

UiO : **Department of Physics**
University of Oslo

Quantum Computing:

Many-Body Methods and Machine Learning

Stian Bilek

Master's Thesis, Autumn 2020



Abstract

In this thesis we have implemented three many-body methods in quantum mechanics on quantum computers, namely the Quantum Phase Estimation (QPE) algorithm, the Variational Quantum Eigensolver (VQE) algorithm and the Quantum Adiabatic Time Evolution (QATE) algorithm. We utilized these methods to approximate the ground state energy of the pairing model Hamiltonian. We provided a comparison between the ideal simulation of a quantum computer and a simulation including the noise model from some of IBM's actual quantum devices. For the VQE algorithm, we also run the method on the IBM Q London five-qubit device. As the time complexity (circuit depth) of a quantum algorithm (quantum circuit) is often the bottleneck on current quantum computers, we also introduced a circuit optimization scheme, called Recursive Circuit Optimization, which can be utilized to approximate the action of a quantum circuit by learning the parameters of a parametrized unitary operation and only evaluating small parts of the circuit at a time. We tested out the method by trying to optimize a random circuit, the QATE algorithm and the unitary coupled cluster doubles (UCCD) ansatz. Finally, we implemented a quantum circuit that allow us to encode a real data set into the amplitudes of a quantum state, and we utilized this circuit to implement several realizations of dense and recurrent neural networks on quantum computers. We proposed neural network layers that are consisting of hardware- and parameter-efficient parametrized quantum circuits, and we have made a Python framework to construct and train deep quantum neural networks. We then tested the neural networks by trying to learn a simple non-linear function as well as finding the smallest eigenvalue of the pairing Hamiltonian Full Configuration Interaction (FCI) matrix.

We found that all the quantum many-body methods showed promise in providing an upper bound on the ground state energy for the ideal simulation of a quantum computer. When performing the simulation with noise and the execution on the real quantum device however, we found that only the VQE algorithm was able to yield reasonable results due to short circuit depth. Still, the success of the method was found to be dependent on the interaction strength g of the quantum system, as the approximation of the ground state energy started to deviate from the actual value as we increased g . For the recursive circuit optimization of a random circuit, we found that we were able to approximate the state given by the random circuit, and we learned that we should expect some error in the approximation due to finite measurements and the flexibility of the parametrized unitary. When utilizing the recursive optimization scheme to approximate the QATE algorithm, we found that all though we were not able to exactly reproduce the action of the operator, we did provide an upper bound to the ground state energy close to the one given by the actual QATE algorithm. In the same time, the circuit depth of the algorithm was drastically decreased. When utilizing the scheme for the VQE algorithm, we got promising results while reducing the circuit depth of the UCCD ansatz.

Over to the neural networks, we managed to approximate a non-linear function utilizing a hardware-efficient three layered network, with only three qubits in the intermediate layers. We showed that our proposed neural network are suitable for near-term quantum computers by successfully approximating the lowest eigenvalue of the FCI matrix when running a simulation with the noise model from the IBM Q London five-qubit computer.

Acknowledgements

I would first like to thank my supervisor, Morten Hjorth-Jensen, for being helpful and motivational throughout my time at the University of Oslo. Your door has always been open when help is needed, which has been a great relief in stressful times.

I would also like to thank my friends and family for believing in me and providing emotional support. Your continuous interest in my studies and well-being has meant a lot to me.

My next thanks goes to Heidi K. J. I highly appreciate your never-ending support.

Finally I would like to thank the computational physics study group. You have been invaluable to me in both a social and educational manner.

Contents

Abstract	i
Acknowledgements	iii
Contents	v
List of Figures	ix
List of Tables	xi
1 Introduction	1
I Machine Learning Algorithms and Standard Many-Body Methods in Quantum Mechanics	3
2 Many-Body Methods in Quantum Mechanics	5
2.1 Slater Determinants	5
2.2 Second Quantization	6
2.3 Reference state	7
2.4 Second Quantized Hamiltonian	7
2.5 Configuration Interaction Theory	8
2.5.1 The Pairing Model	9
2.6 Normal Ordered Hamiltonian	11
2.7 Generalized Wicks Theorem	13
2.8 Coupled Cluster Method	13
2.8.1 Energy equation	14
2.8.2 Amplitude equation	15
3 Supervised Learning	19
3.1 Linear Regression	19
3.2 Logistic Regression	20
3.3 Gradient Descent	21
3.4 Dense Neural Network	22
3.4.1 Backpropagation algorithm	22
3.5 Recurrent Neural Network Layer	24
II Quantum Computing: Machine Learning and Many-Body Methods	25
4 Quantum Computing: Many-Body Methods	27
4.1 Introduction to Quantum Computing	28
4.1.1 Basis	28

4.1.2	Quantum Gates	29
4.1.3	Quantum Circuits	30
	Coin toss example	31
	Circuit depth	32
4.2	Quantum Phase Estimation	33
4.2.1	Quantum Fourier Transform	33
4.2.2	Phase Estimation Algorithm	34
4.2.3	The Suzuki-Trotter transformation	36
4.2.4	The Jordan-Wigner transformation	37
	Jordan-Wigner transformation of Pairing Hamiltonian	37
4.2.5	Hamiltonian Simulation	38
	Getting the complete eigenvalue spectra	40
4.3	Variational Quantum Eigensolvers	41
4.3.1	Max-Cut problem	41
4.3.2	VQE Expectation Values	43
4.3.3	Variational ansatz / Trial state	45
4.3.4	Unitary Coupled Cluster ansatz	46
4.3.5	Simple ansatz for one pair and four spin-orbitals	47
4.3.6	Summary of the variational quantum eigensolver algorithm	48
4.4	Quantum Adiabatic Time Evolution	48
4.5	Validating the results	49
5	Quantum Computing: Machine Learning	51
5.1	Amplitude Encoding	51
5.2	Inner product	53
5.3	Dense Layer on a Quantum Computer	54
5.4	Recurrent Layer on a Quantum Computer	55
5.5	PQC Dense Layer	56
5.5.1	Layers without amplitude encoding	57
5.6	PQC Recurrent Layer	57
5.7	The encoders and ansatzes: U_{enc} and U_a	57
5.8	The entanglers U_{ent}	59
5.9	Parallel calculation of nodes	59
5.10	Learning Unitary Operators	60
5.11	Recursive Circuit Optimization	61
5.12	Eigenvalues with Neural Networks	62
5.12.1	Rayleigh Quotient Minimization [13]	62
III	Implementation	63
6	Methods	65
6.1	Qiskit	65
6.1.1	Circuits and Registers	65
6.1.2	Gates	66
6.1.3	Execute circuit	66
6.1.4	Coin Toss Example	66
6.1.5	Simulating real devices	67
	Noise Model	67
	Basis Gates	67
	Coupling Map	67
	Performing the Simulation	68
6.1.6	Transpiler	68
6.1.7	Quantum Error Correction	68
6.2	Hamiltonian Simulation and Quantum Phase Estimation	69

6.2.1	Quantum Fourier Transformation	69
6.2.2	QPE function	69
6.2.3	Time Evolution Operator	69
6.2.4	Hamiltonian Simulation	70
6.3	Variational Quantum Eigensolvers	72
6.4	Quantum Adiabatic Time Evolution	73
6.5	Quantum Machine Learning	74
6.5.1	Amplitude Encoder	74
6.5.2	Inner Product	75
6.5.3	Neural Network	76
	Encoders and Ansatzes	76
	Entanglers	76
	Setting up a single layer	77
	Setting up a Multi-Layered Neural Network	77
	Learning from a data set	78
6.5.4	Learning Unitary Operators	79
IV	Analysis	83
7	Results	85
7.1	Quantum Phase Estimation	85
7.1.1	Ideal Simulation	85
7.1.2	Noisy Simulation	86
7.1.3	Discussion	87
7.2	Variational Quantum Eigensolver	87
7.2.1	Ideal Simulation	88
7.2.2	Noisy Simulation	89
7.2.3	Execution on IBM Q London five qubit device	90
7.2.4	Discussion	91
7.3	Quantum Adiabatic Time Evolution	92
7.3.1	Ideal Simulation	92
7.3.2	Noisy Simulation	92
7.3.3	Discussion	93
7.4	Quantum Machine Learning	94
7.4.1	Learning the Suzuki-Trotter Approximation	94
	Discussion	95
7.4.2	Learning the Amplitude Encoding of Random Variables	95
	Discussion	97
7.4.3	Recursive Circuit Optimization of Random Circuit	97
	Discussion	99
7.4.4	Recursive Circuit Optimization of the Quantum Adiabatic Time Evolution Algorithm	99
	Discussion	100
7.4.5	Recursive Circuit Optimization of Unitary Coupled Cluster Doubles Ansatz	100
	Discussion	102
7.4.6	Learning a Non-Linear Function with Quantum Neural Network	102
	Discussion	103
7.4.7	Rayleigh Quotient Minimization	103
	Discussion	104
7.4.8	Final Discussion on the Neural Networks	104
	Hardware-efficiency	104
	Exponential advantage in the number of parameters	105
	Expressiveness of the Quantum Neural Networks	105
8	Conclusion	107

Contents

8.1	What we have done	107
8.2	Further Studies	108
8.2.1	Hamiltonian Simulation	108
8.2.2	Hamiltonian Transformation	108
8.2.3	Numerical integration method in Quantum Adiabatic Time Evolution	108
8.2.4	Neural Networks	108
8.2.5	Learning schemes	108
8.2.6	Error estimate for Recursive Circuit Optimization scheme	108
8.2.7	Error Correction	109
Appendices		111
A	CCD Amplitude Equations	113
A.1	Calculations of amplitude equations	113
B	Jordan-Wigner transformation of Pairing Hamiltonian	119
Bibliography		123

List of Figures

2.1	The number of rows/columns required for FCI as a function of spin-orbitals for an $n = 10$ particle system.	9
4.1	The Block sphere. Figure was made by Smite Meister [31]	27
4.2	QFT circuit. Figure is taken from reference [37].	34
4.3	Phase estimation circuit. Figure is taken from reference [37].	35
4.4	Unsolved max-cut graph	42
4.5	Solved max-cut graph	42
6.1	The eigenvalue spectra of an example Hamiltonian found with QPE.	72
7.1	QPE on pairing Hamiltonian (eq. 4.26) with $\delta = 1$, interaction strength $g = 1$ and four spin-orbitals. We plot the amount of times we measure an energy against the energy measurement, for varying number of qubits in the t -register. The FCI energies for four spin-orbitals are marked on the x -axis.	86
7.2	QPE algorithm on pairing Hamiltonian with noise model from the IBM Melbourne Quantum Computer.	87
7.3	Ideal VQE on pairing Hamiltonian with two particles and four spin orbitals. The simple ansatz with one rotation parameter in circuit 4.60 is utilized.	88
7.4	Ideal VQE on Pairing Hamiltonian with four particles and eight spin-orbitals. We utilized the UCCD ansatz explained in section 4.3.4. We compare the results with CCD and FCI for the same system. We plot the ground state energy against the interaction strength g , together with the absolute difference between the FCI energy and the UCCD and CCD energy.	89
7.5	VQE on pairing Hamiltonian with two particles and four spin-orbitals. We utilize the noise model of the IBMQ London 5 qubit computer. We provide the results with and without error correction.	90
7.6	VQE algorithm on IBM Q London five qubit device for pairing Hamiltonian with two particles and four spin-orbitals. We plot the calculated ground state energy with FCI and VQE against the interaction strength g . We also plot as the absolute energy difference between FCI and VQE, against the interaction strength g . We utilized the simple ansatz provided by circuit 4.60.	91
7.7	Ideal QATE simulation for pairing Hamiltonian with two particles and four spin-orbitals.	92
7.8	QATE simulation of pairing Hamiltonian with one pair and four spin-orbitals. The simulation is run with noise model from the five qubit IBMQ London quantum computer. We plot the energy of the system as against the step of the time-ordered exponential.	93
7.9	Approximating a time step of the Suzuki-Trotter approximation utilizing a parametrized ansatz. We plot the absolute value of the inner product between the state given by the Suzuki-Trotter approximation and the state given by the parametrized ansatz, against the iteration of the learning scheme.	95
7.10	The inner product between state produced by a learned ansatz and the amplitude encoding of a normally distributed random vector, against the dimensions of the random vector. The ansatz was given by circuit 5.23. The parameter d is the number of successive applications of the ansatz.	97
7.11	Recursive circuit optimization of random circuit. We plot the absolute value of the inner product between the random circuit state and the corresponding learned ansatz state, against the depth of the random circuit (bottom) and the step of the recursive algorithm (top). This is done for 10^3 and 10^4 measurements in the learning process.	98

7.12	Recursive circuit optimization of the QATE algorithm for pairing Hamiltonian (eq. 4.26) with $\delta = 1$ and $g = 1$. We plot the energy as a function of the step in the recursive scheme.	100
7.13	The VQE algorithm on the pairing Hamiltonian with two particles and four spin orbitals. We approximated the UCCD ansatz utilizing the recursive circuit optimization scheme with two steps and the R_y -rotation ansatz given by circuit 4.50. We plot the ground state energy against the interaction strength g , together with the absolute difference between the FCI energy and the UCCD energy.	101
7.14	Quantum Neural Network trained on the non-linear function $f(x) = 3 \sin x - \frac{1}{2}x$. We plot the actual function values and the values given by the neural network, against x	103
7.15	Rayleigh Quotient minimization of pairing Hamiltonian. We plot the Rayleigh quotient against the interaction strength g . We include the results from both an ideal simulation and a simulation of the IBMQ London 5 qubit quantum computer.	104

List of Tables

6.1	Quantum Neural Network trained on Iris data set. Confusion matrix for separate test set.	79
7.1	8 t-qubit QPE and FCI energy for Pairing model (eq. 4.26). Calculated for four basis states with one and two pairs. $\delta = 1$, $g = 1$. We also provide two standard deviations for the QPE estimate. .	86
7.2	Squared inner product between the states produced by the Euler Rotation ansatz in circuit 4.54 and a Suzuki-Trotter step of the pairing Hamiltonian time evolution operator. We also show the circuit depth of the two circuits.	95
7.3	Comparison of the circuit depth of the amplitude encoder (circuit 5.8) to the depth of the Euler rotation ansatz (circuit 4.54). The parameter d is the number of successive applications of the ansatz, $U_a(\boldsymbol{\theta})$, to the quantum state, that is $U_a(\boldsymbol{\theta}_d)U_a(\boldsymbol{\theta}_{d-1}) \cdots U_a(\boldsymbol{\theta}_1) 0 \cdots 0\rangle$	96
7.4	Learning a random circuit with recursive circuit optimization. We compare the circuit depth of the full random circuit, the circuit depth of the recursive optimization scheme and the circuit depth of the learned ansatz.	99
7.5	Depth of recursive Circuit optimization scheme versus depth of the QATE algorithm.	100
7.6	Recursive optimization scheme of the VQE algorithm with the UCCD Ansatz. We see the depth of the UCCD ansatz and the depth of recursive optimization scheme.	101
7.7	Structure of the neural network used to approximate a non-linear function. The first row corresponds to the input layer, while the final row corresponds to the output layer.	102
7.8	Structure of the neural network used to approximate the eigenvalues of the pairing Hamiltonian. The first row corresponds to the input layer, while the final row corresponds to the output layer. Identity means that we do not apply any operator. In the final layer, we return a number of outputs equal to the dimension of the eigenvector.	104

CHAPTER 1

Introduction

Feynman noted in 1982 that calculating properties of an arbitrary quantum system on a classical device is a seemingly very inefficient thing to do [26]. It is a problem which scales exponentially with the number of particles in the model being simulated, meaning that for complex systems like for example the caffeine molecule, current modelling approaches are at their limits. He suggested that a quantum device might be able to calculate such properties efficiently, meaning that the problem scales polynomially in the number of particles. In 1985, David Deutsch provided a description of such a quantum device, namely an universal quantum computer [15]. Moving forward to 1996, American physicist Seth Lloyd indeed showed that the universal quantum computer could simulate quantum systems efficiently [29].

Today, Quantum computers have moved beyond pen and paper and there are several well known companies working on making these devices a reality, including IBM and Google. IBM even has several quantum computers located around the world, which can be accessed and programmed by anyone over the cloud. Google recently claimed the achievement of quantum supremacy [3] - the task of solving a problem on a quantum computer which cannot be solved on a classical computer in a reasonable amount of time. They performed a task in 200 seconds, which they believed would take approximately 10,000 years for a state-of-the-art classical supercomputer. IBM quickly responded to these claims by showing that tweaking the way the classical supercomputer was set up to solve the problem would decrease the computation time to just a matter of days [39]. Even so, there was no denial that what Google had achieved was an impressive feat.

There are more potential applications for a quantum computer than the efficient simulation of quantum systems. In the financial sector, J.P. Morgan, an American multinational investment bank, are currently hiring quantum computing researchers [8] as there are several potential use cases for quantum computers in this sector. These use cases are believed to include option pricing, portfolio risk calculations, issuance auctions, anti-money laundering operations and identifying fraud. Machine learning is also a field where quantum computers have the potential to outperform classical computers. Support vector machines is an example of a machine learning algorithm where in cases when the classical sampling algorithms require polynomial time, an exponential speed-up is obtained on the universal quantum computer [40]. The 9th of March 2020, Google AI announced TensorFlow Quantum [24], a machine learning library for quantum computers. They believe that with the recent progress in the development of quantum computing, the development of new quantum machine learning models could have a profound impact on the world's biggest problems.

There are still a lot of hurdles that has to be dealt with in order to make these devices useful in practice. With quantum computers we are manipulating fragile quantum systems, incredibly susceptible to external factors such as heat and vibrations. With most of the current devices, this is dealt with in part by the expensive task of cooling the quantum computer chip to near absolute zero. Even so, we are not currently able to maintain a stable quantum state for very long [47]. The practical usage of a quantum computer requires manipulation of the quantum system and this act takes time, which in turn is giving us noisy results. Noisy Intermediate-Scale Quantum (NISQ) technology is the common term for these current noisy quantum devices, and there are lots of research done on algorithms to mitigate the noise in the execution of a quantum computer algorithm (quantum circuits). Even computationally efficient algorithms are hard to run on NISQ devices, as a quantity called circuit

1. Introduction

depth [19], roughly stating the execution time of the algorithm, is often too large for current devices to handle. Hence, one is not only looking for computationally efficient algorithms, but also for algorithms with a sufficiently short circuit depth.

In this thesis we are going to implement three many-body methods in quantum mechanics on quantum computers, namely the Quantum Phase Estimation (QPE) algorithm [37], the Variational Quantum Eigensolvers (VQE) algorithm [44] and the Quantum Adiabatic Time Evolution (QATE) algorithm [16]. We will utilize these methods to approximate the ground state energy of the pairing model Hamiltonian and benchmark them against the ground state energy given by standard methods such as Full Configuration Interaction (FCI) theory [50] and the Coupled Cluster doubles (CCD) method [12]. We will provide a comparison between the ideal simulation of a quantum computer, a simulation with the noise model from some of IBM's actual quantum devices and the execution on a real quantum device. We will also look at how to approximate quantum circuits with less complex parametrized circuits utilizing machine learning. As the depth of a circuit is often the bottleneck on NISQ devices, we will also introduce a circuit optimization machine learning scheme called Recursive Circuit Optimization. This scheme can be utilized to approximate the action of a quantum circuit by learning a parametrized unitary operator and only evaluating small parts of the circuit at a time. Finally, we will represent several dense and recurrent neural network architectures on quantum computers utilizing parametrized quantum circuits (PQC), as well as the encoding of data sets into the amplitudes of a quantum state. A framework will be introduced that allows for easy construction and training of such deep quantum neural networks. All the quantum computing methods in this thesis will be implemented utilizing Qiskit [42] - an open-source Python framework for quantum computing developed by IBM Research.

In part I of this thesis, we will go through the theory of standard many-body methods in quantum mechanics, as well as an introduction to machine learning. The former is done in chapter 2, where we will explain the FCI method and the CCD method. The introduction to machine learning is done in chapter 3, where the goal is to get the reader familiar with neural networks.

In part II we move over to the theory of programming quantum computers. We start this part with chapter 4, which gives an introduction to quantum computing as well as representing the theory behind the QPE, the VQE and the QATE algorithm. We then move over to chapter 5, where we represent machine learning methods on quantum computers.

Part III then goes through how all the quantum computing algorithms were implemented using Python. Finally we represent and analyse our results in part IV. In the first chapter of this part, chapter 7, we show and discuss our results. We finally conclude our analysis and discuss further studies in chapter 8.

All the code utilized to produce our results can be found in the GitHub page for this thesis: <https://github.com/stiandb/Thesis>

PART I

Machine Learning Algorithms and Standard Many-Body Methods in Quantum Mechanics

CHAPTER 2

Many-Body Methods in Quantum Mechanics

Consider the fact that any function in a function space can be represented as a linear combination of basis functions, just like any vector in a vector space can be represented as a linear combination of basis vectors. For quantum mechanics one could argue that since quantum states can be represented by mathematical functions, we should be able to represent any state provided we have a complete set of basis functions as well [1]. This property of basis functions is at the core of a variety of many-body methods in quantum mechanics, as well as the ones we will utilize in this thesis to approximate the ground state energy for quantum systems. Hence, we will start by going through how we define this basis.

2.1 Slater Determinants

Given a complete set of single-particle states $\psi_j(\mathbf{x}_i)$, where j is the relevant quantum numbers and \mathbf{x}_i is the position of particle i , one of the simplest ways to represent an n -particle state is as a simple product of single-particle states:

$$\Psi_{j_1, j_2, \dots, j_n}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) = \psi_{j_1}(\mathbf{x}_1) \psi_{j_2}(\mathbf{x}_2) \dots \psi_{j_n}(\mathbf{x}_n). \quad (2.1)$$

Fermionic systems like the ones of interest in this thesis have to be anti-symmetric when two particles are exchanged. Anti-symmetry with respect to the exchange of two particles means that

$$\begin{aligned} & \Psi_{j_1, j_2, \dots, j_n}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k, \dots, \mathbf{x}_l, \dots, \mathbf{x}_n) \\ &= -\Psi_{j_1, j_2, \dots, j_n}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_l, \dots, \mathbf{x}_k, \dots, \mathbf{x}_n). \end{aligned}$$

The product state in equation 2.1 unfortunately does not have this property. We can however achieve this by expressing the state as a Slater determinant (SD) [50]

$$\Psi_{j_1, j_2, \dots, j_n}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) = \frac{1}{\sqrt{n!}} \begin{vmatrix} \psi_{j_1}(\mathbf{x}_1) & \psi_{j_1}(\mathbf{x}_2) & \psi_{j_1}(\mathbf{x}_3) & \dots & \psi_{j_1}(\mathbf{x}_n) \\ \psi_{j_2}(\mathbf{x}_1) & \psi_{j_2}(\mathbf{x}_2) & \psi_{j_2}(\mathbf{x}_3) & \dots & \psi_{j_2}(\mathbf{x}_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \psi_{j_n}(\mathbf{x}_1) & \psi_{j_n}(\mathbf{x}_2) & \psi_{j_n}(\mathbf{x}_3) & \dots & \psi_{j_n}(\mathbf{x}_n) \end{vmatrix}. \quad (2.2)$$

For example, a two particle state can be expressed with the following SD

$$\Psi_{j_1, j_2}(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{2!} \begin{vmatrix} \psi_{j_1}(\mathbf{x}_1) & \psi_{j_1}(\mathbf{x}_2) \\ \psi_{j_2}(\mathbf{x}_1) & \psi_{j_2}(\mathbf{x}_2) \end{vmatrix} = \frac{1}{2} [\psi_{j_1}(\mathbf{x}_1) \psi_{j_2}(\mathbf{x}_2) - \psi_{j_1}(\mathbf{x}_2) \psi_{j_2}(\mathbf{x}_1)]. \quad (2.3)$$

The exchange of the two particles is done by replacing \mathbf{x}_1 with \mathbf{x}_2 and vice versa

$$\begin{aligned} \Psi_{j_1, j_2}(\mathbf{x}_2, \mathbf{x}_1) &= \frac{1}{2} [\psi_{j_1}(\mathbf{x}_2) \psi_{j_2}(\mathbf{x}_1) - \psi_{j_1}(\mathbf{x}_1) \psi_{j_2}(\mathbf{x}_2)] \\ &= -\Psi_{j_1, j_2}(\mathbf{x}_1, \mathbf{x}_2), \end{aligned}$$

which we can see is anti-symmetric. Given that we have a complete single-particle basis set $\{\psi_j\}$, all possible n -particle SDs formed with this basis set span a complete n -particle basis and can hence in theory express any n -particle wave function [50]. We will use the following shorthand notation for the SD in equation 2.2

$$|j_1 j_2 \dots j_n\rangle, \quad (2.4)$$

2. Many-Body Methods in Quantum Mechanics

where j_i refers to single-particle basis state ψ_{j_i} . The leftmost element in this ket refers to the state occupied by the first particle, while the second element refers to the state the second particle occupies, etc. Normally, a subscript of AS, $|j_1 j_2 \cdots j_n\rangle_{AS}$, is used to specify that our state is anti-symmetric, but we will drop this notation and you can safely assume that the state is anti-symmetric unless otherwise specified. Note that all the SD's $|\Psi_i\rangle$ constructed with a single particle basis are orthonormal, that is

$$\langle \Psi_i | \Psi_j \rangle = \delta_{ij}, \quad (2.5)$$

where $\delta_{ij} = 1$ if $i = j$, else $\delta_{ij} = 0$.

2.2 Second Quantization

Now that we can express an arbitrary n -particle state, we are ready to introduce the second quantization formalism [22]. Second quantization is a formalism used to analyze and describe many-body systems in quantum mechanics. We will learn that we can utilize it as a convenient way to manipulate Slater determinants and achieve a linear combination of them which solves our problem. To do this, we need to introduce the creation operator a_j^\dagger and its hermitian conjugate, the annihilation operator a_j . The creation operator has the following effect on an SD

$$a_{j_k}^\dagger |j_1 j_2 \cdots j_n\rangle = \begin{cases} 0, & \text{if } j_k \in \{j_1, j_2, \cdots j_n\} \\ |j_k j_1 j_2 \cdots j_n\rangle & \text{otherwise.} \end{cases} \quad (2.6)$$

That is, it creates a particle in the state ψ_{j_k} if the state is not contained in the SD, else it terminates the SD. For the annihilation operator we have

$$a_{j_k} |j_1 j_2 \cdots j_n\rangle = \begin{cases} 0, & \text{if } j_k \notin \{j_1, j_2, \cdots j_n\} \\ (-1)^{k-1} |j_1 j_2 \cdots \cancel{j_k} \cdots j_n\rangle & \text{otherwise,} \end{cases} \quad (2.7)$$

where $\cancel{j_k}$ indicates that we have removed the particle in the ψ_{j_k} state from our SD. The factor $(-1)^{k-1}$ ensures that the anti-symmetric properties are preserved. We will also define a vacuum state, which is a state that is not occupied by any particle, denoted $|0\rangle$. The action of the creation and annihilation operators on our vacuum state is

$$a_j^\dagger |0\rangle = |j\rangle \quad \text{and} \quad a_j |0\rangle = 0.$$

We also have that

$$a_j |j\rangle = |0\rangle,$$

and that any SD can be created from our reference vacuum with the application of creation operators

$$|j_1 j_2 \cdots j_n\rangle = a_{j_1}^\dagger a_{j_2}^\dagger \cdots a_{j_n}^\dagger |0\rangle. \quad (2.8)$$

The anti-commutation rules for our creation and annihilation operators is an important tool that later down the road will enable us to rewrite our equations in convenient ways. The first rule can be shown by considering the anti-symmetric property of SD's

$$\begin{aligned} |j_1 j_2 \cdots j_p \cdots j_q \cdots j_n\rangle &= -|j_1 j_2 \cdots j_q \cdots j_p \cdots j_n\rangle \\ \implies a_p^\dagger a_q^\dagger |j_1 j_2 \cdots j_n\rangle &= -a_q^\dagger a_p^\dagger |j_1 j_2 \cdots j_n\rangle \\ \implies a_p^\dagger a_q^\dagger &= -a_q^\dagger a_p^\dagger. \end{aligned}$$

Which gives

$$\{a_p^\dagger, a_q^\dagger\} = a_p^\dagger a_q^\dagger + a_q^\dagger a_p^\dagger = 0.$$

Taking the conjugate transpose of this gives

$$\{a_p, a_q\} = 0.$$

One can also find the anti-commutation rule for $a_p^\dagger a_q$ [22]. All the anti-commutation rules are summarized as

$$\{a_p^\dagger, a_q\} = \delta_{pq} \quad \text{and} \quad \{a_p^\dagger, a_q^\dagger\} = \{a_p, a_q\} = 0, \quad (2.9)$$

where $\delta_{pq} = 1$ if $p = q$, else it is 0.

2.3 Reference state

For a variety of systems in many-body quantum mechanics, as well as the ones of interest in this thesis, one assumes that the Hamiltonian in question consists of a one-body operator \hat{H}_0 and a two-body operator \hat{H}_1 . One can normally solve for the eigenstates $|\psi_k^0\rangle$ of \hat{H}_0 analytically and utilize these states as the single-particle basis from which to express our SD's. We do this so the operator \hat{H}_0 becomes diagonal in our basis, that is

$$\langle \psi_i^0 | \hat{H}_0 | \psi_j^0 \rangle = \epsilon_j \delta_{ij}.$$

We will eventually see that this is going to make our calculations a bit less cumbersome down the road. Assume that ψ_1^0 is the single-particle ground state of our one-body operator, and that the state yielding the next to lowest energy is ψ_2^0 , etc. For an n -body problem, the state yielding the lowest energy for our one-body operator is then given by

$$|\Psi_0\rangle \equiv |c\rangle \equiv a_1^\dagger a_2^\dagger \cdots a_{n-1}^\dagger a_n^\dagger |0\rangle \equiv |\psi_1^0 \psi_2^0 \cdots \psi_n^0\rangle \equiv |1 \ 2 \ 3 \cdots n\rangle. \quad (2.10)$$

We will refer to the state in eq. 2.10 as the reference state for our problem. The single-particle states contained in this SD are said to be below the Fermi (F) level and we will denote these states with the letters $ijkl \leq F$. States above the Fermi level will likewise be denoted with $abcd > F$. Every n -particle SD can be written as some product of excitation and annihilation operators acting on the reference state $|\Psi_0\rangle$, given that the operator product commutes with the number operator

$$\sum_q a_q^\dagger a_q,$$

that is, if it preserves the number of particles. An example is the following state

$$|\Psi_1^{n+1}\rangle = a_{n+1}^\dagger a_1 |c\rangle = |(n+1) \ 2 \ 3 \cdots n\rangle. \quad (2.11)$$

States like these will be referred to as 1hole-1particle (1h-1p) states as we have created a hole in the reference state and added a particle to a new spin orbital. Further, an example of a 2p-2h state is

$$|\Psi_{2,3}^{(n+3),(n+6)}\rangle = a_{n+3}^\dagger a_{n+6}^\dagger a_3 a_2 |c\rangle = |1 \ 4 \cdots n \ (n+3) \ (n+6)\rangle. \quad (2.12)$$

In general, one can write any excited state as

$$|\Psi_{ij\cdots}^{ab\cdots}\rangle. \quad (2.13)$$

When dealing with many-particle systems, writing every n -particle state by applying creation operators on $|0\rangle$ can become quite cumbersome. It is therefore useful to start from the reference state in eq. 2.10 and write excited states by applying the creation and annihilation operators on said like we did in eqs. 2.11 and 2.12.

2.4 Second Quantized Hamiltonian

We are now ready to write out our Hamiltonian in the second quantization formalism. Assume that our Hamiltonian can be written in terms of a one-body operator and a two-body operator, that is

$$\hat{H} = \hat{H}_0 + \hat{H}_1. \quad (2.14)$$

Let's start with writing the one-body part in terms of our single-particle basis $|p\rangle$. This can be done by utilizing the identity operator $\sum_p |p\rangle \langle p|$:

$$\hat{H}_0 = \sum_p |p\rangle \langle p| \hat{H}_0 \sum_q |q\rangle \langle q| = \sum_{pq} |p\rangle \langle p| \hat{H}_0 |q\rangle \langle q|.$$

Further we know that $|p\rangle = a_p^\dagger |0\rangle$ and $\langle q| = \langle 0| a_q$ (eqs. 2.6 and 2.7). This gives

$$\hat{H}_0 = \sum_{pq} \langle p| \hat{H}_0 |q\rangle a_p^\dagger |0\rangle \langle 0| a_q.$$

If we consider that

$$\langle u| a_p^\dagger |0\rangle \langle 0| a_q |v\rangle = \delta_{pu} \delta_{qv} = \langle u| a_p^\dagger a_q |v\rangle,$$

we get the following property

$$a_p^\dagger |0\rangle \langle 0| a_q = a_p^\dagger a_q.$$

We then get

$$\hat{H}_0 = \sum_{pq} \langle p | \hat{H}_0 | q \rangle a_p^\dagger a_q. \quad (2.15)$$

We can now clearly see why we normally chose our single-particle basis as solutions to the one-body Hamiltonian, as this will make the matrix $H_{pq} = \langle p | \hat{H}_0 | q \rangle$ diagonal in our basis. This gives us

$$\hat{H}_0 = \sum_p \langle p | \hat{H}_0 | p \rangle a_p^\dagger a_p. \quad (2.16)$$

For the two-body part we follow the same procedure, but being a two-body operator the basis exchange requires two-particle SD's:

$$\hat{H}_1 = \sum_{pq} |pq\rangle \langle pq| \hat{H}_1 \sum_{rs} |rs\rangle \langle rs| = \sum_{pqrs} |pq\rangle \langle pq| \hat{H}_1 |rs\rangle \langle rs|.$$

Following the same procedure as with the one-body Hamiltonian gives us

$$\hat{H}_1 = \frac{1}{4} \sum_{pqrs} \langle pq | \hat{H}_1 | rs \rangle a_p^\dagger a_q^\dagger a_s a_r. \quad (2.17)$$

The full Hamiltonian in second quantized form is then [22]

$$\hat{H} = \sum_{pq} \langle p | \hat{H}_0 | q \rangle a_p^\dagger a_q + \frac{1}{4} \sum_{pqrs} \langle pq | \hat{H}_1 | rs \rangle a_p^\dagger a_q^\dagger a_s a_r. \quad (2.18)$$

We are now ready to introduce the first many-body method in this thesis, namely Configuration Interaction (CI) Theory.

2.5 Configuration Interaction Theory

We want to solve the time-independent Schrödinger equation

$$\hat{H} |\Phi_k\rangle = (\hat{H}_0 + \hat{H}_1) |\Phi_k\rangle = \epsilon_k |\Phi_k\rangle. \quad (2.19)$$

We just learned that an arbitrary antisymmetric n -particle wave function can be written as a linear combination of Slater determinants. If we choose the K eigenfunctions ψ_k of \hat{H}_0 as our basis, and consider the set $\{|\Psi_0\rangle, |\Psi_i^a\rangle, |\Psi_{ij}^{ab}\rangle, \dots\}$ to be all possible n -particle SD's formed with said basis, the solution to eq. 2.19 is then given by

$$|\Phi_k\rangle = c_0^{(k)} |\Psi_0\rangle + \sum_{ia} c_i^{a(k)} |\Psi_i^a\rangle + \sum_{i<j, a<b} c_{ij}^{ab(k)} |\Psi_{ij}^{ab}\rangle + \dots, \quad (2.20)$$

where we assume the coefficients c to be complex numbers. In order to solve for the coefficients, let's first rewrite our Hamiltonian in the $|\Psi_p\rangle$ basis by utilizing the fact that $\sum_p |\Psi_p\rangle \langle \Psi_p| = I$:

$$\hat{H} = \sum_{pq} |\Psi_p\rangle \langle \Psi_p| \hat{H} |\Psi_q\rangle \langle \Psi_q|. \quad (2.21)$$

By multiplying both sides with $\langle \Psi_l|$ and using eq. 2.19 and 2.20, we see that

$$\langle \Psi_l | \hat{H} | \Phi_k \rangle = \sum_q \langle \Psi_l | \hat{H} | \Psi_q \rangle c_q = \epsilon_k c_l. \quad (2.22)$$

This can be written in matrix notation as [50]

$$\begin{bmatrix} \langle \Psi_0 | \hat{H} | \Psi_0 \rangle & \langle \Psi_0 | \hat{H} | \Psi_1 \rangle & \dots & \langle \Psi_0 | \hat{H} | \Psi_{\binom{K}{n}} \rangle \\ \langle \Psi_1 | \hat{H} | \Psi_0 \rangle & \langle \Psi_1 | \hat{H} | \Psi_1 \rangle & \dots & \langle \Psi_1 | \hat{H} | \Psi_{\binom{K}{n}} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \Psi_{\binom{K}{n}} | \hat{H} | \Psi_0 \rangle & \langle \Psi_{\binom{K}{n}} | \hat{H} | \Psi_1 \rangle & \dots & \langle \Psi_{\binom{K}{n}} | \hat{H} | \Psi_{\binom{K}{n}} \rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{\binom{K}{n}} \end{bmatrix} = \epsilon_k \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{\binom{K}{n}} \end{bmatrix}, \quad (2.23)$$

where we have introduced the binomial coefficient $\binom{K}{n}$ as this represents the ways to choose an (unordered) subset of n elements from a fixed set of K elements. We will refer to this matrix as the Full Configuration Interaction (FCI) matrix. We can see that we can in principle solve eq. 2.19 exactly by finding the eigenfunctions and eigenvalues of the above matrix. In practice however, we require an infinite number of single-particle wave functions, and hence an infinite number of SD's to form a complete set. We will therefore have to truncate the basis at a fixed number K , and thus not achieve the exact solutions to our problem. Nevertheless, the solutions will be exact within the n -particle subspace spanned by our Slater determinants [50]. As we need $\binom{K}{n}$ rows/columns to represent the above matrix, let's plot how this number grows for a set n and as a function of K . For an $n = 10$ particle system we have

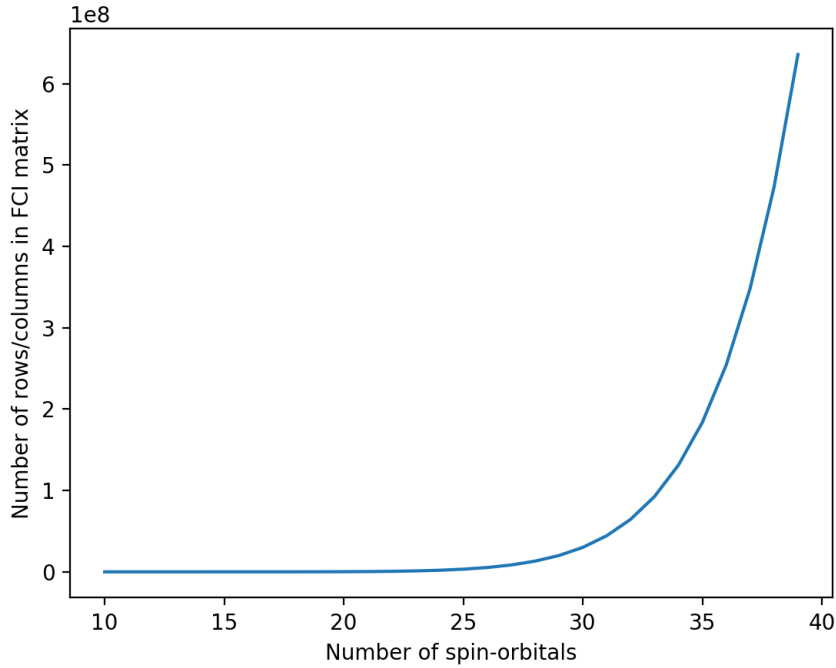


Figure 2.1: The number of rows/columns required for FCI as a function of spin-orbitals for an $n = 10$ particle system.

We can easily see from this plot how utilizing FCI can become a problem for complex systems, as the number of matrix elements quickly explodes as the number of spin-orbitals increases. However, as this method is exact within the n -particle subspace spanned by our Slater determinants, we will use it as a base-line to compare our other methods with. We will now derive the FCI-matrix for the Pairing model.

2.5.1 The Pairing Model

In condensed matter physics, a Cooper pair is two electrons that are bound together at low temperatures, first described in 1956 by Leon Cooper [11]. In collaboration with John Bardeen and John Schrieffer, Leon Cooper developed the BCS theory for which they received the Nobel Prize in Physics in 1972. A (very) brief summary of

this theory is that the Cooper pair state is responsible for superconductivity. The idea that electrons, which we know repel each other, can form bounded pairs may seem far-fetched. Even though Cooper pairing is a quantum effect, one can explain the concept with a simplified classical explanation. If one considers electrons in a metal, one electron is repelled from other electrons due to their negative charge. The electron also attracts the positive ions that make up the lattice of the metal. This attraction causes the ions to move slightly toward the electron, increasing the positive charge density in its vicinity. This increase of charge density attracts other electrons, and this attraction can at long distances overcome the electrons repulsion due to their negative charge, thus causing them to pair up. These realizations motivate the introduction of pairing interactions when studying certain many-body systems and their properties.

One such system is the simple Pairing model of an ideal fermionic system. The Hamiltonian can be written on the form [23]

$$\begin{aligned}\hat{H} &= \hat{H}_0 + \hat{H}_1 \\ &= \sum_p \epsilon_p a_p^\dagger a_p - \frac{1}{2} g \sum_{pq} a_{p+}^\dagger a_{p-}^\dagger a_{q-} a_{q+},\end{aligned}\quad (2.24)$$

where ϵ_p is the single-particle energy of level $p = 1, 2, \dots$ and g is the pairing strength. Treating the pairing strength as a constant is a simplification, but it still serves as a useful tool when studying the pairing interaction. We will assume that the single-particle energy levels are equally spaced and spin-independent, thus we can write $\epsilon_p = (p-1)\xi$ and introduce a sum over the spin values $\sigma = \pm$. This gives us

$$\hat{H} = \xi \sum_{p\sigma} (p-1) \hat{n}_p - \frac{1}{2} g \sum_{pq} P_p^+ P_q^-, \quad (2.25)$$

where $\hat{n}_p = a_p^\dagger a_p$. The operators $P_p^+ = a_{p+}^\dagger a_{p-}^\dagger$ and $P_p^- = a_{p-} a_{p+}$ will be referred to as the pair-creation and pair-annihilation operators, respectively. When finding the FCI matrix for the pairing Hamiltonian, the number of SD's needed corresponds to the ways to choose an (unordered) subset of k particle orbitals from a fixed set of p pairs. Thus we have

$$\text{Number of SD's} = \binom{k}{p}. \quad (2.26)$$

Since the particle pairs can not be broken, we can define the SD basis in terms of the pair-creation and annihilation operators

$$|\Phi_{ij\dots}^{ab\dots}\rangle = P_a^+ P_b^+ \dots P_k^- P_j^- P_i^- |\Phi_0\rangle, \quad (2.27)$$

with the reference state defined as

$$|\Phi_0\rangle = \left(\prod_{i \leq F} P_i^\dagger \right) |0\rangle. \quad (2.28)$$

The letter F denotes the Fermi level. If Φ_s represents element s in our SD basis set $\{\Phi_0, \Phi_{ij\dots}^{ab\dots}\}$, the matrix elements of eq. 2.25 are given by

$$H_{st} = \langle \Phi_s | \hat{H} | \Phi_t \rangle = \langle \Phi_s | \hat{H}_0 | \Phi_t \rangle + \langle \Phi_s | \hat{H}_1 | \Phi_t \rangle. \quad (2.29)$$

We can calculate the one and two-body part separate. Let us first deal with the one-body part:

$$\langle \Phi_s | \hat{H}_0 | \Phi_t \rangle = \xi \sum_{p\sigma} (p-1) \langle \Phi_s | \hat{n}_p | \Phi_t \rangle. \quad (2.30)$$

As we know that each spacial state is occupied with an electron pair with opposite spin, we can multiply the term with 2 instead of summing over σ . Further, \hat{n}_p only gives us non-zero contributions if there is a particle in the p -orbital, hence $\langle \Phi_s | \hat{n}_p | \Phi_t \rangle = \delta_{st}$. With this information applied, we get

$$\langle \Phi_s | \hat{H}_0 | \Phi_s \rangle = 2\xi \sum_p (p-1) \delta_{p \in \Phi_s}, \quad (2.31)$$

where the Kronecker delta $\delta_{p \in \phi_i}$ is zero unless state p is occupied in Φ_s .

For the two-body part we are dealing with the operator $P_p^+ P_q^-$, which can move an existing particle pair to a different spin-orbital. Hence, if Φ_s and Φ_t in $\langle \Phi_s | \hat{H}_1 | \Phi_t \rangle$ has more than one pair in a different orbital, we get no contribution. Let's cover the diagonal elements first:

$$\begin{aligned} \langle \Phi_{ij\dots}^{ab\dots} | \hat{H}_1 | \Phi_{ij\dots}^{ab\dots} \rangle &= -\frac{1}{2}g \sum_{pq} \langle \Phi_{ij\dots}^{ab\dots} | P_p^+ P_q^- | \Phi_{ij\dots}^{ab\dots} \rangle \\ &= -\frac{1}{2}g \sum_{pq} \langle \Phi_{ij\dots}^{ab\dots} | \delta_{pq,p \in \Phi_{ij\dots}^{ab\dots}} | \Phi_{ij\dots}^{ab\dots} \rangle \\ &= -\frac{1}{4}g \cdot n_p, \end{aligned} \quad (2.32)$$

where n_p is the number of particles and $\delta_{pq,p \in \Phi_{ij\dots}^{ab\dots}}$ is zero unless $p = q$ and p is occupied in $\Phi_{ij\dots}^{ab\dots}$. For the off diagonal elements, we have two different scenarios. The first is that we have one pair different above the Fermi level:

$$\begin{aligned} \langle \Phi_{ij\dots}^{ab\dots} | \hat{H}_1 | \Phi_{ij\dots}^{ac\dots} \rangle &= -\frac{1}{2}g \sum_{pq} \langle \Phi_{ij\dots}^{ab\dots} | P_p^+ P_q^- | \Phi_{ij\dots}^{ac\dots} \rangle \\ &= -\frac{1}{2}g \sum_{pq} \delta_{pb} \delta_{qc} = -\frac{1}{2}g. \end{aligned} \quad (2.33)$$

This solution can be explained by the fact that P_q^- will have to remove the pair in the c state from the ket and P_p^+ insert a pair in the b state, since the SD basis is orthonormal. The second case regards when we have one different pair below the Fermi level:

$$\begin{aligned} \langle \Phi_{ij\dots}^{ab\dots} | \hat{H}_1 | \Phi_{ik\dots}^{ab\dots} \rangle &= -\frac{1}{2}g \sum_{pq} \langle \Phi_{ij\dots}^{ab\dots} | P_p^+ P_q^- | \Phi_{ik\dots}^{ab\dots} \rangle \\ &= -\frac{1}{2}g \sum_{pq} \delta_{pj} \delta_{qk} = -\frac{1}{2}g, \end{aligned} \quad (2.34)$$

where we have applied the same reasoning as we did for eq. 2.33. As we have already discussed, the number of terms required for the CI Matrix in eq. 2.23 scales poorly with the number of basis states. Fortunately there is another well known method which scales better with the system size, namely Coupled Cluster method [12]. The derivation for this method requires our second quantization Hamiltonian to be what we call normal ordered form, and the application of what is known as Generalized Wick's theorem. We will now go through these topics.

2.6 Normal Ordered Hamiltonian

A string of operators are normal ordered if all annihilation operators are grouped to the right of the creation operators. Since we will start working from our reference state (eq. 2.10), we will from now on define an annihilation operator as an operator \hat{A} which, with respect to our reference state, yields

$$\hat{A} |c\rangle = 0.$$

Likewise, a creation operator will be defined as an operator which, with respect to our reference state, yields

$$\hat{B} |c\rangle \neq 0.$$

We will denote a string of operators $\hat{A}\hat{B}\hat{C}\dots$ as being normal ordered with the convention

$$\{\hat{A}\hat{B}\hat{C}\dots\}. \quad (2.35)$$

We might have to move one or several operators to the left in the string in order to achieve the normal ordering. For every move, one has to multiply the operator string with -1 . For example, say we want to write out $\{\hat{A}\hat{B}\hat{C}\}$ and we know that the normal order is $\hat{B}\hat{A}\hat{C}$. We then have

$$\{\hat{A}\hat{B}\hat{C}\} = -\hat{B}\hat{A}\hat{C},$$

2. Many-Body Methods in Quantum Mechanics

as we had to move \hat{B} to the left to achieve the normal ordering. Since the string $\{\hat{A}\hat{B}\hat{C}\dots\}$ is normal ordered, we have that the action of it on our reference state $|c\rangle$ is

$$\{\hat{A}\hat{B}\hat{C}\dots\}|c\rangle = 0. \quad (2.36)$$

Normal ordering of the second quantized Hamiltonian given in eq. 2.18 is required in the derivations of the Coupled Cluster equations we soon will encounter. Hence, we will now go through this process. We will use the convention that i, j, k, l, \dots refers to states up to the Fermi level and that a, b, c, d, \dots refers to states above the Fermi level. Then we will split up the sum in eq. 2.18 accordingly. For the one-body part we have

$$\begin{aligned} \sum_{pq} \langle p | \hat{H}_0 | q \rangle a_p^\dagger a_q &= \sum_{iq} \langle i | \hat{H}_0 | q \rangle a_i^\dagger a_q + \sum_{aq} \langle a | \hat{H}_0 | q \rangle a_a^\dagger a_q \\ &= \sum_{ij} \langle i | \hat{H}_0 | j \rangle a_i^\dagger a_j + \sum_{ia} \langle i | \hat{H}_0 | a \rangle a_i^\dagger a_a \\ &\quad + \sum_{ai} \langle a | \hat{H}_0 | i \rangle a_a^\dagger a_i + \sum_{ab} \langle a | \hat{H}_0 | b \rangle a_a^\dagger a_b. \end{aligned}$$

All the sums except for the one over ij consists of normal ordered operators. In order to rewrite the ij sum in normal ordered form, we will use the following relation

$$a_p^\dagger a_q = \delta_{pq} - a_q a_p^\dagger,$$

which is derived from the anti-commutation rule in eq. 2.9. We will insert $\delta_{ij} - a_j a_i^\dagger$ for $a_i^\dagger a_j$. This will finally give us

$$\begin{aligned} \sum_{pq} \langle p | \hat{H}_0 | q \rangle a_p^\dagger a_q &= \sum_i \langle i | \hat{H}_0 | i \rangle - \sum_{ij} \langle i | \hat{H}_0 | j \rangle a_j a_i^\dagger \\ &\quad + \sum_{ai} \langle a | \hat{H}_0 | i \rangle a_a^\dagger a_i + \sum_{ia} \langle i | \hat{H}_0 | a \rangle a_i^\dagger a_a \\ &\quad + \sum_{ab} \langle a | \hat{H}_0 | b \rangle a_a^\dagger a_b. \end{aligned}$$

Which can be summarized as

$$\sum_{pq} \langle p | \hat{H}_0 | q \rangle a_p^\dagger a_q = \sum_i \langle i | \hat{H}_0 | i \rangle + \sum_{pq} \langle p | \hat{H}_0 | q \rangle \{a_p^\dagger a_q\},$$

where $\{\cdot\}$ denotes the normal order. The same procedure on the two-body part of the Hamiltonian yields

$$\begin{aligned} \frac{1}{4} \sum_{pqrs} \langle pq | \hat{H}_1 | rs \rangle a_p^\dagger a_q^\dagger a_s a_r &= \frac{1}{4} \sum_{pqrs} \langle pq | \hat{H}_1 | rs \rangle \{a_p^\dagger a_q^\dagger a_s a_r\} + \sum_{pq i} \langle p i | \hat{H}_1 | q i \rangle \{a_p^\dagger a_q\} \\ &\quad + \frac{1}{2} \sum_{ij} \langle i j | \hat{H}_1 | i j \rangle. \end{aligned}$$

The full Hamiltonian in normal ordered form is then [22]

$$\begin{aligned} \hat{H} &= E_{ref} + \sum_{pq} \hat{f}_p^q \{a_p^\dagger a_q\} + \frac{1}{4} \sum_{pqrs} \langle pq | \hat{H}_1 | rs \rangle \{a_p^\dagger a_q^\dagger a_s a_r\} \\ &= E_{ref} + \hat{H}_N, \end{aligned} \quad (2.37)$$

with

$$E_{ref} = \sum_i \langle i | \hat{H}_0 | i \rangle + \frac{1}{2} \sum_{ij} \langle i j | \hat{H}_1 | i j \rangle,$$

and

$$\hat{f}_p^q = \langle p | \hat{H}_0 | q \rangle + \sum_i \langle p i | \hat{H}_1 | q i \rangle.$$

2.7 Generalized Wicks Theorem

Now we are ready to state the Generalized Wicks Theorem. We define the contraction between two operators A and B as

$$\overline{AB} = \langle c | AB | c \rangle. \quad (2.38)$$

If we calculate contractions between strings of normal ordered operators

$$\{ABC \dots\} \{NMO \dots\},$$

we get

$$\{ABC \dots\} \{NMO \dots\} = (-1)^l [\langle c | AN | c \rangle + \langle c | BM | c \rangle + \langle c | CO | c \rangle + \dots], \quad (2.39)$$

where l is the number of crossing contraction lines. Given two strings of normal ordered operators $\{ABC \dots\}$ and $\{NM \dots\}$, Generalized Wicks theorem states that [53]

$$\begin{aligned} \{ABC \dots\} \{NMO \dots\} &= \{ABC \dots NMO \dots\} \\ &+ \sum_{\text{singlecontractions}} \{ABC \dots\} \{NMO \dots\} \\ &+ \sum_{\text{doublecontractions}} \{ABC \dots\} \{NMO \dots\} \\ &+ \dots \\ &+ \sum_{\text{fullycontracted}} \{ABC \dots\} \{NMO \dots\}. \end{aligned} \quad (2.40)$$

This theorem also extends for an arbitrary number of strings of normal ordered operators. An important thing to realise about eq. 2.40 is that all the terms except the fully contracted terms will be zero. This can be explained by the fact that any of the other terms will have an annihilation operator to the far right, giving zero when acting on our reference state. Hence, we can evaluate any combination of normal ordered operator strings by only considering two and two operators and the expectation value in eq. 2.38. We are now ready to go into the Coupled Cluster (CC) method.

2.8 Coupled Cluster Method

In Coupled Cluster theory [12], we start out with an ansatz for the ground state

$$|\Psi_{CC}\rangle = e^{\hat{T}} |c\rangle, \quad (2.41)$$

where $|c\rangle$ is our reference state (eq. 2.10) and \hat{T} is the cluster operator

$$\begin{aligned} \hat{T} &= \hat{T}_1 + \hat{T}_2 + \dots \\ \hat{T}_m &= \left(\frac{1}{m!} \right)^2 \sum_{abc \dots} \sum_{ijk \dots} t_{ijk \dots}^{abc \dots} a_a^\dagger a_b^\dagger a_c^\dagger \dots a_k a_j a_i. \end{aligned} \quad (2.42)$$

Here m is the number of particles the operator excites and $t_{ijk \dots}^{abc \dots}$ are the cluster amplitudes. The Schrödinger equation (eq. 2.19) then becomes

$$\hat{H} |\Psi_{CC}\rangle = E_{CC} |\Psi_{CC}\rangle \quad \hat{H} e^{\hat{T}} |c\rangle = E_{CC} e^{\hat{T}} |c\rangle. \quad (2.43)$$

One can show that the coupled cluster method gives the same eigenstates as the FCI method when we include all the cluster operators, but doing this would not yield us any advantages in terms of computational complexity.

Hence, it is normal to truncate the cluster operator at some order. In our thesis we will limit ourselves to Coupled Cluster Doubles (CCD), which makes our cluster operator

$$\hat{T} = \hat{T}_2 = \frac{1}{4} \sum_{ab} \sum_{ij} t_{ij}^{ab} a_a^\dagger a_b^\dagger a_j a_i. \quad (2.44)$$

If we left multiply eq. 2.43 with $\langle c| e^{-\hat{T}}$, we get an expression for the energy

$$\langle c| e^{-\hat{T}} \hat{H} e^{\hat{T}} |c\rangle = E_{CC} \langle c| e^{-\hat{T}} e^{\hat{T}} |c\rangle = E_{CC}.$$

We can write this expression in terms of the similarity transformed Hamiltonian

$$\bar{H} = e^{-\hat{T}} \hat{H} e^{\hat{T}}, \quad (2.45)$$

as

$$\langle c| \bar{H} |c\rangle = E_{CC}. \quad (2.46)$$

In the same manner, we can left multiply eq. 2.43 with $\langle \Phi_{ij\dots}^{ab\dots} | e^{-\hat{T}}$ to obtain

$$\langle \Phi_{ij\dots}^{ab\dots} | \bar{H} |c\rangle = 0, \quad (2.47)$$

which can be used to solve for the unknowns $t_{ijk\dots}^{abc\dots}$. To write out eqs. 2.46 and 2.47, we can expand our similarity transformed Hamiltonian using the Baker-Campbell-Hausdorff (BCH) expansion [46]

$$\begin{aligned} \bar{H} &= \hat{H} + [\hat{H}, \hat{T}] \\ &\quad + \frac{1}{2} [[\hat{H}, \hat{T}], \hat{T}] \\ &\quad + \frac{1}{6} [[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}]] \\ &\quad + \frac{1}{24} [[[[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}], \hat{T}], \hat{T}]] \\ &\quad + \dots, \end{aligned} \quad (2.48)$$

where the operator \hat{H} is given by the normal ordered Hamiltonian (eq. 2.37)

$$\begin{aligned} \hat{H} &= E_{\text{ref}} + \hat{F} + \hat{H}_I \\ &= E_{\text{ref}} + \sum_{pq} \langle p| \hat{f} |q\rangle \{a_p^\dagger a_q\} + \frac{1}{4} \sum_{pqrs} \langle pq|\hat{v}|rs\rangle \{a_p^\dagger a_q^\dagger a_s a_r\}. \end{aligned} \quad (2.49)$$

We will now go through how to solve the energy equation (eq. 2.46) and amplitude equation (eq. 2.47). We will do this for the general form of the normal ordered Hamiltonian in eq. 2.49. The solution can then be simplified for the pairing model, as we do not allow the breaking of particle pairs. After having obtained the general solution, the simplification can easily be done by restricting the sums in our solution to not include expectation values with broken pairs.

2.8.1 Energy equation

To calculate the Coupled Cluster energy, we need to solve (eq. 2.46)

$$E = \langle c| \bar{H} |c\rangle, \quad (2.50)$$

where \bar{H} is the similarity transformed Hamiltonian in eq. 2.45. Inserting the Baker-Campbell-Hausdorff (BCH) expansion (eq. 2.48), we get

$$E = \langle c| \hat{H} |c\rangle + \langle c| [\hat{H}, \hat{T}_2] |c\rangle + \frac{1}{2} \langle c| [[\hat{H}, \hat{T}_2], \hat{T}_2] |c\rangle + \text{higher order terms}, \quad (2.51)$$

where the Hamiltonian is given by eq. 2.49. From Wick's generalized theorem (section 2.7), we immediately see that the first term is the reference energy,

$$\langle c | \hat{H} | c \rangle = E_{\text{ref}}. \quad (2.52)$$

Further, the second term in eq. 2.51 can be split up into four terms

$$\langle c | [\hat{H}, \hat{T}_2] | c \rangle = \langle c | \hat{H} \hat{T}_2 | c \rangle - \langle c | \hat{T}_2 \hat{H} | c \rangle + \langle c | \hat{H}_I \hat{T}_2 | c \rangle - \langle c | \hat{T}_2 \hat{H}_I | c \rangle, \quad (2.53)$$

where the two former terms obviously do not contribute due to lack of full contractions. The third term can be calculated using Wicks's theorem:

$$\begin{aligned} \langle c | \hat{H}_I \hat{T}_2 | c \rangle &= \frac{1}{16} \sum_{pqrs} \sum_{abij} t_{ij}^{ab} \langle pq | \hat{v} | rs \rangle \langle c | \{a_p^\dagger a_q^\dagger a_s a_r\} \{a_a^\dagger a_b^\dagger a_j a_i\} | c \rangle \\ &= \frac{1}{16} \sum_{pqrs} \sum_{abij} t_{ij}^{ab} \langle pq | \hat{v} | rs \rangle \left(\delta_{pi} \delta_{qj} \delta_{sb} \delta_{ra} - \delta_{pi} \delta_{qj} \delta_{sa} \delta_{rb} + \delta_{pj} \delta_{qi} \delta_{sa} \delta_{rb} - \delta_{pj} \delta_{qi} \delta_{sb} \delta_{ra} \right) \\ &= \frac{1}{4} \sum_{abij} t_{ij}^{ab} \langle ij | \hat{v} | ab \rangle. \end{aligned} \quad (2.54)$$

Finally, the last term in eq. 2.53 becomes zero since \hat{H}_I is normal ordered. We then have

$$\langle c | [\hat{H}, \hat{T}_2] | c \rangle = \frac{1}{4} \sum_{abij} t_{ij}^{ab} \langle ij | \hat{v} | ab \rangle. \quad (2.55)$$

The last term in the energy expression (eq. 2.51) can be simplified

$$\begin{aligned} \langle c | [[\hat{H}, \hat{T}_2], \hat{T}_2] | c \rangle &= \langle c | (\hat{H} \hat{T}_2 - \hat{T}_2 \hat{H}), \hat{T}_2 | c \rangle \\ &= \langle c | (\hat{H} \hat{T}_2^2 - \hat{T}_2 \hat{H} \hat{T}_2 + \hat{T}_2 \hat{H} \hat{T}_2 - \hat{T}_2^2 \hat{H}) | c \rangle \\ &= \langle c | (\hat{H} \hat{T}_2^2 - \hat{T}_2^2 \hat{H}) | c \rangle. \end{aligned} \quad (2.56)$$

This is zero since our Slater determinants are orthogonal and \hat{T}_2^2 creates 4p-4h states (see section 2.3 for the meaning of 4p-4h), while the Hamiltonian is only able to create two 2p-2h states. Following the same reasoning, we get no contribution for the higher order terms in eq. 2.51. The full CCD energy then reads

$$E_{\text{CCD}} = E_{\text{ref}} + \frac{1}{4} \sum_{abij} t_{ij}^{ab} \langle ij | \hat{v} | ab \rangle. \quad (2.57)$$

2.8.2 Amplitude equation

We now need to solve for t_{ij}^{ab} in order to find the CCD energy in eq. 2.57. This can be done by first finding an algebraic expression for the amplitude equation (eq. 2.47). By using the BCH expansion (eq. 2.48), we get

$$\begin{aligned} \langle \Phi_{ij}^{ab} | \bar{H} | c \rangle &= \langle \Phi_{ij}^{ab} | \hat{H} | c \rangle \\ &+ \langle \Phi_{ij}^{ab} | [\hat{H}, \hat{T}_2] | c \rangle \\ &+ \frac{1}{2} \langle \Phi_{ij}^{ab} | [[\hat{H}, \hat{T}_2], \hat{T}_2] | c \rangle \\ &+ \frac{1}{6} \langle \Phi_{ij}^{ab} | [[[\hat{H}, \hat{T}_2], \hat{T}_2], \hat{T}_2] | c \rangle \\ &+ \dots, \end{aligned} \quad (2.58)$$

where \hat{H} is still given by eq. 2.49. The first expression to the right of the equal sign can be broken into three parts. The first part is

$$\langle \Phi_{ij}^{ab} | E_{\text{ref}} | c \rangle = 0,$$

because of the orthogonality of our SDs. Next up is

$$\langle \Phi_{ij}^{ab} | \hat{F} | c \rangle = 0,$$

for the same reason (\hat{F} is only able to create 1p-1h on the reference vacuum). This is known as the Slater-Condon rule [12]. The final part can be simplified using Wicks generalized theorem (section 2.7):

$$\begin{aligned} \langle \Phi_{ij}^{ab} | \hat{H}_I | c \rangle &= \frac{1}{4} \sum_{pqrs} \langle pq | \hat{v} | rs \rangle \langle c | a_i^\dagger a_j^\dagger a_b a_a \{ a_p^\dagger a_q^\dagger a_s a_r \} | c \rangle \\ &= \frac{1}{4} \sum_{pqrs} \langle pq | \hat{v} | rs \rangle (\delta_{ir} \delta_{js} \delta_{ap} \delta_{bq} - \delta_{si} \delta_{jr} \delta_{bq} \delta_{ap} + \\ &\quad \delta_{si} \delta_{jr} \delta_{bp} \delta_{aq} - \delta_{ir} \delta_{js} \delta_{bp} \delta_{aq}) \\ &= \langle ab | \hat{v} | ij \rangle. \end{aligned}$$

The second term in eq. 2.58 can be split into six parts,

$$\begin{aligned} \langle \Phi_{ij}^{ab} | [\hat{H}, \hat{T}_2] | c \rangle &= \langle \Phi_{ij}^{ab} | E_{ref} \hat{T}_2 | c \rangle - \langle \Phi_{ij}^{ab} | \hat{T}_2 E_{ref} | c \rangle \\ &\quad + \langle \Phi_{ij}^{ab} | \hat{F} \hat{T}_2 | c \rangle - \langle \Phi_{ij}^{ab} | \hat{T}_2 \hat{F} | c \rangle \\ &\quad + \langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2 | c \rangle - \langle \Phi_{ij}^{ab} | \hat{T}_2 \hat{H}_I | c \rangle. \end{aligned} \quad (2.59)$$

The first two terms obviously cancel out as E_{ref} is a constant. The third term can again be calculated using Wick's generalized theorem. The result is

$$\langle \Phi_{ij}^{ab} | \hat{F} \hat{T}_2 | c \rangle = \sum_k f_i^k t_{jk}^{ab} \hat{P}(ij) - \sum_c f_c^a t_{ij}^{bc} \hat{P}(ab), \quad (2.60)$$

which is shown in Appendix A. The operator $\hat{P}(pq)$ gives the current term minus a similar term with indices p and q switched

$$\hat{P}(pq) = 1 - \hat{P}_{pq}, \quad (2.61)$$

where \hat{P}_{pq} permutes the indices p and q . The fourth term of eq. 2.59 will be zero since we have the normal ordered term of the Hamiltonian acting on the reference state.

Now to the second-to-last term of eq. 2.59

$$\langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2 | c \rangle = \frac{1}{16} \sum_{cd} \sum_{kl} t_{kl}^{cd} \sum_{pqrs} \langle c | a_i^\dagger a_j^\dagger a_b a_a \{ a_p^\dagger a_q^\dagger a_s a_r \} a_c^\dagger a_d^\dagger a_l a_k | c \rangle. \quad (2.62)$$

By using Wicks generalized theorem we see that this can be written as

$$\langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2 | c \rangle = \frac{1}{2} \sum_{cd} t_{ij}^{cd} \langle ab | \hat{v} | cd \rangle + \frac{1}{2} \sum_{kl} t_{kl}^{ab} \langle kl | \hat{v} | ij \rangle + \hat{P}(ab) \hat{P}(ij) \sum_{kc} t_{jk}^{bc} \langle ak | \hat{v} | ic \rangle, \quad (2.63)$$

because of the anti-symmetry of t_{kl}^{cd} and the anti-symmetry of the matrix elements. The derivation is given in appendix A. The final term of eq. 2.59 will be zero since we have the normal ordered terms of the Hamiltonian acting on the reference state.

For the third term in eq. 2.58 we proceed in the same manner. First we have

$$\langle \Phi_{ij}^{ab} | [[E_{ref}, \hat{T}_2], \hat{T}_2] | c \rangle = 0,$$

since E_{ref} is a constant. Next we have

$$\langle \Phi_{ij}^{ab} | [[\hat{F}, \hat{T}_2], \hat{T}_2] | c \rangle = \langle \Phi_{ij}^{ab} | (\hat{F} \hat{T}_2^2 - \hat{T}_2^2 \hat{F}) | c \rangle = 0,$$

since \hat{T}_2^2 creates 4p-4h states while \hat{F} is a one-body operator. Finally we have

$$\langle \Phi_{ij}^{ab} | [[\hat{H}_I, \hat{T}_2], \hat{T}_2] | c \rangle = \langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2^2 | c \rangle - \langle \Phi_{ij}^{ab} | \hat{T}_2^2 \hat{H}_I | c \rangle,$$

where the final term becomes zero since \hat{H}_I is normal ordered. We then have

$$\begin{aligned} \frac{1}{2} \langle \Phi_{ij}^{ab} | [[\hat{H}_I, \hat{T}_2], \hat{T}_2] | c \rangle &= \frac{1}{2} \langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2^2 | c \rangle = \frac{1}{4} \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{kl}^{ab} t_{ij}^{cd} \\ &\quad - \frac{1}{2} \hat{P}(ij) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{il}^{ab} t_{kj}^{cd} \\ &\quad - \hat{P}(ab) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{ik}^{ac} t_{lj}^{bd} \\ &\quad - \frac{1}{2} \hat{P}(ab) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{ij}^{ac} t_{kl}^{bd}. \end{aligned}$$

All the subsequent terms of eq. 2.58 will be zero. This can be explained by the fact that the n 'th term will contain \hat{T}_2^{n-1} , which for $n \geq 4$ will result in zero due to the Slater-Condon rules [10].

Collecting all the terms finally gives

$$\begin{aligned} \langle \Phi_{ij}^{ab} | \bar{H} | c \rangle &= \langle ab | \hat{v} | ij \rangle + \hat{P}(ij) \sum_k f_i^k t_{jk}^{ab} - \hat{P}(ab) \sum_c f_c^a t_{ij}^{bc} \\ &\quad + \frac{1}{2} \sum_{kl} \langle kl | \hat{v} | ij \rangle t_{kl}^{ab} + \hat{P}(ab) \hat{P}(ij) \sum_{kc} \langle ak | \hat{v} | ic \rangle t_{jk}^{bc} \\ &\quad + \frac{1}{2} \sum_{cd} \langle ab | \hat{v} | cd \rangle t_{ij}^{cd} + \frac{1}{4} \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{kl}^{ab} t_{ij}^{cd} - \frac{1}{2} \hat{P}(ij) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{il}^{ab} t_{kj}^{cd} \\ &\quad - \hat{P}(ab) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{ik}^{ac} t_{lj}^{bd} - \frac{1}{2} \hat{P}(ab) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{ij}^{ac} t_{kl}^{bd} = 0. \end{aligned}$$

By observing that

$$\hat{P}(ij) \sum_k f_i^k t_{jk}^{ab} = \hat{P}(ij) f_i^i t_{ji}^{ab} + \hat{P}(ij) \sum_{k \neq i} f_i^k t_{jk}^{ab} = -f_i^i t_{ij}^{ab} - f_j^j t_{ij}^{ab} + \hat{P}(ij) \sum_{k \neq i} f_i^k t_{jk}^{ab},$$

and that

$$\hat{P}(ab) \sum_c f_c^a t_{ij}^{bc} = \hat{P}(ab) f_a^a t_{ij}^{ba} + \hat{P}(ab) \sum_{c \neq a} f_c^a t_{ij}^{bc} = -f_a^a t_{ij}^{ab} - f_b^b t_{ij}^{ab} + \hat{P}(ab) \sum_{c \neq a} f_c^a t_{ij}^{bc},$$

we can rewrite the above equation as

$$\begin{aligned} t_{ij}^{ab} D_{ab}^{ij} &= \langle ab | \hat{v} | ij \rangle + \hat{P}(ij) \sum_{k \neq i} f_i^k t_{jk}^{ab} - \hat{P}(ab) \sum_{c \neq a} f_c^a t_{ij}^{bc} \\ &\quad + \frac{1}{2} \sum_{kl} \langle kl | \hat{v} | ij \rangle t_{kl}^{ab} + \hat{P}(ab) \hat{P}(ij) \sum_{kc} \langle ak | \hat{v} | ic \rangle t_{jk}^{bc} \\ &\quad + \frac{1}{2} \sum_{cd} \langle ab | \hat{v} | cd \rangle t_{ij}^{cd} + \frac{1}{4} \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{kl}^{ab} t_{ij}^{cd} - \frac{1}{2} \hat{P}(ij) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{il}^{ab} t_{kj}^{cd} \\ &\quad - \hat{P}(ab) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{ik}^{ac} t_{lj}^{bd} - \frac{1}{2} \hat{P}(ab) \sum_{klcd} \langle kl | \hat{v} | cd \rangle t_{ij}^{ac} t_{kl}^{bd}, \end{aligned} \tag{2.64}$$

with $D_{ab}^{ij} = f_i^i + f_j^j - f_a^a - f_b^b$. By dividing the above equation with D_{ab}^{ij} , we can find t_{ij}^{ab} by making an initial guess on the cluster amplitudes and then update them iteratively.

Note that with the FCI method (section 2.5), we had to generate a $\binom{K}{n}$ by $\binom{K}{n}$ matrix for K spin-orbitals and n particles. This task is exponential in the number of spin-orbitals. Iteratively updating one of the $n^2(K-n)^2$ CCD amplitudes requires us to calculate sums consisting of at most $n^2(K-n)^2$ terms. We hence have a computationally efficient method to approximate the ground state energy. However, the CCD method doesn't

2. Many-Body Methods in Quantum Mechanics

necessarily yield us an exact solution within the subspace spanned by our spin-orbitals. One will have to make a trade-off between computational complexity and accuracy when choosing between the two methods. This explains a key motivation for studying many-body methods in quantum mechanics for quantum computers, as one wishes to find methods that gives us the best of both worlds in terms of accuracy and computational complexity.

CHAPTER 3

Supervised Learning

The world's technological advances have lead to an increasing amount of data being collected every second. The variables involved in a single data point in these data sets can range from just a couple to thousands, making analysis with visualization tools a cumbersome and difficult task. Machine learning is a field where one employs algorithms which can identify patterns and learn from data without much human interaction. Machine learning have also gained attraction in the field of many body quantum physics [14], as some machine learning models may have the potential to solve problems that are infeasible with classical methods. We will focus on supervised learning in this thesis. To understand what the task of supervised learning is, suppose we have a data set containing n measurements of p features $x_1^{(i)}, x_2^{(i)}, \dots, x_p^{(i)}$ as well as a measurement of the target variable $y^{(i)}$. The index i denotes the i 'th of the n measurements. In short, we want to utilize the features to predict the target variable, and to do this, we need to generate a data set. Let us as an example say that you are interested in estimating the yearly income of an individual based on their age, years of experience and years of education. The target variable y in this case will be the individuals yearly income and the features x_1, x_2 and x_3 will be their age, years of experience and years of education, respectively. In order to create our data set, we contact five random individuals and gather their yearly income, age, years of experience and years of education. The data set may look something like this

y (10 ³ kr)	x_1 (years)	x_2 (years)	x_3 (years)
350	19	0	0
650	30	3	5
500	25	0	5
470	25	1	5
420	23	1	3

Keeping this example in mind, supervised learning is the task of predicting the yearly income of a new individual by training a model on the previously gathered data. One of the simplest machine learning methods to do so is called linear regression, which will be covered next. Keep in mind that even though we do not use linear regression, nor logistic regression (which will also be covered soon) as methods in this thesis, the procedures learned for these models are frequently used in neural networks.

3.1 Linear Regression

First of, linear regression assumes a linear relationship between our target variable y and the features x_1, \dots, x_p . Mathematically, this can be written as

$$y^{(i)} = \beta_0 + x_1^{(i)}\beta_1 + x_2^{(i)}\beta_2 + \dots + x_p^{(i)}\beta_p + \epsilon^{(i)} = f(x^{(i)}) + \epsilon^{(i)}, \quad (3.1)$$

where β_0 is the intercept and β_j is the increase of y with a unit increase of x_j . We assume that there is some noise in context with the measurement, which is assumed to be normally distributed with the unknown variance σ^2 . That is $\epsilon \sim N(0, \sigma^2)$. A common way to solve for parameters in a supervised learning problem is to minimize what is called a loss function. When defining a loss function, we can look for a function whose value decreases

3. Supervised Learning

when the predicted value is close to the real target value. For linear regression, we use a loss function which is commonly referred to as the mean squared error (MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{f}(x^{(i)}; \hat{\beta}))^2, \quad (3.2)$$

where

$$\hat{f}(x^{(i)}; \hat{\beta}) = \hat{\beta}_0 + x_1^{(i)} \hat{\beta}_1 + x_2^{(i)} \hat{\beta}_2 + \cdots + x_p^{(i)} \hat{\beta}_p.$$

Our goal now is to vary the parameters $\hat{\beta}$ till the MSE is as close to zero as possible. For linear regression, we have an analytical solution for this. To calculate the minimum of the MSE, it is convenient to write the above equation in matrix notation. First of, we can write the sum $\beta_0 + x_1^{(i)} \beta_1 + x_2^{(i)} \beta_2 + \cdots + x_p^{(i)} \beta_p$ for all i as a matrix-vector product. We define the matrix

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_p^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_p^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \cdots & x_p^{(n)} \end{bmatrix},$$

which we will refer to as the design matrix. We also set

$$\hat{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

The MSE can then be rewritten as

$$MSE = \frac{1}{n} (\mathbf{y} - X\hat{\beta})^T (\mathbf{y} - X\hat{\beta}).$$

The optimal parameters can then be found by differentiating the MSE with respect to $\hat{\beta}$ and setting it to zero. The optimal parameters are given by [18]

$$\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}. \quad (3.3)$$

3.2 Logistic Regression

Linear regression is suited for prediction of a continuous variable, but what if we want to predict a discrete variable? For example, we want to predict if an individual is making 500.000 kr or more based on their age, years of experience and years of education. The previous data set would now look like this;

y (binary)	x_1 (years)	x_2 (years)	x_3 (years)
0	19	0	0
1	30	3	5
1	25	0	5
0	25	1	5
0	23	1	3

where $y = 0$ means that the individual makes less than 500.000 kr and $y = 1$ means that they make 500.000 kr or more. One could treat this as a linear regression problem and say that if the model predicts ≥ 0.5 , we have $y = 1$, else we have $y = 0$. However, the assumption that the relationship between the target and feature is linear as presented in equation 3.1 is simply not correct in this setting. We can instead try to predict the

probability of making 500.000 kr or more given the features in question. To do this, we first need to find a function which maps each input to a value between 0 and 1. Consider the Sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \quad (3.4)$$

As x approaches positive infinity, the Sigmoid function outputs a value of 1, while it outputs a value of 0 when x approaches negative infinity. To see how this helps us, consider using the linear combination

$$\sum_{j=0}^p \beta_j x_j^{(i)} = \beta_0 + x_1^{(i)} \beta_1 + x_2^{(i)} \beta_2 + \cdots + x_p^{(i)} \beta_p,$$

where $x_0^{(i)} = 1$, as inputs to the Sigmoid function. What we have now is a parametrized function of our features which gives an output between 1 and 0. We can treat this output as a probability. The probability of $y = 1$ is given by

$$P(y^{(i)} = 1 | \mathbf{x}^{(i)}; \boldsymbol{\beta}) = \frac{1}{1 + e^{-\sum_{j=0}^p \beta_j x_j^{(i)}}}. \quad (3.5)$$

Since we are dealing with a binary problem and probabilities sum to one, we can write the probability of $y = 0$ as

$$P(y^{(i)} = 0 | \mathbf{x}^{(i)}; \boldsymbol{\beta}) = 1 - P(y^{(i)} = 1 | \mathbf{x}^{(i)}; \boldsymbol{\beta}). \quad (3.6)$$

Now that we have explained how a logistic regression model outputs probabilities, we need to go through how to train the model. Like with linear regression, we need a loss function whose value is at its lowest when all predicted points coincide with the target values. That is where maximum likelihood comes into play. Maximum likelihood is a function which tells us the likeliness of our chosen parameters (β_j) given our data (y and x). The function outputs a large value when our predicted probability is close to one (zero) and the actual target value is one (zero). Maximum likelihood can be written as [18]

$$\begin{aligned} L(\hat{\boldsymbol{\beta}}) &= \frac{1}{n} \prod_{i=1}^n P(y^{(i)} = 1 | \mathbf{x}^{(i)}; \hat{\boldsymbol{\beta}})^{y^{(i)}} (1 - P(y^{(i)} = 1 | \mathbf{x}^{(i)}; \hat{\boldsymbol{\beta}}))^{1-y^{(i)}} \\ &= \frac{1}{n} \prod_{i=1}^n (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}, \end{aligned} \quad (3.7)$$

where $\hat{y}^{(i)}$ denotes the predicted probability of the i 'th sample. To train the logistic regression model, we can maximize this function with respect to the model parameters $\hat{\boldsymbol{\beta}}$. However, one usually minimizes its negative logarithm for convenience

$$L_{NLL}(\hat{\boldsymbol{\beta}}) = -\log(L(\hat{\boldsymbol{\beta}})) = -\sum_{i=1}^n y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}). \quad (3.8)$$

This is known as the Negative Log Likelihood loss function (NLL) and its minima has no known analytical solution. To be able to still find a minima, we will now introduce a numerical method called Gradient descent.

3.3 Gradient Descent

Gradient descent is a well known minimization method which is widely used to train supervised models. When dealing with a function $f(\hat{\boldsymbol{\beta}})$, where $\hat{\boldsymbol{\beta}}$ is a vector of parameters, one can find the direction of the largest function increase by calculating the gradient $\nabla_{\hat{\boldsymbol{\beta}}} f$. Likewise, the gradient of $-f$ will point towards the direction of the largest function decrease. Hence, one can find the direction of a minima of a function by calculating $\nabla_{\hat{\boldsymbol{\beta}}}(-f) = -\nabla_{\hat{\boldsymbol{\beta}}} f$, and then update the parameters $\hat{\boldsymbol{\beta}}$ the following way

$$\hat{\boldsymbol{\beta}}^{(j+1)} = \hat{\boldsymbol{\beta}}^{(j)} - \lambda \nabla_{\hat{\boldsymbol{\beta}}^{(j)}} f. \quad (3.9)$$

Since $-\nabla_{\hat{\boldsymbol{\beta}}} f$ is merely a direction, it does not give away how large of a step one would have to make in parameter space to reach a minima. The direction may also change when traversing parameter space. Hence, we introduce

3. Supervised Learning

λ , which is often referred to as the learning rate. The learning rate decides how large of a step to take in the direction of the gradient, and is usually found trial and error.

The parameters of a machine learning model can likewise be updated by calculating the gradient of a loss function with respect to the model parameters. The gradient is then used in eq. 3.9.

3.4 Dense Neural Network

Neural networks are machine learning models that are meant to mimic the way neurons in human brains communicate with each other by sending electric signals. A neuron is activated if the signals it receives exceed an activation threshold. This neuron will then yield an output that may or may not activate other neurons.

One of the simplest neural networks consists of what are called dense layers. Hence, we will start by explaining how to set these up. If our input $X \in \mathcal{R}^{1 \times p}$ is a matrix containing one data sample with p features, the first step of our neural network is to calculate a weighted sum

$$\mathbf{z}^1 = W^1 X^T + \mathbf{b}^1,$$

where

$$W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & \cdots & w_{1p}^1 \\ w_{21}^1 & w_{22}^1 & \cdots & w_{2p}^1 \\ \vdots & \vdots & \vdots & \vdots \\ w_{m1}^1 & w_{m2}^1 & \cdots & w_{mp}^1 \end{bmatrix},$$

and

$$\mathbf{b}^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \\ \vdots \\ b_m^1 \end{bmatrix}.$$

The bias, \mathbf{b}^1 , is introduced to make sure that neurons could activate when all the inputs are zero. We can see that the row-dimensionality of W^1 decides the dimensions of \mathbf{z}^1 . Hence, m rows gives m outputs, called neurons, whose value is contained in \mathbf{z}^1 . Since the neurons are simply a linear combination of the inputs, we are only able to represent linear relationships. Hence, we will use what is referred to as an activation function $f^1(x)$ to introduce non-linearity. The mathematical realization of an input layer for a neural network is then given by

$$\mathbf{a}^1 = f^1(\mathbf{z}^1) = f^1(W^1 X^T + \mathbf{b}^1),$$

where the elements of \mathbf{a}^1 are referred to as activations. Stacking of layers are what gives neural networks expressive power. This can be done by following the same procedures as we just did, but using the previously calculated activations as inputs. Hence, the activations of layer l are given by [18]

$$\mathbf{a}^l = f^l(W^l \mathbf{a}^{l-1} + \mathbf{b}^l), \quad (3.10)$$

or without matrix/vector notation

$$a_i^l = f^l\left(\sum_j W_{ij}^l a_j^{l-1} + b_i^l\right). \quad (3.11)$$

The dimensions of the rows of W and the dimension of \mathbf{b} are arbitrary and specifies how many nodes one wants in intermediate layers. However, for the final layer they are problem-specific, meaning that a problem with a one-dimensional target variable should only have one node in the final layer.

3.4.1 Backpropagation algorithm

The backpropagation algorithm [17] is a common way to train the parameters of a neural network. Here we will go through how this algorithm works. Suppose we have defined a loss function, $L(\mathbf{a}^L, \mathbf{y})$, where $\mathbf{a}^L = f^L(W^L \mathbf{a}^{L-1} + \mathbf{b}^L)$ specifies the activation(s) in the final/output layer. Backpropagation utilizes the chain rule in order to find the gradient of our loss function with respect to the neural network parameters. We can

then utilize gradient descent (section 3.3) to update the parameters. Consider calculating the gradients with respect to the weights in the output layer

$$\frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial W_{ij}^L}.$$

Utilizing the chain rule, we get

$$\frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial W_{ij}^L} = \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_i^L} \frac{\partial a_i^L}{\partial W_{ij}^L} = \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial W_{ij}^L}.$$

Similarly for the bias:

$$\frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial b_i^L} = \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial b_i^L}.$$

Going for the weights in the previous layer, we get

$$\begin{aligned} \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial W_{ij}^{L-1}} &= \sum_k \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_k^L} \frac{\partial a_k^L}{\partial W_{ij}^{L-1}} \\ &= \sum_k \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L} \frac{\partial z_k^L}{\partial a_i^{L-1}} \frac{\partial a_i^{L-1}}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial W_{ij}^{L-1}}, \end{aligned}$$

and

$$\begin{aligned} \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial b_i^{L-1}} &= \sum_k \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_k^L} \frac{\partial a_k^L}{\partial b_i^{L-1}} \\ &= \sum_k \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L} \frac{\partial z_k^L}{\partial a_i^{L-1}} \frac{\partial a_i^{L-1}}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial b_i^{L-1}}. \end{aligned}$$

By denoting [43]

$$\delta_k^l = \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_k^l} = \sum_j \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_j^{l+1}} \frac{\partial a_j^{l+1}}{\partial a_k^l} = \sum_j \delta_j^{l+1} \frac{\partial a_j^{l+1}}{\partial a_k^l}, \quad (3.12)$$

we see that the partial derivatives of weights in an arbitrary layer are given by

$$\frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial W_{ij}^l} = \delta_i^l \frac{\partial a_i^l}{\partial W_{ij}^l}. \quad (3.13)$$

Similarly for the biases

$$\frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial b_i^l} = \delta_i^l \frac{\partial a_i^l}{\partial b_i^l}. \quad (3.14)$$

The backpropagation algorithm goes as follows:

1. Calculate $\delta_k^L = \frac{\partial L(\mathbf{a}^L, \mathbf{y})}{\partial a_k^L}$
2. Use equation 3.12 to recursively calculate δ_k^l for all l .
3. Find the gradient with respect to all the weights and biases with equation 3.13 and 3.14 respectively.
4. Utilize gradient descent (section 3.3) to update the weights.

As we will not be using analytic gradient methods to update our quantum neural networks, we will not go into further detail on the partial derivatives as these are model specific.

3.5 Recurrent Neural Network Layer

The dense neural network is a good choice when dealing with data where each point in the data set can be treated individually. When this is not the case, as for example with time series data, we do not have any way to tell the network explicitly where in the time series each point in the data sample belongs. That is not to say that it can not be learnt by a dense neural network, but it may be wise to use an architecture that knows this explicitly so its expressive power can be used solely to learn other relationships between the predictors. A recurrent neural network layer is an excellent choice for time series data and a node in such a layer can be expressed mathematically as [21]

$$h_i^t = f\left(\sum_j w_{ji}^x x_j^t + b_i^x + \sum_j w_{ji}^h h_j^{t-1} + b_i^h\right). \quad (3.15)$$

Here f is the activation function, w_{ji}^x are the weights to be multiplied with the input data x_j^t and w_{ji}^h are the weights to be multiplied with the hidden layer vector elements h_j^{t-1} . b_i^x and b_i^h are the input and hidden biases respectively. The input data consists of an arbitrary number of time steps $\mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3, \dots$ where \mathbf{x}^t consists of an arbitrary number of predictors x_1^t, x_2^t, \dots . For example, the input vector \mathbf{x}^t could be the fuel level, speed and rpm of a car at time step t . We initialize the hidden layer vector $\mathbf{h}^0 \in \mathcal{R}^k$ with for example random numbers. Feeding in our data to the layer in eq. 3.15 gives us the hidden layer vector for the next time step:

$$h_i^1 = f\left(\sum_j w_{ji}^x x_j^1 + b_i^x + \sum_j w_{ji}^h h_j^0 + b_i^h\right).$$

With \mathbf{h}^1 calculated, we can again proceed to the next time step

$$h_i^2 = f\left(\sum_j w_{ji}^x x_j^2 + b_i^x + \sum_j w_{ji}^h h_j^1 + b_i^h\right),$$

and we continue this way til we have calculated \mathbf{h}^T for the final time step T in our input data. We have now obtained a set $\{\mathbf{h}^0, \mathbf{h}^1, \dots, \mathbf{h}^T\}$ that can be used to generate an output. How to do this is problem specific, but if for example our target variable is a two dimensional vector, one could feed the final hidden vector \mathbf{h}^T as the input to a dense layer with two nodes.

PART II

Quantum Computing: Machine Learning and Many-Body Methods

CHAPTER 4

Quantum Computing: Many-Body Methods

Classical many-body methods in quantum mechanics often trade computational complexity for accuracy when solving for the ground state energy of a system. The recent advances in quantum computing have shown promise in approximating the ground state energy without such a large trade-off [34]. Hence, we will look into a couple of promising methods in this section. Before we start learning about how quantum computing works, we will give a short introduction to the building blocks of classical computers. In classical computing, the basic unit of information is called a bit. The bit is represented by a binary digit, either a 1 or a 0. A collection of bits provides a binary string with information, and this information can be manipulated with what we call logic gates. Logic gates are operations on one or more bits and produces a single output, 1 or 0. A collection of such logic gates are called a circuit and an example of such a circuit could be outputting a 1 if all input bits are put to 1 and output 0 otherwise. The behaviour of all circuits on classical computers are deterministic in the sense that a given input binary string will always produce the same output.

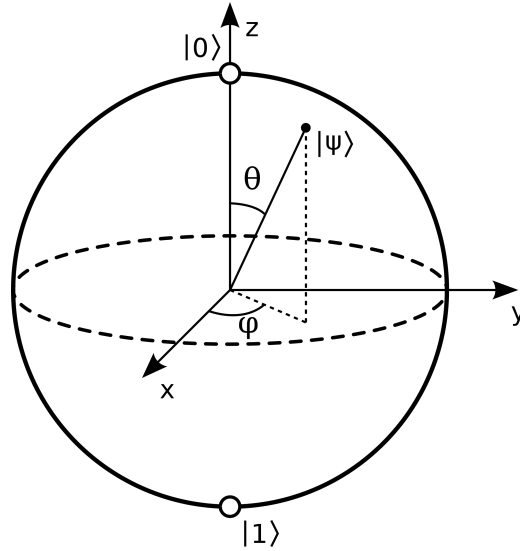


Figure 4.1: The Bloch sphere. Figure was made by Smite Meister [31]

How quantum computing differs from classical computing can partly be illustrated by looking at the surface of the unit sphere in figure 4.1, which is commonly referred to as the Bloch sphere. Whereas a classical bit must either be in the $|0\rangle$ state at the surface on the north pole, or in the $|1\rangle$ state at the surface on the south pole; the quantum bit, namely a qubit, can be in a state located anywhere on the surface of the Bloch sphere. The qubit state $|\psi\rangle$ can be in what we know from quantum mechanics as a superposition,

$$|\psi\rangle = c_0 |0\rangle + c_1 |1\rangle,$$

where a measurement of the qubit will collapse the state, resulting in the state $|0\rangle$ or $|1\rangle$ with probability $|c_0|^2$ or $|c_1|^2$ respectively. These coefficients are referred to as amplitudes and could be any complex number as long as

the measurement probabilities are normalized, that is $|c_0|^2 + |c_1|^2 = 1$. The orientation on the x, y plane of the surface is given by the complex phases. There are also other quantum mechanical properties than superposition that apply to the qubits, namely entanglement. Just like particles can become entangled, several qubits can become entangled, meaning that the measurement outcome of one qubit can directly affect another. How the properties of superposition and entanglement have the potential to be used for our advantage will be shown during this chapter. But first, we will begin by introducing the basis states we are dealing with in quantum computing.

4.1 Introduction to Quantum Computing

Our introduction to quantum computing is mostly based on the book *Quantum Computation and Quantum Information* by Michael A. Nielsen Isaac L. Chuang [37]. It is an excellent read and we highly recommend it if new to quantum computing.

4.1.1 Basis

In order to start understanding how to manipulate one or several qubits to our advantage, we need to specify the basis we are working in. This is important as the manipulation of a qubit can mathematically be represented by matrix-vector multiplications. As we have talked about earlier, the measurement of a qubit will result in either the $|0\rangle$ or the $|1\rangle$ state, and it is these states which forms the computational basis for quantum computing. We can write them in vector form as

$$|0\rangle \equiv \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{and} \quad |1\rangle \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad (4.1)$$

and it is important to note that any linear combination of these states is an allowed state of a qubit, as long as the unit norm is preserved. The preservation of unit norm serves as a clue as to what sort of operations we are allowed to perform on a qubit. Consider U to be a unitary matrix, that is

$$U^\dagger U = U U^\dagger = I,$$

where I is the identity matrix. Now consider an arbitrary qubit state $|\psi\rangle$ with unit norm, that is $\langle\psi|\psi\rangle = 1$. The action of U on this state gives

$$U|\psi\rangle = |\phi\rangle.$$

The norm of the transformed state is then

$$\langle\phi|\phi\rangle = \langle\psi| \underbrace{U^\dagger U}_{=I} |\psi\rangle = \langle\psi|\psi\rangle = 1.$$

Hence, any unitary transformation preserves the norm of the qubit state. This is no coincidence as the second postulate of quantum mechanics states that the evolution of a closed quantum system is described by a unitary transformation [37]. Unitary transformations are the quantum equivalent of classical logic gates, and hence will often be referred to as gates throughout this thesis. A system of multiple qubits are represented by tensor products. For example, for an n -qubit state we may write

$$|\psi_1\rangle |\psi_2\rangle \cdots |\psi_n\rangle \equiv |\psi_1 \psi_2 \cdots \psi_n\rangle \equiv |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle. \quad (4.2)$$

We may also split the qubits into two or more sub-systems, which we will call registers, as this could be convenient when explaining the algorithms. An example is the state

$$|p\rangle |q\rangle,$$

where $|p\rangle$ and $|q\rangle$ are m and n -qubit states, respectively. We can represent manipulations on multi-qubit states by a tensor product of unitary operators;

$$\begin{aligned} (A \otimes B \otimes \cdots \otimes N) |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle &= A |\psi_1\rangle \otimes B |\psi_2\rangle \otimes \cdots \otimes N |\psi_n\rangle, \\ \text{or} \\ A^1 B^2 \cdots N^n |\psi_1\rangle \otimes |\psi_2\rangle \cdots \otimes |\psi_n\rangle &= A^1 |\psi_1\rangle B^2 |\psi_2\rangle \cdots N^n |\psi_n\rangle, \end{aligned} \quad (4.3)$$

where the superscript denotes which qubit the operator acts on. We will now go through some of the most common gates we deal with on a quantum computer.

4.1.2 Quantum Gates

The first set of gates introduced are what we call single qubit gates. Single qubit gates, as you may have guessed from the name, are gates that only act on a single qubit. Since a qubit is represented by a two-dimensional vector, the single qubit gates can be represented by any two-by-two unitary matrix. Even though any such matrices transforms our qubit into a valid state, there are some notable gates that are important for many of the applications we will consider. First up we have

$$\begin{aligned} X = \sigma_x &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \equiv \text{---} \boxed{X} \text{---} \\ Y = \sigma_y &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \equiv \text{---} \boxed{Y} \text{---} \\ Z = \sigma_z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \equiv \text{---} \boxed{Z} \text{---} , \end{aligned} \quad (4.4)$$

which are referred to as the Pauli- X , Pauli- Y and Pauli- Z gates, or just X , Y and Z gates. We have also included the circuit notation for the gates on the far right. We will go through more details on this in the next section. The X gate, for example, flips the qubit. That is

$$\sigma_x |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \quad \sigma_x |1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle .$$

We also have the Hadamard gate;

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \equiv \text{---} \boxed{H} \text{---} , \quad (4.5)$$

which creates a superposition:

$$\begin{aligned} H |0\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ H |1\rangle &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) . \end{aligned}$$

We can also rotate the qubit an arbitrary angle θ about the x , y or z axis on the Bloch sphere (figure 4.1). The gates that allow us to do this are called rotation gates, and they are subfixed with the rotation axis:

$$\begin{aligned} R_x(\theta) &= e^{-i\theta\sigma_x/2} = \cos\left(\frac{\theta}{2}\right)I - i\sin\left(\frac{\theta}{2}\right)\sigma_x = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \equiv \text{---} \boxed{R_x(\theta)} \text{---} \\ R_y(\theta) &= e^{-i\theta\sigma_y/2} = \cos\left(\frac{\theta}{2}\right)I - i\sin\left(\frac{\theta}{2}\right)\sigma_y = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \equiv \text{---} \boxed{R_y(\theta)} \text{---} \\ R_z(\theta) &= e^{-i\theta\sigma_z/2} = \cos\left(\frac{\theta}{2}\right)I - i\sin\left(\frac{\theta}{2}\right)\sigma_z = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \equiv \text{---} \boxed{R_z(\theta)} \text{---} . \end{aligned} \quad (4.6)$$

We also have a gate that applies a phase of $-i$ to the $|1\rangle$ state. Its called the phase shift gate and is given by

$$S \equiv \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix} \equiv \text{---} \boxed{S} \text{---} . \quad (4.7)$$

A gate that will become useful when we get into the quantum Fourier transform is the R_k gate:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \equiv \text{---} \boxed{R_k} \text{---} . \quad (4.8)$$

4. Quantum Computing: Many-Body Methods

The first two-qubit gate we will introduce is the CNOT gate. Its matrix representation is given by

$$CNOT \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \equiv \begin{array}{c} \text{---} \bullet \text{---} \\ | \\ \text{---} \oplus \text{---} \end{array} . \quad (4.9)$$

Operating on two qubits $|c\rangle|t\rangle$, its task is to flip the target qubit $|t\rangle$ if the control qubit $|c\rangle$ is in the $|1\rangle$ -state. If it is in the $|0\rangle$ -state it should act as the identity operator. This kind of gate is called a controlled gate. In the case of the CNOT gate we perform a Pauli- X gate (eq. 4.4) on the target qubit conditioned on the control qubit, but we can of course apply any of the other gates mentioned. Mathematically we will write down such a gate as

$$X_t^c |c\rangle|t\rangle ,$$

where the superfix denotes the control qubit, while the subfix denotes the target qubit. The final gate we will mention is the Swap-operation. Given the n -qubit state

$$|\psi_1\rangle|\psi_2\rangle\cdots|\psi_i\rangle\cdots|\psi_j\rangle\cdots|\psi_n\rangle ,$$

the swap gate applied to qubit i and j simply produces the state

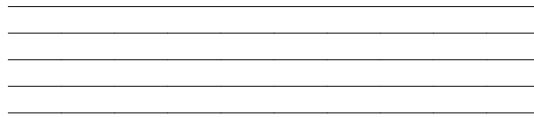
$$\text{SWAP}_i^j |\psi_1\rangle|\psi_2\rangle\cdots|\psi_i\rangle\cdots|\psi_j\rangle\cdots|\psi_n\rangle = |\psi_1\rangle|\psi_2\rangle\cdots|\psi_j\rangle\cdots|\psi_i\rangle\cdots|\psi_n\rangle \quad (4.10)$$

Hence, it puts the j 'th qubit in the i 'th qubit state and vice versa.

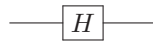
$$\text{SWAP} \equiv \begin{array}{c} \text{---} \times \text{---} \\ | \\ \text{---} \times \text{---} \end{array}$$

4.1.3 Quantum Circuits

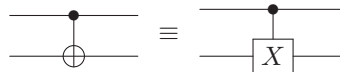
A convenient way to write a quantum algorithm is through quantum circuits. A quantum circuit consists of wires, where each wire represents a qubit:



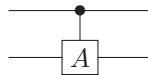
These wires are not physical wires, but you can think of them as representing the passage of time. When reading a quantum circuit, the leftmost operations are performed first, so the circuits are read from left to right. All the qubits are usually initialized in the $|0\rangle$ state unless else is specified. To illustrate that an operation is performed on a qubit, we draw a gate on the wire corresponding to this qubit:



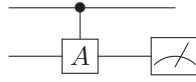
In the above circuit, we only have one qubit which is put in superposition by applying the Hadamard gate (eq. 4.5). To illustrate conditional operations, like the CNOT gate (eq. 4.9), we write the circuit as follows:



The black dot indicates that we put the condition on the first qubit being in the $|1\rangle$ state, while the plus sign surrounded by a circle indicates which qubit to apply the X gate (eq. 4.4) on. We refer to the conditional qubit as the control-qubit, whereas we refer to the qubit for which to eventually perform the operation on as the target qubit. We can also indicate conditional operations with an arbitrary gate A as follows:

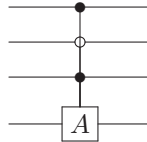


If we want to indicate that we perform a measurement on the bottom qubit in the circuit above, we can use a meter symbol on the wire corresponding to the qubit we measure:

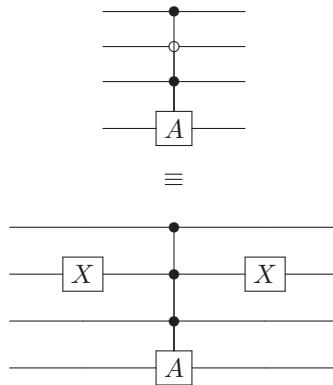


The measurement of a qubit will either result in a 1 or a 0.

There are also multi-controlled qubit gates with more than one control qubit. For example, an operation conditional on three qubits can be written as



You might have noticed that we used a white dot on the second qubit in this scenario. The white dot indicates that we condition on the respective qubit being in the $|0\rangle$ state instead of the $|1\rangle$ state. This is equivalent to acting on the qubit with an X gate (eq. 4.4) before and after the multi-controlled operation, as the X gate flips a qubit from the $|0\rangle$ state to the $|1\rangle$ state and vice versa:



It is relatively easy to read such circuits once the basics are down. However, before diving into the advanced circuits it is a good idea to go through a basic one.

Coin toss example

Say you want to calculate the probability of flipping heads on a coin four times in a row. As we know, each throw has a 50% probability of producing a heads. To get a qubit to mimic a coin toss, we could put it in a superposition with equal probability of measuring the $|0\rangle$ and $|1\rangle$ state. We then refer to the $|1\rangle$ state as a heads and the $|0\rangle$ state as a tails. That is

$$|\psi\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle.$$

As we can see, the probability of measuring the $|0\rangle$ state is given by

$$|\langle 0|\psi\rangle|^2 = \left|\frac{1}{\sqrt{2}}\langle 0|0\rangle\right|^2 = \frac{1}{2},$$

and likewise for the $|1\rangle$ state:

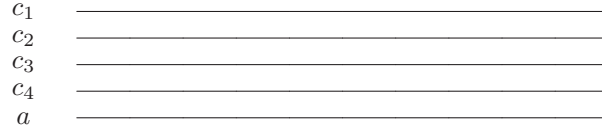
$$|\langle 1|\psi\rangle|^2 = \frac{1}{2}.$$

As we learned earlier, the Hadamard gate (eq. 4.5) produces such a state. The circuit for producing a single coin flip is then

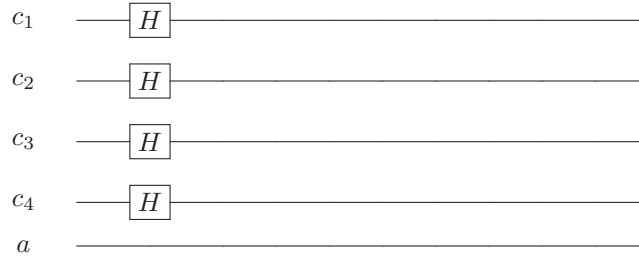


4. Quantum Computing: Many-Body Methods

We could actually use this circuit for our purpose by repeatedly running it four times and counting how many times we get the $|1\rangle$ state in all four runs. However, for educational purposes we will include some more bells and whistles here. Let us instead use a single qubit for each coin toss in our experiment, and also a qubit to save the desired result:



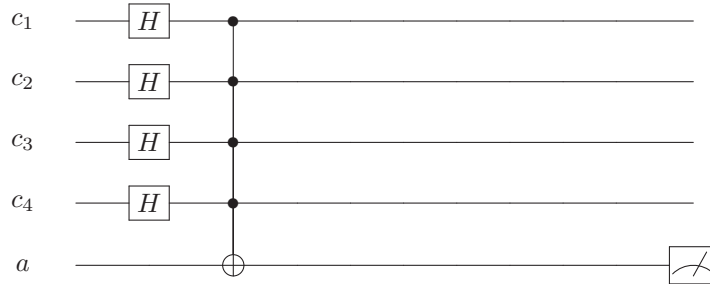
The qubits denoted with a c_i are the simulation qubits, that is, they are used to simulate the four coin tosses. The qubit denoted with an a will be referred to as the ancilla qubit, which is a common name for qubits used to for example encode simulation results. Remember, all the qubits are initialized in the $|0\rangle$ state since nothing else is specified. We start by putting all the simulation qubits in superpositions with the Hadamard gate (eq. 4.5):



The simulation qubits are now representing a separate coin toss. To get more familiar with the notations, one mathematical way to write this circuit is

$$\begin{aligned}
 & H^1 H^2 H^3 H^4 |0\rangle |0\rangle |0\rangle |0\rangle |a=0\rangle \\
 &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) |a=0\rangle \\
 &= \frac{1}{4} \sum_{i=0}^{2^4-1} |i\rangle |a=0\rangle,
 \end{aligned}$$

where the sum runs over all 16 four bit binary strings (0000, 0001, 0010, etc..). In order to extract the probability of measuring four heads, we could use a multi controlled gate along with measurements:



We can see that the ancilla qubit is now flipped to the $|1\rangle$ state if, and only if, all the simulation qubits are in the $|1\rangle$ state, because of the multi-controlled X gate. This property is called entanglement, that is, the state of the ancilla cannot be described independently of the $|c_1\rangle |c_2\rangle |c_3\rangle |c_4\rangle$ state. The wanted probability is now extracted by repeatedly running this circuit and calculating $|\langle 1|a\rangle|^2$, or in other words: Count the number of times $|a\rangle$ is in the $|1\rangle$ state and divide by the total number of experiments. Even though this circuit is simple, it captures most of what is needed to understand more advanced circuits.

Circuit depth

For Noisy Intermediate-Scale Quantum Technology (NISQ), the ability to achieve sensible results relies on the execution time of the circuit for the problem at hand [19]. Circuit depth is an important quantity that is

describing the number of time steps (time complexity) required to perform the circuit [19]. As current quantum devices are not able to maintain a stable state for a long time [47], achieving a short circuit depth is important if one expects to be able to run a quantum algorithm successfully. We will not go into specific details on how to calculate this quantity as it is rather simple to get the depth of a circuit when utilizing Qiskit [42], the Python package we will use to write quantum algorithms. It is nevertheless important to keep the rough explanation of this quantity in mind as we will later talk about methods of reducing it.

4.2 Quantum Phase Estimation

The first we will use to approximate the eigenvalues of the pairing Hamiltonian (eq. 2.25) is called the quantum phase estimation (QPE) algorithm. An important sub-routine of this algorithm is called the quantum Fourier transform.

4.2.1 Quantum Fourier Transform

The quantum Fourier transform (QFT) is a linear transformation on qubits, which is used quite frequently in quantum algorithms. The QFT performed on an orthonormal basis yields the following state [37]

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i j k / N} |k\rangle. \quad (4.11)$$

How this transformation can become useful in different algorithms may not be apparent at first glance, but bear with me. We will quickly see its usefulness in the next section. In order to see how we can implement this transformation on a quantum computer, we first rewrite eq. 4.11 in a more convenient form. First we write the n -qubit state $|j\rangle$ using the binary representation

$$\begin{aligned} j &= j_1 j_2 \cdots j_n \\ j &= j_1 2^{n-1} + j_2 2^{n-2} + \cdots + j_n 2^0. \end{aligned} \quad (4.12)$$

It is also useful to denote

$$0.j_1 j_2 \cdots j_n = j_1/2 + j_2/2^2 + \cdots + j_n/2^n, \quad (4.13)$$

as the binary fraction $0.j$. Eq. 4.11 for an n -qubit state can be written as

$$\frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i j k / 2^n} |k\rangle.$$

Utilizing the binary representation of k (eq. 4.12) gives us

$$\begin{aligned} &= \frac{1}{2^{n/2}} \sum_{k_1=0}^1 \cdots \sum_{k_n=0}^1 e^{2\pi i j (\sum_{l=1}^n k_l 2^{n-l}) / 2^n} |k_1 \cdots k_n\rangle \\ &= \frac{1}{2^{n/2}} \sum_{k_1=0}^1 \cdots \sum_{k_n=0}^1 e^{2\pi i j (\sum_{l=1}^n k_l 2^{-l})} |k_1 \cdots k_n\rangle \\ &= \frac{1}{2^{n/2}} \sum_{k_1=0}^1 \cdots \sum_{k_n=0}^1 \bigotimes_{l=1}^n e^{2\pi i j k_l 2^{-l}} |k_l\rangle \\ &= \frac{1}{2^{n/2}} \bigotimes_{l=1}^n \sum_{k_l=0}^1 e^{2\pi i j k_l 2^{-l}} |k_l\rangle. \end{aligned}$$

Next, we insert $k_l = 0$ and $k_l = 1$

$$= \frac{1}{2^{n/2}} \bigotimes_{l=1}^n [|0\rangle + e^{2\pi i j 2^{-l}} |1\rangle].$$

4. Quantum Computing: Many-Body Methods

We then use the binary representation of j (eq. 4.12)

$$= \frac{1}{2^{n/2}} \bigotimes_{l=1}^n [|0\rangle + e^{2\pi i \sum_{i=1}^n j_i 2^{n-l-i}} |1\rangle].$$

Observe that when $n - l - i \geq 0$, we will just be multiplying the $|1\rangle$ state with 1. Therefore, we have

$$= \frac{1}{2^{n/2}} (|0\rangle + e^{2\pi i 0 \cdot j_n} |1\rangle) (|0\rangle + e^{2\pi i 0 \cdot j_{n-1} j_n} |1\rangle) \cdots (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2 \cdots j_n} |1\rangle), \quad (4.14)$$

which is the desired representation of the Fourier transform. Let us see how to get from an arbitrary state $|j_1 j_2 \cdots j_n\rangle$ to the product representation in eq. 4.14. We will skip any global normalization factors. First we apply a Hadamard gate (eq. 4.5) to the first qubit:

$$H^1 |j_1 j_2 \cdots j_n\rangle = (|0\rangle + e^{2\pi i 0 \cdot j_1} |1\rangle) |j_2 \cdots j_n\rangle.$$

Even though this is not the usual way to write a Hadamard transformed qubit, it is correct since $e^{2\pi i 0 \cdot j_1} = -1$ if $j_1 = 1$ and its 1 if $j_1 = 0$. For the next step, we need to apply a controlled R_2 gate to the first qubit, conditioned on the second qubit (see equation 4.8 for the R_k gate, with $k = 2$). This action results in

$$(|0\rangle + e^{2\pi i 0 \cdot j_1 + 2\pi i j_2 / 2^2} |1\rangle) |j_2 \cdots j_n\rangle = (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2} |1\rangle) |j_2 \cdots j_n\rangle.$$

Continuing with applying a controlled R_k gate on the first qubit conditioned on qubit l for $l = 3, 4, \dots, n$, we will end up with

$$(|0\rangle + e^{2\pi i 0 \cdot j_1 \cdots j_n} |1\rangle) |j_2 \cdots j_n\rangle.$$

We can now apply the Hadamard gate to the second qubit and utilize the controlled R_k gate on the preceding qubits in the same manner. Continuing this way until we reach the final qubit will put us in the desired state. The complete circuit is illustrated below:

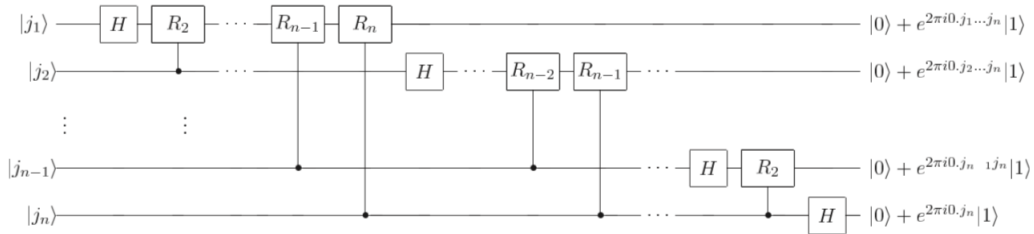


Figure 4.2: QFT circuit. Figure is taken from reference [37].

The gate requirement of the circuit in figure 4.2 is given by [37]

$$\text{QFT Complexity: } \mathcal{O}(n^2), \quad (4.15)$$

where n is the number of qubits.

4.2.2 Phase Estimation Algorithm

Now that we have shown how to do a Quantum Fourier transform, the next question one may ask is how to make use of it. A central procedure in many quantum algorithms is known as quantum phase estimation (QPE). Suppose we have a unitary operator U . This operator has an eigenvector $|u\rangle$, with the corresponding eigenvalue $e^{2\pi i \lambda}$, where λ is unknown. That is

$$U |u\rangle = e^{2\pi i \lambda} |u\rangle. \quad (4.16)$$

The purpose of the QPE algorithm is to estimate λ . The first stage of the algorithm is shown in the figure below:

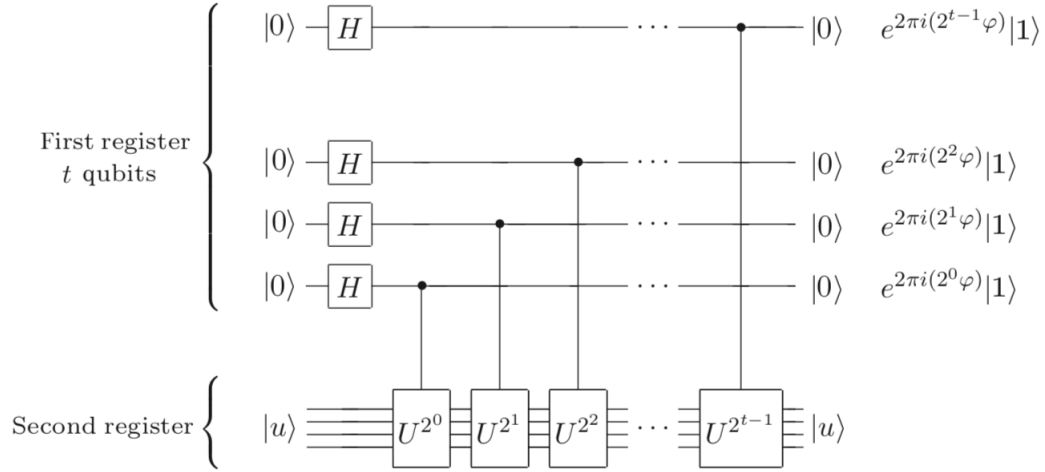


Figure 4.3: Phase estimation circuit. Figure is taken from reference [37].

What we see in fig 4.3 is that we have prepared two quantum registers. The first register has t qubits, all initialised in the $|0\rangle$ state, and the second register is initialized as the eigenvector $|u\rangle$. We will refer to the register with t qubits as the t -register, while we will refer to the second register as the u -register. All the qubits in the t -register are put in a superposition by applying the Hadamard gate (eq. 4.5). We then apply the U^{2^0} operator to the u -register, conditional on the bottom qubit in the t -register. This yields

$$|u\rangle \frac{1}{2^{t/2}} (|0\rangle + |1\rangle) \cdots (|0\rangle + |1\rangle) \left(|0\rangle + e^{2\pi i 2^0 \lambda} |1\rangle \right),$$

which can be seen from eq. 4.16. Applying the U^{2^1} operator to the u -register conditional on the next qubit in the t -register gives

$$|u\rangle \frac{1}{2^{t/2}} (|0\rangle + |1\rangle) \cdots (|0\rangle + |1\rangle) \left(|0\rangle + e^{2\pi i 2^1 \lambda} |1\rangle \right) \left(|0\rangle + e^{2\pi i 2^0 \lambda} |1\rangle \right).$$

Continuing as shown in the circuit gives us finally

$$|u\rangle \frac{1}{2^{t/2}} \left(|0\rangle + e^{2\pi i 2^{t-1} \lambda} |1\rangle \right) \cdots \left(|0\rangle + e^{2\pi i 2^2 \lambda} |1\rangle \right) \left(|0\rangle + e^{2\pi i 2^1 \lambda} |1\rangle \right) \left(|0\rangle + e^{2\pi i 2^0 \lambda} |1\rangle \right).$$

Now suppose that we can write the phase exactly as the binary fraction

$$0.\lambda_1\lambda_2\cdots\lambda_t = \lambda_1/2 + \lambda_2/2^2 + \cdots + \lambda_t/2^t.$$

The bottom qubit in the t -register can then be written as

$$\begin{aligned} |0\rangle + e^{2\pi i 2^0 \lambda} |1\rangle &= |0\rangle + e^{2\pi i (\lambda_1/2 + \lambda_2/2^2 + \cdots + \lambda_t/2^t)} |1\rangle \\ &= |0\rangle + e^{2\pi i 0.\lambda_1\lambda_2\cdots\lambda_t} |1\rangle. \end{aligned}$$

For the next to bottom qubit in the t -register, we can write

$$|0\rangle + e^{2\pi i 2^1 \lambda} |1\rangle = |0\rangle + e^{2\pi i 2(\lambda_1/2 + \lambda_2/2^2 + \cdots + \lambda_t/2^t)} |1\rangle.$$

Since λ_1 has to be either 1 or 0 we can write this as

$$\begin{aligned} |0\rangle + e^{2\pi i 2(\lambda_1/2 + \lambda_2/2^2 + \cdots + \lambda_t/2^t)} |1\rangle &= |0\rangle + e^{2\pi i \lambda_1} e^{2\pi i (\lambda_2/2^1 + \cdots + \lambda_t/2^{t-1})} |1\rangle \\ &= |0\rangle + e^{2\pi i 0.\lambda_2\lambda_3\cdots\lambda_t} |1\rangle. \end{aligned}$$

4. Quantum Computing: Many-Body Methods

Doing this for all the qubits in the t -register gives us

$$\frac{1}{2^{t/2}} (|0\rangle + e^{2\pi i 0 \cdot \lambda_t}) \dots (|0\rangle + e^{2\pi i 0 \cdot \lambda_2 \dots \lambda_t}) (|0\rangle + e^{2\pi i 0 \cdot \lambda_1 \dots \lambda_t}). \quad (4.17)$$

Comparing this state with the QFT state in equation (eq. 4.14), we see that we have the QFT of the state $|\lambda\rangle$. By performing the inverse quantum Fourier transform on the state in eq. 4.17, we will end up with

$$\begin{aligned} (QFT)^{-1} \frac{1}{2^{t/2}} (|0\rangle + e^{2\pi i 0 \cdot \lambda_t}) \dots (|0\rangle + e^{2\pi i 0 \cdot \lambda_2 \dots \lambda_t}) (|0\rangle + e^{2\pi i 0 \cdot \lambda_1 \dots \lambda_t}) |u\rangle \\ = |\lambda_1 \lambda_2 \dots \lambda_t\rangle |u\rangle. \end{aligned} \quad (4.18)$$

In other words, we get the exact phase encoded in the t -register. In reality though, we will not necessarily know the eigenvector $|u\rangle$. To deal with this, we can prepare the u -register in a state $|\psi\rangle = \sum_{i=1}^n c_i |u_i\rangle$ which is a linear combination of the eigenstates of U . Repeated applications of the phase estimation algorithm followed by measurements will then yield a specter of eigenvalues and eigenvectors. We also can not always express the phase exactly as a t -bit binary fraction. It can be shown that we will with high probability produce a pretty good estimation to λ nevertheless [36].

Since all operations on qubits are unitary, we can complex conjugate all the operations in the QFT circuit (fig. 4.2) and apply them in the reverse order to yield the inverse Fourier transform. We can show this by considering some arbitrary unitary operations on a state $|j\rangle$:

$$\begin{aligned} ABCD \dots N |j\rangle &= |k\rangle \\ (ABCD \dots N)^\dagger |k\rangle &= (ABCD \dots N)^\dagger ABCD \dots N |j\rangle \\ &= N^\dagger \dots D^\dagger C^\dagger B^\dagger A^\dagger ABCD \dots N |j\rangle = |j\rangle. \end{aligned}$$

To summarize, the QPE algorithm is performed by first applying the circuit in figure 4.3 to the u and t -register. We then apply the inverse QFT (section 4.2.1) on the t -register. The inverse QFT requires $\mathcal{O}(t^2)$ operations (eq. 4.15), where t is the number of qubits in the t -register. The complexity of the complete QPE algorithm is then dependent on the number of operations required to implement the controlled U^{2^j} operations.

Our plan is to use this method to approximate the eigenvalues of the pairing Hamiltonian (eq. 2.25). Before we do this, we will have to introduce two important formulas.

4.2.3 The Suzuki-Trotter transformation

The Suzuki-Trotter approximation states that given some unitary operators $\hat{A}_1, \hat{A}_2, \hat{A}_3, \dots$ that do not necessarily commute, we have for any real t

$$e^{it(\hat{A}_1 + \hat{A}_2 + \hat{A}_3 + \dots)} = \lim_{m \rightarrow \infty} (e^{i \frac{t}{m} \hat{A}_1} e^{i \frac{t}{m} \hat{A}_2} e^{i \frac{t}{m} \hat{A}_3} \dots)^m. \quad (4.19)$$

This can be shown by utilizing the Taylor expansion of $e^{it(\hat{A}_1 + \hat{A}_2 + \hat{A}_3 + \dots)}$. We can also in the same manner show that [37]

$$e^{i\Delta t(\hat{A}_1 + \hat{A}_2 + \hat{A}_3 + \dots)} = e^{i\Delta t \hat{A}_1} e^{i\Delta t \hat{A}_2} e^{i\Delta t \hat{A}_3} \dots + \mathcal{O}(\Delta t^2), \quad (4.20)$$

hence we can approximate the action of $e^{it(\hat{A}_1 + \hat{A}_2 + \hat{A}_3 + \dots)}$ to arbitrary precision by utilizing eq. 4.20 repeatedly with small enough Δt .

4.2.4 The Jordan-Wigner transformation

The Jordan-Wigner transformation is a Transformation that maps the Pauli gates (eq. 4.4) onto fermionic creation and annihilation operators [35]. The creation and annihilation operators from the second quantization formalism (see section 2.2) can then be represented on quantum computers, and we will be able to rewrite our second quantization Hamiltonian (eq. 2.25) in terms of quantum gates. Suppose that we represent a qubit in state $|0\rangle$ as a state occupied with a fermion and $|1\rangle$ as a state with no fermion. We then see that the operators

$$\begin{aligned}\sigma_+ &= \frac{1}{2}(\sigma_x + i\sigma_y) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \\ \sigma_- &= \frac{1}{2}(\sigma_x - i\sigma_y) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix},\end{aligned}\tag{4.21}$$

have the following effect on the qubit basis states

$$\sigma_+ |1\rangle = |0\rangle \quad \sigma_- |0\rangle = |1\rangle,$$

and

$$\sigma_+ |0\rangle = 0 \quad \sigma_- |1\rangle = 0.$$

Hence, σ_+ acts as a creation operator and σ_- acts as an annihilation operator. However, since fermionic states are anti-symmetric, $a_a^\dagger a_b^\dagger |c\rangle = -a_b^\dagger a_a^\dagger |c\rangle$, we need our quantum gate representation of the creation/annihilation operators to preserve this property. This can be achieved by multiplying the σ_z matrix (eq. 4.4) on all the occupied states leading up to the one we operate on. The complete creation and annihilation operators can then be represented as

$$a_n^\dagger \equiv \left(\prod_{k=1}^{n-1} \sigma_z^k \right) \sigma_+^n \quad a_n \equiv \left(\prod_{k=1}^{n-1} \sigma_z^k \right) \sigma_-^n\tag{4.22}$$

where the superscript tells us which qubit the operator acts on. For convenience, we chose that odd qubits are in a spin up state, while even qubits are in spin down state. For example for the following state

$$\begin{aligned}\text{Qubit state: } &|0 \ 0 \ 1 \ 1\rangle \\ \text{Spin state: } &+ \ - \ + \ - \\ \text{Spacial state: } &1 \ 1 \ 2 \ 2,\end{aligned}$$

the first spacial basis state is occupied with a Fermion pair with opposite spin, while the second spacial state is not occupied with any Fermions.

Jordan-Wigner transformation of Pairing Hamiltonian

We can now write the second quantization pairing Hamiltonian (eq. 2.25) in terms of the Pauli matrices. A detailed derivation is provided in appendix B. Here we simply state the result: For the one body part of the Hamiltonian we have the terms

$$\hat{H}_{0p} = \frac{1}{2} \delta(p-1 - I[p \% 2 = 0]) (I^p + \sigma_z^p),\tag{4.23}$$

where we sum over each qubit p and $\%$ is the modulo operator. We have that $I[f(x) = y] = 1$ if $f(x) = y$ and zero otherwise. For the interaction term we have two possibilities. First for $p = q$ we have:

$$\begin{aligned}\hat{V}_p &= -\frac{1}{8} g \left[I^{\otimes 2p-2} \otimes I \otimes I \otimes I^{\otimes n-2p} \right. \\ &\quad + I^{\otimes 2p-2} \otimes I \otimes \sigma_z \otimes I^{\otimes n-2p} \\ &\quad + I^{\otimes 2p-2} \otimes \sigma_z \otimes I \otimes I^{\otimes n-2p} \\ &\quad \left. + I^{\otimes 2p-2} \otimes \sigma_z \otimes \sigma_z \otimes I^{\otimes n-2p} \right],\end{aligned}\tag{4.24}$$

and for $q - p \geq 1$ we get:

$$\begin{aligned} \hat{V}_{pq} = & -\frac{1}{16}g[I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2q} \\ & - I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2q} \\ & + I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2q} \\ & + I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2q} \\ & + I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2q} \\ & + I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2q} \\ & - I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2q} \\ & + I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2q}]. \end{aligned} \quad (4.25)$$

We have included a factor of two so the sum over p and q here can be restricted to $q > p$. The complete Jordan-Wigner transformed pairing Hamiltonian can then be written as

$$\hat{H} = \sum_p \hat{H}_{0p} + \sum_p \hat{V}_p + \sum_{q>p} \hat{V}_{pq}, \quad (4.26)$$

where \hat{H}_{0p} , \hat{V}_p and \hat{V}_{pq} is given by eqs. 4.23, 4.24 and 4.25, respectively.

4.2.5 Hamiltonian Simulation

We are now ready to find the eigenvalues of a fermionic Hamiltonian using the quantum phase estimation algorithm (section 4.2.2). To see how this can be done, consider that a (time independent) quantum state evolves according to the time evolution operator

$$|\psi(t)\rangle = e^{-i\hat{H}t/\hbar} |\psi(0)\rangle = \hat{U} |\psi(0)\rangle, \quad (4.27)$$

where \hat{H} is the Hamiltonian. We also know that an eigenstate ψ_k of the \hat{H} is also an eigenstate of the time evolution operator. Its eigenvalue is given by

$$e^{-i\hat{H}t/\hbar} |\psi_k\rangle = e^{-iE_k t/\hbar} |\psi_k\rangle, \quad (4.28)$$

where E_k is the k 'th eigenvalue of \hat{H} . The QPE algorithm finds the phase λ_k of a unitary operator with eigenvalue $e^{-i\lambda_k 2\pi}$, so if we can approximate the time evolution operator on a quantum computer, we can also approximate its eigenvalues with the QPE algorithm. We have shown in section 4.2.4 how to write our Hamiltonian in terms of the Pauli matrices:

$$\hat{H} = \sum_p \hat{H}_{0p} + \sum_p \hat{V}_p + \sum_{q>p} \hat{V}_{pq}, \quad (4.29)$$

where \hat{H}_{0p} , \hat{V}_p and \hat{V}_{pq} is given by eqs. 4.23, 4.24 and 4.25, respectively. The corresponding time evolution operator is then given by

$$\hat{U}(t) = e^{-i(\sum_p \hat{H}_{0p} + \sum_p \hat{V}_p + \sum_{q>p} \hat{V}_{pq})t}. \quad (4.30)$$

The Suzuki-Trotter approximation in eq. 4.20 can now be utilized to approximate a small time-step with this operator:

$$\hat{U}(\Delta t) = \prod_p e^{-i\hat{H}_{0p}\Delta t} \prod_p e^{-i\hat{V}_p\Delta t} \prod_{q>p} e^{-i\hat{V}_{pq}\Delta t} + \mathcal{O}(\Delta t^2). \quad (4.31)$$

We will now show how each of the separate exponential operators in eq. 4.31 can be implemented on a quantum computer. Consider that $e^{-i\sigma_z \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z \Delta t}$ can be implemented with the following circuit [37]

$$e^{-i\sigma_z \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z \Delta t} \equiv \text{Circuit}, \quad (4.32)$$

where the operator $e^{-i\Delta t Z}$ is given by the $R_z(\theta)$ gate (eq. 4.6). We can easily extend this circuit with more (less) control qubits to apply longer (shorter) strings of σ_z gates.

With the use of the following identities

$$\sigma_x = H\sigma_z H, \quad (4.33)$$

and

$$\sigma_y = R_z(\pi/2)H\sigma_z H R_z(-\pi/2), \quad (4.34)$$

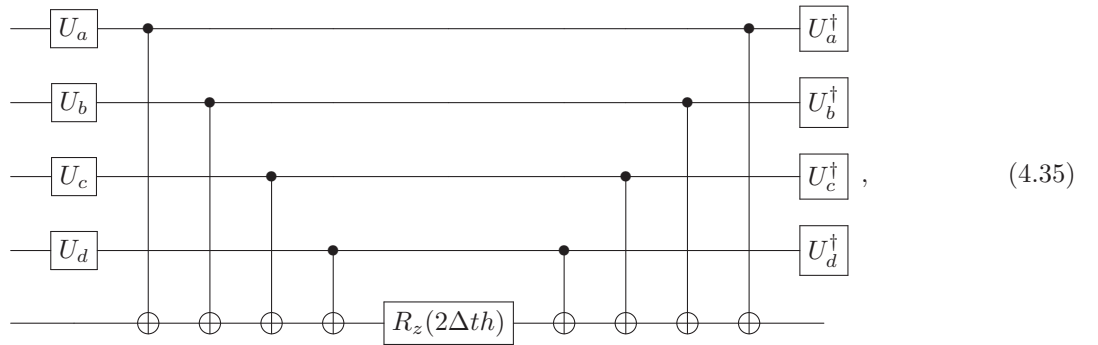
we can use circuit 4.32 to implement the time evolution of an arbitrary string of Pauli matrices. For example, for the tensor product $H = \sigma_x \otimes \sigma_y \otimes \sigma_x \otimes \sigma_y$, we have

$$\begin{aligned} & H \otimes R_z(\pi/2)H \otimes H \otimes R_z(\pi/2)H \\ & \times e^{-i\Delta t \sigma_x \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z} \\ & \times H \otimes H R_z(-\pi/2) \otimes H \otimes H R_z(-\pi/2) \\ & = U e^{-i\Delta t \sigma_x \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z} U^\dagger \\ & = U [\cos(\Delta t)I - i \sin(\Delta t) \sigma_x \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z] U^\dagger \\ & = \cos(\Delta t)I - i \sin(\Delta t) \sigma_x \otimes \sigma_y \otimes \sigma_x \otimes \sigma_y \\ & = e^{-i\Delta t \sigma_x \otimes \sigma_y \otimes \sigma_x \otimes \sigma_y}, \end{aligned}$$

since $UU^\dagger = I$ and

$$\begin{aligned} & U \times (\sigma_z \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z) \times U^\dagger \\ & = (H \otimes R_z(\pi/2)H \otimes H \otimes R_z(\pi/2)H) \\ & \quad \times (\sigma_z \otimes \sigma_z \otimes \sigma_z \otimes \sigma_z) \\ & \quad \times (H \otimes H R_z(-\pi/2) \otimes H \otimes H R_z(-\pi/2)) \\ & = H\sigma_z H \otimes R_z(\pi/2)H\sigma_z H R_z(-\pi/2) \otimes H\sigma_z H \otimes R_z(\pi/2)H\sigma_z H R_z(-\pi/2) \\ & = \sigma_x \otimes \sigma_y \otimes \sigma_x \otimes \sigma_y. \end{aligned}$$

Hence, if we have a Pauli operator σ_a , where $a \in x, y, z$ and $\sigma_a = U_a \sigma_z U_a^\dagger$, we can implement the time evolution $e^{-\Delta t h \sigma_a \otimes \sigma_b \otimes \sigma_c \otimes \sigma_d}$ with the following circuit



where h is a real factor. We can see that this circuit is efficient since we at most require an amount of operations linear in the amount of qubits. Hence, we can simulate the time evolution of any Hamiltonian efficiently as long as the number of terms in the Hamiltonian is polynomial in the amount of qubits [37]. We see from the Jordan-Wigner transformation of our pairing Hamiltonian (section 4.2.4) that this is the case for our problem.

Getting the complete eigenvalue spectra

The phase estimation algorithm is aimed at finding λ_k for a unitary operator \hat{U} with eigenvalue $e^{i2\pi\lambda_k}$, such that

$$\hat{U}|\psi_k\rangle = e^{i2\pi\lambda_k}|\psi_k\rangle.$$

In our case, the time evolution operator applied for a time τ has the eigenvalues $e^{-iE_k\tau}$, which means that the value we read from the phase estimation algorithm is

$$\begin{aligned} i2\pi\lambda_k &= -iE_k\tau \\ \implies \lambda_k &= -\frac{iE_k\tau}{i2\pi} \\ \implies \lambda_k &= -E_k\tau/2\pi. \end{aligned} \tag{4.36}$$

An assumption with the phase estimation algorithm is that we can write the eigenvalue as a binary fraction. Since a binary fraction is a positive number, we need our eigenvalues to be negative for the phase estimation algorithm to work, according to the above equation for λ_k (eq. 4.36). We can force this by subtracting a large enough constant value E_{max} from the Hamiltonian, since

$$(\hat{H} - E_{max})|\phi_k\rangle = (E_k - E_{max})|\phi_k\rangle.$$

This gives us

$$\begin{aligned} \lambda_k &= -(E_k - E_{max})\tau/2\pi \\ \implies \lambda_k 2\pi/\tau &= E_{max} - E_k \\ \implies E_k &= E_{max} - \lambda_k 2\pi/\tau. \end{aligned} \tag{4.37}$$

Further, since the phase estimation algorithm yields the following state for the t -register (eq. 4.18)

$$|\lambda_1\lambda_2\cdots\lambda_{n_t}\rangle = |\lambda 2^{n_t}\rangle,$$

where n_t is the number of qubits in the t -register, we need to transform the measured binary number $\lambda_1\lambda_2\cdots\lambda_{n_t}$ to a binary fraction $0.\lambda_1\lambda_2\cdots\lambda_{n_t}$ before plugging it into the equation for E_k .

We can also see that if we have $\lambda = \lambda' + n > 1$ where $0 < \lambda' < 1$ and n is a positive integer, we get from the phase estimation algorithm

$$e^{i2\pi(\lambda'+n)} = e^{i2\pi n} e^{i2\pi\lambda'} = e^{i2\pi\lambda'},$$

or written in binary form

$$e^{i2\pi(\lambda'+n)} = e^{i2\pi\lambda_1\cdots\lambda_k.\lambda_{k+1}\cdots\lambda_n} = e^{i2\pi 0.\lambda_{k+1}\cdots\lambda_n}.$$

In other words; for eigenvalues greater than one, we lose information. A restriction on $\lambda_k < 1$ in eq. 4.36 gives

$$\begin{aligned} -E_k\tau/2\pi &< 1 \\ \implies -E_k\tau &< 2\pi \end{aligned}$$

or

$$-(E_k - E_{max})\tau < 2\pi.$$

Substituting E_k with E_{min} (the lowest eigenvalue of \hat{H}) gives an upper bound on t in order to yield the whole eigenvalue spectrum

$$t < \frac{2\pi}{E_{max} - E_{min}}. \tag{4.38}$$

We also have to keep in mind the number of qubits to use in the t -register. If we use k qubits we can represent 2^k binary fractions. A quantum state represented by s simulation qubits potentially has 2^s eigenvalues. With a t -register of k qubits, this means that we will have $\frac{2^k}{2^s} = 2^{k-s}$ points for each eigenvalue. Previous research has claimed that a surplus of around 5 qubits in the t -register are usually sufficient to yield the complete eigenvalue-spectra [38].

A summary of the QPE algorithm is given below:

- Subtract a large enough constant E_{max} from the problem Hamiltonian.
- Prepare two registers. A register of t -qubits (t -register) and a register of u qubits (u -register).
- Put the u -register in a linear combinations of the eigenstates of the problem Hamiltonian \hat{H} . This can be done by applying a Hadamard gate (eq. 4.5) to each of the qubits. This will yield a superposition of all the computational basis states.
- Apply the QPE circuit (figure 4.3), where U is given by the Suzuki-Trotter approximation (section 4.2.3) of the Hamiltonian time evolution operator. The evolution time is bounded by eq. 4.38.
- Apply the inverse of the QFT circuit (figure 4.2) to the t -register.
- Measure all qubits in the t -register to yield a binary fraction
- Use eq. 4.37 to obtain the measured eigenvalue from the binary fraction.
- Repeat to obtain a spectre of eigenvalues.

4.3 Variational Quantum Eigensolvers

The variational principle states that the expectation value of the Hamiltonian has to be larger than or equal to the ground state energy of the system. Mathematically this can be expressed as

$$\langle \psi | H | \psi \rangle \geq E_0. \quad (4.39)$$

We can understand this principle intuitively by considering that no single measurement of the energy can be lower than the ground state energy. Hence, the expectation value of the energy can neither. Variational methods make use of this principle by calculating the expectation value in equation 4.39 for what we call a trial wavefunction $|\psi_T(\theta)\rangle$:

$$\langle \psi_T(\theta) | H | \psi_T(\theta) \rangle = E(\theta).$$

The variational parameters $\theta = [\theta_1, \theta_2, \dots, \theta_p]$ are then varied to minimize $E(\theta)$, which hopefully makes a good approximation for E_0 . For variational quantum eigensolvers (VQE), the trial wave function is given by a parametrized n -qubit state

$$U(\theta) |\psi_0\rangle = |\psi_T(\theta)\rangle,$$

where $U(\theta)$ is some parametrized multi-qubit gate and $|\psi_0\rangle$ is the initial state of the qubits. As long as the Hamiltonian can be rewritten as a sum of quantum gates O_i

$$H = \sum_{i=1}^m h_i O_i,$$

we can find its expectation value by considering the expectation of each term

$$\langle \psi_T(\theta) | H | \psi_T(\theta) \rangle = \sum_{i=1}^m h_i \langle \psi_T(\theta) | O_i | \psi_T(\theta) \rangle. \quad (4.40)$$

How to perform these steps are best described by considering an example, namely the Max-Cut problem.

4.3.1 Max-Cut problem

The Max-Cut problem is one of the hardest combinatorial optimization problems to solve, yet its one of the easiest to conceptualize. The aim of this section is to explain a quantum variational eigensolver by solving the Max-Cut problem. Max-cut can be understood by considering this graph

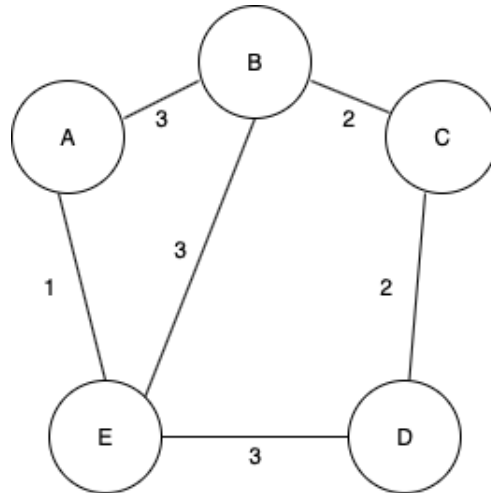


Figure 4.4: Unsolved max-cut graph

The circles are called the nodes of the graph and the lines connecting two nodes are called edges. Now consider that you are allowed to color each node in either red or blue. The numbers next to the edges of the graph are called weights, and they state the number of points gained if the nodes the edge connects to are of different colors. Solving a Max-Cut problem corresponds to coloring the graph in such a way that you have the maximum amount of points. A solution to this Max-Cut problem is represented in the graph below

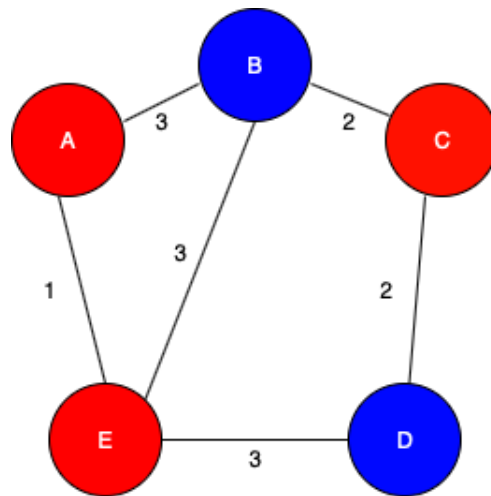


Figure 4.5: Solved max-cut graph

yielding a total of 13 points. We gain points from all the weights except of the one representing the connection from node A to E, as these are both of the same coloring. Given a graph with n nodes, all its information can be expressed with an n by n matrix W . Its entries w_{ij} correspond to the points given by the edge connecting node i and node j . To give an example, the matrix for the graph given in figs. 4.4 and 4.5 is

$$W = \begin{pmatrix} A & B & C & D & E \\ 0 & 3 & 0 & 0 & 1 \\ 3 & 0 & 2 & 0 & 3 \\ 0 & 2 & 0 & 2 & 0 \\ 0 & 0 & 2 & 0 & 3 \\ 1 & 3 & 0 & 3 & 0 \end{pmatrix} \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}.$$

We can express a profit-function for the Max-Cut problem with the help of the matrix elements in any Max-Cut matrix W , by representing the color of node i with a binary number $x_i \in \{0, 1\}$ [32]:

$$C(\mathbf{x}; W) = \sum_{i,j} w_{ij} x_i (1 - x_j). \quad (4.41)$$

The coloring \mathbf{x} which yields this functions highest value is the solution to the Max-Cut problem. We now want to map the function $C(\mathbf{x})$ into a Hamiltonian in such a way that it can be evaluated on a quantum computer. This can be done by first expressing the coloring of a given n -noded graph with an n -qubit state $|q_1\rangle |q_2\rangle \cdots |q_n\rangle$, where q_i corresponds to the coloring x_i . We then do the following mapping in the cost function [32]

$$x_i \rightarrow \frac{1 - \sigma_z^i}{2}, \quad (4.42)$$

where σ_z is the Pauli- Z gate (eq. 4.4). We can see that this will successfully evaluate $C(\mathbf{x})$ since

$$\frac{1 - \sigma_z^i}{2} |0\rangle = 0 \quad \text{and} \quad \frac{1 - \sigma_z^i}{2} |1\rangle = |1\rangle.$$

Inserting eq. 4.42 into the cost function in eq. 4.41 gives

$$\begin{aligned} \sum_{i,j} w_{ij} x_i (1 - x_j) &\rightarrow \sum_{i,j} w_{ij} \frac{1 - \sigma_z^i}{2} \left(1 - \frac{1 - \sigma_z^j}{2}\right) \\ &= \sum_{i,j} w_{ij} \left[\frac{1 - \sigma_z^i}{2} - \frac{1 - \sigma_z^i}{2} \frac{1 - \sigma_z^j}{2} \right] \\ &= \sum_{i,j} w_{ij} \left(\frac{1 - \sigma_z^i}{2} - \frac{1}{4} [1 - \sigma_z^i - \sigma_z^j + \sigma_z^i \sigma_z^j] \right). \end{aligned}$$

Since the terms without any Pauli matrices are constant when varying the quantum state, we can omit these terms from the above equation. Also, when dealing with terms with only a single qubit gate, we are free to exchange the variable i with j without altering the resulting equation. This removes all single-qubit gates. We are then free to multiply the equation with a factor of two to restrict the sum to $i < j$. When realizing that global factors do not contribute to the location of the minima in parameter space, we are left with [32]

$$H = \sum_{i < j} w_{ij} \sigma_z^i \sigma_z^j. \quad (4.43)$$

Now that we have the Hamiltonian for our Max-Cut problem, we need to explain how we evaluate the expectation values in eq. 4.40.

4.3.2 VQE Expectation Values

Like stated earlier (eq. 4.40), the expectation value of our Hamiltonian is the sum of the expectation value of each term. We have set up an Hamiltonian for our problem, so how do we evaluate the expectation values then? Let us first consider how to handle Pauli- Z expectation values, as the Max-Cut Hamiltonian is only concerning these. The first eigenstate of the Pauli- Z matrix is $|0\rangle$ with an eigenvalue of 1 and the second is $|1\rangle$ with an eigenvalue of -1. We know from quantum mechanics that if we act upon a qubit with the Pauli- Z gate and

4. Quantum Computing: Many-Body Methods

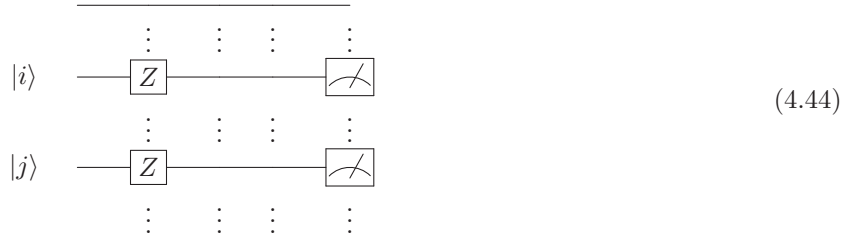
perform a measurement, it will collapse to one of its eigenstates. Therefore, if we measure the state $\sigma_z^i \sigma_z^j |\psi_T(\theta)\rangle$, we can retrieve the resulting eigenvalue by considering the state of qubit i and j :

$ q_i\rangle q_j\rangle$	Eigenvalue
$ 0\rangle 0\rangle$	$1 \cdot 1 = 1$
$ 1\rangle 1\rangle$	$(-1) \cdot (-1) = 1$
$ 0\rangle 1\rangle$	$1 \cdot (-1) = -1$
$ 1\rangle 0\rangle$	$(-1) \cdot 1 = -1$

The expectation value $\langle \psi | \sigma_z^i \sigma_z^j | \psi \rangle$ is then approximated by repeatedly measuring $\sigma_z^i \sigma_z^j | \psi \rangle$ and averaging the obtained eigenvalues. The final step is to multiply the expectation value with the corresponding matrix element w_{ij} (eq. 4.43). This is done separately with each term in eq. 4.43 to finally yield

$$\langle \psi_T(\theta) | H | \psi_T(\theta) \rangle = \sum_{i < j} w_{ij} \langle \psi_T(\theta) | \sigma_z^i \sigma_z^j | \psi_T(\theta) \rangle.$$

The following circuit is to be run for all i and j subject to $i < j$ to obtain the above expectation values



Here $|i\rangle$ denotes the i 'th qubit. From this example, we have learned that the circuit for finding the expectation for a Pauli-Z matrix is



We will sometimes run into other matrices than the Pauli-Z matrix, namely the Pauli-X and Pauli-Y matrices (eq. 4.4). To calculate the expectation values when these operators come into play, we have to introduce some tricks. Let us first consider the eigenvalues and eigenstates of the Pauli-X matrix. They are given by

$$\begin{aligned} X \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ X \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) &= -\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle), \end{aligned} \quad (4.46)$$

where we see that the top state and bottom state has an eigenvalue of 1 and -1 respectively. Let us see what happens if we apply a Hadamard gate (eq. 4.5) to the first eigenstate:

$$H \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |0\rangle.$$

Likewise with the second eigenstate:

$$H \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |1\rangle.$$

Hence, the same gate applied to each of these states transforms them into their own computational basis state. This means that we can obtain the corresponding eigenvalue by running this circuit



and conclude that we have obtained an eigenvalue of 1 when we measure the qubit in the $|0\rangle$ state and an eigenvalue of -1 when we measure the qubit in the $|1\rangle$ state. For the Pauli-Y matrix, we have the following eigenvalues and eigenstates

$$\begin{aligned} Y \frac{1}{2}(|0\rangle + i|1\rangle) &= \frac{1}{2}(|0\rangle + i|1\rangle) \\ Y \frac{1}{2}(|0\rangle - i|1\rangle) &= -\frac{1}{2}(|0\rangle - i|1\rangle), \end{aligned} \quad (4.48)$$

where the first eigenstate and the second eigenstate has an eigenvalue of 1 and -1, respectively. Like we did with the Pauli- X matrix, we can revert each of the eigenstates back to the computational basis with a unitary transformation. Consider

$$HS\frac{1}{2}(|0\rangle + i|1\rangle) = H\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |0\rangle,$$

and

$$HS\frac{1}{2}(|0\rangle - i|1\rangle) = H\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |1\rangle,$$

where S is the phase shift gate, see eq. 4.7. Hence, we can find the eigenvalues with this circuit

$$\text{---} \boxed{Y} \text{---} \boxed{S} \text{---} \boxed{H} \text{---} \boxed{\text{---}} \text{---} \quad (4.49)$$

and follow the same conclusions as we did for the Pauli- X gate.

4.3.3 Variational ansatz / Trial state

Before running the circuits to obtain the expectation values, we need to have set up a parametrized variational state. The variational state $|\psi_T(\boldsymbol{\theta})\rangle$, also called the wavefunction ansatz, is usually set up with a combination of some unitary transformation which causes sufficient entanglement U_{ent} between the qubits and some sort of rotation $U_a(\boldsymbol{\theta})$ on the qubits depending on the angles $\boldsymbol{\theta} = [\theta_1, \theta_2, \dots, \theta_p]$. We will consider a couple of ansatzes in this thesis. The simplest one consists of only $R_y(\theta)$ gates (eq. 4.6) and CNOT gates (eq. 4.9). The circuit to initialize this ansatz is shown below

$$|\psi_T(\boldsymbol{\theta})\rangle = \begin{array}{c} \text{---} \boxed{R_y(\theta_1)} \text{---} \bullet \text{---} \dots \\ \quad \quad \quad \quad \quad \downarrow \\ \text{---} \boxed{R_y(\theta_2)} \text{---} \oplus \text{---} \bullet \text{---} \dots \\ \quad \quad \quad \quad \quad \downarrow \\ \text{---} \boxed{R_y(\theta_3)} \text{---} \oplus \text{---} \bullet \text{---} \dots \\ \quad \quad \quad \quad \quad \downarrow \\ \text{---} \boxed{R_y(\theta_4)} \text{---} \oplus \text{---} \dots \\ \quad \quad \quad \quad \quad \vdots \end{array}, \quad (4.50)$$

where the same pattern of applying $R_y(\theta)$ - gates and CNOT gates to neighboring qubits continues til we reach the final qubit.

Since an arbitrary single-qubit Euler rotation can be expressed in terms of a combination of R_z and R_x gates, the second ansatz will be prepared with some entanglement gate U_{ent} interleaved between such arbitrary rotations. This type of trial state for n qubits can be written as [32]

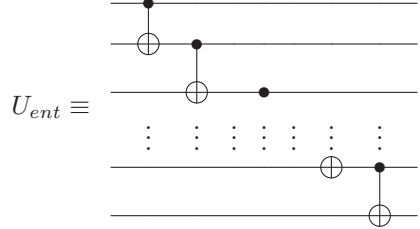
$$\begin{aligned}
|\psi_T(\boldsymbol{\theta})\rangle &= \left[\prod_{q=1}^n U^{[q,d]}(\boldsymbol{\theta}^{[q,d]}) \right] \times U_{ent} \times \left[\prod_{q=1}^n U^{[q,d-1]}(\boldsymbol{\theta}^{[q,d-1]}) \right] \times U_{ent} \\
&\times \cdots \times U_{ent} \times \left[\prod_{q=1}^n U^{[q,1]}(\boldsymbol{\theta}^{[q,1]}) \right] |000 \cdots 0\rangle, \tag{4.51}
\end{aligned}$$

4. Quantum Computing: Many-Body Methods

where n is the number of qubits, d is the number of successive applications of U_{ent} $\left[\prod_{q=1}^n U^{[q,k]}(\boldsymbol{\theta}^{[q,k]})\right]$, and

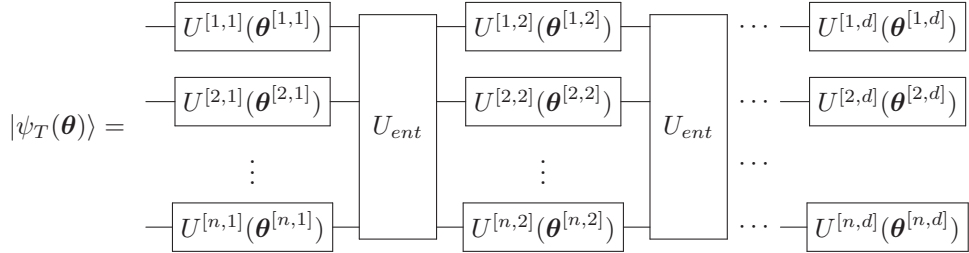
$$U^{[k,l]}(\boldsymbol{\theta}^{[k,l]}) = R_z(\theta_1^{[k,l]})R_x(\theta_2^{[k,l]})R_z(\theta_3^{[k,l]}). \quad (4.52)$$

We restrict ourselves to the following entanglement gate in this thesis:



$$U_{ent} \equiv \quad (4.53)$$

We can see that the number of parameters required to optimize over is $3nd$. The circuit for the Euler rotation ansatz is shown below



$$|\psi_T(\boldsymbol{\theta})\rangle = \quad (4.54)$$

Even though these ansatzes are fit to solve the Max-Cut graph, there is a problem with using these for the pairing Hamiltonian: The ansatzes do not preserve the particle number. For any number of spin-orbitals, we will possibly end up in the lowest energy configuration the ansatz is flexible enough to produce. However, the ansatz is indifferent whether this is a one-particle or fifty-particle state. For the pairing Hamiltonian (eq. 4.26), it is crucial that we can specify the specific configuration we wish to solve for. Hence, we will now introduce a particle-number conserving ansatz, the Unitary Coupled Cluster Ansatz.

4.3.4 Unitary Coupled Cluster ansatz

As we learned in section 2.8 the coupled cluster ansatz is given by

$$|\psi_{CC}\rangle = e^{\hat{T}} |c\rangle,$$

with the cluster operator \hat{T} given by eq. 2.42 and $|c\rangle$ being our reference state (see eq. 2.10). This type of ansatz is not implementable on a quantum computer since $e^{\hat{T}}$ is not a unitary operator. Unitary coupled cluster instead suggest that we write our ansatz as

$$|\psi_{UCC}\rangle = e^{\hat{T}-\hat{T}^\dagger} |c\rangle, \quad (4.55)$$

where \hat{T} is the usual coupled cluster operator (eq. 2.42). One can show that $e^{\hat{T}-\hat{T}^\dagger}$ is unitary [45]. For this thesis, we will restrict ourselves to the unitary coupled cluster doubles (UCCD) method and hence we will deal with the cluster operator

$$\hat{T}_{UCCD} = \hat{T} - \hat{T}^\dagger = \sum_{ijab} t_{ij}^{ab} (a_a^\dagger a_b^\dagger a_j a_i - a_i^\dagger a_j^\dagger a_b a_a), \quad (4.56)$$

where we vary the trial wavefunction over the real cluster amplitudes t_{ij}^{ab} . Note that our ansatz is dependent on the number of particles in our system as we sum over i and j . Hence, we can specify the specific configuration

we wish to solve for, unlike the previously discussed ansatzes. Rewriting eq. 4.56 in terms of Pauli gates by utilizing the Jordan-Wigner transformation (see section 4.2.4) gives [45]

$$\begin{aligned}
 t_{ij}^{ab}(a_a^\dagger a_b^\dagger a_j a_i - a_i^\dagger a_j^\dagger a_b a_a) &= \frac{it_{ij}^{ab}}{8} \bigotimes_{k=i+1}^{j-1} \sigma_z^k \bigotimes_{l=a+1}^{b-1} \sigma_z^l \\
 &\quad (\sigma_x^i \sigma_x^j \sigma_y^a \sigma_y^b + \sigma_y^i \sigma_y^j \sigma_x^a \sigma_x^b \\
 &\quad + \sigma_x^i \sigma_y^j \sigma_y^a \sigma_y^b + \sigma_x^i \sigma_x^j \sigma_x^a \sigma_y^b \\
 &\quad - \sigma_y^i \sigma_x^j \sigma_x^a \sigma_x^b - \sigma_x^i \sigma_y^j \sigma_x^a \sigma_x^b \\
 &\quad - \sigma_y^i \sigma_y^j \sigma_y^a \sigma_x^b - \sigma_y^i \sigma_y^j \sigma_x^a \sigma_y^b),
 \end{aligned} \tag{4.57}$$

where we can assume that $i < j < a < b$. The subscript denotes which qubit we act upon with the Pauli gates. The preparation of this trial wavefunction is done on a quantum computing by first utilizing the Suzuki-Trotter approximation (see section 4.2.3) on the operator in eq. 4.57. Denoting

$$\hat{Z}_{ij}^{ab} = i \frac{t_{ij}^{ab}}{8p} \left(\bigotimes_{k=i+1}^{j-1} \sigma_z^k \right) \left(\bigotimes_{l=a+1}^{b-1} \sigma_z^l \right),$$

the Suzuki-Trotter approximation gives

$$\begin{aligned}
 |\psi(\mathbf{t})\rangle &\approx \left(\prod_{ijab} e^{\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_x^j \sigma_y^a \sigma_y^b} e^{\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_x^a \sigma_x^b} e^{\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_y^j \sigma_y^a \sigma_y^b} e^{\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_x^j \sigma_x^a \sigma_y^b} \right. \\
 &\quad \left. e^{-\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_x^a \sigma_x^b} e^{-\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_y^j \sigma_x^a \sigma_x^b} e^{-\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_y^a \sigma_y^b} e^{-\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_x^a \sigma_y^b} \right)^p |c\rangle,
 \end{aligned} \tag{4.58}$$

where p decides the step size in the Suzuki-Trotter approximation and will be restricted to $p = 1$ in this thesis. The operator in eq. 4.58 can be implemented with Hamiltonian simulation (see section 4.2.5) by utilizing circuit 4.35. Since we are dealing with the pairing Hamiltonian (eq. 2.25) in this thesis, we can reduce the number of terms in our ansatz by not allowing to break particle pairs. The Taylor expansion of the UCCD operator gives us

$$\begin{aligned}
 e^{\sum_{ijab} t_{ij}^{ab}(a_a^\dagger a_b^\dagger a_j a_i - a_i^\dagger a_j^\dagger a_b a_a)} &= I + \sum_{ijab} t_{ij}^{ab}(a_a^\dagger a_b^\dagger a_j a_i - a_i^\dagger a_j^\dagger a_b a_a) \\
 &\quad + [\sum_{ijab} t_{ij}^{ab}(a_a^\dagger a_b^\dagger a_j a_i - a_i^\dagger a_j^\dagger a_b a_a)]^2 / 2! \\
 &\quad + \dots
 \end{aligned} \tag{4.59}$$

We can immediately see that the first sum over i, j, a, b will break pairs if we do not introduce a restriction on the sum. This can be explained by the fact that any non zero t_{ij}^{ab} will include the following term

$$t_{ij}^{ab}(a_a^\dagger a_b^\dagger a_j a_i - a_i^\dagger a_j^\dagger a_b a_a) |c\rangle,$$

which will break pairs if $j \neq i + 1$ and $b \neq a + 1$. Hence we make the restriction $j = i + 1$ and $b = a + 1$.

4.3.5 Simple ansatz for one pair and four spin-orbitals

For one pair and four spin-orbitals, one can set up a simple ansatz for the pairing model (eq. 2.25) with the following circuit

$$|\psi_T(\theta)\rangle = \text{Circuit with 4 qubits, } R_y(\theta) \text{ on qubit 1, } X \text{ on qubit 2, and } X \text{ on qubit 4, with CNOTs connecting them in sequence.} \tag{4.60}$$

where all qubits are initialized in the $|0\rangle$ state. To see that this ansatz conserves the particle number, we can write it out mathematically. First is the application of the $R_y(\theta)$ gate (see eq. 4.6). This gives us the state

$$(\cos \frac{\theta}{2} |0\rangle - i \sin \frac{\theta}{2} |1\rangle) |000\rangle.$$

The CNOT (see eq. 4.9) entangles the second qubit with the first, giving us

$$\cos \frac{\theta}{2} |0000\rangle - i \sin \frac{\theta}{2} |1100\rangle.$$

The subsequent gates flip both of the bottom qubits if the second qubit is in the $|0\rangle$ state. This results in the state

$$\cos \frac{\theta}{2} |0011\rangle - i \sin \frac{\theta}{2} |1100\rangle,$$

which we can see is a parametrized linear combination of the two allowed states.

4.3.6 Summary of the variational quantum eigensolver algorithm

We can summarize the variational quantum eigensolver algorithm as follows

- Apply a parametrized unitary operator (ansatz) $U(\theta)$ to an n -qubit state. This yields the state $|\psi(\theta)\rangle = U(\theta) |\psi\rangle$. Examples of ansatzes are given in sections 4.3.4, 4.3.3 and 4.3.5.
- Calculate the energy expectation value (eq. 4.40) for a Jordan-Wigner transformed (section 4.2.4) Hamiltonian. This is done by evaluating the expectation value of each separate term of the Hamiltonian, as described in section 4.3.2.
- Vary the parameters θ and do the same procedure till the energy expectation value is minimized.

4.4 Quantum Adiabatic Time Evolution

The adiabatic theorem states that if we start out in the ground state Ψ_0 of a Hamiltonian \hat{H}_0 and gradually change the Hamiltonian of the system to \hat{H}_1 , we will eventually end up in the ground state Ψ_1 of \hat{H}_1 given that the gradual change is small enough [16]. This provides the concept for adiabatic quantum computing, an alternative to the standard circuit model of quantum computing. Here we will, however, see how this theorem could be implemented on the standard circuit model to find the ground state energy of a fermionic Hamiltonian. We call this method the quantum adiabatic time evolution (QATE) algorithm. We start out with a mathematical formulation of the change from \hat{H}_0 to \hat{H}_1 . This can be written as a new time dependent Hamiltonian

$$\hat{H}(t) = (1 - \frac{t}{T})\hat{H}_0 + \frac{t}{T}\hat{H}_1, \quad (4.61)$$

with $t \in [0, T]$. Since we are now dealing with a time-dependent Hamiltonian, the time evolution operator is now given by the time-ordered exponential [25]

$$U(t) = e^{-i \int_0^t H(t) dt}. \quad (4.62)$$

We can approximate the integral by utilizing numerical integration. Breaking the time interval into n time steps and utilizing the midpoint rule [51] gives

$$\begin{aligned} \int_0^t H(t) dt &\approx \sum_{k=0}^n H(k\Delta t) \Delta t \\ &= \sum_{k=0}^n \left[(1 - \frac{k\Delta t}{T})\hat{H}_0 + \frac{k\Delta t}{T}\hat{H}_1 \right] \Delta t, \end{aligned} \quad (4.63)$$

with $\Delta t = \frac{T}{n}$. The approximation to the time-ordered exponential operator is then given by

$$U(t) = e^{-i \sum_{k=0}^n [(1 - \frac{k\Delta t}{T}) \hat{H}_0 + \frac{k\Delta t}{T} \hat{H}_1] \Delta t}. \quad (4.64)$$

Provided we utilize small time steps, we can use the Suzuki-Trotter approximation (section 4.2.3) to yield the following operator

$$U(t) \approx \prod_{k=0}^n e^{-i [(1 - \frac{k\Delta t}{T}) \hat{H}_0 + \frac{k\Delta t}{T} \hat{H}_1] \Delta t}, \quad (4.65)$$

which is effectively applied with the time evolution circuit 4.35.

When later using QATE to solve for the ground state energy of the pairing Hamiltonian (eq. 4.26), we will put the initial Hamiltonian to

$$\hat{H}_0 = \frac{1}{5} [\sum_{i=1}^{n_f} \sigma_z - \sum_{i=n_f+1}^{n_s} \sigma_z], \quad (4.66)$$

where n_f is the number of particles we wish to solve for and n_s is the number of spin-orbitals. This Hamiltonian is chosen because of its simplicity, only adding terms linear in the amount of qubits. It also conserves the particle number. The ground state of this Hamiltonian, which also will be our initial state for the QATE algorithm is

$$|\psi_0\rangle = |111 \cdots 1000 \cdots 0\rangle, \quad (4.67)$$

that is, the n_f last qubits are put to zero, while the rest are put to one.

4.5 Validating the results

When dealing with the VQE and the QATE algorithm, a natural question to ask is how we know that we have reached an eigenstate $|\psi_k\rangle$ of our Hamiltonian \hat{H} . When measuring the energy expectation value of our system, we know from quantum mechanics that the variance of the energy estimate should be zero when the measured state is an eigenstate of our Hamiltonian. The variance can be evaluated the following way

$$\sigma_E^2 = \langle \psi | \hat{H}^2 | \psi \rangle - \langle \psi | \hat{H} | \psi \rangle^2. \quad (4.68)$$

For the VQE algorithm, we could minimize the energy and finally evaluate eq. 4.68 to make sure it is below some threshold. For the QATE algorithm, we could measure the variance at each time step and stop the algorithm if the variance is below some threshold. Even though the variance tells us if we have reached an eigenstate or not, we can not be certain that we have not reached one of the excited states rather than the ground state.

CHAPTER 5

Quantum Computing: Machine Learning

Quantum computers may open up a world of new machine learning algorithms, or possibly improve upon already known methods. We will specifically look at neural networks implemented on a quantum computer and see how they potentially could be used to solve for the ground state energy of a quantum mechanical system. We will also see how parametrized quantum circuits (PQC) can be utilized to approximate unitary operators.

5.1 Amplitude Encoding

A crucial part of quantum machine learning algorithms is encoding a data set into our quantum state. There are several ways of doing this, but in this thesis we will only focus on storing our data set in the amplitudes of the quantum state. This is called amplitude encoding.

Amplitude encoding is about encoding a data set into the amplitudes of a quantum state:

$$|D\rangle = \sum_{m=1}^M \sum_{i=1}^N x_i^m |i\rangle |m\rangle, \quad (5.1)$$

where x_i^m denotes the i 'th predictor of the m 'th sample in our data set. Since n qubits can be put in a linear combinations of 2^n quantum states, we can store a data set of 2^n values on a quantum computer of n qubits. Hence we can see that the representation in eq. 5.1 provides an exponential advantage in terms of memory over classical computers [4].

Since the sum of the squared amplitudes of a quantum state needs to be equal to one in a quantum system, we assume that the data set is normalized, that is

$$x_i^m \rightarrow \frac{x_i^m}{\sqrt{\sum_{m=1}^M \sum_{i=1}^N (x_i^m)^2}}.$$

There are several different ways of achieving the state in eq. 5.1 and we will look at the one introduced by Möttönen *et al.* [33]. Since we will only encode a single sample at a time in this thesis, we use just one register for the amplitude encoding rather than the two in eq. 5.1. Suppose that we start with the ideal state

$$|D\rangle = \sum_{i=1}^M x_i |i\rangle, \quad (5.2)$$

where x_i is the i 'th predictor of our data sample. If we find a way to revert this state back to the zero-state $|0\dots 0\rangle$, the wanted state preparation can be done by inverting this operation. The reversion back to $|0\dots 0\rangle$ can be done by applying rotations that focus on doing this for one qubit at a time. Suppose that we need s qubits to encode our data. The idea is to first apply an $R_y(\theta)$ gate (eq. 4.6) on qubit q_s controlled by all the previous qubits q_1, \dots, q_{s-1} . Consider the following two terms of the amplitude encoded state in eq. 5.2:

$$x_{q_1, q_2, \dots, q_{s-1}, 0} |q_1, q_2, \dots, q_{s-1}, 0\rangle,$$

and

$$x_{q_1, q_2, \dots, q_{s-1}, 1} |q_1, q_2, \dots, q_{s-1}, 1\rangle.$$

Applying the gate

$$R_{y|c_1=q_1, c_2=q_2, \dots, c_{s-1}=q_{s-1}}^{c_s}(\theta),$$

which is an $R_y(\theta)$ -rotation (eq. 4.6) on qubit s , conditional on qubit i being in the q_i state for all $i \in \{1, 2, \dots, s-1\}$, gives us the following alteration

$$\begin{aligned} & R_{y|c_1=q_1, \dots, c_{s-1}=q_{s-1}}^{c_s}(\theta) [x_{q_1, q_2, \dots, q_{s-1}, 0} |q_1, q_2, \dots, q_{s-1}, 0\rangle \\ & \quad + x_{q_1, q_2, \dots, q_{s-1}, 1} |q_1, q_2, \dots, q_{s-1}, 1\rangle] \\ &= [\cos(\theta/2)x_{q_1, q_2, \dots, q_{s-1}, 0} - \sin(\theta/2)x_{q_1, q_2, \dots, q_{s-1}, 1}] |q_1, q_2, \dots, q_{s-1}, 0\rangle \\ & \quad + [\sin(\theta/2)x_{q_1, q_2, \dots, q_{s-1}, 0} + \cos(\theta/2)x_{q_1, q_2, \dots, q_{s-1}, 1}] |q_1, q_2, \dots, q_{s-1}, 1\rangle. \end{aligned} \quad (5.3)$$

Now we want to find the θ that results in a amplitude of zero for the $|q_1, q_2, \dots, q_{s-1}, 1\rangle$ state. There are different solutions depending on the value of the two respective data set values, $x_{q_1, q_2, \dots, q_{s-1}, 0}$ and $x_{q_1, q_2, \dots, q_{s-1}, 1}$.

First assume $x_{q_1, q_2, \dots, q_{s-1}, 0}, x_{q_1, q_2, \dots, q_{s-1}, 1} \neq 0$. If we want the amplitude of $|q_1, q_2, \dots, q_{s-1}, 1\rangle$ to be zero, it translates to

$$[\sin(\theta/2)x_{q_1, q_2, \dots, q_{s-1}, 0} + \cos(\theta/2)x_{q_1, q_2, \dots, q_{s-1}, 1}] = 0,$$

which can be written as

$$\frac{\sin(\theta/2)}{\cos(\theta/2)} = \tan(\theta/2) = -\frac{x_{q_1, q_2, \dots, q_{s-1}, 1}}{x_{q_1, q_2, \dots, q_{s-1}, 0}}.$$

Solving this for θ gives us

$$\theta = -2 \arctan\left(\frac{x_{q_1, q_2, \dots, q_{s-1}, 1}}{x_{q_1, q_2, \dots, q_{s-1}, 0}}\right). \quad (5.4)$$

Plugging θ back into eq. 5.3 shows that $|q_1, q_2, \dots, q_{s-1}, 1\rangle$ will now have an amplitude of 0, and the amplitude of $|q_1, q_2, \dots, q_{s-1}, 0\rangle$ will be

$$x_{q_1, q_2, \dots, q_{s-1}, 0} \sqrt{\frac{x_{q_1, q_2, \dots, q_{s-1}, 1}^2}{x_{q_1, q_2, \dots, q_{s-1}, 0}^2} + 1}. \quad (5.5)$$

Now if $x_{q_1, q_2, \dots, q_{s-1}, 0} = 0$ and $x_{q_1, q_2, \dots, q_{s-1}, 1} \neq 0$, the solution for θ in eq. 5.3 is

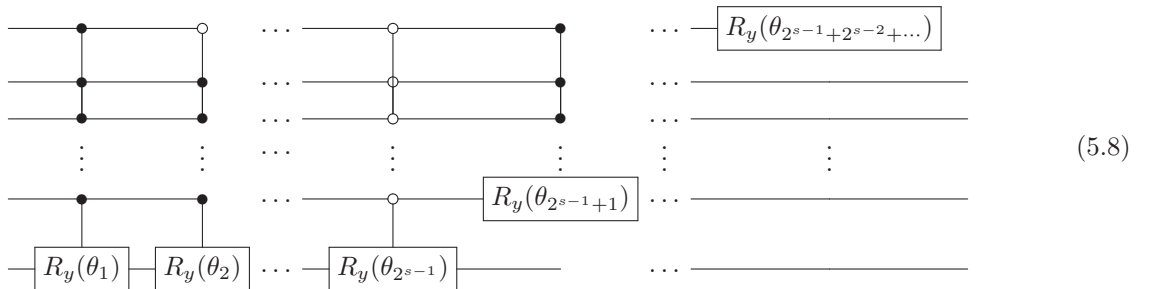
$$\theta = -\pi, \quad (5.6)$$

and the $|q_1, q_2, \dots, q_{s-1}, 0\rangle$ state will now have the amplitude

$$x_{q_1, q_2, \dots, q_{s-1}, 1}. \quad (5.7)$$

For the two final cases, that is $x_{q_1, q_2, \dots, q_{s-1}, 0}, x_{q_1, q_2, \dots, q_{s-1}, 1} = 0$ and the case $x_{q_1, q_2, \dots, q_{s-1}, 0} \neq 0$ and $x_{q_1, q_2, \dots, q_{s-1}, 1} = 0$, we simply do nothing.

If we now do these rotations on q_s for all relevant combinations of q_1, \dots, q_{s-1} , we can neglect qubit q_s as we know its value is deterministically 0. Then we do the same to q_{s-1} for all relevant combinations of q_1, \dots, q_{s-2} , remembering that we got the new amplitudes given by eq. 5.5 and 5.7. Following this procedure until the first qubit, for which we apply a non-controlled R_y gate, will result in the zero-state. The circuit for this procedure looks like this



where the inverse of this circuit is applied to go from the $|00000\dots\rangle$ state to the desired state

$$|D\rangle = \sum_{i=1}^M x_i |i\rangle.$$

The inverse is easily implemented by applying the rotations in reverse order with rotation angle $-\theta$ instead of θ .

To summarize, we have described a method that can be utilized to encode some data set \mathbf{x} of 2^M values into the amplitudes of an M -qubit quantum state, performing $\mathcal{O}(2^M)$ operations [33]. We denoted this transformation by the unitary operator $U_{\mathbf{x}}$, with the following action on the $|0\dots 0\rangle$ state

$$U_{\mathbf{x}} |00\dots 0\rangle = |\mathbf{x}\rangle = \sum_{i=1}^M x_i |i\rangle,$$

where x_i is the i 'th element of vector \mathbf{x} . We deduced how to implement this operator by considering that its complex conjugate, $U_{\mathbf{x}}^\dagger$, should revert the amplitude encoded state back to the zero-state

$$U_{\mathbf{x}}^\dagger \sum_{i=1}^M x_i |i\rangle = |0\dots 0\rangle.$$

We illustrated that this reversion could be done with the application of controlled $R_y(\theta)$ gates, where the rotation angle θ could be derived in terms of the amplitudes x_i .

5.2 Inner product

Several existing machine learning models rely on calculating the inner product between a vector of parameters \mathbf{w} , which are varied to fit the model to our data, and a vector \mathbf{x} containing our features. By looking at the machine learning models introduced in chapter 3, all of them contain at least one dependence on the calculation of an inner product. Hence, we will in this section focus on how an inner product can be estimated on quantum computers. Suppose we have utilized amplitude encoding (see circuit 5.8, section 5.1) to encode a feature vector \mathbf{x} consisting of p features into our quantum state. This operation can be represented with a unitary operator $U_{\mathbf{x}}$, which acts the following way on the qubits

$$U_{\mathbf{x}} |0\dots 0\rangle = \sum_{i=0}^{p-1} x_i |i\rangle = |\mathbf{x}\rangle.$$

Likewise for the parameter vector

$$U_{\mathbf{w}} |0\dots 0\rangle = \sum_{i=0}^{p-1} w_i |i\rangle = |\mathbf{w}\rangle.$$

Following Francesco Tacchino *et al.* [52], we can show that if we define

$$U_{\mathbf{w}}^\dagger U_{\mathbf{x}} |0\dots 0\rangle = \sum_{j=0}^{p-1} c_j |j\rangle = |\phi_{\mathbf{x},\mathbf{w}}\rangle, \quad (5.9)$$

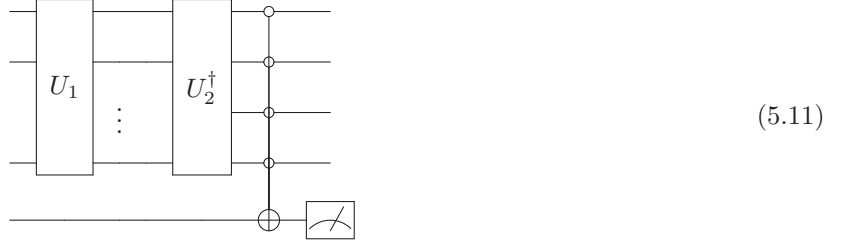
we find that

$$\langle \mathbf{w} | \mathbf{x} \rangle = \langle 0\dots 0 | U_{\mathbf{w}}^\dagger U_{\mathbf{x}} | 0\dots 0 \rangle = \langle 0\dots 0 | \sum_{j=0}^{p-1} c_j |j\rangle = \langle 0\dots 0 | c_0 | 0\dots 0 \rangle = c_0. \quad (5.10)$$

That is, the inner product $\mathbf{w} \cdot \mathbf{x}$ is contained in the amplitude c_0 of the $|0\dots 0\rangle$ term in eq. 5.9. If we introduce an ancilla qubit and flip it conditional on the qubits in the state $|\phi_{\mathbf{x},\mathbf{w}}\rangle = U_{\mathbf{w}}^\dagger U_{\mathbf{x}} | 0\dots 0 \rangle$ being zero, we end up with

$$|\phi_{\mathbf{x},\mathbf{w}}\rangle |0\rangle \rightarrow \sum_{j=1}^{p-1} c_j |j\rangle |0\rangle + c_0 |0\dots 0\rangle |1\rangle.$$

The probability of measuring the ancilla in the $|1\rangle$ state is then $|c_0|^2$ which is the squared inner product according to eq. 5.10. The circuit to perform the explained procedure is therefore given by:



with $U_1 = U_x$ and $U_2 = U_w$.

This circuit can be utilized to calculate the squared inner product between the state $U_1|0 \dots 0\rangle$ and the state $U_2|0 \dots 0\rangle$ for arbitrary U_1 and U_2 . This can be showed by following the same arguments as we did for U_x and U_w in this section, but for two arbitrary unitary operators. We will now go through how to set up a neural network by using this technique.

5.3 Dense Layer on a Quantum Computer

The dense layer we will explain in this section was first proposed by Francesco Tacchino *et al.* [52]. The activation of a neuron in a dense neural network layer is defined as (see section 3.4)

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_j w_{ji}^l a_j^{l-1} + b_i^l\right), \quad (5.12)$$

or

$$\mathbf{a}^l = f^l([\mathbf{W}^l]^T \mathbf{a}^{l-1} + \mathbf{b}^l),$$

where a_i^l is the activation of the i 'th node in the l 'th layer, $f^l(\cdot)$ is the non-linear activation function of the l 'th layer, w_{ji}^l are the model parameters in the l 'th layer and b_i^l is the i 'th bias of the l 'th layer. We see that the calculation of $f^l(z_i^l)$ is composed of the calculation of the inner product between the i 'th column of \mathbf{W}^l and the activation \mathbf{a}^{l-1} , with the addition of a bias. Denote $\mathbf{w}_b^{[l,i]}$ as the vector containing the i 'th column of \mathbf{W}^l , but with the bias b_i^l as its first element, that is

$$\mathbf{w}_b^{[l,i]} = [b_i^l, w_{0i}^l, w_{1i}^l, \dots, w_{[p-1]i}^l]. \quad (5.13)$$

Also denote \mathbf{a}_b^{l-1} as a vector containing the activations of layer $l-1$, but with a 1 added as the first element, that is

$$\mathbf{a}_b^{l-1} = [1, a_0^{l-1}, a_1^{l-1}, \dots, a_{p-1}^{l-1}]. \quad (5.14)$$

The inner product between these two vectors is then

$$\begin{aligned} \mathbf{w}_b^{[l,i]} \cdot \mathbf{a}_b^{l-1} &= [b_i^l, w_{0i}^l, w_{1i}^l, \dots, w_{[p-1]i}^l] \cdot [1, a_0^{l-1}, a_1^{l-1}, \dots, a_{p-1}^{l-1}] \\ &= b_i^l + w_{0i}^l a_0^{l-1} + w_{1i}^l a_1^{l-1} + \dots + w_{[p-1]i}^l a_{p-1}^{l-1} \\ &= \sum_j w_{ji}^l a_j^{l-1} + b_i^l = z_i^l, \end{aligned}$$

hence we have calculated z_i^l in eq. 5.12 as an inner product between two vectors. In the previous section we dealt with calculating the squared inner product on a quantum computer, so if we utilize the following function

$$f^l(z) = z^2,$$

as the activation function for our layers, we can utilize the squared inner product circuit (circuit 5.11) to realize a neural network on a quantum computer. The steps for implementing a dense neural network on a quantum computer are:

1. Set up circuit 5.11 with $U_1 = U_{\mathbf{a}_b^{l-1}}$ being the amplitude encoding (see section 5.1) of \mathbf{a}_b^{l-1} in eq. 5.14, and $U_2^\dagger = U_{\mathbf{w}_b^{[l,i]}}^\dagger$ being the amplitude encoding of $\mathbf{w}_b^{[l,i]}$ in eq. 5.13.
2. The probability of the ancilla being in the $|1\rangle$ state corresponds to the activation a_i^l .
3. Do this for every node i
4. Do this for every layer l

5.4 Recurrent Layer on a Quantum Computer

Inspired by the method found by Francesco Tacchino *et al.* [52], we can with the help of a few tricks propose a recurrent layer.

The activation for a simple recurrent neural network can be represented mathematically as (see section 3.5)

$$h_i^t = f(z_i^t) = f\left(\sum_j w_{ji}^x x_j^t + b_i^x + \sum_j w_{ji}^h h_j^{t-1} + b_i^h\right).$$

Consider that we denote

$$\begin{aligned}\mathbf{w}^{[x,i]} &= [b_i^x, w_{0i}^x, w_{1i}^x, \dots, w_{[p-1]i}^x], \\ \mathbf{x}_b^t &= [1, x_0^t, x_1^t, \dots, x_{p-1}^t], \\ \mathbf{w}^{[h,i]} &= [b_i^h, w_{0i}^h, w_{1i}^h, \dots, w_{[k-1]i}^h],\end{aligned}$$

and

$$\mathbf{h}_b^{t-1} = [1, h_0^{t-1}, h_1^{t-1}, \dots, h_{k-1}^{t-1}].$$

If we concatenate $\mathbf{w}^{[x,i]}$ with $\mathbf{w}^{[h,i]}$

$$\mathbf{u}_i = [b_i^x, w_{0i}^x, w_{1i}^x, \dots, w_{[p-1]i}^x, b_i^h, w_{0i}^h, w_{1i}^h, \dots, w_{[k-1]i}^h], \quad (5.15)$$

and do the same with \mathbf{x}_b^t and \mathbf{h}_b^{t-1}

$$\mathbf{y}^t = [1, x_0^t, x_1^t, \dots, x_{p-1}^t, 1, h_0^{t-1}, h_1^{t-1}, \dots, h_{k-1}^{t-1}], \quad (5.16)$$

the inner product between these two vectors are

$$\begin{aligned}\mathbf{u}_i \cdot \mathbf{y}^t &= b_i^x + w_{0i}^x x_0^t + w_{1i}^x x_1^t + \dots + w_{[p-1]i}^x x_{p-1}^t + b_i^h + w_{0i}^h h_0^{t-1} + w_{1i}^h h_1^{t-1} + \dots + w_{[k-1]i}^h h_{k-1}^{t-1} \\ &= \sum_j w_{ji}^x x_j^t + b_i^x + \sum_j w_{ji}^h h_j^{t-1} + b_i^h = z_i^t.\end{aligned}$$

Hence z_i^t can be calculated as an inner product between two vectors. Similar to what we explained in section 5.3, we can utilize the squared inner product circuit 5.11 to realize a recurrent layer with

$$f(z) = z^2,$$

as the activation function. The steps for implementing a recurrent neural network on a quantum computer goes as follows:

1. Set up circuit 5.11 with $U_1 = U_{\mathbf{y}^t}$ being the amplitude encoding (see section 5.1) of \mathbf{y}^t in eq. 5.16 and $U_2 = U_{\mathbf{u}_i}^\dagger$ being the amplitude encoding of \mathbf{u}_i in eq. 5.15 .
2. The probability of the ancilla being in the $|1\rangle$ state corresponds to the activation h_i^t .
3. Do this for every node i
4. Do this for every time step t

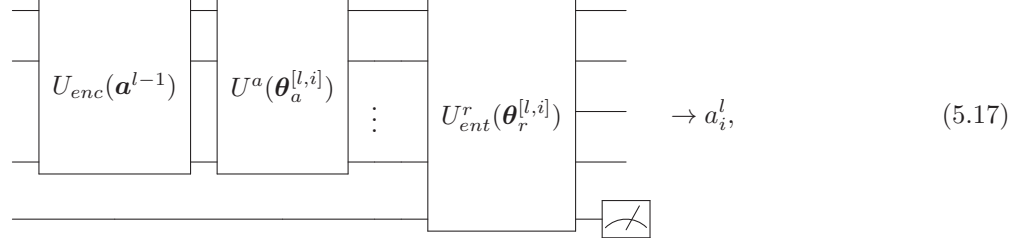
5.5 PQC Dense Layer

Utilizing parametrized quantum circuits (PQC) as machine learning models is an interesting subject that is currently under a lot of research. Marcello Benedetti *et al.* [4] has the following to say in their article concerning utilizing parametrized (variational) circuits as machine learning models:

Similar to the universal approximation theorem in neural networks, there always exists a quantum circuit that can represent a target function within an arbitrary small error. The caveat is that such a circuit may be exponentially deep and therefore impractical. Lin et al. [28] argue that since real datasets arise from physical systems, they exhibit symmetry and locality; this suggests that it is possible to use 'cheap' models, rather than exponentially costly ones, and still obtain a satisfactory result. With this in mind, the variational circuit aims to implement a function that can approximate the task at hand while remaining scalable in the number of parameters and depth. In practice, the circuit design follows a fixed structure of gates. Despite the dimension of the vector space growing exponentially with the number of qubits, the fixed structure reduces the model complexity, resulting in the number of free parameters to scale as a polynomial of the qubit count.

The dense layer and recurrent layer in sections 5.3 and 5.4, respectively, are quantum copies of classical neural networks with the square function as an activation function. For an n -qubit layer, they rely on amplitude encoding 2^n inputs as well as 2^n free parameters; a task that is exponential in the number of qubits (see section 5.1). Even though we achieve an exponential advantage in terms of memory when having the data encoded in the amplitudes of the quantum state, the exponential number of operations required may be impractical, as explained by Marcello Benedetti *et al.*

Neural network layers with a number of free parameters and a number of operations that scale polynomially in the number of qubits are crucial for the implementation on noisy intermediate scale quantum (NISQ) devices. An exponential scaling results in circuit depths (see section 4.1.3 for a brief explanation of this quantity) that are too deep to be practical. We can propose a dense neural network layer architecture that is not reliant on specifically utilizing amplitude encoding, but rather an arbitrary operator that may be hardware-efficient. We propose to calculate the i 'th activation of the l 'th layer in a neural network with the following circuit:



The operator U_{enc} is responsible for encoding some representation of the neural network inputs \mathbf{a}^{l-1} into a quantum register. This operator will be referred to as our encoder and the qubit-register it is acting on will be called the encoder-register. An example of an encoder is the amplitude encoding shown in circuit 5.8, which has the following action on the encoder-register

$$U_{enc}(\mathbf{a}^{l-1}) |0 \cdots 0\rangle = \sum_i a_i^{l-1} |i\rangle.$$

The operator U^a will be referred to as the ansatz. The purpose of the ansatz is to rotate the encoded state $U_{enc}(\mathbf{a}^{l-1}) |0 \cdots 0\rangle$ into a state dependent on the neural network parameters, that is

$$|\psi(\theta_a^{[l,i]}, \mathbf{a}^{l-1})\rangle = U^a(\theta_a^{[l,i]}) U_{enc}(\mathbf{a}^{l-1}) |0 \cdots 0\rangle.$$

The final operator, U_{ent}^r , we will refer to as the entangler and it is responsible for entangling the qubits in the encoder-register with an ancilla qubit:

$$\begin{aligned} U_{ent}^r |0\rangle (\theta_r^{[l,i]}) |\psi(\theta_a^{[l,i]}, \mathbf{a}^{l-1})\rangle &= c_0(\theta_a^{[l,i]}, \theta_r^{[l,i]}, \mathbf{a}^{l-1}) |\psi_0(\theta_a^{[l,i]}, \mathbf{a}^{l-1})\rangle |0\rangle \\ &\quad + c_1(\theta_a^{[l,i]}, \theta_r^{[l,i]}, \mathbf{a}^{l-1}) |\psi_1(\theta_a^{[l,i]}, \mathbf{a}^{l-1})\rangle |1\rangle, \end{aligned}$$

where c_0 and c_1 are some parametrized complex number. The probability of measuring the ancilla in the $|1\rangle$ state

$$a_i^l = |c_1(\theta_a^{[l,i]}, \theta_r^{[l,i]}, \mathbf{a}^{l-1})|^2, \quad (5.18)$$

is then used as the i 'th activation in the l 'th layer. The activation of the neural network layer is hence dependent on the inputs, as well as some model parameters.

The calculation of all the m activations in a layer is done by calculating eq. 5.18 for $i \in \{1, 2, \dots, m\}$ with circuit 5.17. The next layer can then be evaluated by utilizing the calculated activations \mathbf{a}^l as inputs to the encoder for the next layer, $U_{enc}(\mathbf{a}^l)$, and follow the same procedure.

We will soon provide a description of some encoders, ansatzes and entanglers. First we will explain how layers without amplitude encoding can be realized, and also how to set up a recurrent neural network with circuit 5.17.

5.5.1 Layers without amplitude encoding

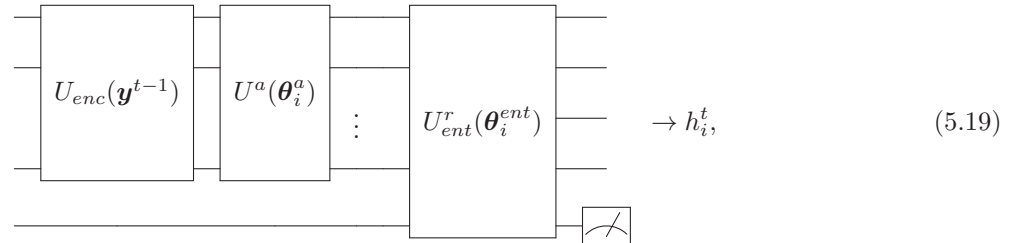
When not utilizing amplitude encoding as the encoder in circuit 5.17, we need to find out how else to encode the inputs. By looking at the amplitude encoding circuit (circuit 5.8), we see that we are in reality just applying rotations to our qubits, which are conveniently calculated beforehand with the help of the relevant inputs (eqs. 5.4 and 5.6). With this in mind, we propose that we could utilize any parametrized unitary operator $U_{enc}(\theta)$ dependent on rotation angles θ as our encoder. The inputs θ to this encoder are the outputs from the previous layer \mathbf{a}^{l-1} , but scaled so they lie in the range $[0, 2\pi]$.

5.6 PQC Recurrent Layer

By concatenating the input vector and the hidden vector like we did in section 5.4, we can also express a parametrized quantum circuit for a recurrent layer. Denote the concatenated vector with

$$\mathbf{y}^{t-1} = [x_0^t, x_1^t, \dots, x_{p-1}^t, h_0^{t-1}, h_1^{t-1}, \dots, h_{m-1}^{t-1}],$$

where p is the number of predictors in our data set and m is the number of dimensions of the hidden vector. The recurrent layer is realized with the following circuit



For an explanation of the operators in this circuit, see section 5.5. The probability of the ancilla (bottom) qubit being in the $|1\rangle$ -state, denoted $|c_1(\theta_i^a, \theta_i^{ent}, \mathbf{y}^{t-1})|^2$, is the i 'th element of the hidden vector at time step t , that is

$$h_i^t = |c_1(\theta_i^a, \theta_i^{ent}, \mathbf{y}^{t-1})|^2. \quad (5.20)$$

Once this quantity is calculated for all $i \in \{1, 2, \dots, m\}$, we can calculate the next time step h_i^{t+1} by denoting

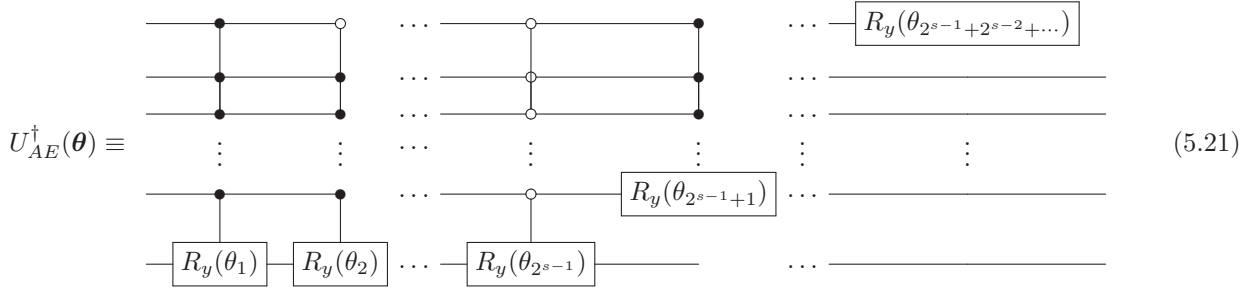
$$\mathbf{y}^t = [x_0^{t+1}, x_1^{t+1}, \dots, x_{p-1}^{t+1}, h_0^t, h_1^t, \dots, h_{m-1}^t],$$

and use this vector as the input to the encoder U_{enc} for the next layer.

5.7 The encoders and ansatzes: U_{enc} and U_a

The PQC neural network layers in section 5.5 and 5.6 rely on encoding the inputs with an operator U_{enc} acting on the encoder-register, and then acting on the same register with an ansatz U_a . Here we will list the encoders

and ansatzes utilized in this thesis. The first is the amplitude encoding circuit explained in section 5.1:



$$U_{AE}^\dagger(\theta) \equiv \dots \quad (5.21)$$

where the rotation angles are given by eqs. 5.5 and 5.7. The action of this circuit is to encode a vector \mathbf{x} into the amplitudes of a quantum state

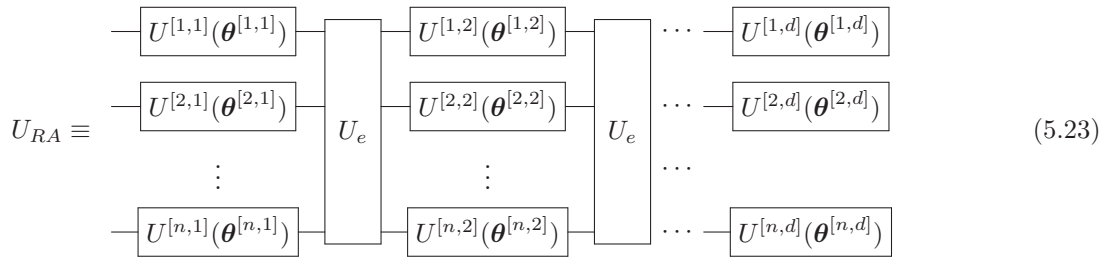
$$U_{AE} |0 \dots 0\rangle = \sum_i^p x_p |i\rangle.$$

When utilizing the amplitude encoder as an ansatz, the number of parameters required for a layer of k nodes and n qubits are

$$\text{Number of parameters: } k2^n \quad (5.22)$$

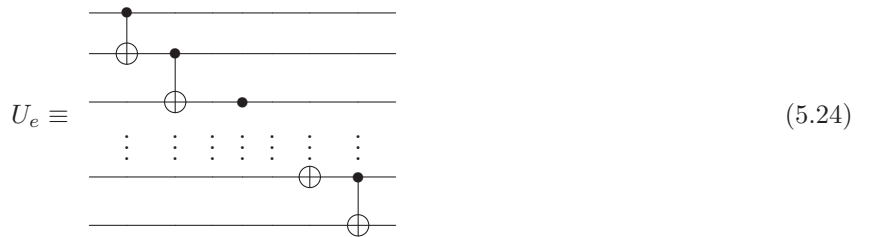
The caveats when utilizing this encoder is the exponential number of operations and free parameters, as we explained in section 5.5.

The other encoder/ansatz we will utilize is the Euler rotation explained in section 4.3.3. It is given by the following circuit



$$U_{RA} \equiv \dots \quad (5.23)$$

where n is the number of qubits and d is the number of times we repeat the U_e operator. The operator U_e is responsible for entangling all the encoder qubits. In this thesis we will only choose U_e as the following operator



$$U_e \equiv \dots \quad (5.24)$$

and we put $U^{[j,k]}$ to either

$$U^{[j,k]}(\theta^{[j,k]}) = R_y(\theta^{[j,k]}), \quad (5.25)$$

or

$$U^{[j,k]}(\theta^{[j,k]}) = R_z(\theta_1^{[j,k]})R_x(\theta_2^{[j,k]})R_z(\theta_3^{[j,k]}), \quad (5.26)$$

where $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$ are given by eq. 4.6. Unlike the amplitude encoding circuit 5.21, this circuit is known to be both hardware-efficient and suitable for near-term quantum devices [32]. We can also choose the parameter d to be large to increase the flexibility of each layer, or it may be chosen small to act as a regularization to avoid our neural network fitting to eventual noise in the data set.

When utilizing circuit 5.23 as the ansatz, a layer with k activations and n qubits requires at most the following number of free parameters

$$\text{Number of parameters: } 3dkn, \quad (5.27)$$

where d is the number of times we apply U_e in circuit 5.23. Hence, we see that we achieve an exponential advantage in the number of parameters over the amplitude encoding (see eq. 5.22), as long as d is not exponential in the number of qubits.

5.8 The entanglers U_{ent}

The neural network circuits in sections 5.5 and 5.6 rely on some entangler U_{ent} causing entanglement between the encoder register and an ancilla qubit. We will utilize two different entanglers in this thesis. The first one is

$$U_{ent}^{MC} \equiv \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \boxed{U_b(\theta_i^b)} \oplus \end{array} \quad (5.28)$$

where the bottom qubit is the ancilla, while the rest of the qubits are in the encoder-register. The operator $U_b(\theta_i^b)$ is a single-qubit rotation responsible for adding the bias of the i 'th node. Looking at the squared inner product circuit 5.11 and the PQC neural network circuit 5.17, we see that the utilization of the above circuit as the entangler, make our activations dependent on the inner product between the state produced by the encoder $U_{enc}|0 \cdots 0\rangle$ and the state produced by the ansatz $U^a|0 \cdots 0\rangle$.

The number of free parameters required for a layer of k nodes and n qubits with this entangler are

$$\text{Number of parameters: } k. \quad (5.29)$$

The second variation of U_{ent} allows for the parallel calculation of nodes. It is given by the following circuit

$$U_{RE} \equiv \begin{array}{c} \text{---} \\ \text{---} \\ \vdots \\ \text{---} \\ \vdots \\ \text{---} \end{array} \begin{array}{c} \bullet \\ \bullet \\ \vdots \\ \bullet \end{array} \begin{array}{c} \text{---} \\ \text{---} \\ \vdots \\ \text{---} \end{array}, \quad (5.30)$$

$$\text{---} \boxed{U_b(\theta_b^i)} \text{---} \boxed{U^r(\theta_1^{[i,r]})} \text{---} \boxed{U^r(\theta_2^{[i,r]})} \cdots \boxed{U^r(\theta_n^{[i,r]})} \text{---}$$

where $U^r(\theta_k^{[i,r]})$ is some single-qubit rotation and $U_b(\theta_b^i)$ is a single-qubit rotation adding bias to the activation. We can choose to skip all but the final conditional rotation if wanted. We will later show that this entangler allows for parallel calculation of nodes.

The number of free parameters required for a layer of k nodes and n qubits with this entangler are

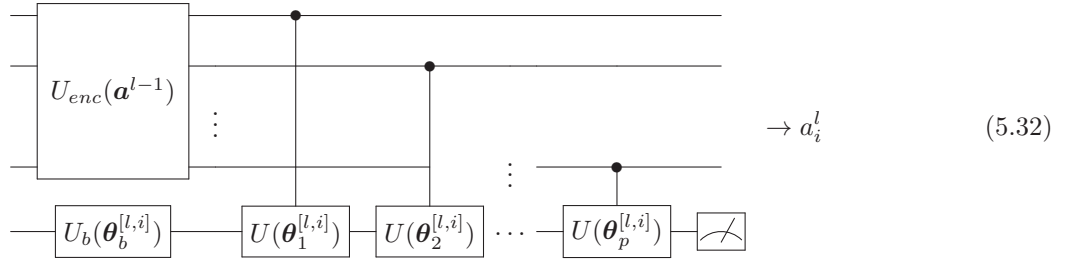
$$\text{Number of parameters: } kn. \quad (5.31)$$

For both of these entanglers, we will only utilize the $R_y(\theta)$ gate (eq. 4.6) for the single-qubit rotations.

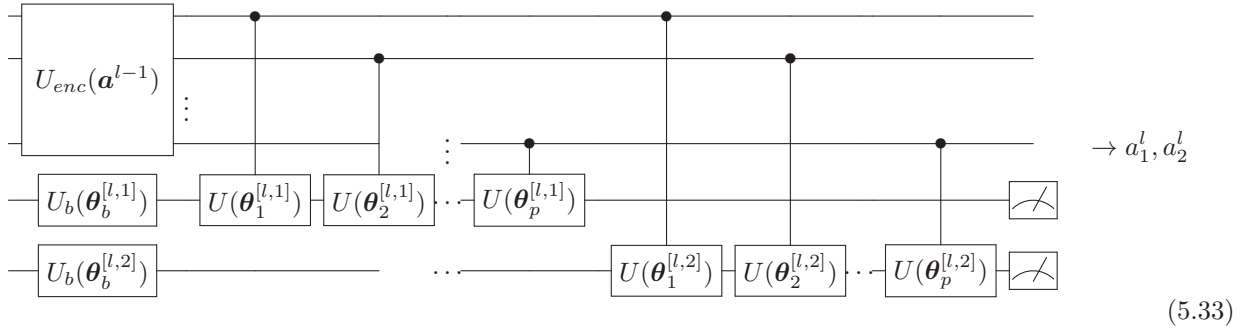
5.9 Parallel calculation of nodes

Suppose we utilize the general dense layer in circuit 5.17, where U_a is the identity operator for simplicity. Utilizing the U_{RE} operator represented in circuit 5.30 as the entangler, one could calculate several nodes in

parallel. A single node is calculated with the following circuit



while the calculation of additional nodes is simply done by utilizing a second ancilla and a number of additional gates linear in the number of qubits:

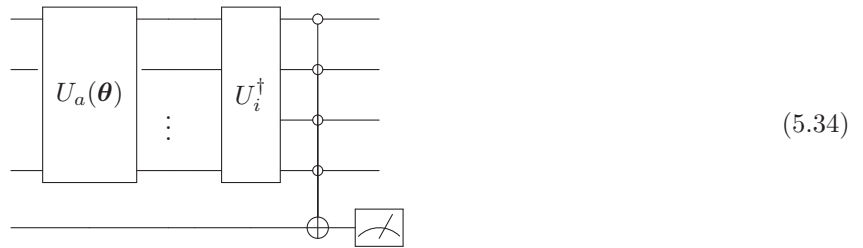


This method is easily extended to calculate an arbitrary number of nodes simultaneously. We require one extra ancilla qubit per activation as well as a number of gates linear in the number of qubits.

5.10 Learning Unitary Operators

Suppose you have a unitary operator U_i and wish to approximate it with another, less complex operator. We may want to do this to reduce the circuit depth of U_i , as this quantity is often the bottleneck on near-term quantum devices. We explained this in section 4.1.3.

We can try to approximate the action of U_i by learning an ansatz $U_a(\theta)$. Recall from section 5.2 that the squared inner product between the state $U_i |0 \cdots 0\rangle$ and the state $U_a(\theta) |0 \cdots 0\rangle$ can be calculated by utilizing the following circuit



where the squared inner product is equal to the probability of the bottom (ancilla) qubit being in the $|1\rangle$ state. Given two complex vectors \mathbf{a} and \mathbf{b} , we have the following relation between their inner product and the angle α between these vectors [20]

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \alpha. \quad (5.35)$$

As our quantum states are of unit length, we can see that their inner product is simply equal to the cosine of the angle between them. If two vectors of unit length are parallel, that is, their squared inner product is equal to one, we can conclude that the vectors are equal except for some global phase. From quantum mechanics, we know that states differing by a global phase are physically indistinguishable [2]. Hence, if we can vary the parameters of $U_a(\theta)$ until the squared inner product in circuit 5.34 is equal to one, we have

$$U_a(\theta) = cU_i,$$

where all $U_a^j(\theta^j)$ are learned by minimizing (maximizing) the probability of the ancilla being in the $|0\rangle$ ($|1\rangle$) state. Since every step of the recursive algorithm only yields an approximation to the operators (due to finite measurements of the squared inner product), we expect that some error will propagate as we perform the steps.

5.12 Eigenvalues with Neural Networks

Solving for the ground state energy of a time independent quantum system reduces to finding the smallest eigenvalue and eigenvector of a hermitian matrix. We will now look at one way to approximate the eigenvectors and eigenvalues by utilizing neural networks.

5.12.1 Rayleigh Quotient Minimization [13]

The Rayleigh quotient is given by

$$R(\mathbf{x}) = \frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}}, \quad (5.38)$$

where $\mathbf{x} \in \mathcal{R}^n$ is a vector of unit length and $A \in \mathcal{R}^{n \times n}$ is the matrix we wish to solve the eigenvalues for. Let us write \mathbf{x} as a linear combination of the n orthogonal eigenvectors \mathbf{v}_i belonging to A :

$$\mathbf{x} = \sum_{i=1}^n c_i \mathbf{v}_i,$$

where c_i is assumed to be some real constant. The Rayleigh quotient then becomes

$$\frac{\sum_{i=1}^n c_i \mathbf{v}_i^T A \sum_{j=1}^n c_j \mathbf{v}_j}{\sum_{i=1}^n c_i \mathbf{v}_i^T \sum_{j=1}^n c_j \mathbf{v}_j} = \frac{\sum_{ij} c_i c_j \mathbf{v}_i^T A \mathbf{v}_j}{\sum_{ij} c_i c_j \mathbf{v}_i^T \mathbf{v}_j}.$$

Since all the eigenvectors \mathbf{v}_i are orthogonal, that is

$$\mathbf{v}_i^T \mathbf{v}_j = \delta_{ij},$$

and because

$$A \mathbf{v}_j = E_j \mathbf{v}_j,$$

where E_j is the j 'th eigenvalue of A , we get

$$\frac{\sum_{ij} c_i c_j \mathbf{v}_i^T A \mathbf{v}_j}{\sum_{ij} c_i c_j \mathbf{v}_i^T \mathbf{v}_j} = \frac{\sum_{i=1}^n c_i^2 E_i}{\sum_{i=1}^n c_i^2} = \sum_{i=1}^n c_i^2 E_i.$$

If we assume that $E_1 < E_2 < E_3, \dots$, we see that the minima of the Rayleigh quotient is given by

$$\mathbf{x} = \mathbf{v}_1.$$

The Rayleigh Quotient is then

$$R(\mathbf{v}_1) = E_1.$$

Hence, we have found the lowest eigenvalue of A . Consider a neural network (section 7.4.6) with output $\mathbf{a}^L \in \mathcal{R}^n$. When this output is fed into the Rayleigh quotient, we know that the minima is the lowest eigenvalue of A . Hence, we can use the Rayleigh quotient as the loss function for the training of the model:

$$L(\mathbf{a}^L; A) = \frac{(\mathbf{a}^L)^T A \mathbf{a}^L}{(\mathbf{a}^L)^T \mathbf{a}^L}. \quad (5.39)$$

PART III

Implementation

CHAPTER 6

Methods

In this chapter, we will go through the implementation of the algorithms described in this thesis. All the algorithms are made to be general-purpose. We hope that one is able to utilize any of the mentioned many-body methods for an arbitrary Jordan-Wigner transformed Hamiltonian (see section 4.2.4), and also generate and train an arbitrary PQC neural network (see section 5.5) after reading through this section. We will start by giving an introduction to Qiskit [42], the Python-package utilized to write our code for quantum computers. Then we will go through each of the quantum algorithms explained in this thesis.

All the text referring to some Python object or to some part of the provided code will be written in **bold**.

The mentioned scripts and functionality can be found on our GitHub page <https://github.com/stiandb/Thesis>. Doc-strings are written for all the functions and classes to explain all their inputs and outputs.

NB. Qiskit version 0.11.1 was utilized and the functionality may or may not have changed

6.1 Qiskit

Qiskit [42] is a convenient python package made by IBM and can be used to write code for quantum computers. This package allows one to either run the code on a simulation of a quantum computer on your own machine, or to actually run the code on a real quantum computer over the cloud. Installing Qiskit can be done via pip with the following command:

```
pip install qiskit
```

and the package is imported to a python program with

```
import qiskit as qk
```

6.1.1 Circuits and Registers

The utilization of Qiskit starts with setting up registers of bits and a circuit. Consider the following code

```
1 qubit_register_1 = qk.QuantumRegister(4)
2 qubit_register_2 = qk.QuantumRegister(2)
3 classical_register = qk.ClassicalRegister(2)
4 circuit = qk.QuantumCircuit(qubit_register_1, qubit_register_2, classical_register)
```

In the first and second line, we set up a four and two-qubit register, respectively. When we measure one or more of the qubits, we want to save the information to a classical register, which is what we initialize in line 3. The number of bits in the classical register should correspond to the number of qubits we measure at the end of the circuit. These three registers are put into a circuit in line 4. We could have included more or less quantum registers as arguments, as long as the classical register is the final one.

6.1.2 Gates

After we have set up a circuit, we are ready to apply gates to the qubits. The gates are methods of the `QuantumCircuit` object and a full list of them can be found in the Qiskit documentation: <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html>. As an example, let's apply the $R_y(\frac{1}{2})$ gate (eq. 4.6) to the second qubit in the `qubit_register_1` register. Then we perform a CNOT (eq. 4.9) to the first qubit in the `qubit_register_2` register, conditional on the second qubit in the `qubit_register_1` register:

```
circuit.ry(1/2,qubit_register_1[1])
circuit.cx(qubit_register_1[1],qubit_register_2[0])
```

To perform a measurement on a qubit register, we need to apply the measurement gate:

```
circuit.measure(qubit_register_2,classical_register)
```

Here we utilize the measurement gate on `qubit_register_2` and specify that we want the results saved to the classical register.

6.1.3 Execute circuit

The execution of a circuit and the retrieval of the results are done the following way:

```
1 job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'), shots=1000)
2 result = job.result().get_counts(circuit)
3 print(result)
```

```
{'00': 945, '01': 55}
```

Other than feeding `circuit` into the Qiskit `execute` function in the first line, we also have to type in some more arguments. The `backend` argument specifies where to run the experiment. In our case we use `qk.Aer.get_backend('qasm_simulator')`, that is, a simulation of a quantum computer on our own hardware. By changing the `backend` argument, one could run the experiment on for example a real quantum computer. The `shots` argument specifies how many times one wishes to run and measure the circuit, in this case 1000. In the second line we retrieve the results as a dictionary. The keys of this dictionary are strings showing the state of each of the qubits. The strings are in the opposite order of the indexes in the measured qubit-register, which is why the second element of the string (rather than the first) is the only qubit we measure in the $|1\rangle$ state. The values of the dictionary tells us how many times we measure said state. To get more familiar with Qiskit, let us go through an example problem.

6.1.4 Coin Toss Example

We will now go through how to set up the circuit for the coin tossing example in section 4.1.3 step by step. First, we initialize the circuit in the following way:

```
simulation_register = qk.QuantumRegister(4)
ancilla_register = qk.QuantumRegister(1)
classical_register = qk.ClassicalRegister(1)
circuit = qk.QuantumCircuit(simulation_register,ancilla_register,classical_register)
```

The first line creates the register of qubits representing our four coin tosses. The second line creates a second quantum register of one qubit, which is our ancilla. Since we only wish to measure one qubit (the ancilla), we make a classical register with one bit in the third line. In the final line, we put these registers together to define a circuit. Now we are ready to apply the qubit operations. The first step is to apply Hadamard gates (eq. 4.5) to the four simulation qubits:

```
for i in range(4):
    circuit.h(simulation_register[i])
```


The next step is to apply a multi-controlled X -gate (eq. 4.4) to the ancilla qubit, conditional on the coin toss qubits. With Qiskit, this is done with a multi-controlled R_x gate, with a rotation angle of π , as one can see from eq. 4.6 that $R_x(\pi) = -iX$:

```
circuit.mcrx(np.pi,[simulation_register[i] for i in range(4)],ancilla_register[0])
```

The first argument specifies the rotation angle, π . The second argument should contain a list of all the control-qubits, which in our case is all the simulation qubits. The third and final argument is the target qubit, which in our case is the one ancilla qubit. Now that the circuit is ready, we need to perform measurements of the ancilla qubit:

```
1 circuit.measure(ancilla_register,classical_register)
2 job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'), shots=10000000)
3 result = job.result().get_counts(circuit)
4 print(result)
```

```
{'0': 9374491, '1': 625509}
```

We see that we measured the ancilla in the $|0\rangle$ state 9374491 times and in the $|1\rangle$ state 625509 times. The approximated probability of getting a heads four times in a row is then

$$\frac{625509}{10000000} = 0.0625509 \approx \frac{1}{16} = 0.0625,$$

which is close to the analytical solution of $\frac{1}{16}$.

6.1.5 Simulating real devices

Qiskit has the ability to simulate models of IBM's real quantum computers, allowing one to get an indication for how an algorithm will fare on a real device. The task of simulating a real quantum computer consists of three main components which will be discussed now.

Noise Model

The noise model is the first component which consists of several factors that will affect the circuit execution. These factors are

- **Decoherence:** The error associated with the interactions from the environment on the quantum system.
- **Gate error:** The error associated with the application of gates on qubits
- **Gate length:** The error associated with the length of the applied gate.
- **Readout Error:** The error associated with the measurement of qubits.

Basis Gates

Different devices have their own set of basis gates that are applicable to the qubits. These gates can in turn be put together to generate any unitary operation. As no gates outside this set can be utilized, we need to obtain the gates specific for the computer we wish to simulate.

Coupling Map

In current multi-qubit quantum computers we rarely have a connection between all the qubit, so we cannot entangle every single qubit. This means that performing multi-qubit gates like for example the CNOT (eq. 4.9), may require swapping the locations of the qubits with the SWAP gate (eq. 4.10). A coupling map is a graph showing which qubits are connected and can be retrieved just like the noise model and basis gate set.

Performing the Simulation

In order to run a simulation of a real device, we need to retrieve the corresponding noise model, basis gates and coupling map. In the `settings.py` script, found in the GitHub linked in the introduction to this chapter, we have these ready for the IBMQ London 5 qubit computer, IBMQ X2 5 bit computer and the IBMQ Melbourne 16 qubit computer:

```
import qiskit as qk
from settings import ibmq_london_noise_model as noise_model
from settings import ibmq_london_basis_gates as basis_gates
from settings import ibmq_london_coupling_map as coupling_map
```

Given that one has generated a Qiskit circuit ready for execution, one can perform the noisy simulation the following way:

```
job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'),shots=1000,noise_model=noise_model,
    basis_gates=basis_gates,coupling_map=coupling_map)
```

Hence we only need to input the noise model, basis gates and coupling map to the Qiskit execute function.

None of the circuits in this thesis are made with a specific hardware in mind. In order to make these executable on a real device, one would have to make a basis change and also make sure the multi-qubit gates are compatible with the coupling map. Fortunately, Qiskit has a function that can do this automatically, while also optimizing the circuit to reduce the number of gates and circuit depth. We will now explain how to set this up.

6.1.6 Transpiler

The Transpiler allows one to make a circuit executable with a given gate basis and coupling map, while also optimizing the circuit depth. Given that we have a circuit ready for execution, we can transpile it the following way:

```
backend=qk.Aer.get_backend('qasm_simulator')
circuit = qk.compiler.transpile(circuit,backend=backend,backend_properties=backend.properties(),
    optimization_level=0,basis_gates=basis_gates,coupling_map=coupling_map)
```

We utilize the `Qiskit.compiler.transpile` function and feed in the basis gates and coupling map in the arguments. The parameter called `optimization_level` specifies how heavy optimization you wish to use, with 0 being no optimization and 3 being the heaviest.

6.1.7 Quantum Error Correction

Even with heavy optimization with the transpiler, one may not be able to achieve good results with the current noisy devices. Quantum error correction aims to protect quantum states from unwanted interactions such as environmental noise [7]. All the python methods in this thesis that are measuring a circuit has an argument in the initialization called `error_mitigator`, which can be set to perform error reduction after measurements. We have also put Qiskit's own error reduction algorithm into a class `ErrorMitigation`, which can be initialized and used as the `error_mitigator` argument. See the example below:

```
import qiskit as qk
from utils import ErrorMitigation
from example import example_method

results = example_method(1,1,noise_model=noise_model,basis_gates=basis_gates,coupling_map=coupling_map,
    error_mitigator=ErrorMitigation() )
```

Having the error reduction method as an input may allow for testing of other error correction schemes down the line.

6.2 Hamiltonian Simulation and Quantum Phase Estimation

We will now go through how to set up and utilize the quantum phase estimation (QPE) algorithm to find the eigenvalues of a Hamiltonian, as explained in section 4.2. The quantum Fourier transformation (QFT) introduced in section 4.2.1 is part of this algorithm. Hence, it is natural to start explaining how to utilize the QFT functionality.

6.2.1 Quantum Fourier Transformation

The QFT algorithm is part of the utilities in the `utils.py` script (found in the GitHub link in the chapter introduction), since it is commonly utilized as a subroutine in quantum circuits. To do a QFT, we simply import the functionality, set up the circuit and registers with Qiskit, and then call on the **QFT** function:

```
1 from utils import QFT
2 import qiskit as qk
3
4 qft_register = qk.QuantumRegister(4)
5 classical_register = qk.QuantumRegister(4)
6 circuit = qk.QuantumCircuit(qft_register, classical_register)
7 registers = [qft_register, classical_register]
8
9 circuit, registers = QFT(circuit=circuit, registers=registers, inverse=False)
```

The registers are put in a list in line 7 and the **QFT** function will perform the transformation on the register that is the first element in this list. In line 9, we call on the **QFT** function. The argument called **inverse** can be set to **True** to instead perform the inverse Fourier transform.

6.2.2 QPE function

A function, **QPE**, is also included in the `utils.py` script as it also may be utilized as a sub routine in algorithms. The QPE circuit in fig. 4.3 can be run by utilizing this function. See the script below:

```
1 from utils import QPE, ControlledTimeEvolutionOperator
2 import qiskit as qk
3
4 hamiltonian_list = [[0.25, [0, 'z']], [0.25, [1, 'z']], [-0.5]]
5 U = ControlledTimeEvolutionOperator(hamiltonian_list, dt=0.001, T=1)
6
7 t_register = qk.QuantumRegister(4)
8 u_register = qk.QuantumRegister(2)
9 ancilla_register = qk.QuantumRegister(1)
10 classical_register = qk.QuantumRegister(4)
11 circuit = qk.QuantumCircuit(t_register, u_register, ancilla_register, classical_register)
12 registers = [t_register, u_register, ancilla_register, classical_register]
13
14 circuit, registers = QPE(circuit=circuit, registers=registers, U=U)
```

We will explain what is done in lines 4 and 5 in the next section. What is important to know is what to input in the function call in line 14. The **circuit** argument should be the Qiskit `QuantumCircuit` instance. The **registers** argument is a list with the *t*-register and *u*-register (see section 4.2) as its first and second element, respectively. The argument **U** should be a callable class or function **U(circuit, registers, control, power)** that applies a controlled U^{power} operation (see QPE circuit in figure 4.3). The U^{power} operation should be done conditional on **registers[0][control]**, and be applied to the relevant qubits in **registers[1]**. The function **U** should return **circuit** as well as the **registers** list.

6.2.3 Time Evolution Operator

In order to calculate the eigenvalue of a Hamiltonian with QPE (section 4.2) and QATE (section 4.4), we need a class that can apply the time evolution operator (circuit 4.35) for any Hamiltonian written with in the Jordan-Wigner transformation (section 4.2.4). In this thesis, all the many-body algorithms assume we have

6. Methods

written the Hamiltonian in a specific format. The separate terms of the Hamiltonian should be contained in a list consisting of the relevant Pauli-matrices (eq. 4.4) and which qubits they act on. An example of such a list is this one:

```
[[3,[0,'x']],[-2,[5,'y'],[3,'z']],[2]].
```

The length of this list is the amount of Hamiltonian terms, so we are here dealing with a Hamiltonian with three terms. The first element of each nested list is the factor, while the consecutive elements are nested lists with the qubit in question as the first element and the gate to apply to this qubit as the second element. A nested list containing only a float represents just an application of the identity operator with a factor. The list provided in the above example represents the following Hamiltonian

$$\hat{H} = 3\sigma_x^1 - 2\sigma_y^6\sigma_z^4 + 2I^{\otimes n},$$

where σ_a^n is application of σ_a on the n 'th qubit (see eq. 4.4 for the Pauli-matrices). The time evolution circuit (circuit 4.35) is run for our example Hamiltonian with this script

```
1 from utils import TimeEvolutionOperator
2 import qiskit as qk
3
4 hamiltonian_list = [[3,[0,'x']],[-2,[5,'y'],[3,'z']],[2]]
5 U = TimeEvolutionOperator(hamiltonian_list,dt=0.001,T=1)
6
7 evo_register = qk.QuantumRegister(6)
8 classical_register = qk.QuantumRegister(4)
9 circuit = qk.QuantumCircuit(evo_register,ancilla_register,classical_register)
10 registers = [evo_register,classical_register]
11
12 circuit,registers = U(circuit=circuit,registers=registers)
```

We initialize the time evolution operator in line 5 by inputting the **hamiltonian_list**, the time step **dt** and the evolution time **T**. We make a call to the function and return the **circuit** and **registers** in line 12. We can also perform the time evolution operation conditional on a control qubit. This is done by defining a control register and utilizing the **ControlledTimeEvolutionOperator** class:

```
1 from utils import ControlledTimeEvolutionOperator
2 import qiskit as qk
3
4 hamiltonian_list = [[3,[0,'x']],[-2,[5,'y'],[3,'z']],[2]]
5 U = ControlledTimeEvolutionOperator(hamiltonian_list,dt=0.001,T=1)
6
7 control_register = qk.QuantumRegister(2)
8 evo_register = qk.QuantumRegister(6)
9 classical_register = qk.QuantumRegister(4)
10 circuit = qk.QuantumCircuit(control_register,evo_register,ancilla_register,classical_register)
11 registers = [control_register,evo_register,classical_register]
12
13 circuit,registers = U(circuit=circuit,registers=registers,control=1,power=2**6)
```

We specify in line 13 that we condition on the second control qubit (**control = 1**) and we also put the whole operator to a power of 2^6 with the **power** argument. Recall from section 4.2 that this is exactly the kind of operator we need to perform a Hamiltonian simulation and find the eigenvalues with the QPE algorithm.

6.2.4 Hamiltonian Simulation

With the previously discussed methods, a class **HamiltonianSimulation** has been made which accepts an arbitrary Hamiltonian in the mentioned format and utilizes the QPE algorithm to calculate its eigenvalues. Below is an example:

```
1 from utils import *
2 from matplotlib.pyplot import *
3 from HamiltonianSimulation import *
4 np.random.seed(10202)
5
```

```

6 u_qubits = 2
7 t_qubits = 8
8
9 hamiltonian_list = [[0.25,[0,'x']], [0.25,[1,'x']], [-0.5]]
10 def initial_state(circuit, registers):
11     return(circuit, registers)
12 E_max = 2
13 hamiltonian_list[-1][0] -= E_max
14
15 dt = 0.005
16 t = 100*dt
17
18 solver = HamiltonianSimulation(u_qubits, t_qubits, hamiltonian_list, initial_state)
19 x, y = solver.measure_eigenvalues(dt, t, E_max, seed_simulator=42)
20
21 plt.plot(x, y)
22 plt.xlabel('Energy')
23 plt.ylabel('Times Measured')
24 plt.title('QPE Example on two-qubit Hamiltonian')
25 plt.show()

```

We want to solve for a two-qubit Hamiltonian utilizing eight qubits for the t -register, hence we put **u_qubits** = 2 and **t_qubits** = 8 in line 6 and 7 (see section 4.2 for explanation of the t -register and u -register). The Hamiltonian we wish to solve for is specified in line 9. We can write it mathematically as

$$\hat{H} = \frac{1}{4}\sigma_x^1 + \frac{1}{4}\sigma_x^2 - \frac{1}{2}I^1I^2.$$

Its eigenvectors $|v\rangle$ and eigenvalues v are given by

$$\begin{aligned}
 |v_1\rangle &= \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle + |1\rangle) & v_1 &= 0, \\
 |v_2\rangle &= \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) & v_2 &= -\frac{1}{2}, \\
 |v_3\rangle &= \frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle + |1\rangle) & v_3 &= \frac{1}{2}, \\
 |v_4\rangle &= \frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle) & v_4 &= -1.
 \end{aligned}$$

We need to decide what initial state to put our qubits in, which is why we define a function called **initial_state** in line 10. This function basically does nothing, since we might as well start in the $|00\rangle$ state. Recall from section 4.2.5 that we want to subtract a value equal to or above the largest eigenvalue of \hat{H} in order to get the whole eigenvalue spectra. This is what is done in line 12 and 13. We initialize the solver in line 18, and calculate the eigenvalue spectra in line 19. Here we input the timestep **dt**, evolution time **t** and the maximum energy **E_max**. We get the following plot:

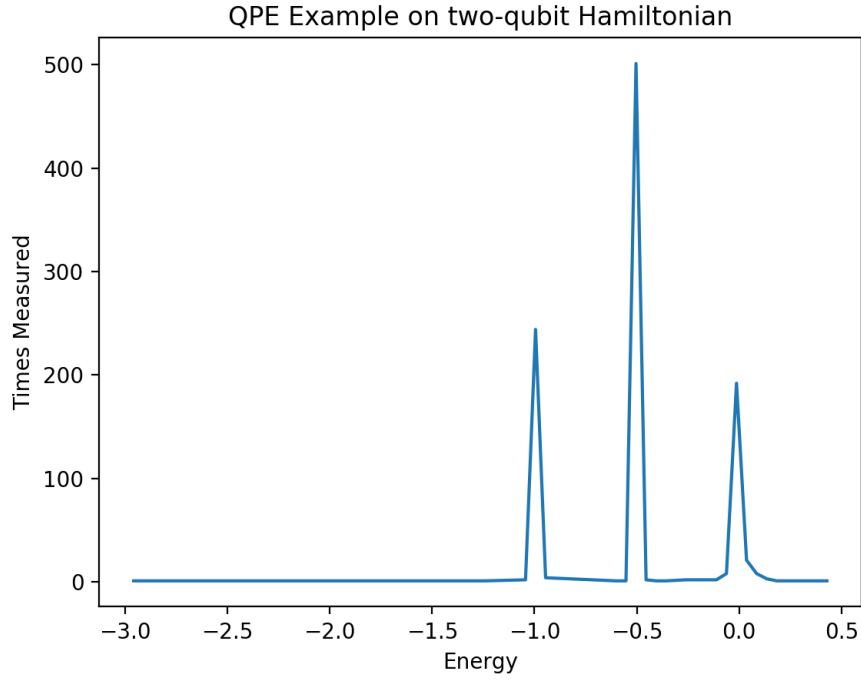


Figure 6.1: The eigenvalue spectra of an example Hamiltonian found with QPE.

We can see that the peaks coincide with the actual eigenvalues.

When we have gotten eigenvalue spectra, we can approximate the j 'th eigenvalue with the mean of the j 'th peak. The equation we will use for this is

$$E[\lambda^{(j)}] = \frac{\sum_i \lambda_i^{(j)} N_i^{(j)}}{\sum_i N_i^{(j)}}, \quad (6.1)$$

where $\lambda_i^{(j)}$ is the i 'th measurement in the j 'th peak. The number of times $\lambda_i^{(j)}$ is measured are given by $N_i^{(j)}$. We can likewise calculate the variance with

$$\frac{1}{\sum_i N_i^{(j)}} \sum_i (E[\lambda^{(j)}] - \lambda_i^{(j)})^2, \quad (6.2)$$

which can be used for uncertainty estimates.

6.3 Variational Quantum Eigensolvers

Next we want to illustrate how to use the variational quantum eigensolver (VQE) explained in section 4.3. We have made a class **VQE** which accepts an arbitrary Hamiltonian in the format discussed earlier as well as an arbitrary ansatz, and gives an upper bound for the ground state energy using Scipy's [48] classical optimization algorithms. To showcase how this class is utilized, we will use it to solve the Max-Cut problem introduced in section 4.3.1. The code to do this is given below

```
1 from utils import *
2 from VQE import *
3 import numpy as np
4 np.random.seed(43)
5 seed_simulator=42
```

```

6
7 W = np.array([[0,3,0,0,1],[3,0,2,0,3],[0,2,0,2,0],[0,0,2,0,3],[1,3,0,3,0]])
8
9 hamiltonian_list = max_cut_hamiltonian(W)
10
11 n_qubits = 5
12 solver = VQE(hamiltonian_list,y_rotation_ansatz,n_qubits,seed_simulator=seed_simulator)
13
14 theta = np.random.randn(n_qubits)
15 theta = solver.classical_optimization(theta,method='Powell')
16
17 ansatz_register = qk.QuantumRegister(n_qubits)
18 classical_register = qk.ClassicalRegister(n_qubits)
19 circuit= qk.QuantumCircuit(ansatz_register,classical_register)
20 registers = [ansatz_register,classical_register]
21
22 circuit,registers = y_rotation_ansatz(theta,circuit,registers)
23
24 circuit.measure(registers[0],registers[1])
25 job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'), shots=1000)
26 result = job.result()
27 result = result.get_counts(circuit)
28 print(result)

```

In line 7, we set up the Max-Cut matrix for the problem given in figure 4.4. In line 9, we use the function `max_cut_hamiltonian`, which creates the Hamiltonian list for an arbitrary Max-Cut matrix. As we wish to solve for a 5 by 5 matrix, we utilize five qubits. The solver is set up in line 12 by inputting the Hamiltonian list, the ansatz and the number of qubits. In this case we use `y_rotation_ansatz`, which is the ansatz given by circuit 4.50. We then initialize a random set of parameters in line 14 and run the solver in line 15. In order to check our solution, we apply the ansatz to a circuit with our learned parameters and measure the results:

```
{'10101': 128, '01010': 872}
```

We can see from figure 4.5 that the solution is found.

6.4 Quantum Adiabatic Time Evolution

Next we explain how to utilize the quantum adiabatic time evolution (QATE) algorithm explained in section 4.4. We have made a class **QATE** which can slowly evolve the Hamiltonian from an initial Hamiltonian \hat{H}_0 to the Hamiltonian we wish to solve for, \hat{H}_1 . For the following example, we start in the initial state of the following four-qubit Hamiltonian

$$\hat{H}_0 = \frac{1}{5}\sigma_z^1 + \frac{1}{5}\sigma_z^2 - \frac{1}{5}\sigma_z^3 - \frac{1}{5}\sigma_z^4.$$

The ground state for this Hamiltonian is

$$|1100\rangle.$$

We wish to utilize the QATE algorithm to evolve into the ground state of

$$\hat{H}_1 = \frac{1}{4}\sigma_x^1 + \frac{1}{4}\sigma_x^2 - \frac{1}{2}I^1I^2,$$

for which we know from section 6.2.4 has a ground state energy of -1 . The code to set up this problem is

```

1 from QATE import *
2
3 n_spin_orbitals=4
4 factor = 0.2
5 H_0 = [[factor,[0,'z']], [factor,[1,'z']], [-factor,[2,'z']], [-factor,[3,'z']]]
6 H_1 = [[0.25,[0,'x']], [0.25,[1,'x']], [-0.5]]
7
8 def initial_state(circuit,registers):
9     for i in range(int(len(registers[0])/2)):

```

6. Methods

```
10     circuit.x(registers[0][i])
11     return(circuit, registers)
12
13
14
15 dt = 0.5
16 t=22
17 solver = QATE(n_spin_orbitals,H_0,H_1,initial_state,dt,t,seed_simulator=42)
18 print(solver.calculate_energy())
```

-0.9915

The initial Hamiltonian, \hat{H}_0 , is set up in line 5 while the Hamiltonian we wish to solve for, \hat{H}_1 , is set up in line 6. We define a function in line 8 which produces the ground state of \hat{H}_0 . In line 15 we decide the step size **dt** in the time-ordered exponential, and we define the evolution time **t** in line 16. In line 17, we initialize the **QATE** class and we calculate the energy of the resulting state in line 18. We see that the found energy is close the actual ground state energy.

6.5 Quantum Machine Learning

One goal of this thesis was to make an intuitive deep learning framework for quantum computers inspired by Pytorch and Tensorflow. This means that one should easily be able to utilize and combine the layers discussed in this thesis to construct and train a deep neural network. The first step was to make an amplitude encoder like the one discussed in section 5.1, which accepts an arbitrary vector **x** and encodes its values into the amplitudes of a quantum state. The **AmplitudeEncoder** class is contained in the `utils.py` script (see GitHub link in introduction) and we will now showcase how to utilize this functionality.

6.5.1 Amplitude Encoder

```
1 from utils import AmplitudeEncoder
2 import numpy as np
3 import qiskit as qk
4
5 np.random.seed(42)
6 x = np.random.randn(4)
7
8 amplitude_register = qk.QuantumRegister(2)
9 classical_register = qk.ClassicalRegister(2)
10 circuit = qk.QuantumCircuit(amplitude_register, classical_register)
11 registers = [amplitude_register, classical_register]
12
13 encoder = AmplitudeEncoder()
14 circuit, registers = encoder(x, circuit, registers, inverse=False)
15
16 shots = 1000000
17 circuit.measure(registers[0], registers[-1])
18 job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'), seed_simulator=42, shots=shots)
19 result = job.result().get_counts(circuit)
20 for key, value in result.items():
21     print('Qubit State:', key[::-1], '. Square root of probability:', np.sqrt(value/shots))
22
23 print('Normalized amplitude encoded vector: ')
24 print(np.abs(x)/np.sqrt(np.sum(x**2)))
```

We first generate a numpy array with random numbers of size 4, called **x**, in line 6. The registers and circuits for the problem are set up in lines 8 through 11. The **AmplitudeEncoder** is initialized in line 13. We simply feed in the **circuit**, the **registers** and the numpy array **x** into the encoder in line 14. The **inverse** variable can be set to **True** if one rather wants to apply the inverse of the amplitude encoding operation. From line 16 and through, we print out the qubit state and the square root of the estimated probability of measuring the

respective state. Finally we print the absolute value of the normalized array \mathbf{x} for comparison. This yields the following output

```
Qubit State: 11 .Square root of probability: 0.8787121257840932
Qubit State: 00 .Square root of probability: 0.28677168618955395
Qubit State: 10 .Square root of probability: 0.3732050374793995
Qubit State: 01 .Square root of probability: 0.07965550828411053
Normalized amplitude encoded vector:
[ 0.28654116 0.07976099 0.37363426 0.87859535]
```

We can see that the amplitude encoding was successful as the square root of the probability of state $|00\rangle$ closely matches the first element of \mathbf{x} , the square root of the probability of state $|01\rangle$ closely matches the second element of \mathbf{x} , etc. As we are only able to apply a finite number of measurements on the circuit, we do not get the exact probabilities.

6.5.2 Inner Product

The neural network layers in section 5.3 and 5.4 rely on calculating the squared inner product between a vector \mathbf{x} and a vector \mathbf{w} , by utilizing circuit 5.11. We have implemented a function called `squared_inner_product` in the `utils.py` script to do this task. Below is a showcase of how to utilize its functionality

```
1 from utils import AmplitudeEncoder
2 from utils import squared_inner_product
3 import numpy as np
4 import qiskit as qk
5
6 np.random.seed(42)
7 x = np.random.randn(4)
8 w = np.random.randn(4)
9
10
11 amplitude_register = qk.QuantumRegister(2)
12 classical_register = qk.ClassicalRegister(1)
13 circuit = qk.QuantumCircuit(amplitude_register, classical_register)
14 registers = [amplitude_register, classical_register]
15
16
17 sq_inner_product = squared_inner_product(x, w, circuit, registers, seed_simulator=42, shots=1000000)
18 x = x/np.sqrt(np.sum(x**2))
19 w = w/np.sqrt(np.sum(w**2))
20 print('Squared inner product calculated classically:', np.sum(x*w)**2)
21 print('Squared inner product calculated by quantum circuit:', sq_inner_product)
```

The two random vectors \mathbf{x} and \mathbf{w} are initialized in lines 7 and 8, respectively. The Qiskit circuits and registers are initialized in lines 11 through 14. We only use one classical qubit (line 12) as the squared inner product circuit only relies on measuring one qubit. We feed the vector \mathbf{x} , \mathbf{w} as well as the **circuit** and **registers** into the `squared_inner_product` function in line 17, which should then return their squared inner product. Finally we compare the classical calculation with the quantum one. This yields the following output

```
Squared inner product calculated classically: 0.4630815670795022
Squared inner product calculated by quantum circuit: 0.463201
```

We can see that the quantum circuit gives a close approximation to the classical calculation. As we only measure the circuits a finite number of times, we do not get an exact match.

6.5.3 Neural Network

All the dense and recurrent neural network layers discussed in this thesis can be written in the general form described in sections 5.5 and 5.6, respectively. Hence, we have made the classes **GeneralLinear** and **GeneralRecurrent** which can be utilized to construct an arbitrary layer consisting of the operators U_{enc} , U_a and U_{ent} . See section 5.5 for an explanation of these. We will now show how these classes are used to realize a neural network layer. The dense layer can be initialized via a call to the **GeneralLinear** class:

```
1 from layers import GeneralLinear
2 GeneralLinear(n_qubits=n_qubits, n_outputs=n_outputs, n_weights_ent=n_weights_ent, n_weights_a=n_weights_a, bias=
  bias, U_enc=U_enc, U_a=U_a, U_ent=U_ent, n_parallel=n_parallel)
```

We of course will have to specify each of the arguments. Since all the arguments are dependent on the encoder **U__enc**, ansatz **U__a** and entangler **U__ent**, we will first discuss how these are set up.

Encoders and Ansatzes

As the initialization of the layers relies on inputting an encoder **U__enc** and an ansatz **U__a**, we will now discuss how these are set up. See section 5.7 for an explanation of what these are used for. The requirement for the encoders and ansatzes is that they are some callable object, lets call it **U**, which is utilized the following way:

```
1 circuit, registers = U(theta, circuit, registers)
```

The argument **theta** should be a one-dimensional numpy array of parameters, **circuit** should be a qiskit QuantumCircuit instance, and **registers** should be a list containing an instance of a qiskit QuantumRegister as its first element. The **circuit** with the applied operation should be returned along with the **registers** list. We can for example create an entangler/ansatz which only applies an R_y -rotation (eq. 4.6) to each qubit:

```
1 def U_a_or_enc(theta, circuit, registers):
2     for i in range(len(registers[0])):
3         circuit.ry(theta[i], registers[0][i])
4     return(circuit, registers)
```

This function is now ready to be implemented as **U__enc** or **U__a** to the initialization of the **GeneralLinear** class.

Entanglers

The set up of the entangler **U__ent** is not so different from the encoders and ansatzes. The purpose of the entangler is discussed in section 5.8. We require some callable object, lets call it **U**, which is utilized the following way

```
1 circuit, registers = U(theta, ancilla, circuit, registers)
```

The first argument, **theta**, is a one-dimensional numpy array with the eventual parameters for the entangler. Recall that the entangler is responsible for entangling the encoder-register with an ancilla register. The second argument, **ancilla**, is the index of the ancilla we wish to entangle. The third argument, **circuit**, should be a Qiskit QuantumCircuit instance. Finally, **registers** should be a list containing the encoder-register as its first element and also the ancilla-register as its second element. As an example, we will now show how to define a variation of the entangler shown in circuit 5.30:

```
1 def U_ent(theta, ancilla, circuit, registers):
2     circuit.ry(theta[-1], registers[1][ancilla])
3     for i in range(len(registers[0])):
4         circuit.cry(theta[i], registers[0][i], registers[1][ancilla])
5     return(circuit, registers)
```

This entangler is now ready to be utilized as input to the **GeneralLinear** class initialization.

Setting up a single layer

Now that we have defined an encoder, ansatz and entangler, we have what we need to initialize and feed our data into a neural network. The code is given below

```

1 from layers import GeneralLinear
2 import numpy as np
3 np.random.seed(42)
4
5 def U_a_or_enc(theta,circuit,registers):
6     for i in range(len(registers[0])):
7         circuit.ry(theta[i],registers[0][i])
8     return(circuit,registers)
9
10 def U_ent(theta,ancilla,circuit,registers):
11     circuit.ry(theta[-1],registers[1][ancilla])
12     for i in range(len(registers[0])):
13         circuit.cry(theta[i],registers[0][i],registers[1][ancilla])
14     return(circuit,registers)
15
16 n = 4
17 p = 2
18 x = np.random.randn(n,p)
19 dense_layer = GeneralLinear(n_qubits=2,n_outputs=3,n_weights_a=2,n_weights_ent=3,U_enc=U_a_or_enc,U_a =
    U_a_or_enc, U_ent=U_ent,n_parallel= 3,seed_simulator=42)
20
21 print('Total number of weights: ',dense_layer.w_size)
22 y = dense_layer(x)
23 print('Output from layer: ')
24 print(y)

```

Total number of weights: 11

Output from layer:

```

[[0.103 0.816 0.973]
 [0.04  0.713 0.805]
 [0.017 0.632 0.616]
 [0.051 0.743 0.906]]

```

In line 18, we generate a data set containing four samples and two predictors of normally distributed random numbers. Our dense layer is initialized in line 19. Let us explain the arguments provided in the initialization. We want to utilize two qubits for our layer. We specify this with the **n_qubits** argument. We want three nodes outputted, hence **n_outputs = 3**. The argument **n_weights_a** specifies how many parameter is required for our ansatz, per activation. The ansatz applies a single rotation to each qubit. For two qubits we specify **n_weights_a = 2**. The number of weights required per activation for the entangler is specified with the **n_weights_ent** argument. The entangler is first applying a single rotation to the ancilla followed by two controlled rotations (one per qubit in the encoder-register). There are three parameters involved in this, hence we set **n_weights_ent = 3**. As we discussed in section 5.9, the entangler we utilize allows for parallel calculation of nodes. We decided that we wanted to calculate all three outputs in parallel, hence we set **n_parallel = 3**. All layer instances has an attribute called **w_size** which is an integer specifying the total number of weights in the network. We print this in line 21. We can see from the output that this layer has 11 weights.

Setting up a Multi-Layered Neural Network

We will now show how to set up a multi-layered neural network. We will utilize the GeneralRecurrent class as well as the GeneralLinear class to show that we can handle time-series data. Assume we have two samples with eight time steps and four predictors in our input. We want to feed this through a recurrent layer and then use the final hidden vector h^t , consisting of three nodes, as inputs to a dense output layer with four nodes. This can be done the following way

```

1 from layers import GeneralLinear, GeneralRecurrent

```

6. Methods

```
2 from utils import AmplitudeEncoder
3 from dl_utils import EntanglementRotation
4 import numpy as np
5 np.random.seed(42)
6 seed_simulator = 42
7
8 x = np.random.randn(2,8,4) #Generate random data (2 samples, 8 time steps, 4 predictors)
9 h_0 = np.ones(3) #Generate initial hidden vector (3 nodes)
10
11 recurrent_layer = GeneralRecurrent(n_qubits=3,n_hidden=3,n_weights_ent=0,n_weights_a=7,bias=True,U_enc=
    AmplitudeEncoder(),U_a=AmplitudeEncoder(inverse=True),U_ent=EntanglementRotation(zero_condition=True),
    seed_simulator=seed_simulator)
12 output_layer = GeneralLinear(n_qubits=2,n_outputs=4,n_weights_ent=0,n_weights_a=4,bias=True,U_enc=
    AmplitudeEncoder(),U_a=AmplitudeEncoder(inverse=True),U_ent=EntanglementRotation(zero_condition=True),
    seed_simulator=seed_simulator+1)
13
14 output = recurrent_layer(x,h_0)[:,-1,:] #Choose the final hidden vector from recurrent layer
15 output = output_layer(output) #Feed this into dense output layer
16
17 print('Output:')
18 print(output)
```

Output:

```
[[0.057 0.307 0.432 0.084]
 [0.008 0.586 0.635 0.006]]
```

In line 2, we import the **AmplitudeEncoder** class as we wish to utilize this as our encoder and ansatz for both layers. In line 3, we import a class called **EntanglementRotation**, which performs the entangler shown in circuit 5.28. Our neural network is then consisting of the classical layers explained in sections 5.3 and 5.4. In line 8 and 9 we generate the input time series data and initial hidden vector, respectively. In line 11, we initialize the recurrent layer. As explained in section 5.4, our recurrent layers requires the concatenation of the input vector and the hidden vector. Since our input has four predictors and our hidden vector has three nodes, we need 3 qubits to encode these. Hence, we set **n_qubits = 3**. We also need to specify the dimension of the hidden vector with the argument **n_hidden**. The rest of the initialization of the network should be familiar from the example in section 6.5.3, except that we have specified **bias = True**. This will add a bias to the weight and a constant to the input vector as explained in section 5.3. In line 14, we feed our input and initial hidden vector through the recurrent layer. We get out the hidden vector at each time step, so we pick the final hidden vector through indexing. We then feed this final hidden vector through the dense output layer in line 15. Finally we print out the output. For those that are familiar with Pytorch, this way of putting together a neural network and combining different kinds of layers is not unlike what one would do utilizing this package. We have in fact included all the entanglers, ansatzes and entanglers discussed in sections 5.7 and 5.8, and they can all be used freely to create a custom neural network with an arbitrary amount of layers.

Learning from a data set

As finding quantum circuits for training the architectures discussed in chapter 5 have not been a focus in this thesis, we utilize classical numerical methods provided by Scipy [48] to train the networks. A Python class called **QDNN** has been made, which can train an arbitrary dense neural network this way. This class is found in the QDNN.py script on our GitHub. We will showcase its functionality by learning a classifier on the Iris data set [41], which is a classification task with four predictors and three classes. First we start out with importing the relevant functionality and setting up the data set:

```
1 import numpy as np
2 import pandas as pd
3 from matplotlib.pyplot import *
4 from QDNN import *
5 from layers import *
6 from loss import *
7 from sklearn.datasets import load_iris
8 from sklearn.metrics import accuracy_score, confusion_matrix
```

```

9 from sklearn.model_selection import train_test_split
10 np.random.seed(22)
11 seed_simulator = 47
12
13 iris = load_iris()
14 X = iris['data']
15 y = iris['target']
16 X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3)
17 print('train dataset shape', X_train.shape)
18 print('test dataset shape', X_test.shape)

```

```

train dataset shape (105, 4)
test dataset shape (45, 4)

```

The next step is to initialize the neural network:

```

1 l1 = X_train.shape[1]
2 l2 = 8
3
4 layer1 = GeneralLinear(2,l2,n_weights_a=2,n_weights_ent=3,U_enc=AmplitudeEncoder(),U_a = YRotationAnsatz(
    linear_entangler),U_ent=YRotation(bias=True),seed_simulator=seed_simulator,n_parallel=l2)
5 layer2 = GeneralLinear(8,3,n_weights_a=l2,n_weights_ent=l2+1,U_enc=YRotationAnsatz(linear_entangler),U_a=
    YRotationAnsatz(linear_entangler),U_ent=YRotation(bias=True),seed_simulator=42)
6
7 layers = [layer1,layer2]
8 loss_fn = cross_entropy()
9
10 model = QDNN(layers,loss_fn,classification=True)

```

We can see from line 10 that the dense neural network initialization requires a list of layers. These are specified in lines 4, 5 and 7. We also need to input a loss function that returns the loss when the predicted values and the target values are passed as arguments. As we are dealing with a classification problem, we pass **classification=True** as well. After this is done, we can simply fit the model and predict on the test set:

```

1 model.fit(X=X_train,y=y_train)
2 y_pred = model.forward(X_test)
3 y_pred = np.argmax(y_pred,axis=1)
4
5 print('Confusion Matrix:')
6 print(confusion_matrix(y_test,y_pred))

```

In line 1, we call the **.fit** function which utilizes a classical optimization scheme to train the network. We pass the training data, **X_train** and **y_train** as arguments. In line 2 and 3, we output the prediction on a separate test set. The predictions on the test set resulted in the following confusion matrix:

Table 6.1: Quantum Neural Network trained on Iris data set. Confusion matrix for separate test set.

	\hat{y}		
y	13	0	0
	0	16	0
	0	10	6

6.5.4 Learning Unitary Operators

In section 5.10, we discussed using machine learning to approximate a unitary operator with another parametrized unitary operator. The **AutoEncoder.py** script in our GitHub contains a class meant to be utilized for this task, namely the **AutoEncoder** class. To show how this class works, let us try to utilize it to approximate a random unitary operator with the Euler rotation ansatz given by circuit 4.54. First, let us generate a random circuit and see what the resulting state looks like:

6. Methods

```
1 from AutoEncoder import *
2 from utils import *
3 from qiskit.circuit.random import random_circuit
4 import qiskit as qk
5
6
7 n_qubits = 3
8 depth = 30
9 rand_circuit = random_circuit(n_qubits,depth,seed=1)
10 q_reg = qk.QuantumRegister(3,'q')
11 c_reg = qk.ClassicalRegister(3)
12 circuit = qk.QuantumCircuit(q_reg,c_reg)
13
14 circuit = circuit.combine(rand_circuit.copy())
15 circuit.measure(q_reg,c_reg)
16 job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'),seed_simulator=42,shots=1000)
17 job = job.result()
18 result = job.get_counts(circuit)
19 for key,value in result.items():
20     print('State: ', key[::-1], 'Measurements: ', value)
```

```
State: 000 Measurements: 144
State: 111 Measurements: 25
State: 010 Measurements: 402
State: 101 Measurements: 67
State: 110 Measurements: 26
State: 001 Measurements: 106
State: 100 Measurements: 105
State: 011 Measurements: 125
```

In line 9, we generate a random 3 qubit circuit with circuit depth of 30. Next, let us use the **AutoEncoder** class to approximate this circuit

```
1 U_1 = EulerRotationAnsatz(linear_entangler)
2 def U_2(theta,circuit,registers):
3     circuit += rand_circuit.copy().inverse()
4     return(circuit,registers)
5
6 optimizer = AutoEncoder(U_1,U_2,n_qubits = 3, n_weights=3*3*3,shots=10000,seed_simulator=42)
7 w = optimizer.fit()
```

```
Optimization terminated successfully.
Current function value: -0.999500
Iterations: 10
Function evaluations: 6245
```

We initialize the ansatz in line 1, and create a function that applies the random circuit in line 2 to 4. We utilized these functions as the first two arguments in the **AutoEncoder** initialization in line 6. We also have to specify the number of qubits, which we do by setting **n_qubits = 3**. The number of parameters we utilize for the ansatz are specified by putting **n_weights = 9** as we want $d = 3$ in the ansatz (see circuit 4.54 for explanation of the parameter d). In the final line, we utilize the **fit** function of the **AutoEncoder** class to learn the parameters with a classical optimization scheme. We see that we achieve a function value of -0.9995, which means that the squared inner product between the state produced by the ansatz and the state produced by the random circuit is 0.9995. Next, let us print the depth of the ansatz and also measure the state produced by it

```
1 circuit = qk.QuantumCircuit(q_reg,c_reg)
2 registers = [q_reg,c_reg]
3 circuit,registers = U_1(w,circuit,registers)
```

```
4 print('Depth of Ansatz: ',circuit.depth())
5
6 circuit.measure(q_reg,c_reg)
7 job = qk.execute(circuit, backend = qk.Aer.get_backend('qasm_simulator'),seed_simulator=42,shots=1000)
8 job = job.result()
9 result = job.get_counts(circuit)
10 for key,value in result.items():
11     print('State: ', key[::-1], 'Measurements: ', value)
```

```
Depth of Ansatz:  15
State:  000 Measurements:  151
State:  111 Measurements:   23
State:  010 Measurements:  393
State:  101 Measurements:   67
State:  110 Measurements:   27
State:  001 Measurements:  105
State:  100 Measurements:  107
State:  011 Measurements:  127
```

We can see that we have produced a pretty close state while utilizing a circuit with half the circuit depth.

The recursive circuit optimization scheme proposed in section 5.11 can be performed by recursively utilizing the **AutoEncoder** class for each of the k circuits in circuit 5.37.

PART IV

Analysis

CHAPTER 7

Results

In this chapter we will present the results from applying the various methods discussed in the previous chapters. How these algorithms were set up to solve for a specific problem will be explained, along with some short comments on the corresponding results. An in-depth discussion of the utilized method will then be provided at the end of each section.

We will first take a look at how the quantum computing many-body methods described in chapter 4 fare on a simple Hamiltonian. We will compare these results with the ones given by standard many-body methods, such as the full configuration interaction (FCI) theory (section 2.5) and the coupled cluster doubles (CCD) theory (section 2.8). For all the quantum computing algorithms, we applied the Jordan-Wigner transformation (section 4.2.4) on our Hamiltonian to rewrite it in terms of the Pauli-gates in order to represent it on a quantum computer.

After having gone through the results for the many-body methods, we will apply the various quantum machine learning algorithms described in chapter 5 on a couple of selected tasks.

7.1 Quantum Phase Estimation

We wanted to see if we could find the eigenvalues of the Jordan-Wigner transformed pairing Hamiltonian in eq. 4.29, utilizing the quantum phase estimation (QPE) algorithm described in section 4.2. We restricted ourselves to solve for a system of four spin-orbitals. Recall that when the QPE algorithm is used to solve for eigenvalues, we are only able to obtain the negative ones. We explained this in section 4.2.5, and showed that we could force only negative eigenvalues by subtracting a constant E_{max} from our Hamiltonian, larger than or equal to its largest eigenvalue. The actual eigenvalues could then be found by utilizing E_{max} in eq. 4.37. In reality, one has to do some educated guess on the maximum eigenvalue of the Hamiltonian, perhaps by approximating it with some other computationally effective method. In our case, we knew the eigenvalues for the problem beforehand and subtracted $E_{max} = 2$. The QPE algorithm also requires us to utilize the Suzuki-Trotter transformation (section 4.2.3) to approximate the time evolution operator for our Hamiltonian. This is in part done by dividing the evolution time t into small time steps Δt . The approximation error in the Suzuki-Trotter approximation (see eq. 4.20) decreases with the size of the time steps. Hence, we chose $\Delta t = 0.005$. In addition, the total evolution time t was shown to have an upper bound for the QPE algorithm to yield all the eigenvalues of our Hamiltonian. This upper bound was given by eq. 4.38 and we chose $t = 100$ with this in mind.

We will first represent the result from an ideal simulation of a quantum computer and benchmark them against the eigenvalues found with full configuration interaction (FCI) theory (section 2.5). Then we will run a simulation with the noise model from one of IBM's current quantum devices.

7.1.1 Ideal Simulation

First we wanted to see if we could find the correct eigenvalues when running an ideal simulation of a quantum computer. Recall from the QPE circuit in figure 4.3 that the QPE algorithm require us to prepare two quantum registers. The t -register will end up with the eigenvalues encoded as a binary fraction (eq. 4.18). The u -register will be acted on by the time evolution operator for our Hamiltonian. Since we solve for a system of four spin-orbitals, the u -register will contain four u -qubits. Since the t -register will hold the eigenvalues as binary fractions, the amount of t -qubits will decide the binary fractions we are able to represent. See the discussion

7. Results

at the end of section 4.2.5. Hence, we varied the amount of qubits in the t -register to see how it affected the obtained eigenvalues. The results can be seen in figure 7.1.

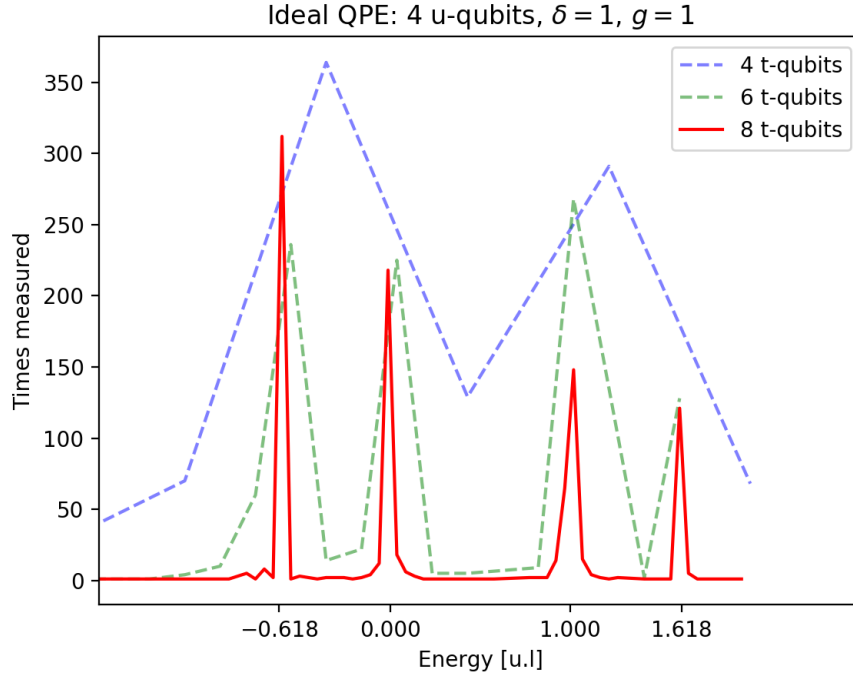


Figure 7.1: QPE on pairing Hamiltonian (eq. 4.26) with $\delta = 1$, interaction strength $g = 1$ and four spin-orbitals. We plot the amount of times we measure an energy against the energy measurement, for varying number of qubits in the t -register. The FCI energies for four spin-orbitals are marked on the x -axis.

We see that as we increase the number of t -qubits, peaks are starting to form by the eigenvalues given by FCI. We can approximate the QPE eigenvalues by calculating the mean of each peak with eq. 6.1. We can also get uncertainty estimates for each eigenvalue by calculating the variance of each peak with eq. 6.2. The results for 8 t -qubits can be seen in table 7.1 below:

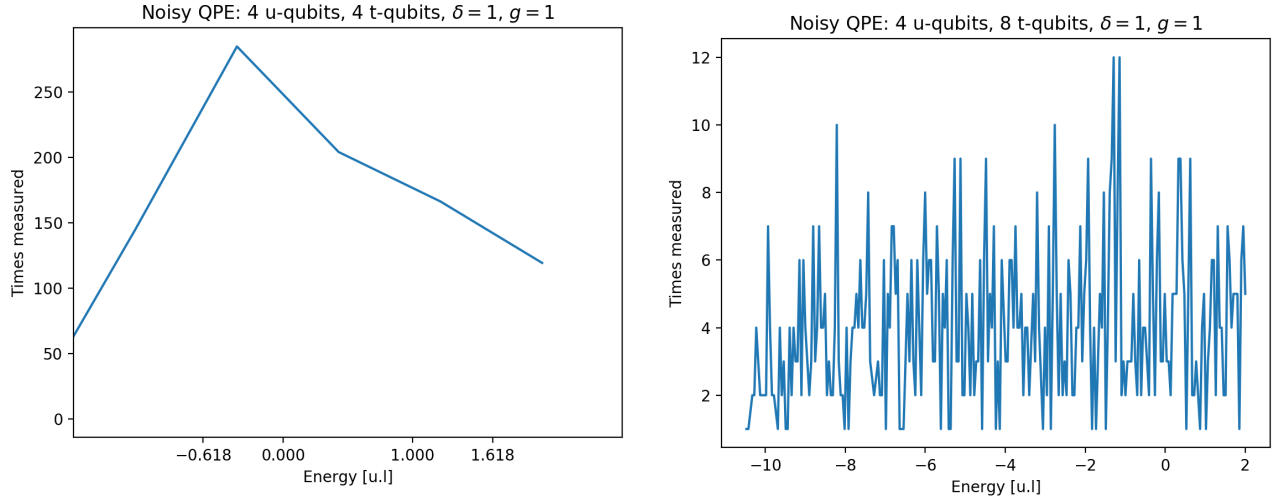
Table 7.1: 8 t -qubit QPE and FCI energy for Pairing model (eq. 4.26). Calculated for four basis states with one and two pairs. $\delta = 1$, $g = 1$. We also provide two standard deviations for the QPE estimate.

Number of pairs	FCI Eigenvalue(s)	Quantum Eigenvalue(s)
0	0	0.0 ± 0.2
1	-0.61803399, 1.61803399	-0.6 ± 0.1 , 1.61 ± 0.06
2	1.00000000	1.0 ± 0.2

We see that the eigenvalues obtained with QPE are close to the FCI eigenvalues.

7.1.2 Noisy Simulation

Next we wanted see how the QPE algorithm performed when running a simulation with the noise model from one of IBM's real quantum devices. We chose the IBM Melbourne 16 qubit quantum computer for this. The simulation is run with 4 and 8 t -qubits. The results can be seen in figure 7.2.



(a) QPE with 4 t -qubits. We plot the amount of times we measure an energy against the energy measurement. The FCI energies for four spin-orbitals are marked on the x -axis. (b) QPE with eight t -qubits. We plot the amount of times we measure an energy against the energy measurement.

Figure 7.2: QPE algorithm on pairing Hamiltonian with noise model from the IBM Melbourne Quantum Computer.

We are not able to reproduce the results from the ideal simulation in figure 7.1.

7.1.3 Discussion

The results from the ideal simulation, figure 7.1 and table 7.1, show that we are able to get sensible estimates for the eigenvalues of the pairing Hamiltonian, provided we utilize enough qubits in the t -register. The figure illustrates that the increase of t -qubits results in narrower peaks about the FCI eigenvalues. Hence, we expect that further increasing the number of qubits in the t -register could provide more accurate eigenvalue-estimates than the ones shown in table 7.1. Figure 7.2a shows the QPE algorithm applied on the same system, but the simulation of the quantum computer includes the noise model from the IBM Q Melbourne 16 qubit device. With only 4 t -qubits, we already see that we are not able to reproduce the results from the ideal simulation in figure 7.1. We explained in section 4.2 that the QPE algorithm requires the application of several steps with the Suzuki-Trotter approximation, in order to simulate the time evolution of our Hamiltonian. We may explain the inability of the algorithm to perform in noisy conditions with the circuit depth achieved as a result of these steps. We mentioned in section 4.1.3 that the circuit depth roughly translates to the execution time of a quantum circuit. On noisy intermediate-scale quantum (NISQ) devices, we are not able to maintain a stable quantum-state a sufficient enough time for the QPE algorithm, and each step of the Suzuki-Trotter approximation is essentially doubling the number of gates required in our quantum circuit (see eq. 4.19). In addition to this, we need to apply the time evolution operator an additional time for every t -qubit we add (see the QPE circuit in figure 4.3). We hence expect the simulation of larger systems (which are requiring more t -qubits) to be even more affected by noise. Figure 7.2b supports this expectation by showing the results from a simulation with the same noise model, but utilizing eight t -qubits instead of four. We see even less resemblance to the ideal simulation.

7.2 Variational Quantum Eigensolver

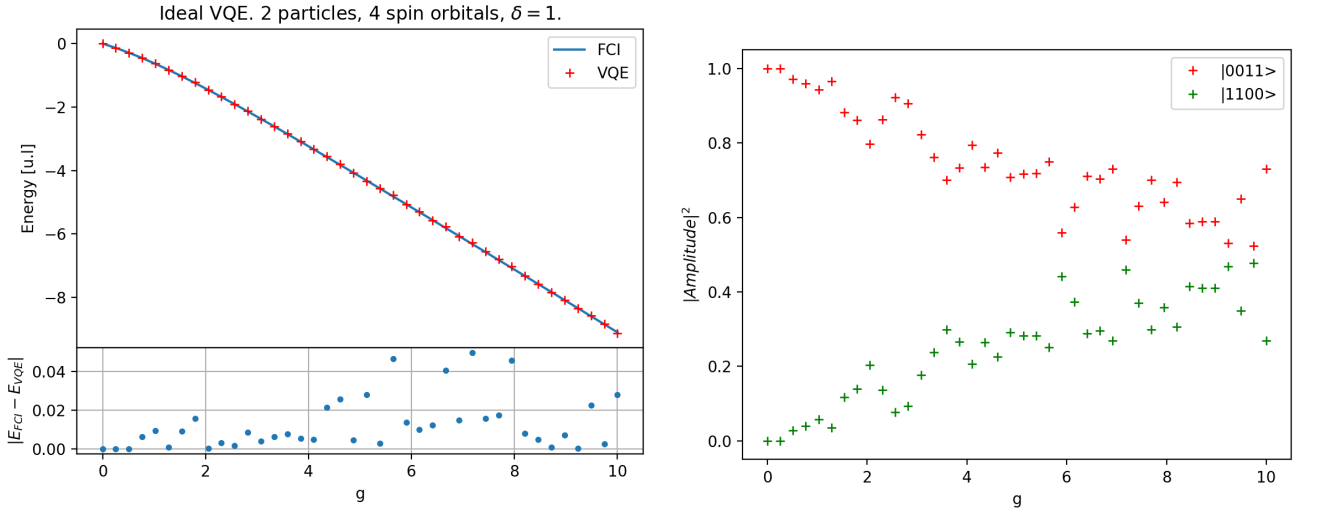
The quantum phase estimation (QPE) algorithm gave reasonable results when running an ideal simulation of a quantum computer, but we were not able to reproduce these results when the simulation included the noise model of a real device. The variational quantum eigensolver (VQE) algorithm (section 4.3) may be able to do better on noisy devices. Since we are evaluating each single term of the Hamiltonian with separate circuits (see eq. 4.40), the circuit depth (time complexity) of our algorithm is mostly dependent on the variational ansatz (see section 4.3.3). In this section, we will solve for the Jordan-Wigner transformed pairing Hamiltonian (eq.

7. Results

4.26) utilizing the VQE algorithm. We will do this for two particles and four spin-orbitals, as well as for four particles and eight spin-orbitals. For the former system, we will compare the results between an ideal simulation, a simulation including the noise model of the IBM Q London five qubit device, as well as an execution on the actual device. For the eight spin-orbital system, we will only consider the ideal simulation.

7.2.1 Ideal Simulation

We will start with the ideal simulation for two particles and four spin-orbitals. The variational ansatz will be put to the simple ansatz with one variational parameter, explained in section 4.3.5 and shown in circuit 4.60. We wanted to see if the result had some dependence on the pairing interaction strength g . Hence, we estimated the ground state energy by varying g and keeping constant $\delta = 1$. We compared our estimates with the eigenvalues obtain using full configuration interaction (FCI) theory (section 2.5). The results can be seen in figure 7.3.



(a) The ground state energy approximation is plotted a function of the interaction strength g and compared with the results from FCI. We also plot the absolute error $|E_{FCI} - E_{VQE}|$ against g . (b) The probability of each term in the ground state given by VQE, against the interaction strength g .

Figure 7.3: Ideal VQE on pairing Hamiltonian with two particles and four spin orbitals. The simple ansatz with one rotation parameter in circuit 4.60 is utilized.

We see that there is no obvious dependence between the energy approximation and g .

For the four-particle, eight spin-orbital system, we utilized the unitary coupled cluster doubles (UCCD) ansatz in section 4.3.4. We expected the energy estimate from the coupled cluster doubles (CCD) method (section 2.8) to deviate from the FCI energy for this system, so we compare our VQE results with both CCD and FCI. The results for varying interaction strength g can be seen in figure 7.4.

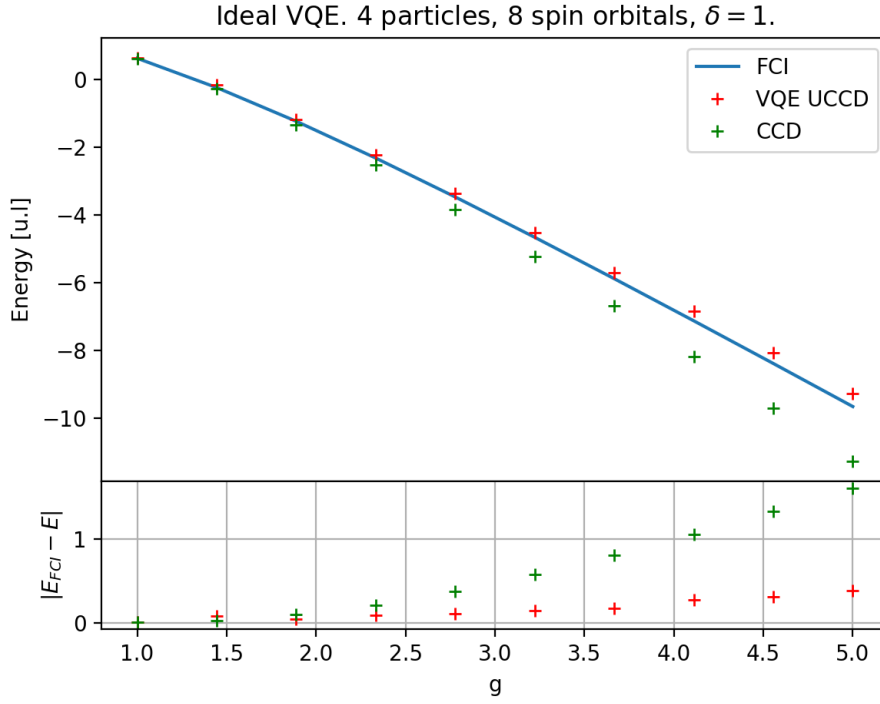


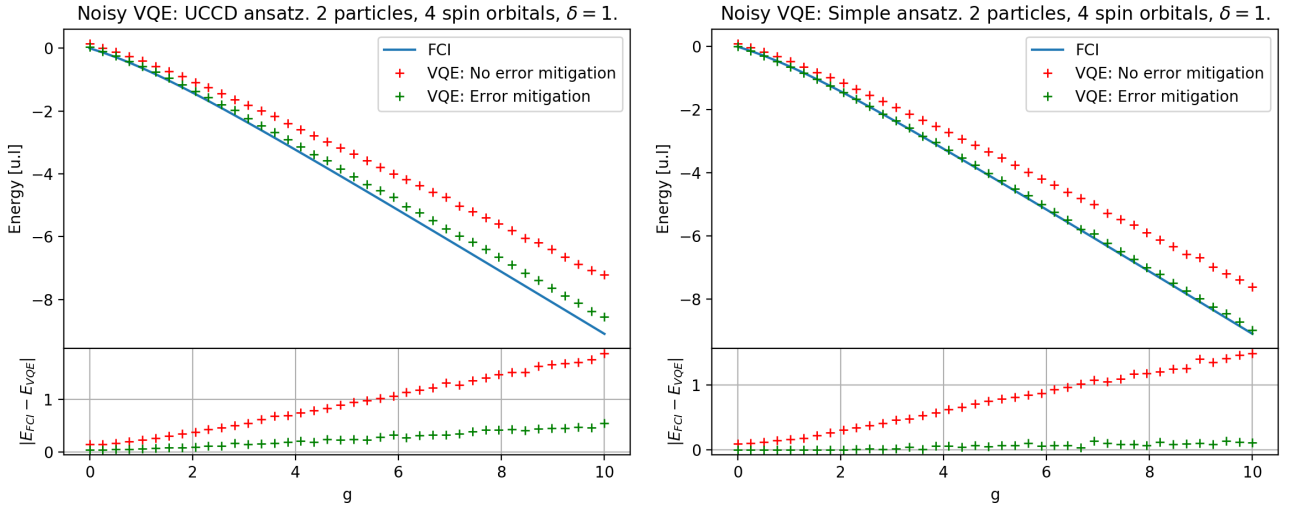
Figure 7.4: Ideal VQE on Pairing Hamiltonian with four particles and eight spin-orbitals. We utilized the UCCD ansatz explained in section 4.3.4. We compare the results with CCD and FCI for the same system. We plot the ground state energy against the interaction strength g , together with the absolute difference between the FCI energy and the UCCD and CCD energy.

We see that the absolute difference between the FCI energy and the UCCD/CCD energy increases with increasing interaction strength g . There is less deviation for UCCD than for CCD.

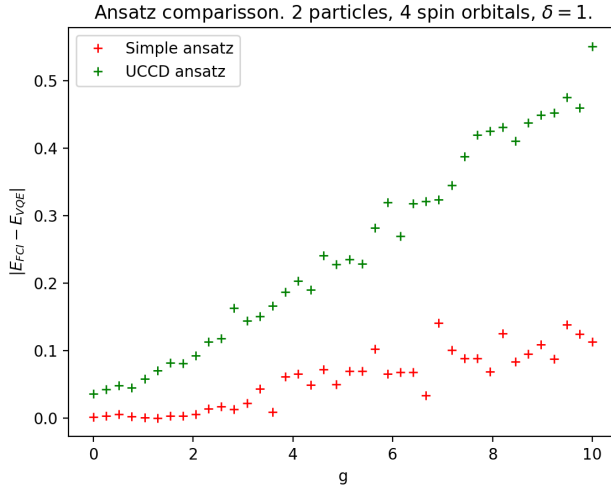
7.2.2 Noisy Simulation

We will try to reproduce the results for two particles and four spin-orbitals in figure 7.3a, but this time with the noise model from the IBM Q London 5 qubit computer. Both the UCCD ansatz (section 4.3.4) and the simple ansatz (circuit 4.60) will be compared, and we will also provide the results with and without error correction (section 6.1.7). The results can be seen in figure 7.5.

7. Results



(a) VQE with the UCCD ansatz. We plot the ground state energy against the interaction strength g , together with the absolute difference between the FCI energy and the UCCD energy. (b) VQE with the simple ansatz in circuit 4.60. We plot the ground state energy against the interaction strength g , together with the absolute difference between the FCI energy and the UCCD energy.



(c) Absolute energy difference between VQE and FCI energy for both ansatzes, against the interaction strength g . With error correction.

Figure 7.5: VQE on pairing Hamiltonian with two particles and four spin-orbitals. We utilize the noise model of the IBMQ London 5 qubit computer. We provide the results with and without error correction.

We see that unlike the ideal simulation seen in figure 7.3a, the results are now dependent on the interaction strength g . The error correction mitigates some of the noise, which can be seen from figures 7.5b and 7.5a. From figure 7.5c, we see that the deviation from the FCI energy increases quicker for the UCCD ansatz than when we are just applying the simple ansatz.

7.2.3 Execution on IBM Q London five qubit device

Unlike what we saw with the QPE algorithm (figures 7.2a and 7.2b), the VQE method was able to provide upper bounds for the ground state energy of the pairing Hamiltonian when simulating with noise. However, we found that the results were dependent on the interaction strength g (see figure 7.5). We will now run the VQE algorithm on the IBM Q London five qubit device to see if the results agree with the simulation with noise. We solve for two particles and four spin-orbitals with the simple ansatz (circuit 4.60), varying interaction strength g

and we utilize error correction. Hence, we would expect the results to resemble those seen from the simulation with noise in figure 7.5b. The results can be seen in figure 7.6.

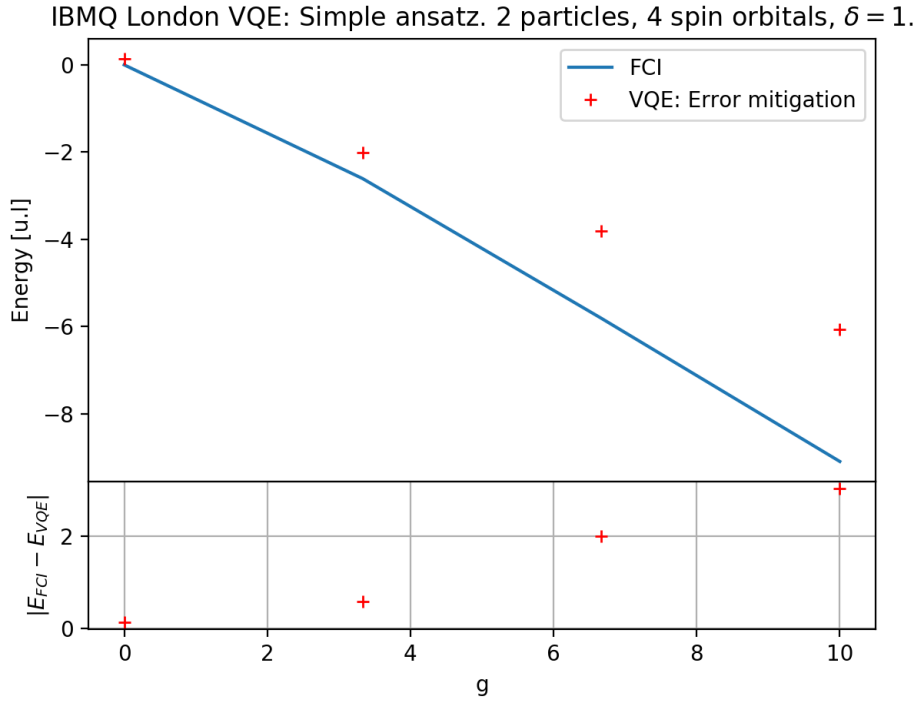


Figure 7.6: VQE algorithm on IBM Q London five qubit device for pairing Hamiltonian with two particles and four spin-orbitals. We plot the calculated ground state energy with FCI and VQE against the interaction strength g . We also plot as the absolute energy difference between FCI and VQE, against the interaction strength g . We utilized the simple ansatz provided by circuit 4.60.

As we saw for the simulation with noise in figure 7.5b, the absolute difference between the FCI energy and the VQE energy increases with increasing interaction strength g .

7.2.4 Discussion

Utilizing the VQE algorithm for the two particle, four spin-orbital system with an ideal simulation of a quantum computer showed that the method had promise when it comes to approximating the ground state energy of a Hamiltonian (see figure 7.3a). When running the simulation of a quantum computer with noise (figure 7.5) and running our algorithm on the actual IBM Q London device (figure 7.6), we learned that the noise caused our results to be dependent on the interaction strength g of our system. This may be explained by looking at figure 7.3b, showing the linear combination of qubit states in the found ground state. As we increase the interaction strength g , the ground state for the system is approaching the entangled state $\frac{1}{\sqrt{2}}(|0011\rangle \pm |1100\rangle)$. This may indicate that applying the VQE method on noisy intermediate-scale quantum (NISQ) devices is not robust when the ground state for the system in question is a highly entangled state, as we require sufficient entanglement between the qubits as well.

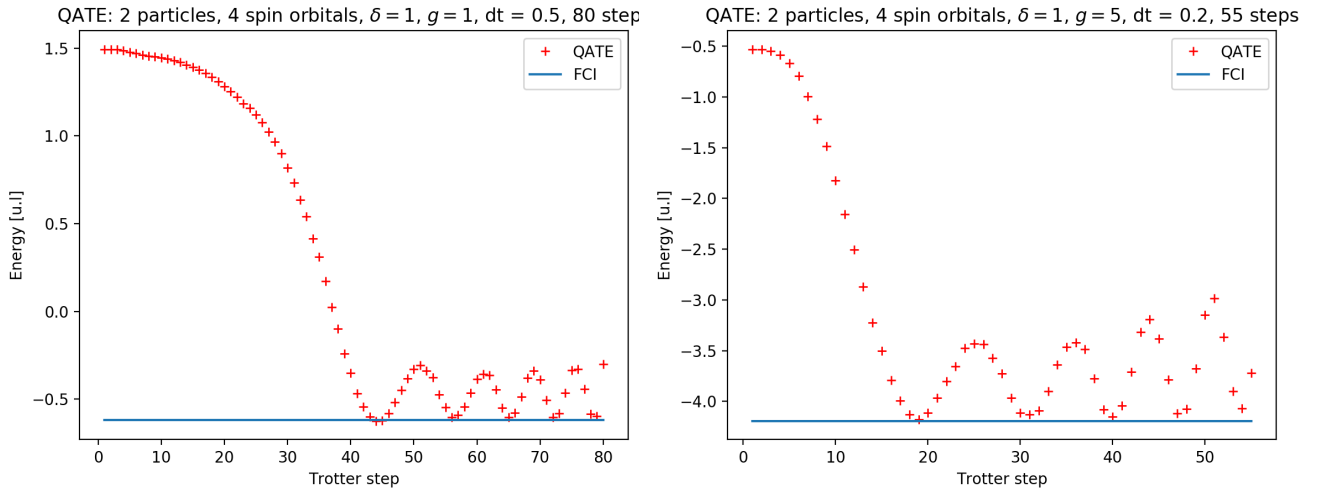
We also run the VQE method with the UCCD ansatz for four particles and eight spin-orbitals, with varying interaction strength g (figure 7.4). As the CCD energy was expected to deviate from the FCI energy for this system, we wanted to compare our VQE results with CCD. The energy given by both methods deviated from the FCI energy as we increased the interaction strength, but interestingly there was less deviation with VQE than CCD. This may be an excellent motivation for studying quantum computing.

7.3 Quantum Adiabatic Time Evolution

Recall that the quantum adiabatic time evolution (QATE) method requires us to start out in the ground state $|\psi_0\rangle$ of an initial Hamiltonian, \hat{H}_0 , and gradually change this Hamiltonian to the problem Hamiltonian, \hat{H}_1 . This results in a new, time dependent Hamiltonian (eq. 4.61) which can be implemented on a quantum computer utilizing numerical integration (eq. 4.64), the Suzuki-Trotter approximation (eq. 4.65) and finally circuit 4.35. In this section, we will see if this method can be used to approximate the ground state energy of the Jordan-Wigner transformed pairing Hamiltonian (eq. 4.26) with two particles and four spin-orbitals. The initial Hamiltonian will be put to the Hamiltonian in eq. 4.66, and our qubits will start in the corresponding ground state (eq. 4.67). First, we are going to test the method on a simulation of an ideal quantum computer, before running a simulation with the noise model of the IBM Q London five qubit device. In both cases, we will compare our results with the eigenvalue obtained with full configuration interaction (FCI) theory.

7.3.1 Ideal Simulation

For the ideal simulation, we first solved for the pairing Hamiltonian with an interaction strength $g = 1$. Then we increased the interaction strength to $g = 5$ to see if this affected the convergence of the algorithm. The results can be seen in figure 7.7.



(a) The interaction strength is put to $g = 1$. We plot the energy of the system as against the step of the time-ordered exponential. (b) The interaction strength is put to $g = 5$. We plot the energy of the system as against the step of the time-ordered exponential.

Figure 7.7: Ideal QATE simulation for pairing Hamiltonian with two particles and four spin-orbitals.

We see that the energy eventually starts oscillating around the eigenvalue given by FCI for both interaction strengths.

7.3.2 Noisy Simulation

To get an impression on how QATE fares when we include noise in the simulation, we utilized the noise model from the IBMQ London quantum computer and performed the same simulation as in figure 7.7b. The results can be seen in figure 7.8.

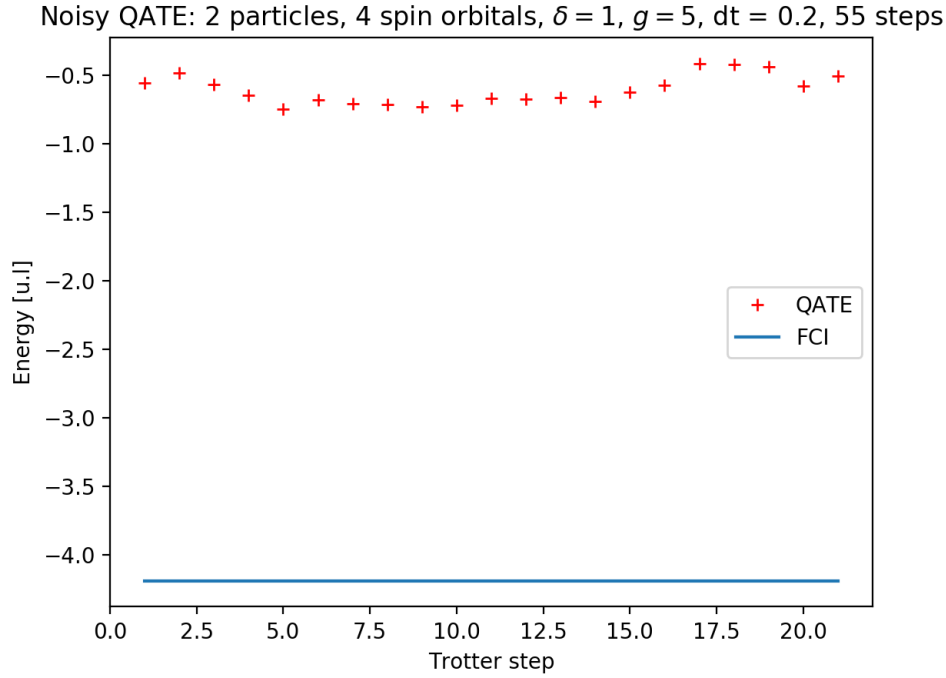


Figure 7.8: QATE simulation of pairing Hamiltonian with one pair and four spin-orbitals. The simulation is run with noise model from the five qubit IBMQ London quantum computer. We plot the energy of the system as against the step of the time-ordered exponential.

We see that we are not able to reproduce the results from the ideal simulation.

7.3.3 Discussion

In figure 7.7, we see an ideal simulation of the QATE algorithm performed on the pairing Hamiltonian with one pair and four spin-orbitals for two different interaction strengths. In both cases, the energy converges towards the FCI eigenvalue before starting to oscillate. This oscillation raises the question of when to stop the algorithm. We could measure the energy at each time step. However, this requires us to execute the algorithm several times. Even though each step can be implemented efficiently, it is important to note that the efficiency of the algorithm is reliant on the total number of time steps required. If this number is exponential in the number of qubits, the algorithm is no longer efficient. It can be shown that the number of time steps are dependent on the minimum energy difference between the two lowest states of the interpolating Hamiltonian [16]. This may be why we see a faster convergence when we increase the interaction strength (figure 7.7b). Evaluating the energy difference is beyond the scope of this thesis.

In figure 7.8 we show the QATE algorithm on a simulation with the noise model from the IBMQ London quantum computer. We see that we are not able to reproduce the results from the ideal simulation. Similar to the argument for the quantum phase estimation (QPE) algorithm (see section 7.1.3), this may be explained with the circuit depth given by continuous applications of the Suzuki-Trotter approximation (section 4.2.3).

7.4 Quantum Machine Learning

In this section we will utilize the machine learning methods explained in chapter 5 on some selected problems. In sections 5.10 and 5.11 we suggested ways to reduce a quantity called circuit depth (section 4.1.3) by minimizing a loss function with machine learning. The importance of reducing this quantity was explained by the fact that the circuit depth is often the bottleneck on noisy intermediate-scale quantum (NISQ) devices. The results in the previous sections of this chapter stresses the importance of circuits with short depth, as even computationally efficient methods such as the quantum phase estimation algorithm struggle when encountering noise (see figure 7.2). We will start by applying the method explained in section 5.10, which is based upon learning unitary operators by maximizing an inner product.

Recall that circuit 5.34 can be utilized to calculate the squared inner product between a state produced by some parametrized unitary operator $U_a(\boldsymbol{\theta})|0\cdots 0\rangle$, and the state produced by some other operator $U_i|0\cdots 0\rangle$. We argued that if the squared inner product between these two states is equal to one, the operators are only differing by a global phase. Hence, the two states are physically indistinguishable. Remembering that the operator $U_a(\boldsymbol{\theta})$ is dependent on the parameters $\boldsymbol{\theta}$, we can vary these until the squared inner product calculated with circuit 5.34 is as close to one as possible.

7.4.1 Learning the Suzuki-Trotter Approximation

We will first utilize this method to approximate a time step with the Suzuki-Trotter approximation (section 4.2.3). Recall that both the quantum phase estimation algorithm (section 4.2) and the quantum adiabatic time evolution (QATE) algorithm (section 4.4) was reliant on Suzuki-Trotter approximating the pairing Hamiltonian time evolution operator (eq. 4.31). Hence, we will try to approximate this operator for two particles and four spin-orbitals while hopefully reducing its circuit depth. The operator $U_a(\boldsymbol{\theta})$ was chosen to be the Euler rotation ansatz given by circuit 4.54. The parameter d is the number of successive applications of the ansatz, $U_a(\boldsymbol{\theta})$, to the quantum state, that is $U_a(\boldsymbol{\theta}_d)U_a(\boldsymbol{\theta}_{d-1})\cdots U_a(\boldsymbol{\theta}_1)|0\cdots 0\rangle$. Hence, we expect that the flexibility of the ansatz increases with d . We chose $d = 3$. The results from the learning process can be seen in figure 7.9.

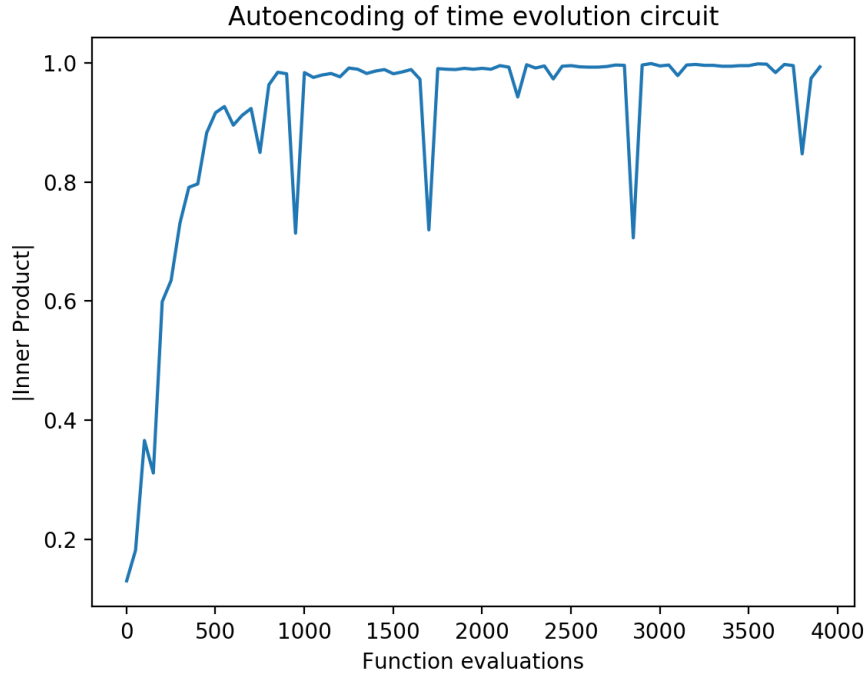


Figure 7.9: Approximating a time step of the Suzuki-Trotter approximation utilizing a parametrized ansatz. We plot the absolute value of the inner product between the state given by the Suzuki-Trotter approximation and the state given by the parametrized ansatz, against the iteration of the learning scheme.

We see from figure 7.9 that we are eventually able to learn a parametrized operator that approximates the Suzuki-Trotter approximation, since the inner product approaches 1. The circuit depth of the two operators is shown in the table below, along with the approximated squared inner product between the states produced by said operators:

Table 7.2: Squared inner product between the states produced by the Euler Rotation ansatz in circuit 4.54 and a Suzuki-Trotter step of the pairing Hamiltonian time evolution operator. We also show the circuit depth of the two circuits.

Time Evolution depth	Ansatz depth	Approximated squared inner product
91	16	1

Discussion

We see from table 7.2 that we were able to approximate the Suzuki-Trotter step while reducing the circuit depth of the operation from 91 to 16. Being able to achieve such a reduction in depth shows promise for this algorithm, especially considering that the Suzuki-Trotter approximation is utilized in all the quantum computing many-body methods discussed in this thesis. Even though we measured 1 for the squared inner product between the parametrized operator and the Suzuki-Trotter approximation, we have to keep in mind that this is only an approximation due to finite measurements.

7.4.2 Learning the Amplitude Encoding of Random Variables

We suggested in section 5.5 that we could utilize hardware-efficient parametrized unitary operators instead of the amplitude encoding operation (section 5.1) as the building blocks for quantum neural network layers. We can test if this is a viable option by following the same procedure as we did when approximating a Suzuki-Trotter

7. Results

step in section 7.4.1. Only this time, we instead try to approximate the amplitude encoding of a random vector \mathbf{x} . Recall that the amplitude encoding $U_{\mathbf{x}}$ of a vector \mathbf{x} encodes a 2^n -dimensional vector into the n -qubit state

$$U_{\mathbf{x}}|0\dots 0\rangle = \sum_i x_i |i\rangle,$$

by applying 2^n rotation gates. That is, the number of gates are exponential in the number of qubits. We will try to approximate the action of $U_{\mathbf{x}}$ with the Euler rotation ansatz given by circuit 4.54. This is a hardware-efficient ansatz with a number of gates linear in the number of qubits. The approximation of the amplitude encoding is done by maximizing the squared inner product between the amplitude encoding and the ansatz, by varying the parameters of the ansatz. We measure the squared inner product between these with circuit 5.34.

We compare the circuit depth of the amplitude encoding circuit with the depth of the Euler rotation ansatz in table 7.3.

Table 7.3: Comparison of the circuit depth of the amplitude encoder (circuit 5.8) to the depth of the Euler rotation ansatz (circuit 4.54). The parameter d is the number of successive applications of the ansatz, $U_a(\boldsymbol{\theta})$, to the quantum state, that is $U_a(\boldsymbol{\theta}_d)U_a(\boldsymbol{\theta}_{d-1})\dots U_a(\boldsymbol{\theta}_1)|0\dots 0\rangle$.

Qubits	Encoding depth	Ansatz depth ($d = 3$)	Ansatz depth ($d = 4$)	Ansatz depth ($d = 5$)
3	28	15	20	25
4	330	16	21	26
5	1386	17	22	27
6	4394	18	23	28

We see that the circuit depth of the amplitude encoder increases drastically when increasing the number of qubits, while the depth of the ansatz only increases by one per qubit.

In figure 7.10, we see the inner product between the learned ansatz and the amplitude encoding of a random vector, as a function of the dimensions of the random vector. The parameter d is the number of successive applications of the ansatz, $U_a(\boldsymbol{\theta})$, to the quantum state, that is $U_a(\boldsymbol{\theta}_d)U_a(\boldsymbol{\theta}_{d-1})\dots U_a(\boldsymbol{\theta}_1)|0\dots 0\rangle$. Hence, we expect that the flexibility of the ansatz increases with d .

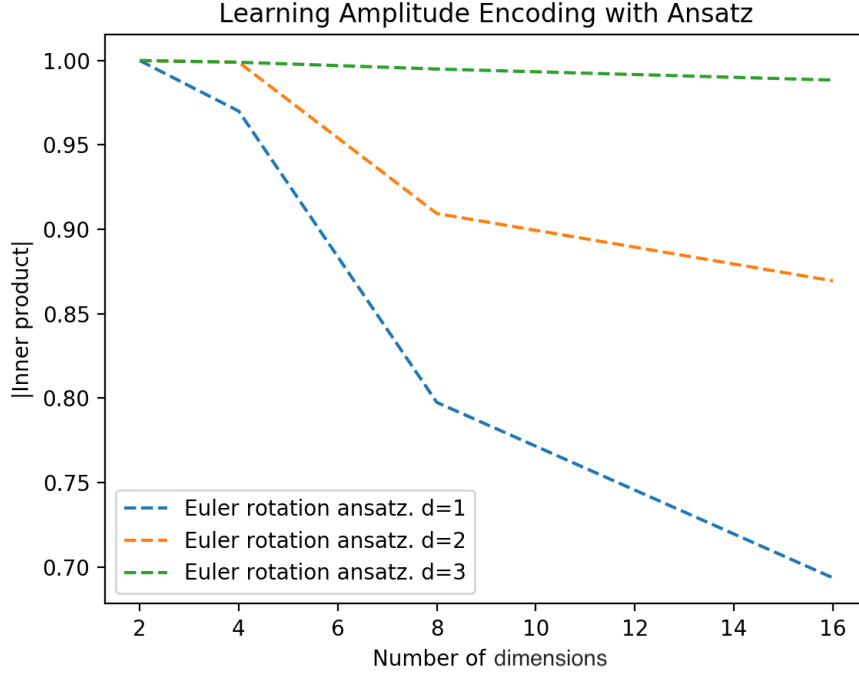


Figure 7.10: The inner product between state produced by a learned ansatz and the amplitude encoding of a normally distributed random vector, against the dimensions of the random vector. The ansatz was given by circuit 5.23. The parameter d is the number of successive applications of the ansatz.

We see from the green line that we were able to learn an ansatz that approximates the amplitude encoding, as the inner product is close to 1 for every dimension. Increasing the number of dimensions of the random vector requires increasing the flexibility of the ansatz, which we can see from the results for different d .

Discussion

The fact that we are able to learn the amplitude encoding operation with a hardware-efficient operator speaks well for the possibility to realize expressive and hardware-efficient neural network layers on quantum computers. In figure 7.10 we see that the squared inner product is close to 1 between the amplitude encoding of 16 random numbers and the Euler rotation ansatz with $d = 3$. As the number of qubits required for this amplitude encoding is 4, we see from table 7.3 that we have approximated the amplitude encoding operation while reducing its circuit depth from 330 to 16.

These results do not just serve as a motivation to further study hardware-efficient neural networks. They also suggest that one may be able to learn a hardware-efficient representation of a data set, which then could be used instead of the amplitude encoder at the start of some machine learning algorithm.

7.4.3 Recursive Circuit Optimization of Random Circuit

The issue with the method used in sections 7.4.1 and 7.4.2 to approximate unitary operators, is that we have to apply the operator we wish to approximate to the qubit-state. Hence, if the circuit depth of the operator is too large, the method will fail. We will now represent our results for the recursive circuit optimization scheme (section 5.11), which is a method we proposed to get past this issue.

A quantum circuit can be represented by some unitary operator U , which we can divide into k parts

$$U = U_k \cdots U_2 U_1.$$

7. Results

Recall that with the recursive circuit optimization scheme (see circuit 5.37), we first learn a parametrized unitary operator, $U_a(\theta_1)$, that approximates U_1 . The operator U can then be written as

$$U \approx U_k \cdots U_2 U_a(\theta_1).$$

We then approximate $U_2 U_a(\theta_1)$ with the parametrized unitary operator $U_a(\theta_2)$. This gives

$$U \approx U_k \cdots U_a(\theta_2).$$

After doing this for all k , we end up with

$$U \approx U_a(\theta_k).$$

Hence, we have approximated the full circuit by only applying smaller parts.

To test out this method, we generated a random three-qubit circuit with a depth of 200. The circuit was divided into $k = 20$ parts, each with a circuit depth of 10. We utilized the Euler rotation ansatz in circuit 4.54 as our parametrized operator $U_a(\theta)$. Recall that the parameter d is the number of successive applications of the ansatz, $U_a(\theta)$, to the quantum state. That is $U_a(\theta_k) = U_a(\theta_k^d) U_a(\theta_k^{d-1}) \cdots U_a(\theta_k^1) |0 \cdots 0\rangle$. We chose $d = 3$.

At each step of the recursive algorithm, we calculated the squared inner product between the random circuit and the learned ansatz. As we expected some error to propagate due to finite measurements in the recursive process, we ran the scheme for both 10^3 and 10^4 measurements. The results can be seen in figure 7.11.

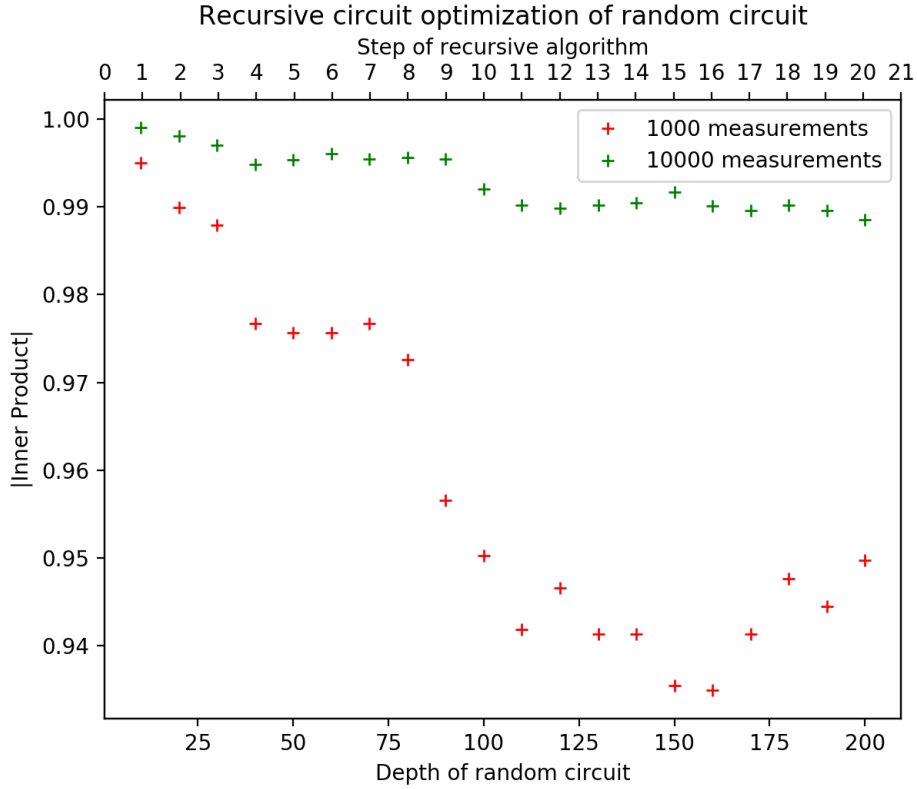


Figure 7.11: Recursive circuit optimization of random circuit. We plot the absolute value of the inner product between the random circuit state and the corresponding learned ansatz state, against the depth of the random circuit (bottom) and the step of the recursive algorithm (top). This is done for 10^3 and 10^4 measurements in the learning process.

We see that the absolute value of the inner product between the random unitary operation and the learned ansatz decreases with the steps of the recursive scheme. We also note that the decrease is steeper with fewer

measurements. In table 7.4, we show the circuit depth of the random circuit, the optimization scheme and the learned ansatz.

Table 7.4: Learning a random circuit with recursive circuit optimization. We compare the circuit depth of the full random circuit, the circuit depth of the recursive optimization scheme and the circuit depth of the learned ansatz.

Random Circuit Depth	Recursive Optimization Circuit Depth	Ansatz Circuit Depth
200	39	15

We see that we have approximated a circuit of depth 200 by only running circuits of depth 39. The resulting approximation of the random circuit has a depth of 15.

Discussion

From figure 7.11, we see that the inner product between our parametrized unitary operator and the respective part of the random circuit decreases as a function of the step of the algorithm. We expect that this behaviour comes as a result of two reasons. Since we are varying the parameters of an operator for it to maximize a squared inner product, the first reason could be that the operator is not flexible enough for this task. We can account for this by using a more flexible operator if the measured squared inner product is below 1. The second reason could be that we are doing finite measurements when we evaluate the squared inner product. Hence, even if the squared inner product is measured as 1, we are in reality not exactly reproducing the corresponding part of the random circuit. We will then see an error propagate as we further proceed with the recursive optimization. This can explain why we get a much better approximation when doing 10^4 measurements instead of 10^3 .

7.4.4 Recursive Circuit Optimization of the Quantum Adiabatic Time Evolution Algorithm

Since the method was successful on a random circuit, we wanted to see if it could reduce the circuit depth of some of the many-body methods utilized in this thesis. Recall that the quantum adiabatic time evolution (QATE) algorithm (section 4.4) requires us to utilize the Suzuki-Trotter approximation (section 4.2.3) to implement the time-ordered exponential. We wanted to recursively learn the Suzuki-Trotter approximation in eq. 4.65, that is

$$U(t) \approx \prod_{k=0}^n e^{-i[(1-\frac{k\Delta t}{T})\hat{H}_0 + \frac{k\Delta t}{T}\hat{H}_1]\Delta t},$$

where U_k in the recursive circuit optimization scheme (circuit 5.37) is the k 'th term in the above product. We utilized the $R_y(\theta)$ -rotation ansatz shown in circuit 4.50 as the parametrized operator $U_a(\theta)$.

The goal was to reproduce the results shown in figure 7.7b. We performed 10^3 measurements in the learning process and the results are shown in figure 7.12.

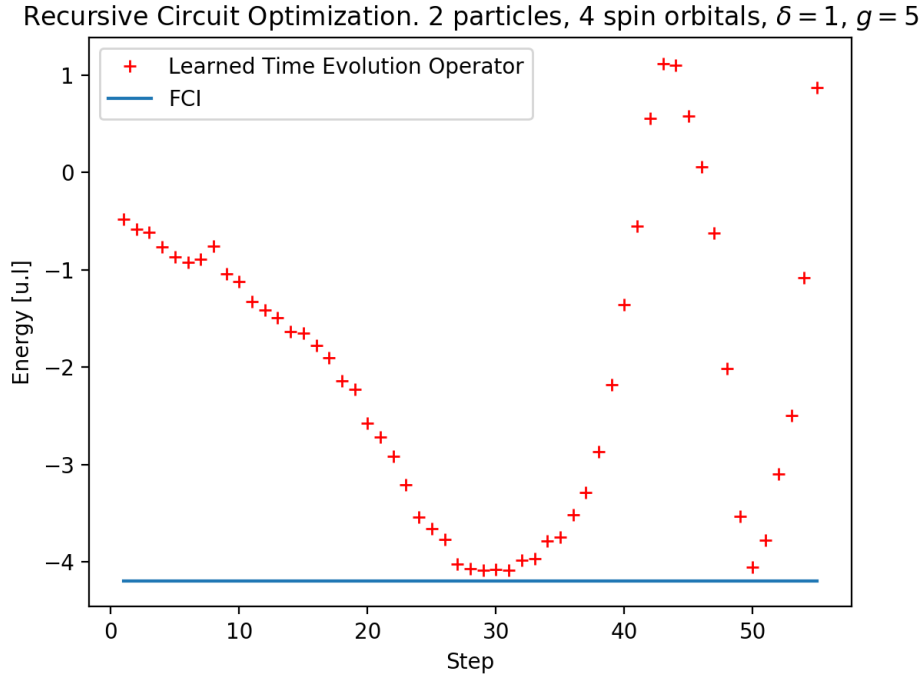


Figure 7.12: Recursive circuit optimization of the QATE algorithm for pairing Hamiltonian (eq. 4.26) with $\delta = 1$ and $g = 1$. We plot the energy as a function of the step in the recursive scheme.

Even though this is not an exact reproduction of what is seen in figure 7.7b, we are still able to approximate the ground state energy.

The circuit depth of the recursive scheme and the circuit depth of the corresponding QATE algorithm is shown in table 7.5.

Table 7.5: Depth of recursive Circuit optimization scheme versus depth of the QATE algorithm.

QATE circuit depth	Recursive Optimization circuit depth
5061	126

We see that we have approximated the QATE algorithm while reducing the circuit depth from 5061 to 126.

Discussion

From figure 7.12 we see that the recursive optimization scheme was not able to exactly reproduce the results from the corresponding QATE algorithm in figure 7.7b. Like we discussed when optimizing the random circuit (section 7.4.3), we expected some error in the approximation due to finite measurements and due to the flexibility of the parameterized operator. Hence, we expect that increasing the number of measurements could provide more satisfactory results. Even though we are not able to exactly reproduce the QATE algorithm, we are still able to approximate the ground state energy while reducing the circuit depth from 5061 to 126 (table 7.5). This drastic decrease of circuit depth could potentially lead to the method being applicable on near-term devices.

7.4.5 Recursive Circuit Optimization of Unitary Coupled Cluster Doubles Ansatz

Finally, we wanted to see if we could utilize the recursive scheme to reduce the circuit depth of the variational quantum eigensolver (VQE) algorithm (section 4.3). Recall that the unitary coupled cluster doubles (UCCD)

ansatz requires us to apply the following Suzuki-Trotter approximation (eq. 4.58)

$$U(t) \approx \left(\prod_{ijab} e^{\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_x^j \sigma_y^a \sigma_y^b} e^{\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_x^a \sigma_x^b} e^{\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_x^j \sigma_y^a \sigma_y^b} e^{\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_x^a \sigma_x^b} \right. \\ \left. e^{-\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_x^j \sigma_x^a \sigma_y^b} e^{-\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_y^j \sigma_x^a \sigma_y^b} e^{-\hat{Z}_{ij}^{ab} \sigma_y^i \sigma_y^j \sigma_x^a \sigma_x^b} e^{-\hat{Z}_{ij}^{ab} \sigma_x^i \sigma_y^j \sigma_y^a \sigma_x^b} \right).$$

To reduce the circuit depth of the VQE algorithm, we divided the above ansatz into $k = 2$ parts. Every time we would apply the ansatz to our state, we utilized the recursive circuit optimization scheme (circuit 5.37) to approximate it with the $R_y(\theta)$ -rotation ansatz (circuit 4.50). We solved for the Jordan-Wigner transformed pairing Hamiltonian (eq. 4.26) with two particles and four spin-orbitals, varying the interaction strength g . Hence, we expect to reproduce the results seen in figure 7.3a.

The results can be seen in figure 7.13.

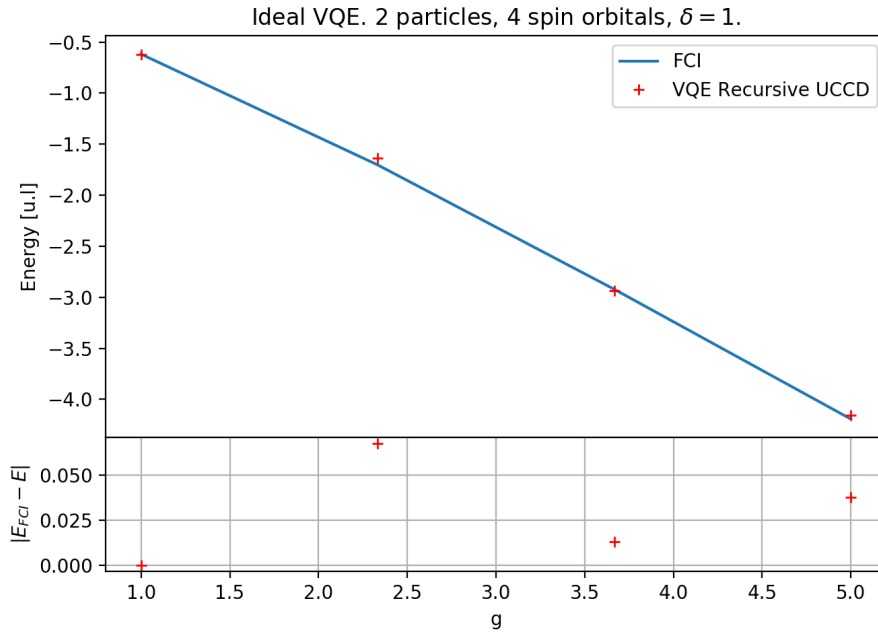


Figure 7.13: The VQE algorithm on the pairing Hamiltonian with two particles and four spin orbitals. We approximated the UCCD ansatz utilizing the recursive circuit optimization scheme with two steps and the R_y -rotation ansatz given by circuit 4.50. We plot the ground state energy against the interaction strength g , together with the absolute difference between the FCI energy and the UCCD energy.

We see that we are able to reproduce the results in figure 7.3a without sacrificing accuracy.

In table 7.6, we see the depth of running the recursive scheme vs. the depth of the VQE algorithm with the UCCD ansatz.

Table 7.6: Recursive optimization scheme of the VQE algorithm with the UCCD Ansatz. We see the depth of the UCCD ansatz and the depth of recursive optimization scheme.

UCCD depth	Recursive circuit depth
245	76

Hence, we have reduced the circuit depth of the VQE algorithm from 245 to 76.

Discussion

In figure 7.5c, we saw the VQE algorithm simulated with the noise model of one of IBM's real quantum computers. When comparing the UCCD ansatz with a much simpler ansatz, the error in the energy estimate was larger for the former. Hence, reducing the circuit depth of the UCCD ansatz could be beneficial. The results in figure 7.13 show that we are able to do this, while achieving an accuracy comparable to what was seen with the VQE algorithm results. Table 7.6 shows that we reduced the circuit depth of the ansatz from 245 to 76.

This concludes the results for the recursive optimization scheme. We will now move over to our results for the quantum neural networks.

7.4.6 Learning a Non-Linear Function with Quantum Neural Network

To test out the quantum neural network proposed in section 5.5, we will try to learn the following non-linear function:

$$f(x) = 3 \sin x - \frac{1}{2}x.$$

We generated ten points to be utilized as training data. These points were evenly distributed in the interval $x \in [0, 2\pi]$. We normalized $f(x)$ between 0 and 1 since the output from the neural network lies in this range.

Recall from section 5.5 that our neural network layer (circuit 5.17) consists three main components. An encoder, $U_{enc}(\mathbf{x})$, which is utilized to encode some representation of the layer inputs \mathbf{x} to a quantum state. An ansatz, $U_a(\theta_a)$, with the purpose of parametrizing the encoded quantum state. And finally the entangler, $U_{ent}(\theta_{ent})$, which should entangle the register acted upon by the encoder and ansatz, with an ancilla qubit.

Table 7.7 shows the structure of the neural network utilized for this problem.

Table 7.7: Structure of the neural network used to approximate a non-linear function. The first row corresponds to the input layer, while the final row corresponds to the output layer.

Qubits	U_{enc}	U_a	U_{ent}	Outputs
1	$R_y(\theta)$ (eq. 4.6)	$R_y(\theta)$	circuit 5.30	6
3	circuit 5.23 with $U^{[j,k]} = R_y(\theta)$	circuit 5.23 with $U^{[j,k]} = R_y(\theta)$	circuit 5.30	6
3	circuit 5.23 with $U^{[j,k]} = R_y(\theta)$	circuit 5.23 with $U^{[j,k]} = R_y(\theta)$	circuit 5.30	1

Going through the structure in table 7.7, the neural network only consists of hardware-efficient encoders, ansatzes and entanglers. The amplitude encoder is not utilized. The importance of this is discussed in section 5.5.

For the training algorithm we utilized the mean squared error (eq. 3.2) as the loss function. To evaluate the trained network, we generated twenty points to make sure that the network could to some extent approximate the function when passed points not coinciding with the training data. These twenty points were also evenly distributed in the interval $x \in [0, 2\pi]$. The results can be seen in figure 7.14.

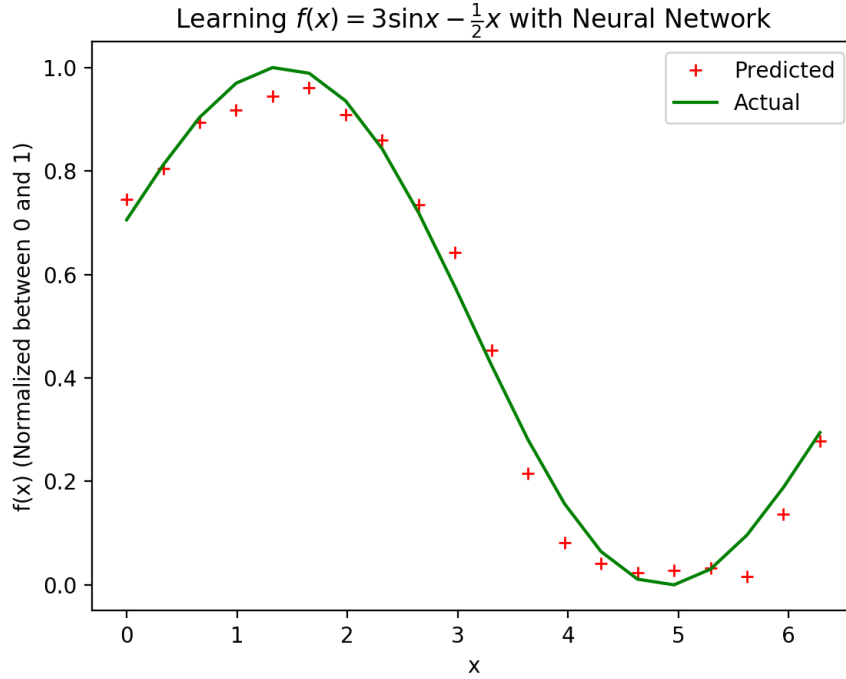


Figure 7.14: Quantum Neural Network trained on the non-linear function $f(x) = 3\sin x - \frac{1}{2}x$. We plot the actual function values and the values given by the neural network, against x .

We see that even for this hardware-efficient network, we are able to approximate a non-linear function.

Discussion

Figure 7.14 shows a plot of the true function together with the approximation given by the neural network. We see that the quantum neural network is able to learn from a data set and also that it is able to learn non-linearity. Even though we are not perfectly replicating the function, we have to keep in mind that we only utilized a three-layered network, with at most three qubits in the intermediate layers (see table 7.7). We expect that adding more layers and/or more qubits will increase the flexibility. Another importance is that the model was not reliant on amplitude encoding (section 5.1), as we utilized the output from the layers as rotation angles for a hardware-efficient unitary encoder in the next layer (see section 5.5.1). In fact, all the components was chosen to be hardware-efficient and suitable for noisy intermediate-scale quantum devices. Hence, we have have illustrated that one could realize hardware-efficient neural network layers on quantum computers that are able to learn non-linear functions.

7.4.7 Rayleigh Quotient Minimization

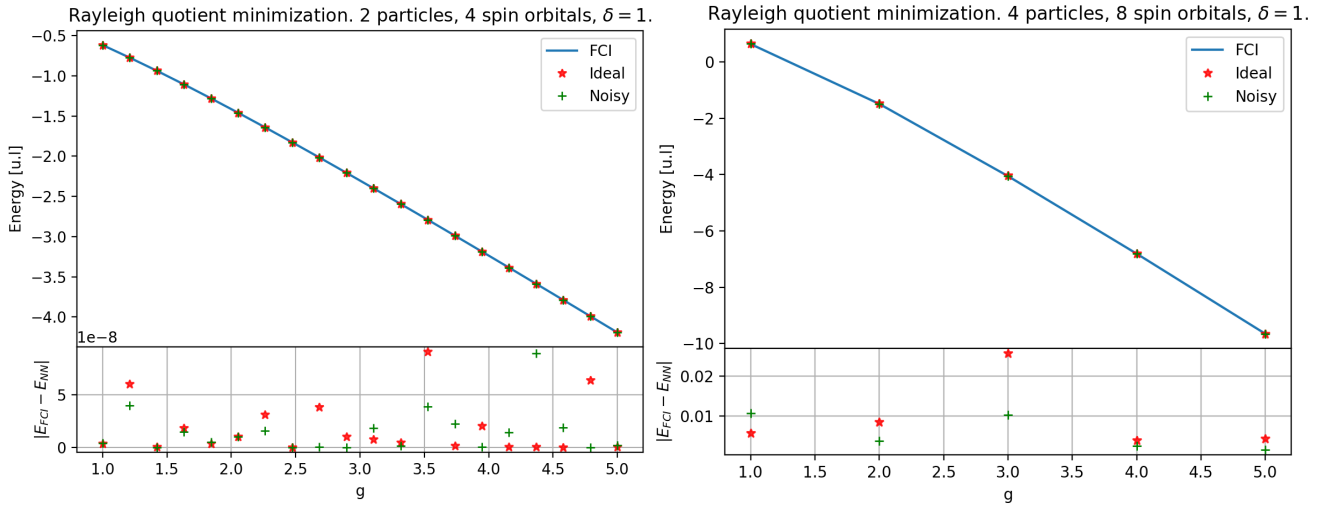
We wanted to utilize a neural network to approximate the lowest eigenvalue of the full configuration (FCI) pairing Hamiltonian matrix (see section 2.5). We did this by minimizing the Rayleigh Quotient as described in section 5.12.1. The network we utilized had two layers given by circuit 5.17. The structure can be seen in table 7.8.

7. Results

Table 7.8: Structure of the neural network used to approximate the eigenvalues of the pairing Hamiltonian. The first row corresponds to the input layer, while the final row corresponds to the output layer. Identity means that we do not apply any operator. In the final layer, we return a number of outputs equal to the dimension of the eigenvector.

Qubits	U_{enc}	U_a	U_{ent}	Outputs
2	Hadamard gate on each qubit (eq. 4.5)	Identity	circuit 5.30	2
2	circuit 5.23 with $U^{[j,k]} = R_y(\theta)$	Identity	circuit 5.30	Dimensions of eigenvector

We test out the neural network for both an ideal simulation of a quantum computer, and a simulation with the noise model from the IBM Q London 5 qubit device. We solved for both a two particle, four spin-orbital system, as well as a four particle, eight spin-orbital system. The results can be seen in figure 7.15.



(a) Two particles and four spin-orbitals. We plot the energy approximation against the interaction strength g . (b) Four particles and eight spin-orbitals. We plot the energy approximation against the interaction strength g .

Figure 7.15: Rayleigh Quotient minimization of pairing Hamiltonian. We plot the Rayleigh quotient against the interaction strength g . We include the results from both an ideal simulation and a simulation of the IBMQ London 5 qubit quantum computer.

We see that we are able to approximate the ground state energy for both systems when running the ideal simulation, as well as when running a simulation with the noise model from the IBM Q London 5 qubit device.

Discussion

In figure 7.15, we see that we are able to approximate the eigenvalues of the FCI matrix by utilizing our proposed neural network architecture. We can also see that the results are not affected by running a simulation with the noise model of the IBM Q London 5 qubit computer. Even though this neural network architecture (table 7.8) is simpler than the architecture we used when approximating the non-linear function (table 7.7), these results supports the idea that these neural networks could be utilized on near-term devices.

7.4.8 Final Discussion on the Neural Networks

Hardware-efficiency

We know that except of the amplitude encoder, the encoders and ansatzes utilized in this thesis (section 5.7) are hardware-efficient and suitable for near-term quantum computers [32]. We can therefore argue that the intermediate layers of our neural networks are hardware-efficient as well, as we have illustrated that we do not

need to use the amplitude encoder for these. For the first layer however, we have to encode some representation of the input data. The most natural way to do this is through amplitude encoding. We have discussed that this may not be feasible on near-term devices (section 5.5). One way to get past this is by utilizing a hybrid quantum-classical neural network. The input layer could be calculated purely classical, and the outputs from this layer are scaled and input to a hardware-efficient quantum layer (see section 5.5.1). Another possibility is to approximate the amplitude encoding of the data set with the Recursive Circuit Optimization scheme (section 5.11).

Exponential advantage in the number of parameters

A classical neural network layer of p inputs and k outputs requires pk (eq. 5.22) parameters to realize. The quantum layers would in the same scenario require $\mathcal{O}(k \log p)$ parameters (eq. 5.27). Hence, there is an exponential advantage in terms of the number of parameters for the quantum neural network proposed in section 5.5.

Many of the best performing classical neural networks require an enormous amount of memory due to the number of weights [5]. This memory requirement can often be the bottleneck for a model. The exponential advantage in the number of weights for the quantum neural networks may allow for deeper neural networks without an excessive memory requirement.

Expressiveness of the Quantum Neural Networks

A natural question that arises from the fact that we have exponentially fewer weights, is whether or not such neural networks will have the same expressive power as its counterpart. None of the work in this thesis gives a clear answer to this question, however one can consider the research that are being done on weight pruning. Weight pruning is the task of systematically removing parameters from a classical neural network with an acceptable decrease of accuracy. A survey of 81 recent papers on weight pruning concludes that it does indeed work, and in fact sometimes can lead to an increase of model accuracy[5]. This could be an indication that it is possible to realize useful layers with less parameters.

There are also several widely used methods to reduce the expressive power of neural networks [27]. The reason for this is that neural networks are often too expressive and will start fitting to the noise in a data set. Hence, parameter-efficiency may not be a caveat when it comes to the proposed quantum neural networks.

CHAPTER 8

Conclusion

8.1 What we have done

We have implemented three quantum computing many body methods: the quantum phase estimation algorithm (QPE), the variational quantum eigensolver (VQE) and the quantum adiabatic time evolution (QATE), and we have illustrated their functionality by solving for the ground state energy of a simple pairing Hamiltonian. Even though the discussed methods are computationally efficient, we have illustrated that there still are challenges in executing them on current quantum hardware. We showed this by simulating the noise from real IBM quantum computers, as well as executing the VQE algorithm on a real quantum device. We saw that we in these cases were not able to exactly reproduce the results we got in the ideal simulation. We found that the VQE algorithm was the only method which subject to noise could, to some extent, approximate the ground state energy of our system. Even still, we found that increasing the interaction strength in the Hamiltonian would increase the error in the approximation.

Circuit depth [19] is a quantity that is describing the number of time steps required to perform a quantum circuit. For noisy intermediate-scale quantum (NISQ) devices, the ability to achieve sensible results relies on the execution time of the algorithm, hence the circuit depth is important. We have illustrated how one could utilize machine learning to approximate circuits while decreasing their depth. We did this by maximizing the squared inner product between a state produced by some parametrized operator and the state produced by the circuit we wished to approximate. As this method required us to run the full circuit for the operator we wished to simplify, which may not be possible on near-term quantum computers, we proposed a circuit optimization scheme which removes this requirement, namely Recursive Circuit Optimization. We provided a proof-of-concept calculation for this method by approximating a random circuit of depth 200 with a parametrized circuit of depth 15. The optimization scheme required us to execute a circuit of depth 39, hence we never had to run the full 200 depth circuit. We then took this proof-of-concept a step further and utilized it to approximate the QATE algorithm as well as the VQE method with the unitary coupled cluster doubles (UCCD) ansats. Although not a perfect replicate of the QATE algorithm, we showed that we were able to get a lower bound on the ground state energy while reducing the QATE circuit depth from 5061 to 126. For the VQE method we were able to accurately approximate the full circuit while reducing the circuit depth from 245 to 76.

We implemented a classical neural network on a quantum computer by the means of amplitude encoding a data set and a set of parameters, as proposed in reference [52]. Inspired by this method, we proposed a way to realize a recurrent neural network in the same manner, by rewriting a recurrent neural network node calculation in terms of an inner product (section 5.4). Next, we proposed a circuit for quantum-specific dense (section 5.5) and recurrent (section 5.6) layers by utilizing parametrized unitary operators instead of the amplitude encoding, as the task of amplitude encoding is known as infeasible for large data sets on NISQ devices. The proposed neural networks are parameter-efficient when compared to their classical counterpart, and the intermediate layers are hardware-efficient and suitable for NISQ devices. We also proposed a way of calculating nodes in parallel by the addition of one ancilla qubit per node, and a number of gates linear in the number of qubits (section 5.9). As a means of illustrating the expressiveness of the proposed neural network layers, we showed for small quantum systems that the parametrized unitaries were able to produce a state close to the state produced by the amplitude encoding of random numbers. The circuit depth of the parametrized unitaries were in addition much

smaller than the circuit depth of the corresponding amplitude encoding. We then showed that we were able to approximate a non-linear function utilizing a parameter-efficient and hardware-efficient three-layered neural network of the kind proposed in this thesis. Finally, we utilized a two-layered quantum neural network to calculate the smallest eigenvalue of the Pairing Hamiltonian. We showed that the neural network was successful when simulation a quantum computer with the noise model from one of IBM's real quantum computers.

8.2 Further Studies

8.2.1 Hamiltonian Simulation

All the quantum many-body methods in this thesis are reliant on applying the time evolution operator of some Hamiltonian. We did this by rewriting the Hamiltonian in terms of a product of smaller operators by utilizing the Suzuki-Trotter approximation. This task is called Hamiltonian simulation. There have been several advances in the field of Hamiltonian simulation, with current state of the art being Hamiltonian simulation by Quantum Signal Processing [30]. This method reduces the gate complexity in terms of the simulation time t and approximation error ϵ [9]. It would be interesting to implement this method instead of the Suzuki-Trotter approximation.

8.2.2 Hamiltonian Transformation

We wrote the second quantization Hamiltonian in terms of quantum gates by applying the Jordan-Wigner transformation. This transformation allows for the representation of a Fermionic operator on n qubits with $\mathcal{O}(n)$ operations. There is another transformation, namely the Bravyi-Kitaev transformation which can reduce this to $\mathcal{O}(\log n)$ operations [6]. We could potentially further reduce the computation complexity of the many-body methods with this transformation.

8.2.3 Numerical integration method in Quantum Adiabatic Time Evolution

We utilized the midpoint method to approximate the integral in eq. 4.62, and it would be interesting to see if other numerical integration method could provide faster convergence or better results for the quantum adiabatic time evolution method.

8.2.4 Neural Networks

We only approximated a non-linear function dependent on one parameter as the test-case for our hardware-efficient neural network layers. It would be interesting to do this for more complicated functions, dependent on several parameters. Also, we did not add any noise to the data set and we would like to test this to see how the neural network handles noisy data.

8.2.5 Learning schemes

All the machine learning methods in this thesis relies on minimizing a loss function $L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta})$ by varying the model parameters $\boldsymbol{\theta}$. We did this by utilizing a classical optimization algorithm provided by the Scipy [48]. It would be interesting to see if one could propose an efficient circuit to calculate the gradients of the loss function and update the parameters utilizing gradient descent. It would also be interesting to try to approximate the gradients by utilizing numerical differentiation.

8.2.6 Error estimate for Recursive Circuit Optimization scheme

For some methods, the difference between the state produced by the recursive optimization scheme and the state produced by the approximated operator is not very important to quantify. If one wishes to utilize the scheme to approximate the amplitude encoding of some data set, one could argue that the performance of the machine learning model on a separate test set trumps any error in the approximation. For other methods however, we do not have the luxury of checking the performance of our algorithm by testing it out on the problem it was created for. Hence, it would be beneficial to research analytical expressions for errors in the approximation.

8.2.7 Error Correction

Error correction will play a central role in the realisation of quantum computing [7]. The quantum state we are manipulating is sensitive to imperfections in the computer and also to noise from the environment [47]. Error correction is a means of protecting the outcome of our algorithm from these factors so we can achieve sensible results. We did not study any such method in this thesis, but it is definitely a topic worth delving into for further studies.

Appendices

APPENDIX A

CCD Amplitude Equations

A.1 Calculations of amplitude equations

The first amplitude term, $\langle \Phi_{ij}^{ab} | \hat{F} \hat{T}_2 | c \rangle$, will be carefully calculated using Wick's generalized theorem and symmetry arguments. Thereafter we will turn to the other terms and skip the contraction lines.

$$\begin{aligned}
 \langle \Phi_{ij}^{ab} | \hat{F} \hat{T}_2 | c \rangle &= \frac{1}{4} \sum_{pq} \sum_{cdkl} f_q^p t_{kl}^{cd} \langle c | \{a_i^\dagger a_j^\dagger a_b a_a\} \{a_p^\dagger a_q\} \{a_c^\dagger a_d^\dagger a_l a_k\} | c \rangle \\
 &= \frac{1}{4} \sum_{pq} \sum_{cdkl} f_q^p t_{kl}^{cd} \hat{P}(ij) \hat{P}(ab) \hat{P}(cd) \langle c | a_i^\dagger a_j^\dagger a_b a_a a_p^\dagger a_q^\dagger a_c^\dagger a_d^\dagger a_l a_k | c \rangle \\
 &\quad + \frac{1}{4} \sum_{pq} \sum_{cdkl} f_q^p t_{kl}^{cd} \hat{P}(ij) \hat{P}(ab) \hat{P}(lk) \langle c | a_i^\dagger a_j^\dagger a_b a_a a_p^\dagger a_q^\dagger a_c^\dagger a_d^\dagger a_l a_k | c \rangle \\
 &= \frac{1}{4} \sum_{pq} \sum_{cdkl} f_q^p t_{kl}^{cd} \hat{P}(ij) \hat{P}(ab) \left[\delta_{ik} \delta_{jl} \delta_{bd} \delta_{ap} \delta_{qc} - \delta_{ik} \delta_{jl} \delta_{bc} \delta_{ap} \delta_{qd} \right. \\
 &\quad \left. - \delta_{ik} \delta_{jq} \delta_{bd} \delta_{ac} \delta_{pl} + \delta_{il} \delta_{jq} \delta_{bd} \delta_{ac} \delta_{pk} \right] \\
 &= \frac{1}{4} \sum_c f_c^a t_{ij}^{cb} \hat{P}(ij) \hat{P}(ab) - \frac{1}{4} \sum_d f_d^a t_{ij}^{bd} \hat{P}(ij) \hat{P}(ab) \\
 &\quad - \frac{1}{4} \sum_l f_j^l t_{il}^{ab} \hat{P}(ij) \hat{P}(ab) + \frac{1}{4} \sum_k f_j^k t_{ki}^{ab} \hat{P}(ij) \hat{P}(ab) \\
 &= \frac{1}{2} \sum_c f_c^a t_{ij}^{cb} \hat{P}(ij) \hat{P}(ab) - \frac{1}{2} \sum_k f_j^k t_{ik}^{ab} \hat{P}(ij) \hat{P}(ab) \\
 &= \sum_k f_i^k t_{jk}^{ab} \hat{P}(ij) - \sum_c f_c^a t_{ij}^{bc} \hat{P}(ab).
 \end{aligned}$$

$$\begin{aligned}
 \langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2 | c \rangle &= \frac{1}{16} \sum_{pqrs} \sum_{cdkl} t_{kl}^{cd} \langle pq | \hat{v} | rs \rangle \langle c | \{a_i^\dagger a_j^\dagger a_b a_a\} \{a_p^\dagger a_q^\dagger a_s a_r\} \{a_c^\dagger a_d^\dagger a_l a_k\} | c \rangle \\
 &= \frac{1}{16} \sum_{pqrs} \sum_{cdkl} t_{kl}^{cd} \langle pq | \hat{v} | rs \rangle \left[\hat{P}(ij) \hat{P}(ab) \hat{P}(pq) \delta_{ir} \delta_{js} \delta_{bd} \delta_{ac} \delta_{pk} \delta_{ql} \right. \\
 &\quad \left. + \hat{P}(ij) \hat{P}(ab) \hat{P}(rs) \delta_{ik} \delta_{jl} \delta_{bq} \delta_{ap} \delta_{sd} \delta_{rc} \right. \\
 &\quad \left. - \hat{P}(ij) \hat{P}(ab) \hat{P}(pq) \hat{P}(sr) \hat{P}(cd) \hat{P}(lk) \delta_{ik} \delta_{js} \delta_{bd} \delta_{ap} \delta_{ql} \delta_{rc} \right] \\
 &= \frac{1}{16} \hat{P}(ij) \hat{P}(ab) \left[\sum_{cd} t_{ij}^{cd} (\langle ab | \hat{v} | cd \rangle - \langle ab | \hat{v} | dc \rangle) \right] \\
 &\quad + \frac{1}{16} \hat{P}(ij) \hat{P}(ab) \left[\sum_{kl} t_{kl}^{ab} (\langle kl | \hat{v} | ij \rangle - \langle lk | \hat{v} | ji \rangle) \right] \\
 &\quad - \frac{1}{16} \hat{P}(ij) \hat{P}(ab) \left[\sum_{lc} t_{il}^{cb} \langle al | \hat{v} | cj \rangle - \sum_{kc} t_{ki}^{cb} \langle ak | \hat{v} | cj \rangle \right. \\
 &\quad \left. - \sum_{ld} t_{il}^{bd} \langle al | \hat{v} | dj \rangle + \sum_{kd} t_{ki}^{bd} \langle ak | \hat{v} | dj \rangle \right. \\
 &\quad \left. - \sum_{lc} t_{il}^{cb} \langle al | \hat{v} | jc \rangle + \sum_{kc} t_{ki}^{cb} \langle ak | \hat{v} | jc \rangle \right. \\
 &\quad \left. + \sum_{ld} t_{il}^{bd} \langle al | \hat{v} | jd \rangle - \sum_{kd} t_{ki}^{bd} \langle ak | \hat{v} | jd \rangle \right. \\
 &\quad \left. + \sum_{lc} t_{il}^{cb} \langle la | \hat{v} | jc \rangle - \sum_{kc} t_{ki}^{cb} \langle ka | \hat{v} | jc \rangle \right. \\
 &\quad \left. - \sum_{ld} t_{il}^{bd} \langle la | \hat{v} | jd \rangle + \sum_{kd} t_{ki}^{bd} \langle ka | \hat{v} | jd \rangle \right. \\
 &\quad \left. - \sum_{lc} t_{il}^{cb} \langle la | \hat{v} | cj \rangle + \sum_{kc} t_{ki}^{cb} \langle ka | \hat{v} | cj \rangle \right. \\
 &\quad \left. + \sum_{ld} t_{il}^{bd} \langle la | \hat{v} | dj \rangle - \sum_{kd} t_{ki}^{bd} \langle ka | \hat{v} | dj \rangle \right] \\
 &= \frac{1}{2} \sum_{cd} \langle ab | \hat{v} | cd \rangle t_{ij}^{cd} + \frac{1}{2} \sum_{kl} \langle kl | \hat{v} | ij \rangle t_{kl}^{ab} + \hat{P}(ab) \hat{P}(ij) \sum_{kc} \langle ak | \hat{v} | ic \rangle t_{jk}^{bc}.
 \end{aligned}$$

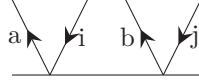
The last term we need to calculate, is the following

$$\langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2^2 | c \rangle = \frac{1}{64} \sum_{pqrs} \sum_{cdkl} \sum_{efmn} t_{kl}^{cd} t_{mn}^{ef} \langle pq | \hat{v} | rs \rangle \langle c | \{a_i^\dagger a_j^\dagger a_b a_a\} \{a_p^\dagger a_q^\dagger a_s a_r\} \{a_c^\dagger a_d^\dagger a_l a_k\} \{a_e^\dagger a_f^\dagger a_n a_m\} | c \rangle,$$

which can be calculated in the same way as the previous terms. However, it is a tedious affair since there are hundreds of ways we can draw the contraction lines. Even though the permutation operators give some simplifications, it is not a tempting exercise to do. Instead, we turn to a diagrammatic representation [49] which is known to be more efficient. First, let us recall the operators

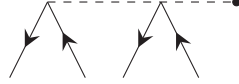
$$\begin{aligned}
 \hat{T}_2 &= \frac{1}{4} \sum_{abij} t_{ij}^{ab} a_a^\dagger a_b^\dagger a_j a_i \\
 \hat{H}_I &= \frac{1}{4} \sum_{pqrs} \langle pq | \hat{v} | rs \rangle \{a_p^\dagger a_q^\dagger a_s a_r\}.
 \end{aligned}$$

As we can see, \hat{T}_2 creates two particles and annihilates two holes, and can by a diagram be represented as



with the convention that particles point upwards and holes point downwards. Further, \hat{H}_I is initially not restricted, i.e, the operators can either be particles or holes.

Nevertheless, the only terms that do contribute, is when the operator annihilates two particles and creates two holes, corresponding to two particles lines into the vertex and two hole lines out from the vertex:



We will henceforth skip labeling of lines. The diagrams that represent the operator $\hat{H}_I \hat{T}_2 \hat{T}_2$ is simply all unique connections between the operators with hole lines on hole lines and particle lines on particle lines. It turns out to be five unique connections,

$$\begin{aligned}
 \langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2^2 | c \rangle = & \text{Diagram 1} + \text{Diagram 2} \\
 & + \text{Diagram 3} + \text{Diagram 4} \\
 & + \text{Diagram 5}
 \end{aligned}$$

which can be represented diagrammatically by

$$\begin{aligned}
\langle \Phi_{ij}^{ab} | \hat{H}_I \hat{T}_2^2 | c \rangle = & \text{Diagram 1} + \text{Diagram 2} \\
& + \text{Diagram 3} + \text{Diagram 4} \\
& + \text{Diagram 5}
\end{aligned}$$

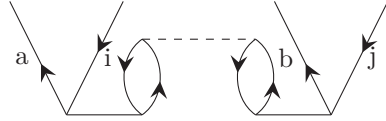
To convert the diagrams to ordinary equations, we need to introduce some diagrammatic rules. We will focus on the rules that we use for those specific diagrams.

1. The sign is found from the rule $(-1)^{h-l}$ where h is the total number of hole lines, and l is the total number of loops (internal and external). External loops are loops that are made with a line going from $ijk \dots$ to $abc \dots$.
2. For each pair of lines which connect the two same vertices in the same direction, one needs to add a factor $1/2$.
3. For each pair of equivalent \hat{T}_m vertices, one needs to add a factor $1/2$. Equivalent \hat{T}_m vertices are vertices with the same number of line pairs that are connected to the interaction vertex in the same way.
4. Any two-particle interaction vertex line is associated with an antisymmetric two-electron integral on the form $\langle \text{left-out, right-out} | | \text{left-in, right-in} \rangle$
5. One should sum over all distinct permutations \hat{P} , i.e, permutations that give unique terms.
6. Disconnected terms (terms with operators that are neither directly nor indirectly connected) do always cancel with other terms, and can be ignored.

For the first diagram, we have $h = 4$ and $l = 2$ due to two external lines, such that the term is positive. We have a pair of lines connecting the right-hand-side \hat{T}_2 -vertex to the interaction vertex, and a pair connecting the left-hand-side \hat{T}_2 -vertex to the interaction line. Since we do not have any equivalent \hat{T}_m vertices nor distinct permutations, we end up with a prefactor $1/4$.

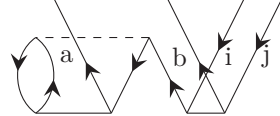
$$\text{Diagram 1} = \frac{1}{4} \sum_{klcd} t_{kl}^{ab} t_{ij}^{cd} \langle kl | \hat{v} | cd \rangle$$

The second diagram has $h = 4$ and $l = 4$, such that the sign again is positive. This diagram has two equivalent \hat{T}_m vertices (due to the symmetry), such that we get a factor $1/2$. We need to permute $\hat{P}(ab)$ and $\hat{P}(ij)$, but it turns out that they give the same terms such that we can choose which one to use and remove the factor in front. The result is



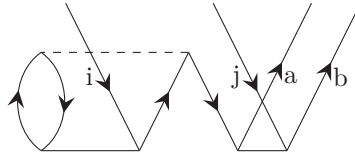
$$= \hat{P}(ab) \sum_{klcd} t_{ik}^{ac} t_{lj}^{db} \langle kl | \hat{v} | cd \rangle$$

Going further to the third term, we see that this term has one internal loop and two external, giving a negative sign. We have a pair of lines connecting the right-hand-side \hat{T}_2 -vertex to the interaction vertex in the same direction, giving a factor $1/2$. In addition we need to permute a, b since they point from two different vertices, and we obtain



$$= -\frac{1}{2} \hat{P}(ab) \sum_{klcd} t_{kl}^{ca} t_{ij}^{cb} \langle kl | \hat{v} | cd \rangle$$

The understanding of the fourth diagram is similar to the third one, but now all the particle lines are replaced by hole lines and vice versa. We have one internal line and two external lines, giving a negative sign. The factor is again $1/2$, due to two lines going between the left-hand-side \hat{T}_2 -vertex and the interaction vertex in the same direction. Now i and j are connected to different vertices, such that we need to permute them.



$$= -\frac{1}{2} \hat{P}(ij) \sum_{klcd} t_{ki}^{cd} t_{lj}^{ab} \langle kl | \hat{v} | cd \rangle$$

The final diagram is affected by the final rule, and cancel since it is both disconnected and unlinked. In fact, it cancels out the energy in the amplitude equations.

APPENDIX B

Jordan-Wigner transformation of Pairing Hamiltonian

Here we show how we write our pairing Hamiltonian (eq. 2.25) in terms of the Pauli matrices. First, we observe that

$$\begin{aligned} a_i^\dagger a_i &= \left(\prod_{k=1}^{i-1} \sigma_z^k \right) \sigma_+^i \left(\prod_{k=1}^{i-1} \sigma_z^k \right) \sigma_-^i \\ &= \sigma_+^i \sigma_-^i = \frac{1}{2} (I^i + \sigma_z^i), \end{aligned}$$

and see that the one-body part of the Hamiltonian can be written as

$$\hat{H}_0 = \frac{1}{2} \xi \sum_p ([(p-1) - (p-1) \% 2] / 2) (I^p + \sigma_z^p).$$

where p now runs over each qubit and $\%$ is the modulo operator. For the interaction term, we get

$$\begin{aligned} \hat{V} = & -\frac{1}{2} g \sum_{pq} \left(\prod_{k=1}^{2p-2} \sigma_z^k \right) \sigma_+^{2p-1} \left(\prod_{k=1}^{2p-1} \sigma_z^k \right) \sigma_+^{2p} \left(\prod_{k=1}^{2q-1} \sigma_z^k \right) \sigma_-^{2q} \left(\prod_{k=1}^{2q-2} \sigma_z^k \right) \sigma_-^{2q-1}. \end{aligned} \quad (\text{B.1})$$

We can see that

$$\begin{aligned} & \left(\prod_{k=1}^{2p-2} \sigma_z^k \right) \sigma_+^{2p-1} \left(\prod_{k=1}^{2p-1} \sigma_z^k \right) \sigma_+^{2p} = \\ & \sigma_z^{\otimes 2p-2} \otimes \sigma_+ \otimes I^{\otimes n-2p-1} \times \sigma_z^{\otimes 2p-1} \otimes \sigma_+ \otimes I^{\otimes n-2p} = \\ & I^{\otimes 2p-2} \otimes \sigma_+ \sigma_z \otimes \sigma_+ \otimes I^{\otimes n-2p}, \end{aligned}$$

so we can rewrite eq. B.1 as

$$\begin{aligned} \hat{V} = & -\frac{1}{2} g \sum_{pq} \left[I^{\otimes 2p-2} \otimes \sigma_+ \otimes \sigma_+ \otimes I^{\otimes n-2p} \right] \left[I^{\otimes 2q-2} \otimes \sigma_- \otimes \sigma_- \otimes I^{\otimes n-2q} \right]. \end{aligned}$$

We have two possibilities here. First for $p = q$:

$$-\frac{1}{2} g \left[I^{\otimes 2p-2} \otimes \sigma_+ \sigma_- \otimes \sigma_+ \sigma_- \otimes I^{\otimes n-2p} \right], \quad (\text{B.2})$$

and for $q - p \geq 1$ we have

$$-\frac{1}{2} g \left[I^{\otimes 2p-2} \otimes \sigma_+ \otimes \sigma_+ \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_- \otimes \sigma_- \otimes I^{\otimes n-2q} \right]. \quad (\text{B.3})$$

B. Jordan-Wigner transformation of Pairing Hamiltonian

For $p - q \geq 1$ we simply exchange p with q and σ_+ with σ_- . To continue, we insert the expressions for σ_{\pm} (eq. 4.21) into these equations. For $p = q$ (eq. B.2) we then have:

$$\begin{aligned} V_{p=q} &= -\frac{1}{8}g \left[I^{\otimes 2p-2} \otimes (I + \sigma_z) \otimes (I + \sigma_z) \otimes I^{\otimes n-2p} \right] \\ &= -\frac{1}{8}g \left[I^{\otimes 2p-2} \otimes I \otimes I \otimes I^{\otimes n-2p} \right. \\ &\quad + I^{\otimes 2p-2} \otimes I \otimes \sigma_z \otimes I^{\otimes n-2p} \\ &\quad + I^{\otimes 2p-2} \otimes \sigma_z \otimes I \otimes I^{\otimes n-2p} \\ &\quad \left. + I^{\otimes 2p-2} \otimes \sigma_z \otimes \sigma_z \otimes I^{\otimes n-2p} \right], \end{aligned}$$

and for $q - p \geq 1$ (eq. B.3) we get

$$-\frac{1}{32}g \left[I^{\otimes 2p-2} \otimes (\sigma_x + i\sigma_y) \otimes (\sigma_x + i\sigma_y) \otimes I^{\otimes 2(q-p-1)} \otimes (\sigma_x - i\sigma_y) \otimes (\sigma_x - i\sigma_y) \otimes I^{\otimes n-2q} \right].$$

This can be rewritten as sixteen four-qubit operations

$$\begin{aligned} &-\frac{1}{32}g \left[I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2qp} \right. \\ &\quad \mp i I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad \mp i I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad - I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad \pm i I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad + I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad + I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad \mp i I^{\otimes 2pq-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad \pm i I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad + I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad + I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad \mp i I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad - I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad \pm i I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2qp} \\ &\quad \pm i I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2qp} \\ &\quad \left. + I^{\otimes 2pq-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(qp-pq-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2qp} \right], \end{aligned}$$

where the subscript is used if $p > q$. We can easily see that when running over the sum over p and q , the terms with \pm and \mp sign in front will cancel out. Also, we can include a factor 2 and limit the sum to $p < q$. Therefore:

$$\begin{aligned}
V_{p \neq q} = & -\frac{1}{16}g[I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2q} \\
& - I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2q} \\
& + I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2q} \\
& + I^{\otimes 2p-2} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2q} \\
& + I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_y \otimes I^{\otimes n-2q} \\
& + I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_x \otimes I^{\otimes n-2q} \\
& - I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_x \otimes \sigma_x \otimes I^{\otimes n-2q} \\
& + I^{\otimes 2p-2} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes 2(q-p-1)} \otimes \sigma_y \otimes \sigma_y \otimes I^{\otimes n-2q}].
\end{aligned}$$

Bibliography

- [1] Ackland, G. *Quantum mechanics lecture notes from University of Edinburgh*.
- [2] Anuj Dawar, t. U. o. C. *Quantum computing lecture notes*.
- [3] Arute, F. et al. ‘Quantum supremacy using a programmable superconducting processor’. In: *Nature* vol. 574, no. 7779 (2019), pp. 505–510.
- [4] Benedetti, M., Lloyd, E., Sack, S. and Fiorentini, M. ‘Parameterized quantum circuits as machine learning models’. In: *Quantum Science and Technology* vol. 4, no. 4 (Nov. 2019), p. 043001.
- [5] Blalock, D., Ortiz, J. J. G., Frankle, J. and Gutttag, J. *What is the State of Neural Network Pruning?* 2020. arXiv: 2003.03033 [cs.LG].
- [6] Bravyi, S. B. and Kitaev, A. Y. ‘Fermionic Quantum Computation’. In: *Annals of Physics* vol. 298, no. 1 (May 2002), pp. 210–226.
- [7] Brun, T. A. *Quantum Error Correction*. 2019. arXiv: 1910.03672 [quant-ph].
- [8] Butcher, S. ‘This is why JPMorgan is hiring in quantum computing’. In: *eFinancialCareers* (2020).
- [9] Childs, A. M., Maslov, D., Nam, Y., Ross, N. J. and Su, Y. ‘Toward the first quantum simulation with quantum speedup’. In: *Proceedings of the National Academy of Sciences of the United States of America* vol. 115, no. 38 (Sept. 2018), pp. 9456–9461.
- [10] Condon, E. U. ‘The Theory of Complex Spectra’. In: *Phys. Rev.* vol. 36 (7 Oct. 1930), pp. 1121–1133.
- [11] Cooper, L. N. ‘Bound Electron Pairs in a Degenerate Fermi Gas’. In: *Phys. Rev.* vol. 104 (4 Nov. 1956), pp. 1189–1190.
- [12] Crawford, T. D. and Schaefer III, H. F. ‘An Introduction to Coupled Cluster Theory for Computational Chemists’. In: (2007), pp. 33–136. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470125915.ch2>.
- [13] Croot, E. *The Rayleigh Principle for Finding Eigenvalues*. 2005.
- [14] Das Sarma, S., Deng, D.-L. and Duan, L.-M. ‘Machine learning meets quantum physics’. In: *Physics Today* vol. 72, no. 3 (2019), pp. 48–54. eprint: <https://doi.org/10.1063/PT.3.4164>.
- [15] Deutch, D. ‘Quantum theory, the Church-Turing principle and the universal quantum computer’. In: (1985).
- [16] Farhi, E., Goldstone, J., Gutmann, S. and Sipser, M. *Quantum Computation by Adiabatic Evolution*. 2000. arXiv: quant-ph/0001106 [quant-ph].
- [17] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [18] Gron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O’Reilly Media, Inc., 2017.
- [19] Gyongyosi, L. and Imre, S. ‘Circuit Depth Reduction for Gate-Model Quantum Computers’. In: *Scientific Reports* vol. 10, no. 1 (2020), p. 11229.
- [20] Henry C. King, t. U. o. M. *Complex inner products*.

- [21] Hewamalage, H., Bergmeir, C. and Bandara, K. *Recurrent Neural Networks for Time Series Forecasting: Current Status and Future Directions*. 2019. arXiv: **1909.00590** [cs.LG].
- [22] Hjorth-Jensen, M. ‘Many-body Hamiltonians, basic linear algebra and Second Quantization’. In: *Lecture Notes, University of Oslo* ().
- [23] Hjorth-Jensen, M., Lombardo, M. P. and Kolck, U. van, eds. *An Advanced Course in Computational Nuclear Physics*. Vol. 936. Springer, 2017.
- [24] Ho, A. and Mohseni, M. ‘Announcing TensorFlow Quantum: An Open Source Library for Quantum Machine Learning’. In: *Google AI Blog* ().
- [25] Howard Haber University of California, S. C. *The time evolution operator as a time-ordered exponential*. 2018.
- [26] Johnson, T. H. ‘What is a quantum simulator?’ In: (2014).
- [27] Labach, A., Salehinejad, H. and Valaee, S. *Survey of Dropout Methods for Deep Neural Networks*. 2019. arXiv: **1904.13310** [cs.NE].
- [28] Lin, H. W., Tegmark, M. and Rolnick, D. ‘Why Does Deep and Cheap Learning Work So Well?’ In: *Journal of Statistical Physics* vol. 168, no. 6 (2017), pp. 1223–1247.
- [29] Lloyd, S. ‘Universal Quantum Simulators’. In: (1996).
- [30] Low, G. H. and Chuang, I. L. ‘Optimal Hamiltonian Simulation by Quantum Signal Processing’. In: *Phys. Rev. Lett.* vol. 118 (1 Jan. 2017), p. 010501.
- [31] Meister, S. *Block Sphere figure*.
- [32] Moll, N. et al. ‘Quantum optimization using variational algorithms on near-term quantum devices’. In: *Quantum Science and Technology* vol. 3, no. 3 (June 2018), p. 030503.
- [33] Möttönen, M., Vartiainen, J., Bergholm, V. and Salomaa, M. ‘Transformation of quantum states using uniformly controlled rotations’. In: *Quantum Information Computation* vol. 5 (Sept. 2005), pp. 467–473.
- [34] Nam, Y. et al. ‘Ground-state energy estimation of the water molecule on a trapped-ion quantum computer’. In: *npj Quantum Information* vol. 6, no. 1 (2020), p. 33.
- [35] Nielsen, M. A. ‘The Fermionic canonical commutation relations and the Jordan-Wigner transform’. In: 2005.
- [36] Nielsen, M. A. and Chuang, I. L. ‘Quantum Computation and Quantum Information’. In: (Mai 2003).
- [37] Nielsen, M. A. and Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011.
- [38] Ovrum, E. ‘Quantum Computing and Many-Body Physics’. In: 2003.
- [39] Pednault, E., Gunnels, J. A., Nannicini, G., Horesh, L. and Wisnieff, R. *Leveraging Secondary Storage to Simulate Deep 54-qubit Sycamore Circuits*. 2019. arXiv: **1910.09534** [quant-ph].
- [40] Rebentrost, P., Mohseni, M. and Lloyd, S. ‘Quantum Support Vector Machine for Big Data Classification’. In: *Physical Review Letters* vol. 113, no. 13 (Sept. 2014).
- [41] Repository, U. M. L. *Iris Data Set*.
- [42] Research, I. *Qiskit home page*.
- [43] Rojas, R. *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer-Verlag, 1996.
- [44] Romero, J., Babbush, R., McClean, J. R., Hempel, C., Love, P. and Aspuru-Guzik, A. *Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz*. 2017. arXiv: **1701.02691** [quant-ph].
- [45] Romero, J., Babbush, R., McClean, J. R., Hempel, C., Love, P. and Aspuru-Guzik, A. *Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz*. 2017. arXiv: **1701.02691** [quant-ph].
- [46] Rossmann, W. *Lie Groups: An Introduction Through Linear Groups*. OUP Catalogue 9780199202515. Oxford University Press, 2006.

-
- [47] Saki, A. A., Alam, M. and Ghosh, S. *Study of Decoherence in Quantum Computers: A Circuit-Design Perspective*. 2019. arXiv: **1904.04323** [cs.ET].
 - [48] Scipy. *Scipy optimization documentation*.
 - [49] Shavitt, I. and Bartlett, R. J. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009.
 - [50] Szabo, A. and Ostlund, N. S. *Modern Quantum Chemistry*. Dover Publications, INC, 1996.
 - [51] T. von Petersdorff, t. U. o. M. *Numerical Integration*.
 - [52] Tacchino, F., Macchiavello, C., Gerace, D. and Bajoni, D. ‘An artificial neuron implemented on an actual quantum processor’. In: *npj Quantum Information* vol. 5, no. 1 (2019), p. 26.
 - [53] Wick, G. C. ‘The Evaluation of the Collision Matrix’. In: *Phys. Rev.* vol. 80 (2 Oct. 1950), pp. 268–272.