Monte Carlo Methods for the Estimation of Value-at-Risk and Related Risk Measures.

Dean Marks*

February 2011

Submitted in partial fulfilment of the Masters of Philosophy degree in Mathematical Finance.

University of Cape Town

Supervised by Prof. Ronald Becker

*The author thanks Prof. Ronald Becker for supervising this thesis.

Plagiarism Declaration

I, Dean Marks, the author of this thesis, understand the meaning of plagiarism and acknowledge that it is morally wrong and prohibited by UCT. I declare that this thesis is free of plagiarism and that all sources used in this thesis are properly cited and referenced.

Date:

Name:

Signature:

In memory of Professor Graeme West.

Abstract

Nested Monte Carlo is a computationally expensive exercise. The main contributions we present in this thesis are the formulation of *efficient* algorithms to perform nested Monte Carlo for the estimation of Value-at-Risk and Expected-Tail-Loss. The algorithms are designed to take advantage of multiprocessing computer architecture by performing computational tasks in parallel. Through numerical experiments we show that our algorithms can improve efficiency in the sense of reducing mean-squared error.

Contents

1 Introduction

The Monte Carlo approach to calculating value-at-risk (VaR), and other portfolio risk-measures, is an attractive numerical framework because it is easily applied to a variety of classes of stochastic models[1]. Analytical solutions are usually enlightening, but are often difficult to find; and in that case a Monte Carlo approach may be preferred.

The main contributions presented in this thesis are *efficient* algorithms to estimate VaR and Expected-Tail-Loss (ETL) in a nested Monte Carlo setting. In particular, the algorithms are implemented using parallel computing methodologies with efficient sampling based on analytical results by Broadie *et al.* (2010). We show, through numerical experimentation, that, subject to a fixed *computational budget*, our algorithms can reduce mean-squared error[2](MSE). Notwithstanding those promising results, we find that our implementations yield unsatisfactory running times. We therefore treat these implementations as proofs-of-concept, rather than definitive guides.

Our thesis progresses as follows. We present three common portfolio risk-measures, namely, Probability-of-Loss (PoL), Value-at-Risk and Expected-Tail-Loss and explain how these may be approximated using Monte Carlo methods.

We then provide a literature review on the topic of risk-measurement and Monte Carlo methods. In the literature review we focus on a paper by Broadie, Du and Moallemi (2010) in which they develop non-uniform sampling algorithms. Their algorithms are more optimal in the sense that, for a given computational budget, the MSE they achieve are substantially lower than that of other algorithms predating them.

In the literature review we also discuss qualitative appraisals of VaR models in practise. In particular, VaR models as applied in the Basel II framework.

Next we formulate extensions to the Broadie, Du and Moallemi algorithm allowing one to estimate VaR and ETL. We then present empirical results for several relevant applications of the algorithm. For comparison, we compare

[1]Many examples are given in the literature. Alexander (2010) provides a survey of a range of possible stochastic models.

[2]MSE is a measure often used to assess the quality of an estimator. It is defined: $MSE = \mathrm{E}[(\hat{\theta} - \theta)^2]$ where $\hat{\theta}$ is the (random) estimator and θ is the true value being estimated. A single Monte Carlo experiment yields one estimate. To calculate MSE the experiment must be repeated many times.

the algorithm to simpler methods. In specifying the modified algorithm we place particular emphasis on parallel-computing.

2 Risk Measure Definitions

Risk measures are important tools used by sophisticated investors for risk budgeting and risk reduction. There are a number of risk measures in common use. We summarise the related risk measures, PoL, VaR and ETL below.

2.1 Probability-of-Loss

The probability of loss given a shortfall / loss-threshold c and an investment horizon h is defined as:

$$\alpha = \Pr[V_0 - V_h > c]$$

where V_h is the portfolio value at time h. [3]

2.2 Value-at-Risk

Value-at-Risk x, given a significance of $\gamma 100\%$ and an investment horizon h is defined as:

$$x = F^{-1}(\gamma)$$

where $F(.)$ is the real-world CDF of the *loss distribution:*

$$L = V_0 - V_h,$$

i.e. L is the decrease in portfolio value (a *loss*) from $t = 0$ to $t = h$ which is stochastic and unknown at $t = 0$. A realisation of L that is negative corresponds to a profit over the investment horizon, since in that case $V_h > V_0$.

[3] All probabilities and expectations in this section are under the real-world probability measure. The value of the portfolio today, V_0, is presumed known and non-stochastic in the probability space used.

Implicitly VaR is the value of x that solves:

$$\begin{aligned} \gamma &= \Pr[V_0 - V_h \leq x] \\ \iff 1-\gamma &= \Pr[V_0 - V_h > x] \end{aligned}$$

and so one can easily see the link between VaR and PoL. Indeed VaR can be viewed as the inverse function of PoL.

In words, VaR is the γ100th percentile of the loss-distribution. Intuitively it says that the portfolio will lose at most x with a probability of γ over the horizon h.

2.3 Expected-Tail-Loss

Expected tail loss y is defined:

$$y = \mathrm{E}[V_0 - V_h | V_0 - V_h > c]$$

where c is the loss-threshold and h the investment horizon. Often c is set to some VaR estimate. It is relevant because it is a measure of the expected severity of an extreme loss.

All of the above measures are dependent on the loss-distribution. Since the true distribution is not known, a model must be specified by the practitioner. For example, models may assume that losses are determined by several underlying stochastic factors (i.e. a multi-factor model). An alternate approach is to assume that the distribution of losses is time-stationary and calculate the measures based on historical losses (over the 250 most recent trading days, for example).

In order for the risk measures to be meaningful the assumptions must resemble reality. Indeed, they are all fully characterised by the loss-distribution, which in turn is fully characterised by the assumed model. The problem that arises, of course, is that it is often necessary to invoke strong assumptions in the model to maintain tractability - so that we may in fact derive a loss-distribution (either analytically or numerically) at all.

3 Monte Carlo Methods

A *Monte Carlo experiment* is a type of numerical approximation of an expectation or integral using a sample statistic on a random, psuedo-random

or quasi-random sample. Each element in the random sample is the outcome of a *trial*. Typically, a deterministic algorithm is used to generate a psuedo- or quasi-random sample that appears "sufficiently random[4]" to be suitable for the particular approximation in question.

If we can cast any quantity (which need not be driven by a stochastic process) as an expectation on a probability measure in which we are able to draw random samples, then the Strong Law of Large Numbers tells us how we may approximate this quantity using a Monte Carlo *experiment*:

Theorem 1. *Let $\{X_i\}_1^\infty$ be a random sample, i.e. it is an infinite sequence independently and identically distributed random variables, then:*

$$\Pr[\lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} X_i = \mathrm{E}[X]] = 1,$$

i.e. the sample average converges in probability to the theoretical expectation.

3.1 Probability-of-Loss

PoL can be easily cast as an expectation as:

$$\begin{aligned}\alpha = \Pr\{V_0 - V_h > c\} &= \int_c^\infty f_L(x)dx = \int_{-\infty}^{\infty} \mathbb{1}_{\{V_0-V_h>c\}} f_L(x)dx \\ &= \mathrm{E}[\mathbb{1}_{\{V_0-V_h>c\}}], \end{aligned} \tag{1}$$

were f_L is the density function of the loss-distribution.

Now, since we will characterise the loss-distribution we will be able to draw a random sample from the expression inside the expectation (a sequence of ones and zeros). Then, as a consequence of the Law of Large Numbers, we may use the sample average to find $\hat{\alpha}$, our PoL estimate.

Formally, define:

$$x_i = \begin{cases} 1 & \text{if} \quad \hat{L}_i > c \\ 0 & \text{otherwise} \end{cases}$$

Then, by the Law of Large Numbers, our PoL estimate is

$$\hat{\alpha} = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{2}$$

[4]See later section on generating random numbers.

where our sample size of loss estimates ($\hat{L}_i$) is n.

3.2 Value-at-Risk

VaR can be approximated by finding c in (1) such that α is the required level of significance using a root finding algorithm. Glasserman *et al.* (2000) summarise the Monte-Carlo a 3 step method for calculating VaR on a high-level. We paraphrase the procedure:

1. Generate a size n random sample of time h risk-factors constituting n time h scenarios.
2. Use the n scenarios to arrive at n time h portfolio values, yielding n realisations from the loss-distribution.
3. Calculate the proportion of values in the random sample of losses that exceed c for several values of c. With those values, use some interpolation scheme to arrive at the γ100th percentile; the approximation to the h-day γ100th VaR, as required.

3.3 Expected-Tail-Loss

Since the loss threshold is a fixed quantity c, the density function of portfolio loss conditional on the loss exceeding c can be cast as an unconditional distribution on another random variable, Y say, that possesses the same distribution as the conditional distribution of $V_0 - V_h|(V_0 - V_h > c)$. In particular, Y is characterised by the CDF:

$$F_Y(y) = \frac{\int_c^y f_L(x)dx}{\int_c^\infty f_L(x)dx},$$

where f_L is the density function of the loss distribution.

The random variable Y will have unconditional moments equal to the conditional moments of $V_0 - V_h$. The ETL is then E[Y], which, by the Strong Law of Large Numbers can be estimated by the sample average of Y realisations.

3.4 Random Samples

In performing Monte Carlo experiments we are faced with the problem of generating so-called random numbers (Alexander, 2009:203). Sobol (1998) states that *true randomness* is a mathematical construct not found in nature.

The practical definition of randomness differs between applications. For instance, in some instances a sample that has no easily discernable pattern may be "good enough". In another case one might require a sequence of random numbers derived from a physical process.

For example, random number generation for use in computer security should be "hard-to-guess" (Barker and Kelsey, 2007), a property termed entropy in Barker and Kelsey (2007). In that case a physical phenomenon may be used as an input to the random number generator (Barker and Kelsey, 2007).

Ultimately we may deem our random number generator *good enough* if we can establish that it possesses the requisite properties to adequately achieve our goal.

A useful re-definition of a uniformly distributed sequence attributed to Weyl and presented in Sobol (1998) and stated here in the uni-dimensional case:

A sequence $\{x_i\}_1^\infty$ with $x_i \in [0,1] \forall i$ is said to be uniformly distributed if, for any bounded Riemann-integrable function g:

$$\frac{1}{N}\sum_{i=1}^{N} g(x_i) \to \int_0^1 g(x)dx$$

when $N \to \infty$. The definition in the multi-dimensional case is analogous.

The above definition says nothing about the randomness of a uniformly distributed sequence either in the intuitive sense or the regarding the entropy it possesses. The definition is useful because it gives us far more freedom in choosing a uniformly distributed sequence, potentially achieving faster convergence in Monte Carlo experiments. The convergence properties are far more important to us than the entropy of a random number generator. For instance, we would like to choose a sequence that exhibits no clustering. Clustering is an undesirable property that could greatly inhibit the speed of convergence because of "holes" in the sampling.

Sobol (1998) says that the main advantage of quasi-random numbers is

the ability to achieve better convergence. Several types of quasi-random numbers are known, such as the Sobol sequence. Therefore, the use of quasi-random numbers is preferred. Monte Carlo methods that use quasi-random samples are often referred to as quasi-Monte Carlo.

There is a technicality concerning quasi-random numbers. Unlike pseudo random numbers, quasi random numbers cannot be generated one-by-one (Sobol, 1998); they must be generated in a batch (Sobol, 1998). We therefore need to know precisely the number of elements in the sequence we require at the start. For this reason quasi-random numbers cannot be used in the algorithm of Broadie *et al.* (2010) (described in section 4.1); a drawback of their method.

3.5 Risk-Neutral Monte Carlo

Risk-neutral Monte Carlo experiments are ones that generate trials on an Equivalent Martingale Measure (often called the risk-neutral measure).

The risk-neutral measure is important in finance because it can be used to find a fair price for derivative securities. The price of a derivative security can be shown to be:

$$V_0 = \mathrm{E}^{\mathbb{Q}}[D_T \phi(S_T)],$$

where T is maturity, D_T is the (possibly stochastic discount factor), $\phi()$ is the payoff function of the security at maturity and S_T is the value of the underlying security or securities at time T and $\mathbb{Q}$ is the risk-neutral measure[5].

Risk-neutral Monte Carlo is simply the application of the above result, performed by generating trials under the risk-neutral measure. The inner Monte Carlo experiments discussed later are risk-neutral.

4 Literature Review

4.1 Efficient Risk Estimation via Nested Sequential Simulation. Broadie, Du and Moallemi (2010)

Broadie *et al.* (2010) formulate three new Monte Carlo algorithms to effi-

[5] $\mathbb{Q}$ is a measure defined such that any security multiplied by the discount factor (the inverse of the numéraire) is a martingale.

ciently calculate PoL. Their algorithms apply to the case where the pricing of the portfolio in question requires risk-neutral Monte Carlo.

That is to say that for every scenario generated by the real-world "outer" Monte Carlo, an "inner" risk-neutral Monte Carlo experiment is performed. This is a relevant tool (known as nested Monte Carlo) for PoL estimation on portfolios containing one of a wide range of common derivatives that require risk-neutral Monte Carlo pricing.

The outer Monte Carlo experiment is performed under the real-world measure in order to estimate a real-world PoL[6]. Each *trial* in the outer Monte Carlo will generate a time h scenario (usually the scenario is specified by, *inter alia*, the time-h prices of the underlying securities).

For example, in the case of a single European call option on a stock the outer Monte Carlo would typically generate a time-h real-world stock price termed a *scenario*. The *scenario* is then used as a parameter into a risk-neutral inner Monte Carlo pricing as if we were standing at time h.

A problem with nested Monte Carlo is that it has quadratic complexity which motivates the goal of reducing the MSE of the nested Monte Carlo experiment subject to the constraint of a fixed number of aggregate trials, which Broadie *et al.* (2010) call the *computational budget*.

Before describing the algorithm of Broadie *et al.* (2010), we describe a simpler method known as uniform nested Monte Carlo sampling presented in Broadie *et al.* (2010) as a means of comparison with their own. The uniform sampling method specifies two parameters upfront, n and m, which are the number of *inner* Monte Carlo *trials* and *outer* Monte Carlo trials respectively. That is to say that n scenarios are generated and nm inner trials are performed. On completion, a collection of n estimates of realisations from the loss-distribution are obtained. To get the PoL estimate, (2) is applied.

Broadie *et al.* (2010) find that it is possible to reduce the MSE by sampling non-uniformly. To do so they provide 3 alternate approaches. One of which, which they term *Sequential Sampling*, is described as follows:

Call the sampling of a single *trial* from a particular *inner* Monte Carlo experiment a *step*. In order for a *step* to be performed a particular inner Monte Carlo must be selected. Broadie *et al.* (2010) use the rule described in the next paragraph to select an *inner* Monte Carlo experiment at each *step*. By

[6]In the case of importance sampling the measure may be cleverly changed to provide better convergence (Alexander, 2009:217)

using their rule as a selection criteria before each *step* is performed, they maximise, myopically (i.e. for each *step*), the probability of the step improving the PoL estimate (Broadie *et al.*, 2010).

A key result derived analytically in Broadie *et al.* (2010) is the following selection rule:

Let i be a particular scenario. The quantities m_i, $\hat{L}_i$ and σ_i are, respectively, the current values of the number of trials, loss estimate and standard deviation of the loss estimate of the inner Monte Carlo associated with scenario i. We select for sampling the inner Monte Carlo associated with scenario i^* where

$$i^* \in \underset{i}{\operatorname{argmin}} \frac{m_i}{\sigma_i} |\hat{L}_i - c|, \tag{3}$$

which they show to be an efficient selection rule.

The logic behind their selection criteria is based on the fact that the PoL is only determined by the number of losses to the right of the loss threshold relative to the number of losses to the left of it. In other words, once that piece of information is known, the magnitude of the loss is irrelevant. They demonstrate that, as a result of rule (3), *inner trial* sampling is focused on scenarios with loss estimates near the loss-threshold. That is to say, where the certainty regarding a scenario's precise loss is most important.

They demonstrate numerically that once the algorithm is completed, m_i peaks sharply with values of $\hat{L}_i$ closest to the loss-threshold.

The crucial difference compared to uniform sampling is that uniform sampling has m_i a constant, i.e. $m_i = m$ $(\forall i)$. In addition to their analytical proof, they find, upon numerical experimentation, that their non-uniform sampling algorithms yield a substantial reduction in MSE for a fixed computational budget.

The other two algorithms they provide are variations on a theme.

The first of which is their *Threshold* algorithm which is a simplified approach which modifies rule (3) so that the $\hat{L}_i$ realisations are i.i.d. They do so by modifying the rule so that it is a stopping rule rather than a selection rule. The purpose of the *Threshold* algorithm is to simplify their analytical exposition.

The third alternative, which they term the *Adaptive* algorithm, actually allocates the computational budget between generating outer scenarios and performing inner trials in an online fashion.

As part of their empirical study they consider 2 portfolios (Gaussian[7]and a European put), 3 loss thresholds corresponding to PoLs of 10%, 1% and 0.1% and 5 different algorithms all with a computational budget of approximately four million trials. They find that in all cases their methods yield much lower MSEs compared to uniform sampling. In their best (worst) case for the Adaptive algorithm the MSE was reduced by a factor of approximately 7 (2) compared to optimal uniform sampling.

For the purpose of our extension we consider their *Sequential* algorithm only.

While their method also has quadratic complexity, Broadie *et al.* (2010) show that the sample size required to achieve comparable mean-squared error is lower and thus less computationally expensive.

There is a technicality that should be mentioned. The parameters $\bar{m}$ and n affect the optimality of the algorithm[8]. In Broadie *et al.* (2010) they address this by, *inter alia*, a brute-force numerical optimisation (i.e. by varying the values of $\bar{m}$ and n). However, the Adaptive algorithm addresses this issue by reallocating the computational between $\bar{m}$ and n in an online fashion.

They find that the Adaptive algorithm performs almost as well as the optimal Sequential, which is reassuring.

4.2 Efficient Monte Carlo Methods for Value-at-Risk, Glasserman, Heidelberger, Shahabuddin (2000)

(Glasserman *et al.*, 2000) contrast two approaches to calculating VaR. The first approach makes the approximation that portfolio values are linear or quadratic in a small number of risk factors (Glasserman *et al.*, 2000). For example, the so-called delta-gamma approximation, which is essentially a second-order Taylor approximation of a portfolios value with respect to the risk-factors (Glasserman *et al.*, 2000).

Computation using this approximation is cheap but unsatisfactory because VaR, which focuses on outcomes in the right tail of the loss distribution, is determined by large movements in risk-factors, i.e. where a second order Taylor approximation comes with a large approximation error.

The second approach reviewed in Glasserman *et al.* (2000) is Monte Carlo

[7]The Gaussian portfolio is a toy example desribed in a later section.

[8]For the Uniform algorithm $\bar{m} = m$

simulation. Their argument is that the second approach provides a large enough improvement in accuracy over the first that it warrants research into reducing its onerous computational cost (Glasserman *et al.*, 2000).

In order to achieve reduced computational cost of the second approach, they examine a number of ways the first approach can be used to inform the Monte Carlo sampling procedure. In particular, they examine "importance-sampling and stratified-sampling based on the delta-gamma approximation" (Glasserman *et al.*, 2000).

Importance sampling is a method that is used to achieve improved accuracy of PoL estimates by casting them as expectations on a changed measure. The measure change is chosen to minimise the variance of the estimator. The technique results in a larger proportion of the loss realisations exceeding the PoL threshold (Glasserman *et al.*, 2000); providing an intuitive interpretation for the reason the change-of-measure is more informative. Realisations of the Radon-Nikodym derivative are used to weight the loss outcomes under the new measure in order to evaluate the PoL (Glasserman *et al.*, 2000), which is the expectation under the original (real-world) measure.

Glasserman *et al.* (2000) establish an importance sampling scheme in the context of PoL estimation in a Monte Carlo experiment using the delta-gamma approximation explained in the next paragraph. In order to do so, they assume that (outer) scenarios are characterised by several correlated Gaussian risk-factors, ΔS. In addition they assume the availability of the partial derivative vector (δ) and the second partial derivative matrix (Γ) of portfolio value with respect to the risk-factors as well as Θ, the partial derivative of the portfolio value with respect to time.

The delta-gamma approximation they state is:

$$L \approx -\Theta\Delta t - \delta\Delta S - \frac{1}{2}\Delta S\Gamma\Delta S$$

Note that ΔS is implicitly a function of Δt.

They perform a decomposition of ΔS into a vector of uncorrelated standard normal variables Z. The importance sampling is achieved by changing the measure so that Z are carefully chosen (correlated) Gaussian random variables. The way in which they choose the particular distribution such that it maximises the probability that the delta-gamma approximation to the portfolio loss exceeds the loss-threshold.

With the above at hand they simply perform Monte Carlo experiments as usual except that they generate Z under the changed measure, retrieve

ΔS by a transformation of Z and adjust the outcome of each inner trial by multiplying it with a realisation of the correct Radon-Nikodym derivative.

In conjunction with the above, they also implement stratified sampling to achieve a further reduction in estimator variance. The distribution of delta-gamma approximation of losses (recall it is a function of Z) is stratified into n equi-probable non-overlapping strata over the full support of the distribution (under the changed measure). The sample approximate loss realisations (under the changed measure) is then constructed such that each stratum holds $\frac{k}{n}$ samples of the delta-gamma approximation, where k is the total number of scenarios. In this way the overall sample provides a better covering of the distribution and hence a more accurate estimate of the PoL.

Glasserman *et al.* (2000) report the results of their numerical experiments to illustrate the reduction in variance of the VaR estimates achieved by their importance- and stratified-sampling methods. They find that applied to several different portfolios consisting of numerous vanilla European options on several underlying assets, that the importance sampling technique with stratified sampling yields a reduction in computational time by amounts between 28 and 327 fold.

An obvious drawback of this method is that it requires the knowledge of the *Greeks* of the portfolio, i.e. the partial derivatives which may not be available.

4.3 Basel Committee on Banking Supervision (2004), International Convergence of Capital Measurement and Capital Standards

The Basel Committee on Banking Supervision (2004) outline the framework for calculating minimum capital adequacy requirements (CAR) known as Basel II. The stated goal is to "strengthen the soundness and stability of the international banking system" (Basel II, 2004).

The key difference between the Basel II accord and its predecessor is the increased focus on the relationship between credit, market and operational risk and minimum CAR (Basel II, 2004). They state that there is an increased focus on both expected and unexpected loss in the Basel II accord.

A key element of the CAR calculation is the Internal Ratings Based (IRB) approach. As part of the IRB they place a particular emphasis on the use of

VaR and internal VaR models. They give a substantial amount of freedom to banks to use any of a wide range of VaR models. In particular they single out three classes of VaR models that they deem appropriate. The three model classes are historical, Monte Carlo, and variance-covariance VaR models.

They state that the 99th percentile of the distribution of quartely excess returns (over the risk-free rate) must be used for the calculation of CAR.

Clear problems arise from the prescribed calculation and use of the VaR estimate. The first problem we identify is the freedom to use variance-covariance models. The use of such models implies that the excess return distribution can be satisfactorily modelled using elliptical distributions (e.g. Gaussian and student-t). It is widely accepted that the level of risk implied by these distributions is grossly underestimated.

Historical VaR models use past history of returns to predict future returns. Our criticism is that structural breaks, for instance during a crash, render historical models invalid, because, virtually by definition, predictions using historical data become far more problematic - i.e. an example of an unexpected event that the accord claims to guard against.

Another key issue with the prescribed VaR approach is the use of the single 99% VaR point estimate. Point-estimates have long been a taboo among statisticians, because they ignore critical information available in higher-moments. The use of the VaR point-estimate alone detracts, in our view, from the spirit of statistical models such as VaR models.

In our view, the Monte Carlo VaR model is superior to the other two because, it gives the modeller freedom to formulate realistic models and the ability to calculate related risk-measures and higher moments of the distribution of tail losses. Therefore, it is a more informative approach.

4.4 Danielsson *et al* (2001), An Academic Response to Basel II

Danielsson *et al.* (2001) posit a strong argument highlighting the deficiency of VaR as a tool for calculating regulatory risk capital. Their argument was in response to the Basel II proposal at the time.

The central claim revolves around the idea that risk capital models internal to individual banks determine a banks risk-aversion. They claim that external regulation of these models (e.g. through the Basel II accord) in-

creases homogeneity of banks. They go on to say that banks will tend to trade in a synchronised fashion as a result of homogeneity of risk-aversion. As a result - they claim - the actions of individual banks will, on aggregate, amplify losses in the event of a market crash. In effect, synchronised trading will tend to evaporate liquidity during crisis.

They argue that VaR - a tool prescribed in the Basel II accord - promotes the above "procyclical" phenomenon (Danielsson *et al.*, 2001) .They claim that the proposed VaR methodology cannot capture "low-probability high-loss" (Danielsson *et al.*, 2001) events[9]for two reasons. Firstly, the proposed models - the use of elliptical distributions - have thin tails (Danielsson *et al.*, 2001). Secondly, a 99% VaR will be breached on average 2.5 times a year (Danielsson *et al.*, 2001). That is far more frequent than the event of a crash. The resultant risk capital is thus not sufficient in the event of a market crash.

A further criticism of the VaR metric they make is that it is not, in general, sub-additive. That is to say that the VaR quantities of segregated assets may understate the aggregate risk of these assets when viewed as a single portfolio (Danielsson *et al.*, 2001). This is because the sum of the segregated VaR quantities may be less than the VaR of the portfolio on aggregate (Danielsson *et al.*, 2001).

In our view, many of the above issues can, at least in part, be addressed by reducing model misspecification by, for example, relaxing unrealistic assumptions. It is widely agreed that loss distributions are asymmetrical and thick tailed. The high degree of homogeneity of risk-aversion among banks is far less of a problem when expectations about market risk do not change suddenly, which is more likely in an environment of wholesale model misspecification. In fact, even a low degree of risk-aversion homogeneity - as is likely the case, since banks tend to be operationally similar - would result in a comparable outcome when sudden changes in market expectations occur.

If well-specified VaR models are used it is likely that the impact of extreme outcomes would be better understood. For instance, once VaR can be accurately estimated at levels of significance substantially higher than 99%, the problem of incorrect expectations becomes less likely. The trouble, however, is that once this risk is known the problem that arises is the ability to hold sufficient risk capital. In our view, negative externalities arise when banks attempt to hold "adequate" risk capital on this basis. Banks ability to extend credit diminishes, impeding the real economy. As it turned out, the banking crisis of 2008 was more-so as a result of banks taking on far more

[9]precisely the type of events that the Basel II accord claims to guard against.

risk than they could guard against with adequate risk-capital. In hindsight, better models and sensible risk control measures may have been the correct safeguard.

5 Implementation of the modified algorithm

5.1 Algorithm Specification (VaR)

The new algorithm we propose is an adaptation of the Broadie *et al.* (2010) sequential algorithm. Their algorithm efficiently calculates PoL. In our adaptation one may calculate VaR with a substantial reduction in mean-squared error compared to the optimal uniform method. In addition, the algorithm runs in parallel, with the potential to greatly reduce computing time on computers with a multi-core processing architecture.

In Glasserman *et al.* (2000), the final step of their suggested recipe for calculating VaR using Monte Carlo is to calculate a number of PoL estimates by varying the loss threshold. With this sequence of PoL estimates they suggest that one interpolate over the loss thresholds as a function of the PoL estimates to estimate VaR. Our algorithm incorporates this idea in extending the Broadie *et al.* (2010) algorithm.

At the outset it is required that a number of loss thresholds be specified, say k of them. The number k may be conveniently chosen to be equal to one less than the number of processing cores available. The increasing sequence of loss thresholds, $\{L\}_1^k$, must be chosen in such a way that the final VaR estimate lies between the minimum and maximum of the sequence. Since the VaR estimate is unknown (it is the value we are trying to calculate), the practitioner may need to experiment by shifting L_1 and L_k to ensure that interpolation (the final step) can be done.

The algorithm is initialised by starting $k+1$ processes. One of the processes, call it the *main process*, creates k priority queues[10], each corresponding to an element in $\{L_i\}$. The remaining k processes, call them *worker processes*, are indistinguishable, each of which waits for a task to be dispatched by the *main process*. A *task* is a positive integer, i, less than or equal to the number of *scenarios* n. When a *worker process* receives the *task* i, it samples one *trial* from the Monte Carlo *experiment* associated with the ith *scenario* and returns it to the *main process*.

The above scheme may result in much shorter runtimes on multi-core processors. The crux of the algorithm is determining the sequence of *tasks* sent to the *worker processes*. Note that it is not important which worker process receives a particular task; recall that they are indistinguishable. In fact, the mechanism by which the *main process* dispatches tasks is with a single first-in-first-out (FIFO) queue shared between all processes. The *worker processes* simply pop tasks off the shared queue one at a time, execute them and return the results to the *main process* using another shared FIFO queue.

The selection sequence is performed in the *main process*. The *main process* pops one *task* from each of the k priority queues, deleting any duplicates, and dispatches the remaining *tasks* in a batch. It then waits to receive the updated loss estimates from all of the dispatched *tasks*. Finally, it recalculates the *task* priorities[11]at each of the loss-thresholds in $\{L_i\}$ and pushes each of them onto their corresponding priority queues. This procedure is repeated until the full computational budget is utilised.

In this way, k PoL estimates are calculated (using (2) with efficient sampling and a shared dataset.

Finally, interpolation over the loss thresholds - as a function of the PoL estimates - is performed to arrive at the require VaR estimate.

The following pseudo-code summarises our Modified Sequential algorithm:

[11]The priority is calculated using the expression to the right of the argmin operator in rule (3).

`ModifiedSequential`(*n*, $\bar{m}$, m_0, *PoLThresholds, significance*)

Scenarios = Generate n scenarios.
q = Number of elements in PoLThresholds
Create q PriorityQueues: priorityQueues
Create variable to store sequence $\{\hat{L}_i\}_1^n$ Create variable to store sequence $\{\hat{\sigma}_i\}_1^n$
Create variable to store sequence $\{m_i\}_1^n$

foreach *Scenario in Scenarios* **do**
    $(\hat{L}, \hat{\sigma}, m_i)$ = SampleInnerTrials(m_0, Scenario) Update sequences.
    foreach *PoLThreshold in PoLThresholds* **do**
        p = CalculatePriority($(\hat{L}, \hat{\sigma}, m_i)$, PoLThreshold)
        priorityqueue = Priority queue corresponding to PoLThreshold
        Push (p, Index of Scenario) onto priorityqueue
        i = Index of Scenario
        Update sequences.
    end
end
Continued on the next page.

Create FIFO Queue: queueIn To send tasks to worker processes.
Create FIFO Qeueue: queueOut To get results from worker processes.

Spawn q worker processes

while $\sum_{i=1}^{n} m_i < \bar{m}n$ **do**
    Tasks = $\emptyset$
    foreach *priorityQueue in priorityQueues* **do**
        Task = Pop task from priorityQueue
        if *Task* $\notin$ *Tasks* **then**
            Push Task onto queueIn
            Tasks = Tasks $\cup$ {Task}
        end
    end

    k = Number of elements in Tasks
    for j = *1 to* k **do**
        Wait for queueOut to be non-empty
        (Scenario Index, $\hat{L}$, $\hat{\sigma}, m_i$) = Pop result from queueOut

        foreach *PoLThreshold in PoLThresholds* **do**
            p = CalculatePriority(($\hat{L}, \hat{\sigma}, m_i$),PoLThreshold)
            priorityQueue = Priority queue corresponding to PoLThreshold
            Push (p, Scenario Index) onto priorityQueue
        end
    end
end

for k = *1 to* q **do**
    PoL_k = Proportion of $\hat{L}_i$ estimates > PoLThreshold$_k$
end

VaREstimate = Interp($\{PoL_i\}_1^q$, PoLThresholds, 1− significance)
return *VaREstimate*

5.2 Future Enhancement

In order to reap improved MSE reduction the following items may prove beneficial:

- The current sampling selection method maximises the probability of

one of the k PoL estimate improving in sequence, due to the work of Broadie *et al.* (2010). In future a method to jointly maximise the probability of the sequence of PoL estimates improving simultaneously, rather than each element in the sequence in isolation, could be found.

- Further research into the best interpolation method may improve VaR estimates.
- Calculation of the correct measure-change would allow for importance-sampling in both the inner and outer Monte Carlo experiments.
- Stratified sampling could be performed on outer Monte Carlo experiments. However, its use on the inner Monte Carlo experiments would prove difficult due to the sequential sampling in the Modified Sequential algorithm.
- Outer Monte Carlo experiments could sample from a quasi-random sequence.
- A method akin to the Adaptive Algorithm due to Broadie *et al.* (2010) would optimise the allocation of the computational budget between inner and outer sampling.

5.3 Algorithm Specification (ETL)

Our algorithm for computing ETL is a two-step procedure. We assume that the loss-threshold c is given[12]. Using this loss threshold we run the unmodified sequential Broadie *et al.* (2010) algorithm as if it were calculating a PoL with loss-threshold c. Instead of using the resultant (non-uniformly sampled) loss distribution to return the PoL statistic, we use the loss sample as input into the second step of the procedure.

We make the assumption that the scenarios with losses close to c are accurately estimated in the first step (the crux of the Broadie *et al* (2010) algorithm). With that assumption we justify discarding all scenarios yielding loss estimates below c. After discarding those scenarios we sample uniformly from the remaining scenarios to derive a loss sample in the tail of the distribution. The ETL is then reported by calculating the sample average.

The pseudo-code below outlines the second step of the algorithm, taking as input the final number of trials in each inner Monte Carlo experiment (m),

[12]If a VaR estimate is to be used, it is calculated separately.

the n scenarios (Scenarios), the loss-sample computed in step 1 ($\{\hat{L}_i\}_1^n$), the number of trials carried out so far in each inner Monte Carlo ($\{m_i\}_1^n$) and the loss-threshold (c):

Input: m, Scenarios, $\{\hat{L}_i\}_1^n$, $\{m_i\}_1^n$, c

$\{\hat{K}_i\} = \{x : x \in \{\hat{L}_i\}_1^n, x \geq c\}$

for $i = 1$ *to* n **do**
  if $\hat{L}_i < c$ **then**
    Mark Scenario$_i$ for deletion.
  end
end
Delete scenarios marked for deletion.

$p =$ Number of elements in $\{\hat{K}\}$
foreach $i = 1$ *to* p **do**
  Update K_i by generating $(m - m_i)^+$ trials from inner Monte Carlo i
end

$\text{ETL} = \frac{1}{p}\sum_{i=1}^{p} K_i$
return *ETL*

5.4 Priority Queues

The *Modified Sequential* algorithm requires that each *task* be assigned a priority (i.e. the value described in the algorithm specification above). The order in which inner Monte Carlo experiments are selected for *trial* sampling depends on these priorities, which are updated after each *step*. The highest priority *task* (i.e. the one that has the lowest priority value) at each *step* is selected.

The above could be achieved by sorting a conventional array, but would result in a negation of the reduced computational complexity. The alternative, as suggested in Broadie, *et al* (2010), is the use of a priority queue.

A priority queue is a special data-structure that allows for an efficient method to retrieve tasks in order of priority. A simple way to implement a priority queue is based on a heap (Ronngren and Ayani, 1997) which is a type of binary tree. A heap (technically a min-heap) has the property that, upon completion of any operation, no parent node has value greater than either

of its child nodes. When a node is inserted or removed, the heap property must always be restored. One such method to do so is described in Python Software Foundation (2011).

In implementing a priority queue a heap is convenient because the root node is always that node with minimum value. A pop operation can be executed efficiently by simply removing the root node. The main challenge, however, is the computational expense of restoring the heap property when each operation is performed. In implementing our algorithms a builtin Python heap is used. The computational complexity of the maintenance of the heap property in Pythons heap implementation is $O(nlog(n))$ (Python Software Foundation, 2011). We are therefore wary of the performance cost for large n, i.e. when the number of tasks in the priority queue grows large.

The issue of the heap implementation of the priority queue is that deleting nodes is non-trivial (Python Software Foundation, 2011). Because of this it is usual that nodes one wishes to remove are marked as deleted rather than actually removed (Python Software Foundation, 2011). The main bottleneck in the *Modified Sequential* algorithm is that tasks are continually reprioritised as a result of maintaining multiple priority queues - in other words n grows large as nodes are marked deleted, but not removed. We have found that in order for the Modified Sequential algorithm to be practical we require a better priority queue implementation. In particular, we need a way to reprioritise nodes efficiently.

A tentative solution to the problem of efficient reprioritisation operations is to modify node priorities (breaking the heap structure) instead of marking them deleted. Python has a *heapify* function (which runs in linear time (Python Software Foundation, 2011)) which could be used to restore the heap structure.

Alternatively a range of priority queue implementations are known (see Ronngren and Ayani (1997)), which may provide better performance.

5.5 Random Number Generation

The *SciPy* module for Python was used to produce pseudo-random samples. *SciPy* has the ability to sample from a list of commonly used uni- and multivariate probability distributions.

5.6 Parallel-computing

The availability of cheap multi-core CPUs and GPUs has made parallel computing an important consideration in algorithm design. With the use of parallel algorithms, running time can be improved by as much as a factor of the reciprocal of the number of cores available compared to purely sequential[13]algorithms requiring similar computing resources.

Algorithms where the processing of a large data-set or a large number of data-sets (for example a set of Monte Carlo trial outcomes) are required that can be subdivided into task subsets that have no between-subset side-effects[14]are candidates for parallelisation, because each task subset can be computed in parallel on separate CPU cores.

Once the parallel tasks are completed, an algorithm will typically process the data-set as a whole. For instance, in nested Monte Carlo, the risk-measure can be calculated using the methods described in sections 3.1 to 3.3 once the entire data-set (i.e. of loss outcomes) is available.[15]

An advantage of the uniform sampling algorithm is that parallelisation is trivial. The nm inner Monte Carlo trials can be subdivided arbitrarily and processed in parallel because no side-effects exist (absent the random number generator). For example, if l is the number of cores available then we can simply subdivide the nm trials so that $l-1$ cores are allocated $[\frac{nm}{l}]$ trials to compute and the remaining core allocated $nm-(l-1)[\frac{nm}{l}]$. This scheme will result in a simulation duration improved by a factor of approximately $\frac{1}{l}$ compared to it running sequentially.

6 Examples

In this section we present 3 example applications of the *Modified Sequential* algorithm, each pertaining to a different portfolio. Examples 1 and 2 are a European put, and the "Gaussian" portfolio (explained in detail below). Those example use the precise setup as those used in Broadie *et al.* (2010). They are simple (albeit unrealistic) examples which allow straightforward

[13]Here this means that instructions are restricted to running in series, which is different to the meaning in the context of the Modified Sequential algorithm.

[14]Essentially what is meant here is that the outcome of a task has no impact on the outcome of tasks processed subsequently. In other words, the order does not matter, and tasks can in fact be executed simultaneously.

[15]This is referred to as reduction. Intermediate reduction steps may be run in parallel.

benchmarking of the *Modified Sequential* algorithm. Results are well presented in Broadie *et al.* (2010), so one can quickly compare their results with those presented in this thesis. Example 3 (a slightly more complicated example) is a European put with underlying equal to the average of a basket of 4 stocks.

6.1 Example 1 - European Put

In this example the portfolio in question is a vanilla European put on a stock. Our investment horizon is one-week (i.e. $h = \frac{1}{52}$). The quantity we wish to estimate using nested Monte Carlo is the 99% one-week VaR. We use the *Modified Sequential* algorithm and, for comparison the Optimal Uniform and Interpolated Optimal Uniform algorithms are also performed.

We use the Black-Scholes assumptions with the following parameters: $S_0 = 100$, $K = 95$, risk-free rate = 3%, real-world drift = 8%, volatility = 20%, T = $\frac{1}{4}$.

For the Modified Sequential algorithm the number of outer scenarios, n, is set to 8000 and average number of inner scenarios $\bar{m}$ is set to 500. The choice of these parameters is not optimised to yield the lowest MSE, therefore, improved results would likely be achieved by optimising them. The difficulty, however, is that the optimisation routine would require us to perform many nested Monte Carlo experiments. The computational budget is $nm = 8000 * 500 = 4000000$.

We may price the put using the usual Black-Scholes formula, we find that $V_0 = 1.669$. To calculate the MSE we would like to know the true 99% one-week VaR, i.e. x such that $\Pr[L > x] = 0.01$. Using a straightforward argument, Broadie *et al.* (2010) calculate the true 99% one-week VaR to be 1.221.

Each trial in the outer Monte Carlo experiment corresponds to a scenario specified by the stock price one week from now, S_h. Using the well-known solution to the SDE of a geometric Brownian motion, we generate outer Monte Carlo trials by substituting W in:

$$S_h = S_0 \exp\left(0.08 - \frac{1}{2}0.2^2\right)\frac{1}{52} + \sqrt{\frac{1}{52}}0.2W,$$

for draws from a standard normal distribution. In words, each trial of the outer Monte Carlo yields one realisation of S_h, which characterises a scenario.

Each trial in the inner Monte Carlo experiments correspond to a risk-neutral discounted terminal payoff given the corresponding S_h values (again calculated using the SDE solution but with drift 3% and time $\frac{1}{4} - \frac{1}{52}$. The time-$h$ values of the portfolio, V_h, under the different scenarios is then the average of the trial outcomes of the associated inner Monte Carlo experiments.

The minimum number of trials generated in each inner Monte Carlo experiment, m_0, is set to 10. In order to calculate a MSE we perform 200 repetitions of the nested Monte Carlo experiment.

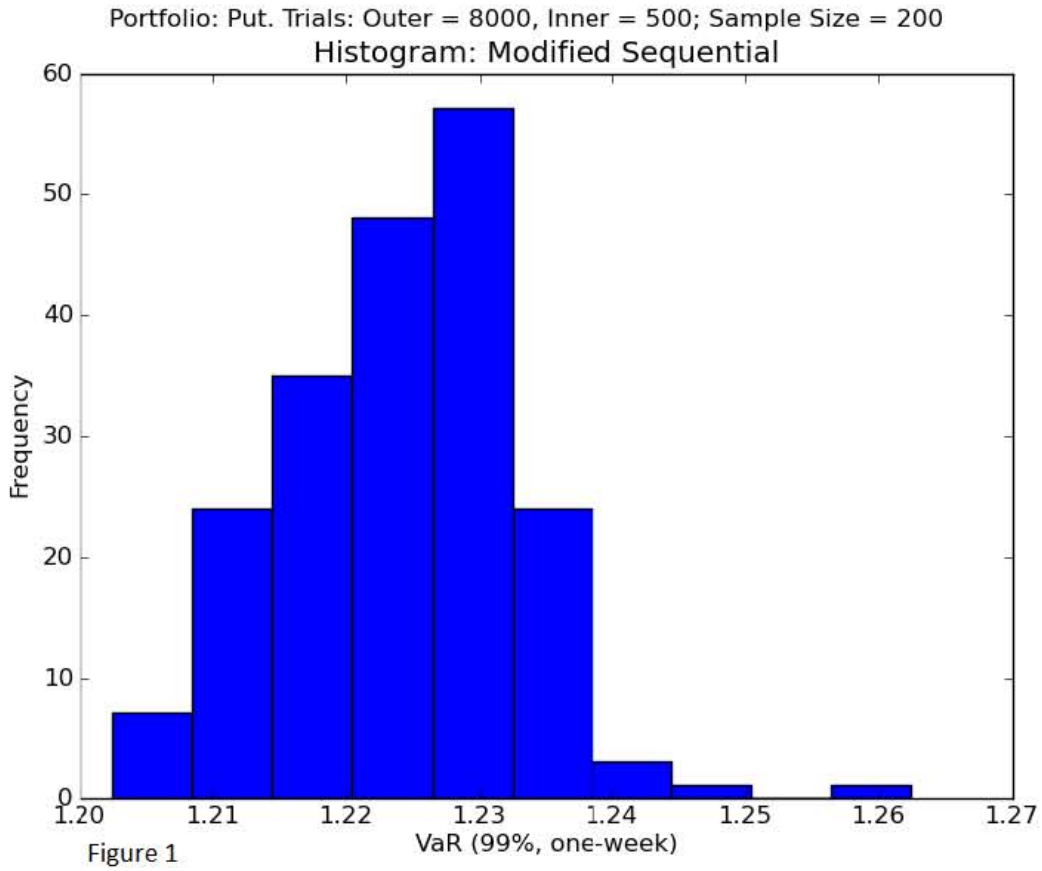
Our threshold guesses (i.e. $\{L\}_1^6$) are $\{1.0, 1.08, 1.15, 1.23, 1.31, 1.38\}$. This is a fairly tight sequence (that is to say that $L_{i+1} - L_i \approx 0.08$). The smaller the range of the sequence $\{L\}$ the lower the MSE is likely to be. It is suggested that the outcome of a run of the Uniform algorithm with a low computational budget be used as a mid-point for the threshold guesses.

A summary of the results yielded by repeating the Monte Carlo estimation 200 times is given below comparing the Modified Sequential algorithm to the Optimal Uniform and Interpolated Uniform algorithms.

Method	Mean	Std. Dev.	Bias	MSE
Modified Sequential	1.224	0.009	0.003	0.00009
Optimal Uniform	1.231	0.017	0.01	0.00040
Uniform Interp	1.243	0.028	0.022	0.00127

By the criterion of MSE and bias (i.e. the difference between the average of the VaR estimates and the true estimate), we see that the Modified Sequential algorithm performed the best and the Interpolated Optimal Uniform algorithm the worst. The ratio of the MSE values for Optimal Uniform to the Modified Sequential algorithms is roughly 4.5, which indicates a substantial improvement in the convergence of nested Monte Carlo.

Figures 1, 2 and 3 show histograms of the distributions of the nested Monte Carlo VaR estimators for the Modified Sequential, Optimal Uniform and Interpolated Optimal Uniform algorithms respectively:


Portfolio: Put. Trials: Outer = 8000, Inner = 500; Sample Size = 200
Histogram: Modified Sequential
Frequency
60
50
40
30
20
10
0
1.20
1.21
1.22
1.23
1.24
1.25
1.26
1.27
VaR (99%, one-week)
Figure 1

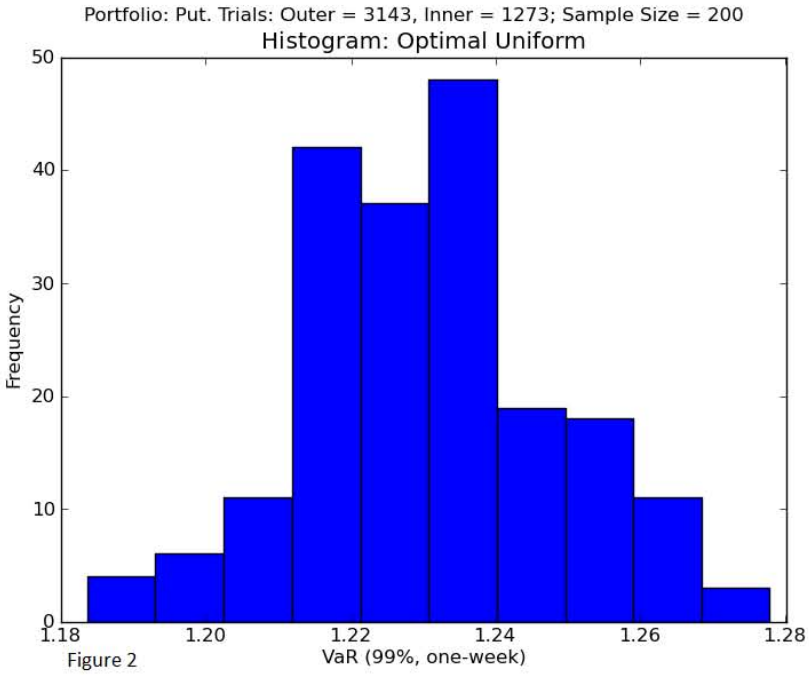


Figure 2

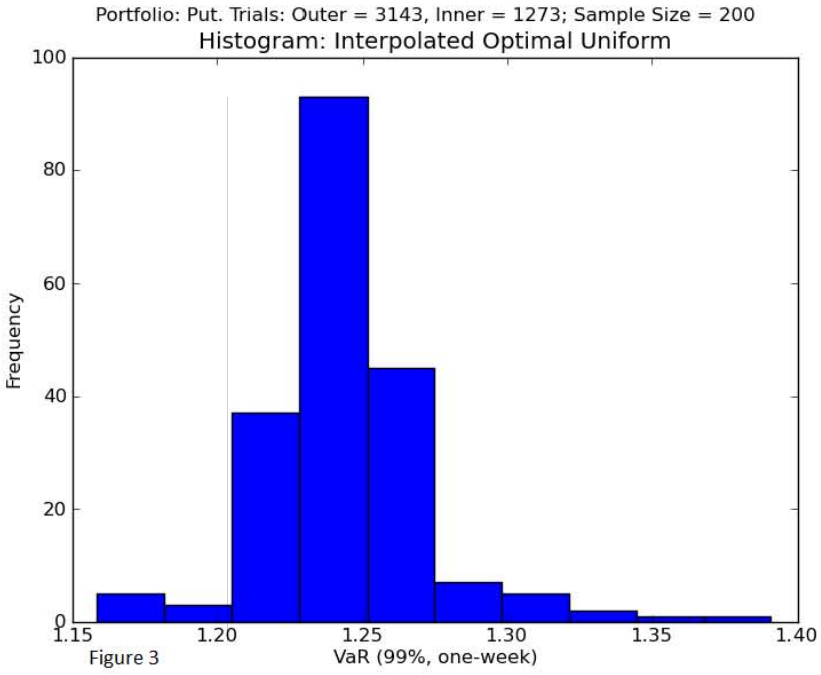


Figure 3

Upon performing the ETL algorithm with loss-threshold equal to 1.221 we find that the one-week ETL is *1.58*.

6.2 Example 2 - "Gaussian Portfolio"

The "Gaussian Portfolio" is a toy example to illustrate the method. Again $h = \frac{1}{52}$. As in example 1, we calculate the 99% one-week VaR and perform the same comparisons.

As outline in Broadie *et al.* (2010):

We define the "Gaussian Portfolio" as follows: We set $V_0 = 0$. Each trial in the outer Monte Carlo draws a single realisation from a standard normal distribution (call the realisation μ). We generate inner trials by taking a draw of a standard normal distribution, w, and setting the outcome of the inner trial to $\mu - 5w$. By the Central Limit Theorem $V_h = \mu$ (the average of the trials in an inner Monte Carlo Experiment). Thus, the true 99% one-week VaR is clearly the value of x such that:

$$\Pr[L(\mu) > x] = \Pr[V_0 - V_h(\mu) > x] = \Pr[-\mu > x] = 0.01.$$

Clearly, VaR $= x = \Phi^{-1}(0.99) \approx 2.326$, where Φ^{-1} is the inverse of the standard normal CDF.

A summary of the results yielded by repeating the Monte Carlo estimation 200 times is given below for their 3 algorithms, namely: The modified Sequential algorithm (specified below), the Optimal Uniform and the Interpolated Optimal Uniform algorithms.

Method	Mean	Std. Dev.	Bias	MSE
Modified Sequential	2.368	0.0516	0.0424	0.0045
Optimal Uniform	2.352	0.0480	0.0258	0.0030
Uniform Interp	2.408	0.0748	0.0817	0.0123

By the criterion of MSE and bias, we see that the Optimal Uniform algorithm performed the best and the Interpolated Optimal Uniform algorithm the worst. The ratio of the MSE values for Optimal Uniform and Modified Sequential algorithms is $\frac{2}{3}$, which indicates a moderate deterioration in the convergence of nested Monte Carlo.

Figures 4, 5 and 6 below show histograms summarising the VaR estimates of the 200 repetitions for the Modified Sequential, Optimal Uniform and Interpolated Optimal Uniform algorithms respectively:

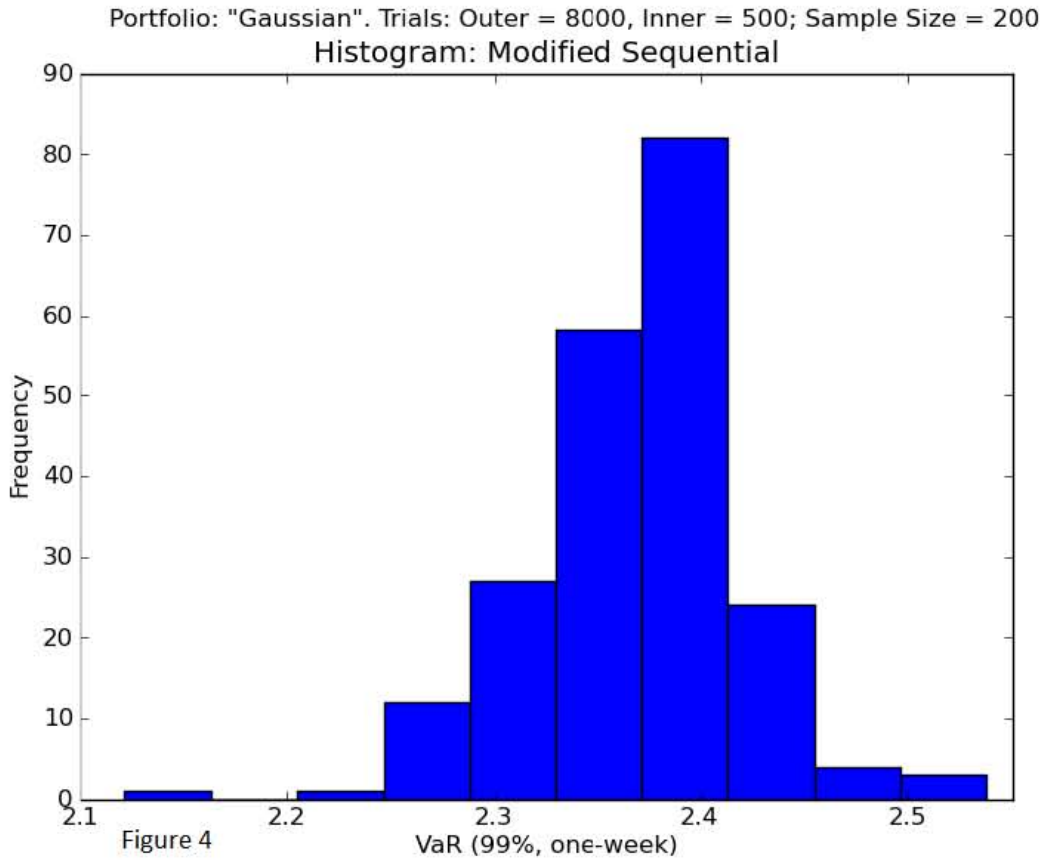

Portfolio: "Gaussian". Trials: Outer = 8000, Inner = 500; Sample Size = 200
Histogram: Modified Sequential
Frequency
0
10
20
30
40
50
60
70
80
90
2.1
2.2
2.3
2.4
2.5
VaR (99%, one-week)

Figure 4

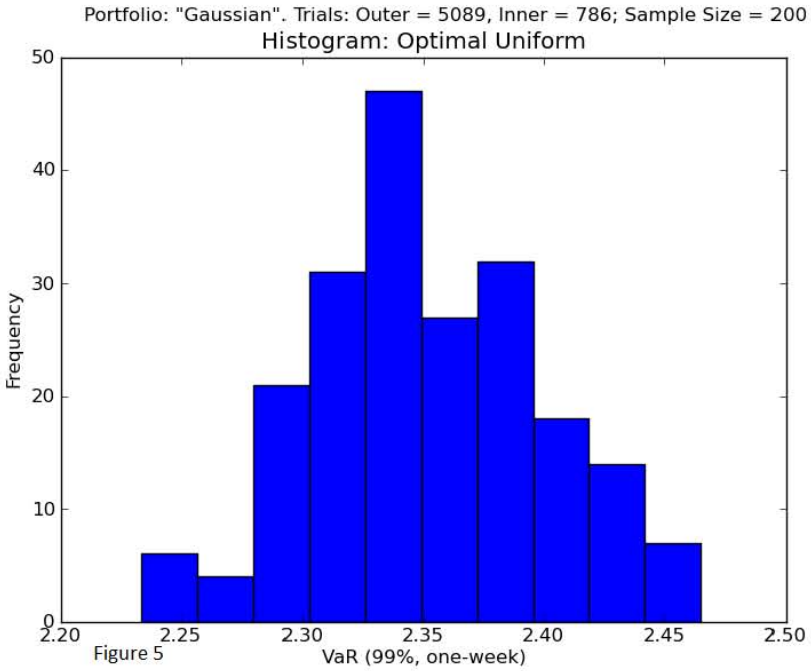


Figure 5

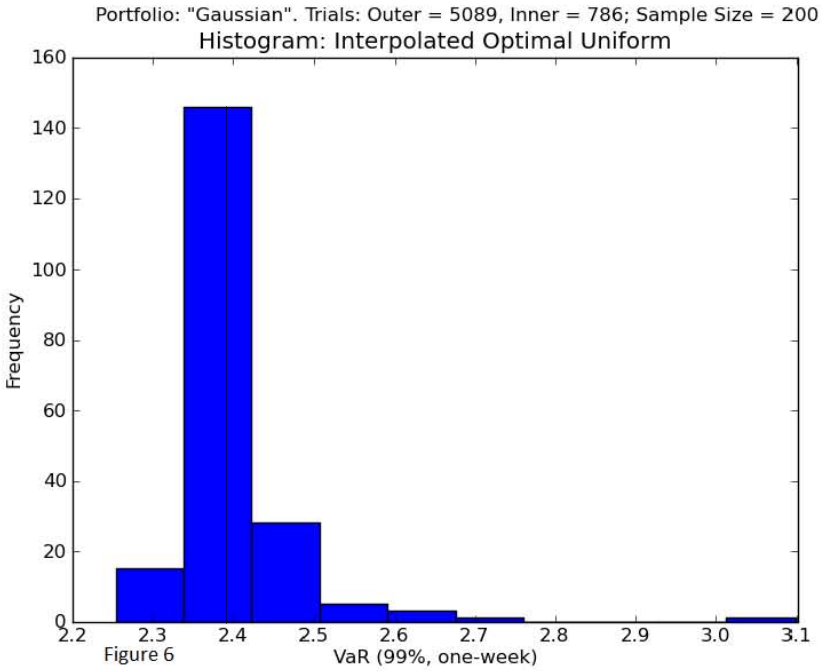


Figure 6

Upon performing the ETL algorithm with loss threshold equal to *2.326* we find that the one-week ETL is *3.23*.

6.3 Example 3 - Basket Option

The portfolio in question is one European put option on a basket of 4 stocks (i.e. a basket option. We assume that the stocks follow geometric Brownian motions. Our parameters are: $S_i = 100$ $(i = 1, 2, 3, 4)$, $K = 105$, $r = 3\%$, $T = \frac{1}{4}$ (the drift vector and convariance matrix are given below).

The model driving the loss distribution assumes that the only source of randomness is a 4-dimensional correlated geometric Brownian Motion which determines the evolution of the underlying stock prices.

In this example, a scenario is specified by the time-h stock-prices of the 4 constituents. Trials in the outer Monte Carlo experiment generate scenarios by drawing a realisation from the multi-variate Gaussian distribution with:

$$\mu = \begin{pmatrix} 0.08 \\ 0.06 \\ 0.09 \\ 0.05 \end{pmatrix}$$

and:

$$\mathbf{\Sigma} = \begin{pmatrix} 0.2 & 0.17 & 0.15 & 0.15 \end{pmatrix} \begin{pmatrix} 1 & 0.56 & 0.54 & 0.47 \\ 0.56 & 1 & 0.51 & 0.48 \\ 0.54 & 0.51 & 1 & 0.55 \\ 0.47 & 0.48 & 0.55 & 1 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.17 \\ 0.15 \\ 0.15 \end{pmatrix}$$

The matrix above happens to be the correlation matrix for the MSFT, AAPL, GOOG and ORCL counters[16].

The 4 time h stock prices were then calculated using the solution to the SDE of the geometric Brownian Motion as in Example 1.

The inner Monte Carlo trials are generated by again sampling from a multi-variate Gaussian distribution, but with $\mu = (0.03, 0.03, 0.03, 0.03)$. The 4 time-$T$ stock prices are then calculated as before but with the risk-free drift and the payoff calculated using the function ψ:

$$\psi(\bar{S}_T) = (K - \bar{S}_T)^+,$$

where $\bar{S}_T$ is the average of the terminal prices of the 4 stocks.

The parameters n, $\bar{m}$ and m_0 are again set to 8000, 500 and 10 respectively.

To find V_0 we perform a Monte Carlo risk-neutral pricing and find that it is *4.5707*. The risk-neutral pricing is essentially similar to performing one inner Monte Carlo experiment with scenario equal to the current stock prices and $h = 0$.

We find that when the nested Monte Carlo experiment is performed we obtain the 99% one-week VaR estimate of *3.71*

If we then perform our ETL algorithm with loss-threshold equal to *3.71*, we find that the ETL at that threshold is *3.91*.

7 Running Time

Of primary concern, in a pragmatic sense, is the total duration in seconds to perform the nested Monte Carlo experiment[17].

Notwithstanding the improved convergence of nested Monte Carlo using the *Modified Sequential* algorithm, we found that running time increased 13 fold compared to the Uniform Algorithm.

We attribute this reduction in performance to two issues. Firstly, the additional overhead that is introduced by maintaining multiple priority queues, which grow large as tasks are reprioritised. The Python run-time profiling output (provided in Appendix B) of the *Modified Sequential* algorithm clearly shows that a large proportion of run-time was spent performing priority queue operations. The second reason, which arises as a result of the first, is that *worker processes* are idle for substantial periods of time.

We do not treat the performance deterioration as a basis for a negative result, but rather as, by in large, an implementation issue. We posit that a more efficient priority queue implementation would address the first problem, and also, by implication, void the second.

Regarding optimising the *Modified Algorithm* itself, we posit that a sub-

[16]Retrieved from http://www.assetcorrelation.com/user/custom

[17]Although CPU-seconds or core-seconds may be more relevant if other applications or simulations are competing for the same CPU time - this is especially relevant when simulations are run on a shared server farm.

stantial performance benefit could be achieved by moving the priority queue operations onto (possibly multiple) processes rather than being executed on the *main process*

8 Conclusion

Despite the rich literature on the subject, the practical use of VaR has been heavily criticised. We have given examples of arguments (from the literature) against VaR as prescribed in Basel II. In response, we have argued the case for the need for robust VaR models, and in particular the need to be the able to efficiently perform nested Monte Carlo experiments.

Our main contributions are the formulation of algorithms to perform efficiently, in the context of parallel computing, nested Monte Carlo. Our numerical experiments show that our algorithms have the potential to improve the efficiency of the method.

In our analysis of the "mechanics" of our algorithm we have identified a number of avenues to achieve further variance reduction in our nested Monte Carlo framework. A serious bottleneck (i.e. the implementation of priority queues) is identified, which is, in our view, the most crucial issue to be addressed in order to reap the performance benefits of parallel computing.

9 Glossary

9.1 Glossary of symbols

The symbols below have the following meanings unless stated otherwise.

V_t - The value of a portfolio at time t.

h - An investment horizon.

L - The portfolio loss over the investment horizon $(V_0 - V_h)$.

L_i - The portfolio loss given that the state of the world at time h is governed by scenario i.

$\hat{L}_i$ - An estimate of L_i

n - The number of scenarios in a nested Monte Carlo experiment. Equivalently, the number of outer trials.

m_i - The number of inner trials generated under scenario i. That is to say, the size of the sample used to compute $\hat{L}_i$

$\bar{m}$ - $\frac{1}{n}\sum_{i=1}^{n} m_i$

c - A loss threshold.

γ - A level of significance.

α - The Probability of Loss.

$\hat{\alpha}$ - An estimate of α.

σ_i - The standard deviation of the estimator $\hat{L}_i$.

k - The number of worker processes, also equal to the number of priority queues.

9.2 Glossary of Terms

First-in First-out (FIFO) Queue - A data-structure that has insert and remove operations. Its key property is that the sequence in which elements are removed is identical to the order in which they were inserted.

Priority Queue - A data-structure with insert and remove operations. Elements are inserted with an associated priority (usually a numeric value). Its key property is that whenever a removal is performed it is the element with the lowest priority value that is returned.

Push - To an insert an element into a data-structure

Pop - Perform a removal operation from a data-structure, and retrieve the value of the element removed.

Parallel computing - A (now common-place) computing paradigm that allows execution of multiple program instructions simultaneously.

Parallel algorithm - An algorithm that uses the parallel computing paradigm.

Process - A mechanism provided by a computer Operating System to execute self contained programs (or parts of programs). Typically, processes do not share memory but may communicate with each other to achieve a single goal.

10 References

Alexander, C. (2009). *Market Risk Analysis: Volume IV: Value at Risk Models*, Wiley.

Barker, E. and Kelsey, J. (2007). Recommendation for random number generation using deterministic random bit generators (revised), *NIST Special publication* **800**: 90.

Basel II (2004). International convergence of capital measurement and capital standards: A revised framework, `http://www.bis.org/publ/bcbs107.htm`.

Broadie, M., Du, Y. and Moallemi, C. (2010). Efficient Risk Estimation via Nested Sequential Simulation, Working Paper, Retrieved from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.163.6063&rep=rep1&type=pdf` [February 2011].

Danielsson, J., Embrechts, P., Goodhart, C. and Keating, C. *et al.* (2001). An academic response to Basel II, *Special Paper 130, Financial Markets Group, London School of Economics* .

Glasserman, P., Heidelberger, P. and Shahabuddin, P. (2000). Efficient Monte Carlo methods for value-at-risk, *Mastering Risk* **2**: 7–20.

Python Software Foundation (2011). Heap Queue Algorithm, `http://docs.python.org/library/heapq.html`.

Ronngren, R. and Ayani, R. (1997). A comparative study of parallel and sequential priority queue algorithms, *ACM Transactions on Modeling and Computer Simulation (TOMACS)* **7**(2): 157–209.

Sobol, I. (1998). On quasi-Monte Carlo integrations, *Mathematics and Computers in Simulation* **47**(2-5): 103–112.

Appendix A

```
import numpy
from priority_queue import PriorityQueue
from numpy.random import RandomState
from multiprocessing import Pool, Queue, Process
import scipy.interpolate
from Queue import Empty
import time

#Initialise RNG.
prng = RandomState()

##This function generates correlated multivariate Gaussian numbers using the SciPy module
def rnormcorrel(n, corr_matrix,vol_vec,t):
    cov_matrix = numpy.array([[0.]*n]*n)
    for i in range(n):
        for j in range(n):
            cov_matrix[i,j] = corr_matrix[i,j] * vol_vec[i] * vol_vec[j] * t
    return numpy.random.multivariate_normal(numpy.array([0]*n),cov_matrix)

## Generate outer Monte Carlo Scenarios
## Parameters: - n = number of scenarios required
##             - type specified the portfolio in question
##                       type = 1 for a European put option
##                       type = 2 for the "Guassian" portfolio
##                       type = 3 for a basket option with 4 constituents
##             - params is a list specifying portfolio parameters known at time 0.
##                       the exact parameter list differs between portfolio type.
##  Returns a list of n scenarios. The exact format of the list differs between type.
##  For instance, if type = 1 then Scenarios will be a list of n floats corresponding to
##  n possible time-h stock prices.
def GenScenarios(n, params, type):
    if type == 1:
        Scenarios =
           params[0] * exp((params[3] - (params[4]**2.)/2.) * params[5] + params[4]*sqrt(params[5])*rnorm(n))
    elif type == 2:
```

```
        Scenarios = rnorm(n,mean=params[0],sd=params[1])
    elif type == 3:
        Scenarios = []
        for i in range(n):
            rvs = rnormcorrel(4,params[7],params[4],params[5])
            prices = []
            for j in range(4):
                prices.append(params[0][j] * exp((params[3][j] - (params[4][j]**2.)/2.) * params[5] + rvs[j]))
            Scenarios.append(prices)
    return Scenarios

## Generate inner Monte Carlo trials
## Parameters:
##      tuplParam is the 5-tuple:
##      (m , Scenario, prev_vals, params, type)
##      where m = The number of new trials to generate in this step (usually = 1)
##            Scenario = The scenario corresponding to the particular inner monte-carlo [e.g. a time-h stock price]
##            prev_vals is a list containing:
##              1) The number of trials computed in the specific inner Monte Carlo so far
##              2) The cumulative sum of losses generated in this scenario so far
##              3) The current estimate of the standard deviation of losses in this scenario

#Type 2 = Gaussian variate params: (mu_out,sigma_out,mu_in,sigma_in)
def GenInner(tuplParam):
    m, Scenario,prev_vals,params,type = tuplParam
    n_computed = prev_vals[0]
    L_hat_sum = prev_vals[1]
    sigma_hat = prev_vals[2]

    #Generate sample of m outcomes
    if type == 1:
        S_RN_term =
           numpy.array(Scenario*exp((params[2]-(params[4]**2.)/2.)*params[6]+params[4]*sqrt(params[6])*rnorm(m)))
        samples = params[7] - numpy.array(map(payoff,params[1] - S_RN_term))*exp(-params[2] * params[6])
    elif type == 2:
        samples = -Scenario + params[2] * rnorm(m)
    elif type == 3:
```

```
        S_RN_sample = []
        for j in range(m):
            rvs = rnormcorrel(4,params[7])
            S_RN_term_acc = 0.
            for i in range(4):
                S_RN_term_acc +=
                   Scenario[i]*exp((params[2]-(params[4][i]**2.)/2.)*params[6]+params[4][i]*sqrt(params[6])*rvs[i])
            S_RN_sample.append(S_RN_term_acc / 4.)
        samples = params[8]-numpy.array(map(payoff,params[1]-numpy.array(S_RN_sample)))*exp(-params[2]*params[6])

    #Update L_hat_sum and sigma_hat
    if n_computed > 0:
        acc = L_hat_sum
        M = (n_computed - 1.) * sigma_hat**2.
        n_minus = n_computed
        for dp in samples:
            M = M + (dp - (acc+dp)/(n_minus+1.)) * (dp - acc/n_minus)
            acc = acc + dp
            n_minus = n_minus + 1.
        sigma_hat = sqrt(M/(n_computed+m-1.))
        L_hat_sum = L_hat_sum + sum(samples)
    else:
        L_hat_sum = sum(samples)
        sigma_hat = samples.std()
        #Failsafe for when we don't have enough trials to compute sigma_hat
        if type == 1:
            sigma_hat = 2

    return [L_hat_sum,sigma_hat]

## Non parallelised uniform sampling algorithm
## Parameters:
##      m = Fixed number of trials to sample from each inner Monte Carlo
##      n = Number of scenarios to generate in the outer Monte Carlo
##      type = 1, 2 or 3 corresponding to various different portfolios
## Returns VaR estimate.
def Uniform(m,n,params,type,percentile):
```

```
    Scenarios = GenScenarios(n,params,type)

    #InnerDataset stores the inner Monte Carlo estimates, updated when new trials are generated
    InnerDataset = numpy.array([[0.,0.]] * n)

    for i in range(n):
        InnerDataset[i,:] = GenInner((m,Scenarios[i],[0,0,0],params,type))

    VaR = numpy.percentile(InnerDataset[:,0] / float(m), percentile*100.)
    return VaR

## Parallelised version of the Uniform Sampling algorithm
## Parameters: Identical to those in the Uniform function above.
## Returns VaR estimate.
def mpUniform(m,n,params,type,percentile):
    #Spawns 5 worker processes.
    pool = Pool(processes=5)

    #Generates scenarios (this part is not parallelised)
    Scenarios = GenScenarios(n,params,type)

    map_Scenarios = pool.map(append_map,Scenarios,130)

    InnerDataset = numpy.array([[0.,0.]] * n)

    #Perform the inner Monte Carlo trials in parallel
    #by dispatching the task to the worker processes.
    InnerDataset[:,:] = pool.map( GenInner,map_Scenarios,130)

    #Wait for pool of processes to terminate.
    pool.close()
    pool.join()

    VaR = numpy.percentile(InnerDataset[:,0] / float(m), percentile*100.)
    return VaR

## Calculates an Inner Monte Carlo trial from the queue (for purposes of running in parallel)
```

```
## This function is invoked by worker processes.
## Parameters:
##      queueIn - The FIFO queue for receiving tasks from the main process.
##      queueOut - The FIFO queue for returning the resultant updated
##                 inner Monte Carlo estimates to the main process.
##      params - list specifying the time-0 parameters of the portfolio in question
##      type - Specifies the type of portfolio in question
def workerFunc(queueIn,queueOut,params,type):
    numpy.seterr(all='raise')

    #In each iteration of the loop below, one "step" is performed.
    while True:
        #Wait to receive a task:
        task = queueIn.get()

        #Terminate process if asked to do so by main process:
        if task == 'STOP':
            print "stopped process"
            break

        #Generate a trial from the correct inner Monte Carlo specified by task
        dst = GenInner((task[0],task[1],task[2],params,type))

        #Calculate priority as per Broadie et al (2010) selection rule.
        diff = dst[0]/float(task[4]) - task[3]
        j =  float(task[4])/dst[1] * abs(diff)
        queueOut.put((task[5], dst, j))
        ++k;

## Parallelised Sequential sampling algorithm
## Parameters:
##      m0 = Minimum number of trials generated by each of the inner Monte Carlos
##      mbar = Average of trials generated by the inner Monte Carlos [a stopping rule]
##      n = Fixed number of outer scenarios required
##      type = Portfolio type
##      percentiles = A list of points on the loss distribution.
##                    For each point in the list a PoL is estimated.
```

```
##      VaR_sig = Significance required, e.g. 99% VaR
##      Returns: Interpolated VaR result.
def Sequential(m0,mbar,n,params,type,percentiles,VaR_sig):
    #Generate Scenarios: (this part not parallelised)
    Scenarios = GenScenarios(n,params,type)

    #InnerDataset holds the updated estimates from the inner Monte Carlos
    InnerDataset = numpy.array([[0.,0.]] * n)
    map_Scenarios = map(append_map,Scenarios)#,130)
    InnerDataset = numpy.array([[0.,0.]] * n)
    InnerDataset[:,:] = map( GenInner,map_Scenarios)#,m0)

    #Create as many priority_queues as there are elements in percentiles
    priority_queues = []
    n_qs = len(percentiles)
    for i in range(n_qs):
        priority_queues.append(PriorityQueue())

    #Perform the initialisation (m0 trials from each inner Monte Carlo)
    for i in range(n):
        for h in range(n_qs):
            diff = InnerDataset[i,0]/float(m0) - percentiles[h]
            j = float(m0)/InnerDataset[i,1] * abs(diff)
            priority_queues[h].add_task(j,i)

    m_vec = [m0] * n
    total_iterations = mbar*n
    q = 1
    k = 1
    CanSwitch = True
    iterations = sum(m_vec)
    pr_l = n_qs
    processes = []
    queue_in = Queue()
    queue_out = Queue()

    #Spawn worker processes
```

```
for il in range(pr_l):
    proc = Process(target = workerFunc,args = (queue_in,queue_out,params,type))
    proc.start()
    processes.append(proc)

#Continue to generate trials until we have reached our computational budget
while iterations < total_iterations:
    tasks = []
    #Select tasks (at most n_qs of them) by popping one element off each priority queue:
    for NU in range(n_qs):
        i , prior = priority_queues[NU].get_top_priority(WithPriority=True)
        if i in tasks:
            priority_queues[NU].add_task(prior,i)
            continue
        tasks.append(i)
        iterations += 1
        #Dispatch task i
        queue_in.put((1,Scenarios[i],[m_vec[i]] + InnerDataset[i,:].tolist(),percentiles[NU],m_vec[i],i))

    #Get updated estimates returned by the worker processes
    #and reprioritise tasks:
    for NU in range(len(tasks)):
        result = queue_out.get()
        m_vec[result[0]] += 1
        InnerDataset[result[0],:] = result[1]
        for b in range(n_qs):
            dst = result[1]
            diff = dst[0]/float(m_vec[result[0]]) - percentiles[b]
            prior_q =  float(m_vec[result[0]])/InnerDataset[result[0],1] * abs(diff)
            priority_queues[b].reprioritize(prior_q,result[0])

#At this point this point the nested Monte Carlo is complete
#Terminate worker processes:
for il in range(pr_l):
    queue_in.put('STOP')
for process_w in processes:
    process_w.join()
```

```
    #Free some memory:
    del queue_in
    del queue_out
    for pq in priority_queues:
        del pq

    #Calculate PoL estimates corresponding to each element in percentiles
    arrPoL = []
    for b in range(n_qs):
        arrPoL.append(sum(InnerDataset[:,0]/numpy.array(map(float,m_vec)) >= percentiles[b])/float(n))

    #Invoke scipy's spline interpolation routine:
    f_interp = scipy.interpolate.interp1d(arrPoL, percentiles,kind=n_qs-1)

    #Return the interpolated VaR estimate along with the (x,y) values used in the interpolation
    return (f_interp(1.-VaR_sig), arrPoL, percentiles) #(VaR_guess, s/float(n))

## Below is an example of performing repetitions of a sampling algorithm:
if __name__ == '__main__':
    print "Repeated VaR simulation using the Sequential algorithm"
    numpy.seterr(all='raise')

    #Specify PoL thresholds: [1.0, 1.0833, 1.1667, 1.2500, 1.3333, 1.4166]
    percentiles =  (1. + numpy.array(range(6))/12.).tolist()
    percentiles.reverse()

    #Create an output text file
    o = open('output.txt','a')
    o.write('Sequential, (Inner) m=500, (Outer) n= 8000, Put option, VaR 99% one-week')
    o.write('\n')
    o.close

    #Perform 200 repetitions.
    for i in range(200):
        t = time.time()
```

```
#Explicitly specify all argument:
m0 = 10; m = 500; n = 8000; S = 100.; K = 95.; rfr = 0.03; mu = 0.08; vol = 0.2
h = 1/52. ; T = 1/4. - 1/52.
V0 = 1.669
type = 1 #European put
significance = 0.99
#Invoke nested Monte Carlo experiment
sim_output = Sequential(m0,m,n,[S,K,rfr,mu,vol,h,T,V0],type,percentiles,significance)
#Uniform or mpUniform routines could also have been used!

o = open('output.txt','a')
print '99% one-week VaR estimate',str(sim_output[0])
o.write(str(sim_output[0]))
o.write(' ')
o.write(str(sim_output[1]))
o.write(' ')
o.write(str(sim_output[2]))
o.write('\n')
print "Running time was", time.time() - t,"seconds"
o.close
```

Appendix B - Profiling of the Modified Sequential Algorithm

Each row represents a particular function call (given in the column to the far right).

Cumtime is the total time the program spent in the function (The sum of time to execute all calls to that particular
function)

Tottime is Cumtime less the time that that function spent calling other functions.

185407623 function calls (185398668 primitive calls) in 1672.000 CPU seconds.

Ordered by: cumulative time
List reduced from 418 to 20 due to restriction <20>

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	378.368	378.368	1672.000	1672.000	varparal_prof.py:180(Sequential)
23520018	78.865	0.000	586.863	0.000	priority_queue.py:36(reprioritize)
23932135	81.527	0.000	513.114	0.000	priority_queue.py:9(add_task)
23932135	431.587	0.000	431.587	0.000	{_heapq.heappush}
3920009	57.978	0.000	309.141	0.000	c:\python26\lib\multiprocessing\qeues.py:73(put)
4284120	60.056	0.000	212.812	0.000	priority_queue.py:18(get_top_priority)
7840026	167.424	0.000	167.424	0.000	{built-in method acquire}
14134632	152.756	0.000	152.756	0.000	{_heapq.heappop}
3920003	31.179	0.000	123.971	0.000	c:\python26\lib\multiprocessing\qeues.py:87(get)
3920010	31.669	0.000	117.584	0.000	c:\python26\lib\threading.py:270(otify)
3920003	73.669	0.000	73.669	0.000	{method 'recv' of '_multiprocessig.PipeConnection' objects}
3920010	15.847	0.000	71.441	0.000	c:\python26\lib\threading.py:219(is_owned)
23568019	30.474	0.000	30.474	0.000	{abs}
7840012	15.471	0.000	15.471	0.000	{method 'acquire' of '_multiprocesing.SemLock' objects}
5440605	12.034	0.000	12.034	0.000	{built-in method release}
7840006	11.817	0.000	11.817	0.000	{method 'release' of '_multiprocesing.SemLock' objects}
3920003	8.737	0.000	8.737	0.000	{method 'tolist' of 'numpy.ndarra' objects}
5356053	8.303	0.000	8.303	0.000	{range}
3920011	6.165	0.000	6.165	0.000	c:\python26\lib\threading.py:64(_ote)
3920010	6.017	0.000	6.017	0.000	{method 'append' of 'collections.eque' objects}