Calculating Stationary States of Open Quantum Systems Using a Neural Network Ansatz

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Thomas Bernhard Malthouse

May 2020

Approved for the Division
(Physics)

Darrell F. Schroeter

Acknowledgements

To Nancy—the best quarantine companion I can imagine.

To Darrell—for all the advice and encouragement throughout this process.

To Joel, and Andrew, and Alison, and Lucas, and John, and Mark, and Jay.

To Nick—for being a great friend and person to talk to.

To Shawn and Purna—for the many nights spent laughing in the bio thesis offices.

To Kees—there for so many problem sets.

To the reactor trainees—an absolute highlight of the year.

To Melinda, and Toria, and Ilana, and the rest of the reactor staff.

To Grayson—for the endless thesis wisdom you bestowed last year.

To Claire and Sophia—fiends of Brag House.

To Stephanie—for wine and cheese.

To Sam and Laura—too good for the department.

To M—chad extraordinaire.

To Will—for hambledurgers.

To Beatrice, and Eli, and Maham, and Amelia, and the rest of the physics seniors.

Table of Contents

List of Figures

Abstract

This thesis studies the time-evolution and equilibrium states of open quantum systems—quantum systems that interact with their environmnent, losing information and coherence over time. We study the Lindblad master equation, a differential equation analagous to the Schrödinger equation for open systems, and discuss various methods of numerically solving this equation. Finally, we propose a neural network ansatz to represent the state of an open quantum system, and use the ansatz to find equilibrium states of a simple model.

Dedication

To my parents, for steadfast support.

Introduction

The quantum mechanics taught at the undergraduate level is—like all physics—a sketch of reality. We suppose that we fully understand our system, that the wavefunction $|\psi\rangle$ and potential energy $\hat{H}$ can be written down with full confidence, and use the Schrödinger equation

$$i\hbar\frac{\mathrm{d}}{\mathrm{d}t}|\psi\rangle = \hat{H}|\psi\rangle$$

to calculate the system's time evolution. The theory is—to the best of our knowledge—rock-solid, and perfectly predicts the behavior of simple, self-contained systems.

Problems arise when we try to analyze more complicated systems—especially the delicate, carefully-engineered configurations designed for quantum computing and experimentation. Try as we might, isolating these systems entirely from the universe around them is impossible, and our models need to account for stray photons and other particles flying through and meddling with our system. Doing this with undergraduate-level QM requires that we consider the whole universe as our system, writing down a $|\psi\rangle$ and $\hat{H}$ that describes every particle in existence.

The theory of open quantum systems allows us to approximate the effects of these interactions. If we specify *how* the environment interacts with our system (such as a stray photon flying through a quantum computer, flipping the spin of an electron) and the timescale of that interaction, analyzing the system with the Lindblad equation reveals how the system evolves over time and the rate of decoherence—which can be thought of as information leaking out of the system into the environment.

The Lindblad equation is, in general, not solvable analytically, and numerical methods are required to model the behavior of open systems. Numerical approaches to quantum mechanics are well-studied, with Monte-Carlo and linear algebraic techniques developed for a wide variety of systems. However, these techniques all scale extremely poorly—the difficulty of modeling a quantum system increases exponentially with degrees of freedom. With viable and practical quantum computers relying on dozens or hundreds of particles held in a delicate quantum state, careful approximations are required to model their behavior.

This thesis explores methods to approximate the steady-state equilibria of open quantum systems—the states that, given enough time, environmental interactions cause a system to decay into—using a neural network ansatz and variational quantum Monte-Carlo techniques to lessen the required calculations. This is an active field of research, especially as it applies to open quantum systems—the wealth of recent research on neural networks has enormous spillover effects for physics, and opens

the possibility of solving previously-intractable numerical problems, including the simulation of open quantum systems.

We'll begin by discussing the theory behind open quantum systems, including a discussion of how to incorporate random environmental interactions into a theory of quantum evolution and some simple toy models to work with. We'll then discuss various non-neural techniques for evaluating the Lindblad equation before setting and optimizing up the neural-network ansatz and applying it to some simple systems to demonstrate its functionality.

Chapter 1

Open Quantum Systems

1.1 States and Operators

Quantum mechanics is a theory of states and operators. Systems are described by states (typically denoted as $|\psi\rangle$), and operators act on states, transforming them into a different state. These operators are typically written with a hat—e.g. $\hat{A}$ would denote "operator A". We could then apply this operator to a state, giving

$$|\phi\rangle = \hat{A}\,|\psi\rangle\,.$$

If we specify a basis to describe the state of our system, the action of an operator on a state can be interpreted as a matrix-vector multiplication, returning a new vector:

$$\vec{\phi} = \mathbf{A}\vec{\psi}.$$

This suggests that operators have associated eigenstates—states such that

$$\hat{A}\,|\psi\rangle = \lambda\,|\psi\rangle\,.$$

Eigenstates take on a special significance for *Hermitian* operators. These are operators where the conjugate transpose of the matrix representation $\mathbf{A}^\dagger$ is equal to the original operator's matrix representation: $\mathbf{A}^\dagger = \mathbf{A}$) (which implies purely real eigenvalues), and are related to quantities that are physically observable. For example, the Hamiltonian $\hat{H}$, when applied to an eigenstate, returns the energy of that eigenstate[2]:

$$\hat{H}\,|\psi\rangle = E\,|\psi\rangle$$

A set of operators that will prove very relevant for this thesis are the Pauli operators:

$$\mathbf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \mathbf{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Note that—like matrices—operators do not necessarily commute.

1.2 Unitary Time Evolution

In quantum mechanics, time evolution is unitary. If we perfectly know the state of a system at time t, we can use the time-dependent Schrödinger equation to calculate the state of the system at any other time, past or future. We can model this evolution using an operator:

$$\hat{U}_t \left|\psi(0)\right\rangle = \left|\psi(t)\right\rangle .$$

Being unitary, time evolution is generated by a Hermitian operator[1]—the Hamiltonian. Given a Hamiltonian $\hat{H}$, the stationary states (states that do not evolve in time) can be found by solving the eigenvalue problem

$$\hat{H} \left|\psi_i\right\rangle = E_i \left|\psi_i\right\rangle .$$

If $\left|\psi(0)\right\rangle = \left|\psi_i\right\rangle$—the system is in one of these eigenstates at t_0—then at some future time t, the state will be

$$\hat{U}_t \left|\psi_i\right\rangle \equiv e^{-i\hat{H}t/\hbar} \left|\psi_i\right\rangle = e^{-iE_i t/\hbar} \left|\psi_i\right\rangle$$

so, although the phase of ψ will change over time (with a frequency proportional to its energy), its magnitude will never change, and the probability of measuring that energy will never change.

Because quantum mechanics is a linear theory, this also carries over to arbitrary superpositions of these eigenstates (which, we can show, form a complete orthonormal basis set):

$$\left|\Psi(0)\right\rangle = \sum_i c_i \left|\psi_i\right\rangle \implies \left|\Psi(t)\right\rangle = \hat{U}_t \left|\Psi(0)\right\rangle = \sum_i c_i \hat{U}_t \left|\psi_i\right\rangle = \sum_i c_i e^{-iE_i t/\hbar} \left|\psi_i\right\rangle \quad (1.1)$$

for any arbitrary $\left|\Psi(0)\right\rangle$. Given one of these states, the expectation value of some observable operator $\hat{A}$ is

$$\langle A \rangle = \left\langle \Psi \right| \hat{A} \left| \Psi \right\rangle . \quad (1.2)$$

1.2.1 Density Matrix Representations

In the example above, we've treated states as column vectors—the state $\left|\psi\right\rangle$ can be represented as

$$\left|\psi(t)\right\rangle \doteq \begin{pmatrix} a_1(t) \\ a_2(t) \\ \vdots \\ a_n(t) \end{pmatrix},$$

where $\doteq$ means "represented by, in a specific basis." This is far from the only way to formulate quantum mechanics—we could define the density operator

$$\hat{\rho} \equiv \left|\Psi\right\rangle \left\langle \Psi \right| .$$

[1] For more information on the relationship between unitary operators and the Hermitian operators that generate them, see [15]

In matrix form, the density matrix for a system in state $\vec{\Psi}$ is given by

$$\boldsymbol{\rho} = \vec{\Psi} \otimes \vec{\Psi}^{\dagger}.$$

The density matrix defined in this way has a few useful features:

$$\begin{aligned} \boldsymbol{\rho^2} &= \boldsymbol{\rho} \\ \boldsymbol{\rho}^{\dagger} &= \boldsymbol{\rho} \\ \mathrm{Tr}(\boldsymbol{\rho}) = \sum_i \rho_{ii} &= 1 \end{aligned}$$

The expectation value of an operator $\hat{A}$ in this state is

$$\langle A \rangle = \mathrm{Tr}(\boldsymbol{\rho A}).$$

For a density matrix $\boldsymbol{\rho}$, the time evolution is given by

$$i\hbar\dot{\boldsymbol{\rho}} = [\boldsymbol{H}, \boldsymbol{\rho}]$$

where $\boldsymbol{H}$ is the Hamiltonian matrix, and $[a, b] = ab - ba$ is the commutator. One can show that these properties follow naturally from Equations 1.1 and 1.2.

This also implies that the steady-states are given by

$$0 = [\boldsymbol{H}, \boldsymbol{\rho}_{\mathrm{SS}}]$$

So far, we haven't seen many advantages intrinsic to this formulation. However, the density matrix can hold quite a bit more information than a quantum state, and we can use that to describe significantly more complicated systems.

1.2.2 Mixed States

So far, we've assumed that we always know the exact state of the system, and can write a $|\psi\rangle$ fully describing it. In practice, this is often far from true—we often do not know with certainty what the quantum state of a particle is. For example, suppose we have two machines that prepare electrons in a quantum state, with one producing spin-up ($|\uparrow\rangle$) electrons and the other spin-down ($|\downarrow\rangle$) electrons. We have an electron produced by one of the machines (but we don't remember which one), and want to write down its state. The state is *either* $|\uparrow\rangle$ *or* $|\downarrow\rangle$, so we can't just write

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle + |\downarrow\rangle) \tag{1.3}$$

which suggests that the state is a linear combination of $|\uparrow\rangle$ *and* $|\downarrow\rangle$. This is a subtle and tricky distinction. When we write a state as a linear combination of other states (as in Eq. 1.3), we are saying our system is in all of those states simultaneously. Here, we know the state is entirely spin-up or spin-down—we just aren't certain about which

one[2][7]. The density matrix (with its additional information capacity) allows us to encode this classical uncertainty:

$$\boldsymbol{\rho} = \frac{1}{2}\left|\uparrow\right\rangle\left\langle\uparrow\right| + \frac{1}{2}\left|\downarrow\right\rangle\left\langle\downarrow\right|$$

In general, if we know that the (classical) probability of the system being in state $|\psi_k\rangle$ is p_k. we can write the density matrix as

$$\boldsymbol{\rho} = \sum_k p_k \vec{\psi}_k \otimes \vec{\psi}_k^\dagger$$

subject to the constraints

$$0 \leq p_k \leq 1 \text{ and } \sum_k p_k = 1.$$

This density matrix has the same properties as the one we introduced above, with the exception that

$$\boldsymbol{\rho}^2 \neq \boldsymbol{\rho}$$

for mixed states. Most importantly, time evolution is still governed by

$$i\hbar\dot{\boldsymbol{\rho}} = [\boldsymbol{H}, \boldsymbol{\rho}].$$

1.3 Heisenberg Chains

A prototypical model in many-body quantum mechanics is the Heisenberg model, consisting of spin-$\frac{1}{2}$ particles arranged in a lattice subject to an external magnetic field and interacting with each other magnetically. The Heisenberg model only accounts for nearest-neighbor interactions—the Pauli exclusion principle means that adjacent spins have a significantly lower potential energy when pointing in opposite directions, but this effect falls off *very* quickly with distance. Some models incorporate second-nearest neighbor effects, but they tend to increase the computational complexity of a system significantly. Figure 1.1 shows the structure of a simple one-dimensional model[9].

Normally, we'd denote each of these spins as $|\uparrow\rangle$ or $|\downarrow\rangle$ (or some linear combination thereof). However, as a nod to computer science (since these spins could represent qubits in a quantum computer), we'll denote spin-up as $|0\rangle$ and spin-down as $|1\rangle$[10]. The state of the system as a whole is then just the Kronecker product of the individual states:

$$|10011101\rangle \equiv |1\rangle \otimes |0\rangle \otimes |0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |1\rangle \otimes |0\rangle \otimes |1\rangle\,. \tag{1.4}$$

There are two observations to be made here. First, it becomes apparent how information can be encoded into the chain by converting it to binary and setting individual bits as required—useful for quantum computing purposes. Second, the dimension of the resulting Hilbert space is 2^N, so the complexity of the system increases exponentially with additional spins. Since any useful quantum computer will need at least a few dozen bits, figuring out how to work with these larger systems is an absolute necessity.

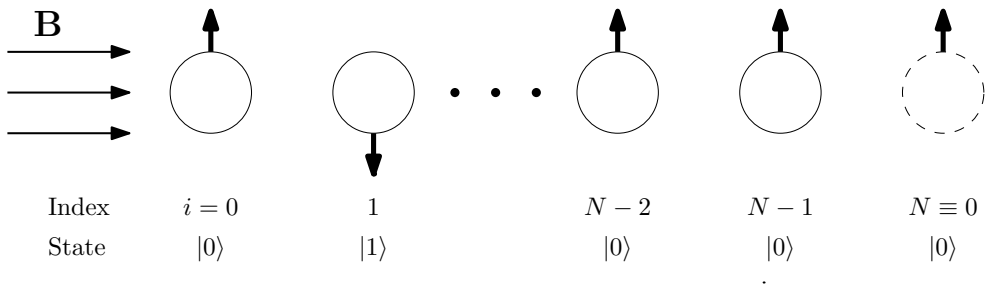


Figure 1.1: The structure of a simple, 1D Heisenberg chain. Note the periodic boundary condition—spin $N + 1$ is the same as spin 1.

1.3.1 Site Operators

Suppose, given the state described in Eq. 1.4, that we want to flip the 5-bit spin[2]. We could write this as

$$\begin{aligned}
&|1\rangle \otimes |0\rangle \otimes \hat{X}\,|0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |1\rangle \otimes |0\rangle \otimes |1\rangle \\
=&\hat{I}\,|1\rangle \otimes \hat{I}\,|0\rangle \otimes \hat{X}\,|0\rangle \otimes \hat{I}\,|1\rangle \otimes \hat{I}\,|1\rangle \otimes \hat{I}\,|1\rangle \otimes \hat{I}\,|0\rangle \otimes \hat{I}\,|1\rangle \\
=&(\hat{I} \otimes \hat{I} \otimes \vec{X} \otimes \hat{I} \otimes \hat{I} \otimes \hat{I} \otimes \hat{I} \otimes \hat{I})\,|10011101\rangle \\
\equiv&\vec{X}_5\,|10011101\rangle = |10111101\rangle\,.
\end{aligned}$$

$\vec{X}_5$ is a *site operator*, applying operator $\vec{X}$ to site 5 and leaving all others alone. A few things to note about these site operators:

- Operators acting on different sites always commute with each other:

$$\hat{X}_i\hat{Y}_j \equiv \hat{Y}_j\hat{X}_i.$$

- Operators acting on the same site do not commute with each other

$$\hat{X}_i\hat{Y}_i \neq \hat{Y}_i\hat{X}_i$$

  just as normal operators don't commute.

- To create an operator acting on multiple sites, simply multiply the constituent site operators together. For example, to create an operator $\hat{X}_{2,5}$ flipping the 2nd and 5th bit, we would do

$$\hat{X}_{2,5} = \hat{X}_2\hat{X}_5.$$

1.3.2 The Heisenberg Hamiltonian

The energy of a dipole with moment $\vec{\mu}$ in a magnetic field is given by

$$V = -\vec{\mu} \cdot \vec{B}.$$

[2]I've been working with little-endian states, so the least significant (1st) bit is to the right. Bits are also zero-indexed, for consistency with Python code.

For a quantum dipole, the moment $\vec{\mu}$ is related to the spin by

$$\vec{\mu} = \gamma \vec{\mathbf{S}}$$

where

$$\vec{\mathbf{S}} = \vec{X}\hat{x} + \vec{Y}\hat{y} + \vec{Z}\hat{z}.$$

Given some external magnetic field $\vec{B}$, the Hamiltonian for a dipole in that field is then

$$\hat{H} = B_x\hat{X} + B_y\hat{Y} + B_z\hat{Z}.$$

For our spin chain, the external magnetic field's contribution to the Hamiltonian is

$$H_{\text{ext}} = \sum_{i=0}^{N-1} B_x\hat{X} + B_y\hat{Y} + B_z\hat{Z}.$$

Without loss of generality, we can pick coordinates so that the magnetic field is pointing in the $\hat{x}$ direction, simplifying this to

$$H_{\text{ext}} = \sum_{i=0}^{N-1} B\hat{X}.$$

We also need to include the dipole-dipole interactions between adjacent spins. In an isotropic material, the potential energy of two spins would be

$$\hat{H} = \hat{X}_0\hat{X}_1 + \hat{Y}_0\hat{Y}_1 + \hat{Z}_0\hat{Z}_1.$$

In practice, we want to model anisotropic systems, so each of these terms has some associated factor $J_{\{x,y,z\}}$. The Hamiltonian for the Heisenberg model is therefore

$$\hat{H} = \frac{V}{4}\sum_{i=0}^{N-1}\left[J_x\hat{X}_i\hat{X}_{i+1} + J_y\hat{Y}_i\hat{Y}_{i+1} + J_z\hat{Z}_i\hat{Z}_{i+1}\right] + \frac{g}{2}\sum_{i=0}^{N-1}\hat{X}_i$$

where $\hat{X}_N \equiv \hat{X}_0$ due to periodic boundary conditions.

A few special cases of this model will be used in this thesis. The XXZ model sets $J_x = J_y \equiv \Delta$, only including anisotropy in the $\hat{z}$ direction (aligned with the basis state spins). A further simplification is the one-dimensional Ising model, which limits the interaction to the $\hat{Z}$ terms:

$$\hat{H} = \frac{V}{4}\sum_{i=0}^{N-1}\hat{Z}_i\hat{Z}_{i+1} + \frac{g}{2}\sum_{i=0}^{N-1}\hat{X}_i. \tag{1.5}$$

This model leads to a simplified Hamiltonian, which is often easier to reason with and work with analytically. We'll be using this model extensively, since its behavior is well-understood even for large chains.

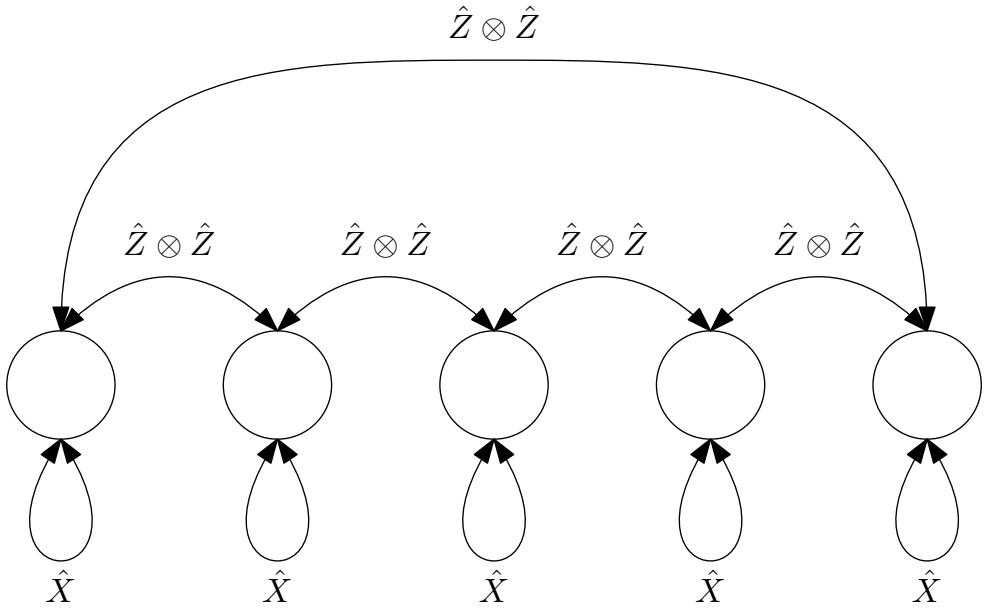


Figure 1.2: A graph representation of the Ising Hamiltonian

1.3.3 Graph Operators

As the spin chains we work with grow longer, the dimension of the associated Hilbert spaces grow exponentially. For example, a 16-spin chain—still too small to be a practical quantum computer—has a Hilbert space of dimension $2^{16} = 65\,536$, which gives a Hamiltonian of size $(2^{16})^2 = 4\,294\,967\,296$. If each element takes 8 bytes of storage—a single-precision complex number—storing the Hamiltonian alone takes 32 GB of memory. Since consumer-grade hardware (the kind we're hoping to use for our simulations) can't handle matrices this large, we need more efficient ways of handling our matrices. Sparse arrays could work, but computing the Lindblad equation for open quantum systems (discussed below) becomes nontrivial with sparse arrays.

Instead, the Hamiltonian (and any other operator) can be modeled as a graph, where the spins are represented by vertices in the graph and operators act on the edges connecting vertices. For example, Fig. 1.2 shows the graph representation of the Ising Hamiltonian. The graph representation of an operator makes addition and multiplication simpler than with the full matrix, and clearly shows the underlying structure of the system operator.

1.4 Open Quantum Systems

Consider a large, complicated quantum system living in Hilbert space $\mathcal{H}$. This larger system is closed—its time evolution is unitary, and it does not exchange information or energy with the outside world. We are only interested, however, in the evolution of a small portion of this larger system of interest—call this the system of interest, living in Hilbert space $\mathcal{H}_S$. The rest of the broader system is called the *bath*, living

in Hilbert space $\mathcal{H}_B$. It follows that

$$\mathcal{H} = \mathcal{H}_S \otimes \mathcal{H}_B.$$

Note that, although the vector space of the combined system is the outer product of the system's space and the bath's space, the state of the system cannot, in general, be expressed as the outer product of a system state and bath state:

$$|\psi\rangle \neq |\psi_S\rangle\,|\psi_B\rangle$$

This is only true in the case where there is no entanglement between the system and bath.

Our job, then, is to approximate the evolution of our system of interest, $\rho_S(t)$, without fully specifying the state of the bath ρ_B or overall system ρ, and without knowing the Hamiltonian for the overall system. To work out bow to do this, consider a simple system where we *can* fully specify the state of both the system and bath.

1.4.1 Simple Electron Relaxation

Consider a simple atom with two energy levels—the ground state at E_0, and an excited state at E_1. If the system is in the ground state, call its state $|0\rangle_S$; and if it is in the excited state, call the state $|1\rangle_S$. If the environment has a photon flying around (from the atom's decay), that's state $|1\rangle_B$—otherwise, it is in state $|0\rangle_B$. We don't know the initial state of the atom, but we can detect the photon flying around in the environment—we are continually measuring the environment (but not the atom itself). If the atom has a probability p of decaying in each timestep, there are two possible evolutions:

$$\begin{aligned}
&|0\rangle_S\,|0\rangle_E \to |0\rangle_S\,|0\rangle_E \\
&|1\rangle_S\,|0\rangle_E \to \sqrt{1-p}\,|1\rangle_S\,|0\rangle_E + \sqrt{p}\,|0\rangle_S\,|1\rangle_E\,.
\end{aligned}$$

The first case represents the situation where the atom was never excited from the start, and so nothing will *ever* happen, no matter how long we watch. The second case begins with the atom in an excited state—with probability p it decays to $|0\rangle_S$ and releases a photon into the environment, setting it to $|1\rangle_E$, and with probability $1-p$ nothing happens[1].

If we only look at the effect of these evolutions on the system (throwing away the component describing the environment), there are two possibilities for the system's evolution every timestep.

$$\boldsymbol{M}_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{pmatrix} \qquad \boldsymbol{M}_1 = \begin{pmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{pmatrix}. \tag{1.6}$$

Every dt, the system's density matrix evolves as

$$S(\rho_S(t)) = \rho_S(t + \mathrm{d}t) = \sum_k M_k \rho_S(t) M_k^\dagger.$$

Kraus Operators

These operators $M_0, M_1, \ldots$ are the *Kraus Operators*, and they describe the possible evolutions of the system of interest under unitary evolution of the overall system. In our specific case, M_0 represents a step with no evolution, while M_1 represents a step where the atom decays. Note that these operators are not Hermitian, and the evolution of the system of interest is non-unitary. A valid set of Kraus operators (which is not unique) requires that

$$\sum_k M_k^\dagger M_k = \mathbf{I}_S. \tag{1.7}$$

This is easy to verify for the operators in Eq. 1.6.

Note that all the requisite properties of the density matrix are preserved by this time evolution:

- $\rho_S(t + \mathrm{d}t)$ remains Hermitian:

$$\rho_S(t+\mathrm{d}t)^\dagger = \left(\sum_k M_k \rho_S(t) M_k^\dagger\right)^\dagger = \sum_k M_k \rho_S(t)^\dagger M_k = \rho_S(t+\mathrm{d}t)$$

- $\mathrm{Tr}[\rho_S(t + \mathrm{d}t)] = 1$. This follows from Eq. 1.7.

Note: Markovian Interactions

By describing the system-environment interactions using these Kraus operators, we've implicitly assumed that these interactions are *Markovian*—that the environment does not markedly change even as our system of interest undergoes its evolution. Mathematically, this requires that $\rho(t + \mathrm{d}t)$ depends only on $\rho(t)$ (and not any $\rho(t - t_r)$). Physically, this suggests that the environment is far larger than the system of interest, and that any changes to the environment (such as an increase in energy) quickly dissipate away.

Density Matrix Evolution

For our example, the density matrix evolves as

$$S(\rho) = M_0 \rho M_0^\dagger + M_1 \rho M_1^\dagger = \begin{pmatrix} \rho_{00} & \sqrt{1-p}\rho_{01} \\ \sqrt{1-p}\rho_{10} & (1-p)\rho_{11} \end{pmatrix} + \begin{pmatrix} p\rho_{11} & 0 \\ 0 & 0 \end{pmatrix}$$

If we apply this propagator n times, then, the ρ_{11} element decays as

$$\rho_{11} \to (1-p)^n \rho_{11}$$

and the off-diagonals decay as

$$\rho_{01} \to (1-p)^{n/2} \rho_{01}.$$

As $n \to \infty$,

$$\rho \to \begin{pmatrix} \rho_{00} + \rho_{11} & 0 \\ 0 & 0 \end{pmatrix}$$

as required to be trace-preserving.

If we let the probability that the atom decays in time $\mathrm{d}t$ be $p = \lambda \mathrm{d}t$, then the probability of the excited state remaining after a time t is

$$(1 - \lambda \mathrm{d}t)^{t/\mathrm{d}t} = e^{-\lambda t}$$

which is the exponential decay law we'd expect. Also note that, regardless of the initial state of the atom, it always ends up in a pure state[1] since $\rho_{00} + \rho_{11} = 1$, so $\rho^2 = 1$ for the final density matrix. This is not possible under unitary evolution, and hints at some of the unusual possibilities of this open evolution. This is a simple example, but demonstrates the mechanics we use to model the evolution of open systems.

1.4.2 Lindblad Dephasing and Bit-flip

Next, we will consider the evolution of a single spin-$\frac{1}{2}$ particle. The only thing driving the evolution of this system is random interactions with environmental photons—the Hamiltonian is zero. All environmental interactions on this system can be modeled as the random application of Pauli operators, with $\hat{X}$ flipping the spin, $\hat{Z}$ flipping the phase, and $\hat{Y}$ doing both[10]. In a timestep $\mathrm{d}t$, let the probability of applying a Pauli operator be p, with the specific operator being randomly chosen. The Kraus operators are then

$$\begin{aligned} \hat{M}_0 &= \sqrt{1-p}\hat{I} \\ \hat{M}_1 &= \sqrt{\frac{p}{3}}\hat{X} \\ \hat{M}_2 &= \sqrt{\frac{p}{3}}\hat{Y} \\ \hat{M}_3 &= \sqrt{\frac{p}{3}}\hat{Z}. \end{aligned}$$

We can show that these operators satisfy the condition that

$$\sum_k M_k M_k^\dagger = \mathbf{I}.$$

After a single timestep, an arbitrary density matrix will evolve as

$$\begin{aligned} S(\rho) = \rho(t + \mathrm{d}t) &= \sum_k M_k^\dagger \rho(t) M_k \\ &= \begin{pmatrix} \left(1 - p + \frac{p}{3}\right)\rho_{00} + 2\frac{p}{3}\rho_{11} & \left(1 - p - \frac{p}{3}\right)\rho_{01} \\ \left(1 - p - \frac{p}{3}\right)\rho_{10} & \left(1 - p + \frac{p}{3}\right)\rho_{11} + 2\frac{p}{3}\rho_{00} \end{pmatrix} \\ &= \begin{pmatrix} \rho_{00} & \rho_{01} \\ \rho_{10} & \rho_{11} \end{pmatrix} + \frac{p}{3}\begin{pmatrix} -2\rho_{00} + 2\rho_{11} & -4\rho_{01} \\ -4\rho_{10} & -2\rho_{11} + 2\rho_{00} \end{pmatrix}. \end{aligned} \tag{1.8}$$

Replace p with $\gamma\,\mathrm{d}t$ (where γ is the characteristic time of these errors). Then we have that

$$\rho(t+\mathrm{d}t) = \rho(t) + \frac{\gamma}{3}\begin{pmatrix} -2\rho_{00}+2\rho_{11} & -4\rho_{01} \\ -4\rho_{10} & -2\rho_{11}+2\rho_{00} \end{pmatrix}\mathrm{d}t.$$

If we take the limit as $\mathrm{d}t \to 0$, we get that

$$\begin{aligned}\frac{\mathrm{d}\rho}{\mathrm{d}t} &= \frac{\rho(t+\mathrm{d}t)-\rho(t)}{\mathrm{d}t} \\ &= \frac{\gamma}{3}\begin{pmatrix} -2\rho_{00}+2\rho_{11} & -4\rho_{01} \\ -4\rho_{10} & -2\rho_{11}+2\rho_{00} \end{pmatrix}.\end{aligned}$$

This—the continuous-time differential version of Eq. 1.8—is the Lindblad master equation, a differential equation describing the time-evolution of an open quantum system. It is typically written

$$\dot{\rho} = \mathcal{L}\hat{\rho} = i\big[\hat{H},\hat{\rho}\big] + \sum_k \left[L_k\rho L_k^\dagger - \frac{1}{2}\left(L_k^\dagger L_k\rho + \rho L_k^\dagger L_k\right)\right],$$

where the L_k are the Lindblad jump operators, a set of operators describing the effects of environmental interactions on the system. There are a few ways to interpret this equation—typically, we consider the environment as stochastically applying the jump operators to our system, but a compelling alternative explanation treats the environment as continuously measuring our system[11] (touching on the question of what a quantum measurement *is*).

For our toy system, the Hamiltonian is zero since we are considering dephasing in the absence of any external fields, so we need to find a set of operators L_k such that

$$\dot{\rho} = \sum_k \left[L_k\rho L_k^\dagger - \frac{1}{2}\left(L_k^\dagger L_k\rho + \rho L_k^\dagger L_k\right)\right] = \frac{\gamma}{3}\begin{pmatrix} -2\rho_{00}+2\rho_{11} & -4\rho_{01} \\ -4\rho_{10} & -2\rho_{11}+2\rho_{00} \end{pmatrix}. \quad (1.9)$$

Since the errors are generated by the random application of Pauli operators, it seems natural to use these operators as our Lindblad operators. And, indeed, we can show that the operators

$$\hat{L}_1 = \sqrt{\frac{\gamma}{3}}\hat{X} \qquad \hat{L}_2 = \sqrt{\frac{\gamma}{3}}\hat{Y} \qquad \hat{L}_3 = \sqrt{\frac{\gamma}{3}}\hat{Z}$$

satisfy equation 1.9. In general, the Lindblad jump operators will be the same as the Kraus operators representing environmental interactions, but with a different coefficient.

Lindblad vs. Kraus Operators

In the examples above, we've used the Kraus operators $\hat{M}_0, \hat{M}_1, \ldots$ to propagate the system forward by discrete timesteps, and the Lindblad equation (with associated Lindblad operators) to find a differential equation and evolution in continuous time. The two models are closely related, with evolution due to the $\hat{M}_0$ operator (which

represents the case of no environmental interaction) being represented by the $-i[\hat{H}, \hat{\rho}]$ term in the Lindblad equation, and evolution due to the other Kraus operators ($\hat{M}_1$, $\hat{M}_2$, ...) being represented by the

$$\sum_k \left[\hat{L}_k \rho \hat{L}_k^\dagger - \frac{1}{2} \left(\hat{L}_k^\dagger \hat{L}_k \rho + \rho \hat{L}_k^\dagger \hat{L}_k \right) \right]$$

term with Lindblad operators $\hat{L}_1$, $\hat{L}_2$, ...

1.5 The Leaky Ising Model

Throughout this thesis, we will be using the simple Ising model described in §1.3.2 combined with random environmnental bit-flips on spin-up sites. The Hamiltonian is the one specified in Eq. 1.5, and the Lindblad operators are

$$\hat{L}_i = \gamma \hat{\sigma}_i^- = \gamma(\hat{X} - i\hat{Y})$$

This is a standard toy model for open many-body quantum systems, being studied in [17], [13], among others. For ease of calculation, we'll largely be looking at chains of length 4 in this thesis, in line with Yoshioka's work in [17].

Chapter 2

Numerical Models

Once we have the Lindblad equation for our system (consisting of the Hamiltonian $\hat{H}$ and a set of Lindblad operators L_k), we need to solve N^2 coupled ODEs (where N is the rank of our density matrix). In toy cases like the one described in §1.4.2, we may be able to analytically find a solution. In general, however, numerical methods (such as Runge-Kutta routines or Monte Carlo methods) are required.

2.1 Runge-Kutta Routines

2.1.1 The Euler Method

When we take the Taylor approximation of a function, we have

$$f(x + \mathrm{d}x) = f(x) + f'(x)\mathrm{d}x + \mathcal{O}(\mathrm{d}x^2).$$

The *Euler Method* uses this first-order approximation to estimate the solution to a differential equation. The value of a function at timestep t_{i+1} is given by

$$f(t_{i+1}) \approx f(t_i) + f'(t_i)\Delta t. \tag{2.1}$$

This method is nice and simple, but it is not very good. Every timestep, we drop terms of order Δt^2—this is the order of *local error*. These errors accumulate with every timestep, and the number of steps required to propagate the solution from t_0 to t_{final} is proportional to $1/\Delta t$. Ultimately, the total error is proportional to Δt—this is the *global error*. This is not great from an efficiency perspective—to reduce our error by a factor of ten, we need to set Δt ten times smaller, and do ten times more function evaluations.

We can work around this by including a second-order term in our Taylor approximation:

$$f(t_{i+1}) = f(t_i) + f'(t_i)\Delta t + \frac{1}{2}f''(t_i)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{2.2}$$

This relies on the second derivative of f, unfortunately, which we may not have or want to calculate (can you *imagine* differentiating the Lindblad equation?)

2.1.2 Fourth-Order Runge-Kutta (RK4)

With some clever finite differencing[6], we can rewrite eq. 2.2 as

$$f(t_{i+1}) = f(t_i) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}\Delta t + \mathcal{O}(\Delta t^3) \tag{2.3}$$

where

$$\begin{aligned} G(f,t) &= \left.\frac{\mathrm{d}f}{\mathrm{d}t}\right|_{t_i} \\ k_1 &= G(f(t_i), t_i) \\ k_2 &= G(f(t_i) + k_1/2, t_i + \Delta t/2) \\ k_3 &= G(f(t_i) + k_2/2, t_i + \Delta t/2) \\ k_4 &= G(f(t_i) + k_3, t_i + \Delta t) \end{aligned}$$

Although this method requires four times the function evaluations as the simple Euler version, its local error is now of order Δt^3, and its global error Δt^2. A ten-fold decrease in step size now leads to a 100-fold increase in accuracy—easily making up for the increase in function evaluations at each step.

2.1.3 Using RK4 with the Lindblad equation

Although the examples shown above described f as a function of a single variable returning a single value, the Runge-Kutta method is equally valid for matrices and vectors. If f is our density matrix, $G = \dot{f}$ is just the Lindblad equation described above. We can propagate some initial ρ_0 forward in time, (hopefully) settling on some form of steady state. To demonstrate, let's attempt to find the steady-state equilibrium of the electron exposed to dephasing and bit-flip errors described in 1.4.2. Recall that we determined

$$\dot{\rho} = \frac{\gamma}{3}\begin{pmatrix} -2\rho_{00} + 2\rho_{11} & -4\rho_{01} \\ -4\rho_{10} & -2\rho_{11} + 2\rho_{00} \end{pmatrix}. \tag{2.4}$$

We can use this to code up G—in Python, it would look like

```
def G(t, rho):
    return (gamma/3) * np.array([
        [  2 * (-rho[0,0]+rho[1,1]), -4 * rho[0,1]              ],
        [ -4 * rho[1,0],              2 * (-rho[1,1]+rho[0,0]) ]
    ])
```

If the system is (arbitrarily) constructed in pure state $\rho_0 = |1\rangle\langle 1|$ and propagated forward through time, the state evolves as shown in Fig. 2.1. A few things to note:

- Although the system began as a pure state, it quickly becomes a mixed state
- The steady state is an equal mixture of the two pure basis states. This is a *Pointer State*, or a state of maximum decoherence[18]. Any information contained in the initial state has been lost to the environment, and we have no way

of determining what the initial state was even if we know its final state perfectly (which implies non-unitary evolution).

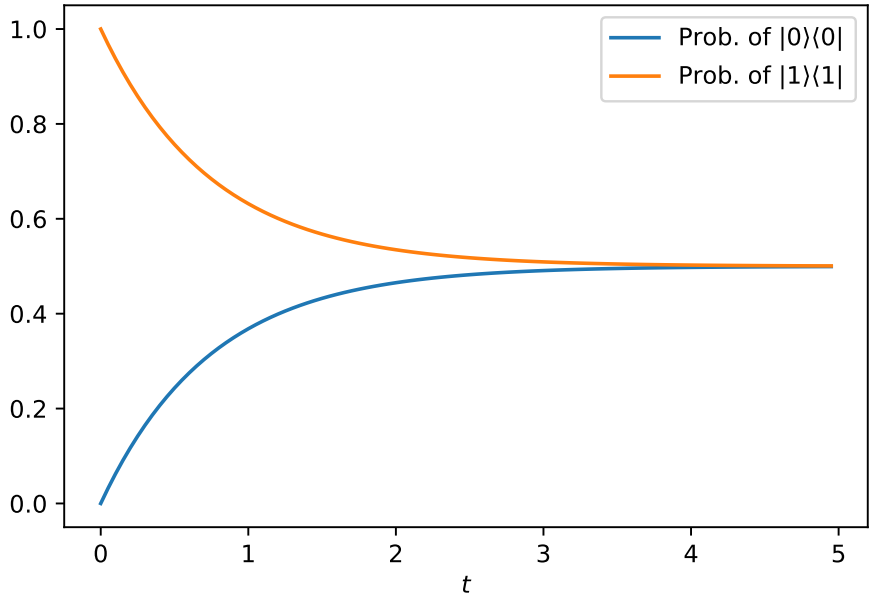


Figure 2.1: The evolution of the diagonal elements of the density matrix over time, using a Runge-Kutta routine.

The RK4 works well enough, and is trivial to code up. Its biggest drawback, however, is its poor scaling—if our matrix is $n \times n$, the Runge-Kutta method has complexity $\mathcal{O}(n^2)$. It can also require running forward in time far longer than we did in our toy example—if environmental interactions are rare, it may take a long time to evolve to an equilibrium state. Since this quickly becomes unworkable for anything but toy systems, we have to resort to more efficient methods—typically, some form of quantum Monte Carlo routine.

2.2 Finding Steady States

In the methods described above, given some differential equation for the evolution of our density matrix, we propagate it forward through time and watch it converge to some steady state ρ_{SS}. For large density matrices or systems with a slow rate of convergence, this proves extremely inefficient, and alternative methods are required.

Consider the Lindblad equation

$$\dot{\rho} = \mathcal{L}\rho,$$

where $\mathcal{L}$ is the Liouvillian superoperator discussed earlier. Once we've converged to a steady state,

$$\mathcal{L}\rho_{\text{SS}} = 0$$

so, if we can find a density matrix that meets this condition, we've found a steady state of the system.

2.2.1 Linearizing the Lindblad Equation

As a superoperator, it can be difficult to reason about the effect of $\mathcal{L}$ on ρ, so we begin by linearizing the Lindblad equation. This involves turning the density matrix ρ into a doubly indexed vector

$$|\rho\rangle\rangle = \frac{1}{C}\sum_{\sigma\,\tau}\rho_{\sigma\,\tau}\,|\sigma,\tau\rangle\rangle \tag{2.5}$$

where C is simply a normalization constant[1]. This is equivalent to simply flattening the matrix and renormalizing—in Python, we could write

```
vec_rho = np.flatten(rho)
vec_rho /= np.linalg.norm(vec_rho)
```

Converting back to a square matrix is then simply a matter of inverting this operation:

$$\hat{\rho} = C\sum_{\sigma\,\tau}\langle\,\langle\sigma,\tau|\rho\rangle\rangle\,|\sigma\rangle\,\langle\tau| \tag{2.6}$$

or, in code,

```
vec_size = len(rho_vec)
dm_size = int(sqrt(vec_size))
rho = np.reshape(rho_vec, (dm_size, dm_size))
```

Linearizing a superoperator is somewhat more complicated, and is described in detail in [17]. If we note that

$$\hat{A}\hat{\rho}\hat{B} = \sum_{\sigma\mu\nu\tau}A_{\sigma\mu}\rho_{\mu\nu}B_{\nu\tau}\,|\sigma\rangle\,\langle\tau|$$

and linearize this expression, we get that

$$\begin{aligned}|A\rho B\rangle\rangle &= \frac{1}{C}\sum_{\sigma\mu\nu\tau}A_{\sigma\mu}\rho_{\mu\nu}(B^T)_{\tau\nu}\,|\sigma,\tau\rangle\rangle\\ &= \hat{A}\otimes\hat{B}^T\,|\rho\rangle\rangle.\end{aligned}$$

Applying this to the Lindblad equation, we get that its linearized form is

$$\hat{\mathcal{L}}\,|\rho\rangle\rangle = \left[-i(\hat{H}\otimes\hat{1}-\hat{1}\otimes\hat{H}^T)+\sum_i\gamma_i\left(\hat{\Gamma}_i\otimes\hat{\Gamma}_i^*-\frac{1}{2}\hat{\Gamma}_i^\dagger\hat{\Gamma}_i\otimes\hat{1}-\frac{1}{2}\hat{1}\otimes\hat{\Gamma}_i^T\hat{\Gamma}_i^*\right)\right]\,|\rho\rangle\rangle. \tag{2.7}$$

This linearized Lindblad operator is, in general, a non-Hermitian matrix with complex eigenvalues. This is not ideal for our purposes, since this makes it difficult to

[1]The density matrix is normalized s.t. $\mathrm{Tr}\,\rho = 1$, while the vector is normalized s.t. $\langle\,\langle\rho|\rho\rangle\rangle = 1$. These are not equivalent, hence the normalization factor.

determine the "goodness" of some ρ—to determine how close it is to a steady state. Many of the methods we use to find steady states require a cost function to minimize, and $\hat{\mathcal{L}}$ in its current state does not fulfill this role[17].

The product of any operator $\hat{A}$ with its conjugate transpose is Hermitian and positive semi-definite—it has an entirely real and non-negative eigenspectrum[16]. Finding a steady-state, then, is equivalent to finding a ρ to minimize

$$\langle \langle \rho | \hat{\mathcal{L}}^\dagger \hat{\mathcal{L}} | \rho \rangle \rangle.$$

One of the most common problems in computational quantum mechanics is finding the ground state of a novel system—that is, given a Hamiltonian, find $|\psi\rangle$ to minimize

$$\hat{H} |\psi\rangle .$$

Since Hamiltonians are also Hermitian, the extensive library of techniques developed to find ground states can also be used in our search for the steady states of open quantum systems. In fact, our problem is slightly easier—we know *a priori* that $\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}} |\rho\rangle\rangle = 0$, while the ground state energy is rarely known. This simplifies many Monte-Carlo algorithms considerably[13].

2.2.2 Exact Methods

The problem of finding the steady state can be treated as an eigenvalue problem. We can rewrite $\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}} |\rho\rangle\rangle = 0$ as

$$\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}} |\rho\rangle\rangle = 0 |\rho\rangle\rangle.$$

Since we said that $\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}}$ has a purely nonnegative eigenspectrum, this corresponds to the smallest eigenvalue(s)[2]. If $\mathcal{L}$ is not of obscene size, this is not difficult to do—linear algebra libraries like ARPACK (used by NumPy) can be used to find only the smallest eigenvalues. Given floating-point imprecision, any sufficiently small eigenvalues (say those smaller than 1×10^{-4}) should be taken to correspond to steady states. The simplicity of this method and its reasonable performance for small systems (less than 4 spins or so) makes it attractive, especially as a reference for other techniques.

The poor scaling of exact diagonalization methods limits their usefulness, but other exact methods scale to larger systems [4][3]. We are trying to find $|\rho\rangle\rangle$ such that

$$\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}} |\rho\rangle\rangle = 0.$$

Treating this as an eigenvalue problem has some advantages—it is theoretically elegant, and allows us to find all the system's stable states (if there are multiple). However, treating this as a simple matrix equation and using standard numerical routines is often significantly faster, and often uses significantly less memory for reasons explained below. At its simplest, this is just a matter of doing

```
rho = np.linalg.solve(LdagL, 0)
```

[2]Degeneracy is a possibility—there may be more than one possible steady state.

[3]NetKet v2.1, in file `exact.py`, line 176

There are a number of optimizations that can be made here. First, we aren't relying on $\hat{\mathcal{L}}^\dagger\hat{\mathcal{L}}$ being Hermitian anymore, so we can instead solve $\mathcal{L}\,|\rho\rangle\rangle = 0$—saving us a matrix multiplication, which can be costly for large systems. $\mathcal{L}$ by itself can be constructed as a sparse matrix from individual site operators, which significantly reduces memory requirements[4] and opens up the possibility of more efficient solvers. NetKet uses the stabilized biconjugate gradient descent algorithm (BiCStab, implemented in Python as `scipy.sparse.linalg.bicgstab`). Using this iterative approach, I was able to handle systems up to 7 spins on a laptop.

[4] I crashed my computer trying to construct $\hat{\mathcal{L}}^\dagger\hat{\mathcal{L}}$ for a 10-spin system.

Chapter 3

Neural Network Methods for Quantum Simulation

3.1 Theory

3.1.1 What is a Quantum State?

As we discussed in Chapter 1, any quantum state is a linear combination of some basis states. If we denote these basis states as $|e_0\rangle$, $|e_1\rangle$, $|e_2\rangle \ldots |e_{n-1}\rangle$, the state could be written

$$|\psi\rangle \equiv \psi(0)\,|e_0\rangle + \psi(1)\,|e_1\rangle + \psi(2)\,|e_2\rangle + \cdots + \psi(n-1)\,|e_{n-1}\rangle\,.$$

This function ψ can be thought of as a "black box"—given an index i, it gives the weight of the i-th basis element in the quantum state. Rigorously, ψ is defined as

$$\psi : \{0, 1, \ldots, n-1\} \to \mathbb{C}$$

with the constraint that

$$\sum_{i=0}^{n-1} |\psi(i)|^2 = 1.$$

Finding some particular quantum state (say some $|\rho\rangle\rangle$ such that $\hat{\mathcal{L}}^\dagger\hat{\mathcal{L}}\,|\rho\rangle\rangle =$) is equivalent, then, to finding the function ψ. Fortunately for us, countless hours of research and unimaginable sums of money have been invested in approximating "black-box" functions in recent years.

3.1.2 Artificial Neural Networks

At its most fundamental, a neural network is a mechanism for approximating an arbitrary function given some cost function—some quantitative way to evaluate the quality of a guess. Our use of neural nets is somewhat different from most high-profile applications—I will briefly outline their more common uses before describing our application.

Figure 3.1: An owl. Image licensed from Dario Sanches[12], CC BY-SA.

Typically, the quality of a neural net is calculated using a large set of inputs and desired outputs. For example, we may want to find a function that takes in an animal picture and returns some categorization—passing in Fig. 3.1 to a well-trained network, for example, would return the categorization "owl," while a poorly-trained network might return "chicken."

The process of creating a neural net that approximates the target function well is known as "training" the network. We repeatedly tweak the network's internal parameters and test it using our quality measure, using some sort of multivariate optimizer to improve our approximation over time. We'll be using a Monte-Carlo process to train our network, but other methods are more common in other uses for neural networks.

3.1.3 Network Structure

So far, we haven't actually discusses the neural networks internal mechanics—simply treated it as a black box, tunable by some parameters. Many different network topologies exist—we'll be focusing on the *Restricted Boltzmann Machine*, a relatively simple network that happens to map very well onto the spin chains studied in this thesis.

The Restricted Boltzmann Machine (RBM) is a network that maps a collection of boolean inputs to a complex-valued output[1] The network consists of two boolean

[1]If all the network's parameters are real, the output is also real. This is frequently the case in

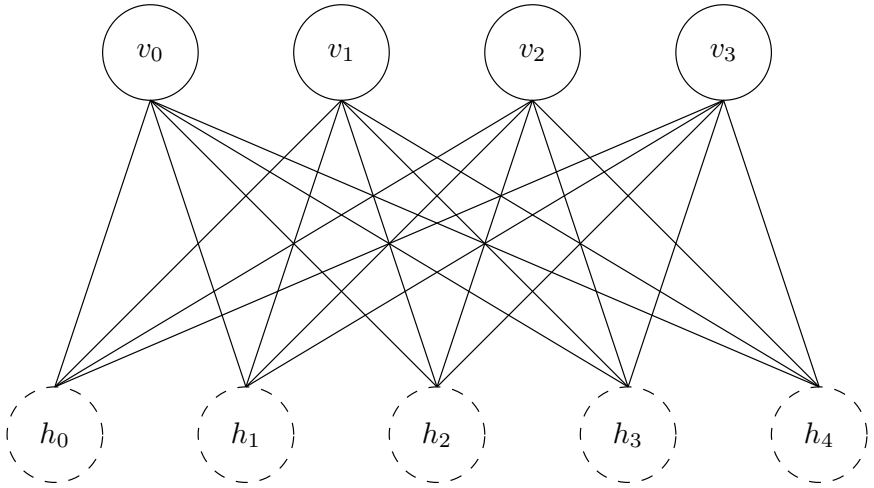


Figure 3.2: The topology of the Restricted Boltzmann Machine.

arrays, the visible layer and the hidden layer. Each visible node is connected to all of the hidden nodes (which also implies the reverse), and no intra-layer connections exist. Fig. 3.2 shows the layout of the RBM. Each of the nodes has some complex-valued weight associated with it ($a_0, a_1, \ldots, a_n$ for the visible nodes, and $b_0, b_1, \ldots, b_m$ for the hidden ones), and each of the connections has a weight (typically denoted W_{ij} where W is a $m \times n$ matrix).

Computing the output for a given input is a two-step process. The "energy" associated with a given configuration of visible and hidden spins is

$$E(v, h) = \sum_i a_i v_i + \sum_j b_j h_j + \sum_i \sum_j v_i W_{ij} h_j \tag{3.1}$$

where v is a boolean vector describing the visible state, and h a boolean vector describing the state of the hidden nodes. The hidden input is nonphysical, though, so we can't rely on it as an input. To eliminate the hidden spins and find the probability associated with some input v, we exponentiate and sum over all possible configurations of the hidden spins:

$$P(v) = \frac{1}{Z} \sum_h e^{E(v,h)}. \tag{3.2}$$

Because the notation $\sum_h$ is somewhat ambiguous—what we mean here is to sum over every possible input boolean combination. For example, if we have four hidden spins, there are 16 (2^4) possible inputs (0000, 0001, 0010, ..., 1111) that we need to sum over. In general, increasing the number of hidden spins increases the accuracy of a well-trained network, at the cost of training being significantly more difficult. The *number ratio* α refers to the ratio of hidden to visible spins, so $\alpha = m/n$[8].

To train the network, we can tweak the parameters a, b, and W. This suggests that, with n visible spins and m hidden ones, there are

$$m + n + mn$$

non-QM applications.

complex parameters to optimize over. Due to the size of the parameter space, Monte-Carlo methods are the only practical way to converge on a minimum. Other multivariate minimization algorithms (like those used by `scipy.optimize.minimize`) do poorly on such a large space, and either converge incredibly slowly or fail entirely.

3.1.4 Modeling Spin Chains with the RBM

At this point, the possibility of using a Restricted Boltzmann Machine to represent the quantum state of a spin chain should be readily apparent. Given a basis state as an input, the neural net returns the strength of that state in the system's overall state. One minor tweak must be made when we encode our input state—instead of coding spin-down sites as '0', we encode them as '-1'. For example, if we have a 4-spin system, configuration #5 would be

$$|5\rangle = |{\downarrow}{\uparrow}{\downarrow}{\uparrow}\rangle = |(-1)\,(1)\,(-1)\,(1)\rangle \implies v = \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \end{pmatrix}. \tag{3.3}$$

To be slightly handwavey, this is because the boolean array of spin states has physical significance (instead of serving only as an indexing mechanism, as in most applications of the RBM). Then, given the network parameters a, b, and W, we can reconstruct the quantum state represented by the network.

However, as discussed earlier, open systems cannot be fully encoded by a normal quantum state, instead requiring a density matrix. In §2.2.1, we showed that a density matrix for a system of n spins can be flattened into a quantum state of $2n$ spins. Each element of the density matrix is specified by two basis states, one for each of the indices.

Most work studying the use of RBMs for the simulation of open quantum systems maintains this distinction in the network, using a three-layered structure (see Fig. 3.3). The spins representing the first index (typically referred to as the "visible spins") are described using one set of weights, and the spins representing the second index ("fictitious spins", a terminology I find very confusing) are described by a second set of weights.

I found it significantly easier to work with a single input layer of size $2n$, which allowed me to use equations 3.1 and 3.2 with no modifications, and flatten the final state using Eq. 2.6. To be entirely clear—this is purely a bookkeeping matter, and the actual calculations are identical. Fig. 3.4 shows the setup of the neural network used in my Python code.

3.2 Variational Quantum Monte-Carlo

To find the set of parameters for our restricted Boltzmann machine that yield a stationary state, we need to search a sizable parameter space. This works well for

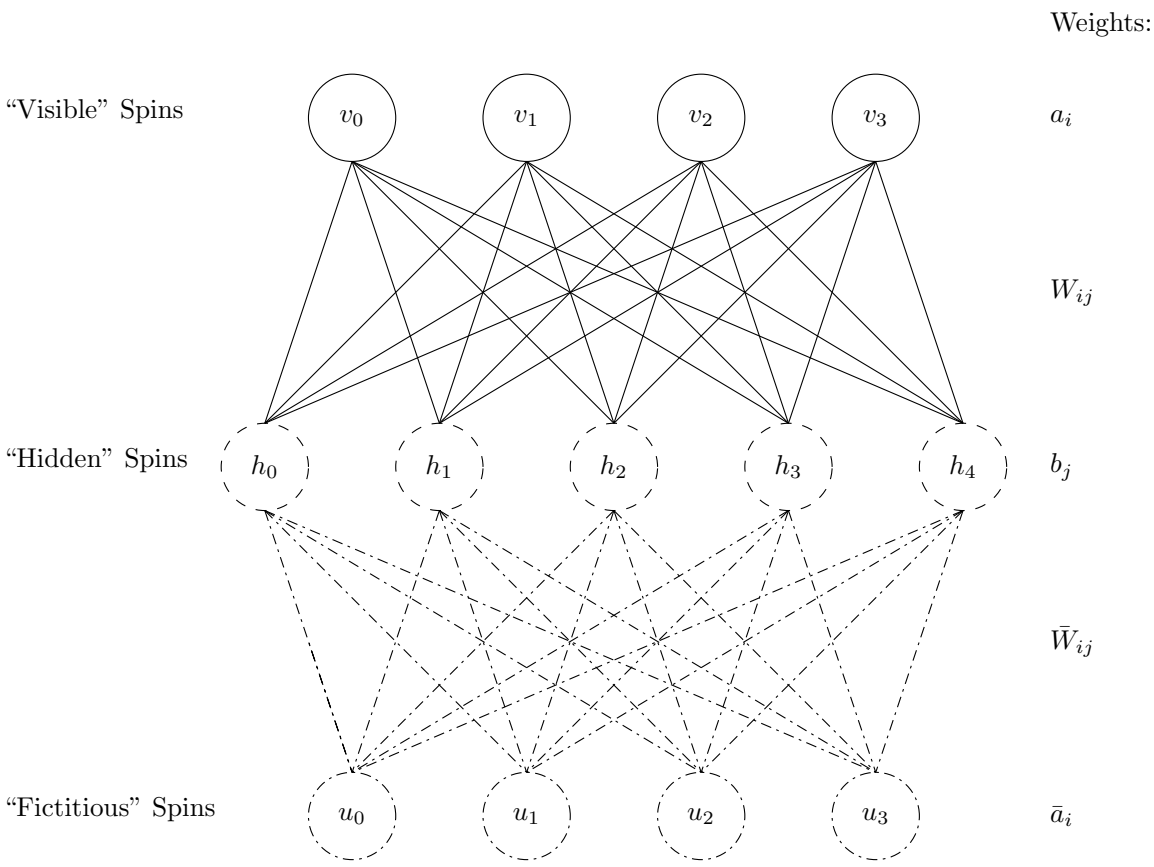


Figure 3.3: The three-layered network typically used in the simulation of open quantum systems.

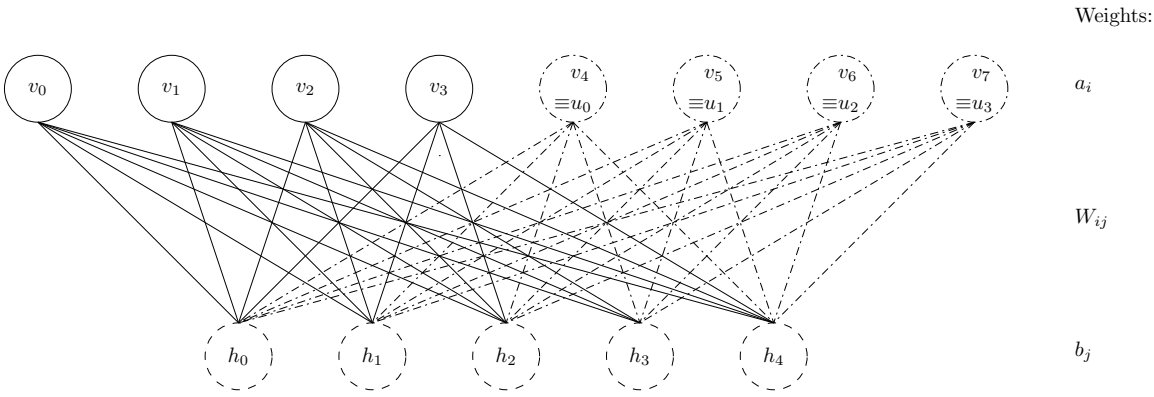


Figure 3.4: The two-layered network used in this thesis. Note that this figure is identical to Fig. 3.3—every spin and every connection is still here, just with different labeling.

smaller systems, but the sheer size of the Hilbert space for longer spin chains or other complex systems means that *many* iterations will be required to converge on a result.

Right now, each iteration is computationally expensive. Calculating the density matrix associated with a neural network scales as $\mathcal{O}(n^4)$—significantly better than the $\mathcal{P}(e^{2n})$ associated with exact methods, but still terrible. Variational Monte-Carlo methods allow us to approximate the stationarity of our state without fully evaluating

$$\frac{\langle\langle \rho_{\text{RBM}}(a)|\, \hat{\mathcal{L}}^\dagger \hat{\mathcal{L}}\, |\rho_{\text{RBM}}(a)\rangle\rangle}{\langle\langle \rho_{\text{RBM}}(a)|\rho_{\text{RBM}}(a)\rangle\rangle}, \tag{3.4}$$

where a represents the parameters of our trial wavefunction.

By only evaluating the stationarity of certain elements of the density matrix, and weighting those terms by their strength in the overall density matrix, we approximate the stationarity of the system as a whole without fully computing the density matrix according to Eq. 3.1. Rewrite Eq. 3.4 as

$$\begin{aligned}\frac{\langle\langle \rho_{\text{RBM}}(a)|\, \hat{\mathcal{L}}^\dagger \hat{\mathcal{L}}\, |\rho_{\text{RBM}}(a)\rangle\rangle}{\langle\langle \rho_{\text{RBM}}(a)|\rho_{\text{RBM}}(a)\rangle\rangle} &= \sum_{\sigma,\tau} P(\sigma,\tau,a) E_{\text{loc}}(\sigma,\tau,a) \\ &= \sum_{\sigma,\tau} \underbrace{\frac{|\rho(\sigma,\tau,a)|^2}{\langle\langle \rho(a)|\rho(a)\rangle\rangle}}_{=P(\sigma,\tau,a)} \underbrace{\frac{\langle\langle \sigma,\tau|\, \hat{\mathcal{L}}^\dagger \hat{\mathcal{L}}\, |\rho(a)\rangle\rangle}{\langle\langle \sigma,\tau|\rho(a)\rangle\rangle}}_{=E_{\text{loc}}(\sigma,\tau,a)}\end{aligned}$$

The stationarity of the system is then the average of the E_{loc}, weighted by the associated P. Instead of iterating over all possible σ, τ combinations, we pick a random subset of them. As with all Monte-Carlo algorithms, the more iterations we take, the more accurate our approximation will be. Since most work on variational Monte-Carlo methods focuses on wavefunctions in continuous position space, I found [14] and [5] very valuable resources in formulating VQMC in a discrete Hilbert space.

3.3 Implementation Details

The neural network discussed above was implemented in Python. The full source code is available in Appendix A, and mostly implements the theory discussed above straightforwardly, but I wanted to briefly discuss a few key details and explain aspects of the code that may not be self-apparent.

3.3.1 Converting Input States to Boolean Arrays

As equation 3.2 suggests, we need to iterate over all possible basis states. These states are indexed between 0 and $2^n - 1$, but the neural network requires that we pass them in as an array of 1s and -1s, as described in Eq. 3.3. To transform from an integer to an array of spins, we can use the function

```
def int_to_spinarr(val, length):
    if int(val) >= 2**length:
```

```
        raise ValueError('State id must be 0<=id<2**len')

    out = np.zeros(length, dtype=np.int64)

    for i in range(length):
        out[i] = (val & (1<<i)) != 0

    return out * 2 - 1
```

This function is not written as clearly as it could be. However, by using only fundamental arithmetic operations and not modifying any variables in the loop body (other than the output array), this function is trivial to compile to native code and parallelize—essential for a function called in a tight inner loop. Working through the function piece-by-piece:

$$\texttt{1<<i} \equiv 2^i.$$

We then take the bitwise-and (`&`) of 2^i and our index. If the i-th bit of the index is 1, this yields 2^i, and yields 0 otherwise. Checking if the result of the bitwise-and is not equal to 0, we get the actual value of the i-th bit. Finally, we convert the array from an array of 0 and 1 to an array of -1 and 1.

3.3.2 Improving Python Code Performance

Python is a convenient language for performing numerical work, but its performance is significantly worse than compiled, typed languages like C. Calculating the state associated with a given RBM is a computationally intensive task, with a 4-layer nested `for` loop, and must be performed many times during our variational Monte-Carlo procedure. Leaving this code in Python (even with Numpy's aggressive optimization) resulted in very slow code.

The `numba` library allows for compiling a limited subset of Python and Numpy code to native functions—essentially, upon starting the program, the library transpiles designated functions to C and compiles them, replacing all calls to those functions with the C equivalent. As demonstrated in Appendix A, line 31, prepending a function definition with

```
@jit(<function signature>, options...)
```

tells the Numba compiler to replace that function with a native equivalent. If we look at the declaration of function `vis_hidden_state_jit` (line 49)

```
@jit(
    complex64(
        int64, int64, int64, int64,
        complex64[:], complex64[:], complex64[:,:]
    ),
    nopython=True, cache=True, nogil=True, parallel=True
)
def vis_hidden_state_jit(v,h,vis_cnt,hid_cnt,a,b,W):
```

we see that this function takes in four scalar integers (the `int64`s, corresponding to `v`, `h`, `vis_cnt`, and `hid_cnt`), two one-dimensional complex arrays (`complex64[:]`, corresponding to `a` and `b`), and a two-dimensional complex array (`complex64[:,:]`, corresponding to `W`); and returns a complex scalar (`complex64`). Of the options specified:

- `nopython` indicated that the function should be fully native, with no Python fallback;
- `cache` tells the compiler to save compilation results across runs;
- `nogil` indicates that the function is pure and does not affect global state, so it is safe to parallelize or run in the background; and
- `parallel` tells the compiler to, if possible, parallelize loops.

The functions used to calculate the quantum state from an RBM were all optimized in this manner, which resulted in performance approximately an order of magnitude faster than a pure-Python implementation.

3.4 NetKet

Although compiling critical functions and taking other measures to ensure performant code led to significant improvements in the Python implementation of the restricted Boltzmann machine, no Python implementation will ever be as efficient as a well-tuned C or C++ implementation. The NetKet library [3] provides a carefully-optimized implementation of the above algorithm (among many others), and is used to perform the relevant calculations in many relevant papers including [17] and [5]. Despite being written largely in C++, the library exposes a Python interface for convenience, allowing us to reuse some code from earlier when setting up our models.

Note: Installing NetKet

NetKet depends on the OpenMPI library for parallelization (a very useful thing). However, upon installation, OpenMPI only sets up Python bindings for the system's version of Python, which tends to be quite old (especially on macOS)—we'd prefer to use a version of Python we installed ourselves. Doing this requires (as far as I can tell) that we overwrite the system Python, which is very much not recommended.

To use NetKet, I created a virtual machine running Linux and installed the requisite libraries on the VM instead—we can clobber the system Python without fear of breaking anything else. This also opens the possibility of setting the VM up through a cloud computing provider, which would offer significantly more computing power than a laptop.

The package maintainers are aware of this, and are working to remove dependence on OpenMPI and a number of other C++ libraries.

3.4.1 Setting up Open Systems in NetKet

NetKet models Hamiltonians and all other operators as a graph of site operators, as detailed in §1.3.3. Unfortunately, figuring out how to convert the linearized Lindblad equation (Eq. 2.7) to a graph is nontrivial, since we extensively transform the site operators used to build the Lindblad equation during the linearization process.

Fortunately, NetKet has support for open quantum systems in v2.1 and later, performing the computationally unpleasant work of linearizing the graph Lindbladian. This functionality is not yet documented anywhere except the package's source code and Git commits, so I will briefly outline the setup and optimization of the Ising model. The full source code for this method is available in Appendix B.

We begin by creating a graph for operators to act upon, and a Hilbert space that the operators live within:

```
# lindbladtest.py, line 30
g = Hypercube(length=L,n_dim=1)
hi = Spin(s=0.5, total_sz=0, graph=g)
```

The `Hypercube` function sets up a graph for our operators, and the `Spin` function sets up the Hilbert space acting on that graph. Two things to note: `s=0.5` tells the library we're working with spin-$\frac{1}{2}$ particles, and `total_sz=0` constrains us to only considering solutions with a total spin of 0. We know *a priori* that the stationary states are states of maximum decoherence which, for an even number of spins, means zero net spin.

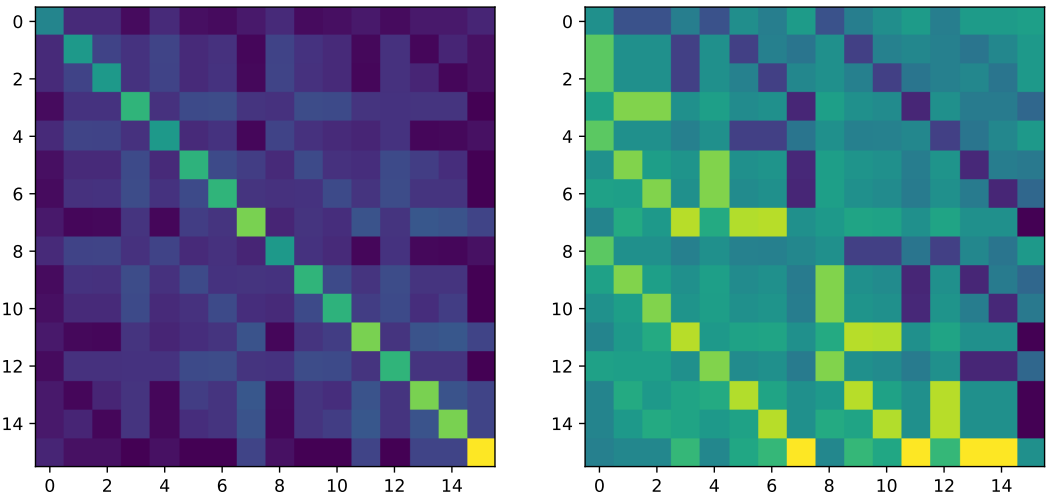
Next, we set up the RBM that represents our density matrix, and initialize it with random parameters.

```
ma = NdmSpinPhase(hilbert=hi, alpha=2, beta=1)
ma.init_random_parameters()
```

This is relatively straightforward—the only thing to note is the meaning of the `alpha` and `beta` parameters. `alpha` is the number ratio as defined in §3.1.3, and `beta` is the ratio of ancilliary spins to real spins (which should always be 1 for our purposes). We then initialize the machine with random parameters.

To create the Hamiltonian for our system, we pass in a list of site operators acting on the graph structure:

```
sx = np.array([
    [0,1],
    [1,0]
])
sz = np.array([
    [1,0],
    [0,-1]
])

ha = nk.operator.LocalOperator(hi)
for i in range(L):
    ha += nk.operator.LocalOperator(hi, (g_factor/2)*sx, [i])
    ha += nk.operator.LocalOperator(hi, (V/4)*np.kron(sz, sz), [i,(i
   +1)%L])
```



(a) The real component of ρ_{SS} (b) The imaginary component of ρ_{SS}

Figure 3.5: The steady-state density matrix for the 4-site Ising model.

For each site, we add the interaction with the external magnetic field, and the nearest-neighbor interaction. Note that we are still in the original Hilbert space here—the library handles the doubling of the Hilbert space (and associated operators).

To turn this into an open system, we need to add the Lindblad jump operators, which we do with

```
lind = nk.operator.LocalLiouvillian(ha)

sigmam = np.array([[0, 0], [1, 0]])
for i in range(L):
    j_op = nk.operator.LocalOperator(hi, gamma * sigmam, [i])
    lind.add_jump_op(j_op)
```

This functionality is not yet documented, and was discovered by reading the package's source code. The `nk.operator.LocalLiouvillian(ha)` function creates the Liouvillian superoperator associated with our Hamiltonian, and the subsequent `for` loop adds the jump operators. At this point, we have constructed the leaky Ising model described in §1.5.

To find the steady state through the Laczos exact diagonalization method, as described in §2.2.2, we can call `rho = nk.exact.steady_state(lind)`. For a four-spin system, this gives the density matrix shown in Fig. 3.5.

We then need to set up the sampler and optimizer to optimize the neural network's parameters to converge on the ground state. We do this with

```
sampler = nk.sampler.LocalKernel(ma)
optimizer = nk.optimizer.Sgd(learning_rate=0.05)
steadystate_vqmc = nk.SteadyState(
    lindblad=lind,
    sampler=sampler,
    optimizer=optimizer,
    n_samples=1000,
```

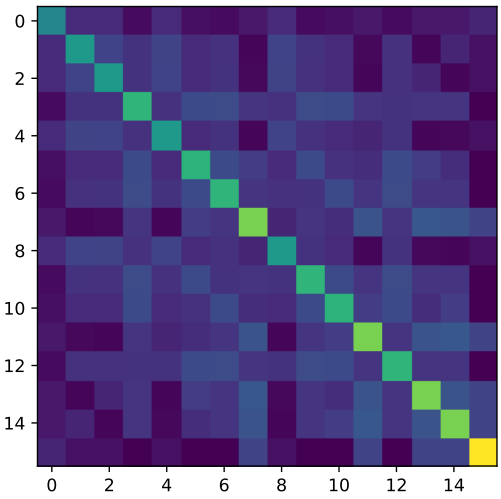
```
    sampler_obs=sampler,
    n_samples_obs=1000
)
print(steadystate_vqmc.info())

steadystate_vqmc.run(output_prefix='test', n_iter=1000)
```

We provide a sampling mechanism (here, a "LocalKernel" optimizer that randomly selects sites) and optimizer (here, a relatively simple stochastic gradient descent method), before setting up the VQMC mechanism and running it for 100 iterations (which was found to provide satisfactory convergence). Upon running, the VQMC code dumps per-iteration statistics and the resulting neural network into files `test.log` and `test.wf`, respectively. To recover the density matrix, we can create a new RBM object, load the parameters, and build the density matrix:

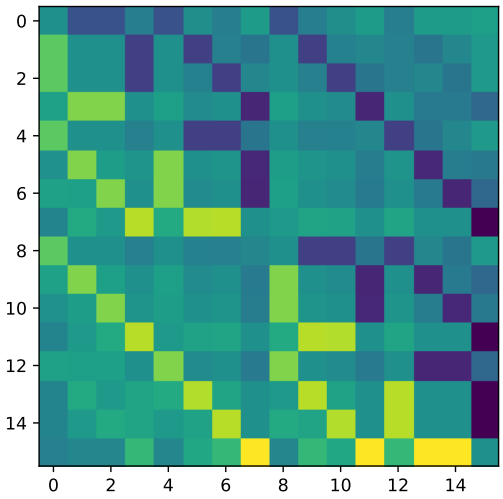
```
output_machine = NdmSpinPhase(hilbert=hi, alpha=2, beta=1)
output_machine.load('test.wf')
dm = output_machine.to_dense()
```

Training the neural network on the Lindbladian for the four-spin Ising model, we recover the density matrix shown in Fig. 3.6. Note that this result looks near-identical to Fig. 3.5—evidently, the neural network ansatz can represent wavefunctions very similar to the true steady state. Fig. 3.6c shows the decrease in $\hat{\mathcal{L}}^\dagger\hat{\mathcal{L}}$ as we train the neural network—note that it is asymptotically converging to some non-zero level. As the quality of our ansatz improves (which in practice means a higher number ratio), this convergence level should drop.

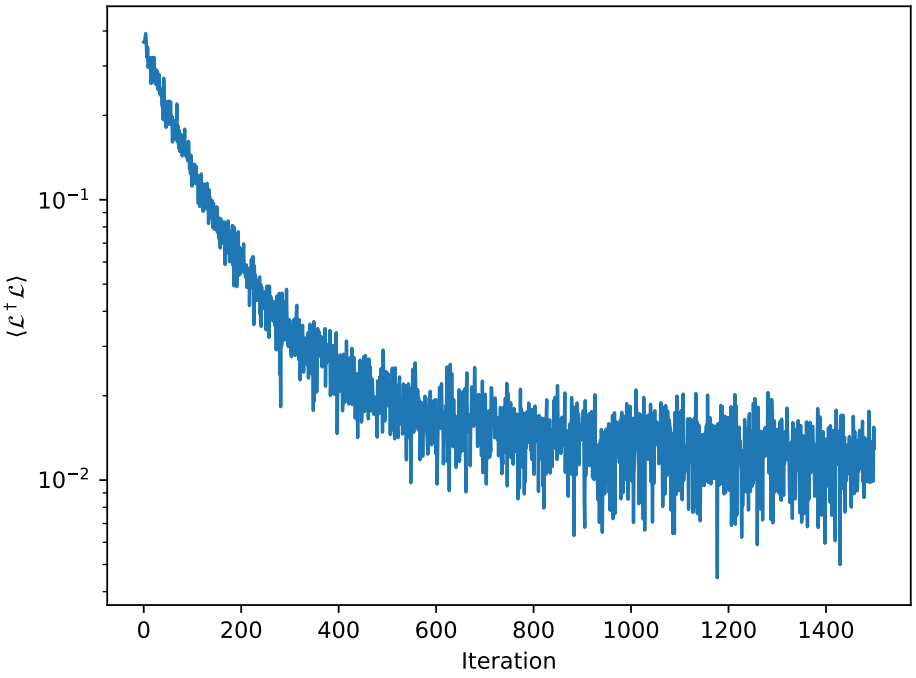
For this simple four-spin example, neural methods are significantly outperformed by exact diagonalization. Finding the eigenspectrum of a 16×16 matrix takes less than a second, while constructing and training the neural network to a suitable level took about ten minutes on a laptop. However, the relatively good (polynomial instead of exponential) scaling of the neural method means that, for systems larger than a dozen spins or so, neural methods are significantly faster than exact diagonalization.



(a) The real component of ρ_{SS}



(b) The imaginary component of ρ_{SS}



(c) The evolution of $\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}}$ as the network is trained. Note that the asymptotic value of $\hat{\mathcal{L}}^\dagger \hat{\mathcal{L}}$ is not zero—our ansatz is not perfect, and raising the number ratio would allow us to find a density matrix closer to the true stationary value.

Figure 3.6: The steady-state density matrix for the 4-site Ising model, calculated using neural network methods. Note that it is near-identical to Fig. 3.5. The density matrix was computed using a number ratio $\alpha = 2$ (for a total of 4 visible spins, 4 fictitious spins, and 16 hidden spins), 500 Monte-Carlo steps per iteration, and 1500 iterations to train the restricted Boltzmann machine.

Conclusion

As quantum mechanics moves from a purely theoretical field of study to one with practical implications for computing and communications, understanding the behavior of imperfect systems—human-engineered devices that can't be fully shielded from the rest of the universe—is critical for engineering reliable, fault-tolerant systems. Billions of dollars and millions of hours have been invested into research on qubit construction and quantum error correction, looking at ways to prevent the information encoded into quantum states from leaking out into the environment.

We understand the mechanisms that cause dephasing—stray cosmic rays, thermal noise, and even radioactive decay—but actually estimating their effects is a far more challenging task. The Lindblad master equation for open quantum evolution is, as demonstrated in this thesis, a challenging equation to work with, scaling exponentially with system complexity. With dozens of qubits in even the simplest proposed quantum computers, traditional numerical methods are not capable of modeling environmental interactions in these devices.

The neural network ansatz studied here is a relatively new technique in computational quantum mechanics, and one that has potential analogies in many other subfields of physics. Neural networks have been used to find solutions to partial differential equations, opening applications in fluid dynamics, general relativity, and field theory—some of which are already being researched. New network architectures and topologies—developed for tasks like computer vision and language processing—have the potential to offer faster training times or higher fidelity, making the neural network ansatz an even better approximation.

Even within the subfield of many-body QM, open problems abound. In this thesis, we considered the steady-states of open systems—useful for understanding the long-term evolution of quantum states and offering insight on thermodynamic questions, but not for understanding how to detect and correct errors in quantum computers. Newer research is attempting to model the time evolution of quantum systems using a neural network ansatz, which would offer further insight into the dephasing process.

The spin-$\frac{1}{2}$ chains discussed in this thesis are a convenient model—simple, well-studied, and topologically similar to the RBM architecture. However, proposed quantum computers tend to use other modalities—like trapped supercold ions or transmon charge devices—for practical reasons. These modalities are subject to environmental interactions different from the ones studied here, and constructing a neural network ansatz for those systems would make it easier to understand their dephasing process.

If someone were interested in continuing the work laid out in these pages, I would

suggest studying the performance of the neural network ansatz on longer chains and higher-dimensional systems—we know machine learning techniques outperform exact methods, but is their runtime fast enough to be actually useful? Open quantum systems raise interesting thermodynamic questions, and attempting to find states that maximized system entropy could be used as an alternative method for finding steady states. Finally—and this is likely too ambitious for a thesis—adapting the neural network ansatz to continuous-basis problems (like finding electron orbital wavefunctions) would be absolutely fascinating, and offer additional insight into some of the best-studied problems in computational quantum mechanics.

Appendices

Appendix A

Python RBM Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from operator import setitem
import math
from tqdm import tqdm
from numba import *
from scipy import linalg, optimize
from numpy.random import default_rng
import warnings
rng = default_rng()

MAX_SPINS = 65

fig_dir = '/Users/Thomas/Documents/Reed/current/thesis/thesis/
    chapters/resources/ch3/'

squares = [x**2 for x in range(MAX_SPINS)]

# returns n random complex numbers in the unit circle
def random_complex(n=1, length=1.0):
    angles = rng.random(n) * 2 * math.pi
    lengths = rng.random(n) * length

    return (np.cos(angles) * lengths + 1.0j * np.sin(angles) *
    lengths).astype(np.complex64, order='C')



@jit(
    int64[:](int64, int64),
    nopython=True, cache=True, parallel=True, nogil=True
)
def int_to_spinarr(val, length): #
    if int(val) >= 2**length:
        raise ValueError('State id must be 0<=id<2**len')

    out = np.zeros(length, dtype=np.int64)
```

```
    for i in prange(length):
        out[i] = (val & 2**i) != 0

    return out * 2 - 1

# Define a few functions for use in the RBMState class.
# We don't define them as class methods so that
# they can be parallelized and JITed
@jit(
    complex64(int64, int64, int64, int64, complex64[:], complex64
    [:], complex64[:,:]),
    nopython=True, cache=True, nogil=True, parallel=True
)
def vis_hidden_state_jit(v,h,vis_cnt,hid_cnt,a,b,W): #
    v_arr = int_to_spinarr(v, vis_cnt).astype(np.complex64)
    h_arr = int_to_spinarr(h, hid_cnt).astype(np.complex64)

    return np.dot(v_arr, a) + np.dot(h_arr, b) + np.dot(v_arr.T, np.
    dot(W, h_arr))



@jit(
    complex64(int64, int64, int64, complex64[:], complex64[:],
    complex64[:,:]),
    nopython=True, parallel=True, cache=True, nogil=True
)
def vis_state_jit(v, vis_cnt, hid_cnt, a, b, W):
    out = np.zeros(2**hid_cnt, dtype=np.complex64)
    for h in prange(2**hid_cnt):
        outval = vis_hidden_state_jit(
            v,h,vis_cnt,hid_cnt,a,b,W
        )

        out[h] = np.exp(outval)

    return np.sum(out)

class RBMState(object):
    def __init__(self, visible_count, num_ratio, random=True):
        if random:
            print("Initializing RBM with random complex numbers")
            genfunc = random_complex
        else:
            print("Initializing RBM with zeroed params")
            genfunc = lambda n: np.zeros(n, dtype=np.complex64)

        self.visible_count = visible_count
        self.a = genfunc(self.visible_count)

        self.hidden_count = visible_count * num_ratio
        self.b = genfunc(self.hidden_count)
```

```
        self.W = genfunc(self.visible_count*self.hidden_count).
    reshape((self.visible_count, self.hidden_count), order='C')


    def vis_state(self, v):
        return vis_state_jit(v, self.visible_count, self.
    hidden_count, self.a, self.b, self.W)

    def quantumstate(self):
        state = np.zeros(2**self.visible_count, dtype=np.complex64)
        for i in range(2**self.visible_count):
            state[i] = self.vis_state(i)

        return state

    def dump_params(self):
        return np.concatenate([self.a, self.b, self.W.flatten()])

    def load_params(self, dump):
        dump = dump.astype(np.complex64)
        self.a = dump[:self.visible_count]
        self.b = dump[self.visible_count:self.visible_count + self.
    hidden_count]
        self.W = dump[self.visible_count+self.hidden_count:].reshape
    ((self.visible_count, self.hidden_count), order='C')


class SpinChain(object):
    def __init__(self, length, initialstate):
        if length >= MAX_SPINS:
            raise ValueError("Cannot␣handle␣a␣spin␣chain␣longer␣than
    ␣{}".format(MAX_SPINS))
        self.spinarr = int_to_spinarr(initialstate, length)
        self.val = initialstate
        self.length = length

    def dm(self):
        if not self.length in squares:
            raise ValueError("This␣spin␣state's␣length␣is␣not␣a␣
    square.␣It␣cannot␣represent␣a␣density␣matrix")
        dmsize = int(math.sqrt(self.length))
        return np.reshape(self.spinarr, (dmsize, dmsize))

# Set up the problem

X = np.array([[0,1],[1,0]])
Y = np.array([[0,-1j],[1j,0]])
Z = np.array([[1,0],[0,-1]])
eye = np.eye(2)

def kronecker_multiproduct(arrs):
    out = arrs[0]
    for arr in arrs[1:]:
        out = np.kron(out,arr)
```

```
    return out

def multi_id(N):
    return kronecker_multiproduct([eye]*N)

def make_site_operator(op, N):
    return np.array([
        np.kron(op, multi_id(N-1)) if i == 0 else
        np.kron(multi_id(N-1), op) if i == N-1 else
        kronecker_multiproduct([multi_id(i), op, multi_id(N - i - 1)
   ])
        for i in range(N)
    ])

def D_sup(Gamma):
    dim = Gamma.shape[0]
    I = np.eye(dim)

    GammaT = Gamma.conj().transpose()


    return (np.kron(Gamma, Gamma.conjugate()) -
        (1/2) * np.kron(GammaT @ Gamma, I) -
        np.kron(I, (1/2) * Gamma.transpose() @ Gamma.conj())

    )

def construct_linearized_lindblad(H, jumps, jump_strengths):
    i = 1.0j
    dim = H.shape[0]
    I = np.eye(dim)

    lindblad = -i * (
        np.kron(H, I) -
        np.kron(I, H.transpose())
    ) + sum([
        gamma * D_sup(Gamma) for
        gamma, Gamma in zip(jump_strengths, jumps)
    ])

    return lindblad

def hermetian_lindblad(H, jumps, jump_strengths):
    L = construct_linearized_lindblad(H, jumps, jump_strengths)

    return L.conj().transpose() @ L

def vec_to_dm(vec):
    len = vec.shape[0]
    dmlen = int(math.sqrt(len))
    dm = np.reshape(vec, (dmlen, dmlen))

    return dm / np.trace(dm)
```

```
# And set up our actual model

# 1D Transverse Field Ising Model
V = 0.3
g = 1

L = 4
gamma = 0.5

Xs = make_site_operator(X, L)
Ys = make_site_operator(Y, L)
Zs = make_site_operator(Z, L)

H = (V/4) * sum([
    Zs[i] @ Zs[(i+1)%L]
    for i in range(L)
]) + (g/2) * sum([
    Xs[i]
    for i in range(L)
])



Gammas = [(X - 1.0j * Y)/2 for X,Y in zip(Xs,Ys)]
gammas = [gamma for _ in range(L)]

LL = hermetian_lindblad(H, Gammas, gammas)

# First---the exact diagonalization

vals, vecs = linalg.eigh(LL, eigvals=(0,1))

v0 = vecs[:, 0]
lin_dm = vec_to_dm(v0)
plt.imshow(lin_dm.real)
plt.savefig(fig_dir + 'dm_real.pdf', bbox_inches='tight')
plt.show()
plt.imshow(lin_dm.imag)
plt.savefig(fig_dir + 'dm_imag.pdf', bbox_inches='tight')
plt.show()
exit(0)


# Then---the NN method, with simple multivariate optimizer

num_ratio = 2

doubled_visible_spins = 2 * L
hidden_spins = num_ratio * doubled_visible_spins

def unpack_args(vis_cnt, hdn_cnt, args):
    return args[:vis_cnt], args[vis_cnt:vis_cnt+hdn_cnt], np.reshape
   (args[vis_cnt+hdn_cnt:](vis_cnt, hdn_cnt))
```

```
def real_vec_to_complex_vec(vec):
    l = vec.shape[0]
    if l % 2 != 0:
        raise ValueError('Cannot␣interpret␣vec.␣of␣len␣{}␣as␣a␣
   complex␣vec'.format(l))

    reals = vec[:l//2]
    imags = vec[l//2:]
    return reals + 1.0j * imags

def complex_vec_to_real_vec(vec):
    reals = vec.real
    imags = vec.imag
    return np.concatenate([reals, imags])


def create_objective_function(vis_cnt, hdn_cnt):
    net = RBMState(vis_cnt, num_ratio, random=True)

    iteration = [0]
    def obj(args):
        net.load_params(real_vec_to_complex_vec(args))

        state = net.quantumstate()

        stationarity = np.linalg.norm(LL.dot(state))

        print("i={}\ t L L ␣|  >>␣=␣{:4f}".format(iteration[0],
   stationarity))
        iteration[0] += 1

        if stationarity < 1e-3:
            # We'd never get this low in reality---indicated FP
   error
            return 1000


        return stationarity

    initial_state = net.dump_params()
    return initial_state, obj


initial_state, obj = create_objective_function(doubled_visible_spins
   , hidden_spins)


out = optimize.minimize(obj, complex_vec_to_real_vec(initial_state),
    method='Powell')


final_params = out.x
finalRBM = RBMState(doubled_visible_spins, num_ratio, False)
finalRBM.load_params(final_params)
```

```
final_state = finalRBM.quantumstate()
nndm = vec_to_dm(final_state)
plt.imshow(nndm.real)
plt.show()
plt.imshow(nndm.imag)
plt.show()
```

Appendix B

NetKet RBM Implementation

```
#!/usr/bin/env python
# coding: utf-8
# In[18]:

import numpy as np
import matplotlib.pyplot as plt
import netket as nk
import math as m

# In[19]:

from netket.machine import NdmSpinPhase
from netket.hilbert import Spin
from netket.graph import Hypercube

# In[20]:

L = 4
V = 0.3
g_factor = 1.0
gamma = 0.5
vv = np.random.rand(L*2)
vl = vv[:L]
vr = vv[L:]
vv2 = np.random.rand(3,L*2)
vl2 = vv2[:,:L]
vr2 = vv2[:,L:]

# In[21]:

g = Hypercube(length=L,n_dim=1)
hi = Spin(s=0.5, total_sz=0, graph=g)
ma = NdmSpinPhase(hilbert=hi, alpha=2, beta=1)
ma.init_random_parameters()

# In[22]:

dd = ma.diagonal()
```

```
# In[23]:

dd.log_val(vl)

# In[24]:


sx = np.array([
    [0,1],
    [1,0]
])
sz = np.array([
    [1,0],
    [0,-1]
])

ha = nk.operator.LocalOperator(hi)
for i in range(L):
    ha += nk.operator.LocalOperator(hi, (g_factor/2)*sx, [i])
    ha += nk.operator.LocalOperator(hi, (V/4)*np.kron(sz, sz), [i,(i
   +1)%L])

# In[25]:

lind = nk.operator.LocalLiouvillian(ha)

# In[26]:
sigmam = np.array([[0, 0], [1, 0]])
for i in range(L):
    j_op = nk.operator.LocalOperator(hi, gamma * sigmam, [i])
    lind.add_jump_op(j_op)

# In[27]:

rho = nk.exact.steady_state(lind)

# In[28]:

plt.imshow(np.imag(rho))

# In[47]:

sampler = nk.sampler.LocalKernel(ma)
optimizer = nk.optimizer.Sgd(learning_rate=0.05)
steadystate_vqmc = nk.SteadyState(
    lindblad=lind,
    sampler=sampler,
    optimizer=optimizer,
    n_samples=10000,
    sampler_obs=sampler,
    n_samples_obs=10000
)
print(steadystate_vqmc.info())
```

```
# In[ ]:

steadystate_vqmc.run(output_prefix='test', n_iter=100)

# In[31]:

arr = ma.to_array()

# In[32]:

import json

# In[33]:

with open('test.wf', 'r') as inf:
    wf = json.load(inf)
for k,v in wf.items():
    print(k, ": ", v)

# In[34]:

ma = ma.load('test.wf')
vec = ma.to_array()
dmsize = int(m.sqrt(len(vec)))
dm = vec.reshape((dmsize,dmsize))
plt.imshow(dm.real)
plt.show()
plt.imshow(dm.imag)
plt.show()
```

References

[1] Quantum theory of radiation interactions. Library Catalog: ocw.mit.edu. 10, 12

[2] Mark (Mark K.) Beck. *Quantum mechanics : theory and experiment*. Oxford University Press. 3, 6

[3] Giuseppe Carleo, Kenny Choo, Damian Hofmann, James E. T. Smith, Tom Westerhout, Fabien Alet, Emily J. Davis, Stavros Efthymiou, Ivan Glasser, Sheng-Hsuan Lin, Marta Mauri, Guglielmo Mazzola, Christian B. Mendl, Evert van Nieuwenburg, Ossian O'Reilly, Hugo Théveniaut, Giacomo Torlai, Filippo Vicentini, and Alexander Wietek. NetKet: A machine learning toolkit for many-body quantum systems. 10:100311. 28

[4] Giuseppe Carleo, Hoffman, and Filippo Vicentini. NetKet. 19

[5] Giuseppe Carleo, Yusuke Nomura, and Masatoshi Imada. Constructing exact representations of quantum many-body systems with deep neural networks. 9(1):1–11. 26, 28

[6] Joel Franklin. *Computational methods for physics*. Cambridge University Press. 16

[7] David J. (David Jeffery) Griffiths. *Introduction to quantum mechanics*. Cambridge University Press, third edition. edition. 6

[8] Michael J. Hartmann and Giuseppe Carleo. Neural-network approach to dissipative quantum many-body dynamics. 122(25):250502. 23

[9] Kira Joel, Davida Kollmar, and Lea F. Santos. An introduction to the spectrum, symmetries, and dynamics of spin-1/2 heisenberg chains. 81(6):450–457. Publisher: American Association of Physics Teachers. 6

[10] N. David Mermin. *Quantum computer science : an introduction*. Cambridge University Press. 6, 12

[11] Fabrizio Minganti, Adam Miranowicz, Ravindra W. Chhajlany, and Franco Nori. Quantum exceptional points of non-hermitian hamiltonians and liouvillians: The effects of quantum jumps. 100(6):062131. Publisher: American Physical Society. 13

[12] Dario Sanches. Athene_cunicullariaa.jpg (JPEG image, 1216 × 1549 pixels) - scaled (46%). 22

[13] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. Neural networks take on open quantum systems. 12:74. 14, 19

[14] Sandro Sorella, Michele Casula, and Dario Rocca. Weak binding between two aromatic rings: Feeling the van der waals attraction by quantum monte carlo methods. 127(1):014105. Publisher: American Institute of Physics. 26

[15] M. H. Stone. Linear transformations in hilbert space: III. operational methods and group theory. 16(2):172–175. Publisher: National Academy of Sciences Section: Mathematics. 4

[16] Eric W. Weisstein. Hermitian operator – from wolfram MathWorld. 19

[17] Nobuyuki Yoshioka and Ryusuke Hamazaki. Constructing neural stationary states for open quantum many-body systems. 99(21):214306. 14, 18, 19, 28

[18] Wojciech Hubert Zurek. Decoherence, einselection, and the quantum origins of the classical. 75(3):715–775. Publisher: American Physical Society. 16