Programmatic Representation of Quantum Many Body Systems

by

Xiu-Zhe Luo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Physics

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Yi-Zhuang You Assistant Professor, Dept. of Physics, University of California San Diego
Supervisor:	Roger G. Melko Professor, Dept. of Physics and Astronomy, University of Waterloo Associate Faculty, Perimeter Institute for Theoretical Physics
Internal Member:	Anton Burkov Professor, Dept. of Physics and Astronomy, University of Waterloo
Internal Member:	Crystal Senko Associate Professor, Institute for Quantum Computing and Dept. of Physics and Astronomy, University of Waterloo

Internal-External Member: Pierre-Nicholas Roy
Professor, Dept. of Chemistry,
University of Waterloo

Other Member(s): Timothy H. Hsieh
Adjunct Faculty, Dept. of Physics and Astronomy,
University of Waterloo
Senior Faculty, Perimeter Institute for Theoretical Physics

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Publications

The paper contains material from the following publications (organized chronologically):

Yao. jl: Extensible, efficient framework for quantum algorithm design
Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, Lei Wang
Quantum, 2020

Quantum optimization of maximum independent set using Rydberg atom arrays
Sepehr Ebadi, Alexander Keesling, Madelyn Cain, Tout T Wang, Harry Levine, Dolev Bluvstein, Giulia Semeghini, Ahmed Omran, J-G Liu, Rhine Samajdar, *Xiu-Zhe Luo*, Beatrice Nash, Xun Gao, Boaz Barak, Edward Farhi, Subir Sachdev, Nathan Gemelke, Leo Zhou, Soonwon Choi, Hannes Pichler, S-T Wang, Markus Greiner, V Vuletić, Mikhail D Lukin.
Science, 2022

Operator Learning Renormalization Group
Xiu-Zhe Luo, Di Luo, Roger Melko
arXiv:2403.03199

Open-Source Software

A substantial portion of the work in this thesis was implemented in the following open-source software packages (organized chronologically by creation):

Yao
Extensible, Efficient Quantum Algorithm Design for Humans. (2019)
https://github.com/QuantumBFS/Yao.jl

ZXCalculus
An implementation of ZX-calculus in Julia (2020)
https://github.com/QuantumBFS/ZXCalculus.jl

YaoCompiler
an SSA-based compiler for quantum circuits (2020)
https://github.com/QuantumBFS/YaoCompiler.jl

Quon
Topological Evaluation of Quantum Information (2020)
https://github.com/QuantumBFS/Quon.jl

OpenQASM
Parsers and Tools for OpenQASM (2020)
https://github.com/QuantumBFS/OpenQASM.jl

Expronicon
Collective tools for metaprogramming on Julia Expr (2021)
https://github.com/Roger-luo/Expronicon.jl

Bloqade.jl
Julia package for the quantum computation and quantum simulation based on the neutral-atom architecture. (2022)
https://github.com/QuEraComputing/Bloqade.jl

bloqade-python
Python package for the quantum computation and quantum simulation based on the neutral-atom architecture. (2023)
https://github.com/QuEraComputing/bloqade-python

teal
Python package for operator learning renormalization group. (2024)
https://github.com/Roger-luo/teal

Breakdown of Contributions

My personal contributions to the above publications and software packages are as follows:

For the paper *Yao.jl: Extensible, efficient framework for quantum algorithm design*, I am the main author of the paper, and one of the main developers of the Yao.jl package. Jinguo Liu also contributed to the development in various features as a main developer. Pan Zhang and Lei Wang was funding the project, and provided guidance on the development of the package.

For the paper *Quantum optimization of maximum independent set using Rydberg atom arrays*, This is a collaborative work with many authors. I contributed to the exact simulation of the experiment and hyperparameter tuning of the optimizer and partially helped build the software that connects the optimizer and experiment. This work led to the software development kit `Bloqade` for neutral atom arrays later. I am the lead developer of the `Bloqade` package.

For the paper *Operator Learning Renormalization Group*, I am the main author of the paper and the main developer of the `teal` package. I independently developed the theory, algorithm, and software package. Di Luo contributed helpful discussions and was thus included as a co-author. Roger Melko provided guidance and various support on the research. All the authors contributed to the writings of the paper.

For the software package `Expronicon, Bloqade, teal`, I am the main developer of the package.

For the software package `ZXCalculus`, I mentor the main developer, Chen Zhao, and provided guidance and ideas on the development of the package.

For the software package `OpenQASM`, I implement the main parser implementation, with the help from Taine Zhao on the parser generator.

For the software package `YaoCompiler`, I am the main developer of the package, Valintin Churavy and William Mose contributed to the maintainance and discussions of the package.

For the software package `Quon`, I organized the development of the package, and contributed the data structure design. Chen Zhao implemented the main idea, and Xun Gao provided theoretical guidance.

Abstract

The problem of simulating quantum many-body systems is fundamental in condensed matter physics, quantum computing, and quantum chemistry. The exact simulation of quantum many-body systems is generally intractable on classical computers, and developing efficient simulation methods is crucial for understanding and utilizing quantum systems. Meanwhile, from the computer science community, the development of formal languages has dramatically improved programming and software efficiency. Thus, it is natural to ask whether we can develop and utilize such representations to simulate quantum many-body systems. We propose so-called *programmatic representations* for simulating quantum many-body systems on computational devices.

We begin with introducing the programmatic representations for quantum circuits, quantum operators, quantum states, pulse sequences, and more general quantum programs with control flows are discussed. We further introduce the transformation of and between these representations, which leads to the development of several software frameworks, including `Yao` and `Bloqade`, which achieved state-of-the-art performance in simulating quantum circuits and Rydberg atom array dynamics. We introduce the transformation for automatic differentiation and show that by utilizing the reversibility of the quantum circuits, only constant memory overhead is needed for the automatic differentiation of quantum circuits in simulators. As a result, we report the differentiation of 10,000-layer quantum circuits that no previous software can achieve.

On top of these technical developments in exact simulation, hardware modeling, and automatic differentiation, we generalize the numerical renormalization group formulations from Wilson and White, namely Wilson's NRG and White's DMRG, which we call the *operator learning renormalization group (OLRG)*. OLRG allows solving general quantum many-body problems with arbitrary operator maps in lieu of a state ansatz. We introduce a theory framework guiding the design of OLRG loss functions, providing a rigorous error bound for real-time evolution. We further show OLRG can solve the quantum many-body problems with arbitrary operator maps such as neural networks using the Operator Matrix Map (OMM), and can be used to generate control parameters for a quantum device using the Hamiltonian Expression Map (HEM). We explore different hyperparameters for both OMM and HEM for a 1D transverse field Ising model and show that our theoretical loss function correctly guides both the OMM and HEM to ground truth using differentiable programming.

We conclude by discussing the future directions of applying programmatic representations to quantum many-body systems and the future directions of quantum many-body system simulation.

Acknowledgements

I would like to thank all the people who made this thesis possible. My PhD journey has been a long and winding road due to the unpredictable global environment. Therefore, I would like to thank my supervisor Roger G. Melko and previous supervisors, Lei Wang and Pan Zhang, who took care of me when I was almost homeless. My achievements are impossible without their effort in maintaining a free and stressless research environment. My work has been unusual in both the physics and computer science communities, and I am grateful for their support and guidance in developing the software and the theory. I thank my Ph.D. committee members Anton Burkov, Crystal Senko, and Tim Hsieh for their helpful discussion and advice. I also want to thank my potential supervisor, Steven Flammia, for trying to help me with the visa for various positions I applied for.

My Ph.D. has also gone through the COVID-19 pandemic, and I would like to thank Sebastian Wetzel from Perimeter Institute Quantum Intelligence Lab (PIQuIL) for his friendship and accompaniment during the pandemic. Special thanks are due to my supervisor, Roger Melko, for the best New Year party in his backyard during the pandemic. I would also like to thank other PIQuIL friends, including Anna Golubeva, Bohdan Kulchytskyy, Giacomo Torlai, Ejaaz Merali, Roeland Wiersema, Alev Orfi, Estelle Inack, Danny Kong, Schuyler Moss, Stefanie Czischek and many others for their friendship and support during my Ph.D.

The software development and the theory in this thesis result from the collaboration of many people and a few failed attempts. I want to thank my collaborators, who supported me along this journey with enlightening discussions and hard work: Jinguo Liu, Di Luo, Chen Zhao, Xun Gao, Valentin Churavy, William Moses, Taine Zhao, Madelyn Cain, Shengtao Wang, Alexander Keesling, Dolev Bluvstein, Jing Chen, Phillip Weinberg.

I also would like to thank my friends and colleagues who contributed helpful discussions in the development of the software and the theory: Tim Besard, Mike Innes, Harrison Grodin, Juan Gomez, Christopher J. Wood, Damian Steiger, Damian Steiger, Craig Gidney, corryvrequan, Johannes Jakob Meyer, Nathan Killoran, Divyanshu Gupta, Wei-Shi Wang, Yi-Hong Zhang, Tong Liu, Yu-Kun Zhang, Si-Rui Lu, Hao Xie, Arthur Peash, Miles Stoudenmire, Matthew Fisherman, Hsin-Yuan Huang, Fangli Liu, Juan Carrasquilla, Qi Yang, Hai-Jun Liao, Hao Xie, Jonathan Wurtz, Hong-Ye Hu.

Last, thanks to the people behind the open-source community who made developing the software and the theory possible. Specifically, I would like to thank Viral Shah, Chris Rackauckas, Jeff Bezanson, Stefan Karpinski, Keno Fischer, and many others who spent their time and effort on issues related to my work.

Dedication

This is dedicated to my parents, Dehui Luo and Zixia Qing, and my partner, Fan Zhang, for their endless support and love.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AD Automatic Differentiation 3, 19, 20, 71–74, 77–81, 84, 91, 92

AST Abstract Syntax Tree 17

BNF Backus-Naur Form xiii, xiv, 15–17, 23, 24, 33, 34, 36

CPU Central Processing Unit 24, 25, 88–92, 147, 164

DMET Density Matrix Embedding Theory 107, 115

DMRG Density Matrix Renormalization Group xi, xv, 1, 3, 10, 96–101, 103, 104, 106, 114, 129, 132, 134, 158, 159, 161–164, 167, 168

GPU Graph Processing Unit xiii, xv, 2, 19, 20, 24, 29, 71, 88, 90, 145, 164

HEM Hamiltonian Expression Map xvi, xvii, 97, 99, 132, 138, 140, 141, 143, 147–154, 157, 159, 162, 167–169

IR Intermediate Representation xiv, 2, 39–47, 49, 78, 166, 167

MPS Matrix Product State xii, xv, 98, 102, 103, 131–133, 135, 151, 159, 162–164

NISQ Noisy Intermediate-Scale Quantum 18, 136

NRG Numerical Renormalization Group xv, 1, 3, 10, 96–101, 103, 106, 129, 132, 134, 158, 159, 163, 167

ODE Ordinary Differential Equation 93, 135, 139, 141, 143, 144, 157, 158

OLRG Operator Learning Renormalization Group xv, 10, 97–99, 101, 102, 106–108, 111, 113, 114, 117, 130, 134, 135, 140, 141, 144, 146, 158, 159, 163–165, 167–169

OMM Operator Matrix Map xv–xvii, 97–99, 132, 134, 141, 144–147, 151–156, 158, 159, 162, 168, 169

QASM Quantum Assembly Language 44

QBIR quantum block intermediate representation xiii, 18, 20, 21, 23, 24, 30, 32, 37, 50, 67, 71, 80, 84, 85, 87, 91

SPMD Single Program Multiple Data 29, 91, 141

SSA Static Single Assignment 43–47, 49, 78, 166, 167

TDVP Time-Dependent Variational Principle xii, 131, 132, 159, 163, 164, 168

TFIM Transverse Field Ising Model xv, xvii, 99–101, 104, 112–116, 118–120, 130, 133, 139, 144, 156–161, 168

TNO Tensor Network Operator 105

TNS Tensor Network State 132

TOBC Time-ordered Boundary Correlator xv, xvii, 111–113, 125, 127, 130–132, 135, 136, 139–143, 145, 153, 154, 156, 158, 160, 161, 164

VMC Variational Monte Carlo 1, 96, 131, 133, 168, 169

VQA Variational Quantum Algorithms 1, 96, 136, 140, 159

Chapter 1

Introduction

Quantum many-body systems are one of the most common and important systems in physics. They are crucial in studying the behavior of a wide range of physical systems, including quantum materials and quantum chemistry [15–17]. While the behavior of small-scale quantum systems has been well understood, the behavior of large-scale quantum systems is still not well understood. This leads to the emergence of condensed matter physics. Quoting from PW Anderson's "More is Different" [18], the behavior of large-scale quantum systems is not simply the sum of the behavior of small-scale quantum systems. New physics may emerge when the number of particles increases. Computational simulations and predictions are crucial in understanding the behavior of large-scale quantum systems. However, the simulation of quantum many-body systems is a challenging task. The general problem has been proven to be hard [19, 20]. This complexity has motivated the development of various classical frameworks to tackle this problem, including quantum Monte Carlo [21–23], linked cluster expansion [24, 25] and variational frameworks such as Wilson's NRG [26], White's DMRG [27, 28] and Variational Monte Carlo (VMC) [29, 30]. While these frameworks have been successful in understanding the behavior of many practical quantum systems, they are not without limitations. Quantum computation has been proposed as a promising hardware solution to simulate and understand the behavior of large-scale quantum systems [31]. Quantum frameworks such as quantum phase estimation [32, 33], Hamiltonian simulation [34–36] and Variational Quantum Algorithms (VQA) [37–39] has also been proposed with the promise of potential quantum advantage. In combination with conventional computational methods, developing new methods and software tools can help understand the behavior of large-scale quantum systems.

Concurrently, the development of hardware has seen significant progress in the past

decade, including progressing fidelity in manipulating quantum many-body systems, including superconducting circuits, Rydberg atoms, and Ion traps [40–42]. This urges the development of new methods and software tools to utilize, control, simulate, and characterize these systems. Developing such new methods and software tools has brought new concepts and techniques to the field of physics. New ways to represent quantum many-body systems have been proposed, including mathematical representations such as ZX calculus [43], Quon [44] and IR for programming quantum hardwares [45, 46]. The connection between computational process and many-body systems has been seen in the development of quantum algorithms and quantum simulations in recent decades [19]. Complex behavior arises from the composition of primitive components that share the same nature as the composition of primitive components in the computational process.

Furthermore, the development of formal languages [47] and especially programming languages [48–50] have brought concepts in understanding complex compositions. A holy grail of designing programming languages is to make the composition of primitive components as simple as possible while still being expressive and versatile across different machines. Solving this problem has led to the development of type systems [51], compiler optimizations [52], etc. On top of programming languages, many representations have been developed to build the digital twin of the physical world [53, 54]. Although the initial motivation of formal languages was to describe the syntax and semantics of natural languages, the development of such languages has brought powerful mathematical tools to represent and understand compositions. From an implementation perspective, the development of computational condensed matter physics can benefit from such tools. Moreover, these concepts and techniques may help physicists revisit existing frameworks and enhance the simulations and predictions.

The development of large-scale scientific facility [55–57] in high-energy physics has led to many breakthrough discoveries since last century. Similarly, starting a decade ago, scientific software development is crucial in many condensed matter physics discoveries. These software and algorithms are the large-scale "accelerators" in this century, such as `Gaussian` [58], `OpenFermion` [59], `ALPS` [60], etc. Another classic example is the deep learning software stacks including software frameworks like `TensorFlow` [7], `PyTorch` [9] and hardware programming facility like `CUDA` [61]. The entire field of deep learning is made possible with these software efforts. Thus, scientific software development has seen significant growth in the past decade. More sophisticated software frameworks have been developed for multiple applications. This imposes challenges in software engineering, such as performance, maintainability, correctness, etc. Moreover, in the twilight of Moore's law, we must seriously consider developing scientific software to fully utilize current and emerging hardware, such as GPU, quantum computers, neuromorphic computers, etc. The

development of representations mentioned previously has brought powerful tools to address these challenges.

Combining the development of formal languages and scientific software development, we can develop new methods and software tools to understand the behavior of large-scale quantum systems. In this thesis, we will refer to these representations as programmatic representations, as their primary purpose is to represent the physical entity in a computational program rather than creating formal definitions. We will introduce programmatic representations explored in the related work in Chapter 2, including representations for quantum circuits, quantum operators, quantum registers, and Rydberg atom arrays. Then, we will introduce the transformation of and between these representations in Chapter 3, including the transformation of quantum circuits for simulation and Automatic Differentiation (AD). Furthermore, we will introduce how we generalize the well-known Willson's NRG and White's DMRG from a programming and machine-learning perspective in Chapter 4. Finally, we conclude this thesis by discussing the future directions of applying programmatic representations in quantum many-body systems in Chapter 5.

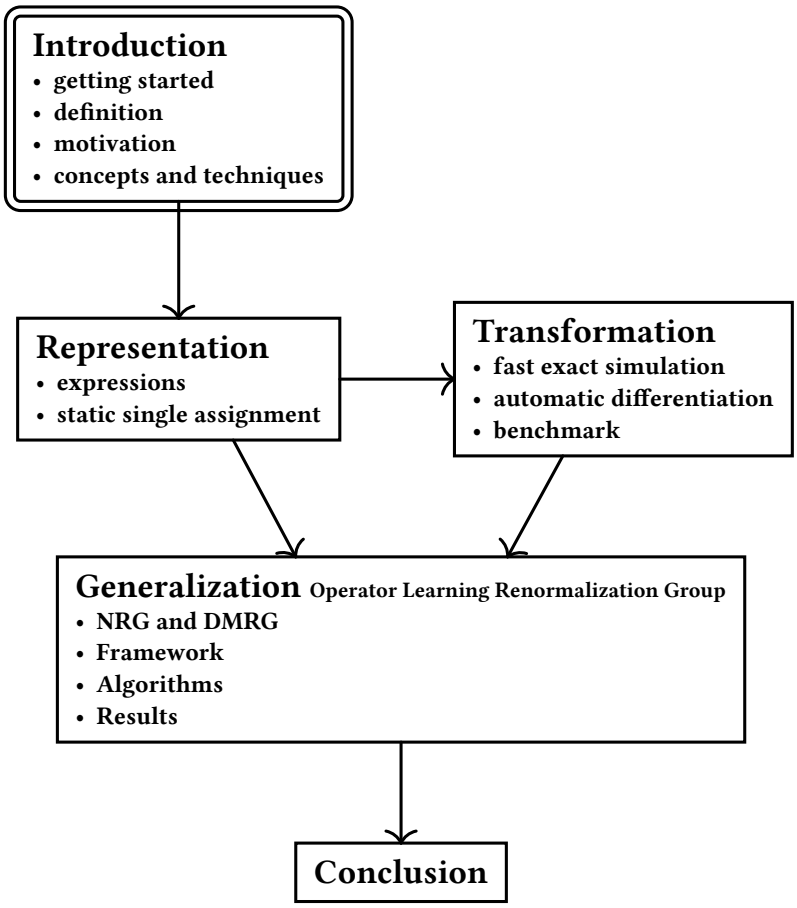

Figure 1.1: Structure of the thesis. Arrows indicate possible orders of reading.

In this chapter, starting with a simple projectile motion example, we will introduce the basic concepts of programmatic representations and discuss their motivation in more detail. We will also introduce the basic concepts and techniques used in this thesis. While much of this thesis's content is referred to as formal language, we will introduce them with more concrete examples for physicists without going deep into formalism unless necessary.

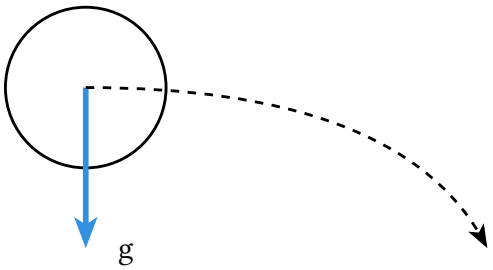

Figure 1.2: Projectile Motion

1.1 Getting Started

In this section, we will introduce the basic concepts of programmatic representations through the very simple example of projectile motion. Let's start with representing the same physical system in different ways.

1.1.1 5 Ways of Describing the Projectile Motion

Let's first start with some standard textbook representations

List of Positions if we are able to observe a projectile motion from the experiment, we can represent this process as a list of positions at different times:

$$x(t_1), x(t_2), \cdots, x(t_n) \tag{1.1}$$

Diagram we can also draw a diagram to represent the projectile motion by plotting the curve of positions:

$$y = \frac{g}{2}x^2 \tag{1.2}$$

Differential Equation using the Newton's law, we can represent the projectile motion as a differential equation:

$$\frac{d^2\vec{r}}{dt^2} = \vec{g} \tag{1.3}$$

Lagrangian using the Lagrangian mechanics, we can represent the projectile motion as a Lagrangian:

$$L = T - V \tag{1.4}$$

What if we ask you to pick one of the above representations and write a computer program to calculate y given x? If x is in our list of positions, then **List of Positions** seems to be the most straightforward representation. The corresponding data structure would be a 1-D array of positions.

However, this is not true if we want to calculate y for a continuous x. In this case, we need to use **Diagram**, **Differential Equation**, or **Lagrangian**. They all can predict y given a random x.

The situation changes when we start thinking about more generic cases. If we are now looking for a variant of projectile motion with acceleration at each $\vec{x}$ and $\vec{y}$ direction, then **Diagram** is not generic enough to describe such system. **Differential Equation** is the best way to describe such a system because the acceleration is a direct parameter in the representation. On the other hand, if we are looking for a variant of path, such that the energy is preserved. The **Lagrangian** becomes a good representation because it has a more explicit representation on energy. However, if we obtain a solution from **Differential Equation**, we still need to convert it to **List of Positions** to visualize the trajectory because **List of Positions** is a more native representation when we draw the curve on a screen.

From here, we can see different representations have their strength. Solving a problem requires one to transform between multiple representations. This is also true when we talk about writing a program.

The most straightforward representation is **List of Positions** because we can represent it directly using `list`, a *primitive* data structure in `python`. **Diagram** is a bit more complicated but still straightforward. It requires one to define the corresponding function. **Differential Equation** will then require one to define a numerical integrator to solve the differential equation. **Lagrangian** is the most complicated one because it requires one to define the Lagrangian function and then use the Euler-Lagrange equation to solve the equation of motion.

1.1.2 What is Programmatic Representation?

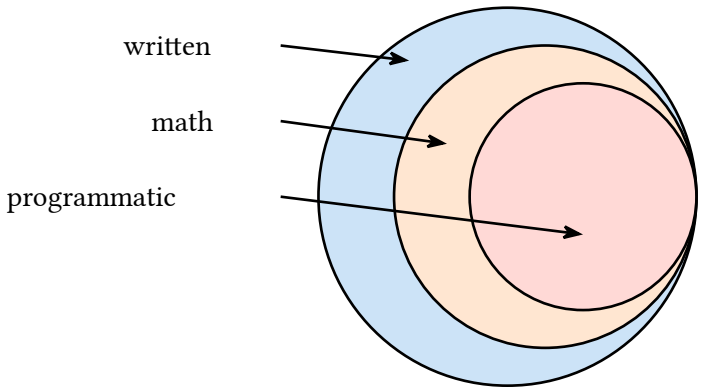

Figure 1.3: Written Representation, Mathematical Representation and Programmatic Representation

When discussing the programmatic representation, we compare it with the written and mathematical representation. The written representation is the most straightforward way to represent a physical system. It is how we describe a physical system in natural symbols, language, and diagrams. Mathematical representation is the way we describe a physical system in mathematical language. The programmatic representation is the way we describe a physical system with the consideration of being able to execute the representation in a computer program. The three representations are shown in Figure 1.3.

An example of written and mathematical representation could be Euclid's Elements, where mathematical and written representations describe geometry. The most common example of programmatic representation is the general-purpose programming languages, such as Python, Julia, C++, etc. Programming languages serve as the most generic way to represent programs that a computer can execute.

As one can imagine, mathematical representations and programmatic representations can be written. Similarly, programmatic representations can always be formally defined by mathematical representations. However, the written or mathematical representations are not necessarily executable by a computer. The programmatic representations are executable by a computer but not necessarily human-readable.

However, creating only general-purpose programming languages is insufficient for representing physical systems. Because the details about the physics are not encoded in the general-purpose programming languages, diagrammatic representations can sometimes be programmatic as well. Thus, specific knowledge and understanding of the representations within the context of quantum many-body physics is necessary.

In summary, the programmatic representation is about the mathematical representations that a computer can execute. It considers the formal definitions, the implementation details, and the domain knowledge. This thesis discusses the more specific programmatic representations that physicists can use in experiments, simulations, and theory.

1.2 Motivation

There are several motivations for using programmatic representations in physics. The most obvious motivation is bridging different fields by "speaking" the same language. Using programmatic representations also leads to powerful software tools and an alternative understanding of theories in physics from an entirely different perspective. In this section, we will discuss some of the motivations that result in the development of work presented in this thesis.

1.2.1 Bridging the Gap between Theoretical, Computational and Experimental Physics

The most obvious motivation for utilizing a programmatic representation appears when experimental, computational, and theoretical physicists work together. The physicists knowing both experimental and theoretical physics are rare. The fields of physics are becoming more and more specialized. Even within the same field, theoretical, computational, and experimental physicists may use different representations and terminologies. This makes communication between physicists difficult, obstructing the development of new theories and the lack of understanding of the physical systems.

Theoretical physicists often use mathematical representations to describe the physical systems. Computational physicists use numerical representations to simulate the physical systems and provide guidance for the experimental physicists. The experimental physicists then use the physical devices to perform experiments and collect data. such workflow works okay if the experiment is only executed a few times. But if many experiments are needed and their configuration and setup are different, then a unified machine-executable

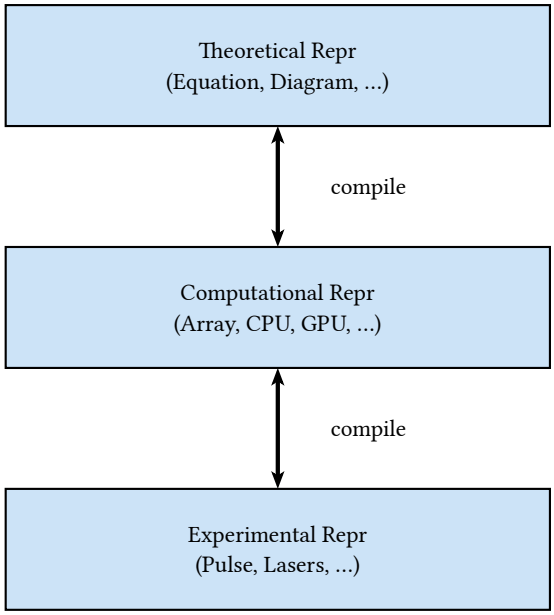

Figure 1.4: Bridging the Gap between Theoretical, Computational and Experimental Physics

representation is needed. This is where the programmatic representation comes in. It can be used to represent the physical system in a way that can be written by theoretical physicists, simulated by computational physicists, and executed by experimental physicists.

Assuming there are three physicists, Alice, Bob, and Charlie. If each of them works with different representations A, B, C, without a unified representation, $\frac{3\cdot(3-1)}{2} = 3$ different conversions are needed, and this scales with the number of physicists as $O(\frac{n(n-1)}{2})$. If there is a unified representation, then only n conversion is needed, and this scales with the number of physicists as $O(n)$. This compilation between hierarchical representations significantly reduces the complexity of communication between physicists.

More specifically, as shown in Figure 1.4, by utilizing programmatic representations, theoretical physicists can write down equations and diagrams and then compile them into machine executable representations such as primitive data structures such as integers, floating points, and arrays. Computational physicists can then use these machine-executable representations to simulate the physical systems and further generate representations that the experimental devices, such as pulse sequences and laser configurations, can execute.

1.2.2 Building Performant, Sophisticated and Multi-purpose Software Framework

From the design patterns in general-purpose software to the design of a domain-specific language, the programmatic representations can help us architect the software framework in a performant, sophisticated, and multi-purpose way while staying composable. Thus, it brings existing methods to a new level of usability and performance. An example is the optimization of the exact methods. Using the circuit representations we introduced in Section 2.1, we can automatically optimize the exact simulation by dispatching the general simulation of a unitary to the simulation of different patterns. The match-and-dispatch technique helps us build one of the fastest exact circuit emulators in Section 3.1. Furthermore, by applying the techniques from the compiler community, we can further automate the optimization process using the rewrite rules. Potentially, we also see future directions in optimizing the program for quantum devices, such as the optimization of quantum circuits for quantum hardware.

1.2.3 A way of thinking

Programmatic representations can unveil alternative methods to solve problems as a way of thinking. By understanding existing methods from a programmatic perspective, we can generalize them more abstractly. This way of thinking can lead to the discovery of alternative methods that solve broader problems. For example, by switching the floating points with tropical numbers, one can solve combinatorial optimization problems using the algorithm used for linear algebra and tensor networks[62]. In Chapter 4, we will show the generalization of NRG[26] and DMRG[27, 28] to the OLRG. This generalization is inspired by revisiting the implementation of the DMRG algorithm and understanding it from a programmatic perspective. By generalizing the DMRG algorithm, we remove the limitation of the numerical RG formulation in higher dimensions and hardware platforms. This led to the discovery of the OLRG algorithm, which utilizes machine learning techniques on conventional and quantum computers.

1.3 Useful Concepts and Techniques

In this section, we will introduce some useful concepts and techniques from computer science that are used in this thesis. We will not discuss the formal definition of these concepts and techniques, but instead, we will introduce them with concrete examples for physicists without going deep into formalism. For readers familiar with these concepts and techniques, they can skip this section.

1.3.1 Expression

Using the expression is the most generic way to represent a programmatic representation. An expression is a combination of symbols that denotes a value. Due to the nature of composition, a tree data structure arises naturally to represent an expression. In parsing, this tree structure is also called a syntax tree. However, we will not go deep into the parsing in this thesis. Instead, we will focus on the tree structure and its operations.

For example, consider the mathematical term $x+y*z$. This can be represented as a tree structure as shown in Figure 1.5 (a). One of the reasons why we are particularly interested in the expression tree is that although it can be very generic, the tree data structure fits into modern computer architecture very well. It can be easily represented using arrays and pointers and traversed using recursion.

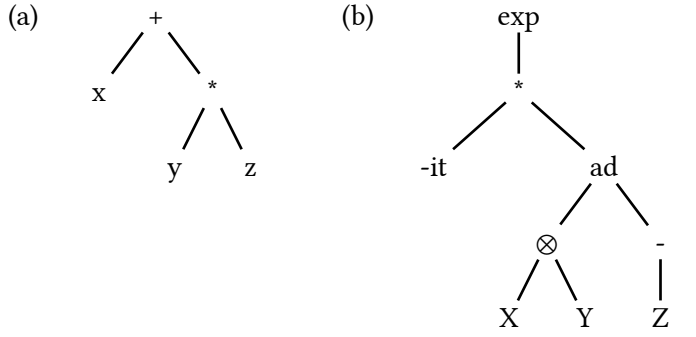

Figure 1.5: Expression Tree of (a) $x + y * z$; (b) $\exp(-it[X \otimes Y, -Z])$

However, in practice, representing the expression tree requires heterogeneous node types because the nodes in an expression tree represent a function that takes several arguments (its children) and returns a value (itself). This results in a non-uniform data structure. As shown in a more complicated example in Figure 1.5 (b), representing an expression of a quantum operator $\exp(-it[X \otimes Y, -Z])$ requires different primitive functions such as `exp`, $\otimes$, `ad` (the commutator), etc. Representing such data structure cannot be done statically using a uniform data structure with the same number of fields. This implementation consideration motivates us to discuss the sum types in the next section.

1.3.2 Sum Types

The sum type is a composite type representing the type formed by combining other types. The sum type is also called tagged union, disjoint union, or variant type. It is one of the common classes of algebraic data types. [1]. In the context of programmatic representation, combined with product types (such as tuples), sum types are beneficial in representing the heterogeneous node types in an expression tree.

A typical sum type represents a choice between a fixed set of alternatives. For example, consider the following definition of nodes in Python:

[1] See also https://en.wikipedia.org/wiki/Algebraic_data_type

Listing 1 A simple sum type in Python

```
from dataclasses import dataclass

@dataclass
class Node:
    pass

@dataclass
class Var(Node):
    name: str

@dataclass
class Const(Node):
    value: float

@dataclass
class Add(Node):
    left: Node
    right: Node

@dataclass
class Mul(Node):
    left: Node
    right: Node

@dataclass
class Neg(Node):
    value: Node
```

In this example, `Const`, `Add`, `Mul`, and `Neg` are possible variants of the sum type `Node`. Each of them is a class that extends the base class `Node`. We can see that the minimum requirement to represent such a data structure is a tag and a list of fields. Because runtime input determines the node type's tag, we have to allocate memory for a maximum number of fields (in this case, 2) for each possible instance of `Node`. Furthermore, the memory layout of the data structure is not guaranteed to be aligned in a cache-friendly way. Thus, in optimizing compilers, the memory layout of the data structure is often rearranged to improve the performance of the program[50, 63, 64]. We will not discuss the details of

memory layout in this thesis, but it is important to note that the memory layout of the data structure is an important factor in the program's performance.

1.3.3 Pattern Matching

Pattern matching is a comprehensive concept. In the context of this thesis, we refer to pattern matching a sub-expression in an expression tree. The sub-expression can be a single node or a subtree. One can describe the pattern as the same expression plus a pattern variable. Pattern matching is a widespread operation in computer programs. Pattern matching aims to provide a concise and readable way to extract information from a data structure. Moreover, pattern matching serves as a fundamental tool in manipulating representations. In this section, we will introduce the pattern matching with some concrete examples in `Python`, `Julia`, and `rust` for readers familiar with different languages.

First, look at the pattern matching in `Python` for a concrete example. Python 3.10 introduced the pattern-matching feature. For instance, we can use the following matching statement to extract the sub-expression of multiplication from an arithmetic expression

Listing 2 A simple example of pattern matching in Python

```
def simplify(expr):
    match expr:
        case Mul(left, Const(1.0)):
            return left
        case _:
            return expr

>>> simplify(Mul(Add(Var("a"), Const(1.0)), Const(1.0)))
Add(left=Var(name="a"), right=Const(value=1.0))
```

In Listing 2, our `simplify` function matches the multiplication expression with a constant 1.0 on the right-hand side. If the pattern matches, the function returns the left-hand side of the multiplication. Otherwise, it returns the original expression. Thus, the symbolic expression can be simplified in this way. In the practice of programming or compiler, pattern matching refers to matching a sub-expression from an expression tree.

In `Julia`, the pattern matching is implemented as a package `MLStyle`. A more maintained version is also implemented for `Liang` (introduced in Section 2.3). [2].

Listing 3 A simple example of pattern matching in Julia. `Mul` is the corresponding sum type in `Julia`.

```
using MLStyle

@active struct Mul
    left
    right
end

@match Mul(left, 1.0) begin
    Mul(left, 1.0) => left
    _ => Mul(left, 1.0)
end
```

With pattern matching, we can now rewrite an existing expression into a new one. Pattern matching is a powerful tool for manipulating representations. In this section, we will introduce the rewrite with some concrete examples in `Python`, `Julia`, and `rust` for readers familiar with different languages. Listing 2 is a simple example of rewrite, where we rewrite the multiplication expression with a constant 1.0 on the right side into the left side of the multiplication.

1.3.4 Backus-Naur Form

In Chapter 2, we will use BNF extensively to introduce several expressions, including circuits, operators, and pulse programs. The Backus-Naur Form is a notation technique for context-free grammars, which is used to describe the syntax of programming languages, command-line interfaces, and communication protocols. It is named after the two computer scientists, John Backus and Peter Naur, who introduced it in the 1960s. The BNF is a standard format for the formal description of the syntax of a language, namely the expressions, statements, and program structures. It is widely used in the field of computer science and software engineering.

[2]The pattern matching package is contained in the repository of `Liang` at the time of writing

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle + \rangle \mid \langle * \rangle \\
\langle + \rangle &::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \\
\langle * \rangle &::= \langle \text{expr} \rangle * \langle \text{expr} \rangle
\end{aligned}
$$

Figure 1.6: The BNF definition of the syntax for simple arithmetic expression with plus and multiplication

The BNF denotes a node in the expression tree using $\langle ... \rangle$, taking Figure 1.5 (a) as an example, the generic expression is denoted as $\langle expr \rangle$, and node `+` representing a plus expression is denoted as $\langle + \rangle$. Similarly, the node `*` representing a multiplication expression is denoted as $\langle * \rangle$. Then, the children are defined using a statement, e.g., $\langle expr \rangle := \langle + \rangle | \langle * \rangle$, meaning the generic expression can be either a plus or multiplication. The children can be further defined recursively, e.g., $\langle + \rangle := \langle expr \rangle + \langle expr \rangle$. The BNF is a powerful tool to describe a language's syntax and a good way to introduce the expression tree of quantum objects. In this thesis, we will focus more on presenting the tree structure using BNF instead of focusing on the concrete syntax of a language.

Chapter 2

Representation

Representations are the most important concept in this thesis. They are the foundation of building algorithms and software. They are also the key to understanding the quantum systems. Good representations can unveil the hidden structure and make the computation more efficient. In this chapter, we will introduce the definition of three different representations for quantum objects: expressions, diagrams, and composite representation. Although the reader may be familiar with the corresponding concepts in physics, we will introduce them while considering their computational implementation, such as the data structure and the formalism. In the next chapter, we will introduce the transformation of and between these representations, which contain more application-oriented content.

We begin with expressions of quantum circuits, operators, and hardware pulse sequences. Expressions are the most common programmatic representation. As we have introduced in Section 1.3.1, they are also referred to as Abstract Syntax Tree (AST), *term*, *expression tree* in different contexts. They can be represented as a tree structure, a natural data structure for computer programs. Expressions are easy to manipulate and transform programmatically. They are also easy to visualize and understand. This section will introduce expressions defined for various quantum objects using BNF introduced in Section 1.3.4.

2.1 Quantum Circuit

Quantum circuits are the most common representation of quantum algorithms. They are a sequence of quantum gates, which are unitary operators and measurements. The quantum

gates are applied to the qubits in the circuit. The most general definition of a quantum circuit is probably just a sequence of unitaries $U_1 U_2 \cdots U_n$. However, in practice, there are more structures within these unitaries. For example, they can be the composition of elementary gates or the unitaries representing an existing quantum algorithm. This motivates us to define a more structured representation of quantum circuits. This section will discuss the QBIR in `Yao` to give readers a concrete example of a quantum circuit representation. Then, we will discuss the process of creating such a representation in a more formal way.

In our previous work `Yao`, we explored the expression tree of a quantum circuit. `Yao` is a software for solving practical problems in quantum computation research. Given the limitations of Noisy Intermediate-Scale Quantum (NISQ) circuits [65], treating quantum devices as co-processors and complementing their abilities with classical computing resources is advantageous. Variational quantum algorithms have emerged as a promising research direction in particular. These algorithms typically involve a quantum circuit with adjustable gate parameters and a classical optimizer. Many of these quantum algorithms, including the variational quantum eigensolver for ground states [37–39], quantum approximate optimization algorithm for combinatorial problems [66], quantum circuit learning for classification and regression [67, 68], and quantum circuit Born machine for generative modeling [69, 70] have had small scale demonstrations in experiments [71–76]. There are still fundamental issues in this field that call for better quantum software alongside hardware advances. For example, variational optimization of random circuits may encounter exponentially vanishing gradients [77] as the qubit number increases. Efficient quantum software is crucial for designing and verifying quantum algorithms in these challenging regimes. Other research demands also call for quantum software that features a small overhead for repeated feedback control, convenient circuit structure manipulations, and efficient gradient calculation besides simply pushing up the number of qubits in experiments.

On the other hand, deep learning and its extension differentiable programming offer great inspiration and techniques for programming quantum computers. Differentiable programming [78] composes differentiable components to a learnable architecture and then learns the whole program by optimizing an objective function. The components are typically, but not limited to, neural networks. The word "differentiable" originates from the usual requirement of a gradient-based optimization scheme, which is crucial for scaling up to high dimensional parameter spaces. Differentiable programming removes laborious human efforts and sometimes produces even better programs than humans can produce themselves [79].

Differentiable programming is a sensible paradigm for variational quantum algorithms, where parameters of quantum circuits are modified within a particular parameter space

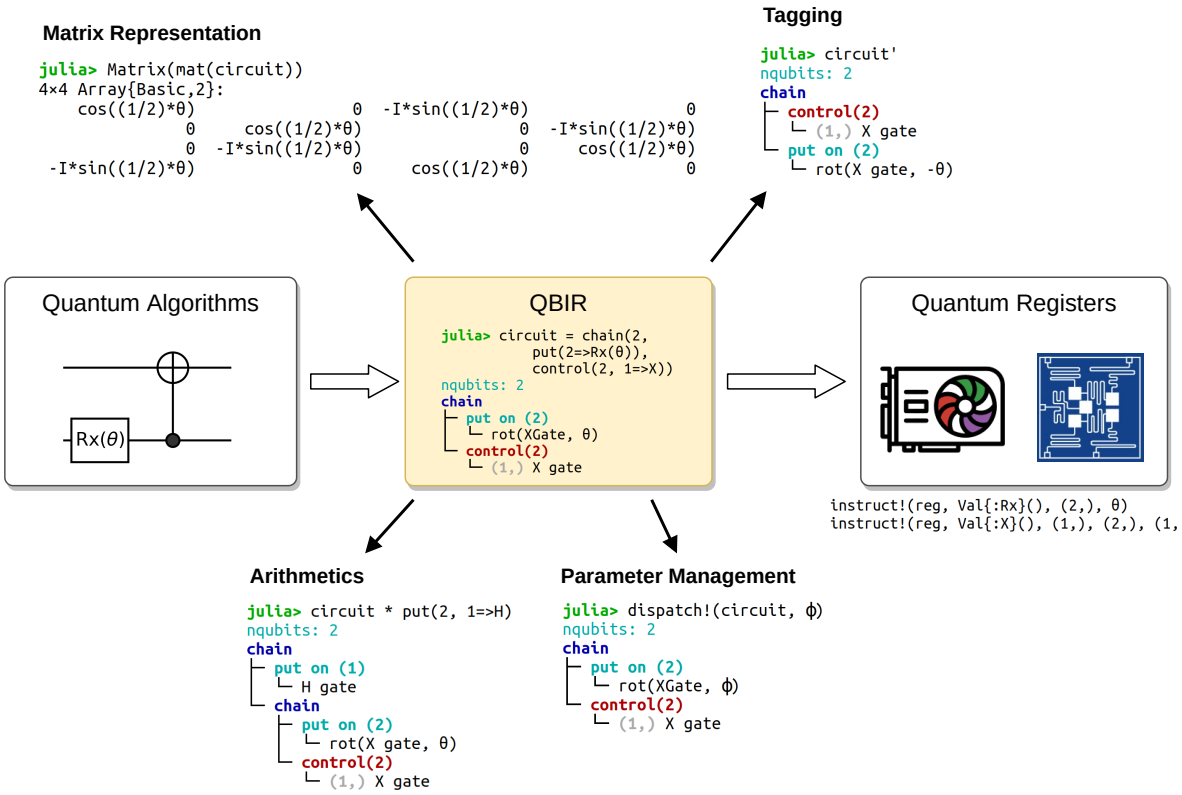

Figure 2.1: Quantum block intermediate representation plays a central role in `Yao`. The images of GPU and quantum circuits are taken from JuliaGPU [1] and IBM q-experience [2].

to optimize a loss function. In this regard, programming quantum circuits in the differentiable paradigms address a much long term issue than the short-term considerations of compensating low-depth noisy quantum circuits with hybrid quantum-classical algorithms. Designing innovative and profitable quantum algorithms is, in general, nontrivial due to the lack of quantum intuitions. Fortunately, differentiable programming offers a new paradigm for devising novel quantum algorithms, much like what has already happened to the classical software landscape [79].

The algorithmic advances in differentiable programming hugely benefit from rapid development in software frameworks [7, 9, 80–83], among which the AD of the computational graph is the key technique behind the scene. A computational graph is a directed acyclic graph that models the computational process from input to output of a program. In order

to evaluate gradients via the automatic differentiation, machine learning packages [7, 9, 80–83] construct computational graphs in various ways.

It is instructive to view quantum circuits from the perspective of computational graphs with additional properties such as reversibility. In this regard, contextual analysis of the quantum computational graphs can be even more profitable than neural networks. For example, uncomputing (adjoint or dagger) a sub-program plays a central role in reversible computing [84] since it returns qubit resources to the pool. While in differentiable programming of quantum circuits, exploiting the reversibility of the computational graph allows differentiating through the quantum circuit with constant memory independent of its depth. Constant memory usage is a significant advantage over the traditional approach of storing intermediate states for backpropagation. The traditional approach requires memory allocation that is linear in the circuit depth. We discuss such approaches in Section 3.2.3.

Inspired by differentiable programming software, we design `Yao` to be around the domain-specific computational graph, the quantum block intermediate representation QBIR). A block refers to a tensor representation of quantum operations, which can be quantum circuits and quantum operators of various granularities (quantum gates, Hamiltonian, or the whole program). As shown in Figure 2.1, QBIR offers a hardware-agnostic abstraction of quantum circuits. It is called an intermediate representation due to its stage in the quantum compilation, which bridges the high-level quantum algorithms and low-level device-specific instructions. `Yao` provides rich functionalities to construct, inspect, manipulate, and differentiate quantum circuits in terms of QBIR.

`Yao` adds a unique solution to the landscape of open-source quantum computing software, includes `Quipper` [85], `ProjectQ` [86], `Q#` [87], `Cirq` [88], `qulacs` [89], `PennyLane` [90], `qiskit` [91], and `QuEST` [92]. References [93–95] contain more complete surveys of quantum software. Most software represents quantum circuits as a sequence of instructions. Thus, users need to define their abstraction for circuits with rich structures. `Yao` offers QBIR and related utilities to compose and manipulate complex quantum circuits. `Yao`'s QBIR is nothing but an abstract syntax tree, which is a commonly used data structure in modern programming languages thanks to its strong expressibility for control flows and hierarchical structures. `Quipper` [85] has adopted a similar strategy for the functional programming of quantum computing. `Yao` additionally introduces `Subroutine` to manage the scope of active and ancilla qubits. Besides these basic features, `Yao` puts a strong focus on differentiable programming of quantum circuits, which will be discussed in Section 3.2.3. In this regards, `Yao`'s batched quantum register with GPU acceleration and built-in AD engine offers significant speedup and convenience compared to `PennyLane` [90] and `qulacs` [89].

The QBIR is a domain-specific representation for quantum operators, including circuits

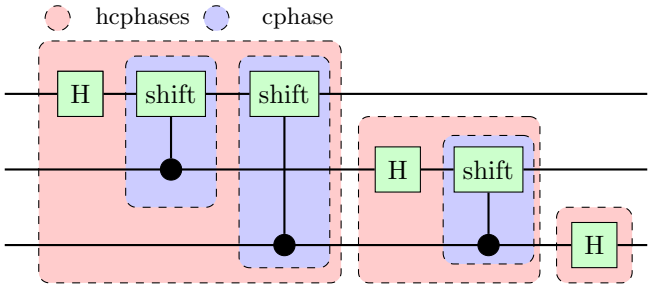

Figure 2.2: Quantum Fourier transformation circuit. The red and blue dashed blocks are built by the **hcphases** and **cphase** functions in the Listing.

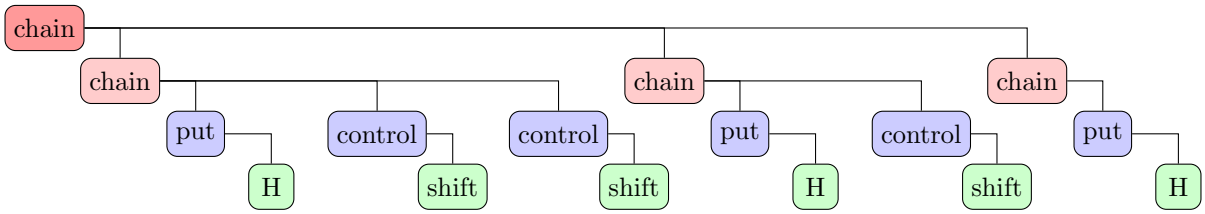

Figure 2.3: Quantum Fourier transformation circuit as a QBIR. The red nodes are roots of the composite **ChainBlock**. The blue nodes indicate the composite **ControlBlock** and **PutBlock**. Green nodes are primitive blocks.

and observables. Figure 2.2 shows the quantum Fourier transformation circuit [96–98] which contains the `hcphases` blocks (marked in red) of different sizes. Each block itself is also a composition of Hadamard gates and `cphase` blocks (marked in blue) on various locations. In `Yao`, it takes three lines of code to construct the QBIR of the QFT circuit.

The function `cphase` defines a control phase shift gate with the `control` and `shift` functions. The function `hcphases` defines the recursive pattern in the QFT circuit, which puts a Hadamard gate in the first qubit of the subblock and then chains it with several control shift gates. The `chain` block is a composition of blocks with the same number of qubits. It is equivalent to matrix multiplication in reverse order mathematically. Finally, one composes the QFT circuit of a given size by chaining the `hcphases` blocks. Overall, these codes construct a tree representation of the circuit shown in Figure 2.3. The subtrees are composite blocks (`ChainBlock`, `ControlBlock`, and `PutBlock`) with different composition relations indicated in their roots. The leaves of the tree are primitive blocks.

In `Yao`, to execute a quantum circuit, one can simply feed a quantum state into the QBIR.

Listing 4 Quantum Fourier transformation circuit in `Yao`.

```
julia> using Yao

julia> cphase(i, j) = control(i, j=> shift(
    2π/(2^(i-j+1))));

julia> hcphases(n, i) = chain(n, i==j ?
   put(i=>H) : cphase(j, i) for j in i:n);

julia> qft(n) = chain(hcphases(n, i)
                for i in 1:n)

julia> qft(3)
nqubits: 3
chain
├─ chain
│  ├─ put on (1)
│  │  └─ H gate
│  ├─ control(2)
│  │  └─ (1,) shift(1.5707963267948966)
│  └─ control(3)
│     └─ (1,) shift(0.7853981633974483)
├─ chain
│  ├─ put on (2)
│  │  └─ H gate
│  └─ control(3)
│     └─ (2,) shift(1.5707963267948966)
└─ chain
   └─ put on (3)
      └─ H gate
```

Listing 5 `apply!` and pipe.

```
julia> rand_state(3) |> qft(3);
        # same as apply!(rand_state(3), qft(3))
```

Listing 6 Inspecting gates.

```
julia> using Yao, SymEngine

julia> @vars θ
(θ,)

julia> shift(θ) |> mat
2×2 LinearAlgebra.Diagonal
{Basic,Array{Basic,1}}:
 1          ⋅
 ⋅   exp(im*θ)

julia> control(2,1,2=>shift(θ)) |> mat
4×4 LinearAlgebra.Diagonal{Basic,
                    Array{Basic,1}}:
 1  ⋅  ⋅          ⋅
 ⋅  1  ⋅          ⋅
 ⋅  ⋅  1          ⋅
 ⋅  ⋅  ⋅   exp(im*θ)
```

Here, we define a random state on 3 qubits and pass it through the QFT circuit. The pipe operator `|>` is overloaded to call the `apply!` function which applies the quantum circuit block to the register and modifies the register **inplace**.

The generic implementation of QBIR in `Yao` allows supporting both numeric and symbolic data types. For example, one can inspect the matrix representation of quantum gates defined in `Yao` with symbolic variables.

Here, the `@vars` macro declares the symbolic variable θ. The `mat` function constructs the matrix representation of a quantum block.

The QBIR in `Yao` is designed as an expression tree for quantum circuits. Readers can get a concrete feeling by inspecting the tree structure from examples of `Yao`. We will now discuss a more formal definition of a toy circuit representation in BNF. To design such a representation, we need to consider the primitive operations in quantum circuits, the elementary gates from various use cases, the composition of quantum circuits, and the measurement of quantum circuits. A simple BNF definition of the syntax for the

$$
\begin{aligned}
\langle \text{circuit} \rangle &::= \langle \text{gate} \rangle \mid \langle \text{gate} \rangle \langle \text{circuit} \rangle \\
\langle \text{gate} \rangle &::= \langle \text{composite} \rangle \mid \langle \text{primitive} \rangle \\
\langle \text{composite} \rangle &::= \langle \text{put} \rangle \mid \langle \text{control} \rangle \mid \langle \text{subroutine} \rangle \\
\langle \text{primitive} \rangle &::= \langle \text{shift} \rangle \mid \langle \text{Pauli} \rangle \mid \ldots \\
\langle \text{put} \rangle &::= \langle \text{gate} \rangle \langle \text{index} \rangle \\
\langle \text{control} \rangle &::= \langle \text{gate} \rangle \langle \text{index} \rangle \langle \text{index} \rangle
\end{aligned}
$$

Figure 2.4: The BNF definition of the syntax for the quantum circuits in `Yao`. ⟨index⟩ refers to an expression representing index

quantum circuits in `Yao` is shown in Figure 2.4. For readers unfamiliar with BNF, we refer to Section 1.3.4 for a brief introduction to simple arithmetic expressions in BNF. The ⟨primitive⟩ expression contains the definitions of basic quantum gates. Thus, it can be extended to include more gates. The ⟨composite⟩ expression contains the definitions of the composition of quantum circuits.

The circuit expression might be the simplest expression for quantum objects. However, the expression we introduced in `Yao` shows a lack of expressiveness when we look at broader quantum operators. We will discuss this in the next section by introducing the `Bloqade` package built on top of `Yao`.

2.2 Quantum Registers

The quantum register stores hardware-specific information about the quantum states. In classical simulation on a Central Processing Unit (CPU), the quantum register is an array containing the quantum wave function. For GPU simulations, the quantum register stores the pointer to a GPU array. In an experiment, the register should be the quantum device that hosts the quantum state. `Yao` handles all of these cases with a unified `apply!` interface, which dispatches the instructions depending on different types of QBIR nodes and registers.

Listing 7 CUDA register

```
julia> using CuYao

# construct the |1010> state
julia> r = ArrayReg(bit"1010");

# transfer data to CUDA
julia> r = cu(r);
```

Instructions on Quantum Registers

Quantum registers store quantum states in contiguous memory, which can either be the CPU memory or other hardware memory, such as a CUDA device.

Each register type has its own device-specific instruction set. They are declared in `Yao` via the "instruction set" interface, which includes

- **gate instruction**: `instruct!`
- **measure instruction**: `measure` and `measure!`
- **qubit management instructions**: `focus!` and `relax!`

The instruction interface provides a clean way to extend support to various backends without the user worrying about changes to frontend interfaces. We note that the function with a `!` suffix modifies the register in place. This particular convention in Julia indicates that the function modifies its argument. In the experiment, the instruction `measure` is impossible to implement due to the non-cloning theorem.

For example, the rotation gate shown in Figure 2.1 is interpreted as `instruct!(reg, Val(:Rx), (2,),` θ`)`. The second parameter specifies the gate, which is a `Val` type with a gate symbol as a type parameter. The `Val` type is a Julia type that carries a value at compile time. This allows the compiler to specialize the function for the specific gate. The third parameter is the qubit to apply, and the fourth parameter is the rotation angle. The CNOT gate is interpreted as `instruct!(reg, Val(:X), (1,), (2,), (1,))`, where the last three tuples are gate locations, control qubits, and configuration of the control qubits (0 for inverse control, 1 for control). Respectively. The `measure` function simulates measurement from the quantum register and provides bit strings, while `measure!` returns the bit string and also collapses the state.

Listing 8 Instructions on quantum registers

```
julia> r = zero_state(4);

julia> instruct!(r, Val(:X), (2, ))
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 4/4

julia> samples = measure(r; nshots=3)
3-element Array{BitBasis.BitStr{4,Int64},1}:
 0010 (2)
 0010 (2)
 0010 (2)

julia> [samples[1]...]
4-element Array{Int64,1}:
 0
 1
 0
 0
```

In the last line of the above example, we convert a bit string $0010_{(2)}$ to a vector `[0, 1, 0, 0]`. Note that the order is reversed since the readout of a bit string is in the little-endian format.

Active qubits and environment qubits

In certain quantum algorithms, one only applies the circuit block to a subset of qubits. For example, see the quantum phase estimation [33] shown in Figure 2.5.

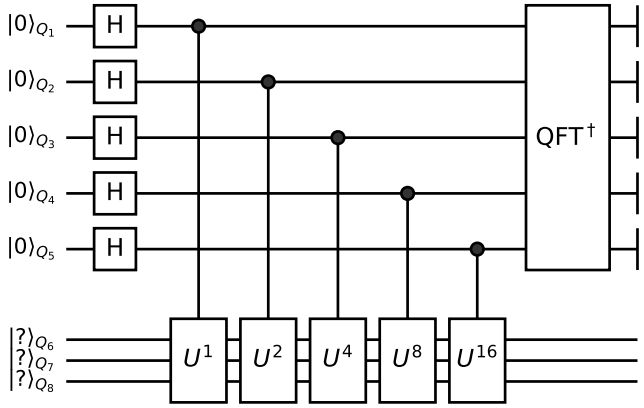

Figure 2.5: 5-qubit quantum Phase estimation circuit. This circuit contains three components. First, apply Hadamard gates to n ancilla qubits. Then, the controlled unitary is applied to $n + m$ qubits, and finally, the inverse QFT is applied to n ancilla qubits.

The QFT circuit block defined in Listing 4 can not be used directly in this case since the block size does not match the number of qubits. We introduce the concept of active and environment qubits to address this issue. Only the active qubits are visible to circuit blocks under operation. We manage the qubit resources with the `focus!` and its reverse `relax!` instructions.

Since it is a recurring pattern to first `focus!`, then `relax!` on the same qubits in many quantum algorithms, we introduce a `Subroutine` node to manage the scope automatically. Hence, the phase estimation circuit in Figure 2.5 can be defined with the following codes.

The `matblock` method in the codes constructs a quantum circuit from a given unitary matrix.

Listing 9 focus! and relax!

```
julia> reg = rand_state(10)

julia> focus!(reg, (3,6,1,2))

julia> reg |> qft(4)

julia> relax!(reg, (3,6,1,2); to_nactive=10)
```

Listing 10 quantum phase estimation

```
PE(n, m, U) = chain(
    n+m, # total number of qubits
    repeat(H, 1:n), # apply H from 1:n
    chain(control(
        k,
        n+1:n+m=>matblock(U^(2^(k-1))))
        for k in 1:n
    ),

    # apply inverse QFT on a local scope
    subroutine(qft(n)', 1:n)
)
```

Listing 11 batched quantum registers

```
julia> reg = rand_state(4; nbatch=5);

julia> reg |> qft(4) |> measure!
5-element Array{BitBasis.BitStr{4,Int64},1}:
 1011 (2)
 1011 (2)
 0000 (2)
 1101 (2)
 0111 (2)
```

Batched Quantum Registers

The batched register is a collection of quantum wave functions. It can be samples of classical data for quantum machine learning tasks [99] or an ensemble of pure quantum states for thermal state simulation [100]. For both applications, having the batch dimension not only provides convenience but may also significantly speed up the simulations.

We adopt the Single Program Multiple Data (SPMD) [101] design in `Yao` similar to modern machine learning frameworks so that it can make use of modern multi-processors such as multi-threading or GPU support (and potentially multi-processor QPUs). Applying a quantum circuit to a batched register means applying the same quantum circuit to a batch of wave functions in parallel, which is extremely friendly to modern multi-processors. SPMD is also adopted in `Bloqade` to save compilation time and improve experimental task latency.

The memory layout of the quantum register is a matrix of the size $2^a \times 2^r B$, where a is the number of system qubits, r is the number of remaining qubits (or environment qubits), B is the batch size. For gates acting on the active qubits, the remaining qubits and batch dimension can be treated on an equal footing. We put the batch dimension as the last dimension because Julia array is column majored. As the last dimension, it favors broadcasting on the batch dimensions.

One can construct a batched register in `Yao` and perform operations on it. These operations are automatically broadcasted over the batch dimension.

Note that we have used the `measure!` function to collapse all batches.

The measurement results are represented in `BitStr` type which is a subtype of `Integer`

and has a static length. Here, it pretty-prints the measurement results and provides a convenient readout of measuring results.

2.3 Quantum Operators

When we start looking at broader quantum operators, additional composition operations are needed. In `Yao`, this is done by providing two extra composition operations – ⟨Add⟩ and ⟨Scale⟩ to represent the addition and scalar multiplication of quantum operators. As a step further, we build the corresponding representations for Hamiltonian on top of QBIR. In `Yao`, the Hamiltonian is represented as a recursive addition of quantum operators. For example, the $1D$ transverse field Ising model

$$H = \sum_i \sigma_i^z \sigma_{i+1}^z - h \sum_i \sigma_i^x \tag{2.1}$$

is represented as the following

Listing 12 the $1D$ transverse field Ising model

```
julia> transverse_ising(4, 1.0)
nqubits: 4
+
├─ +
│  ├─ repeat on (1, 2)
│  │  └─ Z
│  ├─ repeat on (2, 3)
│  │  └─ Z
│  ├─ repeat on (3, 4)
│  │  └─ Z
│  └─ repeat on (1, 4)
│     └─ Z
└─ [+] +
       ├─ put on (1)
       │  └─ X
       ├─ put on (2)
       │  └─ X
       ├─ put on (3)
       │  └─ X
       └─ put on (4)
          └─ X
```

On top of this representation, we developed the representations for Rydberg atoms in `Bloqade`[102] for simulating the dynamics. However, the symbolic expressions are not enough for the numerical simulations. We further developed the representations for simulating dynamics as the sum of linear operators. The data structure of this representation is a composition of a list of linear operators, such as dense matrix, sparse matrix, etc., and a list of time-dependent coefficients. This representation throws away some of the symbolic information but adds the numerical details on sparsity. We are removing the need for sparse matrix addition, which is costly. Moreover, this representation allows further specialization in matrices to utilize special matrix multiplication, such as the general permutation matrix, Pauli matrix, etc.

In the Rydberg atoms, we further developed the representations for representing more structured Hamiltonian coefficients, called the pulse sequence. This representation is a set of time-dependent Hamiltonian coefficients for programming analog Hamiltonian devices.

we will discuss them in the Section 2.4. However, in the development of the representations, we found limitations of QBIR in representing the time-dependent Hamiltonian coefficients as well as broader quantum operators. More specifically, the QBIR is designed mainly for quantum circuits instead of broader quantum operators. Thus, as a result, information about the time-dependent Hamiltonian coefficients is wrapped as a black box in the QBIR. However, the ⟨Add⟩ and ⟨Scale⟩ operations work well for simple and small-scale cases. When working with symmetries and large-scale (> 1000 sites) or high-dimensional ($3D$ or $4D$) expressions, this becomes a bottleneck quickly – the symmetry information is hiding behind the 1000 ⟨Add⟩ nodes, and the memory of storing such expression grows linearly with number of terms. This becomes an obstacle for further compiling and verifying the pulse sequence for hardware. On the other hand, in the case of general quantum operators, the QBIR lacks the information about the basis, which is necessary for the numerical simulations and many other algebraic transforms.

$$
\begin{aligned}
\langle \text{operator} \in \mathcal{H} \rangle &::= \langle \text{const} \rangle \mid \langle \text{primitive} \rangle \mid \langle \text{intrinsic} \rangle \mid \langle \text{call} \rangle \mid \langle \text{annotation} \rangle \\
\langle \text{primitive} \rangle &::= \texttt{0} \mid I \mid X \mid Y \mid Z \cdots \\
\langle \text{intrinsic} \rangle &::= \langle \text{linalg} \rangle \mid \langle \text{subscript} \rangle \mid \langle \text{reduction} \rangle \mid \langle \text{outer} \rangle \\
\langle \text{call} \rangle &::= \langle \text{name} \rangle \, \langle \text{args} \rangle \\
\langle \text{annotation} \rangle &::= \langle \text{operator} \rangle \, \texttt{\%} \, \langle \text{basis} \rangle \\
\langle \text{linalg} \rangle &::= \langle \text{add} \rangle \mid \langle \text{mul} \rangle \mid \langle \text{kron} \rangle \mid \langle \text{comm} \rangle \mid \langle \text{acomm} \rangle \mid \\
&\quad \langle \text{pow} \rangle \mid \langle \text{kronpow} \rangle \mid \langle \text{adjoint} \rangle \mid \langle \mathcal{T} \rangle \mid \langle \text{unary} \rangle \\
\langle \text{subscript} \rangle &::= \langle \text{operator} \rangle \, \texttt{[} \, \langle \text{index} \in \mathbb{N} \rangle \, \texttt{]} \\
\langle \text{reduction} \rangle &::= \langle \text{sum} \rangle \mid \langle \text{prod} \rangle \\
\langle \text{outer} \rangle &::= \langle \text{outer} \rangle \, \langle \text{state} \rangle \, \langle \text{state} \rangle \\
\langle \text{add} \rangle &::= \langle \text{scalar} \rangle \, \langle \text{operator} \rangle \mid \langle \text{scalar} \rangle \, \langle \text{operator} \rangle \, \texttt{+} \, \langle \text{add} \rangle \\
\langle \text{mul} \rangle &::= \langle \text{operator} \rangle \, \texttt{*} \, \langle \text{operator} \rangle \\
\langle \text{kron} \rangle &::= \langle \text{operator} \rangle \otimes \langle \text{operator} \rangle \\
\langle \text{comm expr} \rangle &::= \langle \text{comm bracket} \rangle \, \langle \text{operator} \rangle \, \langle \text{operator} \rangle \mid \\
&\quad \langle \text{comm bracket} \rangle \, \langle \text{operator} \rangle \, \langle \text{operator} \rangle \, \langle \text{power} \in \mathbb{N} \rangle \\
\langle \text{comm bracket} \rangle &::= \texttt{comm} \mid \texttt{acomm} \\
\langle \text{pow} \rangle &::= \texttt{pow} \, \langle \text{operator} \rangle \, \langle \text{operator} \rangle \\
\langle \text{kronpow} \rangle &::= \texttt{kronpow} \langle \text{operator} \rangle \, \langle \text{power} \in \mathbb{N} \rangle \\
\langle \text{adjoint} \rangle &::= \langle \text{operator} \rangle^\dagger \\
\langle \text{time-ordered} \rangle &::= \mathcal{T} \langle \text{operator} \rangle \\
\langle \text{unary} \rangle &::= \langle \text{builtin unary} \rangle \, \langle \text{operator} \rangle \\
\langle \text{builtin unary} \rangle &::= \texttt{exp} \mid \texttt{log} \mid \texttt{inv} \mid \texttt{sqrt} \mid \texttt{conj} \mid \texttt{transpose} \\
\langle \text{outer} \rangle &::= \langle \text{state} \rangle \, \langle \text{state} \rangle
\end{aligned}
$$

Figure 2.6: The BNF definition of the syntax for the quantum operators in `Liang`.

To address these limitations, we developed the symbolic expression for broader quantum operators in `Liang`, including several different expressions for scalar, quantum operators, states, and basis. Their formal definitions are introduced in the following. The formal syntax of symbolic operator expression is defined in Figure 2.6. The expression follows a

more complicated tree structure compared to our circuit expression. We will introduce the details of the expression in the following.

⟨const⟩ and ⟨primitive⟩ operators

The ⟨const⟩ are numerical values similar to a literal value in programming languages. It can be a dense matrix, sparse matrix, Pauli string, or general permutation matrix. The ⟨primitive⟩ are the basic quantum operators, such as I, X, Y, Z, etc. The ⟨primitive⟩ will be lowered into the ⟨const⟩ in the compilation process when the basis is provided.

Basis Annotation

⟨basis⟩ ::= ⟨space⟩ ⟨operator⟩
⟨space⟩ ::= ⟨primitive⟩ | ⟨product⟩ | ⟨pow⟩ | ⟨subspace⟩
⟨primitive⟩ ::= `qubit` | `qudit` ⟨index ∈ ℕ⟩ | `spin` ⟨index⟩

Figure 2.7: The BNF definition of the syntax for the quantum basis in `Liang`.

The design of basis annotation primarily considers the composibility between expressions. Instead of requiring an explicit basis annotation at each operator expression, we ask a compiler to infer the basis information automatically when possible. For example, the Rydberg Hamiltonian

$$H = \sum_{\langle i,j \rangle} \frac{C}{\|\mathbf{r}_i - \mathbf{r}_j\|^6} n_i n_j + \sum_i \Omega_i \sigma_i^x - \sum_i \Delta_i n_i \tag{2.2}$$

can be defined on either ground states $|0\rangle, |r\rangle$ or hyperfine states $|1\rangle, |r\rangle$. Thus, a Rydberg atom array with 3 levels can be written as

$$H = H_{|0r\rangle} + H_{|1r\rangle} \tag{2.3}$$

If we require the definition to be annotated with the basis, e.g

$$H_{|0r\rangle} = \sum_{\langle i,j \rangle} \frac{C}{\|\mathbf{r}_i - \mathbf{r}_j\|^6} (n_i : |0r\rangle)(n_j : |0r\rangle) + \sum_i \Omega_i \sigma_i^x : |0r\rangle - \sum_i \Delta_i n_i : |0r\rangle \tag{2.4}$$

the expression $H_{|0r\rangle}$ will not be able to be repurposed for the expression $H_{|1r\rangle}$. Thus, instead, we allow the basis to be annotated only when necessary, as one naturally writes the expression

$$H = H : |0r\rangle + H : |1r\rangle \tag{2.5}$$

The compiler is responsible for inferring the basic information for the sub-expressions.

Addition and Scalar Multiplication

Inspired by previous work in more general purpose symbolic engine [103], we adopt the compact data structure of storing operations that are communicative and associative using a dictionary of expression and scalar coefficients, where the keys are the scalar coefficients and the values are the expressions. This removes redundancy. Equivalent forms of the expression, such as $2A + 3B$ and $3B + 2A$, are equivalent. All the expressions are further designed to be hash constant, which allows the compiler to check the equivalence of the expressions quickly. Thus, similar term merging is automatically done by dictionary operations.

Subscript and Reduction

The ⟨subscript⟩ expression is a way to define the site index of the quantum operator. When the corresponding ⟨index⟩ is a constant, it becomes a convenient short-hand for ⟨kron⟩ operations. The ⟨reduction⟩ expression is a way to define the operators on a large number of sites. For example, the transverse field Ising model on Kagome lattice can be written as

Listing 13 the transverse field Ising model on Kagome lattice in `Liang`

```
sum(
    Lattice.bonds(Lattice.kagome(5)),
    [index"i", index"j"]=>Op.Z[index"i"] * Op.Z[index"j"]
) + scalar"h" * sum(
    Lattice.sites(Lattice.kagome(5)),
    [index"i"]=>Op.X[index"i"]
)
```

This expression is more generic than what we have in `Yao`, and preserves the geometry information of the lattice without linearly growing the memory usage thus it is more suitable for large-scale representations and potentially can serve as a high-level representation for quantum hardware at 10^4 sites scale. The basis inference is also supported in such expression by virtually evaluating the reduction loop.

Outer Product and States

$$
\begin{aligned}
\langle \text{state} \in \mathcal{H} \rangle &::= \texttt{zero} \mid \langle \text{eigen} \rangle \mid \langle \text{product} \in \mathbb{N}^{\otimes n} \rangle \mid \langle \text{kron} \rangle \mid \langle \text{add} \rangle \mid \langle \text{annotate} \rangle \\
\langle \text{eigen} \rangle &::= \langle \text{operator} \rangle \langle \text{index} \in \mathbb{N} \rangle \\
\langle \text{kron} \rangle &::= \langle \text{state} \rangle \langle \text{state} \rangle \\
\langle \text{add} \rangle &::= \langle \text{scalar} \rangle \langle \text{state} \rangle \mid \langle \text{scalar} \rangle \langle \text{state} \rangle \texttt{ + } \langle \text{add} \rangle \\
\langle \text{annotate} \rangle &::= \langle \text{state} \rangle \texttt{ \% } \langle \text{basis} \rangle
\end{aligned}
$$

Figure 2.8: The BNF definition of the syntax for the quantum states in `Liang`.

The most generic form of the state is a list of complex numbers representing the amplitudes. However, when we look at more specific quantum states, especially in terms of a symbolic expression, the representation can often be just the addition of a few product states. And the product states are usually the eigenstates of a quantum operator. Thus, we define the $\langle$state$\rangle$ expression as an expression composed of $\langle$eigen$\rangle$, $\langle$kron$\rangle$ and $\langle$add$\rangle$. Additionally, one can represent the configuration of a basis state as $|041012\rangle$ without explicitly writing

the operator. When the basis is given for such sites, the compiler can infer the basis information for each site, e.g

Listing 14 inferring basis for product states `Liang`

```
kron(Op.X % Qubit, Op.Z % Spin(1)) * State.Product([0, 1])
```

2.4 Quantum Hardwares

The representation for hardware can have multiple layers ranging from low-level control signals to high-level pulse programs for an analog Hamiltonian. This section will mainly discuss the high-level pulse program of an analog Hamiltonian that we developed in the `bloqade-python` software. The high-level pulse program we discuss in this section are the details of Ω (Rabi frequency), Δ (detuning), and ϕ (phase) of the following analog Rydberg Hamiltonian

$$H = \sum_{\langle i,j \rangle} \frac{C_6}{\|\mathbf{r}_i - \mathbf{r}_j\|} n_i n_j + \sum_i \frac{\Omega}{2} (e^{i\phi_i} |g_i\rangle \langle r_i| + e^{-i\phi_i} |r_i\rangle \langle g_i|) - \sum_i \Delta_i n_i \tag{2.6}$$

where C is the interaction constant that depends on the particular Rydberg state used. For our reference device in [104], we use $C_6 = 862690 \times 2\pi \text{MHz}\mu\text{m}^6$ for $|r\rangle = \left|70S_{1/2}\right\rangle$ of the ^{87}Rb atoms, where $|g_i\rangle = |0\rangle$ or $|g_i\rangle = |1\rangle$ if the hyperfine state is used. In `Bloqade`, the primary goal is to emulate this device at high-level pulse program. Thus the representation is the same as `Yao`'s QBIR with `Add` extension. In `bloqade-python`, the primary goal becomes actually running programs on such device. More consideration has been put into designing the representation of time sequence $\Omega(t)$ and $\Delta(t)$ at both $|0\rangle$ and $|1\rangle$ levels as well as the capability of utilizing arbitrary geometry of the device. Furthermore, considering the use cases from varitional quantum algorithms, we also need to consider parameterized pulse programs, and allow our representation to be capable of optimizing or sweeping over parameters.

We first introduce a high-level representation of the pulse program aiming to ease the construction of such programs. An example of creating a Z2 state in $1D$ Rydberg atom chain is demonstrated in Listing 15. A chain of methods constructs the program calls, namely the "builder"s. This is a typical design pattern for building complicated objects in Python with linear method calls. Such construction is made possible because, in analog mode, the Rydberg atom array cannot have feedback during the execution, thus resulting in

no control flows. The execution procedure behind these pulse programs is some paralleled linear programs. Therefore, the procedure for creating such programs always has a fixed order. With such an assumption, we can use the builder pattern and Python's method chaining to guide the construction of the pulse program with the help of the editor's auto-completion feature. This is a good example of how the representation can be designed to guide the user in constructing the program correctly.

Listing 15 $1D$ Z2 state preparation in `bloqade-python`

```
# Define relevant parameters for the lattice geometry and pulse schedule
n_atoms = 11
lattice_spacing = 6.1
min_time_step = 0.05
omega_max = np.pi * 2 * 4.3
U = 2 * np.pi * 15.0
# Define Rabi amplitude and detuning values.
# Note the addition of a "sweep_time" variable
# for performing sweeps of time values.
rabi_amplitude_values = [0.0, omega_max, omega_max, 0.0]
rabi_detuning_values = [-U, -U, U, U]
durations = [0.3, "sweep_time", 0.3]

time_sweep_z2_prog = (
    Chain(n_atoms, lattice_spacing=lattice_spacing)
    .rydberg.rabi.amplitude.uniform
        .piecewise_linear(durations, rabi_amplitude_values)
    .detuning.uniform.piecewise_linear(durations, rabi_detuning_values)
)

# Allow "sweep_time" to assume values from 0.05 to
# 2.4 microseconds for a total of 20 possible values.
# Starting at exactly 0.0 isn't feasible, so we use
# the `min_time_step` defined previously.
time_sweep_z2_job = time_sweep_z2_prog.batch_assign(
    sweep_time=np.linspace(min_time_step, 2.4, 20)
)
```

The pulse program created in Listing 15 creates a pulse program demonstrated in Fig-

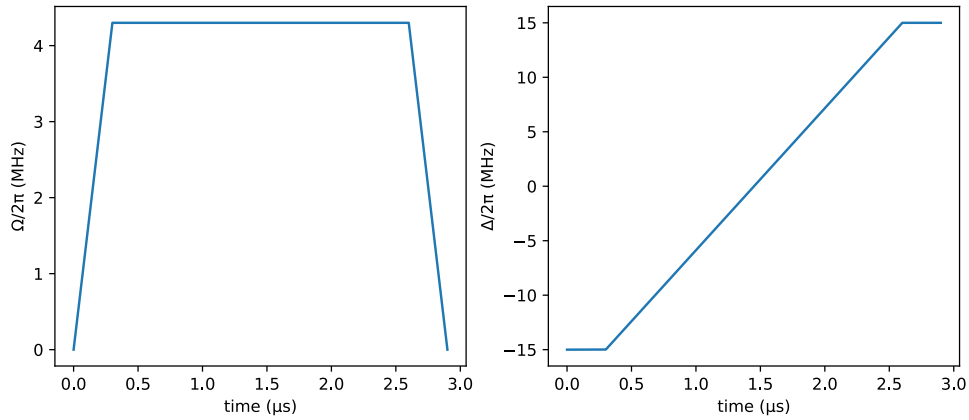

Figure 2.9: Rabi frequency amplitude specified by this pulse program with `sweep_time` assigned to 2.3

ure 2.9. The pulse controls the Rydberg atom array to evolve into a Z2 state adiabatically. We can simulate the time evolution and check the density on the Rydberg (excited) state as shown in Figure 2.10.

A formal IR is designed to model the analog machine Aquilla from QuEra before sending it to the hardware [104]. This IR is designed similarly to the expression structure as mentioned previously, with extra considerations on serialization over web protocols. We introduce the formal definition of this IR in Figure 2.11.

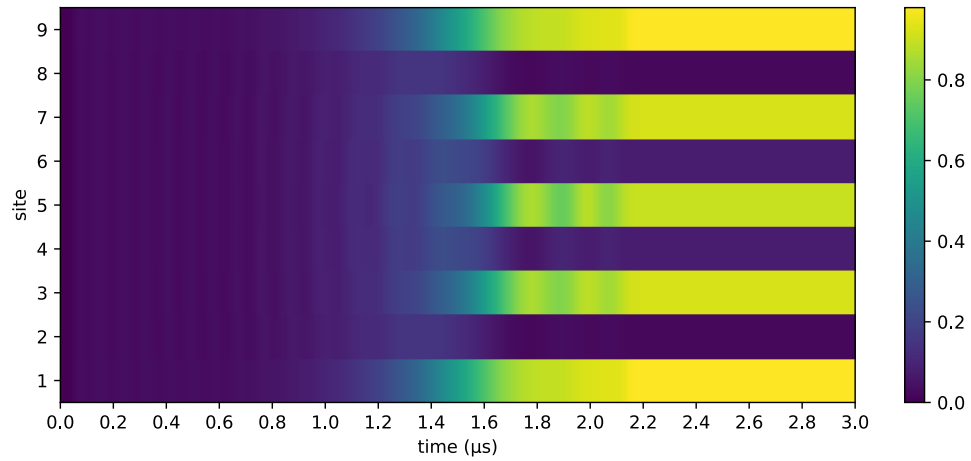

Figure 2.10: Adiabatic evolution that prepares a Z2 state

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{sequence} \rangle \mid \langle \text{named} \rangle \mid \langle \text{slice} \rangle \\
\langle \text{sequence} \rangle &::= \langle \text{sequence} \rangle \langle \text{pulse} \rangle \mid \texttt{sequence} \langle \text{pulse} \rangle \\
\langle \text{named} \rangle &::= \texttt{named} \langle \text{string} \rangle \langle \text{expr} \rangle \\
\langle \text{slice} \rangle &::= \texttt{slice} \langle \text{expr} \rangle \langle \text{interval} \rangle \\
\langle \text{pulse} \rangle &::= \langle \text{level coupling} \rangle \langle \text{pulse expr} \rangle \\
\langle \text{level coupling} \rangle &::= \texttt{rydberg} \mid \texttt{hyperfine} \\
\langle \text{pulse expr} \rangle &::= \langle \text{pulse slice} \rangle \mid \langle \text{fields} \rangle \mid \langle \text{concat} \rangle \\
\langle \text{pulse slice} \rangle &::= \langle \text{pulse expr} \rangle \langle \text{interval} \rangle \\
\langle \text{concat} \rangle &::= \langle \text{concat} \rangle \langle \text{pulse expr} \rangle \mid \langle \text{pulse expr} \rangle \langle \text{pulse expr} \rangle \\
\langle \text{fields} \rangle &::= \langle \text{field name} \rangle \langle \text{channels} \rangle \\
\langle \text{channels} \rangle &::= \langle \text{channel} \rangle \mid \langle \text{channels} \rangle \langle \text{channel} \rangle \\
\langle \text{field name} \rangle &::= \texttt{rabi amplitude} \mid \texttt{rabi phase} \mid \texttt{detuning}
\end{aligned}
$$

Figure 2.11: The IR for the pulse program in `bloqade-python`

The formal definition of the high-level IR is shown in Figure 2.11. The general program is constructed via the definition of atom positions and the ⟨expr⟩. The atom positions are a list of coordinates generated by lattice constructors. The ⟨expr⟩ is a description of the

waveforms at different level coupling (i.e., Rydberg or hyperfine) and various fields of the Rydberg Hamiltonian (i.e., Rabi amplitude Ω_i, Rabi phase ϕ_i, and detuning Δ_i). We define the ⟨sequence⟩ as a list of ⟨pulse⟩, and the ⟨sequence⟩ can be named or sliced to form a ⟨expr⟩. The ⟨pulse⟩ describes the waveforms at the different level coupling. The ⟨pulse expr⟩ describes the waveforms at various fields. Each ⟨field⟩ describes the waveforms at different terms in the Rydberg Hamiltonian. They can be composed by concatenation or slicing at every level of abstraction. We then further discuss the definition of the ⟨channel⟩ in Figure 2.12.

⟨channel⟩	::=	⟨spatial modulation⟩ ⟨waveform⟩
⟨spatial modulation⟩	::=	`uniform` | ⟨variable⟩ | ⟨assigned variable⟩ | ⟨scaled locations⟩
⟨variable⟩	::=	`var` ⟨string⟩
⟨assigned variable⟩	::=	`assign` ⟨variable⟩ ⟨decimal⟩
⟨scaled locations⟩	::=	⟨scaled location⟩ | ⟨scaled locations⟩⟨scaled location⟩
⟨scaled location⟩	::=	⟨location⟩ ⟨scalar⟩

Figure 2.12: The IR for the channel in `bloqade-python`

The ⟨channel⟩ provides information about the spatial modulation of the waveform. The spatial modulation means the atom to which the given waveform will be applied. The ⟨spatial modulation⟩ can take three types

1. ⟨uniform⟩, the waveform is applied to all atoms, which is also referred as the analog Hamiltonian.
2. ⟨variable⟩, the waveform is applied to a variable location determined by the input at runtime. This allows users to reconfigure atom positions between each shot.
3. ⟨assigned variable⟩, this is a special case of ⟨variable⟩ where the variable is assigned a value at the beginning of the program. This is useful for partially evaluated programs generated in the middle of compilation.
4. ⟨scaled locations⟩, the most general case where the waveform is applied to a group of atoms with a rescaling factor.

The ⟨location⟩ is simply an integer representing the unique ID of an atom. This is potentially compatible with representing atom shuttling in the future [105]. The ⟨scalar⟩ is a standard expression representing scalar values in $\mathbb{R}$. We will now introduce the ⟨waveform⟩ in Figure 2.13.

⟨waveform⟩	::=	⟨instruction⟩ | ⟨smooth⟩ | ⟨slice⟩ | ⟨append⟩ |
		⟨negative⟩ | ⟨scale⟩ | ⟨add⟩ | ⟨record⟩ | ⟨sample⟩ | ⟨aligned⟩
⟨instruction⟩	::=	⟨linear⟩ | ⟨constant⟩ | ⟨poly⟩ | ⟨python function⟩
⟨smooth⟩	::=	`smooth` ⟨kernel⟩ ⟨waveform⟩
⟨slice⟩	::=	⟨waveform⟩ ⟨interval⟩
⟨append⟩	::=	⟨waveform⟩ ⟨waveform⟩
⟨negative⟩	::=	`-` ⟨waveform⟩
⟨scale⟩	::=	⟨waveform⟩ ⟨scalar⟩
⟨add⟩	::=	⟨waveform⟩ `-` ⟨waveform⟩
⟨record⟩	::=	`record` ⟨string⟩ ⟨waveform⟩
⟨sample⟩	::=	`sample` ⟨waveform⟩ ⟨interpolation⟩ ⟨scalar⟩
⟨aligned⟩	::=	⟨alignment⟩ ⟨waveform⟩
⟨alignment⟩	::=	`left` | `right`
⟨record⟩	::=	⟨waveform⟩ ⟨variable⟩ ⟨side⟩
⟨side⟩	::=	`start` | `end`

Figure 2.13: The IR for the waveforms in `bloqade-python`

The ⟨waveform⟩ is defined by composing primitive functions defined on $f : \mathbb{R} \to \mathbb{R}$ including ⟨linear⟩, ⟨constant⟩, ⟨poly⟩, and ⟨python function⟩, where ⟨linear⟩ represents a simple linear function, ⟨constant⟩ represents a constant function without varying with time, ⟨poly⟩ represents a polynomial function, and ⟨python function⟩ represents a foreign function defined in python by the user. The ⟨smooth⟩ is a function that smooths the waveform with a kernel, such as a moving average kernel, as used in the previous experiment [106]. The ⟨slice⟩, ⟨append⟩ are similar to previous composition operations. ⟨add⟩ adds up the amplitude of two waveforms. ⟨record⟩ stores the waveform amplitude value at the start or end of the waveform. This semantic exists because in experiment, one will need to scan the duration of waveform. Due to the nature of piecewise waveform, the start and end

points of the waveform will vary. For example, consider the example in **??** of preparing a quantum scar in experiment [41, 107]. The ⟨sample⟩ is a function that samples the waveform into an interpolation of a given number of points. This emulates how the machine works under the scene and thus can help our compiler identify hardware constraints before sending the program to the hardware. The ⟨aligned⟩ node is for aligning the schedule with waveforms on other spatial modulation. This is useful when the program contains multiple local controls. While more flexible asynchronous semantics may be required to express a more complicated schedule, we find such semantic is sufficient for a variety of use cases in the current hardware introduced in [104].

2.5 Static Single Assignment Form

Our previous discussion introduced the expression for quantum circuits, operators, and hardware. However, when dealing with programs with control flows, such as an error correction protocol where one needs to measure a qubit and decide the next operation based on the measurement result, using expression as a representation becomes no longer convenient. This section will introduce a different representation called Static Single Assignment (SSA) for representing quantum programs.

The SSA is usually used as an intermediate representation. Here, IR refers to the representation of a program that is used as an intermediate step between the source code and the target code. It is usually a format designed mainly for the convenience of the compiler, rather than for the convenience of the human reader.

SSA IR [108] is a widely adopted IR in traditional compiler engineering, such as LLVM [49]. This form offers a simple way to handle control flows and run data flow analysis on the program. **Basic Block**, a basic block contains a set of statements, where the last statements are branches that terminate the instruction stream, such as a `goto` statement. The return value of each statement will be assigned to a variable which will only be statically assigned once. **Terminator** A terminator is a control flow statement that jumps to another basic block, such as `goto`, `return`. **PhiNode** a ϕ-node represents the possible value from various branches. Thus, in SSA IR, the basic blocks define the vertices of a control flow graph (CFG), and the terminators define the edges in the CFG. In the Julia compiler, the SSA IR is introduced as `IRCode` or `CodeInfo`.

We extend the static SSA IR for quantum programs using a similar structure to M IR [109]'s region in Julia compiler's SSA IR. Besides the basic block and terminator, we define two new building blocks in the control flow graph of SSA IR, which are the **Quantum**

Block and **Quantum Terminator**. A quantum block is a list of statements that only contains pure quantum operations, such as applying a gate, and a quantum terminator is usually the measurement operation that causes the program jump from a quantum block to a classical statement or a basis change operation which cause the program jump from one quantum block on basis A to basis B. Thus, a quantum block is always inside a basic block. We note that a similar idea has been mentioned in Pennylane [90] referred to as quantum nodes inside the tape for automatic differentiation (also known as Wengert Lists [110]). However, at the time of writing, `PennyLane` does not support control flows or hybrid programs.

An intermediate observation is that a classical statement can be permutated with a quantum operation as long as they do not have variable dependencies. Thus, a straightforward algorithm can be found to group small quantum blocks into larger ones by looking up variable dependencies using standard data flow analysis on the SSA form program.

We do not aim to design an assembly language or architecture at this level; we only aim to provide a high-level intermediate representation for program analysis, compiler optimization, and code generation. Thus, our SSA IR is orthogonal to lower-level languages for machine execution (Quantum Assembly Language (QASM), QUIL, eQASM, etc.) or higher-level languages for better expressiveness (Silq, Q#, etc.).

Intrinsic Semantic

Unlike most quantum programming languages or intermediate languages, where qubits are treated as a primitive type and passed to gate operations in a function-like statement. We define our semantics as an operator-centric language. The operator language has been used in quantum physics for decades - everything happens in quantum mechanics can be described as an operator.

We find as a high-level semantic, this representation naturally becomes compatible with tensor network diagrams and quantum channels. By splitting out the semantics of qubits or registers from the core representation, we are able to generalize the program to an arbitrary basis, which is crucial to non-qubit-based systems such as a 3-level Rydberg atom system. The same operator `X` can be applied in the Rydberg system with either hyperfine or Rydberg pulse. Thus, once the user has defined the operator program, it can be reused on both hyperfine pulse and Rydberg pulse as long as the program does not have basis specifications.

On the other hand, mutability can be a problem in quantum programming due to the non-cloning theorem. Every quantum operation has to be mutable to avoid cloning on

the memory. However, the representation of operators is purely classical and thus can be immutable. Thus, the intrinsic semantics defined by operators can be pure functions whose return value does not depend on the system's state. This is a significant advantage for program analysis and transformation, as it allows the compiler to perform aggressive optimization on the program like the classical compiler.

Instead of an array of qubits objects with indexing semantics, the operator-centric semantic leads to the tensor index semantic of operators. Each operator in the program will have a local location. Calling another user-defined function in a local location results in an automatic location mapping.

Consider the quantum Fourier transform (QFT) example, we recursively call the `qft` function, inside a local location range `2:n`, the callee location space will be automatically mapped into `2:n`.

Listing 16 Quantum Fourier transform

```
@device function qft(n::Int)
    1 => H
    for k in 2:n
        @ctrl k 1 => shift(2π / 2^k)
    end

    if n > 1
        2:n => qft(n - 1)
    end
end
```

The `qft` function in Listing 16 is a recursive function that applies the Hadamard gate on the first qubit, then applies the controlled phase shift gates on the rest qubits, and finally calls itself on the rest qubits. The `@ctrl` macro is used to apply the controlled phase shift gates. The `@ctrl` macro is a syntax sugar that is equivalent to `control(control_location, gate, gate_location)` for the `control` function, which is used to apply a controlled gate. The `control` function takes two arguments: the control qubit and the target qubit. The `control` function is used to apply a controlled gate. The `control` function takes two arguments: the control qubit and the target qubit. This function is forwarded to Julia compiler to generate the SSA IR, combined with the compiler plugin provided by `YaoCompiler`, we obtain the following SSA IR.

Listing 17 A glance of generated SSA IR for QFT. `#<number>` refers to the basic block ID. `%<number> refers to the SSA variable.`

```
1 ── %1   = Base.getfield(var"#op#", :args)::Tuple{Int64}
│    %2   = Base.getfield(%1, 1, true)::Int64
│    %3   = Main.H::Any
│           YaoCompiler.Intrinsics.apply(
│                var"#register#", %3, $(QuoteNode(Locations(1)))
│           )::Any
│    %5   = Base.sle_int(2, %2)::Bool
│    %6   = Base.ifelse(%5, %2, 1)::Int64
│    %7   = Base.slt_int(%6, 2)::Bool
└────      goto "#3" if not %7
2 ──       goto "#4"
3 ──       goto "#4"
4 ┄─ %11  = φ ("#2" => true, "#3" => false)::Bool
│    %12  = φ ("#3" => 2)::Int64
│    %13  = φ ("#3" => 2)::Int64
│    %14  = Base.not_int(%11)::Bool
└────      goto "#10" if not %14
5 ┄─ %16  = φ ("#4" => %12, "#9" => %31)::Int64
│    %17  = φ ("#4" => %13, "#9" => %32)::Int64
│    %18  = invoke Base.power_by_squaring(2::Int64, %16::Int64)::Int64
│    %19  = Base.sitofp(Float64, %18)::Float64
│    %20  = Base.div_float(6.283185307179586, %19)::Float64
│    %21  = Main.shift::Any
⋮

29 ─ %113 = Base.string("got ", %107, " in parent space ", %43)::Any
│    %114 = YaoLocations.LocationError(%113)::Any
│           YaoLocations.throw(%114)::Union{}
└────      unreachable
30 ─       goto "#31"
31 ┄ %118 = φ ("#28" => %111, "#30" => nothing)::Core.Const(nothing)
└────      return %118
32 ─       return nothing
```

This SSA IR of QFT as a generic definition of the QFT circuit allows the creation of the QFT circuit on any number of qubits with classical controls representing the function recursion, given a concrete number of qubits `n`. The compiler can propagate the constant within the callee function, thus resulting in a pure quantum circuit.

Constant Propagation

Constant propagation further allows users to create abstractions and thus reuse the routines created by others. The user can define a high-level routine with control flows from the host language while running such a program on a quantum device that can only execute pure quantum circuits. The following is an example of calling other routines, and the compiler can automatically propagate the constant and generate the pure quantum circuit.

Listing 18 calling other routines with constant propagation

```
@device function test_basic(theta, phi)
    # syntax sugar
    1 => X
    @gate 2 => Z
    @ctrl 1 4 => Rx(theta)
    @ctrl 2 4 => Ry(phi)
    a = @measure 3
    # direct intrinsic
    apply(Y, 3)
    apply(X, 1, 4)
    c = measure(2)
    return (a = a, b = c)
end

@device function test_pure_quantum()
    ret = @gate 1:4 => test_basic(1.0, 2.0)
    @ctrl 2 1 => Rx(2.2)
    return ret
end
```

The compiler is able to generate the following SSA IR, which represents a pure quantum circuit.

Listing 19 A pure quantum circuit generated from Listing 18

```
1 ─ %1  = Main.X::Any
│         YaoCompiler.Intrinsics.apply(
                var"#register#", %1, $(QuoteNode(Locations(1)))
        )::Any
│   %3  = Main.Z::Any
│         YaoCompiler.Intrinsics.apply(
                var"#register#", %3, $(QuoteNode(Locations(2)))
        )::Any
│   %5  = Main.Rx::Any
│   %6  = (%5)(1.0)::Any
│         YaoCompiler.Intrinsics.apply(
                var"#register#", %6, $(QuoteNode(Locations(4))),
                $(QuoteNode(CtrlLocations(1)))
        )::Any
│   %8  = Main.Ry::Any
│   %9  = (%8)(2.0)::Any
│         YaoCompiler.Intrinsics.apply(
                var"#register#", %9,
                $(QuoteNode(Locations(4))), $(QuoteNode(CtrlLocations(2)))
        )::Any
│   %11 = invoke YaoCompiler.Intrinsics.measure(
            var"#register#"::AnyReg,
            $(QuoteNode(Locations(3)))::Locations{Int64}
        )::MeasureResult{Int64}
│   %12 = Main.Y::Any
│         YaoCompiler.Intrinsics.apply(
            var"#register#", %12, $(QuoteNode(Locations(3)))
        )::Any
│   %14 = Main.X::Any
│         YaoCompiler.Intrinsics.apply(
            var"#register#", %14,
            $(QuoteNode(Locations(1))), $(QuoteNode(CtrlLocations(4)))
        )::Any
│   %16 = invoke YaoCompiler.Intrinsics.measure(
            var"#register#"::AnyReg,
            $(QuoteNode(Locations(2)))::Locations{Int64}
        )::MeasureResult{Int64}
│   %17 = %new(
        NamedTuple{(:a, :b), Tuple{MeasureResult{Int64}, MeasureResult{Int64}}}, %11, %16
    )::NamedTuple{(:a, :b), Tuple{MeasureResult{Int64}, MeasureResult{Int64}}}
│   %18 = Main.Rx::Any
│   %19 = (%18)(2.2)::Any
│         YaoCompiler.Intrinsics.apply(
    var"#register#", %19,
    $(QuoteNode(Locations(1))), $(QuoteNode(CtrlLocations(2)))
    )::Any
└──       return %17
```

The inlining and elimination of dead code used by the above compilation are powered by the Julia compiler. Thus, we only provide the aggressive constant propagation strategy and the corresponding semantics of quantum instructions as a compiler plugin.

Discussion

The SSA IR is a convenient format for analyzing data flows between variables. Thus it allows optimization such as constant propagation. On the other hand, it simplifies the data flow analysis such as variable dependency analysis. Such analysis is critical for implementing automatic differentiation Section 3.2.

Supporting hybrid programs at high-level circuit/gate abstraction is not useful in practice due to the lack of hardware support on actually executing these classical instructions, and the latency of today's hardware does not require fast execution of classical program with control flows (i.e., the Rydberg atom array has shot rate only at 3 shot per second level [104]). On the other hand, algorithms require hybrid semantics such as error correction does not require general structured control flows but only requires a simple `if` statement, which can be easily hardcoded as a special intrinsic in the compiler (i.e., the `record` intrinsic in `stim` [111]). Moreover, even the quantum computing device is perfectly co-located with a classical processor. The low-latency control flows are likely programmed in lower-level semantics closer to the controller. Thus, the high-level quantum program is not the right place to handle control flows.

However, the general principle of program analysis and transformation finds its usefulness in lower-level programs in the quantum computation stack. For example, in the Rydberg atom array, the atom shuttling and control of the laser are naturally asynchronous operations that require classical control flows and data flow analysis to optimize the pulse program. Furthermore, real-time control requires optimization on the duration of instruction execution. Basic transformations, such as constant folding, variable dependency analysis, and dead code elimination, already provide value when compiling such programs. The SSA IR thus becomes a perfect fit at this level.

Chapter 3

Transformation

With representations defined in previous sections, we can now delve into the transformation of and between these representations. Almost all the problems can be formalized into transformations between the same or different representations. Transforming between the same representations is often called *optimization*, while transforming between different representations is often called *code generation*. Even simple transformations can be compelling in improving the performance of simulation. In this section, we will introduce the transformation for optimizing the exact simulation of quantum systems and the transformation for automatic differentiation.

3.1 Fast Exact Simulation

The most straightforward application of transformation is to optimize an existing simulation. We will first introduce the expression manipulation in `Yao`

3.1.1 Manipulating Quantum Circuits

In essence, QBIR represents the algebraic operations of a quantum circuit as types. Being an algebraic data type system, QBIR automatically allows pattern matching with Julia's multiple dispatch mechanics. Thus, one can manipulate quantum circuits in a straightforward manner using pattern matching on their QBIR.

For example, consider a practical situation where one needs to decompose the Hadamard gate into three rotation gates [112]. The codes in Listing 20 define compilation passes by

Listing 20 Gate decomposition for a QFT circuit

```
julia> decompose(x::HGate) =
    Rz(0.5π)*Rx(0.5π)*Rz(0.5π);

julia> decompose(x::AbstractBlock) =
    chsubblocks(x, decompose.(subblocks(x)));

julia> qft(3) |> decompose
nqubits: 3
chain
├─ chain
│  ├─ put on (1)
│  │  └─ chain
│  │     ├─ rot(ZGate, 1.5707963267948966)
│  │     ├─ rot(XGate, 1.5707963267948966)
│  │     └─ rot(ZGate, 1.5707963267948966)
│  ├─ control(2)
│  │  └─ (1,) shift(1.5707963267948966)
│  └─ control(3)
│     └─ (1,) shift(0.7853981633974483)
├─ chain
│  ├─ put on (2)
│  │  └─ chain
│  │     ├─ rot(ZGate, 1.5707963267948966)
│  │     ├─ rot(XGate, 1.5707963267948966)
│  │     └─ rot(ZGate, 1.5707963267948966)
│  └─ control(3)
│     └─ (2,) shift(1.5707963267948966)
└─ chain
   └─ put on (3)
      └─ chain
         ├─ rot(ZGate, 1.5707963267948966)
         ├─ rot(XGate, 1.5707963267948966)
         └─ rot(ZGate, 1.5707963267948966)
```

Listing 21 Inverse QFT

```
julia> iqft(n) = qft(n)';

julia> iqft(3)
nqubits: 3
chain
├─ chain
│  └─ put on (3)
│     └─ H gate
├─ chain
│  ├─ control(3)
│  │  └─ (2,) shift(-1.5707963267948966)
│  └─ put on (2)
│     └─ H gate
└─ chain
   ├─ control(3)
   │  └─ (1,) shift(-0.7853981633974483)
   ├─ control(2)
   │  └─ (1,) shift(-1.5707963267948966)
   └─ put on (1)
      └─ H gate
```

dispatching the `decompose` function on different quantum block types. For the generic `AbstractBlock`, we apply `decompose` recursively to all its sub-blocks and use the function `chsubblocks` defined in `Yao` to substitute the blocks. The recursion terminates on primitive blocks where `subblocks` returns an empty set. Due to the specialization of `decompose` method on Hadamard gates, a chain of three rotation gates are returned as a subblock instead.

Besides replacing gates, one can also modify a block by applying tags to it. For example, the `Daggered` tag takes the hermitian conjugate of the block. We use the `'` operator to apply the `Daggered` tag. Similar to the implementation of `Transpose` on matrices in Julia, the dagger operator in `Yao` is "lazy" in the sense that one simply marks the block as `Daggered` unless there are specific daggered rules defined for the block. For example, the hermitian conjugate of a `ChainBlock` reverses the order of its child nodes and propagate the `Daggered` tag to each subblock. Finally, we have the following rules for primitive blocks,

- Hermitian gates are unchanged under dagger operation
- The hermitian conjugate of a rotational gate $R_\sigma(\theta) \to R_\sigma(-\theta)$
- Time evolution block $e^{-iHt} \to e^{-iH(-t^*)}$
- Some special constant gates are hermitian conjugate to each other, e.g. `T` and `Tdag`.

With these rules, we can define the inverse QFT circuit directly in Listing 21.

3.1.2 Quantum Circuit Simulation

We will first introduce some basic routines for exact simulation of quantum circuits. Then in the next subsection, we will use pattern match dispatching matched circuits to these routines.

Brief Review of Operations in Quantum Circuits

To be simple, simulating quantum circuits, or to be more specific simulating how quantum circuits act on a quantum register, is about how to calculate large matrix-vector multiplication that scales exponentially. The most brute-force and accurate way of doing it via full amplitude simulation, which means we do this matrix-vector multiplication directly.

The vector contains the so-called quantum state and the matrices are quantum gate, which are usually small. The diagram of quantum circuits is a representation of these matrix multiplications. For example, the X gate is just a small matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.1}$$

In theory, there is no way to simulate a general quantum circuit (more precisely, a universal gate set) efficiently. However, in practice, we could still do it within a rather small scale with some tricks that make use of the structure of the gates. To understand how to calculate a quantum circuit, we need to introduce two kinds of mathematical operations

Tensor Product/Kronecker Product, this is represented as two parallel lines in the quantum circuit diagram, e.g

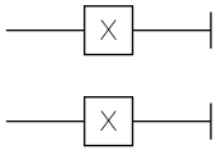

Figure 3.1: Kronecker product of two X gates

and by definition, this can be calculated by

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{12}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \\ a_{21}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{22}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \end{pmatrix} \tag{3.2}$$

Matrix Multiplication, this is the most basic linear algebra operation, we'll skip introducing this. In quantum circuit diagram, this is represented by blocks connected by lines.

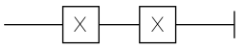

Figure 3.2: Kronecker product of two X gates

As a conclusion of this subsection, one can see simulating how pure quantum circuits act on a given quantum state is about how to implement some special type of matrix-vector multiplication efficiently. For readers familiar with Basic Linear Algebra Subprograms (BLAS), this kind of operations are only BLAS level 2 operations, which does not require any smart tiling technique and are mainly limited by memory bandwidth.

General Unitary Gate Subroutine

Thus the simplest way of simulating a quantum circuit is very straightforward: we can just make use of Julia's builtin functions: `kron` and `*`.

Listing 22 Simplest Quantum Circuit Simulation

```
using LinearAlgebra
function naive_broutine!(r::AbstractVector, U::AbstractMatrix, loc::Int)
    n = Int(log2(length(r))) # get the number of qubits
    return kron(I(1<<(n-loc+1)), U), I(1<<loc)
end
```

However, this is obviously very inefficient:

1. we need to allocate a $2^n \times 2^n$ matrix every time we try to evaluate the gate.
2. the length of the vector can only be 2^n, thus we should be able to calculate it faster with this knowledge.

if we know an integer is 2^n, it is straight forward to find out n by the following method:

Listing 23 `log2i` specialized for 64-bit integers

```
log2i(x::Int64) = if !signbit(x)
    (63 - leading_zeros(x))
else
    throw(ErrorException("nonnegative expected ($x)"))
end

log2i(x::UInt64) = 63 - leading_zeros(x)
```

this is because we already know how long our integer is in the program by looking at its type, thus simply minus the number of leading zeros would give us the answer. But don't forget to raise an error when it's an signed integer type. We can make this work on any integer type by the following way

Listing 24 `log2i` for arbitrary integers

```
for N in [8, 16, 32, 64, 128]
    T = Symbol(:Int, N)
    UT = Symbol(:UInt, N)
    @eval begin
        log2i(x::$T) =
            !signbit(x) ? ($(N - 1) - leading_zeros(x)) :
            throw(ErrorException("nonnegative expected ($x)"))
        log2i(x::$UT) = $(N - 1) - leading_zeros(x)
    end
end
```

the command `@eval` here is called a macro in Julia programming language, it can be used to generate code. The above code generates the implementation of `log2i` for signed and unsigned integer types from 8 bits to 128 bits.

Let's now consider how to write the general unitary gate acting on given locations of qubits.

Listing 25 Signature of `broutine!`

```
function broutine!(
    r::AbstractVector,
    U::AbstractMatrix,
    locs::NTuple{N, Int}
) where N
end
```

this matrix will act on certain qubits in the register, e.g., given an 8×8 matrix, we want it to act on the 1st, 4th, and 5th qubits. Based on the implementation of X and Z, we know this is about multiplying this matrix on the subspace of 1st, 4th, and 5th qubit, which means we need to construct a set of new vectors whose indices iterate over the subspace of `0xx00x`, `0xx01x`, `0xx10x`, `0xx11x` etc. Thus, we must first find a generic way to iterate through the subspace of `0xx00x`. Then, by adding an offset such as `1 << 1` to each index in this subspace, we can get the subspace of `0xx01x` etc.

To iterate through the subspace, we could iterate through all indices in the subspace.

Listing 28 Left moving

```
(xxx & ~0b001) << 1 + (xxx & 0b001) # = xx00x
```

For each index, we move each bit to its position in the whole space (from the first bit to the last). This will give us the first subspace, which is `0xx00x`.

Before we move on, we need to introduce the concept of binary masks: it is an integer that can help us "filter" out some binary values, e.g. we want to know if a given integer's 4th and 5th bit, we can use a mask `0b11000`, where its 4th and 5th bit is 1, the rest is 0, then we can use an and operation get the value. Given the location of bits, we can create a binary mask via the following `bmask` function

Listing 26 `bmask` function

```
function bmask(itr)
    isempty(itr) && return 0
    ret = 0
    for b in itr
        ret += 1 << (b - 1)
    end
    return ret
end
```

where `itr` is some iterable. However, there are quite a few cases which we don't need to create it via a `for`-loop, so we can specialize this function on the following types

Listing 27 `bmask` function specialized on `UnitRange`

```
function bmask(range::UnitRange{Int})
    ((1 << (range.stop - range.start + 1)) - 1) << (range.start - 1)
end
```

To move the bits in the subspace to the correct position, we need to iterate through all the contiguous regions in the bitstring, e.g., for `0xx00x`, we move the 2nd and 3rd bit in subspace by 3 bits together. This can be achieved by using a bit mask `001` and the following binary operation

We define this as a function called `lmove`. Now we need to generate all the masks by counting contiguous regions of the given locations

Listing 29 `group_shift` function

```
function group_shift(locations)
    masks = Int[]
    region_lens = Int[]
    k_prv = -1
    for k in locations
        # if current position in the contiguous region
        # since these bits will be moved together with
        # the first one, we don't need to generate a
        # new mask
        if k == k_prv + 1
            region_lens[end] += 1
        else
            # we generate a bit mask where the 1st to k-th bits are 1
            push!(masks, bmask(0:k-1))
            push!(region_lens, 1)
        end
        k_prv = k
    end
    return masks, region_lens
end
```

Now, to get the index in the whole space, we simply move each contiguous region by the length of their region, where the initial value of ‘index‘ is the subspace index, and after the loop, we will get the index in the whole space. To abstract the iteration further, we will define an abstraction called an iterator `BitSubspace` as the following

Listing 30 `BitSubspace` iterator

```
struct BitSubspace
    n::Int # total number of bits
    n_subspace::Int # number of bits in the subspace
    masks::Vector{Int} # masks
    region_lens::Vector{Int} # length of each region
end
```

And we can construct it via

Listing 31 `BitSubspace` constructor

```
function BitSubspace(n::Int, locations)
    masks, region_lens = group_shift(locations)
    BitSubspace(1 << (n - length(locations)), length(masks), masks, region_lens)
end
```

The corresponding whole-space index of each index in the subspace can be calculated by the following function

Listing 32 `next` function

```
@inline function Base.getindex(it::BitSubspace, i)
    index = i - 1
    for s in 1:it.n_subspace
        @inbounds index = lmove(index, it.masks[s], it.region_lens[s])
    end
    return index
end
```

We can loop through the subspace by overloading a few more interfaces, as shown below.

Listing 33 `iterate` function

```
Base.length(it::BitSubspace) = it.n
Base.eltype(::BitSubspace) = Int
@inline function Base.iterate(it::BitSubspace, st = 1)
    if st > length(it)
        return nothing
    else
        return it[st], st + 1
    end
end

julia> for each in BitSubspace(5, [1, 3, 4])
           println(string(each, base=2, pad=7))
       end
00000
00010
10000
10010
```

The next step is to perform the matrix-vector multiplication in the subspace. This requires generating the indices in the subspace. For example, for a unitary on the 1, 3, 4 qubits of a 5-qubit register, we need to multiply the matrix at `0xx0x`, `0xx1x`, `1xx0x` and `1xx1x`. Thus we can create the subspace of `x00x0` by `BitSubspace(5, [1, 3, 4])` and subspace of `0xx0x` by `BitSubspace(5, [2, 5])`, then add each index in `x00x0` to `0xx0x`, which looks like

Listing 34 Matrix-vector multiplication in the subspace

```
subspace1 = BitSubspace(5, [1, 3, 4])
subspace2 = BitSubspace(5, [2, 5])

# Julia uses 1-based index, we need to convert it
indices = collect(b + 1 for b in subspace2)

@inbounds for i in subspace1
    # add an offset i to all the indices of 0xx0x
    # this will give us 0xx0x, 0xx1x, 1xx0x, 1xx1x
    idx = indices .+ i
    state[idx] = U * state[idx] # matrix multiplication on the subspace
end
```

now we notice `subspace2` is the complement subspace of `subspace1` because the full space if `[1, 2, 3, 4, 5]`, so let's redefine our `BitSubspace` constructor a bit, now instead of define the constructor `BitSubspace(n, locations)` we define two functions to create this object `bsubspace(n, locations)` and `bcomspace(n, locations)` which stands for *binary subspace* and *binary complement space*, the function `bsubspace` will create `subspace1` and the function `bcomspace(n, locations)` will create `subspace2`. They have some overlapping operations, so we move them to an internal function `_group_shift` as shown in Listing 35.

Thus, we have the routine for general unitary gates as shown in Listing 36.

General Controlled Unitary Gates

We have introduced the routine for general unitary gates. A more specific type of general routine is the controlled unitary gates, which have a more specific pattern than the general unitary gates. This allows us to specialize the routine for better performance further. Implementing the controlled unitary gates is similar to the general unitary gates except that instead of iterating in the applied space, the subspace we will look at contains two parts: the bits on control locations are 1s, and the bits on gate locations are 0s.

Listing 35 Refined `group_shift` function

```
@inline function group_shift(locations)
    masks = Int[]
    shift_len = Int[]
    k_prv = -1
    for k in locations
        _group_shift(masks, shift_len, k, k_prv)
        k_prv = k
    end
    return masks, shift_len
end

@inline function complement_group_shift(n::Int, locations)
    masks = Int[]
    shift_len = Int[]
    k_prv = -1
    for k in 1:n
        k in locations && continue
        _group_shift(masks, shift_len, k, k_prv)
        k_prv = k
    end
    return masks, shift_len
end

@inline function _group_shift(
        masks::Vector{Int},
        shift_len::Vector{Int},
        k::Int,
        k_prv::Int
    )
    # if current position in the contiguous region
    # since these bits will be moved together with
    # the first one, we don't need to generate a
    # new mask
    if k == k_prv + 1
        shift_len[end] += 1
    else
        # we generate a bit mask where the 1st to k-th bits are 1
        push!(masks, bmask(0:k-1))
        push!(shift_len, 1)
    end
end
```

Listing 36 General unitary gate routine

```
function broutine!(
        st::AbstractVector,
        U::AbstractMatrix,
        locs::NTuple{N, Int}
    ) where N
    n = log2i(size(st, 1))
    subspace = bsubspace(n, locs)
    comspace = bcomspace(n, locs)
    indices = [idx + 1 for idx in comspace]
    @inbounds @views for k in subspace
        idx = indices .+ k
        st[idx] = U * st[idx]
    end
    return st
end
```

Loop Unroll and Parallelization

The above routine is already very efficient, but we can still improve it by unrolling the loop and parallelizing it because, in most simulations, the gate matrix is only a small 2×2 matrix. The loop unrolling can be implemented by a macro. However, for easier understanding, we will show the plain code here.

As a result, the benchmark of the two routines is shown in Listing 38. We can see that the unrolled routine is faster than the general routine.

Listing 37 Loop unrolling

```
function broutine2x2!(st::AbstractVector{T}, U::AbstractMatrix, locs::Tuple{Int}) where T
    U11 = U[1, 1]; U12 = U[1, 2];
    U21 = U[2, 1]; U22 = U[2, 2];
    step_1 = 1 << (first(locs) - 1)
    step_2 = 1 << first(locs)

    @inbounds if step_1 == 1
        for j in 0:step_2:size(st, 1)-step_1
            ST1 = U11 * st[j + 1] + U12 * st[j + 1 + step_1]
            ST2 = U21 * st[j + 1] + U22 * st[j + 1 + step_1]

            st[j + 1] = ST1
            st[j + 1 + step_1] = ST2
        end
    elseif step_1 == 2
        for j in 0:step_2:size(st, 1)-step_1
            Base.Cartesian.@nexprs 2 i->begin
                ST1 = U11 * st[j + i] + U12 * st[j + i + step_1]
                ST2 = U21 * st[j + i] + U22 * st[j + i + step_1]
                st[j + i] = ST1
                st[j + i + step_1] = ST2
            end
        end
        elseif step_1 == 4
        for j in 0:step_2:size(st, 1)-step_1
            Base.Cartesian.@nexprs 4 i->begin
                ST1 = U11 * st[j + i] + U12 * st[j + i + step_1]
                ST2 = U21 * st[j + i] + U22 * st[j + i + step_1]
                st[j + i] = ST1
                st[j + i + step_1] = ST2
            end
        end
    elseif step_1 == 8
        for j in 0:step_2:size(st, 1)-step_1
            Base.Cartesian.@nexprs 8 i->begin
                ST1 = U11 * st[j + i] + U12 * st[j + i + step_1]
                ST2 = U21 * st[j + i] + U22 * st[j + i + step_1]
                st[j + i] = ST1
                st[j + i + step_1] = ST2
            end
        end
    else
        for j in 0:step_2:size(st, 1)-step_1
            for i in j:8:j+step_1-1
                Base.Cartesian.@nexprs 8 k->begin
                    ST1 = U11 * st[i + k] + U12 * st[i + step_1 + k]
                    ST2 = U21 * st[i + k] + U22 * st[i + step_1 + k]
                    st[i + k] = ST1
                    st[i + step_1 + k] = ST2
                end
            end
        end
    end
    return st
end
```

Listing 38 Benchmark of `broutine!` and `broutine2x2!`

```
julia> U = rand(ComplexF64, 2, 2);

julia> locs = (3, );

julia> st = rand(ComplexF64, 1<<15);

julia> @benchmark broutine!(r, $U, $locs) setup=(r=copy($st))
BenchmarkTools.Trial:
  memory estimate:  512 bytes
  allocs estimate:  8
  --------------
  minimum time:     67.639 μs (0.00% GC)
  median time:      81.669 μs (0.00% GC)
  mean time:        86.487 μs (0.00% GC)
  maximum time:     125.038 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1

julia> @benchmark broutine2x2!(r, $U, $locs) setup=(r=copy($st))
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     21.420 μs (0.00% GC)
  median time:      21.670 μs (0.00% GC)
  mean time:        21.818 μs (0.00% GC)
  maximum time:     45.829 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

Specializing Other Gates

One can further specialize on the matrix entries of the gates. For example, the X and Z gates result in a more straightforward loop pattern and thus can be further specialized. The X gate only requires swapping the two elements in the subspace, and the Z gate only requires multiplying the elements in the subspace by -1. More generally, one can implement the routine of applying a column of specific gates.

Specializing Batch of Circuits

For simulating mid-circuit measurements, sampling the measurement result in the middle of circuit execution is often necessary. This requires simulating a batch of quantum circuits, which can be optimized by specializing the routine on matrix-matrix multiplication in the subspace.

Dispatch by Matching Patterns

The main technique behind the fast, exact simulation of quantum circuits in `Yao` is dispatching by matching circuit patterns. The idea is to encode the pattern of a circuit into Julia's type system. Thus, by utilizing the multiple dispatch mechanism, we can dispatch the simulation of a circuit using a specialized method. This is a very powerful technique, as it allows us to write the simulation of a circuit at a very high level and gradually improve the overall performance by manually specializing every different pattern. A common pattern in quantum circuits is repeatedly applying a single-qubit gate on all the qubits. This pattern can be seen as a special bit subspace and only requires applying a single routine. Thus, reduce the complexity of simulating n gates for n qubits to simulating a single gate for n qubits. Other patterns include the QFT circuits, the time evolution circuits, and the circuits with neighboring CNOTs.

Discussion

This subsection introduced the techniques behind implementing fast circuit simulation routines. These routines can also be used in the context of tensor network contraction because one can see the above matrix-vector multiplication as a contraction of a few legs in a giant tensor (the state) with legs of a small tensor (the gate). Such operations happen frequently in tensor network contraction, and thus, the routines can be helpful for fast tensor network contraction, as demonstrated in a recent work [62].

Listing 39 Heisenberg Hamiltonian

```
julia> using KrylovKit: eigsolve

julia> bond(n, i) = sum([put(n, i=>σ) * put(
    n, i+1=>σ) for σ in (X, Y, Z)]);

julia> heisenberg(n) = sum([bond(n, i)
            for i in 1:n-1]);

julia> h = heisenberg(16);

julia> w, v = eigsolve(mat(h)
              ,1, :SR, ishermitian=true)
```

3.1.3 Generating Matrix

Quantum blocks have a matrix representation of different types for optimized performance. By default, using their matrix representations, the `apply!` method applies quantum blocks to quantum registers. The matrix representation is also useful for determining operator properties such as hermicity, unitarity, reflexivity, and commutativity. Lastly, one can also use `Yao`'s sparse matrix representation for quantum many-body computations such as exact diagonalization and (real and imaginary) time evolution.

For example, one can construct the Heisenberg Hamiltonian and obtain its ground state using the Krylov space solver [113] via the `KrylovKit.jl` [114] in Listing 39. The arithmetic operations `*` and `sum` return `ChainBlock` and `Add` blocks, respectively. It is worth noticing the differences between the QBIR arithmetic operations of the quantum circuits and those of Hamiltonians. Since the Hamiltonians are generators of quantum unitaries (i.e., $U = e^{-iHt}$), it is natural to perform additions for Hamiltonians (and other observables) and multiplications for unitaries. `YaoExtensions` provides some convenience functions for creating Hamiltonians on various lattices and variational quantum circuits.

The `mat` function creates the sparse matrix representation of the Hamiltonian block. To achieve an optimized performance, we extend Julia's built-in sparse matrix types for various quantum gates. In Table 3.3, we summarize the matrix types used for the basic quantum gates.

Gate	Matrix Type
I2	`IMatrix`
Z, T, S, Rz	`Diagonal`
X, Y, CNOT, CZ, SWAP	`PermMatrix`
P0, P1, Pu, Pd, PSwap	`SparseMatrixCSC`
H, Rx, Ry	`Matrix`

Table 3.1: Matrix types of gates in `Yao`.

The `SparseMatrixCSC` type is provided in Julia's builtin `SparseArrays`. The identity matrix `IMatrix` and general permutation matrix `PermMatrix` [115] are defined in `LuxurySparse.jl` [116]. The `PermMatrix` allows having values other than one in the non-zero entries. For example, the matrix of ISWAP gate is given by

Listing 40 ISWAP matrix

```
julia> PermMatrix([1,3,2,4], [1,1.0im,1.0im,1])
4×4 PermMatrix{Complex{Float64},Int64,Array{Complex{Float64},1},Array{Int64,1}}:
 1.0+0.0im      0           0           0
    0           0        0.0+1.0im      0
    0        0.0+1.0im      0           0
    0           0           0        1.0+0.0im
```

where the first argument represents the column indices and the second argument the entries.

These types of specifications for quantum gates allow fast arithmetics. Table 3.2 lists the type conversion under matrix multiplication, Kronecker product, and addition operations.

Besides these specialised sparse matrices, `Yao.AD` uses low rank matrix types for back-propagation, c.f. Equation (3.10) in the main texts. For this we define the `OuterProduct` matrix type for both memory and computation efficiency.

Time evolution under a quantum Hamiltonian invokes the Krylov space method [117], which repeatedly applies the Hamiltonian block to the register. In this case, one can use the `cache` tag to create a `CachedBlock` for the Hamiltonian. Then, the `apply!` method makes

	I	D	P	S	M
I	I/I/D/I	D/D/D/D	P/P/S/D	S/S/S/D	M/S/M/D
D	D/D/D/D	D/D/D/D	P/P/S/D	S/S/S/D	M/S/M/D
P	P/P/S/D	P/P/S/D	P/P/S/P	S/S/S/P	M/S/M/P
S	S/S/S/D	S/S/S/D	S/S/S/P	S/S/S/S	M/S/M/S
M	M/S/M/D	M/S/M/D	M/S/M/P	M/S/M/S	M/S/M/M

Table 3.2: Matrix types conversion under matrix multiplication (*)/kronecker product (kron)/addition (+)/hadamard product (`.*`). Here I, D, P, S, M stands for `IMatrix`, `Diagonal`, `PermMatrix`, `SpasreMatrixCSC` and `Matrix` respectively.

use of the sparse matrix representation cached in the memory. Continuing from Listing 39, the following codes in c41 show that constructing and caching the matrix representation boosts the performance of time-evolution.

On the other hand, in many cases `Yao` can make use of efficient specifications of the `apply!` method for various blocks and apply them on the fly without generating the matrix representation. The codes in Listing 42 show that this approach can be faster for simulating quantum circuits.

3.1.4 Simulating Rydberg Dynamics

Rydberg Blockade is one of the most important properties of neutral-atom quantum computing based on Rydberg states. It naturally encodes the independent set constraint. More specifically, Rydberg blockade implies that two atoms cannot be both excited to the Rydberg state $|r\rangle$ if they are close to each other, whereas independent set constraint means two vertices cannot be both in the independent set when they are connected by an edge. Thus, one can consider atoms in the Rydberg state as vertices in an independent set. See the proposal in [106] for more details.

In particular, one can use the ground state of the Rydberg Hamiltonian to encode the maximum independent set problem, which is to find the largest independent set of a given graph. For a particular subclass of geometric graphs, the so-called unit disk graphs, the Rydberg Hamiltonian can encode the solution without any overhead in the number of qubits. In fact, an experimental demonstration of quantum optimization has been realized

Listing 41 Hamiltonian evolution is faster with cache

```
julia> using BenchmarkTools

julia> te = time_evolve(h, 0.1);

julia> te_cache = time_evolve(cache(h), 0.1);

julia> @btime $(rand_state(16)) |> $te;
  1.042 s (10415 allocations: 1.32 GiB)

julia> @btime $(rand_state(16)) |> $te_cache;
  71.908 ms (10445 allocations: 61.48 MiB)
```

Listing 42 Circuit simulation is faster without cache

```
julia> r = rand_state(10);

julia> @btime r |> $(qft(10));
  550.466 μs (3269 allocations: 184.58 KiB)

julia> @btime r |> $(cache(qft(10)));
  1.688 ms (234 allocations: 30.02 KiB)
```

in solving the maximum independent set problem up to 289 qubits in [118].

The simulation is implemented in the open-source package `Bloqade` [102]. `Bloqade` provides an extension of the operator expressions in QBIR in `Yao`. Taking from the symbolic expression of the Rydberg Hamiltonian and pulse sequence, `Bloqade` then compiles the expression into the sum of linear operators utilizing `Yao`'s specialized routines and matrices as well as CUDA [61] acceleration.

In `Bloqade`, we can also take advantage of this effect by allowing users to run emulation in a truncated subspace, i.e., by throwing out states that violate the blockade constraint. This can help accelerate the simulation and enable simulation for a larger system size. In this section, we will show how to create a blockade subspace, create registers in the subspace, obtain the Hamiltonian matrix in the subspace, and run emulation in the subspace. This enables a 51-atom simulation of the random-graph Rydberg atom array with the help of GPU.

3.2 Automatic Differentiation

AD is a method to evaluate the derivatives of a given program. One may hear this in the context of deep learning as *back-propagation*. However, the situation can be more complicated in a general scientific context. Thus, it'd be necessary to understand the mechanism of automatic differentiation better.

The history of automatic differentiation can be traced back to the 1960s when computer science was still in its infancy. The method has been re-discovered many times in different fields and has been used in many different contexts. The development of automatic differentiation leads to an entirely new philosophy of programming, which is called differentiable programming. In this section, we will briefly introduce the automatic differentiation of forward mode and reverse mode. Then, we will introduce automatic differentiation in the context of quantum circuits and simulating quantum many-body systems.

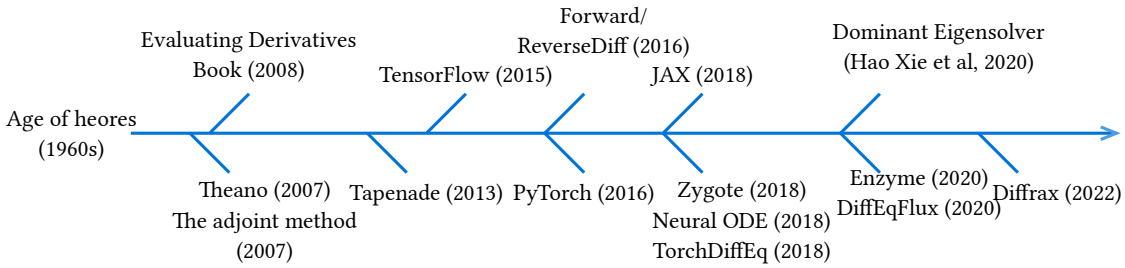

Figure 3.3: A brief history of automatic differentiation and its development in quantum many-body physics. Theano (2007) [3], Evaluating Derivatives Book (2008) [4], Tapenade (2013) [5, 6], TensorFlow (2015) [7], ForwardDiff/ReverseDiff [8], PyTorch (2016) [9], JAX (2018) [10], Zygote (2018) [11], differentiating dominant eigensolver (2020) [12], Enzyme (2020) [13], Diffrax (2022) [14]. Due to space limitations, many other libraries and algorithmic developments around 2015 and after are not included. The selected works represent the development of automatic differentiation relevant to the topic discussed in this thesis.

3.2.1 Forward Mode

The forward mode AD is usually implemented using dual numbers. A dual number is a pair of real numbers (x, x'), where x is the value of the function and x' is the value of the derivative. The dual number is defined as $x + x'\varepsilon$, where $\varepsilon^2 = 0$. The arithmetic operations of dual numbers are defined as follows:

$$\begin{aligned}(a + b\varepsilon) + (c + d\varepsilon) &= (a + c) + (b + d)\varepsilon,\\ (a + b\varepsilon) - (c + d\varepsilon) &= (a - c) + (b - d)\varepsilon,\\ (a + b\varepsilon) \cdot (c + d\varepsilon) &= ac + (ad + bc)\varepsilon,\\ \frac{a + b\varepsilon}{c + d\varepsilon} &= \frac{a}{c} + \frac{bc - ad}{c^2}\varepsilon.\end{aligned} \tag{3.3}$$

Thus one can implement the dual number as a Julia struct as follows:

Any function written as a composition of arithmetic operations can be automatically differentiated using the dual number. Similarly, one can overload other operations, such as sin, cos, exp, log, etc., to support the differentiation of these functions.

The forward mode AD is the simplest way of implementing an AD system. The above dual number is one way to implement forward mode AD with a generic type system. More

Listing 43 Dual number data structure in Julia

```
struct Dual{T<:Real} <: Real
    a::T
    b::T
end

Dual(a) = Dual(a, one(a))
Base.show(io::IO, x::Dual) = print(io, x.a, "+", x.b, "ϵ")
Base.:(*)(x::Dual, y::Dual) = Dual(x.a * y.a, x.a * y.b + x.b * y.a)
Base.:(+)(x::Dual, y::Dual) = Dual(x.a + y.a, x.b + y.b)
```

Listing 44 A simple real value function

```
function f(x)
    x1 = sin(x)
    x2 = sin(x1)
    x3 = cos(x2) + x2
end
```

formally, we say forward mode AD is to apply the chain rule on primal derivatives when we evaluate the value of an expression, e.g., if we are going to evaluate the function `f` defined in Section 3.2.1, that is mathematically defined as $f(x) = \cos(\sin(\sin(x))) + \sin(x)$, we can evaluate the function on dual number algebra to obtain its derivative.

When we put in a dual number, we actually perform the following calculations:

1. evaluate `sin(x)` and its derivative, store them in ‘x1‘.
2. evaluate `sin(x1)` and its derivative, multiply the derivative of ‘sin(x1)‘ to previous derivative since we have $P'(x)b\epsilon$ term in the dual number.
3. evaluate the derivative of `cos(x2)` and do the same thing as above, but let's call the intermediate value of `cos(x2)` to be `y`.
4. evaluate the value of `y+x2` and derivative then multiply and store them in `x3`

Thus, every time we evaluate the derivative, we can throw away the variables we calculated before, but for every single scalar number, we need to calculate the entire function.

As a result, this gives us the time complexity $O(mn)$ where n is the number of parameters and m is the complexity of the original code. The memory complexity is $O(m+n)$, where m is the memory complexity of the original code.

We further formalize the above process in terms of the chain rule. The forward mode means we evaluate the derivatives in the following accumulation:

$$\begin{aligned}\frac{\partial y_n}{\partial x} &= \frac{\partial y_n}{\partial y_{n-1}}\frac{\partial y_{n-1}}{\partial x} \\ &= \frac{\partial y_n}{\partial y_{n-1}}(\frac{\partial y_{n-1}}{\partial y_{n-2}}\frac{\partial y_{n-2}}{\partial x}) \\ &= \cdots \\ &= \frac{\partial y_n}{\partial y_{n-1}}(\frac{\partial y_{n-1}}{\partial y_{n-2}}\cdots(\frac{\partial y_2}{\partial y_1}\frac{\partial y_1}{\partial x}))\end{aligned} \tag{3.4}$$

We can further generalize this to multi-variable cases using Jacobians

$$\begin{aligned}\frac{\partial \vec{y}_n}{\partial x} &= \frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}}\frac{\partial \vec{y}_{n-1}}{\partial x} \\ &= \frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}}(\frac{\partial \vec{y}_{n-1}}{\partial \vec{y}_{n-2}}\frac{\partial \vec{y}_{n-2}}{\partial x}) \\ &= \cdots \\ &= \frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}}(\frac{\partial \vec{y}_{n-1}}{\partial \vec{y}_{n-2}}\cdots(\frac{\partial \vec{y}_2}{\partial \vec{y}_1}\frac{\partial \vec{y}_1}{\partial x}))\end{aligned} \tag{3.5}$$

now we see to calculate the derivative of $\frac{\partial \vec{y}_n}{\partial x}$ it is actually about calculatnig a chain of **Jacobian-vector product**.

3.2.2 Reverse Mode

For a large number of parameters, the reverse mode is more efficient. This method has been rediscovered many times in history in different fields. One may hear it called back-propagation in the context of deep learning. The reverse mode AD can be implemented via operator overloading as well, which has been the main approach of many well-known software packages such as `autograd`, `AutoDiff`.

Similar to forward mode AD, the reverse mode AD is actually about calculating the chain rule in the following accumulation:

$$
\begin{aligned}
\frac{\partial \vec{y}_n}{\partial x} &= (\frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}})\frac{\partial \vec{y}_{n-1}}{\partial x} \\
&= ((\frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}})\frac{\partial \vec{y}_{n-1}}{\partial \vec{y}_{n-2}})\frac{\partial \vec{y}_{n-2}}{\partial x} \\
&= \cdots \\
&= (\cdots((\frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}})\frac{\partial \vec{y}_{n-1}}{\partial \vec{y}_{n-2}}\cdots\frac{\partial \vec{y}_2}{\partial \vec{y}_1})\frac{\partial \vec{y}_1}{\partial x} \\
(\frac{\partial \vec{y}_n}{\partial x})^T &= (\frac{\partial \vec{y}_1}{\partial x})^T(\frac{\partial \vec{y}_2}{\partial \vec{y}_1})^T\cdots(\frac{\partial \vec{y}_{n-1}}{\partial \vec{y}_{n-2}})^T(\frac{\partial \vec{y}_n}{\partial \vec{y}_{n-1}})^T
\end{aligned}
\tag{3.6}
$$

Thus, the reverse mode automatic differentiation is about **Jacobian-transpose-vector product**. An intermediate advantage of this accumulation is that if `x` is a vector, we can directly calculate the derivative of this vector by a chain of matrix multiplication.

This process will be easier to understand if we use a graphical language to describe it. This is the **computational graph**. To demonstrate this better, we will use a more complicated function

$$
y = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}\cdot\mathbf{x} + c \tag{3.7}
$$

We will need to call several functions in Julia to get the result y, which is

1. $\mathbf{z_1} = \mathbf{x}^T$: `transpose` function.
2. $\mathbf{z_2} = \mathbf{z_1}A$ matrix-vector multiplication, which can be `gemv` in `LinearAlgebra.BLAS`, or just `*`.
3. $y_1 = \mathbf{z_2}\mathbf{x}$ vector dot operation, which is `LinearAlgebra.dot`
4. $y_2 = \mathbf{b}\cdot\mathbf{x}$ another vector dot
5. $y_1 + y_2 + c$ a scalar add function, one can calculate it by simply calling `+` operator in Julia.

In fact, we can draw a graph of this expression, which illustrates the relationship between each variable in this expression. Each node in the graph with an output arrow represents a variable and each node with an input arrow represents a function or an operator.

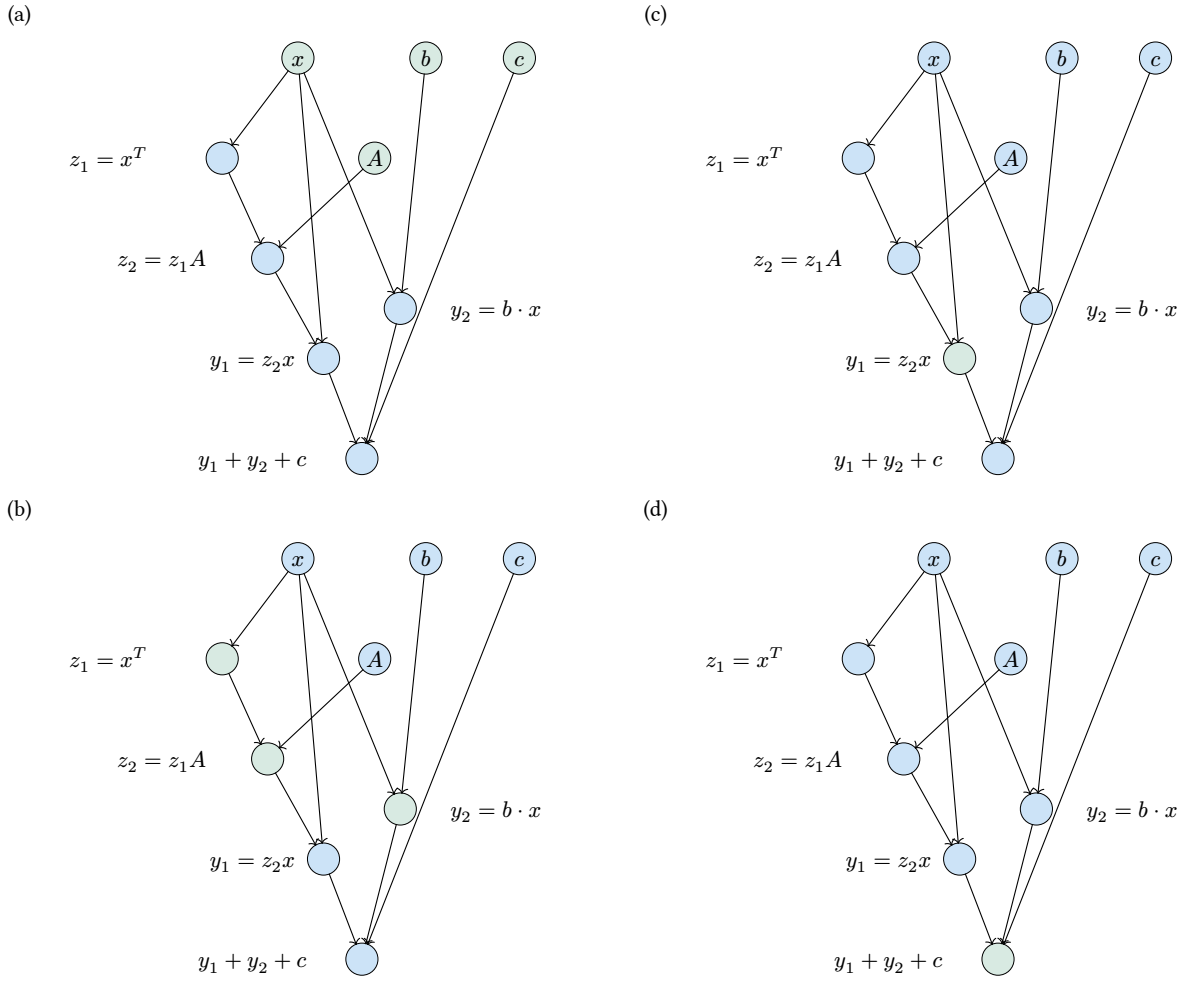

Figure 3.4: The forward process on computational graph of the expression $y = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$

The evaluation of the math equation in Figure 3.4 can then be expressed as a process called forward evaluation. It starts from the leaf nodes, which represent the inputs of the whole expression, e.g., they are $\mathbf{x}, \mathbf{A}, \mathbf{b}, c$ in our expression. Each time we receive the value of a node in the graph, we mark the node with green. The derivative calculation can be then visualized as follows

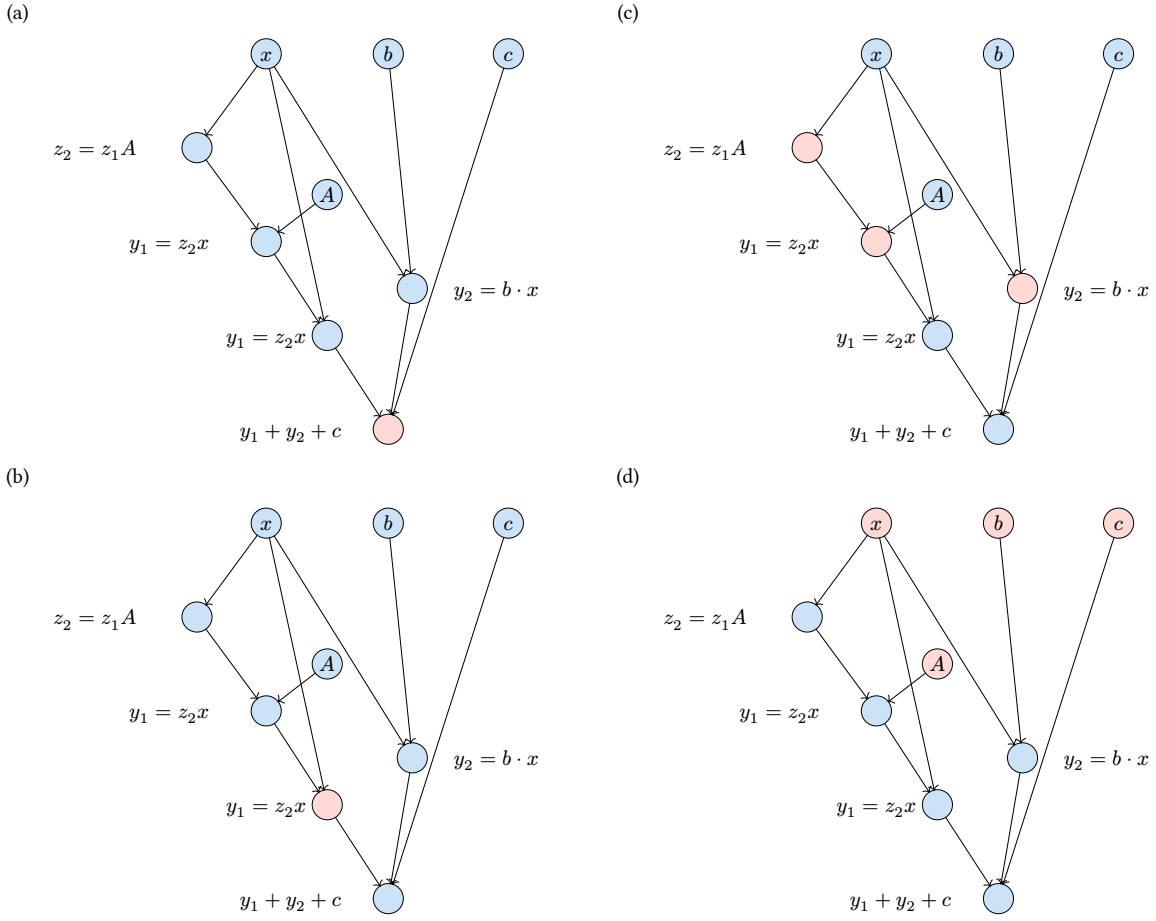

Figure 3.5: The backward process on computational graph of the expression $y = \mathbf{x}^T \mathbf{A}\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$

In Figure 3.5, we demonstrate the backward propagation of the derivatives. Unlike the forward mode, the intermediate results are not kept. The reverse mode must store the intermediate values in its forward propagation, which is necessary to calculate the derivatives in backward propagation. The benefit is that the reverse mode can calculate all the derivatives of the vector without running the forward propagation for every parameter. Thus, we need a specific data structure to store these values. This data structure is often called the computational graph, tape, or Wengert list [110]. The problem of implementing reverse mode AD becomes how to create the tape. Because we need to store the values at

each calculation step, this is precisely a special case of the SSA IR in Section 2.5.

We can do reverse mode 'automatic' differentiation 'manually' first to get a feel for it. We will use the package `ChainRules` here, which contains a function `rrule` that defines the differentiation rules for the primitive function we need to use, such as the `sin` function.

Listing 45 Reverse mode ADfor a single function

```
x = 3.0
y, sin_pullback = rrule(sin, x)
julia> y, sin_pullback = rrule(sin, x)
(0.1411200080598672, ChainRules.var"#sin_pullback#1289"{Float64}(-0.9899924966004454))

julia> sin_pullback(1)
(ChainRulesCore.NoTangent(), -0.9899924966004454)
```

We can manually generate the pullback function for a given function defined on some primal functions. In AD, we usually call these functions **adjoint**s.

Let's use the previous example 'foo' here

```
function adjoint_foo(x)
    x1, x1_pullback = rrule(sin, x)
    x2, x2_pullback = rrule(sin, x1)
    y, y_pullback = rrule(cos, x2)
    x3, x3_pullback = rrule(+, y, x2)

    return x3, function pullback(Δ)
        _, partial_y, partial_x2_1 = x3_pullback(Δ)
        _, partial_x2_2 = y_pullback(partial_y)
        _, partial_x1 = x2_pullback(partial_x2_1 + partial_x2_2)
        _, partial_x = x1_pullback(partial_x1)
        return Zero(), partial_x
    end
end
```

Thus the simplest reverse mode ADis about how to create this tiny `pullback` function automatically, and to know how to create this `pullback` function, we need to know what primal functions are called by the given function, so that we can simply reverse the order of calls and replace them with the `pullback` functions. The easiest way of creating a tape

is using operator overloading. Operator overloading means the same way we implemented our toy symbolic program in the previous section. For example, we can define our own type to dispatch the functions to a `track` function so that we can store the function call into a tape.

Listing 46 A simple ADengine in Julia

```
mutable struct Variable{T} <: ADExpr
    value::T
    grad::T

    Variable(val::T) where T = new{T}(val)
    Variable(val::T, grad::T) where T = new{T}(val)
end

struct Node{FT <: Function, ArgsT <: Tuple, KwargsT <: NamedTuple} <: ADExpr
    f::FT
    args::ArgsT
    kwargs::KwargsT
end
```

then overload some primal functions to track the call into tape

Listing 47 Overloading `sin` function

```
Base.sin(x::ADExpr) = register(Base.sin, x)
```

We will not proceed to implement the full ADengine here. Going forward, One should expect implementing such an ADengine to be quite simple. However, the real challenge is to make it efficient and to make it work with a wide range of programs. This is why research interests have been put into source-to-source AD, which synthesizes the above `pullback` function via a transformation between the same representations.

3.2.3 Making Use of Reversibility

Automatic differentiation efficiently computes the gradient of a program. It is the engine behind the success of deep learning [119]. The technique is particularly relevant to differentiable programming of quantum circuits. In general, there are several modes of AD. The

reverse mode caches the intermediate state and evaluates all gradients in a single backward run. The forward mode computes the gradients in a single pass together with the objective function, which does not require caching the intermediate state but has to evaluate the gradients of all parameters one by one.

`Yao`'s builtin reverse mode AD engine (Section 3.2.2) provides more efficient circuit differentiation for variational quantum algorithms compared to conventional reverse mode differentiation and forward mode differentiation (Section 3.2.1). By taking advantage of the reversible nature of quantum circuits, the memory complexity is reduced to constant compared to typical reverse mode AD [119]. This property allows one to simulate very deep variational quantum circuits. Besides, `Yao` supports the forward mode AD (Section 3.2.4), which is a faithful quantum simulation of the experimental situation. In the classical simulation, the complexity of forward mode is unfavorable compared to reverse mode because one needs to run the circuit repeatedly for each component of the gradient.

The submodule `Yao.AD` is a built-in AD engine. It back-propagates through quantum circuits using the computational graph information recorded in the QBIR.

In general, reverse mode AD needs to cache intermediate states in the forward pass for the backpropagation. Therefore, the memory consumption for backpropagating through a quantum simulator becomes unacceptable as the depth of the quantum circuit increases. Hence simply delegating AD to existing machine learning packages [7, 9, 80–83] is not a satisfiable solution. `Yao`'s customized AD engine exploits the inherent reversibility of quantum circuits [4, 120]. By uncomputing the intermediate state in the backward pass, `Yao.AD` mostly performs in-place operations without allocations. `Yao.AD`'s superior performance is in line with the recent efforts of implementing efficient backpropagation through reversible neural networks [120, 121].

In the forward pass, we update the wave function $|\psi_k\rangle$ with in-place operations

$$\begin{aligned} &\cdots \\ |\psi_{k+1}\rangle &= U_k|\psi_k\rangle, \\ &\cdots \end{aligned} \tag{3.8}$$

where U_k is a unitary gate parametrized by θ_k. We define the adjoint of a variable as $\overline{x} = \frac{\partial \mathcal{L}}{\partial x^*}$ according to Wirtingers derivative [122] for complex numbers, where $\mathcal{L}$ is a real-valued objective function that depends on the final state. Starting from $\overline{\mathcal{L}} = 1$ we can obtain the adjoint of the output state.

To pull back the adjoints through the computational graph, we perform the backward

calculation [123]

$$\begin{aligned} &\cdots \\ |\psi_k\rangle &= U_k^\dagger |\psi_{k+1}\rangle \\ \overline{|\psi_k\rangle} &= U_k^\dagger \overline{|\psi_{k+1}\rangle} \\ &\cdots \end{aligned} \tag{3.9}$$

The two equations above are implemented `Yao.AD` with the `apply_back!` method. Based on the obtained information, we can compute the adjoint of the gate matrix using [123]

$$\overline{U_k} = \overline{|\psi_{k+1}\rangle}\langle\psi_k|. \tag{3.10}$$

This outer product is not explicitly stored as a dense matrix. Instead, it is handled efficiently by customized low rank matrices described in Section 3.1.3. Finally, we use `mat_back!` method to compute the adjoint of gate parameters $\overline{\theta_k}$ from the adjoint of the unitary matrix $\overline{U_k}$.

Figure 3.6 demonstrates the procedure in a concrete example. The black arrows show the forward pass without any allocation except for the output state and the objective function $\mathcal{L}$. In the backward pass, we uncompute the states (blue arrows) and backpropagate the adjoints (red arrows) at the same time. For the block defined as `put(nbit, i=>chain(Rz(`α`), Rx(`β`), Rx(`γ`)))`, we obtain the desired $\overline{\alpha}$, $\overline{\beta}$ and $\overline{\gamma}$ by pushing the adjoints back through the `mat` functions of `PutBlock` and `ChainBlock`. The implementation of the AD engine is generic so that it works automatically with symbolic computation. One can also integrate `Yao.AD` with classical automatic differentiation engines such as `Zygote` to handle mixed classical and quantum computational graphs, see [100].

To demonstrate the efficiency of `Yao`'s AD engine, we use the codes in Listing 48 to simulate the variational quantum eigensolver (VQE) [37] with depth $10,000$ (with $300,010$ variational parameters) on a laptop. The simulation would be extremely challenging without `Yao`, either due to overwhelming memory consumption in the reverse mode AD or unfavorable computation cost in the forward mode AD

Here, `variational_circuit` is predefined in `YaoExtensions` to have a hardware efficient architecture [72] shown in Figure 3.8. The `dispatch!` function with the second parameter specified to `:random` gives random initial parameters. The `expect` function evaluates expectation values of the observables; the second argument can be a wave function or a pair of the input wave function and circuit ansatz like above. `expect'` evaluates the gradient of this observable for the input wave function and circuit parameters. Here,

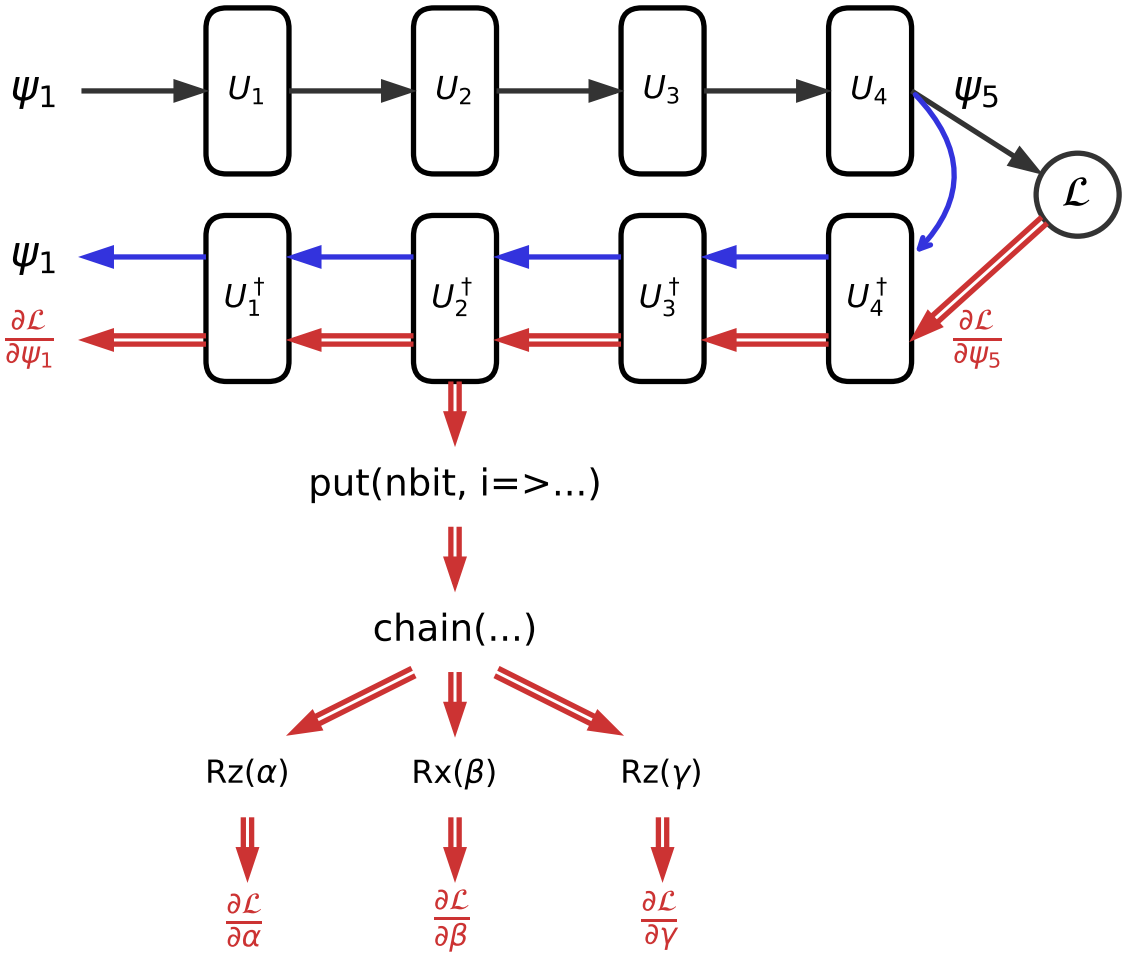

Figure 3.6: Builtin automatic differentiation engine **Yao.AD**. Black arrows represent the forward pass. The blue arrow represents uncomputing. The red arrows indicate the back-propagation of the adjoints.

Listing 48 10000-layer VQE

```
using Yao, YaoExtensions
n = 10; depth = 10000;
circuit = dispatch!(variational_circuit(n, depth),:random)

julia> gatecount(circuit)
Dict{Type{var"#s54"} where var"#s54" <: AbstractBlock,Int64} with 3 entries:
  RotationGate{1,Float64,ZGate} => 200000
  RotationGate{1,Float64,XGate} => 100010
  ControlBlock{10,XGate,1,1}    => 100000

julia> nparameters(circuit)
300010

h = heisenberg(n);

for i = 1:100
    _, grad = expect'(h, zero_state(n)=>circuit)
    dispatch!(-, circuit, 1e-3 * grad)
    println("Step $i, energy = $(expect(h, zero_state(10)=>circuit))")
end
```

we only make use of its second return value. For batched registers, the gradients of circuit parameters are accumulated rather than returning a batch of gradients. `dispatch!(-, circuit, ...)` implements the gradient descent algorithm with energy as the loss function. The first argument is a binary operator that computes a new parameter based on the old parameter in `c` and the third argument, the gradients. Parameters in a circuit can be extracted by calling `parameters(circuit)`, which collects parameters into a vector by visiting the QBIR in depth-first order. The same parameter visiting order is used in `dispatch!`. In case one would like to share parameters in the variational circuit, one can simply use the same block instance in the QBIR. In the training process, gradients can be updated in the same field. After the training, the circuit is fully optimized and returns the ground state of the model Hamiltonian with zero state as input.

3.2.4 Forward Mode: Faithful Quantum Gradients

Compared to the reverse mode, forward mode AD is more closely related to how one measures the gradient in the actual experiment.

The implementation of the forward mode AD is particularly simple for the "rotation gates" $R_\Sigma(\theta) \equiv e^{-i\Sigma\theta/2}$ with the generator Σ being both hermitian and reflexive ($\Sigma^2 = 1$). For example, Σ can be the Pauli gates X, Y and Z, or multi-qubit gates such as CNOT, CZ, and SWAP. Every two-qubit gate can be decomposed into Pauli rotations and CNOTs (or CZs) via gate transformation [124]. Under these conditions, the gradient to a circuit parameter is [68, 125–127]

$$\frac{\partial \langle O \rangle_\theta}{\partial \theta} = \frac{1}{2}\left(\langle O \rangle_{\theta+\frac{\pi}{2}} - \langle O \rangle_{\theta-\frac{\pi}{2}}\right) \tag{3.11}$$

where $\langle O \rangle_\theta$ denotes the expectation of the observable O with the given parameter θ. Therefore, one just needs to run the simulator twice to estimate the gradient. `YaoExtensions` implements Equation (3.11) with Julia's broadcasting semantics and obtains the full gradients with respect to all parameters. Similar features can be found in `PennyLane` [90] and `qulacs` [89]. We refer this approach as the faithful gradient, since it mirrors the experimental procedure on a real quantum device. In this way, one can estimate the gradients in the VQE example Listing 48 using Equation (3.11)

```
# this will be slow
julia> grad = faithful_grad(h, zero_state(n)=>circuit; nshots=100);
```

Listing 49 The eigendecomposition of a QBIR.

```
julia> O = chain(5, put(5,2=>X), put(5,3=>Y))
nqubits: 5
chain
├─ put on (2)
│  └─ X
└─ put on (3)
   └─ Y

julia> E, U = YaoBlocks.eigenbasis(O)
(nqubits: 5
chain
├─ put on (2)
│  └─ Z
└─ put on (3)
   └─ Z
, nqubits: 5
chain
├─ put on (2)
│  └─ H
└─ put on (3)
   └─ chain
      ├─ H
      └─ S
)
```

where one faithfully simulates `nshots` projective measurements. In the default setting `nshots=nothing`, the function evaluates the exact expectation on the quantum state. Note that simulating projective measurement, in general, involves rotating to eigenbasis of the observed operator. `Yao` implements an efficient way to break the measurement into the expectation of local terms by diagonalizing the observed operator symbolically as bellow.

The return value of `eigenbasis` contains two QBIRs `E` and `U` such that `O = U*E*U'`. `E` is a diagonal operator that represents the observable in the measurement basis. `U` is a circuit that rotates computational basis to the measurement basis.

The above gradient estimator Equation (3.11) can also be generalized to statistic functional loss, which is useful for generative modeling with an implicit probability distribution

Listing 50 Gradient of the maximum mean discrepancy

```
julia> target_p = normalize!(rand(1<<5));

julia> kf = brbf_kernel(2.0);

julia> circuit = variational_circuit(5);

julia> mmd = MMD(kf, target_p);

julia> g_reg, g_params = expect'(
    mmd, zero_state(5)=>circuit);

julia> g_params = faithful_grad(
    mmd, zero_state(5)=>circuit);
```

given by the quantum circuits [70]. The symmetric statistic functional of order two reads

$$\mathcal{F}_\theta = \langle K(x,y)\rangle\, x \sim p_\theta, y \sim p_\theta, \tag{3.12}$$

where K is a symmetric function, p_θ is the output probability distribution of a parametrized quantum circuit measured on the computational basis. If the circuit is parametrized by rotation gates, the gradient of the statistic functional is

$$\begin{aligned}\frac{\partial \mathcal{F}_\theta}{\partial \theta} = \quad & \langle K(x,y)\rangle\, x \sim p_{\theta+\frac{\pi}{2}}, y \sim p_\theta \\ & - \langle K(x,y)\rangle\, x \sim p_{\theta-\frac{\pi}{2}}, y \sim p_\theta,\end{aligned} \tag{3.13}$$

which is also related to the measure valued gradient estimator for stochastic optimization [128]. Within this formalism, `Yao` provides the following interfaces to evaluate gradients with respect to the maximum mean discrepancy loss [129, 130], which measures the probabilistic distance between two sets of samples.

3.3 Benchmark

3.3.1 Benchmark: Exact Circuit Simulation

As introduced above, `Yao` features a generic and extensible implementation without sacrificing performance. Our performance optimization strategy heavily relies on Julia's multiple

dispatch. As a bottom line, `Yao` implements a general multi-control multi-qubit arbitrary-location gate instruction as the fallback. We then fine-tune various specifications for better performance. Therefore, in many applications, the construction and operation of QBIR do not even invoke matrix allocation. While in cases where the gate matrix is small (number of qubits smaller than 4), `Yao` automatically employs the corresponding static sized types [131] for better performance. The sparse matrices `IMatrix`, `Diagonal`, `PermMatrix` and `SparseMatrixCSC` introduced in Section 3.1.3 also have their static version defined in `LuxurySparse.jl` [116]. Besides, we also utilize unique structures of frequently used gates and dispatch to specialized implementations. For example, Pauli X gate can be executed by swapping the elements in the register directly.

We benchmark `Yao`'s performance with other quantum computing software. Note that the exact classical simulation of the generic quantum circuit is doomed to be exponential [132–135]. `Yao`'s design puts a strong emphasis on the performance of small to intermediate-sized quantum circuits since the high-performance simulation of such circuits is crucial for the design of near-term algorithms that run repeatedly or in parallel.

Benchmark Setup

Package	Language	Version
`Cirq` [88]	Python	0.8.0
`qiskit` [91]	C++/Python	0.19.2
`qulacs` [89]	C++/Python	0.1.9
`PennyLane` [90]	Python	0.7.0
`QuEST` [92]	C/Python	3.0.0
`ProjectQ` [86]	C++/Python	0.4.2
`Yao`	Julia	0.6.2
`CuYao`	Julia	0.2.2

Table 3.3: Packages in the benchmark.

Although `QuEST` is a package originally written in C, we benchmark it in Python via `pyquest-cffi` [136] for convenience. `Pennylane` is benchmarked with its default backend [137]. Since the package was designed primarily for being run on the quantum hardware, its benchmarks contain a certain overhead that was not present in other frame-

works [138]. `qiskit` is benchmarked with `qiskit-aer` 0.5.1 [139] and `qiskit-terra` 0.14.1 [140] using the statevector method of the qasm simulator.

Software	**Version**
Python	3.8.3
Numpy	1.18.1
MKL	2019.3
Julia	1.5.2

Table 3.4: The environment setup of the machine for benchmark.

Our test machine contains an Intel(R) Xeon(R) Gold 6230 CPU with a Tesla V100 GPU accelerator. SIMD is enabled with **AVX2** instruction set. The benchmark time is measured via `pytest-benchmark` [141] and `BenchmarkTools` [142] with minimum running time. We ignore the compilation time in Julia since one can always get rid of such time by compiling the program ahead of time. The benchmark scripts and complete reports are maintained online at the repository [143]. For more detailed and latest benchmark configuration one should always refer to this repository.

Single Gate Performance

We benchmark several frequently used quantum gates, including the Pauli-X Gate, the Hadamard gate (H), the controlled-NOT gate (CNOT), and the Toffoli Gate. These benchmarks measure the performance of executing one single gate instruction.

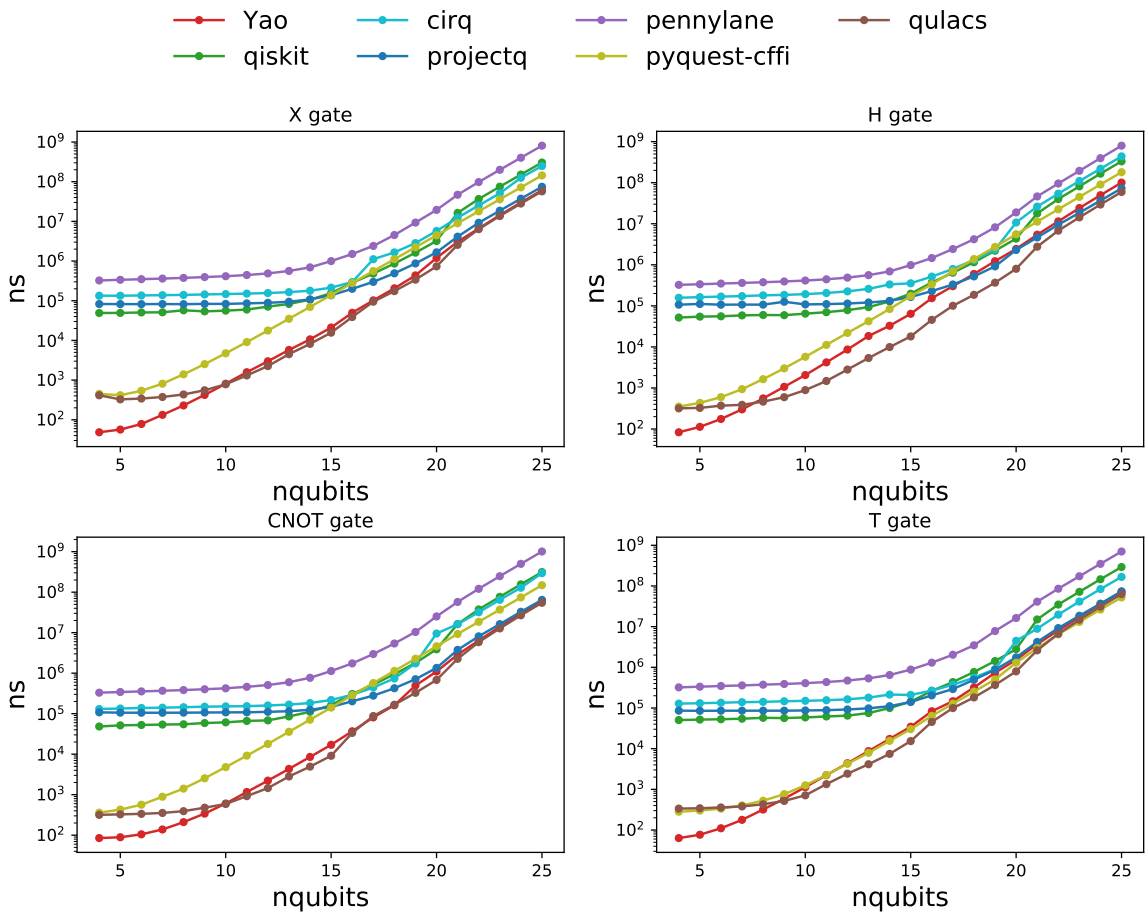

Figure 3.7: Benchmarks of (a) Pauli-X gate; (b) Hadamard gate; (c) CNOT gate; (d) Toffolli gate.

Figure 3.7 shows the running times of various gates applied on the second qubit of the register from size 4 to 25 qubits in each package in the unit of nano seconds. One can see that `Yao`, `ProjectQ`, and `qulacs` reach similar performance when the number of qubits $n > 20$. They are at least several times faster than other packages. Having similar performance in these three packages suggests that they all reached the top performance for this type of full amplitude classical simulation on CPU.

Parametrized Quantum Circuit Performance

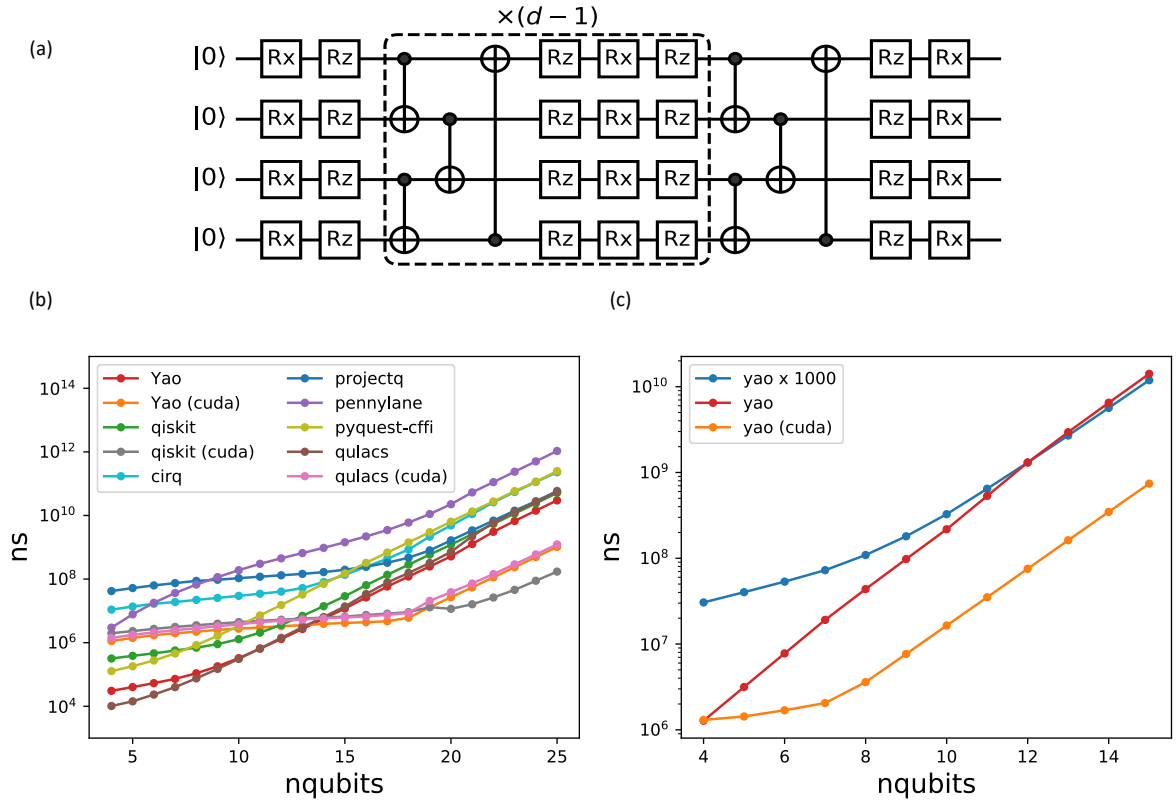

Figure 3.8: (a) A parameterized quantum circuit with single qubit rotation and CNOT gates; (b) Benchmarks of the parameterized circuit; (c) Benchmarks of the parametrized circuit, the batched version. Line "yao" represents the batched registers, "yao (cuda)" represents the batched register on GPU, "yao × 1000" is running on a non-batched register repeatedly for 1000 times.

Next, we benchmark the parameterized circuit of depth $d = 10$ shown in Figure 3.8(a). This type of hardware-efficient circuits was employed in the VQE experiment [72]. These benchmarks further test the performance of circuit abstraction in practical applications.

The results in Figure 3.8(b) shows that `Yao` reaches the best performance for more than 10 qubits on CPU. `qulacs`'s well tuned C++ simulator is faster than `Yao` for fewer qubits. On a CUDA device, `Yao` and `qulacs` show similar performance. `qiskit` cuda backend

shows better performance for more that 20 qubits. These benchmarks also, show that CUDA parallelization starts to be beneficial for a qubit number larger than 16. Overall, `Yao` is one of the fastest quantum circuit simulators for this type of application.

Lastly, we benchmark the performance of batched quantum register introduced in Sec 2.2 in Figure 3.8(c) with a batch size 1000. We only measure `Yao`'s performance due to the lack of native support of SPMD in other quantum simulation frameworks. `Yao`'s CUDA backend (labeled as `yao (cuda)`) offers large speed up (>10x) compared to the CPU backend (labeled as `yao`). For reference, we also plot the timing of a bare loop over the batch dimension on a CPU (labeled as `yao` $\times$ `1000`). One can see that batching offers substantial speedup for small circuits.

The overhead of simulating small to intermediate-sized circuits is particularly relevant for designing variational quantum algorithms where the same circuit may be executed million times during training. `Yao` shows the least overhead in these benchmarks. `qulacs` also did an excellent job of suppressing these overheads.

Matrix Representation and Automatic Differentiation Performance

As discussed in Section 3.1.3 and Section 3.2.3, `Yao` features highly optimized matrix representation and reverse mode automatic differentiation for the QBIR. We did not attempt a systematic benchmark due to the lack of similar features in other quantum software frameworks.

Here, we simply show the timings of constructing the sparse matrix representation of 20 site Heisenberg Hamiltonian and differentiating its energy expectation through a variational quantum circuit of depth 20 (200 parameters) on a laptop. The forward mode AD discussed in Section 3.2.4 is slower by order of a hundred in such simulations.

Listing 51 Benchmark mat and AD performance

```
julia> using BenchmarkTools, Yao,
                    YaoExtensions

julia> @btime mat($(heisenberg(20)));
  6.330 s (10806 allocations: 10.34 GiB)

julia> @btime expect'($(heisenberg(20)),
                      $(zero_state(20))=>
              $(variational_circuit(20)));
  5.054 s (58273 allocations: 4.97 GiB)
```

3.3.2 Benchmark: Exact Rydberg Atom Dynamics Simulation

We benchmark the exact rydberg atom dynamics simulation on a $1D$ chain comparing to the `qutip` [144] in Figure 3.9. We see speedups at 10x in small systems and similar performance at larger system size on CPU. In the CUDA backend, our implementation is slower than CPU before 12 atoms. At 20 atoms, the CUDA provides about 80x speedup. With the blockade subspace approximation, we can see a speedup of 1000x at 20 atoms.

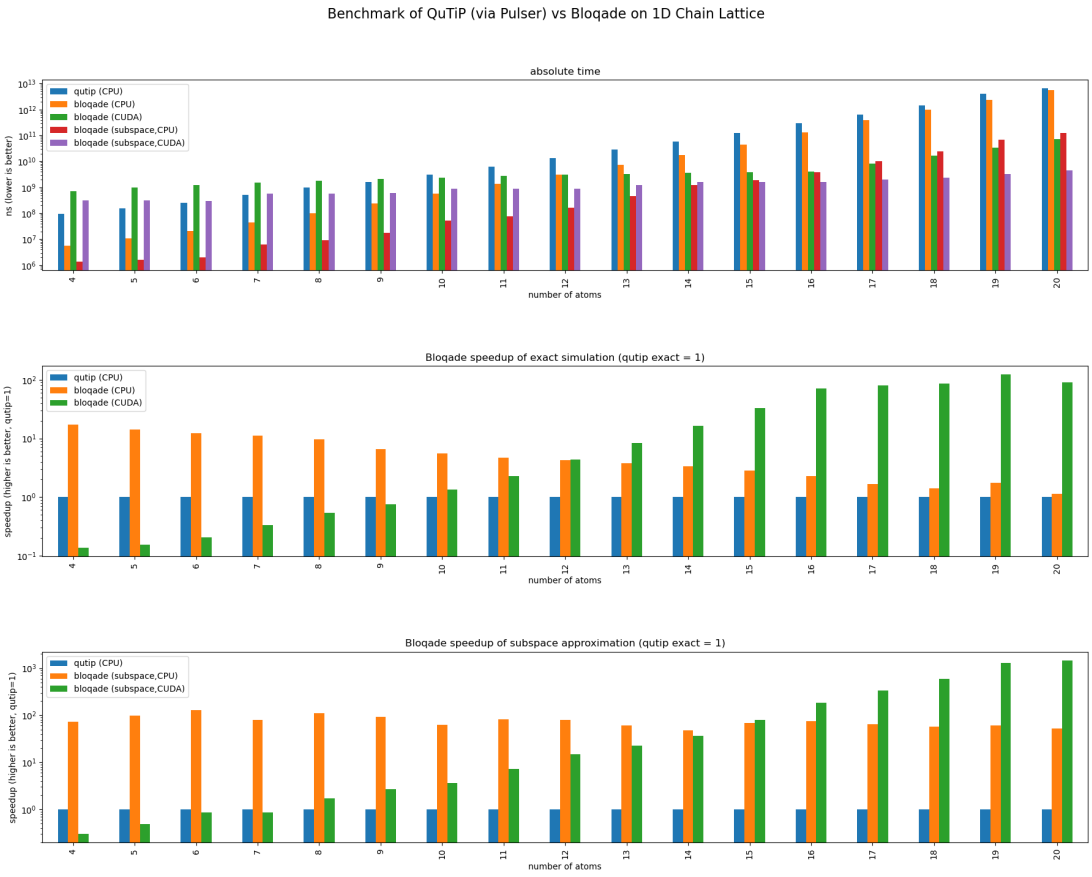

Figure 3.9: Benchmark of the exact Rydberg atom dynamics simulation.

Besides the utilization of hardware acceleration through `CUDA`, comparing to existing implementations, `Bloqade` utilizes the highly optimized matrix representation from `Yao` and the highly optimized Ordinary Differential Equation (ODE) solver from `DiffEq`. Algorithm advancements such as adaptive stepping and the use of subspace approximation also contribute to the performance improvement. On the other hand, at smaller size, our CPU backend sees 10-100x speedup comparing to `qutip`. This is due to the adaptive stepping which reduces the number of steps required during the integration. However, as the exact simulation grows eventually, the performance for the CPU backend converges to the same level as `qutip` at 20 atoms.

3.4 Discussion

In this chapter, we have introduced the transformation from an expression into the subspace matrix-vector multiplication routine and special matrices, which actually execute the simulation. Moreover, we introduced the techniques for implementing these routines and special matrices in the context of exact simulation. On top of these building blocks, we introduced the transformation for automatic differentiation, which is a powerful tool for optimizing the simulation and understanding the physical system. We also benchmarked the performance of the exact simulation and the automatic differentiation and showed that the exact simulation is already approaching the limit of the classical computer. Moreover, we showed that by utilizing the reversibility of the quantum circuits, we can achieve significant speedup in the automatic differentiation that no previous software can achieve. This section discusses the future directions of quantum circuits and Hamiltonian dynamics simulation.

The convergence in our benchmark Section 3.3.1 shows that we are already approaching the limit of the exact simulation on a classical computer. Both benchmarks in `Yao` and `Bloqade` are converging with more straightforward simulation strategies used in other simulations at larger sizes because the simulation engine is designed for general purposes. Thus, there is a lack of understanding of a specific problem. On the other hand, we do not want to lose the ability to simulate a variety of problems. Thus, an important future direction is to find expression transformations that can automatically specialize the simulation on problems with specific properties.

For example, the simulation of quantum circuits can utilize the circuit's structure, thus leading to the exact contraction of a tensor network. While `Yao` provides the functionality of compiling circuit expression into tensor network contraction tree, we may not have the optimal strategy for finding the contraction orders. Previous work has shown a promising routine in utilizing a symbolic rewrite system for finding the optimal contraction order [145]. With the recent advancements in equality saturation [146], we can expect to see more powerful expression transformation techniques in the future.

Similar to the exact simulation of Rydberg Hamiltonian dynamics or other Hamiltonian dynamics. In the worst case, one will need a quantum computer to simulate such dynamics unless P = BQP [31]. We can still expect to see a significant speedup in finite-size simulation by utilizing the structure of the Hamiltonian. For example, the Hamiltonian of Rydberg atoms can be also expressed in a diagonalized Hamiltonian at the short time as

$$\exp[-i\delta H] = \exp\left[-i\delta \sum_{\langle i,j\rangle} V_{ij} n_i n_j\right] \exp\left[-i\delta \sum_i \Omega_i H Z H\right] \exp\left[-i\delta \sum_i \Delta_i n_i\right] \tag{3.14}$$

where the dynamics of Rydberg Hamiltonian at a short time δ can be decomposed into a few diagonal matrices with the similarity transform being a Hadamard matrix. Applying such a unitary approach to a given state can thus be further specialized. This also leads to the utilization of the symplectic structure of the Hamiltonian, which conserves the norm of the state and thus results in better numerical stability when simulating very oscillating dynamics.

Furthermore, one may expect the specialization mentioned above to happen for a large class of real problems. Philosophically, this occurs because the physical entities in reality often contain structures that are not arbitrary. In our latest software framework `Liang`, we aim to answer the question of automatic specialization by thinking about the transformation of more general operator expressions. We see opportunities to utilize the structure of the physical system to specialize in simulation. While the effort of pushing the boundary of exact simulation will have theoretical limitations, the techniques developed for these optimizations will also help compile simulation tasks for quantum computers. Similar to compiling the expression describing the problem into simulation routines, we can also compile the expression into routines of a quantum device. Thus, developing such classical simulations will also lead to the progress of quantum computing.

Chapter 4

Generalization: Operator Learning Renormalization Group

In the previous chapters, we have introduced programmatic representation as a powerful technique to work with computational software and experimental quantum computing hardware. In this section, we will discuss how to rethink the representation and the algorithm using the concepts instead of the techniques we learned from programmatic representations and how this leads to discovering an alternative variational principle. While techniques we introduced in previous chapters will be used, such as automatic differentiation and representations of analog Hamiltonian, unlike previous chapters, this chapter will present in a more traditional physics fashion, focusing more on the theory, algorithm, and numerical results.

Simulating quantum many-body systems is a fundamental problem in physics with many applications, including the understanding and design of quantum materials, molecules and matter [15–17]. However, the general simulation problem has been proven to be hard [19, 20]. This has motivated the development of various classical frameworks to tackle the problem heuristically, including Wilson's Numerical Renormalization Group NRG) [26], White's Density Matrix Renormalization Group DMRG) [27, 28] and VMC [29, 30]. It has also motivated strategies for leveraging quantum hardware for simulation, where frameworks such as quantum phase estimation [32, 33], Hamiltonian simulation [34–36] and variational quantum algorithms VQA) [37–39] have been proposed to take advantage of devices with potential quantum advantage.

Despite the hardness of the problem, a useful observation is that, upon scaling the system size, many properties of interest (i.e., observables, entanglement entropy, spectrum,

etc.) can exhibit minimal fluctuations and demonstrate consistent behavior across adjacent system sizes. [147–149]. This hints that, given an oracle to query observable properties from the $n-1, n-2, \cdots, 1$-site system with tractable cost, predicting a property in n-site system might be possible. Technically, predicting larger system properties by solving smaller system properties is an appealing direction, allowing the observations and theory relevant to small-system solvers to be transferred into large-scale many-body system solvers. Historically, numerical renormalization formulations such as NRG and DMRG have been motivated by this observation.

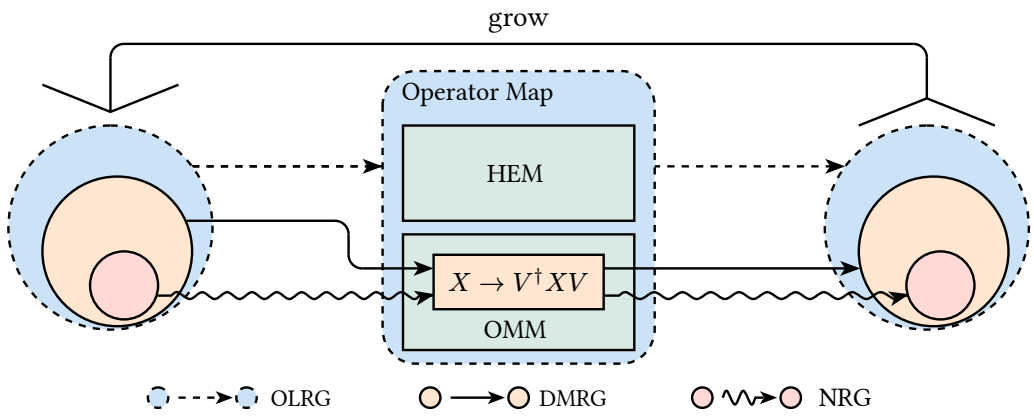

Figure 4.1: Workflow of NRG, DMRG, and OLRG. $X \to V^\dagger XV$ is the basis transform into low-energy spectrum subspace or chosen-state subspace. From the left is the set of relevant operators in the calculation. The operator map finds a virtual set of relevant operators on the right. The red color corresponds to Wilson's NRG, whose relevant operators only contain Hamiltonian. The orange color corresponds to White's DMRG, whose relevant operators contain Hamiltonian and boundary operators. The blue color corresponds to OLRG, whose relevant operators contain arbitrary operators involved in the calculation. Both operator maps for NRG and DMRG are linear basis transforms, which fall into the category of OMM. The operator map for OLRG is an arbitrary operator map, which contains both OMM and HEM.

The success of NRG and especially DMRG relies largely on the linear ansatz for the operator map and the loss function of the spectrum error or average expectation error of a chosen state [26–28, 150], which can be solved directly using an eigensolver. Linearity is one of the key reasons behind the fast convergence of DMRG, where the local optimal point of the loss function can be identified directly using an eigensolver. However, this linearity also

limits the expressiveness, leading to the limitations encountered by MPS in simulating high-dimensional systems or long-time dynamics. While the tensor network formalism [151–164] has been developed to address this issue, it is possible that the limitation of expressiveness in linear functions may be fundamental [165–172]. As for the loss function, the spectrum error or average expectation error of a chosen state is a natural choice. However, compared to optimizing the error of the target property directly, the spectrum error or average expectation error of a chosen state may introduce a bias when these are not the target property themselves [30]. Given these arguments, it is natural to ask whether one could generalize the NRG and DMRG algorithms, such that instead of using a linear operator map for an intermediate target, an arbitrary operator map can be used as an ansatz for the final target.

We view this question through a modern machine-learning perspective. For both ground state and dynamics, predicting properties of a n-site system from smaller system properties can be framed as a machine-learning task. An algorithm can learn from a data set of $n-1, \cdots, 1$-site properties and predict the n-site property. Looking closely at NRG and DMRG, the algorithms take a set of n-site operators as input and generate a compact representation of these operators as output. Then, the compact operators are grown by one site and used as the operators for an $n+1$-site system as shown in Figure 4.1. From a learning perspective, aside from the details of the loss function for a specific problem, this can be seen as a generative learning procedure, where the model tries to generate a set of virtual compact operators relevant to calculating the target property. Thus, we take as the key idea of our algorithm the perspective of learning operator maps, i.e., generalizations of the linear operator maps in NRG and DMRG, as opposed to learning parameterized states as in the tensor network formalism.

We call the algorithm thus the OLRG. In a similar spirit to the renormalization group and embedding theory [173–175], the OLRG framework provides a route to leverage the techniques developed for small-system solvers for large-scale many-body systems. While our framework is general enough to address arbitrary simulation problems, we prove rigorous bounds for the loss function of real-time dynamics in particular. The loss function is designed to minimize the error of a target property directly instead of an intermediate target. The philosophy of removing intermediate steps is commensurate with end-to-end (e2e) learning [176, 177], which has been a core concept in the success of modern deep learning, leading to many state-of-the-art results [178–181]. As a result, instead of modeling a quantum state, arbitrary operator maps are allowed as ansatzes by this variational principle. For classical simulation, the operator map is called an OMM. In this chapter, we will focus on demonstrating OMM implemented by a neural network. Furthermore, this variational principle has a broader application when considering operators not repre-

sented by matrices, such as a pulse sequence in a real quantum device. Considering an operator map of a problem Hamiltonian to a device Hamiltonian expression, this leads to an alternative quantum simulation algorithm for near-term devices that are not fully fault-tolerant [40, 118, 182, 183], which we call the HEM.

This chapter is organized as follows. We first review the NRG and DMRG algorithms in their traditional setting in Section 4.1. In Section 4.2, we introduce the general framework of OLRG, including the *scaling consistency* condition, a general principle guiding the design of the loss function. Then, we explore the concrete scaling consistency condition for the real-time evolution of a geometrically local Hamiltonian. In Section 4.5, we introduce two variant algorithms of OLRG for classical and quantum simulation of real-time dynamics. By viewing the operator map as OMM, we discuss using OLRG as a variational algorithm on conventional computers. By viewing the operator map as HEM, we discuss using OLRG as a variational quantum algorithm for near-term digital-analog quantum devices. In Section 4.7, we study the two-point correlation function of a one-dimensional ($1D$) TFIM to demonstrate our theory and the effects of different hyperparameters for OMM and HEM. Finally, we discuss open questions and potential improvements in Section 4.9.

4.1 NRG and DMRG in the Traditional Formulation

To further understand the motivation and thought process of the NRG and DMRG algorithms, we will review them in their traditional formulations from an operator map perspective. Wilson's NRG starts with a simple idea: to obtain the low-energy properties of a N-site system, where N is a large number or infinity. We can start by dividing the N-site system into identical n-site small systems named a block, assuming $N = 2^q n$. Then, the block Hamiltonian H_{S_n} on a small system S_n of n sites can be compressed from $2^n \times 2^n$ to some size $M \times M$ by finding an approximation of the matrix H_{S_n}. Wilson proposed to use a low-rank approximation of the Hamiltonian $V_n^\dagger H_{S_n} V_n$ such that $V_n^\dagger H_{S_n} V_n$ preserves the low-energy eigenstates of H_{S_n}. Naturally, V_n are the M lowest eigenstates of H_{S_n}, which preserves the low-energy spectrum. Then, we can grow the system by copying the n-site system to form a $2n$-site system and repeat the process. For single particle models such as $H_{S_n} = \sum_i X_i$, this is relatively straightforward. Since each small system of n sites does not interact with each other, the $2n$-site Hamiltonian $H_{S_n} H_{S_n}$ can be written as $H_{S_n} \otimes I + I \otimes H_{S_n}$, and with the compressed Hamiltonian $V_n^\dagger H_{S_n} V_n \otimes I + I \otimes V_n^\dagger H_{S_n} V_n$. With q steps, this process should eventually lead to a $n2^q$-site system, approximating an infinite system. In summary, NRG uses the error of the low-energy spectrum as the optimization target and a basis transform V_n as the ansatz. Thus, at each step, we produce a

virtual Hamiltonian $V_n^\dagger H_{S_n} V_n$ to replace the original one. However, such approximation is sub-optimal for two reasons: (i) the choice of low-energy eigenstates is suboptimal when the only properties of interest are the ground-state properties. (ii) copying the small system does not reflect the effect of boundary conditions. As a result, NRG works well for low-energy spectrum problems without a strong effect on boundary condition [26] but fails for more general quantum lattice ground-state problems in real-space form [184].

Historically, White's DMRG was presented as a generalization of NRG. We will explain the process using $1D$ TFIM Hamiltonian $H_n = \sum_i Z_i Z_{i+1} + h \sum_i X_i$. Assuming a chain of "good" compression $V_1, V_2, \cdots, V_{N-1}$ in a similar RG process has been found but for arbitrary ground state observables in the infinite system. Then, given a n-site system S_n and environment E_n, V_{n+1} should produce a good approximation of $S_n \bullet\bullet E_n$ named a superblock, where $\bullet$ means a new physical site, and its Hamiltonian is written as $H_{S_n} \otimes I^{n+2} + H_{S_n\bullet} + H_{\bullet\bullet} + H_{\bullet E_n} + I^{n+2} \otimes H_{E_n}$, and without compression

$$\begin{aligned} H_{S_n\bullet} &= I^{\otimes n-1} \otimes Z \otimes Z \otimes I^{\otimes n+1} \\ H_{\bullet\bullet} &= I^{\otimes n} \otimes Z \otimes Z \otimes I^{\otimes n} \\ H_{\bullet E_n} &= I^{\otimes n+1} \otimes Z \otimes Z \otimes I^{\otimes n-1} \end{aligned} \tag{4.1}$$

The construction of $S_n \bullet\bullet E_n$ requires addressing the effect of a neighboring site at the boundary $S_n\bullet$ then addressing the effect of environment bath $\bullet E_n$. Thus, a superblock can be a good test of the boundary and environment effect. Then, applying V_{n+1} on $S_n\bullet$ and $\bullet E_n$ will result in a virtual $S_n \bullet\bullet E_n$ system. Comparing an arbitrary ground state observable A on $S_n\bullet$ results in the following error estimation,

$$\left\| \mathrm{tr}(\rho_{S_n\bullet} A) - \mathrm{tr}\left(V_{n+1}^\dagger \rho_{S_n\bullet} A V_{n+1} \right) \right\| \tag{4.2}$$

where $\rho = \mathrm{tr}_{\bullet E_n}(|\psi_0\rangle)$ is the ground state on $S_n\bullet$. Since A is an arbitrary observable, the optimal V_{n+1} should be the isometric map to the low-rank approximation of ρ [27, 150], namely a basis transform into the ground state subspace. To build the Hamiltonian of the next $2n+4$-site superblock, except the virtual Hamiltonian from the $n+1$-site system and environment, we also need the virtual operator $H_{S_{n+1}\bullet}$, $H_{\bullet\bullet}$ and $H_{\bullet E_{n+1}}$, which are $I^{\otimes n} \otimes Z$, $Z \otimes I^{\otimes n}$ and $I^{\otimes n+1}$ in the $n+1$-site space. Then, one can repeat this process until the target system size is N. In summary, in DMRG, the superblock is used instead of a block to test the effect of boundary and bath. Besides the Hamiltonian itself, we keep track of some extra virtual operators to build the superblock. The transform V_{n+1} is then optimized based on the loss function defined on the superblock. A comparison of loss functions is shown in Table 4.1. This thought process of generating virtual operators describing the same n-site system is the key idea of our generalization.

Method	Loss function	RG transformation
NRG [26]	low-energy spectrum error	isometry
DMRG [27, 28]	$\|\rho - \hat{\rho}\|_F$ $\quad \operatorname{rank}\hat{\rho} \leq M$	isometry
OLRG	scaling consistency	arbitrary

Table 4.1: A review of previous RG-like variational methods by loss function at each scale and RG transformation. H denotes the Hamiltonian. ρ denotes the density matrix. M denotes the maximum rank of the low-rank approximation.

4.2 Operator Learning RG Framework

The procedure in Section 4.1 is the key idea of our generalization. The strategy can be summarised as follows:

Instead of considering all n-site operators and having a static definition of operators in the block object, we only focus on the subset of operators required to calculate the target output.

We call these operators as the set of "relevant" operators and denote them as a set S_n for a n-site system. And denote $S_n^{(0)}$ as the ground truth without altering any relevant operators. In NRG, this is only the Hamiltonian $S_n = \{H_n\}$, and in our $1D$ TFIM DMRG example, this is $S_n = \{H_n, I^{\otimes n-1} \otimes Z, Z \otimes I^{\otimes n-1}\}$. We then look at the target output and rough RG procedure to trace back the minimum required operations by removing the details from Section 4.1.

First, we denote the target output at n-site system calculated by these operators as $p_n[S_n]$, where p_n is called a property function. Here by "property function", we mean a function that maps the set S_n to a scalar value, such as the ground state energy, the two-point correlation function, the entanglement entropy, etc. A formal definition of p_n is introduced in Section 4.3.1. Denote the operator map as $f_n^\theta : \mathcal{A}_n \to \mathcal{A}_n$, where $\mathcal{A}_n$ is the space of Hermitian operators and θ are the parameters. The operator map f_n^θ maps a Hermitian operator to another Hermitian operator of the same number of sites. For example, when p_n is the expectation of a two-point correlator at time T evolved by the TFIM Hamiltonian,

$$S_n^{(0)} = \{\rho_0, H_n, B_n, O_n^{ab}\} \tag{4.3}$$

where $B_n = I^{\otimes n-1}Z$, applying f_n^θ onto $S_n^{(0)}$ result in

$$S_n^{(1)} = f_n^\theta[S_n^{(0)}] = \{f_n^\theta[\rho_0], f_n^\theta[H_n], f_n^\theta[B_n], f_n^\theta[O_n^{ab}]\} \tag{4.4}$$

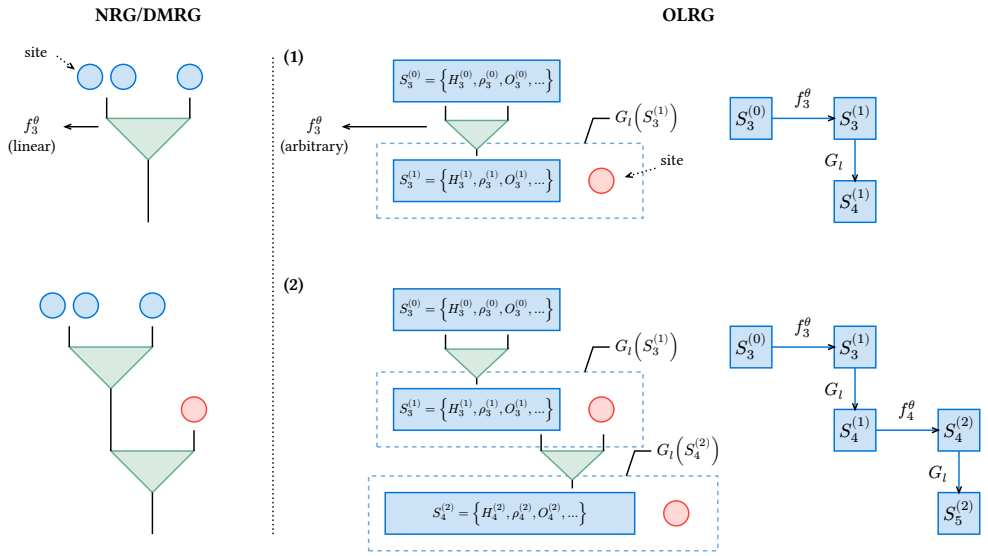

Figure 4.2: Illustration of three OLRG growing steps starts from a 3-site system. G_l denotes the operation of adding k sites into the system. f_n^θ denotes the operator map of n-site system with parameters θ. (*Left*) When f_n^θ is an isometric matrix, this process is equivalent to a canonical MPS. The red circles mark the physical legs, and the blue triangle denotes the isometric matrix. (*Middle*) The blue box depicts the set of operators that are used to calculate the target property. The dashed box denotes the grown box. The arrow represents the operator map f_n^θ. (*Right*) The flow chart of this process. $S_5^{(2)} = G_1[S_4^{(2)}] = (G_1 \circ f_4^\theta)[S_4^{(1)}] = (G_1 \circ f_4^\theta \circ G_1 \circ f_3^\theta)[S_3^{(0)}]$.

In NRG and DMRG, this is the basis transform $f_n^\theta[X] = V_n^\dagger X V_n$. Constructing a $n+1$-site operator from a n-site operator is denoted as G_1. In DMRG, this corresponds to the step that adds one site $\bullet$. Then, starting from an operator X in n-site system, we can write down the output operator in the N-site system after performing the entire RG process as $\underbrace{G_1 \circ f_{N-1}^\theta \circ \cdots \circ G_1 \circ f_n^\theta}_{q \text{ times}}[X]$. This is the corresponding virtual operator in the N-site system, where $N = n + q$. We can then obtain the entire set of virtual relevant operators in the N-site system, denoted as $S_N^{(q)} = \underbrace{G_1 \circ f_{N-1}^\theta \circ \cdots \circ G_1 \circ f_n^\theta}_{q \text{ times}}[S_n^{(0)}]$, where (q) means we transformed the system for q times by applying $f_n^\theta, \cdots, f_{N-1}^\theta$. Thus the error of output is $\left\| p_N(S_N^{(0)}) - p_N(S_N^{(q)}) \right\|$, where $S_N^{(0)} = G_1^q[S_n^{(0)}]$ by definition. If we can minimize this error, we will obtain a set of virtual operators $S_N^{(q)}$ resulting a similar property value. In summary, it doesn't matter if our set of virtual operators $S_N^{(q)}$ is a complete set of real operators describing the N-site system properties. As long as it can compute the property p_N with a good error, it is probably a set of real operators.

Thus, instead of approximating states, the NRG and DMRG algorithms can be viewed as generative learning algorithms [185] that generate a set of virtual relevant operators at each scale. As shown in Figure 4.2 (right), **(1)** starting from $S_3^{(0)} = \{H_3^{(0)}, \rho_3^{(0)}, O_3^{(0)}, \cdots\}$, we generate $S_3^{(1)} = f_3^\theta[S_3^{(0)}] = \{H_3^{(1)}, \rho_3^{(1)}, O_3^{(1)}, \cdots\}$. Assuming this new set of operators is sufficient to approximate the properties we would like to calculate, we use this set of operators as if they were the ground truth. If they are "good" approximations, we should be allowed to grow the virtual system by 1 site (marked by a dashed box). We can obtain the next 4-site system as $S_4^{(1)} = G_1[S_3^{(1)}]$. **(2)** We then use $S_4^{(1)}$ as the input to generate another set of operators $S_4^{(2)} = f_4^\theta[S_4^{(1)}]$ and grow it into $S_5^{(2)}$ to calculate $p_5[S_5^{(2)}]$ as an approximation of $p_5[S_5^{(0)}]$.

In the case of classical simulation, the virtual relevant operators should use less storage than the original to keep the algorithm running within a constant memory. Thus in NRG and DMRG, they are generated by linear isometric functions $f_{n_q}^\theta(X) = V_{n_q}^\dagger X V_{n_q}, n_q = n, n+1, \cdots, N$. These functions take the matrix of the original operator and return a compressed matrix at each scale. As shown in Figure 4.2 (left), the chain of f_{n_q} forms the canonical MPS. Next, the functions $f_{n_q}^\theta$ are optimized by a loss function that is defined on the data of block NRG) or superblock DMRG) generated by a small-system solver, e.g., the low-energy spectrum error or average expectation error of a chosen state. The loss function heuristically controls the final error. Thanks to the linear nature of $f_{n_q}^\theta$, the optimal point of such loss functions can be identified directly using an eigensolver without actually generating the whole data set of operators. This loss function heuristically allows

the set $S_n^{(1)} = f_n^\theta[S_n^{(0)}]$ to grow into the set $S_{n+1}^{(2)} = G_1[S_n^{(1)}]$ with the final error controlled. Thus, the learning process can be repeated until the target system size is reached.

This perspective further guides us to investigate the requirement of implementing proper loss functions, such that the requirement of a linear f_n^θ can be extended. As a result, we suggest a fundamental principle for creating such loss functions, which we call the *scaling consistency condition*. This principle is outlined and compared with other heuristic approaches in Table 4.1. In the following, we define this process and its underlying concepts through formal definitions and corresponding examples. Then, we introduce the error upper bound due to satisfying the scaling consistency condition. Next, we look into the real-time evolution of a geometrically local Hamiltonian and further reduce the loss function to local-observable errors. Last, we discuss these local observables and the corresponding evaluation of the loss function. In fact, this paradigm above shares the same philosophy as so-called `duck typing` in programming languages [186, 187] (e.g as used in a DMRG tutorial [188]). By way of definition,

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

4.2.1 The Scaling Consistency Condition

Next, we explain how to define a tractable loss function such that for arbitrary f_n^θ we can (a) preserve the properties we want to calculate in the target system and (b) allow the system to grow to the target size with final error controlled. This is the main goal of the following definitions and theorems. We first need to formally define G_l to understand what it means to grow the system by l sites. Before diving into the formal definition, we can look at how one rewrites the n-site Hamiltonian H_n of the $1D$ TFIM as the $n-1$-site Hamiltonian

Example 1 (Growing the TFIM Hamiltonian)**.** For example, for the $1D$ TFIM model, the Hamiltonian of an n-site $1D$ system is constructed by extending the Hamiltonian of an $n-1$-site $1D$ system and adding terms that incorporate the n-th site:

$$\begin{aligned} H_n &= G_1[H_{n-1}] \\ &= H_{n-1} \otimes I + \underbrace{I^{\otimes n-1} \otimes Z \otimes Z}_{\text{new site interaction}} + \underbrace{hI^{\otimes n} \otimes X}_{\text{new site field}}, \end{aligned} \tag{4.5}$$

and more generally we can rewrite the $n+l$-site $1D$ TFIM Hamiltonian as n-site $1D$ TFIM

Hamiltonian as follows

$$\begin{aligned} H_{n+l} =& H_n \otimes I^{\otimes l} + I^{\otimes n-1} \otimes Z \otimes Z \otimes I^{\otimes l-1} \underbrace{+\cdots+}_{l-2 \text{ terms}} \\ & I^{\otimes n+l-2} \otimes Z \otimes Z + I^{\otimes n} \otimes h \cdot X \otimes I^{\otimes l-1} + \\ & I^{\otimes n} \otimes h \cdot X \otimes I^{\otimes l-1} \underbrace{+\cdots+}_{l-2 \text{ terms}} I^{\otimes n+l-1} \otimes h \cdot X \end{aligned} \tag{4.6}$$

We can see the Equation (4.5) as breaking the entire system into 1-site fragment, then each time G_l is applied, it puts l fragments back. Naturally, one can define the growing operator as a building operation that puts l fragments back after dividing the total N-site operator. This is the main idea of the following definition.

Definition 1 (Growing operator, informal)**.** A growing operator G_l is a superoperator that increases the size of the system by l-sites. This superoperator formalizes how one grows a given operator A_n of n sites by l sites. In general, G_l can be represented as follows,

$$G_l[A_n] = A_n \otimes R[A_n] + \sum_i B_n^i \otimes R[B_n^i], \tag{4.7}$$

where B_n^i and $R[B_n^i]$ are pairs of operators that connect the n-site system and the l-site environment. We call the operators B_n^i the boundary operators. The index i goes over all possible decomposition and thus can be exponentially large in the most general case. A more detailed definition will be given in Section 4.3.1.

In summary, the concept of a growing operator is pivotal in understanding how an operator of a $n+l$-site system can be expressed in terms of an operator of a n-site system by first dividing the total system of N sites into N/l fragments. This will be particularly clear to those familiar with tensor networks: the growing operator can be analogously represented as a tensor within the Tensor Network Operator (TNO) formalism [189, 190]. Each time applying the tensor creates a few new physical legs. However, in our subsequent theorem, we opt not to use the TNO formalism. Our rationale is to present our proof from an algebraic standpoint, which we find more suitable for our generalization purposes. To further elucidate this concept, the growing operator can also be applied to other operators, such as the density matrix operator of the zero-state.

Example 2 (Growing the zero state)**.** $\rho_n = (|0\rangle\langle 0|)^{\otimes n}$ can be written as,

$$\rho_n = G_1[\rho_{n-1}] = \rho_{n-1} \otimes |0\rangle\langle 0| . \tag{4.8}$$

As mentioned, the growing operator is defined by dividing the total system into fragments and combining them. The two-point correlation function is a typical example of an operator that requires dividing the total system into fragments otherwise the definition of the two-point correlation function could be ambiguous.

Example 3 (Growing the two-point correlator)**.** In a similar vein, and without loss of generality, consider a two-point correlator expressed as

$$O_n^{xy} = I^{\otimes x} \otimes X \otimes I^{\otimes y} \otimes Y \otimes I^{\otimes n-x-y-2}. \tag{4.9}$$

The growing operator from a smaller size $n-1$ to a larger size n can be written as

$$O_n^{xy} = G_1[O_{n-1}^{xy}] = \begin{cases} O_{n-1}^{xy} \otimes I & \text{if } n < x \\ & \text{or } x < n < y \\ & \text{or } n > y \\ O_{n-1}^{xy} \otimes X & \text{if } n = x \\ O_{n-1}^{xy} \otimes Y & \text{if } n = y \end{cases} \tag{4.10}$$

The e2e-style loss function can be written as the error $\left\| p_N[S_N^{(0)}] - p_N[S_N^{(q)}] \right\|$, as demonstrated in Figure 4.2 (blue nodes). Then, our goal will be minimizing this loss function by optimizing the parameters θ in the OLRG steps. The parameters θ can appear in two places: (a) the operator map $f_{n_q}^\theta$ itself, similar to NRG and DMRG; (b) the output operator $f_{n_q}^\theta[X]$. We will discuss them in Section 4.5. However, this quantity, as the loss function, is infeasible to calculate. We wish to simplify it into a more tractable form within each growing step. Naively, as shown in Figure 4.2 (blue nodes), for calculating 5-site system starting from 3-site system, one may use $\left\| p_3[S_3^{(0)}] - p_3[S_3^{(1)}] \right\| + \left\| p_4[S_4^{(1)}] - p_4[S_4^{(2)}] \right\|$ as the loss function instead. However, this does not necessarily bound the final error. In fact, we show that such a loss function fails to bound the error in Section 4.7 as the 0^{th}-order loss function in our theory. This motivates us to introduce the following definition and theorem.

Definition 2 (ϵ-scaling consistency)**.** An operator map $f_n^\theta : \mathcal{A}_n \to \mathcal{A}_n$ is said to satisfy ϵ-scaling consistency for a set of relevant operators S_n and property p_N where $N \geq n$, if $\exists \epsilon > 0, \forall q = 1, 2, \cdots, (N-n)/l$ we always have

$$\left\| p_N[G_l^q[S_n]] - p_N[(G_l^q \circ f_n^\theta)[S_n]] \right\| \leq \epsilon. \tag{4.11}$$

The ϵ-scaling consistency condition measures the error caused by applying the operator map $f_{n_q}^\theta$ at each OLRG step. This error appears because the n-site part within a N-site

system is transformed by f_n^θ, thus resulting in an error between two N-site systems ($G_l^q[S_n]$ and $(G_l^q \circ f_n^\theta)[S_n]$). It is worth noting that the conceptualization of growing operators shares numerous commonalities with Density Matrix Embedding Theory (DMET) [173, 174]. For readers versed in DMET, the terminology of "scaling consistency" also takes inspiration from the self-consistency principle in DMET. In OLRG, the optimization of the operator map is directed not toward aligning the properties of individual fragments with the original system but rather toward achieving consistency in properties across varying scales. Denote one step of growing and transforming as $D_l = G_l \circ f_{n_q}^\theta$ called an OLRG step. For convenience, we let n_q being adaptive to the number of sites in D_l. The target property can be written as $p_N(D_l^q[S_n])$. Then we have the following theorem:

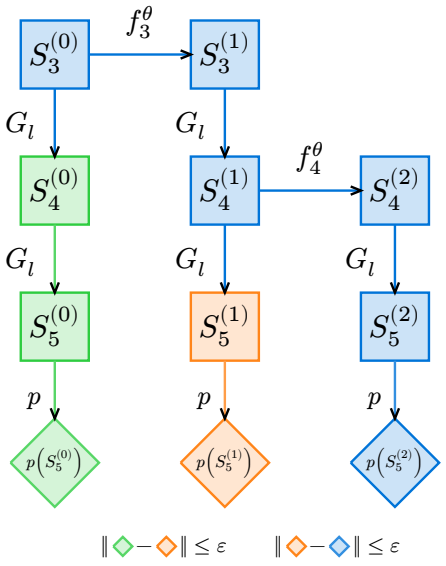

Figure 4.3: Comparing 2 OLRG growing steps and the ground truth starts from a 2-site system. Denote $S_n^{(q)}$ as the system of size n by applying D_l for q times. Green nodes depict the ground truth. Blue nodes represent algorithm growing steps. Orange nodes represent the 5-site system applying only f_3^θ. The red nodes denote the computed property at each 5-site system $p(G_l[S_n^{(q)}])$.

Theorem 1 (System scaling error)**.** *For target system size N and starting system size n, where $N \geq n$, if for $n_q = n, n+l, \cdots, N$, the operator map $f_{n_q}^\theta$ satisfy the ϵ-scaling*

consistency condition for $S_n, S_{n+1}, \cdots, S_{N-1}$, *and* $q = (N-n)/l$ *then* $\exists \epsilon > 0$ *such that*

$$\|p_N[G_l^q[S_n]] - p_N[D_l^q[S_n]]\| \leq q\epsilon. \tag{4.12}$$

Proof. This is obvious by inserting zeros of neighboring values $p(G_l^{q-j} D_l^j[S_n]) - p(G_l^{q-j} D_l^j[S_n])$ into the left-hand-side then use triangular inequality. A visual proof is shown in Figure 4.3. Section 4.3.1 provides a more detailed proof. □

Theorem 1 suggests that to implement an e2e-style loss function rather than minimizing the differences in properties at the current system size, one should optimize the discrepancy between properties at the target system size N at each OLRG step. This concept is exemplified by the error observed between the last blocks of each column in Figure 4.3. Instead of optimizing properties from the blue blocks ($S_3^{(0)}$ and $S_3^{(1)}$, $S_4^{(1)}$ and $S_4^{(2)}$), one should optimize the properties from the blocks at the bottom ($S_5^{(0)}$ and $S_5^{(1)}$, $S_5^{(1)}$ and $S_5^{(2)}$). With Theorem 1, we convert the problem of reducing the error $\|p(G_l^q[S_n]) - p(D_l^q[S_n])\|$ into reducing the error defined by ϵ-scaling consistency (Definition 2). While the quantity in ϵ-scaling consistency is still infeasible to evaluate, intuitively, such error is caused by applying f_n^θ to the n-site system. Thus, the error must come from the change of some operators in the n-site system. To control the error, we only need to expand our set of relevant operators to include these operators. In the next subsection, while the rigorous ϵ-scaling consistency condition for the ground state and imaginary time dynamics remains an open question, we will show what kind of operators in the n-site system will contribute to this error for the real-time evolution of a geometrically local Hamiltonian.

4.2.2 Loss Function for Real-Time Evolution

Nevertheless, when we look closer to a more realistic system, it is usually geometrically local. More specifically, geometrically w-local means given a Hamiltonian of the form $H_n = \sum_a H_a$, each term H_a can only act on neighboring w sites geometrically. In this case, applying G_l for q times will result in the following equation, where by definition, $G_l^q = G_{ql}$ and,

$$G_{ql}[H_n] = H_n \otimes I^{\otimes kq} + \sum_{i \in (\partial H_n)^{G_l}} B_n^i \otimes R(B_n^i) + I^{\otimes n} \otimes K, \tag{4.13}$$

where $(\partial H_n)^{G_l}$ denotes a set of operators acting on the boundary of H_n. The set $(\partial H_n)^{G_l}$ will saturate once the growing operator applies outside the system boundary as demonstrated in Figure 4.4. The size of $(\partial H_n)^{G_l}$ is proportional to the boundary size of the

system S_n and the number of operators $B_n^i \otimes R[B_n^i]$, as previously defined in the context of a growing operator G_l (depicted in the yellow band in Figure 4.4). Furthermore, K represents the Hamiltonian of the environment. A more detailed and formal discussion about the set $(\partial H_n)^{G_l}$ is included in Section 4.3.2, where the geometrically local Hamiltonian is generalized into the geometrically local Hamiltonian with constant non-geometrically local terms. This approach simplifies the criterion for scaling consistency, necessitating consistency only within $(\partial H_n)^{G_l}$, since geometrically, the operator transformation f_n^θ affects only operators within this range. This leads to the following proposition.

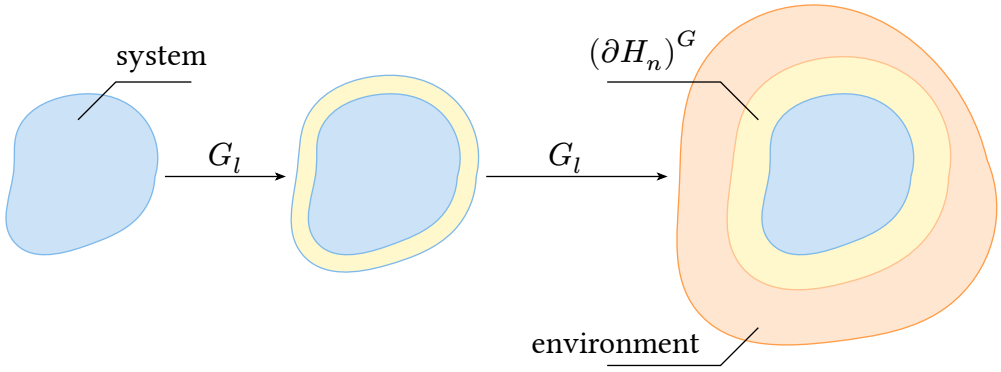

Figure 4.4: For a geometrically w-local Hamiltonian, the growing operator stops changing the system after it grows outside the boundary band of stretch w after applying $G_l^2 = G_{2l}$. This results in a saturated yellow band where only terms within this yellow band interact with the system Hamiltonian.

Proposition 1. If we can effectively break down the property function $p_N(G_l^q[S_n])$ and $p_N(G_l^q[f_n^\theta(S_n)])$ into expectation values separately on system and environment, this might offer a more practical way to develop the e2e-style loss function:

$$\begin{aligned} p_N(G_l^q[S_n]) &= \sum_{i=0}^{\infty} \alpha_i \langle A_i \rangle \langle B_i \rangle \\ p_N(G_l^q[f_n^\theta(S_n)]) &= \sum_{i=0}^{\infty} \alpha_i \langle A_i' \rangle \langle B_i \rangle \end{aligned} \tag{4.14}$$

where i is the index of the series expansion, α_i represents the scalar factor at each order, $\langle A_i \rangle$ represents observables on the n-site system and $\langle B_i \rangle$ corresponds to observables on the kq-site environment. $\langle A_i' \rangle$ is the transformed observable on the n-site system. For example,

if $\langle A_i \rangle = tr(\rho_0 \exp\{itH_n\} A_i \exp\{-itH_n\})$ is the expectation of a time evolved observable, then $\langle A_i' \rangle = tr(f_n^\theta[\rho_0] U'(t)^\dagger f_n^\theta[A_i] U'(t))$ is the expectation value re-calculated using the virtual operators, where $U'(t) = \exp\Big\{-it f_n^\theta[H_n]\Big\}$. If we optimize our operator mapping function such that $\|\langle A_i \rangle - \langle A_i' \rangle\| \leq \epsilon$, it follows that $\Big\|p(G_l^q[S_n^{(0)}]) - p(G_l^q[f_n^\theta(S_n^{(0)})])\Big\| \leq \epsilon \sum_{i=0}^{\infty} \alpha_i \langle B_i \rangle$. Provided that $\sum_{i=0}^{\infty} \alpha_i \langle B_i \rangle$ converges to a finite value, the convergence of the e2e-style loss function can be ensured.

We further explore the expectation value of an observable $O_n(T)$ in the Heisenberg picture $O_n(T) = e^{iTH_n} O_n e^{-iTH_n}$, where T is the total evolution time, H_n is a geometrically local Hamiltonian, with an product state ρ_n as initial state,

$$p(S_n) = tr(\rho_0 e^{iH_n T} O_n e^{-iH_n T}). \tag{4.15}$$

Without loss of generality, we assume O_n is local and $G_l(O_n) = O_n \otimes R[O_n]$. Because ρ_n is a product state such that $G_k(\rho_n) = \rho_n \otimes R[\rho_n]$. This expectation $p(S_n)$ can expand into a series of expectation values in the n-site system and the environment (detailed in Section 4.4). Thus, we find the desired series expansion proposed in Proposition 1. This leads us to the subsequent theorem:

Theorem 2 (Real-time ϵ-scaling consistency)**.** *Given that a w-local Hamiltonian H_n and its growing operator G_l will saturate, denote the set as $(\partial H_n)^{G_l}$. $\exists \epsilon > 0$ and expectation values $\chi(S_n)$, such that if $\forall \chi$ we have*

$$\Big\|\chi(S_n) - \chi(f_n^\theta[S_n])\Big\| \leq \epsilon. \tag{4.16}$$

For $S_n = \{H_n, B_n^i, \rho = \rho_n \otimes R[\rho_n], O = O_n \otimes R[O_n]\}$ and $N = n + kq$ then the error of expectation $p_N[G_k^q[S_n]] = \Big\langle \rho e^{iTH_N} O e^{-iTH_N} \Big\rangle$ is bounded by

$$\begin{aligned} &\Big\|p_N[G_l^q[S_n]] - p_N[(G_l^q \circ f_n^\theta)[S_n]]\Big\| \\ &\leq \epsilon C \exp\Big\{T\Big\|(\partial H_n)^{G_l}\Big\|C/2\Big\}, \end{aligned} \tag{4.17}$$

where C is a constant, T is the total evolution time. A detailed theorem and its proof can be found in Section 4.4

Theorem 2 gives a single step error, thus combined with Theorem 1, we have the total error of q steps upper bounded by

$$q\epsilon C \exp\Big\{T\Big\|(\partial H_n)^{G_l}\Big\|C/2\Big\}. \tag{4.18}$$

This indicates that if we can optimize the error of these expectations χ at $n_q = n, n+l, \cdots, N$-site system due to applying $f^{\theta}_{n_q}$, we should be able to optimize the error of the target property at the target system size N. Since the norm $\left\|(\partial H_n)^{G_l}\right\|$ is a constant, this error is independent of system size N and only accumulates linearly with the number of OLRG steps.

Thus, we can tailor the loss function's design for real-time evolution by considering it as the cumulative error of all observables, as detailed in Theorem 2 with an order cutoff in the series. Then Theorem 2 can guarantee as we increase the order the output will directly move towards the ground truth. This aligns with the e2e learning. Theorem 2 has a very similar bound as Lieb-Robinson bound [147] and other results derived from it [191]. Intuitively, the reason why real-time dynamics can have this bound is also due to the limitation of propagating correlations. However, we do not use Lieb-Robinson bound in our proof in Section 4.4. It is interesting to see if we can derive a similar bound using the Lieb-Robinson bound. This will provide a more general understanding of the error bound in our framework.

Next, based on the proof in Section 4.4, we introduce the definition of χ. Denote the operator B^i_n from Equation (4.13) in the Heisenberg picture as $B^i_n(t) = e^{iH_n t} B^i_n e^{-iH_n t}$ where $0 \leq t \leq T$. The proof of this theorem reveals that the observables are essentially time correlation functions defined on the operator $B^i_n(t)$ and the part of our target observable on the system $O_n(T)$. We refer to these as the TOBC denoted as χ:

$$\langle \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(S_n, T)\rangle = tr(\rho_n \prod_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}} ad_{B^i_n(t),\sigma}[O_n(T)]), \tag{4.19}$$

where ρ_n is the initial state of the system, the multi-index

$$\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma} = i_1, i_2, \cdots, i_k, t_1, t_2, \cdots, t_k, \sigma_1, \sigma_2, \cdots, \sigma_k \tag{4.20}$$

, each index in $\mathbf{i}$ iterates over $(\partial H)^G$, t are the checkpoints in the time evolution, and $\sigma = \pm 1$. As mentioned, the input S_n denotes the set of relevant operators at n-site system. For TOBC specifically, S_n are the primitive operators required to calculate TOBCs defined as $S_n = \{\rho, O_n, H_n, B^i_n\}$ where $B^i_n \in (\partial H_n)^{G_l}$. The notation $ad_{A,\sigma}(B) = AB + \sigma BA$ and $ad_{A,+1}(B) = \{A, B\} = AB + BA, ad_{A,-1}(B) = [A, B] = AB - BA$, their composition denotes the recursive commutators and anti-commutators $ad_{A,+1}ad_{B,+1}(C) = \{A, \{B, C\}\}$, $ad_{A,-1}ad_{B,+1}(C) = [A, \{B, C\}]$. For the kth-order TOBC, the notion of $\prod_{\boldsymbol{i},\boldsymbol{m},\boldsymbol{\sigma}}$ denotes the following product

$$ad_{B^{i_1}_n(t_1),\sigma_1} ad_{B^{i_2}_n(t_2),\sigma_2} \cdots ad_{B^{i_k}_n(t_k),\sigma_k}[O_n(T)]. \tag{4.21}$$

For example, we can write down the TOBC at different orders. For the 0-th order, this refers to the observable $O_n(T)$. For the 1st order, for $0 \leq t \leq T$, we have,

$$\begin{aligned} \chi_{i,t,-1}(S_n, T) &= [B_n^i(t), O_n(T)] \\ \chi_{i,t,+1}(S_n, T) &= \{B_n^i(t), O_n(T)\}. \end{aligned} \tag{4.22}$$

For the 2nd order, for $0 \leq t_1 \leq t_2 \leq T$, we have,

$$\begin{aligned} \chi_{\mathbf{i},\mathbf{t},\{-1,-1\}}(S_n, T) &= [B_n^{i_1}(t_1), [B_n^{i_2}(t_2), O_n(T)]] \\ \chi_{\mathbf{i},\mathbf{t},\{-1,+1\}}(S_n, T) &= [B_n^{i_1}(t_1), \{B_n^{i_2}(t_2), O_n(T)\}] \\ \chi_{\mathbf{i},\mathbf{t},\{+1,+1\}}(S_n, T) &= \{B_n^{i_1}(t_1), \{B_n^{i_2}(t_2), O_n(T)\}\} \\ \chi_{\mathbf{i},\mathbf{t},\{+1,-1\}}(S_n, T) &= \{B_n^{i_1}(t_1), [B_n^{i_2}(t_2), O_n(T)]\}. \end{aligned} \tag{4.23}$$

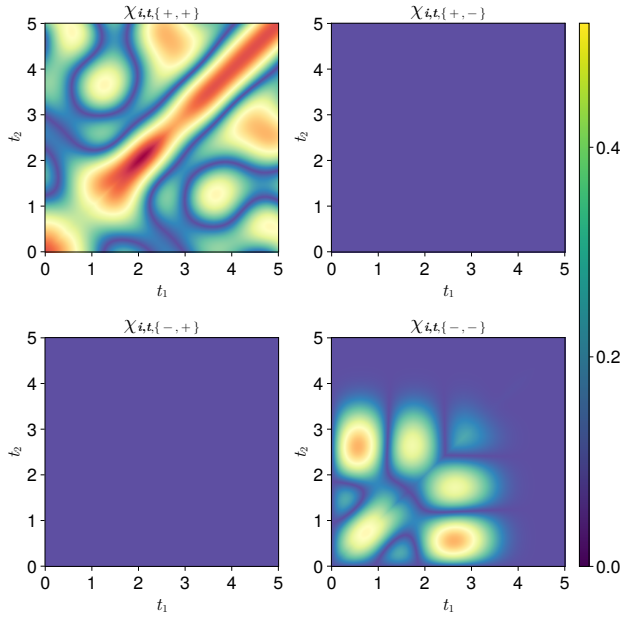

Figure 4.5: 2nd-order TOBC for 5-site $1D$ TFIM at $T = 5.0$ for the two-point correlation function $\langle Z_1 Z_2 \rangle_{T=5.0}$ with $|00000\rangle$ as the initial state and $h = 1.0$.

In practice, the time points $t_1, t_2, \cdots$ are checkpoints from the small-system solver. Many correlators in this setup are nearly zero. These can be further pinpointed by introducing a specific Hamiltonian and observables into the correlator expression. For example,

for a 5-site $1D$ TFIM model, where $B_5^1 = Z_5 = I^{\otimes 4} \otimes Z$, two of its 2nd order TOBC,

$$\begin{aligned} &\langle [Z_5(t_1), \{Z_5(t_2), (Z_1 Z_2)(5.0)\}] \rangle \approx 0 \\ &\langle \{Z_5(t_1), [Z_5(t_2), (Z_1 Z_2)(5.0)]\} \rangle \approx 0, \end{aligned} \tag{4.24}$$

are nearly zero, as shown in Figure 4.5. However, in a worst-case scenario, the number of potentially non-zero TOBC increases exponentially with the order l. Assuming there are M checkpoints, there are $O(2M\|(\partial H_n)^{G_l}\|)^l$ TOBCs. This exponential rise renders a comprehensive evaluation of the entire loss function at higher orders impractical. A uniform sampling at each order without time ordering is proposed as an effective solution. This is because we optimize a summation of the correlators' error, thus resulting in a uniform distribution. When the original dynamics are faithfully approximated, the extra correlators not in time order should also have a small error. In practical terms, this approach involves selecting a batch of operators for sampling, akin to using data batches in Stochastic Gradient Descent (SGD) in conventional deep learning algorithms. Significantly, this sampling technique is highly compatible with advanced accelerated computing frameworks, such as CUDA [61], which are designed for batch operations.

There might be concerns regarding the typically large value of M and the consequent size of each operator batch, potentially leading to an extensive sampling requirement. As shown in Figure 4.5, for nonzero TOBC, in practice, many points are nearly zero and thus contribute little to the loss function. Thus, since M is only a factor in the loss function rather than the small-system solver, if the dynamics of interest are smooth, a fine step size is not necessary in practice to obtain satisfactory results. In Section 4.8.3, we include some additional results on the step size of the TOBC sampling, which does not show a significant difference in the final relative error.

4.3 Formal Definitions

In this section, we provide formal definitions of the OLRG framework. We first introduce the concept of a system, property function, connecting operator, growing operator, and scaling consistency. Then, we prove the error upper bound of the OLRG process.

4.3.1 Scaling Consistency

The scaling consistency condition is generic to arbitrary properties of the system. To demonstrate this, we first introduce the definition of a system, property function, connect-

ing operator, growing operator, and scaling consistency. Then, we prove the error upper bound of the OLRG process

Definition 3 (Many-body Hilbert space)**.** Denote the many-body Hilbert space with d local states and n sites as $\mathcal{H}(\mathbb{C}^d)^{\otimes n}$ and the self-adjoint operators on $\mathcal{H}(\mathbb{C}^d)^{\otimes n}$ as $\mathcal{A}_n$.

Definition 4 (Property function)**.** A property function p_n is a function that maps a set of self-adjoint operators to a real value quantity, denoting the domain of p as $\mathrm{dom}(p_n) \subseteq \mathcal{A}_n \times \cdots \times \mathcal{A}_n$, we have $p_n : \mathrm{dom}(p_n) \to \mathbb{R}$.

Property functions include the expectation value of an observable, the correlation function, the entanglement entropy, energy, etc. As an example, the two-point correlation function on 1st and 2nd sites at time T is defined as

$$\langle Z_1 Z_2 \rangle_T = \mathrm{tr}\Big(\rho_0 U(T)^\dagger Z_1 Z_2 U(T)\Big) \tag{4.25}$$

where ρ_0 is the initial state, $U(T)$ is the time-evolution operator, and Z_1, Z_2 are the Pauli operators acting on the 1st and 2nd sites respectively. Thus, it can be defined as a function on $\mathcal{A}_n \times \mathcal{A}_n$ where one input operator is the initial state ρ_0 and the other is the Hamiltonian H.

Definition 5 (Connecting operator)**.** A connecting operator is the superoperator $R_l : \mathcal{A}_n \to \mathcal{A}_l$ such that given an operator $L \in \mathcal{A}_n$ we have $R[L] \in \mathcal{A}_l$. The expression $L \otimes R_l[L]$ characterizes the connection between the n-site system and l-site system.

To elucidate this concept, consider the following $1D$ TFIM Hamiltonian:

$$H = \sum_i Z_i Z_{i+1} + h \sum X_i \tag{4.26}$$

For $1D$ TFIM, given a $n+l$-site TFIM, the connection between n-site TFIM and l-site TFIM is $Z_n Z_{n+1}$, and the operator on the n-site TFIM is $L = Z_n = I^{\otimes n-1} \otimes Z$, thus the connecting operator on this operator is defined as $R_1(L) = Z$. Similarly, for the Heisenberg Model:

$$H = \sum_i X_i X_j + Y_i Y_j + Z_i Z_j \tag{4.27}$$

There are three types of connections thus $L = X_n, Y_n, Z_n$, and the connecting operators are defined as $R_1(L) = X, Y, Z$ respectively. It is worth noting that although the name "Connecting Operator" was not mentioned in the literature to the best of our knowledge, the concept of the connecting operator has been widely used in the implementation of the DMRG algorithm [188, 192]. The connecting operator describes how one can add new physical sites into an existing system, thus this allows the definition of the growing operator.

Definition 6 (Growing operator)**.** A growing operator is the superoperator $G_l : \mathcal{A}_n \to \mathcal{A}_{n+l}$ such that given an operator $X \in \mathcal{A}_n$ we have $G_l[X] \in \mathcal{A}_{n+l}$. The growing operator has a general form defined using connecting operator R_l as:

$$G_l[X] = X \otimes R_l[X] + \sum_i B_i \otimes R_l[B_i] \tag{4.28}$$

where $\{B_i \in S_n \mid G_l(X) \in \mathcal{H}(\mathbb{C}^d)^{\otimes n+l}\}$, and we call B_i the boundary operators.

The growing operator is defined with a connecting operator R_l. The summation $\sum_i$ does not limit the number of B_i. Thus, such decomposition exists for any operator X. For a Hamiltonian with a general form of n such as the TFIM or Heisenberg Hamiltonian, where the Hamiltonian has a definition over arbitrary n sites, the definition of the growing operator is straightforward.

Corollary 1. For finite operators with definition on a fixed number of sites, because assuming there exists $X_0, X_1, \cdots, X_n$ and $X_0 \in \mathcal{H}(\mathbb{C}^d)$, we have the following relationship

$$\begin{aligned} X_1 &= X_0 R_1[X_0] + \sum_i L_i^1 R_1[L_i^1] \\ X_2 &= X_1 R_1[X_1] + \sum_i L_i^2 R_1[L_i^2] \\ &\cdots \\ X_n &= X_{n-1} R_1[X_{n-1}] + \sum_i R_1[L_i^n] \end{aligned} \tag{4.29}$$

thus, the final operator X_n is a summation of single operator strings. Without limiting the summation $\sum_i$ to be polynomial, we can always decompose a given operator on the summation of n single operator strings denoted as X_n. This allows the definition of all previous operators $X_0, \cdots, X_{n-1}$. Thus, following this procedure, we can define the growing operator for any operator X on a fixed number of sites, such as the two-point correlation function on n-site system at a specific location i, j as we introduced in Section 4.2.

From a different perspective, inspired by DMET [173, 175], one can see such definition as a process of creating fragments of the operator like in DMET. We define the growing operator as the process of adding fragments back. This leads to the definition of the rescalable operator.

Definition 7 (Rescalable Operator)**.** With connecting operator R_l and growing operator G_l, we can define the rescalable operator $\mathcal{X}_n$ as the set $\mathcal{X}_n = \{X_n, \partial X_n, R_l\}$, where X_n is

the operator at current scale, ∂X_n is a set of operators describing the effect of environment on the system, and thus the growing operator G_l of such operator can be recursively defined as

$$G_l[X_n] = X_n \otimes I^{\otimes l} + \sum_{B \in \partial X_n} B \otimes R_l[B] \tag{4.30}$$

where X_0 is a constant operator, $B \in \partial X_n$.

For example, we can define the rescalable Hamiltonian $\mathcal{H}_n$ as the set $\mathcal{H}_n = \{H_n, \partial H_n, R_l\}$, where H_n is the Hamiltonian at current scale, ∂H_n is a set of operators describing the effect of environment on the system referred as the boundary set in the following context, and thus the growing operator G_l of such Hamiltonian operator can be recursively defined as

$$G_l[H_n] = H_n \otimes I^{\otimes l} + \sum_{B \in \partial H_n} B \otimes R_l[B] \tag{4.31}$$

where H_0 is a constant operator, $B \in \partial H_n$.

Definition 8 (Rescalable System)**.** Given a property p_N, where N is the number of sites, we can define the system S_N as a set of operators such that $S_N \in \mathrm{dom}(p_N)$. Then for $n \leq N$, we can define the rescalable system S_n as the set $S_n = \{S_n, \partial S_n, R_l\}$, where S_n is the operator at current scale, ∂S_n is a set of boundary operators, and thus the growing operator G_l of such system can be recursively defined as

$$S_{n+l} = G_l[S_n] = \{G_l[X] \mid X \in S_n\} \tag{4.32}$$

For example, for the two-point correlation function $\langle Z_1 Z_2 \rangle_T$ at time T for 4-site $1D$ TFIM with $|0 \cdots 0\rangle$ as initial state, we have

$$\begin{aligned}
S_4 &= \{|0000\rangle\langle 0000|, H_4, Z \otimes Z \otimes I \otimes I\} & \partial S_4 &= \{Z_4\} \\
S_3 &= \{|000\rangle\langle 000|, H_3, Z \otimes Z \otimes I\} & \partial S_3 &= \{Z_3\} \\
S_2 &= \{|00\rangle\langle 00|, H_2, Z \otimes Z\} & \partial S_2 &= \{Z_2\} \\
S_1 &= \{|0\rangle\langle 0|, H_1, Z\} & \partial S_1 &= \{Z_1\}
\end{aligned} \tag{4.33}$$

where $H_i, i = 1, 2, 3, 4$ is the TFIM Hamiltonian of i sites. The boundary set $\partial S_i, i = 1, 2, 3, 4$ only contains the ∂H_i because $|0 \cdots 0\rangle\langle 0 \cdots 0|$ and $Z, Z \otimes Z, Z \otimes Z \otimes I, Z \otimes Z \otimes I \otimes I$ have no boundary operators. Now, with the definition of the rescalable system, we can study the behavior of an operator map $f_n^\theta : \mathcal{A}_n \to \mathcal{A}_n$ where θ is the parameter of the operator map.

Definition 9 (OLRG step)**.** Given an operator map $f_n^\theta : \mathcal{A}_n \to \mathcal{A}_n$, we define the one OLRG step D_k as applying f_n^θ on all operators in S_n and then growing the system to S_{n+l}, thus we have $D_l = G_l \circ f_n^\theta$. Here we assume D_l is adaptive on the system size n.

Definition 10 (ϵ-scaling consistency)**.** An operator map $f_n^\theta : \mathcal{A}_n \to \mathcal{A}_n$ is said to satisfy ϵ-scaling consistency for system S_n and property p_N where $N \geq n$, if $\exists \epsilon > 0, \forall q = 1, 2, \cdots, (N-n)/l$ we always have

$$\left\| p_N[G_l^q[S_n]] - p_N[(G_l^{q-1} \circ D_l)[S_n]] \right\| \leq \epsilon \tag{4.34}$$

The ϵ-scaling consistency condition allows us to bound the error of the OLRG process. While the OLRG process does not necessarily use the same $f_{n_q}^\theta$ at each step D_l, without loss of generality, we present the following theorem by assuming $f_{n_q}^\theta = f^\theta$ is the same between S_n and S_{n+l} for convenience.

Theorem 3 (System scaling error)**.** *For target system size N and starting system size n, where $N \geq n$, if the operator map f^θ satisfy the ϵ-scaling consistency condition for $S_n, S_{n+1}, \cdots, S_{N-1}$, and $q = (N-n)/l$ then*

$$\| p_N[G_l^q[S_n]] - p_N[D_l^q[S_n]] \| \leq q\epsilon \tag{4.35}$$

Proof. denote $\eta_i = p_N[(G_l^{q-i} \circ D_l^i)[S_n]]$, where $G_l^{q-i} = \underbrace{G_l \circ \cdots \circ G_l}_{q-i \text{ times}}$ and $D_l^i = \underbrace{D_l \circ \cdots \circ D_l}_{i \text{ times}}$

$$= \|\eta_0 - \eta_q\| = \|\eta_0 - \eta_1 + \eta_1 - \eta_q\| \tag{4.36}$$

$$= \left\| \sum_{i=0}^{q-1} \eta_i - \eta_{i+1} \right\| \leq \sum_{i=0}^{q-1} \|\eta_i - \eta_{i+1}\| = q\epsilon \qquad \text{(triangular inequality)} \tag{4.37}$$

□

The above theorem breaks the system error of OLRG into errors between each step at target size N. This allows us to further bound the error of the OLRG process by looking at more specific Hamiltonians and properties.

4.3.2 Growing Operator of Rescalable Local Hamiltonians

In general, the ϵ-scaling consistency cannot be evaluated on a small system directly because Theorem 3 requires evaluating the property function at size N. However, intuitively, the

discrepancy caused by applying f on system S_n can be traced back to the change of some operators in the system of size n. If the interaction of the Hamiltonian is local, the propagation of the discrepancy should not be far. This motivates us to study the rescalable local Hamiltonians defined as follows.

Corollary 2 (Rescalable Local Hamiltonian)**.** For a rescalable Hamiltonian $\mathcal{H}_n$, if $\forall B \in \partial H_n$, B act on x sites and $R(B)$ acts on $w-x$ sites for $x = 0, 1, \cdots, w$, then this Hamiltonian is a w-local Hamiltonian at every scale.

Notably, for local Hamiltonian, $I^{\otimes n} \in \partial H$ because $R[B]$ can act on m at most. Physically, this represents the terms that only affect the environment but not the system. For example, in the TFIM, the $h \cdot X$ term only appears in the environment.

Corollary 3 (Local Hamiltonian)**.** For w-local Hamiltonian of N sites, one can always define the corresponding rescalable w-local Hamiltonian up to N sites.

This is because one can always cut the N-site w-local Hamiltonian into fragments, then we can create the definition of $\mathcal{H}_n$ recursively by defining $\mathcal{H}_1$. Define the H_1 as one fragment, ∂H_1 as the interaction terms between $\mathcal{H}_1$ and another fragment. Thus, we define H_2 as the composition of two fragments and repeat until we have $\mathcal{H}_N$.

Lemma 1 (Boundary set of geometrically local Hamiltonian)**.** For a geometrically local Hamiltonian $\mathcal{H}_{n+l}$, the boundary set ∂H_{n+l} has the following form

$$\partial H_{n+l} = \{I^{\otimes l} \otimes B_i \mid B_i \in \partial H_n\} \tag{4.38}$$

Proof. Without loss of generality, we can always assume $B = \bigotimes_{i=1}^{x} X_i$ where $X_i \in \mathcal{H}(\mathbb{C}^d)^{\otimes}$, because if B is not a tensor product, we can always decompose it onto Pauli basis with coefficients $B = \sum_{\mathbf{b}} c_{\mathbf{b}} \cdot P_{b_1} \otimes P_{b_2} \otimes \cdots \otimes P_{b_x}$, where P_i is a Pauli operator. Thus resulting redefinition of B as $c_{\mathbf{b}} \cdot P_{b_1} \otimes \cdots \otimes P_{b_x}$. The effect of the environment will not change by rescaling, and new terms cannot be applied outside of the m sites at the boundary by the definition of geometrically local, thus $\partial H_{n+l} = \{I^{\otimes l} \otimes B_i \mid B_i \in \partial H_n\}$ □

As shown in Figure 4.4, for geometrically w-local Hamiltonian, applying the growing operator $q > w$ times on the Hamiltonian will saturate the boundary set. This motivates us to define the following concept.

Corollary 4 (Saturated Boundary Set for geometrically local Hamiltonian)**.** For a geometrically w-local Hamiltonian $\mathcal{H}$, the boundary set (∂H_n) will saturate for G_l as l increases.

Denote as $(\partial H_n)^{G_l}$. For $l > w$, $\left\|(\partial H)_l^G\right\|$ scales with the boundary size for geometrically local Hamiltonians as

$$\mathcal{O}(nL^{n-1}) \tag{4.39}$$

where $L = \max|dim_i|, i = 1, \cdots, n$, e.g in $1D$ it scales as $\mathcal{O}(1)$, and in 2D scales as $\mathcal{O}(2L)$.

For 1-D geometrically w-local Hamiltonian, with G_l always adding sites on one side of the original system, we have

$$\left\|(\partial H_n)^{G_l}\right\| = \begin{cases} l\|\partial H_n\| & l < w-1 \\ (w-1)\|\partial H_n\| & l \geq w-1 \end{cases} \tag{4.40}$$

where m is the number of species of connecting operators in G_l.

Corollary 5. the growing operator G_l^q defined on a geometrically w-local Hamiltonian can be rewritten as the following form

$$G_l^q[H_n] = H \otimes I^{\otimes ql} + \sum_{B_i \in (\partial H)^{G_l}} B_i \otimes R[B_i] + I^{\otimes n} \otimes K \tag{4.41}$$

In Section 4.2.2, we mention this is a property of geometrically local Hamiltonian, which can be generalized to rescalable local Hamiltonian with constant non-geometrically local terms. This can be shown by constructing a system with periodic boundary conditions, where the interaction term at the boundary is not geometrically local. Still, there are only a constant number of them. Thus, we have the following example

Example 4 (Saturated Boundary Set for Periodic Boundary)**.** Consider the $1D$ periodic boundary TFIM Hamiltonian of n sites

$$H_n = \sum_{i=1}^{n-1} Z_i Z_{i+1} + Z_n Z_1 + h \cdot \sum_i X_i \tag{4.42}$$

We can define G_1 as follows

$$\begin{aligned} G_1[H_n] = & H_n \otimes I + \underbrace{I^{\otimes n-1} \otimes Z \otimes Z}_{\text{connection with new site}} + \underbrace{I^{\otimes n} \otimes h \cdot X}_{\text{field term on the new site}} \\ & - \underbrace{Z \otimes I^{\otimes n-2} \otimes Z \otimes I}_{\text{old interaction at } n-1\text{-site boundary}} + \underbrace{Z \otimes I^{\otimes n-1} \otimes Z}_{\text{new interaction at } n\text{-site boundary}} \end{aligned} \tag{4.43}$$

applying G_1 twice, we have

$$G_1^2[H_n] = H_n \otimes I^{\otimes 2} + \underbrace{I^{\otimes n-1} \otimes Z \otimes Z \otimes I + I^{\otimes n} \otimes Z \otimes Z}_{\text{connection with new site}} + \underbrace{I^{\otimes n} \otimes h \cdot X + I^{\otimes n+1} \otimes h \cdot X}_{\text{field term on the new site}} - \underbrace{Z \otimes I^{\otimes n-2} \otimes Z \otimes I^{\otimes 2}}_{\text{old interaction at } n-1\text{-site boundary}} + \underbrace{Z \otimes I^{\otimes n} \otimes Z}_{\text{new interaction at } n\text{-site boundary}} \tag{4.44}$$

which still results in a saturated boundary set, equivalent to the saturated boundary set for open boundary $1D$ TFIM Hamiltonian (the geometrically local Hamiltonian) plus the operator at n-site periodic boundary $\{-Z \otimes I^{\otimes n-2} \otimes Z, Z \otimes I^{\otimes n-1}\}$. Note that, unlike geometrically local Hamiltonian, in this case, the result of connecting the operator on $R(Z \otimes I^{\otimes n-1})$ is changing as the system grows $I \otimes Z, I^{\otimes 2} \otimes Z, \cdots$. But this will not affect the set of B_i on the system's boundary.

Thus, for a more general case, we have the following

Corollary 6 (Saturated Boundary Set of Rescalable Local Hamiltonian)**.** For the rescalable w-local Hamiltonian $\mathcal{H}_n$, if there is only a constant number of non-geometrically local terms in ∂H_n, then applying G_l^q for arbitrary q times, will result in a saturated set $(\partial H_n)^{G_l}$.

4.4 Scaling Consistency Condition for Real Time Evolution

To prove the scaling consistency condition for real-time evolution, we need to find a series expansion for our time-evolved observables. We first introduce the following notation of commutators and anti-commutators.

Notation 1 (Adjoint). We denote the commutator for operator A, B as $ad_{A,-1}(B) = [A, B] = AB - BA$, and the anti-commutator as $ad_{A,+1}(B) = \{A, B\} = AB + BA$, and for $\sigma = \pm 1$, we denote $ad_{A,\sigma}(B) = AB + \sigma BA$.

Then we have the following lemma due to linearity of the commutator and anti-commutator.

Lemma 2 (Adjoint expansion)**.** We can expand the adjoint of the sum of operators $\sum_{i=1}^n A_i$ with an operator B as following:

$$ad_{\sum_{i=1}^n A_i,\sigma}(B) = \sum_{i=1}^n ad_{A,\sigma}(B) \tag{4.45}$$

Proof. This is due to the linearity of the commutator and anti-commutator. □

Furthermore, we can denote the composition of the adjoints as following

Notation 2 (Composition of Adjoints). We have the following notation for the adjoint of the composition of operators

$$ad_{A,\sigma}(ad_{B,\sigma}(C)) = ad_{A,\sigma}ad_{B,\sigma}(C) \tag{4.46}$$
$$ad^k_{A,\sigma}(B) = ad_{A,\sigma}(ad^{k-1}_{A,\sigma}(B)) \tag{4.47}$$

And we have the following lemma

Lemma 3 (Adjoint power)**.** We can expand the power of the adjoint of the sum of operators $\sum_{i=1}^n A_i$ with an operator B as following:

$$ad^k_{\sum_{i=1}^n A_i,\sigma}(B) = \sum_{k_1,\cdots,k_n} \prod_{i=1}^n ad_{A_{k_i},\sigma} = (\sum_{k=1}^n ad_{A_k,\sigma})^k \tag{4.48}$$

Proof.

$$ad^k_{\sum_{i=1}^n A_i,\sigma}(B) \tag{4.49}$$
$$= ad^{k-1}_{\sum_{i=1}^n A_i,\sigma}(ad_{\sum_{k_1=1}^n A_{k_1},\sigma}(B)) \tag{4.50}$$
$$= \sum_{k_1=1}^n ad^{k-1}_{\sum_{i=1}^n A_i,\sigma}(ad_{A_{k_1},\sigma}(B)) \tag{4.51}$$
$$= \sum_{k_1=1}^n \cdots \sum_{k_n=1}^n (\prod_{i=1}^k ad_{A_{k_i},\sigma})(B) \tag{4.52}$$

□

We can verify the correctness by checking for $k = 2$, $n = 2, \sigma = -1$, denote $ad_{A,-1} = ad_A$, we have

$$
\begin{aligned}
&(ad_{A_1} + ad_{A_2})^2 && (4.53)\\
&= ad_{A_1}^2 + ad_{A_2}^2 + ad_{A_1} ad_{A_2} + ad_{A_2} ad_{A_1} && (4.54)\\
&ad_{A_1+A_2}^2(B) && (4.55)\\
&= ad_{A_1+A_2}(ad_{A_1+A_2}(B)) && (4.56)\\
&= ad_{A_1+A_2}(ad_{A_1}(B) + ad_{A_2}(B)) && (4.57)\\
&= ad_{A_1+A_2}(ad_{A_1}(B)) + ad_{A_1+A_2}(ad_{A_2}(B)) && (4.58)\\
&= ad_{A_1}^2(B) + ad_{A_2} ad_{A_1}(B) + ad_{A_1} ad_{A_2}(B) + ad_{A_2}^2(B) && (4.59)
\end{aligned}
$$

Lemma 4 (Adjoint of Tensor Product)**.** The power of adjoint of tensor product of operators $ad_{A\otimes B,\sigma}^k$ can be expanded as following:

$$
\begin{aligned}
ad_{A\otimes B,\sigma}^k &= \frac{1}{2^k} \sum_{\sigma_1,\sigma_2,\cdots,\sigma_k \in \{+,-\}} (\prod_{i=1}^k ad_{A,\sigma_i}) \otimes (\prod_{i=1}^k ad_{B,\sigma\sigma_i}) \\
&= \frac{1}{2^k} (\sum_{\sigma_i \in \{+,-\}} ad_{A,\sigma_i} \otimes ad_{B,\sigma\sigma_i})^k
\end{aligned}
\tag{4.60}
$$

where $ad_A \otimes ad_B$ is defined as $ad_A(X) \otimes ad_B(Y) = (ad_A \otimes ad_B)(X \otimes Y)$

Proof. It can be checked that

$$
ad_{A\otimes C,\sigma}(B \otimes D) = \frac{1}{2} \sum_{\sigma=+,-} ad_{A,\sigma}(B) \otimes ad_{C,\sigma\sigma}(D) \tag{4.61}
$$

by iterating this equation,

$$
\begin{aligned}
&ad_{A\otimes C,\sigma}(ad_{A,\sigma_1}(B) \otimes ad_{C,\sigma\sigma_1}(D)) \\
&= \frac{1}{2} \sum_{\sigma_2=+,-} ad_{A,\sigma_2}(ad_{A,\sigma_1}(B)) \otimes ad_{C,\sigma\sigma_2}(ad_{C,\sigma\sigma_1}(D))
\end{aligned}
\tag{4.62}
$$

we can get

$$
\begin{aligned}
&ad_{A\otimes C,\sigma}^k(B \otimes D) \\
&= \frac{1}{2^k} \sum_{\sigma_1,\sigma_2,\cdots,\sigma_k} (\prod_{i=1}^k ad_{A,\sigma_i})(B) \otimes (\prod_{i=1}^k ad_{C,\sigma\sigma_i})(D) \\
&= \frac{1}{2^k} (\sum_{\sigma_i} ad_{A,\sigma_i} \otimes ad_{C,\sigma\sigma_i})^k (B \otimes D)
\end{aligned}
\tag{4.63}
$$

□

Lemma 5 (Lie-Trotter product formula [193])**.** For arbitrary operators $A, B \in \mathcal{H}(\mathbb{C}^d)^{\otimes n}$, we have

$$\exp[A+B] = \lim_{n\to\infty}(\exp[A/n]\exp[B/n])^n \tag{4.64}$$

where $\exp[A] = \sum_{k=0}^{\infty}\frac{A^k}{k!}$.

Lemma 6 (Baker-Campbell-Hausdorff formula [194])**.** For arbitrary operators $X, Y \in \mathcal{H}(\mathbb{C}^d)^{\otimes n}$, we have

$$\exp[X]Y\exp[-X] = \sum_{k=0}^{\infty}\frac{1}{k!}ad_{X,-}^k(Y) \tag{4.65}$$

Lemma 7 (Von Neumann's trace inequality [195])**.** if A, B are complex $n\times n$ matrices with singular values

$$\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_n \geq 0, \quad \beta_1 \geq \beta_2 \geq \cdots \geq \beta_n \geq 0 \tag{4.66}$$

then

$$|\mathrm{tr}(AB)| \leq \sum_{i=1}^{n}\alpha_i\beta_i \tag{4.67}$$

Equipped with the above lemmas, we can now prove an important series expansion for the time-evolved observable that splits the observable into system and environment parts. Although the following lemma can be seen as a variant of the Dyson series on operators. To the best of our knowledge, we did not find a similar lemma in the literature. Thus, we will introduce the proof of this lemma in the following.

Lemma 8 (Growing Dyson series)**.** Given observable defined as $O_S \otimes O_E$ where $O_S \in \mathcal{A}_n$ is the observable of the system and $O_E \in \mathcal{A}_{N-n}$ is the observable of the environment. Providing the rescalable local Hamiltonian $\mathcal{H}_n = \{H_n, \partial H_n, R_l\}$, denote the corresponding growing operator as G_l

$$G_l[H_n] = H_n \otimes I + \sum_{B_i\in\partial H_n} B_i \otimes R_l[B_i] + I^{\otimes n}\otimes K \tag{4.68}$$

and total evolution time as T, we can expand the time-evolved observable $O_S(T)\otimes O_E(T)$ as following:

$$\exp\{iTG_l[H_n]\}O_S\otimes O_E\exp\{-iTG_l[H_n]\} = \lim_{M\to\infty}\sum_k\frac{\delta^k}{2^k k!}(\sum_{m=0}^{M-1}\mathcal{T}_{B(m\delta)})^k(O_S(T)\otimes O_E(T)) \tag{4.69}$$

where $\delta = t/M$, $\mathcal{T}_{B(t)} = \sum_{i,\sigma} ad_{B_i(t),\sigma} \otimes ad_{R_l[B_i](t),-\sigma}$, and $\forall t_1 \leq t_2 \in \mathbb{R}$ we define $\mathcal{T}_{B(t_2)}\mathcal{T}_{B(t_1)} = \mathcal{T}_{B(t_1)}\mathcal{T}_{B(t_2)}$, thus the product of $\mathcal{T}$ is time-ordered. And

$$\begin{aligned} O_S(T) &= \exp\{itH\} O_S \exp\{-itH\}, \quad O_E(T) = \exp\{itK\} O_E \exp\{-itK\} \\ B_i(t) &= \exp\{itH_n\} B_i \exp\{-itH_n\}, \quad R[B_i](t) = \exp\{itK\} R[B_i] \exp\{-itK\} \end{aligned} \tag{4.70}$$

Proof. The proof uses previous lemmas to expand the operator onto system and environment parts, then simplify the series by reorganizing the summation. First by using Lemma 5, we divide our evolution into small time steps t/M

$$e^{itG_l[H_n]} O_S \otimes O_E e^{-itG_l[H_n]} = \lim_{M\to\infty} (e^{it/MG_l[H_n]})^M O_S \otimes O_E (e^{-it/MG_l[H_n]})^M \tag{4.71}$$

We can see this product as M steps of time evolution with time step $\delta = t/M$.

$$\lim_{M\to\infty} e^{i\delta G_l[H_n]} (e^{i\delta G_l[H_n]} \cdots (e^{i\delta G_l[H_n]} O_S \otimes O_E e^{-i\delta G_l[H_n]}) \cdots e^{-i\delta G_l[H_n]}) e^{-i\delta G_l[H_n]} \tag{4.72}$$

Because $\delta \to 0$, we can move the terms only depending on the system or environment onto the observables, leaving only the boundary terms in the time evolution.

$$e^{i\delta G_l[H_n]} O_S \otimes O_E e^{-i\delta G_l[H_n]} = e^{i\delta \sum_i B_i \otimes R_l[B_i]} O_S(\delta) \otimes O_E(\delta) e^{-i\delta \sum_i B_i \otimes R_l[B_i]} \tag{4.73}$$

to further expand the boundary terms, using Lemma 6 we have

$$= \sum_k \frac{(i\delta)^k}{k!} ad^k_{\sum_i B_i \otimes R_l[B_i],-}(O_S(\delta) \otimes O_E(\delta)) \tag{4.74}$$

and Lemma 4 we have

$$= \sum_k \frac{(i\delta)^k}{2^k k!} (\sum_{i,\sigma} ad_{B_i,\sigma} \otimes ad_{R_l[B_i],-\sigma})^k (O_S(\delta) \otimes O_E(\delta)) = \sum_k \frac{(i\delta)^k}{2^k k!} \mathcal{T}^k_{B(0)} (O_S(\delta) \otimes O_E(\delta)) \tag{4.75}$$

and because the product of $\mathcal{T}$ is time-ordered, we always do the multiplication in the order of time steps, making the product commutative. Now, if we apply $e^{i\delta G_l[H_n]} X e^{-i\delta G_l[H_n]}$ again, because e^{-itH_n} cancels e^{itH_n}, they can be merged into the time evolution of each separate system. Resulting in the following

$$\sum_{k_1,k_2} \frac{(i\delta)^{k_1+k_2}}{2^{k_1+k_2} k_1! k_2!} \mathcal{T}^{k_2}_{B(0)} \mathcal{T}^{k_1}_{B(\delta)} (O_L(2\delta) \otimes O_R(2\delta)) \tag{4.76}$$

Because the product of $\mathcal{T}$ is commutative, reorganizing the summation index as $k = k_1 + k_2$ we have

$$= \sum_k \frac{(i\delta)^k}{2^k k!} (\mathcal{T}_{B(0)} + \mathcal{T}_{B(\delta)})^k (O_L(2\delta) \otimes O_R(2\delta)) \tag{4.77}$$

By re-using Equation (4.77) iteratively, we reach the general form

$$\exp\{itG_l[H_n]\} O_S \otimes O_E \exp\{-itG_l[H_n]\} = \lim_{M\to\infty} \sum_k \frac{\delta^k}{2^k k!} (\sum_{m=0}^{M-1} \mathcal{T}_{B(m\delta)})^k (O_S(t) \otimes O_E(t)) \tag{4.78}$$

□

Lemma 8 is exactly the series expansion we are looking for. Before proceeding into the proof, we introduce the notion of multi-index for convenience.

Notation 3 (Multi-index). The multi-index sum and product are defined as the following

$$\begin{aligned} \sum_{\mathbf{i}} &= \sum_{i_1, i_2, \cdots, i_k} \\ \prod_{\mathbf{i}} A_i &= A_{i_1} A_{i_2} \cdots A_{i_k} \\ \sum_{\mathbf{i}} \prod_{\mathbf{i}} A_i &= \sum_{i_1, i_2, \cdots, i_k} A_{i_1} A_{i_2} \cdots A_{i_k} \end{aligned} \tag{4.79}$$

Now, we can check if summing up the components of the environment converges to a value.

Definition 11 (Time-Ordered Boundary Correlator)**.** Given a rescalable Hamiltonian $\mathcal{H}_n = \{H_n, \partial H_n, R_l\}$ and an observable O and an initial state ρ on the current scale, denote the corresponding growing operator as G_l, the total evolution time as T, we can define the k-th order Time-Ordered Boundary Correlator (TOBC) as following:

$$\langle \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(\mathcal{S}_n, T) \rangle = \langle \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(\rho, \mathcal{H}_n, O, T) \rangle = tr(\rho[\mathcal{T} \prod_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}} ad_{B_i(t),\sigma}] O(T)) \tag{4.80}$$

where $\mathcal{S}_n$ is the corresponding rescalable system, $B_i \in (\partial H_n)^{G_l}$, $t \in [0, T]$, $\sigma \in \{+1, -1\}$, and $\boldsymbol{i}, \boldsymbol{t}, \boldsymbol{\sigma}$ is a multi-index of size k. The product is time-ordered, i.e. $ad_{B_i(t_1),\sigma_1} ad_{B_j(t_2),\sigma_2} = ad_{B_j(t_2),\sigma_2} ad_{B_i(t_1),\sigma_1}$ for $t_1 > t_2$.

From a physics perspective, TOBC describes how the environment affects the system. Higher order TOBC corresponds to longer-time, longer-distance correlations. It is derived from the following theorem.

Theorem 4 (Real-time ϵ-scaling consistency)**.** *Given a w-local rescalable Hamiltonian* $\mathcal{H}_n = \{H_n, \partial H_n, R_k\}$ *that has a saturated boundary set* $(\partial H_n)^{G_l}$*. If* $\exists \epsilon > 0$ *for* $\rho = \rho_S \otimes \rho_E, O = O_S \otimes O_E$ *and* $\forall \boldsymbol{i}, \boldsymbol{t}, \boldsymbol{\sigma}$ *such that*

$$\left\| \langle \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(\mathcal{S}_n, T) \rangle - \left\langle \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(f^{\theta}[\mathcal{S}_n], T) \right\rangle \right\| \leq \epsilon \tag{4.81}$$

then for $N = n + kq$*, the error of expectation* $p_N[G_k^q[S_n]] = \left\langle \rho e^{itH_N} O e^{-itH_N} \right\rangle$ *is bounded by*

$$\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \epsilon C \exp\left\{ T \left\| (\partial H_n)^{G_l} \right\| C/2 \right\} \tag{4.82}$$

where C *is the maximum of* $\max\{\|R[B_i]\|_\infty \mid B_i \in \partial H_n\}$ *and* $\|O_R\|_\infty$*.*

Proof. Using Lemma 8 and Corollary 5 on G_l^q we have

$$\exp(itG_l^q[H_n]) O_S \otimes O_E \exp(-itG_l^q[H_n]) = \sum_k \frac{(i\delta)^k}{2^k k!} (\sum_{m=0}^{M-1} \mathcal{T}_{B(m\delta)})^k (O_S(T) \otimes O_E(T)) \tag{4.83}$$

where $B_i \in (\partial H_n)^{G_l}$ instead of ∂H_n, checking the k-th order, where the $\mathcal{T}$ on the right denotes the products are time-ordered, we have

$$(\sum_{m=0}^{M-1} \mathcal{T}_{B(m\delta)})^k = \mathcal{T}(\sum_{i,m,\sigma} ad_{B_i(m\delta),\sigma} \otimes ad_{R[B_i](m\delta),-\sigma})^k \tag{4.84}$$

we can expand the power of sum into the sum of tensor products on the system and environment

$$\sum_{\mathbf{i},\mathbf{m},\boldsymbol{\sigma}} \mathcal{T} \prod_{\mathbf{i},\mathbf{m},\boldsymbol{\sigma}} ad_{B_i(m\delta),\sigma} \otimes ad_{R[B_i](m\delta),-\sigma} \tag{4.85}$$

here, the multi-index notion is defined as the following

$$\sum_{\mathbf{i},\mathbf{m},\boldsymbol{\sigma}} \mathcal{T} \prod_{\mathbf{i},\mathbf{m},\boldsymbol{\sigma}} A_{i,m,\sigma} = \sum_{i_1,i_2,\cdots,i_k} \sum_{m_1,m_2,\cdots,m_k} \sum_{\sigma_1,\sigma_2,\cdots,\sigma_k} \mathcal{T} A_{i_1,m_1,\sigma_1} A_{i_2,m_2,\sigma_2} \cdots A_{i_k,m_k,\sigma_k} \tag{4.86}$$

applying the k-th order back to the observable $O_S(T) \otimes O_R(T)$, we obtained the observables on the system and environment

$$\sum_{\boldsymbol{i},\boldsymbol{m},\boldsymbol{\sigma}} [\mathcal{T} \prod_{\boldsymbol{i},\boldsymbol{m},\boldsymbol{\sigma}} ad_{B_i(m\delta),\sigma}(O_S(T))] \otimes [\mathcal{T} \prod_{\boldsymbol{i},\boldsymbol{m},\boldsymbol{\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))] \tag{4.87}$$

taking the expectation, the left-hand side of the tensor product is the k-th order TOBC, which we assume to be bounded by ϵ, and the right-hand side is the observable on the environment. Thus, we can write the expectation as

$$\begin{aligned}
&p_N[G_k^q[S_n]] = tr([\rho_E \otimes \rho_S] \exp(itG_k^q[H_n])[O_S \otimes O_E] \exp(-itG_k^q[H_n])) \\
&= \sum_k \frac{(i\delta)^k}{2^k k!} \sum_{\boldsymbol{i,m,\sigma}} tr[\rho_S \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{B_i(m\delta),\sigma}(O_S(T))] \cdot tr[\rho_E \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))] \\
&= \sum_k \frac{(i\delta)^k}{2^k k!} \sum_{\boldsymbol{i,m,\sigma}} \langle \chi_{\boldsymbol{i,m,\sigma}}(\mathcal{S}_n, T) \rangle \cdot tr[\rho_E \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))]
\end{aligned} \tag{4.88}$$

Next, because f^θ only applies to the operators in the system, leaving the environment untouched, we can write the error of expectation as

$$\begin{aligned}
&\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| = \\
&\left\| \sum_k \frac{(i\delta)^k}{2^k k!} \sum_{\boldsymbol{i,m,\sigma}} (\langle \chi_{\boldsymbol{i,t,\sigma}}(\mathcal{S}_n, T) \rangle - \left\langle \chi_{\boldsymbol{i,t,\sigma}}(f^\theta[\mathcal{S}_n], T) \right\rangle) \cdot tr[\rho_E \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))] \right\|
\end{aligned} \tag{4.89}$$

Using triangular inequality, we have

$$\begin{aligned}
&\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \\
&\sum_k \frac{\delta^k}{2^k k!} \left\| \sum_{\boldsymbol{i,m,\sigma}} (\langle \chi_{\boldsymbol{i,t,\sigma}}(\mathcal{S}_n, T) \rangle - \left\langle \chi_{\boldsymbol{i,t,\sigma}}(f^\theta[\mathcal{S}_n], T) \right\rangle) \cdot tr[\rho_E \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))] \right\|
\end{aligned} \tag{4.90}$$

Because we assume the TOBC is bounded by $\epsilon > 0$ and $\|a \cdot b\| = \|a\| \cdot \|b\|$, we have

$$\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \epsilon \sum_k \frac{\delta^k}{2^k k!} \left\| \sum_{\boldsymbol{i,m,\sigma}} tr[\rho_E \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))] \right\| \tag{4.91}$$

Notice that sum over all possible commutator and anti-commutator in $\sum_{\boldsymbol{i,m,\sigma}}$ recovers the product of operators in application order. Applying the triangular inequality again, we have

$$\left\| \sum_{\boldsymbol{i,m,\sigma}} tr[\rho_E \mathcal{T} \prod_{\boldsymbol{i,m,\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))] \right\| \leq \sum_{\mathbf{i,m}} \left\| tr[\rho_E \mathcal{T} (\prod_{\mathbf{i,m}} R[B_i](m\delta)) O_E(T)] \right\| \tag{4.92}$$

Using Lemma 7 on the trace, we have

$$\left\| tr[\rho_E \mathcal{T}(\prod_{\mathbf{i},\mathbf{m}} R[B_i](m\delta)) O_E(T)] \right\| \leq \sum_a \alpha_a \beta_a \tag{4.93}$$

where α_a is the eigenvalues of ρ_E and $\sum_a \alpha_a = 1$ by definition of density matrix, β_a is the eigenvalues of $\mathcal{T} \prod_{\boldsymbol{i},\boldsymbol{m},\boldsymbol{\sigma}} ad_{R[B_i](m\delta),-\sigma}(O_E(T))$, taking the maximum of β_a we have

$$\left\| tr[\rho_E \mathcal{T}(\prod_{\mathbf{i},\mathbf{m}} R[B_i](m\delta)) O_E(T)] \right\| \leq \beta_{\max} \sum_a \alpha_a = \beta_{\max} \tag{4.94}$$

$\beta_{\max}$ is equivalent to the operator 2-norm of the operator, thus

$$\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \epsilon \sum_k \frac{\delta^k}{2^k k!} \sum_{\mathbf{i},\mathbf{m}} \left\| \mathcal{T}(\prod_{\mathbf{i},\mathbf{m}} R[B_i](m\delta)) O_E(T) \right\|_{\text{op}} \tag{4.95}$$

Next, we can replace the variable $\delta \to \frac{\delta}{2}$ and rescale the environment Hamiltonian $K \to 2K$, thus $R[B_i](m\delta) \to R[B_i](m\delta/2)$ and $O_E(T) \to O_E(T/2)$, this allows us to remove the factor of 2^k in the summation so that we can find the summation later, now we have

$$\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \epsilon \sum_k \frac{(\delta/2)^k}{k!} \sum_{\mathbf{i},\mathbf{m}} \left\| \mathcal{T}(\prod_{\mathbf{i},\mathbf{m}} R[B_i](m\delta)) O_E(T) \right\|_{\text{op}} \tag{4.96}$$

note that the Hamiltonian for $R[B_i](m\delta/2)$ and $O_E(T/2)$ here is $2K$ instead of K. Next, we can use the sub-multiplicative of norm $\|AB\| \leq \|A\|\|B\|$ This allows us further to break the product into the norm of each operator

$$\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \epsilon \sum_k \frac{(\delta/2)^k}{k!} \sum_{\mathbf{i},\mathbf{m}} \mathcal{T}(\prod_{\mathbf{i},\mathbf{m}} \|R[B_i](m\delta/2)\|_{\text{op}}) \|O_E(T/2)\|_{\text{op}} \tag{4.97}$$

Because time evolution is unitary, the norm of the operator is preserved, thus $\|R[B_i](m\delta/2)\|_{\text{op}} = \|R[B_i]\|_{\text{op}}$, and $\|O_E(T/2)\|_{\text{op}} = \|O_E(T)\|_{\text{op}}$, thus if we have $C = \max\{\|R[B_i]\|_{\text{op}} \mid B_i \in \partial H_n\}$ and $\|O_R\|_{\text{op}}$, we have

$$\left\| p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]] \right\| \leq \epsilon \sum_k \frac{(\delta/2)^k}{k!} \sum_{\mathbf{i},\mathbf{m}} C^{k+1} \tag{4.98}$$

Now we can sum over the multi-index $\mathbf{i}, \mathbf{m}$, the sum of $\mathbf{m}$ is M^k and the sum of $\mathbf{i}$ is $|\partial H_n|^k$, thus we have

$$\left\|p_N[G_k^q[S_n]] - p_N[(G_k^{q-1} \circ D_k)[S_n]]\right\| \leq \epsilon C \sum_k \frac{(\delta M/2)^k}{k!}(|\partial H_n|C)^k = \epsilon C \exp\left\{T\left\|(\partial H_n)^G\right\|C/2\right\} \tag{4.99}$$

□

4.5 OLRG Algorithms

We have successfully relaxed the linearity constraint on the functions $f^\theta_{n_q}, n_q = n, n+l \cdots, N-l$ for a specific observable O_N at N-site system. Since $f^\theta_{n_q}$ can now be an arbitrary operator map, it can act as the map between operator matrices, as well as the map between operator expressions. This allows parameterized $f^\theta_{n_q}$ or parameterized output $f^\theta_{n_q}[X]$ when the output is an expression. This leads to the classical and quantum algorithms we introduce in this section.

One can then search for the optimal parameters for $f^\theta_{n_q}$ or the output $f^\theta_{n_q}[X]$. For special $f^\theta_{n_q}$ or $f^\theta_{n_q}[X]$, like the linear map in NRG and DMRG, the optimal point can be identified directly. In the general case, we employ gradient-based optimization [196] to search for optimal parameters θ. The gradient can be obtained using modern differentiable programming frameworks [7, 10, 13, 90, 197–201] and their automatic differentiation algorithms [4, 12, 121, 202, 203] that work not only on classical computers but also on quantum devices. This leads to the following general variational algorithm (Algorithm 1) that starts with a small system and iteratively enlarges the system until it reaches the target system size (the blue blocks in Figure 4.3).

Algorithm 1 OLRG Algorithm

1. **Input**: a small system of n sites with the set of relevant operators S_n, and a classical or quantum solver.
2. Applying the operator map f_n^θ to the set of relevant operators S_n to obtain the set of virtual operators $f_n^\theta(S_n)$.
3. Sampling a batch of the index $\boldsymbol{i,t,\sigma}$ for the TOBCs $\chi_{\boldsymbol{i,t,\sigma}}$.
4. Evaluate the value $\chi_{\boldsymbol{i,t,\sigma}}(S_n, T)$ by calling the solver.
5. Evaluate the value $\chi_{\boldsymbol{i,t,\sigma}}(f_n^\theta(S_n), T)$ by calling the solver.
6. Evaluate the average error of the sampled TOBCs $\mathcal{L}_n = \sum_{\boldsymbol{i,t,\sigma}} \left\| \chi_{\boldsymbol{i,t,\sigma}}(S_n, T) - \chi_{\boldsymbol{i,t,\sigma}}(f_n^\theta(S_n), T) \right\|$.
7. Enlarge the set of virtual operators and use it as the $n+l$-site relevant operators: $S_{n+l} = G_l(f_n^\theta(S_n))$.
8. Reiterate from step 1, accumulating the loss function $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_n$ until reach target system size N.
9. Optimize the loss function concerning the parameters of f_n^θ. Compute the update Δ of the parameters θ by calling an optimizer, e.g ADAM [196]. Applying the update to parameters $\theta \leftarrow \theta + \Delta$.
10. Repeat the above steps until the loss function converges.

The OLRG algorithm is a general variational algorithm that can be applied to both classical and quantum systems. Before introducing more details about the operator map f_n^θ for the classical and quantum cases, to illustrate the algorithm further, we will go through a concrete example of the algorithm in the context of calculating the real-time evolution of the two-point correlation function $\langle Z_1 Z_2 \rangle_T$ in a $1D$ TFIM model. Starting from a 2-site system, the relevant operators are $S_2 = \{H_2, B_2 = IZ, \rho_2 = |00\rangle\langle 00|, O_2 = ZZ\}$. Applying our operator map f_2^θ we have $f_2^\theta[S_2] = \{f_2^\theta[H_2], f_2^\theta[IZ], f_2^\theta[|00\rangle\langle 00|], f_2^\theta[ZZ]\}$, then we can sample a batch of indices $\boldsymbol{i,t,\sigma}$ with batch size b, evaluate the TOBCs by solving the Heisenberg equation of IZ, ZZ for the 2-site system. One must save the checkpoints at $\boldsymbol{t}$

for the boundary operator IZ. This allows us to calculate the loss function

$$\mathcal{L}_2 = \frac{1}{b} \sum_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}} \left\| \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(S_2, T) - \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(f_2^\theta(S_2), T) \right\|. \tag{4.100}$$

This batch of sampled TOBCs is usually referred as the mini-batch in deep learning. The size of this batch is a crucial hyperparameter controlling the variance of the gradient and thus impacts the optimizer's behavior. When $b = 1$, the algorithm is called Stochastic Gradient Descent (SGD), and when b is all the TOBCs, the algorithm is plain gradient descent. Then, assuming we are taking the simplest growing strategy that grows the system by 1 site at each step, we can grow the relevant operators using G_1, resulting in a new set of relevant operators $S_3 = \{H_3, B_3, \rho_3, O_3\}$ defined as,

$$\begin{aligned} H_3 &= G_1[f_2^\theta[H_2]] \\ &= f_2^\theta[H_2] \otimes I + f_2^\theta[B_2] \otimes Z + I \otimes (h \cdot X) \\ B_3 &= I \otimes Z \\ \rho_3 &= f_2^\theta[|00\rangle \langle 00|] \otimes |0\rangle \langle 0| \\ O_3 &= f_2^\theta[O_2] \otimes I. \end{aligned} \tag{4.101}$$

Here, we assume $f_2^\theta[I] = I$, thus applying I without calculating $f_2^\theta[I]$ in B_3. I automatically adjusts to the size of the corresponding system, e.g. $B_3 = I \otimes Z$; I should share the same size as $f_2^\theta[ZZ]$, and in O_3, I should share the same size as $|0\rangle \langle 0|$. Then we can repeat the previous steps to obtain $\mathcal{L}_3$, and S_4 until we get $\mathcal{L}_{10}$ and S_{10}. We then calculate the total loss as $\mathcal{L} = \mathcal{L}_2 + \cdots + \mathcal{L}_{10}$. Finally, we can differentiate the loss function $\mathcal{L}$ with respect to the parameters θ and update the parameters θ using a gradient-based optimizer. This is called *one epoch* of the algorithm. We can repeat the above steps until the loss function converges.

However, this training process directly optimizes towards a target observable at time T. If one is interested in the time points $0 \leq t_1 \leq t_2 \leq \cdots \leq T$, a transfer learning [204] strategy can be employed. We can optimize the parameters θ at the first time point t_1 resulting in the optimized parameters θ_{t_1}. Then, we use θ_{t_1} as the initial point for optimizing t_2, and so on. This method is similar to the strategy employed in other variational algorithms, such as MPS TDVP and time-dependent VMC [205–208]. However, in our setup, the parameters θ do not always represent an explicit state. In other variational algorithms, the states are passed through to the next time point explicitly by evolving in the parameter space. In our setup, the states are implicitly passed through as the parameters θ. In special cases, an explicit state can be constructed from $f_{n_q}^\theta$, which results in a

similar algorithm as MPS TDVP. This therefore leads to potential improvements of the MPS TDVP algorithm to address long-time correlations. The detailed relation between this transfer learning strategy and the MPS TDVP algorithm is discussed in Section 4.9.5.

Based on how one defines $f^{\theta}_{n_q}$ and the representation of operators, the general algorithm can find different applications. In the following, we will introduce two specific algorithms for the classical and quantum case. For the classical case, we will use $f^{\theta}_{n_q}$ as a parameterized OMM. For the quantum case, we will use $f^{\theta}_{n_q}$ as a map from the problem Hamiltonian expression to the device Hamiltonian expression with parameters, namely HEM.

4.5.1 Classical Algorithm: Operator Matrix Map

The simulation challenge is more pronounced in real-time dynamics on conventional computers than in ground state, where no efficient classical algorithm is known for simulating the general real-time evolution of a quantum system. This further motivates the development of previous variational frameworks for real-time dynamics by employing a variational ansatz to model the system's state and subsequently evolving this ansatz over time by optimizing the variational parameters through the TDVP [205–207]. Born from the development of DMRG, the MPS TDVP is the most successful strategy for solving $1D$ many-body physics. A holy grail of the field is to find an algorithm that performs as well in $D > 1$, with contenders like Tensor Network State (TNS) [151–164] and neural network states (NNS) [208–222] continually making progress.

Our approach mirrors the workflows of NRG and DMRG but relaxes the linearity constraint on the operator map $f^{\theta}_{n_q}$, allowing for a more expressive operator map. We also propose a loss function that directly minimizes the error of the target property, thus allowing us to use the same workflow for real-time dynamics. This is achieved by optimizing the error of the TOBCs, as detailed in Section 4.2.2.

In the same spirit as DMRG, we design $f^{\theta}_{n_q}$ as a parameterized compression function $\mathrm{OMM}^{\theta}_{n_q}[X]$ of the operator matrices X and parameters θ. However, if $\mathrm{OMM}^{\theta}_{n_q}[X]$ is a dense linear function with a fixed size, the operator map is equivalent to an MPS, thus providing no advantage in expressiveness over MPS. From a physics perspective, approximating a larger pure system using a smaller pure system is not always possible. On the other hand, from the expressiveness perspective, the sum of matrix product states requires exponentially more parameters to represent the same state, despite that in certain cases when the tensors contain more structure, one can further compress the MPS via singular value decomposition [155]. Both suggest that letting $\mathrm{OMM}^{\theta}_{n_q}[X]$ be an ensemble of linear maps, which creates an ensemble of small pure systems, will be more expressive.

This shares the idea of using an ensemble of MPS generated by a recurrent neural network to represent the wave function in VMC [223]. In summary, instead of generating a single set of relevant operators from input S_n, we will generate an ensemble of relevant operators sampled by a probability based on the input S_n. Starting from z copies of the pure system, we can sample a single set of relevant operators from each copy and then forward them to the next step. This allows us to sample a chain of ensemble systems while growing the system size. For example, in our previous $1D$ TFIM example, we can start with 10 copies of the 2-site system S_2, then applying OMM_2^θ to each copy will sample a corresponding $\text{OMM}_2^\theta[S_2]$. This results in 10 sets of relevant operators $\text{OMM}_2^\theta[S_2]$ based on a probability distribution defined by OMM_2^θ. The loss function $\mathcal{L}_2$ is instead evaluated as the average the sampled index batch b of these 10 systems

$$\mathcal{L}_2 = \frac{1}{10b} \sum_{S_2} \sum_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}} \left\| \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(S_2, T) - \chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(\text{OMM}_2^\theta[S_2], T) \right\|. \tag{4.102}$$

Then, we can apply G_1 to the 10 sets of relevant operators to obtain the ensemble of 3 sites. Other steps stay the same as the general algorithm.

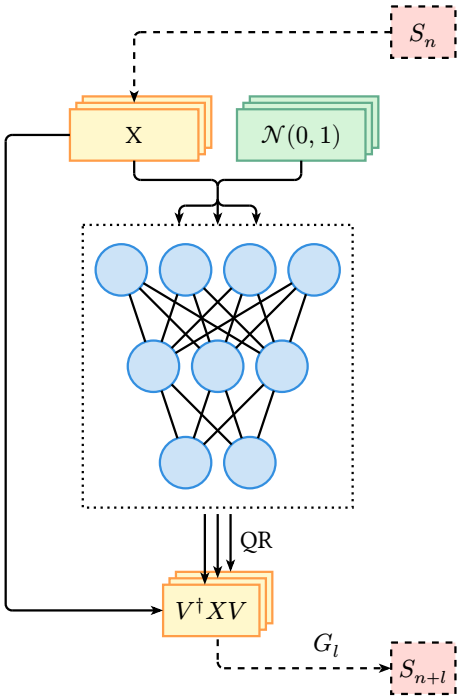

Figure 4.6: Illustration of neural OMM. X is a batch of input relevant operators, QR is the QR decomposition, $V^\dagger XV$ is a batch of output relevant operators by applying the batch of isometric matrices onto X. G_l is the growing operator, S_n is the input set of relevant operators and S_{n+l} is the output set of relevant operators

Treating $f^\theta_{n_q}$ as a compression function from an input operator matrix to an output operator matrix aligns well with the idea of generative models in deep learning, where the model generates a set of outputs from a given input and a noise, which models a conditional probability distribution. For readers familiar with computer vision, this problem is similar to an image compression, generation, or manipulation problem, where we generate a new image based on an input image. Under this context, the linear map in NRG and DMRG can be seen as a similar method of principal component analysis (PCA) for image compression [224]. More modern image generation in deep learning utilize more powerful generative models including Generative Adversarial Networks (GANs) [185], Variational AutoEncoders (VAEs) [225], normalizing flows [226, 227] and diffusion models [228, 229].

In our demonstration, For simplicity, we use the same neural network for each step of the OLRG, thus $\text{OMM}^\theta_{n_q} = \text{OMM}^\theta$. This requires the compression function always reduce

the size from $2^{n+l} \times 2^{n+l}$ to $2^n \times 2^n$ to match the input size for next OLRG step. As depicted in Figure 4.6, we employ the simplest toy neural network architecture used in GAN [185] that takes the operator matrix X and a noise vector $\boldsymbol{z}$ sampled from Gaussian distribution $\mathcal{N}(0,1)$ as input and generates an isometric matrix V as output. The isometric matrix V then applies to the operator matrix X to generate the transformed operator. This guarantees the function does not change I and the trace of the operator. Thus, it may have better numerical stability. In the following, we denote this operator map as $\mathrm{OMM}^{\theta}(X, \boldsymbol{z})$. The neural network part of the operator map is a feed-forward neural network (FFNN) using *ReLu* [230, 231] activation, each layer $\mathrm{layer}_i(\mathbf{x})$ defined as following

$$\mathrm{layer}_i(\mathbf{x}) = \mathrm{ReLU}(\mathbf{W}_i\mathbf{x} + \mathbf{b}_i), \tag{4.103}$$

where $\mathbf{W}_i$ and $\mathbf{b}_i$ are the weight matrix and bias vector of the i-th layer. The neural network's input is the operator matrix reshaped into a vector concatenated with the noise vector sampled from the Gaussian distribution. The neural network's output is reshaped into a square matrix and then performs QR decomposition to generate an isometric matrix V. Then V is applied to the input operator as $V^\dagger X V$.

In evaluating the loss function, the exact solver for solving TOBCs is an ODE solver. Thus, because the same $\mathrm{OMM}^{\theta}_{n_q}$ is shared as OMM^{θ} between OLRG steps D_l, the automatic differentiation needs to go through an ODE solver. Practically, this differentiation is typically achieved using the adjoint method, as detailed in various sources [14, 121, 232–234]. If $\mathrm{OMM}^{\theta}_{n_q}$ is not shared then only trivial linear algebra rules are needed for automatic differentiation.

We opted for a product state as the initial state, primarily due to the clear and well-defined nature of the growing operator in this context. This decision was influenced by the straightforward representation of a $n+k$-site product state as a composition of smaller system product states. For instance, a $n+k$-site zero state can be written as the following composition of a smaller system and thus defines its growing operator:

$$G_k(\underbrace{|0\cdots0\rangle\langle0\cdots0|}_{n \text{ sites}}) = \underbrace{|0\cdots0\rangle\langle0\cdots0|}_{n \text{ sites}} \otimes \underbrace{|0\cdots0\rangle\langle0\cdots0|}_{k \text{ sites}}. \tag{4.104}$$

It is unclear how to write a $n+l$-site state for a non-trivial state as the composition of smaller system states. Intuitively, MPS might be suitable for constructing such a formalism.

4.5.2 Quantum Algorithm: Hamiltonian Expression Map

An alternative approach to simulate the real-time dynamics of quantum many-body systems is to use a quantum computer. The development of quantum simulation [31, 235–237]

demonstrated that a quantum computer can efficiently simulate the real-time evolution of a quantum system. While algorithms based on Hamiltonian simulation [34–36, 238–241] have been proposed with rigorous bounds and polynomial complexity, the resource requirement [242] of these algorithms is beyond the capabilities of NISQ computers [65]. For example, the requirement of circuit depth and noise level are still beyond the capabilities of current devices [40, 104, 105, 182, 243]. As a result, heuristic algorithms such as variational quantum algorithms VQA) [37, 39, 66, 71, 72, 244–262] have been proposed for near-term devices. Notably, digital, analog, and logical resources typically coexist in near-term devices. Evidence in digital-analog quantum algorithms (DAQA) [263–265] show the potential advantages of using the entire device capabilities. Yet, achieving practical quantum advantage remains an open problem for these heuristic algorithms. These challenges motivate us to search for an alternative framework that can inherit the advantages of the above frameworks and potentially lead to different perspectives on the simulation problem.

In our quantum algorithm, because one can utilize real quantum dynamics, the storage complexity is no longer a concern. Instead, the main objective is to translate the problem of Hamiltonian dynamics into the dynamics of the quantum device. This involves finding the appropriate control parameters of the device Hamiltonian that can closely replicate the dynamics of the problem Hamiltonian. Rather than viewing $f_{n_q}^{\theta}$ as an Operator Matrix Map, $f_{n_q}^{\theta} = \mathrm{HEM}_{n_q}$ now maps the input expression of a Hamiltonian into device Hamiltonian expression at each system size $n_q = n, n+l, \cdots, N$, leaving other operators untouched. The expressions are parameterized by control parameters in the device pulse sequence. The process begins with the relevant set of operators for n-site problem $S_n = \{H_n, B_n^i, \rho_n, O_n\}$, applying HEM_n, we have $\mathrm{HEM}_n[S_n] = \{H_n^{\mathrm{dev}}(\theta_n, t_n), B_n^i, \rho_n, O_n\}$. $H_n^{\mathrm{dev}}(\theta_n, t_n)$ is the device Hamiltonian with control parameters θ_n and an input time t_n. Thus the effect of HEM_n is swapping the operator expression from H_n to $H_n^{\mathrm{dev}}(\theta_n, t_n)$. Then we can sample a batch of indices $\boldsymbol{i}, \boldsymbol{t}, \boldsymbol{\sigma}$ with batch size b, evaluate the TOBCs by running a classical solver for the problem Hamiltonian and the device Hamiltonian to compute the first loss function $\mathcal{L}_n$. Next, applying the growing operator G_l on $\mathrm{HEM}_n[S_n]$ result in $S_{n+l} = \{H_{n+l}, B_{n+l}, \rho_{n+l}, O_{n+l}\}$, where $B_{n+l}, \rho_{n+l}, O_{n+l}$ stays the same as problem system, and H_{n+l} is defined by following,

$$\begin{aligned} H_{n+l} &= G_l[H_n^{\mathrm{dev}}(\theta_n, t_n)] \\ &= H_n^{\mathrm{dev}}(\theta_n, t_n) \otimes I + \sum_i B_n^i \otimes R_l(B_n^i). \end{aligned} \tag{4.105}$$

From the second step, we can evaluate the TOBCs for the dynamics described by H_{n+l} using the quantum device. If $l \ll n$, then a large component of the dynamics is governed by the device Hamiltonian. We can then use a product formula, such as trotterization to

simulate the dynamics of H_{n+l} using the quantum circuit depicted in Figure 4.7. Each trotter step results in the following unitary

$$\begin{aligned} &\exp\Big(-i\delta G_l[H_n^{\text{dev}}(\theta_n, t_n)]\Big) \\ &= [\exp\Big(-i\delta H_n^{\text{dev}}(\theta_n, t_n)\Big) \otimes I] \cdot \prod_i \exp\Big(-i\delta B_n^i \otimes R_l(B_n^i)\Big). \end{aligned} \tag{4.106}$$

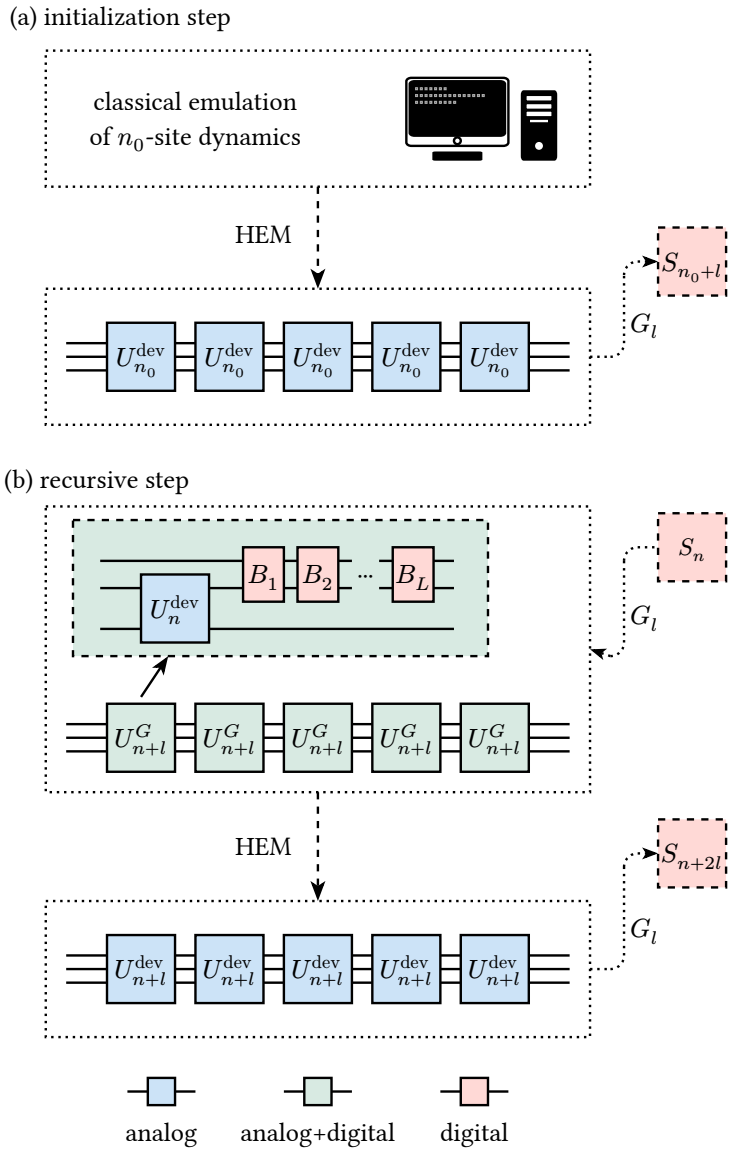

Figure 4.7: Illustration of HEM. (a) In the initialization step, HEM maps the emulation of n_0-site problem Hamiltonian dynamics into the n_0-site device Hamiltonian dynamics. Then the device dynamics is used to build grown system S_{n_0+l}, forwarding to recursive steps; (b) In recursive steps, HEM maps dynamics $U^G_{n+l} = \exp\left[-itG_l[H^{\text{dev}}_n]\right]$ to the device Hamiltonian dynamics $U^{\text{dev}}_{n+l} = \exp\left[H^{\text{dev}}_{n+l}\right]$. B_i are the w-qubit digital gates, $L = \left\|(\partial H_n)^{G_l}\right\|$ is the size of saturated boundary. U^{dev}_n is the dynamics of n-site device Hamiltonian.

If the problem system is w-local, then $B_n^i \otimes R_l[B_n^i]$ only applies on w qubits. Thus, the circuit only requires w-qubit high-quality gates at the boundary of the n-site system. Next, we can evaluate the TOBCs for the dynamics described by $G_l[H_n^{\text{dev}}(\theta_n, t_n)]$ and $H_{n+l}^{\text{dev}}(\theta_{n+l}, t_{n+l})$ to obtain the next loss function $\mathcal{L}_{n+l}$. Other steps stay the same as the general algorithm. In summary, the quantum device here plays the role of an exact solver, and HEM_{n_q} generates the parameterized device Hamiltonian expressions. The control parameters in the device Hamiltonian expressions are optimized to minimize the error of the target property.

Like previous, we explain this algorithm by simulating the real-time dynamics of two-point correlation function $\langle Z_1 Z_2 \rangle_T$ in $1D$ TFIM using a Rydberg atom device, as described in recent experimental demonstrations [104, 105]. The 2-level Rydberg Hamiltonian is defined as follows

$$\begin{aligned} &H_n^{\text{ryd}}(\theta_n, t_n) \\ &= \sum_{\langle i,j \rangle} V_{ij}^{\theta_n} n_i n_j + \Omega^{\theta_n}(t_n) \sum_i X_i - \Delta^{\theta_n}(t_n) \sum_i n_i \end{aligned} \tag{4.107}$$

where V_{ij}^{θ} denote the strength of interaction, and $\Omega^{\theta}(t)$ and $\Delta^{\theta}(t)$ are two time-dependent functions, commonly called pulse functions. Note that the 2-level Rydberg Hamiltonian is not universal and thus is restricted in the expressiveness of representing arbitrary dynamics. We begin with the set of relevant operators for the 2-site system $S_2 = \{H_2, B_2 = IZ, \rho_2 = |00\rangle\langle 00|, O_2 = ZZ\}$, where H_2 is the 2-site TFIM Hamiltonian with $h = 1.0$. We use two FFNNs as the parameterized pulse function $\Omega^{\theta}(t)$ and $\Delta^{\theta}(t)$ and another FFNN representing the strength $V_{i,j}^{\theta}$ so that the effective duration of the device can be controlled. Thus our HEM_{n_q} now maps a given Hamiltonian to a $1D$ Rydberg Hamiltonian with the pulse functions $V_{i,j}^{\theta}$, $\Omega^{\theta}(t)$ and $\Delta^{\theta}(t)$. We first run a classical ODE solver to evaluate the TOBCs. This results in the same loss function in *Equation* (4.100), where $\chi_{\boldsymbol{i},\boldsymbol{t},\boldsymbol{\sigma}}(\text{HEM}_2(S_2), T)$ is the TOBCs for the 2-site parameterized Rydberg Hamiltonian. Next, we apply the growing operator G_1 to the 2-site parameterized Rydberg Hamiltonian to obtain the set of relevant operators for the 3-site system $S_3 = \{H_3, B_3, \rho_3, O_3\}$, where,

$$\begin{aligned} H_3 &= G_1[H_2^{\text{ryd}}(\theta_2, t_2)] \\ &= H_2^{\text{ryd}}(\theta_2, t_2) \otimes I + B_2 \otimes Z + I \otimes (h \cdot X) \\ B_3 &= I^{\otimes 2} \otimes Z \\ \rho_3 &= |000\rangle\langle 000| \\ O_3 &= ZZ \otimes I. \end{aligned} \tag{4.108}$$

Next, we apply HEM_3 on S_3 result in

$$\text{HEM}_3[S_3] = \{H_3^{\text{ryd}}(\theta_3, t_3), B_3, \rho_3, O_3\}. \tag{4.109}$$

The TOBCs for $\mathrm{HEM}_3[S_3]$ can be evaluated on a standard analog Rydberg atom device. We use the circuit in Figure 4.7 to evaluate the TOBCs of $H_3 = G_1[H_2^{\mathrm{ryd}}(\theta_2, t_2)]$. Each trotter step results in the following unitary

$$\begin{aligned}\exp\Big(-i\delta G_1[H_2^{\mathrm{ryd}}(\theta_2,t_2)]\Big) =&[\exp\Big(-i\delta H_2^{\mathrm{ryd}}(\theta_2,t_2)\Big)\otimes I]\cdot\\ &[I\otimes\exp(-i\delta Z\otimes Z)]\cdot\\ &[I^{\otimes 2}\otimes\exp(-i\delta h\cdot X)].\end{aligned} \tag{4.110}$$

Denote $U_G(t) = \exp\Big(-itG_1[H_2^{\mathrm{ryd}}(\theta_2,t_2)]\Big)$. We can write down the 1st order TOBC $\Big\langle\chi_{\boldsymbol{i},\boldsymbol{t},\{-\}}\Big\rangle$ as an example of the full circuit

$$\begin{aligned}&\Big\langle\chi_{\boldsymbol{i},\boldsymbol{t},\{-\}}\Big\rangle(S_3,T)\\ &=\mathrm{tr}\Big[\rho_3 U_G(t_2)^\dagger Z_3 U_G(T-t_2)^\dagger Z_1Z_2U_G(T)\Big]-\\ &\mathrm{tr}\Big[\rho_3 U_G(T)^\dagger Z_1Z_2U_G(T-t_2)Z_3U_G(t_2)\Big].\end{aligned} \tag{4.111}$$

After obtaining the TOBCs for S_3, we can calculate the loss function $\mathcal{L}_3$ and repeat the steps until we reach our target size. Like the general algorithm, we optimize the total loss function until convergence to search for the optimal pulse functions. The HEM-based OLRG is not limited to the product state because the operator map HEM_{n_q} does not alter the state operator. Thus, we do not need an explicit growing operator for the state operator.

Through HEM, OLRG allows us to leverage large analog and a few digital resources. Moreover, by adjusting the l in the growing operator, we can trade off the digital-analog resources. For example, if we grow the system by 1 site at each step, the algorithm is closer to a VQA, and if we grow the system by $1 \ll l$ sites at each step, the algorithm is closer to a product formula. Lastly, OLRG also bridges classical algorithms for simulating dynamics in the first step. Instead of competing with classical algorithms, HEM-based OLRG allows us to use the results from classical algorithms in n-site system as a starting point and then use the quantum device to grow into N-site system where $n < N$. Thus, improvements in the first-step classical simulation will improve HEM-based OLRG, allowing both communities to push the limits of quantum dynamics simulation together.

4.5.3 Error and Resource Estimation

Theoretically, the source of error in our framework originates from the estimation and optimization of the e2e-style loss function, as well as the expressiveness of operator maps.

This aligns with e2e learning. However, it is important to note that the error bound presented in Theorem 2 is not a tight bound. In practice, it intends to predict a large error. Combined with the truncation error arising from estimating the series expansion, the actual error estimation is often inaccurate in our current algorithm. The primary purpose of the theorem is to direct us toward defining a systematically improvable loss function rather than to provide an exact error estimation. A more precise error estimation requires finding a tighter bound in Theorem 2. We discuss intuitions and potential methods to improve this in Section 4.9.1.

In the OMM-based OLRG, denote the time complexity of the evaluation of OMM as W_{OMM^θ} and the time complexity of the small-system solver as Q, for L growing steps, the time complexity of evaluating the loss function is $O(L(W_{\mathrm{OMM}^\theta}+2Q))$. The time complexity of evaluating the derivative of the loss function depends on whether OMM^θ is shared between different scales. We denote the time complexity of differentiating the operator map evaluation as W'_{OMM^θ} and the adjoint method as Q'. Heuristically, $Q' = 2Q$ [14]. Thus, if OMM^θ is shared, the time complexity of evaluating the derivative of the loss function is $O(L(W'_{\mathrm{OMM}^\theta} + 2Q')) \approx O(L(W'_{\mathrm{OMM}^\theta} + 4Q))$. However, if OMM^θ is not shared, then there is no need to differentiate through the exact solver. The time complexity becomes $O(LW'_{\mathrm{OMM}^\theta})$. In terms of storage complexity, aside from the batch and sampling size, we denote the storage complexity of evaluating OMM^θ as S_{OMM^θ}. If OMM^θ is shared, since the pure system ODE is reversible, the best algorithm solving the derivative has a constant overhead by using reversibility [4, 14]. We denote this constant overhead as C_Q. The total storage complexity is only $O(S_{\mathrm{OMM}^\theta} + C_Q L)$. However, if OMM^θ is not shared, the storage complexity becomes $O(LS_{\mathrm{OMM}^\theta})$. Thus, in the OMM-based OLRG, one can trade storage for time complexity and vice versa by deciding how many OMM^θ are shared. For simplicity, we do not discuss the complexity of estimating the k-th order TOBCs in the OMM-based OLRG here because it only requires $O(k)$ times matrix multiplication in the small system. When considering the batch and sampling size, they create a constant factor over the time and storage complexity. It is worth noting that the overhead created by batch and sampling size can be easily reduced by parallelization and distributed storage due to their simplicity. This fits well into the modern processor architecture designed for single-program-multiple-data SPMD) [61, 266].

In HEM-based OLRG, the classical computation components are relatively cheaper. Thus, we focus on discussing the cost of quantum operations. Because our algorithm involves analog circuits, we use the effective pulse duration (i.e., the scaling of the pulse duration to execute the circuit) as the measure instead of using circuit depth. We assume that the optimization only creates a constant prefactor in terms of the pulse duration for

simulating $G_l(H_n)$ as $C_\theta \tau(\left\|(\partial H_n)^{G_l}\right\|)$. C_θ is the overhead caused by variational optimized pulse sequence. $\tau(\left\|(\partial H_n)^{G_l}\right\|)$ is the overhead caused by product formula, e.g. for 1st-order trotterization $\tau(\left\|(\partial H_n)^{G_l}\right\|) = \left\|(\partial H_n)^{G_l}\right\|$. And there are $M \gg 1$ checkpoints and total time T, for evaluating one k-th order TOBC using 1st-order trotterization, the average effective pulse duration is $O(C_\theta\left\|(\partial H_n)^{G_l}\right\|kT)$ and the worst effective pulse duration is $O(2C_\theta\left\|(\partial H_n)^{G_l}\right\|kT)$.

Denote the checkpoints for k-th order TOBC as $t_1, \cdots, t_k$, and assuming the constant overhead in product formula for implementing $O(\left\|(\partial H_n)^{G_l}\right\|)$ number of w-qubit gates result in total. The total effective pulse duration is $O(C_\theta \tau(\left\|(\partial H_n)^{G_l}\right\|)t)$ for the evolution $\exp\{-itG_l(H_n^{\text{dev}})\}$. C_θ is the overhead due to variational optimized pulse in effective pulse duration. The prefactor $\tau(\left\|(\partial H_n)^{G_l}\right\|)$ depends on the product formula, e.g for 1st-order trotterization $\tau(\left\|(\partial H_n)^{G_l}\right\|) = \left\|(\partial H_n)^{G_l}\right\|$. Thus the total effective pulse duration for a single k-th order TOBC with 1st-order trotterization is $C_\theta\left\|(\partial H_n)^{G_l}\right\|(2t_1 + 2t_2 + \cdots + t_k + T - t_k + T) = C_\theta\left\|(\partial H_n)^{G_l}\right\|(2T + 2\sum_{i=1}^{k-1} t_i)$. The worst case effective pulse duration is $O(2C_\theta\left\|(\partial H_n)^{G_l}\right\|kT)$. Because we are sampling b TOBCs for each loss function, we now analyze the average effective pulse duration for a single loss function. For M checkpoints, the average effective pulse duration is $C_\theta\left\|(\partial H_n)^{G_l}\right\|\frac{2T+2\sum_{i=1}^{k-1} t_i}{M^k}$ thus summing over all the k-th order TOBCs, we have

$$\begin{aligned}
\sum_{t_1,\cdots,t_k} \frac{2T + 2\sum_{i=1}^{k-1} t_i}{M^k} &= \sum_{t_2,\cdots,t_k} \frac{2MT + 2(T(1+M)/2 + M\sum_{i=2}^{k-1} t_i)}{M^k} \\
&= \sum_{t_3,\cdots,t_k} \frac{2M^2T + 2(TM(1+M)/2 + TM(1+M)/2 + M^2\sum_{i=3}^{k-1} t_i)}{M^k} \\
&= \frac{2M^kT + (k-1)TM^{k-1}(1+M)}{M^k} = (2 + (k-1)\frac{1+M}{M})T
\end{aligned} \tag{4.112}$$

Taking $M \gg 1$, we have the average effective pulse duration as $O(C_\theta\left\|(\partial H_n)^{G_l}\right\|(k+1)T) = O(C_\theta\left\|(\partial H_n)^{G_l}\right\|kT)$ when using 1st order trotterization. This removes n from the effective pulse duration when comparing to pure trotterization. However, this does not mean we break the optimal bounds such as [241]. Part of the complexity is moved into C_θ which becomes heuristic.

Due to our Hamiltonian has a large component of the dynamics governed by the device Hamiltonian, there is no dependencies of the total number of sites N in the effective

pulse duration. However, this does not mean we break existing gate depth bounds [241]. This complexity is moved into C_θ. Further analysis is required to understand C_θ in our effective pulse duration after reaching the optimal point. As for the digital resources, our HEM circuit requires $O(\left\|(\partial H_n)^{G_l}\right\|T)$ digital gates on w qubits at the boundary depicted in Figure 4.7.

Next, we discuss the complexity of shots. For each k-th order TOBC, there are $O(2^k)$ expectations to evaluate. Thus for batch size b there are $O(b2^k)$ expectations to evaluate. Assuming each expectation requires E shots for L growing steps, the total number of shots required is $O(bLE2^k)$. Last, without loss of generality, we discuss the complexity of evaluating the gradients using the parameter shift rule or finite difference. For other ways of evaluating the gradients [267, 268], one can derive in the same fashion. The complexity of evaluating the gradient of the loss function using finite difference depends on the number of parameters in device Hamiltonian, such as the Ω and Δ in our Rydberg Hamiltonian case. The rest of the parameters in the classical pulse function can be calculated via classical automatic differentiation. Thus, for P parameters in device Hamiltonian, we require $O(bLEP2^{k+1})$ shots in total.

4.6 Transforming Time-dependent Hamiltonians

The representation of a time-dependent Hamiltonian can be written as a sum of different time dependencies

$$H(\theta, t) = \sum_{i=1}^{n} \beta_i(\theta, t) H_i \tag{4.113}$$

where $\beta_i(\theta, t)$ is the time-dependent coefficient of a constant Hamiltonian H_i optionally with parameters θ. There are many ways to transform this Hamiltonian expression into another time-dependent Hamiltonian. However, different ways of transformation may result in different computational costs and may result in loss of information. For example, one may directly transform the matrix of $H(t)$ given the time t as $f^\theta[H(t)]$. However, this requires evaluation of the entire $f^\theta[H(t)]$ within the ODE solver, thus can be expensive. In the context of `jax`, the compiler cannot identify runtime-created closure. This can also easily lead to memory leak due to repeated runtime compilation of the same function [1]. A different approach is to map the constant components ahead of time. This approach is more efficient regarding runtime because fewer matrix operations are required within the

[1] See also the `jax` issue: https://github.com/google/jax/issues/16226

ODE solver loop. Thus we can define $f^\theta[H(t)]$ as follows

$$f^\theta[H(\theta,t)] = \sum_{i=1}^{n} g[\beta_i(\theta,t)]h[H_i] \tag{4.114}$$

where $g : \mathbb{R} \to \mathbb{R}$ is a function modifies the output of $\beta_i(\theta,t)$, and $h : \mathcal{A} \to \mathcal{A}$ is a function of the constant components. This approach only requires evaluation of the time-dependent coefficients within the ODE solver. Thus, this is cheaper compared to the previous approach when simulating classically. For OMM, this map can be written as

$$\mathrm{OMM}_n^\theta[H(t)] = \sum_{i=1}^{n} \beta_i(t)h^\theta[H_i] \tag{4.115}$$

where h^θ is the compression function that compresses the constant components of the Hamiltonian. We do not alter $\beta_i(t)$ here because the time-dependent coefficients can be adjusted by h^θ in the classical case.

This also allows direct transform between Hamiltonian expressions from pulse parameters to pulse parameters. In our OMM demonstration, because the TFIM Hamiltonian is constant, we only use a single $h[H_i]$. In our NEM demonstration, because we always map the constant Hamiltonian to a Rydberg Hamiltonian, thus the transform is defined as follows

$$\mathrm{HEM}_n[G_l(H_n^{\mathrm{ryd}}(\theta_n,t_n))] = g_1(\theta_n,t_n)\sum_i n_i n_{i+1} + g_2(\theta_n,t_n)\sum_i X_i + g_3(\theta_n,t_n)\sum_i n_i \tag{4.116}$$

where we simplified the map from $g(\beta_i(\theta,t))$ to $g_i(\theta,t)$ and set h to map constant components from $G_l(H_n^{\mathrm{ryd}}(\theta_n,t_n))$ to the constant components of the Rydberg Hamiltonian. Here because h is a deterministic function, it contains no parameters.

4.7 Results

To illustrate the convergence of OLRG as a variational principle and the associated classical and quantum algorithms, we applied our algorithm to the TFIM model as previously discussed. We investigated various hyperparameters to understand the algorithm's performance better. The implementation is available at the author's GitHub repository as an early-stage Python package [187]. Additionally, for other hyperparameters and training dynamics, we discuss the training dynamics in Section 4.8.1 for different orders of loss functions. For other hyperparameters without much impact on our reported results, we include the extra results in tuning batch and sampling sizes in Section 4.8.2 and the training using different step sizes of checkpoints in Section 4.8.3.

4.7.1 OMM

For our implementation of OMM, we initialize the system size at $n = 4$ and aim for a target system size of $N = 10$, setting the field parameter at critical point $h = 1.0$. The initial state is $\rho_0 = |0000\rangle$. The results of observable predictions are taken at the epoch with minimum moving average loss of window size 10. OMM is implemented by a neural network as discussed in Section 4.5.1, referred to as neural OMM in the following. The loss function is optimized via the gradient obtained from the adjoint method via `jax.experimental.ode`. The simulation utilizes only a single GPU. Our results are obtained from various different GPUs, including P100, V100, and A100.

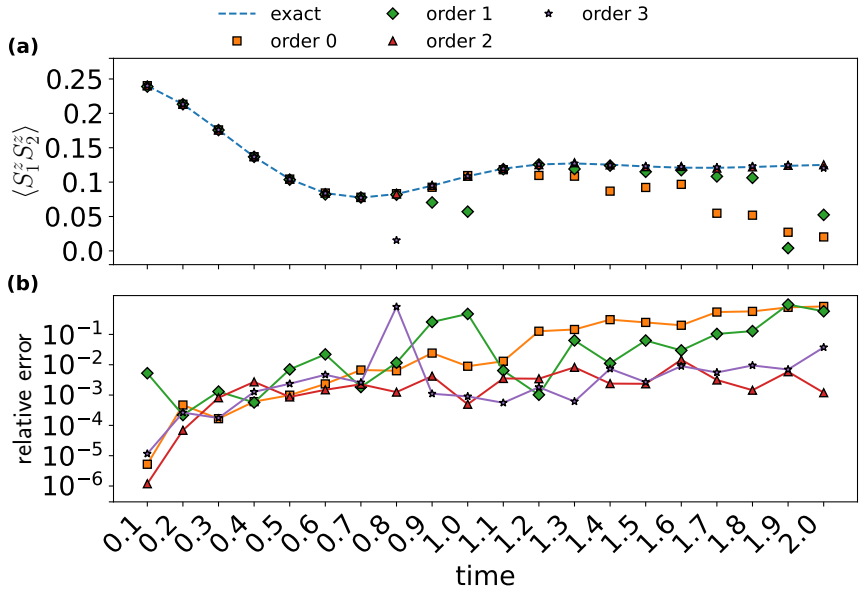

Figure 4.8: Comparison of OMM optimized at different loss function orders. (a) two-point correlation function $\langle S_1^z S_2^z \rangle$; (b) The relative error of the two-point correlation function $\langle S_1^z S_2^z \rangle$.

We first evaluate the performance of loss functions at different TOBC orders. Theoretically, increasing the order should enhance the precision of the loss function in estimating discrepancies, thus resulting in better performance. To test this, we measure the relative error of the time-evolved two-point correlation function $\langle S_z^1 S_z^2 \rangle_t$ against the exact result. In our study, the depth of neural OMM is 8. We train the neural OMM with 6000 epochs at each time point, starting from randomly initialized parameters. As depicted in Figure 4.8,

we observed that at short-time intervals, the order of the loss function does not significantly impact the results. However, at longer times, the 0-order and 1-order loss functions failed to produce the correct results in the OMM-based OLRG.

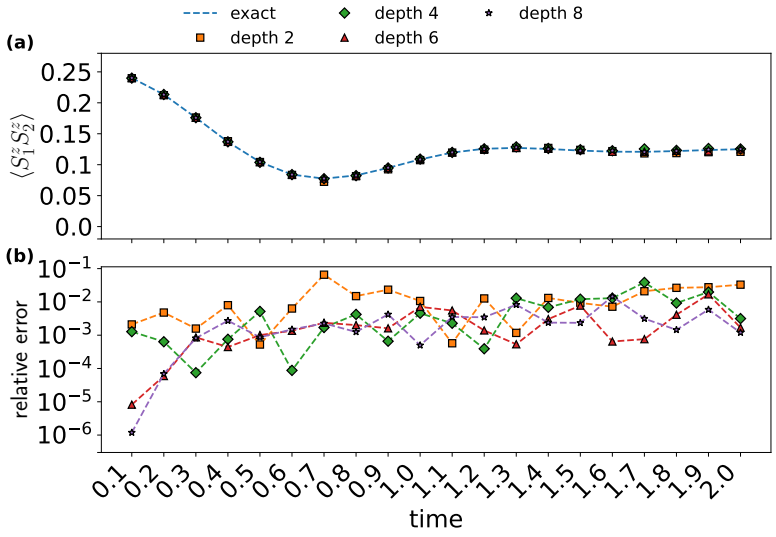

Figure 4.9: Comparison of different depths of the neural network in OMM optimized with 2nd order loss function. (a) The two-point correlation function $\langle S_1^z S_2^z \rangle$; (b) The relative error of the two-point correlation function $\langle S_1^z S_2^z \rangle$.

In neural OMM, the depth of the neural network corresponds to the expressiveness of the operator map. As depicted in Figure 4.9, we find that the depth of the neural network influences the relative error as well as the speed of convergence as shown in Figure 4.10. Deeper networks tend to converge faster and with a lower relative error. This is likely because deeper networks are more expressive and have better local minimums, thus allowing the algorithm to converge to a better solution faster.

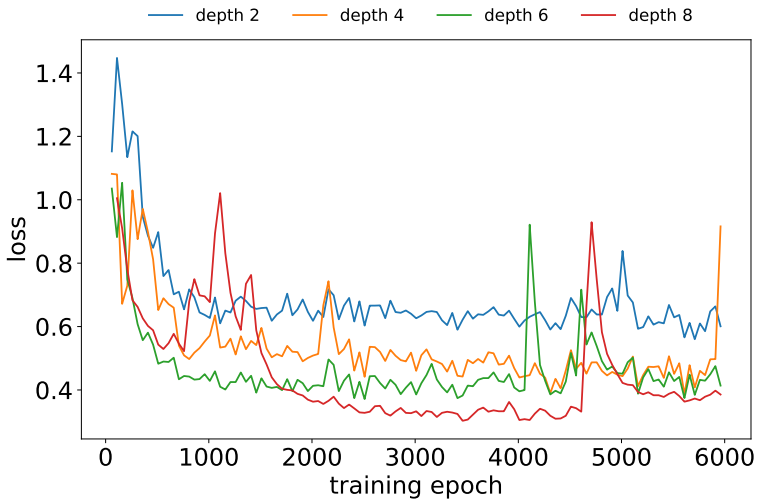

Figure 4.10: The loss function of different depths of neural OMM with 2nd order loss function at $T = 2.0$ with a moving average of window size 5.

4.7.2 HEM

For our implementation of HEM, we initialize the system size at $n = 2$ and aim for a target system size of $N = 6$, setting the field parameter at critical point $h = 1.0$. The initial state is chosen as $\rho_0 = |0 \cdots 0\rangle$. The results of observable predictions are taken at the epoch with the minimum moving average loss with window size 10. We use a 2-level Rydberg Hamiltonian as the target device Hamiltonian. The pulse function is represented by a small feedforward neural network that takes the clock t as input and returns the corresponding pulse value at time t. The simulation of the HEM algorithm is conducted on a single CPU. The loss function is also optimized via the gradient obtained from the adjoint method via *jax.experimental.ode*. In practice, the gradient could also utilize quantum gradient [68, 126, 267–269], finite difference, or other optimization algorithms suitable for the real device.

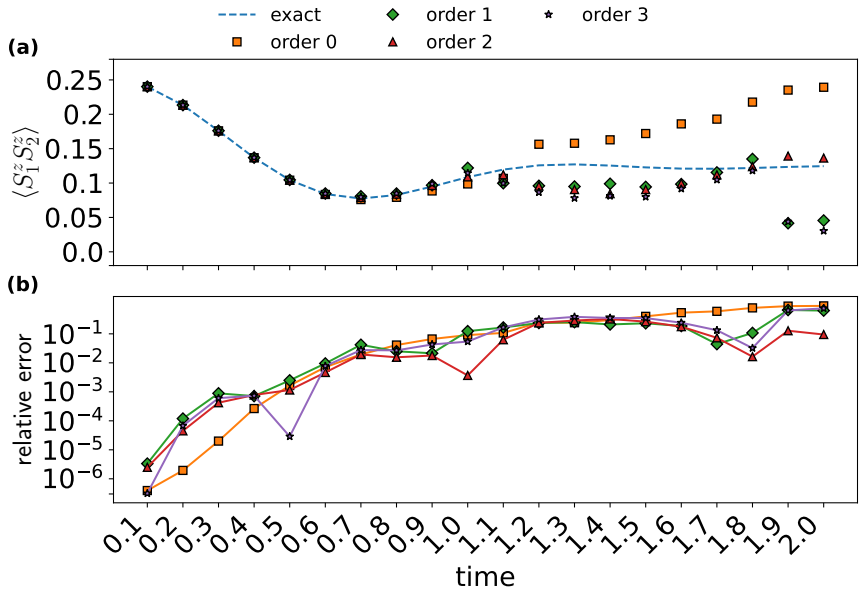

Figure 4.11: Comparison of HEM optimized at different loss function orders. (a) The two-point correlation function $\langle S_1^z S_2^z \rangle$; (b) The relative error of the two-point correlation function $\langle S_1^z S_2^z \rangle$.

We also evaluate the performance of HEM at different orders of the loss function. As depicted in Figure 4.11, similar to the classical algorithm, we observe that at short-time intervals, the order of the loss function does not significantly impact the results. At longer times, the 0-order loss functions drifts more from the exact result. However, the 3-order loss also drifts in $t = 1.9, 2.0$. We suspect this is due to insufficient optimization, because higher orders requires optimizing more discrepancies.

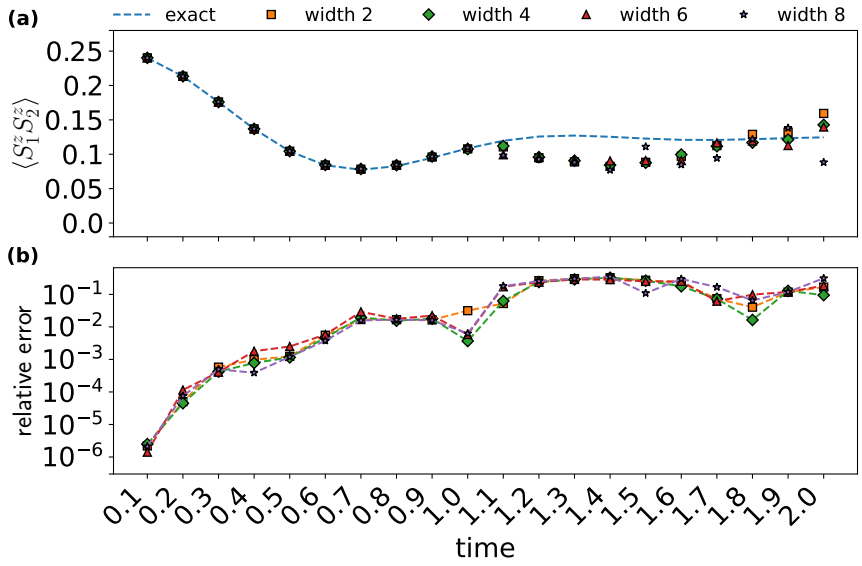

Figure 4.12: Comparison of the HEM optimized at different widths of neural networks with depth 4. (a) The two-point correlation function $\langle S_1^z S_2^z \rangle$; (b) The relative error of the two-point correlation function $\langle S_1^z S_2^z \rangle$.

For HEM, the expressiveness of representing a quantum dynamical process is mainly provided by the device Hamiltonian. Thus, the hyperparameters of the neural network affect the optimization rather than the expressiveness. We conduct a comparative analysis of the neural network's width and depth. As illustrated in Figure 4.12, we find that the width of the neural network (set at a depth of 4) does not significantly influence the algorithm's performance. In contrast, the depth of the neural network (set at a width of 4) shows a notable impact. As depicted in Figure 4.13, a deeper neural network leads to diminished performance, likely due to a vanishing gradient that does not provide a better landscape. Conversely, shallower networks are more successful in identifying an appropriate pulse function. All the results of HEM start drifting after $T = 1.1$. We hypothesize that this is due to the 2-level Rydberg Hamiltonian not being universal.

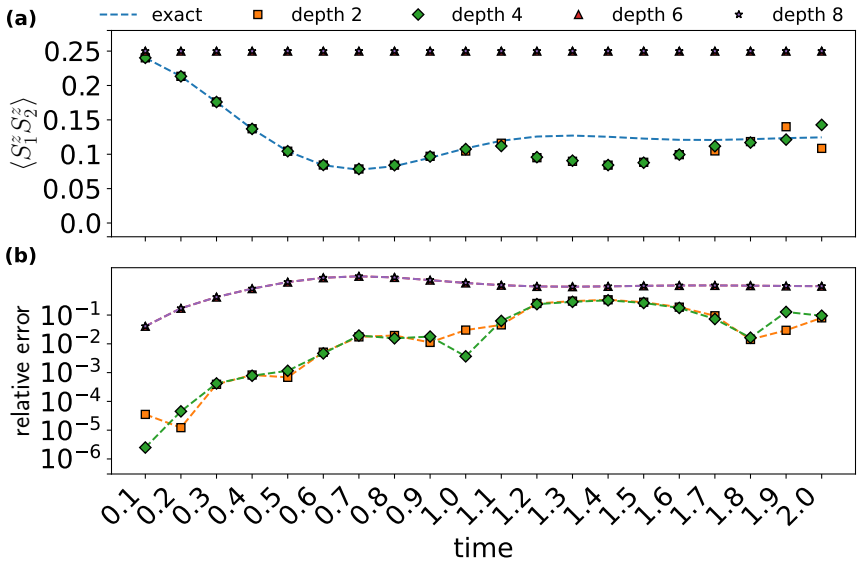

Figure 4.13: Comparison of HEM optimized at different depths of neural networks with width 4. (a) The two-point correlation function $\langle S_1^z S_2^z \rangle$; (b) The relative error of the two-point correlation function $\langle S_1^z S_2^z \rangle$.

4.7.3 Transfer Learning between Time Points

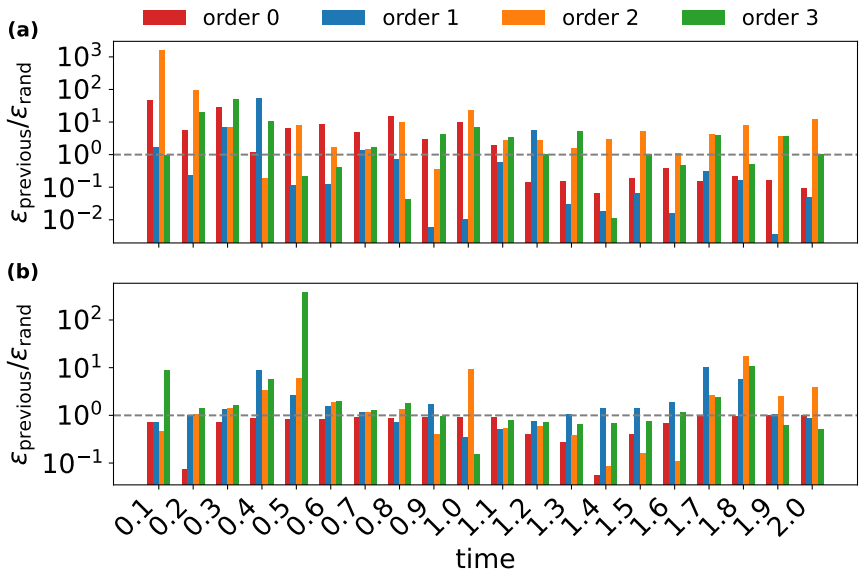

Figure 4.14: Transfer learning to different time points. Compared by a different order of loss function. The y-axis is the ratio between the relative error of initialization from previous time point $\epsilon_{\text{previous}}$ and random initialization ϵ_{rand}. Above the line $y = 10^0$ means random initialization is better; below the line means initialization from the previous time point is better. (a) neural OMM; (b) HEM targeting Rydberg Hamiltonian;

We investigate the transfer learning between time points with uniformly training each time point for a fixed number of epochs. For neural OMM, we allocated 1000 epochs at each time point, while for HEM, we allocate 500 epochs. As depicted in Figure 4.14, this approach reduces the number of epochs needed compared to initialization from random parameters, yet it still delivers similar performance levels. Additionally, initializing from the parameters of the previous time point results in the lower order loss function achieving a better relative error than when starting from random parameters. This improved performance can be attributed to the smooth nature of this specific time evolution, which allows high-order correlations to propagate through the parameter initialization. In contrast, when OMM represents a pure isometry, it is possible to express an explicit state as a MPS. Therefore, initializing from the parameters of the previous time point can be viewed as utilizing an implicit quantum state as input.

4.8 Additional Results

4.8.1 Training History

The training history at different orders of OMM and HEM are shown in Figure 4.15 and Figure 4.16. The left column is OMM, and the right is HEM. Each row is ordered by time. In a short time, there isn't a significant difference between each order. However, with time increase, the higher order loss function approaches higher precision faster and can reach significantly higher precision over a long time than the lower order loss function.

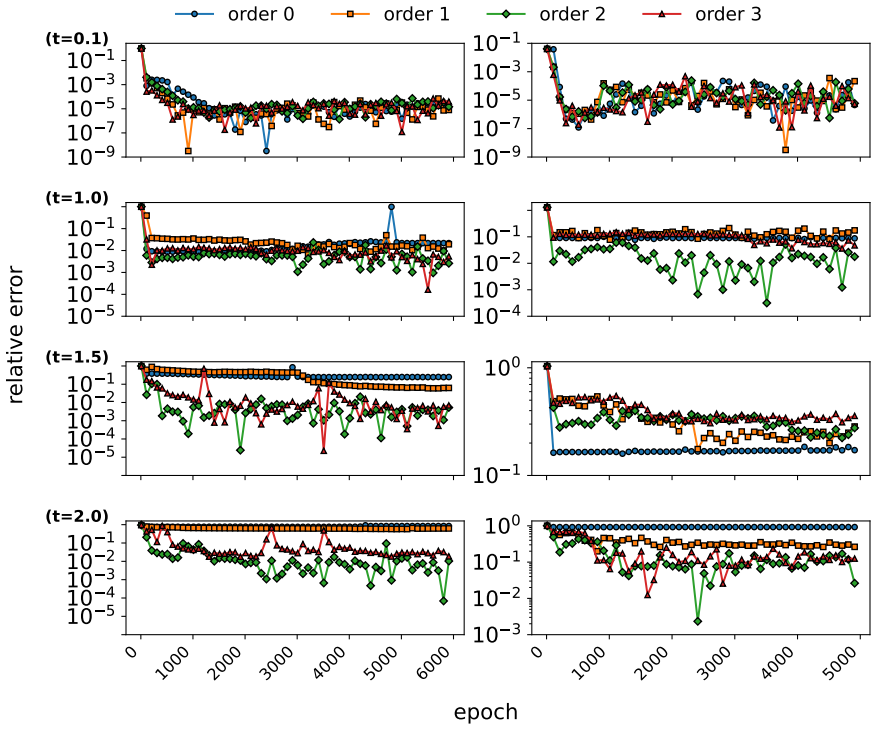

Figure 4.15: Training history of relative error. Left is the training history of the classical algorithm, and right is the training history of the quantum algorithm.

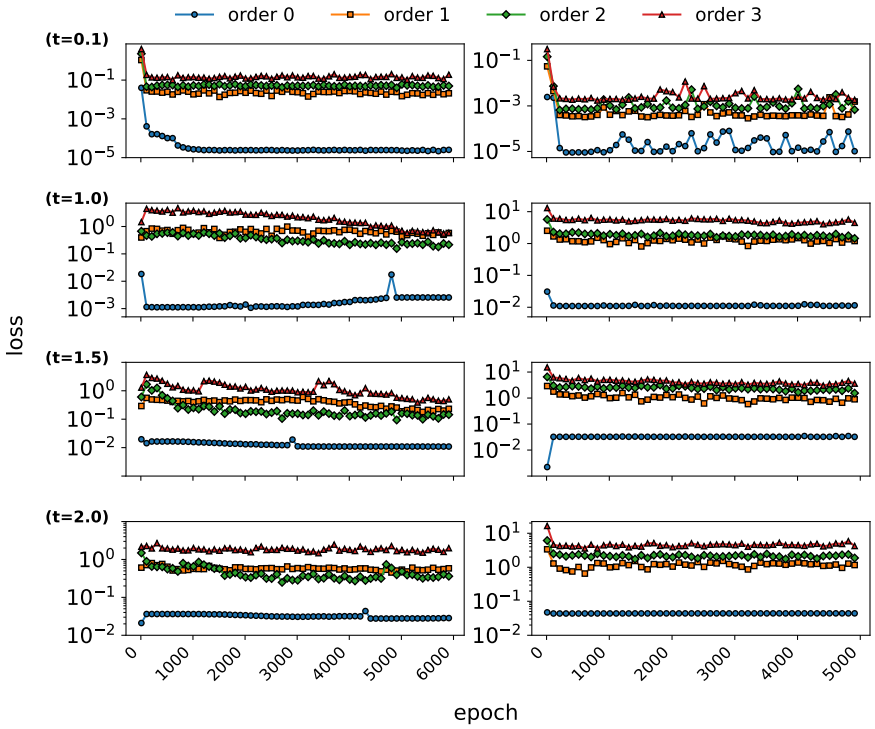

Figure 4.16: Training history of the loss function. Left is the training history of the classical algorithm, and right is the training history of the quantum algorithm.

We also show the history of the loss function when we reuse the previous time point's parameters in Figure 4.17. The left column is the OMM, and the right is the HEM. Each row is ordered by time. The loss function is larger at higher order, we suspect the higher order TOBC is harder to optimize because the search space is larger than the lower order thus resulting in a worse absolute value of the loss function but a better relative error.

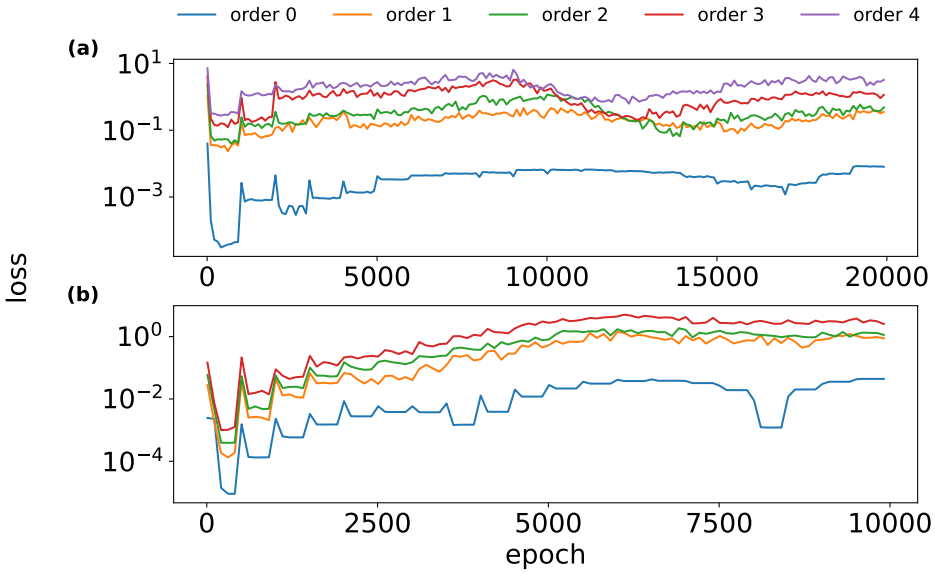

Figure 4.17: Training history of the training by reusing previous time point's parameters. (a) The history of loss function for OMM. (b) The history of loss function for HEM.

This also reflects our theoretical bound in Theorem 2 is not a tight bound. As pointed out in Section 4.9.1, the loss function contains many zero TOBCs, thus they do not actually contribute to the loss function. Gradient-based optimization towards a truth value of zero is often hard due to vanishing gradients. On the other hand, these TOBCs do not contribute to the final error. Thus we expect the loss function can be improved by removing the zero TOBCs.

4.8.2 Batch and Sampling Size

We also compared the batch size and sampling size. The batch size controls the sampling variance of the neural OMM, which parameterizes an ensemble of small systems. Increasing batch size will reduce the variance of gradient estimation. We did not observe a significant difference in tuning batch size. The results are shown in Figure 4.18.

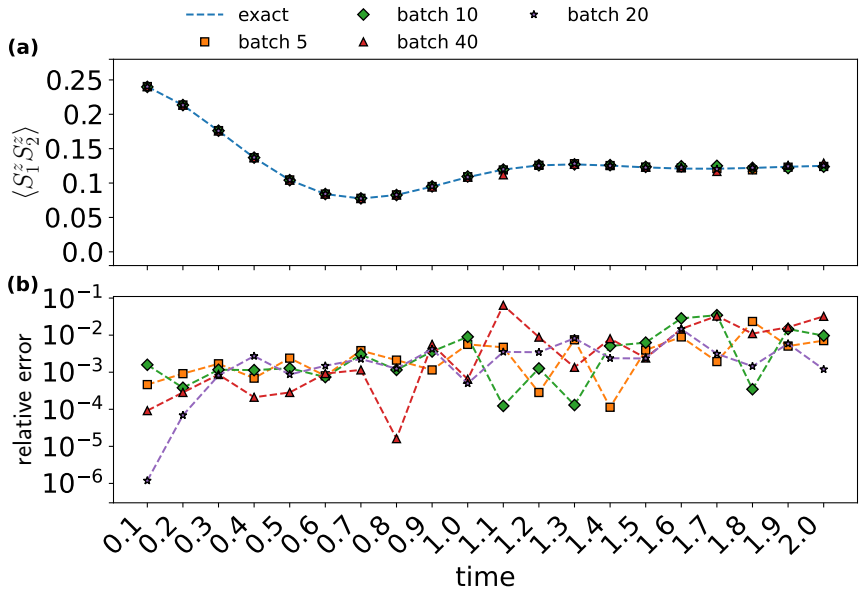

Figure 4.18: Comparison of different batch sizes at order 2, with depth 8 for OMM. (a) The value of $\langle S_1^z S_2^z \rangle$; (b) the relative error.

This is actually reasonable because the batch does not contribute to the expressiveness. When the average error of the ensemble start decreasing in zero, the variance caused by small batch size should not be significant in the optimization as the gradient will approach zero.

The sampling size controls how many observables are sampled to estimate the loss function at each evaluation. Increasing the sampling size should decrease the variance in the gradient. We did not observe a significant difference in tuning sampling size. The results are shown in Figure 4.19 and Figure 4.20.

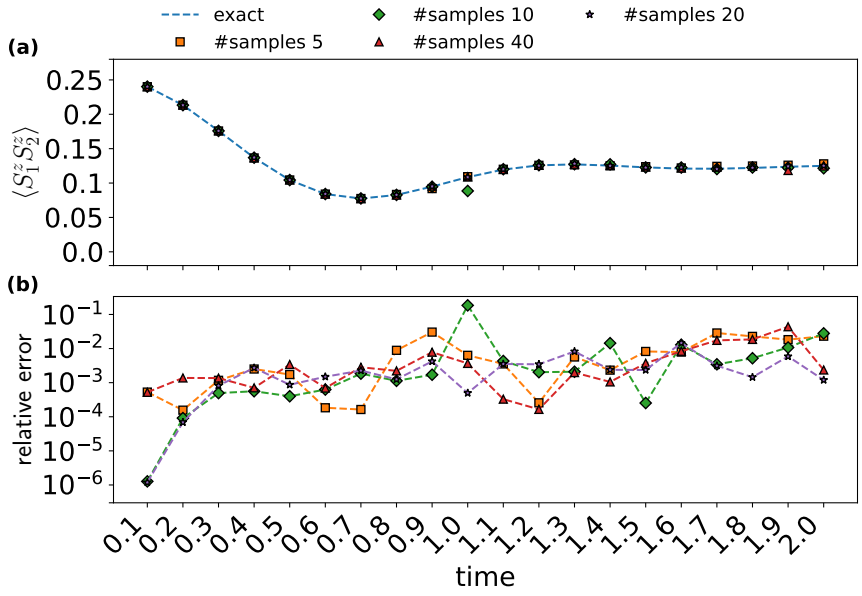

Figure 4.19: Comparison of different sampling sizes at order 2, with depth 8 for OMM. (a) The value of $\langle S_1^z S_2^z \rangle$; (b) the relative error.

Based on the exact results in Figure 4.5 and Figure 4.22, we believe the actual non-zero TOBCs at each order for TFIM could only be polynomial. Thus, increasing the number of samples will not actually "hit" the non-zero TOBCs by a large factor. As a result, this does not increase the performance of gradient-based optimization. We also hypothesize that the variance of the loss function is useful for SGD exploring better minimum, which compensates for the performance drop caused by sampling. On the other hand, the toy problem we ran our simulation with might be too simple to demonstrate the difference. As one may expect, all the TOBCs to be non-zero in a more complex problem. We also observe a flat trend in the relative error of Figure 4.19 and Figure 4.9, indicating the error can stay around 10^{-3} for longer times.

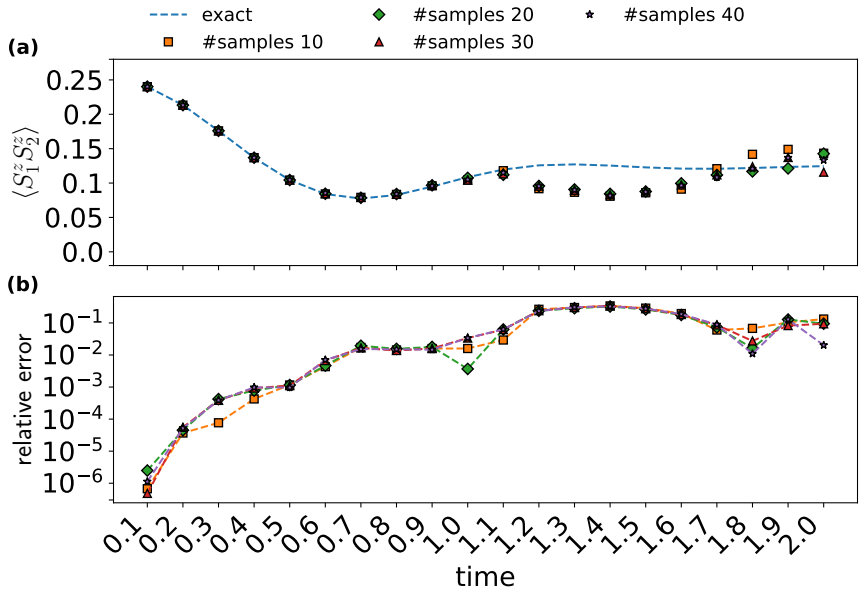

Figure 4.20: Comparison of different sampling sizes at order 2, with depth 8 for HEM. (a) The value of $\langle S_1^z S_2^z \rangle$; (b) the relative error.

For the HEM-based OLRG algorithm, following the discussion from Section 4.7, we observe a consistent bump between $t = 1.2$ to $t = 1.8$ with similar values across different hyperparameters from Figures 4.11 to 4.13 and 4.20. This indicates that the optimization has converged at these time points. The relative error is consistent across different hyperparameters, indicating that the optimal point within the space of the 2-level Rydberg Hamiltonian with a global pulse sequence has been reached. Thus we suspect this performance drop is due to the non-universal nature of the 2-level analog Rydberg Hamiltonian. We see the rise of the relative error decrease at $t = 1.9, 2.0$ with a relatively smooth change in the absolute value. Thus we suspect this is only due to coincidence, where the Rydberg Hamiltonian dynamics are close to the two-point correlation dynamics of the TFIM at these time points.

4.8.3 Step Size

As for the step size δ in the sampling, we tune the number of checkpoints M in an ODE solver, which controls the step size as $\delta = T/M$. While smaller step sizes generally increase the precision of the loss function, we did not observe a significant difference in the tuning

step sizes. The results are shown in Figure 4.21. As discussed in Section 4.2, this is likely because the TOBC in $1D$ TFIM has many zeros and is quite smooth. Thus, the loss function is not sensitive to the step size.

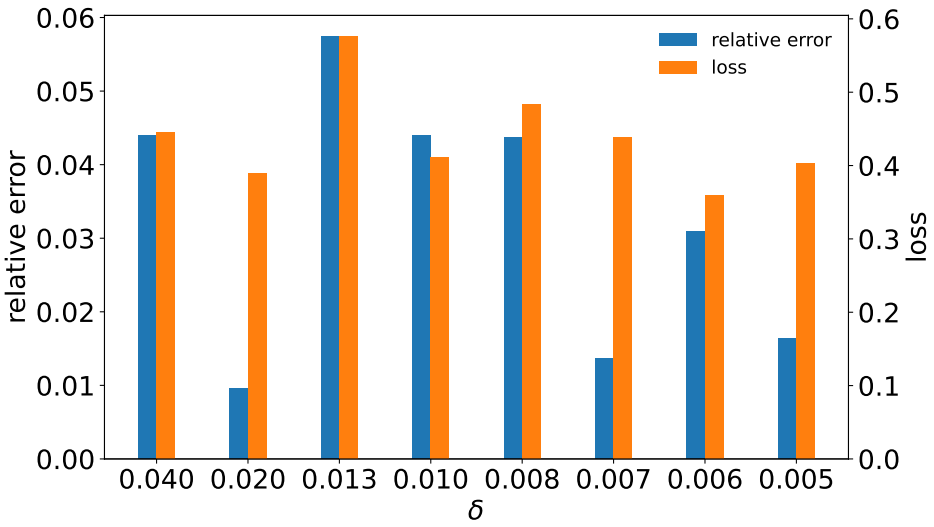

Figure 4.21: Comparison of different step sizes δ at order 2, with depth 8.

This result is consistent with our analysis in Section 4.2 about TOBCs. Because the actual dynamics across the entire time of evolution are smooth. The step size of checkpoints in a small-system solver does not need to be very small. This heuristic analysis eliminates the concerns about fine step sizes in the ODE solver when simulating practical systems. However, we expect the step size to be smaller when simulating fast oscillating dynamics, when the TOBCs are not smooth.

4.9 Discussion

In this work, we have introduced an algorithmic framework named OLRG, an alternative variational principle that generalizes Wilson's NRG and White's DMRG. The algorithm's e2e-style loss function directly bounds the real-time dynamics of target observables. The OLRG framework allows us to introduce different categories of ansatzes for an operator map between a real and a virtual system. We designed operator maps for real-time dynamics simulation on conventional computers and quantum devices. This includes the OMM and

the HEM. OMM opens up new possibilities to provide more expressiveness than the linear operator map in NRG and DMRG. This could lead to opportunities to explore challenging real-time dynamics problems, e.g. in higher dimension lattices, by exploring different forms of the growing operator G_k (Section 4.9.4). As a side product of this work, we also see OLRG as a potentially complementary variational principle to address high-order and long-time correlations for MPS TDVP in Section 4.9.5. In addition, our HEM-based OLRG provides a digital-analog quantum algorithm that integrates the product formula, VQA, and classical simulators for simulating quantum dynamics. Finally, we discussed tuning different hyperparameters and training schedules to improve the algorithm's performance in calculating two-point correlations for TFIM undergoing real-time dynamics.

Advancement to the OLRG framework can be made by enhancing the operator map and loss function. For the loss function, one could derive a more specific loss function for the target problem and further explore the relationship between superblock formalism from DMRG and series expansion (Section 4.9.1). Because Theorem 1 is general for any properties. Another future direction is finding the loss function directly for other properties such as ground state properties, phase transition points, entanglement entropy, etc. (Section 4.9.3). This will align the framework further with e2e learning for calculating other properties. Such loss functions likely exist due to the success of NRG and DMRG in evaluating various properties especially in solving ground-state problems.

For the operator map, as a further step one can consider the implementation of OMM including tensor network ensembles and deep neural network architectures. The target device Hamiltonian of HEM could be expanded to universal neutral atom arrays, ion traps, and superconducting circuits with different control capabilities. We discuss various ansatz designs under the OLRG framework in Section 4.9.2. In this paper, we only investigated the simplest OMM implemented by a feedforward neural network and a HEM targeting the non-universal 2-level Rydberg Hamiltonian. More powerful operator maps remain to be explored in future research. For real quantum devices, skipping the step of compiling the Hamiltonian terms into gates and directly using the pulse sequence may result in a non-trivial pulse sequence that is more efficient than 1 or 2-qubit gates. This is because, for real devices, certain global unitaries are easier to implement with shorter pulse sequences than decomposing into gates [270]. Thus, one may expect the effective pulse duration to be shorter than performing small-qubit gate compilation. The HEM-based OLRG can thus be also viewed as a quantum-assisted quantum compilation algorithm [271]. A future direction is to benchmark the effective pulse duration in this case. Another interesting direction is exploring the generalization capability of the operator map for a larger system size trained at a small system size. By utilizing previous theoretical work about finite size error [149], one may derive the e2e-style loss function for infinite-size systems. Then,

one could attempt to train the operator map to predict properties directly for infinite-size systems.

4.9.1 Improving Loss Function

The theoretical bounds we present for real-time evolution in Theorem 2 is a general estimation for arbitrary geometrically local Hamiltonian. Thus, it is rather a loose bound considering more specific system properties. We believe that a tighter bound can be established for specific system properties. This may result in a better loss function and a more efficient algorithm. Furthermore, the global loss function and modeling by e2e provide advantages in that every learning step optimizes the target problem but also has limitations [272]. The usage of e2e heavily relies on optimization and thus may result in a slow convergence and ill-conditioned optimization. Our framework also allows theoretical improvements through a better theoretical understanding of the problem, such as the analytical or heuristic understanding of the TOBCs. This will incorporate the theoretical knowledge into the loss function and thus may potentially improve the algorithm's efficiency. For example, as shown in Figure 4.5 and Figure 4.22, some TOBCs can be almost perfectly zero, and the nonzero TOBCs are also very sparse in $1D$ TFIM dynamics, where many points are relatively small thus result in a small contribution to the loss function. This suggests that by looking into specific Hamiltonian and TOBCs, we may be able to design a better loss function that can be more efficient in practice.

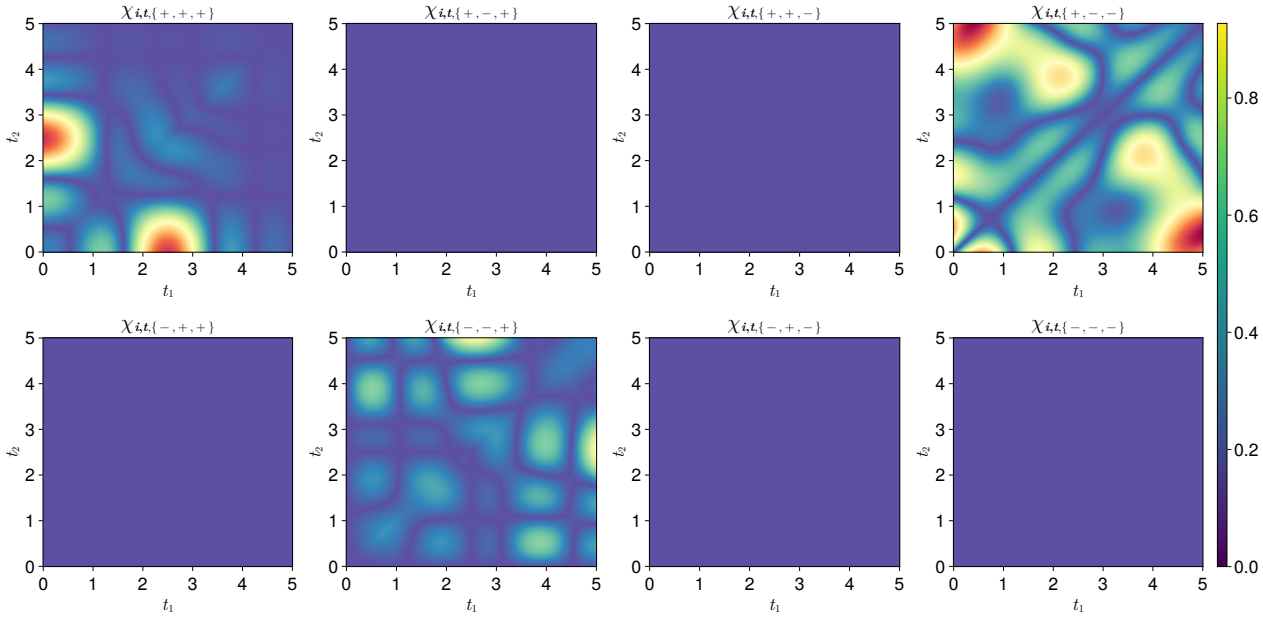

Figure 4.22: 3rd-order TOBC for 5-site $1D$ TFIM at $T = 5.0$ for the two-point correlation function $\langle Z_1 Z_2 \rangle_{T=5.0}$ with $|00000\rangle$ as initial state and $h = 1.0$. We fix $t_3 = 2.5$ and plot the TOBC for $t_1, t_2 \in [0, 5.0]$.

On the other hand, there is a different possible style of constructing the loss function by reusing existing scales. Using the same superblock construction as DMRG, one can see such loss function as the following concept. If we can prepare good representations of the systems at size n_1, n_2, then concatenating them into a system at size $n_1 + n_2$ should be a good representation of the system at size $n_1 + n_2$. This is a very natural assumption, and it is also the core idea of superblocks. From a series expansion perspective, assuming we have the series expansion of a property p as

$$p(S_{n_1} \otimes S_{n_2}) = \sum_{i=0}^{\infty} \alpha_i \langle A_i \rangle \langle B_i \rangle \tag{4.117}$$

Here, we can use the infinite DMRG style loss function as an example. If we are promised to have a good representation of S_{n_1}, copying it then the concatenating system $S_{n_1} \otimes S_{n_1}$ should be a good representation of $2n_1$ system. This is exactly the description of infinite DMRG [150] in the traditional fashion. If we assume the property we are calculating is the same observable. The series expansion becomes a sum of square expressions:

$$p(S_{n_1} \otimes S_{n_1}) = \sum_{i=0}^{\infty} \alpha_i \langle A_i \rangle^2 \tag{4.118}$$

If the loss function is then set as the error of observables on this superblock, we can see we are optimizing the error of some high-order terms in the Dyson series.

$$\begin{aligned} L &= \left\| p(S_{n_1} \otimes S_{n_1}) - p(S'_{n_1} \otimes S'_{n_1}) \right\| \\ &= \left\| \sum_{i=0}^{\infty} \alpha_i \langle A_i \rangle^2 - \sum_{i=0}^{\infty} \alpha_i \langle A'_i \rangle^2 \right\| \end{aligned} \tag{4.119}$$

However, it remains uncertain whether this loss function is upper-bounded in a manner that would rigorously ensure the scaling consistency condition. Utilizing superblocks in practice will improve the efficiency of evaluating the loss function, as now one does not need to solve time-evolved operators but can directly evaluate the discrepancies between expectation values in superblocks instead. This is similar to using the superblock in DMRG to evaluate the system's energy.

4.9.2 Improving Operator Maps

Like all other variational algorithms, the expressiveness of the operator map is crucial to the algorithm's performance. In our current implementation, we only use some vanilla operator maps without much careful design.

For OMM, the power of optimizing operator maps, such as deep neural networks or tensor network ensembles, is yet to be explored. Furthermore, one can use different operator maps for larger system sizes for different scales and only share at closer scales. This naturally creates a hierarchical structure of the operator map, similar to how depth of neural networks are used in deep learning [273]. As we now utilize an operator map for operator mapping rather than a state, the expressiveness of such an operator mapping remains to be determined. More specifically, although our neural OMM has the potential of expressiveness representing MPS with exponential large bonds. Because they are no longer ansatzes for states, it is unclear what the limit of such operator maps is.

For HEM, we only use a simple and small neural network to parameterize the pulse, which does not consider more realistic pulse shapes. Thus, the result pulse shape does not necessarily execute on real hardware due to violation of hardware constraints. On the other hand, our HEM targets a non-universal Hamiltonian, thus resulting in worse performance

over a longer time despite increasing the order. It is important to explore more realistic pulse shapes, universal Hamiltonians, and more detailed device capabilities to develop a better understanding of the algorithm.

4.9.3 Finding the Loss Function for Other Properties

We have demonstrated the existence of a loss function that effectively bounds the real-time dynamics of local observables. However, this methodology might not be entirely end-to-end (e2e) when dealing with target properties that are complex functions not directly derived from local observables, such as phase transition points or entanglement entropy. Additionally, our Theorem 2 does not extend to imaginary-time evolution or ground-state simulations. Because the series expansion we derived does not hold in these cases. Despite these limitations, our framework is not intrinsically confined to real-time dynamics alone, as suggested by Theorem 1. The proven effectiveness of NRG and DMRG inspires the possibility that suitable loss functions for imaginary time and ground state challenges may also exist. To further follow the e2e principle in solving real-world quantum many-body simulation problems, we seek if there is a loss function that guarantees scaling consistency for other properties such as entanglement entropy, phase transition points, etc.

4.9.4 Higher Dimension Lattice and Other Geometry

While our numerical results are confined to a $1D$ lattice in Section 4.7, it's important to note that, like NRG and DMRG, the variational principle, OLRG framework is not inherently limited to this geometry. Indeed, the OLRG framework can be applied to any geometric configuration. However, in geometries other than $1D$, the implementation of the growing operator presents a range of alternative strategies that have yet to be fully explored. Moreover, by incorporating the growing scheme into the loss function, our operator map no longer necessitates an exponential increase in storage, provided that $f^{\theta}_{n_q}$ is not a dense isometric map. Consequently, techniques developed for navigating $1D$ ansatzes, such as MPS [274] and autoregressive neural networks [217], could be adapted and prove beneficial in 2D and other configurations within this framework.

4.9.5 Relation with MPS TDVP

When $f^{\theta}_{n_q}$ is a linear map, the OLRG framework is equivalent to a tensor network. For example, if $f^{\theta}_{n_q}$ is not shared by each OLRG step, and denoting $f^{\theta}_{n_q}$ for q-th OLRG step,

the set $\{f^{\theta}_{n_q}\}$ represents the tensors in an MPS as shown in the left column of Figure 4.2. On the other hand, the MPS TDVP algorithm projects an MPS $|\psi(t)\rangle$ at time t to the MPS $|\psi(t+\delta)\rangle$ at time $t+\delta$ by solving the Schrödinger equation in the subspace of MPS. Assuming the bond dimension does not change from $|\psi(t)\rangle$ to $|\psi(t+\delta)\rangle$, the MPS TDVP should find the optimal MPS representation for $|\psi(t+\delta)\rangle$. By optimal $|\psi(t+\delta)\rangle$ we mean this state has the minimum error for arbitrary observables at $t+\delta$. Thus, this is equivalent to optimizing support of observables at $t+\delta$ using OLRG starting from the initial point $|\psi(t)\rangle$, which is the transfer learning algorithm we introduced in Section 4.7.3.

From this perspective, plugging the small duration δ into Theorem 4, we can see that the loss function of the MPS TDVP algorithm directs the optimization towards the $t+\delta$ time observables instead of the final time T observables. Thus, only the error of $\chi_{\boldsymbol{i},\boldsymbol{t}=\{\delta\},\sigma}(S_n, t+\delta)$ are optimized in the MPS TDVP algorithm for an arbitrary observable $O(t)$, thus missing longer time correlations in the optimization target. This is consistent with the recent analysis of the MPS TDVP algorithm in the ancillary Krylov subspace TDVP [275].

With this observation, we can see that the OLRG framework can be a complementary approach to the MPS TDVP algorithm. A potential improvement to the MPS TDVP algorithm is to add the loss function of the OLRG framework as a regularization term for long-time TOBCs, and thus help the MPS TDVP algorithm to include long-time correlations in the optimization target and increase the efficiency of the MPS TDVP algorithm for longer time by increasing the step size δ. However, the gradient-based optimization in OLRG also has its limitations, as it may not be able to adjust bond dimension variationally, thus for pure MPS, it can only be used as a regularization term instead of the main optimization target.

4.9.6 Implementation

Both classical and quantum algorithms were optimized using the ADAM optimizer [276] and implemented via the recent automatic differentiation frameworks and GPU computing `jax` [10] and `flax` [277] frameworks. Due to the absence of sufficient sparse matrix support when the author implements the software in `jax`, a brute-force solver was employed to compute the observables. This limitation restricted the quantum algorithm's simulation to no more than 6 sites due to memory limitation. The classical algorithms use single NVIDIA GPUs, while the quantum algorithms are executed on 1 CPU cores. From an implementation perspective, the OLRG framework opens the door to adapting good small-system solvers to larger systems. Thus, like DMRG, all the technologies developed for small-system solvers can be transferred into large system calculations. We believe that

by integrating with better small-system solvers, the practical performance of the OLRG framework can be further improved. We thank the support of the open-source community in the development of the following software, they contributed directly in producing our results: `jax` [10], `flax` [277], `optax` [278], `tqdm`, `wandb`, `matplotlib` [279], `Yao` [201], `Makie` [280].

4.9.7 Optimization

Differentiable programming is a powerful technique for optimizing and training arbitrary operator maps. One should not expect gradient-based optimization always to work well. Besides the advantages we have demonstrated in this paper, the disadvantages of differentiable programming include longer convergence time and higher evaluation cost compared to iterative optimization, such as utilizing eigensolvers. Future directions in improving gradient-based optimization include co-designing optimization and operator maps by incorporating symmetries and guidance from real data in medium-size systems [281, 282]. Because a large component in the time complexity can be parallelized, potential technical improvements may be seen in utilizing distributed gradient descent [283].

Chapter 5

Conclusion

In this thesis, starting from the motivation of integrating physicists from different subfields in our community, designing high-performance multi-purpose software frameworks for quantum many-body systems, and understanding the existing methods from a programmatic perspective, we have introduced the concept of programmatic representation and its application in quantum many-body physics. We introduced three tree-based representations for quantum circuits, general quantum operators, and pulse sequences. All these representations have been tested and iterated in many real-world simulations and experiments, including but not limited to previous experiments mentioned in the white paper [104]. Thus we see the success of using the programmatic representation to accelerate scientific discoveries. We further show that more sophisticated representations can be built by combining the semantics of these representations with the concept of static single assignment (SSA) form. This representation as an intermediate representation (IR) allows more complicated analysis of the representation, such as constant propagation and dead code elimination [284]. While the use of SSA IR for high-level circuit-based quantum programs could be an overkill, we discussed the potential of using SSA IR for low-level compilation on quantum devices due to the needs of asynchronous and real-time execution. Thus, we see the opportunities in designing a more sophisticated IR for quantum many-body physics that covers a broader range of problems and potentially leads to better simulation algorithms and performance using both conventional and quantum computers. By incorporating multiple levels of IR, we expect the communities of theoretical, computational, and experimental physicists will once again be united by compilation techniques [64, 109, 285], allowing the community to push the boundary of quantum many-body physics further.

We further introduced the transformations on top of these representations. Starting from the techniques pushing exact simulations to hardware limits, we introduced the trans-

formation from an expression into the subspace matrix-vector multiplication routine and special matrices, which execute the simulation. Moreover, we introduced the techniques for implementing these routines and special matrices in the context of exact simulation. On top of these building blocks, we introduced the transformation for automatic differentiation and the use of SSA IR for implementing automatic differentiation. We also benchmarked the performance of the exact simulation and the automatic differentiation and showed that the exact simulation is already approaching the limit of the classical computer. Thus, our software implementation of these exact simulations achieved state-of-the-art performance compared to other available software with only thousands of lines of code [201]. These performance improvements also lead to the study of using the Rydberg atom array for combinatoric optimization problems [118].

Moreover, we showed that by utilizing the reversibility of the quantum circuits, only constant memory is required for performing automatic differentiation in classical emulation. The memory complexity improvement leads to the example of differentiating a 10,000-layer parameterized quantum circuit, which no previous software can achieve. Thus, the reverse mode AD by reversibility opened the possibility of exploring large variational quantum algorithms with automatic differentiation. We further discuss the future directions of studying the transformations on these representations and the potential of using these transformations to automatically specialize the simulation on problems with specific properties, which we hope can be achieved in the next-generation software framework `Liang`.

Based on our previous progress and understanding in programmatic representations, exact simulation, and automatic differentiation, we revisit the well-known Wilson's NRG and White's DMRG algorithms [26, 27]. By reviewing these two numerical renormalization group formulations proposed a few decades ago. We have shown that the concept of programmatic representation is not only a concept for software engineering but also a concept for designing new theories and methods in our recent work OLRG [286]. We introduced an alternative variational principle for quantum many-body systems allowing arbitrary operator maps as ansatzes in lieu of state ansatzes. A theory guiding the design of end-to-end loss functions is also proposed. We further proved that a loss function exists with rigorous error bound for real-time evolution. We proposed the neural operator matrix map (OMM) for simulation on conventional computers and the Hamiltonian Expression Map (HEM) for simulation on quantum computers. By building on top of classical algorithms, the HEM-based OLRG as a quantum algorithm integrates the conventional computational algorithms and the quantum algorithms. Furthermore, As a generalization of Wilson's NRG and White's DMRG, our framework is also compatible with tensor networks, thus can potentially improve existing tensor network algorithms.

We see our framework aligns well with conventional deep-learning algorithms. The

model processes operators, generates operators as output, and then takes the output operators as input for the next iteration. This hierarchical procedure naturally creates the concept of depth in the model. Our setup also aligns well with the idea of the mini-batch from deep learning, as the model is trained on a dataset of operators. The utilization of batch allows our framework to be highly efficient in utilizing the parallelism of modern hardware such as GPUs and TPUs. Furthermore, the concept of end-to-end learning also allows us to eliminate the unnecessary bias caused by intermediate targets, which aligns well with the concept of end-to-end learning.

An early-stage software framework has been implemented for exploring this direction [187]. Using this software framework, we demonstrate the convergence of our theoretical loss function by solving the two-point correlation function dynamics of TFIM. For the OMM, we explored the effect of different hyperparameters on the relative error of our prediction. We show that the neural network's depth helps improve the relative error and optimization while other hyperparameters remain less important. We also show that the OMM can be used to simulate the dynamics of the two-point correlation function of a 1D 10-site TFIM with a relative error of 10^{-3} at $t = 2.0$. This result is not yet the state of the art. However, we discussed the relationship between OLRG and TDVP. The theoretical analysis indicates that TDVP will lack long-time correlations, and thus, the optimization will throw information necessary for calculating long-time results at each time step. We thus see the potential of OLRG to be state of the art in high-dimensional long-time dynamics. Concurrently, we also demonstrated the HEM-based OLRG, which allows compilation of an input problem Hamiltonian into target quantum device pulse sequence. We show the HEM-based OLRG can achieve 10^{-3} relative error before $t = 1.0$ in simulating the two-point correlation function dynamics for a 1D 6-site TFIM Hamiltonian. Our results show promising precision before $t = 1.0$ compared to the state of the art [262], while after $t = 1.0$, we see a bump in relative error between $t = 1.2$ to $t = 1.7$ consistently on all the hyperparameters we explored. We suspect future research targeting a universal Hamiltonian should be able to solve this problem.

The OLRG framework has also led to many interesting open problems in both theory, numerics and experiments. We discussed them in detail at the end of introducing this work. For example, the loss function, in principle, does not have to be the error of observables but can be the error of the final target directly, e.g., the phase transition point, entanglement entropy. The loss function for these properties remains an open problem in theory. Due to the success of DMRG and time-dependent DMRG, we believe such a loss function should exist. In the context of numerics, there are many interesting generative models that can be directly borrowed from the deep learning community. Compared to VMC, which uses discrete configurations as input, the operator matrices as input and output align well with

images in computer vision. This field has seen significant progress due to deep learning. Unlike using neural networks in VMC, there is no need to engineer complex numbers in OMM. The color channels in images have a natural connection with complex numbers.

Furthermore, the operator matrices are naturally continuous numbers built on top of primitive operators such as Pauli operators. Such matrices as input allow the use of generative models such as flow models [227] and diffusion models [229], which has been hard to incorporate into the VMC framework. Such possibilities open the imagination of designing more expressive models that may lead to better performance. In the context of experiments, the OLRG framework can be used in the context of quantum computing. Especially on Rydberg atom arrays, our HEM-based algorithm can utilize the unique multi-qubit gates and analog Hamiltonian dynamics in Rydberg atom arrays. By using device native operations, the HEM-based OLRG leads to the potential of using Rydberg atom arrays to simulate quantum dynamics by directly programming the control parameters using a few high-quality gates.

Following these applications of programmatic representation, we see that thinking from the programmatic perspective allows physicists to open up the imagination beyond simple representations such as matrices and tensors. A good example is the HEM-based OLRG. The representation of an operator does not have to be a matrix but can be any programmatic representation. Thus, our operator map can also be a map from the input Hamiltonian expression to the expression of a pulse sequence. A similar idea might be applied to other problems in quantum many-body physics. Furthermore, in the age of language models, programmatic representations provide a perfect "language" describing the domain physicists are interested in. One may see future applications of training large language models on programmatic representations rather than natural languages, which is almost always ambiguous due to the lack of formal semantics.

To conclude this thesis, in the age of programming, programmatic representation, as a fundamental concept for creating computational processes, will be an essential way to design new software, theories, and methods in quantum many-body physics.

References

[1] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing julia on gpus. *CoRR*, abs/1712.03112, 2017.

[2] Guillermo García-Pérez, Matteo A. C. Rossi, and Sabrina Maniscalco. Ibm q experience as a versatile experimental testbed for simulating open quantum systems, 2019.

[3] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016.

[4] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[5] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):1–43, 2013.

[6] Valérie Pascual and Laurent Hascoët. Tapenade for c. In *Advances in Automatic Differentiation*, pages 199–209. Springer, 2008.

[7] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[8] Reversediff.jl: Reverse mode automatic differentiation for julia. https://github.com/JuliaDiff/ReverseDiff.jl.

[9] Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer,

F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, volume 32, pages 8024–8035. Curran Associates, Inc., 2019.

[10] James Bradbury, Roy Frostig, Peter Hawkins, et al. JAX: composable transformations of Python+NumPy programs, 2018.

[11] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.

[12] Hao Xie, Jin-Guo Liu, and Lei Wang. Automatic differentiation of dominant eigensolver and its applications in quantum physics. *Physical Review B*, 101(24):245139, 2020.

[13] William S. Moses, Valentin Churavy, Ludger Paehler, et al. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Patrick Kidger. On neural differential equations. *arXiv preprint arXiv: 2202.02435*, 2022.

[15] Alexander Altland and Ben D Simons. *Condensed matter field theory*. Cambridge university press, 2010.

[16] Werner Heisenberg. *Zur theorie des ferromagnetismus*. Springer, 1985.

[17] Garnet Kin-Lic Chan and Sandeep Sharma. The density matrix renormalization group in quantum chemistry. *Annual review of physical chemistry*, 62:465–481, 2011.

[18] Philip W Anderson. More is different: Broken symmetry and the nature of the hierarchical structure of science. *Science*, 177(4047):393–396, 1972.

[19] Alexei Yu Kitaev, Alexander Shen, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.

[20] Sevag Gharibian, Yichen Huang, Zeph Landau, et al. Quantum hamiltonian complexity. *Foundations and Trends® in Theoretical Computer Science*, 10(3):159–282, 2015.

[21] Anders W Sandvik. Stochastic series expansion method with operator-loop update. *Physical Review B*, 59(22):R14157, 1999.

[22] Anders W Sandvik. Stochastic series expansion methods. *arXiv preprint arXiv:1909.10591*, 2019.

[23] WMC Foulkes, Lubos Mitas, RJ Needs, and Guna Rajagopal. Quantum monte carlo simulations of solids. *Reviews of Modern Physics*, 73(1):33, 2001.

[24] KA Brueckner. Many-body problem for strongly interacting particles. ii. linked cluster expansion. *Physical Review*, 100(1):36, 1955.

[25] Ann B. Kallin, Katharine Hyatt, Rajiv R. P. Singh, and Roger G. Melko. Entanglement at a two-dimensional quantum critical point: A numerical linked-cluster expansion study. *Phys. Rev. Lett.*, 110:135702, Mar 2013.

[26] Kenneth G Wilson. The renormalization group: Critical phenomena and the kondo problem. *Reviews of modern physics*, 47(4):773, 1975.

[27] Steven R White. Density matrix formulation for quantum renormalization groups. *Physical review letters*, 69(19):2863, 1992.

[28] Steven R. White. Density-matrix algorithms for quantum renormalization groups. *Physical review. B, Condensed matter*, 48 14:10345–10356, 1993.

[29] William Lauchlin McMillan. Ground state of liquid he 4. *Physical Review*, 138(2A):A442, 1965.

[30] Federico Becca and Sandro Sorella. *Quantum Monte Carlo Approaches for Correlated Systems*. Cambridge University Press, 2017.

[31] Richard P Feynman et al. Simulating physics with computers. *Int. j. Theor. phys*, 21(6/7), 2018.

[32] A Yu Kitaev. Quantum measurements and the abelian stabilizer problem. *arXiv preprint quant-ph/9511026*, 1995.

[33] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge university press, 2010.

[34] Andrew M Childs. Universal computation by quantum walk. *Physical review letters*, 102(18):180501, 2009.

[35] András Gilyén, Yuan Su, Guang Hao Low, et al. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 193–204, 2019.

[36] Jeongwan Haah, Matthew B. Hastings, Robin Kothari, et al. Quantum algorithm for simulating real time evolution of lattice hamiltonians. *SIAM Journal on Computing*, 52(6):FOCS18–250–FOCS18–284, 2023.

[37] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, et al. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5(1):4213, 2014.

[38] Dave Wecker, Matthew B Hastings, and Matthias Troyer. Progress towards practical quantum variational algorithms. *Physical Review A*, 92(4):042303, 2015.

[39] Jarrod R McClean, Jonathan Romero, Ryan Babbush, et al. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[40] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[41] Hannes Bernien, Sylvain Schwartz, Alexander Keesling, Harry Levine, Ahmed Omran, Hannes Pichler, Soonwon Choi, Alexander S Zibrov, Manuel Endres, Markus Greiner, et al. Probing many-body dynamics on a 51-atom quantum simulator. *Nature*, 551(7682):579–584, 2017.

[42] Dolev Bluvstein, Ahmed Omran, Harry Levine, Alexander Keesling, Giulia Semeghini, Sepehr Ebadi, Tout T Wang, Alexios A Michailidis, Nishad Maskara, Wen Wei Ho, et al. Controlling quantum many-body dynamics in driven rydberg atom arrays. *Science*, 371(6536):1355–1359, 2021.

[43] Bob Coecke and Ross Duncan. A graphical calculus for quantum observables. *Preprint*, 2007.

[44] Zhengwei Liu, Alex Wozniakowski, and Arthur M Jaffe. Quon 3d language for quantum information. *Proceedings of the National Academy of Sciences*, 114(10):2497–2502, 2017.

[45] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, et al. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[46] Alexander McCaskey and Thien Nguyen. A mlir dialect for quantum assembly languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 255–264. IEEE, 2021.

[47] Michael A Harrison. *Introduction to formal language theory*. Addison-Wesley Longman Publishing Co., Inc., 1978.

[48] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[49] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[50] Jeff Bezanson, Alan Edelman, Stefan Karpinski, et al. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[51] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[52] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[53] Ross Tate, Michael Stepp, Zachary Tatlock, et al. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[54] Joanna A Ellis-Monaghan and Iain Moffatt. *Graphs on surfaces: dualities, polynomials, and knots*, volume 84. Springer, 2013.

[55] Lyndon Evans and Philip Bryant. Lhc machine. *Journal of instrumentation*, 3(08):S08001, 2008.

[56] Haohuan Fu, Junfeng Liao, Jinzhe Yang, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59:1–16, 2016.

[57] Scott Atchley, Christopher Zimmer, John Lange, et al. Frontier: Exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[58] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, X. Li, M. Caricato, A. V. Marenich, J. Bloino, B. G. Janesko, R. Gomperts, B. Mennucci, H. P. Hratchian, J. V. Ortiz, A. F. Izmaylov, J. L. Sonnenberg, D. Williams-Young, F. Ding, F. Lipparini, F. Egidi, J. Goings, B. Peng, A. Petrone, T. Henderson, D. Ranasinghe, V. G. Zakrzewski, J. Gao, N. Rega, G. Zheng, W. Liang, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, K. Throssell, J. A. Montgomery, Jr., J. E. Peralta, F. Ogliaro, M. J. Bearpark, J. J. Heyd, E. N. Brothers, K. N. Kudin, V. N. Staroverov, T. A. Keith, R. Kobayashi, J. Normand, K. Raghavachari, A. P. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, J. M. Millam, M. Klene, C. Adamo, R. Cammi, J. W. Ochterski, R. L. Martin, K. Morokuma, O. Farkas, J. B. Foresman, and D. J. Fox. Gaussian~16 Revision C.01, 2016. Gaussian Inc. Wallingford CT.

[59] Jarrod R. McClean et al. Openfermion: The electronic structure package for quantum computers, 2017.

[60] Alexander Gaenko, Andrey E Antipov, G Carcassi, T Chen, X Chen, Qiaoyuan Dong, Lukas Gamper, Jan Gukelberger, Ryo Igarashi, Sergei Iskakov, et al. Updated core libraries of the alps project. *Computer Physics Communications*, 213:235–251, 2017.

[61] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[62] Jin-Guo Liu, Lei Wang, and Pan Zhang. Tropical tensor network for ground states of spin glasses. *Physical Review Letters*, 126(9):090506, 2021.

[63] Jeff Bezanson, Stefan Karpinski, Viral B Shah, et al. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[64] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[65] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[66] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[67] Edward Farhi and Hartmut Neven. Classification with quantum neural networks on near term processors. *arXiv e-prints*, page arXiv:1802.06002, February 2018.

[68] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, et al. Quantum circuit learning. *Physical Review A*, 98(3):032309, 2018.

[69] Marcello Benedetti, Delfina Garcia-Pintos, Oscar Perdomo, et al. A generative modeling approach for benchmarking and training shallow quantum circuits. *npj Quantum Information*, 5(1), May 2019.

[70] Jin-Guo Liu and Lei Wang. Differentiable learning of quantum circuit born machines. *Physical Review A*, 98(6):062324, 2018.

[71] Peter JJ O'Malley et al. Scalable quantum simulation of molecular energies. *Physical Review X*, 6(3):031007, 2016.

[72] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242, 2017.

[73] Vojtěch Havlíček, Antonio D Córcoles, Kristan Temme, et al. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209, 2019.

[74] Daiwei Zhu et al. Training of quantum circuits on a hybrid quantum computer. *Science Advances*, 5(10):eaaw9918, 2019.

[75] G. Pagano, A. Bapat, P. Becker, et al. Quantum Approximate Optimization with a Trapped-Ion Quantum Simulator. 2019.

[76] Vicente Leyton-Ortega, Alejandro Perdomo-Ortiz, and Oscar Perdomo. Robust implementation of generative modeling with parametrized quantum circuits. *arXiv preprint arXiv:1901.08047*, 2019.

[77] Jarrod R. McClean, Sergio Boixo, Vadim N. Smelyanskiy, et al. Barren plateaus in quantum neural network training landscapes. *Nat. Commun.*, 9(1):4812, 2018.

[78] Differentiable Programming. https://en.wikipedia.org/wiki/Differentiable_programming.

[79] Karpathy, Andrej. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35.

[80] Tianqi Chen et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[81] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

[82] Michael Innes, Elliot Saba, Keno Fischer, et al. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018.

[83] Mike Innes, Alan Edelman, Keno Fischer, et al. Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

[84] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, Nov 1973.

[85] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, et al. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, volume 48, pages 333–342. ACM, 2013.

[86] Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *arXiv preprint arXiv:1612.08091*, 2016.

[87] Krysta Svore, Martin Roetteler, Alan Geller, et al. Q#: Enabling scalable quantum computing and development with a high-level dsl. *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*, 2018.

[88] Cirq: A Python framework for creating, editing, and invoking noisy intermediate scale quantum (NISQ) circuits. https://github.com/quantumlib/Cirq.

[89] qulacs: Variational Quantum Circuit Simulator for Quantum Computation Research. https://github.com/qulacs/qulacs.

[90] Ville Bergholm, Josh Izaac, Maria Schuld, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[91] Héctor Abraham et al. Qiskit: An open-source framework for quantum computing, 2019.

[92] Tyson Jones, Anna Brown, Ian Bush, et al. Quest and high performance simulation of quantum computers. *Scientific Reports*, 9(1), Jul 2019.

[93] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. Open source software in quantum computing. *PloS one*, 13(12):e0208561, 2018.

[94] Ryan LaRose. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum*, 3:130, March 2019.

[95] Marcello Benedetti, Erika Lloyd, Stefan Sack, et al. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4(4):043001, nov 2019.

[96] D Coppersmith. An approximate fourier transform useful in quantum computing. Technical report, Technical report, IBM Research Division, 1994.

[97] Artur Ekert and Richard Jozsa. Quantum computation and shor's factoring algorithm. *Reviews of Modern Physics*, 68(3):733, 1996.

[98] Richard Jozsa. Quantum algorithms and the fourier transform. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):323–337, 1998.

[99] William James Huggins, Piyush Patil, Bradley Mitchell, et al. Towards quantum machine learning with tensor networks. *Quantum Science and Technology*, 2018.

[100] Jin-Guo Liu, Liang Mao, Pan Zhang, et al. Solving quantum statistical mechanics with variational autoregressive networks and quantum circuits. 2019.

[101] Frederica Darema, David A George, V Alan Norton, et al. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[102] Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture., 2023.

[103] Shashi Gowda, Yingbo Ma, Alessandro Cheli, et al. High-performance symbolic-numerics via multiple dispatch. *ACM Commun. Comput. Algebra*, 55(3):92–96, jan 2022.

[104] Jonathan Wurtz, Alexei Bylinskii, Boris Braverman, et al. Aquila: Quera's 256-qubit neutral-atom quantum computer. *arXiv preprint arXiv:2306.11727*, 2023.

[105] Dolev Bluvstein, Harry Levine nAff, Giulia Semeghini, et al. A quantum processor based on coherent transport of entangled atom arrays. *Nature*, 2022.

[106] Hannes Pichler, Sheng-Tao Wang, Leo Zhou, Soonwon Choi, and Mikhail D. Lukin. Quantum optimization for maximum independent set using rydberg atom arrays. *arXiv preprint arXiv: 1808.10816*, 2018.

[107] Maksym Serbyn, Dmitry A Abanin, and Zlatko Papić. Quantum many-body scars and weak breaking of ergodicity. *Nature Physics*, 17(6):675–685, 2021.

[108] Ron Cytron, Jeanne Ferrante, Barry K Rosen, et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[109] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[110] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, et al. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1-2):171–190, 2000.

[111] C. Gidney. Stim: a fast stabilizer circuit simulator. *QUANTUM*, 2021.

[112] Peter J Karalekas, Nikolas A Tezak, Eric C Peterson, et al. A quantum-classical cloud platform optimized for variational hybrid algorithms. *Quantum Science and Technology*, 5(2):024003, apr 2020.

[113] Jitse Niesen and Will M Wright. Algorithm 919: A krylov subspace algorithm for evaluating the ϕ-functions appearing in exponential integrators. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):1–19, 2012.

[114] Krylovkit.jl: Krylov methods for linear problems, eigenvalues, singular values and matrix functions. https://github.com/Jutho/KrylovKit.jl.

[115] General Permutation Matrix. https://en.wikipedia.org/wiki/Generalized_permutation_matrix.

[116] A luxury sparse matrix package for julia. https://github.com/QuantumBFS/LuxurySparse.jl.

[117] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[118] S. Ebadi, A. Keesling, M. Cain, et al. Quantum optimization of maximum independent set using rydberg atom arrays. *Science*, 376(6598):1209–1215, 2022.

[119] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, et al. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153), 2018.

[120] Aidan N Gomez, Mengye Ren, Raquel Urtasun, et al. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pages 2214–2224, 2017.

[121] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, et al. Neural ordinary differential equations. In *Advances in neural information processing systems*, volume 31, pages 6571–6583, 2018.

[122] Akira Hirose. *Complex-valued neural networks: theories and applications*, volume 5. World Scientific, 2003.

[123] Mike Giles. An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. Technical report, 2008.

[124] Gavin E Crooks. Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition.

[125] Jun Li, Xiaodong Yang, Xinhua Peng, et al. Hybrid quantum-classical approach to quantum optimal control. *Phys. Rev. Lett.*, 118:150503, Apr 2017.

[126] Maria Schuld, Ville Bergholm, Christian Gogolin, et al. Evaluating analytic gradients on quantum hardware. *Phys. Rev. A*, 99(3):032331, 2019.

[127] Ken M Nakanishi, Keisuke Fujii, and Synge Todo. Sequential minimal optimization for quantum-classical hybrid algorithms.

[128] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, et al. Monte carlo gradient estimation in machine learning.

[129] Chun-Liang Li, Wei-Cheng Chang, Yu Cheng, et al. MMD GAN: Towards Deeper Understanding of Moment Matching Network.

[130] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, et al. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[131] Statically sized arrays for Julia. https://github.com/JuliaArrays/StaticArrays.jl.

[132] Thomas Häner and Damian S Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 33. ACM, 2017.

[133] Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[134] Edwin Pednault, John A Gunnels, Giacomo Nannicini, et al. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, 2017.

[135] Fang Zhang et al. Alibaba cloud quantum development kit: Large-scale classical simulation of quantum circuits. *arXiv preprint arXiv:1907.11217*, 2019.

[136] Pyquest-cffi: A python interface to the quest quantum simulator (cffi based). https://github.com/HQSquantumsimulations/PyQuEST-cffi.

[137] PennyLane is a cross-platform Python library for quantum machine learning, automatic differentiation, and optimization of hybrid quantum-classical computations. https://github.com/XanaduAI/pennylane.

[138] Review of PennyLane benchmark. https://github.com/Roger-luo/quantum-benchmarks/pull/7.

[139] Aer is a high performance simulator for quantum circuits that includes noise models. https://github.com/Qiskit/qiskit-aer.

[140] Terra provides the foundations for Qiskit. It allows the user to write quantum circuits easily, and takes care of the constraints of real hardware. https://github.com/Qiskit/qiskit-terra.

[141] py.test fixture for benchmarking code. https://github.com/ionelmc/pytest-benchmark.

[142] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295*, 2016.

[143] Benchmarking Quantum Circuit Emulators For Your Daily Research Usage. https://github.com/Roger-luo/quantum-benchmarks.

[144] J Robert Johansson, Paul D Nation, and Franco Nori. Qutip: An open-source python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 183(8):1760–1772, 2012.

[145] Gleb Kalachev, Pavel Panteleev, and Man-Hong Yung. Multi-tensor contraction for xeb verification of quantum circuits. *arXiv preprint arXiv: 2108.05665*, 2021.

[146] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[147] Elliott H Lieb and Derek W Robinson. The finite group velocity of quantum spin systems. *Communications in mathematical physics*, 28(3):251–257, 1972.

[148] Matthew B. Hastings and Tohru Koma. Spectral gap and exponential decay of correlations. *Communications in Mathematical Physics*, 265:781–804, 2006.

[149] Zhiyuan Wang, Michael Foss-Feig, and Kaden RA Hazzard. Bounding the finite-size error of quantum many-body dynamics simulations. *Physical Review Research*, 3(3):L032047, 2021.

[150] Ulrich Schollwöck. The density-matrix renormalization group. *Reviews of modern physics*, 77(1):259, 2005.

[151] Roger Penrose et al. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971.

[152] David Elieser Deutsch. Quantum computational networks. *Proceedings of the royal society of London. A. mathematical and physical sciences*, 425(1868):73–90, 1989.

[153] Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of physics*, 326(1):96–192, 2011.

[154] Steven R White and Adrian E Feiguin. Real-time evolution using the density matrix renormalization group. *Physical review letters*, 93(7):076401, 2004.

[155] Sebastian Paeckel, Thomas Köhler, Andreas Swoboda, et al. Time-evolution methods for matrix-product states. *Annals of Physics*, 411:167998, 2019.

[156] Glen Evenbly and Guifré Vidal. Tensor network states and geometry. *Journal of Statistical Physics*, 145:891–918, 2011.

[157] Michael P Zaletel and Frank Pollmann. Isometric tensor network states in two dimensions. *Physical review letters*, 124(3):037201, 2020.

[158] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of physics*, 349:117–158, 2014.

[159] Guifre Vidal. Entanglement renormalization: an introduction. *arXiv preprint arXiv:0912.1651*, 2009.

[160] G. Vidal. Class of quantum many-body states that can be efficiently simulated. *Phys. Rev. Lett.*, 101:110501, Sep 2008.

[161] Frank Verstraete, Valentin Murg, and J Ignacio Cirac. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Advances in physics*, 57(2):143–224, 2008.

[162] Guifré Vidal. Efficient simulation of one-dimensional quantum many-body systems. *Phys. Rev. Lett.*, 93:040502, Jul 2004.

[163] F. Verstraete, J. J. García-Ripoll, and J. I. Cirac. Matrix product density operators: Simulation of finite-temperature and dissipative systems. *Phys. Rev. Lett.*, 93:207204, Nov 2004.

[164] Michael P. Zaletel, Roger S. K. Mong, Christoph Karrasch, et al. Time-evolving a matrix product state with long-ranged interactions. *Phys. Rev. B*, 91:165112, Apr 2015.

[165] Xun Gao and Lu-Ming Duan. Efficient representation of quantum many-body states with deep neural networks. *Nature communications*, 8(1):662, 2017.

[166] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

[167] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, et al. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

[168] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on learning theory*, pages 907–940. PMLR, 2016.

[169] Valentin Khrulkov, Alexander Novikov, and Ivan Oseledets. Expressive power of recurrent neural networks. *arXiv preprint arXiv:1711.00811*, 2017.

[170] Maithra Raghu, Ben Poole, Jon Kleinberg, et al. On the expressive power of deep neural networks. In *international conference on machine learning*, pages 2847–2854. PMLR, 2017.

[171] Jing Chen, Song Cheng, Haidong Xie, et al. Equivalence of restricted boltzmann machines and tensor network states. *Phys. Rev. B*, 97:085104, Feb 2018.

[172] Sujie Li, Feng Pan, Pengfei Zhou, et al. Boltzmann machines as two-dimensional tensor networks. *Physical Review B*, 104(7):075154, 2021.

[173] Gerald Knizia and Garnet Kin-Lic Chan. Density matrix embedding: A strong-coupling quantum embedding theory. *Journal of Chemical Theory and Computation*, 9(3):1428–1432, 2013. PMID: 26587604.

[174] Sebastian Wouters, Carlos A. Jiménez-Hoyos, Qiming Sun, et al. A practical guide to density matrix embedding theory in quantum chemistry. *Journal of Chemical Theory and Computation*, 12(6):2706–2719, 2016. PMID: 27159268.

[175] Qiming Sun and Garnet Kin-Lic Chan. Quantum embedding theories. *Accounts of Chemical Research*, 49(12):2705–2712, 2016. PMID: 27993005.

[176] Urs Muller, Jan Ben, Eric Cosatto, et al. Off-road obstacle avoidance through end-to-end learning. *Advances in neural information processing systems*, 18, 2005.

[177] Felp Roza. End-to-end learning, the (almost) every purpose ml method. Medium, 2019.

[178] Ronan Collobert, Jason Weston, Léon Bottou, et al. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[179] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[180] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[181] David Silver, Julian Schrittwieser, Karen Simonyan, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[182] Dolev Bluvstein, Simon J Evered, Alexandra A Geim, et al. Logical quantum processor based on reconfigurable atom arrays. *Nature*, pages 1–3, 2023.

[183] Andrew J Daley, Immanuel Bloch, Christian Kokail, et al. Practical quantum advantage in quantum simulation. *Nature*, 607(7920):667–676, 2022.

[184] Steven R White and Reinhard M Noack. Real-space quantum renormalization groups. *Physical review letters*, 68(24):3487, 1992.

[185] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, et al. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[186] Wikipedia. Duck typing — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Duck%20typing&oldid=1207784910, 2024. [Online; accessed 22-February-2024].

[187] Xiuzhe(Roger) Luo. A software package for operator learning renormalization group. https://github.com/Roger-luo/olrg-teal, 2024.

[188] James R. Garrison and Ryan V. Mishmash. Simple dmrg 1.0, November 2017.

[189] B Pirvu, V Murg, J I Cirac, et al. Matrix product operator representations. *New Journal of Physics*, 12(2):025012, feb 2010.

[190] C. Hubig, I. P. McCulloch, and U. Schollwöck. Generic construction of efficient matrix product operators. *Phys. Rev. B*, 95:035129, Jan 2017.

[191] Andrew M. Childs, Yuan Su, Minh C. Tran, et al. A theory of trotter error. *arXiv preprint arXiv: Arxiv-1912.08854*, 2019.

[192] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. The ITensor Software Library for Tensor Network Calculations. *SciPost Phys. Codebases*, page 4, 2022.

[193] Brian C Hall and Brian C Hall. *Lie groups, Lie algebras, and representations*. Springer, 2013.

[194] Wulf Rossmann. *Lie groups: an introduction through linear groups*, volume 5. Oxford University Press on Demand, 2006.

[195] J. Von Neumann. Some matrix-inequalities and metrization of matrix-space. *Tomsk Univ*, Rev. 1(286-300), 1937. Reprinted in Collected Works (Pergamon Press, 1962), iv, 205-219.

[196] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd ICLR 2015, Conference Track Proceedings*, 2015.

[197] William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. volume 33, 2020.

[198] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *arXiv preprint arXiv:1810.07951*, 2018.

[199] Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[200] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

[201] Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, et al. Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design. *Quantum*, 4:341, October 2020.

[202] Mike Giles. An extended collection of matrix derivative results for forward and reverse mode automatic differentiation, 2008.

[203] Matthias Seeger, Asmus Hetzel, Zhenwen Dai, et al. Auto-differentiating linear algebra. *arXiv preprint arXiv:1710.08717*, 2017.

[204] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[205] P Kramer. A review of the time-dependent variational principle. In *Journal of Physics: Conference Series*, volume 99, page 012009. IOP Publishing, 2008.

[206] J Broeckhove, L Lathouwers, E Kesteloot, et al. On the equivalence of time-dependent variational principles. *Chemical physics letters*, 149(5-6):547–550, 1988.

[207] Xiao Yuan, Suguru Endo, Qi Zhao, et al. Theory of variational quantum simulation. *Quantum*, 3:191, October 2019.

[208] Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, 2017.

[209] Giuseppe Carleo, Federico Becca, Marco Schiró, et al. Localization and glassy dynamics of many-body quantum systems. *Scientific reports*, 2(1):243, 2012.

[210] Giuseppe Carleo, Federico Becca, Laurent Sanchez-Palencia, et al. Light-cone effect and supersonic correlations in one-and two-dimensional bosonic superfluids. *Physical Review A*, 89(3):031602, 2014.

[211] Jan Hermann, Zeno Schätzle, and Frank Noé. Deep-neural-network solution of the electronic schrödinger equation. *Nature Chemistry*, 12(10):891–897, 2020.

[212] Giacomo Torlai, Guglielmo Mazzola, Juan Carrasquilla, et al. Neural-network quantum state tomography. *Nature Physics*, 14(5):447–450, 2018.

[213] Zi Cai and Jinguo Liu. Approximating quantum many-body wave functions using artificial neural networks. *Phys. Rev. B*, 97:035116, Jan 2018.

[214] Juan Carrasquilla and Giacomo Torlai. How to use neural networks to investigate quantum many-body physics. *PRX Quantum*, 2(4):040201, 2021.

[215] Mohamed Hibat-Allah, Martin Ganahl, Lauren E Hayward, et al. Recurrent neural network wave functions. *Physical Review Research*, 2(2):023358, 2020.

[216] Jannes Nys, Zakari Denis, and Giuseppe Carleo. Real-time quantum dynamics of thermal states with neural thermofields. *arXiv preprint arXiv: 2309.07063*, 2023.

[217] Di Luo, Zhuo Chen, Juan Carrasquilla, et al. Autoregressive neural network for simulating open quantum systems via a probabilistic formulation. *Phys. Rev. Lett.*, 128:090501, Feb 2022.

[218] Michael J Hartmann and Giuseppe Carleo. Neural-network approach to dissipative quantum many-body dynamics. *Physical review letters*, 122(25):250502, 2019.

[219] Kenny Choo, Titus Neupert, and Giuseppe Carleo. Two-dimensional frustrated j 1- j 2 model studied with neural network quantum states. *Physical Review B*, 100(12):125124, 2019.

[220] Giuseppe Carleo, Yusuke Nomura, and Masatoshi Imada. Constructing exact representations of quantum many-body systems with deep neural networks. *Nature communications*, 9(1):5322, 2018.

[221] Kenny Choo, Antonio Mezzacapo, and Giuseppe Carleo. Fermionic neural-network states for ab-initio electronic structure. *Nature communications*, 11(1):2368, 2020.

[222] Alessandro Sinibaldi, Clemens Giuliani, Giuseppe Carleo, et al. Unbiasing time-dependent variational monte carlo by projected quantum evolution. *arXiv preprint arXiv:2305.14294*, 2023.

[223] Zhuo Chen, Laker Newhouse, Eddie Chen, et al. Autoregressive neural tensornet: Bridging neural networks and tensor networks for quantum many-body simulation. *NEURIPS*, 2023.

[224] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[225] Diederik P. Kingma and M. Welling. Auto-encoding variational bayes. *International Conference on Learning Representations*, 2013.

[226] Esteban G Tabak and Cristina V Turner. A family of nonparametric density estimation algorithms. *Communications on Pure and Applied Mathematics*, 66(2):145–164, 2013.

[227] Papamakarios George, Nalisnick Eric, Rezende Danilo Jimenez, et al. Normalizing flows for probabilistic modeling and inference. *The Journal of Machine Learning Research*, 2021.

[228] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, et al. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015.

[229] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851. Curran Associates, Inc., 2020.

[230] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):322–333, 1969.

[231] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In Shun-ichi Amari and Michael A. Arbib, editors, *Competition and Cooperation in Neural Nets*, pages 267–285, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[232] L. S. Pontryagin et al. *The mathematical theory of optimal processes.* International Series of Monographs in Pure and Applied Mathematics. Interscience, New York., 1962.

[233] William W. Hager. Runge-kutta methods in optimal control and the transformed adjoint system, December 2000.

[234] *CVODES: The Sensitivity-Enabled ODE Solver in SUNDIALS*, volume Volume 6: 5th International Conference on Multibody Systems, Nonlinear Dynamics, and Control, Parts A, B, and C of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 09 2005.

[235] Aram W Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203–209, 2017.

[236] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, et al. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595, 2018.

[237] Charles Neill, Pedran Roushan, K Kechedzhi, et al. A blueprint for demonstrating quantum supremacy with superconducting qubits. *Science*, 360(6385):195–199, 2018.

[238] Sergio Blanes and Fernando Casas. *A concise introduction to geometric numerical integration*. CRC press, 2017.

[239] Masuo Suzuki. General theory of fractal path integrals with applications to many-body theories and statistical physics. *Journal of Mathematical Physics*, 32(2):400–407, 1991.

[240] Andrew M. Childs, Yuan Su, Minh C. Tran, et al. Theory of trotter error with commutator scaling. *Phys. Rev. X*, 11:011020, Feb 2021.

[241] Jeongwan Haah, Matthew B Hastings, Robin Kothari, et al. Quantum algorithm for simulating real time evolution of lattice hamiltonians. *SIAM Journal on Computing*, (0):FOCS18–250, 2021.

[242] Dave Wecker, Bela Bauer, Bryan K Clark, et al. Gate-count estimates for performing quantum chemistry on small quantum computers. *Physical Review A*, 90(2):022305, 2014.

[243] Youngseok Kim, Andrew Eddins, Sajant Anand, et al. Evidence for the utility of quantum computing before fault tolerance. *Nature*, 618(7965):500–505, 2023.

[244] Marcello Benedetti, Erika Lloyd, Stefan Sack, et al. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4(4):043001, 2019.

[245] Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti. Structure optimization for parameterized quantum circuits. *Quantum*, 5:391, 2021.

[246] Sukin Sim, Peter D Johnson, and Alán Aspuru-Guzik. Expressibility and entangling capability of parameterized quantum circuits for hybrid quantum-classical algorithms. *Advanced Quantum Technologies*, 2(12):1900070, 2019.

[247] Tobias Haug, Kishor Bharti, and MS Kim. Capacity and quantum geometry of parametrized quantum circuits. *PRX Quantum*, 2(4):040309, 2021.

[248] Tobias Haug and MS Kim. Natural parametrized quantum circuit. *Physical Review A*, 106(5):052611, 2022.

[249] Jules Tilly, Hongxiang Chen, Shuxiang Cao, et al. The variational quantum eigensolver: a review of methods and best practices. *Physics Reports*, 986:1–128, 2022.

[250] Jin-Guo Liu, Yi-Hong Zhang, Yuan Wan, et al. Variational quantum eigensolver with fewer qubits. *Phys. Rev. Research*, 1(2):023025, Sep 2019.

[251] Daochen Wang, Oscar Higgott, and Stephen Brierley. Accelerated variational quantum eigensolver. *Physical review letters*, 122(14):140504, 2019.

[252] Ya Wang, Florian Dolde, Jacob Biamonte, et al. Quantum simulation of helium hydride cation in a solid-state spin register. *ACS nano*, 9(8):7769–7774, 2015.

[253] Yangchao Shen, Xiang Zhang, Shuaining Zhang, et al. Quantum implementation of the unitary coupled cluster for simulating molecular electronic structure. *Physical Review A*, 95(2):020501, 2017.

[254] Stefano Paesani, Andreas A Gentile, Raffaele Santagati, et al. Experimental bayesian quantum phase estimation on a silicon photonic chip. *Physical review letters*, 118(10):100503, 2017.

[255] James I Colless, Vinay V Ramasesh, Dar Dahlen, et al. Computation of molecular spectra on a quantum processor with an error-resilient algorithm. *Physical Review X*, 8(1):011021, 2018.

[256] Raffaele Santagati, Jianwei Wang, Antonio A Gentile, et al. Witnessing eigenstates for quantum simulation of hamiltonian spectra. *Science advances*, 4(1):eaap9646, 2018.

[257] Abhinav Kandala, Kristan Temme, Antonio D Córcoles, et al. Error mitigation extends the computational reach of a noisy quantum processor. *Nature*, 567(7749):491–495, 2019.

[258] Cornelius Hempel, Christine Maier, Jonathan Romero, et al. Quantum chemistry calculations on a trapped-ion quantum simulator. *Physical Review X*, 8(3):031022, 2018.

[259] Christian Kokail, Christine Maier, Rick van Bijnen, et al. Self-verifying variational quantum simulation of lattice models. *Nature*, 569(7756):355–360, 2019.

[260] Ying Li and Simon C Benjamin. Efficient variational quantum simulator incorporating active error minimization. *Physical Review X*, 7(2):021050, 2017.

[261] Ken M Nakanishi, Kosuke Mitarai, and Keisuke Fujii. Subspace-search variational quantum eigensolver for excited states. *Physical Review Research*, 1(3):033062, 2019.

[262] Suguru Endo, Jinzhao Sun, Ying Li, et al. Variational quantum simulation of general processes. *Phys. Rev. Lett.*, 125:010501, Jun 2020.

[263] Adrian Parra-Rodriguez, Pavel Lougovski, Lucas Lamata, et al. Digital-analog quantum computation. *Physical Review A*, 101(2):022305, 2020.

[264] Ana Martin, Lucas Lamata, Enrique Solano, et al. Digital-analog quantum algorithm for the quantum fourier transform. *Phys. Rev. Res.*, 2:013012, Jan 2020.

[265] Jonathan Z. Lu, Lucy Jiao, Kristina Wolinski, et al. Digital-analog quantum learning on rydberg atom arrays. *arXiv preprint arXiv: 2401.02940*, 2024.

[266] F. Darema, D.A. George, V.A. Norton, et al. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[267] Leonardo Banchi and Gavin E Crooks. Measuring analytic gradients of general quantum evolution with the stochastic parameter shift rule. *Quantum*, 5:386, 2021.

[268] David Wierichs, J. Izaac, C. Wang, et al. General parameter-shift rules for quantum gradients. *QUANTUM*, 2021.

[269] Jiaqi Leng, Yuxiang Peng, Yi-Ling Qiao, et al. Differentiable analog quantum computing for optimization and control. *Advances in Neural Information Processing Systems*, 35:4707–4721, 2022.

[270] Harry Levine, Alexander Keesling, Giulia Semeghini, et al. Parallel implementation of high-fidelity multiqubit gates with neutral atoms. *Physical review letters*, 123(17):170503, 2019.

[271] Sumeet Khatri, Ryan LaRose, Alexander Poremba, et al. Quantum-assisted quantum compiling. *Quantum*, 3:140, May 2019.

[272] Tobias Glasmachers. Limits of end-to-end learning. In *Asian conference on machine learning*, pages 17–32. PMLR, 2017.

[273] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[274] Edwin M Stoudenmire and Steven R White. Studying two-dimensional systems with the density matrix renormalization group. *Annu. Rev. Condens. Matter Phys.*, 3(1):111–128, 2012.

[275] Ming-Jay Yang and Steven R. White. Time-dependent variational principle with ancillary krylov subspace. *Physical Review B*, 102, 2020.

[276] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *International Conference on Learning Representations*, 2017.

[277] Jonathan Heek, Anselm Levskaya, Avital Oliver, et al. Flax: A neural network library and ecosystem for JAX, 2023.

[278] DeepMind, Igor Babuschkin, Kate Baumli, et al. The DeepMind JAX Ecosystem, 2020.

[279] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[280] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*, 6(65):3349, 2021.

[281] Stefanie Czischek, M. Schuyler Moss, Matthew Radzihovsky, et al. Data-enhanced variational monte carlo simulations for rydberg atom arrays. *Phys. Rev. B*, 105:205108, May 2022.

[282] M. Schuyler Moss, Sepehr Ebadi, Tout T. Wang, et al. Enhancing variational monte carlo using a programmable quantum simulator. *arXiv preprint arXiv: 2308.02647*, 2023.

[283] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[284] YaoLang: The next DSL for Yao and quantum programs. https://github.com/QuantumBFS/YaoLang.jl.

[285] J. Bezanson. "Why is Julia fast? Can it be faster?" 2015, JuliaCon India. https://www.youtube.com/watch?v=xUP3cSKb8sI.

[286] Xiu-Zhe Luo, Di Luo, and Roger G. Melko. Operator learning renormalization group. *arXiv preprint arXiv: 2403.03199*, 2024.

[287] Bidirectional transformation between Yao Quantum Block IR and QASM. https://github.com/QuantumBFS/YaoQASM.jl.

[288] Multi-language suite for high-performance solvers of differential equations. https://github.com/JuliaDiffEq/DifferentialEquations.jl.

[289] Symbolics.jl: A symbolic math library written in Julia modelled off scmutils. https://github.com/MasonProtter/Symbolics.jl.

[290] SymEngine is a fast symbolic manipulation library, written in C++. https://github.com/symengine/symengine.

[291] An efficient symbolic term rewriting engine. https://github.com/HarrisonGrodin/Rewrite.jl.

[292] ZXCalculus.jl: An implementation of ZX-calculus in Julia. https://github.com/QuantumBFS/ZXCalculus.jl.

[293] YaoTensorNetwork: Dump a quantum circuit in Yao to a tensor network graph model. https://github.com/QuantumBFS/YaoTensorNetwork.jl.

[294] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, Proceedings 16th International Conference on *Quantum Physics and Logic,* Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020.

[295] Edward Grant, Marcello Benedetti, Shuxiang Cao, et al. Hierarchical quantum classifiers. *npj Quantum Information*, 4(1):65, 2018.

[296] Dmitri Maslov, Gerhard W Dueck, D Michael Miller, et al. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, 2008.

[297] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, et al. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 866–874. IEEE, 2016.

[298] Vivek V. Shende, Igor L. Markov, and Stephen S. Bullock. Minimal universal two-qubit controlled-not-based circuits. *Phys. Rev. A*, 69:062321, Jun 2004.

[299] Roger B Sidje. Expokit: A software package for computing matrix exponentials. *ACM Transactions on Mathematical Software (TOMS)*, 24(1):130–156, 1998.

[300] Raban Iten, David Sutter, and Stefan Woerner. Efficient template matching in quantum circuits. *arXiv preprint arXiv:1909.05270*, 2019.

[301] Manuel Krebber. Non-linear associative-commutative many-to-one pattern matching with sequence variables. *arXiv preprint arXiv:1705.00907*, 2017.

[302] Xiang Fu et al. eqasm: An executable quantum instruction set architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 224–237. IEEE, 2019.

[303] Adam Paszke, Sam Gross, Soumith Chintala, et al. Automatic differentiation in pytorch. 2017.

[304] James T. Moore and Jonathan F. Bard. The Mixed Integer Linear Bilevel Programming Problem. *Operations Research*, 38(5):911–921, oct 1990.

[305] Aleks Kissinger and John van de Wetering. Reducing T-count with the ZX-calculus. *arXiv preprint arXiv:1903.10477*, 2019.

[306] Michael R Geller, John M Martinis, Andrew T Sornborger, et al. Universal quantum simulation with prethreshold superconducting qubits: Single-excitation subspace method. *Physical Review A*, 91(6):062309, 2015.

[307] Diego Ristè, Marcus P da Silva, Colm A Ryan, et al. Demonstration of quantum advantage in machine learning. *npj Quantum Information*, 3(1):16, 2017.

[308] Julian Kelly et al. State preservation by repetitive error detection in a superconducting quantum circuit. *Nature*, 519(7541):66, 2015.

[309] Jinfeng Zeng, Yufeng Wu, Jin-Guo Liu, et al. Learning and inference on generative adversarial quantum circuits. *Physical Review A*, 99(5):052306, 2019.

[310] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[311] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[312] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[313] Craig Gidney and Martin Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *arXiv preprint arXiv:1905.09749*, 2019.

[314] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.

[315] Nathan Wiebe, Ashish Kapoor, and Krysta M Svore. Quantum deep learning. *arXiv preprint arXiv:1412.3489*, 2014.

[316] Keno Fischer and Elliot Saba. Automatic full compilation of julia programs and ml models to cloud tpus. *arXiv preprint arXiv:1810.09868*, 2018.

[317] Tim Besard, Bjorn De Sutter, Andrés Frías-Velázquez, et al. Case study of multiple trace transform implementations. *The International Journal of High Performance Computing Applications*, 29(4):489–505, 2015.

[318] Mike Innes et al. On machine learning and programming languages. Association for Computing Machinery (ACM), 2018.

[319] Daniel R Simon. On the power of quantum computation. *SIAM journal on computing*, 26(5):1474–1483, 1997.

[320] Peter W Shor. Scheme for reducing decoherence in quantum computer memory. *Physical review A*, 52(4):R2493, 1995.

[321] Bert Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980.

[322] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta informatica*, 10(1):27–52, 1978.

[323] Thomas Häner, Damian S Steiger, Krysta Svore, et al. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, 2018.

[324] Mu Li, Li Zhou, Zichao Yang, et al. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[325] Andrew W Cross, Lev S Bishop, John A Smolin, et al. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[326] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[327] The Q# programming language. https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview.

[328] CuYao.jl: CUDA extension for Yao.jl. https://github.com/QuantumBFS/CuYao.jl.

[329] Andrew Cross. The IBM Q experience and QISKit open-source quantum computing software. In *APS Meeting Abstracts*, 2018.

[330] Nathan Killoran, Josh Izaac, Nicolás Quesada, et al. Strawberry fields: A software platform for photonic quantum computing. *Quantum*, 3:129, 2019.

[331] Dougal Maclaurin. *Modeling, inference and optimization with composable differentiable procedures*. PhD thesis, 2016.

[332] RBNF: A DSL for modern parsing. https://github.com/thautwarm/RBNF.jl.

[333] David E Rumelhart and Geoffrey E Hintonf. Learning representations by back-propagating errors. *Nature*, 323:9, 1986.

[334] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[335] Zhao-Yu Han, Jun Wang, Heng Fan, et al. Unsupervised generative modeling using matrix product states. *Phys. Rev. X*, 8:031012, Jul 2018.

[336] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, et al. Simulation of low-depth quantum circuits as complex undirected graphical models. *arXiv preprint arXiv:1712.05384*, 2017.

[337] Jianxin Chen, Fang Zhang, Mingcheng Chen, et al. Classical simulation of intermediate-size quantum circuits. *arXiv preprint arXiv:1805.01450*, 2018.

[338] Chu Guo, Yong Liu, Min Xiong, et al. General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier. *Phys. Rev. Lett.*, 123:190501, Nov 2019.

[339] Michael Levin and Cody P Nave. Tensor renormalization group approach to two-dimensional classical lattice models. *Physical review letters*, 99(12):120601, 2007.

[340] Song Cheng, Jing Chen, and Lei Wang. Information perspective to probabilistic modeling: Boltzmann machines versus born machines. *Entropy*, 20(8):583, 2018.

[341] E Miles Stoudenmire and David J Schwab. Supervised learning with quantum-inspired tensor networks. *arXiv preprint arXiv:1605.05775*, 2016.

[342] Miriam Backens. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, sep 2014.

[343] Weishi Wang, Jin-Guo Liu, and Lei Wang. A variational quantum state compression algorithm. *to appear*.

[344] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *arXiv preprint quant-ph/0301141*, 2003.

[345] Song Cheng, Lei Wang, Tao Xiang, et al. Tree tensor networks for generative modeling. *Phys. Rev. B*, 99:155131, Apr 2019.

[346] Ivan Glasser, Ryan Sweke, Nicola Pancotti, et al. Expressive power of tensor-network factorizations for probabilistic modeling. In *Advances in Neural Information Processing Systems*, pages 1496–1508, 2019.

[347] Tai-Danae Bradley, E M Stoudenmire, and John Terilla. Modeling sequences with quantum states: a look under the hood. *Machine Learning: Science and Technology*, 1(3):035008, jul 2020.

[348] Feng Pan, Pengfei Zhou, Sujie Li, et al. Contracting arbitrary tensor networks: general approximate algorithm and applications in graphical models and quantum circuit simulations. *arXiv preprint arXiv:1912.03014*, 2019.

[349] T. Altenkirch and J. Grattage. A functional quantum programming language. *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*.

[350] Leonardo Banchi, Nicola Pancotti, and Sougato Bose. Quantum gate learning in qubit networks: Toffoli gate without time-dependent control. *npj Quantum Information*, 2(1), Jul 2016.

[351] Xiaoguang Wang, Zhe Sun, and Z. D. Wang. Operator fidelity susceptibility: An indicator of quantum criticality. *Physical Review A*, 79(1), Jan 2009.

[352] Lukasz Cincio, Yiğit Subaşı, Andrew T Sornborger, et al. Learning the quantum algorithm for state overlap. *New Journal of Physics*, 20(11):113022, Nov 2018.

[353] Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24):615, 2018.

[354] Richard H Byrd, Peihuang Lu, Jorge Nocedal, et al. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[355] Cristina Cirstoiu, Zoe Holmes, Joseph Iosue, et al. Variational fast forwarding for quantum simulation beyond the coherence time, 2019.

[356] Ryan LaRose, Arkin Tikku, Étude O'Neel-Judy, et al. Variational quantum state diagonalization. *npj Quantum Information*, 5(1), Jun 2019.

[357] Piotr Gawron, Dariusz Kurzyk, and Łukasz Pawela. Quantuminformation.jl—a julia package for numerical computation in quantum information theory. *PLOS ONE*, 13(12):e0209358, Dec 2018.

[358] Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Phys. Rev. Lett.*, 122:040504, Feb 2019.

[359] Giulia Semeghini, Harry Levine, Alexander Keesling, et al. Probing topological spin liquids on a programmable quantum simulator. *Science*, 374(6572):1242–1247, 2021.

[360] Xizhi Han, Sean A Hartnoll, Jorrit Kruthoff, et al. Bootstrapping matrix quantum mechanics. *Physical Review Letters*, 125(4):041601, 2020.

[361] David Silver, Thomas Hubert, Julian Schrittwieser, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[362] J Werschnik and EKU Gross. Quantum optimal control theory. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 40(18):R175, 2007.

[363] Sergio Blanes, Fernando Casas, Jose-Angel Oteo, et al. The magnus expansion and some of its applications. *Physics reports*, 470(5-6):151–238, 2009.

[364] Pierre Nataf and Frédéric Mila. Exact diagonalization of heisenberg su (n) models. *Physical review letters*, 113(12):127204, 2014.

[365] Michel Caffarel and Werner Krauth. Exact diagonalization approach to correlated fermions in infinite dimensions: Mott transition and superconductivity. *Physical review letters*, 72(10):1545, 1994.

[366] Daniel Alonso and Inés de Vega. Multiple-time correlation functions for non-markovian interaction: Beyond the quantum regression theorem. *Phys. Rev. Lett.*, 94:200403, May 2005.

[367] Ryogo Kubo. Generalized cumulant expansion method. *Journal of the Physical Society of Japan*, 17(7):1100–1120, 1962.

[368] Sepehr Ebadi, Tout T Wang, Harry Levine, et al. Quantum phases of matter on a 256-atom programmable quantum simulator. *Nature*, 595(7866):227–232, 2021.

[369] Hsin-Yuan Huang, R. Kueng, and J. Preskill. Predicting many properties of a quantum system from very few measurements. *Nature Physics*, 2020.

[370] Hsin-Yuan Huang, Richard Kueng, Giacomo Torlai, et al. Provably efficient machine learning for quantum many-body problems. *Science*, 377(6613):eabk3333, 2022.

[371] Hai-Jun Liao, Jin-Guo Liu, Lei Wang, et al. Differentiable programming tensor networks. *Physical Review X*, 9(3):031041, 2019.

[372] Juan Carrasquilla, Giacomo Torlai, Roger G Melko, et al. Reconstructing quantum states with generative models. *Nature Machine Intelligence*, 1(3):155–161, 2019.

[373] J Ignacio Cirac and Frank Verstraete. Renormalization and tensor product states in spin chains and lattices. *Journal of physics a: mathematical and theoretical*, 42(50):504004, 2009.

[374] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[375] Albert Gordo, Jon Almazan, Jerome Revaud, et al. End-to-end learning of deep visual representations for image retrieval. *International Journal of Computer Vision*, 124(2):237–254, 2017.

[376] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[377] Frank Verstraete, Michael M Wolf, David Perez-Garcia, et al. Criticality, the area law, and the computational power of projected entangled pair states. *Physical review letters*, 96(22):220601, 2006.

[378] Matthew B Hastings. Solving gapped hamiltonians locally. *Physical review b*, 73(8):085115, 2006.

Glossary

differentiable programming A programming paradigm that allows the automatic differentiation of programs 18–20, 71, 79, 129

end-to-end A machine learning paradigm that learns a system from input to output 98, 163, 167, 168

faithful gradient A gradient that can be evalulated on quantum computer 84

syntax sugar A syntactic feature that makes the code easier to read or write 45