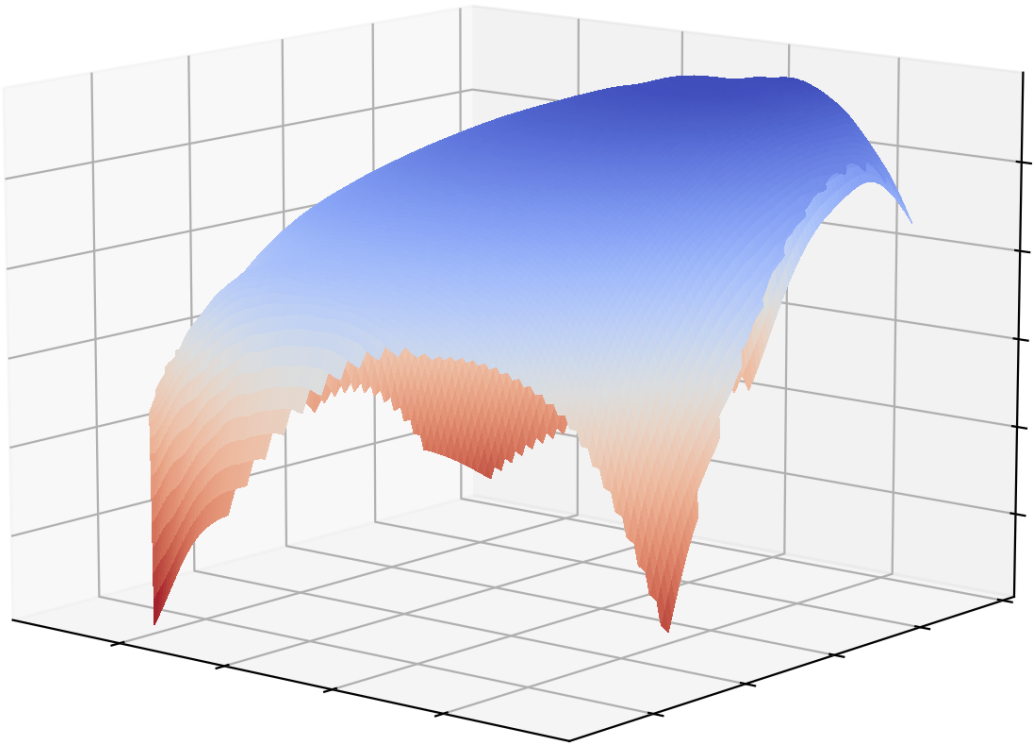
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Master's thesis

Kasper Unn Weihe PXH755

Convex Optimization and Parallel Computing for Portfolio Optimization

Supervisor: Martin Elsman

Handed in: July 3, 2023

Abstract

This thesis proposes a parallel portfolio optimization strategy that integrates environmental, social, and corporate governance (ESG) factors, while also balancing risk and return. Convex optimization techniques, particularly for constrained problems, form the basis for addressing this challenge.

To efficiently compute multi-dimensional frontiers, symbolizing optimal portfolios, the high-performance parallel programming language, Futhark [1], is utilized. The research addresses the high computational demands inherent in solving a large grid of convex optimization problems or a single, very large problem, harnessing general-purpose graphics processing units (GPGPUs). Different parallel computing strategies are investigated to enhance computational efficiency. The investigation seeks to develop a data parallel method to solve convex optimization problems, with a focus on portfolio optimization. The research aims to investigate how convex optimization can be parallel and applied to portfolio management.

The findings of this research are anticipated to provide valuable contributions to the domains of finance and computer science, presenting insights into the application of convex optimization techniques and parallel computing for investment strategies. The relevance of solving large multi-dimensional optimization problems extends beyond portfolio optimization and can be applied to a multitude of problems in diverse fields. Additionally, this thesis examines the efficacy of Futhark as a high-performance parallel programming language for solving convex optimization problems. Three Futhark modules are presented; the first one is for solving linear systems of equations, which is used as a building block for the second module, a convex optimization module, capable of solving multi-dimensional optimization problems, while the third is a portfolio optimization module, tailored to optimize portfolios based on various constraints, which is implemented using the aforementioned convex optimization module.

Keywords: Convex optimization, Portfolio Optimization, Futhark

Contents

Chapter 1

Introduction

There is a growing interest in incorporating environmental, social, and governance (ESG) factors into investment decisions. Many exchange traded funds (ETFs) and mutual funds are now incorporating ESG factors into their investment strategies [2]. This thesis aims to explore the application of convex optimization techniques and the data parallel programming language Futhark [1] for solving convex optimization problems, focusing on portfolio optimization as a case study. The research emphasizes the importance of parallel computing strategies, leveraging many-core general-purpose graphics processing units (GPGPUs) to optimize computational efficiency. Solving optimization problems can be computationally expensive, especially when dealing with a large number of variables or a large number of optimization problems. The research aims to investigate how we can use parallelization to solve portfolio/convex optimization problems more efficiently. Furthermore, the study aims to evaluate the effectiveness of Futhark as a high-performance parallel programming language for solving multi-dimensional optimization problems. Optimization is not only used for portfolio optimization but also for many other problems in all kinds of fields. The findings should be transferable to other fields.

1.1 Related Work

Convex optimization problems are common and can be tackled using a variety of mathematical techniques and algorithms. Fundamental to these are Newton's Method and Gradient Descent. Additionally, more advanced techniques, such as the Alternating Direction Method of Multipliers (ADMM), exist. ADMM is an advanced optimization technique that breaks down complex or large-scale convex problems into smaller, easier-to-solve subproblems. It is a form of decomposition coordination technique that has seen widespread application in various areas due

to its ability to handle problems efficiently.

Several software libraries and frameworks, such as CVXPY [3], facilitate the application of these methods to solve optimization problems. CVXPY, an embedded Python modeling language designed specifically for convex optimization problems, leverages these mathematical techniques to achieve solutions.

Thus, while CVXPY will be utilized as a baseline for convex and portfolio optimization, it is important to acknowledge that it essentially encapsulates and applies mathematical techniques such as ADMM to provide optimized solutions.

1.2 Thesis Objective

The primary objective of this thesis is to develop data-parallel strategies aimed at solving large convex optimization problems. The thesis will focus on portfolio optimization with ESG data as a case study. As such, we will investigate how the efficiency of a convex optimizer written in Futhark compares to established optimizers such as CVXPY for solving many optimization problems in parallel or a single optimization problem with a large number of variables.

1.3 Thesis Structure

Chapter 2 provides a comprehensive overview of the data-parallelism concept and introduces Futhark. Moving forward to Chapter 3, the principles underpinning portfolio optimization are discussed. In Chapter 4, we delve into convex optimization, discussing its theory and various algorithms that effectively tackle convex optimization issues.

Chapter 5 pivots to detail different algorithms devised to address linear systems, forming an integral foundation for solving convex optimization problems. Subsequently, Chapter 6 unveils the design and implementation of three distinct Futhark modules: the first aimed at solving linear systems of equations, the second dedicated to resolving convex optimization problems, and the third devoted to portfolio optimization.

Chapter 7 contains benchmarking and tests of these modules with CVXPY. Bringing this thesis to a close, Chapter 8 presents the conclusion, summarizing the findings and suggesting areas for potential future research.

Chapter 2

Data-Parallel Programming and Futhark

Data parallelism is a type of parallel computing where the same set of operations are performed simultaneously on multiple processing units or cores, each with a different subset of the data. This technique is particularly useful when processing large datasets, as it splits the data across different cores, allowing computations to be done in parallel, potentially leading to faster execution times. In this chapter, we will introduce the data-parallel programming language Futhark, which is used to implement the methods presented in this thesis. It is worth noting that Futhark is a tool for expressing data-parallel programs. As such, the parallelization techniques presented in this thesis are not limited to Futhark but can be applied to other data-parallel programming languages as well. The goal of this thesis is to investigate the parallelization of convex optimization problems, using Futhark as a tool to express parallelism.

2.1 Introduction to Futhark

Futhark is a data-parallel functional programming language designed to be compiled to efficient parallel code through OpenCL or CUDA to run on GPUs [4]. However, Futhark can also run on other hardware, such as multicore CPUs. Futhark is a pure functional and statically typed programming language in the ML family. The programs consist primarily of Second-Order Array Combinators (SOACs) such as map, reduce, and scan. It is worth noting that Futhark is not parallel by default. The programmer has to explicitly construct the programs to be parallel.

As mentioned, Futhark uses SOACs to express parallelism. SOACs are higher-

order functions that perform bulk operations on one or more arrays. The most common SOACs are map, reduce, and scan. Map applies a function to each element of an array. Reduce applies a binary associative operator/function to elements of an array. Scan is similar to reduce, but it returns an array of intermediate results. Using these SOACs, we can implement functions for dot product and matrix multiplication.

```
1 def dotproduct [n] (u: [n]f64) (v: [n]f64): f64 =
2   reduce (+) 0.0 (map2 (*) u v)
3
4 def matmul [n][p][m] (xss: [n][p]t) (yss: [p][m]t): *[n][m]t =
5   map (\xs -> map (dotprod xs) (transpose yss)) xss
```

Listing 2.1: Futhark code for dot product and matrix multiplication

The [n][p][m] preceding the value parameters (xss and yss) are referred to as size specifiers, which allow us to specify the dimensions of the value parameters. For example, for matrix multiplication, we must have that the number of columns in the first matrix is equal to the number of rows in the second matrix. The size parameter lets us express this constraint in the type of the function and the constraint is checked at compile time [5].

Futhark has a series of rules that enable it to generate efficient code while maintaining correctness. For instance, irregular arrays are prohibited in the language using size constraints, and the operator used in a reduce and scan operation must be associative to ensure that the result is independent of the order of the operations when parallelized.

In order to reduce and scan in parallel the operator must be associative and have a neutral element.

It is the responsibility of the developer to ensure that the operator used in a reduce and scan operation is associative. If the operator is not associative, the result of the operation will likely not be what the developer intended because "the order" of the operations is not guaranteed.

2.1.1 A Parallel Cost Model for Futhark Programs

An algorithm's dependencies between operations can be effectively represented using a directed acyclic graph (DAG), where each computational unit is illustrated as a node, and the directional arcs demonstrate their dependencies [6]. This representation results in a connected, acyclic tree structure where the root node represents the output of the algorithm, and its child nodes symbolize dependencies. Each layer of this tree, denoted by nodes, allows for concurrent computation

of operations, as no computation within the same layer depends on another. However, operations across different layers are unable to run in parallel due to their dependencies. The execution of the algorithm begins sequentially from the leaves of the tree, which are free of prior dependencies and progresses toward the root node.

The Futhark programming language utilizes a parallel cost model to estimate a program or an algorithm's runtime, which is based on the algorithm's work and span [7]. The **work**, equivalent to the number of nodes in the DAG, is the cumulative count of primitive operations executed by the program, and it proportionally impacts the total execution time. On the other hand, the **span** of a program signifies the length of the longest dependency chain between primitive operations, forming a theoretical lower bound on the program's execution time, irrespective of the number of processors. The span is crucial because regardless of the processor count, the program cannot complete faster than this span as these dependent operations must be executed sequentially. If T_1, T_p, and T_∞ are the execution times of a program on a single processor, p processors, and an infinite number of processors, respectively, then using Brent's theorem, we can express the execution time of a program as follows [6]:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty \tag{2.1}$$

Given that $\frac{T_1}{p}$ is optimal, it becomes evident that T_∞ indeed provides a valuable perspective on the extent to which our algorithm's performance deviates from the optimal version of the parallel algorithm. Essentially, T_∞ can be viewed as a metric to gauge the level of parallelism in an algorithm. Observe Listing 2.1 where we take the dot product of two vectors u and v of size n. For each element in the vectors u and v, we multiply them together and add the result to the sum variable. This operation is repeated n times, and thus the work is $\mathcal{O}(n)$. However, all of the operations of a map are independent of each other, and thus the span of the map is $\mathcal{O}(1)$. However, the analysis of reduce is a little more involved. The reduce operation is a tree reduction, where the tree is constructed by repeatedly applying the associative operator to pairs of elements. The height of the tree is $\mathcal{O}(\log n)$, and thus the span is $\mathcal{O}(\log n)$ and is efficiently parallelized in Futhark. Furthermore, Futhark applies all sorts of optimizations to the programs, such as loop fusion, loop tiling, and loop interchange [1].

2.1.2 Auto-differentiation in Futhark

Auto-differentiation is a technique for automatically computing the derivative of a function. It is relatively new in Futhark and is still under active development as the rest of the language [8]. Futhark has two modes of auto-differentiation, forward

mode, and reverse mode, through the built-in functions `jvp` and `vjp`, respectively. Forward mode uses forward accumulation to compute the derivative of a function. Reverse mode uses reverse accumulation to compute the derivative of a function. Reverse mode should be more efficient than forward mode for functions with many inputs and few outputs and vice versa for forward mode. This is useful for computing the gradient, Hessian, or jacobian of a function. Which we will use extensively in this thesis for solving optimization problems. The performance of the auto-differentiation will be compared to the performance of the hand-written derivatives in chapter 7. Here is a short example of auto-differentiation of the `sqrt` function in Futhark.

```
1 def f x = f64.sqrt x
2 def f' x = jvp f x 1
3 > f' 2f64
4 0.353553f64
```

In this example, we compute the derivative of the `sqrt` function at $x = 2$. The result is 0.353553. `jvp` can be exchanged with `vjp` to compute the derivative of the function using reverse accumulation.

2.1.3 Modularity in Futhark

Futhark has an ML-style higher-order module system [9]. It features module types that allow us to define what a module contains. These include detailed type descriptions of what should be implemented within a module. We can, for instance, define a module type `M` as follows:

```
1 module type M = {
2   type t
3   val add: t -> t -> t
4 }
```

It specifies a type `t` and a function `add` that takes two values of type `t` and returns a `t`. Futhark has parametric modules, which can take other modules as arguments, and thus we define a module `m` that takes a module of type `real` as argument:

```
1 module m (R: real): M = {
2   type t = R.t
3   def add (a: t) (b: t): t = a R.+ b
4 }
```

Here we use the + operator of the `real` module to implement the `add` function. The real module is a builtin module for real numbers (f32, f64, etc.) in Futhark. We can then use the module `m` as follows:

```
1 import "m"
2 module m64 = m f64
3 > m64.add 1.0f64 2.0f64
```

Chapter 3

Portfolio Optimization

Portfolio optimization is a concept in financial theory focusing on the maximization of expected return for a given level of risk or the minimization of risk for a given level of expected return. This chapter dives into the theory of portfolio optimization, examining its theoretical foundations, practical applications, and the role of computational techniques in facilitating such optimizations.

In 1952 Harry Markowitz published a paper on portfolio selection in The Journal of Finance, which laid the foundations for modern portfolio theory [10], which he was later awarded the Nobel Prize in Economics for in 1990 [11]. The paper presents a mathematical framework for selecting a portfolio of investments that maximizes expected return while minimizing risk. Markowitz divides the process of portfolio selection into two stages. The first stage is forming relevant beliefs about the future of securities based on observations of the past, and the second stage is the optimization of the portfolio based on those beliefs. The paper focuses on the second stage, which is the optimization of the portfolio. This aspect is also the focus of this thesis.

Markowitz introduces the concept of efficient portfolios, also known as the efficient frontier. An efficient portfolio is a portfolio that maximizes expected return for a given level of risk or minimizes risk for a given level of expected return. The efficient frontier is the set of all efficient portfolios.

Imagine Y is a variable whose value is determined randomly, and it can only be one of several specific values (y_1 to y_N). Each of these values has a probability (p_1 to p_N). The average (or mean) of Y is the sum of each value multiplied by its probability. The variance of Y is a way to measure how spread out the values are. It is the sum of the square of the difference between each value and the mean, each multiplied by its probability.

Now, suppose we have a set of random variables: $R_1, \ldots, R_n$ and R is a weighted sum of the R_i, and hence, is also a random variable. a_i is the weight of R_i in the sum.

$$R = a_1 R_1 + a_2 R_2 + \ldots + a_n R_n \tag{3.1}$$

However, the variance of R is more complex and involves the concept of 'covariance,' which measures how two variables change together. Covariance can be expressed in terms of the correlation coefficient ρ_{ij}, a measure of the strength and direction of the relationship between two variables. We define the covariance of R_i and R_j as:

$$\sigma_{ij} = \rho_{ij} \sigma_i \sigma_j \tag{3.2}$$

σ_i and σ_j are the standard deviations of R_i and R_j respectively. The variance of the weighted sum is then:

$$V(R) = \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j \sigma_{ij} \tag{3.3}$$

Now, if we let R_i represents the return on the i'th security and X_i is the percentage of assets allocated to that security, the total return R is the sum of the returns on each security multiplied by their allocation.

$$R = \sum_{i=1}^{n} R_i X_i \tag{3.4}$$

$$\sum_{i=1}^{n} X_i = 1 \tag{3.5}$$

The expected return of the portfolio is the sum of the expected returns on each security, each multiplied by their allocation. The variance (risk) of the portfolio is the sum of the covariances of each pair of securities, each pair multiplied by their respective allocations. Let μ_i be the expected return of R_i and σ_{ii} be the variance of R_i. The expected return and variance of the portfolio are then:

$$E = \sum_{i=1}^{N} X_i \mu_i \tag{3.6}$$

$$V = \sum_{i=1}^{N} \sum_{j=1}^{N} \sigma_{ij} X_i X \tag{3.7}$$

Given a set of fixed probability beliefs represented as (μ_i, σ_{ij}), the investor has the ability to select different combinations of expected return (E) and variance (V) by varying the composition of the portfolio weights $X_1, \ldots, X_N$. We assume no short selling (negative weights) and no borrowing, so the sum of the weights must be 1. Imagine that all possible combinations of (E, V) are illustrated in Figure 1. The EV principle suggests that the investor ought to choose one of those portfolios which are represented as efficient (red) in the figure [10]. That is, the portfolios that offer the least variance V for a given or higher expected return E, or the portfolios which provide the highest expected return E for a given or lower variance V.

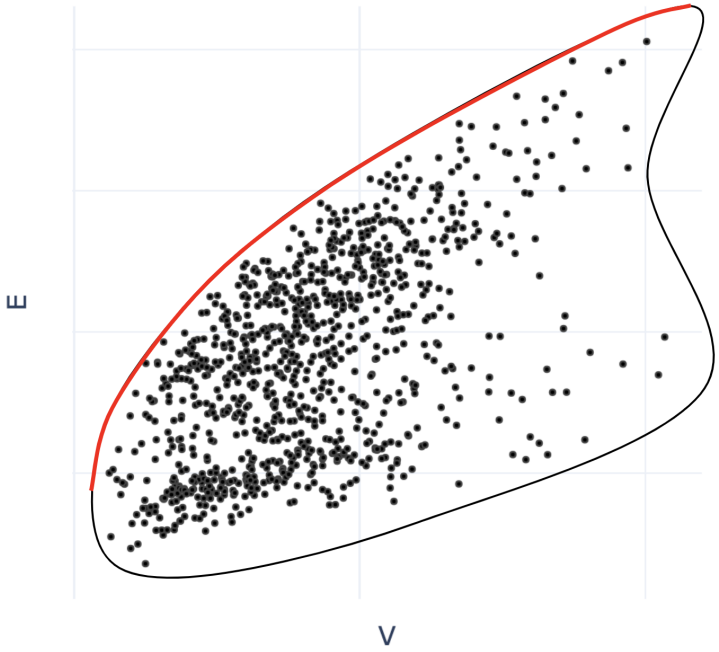


Figure 3.1: The black dots represent random portfolios. The red line represents the efficient frontier. The black outline is a rough sketch of the set of all possible portfolios.

In this thesis, we will focus on the optimization of the portfolio weights $X_1, \ldots, X_N$, given a set of fixed probability beliefs represented as (μ_i, σ_{ij}), but we also want to take into account other parameters such as the ESG score of the involved companies. The ESG score is a measure of the sustainability and ethical impact of a company. The ESG score is a number between 0 and 100, where 0 is the worst possible score, and 100 is the best possible score. Instead of maximizing the expected return at a given level of risk, we want to maximize the expected return at a given level of risk and ESG score. The optimization problem can be formulated as a convex optimization problem.

Chapter 4

Convex Optimization

Convex optimization, a specialized branch of mathematical optimization, focuses on minimizing convex functions within the boundaries of convex sets [12]. The concepts of 'convex function' and 'convex set' are essential to grasp this idea. Specifically, in a real vector space, a set is considered convex if it encompasses the entire line segment joining any two points within the set. Similarly, A function is convex if, for any two points in its domain, the function evaluated at any point on the line segment joining these two points is less than or equal to the weighted average of the function values at the two points [12]. In other words, a function is convex if, for any two points x and y in its domain and for any t in the range $[0, 1]$, the following condition is satisfied:

$$f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y) \tag{4.1}$$

This property can be represented graphically as the function lying below the line segment connecting the points $(x, f(x))$ and $(y, f(y))$ for all x and y in the domain of the function. If this condition is satisfied for a function, it is said to be convex. Conversely, if the inequality is reversed, the function is said to be concave.

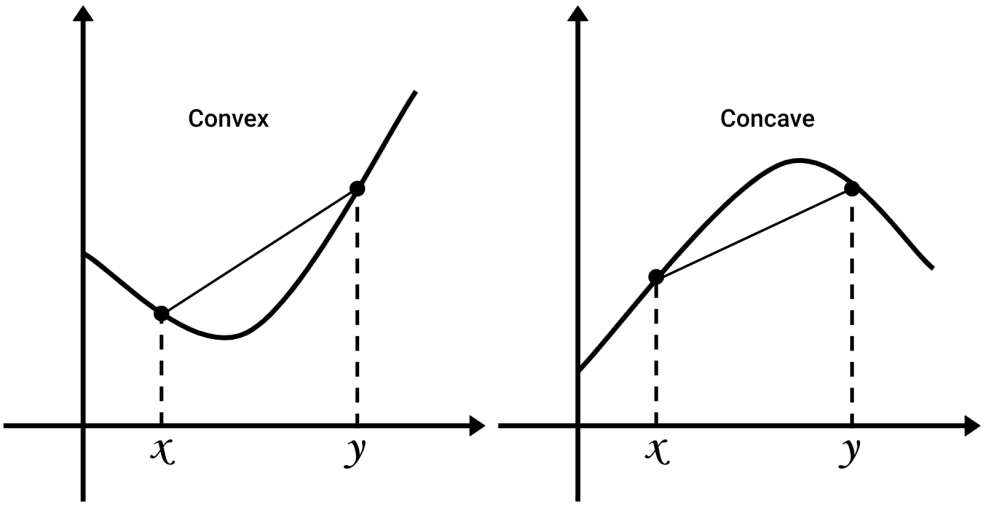


Figure 4.1: A convex and a concave function. The convex function lies below the line segment connecting the points $(x, f(x))$ and $(y, f(y))$ for all x and y in the domain of the function. The concave function lies above the line segment connecting the points $(x, f(x))$ and $(y, f(y))$ for all x and y in the domain of the function. This generalizes to higher dimensions.

Convexity gives the function a 'bowl-shaped' or 'U-shaped' curve, which facilitates efficient optimization because there is only one local minimum, which is also the global minimum. Convex optimization problems can be visualized as the problem of finding the lowest point in a landscape where it is such that if you start at any point and move downhill, you are guaranteed to arrive at the lowest point. This property makes convex optimization problems particularly appealing, as they avoid the problem of local minima that plague many optimization problems. In the realm of practical applications, convex optimization plays an essential role in several disciplines, such as machine learning, computer science, statistics, finance, engineering, economics, and more. Algorithms for convex optimization include gradient descent and its variants, Newton's method, interior-point methods, and others [12].

However, not all optimization problems are convex. Many non-convex optimization problems can be approximated or reformulated as convex ones, making convex optimization a very important tool in the field of optimization. Also, convex optimization problems constitute a broad class of problems that include linear programming problems and least-squares problems [12].

4.1 Convex Optimization Problem

A convex optimization problem is a problem of the form:

$$\text{minimize } f_0(x) \tag{4.2}$$
$$\text{subject to } f_i(x) \leq 0, i = 1, \ldots, m \tag{4.3}$$
$$h_i(x) = 0, i = 1, \ldots, p \tag{4.4}$$

where $x \in \mathbb{R}^n$ is the optimization variable, $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $f_i : \mathbb{R}^n \to \mathbb{R}$ are the inequality constraint functions, and $h_i : \mathbb{R}^n \to \mathbb{R}$ are the equality constraint functions. The following requirements must be met for the problem to be a convex optimization problem [12]:

- The objective function f_0 must be convex.
- The inequality constraint functions f_i must be convex.
- The equality constraint functions h_i must be affine ($h_i(x) = ax + b$), for some a and b.

4.2 Unconstrained Convex Optimization

An unconstrained convex optimization problem is a convex optimization problem of the form:

$$\text{minimize } f(x) \tag{4.5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function. These problems are relatively easy to solve since there are no constraints to take into account. The solution (if one such exists) to the problem is the global minimum of the function f. This solution can be identified by finding the point where the gradient of the function is zero or approximately zero. Consequently, the minimum of such a function is located at the point where the first derivative, often referred to as the gradient, equals zero. This criterion marks the position where the rate of change of the function transitions from negative to non-negative, thereby establishing a minimum.

4.2.1 Gradient Descent

Descent methods are a class of algorithms for solving unconstrained convex optimization problems. The idea is to start at an initial point $x^{(0)}$ and then iteratively move in the direction of the negative gradient. At each iteration, we calculate a new point $x^{(k+1)}$ by moving in the direction of the negative gradient of the function f at the current point $x^{(k)}$. This is represented as:

$$x^{(k+1)} = x^{(k)} + t^{(k)} \Delta x^{(k)}$$

Here, $\Delta x^{(k)}$ is a vector indicating the direction we should move, also called the 'search direction'. The $t^{(k)}$ is a number that tells us how far to move in that direction, which we call the 'step size'. We always want this number to be positive unless we have already found the optimal point.

Now, the aim is to find the smallest value of a function $f(x)$. The method is working as expected if the function value at our new point, $f(x^{(k+1)})$, is less than the function value at our previous point, $f(x^{(k)})$. In other words, we want to keep moving to points where the function value is decreasing unless we have found the optimal point. A common choice for the search direction is the negative gradient of the function, $-\nabla f(x^{(k)})$. The resulting algorithm is called gradient descent:

Algorithm 1 Gradient Descent Method

1: **Given** a starting point $x \in \operatorname{dom} f$.
2: **repeat**
3: $\quad \Delta x := -\nabla f(x)$
4: $\quad$ **Line search.** Choose step size t via exact or backtracking line search.
5: $\quad x := x + t\Delta x$
6: **until** stopping criterion is satisfied

The stopping criterion is usually that the norm of the gradient is less than some small number ϵ, i.e., $|\nabla f(x)| \leq \epsilon$. This means that the algorithm will stop when the gradient is close to zero, that is when we are close to the optimal point [12]. Notice that the algorithm requires a line search. This selection of the step size t determines where along the line $\{x + t\Delta x \mid t \in \mathbf{R}_+\}$ the next iterate will be. If the step size is too small, the algorithm will take a long time to converge. If the step size is too large, it might overshoot the minimum and end up with a larger function value than before, potentially leading to non-termination as visualized in Figure 4.2.

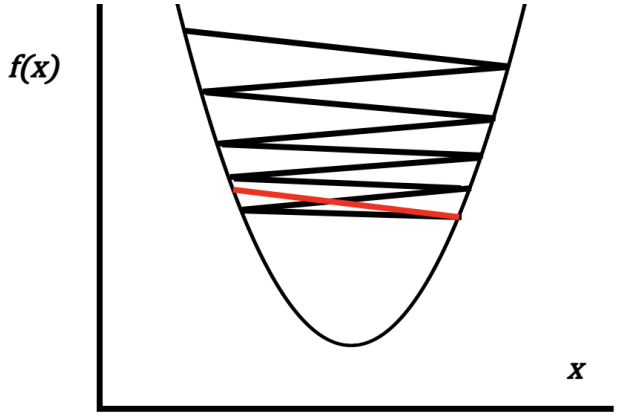


Figure 4.2: Gradient descent with a step size that is too large. The algorithm overshoots and ends up with a larger function value than before (marked with a red line). This can lead to non-termination or slow convergence.

In Algorithm 1, we use backtracking line search, which is a simple and effective line search method to find a suitable step size t:

Algorithm 2 Backtracking Line Search

Given $\alpha \in (0, 0.5), \beta \in (0, 1)$.
Input $x \in \operatorname{dom} f$.
Input $\Delta x \in \mathbb{R}^n$ with $\nabla f(x)^T \Delta x < 0$.
Input $t := 1$.
while $f(x + t\Delta x) > f(x) + \alpha t \nabla f(x)^T \Delta x$ **do**
    $t := \beta t$
end while
return t.

The aim of backtracking line search is to find a step size t that satisfies the Armijo-Goldstein condition, which ensures sufficient decrease in the function value [13]. The condition is:

$$f(x + t\Delta x) \leq f(x) + \alpha t \nabla f(x)^T \Delta x \tag{4.6}$$

$f(x+t\Delta x)$ represents the value of the function f at a new point which is a step $t \cdot \Delta x$ away from the current point x. The right-hand side of this inequality represents a linear approximation to the function f at the point x, scaled by a factor of t along the direction given by Δx. The parameter α is a constant that controls the sufficient decrease level.

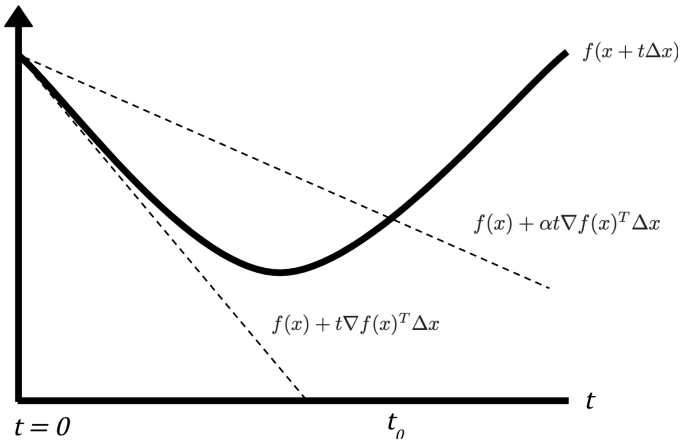


Figure 4.3: Backtracking line search: Lower and upper dashed lines represent f's linear extrapolation and a slope smaller by α respectively. f must fall under the upper dashed line, with $0 \leq t \leq t_0$.

The Armijo-Goldstein condition ensures the function value at a new point is no greater than the original point's function value, offset by a parameter α. Constant α indicates the sufficient decrease level, while β controls step size reduction if the condition isn't satisfied. Starting from step size $t = 1$, the algorithm reduces t by β until the condition is met, then returns this t.

4.2.2 Newton's method

Newton's method is an iterative algorithm used to find the roots or minima of a function. It involves calculating the Newton step and updating the current approximation to minimize the function, guided by a specific search direction and a stopping criterion. The Newton step, Δx_{nt}, for a function f at a point x is given by the equation $\Delta x_{\text{nt}} = -\nabla^2 f(x)^{-1} \nabla f(x)$. This step can be seen as a descent direction unless x is already optimal [1]. It is calculated by minimizing the second-order Taylor approximation (also known as the quadratic model) of f at x [14]. $\nabla f(x)$ is the gradient of f at x, and $\nabla^2 f(x)$ is the Hessian matrix of f at x. The gradient of f at x is the vector of partial derivatives of f at x, and the Hessian matrix is the matrix of second-order partial derivatives of f at x:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{4.7}$$

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{4.8}$$

When the Newton method is applied to a twice differentiable function, the Newton step provides a very good estimate for the minimizer of the function if the function is approximately quadratic around the point x [14].

A key component of Newton's method is the Newton decrement, $\lambda(x)$, which is calculated as $\lambda(x) = \left(\nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x)\right)^{1/2}$. This quantity is significant in the analysis of Newton's method and provides a stopping criterion for the algorithm. The value of $\lambda^2/2$ is an estimate of how far the function value at the current approximation x is from the optimal value of the function, p^* [14]. The Newton method can be represented by the following algorithm:

Algorithm 3 Newton's Method

1: **Given** a starting point $x \in \operatorname{dom} f$, tolerance $\epsilon > 0$.
2: **repeat**
3: **Compute the Newton step and decrement.**
4: $\Delta x_{\mathrm{nt}} := -\nabla^2 f(x)^{-1} \nabla f(x); \quad \lambda^2 := \nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x).$
5: **Stopping criterion.** quit if $\lambda^2/2 \leq \epsilon$.
6: **Line search.** Choose step size t by backtracking line search.
7: **Update.** $x := x + t\Delta x_{\mathrm{nt}}$.
8: **until** stopping criterion is satisfied

These steps are repeated until the stopping criterion is met [14]. Newton's method often has faster convergence than gradient descent because it takes into account the curvature of the function. The downside of Newton's method is that it can be computationally expensive, particularly in high-dimensional spaces. This is because it requires calculating and inverting the Hessian matrix, which can be costly both in terms of memory and compute time.

The convergence rate for gradient descent depends on many factors, but it has been proven to have sublinear convergence under the assumption that f is convex [15]. Newton's method, on the other hand, has quadratic convergence for convex functions, which means that the error reduces by a constant factor squared at each iteration. This means that Newton's method will likely converge with many fewer iterations than gradient descent, but each step is also more computationally expensive [12].

4.3 Equality constrained convex optimization

An equality-constrained convex optimization problem presents a unique type of optimization challenge. Specifically, this problem format is expressed as follows:

$$\text{minimize } f(x) \tag{4.9}$$
$$\text{subject to } Ax = b \tag{4.10}$$

Here, $f : \mathbb{R}^n \to \mathbb{R}$ is a convex function, $A \in \mathbb{R}^{p \times n}$ denotes a matrix, and $b \in \mathbb{R}^m$ symbolizes a vector. The matrix A has a rank of m with the condition $m \leq n$, thereby implying that the quantity of constraints does not exceed the number of variables. To solve such a problem, one may utilize Newton's method, incorporating the constraints into the calculation. The initial step involves defining the Newton step for an unconstrained optimization situation:

$$\Delta x_{\text{nt}} = -\nabla^2 f(x)^{-1} \nabla f(x) \tag{4.11}$$

However, when constraints are present, as in our case with the $Ax = b$ stipulation, a direct application of the Newton step, as is done in unconstrained optimization, is insufficient. Instead, the Newton step, when subjected to the equality constraint, is defined by the following system of equations:

$$\begin{bmatrix} \nabla^2 f(x) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_{\text{nt}} \\ w \end{bmatrix} = \begin{bmatrix} -\nabla f(x) \\ 0 \end{bmatrix}$$

In this system, w represents a vector of Lagrange multipliers. In this context, the Lagrange multipliers w act as indicators for the 'tightness' of the constraints. They provide valuable information about the solution: if a multiplier associated with a constraint is zero, then that constraint is nonbinding at the solution; that is, it does not actively constrain the solution. On the other hand, if a multiplier is non-zero, the corresponding constraint is binding at the solution, meaning it actively shapes the solution [12].

This system of equations is known as the Karush-Kuhn-Tucker (KKT) system, which is an essential condition for attaining optimality. As a system of linear equations, the KKT system can be solved using various methods, such as Gaussian elimination or LU decomposition. Detailed analysis and discussion on these methods will be provided in Chapter 5. The resulting algorithm is called Newton's method with equality constraints:

Algorithm 4 Newton's Method for Equality Constrained Minimization

Given starting point $x \in \operatorname{dom} f$ with $Ax = b$, tolerance $\epsilon > 0$.
repeat
    Compute the Newton step and decrement $\Delta x_{nt}, \lambda(x)$.
    Stopping criterion. quit if $\lambda^2/2 \leq \epsilon$.
    Line search. Choose step size t by backtracking line search.
    $x := x + t\Delta x_{nt}$
until Stopping criterion is met

The algorithm starts with a starting point x and a tolerance ϵ. It then computes the Newton step and decrement, which are used to determine whether the algorithm should stop. If the algorithm should not stop, it then performs a line search to determine the step size t to take. The algorithm then updates the current point x by adding the Newton step multiplied by the step size t. The algorithm repeats this process until the stopping criterion is met. The algorithm, as described here, is a feasible descent method, meaning that the initial point x must satisfy $Ax = b$ and all subsequent iterates must satisfy $Ax = b$, with $f(x^{(k+1)}) < f(x^{(k)})$ unless $x^{(k)}$ is optimal [12].

4.4 Inequality constrained convex optimization

An inequality constrained convex optimization problem is a convex optimization problem of the form:

$$\text{minimize } f_0(x) \tag{4.12}$$
$$\text{subject to } f_i(x) \leq 0, \quad i = 1, \ldots, m \tag{4.13}$$
$$Ax = b, \tag{4.14}$$

where $f_0, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$ are convex functions, $A \in \mathbb{R}^{p \times n}$ is a matrix, and $b \in \mathbb{R}^p$ is a vector. The rank of A is p and $p \leq n$, meaning that the number of constraints is less than or equal to the number of variables.

4.4.1 Barrier method

Interior point methods are a class of algorithms for solving inequality-constrained convex optimization problems. The Barrier method is an interior point method that solves the problem by solving a sequence of equality-constrained problems

by making the inequality constraints implicit in the objective function. The logarithmic barrier method solves the following sequence of problems:

$$\text{minimize } tf_0(x) + \phi(x) \tag{4.15}$$
$$\text{subject to } Ax = b \tag{4.16}$$

where $t > 0$ is a parameter that determines how close to the boundary of the feasible region the solution will be and $\phi(x)$ is the logarithmic barrier function:

$$\phi(x) = -\sum_{i=1}^{m} \log\left(-f_i(x)\right) \tag{4.17}$$

As t approaches infinity, the solution approaches the solution of the original problem. The logarithmic barrier method solves this sequence of problems using Newton's method with equality constraints, which was discussed in the previous section. The algorithm is represented as:

Algorithm 5 Barrier Method

1: **Given** strictly feasible x, $t := t^{(0)} > 0$, $\mu > 1$, and tolerance $\epsilon > 0$.
2: **repeat**
3: **Centering step.** Compute $x^\star(t)$ by minimizing $tf_0+\phi$, subject to $Ax = b$, starting at x.
4: **Update.** $x := x^\star(t)$.
5: **Stopping criterion.** Quit if $m/t < \epsilon$.
6: **Increase** t**.** $t := \mu t$.
7: **until** stopping criterion is satisfied

Here t is increased by a factor of μ at each iteration. The stopping criterion is satisfied when the duality gap is less than ϵ. The duality gap is defined as m/t, where m is the number of inequality constraints. The duality gap is a measure of how far the current solution is from the optimal solution. As t increases, the duality gap decreases, and the solution approaches the solution of the original problem. μ is a parameter that determines how quickly t increases. With a larger value of μ, we will expect fewer iterations of the outer loop but more iterations of the inner loop and vice versa for a smaller value of μ.

The barrier method is a feasible descent method, meaning that the initial point x^0 must be feasible, that is. It must satisfy all of the constraints. When such a point is not known, the barrier method is preceded by a phase I method, which finds a

strictly feasible point, that is, a point that satisfies all of the constraints strictly. We form the following optimization problem to find a strictly feasible point:

$$\text{minimize } s \tag{4.18}$$

$$\text{subject to } f_i(x) \leq s, \quad i = 1, \ldots, m \tag{4.19}$$

$$Ax = b \tag{4.20}$$

Where s is a slack variable, this problem is feasible only if the original problem is feasible. We can solve this problem using the barrier method, and if the optimal value of s is zero or negative, then the original problem is feasible. If the optimal value of s is positive, then the original problem is infeasible. In this case, we can use the optimal value of x as a strictly feasible point for the original problem. This problem also needs a strictly feasible starting point, which can be found by solving $Ax = b$ for x to get $x^{(0)}$ and setting s to any number greater than $\max_{i=1,\ldots,m} f_i\left(x^{(0)}\right)$.

4.4.2 Alternating direction method of multipliers

The Alternating Direction Method of Multipliers (ADMM) is an algorithm that is commonly used to solve optimization problems with equality constraints. The general form of the problem that ADMM is aimed at solving is expressed as follows [16]:

$$\text{minimize } f(x) + g(z) \tag{4.21}$$

$$\text{subject to } Ax + Bz = c \tag{4.22}$$

Where $x \in \mathbf{R}^n$ and $z \in \mathbf{R}^m$ are variables, $A \in \mathbf{R}^{p\times n}, B \in \mathbf{R}^{p\times m}$, and $c \in \mathbf{R}^p$. Here f and g are assumed to be convex. This problem is set up with the objective function separable across the splitting of the variable [16]. The method of ADMM uses the augmented Lagrangian for the problem:

$$L_\rho(x, z, u) = f(x) + g(z) + u^T(Ax + Bz - c) + (\rho/2)\|Ax + Bz - c\|_2^2 \tag{4.23}$$

Where $\rho > 0$ is the augmented Lagrangian parameter. The algorithm then consists of alternating minimization steps with respect to x and z, and a dual variable u [16]:

$$x^{(k+1)} := \arg\min_x L_\rho\left(x, z^{(k)}, u^{(k)}\right) \tag{4.24}$$

$$z^{(k+1)} := \arg\min_z L_\rho\left(x^{(k+1)}, z, u^{(k)}\right) \tag{4.25}$$

$$u^{(k+1)} := u^{(k)} + \rho\left(Ax^{(k+1)} + Bz^{(k+1)} - c\right), \tag{4.26}$$

ADMM can also be applied to solve optimization problems with inequality constraints. The approach involves transforming the inequality constraint into an equality constraint by introducing a penalty function and using variable substitution [17]. For instance, a constraint of the form $Ax \geq 0$ can be rewritten as $Ax - z = 0$ where $z \geq 0$. The constraint is hereby incorporated into the objective function, and the ADMM procedure is then applied as usual. The transformed problem can be expressed as [17]:

$$\arg\min_{x,y} f(x) + I(z) \tag{4.27}$$

$$Ax - z = 0 \tag{4.28}$$

where $I(z)$ is an indicator function that takes the value $+\infty$ for $z < 0$ and 0 for $z \geq 0$. The inequality constrained problem is then converted into an augmented Lagrangian (primal-dual) problem and solved using ADMM [17]. In the case of inequality constraints, the solution to the first primal descent step, also known as the x-update step, can be obtained using a Newton update. This update uses the gradient and Hessian of the original objective function, which are assumed to be known, and the current values of z and u [17]. The Newton update is given by [17]:

$$x^{k+1}_{\text{Newton}} = \left(h + \rho A^T A\right)^{-1} \left(g + \rho A^T \left(Ax^k - z^k + u^k\right)\right) \tag{4.29}$$

where h and g are the Hessian and gradient of the original objective function, respectively. It is important to note that the Newton update step typically assumes the function $f(x)$ is twice differentiable and convex. The algorithm for ADMM with inequality constraints $(Ax \geq 0)$ is given in Algorithm 6, applying Newton updates for the primal variable x:

Algorithm 6 ADMM with Newton Update

Initialization: $k \leftarrow 0, x^k \leftarrow 0, z^k \leftarrow 0, u^k \leftarrow 0$
repeat
  Primal Descent 1 (Newton Update).
  $x^{k+1}_{\text{Newton}} \leftarrow \left(h + \rho A^T A\right)^{-1} \left(g + \rho A^T \left(Ax^k - z^k + u^k\right)\right)$
  Primal Descent 2. $z^{k+1} \leftarrow \max\left(0, Ax^{k+1} + u^k\right)$
  Dual Ascent. $u^{k+1} \leftarrow u^k + Ax^{k+1} - z^{k+1}$
  Update. $x^* \leftarrow x^k$
  Increase k**.** $k \leftarrow k + 1$
until convergence is reached

In conclusion, ADMM is a flexible method that can be applied to solve a wide range of constrained optimization problems, including those with inequality con-

straints. ADMM does not require a strictly feasible starting point, unlike the barrier method. It alternates between updating the primal and dual variables and uses methods such as the Newton update to efficiently compute these updates when additional information about the problem, such as the gradient and Hessian of the cost function, is available [16][17].

Chapter 5

Solving Linear Systems

Solving linear systems in a common subproblem in convex optimization. In this chapter, we will discuss different methods for solving linear systems. We will discuss Gaussian elimination, LU decomposition, Cholesky decomposition, and the conjugate gradient method. All of these methods can be used to solve linear systems, but they have different properties and are suited for different problems. We will discuss the properties of each method and when they are suited to be used.

5.1 Gaussian elimination

Gaussian elimination is a simple method for solving linear systems. It is a direct method, signifying that it provides an exact solution within a finite number of steps, assuming that there are no rounding errors due to computer arithmetic. Gaussian elimination is a method for solving linear systems of the form $Ax = b$, where A is a square matrix and b is a vector [15]. Gaussian elimination works by transforming the system into an upper triangular system, which can be solved by back substitution.

Algorithm 7 Gaussian elimination

Given $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$.
for $k = 1, \ldots, n-1$ **do**
    for $i = k+1, \ldots, n$ **do**
        $l_{ik} := \frac{a_{ik}}{a_{kk}}$
        $a_{ik} := 0$
        for $j = k+1, \ldots, n$ **do**
            $a_{ij} := a_{ij} - l_{ik} a_{kj}$
        end for
        $b_i := b_i - l_{ik} b_k$
    end for
end for

In certain cases, Gaussian elimination can be unstable due to the division by a_{kk} in the algorithm. This issue can be resolved by using pivoting. Pivotting is a technique that swaps rows and columns in the matrix to avoid numerical instability. There are different types of pivoting, such as simple partial pivoting, scaled partial pivoting, and full pivoting [15]. In this thesis, we will only discuss simple partial pivoting. However, it is worth noting that there are other types of pivoting that can be better at avoiding numerical instability [15]. Partial pivoting works by swapping rows to ensure that the diagonal element is the largest element in the column. It is performed by finding the largest element in the column and swapping the rows. The algorithm for Gaussian elimination with partial pivoting is given in Algorithm 8.

Algorithm 8 Partial Pivoting Gaussian Elimination

Input $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$.
for $k = 1$ to $n-1$ **do**
    $p := \arg\max_{i \geq k} |a_{ik}|$
    if $p \neq k$ **then**
        Swap rows k, p in A, b
    end if
    for $i = k+1$ to n **do**
        $l_{ik} := \frac{a_{ik}}{a_{kk}}$, $a_{ik} := 0$, $b_i -= l_{ik} b_k$
        for $j = k+1$ to n **do**
            $a_{ij} -= l_{ik} a_{kj}$
        end for
    end for
end for

Once the matrix A has been transformed into an upper triangular matrix, the system can be solved by back substitution. Back substitution is a method for solving upper triangular systems. It works by solving the last equation for the last variable, then substituting the value of the last variable into the second-to-last equation and solving for the second-to-last variable. This process is repeated until all variables have been solved for. Back substitution is represented by:

Algorithm 9 Back substitution

Input $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$.
$x_n := \frac{b_n}{a_{nn}}$
for $i = n - 1$ to 1 **do**
    $x_i := \frac{b_i - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}$
end for

5.2 LU decomposition

The LU decomposition is a practical algorithm for solving linear systems of equations. Regarded as a direct method, it operates by breaking down a square matrix A into two components: a lower triangular matrix L and an upper triangular matrix U. This technique is applied for linear systems written as $Ax = b$, where A is the aforementioned square matrix and b signifies a vector.

The matrix decomposition transforms the initial system into the equivalent $LUx = b$. Solving this revised system is achieved in two steps. First, we solve the equation $Ly = b$ to determine y, followed by resolving $Ux = y$ to find the value of x. LU decomposition provides benefits over Gaussian elimination in solving systems of linear equations. Once the LU decomposition is performed, the resulting triangular matrices can be used to solve for different right-hand side vectors more efficiently, as they bypass the need for repeated Gaussian elimination. This is especially useful when solving systems of equations with multiple right-hand side vectors, as the LU decomposition can be performed once and then used to solve for each vector [15]. The representation of the LU decomposition is as follows:

Algorithm 10 LU Decomposition

Given $A \in \mathbb{R}^{n \times n}$.
Initialize $L = I_n$ and $U = A$.
for $i = 1, \ldots, n$ **do**
    for $j = i + 1, \ldots, n$ **do**
        $L_{ji} := \frac{U_{ji}}{U_{ii}}$
        for $k = i, \ldots, n$ **do**
            $U_{jk} := U_{jk} - L_{ji} \cdot U_{ik}$
        end for
    end for
end for
Return L and U.

As with Gaussian elimination, the LU decomposition can be unstable due to the division by U_{ii} in the algorithm, and the issue can be resolved by using pivoting [15]. We will not discuss pivoting for the LU decomposition in this thesis. When we have performed LU decomposition, we can solve the system $Ax = b$ by first solving $Ly = b$ for y and then solving $Ux = y$ for x. Solving $Ly = b$ for y is done by forward substitution, and solving $Ux = y$ for x is done by a backward substitution. Back substitution has already been discussed in algorithm 9, and forward substitution is given by:

Algorithm 11 Forward substitution

Given $L \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$.
$y_1 := b_1 / l_{11}$
for $i = 2, \ldots, n$ **do**
    $y_i := \left(b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right) / l_{ii}$
end for

5.3 Cholesky decomposition

Cholesky decomposition is also a direct method for solving linear systems. Cholesky decomposition is a method for solving linear systems of the form $Ax = b$, where A is a symmetric (i.e., $A = A^T$) and positive definite matrix (i.e., $x^T Ax > 0$ for all $x \neq 0$) and b is a vector. Cholesky decomposition works by decomposing the matrix A into a lower triangular matrix L and its transpose L^T. The system $Ax = b$ can then be written as $LL^T x = b$. We can then solve the system by first solving $Ly = b$ for y and then solving $L^T x = y$ for x. The Cholesky decomposition is given by:

Algorithm 12 Cholesky Decomposition

Given $A \in \mathbb{R}^{n \times n}$.
Initialize L as zero matrix of size A.
for $i = 1$ to n **do**
    for $j = 1$ to i **do**
        s $:= \sum_{k=1}^{j-1} L_{ik} L_{jk}$
        $L_{ij} := \frac{1}{L_{jj}}(A_{ij} - \text{s})$ if $i \neq j$ else $\sqrt{A_{ii} - \text{s}}$
    end for
end for

If matrix A is both symmetric and positive definite, it can be more efficiently decomposed into an LL^T form, which is referred to as the Cholesky decomposition. If $LL^T \neq A$, then A we know that A is not symmetric and positive definite.

This process requires roughly half the computational effort and storage compared to an LU decomposition. When we have performed Cholesky decomposition, we can solve the system $Ax = b$ by first solving $Ly = b$ for y and then solving $L^T x = y$ for x. Solving $Ly = b$ for y can be done by forward substitution, and solving $L^T x = y$ for x can be done by a backward substitution.

5.4 Conjugate gradient method

The conjugate gradient method is not a direct method. It is an iterative method, meaning that it gradually improves an initial guess for the solution at each step. The conjugate gradient method is a method for solving linear systems of the form $Ax = b$, where A is a symmetric and positive definite matrix and b is a vector [15]. The conjugate gradient method is represented as:

Algorithm 13 Conjugate Gradient Method

Given $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, and an initial guess $x^{(0)} \in \mathbb{R}^n$.
Compute $r^{(0)} := b - Ax^{(0)}$, set $p^{(0)} := r^{(0)}$.
for $k = 0, 1, 2, \ldots$ until convergence **do**
    Compute $\alpha^{(k)} := \frac{{r^{(k)}}^T r^{(k)}}{{p^{(k)}}^T A p^{(k)}}$.
    Update $x^{(k+1)} := x^{(k)} + \alpha^{(k)} p^{(k)}$.
    Compute $r^{(k+1)} := r^{(k)} - \alpha^{(k)} A p^{(k)}$.
    if $||r^{(k+1)}||_2 < \varepsilon$ (a small tolerance) **then**
        Break
    end if
    Compute $\beta^{(k)} := \frac{{r^{(k+1)}}^T r^{(k+1)}}{{r^{(k)}}^T r^{(k)}}$.
    Update $p^{(k+1)} := r^{(k+1)} + \beta^{(k)} p^{(k)}$.
end for

This method begins with a rough guess for the solution. Then, it calculates the difference between our guess and the real solution. This difference is used to guide the first step of the process.

Each step of the process moves in a certain direction to reduce the gap between our guess and the actual solution. It does this by finding the best length for each step. This length then helps to improve our guess [18].

Next, the process calculates a new difference value, which will be perpendicular (or "orthogonal") to all previous ones. This step is crucial to keep the errors orthogonal. The process then combines the new difference with the previous direction to create a new direction. The new direction is kept orthogonal to the previous ones using certain coefficients. This way, the orthogonality of differences is maintained even after this update [18].

The process keeps repeating these steps until the differences, or errors, become small enough to ignore. When this happens, the process considers the latest guess as the solution. The advantage of the conjugate gradient method is that it will, in some cases, be able to provide an approximate solution in very few iterations. Therefore, the conjugate gradient method can potentially be faster than direct methods for some problems. The maximum number of iterations that the conjugate gradient method theoretically requires corresponds to the number of distinct eigenvalues of matrix A; that is, it is capped at n. This characteristic renders the conjugate gradient method appealing for handling large cases [18].

Chapter 6

Design and Implementation

In this chapter, we delve into the design and implementation aspects of our project, highlighting three key modules: `Linear`, `Convex`, and `Portfolio`.

- `Linear` Module: This module is designed to tackle linear systems of equations. It houses a collection of functions that work together to solve these systems efficiently and accurately.
- `Convex` Module: This module centers around solving convex optimization problems. It incorporates numerous functions, each designed to handle different aspects of the optimization process, thereby providing robust solutions.
- `Portfolio` Module: Tailored for the specific needs of portfolio optimization, this module encapsulates an array of functions that help solve complex portfolio optimization problems.

In the following sections, we will delve into the design choices and implementation details of the algorithms encompassed within these modules. This in-depth exploration aims to foster a comprehensive understanding of the mechanics powering these computational tools and the intricate design principles they embody. It is worth noting that all of the modules utilize the official linear algebra module for Futhark, a module I have previously contributed to [19]. This linear algebra module consists of a variety of functions dedicated to matrix and vector operations within the realm of linear algebra.

6.1 Linear Module

The `Linear` module contains functions for solving linear systems of equations with the following methods: LU decomposition, Cholesky decomposition, Gaussian elimination, and the conjugate gradient method. The module implements the following API:

```
local module type linear = {
  type t
  val solve_Ab_gauss          [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_gauss_np       [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_lu_blocked     [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_lu             [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_lu_seq         [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_seq_seq        [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_flat_cholesky  [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_cholesky_dot   [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_cholesky_outer [n] : [n][n]t -> [n]t -> [n]t
  val solve_Ab_cg             [n] : [n][n]t -> [n]t -> [n]t
                                        -> t -> i64 -> [n]t
}
```

While these functions are at the core of the `Linear` module, the module also contains a number of other functions, such as functions solving linear systems with multiple right-hand sides and functions for computing the inverse of a matrix. These functions are not discussed in this report, but they are available in the source code and exported in the API. The following sections will discuss the design and implementation of the aforementioned functions.

6.1.1 Gaussian Elimination

In the pseudocode in Algorithm 7 (page 28), we have three nested loops, and inside each loop, we perform roughly n operations, and on average $(n/2)$ arithmetic operations [15]. Therefore, the work complexity of the algorithm is $\mathcal{O}(n^3)$. The span complexity is $\mathcal{O}(n)$, since we can update the matrix in parallel for each row. The algorithm is, therefore work-efficient, but not span-efficient. This gives rise to the following implementation, where we perform the elimination in parallel for each row:

```
def gauss_np [m][n] (A:[m][n]t) =
  loop A = copy A for i < i64.min m n do
    let irow = map (T./A[i,i]) A[i]
    in tabulate m (\j ->
        let scale = A[j,i]
        in map2 (\x y -> if j != i then y T.- scale T.* x else x) irow A[j]
    )
```

`tabulate` is a simple function that performs a map operation on a range of numbers. It is implemented as follows:

```
def tabulate 'a (n: i64) (f: i64 -> a): *[n]a =
  map1 f (iota n)
```

The span of `tabulate` is $\mathcal{O}(S(f))$ where $S(f)$ is the span of f. Therefore, we have that the span of the loop body is $\mathcal{O}(1)$ and the span of the loop is $\mathcal{O}(n)$, which is the span of the algorithm. The implementation is therefore span-efficient. For partial pivoting we need to find the largest element in the column with `argmax`, which can be done in parallel, although not with a span of $\mathcal{O}(1)$. The `argmax` function involes a `reduce` operation, which has a span of $\mathcal{O}(\log n)$. Therefore, the span of the function becomes $\mathcal{O}(n \log n)$. The implementation is shown below:

```
def gauss [m] [n] (A:[m][n]t) =
  loop A = copy A for i < i64.min m n do
  let p = A[i:,i] |> map T.abs |> argmax |> (.1) |> (+i)
      let A = if p != i then swap i p A else A
      let irow = map (T./A[i,i]) A[i]
      in tabulate m (\j ->
          let scale = A[j,i]
          in map2 (\x y -> if j != i then y T.- scale T.* x else x) irow A[j]
      )
```

6.1.2 LU decomposition

In the pseudocode in algorithm 10 (page 30), we have 3 nested loops and inside each loop and each loop performs $\mathcal{O}(n)$ operations. Therefore, the work complexity of the algorithm is $\mathcal{O}(n^3)$. The span complexity is $\mathcal{O}(n)$, since we can update the matrix in parallel for each row, but each row depends on updates from the previous row. A simple sequential implementation can be written as follows:

```
def seq_lu [n] (A: [n][n]t): ([n][n]t, [n][n]t) =
  let L = linalg.eye n
  let U = copy A
  in loop (L, U) for i in 0..<n do
      let (L, U) = loop (L, U) for j in (i+1)..<n do
          let Lji = U[j,i] T./ U[i,i]
          let L[j,i] = Lji
          let (L, U) = loop (L, U) for k in i..<n do
              let Ujk = U[j,k] T.- (Lji T.* U[i,k])
              let U[j,k] = copy Ujk
              in (L, U)
          in (L, U)
      in (L, U)
```

While this implementation may be effective for small matrices, it is not efficient for large matrices since the algorithm is not parallel. In the sequential version, the outer loop goes over each row of the matrix, and the inner loops adjust the elements in the lower and upper matrices based on the pivot (diagonal) element. These adjustments are made using scalar operations and nested loops, which are not parallelized.

When we convert this to use parallel matrix and vector operations, the goal is to replace the scalar operations and loops with operations that can be vectorized and run more efficiently on parallel hardware. The key observation is that the operations performed on each row in the inner loops are linear operations (addition and multiplication), which can be expressed as matrix and vector operations. This allows us to take advantage of the fact that Futhark can perform these operations in parallel.

Instead of dividing each element in a row by the pivot element, we can divide the entire row by the pivot. This insight allows us to replace a loop with a parallel operation. Instead of updating each element in a row one at a time, we can update the entire row in one operation. This is done by subtracting the product of the pivot row and the scaling factor from the current row. Again, we replace a loop with a vector operation.

```
1  def lud [n] (A: [n][n]t): ([n][n]t, [n][n]t) =
2    let L = linalg.eye n
3    let U = copy A
4    in loop (L, U) for j in 0..<(n-1) do
5        let factor = map (T./ U[j, j]) U[j+1:, j]
6        let L[j+1:, j] = factor
7        let U[j+1:] = map (\i ->
8            map2 (T.-) U[j+1+i] (map (T.*factor[i]) U[j])
9        ) (0..<(n - j - 1))
10   in (L, U)
```

The implementation is span-efficient, since the span of the loop body is $\mathcal{O}(1)$ and the span of the loop is $\mathcal{O}(n)$.

The `linalg` library in Futhark incorporates an LU decomposition function, adapted from a Rodinia benchmark [20]. Utilizing a blocked algorithm, this function is poised to deliver better efficiency when dealing with large matrices. The blocked LU decomposition method operates by subdividing the matrix into blocks and applying LU decomposition to each individual block. Although this does not present superior span complexity compared to the `lud` function, it exhibits increased parallelism and superior cache locality for large matrices. This function is also accessible through the `linear` module, albeit with a different interface. The function is designed to relieve the user from the task of determining the block

size, as it computes it automatically based on the matrix size. While not always delivering the optimal block size, the implementation suffices for most scenarios. The definition of the function is as follows:

```
def blocked_lud [n] (A: [n][n]t): ([n][n]t, [n][n]t) =
  let block_size = i64.max 1 (i64.min 32 (n / 4))
  in lu.lu2 block_size A
```

The blocksize has been chosend based on the results of the benchmarks. Once we have decomposed the matrix into lower and upper triangular matrices, we can solve the system of equations using forward and backward substitution. The `solveLUb` function below performs forward and backward substitution:

```
def forward_substitution [n] (L: [n][n]t) (b: [n]t): [n]t =
  let y = linalg.veczeros n
  in loop y for i in 0..<n do
    let sumy = linalg.dotprod L[i,:i] y[:i]
    let y[i] = copy (b[i] T.- sumy) T./ L[i,i]
    in y
def back_substitution [n] (U: [n][n]t) (y: [n]t): [n]t =
  let x = linalg.veczeros n
  in loop (x) for j in 0..<n do
    let i = n - j - 1
    let sumx = linalg.dotprod U[i,i+1:n] x[i+1:n]
    let x[i] = copy (y[i] T.- sumx) T./ U[i,i]
    in x
def solveLUb [n] (L: [n][n]t) (U: [n][n]t) (b: [n]t) =
  forward_substitution L b |> back_substitution U
```

6.1.3 Cholesky decomposition

The Cholesky function used by algorithm 12 (page 31) has 2 nested loops. However, we also have a summation inside the inner loop, which is essentially another loop. Therefore, the work complexity of the algorithm is $\mathcal{O}(n^3)$. The span is $\mathcal{O}(n)$, since we can update the matrix in parallel for each row, but each row depends on updates from the previous row, similar to the LU decomposition. A simple sequential implementation of Cholesky decomposition is as follows:

```
def cholesky_banachiewicz [n] (A: [n][n]f64): [n][n]f64 =
  let L = replicate n (replicate n 0.0)
  in loop L for i in 0..<n do
    loop L for j in 0...i do
      let sum = loop sum = 0.0 for k in 0..<j do
        L[i,k] * L[j,k] + sum
      in if i == j then
        let L[i,j] = f64.sqrt (A[i,i] - sum)
        in L
      else
        let L[i,j] = (1.0 / L[j,j] * (A[i,j] - sum))
        in L
```

This implementation employs the Banachiewicz algorithm, a variant of the Cholesky decomposition algorithm that initiates from the upper left corner and proceeds by calculating the matrix row by row [21]. Alternatively, we can employ the Cholesky-Crout algorithm, which allows us to proceed column by column [21]. This method is also available in Futhark and is implemented in the `linear` module. While these sequential implementations may excel for smaller matrices, their efficiency diminishes for larger matrices. However, we can circumvent this limitation as the algorithm allows parallelization, primarily because most calculations have no dependencies. Bearing this in mind, we can design a parallel version of the algorithm as detailed below:

```
1  def cholesky [n] (A: [n][n]t): [n][n]t =
2    let A = loop A = copy A for j in 0..<n do
3    let A[j,j] = T.sqrt (copy A[j,j])
4        let A[j+1:,j] = map (T./A[j,j]) A[j+1:n,j]
5        let m = n - j - 1
6        let v = A[j+1:n,j] :> [m]t
7        let outer_product = outer v v :> [m][m]t
8        let mat = A[j+1:n,j+1:n] :> [m][m]t
9        let A[j+1:n,j+1:n] = linalg.matsub mat outer_product
10       in A
11   in tril A
```

All of the operations in the loop body are span $\mathcal{O}(1)$. The `outer` function is implemented using two nested maps, and the span of the rest of the operation can be trivially inferred. The span of the loop is $\mathcal{O}(n)$, since each iteration depends on the previous one. Therefore, we have that the span complexity of the implementation is $\mathcal{O}(n)$. While it should be much faster than the sequential version for large matrices, it is still not optimal. We are only reading and writing to the lower triangular part of the matrix, but we are still performing calculations on the upper triangular part, and the memory access pattern is not optimal. We also offer a more direct parallelization of the sequential algorithm in the `linear` module, but it is not span efficient because it computes dot products in the loop body. We can improve the implementation above by flattening the matrix and only keeping the lower triangular part in memory. This is a good idea because we can reduce the memory footprint of the algorithm by half. We can map the lower triangular part of the matrix to a 1D array using the `tril_indices` below:

```
def tri_num (n: i64): i64 = n * (n + 1) / 2
def index_to_ij (k: i64) (n: i64): (i64, i64) =
    let kp = n * (n + 1) / 2 - k - 1
    let p = i64.f64 ((f64.sqrt (f64.i64 (1 + 8 * kp)) - 1) / 2)
    let i = n - (kp - p * (p + 1) / 2) - 1
    let j = n - p - 1
    in (i, j)
def tril_indices (n: i64): *[](i64, i64) =
    let l = (tri_num n)
    in map (\k -> index_to_ij k n) (iota l)
```

`tril_indices` a formula derived in [22]. We could achieve the same with a map running for $i \in [0, n^2)$, and perform `filter` on the resulting array to only keep the indices that correspond to the lower triangular part of the matrix. However, this would be less efficient because the span of the `filter` operation is $\mathcal{O}(\log n)$ and the `tril_indices` implementation above has span $\mathcal{O}(1)$. We can now implement the parallel version of the Cholesky decomposition algorithm as follows, by first mapping the lower triangular part of the input matrix A to a 1D array and then performing the Cholesky decomposition on the 1D array:

```
 def flat_cholesky [n] (A: [n][n]t): []t =
  let t_i = tril_indices n
  let l = length t_i
  let L = map (\(i, j) -> A[i, j]) t_i
  let m = tri_num n
  let L = loop L for j in 0..<n-1 do
      let di = m - (tri_num (n - j))
      let sdi = T.sqrt (copy L[di])
      let L[di] = sdi
      let st = n - j - 1
      let st_i = di + 1
      let end = st_i + st
      let L[st_i:end] = map (T./sdi) L[st_i:end]
      let s_c = j + 1
      let r = n - s_c
      let k = tri_num r
      let o_i = t_i[l-k:]
      let o_i = map (\(a, b) -> let (c, d) = o_i[0] in (a - c, b - d)) o_i
      let c = L[st_i:end] :> *[st]t
      let o = (low_outer c o_i) :> [k]t
      let L[m-k:] = map2 (\x y -> x T.- y) (L[m-k:] :> [k]t) o
      in L
  let di = m - 1
  let L[di] = T.sqrt (copy L[di])
  in L
```

Once we have performed the Cholesky decomposition on the 1D array, we can map the array back to a 2D array using the inverse of the `index_to_ij` function.

```
def i_j_to_index (i: i64) (j: i64) (n: i64): i64 =
  let p = n - j - 1
  let kp = p * (p + 1) / 2 + (n - i - 1)
  let k = n * (n + 1) / 2 - kp
  in k - 1
def cholesky_unflat [n] A: [n][n]t =
  let m = tri_num n
  let L = flat_cholesky A
  in map (\i ->
      map (\j ->
          if i < j then (T.f64 0.0) else L[i_j_to_index i j m]
      ) (0..<n)
  ) (0..<n)
```

Another way we could enhance this algorithm's performance is through a block-wise approach, splitting the matrix into blocks for separate Cholesky decompositions. This method potentially minimizes cache misses and could offer a higher degree of parallelism [23].

6.1.4 Conjugate gradient method

The conjugate gradient method is an iterative method for solving systems of linear equations where the matrix is symmetric and positive definite. We can implement Algorithm 13 (page 32) in Futhark as follows:

```
def solve_Ab_cg [n] (A: [n][n]t) (b: [n]t) (x0: [n]t) (tol: t) (max_iter: i64)
    =
  let r = map2 (T.-) b (linalg.matvecmul_row A x0)
  let rs_old = linalg.dotprod r r
  let p = r
  let x = x0
  let i = 0
  let running = true
  let res = loop (x, p, r, rs_old, i, running) while running && i < max_iter
    do
      let i = i + 1
      let Ap = linalg.matvecmul_row A p
      let alpha = rs_old T./ linalg.dotprod p Ap
      let x = map2 (T.+) x (map (alpha T.*) p)
      let r = map2 (T.-) r (map (alpha T.*) Ap)
      let rs_new = linalg.dotprod r r
      in if T.sqrt rs_new T.< tol then
          (x, p, r, rs_old, i, false)
      else
          let p = map2 (T.+) r (map ((rs_new T./ rs_old)T.*) p)
          in (x, p, r, rs_new, i, running)
  in res.0
```

The function begins by initializing several variables. The residual vector `r` is calculated by subtracting the matrix-vector product of `A` and `x0` from `b`. The scalar `rs_old` is the dot product of `r` with itself. Vector `p` is initialized to `r`, `x` is

initialized to `x0`, the iteration count `i` is set to zero, and a boolean flag `running` is set to true. Following the initialization, the main iterative loop of the function begins. This loop runs until either the maximum number of iterations is reached or the `running` flag is set to false. In each iteration, the function calculates several values. First, it computes `Ap` as the matrix-vector product of `A` and `p`. The scalar `alpha` is calculated as the ratio of `rs_old` to the dot product of `p` and `Ap`. Then, `x` and `r` are updated. `x` is incremented by `alpha` times `p`, while `r` is decremented by `alpha` times `Ap`. The scalar `rs_new` is computed as the dot product of the updated `r` with itself. If the square root of `rs_new` is greater than `tol`, the `running` flag remains true; otherwise, it is set to false. Next, `p` is updated by adding `r` to the product of the ratio `rs_new/rs_old` and `p`. The old `rs_old` value is replaced with `rs_new`, and the iteration count `i` is incremented by one. The loop returns a tuple containing `x`, `p`, `r`, `rs_old`, `i`, and `running`. Finally, the function returns the first element of this tuple, `x`, which represents the solution to the system of linear equations.

6.2 Convex Module

The `Convex` modules contains functions for solving convex optimization problems using the following methods: gradient descent, Newton's method, barrier method, ADMM, and more. The module implements the following API (some function have been omitted for brevity):

```
local module type solver = {
  type t
  gradient_descent
  newtons_method
  newton_equality
  barrier_method
  admm
  grad_jvp
  grad_vjp
  hess_jvp
  hess_vjp
  solve_qp
}
```

6.2.1 Auto differentiation

Auto differentiation is very useful for solving optimization problems. It can be used to compute the gradient and Hessian of functions. Finding and implementing the gradient and Hessian of a function by hand can be very tedious and error-prone, but with auto differentiation, it is straightforward for the user. We use the

built-in functions `jvp` and `vjp` (explained in 2.1.2) to compute the gradient and Hessian of a function.

```
-- Compute gradient of f at x
def grad_jvp [n] (f: [n]t -> t) (x: [n]t) =
    map (\i ->
        let X = veczeros n
        let X[i] = T.f64 1.0
        in jvp f x X
    ) (iota n)

-- Compute Hessian of f at x
def hess_jvp [n] (f: [n]t -> t) (x: [n]t) =
    map (\i ->
        map (\j ->
            let X = veczeros n
            let X[i] = T.f64 1.0
            let Y = veczeros n
            let Y[j] = T.f64 1.0
            in jvp (\w -> jvp f w X) x Y
        ) (iota n)
    ) (iota n)
```

We can also implement functions for gradient and Hessian using `vjp`. Although `jvp` cannot just be replaced by `vjp` in the above code, because `vjp` does not take a one hot seed vector as input.

```
let f x = reduce (+) 0.0 (map (**2.0) x)
let g = grad f x
let h = hess f x
```

```
g = [2.0, 4.0, 6.0]
h = [[2.0, 0.0, 0.0],
     [0.0, 2.0, 0.0],
     [0.0, 0.0, 2.0]]
```

We will examine the performance impact of using these functions in chapter 7. The impact will be evaluated for these functions using `jvp` and `vjp`, and gradient and Hessian functions derived and implemented by hand. We expect that `vjp` will be faster for our purposes, as it is most efficient for functions that have more inputs than outputs [8].

6.2.2 Gradient descent

We need to implement backtracking line search for gradient descent. We use the Armijo-Goldstein condition to determine the step size in gradient descent. In the Futhark code we introduce an extra parameter to algorithm 2 called `max_iter`. This parameter is used to prevent the algorithm from running forever if the condition is never satisfied.

```
def armijo f x delta_x grad alpha t =
  let left t = f (map2 (+) x (map (*t) delta_x))
  let right t = f x + alpha * t * linalg.dotprod grad delta_x
  in left t <= right t

def line_search f x delta_x grad alpha beta max_iter =
  let (t, _) = loop (t, i) = (1.0, 0) while i <= max_iter && !(armijo f x
    delta_x grad alpha t) do
    (t * beta, i + 1)
  in t
```

Backtracking line search can be fairly elegantly implemented in Futhark. We use a while loop to iterate until the condition is satisfied or the maximum number of iterations is reached. Even though the Armijo function can be executed in parallel with a span of $\mathcal{O}(\log(n) \cdot \mathrm{S}(f(x)))$, the while loop is executed sequentially, so the span of the line search algorithm is the span of the `armijo` function multiplied by the number of iterations until the condition is satisfied. We can improve it by testing multiple values of t in parallel using a map and then selecting the first value that satisfies the condition (using reduce). Then we can remove the while loop and only perform the line search for a select number of t values.

```
def parallel_line_search [n] f x delta_x grad alpha beta n_ts =
  let ts = map (\i -> 1.0 / (beta ** (f64.i64 i))) (iota n_ts)
  let conds = map (\t -> (t, armijo f x delta_x grad alpha t)) ts
  in (reduce (\(t1, b1) (t2, b2) -> if b1 then (t1, b1) else (t2, b2)) (0.0,
    false) conds).0
```

Now gradient descent can be implemented as follows.

```
def gradient_descent [n] f f_grad x0 alpha beta max_iter =
  let (x, _) = loop (x, i) = (x0, 0) while i <= max_iter do
  let grad = f_grad x
  let delta_x = map (*(-1.0)) grad
  let t = parallel_line_search f x delta_x grad alpha beta 10
  let x' = map2 (+) x (map (*t) delta_x)
  in (x', i + 1)
  in x
```

As a result, we no longer have sequential equation of a loop for backtracking line search.

6.2.3 Newton's method

Newton's method has a similar structure to gradient descent, but instead of using the gradient to determine the direction of the next step, it uses the inverse of the Hessian. We use the same backtracking line search as in gradient descent or unit step size. The version shown below is with unit step size for simplicity. Newton's

method should not be as sensitive to the step size as gradient descent [14]. The implementation of Newton's method is shown below.

```
1 def newtons_method f_grad f_hess x0 epsilon max_iter =
2   let res = loop (x, i, r) = (x0, 0, true) while r && i < max_iter do
3       let grad = f_grad x
4       let hess = f_hess x
5       let step = linear.solveAb_cholesky hess (map (T.neg) grad)
6       let x = map2 (T.+) x step
7       let r = linalg.vecnorm step T.< epsilon
8       in (x, i + 1, r)
9   in res.0
```

We may use the same backtracking line search as in gradient descent. We use the norm of the step vector to determine if the algorithm has converged. This is not a very good measure of convergence but should be sufficient for our purposes.

Notice that we do not invert the Hessian as we did in the mathematical description of the algorithm (page 20). Instead, we use the linear solver `linear.solveAb_cholesky` to solve the linear system $Hx = -\nabla f(x)$. This amounts to the same thing as inverting the Hessian and multiplying it with $-\nabla f(x)$, but it is more efficient. Inverting the matrix would require us to solve the linear system $Hx = I$, where I is the identity matrix. This is equivalent to solving n linear systems $Hx_i = e_i$, where e_i is the i'th column of the identity matrix. These equations can be solved in parallel, but it is still more efficient to solve the linear system $Hx = -\nabla f(x)$ directly because we only need to solve one linear system instead of n.

A large part of this algorithm is the computation of the linear system $Hx = -\nabla f(x)$. If the function f is convex, then the Hessian is symmetric positive definite, which means that we can use the Cholesky and conjugate gradient to solve the linear system. However, we can also use LU decomposition or Gauss elimination. We will compare and examine these in chapter 7.

6.2.4 Newton's method with equality constraints

We extend Newton's method to handle equality constraints. We construct the KKT matrix using the Futhark slicing syntax. The KKT system cannot be solved using Cholesky decomposition or conjugate gradient, because the KKT matrix is not guaranteed to be symmetric positive definite. Instead, we use LU decomposition or Gaussian elimination to solve the linear system. The performance of these two algorithms is compared in chapter 7.

```
1  def newton_equality f f_grad f_hess A b x0 max_iter =
2    let f_grad = grad f
3    let f_hess = hess f
4    let (x, r) = loop (x, i, r) = (x0, 0, true) while i < max_iter && r do
5      let grad_val = f_grad x
6      let Hessian_val = f_hess x
7      let KKT = matzeros (n + m) (n + m)
8      let rhs = veczeros (n + m)
9      let KKT[:n, :n] = Hessian_val
10     let KKT[:n, n:] = transpose A
11     let KKT[n:, :n] = A
12     let rhs[:n] = map T.neg grad_val
13     let rhs[n:] = map2 (T.-) b (matvecmul_row A x)
14     let delta_x_lam = solve_Ab_lu_blocked KKT rhs
15     let delta_x = delta_x_lam[:n]
16     let x_new = map2 (T.+) x (map (T.*t) delta_x)
17     in if vecnorm delta_x T.< epsilon then
18       (x, false)
19     else
20       (x_new, true)
21   in x
```

In the code above, we apply unit step size to the direction/step vector. We can also use a line search to determine the step size. We will not implement this here. We use the norm of the step vector to determine if the algorithm has converged. This method is not a very good measure of convergence but should be sufficient for our purposes. Depending on the problem, one might want to use more sophisticated stopping criteria, for instance, a relative decrease in the objective function value or a decrease in the duality gap.

One must be cautious about the initialization of the algorithm. The initial guess x_0 can significantly affect the convergence of the algorithm and may even prevent convergence if it is chosen poorly. The algorithm is a feasible descent method, which means that it will only converge if the initial guess is feasible. The method also assumes that the constraints are linear. Nonlinear constraints can be linearized using a Taylor series expansion, but this can introduce additional complexity and may require multiple iterations to achieve an acceptable level of accuracy [12].

6.2.5 Barrier method

The barrier method is an extension of Newton's method with equality constraints. It uses a barrier function to approximate the inequality constraints. The barrier function is defined as follows, where `fi` are the inequality constraints:

```
1 let phi x = T.neg (reduce (+) 0.0 (map (T.log <-< T.neg) (fi x)))
```

The barrier function is then added to the objective function and Newton's method with equality constraints is used to minimize the objective function. Then we minimize this new objective function with Newton's method with equality constraints. This process is repeated until the solution converges. We have that it converges when $m/t < \epsilon$, where m is the number of inequality constraints, ϵ is the tolerance, and t is a parameter that controls the scaling of the barrier function. The approximation parameter t is scaled by μ at each iteration. Since m, μ, and ϵ are constants; we can compute the number of iterations required to converge before running the algorithm. This can potentially speed up the program because Futhark may be able to unroll the loop. We solve the following inequality for i, where i is the iteration number:

$$
\begin{aligned}
m/(t \cdot \mu^i) &< \epsilon \\
m &< \epsilon \cdot t \cdot \mu^i \\
m/(\epsilon \cdot t) &< \mu^i \\
\log_\mu(m/(\epsilon \cdot t)) &< i \\
\frac{\log(m/(\epsilon \cdot t))}{\log(\mu)} &< i
\end{aligned}
$$

Therefore, we can compute the number of iterations `iters` required to converge as follows:

```
1 let iters = i64.f64 ((f64.log (m / (epsilon * t))) / (f64.log (mu)))
```

The barrier method is implemented as follows:

```
def barrier f0 f0_grad f0_hess phi phi_grad phi_hess mm A b x x0 t0 mu eps =
  let iters = T.to_i64 (T.log (T.i64 mm T./ (eps T.* t0)) T./ T.log mu T.+ T.
    f64 1.0)
  let res = loop (x, t) = (x0, t0) for i < iters do
      let f x = t T.* (f0 x) T.+ phi x
      let f_grad x = map (T.* t) (f0_grad x) |> map2 (T.+) (phi_grad x)
      let f_hess x = map (map (T.* t)) (f0_hess x) |> map2 (map2 (T.+)) (
    phi_hess x)
      let x_new = newton_equality_nan f f_grad f_hess A b x max_iter
      in (x_new, t T.* mu)
  in res.0
```

This function takes many arguments. The arguments `f0`, `f0_grad`, and `f0_hess` are the objective function, its gradient, and its Hessian matrix, respectively. The arguments `phi`, `phi_grad`, and `phi_hess` are the barrier function, its gradient, and its Hessian matrix, respectively. The argument `mm` is the number of inequality constraints. The arguments `A` and `b` are the equality constraints. The argument `x0` is the initial guess. The argument `t0` is the initial value of t, used to scale the barrier function. The argument `mu` is the scaling factor for t. The argument `eps` is the tolerance. The function returns the solution vector `x`. In the module we also offer a function `barrier_auto`, which automatically computes the gradient and Hessian matrix of the barrier function.

We see that the span of this function is largely influenced by the span of Newton's method with equality constraints, and this, in turn, is primarily influenced by the span of solving the linear system, so if we want to reduce the span of this function further, we would have to focus on reducing the span of solving the optimization problem with equality constraints.

In order to find a feasible point, we solve the optimization problem described in section 4.4.1. We use the barrier method to solve this problem. We also provide a function `solve_qp` that finds and feasible point and solves the optimization problem.

```
def find_feasible_point [n][m] (fi: [n]t -> []t) (A: [m][n]t) (b: [m]t): [n]t
    =
  let x = least_squares A b -- Solving Ax = b
  -- Compute s, an upper bound for the maximum constraint violation
  let s = (fi x |> reduce T.max (T.f64 0.0)) T.+ (T.f64 0.001)
  let n' = n + 1
  let x = x ++ [s] :> [n']t
  let A = transpose ((transpose A) ++ [veczeros m]) :> [m][n']t
  let f [n'] (x: [n']t): t = x[n'-1] -- New objective function
  let fi_new [n'] (x: [n']t): []t = -- Define the modified constraint function
      let s = x[n'-1]
      let x = x[0: n'-1] :> [n]t
      in map (\x' -> x' T.- s) (fi x)
  let x = barrier_method_default fi_new f A b x
  in x[0: n] :> [n]t
```

We can wrap the barrier method with a `map` to solve a batch of problems. We can also do this for Newton's method with and without equality constraints and the gradient descent method. However, we cannot do this when the implementation uses the blocked LU decomposition because the Futhark compiler has rejected this with the message "Cannot handle un-sliceable allocation size. Likely cause: irregular nested operations inside parallel construct". This is a current compiler limitation and will hopefully be fixed in the future.

6.2.6 Alternating direction method of multipliers

The `admm` function utilizes the Alternating Direction Method of Multipliers (ADMM) algorithm that solves optimization problems by dividing them into small-er, easier subproblems (see page 24). ADMM is particularly useful in large-scale optimization and distributed computation because we can split each subproblem into many subproblems that can be solved in parallel. However, for this implementation, we only parallelize the computation of each subproblem. We alternate between updating the variables `x`, `z`, and `u`. The variable `x` is the variable we want to optimize. We do this by minimizing the augmented Lagrangian function with respect to `x`:

```
let update_x (x: [n]t) (z: [m]t) (u: [m]t) =
  let Ax = linalg.matvecmul_row A x
  let g1 = f_grad x
  let Ax_z = map2 (T.-) Ax z
  let Ax_z_u = map2 (T.+) Ax_z u
  let At_t = linalg.matvecmul_row (transpose A) Ax_z_u
  let g2 = map (rho T.*) At_t
  let gradient = map2 (T.+) g1 g2
  let h1 = f_hess x
  let At_A = linalg.matmul (transpose A) A
  let h2 = map (map (rho T.*)) At_A
  let Hessian = map2 (map2 (T.+)) h1 h2
 (veczeros n) (T.f64 1e-8) n
  let delta_x = linear.solve_Ab_cholesky Hessian gradient
  in map2 (T.-) x delta_x
```

We see that we use Newton's method to compute the update for `x`. Cholesky decomposition is used to solve the linear system because the Hessian matrix is symmetric and positive definite. Therefore, we can also use all of the other methods that we have implemented for solving linear systems. When `x` has been updated, we update `z` and `u` as follows:

```
  let update_z (x: [n]t) (u: [m]t) =
    let Ax = linalg.matvecmul_row A x
    let zu = map2 (T.+) Ax u
    in map2 T.max b zu
```

And finally, we update `u` as follows:

```
let update_u x z u =
    let Ax = linalg.matvecmul_row A x
    let Ax_z = map2 (T.-) Ax z
    in map2 (T.+) u Ax_z
```

In the function's implementation, it alternates between updating `x`, `z`, and `u`. Each update represents one part of the overall optimization and leverages data from the other components to adjust its own value, which in turn influences the other elements' updates in the next iteration. The updates are derived based on the nature of the problem and the structure of the constraints, leveraging gradient and Hessian of the objective function and the system matrix `A`.

The initial inputs for `x`, `z`, and `u` are defined outside the main iteration loop. For `x`, the initial point is provided by the user, while `z` and `u` are initialized to zero.

The function continues updating these values until either a maximum iteration count `max_iter` is reached or the convergence criterion is met. Convergence is measured as the relative change in `x` and is considered achieved if it falls below a specified tolerance level (`tol`). It is a typical convergence measure in iterative methods ensuring the result is sufficiently close to the true solution.

It is important to note the role of the `rho` parameter, which is a user-specified penalty parameter. This value can greatly impact the efficiency of the method and may need to be tuned depending on the problem at hand. A large value of `rho` will put more emphasis on the constraint violation, which may be useful if the constraints are particularly important. The result of the algorithm is largely influenced by `rho`, so it is important to choose a good value. For this implementation, we use a fixed value of `rho`, but more advanced schemes exist for choosing `rho` dynamically [24].

6.3 Portfolio module

The `portfolio` module contains functions for solving portfolio optimization problems. The module implements the following API:

```
local module type portfolio = {
    type t
    val efficient_return         [n] : t -> [n]t -> [n][n]t -> [n]t
    val efficient_risk           [n] : t -> [n]t -> [n][n]t -> [n]t
    val min_volatility           [n]   : [n][n]t -> [n]t
    val efficient_return_and_esg [n] : t -> t -> [n]t -> [n]t
                                           -> [n][n]t -> [n]t

    }
```

We see that the module contains functions for solving the efficient return, efficient risk, and minimum volatility problems. The module also contains a function for solving the efficient return and ESG problem. This amounts to the following optimization problem:

$$\begin{aligned}
\text{Minimize} \quad & x^T \Sigma x && (6.1)\\
\text{Subject to} \quad & \mu^T x \geq r && (6.2)\\
& esg^T x \geq e && (6.3)\\
& x \geq 0 && (6.4)\\
& \sum_{i=1}^{n} x_i = 1 && (6.5)
\end{aligned}$$

where:

- μ is a vector in $\mathbb{R}^n$ of expected returns.
- Σ is an $n \times n$ covariance matrix of returns.
- x is the vector in $\mathbb{R}^n$ that we're solving for.
- r is a scalar parameter that represents the minimum expected return.
- esg is a vector in $\mathbb{R}^n$ of ESG scores.
- e is a scalar parameter that represents the minimum ESG score.

Solving this problem will give use the portfolio with minimum volatility with expected return greater than r and expected ESG score greater than e, where we allow no short selling ($x \geq 0$) and require that the sum of the weights is equal to one ($\sum_{i=1}^{n} x_i = 1$).

The following function can be used to find the portfolio with minimum volatility for minimum expected return and minimum expected ESG score.

```
def efficient_return_and_esg [n]
  (target_return: t)
  (target_esg: t)
  (mus: [n]t)
  (esg: [n]t)
  (covs: [n][n]t)  =
  -- Inequality constraints
  let fi (x: []t) =
      in [-- expected_esg >= target_esg
          target_esg T.- (expected_esg x esg)
          -- expected_return >= target_return
          target_return T.- (expected_return x mus)
          -- x >= 0
      ] ++ map (T.neg) x
  -- Equality constraints
  -- sum(x) = 1
  let A = [replicate n (T.f64 1.0)]
  let b = [(T.f64 1.0)]
  -- Objective function
  let f0 (x: [n]t) = expected_risk x covs
  -- Solve the problem
  in convex.solve_qp f0 fi A b
```

We have the following three functions to compute the expected return, expected risk, and expected ESG score:

```
let expected_risk x Sigma = linalg.dotprod x (linalg.matvecmul_row Sigma x)
let expected_return x mus = linalg.dotprod x mus
let expected_esg x esg = linalg.dotprod x esg
```

`efficient_return_and_esg` is a simple example of how the `convex` module can be used with minimum effort. The inequality constraints are defined in the function `fi`. The inequality constraints are expressed as a list of expressions that must evaluate to a non-negative value. If `n` is the number of assets, then we have `n + 2` inequality constraints. The first `n` constraints are the non-negativity constraints. The other two constraints are the minimum expected return and minimum expected ESG score. The equality constraints are defined as `A` and `b`. The objective function is defined in the function `f0`.

An important note, is that we can exhibit much better (as shown in Chapter 7) than this simple example by deriving the derivatives by hand and passing them to the `convex` module. We can calculate the derivatives of the objective functions a follows:

$$\begin{aligned}
\nabla(x^T\Sigma x) &= (\Sigma + \Sigma^T)x \\
&= 2\Sigma x && \text{(since } \Sigma \text{ is symmetric)} \\
\nabla^2(x^T\Sigma x) &= \nabla(2\Sigma x) \\
&= 2\Sigma
\end{aligned}$$

```
1 let f x = linalg.dotprod x (linalg.matvecmul_row Sigma x)
2 let g x = map (*2) (linalg.matvecmul_row Sigma x)
3 let h _ = map (map (*2)) Sigma
```

We can do the same for the barrier function ϕ (this is a bit more involved, but the idea is the same). We would then use the derived functions to solve the same problem. The performance of manually derived derivatives will be evaluated against the auto-differentiation detailed in Chapter 7.

Chapter 7

Benchmarking and testing

This chapter addresses the benchmarking and performance testing of the Futhark implementation. We will solve different portfolio optimization problems and compare the performance of the Futhark implementation against the Python implementation. We will also compare the performance of the Futhark implementation against Python libraries for solving linear systems. The modules have been instantiated with `f64` for all of the benchmarks, but it is trivial to switch the type to `f32` or any other `real` type due to Futhark's polymorphic module system.

Every Futhark benchmark has been conducted using `futhark bench` and with CUDA as the backend. The GPU leveraged is an NVIDIA A100 with 40GB of VRAM. The CPU used for the benchmarks is an AMD EPYC 7352 24-Core Processor, equipped with 527 GB of RAM. The versions of Futhark and Python utilized are 0.22.3 and 3.10, respectively. The Python libraries used are `NumPy` (version 1.23.3), `scipy` (version 1.9.3), and `CVXPY` (version 1.3.0).

7.1 Linear Module Evaluation

This section focuses on the evaluation of the `linear` module, where we will contrast the performance of Cholesky and LU decomposition and the effectiveness of solving a linear system of equations represented as $Ax = b$. This entails benchmarking the performance of the Futhark implementations against Python libraries such as `numpy` and `scipy`. We will also gauge the performance when solving linear systems in a batched manner, a distinct feature of Futhark that proves beneficial for handling large-scale problems and distributed computation. Optimizing batched performance is also crucial for the convex optimization algorithms when aiming to solve multiple problems in parallel.

7.1.1 Cholesky

In this section we will compare the performance of the Cholesky implementations, batched and non-batched. We will compare the performance of the Futhark implementation against the Python implementation.

Method \ Dim	100x100	500x500	1000x1000	5000x5000	10000x10000
Flat	2251µs	13116µs	28696µs	649778µs	4720386µs
Crout/Seq	562549µs	68469887µs	-	-	-
Banachiewicz/Seq	550646µs	67851051µs	-	-	-
Outer	2295µs	11501µs	23247µs	1128661µs	9725639µs
Dot	2578µs	13024µs	28017µs	267419µs	1323244µs
NumPy	63µs	1693µs	7939µs	499338µs	2788463µs

Table 7.1: Benchmark results of different Cholesky methods and NumPy. Time in µs is reported. "-" indicates that the method exceeded the maximum time limit of 10 minutes.

The Crout and Banachiewicz methods are noticeably slower in comparison to other techniques. The reason for this is their lack of parallelization and naive sequential implementation. The NumPy method proves to be the most efficient for matrices that are smaller than 1000x1000. This can be attributed to the fact that NumPy leverages a highly optimized C implementation of the Cholesky decomposition, and we benefit more from the parallelization as the matrix size increases.

The flat version of Cholesky is almost double the speed of the implementation is was derived from (the outer product version). Surprisingly, we see that the "dot-version" derived directly from Banachiewicz is very fast for large matrices and is the only method that outperforms NumPy for the 5000x5000 matrix and 10000x10000. This is surprising because it has a span of $\mathcal{O}(n \log n)$ because it computes dot products in the body of the outer loop. The "outer" and "flat" versions have a span of $\mathcal{O}(n)$, but scale worse in practice. I suspect this is because the "dot" version has a better memory access pattern and thus benefits more from the cache, but it could also be that Futhark makes some unfortunate optimizations for the "outer" and "flat" versions.

Next, we'll delve into the batched versions of the Cholesky decomposition. This means we'll examine and compare the performance when executing the Cholesky decomposition on a large set of matrices

Method \ Dim	1000000x 5x5	1000000x 10x10	100000x 20x20	10000x 50x50	1000x 100x100	100x 500x500
Flat	5265µs	45702µs	35286µs	56711µs	240236µs	29102880µs
Crout/Seq	1942µs	10400µs	4565µs	11628µs	43709µs	7973984µs
Banachiewicz/Seq	1800µs	9672µs	4254µs	6238µs	22206µs	2105505µs
Outer	6495µs	60511µs	51839µs	131924µs	537358µs	58620970µs
Dot	7152µs	56095µs	42474µs	125089µs	533273µs	58408621µs
NumPy	1841378µs	2033527µs	325007µs	98811µs	85831µs	165973µs

Table 7.2: Benchmark results of different batched Cholesky methods. Time in µs is reported.

We observe numerous interesting trends within this table. The flat version consistently outperforms all of the other parallel methods, yet the sequential versions (Crout and Banachiewicz) prove to be the fastest, with Banachiewicz taking the lead. They display impressive speed when dealing with small matrices, but the results for larger matrices are intriguingly peculiar. It appears that the sequential implementations can compute 100 matrices of size 500x500 faster than a single matrix of the same size in the non-batched benchmark. This oddity remains unexplained, but I speculate that when it is not encapsulated by a map function, the Futhark compiler might make some unfortunate optimizations.

Interestingly, the NumPy implementation is the fastest for the 500x500 matrices. All of the parallel implementations seem to experience significant scaling issues when encapsulated by a map. This is evident as the runtime for the "dot" version with 1000x100x100 exceeds the runtime of a single 100x100 (as shown in table 7.1) multiplied by 1000. The same observation applies to 100x500x500 for all of the parallel versions, hinting that we might as well run the map sequentially for large matrices.

7.1.2 LU Decomposition

In this section, we will compare the performance of both batched and non-batched LU implementations. We'll assess the performance of the Futhark implementation in contrast to the LU decomposition from SciPy. SciPy employs LAPACK for LU decomposition, a highly optimized Fortran implementation by Intel.

Method \ Matrix	100x100	500x500	1000x1000	5000x5000	10000x10000
LUD	1705µs	9165µs	20163µs	1584932µs	11885985µs
Seq LUD	960931µs	111392020µs	-	-	-
Blocked LUD	2356µs	5958µs	12995µs	216774µs	1112225µs
SciPy	27320µs	29859µs	41966µs	717490µs	4905548µs

Table 7.3: Benchmark results of different LU decomposition methods. Time in µs is reported.

We observe that the sequential LUD is exceptionally slow in comparison to other methods. Its performance is so slow that it couldn't compute matrices larger than 500x500 within 10 minutes, hence its exclusion from the benchmark. This can be attributed to its non-parallel, naive sequential implementation. Conversely, the blocked version outperforms in all matrices larger than 100x100, demonstrating superior performance.

Surprisingly, SciPy performs quite poorly compared to Cholesky in NumPy (a lot more than 2x slower). Cholesky should only have half the number of operations as LU, so we expected it to be faster. We definitely don't expect it to be more than 100 times slower in some instances, but that turned out to be the case.

The blocked LU decomposition exhibits impressive speed for matrices exceeding 100x100 when compared to the other methods. This high-speed implementation, translated from a Rodinia benchmark [20], has been a benchmark component for Futhark for some time. As such, it is probable that any regressions that this implementation has experienced during Futhark's development have been caught. Additionally, the implementation has likely received considerable attention regarding compiler optimizations. We will now benchmark the LU implementations for solving multiple matrices in parallel.

Method \ Matrix	1000000x 5x5	1000000x 10x10	100000x 20x20	10000x 50x50	1000x 100x100	100x 500x500
LUD	4237µs	23685µs	15110µs	19187µs	14703µs	161251µs
Seq LUD	2127µs	12636µs	6033µs	19407µs	112546µs	13716390µs
Blocked LUD	16703µs	42849µs	22863µs	26905µs	3654µs	23886µs
SciPy	4980473µs	7842934µs	1141277µs	305864µs	415662µs	547019µs

Table 7.4: Benchmark results of different batched LU decomposition. Time in µs is reported.

From the batched benchmarks, we observe that the sequential version is exceedingly fast for matrices under 50x50. We suspect this speed is due to the overhead of the parallel constructs, such as map and reduce, being too significant for small

matrices. In these benchmarks, the compiler is unaware of the matrices' sizes at compile time, thus hindering it from performing optimizations accordingly. The straightforward parallel LUD implementation is quickest for a batch of matrices sized 50x50. However, for matrices of 100x100 and 500x500, the blocked version proves to be the fastest. Similar to Cholesky, we note that the sequential implementation is actually computes 1000x100x100 faster than a single 100x100 matrix. This unusual occurrence leads us to suspect that the compiler may have conducted some unfortunate optimizations. As expected, the SciPy implementation is considerably slow, but this outcome aligns with our expectations.

7.1.3 Solving the system ($Ax = b$)

In this section, we will compare the performance of various methods used for solving a single vector equation. For our benchmarking, we will gauge the performance of Futhark implementations against the NumPy implementation `numpy.linalg.solve`. The methods that will be benchmarked include: sequential and parallel Cholesky, sequential and blocked LU decomposition, Gauss-Jordan elimination, and the Conjugate Gradient method. We will also assess the performance of the batched versions of these methods. Given that we are benchmarking both Cholesky and Conjugate Gradient, it is required that A is symmetric positive definite. Accordingly, we will generate a random symmetric positive definite matrix A for each benchmark and a random vector b.

Method \ Matrix	100x100	500x500	1000x1000	5000x5000	10000x10000
Cholesky Dot	4019µs	20123µs	43868µs	351368µs	1493065µs
LU Blocked	4261µs	11726µs	25118µs	297169µs	2543884µs
Gauss	5949µs	29988µs	59787µs	1687173µs	12111056µs
Gauss NP	1251µs	5717µs	14226µs	1508537µs	11739530µs
CG	5531µs	31212µs	70790µs	1090442µs	6805977µs
NumPy	414µs	1861µs	10192µs	595741µs	4100007µs

Table 7.5: Benchmark results of different linear equation solver methods. Time in µs is reported.

We did not include the sequential versions of LU and Cholesky for the benchmark shown in table 7.5. This is because they are very slow for large matrices, as we have seen in the previous benchmarks. We see that Guass elimination without pivoting is the fastest Futhark method for 100x100 although still slower than NumPy. For matrices 500x500 and larger the blocked LU implementation is the fastest, except for 10000x10000, where Cholesky pulls ahead. Conjugate gradient

(CG) is also fairly fast, but is not able to beat any of the other methods for any of the sizes. In our tests, the conjugate gradient method has been shown to use a lot more iterations for larger matrices. For matrices smaller 100x100 ($n = 100$), it often converges to a good solution with much less than n iterations, but this is not the case for larger matrices.

Method \ Matrix	1000000x 5x5	1000000x 10x10	100000x 20x20	10000x 50x50	1000x 100x100	100x 500x500
Cholesky Dot	16490µs	134174µs	112757µs	128873µs	542807µs	76504121µs
Cholesky Banachiewicz/Seq	3277µs	16538µs	6785µs	13612µs	47305µs	8005949µs
LU Blocked	22105µs	121299µs	55674µs	19744µs	16670µs	165624µs
LU Seq	4287µs	23794µs	10142µs	22631µs	115883µs	13773200µs
Gauss	4136µs	30050µs	21799µs	134416µs	551759µs	59807991µs
Gauss NP	3498µs	18227µs	11890µs	16575µs	13325µs	155387µs
CG	2343µs	13164µs	9343µs	8986µs	9072µs	117261µs
NumPy	2397694µs	2734681µs	1036028µs	373783µs	140247µs	202333µs

Table 7.6: Benchmark results for the different linear algebra methods.

We did not include the sequential versions of LU and Cholesky in the benchmark presented in table 7.5. This is due to their sluggish performance with large matrices. We observed that Gauss elimination without pivoting is the fastest Futhark method for 100x100 matrices, although it is still slower than NumPy. For matrices sized 500x500 and larger, the blocked LU implementation proves to be the fastest, except for 10000x10000 matrices, where Cholesky takes the lead. The Conjugate Gradient (CG) method is also relatively quick, but it cannot outperform any of the other methods for any of the given sizes. In our testing, the CG method was found to require significantly more iterations for larger matrices. For matrices smaller than 100x100 ($n = 100$), it often converges to a satisfactory solution with less than n iterations. However, this is not the case for larger matrices. It is worth noting that for the optimization algorithms, we are likely to have a better starting point for the conjugate method. This is because we can utilize the solution from the previous iteration as a starting point for the next iteration. For these benchmarks, we simply selected a starting point of all zeros.

7.2 Portfolio Optimization

In this section, we will look at the benchmarks for solving various portfolio optimization problems. We will look at the different optimization algorithms and how they perform for different sizes of the problem and with different methods for solving the linear systems of equations. The results will be compared to CVXPY.

7.2.1 Unconstrained

For the unconstrained problem we will look at the following mean-variance problem:

$$\text{Minimize} \quad \lambda \mu^T x - x^T \Sigma x \tag{7.1}$$

where:

- λ is a scalar parameter that balances return and risk.
- μ is a vector in $\mathbb{R}^n$ of expected returns.
- Σ is an $n \times n$ covariance matrix of returns.
- x is the vector in $\mathbb{R}^n$ that we're solving for.

Solving this problem amounts to finding an efficient portfolio, but we have no constraints on the sum of the weights, and we allow negative values. This means that we can short-sell and leverage as much as we want. We will benchmark Newton's method and gradient descent with and without parallel backtracking.

Method \ Dim	100x100	500x500	1000x1000	5000x5000	10000x10000
LU Blocked	4238µs	12118µs	26513µs	398021µs	3339618µs
LU Blocked LS	4382µs	11984µs	26384µs	397840µs	3337620µs
LU Blocked Par-LS	4331µs	12324µs	27367µs	401075µs	3347706µs
Cholesky Dot	4395µs	22026µs	43337µs	444724µs	2280493µs
CVXPY	301778µs	1030464µs	9981167µs	-	-

Table 7.7: Benchmark results of Newtons method with different linear equation solvers compared to CVXPY. Time in µs is reported. "-" indicates the method exceeded the time limit of 10 minutes. LS indicates that it used backtracking line search and Par-LS indicates that it used parallel backtracking line search. All other methods use fixed step size of 1.

Table 7.7 displays the results of the benchmark. It is evident that we can solve the problem significantly faster than CVXPY for all sizes. Newton's method converges incredibly quickly for this problem. Within 2-3 iterations, it delivers a solution that falls below the tolerance of $1e{-}8$. We see that the performance impact of line search is not very big, but the parallel version is actually slower than the sequential version. This is likely because the sequential version rarely needs to backtrack, so the overhead of the parallel version is not worth it.

Even though the underlying implementation for the default solver (OSQP) in CVXPY is written in C/C++, the user defines the problem in Python. Conse-

quently, the performance is subject to the considerable overhead of the Python interpreter. Moreover, OSQP is a general-purpose solver, considerably more robust than our specialized solver. It is likely to provide a more accurate solution. This is possibly why we are able to surpass CVXPY by such a large margin. CVXPY could not produce a solution for 5000x5000 and 10000x10000 matrices within the time limit of 10 minutes. We observe the same trends as in the previous benchmarks for solving linear systems of equations. The blocked LU implementation proves to be the fastest for all matrices, excluding 10000x10000.

Method \ Matrix	1000000x 5x5	1000000x 10x10	100000x 20x20	10000x 50x50	1000x 100x100	100x 500x500
Cholesky Banachiewicz/Seq	1840µs	26570µs	21636µs	17048µs	38656µs	6820933µs
Cholesky Banachiewicz/Seq LS	2277µs	25879µs	15691µs	16761µs	38950µs	5603183µs
Cholesky Banachiewicz/Seq Par-LS	4575µs	41468µs	18845µs	24962µs	126396µs	7935806µs
Cholesky Dot	4010µs	11145µs	18422µs	39673µs	64009µs	13008815µs
CG	3318µs	22163µs	16825µs	50623µs	285931µs	30334389µs
Cvxpy	1654617786µs	1724611043µs	200374889µs	55150985µs	313039350µs	122469816µs

Table 7.8: Benchmark results of Newtons method solving a batch of problems with different linear equation solvers. Time in µs is reported. LS indicates that it used backtracking line search and Par-LS indicates that it used parallel backtracking line search. All other methods use fixed step size of 1.

As previously mentioned in section 6.2.5, we can not use the blocked LU decomposition because this is rejected by the compiler. We achieve very good results with the sequential Cholesky implementation. CVXPY has no fair chance here, as it is not designed to solve many small problems in parallel.

7.2.2 Equality Constrained

For the equality constrained problem we will look at the following mean-variance problem:

$$\text{Minimize} \quad x^T \Sigma x \tag{7.2}$$

$$\mu^T x = e \tag{7.3}$$

where:

- μ is a vector in $\mathbb{R}^n$ of expected returns.
- Σ is an $n \times n$ covariance matrix of returns.
- x is the vector in $\mathbb{R}^n$ that we're solving for.
- e is a scalar parameter that represents the expected return.

This problem is similar to the unconstrained problem, but we have a constraint that the expected return of the portfolio is equal to a given value e. We will benchmark Newton's method with equality constraints against CVXPY.

Method \ Dim	100x100	500x500	1000x1000	5000x5000	10000x10000
Blocked LU	10099µs	23710µs	50771µs	596213µs	7645244µs
LU	6916µs	35543µs	74481µs	3364351µs	37224828µs
CVXPY	290593µs	3208202µs	12529201µs	-	-

Table 7.9: Benchmark results of Newtons method for equality constraints with Blocked LU and normal parallel LU. Time in µs is reported. "-" indicates that the problem was not solved within 10 minutes.

We see that we are still able to beat CVXPY with a sizeable margin for all problem sizes. However, the margin has decreased compared to the unconstrained problem. Blocked LU still offers superior performance.

Method \ Matrix	1000000x 5x5	1000000x 10x10	100000x 20x20	10000x 50x50	1000x 100x100	100x 500x500
Seq LU	5874µs	29754µs	17829µs	46418µs	211924µs	21089124µs
Cvxpy	1506617546µs	1538765192µs	172541308µs	52582622µs	277966309µs	112454667µs

Table 7.10: Benchmark results of Newtons method for equality constraints solving a batch of problems. Time in µs is reported.

Again, we are not able to use the blocked LU decomposition because of compiler limitations, but we see very good results with the sequential Cholesky implementation. CVXPY still has no fair chance, as it is not designed to solve many small problems at once.

7.2.3 Inequality Constrained

For the inequality constrained problem we will look at the following mean-variance problem:

$$\begin{aligned} \text{Minimize} \quad & x^T \Sigma x && (7.4)\\ & \mu^T x \geq r && (7.5)\\ & esg^T x \geq e && (7.6)\\ & x \geq 0 && (7.7)\\ & \sum_{i=1}^{n} x_i = 1 && (7.8) \end{aligned}$$

where:

- μ is a vector in $\mathbb{R}^n$ of expected returns.
- Σ is an $n \times n$ covariance matrix of returns.
- x is the vector in $\mathbb{R}^n$ that we're solving for.
- r is a scalar parameter that represents the minimum expected return.
- esg is a vector in $\mathbb{R}^n$ of ESG scores.
- e is a scalar parameter that represents the minimum ESG score.

This problem is similar to the equality constrained problem, but we have additional constraints that the expected return of the portfolio is greater than or equal to a given value r, the ESG score of the portfolio is greater than or equal to a given value e, and that the portfolio is long only. We will benchmark the barrier method and ADMM against CVXPY.

Method \ Dim	100x100	500x500	1000x1000	5000x5000	10000x10000
Barrier Blocked LU	118998µs	281977µs	657501µs	11423628µs	98800274µs
Barrier Blocked LU JVP	216995µs	17840563µs	-	-	-
Barrier Blocked LU VJP	405230µs	29044586µs	-	-	-
ADMM Blocked LU	1197242µs	3704565µs	8030992µs	124910869µs	-
ADMM CG	1767997µs	10400217µs	19616014µs	-	-
CVXPY	333689µs	1039506µs	12988537µs	-	-

Table 7.11: Benchmark results of barrier method and ADMM with different linear equation solver. Time in µs is reported. "-" indicates the problem was not solved within 10 minutes or ran out of memory.

We see that we are still able to beat CVXPY for all sizes, but the margin has decreased again. We also see that the ADMM method is not able to solve the larger problems within 10 minutes. This is because the ADMM method requires a lot of iterations to converge, although each iteration is very fast. It is very susceptible to the choice of its parameter ρ and the stopping criteria. The barrier method is able to solve larger problems, but it is also very sensitive to the choice of its parameter μ. A μ of 3 has been chosen for these benchmarks, but a larger value would likely cause it not to converge. The barrier method converges after roughly 21 iterations of its outer loop. A lower value would cause it to have a higher number of iterations and thus be slower. We also benchmarked with the use of the built-in functions for auto-differentiation `jvp` and `vjp`. We ran out of memory with the `jvp` function for the 1000x1000 problem. This is likely a bug within the compiler. `vjp` exceeded the 10-minute time limit for problems above 500x500, and we see that `jvp` actually performs better (more comments on this below).

Method \ Matrix	1000000x 5x5	1000000x 10x10	100000x 20x20	10000x 50x50	1000x 100x100	100x 500x500
Barrier	381133µs	1823316µs	552519µs	879378µs	3281759µs	282061036µs
Barrier JVP	920946µs	4887080µs	5974507µs	15708619µs	27219190µs	-
Barrier VJP	275395µs	1320658µs	725943µs	1402458µs	4266512µs	-
ADMM LU	2507170µs	10113011µs	4766267µs	37977384µs	139446494µs	-
ADMM CG	3664286µs	17110162µs	9227441µs	51621440µs	344393814µs	-
Cvxpy	2034657001µs	2039673566µs	220676279µs	57815551µs	304475522µs	167484545µs

Table 7.12: Benchmark results of different batched LU decomposition methods. Time in µs is reported.

We see that the barrier method is incredibly fast at solving many small optimization problems in parallel. ADMM is quite a lot slower than the barrier method, but it is still faster than CVXPY. ADMM is slower than the barrier method because it requires more iterations to converge. The barrier method was the only out of the Futhark implementations able to provide a solution within 10 minutes for 100 optimization problems with 500 variables/securities (100x500x500). Also, it is clear from table 7.11 that 500x500 problems should just be solved sequentially because the barrier method is more than 100 times faster at solving one 500x500 than 100 of them in a map.

We see that the effect on performance for `jvp` and `vjp` is much smaller for small problem sizes. We would expect `vjp` to be faster for our implementation than `jvp`, because `vjp` should be most efficient for functions that have more inputs than outputs [8]. This is also what we are seeing and is actually faster than hand-derived derivatives for smaller matrices. For the unbatched benchmarks, we saw the opposite, and the reason remains unknown.

7.3 Test on S&P500

The S&P500 is a stock market index that measures the stock performance of 500 of the largest companies listed on stock exchanges in the United States. It is one of the most commonly followed equity indexes. In this section, we will test our implementation on the S&P500 index along with ESG scores, aiming to find the portfolios with the lowest risk for a certain level of return and ESG. The esg provider is Apex ESG Enterprise [25]. This data spans a period of 5 years, from 2013 to 2018. The ESG values range from 537 to 1699, with higher values being better. Similarly, the return values range from -0.69 to 0.79, with higher being better. The return values indicate the anual mean return.

We will tackle two problems. The first is to find the portfolio with the lowest risk but with a return and ESG higher than 0.30 and 1000, respectively. The second problem is to solve a grid of optimization problems optimizing for risk at different levels of return and ESG and display the results as a 3D plot. We will solve for 40000 different combinations of return and ESG, but we will only include 20 stocks in the portfolio. The results will then be compared to those obtained using CVXPY. For the first problem, we present the following results:

Measurement	**CVXPY**	**Futhark**
ESG	1115.816	1113.103
Return	0.3	0.3
Risk	0.001466	0.001442
Weight Sum	1.000000	1.000003
Ticker and Weight (Largest 5)	'SO': 0.034953 'HLT': 0.096116 'DXC': 0.141200 'DWDP': 0.141894 'APTV': 0.426892	'SO': 0.034755 'HLT': 0.097354 'DXC': 0.141700 'DWDP': 0.142108 'APTV': 0.421119

Table 7.13: S&P500 Portfolio with lowest risk for return and ESG higher than 0.30 and 1000 respectively.

We can see that we actually achieve a lower risk than CVXPY while still satisfying the constraints. This is surprising, but the differences are minor. The weights appear very similar. The second task is to efficiently solve a grid of optimization problems, each with varying levels of return and ESG, and display the results as a 3D scatter plot. We will solve for 40,000 different combinations of return and ESG. Observe the figures below:

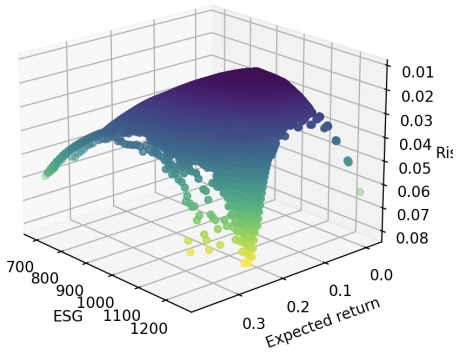


Figure 7.1: CVXPY

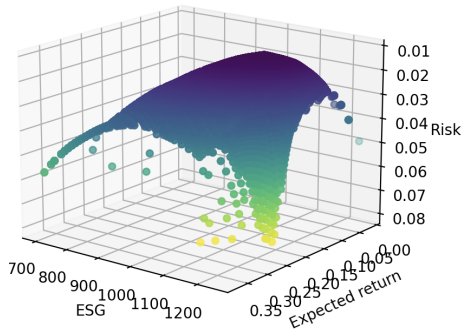


Figure 7.2: Futhark

The results are once again very similar. The overall shape of the surfaces is the same, but we see some noise in different places for both solutions. We observe that the noise occurs around the edges of the surfaces, which indicates that those portfolios are likely not optimal or feasible. The results serve as a good confidence boost in the Futhark implementation's ability to solve these problems at a satisfactory level of precision.

The Futhark implementation is also much faster than CVXPY, taking only 2.3 seconds to solve all 40,000 problems, while CVXPY takes 2 minutes on CPU.

Chapter 8

Conclusion and Future work

8.1 Conclusion

The objective of this thesis was to explore the parallelization of convex optimization for portfolio optimization problems. This investigation led to the introduction of three new Futhark libraries. The first library offers techniques for solving linear systems of equations. The second library leverages the first to solve convex optimization problems with various types of constraints, using Newton's method, the barrier method, and ADMM. The third library utilizes the second for solving portfolio optimization problems.

It effectively compiles efficient GPU code, and benchmarks indicate that it in some cases outperforms CVXPY, a renowned Python library for solving convex optimization problems. The Futhark implementation particularly shines in handling numerous small optimization problems in parallel or problems with an extensive number of variables.

Tested on S&P500 data, the implementation's results align closely with those achieved via CVXPY. Although portfolio optimization served as the case study for this work, the implementation is versatile and can be applied to other convex optimization problems, extending its utility to various fields.

Futhark's high-level language properties make it an excellent tool for prototyping and developing data-parallel algorithms for convex optimization problems. The insights from this thesis can serve as foundational knowledge for future research in the realm of parallel convex optimization.

8.2 Future Work

The work presented in this thesis can serve as a basis for further investigations into parallel convex optimization. Here are several suggestions for continued research:

- Explore alternative methods for addressing convex optimization problems, such as the application of the primal-dual interior-point method.
- Enhance the robustness of the implementation by incorporating additional checks for infeasibility and unboundedness.
- Assess the potential for a blocked Cholesky decomposition approach to solving linear equation systems.
- Consider the use of infeasible-start Newton's method as a means to find a feasible starting point for the barrier method.
- Contemplate partitioning the ADMM subproblems into smaller components and parallelizing their solutions.
- Examine the influence of the builtin auto-differentiation on performance and seek explanations for any observed decline in performance when it is employed.
- Explore the potential application of the module to other convex optimization problems.

Bibliography

[1] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates", in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: ACM, 2017, pp. 556–571, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062354.

[2] ETF Database, *Socially responsible etf list*, https://etfdb.com/esg-investing/social-issues/, Accessed May 15, 2023.

[3] *Welcome to CVXPY 1.3 2014; CVXPY 1.3 documentation — cvxpy.org*, https://www.cvxpy.org/, [Accessed 15-May-2023].

[4] *Why Futhark? — futhark-lang.org*, https://futhark-lang.org/index.html, [Accessed 15-May-2023].

[5] T. Henriksen and M. Elsman, "Towards size-dependent types for array programming", in *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ser. ARRAY 2021, Virtual, Canada: Association for Computing Machinery, 2021, 1â€“14, ISBN: 9781450384667. DOI: 10.1145/3460944.3464310. [Online]. Available: https://doi.org/10.1145/3460944.3464310.

[6] R. Zadeh. "Cme 323: Distributed algorithms and optimization, lecture 1: Overview, models of computation, brent's theorem". Scribed by Andreas Santucci. (Apr. 3, 2017), [Online]. Available: https://stanford.edu/~rezab/dao/notes/lecture01/cme323_lec1.pdf.

[7] M. Elsman, T. Henriksen, and C. E. Oancea, "Parallel programming in futhark", in *Parallel Programming in Futhark — futhark-book.readthedocs.io*, ch. 6. A Parallel Cost Model for Futhark Programs. [Online]. Available: https://futhark-book.readthedocs.io/en/latest/index.html# (visited on 05/15/2023).

[8] R. Schenck, O. RÃ¸nning, T. Henriksen, and C. E. Oancea, "Ad for an array language with nested parallelism", in *2022 SC22: International Confer-*

ence for High Performance Computing, Networking, Storage and Analysis (SC) (SC), Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2022, pp. 829–843. [Online]. Available: https://doi.ieeecomputersociety.org/.

[9] M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea, "Static interpretation of higher-order modules in futhark: Functional gpu programming in the large", *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, 97:1–97:30, Jul. 2018, ISSN: 2475-1421. DOI: 10.1145/3236792. [Online]. Available: http://doi.acm.org/10.1145/3236792.

[10] H. Markowitz, "Portfolio selection*", *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952. DOI: https://doi.org/10.1111/j.1540-6261.1952.tb01525.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1540-6261.1952.tb01525.x. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-6261.1952.tb01525.x.

[11] *The Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel 1990 — nobelprize.org*, https://www.nobelprize.org/prizes/economic-sciences/1990/press-release/, [Accessed 16-May-2023].

[12] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[13] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2e. New York, NY, USA: Springer, 2006.

[14] S. Boyd and L. Vandenberghe, "Convex optimization", in Cambridge university press, 2004, ch. 9.5, pp. 484–487.

[15] G. Garrigos and R. M. Gower, *Handbook of Convergence Theorems for (Stochastic) Gradient Methods*. Paris, France: Université Paris Cité and Sorbonne Université, CNRS, Laboratoire de Probabilités, Statistique et Modélisation; Center for Computational Mathematics Flatiron Institute, New York, Jan. 2023.

[16] S. Boyd, *Distributed optimization and statistical learning via the alternating direction method of multipliers*, 2010. DOI: 10.1561/9781601984616.

[17] Z. Liu, *An ADMM-Newton method for inequality constrained optimization — towardsdatascience.com*, https://towardsdatascience.com/an-admm-newton-method-for-inequality-constrained-optimization-37a470c58a5c, [Accessed 21-May-2023].

[18] A. Roberts, A. Shi, and Y. Sun, *Conjugate gradient methods*, [Online; accessed 28-May-2023], 2021. [Online]. Available: https://optimization.cbe.cornell.edu/index.php?title=Conjugate_gradient_methods.

[19] K. U. Weihe, K. Q. Hansen, and P. K. Larsen, "Linear algebra in futhark", Department of Computer Science, University of Copenhagen, Tech. Rep., Feb. 2021.

[20] K. Wang, S. Che, and K. Skadron, *Rodinia benchmark suite 3.1*, https://github.com/yuhc/gpu-rodinia, Contact: kw5na@virginia.edu, sc5nf@cs.virginia.edu, skadron@cs.virginia.edu, 2023. [Online]. Available: https://github.com/yuhc/gpu-rodinia.

[21] Wikipedia, *Cholesky decomposition*, (Accessed on May 30, 2023), 2023. [Online]. Available: https://en.wikipedia.org/wiki/Cholesky_decomposition.

[22] M. Angeletti, J.-M. Bonny, and J. Koko, "Parallel euclidean distance matrix computation on big datasets", HAL, Tech. Rep. ffhal-02047514f, 2019. [Online]. Available: https://hal.archives-ouvertes.fr/ffhal-02047514f.

[23] J. H. Jung, "Cholesky decomposition and linear programming on a gpu", Scholarly Paper Directed by Dianne P. O'Leary, 2023.

[24] Y. Xu, M. Liu, Q. Lin, and T. Yang, "Admm without a fixed penalty parameter: Faster convergence with new adaptive penalization", 2023.

[25] *Apex esg enterprise*, https://www.esgenterprise.com, Accessed: 2023-07-03, 2023. [Online]. Available: https://www.esgenterprise.com.