23, 4.24, and 4.25 is stiff, an implicit numerical method is more suitable than the explicit RK4 method. I describe here a procedure for implementing such a method, namely the implicit Euler method

$$\mathbf{x}' = \mathbf{x} + \Delta t \mathbf{f}(\mathbf{x}') \tag{4.27}$$

where $\mathbf{x}'$ is the next time step after $\mathbf{x}$. This equation cannot be solved explicitly for $\mathbf{x}'$, and is instead a root-finding problem, which is what is meant by an 'implicit' method. Rearranging Equation 4.27, the root-finding problem is to find the $\mathbf{x}'$ for which the residual

$$\mathbf{x}_{\text{res}} = \mathbf{x} - \mathbf{x}' + \Delta t \mathbf{f}(\mathbf{x}') \tag{4.28}$$

is zero. Breaking this down into α, β, and $\hat{\rho}_\ell$ components and substituting the relevant differential equations for $\mathbf{f}$,

$$\alpha_{\text{res}} = \alpha - \alpha' - i\Delta t \sum_{\ell=1}^{n} w_\ell g^*_{o,\ell} \rho'_{j_o i_o,\ell} - \frac{\gamma_o \Delta t}{2}\alpha' \tag{4.29}$$

$$\beta_{\text{res}} = \beta - \beta' - i\Delta t \sum_{\ell=1}^{n} w_\ell g^*_{\mu,\ell} \rho'_{j_\mu i_\mu,\ell} - \frac{\gamma_\mu \Delta t}{2}\beta' \tag{4.30}$$

$$\hat{\rho}_{\text{res},\ell} = \hat{\rho}_\ell - \hat{\rho}'_\ell + \Delta t \mathcal{L}_\ell(\alpha', \beta')\hat{\rho}'_\ell. \tag{4.31}$$

In Equation 4.31, each ℓ is independent, and so they can be solved for $\hat{\rho}_{\text{res},\ell} = \hat{0}$ to obtain

$$[\Delta t \mathcal{L}_\ell(\alpha', \beta') - \mathbb{1}]\hat{\rho}'_\ell = -\hat{\rho}_\ell. \tag{4.32}$$

Therefore, for a given guess of cavity amplitudes α' and β', guesses of $\hat{\rho}'_\ell$ can be produced so that α_{res} and β_{res} are the only nonzero residuals. This root-finding problem has therefore been reduced from having $9n + 4$ real degrees of freedom to just the four real degrees of freedom of the cavities.

Chapter 5

Conclusion

Hybrid microwave-optical quantum systems have a key role to play in efforts to produce large-scale quantum technology systems, as well as interoperating different quantum technologies. In particular, hybrid microwave-optical transducers and entangled photon pair generators are important tools for producing entanglement across lengths and inside volumes too large to cool to cryogenic temperatures. Atomic systems, in particular using rare-earth atoms in crystals, are an appealing platform with which to build such technologies. However, our understanding of these systems, and therefore our ability to optimise them, is incomplete. I have produced, and presented in this thesis, numerical models of such systems in realistic parameter spaces. These can be used as an aid to improve our understanding of these systems.

I produced a model that computes the output power of atomic ensemble based transduction, that accounts for four atomic energy levels, rather than just three as in prior modelling work. This allows the model to capture the effect of interference between the outputs of two atomic transitions, which can have a large effect on the efficiency of such a transducer. Even 'three-level' transduction schemes often have a fourth level nearby, making interference effects broadly relevant. The formalism can readily be extended to systems with more than four levels and more than two interfering transitions.

This four-level transduction model shows qualitative agreement with experimental data, reproducing the essential features measured in a frequency sweep. There is still some quantitative discrepancy between model results and experimental data, which reflects uncertainty in the experimental parameters. Future work could involve better evaluating the model by either further refining the parameters for a better match with the experimental data used here, or identifying or producing another experimental dataset whose parameters relevant to the model are known with more certainty.

Separately, I produced a model for the photon pair generation rate of an atomic ensemble based biphoton generation process using a three-level system. The dynamical behaviours produced by the model are qualitatively explainable by and consistent with both theory and analogy to experimental results in other light-matter interaction systems. The results also demonstrated that the ODE problem in the model is stiff, which highlights that the results could be improved with a more suitable numerical approach than the RK4 method used. I theorised an efficient implementation of such a method.

Future work on this biphoton generation model could include implementing better numerical methods, as well as performing biphoton generation experiments to produce a dataset with which to validate the model. Separately, future work could be to build from this model to produce a model that predicts not only generation rate, but also the degree of entanglement generated within photon pairs. This is important because the degree of entanglement, not only the rate of photon pair generation, affects the rate at which quantum information can be transmitted. This could be done by replacing the mean-field approximation with a 'Gaussian-field' approximation that incorporates the (co)variances of electromagnetic field operators in addition to expectation values.

In conclusion, the numerical models developed in this project show potential to be useful for understanding microwave-optical transducers and pair generators, thereby aiding the development of such technologies. However, they lack conclusive benchmarks against experimental results, such benchmarking being a prime avenue of future work.

Bibliography

[1] Xavier Fernandez-Gonzalvo et al. "Cavity-enhanced Raman heterodyne spectroscopy in Er^{3+} : Y_2SiO_5 for microwave to optical signal conversion". In: *Phys. Rev. A* 100 (3 Sept. 2019), p. 033807. DOI: 10.1103/PhysRevA.100.033807. URL: https://link.aps.org/doi/10.1103/PhysRevA.100.033807.

[2] Peter S. Barnett and Jevon J. Longdell. "Theory of microwave-optical conversion using rare-earth-ion dopants". In: *Phys. Rev. A* 102 (6 Dec. 2020), p. 063718. DOI: 10.1103/PhysRevA.102.063718. URL: https://link.aps.org/doi/10.1103/PhysRevA.102.063718.

[3] John G. Bartholomew et al. "On-chip coherent microwave-to-optical transduction mediated by ytterbium in YVO4". en. In: *Nature Communications* 11.1 (June 2020), p. 3266. ISSN: 2041-1723. DOI: 10.1038/s41467-020-16996-x. URL: https://www.nature.com/articles/s41467-020-16996-x (visited on 08/08/2023).

[4] David P. DiVincenzo. "The Physical Implementation of Quantum Computation". In: *Fortschritte der Physik* 48.9-11 (2000), pp. 771–783. DOI: https://doi.org/10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E.

[5] Ryan J. MacDonell et al. "Analog quantum simulation of chemical dynamics". In: *Chem. Sci.* 12 (28 2021), pp. 9794–9805. DOI: 10.1039/D1SC02142G. URL: http://dx.doi.org/10.1039/D1SC02142G.

[6] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. "Quantum-Enhanced Measurements: Beating the Standard Quantum Limit". en. In: *Science* 306.5700 (Nov. 2004), pp. 1330–1336. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1104149. URL: https://www.science.org/doi/10.1126/science.1104149 (visited on 12/11/2023).

[7] Charles H. Bennett and Gilles Brassard. "Quantum cryptography: Public key distribution and coin tossing". en. In: *Theoretical Computer Science* 560 (Dec. 2014), pp. 7–11. ISSN: 03043975. DOI: 10.1016/j.tcs.2014.05.025. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397514004241 (visited on 12/11/2023).

[8] Hua-Lei Yin et al. "Measurement-Device-Independent Quantum Key Distribution Over a 404 km Optical Fiber". en. In: *Physical Review Letters* 117.19 (Nov. 2016), p. 190501. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.117.190501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.117.190501 (visited on 07/25/2023).

[9] Wenjamin Rosenfeld et al. "Event-Ready Bell Test Using Entangled Atoms Simultaneously Closing Detection and Locality Loopholes". en. In: *Physical Review Letters* 119.1 (July 2017), p. 010402. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.119.010402. URL: http://link.aps.org/doi/10.1103/PhysRevLett.119.010402 (visited on 12/11/2023).

[10] H.-J. Briegel et al. "Quantum Repeaters: The Role of Imperfect Local Operations in Quantum Communication". In: *Phys. Rev. Lett.* 81 (26 Dec. 1998), pp. 5932–5935. DOI: 10.1103/PhysRevLett.81.5932. URL: https://link.aps.org/doi/10.1103/PhysRevLett.81.5932.

[11] Alfredo Rueda et al. "Efficient microwave to optical photon conversion: an electro-optical realization". In: *Optica* 3.6 (June 2016), pp. 597–604. DOI: 10.1364/OPTICA.3.000597. URL: https://opg.optica.org/optica/abstract.cfm?URI=optica-3-6-597.

[12] Alfredo Rueda et al. "Electro-optic entanglement source for microwave to telecom quantum state transfer". In: *npj Quantum Information* 5.1 (2019), p. 108.

[13] R. Sahu et al. "Entangling microwaves with light". In: *Science* 380.6646 (2023), pp. 718–721. DOI: 10.1126/science.adg3812. eprint: https://www.science.org/doi/pdf/10.1126/science.adg3812. URL: https://www.science.org/doi/abs/10.1126/science.adg3812.

[14] Joerg Bochmann et al. "Nanomechanical coupling between microwave and optical photons". In: *Nature Physics* 9.11 (2013), pp. 712–716.

[15] Andrew P Higginbotham et al. "Harnessing electro-optic correlations in an efficient mechanical converter". In: *Nature Physics* 14.10 (2018), pp. 1038–1042.

[16] Matthias U Staudt et al. "Coupling of an erbium spin ensemble to a superconducting resonator". In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 45.12 (June 2012), p. 124019. DOI: 10.1088/0953-4075/45/12/124019. URL: https://dx.doi.org/10.1088/0953-4075/45/12/124019.

[17] S. Probst et al. "Anisotropic Rare-Earth Spin Ensemble Strongly Coupled to a Superconducting Resonator". In: *Phys. Rev. Lett.* 110 (15 Apr. 2013), p. 157001. DOI: 10.1103/PhysRevLett.110.157001. URL: https://link.aps.org/doi/10.1103/PhysRevLett.110.157001.

[18] Nicholas J. Lambert et al. "Coherent Conversion Between Microwave and Optical Photons—An Overview of Physical Implementations". In: *Advanced Quantum Technologies* 3.1 (2020), p. 1900077. DOI: https://doi.org/10.1002/qute.201900077. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qute.201900077. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qute.201900077.

[19] Nikolai Lauk et al. "Perspectives on quantum transduction". In: *Quantum Science and Technology* 5.2 (Mar. 2020), p. 020501. DOI: 10.1088/2058-9565/ab788a. URL: https://dx.doi.org/10.1088/2058-9565/ab788a.

[20] Miloš Rančić et al. "Coherence time of over a second in a telecom-compatible quantum memory storage material". In: *Nature Physics* 14.1 (2018), pp. 50–54.

[21] Manjin Zhong et al. "Optically addressable nuclear spins in a solid with a six-hour coherence time". In: *Nature* 517.7533 (2015), pp. 177–180.

[22] Thomas Böttger et al. "Effects of magnetic field orientation on optical decoherence in Er^{3+} : Y_2SiO_5". In: *Phys. Rev. B* 79 (11 Mar. 2009), p. 115104. DOI: 10.1103/PhysRevB.79.115104. URL: https://link.aps.org/doi/10.1103/PhysRevB.79.115104.

[23] J. H. Van. Vleck. "The Puzzle of Rare-earth Spectra in Solids." In: *The Journal of Physical Chemistry* 41.1 (1937), pp. 67–80. DOI: 10.1021/j150379a006. eprint: https://doi.org/10.1021/j150379a006. URL: https://doi.org/10.1021/j150379a006.

[24] B.G. Wybourne. *Spectroscopic Properties of Rare Earths*. Interscience Publishers, 1965. ISBN: 9780470965078. URL: https://books.google.com.au/books?id=I91EAAAAIAAJ.

[25] E.T. Jaynes and F.W. Cummings. "Comparison of quantum and semiclassical radiation theories with application to the beam maser". In: *Proceedings of the IEEE* 51.1 (1963), pp. 89–109. DOI: 10.1109/PROC.1963.1664.

[26] Christopher Gerry and Peter Knight. *Introductory Quantum Optics*. Cambridge University Press, 2005. ISBN: 9780521527354.

[27] D.F Walls and Gerard J Milburn. *Quantum Optics*. eng. 2nd ed. Berlin, Heidelberg: Springer Nature, 2008. ISBN: 3540285741.

[28] J. J. (Jun John) Sakurai and Jim Napolitano. *Modern Quantum Mechanics*. eng. Third edition. Cambridge: Cambridge University Press, 2021. ISBN: 9781108587280.

[29] C. W. Gardiner and M. J. Collett. "Input and output in damped quantum systems: Quantum stochastic differential equations and the master equation". In: *Phys. Rev. A* 31 (6 June 1985), pp. 3761–3774. DOI: 10.1103/PhysRevA.31.3761. URL: https://link.aps.org/doi/10.1103/PhysRevA.31.3761.

[30] Heinz-Peter Breuer and Francesco Petruccione. *The Theory of Open Quantum Systems*. Oxford University Press, Jan. 2007. ISBN: 9780199213900. DOI: 10.1093/acprof:oso/9780199213900.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780199213900.001.0001.

[31] Lewis A. Williamson, Yu-Hui Chen, and Jevon J. Longdell. "Magneto-Optic Modulator with Unit Quantum Efficiency". In: *Phys. Rev. Lett.* 113 (20 Nov. 2014), p. 203601. DOI: 10.1103/PhysRevLett.113.203601. URL: https://link.aps.org/doi/10.1103/PhysRevLett.113.203601.

[32] E Brion, L H Pedersen, and K Mølmer. "Adiabatic elimination in a lambda system". In: *Journal of Physics A: Mathematical and Theoretical* 40.5 (Jan. 2007), p. 1033. DOI: 10.1088/1751-8113/40/5/011. URL: https://dx.doi.org/10.1088/1751-8113/40/5/011.

[33] A. C. Faul. *A concise introduction to numerical analysis*. en. Boca Raton: CRC Press, Taylor & Francis Group, 2016. ISBN: 978-1-4987-1218-7.

[34] Peter S. Barnett. "Theory of Microwave to Optical Photon Upconversion Using Erbium Doped Crystals". MA thesis. University of Otago, 2019.

[35] Jonathan M. Kindem et al. "Characterization of Yb 3 + 171 : YVO 4 for photonic quantum technologies". en. In: *Physical Review B* 98.2 (July 2018), p. 024404. ISSN: 2469-9950, 2469-9969. DOI: 10.1103/PhysRevB.98.024404. URL: https://link.aps.org/doi/10.1103/PhysRevB.98.024404 (visited on 09/11/2023).

[36] U. Ranon. "Paramagnetic resonance of Nd3+, Dy3+, Er3+ and Yb3+ in YVO4". In: *Physics Letters A* 28.3 (1968), pp. 228–229. ISSN: 0375-9601. DOI: https://doi.org/10.1016/0375-9601(68)90218-1. URL: https://www.sciencedirect.com/science/article/pii/0375960168902181.

[37] Jonathan M. Kindem et al. "Control and single-shot readout of an ion embedded in a nanophotonic cavity". en. In: *Nature* 580.7802 (Apr. 2020), pp. 201–204. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-2160-9. URL: https://www.nature.com/articles/s41586-020-2160-9 (visited on 09/22/2023).

[38] M. Baur et al. "Measurement of Autler-Townes and Mollow Transitions in a Strongly Driven Superconducting Qubit". In: *Phys. Rev. Lett.* 102 (24 June 2009), p. 243602. DOI: 10.1103/PhysRevLett.102.243602. URL: https://link.aps.org/doi/10.1103/PhysRevLett.102.243602.

[39] Yu-Hui Chen et al. "Optically Unstable Phase from Ion-Ion Interactions in an Erbium-Doped Crystal". In: *Phys. Rev. Lett.* 126 (11 Mar. 2021), p. 110601. DOI: 10.1103/PhysRevLett.126.110601. URL: https://link.aps.org/doi/10.1103/PhysRevLett.126.110601.

[40] Riku Fukumori Tian Xie and Andrei Faraon. personal communication. Jan. 23, 2024.

Appendix A

Replicating and Reverse-Engineering Rabi Frequencies for Barnett and Longdell 2020

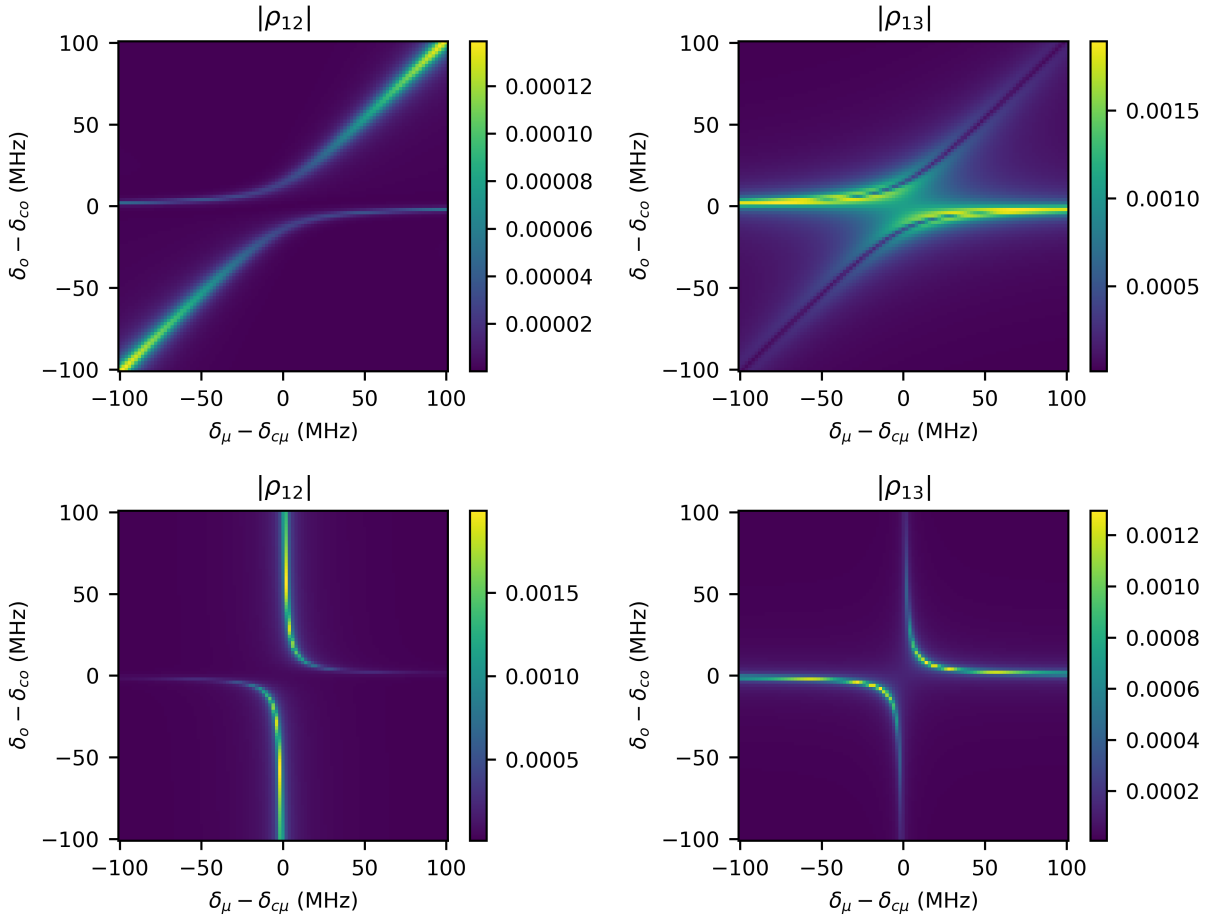


Figure A.1: A replication of Figure 2 from Barnett and Longdell 2020[2], using my own implementation of the model. The top row uses a 1 µW optical pump and 5 dBm microwave drive, and the bottom row uses a 100 mW optical pump and −75 dBm microwave drive.

Barnett and Longdell 2020[2] does not specify pump Rabi frequencies used, but it does give pump powers, and pump powers P_p and Rabi frequencies Ω_p are related by

$$\Omega_p = \frac{\langle 3|\,\hat{d}\,|2\rangle}{\hbar}\sqrt{\frac{2\mu_0 c P_p}{A}} \tag{A.1}$$

where A is the pump laser beam area. When replicating the paper's results, I found that an area corresponding to a 0.1 mm beam diameter gave good results for the high-pump data, and for the low-pump data when using 1 µW instead of the paper's stated 1 pW, reasoning that the latter may have been a typo. This corresponds to Rabi frequencies of approximately 35 kHz and 11 MHz.

Appendix B

Waveguide Transducer Efficiency Fit

This is the data and fit for the input efficiency of the waveguide transduction device that is the subject of Chapter 3, used to calibrate the optical pump power.

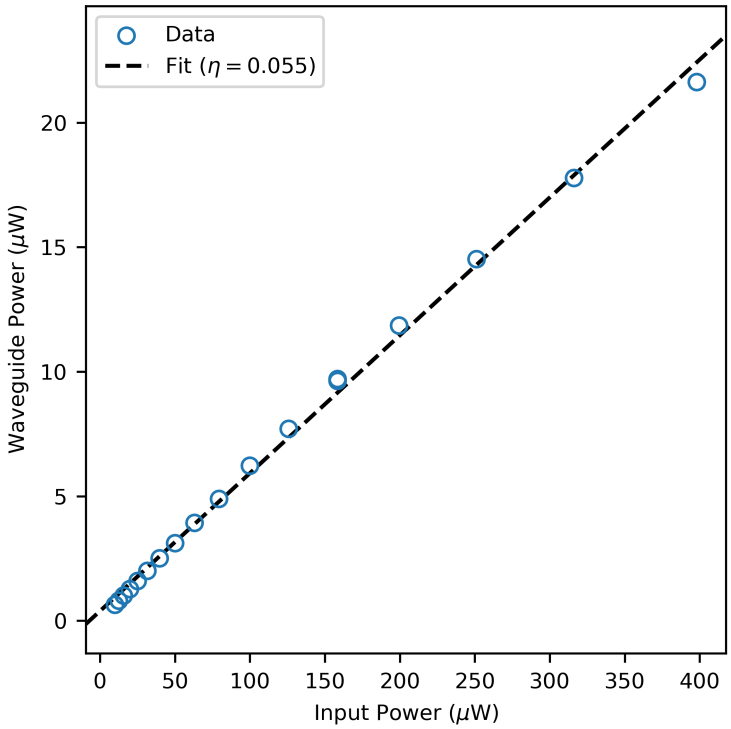


Figure B.1: Plot and curve fit (efficiency $\eta = 0.055$) of waveguide power vs input power. Data in Table B.1

Input Power (dBm)	Waveguide Power (µW)
−8	9.703 703 703 703 702
−4	21.629 629 629 629 626
−5	17.777 777 777 777 78
−6	14.518 518 518 518 519
−7	11.851 851 851 851 851
−8	9.629 629 629 629 63
−9	7.703 703 703 703 703
−10	6.222 222 222 222 222
−11	4.888 888 888 888 888
−12	3.925 925 925 925 925 6
−13	3.111 111 111 111 111
−14	2.503 703 703 703 703 5
−15	2.0
−16	1.592 592 592 592 592 3
−17	1.266 666 666 666 666 6
−18	1.007 407 407 407 407 5
−19	0.8
−20	0.637 037 037 037 037

Table B.1: Waveguide power vs input power.

Appendix C

Code Listings

This appendix contains the 'core' model code. Full code for generating the figures in this thesis, and the data used in those figures, can be found in the GitHub repository for this thesis `https://github.com/Quantum-Integration-Laboratory/MariaNicolaeHonoursThesis`.

C.1 Three-Level Transduction Replication

These codes are my own implementations of the prior three-level transduction models of Chapter 2. They were written quite early in the project, and so follow the notation of the original sources rather than the notation in this thesis.

C.1.1 Single Cavity

This Python code replicates the single-cavity model in Reference [1]. When run as a script, it replicates Figure 4(e) in that paper.

```
import numpy as np
from scipy import integrate, optimize, stats
import matplotlib.pyplot as plt
import sympy

def getL():
    # symbols
    s12 = sympy.Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]])
    s13 = sympy.Matrix([[0, 0, 1], [0, 0, 0], [0, 0, 0]])
    s23 = sympy.Matrix([[0, 0, 0], [0, 0, 1], [0, 0, 0]])
    s21 = s12.H
    s31 = s13.H
    s32 = s23.H
    s11 = s12*s21
    s22 = s21*s12
    s33 = s31*s13

    delta_mu = sympy.symbols('delta_mu', real=True)
    delta_s = sympy.symbols('delta_s', real=True)
    gamma_mu = sympy.symbols('gamma_mu', real=True)
    gamma31, gamma32 = sympy.symbols('gamma3(1:3)', real=True)
    gamma2d, gamma3d = sympy.symbols('gamma(2:4)d', real=True)
    nbath = sympy.symbols('nbath', real=True)
    Omega_mu = sympy.symbols('Omega_mu', complex=True)
    Omega_o = sympy.symbols('Omega_o', complex=True)
    A = sympy.symbols('A', complex=True)

    def master_equation_rhs(rho):
        H = Omega_o*s32 + Omega_mu*s21 + A*s31
        H = H + H.H
        H = H + delta_mu*s22 + delta_s*s33

```

```
33          L21 = gamma_mu/2 * (nbath+1) * (2*s12*rho*s21 - s22*rho - rho*s22)
34          L12 = gamma_mu/2 * nbath * (2*s21*rho*s12 - s11*rho - rho*s11)
35          L32 = gamma32/2 * (2*s23*rho*s32 - s33*rho - rho*s33)
36          L31 = gamma31/2 * (2*s13*rho*s31 - s33*rho - rho*s33)
37          L22 = gamma2d/2 * (2*s22*rho*s22 - s22*s22*rho - rho*s22*s22)
38          L33 = gamma3d/2 * (2*s33*rho*s33 - s33*s33*rho - rho*s33*s33)
39          loss = L21 + L12 + L32 + L31 + L22 + L33
40
41          return -sympy.I*(H*rho - rho*H) + loss
42
43      # obtain matrix representation of differential operator L
44      L = sympy.zeros(9)
45      for i in range(3):
46          for j in range(3):
47              rho = sympy.zeros(3)
48              rho[i,j] = 1
49              Lcol = master_equation_rhs(rho)
50              col = 3*i+j
51              for ip in range(3):
52                  for jp in range(3):
53                      row = 3*ip+jp
54                      L[row,col] = Lcol[ip,jp]
55
56      # replace first row with row computing the trace of rho
57      L[0,:] = sympy.Matrix([1, 0, 0, 0, 1, 0, 0, 0, 1]).T
58
59      # lambdify
60      args = (delta_mu, delta_s, gamma_mu, gamma31, gamma32, gamma2d, gamma3d, nbath,
        Omega_mu, Omega_o, A)
61      Lfunc = sympy.lambdify(args, L, 'numpy')
62
63      return L, Lfunc
64
65  L, Lfunc = getL()
66
67  # fundamental constants
68  hbar = 1.05457e-34
69  kB = 1.380649e-23
70  c = 299792458
71  eps0 = 8.854187817e-12
72  mu0 = 4*np.pi*1e-7
73  muB = 9.274009994e-24
74
75  def rho_steady_state_many_args(delta_mu, delta_s, gamma_mu, gamma31, gamma32,
        gamma2d, gamma3d, nbath, Omega_mu, Omega_o, A):
76      Lmatrix = Lfunc(
77          delta_mu=delta_mu,
78          delta_s=delta_s,
79          gamma_mu=gamma_mu,
80          gamma31=gamma31,
81          gamma32=gamma32,
82          gamma2d=gamma2d,
83          gamma3d=gamma3d,
84          nbath=nbath,
85          Omega_mu=Omega_mu,
86          Omega_o=Omega_o,
87          A=A
88      )
89      b = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0])
90      x = np.linalg.solve(Lmatrix, b)
91      rho = np.array([[x[0], x[1], x[2]],
92                      [x[3], x[4], x[5]],
93                      [x[6], x[7], x[8]]])
94      return rho
95
96  def Omega_mu_from_Pin(Pin, omega_mu):
```

```
97      V_mu_cavity = Vsample / fill_factor
98      Pmu = 1e-3 * 10**(Pin/10)
99      Q = omega_mu / (kappa_mi + 2*kappa_mc)
100     S21 = 4 * kappa_mc**2 / (kappa_mi + 2*kappa_mc)**2
101     energy_mu_cavity = 2*Pmu*Q*np.sqrt(S21) / omega_mu
102     Bmu = np.sqrt(mu0*energy_mu_cavity / (2*V_mu_cavity))
103     return -mu12*Bmu / hbar
104
105 def Omega_o_from_Pin(Pin, omega_o):
106     Po = 1e-3 * 10**(Pin/10)
107     pflux = Po / (hbar*omega_o)
108     n_in = 4*pflux*kappa_oc / (kappa_oc+kappa_oi)**2
109     Sspot = np.pi * Woc**2
110     V_o_cavity = (Sspot*Loc + Sspot*Lsample*nYSO**3) / 2
111     Eo = np.sqrt(n_in*hbar*omega_o / (2*eps0*V_o_cavity))
112     return -d23*Eo / hbar
113
114 def rho_steady_state(Pin_mu, Pin_o, delta_mu, delta_o, a):
115     omega_mu = omega_12 - delta_mu
116     nbath = 1 / (np.exp(hbar*(omega_mu)/(kB*T))-1)
117     rho = rho_steady_state_many_args(
118         delta_mu = delta_mu,
119         delta_s = delta_o - delta_mu,
120         gamma_mu = 1/tau2 * 1/(nbath+1),
121         gamma31 = 1/tau3 * d13**2 / (d13**2 + d23**2),
122         gamma32 = 1/tau3 * d23**2 / (d13**2 + d23**2),
123         gamma2d = 1e6,
124         gamma3d = 1e6,
125         nbath = nbath,
126         Omega_mu = Omega_mu_from_Pin(Pin_mu, omega_12 - delta_mu),
127         Omega_o = Omega_o_from_Pin(Pin_o, omega_23 - delta_o),
128         A = g*a
129     )
130     return rho
131
132 def rho13_steady_state_ensemble(Pin_mu, Pin_o, mean_delta_mu, mean_delta_s, a):
133     standard_norm = lambda z: np.exp(-z**2/2) / np.sqrt(2*np.pi)
134     zrange = 3
135
136     def ensemble_integrand(z_mu, z_s):
137         delta_mu = mean_delta_mu + w12*z_mu
138         delta_s = mean_delta_mu + w13*z_s
139         envelope = standard_norm(z_mu)*standard_norm(z_s) / (w12*w13)
140         rho13 = rho_steady_state(Pin_mu, Pin_o, delta_mu, delta_s, a)[0,2]
141         jacobian = w12 * w13
142         return envelope * rho13 * jacobian
143
144     real_integrand = lambda z_mu, z_s: np.real(ensemble_integrand(z_mu, z_s))
145     imag_integrand = lambda z_mu, z_s: np.imag(ensemble_integrand(z_mu, z_s))
146
147     y_re, abserr_re = integrate.dblquad(real_integrand, -zrange, zrange, -zrange,
        zrange)
148     y_im, abserr_im = integrate.dblquad(imag_integrand, -zrange, zrange, -zrange,
        zrange)
149     y = y_re + 1j*y_im
150     return y
151
152 def steady_a(Pin_mu, Pin_o, delta_oc, rescaling=1):
153     def S13(a):
154         return N*g*rho13_steady_state_ensemble(Pin_mu, Pin_o, 0, 0, a)
155
156     def ffunc(a):
157         return -1j*delta_oc*a - 1j*S13(a) - (kappa_oi+kappa_oc)*a/2
158
159     def ffunc_R2toR2(a):
160         [a_re, a_im] = a
```

```
        a = a_re + 1j*a_im
        fa = ffunc(a)
        fa *= rescaling
        return [np.real(fa), np.imag(fa)]

    result = optimize.root(ffunc_R2toR2, [0, 0])
    [a_re, a_im] = result.x
    return a_re + 1j*a_im, result

if __name__ == '__main__':
    # recreate Figure 4c from Fernandez-Gonzalvo et. al. 2019
    # (Phys. Rev. A 100, 033807)

    nYSO = 1.76 # refractive index of YSO
    T = 4.6 # experiment temperature
    N = 1.28e15 # erbium number density
    g = 51.9 # s13 to optical coupling

    omega_12 = 2*np.pi*5.186e9
    omega_23 = 2*np.pi*195113.36e9 # 1536.504 nm
    w12 = 2*np.pi*25e6
    w13 = 2*np.pi*170e6
    mu12 = 4.3803*muB
    d13 = 1.63e-32
    d23 = 1.15e-32
    tau2 = 1e-3
    tau3 = 11e-3

    kappa_mi = 2*np.pi*650e3
    kappa_mc = 2*np.pi*70e3
    kappa_oi = 2*np.pi*1.7e6
    kappa_oc = 2*np.pi*7.95e6

    Woc = 0.6e-3
    Loc = 49.5e-3
    fill_factor = 0.8 # fraction of microwave cavity filled by sample

    dsample = 5e-3
    Lsample = 12e-3
    Vsample = np.pi * dsample**2 * Lsample / 4

    Pin_mu = 0
    Pin_o = 10.4135

    a, _ = steady_a(-20, Pin_o, 0, rescaling=1e-6)

    delta_mu_v = 30e6 * np.linspace(-2*np.pi, 2*np.pi, 101) / (2*np.pi)
    delta_o_v = 50e6 * np.linspace(-2*np.pi, 2*np.pi, 101) / (2*np.pi)
    delta_mu, delta_o = np.meshgrid(delta_mu_v, delta_o_v)

    def steady_pop(Pin_mu, Pin_o, delta_mu, delta_o, a):
        rho = rho_steady_state(Pin_mu, Pin_o, delta_mu, delta_o, a)
        return np.real(np.diag(rho))

    sig = '(),(),(),()->(n)'
    pop = np.vectorize(steady_pop, signature=sig)(-20, Pin_o, delta_mu, delta_o, a)

    p11 = pop[:,:,0]
    p33 = pop[:,:,2]
    mhz = 1e6

    fig, ax = plt.subplots(1, 1)
    ax.pcolormesh(delta_o/mhz, delta_mu/mhz, p11-p33, vmin=0, vmax=0.05)
    ax.invert_yaxis()
    ax.set_aspect('equal')
    ax.set_title('rho11 - rho33 (Mine)')
```

```
    ax.set_xlabel('delta_o (rad MHz)')
    ax.set_ylabel('delta_mu (rad MHz)')
    plt.show()
```

C.1.2 Double Cavity

This Python code replicates the double-cavity model in Reference [2]. It is 'library' code that is not a script in its own right. The GitHub repository contains scripts that `import` this code.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate, optimize, stats
import sympy
import pickle
import itertools

gauss_lobatto_n = 20

legendre_coeffs = (0,)*(gauss_lobatto_n-1) + (1,)
roots = np.polynomial.legendre.Legendre(legendre_coeffs).deriv().roots()
gauss_lobatto_points = np.concatenate([[-1], roots, [1]])
gauss_lobatto_points = (gauss_lobatto_points+1) / 2

k = np.arange(gauss_lobatto_n)
i, j = np.indices((gauss_lobatto_n, gauss_lobatto_n))
M = gauss_lobatto_points[j]**i
b = 1/(k+1)
gauss_lobatto_weights = np.linalg.solve(M, b)

# flattening and unflattening arrays
unravel = np.array([
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8]
])

ravel_i = np.zeros(9, dtype=int)
ravel_j = np.zeros(9, dtype=int)
for i in range(3):
    for j in range(3):
        k = unravel[i,j]
        ravel_i[k] = i
        ravel_j[k] = j
ravel = (ravel_i, ravel_j)

def get_symbolic():
    # parameters
    g_o = sympy.symbols('g_o', real=True)
    g_mu = sympy.symbols('g_mu', real=True)
    Omega = sympy.symbols('Omega')
    alpha = sympy.symbols('alpha')
    beta = sympy.symbols('beta')
    delta_mu = sympy.symbols('delta_mu', real=True)
    delta_o = sympy.symbols('delta_o', real=True)
    delta_amu = sympy.symbols('delta_amu', real=True)
    delta_ao = sympy.symbols('delta_ao', real=True)
    gamma_12 = sympy.symbols('gamma_12', real=True)
    gamma_13 = sympy.symbols('gamma_13', real=True)
    gamma_23 = sympy.symbols('gamma_23', real=True)
    gamma_2d = sympy.symbols('gamma_2d', real=True)
    gamma_3d = sympy.symbols('gamma_3d', real=True)
    n_bath = sympy.symbols('n_bath', real=True)
    symbols = {
        'g_o': g_o,
        'g_mu': g_mu,
```

```
        'Omega': Omega,
        'alpha': alpha,
        'beta': beta,
        'delta_mu': delta_mu,
        'delta_o': delta_o,
        'delta_amu': delta_amu,
        'delta_ao': delta_ao,
        'gamma_12': gamma_12,
        'gamma_13': gamma_13,
        'gamma_23': gamma_23,
        'gamma_2d': gamma_2d,
        'gamma_3d': gamma_3d,
        'n_bath': n_bath
    }

    # matrix symbols
    s12 = sympy.Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]])
    s13 = sympy.Matrix([[0, 0, 1], [0, 0, 0], [0, 0, 0]])
    s23 = sympy.Matrix([[0, 0, 0], [0, 0, 1], [0, 0, 0]])
    s21 = s12.H
    s31 = s13.H
    s32 = s23.H
    s11 = s12*s21
    s22 = s21*s12
    s33 = s31*s13

    # Hamiltonian
    H_0 = (delta_amu-delta_mu)*s22 + (delta_ao-delta_o)*s33 + Omega*s32 +
    Omega.conjugate()*s23
    H_alpha = g_o*s31
    H_alpha_c = g_o*s13
    H_beta = g_mu*s21
    H_beta_c = g_mu*s12
    H = H_0 + alpha*H_alpha + alpha.conjugate()*H_alpha_c + beta*H_beta +
    beta.conjugate()*H_beta_c

    # loss superoperator
    def loss_superoperator(rho):
        L21 = gamma_12/2 * (n_bath+1) * (2*s12*rho*s21 - s22*rho - rho*s22)
        L12 = gamma_12/2 * n_bath * (2*s21*rho*s12 - s11*rho - rho*s11)
        L32 = gamma_23/2 * (2*s23*rho*s32 - s33*rho - rho*s33)
        L31 = gamma_13/2 * (2*s13*rho*s31 - s33*rho - rho*s33)
        L22 = gamma_2d/2 * (2*s22*rho*s22 - s22*rho - rho*s22)
        L33 = gamma_3d/2 * (2*s33*rho*s33 - s33*rho - rho*s33)
        loss = L21 + L12 + L32 + L31 + L22 + L33
        return loss

    # get liouvillan superoperator matrix
    def liouvillan_matrix(H, loss=None):
        def master_equation(rho):
            master_operator = -sympy.I * (H*rho-rho*H)
            if loss is not None:
                master_operator += loss(rho)
            return master_operator

        L = sympy.zeros(9)
        for column, (icol, jcol) in enumerate(zip(*ravel)):
            # basis matrix
            rho = sympy.zeros(3)
            rho[icol,jcol] = 1

            # get column of supermatrix
            L_column = master_equation(rho)
            for row, (irow, jrow) in enumerate(zip(*ravel)):
                L[row,column] = L_column[irow,jrow]

```

```
        return L

    L_0 = liouvillan_matrix(H_0, loss=loss_superoperator)
    L_alpha = liouvillan_matrix(H_alpha)
    L_alpha_c = liouvillan_matrix(H_alpha_c)
    L_beta = liouvillan_matrix(H_beta)
    L_beta_c = liouvillan_matrix(H_beta_c)
    L = L_0 + alpha*L_alpha + alpha.conjugate()*L_alpha_c + beta*L_beta +
    beta.conjugate()*L_beta_c

    return {
        'symbols': symbols,
        'H': H,
        'L': L,
        'L_0': L_0,
        'L_alpha': L_alpha,
        'L_alpha_c': L_alpha_c,
        'L_beta': L_beta,
        'L_beta_c': L_beta_c
    }

symbolic = get_symbolic()

symbolic_args_full = (
    symbolic['symbols']['g_o'],
    symbolic['symbols']['g_mu'],
    symbolic['symbols']['Omega'],
    symbolic['symbols']['gamma_12'],
    symbolic['symbols']['gamma_13'],
    symbolic['symbols']['gamma_23'],
    symbolic['symbols']['gamma_2d'],
    symbolic['symbols']['gamma_3d'],
    symbolic['symbols']['n_bath'],
    symbolic['symbols']['delta_mu'],
    symbolic['symbols']['delta_o'],
    symbolic['symbols']['delta_amu'],
    symbolic['symbols']['delta_ao'],
    symbolic['symbols']['alpha'],
    symbolic['symbols']['beta']
)

symbolic_args_0 = (
    symbolic['symbols']['g_o'],
    symbolic['symbols']['g_mu'],
    symbolic['symbols']['Omega'],
    symbolic['symbols']['gamma_12'],
    symbolic['symbols']['gamma_13'],
    symbolic['symbols']['gamma_23'],
    symbolic['symbols']['gamma_2d'],
    symbolic['symbols']['gamma_3d'],
    symbolic['symbols']['n_bath'],
    symbolic['symbols']['delta_mu'],
    symbolic['symbols']['delta_o'],
    symbolic['symbols']['delta_amu'],
    symbolic['symbols']['delta_ao']
)

symbolic_args_H = (
    symbolic['symbols']['g_o'],
    symbolic['symbols']['g_mu'],
    symbolic['symbols']['Omega'],
    symbolic['symbols']['delta_mu'],
    symbolic['symbols']['delta_o'],
    symbolic['symbols']['delta_amu'],
    symbolic['symbols']['delta_ao'],
    symbolic['symbols']['alpha'],
```

```
    symbolic['symbols']['beta']
)

H = sympy.lambdify(symbolic_args_H, symbolic['H'], 'numpy')
L = sympy.lambdify(symbolic_args_full, symbolic['L'], 'numpy')
L_0 = sympy.lambdify(symbolic_args_0, symbolic['L_0'], 'numpy')
L_alpha = sympy.lambdify(symbolic['symbols']['g_o'], symbolic['L_alpha'], 'numpy')
L_alpha_c = sympy.lambdify(symbolic['symbols']['g_o'], symbolic['L_alpha_c'],
    'numpy')
L_beta = sympy.lambdify(symbolic['symbols']['g_mu'], symbolic['L_beta'], 'numpy')
L_beta_c = sympy.lambdify(symbolic['symbols']['g_mu'], symbolic['L_beta_c'], 'numpy')

def rho_steady_state(g_o, g_mu, Omega, gamma_12, gamma_13,
                     gamma_23, gamma_2d, gamma_3d, n_bath,
                     delta_mu, delta_o, delta_amu, delta_ao, alpha, beta):
    L_matrix = L(
        g_o=g_o,
        g_mu=g_mu,
        Omega=Omega,
        gamma_12=gamma_12,
        gamma_13=gamma_13,
        gamma_23=gamma_23,
        gamma_2d=gamma_2d,
        gamma_3d=gamma_3d,
        n_bath=n_bath,
        delta_mu=delta_mu,
        delta_o=delta_o,
        delta_amu=delta_amu,
        delta_ao=delta_ao,
        alpha=alpha,
        beta=beta
    )
    L_matrix[0,:] = np.array([1, 0, 0, 0, 1, 0, 0, 0, 1])
    b = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0])
    rho = np.linalg.solve(L_matrix, b)
    return rho[unravel]

def rho_steady_state_ensemble(g_o, g_mu, Omega, gamma_12, gamma_13,
                              gamma_23, gamma_2d, gamma_3d, n_bath,
                              delta_mu, delta_o, delta_amu, delta_ao,
                              sigma_mu, sigma_o, alpha, beta):
    # degenerate dressed state detuning
    def minor_det(M, i, j):
            return M[i,i]*M[j,j] - M[i,j]*M[j,i]

    def H_disc(delta_amu, delta_ao):
        H_matrix = H(
            g_mu=g_mu,
            g_o=g_o,
            Omega=Omega,
            alpha=alpha,
            beta=beta,
            delta_mu=delta_mu,
            delta_o=delta_o,
            delta_amu=delta_amu,
            delta_ao=delta_ao
        )
        a = -1
        b = np.trace(H_matrix)
        c = -(minor_det(H_matrix, 0, 1)
            + minor_det(H_matrix, 0, 2)
            + minor_det(H_matrix, 1, 2))
        d = np.linalg.det(H_matrix)
        Delta = 18*a*b*c*d - 4*b**3*d + b*b*c*c - 4*a*c**3 - 27*a*a*d*d
        return np.abs(Delta)

```

```
    def delta_amu_degenerate(delta_ao):
        # get close guess
        if delta_ao == delta_o:
            delta_amu0 = delta_mu
        elif np.abs(beta) < np.abs(Omega):
            delta_amu0 = -np.abs(Omega)**2/(delta_ao-delta_o) + delta_ao - delta_o +
    delta_mu
        else:
            delta_amu0 = np.abs(g_mu*beta)**2/(delta_ao-delta_o) + delta_mu

        f = lambda d_amu: H_disc(d_amu, delta_ao)
        result = optimize.minimize_scalar(f, bracket=(delta_amu0-sigma_mu,
    delta_amu0+sigma_mu))
        return result.x

    # gauss-lobatto integration
    def gauss_lobatto_nodes_weights(a, b):
        nodes = a + gauss_lobatto_points*(b-a)
        weights = gauss_lobatto_weights*(b-a)
        return nodes, weights

    def composite_gauss_lobatto_nodes_weights(bounds):
        bounds = np.sort(bounds)
        nodes = np.array([], dtype=float)
        weights = np.array([], dtype=float)
        for a, b in zip(bounds[:-1], bounds[1:]):
            new_nodes, new_weights = gauss_lobatto_nodes_weights(a, b)
            nodes = np.concatenate([nodes, new_nodes])
            weights = np.concatenate([weights, new_weights])
        return nodes, weights

    # rho helper
    def rho_fewer_args(d_ao, d_amu):
        return rho_steady_state(
            g_o=g_o,
            g_mu=g_mu,
            Omega=Omega,
            gamma_12=gamma_12,
            gamma_13=gamma_13,
            gamma_23=gamma_23,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            n_bath=n_bath,
            delta_mu=delta_mu,
            delta_o=delta_o,
            delta_amu=d_amu,
            delta_ao=d_ao,
            alpha=alpha,
            beta=beta
        )

    # generate outer integral nodes and weights
    bounds = [
        delta_ao,
        delta_ao - sigma_o,
        delta_ao + sigma_o,
        delta_ao - 3*sigma_o,
        delta_ao + 3*sigma_o,
        delta_ao - 10*sigma_o,
        delta_ao + 10*sigma_o,
        delta_o,
        delta_o - gamma_3d,
        delta_o + gamma_3d,
        delta_o - 5*gamma_3d,
        delta_o + 5*gamma_3d
    ]
```

```
    d_ao_nodes, d_ao_weights = composite_gauss_lobatto_nodes_weights(bounds)

    # outer integral envelope function
    z_ao_nodes = (d_ao_nodes-delta_ao) / sigma_o
    G_ao_nodes = stats.norm.pdf(z_ao_nodes) / sigma_o

    # perform integrals
    rho_integral = np.zeros((3, 3), dtype=complex)
    for d_ao, d_ao_w, G_ao in zip(d_ao_nodes, d_ao_weights, G_ao_nodes):
        # generate inner integral nodes and weights
        bounds = [
            delta_amu,
            delta_amu - sigma_mu,
            delta_amu + sigma_mu,
            delta_amu - 3*sigma_mu,
            delta_amu + 3*sigma_mu,
            delta_amu - 10*sigma_mu,
            delta_amu + 10*sigma_mu,
            delta_mu,
            delta_mu - gamma_2d,
            delta_mu + gamma_2d,
            delta_mu - 5*gamma_2d,
            delta_mu + 5*gamma_2d,
            delta_amu_degenerate(d_ao)
        ]
        d_amu_nodes, d_amu_weights = composite_gauss_lobatto_nodes_weights(bounds)

        # inner integral envelope function
        z_amu_nodes = (d_amu_nodes-delta_amu) / sigma_mu
        G_amu_nodes = stats.norm.pdf(z_amu_nodes) / sigma_mu

        # perform inner integral
        for d_amu, d_amu_w, G_amu in zip(d_amu_nodes, d_amu_weights, G_amu_nodes):
            rho = rho_fewer_args(d_ao, d_amu)
            rho_integral += G_ao * G_amu * rho * d_ao_w * d_amu_w

    return rho_integral

def alpha_beta_langevin_differential(N_o, N_mu, g_o, g_mu, Omega, gamma_12, gamma_13,
                                     gamma_23, gamma_2d, gamma_3d, gamma_muc,
    gamma_mui,
                                     gamma_oc, gamma_oi, n_bath, delta_mu, delta_o,
    delta_amu,
                                     delta_ao, sigma_mu, sigma_o, alpha, beta,
    alpha_in, beta_in,
                                     use_rho_ji=True):
    rho_steady = rho_steady_state_ensemble(
        g_o=g_o,
        g_mu=g_mu,
        Omega=Omega,
        gamma_12=gamma_12,
        gamma_13=gamma_13,
        gamma_23=gamma_23,
        gamma_2d=gamma_2d,
        gamma_3d=gamma_3d,
        n_bath=n_bath,
        delta_mu=delta_mu,
        delta_o=delta_o,
        delta_amu=delta_amu,
        delta_ao=delta_ao,
        sigma_mu=sigma_mu,
        sigma_o=sigma_o,
        alpha=alpha,
        beta=beta
    )
    S12 = N_mu * g_mu * (rho_steady[1,0] if use_rho_ji else rho_steady[0,1])
```

```
    S13 = N_o * g_o * (rho_steady[2,0] if use_rho_ji else rho_steady[0,2])
    alpha_diff = 1j*delta_o*alpha - 1j*S13 - (gamma_oc+gamma_oi)*alpha/2 +
    np.sqrt(gamma_oc)*alpha_in
    beta_diff = 1j*delta_mu*beta - 1j*S12 - (gamma_muc+gamma_mui)*beta/2 +
    np.sqrt(gamma_muc)*beta_in
    return alpha_diff, beta_diff

def alpha_beta_steady_state(N_o, N_mu, g_o, g_mu, Omega, gamma_12, gamma_13,
                            gamma_23, gamma_2d, gamma_3d, gamma_muc, gamma_mui,
                            gamma_oc, gamma_oi, n_bath, delta_mu, delta_o, delta_amu,
                            delta_ao, sigma_mu, sigma_o, alpha_in, beta_in,
    use_rho_ji=True):
    def root_function(alpha_beta_vec):
        alpha_r = alpha_beta_vec[0]
        alpha_i = alpha_beta_vec[1]
        beta_r = alpha_beta_vec[2]
        beta_i = alpha_beta_vec[3]
        alpha = alpha_r + 1j*alpha_i
        beta = beta_r + 1j*beta_i
        alpha_res, beta_res = alpha_beta_langevin_differential(
            N_o=N_o,
            N_mu=N_mu,
            g_o=g_o,
            g_mu=g_mu,
            Omega=Omega,
            gamma_12=gamma_12,
            gamma_13=gamma_13,
            gamma_23=gamma_23,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            gamma_muc=gamma_muc,
            gamma_mui=gamma_mui,
            gamma_oc=gamma_oc,
            gamma_oi=gamma_oi,
            n_bath=n_bath,
            delta_mu=delta_mu,
            delta_o=delta_o,
            delta_amu=delta_amu,
            delta_ao=delta_ao,
            sigma_mu=sigma_mu,
            sigma_o=sigma_o,
            alpha=alpha,
            beta=beta,
            alpha_in=alpha_in,
            beta_in=beta_in,
            use_rho_ji=use_rho_ji
        )
        alpha_beta_res_vec = np.zeros(4, dtype=float)
        alpha_beta_res_vec[0] = np.real(alpha_res)
        alpha_beta_res_vec[1] = np.imag(alpha_res)
        alpha_beta_res_vec[2] = np.real(beta_res)
        alpha_beta_res_vec[3] = np.imag(beta_res)
        return alpha_beta_res_vec

    # initial guess for root-finding; root for zero atomic interaction
    alpha_0 = alpha_in*np.sqrt(gamma_oc) / ((gamma_oi+gamma_oc)/2 - 1j*delta_o)
    beta_0 = beta_in*np.sqrt(gamma_muc) / ((gamma_mui+gamma_muc)/2 - 1j*delta_mu)

    # perform root finding
    x0 = np.zeros(4, dtype=float)
    x0[0] = np.real(alpha_0)
    x0[1] = np.imag(alpha_0)
    x0[2] = np.real(beta_0)
    x0[3] = np.imag(beta_0)
    result = optimize.root(root_function, x0=x0, tol=1e-12)

```

```
    # restore result to complex numbers
    alpha_r = result.x[0]
    alpha_i = result.x[1]
    beta_r = result.x[2]
    beta_i = result.x[3]
    alpha = alpha_r + 1j*alpha_i
    beta = beta_r + 1j*beta_i

    return (alpha, beta), result
```

C.2 Four-Level Transduction

This Python code implements the four-level transduction model in Chapter 3. When imported into a Python script, it expects to be able to save and load a file into a directory called `result-cache`. When run as a script, it does the aforementioned saving and loading and nothing else. The notation in this code differs from that in this thesis, mainly in that the 23, 13, 24, and 14 transitions are labelled A, B, D, and E respectively. Additionally, δ_p is instead called δ_B.

```
import numpy as np
from scipy import signal
import sympy
import pickle
import os

# fundamental constants

hbar = 1.054571817e-34
kB = 1.380649e-23

# flattening and unflattening of density matrices

unflatten = np.array([
    [ 0,  1,  2,  3],
    [ 4,  5,  6,  7],
    [ 8,  9, 10, 11],
    [12, 13, 14, 15]
])

flatten_i = np.zeros(16, dtype=int)
flatten_j = np.zeros(16, dtype=int)
for i in range(4):
    for j in range(4):
        k = unflatten[i,j]
        flatten_i[k] = i
        flatten_j[k] = j
flatten = (flatten_i, flatten_j)

# computer algebra for Liouvillan matrix

s11 = sympy.Matrix([[1,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
s12 = sympy.Matrix([[0,1,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
s13 = sympy.Matrix([[0,0,1,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
s14 = sympy.Matrix([[0,0,0,1],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
s21 = s12.T
s31 = s13.T
s41 = s14.T
s22 = s21*s12
s23 = s21*s13
s24 = s21*s14
s32 = s23.T
s42 = s24.T
s33 = s31*s13
s34 = s31*s14
```

```
s43 = s34.T
s44 = s43*s34

omega12 = sympy.symbols('omega_12', real=True)
delta_B = sympy.symbols('delta_B', real=True)
delta_mu = sympy.symbols('delta_mu', real=True)
Omega_mu = sympy.symbols('Omega_mu')
Omega_A = sympy.symbols('Omega_A')
Omega_B = sympy.symbols('Omega_B')
Omega_D = sympy.symbols('Omega_D')
Omega_E = sympy.symbols('Omega_E')

gamma12, gamma13, gamma14 = sympy.symbols('gamma_1(2:5)', real=True)
gamma23, gamma24 = sympy.symbols('gamma_2(3:5)', real=True)
gamma34 = sympy.symbols('gamma_34', real=True)
gamma2d, gamma3d, gamma4d = sympy.symbols('gamma_(2:5)d', real=True)
nbath_12 = sympy.symbols('n_12', real=True)
nbath_34 = sympy.symbols('n_34', real=True)

H = Omega_A*s32 + Omega_B*s31 + Omega_D*s42 + Omega_E*s41 + Omega_mu*s43
H += H.H
H += omega12*s22 + delta_B*s33 + (delta_B+delta_mu)*s44

def loss_superoperator(rho):
    L12 = gamma12*(nbath_12+1)/2 * (2*s12*rho*s21 - s22*rho - rho*s22)
    L21 = gamma12*nbath_12/2 * (2*s21*rho*s12 - s11*rho - rho*s11)
    L13 = gamma13/2 * (2*s13*rho*s31 - s33*rho - rho*s33)
    L14 = gamma14/2 * (2*s14*rho*s41 - s44*rho - rho*s44)
    L23 = gamma23/2 * (2*s23*rho*s32 - s33*rho - rho*s33)
    L24 = gamma24/2 * (2*s24*rho*s42 - s44*rho - rho*s44)
    L34 = gamma34*(nbath_34+1)/2 * (2*s34*rho*s43 - s44*rho - rho*s44)
    L43 = gamma34*nbath_34/2 * (2*s43*rho*s34 - s33*rho - rho*s33)
    L2d = gamma2d/2 * (2*s22*rho*s22 - s22*rho - rho*s22)
    L3d = gamma3d/2 * (2*s33*rho*s33 - s33*rho - rho*s33)
    L4d = gamma4d/2 * (2*s44*rho*s44 - s44*rho - rho*s44)
    return L12 + L21 + L13 + L14 + L23 + L24 + L34 + L43 + L2d + L3d + L4d

def liouvillan_superoperator(rho):
    return -sympy.I*(H*rho-rho*H) + loss_superoperator(rho)

# construct Liouvillan matrix
L = sympy.zeros(16)
for column, (icol, jcol) in enumerate(zip(*flatten)):
    # basis matrix
    rho = sympy.zeros(4)
    rho[icol,jcol] = 1

    # get column of supermatrix
    L_column = liouvillan_superoperator(rho)
    for row, (irow, jrow) in enumerate(zip(*flatten)):
        L[row,column] = L_column[irow,jrow]

# matrix to convert complex Hermitian to real nonsymmetric
C = sympy.zeros(16)
for k in range(4):
    ik = unflatten[k,k]
    C[ik,ik] = 1
for j in range(3):
    for k in range(j+1, 4):
        ij = unflatten[j,k]
        ik = unflatten[k,j]
        C[ij,ij] = sympy.Rational(1, 2)
        C[ij,ik] = sympy.Rational(1, 2)
        C[ik,ij] = -sympy.I/2
        C[ik,ik] = sympy.I/2

```

```
# Liouvillan matrix converted to real
C_inv = C.inv()
Lreal = sympy.re(C*L*C.inv())

# lambdified functions
def short_lambdify(params, expr):
    return sympy.lambdify(params, expr, 'numpy', cse=True, docstring_limit=0)

Hsymbols = (omega12, delta_B, delta_mu, Omega_mu,
            Omega_A, Omega_B, Omega_D, Omega_E)
Lsymbols = Hsymbols + (gamma12, gamma13, gamma14, gamma23, gamma24, gamma34,
                       gamma2d, gamma3d, gamma4d, nbath_12, nbath_34)
Hfunc = short_lambdify(Hsymbols, H)
Lrealfunc = short_lambdify(Lsymbols, Lreal)

# gross hack to get numerical matrices from sympy
x = sympy.symbols('x')
complex_to_real = short_lambdify(x, C)(None)
real_to_complex = short_lambdify(x, C_inv)(None)

# discriminant symbolic and lambdified functions for numerical methods

def expr_to_poly_coeffs(expr, x):
    poly = sympy.poly(expr, x)
    n = poly.degree(x)
    coeffs = sympy.zeros(n+1, 1)
    for (k,), coeff in zip(poly.monoms(), poly.coeffs()):
        coeffs[k] = coeff
    return coeffs

def diff_poly_coeffs(coeffs):
    n = coeffs.shape[0]-1
    diff_coeffs = sympy.zeros(n, 1)
    for k in range(1, n+1):
        diff_coeffs[k-1,0] = sympy.Integer(k)*coeffs[k,0]
    return diff_coeffs

# restore the symbolic expressions from a cache, if it exists
fname = 'result-cache/Delta_expr.pkl'
if os.path.isfile(fname):
    with open(fname, 'rb') as f:
        Delta_expr = pickle.load(f)
        Delta = Delta_expr['Delta']
        Delta_coeffs_delta_B = Delta_expr['Delta_coeffs_delta_B']
        Delta_coeffs_delta_mu = Delta_expr['Delta_coeffs_delta_mu']
        dmu_Delta = Delta_expr['dmu_Delta']
        dmu_Delta_coeffs_delta_B = Delta_expr['dmu_Delta_coeffs_delta_B']
        dmu_Delta_coeffs_delta_mu = Delta_expr['dmu_Delta_coeffs_delta_mu']
        dB_Delta = Delta_expr['dB_Delta']
        dB_Delta_coeffs_delta_B = Delta_expr['dB_Delta_coeffs_delta_B']
        dB_Delta_coeffs_delta_mu = Delta_expr['dB_Delta_coeffs_delta_mu']
        d2mu_Delta = Delta_expr['d2mu_Delta']
        d2B_Delta = Delta_expr['d2B_Delta']
else:
    # compute the discriminant Delta of the characteristic polynomial
    # of the Hamiltonian; this slightly convoluted method of using
    # a generic quartic is faster than the obvious way
    print('Computing the symbolic polynomial coefficients of the discriminant')
    print('This takes about 20 minutes on my machine, so get ready to wait')
    print(f'The result will be saved as {fname} to be re-used in future builds')
    a = [a0, a1, a2, a3, a4] = sympy.symbols('a_(0:5)', real=True)
    x = sympy.symbols('x', real=True)
    poly = sympy.poly(a4*x**4 + a3*x**3 + a2*x**2 + a1*x + a0, x)
    Delta = poly.discriminant()
    charpoly = H.charpoly()
    for (k,), coeff in zip(charpoly.monoms(), charpoly.coeffs()):
```

```
178         Delta = Delta.subs(a[k], coeff)
179
180     # compute various polynomial coefficients and derivatives of Delta
181     Delta_coeffs_delta_B = expr_to_poly_coeffs(Delta, delta_B)
182     Delta_coeffs_delta_mu = expr_to_poly_coeffs(Delta, delta_mu)
183     dmu_Delta = sympy.diff(Delta, delta_mu)
184     dmu_Delta_coeffs_delta_B = sympy.diff(Delta_coeffs_delta_B, delta_mu)
185     dmu_Delta_coeffs_delta_mu = diff_poly_coeffs(Delta_coeffs_delta_mu)
186     dB_Delta = sympy.diff(Delta, delta_B)
187     dB_Delta_coeffs_delta_B = diff_poly_coeffs(Delta_coeffs_delta_B)
188     dB_Delta_coeffs_delta_mu = sympy.diff(Delta_coeffs_delta_mu, delta_B)
189     d2mu_Delta = sympy.diff(Delta, delta_mu, 2)
190     d2B_Delta = sympy.diff(Delta, delta_B, 2)
191     Delta_expr = {
192         'Delta': Delta,
193         'Delta_coeffs_delta_B': Delta_coeffs_delta_B,
194         'Delta_coeffs_delta_mu': Delta_coeffs_delta_mu,
195         'dmu_Delta': dmu_Delta,
196         'dmu_Delta_coeffs_delta_B': dmu_Delta_coeffs_delta_B,
197         'dmu_Delta_coeffs_delta_mu': dmu_Delta_coeffs_delta_mu,
198         'dB_Delta': dB_Delta,
199         'dB_Delta_coeffs_delta_B': dB_Delta_coeffs_delta_B,
200         'dB_Delta_coeffs_delta_mu': dB_Delta_coeffs_delta_mu,
201         'd2mu_Delta': d2mu_Delta,
202         'd2B_Delta': d2B_Delta
203     }
204     with open(fname, 'wb') as f:
205         pickle.dump(Delta_expr, f)
206
207 H_symbols_common = (omega12, Omega_A, Omega_B, Omega_D, Omega_E, Omega_mu)
208 H_symbols_no_dB = H_symbols_common + (delta_mu,)
209 H_symbols_no_dmu = H_symbols_common + (delta_B,)
210 H_symbols_all = Hsymbols
211
212 # lambdify these expressions; this can't be pickled
213 Delta_func = short_lambdify(H_symbols_all, Delta)
214 Delta_coeffs_delta_B_func = short_lambdify(H_symbols_no_dB,
215                                            Delta_coeffs_delta_B)
216 Delta_coeffs_delta_mu_func = sympy.lambdify(H_symbols_no_dmu,
217                                             Delta_coeffs_delta_mu)
218 dmu_Delta_func = sympy.lambdify(H_symbols_all, dmu_Delta)
219 dmu_Delta_coeffs_delta_B_func = sympy.lambdify(H_symbols_no_dB,
220                                                dmu_Delta_coeffs_delta_B)
221 dmu_Delta_coeffs_delta_mu_func = sympy.lambdify(H_symbols_no_dmu,
222                                                 dmu_Delta_coeffs_delta_mu)
223 dB_Delta_func = sympy.lambdify(H_symbols_all, dB_Delta)
224 dB_Delta_coeffs_delta_B_func = sympy.lambdify(H_symbols_no_dB,
225                                               dB_Delta_coeffs_delta_B)
226 dB_Delta_coeffs_delta_mu_func = sympy.lambdify(H_symbols_no_dmu,
227                                                dB_Delta_coeffs_delta_mu)
228 d2mu_Delta_func = sympy.lambdify(H_symbols_all, d2mu_Delta)
229 d2B_Delta_func = sympy.lambdify(H_symbols_all, d2B_Delta)
230
231 # helper function for multivariate normal distribution; scipy's has a bad API
232
233 def multivariate_normal_pdf(Sigma, *args):
234     d = len(args)
235     x = np.array(args)
236     coef = 1 / np.sqrt((2*np.pi)**d * np.linalg.det(Sigma))
237
238     # permutation sending the first dimension to the second-last
239     axes = list(range(x.ndim))
240     axes[0] = -2
241     axes[-2] = 0
242
243     xDivSigma = np.linalg.solve(Sigma, np.transpose(x, axes=axes))
```

```
    xDivSigma = np.transpose(xDivSigma, axes=axes)

    return coef * np.exp(-np.einsum('i...,i...', x, xDivSigma)/2)

# computes a grid of atom output signal, without multisampling

def rho_steady_state(omega_12, omega_34, delta_B, delta_mu, Omega_A,
                     Omega_B, Omega_D, Omega_E, Omega_mu, tau_12,
                     tau_34, gamma_13, gamma_14, gamma_23, gamma_24,
                     gamma_2d, gamma_3d, gamma_4d, T):
    n_12 = 1 / np.expm1(hbar*omega_12/(kB*T))
    n_34 = 1 / np.expm1(hbar*omega_34/(kB*T))
    L = Lrealfunc(
        omega_12=omega_12,
        delta_B=delta_B,
        delta_mu=delta_mu,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu,
        gamma_12=1/(tau_12*(n_12+1)),
        gamma_13=gamma_13,
        gamma_14=gamma_14,
        gamma_23=gamma_23,
        gamma_24=gamma_24,
        gamma_34=1/(tau_34*(n_34+1)),
        gamma_2d=gamma_2d,
        gamma_3d=gamma_3d,
        gamma_4d=gamma_4d,
        n_12=n_12,
        n_34=n_34
    )
    L[0,:] = np.identity(4)[flatten]

    b = np.zeros(16)
    b[0] = 1

    rho_real = np.linalg.solve(L, b)
    rho = real_to_complex @ rho_real
    return rho[unflatten]

def atom_signal(omega_12, omega_34, delta_B, delta_mu, Omega_A,
                Omega_B, Omega_D, Omega_E, Omega_mu, tau_12, tau_34,
                gamma_13, gamma_14, gamma_23, gamma_24, C_14,
                C_24, gamma_2d, gamma_3d, gamma_4d, T):
    rho = rho_steady_state(
        omega_12=omega_12,
        omega_34=omega_34,
        delta_B=delta_B,
        delta_mu=delta_mu,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu,
        tau_12=tau_12,
        tau_34=tau_34,
        gamma_13=gamma_13,
        gamma_14=gamma_14,
        gamma_23=gamma_23,
        gamma_24=gamma_24,
        gamma_2d=gamma_2d,
        gamma_3d=gamma_3d,
        gamma_4d=gamma_4d,
        T=T
```

```
    )

    signal_D = C_24 * rho[3,1]
    signal_E = C_14 * rho[3,0]
    photon_rate_out = np.abs(signal_D + signal_E)**2
    return photon_rate_out

def atom_scan(delta_mu_min, delta_mu_max, delta_mu_points,
              delta_B_min, delta_B_max, delta_B_points,
              omega_12, omega_34, Omega_A, Omega_B, Omega_D, Omega_E,
              Omega_mu, tau_12, tau_34, gamma_13, gamma_14, C_14, C_24,
              gamma_23, gamma_24, gamma_2d, gamma_3d, gamma_4d, T):
    delta_mu = np.linspace(delta_mu_min, delta_mu_max, delta_mu_points)
    delta_B = np.linspace(delta_B_min, delta_B_max, delta_B_points)
    delta_mu, delta_B = np.meshgrid(delta_mu, delta_B, indexing='ij')

    scan = np.zeros_like(delta_mu)
    for i in range(scan.shape[0]):
        for j in range(scan.shape[1]):
            scan[i,j] = atom_signal(
                omega_12=omega_12,
                omega_34=omega_34,
                delta_B=delta_B[i,j],
                delta_mu=delta_mu[i,j],
                Omega_A=Omega_A,
                Omega_B=Omega_B,
                Omega_D=Omega_D,
                Omega_E=Omega_E,
                Omega_mu=Omega_mu,
                tau_12=tau_12,
                tau_34=tau_34,
                gamma_13=gamma_13,
                gamma_14=gamma_14,
                gamma_23=gamma_23,
                gamma_24=gamma_24,
                gamma_2d=gamma_2d,
                gamma_3d=gamma_3d,
                gamma_4d=gamma_4d,
                C_14=C_14,
                C_24=C_24,
                T=T
            )

    return delta_mu, delta_B, scan

# multisampling functions

def poly_real_roots(poly_coeffs):
    poly = np.polynomial.Polynomial(poly_coeffs)
    roots = poly.roots()
    real_roots = np.array([np.real(x) for x in roots if np.imag(x)==0])
    return real_roots

def poly_positive_minima(poly_func, diff_coeffs):
    critical_x = poly_real_roots(diff_coeffs)
    critical_y = poly_func(critical_x)

    # base cases
    if len(critical_x) == 0:
        return []
    if len(critical_x) <= 1:
        return critical_x

    minima = []

    # check if leftmost critical point is minimum
```

```
    if critical_y[0] < critical_y[1]:
        minima.append(critical_x[0])

    # check for mimima between critical points
    for i in range(len(critical_x)-2):
        y1 = critical_y[i]
        x2 = critical_x[i+1]
        y2 = critical_y[i+1]
        y3 = critical_y[i+2]
        if y1 > y2 and y3 > y2:
            minima.append(x2)

    # check if rightmost critical point is mimimum
    if critical_y[-1] < critical_y[-2]:
        minima.append(critical_x[-1])

    return minima

def H_get_minimal_delta_B(omega_12, delta_mu, Omega_A,
                          Omega_B, Omega_D, Omega_E, Omega_mu):
    # minima along delta_B
    dB_Delta_coeffs = dB_Delta_coeffs_delta_B_func(
        omega_12=omega_12,
        delta_mu=delta_mu,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )[:,0]
    Delta_short_func = lambda delta_B: Delta_func(
        omega_12=omega_12,
        delta_mu=delta_mu,
        delta_B=delta_B,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )
    minima = list(poly_positive_minima(Delta_short_func, dB_Delta_coeffs))
    is_parallel = [True]*len(minima)

    # minima along delta_mu
    dmu_Delta_coeffs = dmu_Delta_coeffs_delta_B_func(
        omega_12=omega_12,
        delta_mu=delta_mu,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )[:,0]
    roots = poly_real_roots(dmu_Delta_coeffs)
    for root in roots:
        curvature = d2mu_Delta_func(
            omega_12=omega_12,
            delta_mu=delta_mu,
            delta_B=root,
            Omega_A=Omega_A,
            Omega_B=Omega_B,
            Omega_D=Omega_D,
            Omega_E=Omega_E,
            Omega_mu=Omega_mu
        )
        if curvature >= 0:
```

```
            minima.append(root)
            is_parallel.append(False)

    return minima, is_parallel

def H_get_minimal_delta_mu(omega_12, delta_B, Omega_A,
                           Omega_B, Omega_D, Omega_E, Omega_mu):
    # minima along delta_mu
    dmu_Delta_coeffs = dmu_Delta_coeffs_delta_mu_func(
        omega_12=omega_12,
        delta_B=delta_B,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )[:,0]
    Delta_short_func = lambda delta_mu: Delta_func(
        omega_12=omega_12,
        delta_mu=delta_mu,
        delta_B=delta_B,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )
    minima = list(poly_positive_minima(Delta_short_func, dmu_Delta_coeffs))
    is_parallel = [True]*len(minima)

    # minima along delta_mu
    dB_Delta_coeffs = dB_Delta_coeffs_delta_mu_func(
        omega_12=omega_12,
        delta_B=delta_B,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )[:,0]
    roots = poly_real_roots(dB_Delta_coeffs)
    for root in roots:
        curvature = d2B_Delta_func(
            omega_12=omega_12,
            delta_mu=root,
            delta_B=delta_B,
            Omega_A=Omega_A,
            Omega_B=Omega_B,
            Omega_D=Omega_D,
            Omega_E=Omega_E,
            Omega_mu=Omega_mu
        )
        if curvature >= 0:
            minima.append(root)
            is_parallel.append(False)

    return minima, is_parallel

def find_curve_pixel_intersections(delta_mu_min, delta_mu_max, delta_mu_points,
                                   delta_B_min, delta_B_max, delta_B_points,
                                   omega_12, Omega_A, Omega_B,
                                   Omega_D, Omega_E, Omega_mu):
    def linspace_edges(start, stop, count):
        dx = (stop-start) / (count-1)
        edges = np.linspace(start-dx/2, stop+dx/2, count+1)
        return edges
```

```
508
509     delta_mu_edges = linspace_edges(delta_mu_min, delta_mu_max, delta_mu_points)
510     delta_B_edges = linspace_edges(delta_B_min, delta_B_max, delta_B_points)
511     delta_mu_edges_min, delta_mu_edges_max = delta_mu_edges[0], delta_mu_edges[-1]
512     delta_B_edges_min, delta_B_edges_max = delta_B_edges[0], delta_B_edges[-1]
513
514     intersection_points = []
515     intersecting_pixels = dict()
516     is_parallel = []
517
518     for i, dmu in enumerate(delta_mu_edges):
519         dBs, paras = H_get_minimal_delta_B(
520             omega_12=omega_12,
521             delta_mu=dmu,
522             Omega_A=Omega_A,
523             Omega_B=Omega_B,
524             Omega_D=Omega_D,
525             Omega_E=Omega_E,
526             Omega_mu=Omega_mu
527         )
528         for dB, para in zip(dBs, paras):
529             if not delta_B_edges_min <= dB <= delta_B_edges_max:
530                 continue
531             point = (dmu, dB)
532             intersection_points.append((dmu, dB))
533             is_parallel.append(para)
534
535             j = int(delta_B_points * (dB-delta_B_edges_min) /
    (delta_B_edges_max-delta_B_edges_min))
536             if i > 0:
537                 if (i-1,j) not in intersecting_pixels:
538                     intersecting_pixels[i-1,j] = []
539                 intersecting_pixels[i-1,j].append(point)
540             if i < delta_mu_points:
541                 if (i,j) not in intersecting_pixels:
542                     intersecting_pixels[i,j] = []
543                 intersecting_pixels[i,j].append(point)
544
545     for j, dB in enumerate(delta_B_edges):
546         dmus, paras = H_get_minimal_delta_mu(
547             omega_12=omega_12,
548             delta_B=dB,
549             Omega_A=Omega_A,
550             Omega_B=Omega_B,
551             Omega_D=Omega_D,
552             Omega_E=Omega_E,
553             Omega_mu=Omega_mu
554         )
555         for dmu, para in zip(dmus, paras):
556             if not delta_mu_edges_min <= dmu <= delta_mu_edges_max:
557                 continue
558             point = (dmu, dB)
559             intersection_points.append((dmu, dB))
560             is_parallel.append(para)
561
562             i = int(delta_mu_points * (dmu-delta_mu_edges_min) /
    (delta_mu_edges_max-delta_mu_edges_min))
563             if j > 0:
564                 if (i,j-1) not in intersecting_pixels:
565                     intersecting_pixels[i,j-1] = []
566                 intersecting_pixels[i,j-1].append(point)
567             if j < delta_B_points:
568                 if (i,j) not in intersecting_pixels:
569                     intersecting_pixels[i,j] = []
570                 intersecting_pixels[i,j].append(point)
571
```

```
572     return delta_mu_edges, delta_B_edges, intersection_points, intersecting_pixels,
        is_parallel
573
574 # precompute Gauss–Lobatto quadrature nodes and weights
575 gauss_lobatto_nodes_weights = dict()
576 for n in range(2, 100):
577     legendre_coeffs = (0,)*(n-1) + (1,)
578     legendre_poly = np.polynomial.Legendre(legendre_coeffs)
579     nodes = legendre_poly.deriv().roots()
580     nodes = np.concatenate([[-1.0], nodes, [1.0]])
581     weights = 2 / (n*(n-1)*legendre_poly(nodes)**2)
582
583     # convert from [-1, 1] to [0, 1]
584     nodes = (nodes+1)/2
585     weights = weights/2
586
587     gauss_lobatto_nodes_weights[n] = (nodes, weights)
588
589 def composite_gauss_lobatto_nodes_weights(n, a, b, intermediates=[]):
590     if n not in gauss_lobatto_nodes_weights:
591         raise ValueError
592     if a >= b:
593         raise ValueError
594
595     intermediates = [x for x in intermediates if a < x < b]
596     points = sorted([a, b] + intermediates)
597     num_points = len(points)
598     base_nodes, base_weights = gauss_lobatto_nodes_weights[n]
599     if num_points == 2:
600         L = points[1] - points[0]
601         nodes = points[0] + base_nodes*L
602         weights = base_weights*L
603         return nodes, weights
604
605     count = (num_points-1)*(n-1) + 1
606     nodes = np.zeros(count)
607     weights = np.zeros(count)
608
609     # first and last node
610     nodes[0] = points[0]
611     weights[0] = base_weights[0] * (points[1]-points[0])
612     nodes[1] = points[-1]
613     weights[1] = base_weights[-1] * (points[-1]-points[-2])
614     filled = 2
615
616     # nodes at intermediate points
617     for i in range(1,num_points-1):
618         nodes[filled] = points[i]
619         weights[filled] = base_weights[-1]*(points[i]-points[i-1])
620         weights[filled] += base_weights[0]*(points[i+1]-points[i])
621         filled += 1
622
623     # nodes between points
624     if n == 2:
625         return nodes, weights
626     for i in range(num_points-1):
627         L = points[i+1]-points[i]
628         nodes[filled:filled+n-2] = points[i] + base_nodes[1:-1]*L
629         weights[filled:filled+n-2] = base_weights[1:-1]*L
630         filled += n-2
631
632     return nodes, weights
633
634 def atom_signal_integrated(delta_mu_min, delta_mu_max, delta_B_min, delta_B_max,
635                            edge_points, omega_12, omega_34, Omega_A, Omega_B,
    Omega_D,
```

```
                            Omega_E, Omega_mu, tau_12, tau_34, gamma_13, gamma_14,
                            gamma_23, gamma_24, gamma_2d, gamma_3d, gamma_4d, C_14,
    C_24, T):
    outer_order = 5
    inner_order = 10

    delta_B_linewidth = gamma_3d
    delta_mu_linewidth = gamma_3d + gamma_4d

    area = (delta_mu_max-delta_mu_min) * (delta_B_max-delta_B_min)

    def lerp(a, b, t):
        return a + t*(b-a)
    def ilerp(a, b, x):
        return (x-a)/(b-a)

    def normalise_delta_mu(delta_mu):
        return ilerp(delta_mu_min, delta_mu_max, delta_mu)
    def unnormalise_delta_mu(t):
        return lerp(delta_mu_min, delta_mu_max, t)
    def normalise_delta_B(delta_B):
        return ilerp(delta_B_min, delta_B_max, delta_B)
    def unnormalise_delta_B(t):
        return lerp(delta_B_min, delta_B_max, t)

    def bounds_range(array):
        if len(array) == 0:
            return [], 0
        if len(array) == 1:
            return list(array), 0

        a = np.min(array)
        b = np.max(array)
        return [a, b], b-a

    def atom_signal_short(delta_mu, delta_B):
        return atom_signal(
            omega_12=omega_12,
            omega_34=omega_34,
            delta_mu=delta_mu,
            delta_B=delta_B,
            Omega_A=Omega_A,
            Omega_B=Omega_B,
            Omega_D=Omega_D,
            Omega_E=Omega_E,
            Omega_mu=Omega_mu,
            tau_12=tau_12,
            tau_34=tau_34,
            gamma_13=gamma_13,
            gamma_14=gamma_14,
            gamma_23=gamma_23,
            gamma_24=gamma_24,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            gamma_4d=gamma_4d,
            C_14=C_14,
            C_24=C_24,
            T=T,
        )

    edge_mu_points = [x[0] for x in edge_points]
    edge_B_points = [x[1] for x in edge_points]

    # get direction for outer integral
    if len(edge_points) == 0:
        outer = 'delta_mu' # arbitrary
```

```
    elif len(edge_points) == 1:
        edge_mu = edge_mu_points[0]
        edge_B = edge_B_points[0]
        if normalise_delta_mu(edge_mu) > normalise_delta_B(edge_B):
            outer = 'delta_mu'
        else:
            outer = 'delta_B'
    else:
        edge_mu_min = np.min(edge_mu_points)
        edge_mu_max = np.max(edge_mu_points)
        edge_B_min = np.min(edge_B_points)
        edge_B_max = np.max(edge_B_points)
        width_mu = (edge_mu_max-edge_mu_min) / (delta_mu_max-delta_mu_min)
        width_B = (edge_B_max-edge_B_min) / (delta_B_max-delta_B_min)
        if width_mu > width_B:
            outer = 'delta_mu'
        else:
            outer = 'delta_B'

    # orientation-neutral definitions
    def atom_signal_short_neutral(delta_outer, delta_inner):
        if outer == 'delta_mu':
            delta_mu = delta_outer
            delta_B = delta_inner
        else:
            delta_mu = delta_inner
            delta_B = delta_outer
        return atom_signal_short(delta_mu, delta_B)

    def H_get_minimal_delta_inner(delta_outer):
        if outer == 'delta_mu':
            return H_get_minimal_delta_B(
                omega_12=omega_12,
                delta_mu=delta_outer,
                Omega_A=Omega_A,
                Omega_B=Omega_B,
                Omega_D=Omega_D,
                Omega_E=Omega_E,
                Omega_mu=Omega_mu
            )
        else:
            return H_get_minimal_delta_mu(
                omega_12=omega_12,
                delta_B=delta_outer,
                Omega_A=Omega_A,
                Omega_B=Omega_B,
                Omega_D=Omega_D,
                Omega_E=Omega_E,
                Omega_mu=Omega_mu
            )

    if outer == 'delta_mu':
        delta_outer_min = delta_mu_min
        delta_outer_max = delta_mu_max
        delta_inner_min = delta_B_min
        delta_inner_max = delta_B_max
        delta_inner_linewidth = delta_B_linewidth
        edge_outer_points = edge_mu_points
        edge_inner_points = edge_B_points
    else:
        delta_outer_min = delta_B_min
        delta_outer_max = delta_B_max
        delta_inner_min = delta_mu_min
        delta_inner_max = delta_mu_max
        delta_inner_linewidth = delta_mu_linewidth
        edge_outer_points = edge_B_points
```

```
        edge_inner_points = edge_mu_points

    edge_outer_bounds, _ = bounds_range(edge_outer_points)

    # set up diagnostic logging
    log = {
        'delta_mu_min': delta_mu_min,
        'delta_mu_max': delta_mu_max,
        'delta_B_min': delta_B_min,
        'delta_B_max': delta_B_max,
        'edge_points': edge_points,
        'omega_12': omega_12,
        'omega_34': omega_34,
        'Omega_A': Omega_A,
        'Omega_B': Omega_B,
        'Omega_D': Omega_D,
        'Omega_E': Omega_E,
        'Omega_mu': Omega_mu,
        'tau_12': tau_12,
        'tau_34': tau_34,
        'gamma_13': gamma_13,
        'gamma_14': gamma_14,
        'gamma_23': gamma_23,
        'gamma_24': gamma_24,
        'gamma_2d': gamma_2d,
        'gamma_3d': gamma_3d,
        'gamma_4d': gamma_4d,
        'C_14': C_14,
        'C_24': C_24,
        'T': T,
        'outer': outer
    }

    # perform integral
    integral = 0
    outer_nodes, outer_weights = composite_gauss_lobatto_nodes_weights(
        outer_order, delta_outer_min, delta_outer_max,
    intermediates=edge_outer_bounds)
    log['outer_integral'] = {
        'intermediates': edge_outer_bounds,
        'nodes': outer_nodes,
        'weights': outer_weights,
        'inner_integrals': []
    }
    for outer_node, outer_weight in zip(outer_nodes, outer_weights):
        inner_intersections, _ = H_get_minimal_delta_inner(outer_node)
        inner_intermediates = [z*delta_inner_linewidth+x
                               for x in inner_intersections
                               for z in (-10, -3, -1, 0, 1, 3, 10)]
        log_inner_integral = {
            'intersections': inner_intersections,
            'intermediates': inner_intermediates
        }
        inner_nodes, inner_weights = composite_gauss_lobatto_nodes_weights(
            inner_order, delta_inner_min, delta_inner_max,
    intermediates=inner_intermediates)
        log_inner_integral['nodes'] = inner_nodes
        log_inner_integral['weights'] = inner_weights
        log_inner_integral['values'] = []
        for inner_node, inner_weight in zip(inner_nodes, inner_weights):
            weight = outer_weight * inner_weight
            sample = atom_signal_short_neutral(outer_node, inner_node) / area
            log_inner_integral['values'].append(sample)
            integral += weight * sample
        log['outer_integral']['inner_integrals'].append(log_inner_integral)

```

```
    log['integral'] = integral
    return integral, log

def atom_scan_integrated(delta_mu_min, delta_mu_max, delta_mu_points,
                         delta_B_min, delta_B_max, delta_B_points,
                         omega_12, omega_34, Omega_A, Omega_B, Omega_D, Omega_E,
                         Omega_mu, tau_12, tau_34, gamma_13, gamma_14, gamma_23,
                         gamma_24, gamma_2d, gamma_3d, gamma_4d, C_14, C_24, T):

    (delta_mu_edges,
     delta_B_edges,
     intersection_points,
     intersecting_pixels, _) = find_curve_pixel_intersections(
        delta_mu_min=delta_mu_min,
        delta_mu_max=delta_mu_max,
        delta_mu_points=delta_mu_points,
        delta_B_min=delta_B_min,
        delta_B_max=delta_B_max,
        delta_B_points=delta_B_points,
        omega_12=omega_12,
        Omega_A=Omega_A,
        Omega_B=Omega_B,
        Omega_D=Omega_D,
        Omega_E=Omega_E,
        Omega_mu=Omega_mu
    )

    scan = np.full((delta_mu_points, delta_B_points), np.nan, dtype=float)
    logs = dict()
    for (i,j), edge_points in intersecting_pixels.items():
        scan[i,j], logs[i,j] = atom_signal_integrated(
            delta_mu_min=delta_mu_edges[i],
            delta_mu_max=delta_mu_edges[i+1],
            delta_B_min=delta_B_edges[j],
            delta_B_max=delta_B_edges[j+1],
            edge_points=edge_points,
            omega_12=omega_12,
            omega_34=omega_34,
            Omega_A=Omega_A,
            Omega_B=Omega_B,
            Omega_D=Omega_D,
            Omega_E=Omega_E,
            Omega_mu=Omega_mu,
            tau_12=tau_12,
            tau_34=tau_34,
            gamma_13=gamma_13,
            gamma_14=gamma_14,
            gamma_23=gamma_23,
            gamma_24=gamma_24,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            gamma_4d=gamma_4d,
            C_14=C_14,
            C_24=C_24,
            T=T
        )

    delta_mu = np.linspace(delta_mu_min, delta_mu_max, delta_mu_points)
    delta_B = np.linspace(delta_B_min, delta_B_max, delta_B_points)
    delta_mu, delta_B = np.meshgrid(delta_mu, delta_B, indexing='ij')
    for i in range(scan.shape[0]):
        for j in range(scan.shape[1]):
            if not np.isnan(scan[i,j]):
                continue
            scan[i,j] = atom_signal(
                omega_12=omega_12,
```

```
                omega_34=omega_34,
                delta_B=delta_B[i,j],
                delta_mu=delta_mu[i,j],
                Omega_A=Omega_A,
                Omega_B=Omega_B,
                Omega_D=Omega_D,
                Omega_E=Omega_E,
                Omega_mu=Omega_mu,
                tau_12=tau_12,
                tau_34=tau_34,
                gamma_13=gamma_13,
                gamma_14=gamma_14,
                gamma_23=gamma_23,
                gamma_24=gamma_24,
                gamma_2d=gamma_2d,
                gamma_3d=gamma_3d,
                gamma_4d=gamma_4d,
                C_14=C_14,
                C_24=C_24,
                T=T
            )

    return (delta_mu, delta_B, scan), logs

def signal_scan(delta_mu_min, delta_mu_max, delta_mu_points,
                delta_B_min, delta_B_max, delta_B_points,
                omega_12, omega_34, Omega_A, Omega_B, Omega_D, Omega_E,
                Omega_mu, tau_12, tau_34, gamma_13, gamma_14,
                gamma_23, gamma_24, gamma_2d, gamma_3d, gamma_4d, T,
                C_14, C_24, Sigma, z_max=5, integrated=True):
    delta_mu_res = (delta_mu_max-delta_mu_min) / (delta_mu_points-1)
    delta_B_res = (delta_B_max-delta_B_min) / (delta_B_points-1)

    # produce Gaussian filter kernel
    sigma_13 = np.sqrt(Sigma[0,0])
    sigma_34 = np.sqrt(Sigma[1,1])
    sigma_13_px = sigma_13 / delta_B_res
    sigma_34_px = sigma_34 / delta_mu_res
    radius_13_px = int(sigma_13_px*z_max)
    radius_34_px = int(sigma_34_px*z_max)
    delta_13_points = 2*radius_13_px + 1
    delta_34_points = 2*radius_34_px + 1
    delta_13 = np.linspace(-sigma_13*z_max, sigma_13*z_max, delta_13_points)
    delta_34 = np.linspace(-sigma_34*z_max, sigma_34*z_max, delta_34_points)
    delta_34, delta_13 = np.meshgrid(delta_34, delta_13, indexing='ij')
    kernel = multivariate_normal_pdf(Sigma, delta_13, delta_34)
    kernel /= np.sum(kernel)

    # collect atomic emission data
    deltap_B_min = delta_B_min - delta_B_res*radius_13_px
    deltap_B_max = delta_B_max + delta_B_res*radius_13_px
    deltap_B_points = delta_B_points + delta_13_points - 1
    deltap_mu_min = delta_mu_min - delta_mu_res*radius_34_px
    deltap_mu_max = delta_mu_max + delta_mu_res*radius_34_px
    deltap_mu_points = delta_mu_points + delta_34_points - 1

    if integrated:
        (_, _, atom_scan), _ = atom_scan_integrated(
            delta_mu_min=deltap_mu_min,
            delta_mu_max=deltap_mu_max,
            delta_mu_points=deltap_mu_points,
            delta_B_min=deltap_B_min,
            delta_B_max=deltap_B_max,
            delta_B_points=deltap_B_points,
            omega_12=omega_12,
            omega_34=omega_34,
```

```
            Omega_A=Omega_A,
            Omega_B=Omega_B,
            Omega_D=Omega_D,
            Omega_E=Omega_E,
            Omega_mu=Omega_mu,
            tau_12=tau_12,
            tau_34=tau_34,
            gamma_13=gamma_13,
            gamma_14=gamma_14,
            gamma_23=gamma_23,
            gamma_24=gamma_24,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            gamma_4d=gamma_4d,
            C_14=C_14,
            C_24=C_24,
            T=T
        )
    else:
        _, _, atom_scan = atom_scan(
            delta_mu_min=deltap_mu_min,
            delta_mu_max=deltap_mu_max,
            delta_mu_points=deltap_mu_points,
            delta_B_min=deltap_B_min,
            delta_B_max=deltap_B_max,
            delta_B_points=deltap_B_points,
            omega_12=omega_12,
            omega_34=omega_34,
            Omega_A=Omega_A,
            Omega_B=Omega_B,
            Omega_D=Omega_D,
            Omega_E=Omega_E,
            Omega_mu=Omega_mu,
            tau_12=tau_12,
            tau_34=tau_34,
            gamma_13=gamma_13,
            gamma_14=gamma_14,
            gamma_23=gamma_23,
            gamma_24=gamma_24,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            gamma_4d=gamma_4d,
            C_14=C_14,
            C_24=C_24,
            T=T
        )

    # convolve
    ensemble_signal = signal.fftconvolve(atom_scan, kernel, mode='valid')

    # return results
    delta_mu = np.linspace(delta_mu_min, delta_mu_max, delta_mu_points)
    delta_B = np.linspace(delta_B_min, delta_B_max, delta_B_points)
    delta_mu, delta_B = np.meshgrid(delta_mu, delta_B, indexing='ij')
    return delta_mu, delta_B, ensemble_signal
```

C.3 Biphoton Generation

These codes implement the three-level biphoton generation models in Chapter 4.

C.3.1 Steady State

This Python code implements the steady-state model. When run as a script, it performs root finding for cavity steady-states, and prints the (non-convergent) results of this root finding, for two sets of detuning parameters.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize, stats
import sympy

hbar = 1.054571817e-34
kB = 1.380649e-23

gauss_lobatto_nodes_weights = dict()

for n in range(2, 100):
    legendre_coeffs = (0,)*(n-1) + (1,)
    legendre_poly = np.polynomial.Legendre(legendre_coeffs)
    nodes = legendre_poly.deriv().roots()
    nodes = np.concatenate([[-1.0], nodes, [1.0]])
    weights = 2 / (n*(n-1)*legendre_poly(nodes)**2)

    # convert from [-1, 1] to [0, 1]
    nodes = (nodes+1)/2
    weights = weights/2

    gauss_lobatto_nodes_weights[n] = (nodes, weights)

def composite_gauss_lobatto_nodes_weights(n, points):
    if n not in gauss_lobatto_nodes_weights:
        raise ValueError

    num_points = len(points)
    if num_points < 2:
        raise ValueError

    base_nodes, base_weights = gauss_lobatto_nodes_weights[n]
    if num_points == 2:
        L = points[1] - points[0]
        nodes = points[0] + base_nodes*L
        weights = base_weights*L
        return nodes, weights

    count = (num_points-1)*(n-1) + 1
    nodes = np.zeros(count)
    weights = np.zeros(count)

    # first and last node
    nodes[0] = points[0]
    weights[0] = base_weights[0] * (points[1]-points[0])
    nodes[1] = points[-1]
    weights[1] = base_weights[-1] * (points[-1]-points[-2])
    filled = 2

    # nodes at intermediate points
    for i in range(1,num_points-1):
        nodes[filled] = points[i]
        weights[filled] = base_weights[-1]*(points[i]-points[i-1])
        weights[filled] += base_weights[0]*(points[i+1]-points[i])
        filled += 1

    # nodes between points
    if n == 2:
        return nodes, weights
    for i in range(num_points-1):
```

```
        L = points[i+1]-points[i]
        nodes[filled:filled+n-2] = points[i] + base_nodes[1:-1]*L
        weights[filled:filled+n-2] = base_weights[1:-1]*L
        filled += n-2

    return nodes, weights

# unit matrices
s12 = sympy.Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]])
s13 = sympy.Matrix([[0, 0, 1], [0, 0, 0], [0, 0, 0]])
s21 = s12.H
s31 = s13.H
s11 = s12*s21
s22 = s21*s12
s23 = s21*s13
s32 = s23.H
s33 = s31*s13

# Hamiltonians for Lambda-system and V-system

Omega_p = sympy.symbols('Omega_p')
Omega_o = sympy.symbols('Omega_o')
Omega_mu = sympy.symbols('Omega_mu')
delta_o = sympy.symbols('delta_o', real=True)
delta_mu = sympy.symbols('delta_mu', real=True)

HLambda = s21*Omega_mu + s32*Omega_o + s31*Omega_p
HLambda += HLambda.H
HLambda += s22*delta_mu + s33*(delta_mu+delta_o)

HV = s21*Omega_o + s32*Omega_mu + s31*Omega_p
HV += HV.H
HV += s22*delta_o + s33*(delta_mu+delta_o)

systems = ('Lambda', 'V')
H = {'Lambda': HLambda, 'V': HV}

# decomposition of Hamiltonians into linear components
Omega_or, Omega_oi = sympy.symbols('Omega_or Omega_oi', real=True)
Omega_mur, Omega_mui = sympy.symbols('Omega_\\mu\\ r Omega_\\mu\\ i', real=True)
subs = [
    (Omega_o, Omega_or+sympy.I*Omega_oi),
    (Omega_mu, Omega_mur+sympy.I*Omega_mui)
]

H_subs = dict()
for sys in systems:
    H_subs[sys] = H[sys].subs(subs)

H0 = dict()
Hor = dict()
Hoi = dict()
Hmur = dict()
Hmui = dict()
for sys in systems:
    H0[sys] = H[sys].subs([(Omega_o, 0), (Omega_mu, 0)])
    Hor[sys] = sympy.diff(H_subs[sys], Omega_or)
    Hoi[sys] = sympy.diff(H_subs[sys], Omega_oi)
    Hmur[sys] = sympy.diff(H_subs[sys], Omega_mur)
    Hmui[sys] = sympy.diff(H_subs[sys], Omega_mui)

# discriminant of characteristic polynomial of Hamiltonian

Delta_poly_coeffs = dict()
for sys in systems:
    Delta = H[sys].charpoly().discriminant()
```

```
127     Delta_poly = sympy.poly(Delta, delta_mu, delta_o)
128     n_delta_mu = Delta_poly.degree(delta_mu)
129     n_delta_o = Delta_poly.degree(delta_o)
130     Delta_poly_coeffs[sys] = sympy.zeros(n_delta_mu+1, n_delta_o+1)
131     for (i, j), coeff in zip(Delta_poly.monoms(), Delta_poly.coeffs()):
132         Delta_poly_coeffs[sys][i,j] = coeff
133
134 # liouvillan superoperators with loss
135
136 gamma_12, gamma_13 = sympy.symbols('gamma_1(2:4)', real=True, negative=False)
137 gamma_23 = sympy.symbols('gamma_23', real=True, negative=False)
138 gamma_2d, gamma_3d = sympy.symbols('gamma_(2:4)d', real=True, negative=False)
139 n_b = sympy.symbols('n_b', real=True, negative=False)
140
141 def loss_operator_common(rho):
142     L13 = gamma_13/2 * (2*s13*rho*s31 - rho*s33 - s33*rho)
143     L2d = gamma_2d/2 * (2*s22*rho*s22 - rho*s22 - s22*rho)
144     L3d = gamma_3d/2 * (2*s33*rho*s33 - rho*s33 - s33*rho)
145     return L13 + L2d + L3d
146
147 def loss_operator_Lambda(rho):
148     L12 = gamma_12*(n_b+1)/2 * (2*s12*rho*s21 - rho*s22 - s22*rho)
149     L21 = gamma_12*n_b/2 * (2*s21*rho*s12 - rho*s11 - s11*rho)
150     L23 = gamma_23/2 * (2*s23*rho*s32 - rho*s33 - s33*rho)
151     return L12 + L21 + L23 + loss_operator_common(rho)
152
153 def loss_operator_V(rho):
154     L12 = gamma_12/2 * (2*s12*rho*s21 - rho*s22 - s22*rho)
155     L23 = gamma_23*(n_b+1)/2 * (2*s23*rho*s32 - rho*s33 - s33*rho)
156     L32 = gamma_23*n_b/2 * (2*s32*rho*s23 - rho*s22 - s22*rho)
157     return L12 + L23 + L32 + loss_operator_common(rho)
158
159 loss_operator = {'Lambda': loss_operator_Lambda, 'V': loss_operator_V}
160
161 def liouvillan_superoperator(H, rho, loss=None):
162     Lrho = -sympy.I*(H*rho - rho*H)
163     if loss is not None:
164         Lrho += loss(rho)
165     return Lrho
166
167 # liouvillan matrices
168 def flattening_indices(order):
169     n = order.shape[0]
170     unflatten = order
171
172     flatten_i = np.zeros(n**2, dtype=int)
173     flatten_j = np.zeros(n**2, dtype=int)
174     for i in range(n):
175         for j in range(n):
176             k = order[i,j]
177             flatten_i[k] = i
178             flatten_j[k] = j
179     flatten = (flatten_i, flatten_j)
180
181     return unflatten, flatten
182
183 def liouvillan_matrix(H, order, loss=None):
184     n = order.shape[0]
185     mtx = sympy.zeros(n**2)
186     for icol in range(n):
187         for jcol in range(n):
188             col = order[icol,jcol]
189             rho = sympy.zeros(n)
190             rho[icol,jcol] = 1
191             Lrho = liouvillan_superoperator(H, rho, loss=loss)
192
```

```
            for irow in range(n):
                for jrow in range(n):
                    row = order[irow,jrow]
                    mtx[row,col] = Lrho[irow,jrow]
    return mtx

order = np.array([
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
])
unflatten, flatten = flattening_indices(order)

L0 = dict()
Lor = dict()
Loi = dict()
Lmur = dict()
Lmui = dict()
L = dict()
for sys in systems:
    L0[sys] = liouvillan_matrix(H0[sys], order, loss=loss_operator[sys])
    Lor[sys] = liouvillan_matrix(Hor[sys], order)
    Loi[sys] = liouvillan_matrix(Hoi[sys], order)
    Lmur[sys] = liouvillan_matrix(Hmur[sys], order)
    Lmui[sys] = liouvillan_matrix(Hmui[sys], order)
    L[sys] = liouvillan_matrix(H[sys], order, loss=loss_operator[sys])

# real matrices

def hermitian_complex_to_real(order):
    n = order.shape[0]
    C = sympy.zeros(n**2)

    # diagonals are already real, so keep them as-is
    for k in range(n):
        ik = order[k,k]
        C[ik,ik] = 1

    # transform off-diagonal pairs from (z, z*) to (Re z, Im z)
    for j in range(n-1):
        for k in range(j+1, n):
            i_upper = order[j,k]
            i_lower = order[k,j]
            C[i_upper,i_upper] = sympy.Rational(1, 2)
            C[i_upper,i_lower] = sympy.Rational(1, 2)
            C[i_lower,i_upper] = -sympy.I/2
            C[i_lower,i_lower] = sympy.I/2

    return C

CtoR = hermitian_complex_to_real(order)
RtoC = CtoR.inv()

L0_real = dict()
Lor_real = dict()
Loi_real = dict()
Lmur_real = dict()
Lmui_real = dict()
L_real = dict()
for sys in systems:
    L0_real[sys] = sympy.re(CtoR * L0[sys] * RtoC)
    Lor_real[sys] = sympy.re(CtoR * Lor[sys] * RtoC)
    Loi_real[sys] = sympy.re(CtoR * Loi[sys] * RtoC)
    Lmur_real[sys] = sympy.re(CtoR * Lmur[sys] * RtoC)
    Lmui_real[sys] = sympy.re(CtoR * Lmui[sys] * RtoC)
    L_real[sys] = sympy.re(CtoR * L[sys] * RtoC)
```

```
259
260 # numerical matrices
261
262 def lambdify_wrapper(args, expr):
263     return sympy.lambdify(args, expr, 'numpy', cse=True, docstring_limit=0)
264
265 def numerify_zero_args_expr(expr):
266     # evil hack
267     x = sympy.symbols('x')
268     return lambdify_wrapper(x, expr)(None)
269
270 CtoR_num = numerify_zero_args_expr(CtoR)
271 RtoC_num = numerify_zero_args_expr(RtoC)
272
273 args_d = (Omega_p, Omega_mu, Omega_o)
274 args_h0 = (delta_mu, delta_o, Omega_p)
275 args_in = (Omega_mu, Omega_o)
276 args_decay = (gamma_12, gamma_13, gamma_23, gamma_2d, gamma_3d, n_b)
277
278 args_h = args_h0 + args_in
279 args_l0 = args_decay + args_h0
280 args_l = args_l0 + args_in
281
282 H0_func = dict()
283 Hor_num = dict()
284 Hoi_num = dict()
285 Hmur_num = dict()
286 Hmui_num = dict()
287 H_func = dict()
288 for sys in systems:
289     H0_func[sys] = lambdify_wrapper(args_h0, H0[sys])
290     Hor_num[sys] = numerify_zero_args_expr(Hor[sys])
291     Hoi_num[sys] = numerify_zero_args_expr(Hoi[sys])
292     Hmur_num[sys] = numerify_zero_args_expr(Hmur[sys])
293     Hmui_num[sys] = numerify_zero_args_expr(Hmui[sys])
294     H_func[sys] = lambdify_wrapper(args_h, H[sys])
295
296 Delta_poly_coeffs_func = dict()
297 for sys in systems:
298     Delta_poly_coeffs_func[sys] = lambdify_wrapper(args_d, Delta_poly_coeffs[sys])
299
300 L0_func = dict()
301 Lor_num = dict()
302 Loi_num = dict()
303 Lmur_num = dict()
304 Lmui_num = dict()
305 L_func = dict()
306 for sys in systems:
307     L0_func[sys] = lambdify_wrapper(args_l0, L0[sys])
308     Lor_num[sys] = numerify_zero_args_expr(Lor[sys])
309     Loi_num[sys] = numerify_zero_args_expr(Loi[sys])
310     Lmur_num[sys] = numerify_zero_args_expr(Lmur[sys])
311     Lmui_num[sys] = numerify_zero_args_expr(Lmui[sys])
312     L_func[sys] = lambdify_wrapper(args_l, L[sys])
313
314 L0_real_func = dict()
315 Lor_real_num = dict()
316 Loi_real_num = dict()
317 Lmur_real_num = dict()
318 Lmui_real_num = dict()
319 L_real_func = dict()
320 for sys in systems:
321     L0_real_func[sys] = lambdify_wrapper(args_l0, L0_real[sys])
322     Lor_real_num[sys] = numerify_zero_args_expr(Lor_real[sys])
323     Loi_real_num[sys] = numerify_zero_args_expr(Loi_real[sys])
324     Lmur_real_num[sys] = numerify_zero_args_expr(Lmur_real[sys])
```

```
    Lmui_real_num[sys] = numerify_zero_args_expr(Lmui_real[sys])
    L_real_func[sys] = lambdify_wrapper(args_l, L_real[sys])

def poly_bivariate_coeffs_diff_left(coeffs):
    nx = coeffs.shape[0]-1
    ny = coeffs.shape[1]-1
    i, j = np.indices((nx, ny+1))
    return (i+1) * coeffs[1:,:]

def poly_bivariate_coeffs_diff_right(coeffs):
    nx = coeffs.shape[0]-1
    ny = coeffs.shape[1]-1
    i, j = np.indices((nx+1, ny))
    return (j+1) * coeffs[:,1:]

def poly_coeffs_diff(coeffs):
    n = len(coeffs)-1
    k = np.arange(1, n+1)
    return k * coeffs[1:]

def poly_bivariate_coeffs_evaluate_left(coeffs, x):
    n = coeffs.shape[0]-1
    k = np.arange(n+1)
    return (x**k) @ coeffs

def poly_bivariate_coeffs_evaluate_right(coeffs, y):
    n = coeffs.shape[1]-1
    k = np.arange(n+1)
    return coeffs @ (y**k)

def poly_bivariate_coeffs_evaluate(coeffs, x, y):
    i, j = np.indices(coeffs.shape)
    return np.sum(coeffs * x**i * y**j)

def poly_coeffs_evaluate(coeffs, x):
    n = len(coeffs)-1
    k = np.arange(n+1)
    return np.sum(coeffs * x**k)

def poly_coeffs_roots(coeffs):
    roots = np.polynomial.polynomial.polyroots(coeffs)
    roots = np.unique(roots)
    is_real = (np.imag(roots)==0)
    return np.real(roots[is_real])

def rho_steady_state(gamma_12, gamma_13, gamma_23, gamma_2d, gamma_3d, n_b,
                     delta_mu, delta_o, Omega_p, Omega_mu, Omega_o, sys):
    L_mtx = L_real_func[sys](
        gamma_12=gamma_12,
        gamma_13=gamma_13,
        gamma_23=gamma_23,
        gamma_2d=gamma_2d,
        gamma_3d=gamma_3d,
        n_b=n_b,
        delta_mu=delta_mu,
        delta_o=delta_o,
        Omega_p=Omega_p,
        Omega_mu=Omega_mu,
        Omega_o=Omega_o
    )
    L_mtx[0,:] = np.identity(3)[flatten]
    b = np.zeros(9)
    b[0] = 1
    rho_real = np.linalg.solve(L_mtx, b)
    rho = RtoC_num @ rho_real
    rho = rho[unflatten]
```

```
391     return rho
392
393 def rho_linear_steady_state_components(gamma_12, gamma_13, gamma_23, gamma_2d,
        gamma_3d,
394                                        n_b, delta_mu, delta_o, Omega_p, sys):
395     L0_mtx = L0_real_func[sys](
396         gamma_12=gamma_12,
397         gamma_13=gamma_13,
398         gamma_23=gamma_23,
399         gamma_2d=gamma_2d,
400         gamma_3d=gamma_3d,
401         n_b=n_b,
402         delta_mu=delta_mu,
403         delta_o=delta_o,
404         Omega_p=Omega_p
405     )
406     L0_mtx[0,:] = np.identity(3)[flatten]
407
408     b = np.zeros(9)
409     b[0] = 1
410     rho_0_real = np.linalg.solve(L0_mtx, b)
411     rho_0 = RtoC_num @ rho_0_real
412
413     b = -Lor_real_num[sys] @ rho_0_real
414     b[0] = 0
415     rho_or_real = np.linalg.solve(L0_mtx, b)
416     rho_or = RtoC_num @ rho_or_real
417
418     b = -Loi_real_num[sys] @ rho_0_real
419     b[0] = 0
420     rho_oi_real = np.linalg.solve(L0_mtx, b)
421     rho_oi = RtoC_num @ rho_oi_real
422
423     b = -Lmur_real_num[sys] @ rho_0_real
424     b[0] = 0
425     rho_mur_real = np.linalg.solve(L0_mtx, b)
426     rho_mur = RtoC_num @ rho_mur_real
427
428     b = -Lmui_real_num[sys] @ rho_0_real
429     b[0] = 0
430     rho_mui_real = np.linalg.solve(L0_mtx, b)
431     rho_mui = RtoC_num @ rho_mui_real
432
433     rho_0 = rho_0[unflatten]
434     rho_or = rho_or[unflatten]
435     rho_oi = rho_oi[unflatten]
436     rho_mur = rho_mur[unflatten]
437     rho_mui = rho_mui[unflatten]
438     return rho_0, rho_or, rho_oi, rho_mur, rho_mui
439
440 def rho_steady_state_ensemble(gamma_12, gamma_13, gamma_23, gamma_2d, gamma_3d,
441                               n_b, delta_mu, delta_o, Omega_p, Omega_mu,
442                               Omega_o, sigma_mu, sigma_o, sys, logging=False):
443     gauss_lobatto_order = 20
444     gamma_oh = gamma_2d + (gamma_3d if sys=='Lambda' else 0)
445     gamma_muh = gamma_2d + (gamma_3d if sys=='V' else 0)
446
447     # distribution envelope
448     G_mu = stats.norm(loc=delta_mu, scale=sigma_mu).pdf
449     G_o = stats.norm(loc=delta_o, scale=sigma_o).pdf
450     G = lambda dp_mu, dp_o: G_mu(dp_mu) * G_o(dp_o)
451
452     # set up for curve finding
453     Delta_coeffs_2 = Delta_poly_coeffs_func[sys](
454         Omega_p=Omega_p,
455         Omega_mu=Omega_mu,
```

```
456             Omega_o=Omega_o
457         )
458         dmu_Delta_coeffs_2 = poly_bivariate_coeffs_diff_left(Delta_coeffs_2)
459         do_Delta_coeffs_2 = poly_bivariate_coeffs_diff_right(Delta_coeffs_2)
460 
461         # get integral points
462 
463         node_deltap_o = np.array([], dtype=float)
464         node_deltap_mu = np.array([], dtype=float)
465         node_weight = np.array([], dtype=float)
466 
467         # set up outer integral
468         deltap_o_intervals = [delta_o + z*sigma_o
469                               for z in (-10, -3, -1, 0, 1, 3, 10)]
470         deltap_o_intervals += [z*gamma_oh for z in (-5, -1, 0, 1, 5)]
471         deltap_o_intervals = sorted(deltap_o_intervals)
472         deltap_o_nodes, deltap_o_weights = composite_gauss_lobatto_nodes_weights(
473             gauss_lobatto_order, deltap_o_intervals)
474 
475         # set up inner integral
476         for dp_o, w_o in zip(deltap_o_nodes, deltap_o_weights):
477             dmu_Delta_coeffs = poly_bivariate_coeffs_evaluate_right(dmu_Delta_coeffs_2,
        dp_o)
478             do_Delta_coeffs = poly_bivariate_coeffs_evaluate_right(do_Delta_coeffs_2,
        dp_o)
479             dmu_critical = np.concatenate([
480                 poly_coeffs_roots(dmu_Delta_coeffs),
481                 poly_coeffs_roots(do_Delta_coeffs)
482             ])
483 
484             deltap_mu_intervals = [delta_mu + z*sigma_mu
485                                    for z in (-10, -3, -1, 1, 3, 10)]
486             deltap_mu_intervals += [z*gamma_muh for z in (-5, -1, 0, 1, 5)]
487             deltap_mu_intervals += list(dmu_critical)
488 
489             deltap_mu_intervals = sorted(deltap_mu_intervals)
490             deltap_mu_nodes, deltap_mu_weights = composite_gauss_lobatto_nodes_weights(
491                 gauss_lobatto_order, deltap_mu_intervals)
492 
493             node_deltap_o = np.concatenate([node_deltap_o, np.full_like(deltap_mu_nodes,
        dp_o)])
494             node_deltap_mu = np.concatenate([node_deltap_mu, deltap_mu_nodes])
495             node_weight = np.concatenate([node_weight, w_o*deltap_mu_weights])
496 
497         node_G = G(node_deltap_mu, node_deltap_o)
498 
499         # perform integral
500 
501         integral = np.zeros((3,3), dtype=complex)
502         for dp_o, dp_mu, w, g in zip(node_deltap_o, node_deltap_mu, node_weight, node_G):
503             integral += w*g * rho_steady_state(
504                 gamma_12=gamma_12,
505                 gamma_13=gamma_13,
506                 gamma_23=gamma_23,
507                 gamma_2d=gamma_2d,
508                 gamma_3d=gamma_3d,
509                 n_b=n_b,
510                 delta_mu=dp_mu,
511                 delta_o=dp_o,
512                 Omega_p=Omega_p,
513                 Omega_mu=Omega_mu,
514                 Omega_o=Omega_o,
515                 sys=sys
516             )
517 
518         if logging:
```

```
519         return (node_deltap_mu, node_deltap_o, node_weight, node_G), integral
520     else:
521         return integral
522 
523 def Omega_cavity(g, alpha):
524     return g * np.exp(1j*np.angle(alpha)) * np.sqrt(np.abs(alpha)**2 + 1)
525 
526 def cavity_langevin_diff(gamma_12, gamma_13, gamma_23, gamma_2d, gamma_3d,
527                          n_b, gamma_oi, gamma_oc, gamma_mui, gamma_muc,
528                          N_o, N_mu, g_o, g_mu, Omega_p, alpha, beta,
529                          alpha_in, beta_in, delta_mu, delta_o, delta_co,
530                          delta_cmu, sigma_mu, sigma_o, sys, return_S=False):
531     S = rho_steady_state_ensemble(
532         gamma_12=gamma_12,
533         gamma_13=gamma_13,
534         gamma_23=gamma_23,
535         gamma_2d=gamma_2d,
536         gamma_3d=gamma_3d,
537         n_b=n_b,
538         delta_mu=delta_mu,
539         delta_o=delta_o,
540         Omega_p=Omega_p,
541         Omega_o=Omega_cavity(g_o, alpha),
542         Omega_mu=Omega_cavity(g_mu, beta),
543         sigma_o=sigma_o,
544         sigma_mu=sigma_mu,
545         sys=sys
546     )
547     S_alpha = N_o*np.conj(g_o) * (S[1,0] if sys=='V' else S[2,1])
548     S_beta = N_mu*np.conj(g_mu) * (S[2,1] if sys=='V' else S[1,0])
549 
550     d_alpha_S = -1j*S_alpha
551     d_beta_S = -1j*S_beta
552     d_alpha_not_S = -1j*delta_co*alpha - (gamma_oi+gamma_oc)*alpha/2 +
    np.sqrt(gamma_oc)*alpha_in
553     d_beta_not_S = -1j*delta_cmu*beta - (gamma_mui+gamma_muc)*beta/2 +
    np.sqrt(gamma_muc)*beta_in
554     d_alpha = d_alpha_S + d_alpha_not_S
555     d_beta = d_beta_S + d_beta_not_S
556 
557     if return_S:
558         return d_alpha_S, d_beta_S, d_alpha_not_S, d_beta_not_S
559     else:
560         return d_alpha, d_beta
561 
562 def cavity_steady_state(gamma_12, gamma_13, gamma_23, gamma_2d, gamma_3d,
563                         n_b, gamma_oi, gamma_oc, gamma_mui, gamma_muc,
564                         N_o, N_mu, g_o, g_mu, Omega_p, alpha_in, beta_in,
565                         delta_mu, delta_o, delta_co, delta_cmu, sigma_mu,
566                         sigma_o, sys, alpha_0=1, beta_0=1):
567     def pack_vector(alpha, beta):
568         vec = np.zeros(4)
569         vec[0] = np.real(alpha)
570         vec[1] = np.imag(alpha)
571         vec[2] = np.real(beta)
572         vec[3] = np.imag(beta)
573         return vec
574 
575     def unpack_vector(vec):
576         alpha_r = vec[0]
577         alpha_i = vec[1]
578         beta_r = vec[2]
579         beta_i = vec[3]
580         alpha = alpha_r + 1j*alpha_i
581         beta = beta_r + 1j*beta_i
582         return alpha, beta
```

```
583 |
584 |     def diff_func(vec):
585 |         alpha, beta = unpack_vector(vec)
586 |         d_alpha, d_beta = cavity_langevin_diff(
587 |             gamma_12=gamma_12,
588 |             gamma_13=gamma_13,
589 |             gamma_23=gamma_23,
590 |             gamma_2d=gamma_2d,
591 |             gamma_3d=gamma_3d,
592 |             n_b=n_b,
593 |             gamma_oi=gamma_oi,
594 |             gamma_oc=gamma_oc,
595 |             gamma_mui=gamma_mui,
596 |             gamma_muc=gamma_muc,
597 |             N_o=N_o,
598 |             N_mu=N_mu,
599 |             g_o=g_o,
600 |             g_mu=g_mu,
601 |             Omega_p=Omega_p,
602 |             alpha=alpha,
603 |             beta=beta,
604 |             alpha_in=alpha_in,
605 |             beta_in=beta_in,
606 |             delta_mu=delta_mu,
607 |             delta_o=delta_o,
608 |             delta_co=delta_co,
609 |             delta_cmu=delta_cmu,
610 |             sigma_mu=sigma_mu,
611 |             sigma_o=sigma_o,
612 |             sys=sys
613 |         )
614 |
615 |         d_vec = pack_vector(d_alpha, d_beta)
616 |         return d_vec
617 |
618 |     v0 = pack_vector(alpha_0, beta_0)
619 |     result = optimize.root(diff_func, v0)#, method='broyden1',
    |     options={'ftol':1e-12})
620 |     return result
621 |
622 | def planck_excitation(T, omega):
623 |     return 1 / np.expm1(hbar*omega/(kB*T))
624 |
625 | if __name__ == '__main__':
626 |     sys = 'Lambda'
627 |     omega_12 = 2*np.pi*5.186e9
628 |     d13 = 1.63e-32
629 |     d23 = 1.15e-32
630 |     tau_12 = 11
631 |     tau_3 = 0.011
632 |     gamma_2d = 1e6
633 |     gamma_3d = 1e6
634 |     sigma_o = 2*np.pi*419e6
635 |     sigma_mu = 2*np.pi*5e6
636 |     N = 1e16
637 |     gamma_oi = 2*np.pi*7.95e6
638 |     gamma_oc = 2*np.pi*1.7e6
639 |     gamma_mui = 2*np.pi*650e3
640 |     gamma_muc = 2*np.pi*1.5e6
641 |     g_o = 51.9
642 |     g_mu = 1.04
643 |
644 |     T = 4.6
645 |     n_b = planck_excitation(T, omega_12)
646 |     tau_13 = tau_3 * d13**2 / (d13**2 + d23**2)
647 |     tau_23 = tau_3 * d23**2 / (d13**2 + d23**2)
```

```
    gamma_12 = 1 / (tau_12*(n_b+1))
    gamma_13 = 1 / tau_13
    gamma_23 = 1 / tau_23
    N_o = N
    N_mu = N

    Omega_p = 35000.0

    delta_o_small = -100e3
    delta_mu_small = 1e6
    delta_o_large = -6.5*sigma_o
    delta_mu_large = 8*sigma_mu

    iterator = [(delta_o_small, delta_mu_small, 'Small detuning'),
                (delta_o_large, delta_mu_large, 'Large detuning')]
    for delta_o, delta_mu, regime in iterator:
        print(f'\n{regime} regime')
        result = cavity_steady_state(
            gamma_12=gamma_12,
            gamma_13=gamma_13,
            gamma_23=gamma_23,
            gamma_2d=gamma_2d,
            gamma_3d=gamma_3d,
            n_b=n_b,
            gamma_oi=gamma_oi,
            gamma_oc=gamma_oc,
            gamma_mui=gamma_mui,
            gamma_muc=gamma_muc,
            N_o=N_o,
            N_mu=N_mu,
            g_o=g_o,
            g_mu=g_mu,
            Omega_p=Omega_p,
            alpha_in=0.0,
            beta_in=0.0,
            delta_mu=delta_mu,
            delta_o=delta_o,
            delta_co=0.0,
            delta_cmu=0.0,
            sigma_mu=sigma_mu,
            sigma_o=sigma_o,
            sys=sys,
            alpha_0=1,
            beta_0=1
        )
        print(result)
```

C.3.2 Super-Atom Dynamics

This CUDA code implements the super-atom dynamical model. When compiled and ran, it performs a simulation and saves the results as binary files in the working directory.

```
// compile: nvcc biphoton_super_atom.cu -o sim -O3 -rdc=true -lm -arch=sm_60
#include <cuda.h>
#include <cuda_runtime.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define PI 3.14159265358979323846264338327950
const double HBAR = 1.054571817e-34;
const double K_B = 1.380649e-23;

#define NUM_THREAD_BLOCKS 256
```

```
14 #define NUM_THREADS_IN_BLOCK 512
15
16 enum SystemType
17 {
18     LAMBDA_SYSTEM,
19     V_SYSTEM
20 };
21 typedef enum SystemType SystemType;
22
23 struct DensityMatrix
24 {
25     double r11;
26     double r22;
27     double r33;
28     double r12;
29     double i12;
30     double r13;
31     double i13;
32     double r23;
33     double i23;
34 };
35 typedef struct DensityMatrix DensityMatrix;
36
37 const DensityMatrix GROUND_STATE_MATRIX = {
38     1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
39 };
40
41 struct AtomSample
42 {
43     double weight;
44     double g_or;
45     double g_oi;
46     double g_mur;
47     double g_mui;
48     double delta_12;
49     double delta_23;
50 };
51 typedef struct AtomSample AtomSample;
52
53 struct SystemState
54 {
55     double alpha_r;
56     double alpha_i;
57     double beta_r;
58     double beta_i;
59     DensityMatrix rho[];
60 };
61 typedef struct SystemState SystemState;
62
63 size_t SizeofStateStruct(size_t nAtomSamples)
64 {
65     return sizeof(SystemState) + nAtomSamples * sizeof(DensityMatrix);
66 }
67
68 bool HostAllocateStateStruct(SystemState **ptr, size_t nAtomSamples)
69 {
70     *ptr = (SystemState *) malloc(SizeofStateStruct(nAtomSamples));
71     return (*ptr != NULL);
72 }
73
74 bool DeviceAllocateStateStruct(SystemState **ptr, size_t nAtomSamples)
75 {
76     cudaError_t result = cudaMalloc(ptr, SizeofStateStruct(nAtomSamples));
77     return (result == cudaSuccess);
78 }
79
```

```
80 void CopyStateStruct(SystemState *dst, const SystemState *src,
81                       size_t nAtomSamples, cudaMemcpyKind kind)
82 {
83     cudaMemcpy(dst, src, SizeofStateStruct(nAtomSamples), kind);
84 }
85
86 bool HostAllocateAtomSamples(AtomSample **ptr, size_t nAtomSamples)
87 {
88     size_t size = nAtomSamples * sizeof(AtomSample);
89     *ptr = (AtomSample *) malloc(size);
90     return (*ptr != NULL);
91 }
92
93 bool DeviceAllocateAtomSamples(AtomSample **ptr, size_t nAtomSamples)
94 {
95     size_t size = nAtomSamples * sizeof(AtomSample);
96     cudaError_t result = cudaMalloc(ptr, size);
97     return (result == cudaSuccess);
98 }
99
100 void CopyAtomSamples(AtomSample *dst, const AtomSample *src,
101                      size_t nAtomSamples, cudaMemcpyKind kind)
102 {
103     cudaMemcpy(dst, src, nAtomSamples * sizeof(AtomSample), kind);
104 }
105
106 __device__
107 void MasterDerivative(DensityMatrix *diff, SystemType sys,
108     const DensityMatrix *rho, double Omega_mur, double Omega_mui,
109     double Omega_or, double Omega_oi, double Omega_pr, double Omega_pi,
110     double deltap_mu, double deltap_o, double n_b, double gamma_12,
111     double gamma_13, double gamma_23, double gamma_2d, double gamma_3d)
112 {
113     DensityMatrix tmp = *rho;
114
115     // code generated using computer algebra
116     if (sys == LAMBDA_SYSTEM)
117     {
118         diff->r11 = -2*Omega_mui*tmp.r12 - 2*Omega_mur*tmp.i12 - 2*Omega_pi*tmp.r13
    - 2*Omega_pr*tmp.i13 - gamma_12*n_b*tmp.r11 + gamma_12*tmp.r22*(n_b + 1) +
    gamma_13*tmp.r33;
119         diff->r22 = 2*Omega_mui*tmp.r12 + 2*Omega_mur*tmp.i12 - 2*Omega_oi*tmp.r23 -
    2*Omega_or*tmp.i23 + gamma_12*n_b*tmp.r11 - gamma_12*tmp.r22*(n_b + 1) +
    gamma_23*tmp.r33;
120         diff->r33 = 2*Omega_oi*tmp.r23 + 2*Omega_or*tmp.i23 + 2*Omega_pi*tmp.r13 +
    2*Omega_pr*tmp.i13 - gamma_13*tmp.r33 - gamma_23*tmp.r33;
121         diff->r12 = Omega_mui*tmp.r11 - Omega_mui*tmp.r22 - Omega_oi*tmp.r13 -
    Omega_or*tmp.i13 - Omega_pi*tmp.r23 - Omega_pr*tmp.i23 - deltap_mu*tmp.i12 -
    1.0/2.0*gamma_12*n_b*tmp.r12 - 1.0/2.0*gamma_12*tmp.r12*(n_b + 1) -
    1.0/2.0*gamma_2d*tmp.r12;
122         diff->i12 = Omega_mur*tmp.r11 - Omega_mur*tmp.r22 - Omega_oi*tmp.i13 +
    Omega_or*tmp.r13 + Omega_pi*tmp.i23 - Omega_pr*tmp.r23 + deltap_mu*tmp.r12 -
    1.0/2.0*gamma_12*n_b*tmp.i12 - 1.0/2.0*gamma_12*tmp.i12*(n_b + 1) -
    1.0/2.0*gamma_2d*tmp.i12;
123         diff->r13 = -Omega_mui*tmp.r23 + Omega_mur*tmp.i23 + Omega_oi*tmp.r12 -
    Omega_or*tmp.i12 + Omega_pi*tmp.r11 - Omega_pi*tmp.r33 -
    1.0/2.0*gamma_12*n_b*tmp.r13 - 1.0/2.0*gamma_13*tmp.r13 -
    1.0/2.0*gamma_23*tmp.r13 - 1.0/2.0*gamma_3d*tmp.r13 + tmp.i13*(-deltap_mu -
    deltap_o);
124         diff->i13 = -Omega_mui*tmp.i23 - Omega_mur*tmp.r23 + Omega_oi*tmp.i12 +
    Omega_or*tmp.r12 + Omega_pr*tmp.r11 - Omega_pr*tmp.r33 -
    1.0/2.0*gamma_12*n_b*tmp.i13 - 1.0/2.0*gamma_13*tmp.i13 -
    1.0/2.0*gamma_23*tmp.i13 - 1.0/2.0*gamma_3d*tmp.i13 - tmp.r13*(-deltap_mu -
    deltap_o);
125         diff->r23 = Omega_mui*tmp.r13 + Omega_mur*tmp.i13 + Omega_oi*tmp.r22 -
    Omega_oi*tmp.r33 + Omega_pi*tmp.r12 + Omega_pr*tmp.i12 + deltap_mu*tmp.i23 -
```

```
    1.0/2.0*gamma_12*tmp.r23*(n_b + 1) - 1.0/2.0*gamma_13*tmp.r23 -
    1.0/2.0*gamma_23*tmp.r23 - 1.0/2.0*gamma_2d*tmp.r23 - 1.0/2.0*gamma_3d*tmp.r23 +
    tmp.i23*(-deltap_mu - deltap_o);
126         diff->i23 = Omega_mui*tmp.i13 - Omega_mur*tmp.r13 + Omega_or*tmp.r22 -
    Omega_or*tmp.r33 - Omega_pi*tmp.i12 + Omega_pr*tmp.r12 - deltap_mu*tmp.r23 -
    1.0/2.0*gamma_12*tmp.i23*(n_b + 1) - 1.0/2.0*gamma_13*tmp.i23 -
    1.0/2.0*gamma_23*tmp.i23 - 1.0/2.0*gamma_2d*tmp.i23 - 1.0/2.0*gamma_3d*tmp.i23 -
    tmp.r23*(-deltap_mu - deltap_o);
127     }
128     else // sys == V_SYSTEM
129     {
130         diff->r11 = -2*Omega_oi*tmp.r12 - 2*Omega_or*tmp.i12 - 2*Omega_pi*tmp.r13 -
    2*Omega_pr*tmp.i13 + gamma_12*tmp.r22 + gamma_13*tmp.r33;
131         diff->r22 = -2*Omega_mui*tmp.r23 - 2*Omega_mur*tmp.i23 + 2*Omega_oi*tmp.r12
    + 2*Omega_or*tmp.i12 - gamma_12*tmp.r22 - gamma_23*n_b*tmp.r22 +
    gamma_23*tmp.r33*(n_b + 1);
132         diff->r33 = 2*Omega_mui*tmp.r23 + 2*Omega_mur*tmp.i23 + 2*Omega_pi*tmp.r13 +
    2*Omega_pr*tmp.i13 - gamma_13*tmp.r33 + gamma_23*n_b*tmp.r22 -
    gamma_23*tmp.r33*(n_b + 1);
133         diff->r12 = -Omega_mui*tmp.r13 - Omega_mur*tmp.i13 + Omega_oi*tmp.r11 -
    Omega_oi*tmp.r22 - Omega_pi*tmp.r23 - Omega_pr*tmp.i23 - deltap_o*tmp.i12 -
    1.0/2.0*gamma_12*tmp.r12 - 1.0/2.0*gamma_23*n_b*tmp.r12 -
    1.0/2.0*gamma_2d*tmp.r12;
134         diff->i12 = -Omega_mui*tmp.i13 + Omega_mur*tmp.r13 + Omega_or*tmp.r11 -
    Omega_or*tmp.r22 + Omega_pi*tmp.i23 - Omega_pr*tmp.r23 + deltap_o*tmp.r12 -
    1.0/2.0*gamma_12*tmp.i12 - 1.0/2.0*gamma_23*n_b*tmp.i12 -
    1.0/2.0*gamma_2d*tmp.i12;
135         diff->r13 = Omega_mui*tmp.r12 - Omega_mur*tmp.i12 - Omega_oi*tmp.r23 +
    Omega_or*tmp.i23 + Omega_pi*tmp.r11 - Omega_pi*tmp.r33 -
    1.0/2.0*gamma_13*tmp.r13 - 1.0/2.0*gamma_23*tmp.r13*(n_b + 1) -
    1.0/2.0*gamma_3d*tmp.r13 + tmp.i13*(-deltap_mu - deltap_o);
136         diff->i13 = Omega_mui*tmp.i12 + Omega_mur*tmp.r12 - Omega_oi*tmp.i23 -
    Omega_or*tmp.r23 + Omega_pr*tmp.r11 - Omega_pr*tmp.r33 -
    1.0/2.0*gamma_13*tmp.i13 - 1.0/2.0*gamma_23*tmp.i13*(n_b + 1) -
    1.0/2.0*gamma_3d*tmp.i13 - tmp.r13*(-deltap_mu - deltap_o);
137         diff->r23 = Omega_mui*tmp.r22 - Omega_mui*tmp.r33 + Omega_oi*tmp.r13 +
    Omega_or*tmp.i13 + Omega_pi*tmp.r12 + Omega_pr*tmp.i12 + deltap_o*tmp.i23 -
    1.0/2.0*gamma_12*tmp.r23 - 1.0/2.0*gamma_13*tmp.r23 -
    1.0/2.0*gamma_23*n_b*tmp.r23 - 1.0/2.0*gamma_23*tmp.r23*(n_b + 1) -
    1.0/2.0*gamma_2d*tmp.r23 - 1.0/2.0*gamma_3d*tmp.r23 + tmp.i23*(-deltap_mu -
    deltap_o);
138         diff->i23 = Omega_mur*tmp.r22 - Omega_mur*tmp.r33 + Omega_oi*tmp.i13 -
    Omega_or*tmp.r13 - Omega_pi*tmp.i12 + Omega_pr*tmp.r12 - deltap_o*tmp.r23 -
    1.0/2.0*gamma_12*tmp.i23 - 1.0/2.0*gamma_13*tmp.i23 -
    1.0/2.0*gamma_23*n_b*tmp.i23 - 1.0/2.0*gamma_23*tmp.i23*(n_b + 1) -
    1.0/2.0*gamma_2d*tmp.i23 - 1.0/2.0*gamma_3d*tmp.i23 - tmp.r23*(-deltap_mu -
    deltap_o);
139     }
140 }
141
142 __device__
143 void Omega(double *Omega_r, double *Omega_i,
144            double g_r, double g_i,
145            double alpha_r, double alpha_i)
146 {
147
148     double abs_alpha_sqr = alpha_r*alpha_r + alpha_i*alpha_i;
149     if (abs_alpha_sqr == 0.0)
150     {
151         *Omega_r = g_r;
152         *Omega_i = g_i;
153         return;
154     }
155
156     double alpha_rescale = sqrt(1.0 + 1.0/abs_alpha_sqr);
157     alpha_r *= alpha_rescale;
```

```
158     alpha_i *= alpha_rescale;
159
160     *Omega_r = g_r*alpha_r - g_i*alpha_i;
161     *Omega_i = g_r*alpha_i + g_i*alpha_r;
162 }
163
164 __global__
165 void EnsembleDerivativeAndSum(size_t nAtomSamples, SystemState *diff,
166     const SystemState *state, const AtomSample *atomSamples, SystemType sys,
167     double alpha_r, double alpha_i, double beta_r, double beta_i,
168     double Omega_pr, double Omega_pi, double delta_mu, double delta_o,
169     double n_b, double gamma_12, double gamma_13,
170     double gamma_23, double gamma_2d, double gamma_3d)
171 {
172     size_t threadId = blockIdx.x*blockDim.x + threadIdx.x;
173     size_t threadCount = gridDim.x*blockDim.x;
174     for (size_t k = threadId; k < nAtomSamples; k += threadCount)
175     {
176         // get sample-specific variables
177         double g_or = atomSamples[k].g_or;
178         double g_oi = atomSamples[k].g_oi;
179         double g_mur = atomSamples[k].g_mur;
180         double g_mui = atomSamples[k].g_mui;
181         double delta_12 = atomSamples[k].delta_12;
182         double delta_23 = atomSamples[k].delta_23;
183         double w = atomSamples[k].weight;
184         const DensityMatrix *rho = &(state->rho[k]);
185         DensityMatrix *rhoDiff = &(diff->rho[k]);
186
187         // contribution to ensemble terms
188         double rho_or = ((sys == V_SYSTEM) ? rho->r12 : rho->r23);
189         double rho_oi = ((sys == V_SYSTEM) ? -rho->i12 : -rho->i23);
190         double rho_mur = ((sys == V_SYSTEM) ? rho->r23 : rho->r12);
191         double rho_mui = ((sys == V_SYSTEM) ? -rho->i23 : -rho->i12);
192         double S_alpha_r = w * (g_or*rho_or + g_oi*rho_oi);
193         double S_alpha_i = w * (g_oi*rho_or - g_or*rho_oi);
194         double S_beta_r = w * (g_mur*rho_mur + g_mui*rho_mui);
195         double S_beta_i = w * (g_mui*rho_mur - g_mur*rho_mui);
196         atomicAdd(&(diff->alpha_r), S_alpha_i);
197         atomicAdd(&(diff->alpha_i), -S_alpha_r);
198         atomicAdd(&(diff->beta_r), S_beta_i);
199         atomicAdd(&(diff->beta_i), -S_beta_r);
200
201         // drive Rabi frequencies
202         double Omega_mur;
203         double Omega_mui;
204         double Omega_or;
205         double Omega_oi;
206         Omega(&Omega_mur, &Omega_mui, g_mur, g_mui, beta_r, beta_i);
207         Omega(&Omega_or, &Omega_oi, g_or, g_oi, alpha_r, alpha_i);
208
209         // inhomogeneous shift
210         double deltap_mu = delta_mu + (sys == V_SYSTEM ? delta_23 : delta_12);
211         double deltap_o = delta_o + (sys == V_SYSTEM ? delta_12 : delta_23);
212
213         MasterDerivative(
214             rhoDiff,
215             sys,
216             rho,
217             Omega_mur,
218             Omega_mui,
219             Omega_or,
220             Omega_oi,
221             Omega_pr,
222             Omega_pi,
223             deltap_mu,
```

```
            deltap_o,
            n_b,
            gamma_12,
            gamma_13,
            gamma_23,
            gamma_2d,
            gamma_3d);
    }
}

void SystemDerivative(size_t nAtomSamples,
    SystemState *diff, const SystemState *state,
    const AtomSample *atomSamples, SystemType sys,
    double Omega_pr, double Omega_pi, double delta_mu, double delta_o,
    double n_b, double gamma_12, double gamma_13, double gamma_23,
    double gamma_2d, double gamma_3d, double gamma_o, double gamma_mu)
{
    double alpha_r;
    double alpha_i;
    double beta_r;
    double beta_i;
    cudaMemcpy(&alpha_r, &(state->alpha_r),
        sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&alpha_i, &(state->alpha_i),
        sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&beta_r, &(state->beta_r),
        sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(&beta_i, &(state->beta_i),
        sizeof(double), cudaMemcpyDeviceToHost);

    double d_alpha_r = -alpha_r * gamma_o/2.0;
    double d_alpha_i = -alpha_i * gamma_o/2.0;
    double d_beta_r = -beta_r * gamma_mu/2.0;
    double d_beta_i = -beta_i * gamma_mu/2.0;
    cudaMemcpy(&(diff->alpha_r), &d_alpha_r,
        sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(&(diff->alpha_i), &d_alpha_i,
        sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(&(diff->beta_r), &d_beta_r,
        sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(&(diff->beta_i), &d_beta_i,
        sizeof(double), cudaMemcpyHostToDevice);

    EnsembleDerivativeAndSum<<<NUM_THREAD_BLOCKS,NUM_THREADS_IN_BLOCK>>>(
        nAtomSamples,
        diff,
        state,
        atomSamples,
        sys,
        alpha_r,
        alpha_i,
        beta_r,
        beta_i,
        Omega_pr,
        Omega_pi,
        delta_mu,
        delta_o,
        n_b,
        gamma_12,
        gamma_13,
        gamma_23,
        gamma_2d,
        gamma_3d);
    cudaDeviceSynchronize();
}
```

```
__global__
void GlobalDensityMatricesWeightedSum(size_t nAtomSamples,
    DensityMatrix *dst, const DensityMatrix *a, double aw,
    const DensityMatrix *b, double bw)
{
    size_t threadId = blockIdx.x*blockDim.x + threadIdx.x;
    size_t threadCount = gridDim.x*blockDim.x;
    for (size_t k = threadId; k < nAtomSamples; k += threadCount)
    {
        dst[k].r11 = aw*a[k].r11 + bw*b[k].r11;
        dst[k].r22 = aw*a[k].r22 + bw*b[k].r22;
        dst[k].r33 = aw*a[k].r33 + bw*b[k].r33;
        dst[k].r12 = aw*a[k].r12 + bw*b[k].r12;
        dst[k].i12 = aw*a[k].i12 + bw*b[k].i12;
        dst[k].r13 = aw*a[k].r13 + bw*b[k].r13;
        dst[k].i13 = aw*a[k].i13 + bw*b[k].i13;
        dst[k].r23 = aw*a[k].r23 + bw*b[k].r23;
        dst[k].i23 = aw*a[k].i23 + bw*b[k].i23;
    }
}

__global__
void GlobalCavityStateWeightedSum(size_t nAtomSamples,
    SystemState *dst, const SystemState *a, double aw,
    const SystemState *b, double bw)
{
    dst->alpha_r = aw*a->alpha_r + bw*b->alpha_r;
    dst->alpha_i = aw*a->alpha_i + bw*b->alpha_i;
    dst->beta_r = aw*a->beta_r + bw*b->beta_r;
    dst->beta_i = aw*a->beta_i + bw*b->beta_i;
}

void SystemStateWeightedSum(size_t nAtomSamples, SystemState *dst,
    const SystemState *a, double aw, const SystemState *b, double bw)
{
    GlobalCavityStateWeightedSum<<<1,1>>>(nAtomSamples, dst, a, aw, b, bw);
    cudaDeviceSynchronize();
    GlobalDensityMatricesWeightedSum<<<NUM_THREAD_BLOCKS,
        NUM_THREADS_IN_BLOCK>>>(nAtomSamples,
        dst->rho, a->rho, aw, b->rho, bw);
    cudaDeviceSynchronize();
}

void SystemRK4Step(size_t nAtomSamples, SystemState *__restrict__ state,
    const AtomSample *__restrict__ atomSamples,
    SystemState *__restrict__ tmp1, SystemState *__restrict__ tmp2,
    SystemType sys, double dt, double Omega_pr, double Omega_pi,
    double delta_mu, double delta_o, double n_b, double gamma_12,
    double gamma_13, double gamma_23, double gamma_2d,
    double gamma_3d, double gamma_o, double gamma_mu)
{
    SystemDerivative( // k1
        nAtomSamples,
        tmp1,
        state,
        atomSamples,
        sys,
        Omega_pr,
        Omega_pi,
        delta_mu,
        delta_o,
        n_b,
        gamma_12,
        gamma_13,
        gamma_23,
        gamma_2d,
```

```
356         gamma_3d,
357         gamma_o,
358         gamma_mu);
359     SystemStateWeightedSum(nAtomSamples, tmp2, state, 1.0, tmp1, dt/6.0);
360     SystemStateWeightedSum(nAtomSamples, tmp1, state, 1.0, tmp1, dt/2.0);
361     SystemDerivative( // k2
362         nAtomSamples,
363         tmp1,
364         tmp1,
365         atomSamples,
366         sys,
367         Omega_pr,
368         Omega_pi,
369         delta_mu,
370         delta_o,
371         n_b,
372         gamma_12,
373         gamma_13,
374         gamma_23,
375         gamma_2d,
376         gamma_3d,
377         gamma_o,
378         gamma_mu);
379     SystemStateWeightedSum(nAtomSamples, tmp2, tmp1, dt/3.0, tmp2, 1.0);
380     SystemStateWeightedSum(nAtomSamples, tmp1, state, 1.0, tmp1, dt/2.0);
381     SystemDerivative( // k3
382         nAtomSamples,
383         tmp1,
384         tmp1,
385         atomSamples,
386         sys,
387         Omega_pr,
388         Omega_pi,
389         delta_mu,
390         delta_o,
391         n_b,
392         gamma_12,
393         gamma_13,
394         gamma_23,
395         gamma_2d,
396         gamma_3d,
397         gamma_o,
398         gamma_mu);
399     SystemStateWeightedSum(nAtomSamples, tmp2, tmp1, dt/3.0, tmp2, 1.0);
400     SystemStateWeightedSum(nAtomSamples, tmp1, state, 1.0, tmp1, dt);
401     SystemDerivative( // k4
402         nAtomSamples,
403         tmp1,
404         tmp1,
405         atomSamples,
406         sys,
407         Omega_pr,
408         Omega_pi,
409         delta_mu,
410         delta_o,
411         n_b,
412         gamma_12,
413         gamma_13,
414         gamma_23,
415         gamma_2d,
416         gamma_3d,
417         gamma_o,
418         gamma_mu);
419     SystemStateWeightedSum(nAtomSamples, state, tmp1, dt/6.0, tmp2, 1.0);
420 }
421
```

```
422 double PlanckExcitation(double T, double omega)
423 {
424     return 1.0 / expm1(HBAR*omega / (K_B*T));
425 }
426
427 double RandomUniform(void)
428 {
429     return ((double) rand()) / ((double) RAND_MAX);
430 }
431
432 double RandomGaussian(void)
433 {
434     double U1 = RandomUniform();
435     double U2 = RandomUniform();
436
437     // U1 == 0.0 results in an error when taking its log
438     while (U1 == 0.0)
439     {
440         U1 = RandomUniform();
441     }
442
443     double R = sqrt(-2.0 * log(U1));
444     double Theta = 2.0*PI * U2;
445     return R * cos(Theta);
446 }
447
448 int main(void)
449 {
450     // system constants
451     const SystemType sys = LAMBDA_SYSTEM;
452     const double omega_12 = 2.0*PI*5.186e9;
453     const double d13 = 1.63e-32;
454     const double d23 = 1.15e-32;
455     const double tau_12 = 11.0;
456     const double tau_3 = 0.011;
457     const double gamma_2d = 1e6;
458     const double gamma_3d = 1e6;
459     const double sigma_o = 2.0*PI*419e6;
460     const double sigma_mu = 2.0*PI*5e6;
461     const double N = 1e16;
462     const double gamma_oi = 2.0*PI*7.95e6;
463     const double gamma_oc = 2.0*PI*1.7e6;
464     const double gamma_mui = 2.0*PI*650e3;
465     const double gamma_muc = 2.0*PI*1.5e6;
466     const double g_or = 51.9;
467     const double g_oi = 0.0;
468     const double g_mur = 1.04;
469     const double g_mui = 0.0;
470
471     const double T = 4.6;
472     const double n_b = PlanckExcitation(T, omega_12);
473     const double tau_13 = tau_3 * d13*d13 / (d13*d13 + d23*d23);
474     const double tau_23 = tau_3 * d23*d23 / (d13*d13 + d23*d23);
475     const double gamma_12 = 1.0 / (tau_12*(n_b+1.0));
476     const double gamma_13 = 1.0 / tau_13;
477     const double gamma_23 = 1.0 / tau_23;
478
479     const double sigma_12 = ((sys == V_SYSTEM) ? sigma_o : sigma_mu);
480     const double sigma_23 = ((sys == V_SYSTEM) ? sigma_mu : sigma_o);
481
482     // parameters
483     size_t nAtomSamples = 1'000'000;
484     const double weight = N / ((double) nAtomSamples);
485     const double delta_mu = 2.0*sigma_mu;
486     const double delta_o = 2.0*sigma_o;
487     const double Omega_pr = 35000.0;
```

```
488     const double Omega_pi = 0.0;
489     const double alpha_0r = 1.0;
490     const double alpha_0i = 0.0;
491     const double beta_0r = 1.0;
492     const double beta_0i = 0.0;
493
494     // allocate arrays
495     AtomSample *atomSamples;
496     if (!(HostAllocateAtomSamples(&atomSamples, nAtomSamples)))
497     {
498         printf("Host memory allocation failure\n");
499         return -1;
500     }
501
502     AtomSample *deviceAtomSamples;
503     if (!(DeviceAllocateAtomSamples(&deviceAtomSamples, nAtomSamples)))
504     {
505         free(atomSamples);
506         printf("Device memory allocation failure\n");
507         return -1;
508     }
509
510     SystemState *state;
511     if (!(HostAllocateStateStruct(&state, nAtomSamples)))
512     {
513         free(atomSamples);
514         cudaFree(deviceAtomSamples);
515         printf("Host memory allocation failure\n");
516         return -1;
517     }
518
519     SystemState *deviceState;
520     if (!(DeviceAllocateStateStruct(&deviceState, nAtomSamples)))
521     {
522         free(atomSamples);
523         cudaFree(deviceAtomSamples);
524         free(state);
525         printf("Device memory allocation failure\n");
526         return -1;
527     }
528
529     SystemState *tmp1;
530     if (!(DeviceAllocateStateStruct(&tmp1, nAtomSamples)))
531     {
532         free(atomSamples);
533         cudaFree(deviceAtomSamples);
534         free(state);
535         cudaFree(deviceState);
536         printf("Device memory allocation failure\n");
537         return -1;
538     }
539
540     SystemState *tmp2;
541     if (!(DeviceAllocateStateStruct(&tmp2, nAtomSamples)))
542     {
543         free(atomSamples);
544         cudaFree(deviceAtomSamples);
545         free(state);
546         cudaFree(deviceState);
547         cudaFree(tmp1);
548         printf("Device memory allocation failure\n");
549         return -1;
550     }
551
552     // populate arrays and copy to device memory
553     state->alpha_r = alpha_0r;
```

```
554     state->alpha_i = alpha_0i;
555     state->beta_r = beta_0r;
556     state->beta_i = beta_0i;
557 
558     for (size_t k = 0; k < nAtomSamples; k++)
559     {
560         atomSamples[k].g_or = g_or;
561         atomSamples[k].g_oi = g_oi;
562         atomSamples[k].g_mur = g_mur;
563         atomSamples[k].g_mui = g_mui;
564         atomSamples[k].weight = weight;
565         atomSamples[k].delta_12 = sigma_12 * RandomGaussian();
566         atomSamples[k].delta_23 = sigma_23 * RandomGaussian();
567 
568         state->rho[k] = GROUND_STATE_MATRIX;
569     }
570 
571     CopyAtomSamples(deviceAtomSamples, atomSamples, nAtomSamples,
572                     cudaMemcpyHostToDevice);
573     CopyStateStruct(deviceState, state, nAtomSamples, cudaMemcpyHostToDevice);
574 
575     // run simulation and save results to binary files
576     const char *dirname = ".";
577     char atomSamplesFpath[256];
578     sprintf(atomSamplesFpath, "%s/atom_samples", dirname);
579     FILE *atomSamplesFp = fopen(atomSamplesFpath, "wb");
580     fwrite(atomSamples, sizeof(AtomSample), nAtomSamples, atomSamplesFp);
581     printf("Saved atom sample data as %s\n", atomSamplesFpath);
582     fclose(atomSamplesFp);
583 
584     const double dt = 10e-12;
585     size_t numSteps = 0;
586 
587     size_t numPrint = 0;
588     while (true)
589     {
590         size_t toPrint;
591         if (numPrint < 100)
592         {
593             toPrint = numPrint;
594         }
595         else
596         {
597             size_t n = (numPrint-100) / 900;
598             toPrint = (numPrint-100) % 900 + 100;
599             for (size_t i = 0; i < n; i++)
600             {
601                 toPrint *= 10;
602             }
603         }
604 
605         while (numSteps < toPrint)
606         {
607             SystemRK4Step(
608                 nAtomSamples,
609                 deviceState,
610                 deviceAtomSamples,
611                 tmp1,
612                 tmp2,
613                 sys,
614                 dt,
615                 Omega_pr,
616                 Omega_pi,
617                 delta_mu,
618                 delta_o,
619                 n_b,
```

```
                gamma_12,
                gamma_13,
                gamma_23,
                gamma_2d,
                gamma_3d,
                gamma_oi+gamma_oc,
                gamma_mui+gamma_muc);
            numSteps++;
        }


        char fname[256];
        sprintf(fname, "state_step_%zd_dt_%zd_ps",
            numSteps, (size_t) round(dt*1e12));
        char fpath[256];
        sprintf(fpath, "%s/%s", dirname, fname);

        FILE *fp = fopen(fpath, "wb");
        CopyStateStruct(state, deviceState,
            nAtomSamples, cudaMemcpyDeviceToHost);
        fwrite(state, SizeofStateStruct(nAtomSamples), 1, fp);
        fclose(fp);

        printf("\nSaved step %zd as %s\n", numSteps, fpath);
        printf("alpha_r %f\n", state->alpha_r);
        printf("alpha_i %f\n", state->alpha_i);
        printf("beta_r %f\n", state->beta_r);
        printf("beta_i %f\n", state->beta_i);
        printf("rho[0].r11 %f\n", state->rho[0].r11);
        printf("rho[0].r22 %f\n", state->rho[0].r22);
        printf("rho[0].r33 %f\n", state->rho[0].r33);
        printf("rho[0].r12 %f\n", state->rho[0].r12);
        printf("rho[0].i12 %f\n", state->rho[0].i12);
        printf("rho[0].r13 %f\n", state->rho[0].r13);
        printf("rho[0].i13 %f\n", state->rho[0].i13);
        printf("rho[0].r23 %f\n", state->rho[0].r23);
        printf("rho[0].i23 %f\n", state->rho[0].i23);

        numPrint++;
    }

    free(atomSamples);
    cudaFree(deviceAtomSamples);
    free(state);
    cudaFree(deviceState);
    cudaFree(tmp1);
    cudaFree(tmp2);
    return 0;
}
```