MA323 Course Project

Report

Candidate Number: 40946

Date Submitted: 04/05/2020

Time Submitted: 19:00 GMT

Problem 1.

1.1.

We note than for f to be a probability density function, it must integrate over the range of $x \in [0,1]$ to yield a value of 1, i.e. we must have $\int_0^1 f(x)\,dx = 1$.

Thus

$$f: [0,1] \to \mathbb{R}, f(x) = \alpha(x - 2x^2 + x^3)$$

$$1 = \int_{-\infty}^{\infty} f(x)dx = \int_0^1 \alpha(x - 2x^2 + x^3)dx = \alpha\left[\frac{x^2}{2} - \frac{2}{3}x^3 + \frac{x^4}{4}\right]_0^1 = \frac{\alpha}{12}$$

$$\Rightarrow \alpha = 12$$

1.2.

To generate a sample from f using *Von Neumann's acceptance rejection method*, we must first specify a distribution g and $c > 0$ s.t. $f(x) \leq cg(x)$ for all $x \in [0,1]$. A logical choice for g is the $Uniform[0,1]$ distribution, due to its equivalent range for x and simplicity for sampling values. Whilst we could theoretically take any $c > 0$ s.t. the above condition holds, we wish to pick c which maximises the acceptance region of the algorithm. Thus, we take $c = \max\{f(x) : x \in [0,1]\}$.

Calculating C:

$$C = \max\{f(x) : x \in [0,1]\}$$

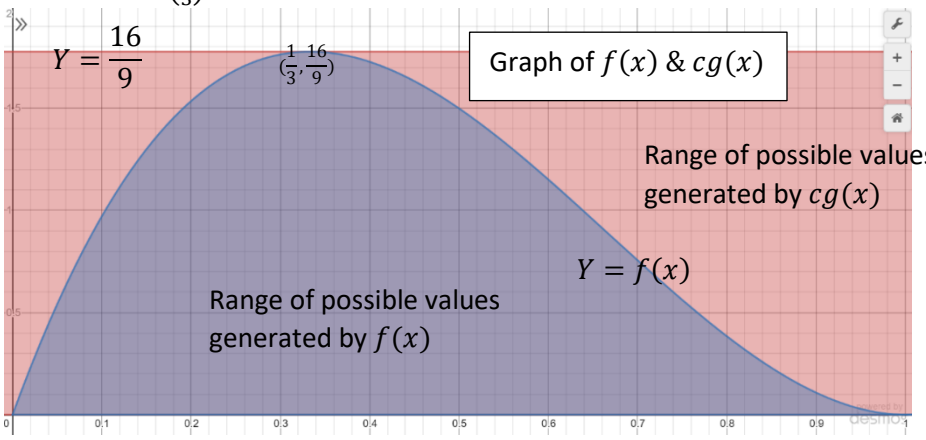
$$FOC: 0 = f'(x) = 12(3x^2 - 4x + 1)$$

$$\Rightarrow 0 = x^2 - \frac{4}{3}x + \frac{1}{3} = \left(x - \frac{2}{3}\right)^2 - \frac{1}{9} \Rightarrow x = \frac{2 \pm 1}{3}$$

$$\Rightarrow x = 1 \text{ or } x = \frac{1}{3}$$

We note $f(1) = 0$ & therefore must correspond to a local minimum on $x \in [0,1]$ as $f(x) \geq 0$.

However, we note $f\left(\frac{1}{3}\right) = \frac{16}{9}$ which we can deduce as the local maximum on [0,1] either by taking the second derivative, $f''\left(\frac{1}{3}\right) = -24 < 0$, or graphically as seen below.



Thus, we have $f(x) \leq \frac{16}{9} = \frac{16}{9} * \frac{1}{1-0} = \frac{16}{9} g(x)$, s.t. $g \sim uniform[0,1]$ & $c = \frac{16}{9}$.

Our *Von Neumann* algorithm is then rather simple:

1. Generate U_1 & U_2 independently from $uniform[0,1]$.
2. If $U_2 \leq \frac{f(U_1)}{cg(U_1)} = \frac{f(U_1)}{cU_1}$, then accept $U_1 = X$ and return it. Otherwise, return to step 1.

We note that given g, the best possible proportion of numbers our Von Neumann algorithm accepts (for a large enough sample) is

$P(We\ Accept\ U_1) = P(U_1 = X) = P\left(U_2 \leq \frac{f(U_1)}{cg(U_1)}\right) = \frac{1}{c} = \frac{9}{16} = 56.25\%$

with the final equality given in (2.5) in the lecture notes.

1.3.

If we wish to apply the *inverse transform method* to sample from f, we note we must first calculate the inverse of the cdf of f. Calculating the CDF is easy enough

$$F(x) = 12 \int x - 2x^2 + x^3 \, dx = 6x^2 - 8x^3 + 3x^4 + C$$

Where it is trivial to see $C = 0$. However, inverting F to find $F^{-1}(u) = x$ is less simple, as $F(x)$ is quartic polynomial and will therefore have 4 roots. Whilst not all these roots will lie within our range of $x \in [0,1]$, it is no longer trivial to solve. Further analysis reveals that $f(x) \equiv pdf\ of\ Beta(2,5)$ and therefore $F(x) = I_x(2,5)$ -> an example of the regularised incomplete beta function, which itself has no analytical inverse solution. Thus, in order to use the *inverse transform method,* we must first also apply a numerical method for solving $F^{-1}(u) = x$ as a part of our algorithm. A suitable numerical method we could apply to calculate this inverse, would be *Newton's method,* as $F\ is\ C^1$ and we are given $F'(x) = f(x)$, allowing us calculate the recursive formula for x:

$$x_{n+1} = x_n - \frac{F(x_n) - u}{F'(x_n)}$$

Therefore, to use the inverse transform method in this case, we have the following algorithm:

1. Generate $U \sim Uniform[0,1]$
2. Use Newton's Method to gain a numerical approximation for the solution to $x = F^{-1}(U)$
3. Return x

However, we note that relative to the *Von Neumann's method,* the *inverse transform method* introduces additional error into your estimates of the distribution of x, through to the use of the numerical approximation of $F^{-1}(U)$, whilst also being more computationally intensive to compute due to the requirement of an additional for-loop to iterate through the numerical scheme -> comparing step 2 of both processes together, we note that the *Von Neumann's* method's is of $O(1)$ vs $O(n)$ for the *inverse transform's.* Furthermore, as we have $P(x \in A \,|U \leq \frac{f(x)}{cg(x)}) = \int_A f(x)\, dx$ given by (2.6) in the lecture notes, the *Vonn Neumann* method is also an exact sampling scheme, leading us to conclude it to be more suitable in this case.

Alternatively, recognising $x \sim Beta(2,5)$, we can also generate a sample of x using the following method:

1. Generate $W \sim \Gamma(2,1)$ and $Y \sim \Gamma(5,1)$ such that W and Y are independent
2. $x = \frac{W}{W+Y}$
3. Return x

We note, as $\frac{W}{W+Y} \sim Beta(2,5)$ this has the advantage of being an exact scheme, and whilst we do have the additional step of generating W & Y, this simply corresponds to generating and summing 2 and 5 $\exp(1)$ variables respectfully, which are themselves highly computationally simple to generate. Thus, this scheme has the advantage of accepting all samples generated, relative to only 56.25% (at best) of *Von Neumann method* whilst being more computationally efficient than the *inverse transform method.*

Problem 2.

$$I_1 = \int_a^b f(x)\,dx \qquad I_2 = \int_{-\infty}^{\infty} g(x)\,dx$$

2.1.

We observe that applying a numerical scheme to approximate I_1 is rather simple, with most using the fact that we can decompose the integral, from a to b, into smaller sub-integrals, x_{i-1} to x_i, and then approximating the function f by some simpler polynomial p_m of order m. We note that whilst high order polynomials can be used, which leads to a better approximation of f on the sub-interval, these come with the trade off of greater computational complexity, whilst also only improving marginally upon the estimates of a lower m.

The trapezoidal rule is one such numerical scheme, which uses the approximation $f(x) \approx p_1(x) = x\frac{f(c)-f(d)}{c-d} + \frac{cf(d)-df(c)}{c-d}$ for the sub-interval $[c,d]$, i.e. a degree 1 polynomial.

Thus partitioning $[a,b]$ into n equal sub-intervals $[x_{i-1}, x_i]$ we obtain:

$$\begin{aligned} I_1 = \int_a^b f(x)\,dx &= \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} p_1(x)\,dx \\ &= \left(\frac{b-a}{n}\right)\left(\frac{f(a)-f(b)}{2} + \sum_{i=1}^{n-1} f\left(a + \frac{i(b-a)}{n}\right)\right) \end{aligned}$$

This sum can be easily computed using a for-loop in C++ (or another suitable language), yielding a numerical approximation for I_1.

A numerical approximation of I_2 is also possible, however it requires us to first implement a change of variable/substitution $h(t) = x$, in order to covert it from an improper integral and into one of the type I_1, which can then be calculated as outlined above.

2.2.

We have

$$I_1 = \int_a^b f(x)\,dx = \int_a^b \frac{b-a}{b-a} f(x)dx \quad dx = (b-a)\int_a^b f(x)h(x)\,dx = (b-a)E_h[f(x)]$$

&

$$I_2 = \int_{-\infty}^{\infty} g(x)\,dx = \int_{-\infty}^{\infty} \frac{g(x)}{\emptyset(x)}\emptyset(x)\,dx = E_{\emptyset}\left[\frac{g(x)}{\emptyset(x)}\right]$$

Where h and $\emptyset$ are the pdfs of the $Uniform(a,b)$ and $Z(0,1)$ distributions respectfully.

As we are able to express both $I_1$and I_2 as expectations, we can therefore approximate their values using a Monte-Carlo estimator $\hat{I}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i$, with Y_i $i.i.d.$

In the case of I_1 we have $Uniform[a,b] \sim x_i =^d= a + (b-a)u_i\ s.t.\ u_i \sim Uniform[0,1]$. Therefore, we generate x_i using the inverse transform method, evaluate $(b-a)f(x_i)$ and then repeat n times, yielding the Monte-Carlo estimator:

$$\hat{I}_{1,n} = \frac{b-a}{n}\sum_{i=i}^{n} f(a+(b-a)u_i),\ \ u_i \sim IID\ Uniform(0,1).$$

For I_2 we apply a similar method, but note as it is an improper integral, we elected to sample x_i from the standard normal distribution which has equal range on $\mathbb{R}$. Therefore, generating $z_i \sim N(0,1)$ via some suitable $i.i.d.$ method, such as the Box-Muller method, we obtain

$$\hat{I}_{2,n} = \frac{1}{n}\sum_{i=1}^{n} \frac{g(z_i)}{\emptyset(z_i)},\ \ z_i \sim IID\ N(0,1)$$

2.3.

Numerical schemes and Monte-Carlo estimators both produce numerical estimates of definite integrals in fundamentally different ways:

Numerical schemes, such as the trapezoidal rule, approximate the integral by first partitioning it into many smaller sub-integrals in a summation, and then approximating the function f by a polynomial p_m, of degree m, which is simpler to integrate. Therefore, the error associated with such numerical scheme arises, in part, as a result of the accuracy of using such a polynomial approximation relative to their sub-integrals. We also note that the error associated with such schemes is also contingent upon the fineness of the grid of partitions. To illustrate, in the extreme case where we don't partition at all, we have

$$\int_a^b f(x)\,dx \approx \int_a^b p_m(x)dx$$

However, when $a \ll b$ this approximation will never be close to accurate, regardless of what m we take, unless f is itself also polynomial, and in which case a numerical scheme would not be required as the original integral can be solved directly.

Thus, we have two sources of error, which are not themselves independent, as the finer our grid of partitions, the more accurate the approximation of p_m is on our sub-intervals.

$$\lim_{n\to\infty} T(n) = \int_a^b f(x)dx$$

Where $T(n)$ is the trapezoidal approximation under n even partitions of $[a,b]$.

Therefore, when quantifying the error of our numerical scheme approximation, we would expect n to feature heavily, and in fact we find in the case of the trapezoidal scheme:

$$Error\ of\ each\ subinterval = E_{i-1} = \frac{1}{n^3}\frac{(b-a)^3}{12}f''(\varepsilon_{i-1}), \qquad \epsilon_{i-1} \in (x_{i-1}, x_i)$$

$$Total\ error = T(n) - \int_a^b f(x)dx = \frac{b-a}{12n^2}f''(\varepsilon), \qquad \varepsilon \in (a,b)$$

Monte-Carlo estimators however, produce numerical estimates by utilising the law of large numbers, $\lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n} x_i = \mu = E[x]$. Thus, one major source of error inherent in the Monte-Carlo scheme concerns how large n is & thus how fast $\bar{x}_n$ converges to $E[x]$. However, we note that the CLT also states that $\bar{x}_n \sim N\left(\mu, \frac{\sigma^2}{n}\right)$, yielding a 100(1-$\epsilon$)% confidence interval of $\left(\bar{x}_n - a_{\frac{\epsilon}{2}}\frac{\sigma}{\sqrt{n}}, \bar{x}_n + a_{\frac{\epsilon}{2}}\frac{\sigma}{\sqrt{n}}\right)$ for μ thus the speed of convergence also depends upon $\sigma\ (= Var(x))$. We therefore can quantify/evaluate the error of $\bar{x}_n$, by the width of the confidence interval:

$$CI\ width = 2a_{\frac{\epsilon}{2}}\frac{\sigma}{\sqrt{n}} \to 0\ for\ \sigma \to 0\ and/or\ n \to \infty$$

In practice, the use of either method is not dictated solely by their accuracy, but also by their speed of convergence relative to one another. Here we see numerical schemes perform very well, with the trapezoidal rule requiring little computational complexity ($O\left(n^{-\frac{2}{d}}\right)$) and converging very fast in low dimensional settings. In contrast, Monte-Carlo simulation is much slower to converge ($O\left(\sqrt{n}\right)$) in lower dimensions, due to its use/requirement of a very large n for the CLT to hold. However, we note that as d (the dimension) increases, Monte-Carlo simulation maintains it speed of convergence, whilst numerical schemes introduce far more error as computational complexity increases linearly with the dimension.

Thus, for small d, numerical schemes may be more useful, however for $d \geq 2$, Monte-Carlo simulation will likely start to outperform such methods.

2.4.

We note, when picking an I_2 type integral, it is preferable for g to be symmetric around a particular point, allowing us to separate I_2 into 2 equal integrals and for a simpler substitution. We demonstrate this using the example:

$$I_2 = \int_{-\infty}^{\infty} g(x)dx \quad g(x) = \frac{1}{1+\exp(x^2)}$$

We note that our choice of g is twice continuously differentiable:

$$g'(x) = -1(2x\exp(x^2))(1+\exp(x^2))^{-2} = -\frac{2x\exp(x^2)}{[1+\exp(x^2)]^2}$$

$$g''(x) = -2\exp(x^2)\,(1+\exp(x^2))^{-2} - 2x(2x\exp(x^2))(1+\exp(x^2))^{-2} - 2x\exp(x^2)\,(-2)(2x\exp(x^2)\,(1+\exp(x^2))^{-3} = -\frac{2\exp(x^2)}{[1+\exp(x^2)]^2}\left(1+2x^2-\frac{4x^2\exp(x^2)}{1+\exp(x^2)}\right)$$

As $\exp(x^2)+1>0\ \forall\ x\in\mathbb{R}$ and therefore there are no points of discontinuity in $g, g'or\ g''$.

Furthermore, $g(x)>0\ \forall\ x\in\mathbb{R}\ \Rightarrow\ I_2>0$

For the trapezoidal rule approximation, we obtain:

$$I_2 = \int_{-\infty}^{\infty} g(x)dx = \int_{-\infty}^{\infty}\frac{1}{1+\exp\,(x^2)}dx = \int_0^{\infty}\frac{dx}{1+\exp\,(x^2)} + \int_{-\infty}^{0}\frac{dx}{1+\exp\,(x^2)} = 2\int_0^{\infty}\frac{dx}{1+\exp\,(x^2)} = (*)$$

Using the substitution: $t=\exp(-x)$, we have: $t_1=\exp(-x_1)=\exp(0)=1$,

$$t_2 = \lim_{x_2\to\infty}\exp(-x_2) = 0$$

$$\frac{dt}{dx} = -\exp(-x) = -t\ \Rightarrow dx = -\frac{1}{t}dt$$

$$\Rightarrow (*) = 2\int_1^0 \frac{1}{1+\exp[ln^2(t)]} - \frac{dt}{t} = 2\int_0^1 \frac{1}{t(1+\exp[ln^2(t)])}dt$$

Let $w(t) := \frac{1}{t(1+\exp[ln^2(t)])}$, then we note that whilst $w(t)$ is undefined at $t=0$, the limit

$$\lim_{t\to 0} w(t) = \lim_{t\to 0}\frac{1}{t(1+\exp[ln^2(t)])} = 0$$

and thus, we use this limit in place of $w(0)$ when we later apply our numerical scheme.

$$w(1) = \frac{1}{1(1+\exp[ln^2(1)])} = \frac{1}{2}$$

Our numerical approximation using the trapezoidal rule therefore is

$$I_2 = \int_{-\infty}^{\infty}\frac{1}{1+\exp\,(x^2)}dx = 2\int_0^1\frac{1}{t(1+\exp[ln^2(t)])}dt \approx 2\frac{1-0}{n}\left[\frac{w(0)+w(1)}{2} + \sum_{i=1}^{n-1} w\left(0+\frac{i(1-0)}{n}\right)\right] = \frac{2}{n}\left[\frac{1}{4}+\sum_{i=1}^{n-1} w\left(\frac{i}{n}\right)\right]$$

Hence our trapezoidal rule estimator, $\hat{I^{TR}}_{2,n}$, for our given I_2 type integral is given by:

$$\hat{I^{TR}}_{2,n} = \frac{2}{n}\left[\frac{1}{4}+\sum_{i=1}^{n-1} w\left(\frac{i}{n}\right)\right] = \frac{2}{n}\left(\frac{1}{4}+\sum_{i=1}^{n-1}\frac{n}{i\left[1+\exp\left(ln^2\left(\frac{i}{n}\right)\right)\right]}\right)$$

We note that whilst we used $\lim_{t\to 0} w(t)=0$ in place of $w(0)$ above, we could have equally taken $w(\epsilon)$, where ϵ is arbitrarily close to 0. However, we note for any $\epsilon\approx 0$, we still obtain $w(\epsilon)\to 0$ due to the fast rate of convergence of $w(t)$, and thus the use of $w(\epsilon)$ over the limit makes no tangible difference, primarily because the area bound by $w(t)$ and the axis (and therefore by $g(x)$) is finite i.e. $w(t)$ does not tend to $\pm\infty$ at $t=0$.

In comparison, constructing a Monte-Carlo estimator is far simpler, as we can simply use the methodology outlined above in 2.2. to obtain:

$$\hat{I}^{MC}{}_{2,n} = \frac{1}{n}\sum_{i=1}^{n}\frac{g(z_i)}{\emptyset(z_i)} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\emptyset(z_i)}\frac{1}{1+\exp(z_i^2)}, \qquad z_i \sim IID\ N(0,1)$$

As stated above in 2.3. numerical schemes such as the trapezoidal rule tend to outperform Monte-Carlo estimation on lower dimensions, and thus we can expect the numerical scheme to be more suitable with a faster rate of convergence $O(n^{-2})$ vs $O\left(\sqrt{n}\right)$ for Monte-Carlo, and thus a faster rate and therefore lower error.

Problem 3.

$$S_t = S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t\right], \qquad P = \begin{cases} C, & if\ S_T > H \\ 0, & Otherwise \end{cases}$$

3.1.

We note that we can write the payoff of the *cash-or-nothing digital option* in the form of the indicator function $\mathbf{1}_{\{S_T>H\}}C$. As the price of our option is given by the discounted risk-neutral expectation of the payoff, we therefore have: Black-Scholes cash-or-nothing option price $V_0^{BS} = E^{\mathbb{Q}}[\mathbf{1}_{\{S_T>H\}}Ce^{-rt}]$, where S_T follows the distribution outlined above. Evaluating this risk-neutral expectation we find:

$$V_0^{BS} = E^{\mathbb{Q}}[Pe^{-rt}] = E^{\mathbb{Q}}\left[\mathbf{1}_{\{S_T>H\}}Ce^{-rt}\right] = Ce^{-rt}\mathbb{P}(S_T > H) = Ce^{-rt}\mathbb{P}[\log(S_T) > \log(H)] = (*)$$

$$log(H) < \log(S_T) = \log(S_0) + \left(r - \frac{\sigma^2}{2}\right)T + \sigma W_T \Rightarrow -\sigma W_T < \log\left(\frac{S_0}{H}\right) + \left(r - \frac{\sigma^2}{2}\right)T$$

$$W_T \equiv \sqrt{T}Z\ , Z \sim N(0,1)$$

$$\Rightarrow Z > -\frac{\log\left(\frac{S_0}{H}\right) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} =: -d_2$$

$$\Rightarrow (*) = Ce^{-rt}\mathbb{P}[Z > -d_2] = Ce^{-rt}\mathbb{P}[Z \leq d_2] = Ce^{-rt}\Phi(d_2)$$

Where the third equality uses the fact that $\log(*)$ is a monotonically (strictly) increasing function, and $\Phi(*)$ corresponds to the CDF of the standard normal distribution.

Thus the analytical Black-Scholes formula for the time-0 price of a *cash-or-nothing digital option* is given by $V_0^{BS} = Ce^{-rt}\Phi(d_2)$, when S_t follows a geometric brownian motion and Black-Scholes assumptions hold.

3.2.

a)

As we exist within the *Black-Scholes* world, theorem 54 in the lecture notes states that we can approximate the price of a derivative V_0, with payoff $h(S_T)$ by the following Monte-Carlo estimator:

$$V_0^{MC}(n) = \frac{1}{n}\sum_{i=1}^{n} e^{-rt}h(S_i), \qquad where\ S_1, \dots, S_n \sim i.i.d.\ lognorm\left(\log(S_0) + \left(r - \frac{\sigma^2}{2}\right)T, \sigma^2 T\right)$$

Thus for a *cash-or-nothing digital option* with payoff $h(S_T) := P = \mathbf{1}_{\{S_T > H\}}C$ we have the following estimator:

$$V_0^{MC}(n) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{S_i > H\}}Ce^{-rt}, \qquad where\ S_1, \dots, S_n \sim i.i.d.\ lognorm\left(\log(S_0) + \left(r - \frac{\sigma^2}{2}\right)T, \sigma^2 T\right)$$

b)

We note that in order to compute our Monte-Carlo estimator, V_0^{MC}, we first need to find a method to simulate the lognormal variables S_i. This is rather simple however, as using the distributional formula for S_i outlined above in 3.2.a, we have

$$S_i := S(Z_i) = S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}Z_i\right], \qquad Z_i \sim i.i.d.\ N(0,1)$$

Thus, to generate V_0^{MC} using a C++ program (or another suitable programming language), specify a function myUniform() which generates pseudo-random numbers from the $Uniform(0,1)$ distribution. Then using the Box-Muller method, generate a random variable $Z \sim N(0,1)$, allowing you to generate a sample path of $S_i = S(Z_i)$ and therefore evaluate the discounted payoff function $h(S_i) = \mathbf{1}_{\{S_i > H\}}Ce^{rt}$. Then, use a for-loop to repeat this process and generate n of these $h(S_i)$ random variables, before summing and finally averaging the result to obtain your Monte-Carlo price estimator. To summarise:

1) Generate $U_1, U_2 \sim i.i.d.\ Uniform(0,1)$
2) Set $R = -2\log(U_1)$ & $\theta = 2\pi U_2$
3) Set $Z_i = \sqrt{R}\cos(\theta) \Rightarrow Z_i \sim N(0,1)$
4) Set $S_i = S(Z_i) = S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}Z_i\right]$ to simulate a sample path of S_T

5) Evaluate the option payoff function $h_i := h(S_i) = \mathbf{1}_{\{S_i>H\}} C e^{rt}$
6) Return $V_0^{MC}(n) = \frac{1}{n}\sum_{i=1}^{n} h(S_i)$ as your Monte-Carlo price estimate

c)

$$Var\left(V_0^{MC}(n)\right) = Var\left(\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{S_i>H\}} C e^{-rt}\right) = \frac{1}{n^2} C^2 e^{-2rt} \sum_{i=1}^{n} Var\left(\mathbf{1}_{\{S_i>H\}}\right)$$
$$= \frac{1}{n} C^2 e^{-2rt} Var\left(\mathbf{1}_{\{S_T>H\}}\right) = (*)$$

$$Var\left(\mathbf{1}_{\{S_T>H\}}\right) = E^2\left[\mathbf{1}_{\{S_T>H\}}\right] - E\left[\mathbf{1}_{\{S_T>H\}}\right]^2 = \mathbb{P}(S_T > H) - \mathbb{P}(S_T > H)^2 = \Phi(d_2)[1 - \Phi(d_2)]$$
$$= \Phi(d_2)\Phi(-d_2)$$

$$Therefore\ Var\left(V_0^{MC}(n)\right) = (*) = \frac{1}{n} C^2 e^{-2rt}\, \Phi(d_2)\Phi(-d_2)$$

Where the third equality on the second line uses the main result from 3.1. We therefore see that the variance of the Monte-Carlo estimator decreases asymptotically in n.

d)

By the CLT we have: $for\ n \to \infty,$ $V_0^{MC}(n) \sim N(V_0, Var\left(V_0^{MC}(n)\right)$

Implying a $100(1-\epsilon)\%$ asymptotic confidence interval of:

$$V_0 \in \left(V_0^{MC}(n) - \alpha_{\frac{\epsilon}{2}} * \sqrt{Var\left(V_0^{MC}(n)\right)}, V_0^{MC}(n) + \alpha_{\frac{\epsilon}{2}} * \sqrt{Var\left(V_0^{MC}(n)\right)}\right), n \to \infty$$

Where $\alpha_{\frac{\epsilon}{2}}$ corresponds to the $100\left(1 - \frac{\epsilon}{2}\right)\%$ percentile of the standard normal distribution and $V_0^{MC}(n)$ and $Var\left(V_0^{MC}(n)\right)$ are defined in 3.2.a and 3.2.c respectfully.

We find $\alpha_{2.5} = 1.95$ and $\alpha_{0.05} = 2.576$, thus:

$$Our\ 95\%\ CI\ is = \left(V_0^{MC}(n) - 1.95 * \sqrt{Var\left(V_0^{MC}(n)\right)}, V_0^{MC}(n) + 1.95 * \sqrt{Var\left(V_0^{MC}(n)\right)}\right)$$

$$Our\ 99\%\ CI\ is = \left(V_0^{MC}(n) - 2.576 * \sqrt{Var\left(V_0^{MC}(n)\right)}, V_0^{MC}(n) + 2.576 * \sqrt{Var\left(V_0^{MC}(n)\right)}\right)$$

3.3.

a) Code located in appendix & zip file

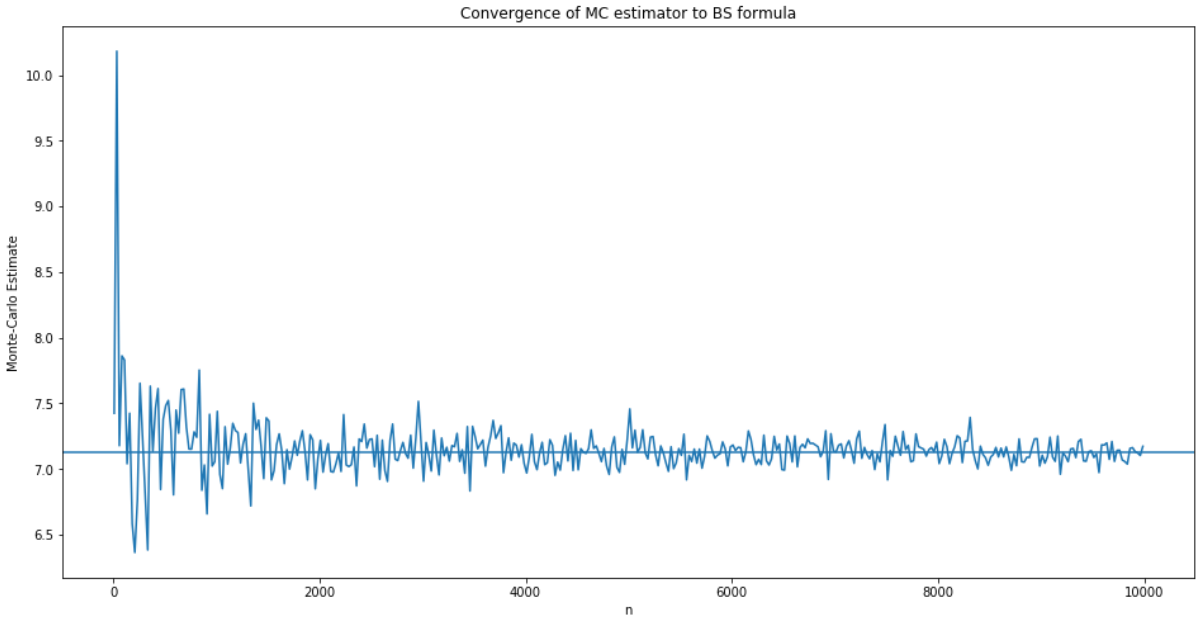
b) Code located in appendix & zip file.

Having computed the analytical variance, $Var\left(V_0^{MC}(n)\right) = (*) = \frac{1}{n} C^2 e^{-2rt}\, \Phi(d_2)\Phi(-d_2)$, in 3.2.c, therefore when producing our asymptotic confidence intervals for our *cash-or-nothing digital*

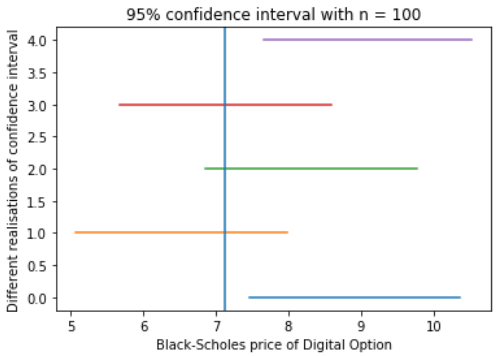
option, we do not need to approximate $Var\left(V_0^{MC}(n)\right)$ by the sample variance S^2, using our error free, analytical result instead.

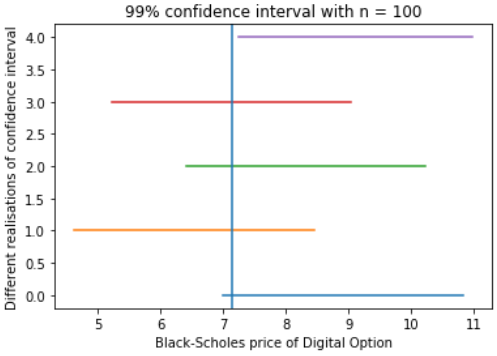
3.4.

We note that the accuracy of the Monte-Carlo estimator depends greatly upon the size on $n = number\ of\ iterations$, as a result of $Var\left(V_0^{MC}(n)\right)$ asymptotic in n.



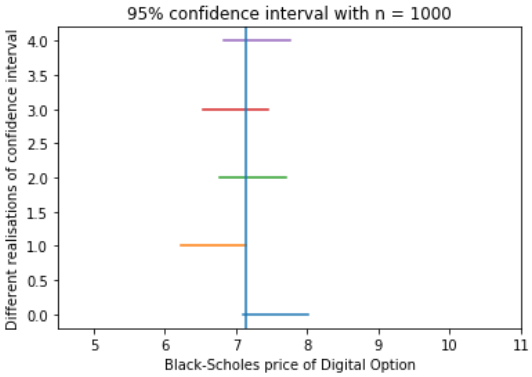
This asymptotic relationship can be clearly viewed through the reverse heteroscedasticity of the graph above and indicates a reasonable rate of convergence, even for relatively small n. This is a highly desirable trait, as the computational intensity is $O\left(\sqrt{n}\right)$ for a Monte-Carlo estimator, indicating that we may be able to have both accuracy and fast convergence.

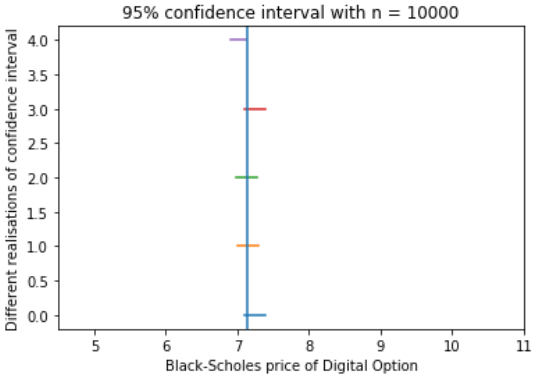


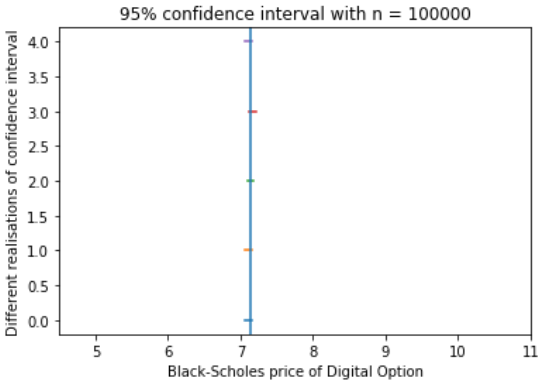


However, looking at our confidence intervals we note that we still have to be very careful with our choice of n, as for smaller values such as $n = 100$, we observe extremely large interval widths yet two scenarios where the 95% CI does not contain the true V_0^{BS}, and therefore extremely poor convergence. As expected however, our interval estimates do become far more accurate as n increases:

(Note we maintained the scale of the x axis for easy comparison)







However, whilst we do observe fast convergence of our estimator with respect to the (arbitrary) metric $\left|V_0^{MC}(n) - V_0^{BS}\right| < 0.5$ for $n \approx 1000$, we note our values only becomes accurate/correct to 1dp for $n \geq 100{,}00$. This is a highly significant factor, as even though the price difference between a single digital option's analytical Black-Scholes and Monte-Carlo price may be small in magnitude, financial derivates are often purchased at scale, in quantities of 00's or 000's, & thus such errors could result in large mispricing's in aggregate.

However, we note that it is not desirable to reduce this error, as measured by the confidence interval width: $2\alpha_{\frac{\epsilon}{2}} * \sqrt{Var\left(V_0^{MC}(n)\right)}$, by using a large n. This is because such error decreases only linearly with $\sqrt{n}$ & thus a 10x increase in n corresponds only to a much smaller decrease in the confidence interval width by $\sqrt{10}x \approx 3.16x$. Furthermore, when performing option pricing at scale, it is highly undesirable to use an arbitrarily large n, due to the extreme computational run-time → i.e. banks usually must price thousands of derivates daily in order to accurately measure their risk exposure, and thus performing 100,000 Monte-Carlo simulations on each option is both highly undesirable and unrealistic. A much more efficient method of price estimation would be to incorporate variance reduction techniques, such as Control Variates estimates. Such methods attempt to reduce $\sqrt{Var\left(V_0^{MC}(n)\right)}$ without significantly impacting the computational intensity of the calculations or increasing n, & thus leading to a lower overall error and narrower confidence intervals for V_o^{BS} than the corresponding Monte-Carlo estimator with the same n.

3.5.

Building upon 3.4., we reiterate that it is highly desirable to reduce $\sqrt{Var\left(V_0^{MC}(n)\right)}$ without increasing computational run-time via larger and larger n. The control variates method is one such technique of variance reduction which is outlined below:

$Let\ X\ \&\ Y\ be\ random\ variables\ s.t.\ Y = f(X)\ and\ E[X]\ known. Given\ i.i.d.\ (X_i, Y_i)\ we\ define$:

$$Y_i(b) = Y_i - b(X_i - E[X])$$

It is easy to see $E[Y_i(b)] = E[Y]$ & $Var(Y_i(b)) = Var(Y) - 2bcov(X,Y) + b^2Var(X)$ which is a function of b and can therefore be minimised. Selecting

$$b^* = \frac{cov(X,Y)}{Var(X)}$$

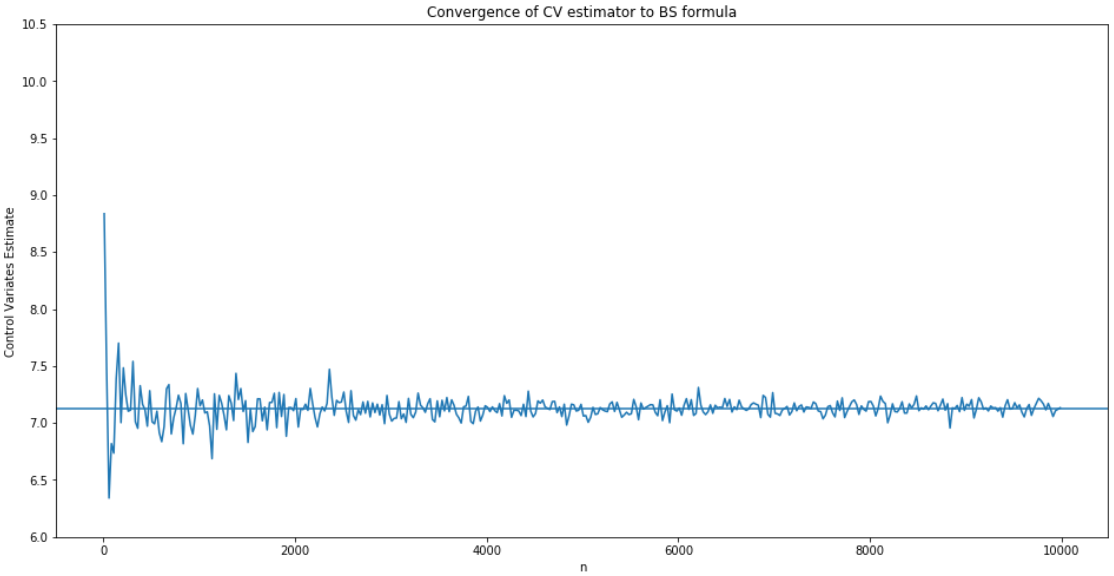
we obtain our optimal variance reduction, with $Var(\bar{Y}_n(b^*)) = \frac{1}{n}\left[Var(Y) - \frac{cov(X,Y)^2}{Var(X)}\right] < Var(\bar{Y}_n)$ where $\bar{Y}_n$ is the standard Monte-Carlo estimator $\bar{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i$, and $\bar{Y}_n(b)$ our control variates estimator:

$$\bar{Y}_n(b) := \frac{1}{n}\sum_{i=1}^{n} Y_i - b(X_i - E[X])$$

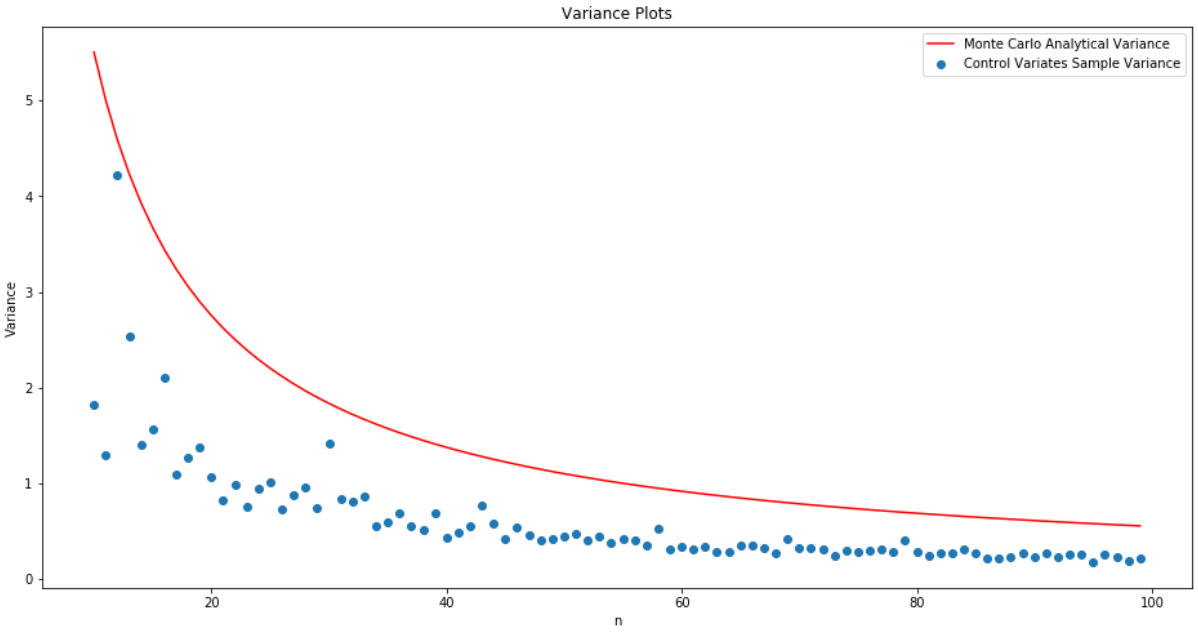
We note that as we have $E^{\mathbb{Q}}[S_T] = S_0 e^{rT}$ and $Pe^{rt} = \mathbf{1}_{\{S_T>H\}}Ce^{rt} = f(S_T)$, under Black-Scholes assumptions, we are therefore able to apply the above scheme, setting $Y_i := e^{rt}P_i = \mathbf{1}_{\{S_i>H\}}Ce^{rt}$ & $X_i := S_i$. Thus, our Control Variates estimator becomes:

$$\bar{V}^{CV}{}_n(b) = \frac{1}{n}\sum_{i=1}^{n}\left[\mathbf{1}_{\{S_i>H\}}Ce^{-rt} - \hat{b}^*(S_i - S_0e^{rt})\right]$$

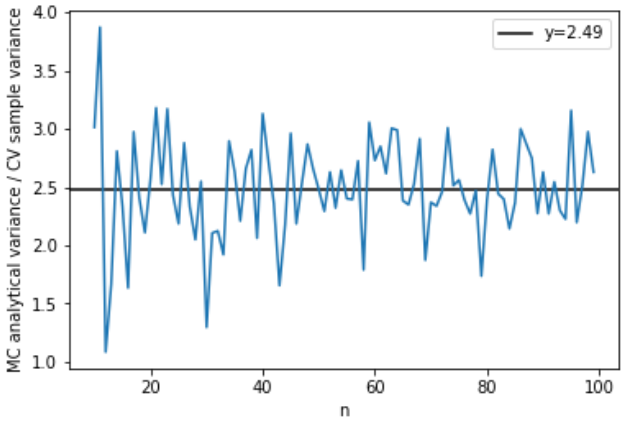
Where $\hat{b}^* = \frac{\sum_{j=1}^{n}(S_j - S_0e^{rt})(\mathbf{1}_{\{S_i>H\}}Ce^{-rt} - V_0^{MC}(n))}{\sum_{j=1}^{n}(S_j - S_0e^{rt})^2}$ (our estimator of b^*) and S_i are the simulated sample paths of S_T outlined in 3.2.



Maintaining the same axis and scale as graph 1 in 3.4. for comparison, we observe a much faster decline in the variation of $\bar{V}^{CV}{}_n(b)$ and faster rate of convergence relative to our Monte-Carlo estimator. Furthermore, even for very small n, i.e. $n \approx 100$, we have our control variates estimator satisfying $\left|\bar{V}^{CV}{}_n(b) - V_0^{BS}\right| < 0.5$, a metric our Monte-Carlo estimator fails to satisfy consistently until $n \approx 2000$.



Plotting the sample variance of the control variates estimator against n on the same graph as the analytical variance of the Monte-Carlo estimator, the size and scale of the variance reduction becomes immediately apparent, as does the relationship between n and $Var\left(\bar{V}^{CV}{}_n(b)\right)$ which must converge faster to 0 in n than the asymptotic relationship exhibited by $Var(V_n^{MC})$.



However, we note that (whilst containing a lot of noise) $\frac{Var(V_n^{MC})}{Var\left(\bar{V}^{CV}{}_n(b)\right)}$ does appear stationary around y=2.49, indicating that $Var\left(\bar{V}^{CV}{}_n(b)\right)$ is still asymptotic in n, but simply scaled by approximately $\frac{1}{2.5} = 0.4$ relative to the Monte-Carlo variance. Such a scaling would imply a variance reduction of 60%, a figure which is supported by running multiple simulations whilst varying n, demonstrating this stationarity does not just occur locally for small n.

For example, taking $n = 10{,}000$ we obtain a sample variance ≈ 0.00223, which is arbitrarily small enough for us to assume our control variance estimates are stable/reliable, whilst also corresponding to a variance reduction of 61.85% relative to the standard Monte-Carlo estimator, implying a $\frac{Var(V_{10,000}^{MC})}{Var\left(\bar{V}^{CV}{}_{10,000}(b)\right)}$ value of 0.3815, well within a reasonable range of $\frac{1}{2.5} + \epsilon$.

Thus it is reasonable to conclude, that whilst our control variates estimator does lead to a 60% variance reduction for our given $S_0, C, H, r, \sigma, \& T$, there remains an asymptotic relationship between $Var\left(\bar{V}^{CV}{}_n(b)\right) \to 0, n \to \infty$, and thus $E[\bar{V}^{CV}{}_n(b)] \to V_0^{BS}, n \to \infty$ and therefore the same overall speed of convergence to V_0^{BS} for n in the limit.

Problem 4.

$$dS_t = rS_t dt + \sigma S_t^{\gamma} dW_t$$

4.1.

We note that when we attempt to generate a sample path, $S = (S_t)_{t\geq 0}$, as S represents the price of a risky asset, it must satisfy the constraint $S_t \geq 0$. However, we find that without applying an appropriate transformation, many sample path generation schemes fail to satisfy this constraint. Thus, we denote: $Y_t = \log(S_t)$ as our transformation. Using the fact that we can now write $Y_t = f(S_t, t)$ and we have Ito's formula:

$$df(S_t,t) = f_t(S_t,t)dt + f_S(S_t,t)dS_t + \frac{1}{2}f_{SS}(S_t,t)d[S,S]_t$$

We obtain:

$$f_t = 0, \qquad f_S = \frac{1}{S_t}, \qquad f_{SS} = -\frac{1}{S_t^2}$$

$$[S,S]_t = \int (\sigma S^{\gamma})^2 dt \Rightarrow d[S,S]_t = \sigma^2 S^{2\gamma} dt$$

$$\rightarrow dY_t = dlog(S_t) = 0dt + \frac{1}{S_t}dS_t + \frac{1}{2}\left(-\frac{1}{S_t^2}\right)\sigma^2 S_t^{2\gamma} dt = \frac{rS_t dt + \sigma S_t^{\gamma} dW_t}{S_t} - \frac{1}{2}\sigma^2 S_t^{2\gamma-2} dt$$
$$= \left(r - \frac{\sigma^2}{2}S_t^{-2(1-\gamma)}\right)dt + \sigma S_t^{-1(1-\gamma)} dW_t$$

We note that as $\gamma \in \left[\frac{1}{2}, 1\right] \Rightarrow -(1-\gamma) \leq 0$ implying we have the additional constraint $S_t > 0$. However, this condition should be satisfied by any good sample scheme, as in practice even during extreme circumstances, risky assets always trade above the zero-lower bound $S_t > 0$.

Method 1 for generating $Y_t = \log(S_t)$: First order Euler Scheme

Denoting $\tilde{\mu}(S_t) := \left(r - \frac{\sigma^2}{2}S_t^{-2(1-\gamma)}\right)$ & $\tilde{\sigma}(S_t) := \sigma S_t^{-1(1-\gamma)}$, then the first order Euler scheme for the SDE $dX_t = \mu(X_t)dt + \sigma(X_t)dW_t$ is given by:

$$\hat{X}_{(j+1)h} = \hat{X}_{jh} + \mu(\hat{X}_{jh})h + \sigma(\hat{X}_{jh})\sqrt{h}Z_{j+1}, \qquad h\ is\ the\ grid\ defined\ in\ Q4, \qquad Z_j \sim i.i.d.N(0,1)$$

Thus, we can generate sample paths of Y using the formula:

$$\hat{Y}_{(j+1)h} = \hat{Y}_{jh} + \left(r - \frac{\sigma^2}{2}\hat{S}_{jh}^{-2(1-\gamma)}\right)h + \sigma\hat{S}_{jh}^{-(1-\gamma)}\sqrt{h}Z_{j+1}$$

As we now have a formula for generating $Y = \log(S)$ we therefore have $S = e^Y$

$$\Rightarrow \hat{S}_{(j+1)h} = e^{\hat{Y}_{(j+1)h}} = \hat{S}_{jh}\exp\left[\left(r - \frac{\sigma^2}{2}\hat{S}_{jh}^{-2(1-\gamma)}\right)h + \sigma\hat{S}_{jh}^{-(1-\gamma)}\sqrt{h}Z_{j+1}\right], \qquad \hat{S}_0 = S_0$$

Method 2 for generating $Y_t = \log(S_t)$: Second order Milstein Scheme

Using the same definitions $\tilde{\mu}(S_t) := \left(r - \frac{\sigma^2}{2}S_t^{-2(1-\gamma)}\right)$ & $\tilde{\sigma}(S_t) := \sigma S_t^{-1(1-\gamma)}$, as in method 1, the Milstein scheme for the SDE $dX_t = \mu(X_t)dt + \sigma(X_t)dW_t$ is given by:

$$\hat{X}_{(j+1)h} = \hat{X}_{jh} + \mu(\hat{X}_{jh})h + \sigma(\hat{X}_{jh})\sqrt{h}Z_{j+1} + \frac{1}{2}\sigma(\hat{X}_{jh})\sigma'(\hat{X}_{jh})h(Z_{j+1}^2 - 1), \qquad Z_j \sim i.i.d.N(0,1)$$

Thus, we can generate sample paths of Y using the formula:

$$\hat{Y}_{(j+1)h} = \hat{Y}_{jh} + \left(r - \frac{\sigma^2}{2}\hat{S}_{jh}^{-2(1-\gamma)}\right)h + \sigma\hat{S}_{jh}^{-(1-\gamma)}\sqrt{h}Z_{j+1} - \frac{1}{2}\sigma^2(1-\gamma)\hat{S}_{jh}^{-(3-2\gamma)}h(Z_{j+1}^2 - 1)$$

And as in method 1, we now have a formula for generating $Y = \log(S)$, thus:

$$\hat{S}_{(j+1)h} = \hat{S}_{jh}\exp\left[\left(r - \frac{\sigma^2}{2}\hat{S}_{jh}^{-2(1-\gamma)}\right)h + \sigma\hat{S}_{jh}^{-(1-\gamma)}\sqrt{h}Z_{j+1} - \frac{1}{2}\sigma^2(1-\gamma)\hat{S}_{jh}^{-(3-2\gamma)}h(Z_{j+1}^2 - 1)\right]$$
$$\hat{S}_0 = S_0$$

However, we note that the Milstein scheme requires $\tilde{\sigma}(S_t) \in C^2 \Rightarrow \gamma < 1 \Rightarrow \gamma \in \left[\frac{1}{2}, 1\right)$.

We note that both schemes are of the form $\hat{S}_{(j+1)h} = \hat{S}_{jh}\exp\left[g(\hat{S}_{jh}, Z_{j+1})\right]$, where g is some function. Thus as we have $S_0 > 0$ and $\exp(*)$ strictly increasing, $\Rightarrow \hat{S}_{(j+1)h} > 0\ \forall j, h$, thus satisfying our additional $S_t > 0$ constraint outlined earlier, allowing us to use either scheme to generate a sample path of S.

Therefore, to generate a sample path S, (using either scheme), given a discrete time grid of m evenly spaced partitions, we define $h := \frac{T}{m}$. Then for $i = 1, \dots, n$ & $j = 1, \dots, m$, denote $\hat{S}_{jh}^{(i)}$ as the value of $\hat{S}^{(i)}$ at time $jh = \frac{jT}{m}$ for the i^{th} sample path. First, we set $\hat{S}_0^{(i)} = S_0\ \forall i$ and then for each i & j, generate $Z_j^{(i)} \sim N(0,1)$ using the Box-Muller method outlined in 3.2.b. Then, set $\hat{S}_{jh}^{(i)} = \hat{S}_{(j-1)h}^{(i)}\exp\left[g(\hat{S}_{jh}, Z_{j+1})\right]$ where g is the function corresponding to the chosen scheme, outlined above.

4.2.

$$P = \begin{cases} C, & if\ S_T > H \\ 0, & Otherwise \end{cases}$$

As in 3.2.a, we have the time-0 price of the *cash-or-nothing digital option* is given by the risk neutral expectation: $V_0 = E^{\mathbb{Q}}[Pe^{-rt}] = E^{\mathbb{Q}}[\mathbf{1}_{\{S_T > H\}}Ce^{-rt}]$. We can compute the price of this option using the Monte-Carlo estimator $\hat{I} = \frac{1}{n}\sum_{i=1}^{n} Y_i$, where the Y_i are $i.i.d$.

Denoting $Y_i = \mathbf{1}_{\left\{S_T^{(i)} > H\right\}}Ce^{-rt}$, where $S_T^{(i)}$ corresponds to the terminal, time-T value of the i^{th} sample path ($j = m$) and are generated as described in 4.1. , our CEV-Monte-Carlo estimator is therefore given by:

$$V_0^{CEV\ MC}(n) = \frac{1}{n}\sum_{i=i}^{n} \mathbf{1}_{\left\{S_T^{(i)} > H\right\}}Ce^{-rt}$$

4.3.

Code located in appendix & zip file

We note that we computed the Monte-Carlo time-0 price of the option for both the Euler & Milstein schemes. Using the Milstein scheme we obtain $V_{10,000}^{CEV\ MC} = 7.66002, Var\left(V_{10,000}^{CEV\ MC}\right) = 0.00550866$, 95% & 99% confidence intervals $(7.51455, 7.80548)$ & $(7.46884, 7.85119)$ respectfully.

4.4.

We note that whilst we could continue to use either the Euler or Milstein scheme (or even both) to produce control variate estimators, as the Milstein scheme is itself a refinement of the first-order Euler scheme, with an improved strong order of convergence of 1 (vs $\frac{1}{2}$ for the Euler scheme), it possesses the theoretical property of faster convergence to (and more accurately modelling of) the true CEV stock price model. Thus, we will be using the Milstein approximation scheme for our control variates methods.

As outlined in 3.5, the control variates estimator is of the form:

$$\bar{Y}_n(b) := \frac{1}{n}\sum_{i=1}^{n} Y_i - b(X_i - E[X])$$

$$s.t.\ Y_i(b) := Y_i - b(X_i - E[X])$$

Where X & Y are random variables s.t. $Y = f(X)$, $E[X]$ known & $(X_i, Y_i)\ i.i.d.$

As $E^{\mathbb{Q}}[S_T] = S_0 e^{rT}$ is a model free result, it therefore also holds for the CEV model for S_t, and given $Pe^{rt} = \mathbf{1}_{\left\{S_T^{(i)} > H\right\}} Ce^{-rt} = f\left(S_T^{(i)}\right)$ we therefore have the CEV control variates estimator:

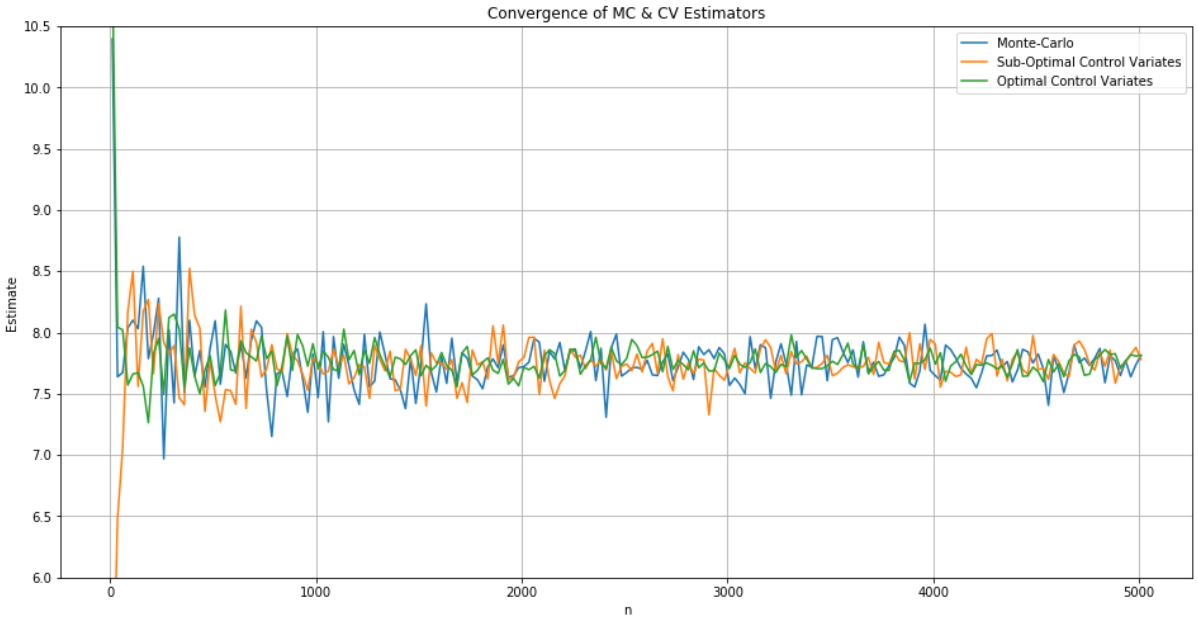
$$\bar{V}^{CEV\ CV}{}_n(b) = \frac{1}{n}\sum_{i=1}^{n}\left[\mathbf{1}_{\left\{S_T^{(i)} > H\right\}} Ce^{-rt} - \hat{b}^*\left(S_T^{(i)} - S_0 e^{rt}\right)\right]$$

Where $\hat{b}^* = \dfrac{\sum_{j=1}^{n}\left(S_T^{(j)} - S_0 e^{rt}\right)\left(\mathbf{1}_{\left\{S_T^{(j)} > H\right\}} Ce^{-rt} - V_0^{CEV\ MC}(n)\right)}{\sum_{j=1}^{n}\left(S_j - S_0 e^{rt}\right)^2}$ is our estimator of b^* and $S_T^{(i)}$ are the CEV sample paths of S_T, simulated using the Milstein approximation scheme outlined in 4.1.

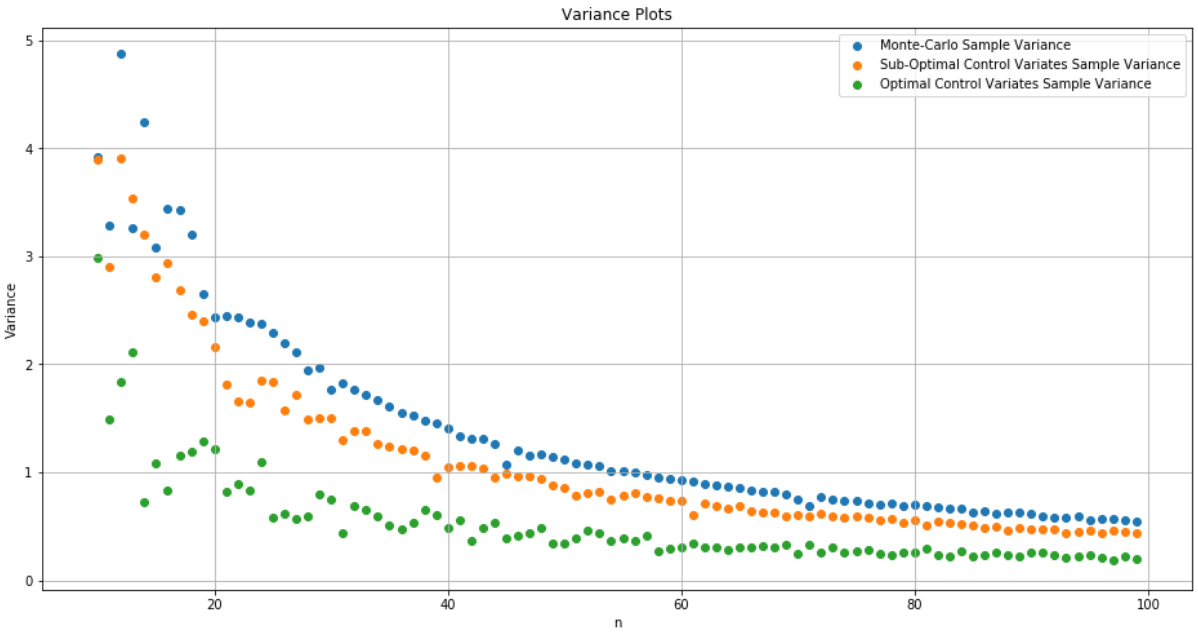
Choosing the following two control variates estimators:

1) Our 'optimal' control variates estimator $\bar{V}^{CEV\ CV}{}_n\left(\hat{b}^*\right),\ s.t.\ \hat{b}^*$ is our sample estimator for b^* outlined above.
2) Our 'sub-optimal' control variates estimator $\bar{V}^{CEV\ CV}{}_n(b)\ , s.t.\ b = 1$ (Note we were careful to check $b^* \neq 1$).

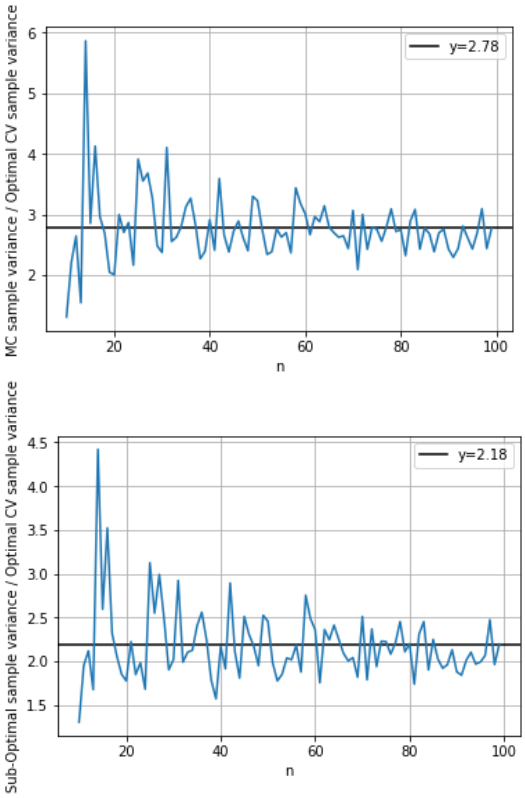
Implementing both control variates estimators and the Monte-Carlo estimator from 4.3., we immediately notice a large increase in the run-time using the Milstein scheme, relative to the geometric Brownian motion approach taken in 3.4. & 4.5. This computational run-time is due to the generation of $n * m$ sample paths, $\hat{S}_{jh}$, required by the use of the Milstein scheme, and thus depends upon both the fineness of the grid of partitions as well as the number of Monte-Carlo iterations (we will build upon this in 4.5.).



Performing similar analysis as in 3.5., we observe (as expected) both control variates schemes converging towards the true (unknown) option price V_0^{CEV} at a faster rate than our Monte-Carlo estimator. This can be seen either directly on the graph above, through the more volatile Monte-Carlo plot $\forall n$, or by separately plotting the sample variances of all three estimators as seen below.



This also allows us to confirm the variance of control variates scheme is clearly much lower $\forall n$ when our estimator utilises b^*, and in fact our C++ program shows this 'optimal' control variates estimator leads to a variance reduction of 53.29% relative to our 'sub-optimal' choice of $b = 1$ and a 63.43% reduction relative to our Monte-Carlo estimator, for $n = 10{,}000$. As in 3.5. we can also deduce, through the stationarity exhibited on graphs below, this variance reduction is not simply localised for large n, but seems to occur throughout at a constant rate. We observe $\frac{Var\left(V_n^{CEV\ MC}\right)}{Var\left(\bar{V}^{CEV\ Optimal\ CV}{}_n(b)\right)}$ and $\frac{Var\left(V_n^{CEV\ Sub-optimal\ CV}\right)}{Var\left(\bar{V}^{CEV\ Optimal\ CV}{}_n(b)\right)}$ stationary around $y = 2.76$ and $y = 2.18$ respectfully, implying all three estimators converge to V_0^{CEV} in the same order of n (which appears somewhat asymptotic) and thus



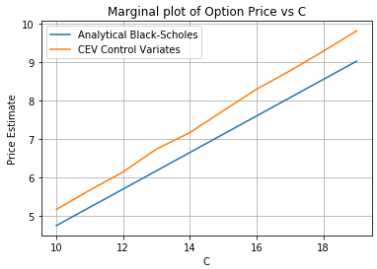
their variances are multiples of each over, i.e. $Var\left(\bar{V}^{CEV\ Optimal\ CV}{}_n(b)\right) \approx \frac{1}{2.78} Var(V_n^{CEV\ MC})$ & $Var\left(\bar{V}^{CEV\ Optimal\ CV}{}_n(b)\right) \approx \frac{1}{2.18} Var\left(V_n^{CEV\ Sub-optimal\ CV}\right)$. Such relationships would imply the 'optimal' CEV control variates estimator would result in average variance reductions of 64.03% and 54.13% relative to the CEV Monte-Carlo estimator and CEV 'sub-optimal' control variates estimator respectfully, figures coinciding within 1% of our $n = 10{,}000$ C++ simulation.

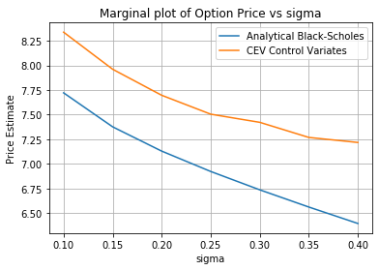
Thus, we note that whilst a control variates technique can reduce the variance of a Monte-Carlo estimator even for sub-optimal choices of b, the most significant improvement is observed when using b^*.

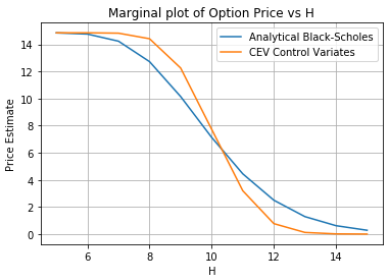
4.5.

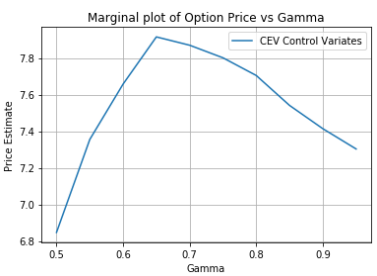
When comparing the effect of different model parameters upon the price of the option, it is important to choose an estimator corresponding to the lowest possible variance, as to ensure the stability of our results. For this purpose, we will therefore be comparing the analytical Black-Scholes price $V_0^{BS} = Ce^{-rt}\Phi(d_2)$, which is deterministic and therefore zero variance, with the $\bar{V}^{CEV\ CV}{}_{10{,}000}(\hat{b}^*)$, which we showed in 4.4. is the CEV estimator with the lowest variance.

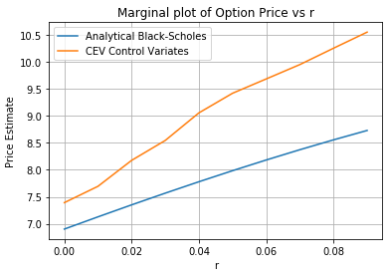
$(*)$ **Note the following individual data points have been computed by varying parameters using my C++ program. I ONLY used python for creating the graphs.**

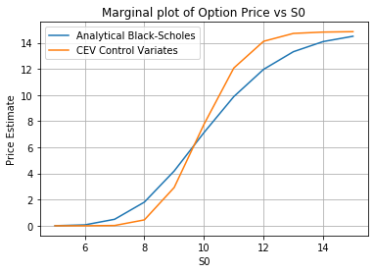


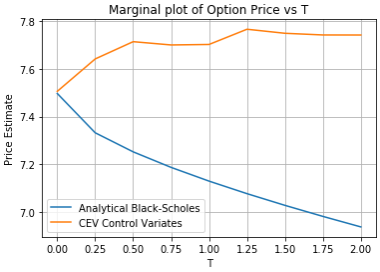












We are therefore able to see, with the exception of T, the relationships between the price estimates of the option and variables are of the same direction for both estimators, i.e. both $\bar{V}^{CEV\ CV}{}_{10,000}(\hat{b}^*)$ and V_0^{BS} are increasing in C, r and S_0, whilst both are decreasing σ & H.

Variable	$\frac{\partial V_0}{\partial variable}$	Explanation
C	> 0	All else equal (and therefore no variable risk), any asset which offers a higher payoff in a given state, must command a higher price, otherwise there exists an arbitrage opportunity. Thus, V_0 must be increasing in C for both models.
r	> 0	We have $E^{\mathbb{Q}}[S_T] = S_0 e^{rt}$, therefore as r increases, so too does our expected value of S_T, which increases the probability that $S_T > H$ at the maturity date of our option, thus increasing its value. As $E^{\mathbb{Q}}[S_T] = S_0 e^{rt}$ is a model free result, we therefore must obtain this relationship for both Black-Scholes and CEV models.
S_0	> 0	Similar to r, a larger S_0 implies that the option is more likely to be in the money ($S_T > H$), increasing the expected value of the option, V_0.
σ	< 0	We note that as we have $S_0 = H$ and we are holding these two values equal, our option is at the money at time-0. Therefore, larger σ implies there is a higher risk that at maturity T, the option will no longer be in the money and thus has a lower expected value at time-0 ⇒ Lower price V_0 for both models.
H	< 0	The argument for V decreasing in H is equivalent to the case of $-S_0 \rightarrow$ Larger H implies a lower probability that the option will be in the money at time-T. Thus the option price must be decreasing in H.

We therefore note that these relationships correspond to what we would expect from a financial asset derivative and therefore both models are suitable in regards to modelling their effects.

Whilst we can compare the sensitivity of the option to the underlying variables simply by looking at the plots above, it may be more initiative to compute linear estimates of our data using OLS and first order polynomial approximations (although γ is clearly at least quadratic, and thus we used a polynomial of degree 2 here), and then compare the relevant partial derivatives.

Variable	S_0	H	C	σ	r
Partial Derivative of V_0^{BS}	$\frac{\partial V}{\partial S_0} = 1.75$	$\frac{\partial V}{\partial H} = -1.77$	$\frac{\partial V}{\partial C} = 0.47$	$\frac{\partial V}{\partial \sigma} = -4.29$	$\frac{\partial V}{\partial r} = 20.36$
Partial Derivative of $\bar{V}^{CEV\ CV}{}_{10,000}(\hat{b}^*)$	$\frac{\partial V}{\partial S_0} = 1.95$	$\frac{\partial V}{\partial H} = -1.95$	$\frac{\partial V}{\partial C} = 0.47$	$\frac{\partial V}{\partial \sigma} = -3.59$	$\frac{\partial V}{\partial r} = 35.76$

Variable	T	γ
Partial Derivative of V_0^{BS}	$\frac{\partial V}{\partial T} = -0.27$	$\frac{\partial V}{\partial \gamma} = 0$
Partial Derivative of $\bar{V}^{CEV\ CV}{}_{10,000}(\hat{b}^*)$	$\frac{\partial V}{\partial T} = 0.06$	$\frac{\partial V}{\partial \gamma} = 23.03 - 31.24\gamma$

Looking at these partial derivatives, we see proof of the relationship between S_0 and H described above, with a unit increase in S_0 being equivalent to a unit decrease in H for both the Black-Scholes formula and our CEV control variates estimator. We also note that as our Black-Scholes formula is slightly less sensitive to changes in the initial underlying price, S_0 and by extension therefore changes in H. This difference must be attributed to the use of different asset price models, with the CEV model being able to incorporate the leverage effect in its path simulations (negative price movements corresponding to greater volatility than positive ones) and thus is more sensitive to S_0.

We also see the CEV model is more sensitive to r than the Black-Scholes, however both remain linear in r all else equal. Conversely, the CEV model is less sensitive towards chances in σ, but appears to form a slight curve in this variable, whilst the Black-Scholes is linear, suggesting the CEV model contains a more complex relationship with σ than the former (and one only has to look at the Milstein formula to see this is in fact the case).

We also note, not only does V increases linearly in C, but both the Black-Scholes formula and CEV control variates estimator move in tandem to each over, i.e. with the same gradient. This is not too surprising however, as in their respective mathematical equations, the variable C occurs once and in the same location, and thus they likely have very similar partial derivatives with respect to C.

As previously mentioned above, the incorporation of γ into the CEV model allows the model to capture the leverage effect within stochastic volatility. We see fluctuating our value of γ, all else equal, yields a parabola in its marginal plot, which appears to be maximised between $0.6 < \gamma < 0.7$. As the Black-Scholes model does not include the γ parameter, it therefore has no such marginal plot and a partial derivative of 0 for the variable.

Interestingly, whilst the Black-Scholes and CEV models do agree upon the effect of nearly all variables upon the value of the digital option, the exception is the time parameter T. Here we see that whilst increasing T leads to a somewhat linear decrease in V_0^{BS}, it conversely results in an increasing relationship with $V_n^{CEV\ CV}(b^*)$. This is a key point of interest, as for a long position on a digital option, we would expect to observe the price of the option increasing in T, as a longer dated maturity increases the probability the option will expire in the money. Thus, this provides evidence of a clear flaw in the Black-Scholes model, which the CEV corrects.

Conclusion

Whilst, as stated above, the CEV model does possess two important characteristics the Black-Scholes lacks → it captures the leverage effect upon S and the negative effect of time decay upon option value ($\frac{\partial V}{\partial T} > 0$), we have yet to take into account the difference in computational run-time and the option values.

Addressing the former, it is clear the simpler, deterministic Black-Scholes formula has the advantage with an $O(1)$ run time, whilst the CEV control variates estimator takes far longer to compute, due to the need to generate $n * m$ sample paths. As for the latter, both models result in different prices for V_0, with non-overlapping 99% confidence intervals of the Monte-Carlo estimators for both schemes and $\bar{V}^{CEV\ CV}{}_n(\hat{b}^*)$ appearing systematically larger than V_0^{BS}. This implies the difference between the two estimates is not simply down to chance, yet as both methods use different asset path schemes, it is reasonable to expect different values for their risk-neutral expectations, and thus in order to truly verify which produces the more accurate V_0 price estimate (the most important property of an estimator) we would have to compare these values to market prices.

Appendix

```
// Question3.h

#ifndef QUESTION3_H
#define QUESTION3_H

#include<iostream>
#include<cmath>
#include<cstdlib>
#include<ctime>

// QUESTION 3 CODE //

// FUNCTIONS //

double analytical(double C, double H, double S0, double r, double T, double sigma);
// Q3.3a: Computes the t=0 price of the cash-or-nothing digital option using the analytical formula.



// CLASSES //

class Question3Base{

  public:

    virtual double getEstimate(void) = 0;
    // A virtual function common to both derived classes.

    virtual double S_T(double S0, double r, double T, double sigma);
```

```
    // Computes/Simulates a stock path modelled upon a Geometric Brownian Motion.

};

class MonteCarlo:public Question3Base{

  public:

    MonteCarlo(double C, double H, double S0, double r, double T, double sigma, int n);
    // Our Constructor

    virtual double getEstimate(void);
    // Q3.3b: computes the Monte-Carlo estimate of the time-0 price of the cash-or-nothing digital
option

    double ConfidenceInterval(double epsilon);
    // Q3.3 Calculates our (1-epsilon) asymptotic confidence interval for our Monte-Carlo estimate.
    // Note, this returns the width of the (1-epsilon) interval. To access the values
    // myanalyticalVariance, myleftCI and myrightCI, which are the digital option's analytical
    // variance and (1-epsilon) confidence interval: [myleftCI,myrightCI], run this function
    // and then call these public member variables via:  MC.yourDesiredValue.

    double  myanalyticalVariance, myleftCI, myrightCI;
    // The public member variables described above.


  private:

    double myC, myH, myS0, myr, myT, mysigma, myEstimate;
    int myn;
    // Our private member variables used in calculations.
```

```
};

class ControlVariates:public Question3Base{

  public:

    ControlVariates(double C, double H, double S0, double r, double T, double sigma, int n);
    // Our Constructor

    double bstarhat(void);
    // Our function for approximating b*

    virtual double getEstimate(void);
    // Q3.5: computes the Control Variates estimate of the time-0 price of the cash-or-nothing digital option

    double myvariance;
    // Public member variable which stores the value of the Control Variates estimate
    // of the sample variance of the CV estimator.

    private:

      double myC, myH, myS0, myr, myT, mysigma, myEstimate;
      int myn;
      // Our private member variables used in calculations.
};

#endif // QUESTION3_H
```

```
// Question4.h

#ifndef QUESTION4_H
#define QUESTION4_H

#include<iostream>
#include<cmath>
#include<cstdlib>
#include<ctime>

// QUESTION 4 CODE //

// CLASSES //

class Question4Base{

  public:

     virtual double S_T_Euler(double S0, double r, double T, double sigma, double gamma, int m);
     // Computes/Simulates a stock path using the Constant Elasticity of Variance (CEV) model
     // for the price of a risky asset and a first order Euler approximation scheme.

     virtual double S_T_Milstein(double S0, double r, double T, double sigma, double gamma, int m);
     // Computes/Simulates a stock path using the Constant Elasticity of Variance (CEV) model
     // for the price of a risky asset and the second order Milstein approximation scheme.

  private:

      virtual double EulerIncrement(double S_tj, double h, double r, double sigma, double gamma);
      // Calculates the S_t(j+1), i.e. the next step/increment of S_tj, using the Euler Scheme.
```

```
    virtual double MilsteinIncrement(double S_tj, double h, double r, double sigma, double
gamma);
    // Calculates the S_t(j+1), i.e. the next step/increment of S_tj, using the Milstein Scheme.

};

class CEVMonteCarlo:public Question4Base{

  public:

    CEVMonteCarlo(double C, double H, double S0, double r, double T, double sigma, int n, double
gamma, int m);
    // Our Constructor

    virtual double getEstimate(void);
    // Q4.3: computes both the Monte-Carlo estimates of the time-0 price of the cash-or-nothing
digital option,
    // using:
    // 1) the CEV model for the risky asset price and the first order Euler approximation.
    // 2) the CEV model for the risky asset price and the second order Milstein approximation.
    // This function also calculate the asymptotic variance of both monte-carlo estimators.
    // Returns the Milstein scheme result.

    double ConfidenceInterval(double epsilon);
    // Q4.3 Calculates our (1-epsilon) asymptotic confidence interval for our both Monte-Carlo
estimates
    //-> one using the Euler CEV approximation and the other the Milstein CEV approximation.
    // Note, this returns the width of the (1-epsilon) interval for the Euler scheme. To access the
values
    // myEulerVariance, myMilsteinVariance, EulerLeftCI, EulerRightCI, MilsteinLeftCI,
MilsteinRightCI and
    // MilsteinCIWidth, which are the digital option's analytical variance and (1-epsilon) confidence
interval:
```

```
    // [LeftCI,RightCI] and the CI width for the Milstein scheme, run this function and then call these

    // public member variables via:  MC.yourDesiredValue.


    double myEulerVariance, myMilsteinVariance, EulerLeftCI, EulerRightCI;

    double MilsteinLeftCI, MilsteinRightCI, myEulerEstimate, myMilsteinEstimate, EulerCIWidth,
MilsteinCIWidth;

    // The public member variables described above.



  private:


    double myC, myH, myS0, myr, myT, mysigma, mygamma;

    int myn, mym;

    // Our private member variables used in calculations.


};


class CEVControlVariates:public Question4Base{


  // All functions utilise the Milstein CEV scheme


  public:


    CEVControlVariates(double C, double H, double S0, double r, double T, double sigma, int n,
double gamma, int m);

    // Our Constructor


    double bstarhat(void);

    // Our function for approximating b*


    virtual double getEstimate(double b);
```

```
    // Computes the CEV Control Variates estimate of the time-0 price of the cash-or-nothing digital
option


    double myvariance;
    // Public member variable which stores the value of the Control Variates estimate
    // of the sample variance of the CV estimator.


    private:


      double myC, myH, myS0, myr, myT, mysigma, mygamma;
      int myn, mym;
      // Our private member variables used in calculations.
};



#endif // QUESTION4_H
```

```
//RandomNumberGenerators.h

#ifndef RANDOMNUMBERGENERATORS_H
#define RANDOMNUMBERGENERATORS_H

double myuniform(void);

double uniformab(double a,double b);

double exponential_rv(double lambda);

double normal_rv(double mu,double sigma);

#endif // RANDOMNUMBERGENERATORS_H
```

```
//
//
//                  Normals.h
//
//
#ifndef NORMALS_H
#define NORMALS_H

double NormalDensity(double x);

double CumulativeNormal(double x);

double InverseCumulativeNormal(double x);

#endif

/*
 *
 * Copyright (c) 2002
 * Mark Joshi
 *
 * Permission to use, copy, modify, distribute and sell this
 * software for any purpose is hereby
 * granted without fee, provided that the above copyright notice
 * appear in all copies and that both that copyright notice and
 * this permission notice appear in supporting documentation.
 * Mark Joshi makes no representations about the
 * suitability of this software for any purpose. It is provided
 * "as is" without express or implied warranty.
*/
```

```
// Question3.cpp

#include "Question3.h"
#include "Normals.h"
#include "RandomNumberGenerators.h"

// QUESTION 3 CODE //

double analytical(double C, double H, double S0, double r, double T, double sigma){

   double d = ( log((S0/H)) + (r-(0.5*sigma*sigma))*T )/(sigma*sqrt(T));
   double result = C*exp(-1*r*T)*CumulativeNormal(d);
   return result; // The analytical price of our digital option.
}

double Question3Base::S_T(double S0,double r,double T,double sigma){

        double Z, diffusion, drift = (r-(0.5*sigma*sigma))*T;
        Z = normal_rv(0.0,1.0);
        diffusion = sigma*sqrt(T)*Z;
        return (S0*exp(drift + diffusion)); // Our simulated GBM stock price
}

MonteCarlo::MonteCarlo(double C, double H, double S0, double r, double T, double sigma, int n){
   // Initialising private member variables.
      myC    = C;
      myH    = H;
      myS0   = S0;
      myr    = r;
      myT    = T;
      mysigma = sigma;
```

```
    myn     = n;

}

double MonteCarlo::getEstimate(void){

  double ST, sum = 0;
  for(int i = 0; i < myn; i++){
    // Performing myn = n Monte-Carlo iterations, to produce a numerical estimate of the
    // digital option price.

    // Generating Our stock path.
    ST = S_T(myS0,myr,myT,mysigma);

    // Our Indicator Function.
    if( ST > myH ){
      sum += (myC*exp(-1*myr*myT));
    }
  }

  myEstimate = sum/myn; // Our Monte-Carlo Estimate.


  // Calculating Option analytical Variance.
  double d = ( log(myS0/myH) + (myr-(0.5*mysigma*mysigma))*myT )/(mysigma*sqrt(myT));
  myanalyticalVariance = myC*myC*exp(-2*myr*myT)*CumulativeNormal(d)*CumulativeNormal(-
1*d)/myn; // assigning value to respective private member variable.

  return myEstimate;
}
```

```
double MonteCarlo::ConfidenceInterval(double epsilon){
   // Calculating Confidence Interval.
   // Function returns the interval width.

   double alpha = InverseCumulativeNormal(1.0 - (epsilon/2.0));
   myleftCI = myEstimate - alpha*sqrt(myanalyticalVariance);  // assigning values to respective
   myrightCI = myEstimate + alpha*sqrt(myanalyticalVariance); // private member variables.
   return myrightCI - myleftCI; // The CI interval width.
}

ControlVariates::ControlVariates(double C, double H, double S0, double r, double T, double sigma,
int n){
   // Initialising private member variables.
      myC = C;
      myH = H;
      myS0 = S0;
      myr = r;
      myT = T;
      mysigma = sigma;
      myn = n;
}

double ControlVariates::bstarhat(void){

   double numeratorSum = 0, denominatorSum = 0;
   double Xi,Yi;

   // Calculating EX and EY =(approx)= Monte-Carlo(Y).
   double EX = exp(myr*myT)*myS0;
   MonteCarlo MC(myC,myH,myS0,myr,myT,mysigma,myn);
   double EY = MC.getEstimate();
```

```
    for(int i = 0; i < myn; i++){
        Xi = S_T(myS0,myr,myT,mysigma);
        Yi = 0;
        if( Xi > myH ){
            Yi = myC*exp(-1*myr*myT);
        }
        numeratorSum += ((Xi - EX)*(Yi - EY));
        denominatorSum += ((Xi - EX)*(Xi - EX));
    }
    return numeratorSum/denominatorSum; // Our B*hat estimate.

}

double ControlVariates::getEstimate(void){

    double yib, ST, CVEstimate, sum = 0, squaresum = 0;
    double yi,Xi, EX = exp(myr*myT)*myS0; // E[X] = E[ST].
    double b = bstarhat();              // Our b* estimate.
    for(int i = 0; i < myn; i++){

        // Generating Our stock path.
        ST = S_T(myS0,myr,myT,mysigma);

        // Calculating Yi.
        yi = 0;
        if( ST > myH ){
            yi = myC*exp(-1*myr*myT);
        }

        // Calculating Yi(b).
```

```
        Xi = ST;
        yib = yi - b*(Xi - EX);

        sum += yib;
        squaresum += (yib*yib);

    }

    // Calculating statistics.
    CVEstimate = sum/myn;
    myvariance = (squaresum - (myn*CVEstimate*CVEstimate))/(myn*(myn-1)); // Our sample variance estimate of our Control Variates estimator.

    return CVEstimate; // Our Control Variates Estimate.
}
```

```
// Question4.cpp

#include "Question4.h"
#include "Normals.h"
#include "RandomNumberGenerators.h"

// QUESTION 4 CODE //

double Question4Base::S_T_Euler(double S0, double r, double T, double sigma, double gamma, int m){

   // Assumes/uses an evenly divided time grid of m partitions, S0 > 0 and 0.5 <= gamma <= 1.
   double h = T/m;
   double S_tj = S0;
   for(int i = 0; i < m ; i++){ S_tj = EulerIncrement(S_tj, h, r, sigma, gamma);}

   return S_tj;
}

double Question4Base::S_T_Milstein(double S0,  double r, double T, double sigma, double gamma, int m){

   // Assumes/uses an evenly divided time grid of m partitions, S0 > 0 and 0.5 <= gamma < 1.
   double h = T/m;
   double S_tj = S0;
   for(int i = 0; i < m ; i++){ S_tj = MilsteinIncrement(S_tj, h, r, sigma, gamma);}

   return S_tj;
}

double Question4Base::EulerIncrement(double S_tj, double h, double r, double sigma, double gamma){
```

```
    double phi = gamma - 1;
    double drift = ( r - 0.5*sigma*sigma*pow(S_tj,2*phi) )*h;
    double Z = normal_rv(0.0,1.0);
    double diffusion = sigma*pow(S_tj,phi)*sqrt(h)*Z;

    return S_tj*exp(drift + diffusion);
}

double Question4Base::MilsteinIncrement(double S_tj, double h, double r, double sigma, double
gamma){
    double phi = gamma - 1;
    double omega = 2*gamma - 3;
    double drift = ( r - 0.5*sigma*sigma*pow(S_tj,2*phi) )*h;
    double Z = normal_rv(0.0,1.0);
    double diffusion = sigma*pow(S_tj,phi)*sqrt(h)*Z;
    double MilsteinTerm = -0.5*sigma*sigma*phi*pow(S_tj,omega)*h*(Z*Z - 1);

    return S_tj*exp(drift + diffusion + MilsteinTerm);
}

CEVMonteCarlo::CEVMonteCarlo(double C, double H, double S0, double r, double T, double sigma,
int n, double gamma, int m){
    // Initialising private member variables.
        myC     = C;
        myH     = H;
        myS0    = S0;
        myr     = r;
        myT     = T;
        mysigma = sigma;
        myn     = n;
        mygamma = gamma;
        mym     = m;
```

```
}

double CEVMonteCarlo::getEstimate(void){

   double temp, ST_Euler, ST_Milstein, sumEuler = 0, sumEulerSquared = 0, sumMilstein = 0,
sumMilsteinSquared = 0;
   for(int i = 0; i < myn; i++){
      // Performing myn = n Monte-Carlo iterations, to produce a numerical estimate of the
      // digital option price.

      // Generating our CEV stock path using the Euler approximation .
      ST_Euler = S_T_Euler(myS0,myr,myT,mysigma,mygamma,mym);

      // Generating our CEV stock path using the Milstein approximation .
      ST_Milstein = S_T_Milstein(myS0,myr,myT,mysigma,mygamma,mym);

      // Our Indicator Functions.
      if( ST_Euler > myH ){
         temp = (myC*exp(-1*myr*myT));
         sumEuler += temp;
         sumEulerSquared += (temp*temp);
      }
      if( ST_Milstein > myH ){
         temp = (myC*exp(-1*myr*myT));
         sumMilstein += temp;
         sumMilsteinSquared += (temp*temp);
      }
   }

   myEulerEstimate = sumEuler/myn;       // Our Euler CEV Monte-Carlo Estimate.
```

```
    myMilsteinEstimate = sumMilstein/myn; // Our Euler CEV Monte-Carlo Estimate.



    // Calculating Option asymptotic Variance.
    myEulerVariance = ( sumEulerSquared - (myn*myEulerEstimate*myEulerEstimate) )/(myn*(myn-
1));
    myMilsteinVariance = ( sumMilsteinSquared - (myn*myMilsteinEstimate*myMilsteinEstimate)
)/(myn*(myn-1)); // assigning value to respective private member variable.


    return myMilsteinEstimate;
}


double CEVMonteCarlo::ConfidenceInterval(double epsilon){
    // Calculating Confidence Interval.
    // Function returns the Milstein interval width.


    double alpha = InverseCumulativeNormal(1.0 - (epsilon/2.0));


    // Calculating Euler Interval
    EulerLeftCI = myEulerEstimate - alpha*sqrt(myEulerVariance);  // assigning values to respective
    EulerRightCI = myEulerEstimate + alpha*sqrt(myEulerVariance); // private member variables.


    // Calculating Milstein Interval
    MilsteinLeftCI = myMilsteinEstimate - alpha*sqrt(myMilsteinVariance);  // assigning values to
respective
    MilsteinRightCI = myMilsteinEstimate + alpha*sqrt(myMilsteinVariance); // private member
variables.
    EulerCIWidth = EulerRightCI - EulerLeftCI;
    MilsteinCIWidth = MilsteinRightCI - MilsteinLeftCI;



    return MilsteinCIWidth; // The Milstein scheme CI interval width.
```

```
}

CEVControlVariates::CEVControlVariates(double C, double H, double S0, double r, double T, double
sigma, int n,double gamma, int m){
   // Initialising private member variables.
      myC     = C;
      myH     = H;
      myS0    = S0;
      myr     = r;
      myT     = T;
      mysigma = sigma;
      myn     = n;
      mygamma = gamma;
      mym     = m;

}

double CEVControlVariates::bstarhat(void){

   double numeratorSum = 0, denominatorSum = 0;
   double Xi,Yi;

   // Calculating EX and EY =(approx)= Monte-Carlo(Y).
   double EX = exp(myr*myT)*myS0;
   CEVMonteCarlo CEV(myC,myH,myS0,myr,myT,mysigma,myn,mygamma,mym);
   double EY = CEV.getEstimate();

   for(int i = 0; i < myn; i++){
      Xi = S_T_Milstein(myS0,myr,myT,mysigma,mygamma,mym);
      Yi = 0;
      if( Xi > myH ){
```

```
            Yi = myC*exp(-1*myr*myT);
        }
        numeratorSum += ((Xi - EX)*(Yi - EY));
        denominatorSum += ((Xi - EX)*(Xi - EX));
    }
    return numeratorSum/denominatorSum; // Our B*hat estimate.

}

double CEVControlVariates::getEstimate(double b){

    double yib, ST, CVEstimate, sum = 0, squaresum = 0;
    double yi,Xi, EX = exp(myr*myT)*myS0; // E[X] = E[ST].
    for(int i = 0; i < myn; i++){

        // Generating Our stock path.
        ST = S_T_Milstein(myS0,myr,myT,mysigma,mygamma,mym);

        // Calculating Yi.
        yi = 0;
        if( ST > myH ){
            yi = myC*exp(-1*myr*myT);
        }

        // Calculating Yi(b).
        Xi = ST;
        yib = yi - b*(Xi - EX);

        sum += yib;
        squaresum += (yib*yib);
```

```
    }


    // Calculating statistics.
    CVEstimate = sum/myn;
    myvariance = (squaresum - (myn*CVEstimate*CVEstimate))/(myn*(myn-1)); // Our sample variance estimate of our Control Variates estimator.


    return CVEstimate; // Our Control Variates Estimate.
}
```

```
// RandomNumberGenerators.cpp

#include "RandomNumberGenerators.h"
#include<cstdlib>
#include<cmath>

double myuniform(void){
   int myrand = rand();
   while((myrand==0)||(myrand==RAND_MAX)){myrand = rand();}
   double myuni = myrand/static_cast<double>(RAND_MAX);
   return myuni;
}

double uniformab(double a,double b){
   double u = myuniform();
   return a + ((b-a)*u);
}

double exponential_rv(double lambda){
   double u = myuniform();
   return -1*log(u)/lambda;
}

double normal_rv(double mu, double sigma){
   double myuni1 = myuniform(), myuni2 = myuniform();
   double myexp = -2.0*log(myuni1);
   double mytheta = 2.0*M_PI*myuni2;
   double normal = mu + sigma*(sqrt(myexp)*cos(mytheta));
   return normal;
}
```

```
//
//
//              Normals.cpp
//
//
/*
code to implement the basic distribution functions necessary in mathematical finance
via rational approximations
 */

#include <cmath>
#include "Normals.h"

// the basic math functions should be in namespace std but aren't in VCPP6
#if !defined(_MSC_VER)
using namespace std;
#endif

const double OneOverRootTwoPi = 0.398942280401433;

// probability density for a standard Gaussian distribution
double NormalDensity(double x)
{
   return OneOverRootTwoPi*exp(-x*x/2);
}

// the InverseCumulativeNormal function via the Beasley-Springer/Moro approximation
double InverseCumulativeNormal(double u)
{
```

```
  static double a[4]={  2.50662823884,
                       -18.61500062529,
                        41.39119773534,
                       -25.44106049637};

  static double b[4]={-8.47351093090,
                       23.08336743743,
                      -21.06224101826,
                        3.13082909833};

  static double c[9]={0.3374754822726147,
                       0.9761690190917186,
                       0.1607979714918209,
                       0.0276438810333863,
                       0.0038405729373609,
                       0.0003951896511919,
                       0.0000321767881768,
                       0.0000002888167364,
                       0.0000003960315187};


  double x=u-0.5;
  double r;

  if (fabs(x)<0.42) // Beasley-Springer
  {
     double y=x*x;

     r=x*(((a[3]*y+a[2])*y+a[1])*y+a[0])/
           ((((b[3]*y+b[2])*y+b[1])*y+b[0])*y+1.0);
```

```
    }
    else // Moro
    {

        r=u;

        if (x>0.0)
            r=1.0-u;

        r=log(-log(r));

        r=c[0]+r*(c[1]+r*(c[2]+r*(c[3]+r*(c[4]+r*(c[5]+r*(c[6]+
                r*(c[7]+r*c[8])))))));

        if (x<0.0)
            r=-r;

    }

    return r;
}


// standard normal cumulative distribution function
double CumulativeNormal(double x)
{
    static double a[5] = { 0.319381530,
                          -0.356563782,
                           1.781477937,
                          -1.821255978,
                           1.330274429};
```

```
    double result;

    if (x<-7.0)
        result = NormalDensity(x)/sqrt(1.+x*x);

    else
    {
        if (x>7.0)
            result = 1.0 - CumulativeNormal(-x);
        else
        {
            double tmp = 1.0/(1.0+0.2316419*fabs(x));

            result=1-NormalDensity(x)*
                    (tmp*(a[0]+tmp*(a[1]+tmp*(a[2]+tmp*(a[3]+tmp*a[4])))));

            if (x<=0.0)
                result=1.0-result;

        }
    }

    return result;
}

/*
 *
 * Copyright (c) 2002
 * Mark Joshi
 *
```

```
 * Permission to use, copy, modify, distribute and sell this
 * software for any purpose is hereby
 * granted without fee, provided that the above copyright notice
 * appear in all copies and that both that copyright notice and
 * this permission notice appear in supporting documentation.
 * Mark Joshi makes no representations about the
 * suitability of this software for any purpose. It is provided
 * "as is" without express or implied warranty.
*/
```

```
//Main.cpp
// *** Main file for MA323 Project Q3 & Q4 -- Candidate Number: 40946 *** //

using namespace std;

// Header Files
#include "Normals.h"
#include "Question3.h"
#include "Question4.h"
#include "RandomNumberGenerators.h"

int main(){
   srand(time(NULL));

   // Setting Global Variables //
   int n = 10000, m = 1000;                                    // Simulation Variables.
   double C = 15, H = 10, S0 = 10, r = 0.01, T = 1, sigma = 0.2, gamma = 0.75; // Option Variables.



   // *** QUESTION 3 CODE *** //



   // Calculating Analytical Black-Scholes Solution //
   double ANSolution = analytical(C,H,S0,r,T,sigma);

   // Calculating Monte-Carlo estimate and Confidence Intervals //
   MonteCarlo MC(C,H,S0,r,T,sigma,n);                  // Our Monte-Carlo Object.
   double MCEstimate = MC.getEstimate(), MCvar = MC.myanalyticalVariance;

   // Calculating 95% CI
   double width95 = MC.ConfidenceInterval(0.05); // epsilon = 0.05
```

```
    double left95 = MC.myleftCI, right95 = MC.myrightCI;

    // Calculating 99% CI
    double width99 = MC.ConfidenceInterval(0.01); // epsilon = 0.01
    double left99 = MC.myleftCI, right99 = MC.myrightCI;

    // Calculating Control-Variates estimate //
    ControlVariates CV(C,H,S0,r,T,sigma,n);
    double CVEstimate = CV.getEstimate(), CVvar = CV.myvariance;

    // QUESTION 3 OUTPUT STREAM //
    cout<<"Question 3\n\n";
    cout<<"Pricing Method\t\tPrice\t\tVariance\t95% CI\t\t\t99% CI";
    cout<<"\n\nAnalytical\t\t"<<ANSolution<<"\t\t"<<0.0;
    cout<<"\n\nMonte-
Carlo*\t\t"<<MCEstimate<<"\t\t"<<MCvar<<"\t("<<left95<<","<<right95<<")\t("<<left99<<","<<right
99<<")";
    cout<<"\n\nControl Variates*\t"<<CVEstimate<<"\t\t"<<CVvar;
    cout<<"\n\n95% Monte-Carlo CI width: "<<width95;
    cout<<"\n99% Monte-Carlo CI width: "<<width99;
    cout<<"\nControl Variates variance reduction (wrt Monte-Carlo Estimation) of: "<<(1.0 -
CVvar/MCvar)*100<<"%";
    cout<<"\n\n*Using "<<n<<" iterations";

    // *** QUESTION 4 CODE *** //


    cout<<"\n\n\n\nQuestion 4 (May take a while to compute)\n\n";

    // Calculating Monte-Carlo estimate and Confidence Intervals using both CEV asset pricing
schemes //
    CEVMonteCarlo CEV_MC(C,H,S0,r,T,sigma,n,gamma,m); // Our CEV Monte-Carlo Object.
```

```
   double MilsteinSchemeMCEstimate = CEV_MC.getEstimate();
   double EulerSchemeMCEstimate = CEV_MC.myEulerEstimate;
   double EulerVar = CEV_MC.myEulerVariance, MilsteinVar = CEV_MC.myMilsteinVariance;

   // Calculating 95% CIs
   double MilsteinWidth95 = CEV_MC.ConfidenceInterval(0.05), EulerWidth95 =
CEV_MC.EulerCIWidth;
   double EulerLeft95 = CEV_MC.EulerLeftCI, EulerRight95 = CEV_MC.EulerRightCI;
   double MilsteinLeft95 = CEV_MC.MilsteinLeftCI, MilsteinRight95 = CEV_MC.MilsteinRightCI;

   // Calculating 99% CIs
   double MilsteinWidth99 = CEV_MC.ConfidenceInterval(0.01), EulerWidth99 =
CEV_MC.EulerCIWidth;
   double EulerLeft99 = CEV_MC.EulerLeftCI, EulerRight99 = CEV_MC.EulerRightCI;
   double MilsteinLeft99 = CEV_MC.MilsteinLeftCI, MilsteinRight99 = CEV_MC.MilsteinRightCI;

   // Calculating CEV Milstein Control-Variates estimate //
   CEVControlVariates CEV_CV(C,H,S0,r,T,sigma,n,gamma,m); // Our CEV Monte-Carlo Object.
   double bstarhatCEV = CEV_CV.bstarhat(); // our b*hat i.e. the optimal choice of b.
   double b_SubOptimal = 1.0; // a sub-optimal choice for b.
   double CEVMilstein_bstarhat = CEV_CV.getEstimate(bstarhatCEV);
   double optimalVar = CEV_CV.myvariance;
   double CEVMilstein_b_SubOptimal = CEV_CV.getEstimate(b_SubOptimal);
   double subOptimalVar = CEV_CV.myvariance;

   // QUESTION 4 OUTPUT STREAM //
   cout<<"Pricing Method\t\t\t\tPrice\t\tVariance\t95% CI\t\t\t99% CI";
   cout<<"\n\nMonte-
Carlo**\t\t\t\t"<<EulerSchemeMCEstimate<<"\t\t"<<EulerVar<<"\t("<<EulerLeft95<<","<<EulerRigh
t95<<")\t("<<EulerLeft99<<","<<EulerRight99<<")";
   cout<<"\n(CEV & Euler approx)";
```

```
    cout<<"\n\nMonte-
Carlo**\t\t\t\t"<<MilsteinSchemeMCEstimate<<"\t\t"<<MilsteinVar<<"\t("<<MilsteinLeft95<<","<<
MilsteinRight95<<")\t("<<MilsteinLeft99<<","<<MilsteinRight99<<")";

    cout<<"\n(CEV & Milstein approx)";


    cout<<"\n\nOptimal b Control Variates**\t\t"<<CEVMilstein_bstarhat<<"\t\t"<<optimalVar;

    cout<<"\n(CEV & Milstein approx)";

    cout<<"\n\nSub-optimal b Control
Variates**\t"<<CEVMilstein_b_SubOptimal<<"\t\t"<<subOptimalVar;

    cout<<"\n(CEV & Milstein approx)";



    cout<<"\n\n95% Monte-Carlo (CEV Euler approx) CI width: "<<EulerWidth95;

    cout<<"\n99% Monte-Carlo (CEV Euler approx) CI width: "<<EulerWidth99;

    cout<<"\n95% Monte-Carlo (CEV Milstein approx) CI width: "<<MilsteinWidth95;

    cout<<"\n99% Monte-Carlo (CEV Milstein approx) CI width: "<<MilsteinWidth99;

    cout<<"\n\nOptimal b = b*hat = "<<bstarhatCEV;

    cout<<"\nSub-optimal b = "<<b_SubOptimal;

    cout<<"\nControl Variates Optimal b variance reduction (wrt CEV Monte-Carlo Estimation) of:
"<<(1.0 - optimalVar/MilsteinVar)*100<<"%";

    cout<<"\nControl Variates Sub-optimal b variance reduction (wrt CEV Monte-Carlo Estimation) of:
"<<(1.0 - subOptimalVar/MilsteinVar)*100<<"%";

    cout<<"\nControl Variates Optimal b variance reduction wrt Sub-optimal b of: "<<(1.0 -
optimalVar/subOptimalVar)*100<<"%";


    cout<<"\n\n**Using "<<n<<" iterations and "<<m<<" grid partitions\n\n";

    return 0;

}
```