Pricing Asian Options Using Monte Carlo

Joy Tolia

Contents

1 Introduction

- The objective of this assignment is to implement Monte-Carlo methods within Matlab to price different Asian options and to compare the different results.
- I chose Matlab as I have used it before and I thought it would be interesting to find out how Monte-Carlo will behave in Matlab.

1.1 Implementation

- Matlab is very fast at doing array operations, much faster than using for loops. So I wanted to find a way to have as much of my implementation as possible using array operations.
- The problem becomes memory. I found that the computer could only handle arrays with 10^7 elements at once.
- First the decide the size δt and anything to do with averaging across time will be done with array operations. Then using the information about my computer and its memory restrictions, I would do the averaging across samples in chunks. For examples if $\delta t = 10^{-2}$ then I would choose 10^5 for the number of samples in each chunk to average over.
- By doing this I am hoping to get the most of out Matlab in terms of speed.

1.2 Types of Errors

- There are 3 types of errors we would like to analyse; error from approximating the underlying model, error from averaging over samples and error from size of the timestep δt.
- We will get error from approximating the underlying model as we will be comparing Euler-Maruyama and Milstein schemes. Note that for our underlying model we do have a closed form formula:
$$S_{t+\delta t} = S_t e^{\left(r-\sigma^2/2\right)\delta t + \sigma\phi\sqrt{\delta t}}, \quad \phi \sim \mathrm{N}\left(0,1\right)$$
However, it will not always be the case where we will find a closed form solution for the underlying.
- We will get error from averaging over samples which is the main error in Monte-Carlo methods and will be looking at the antithetics scheme to see how they affect the number of samples needed to get an accurate answer.
- We will assume we want to price continuous Asian options hence our $\delta t \to 0$, however we will have to take $\delta t > 0$ for example $\delta t = 0.01$ which will give us an error from averaging over time.

1.3 Error Analysis

- Having spent quite a while trying to work out the best way to present results, the hardest part was to assume we didn't have exact solutions and to have a way to working out if the solution converged or not.

- We use the following method to check if a solution has converged; we keep getting chunks of new samples and keeping track of the running average. we say a solution is reached once the standard deviation of the last 1000 elements of the running average is within some tolerance ϵ. we will use $\epsilon = 10^{-5}$.

2 Model Error

In this section, we will look at the error that we will get from approximating the underlying model, as we have the closed form formula in this case, we can compare that to the approximations. Let $\phi \sim \mathrm{N}(0,1)$, then we have the following iterative formula that we can implement:

$$S_{t+\delta t} = S_t e^{\left(r-\sigma^2/2\right)\delta t+\sigma\phi\sqrt{\delta t}} \tag{1}$$

$$S_{t+\delta t} = S_t + rS_t\delta t + \sigma S_t\phi\sqrt{\delta t} \tag{2}$$

$$S_{t+\delta t} = S_t + rS_t\delta t + \sigma S_t\phi\sqrt{\delta t} + \frac{\sigma^2 S_t}{2}\left(\phi^2-1\right)\delta t \tag{3}$$

Where Equation (1) is the closed form solution, Equation (2) is the Euler-Maruyama approximation and Equation (3) is the Milstein approximation. More information on the derivations on each of the formula can be found is Section 6.1. Let us expand the closed form solution to be able to compare it with the approximations:

$$\begin{aligned} S_{t+\delta t} &= S_t e^{\left(r-\sigma^2/2\right)\delta t+\sigma\phi\sqrt{\delta t}} \\ &= S_t\left(1+\left(r-\frac{\sigma^2}{2}\right)\delta t+\sigma\phi\sqrt{\delta t}+\frac{\sigma^2}{2}\phi^2\delta t+O\left(\delta t^{3/2}\right)\right) \\ &= \underbrace{S_t + rS_t\delta t + \sigma S_t\phi\sqrt{\delta t}}_{\text{Euler-Maruyama approximation}} + \underbrace{\frac{\sigma^2 S_t}{2}\left(\phi^2-1\right)\delta t}_{\text{Miltstein correction}} + O\left(\delta t^{3/2}\right) \end{aligned}$$

From the previous calculation we can see intuitively that the Euler-Maruyama has an order of convergence of $\sqrt{\delta t}$ and that the Milstein approximation has this extra correction term which gives it an order of convergence of δt which is better than the Euler-Maruyama approximation. Let us now test this.

At first I thought I could distinguish between error from the size of the time step and error from approximating the model however, we both affect each other so I have to analyse them together.

2.1 Implementation

- We will run all three method together so we will be using the same samples for each method to make them fully comparable.
- As described earlier, we will define the error for convergence as the standard deviation of the last 100 running averages and we will say the solution has converged where the error is below 10^{-3}.

- In this case, we will take the maximum standard deviation of all three methods so each method will have the same number of samples in the end.
- All the code used in this section can be found in Section 1 within the Code document.
- We define the error for the different methods as the absolute difference between the converged solution of Euler-Maruyama (Equation (2)) and Milstein (Equation (3)) method versus the benchmark case of the closed form (Equation (1)).

2.2 Results

The full set of results for each option can be found in Section 6.2. The following are the results for the arithmetic fixed strike Asian call and put options:

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0088	5.6402×10^{-4}	2.9598×10^{-4}	4.3800×10^{-5}	3.3715×10^{-5}
Milstein	0.0233	0.0026	2.5500×10^{-4}	2.5847×10^{-5}	2.5360×10^{-6}

Table 1: Error from the benchmark closed form solution for the arithmetic fixed strike Asian call option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	3.3435×10^{-4}	9.5593×10^{-4}	1.1241×10^{-4}	6.3920×10^{-5}	3.7049×10^{-5}
Milstein	0.0184	0.0019	1.9486×10^{-4}	1.9174×10^{-5}	1.9412×10^{-6}

Table 2: Error from the benchmark closed form solution for the arithmetic fixed strike Asian put option

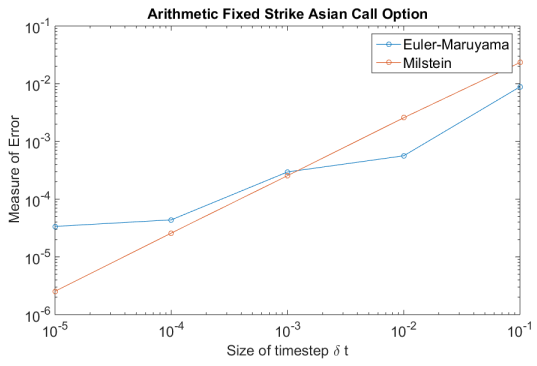

Figure 1: Error from the benchmark closed form solution for the arithmetic fixed strike Asian call option

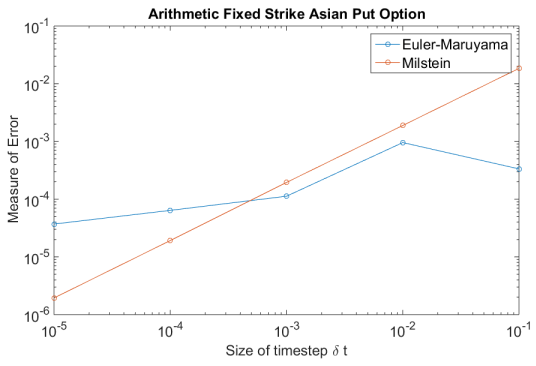

Figure 2: Error from the benchmark closed form solution for the arithmetic fixed strike Asian put option

2.3 Conclusion

- The error we talk about in this part is the error of Euler-Maruyama (Equation (2)) and Milstein (Equation (3)) methods versus the benchmark case which is the closed form method (Equation (1)).
- The results for each options are fairly similar which means the conclusion is robust.
- We find that with the Milstein method the error reduces linearly with size of the timestep δt as seen from the straight lines in the error plots (Figures 1 and 2). This is what we found at the start of the section with Milstein having an order of convergence of δt.
- We find that the Euler-Maruyama method is a bit less predictive and it was hard to say how it changes with the size of the timestep δt.
- The interesting result is that Euler-Maruyama method has smaller error until $\delta t = 10^{-3}$ then the Milstein method but then the Milstein method has smaller error past that point as seen in Figure 1.
- The conclusion from this is that if you are using $\delta t < 10^{-3}$ then to use the Euler-Maruyama method and for $\delta t > 10^{-3}$ then to use the Milstein Method.

3 Rate of Convergence

In this section we will use the antithetics method and compare that to the regular Monte-Carlo. Antithetics leverages on the fact that a standard normal random variable X is symmetric which means we have for a payoff function f, we have $\mathbb{E}[f(X)] = \mathbb{E}[f(-X)]$ hence we can use the following:

$$\mathbb{E}[f(X)] = \mathbb{E}\left[\frac{f(X) + f(-X)}{2}\right]$$

This means that given one standard normal sample, we can use it twice by taking the negative of that sample. But what would be interesting is that if this would mean if we would converge double the rate as in essence we have double the samples. Looking at the variance of the antithetics method we have the following:

$$\begin{aligned}
\mathbb{V}\left[\frac{f(X) + f(-X)}{2}\right] &= \frac{1}{4}\left(\mathbb{V}[f(X)] + \mathbb{V}[f(-X)] + 2\mathrm{Cov}[f(X), f(-X)]\right) \\
&= \frac{1}{2}\left(\mathbb{V}[f(X)] + \rho\mathbb{V}[f(X)]\right) \\
&= \frac{1}{2}\mathbb{V}[f(X)](1 + \rho)
\end{aligned}$$

Where ρ is the correlation between $f(X)$ and $f(-X)$. This means that the variance of antithetics is only lower than the standard Monte-Carlo method when the correlation is negative.

In our case, the correlation will be negative as a negative sample usually means the lower simulated path hence the payoff is lower whereas a positive sample means higher simulated path hence a

higher payoff.

We have calculated the correlations between the payoffs from samples and their negative equivalents for 10^4 samples with $\delta t = 10^{-3}$ to get an understanding:

	Call				Put			
	Arith		Geom		Arith		Geom	
	Fix	Float	Fix	Float	Fix	Float	Fix	Float
ρ	-0.52	-0.48	-0.51	-0.47	-0.41	-0.44	-0.41	-0.43

Table 3: Correlations between the payoffs of 10^{-4} samples and their negative equivalents with $\delta t = 10^{-3}$

From this we can see that the call options has a more negative correlation than the put options. So we should see that the call options should converge faster than the put options. Also no note that as $\rho \approx -0.5$, we would have quartered the variance hence halved the standard deviation for these options.

3.1 Implementation

- We will use the same definition of convergence error as before but we will use the standard deviation of the last 1000 running averages from Monte Carlo. We say that we have converged to a solution when this error is less than 10^{-4}.
- We will keep the size of the timestep δt constant at 10^{-3} as we are analysing the rate of convergence more than the accuracy of the results.
- We will use the Milstein method to model the underlying.
- All the code used in this section can be found in Section 2 within the Code document.

3.2 Results

The full set of results can be found in Section 6.3. Here are the results for the arithmetic fixed strike Asian options:

	Call		Put	
	Standard	Antithetic	Standard	Antithetic
Final Value	5.7616	5.7654	3.3468	3.3462
Final Error ($\times 10^{-6}$)	9.9	7.7	9.7	8.2
Samples ($\times 10^5$)	42.4	15.2	30.4	12.8
Time (seconds)	253	154	124	91

Table 4: Data for the arithmetic fixed strike Asian options

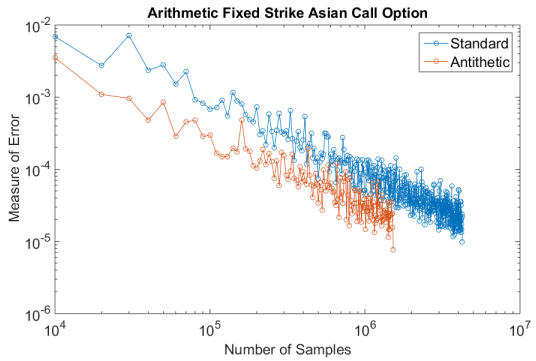

Figure 3: Convergence error measure for the arithmetic fixed strike Asian call option

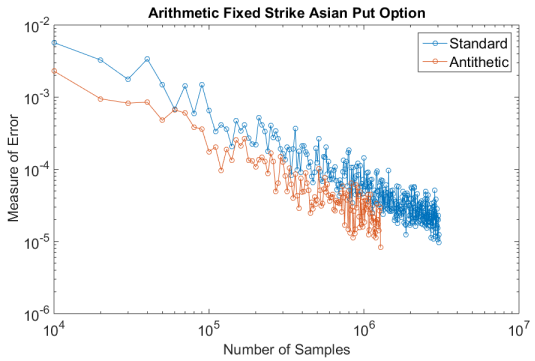

Figure 4: Convergence error measure for the arithmetic fixed strike Asian put option

3.3 Conclusion

- We find similar results across all options which again shows that we can interpret our results with confidence.
- We see that the antithetics method works very well from our results, as seen in Figures 3 and 4.
- We see that the number of samples needed to achieve convergence have more than halved as expected as seen in Table 4.
- We also find that the antithetics method works slightly better on call options than the put options as hypothesised earlier.

4 Size of Timestep δt

In this section, we will analyse how the size of the timestep affects accuracy of the results.

4.1 Implementation

- We will use the closed form solution for the underlying as we want to get rid of the error from approximating the underlying.
- We will only work with one option in this case which is the geometric fixed strike Asian call option and assume that the exact solution is 5.5468 which is from [1].
- Once the solution has converged, we will compared the converged solution with the exact solution and look at the absolute difference between the two and use that as our error.
- All the code for this part can be found in Section 3 of the code document.

4.2 Results

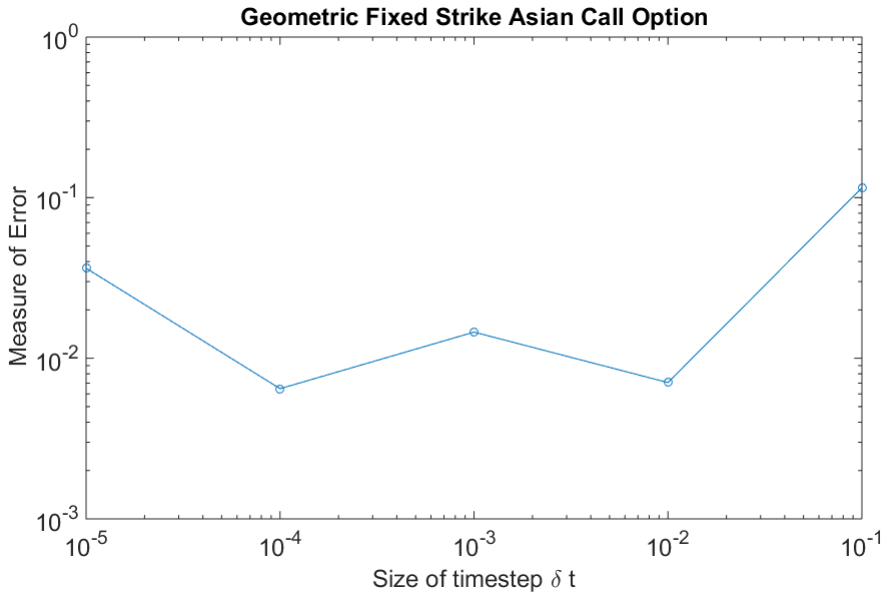

Figure 5: Error of converged solution versus the exact solution for different timesteps

4.3 Conclusion

- Having run this several times and getting results similar to Figure 5, we conclude that the results are weak and hard to interpret.
- We would have expected the error to reduce as δt went down. However, the error from Monte-Carlo of using random samples might outweigh the error from the timestep δt making it hard to see the affect of δt on the accuracy of the results.

5 Multi Level Monte-Carlo

In summer 2015, I did a research project under a supervisor where we will looking at calculating values for stochastic integrals. One of the papers I came across was [2] on Multi Level Monte-Carlo which looked very interesting as it has a lower computational complexity than the standard Monte-Carlo but I never got to implement it. I have taken the chance in this assignment to implement the algorithm.

5.1 Methodology

Consider Monte Carlo path simulations with different timesteps $h_l = M^{-l}T$, $l = 0, 1, \ldots, L$. Let P denote the discounted risk neutral payoff and $\hat{P}_l$ denote approximations of P using a numerical discretisation with timestep h_l. We have:

$$\mathbb{E}\left[\hat{P}_L\right] = \mathbb{E}\left[\hat{P}_0\right] + \sum_{l=1}^{L} \mathbb{E}\left[\hat{P}_l - \hat{P}_{l-1}\right]$$

Let $\hat{Y}_0$ be the estimator for $\mathbb{E}\left[\hat{P}_0\right]$ using N_0 samples and $\hat{Y}_l$, for $l > 0$, be the estimator for $\mathbb{E}\left[\hat{P}_l - \hat{P}_{l-1}\right]$ using N_l samples. For us it is:

$$\hat{Y}_0 = \frac{1}{N_0}\sum_{i=1}^{N_0} \hat{P}_0^{(i)}$$

$$\hat{Y}_l = \frac{1}{N_l}\sum_{i=1}^{N_l} \left(\hat{P}_l^{(i)} - \hat{P}_{l-1}^{(i)}\right), \quad l > 0$$

The key point here is that the quantity $\hat{P}_l^{(i)} - \hat{P}_{l-1}^{(i)}$ comes from two discrete approximations with different timesteps but the same Brownian motion which reduces the variance. This is easily implemented by first constructing the Brownian increments for the simulation of the discrete path leading to the evaluation of $\hat{P}_l^{(i)}$ and then summing them in groups of M to give discrete Brownian increments for the evaluation of $\hat{P}_{l-1}^{(i)}$.

5.2 Implementation

The following is the algorithm we will follow completely based from [2] where: The equation for the optimal N_l is:

$$N_l = \left\lceil 2\epsilon^{-2}\sqrt{V_l h_l}\left(\sum_{l=0}^{k}\sqrt{\frac{V_k}{h_k}}\right)\right\rceil \tag{4}$$

Where ϵ is a user defined level of accuracy and V_l is the variance of $\left(\hat{P}_0^{(i)}\right)_{i\leq N_l}$ for $l = 0$ and $\left(\hat{P}_l^{(i)} - \hat{P}_{l-1}^{(i)}\right)_{i\leq N_l}$ for $l > 0$.

Step 1. Start with L = 0.
Step 2. Estimate V_L using an initial set of $N_L = 10^4$ samples.
Step 3. Define the optimal N_l, $l = 0, \ldots, L$, using Equation (4).
Step 4. Evaluate extra sample at each level as needed fo new N_l.
Step 5. If $L \geq 2$, test for convergence using Equation (11) from [2].
Step 6. If $L < 2$ or it is not converged, set $L := L + 1$ and go to Step 2.

- We define convergence as shown in Equation (11) from [2].
- In this part, we use the close form solution for the underlying model.
- All the code can be seen in Section 4 of the code document.

5.3 Results & Conclusion

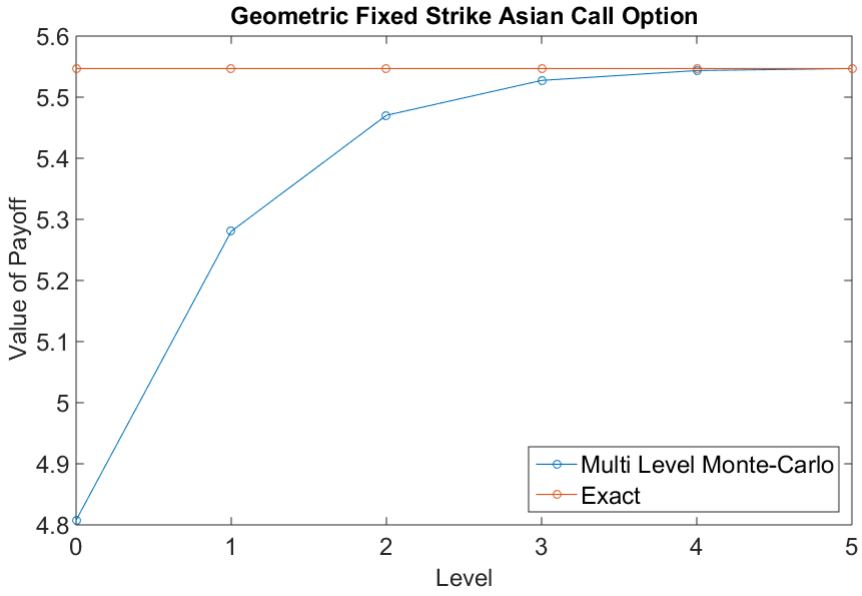

Figure 6: Value from Multi Level Monte-Carlo with increasing levels

- The code ran in less than two seconds due to the vectorisation from Matlab and as the algorithm works well.
- Accuracy was set to $\epsilon = 0.01$ which is fairly low. Even though Figure 6 shows very good convergence, that was for only one run, there is still error (as expected) from different runs.
- One problem currently is when ϵ is increased, my computer cannot handle the sizes of arrays produced in the algorithm which can be tackled by using for loops rather than array operations. However, speed will be lost in doing so.
- As each level greater than 0 is a difference between two discretisations, the variance is lower because both discretisation uses the same Brownian paths, making this the principal reason that it is better than the standard Monte-Carlo.

6 Appendix

6.1 Underlying Model

The following is the stochastic process that we assume the underlying asset follows in a risk neutral world:

$$dS_t = rS_t dt + \sigma S_t dW_t$$

where $r, \sigma \in \mathbb{R}$ and $W_t = (W_t : t > 0)$ is a standard Brownian motion. In this case, we have a closed form solution for the $S_{t+\delta t}$. We get this by using Ito's lemma on $\log(S_t)$:

$$\begin{aligned} d(\log(S_t)) &= \frac{\partial}{\partial t}(\log(S_t))\, dt + \frac{\partial}{\partial S_t}(\log(S_t))\, dS_t + \frac{1}{2}\frac{\partial^2}{\partial S_t^2}(\log(S_t))\, dS_t^2 \\ &= \frac{1}{S_t}(rS_t dt + \sigma S_t dW_t) - \frac{1}{2S_t^2}\sigma^2 S_t^2 dt \\ &= \left(r - \frac{\sigma^2}{2}\right) dt + \sigma dW_t \end{aligned}$$

Using the integral equivalent over a timestep δt of the above relation gives the following equation:

$$\log(S_{t+\delta t}) - \log(S_t) = \left(r - \frac{\sigma^2}{2}\right)\delta t + \sigma(W_{t+\delta t} - W_t)$$

Finally as $(W_{t+\delta t} - W_t) \sim \mathrm{N}(0, \delta t)$, letting $\phi \sim \mathrm{N}(0,1)$, we have the following:

$$S_{t+\delta t} = S_t \exp\left(\left(r - \frac{\sigma^2}{2}\right)\delta t + \sigma\phi\sqrt{\delta t}\right) \tag{5}$$

This is what we would like to implement for the underlying model if we were pricing using Monte-Carlo methods. However, this could be computationally costly due to the exponential function and the model could be more complex and might not have a closed form solution so there are other schemes that we can use. Let us start with a more general model:

$$dS_t = a(S_t, t)\, dt + b(S_t, t)\, dW_t$$

Now using the integral equivalence of the above formula:

$$S_{t+\delta t} = S_t + \int_t^{t+\delta t} a(S_s, s)\, ds + \int_t^{t+\delta t} b(S_s, s)\, dW_s \tag{6}$$

Using Ito's for $a(S_s, s)$ and $b(S_s, s)$, we have the following:

$$\begin{aligned} a(S_s, s) &= a(S_t, t) + \int_t^s \left(\frac{\partial a}{\partial u}(S_u, u) + a(S_u, u)\frac{\partial a}{\partial S_u}(S_u, u) + \frac{1}{2}b^2(S_u, u)\frac{\partial^2 a}{\partial S_u^2}(S_u, u)\right) du + \cdots \\ &\quad \cdots + \int_t^s b(S_u, u)\frac{\partial a}{\partial S_u}(S_u, u)\, dW_u \end{aligned}$$

Similarly:

$$
\begin{aligned}
b\left(S_s, s\right) &= b\left(S_t, t\right) + \int_t^s \left(\frac{\partial b}{\partial u}\left(S_u, u\right) + a\left(S_u, u\right) \frac{\partial b}{\partial S_u}\left(S_u, u\right) + \frac{1}{2} b^2\left(S_u, u\right) \frac{\partial^2 b}{\partial S_u^2}\left(S_u, u\right) \right) du + \cdots \\
&\quad \cdots + \int_t^s b\left(S_u, u\right) \frac{\partial b}{\partial S_u}\left(S_u, u\right) dW_u
\end{aligned}
$$

Now we can sub in $a\left(S_u, u\right) = rS_u$ and $b\left(S_u, u\right) = \sigma S_u$ to get the following:

$$
\frac{\partial a}{\partial u}\left(S_u, u\right) = 0, \quad \frac{\partial a}{\partial S_u}\left(S_u, u\right) = r, \quad \frac{\partial^2 a}{\partial S_u^2}\left(S_u, u\right) = 0
$$

$$
\frac{\partial b}{\partial u}\left(S_u, u\right) = 0, \quad \frac{\partial b}{\partial S_u}\left(S_u, u\right) = \sigma, \quad \frac{\partial^2 b}{\partial S_u^2}\left(S_u, u\right) = 0
$$

Therefore we get:

$$
\begin{aligned}
a\left(S_s, s\right) &= rS_t + \int_t^s r^2 S_u du + \int_t^s r\sigma S_u dW_u \\
b\left(S_s, s\right) &= \sigma S_t + \int_t^s r\sigma S_u du + \int_t^s \sigma^2 S_u dW_u
\end{aligned}
$$

Finally subbing the above equation into Equation (6):

$$
\begin{aligned}
S_{t+\delta t} &= S_t + \int_t^{t+\delta t} a\left(S_s, s\right) ds + \int_t^{t+\delta t} b\left(S_s, s\right) dW_s \\
&= S_t + \int_t^{t+\delta t} \left(rS_t + \int_t^s r^2 S_u du + \int_t^s r\sigma S_u dW_u \right) ds + \cdots \\
&\quad \cdots + \int_t^{t+\delta t} \left(\sigma S_t + \int_t^s r\sigma S_u du + \int_t^s \sigma^2 S_u dW_u \right) dW_s \\
&= S_t + rS_t \delta t + \sigma S_t \left(W_{t+\delta t} - W_t \right) + \cdots \\
&\quad \cdots + \int_t^{t+\delta t} \int_t^s r^2 S_u \underbrace{du ds}_{O(\delta t^2)} + \int_t^{t+\delta t} \int_t^s r\sigma S_u \underbrace{dW_u ds}_{O\left(\delta t^{3/2}\right)} + \cdots \\
&\quad \cdots + \int_t^{t+\delta t} \int_t^s r\sigma S_u \underbrace{du dW_s}_{O\left(\delta t^{3/2}\right)} \pm \int_t^{t+\delta t} \int_t^s \sigma^2 S_u \underbrace{dW_u dW_s}_{O(\delta t)} \\
&= S_t + rS_t \delta t + \sigma S_t \left(W_{t+\delta t} - W_t \right) + \sigma^2 \int_t^{t+\delta t} \int_t^s S_u dW_u dW_s + O\left(\delta t^{3/2}\right)
\end{aligned}
$$

Let focus on the integral from above:

$$\begin{aligned}
\sigma^2 \int_t^{t+\delta t} \int_t^s S_u dW_u dW_s &= \sigma^2 \int_t^{t+\delta t} S_t \left(W_s - W_t\right) dW_s + O\left(\delta t\right) \\
&= \sigma^2 S_t \underbrace{\int_t^{t+\delta t} W_s dW_s}_{\text{Using Ito's} = \left(W_{t+\delta t}^2 - W_t^2 - \delta t\right)/2} - \sigma^2 S_t W_t \left(W_{t+\delta t} - W_t\right) + O\left(\delta t\right) \\
&= \frac{\sigma^2 S_t}{2}\left(W_{t+\delta t}^2 - W_t^2 - \delta t + 2W_t^2 - 2W_{t+\delta t} W_t\right) + O\left(\delta t\right) \\
&= \frac{\sigma^2 S_t}{2}\left(\left(W_{t+\delta t} - W_t\right)^2 - \delta t\right) + O\left(\delta t\right)
\end{aligned}$$

Therefore we have:

$$S_{t+\delta t} = S_t + rS_t \delta t + \sigma S_t \left(W_{t+\delta t} - W_t\right) + \frac{\sigma^2 S_t}{2}\left(\left(W_{t+\delta t} - W_t\right)^2 - \delta t\right) + O\left(\delta t\right)$$

Let $\phi \sim \mathrm{N}\left(0, 1\right)$ then:

$$\begin{aligned}
S_{t+\delta t} &\approx S_t + rS_t \delta t + \sigma S_t \phi \sqrt{\delta t} + \frac{\sigma^2 S_t}{2}\left(\left(\phi\sqrt{\delta t}\right)^2 - \delta t\right) \\
&= \underbrace{S_t + rS_t \delta t + \sigma S_t \phi \sqrt{\delta t}}_{\text{Euler-Maruyama approximation}} + \underbrace{\frac{\sigma^2 S_t}{2}\left(\phi^2 - 1\right)\delta t}_{\text{Milstein Correction}}
\end{aligned}$$

6.2 Model Error Results

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0088	5.6402×10^{-4}	2.9598×10^{-4}	4.3800×10^{-5}	3.3715×10^{-5}
Milstein	0.0233	0.0026	2.5500×10^{-4}	2.5847×10^{-5}	2.5360×10^{-6}

Table 5: Error from the benchmark closed form solution for the arithmetic fixed strike Asian call option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	3.3435×10^{-4}	9.5593×10^{-4}	1.1241×10^{-4}	6.3920×10^{-5}	3.7049×10^{-5}
Milstein	0.0184	0.0019	1.9486×10^{-4}	1.9174×10^{-5}	1.9412×10^{-6}

Table 6: Error from the benchmark closed form solution for the arithmetic fixed strike Asian put option

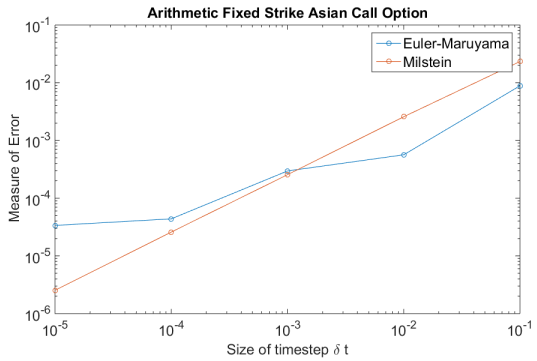

Figure 7: Error from the benchmark closed form solution for the arithmetic fixed strike Asian call option

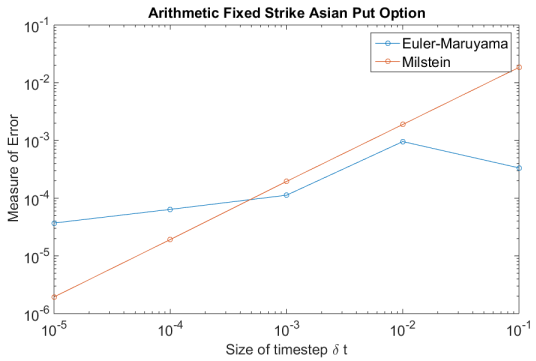

Figure 8: Error from the benchmark closed form solution for the arithmetic fixed strike Asian put option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0042	1.1359×10^{-4}	6.9972×10^{-4}	7.7009×10^{-5}	6.5832×10^{-5}
Milstein	0.0254	0.0026	2.6492×10^{-4}	2.6019×10^{-5}	2.5930×10^{-6}

Table 7: Error from the benchmark closed form solution for the arithmetic floating strike Asian call option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0024	0.0012	5.6068×10^{-4}	1.8113×10^{-5}	4.9870×10^{-5}
Milstein	0.0193	0.0020	1.9384×10^{-4}	1.9620×10^{-5}	1.9405×10^{-6}

Table 8: Error from the benchmark closed form solution for the arithmetic floating strike Asian put option

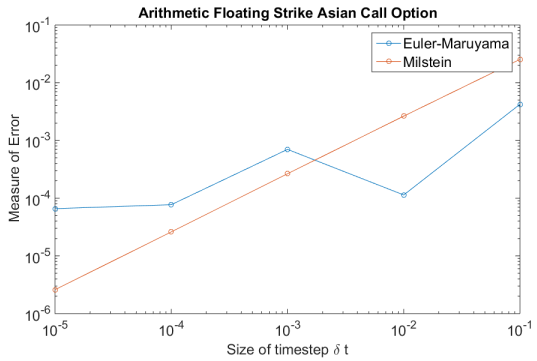

Figure 9: Error from the benchmark closed form solution for the arithmetic floating strike Asian call option

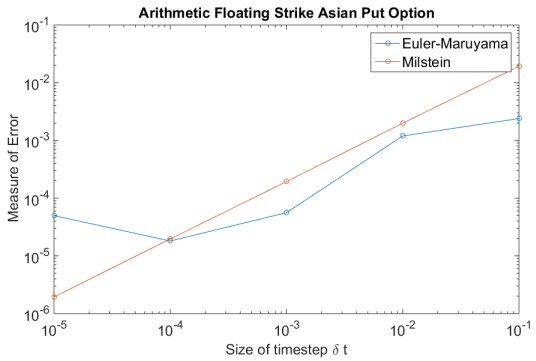

Figure 10: Error from the benchmark closed form solution for the arithmetic floating strike Asian put option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0045	0.0014	2.8011×10^{-4}	8.6369×10^{-5}	1.9019×10^{-5}
Milstein	0.0227	0.0023	2.3528×10^{-4}	2.3503×10^{-5}	2.3423×10^{-6}

Table 9: Error from the benchmark closed form solution for the geometric fixed strike Asian call option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	9.2513×10^{-5}	0.0014	2.2584×10^{-4}	7.5734×10^{-5}	8.7579×10^{-6}
Milstein	0.0197	0.0021	2.0697×10^{-4}	2.0652×10^{-5}	2.0156×10^{-6}

Table 10: Error from the benchmark closed form solution for the geometric fixed strike Asian put option

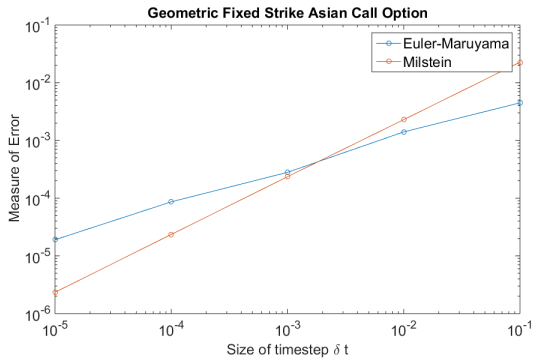

Figure 11: Error from the benchmark closed form solution for the geometric fixed strike Asian call option

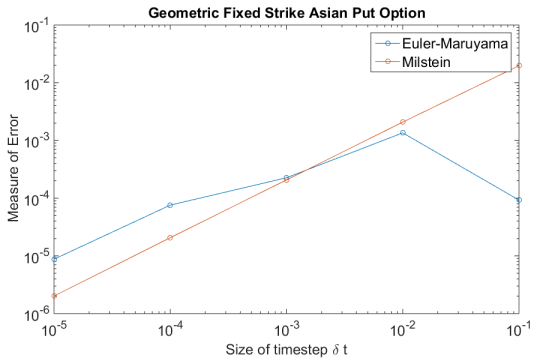

Figure 12: Error from the benchmark closed form solution for the geometric fixed strike Asian put option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0070	1.1555×10^{-4}	7.4598×10^{-5}	2.5574×10^{-5}	1.7367×10^{-5}
Milstein	0.0276	0.0028	2.8532×10^{-4}	2.8003×10^{-5}	2.8098×10^{-6}

Table 11: Error from the benchmark closed form solution for the geometric floating strike Asian call option

	δt				
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Euler-Maruyama	0.0020	3.4913×10^{-4}	7.7019×10^{-4}	8.1794×10^{-5}	3.4532×10^{-6}
Milstein	0.0171	0.0018	1.8251×10^{-4}	1.8463×10^{-5}	1.8594×10^{-6}

Table 12: Error from the benchmark closed form solution for the geometric floating strike Asian put option

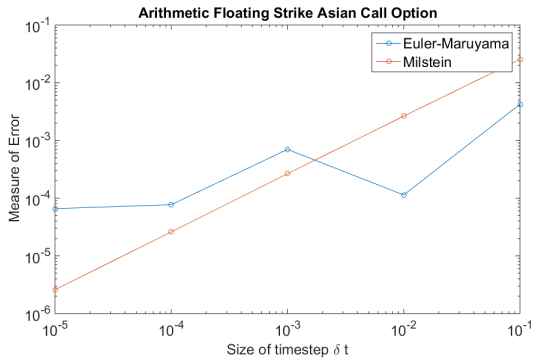

Figure 13: Error from the benchmark closed form solution for the geometric floating strike Asian call option

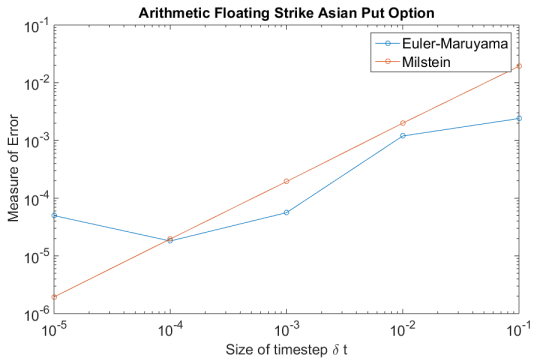

Figure 14: Error from the benchmark closed form solution for the geometric floating strike Asian put option

6.3 Rate of Convergence Results

6.3.1 Arithmetic - Fixed Strike

	Call		Put	
	Standard	Antithetic	Standard	Antithetic
Final Value	5.7616	5.7654	3.3468	3.3462
Final Error ($\times 10^{-6}$)	9.9	7.7	9.7	8.2
Samples ($\times 10^5$)	42.4	15.2	30.4	12.8
Time (seconds)	253	154	124	91

Table 13: Data for the arithmetic fixed strike Asian options

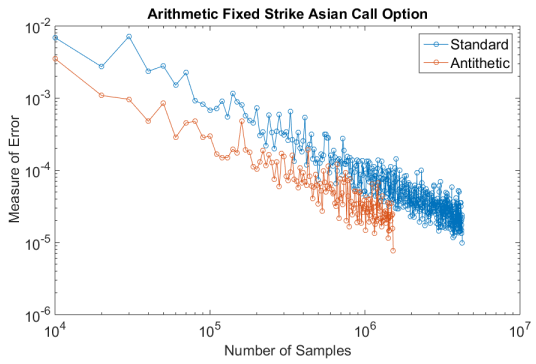

Figure 15: Convergence error measure for the arithmetic fixed strike Asian call option

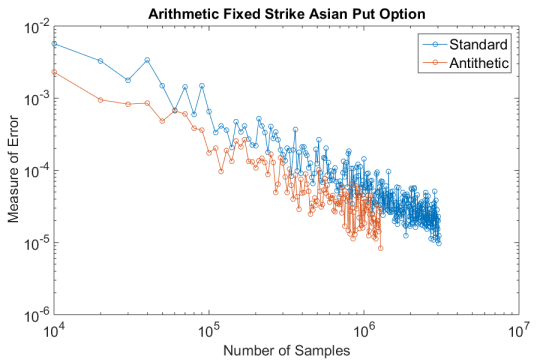

Figure 16: Convergence error measure for the arithmetic fixed strike Asian put option

6.3.2 Arithmetic - Floating Strike

	Call		Put	
	Standard	Antithetic	Standard	Antithetic
Final Value	5.8630	5.8589	3.4035	3.4027
Final Error ($\times 10^{-6}$)	8.9	9.9	9.9	9.6
Samples ($\times 10^5$)	46.9	18.4	26.8	13.1
Time (seconds)	152	124	141	118

Table 14: Data for the arithmetic floating strike Asian options

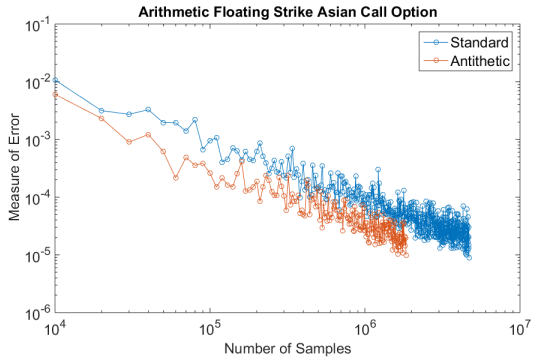

Figure 17: Convergence error measure for the arithmetic floating strike Asian call option

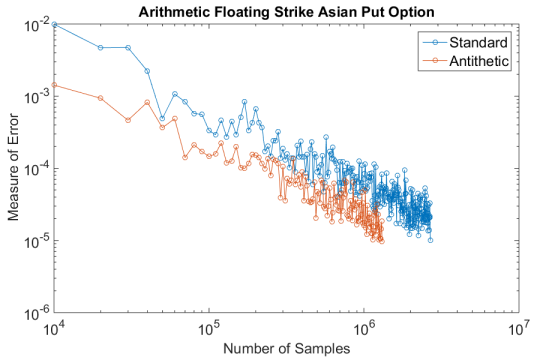

Figure 18: Convergence error measure for the arithmetic floating strike Asian put option

6.3.3 Geometric - Fixed Strike

	Call		Put	
	Standard	Antithetic	Standard	Antithetic
Final Value	5.5445	5.5427	3.4652	3.4639
Final Error ($\times 10^{-6}$)	9.9	9.8	9.8	5.7
Samples ($\times 10^5$)	35.5	17.9	24.2	14.2
Time (seconds)	271	243	149	149

Table 15: Data for the geometric fixed strike Asian options

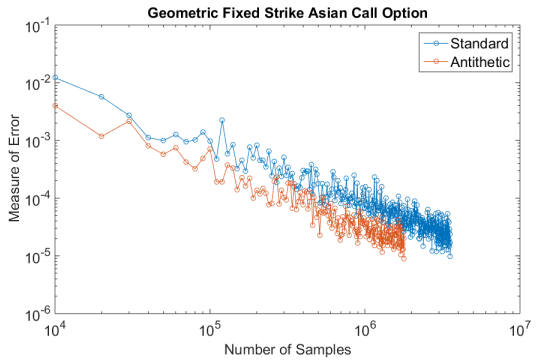

Figure 19: Convergence error measure for the geometric fixed strike Asian call option

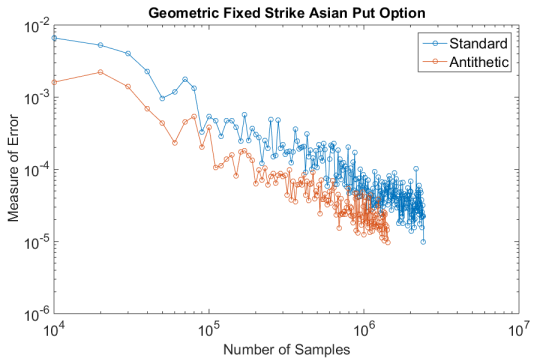

Figure 20: Convergence error measure for the geometric fixed strike Asian put option

6.3.4 Geometric - Floating Strike

	Call		Put	
	Standard	Antithetic	Standard	Antithetic
Final Value	6.0729	6.0752	3.2790	3.2762
Final Error ($\times 10^{-6}$)	9.1	9.2	9.8	9.9
Samples ($\times 10^{5}$)	48.8	16.1	35.9	12.5
Time (seconds)	345	215	182	148

Table 16: Data for the geometric floating strike Asian options

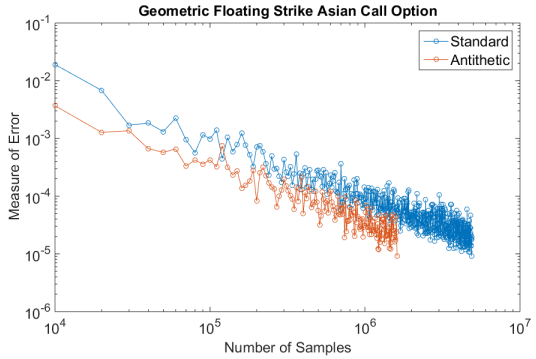

Figure 21: Convergence error measure for the geometric floating strike Asian call option

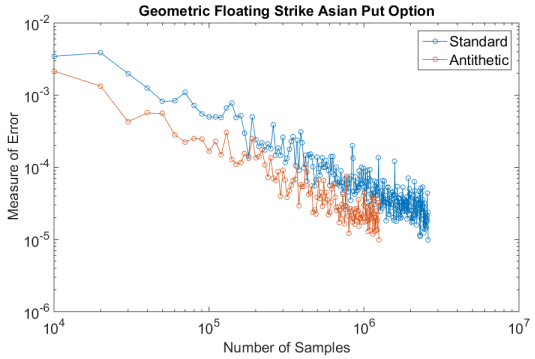

Figure 22: Convergence error measure for the geometric floating strike Asian put option

6.4 Code

6.4.1 Model Error Code

The following is the main script for the analysis:

```
% Data inputs
T = 1;
t = 0;
S0 = 100;
r = 0.05;
K = 100;
sigma = 0.2;

% Accuracy Inputs
dt = [1e-1 1e-2 1e-3 1e-4 1e-5];
numSamples = 1e2;
randseed = rng;
eps = 1e-3;

allError = zeros(2,length(dt));
allTime = zeros(1,length(dt));

% Loop through each timestep and calculate error
for i = 1:length(dt)
    disp(['dt = ' num2str(dt(i))]);
    tic;
    [currentValue,currentSamples,errArray,sampleArray] =...
        modelMonteCarlo(dt(i),t,T,r,sigma,S0,K,randseed,eps,numSamples);
    allTime(i) = toc;
    allError(1,i) = abs(currentValue(1)-currentValue(2));
    allError(2,i) = abs(currentValue(1)-currentValue(3));
    disp(currentSamples);
end

%% Plotting
figHandle = figure;
set(figHandle,'Position',[20,20,1000,600])

loglog(dt,allError(1,:),'-o',dt,allError(2,:),'-o');


title('Geometric Floating Strike Asian Put Option')
legend('Euler-Maruyama','Milstein')
xlabel('Size of timestep \delta t')
ylabel('Measure of Error')
hold on;

set(findall(figHandle,'-property','FontSize'),'FontSize',18)
```

The following is a function which runs the Monte-Carlo methods:

```
function [currentValue,currentSamples,errArray,sampleArray] =...
    modelMonteCarlo( dt, t, T, r, sigma, S0, K, randseed, eps ,numSamples)
    % Get the number of time periods needed
    numTimes = round((T-t)/dt) + 1;
    err = 100;
    rng(randseed);
    % Initialise arrays to hold information
    errArray1 = [];sampleArray1 = [];currentValue1 = 0;currentSamples1 = 0;
```

```
    errArray2 = [];sampleArray2 = [];currentValue2 = 0;currentSamples2 = 0;
    errArray3 = [];sampleArray3 = [];currentValue3 = 0;currentSamples3 = 0;
    while err > eps
        % normal random samples
        phi = randn(numSamples,numTimes-1);
        % Use different formula to get each path
        % Closed form
        phi1 = exp((r-0.5*sigma^2)*dt + sigma*phi*sqrt(dt));
        % Euler-Maruyama
        phi2 = 1 + r*dt + sigma*phi*sqrt(dt);
        % Milstein
        phi3 = 1 + (r+0.5*sigma^2*(phi.^2-1))*dt + sigma*phi*sqrt(dt);
        % Get paths for each formula
        S1 = S0*ones(numSamples,numTimes);
        S1(:,2:end) = S1(:,2:end).*cumprod(phi1,2);
        S2 = S0*ones(numSamples,numTimes);
        S2(:,2:end) = S2(:,2:end).*cumprod(phi2,2);
        S3 = S0*ones(numSamples,numTimes);
        S3(:,2:end) = S3(:,2:end).*cumprod(phi3,2);
        % Get values at the end of each path
        ST1 = S1(:,end);
        ST2 = S2(:,end);
        ST3 = S3(:,end);
        % Decide if arithmetic or geometric
%         S1 = mean(S1,2);
%         S2 = mean(S2,2);
%         S3 = mean(S3,2);
        S1 = exp(mean(log(S1),2));
        S2 = exp(mean(log(S2),2));
        S3 = exp(mean(log(S3),2));
        % Calculate the payoff and the error in the last 100 terms
        [err1,currentValue1,currentSamples1] = checkError(S1,ST1,K,r,T,t,...
            currentSamples1,currentValue1,numSamples);
        errArray1 = [errArray1 err1];
        sampleArray1 = [sampleArray1 currentSamples1];
        [err2,currentValue2,currentSamples2] = checkError(S2,ST2,K,r,T,t,...
            currentSamples2,currentValue2,numSamples);
        errArray2 = [errArray2 err2];
        sampleArray2 = [sampleArray2 currentSamples2];
        [err3,currentValue3,currentSamples3] = checkError(S3,ST3,K,r,T,t,...
            currentSamples3,currentValue3,numSamples);
        errArray3 = [errArray3 err3];
        sampleArray3 = [sampleArray3 currentSamples3];
        % Get the maximum error
        err = max([err1,err2,err3]);
    end
    % Store all the data
    currentSamples = [currentSamples1,currentSamples2,currentSamples3];
    currentValue = [currentValue1, currentValue2, currentValue3];
    errArray = [errArray1', errArray2', errArray3'];
    sampleArray = [sampleArray1', sampleArray2', sampleArray3'];

end
```

The following is a function which calculates the running average from the new samples and gets the convergence error:

```
function [err,currentValue,currentSamples] =...
    checkError(S,ST,K,r,T,t,currentSamples,currentValue,numSamples)
    % Check if this is the first calculation or not
    if currentSamples == 0
        arraySamples = (currentSamples+1:currentSamples+numSamples)';
        payoffValue = exp(-r*(T-t))*optionPayoff(S,ST,K);
    else
        arraySamples = (currentSamples:currentSamples+numSamples)';
        payoffValue = [currentSamples*currentValue;...
            exp(-r*(T-t))*optionPayoff(S,ST,K)];
    end
    % Calculate the cumulative mean
    meanValue = cumsum(payoffValue)./arraySamples;
    % Calculate the latest average
    currentValue = meanValue(end);
    currentSamples = currentSamples+numSamples;
    % Calculate the standard deviation of the last 1e3 averages
    err = std(meanValue(end-min(1e3,numSamples-1):end));
end
```

The following is a function which calculate the payoff at maturity:

```
function [ P ] = optionPayoff( S, ST, K )
    % Payoff of the fixed strike call option
    P = max(S-K,0);
    % Payoff of the fixed strike put option
%     P = max(K-S,0);
    % Payoff of the floating strike call option
%     P = max(ST-S,0);
    % Payoff of the floating strike put option
%     P = max(S-ST,0);
end
```

6.4.2 Rate of Convergence Code

The following is the main script for the analysis:

```
% Data inputs
T = 1;
t = 0;
S0 = 100;
r = 0.05;
K = 100;
sigma = 0.2;
% Accuracy Inputs
dt = 1e-3;
randseed = rng;
% Run the standard Monte-Carlo
tic;
[standardCurrentValue,standardCurrentSamples,...
    standardErrArray,standardSampleArray] =...
    standardModelMonteCarlo(dt,t,T,r,sigma,S0,K,randseed);
```

```
standardTime = toc;
% Run the antithetics Monte-Carlo
tic;
[antitheticCurrentValue,antitheticCurrentSamples,...
    antitheticErrArray,antitheticSampleArray] =...
    antitheticStandardModelMonteCarlo(dt,t,T,r,sigma,S0,K,randseed);
antitheticTime = toc;
%% Store Data
allData = zeros(4,2);
allData(:,1) = [standardCurrentValue;standardErrArray(end);...
    standardCurrentSamples;standardTime];
allData(:,2) = [antitheticCurrentValue;antitheticErrArray(end);...
    antitheticCurrentSamples;antitheticTime];
%% Plotting
figHandle = figure;
set(figHandle,'Position',[20,20,1000,600])
loglog(standardSampleArray,standardErrArray,'-o',antitheticSampleArray,...
    antitheticErrArray,'-o');
title('Arithmetic Floating Strike Asian Put Option')
legend('Standard','Antithetic')
xlabel('Number of Samples')
ylabel('Measure of Error')
hold on;
set(findall(figHandle,'-property','FontSize'),'FontSize',18)
```

The following is a function which runs the standard Monte-Carlo method:

```
function [currentValue,currentSamples,errArray,sampleArray] = ...
    standardModelMonteCarlo(dt,t,T,r,sigma,S0,K,randseed)
    % Get the number of time periods needed
    numTimes = (T-t)/dt + 1;
    % Check is its an integer
    if rem(numTimes,1) ~= 0
        error('dt does not divide (T-t)');
    end
    % Initialise variables
    numSamples = 1e4;
    currentValue = 0;
    currentSamples = 0;
    % Accurancy for the convergence
    eps = 1e-5;
    % Random seed
    rng(randseed);
    err = 100;
    errArray = [];
    sampleArray = [];
    % Run the while loop until convergence reached
    while err > eps
        % Get normal random samples
        phi = randn(numSamples,numTimes-1);
        % Get the Milstein method to calculate paths for the underlying
        phi = 1 + r*dt + phi*sqrt(dt)*sigma + 0.5*sigma^2*(phi.^2-1)*dt;
        % Calculate the paths of the underlying
        S = S0*ones(numSamples,numTimes);
        S(:,2:end) = S(:,2:end).*cumprod(phi,2);
```

```
        % Calculate the end of each path
        ST = S(:,end);
        % Arithmetic or geometric sampling
        S = mean(S,2);
%         S = exp(mean(log(S),2));
        % Put everything together and check error
        [err,currentValue,currentSamples] =...
            checkError(S,ST,K,r,T,t,currentSamples,currentValue,numSamples);
        errArray = [errArray err];
        sampleArray = [sampleArray currentSamples];
    end
end
```

The following is a function which runs the antithetic Monte-Carlo method:

```
function [currentValue,currentSamples,errArray,sampleArray] = ...
    antitheticStandardModelMonteCarlo(dt,t,T,r,sigma,S0,K,randseed)
    % Get the number of time periods needed
    numTimes = (T-t)/dt + 1;
    % Check is its an integer
    if rem(numTimes,1) ~= 0
        error('dt does not divide (T-t)');
    end
    % Initialise variables
    numSamples = 1e4;
    currentValue = 0;
    currentSamples = 0;
    % Accurancy for the convergence
    eps = 1e-5;
    % Random seed
    rng(randseed);
    err = 100;
    errArray = [];
    sampleArray = [];
    % Run the while loop until convergence reached
    while err > eps
        % Get normal random samples
        phi = randn(numSamples,numTimes-1);
        % Adjust for calculate paths for the underlying using Milstein for
        % both positive and negative phi
        phiPos = 1 + r*dt + phi*sqrt(dt)*sigma + 0.5*sigma^2*(phi.^2-1)*dt;
        phiNeg = 1 + r*dt - phi*sqrt(dt)*sigma + 0.5*sigma^2*(phi.^2-1)*dt;
        % Calculate the paths of the underlying
        SPos = S0*ones(numSamples,numTimes);
        SPos(:,2:end) = SPos(:,2:end).*cumprod(phiPos,2);
        SNeg = S0*ones(numSamples,numTimes);
        SNeg(:,2:end) = SNeg(:,2:end).*cumprod(phiNeg,2);
        % Calculate the end of each path
        ST = zeros(2*numSamples,1);
        ST(1:2:2*numSamples-1) = SPos(:,end);
        ST(2:2:2*numSamples) = SNeg(:,end);
        % Arithmetic or geometric sampling
        SPos = mean(SPos,2);
        SNeg = mean(SNeg,2);
%         SPos = exp(mean(log(SPos),2));
```

```
%         SNeg = exp(mean(log(SNeg),2));
        % Put everything together and check error
        S = zeros(2*numSamples,1);
        S(1:2:2*numSamples-1) = SPos;
        S(2:2:2*numSamples) = SNeg;
        [err,currentValue,currentSamples] = checkError(S,ST,K,r,T,t,...
            currentSamples,currentValue,2*numSamples);
        errArray = [errArray err];
        sampleArray = [sampleArray currentSamples];
    end
    % Half number of samples as antithetic
    sampleArray = sampleArray/2;
    currentSamples = currentSamples/2;

end
```

The following is a function which calculates the running average from the new samples and gets the convergence error:

```
function [err,currentValue,currentSamples] =...
    checkError(S,ST,K,r,T,t,currentSamples,currentValue,numSamples)
    % Check if this is the first calculation or not
    if currentSamples == 0
        arraySamples = (currentSamples+1:currentSamples+numSamples)';
        payoffValue = exp(-r*(T-t))*optionPayoff(S,ST,K);
    else
        arraySamples = (currentSamples:currentSamples+numSamples)';
        payoffValue = [currentSamples*currentValue;...
            exp(-r*(T-t))*optionPayoff(S,ST,K)];
    end
    % Calculate the cumulative mean
    meanValue = cumsum(payoffValue)./arraySamples;
    % Calculate the latest average
    currentValue = meanValue(end);
    currentSamples = currentSamples+numSamples;
    % Calculate the standard deviation of the last 1e3 averages
    err = std(meanValue(end-min(1e3,numSamples-1):end));
end
```

The following is a function which calculate the payoff at maturity:

```
function [ P ] = optionPayoff( S, ST, K )
    % Payoff of the fixed strike call option
    P = max(S-K,0);
    % Payoff of the fixed strike put option
%     P = max(K-S,0);
    % Payoff of the floating strike call option
%     P = max(ST-S,0);
    % Payoff of the floating strike put option
%     P = max(S-ST,0);
end
```

6.4.3 Size of Timestep δt Code

The following is the main script for the analysis:

```
% Data inputs
T = 1;
t = 0;
S0 = 100;
r = 0.05;
K = 100;
sigma = 0.2;
% Accuracy Inputs
dt = [1e-1 1e-2 1e-3 1e-4 1e-5];
numSamples = 1e2;
randseed = rng;
eps = 1e-4;
allError = zeros(1,length(dt));
allTime = zeros(1,length(dt));
currentValue = cell(1,length(dt));
currentSamples = cell(1,length(dt));
errArray = cell(1,length(dt));
sampleArray = cell(1,length(dt));
% Loop through each timestep and calculate error
for i = 1:length(dt)
    disp(['dt = ' num2str(dt(i))]);
    % Get the converged solutiona and compare it to the exact solution
    tic;
    [currentValue{i},currentSamples{i},errArray{i},sampleArray{i}] =...
        timestepMonteCarlo(dt(i),t,T,r,sigma,S0,K,randseed,eps,numSamples);
    allTime(i) = toc;
    allError(1,i) = abs(currentValue{i}-5.5468);
    disp(currentSamples);
end
%% Plotting
figHandle = figure;
set(figHandle,'Position',[20,20,1000,600])
loglog(dt,allError,'-o');
title('Geometric Fixed Strike Asian Call Option')
% legend('Euler-Maruyama','Milstein')
xlabel('Size of timestep \delta t')
ylabel('Measure of Error')
set(findall(figHandle,'-property','FontSize'),'FontSize',18)
```

The following is a function which runs the standard Monte-Carlo method:

```
function [currentValue,currentSamples,errArray,sampleArray] =...
    timestepMonteCarlo( dt, t, T, r, sigma, S0, K, randseed, eps ,numSamples)
    % Get the number of time periods needed
    numTimes = round((T-t)/dt) + 1;
    err = 100;
    rng(randseed);
    % Initialise arrays to hold information
    errArray = [];sampleArray = [];currentValue = 0;currentSamples = 0;
    while err > eps
```

```
        % normal random samples
        phi = randn(numSamples,numTimes-1);
        % Use different formula to get each path
        % Closed form
        phi1 = exp((r-0.5*sigma^2)*dt + sigma*phi*sqrt(dt));
        % Get paths for each formula
        S = S0*ones(numSamples,numTimes);
        S(:,2:end) = S(:,2:end).*cumprod(phi1,2);
        % Get values at the end of each path
        ST = S(:,end);
        % Decide if arithmetic or geometric
%         S1 = mean(S1,2);
        S = exp(mean(log(S),2));
        % Calculate the payoff and the error in the last 100 terms
        [err,currentValue,currentSamples] = checkError(S,ST,K,r,T,t,...
            currentSamples,currentValue,numSamples);
        errArray = [errArray err];
        sampleArray = [sampleArray currentSamples];
    end
end
```

The following is a function which calculates the running average from the new samples and gets the convergence error:

```
function [err,currentValue,currentSamples] =...
    checkError(S,ST,K,r,T,t,currentSamples,currentValue,numSamples)
    % Check if this is the first calculation or not
    if currentSamples == 0
        arraySamples = (currentSamples+1:currentSamples+numSamples)';
        payoffValue = exp(-r*(T-t))*optionPayoff(S,ST,K);
    else
        arraySamples = (currentSamples:currentSamples+numSamples)';
        payoffValue = [currentSamples*currentValue;...
            exp(-r*(T-t))*optionPayoff(S,ST,K)];
    end
    % Calculate the cumulative mean
    meanValue = cumsum(payoffValue)./arraySamples;
    % Calculate the latest average
    currentValue = meanValue(end);
    currentSamples = currentSamples+numSamples;
    % Calculate the standard deviation of the last 1e3 averages
    err = std(meanValue(end-min(1e3,numSamples-1):end));
end
```

The following is a function which calculate the payoff at maturity:

```
function [ P ] = optionPayoff( S, ST, K )
    % Payoff of the fixed strike call option
    P = max(S-K,0);
    % Payoff of the fixed strike put option
%     P = max(K-S,0);
    % Payoff of the floating strike call option
%     P = max(ST-S,0);
    % Payoff of the floating strike put option
```

```
%      P = max(S-ST,0);
end
```

6.4.4 Multi Level Monte-Carlo

The following is the main script for the analysis:

```
% Data inputs
T = 1;
t = 0;
S0 = 100;
r = 0.05;
K = 100;
sigma = 0.2;
% Accuracy Inputs
randseed = rng;
eps = 1e-2;
M = 4;
% Run Multi Level Monte Carlo
tic;
[multiCurrentValue,multiCurrentSamples,multiLevel,allValues] = ...
    multiLevelModelMonteCarlo(t,T,r,sigma,S0,K,eps,M,randseed );
multiTime = toc;
%% Store Data
allData = [multiCurrentValue;multiLevel;multiCurrentSamples;multiTime];
%% Plotting
figHandle = figure;
set(figHandle,'Position',[20,20,1000,600])
plot(0:multiLevel-1,allValues,'-o',0:multiLevel-1,...
    5.5468*ones(1,multiLevel),'-o');
title('Geometric Fixed Strike Asian Call Option')
legend('Multi Level Monte-Carlo','Exact','Location','southeast')
xlabel('Level')
ylabel('Value of Payoff')
set(findall(figHandle,'-property','FontSize'),'FontSize',18)
```

The following is a function which runs the Multi Level Monte-Carlo method:

```
function [YL,currentSamples,L,allValues] =...
    multiLevelModelMonteCarlo(t,T,r,sigma,S0,K,eps,M,randseed)
    % Set random number generator seed
    rng(randseed)
    terminateFlag = 1;
    initN = 1e4;
    allValues = [];
    % Do the zero case
    % Get the estimated mean and variance
    [firstYl,firstVl] = getFirstEstimate(initN,t,T,r,sigma,S0,K);
    % Get the optimal NL
    NL = max(ceil(2*eps^(-2)*firstVl),initN);
    % Finish the iteration from getting the remaining samples
    [Yl,Vl] = getFirstEstimate(NL,t,T,r,sigma,S0,K);
    % Update running mean
```

```
    YL = (initN*firstYl+NL*Yl)/(initN+NL);
    allValues = [allValues YL];
    % Update variance/time step sum
    sumVL = sqrt(Vl);
    currentSamples = NL + initN;
    L = 1;
    while terminateFlag
        % Set the time step size
        hl = M^(-L)*(T-t);
        % Get the estimated mean and variance
        [firstYl,firstVl] = getEstimate(initN,M,L,t,T,r,sigma,S0,K);
        % Get the optimal NL (12)
        NL = max(ceil(2*eps^(-2)*sqrt(firstVl*hl)*...
            (sumVL+sqrt(firstVl/hl))),1);
        % Finish the iteration from getting the remaining samples
        [Yl,Vl] = getEstimate(NL,M,L,t,T,r,sigma,S0,K);
        % Update running mean
        oldYL = YL;
        YL = YL + (initN*firstYl+NL*Yl)/(initN+NL);
        allValues = [allValues YL];
        % Update variance/time step sum
        sumVL = sumVL + sqrt(Vl/hl);
        % Check if converged
        if abs(YL - oldYL/M) < (M^2-2)*eps/sqrt(2) || L == 5
            terminateFlag = 0;
        end
        currentSamples = currentSamples + NL + initN;
        L = L+1;
    end

end
```

The following are functions to calculate the averages of the payoffs and the variances:

```
function [YL,VL] = getFirstEstimate(numSamples,t,T,r,sigma,S0,K)
    % Generate random samples
    phi = randn(numSamples,1);
    % Get the (1 + r*dt + sigma*phi*sqrt(dt))
    phi = exp((r-0.5*sigma^2)*(T-t) + phi*sqrt(T-t)*sigma);
    % Initialise S
    S = S0*ones(numSamples,2);
    S(:,end) = S(:,end).*phi;
    ST = S(:,end);
%     S = mean(S,2);
    S = exp(mean(log(S),2));
    % Get the payoff then find the mean and variance
    Y = exp(-r*(T-t))*optionPayoff(S,ST,K);
    YL = mean(Y);
    VL = var(Y);
end
```

```
function [YL,VL] = getEstimate(numSamples,M,L,t,T,r,sigma,S0,K)
    % Initialise numTimes and generate random samples
```

```
    numTimes = M^L*(T-t) + 1;      % CHECK FOR T-t!!!
    hl = M^(-L)*(T-t);
    phi2 = randn(numSamples,numTimes-1);
    phi1 = reshape(sum(reshape(phi2',M,[])),[],numSamples)';
    % Get the (1 + r*dt + sigma*phi*sqrt(dt))
    phi2 = exp((r-0.5*sigma^2)*hl + phi2*sqrt(hl)*sigma);
    phi1 = exp((r-0.5*sigma^2)*M*hl + phi1*sqrt(hl)*sigma);
    % Initialise S
    S2 = S0*ones(numSamples,numTimes);
    S2(:,2:end) = S2(:,2:end).*cumprod(phi2,2);
    ST2 = S2(:,end);
    S1 = S0*ones(numSamples,size(phi1,2)+1);
    S1(:,2:end) = S1(:,2:end).*cumprod(phi1,2);
    ST1 = S1(:,end);
    % Get the average over times for the two levels
%     S2 = mean(S2,2);
%     S1 = mean(S1,2);
    S2 = exp(mean(log(S2),2));
    S1 = exp(mean(log(S1),2));

    % Get the payoff then find the mean and variance
    Y = exp(-r*(T-t))*(optionPayoff(S2,ST2,K) - optionPayoff(S1,ST1,K));

    YL = mean(Y);
    VL = var(Y);
end
```

The following is a function which calculates the running average from the new samples and gets the convergence error:

```
function [err,currentValue,currentSamples] =...
    checkError(S,ST,K,r,T,t,currentSamples,currentValue,numSamples)
    % Check if this is the first calculation or not
    if currentSamples == 0
        arraySamples = (currentSamples+1:currentSamples+numSamples)';
        payoffValue = exp(-r*(T-t))*optionPayoff(S,ST,K);
    else
        arraySamples = (currentSamples:currentSamples+numSamples)';
        payoffValue = [currentSamples*currentValue;...
            exp(-r*(T-t))*optionPayoff(S,ST,K)];
    end
    % Calculate the cumulative mean
    meanValue = cumsum(payoffValue)./arraySamples;
    % Calculate the latest average
    currentValue = meanValue(end);
    currentSamples = currentSamples+numSamples;
    % Calculate the standard deviation of the last 1e3 averages
    err = std(meanValue(end-min(1e3,numSamples-1):end));
end
```

The following is a function which calculate the payoff at maturity:

```
function [ P ] = optionPayoff( S, ST, K )
    % Payoff of the fixed strike call option
```

```
3       P = max(S-K,0);
4       % Payoff of the fixed strike put option
5   %     P = max(K-S,0);
6       % Payoff of the floating strike call option
7   %     P = max(ST-S,0);
8       % Payoff of the floating strike put option
9   %     P = max(S-ST,0);
10  end
```

References

[1] Kyriaki Stavri, *Theoretical and Numerical Schemes for Pricing Exotics*, UCL, 2004.

[2] Michael B. Giles, *Multilevel Monte Carlo Path Simulation*, 2008, `https://people.maths.ox.ac.uk/gilesm/files/opre.pdf`.