Utrecht University

Mathematics

Bachelor thesis

Markowtiz' Critical Line Algorithm

Author:
Michael van den Hoogenband
3758230

Supervisor:
Dr. Karma Dajani

January 18, 2017

Universiteit Utrecht

Abstract

The goal of this thesis is to give a detailed theoretical background into the workings of the Critical Line Algorithm created by Harry Markowitz. First, we will give an introduction into probability theory, and then some background information on portfolio theory and optimization theory. Furthermore there will be a description, and solution of the problem for selecting an optimal portfolio. Finally the used algorithm is explained, both the working and the code.

Contents

1 Introduction

In the current society, the industry is focused on profit. How can I make as much profit as possible in the shortest amount of time with the least amount of work. We want to have efficient ways to make a lot money. A way of earning money is by investing in assets. We want to put together a collection of assets which we think will give use the best profit, without having to big of a risk. Nowadays, there are many theories on how to assemble such a collection, i.e. a portfolio. There are even companies who specialize in selecting these portfolios, but all of this is made possible by the groundbreaking work of Harry Markowitz.

In 1952, Markowitz published his article *Portfolio Selection* in the Journal of Finance [2]. In this article Markowitz explained how one can derive the optimal combination of a high expected return against a low variance of return on a portfolio. Although people were well aware that one must always look at the return as well as the risk of an investment, Markowitz article gave a well detailed theory on portfolio optimization. This formed the foundation of research in financial mathematics. Markowitz even recieved the Nobel prize in Economics for his work, together with William Sharpe and Merton Miller in 1990.

In the first section of this thesis, we will discuss a lot of theory. First on probability theory, then optimization theory and finally the work of Markowitz considering portfolio theory will be covered. The remaining part of section 2 is devoted to forming a mathematical understanding of the problem of portfolio optimization and its solution. In section 3 we will first look at the working of the algorithm created by Markowitz before we describe the exact working of the algorithms code in Python, as it is provided in [1]. We will finish with a small numerical example showing the working of the algorithm. The Python-code we discuss in section 3 can be found in appendix A. In this appendix there are some extra functionalities which this thesis does not cover, like finding the maximum Sharpe ratio. If one is interested in these functionalities, the article of Bailey gives some explanations as well as some references.

When reading this thesis, one should eventually understand the working of the CLA. One should also be able to understand the given code and, with enough experience in a different programming language, should be able to rewrite this code to that language.

2 Optimizing your portfolio

Before we can give a good description of the problem, we first want to have some theoretical background with respect to financial mathematics and probability theory[1].

2.1 Theoretical background

To get a clear view on the use of the algorithm, we first want to give some background on financial mathematics and some crucial definitions. For the optimization problem we face here, which is a constrained problem, we want to compute the *Efficient Frontier.*

Definition 2.1. *We call an investment risk-free if it has a guaranteed future return.*

The risk of an investment is measured by the standard deviation on its return, which is defined in definition 2.11. An example of a risk-free investment is a government bond. A government bond is a bond the government put op for sale. This bond returns a certain interest each agreed period, such as a month, and return the face value at maturity. The inflation and currency risks are negligible, so you have a certainty that you receive the money you loaned to the government and the interest.

Definition 2.2. *The excess return is the return rate on an investment relative to the return rate on a risk-free investment.*

Suppose the return rate of the investment you want to make is 18%. You then look at the return rate of a risk-free investment which you can make with the same amount of money, which is for example 15%. Your excess return is then $18\% - 15\% = 3\%$.

Definition 2.3. *The Efficient Frontier is the set of portfolios that yield the highest achievable mean excess return, in excess of the risk-free rate, for any given level of risk.*

We can now give the formulation of the optimization problem. We want to minimize the portfolio's variance subject to a targeted excess return. To define the variance, we first need to give a definition of the mean value of a random variable, as well as the definition of a random variable.

Definition 2.4. *Suppose we have an experiment E with several outcomes. The set of all possible outcomes is called a sample space and is denoted by Ω.*

For an example, let us look at the experiment of throwing a die. The possible outcomes of our experiment are the values 1 to 6, so $\Omega = \{1, 2, 3, 4, 5, 6\}$.

[1]Although the theory supplied in this thesis should be sufficient, one can find more information on probability and statistics in [6] and [7]

Definition 2.5. *Suppose $\mathcal{F}$ is a family of subsets of a sample space Ω. We call $\mathcal{F}$ a σ-algebra if it satisfies the following three properties:*

1. $\emptyset, \Omega \in \mathcal{F}$

2. *Suppose $A \subset \Omega$. $A \in \mathcal{F} \Rightarrow A^c \in \mathcal{F}$*

3. *Suppose $A_1, A_2, \ldots, A_n, \ldots$ is a countable family of subsets in $\mathcal{F}$. Then $\cup_{i \geq 1} A_i \in \mathcal{F}$.*

We call the elements of $\mathcal{F}$ events.

Definition 2.6. *We call a function $F : \Omega \to \mathbb{R}$ measurable with respect to a σ-algebra $\mathcal{F}$ if the inverse image of every interval $[a, b]$, $a \leq b \in \mathbb{R}$, is in $\mathcal{F}$. More precisely if*

$$a \leq F \leq b \equiv \{\omega \in \Omega : a \leq F(\omega) \leq b\} \in \mathcal{F}.$$

F is then called $\mathcal{F}$-measurable.

Definition 2.7. *A function $\mathbb{P} : \mathcal{F} \to [0, 1]$ is called a probability measure on a sample space Ω with σ-algebra $\mathcal{F}$ if*

1. $\mathbb{P}(\Omega) = 1$

2. *$A_1, A_2, \ldots, A_n, \ldots$ is a countable family of disjoint events in $\mathcal{F}$, then*

$$\mathbb{P}\left(\cup_i A_i\right) = \sum_i \mathbb{P}(A_i).$$

Definition 2.8. *A probability space is a triple $(\Omega, \mathcal{F}, \mathbb{P})$, where $\mathbb{P}$ is a probability measure on the sample space Ω with σ-algebra $\mathcal{F}$.*

Definition 2.9. *Suppose we have a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. A random variable on this probability space is a function $X : \Omega \to \mathbb{R}$ which is $\mathcal{F}$-measurable.*

Definition 2.10. *The mean, or expected, value μ of a discrete random variable X is the average of all possible values of X, weighted by their probabilities and is denoted by E[X]. More precisely, suppose $\mathbb{P}(X = x_i) = p_i$, $x_i \in X$, then E[X] is denoted by*

$$E[X] = \sum_i p_i x_i.$$

When X has a continuous distribution, its expected value is defined by

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx,$$

where the probability measure $\mathbb{P}$ is defined by f(x).

Definition 2.11. *The variance of X, Var(X), is the mean squared deviation of X from its expected value E[X]. It is given by*

$$Var(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2.$$

The standard deviation of a random variable X is denoted by σ and defined as

$$\sigma(X) = \sqrt{Var(X)}.$$

In constructing the problem we will also come across the covariance and more specifically the covariance matrix.

Definition 2.12. *Suppose we have two random variables X and Y. The distribution of (X, Y) is called a joint distribution and is determined by*

$$\mathbb{P}(x, y) = \mathbb{P}(X = x, Y = y),$$

which must satisfy

$$\mathbb{P}(x, y) \geq 0 \quad \textit{and} \quad \sum_{\textit{all } (x,y)} \mathbb{P}(x, y) = 1.$$

A quick and simple example of the joint distribution of $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$ can be found in the next table.

	x_1	x_2
y_1	0.2	0.15
y_2	0.4	0.25

Definition 2.13. *The covariance of two jointly distributed random variables X and Y is the average value of the product of the deviation of X from its mean and the deviation of Y from its mean. So*

$$Cov(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y].$$

Notice that $Cov(X, X) = Var(X)$. Now suppose we have two random variables X and Y. The covariance matrix of X and Y is given by

$$\begin{pmatrix} Cov(X, X) & Cov(X, Y) \\ Cov(Y, X) & Cov(Y, Y) \end{pmatrix}.$$

We can clearly see that $Cov(X, Y) = Cov(Y, X)$, which means that our covariance matrix is symmetric. With the covariance defined, we can take a look at the variance of the sum of random variables.

Theorem 2.1. *Let X and Y be random variables. The variance of the sum of these random variables is given by*

$$Var(X + Y) = Var(X) + Var(Y) + 2Cov(X, Y).$$

Now let $X_1, \ldots, X_n$ all be random variables. The variance of $\sum_{i=1}^{n} X_i$ is given by

$$Var\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} Var(X_i) + 2\sum_{i \neq j} Cov(X_i, X_j).$$

Proof. We will use the definition of the variance to proof our theorem. We know that $Var(X) = E[X^2] - E[X]^2$. Now we substitute X for $X + Y$. We use the linearity property of the expected value to get

$$\begin{aligned} Var(X+Y) &= E[(X+Y)^2] - E[(X+Y)]^2 = E[X^2 + 2XY + Y^2] - (E[X] + E[Y])^2 \\ &= E[X^2] + 2E[XY] + E[Y^2] - E[X]^2 - 2E[X]E[Y] - E[Y]^2 \\ &= E[X^2] - E[X]^2 + E[Y^2] - E[Y]^2 + 2(E[XY] - E[X]E[Y]) \\ &= Var(X) + Var(Y) + 2Cov(X, Y). \end{aligned}$$

For the linear combination of more than two random variables, we can just repeat the steps. □

Definition 2.14. *We call a $(n \times n)$-matrix M positive definite if for every non-zero column vector v of n elements, the scalar $v^T M v$ is positive.*

Definition 2.15. *A stochastic process is a collection of random variables on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ that is indexed by a set T and can be written as*

$$\{X(t) : t \in T\}.$$

Definition 2.16. *A time homogeneous invariant is a stochastic process whose distribution does not depend on the reference time t.*[2] *More precisely, suppose we have an stochastic process $X(t)$, $t \in \mathbb{R}$. We call this process time homogeneous if*

$$\mathbb{P}(X(t_i)|X(t_i - a)) = \mathbb{P}(X(t_j)|X(t_j - a)), \quad \forall t_{i,j}, \quad \forall a > 0.$$

2.2 Optimization theory

Also essential for our theoretical background, is some information on general optimization theory. Simply put, optimization is about finding the best possible outcome of a problem in a big pool of solutions. Mathematically speaking, we want to find a solution vector $x^* \in \mathbb{R}^n$, such that, for a objective function $f : \mathbb{R}^n \to \mathbb{R}$, holds

$$f(x^*) = \min\{f(x) : x \in \mathbb{R}^n\}.$$

A simple example is finding the minimum value of a function. Suppose $f(x) = x^2 + 3$. We know we can find the extreme point of this function by differentiating the function and setting it equal to 0. When we do this, we see that $f'(x) = 2x = 0 \Rightarrow x = 0$. This solution method is known for this kind of simple problems, but mostly we will be faced with much more complex problems.

One of the major fields of optimization theory is convex optimization. Let us first look at the definition of a *convex set* and a *convex function*.

Definition 2.17. *Suppose we have a set $X \in \mathbb{R}^n$, two points $x_i, x_j \in X$, $i, j \in \{1, \ldots, n\}$, and a straight line y connecting x_i and x_j. We call X a convex set if for every point t on y holds that $t \in X$.*

[2]More information about this subject can be found in [4].

Definition 2.18. *Suppose we have a convex set X. A function $f : X \to \mathbb{R}$ is called convex if $\forall\theta$, $0 \leq \theta \leq 1$ and $\forall x_i, x_j \in X$, $i, j \in \{1, \dots, n\}$ holds that*

$$f(\theta x_i + (1-\theta)x_j) \leq \theta f(x_i) + (1-\theta)f(x_j).$$

We call a function strictly convex if $\forall\theta$, $0 < \theta < 1$ and $\forall x_i \neq x_j \in X$, $i, j \in \{1, \dots, n\}$ we have

$$f(\theta x_i + (1-\theta)x_j) < \theta f(x_i) + (1-\theta)f(x_j).$$

In a convex optimization problem, we have a convex objective function $f(x) : \mathbb{R}^n \to \mathbb{R}$ which we want to minimize, given a certain set of equality and inequality constraints, which our optimal solution x^* must satisfy. This gives us the following system to solve:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0 \\ & h_i(x) = 0. \end{aligned}$$

Here $g_i(x)$ are all convex functions and $h_i(x)$ are linear functions.

There are different methods for solving convex optimization problems. When we have a problem that only has equality constraints, we can use the Lagrange function. With the help of Lagrange multipliers we create a new objective function, which we call the Lagrangian. The main idea is that, when we have a solution for the objective function without considering the constraints, we can find a Lagrange multiplier such that we find a solution for the Lagrangian, and with that we can solve the problem with the equality conditions. As an example, suppose we have a two-dimensional problem.

$$\begin{aligned} \text{minimize} \quad & f(x, y) \\ \text{subject to} \quad & g(x, y) = c. \end{aligned}$$

Our goal is to find a solution for minimizing $f(x, y)$. We can do this by solving the following system of equations:

$$\begin{aligned} \frac{\partial f(x,y)}{\partial x} &= 0 \\ \frac{\partial f(x,y)}{\partial y} &= 0. \end{aligned}$$

We now find a set of points (x_0, y_0) which are extreme points on f. However, we want to know if these extreme points also lay on our constraint function g. Therefor, the equation $\nabla f = \lambda \nabla g$ must hold, where $\nabla f = (\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y})$ and is called the gradient of f. Here λ is called a Lagrange multiplier and we can compute the Lagrangian:

$$L[x, y, \lambda] = f(x, y) - \lambda(g(x, y) - c).$$

When we find the solution for $\nabla L = 0$, which gives us the extreme points of the Lagrangian, we find a solution for $\nabla f = \lambda \nabla g$. Notice that $\nabla L = 0$ gives us:

$$\begin{aligned}\frac{\partial L}{\partial x} &= \frac{\partial f}{\partial x} - \lambda \frac{\partial g}{\partial x} = 0 \\ \frac{\partial L}{\partial y} &= \frac{\partial f}{\partial y} - \lambda \frac{\partial g}{\partial y} = 0 \\ \frac{\partial L}{\partial \lambda} &= -(g(x,y) - c) = 0,\end{aligned}$$

which is exactly the system that needs to be solved for $\nabla f = \lambda \nabla g$, together with our original constraint. When solved, we can fill in the x and y components of the extrema of the Lagrangian in our objective function. This gives us all the local minima and maxima, as well as the global minimum and maximum, the latter of which are the solution of our optimization problem.

The problem we have at hand here is a quadratic programming problem which is subject to some linear constraints, which we will sketch later on. Let us first look at the general form of a quadratic programming problem.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is called quadratic if it has the form

$$f(x) = \alpha + \sum_{j=1}^{n} c_j x_j + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} q_{ij} x_i x_j,$$

whereas the factor 1/2 is for simplifying the function when we look at the first derivative of f. When we rewrite this to a form with matrices, we get

$$f(x) = \alpha + c^T x + \frac{1}{2} x^T Q x,$$

with

$$c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \quad Q = \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & \ddots & & \vdots \\ \vdots & & & \\ q_{n1} & \cdots & & q_{nn} \end{pmatrix}$$

Suppose we have a n-dimensional problem with a given total of m constraints, either equalities of inequalities. As follows from above, what we need for the problem are a $(n \times 1)$ vector $\mathbf{v}$, a symmetric $(n \times n)$-matrix M, a $(m \times n)$ matrix A and a $(m \times 1)$ vector $\mathbf{b}$. All elements of these matrices and vectors must be real-valued. The reason that M should be symmetric is for convenience in solving the problem. We assume M is symmetric because

$$x^T M x = (x^T M x)^T = x^T M^T x = \frac{1}{2}\left(x^T M x + x^T M^T x\right) = x^T \left(\frac{M + M^T}{2}\right) x,$$

so we can replace our matrix M by the symmetric matrix $(M + M^T)/2$. Now our goal is to find a vector $\mathbf{x} \in \mathbb{R}^n$ which satisfies the following conditions:

$$\begin{aligned} &\text{minimize} && \frac{1}{2}x^T Mx - v^T x \\ &\text{subject to} && Ax \leq b \end{aligned}$$

These problems are in general more complicated than the convex problems we discussed earlier. Depending on what kind of constraints you have, quadratic or linear, the problem gets easier to solve. When our matrix M is positive definite, the quadratic problem becomes a convex problem, which we can solve with a lot more ease.

2.3 Portfolio theory

When Markowitz published his paper on selecting an optimal portfolio he made some assumptions. The first one is that an investor always wants to maximize the expected return of his portfolio. The second one is that an investor wants as little risk as possible when investing. Hence, he wants to minimize the variance of his return, because the greater the variance, the greater the standard deviation, which leads to a bigger risk. This would mean that an investor does not prefer a diversified portfolio over a non-diversified portfolio. This contradicts the logic of choosing a diversified portfolio because it is less risky to spread your investments. Furthermore, the process is done in a single time-period, which means we buy all our assets at time $t = 0$ and sell them all again at time $t = 1$. The last one is that short selling assets is not permitted. When short selling assets, you sell assets that are not in your possession but are lent to you, but which you need to buy back after a certain amount of time. The reason why you might want to do this, is if you suspect the price of a certain asset will drop in the future. Therefore you can sell for a certain price and buy the assets back at a later time when the price is dropped. This brings a lot of risk with it, because you need to buy back the assets, even if their price has risen in the meantime, which means you have lost money.

Under these assumptions Markowitz created a model for selecting the optimal portfolio, which is called the Markowitz model and which we will describe here. Suppose we have a portfolio with n assets. Let r_i be a random variable which is the actual return of asset $i \in N = \{1, 2, \ldots, n\}$, μ_i be the expected return of asset i and let X_i be the relative amount invested in asset i. Because we do not allow short selling it holds that $X_i \geq 0$ and since X_i is the relative amount invested, we have the constraint that $\sum_{i=1}^{n} X_i = 1$. Now, the return R of your complete portfolio can be denoted by

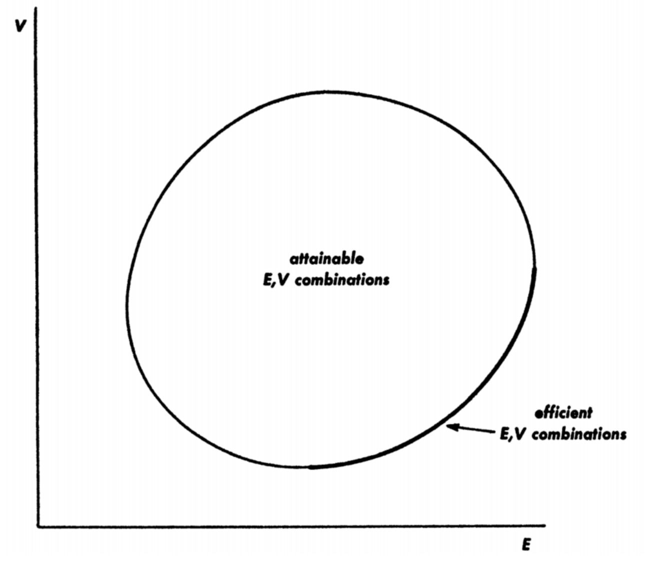
$$R = \sum_{i=1}^{n} r_i X_i.$$

Since each of the r_i are random variables, we notice that R is also a random variable, for it is a linear combination of random variables. We now further denote the covariance between two returns r_i and r_j as σ_{ij}. We can now give descriptions for the expected value and the variance of R, which are

$$E[R] = \sum_{i=1}^{n} X_i \mu_i$$
$$Var(R) = \sum_{j=1}^{n} \sum_{i=1}^{n} \sigma_{ij} X_i X_j$$

Since in our assumptions we have said that an investor wants maximize the expected return E while minimizing the variance V, we introduce the so called $E-V$ rule. This rule says that an investor, when faced with a constant variance V, wants to maximize his expected return E or, when faced with a constant expected return E, wants to minimize the variance V. Markowitz' represented this graphically in figure 1.

Figure 1: Graphical representation of E-V rule [2]



Our algorithm computes these optimal $E-V$ combinations. However, just as Markowitz, we will also give a geometrical illustration of how to find a solution to the portfolio selection problem for a small amount of assets.

Let us consider the case in which we have three assets. Having three assets, the

Markowitz model reduces to the following form:

$$
\begin{aligned}
&E = \sum_{i=1}^{3} X_i \mu_i \\
&V = \sum_{j=1}^{3} \sum_{i=1}^{3} \sigma_{ij} X_i X_j \\
&\sum_{i=1}^{3} X_i = 1 \\
&X_i \geq 0, \quad i = 1, 2, 3
\end{aligned}
$$

We can write $X_3 = 1 - X_1 - X_2$ and therefore we can write our system of equations in a two dimensional way, which makes it easier to give a graphical representation. Our problem reduces to

$$
\begin{aligned}
&E = E(X_1, X_2) \\
&V = V(X_1, X_2) \\
&X_1, X_2 \geq 0 \\
&1 - X_1 - X_2 \geq 0
\end{aligned}
$$

The precise formulas are not important for our cause right now, but they can be written out. Our attainable set of portfolios are the ones that satisfy the last constraint and $X_3 = 1 - X_1 - X_2$. The feasible set of portfolios is represented in figure 2 as the triangle a, b, c. It is clear why every point outside of this triangle does not satisfy the conditions. We now define the isomean curve and the isovariance curve.

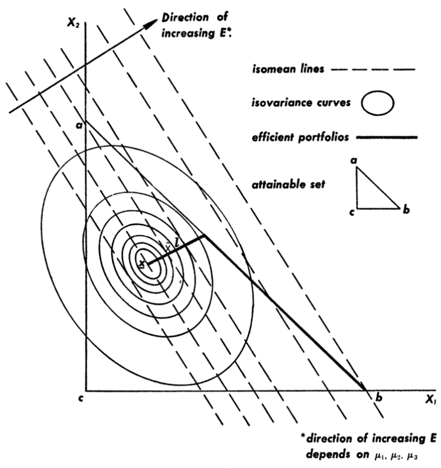
Definition 2.19. *An isomean curve is the set of portfolios which have a given expected return and variable variance.*
An isovariance curve is the set of portfolios which have a given variance of return and a variable expected return.

By looking at the formulas for our expected value and variance, we see that the isomean curves take on the form of a system of straight parallel lines. This can be concluded by expressing X_2 in terms of X_1

$$X_2 = \frac{E - \mu_3}{\mu_2 - \mu_3} - \frac{\mu_1 - \mu_3}{\mu_2 - \mu_3} X_1.$$

We see that the slope of the line is equal for every E and therefore only the intercept of the line will change when we vary E. Thus we conclude that these are indeed a set of parallel straight lines. Moreover we can see that our isovariance curves form a system of concentric ellipses. We will also show this, although it will be a bit more complicated than our previous claim. We define X as the so-called center of the curves, which is the point with minimal V. We will denote the expected value and variance of X by E_X and V_X respectively. Now for every point $Y \neq X$ it holds that $V_Y > V_X$. X does not need to lay within the set of feasible solutions. We can even say that when X is indeed inside our set of feasible solutions, it is the optimal solution. Because there is no portfolio with a variance smaller than V_X for a fixed E or a higher expected return than E_X for a fixed V. So we call our point X efficient. When we fix E for all our isomean curves, we can find a set of solutions $\hat{X}(E)$ for which we find a minimal V, which will be where the isomean curve lies tangent to the isovariance curve. Together with the constraints on X_1, X_2 and X_3, we can find the efficient line l, which is shown in figure 2. Figure 3 shows an example for which X lays outside of the feasible area. The line l is a combination of little line segments, between each of the $\hat{X}(E)$. One end is the point were we have minimum variance, while the other end is the point of maximum expected return.

Figure 2: Geometrical representation of the isomean and isovarance curves[2]


X_2
Direction of increasing E*
isomean lines
isovariance curves
efficient portfolios
attainable set
a
b
c
X_1
l
*direction of increasing E depends on μ_1, μ_2, μ_3

Now that we have seen how the set of efficient portfolios behave in the case of three assets, we can look at how the $E - V$ combinations behave when we set out E against V. We can see that E is of the form $E = a_o + a_1X_1 + a_2X_2$, which gives us a plane, and that V is of the form $V = b_0 + b_1X_1 + b_2X_2 + b_{12}X_1X_2 + b_{11}X_1^2 + b_{22}X_2^2$, which is a paraboloid. When we plot this plane and paraboloid over the set of efficient portfolios (figure 5), we see that our E provides a set of connected line segments and our V provides a set of connected parabola segments. Now plotting our E against our E over the efficient portfolios, we obtain again a set of connected parabola segments (figure 4). This result for our three asset universe is easily expanded to a universe with n assets.

Figure 3: Isomean and Isovariance curve with X outside of the feasible area[2]

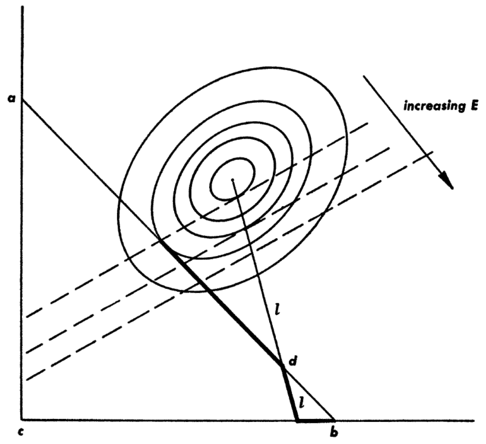
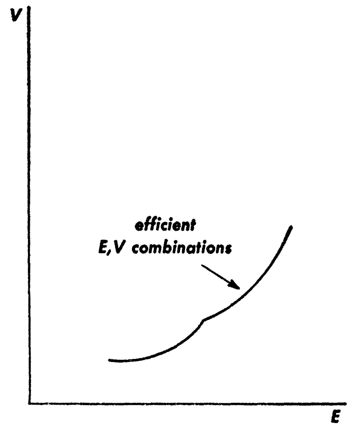
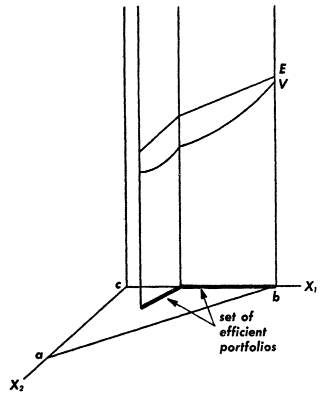


Figure 4: E plotted against V[2]



Although the $E - V$ rule does not always imply that a diverse portfolio is desirable over a non diverse portfolio, it gives a much bigger set of possibilities. Of course it can still happen that there is one portfolio which has the highest expected return and the minimal variance over all the other portfolios. In that case one might want to consider going against his reason and invest in the non-diversified portfolio.

Figure 5: E and V plotted over efficient portfolios[2]



2.4 The problem

Now that we have some definitions covered, it is time to look at the problem at hand. We consider an investment universe of n assets. Given are the positive definite $(n \times n)$ covariance matrix Σ and the returns of the assets which have mean μ, which is represented in a $(n \times 1)$ vector. Our observations must be time-homogeneous, because then we do not have to re-estimate our μ and Σ in every step. That is why we compute our μ and Σ on time-homogeneous invariants.

Because we want to optimize our portfolio, we want to find the linear combination of assets weights which give the highest excess return, or better said, we want to minimize the variance of our return. These weights are represented in the $(n \times 1)$ vector $\omega= (\omega_1, \omega_2, \dots, \omega_n)$. Each weight ω_i has a lower bound l_i and an upper bound u_i, which are represented by the $(n \times 1)$ vector l and u, so we have that $l_i \leq \omega_i \leq u_i$. Furthermore we have the constraint that $\sum_{i=1}^{n} \omega_i = 1$. Because we want to minimize the variance subject to a targeted excess return, we also have an extra constraint with respect to that targeted return μ_p. Let us sum up our problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\omega^T \Sigma \omega \\ \text{subject to} & l_i \leq \omega_i \leq u_i \\ & \sum_{i=1}^{n} \omega_i \cdot \mu_i = \mu_p \\ & \sum_{i=1}^{n} \omega_i = 1 \end{array}$$

To see that our objective function really is about minimizing the variance of return, we will write out our matrices. First we denote the covariance of assets i and j by m_{ij}.

$$f(\omega_1, \omega_2, \ldots, \omega_n) = \frac{1}{2} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{pmatrix}^T \begin{pmatrix} m_{11} & m_{12} & \ldots & m_{1n} \\ m_{21} & \ddots & & \vdots \\ \vdots & & & \\ m_{n1} & \ldots & & m_{nn} \end{pmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{pmatrix}^T \begin{pmatrix} \omega_1 m_{11} + \omega_2 m_{12} + \ldots + \omega_n m_{1,n} \\ \omega_1 m_{21} + \omega_2 m_{22} + \ldots + \omega_n m_{2n} \\ \vdots \\ \omega_1 m_{n1} + \omega_2 m_{n2} + \ldots + \omega_n m_{nn} \end{pmatrix}$$

$$= \frac{1}{2}(\omega_1^2 m_{11} + \omega_1\omega_2 m_{12} + \ldots + \omega_1\omega_n m_{1n} + \omega_2\omega_1 m_{21} + \omega_2^2 m_{22} + \ldots + \omega_n\omega_2 m_{2n}$$
$$+ \omega_n\omega_1 m_{n1} + \omega_n\omega_2 m_{n2} + \ldots + \omega_n^2 m_{nn}.)$$

Since our covariance matrix is symmetric, we see that $m_{ij} = m_{ji}$. There we can rewrite the last equation to

$$f(\omega) = \frac{1}{2}\left(\omega_1^2 m_{11} + \omega_2^2 m_{22} + \ldots + \omega_n^2 m_{nn} + 2\omega_1\omega_2 m_{12} + \ldots + 2\omega_n\omega_{n-1} m_{n-1n}\right)$$
$$= \frac{1}{2}\left(\sum_{i=1}^{n} \omega_i^2 m_{ii} + 2\sum_{i,j=1, i\neq j}^{n} \omega_i\omega_j m_{ij}\right)$$
$$= \frac{1}{2}\left(\sum_{i=1}^{n} \omega_i^2 Var(X_i) + 2\sum_{i,j=1, i\neq j}^{n} \omega_i\omega_j Cov(X_i, X_j)\right),$$

which is the weighted variance of the sum of our assets. Like we said earlier, because our constraints are linear and our covariance matrix Σ is positive definite, we can solve this problem as a convex problem and thus use the Lagrangian. This will be done in the next subsection.

We also define a set of free assets.

Definition 2.20. *An asset is called free when its weight is free, which means that the weight of the asset lies strictly between its bounds, so* $l_i < \omega_i < u_i$*.*
A bounded asset is an asset whose weight lies exactly on one of its bounds, so $l_i = \omega_i$ *or* $\omega_i = u_i$*.*

The set of free assets is denoted by $\mathbb{F}$ and the set of bounded assets is denoted by $\mathbb{B}$. We can now rewrite our vectors and matrix in terms of $\mathbb{F}$ and $\mathbb{B}$, where $\mathbb{F}$ has size k and $\mathbb{B}$ has size $n-k$. This gives us

$$\mu = \begin{bmatrix} \mu_F \\ \mu_B \end{bmatrix}, \omega = \begin{bmatrix} \omega_F \\ \omega_B \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_F & \Sigma_{FB} \\ \Sigma_{BF} & \Sigma_B \end{bmatrix}$$

It holds that $\Sigma_{BF} = \Sigma_{FB}^T$

2.5 Approaching the solution

We do not know yet how to solve exactly the constrained problem described above. We call this the constrained problem, because of the lower and upper bounds of each weight. However, we do know how to solve the unconstrained problem. In this case we still have the constraint that the weights should add up to one. Since our goal is to minimize the portfolios variance with respect to a targeted excess return μ_p, we can use Lagrange Function. We add the Lagrange multipliers λ and γ, to create a new optimization problem, without these constraints. This is given by

$$L[\omega, \gamma, \lambda] = \frac{1}{2}\omega^T \Sigma \omega - \gamma(\omega^T 1_n - 1) - \lambda(\omega^T \mu - \mu_p).$$

In the function 1_n is the $(n \times 1)$ vector of ones. When we differentiate with respect to all the parameters of the Lagrange function, setting them equal to zero, we retrieve a system of $(n+2)$ linear equations which we can solve. We call the solution ω^*.

When we find an optimal portfolio whose assets satisfy the boundary conditions, we call such a portfolio a constrained minimum variance portfolio.

Definition 2.21. *We call a constrained minimum variance portfolio a turning point when all other surrounding constrained minimum variance portfolios contain different free assets.*

This is useful in solving our constrained problem. When we look at the solution space, specifically away from the turning points, we see that in these regions the upper and lower bounds are negligible with respect to the free assets. This means that between two turning points, the solution of the constrained problem can be given by solving the unconstrained problem on the free assets. This way, we have divided our problem into multiple unconstrained problems.

In other words, we need to solve

$$L[w,\gamma,\lambda] = \frac{1}{2}\begin{bmatrix}\omega_F\\ \omega_B\end{bmatrix}^T \begin{bmatrix}\Sigma_F & \Sigma_{FB}\\ \Sigma_{BF} & \Sigma_B\end{bmatrix}\begin{bmatrix}\omega_F\\ \omega_B\end{bmatrix} - \gamma\left(\begin{bmatrix}\omega_F\\ \omega_B\end{bmatrix}^T \cdot \begin{bmatrix}1_k\\ 1_{n-k}\end{bmatrix} - 1\right)$$
$$- \lambda\left(\begin{bmatrix}\omega_F\\ \omega_B\end{bmatrix}^T \cdot \begin{bmatrix}\mu_f\\ \mu_B\end{bmatrix} - \mu_p\right)$$

Here ω_B is fixed, while ω_F is unknown and is part of the constraints of our minimization problem. The algorithm works by starting with the turning point with the highest expected return and then calculating the next lower turning point. From [2] and [3] we have learned that the we can derive the efficient frontier by constructing a convex combination of two neighboring turning points, following the results given in section 2.3. That is why we want to find the optimal portfolio in each turning point. To do this, we want to find an expression for γ, which we can use, together with a value of λ, to find the value of ω_F in the next turning point. We start by differentiating our Lagrangian with respect to ω_F. For a clearer view of our function, we will first rewrite our Lagrangian as a function without matrices. This gives us

$$\begin{aligned} L[\omega,\gamma,\lambda] = &\frac{1}{2}\left(\omega_F^T\Sigma_F\omega_F + \omega_F^T\Sigma_{FB}\omega_B + \omega_B^T\Sigma_{BF}\omega_F + \omega_B^T\Sigma_B\omega_B\right)\\ &- \gamma(\omega_F^T 1_k + \omega_B^T 1_{n-k} - 1) - \lambda(\omega_F^T\mu_F + \omega_B^T\mu_B - \mu_p)\end{aligned}$$

When we differentiate this, we get

$$\begin{aligned}\frac{\partial L}{\partial\omega_F} &= \frac{1}{2}(\Sigma_F\omega_F + \omega_F^T\Sigma_F + \Sigma_{FB}\omega_B + \omega_B^T\Sigma_{BF}) - \gamma 1_k - \lambda\mu_F\\ &= \frac{1}{2}(\Sigma_F\omega_F + (\omega_F^T\Sigma_F)^T + \Sigma_{FB}\omega_B + (\omega_B^T\Sigma_{BF})^T) - \gamma 1_k - \lambda\mu_F\\ &= \Sigma_F\omega_F + \Sigma_{FB}\omega_B - \gamma 1_k - \lambda\mu_F.\end{aligned}$$

Setting this function equal to 0, we find that

$$\gamma 1_k = \Sigma_F\omega_F + \Sigma_{FB}\omega_B - \lambda\mu_F.$$

Because we now make a distinction between free and bounded assets, we rewrite our constraint on the weight vector $\omega\cdot 1 = 1$ as $1_k^T\omega_F = 1 - 1_{n-k}^T\omega_B$. Now we can find an expression for γ, by combining our found function and the rewritten constraint.

$$\begin{aligned}\gamma 1_k &= \Sigma_F\omega_F + \Sigma_{FB}\omega_B - \lambda\mu_F\\ \Sigma_F^{-1}\gamma 1_k &= \omega_F + \Sigma_F^{-1}\Sigma_{FB}\omega_B - \Sigma_F^{-1}\lambda\mu_F\\ \gamma 1_k^T\Sigma_F^{-1}1_k &= 1_k^T\omega_F + 1_k^T\Sigma_F^{-1}\Sigma_{FB}\omega_B - \lambda 1_k^T\Sigma_F^{-1}\mu_F\\ \gamma &= \frac{1 - 1_{n-k}^T\omega_B + 1_k^T\Sigma_F^{-1}\Sigma_{FB}\omega_B}{1_k^T\Sigma_F^{-1}1_k} - \lambda\frac{1_k^T\Sigma_F^{-1}\mu_F}{1_k^T\Sigma_F^{-1}1_k}\end{aligned}$$

As we can see, γ now depends only on the value of λ, because all the other values are fixed at this point. With the expression we have found for γ, we can also find an expression for λ. By substituting the expression for γ in the function

$$\omega_F = -\Sigma_F^{-1}\Sigma_{FB}\omega_B + \Sigma_F^{-1}\lambda\mu_F + \Sigma_F^{-1}\gamma 1_k.$$

This gives us ω_F as a linear function which only depends on λ. We can find the value of $\lambda^{(i)}$, which is the value for the specific asset i, with the function

$$\begin{aligned}\lambda^{(i)} = &\frac{1}{C_i}[(1 - 1_{n-k}^T\omega_B + 1_k^T\Sigma_F^{-1}\Sigma_{FB}\omega_B)(\Sigma_F^{-1}1_k)_i \\ &- (1_k^T\Sigma_F^{-1}1_k)(b_i + (\Sigma_F^{-1}\Sigma_{FB}\omega_B)_i)]\end{aligned}$$

in which

$$C_i = -(1_k^T\Sigma_F^{-1}1_k)(\Sigma_F^{-1}\mu_F)_i + (1_k^T\Sigma_F^{-1}\mu_F)(\Sigma_F^{-1}1_k)_i$$

and the value of b_i depends on the situation we are facing, which will become clearer in the next section.

3 Critical Line Algorithm (CLA)

3.1 Explaining of the algorithm

Now it is time to take a real look at the algorithm. We first have to find a starting solution, from which we can start the iteration to find our optimal portfolio. The starting solution will be the turning point with the highest expected return value. To find this turning point, we first want to order our assets based on their expected return value. So the asset i with the highest expected return μ_i will become asset number 1 with expected return $\mu_1 \geq \mu_2 \geq \ldots \geq \mu_n$. We further define the value of λ that belongs to the turning point with the highest expected return value as λ_1. We claim that $\lambda_1 > \lambda_2 > \ldots > \lambda_T$, where T is the number of turning points. This claim is justified by proposition (12.2) from [5] and its proof, which we will now describe here as well. We say that λ and $\mu^T\omega$ (the expected return of a turning point) are related linearly.

Proposition 3.1. *[5] Between two turning point, λ and $\mu^T\omega$ are linearly related with a positive slope*

$$\frac{\partial \mu^T \omega(\lambda)}{\partial \lambda} > 0$$

Proof. First we determine three constants $C_{11}, C_{1\mu}$ and $C_{\mu\mu}$. These are defined as

$$C_{11} = 1_k^T \Sigma_F^{-1} 1_k, \quad C_{1\mu} = 1_k^T \Sigma_F^{-1} \mu_F, \quad C_{\mu\mu} = \mu_F^T 1_k \Sigma_F^{-1} \mu_F.$$

From our statements in section 2, we can now rewrite $\mu^T\omega$ as

$$\begin{aligned}\mu^T\omega &= \mu_F^T\omega_F + \mu_B^T\omega_B = -\mu^T\Sigma_F^{-1}\Sigma_{FB}\omega_B + \lambda\mu_F^T\Sigma_F^{-1}\mu_F + \gamma\mu_F^T\Sigma_F^{-1}1_k + \mu_B^T\omega_B \\ &= -\mu^T\Sigma_F^{-1}\Sigma_{FB}\omega_B + \lambda C_{\mu\mu} + \gamma C_{1\mu}^T + \mu_B^T\omega_B.\end{aligned}$$

Differentiating this formula with respect to λ, will give us

$$\frac{\partial \mu\omega}{\partial \lambda} = C_{\mu\mu} - \frac{(C_{1\mu}^2)^T}{C_{11}}.$$

Since Σ_F^{-1} is symmetric and it does not matter if we multiply with the vector 1_k or μ_F, we will ignore the transposing of $C_{1\mu}$. Putting that aside, since Σ_F does not change between two turning points, we conclude that $\mu\omega(\lambda)$ is indeed linear in λ with the above slope. Now all that is left is to prove that this slope is indeed positive.
Since Σ is positive-definite, it follows that Σ_F^{-1} is too positive-definite. We define a vector $x = 1_k - \alpha\mu_F$, $\alpha \in \mathbb{R}$. Then we can write $x^T\Sigma_F^{-1}x$ as

$$(1_k - -\alpha\mu_F)^T\Sigma_F^{-1}(1_k - \alpha\mu_F) = C_{11} - 2\alpha C_{1\mu} + \alpha^2 C_{\mu\mu} > 0.$$

We see that, because of the positive-definiteness of Σ_F^{-1}, there is no solution in α for $x^T\Sigma_F^{-1}x = 0$, which means the discriminant is negative and therefore

$$C_{11}C_{\mu\mu} - C_{1\mu}^2 > 0.$$

□

With this sorted list of assets, we now continue to change the weights ω_i. First we set all the weights equal to their lower bounds, such that $\forall i \in \{1, 2 \ldots, n\}$, $\omega_i = l_i$, which means all assets are in $\mathbb{B}$. The next step is to raise the weights one by one to their upper bounds, starting with $\omega_1 = u_1$. We continue to raise the next weight to its upper bound until $\sum_{i=1}^{n} \omega_i > 1$. There is an asset for which the sum of the weights goes beyond its borders, let us say asset j. We want to adapt the weight of asset j such that the sum of the weights equals 1 again, which means that $l_j < \omega_j < u_j$. Asset j will be our first free asset and will be moved to the set of free assets. So the weight of this one free asset is

$$\omega_j = 1 - \omega_{\mathbb{B}} \cdot 1_{n-1}.$$

So now we have found a weight vector ω in which all the elements except one lie on one of their bounds and we can use the Lagrangian to solve the problem. To find such a starting solution, or any solution at all, there are constraints on the bounds that we must consider. We can see clearly that for this to work, the inequality $\sum_{i=1}^{n} l_i \leq 1 \leq \sum_{i=1}^{n} u_i$ most hold. When either $\sum_{i=1}^{n} l_i = 1$ or $\sum_{i=1}^{n} u_i = 1$ holds, there will only be one portfolio which makes up the whole efficient frontier. This is the portfolio with $\omega_i = l_i$ or $\omega_i = u_i$ $\forall i \in N = \{1, 2, \ldots, n\}$ respectively. Moreover, when $\sum_{i=1}^{n} l_i > 1$ or $\sum_{i=1}^{n} u_i < 1$, there is no solution to our optimization problem.

With our starting solution found, we can go on and find the next turning point. We do this by lowering the λ belonging to our found turning point. When we do this, there are two possible outcomes. Either one of the free assets will shift and will be set on one of its bounds or one of the bounded assets will go away from its bound and become free. In order to compute the ω and λ of the next turning point we need to consider both these cases.

Let us start with the possibility that one of the free assets goes to one of its bounds. Remember that we have defined $\lambda^{(i)}$ as

$$\begin{aligned}\lambda^{(i)} = \frac{1}{C_i}[&(1 - 1_{n-k}^T \omega_B + 1_k^T \Sigma_F^{-1} \Sigma_{FB} \omega_B)(\Sigma_F^{-1} 1_k)_i \\ &- (1_k^T \Sigma_F^{-1} 1_k)(b_i + (\Sigma_F^{-1} \Sigma_{FB} \omega_B)_i)]\end{aligned}$$

in which

$$C_i = -(1_k^T \Sigma_F^{-1} 1_k)(\Sigma_F^{-1} \mu_F)_i + (1_k^T \Sigma_F^{-1} \mu_F)(\Sigma_F^{-1} 1_k)_i.$$

We define $\lambda_{current}$ as the λ belonging to our most recently found turning point. Let $\mathbb{F}$ be set of free assets such that it is just below this turning point, in other words such that $\lambda_{current} = \lambda_t > \lambda > \lambda_{t+1}$. We want to find the asset $i \in \mathbb{F}$ which goes to its bound, so we can compute our value $\lambda^{(i)}$ as described in the previous section. Notice that our $\mathbb{F}$ has k elements and that $\mathbb{F} = \{i_1, i_2, \ldots, i_k\}$ and that our i still has a value between 1 an n, as we sorted them while finding the starting solution. $\lambda^{(i)}$ is the point where the asset that moves to its bound actually reaches it. Here the value of b_i depends on the value of C_i. So

$$b_i = \begin{cases} u_i & \text{if } C_i > 0, \\ l_i & \text{if } C_i < 0. \end{cases}$$

One of our conditions is that $\mathbb{F}$ must always contain at least one element, so that means that this case can only happen when $k > 1$. C_i is equal to 0 if $\mu_i = \mu_j \; \forall i, j \in \mathbb{F}$. Now the $\lambda < \lambda_{current}$ at which a free asset goes to its bound and therefore will leave our subset $\mathbb{F}$ is

$$\lambda_{inside} = \max_{i \in \mathbb{F}} \{\lambda^{(i)}\}.$$

If either $k = 1$ or $C_i = 0 \; \forall i$, there does not exists a λ_{inside} at which a asset will leave $\mathbb{F}$. Our found λ_{inside} gives us the next lower turning point, but only when there is not a bounded asset that moves away from its bounds and thus becomes free and there is not a portfolio for which its respective λ satisfies the inequality $\lambda_{current} < \lambda < \lambda_{inside}$.

We now consider the other case, namely the case in which a bounded asset goes away form its bounds and becomes a free asset. We then find a portfolio in which we redefine our former $\mathbb{B}$ and $\mathbb{F}$. Let $i \in \mathbb{B}$ be the asset that goes away from its bound. We redefine our subsets to

$$\mathbb{F}_i = \mathbb{F} \cup \{i\}$$
$$\mathbb{B}_i = \mathbb{B} \setminus \{i\}.$$

We once again want to determine our value $\lambda^{(i)}$. Here the value of b_i is simply the value of $\omega_i = (\omega_{\mathbb{F}_i})_i$, which is either u_i or l_i, depending on which bound asset i lay. The λ at which our asset will become free will be denoted by $\lambda_{outside}$. It is defined by

$$\lambda_{outside} = \max_{i \in \mathbb{B}} \{\lambda^{(i)} | \lambda^{(i)} < \lambda_{current}\}.$$

Notice that if there is not a $\lambda^{(i)} < \lambda_{current}$, there does not exists a $\lambda_{outside}$.

With the cases we described above, we can now use our algorithm to find the next turning point. Which of the cases will occur, depends on the values of λ_{inside} and $\lambda_{outside}$. Suppose we have found a value for both λ_{inside} and $\lambda_{outside}$. Then our next turning point will have a value λ_{new}, where λ_{new} is defined by

$$\lambda_{new} = \max\{\lambda_{inside}, \lambda_{outside}\}.$$

So if $\lambda_{inside} > \lambda_{outside}$, a free asset will move towards its bounds. Of course, when one of our value λ_{inside} and $\lambda_{outside}$ does not have a value, the case of the one that has a value occurs. When λ_{new} is determined, we adjust our $\mathbb{F}$ and $\mathbb{B}$ by removing the asset i from its current subset and add it to the other one. When no solution is found for either λ_{inside} or $\lambda_{outside}$ or if $\lambda_{new} < 0$, we will terminate the algorithm. Since we follow the algorithm given in [1], we must calculate the *Minimum Variance portfolio*. This is the global minimum variance solution and serves as the most left bound of the efficient frontier. This portfolio has $\lambda = 0$ and the vector of means of the free assets $\mu_{\mathbb{F}}$ only consists of zeroes.

With every new turning point, we have different subsets $\mathbb{F}$ and $\mathbb{B}$. This means that in every iteration, the algorithm must calculate the covariance matrix and its inverse, which are quite expensive calculations. To reduce these costs, section 12.3.3 of [5] gives two lemmas, one for the case in which an asset is added to $\mathbb{F}$ and one for the case where an asset is removed from $\mathbb{F}$.

3.2 Implementation in Python

__init__

In this part we will describe the code of the algorithm which can be found in appendix A. At first, there are a few things that should be initialized. We initialize the class with four parameters, the $(n \times 1)$ vectors $mean, lB$ and uB, containing the means, the lower bounds and the upper bounds of the assets respectively. We also define the $(n \times n)$ matrix $covar$, which is the covariance matrix. Each of these parameters is defined again in the class, assigning the given value to them. Furthermore we define four empty arrays, which will contain the solution vector ω, the λ, γ and assets in $\mathbb{F}$ belonging to a turning point.

initAlgo

The next function that is called is the solve function, which computes the turning points, the free set $\mathbb{F}$ and the corresponding weights for each turning point. However since this function calls upon many other functions, we will discuss this function later. We will first discuss the function *initAlgo*, which initializes the algorithm. First there is created an array a of tuples $(0, 0)$, in which the first 0 will function as the index of the asset and the second denotes its expected return. The expected values of the assets are denoted in an other array b. Then the tuples in a become (i, μ_i) for all assets i. Our array b now becomes the sorted version of a, where the array is sorted by the value of μ. So for example, we will get $b = [(3, 7), (1, 3), (2, 2.5), (0, 1)]$. Then the first free weight is determined. First we denote i as the number of assets $+1$ and give a solution vector w which is equal to the vector of lower bounds lB. As long as the sum of w is smaller than 1, we will decrease i with 1 and set the weight of asset i to its upper bound. When the loop as ended, the weight of asset i is determined and the function will return the solution vector w and the index of the asset that becomes free.

getB, diffLists, getMatrices& reduceMatrices

The next function being called in *solve* is the function *getMatrices*. This function takes the array f, the array of free assets, and reduces all our parameters.

To reduce a matrix or vector, it calls upon the function $reduceMatrix$. This function has three parameters, a matrix (or vector), $matrix$, to be reduced, and two lists, one who provides a list of rows, $listX$, and one who provides a list of columns, $listY$. If either of this lists is empty, the function terminates. Otherwise, a new matrix will be created, $matrix\_$, which contains only the column of $matrix$ of element $listY[0]$. Then for every remaining element in $listY$, the columns of $matrix$ corresponding to those elements will be added to $matrix\_$. Then another new matrix is created, $matrix\_\_$. This matrix consists of row $listX[0]$ of $matrix\_$. Then, just as before, for the remaining elements of $listX$, the rows of $matrix\_$ will be added to $matrix\_\_$. Then $matrix\_\_$ is our reduced matrix and will be returned. Now $getMatrices$ determines $covarF$ with $matrix = covar$ and $listX = listY$ and $meanF$ with $matrix = mean$, $listX = f$ and $listY = [0]$.
To determine $covarFB(\Sigma_{FB})$, $getMatrices$ calls upon a list b, which is determined by $getB$. $getB$ depends only on the parameter f and calls upon the function $diffLists$, which has $list1$ and $list2$ as parameters. $diffLists$ returns a list of the elements that are in $list1$ but not in $list2$. $getB$ uses a list of all the assets as $list1$ and uses the list of free assets as $list2$. Now $covarFB$ can be determined using $matrix = covar$, $listX = f$ and $listY = b$. Finally wB is determined using the last element the current solution w, which is declared in $solve$, as $matrix$ and using $listX = b$ and $listY = [0]$, the list of one element. All of these matrices and vectors are returned.

computeLambda & computeBi

computeLambda is used in the determination of λ_{inside} and $\lambda_{outside}$. The function calls upon a lot of parameters, most of which are given by *getMatrices*. So we need matrices *covarF_inv* (Σ_F^{-1}) and *covarFB* (Σ_{FB}), vectors *meanF* (μ_f) and *wB*, an index i for computing the lambda belonging to an asset and a variable *bi*. *covarF_inv* is computed in the *solve* method, by using the built-in function of python *numpy.linalg.inv*(*covarF*). First the value C is computed. This is done by computing the vector 1_k, where k is the length of *meanF*. Then the constants c_1 and c_3 and the vector c_2 and c_4 are computed using the built-in function *numpy.dot* which provides a matrix multiplication. It gives us

$$c_1 = 1_k^T \Sigma_F^{-1} 1_k, \quad c_2 = \Sigma_F^{-1} \mu_F, \quad c_3 = 1_k^T \Sigma_F^{-1} \mu_F, \quad c_4 = \Sigma_F^{-1} 1_k.$$

The c is computed by $c = -c_1 * c_2[i] + c_3 * c_4[i]$. If c is equal to 0, the function is terminated. With this c and the given parameter *bi*, the function *computeBi* is called upon, if the parameter *bi* is a list, which in our case is the list consisting of the lower and upper bound of an assets weight. When *computeBi* is called, it returns $uB[i]$ if $c > 0$ and $lB[i]$ if $c < 0$. Then λ is calculated. If wB is empty, it means that all assets are free and $\lambda = (c_4[i] - c_1 * bi)/c$ and λ and *bi* are returned. If there are elements in wB, then $1_{n-k} = 1_B$, the vector of ones the same size as wB, is determined. Then l_1, l_2 and l_3 are determined. $l_1 = 1_B^T wB$, l_2 is first determined as $l_2 = \Sigma_F^{-1}\Sigma_{FB}$, then $l_3 = l_2 wB$ and l_2 is redefined as $l_2 = 1_k l_3$. Then λ and *bi* are returned, where $\lambda = ((1 - l_1 + l_2) * c_4[i] - c_1 * (bi + l_3[i]))/c$.

computeW

This function computes vector of the weights of the assets ω. As parameters is uses again *covarF_inv*, *covarFB*, *meanF* and *wB*, for which we will use the same notation as we used for *computeLambda*. First the corresponding γ is computed. To do this, once again 1_k is determined. Then g_1 and g_2 are computed as

$$g_1 = 1_k^T \Sigma_F^{-1} \mu_F, \quad g_2 = 1_k^T \Sigma_F^{-1} 1_k.$$

Eventually $\omega = -w_1 + g * w_2 + \lambda * w_3$ and $g = \gamma$ are returned, but w_1, w_2, w_3 and g need to be determined. Here λ is the last element of the vector l of lambdas, where the λ found in *computeLambda* is added to, depending on the case, which are described in section 3.1 and which we will see in the solve function. Again, if wB has no elements, $\gamma = -\lambda * g_1/g_2 + 1/g_2$ and $w_1 = 0$. Otherwise, 1_B is determined, just as in *computeLambda* and g_3, g_4 and w_1 are determined as

$$g_3 = 1_B^T wB, \quad g_4 = \Sigma_F^{-1} \Sigma_{FB}, \quad w_1 = g_4 wB, \quad g_4 = 1_k^T w_1,$$

where g_4 is redetermined when w_1 is determined. Then $\gamma = -\lambda * g_1/g_2 + (1 - g_3 + g_4)/g_2$. Finally $w_2 = \Sigma_F^{-1} 1_k$ and $w_3 = \Sigma_F^{-1} \mu_F$ are determined. Then the weight vector $\omega = -w_1 + g * w_2 + \lambda * w_3$ and g are returned.

solve

Now that we have defined all the functions necessary to find a solution, we can look at the *solve* function. First f, w are computed through *initAlgo*. Then the returned value of w from *initAlgo* is stored in the solution vector w defined in *__init__*. The value $None$ is added to the vectors l, g of λ and γ from *__init__* and the free weight returned from *initAlgo* is stored in the list f from *__init__*. Then $\lambda_{inside} = l\_in$ and $\lambda_{outside} = l\_out$ are computed, along with the minimum variance portfolio. First we look at computing l_in. l_in is declared as $None$. Since a constraint on $\mathbb{F}$ is that it should never be empty, the function demands that $len(f) > 1$. When this is true, *covarF*, *covarFB*, *meanF* and *wB* are determined with $getMatrices$ for parameter f and *covarF_inv* is calculated like in *computeLambda*. A counter j is set equal to 0. Then, for every element i in f, l and bi are determined using *computeLambda*. For the parameter i of *computeLambda* we use j and for the variable bi of *computeLambda*, the list with the lower and upper bound of i is used. If the returned value of l is bigger than l_in, l_in, i_in and bi_i are set to l, i and bi respectively. Otherwise, they remain at their current value. In any case, j is raised by one, and we find the biggest l as l_in.
For l_out we do something similar. First l_out is set to $None$. Then the demand is made that the size of f should be smaller than the length of the mean vector given as parameter in *__init__*. If this is the case, b is determined by $getB$, given the parameter f. Then for every element i in b *covarF*, *covarFB*, *meanF* and *wB* are determined with $getMatrices$ for parameter $f + [i]$ and *covarF_inv* is calculated. Then again l and bi are calculated with *computeLambda*, where the

parameter i is the size of $meanF$ -1 and the variable bi is the current weight ($lB[i]$ or $uB[i]$) of i in b. Then if the last element of the vector l is $None$ or the returned variable l from *computeLambda* is smaller than the last element of the vector l, the variable l is smaller than l_out, then l_out and i_out are set to l and i. Then we have found the biggest l_out.
If l_in and l_out are either equal to $None$ or smaller than 0, the minimum variance portfolio is calculated by adding 0 to the list of lambdas, computing $covarF$, $covarFB$, $meanF$ and wB, calculating $covarF\_inv$ and setting all the elements of $meanF$ to 0. Otherwise, either l_in or l_out is added to our vector l. If l_in is bigger than l_out, l_in is added to our vector l, asset i_in is removed from the list f and $w[i\_in]$ is set to bi. If l_out is bigger, l_out is added to our vector l and asset i_out is added to our list f. With these new f, $covarF$, $covarFB$, $meanF$ and wB are determined and $covarF\_inv$ is calculated, which we will use in computing our solution vector w, which is the last step.
First WF and g are determined with the help of *computeW* with the parameters given above. Then, for every element i in the range of the length of f $(0, \ldots, k)$, the weight of element $f[i]$ is set to the weight of element i in wF, so $w[f[i]] = wF[i]$. The solution is again stored in our total solution vector w for $\_\_init\_\_$, the found γ is added to the vector g and the new f replaces the old one. This whole process, from the determination of l_in and l_out to finding the solution vector w, is repeated until the last item of the vector l is 0, which means that the minimum variance portfolio is calculated and we have our entire solution set.

3.3 A small numerical example

To illustrate the algorithm, we will look at the numerical example given in [1]. In this case we have an investment universe of ten assets. The bounds on our weights are all equal, the lower bounds are set to 0 and the upper bounds are set to 1. We also set an implicit condition that $\sum_{i=1}^{10} \omega_i = 1$. The values of the mean vector and the covariance matrix, as well as the values of the lower and upper bounds can be found in table 1. Our program will read in this information

Table 1: Lower bounds, Upper bounds, Mean and Covariance

LB	0	0	0	0	0	0	0	0	0	0
UB	1	1	1	1	1	1	1	1	1	1
μ	1.175	1.19	0.396	1.12	0.346	0.679	0.089	0.73	0.481	1.08
Cov	0.4075516									
	0.0317584	0.9063047								
	0.0518392	0.0313639	0.194909							
	0.056639	0.0268726	0.0440849	0.1952847						
	0.0330226	0.0191717	0.0300677	0.0277735	0.3405911					
	0.0082778	0.0093438	0.0132274	0.0052667	0.0077706	0.1598387				
	0.0216594	0.0249504	0.0352597	0.0137581	0.0206784	0.0210558	0.6805671			
	0.0133242	0.0076104	0.0115493	0.0078088	0.0073641	0.0051869	0.0137788	0.9552692		
	0.0343476	0.0287487	0.0427563	0.0291418	0.0254266	0.0172374	0.0462703	0.0106553	0.3168158	
	0.022499	0.0133687	0.020573	0.0164038	0.0128408	0.0072378	0.0192609	0.0076096	0.0185432	0.1107929

from a *.csv* file and will then solve the problem. It will return some output, which are a list of turning point (TP), and for each turning point the return (R), the risk, the corresponding values of λ and γ and the weights of the assets in the free set. Note that when an asset $X(i)$ does not belong to $\mathbb{F}$, it weight will have value 0, for it will be added to $\mathbb{F}$ later. The results can be found in table 2. As we can see from table 2 the starting solution is found when asset 2 is free.

Table 2: Return, Risk, λ and $\omega_i \in \mathbb{F}$ of the turning points

TP	R	Risk	λ	$X(1)$	$X(2)$	$X(3)$	$X(4)$	$X(5)$	$X(6)$	$X(7)$	$X(8)$	$X(9)$	$X(10)$
1	1.190	0.952	58.303	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	1.180	0.546	4.174	0.649	0.351	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
3	1.160	0.417	1.946	0.434	0.231	0.000	0.335	0.000	0.000	0.000	0.000	0.000	0.000
4	1.111	0.267	0.165	0.127	0.072	0.000	0.281	0.000	0.000	0.000	0.000	0.000	0.520
5	1.108	0.265	0.147	0.123	0.070	0.000	0.279	0.000	0.000	0.000	0.006	0.000	0.521
6	1.022	0.230	0.056	0.087	0.050	0.000	0.224	0.000	0.174	0.000	0.030	0.000	0.435
7	1.022	0.230	0.056	0.087	0.050	0.000	0.224	0.000	0.174	0.000	0.030	0.000	0.435
8	0.973	0.220	0.037	0.074	0.044	0.000	0.199	0.026	0.198	0.000	0.033	0.028	0.398
9	0.950	0.216	0.031	0.068	0.041	0.015	0.188	0.034	0.202	0.000	0.034	0.034	0.383
10	0.803	0.205	0.000	0.037	0.027	0.095	0.126	0.077	0.219	0.030	0.036	0.061	0.292

Then the algorithm consecutively adds assets 1,4,10,8,6,9,5,3 and 7 to the free assets, by lowering the λ, until we have the minimum variance portfolio, which is turning point 10. We see that turning point 4 is a relatively good turning point, since it has a much higher return than our minimum variance portfolio, while only having a little more risk. On the other hand, its return is not that much lower with respects to turning point 3, but the risk is considerably lower.

4 Discussion

We have seen Markowitz' Critical Line Algorithm at work for the most common problem in portfolio optimization. However, we can see that this version of the algorithm does not work when we have more inequality constraints, for we can not easily change the problem so we get rid of these constraints. A way of solving such a problem is with the help of the simplex method. Philip Wolfe published a paper describing this simplex method for quadratic programming in [8]. Since this method can be used for more cases, it can be seen as a better way to solve the portfolio optimization problem. However, since the inequality constraints on the linear combinations of the weights does not appear much in real life, the method of Markowitz' is chosen above the method of Wolfe.

While working on this thesis, I got the idea to rewrite the code for the Critical Line Algorithm, as provided in Python in [1], in C++. My main motivation for this was to give myself the challenge of learning a new programming language, as well as providing an open-source implementation for people who are not familiar with Python but are familiar with C++. However during the process of rewriting the code, I concluded that my knowledge of C++ was not big enough to provide a working program for the algorithm. I am not familiar enough with the structures in C++ to give the same methods and outcomes as Python. The fact that I needed to understand the algorithm in Python, learn new theory about both portfolio theory and optimization theory and learn a new programming language, turned out to be a too big of a challenge for the given time. Therefore I hope that, with the explanation I have given about the algorithm in Python, someone who has more experience in programming in C++ can use this thesis to actually write a working program. Although I am disappointed about the fact that I could not provide a working program, I am satisfied with what I have learned and what I have provided.

5 Conclusion

In this thesis we have seen the working of the Critical Line Algorithm of Harry Markowitz. By giving a foundation on some theory needed to understand the mathematical problem of portfolio optimization, as well as the beginnings of portfolio theory, this thesis should provide enough information for one to understand how to solve the problem of finding the portfolio with the highest return, or the minimal variance of return, subject to an expected excess return. We have also given an detailed description of the working of the CLA, both the code and the algorithm. With the help of this thesis, one now should be able to write the algorithm as provided in Python in other languages, such as C++ or Fortan.

The model provided by Markowitz is not the most optimal model available for solving portfolio optimization problems. We have seen that the inequality constraints on the weights are dismissed by working with free and bounded assets. However, when there are more inequality conditions, our model can not find solutions which satisfy this inequalities. In these cases, some other solution method should be used, such as the simplex algorithm created by Wolfe. That being said, Markowitz' model is commonly the more used method. The problems faced are often of the form as described in this thesis. Furthermore, the CLA gives the complete solution space, the whole efficient portfolio, which means that one can make a well-considered choice in selecting his portfolio. On top of that, the model is relatively easy and has a higher performance than for example the standard Matlab optimization tool. Since the difficulty of programming in Python is also lower than other languages such as C++, we can conclude that the algorithm described in this thesis is preferable over most more complex methods and gives financial practitioners a good foundation in their search for optimal portfolios.

A CLA in Python code[1]

This is the code of the class for the Critical Line Algorithm, as described in this thesis.

```
#!/usr/bin/env python
# On 20130210, v0.2
# Critical Line Algorithm
# by MLdP <lopezdeprado@lbl.gov>
import numpy as np
#---------------------------------------------------------------
#---------------------------------------------------------------
class CLA:
    def __init__(self,mean,covar,lB,uB):
        # Initialize the class
        self.mean=mean
        self.covar=covar
        self.lB=lB
        self.uB=uB
        self.w=[] # solution
        self.l=[] # lambdas
        self.g=[] # gammas
        self.f=[] # free weights
#---------------------------------------------------------------
    def solve(self):
        # Compute the turning points,free sets and weights
        f,w=self.initAlgo()
        self.w.append(np.copy(w)) # store solution
        self.l.append(None)
        self.g.append(None)
        self.f.append(f[:])
        while True:
            #1) case a): Bound one free weight
            l_in=None
            if len(f)>1:
                covarF,covarFB,meanF,wB=self.getMatrices(f)
                covarF_inv=np.linalg.inv(covarF)
                j=0
                for i in f:
                    l,bi=self.computeLambda(covarF_inv,covarFB,meanF,wB,j,\
                    [self.lB[i],self.uB[i]])
                    if l>l_in:l_in,i_in,bi_in=l,i,bi
                    j+=1
            #2) case b): Free one bounded weight
            l_out=None
            if len(f)<self.mean.shape[0]:
                b=self.getB(f)
                for i in b:
                    covarF,covarFB,meanF,wB=self.getMatrices(f+[i])
                    covarF_inv=np.linalg.inv(covarF)
                    l,bi=self.computeLambda(covarF_inv,covarFB,meanF,wB,meanF.shape[0]-1, \
                        self.w[-1][i])
                    if (self.l[-1]==None or l<self.l[-1]) and l>l_out:l_out,i_out=l,i
            if (l_in==None or l_in<0) and (l_out==None or l_out<0):
                #3) compute minimum variance solution
                self.l.append(0)
                covarF,covarFB,meanF,wB=self.getMatrices(f)
                covarF_inv=np.linalg.inv(covarF)
                meanF=np.zeros(meanF.shape)
            else:
                #4) decide lambda
                if l_in>l_out:
                    self.l.append(l_in)
                    f.remove(i_in)
                    w[i_in]=bi_in # set value at the correct boundary
                else:
                    self.l.append(l_out)
                    f.append(i_out)
                covarF,covarFB,meanF,wB=self.getMatrices(f)
                covarF_inv=np.linalg.inv(covarF)
            #5) compute solution vector
            wF,g=self.computeW(covarF_inv,covarFB,meanF,wB)
            for i in range(len(f)):w[f[i]]=wF[i]
            self.w.append(np.copy(w)) # store solution
            self.g.append(g)
            self.f.append(f[:])
            if self.l[-1]==0:break
        #6) Purge turning points
        self.purgeNumErr(10e-10)
        self.purgeExcess()
#---------------------------------------------------------------
    def initAlgo(self):
        # Initialize the algo
        #1) Form structured array
        a=np.zeros((self.mean.shape[0]),dtype=[('id',int),('mu',float)])
        b=[self.mean[i][0] for i in range(self.mean.shape[0])] # dump array into list
        a[:]=zip(range(self.mean.shape[0]),b) # fill structured array
        #2) Sort structured array
        b=np.sort(a,order='mu')
        #3) First free weight
        i,w=b.shape[0],np.copy(self.lB)
        while sum(w)<1:
            i-=1
            w[b[i][0]]=self.uB[b[i][0]]
        w[b[i][0]]+=1-sum(w)
        return [b[i][0]],w
#---------------------------------------------------------------
    def computeBi(self,c,bi):
        if c>0:
            bi=bi[1][0]
        if c<0:
            bi=bi[0][0]
        return bi
#---------------------------------------------------------------
    def computeW(self,covarF_inv,covarFB,meanF,wB):
        #1) compute gamma
        onesF=np.ones(meanF.shape)
        g1=np.dot(np.dot(onesF.T,covarF_inv),meanF)
        g2=np.dot(np.dot(onesF.T,covarF_inv),onesF)
        if wB==None:
            g,w1=float(-self.l[-1]*g1/g2+1/g2),0
        else:
            onesB=np.ones(wB.shape)
            g3=np.dot(onesB.T,wB)
            g4=np.dot(covarF_inv,covarFB)
            w1=np.dot(g4,wB)
            g4=np.dot(onesF.T,w1)
            g=float(-self.l[-1]*g1/g2+(1-g3+g4)/g2)
        #2) compute weights
        w2=np.dot(covarF_inv,onesF)
        w3=np.dot(covarF_inv,meanF)
        return -w1+g*w2+self.l[-1]*w3,g
#---------------------------------------------------------------
    def computeLambda(self,covarF_inv,covarFB,meanF,wB,i,bi):
        #1) C
        onesF=np.ones(meanF.shape)
        c1=np.dot(np.dot(onesF.T,covarF_inv),onesF)
        c2=np.dot(covarF_inv,meanF)
        c3=np.dot(np.dot(onesF.T,covarF_inv),meanF)
        c4=np.dot(covarF_inv,onesF)
        c=-c1*c2[i]+c3*c4[i]
        if c==0:return None,None
        #2) bi
        if type(bi)==list:bi=self.computeBi(c,bi)
        #3) Lambda
        if wB==None:
            # All free assets
            return float((c4[i]-c1*bi)/c),bi
        else:
            onesB=np.ones(wB.shape)
            l1=np.dot(onesB.T,wB)
            l2=np.dot(covarF_inv,covarFB)
            l3=np.dot(l2,wB)
            l2=np.dot(onesF.T,l3)
            return float(((1-l1+l2)*c4[i]-c1*(bi+l3[i]))/c),bi
#---------------------------------------------------------------
    def getMatrices(self,f):
        # Slice covarF,covarFB,covarB,meanF,meanB,wF,wB
        covarF=self.reduceMatrix(self.covar,f,f)
        meanF=self.reduceMatrix(self.mean,f,[0])
        b=self.getB(f)
        covarFB=self.reduceMatrix(self.covar,f,b)
        wB=self.reduceMatrix(self.w[-1],b,[0])
        return covarF,covarFB,meanF,wB
#---------------------------------------------------------------
    def getB(self,f):
        return self.diffLists(range(self.mean.shape[0]),f)
#---------------------------------------------------------------
    def diffLists(self,list1,list2):
        return list(set(list1)-set(list2))
#---------------------------------------------------------------
    def reduceMatrix(self,matrix,listX,listY):
        # Reduce a matrix to the provided list of rows and columns
        if len(listX)==0 or len(listY)==0:return
        matrix_=matrix[:,listY[0]:listY[0]+1]
        for i in listY[1:]:
            a=matrix[:,i:i+1]
            matrix_=np.append(matrix_,a,1)
        matrix__=matrix_[listX[0]:listX[0]+1,:]
        for i in listX[1:]:
            a=matrix_[i:i+1,:]
            matrix__=np.append(matrix__,a,0)
        return matrix__
#---------------------------------------------------------------
def getMinVar(self):
        # Get the minimum variance solution
        var=[]
        for w in self.w:
            a=np.dot(np.dot(w.T,self.covar),w)
            var.append(a)
        return min(var)**.5,self.w[var.index(min(var))]
```

These functions were provided by [1], but are not covered in this thesis. One can consult [1] if he wants to know more about these functions.

```
#---------------------------------------------------------------
    def purgeNumErr(self,tol):
        # Purge violations of inequality constraints (associated with ill-conditioned covar
        i=0
        while True:
            flag=False
            if i==len(self.w):break
            if abs(sum(self.w[i])-1)>tol:
                flag=True
            else:
                for j in range(self.w[i].shape[0]):
                    if self.w[i][j]-self.lB[j]<-tol or self.w[i][j]-self.uB[j]>tol:
                        flag=True;break
            if flag==True:
                del self.w[i]
                del self.l[i]
                del self.g[i]
                del self.f[i]
            else:
                i+=1
        return
#---------------------------------------------------------------
def purgeExcess(self):
        # Remove violations of the convex hull
        i,repeat=0,False
        while True:
            if repeat==False:i+=1
            if i==len(self.w)-1:break
            w=self.w[i]
            mu=np.dot(w.T,self.mean)[0,0]
            j,repeat=i+1,False
            while True:
                if j==len(self.w):break
                w=self.w[j]
                mu_=np.dot(w.T,self.mean)[0,0]
                if mu<mu_:
                    del self.w[i]
                    del self.l[i]
                    del self.g[i]
                    del self.f[i]
                    repeat=True
                    break
                else:
                    j+=1
        return
#---------------------------------------------------------------
    def getMaxSR(self):
        # Get the max Sharpe ratio portfolio
        #1) Compute the local max SR portfolio between any two neighbor turning points
        w_sr,sr=[],[]
        for i in range(len(self.w)-1):
            w0=np.copy(self.w[i])
            w1=np.copy(self.w[i+1])
            kargs={'minimum':False,'args':(w0,w1)}
            a,b=self.goldenSection(self.evalSR,0,1,**kargs)
            w_sr.append(a*w0+(1-a)*w1)
            sr.append(b)
        return max(sr),w_sr[sr.index(max(sr))]
#---------------------------------------------------------------
    def evalSR(self,a,w0,w1):
        # Evaluate SR of the portfolio within the convex combination
        w=a*w0+(1-a)*w1
        b=np.dot(w.T,self.mean)[0,0]
        c=np.dot(np.dot(w.T,self.covar),w)[0,0]**.5
        return b/c
#---------------------------------------------------------------
    def goldenSection(self,obj,a,b,**kargs):
        # Golden section method. Maximum if kargs['minimum']==False is passed
        from math import log,ceil
        tol,sign,args=1.0e-9,1,None
        if 'minimum' in kargs and kargs['minimum']==False:sign=-1
        if 'args' in kargs:args=kargs['args']
        numIter=int(ceil(-2.078087*log(tol/abs(b-a))))
        r=0.618033989
        c=1.0-r
        # Initialize
        x1=r*a+c*b;x2=c*a+r*b
        f1=sign*obj(x1,*args);f2=sign*obj(x2,*args)
        # Loop
        for i in range(numIter):
            if f1>f2:
                a=x1
                x1=x2;f1=f2
                x2=c*a+r*b;f2=sign*obj(x2,*args)
            else:
                b=x2
                x2=x1;f2=f1
                x1=r*a+c*b;f1=sign*obj(x1,*args)
        if f1<f2:return x1,sign*f1
        else:return x2,sign*f2
```

```
#---------------------------------------------------------------
    def efFrontier(self,points):
        # Get the efficient frontier
        mu,sigma,weights=[],[],[]
        a=np.linspace(0,1,points/len(self.w))[:-1] # remove the 1, to avoid duplications
        b=range(len(self.w)-1)
        for i in b:
            w0,w1=self.w[i],self.w[i+1]
            if i==b[-1]:a=np.linspace(0,1,points/len(self.w)) # include the 1 in the last it
            for j in a:
                w=w1*j+(1-j)*w0
                weights.append(np.copy(w))
                mu.append(np.dot(w.T,self.mean)[0,0])
                sigma.append(np.dot(np.dot(w.T,self.covar),w)[0,0]**.5)
        return mu,sigma,weights
#---------------------------------------------------------------
#---------------------------------------------------------------
```

This part of code provides some output for the CLA class, such as the results in our numerical example. However, the details are not discussed in this thesis.

```
#---------------------------------------------------------------
def plot2D(x,y,xLabel='',yLabel='',title='',pathChart=None):
    import matplotlib.pyplot as mpl
    fig=mpl.figure()
    ax=fig.add_subplot(1,1,1) #one row, one column, first plot
    ax.plot(x,y,color='blue')
    ax.set_xlabel(xLabel)
    ax.set_ylabel(yLabel,rotation=90)
    mpl.xticks(rotation='vertical')
    mpl.title(title)
    if pathChart==None:
        mpl.show()
    else:
        mpl.savefig(pathChart)
    mpl.clf() # reset pylab
    return
#---------------------------------------------------------------
def main():
    import numpy as np
    import CLA
    #1) Path
    path='/CLA/CLA_DATA.csv'
    #2) Load data, set seed
    headers=open(path,'r').readline()[:-1].split(',')
    data=np.genfromtxt(path,delimiter=',',skip_header=1) # load as numpy array
    mean=np.array(data[:1]).T
    lB=np.array(data[1:2]).T
    uB=np.array(data[2:3]).T
    covar=np.array(data[3:])
    #3) Invoke object
    cla=CLA.CLA(mean,covar,lB,uB)
    cla.solve()
    print (cla.w) # print all turning points
    #4) Plot frontier
    mu,sigma,weights=cla.efFrontier(100)
    plot2D(sigma,mu,'Risk','Expected Excess Return','CLA-derived Efficient Frontier')
    #5) Get Maximum Sharpe ratio portfolio
    sr,w_sr=cla.getMaxSR()
    print (np.dot(np.dot(w_sr.T,cla.covar),w_sr)[0,0]**.5,sr)
    print (w_sr)
    #6) Get Minimum Variance portfolio
    mv,w_mv=cla.getMinVar()
    print (mv)
    print (w_mv)
    return
#----------------------------------------------------------------
# Boilerplate
if __name__=='__main__':main()
```

References

[1] Bailey, D. H., and López de Prado, M. An open-source implementation of the critical-line algorithm for portfolio optimization. *Algorithms 6*, 1 (2013), 169–196.

[2] Markowitz, H. Portfolio selection. *The journal of finance 7*, 1 (1952), 77–91.

[3] Markowitz, H. The optimization of a quadratic function subject to linear constraints. *Naval research logistics Quarterly 3*, 1-2 (1956), 111–133.

[4] Meucci, A. *Risk and Asset Allocation.* Springer Finance Textbooks. Springer, 2009.

[5] Niedermayer, A., and Niedermayer, D. Applying markowitz's critical line algorithm. In *Handbook of portfolio construction.* Springer, 2010, pp. 383–400.

[6] Pitman, J. *Probability.* Springer Texts in Statistics. Springer New York, 1999.

[7] Rice, J. *Mathematical Statistics and Data Analysis.* Cengage Learning, 2006.

[8] Wolfe, P. The simplex method for quadratic programming. *Econometrica: Journal of the Econometric Society* (1959), 382–398.