Utility-based Shortfall Risk: Implementation using stochastic, deterministic root-finding algorithms and Fourier Transform

Charis Chalvatzis

December 16, 2013

Master Thesis for the Interdepartmental Postgraduate Course Programme
Mathematical Modelling in Modern Technologies and Financial Engineering

Faculty of Applied Mathematical and Physical Sciences
National Technical University of Athens

Utility-based Shortfall Risk: Implementation using stochastic, deterministic root-finding algorithms and Fourier Transform

Charis Chalvatzis

Master Thesis for the Interdepartmental Postgraduate Course Programme
Mathematical Modelling in Modern Technologies and Financial Engineering

It has been approved by the advisory committee

....	....	....
Prof. Dr. Antonis Papapadoleon	Prof. Dr. Ioannis Spiliotis	Prof. Dr. Michail Loulakis

Faculty of Applied Mathematical and Physical Sciences
National Technical University of Athens
December, 2013

Contents

Abstract

In this thesis, we begin by reviewing the characteristics of different risk measures and we are drawn particularly for convex risk measures. One of these is the utility-based shortfall risk whose value we will try to determine by employing deterministic and stochastic root-finding algorithms, which will give us the opportunity to study the latter algorithms as well. Then we will compare them in order to see which one is better in terms of speed and convergence. On the implementation side, we will take a step further by using parallel programming for the stochastic case. This feature proves to be quite interesting in terms of speed. Then we will compare and contrast the Fourier transform with the Monte Carlo simulation, regarding the computation of the expected value, which is embedded into the root-finding problem. Our ambition is to test whether stochastic scheme is faster than deterministic ones and whether direct computation of the integral (i.e. Fourier transform) is indeed faster than simulation (i.e. Monte Carlo).

Acknowledgements

I would like to thank first and foremost my supervisor Prof. Dr. Antonis Papapadoleon whose guidance and knowledge where of utmost importance and without him, the present thesis would not have concluded. Collaborating with him has broaden my horizons and way of thinking in mathematics. He was always available, despite his huge workload and readily explained all I needed it to know in order to remain on the right track. I would also like to thank Prof. Dr. Ioannis Spiliotis, because it was his course "S.D.E.s and applications to finance" that introduced me to the fascinating world of Stochastic Calculus. In addition, I would like to thank Prof. Dr. Michail Loulakis for his constructive feedback during the review period.
Next, I would also like to thank my former boss Patrick Maccury and colleagues, while working at European Central Bank, whose understanding and support was of equal importance. Last but definitely not least, I would like to express my deepest gratitude to my family and my beloved Ioanna, whose constant support and patience equipped me with the psychological boost and strength needed to accomplish my thesis.

1 Introduction

In recent years, when the crisis evolved and nowadays, the notion and idea of risk became rather important. The focus on handling, measuring and maintaining risk was one of the first priorities among public and private shareholders. But what exactly do we mean when we mention risk? In our context, we mean the financial risk that is the risk associated with financing, including financial transactions. Then risk could be defined, broadly, as any event or action that may adversely affect an organization's ability to achieve its objectives and execute its strategies or, alternatively, the quantifiable likelihood of loss or less-than-expected returns. We bear in mind that because risk management in a very large field and subject we cannot define it properly and address all contexts.[1]
Having the notion of risk defined, we go a step further and look its different aspects. Depending on the activity, there are different kinds of risk. Since we are talking about financial transactions it is natural to assume that every transaction carries on a risk. For example, when we are dealing with loans (either to companies, corporations, households or governments), the risk of a borrower not being able to repay its debt, which in financial terms is defined as going default, is known as credit risk. In banking sector, another term of risk arises more than often: market risk. This is the risk of a change in the value of a financial position due to changes in the value of the underlying components on which that position depends, such as stock and bond prices, exchange rates, commodity prices, etc[1]. Obviously we could break down the components of market risk. Since we are talking about stock and bonds it is evident that equity or interest rate risk is apparent. The former refers that stocks prices will change and the latter that the interest rate of the bonds will change. Another important risk regarding the liquidity of the assets is, of course, liquidity risk. A definition given by the Federal Reserve Bank of San Francisco [22] defines it as the risk *that a firm will not be able to meet its current and future cash flow and collateral needs, both expected and unexpected, without materially affecting its daily operations or overall financial condition.* The list of risks does not stop here: refinancing, operational, legal, political, reputational, volatility, settlement, profit or systemic risk, just to name the most important ones. In order to address these risks a specific regulatory framework was developed. Particularly known as International regulatory framework for banks, with the latest framework being known as Basel III. The framework is a comprehensive set of reform measures, developed by the Basel Committee on Banking Supervision, to strengthen the regulation, supervision and risk management of the banking sector. These measures aim to improve the banking sector's ability to absorb shocks

arising from financial and economic stress, whatever the source, improve risk management and governance and finally strengthen banks' transparency and disclosures [2].

From the above if it has not been apparent by now, there is a constant effort to define, handle and measure risk. For the third part, several measures have been introduced: Value at Risk (V@R), Expected Shortfall Risk (ES) or utility-based shortfall risk. Our thesis falls exactly under this scope. We briefly present these measure and we implement the utility-based shortfall risk. The implementation is where all the interesting things appear. We employ stochastic and deterministic root-finding algorithms and try to compare and contract its differences, if any. How we end up with a root-finding problem is very interesting and it will be presented. In addition to these, we also use the Fourier Transform, where we expect a significant gain, in terms of computation time.

The aforementioned points stand from a mathematical point of view. From a computer science perspective, we are also interested on how to make faster, better and efficient algorithms. We will handle with the first: speed. Here lies the advantage of using stochastic root-finding algorithms instead of the deterministic ones. All the above finally lead on using parallel programming. Apart from the financial crisis era, we are living in the computing era, where we now have in our hands computing devices with large amount of memory able to carry on difficult calculations quite fast. The benefit of using parallel computing is obviously speed. We will see that we gain quite in terms of speed while achieving the same level accuracy as before.

The goal of the thesis is to investigate and conclude whether the use of utility-based shortfall risk by using stochastic root-finding algorithms and by employing parallel programming, achieves the same level of accuracy but much faster than using the industry standards risk measures, such as V@R. Having the same information but quicker, yields significant advantage for financial institutions which leads to better risk handling.

2 Risk Measures

One critical aspect when discussing about risk is the notion of measurement of risk. That is the need to quantify risk in many different situations. For example, an investment bank, or a financial institution in general, would like to have an amount of capital that would be used as a buffer in case unexpected losses would occur. Obviously, this amount is a number and is specified by a model that is used for measuring the risk, which in this case is related to the event of unexpected losses. Based on that number, decisions could be taken. [1, p41, Ch2] In order to arrive to such conclusions we need to set the theoretical framework, from which one could then deduce the implementation according to each needs. We therefore introduce the notion of Risk Measures. This idea will be the vehicle that would drive us through the general framework and would later on provide us with an easy way of implementation.

2.1 Definitions and Properties

Let us define a few things in order to proceed to the formal definition. First of all we have a set of scenarios, denoted by Ω. A *financial position* is a mapping from $X : \Omega \longrightarrow \mathbb{R}$, where $X(\omega)$ is the discounted net worth of the position at the end of the trading period, if the scenario $\omega \in \Omega$ is realized. Our goal is again to try and specify the risk X by some number $\rho(X)$, where X belongs to a given class $\mathcal{X}$ of financial positions. This will be our trivial assumptions. To sum up,

- Ω :fixed set of scenarios
- X: financial position, where $X(\omega)$ is the discounted net worth of the position at the end of the trading period if the scenario ω is realized
- $\rho(X)$: number, specifying the risk of the financial positions
- $\mathcal{X}$: Class of financial positions

Now we are now in place for our first definitions which is that of *monetary measure of risk*:

Definition 1. *A mapping ρ: $\mathcal{X} \longrightarrow \mathbb{R}$ is called a* ***monetary measure of risk*** *if it satisfies the following conditions for all $X, Y \in \mathcal{X}$:*

- *Monotonicity : if $X \leq Y$, then $\rho(X) \geq \rho(Y)$*
- *Translation Invariance: if $m \in \mathbb{R}$, then $\rho(X + m) = \rho(X) - m$*

[9, p168]

The interpretation of monotonicity is that when the payoff is increased then the associated risk is reduced. Translation invariance, emphasizes the fact that $\rho(X)$ could also be viewed as a capital requirement (remember our earlier example of the investment bank), that is $\rho(X)$ is the amount of capital to be added to the position X in order to make it acceptable from a point of a supervising agency (the investment bank holds a buffer because an agency has also requested to do so). Thus if amount m is added to the position, the risk is reduced by the same amount. This idea is used with the famous standard practice tool, Value-at-Risk (V@R), which will encounter later.

Apart from these two basic properties, we could enhance our mapping with other as well. To this end we provide two more definitions of risk measures, which will be used throughout our framework, and their differences will be our main point of discussion (we will later see that V@R and utility-based shortfall risk fall each on one of this categories).

Definition 2. *A monetary measure of risk $\rho : \mathcal{X} \longrightarrow \mathbb{R}$ is called* ***convex measure of risk*** *if it satisfies the following:*

- *Convexity :* $\rho(\lambda X + (1-\lambda)Y) \leq \lambda\ \rho(X) + (1\text{-}\ \lambda\ )Y$, *for all* $0 \leq \lambda \leq 1$.

If it satisfies the *positive homogeneity* then it is called coherent measure of risk. Therefore,

Definition 3. *A convex measure of risk is called* **coherent** *measure of risk if it satisfies the following:*

- *Positive Homogeneity: if* $\lambda \geq 0$, *then* $\rho(\lambda\ X) = \lambda\ \rho(X)$.

The property of convexity is used to explain and match the economic definition of diversification. Consider an investor who can invest a fraction of his wealth first to an option, leading to a position of X and the remaining of his wealth (1-λ) to the second option, leading to a position Y. At the end the investor will have $\lambda\ X$ +(1-λ)Y. In the finance context, this means that the investor has lower risk because he has diversified the systematic risk of his/her investments. So, convexity gives a precise meaning to the idea, that diversification should not increase the risk. (In the portfolio management context this is proved by the use of variances and correlations. The more uncorrelated our investments are, the lower the risk. Therefore by diversifying our portfolio, we decrease our (systematic) risk).

The positive homogeneity property means that the measure of risk is normalized, for example $\rho(0) = 0$. Under this condition the convexity is equivalent with sub-additivity condition:

- Sub-additivity : $\rho\ (X + Y) \leq \rho(X) + \rho(Y)$

Through sub-additivity we could aggregate all risk together and be sure that the overall risk will be lower than having each risk separately or in the worst case equal (in economic terms is similar with the diversification benefit). To sum up, with have the following definitions:

Lemma 1. *A mapping $\rho : \mathcal{X} \longrightarrow \mathbb{R}$ is called:*

1. *Monetary measure of risk, if it satisfies*
   a. *Monotonicity: if $X \leq Y$, then $\rho(X) \geq \rho(Y)$*
   b. *Translation Invariance: if $m \in \mathbb{R}$, then $\rho(X + m) = \rho(X) - m$*
2. *Convex measure of risk, if it satisfies*
   a. *Monetary measure of risk conditions*
   b. *Convexity : $\rho(\lambda X + (1\text{-}\lambda)Y) \leq \lambda \rho(X) + (1\text{-}\lambda)Y$, for all $0 \leq \lambda \leq 1$.*
3. *Coherent measure of risk, if it satisfies*
   a. *Convex measure of risks conditions*
   b. *Positive Homogeneity: if $\lambda \geq 0$, then $\rho(\lambda X) \geq \lambda$*

[9, p169]

Proof 1. *We will prove the following: Positive Homogeneity and Sub-Additivity lead us to Convexity.*
We set $M = \lambda X$ and $K = (1 - \lambda)Y$ for $0 \leq \lambda \leq 1$. We then have,

$$\rho(M + K) \leq \rho(M) + \rho(K) \leq \rho(\lambda X) + \rho((1 - \lambda)Y) \overset{[1]}{\Rightarrow} \tag{1}$$

$$\rho(M + K) \leq \lambda\rho(X) + (1 - \lambda)\rho(Y), \overset{[2]}{\Rightarrow} \tag{2}$$

$$\rho(M + K) = \rho(\lambda(X) + (1 - \lambda)Y) \leq \lambda\rho(X) + (1 - \lambda)\rho(Y), \tag{3}$$

where the last inequality is the convexity property, (1) follows from sub-additivity property and (2) follows from positive homogeneity.

Based on the above definitions several risk measures have been introduced, each with different properties and different usage, such as Value at risk, average value at risk, utility-based shortfall risk. We present them, in the next subsection.

2.2 Value at Risk

In order to introduce the definition of value at risk, we have to first specify what is a quantile distribution, or a λ-quantile. Then specifying the risk of a financial position X will consist in determining a quantile of the distribution of X under a given probability measure $\mathcal{P}$.

Definition 4. *Like before we assume a probability space $(\Omega, \mathcal{F}, \mathcal{P})$. A λ-quantile of a random variable X is any real number q with the property:*

- *$\mathcal{P}[X \leq q] \geq \lambda$ and $\mathcal{P}[X < q] \leq \lambda$*

Definition 5. *We fix some level $\lambda \in (0,1)$. For a financial position X, we define the Value at Risk as follows:*

- $V@R_\lambda(X)$:= inf $\{ c \in \mathbb{R}, \mathcal{P}[X + c < 0] \leq \lambda \}$

In financial context, $V@R_\lambda(X)$ is the smallest amount of capital which, if added to the position X and invested in the risk-free asset, keeps the probability of a loss under a certain level λ. Typical values for λ are 0.05 and 0.01. Stated differently, V@R could be defined as the loss level that will not be exceeded with a certain confidence level during a certain period of time. For example, if a bank's 10-day 99% V@R is 3 million USD, there is considered to be only a 1% chance that losses will exceed 3 million USD in 10 days.
However, as simple as it may be to calculate value at risk, it has two drawbacks. First of all, it fails to capture the diversification benefit, so it may penalize diversification instead of encouraging it. From a mathematical point of view, this is explained due to the fact that V@R is not a convex risk measure (no sub-additivity). Second, it does not take into account the amount of the loss, if one occurs, just the probability. This will be better explained by the following example as taken from Dunkel and Weber's paper [13].
Suppose we have two portfolios modeled by random variables X_1, X_2, where

$$X_1 = \begin{cases} +1 & \text{with probability } 99\% \\ -1 & \text{with probability } 1\% \end{cases}$$

and

$$X_2 = \begin{cases} +1 & \text{with probability } 99\% \\ -10^{10} & \text{with probability } 1\% \end{cases}$$

A value $X_i \geq 0$ corresponds to the event 'no loss', whereas X_i <0 means 'loss', $i = 1, 2$. Setting λ=0.01 we find that

$$\text{V@R}(X_1) = \text{V@R}(X_2) = -1 \leq 0$$

Therefore according to Value at Risk both portfolios are equally acceptable (a position is acceptable if its risk measure is negative). But as we see there is a huge difference in terms of the amount of loss. For the first portfolio this loss is only -1 whereas for the second portfolio is -10^{10}. Clearly the first portfolio is more preferable. Of course this example is trivial and is easy to locate which portfolio to choose but in real life this distinction becomes harder. So, risk allocation on V@R may lead to the portfolio with the smallest loss probability, even if the loss associated with it is extremely large. More generally stated the following problem holds for V@R: once we know the seperate V@R's of different branches (variables), how do I estimate the global risk if V@R is not subadditive (sum of the variables)? The next measures try to address this inefficiency [14]

2.3 Expected Shortfall Risk (ES)

We begin by providing the definition of the Expected Shortfall risk.

Definition 6. *If $X \epsilon \mathcal{X}$ is the payoff of a portfolio at some future time and $0 < \alpha < 1$, we then define the expected shortfall as:*

$$ES_\alpha = \frac{1}{\alpha} \int_0^\alpha V@R_\gamma(X) d\gamma \tag{4}$$

where $V@R_\gamma$ is the Value at Risk defined in the previous section.

Apart from expected shortfall, sometimes is also called as average value at risk (AV@R) or conditional value at risk (CV@R). Despite the debate over which term is more appropriate what expected shortfall does, is to average over the worst case scenarios (which also justifies the use of the integral in the definition). So the "expected shortfall at q% level" is the expected return on the portfolio in the worst % of the cases. Alternatively, we could say that ES is the expected loss of portfolio value given that a loss is occurring at or below the q-quantile.

Expected Shortfall evaluates the value (or risk) of an investment in a conservative way, focusing on the less profitable outcomes. For high values of q it ignores the most profitable but unlikely possibilities, for small values of q it focuses on the worst losses. On the other hand, even for lower values of q, expected shortfall does not consider only the single most catastrophic outcome. A value of q often used in practice is 5%.

The motivation behind Expected Shortfall was the concern to better handle random variables with discontinuous distributions. Examples include not-traded loans portfolio (purely discrete distributions) or portfolios containing derivatives (mixture of continuous and discrete distributions). Risk measures like V@R, have problems because of their sensitivity in the confidence interval α. On the contrary, the value of the ES will not change a lot, since it is not affected much by a small change in the confidence interval [14]. Moreover, the Expected shortfall risk is aware of the shape of the conditional distribution of p% worst events while V@R is not aware at all.

The big difference between ES and V@R, is of course that that the former measure is coherent and the latter is not, although ES is derived or constructed from V@R.

Probability of Event	Ending value Portfolio	Profit	q	Expected Shortfall
10%	0	-100	10%	-100
30%	80	-20	20%	-60
40%	100	0	40%	-40
20%	150	50	100%	-6

From this table let us calculate the expected shortfall for a few values of q:
To see how these values were calculated, consider the calculation of ES for $q = 10\%$, the expectation in the worst 10% of cases. These cases belong to (are a subset of) row 1 in the profit table, which have a profit of -100 (total loss of the 100 invested). The expected profit for these cases is 100.
Now consider the calculation of ES for $q = 20\%$, the expectation in the worst 20 out of 100 cases. These cases are as follows: 10 cases from row one, and 10 cases from row two (note that $10 + 10$ equals the desired 20 cases). For row 1 there is a profit of 100, while for row 2 a profit of 20. Using the expected value formula we get:

$$\frac{\frac{10}{100} * (-100) + \frac{10}{100} * (-20)}{\frac{20}{100}} = -60$$

Similarly we calculate for any value of q. We select as many rows starting from the top as are necessary to give a cumulative probability of q and then calculate an expectation over those cases. In general the last row selected may not be fully used (for example in calculating ES for $q = 20\%$ we used only 10 of the 30 cases per 100 provided by row 2).
From the above example, it is evident the differences between V@R and Expected Shortfall. The latter does not only "cut off" at the worst point but rather tries to find out, past that point, how worse could things go (by averaging the values).

2.4 Utility-based Shortfall Risk

It is time to present the class of utility-based shortfall risk measures in the spirit of [9] and [13]. We remind that we have a probability space $(\Omega,\ \mathcal{F}, \mathcal{P})$ already in place from chapter 2. Let l

denote a convex loss function and λ be a point in the interior of the range of l. We define the acceptance set by

$$\mathcal{A} := \{ \ X \in L^{\infty} : E[\ l(\text{-}X)] \leq \lambda \ \}$$

And the corresponding shortfall risk measure by

$$SR_{l,\lambda} := \inf \ \{ \ m \in \mathbb{R} : \ X + m \in A\}$$

As a reminder we provide the definition of a loss function:

Definition 7. *A function l: $\mathbb{R} \to \mathbb{R}$ is called loss function if it is* increasing *and* not identically constant *[9].*

The most important proposition is the following, which transforms the computation of a risk measure into a root-finding problem and justifies the use of stochastic and deterministic root-finding algorithms.

Proposition 1. *Let $SR_{l,\lambda}$ be the utility-based shortfall risk associated with the convex loss function l and threshold level λ. Suppose that $X \in L^{\infty}$. Then the following statements are equivalent:*

- $SR_{l,\lambda} \ = s^*$
- $\mathbb{E}[l(-X - s^*)] \ = \lambda$

Proof:
We follow a similar reasoning to [9]. We have:

$$SR_{l,\lambda} := \inf \ \{ \ m \in \mathbb{R} : \ X + m \in A\} = \inf\{m \in \mathbb{R} : \mathbb{E}[l(-X - m)] \leq \lambda\} = \rho(X).$$

We thus want to prove that $\rho(X) = m$ is the unique solution of $\mathbb{E}[l(-X - m)] = \lambda$. First note that, $\mathcal{A}$ is a convex set because it satisfies from construction the properties (a) and (b) from proposition 4.5 on page 160 of [9]. This implies that ρ is convex measure of risk.
Next, for a sequence $X_1 > X_2 > ... > X_n \in \mathcal{X}$, which converges to $X \in \mathcal{X}$ then by monotonicity $\rho(X_1) < \rho(X_2) < ... < \rho(X_n)$ so $X_n \searrow X$ implies $\rho(X_n) \nearrow \rho(X) = m$. If we combine the fact that $X \in L^{\infty}$ and that l is continuous then by applying dominated convergence we get that $\mathbb{E}[l(-X - \rho(X))] = \mathbb{E}[l(-X - m))] = \lambda$. Because l is strictly increasing to a neighbourhood very close to λ the solution is unique.

3 Root-finding Problems

3.1 Introduction

In the previous chapter we presented what risk measures are all about and introduced three of them. In this chapter we will introduce and explain the root-finding problems, which as we will witness later, play a significant role in the computation of risk measures.
There are many cases where someone would like to know the roof of a function. This information is quite valuable. For example, we could be looking for a maximum or minimum of a function which could have different real life interpretations: minimizing cost, minimizing friction, maximizing output produced. For well know functions this would be fairly easy not to mention quick procedure. Let's take for example the function f(x) = x^2 -x. In order to find its roots we equate f(x) with zero and therefore have the following:

$$\text{f(x)=0} \Rightarrow x^2 \text{ -x=0} \Rightarrow \text{x(x-1)=0} \Rightarrow$$
$$\text{Roots: } x_1\text{=0 and } x_2\text{=1}$$

But there are other times where the function is too complicated and solution cannot be given analytically. In this case, numerical methods should be employed and try to approximate the solution. Generally, we have a known function f: $\mathbb{R}^n \to \mathbb{R}^n$, a predefined target $\gamma \in \mathbb{R}^n$ and an unknown root $x^* \in \mathbb{R}^n$, whose value we are interested in finding. Therefore the problem is to determine the unique root of x = x^* when f reaches the target value γ. The target level γ is usually the desired level of a system's performance, which is obtained by manipulating the input vector x [17].
The algorithms have been divided into two major groups: the direct and the iterative. The difference between the two lies in time. That is, the direct algorithms complete and provide the solution in a finite number of steps where the iterative algorithms converge to the solution over time. They stop until a convergence criterion is met. Since these methods converge to the solution it becomes quite apparent that it is of utmost importance to know and track the convergence rate of the solution that is how quickly the algorithm converges to the solution. Obviously the fastest the better. [5,6]
The distinction between direct and iterative algorithms has given rise to two different classes of problems: deterministic root-finding problems and stochastic root-finding problems. An example of a problem deterministic in nature has already being shown, as simple as it may be. On the other hand a stochastic root-finding problem could be the following: We would like to minimize

the function $z(\theta) = \mathbf{E}[\theta X_1 + X_2,\ (1\text{-}\theta)X_3]$, where X_1, X_2, X_3 are Erlang(2) with means 1,2,3, and $0 < \theta < 1$ [p.253, 10]. It is often the case where the computation of the expected value will be difficult to compute with analytical methods and therefore simulation will be used. In our context, this is the case, the computation of expected values using either deterministic or stochastic root-finding algorithms.

3.1.1 The problem

Simply put, we are interested in estimating the value of a specific risk measure (utility-based shortfall risk) associated with a portfolio (first step). In order to achieve that, we transform the initial problem of estimating the value of a risk measure, to an equivalent problem, of that of finding the root of a function and hence the naming of root-finding problems (second step). This is how risk measures and RFPs are linked. Finally, what is left is to find ways to solve the RFP (third step).

As we will see later on, apart from the computation of the root of a function, we will also be interested to compute an expected value. Therefore, another issue of concern is what happens if we shift the focus from the root-finding algorithm to the computation of the expected value. That is, we are interested in comparing the Monte Carlo method with the Fourier Transform. This come later on the presentation and we will fully explain and define what we mean and what we are looking for.

In this chapter we will present the RFPs, the two major categories of algorithms, deterministic and stochastic, and finally we will present results concerning the convergence and the convergence rates of the problems.

3.1.2 Motivation

Our motivation lies at the different mathematical and computational properties of risk measures and how could these potentially affect the outcome. Firstly, V@R is not a coherent risk measure and utility-based shortfall risk is. As we have already demonstrated by the example in the section 2.2, this is a quite important feature. Secondly, we would like to compare and contrast the different algorithms that exist for solving RFPs problems and what useful information might come from that. Thirdly, we would like to see if the use of stochastic algorithms could give us a computational edge (parallel programming), when it comes to computing the value of the risk measure. Therefore, we will be involved in the above three steps of the problem, in some of them more, in others less. Last but not least, we are highly interested in observing whether the

Fourier transform will be quicker, as we expect, from the simulation method, as Monte Carlo is.

3.2 Deterministic Root Finding

We begin with the simpler of the two cases, that of the deterministic case. The generic case is the following:

We are interested in finding x: f(x) = 0 where f: $\mathbb{R}^n$-$\mathbb{R}^n$ *denotes a system of n non-linear equations and x is the n-dimensional root. The preceding example of f(x) =* x^2 *-x was a simple case with n=1 and rather simple function f.*

3.2.1 Methods

There have been proposed several methods in order to address the problem of finding the root [5]. We will briefly introduce them:

- Fixed point iteration is a very basic algorithm and it works only if the iteration function is convergent.

  Description: Given f(x)=0 we rewrite as x_{new}=g(x_{old})

  Algorithm. Fixed Point Iteration

  ```
  Initialize: $x_0$=..
  for $k = 1, 2, 3..$ do
      $X_k$= g($x_{k-1}$)
  end for
  if converged then
      stop
  end if
  ```

- Bisection:

  The bisection method is among the simplest and most robust algorithms for finding the root of a one-dimensional continuous function on a closed interval. Suppose that f(.) is a continuous function defined over an interval [a,b] and f(a) and f(b) have opposite signs. By the intermediate value theorem, there exists at least one m [a,b] such as f(m)=0. The key idea of the method is that it is iterative and each iteration starts by breaking the current interval thus bracketing the roots into two subintervals of equal length (hence the name bisection). The interval with the opposite signs on each endpoint becomes the new interval and the next iteration begins. Continuing this procedure with diminish our interval as much as possible and stops when a specific level of error tolerance has been reached. [5,6]

Description: Given an initial interval, break it into two subinterval. Find the one with opposite sings at each endpoints, make it the new interval and continue until it converged.

Algorithm. Bisection

```
Initialize: a =..., b =...
for $k = 1, 2, 3..$ do
  $x_m$= $\alpha$ + ($\beta$ - $\alpha$)/2
  if $sign(f(x_m)) = sign(f(x_\alpha))$ then
    $\alpha = x_m$
  else
    $\beta = x_m$
  end if
  if converged then
    stop
  end if
end for
```

Due to the simplicity of the methods we will not mention any convergence criteria. We point to the bibliography for further reading [6].

- Newton: The basic idea behind Newton's method is that for a given initial guess x_k, we use the f(x_k) and the derivative f'(x_k) to predict where f(x) crosses the x axis and therefore be the root of the function. Given an initial guess x_1 of the root we could approximate the function using Taylor expansion (the order depends on the problem) about x_1:

$$\text{f}(x_k + \Delta\text{x}) = \text{f}(x_k) + \Delta\text{x}\ \frac{df}{dx}\ |x_k + \frac{(\Delta x)^2}{2}\ \frac{d^2 f}{dx^2}\ |x_k + ...$$

By substituting Δx = $x_{k+1} - x_k$ and neglect the second term we arrive to

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k), where f'(x_k) = \frac{df}{dx}\ |x_k$$

Keeping in mind that our goal is to find x such that f(x) = 0 we set $f(x_{k+1})$ =0 and solve for x_{k+1} we get

$$\text{f}(x_k) + (x_{k+1} - x_k)\text{f'}(x_k) = 0 \Rightarrow x_{k+1} = x_k \text{ - } \frac{f(x_k)}{f'(x_k)}$$

Newton's algorithm displays very good properties in terms of convergence, especially with high accuracy in the neighbourhood close to the solution.

```
**Algorithm** Newton
*Initialize*: $x_1$ =...
for $k = 1, 2, 3..$ do
  $x_k = x_{k-1} - f(x_{k-1})/f'(x_{k-1})$
  if converged then
    stop
  end if
end for
```

The benefit of using Newton instead of bisection lies at the faster convergence rate. But the gain comes with costs. The function in hand must be differentiable and the derivative must be supplied. Quite frequent we have functions that are differentiable but the computation of the derivatives is extremely difficult, especially when talking about n >1 where we involve the Jacobian matrix of partial derivatives.
As far as the drawbacks are concerned, Newton's methods depend heavily on the initial guess. As a consequence, the global convergence is likely to fail and the method does not always guarantee that it will converge. For more details on the subject[4,5,6]

- Secant The major drawback in terms of computation for the newton method is that it is very difficult to compute and evaluate the derivative both at the beginning and at every step of the method. So a better choice would be to avoid the computation of the derivative and try to evaluate the function instead. These methods are called Quasi-Newton methods because they consist of iterations over successive linear equation solving problems but they differ in the sense that they do not require the computation of the derivative. Obviously this gain comes with the drawback of slower convergence compared to the Newton's method.
What secant method suggests is that when evaluation of the derivative is a problem, we could do the following iteration:

$$x_{k+1} = x_k - \frac{f(x_k)}{m}$$

where m is the approximation of the derivative and is the following:

$$m_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

This number, which is actually a difference quotient, uses the already-computed values of the function f from the current as well as the previous iterate. Thus we have:

$$x_{k+1} = x_k - \frac{f(x_k)}{\frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}} \Rightarrow x_{k+1} = x_k - f(x_k) * \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

It has been shown that the convergence of the secant algorithm is superlinear (quotient 1.62) which is faster than bisection but slower than Newton. Nevertheless when evaluating an algorithm the convergence rate is not the only factor we take into consideration. The number of floating-point operations, or just flops, per iteration should also be examined. Even though an algorithm may have a faster rate of convergence if it requires many flops then it takes longer to reach a certain level of accuracy. This is particularly the reason why Secant is faster, in practice, than Newton. Moreover, the Secant method doesn't require the knowledge and hence the computation of the derivative, just one function. On the negative side the algorithm is also not so robust and also requires two initial guesses. [4,5,6]

```
Algorithm Secant
Initialize: $x_1$ =..., $x_2$ =...
for $k = 1, 2, 3...$ do
  $x_{k+1} = x_k - f(x_k) * \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$
  if converged then
    stop
  end if
end for
```

3.3 Stochastic Root Finding Problems (SFRP)

3.3.1 Introduction

The generalization of the deterministic root finding problem is called the stochastic root finding problem. The significant difference between the deterministic and the stochastic case is that in the former the function f is known where in the latter all we have is an estimator $\overline{Y}_m(x)$ (which has to hold certain attributes for example consistency) of f(x). What this means in practise is that a large sample is required in order to obtain an accurate value f(x) for any value of x in the domain of f. We now state a formal definition of the SRFP as found on *Chen and Schmeiser, 1994b, 2001:*

Given a constant vector $\gamma \in \mathbb{R}$ and a (computer) procedure for generating, for any $x \in \mathbb{R}$, a q-dimensional consistent estimate $Y_m(x)$ of $g(x)$ such that $Y_m(x) \Rightarrow g(x)$ as $m \Rightarrow \infty$ for all $x \in \mathbb{R}$ find the unique root of x^* satisfying $g(x)= \gamma$ using only the procedure.

3.3.2 Stochastic Approximation

We begin again by recalling the well-known Newton's method for the DRFPs on the function g: $\mathbb{R}^n \to \mathbb{R}^n$:

$$x_{n+1} = x_n \text{ - } (\nabla \text{ g}(x_n))^{-1} g(x_n)$$

where the function g and the gradient ∇ -g are assumed to be known, which is a major difference for the SRFPs as we have already mentioned, since in the latter only a consistent estimator of g is available.

In 1951 Herber Robbins and Sutton Monro introduced a variation of the above iteration by substituting the estimator G for g and including a carefully chosen sequence a_n aimed at eliminating the effects of randomness:

$$X_{n+1} = X_n - a_n \cdot G(X_n) \text{ (2)}$$

This algorithm is what we call today the "Robbins-Monro" algorithm, it has been thoroughly studied since and a few important variations as well improvement have been made such as the one suggested by the Pollyart- Ruppert.

3.3.3 Robbins - Monro

Again we will present the general case and later on we will see how this algorithm will be tailored to our case for the utility based shortfall risk. We would like to minimize a function $g(\theta)$ over some region Θ subset of $\mathbb{R}^d$. We assume that the function is smooth. Let us also assume that the algorithm is at iteration n and has produced an approximation θ_n to θ^*, it then generates a r.v. Y_{n+1} having an expectation close to $\nabla(\theta_n)$. If we are able to obtain an unbiased gradient estimator then the conditional expectation of Y_{n+1} given θ_n is exactly $\nabla(\theta\text{n})$. Therefore, at every iteration the algorithm tries to converge to the true zero of the $\nabla(\theta_n)$. The use of random variables along with expected value is what makes the algorithm "random" in nature. [10]

3.3.4 Polyak - Ruppert

The convergence of the Robbins-Monro algorithm depends on the choice of $\frac{c}{n^\gamma}$. Ruppert (1988), Ruppert (1991), Polyak (1990), and Polyak and Juditsky (1992) suggested that instead of just relying on the last iteration x_n, to average the results of the slowed Robbings Monro algorithm. In our examples, we treat the Polyak-Ruppert algorithm as an average of the values produced from the Robbins-Monro algorithm. That is, if x_i is the result of the i simulation we average producing the following:

$$\bar{x} = \textstyle\sum_1^n x_i$$

Therefore instead or running the algorithm just n times, because we are averaging we also impose N simulations, each consisting of n steps. Therefore, in the Robbins-Monro context, the algorithms concludes after n steps and in the Polyak-Ruppert case, the algorithm concludes in Nxn steps. The latter will serve our framework for the analysis and implementation of our examples.[13]

For terms of complete presentation we will briefly refer here the five criteria by which the above and the rest of the algorithms for SRFPs can be evaluated and these are the following:

1. Numerical Stability

   Numerical stability is the actual performance of the algorithm when implemented into a computer in contrast with the theoretical performance that measures that don't take into account factors such as computer arithmetic or floating-point arithmetic.

2. Robustness

   Robustness addresses the issue of how sensitive is the algorithm to the changes of the initial values of the parameters in the algorithm. This means that a good algorithm shouldn't rely on specific initial values to and should perform well irrespective of them.

3. Convergence

   Convergence refers to asymptotic convergence in some probability measure. Obviously we prefer algorithms that guarantee convergence over those that don't, although convergence may not be indicative of the finite time performance of the algorithm.

4. Computational efficiency

   The relation between the quality of the solution and the amount of computing effort involved is expressed by a (tradeoff) function. One suggested measure is the expected value of the product of work with squared error. The smaller values indicate higher efficiencies. Computational efficiency is interesting because it combines the finite and asymptotic performance of the algorithm in terms of the solution quality and the effort needed in obtaining the solution.

5. Ability to report solution accuracy

   The main issue here is that since we only have an estimator of the function, the solution at each stage is random. The use of Variance is the only measure we could use but this means that the variance itself is an unbiased estimator. If the bias though is persisted through the iterations of the algorithm then the variance will be a poor estimator and therefore a poor measure of accuracy.

[17]

3.3.5 Convergence

What is of utmost importance of course if the convergence of the method for which a rich literature lies [7,8,10,11]. We base our presentation now on the paper of the R. Pasupathy and S. Kim [11].

As we have seen in section 3.3.2, Robbins-Monro (1951) introduced the estimator function G instead of g. In SFRP in general, no assumptions about the nature of the estimator function $G_m(x)$ are made apart from the fact that as $m \to +\infty$ it should converge in probability to g(x): i.e. $G_m(x) \xrightarrow{d} g(x)$ as $m \to +\infty$. Therefore all we know and we are given at the beginning is the following (as taken from [11]):

A simulation capable of generating, for any $x \in D \subset \mathbb{R}^q$, an estimator $G_m(x)$ of the function $g : D \to \mathbb{R}^q$ such that $G_m(x) \xrightarrow{d} g(x)$ as $m \to +\infty$ for all $x \in D$."

We return to (2) and write it a little bit different but having a clear goal in mind: separating the deterministic and stochastic parts in the equation:

$$X_{n+1} = X_n - a_n G(X_{n)}$$

$$X_{n+1} = \overbrace{(X_n - a_n g(X_n))}^{deterministic} - a_n \overbrace{\{G(X_n) - g(X_n)\}}^{stochastic} \; (3)$$

where the part in parenthesis is the "deterministic" and the part in brackets represents the "stochastic" part. (We insert the quotes because X_n is a random variable and so the term deterministic doesn't really hold). The important here is to observe that apart from the conditions

that function g must hold (e.g. monotonicity), condition(s) for the sequence a_n should hold as well in order to drive the "stochastic" part to zero.
Three criteria that guarantee convergence are the ones set by Robbins Monro original paper [7]:

(a) The function satisfies

$$g(x^*) = 0 \begin{cases} g(x){<}0 & \text{for } x < x^* \\ g(x){>}0 & \text{for } x > x^* \end{cases}$$

(b) The positive valued sequence $\{a_n\}$ satisfies $\sum_{n=1}^{\infty} \mathrm{a}_n = +\infty$ and
(c) The sum $\sum_{n=1}^{\infty} \mathrm{a}_n^2 < +\infty$

The first two conditions drive the deterministic part towards the zero of the function g (which is what we are looking for) and the last condition is aimed at driving the stochastic component to zero (we would like to get rid of the "randomness").
As the literature evolved other conditions have been added. For example Blum [1954a] established a.s. convergence of (2) under the same conditions as Robbins and Monro [1951] [7] but weakens the conditions for the structure of the root-finding function g. This is the following theorem:

Theorem 1. *Convergence of Stochastic Algorithms in one Dimension.*
We assume the following:
A1. $\sum_{n=1}^{\infty} a_n = \infty$
A2. $\sum_{n=1}^{\infty} a_n^2 < +\infty$
A3. $|g(x)| < c(|x - x^*| + 1)$ *for all x and some* $c > 0$
A4. $Var(G_x(X_n)) < \sigma^2 < \infty$ *for all x*
A5.

$$\begin{cases} g(x){<}0 & \textit{for } x < x^* \\ g(x){>}0 & \textit{for } x > x^* \end{cases}$$

A6. $\inf_{\delta_1 < |x - x^*| < \delta_2} g(x) > 0$ *for every pair of numbers* δ_1 , δ_2 *satisfying* $0 < \delta_1 < \delta_2 < \infty$.
Then $\{X_n\} \rightarrow x^*$ *a.s. as* $n \rightarrow \infty$ *[11]*

The conditions A3, A5, A6 specify the structure of the function g: A3 sets the maximum growth rate, A5 ensures the maximum of one root and the last condition A6 ensures that the iterates do not accumulate at any finite point. These three conditions along with A1 are focused at the deterministic part of the recursion (3). The remaining conditions (A2,A4) are used to drive the stochastic component in (3) to zero. Therefore we observe the similarity in conditions

between those set by Robbins and Monro and Blum. We also have to notice that a_n is a sequence of positive real numbers, which practically means that they are predetermined ahead of time and cannot therefore be a function of the iterates X_n.
In our thesis we are particularly interested in problems in one dimension. Nevertheless for purposes of complete presentation we will just present the conditions that guarantee convergence in multiple dimensions and these are the following:

Theorem 2. *Convergence of SA methods in multiple dimensions.*
Assume the following:
A1. $\sum_{n=1}^{\infty} a_n = \infty$
A2. $\sum_{n=1}^{\infty} a_n^2 < +\infty$
A3. There exists a positive-valued function $f(x)$: $D \Rightarrow \mathbb{R}$ *with unique minimum at* x^*, *and having continuous first and second partial derivatives such that:*

- $\sup_{\epsilon \leq ||x-x^*||} \nabla f(x)^T g(x) < 0$ *for all* $\epsilon > 0$, *where* $\nabla f(x)$ *is the column vector of first partial derivatives of the function* f *at* x, *and,*
- $E[G_x(X_n)^T H(x) G_x(X_n)] < \infty$, *where* $H(x)$ *is the matrix of second partial derivatives of the function* f *at* x.

Then $\{||X_n - x^*||\} \to 0$ *a.s. as* $n \to \infty$. *[11]*

3.3.6 Convergence Rates

Obviously in our context, the convergence of the algorithms is of utmost importance. After all an algorithm that doesn't converge at all or partially is not useful algorithm. Therefore the guarantee that the algorithm will converge is essential but not enough. Another characteristic we ought to closely look is the convergence rate of the algorithm, that is how fast the algorithm convergences. In the best case scenario we would like to have an algorithm that converges and it does it fast. Fast convergence means small computation time.
Particularly the a.s. convergence of the SA algorithms ensures that the distance between the iterates X_n and the solution we are seeking x^* converges to zero. But it doesn't provide any additional information about how fast the iterates approach the solution, an information that we would like to possess. The convergence rates can be obtained by establishing the asymptotic distribution for the iterate X_n.

Theorem 3. *Convergence Rates for SA in one dimension.*
Consider the unconstrained SA algorithm. Assume that the assumptions for a.s. convergence hold. Further, assume
A1. g has a positive derivative $g'(x^*)$ at x^*
A2. α is in $(\frac{1}{2}$,1], n^α a_n for some α >0, and if α =1, a >$\frac{1}{2g'(x^*)}$
A3. the function $\sigma^2(x) = Var(G_x(X_n))$ is continuous at x^*
A4. $\sup_{|x-x^*|<\epsilon} E[(G_x(X_n) - g(x))^2 I\{|G_x(X_n) - g(x)| > K\}] \to$ 0 as $K \to \infty$, for some $\epsilon >$ 0.
Then, $n^{\frac{\alpha}{2}}(X_n - x^*) \xrightarrow{d}$ N(0,s^2) is a Gaussian random variable with mean zero and variance s^2. The asymptotic variance s^2 = $\alpha\sigma^2(x^*)/2g'(x^*)$-1 if α=1, and $\alpha\sigma^2(x^*)/2g'(x^*)$ otherwise. [11]

Let us comment the above conditions. Condition A1 just describes that function g is strictly increasing in a neighborhood of x^* while A2 sets the decay rate of the sequence a_n. The assumption A3 implies that the variance of the stochastic components converges at x^* while A4 provides an essential condition to obtain asymptotic normality of the iterates.
The main point is that the convergence of the SA depends on the sequence a_n. If $a = 1$ the optimal convergence rate $O(1/\sqrt{n})$ can be guaranteed which is in line with the result in a typical central limit theorem.
Since we have established the criteria for convergence and convergence rates, it is time to proceed to the presentation of our project, where we use the algorithm of Robbins Monro (to be explained later) along with stochastic and deterministic root finding algorithms to solve the problem of the utility-based shortfall Risk. But first let us present the powerful and interesting connection between the stochastic algorithms and the parallel programming on the implementation side of things.

3.3.7 Stochastic algorithms and parallel programming

Two of the reasons we employ stochastic algorithms to solve root-finding problems are apparently that they are robust and fast. The first characteristic, robustness, refers to the structure of the algorithm and we can't improve it unless we have the function and the design in hand. As for the speed of the algorithm, there we could improve it by employing parallel programming. The justification for the use of parallel programming stems from the fact that stochastic algorithms are loop-independent (the steps of the loop body are independent of one another), meaning that we could run each iteration of the algorithm separately, which translates to one iteration per core (or per CPU if you prefer). All we have to do then is to combine the final result. This feature

allows us to experience important change in terms of time, as we will present later.
Parallel programming as an idea is nothing more than running your code in parallel which means in practice running your code on a separate core, hence the name parallel. We therefore have a program to run and multiple cores to use. But when we run the code at the same core, it seems that the code is running on parallel but what the CPU actually does, is that is uses an algorithm for time sharing but optimized. Even if we have multiple cores, the CPU doesn't utilize the other cores unless is instructed to do so. So if we have large project we enormous data, as it is often the case in the finance industry, then, running it on a simple core we experience a lot of latency. Instead if we could run the code, or even better, if we could create or structure the code in such way that enables the program to run on multiple cores then we get the computational advantage of speed and accuracy.
In practice parallel programming is more than just specifying how many cores to use. We have to create the program so that it uses every possible degree of parallelism, which depends on the hardware architecture. Therefore the design of the algorithm is very important. Even if we have an already established program we have to re-design it in order to use this computational benefit. On the pure implementation side several environments offer the option of executing the code in parallel. The C++ language offers the Parallel Patterns Library or PPL which is come with the 2010 visual studio and sets all the essential details needed on the hardware level in order to execute the code. In addition to C++, Matlab also offers the same possibility. In our examples we used matlab and the parafor loop feature.

4 Applications

Proposition 1 implies that the value of the $SR_{l,\lambda} = s^*$ is the unique zero of the function $g(s) = E[l(-X - s^*)]$ - λ. Therefore the problem is now how to find the roots of a given function, hence it's a root-finding problem. In order to do that we split-up the problem into two phases:

1. Compute the expected value $E[l(-X - s^*)] = \lambda$
2. Employ root-finding algorithms to find the root, either stochastic or deterministic.

There are a number of ways to address the above two points. First of all we are facing the problem of computing the expected value which could be done by either of the following methods:

(a) Monte Carlo.

Generate sample paths of the underlying distribution from a very large sample. Then due to the law of large numbers the mean value we are calculating will converge to the expected value we are trying to find.

(b) Fourier Transform

Instead of generating sample path and therefore doing simulation we could instead try to compute the integral itself (since the expected value over a given probability measure is an integral). The Fourier transform will do exactly that, by transforming the loss function and then computing the integral (by numerical integration but that's not a simulation).

Second of all, we will use both deterministic and stochastic root-finding methods, such as Secant and Robbins - Monro algorithm, respectively, to find the root of the function. The final objective will be to compare and contrast all the different methods and finally suggest the best solution to the problem in hand. From all the above plausible combinations, we summarize on the following matrix the methods we will use:

Root-Finding Algorithm	**Expected Value computation**
Deterministic	Monte Carlo
Stochastic	Monte Carlo
Deterministic	Fourier.

(I) **Deterministic RF with Monte Carlo**

As we have already define we have the function $g(s) = E[l(-X - s^*)]$ - λ. In order to find the roots of the function we will use the Secant method which is deterministic in nature and in order to compute the expected value we will use Monte Carlo. We remind that the secant method is the following:

$$s_{n+1} = s_n - g(s_n)\frac{(s_n - s_{n-1})}{(g(s_n) - g(s_{n-1}))}, \tag{5}$$

where the function $g(s)$ is the one above (*). The logic is quite simple. Loop through a number of times while calculating the expected value at each iteration using Monte Carlo simulation. At the end, the final value will be $s_N = s^*$ which is the root of the function and the value of the shortfall risk. This is the simplest method we could use to solve the problem. We will see whether this method is the best in terms of speed and accuracy.

(II) **Stochastic RF with Monte Carlo**

Instead of using the Secant method we will use the Robbins-Monro method. We will now describe how the algorithm is applied to the context of the shortfall risk and we will follow the presentation as in [13]. As a reminder from earlier the algorithm has the following structure:

$$X_{n+1} = X_n \text{ - } a_n \; G(X_n) \; (*)$$

In order to compute the expected value we will use a measurable function G_X: [0,1] $\rightarrow$ $\mathbb{R}$ such that $g(X) = E[\ G_X(U)\ ]$ for any on [0,1] uniformly distributed random variable U. The function $G(.)$ should be carefully chosen in order to increase the efficiency of the stochastic root finding scheme but a good choice, at least for our examples, is $G_X(U) =$ l(-$q(U) - X)$ - λ, where q is the quantile function of X. We would also like to bound and restrict the domain and to this end we define a projection function Π :$\mathbb{R} \rightarrow [a, b]$ onto the interval $[a, b]$ by

$$\Pi(x) \begin{cases} a & \text{for x} \leq \text{a} \\ x & \text{for a <x <b} \\ b & \text{for b} \leq \text{a} \end{cases}$$

Now the Robbins-Monro algorithm can be constructed in the following way:

(a) Choose a constant $\gamma \in$(1/2, 1], c >0 and a starting value $X_1 \in [a, b]$

(b) For $n \in \mathbb{N}$ we define recursively:

$$X_{n+1} = \Pi[\ X_n + \frac{c}{n^\gamma} * G_{X_n}(U_n)]$$

where the $\frac{c}{n^\gamma}$ is the a_n sequence of (*) and $G_{X_n}(U_n)$, for a sequence (U_n) of independent, uniform [0,1] distributed random variables.

We could observe the similarity in both the secant method and Robbins-Monro algorithm. The difference lies in the way we choose the starting values, the use of random variables and the fact that the Monte Carlo method is used only once, for the whole algorithm, rather than every step of the loop. The advantage of this algorithm is the possibility to run each loop on a separate core and therefore taking advantage of the hardware architecture by utilizing parallel programming while the secant method cannot be employed on a parallel environment.

(III) **Deterministic RF with Fourier Transform**

In this case we will use the Secant method again but instead of using the Monte Carlo simulation to compute the integral we will use the Fourier transform. For this presentation we will follow the presentation as in [3]. We denote the dampened function $g(x) = e^{-Rx}l(x)$ for some $R \epsilon \mathbb{R}$. Moreover, let $\hat{g}$ denote the Fourier transform of a function g, i.e., $\hat{g}(u) = \int e^{iux}g(x)dx$ and M_X the moment generating function of X, i.e., $M_X = E[e^{uX}]$ for suitable $u \epsilon \mathbb{C}$. We also define the domain of R to be the intersection of the following two sets:

$$I: = \{ \text{ R} \in \mathbb{R} : E[e^{Rx}] < \infty \} \text{ and}$$
$$J := \{ \text{ R} \in \mathbb{R} : l_R \in L^1_{bc}(R) \text{ and } l_R \in L^1(R) \}$$

Then the expected value can be computed as follows:

$$E[l(-X - s)] = \frac{1}{2\pi} \int_{\mathbb{R}} e^{(iu-R)s} M_{X_T}(iu - R) \cdot \hat{l}(u + iR)du$$

where $R \in I \cap J$.

Proof:

First we work on the dampened function

$$g(x) = e^{-Rx}l(x) \Rightarrow l(x) = g(x)e^{Rx} \tag{6}$$
$$l(-X - s) = g(-X - s)e^{R(-X-s)} = g(-X - s)e^{(-RX-Rs)} \tag{7}$$

where in the last equality we have substituted $x = (-X - s)$.

$$E[l(-X - s)] = \int_\Omega l(-X - s)dP = \int_{\mathbb{R}} l(-X - s)P_{X_T}(dx) = \int_{\mathbb{R}} g(-x - s)e^{R(-x-s)}P_{X_T}(dx) \tag{8}$$
$$= e^{-Rs} \int_{\mathbb{R}} e^{-Rx} g(-x - s)P_{X_T}(dx) \tag{9}$$

where in (8) we have used (7).

In order to derive the inverse of g we rely on [3,p6] where the following are established: By assumption, $g \ \epsilon L^1(\mathbb{R})$ and the Fourier transform of g, as we have already introduced:

$$\hat{g}(u) = \int e^{iux}g(x)dx$$

is well defined for every $u \ \epsilon \ \mathbb{R}$ and is also continuous and bounded. Additionally, by assumption $\hat{g} \ \epsilon \ L^1(\mathbb{R})$, we have also $\hat{g} \ \epsilon \ L^1_{bc}(\mathbb{R})$. Therefore, using the Inversion Theorem (cf. Deimtar 2004, Theorem 3.4.4), $\hat{g}$ can be inverted and g can be represented, for all $x \epsilon \mathbb{R}$, as

$$\begin{aligned} g(x) &= \frac{1}{2\pi} \textstyle\int_{\mathbb{R}} e^{-ixu} \hat{g}(u) du \\ g(-X-s) &= \frac{1}{2\pi} \textstyle\int_{\mathbb{R}} e^{-i(-x-s)u} \hat{g}(u) du \\ g(-X-s) &= \frac{1}{2\pi} \textstyle\int_{\mathbb{R}} e^{i(x+s)u} \hat{g}(u) du \end{aligned}$$

Now, we return to (9), we get

$$E[l(-X-s)] = e^{-Rs} \int_{\mathbb{R}} e^{-Rx} g(-x-s) P_{X_T}(dx) = e^{-Rs} \int_{\mathbb{R}} e^{-Rx} \Big(\frac{1}{2\pi} \int_{\mathbb{R}} e^{i(x+s)u} \hat{g}(u) du \Big) P_{X_T}(dx) \tag{10}$$

$$= \frac{e^{-Rs}}{2\pi} \int_{\mathbb{R}} e^{-Rx} \Big(\int_{\mathbb{R}} e^{ixu+isu} \hat{g}(u) du \Big) P_{X_T}(dx) \tag{11}$$

$$= \frac{e^{-Rs}}{2\pi} \int_{\mathbb{R}} e^{isu} \Big(\int_{\mathbb{R}} e^{ixu-Rx} P_{X_T}(dx) \Big) \hat{g}(u) du \tag{12}$$

$$= \frac{1}{2\pi} \int_{\mathbb{R}} e^{isu-Rs} \Big(\int_{\mathbb{R}} e^{(iu-R)x} P_{X_T}(dx) \Big) \hat{g}(u) du \tag{13}$$

$$= \frac{1}{2\pi} \int_{\mathbb{R}} e^{(iu-R)s} \Big(\int_{\mathbb{R}} e^{i(u+iR)x} P_{X_T}(dx) \Big) \hat{g}(u) du \tag{14}$$

$$= \frac{1}{2\pi} \int_{\mathbb{R}} e^{(iu-R)s} \Big(\phi_{X_T}(u+iR) \Big) \hat{g}(u) du \tag{15}$$

$$= \frac{1}{2\pi} \int_{\mathbb{R}} e^{(iu-R)s} M_{X_T}(iu-R) \hat{g}(u) du \tag{16}$$

$$= \frac{1}{2\pi} \int_{\mathbb{R}} e^{(iu-R)s} M_{X_T}(iu-R) \hat{l}(u+iR) du \tag{17}$$

where the for (11) to (12) we have applied Fubini's theorem and for the last equality we have:

$$\hat{g}(u) = \int_{\mathbb{R}} e^{iux} e^{-Rx} l(x) = \int_{\mathbb{R}} e^{iux-Rx} l(x) = \int_{\mathbb{R}} e^{i(u+iR)x} l(x) = \hat{l}(u+iR) \tag{18}$$

So we have proved the following:

$$E[l(-X-s)] = \frac{1}{2\pi} \int_{\mathbb{R}} e^{(iu-R)s} M_{X_T}(iu-R) \cdot \hat{l}(u+iR) du \tag{19}$$

What is left then is to define the moment generation function in hand and compute the fourier transform of the loss function. Both actions are relatively easy for well-known distributions and functions. The Secant method is as described in the case (I) and we note that at each step the expected value will be computed using the fourier method. The advantage we foresee is the computation speed in the calculation of the expected value between the simulations of the Monte Carlo method and the "pure" computation using the Fourier transform

4.1 Examples

We are now in place to apply the above using two specific loss functions: the exponential and the piecewise polynomial function. We note that we will use the loss amount L instead of the portfolio value X. L is simply the negative of X, that is X = -L. What this means is that positive values of L correspond to losses where negative values to gains.

4.1.1 Example I-1: Deterministic - Exponential Loss function

In the first example we will use the exponential loss function

$$l(-x)=e^{-\beta x},\ \beta>0$$

with normal (Gaussian) distribution, $l \sim N(\mu_0,\sigma_o^2)$. In this case the value of the utility-based shortfall risk can be calculated in a closed form:

$$E[l(-X-s)] = E[l(e^{(-X-s)\beta})] = e^{-s\beta}E[e^{-X\beta}] = e^{-s\beta}e^{-\beta\mu+\frac{1}{2}\beta^2\sigma^2} = \lambda \Leftrightarrow \tag{20}$$

$$-s\beta - \beta\mu + \frac{1}{2}\beta^2\sigma^2 = \log\lambda \Leftrightarrow \tag{21}$$

$$s = -\mu_o + \frac{1}{2}\beta\sigma^2 - \frac{\log\lambda}{\beta} \tag{22}$$

where we used the fact that $E[e^{-X\beta}]$ is the moment generating function of $N(\mu_o,\sigma^2)$, $E[e^{-X\beta}] = e^{-\mu\beta+\frac{1}{2}\beta^2\sigma^2}$.

So by plugging in values of β=0.5, λ=0.05, μ_o=0, σ_0=1 we obtain the value SR = s^* = 6.2415. We will now perform the same calculation at Matlab, by using the Secant and Monte Carlo method. We have to choose the N simulation runs and n the number of steps per simulation run (the algorithm will run Nxn times in total). For this example we will use N =1000 and n=100, with initial values α_1= 10, α_2=1. The execution took 1.98 seconds and we got the result of s^* = 6.2431 which is close enough to the value computed by the closed form formula. As we have already mentioned for the Secant method the two initial guesses, which are the starting values, influence the performance of the algorithm. If we test for example negative values or values above 12 the algorithm diverges. So it is a rather quick calculation but we have to be careful of the initial choices.

Deterministic Case - Monte Carlo		
N=1.000,n=100,μ_o=0,σo=1,λ=0.05	**Exponential loss(β=0.5)**	**Piecewise loss (η=2)**
Time (in sec)	1.98	1.92
SR value = s^*	6.2431	0.8303
Note: The convergence of the algorithm (Secant) depends heavily on the choice of the two initial starting values.		

Table 1: The results of our first two examples for each loss function, using the Secant method and Monte Carlo simulation for the computation of the expected value. We present the time of execution as well as the value s^*.

4.1.2 Example I-2: Deterministic - Piecewise Loss Function

Unlike the previous example, the piecewise loss function cannot be evaluated on a closed form solution. Therefore we have to compute it numerically by using a computer. The piecewise loss function is the following:

$$l_\eta(L-s) = \eta^{-1}(L-s)^\eta 1_{L>s} = \begin{cases} \eta^{-1}(L-s)^\eta & \text{for } L \geq s \\ 0 & \text{for } L < s \end{cases}$$

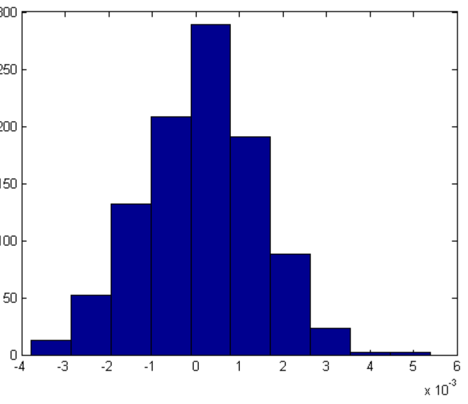
again with normal (Gaussian) distribution, $L \sim N(\mu_o, \sigma_0^2)$

We set N=1000, n=100, η=2, μ_o=0, σ_o=1, λ=0.05 and we run again the Secant and Monte Carlo method. The algorithm ran for 1.92 seconds and returned the value of $s^* = 0.8303$. Again we see that the computation is rather fast but again we have the dependence on the initial guesses. If we try negative values, zero or values that are far from the solution then the algorithm diverges.
We summarize the findings of our two first examples (I-1,I-2) by using the deterministic root-finding algorithm Secant and computing the expected value using Monte Carlo simulation.

4.1.3 Example II-1: Stochastic - Exponential Loss Function

We now turn our attention to the stochastic root-finding algorithms again with Monte Carlo simulation. To this end we will use the Robbins-Monro algorithm. We define the following inputs,

β=0.5, λ=0.05, γ=0.7, n=100, N=1000. Next we define the domain close to the root (since we have already analytically obtained the root $s^* = 6.2415$ from (22)) as $[a, b] = [s^* - 10, s^* + 10]$=[-3.7585,16.2415]. The algorithm runs for 2.23 seconds and returns the value s^*= 6.2508, which is close enough to the computed value from the closed form. We present here the histogram of the s_n values to check whether or not the distribution is normal. We could state that the distribution is standard normal.



We have a few observations to make. It took a little more time for the algorithm to execute but the range of the domain is quite broad, accepting also negative values, a flaw the secant method had. So Robbins-Monro doesn't have a dependence on the initial values, as we previously had. It is a mere a feature of how well we know the neighborhood of the root. If we are certain we could look around a rather narrow domain.But we have to note the dependence on the sequence $a_n = \frac{c}{n}$. The choice of this sequence is of utmost importance because by changing these numbers the algorithm responds differently.

At this point we will prove that the sequence used in exponential loss function (and in turn in piecewise loss function later) satisfies the three (3) conditions set by Robbins-Monro that guarantee convergence. We have the function $g(s) = E[l(-X - s)] - \lambda$ and the sequence $a_n = \frac{c}{n}$. The second and third condition requires the divergence and convergence of the sequence, respectively. In order to arrive at such conclusions we will use the integral test to examine the convergence or not of the sequence. We examine the second condition first; the third condition will follow naturally from the second. We note that the sequence is just like the sequence of $\frac{1}{x^p}$. So if we could derive a result for this sequence, then the same applies to our sequence in hand as well.

$$\int_1^t \frac{1}{x^p} dx = \frac{t^{-p+1}}{-p+1} - \frac{1}{-p+1} = \frac{1}{(-p+1)t^{p-1}} + \frac{1}{p-1} \tag{23}$$

$$\begin{cases} p = 1 & \text{the limit does not exist} \\ p{>}1 & \text{the integral converges} \\ p{<}1 & \text{the integral does not converges} \end{cases}$$

So for $p < 1$ the sum does not converge. Similar is the case for the third condition, where we want the squared sum of the sequence to be finite.

$$\int_1^t \frac{1}{x^{2p}}dx = \frac{t^{-2p+1}}{-2p+1} - \frac{1}{-2p+1} = \frac{1}{(-2p+1)t^{2p-1}} + \frac{1}{2p-1} \tag{24}$$

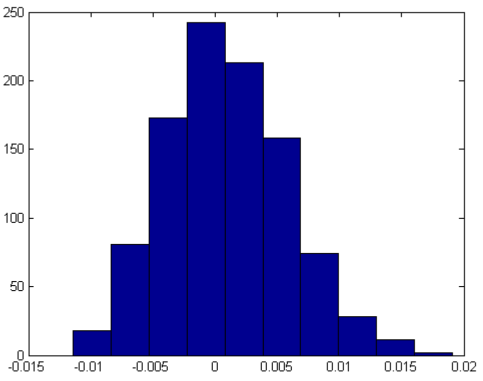
$$\begin{cases} p = \frac{1}{2} & \text{the limit does not exist} \\ p{>}\frac{1}{2} & \text{the integral converges} \\ p{<}\frac{1}{2} & \text{the integral does not converges} \end{cases}$$

In this case we want the integral to converge, so we choose the $p{>}\frac{1}{2}$ values. Therefore, we established the conditions set by Robbins-Monro so the algorithm is guaranteed to converge. We also derive the range of the γ value which is $\gamma \in (\frac{1}{2}, 1]$.

4.1.4 Example II-1: Stochastic - Piecewise Loss Function

We try the second example with piecewise loss function by using Robbins - monro algorithm. This time we use N=1000 and n=1000 because with n=100 the algorithm doesn't converge to the value. The domain is $[a, b]$ = [s^*-5,s^*+5]=[-4.1306,5.8694]. In case there is no analytical solution available much wider bracket $[a, b]$ must be provided. The rest of the parameters remain almost the same as with the previous example: c=100, γ=1, λ=0.05. The algorithm runs for 21.79 seconds and returns the value of s^*=0.8980 which is not so close to the observed value of 0.8693.

We observe that the algorithm took much more time to execute than the deterministic case but this is mainly because of the number of steps per simulation (n) which is now 1000 instead of 100. If we put the same n at secant method we also get the same value of 0.8928 and 22.065 seconds of execution. The advantage is again the independence from the initial values. We also present the histogram of the s_n values to see whether or not it follows a normal distribution. We could state that the values follow a standard normal distribution.



We summarize here our finding, after running both examples with the Robbins-Monro algorithm and Monte Carlo simulation:

Stochastic Case - Monte Carlo		
N**=1.000,**n**=100,**μ_o**=0,**σo**=1,**λ**=0.05**	**Exponential loss(**β**=0.5)**	**Piecewise loss (**η**=2)**
Time (in sec)	2.23	21.79
SR value (s^*)	6.2508	0.8980

Table 2: The results of our first two examples for each loss function, using the Secant method and Monte Carlo simulation for the computation of the expected value. We present the time of execution as well as the value s^*.

Note: The algorithm took longer time to execute but the convergence is much more guaranteed than in deterministic case. The algorithm is independent of the initial starting values since they are randomly generated from a predefined interval when the algorithm begins.

4.1.5 Example III: Deterministic - Piecewise Loss Function

Here instead of changing the root-finding algorithm, we keep the deterministic Secant algorithm and change the way we compute the expected value. Instead of using simulation, we are computing directly the integral through Fourier transform. Obviously we resort to numerical integration at the implementation level. In addition, we are only going to focus on our second example, i.e. on the piecewise loss function because we cannot apply the fourier transform to the first example (on the exponential function). As we have already proved the expected value under the Fourier transform is given by:

$$E[l(-X-s)] = \frac{1}{2\pi}\int_{\mathbb{R}} e^{(iu-R)s} M_{X_T}(iu-R)\cdot \hat{l}(u+iR)du \tag{25}$$

We have to derive the moment generating function and the fourier transform of the loss function. The following calculations explain how to do so.

Fourier Transform of the Piecewise Loss Function

The piecewise loss function is the following:

$$l_2(L-s) = 2^{-1}(L-s)^2 1_{L>s} = \begin{cases} 0.5(L-s)^2 & \text{for } L \geq s \\ 0 & \text{for } L < s \end{cases}$$

We take for $s=0$ in the above expression and derive the fourier transform of the function l for $z \in \mathbb{C}$

$$\hat{l}(z) = 0.5\int_0^{+\infty} e^{izx}\cdot x^2 dx = 0.5(e^{izx}(\frac{x^2}{iz} - \frac{2x}{(iz)^2} + \frac{2}{(iz)^3}))|_0^{+\infty} \tag{26}$$

The exponential in order not to explode we have to impose the restriction: $\mathfrak{Im}(z) > 0$ which means $b > 0$.

$$lim_{x\to+\infty} e^{i(a+bi)x} = lim_{x\to+\infty} e^{(ia-b)x} = 0,\, given\, b > 0 \tag{27}$$

Also the exponential will converge much faster than the polynomial x^2 so the first term will be zero. Therefore,

$$\hat{l}(z) = 0.5\int_0^{+\infty} e^{izx}\cdot x^2 dx = 0.5(e^{izx}(\frac{x^2}{iz} - \frac{2x}{(iz)^2} + \frac{2}{(iz)^3}))|_0^{+\infty} \tag{28}$$

$$= 0.5(0 - e^0(\frac{0}{iz} - \frac{2\cdot 0}{(iz)^2} + \frac{2}{(iz)^3})) = -0.5\frac{2}{(iz)^3} = -\frac{1}{(iz)^3} \tag{29}$$

Therefore

$$\hat{l}(z) = -\frac{1}{(iz)^3} \tag{30}$$

Moment generating function

This is easy, since L follows $N(0,1)$ the moment generating function with $\mu = 0$ and $\sigma = 1$ is:

$$M_{X_T}(u) = e^{\mu u + \frac{1}{2}\sigma^2 u^2} = e^{\frac{1}{2}u^2} \tag{31}$$

$$M_{X_T}(iu-R) = e^{\frac{1}{2}(iu-R)^2} \tag{32}$$

Combing the above results (30),(32), our expected value now has the following representation:

$$E[l(-X-s)] = -\frac{1}{2\pi}\int_{\mathbb{R}} e^{(iu-R)s} e^{\frac{1}{2}(iu-R)^2} \cdot \frac{1}{(iu-R)^3} du \tag{33}$$

Implementing the above formula into Matlab, or any other software with numerical capabilities, is relatively easy and straight forward. By choosing a value for $R = 5$ we get the value 0.8694 in time almost close to 0.052116 seconds. Therefore, we already see how accurate and quick the fourier transform could be.

4.2 Comparison of I-II: Deterministic - Stochastic RFA

Now we will perform a series of tests each time changing a parameter and observing how the algorithm will perform in terms of time and convergence. We will mainly focus on the number of simulations (N) and on the number of steps per simulation (n). We will take the histogram and the q-q plot each time along with the s^* values and the time needed to execute the algorithm. Through these case studies we will observe how each algorithm performs and try to conclude which is better in terms of computation time and desired outcome.

4.2.1 Exponential Loss Function

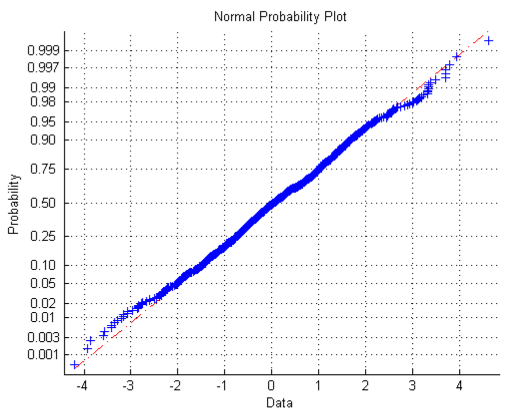
For the first case study we keep constant the number of simulations (N) and change each time the number of steps (n). For this reason we will keep N equal to 1000 and n will gradually take the following values: 100, 500, 1000, 5000, 10.000, 50.000, 100.000 and once the algorithm concludes we will save the histogram and the q-q plot of the stochastic algorithm apart from the time taken and the value s^*. After running both the stochastic and the deterministic case we get the following results:

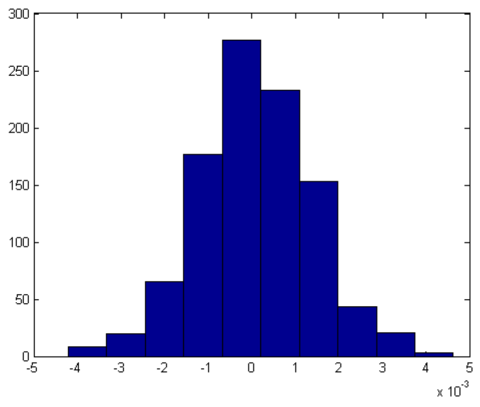
	Stochastic		Deterministic	
N=1.000	Time of execution	Value	Time of execution	Value
n=100	2.2	6.2515	2.09	6.2346
500	11.13	6.2497	11.42	6.2401
1.000	20.38	6.2422	20.92	6.257
5.000	114.26	6.2438	114.18	6.2351
10.000	218.44	6.2417	204.21	6.2511
50.000	1167.91	6.2407	1154.5	6.2301
100.000	2337.81	6.2417	2069.68	6.2667

We observe that the time taken for both algorithms to complete is almost the same as the ratio of time of the stochastic against the time of the deterministic is almost 1 for all the cases. As for the values we see that the stochastic gives us better and more stable results since the value is always in the 6.2407-6.2497 range. The deterministic gives a broader and not so stable range from 6.2351 - 6.2667. Even at the last case with 100.000 number of steps per simulation where we expect good results the deterministic algorithm fails to monitor closely the desired value whereas the stochastic algorithm gave the exact value of s^*.

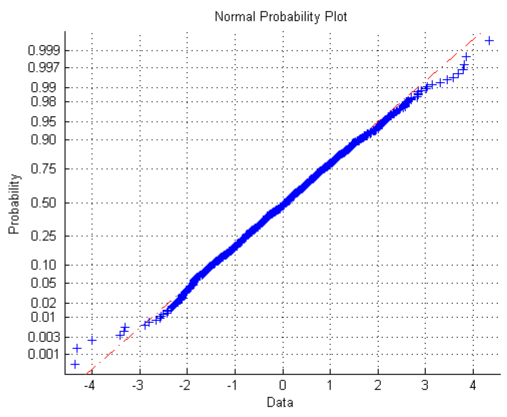
As for the computation time we could easily see that every time we increase the number of steps by a multiple of ten (100-1000-10.000) we also have an increase in time in multiple of 10 (2.2 - 20.38 -218.38). In addition every time we increase the number of steps by a multiple of 5 (100-500, 1000-5000) we also notice an increase in time of execution of a multiple of 5 (2.2-11.13, 20.38-114,26). In general, we observe the almost linear relationship between the number of steps and the time of execution. We present the histogram and the q-q plot of the stochastic algorithm for the exponential loss function:

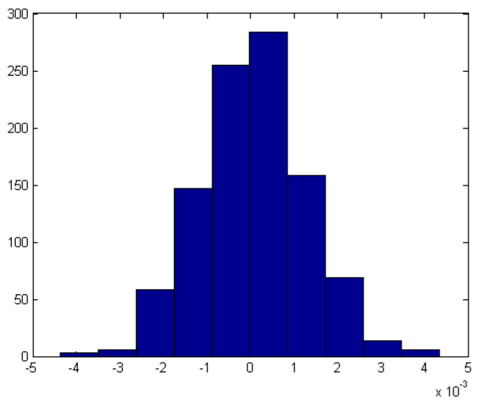
n=100



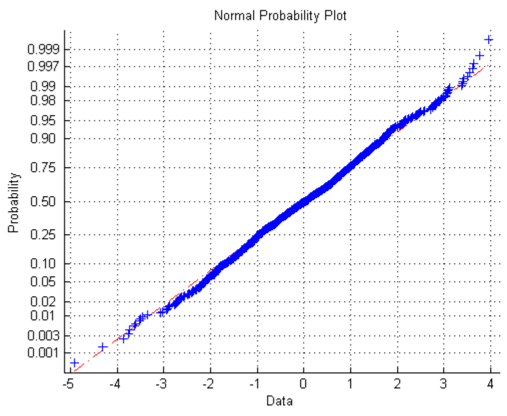


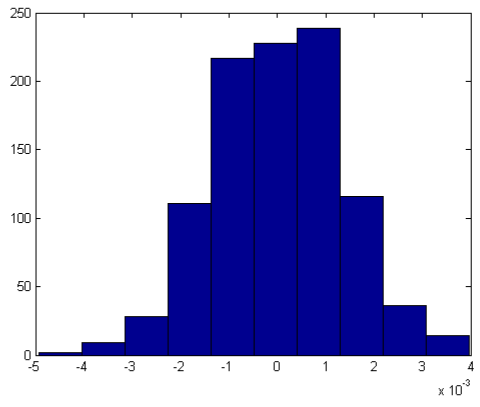
n=500



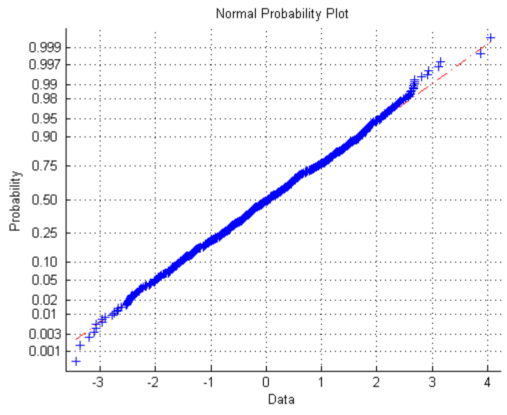


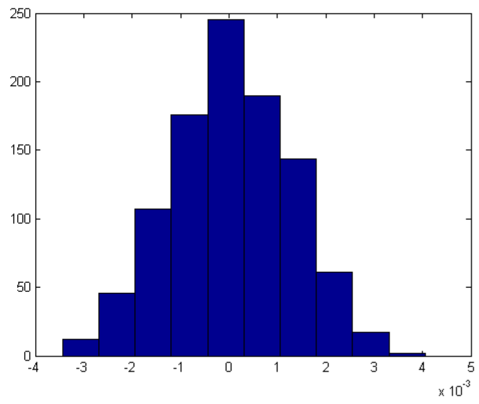
n=1.000



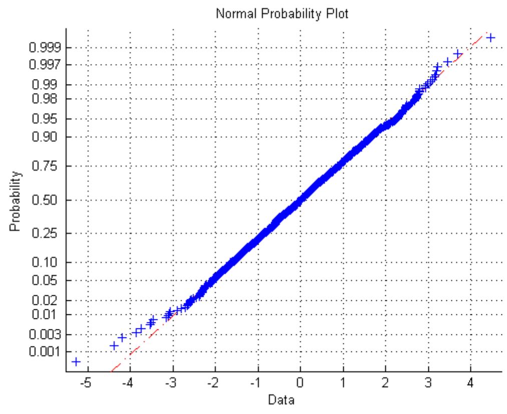


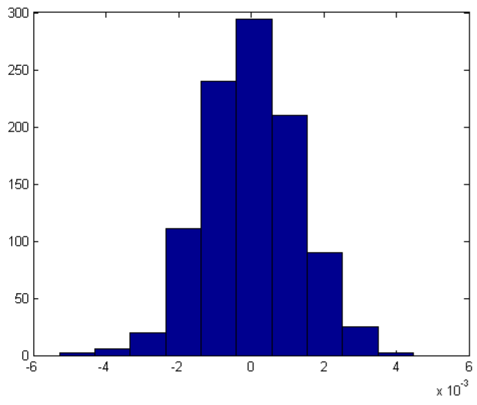
n = 5.000



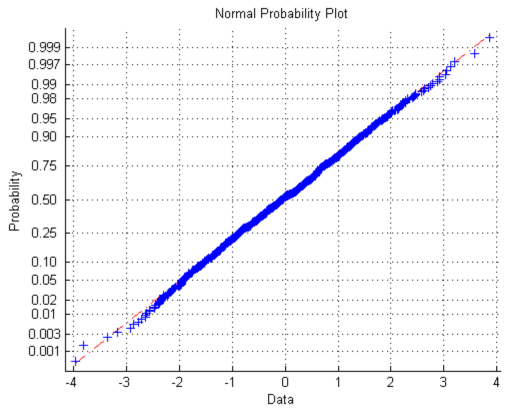


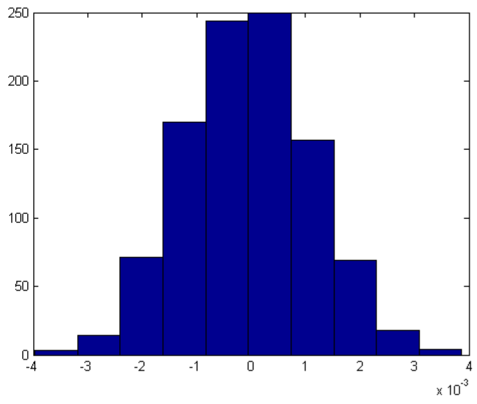
n = 10.000

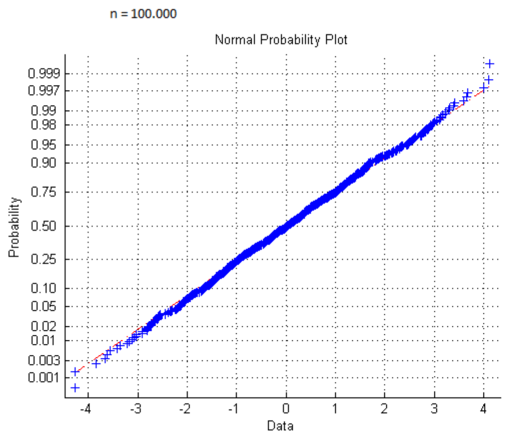


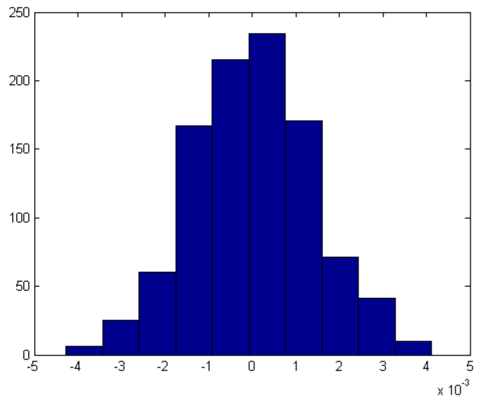


n = 50.000







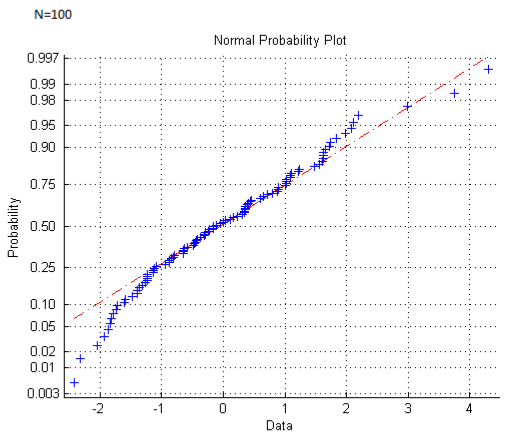


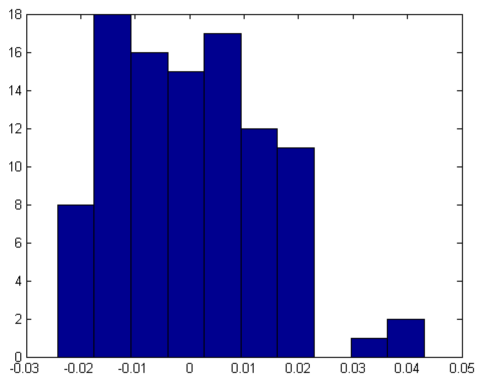
We could conclude that the errors (the computed value against the actual solution) follow a standard normal distribution. For less than 1000 steps we could also argue that the fit is not so good, we observe the "bad fit" at the q-q plot. But as we increase the number of steps it gets better and better. In the last example with $n = 100.000$ the fit is very good; only some values are not on the q-q plot.

For the second case study we hold the number of steps per simulation constant (n) and we modify the simulations runs (N) in order to observe how the behavior of the algorithm changes when the number of simulations changes. We follow the same pattern, that is, we keep the number of steps per simulation run (n) constant at 1000 and for the N we tried the following values: 100, 500, 1000, 5000, 10.000, 50.000, 100.000. After the execution of the algorithm we save the histogram and the q-q plot of the stochastic algorithm as well as the time of execution and the value s^*. After running both the stochastic and the deterministic case we get the following results:

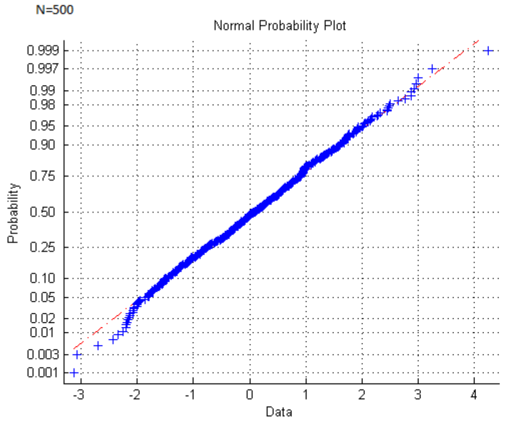
	Stochastic		Deterministic	
n=1.000	Time of execution	Value	Time of execution	Value
N=100	2.225	6.2436	2.26	6.2052
500	10.872	6.249	11.85	6.322
1.000	22.41	6.2462	22.28	6.2541
5.000	103	6.2443	121.66	6.2451
10.000	219.73	6.2418	217.95	6.2345
50.000	1094	6.2443	1122.9	6.2442
100.000	2226	6.2447	2312	6.2421

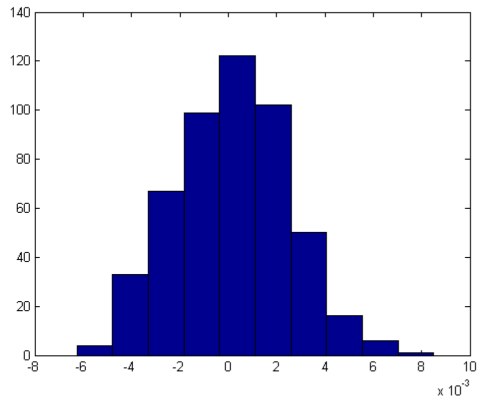
We observe almost the same results as previously. The stochastic algorithm keeps better track of the real value s_n, as it quickly converges even with the first try of one hundred simulation runs. The best approximation is achieved with 10.000 simulations. The deterministic part on the other hand, has a broader range of values but after increasing the simulation runs on 50.000 the value of s* is closer to the real value. The time taken is almost the same for both algorithms again and the relationship between time and simulation runs is also linear. We could easily observe that as we increase the simulations by ten the time of execution also increases by ten (100-2.225, 1.000 - 22.41).
What remains now is to see whether the stochastic algorithm follows the standard normal distribution and conclude which case is the best in terms of how well the results fit the normal distribution. We do not expect significant differences from the previous case.



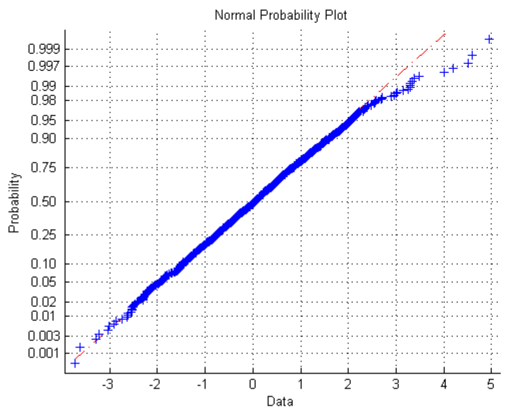


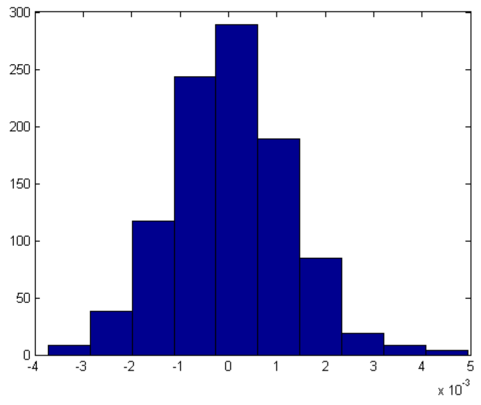
N=500



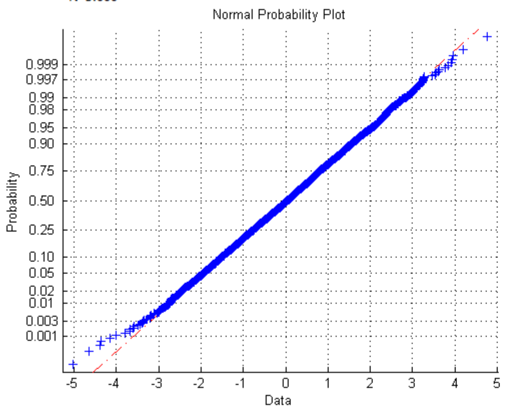


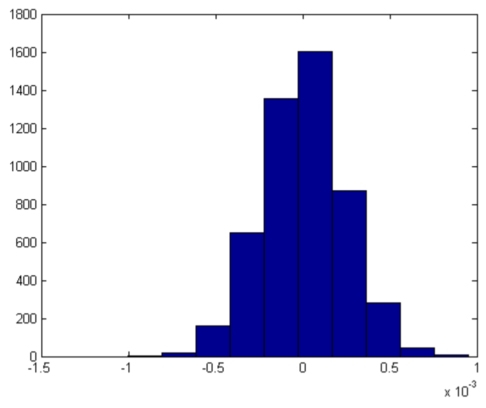
N=1.000



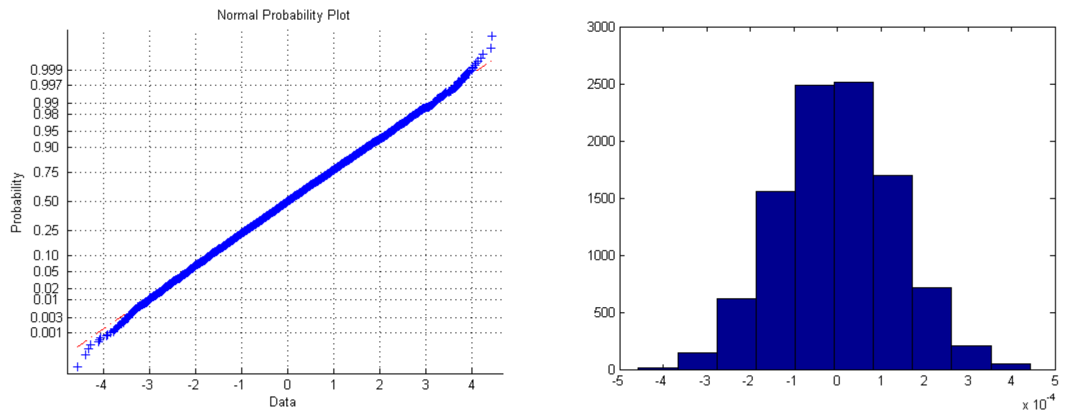


N=5.000

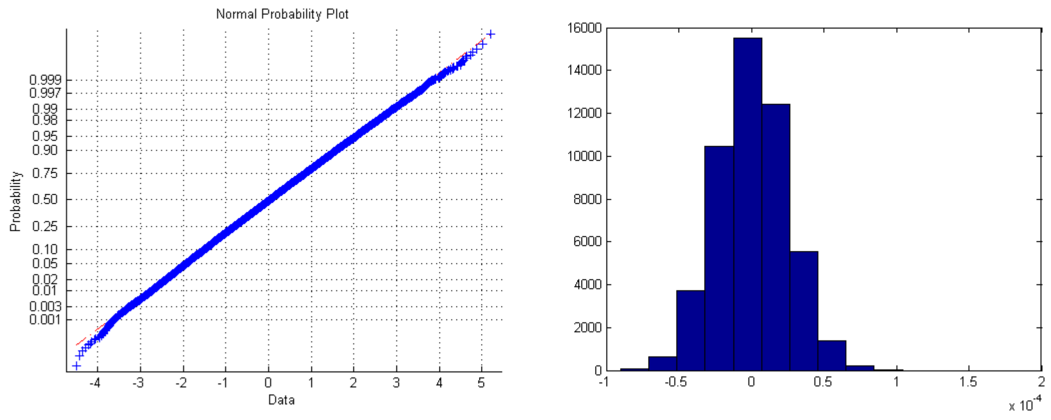




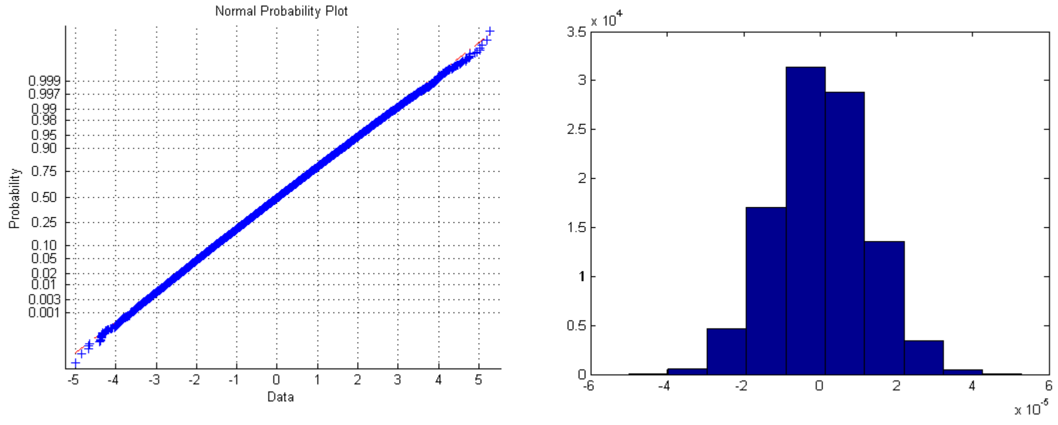
N=10.000



N=50.000



N=100.000



The best fit is observed for 100.000 simulation runs which rational given that the more

simulations we add the more the value will converge to the normal distribution (central limit theorem). Nevertheless, we observe from the q-q plot that the fit is almost perfect, evidence supports by the histogram as well. The bad fit is observed for small number of simulations, 100 or 500, especially in the former case. We start to get good results from 5.000 simulations and above. In the case of 1.000 the fit is quite good apart from some values that to deviate, which is also seen in the histogram, some values on the right.
We conclude the first case study regarding the exponential loss function that since the stochastic algorithm gives us better approximations, closer to the real value, than the deterministic and since the distribution is standard normal, as we expected to be, we derive as a first outcome that the use of the Robbins-Monro algorithm gives better implementation results but not to a significant degree.

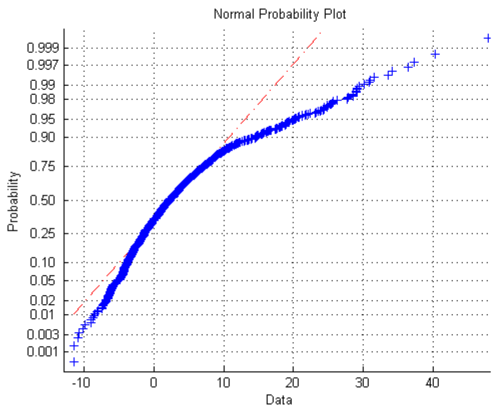
4.2.2 Piecewise Loss Function

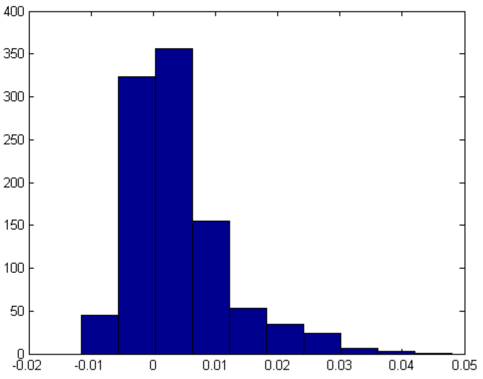
We turn our attention to the second example and we will use the piecewise loss function. The distribution is again standard normal. The main point here is that we cannot evaluate the function numerically by hand and therefore we have to rely on the numerical methods to derive the solution. Therefore we cannot know beforehand the solution as we did with the exponential loss function. The tests will be of the same format. First, we will fix the number of simulations (N) and secondly, we will keep the numbers of steps per simulation fixed (n) and observe both times how the algorithm behaves.
For the first case study we keep constant the number of simulations (N) and change each time the number of steps (n). For this reason we will keep N equal to 1000 and n will gradually take the following values: 100, 500, 1000, 5000, 10.000, 50.000, 100.000 and once the algorithm concludes we will save the histogram and the q-q plot of the stochastic algorithm apart from the time taken and the value s^*. After running both the stochastic and the deterministic case we get the following results:

	Stochastic		Deterministic	
N=1.000	Time of execution	Value	Time of execution	Value
n=100	2.26	1.2622	2.25	0.843
500	11.45	0.9313	10.854	0.8875
1.000	22.93	0.9002	21.9696	0.8759
5.000	111.36	0.8741	103.822	0.8859
10.000	220.96	0.8726	214.84	0.864
50.000	1102.93	0.8704	1109.48	0.8537
100.000	2170.3	0.8693	2290.86	0.836

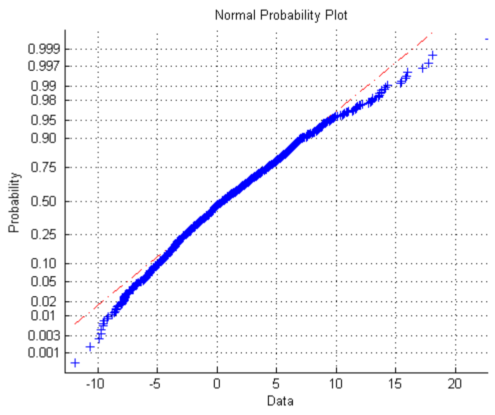
We observe that the time taken for both algorithms to complete is almost the same as the ratio of time of the stochastic against the time of the deterministic is close to one for all the cases. As the number of steps per simulation increases the better results we get and the closer we get to the real solution. For n=1.000 we have s^* =0.9002 and if we increase the steps ten times we get s^*=0.8726, which is close enough to solution. For the deterministic we see a better approximation early enough, even with 500 steps, but the more we increase the steps the approximation fails to follow the solution closely. It fluctuates from 0.83 to 0.88. As for the values of the stochastic, they are more stable since the value is always in the 0.8704-0.8793 range. For the last case of n=100.000, the stochastic returned the correct value (0.8393) while the deterministic failed to provide one (0.836).
As for the computation time we could easily see that every time we increase the number of steps by a multiple of ten (100-1000-10.000-) we also have an increase in time in multiple of 10 (2.26 22.93 -220.96). In general, we observe the almost linear relationship between the number of steps and the time of execution. We derived the same conclusion as we did with the previous example of the exponential loss function. We now present the histogram and the q-q plot of the stochastic algorithm for the piecewise loss function:

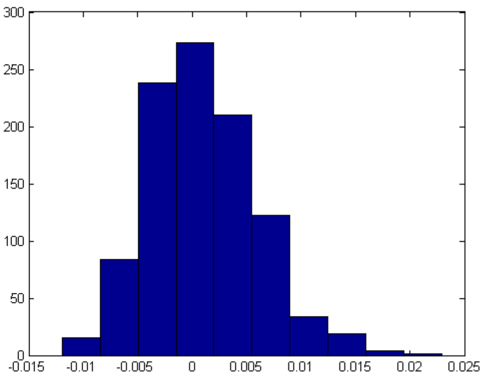
n=100



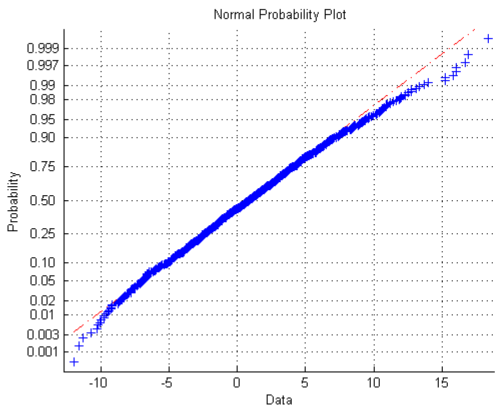


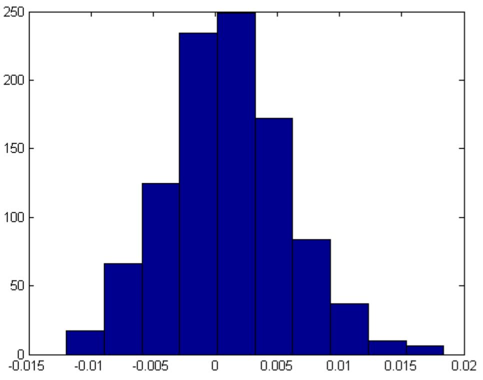
n=500



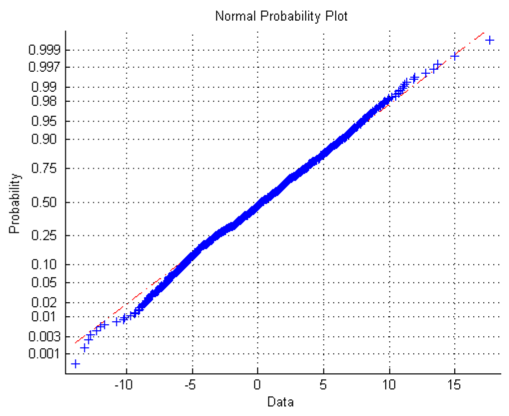


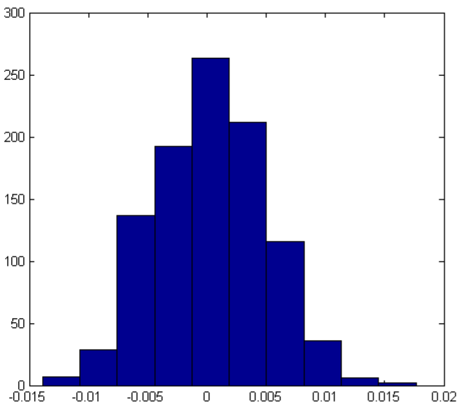
n=1.000



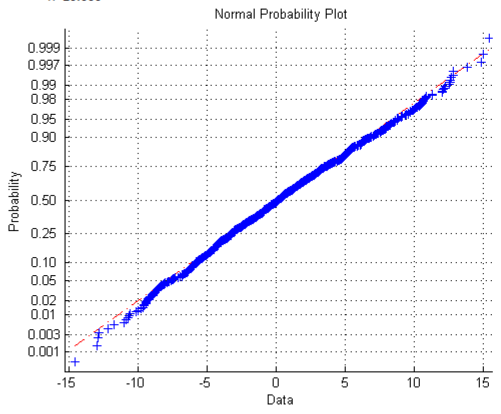


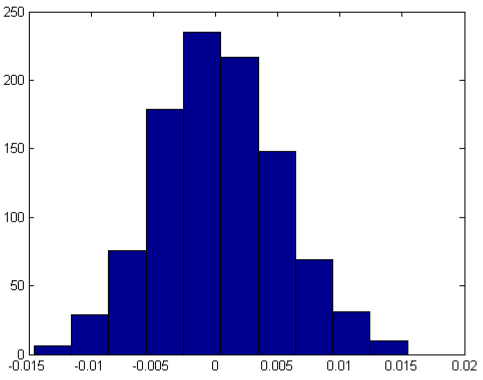
n=5.000



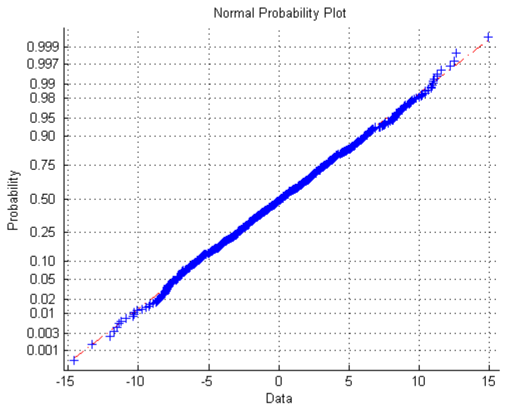


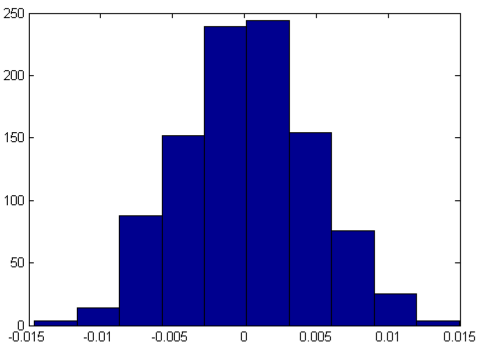
n=10.000



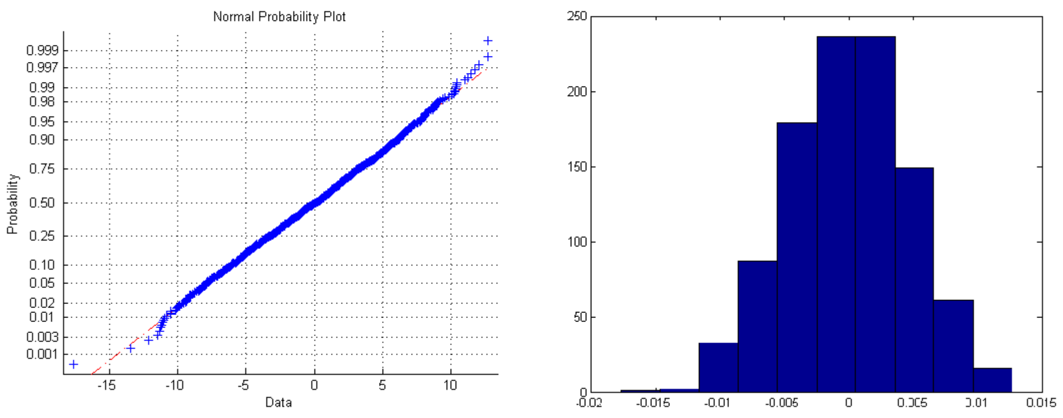


n=50.000





n=100.000

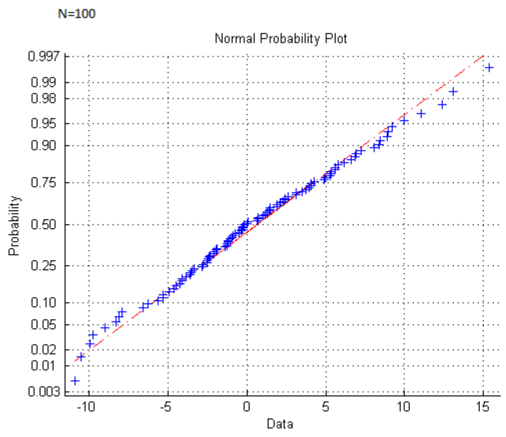


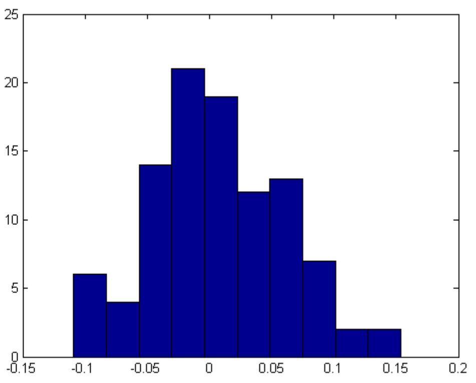
We could conclude that the errors follow a standard normal distribution. For less than and including 1.000 steps we could also argue that the fit is not so good, we observe the bad fit at the q-q plot. Especially for n=100 and n = 500 the distribution deviates a lot from the standard normal. The histograms also support this. But as we increase the number of steps the distribution converges to that of a standard normal. In the last example with n equal to 100.000 the fit is very good.

For the second case study we hold the number of steps per simulation constant (n) and we modify the simulations runs (N) in order to observe how the behavior of the algorithm changes when the number of simulations changes. We follow the same pattern, that is, we keep the number of steps per simulation run (n) constant at 1000 and for the N we tried the following values: 100, 500, 1000, 5000, 10.000, 50.000, 100.000. After the execution of the algorithm we save the histogram and the q-q plot of the stochastic algorithm as well as the time of execution and the value s^*. After running both the stochastic and the deterministic case we get the following results:

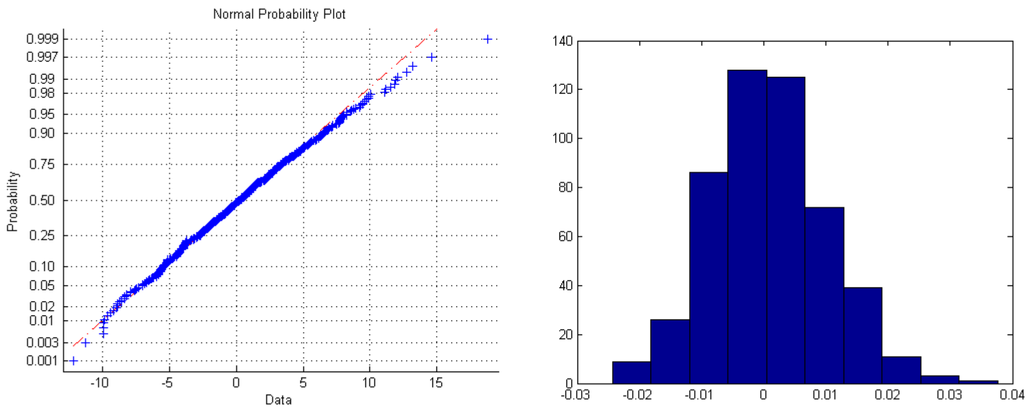
	Stochastic		Deterministic	
n=1.000	Time of execution	Value	Time of execution	Value
N=100	2.37	0.8885	2.29	0.9286
500	11.69	0.8817	11.5	0.8054
1.000	22.36	0.8929	21.51	0.8498
5.000	117.59	0.892	112.36	0.8653
10.000	249.45	0.8915	244.093	0.8684
50.000	1035.54	0.8929	1009.14	0.8668
100.000	2038.79	0.8932	1980.99	0.8645

The behavior of the results does not differ from the first example. The Robbins-Monro algorithm does not improve significant as we increase the number of simulations. On the contrary, the value stays at the 0.89xx level, far from the value of the 0.8693. On the other side of things, deterministic algorithm seems to give better results. It starts with the value 0.9286, which is far away from the correct solution, and gradually approaches the s^*. By 10.000 simulations we get the value of 0.8684 and similar results we get for 50.000 and 100.000 simulations with values of 0.8668 and 0.8645, respectively. Thus the deterministic seems to benefit from the increase of the simulations while the stochastic seems to have no advantage. We now present the q-q plot and the histograms to determine whether the values of the stochastic algorithm follow a standard normal distribution or not.

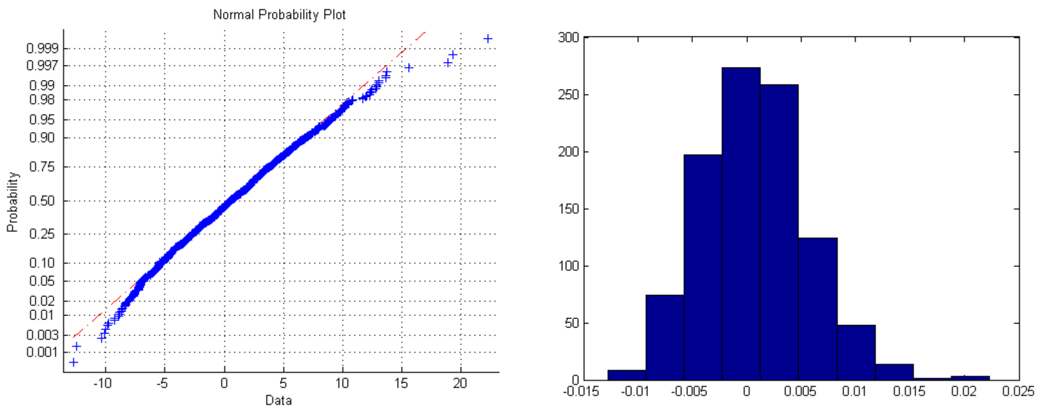




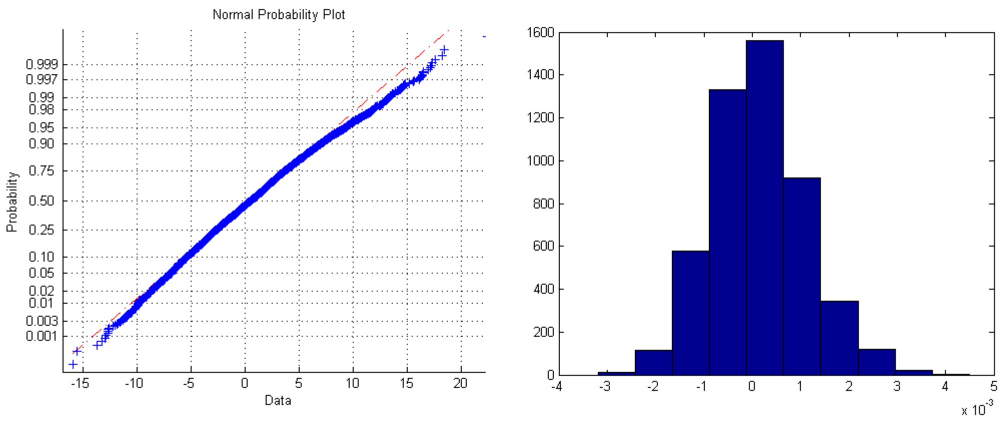
N=500



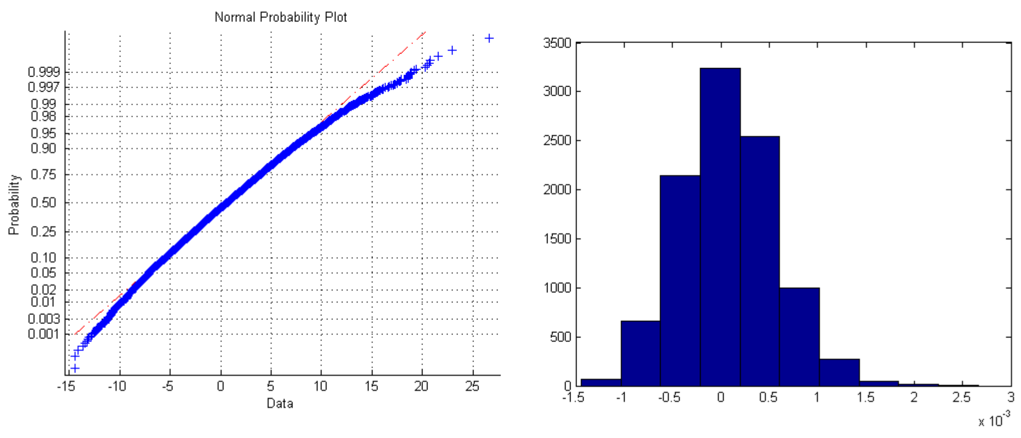
N=1.000



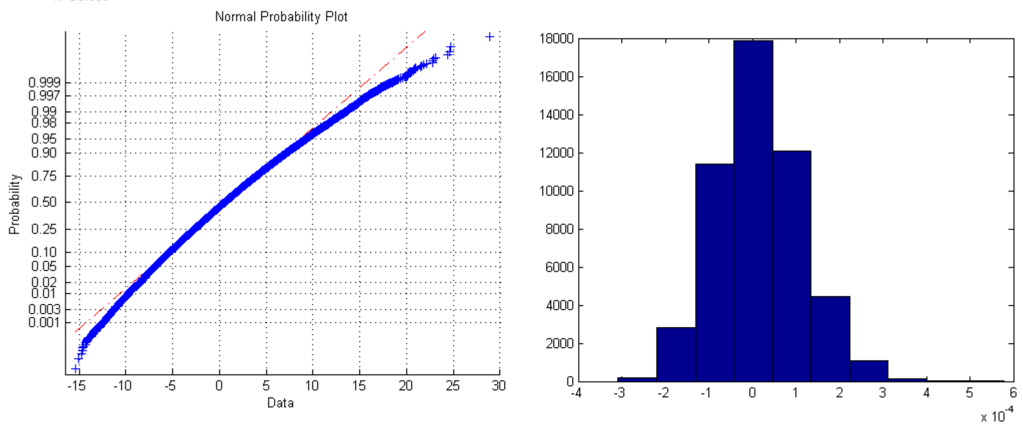
N=5.000



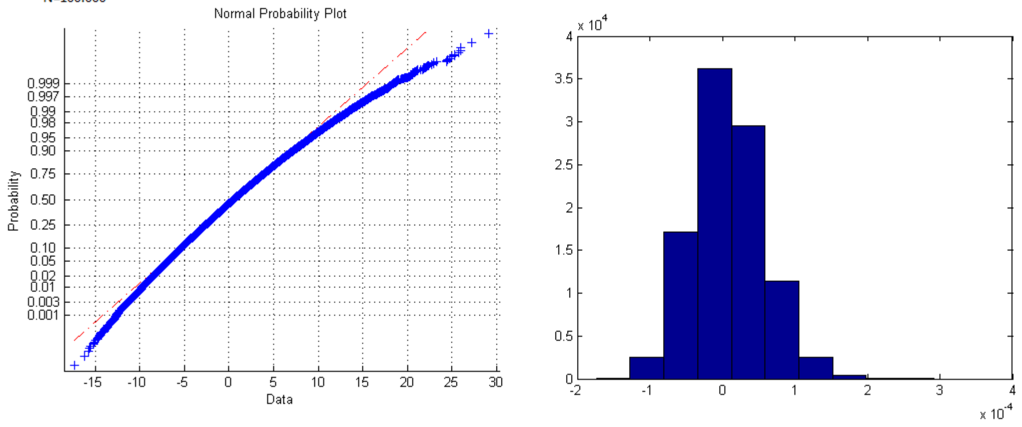
N=10.000



N=50.000



N=100.000



From early on we discern that the fit to the standard normal is not so good. As the simulations increase the fit gets better but there are still values that deviate, evidence supported by both the q-q- plots and the histograms. Even with 50.000 or 100.000 simulations the distribution is still not converging to that of a standard normal. This fact is in line with the previous results, as the stochastic algorithm failed to converge (closely) to the true value. If we had a better convergence of the distribution to the standard normal then the value would also converge to the correct value of s^*.
As a conclusion, we would use the stochastic algorithm by using a moderate number of simulations, 1.000 or 10.000, but with increased number of steps per simulations, 10.000 or more, as with these values we got the best approximations (n =100.000 and N=1.000 $\rightarrow$ s^*=0.8693). If we had to use the deterministic algorithm we would increase the number of simulations because by doing so the value converges to the correct value and keep the number of steps per simulations at a moderate level of 1.000 or 5.000. We therefore see that each algorithm requires different choices. In both examples, the distributions converged to a standard normal.
Now looking at both examples, we do not find any difference in the time of execution, as both algorithms in all four (4) examples required the almost the same amount of time to execute. The Robbins-Monro algorithm gave us better approximating values, in the sense that is better converged to the real value of s^*. The deterministic also converges but fluctuates over a broader range of values. In all four cases, the distributions were converging to that of a standard normal. Therefore, if we had to choose we would most probably choose the Robbins-Monro because of its better convergence behaviour.

4.2.3 Introducing and using Parallel Programming

Now we will present why and how the Polyart Ruppert, and the stochastic algorithms in general, could be proven to be superior to the deterministic ones. To this end, we will get advantage of the design of the algorithm. In the construction of the Polyart Ruppert algorithm, we performed N simulations, each having n steps per simulation. Because each simulation run is independent from the rest, we can isolate the calculation of each simulation and execute them independently. In the end, we collect the output of each simulation and construct the final result. This feature makes the algorithm a very good candidate to be expressed and executed as a parallel program. All we have to do, is to redesign the algorithm in order to fit to the standards applied by the parallel programming. The reason for this is because the sequential algorithm runs and behaves

in a different manner than the parallel equivalent algorithm. On top of that, each language and developing environment sets its own commands and design considerations.
By not having the algorithm any cross-loop dependencies it means that each loop is independent of the previous and the following loop, which practically means that it does not require any information from the other loops. Therefore, we could separate each loop and not lose any information or harm the algorithm, since each loop is independent (in terms of information). Since we could separate the loops, we could proceed a step further by distributing and executing them on separate cores and therefore execute them parallel in terms of time (the number of cores depends on the underlying infrastructure, but given the today advances in technology and that most financial institutions companies have access to this technology, this should not pose a problem.) So what we end up with, is executing each loop on a separate core therefore gaining significant in terms of speed, since now we dont use only one core but multiple, therefore running the program in parallel rather than sequential. Instead of waiting each loop to conclude and then proceed to the next loop, we now distribute the loops to the different number of cores, and once one loop is terminated the new loop is distributed to that core.
For example, in the sequential example we have only one core. Lets say that we have 100 iteration. Each time, only one iteration can be executed at the core and therefore each iteration should wait until the previous iteration has concluded in order to be executed. By having more cores, lets assume for example four (4), we could distribute the first four iterations to the four cores and each time that one iteration terminates we proceed and feed that core with the next iteration. There is no guarantee or assumption that the four iterations will conclude all at the same time (in order to say that we will feed the next four iterations to the four cores). For this reason, the 5th iteration could be fed to any of the four cores. For example if the 2nd iteration concludes first from the other four, then the 5th iteration will be fed to the 2nd core, and so on. Exactly this is the fundamental characteristic of the parallel algorithm. Spot any parts that could be executed on parallel and re-design the algorithm or design it from the beginning by having this characteristic in mind. For our examples we used the Matlab command of parfor. The only difference is that instead of using the normal for command, we use the parafor command, which is used for parallel execution. Obviously, we have to be certain that our program could be executed in parallel, as we have explained on the previous paragraph.
We expect therefore a noteworthy improvement in terms of time when we will execute the stochastic algorithms. We also have to keep in mind, that in order for the parallel programming to be effective we have to have a lot of iterations and/or many commands or calculations that

need to be executed. The reason for this is because of the overhead of the parallelism. We may win time by executing the program in many cores but we lose initially time by setting up the program, as it needs many more variables from the compiler point of view, to keep track of the different variables and executions. So if we dont have many commands or iterations this overhead could be to our disadvantage. This could be more easily explained if we consider an example of printing a number 100 times. For this example the sequential program will be faster because of this overhead. It will take some time until the parallel program sets up everything that it requires and therefore the gain will not be significant. Our examples though, as the most situations in the real world, are complex enough that gain from the use of the parallelism. The worst we could expect is that the parallel algorithm could perform as the sequential (every iteration would run one after the other). So anything above that we would beneficial.
We try to run on parallel both loss function, including the two different cases we have, one for the fixed number of simulations and one for the fixed number of steps per simulation. Therefore, we will be able to compare and contrast the three different cases: deterministic, stochastic-sequential, stochastic-parallel.

Exponential loss function We begin again from the exponential loss function. We ran the parallel program for the first case of fixed number of simulations (N=1.000) and we will present the results combined with the stochastic-sequential and deterministic cases, in order to be easy to compare and contrast:

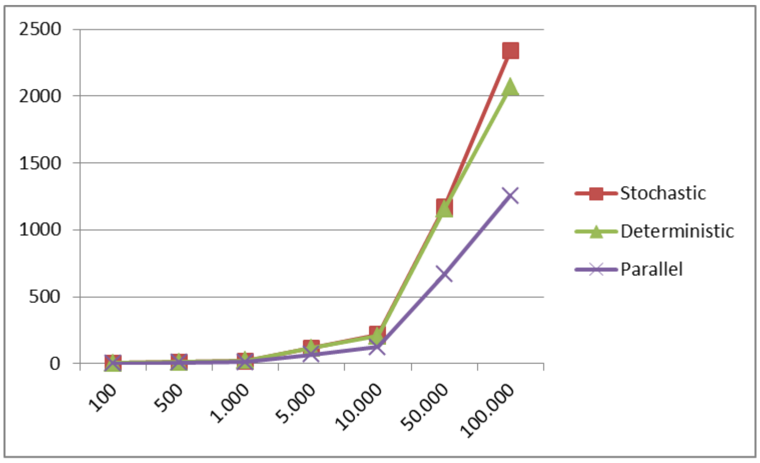
	Stochastic		Deterministic		Parallel	
N=1.000	Time of execution	Value	Time of execution	Value	Time of execution	Value
n=100	2.2	6.2515	2.09	6.2346	1.33	6.2575
500	11.13	6.2497	11.42	6.2401	6.56	6.2446
1.000	20.38	6.2422	20.92	6.257	12.75	6.2429
5.000	114.26	6.2438	114.18	6.2351	68.24	6.2449
10.000	218.44	6.2417	204.21	6.2511	123.68	6.242
50.000	1167.91	6.2407	1154.5	6.2301	665.54	6.2409
100.000	2337.81	6.2417	2069.68	6.2667	1256.49	6.242

As we observe the parallel algorithm runs a lot faster while it returns correct values, which converge to the real solution. The gain in terms of execution is quite important as we increase

the number of steps. In particular, for 10.000 numbers of steps, we see that both the stochastic and the deterministic perform close to the 210 seconds, the parallel executed in 123 seconds. The same gain is observed with 100.000 numbers of steps, with execution times for the stochastic, deterministic and parallel being 2337, 2089 and 1256 seconds respectively.
If we look closely the time of execution between the stochastic and the parallel algorithm we see that the ratio of time of execution of the parallel to the time of execution of the stochastic is in most cases roughly 0.60. This means that the improvement in terms of time is 0.4 percent. In particular, we present the ratios and the corresponding improvement for each case:

n	Ratio Parallel/Stochastic	Improvement
100	0.6045	39
500	0.5893	41
1.000	0.6256	37
5.000	0.5981	40
10.000	0.5661	43
50.000	.5698	43
100.000	0.5374	43

Therefore there is a significant gain in terms of execution time in using the parallel algorithm instead of the sequential algorithm. In our simple example we managed to turn a 38.95-minutes program run only in 20.93 minutes. Lastly, we present the graph of the execution time of each case in order to confirm graphically the improvement in time. On the vertical axis we have time in seconds and on the horizontal the number of steps per simulation.



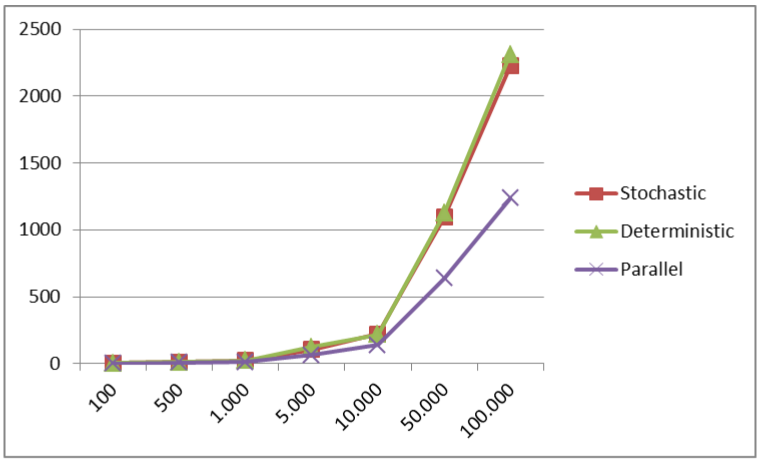
We observe how closely move, in terms of time, the stochastic and the deterministic algorithm. We also see that once the numbers of steps increases significantly, for example after 10.000, the improvement from the parallel algorithm is becoming more and more significant.
For the second case, we fix the number of steps per simulation (n) and we change the simulations number (N) each time. We present here the results:

	Stochastic		Deterministic		Parallel	
n=1.000	Time of execution	Value	Time of execution	Value	Time of execution	Value
N=100	2.225	6.2536	2.26	6.2052	2.17	6.2485
500	10.872	6.249	11.85	6.322	6.56	6.2483
1.000	22.41	6.2462	22.28	6.2541	12.46	6.246
5.000	103	6.2443	121.66	6.2451	64.0899	6.246
10.000	219.73	6.2418	217.95	6.2345	137.12	6.2432
50.000	1094	6.2443	1122.9	6.2442	637.8	6.2444
100.000	2226	6.2447	2312	6.2421	1235.88	6.2442

We get the same results as we previously. The parallel algorithm runs a lot faster while converging to the real value as well. As we increase the number of simulations the gain of the parallel becomes apparent and more useful. For 10.000 simulations the stochastic algorithm concluded in 219.73 seconds, a time similar with the deterministic 217.95 seconds. On the other hand the parallel algorithm ran in only 137.12 seconds. On the last case of 100.000 simulations both stochastic and deterministic finished in 2226 (37.1min) and 2312 (38.5 min) seconds respectively. The parallel finished in only 1235.88 (20.5 min) seconds which yields an improvement of 46 percent.

n	Ratio Parallel/Stochastic	Improvement
100	0.59	40.67
500	0.603	39.66
1.000	0.556	44.4
5.000	0.622	37.78
10.000	0.624	37.60
50.000	0.582	41.7
100.000	0.555	44.48

So once again the parallel algorithm has shown to be faster in terms of execution time. In both cases it runs faster while it converges to the real solution. We also provide the graph of the execution times for each case:



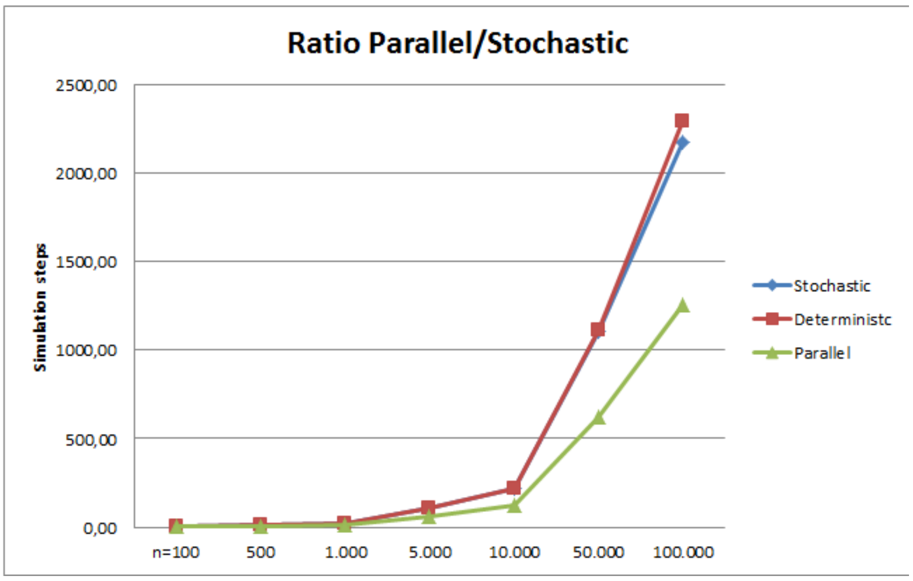
Piecewise loss function Just like we did with the exponential function we execute the parallel program for the first case of fixed number of simulations (N=1.000) and we will present the results combined with the stochastic-sequential and deterministic cases, in order to be easy to compare and contrast:

	Stochastic		Deterministic		Parallel	
N=1.000	Time of execution	Value	Time of execution	Value	Time of execution	Value
n=100	2.26	1.2622	2.25	0.843	1.36	1.2589
500	11.45	0.9313	10.854	0.8875	6.38	0.9071
1.000	22.93	0.9002	21.9696	0.8759	12.71	0.8993
5.000	111.36	0.8741	103.822	0.8859	63.65	0.8733
10.000	220.96	0.8726	214.84	0.864	124.57	0.8729
50.000	1102.93	0.8704	1109.48	0.8537	624.01	0.8691
100.000	2170.3	0.8693	2290.86	0.836	1254.91	0.8708

First of all we notice that the results are in accordance with the previous case, that is, the parallel execution was much faster. In particular, the gain in terms of execution is again significant as we increase the number of steps. For $n = 500$ the algorithm performs poorly, by returning the value $s^* = 0.9071$, despite the fact that concluded almost 40% faster. Better results we observe for $n = 10.000$ or $n = 50.000$, where that latter returns value $s^* = 0.8691$ in almost 624.01 seconds while the other algorithms conclude in 1100 seconds. Therefore the speed-up is definitely not negligible. Next we take a look at the ratio of the time of execution between the stochastic and the parallel algorithm.

n	Ratio Parallel/Stochastic	Improvement
100	0.60	40%
500	0.55	45%
1.000	0.55	45%
5.000	0.57	42%
10.000	0.56	43%
50.000	0.56	43%
100.000	0.57	43%

Once again we verify the fact that there is a significant gain in terms of execution time in using parallel instead of sequential execution. The speed-up in overall time is of the same magnitude as in the exponential example. Lastly, we present the graph of the execution time per number of steps to visualize this improvement. On the vertical axis we have the time in seconds and on the horizontal axis the number of steps per simulation.



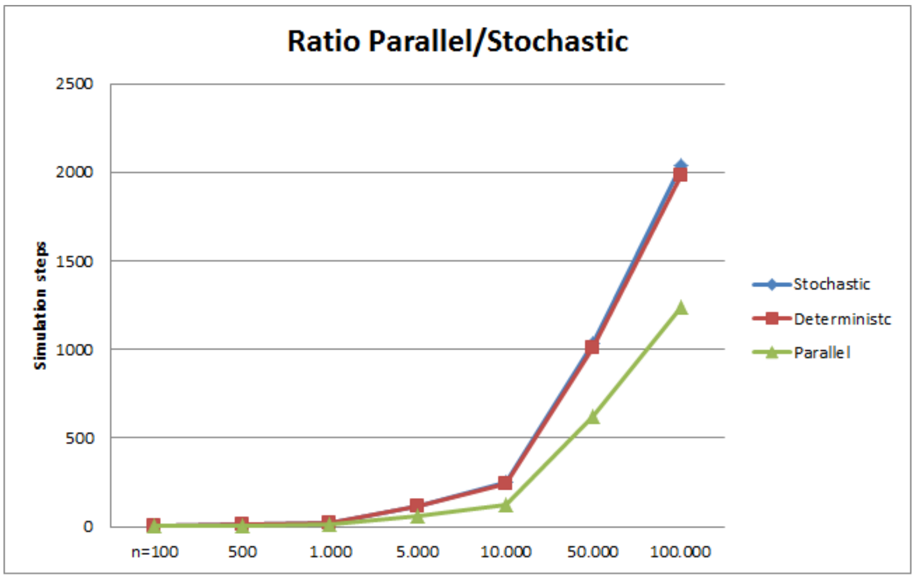
It is apparent that as we increase the number of steps per simulation the more we gain from the parallel schema. This is quite evident from the graph. We continue with the second case, where we fix the number of steps per simulation (n) and we change the simulations number (N) each time. We present first the results:

	Stochastic		Deterministic		Parallel	
n=1.000	Time of execution	Value	Time of execution	Value	Time of execution	Value
N=100	2.37	0.8885	2.29	0.9286	1.38	0.8775
500	11.69	0.8817	11.5	0.8054	6.43	0.8927
1.000	22.36	0.8929	21.51	0.8498	12.47	0.9012
5.000	117.59	0.892	112.36	0.8653	61.47	0.8904
10.000	249.45	0.8915	244.093	0.8684	123.35	0.8953
50.000	1035.54	0.8929	1009.14	0.8668	623.33	0.8923
100.000	2038.79	0.8932	1980.99	0.8645	1239.82	0.8920

Obviously the results of the parallel algorithm could not differ from the stochastic case. We also observe that the results do not quite converge to the real solution. We also see that across the different simulation runs the solution remains at the $0.89xx$ level. The only gain was as expected in the execution time. The results are almost the same as in the stochastic case but the time has significantly dropped. Therefore it we could state that for the piecewise loss function the increase in simulation runs does not bring any benefit nor any gain. Instead it is beneficial the increase in the number of steps per simulation. If had to choose, we would pick a specific number of simulations and we would change the number of steps, opting for the parallel execution. The ratio of parallel to stochastic time

n	Ratio Parallel/Stochastic	Improvement
100	0.58	42%
500	0.55	45%
1.000	0.56	44%
5.000	0.52	48%
10.000	0.49	51%
50.000	0.60	40%
100.000	0.61	39%

No big surprises here, the speed-up almost 56%. But having in mind the deviation from the true value the benefit is not so much beneficial. Yet the fact that we can employ a parallel scheme of execution by using the stochastic algorithm, holds. To remain consistent, we present the graph of execution times:



To sum up, by using parallel programming we observe a speed up of roughly 40%. The converge of the algorithms remains the same as in the sequential stochastic case. The only exception is for the piecewise loss function, where the increase in simulations does not add any value.

4.3 Comparison of I,II - III: Monte Carlo - Fourier Transform

In this section we let aside the root-finding algorithms and focus and compare the computation of the expected value. The root-finding algorithm will be deterministic in the case of Fourier. As we have already seen in section 4.1.5 on page 40 the execution time needed for the calculation of the integral while executing the deterministic root-finding algorithm, was minimal, approximately 0.05 seconds. This time is considerably faster from all the previous cases we have seen so far, including the parallel computation as well.
Particularly, for the piecewise loss function the times of execution with 1.000 simulations (N) and 5.000 steps per simulation (n) are 111.36 seconds, for the stochastic case, and 103.82 seconds, for the deterministic case. Even when we use parallel programming to take advantage of the stochastic scheme, the time of execution was 63.65 seconds. If we increase the number of steps per simulation in order to increase the accuracy, the time of execution increases to 220.96,214.84,124.57, for the stochastic, determinist and parallel case respectively. Therefore, we see that that computational gain is tremendous, even when compared to the parallel execution (0.05 sec to 63 sec). The same arguments hold when we examine the other case of holding 1.000 steps per simulation and 5.000 and 10.000 simulations (N). We summarise these results and present the times for the other cases in the following table:

N=1000	Stochastic	Deterministic	Parallel	Fourier
n=5.000	111.36	103.822	63.65	0.05
n=10.000	220.96	214.84	124.57	0.05

Table 3: N constant. All times in seconds

n=1000	Stochastic	Deterministic	Parallel	Fourier
N=5.000	117.59	112.36	61.47	0.05
N=5.000	249.45	244.09	123.35	0.05

Table 4: n constant. All times in second

Moreover, one good feature of applying Fourier transform is that it does not depend on any choice of N and n. Nevertheless, we need to have (or have a way to derive) the moment generating function and the fourier transform of the loss function should be solvable.
Therefore, in order to conclude the the comparison between the Fourier transform and the Monte Carlo simulation, we would choose the former, given the simplicity of implementation and the very very fast execution. It is also very simple to extend. We just need to specify the moment generating function and the new fourier transform. Thus we could create a library of function and choose even on run-time which one to calculate.

5 Final Conclusions

We have presented what risk measures are, which are the root-finding problems and algorithms, namely we have demonstrated numerically and explained the differences of deterministic and stochastic root-finding algorithms. Among the calculations required for the root-finding algorithms, expected value was one of them. We specifically focused on calculating the expected value using Monte Carlo simulation and the Fourier transform. All the aforementioned where applied to a specific risk measure, the utility-based shortfall risk and provided two examples, the exponential and piecewise loss functions.
From our simulations and programs, we did not observe any significant difference between the deterministic and the stochastic root-finding algorithms, with respect to time. There was though, a slightly better convergence of the stochastic case and therefore if we had to choose the one or the other, we would opt for that. Next we applied parallel programming techniques to the stochastic root-finding algorithm in order to gain advantage of the fact that each simulation is independent from the next as we are only interested in the final value of each simulation. There, we observed a significant speed-up which on average was of 40% magnitude, which definitely is a non-negligible percentage. This practically means that if we have no resources, then the choice of algorithm is just a matter of taste but if we have access to computational resources, which most big companies and firms have, then the choice is obviously the use of stochastic root-finding.
Finally, we examined how extremely fast fourier transform proved to be by computing the exact same result as the Monte Carlo simulation in just over 0.05 seconds, instead of at least 63 seconds, and this includes the time of the parallel scheme not the normal case. Thus, if we are in place of calculating the moment-generating function and the fourier transform of the loss function, then fourier transform method is the appropriate choice, since it is faster.

6 Bibliography

References

[1] A. J. McNeil, R. Frey, P. Embrechts, *Quantitative Risk Management*, Princeton University Press, 2005

[2] Bank of international Settlements : http://www.bis.org/bcbs/basel3.htm

[3] S. Drappeau, M. Kupper, A. Papapadoleon, *A Fourier Approach to the Computation of CV@R and Optimized Certainty Equivalents* January 1, 2013.

[4] I. Gladwell, J.G. Nagy, *Introduction to Scientific Computing*, Chapter 6, Root Finding, 2004.

[5] G.W. Recktenwald, *Finding The Roots Of f(x)=0* , supplement to the book Numerical Methods with Matlab: Implementations and Applications, August 11, 2006.

[6] K. A. Kopecky, *Root-Finding Methods*, lecture notes of Economics 613/614 - Advanced Macroeconomics I and II Computational Methods for Macroeconomics and Applications.

[7] H. Robbins, S. Monro,*A stochastic approximation method*, Ann. Math. Statist. Volume 22, Number 3 (1951), 400-407.

[8] T. L. Lai, *Stochastic Approximation*, Technical report No. 2002-31, September 2002.

[9] H. Follmer, A. Schied, *Stochastic Finance, An Introduction in Discrete Time*, Walter de Gruyter, Berlin, New York 2002.

[10] S. Asmussen, P.W. Glynn, *Stochastic Simulation, Algorithms and Analysis*, Springer, February, 2007.

[11] R. Pasupathy, S. Kim *The Stochastic Root Finding Problem: Overview, Solutions, and Open Questions*, ACM Transactions on Modeling and Computer Simulation, Volume 21, Issue 3, Article No. 19, March 2011.

[12] Fr. Delbaen, *Coherent Risk Measures*, Lecture Notes, Pisa, February, 28- March, 82000.

[13] J. Dunkel, St. Weber, *Stochastic Root Finding and Efficient Estimation of Convex Risk Measures*, Research Article, May, 28, 2009.

[14] C. Acerbi, C. Nordio, C. Sirtori, *Expected Shortfall as a Tool for financial Risk Management*, Research Article, Milano, Italy, February $1, 2008$.

[15] E. Eberlein, K. Glau, and A. Papapantoleon, *Analysis of Fourier transform valuation formulas and applications*, Appl. Math. Finance, $17 : 211240, 2010$.

[16] Ch. Bayer, *Computational Finance*, Lecture notes for Computational Finance course, TU Berlin, July, 152010.

[17] R. Pasupathy, Br. W. Schmeiser, *Some issues in multivariate stochastic root finding*, Research Article, Winter Simulation Conference, 2003.

[18] St. Weber, *Utility-Based Shortfall Risk and Optimized Certainty Equivalents*, Lecture notes for Computational Finance course, Leibniz Universitat Hannover, July, 152010.

[19] P. Carr, D. B. Madan, *Option valuation using the fast Fourier transform*, J. Comput. Finance $2(4), 6173, 1999$.

[20] H. Chen, B.W. Schmeiser, *Retrospective approximation algorithms for stochastic root finding*, In Proceedings of the 1994, Winter Simulation Conference,ed. J.D. Tew, S. Manivannan, D.A. Sadowski, and A.F Seila, 255261, Piscataway, New Jersey: Institute of Electrical and Electronics Engineers,1994b.

[21] H. Chen, B.W. Schmeiser, *Stochastic root finding via retrospective approximation* IIE Transactions $33 : 259275, 2011$

[22] Federal Reserve Bank of San Francisco, Economic Research: http://www.frbsf.org/economic-research/publications/economic-letter/2008/october/liquidity-risk/