

FAKULTÄT FÜR PHYSIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

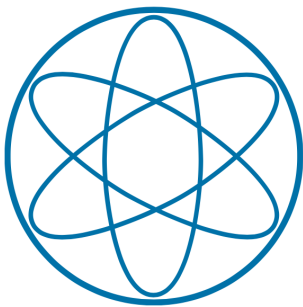
Master's Thesis

Optimizing Tensor Networks with Automatic Differentiation

Optimierung von Tensornetzwerken mit automatischer Differentiation

| | |
|------------------|--------------------------|
| Author: | Jakob Unfried |
| Supervisor: | Prof. Dr. Michael Knap |
| Second Assessor: | Prof. Dr. Frank Pollmann |
| Submission Date: | October 2nd, 2020 |

Chair for Collective Quantum Dynamics (T73)



Abstract

We study gradient-based optimization methods to find optimal tensor network approximations for ground states and time evolution of quantum lattice models. We formulate the desired state as the minimization problem of an appropriate cost function in terms of the (complex-valued) tensors and optimize them using the L-BFGS (limited memory Broyden, Fletcher, Goldfarb, Shanno) algorithm, a quasi Newton method. Gradients of the cost function are computed using automatic differentiation. We benchmark the method by finding projected entangled pair state (PEPS) approximations of ground states of the transverse field Ising model on finite and infinite 2D square lattices. Additionally, we perform time evolution of the PEPS state after a local quench from the ground state. We find, as a proof of principle, that gradient-based optimization is a viable method and discuss strategies to improve computational efficiency and make it competitive to established PEPS methods.

Zusammenfassung

In dieser Abschlussarbeit untersuchen wir gradientenbasierte Optimierungsmethoden, die es erlauben Tensornetzwerke zu finden, die Grundzustände oder zeitentwickelte Zustände von quantenmechanischen Modellen auf Gittern approximieren. Die gewünschten Zustände werden als das Minimierungsproblem einer entsprechend gewählten Kostenfunktion der (komplexwertigen) Tensoren formuliert. Diese Tensoren werden dann mit dem L-BFGS Algorithmus optimiert. Die dafür benötigten Gradienten der Kostenfunktion berechnen wir mithilfe von automatischer Differenzierung. Wir testen die Methode, indem wir PEPS Grundzustände für das Ising Modell im transversen Feld, sowohl für unendliche als auch endliche zweidimensionale Quadratgitter optimieren. Des Weiteren optimieren wir PEPS Zustände für die Zeitentwicklung desselben Modells nach einer lokalen Anregung auf dem endlichen Gitter. Wir zeigen, dass gradientenbasierte Optimierung eine valide und vielversprechende Methode ist und diskutieren Strategien, wie sie weiter verbessert werden kann, um den Rechenaufwand zu reduzieren und eine attraktive Alternative zu bestehenden Tensornetzwerkmethoden zu werden.

Contents

| | | |
|----------|---|------------|
| 1 | Introduction | 1 |
| 2 | Gradient-Based Optimization | 5 |
| 2.1 | Line Search Methods | 7 |
| 2.2 | Gradient Descent and Newton's Method | 13 |
| 2.3 | Quasi-Newton Methods: BFGS and L-BFGS | 14 |
| 2.4 | Functions in Complex Variables | 20 |
| 3 | Automatic Differentiation | 25 |
| 3.1 | Jacobian Vector Products and Vector Jacobian Products | 27 |
| 3.2 | Chained-Primitive Functions | 30 |
| 3.3 | General Computation Graphs | 37 |
| 3.4 | Automatic Differentiation for Complex Functions | 46 |
| 3.5 | Implementation in Jax for Python | 50 |
| 3.6 | Review of Automatic Differentiation Tools | 51 |
| 3.7 | A Short Glossary of Terminology and Jargon | 54 |
| 4 | Tensor Networks and Algorithms | 55 |
| 4.1 | Diagrammatic Notation of Tensor Networks | 56 |
| 4.2 | Projected Entangled Pair States (PEPS) | 57 |
| 4.3 | PEPOs (PEPS Operators) for Nearest Neighbor Operators | 59 |
| 4.4 | PEPOs for Time Evolution Operators | 63 |
| 4.5 | Expectation Values and Matrix Elements in PEPS States | 65 |
| 4.6 | Corner Transfer Matrix Renormalization Group (CTMRG) | 70 |
| 5 | Tensor network Applications of Automatic Differentiation | 75 |
| 5.1 | Ground states | 76 |
| 5.2 | Time Evolution | 77 |
| 5.3 | Automatic Differentiation of PEPS Contractions | 78 |
| 6 | Benchmark: Transverse Field Ising Model | 81 |
| 6.1 | The Model and its Phase Diagram | 82 |
| 6.2 | Ground states on the Infinite Square Lattice | 86 |
| 6.3 | Ground states on Finite Square Lattices | 89 |
| 6.4 | Quench Dynamics | 92 |
| 7 | Conclusion and Outlook | 97 |
| | Acknowledgements | 101 |

Appendices

| | |
|--|------------|
| A Numerics | 103 |
| A.1 Code Source Files and Numerical Data | 103 |
| A.2 Interpolation Subroutines for Line Search | 103 |
| A.3 Extracting the Dynamical Structure Factor | 104 |
| B Complex (Wirtinger) Calculus | 107 |
| B.1 Scalar Functions | 107 |
| B.2 Vector Functions | 109 |
| C Derivations for Automatic Differentiation | 111 |
| C.1 Rigorous Treatment of General Computation Graphs | 111 |
| C.2 Derivations for Complex Functions | 113 |
| C.3 Collection of Automatic Differentiation Formulae | 117 |
| References | 120 |

Chapter 1

Introduction

The study of correlated quantum many-body systems is a highly active research field. It comprises a multitude of exciting phenomena, such as high-temperature superconductivity, the anomalous Hall effect, spin-liquid phases or symmetry-protected topological phases, all of which have seen great interest and progress in recent years. Applications range from materials research and the computer industry to energy management, increasing the efficiency of solar cells [1] or developing infrastructure for energy transport using high-temperature superconductors [2]. In light of climate change, such technologies contribute to sustaining quality of life for the human population.

The qualitative nature, as well as some quantitative properties of many phenomena that arise in quantum many body systems are, in a sense, universal. That is, they do not depend on microscopic details of the physical system in which they arise. The prototypical example of such universality are phase transitions, as described by Landau-Ginzburg theory [3, 4]. In this framework, the description of the system can be reduced to only a few properties such as symmetries and order parameters. Topological phases of matter in non-interacting systems have also been classified [5, 6], and many properties of a topological phase can be understood just by determining into which class it falls.

This universality allows us to study idealized models instead of complicated realistic systems. For quantum magnets, that would involve a full description of all protons, neutrons and electrons in a crystalline solid that could contain impurities, various interactions with a noisy environment, and many more messy details of the real world. Instead, simplified models, such as spin-models on completely regular and symmetric lattices can be studied theoretically. Due to the common pattern of universality, we can hope to understand the mechanisms that govern the behavior of complicated real-world systems by understanding them in idealized settings, that at first glance may look like over-simplified toy models.

A prominent example is the J_1 - J_2 model, which describes $S = 1/2$ spins on a two-dimensional square lattice. It only includes interaction of nearest- and next-nearest-neighbor spins, and yet this simple model exhibits a rich phase diagram and is the prototypical model to study magnetic order in frustrated magnets [7, 8] and spin liquid states [9, 10]. It is “simple” in the sense that it only includes the spin degrees of freedom, only considers two short-range interaction terms and only has two parameters (the energy scales J_1 and J_2 of the interactions). It is, however, surprisingly difficult to study its properties analytically. Even the phase diagram is still subject of ongoing research [11, 12].

This difficulty is a common pattern in theoretical condensed matter physics. Even the simplified models (especially in more than one spatial dimension) can rarely be solved exactly. The quantum transverse field Ising model (TFIM) [13] is arguably the most simple non-trivial model, and yet in two and more spatial dimensions, no exact solutions are known¹. In order to understand their properties, (approximate) numerical methods are essential.

The central challenge in numerical simulation of quantum-many-body systems is the huge dimension of the Hilbert space, which generically scales exponentially with the system size. This means that straight-forward methods like exact diagonalization of the Hamiltonian, even with Krylov-subspace methods, are only viable for comparatively small systems. In some situations, quantum Monte Carlo methods [16, 17] are a powerful tool to study systems directly in the thermodynamic limit. However, they suffer from limitations, in particular in the presence of geometric frustration, as well as the infamous sign problem in fermionic systems [18].

In these cases in particular, tensor network methods offer powerful tools to simulate quantum-many-body systems. They also allow access to dynamical properties via real time evolution. The key to the approximation in terms of tensor network states is the low (quantified in terms of the area law [19–21]) entanglement of ground states and low-energy excited states of local gapped models.

For one-dimensional systems, matrix product states (MPS) [22, 23] are an exceedingly powerful and versatile tool to study ground states, thermal states or perform time evolution. Much progress has been made to improve algorithms for MPS, such as the density matrix renormalization group (DMRG) [24, 25], which allows finding ground states, and methods for time evolution, namely time evolving block decimation (TEBD) [26–30] and the time dependent variational principle (TDVP) [31–33].

The natural generalization of MPS to two dimensions are projected entangled pair states (PEPS) [22, 23, 34]. While they offer similar versatility for the study of 2D systems, algorithms, such as simple update (SU) [35–37] and full update (FU) [38–40] which implement imaginary time evolution to find PEPS ground states, as well as gradient-based methods² [41, 42], and time evolution methods [43, 44] are typically less powerful. In short, this is due to the fact that there is no higher-dimensional strict analogue of the canonical form, which makes MPS calculations so efficient [45]. This contrast goes so far, that it is often viable to study two-dimensional systems by rewriting them as a one-dimensional problem³ and using MPS methods, rather than resorting to PEPS.

In this thesis, we investigate a novel approach to finding good PEPS approximations for the true states in a given problem: direct optimization of the PEPS tensors using gradients from automatic differentiation [47, 48]. We formulate the task as a minimization (or maximization) problem. Finding the ground state, for example, is nothing else than finding the state with the lowest energy expectation value. If we can argue that the ground state can be well approximated by a PEPS⁴, we can find a good approximation of the true ground state, namely the PEPS state with the smallest (minimal) energy. Additionally, we provide

¹The classical model can be solved in one or two dimensions [14, 15], while exact solutions for the quantum model are only known in 1D.

²This refers to analytical expressions of PEPS gradients, not to gradients from automatic differentiation, which we employ in this work.

³A 2D stripe (thin in one direction, and long, possibly infinite, in the other) is treated as a 1D system, by tiling it with a chain in a snake-like geometry. E.g. [46].

⁴This is the case e.g. for ground states of local gapped Hamiltonians [19, 20, 25].

a proof-of-principle, that optimization methods can be used to perform time evolution, by maximizing an appropriate overlap⁵. Such gradient-based optimization methods are not exclusive to PEPS states, and can in principle be applied to any variational ansatz. We will, however, focus exclusively on optimizing PEPS states on two dimensional square lattices.

In order to perform the optimization, we need three major ingredients:

1. The variational ansatz, which we understand as a parametrization of states in the *variational manifold* (a subset of the full Hilbert space) using tensors. We need to argue that the state of interest (e.g. the ground state or the time evolved states) can be well-approximated within the variational manifold. The task of approximating the state of interest is then formulated as the minimization problem of an appropriately chosen (scalar-valued) *cost function* of the variational parameters (e.g. the energy, for finding ground states). We need a numerical implementation to evaluate the cost function for any given parametrization. We have developed `python` code for these tasks.
2. Computing gradients of the cost function. Automatic differentiation is perfectly suited for this task, since it can compute gradients of any (differentiable) function, if a suitable implementation of the function itself is available. In the numerics, we let the `python` package `jax` [49] take care of this entirely, but we explain in detail what happens “under the hood”. Many automatic differentiation tools do not (or not fully) support complex functions. This is, however, a requirement for our scheme, since the PEPS states are parametrized with complex-valued tensors⁶.
3. An optimization algorithm. The gradients (and function values) of the cost function are used to find the state that minimizes the cost function in the variational manifold. The simplest algorithm would be gradient descent. However, the gradient computations are expensive and a more sophisticated optimization algorithm, that needs fewer gradient evaluations to reach a given precision, is desirable. The optimization algorithm also needs to allow for a cost function of complex arguments (the complex PEPS tensors). We implement a generalized (to complex variables) L-BFGS algorithm with `jax` as a numerical backend.

Structure

This thesis is organized as follows: We first discuss the three tools above separately and independently.

In chapter 2, we discuss the optimization algorithm. In order to introduce the relevant concepts, we touch on gradient descent and Newton’s method and then introduce the L-BFGS algorithm, which we use for the numerics, as well as its generalization to cost functions in complex variables.

We introduce automatic differentiation (autodiff) in detail in chapter 3. We aim to provide an instructive explanation that balances intuitive accessibility and mathematical rigor. This includes a review of the core principles, strategies for implementation, an overview of the interface of `jax`, as well as a short review of available autodiff tools for various languages. A few autodiff formulae of particular interest to our application (complex

⁵If $|\langle \phi | \exp(-iH\delta t) | \psi(t) \rangle|$ reaches its maximal value of 1, then $|\phi\rangle$ is (up to a phase) $|\psi(t + \delta t)\rangle$.

⁶In some cases (e.g. finding ground states of real Hamiltonians), a real parametrization can be chosen and complex autodiff is not necessary.

SVD, fixed points of iterative maps and contraction of tensor networks) are highlighted in the appendix C.3.

In chapter 4, we introduce the tensor network methods relevant to this work. This includes an introduction to projected entangled pair states (PEPS). We discuss methods to numerically evaluate expectation values and matrix elements of relevant operators in finite and infinite PEPS states. As a key ingredient, we develop a scheme for writing the Hamiltonian⁷ and the time evolution operator as PEPOs (PEPS operators). This reduces the number of required PEPS contractions to $\mathcal{O}(1)$.

In chapter 5, we put the three tools for gradient-based optimization together and detail how they can be used to find ground states, both for infinite and finite systems, as well as perform time evolution after a local quench in finite systems.

In chapter 6, we benchmark the methods, considering the transverse field Ising model (TFIM) on a 2D square lattice. We optimize infinite PEPS for the ground state of the TFIM on the infinite square lattice and analyze the phase diagram. We compare the variational energy of PEPS ground states on finite lattices to other PEPS methods and finally perform time evolution (on the finite lattice) after a local quench. We compare the dynamics of local observables, spin-spin correlation functions and the dynamical (spin) structure factor to exact diagonalization data. While we find that more development will be necessary to make gradient-based optimization competitive with other tensor network methods, we understand this benchmark as a proof-of-principle that demonstrates the viability of this tool and motivates further development.

We summarize our findings and discuss strategies to further improve the method and make larger system-sizes and bond dimensions accessible in chapter 7.

All code that was developed during the course of this project is available on GitHub [50].

⁷The Hamiltonian is written as the sum of two PEPOs.

Chapter 2

Gradient-Based Optimization of Real-Valued Functions in Many Real or Complex Variables

Contents

| | | |
|------------|--|-----------|
| 2.1 | Line Search Methods | 7 |
| 2.1.1 | Line Search Criteria | 8 |
| 2.1.2 | Line Search Algorithms | 9 |
| 2.2 | Gradient Descent and Newton's Method | 13 |
| 2.2.1 | The Gradient Descent Method | 13 |
| 2.2.2 | Newton's Method | 14 |
| 2.3 | Quasi-Newton Methods: BFGS and L-BFGS | 14 |
| 2.3.1 | The BFGS Method | 16 |
| 2.3.2 | The L-BFGS Method | 18 |
| 2.4 | Functions in Complex Variables | 20 |
| 2.4.1 | Line Search | 22 |
| 2.4.2 | The Generalized Gradient Descent Method | 22 |
| 2.4.3 | The Generalized L-BFGS Method | 23 |

In a general setting, the goal of numerical optimization is to find the inputs to a scalar-valued function, that make its value as small as possible. That is, we want to find the input $\mathbf{x}_{\min} \in \mathbb{R}^n$ for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ that minimizes it:

$$\mathbf{x}_{\min} := \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} f(\mathbf{x}). \quad (2.1)$$

Numerical optimization (of a function of continuous variables) finds applications in countless fields, ranging from engineering to finance. To mention but a few, numerical optimization can be used to find optimal trajectories for robot arms in assembly lines, compile investment portfolios with optimal risk/reward ratios or determine the shape of an aircraft component that minimizes air resistance.

In this thesis, we use numerical optimization to find optimal tensor network states. We dedicate chapter 5 to discuss how optimization can be used in the context of tensor networks in condensed matter physics and focus on the optimization of an abstract function in this chapter.

We are ultimately interested in optimizing functions in complex variables, since many-body wave functions are naturally complex. We first discuss optimization of functions in real variables, then show how optimization algorithms can be adapted to cover functions in complex variables in section 2.4. The algorithms for complex functions can be implemented in such a way, that they include the optimization of real functions as a special case.

Throughout this chapter, we will focus on minimization problems, i.e. finding the inputs of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ (or $f : \mathbb{C}^n \rightarrow \mathbb{R}$ for the complex case) that yield the *smallest* possible function value. Maximization problems can be reduced to a minimization problem. A function is maximized when its negative is minimized.

We assume that we have the following tools available:

- An initial guess $\mathbf{x}_0 \in \mathbb{R}^n$. Usually initial guesses are obtained from an intuitive understanding of the problem. If that is not viable, \mathbf{x}_0 can be chosen randomly.
- Numerically evaluating function values $f(\mathbf{x})$ for any $\mathbf{x} \in \mathbb{R}^n$.
- Numerically evaluating gradients $\nabla f(\mathbf{x})$ for any $\mathbf{x} \in \mathbb{R}^n$. We intend to use automatic differentiation (see chapter 3) to obtain gradients, but any method is acceptable, including exact formulae or numerical approximation of an analytic expression.
- For Newton's method discussed in section 2.2.2, it is necessary to evaluate products of the Hessian matrix of f with any vector.

Our goal is to obtain an algorithm that computes the global minimizer $\mathbf{x}_{\min} \in \mathbb{R}^n$, at least approximately. The algorithms will be iterative procedures, i.e. they will produce a sequence of $\mathbf{x}_k \in \mathbb{R}^n$ that converges to \mathbf{x}_{\min} .

For the numerics of this work the L-BFGS algorithm was used (see section 2.3), in particular algorithms 5 and 6, as well as the generalization to complex variables, algorithm 7. We have developed an implementation with `jax` for `python` as a numerical backend. It is available on GitHub [50].

In section 2.1, we discuss a general framework for a broad class of optimization algorithms: line search algorithms. We introduce the gradient descent and Newtons method in section 2.2 and focus on the quasi-Newton methods BFGS and L-BFGS in section 2.3. All of the above will deal with functions in real variables and we extend the algorithms to functions of complex variables in section 2.4.

2.1 Line Search Methods

We present a general framework for optimization algorithms: line search based methods. Consider a continuously differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. The goal is to find $\mathbf{x}_{\min} \in \mathbb{R}^n$ (at least approximately), such that $f(\mathbf{x}_{\min}) \leq f(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{R}^n$.

The requirements on f are usually not too strict, and mostly comprise being bounded from below (such that a global minimum is well defined) and being differentiable to a low finite order (so that we have access to gradients and Hessians, either for the actual algorithms or in order to prove the convergence of the algorithm) [51, chpt. 3].

The optimization starts with an initial guess \mathbf{x}_0 that is usually obtained from an intuitive knowledge of the function f . For example, when minimizing the energy of quantum state, an appropriate product state could be the initial guess¹. When sweeping through parameters of the Hamiltonian to characterize a phase transition, the ground state of the previous parameter can be a good initial guess. In the worst case just choosing \mathbf{x}_0 randomly, or $\mathbf{x}_0 = 0$ can be tried. Choosing a starting point close to a critical point of f , that is a point where the gradient $\nabla f(\mathbf{x}_0) = 0$ vanishes, should be avoided. Most optimization algorithms will terminate if the gradient vanishes, since that is a criterion for having found the minimum.

Starting from the initial guess \mathbf{x}_0 , an iterative procedure $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ is devised. The k^{th} step \mathbf{s}_k is comprised of a *search direction* $\mathbf{p}_k \in \mathbb{R}^n$ and a *step length* $\alpha_k > 0$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k. \quad (2.2)$$

To determine the next iterate \mathbf{x}_{k+1} , the objective function f is approximated around \mathbf{x}_k by a model:

$$m_k(\mathbf{s}) \approx f(\mathbf{x}_k + \mathbf{s}). \quad (2.3)$$

This model is often a truncated Taylor expansion of f around \mathbf{x}_k . Generically, the model is more accurate, the smaller \mathbf{s} is in magnitude.

Two classes of optimization strategies arise from how the information of the model is treated.

In a trust-region method [51, chpt. 4], the model is only trusted to be a good approximation of f in a finite region around \mathbf{x}_k . The next iterate \mathbf{x}_{k+1} is then chosen as the minimizer of the model within that trust-region. Note that this might be a step with both different length and different direction then towards the global minimizer of the model. Trust-region methods then differ by the choice of model and by how the size (or even shape) of the trust-region is determined. We will not discuss these methods in further detail.

In a line search method [51, chpt. 3], on the other hand, the global minimizer of the model

$$\mathbf{p}_k := \operatorname{argmin}_{\mathbf{s} \in \mathbb{R}^n} m_k(\mathbf{s}) \quad (2.4)$$

¹Product states as initial guesses can cause problems, however. They can be critical points of the cost function (i.e. have a vanishing gradient). In order to remedy such problems, we recommend adding a small random perturbation to the product state.

is used as a search direction. Then, the step length α_k is determined by analyzing f on the ray $\{\mathbf{x}_k + \alpha \mathbf{p}_k \mid \alpha > 0\}$. The result is a step in the direction of the model's global minimizer, but with a potentially quite different length.

We will present different algorithms for finding search directions \mathbf{p}_k in sections 2.2 (gradient-descent and Newton's method) and 2.3 (quasi-Newton methods). In the following we will focus on the line search, i.e. finding good step lengths α_k , once the direction \mathbf{p}_k has been determined.

2.1.1 Line Search Criteria

We assume that we have a search direction $\mathbf{p}_k \in \mathbb{R}^n$ and we want to find a suitable step length $\alpha_k > 0$. We introduce the univariate function $\phi(\alpha) := f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ which is just f evaluated on the ray given by the direction \mathbf{p}_k starting from \mathbf{x}_k . Its derivative is

$$\phi'(\alpha) = \nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k) \cdot \mathbf{p}_k. \quad (2.5)$$

We require that \mathbf{p}_k is a *descent direction*:

$$\mathbf{g}_k \cdot \mathbf{p}_k = \phi'(0) \stackrel{!}{<} 0, \quad (2.6)$$

where we have introduced the shorthand

$$\mathbf{g}_k := \nabla f(\mathbf{x}_k) \quad (2.7)$$

for the gradient at the current iterate.

Equation (2.6) guarantees that f can be decreased by taking steps in the search direction. In particular, it guarantees that f will decrease if the step length is short enough.

Let us now discuss the requirements that a good step length α should fulfill. In principle, the global minimizer of ϕ over all positive α would be a good choice but it is expensive to compute it in a stable manner. It is also not necessary, since the minimizer of f that we are aiming for might not be on the search ray or even anywhere close to it. It is important to remember that the line search is just a subproblem. What we actually want to accomplish is to guarantee that the iterative procedure (2.2) converges to the minimizer of f in a stable and efficient manner.

How long should our step be then?

A common requirement [51–53] is that the step should provide a significant decrease of the objective function f . This is commonly quantified in terms of the *sufficient decrease condition* (also known as the Armijo condition):

$$\phi(\alpha_k) \leq \phi(0) + c_1 \alpha_k \phi'(0), \quad (2.8)$$

with a constant $c_1 \in (0, 1)$. We can intuitively understand this condition as demanding that the step reduces the objective function f at least as much as a decreasing line with slope $c_1 \phi'(0)$.

Note that short enough steps will always fulfill the Armijo condition, since \mathbf{p}_k is a descent direction and $c_1 < 1$. Ideally, we want to find the largest α that fulfills the Armijo condition. This, again, would be quite expensive to determine and is not necessary. We only want a criterion to rule out steps that are much too short.

This is typically [51–53] done by imposing the *curvature condition*:

$$\phi'(\alpha_k) \geq c_2 \phi'(0), \quad (2.9)$$

with a constant $c_2 \in (c_1, 1)$. Note that the right hand side (RHS) is always negative and the left hand side (LHS) is often negative.

Intuitively, if the curvature condition is not fulfilled, the slope of ϕ at a proposed step length is more negative than $c_2 \phi'(0)$. This means we can make better progress by stepping further in the same direction and therefore reject the proposed α_k .

The sufficient decrease and curvature conditions are commonly referred to as the *Wolfe conditions*. Let us summarize and rewrite them in terms of f :

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \mathbf{g}_k \cdot \mathbf{p}_k; \quad (2.10a)$$

$$\nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \cdot \mathbf{p}_k \geq c_2 \mathbf{g}_k \cdot \mathbf{p}_k, \quad (2.10b)$$

with parameters $0 < c_1 < c_2 < 1$. Typically c_1 is chosen to be quite small, for example $c_1 = 10^{-4}$. For quasi-Newton algorithms, $c_2 = 0.9$ is recommended [51, p.62].

We can also rule out large positive slopes after the step with minimal additional effort, by taking the absolute in the curvature condition. With this modification, the two conditions are known as the *strong Wolfe conditions*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \mathbf{g}_k \cdot \mathbf{p}_k; \quad (2.11a)$$

$$|\nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \cdot \mathbf{p}_k| \leq -c_2 \mathbf{g}_k \cdot \mathbf{p}_k. \quad (2.11b)$$

It can be shown that a step length that fulfills the strong Wolfe conditions (and therefore also the Wolfe conditions) always exists, if f is continuously differentiable and bounded from below on the search ray [51, p.35].

An example illustration for acceptable step length for a generic function is given in figure 2.1.

2.1.2 Line Search Algorithms

We have discussed various criteria for viable step lengths. Now we need algorithms to find them in practice.

There is a plethora of line search algorithms available in the public domain [54–58]. We will present in detail two algorithms: The first is guaranteed to find a step length that fulfills the Armijo condition (2.8) and is, in a lax sense, not too short. It does, however, not guarantee any curvature conditions. We refer to it as *Armijo backtracking*.

The second algorithm, which we refer to as *strong Wolfe line search* is guaranteed to find a step length that fulfills the strong Wolfe conditions (2.11). In contrast to Armijo backtracking, the strong Wolfe line search algorithm requires computing the derivative of ϕ (which involves the gradient of f) at trial step lengths. It is therefore (in general) more expensive.

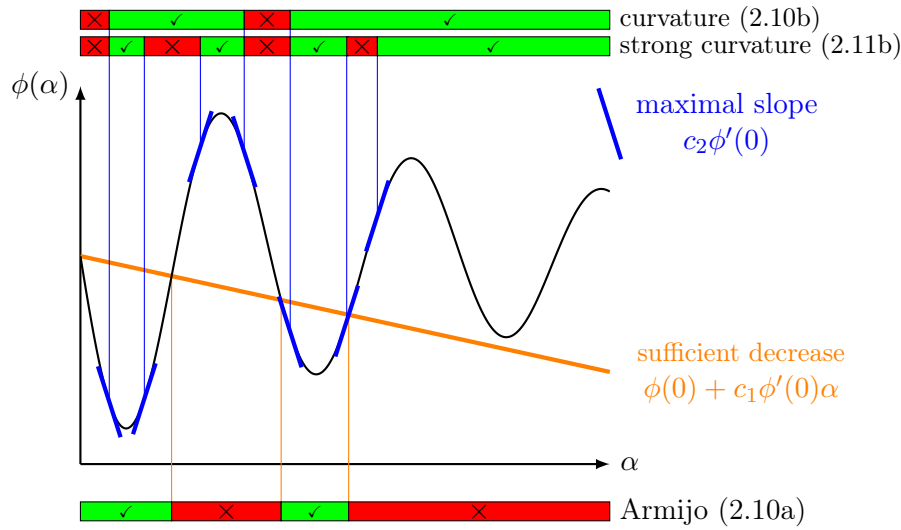


Figure 2.1: Acceptable step lengths under different line search conditions for a generic example function $\phi(\alpha)$ (black). The Armijo condition (2.10a) is fulfilled, if $\phi(\alpha)$ lies below the line with slope $-c_1\phi'(0)$ (orange). The curvature condition (2.10b) is fulfilled, if the slope $\phi'(\alpha)$ is greater then $c_2\phi'(0)$ (blue), which is negative. For the strong Wolfe conditions, the “strong curvature condition” (2.11b), the absolute of the slope needs to be smaller then $-c_2\phi'(0)$.

Armijo backtracking

For Armijo backtracking line search, we assume that we have an initial guess $\alpha_k^{(0)}$ for the step length. What choices are sensible depends on how the search-direction is scaled. If the search-directions are well-scaled, like they are in quasi-Newton methods, $\alpha_k^{(0)} = 1$ should be used as the initial guess [51, p.58].

We want to find a step length $\alpha_k \in (0, \alpha_k^{(0)}]$ that fulfills the Armijo condition (2.8), while not being too small. We propose smaller and smaller trial steps $\alpha_k^{(m)} < \alpha_k^{(m-1)}$ until one fulfills the Armijo condition. This will be the result α_k . Here, $\alpha_k^{(m)}$ is the m^{th} trial value (m^{th} iteration of the line search algorithm) for the step length α_k at the k^{th} iteration of the over-all optimization algorithm.

We obtain a new trial value $\alpha_k^{(m)}$ by interpolating ϕ using the information we already have available. From determining the search direction, we already know $\phi(0) = f(\mathbf{x}_k)$ and $\phi'(0) = \mathbf{g}_k \cdot \mathbf{p}_k$. Since we have checked the Armijo condition for the previously proposed steps, we have already evaluated $\phi(\alpha_k^{(m-1)})$ and $\phi(\alpha_k^{(m-2)})$. Thus, we can interpolate ϕ between 0 and $\alpha_k^{(m-2)}$ using a cubic polynomial that matches these values and choose $\alpha_k^{(m)}$ as its minimizer. Note that at $m = 1$ we only have $\phi(\alpha_k^{(m-1)})$ and thus interpolate with a quadratic polynomial.

The subroutines `quadratic_interpolate` and `cubic_interpolate` are discussed in appendix A.2. They do what their name suggests, and return the minimizer of the quadratic (cubic) polynomial that matches ϕ at the given function and derivative values.

Algorithm 1: Armijo Backtracking with gradient-free Interpolation

Input: $\phi(0)$, $\phi'(0)$, $\alpha_0 > 0$, $c_1 \in (0, 1)$ **Output:** Step Length $\alpha \in (0, \alpha_0]$ that fulfills the Armijo condition (2.8)

```

1 Evaluate  $\phi(\alpha_0)$ 
2 if  $\phi(\alpha_0) \leq \phi(0) + c_1\alpha_0\phi'(0)$  then
3   return  $\alpha_0$ 
4  $\alpha \leftarrow \text{quadratic\_interpolate}(0, \phi(0), \phi'(0), \alpha_0, \phi(\alpha_0))$ 
5 Evaluate  $\phi(\alpha)$ 
6 if  $\phi(\alpha) \leq \phi(0) + c_1\alpha\phi'(0)$  then
7   return  $\alpha$ 
8 while  $\phi(\alpha) > \phi(0) + c_1\alpha\phi'(0)$  do
9    $\alpha_{\text{old}} \leftarrow \alpha$ 
10   $\alpha \leftarrow \text{cubic\_interpolate}(0, \phi(0), \phi'(0), \alpha, \phi(\alpha), \alpha_{\text{old}}, \phi(\alpha_{\text{old}}))$ 
11  Evaluate  $\phi(\alpha)$ 
12 return  $\alpha$ 

```

Strong Wolfe Line-Search

For the strong Wolfe line search, the goal is to find a step length α_k that fulfills the strong Wolfe conditions (2.11).

We require an initial guess $\alpha_k^{(1)}$. It is the step length that will be chosen “by default”, anytime it is acceptable. For quasi-Newton methods, $\alpha_k^{(1)} = 1$ should be chosen, to take full advantage of their convergence properties [51, p.58]. In order to avoid an infinite loop, we introduce an upper bound α_{max} . This upper bound should be chosen quite large, and we suggest to leave it out at first, and gain an understanding, which step lengths are typically obtained for the given problem.

In terms of efficiency, the goal is to evaluate ϕ' , which requires ∇f , as few times as possible.

We adapt the algorithm by R. Fletcher [57] to our notation and explain it in detail. See algorithm 2.

The idea of the algorithm is to first find an interval of step lengths that contains an acceptable α , i.e. a step length that fulfills the strong Wolfe conditions (2.11). The second part of the algorithm, **zoom**, then narrows this window down, until a suitable step length is found. It is detailed in algorithm 3. Let us first discuss, how the main part of the line search (algorithm 2) works.

The algorithm creates a sequence of trial values $\alpha_k^{(m)}$, which is increasing monotonously.

If the **if** statement in line 5 triggers, we know that $\alpha_k^{(m)}$ is the *first* trial step length to violate the Armijo condition. This means that ϕ must have curved upward between $\alpha_k^{(m-1)}$ and $\alpha_k^{(m)}$. This interval therefore contains a step length that fulfills the strong Wolfe conditions. See figure 2.1 for an intuitive picture.

If the **if** statement in line 8 triggers, we know that $\alpha_k^{(m)}$ fulfills the curvature condition. And since it did not trigger the **if** in line 5, it also fulfills the Armijo condition. Therefore, $\alpha_k^{(m)}$ is acceptable, i.e. fulfills both strong Wolfe conditions.

Algorithm 2: Strong Wolfe Line search

Input: $\alpha_{\max} > 0$, $\alpha_k^{(1)} \in (0, \alpha_{\max})$, $c_1 \in (0, 1)$, $c_2 \in (c_1, 1)$ **Output:** Step Length $\alpha_k \in (0, \alpha_{\max}]$ that fulfills the Strong Wolfe conditions (2.11)

```

1  $\alpha_k^{(0)} \leftarrow 0$ 
2  $m \leftarrow 1$ 
3 while  $\alpha_k^{(m)} < \alpha_{\max}$  do
4   Evaluate  $\phi(\alpha_k^{(m)})$ 
5   if  $\phi(\alpha_k^{(m)}) > \phi(0) + c_1 \alpha_k^{(m)} \phi'(0)$  then
6     return  $\text{zoom}(\alpha_k^{(m-1)}, \alpha_k^{(m)})$ 
7   Evaluate  $\phi'(\alpha_k^{(m)})$ 
8   if  $|\phi'(\alpha_k^{(m)})| \leq -c_2 \phi'(0)$  then
9     return  $\alpha_k^{(m)}$ 
10  if  $\phi'(\alpha_k^{(m)}) \geq 0$  then
11    return  $\text{zoom}(\alpha_k^{(m)}, \alpha_k^{(m-1)})$  // Note the order of arguments!
12  Choose  $\alpha_k^{(m+1)} \in (\alpha_k^{(m)}, \alpha_{\max})$ 
13   $m \leftarrow m + 1$ 

```

If the **if** statement in line 10 triggers, we know that both $\alpha_k^{(m)}$ and $\alpha_k^{(m-1)}$ fulfill the Armijo condition (since neither one triggered line 5) and that $\alpha_k^{(m)}$ violates the (strong) curvature condition, because the slope is too positive, not too negative. Thus, the step $\alpha_k^{(m)}$ is already too far and the interval $(\alpha_k^{(m-1)}, \alpha_k^{(m)})$ contains acceptable step lengths.

If we reach line 12, we know that $\alpha_k^{(m)}$ is too short to fulfill the strong Wolfe conditions. So we choose a larger next trial value. We recommend to choose $\alpha_k^{(m+1)} = \min(2\alpha_k^{(m)}, \alpha_{\max})$ and to allow one more loop iteration if $\alpha_k^{(m+1)} = \alpha_{\max}$. Alternatively, it can be promising to extrapolate ϕ from the given information, similar to the interpolation in algorithm 1.

Let us now discuss the zoom phase of the strong Wolfe line search (algorithm 3). It is called with two arguments, $\alpha_{\text{lo}}, \alpha_{\text{hi}} > 0$ that are *not necessarily ordered by magnitude*. They fulfill the following properties:

1. The interval bounded by $\alpha_{\text{lo}}, \alpha_{\text{hi}}$ contains an α that fulfills the strong Wolfe conditions;
2. α_{lo} has the smallest function value among all previously generated step lengths that fulfill the Armijo condition;
3. $\phi'(\alpha_{\text{lo}})(\alpha_{\text{hi}} - \alpha_{\text{lo}}) < 0$.

The **zoom** algorithm finds an α_j in the interval bounded by $\alpha_{\text{lo}}, \alpha_{\text{hi}}$. If α_j fulfills the strong Wolfe conditions, it is the desired output. If not, it can replace either α_{hi} or α_{lo} , such that the above conditions still hold. For more details and references see [51, p. 62f].

Algorithm 3: zoom**Input:** $\alpha_{lo}, \alpha_{hi} > 0$ that fulfill the requirements discussed in the main text $c_1 \in (0, 1), c_2 \in (c_1, 1)$ **Output:** Step Length α_k in the interval bounded by α_{lo} and α_{hi} that fulfills the strong Wolfe conditions

```

1 while Resources not exceeded do
2   Determine  $\alpha$  between  $\alpha_{lo}$  and  $\alpha_{hi}$  by interpolation
3   Evaluate  $\phi(\alpha)$ 
4   if  $\phi(\alpha) > \phi(0) + c_1\alpha\phi'(0)$  or  $\phi(\alpha) \geq \phi(\alpha_{lo})$  then
5      $\alpha_{hi} \leftarrow \alpha$ 
6   else
7     Evaluate  $\phi'(\alpha)$ 
8     if  $|\phi'(\alpha)| \leq -c_2\phi'(0)$  then
9       return  $\alpha$ 
10    if  $\phi'(\alpha)(\alpha_{hi} - \alpha_{lo}) \geq 0$  then
11       $\alpha_{hi} \leftarrow \alpha$ 
12     $\alpha_{lo} \leftarrow \alpha$ 

```

2.2 Gradient Descent and Newton's Method

In this section, we briefly introduce the gradient-descent method and Newton's method. Both of them will not be applied in this thesis, but serve demonstrative purposes when we introduce the methods that are used.

2.2.1 The Gradient Descent Method

The gradient descent algorithm fits in the framework of line search methods (section 2.1), if the model (equation (2.3)) is chosen to be the first-order Taylor-approximation. For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ and current iterate $\mathbf{x}_k \in \mathbb{R}^n$, that is the model

$$m_k(\mathbf{s}) := f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k) \cdot \mathbf{s}, \quad (2.12)$$

where $\mathbf{s} \in \mathbb{R}^n$ is the proposed step to the next iterate. This model agrees with $f(\mathbf{x}_k + \mathbf{s})$ up to corrections of order $\mathcal{O}(|\mathbf{s}|^2)$.

The linear model m_k is not bounded from below and therefore has no minimizer. Equation (2.4) can not be taken literally here. However, the model does determine a direction, namely the direction in which it can be decreased the fastest:

$$\mathbf{p}_k = -\nabla f(\mathbf{x}_k), \quad (2.13)$$

which is also the direction of steepest descent of f at \mathbf{x}_k . For this reason, the gradient-descent method is also known as the method of steepest descent.

This choice of search direction can be paired with any line search algorithm, including not performing line search at all and just taking steps of pre-determined lengths.

2.2.2 Newton's Method

We get Newton's method by taking the second-order Taylor approximation of f around the current iterate \mathbf{x}_k as the model

$$m_k(\mathbf{s}) := f(\mathbf{x}_k) + \mathbf{g}_k \cdot \mathbf{s} + \frac{1}{2} \mathbf{s}^T \cdot H_k \cdot \mathbf{s}, \quad (2.14)$$

where we have abbreviated the gradient $\mathbf{g}_k := \nabla f(\mathbf{x}_k)$ at the current iterate and the Hessian matrix

$$H_k := \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}_k) \right)_{ij} \in \mathbb{R}^{n \times n}, \quad (2.15)$$

which is a symmetric matrix by Schwarz's theorem.

The model (2.14) agrees with $f(\mathbf{x}_k + \mathbf{s})$ up to corrections of order $\mathcal{O}(|\mathbf{s}|^3)$. Its minimizer (if it exists), and therefore the search direction is

$$\mathbf{p}_k = -H_k^{-1} \cdot \mathbf{g}_k. \quad (2.16)$$

In general (when f is not convex) the Hessian H_k is not positive-definite. This means that the search-direction \mathbf{p}_k is not guaranteed to be a descent direction. Intuitively, the minimizer could lie on the opposing side of a saddle point. This makes much of our discussion in section 2.1 obsolete.

In practice, this is remedied by modifying the Hessian matrix to become positive definite [51, chpt. 3.4]. We will not discuss these modified Newton methods in further detail and focus instead on quasi-Newton methods. They do not use the exact Hessian, but only an approximation. This makes it possible to simply demand that the Hessian approximation be positive definite.

The Newton method, even after the problem of positive definiteness is remedied, still requires to compute the product of the Hessian (or a modification thereof) with a vector. Quasi-Newton methods only require gradients. While it is possible to compute such Hessian-vector products with automatic differentiation [59], they can be quite expensive. The limiting factor for optimizations in this thesis is the cost of computing derivatives, therefore the Hessian-vector products are not a viable option.

2.3 Quasi-Newton Methods: BFGS and L-BFGS

In this section, we will present quasi-Newton methods as approximations to Newton's method. We will discuss some general features of quasi-Newton methods and then focus on the BFGS (Broyden, Fletcher, Goldfarb, Shanno) method, as well as its limited-memory version, the L-BFGS (limited memory BFGS) method.

The idea behind quasi-Newton methods is to still employ a quadratic model, like (2.14), without having to compute second derivatives of the objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

We use models of the form

$$m_k(\mathbf{s}) = f(\mathbf{x}_k) + \mathbf{g}_k \cdot \mathbf{s} + \frac{1}{2} \mathbf{s}^T \cdot B_k \cdot \mathbf{s}, \quad (2.17)$$

where B_k is an approximation of the exact Hessian H_k and $\mathbf{g}_k := \nabla f(\mathbf{x}_k)$ is the gradient at the current iterate. The Hessian-approximation B_k is updated at each optimization

step. The different quasi-Newton methods differ by how the update formula for B_k is obtained.

The minimizer of the model (2.17) is

$$\mathbf{p}_k = -B_k^{-1} \cdot \mathbf{g}_k. \quad (2.18)$$

We demand that B_k (and therefore also its inverse) be positive definite. This guarantees that \mathbf{p}_k is a descent direction and, as discussed in section 2.1, we can find a step length α_k that fulfills the strong Wolfe conditions (2.11). We then obtain the next iterate

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (2.19)$$

and the new model at \mathbf{x}_{k+1} is

$$m_{k+1}(\mathbf{s}) = f(\mathbf{x}_{k+1}) + \mathbf{g}_{k+1} \cdot \mathbf{s} + \frac{1}{2} \mathbf{s}^T \cdot B_{k+1} \cdot \mathbf{s}, \quad (2.20)$$

for which we need to determine B_{k+1} . In order to do that, we impose more and more conditions on it, until it is determined uniquely. These conditions should not involve information that is expensive to compute and ideally only require the iterates \mathbf{x}_k , gradients \mathbf{g}_k and/or function values, since these are likely already available.

A reasonable demand, common among quasi-Newton methods [51, p.137], is that the gradient of the new model m_{k+1} should match the gradient of f at the two points, \mathbf{x}_{k+1} and \mathbf{x}_k . These points are chosen, because the respective gradients \mathbf{g}_k and \mathbf{g}_{k+1} will be computed by the algorithm anyway. The gradient of the new model is

$$\nabla m_{k+1}(\mathbf{s}) = \mathbf{g}_{k+1} + B_{k+1} \cdot \mathbf{s}. \quad (2.21)$$

The gradients of m_k and m_{k+1} match at \mathbf{x}_{k+1} (corresponding to $\mathbf{s} = 0$) by design of the model. We also demand that the gradients match at \mathbf{x}_k . For the model m_{k+1} this point corresponds to the step $-\mathbf{s}_k = -\alpha_k \mathbf{p}_k$, “back to \mathbf{x}_k ”:

$$\mathbf{g}_k = \nabla f(\mathbf{x}_k) = \nabla m_k(0) \stackrel{!}{=} \nabla m_{k+1}(-\mathbf{s}_k) = \mathbf{g}_{k+1} - B_{k+1} \cdot \mathbf{s}_k. \quad (2.22)$$

If we define the change in gradient $\mathbf{y}_k := \mathbf{g}_{k+1} - \mathbf{g}_k$, we can rewrite equation (2.22) to obtain the *secant equation*

$$B_{k+1} \mathbf{s}_k \stackrel{!}{=} \mathbf{y}_k, \quad (2.23)$$

as it is commonly found in literature.

Since we demand B_{k+1} to be positive definite, the secant equation can only be fulfilled if $\mathbf{s}_k^T \cdot \mathbf{y}_k > 0$. This can be seen by multiplying equation (2.23) with \mathbf{s}_k^T from the left.

It is therefore crucial to guarantee that $\mathbf{s}_k^T \cdot \mathbf{y}_k > 0$ at all steps. Fortunately, the second strong Wolfe condition (2.11b) guarantees just that:

$$\mathbf{s}_k^T \cdot \mathbf{y}_k = (\nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)) \cdot \alpha_k \mathbf{p}_k \stackrel{(2.11b)}{\geq} (c_2 - 1) \alpha_k \nabla f(\mathbf{x}_k) \cdot \mathbf{p}_k > 0, \quad (2.24)$$

since $c_2 < 1$, $\alpha_k > 0$ and \mathbf{p}_k is a descent direction. As we can see, a line search method that guarantees the strong Wolfe conditions is crucial for quasi-Newton methods.

In order to determine B_{k+1} uniquely, we need to impose more restrictive conditions than just the secant equation. Quasi-Newton methods differ by the choice of further conditions. Many such methods are explored in [51, chpt.6]. We will only further discuss the conditions that lead to the BFGS method, which is widely accepted [51, p.141] as the most efficient general-purpose quasi-Newton method.

2.3.1 The BFGS Method

The BFGS method was developed by and is named after **B**royden [60], **F**letcher [61], **G**oldfarb [62] and **S**hanno [63], and directly operates on the *inverse* of the Hessian-approximation

$$C_k := B_k^{-1}. \quad (2.25)$$

This is convenient, since C_k is the matrix that is actually required for the search direction in equation (2.18).

The BFGS updating formula for C_{k+1} is obtained by imposing, in addition to the secant condition (2.23), that C_{k+1} be symmetric (as is the inverse of any exact Hessian) and close to the last C_k w.r.t the weighted Frobenius norm. This choice of norm is not obvious, and a different choice would lead to a different algorithm.

$$C_{k+1} := \underset{C}{\operatorname{argmin}} \quad \|C - C_k\|_W \quad (2.26)$$

$$\text{subject to} \quad C^T = C \quad \text{and} \quad C \cdot \mathbf{y}_k = \mathbf{s}_k, \quad (2.26a)$$

where $\|A\|_W := \|W^{1/2} \cdot A \cdot W^{1/2}\|_F$ is the weighted Frobenius norm with weight matrix W , which can be chosen as *any* matrix that satisfies $W \cdot \mathbf{s}_k = \mathbf{y}_k$. The result C_{k+1} does not depend on the choice of W [51, p.138f].

Note that positive definiteness is not among the conditions (2.26a). It is straight-forward to check that the solution (2.27) is positive definite if and only if C_k is. Therefore, to ensure positive-definiteness for all k , we only need to start with a positive definite C_0 .

The unique solution of (2.26) is given by [51, p.140]:

$$C_{k+1} = (\mathbb{1} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) C_k (\mathbb{1} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T, \quad (2.27)$$

where

$$\rho_k := (\mathbf{y}_k^T \cdot \mathbf{s}_k)^{-1}. \quad (2.28)$$

This gives us the BFGS algorithm, which we summarize in algorithm 4.

Algorithm 4: BFGS

Input: Initial guess \mathbf{x}_0 , Threshold $\epsilon > 0$, Positive-definite initial inverse-Hessian approximation C_0

Output: Minimizer \mathbf{x}_{\min} of $f : \mathbb{R}^n \rightarrow \mathbb{R}$

```

1 Evaluate  $\nabla f(\mathbf{x}_0)$ 
2 if  $\|\nabla f(\mathbf{x}_0)\| < \epsilon$  then
3   return  $\mathbf{x}_0$ 
4  $k \leftarrow 0$ 
5 while Resources not exceeded do
6   Compute search direction  $\mathbf{p}_k \leftarrow -C_k \cdot \nabla f(\mathbf{x}_k)$ 
7   Find step length  $\alpha_k$  that fulfills strong Wolfe conditions (2.11), e.g. algorithm 2
8    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$ 
9   Evaluate  $\nabla f(\mathbf{x}_{k+1})$ 
10  if  $\|\nabla f(\mathbf{x}_{k+1})\| < \epsilon$  then
11    return  $\mathbf{x}_{k+1}$ 
12   $\mathbf{s}_k \leftarrow \mathbf{x}_{k+1} - \mathbf{x}_k$ 
13   $\mathbf{y}_k \leftarrow \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$ 
14  Compute  $C_{k+1}$  using equation (2.27)
15   $k \leftarrow k + 1$ 

```

Under reasonable assumptions about the cost function, the BFGS algorithm has a super-linear convergence rate, if the step length $\alpha_k = 1$ is chosen whenever it fulfills the strong Wolfe conditions [51, chpt. 6.4]. The line search parameters in the strong Wolfe conditions are typically [51, p.62] $c_1 = 10^{-4}$ and $c_2 = 0.9$.

Good choices for C_0 are problem-specific and not very obvious. For more details and comments on the implementation of the BFGS algorithm, as well as a detailed convergence analysis we refer to Wright & Nocedal [51, p.136ff], as well as [64, 65]. As a one-size-fits-all solution, we propose the following strategy: Use $C_0 = \mathbb{1}$ for determining the first search direction in line 6, but then rescale it to $C_0 = \gamma \mathbb{1}$ with $\gamma = (\rho_0 \|\mathbf{y}_0\|^2)^{-1}$ before it is used to update C_1 in line 14.

Note that initial guesses with very small gradients (close to critical points of f) should be avoided, since they could trigger premature termination of the algorithm and/or require very large or very small step lengths², which would require many line search iterations.

Let us discuss the computational cost and memory cost of the BFGS algorithm. Without any assumptions about the problem, we can not assume anything about the cost of evaluating function values or gradients of the objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. So we limit the discussion to the rest of the algorithm. The dominant memory cost is storing the inverse Hessian approximation $C_k \in \mathbb{R}^{n \times n}$, so we have a memory cost of $\mathcal{O}(n^2)$. The dominant contributions to the computational cost are products of C_k with vectors in \mathbb{R}^n , as well as outer products of \mathbb{R}^n vectors, so we have a computational cost of $\mathcal{O}(n^2)$. This cost can become unfeasible, if n gets large, which motivates the Limited-Memory-BFGS (or L-BFGS) method, which we will discuss in the following.

²At the beginning of the algorithm, the search directions are not yet well-scaled.

2.3.2 The L-BFGS Method

First, we notice that the inverse Hessian approximation C_k of the BFGS algorithm is uniquely determined by the vectors \mathbf{y}_k and \mathbf{s}_k and the initial approximation C_0 . Note also that the algorithm only needs C_k at one point, namely when it computes the product of C_k with a vector. So for the first $n/2$ iterations, it would require less memory to just store $\{\mathbf{y}_k\}$ and $\{\mathbf{s}_k\}$ and compute the product $C_k \cdot \mathbf{g}_k$ on the fly.

Let us illustrate how that would happen. If we want to compute the product $C_k \cdot \mathbf{q}$ for a vector $\mathbf{q} \in \mathbb{R}^n$, we can do this by iterating backwards through the BFGS updates (2.27). We define recursively

$$\mathbf{q}_{k-1} := \mathbf{q}_k - a_{k-1} \mathbf{y}_{k-1}, \quad (2.29)$$

starting at $\mathbf{q}_k := \mathbf{q}$, with scale factors $a_k := \rho_k \mathbf{s}_k^T \cdot \mathbf{q}_{k+1}$. Furthermore define

$$\mathbf{r}_k := C_k \cdot \mathbf{q}_k \quad (2.30)$$

and scale factors $b_k := \rho_k \mathbf{y}_k^T \cdot \mathbf{r}_k$. Then we can expand $C_k \cdot \mathbf{q}$ using the BFGS update (2.27):

$$\begin{aligned} \mathbf{r}_k &= C_k \cdot \mathbf{q}_k \\ &= (\mathbb{I} - \rho_{k-1} \mathbf{s}_{k-1} \mathbf{y}_{k-1}^T) \cdot C_{k-1} \cdot (\mathbb{I} - \rho_{k-1} \mathbf{y}_{k-1} \mathbf{s}_{k-1}^T) \cdot \mathbf{q}_k + \rho_{k-1} \mathbf{s}_{k-1} \mathbf{s}_{k-1}^T \cdot \mathbf{q}_k \\ &= (\mathbb{I} - \rho_{k-1} \mathbf{s}_{k-1} \mathbf{y}_{k-1}^T) \cdot C_{k-1} \cdot \mathbf{q}_{k-1} + a_{k-1} \mathbf{s}_{k-1} \\ &= \mathbf{r}_{k-1} - b_{k-1} \mathbf{s}_{k-1} + a_{k-1} \mathbf{s}_{k-1}. \end{aligned} \quad (2.31)$$

This means we can compute $C_k \cdot \mathbf{q}$ by first iterating from \mathbf{q}_k to \mathbf{q}_0 using equation (2.29). Here, we can perform $\mathbf{r}_0 = C_0 \cdot \mathbf{q}_0$, since C_0 is known. Typically, C_0 is diagonal or even a multiple of \mathbb{I} , which makes this product cheap to compute. Then we can iterate from \mathbf{r}_0 to \mathbf{r}_k (which is the desired result), using equation (2.31).

The L-BFGS method [66, 67] uses this scheme to compute products of the approximate inverse-Hessian with a vector, but with one key modification [51, p.177f]. It only goes back to \mathbf{q}_{k-m} , where m is the *history size*³. Then it makes an educated guess for C_{k-m} , and finally iterates from \mathbf{r}_{k-m} to \mathbf{r}_k . Intuitively, it only considers curvature information of the last m iterates to approximate the inverse Hessian.

Computing $C_k \cdot \mathbf{q}$ in this fashion is known as the *two loop recursion*, which we summarize in algorithm 5.

For the L-BFGS algorithm, we supply the two-loop recursion algorithm with an educated guess \tilde{C}_{k-m} for C_{k-m} . Note that \tilde{C}_{k-m} has to be sparse, or else the memory benefits become obsolete. A properly scaled multiple of the identity has proven to be effective [68]:

$$\tilde{C}_{k-m} := \frac{1}{\rho_{k-1} \|\mathbf{y}_{k-1}\|^2} \mathbb{I}. \quad (2.32)$$

In summary, we get the L-BFGS method by replacing the exact matrix-multiplication $C_k \cdot \nabla f(\mathbf{x}_k)$ in the BFGS method (algorithm 4) by the two-loop recursion (algorithm 5). We summarize the L-BFGS method in algorithm 6.

³For $k < m$ it only goes back to \mathbf{q}_0 .

Algorithm 5: L-BFGS two-loop recursion

Input: Histories $\{\mathbf{s}_i\}$ and $\{\mathbf{y}_i\}$ for $i = k - m, \dots, k - 1$ A vector $\mathbf{q} \in \mathbb{R}^n$ A matrix $C_{k-m} \in \mathbb{R}^{n \times n}$ (typically sparse)**Output:** The product $C_k \cdot \mathbf{q}$, where C_k is related to C_{k-m} by equation (2.27)

```

1 for  $i = k - 1, \dots, k - m$  do
2    $\rho_i \leftarrow (\mathbf{y}_i \cdot \mathbf{s}_i)^{-1}$ 
3    $\mathbf{a}_i \leftarrow \rho_i \mathbf{s}_i^T \cdot \mathbf{q}$ 
4    $\mathbf{q} \leftarrow \mathbf{q} - \mathbf{a}_i \mathbf{y}_i$ 
5  $\mathbf{r} \leftarrow C_{k-m} \cdot \mathbf{q}$ 
6 for  $i = k - m, \dots, k - 1$  do
7    $\mathbf{b} \leftarrow \rho_i \mathbf{y}_i^T \cdot \mathbf{r}$ 
8    $\mathbf{r} \leftarrow \mathbf{r} + (\mathbf{a}_i - \mathbf{b}) \mathbf{s}_i$ 
9 return  $\mathbf{r}$ 

```

Algorithm 6: L-BFGS

Input: Initial guess \mathbf{x}_0 Threshold $\epsilon > 0$ History size $m \in \mathbb{N}_{>0}$ **Output:** Approximate minimizer \mathbf{x}_{\min} of $f : \mathbb{R}^n \rightarrow \mathbb{R}$

```

1 Evaluate  $\nabla f(\mathbf{x}_0)$ 
2 if  $\|\nabla f(\mathbf{x}_0)\| < \epsilon$  then
3   return  $\mathbf{x}_0$ 
4  $k \leftarrow 0$ 
5 repeat
6   Choose  $\tilde{C}_{k-m}$ , e.g. equation (2.32)
7   Compute search direction  $\mathbf{p}_k \leftarrow -C_k \cdot \nabla f(\mathbf{x}_k)$  using algorithm 5
8   Find step length  $\alpha_k$  that fulfills strong Wolfe conditions (2.11), e.g. algorithm 2
9    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$ 
10  Evaluate  $\nabla f(\mathbf{x}_{k+1})$ 
11  if  $\|\nabla f(\mathbf{x}_{k+1})\| < \epsilon$  then
12    return  $\mathbf{x}_{k+1}$ 
13  if  $k > m$  then
14    Discard  $\mathbf{s}_{k-m}, \mathbf{y}_{k-m}$  from memory
15     $\mathbf{s}_k \leftarrow \mathbf{x}_{k+1} - \mathbf{x}_k$ 
16     $\mathbf{y}_k \leftarrow \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$ 
17     $k \leftarrow k + 1$ 

```

At $k = 0$ we choose $C_0 = \mathbb{1}$.

This procedure only requires $\{\mathbf{y}_k, \dots, \mathbf{y}_{k-m}\}$ and $\{\mathbf{s}_k, \dots, \mathbf{s}_{k-m}\}$ to be stored in memory instead of the full matrix C_k , making the memory cost $\mathcal{O}(nm)$. The computational cost also scales as $\mathcal{O}(nm)$, since the two-loop recursion requires scalar products of n -dimensional vectors in $2m$ loop iterations.

From a standpoint of implementation, the L-BFGS algorithm (in contrast to the BFGS) has the advantage that it only operates on vectors of the same type as the arguments \mathbf{x} of the cost function, and not on matrices. This means that as long as we can perform vector addition, multiplication with scalars and scalar products of two vectors, we can allow the vectors to have more complicated data structures than simple row-vectors. The BFGS method, on the other hand would require a “matrix” datatype, which is only readily available if the “vector” datatype is a row-vector or higher-dimensional tensor.

2.4 Functions in Complex Variables

In the previous sections we have discussed strategies for optimizing real-to-real functions. In quantum many-body systems, we are, however, typically dealing with functions of complex arguments. We do not need to deal with complex function-values, since that would make a minimizer ill-defined.

In principle, we could optimize a function $f : \mathbb{C}^n \rightarrow \mathbb{R}$, just by considering its real counterpart

$$\tilde{f} : \mathbb{R}^{2n} \rightarrow \mathbb{R}, \quad \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \mapsto \tilde{f}(\mathbf{x}, \mathbf{y}) := f(\mathbf{x} + \mathbf{i}\mathbf{y}). \quad (2.33)$$

Then we get the minimizer \mathbf{z}_{\min} of f from the minimizer $(\mathbf{x}_{\min}, \mathbf{y}_{\min})$ of \tilde{f} as

$$\mathbf{z}_{\min} = \mathbf{x}_{\min} + \mathbf{i}\mathbf{y}_{\min}. \quad (2.34)$$

So, in principle, we could apply the algorithms we have developed to \tilde{f} , obtain the minimizer of f via equation (2.34) and call it a day.

This approach, however, has drawbacks and a formulation in terms of complex iterates and holomorphic derivatives is desirable. If the derivatives are obtained analytically or by numerically approximating an analytical expression, the real derivatives (of \tilde{f} w.r.t. \mathbf{x} and \mathbf{y}) could disguise structures which might be exploited. If we want to use automatic differentiation to obtain the gradients used for optimization, the result will be the holomorphic derivative of f , not the derivatives of \tilde{f} . Additionally, the complex formulation which we present in the rest of this section allows for a shared implementation of real and complex optimization algorithms which is not possible with the “real” approach of optimizing \tilde{f} .

We therefore want to develop optimization algorithms for complex functions, that use their holomorphic (and/or anti-holomorphic) derivatives as building blocks. In [69], the derivation of such algorithms is discussed in detail, essentially by repeating all steps that we have outlined in the previous sections, but for the complex function. In particular, the authors construct a complex Taylor expansion, use it as a model and develop line search methods from it.

We will follow a more direct approach and consider the optimization algorithms for \tilde{f} , and then translate all computations from real iterates in \mathbb{R}^{2n} , function values of \tilde{f} and derivatives of \tilde{f} to complex iterates in \mathbb{C}^n , as well as function values and (holomorphic and/or anti-holomorphic) derivatives of f .

We discuss the complex (Wirtinger) calculus [70], holomorphic and anti-holomorphic derivatives in detail in the appendix B. For a brief recap, the holomorphic and anti-holomorphic derivatives of f are respectively:

$$\begin{aligned}\frac{\partial f}{\partial z_j}(\mathbf{x} + \mathbf{i}\mathbf{y}) &= \frac{1}{2} \left(\frac{\partial \tilde{f}}{\partial x_j}(\mathbf{x}, \mathbf{y}) - \mathbf{i} \frac{\partial \tilde{f}}{\partial y_j}(\mathbf{x}, \mathbf{y}) \right); \\ \frac{\partial f}{\partial z_j^*}(\mathbf{x} + \mathbf{i}\mathbf{y}) &= \frac{1}{2} \left(\frac{\partial \tilde{f}}{\partial x_j}(\mathbf{x}, \mathbf{y}) + \mathbf{i} \frac{\partial \tilde{f}}{\partial y_j}(\mathbf{x}, \mathbf{y}) \right).\end{aligned}\tag{2.35}$$

Since f is real-valued, we get the relation

$$\frac{\partial f}{\partial z_j^*} = \left(\frac{\partial f^*}{\partial z_j} \right)^* = \left(\frac{\partial f}{\partial z_j} \right)^*,\tag{2.36}$$

where * denotes complex conjugation. We write for the holomorphic and anti-holomorphic gradient

$$\nabla f := \begin{pmatrix} \frac{\partial f}{\partial z_1} \\ \vdots \\ \frac{\partial f}{\partial z_n} \end{pmatrix} \quad ; \quad \nabla^* f := \begin{pmatrix} \frac{\partial f}{\partial z_1^*} \\ \vdots \\ \frac{\partial f}{\partial z_n^*} \end{pmatrix}.\tag{2.37}$$

Throughout this section, we will denote the real counterparts of complex objects by adding a tilde on top. In order to express the relation between the two counterparts, it will be convenient to use the matrix J :

$$J := \begin{pmatrix} \mathbb{1} & \mathbf{i}\mathbb{1} \\ \mathbb{1} & -\mathbf{i}\mathbb{1} \end{pmatrix} \in \mathbb{C}^{2n \times 2n} \quad ; \quad \frac{1}{2} J^\dagger J = \mathbb{1}.\tag{2.38}$$

For example, we can define iterates $\mathbf{z}_k \in \mathbb{C}^n$ from the iterates $(\mathbf{x}_k, \mathbf{y}_k) \in \mathbb{R}^{2n}$ in the optimization of \tilde{f} as

$$\begin{pmatrix} \mathbf{z}_k \\ \mathbf{z}_k^* \end{pmatrix} := J \begin{pmatrix} \mathbf{x}_k \\ \mathbf{y}_k \end{pmatrix} = \begin{pmatrix} \mathbf{x}_k + \mathbf{i}\mathbf{y}_k \\ \mathbf{x}_k - \mathbf{i}\mathbf{y}_k \end{pmatrix} = \begin{pmatrix} \mathbf{x}_k + \mathbf{i}\mathbf{y}_k \\ (\mathbf{x}_k + \mathbf{i}\mathbf{y}_k)^* \end{pmatrix}\tag{2.39}$$

and by definition, the complex derivatives of f are

$$\begin{pmatrix} \nabla f(\mathbf{x} + \mathbf{i}\mathbf{y}) \\ \nabla^* f(\mathbf{x} + \mathbf{i}\mathbf{y}) \end{pmatrix} = \frac{1}{2} J^* \cdot \nabla \tilde{f}(\mathbf{x}, \mathbf{y}) = \frac{1}{2} J^* \cdot \begin{pmatrix} \nabla_x \tilde{f}(\mathbf{x}, \mathbf{y}) \\ \nabla_y \tilde{f}(\mathbf{x}, \mathbf{y}) \end{pmatrix}.\tag{2.40}$$

A common pattern throughout this section will be to write equations like (2.39) as two redundant lines of a complex quantity and its conjugate. This apparent redundancy will allow our notation to be more concise. In particular, we are able to insert a resolution of the identity in terms of J .

2.4.1 Line Search

We first discuss how the line search methods of section 2.1 can be translated to the complex case.

We assume that we have a real search direction $\tilde{\mathbf{p}}_k = (\tilde{\mathbf{p}}_{k,x}, \tilde{\mathbf{p}}_{k,y}) \in \mathbb{R}^{2n}$ for \tilde{f} and define the complex search direction

$$\begin{pmatrix} \mathbf{p}_k \\ \mathbf{p}_k^* \end{pmatrix} := J \cdot \tilde{\mathbf{p}}_k = \begin{pmatrix} \tilde{\mathbf{p}}_{k,x} + i\tilde{\mathbf{p}}_{k,y} \\ \tilde{\mathbf{p}}_{k,x} - i\tilde{\mathbf{p}}_{k,y} \end{pmatrix}. \quad (2.41)$$

When performing the line search, for example in algorithms 1 and 2, we only worked with the univariate (real-to-real) function $\phi(\alpha)$ defined on the line starting at $\tilde{\mathbf{z}}_k := (\tilde{\mathbf{x}}_k, \tilde{\mathbf{y}}_k)$ and going in the direction $\tilde{\mathbf{p}}_k$. We can express ϕ in terms of f instead of \tilde{f}

$$\phi(\alpha) = \tilde{f}(\tilde{\mathbf{z}}_k + \alpha\tilde{\mathbf{p}}_k) = f(\mathbf{z}_k + \alpha\mathbf{p}_k) \quad (2.42)$$

and the derivative of ϕ can be computed as

$$\begin{aligned} \phi'(\alpha) &\stackrel{(2.5)}{=} \nabla \tilde{f}(\tilde{\mathbf{z}}_k + \alpha\tilde{\mathbf{p}}_k)^T \cdot \tilde{\mathbf{p}}_k \\ &\stackrel{(2.38)}{=} \begin{pmatrix} \nabla f(\mathbf{z}_k + \alpha\mathbf{p}_k)^T & \nabla^* f(\mathbf{z}_k + \alpha\mathbf{p}_k)^T \end{pmatrix} \cdot \begin{pmatrix} \mathbf{p}_k \\ \mathbf{p}_k^* \end{pmatrix} \\ &\stackrel{(2.36)}{=} 2 \operatorname{Re} [\nabla f(\mathbf{z}_k + \alpha\mathbf{p}_k)^T \cdot \mathbf{p}_k]. \end{aligned} \quad (2.43)$$

With these modifications, all our discussion about line search algorithms still applies. Also note that the real and complex cases can share a single implementation, since for a real-to-real function the holomorphic derivatives are just half of the conventional real derivatives, and taking the real part has no effect, if all quantities are real to begin with.

In summary, to perform line search for a complex-to-real function $f : \mathbb{C}^n \rightarrow \mathbb{R}$, the algorithms 1 and 2 are still valid, if ϕ and ϕ' are computed according to equations (2.42) and (2.43), respectively.

2.4.2 The Generalized Gradient Descent Method for Functions in Complex Variables

In the gradient descent method (see section 2.2), we use the gradient of the objective function to update the iterates.

If the objective function $f : \mathbb{C}^n \rightarrow \mathbb{R}$ is a function of complex variables, it is convenient to define

$$\mathbf{g}_k := 2\nabla^* f(\mathbf{z}_k) = 2 \begin{pmatrix} \frac{\partial f}{\partial z_1^*}(\mathbf{z}_k) \\ \vdots \\ \frac{\partial f}{\partial z_n^*}(\mathbf{z}_k) \end{pmatrix}, \quad (2.44)$$

which is the vector of anti-holomorphic derivatives of f , with an additional factor two. This factor serves two purposes. First, its conjugate \mathbf{g}_k^\star is the result of automatic differentiation for the complex function f . See section 3.4 for more details. Secondly, this definition allows us to write a shared implementation for the complex and real cases, since if f were a function of real variables, \mathbf{g}_k would be its standard real gradient evaluated at the k^{th} iterate.

In the gradient-descent algorithm for the real counterpart \tilde{f} , the iterates are updated as $\tilde{\mathbf{z}}_{k+1} = \tilde{\mathbf{z}}_k + \alpha_k \nabla \tilde{f}(\tilde{\mathbf{z}}_k)$. Using equation (2.39) we obtain the complex iterates

$$\begin{pmatrix} \mathbf{z}_{k+1} \\ \mathbf{z}_{k+1}^\star \end{pmatrix} = J \cdot \begin{pmatrix} \mathbf{x}_{k+1} \\ \mathbf{y}_{k+1} \end{pmatrix} = J \left[\begin{pmatrix} \mathbf{x}_k \\ \mathbf{y}_k \end{pmatrix} + \alpha_k \begin{pmatrix} \nabla_x \tilde{f}(\mathbf{x}_k, \mathbf{y}_k) \\ \nabla_y \tilde{f}(\mathbf{x}_k, \mathbf{y}_k) \end{pmatrix} \right] = \begin{pmatrix} \mathbf{z}_k + \alpha_k \mathbf{g}_k \\ \mathbf{z}_k^\star + \alpha_k \mathbf{g}_k^\star \end{pmatrix}. \quad (2.45)$$

2.4.3 The Generalized L-BFGS Method for Functions in Complex Variables

In order to translate the computations of the (real) L-BFGS algorithm of the real counterpart $\tilde{f} : \mathbb{R}^{2n} \rightarrow \mathbb{R}$ of a complex objective function $f : \mathbb{C}^n \rightarrow \mathbb{R}$, we define complex variables corresponding to the real variables in the optimization algorithm of \tilde{f} . To highlight their correspondence, we denote them by the same symbols, and decorate the real variables with a tilde on top.

In particular, we define

$$\begin{pmatrix} \mathbf{s}_k \\ \mathbf{s}_k^\star \end{pmatrix} := J \cdot \tilde{\mathbf{s}}_k = J \cdot (\tilde{\mathbf{z}}_{k+1} - \tilde{\mathbf{z}}_k) = \begin{pmatrix} \mathbf{z}_{k+1} - \mathbf{z}_k \\ (\mathbf{z}_{k+1} - \mathbf{z}_k)^\star \end{pmatrix}, \quad (2.46)$$

$$\begin{pmatrix} \mathbf{g}_k \\ \mathbf{g}_k^\star \end{pmatrix} := J \cdot \tilde{\mathbf{g}}_k = J \cdot \begin{pmatrix} \nabla_x \tilde{f}(\tilde{\mathbf{z}}_k) \\ \nabla_y \tilde{f}(\tilde{\mathbf{z}}_k) \end{pmatrix} = 2 \begin{pmatrix} \nabla^\star f(\mathbf{z}_k) \\ \nabla f(\mathbf{z}_k) \end{pmatrix} = \begin{pmatrix} 2\nabla^\star f(\mathbf{z}_k) \\ (2\nabla^\star f(\mathbf{z}_k))^\star \end{pmatrix}, \quad (2.47)$$

$$\begin{pmatrix} \mathbf{y}_k \\ \mathbf{y}_k^\star \end{pmatrix} := J \cdot \tilde{\mathbf{y}}_k = J \cdot (\tilde{\mathbf{g}}_{k+1} - \tilde{\mathbf{g}}_k) = \begin{pmatrix} \mathbf{g}_{k+1} - \mathbf{g}_k \\ (\mathbf{g}_{k+1} - \mathbf{g}_k)^\star \end{pmatrix}. \quad (2.48)$$

This allows us to translate the computations in the L-BFGS two-loop recursion and in the main loop. For example,

$$\frac{1}{\rho_k} = \tilde{\mathbf{y}}_k^\top \cdot \tilde{\mathbf{s}}_k = \frac{1}{2} \tilde{\mathbf{y}}_k^\top \cdot J^\dagger \cdot J \cdot \tilde{\mathbf{s}}_k = \frac{1}{2} \begin{pmatrix} \mathbf{y}_k^\dagger & \mathbf{y}_k^\top \end{pmatrix} \cdot \begin{pmatrix} \mathbf{s}_k \\ \mathbf{s}_k^\star \end{pmatrix} = \frac{1}{2} \mathbf{y}_k^\dagger \cdot \mathbf{s}_k + \text{c.c.} = \text{Re}[\mathbf{y}_k^\dagger \cdot \mathbf{s}_k]. \quad (2.49)$$

Translating all computations of algorithm 5 in this way, gives us the *generalized L-BFGS two-loop recursion*, which we summarize in algorithm 7.

Contrary to algorithm 5, we have fixed the choice of initial Hessian approximation to equation (2.32), though in principle other choices for \tilde{C}_{k-m} can be incorporated by adapting lines 6 and 7.

Note that algorithm 7 is equivalent to the result of [69].

Algorithm 7: Generalized L-BFGS two-loop recursion

Input: Histories $\{\mathbf{s}_i\}$ and $\{\mathbf{y}_i\}$ for $i = k - m, \dots, k - 1$
 \mathbf{g}_k as defined in (2.47)

Output: L-BFGS search direction \mathbf{p}_k as defined in equation (2.41)

```

1  $\mathbf{q} \leftarrow -\mathbf{g}_k$ 
2 for  $i = k - 1, \dots, k - m$  do
3    $\rho_i \leftarrow \text{Re}[\mathbf{y}_i^\dagger \cdot \mathbf{s}_i]^{-1}$ 
4    $a_i \leftarrow \rho_i \text{Re}[\mathbf{s}_i^\dagger \cdot \mathbf{q}]$ 
5    $\mathbf{q} \leftarrow \mathbf{q} - a_i \mathbf{y}_i$ 
6  $\gamma \leftarrow \left( \rho_{k-1} \mathbf{y}_{k-1}^\dagger \cdot \mathbf{y}_{k-1} \right)^{-1}$ 
7  $\mathbf{r} \leftarrow \gamma \mathbf{q}$ 
8 for  $i = k - m, \dots, k - 1$  do
9    $b \leftarrow \rho_i \text{Re}[\mathbf{y}_i^\dagger \cdot \mathbf{r}]$ 
10   $\mathbf{r} \leftarrow \mathbf{r} + (a_i - b) \mathbf{s}_i$ 
11 return  $\mathbf{r}$ 

```

With this change of the two-loop recursion, the L-BFGS method (algorithm 6) can also be used on functions of complex variables. Moreover, a shared implementation for the cases of real or complex variables is possible, since algorithm 7 reduces to algorithm 5 if all quantities are real.

Chapter 3

Automatic Differentiation

Contents

| | | |
|------------|--|-----------|
| 3.1 | Jacobian Vector Products and Vector Jacobian Products . . . | 27 |
| 3.1.1 | Jacobian Vector Products | 27 |
| 3.1.2 | Vector Jacobian Products | 28 |
| 3.1.3 | Chain Rules | 29 |
| 3.2 | Chained-Primitive Functions | 30 |
| 3.2.1 | Primitive Functions | 30 |
| 3.2.2 | Forward Mode for Chained-Primitive Functions | 31 |
| 3.2.3 | Backward Mode for Chained-Primitive Functions | 33 |
| 3.2.4 | An Example: Derivatives from forward and backward mode . . . | 35 |
| 3.3 | General Computation Graphs | 37 |
| 3.3.1 | Computation Graphs | 37 |
| 3.3.2 | Forward mode on General Computation Graphs | 39 |
| 3.3.3 | Backward mode on General Computation Graphs | 40 |
| 3.3.4 | Summary: How-To Forward Mode | 43 |
| 3.3.5 | Summary: How-To Backward Mode | 44 |
| 3.4 | Automatic Differentiation for Complex Functions | 46 |
| 3.4.1 | Complex JVPs and VJPs | 46 |
| 3.4.2 | Forward Mode | 48 |
| 3.4.3 | Backward Mode | 49 |
| 3.5 | Implementation in Jax for Python | 50 |
| 3.6 | Review of Automatic Differentiation Tools | 51 |
| 3.7 | A Short Glossary of Terminology and Jargon | 54 |

Automatic differentiation (autodiff) is a tool that can compute derivatives of functions numerically, if a numerical implementation of the function itself is available [47, 71–75]. Interest in automatic differentiation has arisen in the field of machine learning and artificial intelligence, where derivatives of a target function (loss function) are used to train neural networks. Additional applications include, to name just a few, optimisation of race car performance [76], meteorological modelling [77], simulation of fluid dynamics [78] and finance [79]. In this thesis, we will use automatic differentiation to supply the numerical optimization algorithms of chapter 2 with gradients.

In the following sections of this chapter, we introduce Jacobian-Vector-Products (JVPs) and Vector-Jacobian-Products (VJPs), the mathematical language of autodiff (section 3.1). We demonstrate how they can be used in a simplified application (section 3.2), thereby introducing the two modes of operation, forward mode and backward mode. In section 3.3, we discuss autodiff for the general setting. We aim to compile an instructive explanation of all topics involve in the main text and found a more rigorous but less readable approach to the problem, which we discuss in appendix C.1.

All of the above is limited to real-to-real functions. Autodiff for complex functions is discussed in section 3.4. We showcase the interface of `jax`, an autodiff package for `python` in section 3.5. Finally, we review autodiff packages for various languages (and in particular showcase `jax` for `python`) in section 3.6 and provide a short glossary of the various jargon and terminology in section 3.7. An exemplary collection of autodiff formulae (jvp and vjp rules) can be found in appendix C.3.

3.1 Jacobian Vector Products and Vector Jacobian Products

3.1.1 Jacobian Vector Products

The *Jacobian vector product* (JVP) of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ can be thought of as a function evaluating the product of the Jacobian of f with a vector. We present automatic differentiation (in *forward mode*) as a scheme to obtain a numerically efficient algorithm for computing the JVP of a function, given a numerically efficient implementation of the function itself. This can be used to obtain derivatives, as shown in equation (3.5).

Definition

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a differentiable function. For any point $\mathbf{x} \in \mathbb{R}^n$, the Jacobian of f at \mathbf{x} is a linear map $Df(\mathbf{x})$. Equivalently, after choosing the standard bases of \mathbb{R}^n and \mathbb{R}^m , it is a matrix $Df(\mathbf{x}) \in \mathbb{R}^{m \times n}$ with $(Df(\mathbf{x}))_{ij} = \frac{\partial f_i}{\partial x_j}(\mathbf{x})$. Therefore, the Jacobian Df can be thought of as a map from \mathbb{R}^n to the space of linear maps from \mathbb{R}^n to \mathbb{R}^m , i.e. $\text{Hom}_{\mathbb{R}}(\mathbb{R}^n, \mathbb{R}^m)$. That is,

$$Df : \mathbb{R}^n \rightarrow \text{Hom}_{\mathbb{R}}(\mathbb{R}^n, \mathbb{R}^m), \mathbf{x} \mapsto Df(\mathbf{x}). \quad (3.1)$$

The Jacobian vector product JVP_f of f can be thought of as a de-curried version of the Jacobian: Df is a function (of a point $\mathbf{x} \in \mathbb{R}^n$) that returns another function (say of a vector $\mathbf{v} \in \mathbb{R}^n$). The Jacobian vector product JVP_f directly takes \mathbf{x} and \mathbf{v} as inputs and returns the same output:

$$\text{JVP}_f : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m, (\mathbf{x}, \mathbf{v}) \mapsto Df(\mathbf{x}) \cdot \mathbf{v}. \quad (3.2)$$

This definition can be extended to functions between general real vector spaces. Let U and V be normed \mathbb{R} -vector spaces that are complete w.r.t. their respective norms and let $g : U \rightarrow V$ be a differentiable¹ function. We define the Jacobian vector product JVP_g of g as

$$\text{JVP}_g : U \times U \rightarrow V, (x, v) \mapsto \lim_{h \rightarrow 0} \frac{g(x + hv) - g(x)}{h}. \quad (3.3)$$

This definition includes equation (3.2) as a special case.

The first argument (here \mathbf{x} or x) of the JVP is commonly referred to as the *primal value*, the second argument (here \mathbf{v} or v) as the *tangent*.

Compatibility with Isomorphisms

In practice, we are often dealing with data structures that are a collection of real numbers, such as lists, tuples, arrays or nested structures of such containers. These structures are isomorphic to \mathbb{R}^n for some $n \in \mathbb{N}$. A possible isomorphism involves reordering the real numbers of the data structure into a column vector in any fixed order.

Intuitively the JVP of a function that maps between such spaces of data structures is “the equivalent of” equation (3.2). To make this notion more rigorous we show that the JVP is compatible with isomorphisms.

¹This just means that the limit in equation (3.3) exists for all x and v .

Let U, V and \tilde{U}, \tilde{V} be normed and complete \mathbb{R} -vector spaces, and $\alpha : U \xrightarrow{\cong} \tilde{U}$ and $\beta : V \xrightarrow{\cong} \tilde{V}$ isomorphisms. For any $x, v \in U$, the Jacobian vector product of a differentiable function $g : U \rightarrow V$ fulfills

$$\text{JVP}_g(x, v) = \beta^{-1}(\text{JVP}_{\tilde{g}}(\alpha(x), \alpha(v))), \quad (3.4)$$

where $\tilde{g} : \tilde{U} \rightarrow \tilde{V}$, $\tilde{x} \mapsto \beta(g(\alpha^{-1}(\tilde{x})))$.

Due to the above property, we can from now on just write datastructures as vectors in \mathbb{R}^n and work with the definition (3.37).

Derivatives from JVPs

We can obtain the derivatives of all components of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ w.r.t. one of its input components, say x_j for some $j \in \{1, \dots, n\}$, via the JVP of f by using an appropriate unit vector as the second argument. For any $\mathbf{x} \in \mathbb{R}^n$

$$\frac{\partial f}{\partial x_j}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_j}(\mathbf{x}) \\ \vdots \\ \frac{\partial f_m}{\partial x_j}(\mathbf{x}) \end{pmatrix} = Df(\mathbf{x}) \cdot \mathbf{e}_j = \text{JVP}_f(\mathbf{x}, \mathbf{e}_j). \quad (3.5)$$

More generally, for a point $\mathbf{x} \in \mathbb{R}^n$ and a direction $\mathbf{v} \in \mathbb{R}^n$, the directional derivative of f at \mathbf{x} in direction \mathbf{v} is given by $\text{JVP}_f(\mathbf{x}, \mathbf{v})$.

3.1.2 Vector Jacobian Products

For functions between real coordinate spaces (\mathbb{R}^n for some $n \in \mathbb{N}$), the JVP (introduced above) is a product of the Jacobian matrix with a (column-)vector (on its right). It will prove useful to also define a product of a (row-)vector (on the left) with the Jacobian.

We present automatic differentiation (in *backward mode*) as a scheme to obtain a numerically efficient algorithm for computing such vector Jacobian products of functions, given a numerically efficient implementation of the function itself. This can be used to obtain derivatives, as in equation (3.9).

Definition

We define the *vector Jacobian product* VJP_f of a differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ as

$$\text{VJP}_f : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n, (\mathbf{x}, \mathbf{v}^T) \mapsto \mathbf{v}^T \cdot Df(\mathbf{x}). \quad (3.6)$$

Here, we have simply written \mathbb{R}^m to also mean the space of row-vectors.

In a rigorous definition, the second argument, as well as the output of the VJP live in the dual space of the target space of f . The dual space of a vector space U is the space of all linear forms on U , in other words all maps

$$U^* = \{w : U \rightarrow \mathbb{R} \mid w \text{ linear}\}, \quad (3.7)$$

where we have assumed the reals as the underlying field for concreteness.

For \mathbb{R}^m (and in fact any space with a scalar product), we can identify elements of the original space with elements of the dual space. For example, with $\mathbf{v} \in \mathbb{R}^m$ we associate the

covector ($\mathbf{a} \mapsto \mathbf{v}^T \cdot \mathbf{a}$) $\in (\mathbb{R}^m)^*$ in the dual space. In equation (3.38), we have just written the row-vector \mathbf{v}^T for this map and understand $\text{VJP}_f(\mathbf{x}, \mathbf{v}^T)$, which is another row-vector, as a map of the same form, and therefore an element of the dual space of \mathbb{R}^n .

For a differentiable function $g : U \rightarrow V$ between general (real, normed, complete) vector spaces U and V , we define the VJP as the transpose of the JVP (in their respective second arguments). That is,

$$\text{VJP}_g : U \times V^* \rightarrow U^*, (x, v) \mapsto \text{VJP}_g(x, v) := \left[a \mapsto v(\text{JVP}_g(x, a)) \right] \in U^*. \quad (3.8)$$

Note that this definition includes equation (3.6) as a special case. Due to the compatibility with isomorphisms (3.4), we can work with coordinate spaces \mathbb{R}^n for all data structures that are isomorphic to them.

The first argument (here \mathbf{x} or x) of the VJP is commonly referred to as the *primal value*, the second argument (here \mathbf{v} or v) as the *cotangent*.

The VJP corresponds to the “adjoined function” in [48].

Derivatives from VJPs

We can obtain the derivatives of a single component of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, say f_i for some $i \in \{1, \dots, m\}$, w.r.t. all of its input components via the VJP of f :

$$\frac{\partial f_i}{\partial \mathbf{x}}(\mathbf{x}) = \left(\frac{\partial f_i}{\partial x_1}(\mathbf{x}) \quad \dots \quad \frac{\partial f_i}{\partial x_n}(\mathbf{x}) \right) = \mathbf{e}_i^T \cdot Df(\mathbf{x}) = \text{VJP}_f(\mathbf{x}, \mathbf{e}_i^T). \quad (3.9)$$

3.1.3 Chain Rules

The JVP fulfills a chain rule. For differentiable functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$ and $g : \mathbb{R}^k \rightarrow \mathbb{R}^m$, a primal value $x \in \mathbb{R}^n$ and a tangent $v \in \mathbb{R}^k$ we obtain

$$\text{JVP}_{g \circ f}(\mathbf{x}, \mathbf{v}) = \text{JVP}_g(f(\mathbf{x}), \text{JVP}_f(\mathbf{x}, \mathbf{v})). \quad (3.10)$$

This follows directly from the chain-rule for the Jacobian of $g \circ f$.

Similarly, we also get a chain-rule for the VJP. For a primal $\mathbf{x} \in \mathbb{R}^n$ and a cotangent $\mathbf{w}^T \in \mathbb{R}^m$ we obtain

$$\text{VJP}_{g \circ f}(\mathbf{x}, \mathbf{w}^T) = \text{VJP}_f(\mathbf{x}, \text{VJP}_g(f(\mathbf{x}), \mathbf{w})). \quad (3.11)$$

3.2 Chained-Primitive Functions an Introduction to Automatic Differentiation

As mentioned in section 3.1, the goal of automatic differentiation (autodiff) is to find a way to compute JVPs and/or VJP of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ in a numerically efficient manner, *given a numerically efficient implementation of f* . This allows us to compute derivatives of f (see equations (3.5) and (3.9)).

In this section, we demonstrate how autodiff works for a class of functions with simple implementations and introduce some terminology on the way.

Let us first clarify this “simple” class of implementations. In this section, we only consider functions f , which are implemented by sequentially applying primitive functions (see the following subsection 3.2.1) to the input. This means that $f = g_N \circ \dots \circ g_1$ for some primitive functions g_1, \dots, g_N .

3.2.1 Primitive Functions

We use the concept of *primitive functions*. Which functions qualify as primitive is not a clear distinction, but rather a design choice that developers of autodiff packages have to make. Primitive functions are those functions for which the autodiff package implements the JVP and VJP directly. I.e. for every primitive function g , the package implements (or imports an external implementation of) an algorithm to compute the values of g , as well as an algorithm for JVP $_g$ or VJP $_g$ (ideally both). Note that this implies that primitives have to be differentiable.

A good intuitive picture is that primitive functions are all the numeric functions of the `numpy` package for `python`, except for those which are not differentiable, such as index-lookups or boolean-valued functions.

Autodiff can only be applied to functions that are implemented using only primitive functions as building blocks. Note that the primitives do not need to be applied sequentially.

For now, we will assume, that primitives are of the form $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$. This assumption is a significant simplification in two cases:

- The primitives could be have multiple inputs or multiple outputs. We discuss such primitives in section 3.3.
- The primitive could be defined only on a subset of \mathbb{R}^n . We will be pragmatic about this and just assume that the numerical implementation will give an error, if a function is ever called with an argument outside of its domain and stop worrying about function domains all together.

Implementation in Terms of Primitives

In practice, automatic differentiation happens as follows: The user implements code for a function f , using the toolbox of primitives that the autodiff package supplies.

For example, the user could write code for a Gaussian:

```
import jax.numpy as np

mu = np.array([1, 2, 3])
sigma = np.array([3, 4, 3])

def f(x):
    norm = np.sqrt(2 * np.pi) * sigma
    return np.exp(-.5 * np.sum( ((x - mu) / sigma) ** 2 )) / norm
```

and then ask the autodiff package (in this case `jax`) to differentiate it:

```
from jax import grad

x_trial = [42., 43., 42.]
g = grad(f)(x_trial)
```

In this case, the primitives are (in no particular order) the exponential function `np.exp`, multiplication `*`, sum over all elements `np.sum`, elementwise subtraction `-`, elementwise division `/` and elementwise exponentiation `**`.

Mathematically, we consider the function f as the composition of these primitive functions. The autodiff package has implementation for JVPs and/or VJPs of the primitives, and we need to obtain an algorithm for the JVP and/or VJP of the *top-level* function f .

3.2.2 Forward Mode for Chained-Primitive Functions

We will consider the simple case, where the implementation of the top-level function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, which we want to differentiate, consists only of sequentially applying three primitives. I.e. there are some primitives $g_1 : \mathbb{R}^n \rightarrow \mathbb{R}^k$, $g_2 : \mathbb{R}^k \rightarrow \mathbb{R}^l$ and $g_3 : \mathbb{R}^l \rightarrow \mathbb{R}^m$, such that $f = g_3 \circ g_2 \circ g_1$, or in terms of function values

$$f(\mathbf{x}) = g_3(g_2(g_1(\mathbf{x}))). \quad (3.12)$$

It is easy to generalize our argumentation to any number of primitives, and $f = g_N \circ \dots \circ g_1$, but this would only clutter our notation with many dots.

Using the JVP chain rule (equation (3.10)) twice, we obtain

$$\text{JVP}_f(\mathbf{x}, \mathbf{v}) = \text{JVP}_{g_3}\left(g_2(g_1(\mathbf{x})), \text{JVP}_{g_2}(g_1(\mathbf{x}), \text{JVP}_{g_1}(\mathbf{x}, \mathbf{v}))\right). \quad (3.13)$$

So we see that we can compute $\text{JVP}_f(\mathbf{x}, \mathbf{v})$, using only the primitive functions g_n and their JVPs. We also see that the intermediate values $g_1(\mathbf{x})$ and $g_2(g_1(\mathbf{x}))$ are necessary. Therefore, when we perform this computation, we can also compute the function value $f(\mathbf{x})$ at the same time, almost for free (only the last primitive g_3 is required additionally).

The notation of equation (3.13) is hard to read. Let us introduce abbreviations, that make it easier to understand and that are closer to code.

In `python`, an implementation of f might look like figure 3.1.

```

from favorite_autodiff_package import g1, g2, g3

def f(x):
    w1 = g1(x)
    w2 = g2(w1)
    y = g3(w2)
    return y

```

Figure 3.1: Example implementation of a chained-primitive function $f = g_3 \circ g_2 \circ g_1$, where g_1 , g_2 and g_3 are primitives of a generic autodiff package.

We explicitly named the intermediate and output values to refer to them more easily. In particular, we define $\mathbf{w}_1 := g_1(\mathbf{x})$, $\mathbf{w}_2 := g_2(\mathbf{w}_1)$ and the output $\mathbf{y} := g_3(\mathbf{w}_2) = f(\mathbf{x})$. These symbols represent the *primal values* (the respective first arguments of the JVPs in equation (3.13)).

Additionally, we abbreviate the tangents (the second arguments of the JVPs). Any primal value \mathbf{a} (either the input \mathbf{x} , one of the intermediates \mathbf{w}_n or the output \mathbf{y}) can be computed from the input \mathbf{x} via some function $\mathbf{a} = \phi(\mathbf{x})$. For example $\mathbf{w}_2 = (g_2 \circ g_1)(\mathbf{x})$. We define the *tangents* as

$$\dot{\mathbf{a}} := \text{JVP}_{\phi}(\mathbf{x}, \mathbf{v}). \quad (3.14)$$

They implicitly depend on \mathbf{x} and \mathbf{v} , which are understood from context since the overall goal of forward mode autodiff is to compute $\text{JVP}_f(\mathbf{x}, \mathbf{v})$ for some fixed \mathbf{x} and \mathbf{v} .

It follows from the chain rule (equation (3.10)) that

$$\dot{\mathbf{w}}_n = \text{JVP}_{g_n}(\mathbf{w}_{n-1}, \dot{\mathbf{w}}_{n-1}). \quad (3.15)$$

The tangents have an interpretation as derivatives (see equation (3.5)). If $\mathbf{v} = \mathbf{e}_j$ is a standard basis vector, then for any \mathbf{a} (either the input \mathbf{x} , one of the intermediates \mathbf{w}_n or the output \mathbf{y}) we have

$$\dot{\mathbf{a}} = \frac{\partial \mathbf{a}}{\partial x_j}. \quad (3.16)$$

Let us again re-write equation (3.13), this time in terms of the tangents of the intermediates.

$$\begin{aligned}
 \dot{\mathbf{y}} &= \text{JVP}_f(\mathbf{x}, \mathbf{v}) = \text{JVP}_{g_3}(\mathbf{w}_2, \dot{\mathbf{w}}_2), \\
 &\text{where } \dot{\mathbf{w}}_2 := \text{JVP}_{g_2}(\mathbf{w}_1, \dot{\mathbf{w}}_1), \\
 &\text{where } \dot{\mathbf{w}}_1 := \text{JVP}_{g_1}(\mathbf{x}, \dot{\mathbf{x}}), \\
 &\text{where } \dot{\mathbf{x}} := \mathbf{v}.
 \end{aligned} \quad (3.17)$$

Forward Mode Summarized

Equation (3.17) and its generalization to any number of primitives gives us the forward mode autodiff scheme for chained-primitive functions. We want to compute the Jacobian vector product $\text{JVP}_f(\mathbf{x}, \mathbf{v})$ of a chained-primitive function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ that is implemented as $f = g_N \circ \dots \circ g_1$ for $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{v} \in \mathbb{R}^n$. We require knowledge about the

order and nature of the primitives g_n , which we can obtain by analyzing the users implementation of f . We need to assume that the autodiff package has implementations for all primitives g_n and their JVPs.

We can follow these steps to implement equation (3.17)²:

- (i) Set the input tangent to $\dot{\mathbf{x}} := \mathbf{v}$.
- (ii) Go through the primitives g_1, g_2, \dots, g_{N-1} . At every primitive g_n , compute the next primal value $\mathbf{w}_n := g_n(\mathbf{w}_{n-1})$, as well as its tangent $\dot{\mathbf{w}}_n := \text{JVP}_{g_n}(\mathbf{w}_{n-1}, \dot{\mathbf{w}}_{n-1})$. In this notation we understand $\mathbf{w}_0 = \mathbf{x}$, such that $\mathbf{w}_1 = g_1(\mathbf{w}_0)$.
- (iii) Optionally, one can obtain the primal value of the output $\mathbf{y} = f(\mathbf{x}) = g_N(\mathbf{w}_{N-1})$ at this point with minimal additional effort.
- (iv) Compute the output tangent $\dot{\mathbf{y}} = \text{JVP}_{g_N}(\mathbf{w}_{N-1}, \dot{\mathbf{w}}_{N-1})$. This is the result $\dot{\mathbf{y}} = \text{JVP}_f(\mathbf{x}, \mathbf{v})$.

Since we compute tangents and primals in parallel in step (ii), practical implementations of JVPs often include the computation of primal values. The `python` function `jax.jvp`, for example, returns both the primal value and the tangent of the next intermediate.

Finally, we visualize this process in a computation graph in figure 3.2 (for a rigorous definition of computation graphs, see section 3.3).

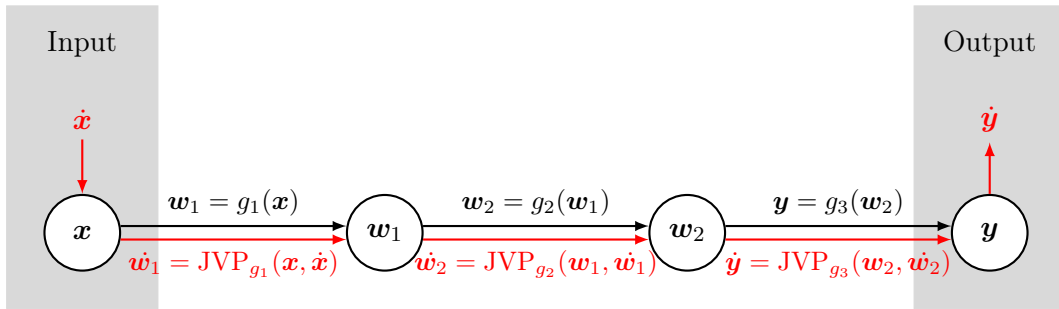


Figure 3.2: Computation graph for forward mode autodiff with the chained-primitive function $f = g_3 \circ g_2 \circ g_1$ (possible implementation in figure 3.1). The *primal pass* (black arrows) computes the output value \mathbf{y} from the input \mathbf{x} using the primitive functions g_n . The *tangent pass* (red arrows) computes the output tangent $\dot{\mathbf{y}}$ from the input tangent $\dot{\mathbf{x}}$ using JVPs of the primitive functions. We can see that at each step of the tangent pass, e.g. from \mathbf{w}_1 to \mathbf{w}_2 , we need the previous primal value \mathbf{w}_1 and the previous tangent $\dot{\mathbf{w}}_1$. Therefore, the tangent and primal pass can efficiently be performed in parallel.

3.2.3 Backward Mode for Chained-Primitive Functions

Similar to what we did with the JVP in equation (3.13), we can also write the VJP of a top-level function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m := g_3 \circ g_2 \circ g_1$ as the chained application of VJPs.

We use the same setup, i.e. we give names to the intermediates $\mathbf{w}_1 := g_1(\mathbf{x})$, $\mathbf{w}_2 := g_2(\mathbf{w}_1)$ and the output $\mathbf{y} := g_3(\mathbf{w}_2)$. Using the chain rule (equation (3.10)) we get

$$\text{VJP}_f(\mathbf{x}, \mathbf{v}^T) = \text{VJP}_{g_1}\left(\mathbf{x}, \text{VJP}_{g_2}(\mathbf{w}_1, \text{VJP}_{g_3}(\mathbf{w}_2, \mathbf{v}^T))\right). \quad (3.18)$$

²More precisely: its analogue for N primitives

Here, we see the first hint at why the resulting autodiff scheme will be called backward mode. The VJPs of the primitives are applied in reverse order (VJP_{*g*₃} first, VJP_{*g*₁} last) compared to the primitives (*g*₁ first, *g*₃ last).

Similar to the tangents corresponding to intermediate values in forward mode, we abbreviate the cotangents (the second arguments) of the VJPs in equation (3.18). From any primal value **a** (either the input **x**, one of the intermediates **w_n** or the output **y**), the output **y** can be computed via some function **y** = *ψ*(**a**). We abbreviate the *cotangent* corresponding to **a** as

$$\bar{\mathbf{a}} := \text{VJP}_{\psi}(\mathbf{a}, \mathbf{v}^T). \quad (3.19)$$

They implicitly depend on **v^T**, which is understood from context since the overall goal of backward mode autodiff is to compute VJP_{*f*}(**x**, **v^T**).

It follows from the chain rule (equation (3.11)) that

$$\overline{\mathbf{w}_{n-1}} = \text{VJP}_{g_n}(\mathbf{w}_{n-1}, \overline{\mathbf{w}_n}). \quad (3.20)$$

The cotangents have an interpretation as derivatives. If **v^T** = **e_i** is a standard basis vector then for any **a** (either the input **x**, one of the intermediates **w_n** or the output **y**) we have

$$\bar{\mathbf{a}} = \frac{\partial y_i}{\partial \mathbf{a}} = \begin{pmatrix} \frac{\partial y_i}{\partial a_1} & \dots & \frac{\partial y_i}{\partial a_k} \end{pmatrix}. \quad (3.21)$$

Let us rewrite equation (3.18) in terms of cotangents of the intermediates:

$$\begin{aligned} \bar{\mathbf{x}} &= \text{VJP}_f(\mathbf{x}, \mathbf{v}^T) = \text{VJP}_{g_1}(\mathbf{x}, \overline{\mathbf{w}_1}), \\ &\text{where } \overline{\mathbf{w}_1} = \text{VJP}_{g_2}(\mathbf{w}_1, \overline{\mathbf{w}_2}), \\ &\text{where } \overline{\mathbf{w}_2} = \text{VJP}_{g_3}(\mathbf{w}_2, \overline{\mathbf{y}}), \\ &\text{where } \overline{\mathbf{y}} = \mathbf{v}^T. \end{aligned} \quad (3.22)$$

Equation (3.22) gives us the backward mode autodiff scheme.

Backward Mode Summarized

We want to compute the vector-jacobian product VJP_{*f*}(**x**, **v^T**) for a chained-primitive function *f* : ℝ^{*n*} → ℝ^{*m*} that is implemented as *f* = *g_N* ∘ ... ∘ *g₁*. We require knowledge about the order and nature of the primitives *g_n*, which we can obtain by analyzing the users implementation of *f*. We need to assume that the autodiff package has implementations for all primitives *g_n* and their VJPs.

We can follow these steps to implement equation (3.22)³:

- (i) Compute all intermediate primals, **w₁**, ..., **w_{N-1}**, via **w₁** = *g*₁(**x**) and **w_n** = *g_n*(**w_{n-1}**). This step (storing all the primals) makes backward mode memory intensive.
- (ii) Optionally, one can obtain the primal value of the output **y** = *f*(**x**) = *g_N*(**w_{N-1}**) at this point, with minimal additional effort.
- (iii) Set the output cotangent to **y** := **v^T**.

³More precisely: its analogue for *N* primitives

- (iv) Go through the primitives g_N, \dots, g_2 in reverse order.
At every primitive g_n , compute the cotangent corresponding to its input, via $\bar{\mathbf{w}}_{n-1} = \text{VJP}_{g_n}(\mathbf{w}_{n-1}, \bar{\mathbf{w}}_n)$.
- (v) Finally, compute the input cotangent $\bar{\mathbf{x}} = \text{VJP}_{g_1}(\mathbf{x}, \bar{\mathbf{w}}_1)$.
This is the result $\bar{\mathbf{x}} = \text{VJP}_f(\mathbf{x}, \mathbf{v}^T)$.

We visualize this process in a computation graph. See figure 3.3.

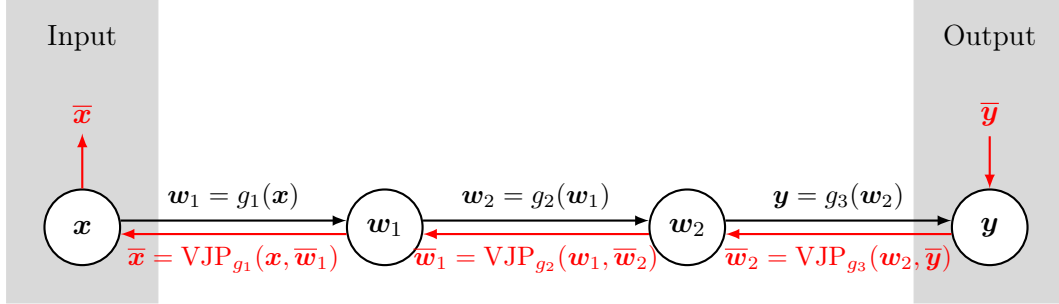


Figure 3.3: Computation graph for backward mode autodiff with the chained primitive function $f = g_1 \circ g_2 \circ g_3$, according to its implementation in figure 3.1. The *primal pass* (black arrows) computes the output value \mathbf{y} from the input \mathbf{x} using the primitive functions g_n . The *cotangent pass* - also called backward pass - (red arrows) computes the input cotangent $\bar{\mathbf{x}}$ from the output cotangent $\bar{\mathbf{y}}$. Note that the order is reversed (compared to the primal pass), giving rise to the name backward mode. Since the cotangent pass starts from the left, and needs all primals to propagate cotangents, the primal pass needs to be performed first, and all primals need to be stored in memory.

3.2.4 An Example: Derivatives from forward and backward mode

We conclude this section with a pedagogical example. Consider the function

$$f : \mathbb{R}^2 \rightarrow \mathbb{R}, \mathbf{x} \mapsto f(\mathbf{x}) := 9x_1^4 + 4x_1^2x_2^2. \quad (3.23)$$

We will express it as the composition of three primitive functions $f = g_3 \circ g_2 \circ g_1$. These primitives are somewhat arbitrary, and would not appear as primitives in an autodiff package⁴. They serve demonstrative purposes. We define them as follows and list the formulae for their JVPs and VJPs:

| | $Dg_n(\mathbf{x})$ | $\text{JVP}_{g_n}(\mathbf{x}, \mathbf{v})$ | $\text{VJP}_{g_n}(\mathbf{x}, \mathbf{v}^T)$ |
|--|---|--|--|
| $g_1(\mathbf{x}) := \begin{pmatrix} x_1^2 \\ x_1x_2 \end{pmatrix}$ | $\begin{pmatrix} 2x_1 & 0 \\ x_2 & x_1 \end{pmatrix}$ | $\begin{pmatrix} 2x_1v_1 \\ x_2v_1 + x_1v_2 \end{pmatrix}$ | $(2x_1v_1 + v_2x_2 \quad v_2x_1)$ |
| $g_2(\mathbf{x}) := \begin{pmatrix} 3x_1 \\ 2x_2 \end{pmatrix}$ | $\begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$ | $\begin{pmatrix} 3v_1 \\ 2v_2 \end{pmatrix}$ | $(3v_1 \quad 2v_2)$ |
| $g_3(\mathbf{x}) := x_1^2 + x_2^2$ | $(2x_1 \quad 2x_2)$ | $2x_1v_1 + 2x_2v_2$ | $(2x_1v \quad 2x_2v)$ |

Note that in the last line, the shapes change, since $g_3 : \mathbb{R}^2 \rightarrow \mathbb{R}$ is scalar valued. This means that the output tangent \dot{y} , as well as the output cotangent \bar{y} are scalars.

⁴except for possibly g_3 , which is the norm of a vector.

Let us do the calculations by hand, evaluating at $\mathbf{x}_0 = (1 \ 2)^\top$. For the forward mode example we choose to compute $\text{JVP}_f(\mathbf{x}_0, \mathbf{e}_1) = \frac{\partial f}{\partial x_1}(\mathbf{x}_0)$ and for the backward mode example we choose $\text{VJP}_f(\mathbf{x}_0, 1) = \frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}_0)$. This choice determines the initially set input tangent $\dot{\mathbf{x}}$ in the forward pass and output cotangent \bar{y} in the backward pass.

| Primal goal: $y = f(\mathbf{x}_0)$ | Forward goal: $\dot{y} = \frac{\partial f}{\partial x_1}(\mathbf{x}_0)$ | Backward goal: $\bar{\mathbf{x}} = \frac{\partial f_1}{\partial \mathbf{x}}(\mathbf{x}_0)$ |
|---|--|---|
| $\mathbf{x} := \mathbf{x}_0 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ | $\dot{\mathbf{x}} := \mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ | $\bar{\mathbf{x}} = \text{VJP}_{g_1}(\mathbf{x}, \bar{\mathbf{w}}_1) = (68 \ 16)$ |
| $\mathbf{w}_1 = g_1(\mathbf{x}) = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ | $\dot{\mathbf{w}}_1 = \text{JVP}_{g_1}(\mathbf{x}, \dot{\mathbf{x}}) = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$ | $\bar{\mathbf{w}}_1 = \text{VJP}_{g_2}(\mathbf{w}_1, \bar{\mathbf{w}}_2) = (18 \ 16)$ |
| $\mathbf{w}_2 = g_2(\mathbf{w}_1) = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ | $\dot{\mathbf{w}}_2 = \text{JVP}_{g_2}(\mathbf{w}_1, \dot{\mathbf{w}}_1) = \begin{pmatrix} 6 \\ 4 \end{pmatrix}$ | $\bar{\mathbf{w}}_2 = \text{VJP}_{g_3}(\mathbf{w}_2, \bar{y}) = (6 \ 8)$ |
| $y = g_3(\mathbf{w}_2) = 25$ | $\dot{y} = \text{JVP}_{g_3}(\mathbf{w}_2, \dot{\mathbf{w}}_2) = 68$ | $\bar{y} := e_1 = 1$ |

See the computation graphs (figures 3.2 and 3.3) for the order of computation. The table-like presentation of the computation is inspired by [47].

We can check that these results are indeed correct by comparing to the derivatives of f , which we can just write down (symbolic differentiation by hand):

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} 36x_1^3 + 8x_1x_2^2 \\ 8x_1^2x_2 \end{pmatrix} \quad ; \quad \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}_0) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}_0) \end{pmatrix} = \begin{pmatrix} 68 \\ 16 \end{pmatrix}. \quad (3.25)$$

3.3 Automatic Differentiation for General Computation Graphs

In section 3.2, we assumed that the computational graph is sequential (see figure 3.2). That means that the primitive functions each have one input and one output and the computation consists only of sequentially applying the primitives.

In practice, computations are more complicated than that. Implementations of interesting functions usually have a more complicated structure than the implementation in figure 3.1. Primitives can have multiple inputs and/or multiple outputs. Even very elementary primitives, such as addition are most naturally expressed as having two inputs. Matrix decompositions, such as the singular value decomposition (SVD), have multiple outputs. And these multiple inputs can be the result of *different* previously executed primitives (analogously, multiple outputs can be used in *different* successively executed primitives).

This means that (in general) the computation graphs are not sequential (as in figure 3.2), but branch, (as in figure 3.5). In this section, we illustrate how to deal with general computation graphs. We require that the function of interest can be computed using only primitives (which are differentiable⁵). These requirements guarantee that the JVP (3.2) and/or VJP (3.2) exist. We first give an intuitive explanation, then state the results, which we show to be correct with more rigor in appendix C.1.

3.3.1 Computation Graphs

Let us first formally introduce the concept of computation graphs. We have already used them in figures 3.2 and 3.3, but we have not fully explained their structure.

The central idea behind the computation graph is the *Wengert list* [47, 80]. It comes from a theoretical version of an implementation, that never overwrites any values in memory. This means that the result of any intermediate step is given a name. The implementation tells us how to obtain each intermediate result from the previous ones.

So we identify the intermediate values, as well as the primitives that are used to compute intermediates and outputs from the inputs. We organize this information in a directed graph, the *computation graph*. The computation graph has a node for every input, every intermediate value and every output and we label them by their primal values. The edges of the graph correspond to the primitive functions. A primitive function is represented by one or multiple edges. The edges corresponding to a primitive connect every one of its inputs to every one of its outputs.

Let us consider an abstract example. An implementation of a function

$$f : U_w \times U_x \rightarrow V_y \times V_z \quad (3.26)$$

with (real) coordinate spaces U_w, U_x, V_y, V_z (i.e. $U_w = \mathbb{R}^n$ for some n etc.) might look like figure 3.4.

Let us build the computation graph for this implementation. We have inputs w and x , intermediate values a, b, c, d, e and output values y and z , which form the nodes in the graph.

⁵This implies that the function itself must be differentiable

```

from favorite_autodiff_package import primitive_g, primitive_h, \
    primitive_i, primitive_j, primitive_k, primitive_l

def f(w, x):
    a, b = primitive_g(w, x)
    c = primitive_h(x)
    d = primitive_i(a, b, c)
    e = primitive_j(b, c)
    y = primitive_k(d)
    z = primitive_l(d, e)
    return y, z

```

Figure 3.4: Implementation of a generic function (equation (3.26)). In the spirit of Wengert lists, we have gone out of our way to give different names to every intermediate result. We have given generic names to the functions that are called. We assume that all of them are primitives of the autodiff package. From the mathematical viewpoint, the primitives map between (cartesian products of) real coordinate spaces, e.g. $g : U_w \times U_x \rightarrow U_a \times U_b$.

We also have primitives g, h, i, j, k, l which form the edges of the graphs. The primitives h and k are represented by one edge each, since they each have a single input and a single output, but all other primitives correspond to multiple edges.

The computation graph of f is shown in figure 3.5.

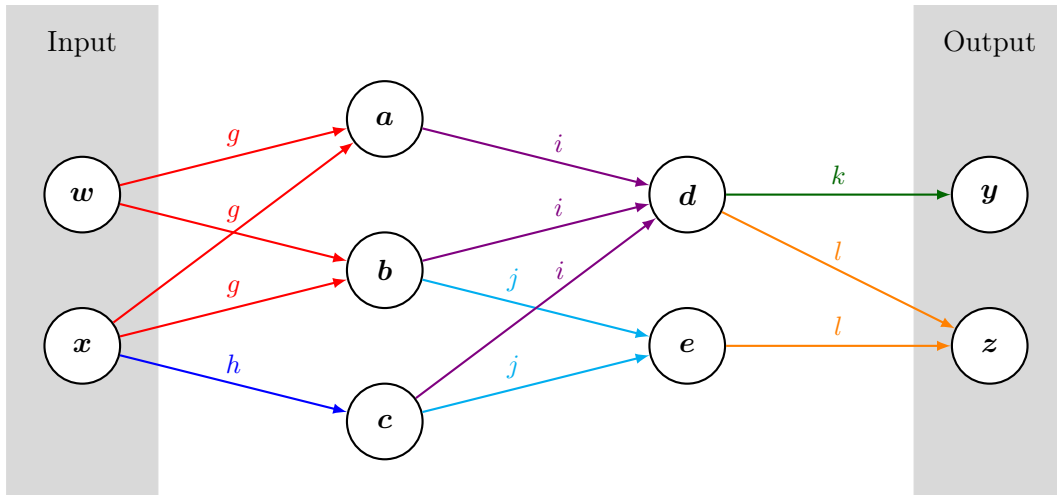


Figure 3.5: Computation graph of a generic function, see figure 3.4. Colors are used to distinguish the primitives visually.

A common strategy to store the necessary information about the computation graph is to wrap the numeric types of the programming language in a `Node` object [71]. Each node object, for example b in figure 3.5, stores the “local” information about the computation graph:

- Its parent nodes. These are the nodes that are used to compute the value of the current node.

The parent nodes of b are w and x .

- Its child nodes. These are the nodes that use the value of the current node to compute their value.

The child nodes of b are d and e .

- Its primitive function. This is the primitive that is used to compute the value of the node. The full book-keeping also includes in which order the parent nodes are used as its inputs and which of its outputs corresponds to the current node.

The primitive function of the node b is g and b is its second output.

This information can be gathered by analyzing the user’s implementation of f (in this example that is figure 3.4). If a primitive is called with `Node` objects as arguments, it first computes the primal value. It then modifies the `Node` objects involved in the computation, in order to store the information as enumerated above. Finally, a new `Node` object is returned.

The package `jax`, which we use for the numerics of this work, follows a different approach. It calls the user’s implementation with *abstract arrays* as arguments, obtaining an abstract expression (a “Jaxpression”) of the program which can then be transformed to a Jaxpression of the JVP or VJP.

3.3.2 Forward mode on General Computation Graphs

Let us now gain an intuitive understanding of how autodiff in forward mode works, if the computation graph of the top-level function is no longer sequential. For a more rigorous but less intuitive approach, see appendix C.1. Here, we focus on the more intuitive viewpoint of the JVP as a tool for computing derivatives w.r.t. one of the input components and understand tangents as derivatives w.r.t. that component. This is not necessary to make the arguments.

We want to compute derivatives of all outputs w.r.t one component of one of the inputs, say x_j . We associate a tangent with every (input, intermediate and output) primal value. We can trivially set the tangents for the inputs. They are zero everywhere, except for the single component x_j , where the value is one.

We then devise a scheme to propagate the tangents through the computation graph. At the end of it, we will have obtained tangents for all the outputs. These are the derivatives of the outputs w.r.t x_j , or equivalently the $\text{JVP}_f(\mathbf{x}, \mathbf{e}_j)$.

Since we propagate the tangents forward iteratively, we can always assume that the tangents at the parent nodes are known. The task is then to compute the tangent of the current node.

Let us consider a generic node in the graph (figure 3.6). For concreteness, we consider a node with three parents and two children, but it is easy to see how this generalizes to any number of parents / children.

Let us assume that $\mathbf{a} \in \mathbb{R}^{n_a}$, $\mathbf{b} \in \mathbb{R}^{n_b}$, $\mathbf{c} \in \mathbb{R}^{n_c}$ and $\mathbf{d} \in \mathbb{R}^{n_d}$. So the primal used to compute \mathbf{d} from $\mathbf{a}, \mathbf{b}, \mathbf{c}$ is of the form $i : \mathbb{R}^{n_a} \times \mathbb{R}^{n_b} \times \mathbb{R}^{n_c} \rightarrow \mathbb{R}^{n_d} \times V$, $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \mapsto (\mathbf{d}, \mathbf{o})$, where we have summarized all possible other outputs of i as \mathbf{o} in some space V .

From the structure of the computation graph, we see that the primal \mathbf{d} can depend on t only through \mathbf{a} , \mathbf{b} and \mathbf{c} . We can express the tangent $\dot{\mathbf{d}}$ via the chain rule. Let us consider its i^{th} component

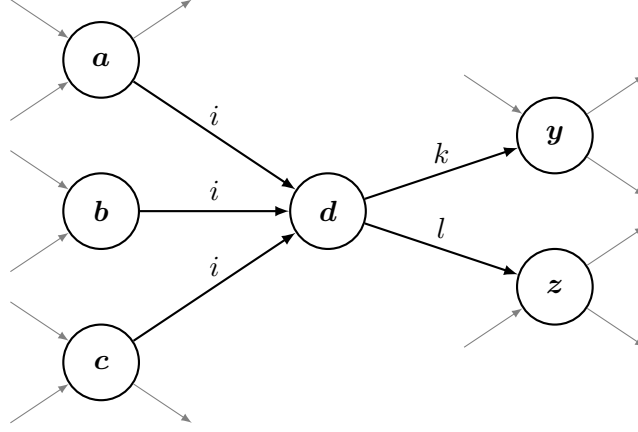


Figure 3.6: A generic node d in a computation graph. The parent nodes of d are a , b and c . The primal d is computed from the parent primals using the primitive function i . The child nodes of d are y and z . The primitive i can have (in general) other outputs (other than d) and the primitives k and l can have other inputs.

$$\begin{aligned}
 (\dot{\mathbf{d}})_i &= \frac{\partial d_i}{\partial x_j} = \sum_{k=1}^{n_a} \frac{\partial d_i}{\partial a_k} \frac{\partial a_k}{\partial x_j} + \sum_{k=1}^{n_b} \frac{\partial d_i}{\partial b_k} \frac{\partial b_k}{\partial x_j} + \sum_{k=1}^{n_c} \frac{\partial d_i}{\partial c_k} \frac{\partial c_k}{\partial x_j} \\
 &= \left[(D_a d \quad D_b d \quad D_c d) \cdot \begin{pmatrix} \dot{\mathbf{a}} \\ \dot{\mathbf{b}} \\ \dot{\mathbf{c}} \end{pmatrix} \right]_i.
 \end{aligned} \tag{3.27}$$

We recognize this as a JVP. In particular, we recognize that the JVP defined in equation (3.2) does exactly this; It propagates the tangents. So for this primitive $i : (\mathbf{a}, \mathbf{b}, \mathbf{c}) \mapsto (\mathbf{d}, o)$, we have the relation

$$\text{JVP}_i((\mathbf{a}, \mathbf{b}, \mathbf{c}), (\dot{\mathbf{a}}, \dot{\mathbf{b}}, \dot{\mathbf{c}})) = (\dot{\mathbf{d}}, \dot{o}), \tag{3.28}$$

which is the direct generalization of equation (3.17).

Equation (3.28) and its analogue at the other nodes enable us to propagate the tangents all the way through the computation graph and thus gives us the forward mode autodiff scheme.

We summarize the tangent propagation at a single node in figure 3.7 and the entire forward mode algorithm in section 3.3.4.

3.3.3 Backward mode on General Computation Graphs

For backward mode autodiff on general computation graphs, there is a subtlety that does not appear in the forward mode. In this section, we aim to show intuitively, how it arises. We show that the resulting scheme is correct with more rigor in appendix C.1. For an intuitive perspective, we take the viewpoint of computing derivatives (see equation (3.9)).

We are interested in derivatives of one component of one of the outputs of the example function f , say f_j . See figure 3.4 for the implementation of f and figure 3.5 for its computation graph. To make notation a bit easier, let us assume for now that all primals are

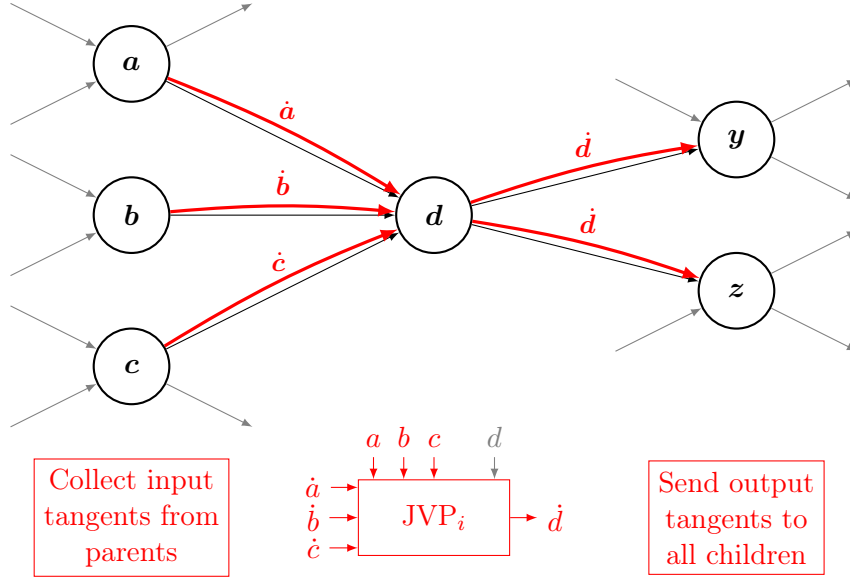


Figure 3.7: Forward-Propagation of tangents through a node with primal d . The JVP used to compute \dot{d} is the JVP of the primal function (here i) that is used to compute d in the primal pass, see equation (3.28). In practice, the implementation of JVP_i can also use the output primal $d = i(a, b, c)$, if that is more efficient.

real scalars. We also assume that we know the cotangent $\bar{d} = \frac{\partial f_j}{\partial d}$ of the central node d of figure 3.6.

If we want to compute the cotangents of the parent nodes of d , for example $\bar{a} = \frac{\partial f_j}{\partial a}$, the first intuition is that we can just use the VJP of the primal i , since we could do the analogue with JVPs in forward mode. The VJP gives us

$$\text{VJP}_i((a, b, c), \bar{d}) = \begin{pmatrix} \frac{\partial f_j}{\partial d} \frac{\partial d}{\partial a} & \frac{\partial f_j}{\partial d} \frac{\partial d}{\partial b} & \frac{\partial f_j}{\partial d} \frac{\partial d}{\partial c} \end{pmatrix}. \quad (3.29)$$

The first entry (corresponding to the first input a) of $\text{VJP}_i((a, b, c), \bar{d})$ is

$$\frac{\partial f_j}{\partial d} \frac{\partial d}{\partial a}, \quad (3.30)$$

which almost looks like the chain-rule for $\frac{\partial f_j}{\partial a}$. But there is one caveat: the output component f_j could depend on a through other intermediates (other than d). These other “paths” are omitted in figure 3.6, and hinted at with the gray edges.

Such other paths exist if more primitives (other than i) use a as an input. So for every primitive g that has a as an input, there is another contribution, just like (3.30).

We then get the cotangent of a as

$$\bar{a} = \frac{\partial f_j}{\partial a} = \sum_{g \text{ with } g(a) = x} \frac{\partial f_j}{\partial x} \frac{\partial x}{\partial a} =: \sum_{\{g \mid a \text{ input of } g\}} \bar{a}^{(g)}. \quad (3.31)$$

Here, we have abbreviated the contribution to the cotangent \bar{a} of a from the primitive g (that uses a as an input) as $\bar{a}^{(g)}$. We recognize that this is the output of the VJP in equation (3.29),

So the output of the VJP is actually

$$\text{VJP}_i((a, b, c), \bar{d}) = (\bar{a}^{(i)} \quad \bar{b}^{(i)} \quad \bar{c}^{(i)}) . \quad (3.32)$$

and to obtain \bar{a} , we need to sum all such contributions (equation (3.31)).

Intuitively, the VJP is the *local* cotangent propagation. It does not know about the others paths, which are a global property of the graph.

We illustrate the process of cotangent propagation at a single node in figure 3.8.

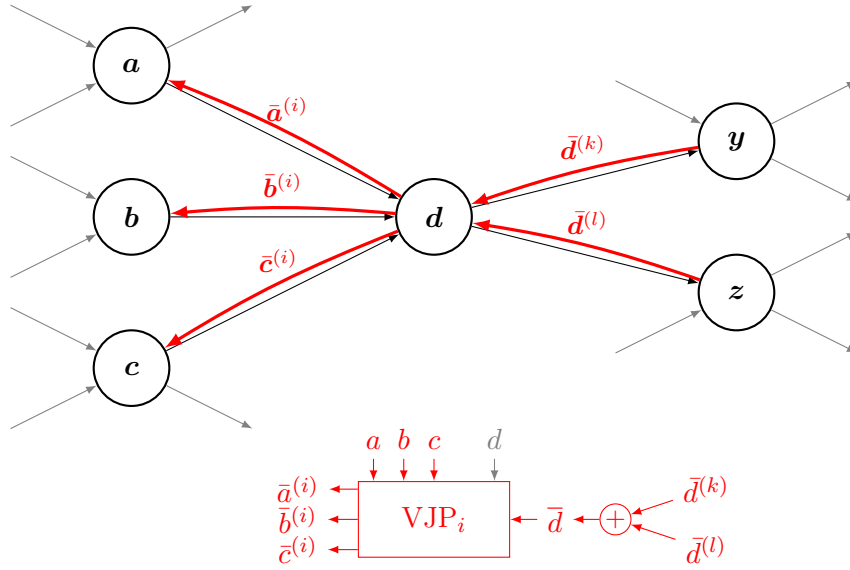


Figure 3.8: Backward-Propagation of cotangents at a generic node d in a computation graph (see figures 3.6 and 3.5). The cotangent \bar{d} is the sum of contributions from all primitives that use d as an input (see equation (3.31)). These are collected from all the child nodes of d . The contribution to the cotangents of the parent nodes a , b and c of d is given by the VJP, see equation (3.32). In practice, the implementation of VJP_i can also use the output primal $d = i(a, b, c)$, if that is more efficient.

3.3.4 Summary: How-To Forward Mode

We summarize the procedure for forward mode autodiff on general computation graphs. This differs only slightly from the corresponding list in section 3.2.

We want to compute the JVP of a generic function

$$f : \prod_{n=1}^N X_n \rightarrow \prod_{m=1}^M Y_m, (x_1, x_2, \dots, x_N) \mapsto f(x_1, x_2, \dots, x_N) := (y_1, y_2, \dots, y_M), \quad (3.33)$$

with N inputs x_n that live in real coordinate spaces X_n , and M outputs y_m living in real coordinate spaces Y_m .

The goal is to compute $\text{JVP}_f(\mathbf{x}, \mathbf{v})$ for a given set of inputs $\mathbf{x} = (x_1, x_2, \dots, x_N) \in \prod_{n=1}^N X_n$ and input tangents $\mathbf{v} = (v_1, v_2, \dots, v_N) \in \prod_{n=1}^N X_n$.

In most practical cases, the spaces X_n are collections of real numbers in a specific shape, e.g. the space of tensors of a given shape. Then, if \mathbf{v} is only non-zero at one component of one of these tensors (and equals one there), the $\text{JVP}_f(\mathbf{x}, \mathbf{v})$ is the derivative of f at \mathbf{x} w.r.t that one component of its inputs.

We assume that we have analyzed the implementation of f and have obtained its computation graph. That is, we have identified all intermediate primal values $\mathbf{w}_i \in W_i$, which again live in some real coordinate spaces W_i , as well as the primals g_j which map from intermediates (and/or inputs) to other intermediates (and/or outputs). From the implementation of f , we also have an order for the primals g_1, \dots, g_J , in which they are applied when $f(\mathbf{x})$ is computed. In practice, analyzing the computation graph can be done on-the-fly.

The forward mode autodiff scheme to compute $\text{JVP}_f(\mathbf{x}, \mathbf{v})$ is:

- (i) Set the input tangents to $(\dot{x}_1, \dots, \dot{x}_N) := \dot{\mathbf{x}} := \mathbf{v}$.
- (ii) Go through the primitives g_1, \dots, g_J . At every primitive, compute the primal values $\mathbf{w}_{j,\text{out}} = g_j(\mathbf{w}_{j,\text{in}})$ for its outputs. Then, compute the output tangents using the JVP of g_j as $\dot{\mathbf{w}}_{j,\text{out}} = \text{JVP}_{g_j}(\mathbf{w}_{j,\text{in}}, \dot{\mathbf{w}}_{j,\text{in}})$. See figure 3.7.
Here, we write $\mathbf{w}_{j,\text{in}}$ for all intermediate values, that are inputs of g_j . They live in the cartesian product of *some* of the X_n and/or W_i . Similarly, $\mathbf{w}_{j,\text{out}}$ are all of its outputs, which live in the product of *some* of the W_i and/or Y_m .
- (iii) We have computed the primal values for the outputs at this point. Optionally, they can be returned as well.
- (iv) After treating the last primitive in step (ii), we have computed all output tangents. These are the result that we return: $\text{JVP}_f(\mathbf{x}, \mathbf{v}) = \dot{\mathbf{y}} = (\dot{y}_1, \dots, \dot{y}_M)$.

We illustrate this process in figure 3.9.

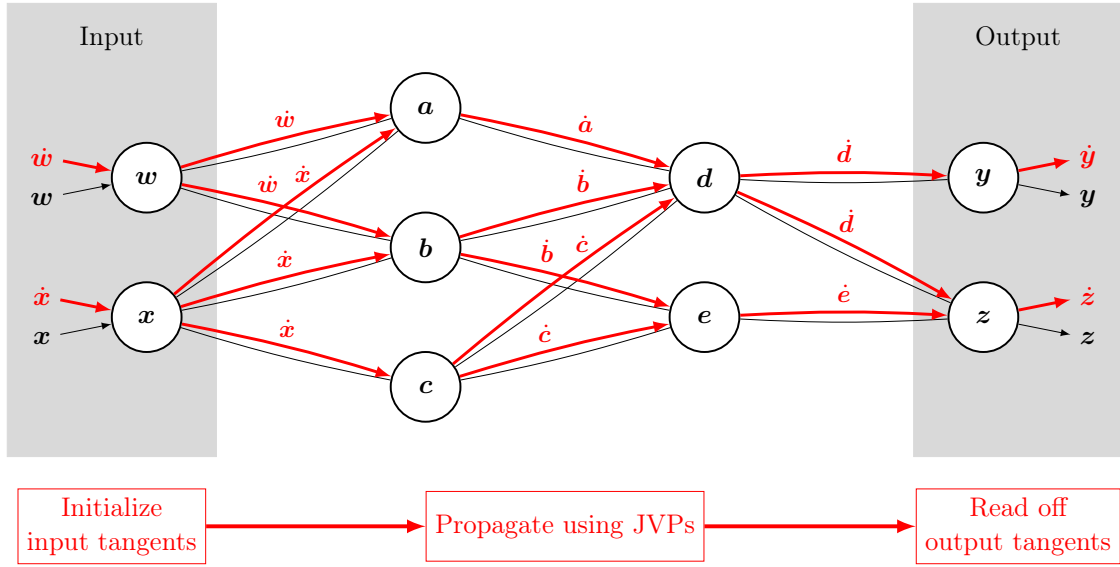


Figure 3.9: Illustration of the forward mode autodiff scheme for a generic computation graph. See figure 3.4 for an implementation of such a function and figure 3.7 for a detailed view on the propagation process at a single node.

3.3.5 Summary: How-To Backward Mode

We use the same setup as above. That is we have a function $f : \prod_{n=1}^N X_n \rightarrow \prod_{m=1}^M Y_m$ with inputs $\mathbf{x}_1, \dots, \mathbf{x}_N$ and outputs $\mathbf{y}_1, \dots, \mathbf{y}_M$. Its implementation generates intermediate values $\mathbf{w}_1, \dots, \mathbf{w}_I$ and uses primitives g_1, \dots, g_J .

In backward mode, we are interested in computing $\text{VJP}_f(\mathbf{x}, \mathbf{v})$ for some inputs $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_N) \in \prod_{n=1}^N X_n$ and cotangents $\mathbf{v}^T = (\mathbf{v}_1^T \dots \mathbf{v}_M^T) \in \prod_{m=1}^M Y_m$.

- (i) Compute all intermediates \mathbf{w}_i , with the user-supplied implementation of f . This also lets the algorithm obtain the computation graph. In particular, we obtain the order g_1, \dots, g_J , in which the primitives are applied. All intermediate primals need to be stored in memory. This makes backward mode RAM-intensive.
- (ii) Optionally, the output primals \mathbf{y}_m can be computed at this time, with minimal additional effort.
- (iii) Initialize the output cotangent(s): $(\bar{\mathbf{y}}_1 \dots \bar{\mathbf{y}}_M) = \bar{\mathbf{y}} := \mathbf{v}^T = (\mathbf{v}_1^T \dots \mathbf{v}_M^T)$.
- (iv) Go through the primitives g_J, \dots, g_1 in reverse order.
 - (a) At the primitive g_j , let us write $(\mathbf{o}_1, \dots, \mathbf{o}_{n_j})$ for the outputs of g_j and $(\mathbf{i}_1, \dots, \mathbf{i}_{m_j})$ for its inputs. These are some (j -dependent) subsets of the inputs, intermediates and outputs.
 - (b) Compute the cotangents $(\bar{\mathbf{o}}_1 \dots \bar{\mathbf{o}}_{n_j})$ of the outputs of g_j .
If \mathbf{o}_k is an output of the top-level function f , its cotangent is already set. If not, sum up the contributions $\bar{\mathbf{o}}_k = \sum_{\{g \mid \mathbf{o}_k \text{ input of } g\}} \bar{\mathbf{o}}_k^{(g)}$ from all primitives g that use \mathbf{o}_k as an input.

(c) Compute the contributions $\bar{\mathbf{i}}_l^{(g_j)}$ to all cotangents $\bar{\mathbf{i}}_l$ of the inputs using the VJP:

$$(\bar{\mathbf{i}}_1^{(g_j)} \dots \bar{\mathbf{i}}_{m_j}^{(g_j)}) = \text{VJP}_{g_j}((\mathbf{i}_1, \dots, \mathbf{i}_{m_j}), (\bar{\mathbf{o}}_1, \dots, \bar{\mathbf{o}}_{n_j}))$$

(v) For every input \mathbf{x}_n , sum up the contributions $\bar{\mathbf{x}}_n = \sum_{\{g \mid \mathbf{x}_n \text{ input of } g\}} \bar{\mathbf{x}}_n^{(g)}$ to obtain its cotangent.

(vi) Read off the result from the input tangents: $\text{VJP}_f(\mathbf{x}, \mathbf{v}) = \bar{\mathbf{x}} = (\bar{\mathbf{x}}_1 \dots \bar{\mathbf{x}}_N)$.

We illustrate this process in figure 3.10.

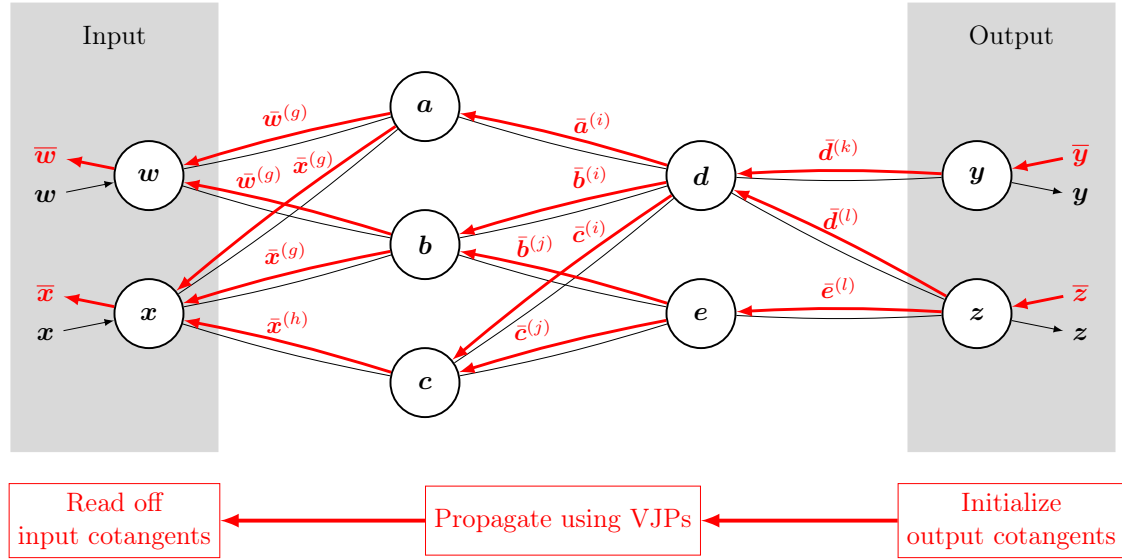


Figure 3.10: Illustration of the backward mode autodiff scheme for a generic computation graph (see main text). See figure 3.5 for an implementation of a function with this computation graph and figure 3.8 for a detailed view on the propagation of cotangents at a single node of the graph.

3.4 Automatic Differentiation for Complex Functions

In the previous sections, we have assumed that all functions (the top-level function, as well as the primitives) map between real vector spaces. It is possible to extend autodiff to functions that map between complex spaces.

We understand data structures that are a collection of complex numbers, like a list of (complex) matrices as column-vectors in \mathbb{C}^n (for some appropriate $n \in \mathbb{N}$), by re-arranging all entries into a column-vector.

In this section, we heavily use the *Wirtinger calculus* [70] (in particular, holomorphic and anti-holomorphic derivatives). We discuss Wirtinger calculus in detail in appendix B. Let us briefly recap the notation for scalar functions $f : \mathbb{C} \rightarrow \mathbb{C}$.

We write $z = x + iy$ and $f(z) = u(x, y) + iv(x, y)$ with $x, y \in \mathbb{R}$ and $u, v : \mathbb{R}^2 \rightarrow \mathbb{R}$. Then the holomorphic and anti-holomorphic derivatives are respectively

$$\frac{\partial f}{\partial z}(z) = \frac{1}{2} \left(\frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) (u(x, y) + iv(x, y)); \quad (3.34a)$$

$$\frac{\partial f}{\partial z^*}(z) = \frac{1}{2} \left(\frac{\partial}{\partial x} + i \frac{\partial}{\partial y} \right) (u(x, y) + iv(x, y)). \quad (3.34b)$$

For a function $F : \mathbb{C}^n \rightarrow \mathbb{C}^m$, there are two independent derivative objects, the “Jacobian”-matrix of all holomorphic derivatives and the analogous anti-holomorphic “Jacobian”:

$$DF : \mathbb{C}^n \rightarrow \mathbb{C}^{m \times n}, z \mapsto DF(z) := \left(\frac{\partial F_i}{\partial z_j} \right)_{ij}; \quad (3.35a)$$

$$D^*F : \mathbb{C}^n \rightarrow \mathbb{C}^{m \times n}, z \mapsto D^*F(z) := \left(\frac{\partial F_i}{\partial z_j^*} \right)_{ij}. \quad (3.35b)$$

Unlike for real functions $F_r : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where all derivative information can be expressed as a matrix in $\mathbb{R}^{m \times n}$ (the Jacobian), there is (in general) no single object in $\mathbb{C}^{m \times n}$ that captures all derivative information of the complex-to-complex function F . We are therefore unable to construct an autodiff scheme that works in *exactly* the same way as in the real case and gives us all derivative information of F . We therefore relax the requirement that our autodiff scheme should compute both the holomorphic and anti-holomorphic derivatives.

3.4.1 Complex JVPs and VJP

We define the JVP and VJP for complex functions “out of the blue”. We then show that autodiff in forward (backward) mode can be used to compute the JVP (VJP) of top-level functions from the JVPs (VJPs) of primitives. Finally, we show that useful derivative information can be obtained from the JVP or VJP of top-level functions.

We extend the definitions of the real JVP (equation (3.2)) and VJP (equation (3.6)) to the complex case in a compatible way. That means that if the complex JVP (VJP) is used on a real-to-real function and fed real primals and real tangents (cotangents), it should

behave just like the real JVP (VJP). This will also enable autodiff packages to have one implementation that covers both cases.

We define the JVP of a complex function $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$. For clarity, we express everything once in terms of f and once in terms of its real-to-real counterpart

$$g : \mathbb{R}^{2n} \rightarrow \mathbb{R}^{2m} : \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \mapsto \begin{pmatrix} \operatorname{Re}[f(\mathbf{x} + i\mathbf{y})] \\ \operatorname{Im}[f(\mathbf{x} + i\mathbf{y})] \end{pmatrix} =: \begin{pmatrix} u(\mathbf{x}, \mathbf{y}) \\ v(\mathbf{x}, \mathbf{y}) \end{pmatrix}, \quad (3.36)$$

which is defined such that $f(\mathbf{x} + i\mathbf{y}) = u(\mathbf{x}, \mathbf{y}) + iv(\mathbf{x}, \mathbf{y})$.

We write $D_x u := \left(\frac{\partial u_i}{\partial x_j} \right)_{ij}$ for the “Jacobian”-matrix of all partial derivatives of u w.r.t to \mathbf{x} , and analogously for $D_y u$, $D_x v$ and $D_y v$. An unmarked D stands for holomorphic Jacobians (which are not the same as real Jacobians!).

We define the JVP of a differentiable function $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ to be

$$\begin{aligned} \text{JVP}_f : \mathbb{C}^n \times \mathbb{C}^n &\rightarrow \mathbb{C}^m, (\mathbf{z}, \mathbf{v}) \mapsto Df(\mathbf{z}) \cdot \mathbf{v} + D^* f(\mathbf{z}) \cdot \mathbf{v}^* \\ &= \begin{pmatrix} \mathbb{1} & i\mathbb{1} \end{pmatrix} \cdot \begin{pmatrix} D_x u(\mathbf{x}, \mathbf{y}) & D_y u(\mathbf{x}, \mathbf{y}) \\ D_x v(\mathbf{x}, \mathbf{y}) & D_y v(\mathbf{x}, \mathbf{y}) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \end{pmatrix}, \end{aligned} \quad (3.37)$$

where the primal argument $\mathbf{z} = \mathbf{x} + i\mathbf{y}$ and the tangent $\mathbf{v} = \mathbf{c} + i\mathbf{d}$ are expressed via their real and imaginary parts in the second line.

We can see that this is compatible with the real case of equation (3.2), since then all imaginary parts vanish $\mathbf{y} = \mathbf{d} = v = 0$ and u is only a function of \mathbf{x} (and thus $D_y u = 0$). Equation (3.37) therefore reduces to $(\mathbf{x}, \mathbf{c}) \mapsto D_x u(\mathbf{x}, 0) \cdot \mathbf{c}$, which is the real JVP as defined in equation (3.2).

We define the complex VJP of a differentiable function $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ to be

$$\begin{aligned} \text{VJP}_f : \mathbb{C}^n \times \mathbb{C}^m &\rightarrow \mathbb{C}^n, (\mathbf{z}, \mathbf{w}^T) \mapsto \mathbf{w}^T \cdot Df(\mathbf{z}) + \mathbf{w}^\dagger \cdot Df^*(\mathbf{z}) \\ &= \begin{pmatrix} \mathbf{c}^T & -\mathbf{d}^T \end{pmatrix} \cdot \begin{pmatrix} D_x u(\mathbf{x}, \mathbf{y}) & D_y u(\mathbf{x}, \mathbf{y}) \\ D_x v(\mathbf{x}, \mathbf{y}) & D_y v(\mathbf{x}, \mathbf{y}) \end{pmatrix} \cdot \begin{pmatrix} \mathbb{1} \\ -i\mathbb{1} \end{pmatrix}, \end{aligned} \quad (3.38)$$

where the primal argument $\mathbf{z} = \mathbf{x} + i\mathbf{y}$ and the cotangent $\mathbf{w}^T = \mathbf{c}^T + i\mathbf{d}^T$ are expressed via their real and imaginary parts in the second line.

In a more rigorous treatment, the cotangents (the second argument and the output of the VJP) would live in the dual space $(\mathbb{C}^n)^*$. In equation (3.38), we understand the column-vector \mathbf{w}^T to represent the covector $(\mathbf{a} \mapsto \mathbf{w}^T \cdot \mathbf{a}) \in (\mathbb{C}^n)^*$. In practice, we suggest to operate on the representation $\mathbf{w} = \mathbf{c} + i\mathbf{d} \in \mathbb{C}^n$ in memory. To make this more apparent, we choose to sacrifice some rigor and define the VJP with cotangents in \mathbb{C}^m .

Since we are dealing with a complex vector space, it is not obvious if the map from vectors to covectors should involve complex conjugation, which can cause errors. We suggest to be very pragmatic about this. If vectors and covectors are ever mixed up, the result is conjugation of the cotangents in the wrong places. An easy way to check that things are

working as intended, is to consider the VJP of the identity map and of the conjugation map:

$$\text{VJP}_{(z \mapsto z)}(z, v^T) = v^T \quad ; \quad \text{VJP}_{(z \mapsto z^*)}(z, v^T) = v^\dagger. \quad (3.39)$$

Notice that the VJP of the identity map acts trivially on cotangents, while the VJP of complex conjugation also acts as complex conjugation on cotangents. A mix-up between vectors and covectors would result in a mix-up of these two results.

Again, we can check that the definition (3.38) reduces to the real VJP defined in equation (3.6). In the real case, all imaginary parts vanish $y = d = v = 0$ and u is only a function of x (and thus $D_y u = 0$). All that remains is $c^T \cdot D_x u(x, 0)$, which is the correct real VJP.

Our definitions for the complex JVP and VJP agree with the convention of `jax` [59]. They are not the only choice. For example, the “dual function” of Guo et. al. [48, eq.15] differs from equation (3.38) by two complex conjugations, in particular their dual function can be obtained from the VJP as $\tilde{g}(\nu^*)_z = \text{VJP}_g(z, \nu)^*$.

We would like to comment that, unlike the real JVP and VJP, the complex versions are (in general) not linear in their second argument. Also unlike in the real case, the VJP is not the transpose of the JVP.

3.4.2 Forward Mode

So far, we have just defined a JVP in equation (3.37). In the appendix C.2, we show that it fulfills the necessary chain-rule to be used in the autodiff scheme of section 3.3. There, we also show the results, which we only state here.

If we compute $\text{JVP}_f(z, e_t)$ for some canonical basis vector e_t , we expect the result to be a derivative of f w.r.t the t -component of its input at the point z . This is true in the following sense:

- For real-to-real functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ we get the standard real derivative

$$\text{JVP}_f(x, e_t) = \frac{\partial f}{\partial x_t}(x).$$

We have already convinced ourselves that the complex JVP reduces to the real case, if everything (primals, tangents and primitives) is real.

If a real-to-real function internally uses complex intermediate values, (e.g. a fourier transformation), the complex JVP formula will be used. The behavior of autodiff is the same, as if the function was implemented using only real intermediates.

- For functions $f : \mathbb{R}^n \rightarrow \mathbb{C}^m$ in real-variables, we get the derivative

$$\text{JVP}_f(x, e_t) = \frac{\partial f}{\partial x_t}(x) = \frac{\partial u}{\partial x_t}(x) + i \frac{\partial v}{\partial x_t}(x).$$

- For holomorphic functions $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ we get the *holomorphic* derivative

$$\text{JVP}_f(z, e_t) = \frac{\partial f}{\partial z_t}(z).$$

This is all we will ever need from this function, since it is holomorphic and therefore $\frac{\partial f}{\partial z_i^*}(z) = 0$.

Note that we do not need to require the primitives to be holomorphic.

In the other cases (complex-to-real and non-holomorphic complex-to-complex), we can still extract useful information from the JVP of f , but the results can be counter-intuitive. We discuss them in appendix C.2.

In these cases, the autodiff package should ensure that the user does not get unexpected results (i.e. knows what they are doing). In `jax`, for example, to use forward mode on a function of complex arguments, the user needs to set the `holomorphic=True` flag. This does not change the behavior of the algorithm, it only disables the error messages, that would otherwise occur with complex inputs. If the function actually is holomorphic, the user gets expected results (as above). If it is not, `jax` still computes the JVP, but the meaning of the result might not be obvious to the user, unless they are familiar with the background.

3.4.3 Backward Mode

Like for the forward mode, we show that the VJP fulfills the necessary chain-rule to be used in the autodiff scheme of section 3.3, in appendix C.2. Here, we only state the results, in particular what derivative information we can obtain from the VJP of a top-level function f .

If we compute $\text{VJP}_f(z, e_j^T)$ for some canonical basis vector e_j , we expect something like the derivatives of the j^{th} component of f w.r.t. all input components at the point z . This is true in the following sense:

- For real-to-real functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ we get the standard real derivative

$$\text{VJP}_f(x, e_j^T) = \frac{\partial f_j}{\partial x}(x) = (\nabla f_j(x))^T.$$

We have already convinced ourselves, that the complex VJP reduces to the real case, if everything (primals, cotangents and primitives) is real.

If a real-to-real function internally uses complex intermediate values, we still get the correct derivatives.

- For real-valued functions $f : \mathbb{C}^n \rightarrow \mathbb{R}^m$ we get the holomorphic derivative with an additional factor two

$$\text{VJP}_f(z, e_j^T) = 2 \frac{\partial f_j}{\partial z}(x).$$

This factor of two can even be beneficial, for example when f_j is to be optimized. See section 2.4.

- For holomorphic functions $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ we get the holomorphic derivative

$$\text{VJP}_f(z, e_j^T) = \frac{\partial f_j}{\partial z}(x).$$

Note that we do not need to require the primitives to be holomorphic.

Similar to forward mode, it is possible to extract useful information from backward mode autodiff in the other cases (real-to-complex or non-holomorphic complex-to-complex functions), but the results may not be intuitive. We discuss them in appendix C.2.

3.5 Implementation in Jax for Python

In this section, we explain the interface of `jax` [49], a `python` package that has (among other features) full autodiff functionality and in particular supports autodiff for complex functions. Naturally, this references the interface of `jax` *at the time of writing* and serves to make the abstract discussion of the previous sections more tractable.

We have based the previous sections, in particular the complex JVP and VJP definitions, on the conventions of `jax`, which are introduced in their “autodiff cookbook” [59].

In `jax`, the functions that provide an interface with the execution of forward and backward mode autodiff are `jax.jvp` and `jax.vjp`. They employ an autodiff formalism very similar to what we have developed in section 3.3 to compute the JVP (or VJP) of any `python` function⁶ using its internal implementations of the JVPs (or VJP) of the primitives.

Since we get the primal value of a function almost for free when we perform autodiff, they also return them. Their usage is outlined in figures 3.11 and 3.12.

There are convenience wrappers, namely `jax.grad` for computing the gradient of a scalar function in backward mode, as well as `jax.jacfwd` and `jax.jecrev`, which compute the full Jacobian of a function using (many iterations of) automatic differentiation in forward mode or backward mode, respectively.

⁶As long as it is implemented in terms of `jax` primitives and `python` built-in numerical operations (such as addition etc.).

```

from jax import jvp

def f(x, y, z):
    a = x + 2 * y
    b = x - z
    return a, b

# input primals put into a tuple, such that f(*primals_in) is valid
primals_in = (1., 2., 3.)

# input tangents, same structure.
tangents_in = (0., 1., 0.)

primals_out, tangents_out = jvp(f, primals_in, tangents_in)

a, b = primals_out
# same as f(*primals_in)
# a == 5. and b == -2.

# for this choice of tangents_in, these are the derivatives
# w.r.t the 2nd argument (y).
da_dy, db_dy = tangents_out
# da_dy == 2. and db_dy == 0.

```

Figure 3.11: Example usage of forward mode autodiff in `jax`. The API function `jax.jvp` performs forward mode autodiff according to section 3.3.4. It returns both the output primals $(a, b) = f(x, y, z)$, as well as the output tangents $(\dot{a}, \dot{b}) = \text{JVP}_f((x, y, z), (\dot{x}, \dot{y}, \dot{z}))$.

3.6 Review of Automatic Differentiation Tools

We review different tools for automatic differentiation in table 3.1.

We would like highlight the `python` package `autodidact`. It is a pedagogical version of `autograd` that was designed to be easy to read and understand. In order to understand the inner workings of an autodiff package, it is worthwhile to read the sourcecode of `autodidact`, which can be found at <https://github.com/mattjj/autodidact>.

In the early stages of this thesis, we used `PyTorch`, but transitioned to `jax` because we need complex autodiff in order to treat time evolution (see sections 5.2 and 6.4).

`Jax` has an accessible interface, even for beginners and is well documented. No knowledge about how automatic differentiation works is required to use it. The numeric backend is stable and efficient and there is a small but active community on GitHub that develops features and is open for discussion and requests.

```
from jax import vjp

def f(x, y, z):
    a = x + 2 * y
    b = x - z
    return a, b

primals_in = (1., 2., 3.)
primals_out, f_vjp_fun = vjp(f, *primals_in)

a, b = primals_out
# same as f(*primals_in)
# a == 5. and b == -2.

# output cotangents. same structure as output of f.
cotangents_out = (1., 0.)

cotangents_in = f_vjp_fun(cotangents_out)

# for this choice of cotangents_out, the VJP computes derivatives
# of the first output (a).
da_dx, da_dy, da_dz = cotangents_in
# da_dx == 1. and da_dy == 2. and da_dz == 0.
```

Figure 3.12: Example usage of backward mode autodiff in `jax`. The API function `jax.vjp` performs backward mode autodiff according to section 3.3.5. When `jax.vjp` is called the primal pass is performed and the output primals are returned, as well as a function (here `f_vjp_fun`) that executes the backward pass when it is called.

| Language | Package | Fwd | Bwd | ℂ | Ref. | URL | Notes |
|----------|----------------|-----|-----|---|------|---|--|
| AMPL | AMPL | ✓ | ✓ | ✗ | [81] | https://ampl.com/ | commercial software |
| C, C++ | ADIC | ✓ | ✓ | ✗ | [82] | http://www.mcs.anl.gov/adic/ | open-source |
| | ADOL-C | ✓ | ✓ | ✗ | [83] | https://github.com/coin-or/ADOL-C | |
| C++ | Ceres Solver | ✓ | ✗ | ✗ | | http://ceres-solver.org/ | |
| | CppAD | ✓ | ✓ | ✗ | [84] | https://coin-or.github.io/CppAD/doc/cppad.htm | |
| | FADBAD++ | ✓ | ✓ | ✗ | [85] | http://www.fadbad.com/fadbad.html | |
| C# | AutoDiff | ✗ | ✓ | ✗ | [86] | https://github.com/alexshtf/autodiff | |
| C#, F# | DiffSharp | ✓ | ✓ | ✗ | [47] | http://diffsharp.github.io/DiffSharp/index.html | |
| Fortran | ADIFOR | ✓ | ✓ | ✓ | [87] | https://www.mcs.anl.gov/research/projects/adifor/ | commercial software |
| | NAGWare | ✓ | ✓ | ✗ | [88] | https://www.nag.co.uk | |
| | TAMC | ✓ | ✓ | ✗ | [89] | http://autodiff.com/tamc/ | |
| Haskell | ad | ✓ | ✓ | ✗ | | https://github.com/ekmett/ad | |
| Java | Deriva | ✗ | ✓ | ✗ | | https://github.com/lambder/Deriva/ | complex support on road-map |
| Julia | JuliaDiff | ✓ | ✓ | ✗ | [90] | https://www.juliadiff.org/ | ℂ-support might get implemented |
| Lua | torch-autograd | ✓ | ✓ | ✗ | | https://github.com/twitter/torch-autograd | |
| MATLAB | INTLab | ✓ | ✗ | ✗ | [91] | http://www.ti3.tu-harburg.de/rump/intlab/ | |
| Python | autograd | ✓ | ✓ | ✓ | [71] | https://github.com/HIPS/autograd | active development moved to Jax |
| | autodidact | ✓ | ✓ | ✓ | | https://github.com/mattjj/autodidact | pedagogical version of autograd |
| | PyTorch | ✓ | ✓ | ✗ | [92] | https://pytorch.org/ | |
| | Tangent | ✓ | ✓ | ✗ | [93] | https://github.com/google/tangent | |
| | Jax | ✓ | ✓ | ✓ | [94] | https://github.com/google/jax | Combining features of autograd and XLA |

Table 3.1: Overview of Autodiff packages. This list is a revized and updated version of a similar list in [47]. The Fwd and Bwd columns signify if the package supports forward mode and backward mode autodiff respectively. The column labelled ℂ signifies if the package supports autodiff of complex functions.

3.7 A Short Glossary of Terminology and Jargon

autodiff Automatic differentiation.

backward mode An operational mode of automatic differentiation that allows to compute VJPs. This can be used to compute derivatives of one output components w.r.t all input components. See sections 3.3.3 and 3.3.5

backward pass See cotangent pass.

cost function A term borrowed from machine learning and neural networks. A scalar function. Typically, in the context of obtaining its gradient via backward mode autodiff in order to minimize it.

cotangent In backward mode autodiff, every primal value (every input, intermediate and output) is associated with a cotangent. Backward mode autodiff is a scheme to compute input cotangents from output cotangents, which practically evaluates the VJP of a top-level function. See equation (3.19).

cotangent pass The process of computing the input cotangents from the output cotangents in backward mode. See figure 3.10.

forward mode An operational mode of automatic differentiation that allows to compute JVPs. This can be used to compute derivatives of all output components w.r.t one input component. See sections 3.3.2 and 3.3.4.

forward pass Only in context of backward mode automatic differentiation. See primal pass.

JVP Jacobian Vector product. See equation (3.37)

primal A term to distinguish “normal” (=primal) values from tangents and/or cotangents. Primal can refer to an input, intermediate or output value.

primal pass The process of computing the function value, using the built-in primitives.

primitive Primitive functions are the building-blocks of a numerical autodiff package. The package should have implementations for the primitive functions, as well as their JVP and/or VJP. In the primal pass, the primitives are used to compute the function value. In the tangent (cotangent) pass, the JVPs (VJPs) of the primitives are used to compute the JVP (VJP) of the function. See section 3.2.1.

tangent In forward mode autodiff, every primal value (every input, intermediate and output) is associated with a tangent. Forward mode autodiff is a scheme to compute output tangents from input tangents, which practically evaluates the JVP of input-to-output function. See equation (3.14).

tangent pass The process of computing the output tangents from the input tangents in forward mode. See figure 3.9

VJP Vector Jacobian product. See equation (3.38).

Chapter 4

Tensor Networks and Algorithms

Contents

| | | |
|------------|--|-----------|
| 4.1 | Diagrammatic Notation of Tensor Networks | 56 |
| 4.2 | Projected Entangled Pair States (PEPS) | 57 |
| 4.2.1 | PEPS States | 57 |
| 4.2.2 | Infinite PEPS (iPEPS) States | 58 |
| 4.2.3 | Variational Power of PEPS States | 59 |
| 4.3 | PEPOs (PEPS Operators) for Nearest Neighbor Operators . . | 59 |
| 4.3.1 | An MPO for the 1D TFIM | 60 |
| 4.3.2 | A PEPO for the 2D TFIM | 60 |
| 4.3.3 | PEPOs for General Nearest Neighbor Operators | 62 |
| 4.4 | PEPOs for Time Evolution Operators | 63 |
| 4.5 | Expectation Values and Matrix Elements in PEPS States . . . | 65 |
| 4.5.1 | Brute-Force Contraction | 66 |
| 4.5.2 | Contraction via Boundary-MPS Compression | 67 |
| 4.6 | Corner Transfer Matrix Renormalization Group (CTMRG) . . | 70 |

4.1 Diagrammatic Notation of Tensor Networks

When using tensor networks, we need to deal with contraction of tensors (objects with many indices). For a generic example consider the contraction of five tensors A , B , C , D and E :

$$v_i := \sum_{\alpha\beta\gamma\delta\epsilon\zeta\eta} A_{\alpha\eta\zeta\zeta\beta} B_{\eta\alpha\gamma\delta} C_{\gamma\beta} D_{\epsilon\delta i} E_{\epsilon}. \quad (4.1)$$

The above notation is cumbersome and not particularly insightful. It is instead more intuitive to use the typical *diagrammatic notation*, which we introduce in this section.

The central idea is to represent tensors (objects with indices) as shapes, and the indices as lines or “legs”. A vector, that is an object \mathbf{a} with entries a_i and a single index i is represented by a shape with a single *open leg*:

$$a_i = \begin{array}{c} \textcircled{a} \\ | \\ i \end{array}. \quad (4.2)$$

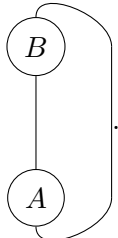
Similarly, a matrix A (which is a rank-2 tensor, an object with two indices) is represented by a shape with two open legs:

$$A_{ij} = \begin{array}{c} j \\ | \\ \textcircled{A} \\ | \\ i \end{array}. \quad (4.3)$$

Higher-rank tensors, such as the wave function coefficients of a many-body state are represented by shapes with as many legs, as the tensor has indices. E.g.

$$\langle i_1, \dots, i_7 | \psi \rangle = \psi_{i_1, \dots, i_7} = \begin{array}{c} \boxed{\psi} \\ | \quad | \quad | \quad | \quad | \quad | \quad | \\ i_1 \quad i_2 \quad i_3 \quad i_4 \quad i_5 \quad i_6 \quad i_7 \end{array}. \quad (4.4)$$

Indices that are contracted (i.e. summed over) are represented by *closed legs*, legs that connect two shapes. For example, the trace of a matrix product can be written as

$$\text{Tr}(A \cdot B) = \sum_{ij} A_{ij} B_{ji} = \begin{array}{c} \textcircled{B} \\ | \\ \textcircled{A} \\ | \end{array} \quad (4.5)$$


As a final example, let us write equation (4.1) in diagrammatic form:

$$v_i = \sum_{\alpha\beta\gamma\delta\epsilon\zeta\eta} A_{\alpha\eta\zeta\zeta\beta} B_{\eta\alpha\gamma\delta} C_{\gamma\beta} D_{\epsilon\delta i} E_{\epsilon} =$$
(4.6)

The labels on the closed legs are typically omitted and only added here for clarity. We refer to the left side of equation (4.6) as (the contraction of) a *tensor network* or *diagram*.

4.2 Projected Entangled Pair States (PEPS)

In this section, we introduce projected entangled pair states (PEPS) [22, 23, 34], as well as their thermodynamic limit, infinite PEPS (iPEPS) [38]. They are an ansatz class of states on two-dimensional lattices¹. We understand PEPS states as direct generalizations of Matrix Product States (MPS) to two-dimensional systems.

4.2.1 PEPS States

We understand the MPS for 1D chains to be built from the following recipe: At each site of the lattice (which is a chain in 1D), put a tensor. This tensor has a physical index/leg for the local Hilbert space of that lattice site and one virtual index/leg for each lattice bond ending at that site. The coefficients of the many-body wave function are obtained by contracting all virtual indices in the resulting tensor network.

Generalizing this recipe to a $L_x \times L_y$ square lattice, we get a parametrization of the many-body state in terms of tensors $\{a_{x,y}\}$ at every lattice site:

$$|\psi(\{a_{x,y}\})\rangle = \sum_{\mathbf{s}} \psi_{\mathbf{s}}(\{a_{x,y}\}) |\mathbf{s}\rangle, \quad (4.7)$$

where $|\mathbf{s}\rangle$ is the product state corresponding to the classical spin configuration $\mathbf{s} = \{s_{x,y}\}$ and $s_{x,y} = \pm 1$ is the orientation of the spin at site (x, y) . In the numerics of this work we use the z -basis, i.e. $|s = \pm 1\rangle = |\uparrow\rangle$ ($|\downarrow\rangle$) is the eigenstate of σ^z with eigenvalue ± 1 .

The coefficients $\psi_{\mathbf{s}}(\{a_{x,y}\})$ are given by the contraction of a tensor network:

$$\psi_{\mathbf{s}}(\{a^{(x,y)}\}) = \sum_{\{u_{x,y}\}, \{r_{x,y}\}} \prod_{x,y} a_{s_{x,y}; u_{x,y}; r_{x-1,y}; u_{x,y-1}; r_{x,y}}^{(x,y)}, \quad (4.8)$$

where $u_{x,y}$ is the upper virtual index of the tensor $a^{(x,y)}$, which coincides with the lower virtual leg of $a^{(x,y+1)}$ (see figure 4.1). Analogously, $r_{x,y}$ is the right virtual index of $a^{(x,y)}$.

¹Some authors also call higher-dimensional MPS-analogues PEPS states. We refer to these as tensor product states (TPS), and understand PEPS (with its historical name) as the TPS in two dimensions.

The number of indices makes equation (4.8) barely readable and we will therefore prefer diagrammatic notation. It is illustrated in figure 4.1, where a 4×4 square lattice is depicted as an example.

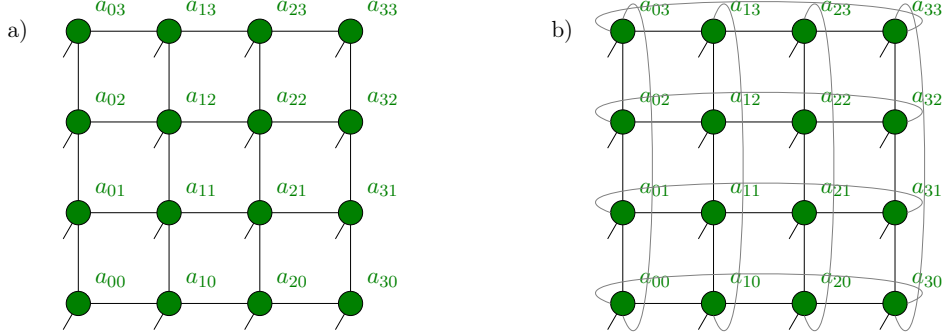


Figure 4.1: PEPS states on a 4×4 square lattice a) for open boundary conditions, b) for periodic boundary conditions. The contraction of the shown tensor network gives the wave function coefficients $\psi_s(\{a_{x,y}\})$ (see equation (4.7)). The physical leg below the tensor $a^{(x,y)}$ is $s_{x,y}$.

Generalized PEPS tensor networks can also be defined on any other lattice and in any number of spatial dimensions. In this work we focus on the 2D square lattice.

4.2.2 Infinite PEPS (iPEPS) States

We can naturally extend the PEPS states from finite lattices to infinite lattices. This gives rise to the infinite PEPS (iPEPS) states. The state of the infinite system is parametrized using a finite number of tensors by choosing a unit cell and tiling the infinite plane with it. See figure 4.2.

In this thesis we only work with translationally invariant iPEPS. That is, the unit cell only consists of a single site and the state is therefore parametrized by a single tensor. Additionally we only consider C_4v symmetric iPEPS states. That is, the iPEPS tensor is invariant under rotations and reflections:

While imposing this symmetry on the virtual bonds might restrict the variational power of the iPEPS, it significantly decreases computational costs and has proven successful in practice [22, 95–97].

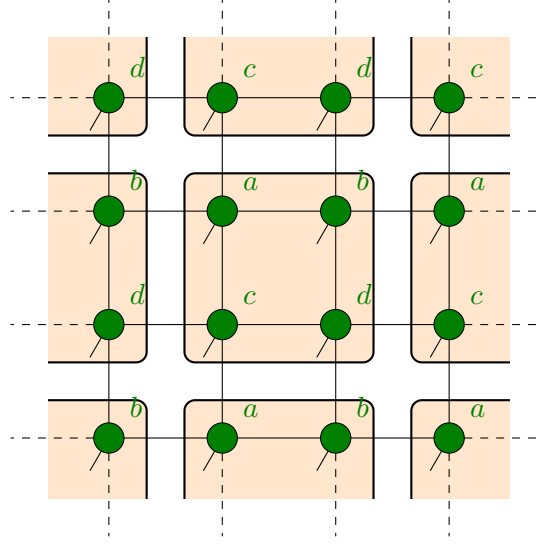


Figure 4.2: Infinite PEPS (iPEPS) state. The PEPS states of figure 4.1 are extended to the infinite square lattice by choosing a unit cell (orange areas) and repeating the tensors of the unit cell (here a, b, c, d) to fill the infinite plane.

4.2.3 Variational Power of PEPS States

On a finite (square) lattice with a finite-dimensional local Hilbert space, any state can be written as a PEPS, if the bond dimension χ is large enough [22, p.25]. The bond dimension required to represent any state in the Hilbert space grows exponentially with system size.

In 1D, ground states of any local gapped Hamiltonian can be approximated to any given accuracy by MPS of a finite bond dimension *independent of system size* [22, 98]. We can expect that, also in 2D, low-energy states of interesting models can be well approximated by PEPS of a small, finite bond dimension *independent of system size* [19, 20, 25].

While there is no rigorous proof for the general statement, there is promising empirical evidence that is true for a wide class of interesting applications [99–103]. Even states with polynomially decaying correlations (critical states) can be represented by PEPS, already with the lowest non-trivial bond dimension of $\chi = 2$ [102, 103].

4.3 PEPOs (PEPS Operators) for Nearest Neighbor Operators

In order to evaluate expectation values (or matrix elements) in states that are parametrized as PEPS, it will prove to be convenient to write them as PEPOs (PEPS-like operator).

That is, we write an operator O in terms of its coefficients in a local basis;

$$O = \sum_{s,s'} o_{s,s'} |s\rangle\langle s'| \quad ; \quad o_{s,s'} := \langle s|O|s'\rangle \quad (4.10)$$

and represent the coefficients $o_{s,s'}$ as a tensor network, similar to the PEPS tensor networks

for the wave-function coefficients of states in figures 4.1 and 4.2. The only difference is that each PEPO tensor $a^{(x,y)}$ has two physical legs, one for $s_{x,y}$ and one for $s'_{x,y}$.

In this section, we develop a strategy to write nearest-neighbor operators, such as the Hamiltonians of the Ising or Heisenberg models as (the sum of two) PEPO operators.

We construct a PEPO for the transverse field Ising model

$$H = - \sum_{\langle i,j \rangle} \sigma_x^{[i]} \sigma_x^{[j]} - g \sum_i \sigma_z^{[i]}, \quad (4.11)$$

where $\sigma_x^{[i]}$ is the Pauli x operator acting on site i . We show how the procedure can be generalized to any nearest-neighbor operator in subsection 4.3.3. Throughout this section we will denote which site(s) of the lattice an operator acts on by a superscript in square brackets.

4.3.1 An MPO for the 1D TFIM

For demonstrative purposes and as a building block for the 2D case, we will first present a matrix product operator (MPO) for the transverse field Ising model (equation (4.11)) on the 1D chain.

The MPO, for open boundary conditions, is given by

$$H = - \sum_{x=1}^{L-1} \sigma_x^{[x]} \sigma_x^{[x+1]} - g \sum_{x=1}^L \sigma_z^{[x]} = \mathbf{v}_L^{(3)} \cdot C^{[1]} \cdot C^{[2]} \cdot \dots \cdot C^{[L]} \cdot \mathbf{v}_R^{(3)}, \quad (4.12)$$

where the boundary vectors $\mathbf{v}_{L/R}^{(3)}$ and the operator-valued matrices $C^{[x]}$ are defined as

$$\mathbf{v}_L^{(3)} := \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}, \quad C^{[x]} := \begin{pmatrix} \mathbb{1}^{[x]} & 0 & 0 \\ -\sigma_x^{[x]} & 0 & 0 \\ g\sigma_z^{[x]} & \sigma_x^{[x]} & \mathbb{1}^{[x]} \end{pmatrix}, \quad \mathbf{v}_R^{(3)} := \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \quad (4.13)$$

4.3.2 A PEPO for the 2D TFIM

We divide the Hamiltonian for the 2D TFIM into a “horizontal” and a “vertical” part. The horizontal part contains all bond-terms for horizontal bonds and the vertical part the terms for all vertical bonds. The site-terms are distributed evenly. In particular we define

$$H_{\text{hor}} := \sum_{y=1}^{L_y} \left[- \sum_{x=1}^{L-1} \sigma_x^{[x,y]} \sigma_x^{[x+1,y]} - \frac{g}{2} \sum_{x=1}^L \sigma_z^{[x,y]} \right] =: \sum_{y=1}^{L_y} h^{[:,y]} \quad (4.14)$$

and analogously (exchanging the roles of x and y) define H_{vert} , such that $H = H_{\text{hor}} + H_{\text{vert}}$.

The horizontal part is a sum of terms $h^{[:,y]}$, which are essentially TFIM Hamiltonians acting on y^{th} row as a 1D chain, decorated with a factor $1/2$ at the transverse field g . Inspired by the MPO for 1D chains we can write down a PEPO for H_{hor} and H_{vert} :

$$H_{\text{hor}} = \text{diagram} \quad ; \quad H_{\text{vert}} = \text{diagram} \quad (4.15)$$

where

$$\begin{aligned} \bigcirc &:= \mathbf{v}_L^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad , \quad -\bigcirc &:= \mathbf{v}_R^{(3)} := \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\ \bullet &:= \mathbf{v}_L^{(2)} := \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad , \quad \bullet &:= \mathbf{v}_R^{(2)} := \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ I_{ij}^{[x,y]} &:= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mathbb{1}[x,y] & 0 & 0 \end{pmatrix}_{ij} \quad , \quad i - \text{orange square} - j &:= C_{ij}^{[x,y]} := \begin{pmatrix} \mathbb{1}[x,y] & 0 & 0 \\ -\sigma_x^{[x,y]} & 0 & 0 \\ -\frac{g}{2}\sigma_z^{[x,y]} & \sigma_x^{[x,y]} & \mathbb{1}[x,y] \end{pmatrix}_{ij} \\ i - \text{green circle} - j &:= D_{ijkl}^{[x,y]} := \begin{pmatrix} I_{ij}^{[x,y]} & 0 \\ C_{ij}^{[x,y]} & I_{ij}^{[x,y]} \end{pmatrix}_{kl} \end{aligned} \quad (4.16)$$

We can see that this is indeed a PEPO for H_{hor} by contracting the horizontal rows first:

$$E_{kl}^{[:,y]} := \mathbf{v}_L^{(3)} C^{[1,y]} C^{[2,y]} \dots C^{[L_x/2-1,y]} D_{::,k,l}^{[L_x/2,y]} C^{[L_x/2+1,y]} \dots C^{[L_x,y]} \mathbf{v}_R^{(3)}, \quad (4.17)$$

where $D_{::,k,l}$ is the (operator-valued) matrix with entries $(D_{::,k,l})_{ij} = D_{ijkl}$. This notation is inspired by Numpy array-slicing.

The bottom-left entry of this row-matrix is an MPO of the 1D TFIM on the y^{th} row, so it evaluates to $h^{[:,y]}$

$$E_{2,1}^{[:,y]} = \mathbf{v}_L^C C^{[1,y]} C^{[2,y]} \dots C^{[L_x/2-1,y]} C^{[L_x/2,y]} C^{[L_x/2+1,y]} \dots C^{[L_x,y]} \mathbf{v}_R^C = h^{[:,y]}. \quad (4.18)$$

On the diagonal we can use that $I^{[x,y]} C^{[x+1,y]} = I^{[x,y]} I^{[x+1,y]}$ and $C^{[x-1,y]} I^{[x,y]} = I^{[x-1,y]} I^{[x,y]}$ to obtain

$$E_{1,1}^{[:,y]} = E_{2,2}^{[:,y]} = \mathbf{v}_L^C C^{[1,y]} \dots C^{[L_x/2-1,y]} I^{[L_x/2,y]} C^{[L_x/2+1,y]} \dots C^{[L_x,y]} \mathbf{v}_R^C = \mathbb{1}^{[:,y]}. \quad (4.19)$$

With the trivial entry $E_{1,2}^{[:,y]} = 0$, we then get the full PEPO. The vertical contraction looks like another MPO, but the matrix-entries are operators that each act on entire rows of

sites

$$\begin{aligned} \mathbf{v}_L^{(2)} E^{[:,1]} E^{[:,2]} \dots E^{[:,L_y]} \mathbf{v}_R^{(2)} &= \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbb{1}^{[:,1]} & 0 \\ h^{[:,1]} & \mathbb{1}^{[:,1]} \end{pmatrix} \cdots \begin{pmatrix} \mathbb{1}^{[:,L_y]} & 0 \\ h^{[:,L_y]} & \mathbb{1}^{[:,L_y]} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \sum_{y=1}^{L_y} h^{[:,y]} = H_{\text{hor}}. \end{aligned} \quad (4.20)$$

The construction for H_{vert} is completely analogous, just rotated by 90° .

4.3.3 PEPOs for General Nearest Neighbor Operators

The construction in equation (4.15) can be generalized to any nearest-neighbor operator. Consider a general operator O with N nearest-neighbor terms:

$$O = \sum_{\langle i,j \rangle} \sum_{n=1}^N A_n^{[i]} B_n^{[j]} + \sum_i S^{[i]}. \quad (4.21)$$

We assume that any scalar coefficients are absorbed into the on-site operators $A_n^{[i]}$, $B_n^{[i]}$ and $C^{[i]}$.

We can get PEPOs for O_{hor} and O_{vert} , such that $O = O_{\text{hor}} + O_{\text{vert}}$, by making the following adjustments to equation (4.16).

First, replace $\mathbf{v}_{L/R}^{(3)}$ with $\mathbf{v}_L^{(N)} := \mathbf{e}_N \in \mathbb{R}^N$ and $\mathbf{v}_R^{(N)} := \mathbf{e}_1 \in \mathbb{R}^N$. Secondly, (re-)define

$$I^{[x,y]} := \begin{pmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \\ \mathbb{1}^{[x,y]} & 0 & \dots & 0 \end{pmatrix}; \quad C^{[x,y]} := \begin{pmatrix} \mathbb{1}^{[x,y]} & 0 & \dots & 0 & 0 \\ A_1^{[x,y]} & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_N^{[x,y]} & 0 & \dots & 0 & 0 \\ \frac{1}{2}S^{[x,y]} & B_1^{[x,y]} & \dots & B_N^{[x,y]} & \mathbb{1}^{[x,y]} \end{pmatrix}. \quad (4.22)$$

Anisotropic nearest-neighbor interactions can also be introduced by using different versions of $C^{[x,y]}$ in the horizontal and vertical parts.

4.4 PEPOs for Time Evolution Operators

In this section, we develop a PEPO for the time evolution operator of the transverse field Ising model

$$H = - \sum_{\langle i,j \rangle} \sigma_i^x \sigma_j^x - g \sum_i \sigma_i^z =: H_{\text{bond}} + H_{\text{site}}. \quad (4.23)$$

Our approach is inspired by similar work in [22]. We approximate the time evolution operator $U(\delta t) = \exp(-iH\delta t)$ using the second-order Suzuki-Trotter expansion [104] of the exponential

$$\begin{aligned} U(\delta t) &= \exp(-i(H_{\text{bond}} + H_{\text{site}})\delta t) \\ &\approx \exp(-iH_{\text{site}}\delta t/2) \exp(-iH_{\text{bond}}\delta t) \exp(-iH_{\text{site}}\delta t/2) + \mathcal{O}(\delta t^3). \end{aligned} \quad (4.24)$$

The contribution to (4.24) from the site-terms factorizes into single-site operators

$$U_{\text{site}}(\delta t) := \exp(-iH_{\text{site}}\delta t) = \exp\left(ig\delta t \sum_i \sigma_i^z\right) = \prod_i \exp(ig\delta t \sigma_i^z) =: \prod_i v_i(\delta t). \quad (4.25)$$

The contribution to (4.24) from the bond-terms factorizes into bond operators

$$\begin{aligned} U_{\text{bond}}(\delta t) &:= \exp(-iH_{\text{bond}}\delta t) = \exp\left(i\delta t \sum_{\langle i,j \rangle} \sigma_i^x \sigma_j^x\right) \\ &= \prod_{\langle i,j \rangle} \exp(i\delta t \sigma_i^x \sigma_j^x) =: \prod_{\langle i,j \rangle} w_{ij}(\delta t). \end{aligned} \quad (4.26)$$

We can split these bond operators analytically by using that $(\sigma^x)^2 = \mathbb{1}$:

$$\begin{aligned} w_{ij}(\delta t) &= \exp(i\delta t \sigma_i^x \sigma_j^x) = \cos(\delta t \sigma_i^x \sigma_j^x) + i \sin(\delta t \sigma_i^x \sigma_j^x) \\ &= \cos(\delta t) \mathbb{1}_i \mathbb{1}_j + i \sin(\delta t) \sigma_i^x \sigma_j^x = \sum_{k=0}^1 A_i^{(k)}(\delta t) B_j^{(k)}, \end{aligned} \quad (4.27)$$

with $A(\delta t)$ and B defined by

$$A^{(0)}(\delta t) = \cos(\delta t) \mathbb{1} \quad , \quad B^{(0)} = \mathbb{1} \quad , \quad A^{(1)}(\delta t) = \sin(\delta t) \sigma^x \quad , \quad B^{(1)} = \sigma^x. \quad (4.28)$$

Or in terms of diagrams,

$$\begin{array}{c} | \\ \boxed{w(\delta t)} \\ | \end{array} = \begin{array}{c} | \\ \boxed{A(\delta t)} \end{array} \overset{k}{\text{---}} \begin{array}{c} | \\ \boxed{B} \\ | \end{array}, \quad (4.29)$$

where the left (right) legs are the physical indices of the operator acting on site i (j).

We can therefore build a PEPO for $U(\delta t)$ from the PEPO-tensor b defined in figure 4.3.

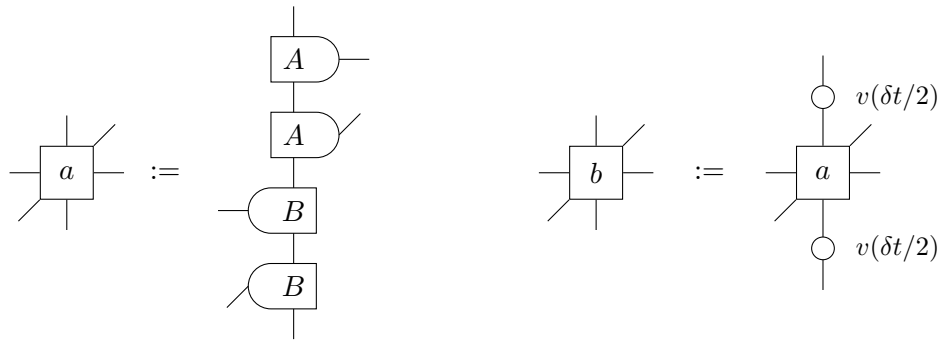


Figure 4.3: PEPO tensors for a) the bond-evolution operator $U_{\text{bond}}(\delta t)$ b) the evolution operator $U(\delta t)$. The tensors A and B are given by equation (4.28) and $v(\delta t/2)$ is given by equation (4.25). On the left side, the PEPO tensor a is designed in such a way, that contracting a layer of such tensors always results in the contraction of an A tensor with the corresponding B tensor of the neighbour. This realizes equation (4.29) on all bonds and therefore a PEPO built from only a -tensors is the operator $U_{\text{bond}}(\delta t)$ (equation (4.26)). On the right side, the b tensor is obtained by applying $v(\delta t/2)$ before and after a . This realizes the second-order Suzuki-Trotter expansion (4.24) and therefore a PEPO built as a layer of repeated b -tensors is the evolution operator $U(\delta t)$ in Suzuki-Trotter approximation. For periodic boundary conditions, a full layer of only b tensors can be used. For open boundary conditions, the tensors at the boundary are modified by simply omitting the A or B tensor whose leg would point out of the boundary.

4.5 Expectation Values and Matrix Elements in PEPS States

In this section, we discuss how expectation values and matrix elements are obtained numerically, if the states are parametrized as PEPS and the operators are either local operators or PEPOs. In particular, we want to compute $\langle \phi | O | \psi \rangle$ for PEPS $|\phi\rangle$ and $|\psi\rangle$ and operators O that are either local or PEPOs as discussed in sections 4.3 and 4.4. In either case, the matrix element can be expressed as the contraction of a tensor network. We illustrate the network that needs to be contracted for single-site operators in figure 4.4, for bond operators in figure 4.5 and for PEPO operators in figure 4.6.

For the rest of this section we discuss strategies for numerically contracting these diagrams.

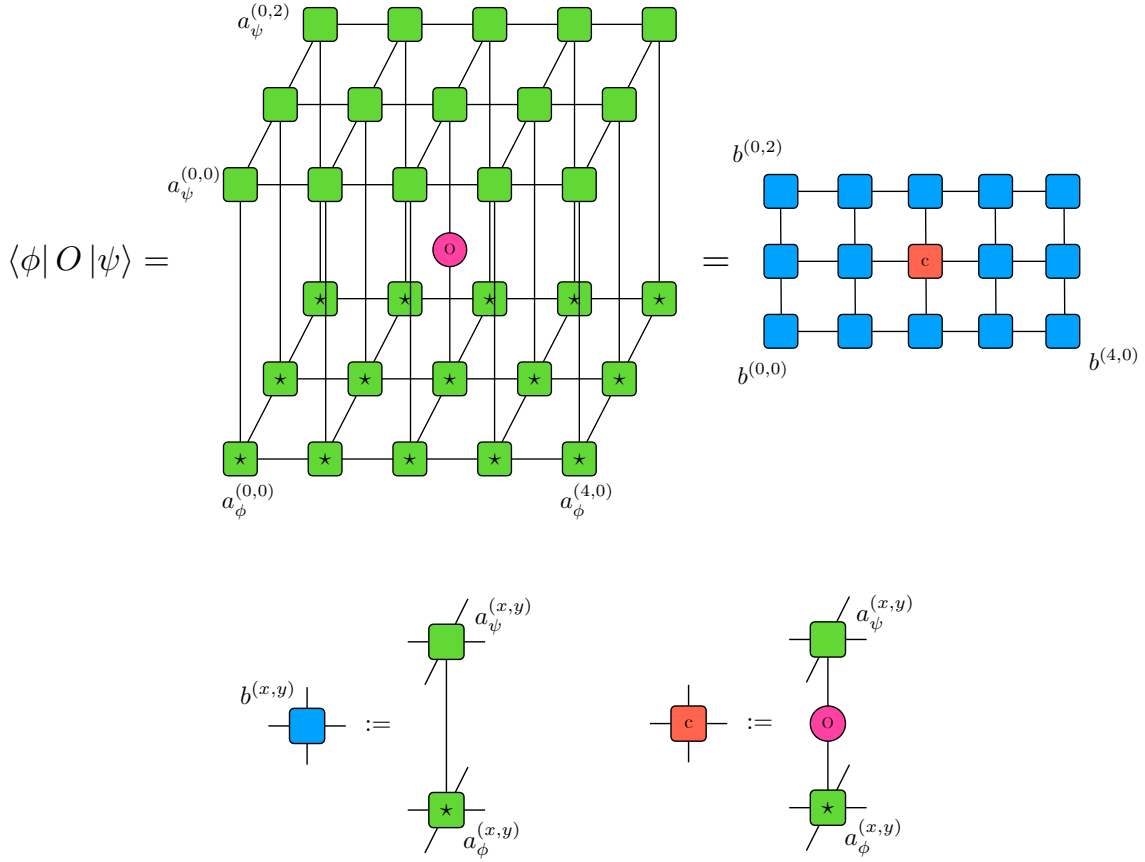


Figure 4.4: Matrix element of a single-site operator O in PEPS states $|\phi\rangle$ and $|\psi\rangle$, as an example on a 5×3 system. The PEPS tensors of $|\phi\rangle$ ($|\psi\rangle$) are $a_\phi^{(x,y)}$ ($a_\psi^{(x,y)}$) and \star denotes complex conjugation of a tensor. For convenience, we define intermediate contractions $b^{(x,y)}$ and c such that the contraction can be drawn as a flat diagram. Note that this does not reflect the optimal contraction order.

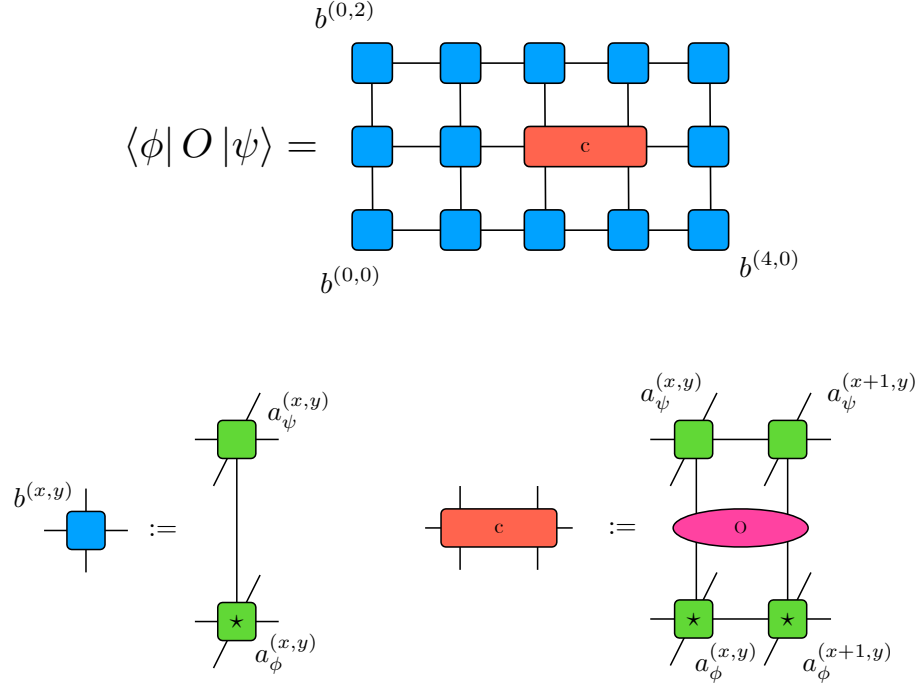


Figure 4.5: Matrix element of a bond operator O in PEPS states $|\phi\rangle$ and $|\psi\rangle$, as an example on a 5×3 system. The PEPS tensors of $|\phi\rangle$ ($|\psi\rangle$) are $a_\phi^{(x,y)}$ ($a_\psi^{(x,y)}$) and \star denotes complex conjugation of a tensor. For convenience, we define intermediate contractions $b^{(x,y)}$ and c such that the contraction can be drawn as a flat diagram. Note that this does not reflect the optimal contraction order.

4.5.1 Brute-Force Contraction

If the system size and PEPS bond dimension are small enough², the diagrams in figures 4.4, 4.5 and 4.6 can be contracted exactly. That means all tensor contractions can be explicitly performed without any approximations.

The efficiency of such contractions depends heavily on the order in which they are performed. As a simple example, consider three matrices $A \in \mathbb{R}^{10 \times 20}$, $B \in \mathbb{R}^{20 \times 100}$, $C \in \mathbb{R}^{100 \times 5}$. To compute $A \cdot B \cdot C$, we can either first multiply $A \cdot B$ and multiply the result with C , or first multiply $B \cdot C$ and multiply A with that result. In order to multiply a $m \times k$ matrix with a $k \times n$ matrix, a total of $m \cdot k \cdot n$ scalar multiplications are required. Therefore, the cost of the first option, in terms of number of scalar multiplications is $10 \cdot 20 \cdot 100 + 10 \cdot 100 \cdot 5 = 25000$, while the cost of the second option is only $20 \cdot 100 \cdot 5 + 10 \cdot 20 \cdot 5 = 11000$. The difference in cost between different contraction orders becomes even more pronounced with the higher connectivity of the tensor networks for PEPS contractions.

There are algorithms for finding optimal contraction orders in the public domain [105, 106]. A tool that proved particularly useful is `tensortrace`³, a GUI tool that allows the user to specify the tensor network in a graphical interface and obtain code for the contraction in optimal order for `python`, `MATLAB` or `Julia`. The `python` code uses `numpy` as a backend.

²For evaluating gradients with automatic differentiation, we found this approach feasible up to system size 8×8 for bond dimension $\chi = 2$ and up to 5×5 for bond dimension $\chi = 3$.

³See <https://www.tensortrace.com>

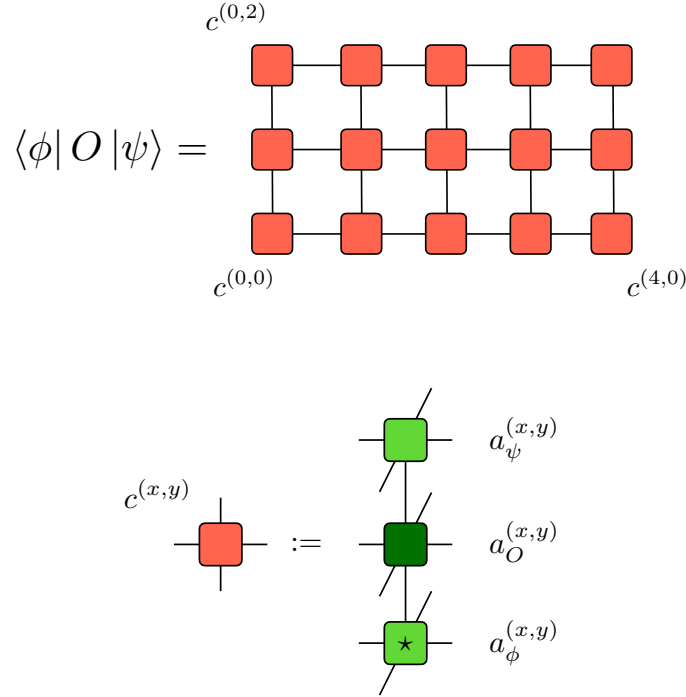


Figure 4.6: Matrix element of a PEPO operator O in PEPS states $|\phi\rangle$ and $|\psi\rangle$, as an example on a 5×3 system. The PEPS tensors of $|\phi\rangle$ ($|\psi\rangle$) are $a_\phi^{(x,y)}$ ($a_\psi^{(x,y)}$) and \star denotes complex conjugation of a tensor. The PEPO tensors of O are $a_O^{(x,y)}$. For convenience, we define intermediate contractions $c^{(x,y)}$ such that the contraction can be drawn as a flat diagram. Note that this does not reflect the optimal contraction order.

We have adapted the relevant `python` library to use `jax` as a backend⁴.

Optimal contraction orders can only be found for rather small networks (up to around 30 tensors) and are not viable for the contraction of a PEPO matrix element in PEPS states. Even for a 4×4 system, the diagram already consists of 48 tensors. We instead a priori find a recipe that gives us the contraction order for any system size. This contraction order is illustrated in figure 4.7.

4.5.2 Contraction via Boundary-MPS Compression

Since the brute-force contraction method is too expensive to be feasible for larger system sizes or higher bond dimensions, approximate contraction methods are typically employed. Additional work is necessary to make approximate contractions perform well with gradient-based optimization, which lies beyond the scope of this work. We will therefore only briefly introduce the boundary MPS (bMPS) contraction method [22, 107].

The central idea is to view the columns of the flat contraction diagrams (see e.g. figure 4.6) as MPS (for the two outermost columns) or MPOs (for the inner columns) and employ MPS methods to evaluate the diagram efficiently.

Assume that we are contracting a matrix element $\langle\phi|O|\psi\rangle$ of a PEPO O with bond dimension η in PEPS states $|\psi\rangle$ and $|\phi\rangle$, both with bond dimension χ . The leftmost column

⁴In order to avoid licencing issues, we will not publish this code.

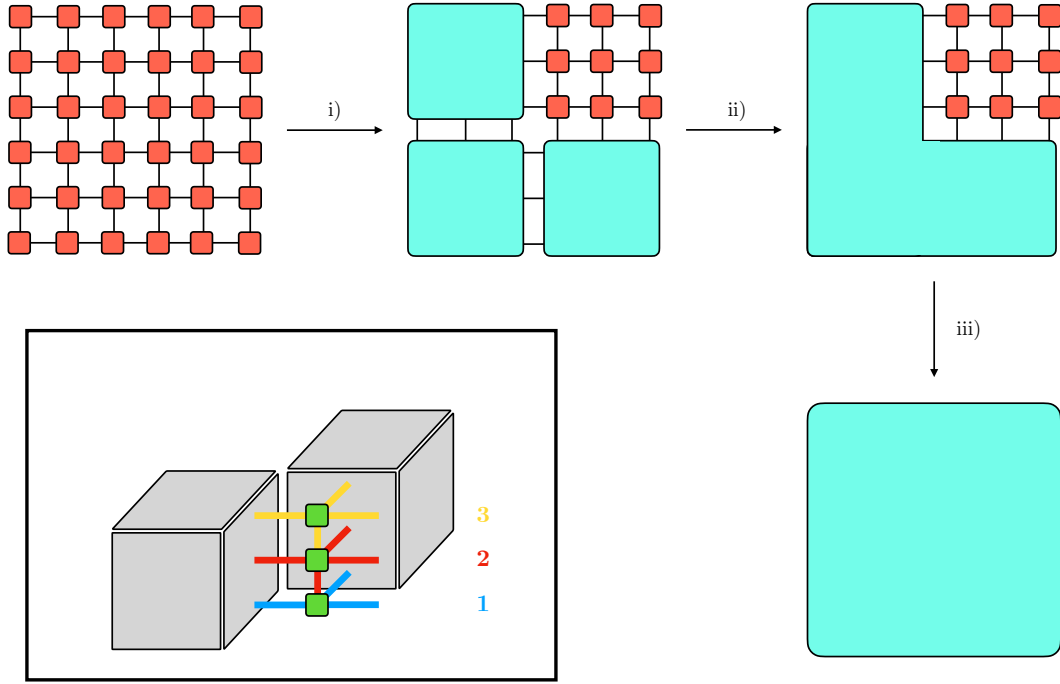


Figure 4.7: Contraction order for brute-force contraction of PEPS matrix elements. In step i) the system is divided into quadrants, then all tensors in three of the quadrants are contracted. This is done either by a process similar to the entire panel or for small quadrants (3×3 and smaller), just “row-wise”. In step ii) the quadrants are contracted with each other. In step iii) the rest of the network is contracted with the big tensor. If the remaining quadrant is small, this is done row-wise. If it is large, it is again divided into quadrants, which are contracted with the big tensor one by one. The inset shows the three-dimensional structure of the diagrams that was omitted in the outer overview. A red square in the flat diagram represents three tensors of the network (one from the ket, one from the operator and one from the bra). It is more efficient to contract them with the big tensor successively, instead of first contracting the three tensors to the flat (red) tensor and then contracting it with the big tensor. The overall cost, in terms of scalar multiplications is dominated by step ii) and is given by $\mathcal{O}(D^{L/2})$, where $D := \eta\chi^2$ is the bond dimension in the flat diagrams and η and χ are the bond dimensions of the PEPO operator and PEPS states, respectively. The cost is exponential in the (linear) system size. Therefore, only small systems are accessible using brute-force contraction.

is interpreted as an MPS, which we refer to as a *boundary MPS* (bMPS). Note that the $D := \eta\phi^2$ dimensional “physical” legs of the MPS are virtual bonds of the original PEPS and PEPO. The following columns are interpreted as *boundary MPOs* (bMPOs) with physical dimension D and virtual bond dimension D . We can therefore (exactly) express a partial contraction of the diagram, say the n leftmost columns as a bMPS with bond dimension D^n , simply by contracting the horizontal bonds. It is conjectured, and shown empirically [22], that the partial contraction can be well approximated by a bMPS with lower bond dimension.

A maximal bMPS bond dimension $\tilde{D} > D$ is chosen. Starting from the left, the product

of the current leftmost bMPS with the bMPO of the following column is performed, either exactly if the resulting bond dimension is lower than \tilde{D} , or else approximately.

Successively, more columns are absorbed into the left bMPS. A similar process is performed from the right and finally, when all columns have been absorbed into either the left or right bMPS, the diagram can be evaluated as their “scalar product”. Note that this is not a physical scalar product and neither bMPS is conjugated (since neither is a physical bra state). Let us illustrate the process as an overview:

The approximate bMPS - bMPO product can either be performed by first computing a bMPS of the exact product by contracting the horizontal legs and then compressing it⁵:

or alternatively using a variational method, similar to DMRG [24, 25]. A new bMPS for the bMPS - bMPO product is found by variationally optimising the overlap

with sweeps akin to DMRG.

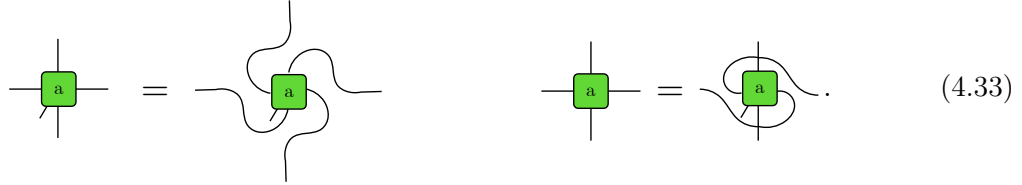
⁵The compression can be achieved by truncating the Schmidt-values in a canonical form of the bMPS.

4.6 Expectation Values in iPEPS states: Corner Transfer Matrix Renormalization Group (CTMRG)

In this section, we review the corner transfer matrix renormalization group (CTMRG) [96, 97, 108], a method for evaluating expectation values of local operators in iPEPS states.

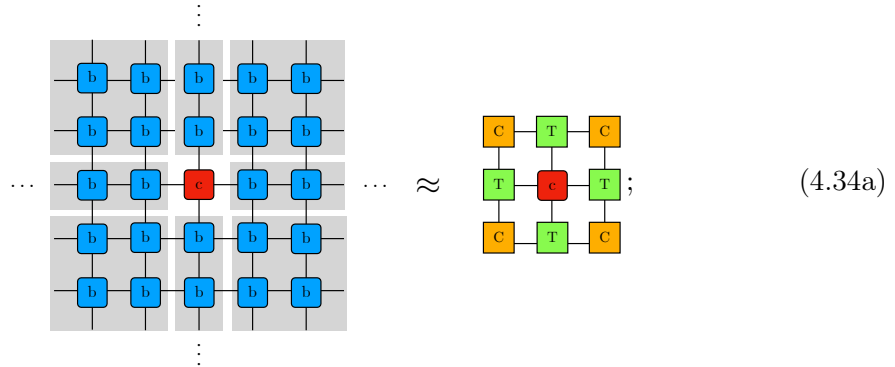
Operators of interest (Hamiltonians, order parameters) are typically translationally invariant sums of equivalent terms with small local support. Since the iPEPS state is by construction translationally invariant, it typically suffices to compute the expectation value of a few of these terms, such that they commensurate the unit cell of the iPEPS. The details depend on the unit cell of the iPEPS states and of the operator. Rotational symmetries can be exploited as well.

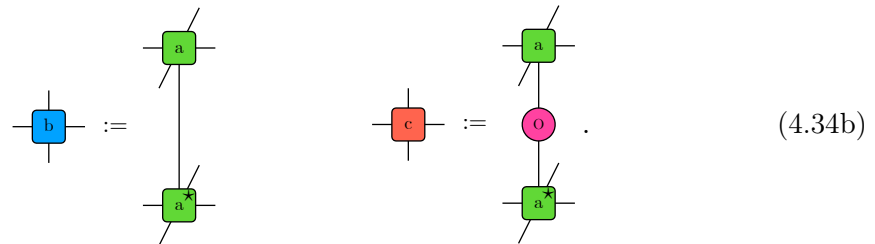
We focus on translationally invariant (single-site unit cells) and C_{4v} symmetric states and operators. We consider only single-site operators (such as the TFIM order parameter) and nearest-neighbor operators (such as the TFIM Hamiltonian) and require the iPEPS state to be C_{4v} symmetric. That is,



The method is, however, readily generalized to operators of larger support.

The diagram we need to contract is the infinitely extended version of figure 4.5. The central idea is to find *corner transfer matrices* C and T that behave like the infinite environment around the support of the operator, e.g. for a single site operator O :





The C_{4v} symmetry (4.33) translates to symmetries of the *CTMRG environment* $\{C, T\}$:

$$\text{Diagram (4.35)} \quad (4.35)$$

$$\text{Diagram (4.36)} \quad (4.36)$$

The CTMRG environment is found as the fixed point of the following renormalization scheme [96, 97]:

$$\text{Diagram (4.37)} \quad (4.37)$$

where U and V are given by the singular value decomposition (SVD)

$$\text{Diagram (4.38)} \quad (4.38)$$

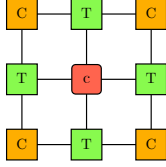
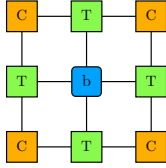
We derive it in figure 4.8. Note that the renormalization step (4.37) is invariant under the gauge freedom of the SVD (see section C.3.2).

A measure for the convergence is the *boundary variance*

$$\epsilon_b := \left\| \begin{array}{c} \text{4x4 grid of } c, T, b \\ \text{3x3 grid of } c, T, b \end{array} \right\|^2 - \left\| \begin{array}{c} \text{3x3 grid of } c, T, b \\ \text{2x2 grid of } c, T, b \end{array} \right\|^2, \quad (4.39)$$

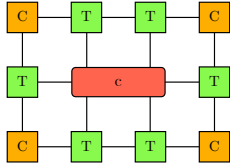
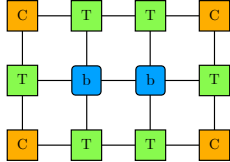
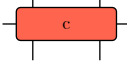
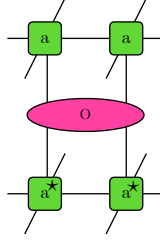
which vanishes, if the CTMRG environment $\{C, T\}$ is converged [95]. In practice, a threshold slightly above machine precision is desirable.

After the environment is converged (as measured by the boundary variance ϵ_b), expectation values can be computed. For a single-site operator O and an iPEPS state $|\psi\rangle$ with iPEPS tensor a we get

$$\langle\psi|O|\psi\rangle = \frac{\text{Diagram 1}}{\text{Diagram 2}}, \quad (4.40)$$



where b and c are given by equation (4.34b)

Since the CTMRG matrix T behaves just like an infinitely extending half column, we can insert another pair of T matrices to obtain the expectation value of a two-site operator O in an iPEPS state $|\psi\rangle$ with iPEPS tensor a :

$$\langle\psi|O|\psi\rangle = \frac{\text{Diagram 3}}{\text{Diagram 4}}, \quad \text{Diagram 5} := \text{Diagram 6}. \quad (4.41)$$





Since the normalization of the expectation values (4.40) and (4.41) is computed using the same CTMRG environment $\{C, T\}$, we do not need to keep track of multiplicative factors of C and T , since they will cancel anyway.

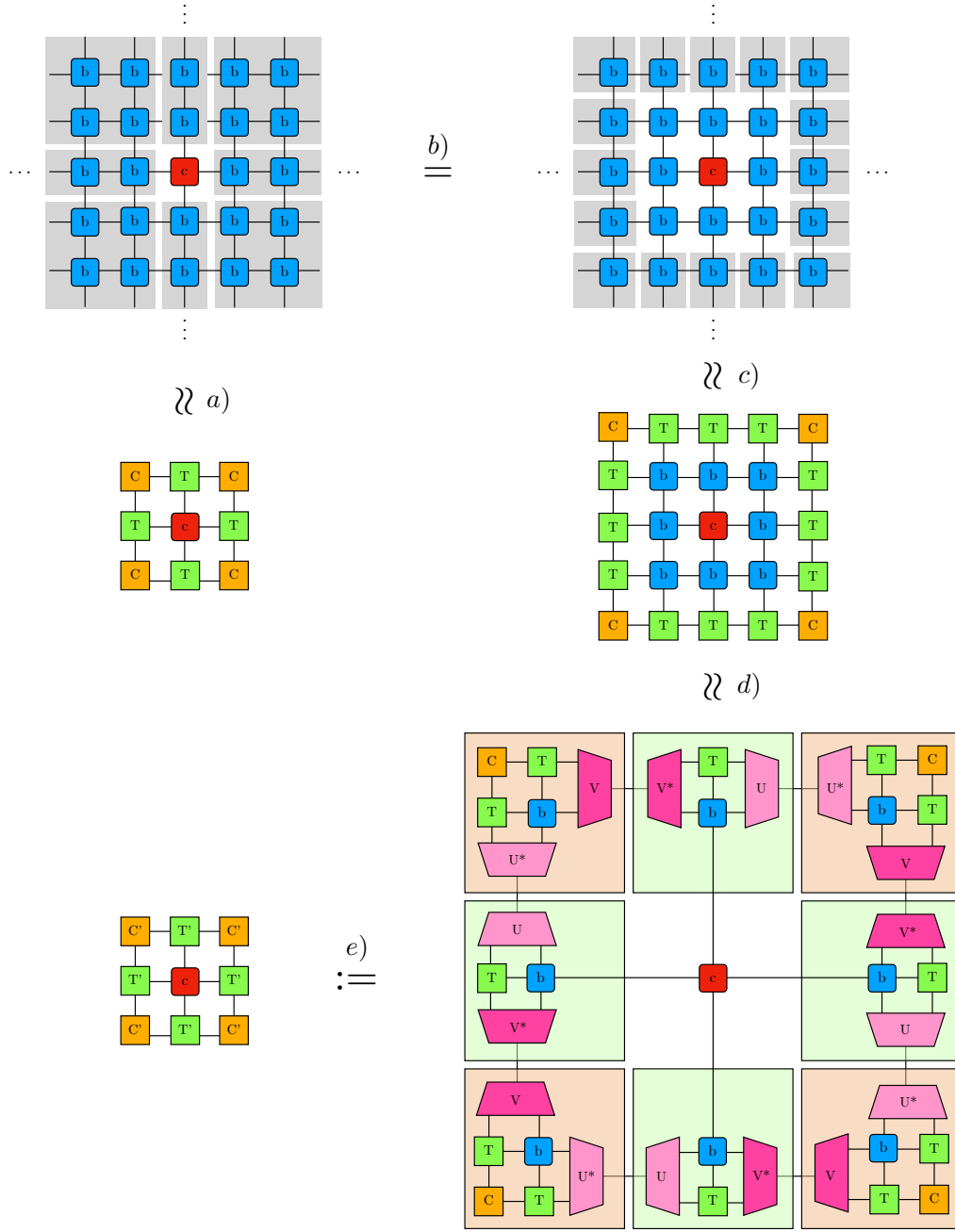


Figure 4.8: Derivation of the renormalisation procedure to find the CTMRG environment $\{C, T\}$ (see equation (4.34a)) for a translationally invariant, C_4 -symmetric iPEPS state with tensor a . The tensor b and c are given by equation (4.34b). a) The CTMRG environment approximates the infinite plane of the iPEPS diagram. b) We reframe the perspective and explicitly include more b tensors (“insertion”). c) Since C approximates a corner and T a half column of the infinite plane, they also do so for the outer corners and half columns. d) Projectors are inserted to bring the environment back to the original form and bond dimension (“renormalization”). The isometric tensors U and V are given by the singular value decomposition (4.38). e) By grouping and contracting tensors as indicated, we obtain the renormalized environment $\{\tilde{C}, \tilde{T}\}$.

A converged CTMRG environment, that correctly approximates the infinite plane will not change under the renormalization procedure $\{C, T\} \mapsto \{\tilde{C}, \tilde{T}\}$. Moreover, such an environment can be found as a fixed point, by iterating the renormalization [95, 97].

Chapter 5

Applications of Automatic Differentiation to Tensor network Methods

Contents

| | | |
|------------|---|-----------|
| 5.1 | Ground states | 76 |
| 5.2 | Time Evolution | 77 |
| 5.3 | Automatic Differentiation of PEPS Contractions | 78 |
| 5.3.1 | iPEPS Expectation Values from CTMRG | 79 |
| 5.3.2 | Finite PEPS Expectation Values using Brute-Force Contraction . | 79 |
| 5.3.3 | Finite PEPS Expectation Values using Approximate Contraction | 80 |

In this chapter we discuss applications of automatic differentiation (autodiff) in the framework of tensor network methods. This can be broadly divided into two categories.

Automatic differentiation can be used to find optimal tensor networks for a given task. We focus on two tasks, namely finding ground states and finding time evolved states. The tensor network is understood as a variational ansatz to approximate the true state that we are looking for. The best PEPS parametrization for the given task is then found by minimizing (or maximizing) an appropriate cost function, e.g. the energy of the state for finding ground states [95, 109].

Secondly, automatic differentiation can be used to obtain derivatives of tensor network contractions, when the tensors of the network are already known. For example, the free energy for classical spin systems can often be expressed as the contraction of a tensor network. Thermodynamic properties of the system can then be obtained by computing derivatives of the free energy [109].

We will focus on the first application in the scope of this thesis.

5.1 Ground states

We can find PEPS approximation of specific states in the Hilbert space (like the ground state) using numerical optimization (chapter 2) with autodiff gradients (chapter 3). In order to do that, we need to formulate the task as an optimization problem, i.e. we need to characterize the state as the minimizer of an appropriately chosen cost function.

The ground states are naturally expressed in such a way; they are the states with the lowest energy-expectation value. The ground state $|\text{GS}\rangle$ of a generic Hamiltonian H on a generic Hilbert space \mathcal{H} is the minimizer of the *variational energy* $E[|\phi\rangle] = \langle\phi|H|\phi\rangle \in \mathbb{R}$:

$$|\text{GS}\rangle = \underset{|\phi\rangle \in \mathcal{H}}{\text{argmin}} \langle\phi|H|\phi\rangle. \quad (5.1)$$

Here, the argmin is not meant strictly. If $|\phi\rangle$ has a minimal variational energy, so does $e^{i\alpha}|\phi\rangle$ with an arbitrary phase. Such phase factors, as well as possible degeneracies imply that many states have the same minimal energy. We understand argmin to mean *any* one of these states. It is an arbitrary state in the possibly degenerate ground state subspace.

Ground states of local gapped Hamiltonians can be well-approximated by PEPS states with low finite bond dimension (see subsection 4.2.3). This means there is a PEPS state $|\psi\rangle \in \text{PEPS}(\chi)$, such that $|\psi\rangle \approx |\text{GS}\rangle$. Therefore, we can obtain a PEPS state that is a good approximation for the ground state by finding the PEPS state (of a fixed finite bond dimension χ) with the lowest variational energy:

$$|\text{GS}\rangle = \underset{|\phi\rangle \in \mathcal{H}}{\text{argmin}} \langle\phi|H|\phi\rangle \approx \underset{|\phi\rangle \in \text{PEPS}(\chi)}{\text{argmin}} \langle\phi|H|\phi\rangle. \quad (5.2)$$

The PEPS state $|\phi\rangle \in \text{PEPS}(\chi)$ is parametrized by its tensors $\{a^{(x,y)}\}$, and we can find the optimal tensors that realize equation (5.2) with numerical optimization. The cost function to be minimized is the variational energy

$$E\left(\left\{a^{(x,y)}\right\}\right) = \left\langle\phi\left(\left\{a^{(x,y)}\right\}\right)\left|H\right|\phi\left(\left\{a^{(x,y)}\right\}\right)\right\rangle \rightarrow \text{MIN}. \quad (5.3)$$

In practice, the key ingredient is numerical code that evaluates the variational energy for any given PEPS state (from the PEPS tensors). We can then obtain gradients of the variational energy using automatic differentiation and minimize it using the L-BFGS algorithm (algorithm 6 for real tensors, or algorithm 7 for complex tensors).

If the Hamiltonian is real (in a given basis), the eigenstates can be chosen with real coefficients (in that basis) and therefore the tensor entries of the PEPS states can be real as well. For spin systems in the z -basis, this is the case if the Hamiltonian can be written in terms of only σ^z , σ^x and σ^\pm operators, as is the case for Ising and Heisenberg models. It has, however, proven to be beneficial to allow complex PEPS tensors in order to avoid getting stuck in local minima of the cost function¹. We observe a slightly better (that is, lower) variational energy using complex tensors.

It is important to note that we have no control over which state of the ground state subspace the optimization will converge to. If specific states of a degenerate ground state manifold are better approximated by PEPS of a finite bond dimension, the optimization will favor these states, since it can reduce the variational energy further, if it can better approximate the true ground states at this point of the variational manifold. This is the case in the TFIM which we benchmark in chapter 6. The ground states in the symmetry-broken phase is two-fold degenerate. The optimization favors the explicitly symmetry-breaking states (with maximal/minimal values of the order parameter) over the symmetrized (symmetric and anti-symmetric) states. In the limit $g = 0$, the symmetry-breaking ground states, e.g. $|\uparrow\uparrow\rangle = |\uparrow\uparrow\uparrow\dots\rangle$ can be written as PEPS with trivial bond dimension $\chi = 1$, while the same is not possible for the symmetrized states $|\uparrow\uparrow\rangle \pm |\downarrow\downarrow\rangle$.

If this behavior does not favor the desired state, or does not favor it *reliably*, the desired state can be obtained by introducing an additional term in the Hamiltonian that opens a (small) gap in the ground state manifold and let that term go to zero. In the ordered phase of the TFIM, the x + polarized state (with maximal positive order parameter) could be favored by adding a longitudinal field $-h \sum_i \sigma_i^x$ and letting $h \rightarrow 0^+$.

5.2 Time Evolution

We can also perform time evolution using the optimization method.

First, we illustrate a more general procedure for approximating any given state. We can find the best approximation of a state $|\eta\rangle \in \mathcal{H}$ by maximizing the overlap $|\langle\phi|\eta\rangle|$ of the trial state $|\phi\rangle$ with $|\eta\rangle$. In the context of time evolution, we can find an approximation to the time evolved state $|\psi(t + \delta t)\rangle$ by maximizing the overlap with $|\eta\rangle = U(\delta t)|\psi(t)\rangle$, where $U(t)$ is the time evolution operator.

If the state $|\eta\rangle$ can be exactly represented within the variational manifold, the optimization of the overlap will yield a state $|\phi\rangle$ such that $|\langle\phi|\eta\rangle| = 1$. This does, however, not mean that $|\phi\rangle = |\eta\rangle$. They could differ by a phase factor. More generally the optimized PEPS state

$$|\varphi\rangle := \operatorname{argmax}_{|\phi\rangle \in \text{PEPS}(\chi)} |\langle\phi|\eta\rangle| \quad (5.4)$$

¹Laurens Vanderstraeten, private communication

is the best approximation of $|\eta\rangle$ within the ansatz class *up to a phase*. We can recover the phase factor by considering the scalar product

$$\langle\varphi|\eta\rangle =: r e^{i\alpha}. \quad (5.5)$$

Here, $r \in [0, 1]$ is the absolute value of the scalar product (the overlap), and if $|\eta\rangle$ can be well-approximated within the variational manifold, we have $r \approx 1$ and therefore

$$|\eta\rangle \approx e^{i\alpha} |\varphi\rangle. \quad (5.6)$$

In practice, such a phase can be absorbed into a PEPS state by multiplying all entries of a single tensor with the phase factor.

We can use this procedure to approximate many types of states. The requirements are that the state should be well approximated within the ansatz class, and that scalar products with any trial state can be computed. A prominent example is, when the state of interest is obtained by applying an operator to a known state in the variational manifold.

This is the case for time evolution. In order to find the PEPS tensors $\{a^{(x,y)}\}$ that approximate $|\psi(t + \delta t)\rangle = U(\delta t) |\psi(t)\rangle$, given a PEPS for $|\psi(t)\rangle$, we maximize the overlap as the cost function

$$\Omega\left(\left\{a^{(x,y)}\right\}\right) := \left|\left\langle\phi\left(\left\{a^{(x,y)}\right\}\right)\right|U(\delta t)\left|\psi(t)\right\rangle\right| \rightarrow \text{MAX}. \quad (5.7)$$

The time evolution operator can be expressed as a PEPO, using the second-order Suzuki-Trotter decomposition, if the time step δt is small (see section 4.4). Since the time step is small, we expect that $|\psi(t + \delta t)\rangle$ is not far from $|\psi(t)\rangle$, not only in the Hilbert space, but also in terms of PEPS tensors. We expect that there is a good PEPS approximation of $|\psi(t + \delta t)\rangle$ with tensor-entries that are numerically close to those of $|\psi(t)\rangle$. Therefore, we expect the optimization to converge rather quickly, if we choose the tensors of $|\psi(t)\rangle$ as the initial guess for the optimization.

In order to perform time evolution of some initial state $|\psi_0\rangle$ to $|\psi(T)\rangle = U(T) |\psi_0\rangle$, we divide the time interval into small time-steps $t_n = n * \delta t$ with $n = 1, 2, \dots, T/\delta t$. At each n , we obtain $|\psi(t_n)\rangle$ from $|\psi(t_{n-1})\rangle$ by maximizing the overlap (equation (5.7)).

Performing time evolution in the variational manifold is only valid, if the time evolved state can be well-approximated within the ansatz class. The time evolution of interesting states² generically leads to a growth of entanglement over time. This means that after some time, the approximation of the true time evolved state in the variational manifold can become unjustified. This imposes an upper time-limit, which depends on the ability of the ansatz to represent highly entangled states. In the case of PEPS, it depends on the bond dimension.

5.3 Automatic Differentiation of PEPS Contractions

In this section, we summarize remarks about and lessons from using automatic differentiation to compute derivatives of scalar quantities that are given by contractions of PEPS

²This includes time evolution after a local or global quench. In particular time evolution of an energy-eigenstate is trivial, not interesting.

diagrams. Gradients of scalar quantities are most efficiently computed in backwards mode automatic differentiation. They therefore require VJP formulae for all involved primitives.

5.3.1 iPEPS Expectation Values from CTMRG

In order to find iPEPS ground states of local Hamiltonians, we minimize their variational energy (see section 5.1). For the optimization we need gradients of the variational energy, which is computed using the CTMRG method (see section 4.6).

Computing the energy expectation values involves finding the CTMRG environment (in particular the transfer matrices C and T) as the fixed point of a renormalization scheme (see equation (4.37)). We might be tempted to use the VJP formula (subsection C.3.1) for fixed points of iterative maps.

This is, however, invalid. The CTMRG environment has a gauge freedom on the virtual bonds. The redefined the transfer matrices C' and T'

$$\begin{array}{c} \text{C}' \end{array} \text{---} := \begin{array}{c} \text{C} \text{---} X^{-1} \\ | \\ X \end{array} \quad \begin{array}{c} \text{T}' \end{array} \text{---} := \begin{array}{c} X \\ | \\ \text{---} \text{T} \text{---} X^{-1} \\ | \end{array} \quad (5.8)$$

with an invertible matrix X , leave the contraction of the environment unchanged:

$$\begin{array}{ccc} \text{C}' & \text{T}' & \text{C}' \\ | & | & | \\ \text{T}' & & \text{T}' \\ | & | & | \\ \text{C}' & \text{T}' & \text{C}' \end{array} = \begin{array}{ccc} \text{C} & \text{T} & \text{C} \\ | & | & | \\ \text{T} & & \text{T} \\ | & | & | \\ \text{C} & \text{T} & \text{C} \end{array} . \quad (5.9)$$

The CTMRG renormalization procedure can only ever converge up to such gauge transformations. That is, even for a converged environment, additional renormalization steps can act as a gauge transformation (5.8). While this would leave expectation values unchanged, it implies that the tensors C and T do not necessarily converge *individually*. Hence, the VJP formula for fixed points is not applicable.

This problem could be remedied by fixing the gauge of the CTMRG environment.

5.3.2 Finite PEPS Expectation Values using Brute-Force Contraction

Contracting expectation values (or matrix elements) in PEPS states exactly (“brute-force”) is discussed in section 5.3.2. Evaluating them only involves repeatedly applying tensor contractions (e.g. in `python jax.numpy.tensordot`). There are caveats when gradients of these expectation values w.r.t. the PEPS tensors are to be computed.

Since all intermediate primal values need to be stored in memory in backwards mode autodiff, the gradient computation is expensive in terms of RAM. We observe memory

demands of $\mathcal{O}(200 \text{ GB})$ for evaluating the gradient of a matrix element on a 7×7 system with PEPS of bond dimension $\chi = 2$ and PEPOs of bond dimension $\eta = 2$. This can be remedied by using checkpointing [110]. Checkpointing is currently not a native feature of `jax` and was not tested in the scope of this thesis.

Additionally, the contractions are also expensive in terms of CPU time. It scales exponentially in the (linear) system size. Therefore, approximate contraction methods are desirable.

5.3.3 Finite PEPS Expectation Values using Approximate Contraction

In principle, code for approximate PEPS contractions can be fed to automatic differentiation and derivatives can be computed. However, this would violate a paradigm of numerical differentiation: *Thou shalt not differentiate approximations, thou shalt instead approximate the derivatives* [111]. Derivatives are naturally more sensitive to errors introduced by the approximation.

Additionally, using the naive gradients (differentiating the approximation) to optimize ground states will minimize the energy *as given by the approximate contraction*. In particular, the optimization will favor states, which give a low energy, because the approximations in the contractions are bad. We observed energies that are well below³ semi-exact values (exact diagonalization or DMRG methods) when optimizing ground states with naive gradients of the boundary MPS contraction method (see section 4.5.2). The optimization has found a state whose energy expectation value is badly approximated by the bMPS method, i.e. it found a way to cheat the approximate contraction.

This problem can be remedied by approximating the derivatives, e.g. using the VJP formula for tensor network contractions discussed in subsection C.3.3. The VJP of a contracted network is expressed using other network contractions of a similar structure. Those can again be performed using an approximate contraction method. For the bMPS method, this computation can be done quite efficiently, since the bMPS columns found in previous diagrams can be reused in subsequent diagrams.

³by $\mathcal{O}(10\%)$

Chapter 6

Benchmark: Transverse Field Ising Model on the 2D Square Lattice

Contents

| | | |
|------------|---|-----------|
| 6.1 | The Model and its Phase Diagram | 82 |
| 6.1.1 | Z2 Symmetry | 82 |
| 6.1.2 | Phase Diagram | 83 |
| 6.2 | Ground states on the Infinite Square Lattice | 86 |
| 6.3 | Ground states on Finite Square Lattices | 89 |
| 6.4 | Quench Dynamics | 92 |

In order to benchmark the methods discussed in chapter 5, we apply them to the transverse field Ising model on a 2D square lattice.

We first introduce the model, and discuss (qualitatively) its phase diagram in section 6.1. In section 6.2, we use gradient-based optimization of iPEPS states to find the ground state of the TFIM on an infinite square lattice. We confirm the validity of the approach by studying the phase diagram. In particular we extract the critical point from the spontaneous magnetization close to the phase transition and compare to other numerical works. In section 6.3, we optimize PEPS states to find the ground state for finite square lattices. We compare the error in the achieved variational energy to other PEPS methods and observe an improvement of roughly three orders of magnitude at the same bond dimensions, albeit at higher computational cost. Finally, we use gradient-based optimization to perform time evolution after a local quench (section 6.4). We confirm that the evolution of local observables as well as the dynamical structure factor¹ matches with exact diagonalization data.

6.1 The Model and its Phase Diagram

In this section, we introduce the quantum transverse field Ising model (TFIM). We consider the TFIM on a 2D square lattice. A spin-one-half degree of freedom is located at every site of the lattice and the Hamiltonian of the TFIM is given by

$$H = -J \sum_{\langle i,j \rangle} \sigma_i^x \sigma_j^x - gJ \sum_i \sigma_i^z, \quad (6.1)$$

where $J > 0$ is an energy scale, which we set to $J = 1$ by rescaling the units of energy. The first term models interactions of nearest neighbor spins at sites i and j . The second term models the interaction of the spins with an external field with the (dimensionless) relative field strength $g > 0$. While the spin-spin interactions are modelled as interactions only with the x components of the spins, the field is in the z direction, hence it is “transverse”.

We write σ_i^α with $\alpha \in \{x, y, z\}$ for the Pauli operators. They are dimensionless operators with eigenvalues ± 1 , which are related to the spin operators by a factor $\hbar/2$.

We write $|\uparrow\rangle$ and $|\downarrow\rangle$ for the eigenstates of σ^z , with eigenvalues $+1$ and -1 , respectively. These two states form a basis for the on-site Hilbert space, and we use it as the preferred basis for numerics. For the eigenstates of σ^x we write $|x+\rangle = |\rightarrow\rangle$ and $|x-\rangle = |\leftarrow\rangle$, such that $\sigma^x |x\pm\rangle = \pm |x\pm\rangle$.

6.1.1 \mathbb{Z}_2 Symmetry

The TFIM has a \mathbb{Z}_2 symmetry. Intuitively, its is given by a simultaneous spin-flip on all sites. The flip is along the x axis, essentially $|x+\rangle \leftrightarrow |x-\rangle$.

Formally this is the action of σ^z at every site, and the symmetry is expressed as an operator Σ that commutes with the Hamiltonian

$$\Sigma := \prod_i \sigma_i^z \quad ; \quad [H, \Sigma] = 0. \quad (6.2)$$

¹Which we extract from time-dependent correlation functions

The symmetry Σ is idempotent $\Sigma^2 = \mathbb{1}$. Therefore, its eigenvalues square to one (they are either $+1$ or -1). It is a \mathbb{Z}_2 symmetry in the sense that Σ is the generator of $\{\mathbb{1}, \Sigma\} \cong \mathbb{Z}_2$ as a multiplicative group.

We call states *symmetric*, if they are invariant under the action of the symmetry, i.e. $|\psi\rangle$ is symmetric if $\Sigma|\psi\rangle = |\psi\rangle$, or in other words, if it is an eigenstate of Σ with eigenvalue 1 . We call eigenstates of Σ with eigenvalue -1 *anti-symmetric*.

Since the symmetry commutes with the Hamiltonian, all eigenstates of H can be chosen as simultaneous eigenstates of Σ .

If the (possibly degenerate) ground state eigenspace of the Hamiltonian contains only symmetric states (with eigenvalue $+1$ of Σ), the symmetry of the Hamiltonian is *preserved* (by the ground state(s)). If there are only antisymmetric states, then they all preserve the symmetry $-\Sigma$.

If there are both symmetric and anti-symmetric ground states, the symmetry is *spontaneously broken* (by the ground state(s)). From a symmetric ground state $|\text{GS}+\rangle$ and an anti-symmetric $|\text{GS}-\rangle$, we can construct two states

$$|\gamma_{\pm}\rangle := 1/\sqrt{2}(|\text{GS}+\rangle \pm |\text{GS}-\rangle) \quad (6.3)$$

that explicitly break the symmetry. They are not eigenstates of the symmetry, but the action of the symmetry transforms one into the other

$$\Sigma|\gamma_{\pm}\rangle = |\gamma_{\mp}\rangle. \quad (6.4)$$

Any slight perturbation, e.g. an external field with an x -component will favor one of the symmetry breaking states (6.3).

6.1.2 Phase Diagram

In order to understand the phase diagram of the TFIM as the field strength g is tuned, let us first consider the extreme cases $g = 0$ and $g \rightarrow \infty$.

At zero field, the Hamiltonian (6.1) only consists of the ferromagnetic interaction term. Both fully x -polarized states

$$|\Rightarrow\rangle := \bigotimes_i |\rightarrow\rangle_i \quad ; \quad |\Leftarrow\rangle := \bigotimes_i |\leftarrow\rangle_i \quad (6.5)$$

are ground states in this limit. The ground state subspace is two-fold degenerate and spanned by $|\Rightarrow\rangle$ and $|\Leftarrow\rangle$. They have an energy of $-2N$ where N is the number of sites and spontaneously break the symmetry (6.2) ($\Sigma|\Rightarrow\rangle = |\Leftarrow\rangle$ and vice versa).

For dominant field strengths $g \rightarrow \infty$, the interaction term is negligible and the behavior is determined only by the field-term. It has the non-degenerate ground state

$$|\uparrow\rangle := \bigotimes_i |\uparrow\rangle_i, \quad (6.6)$$

which has an energy of $-gN$ and conserves the symmetry Σ .

An order-parameter that detects the breaking of the symmetry (6.2) is the (average) magnetization m along the x -axis

$$\phi := 1/N \sum_i \sigma_i^x, \quad (6.7)$$

which anticommutes with the Hamiltonian:

$$\{\phi, \Sigma\} = \frac{1}{N} \sum_i \left\{ \sigma_i^x, \prod_j \sigma_j^z \right\} = \frac{1}{N} \sum_i \left(\{\sigma_i^x, \sigma_i^z\} \prod_{j \neq i} \sigma_j^z - \sigma_i^z \left[\sigma_i^x, \prod_{j \neq i} \sigma_j^z \right] \right) = 0. \quad (6.8)$$

The magnetization m is the expectation value of the operator ϕ

$$m := \langle \psi | \phi | \psi \rangle \begin{cases} = 0 & \Sigma |\psi\rangle = |\psi\rangle \text{ (symmetric)} \\ \in [-1, 1] & \text{in general} \end{cases}, \quad (6.9)$$

where N is the number of spins (= number of sites) in the system. It vanishes for symmetric states².

The order parameter is $m(g=0) = \pm 1$ and $m(g \rightarrow \infty) = 0$ in the ground states (6.5) and (6.6) of the extreme cases.

Using perturbation theory and a low- g or high- g expansion [14, chpt 1.4.1], we can conclude that a symmetry-breaking phase with non-zero order parameter extends from $g=0$ to small finite g . This is the *ordered* (ferromagnetic) phase. A symmetric phase with vanishing order parameter extends from $g \rightarrow \infty$ to large finite g . This is the *disordered* (paramagnetic) phase.

From Landau Ginzburg theory we know that there is a phase transition that separates the two phases³ of different symmetry at some finite g_c . In the thermodynamic limit, the order parameter approaches zero with a universal power law as the critical point is reached from below ($g \nearrow g_c$), and vanishes for all $g > g_c$:

$$\text{for } g \approx g_c \quad : \quad m(g) \sim \begin{cases} |g - g_c|^\beta & g \leq g_c \\ 0 & g > g_c \end{cases} \quad (6.10)$$

with the critical exponent β . The order parameter is (as a function of g) continuous everywhere, but not differentiable at g_c .

We summarize the phase diagram in figure 6.1.

²For a symmetric state $|\psi\rangle$ we have $m = \langle \psi | \phi | \psi \rangle = \langle \psi | \phi \Sigma | \psi \rangle = -\langle \psi | \Sigma \phi | \psi \rangle = -\langle \psi | \phi | \psi \rangle = -m$ and therefore $m = 0$.

³The fact that only the two phases discussed above exist, is an observation that can not be concluded from the symmetry-behavior of the two extreme cases alone.

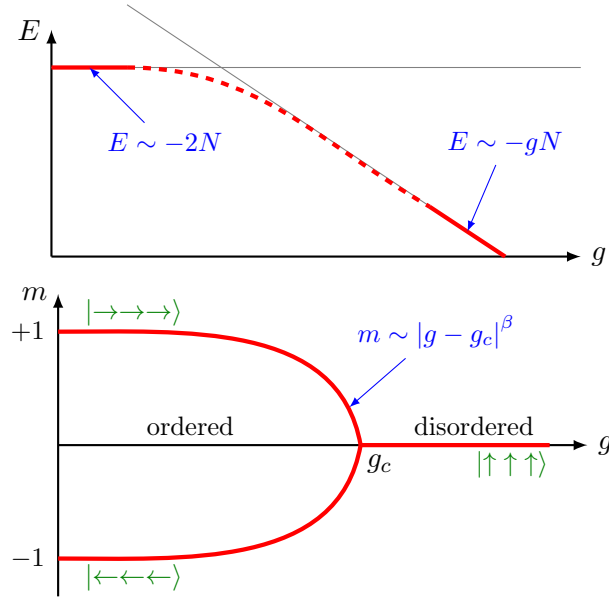


Figure 6.1: Qualitative phase diagram of the transverse field Ising model (TFIM) as the relative field strength g is varied. For $g = 0$ the two degenerate ground states are the x -polarized states. Their energy is $-2N$ where N is the number of sites on the lattice. They have an average x -magnetization (order parameter) m of ± 1 . Extending to finite $g < g_c$ is the ferromagnetic phase with a doubly degenerate ground state and non-zero magnetization. The global spin-flip symmetry Σ is spontaneously broken in this phase. At $g \rightarrow \infty$ the ground state is the non-degenerate z -polarized state. Its energy is $-gN$ and its magnetization vanishes. Extending to finite $g > g_c$ is the paramagnetic phase, where the ground state is still unique and the magnetization vanishes. The symmetry Σ is preserved in this phase. Between the phases is the critical point at $g_c \approx 3.05$ with universal scaling properties (see main text).

6.2 Ground states on the Infinite Square Lattice

Using optimization with autodiff gradients to find iPEPS approximations of ground states with translational invariance (trivial one-site unit cell) or a small unit cell has been proven to be successful in previous works [95, 109].

In this section, we show as a proof of principle that the ground state of the TFIM on the infinite 2D lattice can be well approximated using the autodiff method.

We optimize an iPEPS for different bond dimensions $\chi = 2, 3$, while sweeping the relative field-strength g . The cost function to be optimized is the energy expectation value. Its computation is discussed in section 4.6. We let the `python` package `jax` handle the computation of gradients and find the iPEPS tensor, which minimizes the variational energy using the L-BFGS algorithm (algorithm 6). Since the Hamiltonian is real (in the z -basis, which we use for computations), we can restrict the iPEPS tensors to be real. Since the Hamiltonian possesses this symmetry, we choose a translationally invariant, C_4v -symmetric iPEPS. While we can not know a priori that the symmetry is not spontaneously broken by the ground state, numerical evidence shows that it is not [112].

We qualitatively reproduce the expectations of the phase diagram (figure 6.1) in figure 6.2 and for a more quantitative analysis, obtain the critical point g_c (see equation (6.10)) from the ground state magnetization $m(g)$ in figure 6.3.

We find good agreement with existing numerical work and conclude that autodiff optimization is a viable tool to obtain iPEPS ground state optimization of local models.

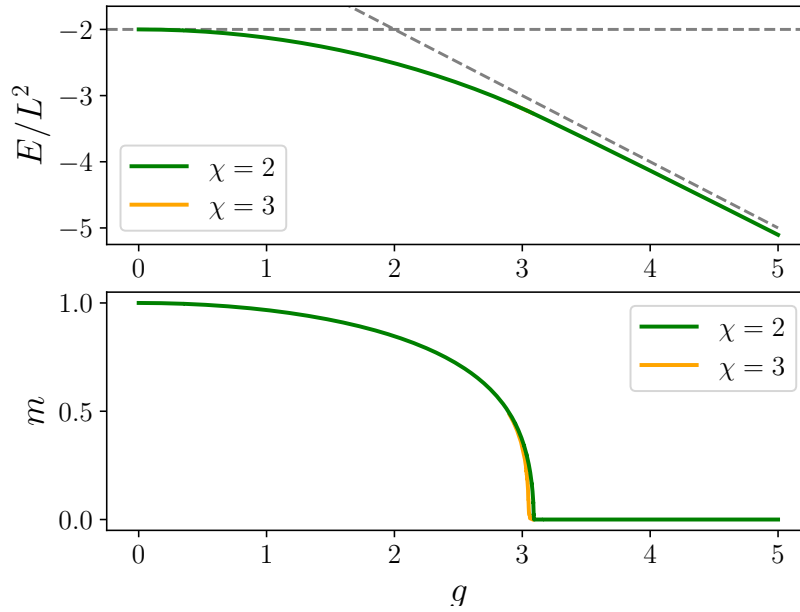


Figure 6.2: Variational energy and magnetization (order parameter) for the optimized iPEPS ground states of the TFIM on the infinite square lattice. The energy of an iPEPS state is computed via the CTMRG method (section 4.6) and the ground state is found by optimizing the iPEPS tensors using autodiff gradients of the variational energy (see section 5.1) with a CTMRG bond dimension of $D_{opt} = 50$. The observables (energy and magnetization) are evaluated in the variational ground state with a CTMRG bond dimension of $D_{obs} = 100$. Results for iPEPS of different bond dimensions χ are shown. The difference between them is barely visible at this scale. It is more pronounced in figure 6.3, where we analyze the magnetization m around the critical point $g_c \approx 3.05$. The dashed lines are the asymptotical energies $E(0) = -2N$ and $E(g \rightarrow \infty) = -gN$.

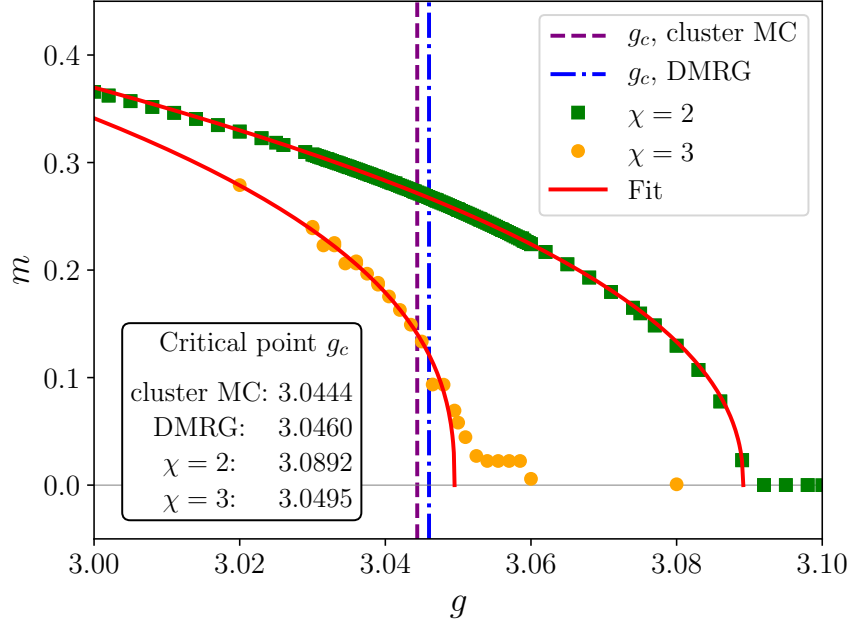


Figure 6.3: Magnetization of the variational ground state (see figure 6.2 for details) close to the critical point $g_c \approx 3.05$. The vertical lines are values for the critical point obtained by different numerical methods, namely Density Matrix Renormalization Group (DMRG) [112] and cluster Monte Carlo (cluster MC) [113]. As is expected, given the diverging correlation length, the difference between iPEPS bond dimensions is most pronounced around the critical point. Low- χ iPEPS are particularly bad approximations of the true ground states in this region of the phase diagram. We fit a power law (equation (6.10)) to the magnetization curves to obtain the critical point g_c . We find good agreement, even at the low bond dimensions available. The optimization for $\chi = 3$ around $g \in [3.05, 3.06]$ is not fully converged. Since iPEPS optimisation is not the main focus of this thesis, we did not exhaust the computational efficiency and available resources. IPEPS of bond dimension up to $\chi = 7$ have been optimized using automatic differentiation in previous works [95].

6.3 Ground states on Finite Square Lattices

Contrary to the iPEPS discussed in section 6.2, we are unaware⁴ of any published works applying autodiff-based optimization to find PEPS ground states for finite systems.

We suspect that this is, in part, due to the increased computational complexity of evaluating the variational energy. This complexity arises from the extensive number of terms in the Hamiltonian, which unlike in the translationally invariant iPEPS can not be reduced to $\mathcal{O}(1)$ many terms. We have managed to overcome this problem by writing the Hamiltonian as a PEPO. Additionally, exact PEPS contractions are expensive. It is therefore necessary to find a way to make approximate contraction methods perform well with automatic differentiation. We did not manage to achieve this in the scope of this thesis but discuss strategies in the outlook (chapter 7).

The numerical simulations we perform are to be understood as a proof-of-principle, that autodiff-based optimization is a viable method for finding ground states.

We find ground states of the TFIM on $L \times L$ finite 2D square lattices. The cost function to be minimized is the variational energy. We express the Hamiltonian as the sum of two PEPOs (see section 4.3) and evaluate its expectation value in the trial states by exactly contracting the entire tensor network (see section 4.5.1). We find the ground state (or rather its complex-valued PEPS tensors) by minimizing the cost function (algorithm 6).

We benchmark the result by analyzing the energy error $\Delta E = E_{\text{opt}} - E_0$ where E_{opt} is the variational energy of the approximate ground state we find with autodiff-optimization and E_0 is a semi-exact (up to numerical precision) value. For system sizes up to 5×5 , we obtain E_0 from exact diagonalization and for larger system sizes, we use a DMRG algorithm of an MPS with large bond dimension (up to $\mathcal{O}(1000)$ at the center bonds) that tiles the 2D lattice in a snake-geometry. For the latter, the built-in functionality of the `python` package TenPy [98] was used.

The convergence of the variational energy during the optimization is depicted in figure 6.4. We compare the energy error ΔE to other PEPS methods in figure 6.5.

We conclude that for a given bond dimension, the autodiff-optimized PEPS performs better than ground states obtained with other PEPS methods. This is expected, since the optimization (if it converges) yields the best⁵ possible PEPS approximation of the true ground state. However, for a given system size and bond dimension, the autodiff-optimization is more expensive. Only comparatively small systems and low bond dimensions are accessible using brute-force contraction of the tensor network that forms the cost function. In order to be competitive with other PEPS methods, approximate contraction methods will have to be incorporated, as discussed in section 5.3.

⁴at the time of writing

⁵As measured by the cost function. In this case, it yields the PEPS with the lowest possible energy for the given bond dimension

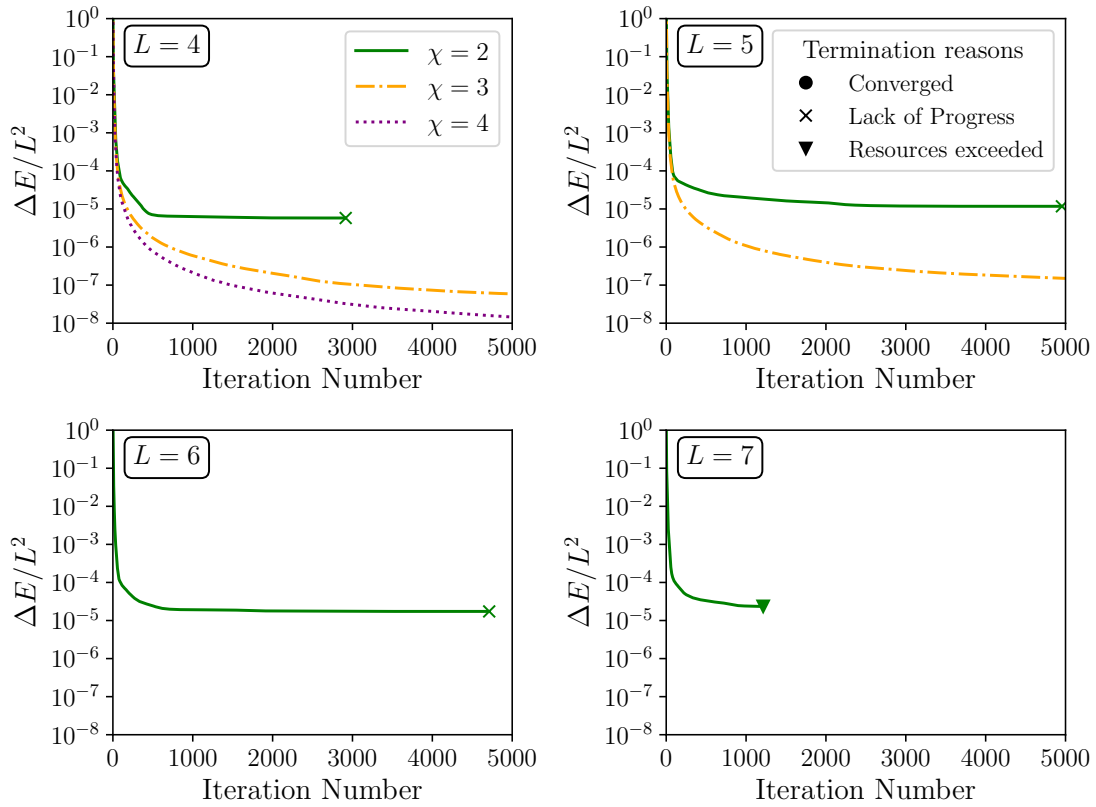


Figure 6.4: Progress of the optimization algorithm for finding the variational PEPS ground state of the transverse field Ising model (TFIM) on a $L \times L$ square lattice.

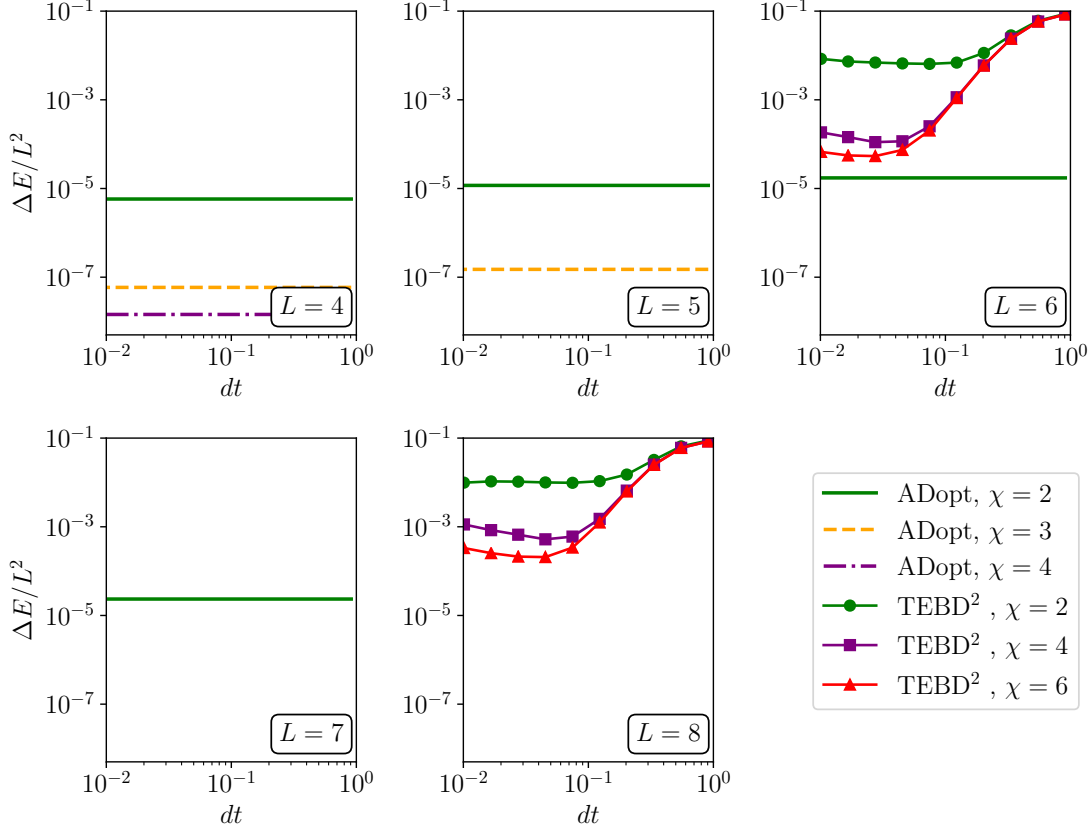


Figure 6.5: Comparison of optimized PEPS ground states of the transverse field Ising model (TFIM) on a $L \times L$ square lattice at $g = 3.5$ to other PEPS methods. The error in the variational energy $\Delta E = E_{\text{var}} - E_0$ is computed relative to a semi-exact value E_0 (exact diagonalization up to $L = 5$, DMRG for larger systems, see main text). We compare our results (ADopt) to the TEBD² algorithm [114]. The time step dt on the horizontal axis is only relevant for the TEBD² data, where it is the time step of the trotterized imaginary time evolution used to find the ground state. At the same PEPS bond dimension we find variational energies that are better by roughly three orders of magnitude. This comes, however, at a higher computational cost. Reaching larger system sizes and higher bond dimensions will require incorporating approximate PEPS contractions (discussed in section 5.3 and chapter 7).

6.4 Quench Dynamics

We can also apply the autodiff-optimization method to perform time evolution in finite systems. As in the previous section we are unaware of any published work applying the method to time evolution. In addition to the increased computational complexity of contracting the tensor network that forms the cost function, time evolution requires complex-valued PEPS tensors. This in turn, makes both complex autodiff as well as a complex optimization algorithm necessary. Many autodiff tools do not support complex numbers. Additionally, every single time step requires optimizing a PEPS state. This makes the time evolution quite expensive.

The numerical simulations in this section are to be understood as a proof-of-principle. In order to make the method competitive to alternative algorithms for PEPS time evolution, approximate contraction algorithms will need to be incorporated. Using brute-force contraction, we show that time evolution can be done using autodiff-optimization.

We simulate the time evolution of the TFIM in the paramagnetic phase (at $g = 5$) after a local quench at the central site of the finite square lattice. We consider a system of size 5×5 , since it is the largest system for which we can perform a significant number of time steps using brute-force contraction. The cost of a single time step is comparable to finding the ground state for the given system, see section 6.3. Larger systems are therefore theoretically accessible, but only a few time steps could be done.

We start the time evolution with the quenched state

$$|\psi(t=0)\rangle := \sigma_c^x |\text{GS}\rangle, \quad (6.11)$$

where c is the central site of the lattice and $|\text{GS}\rangle$ is the ground state of the TFIM at $g = 5$, which we also obtain using autodiff-optimization (analogous to section 6.3). Since the ground state this far in the paramagnetic phase is almost $z+$ polarized, the quench intuitively acts as a spin-flip and excites a spin-wave.

We then time evolve it using the TFIM Hamiltonian H , again at $g = 5$:

$$|\psi(t)\rangle = e^{-iHt} |\psi(t=0)\rangle = e^{-iHt} \sigma_c^x |\text{GS}\rangle. \quad (6.12)$$

We divide the time evolution into small time-steps of size δt , such that we can write the time evolution operator $U(\delta t)$ as a PEPO using Suzuki-Trotter decomposition (see section 4.4). For each time-step we perform a separate optimization. The cost function to be maximized is the overlap of the trial state, with the time evolution operator applied to the previous state (see section 5.2), which we compute using brute-force contraction (see section 4.5.1).

Our goal is to compare relevant time-dependent quantities to exact diagonalization data in order to confirm that the time evolution is valid.

At each time step we evaluate the expectation value

$$\langle \psi(t) | \sigma_i^z | \psi(t) \rangle \quad (6.13)$$

for all lattice sites i . We compare the result to exact diagonalization (ED) in figure 6.6.

Additionally, we obtain the time-dependent correlation function

$$C^{xx}(\mathbf{r}_i, t) := \langle \text{GS} | \sigma_i^x(t) \sigma_c^x | \text{GS} \rangle = e^{iE_0 t} \langle \text{GS} | \sigma_i^x | \psi(t) \rangle \quad (6.14)$$

from the matrix element on the RHS, where c is the central site of the lattice and E_0 is the energy of the ground state. This is an example application which allows insight into the excitation spectrum of the model and is directly probed by neutron scattering. We compare the correlation function $C^{xx}(\mathbf{r}_i, t)$ to ED data in figure 6.7.

From the correlation function we extract the dynamic (spin) structure factor (DSF)

$$S^{xx}(\mathbf{k}, \omega) := \sum_i e^{-i\mathbf{k} \cdot \mathbf{r}_i} \frac{1}{2\pi} \int_{-\infty}^{\infty} dt e^{i\omega t} C^{xx}(\mathbf{r}_i, t) \quad (6.15)$$

and compare to ED in figure 6.8. We explain the details of performing the time-to-frequency Fourier transform in appendix A.3.

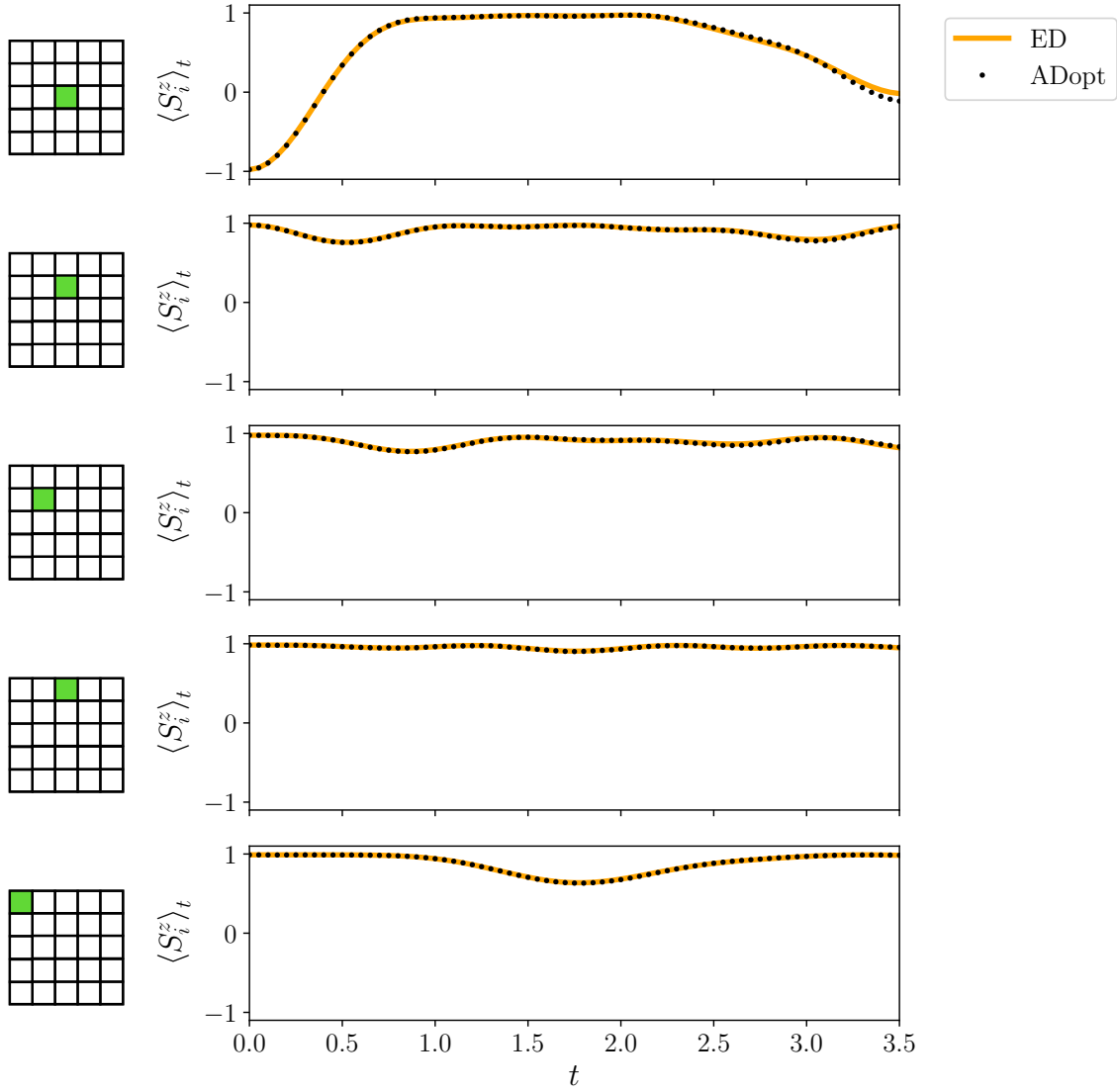


Figure 6.6: Dynamics of the local observable $\langle \sigma_i^z \rangle_t := \langle \psi(t) | \sigma_i^z | \psi(t) \rangle$ after a local quench (see main text). We compare our method (ADopt) for time evolution to exact diagonalization (ED) data obtained with `scipy` for `python`. For our method, a PEPS of bond dimension $\chi = 2$ is optimized at every time step (see main text). The position of the lattice site i in the 5×5 square lattice is depicted in the left column. We see exact agreement up to a time of $t \sim 3$, where slight deviations at the center site (top graph) start to manifest. Due to growth of entanglement after the quench, we expect the approximation of the true-time evolved state as a PEPS of such low bond dimension to fail after some time. Additionally, the optimization was terminated after 500 iterations due to limited computational resources. Any errors introduced by these two effects accumulate, since the approximate time evolved state is used for the respective next time step. For higher bond dimensions and more iterations of the optimization algorithm, we expect to see correct results up to substantially longer time scales. In order to deal with the increased computational cost, and to reach larger system sizes, approximate contraction methods will have to be incorporated.

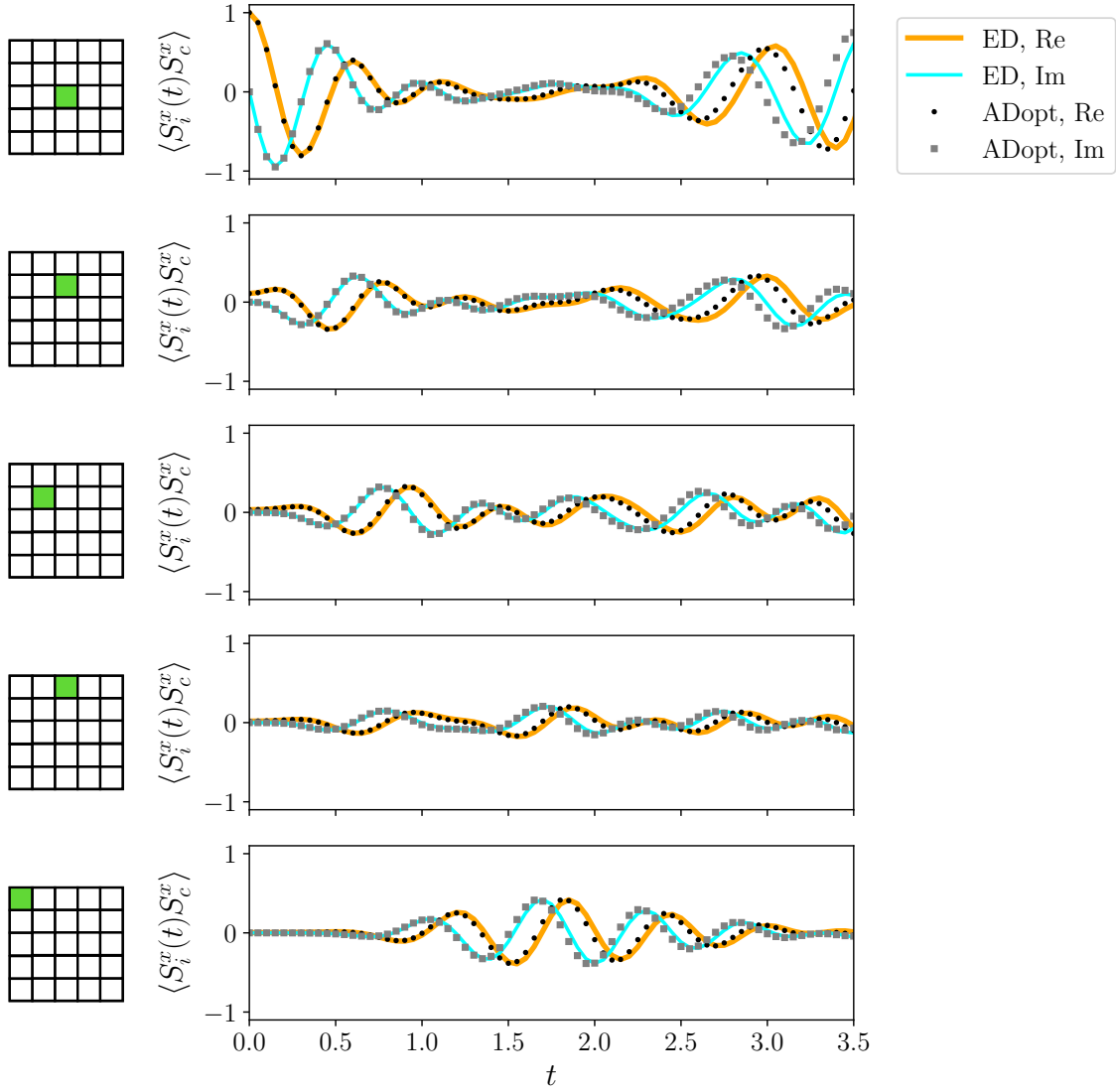


Figure 6.7: Dynamical spin-spin correlation function $C^{xx}(\mathbf{r}_i, t) = \langle \sigma_i^x(t) \sigma_c^x \rangle$ obtained from time evolution after a local quench (see main text). We compare our method (ADopt) for time evolution to exact diagonalization (ED) data obtained with `scipy` for `python`. We plot the real (Re) and imaginary (Im) part of $C^{xx}(\mathbf{r}_i, t)$. For our method, a PEPS of bond dimension $\chi = 2$ is optimized at every time step (see main text). The position of the lattice site i in the 5×5 square lattice is depicted in the left column. We see good agreement up to a time of $t \sim 2.5$, where deviations at bulk sites start to manifest. See the discussion under figure 6.6.

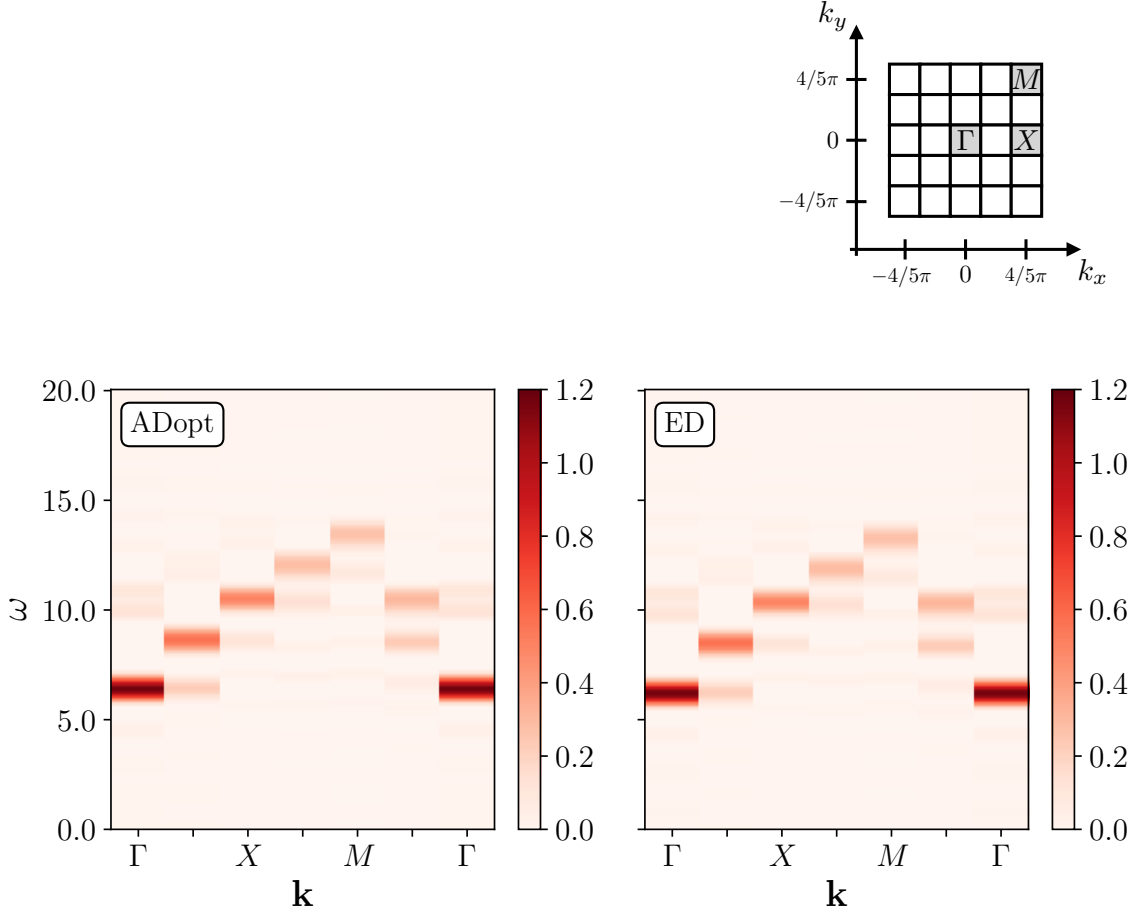


Figure 6.8: Dynamical (spin) structure factor $S^{xx}(\mathbf{k}, \omega)$ (see equation (6.15)), obtained from the correlation function depicted in figure 6.7. We compare our method (ADopt) for time evolution to exact diagonalization (ED) data obtained with `scipy` for `python`. We use linear predictions [115, 116] and a Gaussian envelope to avoid artifacts due to the time-cutoff in the correlation function data. The Gaussian envelope of width $\sigma_t \sim 3.54$ causes a Gaussian broadening of $\sigma_\omega \sim 0.28$ in frequency space. For details about the numerics of performing the Fourier transform from real time to frequency space see appendix A.3. For better comparison, the correlation function data from exact diagonalisation was only used at the same time steps and to the same cutoff as the ADopt data. We see agreement in all features. We interpret the brightest features at each \mathbf{k} as physical. We expect them to be sharp (delta peaks) in the thermodynamic limit. All other features can be interpreted as ringing artifacts introduced by the finite-time cutoff. Getting rid of these artifacts will require larger system sizes and longer times, both of which we expect to achieve once approximate contraction methods for autodiff optimisation are implemented.

Chapter 7

Conclusion and Outlook

We have investigated gradient-based optimization methods with gradients from automatic differentiation (autodiff) for PEPS tensor network states on finite and infinite lattices.

Optimizing iPEPS for ground states of the transverse field Ising model (TFIM) on the infinite 2D square lattice, we have confirmed the validity of our approach by reproducing the phase diagram with low bond dimension iPEPS. While we did not refine the computation enough to reach bond dimensions as high as other works [95, 109], we confirm that the method, and in particular the optimization algorithms we have developed are valid.

The main results of this thesis are proofs of principle that autodiff optimized PEPS for ground states *and* time evolution on finite systems show promising success.

First, we showed that the variational energies obtained for the autodiff optimized ground states of the TFIM on finite lattices are significantly better (that is, lower) than those obtained with other PEPS methods. This is made possible by the PEPO representation of the Hamiltonian that we have developed. Our investigation was limited to brute-force contractions of the PEPS expectation values that constitute the cost function to be optimized (here the variational energy). The high computational cost of this contraction method heavily limits the achievable system sizes and PEPS bond dimensions. The good results we see in the comparatively small systems, however, lead us to expect similar success in larger systems, when approximate contraction methods are incorporated.

We find that established PEPS methods do not fully exhaust the variational power of the PEPS ansatz. The autodiff optimized ground states achieved significantly better variational energies at the same bond dimensions. They are the best possible PEPS approximations¹ of the true ground state (given that the optimization converges) and can possibly allow further insight into the variational power of PEPS states.

Secondly, we showed that time evolution after a local quench from the ground state of the TFIM can be simulated with PEPS bond dimension as low as $\chi = 2$, to good agreement with exact diagonalization data on a 5×5 system. We see exact agreement at early times and only slight deviations at later times. We attribute the deviations to the approximation of the true time evolved state as a low bond dimension PEPS breaking down due to entanglement growth after the quench, as well as accumulated error from the optimization

¹They are the best PEPS of a given bond dimension, as measured by the cost function. I.e. they have the lowest possible variational energy.

algorithm. It was terminated before full convergence, due to the high computational cost of the brute-force contraction method. We can expect both problems to be remedied when approximate contraction methods are incorporated, allowing PEPS of higher bond dimensions to be optimized as well as making the individual optimization steps less expensive. Additionally, larger system sizes will be achievable.

The main challenge when using approximate contraction methods, is that they can not be directly fed into an autodiff optimization algorithm. With such a naive approach, the optimization would favor states which have a lower variational energy, not because they are good ground states, but because their energy is badly approximated (*underestimated*) by the contraction method. This can be remedied by implementing an approximate formula for the exact derivative, in particular the VJP of the tensor network contraction, instead of differentiating the approximation. For further discussion see section 5.3.

Further potential for improvement of the numerical method includes

1. The initial guesses for the optimization method warrant further investigation. For optimizing ground states, we found no significant dependence on the initial guess. Bad initial guesses only set the optimization behind by ~ 10 iterations of the optimization algorithm and will eventually converge just as well. For time evolution, however, every little advantage counts since it will affect the computational budget at every single time step. Viable options for initial states for the optimization of $|\psi(t + \delta t)\rangle$ include the old state $|\psi(t)\rangle$ (which we have used in this work), absorbing only the on-site part of the time evolution $e^{-iH_{\text{site}}\delta t} |\psi(t)\rangle$ (which can be done exactly on the level of PEPS tensors), or even any other method for PEPS time evolution, such as TEBD² [114].
2. The details of the optimization algorithm warrant further investigation. While the L-BFGS algorithm is generally believed to be the best all-purpose tool in light high-dimensional problems and expensive gradients, it is not clear that it is the best algorithm for this specific task. Additionally, a detailed investigation of the optimization parameters could yield further improvement in computational cost.
3. If the gauge freedom on the virtual bonds of the CTMRG environments can be fixed, gradients of iPEPS expectation values can be computed efficiently using the VJP formula for fixed points of iterative maps. See the discussion in section 5.3.1.
4. Making full use of the `jax` backend, which supports GPU acceleration and specific hardware (TPUs, “Tensor Processing Units”).
5. Checkpointing to reduce the memory cost of gradients, if the available RAM is limited. Checkpointing [110] is a common technique for autodiff in backwards mode. Not all intermediate primals are stored in memory, but only some at crucial “checkpoints” of the computation graph. The other primals are then recomputed on demand, effectively trading some CPU time for memory.

Future applications of autodiff optimization methods include, but are not limited to

1. Thermal states. The time evolution method can also be used to perform imaginary time evolution, giving access to thermal states, by time evolving an infinite-temperature purification to finite T .
2. Other models. On infinite systems, ground states of any model with local interactions can be optimized. The CTMRG algorithm that finds the converged environment only

depends on the chosen unit cell of the state, and operators with terms of any local support can be computed from it. For efficient contractions of the cost function on finite systems, we need PEPO representations of the Hamiltonian (for ground states) and/or the time evolution operator (for time evolution). We have introduced a PEPO representation for any nearest-neighbor model. We expect that a similar approach is possible for next-nearest-neighbor interactions. A natural next step would be to investigate frustrated quantum magnets, such as the $J_1 - J_2$ model, as well as implementing fermionic degrees of freedom.

3. Tensor network states on other lattice geometries can be studied using autodiff optimization, as long as the relevant cost functions can be evaluated efficiently using the primitives of the autodiff package.
4. Other classes of variational ansatz. Under the same conditions, the autodiff optimization method can be applied to find the optimal realization of any variational ansatz in quantum mechanical problems. This includes higher dimensional tensor product states or even parametrised electronic wave functions of complex molecules.

Acknowledgements

Thank you to my supervisor, Michael Knap and also to Frank Pollmann for providing guidance and feedback throughout this project and making my research possible. Thanks also to Johannes Feldmeier, Alexander Schuckert, ShengHsuan Lin and Laurens Vanderstraeten for stimulating discussion and countless pieces of help and advice. To Stefan Birnkammer, Amelie Flatt, Wolfgang Gottwald, Rika Unkelbach, Nina Schwarz, Leonie Spitz, Nepomuk Ritz and Ansgar Burchards. To my parents for their support that has made me who I am.

Appendix A

Numerics

A.1 Code Source Files and Numerical Data

The code developed during the course of this thesis is available on GitHub [50]. An implementation of the optimization algorithms (chapter 2) can be found in `jax-optimize`. They accept cost functions whose arguments can be nested structures of `python` containers (lists, tuples, dictionaries, etc.), containing `python` built-in scalars or `jax.numpy` arrays. Algorithms for tensor networks and specifically PEPS and the transverse field Ising model using `jax` as a backend are implemented in `grad-tn`. The numerical data was obtained using the functions in `mas-projects`. It is available in `data`.

A.2 Interpolation Subroutines for Line Search

In this section, we discuss the interpolation subroutines for the line search algorithm
12

Quadratic Interpolation

For the subroutine `quadratic_interpolate(a, fa, f'a, b, fb)`, we want to find the minimizer of the quadratic polynomial $q(x) = Ax^2 + Bx + C$. The coefficients A, B, C are determined by with $q(a) = f_a$, $q'(a) = f'_a$ and $q(b) = f_b$. And we want to obtain the minimizer in terms of these function (and derivative) values.

We can for now assume that $a = 0$ by shifting the x -axis:

$$\begin{aligned} \text{quadratic_interpolate}(a, f_a, f'_a, b, f_b) \\ = \text{quadratic_interpolate}(0, f_a, f'_a, b - a, f_b) + a \end{aligned} \quad (\text{A.1})$$

So from now on, assume $a = 0$. By matching coefficients of the conditions we get

$$A = \frac{f_b - f_a - f'_a}{b^2} \quad ; \quad B = f'_a \quad ; \quad C = f_a \quad (\text{A.2})$$

The minimizer of q for $a = 0$ and thus the output of `quadratic_interpolate` is

$$\text{quadratic_interpolate}(0, f_a, f'_a, b - a, f_b) + a = \tilde{x}_{a=0} = -\frac{B}{2A} = \frac{f'_a b^2}{2(f_b - f_a - f'_a b^2)} \quad (\text{A.3})$$

and the general case ($a \neq 0$) can be recovered using equation (A.1).

Cubic Interpolation

For the subroutine `cubic_interpolate`($a, f_a, f'_a, b, f_b, c, f_c$), we want to find the minimizer of the cubic polynomial $p(x) = Ax^3 + Bx^2 + Cx + D$ with $p(a) = f_a$, $p'(a) = f'_a$, $p(b) = f_b$ and $p(c) = f_c$.

Again we can assume $a = 0$ for now, and recover $a \neq 0$ by using

$$\begin{aligned} & \text{cubic_interpolate}(a, f_a, f'_a, b, f_b, c, f_c) \\ &= \text{cubic_interpolate}(0, f_a, f'_a, b - a, f_b, c - a, f_c) + a \end{aligned} \quad (\text{A.4})$$

The conditions imply

$$D = f_a \quad ; \quad C = f'_a \quad ; \quad Ab^3 + Bb^2 = f_b - f_a - f'_a b \quad ; \quad Ac^3 + Bc^2 = f_c - f_a - f'_a c \quad (\text{A.5})$$

The last two are solved by

$$\begin{pmatrix} A \\ B \end{pmatrix} = \frac{1}{b^2 c^2 (b - c)} \begin{pmatrix} c^2 & -b^2 \\ -c^3 & b^3 \end{pmatrix} \cdot \begin{pmatrix} f_b - f_a - f'_a b \\ f_c - f_a - f'_a c \end{pmatrix} \quad (\text{A.6})$$

and the minimizer of p is one of the two roots of $p'(\tilde{x}) = 0$, namely

$$\tilde{x}_{\pm; a=0} = \frac{-B \pm \sqrt{B^2 - AC}}{3A} \quad (\text{A.7})$$

In a line search context, we can safely choose the positive solution and therefore, the output of `cubic_interpolate` for $a = 0$ is $\tilde{x}_{+; a=0}$. The general case ($a \neq 0$) can be recovered using equation (A.4).

A.3 Extracting the Dynamical Structure Factor

In this section, we discuss how the dynamical (spin) structure factor

$$S^{\alpha\beta}(\mathbf{k}, \omega) := \sum_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \frac{1}{2\pi} \int_{-\infty}^{\infty} dt e^{i\omega t} C^{\alpha\beta}(\mathbf{r}, t) \quad (\text{A.8})$$

can be extracted from numerical data of the real-time correlation function

$$C^{\alpha\beta}(\mathbf{r}, t) := \left\langle \sigma_{\mathbf{r}}^{\alpha}(t) \sigma_{\mathbf{0}}^{\beta}(0) \right\rangle. \quad (\text{A.9})$$

In our case $\langle \cdot \rangle$ is the expectation value w.r.t. the ground state $|\text{GS}\rangle$.

The main challenge in computing $S^{\alpha\beta}(\mathbf{k}, \omega)$ is performing the time-to-frequency Fourier transform, given that $C^{\alpha\beta}(\mathbf{r}, t)$ is only available at positive, discrete times $t_m = m \delta t$ for $m = 0, \dots, M$ up to a cutoff $T = M \delta t$.

First of all, consider the behaviour of $C^{\alpha\beta}(\mathbf{r}, t)$ under time-reversal:

$$\begin{aligned} C^{\alpha\beta}(\mathbf{r}, -t) &= \left\langle e^{-iHt} \sigma_{\mathbf{r}}^{\alpha} e^{iHt} \sigma_{\mathbf{0}}^{\beta} \right\rangle = e^{-iE_0 t} \left\langle \sigma_{\mathbf{r}}^{\alpha} e^{iHt} \sigma_{\mathbf{0}}^{\beta} \right\rangle = \left\langle \sigma_{\mathbf{r}}^{\alpha} e^{iHt} \sigma_{\mathbf{0}}^{\beta} e^{-iHt} \right\rangle \\ &= \left\langle e^{iHt} \sigma_{\mathbf{0}}^{\beta} e^{-iHt} \sigma_{\mathbf{r}}^{\alpha} \right\rangle^* = \left\langle \sigma_{\mathbf{0}}^{\beta}(t) \sigma_{\mathbf{r}}^{\alpha} \right\rangle^* = \left\langle \sigma_{\mathbf{r}}^{\beta}(t) \sigma_{\mathbf{0}}^{\alpha} \right\rangle^* = C^{\beta\alpha}(\mathbf{r}, t)^*, \end{aligned} \quad (\text{A.10})$$

where we have used that the expectation value is in the ground state (with energy E_0), as well as invariance under spatial translations¹ and inversion.

In order to account for the cut-off in time, we introduce a Gaussian² windowing function $W(t)$ of width σ_t , which we choose such that $W(T) =: \eta$ is sufficiently decayed³. In frequency space, this results in convoluting the structure factor with a Gaussian $\tilde{W}(\omega)$ of width $\sigma_{\omega} = \sigma_t^{-1}$, effectively broadening all features:

$$S^{\alpha\beta}(\mathbf{k}, \omega) \approx \tilde{W}(\omega) * S^{\alpha\beta}(\mathbf{k}, \omega) = \sum_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \frac{1}{2\pi} \int_{-\infty}^{\infty} dt e^{i\omega t} W(t) C^{\alpha\beta}(\mathbf{r}, t), \quad (\text{A.11})$$

where $*$ stands for the convolution, i.e. $(f * g)(x) = \int dy f(y)g(x - y)$.

For $\alpha = \beta$, we can reduce the time integration in equation (A.8) to positive times:

$$\begin{aligned} S^{\alpha\alpha}(\mathbf{k}, \omega) &\approx \tilde{W}(\omega) * S^{\alpha\alpha}(\mathbf{k}, \omega) \\ &= \sum_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \frac{1}{2\pi} \int_0^{\infty} dt (e^{i\omega t} W(t) C^{\alpha\alpha}(\mathbf{r}, t) + e^{-i\omega t} W(t) C^{\alpha\alpha}(\mathbf{r}, -t)) \\ &= \sum_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \frac{1}{2\pi} \int_0^{\infty} dt 2 \operatorname{Re} [e^{i\omega t} W(t) C^{\alpha\alpha}(\mathbf{r}, t)] \end{aligned} \quad (\text{A.12})$$

where we have used that the Gaussian $W(t) = W(-t)^*$ is real and symmetric.

Due to the decaying windowing function in the integrand, the integral can be truncated at the cut-off time, and if the time step δt is sufficiently small it can be approximated with an appropriate sum

$$S^{\alpha\alpha}(\mathbf{k}, \omega) \approx \tilde{W}(\omega) * S^{\alpha\alpha}(\mathbf{k}, \omega) \approx \sum_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \frac{1}{M} \sum_{m=0}^M 2 \operatorname{Re} [e^{i\omega t_m} W(t_m) C^{\alpha\alpha}(\mathbf{r}, t_m)] \quad (\text{A.13})$$

Before equation (A.13) is evaluated, the correlation data $C^{\alpha\alpha}(\mathbf{r}, t_m)$ can be extended to longer times using linear predictions [115, 116].

¹While translational invariance is not given for our finite system with open boundary conditions, it nevertheless approximates the behavior of a translationally invariant system up to time-scales when the wavefront of the excitation meets the boundary and finite-size effects become relevant.

²An exponential damping could be used instead. This would result in a Lorentzian broadening of features in the structure factor.

³We found that $\eta = 0.1$ is a good balance between broadening the features too much and introducing too many artifacts.

Artifacts Introduced by the Approximations

Given that the time step δt is sufficiently small, we have introduced two approximations to the structure factor $S^{\alpha\alpha}(\mathbf{k}, \omega)$, namely the Gaussian windowing $W(t)$ and the finite time cut-off T . We have effectively inserted a factor

$$V(t) := W(t)\theta(|T - t|) \quad (\text{A.14})$$

to the integrand of equation (A.8). The result is that the extracted structure factor is convoluted with its Fourier transform

$$\tilde{V}(\omega) := \frac{1}{2\pi} \int_{-\infty}^{\infty} dt e^{i\omega t} V(t). \quad (\text{A.15})$$

The convolution with $\tilde{V}(\omega)$ can also result in negative values of the structure factor, which are not physical.

Appendix B

Complex (Wirtinger) Calculus

We introduce the complex (Wirtinger) calculus and complex derivatives [70, 117].

B.1 Scalar Functions

In this section, we introduce some terminology, define the holomorphic and anti-holomorphic derivative and show (two versions of) the chain-rule, for scalar functions.

It will be convenient to interchangeably use complex quantities and their real counterparts (real and imaginary part). We can express a complex number $z \in \mathbb{C}$, through two real quantities, its real part $x := \text{Re}[z] \in \mathbb{R}$ and imaginary part $y = \text{Im}[z] \in \mathbb{R}$ such that

$$z = x + iy \quad (\text{B.1})$$

It will be convenient to express such relations using the matrix $J \in \mathbb{C}^{2 \times 2}$ and its inverse

$$J = \begin{pmatrix} 1 & i \\ 1 & -i \end{pmatrix} \quad ; \quad J^{-1} = \frac{1}{2} J^\dagger = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ -i & i \end{pmatrix} \quad (\text{B.2})$$

Any function $f : \mathbb{C} \rightarrow \mathbb{C}$ corresponds uniquely to its real counterpart $g : \mathbb{R}^2 \rightarrow \mathbb{R}^2$.

$$g(x, y) := \begin{pmatrix} \text{Re}[f(x + iy)] \\ \text{Im}[f(x + iy)] \end{pmatrix} =: \begin{pmatrix} u(x, y) \\ v(x, y) \end{pmatrix} \quad (\text{B.3})$$

with the inverse relation

$$f(z) = \begin{pmatrix} 1 & i \end{pmatrix} \cdot g(\text{Re}[z], \text{Im}[z]) = u(\text{Re}[z], \text{Im}[z]) + iv(\text{Re}[z], \text{Im}[z]) \quad (\text{B.4})$$

We have a derivative objects for g , namely its Jacobian

$$Dg : \mathbb{R}^2 \rightarrow \mathbb{R}^{2 \times 2}, \quad \begin{pmatrix} x \\ y \end{pmatrix} \mapsto Dg(x, y) := \begin{pmatrix} \frac{\partial u}{\partial x}(x, y) & \frac{\partial u}{\partial y}(x, y) \\ \frac{\partial v}{\partial x}(x, y) & \frac{\partial v}{\partial y}(x, y) \end{pmatrix} \quad (\text{B.5})$$

For demonstrative purposes, let us consider the first-order Taylor approximation of g

$$g(x + \Delta x, y + \Delta y) \approx g(x, y) + Dg(x, y) \cdot \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \quad (\text{B.6})$$

we can rewrite this in terms of f , $z := x + iy$ and $\Delta z := \Delta x + i\Delta y$, to get a Taylor-like approximation for f :

$$f(z + \Delta z) = (1 \quad i) \cdot g(x + \Delta x, y + \Delta y) \approx (1 \quad i) \cdot g(x, y) + (1 \quad i) \cdot Dg(x, y) \cdot \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \quad (\text{B.7})$$

we can forcibly write this in terms of Δz by inserting an identity $\mathbb{1} = 1/2 J^\dagger J$ in terms of J :

$$\begin{aligned} f(z + \Delta z) &\approx f(z) + \underbrace{(1 \quad i) \cdot \begin{pmatrix} \frac{\partial u}{\partial x}(x, y) & \frac{\partial u}{\partial y}(x, y) \\ \frac{\partial v}{\partial x}(x, y) & \frac{\partial v}{\partial y}(x, y) \end{pmatrix} \cdot \frac{1}{2} \begin{pmatrix} 1 & 1 \\ -i & i \end{pmatrix}}_{(\square)} \cdot \begin{pmatrix} 1 & i \\ 1 & -i \end{pmatrix} \cdot \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \\ &= f(z) + \frac{\partial f}{\partial z}(z) \Delta z + \frac{\partial f}{\partial z^\star}(z) \Delta z^\star \end{aligned} \quad (\text{B.8})$$

where we have defined the two components of (\square)

$$\begin{aligned} \frac{\partial f}{\partial z}(x + iy) &:= \frac{1}{2} \left(\frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) (u(x, y) + iv(x, y)) \\ \frac{\partial f}{\partial z^\star}(x + iy) &:= \frac{1}{2} \left(\frac{\partial}{\partial x} + i \frac{\partial}{\partial y} \right) (u(x, y) + iv(x, y)) \end{aligned} \quad (\text{B.9})$$

which are called the *holomorphic derivative* and *anti-holomorphic derivative* of f , respectively.

Their complex conjugates are derivatives of f^\star :

$$\begin{aligned} \left(\frac{\partial f}{\partial z} \right)^\star &= \frac{1}{2} \left(\frac{\partial}{\partial x} + i \frac{\partial}{\partial y} \right) (u - iv) = \frac{\partial f^\star}{\partial z^\star} \\ \left(\frac{\partial f}{\partial z^\star} \right)^\star &= \frac{1}{2} \left(\frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) (u - iv) = \frac{\partial f^\star}{\partial z} \end{aligned} \quad (\text{B.10})$$

We note that equation (B.8) looks like z and z^\star are treated as independent variables in the frame-work of Wirtinger calculus. That is a good intuition and it is true in the following sense

$$\frac{\partial}{\partial z} z = 1 = \frac{\partial}{\partial z^\star} z^\star \quad ; \quad \frac{\partial}{\partial z} z^\star = 0 = \frac{\partial}{\partial z^\star} z \quad (\text{B.11})$$

keeping this in mind, the holomorphic and anti-holomorphic derivatives fulfill the usual derivative rules for products and sums.

We expect that we can also obtain a chain rule, which formally will look like z and z^* are treated as independent variables. We can save ourselves some lengthy notation by defining

$$\mathcal{D}f := \begin{pmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial z^*} \\ \frac{\partial f^*}{\partial z} & \frac{\partial f^*}{\partial z^*} \end{pmatrix} = J \cdot Dg \cdot J^{-1} \quad (\text{B.12})$$

That second row of $\mathcal{D}f$ contains just the complex conjugates of the first row, so the information is redundant. This redundancy, however, allows us to use the chain-rule for Dg .

For two complex-to-complex functions $f_1, f_2 : \mathbb{C} \rightarrow \mathbb{C}$, and their real counter-parts g_1, g_2 , defined analogously to equation (B.3) we get

$$\begin{aligned} \mathcal{D}(f_1 \circ f_2) &= J \cdot D(g_1 \circ g_2) \cdot J^{-1} = J \cdot Dg_2 \cdot Dg_1 \cdot J^{-1} \\ &= J \cdot Dg_2 \cdot J^{-1} \cdot J \cdot Dg_1 \cdot J^{-1} = \mathcal{D}f_2 \cdot \mathcal{D}f_1 \end{aligned} \quad (\text{B.13})$$

considering the first entry, we get the chain-rule for the holomorphic derivative of $f := f_1 \circ f_2$

$$\frac{\partial f}{\partial z}(z) = \frac{\partial f_2}{\partial z}(f_1(z)) \frac{\partial f_1}{\partial z}(z) + \frac{\partial f_2}{\partial z^*}(f_1(z)) \frac{\partial f_1^*}{\partial z}(z) \quad (\text{B.14})$$

which intuitively looks like f depends on z via the two independent f_1 and f_1^* .

For mixed higher-order derivatives, that is mixed between holomorphic and anti-holomorphic derivatives, the order does not matter, since

$$\begin{aligned} \frac{\partial}{\partial z} \frac{\partial f}{\partial z^*} &= \frac{1}{2} \left(\frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) \frac{1}{2} \left(\frac{\partial}{\partial x} + i \frac{\partial}{\partial y} \right) (u + iv) \\ &= \frac{1}{4} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) (u + iv) = \frac{\partial}{\partial z^*} \frac{\partial f}{\partial z} \end{aligned} \quad (\text{B.15})$$

which is reminiscent of Schwarz' theorem for real partial derivatives.

B.2 Vector Functions

For a complex function $F : \mathbb{C}^n \rightarrow \mathbb{C}^m$ we write

$$DF : \mathbb{C}^n \rightarrow \mathbb{C}^{m \times n}, \quad z \mapsto DF(z) = \left(\frac{\partial F_i}{\partial z_j}(z) \right)_{ij} \quad (\text{B.16})$$

for the holomorphic Jacobian of f and

$$D^*F : \mathbb{C}^n \rightarrow \mathbb{C}^{m \times n}, \quad \mathbf{z} \mapsto DF(\mathbf{z}) = \left(\frac{\partial F_i}{\partial z_j^*}(\mathbf{z}) \right)_{ij} \quad (\text{B.17})$$

for the anti-holomorphic Jacobian. Similar to equation (B.10), we get that

$$(DF)^* = D^*F^* \quad ; \quad (D^*F)^* = DF^* \quad (\text{B.18})$$

To obtain a chain rule, let us define

$$\mathcal{D}F := \begin{pmatrix} DF & D^*F \\ DF^* & D^*F^* \end{pmatrix} = J \cdot DG \cdot J^{-1} \quad (\text{B.19})$$

where $G : \mathbb{R}^{2n} \rightarrow \mathbb{R}^{2m}$ is the real counter-part of F , analogous to equation (B.3), and essentially $F(\mathbf{x} + i\mathbf{y}) = (\mathbb{1} \quad i\mathbb{1}) \cdot G(\mathbf{x}, \mathbf{y})$.

Then for $F_1 : \mathbb{C}^n \rightarrow \mathbb{C}^k$ and $F_2 : \mathbb{C}^k \rightarrow \mathbb{C}^m$ and $F = F_1 \circ F_2$ we get the chain rule

$$\begin{aligned} \mathcal{D}F &= J \cdot D(G_1 \circ G_2) \cdot J^{-1} = J \cdot DG_2 \cdot DG_1 \cdot J^{-1} \\ &= J \cdot DG_2 \cdot J^{-1} \cdot J \cdot DG_1 \cdot J^{-1} = \mathcal{D}F_2 \cdot \mathcal{D}F_1 \end{aligned} \quad (\text{B.20})$$

the upper-left block of which reads

$$DF(\mathbf{z}) = DF_2(F_1(\mathbf{z})) \cdot DF_1(\mathbf{z}) + D^*F_2(F_1(\mathbf{z})) \cdot DF_1^*(\mathbf{z}) \quad (\text{B.21})$$

Appendix C

Derivations for Automatic Differentiation

C.1 Rigorous Treatment of General Computation Graphs

In this section, we make the intuitive arguments of section 3.3 more rigorous.

We consider a function

$$f : \prod_{n=1}^N W_n \rightarrow \prod_{n=N+K+1}^{N+K+M} W_n \quad (\text{C.1})$$

with an implementation that gives us its computation graph. We extract the following (general) information from the computation graph:

- There are N input nodes $\mathbf{w}_1, \dots, \mathbf{w}_N$.
- There are K intermediate-value nodes $\mathbf{w}_{N+1}, \dots, \mathbf{w}_{N+K}$.
- There are M output nodes $\mathbf{w}_{N+K+1}, \dots, \mathbf{w}_{N+K+M}$.
- All of these primal values $\mathbf{w}_n \in W_n$ each live in real coordinate spaces $W_n = \mathbb{R}^{d_n}$.
- There are J primitives g_1, \dots, g_J . They can be applied *in this order* to implement f . Each primitive $g_j : \prod_{n \in \mathcal{I}_j} W_n \rightarrow \prod_{n \in \mathcal{O}_j} W_n$ maps from some subset $\{\mathbf{w}_n \mid n \in \mathcal{I}_j\}$ of all the primals to some other subset $\{\mathbf{w}_n \mid n \in \mathcal{O}_j\}$ of all the primals.

The index sets $\mathcal{I}_j \subset \{1, \dots, N+K\}$ and $\mathcal{O}_j \subset \{N+1, \dots, N+K+M\}$ are determined by the computation graph and are disjoint (inputs of a primitive cannot be outputs of the same primitive).

In order to deal with the general computation graph and its branching nature, we define a second computation graph that also implements f . From a computational perspective it would be wildly inefficient, but it is easier to treat analytically. The idea is to operate on the (very large) space

$$\Omega := \prod_{n=1}^{N+K+M} W_n \quad (\text{C.2})$$

and translate primitives to it. Intermediate values are kept at a trivial value, say 0, until they are computed in the original computation graph. They are then overwritten exactly ones and then “kept active” until the very end. This makes the new computation graph sequential and easier to deal with.

The first map in the new computation graph is to embed the inputs into Ω

$$\gamma_0 : \prod_{n=1}^N W_n \rightarrow \Omega, (\mathbf{w}_1, \dots, \mathbf{w}_N) \mapsto (\mathbf{w}_1, \dots, \mathbf{w}_N, \mathbf{0}, \dots, \mathbf{0}) \quad (\text{C.3})$$

Then for every primitive $g_j : \prod_{n \in \mathcal{I}_j} W_n \rightarrow \prod_{n \in \mathcal{O}_j} W_n$ of the original computation graph, we define a map on Ω . If $m \in \mathcal{O}_j$, we write $g_j^{(m)} : \prod_{n \in \mathcal{I}_j} W_n \rightarrow W_m$ for the component of g_j that lives in W_m . Essentially, if m is the k^{th} element of \mathcal{O}_j , then $g_j^{(m)}$ is the k^{th} component of g_j . We define

$$\begin{aligned} \gamma_j : \Omega \rightarrow \Omega, \mathbf{w} \mapsto & \left(\gamma_j^{(1)}(\mathbf{w}), \gamma_j^{(2)}(\mathbf{w}), \dots \right); \\ \gamma_j^{(m)} : \Omega \rightarrow W_m, (\mathbf{w}_1, \mathbf{w}_2, \dots) \mapsto & \begin{cases} g_j^{(m)}(\{\mathbf{w}_n \mid n \in \mathcal{I}_j\}) & m \in \mathcal{O}_j \\ \mathbf{w}_m & \text{else} \end{cases}. \end{aligned} \quad (\text{C.4})$$

Essentially, this map γ_j acts trivially on all components of $\mathbf{w} \in \Omega$, except for the ones that represent outputs of the original primitive g_j . Those components are “overwritten” with the output of g_j .

The last map we need is extracting outputs from Ω

$$\gamma_{-1} : \Omega \rightarrow \prod_{n=N+K+1}^{N+K+M} W_n, (\mathbf{w}_1, \dots, \mathbf{w}_{N+K+M}) \mapsto (\mathbf{w}_{N+K+1}, \dots, \mathbf{w}_{N+K+M}) \quad (\text{C.5})$$

By construction these maps implement the top-level function f :

$$f = \gamma_{-1} \circ \gamma_J \circ \dots \circ \gamma_1 \circ \gamma_0 \quad (\text{C.6})$$

Let us abbreviate the intermediate primals of the new computation graph. We write $\mathbf{x} := (\mathbf{w}_1, \dots, \mathbf{w}_N) \prod_{n=1}^N W_n$ for the inputs and $\mathbf{a}^{(0)} := \gamma_0(\mathbf{x}) \in \Omega$ for the first intermediate of the new graph. Subsequently for $j = 1, \dots, J$, we define $\mathbf{a}^{(j)} := \gamma_j(\mathbf{a}^{(j-1)}) \in \Omega$, and finally the output $\mathbf{y} := \gamma_{-1}(\mathbf{a}^{(J)}) = f(\mathbf{x}) \prod_{n=N+K+1}^{N+K+M} W_n$.

We can apply the chain-rule (3.10) of the JVP to get an expression for the JVP of f . In particular for the tangent $\mathbf{v} = (\mathbf{v}_1, \dots, \mathbf{v}_N) \in \prod_{n=1}^N W_n$ we get

$$\text{JVP}_f(\mathbf{x}, *) = \text{JVP}_{\gamma_{-1}}(\mathbf{a}^{(J)}, *) \circ \text{JVP}_{\gamma_J}(\mathbf{a}^{(J-1)}, *) \circ \dots \circ \text{JVP}_{\gamma_1}(\mathbf{a}^{(0)}, *) \circ \text{JVP}_{\gamma_0}(\mathbf{x}, *) \quad (\text{C.7})$$

where we abbreviated the partially evaluated $\text{JVP}_f(\mathbf{x}, *) : \mathbf{v} \mapsto \text{JVP}_f(\mathbf{x}, \mathbf{v})$.

The JVP of γ_0 is straight-forward; For a tangent $\mathbf{v} = (\mathbf{v}_1, \dots, \mathbf{v}_N) \in \prod_{n=1}^N W_n$

$$\text{JVP}_{\gamma_0}(\mathbf{x}, \mathbf{v}) = (\mathbf{v}_1, \dots, \mathbf{v}_N, \mathbf{0}, \dots, \mathbf{0}). \quad (\text{C.8})$$

So is the JVP of γ_{-1} ;

$$\text{JVP}_{\gamma_{-1}}(\mathbf{a}^{(J)}, (\mathbf{v}_1, \dots, \mathbf{v}_{N+K+M})) = (\mathbf{v}_{N+K+1}, \dots, \mathbf{v}_{N+K+M}). \quad (\text{C.9})$$

The Jacobians of the γ_j have a block structure

$$D\gamma_j = \begin{pmatrix} \mathbb{1} & 0 & \dots & \dots & \mathcal{I}_j & \mathcal{O}_j & \dots & \dots & 0 \\ 0 & \mathbb{1} & & & & & & & \vdots \\ \vdots & & \ddots & & & & & & \vdots \\ \vdots & & & \mathbb{1} & 0 & & & & \vdots \\ \vdots & & & Dg_j & 0 & & & & \vdots \\ \vdots & & & & & \ddots & & & \vdots \\ \vdots & & & & & & \mathbb{1} & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 & \mathbb{1} \end{pmatrix} \begin{matrix} \mathcal{I}_j \\ \mathcal{O}_j \end{matrix}, \quad (\text{C.10})$$

where we have assumed that \mathcal{I}_j and \mathcal{O}_j are both contiguous and mutually adjacent to make notation easier. This is, however, not required to make the argument. Therefore, the m^{th} component (living in W_m) of the JVP of γ_j is given by

$$\begin{aligned} & \text{JVP}_{\gamma_j}((\mathbf{a}_1, \dots, \mathbf{a}_{N+K+M}), (\mathbf{v}_1, \dots, \mathbf{v}_{N+K+M}))_m \\ &= \begin{cases} \text{JVP}_{g_j^{(m)}}(\{\mathbf{a}_n \mid n \in \mathcal{I}_j\}, \{\mathbf{v}_n \mid n \in \mathcal{I}_j\}) & m \in \mathcal{O}_j \\ \mathbf{v}_m & \text{else} \end{cases} \end{aligned} \quad (\text{C.11})$$

The JVP of γ_j acts as JVP_{g_j} on tangents which correspond to outputs of g_j , and trivially on others. The JVPs of γ_0 and γ_{-1} act as “setting” the input tangents and reading out the result from the output tangents, respectively. Therefore, equation (C.7) is equivalent to the algorithm presented in section 3.3.4.

Analogously, the backward mode algorithm presented in section 3.3.5 can be verified by studying the VJPs of the γ_j . The summation of cotangent contributions arises from the fact, that two blocks of the jacobian (in the \mathcal{I}_j column of equation (C.10)) contribute to the cotangent component of the inputs of γ_j . Therefore, all primitives that use a given primal as an input contribute additively to its cotangent.

C.2 Derivations for Complex Functions

In this section, we derive the results states in section 3.4.

First, we need chain rules for the complex JVP and VJP. Consider functions $g : \mathbb{C}^n \rightarrow \mathbb{C}^k$ and $h : \mathbb{C}^k \rightarrow \mathbb{C}^m$.

Chain Rule for the Complex JVP

The complex JVP fulfills the chain rule

$$\text{JVP}_{h \circ g}(\mathbf{z}, \mathbf{v}) = \text{JVP}_h(g(\mathbf{z}, \text{JVP}_g(\mathbf{z}, \mathbf{v}))) \quad (\text{C.12})$$

for $\mathbf{z}, \mathbf{v} \in \mathbb{C}^n$. To prove it, it is convenient to write equation (C.12) and its complex conjugate at the same time and define $f := h \circ g$:

$$\begin{aligned} \begin{pmatrix} \text{RHS} \\ \text{RHS}^* \end{pmatrix} &\stackrel{(3.37)}{=} \begin{pmatrix} Dh(g(\mathbf{z})) & D^*h(g(\mathbf{z})) \\ Dh^*(g(\mathbf{z})) & D^*h^*(g(\mathbf{z})) \end{pmatrix} \cdot \begin{pmatrix} \text{JVP}_g(\mathbf{z}, \mathbf{v}) \\ \text{JVP}_g(\mathbf{z}, \mathbf{v})^* \end{pmatrix} \\ &\stackrel{(3.37)}{=} \begin{pmatrix} Dh(g(\mathbf{z})) & D^*h(g(\mathbf{z})) \\ Dh^*(g(\mathbf{z})) & D^*h^*(g(\mathbf{z})) \end{pmatrix} \cdot \begin{pmatrix} Dg(\mathbf{z}) & D^*g(\mathbf{z}) \\ Dg^*(\mathbf{z}) & D^*g^*(\mathbf{z}) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{v} \\ \mathbf{v}^* \end{pmatrix} \\ &\stackrel{(\text{B.20})}{=} \begin{pmatrix} Df(\mathbf{z}) & D^*f(\mathbf{z}) \\ Df^*(\mathbf{z}) & D^*f^*(\mathbf{z}) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{v} \\ \mathbf{v}^* \end{pmatrix} \stackrel{(3.37)}{=} \begin{pmatrix} \text{JVP}_f(\mathbf{z}, \mathbf{v}) \\ \text{JVP}_f(\mathbf{z}, \mathbf{v})^* \end{pmatrix} = \begin{pmatrix} \text{LHS} \\ \text{LHS}^* \end{pmatrix} \end{aligned} \quad (\text{C.13})$$

Chain Rule for the Complex VJP

Similarly we get the VJP chain rule

$$\text{VJP}_{h \circ g}(\mathbf{z}, \mathbf{w}^T) = \text{VJP}_g(\mathbf{z}, \text{VJP}_h(g(\mathbf{z}, \mathbf{w}^T))) \quad (\text{C.14})$$

for $\mathbf{z} \in \mathbb{C}^n$ and $\mathbf{w} \in \mathbb{C}^m$. Proof:

$$\begin{aligned} \begin{pmatrix} \text{RHS} & \text{RHS}^* \end{pmatrix} &\stackrel{(3.38)}{=} \begin{pmatrix} \text{VJP}_h(g(\mathbf{z}), \mathbf{w}^T) & \text{VJP}_h(g(\mathbf{z}), \mathbf{w}^T)^* \end{pmatrix} \cdot \begin{pmatrix} Dg(\mathbf{z}) & D^*g(\mathbf{z}) \\ Dg^*(\mathbf{z}) & D^*g^*(\mathbf{z}) \end{pmatrix} \\ &\stackrel{(3.38)}{=} \begin{pmatrix} \mathbf{w}^T & \mathbf{w}^\dagger \end{pmatrix} \cdot \begin{pmatrix} Dh(g(\mathbf{z})) & D^*h(g(\mathbf{z})) \\ Dh^*(g(\mathbf{z})) & D^*h^*(g(\mathbf{z})) \end{pmatrix} \cdot \begin{pmatrix} Dg(\mathbf{z}) & D^*g(\mathbf{z}) \\ Dg^*(\mathbf{z}) & D^*g^*(\mathbf{z}) \end{pmatrix} \\ &\stackrel{(\text{B.20})}{=} \begin{pmatrix} \mathbf{w}^T & \mathbf{w}^\dagger \end{pmatrix} \cdot \begin{pmatrix} Df(\mathbf{z}) & D^*f(\mathbf{z}) \\ Df^*(\mathbf{z}) & D^*f^*(\mathbf{z}) \end{pmatrix} \\ &\stackrel{(3.38)}{=} \begin{pmatrix} \text{VJP}_f(\mathbf{z}, \mathbf{w}^T) \end{pmatrix} = \begin{pmatrix} \text{LHS} & \text{LHS}^* \end{pmatrix} \end{aligned} \quad (\text{C.15})$$

Derivatives from the Complex JVP

Given the chain rule (C.12), we can use the autodiff formalism developed in section 3.3 to compute the JVP of complex top-level functions. Let us now show the derivative relations presented in section 3.4.2.

Consider a function

$$f : \mathbb{C}^n \rightarrow \mathbb{C}^m, \mathbf{z} := \mathbf{x} + i\mathbf{y} \mapsto f(\mathbf{z}) := u(\mathbf{x}, \mathbf{y}) + i v(\mathbf{x}, \mathbf{y}) \quad (\text{C.16})$$

expressed via real and imaginary parts. We use the second part of the definition (3.37).

- Real-to-Real:

If $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is actually a real-valued function of real variables, we have $v = 0$, $\mathbf{y} = 0$, $D_y u = 0$, $D_y v = 0$ and therefore

$$\text{JVP}_f(\mathbf{z}, \mathbf{e}_j) = D_x u(\mathbf{x}, 0) \cdot \mathbf{e}_j = \frac{\partial u(\mathbf{x}, 0)}{\partial x_j} = \frac{\partial f(\mathbf{x})}{\partial x_j} \quad (\text{C.17})$$

- Real-to-Complex:

We have $\mathbf{y} = 0$, $D_y u = 0$, $D_y v = 0$ and therefore

$$\text{JVP}_f(\mathbf{z}, \mathbf{e}_j) = (D_x u(\mathbf{x}, 0) + i D_x v(\mathbf{x}, 0)) \cdot \mathbf{e}_j = \frac{\partial u(\mathbf{x}, 0) + i v(\mathbf{x}, 0)}{\partial x_j} = \frac{\partial f(\mathbf{x})}{\partial x_j} \quad (\text{C.18})$$

- Holomorphic Complex-to-Complex:

We have $D^* f = 0$ and therefore

$$\text{JVP}_f(\mathbf{z}, \mathbf{e}_j) = Df \cdot \mathbf{e}_j = \frac{\partial f(\mathbf{z})}{\partial z_j} \quad (\text{C.19})$$

- Non-Holomorphic Complex-to-Complex:

$$\text{JVP}_f(\mathbf{z}, \mathbf{e}_j) = \frac{\partial f(\mathbf{z})}{\partial z_j} + \frac{\partial f(\mathbf{z})}{\partial z_j^*} \quad ; \quad \text{JVP}_f(\mathbf{z}, i\mathbf{e}_j) = i \frac{\partial f(\mathbf{z})}{\partial z_j} - i \frac{\partial f(\mathbf{z})}{\partial z_j^*} \quad (\text{C.20})$$

and therefore

$$\frac{\partial f(\mathbf{z})}{\partial z_j} = \frac{1}{2} \text{JVP}_f(\mathbf{z}, \mathbf{e}_j) - \frac{i}{2} \text{JVP}_f(\mathbf{z}, i\mathbf{e}_j) \quad (\text{C.21})$$

$$\frac{\partial f(\mathbf{z})}{\partial z_j^*} = \frac{1}{2} \text{JVP}_f(\mathbf{z}, \mathbf{e}_j) + \frac{i}{2} \text{JVP}_f(\mathbf{z}, i\mathbf{e}_j) \quad (\text{C.22})$$

- Complex-to-Real:

Same as Non-Holomorphic Complex-to-Complex.

Derivatives from the Complex VJP

Similarly for the VJP:

- Real-to-Real:

If $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is actually a real-valued function of real variables, we have $v = 0$, $\mathbf{y} = 0$, $D_y u = 0$, $D_y v = 0$ and therefore

$$\text{VJP}_f(\mathbf{z}, \mathbf{e}_j^T) = \mathbf{e}_j^T \cdot D_x u(\mathbf{x}, 0) = \frac{\partial u_j(\mathbf{x}, 0)}{\partial \mathbf{x}} = \frac{\partial f_j(\mathbf{x})}{\partial \mathbf{x}} \quad (\text{C.23})$$

- Complex-to-Real:

Use $f^* = f$ and $\mathbf{e}_j^\dagger = \mathbf{e}_j^T$:

$$\text{VJP}_f(\mathbf{z}, \mathbf{e}_j^T) = 2\mathbf{e}_j^T \cdot Df(\mathbf{z}) = 2 \left(\frac{\partial f_j}{\partial z_1}(\mathbf{z}) \quad \dots \quad \frac{\partial f_j}{\partial z_n}(\mathbf{z}) \right) \quad (\text{C.24})$$

- Holomorphic Complex-to-Complex:

We have $Df^\star = (D^\star f)^\star = 0$ and therefore

$$\text{VJP}_f(\mathbf{z}, \mathbf{e}_j^\text{T}) = \mathbf{e}_j^\text{T} \cdot Df(\mathbf{z}) = \left(\frac{\partial f_j}{\partial z_1}(\mathbf{z}) \quad \dots \quad \frac{\partial f_j}{\partial z_n}(\mathbf{z}) \right) \quad (\text{C.25})$$

- Non-Holomorphic Complex-to-Complex:

$$\frac{\partial f_j}{\partial \mathbf{z}} = \frac{1}{2} \text{VJP}_f(\mathbf{z}, \mathbf{e}_j^\text{T}) - \frac{\mathbf{i}}{2} \text{VJP}_f(\mathbf{z}, \mathbf{i} \mathbf{e}_j^\text{T}) \quad (\text{C.26})$$

$$\frac{\partial f_j^\star}{\partial \mathbf{z}} = \frac{1}{2} \text{VJP}_f(\mathbf{z}, \mathbf{e}_j^\text{T}) + \frac{\mathbf{i}}{2} \text{VJP}_f(\mathbf{z}, \mathbf{i} \mathbf{e}_j^\text{T}) \quad (\text{C.27})$$

There is no way to obtain anti-holomorphic derivatives.

- Real-to-Complex

$$\frac{\partial f_j}{\partial \mathbf{x}} = \text{VJP}_f(\mathbf{z}, \mathbf{e}_j^\text{T}) - \text{VJP}_f(\mathbf{z}, \mathbf{i} \mathbf{e}_j^\text{T}) \quad (\text{C.28})$$

C.3 Collection of Automatic Differentiation Formulae

In this section, we discuss in detail a few JVP and VJP formulae for primitives that are particularly relevant to tensor network methods, such as finding fixed points of an iterative procedure, the complex Singular Value Decomposition and contractions of generic tensor networks.

More basic autodiff formulae can be found e.g. in [118, 119].

C.3.1 Fixed Points

An operation that often appears in tensor network methods is to repeat an action on a data structure until it is sufficiently converged. We might formalize this as considering the sequence $\mathbf{x}_k \in \mathbb{R}^n$ generated by

$$\mathbf{x}_{k+1} := f(\mathbf{a}, \mathbf{x}_k), \quad (\text{C.29})$$

where the map $f : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ formalizes the action that is repeatedly applied and $\mathbf{a} \in \mathbb{R}^m$ is a vector of parameters (unchanged by the action). We are then interested in the fixed-point $\mathbf{x}_\star := \lim_{k \rightarrow \infty} \mathbf{x}_k$. Since this limit depends on the parameter \mathbf{a} , this implicitly defines a function

$$\mathbf{x}_\star : \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad \mathbf{a} \mapsto \mathbf{x}_\star(\mathbf{a}), \quad (\text{C.30})$$

such that for all $\mathbf{a} \in \mathbb{R}^m$

$$\mathbf{x}_\star(\mathbf{a}) = f(\mathbf{a}, \mathbf{x}_\star(\mathbf{a})). \quad (\text{C.31})$$

We can write down the VJP of the map \mathbf{x}_\star of equation (C.30). We adopt the procedure laid out in [49]¹ to our notation. First we differentiate equation (C.31) w.r.t. \mathbf{a} :

$$D\mathbf{x}_\star(\mathbf{a}) = \underbrace{D_1 f(\mathbf{a}, \mathbf{x}_\star(\mathbf{a}))}_{=:B} + \underbrace{D_2 f(\mathbf{a}, \mathbf{x}_\star(\mathbf{a}))}_{=:A} \cdot D\mathbf{x}_\star(\mathbf{a}). \quad (\text{C.32})$$

This implies that we can evaluate the VJP as

$$\text{VJP}_{\mathbf{x}_\star}(\mathbf{a}, \mathbf{v}^\text{T}) = \mathbf{v}^\text{T} \cdot D\mathbf{x}_\star(\mathbf{a}) \stackrel{(\text{C.32})}{=} \mathbf{v}^\text{T} \cdot (\mathbb{1} - A)^{-1} \cdot B =: \mathbf{w}^\text{T} \cdot B, \quad (\text{C.33})$$

where we have defined $\mathbf{w}^\text{T} := \mathbf{v}^\text{T} \cdot (\mathbb{1} - A)^{-1}$, which means that $\mathbf{w}^\text{T} = \mathbf{v}^\text{T} + \mathbf{w}^\text{T} \cdot A$. In other words, \mathbf{w}^T is the fixed point of the map

$$\mathbf{u}^\text{T} \mapsto \mathbf{v}^\text{T} + \mathbf{u}^\text{T} \cdot A. \quad (\text{C.34})$$

In summary, we can compute the VJP of the fixed-point map $\mathbf{a} \mapsto \mathbf{x}_\star(\mathbf{a})$ by finding \mathbf{w}^T as the fixed point of equation (C.34), and then applying equation (C.33).

The vector-matrix products are themselves VJPs - but of f . In particular, $\mathbf{u}^\text{T} \cdot A$ is a VJP of the map $\mathbf{x}' \mapsto f(\mathbf{a}, \mathbf{x}')$ and $\mathbf{w}^\text{T} \cdot B$ is a VJP of the map $\mathbf{a}' \mapsto f(\mathbf{a}', \mathbf{x}_\star(\mathbf{a}))$.

¹In particular, details can be found at https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html

A necessary requirement for this procedure to be valid is that the iteration converges to a single fixed point (i.e. that equation (C.31) holds). In tensor network methods, such as CTMRG or DMRG sweeps, this is not necessarily the case, since there is a gauge freedom on virtual bonds of tensor networks. In brief, $\mathbb{1} = U^{-1}U$ for any invertible matrix U can be inserted at virtual bonds, thus different tensors yield the same contraction of the network. Iterative algorithms could converge in the sense, that further iterations *only* act as such “gauge transformations”. If they are non-trivial, however, equation (C.31) is no longer valid and therefore neither is the VJP formula.

C.3.2 The Complex Singular Value Decomposition

The formula for backward mode autodiff of the complex-valued singular value decomposition (SVD) has recently been discovered [120]. Here, we adapt it to our notation.

A singular value decomposition (SVD) of a matrix $A \in \mathbb{C}^{n \times m}$ is given by isometric matrices $U \in \mathbb{C}^{n \times k}$ and $V \in \mathbb{C}^{m \times k}$, and a real diagonal matrix $S = \text{diag}(s_1, \dots, s_k) \in \mathbb{R}^{k \times k}$ with the non-negative singular values $s_i \geq 0$, such that $A = USV^\dagger$, $U^\dagger U = \mathbb{1}$ and $V^\dagger V = \mathbb{1}$. Here, $k = \min(n, m)$.

A singular value decomposition is guaranteed to exist, but it is not unique. First, we can demand that the singular values be arranged in descending order, i.e. $s_1 \geq s_2 \geq \dots \geq s_k$. But still, for a diagonal matrix $\Lambda \in \mathbb{C}^{k \times k}$ of phase factors, that is $\Lambda = \text{diag}(e^{i\theta_1}, \dots, e^{i\theta_k})$, $U' := U\Lambda$, S and $V' := V\Lambda$ is also an SVD of $A = USV^\dagger$. Therefore, the SVD is not a well-defined map from A to (U, S, V) in the strict mathematical sense. Automatic differentiation can, however, still be applied, if the top-level function, say f , is gauge-invariant. That is if $f(U, S, V) = f(U\Lambda, S, V\Lambda)$ for all Λ of the above form.

The VJP formula for $\bar{A} = \text{VJP}_{\text{SVD}}(A, (\bar{U}, \bar{S}, \bar{V}))$ is given by [120]:

$$\begin{aligned} \bar{A} = & U^\star \bar{S} V^T + U^\star (J + J^\dagger) S V^T + U^\star S (K + K^\dagger) V^T \\ & + \frac{1}{2} U^\star S^{-1} (L^\dagger - L) V^\dagger \\ & + (\mathbb{1} - U^\star U^T) \bar{U} S^{-1} V^T + U^\star S^{-1} \bar{V}^\dagger (\mathbb{1} - V^\star V^T), \end{aligned} \quad (\text{C.35})$$

where

$$\begin{aligned} J &:= F \circ (U^T \bar{U}) \quad ; \quad K := F \circ (V^T \bar{V}) \quad ; \quad L := \mathbb{1} \circ (V^T \bar{V}) \\ F_{ij} &= \begin{cases} (s_j^2 - s_i^2)^{-1} & i \neq j \\ 0 & i = j \end{cases}. \end{aligned} \quad (\text{C.36})$$

Here, \circ is the elementwise product $(A \circ B)_{ij} = A_{ij} B_{ij}$. The term in the second line (red) of equation (C.35) vanishes for the real-valued SVD. The terms in the third line (blue) vanish if A is a square matrix.

C.3.3 Tensor Network Contractions

We can write down formulae for the JVP and VJP of a tensor network contraction. We understand such contractions as functions

$$f : V_1 \times \dots \times V_N \rightarrow U, (A_1, \dots, A_N) \mapsto f(A_1, \dots, A_N) \quad (\text{C.37})$$

between spaces V_1, \dots, V_N, U of tensors, e.g. $U = \mathbb{R}^{n \times k \times m}$ for some $n, k, m \in \mathbb{N}$, that are multilinear. That is, for any position $j = 1, \dots, N$, tensors $A_i \in V_i$, $B \in V_j$ and scalar λ

$$\begin{aligned} f(A_1, \dots, A_{j-1}, A_j + \lambda B, A_{j+1}, \dots, A_N) \\ = f(A_1, \dots, A_N) + \lambda f(A_1, \dots, A_{j-1}, B, A_{j+1}, \dots, A_N). \end{aligned} \quad (\text{C.38})$$

Consider, for example, the (tensor-valued) function $g(A, X, Y, Z)$ with components

$$g(A, X, Y, Z)_{ijk} = \sum_{\alpha\beta\gamma\delta} A_{\alpha i} X_{\beta\beta\alpha\gamma} Y_{\delta\gamma k} Z_{j\delta} =$$

$$\quad . \quad (\text{C.39})$$

The JVP formula is straight-forward to write down:

$$\text{JVP}_f \left((A_1, \dots, A_N), (\dot{A}_1, \dots, \dot{A}_N) \right) = \sum_{j=1}^N f(A_1, \dots, A_{j-1}, \dot{A}_j, A_{j+1}, \dots, A_N). \quad (\text{C.40})$$

There are as many terms as there are tensors. In each term, a different tensor is replaced by the corresponding tangent.

As an example consider the components of $\dot{B} := \text{JVP}_g \left((A, X, Y, Z), (\dot{A}, \dot{X}, \dot{Y}, \dot{Z}) \right)$:

$$\dot{B}_{ijk} =$$

$$+ \quad + \quad + \quad . \quad (\text{C.41})$$

The formula for the VJP is given by

$$\begin{aligned} \text{VJP}_f \left((A_1, \dots, A_N), \overline{B} \right) &= (\overline{A}_1 \dots \overline{A}_N) \\ \text{where } \overline{A}_j &= \overline{B} \left(f(A_1, \dots, A_{j-1}, \cancel{A_j}, A_{j+1}, \dots, A_N) \right), \end{aligned} \quad (\text{C.42})$$

where the crossed out tensor signifies simply omitting it in the contraction diagram and leaving the corresponding legs open. For example

$$\overline{Y} := \overline{B} \left(g(A, X, \cancel{X}, Z) \right)_{\delta\gamma k} =$$

$$\quad . \quad (\text{C.43})$$

We have JVP and VJP formulas for tensor network contractions. They both require to contract N diagrams of similar type and connectivity², where N is the number of tensors in the network.

Therefore, we can implement them by only slightly modifying the original contraction code. With clever bookkeeping, the increase in cost (by a factor N) can be drastically reduced. The N diagrams required in the computation of the JVP (or the VJP) are very similar and the previously computed partial contractions of one diagram can be reused in the others.

²For the JVP, the network is exactly the same. For the VJP one tensor is omitted, while the output cotangent is added to the network at a different place.

Bibliography

- [1] Taesoo D Lee and Abasifreke U Ebong. “A review of thin film solar cell technologies and challenges”. In: *Renewable and Sustainable Energy Reviews* 70 (2017), pp. 1286–1297.
- [2] Heiko Thomas et al. “Superconducting transmission lines–Sustainable electric energy transfer with higher public acceptance?”. In: *Renewable and Sustainable Energy Reviews* 55 (2016), pp. 59–72.
- [3] Vitaly L Ginzburg. “Nobel Lecture: On superconductivity and superfluidity (what I have and have not managed to do) as well as on the “physical minimum” at the beginning of the XXI century”. In: *Reviews of Modern Physics* 76.3 (2004), p. 981.
- [4] Vitaly L Ginzburg and Lev D Landau. “On the theory of superconductivity”. In: *On Superconductivity and Superfluidity*. Springer, 2009, pp. 113–137.
- [5] Alexei Kitaev. “Periodic table for topological insulators and superconductors”. In: *AIP conference proceedings*. Vol. 1134. 1. American Institute of Physics. 2009, pp. 22–30.
- [6] Alexander Altland and Martin R. Zirnbauer. “Nonstandard symmetry classes in mesoscopic normal-superconducting hybrid structures”. In: *Phys. Rev. B* 55 (2 Jan. 1997), pp. 1142–1161.
- [7] RF Bishop, PHY Li, and CE Campbell. “Valence-bond crystalline order in the $s=1/2$ J1–J2 model on the honeycomb lattice”. In: *Journal of Physics: Condensed Matter* 25.30 (2013), p. 306002.
- [8] Luca Capriotti and Sandro Sorella. “Spontaneous Plaquette Dimerization in the J1-J2 Heisenberg Model”. In: *Physical Review Letters* 84.14 (2000), p. 3173.
- [9] Zhenyue Zhu and Steven R White. “Spin liquid phase of the $S=1/2$ J1-J2 Heisenberg model on the triangular lattice”. In: *Physical Review B* 92.4 (2015), p. 041105.
- [10] Wen-Jun Hu et al. “Competing spin-liquid states in the spin-1 2 Heisenberg model on the triangular lattice”. In: *Physical Review B* 92.14 (2015), p. 140403.
- [11] Fabio Mezzacapo and Massimo Boninsegni. “Ground-state phase diagram of the quantum J1- J2 model on the honeycomb lattice”. In: *Physical Review B* 85.6 (2012), p. 060402.
- [12] Fabio Mezzacapo. “Ground-state phase diagram of the quantum J1- J2 model on the square lattice”. In: *Physical Review B* 86.4 (2012), p. 045115.
- [13] Pierre Pfeuty. “The one-dimensional Ising model with a transverse field”. In: *ANNALES of Physics* 57.1 (1970), pp. 79–90.
- [14] Subir Sachdev. “Quantum phase transitions”. In: *Physics world* 12.4 (1999), p. 33.

- [15] Lars Onsager. “Crystal statistics. I. A two-dimensional model with an order-disorder transition”. In: *Physical Review* 65.3-4 (1944), p. 117.
- [16] WMC Foulkes et al. “Quantum Monte Carlo simulations of solids”. In: *Reviews of Modern Physics* 73.1 (2001), p. 33.
- [17] Alexey N Rubtsov, Vladimir V Savkin, and Alexander I Lichtenstein. “Continuous-time quantum Monte Carlo method for fermions”. In: *Physical Review B* 72.3 (2005), p. 035122.
- [18] Matthias Troyer and Uwe-Jens Wiese. “Computational complexity and fundamental limitations to fermionic quantum Monte Carlo simulations”. In: *Physical review letters* 94.17 (2005), p. 170201.
- [19] Matthew B Hastings. “An area law for one-dimensional quantum systems”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2007.08 (2007), P08024.
- [20] Mark Fannes, Bruno Nachtergaele, and Reinhard F Werner. “Finitely correlated states on quantum spin chains”. In: *Communications in mathematical physics* 144.3 (1992), pp. 443–490.
- [21] Marcus Cramer et al. “Entanglement-area law for general bosonic harmonic lattice systems”. In: *Physical Review A* 73.1 (2006), p. 012309.
- [22] Román Orús. “A practical introduction to tensor networks: Matrix product states and projected entangled pair states”. In: *Annals of Physics* 349 (2014), pp. 117–158.
- [23] Frank Verstraete, Valentin Murg, and J Ignacio Cirac. “Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems”. In: *Advances in Physics* 57.2 (2008), pp. 143–224.
- [24] Steven R White. “Density-matrix algorithms for quantum renormalization groups”. In: *Physical Review B* 48.14 (1993), p. 10345.
- [25] Ulrich Schollwöck. “The density-matrix renormalization group in the age of matrix product states”. In: *Annals of physics* 326.1 (2011), pp. 96–192.
- [26] Sebastian Paeckel et al. “Time-evolution methods for matrix-product states”. In: *Annals of Physics* 411 (2019), p. 167998.
- [27] Guifré Vidal. “Efficient classical simulation of slightly entangled quantum computations”. In: *Physical review letters* 91.14 (2003), p. 147902.
- [28] Guifré Vidal. “Classical simulation of infinite-size quantum lattice systems in one spatial dimension”. In: *Physical review letters* 98.7 (2007), p. 070201.
- [29] Andrew John Daley et al. “Time-dependent density-matrix renormalization-group using adaptive effective Hilbert spaces”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2004.04 (2004), P04005.
- [30] Steven R White and Adrian E Feiguin. “Real-time evolution using the density matrix renormalization group”. In: *Physical review letters* 93.7 (2004), p. 076401.
- [31] Jutho Haegeman et al. “Time-dependent variational principle for quantum lattices”. In: *Physical review letters* 107.7 (2011), p. 070601.
- [32] Jutho Haegeman, Tobias J Osborne, and Frank Verstraete. “Post-matrix product state methods: To tangent space and beyond”. In: *Physical Review B* 88.7 (2013), p. 075133.
- [33] Jutho Haegeman et al. “Unifying time evolution and optimization with matrix product states”. In: *Physical Review B* 94.16 (2016), p. 165116.

- [34] Frank Verstraete and J Ignacio Cirac. “Renormalization algorithms for quantum-many body systems in two and higher dimensions”. In: *arXiv preprint cond-mat/0407066* (2004).
- [35] Hong-Chen Jiang, Zheng-Yu Weng, and Tao Xiang. “Accurate determination of tensor network state of quantum lattice models in two dimensions”. In: *Physical review letters* 101.9 (2008), p. 090603.
- [36] Philippe Corboz, Jacob Jordan, and Guifré Vidal. “Simulation of fermionic lattice models in two dimensions with projected entangled-pair states: Next-nearest neighbor Hamiltonians”. In: *Physical Review B* 82.24 (2010), p. 245119.
- [37] Henning Kalis et al. “Fate of the cluster state on the square lattice in a magnetic field”. In: *Physical Review A* 86.2 (2012), p. 022317.
- [38] Jacob Jordan et al. “Classical simulation of infinite-size quantum lattice systems in two spatial dimensions”. In: *Physical review letters* 101.25 (2008), p. 250602.
- [39] Ho N Phien et al. “Infinite projected entangled pair states algorithm improved: Fast full update and gauge fixing”. In: *Physical Review B* 92.3 (2015), p. 035142.
- [40] Ho N Phien, Ian P McCulloch, and Guifré Vidal. “Fast convergence of imaginary time evolution tensor network algorithms by recycling the environment”. In: *Physical Review B* 91.11 (2015), p. 115137.
- [41] Philippe Corboz. “Variational optimization with infinite projected entangled-pair states”. In: *Physical Review B* 94.3 (2016), p. 035133.
- [42] Laurens Vanderstraeten et al. “Gradient methods for variational optimization of projected entangled-pair states”. In: *Physical Review B* 94.15 (2016), p. 155123.
- [43] Iztok Pižorn, Ling Wang, and Frank Verstraete. “Time evolution of projected entangled pair states in the single-layer picture”. In: *Physical Review A* 83.5 (2011), p. 052321.
- [44] Claudius Hubig and J Ignacio Cirac. “Time-dependent study of disordered models with infinite projected entangled pair states”. In: *SciPost Physics* 6.3 (2019).
- [45] Glen Evenbly. “Gauge fixing, canonical forms, and optimal truncations in tensor networks with closed loops”. In: *Physical Review B* 98.8 (2018), p. 085155.
- [46] Ruben Verresen, Roderich Moessner, and Frank Pollmann. “Avoided quasiparticle decay from strong quantum interactions”. In: *Nature Physics* 15.8 (2019), pp. 750–753.
- [47] Atılım Güneş Baydin et al. *Automatic differentiation in machine learning: A survey*. 2018. arXiv: 1502.05767.
- [48] Chu Guo and Dario Poletti. *A scheme for automatic differentiation of complex loss functions*. 2020. arXiv: 2003.04295 [cs.LG].
- [49] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.1.55. 2018. URL: <http://github.com/google/jax>.
- [50] Jakob Unfried. *msc-legacy*. <https://github.com/Jakob-Unfried/msc-legacy>. 2020. URL: <https://github.com/Jakob-Unfried/msc-legacy>.
- [51] S. Nocedal J. & Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2006.
- [52] Philip Wolfe. “Convergence conditions for ascent methods”. In: *SIAM review* 11.2 (1969), pp. 226–235.

- [53] James M Ortega and Werner C Rheinboldt. *Iterative solution of nonlinear equations in several variables*. SIAM, 2000.
- [54] Jorge J Moré and David J Thuate. “Line search algorithms with guaranteed sufficient decrease”. In: *ACM Transactions on Mathematical Software (TOMS)* 20.3 (1994), pp. 286–307.
- [55] William W Hager and Hongchao Zhang. “A new conjugate gradient method with guaranteed descent and an efficient line search”. In: *SIAM Journal on optimization* 16.1 (2005), pp. 170–192.
- [56] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM, 1996.
- [57] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.
- [58] Claude Lemaréchal. “A view of line-searches”. In: *Optimization and Optimal Control*. Springer, 1981, pp. 59–78.
- [59] Matthew Johnson and Alex Wiltschko. *The Autodiff Cookbook*. July 12, 2020. URL: https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html.
- [60] Charles G Broyden. “The convergence of a class of double-rank minimization algorithms: 2. The new algorithm”. In: *IMA journal of applied mathematics* 6.3 (1970), pp. 222–231.
- [61] Roger Fletcher. “A new approach to variable metric algorithms”. In: *The computer journal* 13.3 (1970), pp. 317–322.
- [62] Donald Goldfarb. “A family of variable-metric methods derived by variational means”. In: *Mathematics of computation* 24.109 (1970), pp. 23–26.
- [63] David F Shanno. “Conditioning of quasi-Newton methods for function minimization”. In: *Mathematics of computation* 24.111 (1970), pp. 647–656.
- [64] MJD Powell. “Some global convergence properties of a variable metric algorithm for minimization without exact line searches-nonlinear Programming, Vol. 4, SIAM-AMS Proceedings”. In: *SIAM, Philadelphia, PA* (1976).
- [65] Jorge Nocedal. “Theory of algorithms for unconstrained optimization”. In: *Acta numerica* 1.1 (1992), pp. 199–242.
- [66] Dong C Liu and Jorge Nocedal. “On the limited memory BFGS method for large scale optimization”. In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.
- [67] Jorge Nocedal. “Updating quasi-Newton matrices with limited storage”. In: *Mathematics of computation* 35.151 (1980), pp. 773–782.
- [68] Jean Charles Gilbert and Claude Lemaréchal. “Some numerical experiments with variable-storage quasi-Newton algorithms”. In: *Mathematical programming* 45.1-3 (1989), pp. 407–435.
- [69] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. “Unconstrained optimization of real functions in complex variables”. In: *SIAM Journal on Optimization* 22.3 (2012), pp. 879–898.
- [70] Wilhelm Wirtinger. “Zur formalen theorie der funktionen von mehr komplexen veränderlichen”. In: *Mathematische Annalen* 97.1 (1927), pp. 357–375.
- [71] Dougal Maclaurin. “Modeling, Inference and Optimization With Composable Differentiable Procedures”. PhD thesis. Harvard University, Graduate School of Arts & Sciences, 2016.

- [72] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.
- [73] Louis B Rall and George F Corliss. “An introduction to automatic differentiation”. In: *Computational Differentiation: Techniques, Applications, and Tools* 89 (1996).
- [74] Uwe Naumann. “Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph”. In: *Mathematical Programming* 99.3 (2004), pp. 399–421.
- [75] David M Gay. “More AD of nonlinear AMPL models: computing Hessian information and exploiting partial separability”. In: *Computational Differentiation: Applications, Techniques, and Tools* (1996), pp. 173–184.
- [76] Daniele Casanova et al. “Application of automatic differentiation to race car performance optimisation”. In: *Automatic differentiation of algorithms*. Springer, 2002, pp. 117–124.
- [77] Isabelle Charpentier and Mohammed Ghemires. “Efficient adjoint derivatives: application to the meteorological model Meso-NH”. In: *Optimization Methods and Software* 13.1 (2000), pp. 35–63.
- [78] Jeffrey P Thomas, Earl H Dowell, and Kenneth C Hall. “Using automatic differentiation to create a nonlinear reduced-order-model aerodynamic solver”. In: *AIAA journal* 48.1 (2010), pp. 19–24.
- [79] Christian H Bischof, H Martin Bückner, and Bruno Lang. “Automatic differentiation for computational finance”. In: *Computational methods in decision-making, economics and finance*. Springer, 2002, pp. 297–310.
- [80] R. E. Wengert. “A Simple Automatic Derivative Evaluation Program”. In: *Commun. ACM* 7.8 (Aug. 1964), pp. 463–464. ISSN: 0001-0782.
- [81] Robert Fourer, David M Gay, and Brian W Kernighan. “AMPL. A modeling language for mathematical programming”. In: (2003).
- [82] Christian H Bischof, Lucas Roh, and Andrew J Mauer-Oats. “ADIC: an extensible automatic differentiation tool for ANSI-C”. In: *Software: Practice and Experience* 27.12 (1997), pp. 1427–1456.
- [83] Andrea Walther and Andreas Griewank. “Getting Started with ADOL-C.” In: *Combinatorial scientific computing* 09061 (2009), pp. 181–202.
- [84] Christian H Bischof et al. “Advances in automatic differentiation”. In: (2008).
- [85] Claus Bendtsen and Ole Stauning. *FADBAD, a flexible C++ package for automatic differentiation*. Tech. rep. Technical Report IMM-REP-1996-17, Department of Mathematical Modelling ..., 1996.
- [86] Alex Shtof et al. “Geosemantic Snapping for Sketch-Based Modeling”. In: *Computer Graphics Forum* 32.2 (2013), pp. 245–253. DOI: 10.1111/cgf.12044. URL: <http://diglib.eg.org/EG/CGF/volume32/issue2/v32i2pp245-253.pdf>.
- [87] Christian Bischof et al. “ADIFOR 2.0: Automatic differentiation of Fortran 77 programs”. In: *IEEE Computational Science and Engineering* 3.3 (1996), pp. 18–32.
- [88] Uwe Naumann and Jan Riehme. “Computing adjoints with the NAGWare Fortran 95 compiler”. In: *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006, pp. 159–169.

- [89] Ralf Giering and Thomas Kaminski. “Recipes for adjoint code construction”. In: *ACM Transactions on Mathematical Software (TOMS)* 24.4 (1998), pp. 437–474.
- [90] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. “Forward-mode automatic differentiation in Julia”. In: *arXiv preprint arXiv:1607.07892* (2016).
- [91] Siegfried M Rump. “INTLAB—interval laboratory”. In: *Developments in reliable computing*. Springer, 1999, pp. 77–104.
- [92] Adam Paszke et al. “Automatic differentiation in pytorch”. In: (2017).
- [93] Bart van Merriënboer, Dan Moldovan, and Alexander Wiltschko. “Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 6256–6265.
- [94] Roy Frostig, Matthew James Johnson, and Chris Leary. “Compiling machine learning programs via high-level tracing”. In: *Systems for Machine Learning* (2018).
- [95] Juraj Hasik. “Towards next-generation methods to optimize two-dimensional tensor networks : Algorithmic differentiation and applications to quantum magnets”. In: (2019).
- [96] Román Orús. “Exploring corner transfer matrices and corner tensors for the classical simulation of quantum lattice systems”. In: *Physical Review B* 85.20 (2012), p. 205117.
- [97] Román Orús and Guifré Vidal. “Simulation of two-dimensional quantum systems on an infinite lattice revisited: Corner transfer matrix for tensor contraction”. In: *Physical Review B* 80.9 (2009), p. 094403.
- [98] Johannes Hauschild and Frank Pollmann. “Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy)”. In: *SciPost Phys. Lect. Notes* (2018). Code available from <https://github.com/tenpy/tenpy>, p. 5. arXiv: 1805.00055.
- [99] Michael M Wolf et al. “Area laws in quantum systems: mutual information and correlations”. In: *Physical review letters* 100.7 (2008), p. 070502.
- [100] Andras Molnar et al. “Approximating Gibbs states of local Hamiltonians efficiently with projected entangled pair states”. In: *Phys. Rev. B* 91 (4 Jan. 2015), p. 045138.
- [101] M. B. Hastings. “Solving gapped Hamiltonians locally”. In: *Phys. Rev. B* 73 (8 Feb. 2006), p. 085115.
- [102] Frank Verstraete et al. “Criticality, the area law, and the computational power of projected entangled pair states”. In: *Physical review letters* 96.22 (2006), p. 220601.
- [103] David Perez-Garcia et al. *PEPS as unique ground states of local Hamiltonians*. 2007. arXiv: 0707.2260 [quant-ph].
- [104] Masuo Suzuki. “General theory of fractal path integrals with applications to many-body theories and statistical physics”. In: *Journal of Mathematical Physics* 32.2 (1991), pp. 400–407.
- [105] Robert NC Pfeifer et al. “NCON: A tensor network contractor for MATLAB”. In: *arXiv preprint arXiv:1402.0939* (2014).
- [106] Robert NC Pfeifer, Jutho Haegeman, and Frank Verstraete. “Faster identification of optimal contraction sequences for tensor networks”. In: *Physical Review E* 90.3 (2014), p. 033315.

- [107] Michael Lubasch, J Ignacio Cirac, and Mari-Carmen Banuls. “Unifying projected entangled pair state contractions”. In: *New Journal of Physics* 16.3 (2014), p. 033014.
- [108] Tomotoshi Nishino and Kouichi Okunishi. “Corner transfer matrix renormalization group method”. In: *Journal of the Physical Society of Japan* 65.4 (1996), pp. 891–894.
- [109] Hai Jun Liao et al. “Differentiable Programming Tensor Networks”. In: *Physical Review X* 9.3 (2019). ISSN: 21603308.
- [110] Benjamin Dauvergne and Laurent Hascoët. “The data-flow equations of checkpointing in reverse automatic differentiation”. In: *International Conference on Computational Science*. Springer. 2006, pp. 566–573.
- [111] Ziv Sirkes and Eli Tziperman. “Finite difference of adjoint or adjoint of finite difference?” In: *Monthly weather review* 125.12 (1997), pp. 3373–3378.
- [112] MSL du Croo de Jongh and JMJ Van Leeuwen. “Critical behavior of the two-dimensional Ising model in a transverse field: A density-matrix renormalization calculation”. In: *Physical Review B* 57.14 (1998), p. 8494.
- [113] Henk WJ Blöte and Youjin Deng. “Cluster Monte Carlo simulation of the transverse Ising model”. In: *Physical Review E* 66.6 (2002), p. 066110.
- [114] Michael P Zaletel and Frank Pollmann. “Isometric tensor network states in two dimensions”. In: *Physical review letters* 124.3 (2020), p. 037201.
- [115] Steven R White and Ian Affleck. “Spectral function for the $S=1$ Heisenberg antiferromagnetic chain”. In: *Physical Review B* 77.13 (2008), p. 134437.
- [116] William H Press et al. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.
- [117] Reinhold Remmert. *Theory of complex functions*. Vol. 122. Springer Science & Business Media, 1991.
- [118] Mike B. Giles. *Collected matrix derivative results for forward and reverse mode algorithmic differentiation*. Tech. rep. 2008, pp. 35–44.
- [119] James Townsend. “Differentiating the Singular Value Decomposition”. In: 1.6 (2016). URL: <https://j-towns.github.io/papers/svd-derivative.pdf>.
- [120] Zhou-Quan Wan and Shi-Xin Zhang. “Automatic Differentiation for Complex Valued SVD”. In: (2019). arXiv: 1909.02659.