SDE Theory For Stock Price Prediction

By Callum Hurd and Ralph Davies

Department Of Mathematical Sciences
September 2023

Supervised by Dr. Kohei Suzuki

Abstract

The first official stock exchange was the Amsterdam Stock Exchange established in 1602, with the sole purpose of raising capital for the Dutch East India Company, making it the first to be publicly traded. Since then the market has grown exponentially, hosting over 1.5 million retail investors in 2022 [1]. In this paper we investigate the modern realm of quantitative trading and the theory of stochastic differential equations that underpins it. We cover both analytic and numerical methods for solving these equations and use this knowledge, along with statistical learning methods, to create our own financial model that forms the basis for a quantitative trading algorithm.

Contents

Preliminaries

This paper requires a foundational knowledge of Stochastic Calculus and Statistics. Along with a robust understanding of calculus, probability and intermediate programming strength in python and R.

1 Introduction

The stochastic differential equations (SDE's) date back to 1900, where French mathematician Louis Bachelier became the first person to model Brownian motion. Their use was later popularised by none other than Albert Einstein who focused on modelling random particle motion in fluids [5]. Today they are explored extensively in both pure and applied mathematics, with uses ranging from random growth models in physics to mathematical finance and stock price prediction. It is the latter that we intend to focus on in this paper.

In 1973, Fischer Black, Robert Merton and Myron Scholes developed the world renowned Black-Scholes model. It was a stock price model and the first widely used mathematical method to calculate the theoretical value of an options contract. Incorrect pricing of an option is generally a big mistake to make, as it can lead to arbitrage opportunities that can be fatal to a market maker. For this reason, in 1997 Black and Scholes were awarded a Nobel memorial prize in Economic Sciences; with Merton being absent due to his passing [2].

More than 20 years later, quantitative finance is a huge part of the financial industry, specifically in trading. Times have also progressed and the Black-Scholes model, albeit brilliant for its time, is now too simple and outdated. SDE theory has been developed, computers are more powerful, and a whole new industry of quantitative trading has been born.

Throughout this paper we will explore some of these advances in the theory, tackling both analytic and unsolvable SDE's along the way. We investigate deeply the use of numerical techniques for approximating solutions to these equations, and evaluate their use in real-world modelling.

Our aim is to utilise the theory of SDE's to create a more pertinent mathematical model for stock price, which we will then attempt to incorporate into an original quantitative trading algorithm. To to this, we will have to use our model, in conjunction with statistical techniques, to generate stochastic indicators to trade of off.

We will make heavy use of programming languages R and python to implement these ideas in practice; showcasing our code and synthetic models along the way.

2 Stochastic Differential Equations

In this section we will provide a mathematical introduction to stochastic differential equations (SDE's), and where possible, provide closed form solutions.

Definition 1. *A **Stochastic Differential Equation** (SDE) is an ordinary differential equation where one or more of the terms is a (or a function of a) stochastic process. The general form of a first order SDE for some stochastic process X_t is given as:*

$$dX_t = b(t, X_t)dt + \sigma(t, X_t)dW_t, \tag{1}$$

with some initial condition X_0. Which we interpret under the Itô formalism as:

$$X_t = X_0 + \int_0^t b(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s \tag{2}$$

The specific class of processes, X_t that we will investigate are called *diffusion processes.*
The existence and uniqueness of solutions to the various SDE's can be guaranteed if certain conditions (which we will not discuss here) are met.

2.1 Linear SDE's

The First class of SDE's that we will investigate are called **Linear SDE's** which have the general form:

$$dX_t = (b_1(t)X_t + b_2(t))dt + (\sigma_1(t)X_t + \sigma_2(t))dW_t \tag{3}$$

A closed form solution can be found via implementation of Itô's formula, which we assume the reader is familiar with. In order to reach this solution we will first investigate the solution to the *homogeneous form* (3). The homogeneous form omits the terms $b_2(t)$ and $\sigma_2(t)$ so that we are left with:

$$dY_t = b_1(t)Y_t dt + \sigma_1(t)Y_t dW_t, \tag{4}$$

where Y_t is the solution to (4).

Lemma 1. *The general solution to the homogeneous linear SDE (4) is given by:*

$$Y_t = Y_0 exp\left\{\int_0^t b_1(s) - \frac{1}{2}\sigma^2(s)ds + \int_0^t \sigma_1(s)dW_s\right\}, \tag{5}$$

where Y_0 is the (potentially stochastic) initial condition.

Proof
In order to prove this lemma we will make use of Itô's formula and consider its application to the function $f(t, x) = \log x$. To begin, we calculate the relevant derivatives of f:

$$f_t = 0, \qquad f_x = 1/x, \qquad f_{xx} = -1/x^2.$$

Thus, using Itô's formula, we obtain the following equation:

$$d(logY_t) = \frac{1}{Y_t}dY_t - \frac{1}{2}\frac{1}{Y_t^2}dY_t dY_t. \tag{6}$$

But we know the form of dY_t given in (4), so we can substitute this into (6). Furthermore, employing the use of the box calculus rules in Itô's formalism, namely: $dtdt = 0$; $dtdW_t = 0$; $dW_t dW_t = dt$, we can simplify the $dY_t dY_t$ term:

$$\begin{aligned} dY_t dY_t &= (b_1(t)Y_t dt + \sigma_1(t)Y_t dW_t)(b_1(t)Y_t dt + \sigma_1(t)Y_t dW_t) \\ &= \sigma^2(t)Y_t^2 dt \end{aligned} \tag{7}$$

bringing this all together, (6) becomes:

$$\frac{1}{Y_t}(b_1(t)Y_t dt + \sigma_1(t)Y_t dW_t) - \frac{1}{2}\frac{1}{Y_t^2}(\sigma^2(t)Y_t^2 dt).$$

From which we observe that the Y_t terms conveniently cancel out to leave us with:

$$d(logY_t) = (b_1(t)dt + \sigma_1(t)dW_t) - \frac{1}{2}(\sigma^2(t)Y_t^2 dt). \tag{8}$$

Collecting terms in dt and dW_t yields:

$$d(logY_t) = (b_1(t) - \frac{1}{2}\sigma_1^2(t))dt + \sigma_1(t)dW_t. \tag{9}$$

From which the desired result follows by converting back to integral form and taking the exponential. □

2.2 A General Solution

Now that we have solved the homogeneous equation (4) and obtained the solution (5), we will attempt to use this to find a solution to the in-homogeneous equation (3). This is in similar fashion to the techniques used when solving ODE's, where one first obtains a homogeneous solution and then uses this to solve the general equation.
Unsurprisingly, we will employ Itô's formula to help us find the general solution. It should also be noted that the trick we will use can extend far beyond the simple case of linear SDE's.

We begin by recalling the homogeneous case (4), and we let this have solution Y_t. Then we let the solution to the general equation (3) be X_t. We now wish to employ Itô's formula on the process $\frac{X_t}{Y_t}$. For simplicity, we invoke the relabelling:

$$X_t^{(1)} = Y_t^{-1}, \qquad X_t^{(2)} = X_t. \tag{10}$$

Under Itô's formula we then have that,

$$d(X_t^{(1)}X_t^{(2)}) = d(X_t^{(1)})X_t^{(2)} + d(X_t^{(2)})X_t^{(1)} + d(X_t^{(1)})d(X_t^{(2)}) \tag{11}$$

Notice that we already have an expanded expression for $dX_t^{(2)}$ as it is simply the left-hand side of the general equation we are trying to solve. To progress further, we therefore need to consider $dX_t^{(1)}$. To do this we again use Itô and consider the function $f(t,y) = \frac{1}{y}$ (since $X_t^{(1)} = Y_t^{-1}$). Its relevant partial derivatives are given by:

$$f_t = 0, \qquad f_y = \frac{-1}{y^2}, \qquad f_{yy} = \frac{2}{y^3}. \tag{12}$$

Thus we find the following,

$$dX_t^{(1)} = \frac{-1}{Y_t^2}dY_t + \frac{1}{Y_t^3}dY_t dY_t. \tag{13}$$

Now, recalling that Y_t is nothing other than the solution to the homogeneous equation, we are able to write dY_t as in (4).
Before we hurry to substitute this into (13), we should observe that we can simplify the quadratic term $dY_t dY_t$, just as in the previous lemma. Therefore we can use,

$$dY_t dY_t = \sigma_1^2(t)Y_t^2 dt, \tag{14}$$

and substituting this back into (13) along with our expression for dY_t gives us:

$$dX_t^{(1)} = \frac{-1}{Y_t^2}\left[b_1(t)Y_t dt + \sigma_1(t)Y_t dW_t\right] + \frac{1}{Y_t^3}\left[\sigma_1^2(t)Y_t^2 dt\right]. \tag{15}$$

Which, after simplification of Y_t terms, (by multiplying though the bracketed terms) and a re-labelling of $Y_t^{-1} = X_t^{(1)}$, we have

$$dX_t^{(1)} = (-b_1(t) + \sigma_1^2(t))X_t^{(1)}dt - \sigma_1(t)X_t^{(1)}dW_t \tag{16}$$

Now that we have expanded our terms for $dX_t^{(1)}$ and $dX_t^{(2)}$ we can fully expand the right-hand side of (11). Once again, however, we pause in our computation of this expression to take notice of the fact that only the $dW_t dW_t$ cross term will not vanish in the expansion of $dX_t^{(1)}dX_t^{(2)}$. This time we do this all in one step and what remains is:

$$\begin{aligned} d(X_t^{(1)}X_t^{(2)}) = &\left[(-b_1(t) + \sigma_1^2(t))X_t^{(1)}dt - \sigma_1(t)X_t^{(1)}dW_t\right]X_t^{(2)} + \\ &X_t^{(1)}\left[(b_1(t)X_t^{(2)} + b_2(t))dt + (\sigma_1(t)X_t^{(2)} + \sigma_2(t))dW_t\right] \\ &-\sigma_1(t)X_t^{(1)}\left[\sigma_1(t)X_t^{(2)} + \sigma_2(t)\right]dt. \end{aligned} \tag{17}$$

Conveniently so, it should be observed that all of the terms containing $X_t^{(2)}$ on the right-hand side, cancel with each other leaving only the following:

$$d(X_t^{(1)}X_t^{(2)}) = X_t^{(1)}b_2(t)dt + X_t^{(1)}\sigma_2(t)dW_t - X_t^{(1)}\sigma_1(t)\sigma_2(t)dt. \tag{18}$$

Collecting like the dt and dW_t terms respectively, we yield,

$$d(X_t^{(1)}X_t^{(2)}) = X_t^{(1)}\left[b_2(t) - \sigma_1(t)\sigma_2(t)\right]dt + X_t^{(1)}\sigma_2(t)dW_t, \tag{19}$$

and reverting back to our original variables gives us,

$$d\left(\frac{X_t}{Y_t}\right) = Y_t^{-1}\left[b_2(t) - \sigma_1(t)\sigma_2(t)\right]dt + Y_t^{-1}\sigma_2(t)dW_t. \tag{20}$$

Integrating over $[0, t]$ and multiplying through by Y_t gives us the desired result.

Theorem 1. *General solution to the In-homogeneous Linear SDE*

$$X_t = Y_t\left(X_0 + \int_0^t Y_s^{-1}(b_2(s) - \sigma_1(s)\sigma_2(s))ds + \int_0^t Y_s^{-1}\sigma_2(s)dW_s\right), \tag{21}$$

Where X_0 is the given initial condition to (3), Y_t is the solution to the homogeneous case and Y_0 has been taken to be 1.

□

2.3 Applications to Linear SDE's

We will now apply **lemma 1** to one of the most important examples in mathematical finance, namely the Black-Scholes equation. The equation is given for a stock S_t which is a stochastic process, has initial price S_0, and is the solution to the stochastic differential equation,

$$dS_t = \mu S_t dt + \sigma S_t dW_t. \tag{22}$$

We see here that the functions $b(t) = \mu$ and $\sigma(t) = \sigma$ are both constant in time. This means that we will have little trouble calculating the stochastic portion of the integrals given in the lemma. Applying **lemma 1** gives us,

$$S_t = S_0 \exp\left(\int_0^t \mu - \frac{1}{2}\sigma^2 ds + \int_0^t \sigma dW_s\right). \tag{23}$$

Which can be evaluated to give,

$$S_t = S_0 \exp\left((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right). \tag{24}$$

So under the Black-Scholes model, the price of a stock at time t, grows exponentially with some additional Gaussian white noise that is scaled by the volatility

term σ. The observant reader may then question why is it that stocks, in general look so 'jagged'? The answer to this question is that we typically have a much larger value of σ than of μ, thus allowing for significant deviation from the mean path.

The efficacy and validity of this model will be discussed later in the paper, where we provide possible approaches to improve it.

3 Numerical estimation via SDE's with R examples

In many cases, SDE's are unsolvable analytically, so we must take to numerical methods to estimate the trajectories and final values on a finite horizon, T . In this section we cover some of the methods used to numerically estimate those SDE's and endeavour to find relationships between them.

3.1 Lamperti transform

The Lamperti transform is used in simulations to reduce the time taken to run code and also to make calculations with an SDE simpler. Consider an SDE of the form:

$$d(X_t) = b(t, X_t)dt + \sigma(X_t)dW_t \tag{25}$$

In other words we have a SDE where the coefficient of the Brownian motion term is dependent on the state variable only. The Lamperti transform chooses a function $y = F(X_t)$, such that under the transformation the volatility coefficient is a constant.

Definition 2. *Let z be any arbitrary value in the state space of X_t, then we define the Lamperti transform to be of the form:*

$$Y_t = F(t) = \int_z^{X_t} 1/\sigma(u)du. \tag{26}$$

This process solves the SDE,

$$dY_t = \left(\frac{b(t, X_t)}{\sigma(X_t)} - \frac{1}{2}\sigma_x(X_t)\right)dt + dW_t. \tag{27}$$

Proof. In order to prove this fact we consider Ito's lemma with $f(t, x) = \int_z^x 1/\sigma(u)du$ then clearly we have

$$f_t(t, x) = 0 \tag{28}$$

$$f_x(t, x) = 1/\sigma(x) \tag{29}$$

$$f_{xx}(t, x) = -\sigma_x(x)/\sigma^2(x) \tag{30}$$

then using Ito's lemma as usual we take

$$dF(t, x) = \frac{b(t, X_t)}{\sigma(X_t)}dt + dW_t - \frac{1}{2}\sigma_x(t, X_t)dt \tag{31}$$

where we have used box calculus to omit $dtdW_t$, dt^2 terms and dW_t^2 has been replaced by dt. □

3.2 Convergence

Definition 3. *An approximation Y_δ to a continuous function Y on a finite horizon T is of strong order of convergence γ if*

$$E(|Y_\delta(T) - Y(T)|) < C\delta^\gamma \tag{32}$$

Definition 4. *For a fixed horizon T and any $2(\beta+1)$ continuous differentiable function of polynomial growth is of weak order of convergence β*

$$|Eg(Y(T)) - Eg(Y_\delta(T))| < C\delta^\beta \tag{33}$$

In financial mathematics, when approximating an SDE that models a stock price, in general, it is the weak convergence that matters the most as it calculates the error in expected payoff. However, if we are considering some other Monte Carlo estimation the strong convergence could be more important as this tells us about the overall error of estimation.

3.3 Euler and Milstein's approximation scheme

3.3.1 Euler's approximation scheme

Euler's approximation scheme is the first scheme we will look at, with which we can approximate numerically an unsolvable SDE. For future reference, this scheme has strong order of convergence $\gamma = 1/2$, which implies that we are as likely to have positive or negative errors. It also has a weak order of convergence $\beta = 1$, which suggests that the error in the final value is directly proportional to the mesh (interval length) that we choose.

We first consider a generic SDE,

$$d(X_t) = b(t, X_t)dt + \sigma(t, X_t)dW_t, \tag{34}$$

where we let $X_0 = x$ and have some arbitrary intervals $\pi([0,T]) := 0 = t_1 < ... < t_N = T$. Then we have that the Euler scheme is a step function given by the stochastic process Y defined by:

$$Y_{i+1} = Y_i + b(t_i, Y_i)(t_{i+1} - t_i) + \sigma(t_i, Y_i)(W_{i+1} - W_i), \tag{35}$$

where $y_0 = X_0$.
Clearly this estimation is purely dependent on some Brownian Motion and time T. So as long as we have Brownian data we can estimate the SDE. This will be useful in estimating stock prices when the SDE being followed is unsolvable.

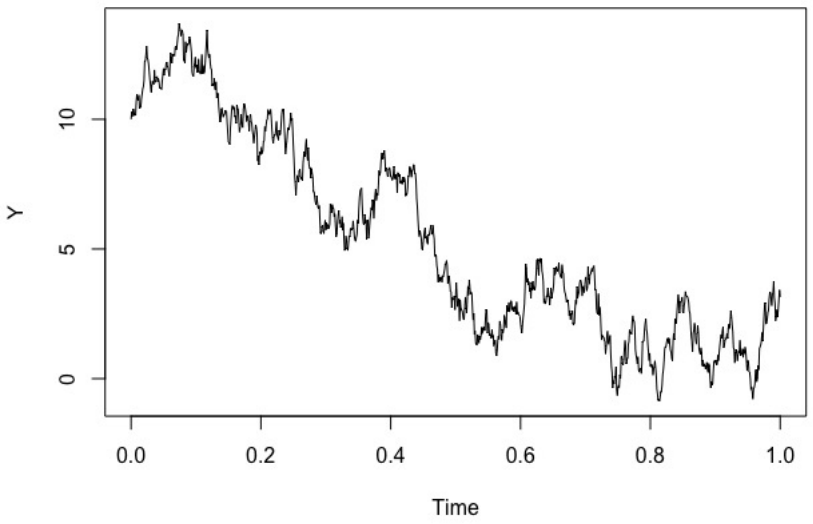
We also note that this scheme estimates the SDE by providing a step function dependent wholly on known constants. We then use linear interpolation to provide a function,

$$Y(t) = Y_i + \frac{t - t_i}{t_{i+1} - t_i}(Y_{i+1} - Y_i). \tag{36}$$

To illustrate this, we consider the Ornstein-Uhlenbeck process, which has the following form:

$$dX_t = (\theta_1 - \theta_2 X_t)dt + \theta_3 dW_t. \tag{37}$$

Where $(\theta_1, \theta_2, \theta_3)$ are constants to be fixed. Below is an example of this scheme applied to the process with parameter values $(2, 4, 10)$



(38)

We have conveniently made use of the R package 'sde'.

```
d <- expression(2-4*x)
s <- expression (10)
sde.sim(X0=10, drift=d, sigma=s) -> X
plot(X, main="Ornstein-Uhlenbeck")
```

3.3.2 Milstein scheme

The Milstein scheme builds upon the euler scheme, adding higher order terms to improve accuracy. The Milstein scheme has a strong convergence of order 1, which improves upon Euler. But the same weak order of convergence taking value 1. This tells us that the Milstein scheme generates a more accurate description of the overall path, but our final value cannot be expected to be more accurate than Euler. This is important to consider when deciding what is necessary in your model. If you are only concerned with the final price of a stock - which is typically the case, then it makes much more sense to utilise the

Euler scheme as it is more computationally efficient.

It will become apparent as we progress through this paper, that one of the main obstacles to success is the optimisation and efficiency of the techniques used in practice. One could come up with the perfect model, but if its implementation is sub-par, it becomes effectively useless!

The algorithm again generates a step function in the same setting as previously, with a stochastic process Y being defined as

$$\begin{aligned} Y_{i+1} = Y_i + b(t_i, Y_i)(t_{i+1} - t_i) + \sigma(t_i, Y_i)(W_{i+1} - W_i) \\ +1/2\sigma(t_i, Y_i)\sigma_x(t_i, Y_i)[(W_{i+1} - W_i)^2 - (t_{i+1} - t_i)] \end{aligned} \tag{39}$$

Once again, this can be implemented smoothly using the sde.sim function.

3.3.3 Comparison of Euler and Milstein

The comparison between the Euler and Milstein can be derived using Ito's lemma and the Lamperti transform from the beginning of This section.

Theorem 2. *The Milstein scheme is directly related to the Euler scheme by taking Lamperti transforms. That is, for a general SDE of X_t, the Milstein scheme is equivalent to the Euler scheme of the Lamperti transform of X_t, Y_t. Up to and including order Δt.*

Proof. We first consider a general SDE

$$dX_t = b(t, X_t)dt + \sigma(t, X_t)dW_t \tag{40}$$

we want to consider the Lamperti transform of the general SDE, so first let $y = F(x)$ which has inverse $x = G(y)$ then by ito's lemma,

$$dF(X_t) = F_x dX_t + \frac{1}{2}F_{xx}dX_t dX_t \tag{41}$$

$$= F_x(b(t, X_t)dt + \sigma(t, X_t)dW_t) + \frac{1}{2}F_{xx}(b(t, X_t)dt + \sigma(t, X_t)dW_t)^2 \tag{42}$$

$$= (F_x b(t, X_t) + \frac{1}{2}F_{xx}\sigma^2(t, X_t))dt + F_x\sigma(t, X_t)dW_t \tag{43}$$

Now letting $F(X_t)$ be the Lamperti transform we get

$$F'(x) = \frac{1}{\sigma(t, x)} \tag{44}$$

$$F''(x) = -\frac{\sigma_x(t, x)}{\sigma^2(t, x)} \tag{45}$$

$$\implies dY_t = \left(\frac{b(t, X_t)}{\sigma(t, X_t)} - \frac{1}{2}\sigma_x(t, X_t)\right)dt + dW_t \tag{46}$$

We know that because the Lamperti transform unitises the volatility term, that the Euler scheme is the same as the Milstein scheme on the transformed SDE.

Now consider the Taylor expansion of the inverse Lamperti transform $G(Y)$, we will use this to find that the Milstein scheme with the inverse transform is equivalent to the Euler scheme, up to and including δt.

$$F(G(y)) = y \tag{47}$$

$$\implies \frac{d}{dy}F(G(y)) = 1 \tag{48}$$

$$F'(G(y))G'(y) = 1 \qquad (chainrule) \tag{49}$$

$$\implies G'(y) = \frac{1}{F'(G(y))} \tag{50}$$

Thus we have that,

$$G''(y) = G'(y)\sigma_x(t, G(y)) = \sigma(t, G(y))\sigma_x(t, G(y)). \tag{51}$$

This means that we can now substitute these values into the order 2 Taylor expansion of the inverse function, which shows that up to order δt, the Milstein scheme and the Euler scheme are the same. I.e. The the previous gives,

$$G(Y_i + \delta y) = G(Y_i) + (b(t_i, X_i) - \frac{1}{2}\sigma(t_i, X_i)(\sigma_x(t_i, X_i))dt \tag{52}$$

$$+\sigma(t_i, X_i)dWt + \frac{1}{2}\sigma(t_i, X_i)(\sigma_x(t_i, X_i)dWtdWt \tag{53}$$

$$+O(\delta t^(3/2)) \tag{54}$$

Hence taking the $G(Y_i)$ on the other side we see that we obtain the Milstein scheme, thus implying that the Euler scheme on the transformed process and the Milstein scheme on the original process are equivalent. □

Since the Euler scheme does not require calculating any derivatives, the calculation time for a simulation is a lot smaller. Therefore when possible we should use the Euler scheme on the Lamperti transform to give an equivalent estimation, with a faster speed than the Milstein scheme.

(Note that we can also use Milstien 2 scheme for a weak second order of convergence and KPS scheme for a strong order of converge of 1.5).

3.3.4 Predictor Corrector method

When considering the Euler and Milstein schemes, although they have good strong and weak orders of convergence, we notice that they assume that the drift and volatility terms are constant over the discrete time intervals $[t, t + \Delta t]$. We will see that this is not the case for the Ozaki method and Shoji-Ozaki linearization methods which allow the drift term to depend on t. The predictor

corrector method is how we attempt to fix the damage caused by the assumption. The method works by predicting what we think the next step will look like (the predictor), and then upon s, takes a weighted average of the prediction and the new data point to find the corrected approximation to the SDE.

The algorithm is as follows:

$$\tilde{Y}_{i+1} = Y_i + b(t_i, Y_i)\Delta t + \sigma(t_i, Y_i)\sqrt{\Delta t}Z \tag{55}$$

where Z is standard normal as usual. Then we choose some α and some η such that we can correct the prediction using the weighted average, i.e the corrector formula

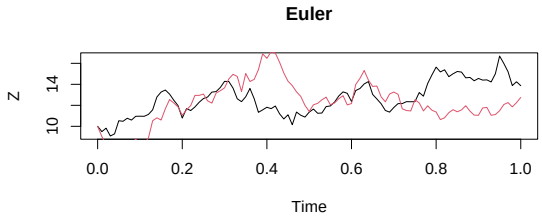
$$Y_{i+1} = Y_i + (\alpha\tilde{b}(t_{i+1}, \tilde{Y}_{i+1}) + (1-\alpha)\tilde{b}(t_i, Y_i)\Delta t \tag{56}$$

$$+(\eta\sigma(t_{i+1}, \tilde{Y}_{i+1}) + (1-\eta)\sigma(t_i, Y_i)\sqrt{\delta t}Z \tag{57}$$

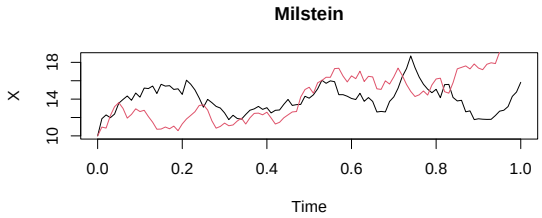
where we use

$$\tilde{b}(t_i, Y_i) = b(t_i, Y_i) - \eta\sigma(t_i, Y_i)\sigma_x(t_i, Y_i) \tag{58}$$

This is useful in the respects of using it for simulations to get a more accurate Euler or Milstein approximation, however it does compromise in efficiency slightly. Below are two graphs depicting the simulated Euler approximation scheme and the Milstein approximation scheme, overlaid with the predictor corrected version in red.



(59)



(60)

3.4 Using Transition Densities

In some specific cases it is possible that we know the conditional distribution of a process at some time $t < T$, this is the case for processes such as Ornstien-Uhlenbeck and CIR. Most importantly for us, the Geometric Brownian Motion

(the process which models stock price over time) has a Log-normal distribution with mean

$$\mu = log(X_0) + (\theta_1 - 1/2\theta_2^2)(T-t) \tag{61}$$
$$\tag{62}$$

and volatility,

$$\sigma = \sqrt{(T-t)}\theta_2. \tag{63}$$

This means that we can predict the path of the SDE and hence the final value. For example, if we take a Geometric Brownian Motion where the trajectories are known up until time t, then we are interested in

$$P(X_T > k|\mathscr{F}_t) \tag{64}$$

or

$$P(X_T < k|\mathscr{F}_t) \tag{65}$$

Which is a simple calculation, given we know the random variables distribution. This becomes especially useful when pricing options that are linear combinations of X_T, since they will also follow a Geometric Brownian Motion distribution.

3.5 The Ozaki and the Shoji-Ozaki methods for linearization

Both of the methods that will be presented in this section are local linearization methods. That is, we approximate the drift of an SDE with a linear function on some small interval $[t, t+\Delta t)$. Instead of a constant function, as done in the Euler and Milstein schemes. A linear function will provide a more accurate approximation.

3.5.1 Ozaki Method

The first of these methods is the 'Ozaki' method and is a base version of the 'Shoji-Ozaki' method. We show it here for completeness, but as we will see, it makes more sense in practice to use the improved version.

The Ozaki method attempts to approximate an SDE of the following form,

$$dX_t = b(X_t)dt + \sigma dW_t. \tag{66}$$

where the volatility is constant. In the Euler and Milstein schemes, the drift term $b(X_t)$ has been approximated as a constant over the interval $[t, t+\Delta t)$, but as stated above, we will assume a linear form in this method. Thus, in deterministic form, we invoke that $b_x(x_t)$ is constant on the interval. We start by observing that x_t obeys,

$$\frac{dx_t}{dt} = b(x_t), \tag{67}$$

where x_t is assumed to be a twice differentiable function of t. Differentiating again gives us,

$$\frac{d^2x_t}{dt^2} = b_x(x_t)\frac{dx_t}{dt}, \tag{68}$$

where we have used the chain rule on b. So what we are left with is a (partially informed) 2nd order ODE for $x(t)$, that we wish to find a solution to. In order for this to properly make sense, we re-parameterise in s as follows:

$$\frac{d^2x(s)}{ds^2} = b_x(x(s))\frac{dx(s)}{ds}, \qquad s \in [t, t+\Delta t). \tag{69}$$

Now, recalling that on the interval $[t, t+\Delta t)$, b_x is constant, we shall re-label this as C. We are left with the 2nd order ODE:

$$\frac{d^2x(s)}{ds^2} = C\frac{dx(s)}{ds}, \qquad s \in [t, t+\Delta t). \tag{70}$$

There are numerous elementary ways to solve this ODE and thus we will omit the calculations which are un illuminating. One possible approach is to integrate twice, first from t to some pseudo-variable u and then integrate again over the entire interval $[t, t+\Delta t)$ with respect to u.

Following these steps results in the following equation:

$$x(t+\Delta t) = x(t) + \frac{b(x_t)}{b_x(x_t)}\left(e^{b_x(x_t)\Delta t} - 1\right). \tag{71}$$

This can then be used to form a solution to the stochastic differential equation. Importantly this all relies on the fact that on the interal $[t, t+\Delta t)$, b_x is a constant. Hence $b(X_t)$ is a linear function and thus we get the name 'local linearisation method'. From this point, to form a model, one needs to derive the transition densities, which is a simple exercise in probability that we omit here. Instead we turn our focus to the more advanced, more useful Shoji-Ozaki method which is an improvement on regular Ozaki.

3.5.2 Shoji-Ozaki method

The Shoji-Ozaki method is an extension of the Ozaki method that we can use on a more general set of SDES. it is again a local linearization process which estimates an SDE of the form $dX_t = b(t, X_t)dt + \sigma(X_t)dW_t$ on the interval $[s, s+\Delta s)$. For functions with a diffusion coefficient which is not constant we must first use the Lamperti transform, but then the transformed SDE can be approximated on a small interval using

$$dX_t = (L_sX_t + tM_s + N_s)dt + \sigma dW_t \tag{72}$$

where we have that:

$$L_s = b_x(s, X_s) \tag{73}$$

$$M_s = \frac{\sigma^2}{2} b_{xx}(s, X_s) + b_t(s, X_s) \tag{74}$$

$$N_s = b(s, X_s) - X_s b_x(s, X_s) - sM_s \tag{75}$$

We then use ito's lemma on a function $Y_t = e^{-L_s t} X_t$ to find the solution

$$dY_t = -L_s e^{-L_s t} X_t dt + e^{-L_s t} dX_t \tag{76}$$

$$\implies dY_t = -L_s e^{-L_s t} X_t dt + e^{-L_s t}((L_s X_t + tM_s + N_s)dt + \sigma dW_t) \tag{77}$$

$$\implies dY_t = (tM_s + N_s)e^{-L_s t} dt + \sigma e^{-L_s t} dW_t \tag{78}$$

which upon integration gives us the required solution

$$Y_t - Y_s = \int_s^t (uM_s + N_s)e^{-L_s u} du + \sigma \int_s^t e^{-L_s u} dW_u. \tag{79}$$

We can derive the solution to X_t from this equation by computing the integrals and considering the stochastic integral as a function of a standard normal. We now derive the solution. We know the discretization of the solution will be of the form

$$X_{s+\Delta s} = A(X)X_s + B(X)Z \tag{80}$$

where Z is the standard normal distribution since the form of Y_t is the sum of a deterministic integral and a stochastic integral. consider equation (74) in two parts, the deterministic integral plus Y_s first and then the stochastic integral second.

1.) The deterministic integral can be computed by parts, this is due to the fact none of the defined variables involve u.

$$A(x) - Y_s = \int_s^{s+\Delta s} (M_s u + N_s)e^{-L_s u} du \tag{81}$$

$$= -\frac{(M_s(s+\Delta s) + N_s)}{L_s} e^{-L_s(s+\Delta s)} + \frac{(M_s u + N_s)}{L_s} e^{-L_s(s)} \tag{82}$$

$$-\frac{M_s}{L_s^2} e^{-L_s(s+\Delta s)} - e^{-L_s(s)} \tag{83}$$

(multiplying through by $e^{L_s\Delta s}$ and substituting for N_s)

$$= -\frac{(M_s(\Delta s) + (b(s,X_s) - X_s b_x(s,X_s)))}{L_s} + \frac{b(s,X_s) - X_s b_x(s,X_s)}{L_S} e^{L_s\Delta s} \tag{84}$$

$$+ \frac{M_s}{L_s^2}(e^{L_s\Delta s} \tag{85}$$

(splitting the two fractions up and then multiplying the first part by L_s on the top and bottom)

$$-\frac{M_s L_s \Delta s}{L_s^2 X_s} - \frac{b(s,X_s)}{L_s X_s} + \frac{b_x(s,X_s}{L_s} + \frac{b(s,X_s)e^{L_s\Delta s}}{L_s X_s} - \frac{b_x(s,X_s)e^{l_s\Delta s}}{L_s} + \frac{M_s}{L_s^2 X_s}(e^{l_s\Delta s} - 1) \tag{86}$$

(factorising, moving the Y_s across and then cancelling the $b_x(s,X_s)$ and l_s we get our final answer)

$$A(x) = 1 + \frac{b(s,X_s)}{L_s X_s}(e^{l_s\Delta s} - 1) + \frac{M_s}{L_s^2 X_s}(e^{l_s\Delta s} - 1 - L_s\Delta s). \tag{87}$$

Fortunately the Z coefficient is much simpler, rather it just requires slightly more calm thinking. Since the integral is stochastic, we cannot directly integrate the expression so we use Itô Isometry and properties of martingales (0 expectation) to find the expectation and variance. First we have that the expectation of the stochastic integral is 0 but,

$$Var(\sigma \int_s^{s+\Delta s} e^{-L_s u} dW_u) = \sigma^2 \frac{e^{L_s\Delta s} - 1}{2L_s} \tag{88}$$

and since the variance of the deterministic part is 0, square rooting gives the standard deviation hence the answer follows,

$$B(X) = \sigma\sqrt{\frac{e^{L_s\Delta s} - 1}{2L_s}}. \tag{89}$$

Now that we have this discretisation, we can say something about the conditional distribution. Since the parameters depend on the state variable, if we condition on s we can define the conditional distribution as,

$$X_{s+\Delta s}|X_s \sim N(A(X)x, B(X)^2). \tag{90}$$

When trying to use the Shoji-Ozaki method we must be careful that the drift function does actually depend on x. If it does not, the final values of $A(X_s)$ and $B(X_s)$ will not be well defined. The Shoji-Ozaki method has some advantages over the Euler and Milstein schemes, for example when Δt is perceived to be large i.e $\Delta t > 0.25$, then the Euler, Milstein and Ozaki schemes do not provide

accurate results since they tend to blow up as the time increment increases. However Shoji-Ozaki does not and continues to provide accurate results past a Δt of 1. In fact as the interval gets larger, Shoji-Ozaki gets more stable. In the appendix we supply the code which simulates an SDE with polynomial drift and constant diffusion with differing time intervals, to display the effectiveness of the different schemes (7.1).

3.6 Monte Carlo for pricing options

In general Monte Carlo simulations are used to find the distribution of a set of data. By artificially simulating points from a distribution and making use of the central limit theorem, we can find a good approximation to the mean and variance of a set of data.

Provided below are two Monte Carlo simulations using synthetic data drawn from a uniform distribution and then a normal distribution respectively. We provide the R code to illustrate how to perform this procedure:

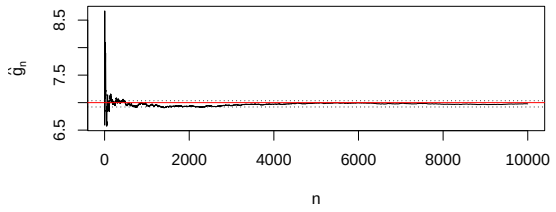
```
n=10000
x=runif(10000, min=-1, max=1 )
g=5*x+7
```

True value of E[G]=7 and the true value of sd= $1/\sqrt{3}$ from calculations by hand

```
mc.mean=mean(g)
mc.sd=sd(g)

plot(1:n, cumsum(g)/(1:n),type='l', xlab='n',
ylab=expression(hat(g)[n]))
abline(h=7,col='red')
abline(h=mc.mean-mc.sd*1.96/sqrt(n),lty=3)
# MC conf interval
abline(h=mc.mean+mc.sd*1.96/sqrt(n),lty=3)
```

This plot provides how accurate the estimation is to the true value as n increases



(91)

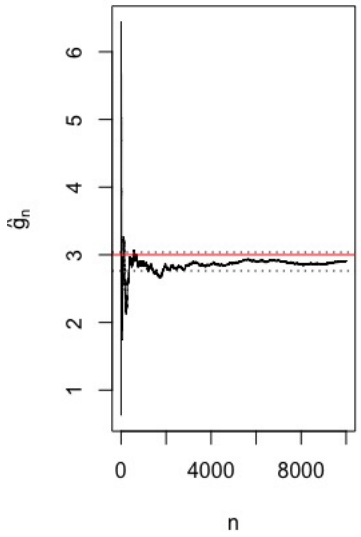
We do this again with normal synthetic data as a second example.

```
 n=10000
x=rnorm(n)
g=7*x+3
```

g is a linear function of a normal distribution which we are using to create synthetic data therefore we have from calculations by hand

```
true.mean=3
true.sd=7
mc.mean=mean(g)
mc.sd=sd(g)

plot(1:n, cumsum(g)/(1:n), type='l', xlab='n',
     ylab=expression(hat(g)[n]))
abline(h=true.mean, col='red')
abline(h=mc.mean-mc.sd*1.96/sqrt(n),lty=3)
# MC conf interval
abline(h=mc.mean+mc.sd*1.96/sqrt(n),lty=3)
```



(92)

From the R code we can see the dashed lines represent an upper and lower bound for a 95 percent confidence interval for the true mean and the red line represents the actual true mean. In both of the plots we see that the true mean is inside the confidence interval, which gives credits to its accuracy, however we need to be careful. If the variance is too high, the Monte Carlo method can give an incorrect approximation.

3.6.1 Pricing options via simulation

Monte Carlo methods can be used to calculate option prices, in this section we will discuss how to use simulated Geometric Brownian Motion walks to accurately price a European call, keeping in mind that this can be modified into any put or call given that the underlying asset follows a Geometric Brownian motion [3] . Consider a filtration $\mathscr{F}_t$ generated by S_t where $dS_t = \mu S_t dt + \sigma S_t dW_t$. When pricing an option we are trying to find the present value of the expected final stock price at some horizon time, T. By using Monte Carlo we can simulate a large number of GBM random walks, beginning at time t, with a set mean and volatility, which means that the histogram of the end values of the walks will provide an approximate probability density function for S_T.

First we need to derive the SDE for stock price using Ito's lemma. Consider the function $Y_t = log(S_t)$ where dS_t is defined as earlier. Then we have that

$$d(log(S_t)) = \frac{1}{S_t}(rS_t dt + \sigma S_t dW_t) - \frac{1}{S_t^2}(\sigma^2 S_t^2)dt \tag{93}$$

$$\implies log(S_{t+\Delta t}) = log(S_t) + (r - \frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}Z \tag{94}$$

where $Z \sim N(0,1)$ and we have discretized in the second line.

Now taking the exponential and rearranging we get the required result that we will work with

$$S_{t+\Delta t} = S_t e^{(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}Z}. \tag{95}$$

(On a side-note notice that S_t is log-normal with $\mu = (r - \frac{\sigma^2}{2})\Delta t + log(S_t)$ and $\sigma^2 = \sigma^2\Delta t$. That is $log(S_t)$ is normally distributed with these parameters).

Now we are ready to use simulation to calculate the price of a European call option. Again, we provide the python code here to illustrate the procedure.

```
from math import *
import matplotlib.pyplot as plt
from time import time
import random
import matplotlib
from numpy import *
from matplotlib import cm
t0=time()
```

We need to define all of the constants such as the price at the current point in the filtration, the time intervals we discretise on, and parameters for the Geometric Brownian Motion.

```
St = 100.; K = 105.; T = 1.0; r = 0.05; sigma = 0.2
M = 50; dt = T / M; I = 100
```

We define S_t as current price, K as strike price, T as execution time, r as risk free interest rate, (needed to have a complete market). We have defined M as the number of intervals we want, then curated dt with the corresponding interval size.

```
S = St * exp(cumsum((r - 0.5 * sigma ** 2) * dt
+ sigma * sqrt(dt)
* random.standard_normal((M + 1, I)), axis=0))
```

This is the solution to the SDE for dSt found using Ito's lemma on log(St) and then discretized

```
S[0] = St
```

```
C0 = math.exp(-r * T) * sum(maximum(S[-1] - K, 0)) / I
print('The European Option Value is: ', C0)
# The European Option Value is:  8.165807966259603
```

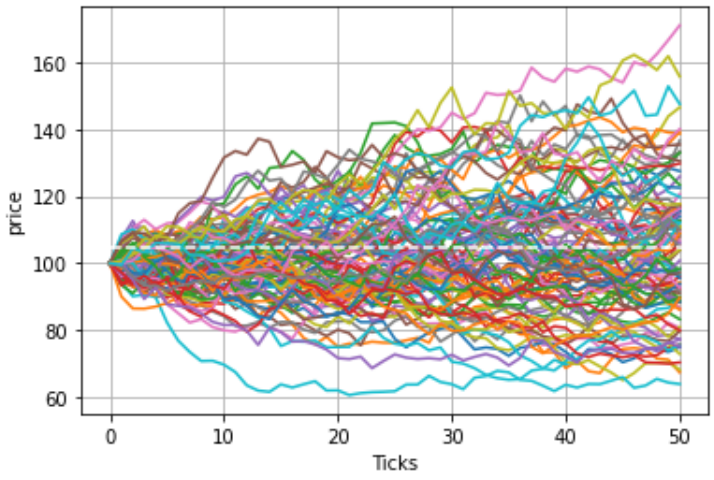
Here we take the average between all of the call options at time T the final values of the GBM paths which constitutes as the mean. we then discount it using continuous time interest

```
mnte_crlo_avrg=sum(S[-1])/I
# The monte carlo simulation #average is 104.94192865350077

avrg=100+np.log(S0)+(r-(sigma**2)/2)*(dt)
# The average final value by
# analtical calculation is 104.60577018598809

plt.plot(S[:,:100])
plt.axhline(y=avrg,color='white', linestyle='--')
plt.grid(True)
plt.xlabel('Ticks')
plt.ylabel('price')
```

This section of code provides the plot of the GBM paths where we can see the normal distribution of values spread around the mean



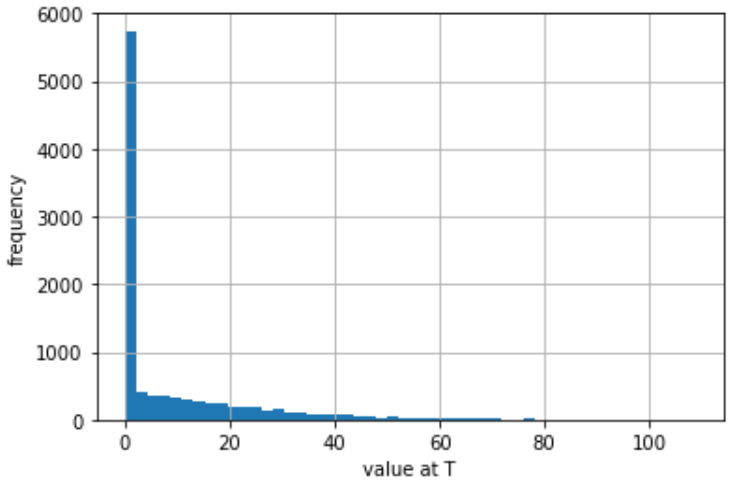
(96)

This code can be easily modified to fit for any put or call option by modifying the function inside the sum in the C0 variable to whatever option you plan to price.

Analysing this further we can see the option chosen is a European call option $(S_t - 105)^+$. An options trader may wonder whether or not this option is likely

to make them money, plotting the histogram of final S_t values will enlighten us further.

```
plt.hist(maximum(S[-1] - K, 0), bins=50)
plt.grid(True)
plt.xlabel('value at T')
plt.ylabel('frequency')
plt.ylim(0, 6000)
```



In this simulation we increased the number of paths for a better histogram, but the premise still stands, here we see that about 63.5% of the time this option will expire worthless, which could stave off any risk averse traders!

3.7 Final notes on simulation

In this section we will discuss in a practical sense a couple of simulation methods which we haven't yet mentioned in the previous sections, and then summarise the sde.sim function which encapsulates all of the simulation methods we've explored so far.

3.7.1 Brownian and diffusion bridges

In general, a bridge simulation is useful for a lot of reasons in pricing options. A barrier option (up and out , down and in etc) only pays out if you are either over or under a certain price point depending on the option. By simulating bridges, we can directly find the probability that the price point is reached.

Brownian bridge simulations are useful when you want to simulate the paths between two observations O_1 and O_2 at times t_1 and t_2 respectively using a regular Brownian motion. The function to compute this in R is the built in function BBridge namely

```
BBridge(x,y,t0,T,N)
```

Where x and y define the beginning and end points, t0 and T are the start and end times and N is the amount of ticks.

Diffusion bridges will generally be more useful since you can input any SDE including geometric Brownian motion, CIR or any other SDE which your underlying asset follows. Diffusion bridges are more complicated in the sense they rely on time reversibility of Markov processes to compute a bridge. The function generates two processes, reverses one and combines them at the crossing point to create a simulated path which has the same distribution as the two individual paths, thus creating a bridge.

```
DBridge(x,y, delta, drift, sigma,...)
```

The '...' indicates where you can choose the scheme you want to use to simulate, but also for any arguments you want passed through the sde.sim function.

3.7.2 The sde.sim function

The formal definition of the sde.sim function directly from Stefan M. Iacus's book is as follows

```
sde.sim(t0 = 0, T = 1, X0 = 1, N = 100, delta, drift, sigma,
        drift.x, sigma.x, drift.xx, sigma.xx, drift.t,
        method = c("euler", "milstein", "KPS",
        "milstein2", "cdist", "ozaki","shoji","EA"),
        alpha = 0.5, eta = 0.5, pred.corr = T,
        rcdist = NULL, theta = NULL, model = c("CIR",
        "VAS", "OU", "BS"), k1, k2, phi,
        max.psi = 1000, rh, A, M=1).
```

Where we have inserted this to display every parameter you can make use of inside the function. In reference to the schemes, there are a few rules of thumb

we must try to follow when simulating.

We have mentioned in this chapter already, that if one wishes to improve the efficiency of code then we must perform the Lamperti transform on the SDE first. We must do this in any case if we want to to use the Ozaki, Shoji-Ozaki or the Exact sampling algorithm.

If we can draw from the distribution of previous data then we should, this will give a much more accurate simulation than a discretisation method and will avoid any problems such as negative values. We do this via the exact sampling algorithm.

4 Model Selection and Synthetic Data

4.1 Navigating synthetic data in R

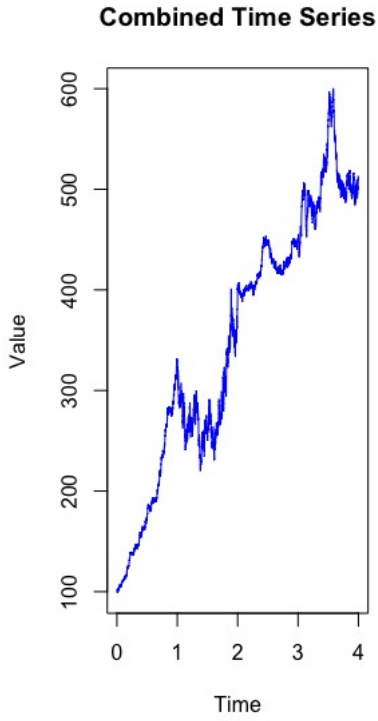
The main issue with approximating stock prices with geometric Brownian motion is that volatility and drift are not always the same across a time period. We cannot say for certain, even on small time periods that the stock price follows a specific GBM model.

Looking at real stock price data over the course of a year, we might come to the conclusion that in general, the price is quite stable and increasing on average by 8% per Annum (general consensus for s&p 500, average increase across all stocks). However, around the time of earnings calls or pre-determined, important events for the company, the stock could become more volatile. In addition, depending on how good the event is for the company, it could positively or negatively affect the mean for a certain period of time. This is something we should take into account when using past data to help predict parameters of a model.

In general it is okay to model very short time periods by one model as the probability that external factors affect the model is slim. However, if we want to have one piece of code that runs continuously whilst receiving new data, we will have to ensure that it allows the model to constantly change. Another issue rises in creating synthetic data for testing: if we use only one model, the methods in this section will accurately determine the parameters. However, running under the assumption that real data should contain multiple GBM's, we must devise a way to determine when these changes occur.

The overarching point is that we want to find an algorithm which can tell us the parameters of a GBM model, tell us when the model changes, getting rid of the old data and then tell us the new model parameters.

First we consider the code for plotting a graph of multiple GBMs sewn together over a time period of 4 units. The code can be found at (7.2) in the appendix.



This model sews together 4 GBM simulations each starting from the end point of the previous GBM, and each lasting for exactly one time period. Obviously this would only be accurate for a real stock price if we knew what the model would be in advance, and also knew when the changes in parameter values were likely to occur. Most of the time we don't know this, but we can take a good guess as to where they might be. To fix this, we consider modelling the 'change-points' of the model with an exponential distribution of fixed parameter.

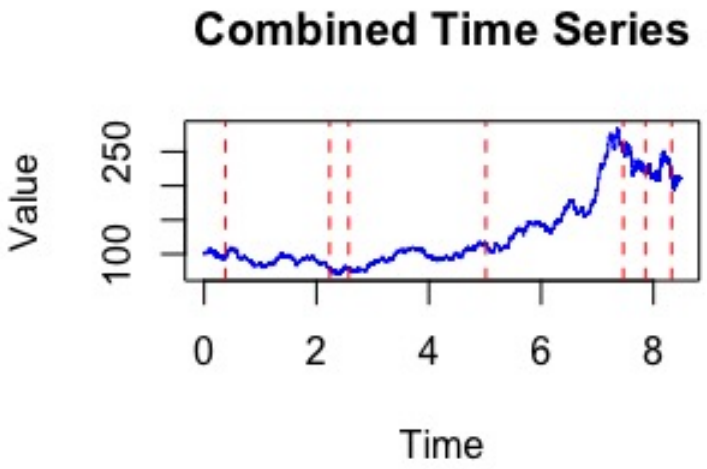
We introduce a more complete version that we could use for our synthetic data if we continued in R. We won't be using this in the future sections, but we decided to cover for completeness. It provides good, randomly generated synthetic data which has close similarities to stock data. This new code works in the following way:

1. It generates a random number of change points between 0 and 10 (inclusive) using sample(0:10, 1)
2. For each change point, it generates a random value from an exponential distribution with a rate of 0.7 using rexp(1, rate = 0.7). These change points represent the times when the SDE parameters change. we can obviously change the parameters to suit our needs.

3. The code then iterates through the change points and simulates the SDE for each segment using the sde.sim function from the sde library
4. Simulated values and time points for each segment are stored in separate vectors
5. After simulating all segments, the code combines the simulated values and time points into a single data frame, combined_data
6. The combined_data is then ordered by time points
7. Finally, the code plots the combined time series using plot, and adds red dashed lines at each change point using abline

Once again, a detailed parsing of the code can be found in the appendix. (7.3.1)

Below is the final product,



4.2 The switch to python

So far, we have relied heavily on R and the sde.sim function to simulate our data, provide inference and draw conclusions from. Moving forwards we decided to make the switch to python programming language for a few reasons:

- Python is quicker than R for the type of simulations we are running, especially when one makes use of the numpy library.

- It is less restrictive in the way we choose to handle our data - which is very useful when we are simply trying to simulate synthetic data.

- Moving forward, we would like to make use of the Machine learning and A.I libraries in python such as pytorch, and incorporate them into a real-time, trading algorithm. The diversity of python will allow us to do this more seamlessly than in R.

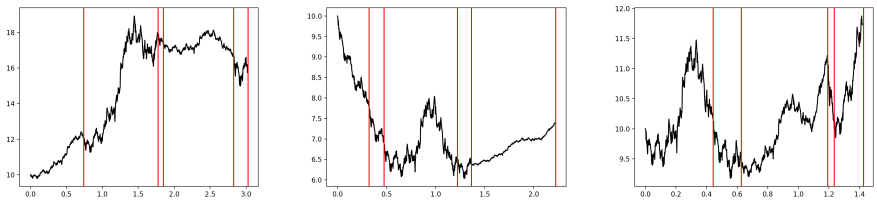
4.2.1 Synthetic data in python

The model in python will remain the same as in R, however we now take a more functional approach so that consecutive models can be simulated with differing coefficients much more easily. The code is provided at (7.4) in the appendix; a brief overview of its purpose:

1. Generate a simulated Black-Scholes time-series given parameters mu and sigma, some end time T, and a refinement N - which will dictate the number of steps in the series.
2. Pre-determine the change points from an exponentially distributed distribution, given a specified number of them required.
3. Generate arrays of μ and σ values to be used in the different models.
4. Sew these models together to create one combined time-series.

This new code architecture is more interpretable and will more easily enable us to make comparisons between pre-specified and inferred data values, which is important for assessing the validity of our statistical inference methods.

This code produces the plots of the data shown below in (97) where the vertical red lines indicate the change points of the stock. Note: the differences in paths due to the random nature of parameter selection (where we have neglected the use of the seed parameter) .



(97)

The curious reader is encouraged to play around with the code and observe the key changes that occur with varying values of μ and σ.

5 The Trading Algorithm

Over the course of this paper we have kept in mind that our main goal is to utilise these mathematical techniques to create a trading algorithm which is not only efficient, but profits on real time stock data.

As mentioned previously, in order to proceed with real stock data, we must first provide a way to learn the values of μ and σ with some degree of accuracy.

5.1 Parameter Estimnation

There are many ways one could approach the problem of learning μ and σ. In this paper we focus on a statistical learning technique called the Maximum Likelihood Estimation method. Other possible routes could include more sophisticated machine learning algorithms including pattern recognition (if one were to believe historical data could predict the parameters).

5.1.1 The MLE Method

(Proceed with caution - the fact that m and v are not independent causes an issue. Please read the drawbacks of the MLE method section afterwards.)

The MLE method is a bridge between Frequentist and Bayesian statistics it aims to maximise something called the likelihood function, which is the joint probability density of observed data *given* the parameter values. The aim is to find estimates of the parameter values that maximise this probability density. As these can be thought of as the values that would most *likely* lead to the observed data... hence the name.

The fact that this concept can also be integrated with the Bayesian framework is perfect for us since, in later iterations, we could incorporate prior beliefs about our system that may be obtained from historical data. We now move on to the method.

The Geometric Brownian Motion has a known distribution, in fact we have already discussed its log-normality. This means upon taking the log of $\frac{S_{t+1}}{S_t}$ we can say the log data is normally distributed with these parameters.

We derive the PDF of the Log-normal distribution and the log-likelihood functions starting with the GBM formula for stock price

$$S_t = S_0 \exp\left((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right) \tag{98}$$

$$\implies log(\frac{S_{t_{k+1}}}{S_{t_k}}) = (\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}Z = X(t_k) \tag{99}$$

$$\implies E[X(t_k)] = (\mu - \frac{1}{2}\sigma^2)\Delta t =: m \tag{100}$$

$$\implies Var[X(t_k)] = \sigma^2\Delta t =: v \tag{101}$$

m and v are the actual mean and variance of a log normal PDF on the geometric Brownian motion. Keep in mind upon estimating the parameters μ and σ, these parameters will obtain hats which imply they are also estimates.

Now, using this distribution and writing the likelihood using the normal pdf:

$$f(X(t_k|m, v)) = \frac{1}{\sqrt{2\pi v}} \exp\left(-\frac{(x-m)^2}{2v}\right) \tag{102}$$

$$\implies \mathcal{L}(m, v) = \frac{n}{2}log(2\pi v) - \frac{\sum_{i=1}^{n} x_i^2 + 2m\sum_{i=1}^{n} x_i - nm}{2v}. \tag{103}$$

$$\tag{104}$$

By differentiating by each parameter separately and setting equal to zero we find that,

$$\frac{\partial \mathcal{L}}{\partial v} = -\frac{nv}{2} + \frac{\sum_{i=0}^{n-1}(x_i - m)^2}{2} = 0 \tag{105}$$

$$\implies \hat{v} = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - m)^2 \tag{106}$$

$$\frac{\partial \mathcal{L}}{\partial m} = \frac{2\sum_{i=0}^{n-1}(x_i - m)}{\frac{1}{n}\sum_{i=0}^{n-1}(x_i - m)^2} = 0 \tag{107}$$

$$\implies \hat{m} = \frac{1}{n}\sum_{i=0}^{n-1} x_i. \tag{108}$$

Using the expectation and variance from before we have that,

$$(\hat{\mu} - \frac{1}{2}\hat{\sigma}^2)\Delta t = \hat{m} \tag{109}$$

$$\hat{\sigma}^2\Delta t = \hat{v} \tag{110}$$

$$\implies \hat{\sigma}^2 = \frac{\hat{v}}{\Delta t} \tag{111}$$

$$\hat{\mu} = \frac{\hat{m}}{\Delta t} + \frac{1}{2}\hat{\sigma}^2. \tag{112}$$

Given these formulae, we can easily write code that estimates our parameters. In the case that we started with an SDE that was unsolvable, thus leaving us without an explicit distribution, we would turn to pseudo-likelihood methods for estimating parameters. They incorporate some of the estimation techniques we have already discussed in this paper.

We outline the implementation of our MLE in python code below:

```
def MLE_finder(price_data, delta_t= 1):
    stock_data = np.log(price_data)
    N = len(stock_data)
    x = np.array([stock_data[i] - stock_data[i-1] for i
    in range(1,N)])
    sum_x = sum(x)
    sum_x_squared = sum(x**2)
    m_hat = sum(x)/N
    v_hat = (1/N) * (sum_x_squared
    - 2*m_hat*sum_x) + m_hat**2

    sig_sq =v_hat/delta_t
    sig_hat = np.sqrt(sig_sq)
    mu_hat = m_hat/delta_t + 0.5*sig_sq

    return mu_hat , sig_hat
```

This code takes time series data of the form (price_data) and outputs the estimated parameters of your GBM.

5.1.2 Drawbacks of the MLE method

There are however, some unfortunate drawbacks to using the MLE approach for estimating the parameters. The issues that arise stem from the fact that geometric Brownian motion is inherently random. Thus even for a known, fixed set of parameters μ and σ, there will always be a significant variation in our MLE estimates without access to extremely high resolution data. We will illustrate this both mathematically and numerically below.

We will focus on the estimation of μ in this paper and note that analogous results can be proven for σ. To begin, let us recall the formulas for $\hat{\mu}$ and $\hat{\sigma}$ respectively. They are given as:

$$\hat{\mu}_n = \frac{\hat{m}_n}{\Delta t} + \frac{1}{2}\hat{\sigma}_n^2, \qquad \hat{\sigma}_n^2 = \frac{\hat{v}_n}{\Delta t}, \tag{113}$$

Where we have introduced a subscript n, into our notation in order to denote the number of elements in our sample. Furthermore, note that Δt is implicitly

a constant value. We first observe that $\hat{\mu}_n$ is not an unbiased estimator of μ. To do this we must examine the properties of $\hat{m}_n$ and $\hat{v}_n$:

$$\begin{aligned} E[\hat{m_n}] &= E\left[\frac{1}{n}\sum_{i=0}^{n-1} X_i\right] \\ &= \frac{1}{n}\sum_{i=0}^{n-1} E[X_i] \\ &= \frac{1}{n}\sum_{i=0}^{n-1} m \\ &= m. \end{aligned} \tag{114}$$

And similarly for $\hat{v}_n$, it can be easily shown that $E[\hat{v}_n] = \frac{n-1}{n}v$. Now performing the same calculation for $\hat{\mu}_n$ gives:

$$\begin{aligned} E[\hat{\mu}_n] &= E\left[\frac{\hat{m}_n}{\Delta t} + \frac{\hat{v}_n}{2\Delta t}\right] \\ &= \frac{1}{\Delta t}\left(m + \frac{1}{2}\frac{n-1}{n}v\right) \\ &= \frac{m}{\Delta t} + \frac{1}{2}\frac{n-1}{n}\frac{v}{\Delta t}, \end{aligned} \tag{115}$$

as required. Where we have used that $v = \sigma^2\Delta t$. It is important to remember that in reality, Δt must be chosen *prior* to asserting a sample size; consequently it is considered fixed throughout, and discussing its limiting properties in this context does not make sense. For this reason, we will normalise Δt to 1 moving forwards. This makes sense empirically as we will typically be modelling data with a given unit, be that 1 day, 1 hour or 1 minute in real time. This will obviously have a scaling effect in our estimates for $\hat{\mu}_n$ but so long as we are consistent, this will not change the model at all.

Looking now at the variances, calculating $Var[\hat{m}_n]$ is easy since $\hat{m}_n$ is simply the sum of independent, normal random variables;

$$Var[\hat{m}_n] = Var\left[\frac{1}{n}\sum_{i=0}^{n-1} X_i\right] \tag{116}$$

$$= \frac{1}{n^2}\sum_{i=0}^{n-1} Var[X_i] \tag{117}$$

$$= \frac{v}{n}. \tag{118}$$

And thus we now begin to observe some useful properties of our estimate constituents. As our sample size, n, increases, the variance in our estimate for m

decreases at rate $\frac{1}{n}$.
It is at this point, we must address a slight inaccuracy in our current approach to finding $\hat{\mu}$ and $\hat{\sigma}$. The problem is that when calculating maximum likelihood estimates, we have been calculating $\hat{m}$ and $\hat{v}$ which were superficially defined to make calculations cleaner. The issue with this, is that it does not actually guarantee that $\hat{\mu}$ and $\hat{\sigma}^2$ are maximum likelihood estimates themselves. As it turns out, $\hat{\mu}$ is indeed $\hat{\mu}_{MLE}$. This simply falls out of the log-likelihood when taking partial derivatives with respect to μ and maximising. We obtain the following equation,

$$\frac{\partial \mathcal{L}}{\partial \mu} = \frac{1}{\sigma^2} \sum_{i=0}^{n-1} \left(X_i - \mu + \frac{1}{2}\sigma^2 \right). \tag{119}$$

From which it can easily be shown that $\hat{\mu}_{MLE} = \frac{1}{n} \sum_{i=0}^{n-1} x_i + \frac{1}{2}\hat{\sigma}^2$. To investigate the properties of $\hat{\mu}_{MLE}$ we can therefore make use of the already well-established MLE distribution theory which states the following:

Theorem 3. *Large-sample distribution of a maximum likelihood estimate.*
Let $\mathbf{X} = (x_1, ..., x_n)$ *be an i.i.d sample of size* n *from* $f(x|\theta)$ *with a regular likelihood. Then the MLE* $\hat{\theta}$ *for* θ *is such that,*

$$\hat{\theta} \to \mathcal{N}\left(\theta, \frac{-1}{\mathcal{L}''(\theta)} \right) \tag{120}$$

(in distribution) as $n \to \infty$. □

This is perfect for us as it allows easy modelling of the behaviour of $\hat{\mu}$ for sufficiently large n. If we calculate partial derivatives we find the very nice result,

$$\mathcal{L}''(\hat{\mu}) = -\frac{n}{\sigma^2}. \tag{121}$$

This implies that for a sufficiently large n we have,

Proposition 1.

$$\hat{\mu}_{MLE} \dot{\sim} \mathcal{N}\left(\mu, \frac{\sigma^2}{n} \right). \tag{122}$$

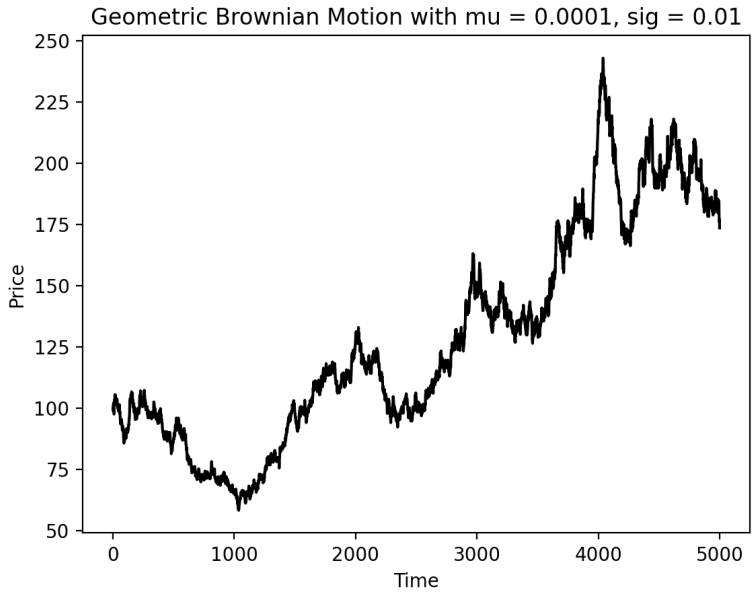
Of course, in order to use this in practice, we will have to substitute our estimate $\hat{\sigma}^2$ into the above. But importantly we now have some concept of how our estimate for $\hat{\mu}_{MLE}$ will vary in relation to the size of our sample.

5.1.3 Empirical results

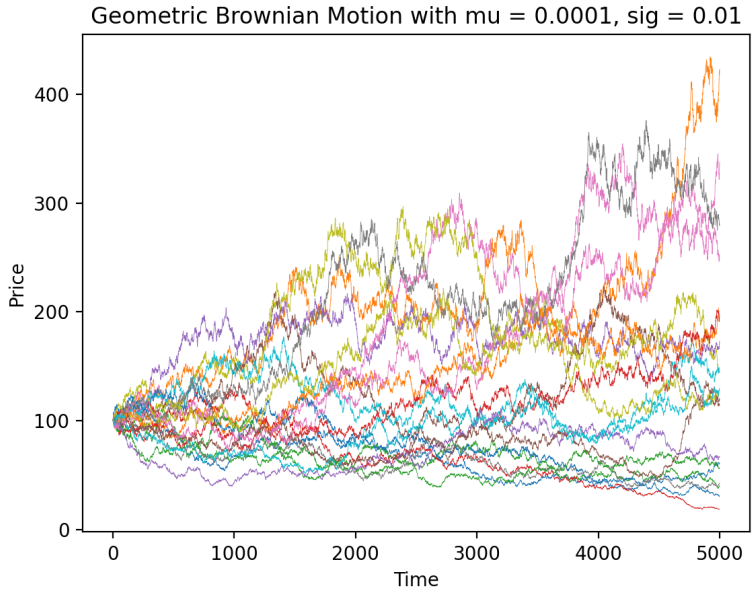
Now that we have a theoretical base, we can take a look at how this pans out empirically. Our aim is to make an observation of how our estimate for μ varies depending on the size of our sample. At this point it would be beneficial to recall that our sample X_i are extracted by taking the differences between consecutive stock prices. Therefore our N in the gbm python function is the appropriate candidate. We will illustrate this in the following way:

- Choose a set values for μ and σ.
- Generate multiple GBM's in python with the same parameters μ, σ.
- Learn the values of μ and σ with the MLE method varying the number of points in our sample.
- Plot histograms of the results.

Before we look at the statistics, we can look at the possible paths the stock could take by simulating a large number of gbms. For this example we pick $N = 5000$, $\mu = 0.0001$, $\sigma = 0.01$ and a starting price of $S0 = 100$. An example of what this could look like is given below:

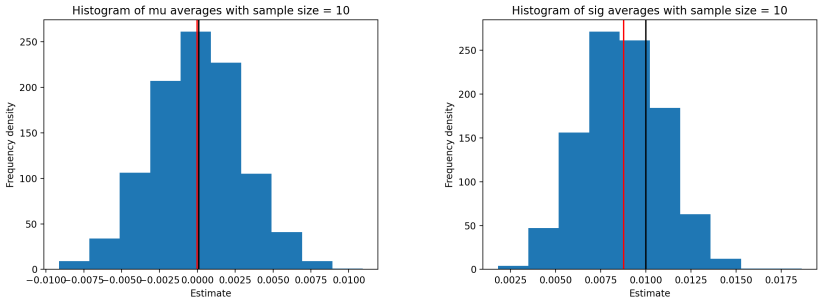


But of course, this is one specific case of an infinite possible number of paths that could have been taken under the exact same parameters. The deviation from the mean is determined by the relative size of the volatility. We illustrate the potential deviation below:

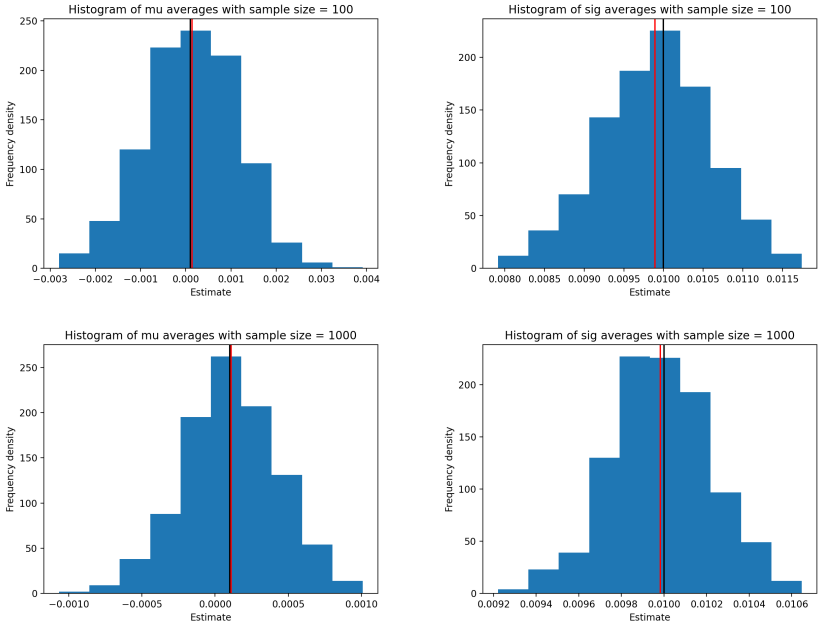


As you can see, there is significant variation in the possible paths that can be taken, which is why it will take quite a significant number of points in our sample to determine the values of μ and σ accurately.

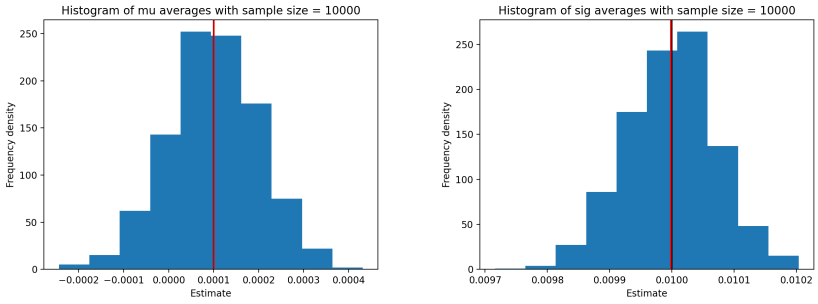
We will now go through the steps defined above to assess the variation in the parameter estimation. In the following histograms, the black line represents the true value of the parameter and the red line represents the average of the estimated values.



As expected, the variation in the estimate of our μ parameter is very large for such a small sample. On top of this, there is a clear deviation in the mean value of $\hat{\sigma}$ and its true value. This is due to the heuristics involved in our calculations and something to be corrected moving forward.

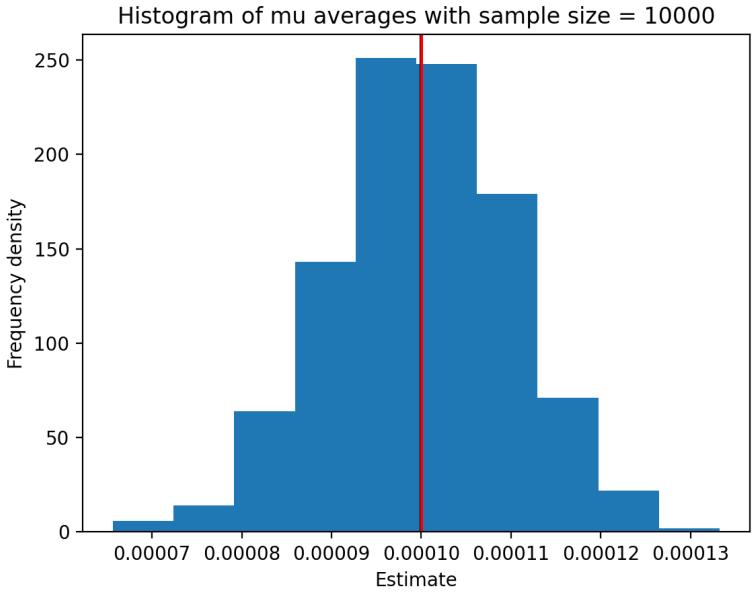


By the time we have reached 1000 sample points, it is clear that any of our problems in estimating σ are significantly outweighed by the variation present in the calculation of $\hat{\mu}$. Even with the true value of σ, this variation is unavoidable, as shown in the previous section. We show our best result using 10,000 data points



below:

As we have seen, the issue truly lies in the ratio between μ and σ. In this example, $\sigma = 0.01$ is large in comparison to the size of μ. A decrease in σ to 0.001, leaves us in a much better situation:



5.2 The change point detector algorithm

Definition 5. *Given an underlying asset with stock price S_t modelled using estimated and predetermined parameters of a GBM, let $X_t = log(S_t) - log(S_{t-1})$. A **Change Point** is a stopping time t, such that X_t is outside of a t-distributed prediction interval, where X_t is defined as the difference between the present and previous value of the asset at time t.*

That is for some time t+1

$$\bar{X}_t - t_{n-1,\alpha/2} s_n \sqrt{1 + \frac{1}{n}} \leq X_{t+1} \leq \bar{X}_t + t_{n-1,\alpha/2} s_n \sqrt{1 + \frac{1}{n}} \tag{123}$$

where α corresponds to the 100(1-$\frac{\alpha}{2}$)% level and s_n is simply the standard deviation over the n points.

This can be extended to consider just the log of the stock by rearranging the interval, ie. move the $log(S_{t-1})$ to the other side. This is what we use in our algorithms.

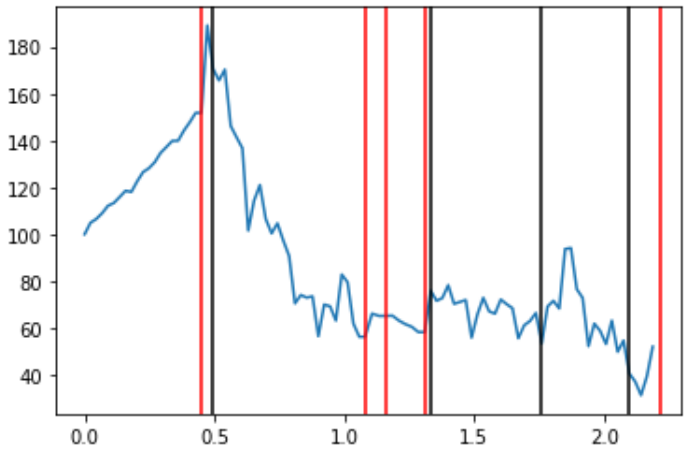
We also want to define the notion of a Reverse Point.

Definition 6. *Under the same setting as the previous definition and if the predetermined drift is positive. If X_t is the lower than the lower bound of the prediction interval, then this is a **Reverse Point**. If the pre-determined drift is negative, the opposite applies.*

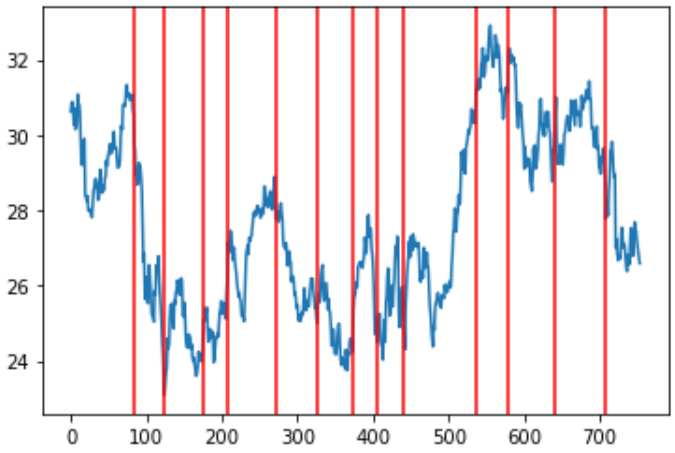
Upon creating the MLE finder algorithm, we had found a direct route to change point detection in stock market data. The change point detector algorithm takes a time series data then creates a t-distributed prediction interval on the log data. It then sequentially takes each data point, assesses if it is within the prediction interval, if it is then you recalculate the MLE's including the new point, if it is not, then it detects a change point, considers the next N points and starts over. The code produces a list of the times of change points, along with an array of the corresponding μ and σ values on each interval. This means that at any time t, you use the most up to date model estimation and can price options based on the current data model, or looking at the drift and volatility of the SDE's you can make decisions on whether or not to acquire shares or short the stock. The reason why we are using a t-distribution for the prediction interval rather than a normal distribution, is because we are estimating both the mean and standard deviation.

Please note that in this iteration, we are considering actual change points. When it comes to actually trading, we will consider reverse points. The code can be found at (7.6.1) in the appendix.

The red lines in the plot below are pre-determined change points, the black lines are change points detected by the algorithm. It is interesting to see that although the pre-determined change points are detected when a significant change has occurred, small changes in direction are not detected. This is generally a positive as it allows you to trade for longer periods of time without having to keep executing orders.



In real time stock data this would look slightly different, there would be no pre-determined change points so it relies heavily on being able to accurately predict when it will go up or down. The next plot is a plot of the change-point detector algorithm on MSFT (Microsoft) data.



5.2.1 Cautions

We now discuss some things to consider when using such a change point algorithm.

The change-point detection relies directly on the underlying mathematical model that is being used to approximate and predict stock price. What this means is that any inaccuracies in model suitability or parameter estimation will have a direct carryover into this detection scheme. Thankfully, the change point detection itself only relies on the parameter σ which is generally estimated without much error. But when we begin to use this information to make trades, we must understand that there are drawbacks. An illustration of this should be the fact that when considering stock data, we will generally be testing a lot of points. Therefore even with a perfect model and parameter knowledge, a 95% prediction interval will fail 5% of the time! This might not sound like a lot, but when we are considering hundreds of points in between trades, this could trigger more than enough false positives to neuter any strategy we wish to employ!

5.3 Profitable Prospects

We can now take a look at the strengths of our trading decisions. Let us suppose that for any given trade, we want there to be a 60 percent chance of being profitable. In between trades, the assumption is that the parameters μ and σ remain constant. And so the standard Black-Scholes model for stock price will suffice:

$$S_t = S_0 \exp\left((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right) \tag{124}$$

Let us also assume that there is a pre-determined time, T, at which we will close our position. This is a reasonable assumption to make, since we will have an

idea about the length of time for which gbm is valid.
We know then, that we can model the log of the final price normally:

$$\log(\frac{S_T}{S_0}) \sim \mathcal{N}\left((\mu - \frac{1}{2}\sigma^2)T, \sigma^2 T\right). \tag{125}$$

Therefore all that is left to do is define our question probabilistically in terms of the stock price. Since we aim to trade based off of the value of μ, we will look for a critical value μ^* that μ must exceed/fall short of, depending on whether we are intending to buy or sell, respectively.
We will formulate this based off of our risk tolerance, which can be derived directly from how we would like to profit. If we let the probability that we *do not* profit be p, then this equates to a $100(1-p)$ percent chance of profit. So we are interested in the following probability equation:

$$P\left[S_T < S_0\right] = p. \tag{126}$$

Given that we are trying to calculate a value for μ, we will have to provide the value of σ, which in practice will be taken to be our estimate $\hat{\sigma}$.

Now all that is left is to follow through with the calculations. Taking logs yields,

$$P\left[\log(S_T) < \log(S_0)\right] = p \tag{127}$$

Then we can rearrange,

$$\iff P\left[\log\left(\frac{S_T}{S_0}\right) < 0\right] = p, \tag{128}$$

and transform such that we are left with a statement regarding a standard-normal random variable. Letting $m = (\mu - \frac{1}{2}\sigma^2)T$ and $v = \sigma^2 T$

$$\begin{aligned} p &= P\left[\log\left(\frac{S_T}{S_0}\right) < 0\right] \\ &= P\left[\frac{\log\left(\frac{S_T}{S_0}\right) - m}{\sqrt{v}} < -\frac{m}{\sqrt{v}}\right] \\ &= P\left[Z < -\frac{m}{\sqrt{v}}\right]. \end{aligned} \tag{129}$$

Which is now precisely statement of the standard normal cumulative distribution function, $\Phi(x)$, and can be re-written as,

$$p = \Phi\left(-\frac{m}{\sqrt{v}}\right). \tag{130}$$

From here we will make use of the inverse cumulative function, with the understanding that this is easily programmed into python.

$$\begin{aligned} \Phi^{-1}(p) &= -\frac{m}{\sqrt{v}} \\ \iff \Phi^{-1}(p) &= -\frac{(\mu^* - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}. \end{aligned} \tag{131}$$

Simplifying and rearranging yields the following formula for μ^*:

Proposition 2. *The critical value, μ^*, required to satisfy a risk tolerance p, is given by:*

$$\mu^* = \frac{1}{2}\sigma^2 - \frac{\sigma}{\sqrt{T}}\Phi^{-1}(p) \tag{132}$$

where one should buy when $\mu \geq \mu^*$ and sell when $\mu \leq -\mu^*$ □

5.4 Implementation

Finally, we have reached the point where all of our hard work can be implemented into a fully automatic, dynamic, real-time quantitative trading algorithm! It must be made clear that this is a first edition, and to use our framework with with money will require further iterations and risk-assessment.

5.4.1 The Backtrader Library

One of the reasons we made the switch to python was so we could make use of the many useful libraries it has access to. The most helpful that we found was the Backtrader library [4]. It is, as they describe, "A feature-rich python framework for back-testing and trading". It has every elementary feature that one could ask for in a back-test environment, incorporating a virtual broker system that simulates the order placement of trades, not just the long/short positions. This makes our testing more robust as we are also accounting for real-world problems such as slippage and order-rejection in our testing process.

To learn more about how the library functions, please refer to the documentation referenced.

5.4.2 The Algorithm

We will now outline the basic algorithm that we used in this paper. It operates in the following way:

1. Learn the parameters based on the first N points of data.
2. If $\mu > 0$ and exceeds the critical value μ^*, BUY. If $\mu < 0$ and $\mu < \mu^*$, SELL. Else, do not execute any trades until the next data point.

3. If you are holding a position and a reverse point is detected at 95% level, close the position.

4. REPEAT steps starting from next data point.

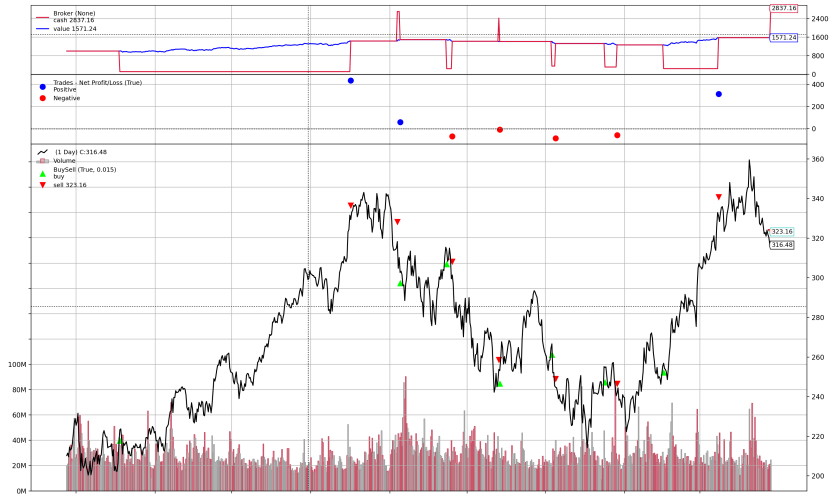
As you can see, the algorithm itself is not very complicated, but relies heavily on the mathematical model and change-point detection that we have developed. We have also briefly experimented with incorporating a stop loss, but as it turns out, our change point detection is currently strong enough such that this doesn't make much difference.

There are of course endless possibilities for tweaks and tinkering that we do not dive into in this paper. But the reader should note that this is by no means a final product. After discussing the results of this algorithm, we will briefly explore the different directions/refinements that could be incorporated to make it better.

5.4.3 Some practical examples

At last, the time has come to try out the algorithm on some stocks. There are many different ways for us to do this. Using yahoo finance, we have access to the majority of open/close data ranging from yearly refinement down to 1 minute intervals. Unfortunately, down at the 1-minute level, we can only access a maximum of the past 4-weeks of data. This is an issue when it comes to the robustness of our back-testing ability.

Here is what it looks like in practice. For illustrative purposes, we have provided trading data for "MSFT" stock between August 2020 and August 2023. The reason for the long time-frame is primarily because the back-trader illustrations are much more interpret able on daily data, since they operate on a daily data-time system. It makes it a lot easier to qualitatively see where trades are being made.



Explanations are in order. Over the 3-year period, Microsoft stock went from \$210 per share, to \$316 per share; a 50.5% increase. We chose a starting portfolio value of \$1000 and traded a consistent 4 shares per trade, finishing with a final portfolio value of \$1571; a 57.1% increase. So in this specific case we managed to beat the stock (just)! But it is also worth mentioning that throughout the same period of time, the S&P500 only returned 30% on your investment!

Taking a look at the graph, we will explain some of its details. As you may have guessed, the jagged black line running through the centre is the stock price, with the price labelled on the right-hand y-axis. Along the line, green and red triangles can be seen. These represent BUY/SELL orders that have been executed. They happen sequentially, and half of them signify a closing of the currently open LONG/SHORT positions. I.e, if you have taken a short position, the graph will show a red triangle; the next triangle will always be green, and it will signify a BUY order to close your current position. After that green triangle, you will currently hold no position until the algorithm decides to make another order.

Above the stock price, you can see the results of the trades being made. As labelled on the diagram, a blue circle indicates a profitable trade has been closed, and a red circle indicates a losing trade. The section above that one is a running valuation of the portfolio, along with the amount of cash it has access to (*due to the complex nature of short positions, these are far from the same thing*).
Finally, the bottom of the diagram shows the trading volume on that time interval, labelled on the left-hand y-axis. However, we did not make use of this

data in this current iteration.

5.4.4 Results

We mainly focused on minute-minute data throughout our testing, as this fits with our hypothesis that the model remains constant on small time periods. Over the 3-week period from 2023-08-07 to 2023-09-25, we back-tested our algorithm on hundreds of different stocks to assess how it performed under the most recent market conditions.

There were some notable winners such as 'NIO' which achieved a staggering 12.4% over this period as well as 'MNST' (Monster) which achieved a respectable 4%. This goes along with many many other stocks that did profit under this strategy, although the gains were comparably small (in the <1% region). Of course there were also cases where the strategy took losses, and so, through the hundred or so stocks that we tested, we found an average profit of -0.25% - which is calculated relative to the initial price of the stock.

Was this what was expected? Yes, in fact this, for us is a very promising result. The algorithm being used so far is extremely elementary, and essentially trades off a single stochastic indicator. In the realm of day trading, this is a really stupid thing to do! The reason this is promising is because there are numerous things that we can add that would play a part in the following:

- **Maximising Profits**
- **Minimising Losses**

Ultimately, these two fundamental concepts are what trading is built upon. Our model may be perfect, and our indicators may be great, but if we ineffectively leverage these instruments, we will find ourselves stuck in the mud. We will cover some of the potential improvements in the next section.

These results may seem concrete, but there are a few things still to consider.

- **Leverage.**

In our testing, we only used 1 of each stock to buy and sell. Of course, if this is all you can afford then that's a good representation of what one could expect. However, this is ignoring the fact that while trading, you have access to an entire realm of financial instruments, one of which is called leverage.

Leverage can be achieved in numerous different ways that we will not discuss, but it effectively allows a trader to trade with more money than they actually have; up to 100x for retail traders under some brokers! Now this changes the game completely. It means that if you were initially trading one stock at a time, costing $100 per share, then you could now trade up to 100 shares for the same

initial cost! For most of you, this may sound too good to be true, and that's because it is. The caveat is that because you now hold 100x the number of shares, you can incur 100x the amount of loss that you would have previously. We illustrate this with an example:

Suppose you have $150 in your trading account and you wish you trade a stock that is currently offering $100.00/share. Without leverage you can buy only one stock. With 100x leverage, you pay $100.00 to (loosely) own 100 shares temporarily; this can typically be done through a Contract For Difference (CFD). Lets say person A completes the trade without leverage, and person B completes the trade with 100x leverage. We document this:

- Person A: **Shares:** 1, **Cash: $50**
- Person B: **CFD Shares:** 1, **Cash: $50**

We now suppose that there's been a poor earnings report for the company in question. In a freight of panic, many short-term investors sell their shares/ take short positions. Very quickly the stock plummets 10% down to $90/share. Person A and Person B both close their positions at this new price and are left with the following portfolios:

- Person A: **Shares:** 0, **Cash: $140**
- Person B: **CFD Shares:** 0, **Cash: -$850**

Devastating! Person A lost $10 on from their portfolio, Person B lost $100 \times \$10 = \$1000$, and has been left with a severely negative account balance. Hopefully this illustrates the dangers present in using leverage.

So how does this apply to our trading strategy, won't we just lose more money? It's not that simple. Using leverage would effectively change the way in which we handle risk. Instead of looking for a strategy that maximises the upside profits: i.e entire percent's per week, our strategy would aim to merely skim a tiny percent of profit (based solely off the stock price), but in making this sacrifice, overly minimise the losses of any single trade. This means that we can avoid the leverage from making us bankrupt whilst 'scalping' small percentage profits and effectively multiplying them by a number of our choice. This is the essence of high frequency trading.

So to clarify, we are really just looking for a method that can win/lose 51%/49% of the time with near **certainty**. As opposed to finding a strategy that wins 75% of the time with a great **uncertainty**.

5.5 Possible Improvements

We have already opened the discussion throughout this paper on the improvements that can be made. Here we will break these down into four different categories and provide an outline for each. These categories are:

- **Model Selection**
- **Statistical Methods**
- **Algorithm & Trading Methodology**
- **Machine Learning, A.I, Data & More Data!**

5.5.1 Model Selection

The model we have chosen to follow in this paper is a slight alteration of the standard Black-Scholes model. We have made the assumption that given a small enough time frame, the Black-Scholes model is a good fit. In addition to this, our 'step-function' method occurs dynamically; we do not predict the changes in parameters μ and σ in the future, which means that we can only make prediction's on a small time frame. This isn't necessarily an issue, but perhaps attempting to change this could produce useful results.

A big problem with the model is that it imposes a Markovian property on stock-price. This is inherited from the Brownian motion that underpins it, which holds the property that for $0 < t_1 < t_2 < t$ and Brownian motion W_t, we have that the increments,

$$W_t - W_{t_2}, \;\; W_{t_2} - W_{t_1}, \tag{133}$$

are independent. This is implying that previous sections of stock price **do not** have any effect on current price! If you consider this for a moment it is truly absurd. A possible fix to this would be to consider Fractional Brownian Motion (FBM), in which the increments are not independent. Instead they are related to each other by a strength known as the Hurst factor.
Other possibilities include incorporating a stochastic element into the parameters so that we would have μ_t and σ_t. An example could be an SDE of the form,

$$dS_t = \mu_t S_t dt + \sigma_t S_t dW_t. \tag{134}$$

5.5.2 Statistical Methods

In this paper we focused primarily on Maximum Likelihood Estimation for estimating our parameters. Whilst this could provide us with an estimate for the uncertainty in our estimation (thanks to distribution theory), we had no choice but to use it as a point estimator. We have seen already that due to the noisy data (large σ), there was a significant variation in the estimation of μ. A hurdle to this was of course our lack of resolution in the price data we could obtain.

In a very liquid stock or currency pair, there could be hundreds of 'ticks' per second. This would mean we could theoretically access over 1000 data points per minute! So it is important to take this into consideration when discussing the efficacy of the methods.

Nonetheless, the market is still noisy. An approach that could help us overcome this is the Bayesian paradigm. In this, we can incorporate our prior beliefs about the parameters - which may be influenced through various external factors such as perceived market sentiment, financial reports, elections and so on. Initially this would be performed quite heuristically, but further down the line, incorporating machine learning in this context is certainly possible.

A final note must also be made on testing the efficacy of the algorithm. In this paper we did not statistically analyse the results of our algorithm in great detail. With something like this it's not as simple as to say 'the algorithm works' just because we have seen profitability in some stocks. Moving forward we will need to perform more robust tests to try and deduce *when* the conditions are right for the algorithm to work. As an example, one thing we could look at would be the number of winning trades to losing ones.

5.5.3 Algorithm & Trading methodology

We have already discussed some of the shortcomings of our algorithm. Here we will discuss in more detail what went well, and what can be changed for the better.

Change point detection
The change point detector underpins the entire algorithm. This is a weakness in our strategy as it makes it one-dimensional and thus less resilient. In addition to this, we have already discussed the probabilistic downfall of this method. A slick fix to this would be to require 2 change points in a row, both of which falling outside the previous expectations of the model. This greatly reduces the probability that we falsely detect a change-point and exit our trade early.

Riding winners
This leads directly into the next criticism of the algorithm which is its elementary exit strategy. Currently, if we are holding a position and a change point is detected, our algorithm will exit the trade. This is too strict, as there is the potential that we have just exited a big winner. Instead, this change point should be taken as a warning, and if we continue to lose then we should exit. This strategy enables some leeway in the prediction accuracy, which is necessary given the uncertainty in our decisions.

In addition to this, when we make a successful decision, we can use our change-point method to add more to winning trades. For example if we have a long position, and the price jumps even higher outside of the upper prediction

interval, we could perceive this is a form of momentum, and add more to our position. Here we are illustrating the concept 'maximising winners'.

5.5.4 Machine Learning, A.I, Data & More Data!

Throughout this paper we have mentioned how ML and A.I could be a useful contributor to our trading strategy; here we will outline some specific use cases.

Parameter Estimation
In our current method, we are limited by the number of price-data points we have access to in real-time to estimate the Black-Scholes parameters. However, it is possible that there are patterns in the occurrence of these parameter values that we may be able to identify with machine learning. For example, going through historical data, we can in hindsight provide a value for μ in a given time interval. We can then use this as a response variable in an Artificial Neural Network (ANN). It is then a case of finding appropriate predictor variables such as trading volume, time of day, day of the week, weather... and enabling the ANN to try to learn a pattern in the values. If we can have *any* level of predictive accuracy in what the parameter values are going to be before they occur, we can make more informed trading decisions.

Sentimental Analysis
In the previous subsections we discussed the possibility of our algorithm functioning better in different market conditions. The question then, is how do you define a market condition? One possible answer to this is the quantification of a market sentiment. With new A.I language models such as Chat GPT, it is becoming ever more possible to use computers to analyse text data. For example, at the start of each trading session, an A.I could be used to give an indication on how the markets will operate on a given day. Perhaps there will be certain sentiments under which our strategy is more profitable, and some that we should avoid.

A.I Trading Bot
These ideas can be taken even further, and in the limit of quantitative trading we could aim to have an A.I that simply learns *how* to trade. It is unclear to us exactly how this would be done as of now, however incorporating some well known techniques such as LSTM models (Long-Short Term Memory) for time series analysis could be a direction to look into.

Data
What is certainly clear is that we must try to incorporate more data into our model. Whether we try to use A.I or not, it only makes sense to try and incorporate as much information as possible into our trading decisions. Building a mathematical model that incorporates some of the ideas we have already talked about, inevitably requires us to consider more than just price. Furthermore

the quality of this data is important. Access to live order flow would be the gold standard for a quantitative approach like ours, as there is no hope to be competitive when we are fighting with blunted tools!

5.6 A Machine Learning Example

Here we will look into a practical example of how Machine Learning could be used to influence our trading decisions. We take a training set of data and attempt to learn patterns which can then be applied to a test set of data to yield better prediction results. For our uses, we would take past stock market data, train a model and then test it on some current or recent data with the hopes of predicting more accurately whether a stock is going up or down.

In this example we will look at a simple, but effective method known as the Random Forest method.

5.6.1 Random forests

A Random Forest is a Supervised machine learning algorithm (supervised referring to using a training set of data). It generates a forest of decision trees which upon each arc have random probabilities; the outcome of each decision tree counts as a vote for the decision it took, and hence when merged together will give a more accurate outcome than just randomly guessing. For us, random forests are extremely convenient, with them being a package in sklearn in python, they are easily implemented into code. They are commonly used in stock price prediction, loan repayments or in medicine to help detect diseases based on past medical history.

The code for our machine learning is provided below with comments explaining the methodology. Note that the code can be optimized with libraries, which also will make it more concise. Here we simply illustrate the concept.

The code is detailed below:

```
# Import the libraries that we need for machine
# learning and the algorithms

import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd
import yfinance as yf
import talib
import scipy.stats as stats
from sklearn.ensemble import RandomForestClassifier
```

We first alter our change point detector code from the previous section in order to provide a binary output which is compatible with a Random Forest machine learning model (7.6.1).

Next we define the stock data using Yahoo Finance and the test and train set to prevent leakage. Then removing unnecessary columns from the pandas data frame that could affect the efficiency and cause leakage.

```
stock=yf.Ticker('AAPL')
stock=stock.history(period='max')

stock['Tomorrow']=stock['Close'].shift(-1)
stock['Target']= (stock['Tomorrow']>stock['Close']).astype(int)
stock=stock.loc['1990-01-01':].copy()


train= stock.iloc[:-100]
test=stock.iloc[-100:]
```

We then wish to define a predict function which we will use to Train our model and then test the model that we have trained. We use the fit function which is built into SKlearn. In order to increase accuracy, we add the constraint of only giving a price move prediction if the model has greater than 60% certainty.

```
def predict(train, test, predictors, model):
    model.fit(train[predictors],train['Target'])
    preds=model.predict_proba(test[predictors])[:,1]
    preds[preds>=.6]=1
    preds[preds<.6]=0
    preds=pd.Series(preds,index=test.index,name='
        Predictions')
    combined=pd.concat([test['Target'],preds], axis=1)
    return combined

def backtest(data, model,predictors, start=2500,step=250)
    :
    all_predictions=[]

    for i in range(start, data.shape[0], step):
        train=data.iloc[0:i].copy()
        test=data.iloc[i:(i+step)].copy()
        predictions=predict(train, test ,predictors ,
            model)
        all_predictions.append(predictions)
    return pd.concat(all_predictions)
```

Once these functions have been defined, we can use the functions defined earlier to create predictors that the model can learn from. The functions we create are based on a simple moving average, a trend indicator and the change points. These predictors provide binary outputs which are accessible for the fit function to train the model on.

The code can be found in the appendix here (7.7). Note the changes to the change point detection function, we now produce a binary output of 1's and 0's: 1 - stock moving upwards, 0 - stock moving downwards.

Finally we define the predictors and drop 'n/a' data point values.

```
new_predictors += ['Close', 'Dividends','Stock_Splits',
    ratio_column, trend_column, changepointinfo_Column]

stock=stock.dropna()

model= RandomForestClassifier(n_estimators=200,
    min_samples_split=50, random_state=1)

predictions = backtest(stock, model, new_predictors)
```

We can now run the final piece of code and observe the results!

```
predictions['Predictions'].value_counts()
precision_score(predictions['Target'], predictions['
    Predictions'])
```

5.6.2 Results

Testing the model on Apple stock (AAPL), our average correct prediction rate was between 52% and 53.5% when using the pre-determined trend and ratio indicators. However, adding in the change-point function increased this to 54%-57%. This is a tangible increase and is good considering we are only considering the price data and fundamentals have not been considered. The main reason for the ineffectiveness is due to the nature of the change-point detector function (7.6.1). Although it works as required, it takes a minimum of 50 days of data to calculate the Black-Scholes parameters. This means at most there will be N/50 change-points where N is the number of days in the entire data set. If N is comparatively small, it is unlikely the model has a sufficient amount of data to make accurate predictions. A way to negate these issues would be to find minute data over a large period of time.

6 Conclusion

Beginning this paper, we aimed to build a strong foundation in SDE theory which we would then use to build our own financial model. This model would then form the core of an original, mathematically rigorous, quantitative trading algorithm.

The path we decided to take was to develop and improve upon the pre-existing Black-Scholes financial model. We addressed some of its drawbacks such as the assumptions of constant drift and volatility. We then successfully combined this model with statistical techniques such as Maximum Likelihood Estimation (MLE), and novel trading strategies, to create and back-test a quantitative algorithm in python. The algorithm was profitable in certain scenarios, achieving results of up to 12% profits in 3 weeks of trading.

Our work serves as a foundation for future development and uncovered many areas for improvement and further research. Specifically, it is apparent that a more drastic re-invention of the Black-Scholes model is necessary for trading; this could include ideas such as Fractional Brownian Motion (FBM) and utilisation of stochastic parameters. Furthermore, integrating more sophisticated machine learning methods is crucial, especially within the algorithm evaluation, as this will allow for a more concrete risk profile.

This concludes our paper, and we would like to give final thanks to Dr. Kohei Suzuki for his supervision and encouragement.

7 Appendix

7.1 Comparison of different schemes plot

```
bX <- expression((5 - 11 * x + 6 * x^2 - x^3))
x0 <- 5
DT <- 0.1
par(mfrow=c(4,3))
X <- sde.sim(drift=bX, delta=DT,X0=x0)
Y <- sde.sim(drift=bX, method="ozaki",delta=DT,X0=x0)
Z <- sde.sim(drift=bX, method="shoji",delta=DT,X0=x0)
plot(X,main="Euler")
plot(Y,main="Ozaki")
plot(Z,main="Shoji -Ozaki")
DT <- 0.25
X <- sde.sim(drift=bX, delta=DT,X0=x0)
Y <- sde.sim(drift=bX, method="ozaki",delta=DT,X0=x0)
Z <- sde.sim(drift=bX, method="shoji",delta=DT,X0=x0)
plot(X, main="Euler")
plot(Y,main="Ozaki")
plot(Z,main="Shoji -Ozaki")
DT <- 0.5
X <- sde.sim(drift=bX, delta=DT,X0=x0)
Y <- sde.sim(drift=bX, method="ozaki",delta=DT,X0=x0)
Z <- sde.sim(drift=bX, method="shoji",delta=DT,X0=x0)
plot(X, main="Euler")
plot(Y,main="Ozaki")
plot(Z,main="Shoji -Ozaki")
DT <- 1
X <- sde.sim(drift=bX, delta=DT,X0=x0)
Y <- sde.sim(drift=bX, method="ozaki",delta=DT,X0=x0)
Z <- sde.sim(drift=bX, method="shoji",delta=DT,X0=x0)
plot(X, main="Euler")
plot(Y,main="Ozaki")
plot(Z,main="Shoji-Ozaki")
```

7.2 Multiple GBM's in R

```
x=sde.sim(model = 'BS', t0 = 0, T = 1, X0 = 100,
N = 1000, theta = c(1, 0.2))
y=sde.sim(model = 'BS', t0 = 1, T = 2, X0 = tail(x, 1),
N = 1000, theta = c(-0.4, 0.5))
a=sde.sim(model = 'BS', t0 = 2, T = 3, X0 = tail(y, 2),
N = 1000, theta = c(0.2, 0.1))
```

```
b=sde.sim(model = 'BS', t0 = 3, T = 4, X0 = tail(a, 3),
N = 1000, theta = c(0.1, 0.2))

# Combine time points and values of x and y
combined_data <- data.frame(
  t = c(seq(0, 1, length.out = length(x)),
        seq(1, 2, length.out = length(y)),
        seq(2, 3, length.out = length(a)),
        seq(3, 4, length.out = length(b))),
  X = c(x, y, a, b)
)


# Plot the combined time series
plot(combined_data$t, combined_data$X, type = "l",
     col = "blue", xlab = "Time", ylab = "Value",
      main = "Combined Time Series",
      xlim = c(0, 4))
```

7.3 Exponential Synthetic Data in R

```
change1=rexp(1,0.7)
change2=change1+rexp(1,0.4)
change3=change2+rexp(1,1)
change4=change3+rexp(1,0.8)
vectchange=c(change1,change2,change3,change4)

x=sde.sim(model = 'BS', t0 = 0,
T = change1, X0 = 100, N = 1000,
           theta = c(rnorm(1,0.5,0.1),rnorm(1,0.2,0.5)))
y=sde.sim(model = 'BS', t0 =change1 ,
T = change2 , X0 = tail(x, change1), N = 1000,
           theta = c(rnorm(1,-0.4,0.2), rnorm(1,0.5,0.1)))
a=sde.sim(model = 'BS', t0 = change2,
T = change3, X0 = tail(y, change2), N = 1000,
           theta = c(rnorm(1,0.2,0.2), rnorm(1,0.1,0.05)))
b=sde.sim(model = 'BS', t0 = change3,
T = change4, X0 = tail(a, change3), N = 1000,
           theta = c(rnorm(1,0.1,0.3), rnorm(1,0.2,0.01)))

combined_data <- data.frame(
  t = c(seq(0, change1, length.out = length(x)),
        seq(change1, change2, length.out = length(y)),
        seq(change2, change3, length.out = length(a)),
        seq(change3, change4, length.out = length(b))),
```

```
  X = c(x, y, a, b)
)


# Plot the combined time series
plot(combined_data$t, combined_data$X, type = "l", col = "blue",
     xlab = "Time", ylab = "Value", main = "Combined Time Series",
     xlim = c(0, change4))
abline(v = c(change1, change2, change3), col = "red", lty = 2)
```

7.3.1 Final R Synthetic data

We parse through the code step by step for clarity.
Generate the random number of change points between 0 and 10 (inclusive) and then generate random exponential change points

```
library(sde)


num_change_points <- sample(0:10, 1)
num_change_points

change_points <- numeric(num_change_points)
prev_change_point <- 0

for (i in 1:num_change_points) {
  change_points[i] <- prev_change_point + rexp(1, rate = 0.7)
  prev_change_point <- change_points[i]
}
```

Initialize vectors to store simulated values and time points and then simulate the SDE's using the same methods as the previous models such that we are left with a matrix of times and values

```
simulated_values <- numeric(0)
time_points <- numeric(0)

for (i in 1:(num_change_points + 1)) {
  if (i == 1) {
    start_time <- 0
    end_time <- change_points[i]
  } else if (i == (num_change_points + 1)) {
    start_time <- change_points[i - 1]
    end_time <- tail(change_points, 1) + rexp(1, rate = 0.8)
  } else {
    start_time <- change_points[i - 1]
    end_time <- change_points[i]
```

```
  }

  segment_sim <- sde.sim(model = 'BS', t0 = start_time,
              T = end_time,
              X0 = ifelse(i == 1, 100, tail(simulated_values, 1)),
              N = 1000, theta = c(rnorm(1, 0.1, 0.3),
              rnorm(1, 0.2, 0.01)))
  simulated_values <- c(simulated_values, segment_sim)
  time_points <- c(time_points, seq(start_time, end_time,
  length.out = length(segment_sim)))
}

combined_data <- data.frame(t = time_points, X = simulated_values)
combined_data <- combined_data[order(combined_data$t), ]
```

finally plotting the graph of the synthetic data

```
plot(combined_data$t, combined_data$X, type = "l", col = "blue",
     xlab = "Time", ylab = "Value", main = "Combined Time Series",
     xlim = c(0, max(combined_data$t)))
for (cp in change_points) {
  abline(v = cp, col = "red", lty = 2)
}
```

7.4 Python synthetic data

```
import numpy as np
import matplotlib.pyplot as plt
from SDE_sim_practice import gbm


def gbm(mu, sig, S0, T, N, seed):
    """ A function to generate a geometric brownian
        motion,
    given the parameters:
    mu, sig - as drift and volatility
    S0: Starting price
    T: End time
    N: Number of discrete steps

    returns: Numpy array of time steps and stock price.
        """
    # Initialise the parameters
    np.random.seed(seed) # setting the seed
    Wt = np.array([0.0 for i in range(N)])
    delta_t = T / N
```

```
    sd = np.sqrt(delta_t)
    t = np.array([j * delta_t for j in range(N)])
    # Generate an array for the brownian motion values
    for i in range(0, N - 1):
        z = np.random.normal(0,1)
        z *= sd
        Wt[i + 1] = Wt[i] + z # update step
        # print(Wt[i + 1])

    St = S0 * np.exp((mu - 0.5*sig**2)*t + sig * Wt)
    return np.array([t, St])


def multiple_gbm_sew(mu_arr, sig_arr, S0, T_arr, N_arr,
    seed = 0):
    """ A function to generate a combined time series of
        mutiple GBM's
    given the parameters and lengths as arrays themselves
        .

    mu_arr: Array of drift values
    sig_array: Array of volatility values
    S0: Starting price
    T_arr: Array of termination times
    N_arr: Array of step numbers"""
    n = len(mu_arr)
    St_arr = []
    t_arr = []
    t_sum = 0

    for i in range(n):
        # Generate the time series needed
        St = gbm(mu_arr[i], sig_arr[i], S0, T_arr[i],
            N_arr[i], seed)
        # Append to the combined array
        St_arr = St_arr + list(St[1])
        t_arr = t_arr + list(St[0] + t_sum)
        t_sum += T_arr[i]
        m = len(St_arr)
        # Define the new starting point for the next gbm
        S0 = St_arr[m-1]

    return np.array([t_arr, St_arr])


def N_vec(cps, N):
```

```
    """A function to generate the number of discrete
        steps
    in a gbm that is proportional to its relative size in
         the overall
    time series.
    returns: A list of N values

    Parameters:
       cps: change points
        N : Total number of time steps"""
    N_arr =[]
    n = len(cps)
    T_arr = list(cp_converter(cps))
    T_max = cps[n-1]
    for i in range(n):
        N_arr.append(int(np.floor((T_arr[i]/T_max)*N)))

    return N_arr


def cp_converter(cps):
    """A function that takes in the change points of a
        stock
    and returns an array of time-values to be used in
        gbm_sew"""
    n = len(cps)
    t_arr = [np.round(cps[0], 2)]
    diff = 0
    for i in range(n-1):
        diff = cps[i + 1] - cps[i]
        t_arr.append(np.round(diff, 2))
    return t_arr


def changepoints(n):
    """ A function to generate n, exponentially
        distributed change points"""
    vect_changepoints = [np.random.exponential(0.1*np.
       random.uniform(1,10))]
    for i in range(1, n):
        next_changepoint = vect_changepoints[i-1] + np.
            random.exponential(0.1 * np.random.uniform(1,
            10))
        vect_changepoints.append(next_changepoint)

    # print('vector of changepoints:', vect_changepoints)
```

```
    return vect_changepoints


def param_generator(n):
    """A function to generate n sets of mu and sigma
       parameters from a normal distribution"""
    np.random.seed()
    mu = []
    sig = []
    for i in range(n):
        mu.append(np.random.normal(0, 0.4)) # Random mu
            value from normal distribution with mean 0 and
             std deviation 0.4
        sig.append(abs(np.random.normal(0.2, 0.1))) #
            Random positive sigma value from normal
            distribution with mean 0 and std deviation 0.2
        # print(empty_matrix)

    return np.array([mu, sig])
```

The functions can then be implemented in the following, compact way:

```
    N = 10000
num_param = 5
params = param_generator(num_param)
cps = changepoints(num_param)
t_arr = cp_converter(cps)
N_arr = N_vec(cps, 1000)
mu_arr = list(params[0])
sig_arr = list(params[1])
S0 = 10
St = multiple_gbm_sew(mu_arr, sig_arr, S0, t_arr, N_arr)
plt.plot(St[0], St[1], color = 'black')


for cp in cps:
    plt.axvline(cp, color = 'red')
plt.show()
```

7.5 Confidence interval

```
def confIntCalc(X, log_X, N, k, i, mu, sig, alpha,
    postive = True):
    if postive:
        if ((log_X[(k + 1) * N + i + 1] < log_X[(k + 1) *
            N + i] + mu - stats.t.ppf(1 - (alpha) / 2,len
```

```
            (X)) * sig * np.sqrt(1 + 1 / max(len(X), 1))))
            :
            return True

        else:
            return False
    else:
        if log_X[(k + 1) * N + i + 1] > log_X[(k + 1) * N
            + i] + mu + stats.t.ppf(1 - (alpha) / 2, len(
           X)) * sig * np.sqrt(1 + 1 / max(len(X), 1)):
            return True

        else:
            return False
```

7.6 Change point detection

7.6.1 Change points

```
def changepoint_detector(combined_data, N,alpha):
        X = combined_data[1][:N]
        log_X = np.log(combined_data[1])
        k = 0
        j=0
        step_func = np.zeros((1, 3))
        change_points = []
        Params = MLE_finder(X)

        for i in range(len(combined_data[0]) - N - 1):
            if ((k+1)* N + i + 2 > len(combined_data[0])):
                break
            if (log_X[(k+1)*N + i + 1] <=
            log_X[(k+1)*N+ i]
            + Params[0] + stats.t.ppf(1-(alpha)/2 ,
            len(X)-1)*Params[1]*np.sqrt(1+1/max(len(X),1))
                and (log_X[(k+1)*N + i + 1] >=
                log_X[(k+1)*N+ i]
                + Params[0] - stats.t.ppf(1-(alpha)/2 ,
                len(X)-1)*Params[1]*np.sqrt(1+1/max(len(X),1)))):
                 X = combined_data[1][k*N:(k+1)*N + i + 1]
                 Params = MLE_finder(X)
                 step_func[j]=(Params[0],Params[1],
                 combined_data[0][N+i])
            else:
                 change_points.append(combined_data[0][(k+1)*N
```

```
                + i + 1])
                k+=1
                X = combined_data[1][k*N + i: (k+1)*N + i]
                j+= 1
                Params=MLE_finder(X)
                new_row=np.array([Params[0],Params[1],
                combined_data[0][(k)*N+i]])
                step_func=np.vstack([step_func,new_row])
        return change_points , step_func
```

7.6.2 Reverse points

```
def generate_vector(length):
    return np.ones(length)
    # Return an array of ones with the specified length

def generate_alternate_vector(length):
    return np.zeros(length)
    # Return an array of zeros with the specified length

def changepoint_detector(combined_data, N, alpha):
    X = combined_data[:N]
    log_X = np.log(combined_data)
    k = 0
    j = 0
    step_func = np.zeros((1, 3))
    change_points = []
    params = MLE_finder(X)
    upper = 0
    lower = 0

    for i in range(len(combined_data) - N - 1):
        # Define the parameters
        mu, sig = params[0], params[1]

        if (k + 1) * N + i + 2 > len(combined_data):
            # Break the loop if there's not enough data
            break

        upper = log_X[(k + 1) * N + i] +
        mu + stats.t.ppf(1 - (alpha) / 2,
        len(X)) * sig * np.sqrt(1 + 1 / max(len(X), 1))

        lower = log_X[(k + 1) * N + i] +
        mu - stats.t.ppf(1 - (alpha) / 2,
```

```
len(X)) * sig * np.sqrt(1 + 1 / max(len(X), 1))

if upper >= log_X[(k + 1) * N + i + 1] >= lower:
    # print('safe', i)

    X = combined_data[k * N:(k + 1) * N + i + 1]
    params = MLE_finder(X)
    step_func[j] = (params[0], params[1], (k+1)*N
        +i)

# Check if mu is positive or negative
elif mu > 0:
    if confIntCalc(X, log_X, N, k, i, mu=params
        [0],
    sig=params[1], alpha=alpha, postive=True):
        # Then we have detected a change point
        #and we run the following code:

        change_points.append(combined_data.index
            [(k + 1) * N +
        i + 1])
        k += 1
        X = combined_data[k * N + i: (k + 1) * N
            + i]
        j += 1
        params = MLE_finder(X)
        new_row = np.array([params[0], params[1],
             k * N + i+1])
        step_func = np.vstack([step_func, new_row
            ])

    else:
        # print('safe', i)
        X = combined_data[k * N:(k + 1) * N + i +
             1]
        params = MLE_finder(X)
        step_func[j] = (params[0], params[1], (k
            +1)*N + i)
elif mu < 0:
    if confIntCalc(X, log_X, N, k, i, mu=params
        [0],
    sig=params[1], alpha=alpha, postive=True):
        # Then a change point has been detected
        #and run the following code
        change_points.append((k + 1) * N + i + 1)
        k += 1
```

```
                X = combined_data[k * N + i: (k + 1) * N
                    + i]
                j += 1
                params = MLE_finder(X)
                new_row = np.array([params[0], params[1],
                     (k) * N + i+1])
                step_func = np.vstack([step_func, new_row
                    ])

            else:
                # We are still safe
                # print('safe', i)
                X = combined_data[k * N:(k + 1) * N + i +
                     1]
                params = MLE_finder(X)
                step_func[j] = (params[0], params[1], (k
                    +1)*N+i)

    if step_func[0][0] >= 0:
        vectofindex = []
        for i in range(1, len(step_func.transpose()[2]) -
             1):
            if i % 2:
                vectofindex.append(generate_vector(int(
                    step_func.transpose()[2][i] -
                    step_func.transpose()[2][i - 1])))
            else:
                vectofindex.append(
                    generate_alternate_vector(int(
                    step_func.transpose()[2][i] -
                    step_func.transpose()[2][i - 1])))
    elif step_func[0][0] < 0:
        vectofindex = []
        for i in range(1, len(step_func.transpose()[2]) -
             1):
            if i % 2:
                vectofindex.append(
                    generate_alternate_vector(int(
                    step_func.transpose()[2][i] -
                    step_func.transpose()[2][i - 1])))
            else:
                vectofindex.append(generate_vector(int(
                    step_func.transpose()[2][i] -
                    step_func.transpose()[2][i - 1])))
    returnvect=[]
    for i in range(len(vectofindex)):
```

```
        for j in range(len(vectofindex[i])):
            returnvect.append(vectofindex[i][j])



    # print("Change Points:", change_points)
    # print("Step Functions:")
    # print(step_func)
    return change_points, step_func, returnvect
```

7.7 Predictors

```
horizons=[2,5,60,250,1000]
intervals=[10,30,50]
new_predictors=[]

for horizon in horizons:
        rolling_averages=stock.rolling(horizon).mean()

        ratio_column= f'Close_Ratio_{horizon}'
        stock[ratio_column]= stock['Close']/
            rolling_averages['Close']

        trend_column=f'Trend_{horizon}'
        stock[trend_column]=stock.shift(1).rolling(horizon
            ).sum()['Target']

for interval in intervals:
        Mu_column = f'Mu_{interval}'
        Sig_column = f'Sig_{interval}'
        lst_mu=[]
        lst_sigma=[]
        for i in range(len(stock['Close'])):
            start_idx = max(i - interval, 0)
            end_idx = i
            if start_idx == 0:
                lst_mu.append(np.nan)
                lst_sigma.append(np.nan)
            else:
                lst_mu.append(MLE_finder(stock['Close'][
                    start_idx:end_idx])[0])
                lst_sigma.append(MLE_finder(stock['Close'
                    ][start_idx:end_idx])[1])
        stock[f'Mu_{interval}'] = lst_mu
        stock[f'Sig_{interval}']=lst_sigma
```

```
X=changepoint_detector(stock['Close'], 50, 0.05)
for i in range(len(stock['Close'])-len(X[2])):
    X[2].append(X[2][-1])
changepointinfo_Column=f'changepointinfo_Column'
stock[f'changepointinfo_Column']= X[2]
```

References

[1] AMF. Market participants.

[2] Investopedia. History of the black-scholes model.

[3] Kevin Mekulu. Option pricing using monte carlo simulations, 2020.

[4] Rodriguez. Backtrader documentation. 2020.

[5] Wikipedia. Origins of stochastic differential equations.