Project 1: Vanilla Options in a Black–Scholes World

Thomas Johnson

May 2, 2022

1 Introduction

The purpose of this project is to implement the pricing of some vanilla options in a Black–Scholes model using multiple methods. Some functions implemented here will be very useful for later projects.

All the code for this project can be found in the relevant folders in the github repository.

2 The Black–Scholes model

Let us begin by reviewing the Black–Scholes model. Recall that in this market model, the market consists of at least two assets, one of them a risky asset (typically called the stock) and the other a riskless asset (called the money market account or riskless bond). Moreover the price process of a unit of the stock S is assumed to follow geometric Brownian motion,

$$dS_t = \mu_t S_t dt + \sigma_t S_t dW_t$$

where μ_t, σ_t are given deterministic functions and W_t is a standard Brownian motion relative to some "real world" probability measure $\mathcal{P}$, whilst the price process of the riskless bond B grows at a given risk free rate r_t:

$$dB_t = r_t B_t.$$

Further assumptions on the market are as follows:

1. We can borrow/lend any amount of cash at the riskless rate, and we can do so:
   - at any time
   - instantaneously
   - with 0 transaction costs
2. Similarly, we can go long/short any amount of stock, and we can do so:
   - at any time
   - instantaneously
   - with 0 transaction costs
3. The market is free of arbitrage.

To explain the last point, we recall the notion of an arbitrage portfolio:

Definition 2.1. *A portfolio P is said to be an arbitrage portfolio relative to $\mathcal{P}$ if:*

1. *It has non-positive value today.*
2. *There is zero probability (with respect to $\mathcal{P}$) of it having negative value in the future.*

The existence of arbitrage portfolios thus implies the possibility of a risk-free profit, and to say the market is free of arbitrage is to say that such portfolios do not exist.

Now it turns out that these assumptions together imply that a vanilla option O on the stock which pays $f(S_T)$ at some time T necessarily has a unique, arbitrage free price. There are two approaches to deriving said price; replication and risk-neutral pricing.

2.1 Pricing by replication

Pricing by replication utilises the fact that in an arbitrage free market, a portfolio of assets whose price process matches that of the option at the terminal time must be equal for all times.

Consider a portfolio P which at any time t consists of $\alpha(S_t, t)$ units of stock and $\beta(S_t, t)$ units of riskless bonds, where α, β are functions that are sufficiently regular for the following arguments to apply. Assume moreover that P satisfies the following:

1. P is self-financing (that is, any changes in the value of P are purely due to changes in holdings of the stock and bonds. In particular, there is no outside injection of stock or cash.
2. P at time T has value $f(S_T)$.

Then the fact that the market is free of arbitrage implies that P has the same value as O at *any* time t. Indeed, if not then one of

$$P - O, O - P$$

defines an arbitrage portfolio by the assumptions on P and the market. On the other hand, we claim that the assumptions on P actually uniquely specifies a value for the price processes V_t for P at any time t[1].

Indeed, let us first observe that we must have[2]

$$V(S,t) = \alpha(S,t)S + \beta(S,t)B_t.$$

The self-financing condition then implies

$$\begin{aligned} dV &= \alpha dS + \beta dB \\ &= (\alpha\mu S + \beta B)dt + \alpha\sigma S dW \end{aligned}$$

On the other hand by Ito's lemma

$$dV = (\partial_t V + \frac{1}{2}\sigma^2 S^2 \partial_S^2 V)dt + \sigma S \partial_S V dW.$$

[1]Technically we also need to assume that we can write $V = V(S,t), C = C(S,t)$ where V, C are sufficiently regular for Ito's lemma to apply).

[2]We assume here that the stock S pays no dividends – see section 3 where we remove this assumption.

Equating and then taking expecations and variances implies

$$\begin{aligned}\alpha &= \partial_S V,\\ \beta &= \frac{1}{rB}\Big(\partial_t V + \frac{1}{2}\sigma^2 S^2 \partial_S^2 V\Big).\end{aligned}$$

We therefore conclude that $V(S,t)$ satisfies the boundary value problem

$$r_t V(S,t) = \partial_t V(S,t) + r_t S \partial_S V(S,t) + \frac{1}{2}\sigma_t^2 S^2 \partial_S^2 V(S,t), \quad t \in [0,T], \tag{1}$$

$$V(S,T) = f(S_T). \tag{2}$$

Equation (1) is the *Black–Scholes equation.* Under mild assumptions at infinity, the above boundary value problem can be solved uniquely. Indeed we have (cf. the appendix).

Theorem 2.1. *Under mild conditions at infinity, the unique solution to the above terminal boundary value problem is given by*

$$C(S,t) = e^{-\overline{r}(t,T)(T-t)} E\Big[f\Big(S\exp\Big((\overline{r}(t,T) - \frac{1}{2}\sigma_{eff}^2(t,T))(T-t) + \sigma_{eff}(t,T)\sqrt{T-t}N(0,1)\Big)\Big)\Big]$$

where $\overline{r}$ and σ_{eff} are the average short rate and effective[3] *volatility over the interval $[t,T]$:*

$$\overline{r}(t_1,t_2) := \frac{1}{t_2 - t_1}\int_{t_1}^{t_2} r_t dt, \tag{3}$$

$$\sigma_{eff}(t_1,t_2) := \sqrt{\frac{1}{t_2 - t_1}\int_{t_1}^{t_2} \sigma_t^2 dt}. \tag{4}$$

It therefore follows from the above that if such a portfolio P as defined prior exists then the assumption of no arbitrage enforces the price of the option O at time t to be given by $C(S_t,t)$. On the other hand, defining P by

$$\begin{aligned}\alpha(S,t) &= \partial_S C(S,t),\\ \beta(S,t) &= \frac{1}{r_t B_t}\Big(\partial_t C(S,t) + \frac{1}{2}\sigma^2 S^2 \partial_S^2 C(S,t)\Big)\end{aligned}$$

we see that P is the desired replicating portfolio. We therefore conclude:

Theorem 2.2. *In a Black–Scholes world, the price of a vanilla option O on S with payoff function f is given at all times by*

$$C(S,t) = e^{-\overline{r}(t,T)(T-t)} E\Big[f\Big(S\exp\Big((\overline{r}(t,T) - \frac{1}{2}\sigma_{eff}^2(t,T))(T-t) + \sigma_{eff}(t,T)\sqrt{T-t}N(0,1)\Big)\Big)\Big].$$

2.2 Risk-neutral pricing

Risk-neutral pricing is based on the idea that if the price process of is a martingale, then it must be arbitrage free.

We begin by observing the following for a general market measure $\mathcal{P}$:

[3]It is so called because it is the expression one must insert in the pricing formula to get the same expression as in the constant vol case.

1. Suppose $\mathcal{Q}$ is an equivalent[4] measure to $\mathcal{P}$. Then $\mathcal{Q}$ defines an arbitrage free measure on the market iff $\mathcal{P}$ defines an arbitrage free measure on the market.
2. If N_t denotes the price process of any tradeable asset in the market and for the price process A_t of any asset we have that A_t/N_t is a martingale wrt to a market measure $\mathcal{Q}_N$ then the market with measure $\mathcal{Q}_N$ is necessarily arbitrage free[5].

It then follows that if there exists an equivalent measure Q_N to the real world measure $\mathcal{P}$ for which 2. holds then necessarily $\mathcal{P}$ is arbitrage free and an arbitrage free price of a vanilla option with payoff f on a market asset S is given by

$$C(S,t) = N_t E_{Q_N}(f(S_T)/N_T|\mathcal{F}_t).$$

It can also be shown that the price is invariant under changes in the numeraire N_t[6]. In particular if for a fixed numeraire the measure Q_N is unique then the above gives the unique abritrage free price of the option.

Let us return now to the Black–Scholes setting. We recall the Girsanov theorem:

Theorem 2.3. *(rough version) Let $\mathcal{P}$ be a probability measure and let W_t be a Brownian motion relative to $\mathcal{P}$. Then for any sufficiently regular function γ there exists a unique equivalent measure $\mathcal{Q}_\gamma$ to $\mathcal{P}$ for which*

$$W'_t = W_t - \int_0^t \gamma_s ds$$

is a Brownian motion relative to $\mathcal{Q}_\gamma$.

Recalling further that, up to some technical assumptions, an Ito process is a martingale iff it has vanishing drift we infer the following:

Proposition 2.1. *In a Black–Scholes world there is a unique equivalent measure $\mathcal{Q}_B$ in which S_t/B_t is a martingale.*

Proof. Let W_t be the Brownian motion defined relative to $\mathcal{P}$ that drives the movement of S. Then by Girsanov there exists a unique equivalent measure $\mathcal{Q}_B$ for which

$$W'_t := W_t - \int_0^t \frac{\mu_s - r_s}{\sigma_s} ds$$

is a Brownian motion relative to $\mathcal{Q}_B$. On the other hand

$$dS_t = r_t S_t dt + \sigma_t S_t dW'_t$$

[4]Recall this means that both measures have the same measure 0 sets.

[5]Indeed, let P be any portfolio which has non-positive value today. Now by assumption P/N must be a martingale, hence for any time t we have $E[P_t/N_t] = P_0/N_0 \leq 0$. There must therefore be a non-zero probability that P_t has non-positive value in the future.

[6]Here's the idea. Suppose we price with respect to some numeraire N_1 and want to change to N_2. Now by assumption $(N_2(T)/N_1(T))/(N_2(t)/N_1(t))$ is a martingale, hence we can define a change of measure $dP_{N_2} = (N_2(T)/N_1(T))/(N_2(t)/N_1(t))dP_{N_1}$ and in this measure

$$\begin{aligned} E_{N_2}[X \cdot N_2(t)/N_2(T)] &= E_{N_1}[X \cdot N_2(t)/N_2(T)N_2(T)(N_2(T)/N_1(T))/(N_2(t)/N_1(t))] \\ &= E_{N_1}[X \cdot N_1(t)/N_1(T)]. \end{aligned}$$

which yields

$$d(S_t/B_t) = \sigma_t(S_t/B_t)dW'_t$$

so that S_t/B_t is a martingale relative to $\mathcal{Q}_B$. $\square$

We conclude that in a Black–Scholes world the unique arbitrage free price of the option is given by

$$C(S,t) = B_t E_{Q_B}(f(S_T)/B_T|S_t)$$

where

$$dS_t = r_t S_t dt + \sigma_t S_t dW'_t$$

with W'_t a Brownian motion relative to $\mathcal{Q}_B$. Moreover, since $\mathcal{P}$ was assumed to be free of arbitrage, this is the only price that the option can have.

Of course, solving the above SDE one finds the same pricing formula as before.

We call the above measure a *risk-neutral* measure for it predicts that the stock, a supposed risky asset, grows at the risk-free rate. One could also take S itself as numeraire[7], in which case the dynamics of S in the new measure are determined by B/S being a martingale in this measure. This gives

$$C(S,t) = S_t E_{Q_S}(f(S_T)/S_T|S_t)$$

where

$$dS_t = (r_t - \frac{1}{2}\sigma_t^2)S_t dt + \sigma_t S_t dW'_t.$$

This gives:

Theorem 2.4. *In a Black–Scholes world, the price of a vanilla option O on S with payoff function f can be expressed as*

$$C(S,t) = S_t E\Big[f(S_T)/S_T\Big]$$

where

$$S_T = S_t \exp\Big((\overline{r}(t,T) - \sigma_{eff}^2(t,T))(T-t) + \sigma_{eff}(t,T)\sqrt{T-t}N(0,1).$$

The expression one chooses ultimately depends upon which gives a simpler integrand in the expectation.

2.3 Dividend paying stocks

Many stocks pay dividends. In analogy with foreign exchange, a typical assumption is that such dividends are paid as units of stock at a continuous rate d_t. Thus a portfolio P that consists of one unit of stock at time T will consist of $e^{-\overline{d}(t,T)(T-t)}$ units of stock at time $t < T$. In particular, recalling that S_t is the value of one unit of stock at time t, we have that the price-process of P satisfies

$$X_t = e^{-\overline{d}(t,T)(T-t)}S_t.$$

This means that:

[7] Provided that S pays no dividends.

1. When applying our hedging argument we need to replace S_t with the stochastic process X_t verifying
$$dX_t = (\mu_t + d_t)X_t dt + \sigma_t X_t dW.$$
2. When applying risk-neutral pricing we need to find a measure which ensures X_t/B_t is a martingale as S_t is not the price process of any *tradeable* asset. The associated risk-neutral dynamics of S are then given by
$$dS_t = (r_t - d_t)S_t dt + \sigma_t S_t dW.$$

In either case, one arrives at the following pricing formula in the case of non-zero dividends.

Theorem 2.5. *In a Black–Scholes world, the price of a vanilla option O on a divided paying asset S with payoff function f is given at all times by*
$$C(S,t) = e^{-\overline{r}(t,T)(T-t)} E\Big[f\Big(S\exp\Big((\overline{r}-\overline{d})(t,T) - \frac{1}{2}\sigma_{eff}^2(t,T))(T-t) + \sigma_{eff}(t,T)\sqrt{T-t}N(0,1)\Big)\Big)\Big]$$
with this expression equivalent to
$$C(S,t) = S_t E\Big[f(S_T)/S_T\Big]$$
where
$$S_T = S_t \exp\Big((\overline{r}(t,T) - \overline{d}(t,T) - \sigma_{eff}^2(t,T))(T-t) + \sigma_{eff}(t,T)\sqrt{T-t}N(0,1)\Big).$$

Note in particular that the prefactor outside the expectation is *unchanged* since this comes from discounting and not the drift term.

3 Pricing formulae

In this section we derive explicit solution formulae for various simple options and also discuss their implementation as functions in C++. The subsequent code can be found in the header file "BlackScholesSolutionFormulae.h" in the project folder "Project 1. Vanilla options in a BS world".

3.1 Deriving the formulae

In the sequel we use the following easy to prove identity: For any α, y
$$\int_y^\infty e^{\alpha x} N'(x)dx = e^{\frac{\alpha^2}{2}} N(\alpha - y).$$
In particular
$$E[e^{\alpha N(0,1)}] = e^{\frac{\alpha^2}{2}}.$$
Here N is the cdf of an $N(0,1)$ variable. In addition the above makes use of the identity
$$1 - N(x) = N(-x).$$

1. **Forward.** Recall a forward contract is the obligation to purchase an asset S for a price K at time T. Its payoff function is therefore

$$f(S_T) = S_T - K.$$

Plugging this into the solution formula we then find

$$C(S,t) = e^{-\overline{d}(t,T)(T-t)}S_t - Ke^{-\overline{r}(t,T)(T-t)}.$$

Note this can easily be derived in a model independent fashion by taking into account dividend payments and the evolution equation for the riskless bond.

2. **Call.** Recall a call gives the holder the option to purchase an asset S at price K at time T. It therefore has payoff function

$$f(S_T) = (S_T - K)_+.$$

Plugging this into the solution formula then yields

$$C(S,t) = e^{-\overline{r}(t,T)(T-t)}\int_{-d_2}^{\infty}\left(S_t e^{[(\overline{r}-\overline{d}-\frac{1}{2}\sigma_{eff}^2)(t,T))(T-t)+\sigma_{eff}(t,T)\sqrt{T-t}x]} - K\right)N'(x)dx$$

where

$$d_2 := \frac{\log(S/K) + (\overline{r} - \overline{d} - \frac{1}{2}\sigma_{eff}^2)(t,T)(T-t)}{\sigma_{eff}(t,T)\sqrt{T-t}}.$$

From this one concludes

$$C(S,t) = e^{-\overline{d}(t,T)(T-t)}S_t N(d_1) - Ke^{-\overline{r}(t,T)(T-t)}N(d_2)$$

where

$$d_1 := \frac{\log(S/K) + (\overline{r} - \overline{d} + \frac{1}{2}\sigma_{eff}^2)(t,T)(T-t)}{\sigma_{eff}(t,T)\sqrt{T-t}}.$$

Note in particular that $d_2 = d_1 - \sigma_{eff}(t,T)\sqrt{T-t}$.

3. **Put.** Recall a put gives the holder the option to sell an asset S at price K at time T. It therefore has payoff function

$$f(S_T) = (K - S_T)_+.$$

Now

$$f(S_T) = -(S_T - K) + (S_T - K)_+.$$

Since the solution formula is linear in f we therefore conclude

$$C(S,t) = C_{Call}(S,t) - C_{Forward}(S,t).$$

This is known as *put-call parity.* It could similarly be derived by appealing to no-arbitrage arguments (although this is implicit in the solution formula being linear in the payoff).

4. **Digital Call.** Recall a digital call of strike K gives the holder one unit of cash if at time T we have $S_T > K$, and nothing otherwise. It therefore has payoff function
$$f(S_T) = I_{S_T > K}.$$
Plugging this into the solution formula then yields
$$C(S,t) = e^{-\overline{r}(t,T)(T-t)} N(d_2).$$

5. **Digital Put.** Recall a digital put of strike K gives the holder one unit of cash if at time T we have $S_T < K$, and nothing otherwise. It therefore has payoff function
$$f(S_T) = I_{S_T < K}.$$
Now we can write
$$f(S_T) = 1 - I_{S_T > K}$$
so we therefore conclude
$$C(S_T) = e^{-\overline{r}(t,T)(T-t)} - C_{DigitalCall}(S,T).$$

3.2 Implementation in C++: the Parameters class

The only non-trivial thing about the implementation of the above is how to allow for variable parameters[8].

Let us first observe that all that enters the solution formula are the averages or root mean squares. The best solution is therefore to introduce an abstract base class "Parameters" which has methods for computing the average and the root mean square. The exact implementation of these methods will then be left to concrete inherited classes (for instance, one class would be one to represent constant parameters).

We therefore design a class "Parameters" which has pure virtual methods "Integral(x, y)" and "IntegralSquare(x,y)". Our Black–Scholes functions will then take in Parameters class object as parameters to represent the volatility and short and dividend rates. Since "Parameters" is an abstract base class, this will have to be done by reference to prevent slicing. We will also need to make the destructor virtual in case any object of an inherited class is destroyed via a pointer to its base. Note that we design methods for computing just the integrals and not the weighted integrals appearing in the solution formulae because although these weights are cancelled out in the exact formulas, when implemented on a computer it could lead to rounding errors.

We give also an example of a concrete inherited class, namely the "ConstantParameter" class. This takes in a double in its constructor which it stores along with its square as a data member. The pure virtual methods are then implemented in the obvious fashion.

The implementation of these classes can be found in the header file "Parameter.h" of the project folder.

Remark 3.1. *Note that the use of the keyword override is like synatic sugar in that all it does is return a compiler error if one is trying to override a function which wasn't declared virtual in a class higher up in the hierarchy. It is nevertheless good coding practice however to include it!*

[8]Actually this isn't strictly true as we also need to worry about how to implement the cdf of a Normal variable. However for this we can use the erfc function which is implemented in ¡cmath¿ and defined via
$$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = 2N(-\sqrt{2}x).$$
We can therefore just use this function after defining a const variable which represents the value of $1/\sqrt{2} = 0.707106781186547524401...$

4 Consistency checks

In this section we run various consistency checks so as to verify that we have implemented the formulas correctly.

4.1 Comparing functions and verifying monotonicity: lambda expressions

All of the checks involve either bounding one function by another or verifying that a function is monotonic. Since this is not something we will use again/do not want to spend too much time writing elegant code the solution to this is to utilise *lambda expressions* from C+11.

4.1.1 lambda expressions

In C++11 and later, a lambda expression—often called a lambda—is a convenient way of defining an anonymous function object right at the location where it's invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous functions and not used elsewhere. In particular they can be used instead of littering the code with loads of little helper functions.

The syntax is the following:

$$[CaptureClause](Parameterlist)\{lambdabody\};$$

Here the capture clause consists of those variables which the lambda is to capture from outside its parameter list or scope (so its a little like data members for a class object), the Parameter list is just a set of parameters as for a normal function whilst the lambda body is just a normal function body.

Lambda expressions are treated as function objects (defined by the type of the parameters and the return type of the function body) and can go anywhere the compiler expects a function object of that type. Here is a standard example of a Lambda expression:

```
    auto max [ ] (auto x, auto y) {
return max(x, y) };
```

The lambda expression max will then return the max of two input variables, provided such a comparison makes sense between the two types[9].

4.1.2 Monotonicity

A few of our checks involve verifying that the BS functions are monotonic in certain variables. The most obvious way to do this is to represent the functions in question as functors (by freezing the other parameters) and then write a little routine that checks whether a 1D function is increasing (note that a function f is increasing/decreasing iff $-f$ is decreasing/increasing) by passing its functor representation as a parameter. But the representation of the BS functions as functors can easily be done in a fairly resuable way as a lamdba expression by simply passing in the frozen paremetrs to the capture clause. Note prior to C+11 the only way to do this (as far as I'm aware) would be to define classes for each of the possible 1D functions, which is a headache[10]!

[9]Note this is very similar to a template, although one key difference is that for a template one can specify that x and y have the same type.

[10]In our case, this also leads to the business of having data members of types an abstract base class which leads to the issue of cloning and smart pointers etc.!

Since the function that is to do the checking will also not be used again, we shall also represent this as a lambda expression which takes in a function object. The code, along with the BS functions, can be found in the .cpp file "Consistency Checks" in the project folder (although we exclude it from the project solution as we won't need it again).

4.1.3 Function bounds

Part of the checks also involve checking that one function bounds another. This can be implemented in a similar way as above so we do not detail here.

4.2 The checks

We now go through the checks.

1. The price of a call should be monotone decreasing with strike.

   This is intuitively clear as the larger the strike the less likely the option will finish in the money. It can be verified by differentiating the solution formula/implementing the code as above.

   Note in checking we make use of the following "reasonable" set of parameters to fix all the other variables in the BS formula:

   $$d = 0, T = 1, r = 0.05, S = 100, K = 100, vol = 0.2.$$

2. The bounds $S - Ke^{-rT} < Call(S, t) < S_t$.

   These bounds follow from no arbitrage arguments (hence are model independent). The upper bound is obvious. For the lower bound we observe at time T we have

   $$(S_T - K)_+ + K \geq \max(S_T, K) \geq S_T$$

   whence the result.

   We verify this is C++ using our bounds function.

   Note that in the case of zero dividend and short rate, the above says that it is never optimal to early exercise an American call. Indeed, the above bound proves that n this case a European call is always worth more than its intrinsic value, hence necessarily it is the same for the American call.

3. A call should be monotone increasing in vol.

   This is intuitively clear as a call has limited downside but unlimited upside, and increasing the vol makes it more likely the spot will rise to a huge value.

   It can be verified by differentiating the formula, and also by running our code.

4. If $d = 0$ then the call increasing with t-t-m.

   Again, this is intuitively clear from the limited downside, unlimited upside argument (as the stock price how longer to grow).

   It can be proved by differentiating, and we confirm it with our code.

5. A call should be a convex[11] function of strike.

   This follows from the fact that the payoff is convex in K. Indeed for $K_1 \leq K_2$

$$(S_T - K_1 - t(K_2 - K_1))_+ \leq (S_T - K_1)_+ + t((S_T - K_2)_+ - (S_T - K_1)_+)$$

   and the left hand side is the payoff of a call option struck at $K_1 + t(K_2 - K_1)$ whilst the right hand side is the sum of the pay offs of $(1-t)$ call options stuck at K_1 and t call options struck at K_2. This is therefore a model independent bound.

   We easily verify it in our code.

6. A call spread approximates the price of a digital call.

   Recall a call spread is defined as follows. For any $\epsilon > 0$

$$CallSpread_\epsilon(K) = -\frac{Call(K+\epsilon) - Call(K-\epsilon)}{2\epsilon}.$$

   It is easily seen that this has payoff the same as a digital call struck at K except for $S \in (K-\epsilon, K+\epsilon)$ and so in the limit we get arbitrarily close.

   This can be easily verified in code by first defining a BS call spread function and then bumping the ϵ parameter.

   Incidentally, this also show

$$DigitalCall(K) = -\partial_K Call(K).$$

5 Validation via Monte Carlo

We now move on to verifying our formulae using a Monte Carlo pricer. In fact, we will want to design said pricer so as to give pricers for any vanilla option.

5.1 The theory behind Monte Carlo pricing

Let us begin by reviewing the theory of using Monte Carlo simulations to evaluate integrals.

We recall the central limit theorem.

Theorem 5.1. *Let $X, X_1, ..., X_N$ be i.i.d. and define*[12]

$$Z_N := \frac{\frac{1}{N}\sum_1^N X_i - \mu}{\sigma/\sqrt{N}}.$$

where $\mu := E[X]$ and $\sigma^2 = Var[X]$. Then

$$Z_N \to N(0,1)$$

in the sense of distributions.

[11] Recall f is convex on the interval $[a,b]$ if for every $t \in [0,1]$ we have $f(a + t(b-a)) \leq f(a) + t(f(b) - f(a))$. In particular the line through $(a, f(a))$ and $(b, f(b))$ lies above the graph of f.

[12] Observe that $E[Z_N] = 0, Var[Z_N] = 1$.

As a simple corollary we then have the law of large numbers

$$E[X] = \frac{1}{N}\sum_{1}^{N} X_i + \frac{\sigma}{\sqrt{N}}N(0,1) + o(N^{-1/2})$$

which gives us a numerical method for evaluating expectations[13] – make N "draws"[14] from X, take the average. The resulting sum will then converge to $E[X]$ at a rate of order $N^{-1/2}$ with leading order coefficent proporional to the standard deviation of X.

Of course, in practice σ will be unknown e.g. when pricing Vanillas we do not have a closed expression for the variance of $f(S_T)$. We can however approximate σ using the *sample variance*:

$$\hat{\sigma}_N^2 := \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \overline{X}_N)^2, \qquad X_N = \frac{1}{N}\sum_{i=1}^{N} X_i.$$

Indeed by the CLT $N^{-1}\sum_{i=1}^{N}(X_i-\mu)^2$ is an estimate for σ^2 whilst $\overline{X}_N$ is an estimator for μ. Note in particular the reciprocal of $N-1$ instead of N. This means that our estimator is *unbiased*:

$$E[\hat{\sigma}_N^2] = \sigma^2.$$

In the limit we can therefore write

$$E[X] = \hat{\mu}_N + \frac{\hat{\sigma}_N}{\sqrt{N}}N(0,1), \qquad \hat{\mu}_N := \overline{X}_N.$$

The second term on the right is known as the *standard error*. We can use it to compute confidence intervals for our estimate of the expectation

$$P(|E[X] - \hat{\mu}_N| \leq \alpha) = 2N(\alpha\sqrt{N}/\hat{\sigma}_N) - 1.$$

In particular if we want to be 99% sure that our estimate is within 2 decimal places of the true expecation then we need to choose N so that

$$\sqrt{N} \geq 10^3\hat{\sigma}_N N^{-1}(0.5 * 1.99).$$

Interestingly $N^{-1}(0.5*1.99) \approx 6.63$ so the number of paths we need to take to achieve this is heavily determined by the variance. In particular if the variance is of size 1 then we can make do with around 10 million paths, which a computer can simulate relatively quickly. It is in fact possible in some cases to reduce the variance by implementing certain control variates.

Of course at first glance the $N^{-1/2}$ convergence is still rather slow; for instance in 1 dimensions the Trapezium rule gives an error of N^{-2}. However in higher dimensions this becomes $N^{-2/d}$ (essentially because the number of sampling points grows exponentially in the dimension) whereas the Monte Carlo convergence is invariant under dimension. Thus MC techniques really come into their own in higher dimensions. In particular they do not suffer from the "curse of dimensionality".

[13]In fact, since $\int_a^b g(x)dx = E[g(U)]$ for U uniform on $[a,b]$, it follows that Monte Carlo can be used to approximate any integral!

[14]By draws we mean sampling points in the image of X which respects the distribution function. For instance if X is normal then the draws when plotted should more and more resemble the classical bell curve.

5.2 The design

We now move onto the design of our Monte Carlo pricing engine.

We assume we have been given a product to price. We can then break down the procedure our engine should follow into 3 steps:

1. Given the exercise time from the product, evolve the spot according to some dynamics up to this time.

2. Given this final spot time, get the payoff from the product and compute the value of the option on that path (including any discounting).

3. Repeat steps 1. and 2. for the desired number of paths, and then compute the average over all paths.

We can also add extra functionality to the engine by allowing the user to choose the method of random number generation implicit in the drawing of the paths[15], aswell as allowing the user to decided what statistics to compute for the path (for instance the user might want the sample variance). We therefore need to implement the following type of pseudo-code:

```
RunSimulation(Product, ShortRate, StatisticToGather, NumberOfPaths, SpotDynamics, ModelParameters, RandomNumberGenerator)
{   DoOnePath(Product, ShortRate, StatisticToGather, SpotDynamics, ModelParameters, RandomNumberGenerator)
    Repeat for number of paths
    Compute statistic for all paths }
```

where

```
DoOnePath(Product, ShortRate, StatisticToGather, SpotDynamics, ModelParameters, RandomNumberGenerator)
{   GetFinalSpot(Product, ShortRate, StatisticToGather, SpotDynamics, ModelParameters, RandomNumberGenerator)
    Compute value of product with that spot and discount
    Get relevant data for statistic from product value }
```

where

```
GetFinalSpot(Product, SpotDynamics, ModelParameters, RandomNumberGenerator)
{   Get exercise time from product
    Evolve spot up to that time according to given dynamics, with model parameters and random generator used in the simulation }
```

We therefore need to encode the following:

1. The notion of a product, which knows the exercise time and the payoff.

[15]There are three main reasons for this; 1. Only certain number generators are valid in high dimensions 2. The method of number generation, in particular the use of low discrepancy sequences, can greatly speed up the convergence 3. In built random number generators are compiler dependent so code using them will not be portable from the point of view of testing.

2. The notion of a statistics gatherer, which knows how to get relevant data from one path and how to compute the statistic it describes for all the paths.
3. The notion of a random number generator which knows how to generate elements in a sequence of random numbers according to the generator it describes.
4. The notion of evolving the spot, which when given a final time, model parameters and a method of random number generations knows how to evolve the spot up to that time according to the stochastic model it describes and using the random number generator to simulate the stochastic part.

Given these, the Monte Carlo pricer is easy to implement in code according to the above psuedo-code, up to issues of optimisation and preference in the design.

5.3 The VanillaOption class

We begin with representing the product in code.

We choose to design a class which contains the same information as would be on the contract that specifies the option, namely the exercise time and the payoff function. Our VanillaOption class will therefore have as data members a double representing the exercise time and a nested class PayOff representing the possible payoff functions. In particular, it will be an abstract base class having function like syntax with the exact implementation of the overloading of () left to derived classes.

The only non-trivial thing about this is then the fact that the VanillaOption class wants to have as data member an object whose type at compile time is that of an abstract class, and this is not allowed in C++ (because this would require fully constructing the object, which cannot be done for an abstract class). We could instead store by reference or pointer but then this would lead to an undesirable coupling between VanillaOptions objects and the PayOff objects used to initialise them. In order to get around this we need a type of "virtual copy constructor".

5.3.1 "Virtual copy construction": the clone method

To get around this we introduce a virtual clone method to the Payoff class. Recall when called by an object this method returns a newly allocated pointer to a derived class object which holds a copy of the data held by the calling object. Since pointers respect class polymorphism, this means we can then have a pointer to a base class object as a data member in our class which is initialised upon construction by the "clone" of whatever object in the class hierarchy is fed in as a parameter.

The clone method thus acts as a "virtual copy constructor" in that it produces a copy of an object whose type is not known at compile time. Of course, its not really a copy constructor as it returns a pointer. Indeed C++ does not allow virtual copy constructor because in C++ an objects can only be created if one knows everything about it.

The downside to the lack of a true virtual copy constructor in C++ is now we have to worry about memory leakage. Indeed the clone method uses new, so this will need to be deallocated somewhere. This leads us to *smart pointers.*

5.3.2 Smart pointers and modern C++

The way to deal with this issue is to design a wrapper class for pointers which handles the memory management for you. That is, new memory is allocated upon construction of an object in the wrapper class and deallocated when said object is destroyed. This is the "resource acquisition is initialisation (RAII)" paradigm.

In fact in C++11 such a wrapper class already exists, specifically *unique pointer.* This is defined in the package <memory> and has a template parameter much like the STL vector[16]:

unique-ptr<typename> P...

They are constructed from pointers allocated on the free store,

unique-ptr<typename> P {new typename}

and act exactly like pointers with two key exceptions:

1. A unique pointer *owns* the object is points to. In particular, when it is destroyed it deletes the object it points to.

2. A unique pointer cannot be copied.

Note the second make senses, since two objects cannot both own the same object.

Thus our VanillaOption class will have unique pointers to Payoff class objects as data members and these will be initialised upon construction by using the clone method of whatever Payoff class object is passed in to the constructor as a parameter.

The respective class definitions can be found in the "PayOff" and "VanillaOption" header files.

Remark 5.1. *Note that trying to (default) copy an object of class which has a unique pointer as a data member will cause a compile time error. Thus if this is something that the class will do one will need to implement either a copy or move constructor for this class. We will not to this here however.*

Remark 5.2. *When coding up the VanillaOption class I used a type alias for unique pointers to PayOff objects. I included this type alias in the class definition, which means that the type alias is* restricted *to that namespace. In particular, it cannot be used outside the class body without explicitly referring to that namespace and also provided that it was declared in the public part of the class definition.*

5.4 The StatisticsGatherer class

We now turn to represent the notion of statistic gathering in code.

5.4.1 Class design

From the above discussion, all such an object needs to do is know how to get data and compute statistics from paths. We therefore represent statistics gathering by introducing an abstract base class "StatisticsGatherer" having as pure virtual methods "GetDataFromPath" and "ComputeStatistic". The former shall be designed so as to take in a double (representing the value of the product on one path), compute data from that double relevant for the statistic, and then store the result as a data member. The latter method will then use this data to compute the statistic. Note this means that the former method will be non-const.

For instance, consider the statistic of gathering the mean. For this statistic, the relevant data is a running average of the values seen on the paths so far. Therefore for each path "GetDataFromPath" will just need to get the product value, and then recompute the average. The optimal way to do this would then be to have as data members "RunningSum" and "PathsDone" and each time the

[16]We use - in place of underscore.

method is called it adds a value to the "RunningSum" and increments "PathsDone" by 1. The method "ComputeStastic" then simply divides "RunningSum" by "PathsDone".

Of course, we will also want to give this class a clone() method so as to allow objects in this class to be used as data members which are stored by value (through the use of a smart pointer). The destructor will also need to be made virtual to handle the case where objects in this class are deleted via a pointer to their base.

Remark 5.3. *The design pattern of leaving the implementation of an algorithm (in this case the Monte Carlo simulation) to a seperate class (in this case what statistic to gather) is known as the* strategy pattern*.*

5.4.2 Aside: what data structure to store PathsDone in?

Recall the error in a Monte Carlo simulation is $O(N^{-1/2}$. This means that if we want to approximate the expectation to within 5 decimal places we might require an order of 10^10 paths. We therefore need to store PathsDone in a data structure that is capable of handling such a large number (or at least as close to).

Let us list now a few of the standard data types along with the number of bytes each store as standard in visual studio:

$$\begin{aligned} int &= 4\text{bytes} \\ unsignedint &= 4\text{bytes} \\ long &= 4\text{bytes} \\ unsignedlong &= 4\text{bytes} \\ longlong &= 8\text{bytes} \\ unsignedlonglong &= 8\text{bytes} \end{aligned}$$

Now x bytes are equivalent to 8 bits, and each bit is represented by a 0 or 1 in memory. Therefore, using the binary representation of an integer, in x bytes we can represent all numbers from

$$[-X, X], \quad X := 2^{8x-1} + 2^{8x-2} + ... + 2^0$$

whilst if we know the number has a sign we can go up to $2X$. It follows that an int or a long can store any number in the range

$$-2,147,483,648, ..., 2,147,483,647,$$

and unsigned int or an unsigned long can store any number in the range

$$0, ..., 4,294,967,295,$$

a long long can store any number in the range

$$-9,223,372,036,854,775,808, ..., 9,223,372,036,854,775,807$$

and finally an unsigned long long can store any number in the range

$$0, ..., 18,446,744,073,709,551,615.$$

In particular, we need to store the number of paths in an unsigned long long.

5.4.3 Additional functionality I: Convergence tables

When actually performing the simulation it will be very useful to have a convergence table for the statistic which outputs the results for, say, powers of 2. We can easily add this functionality to our Statistics gathering without changing the interface (i.e. the design of the abstract base class) as follows.

We define an inherited class "ConvergenceTable" which takes in a StatisticsGatherer object as a data member aswell as an integer representing the number of paths done. Each time the data gathering method for this class is called it then calls the data gathering method of the inner object aswell as incrementing its paths done data member. Moreover whenever the number of paths done is a power of 2 it calls the "ComputeStatistic" object of the inner object and stores the result in a vector of doubles, in turn stored as a data member. The j'th entry in this vector will thus represent the value of the statistic described by the inner object evaluated after 2^{j+1} paths have been done, and it is this vector that we want to output when the ComputeStatistic of the convergence table is called.

For this reason we will have to slighly change our original design so that "ComputeStatistic" returns a vector of doubles. It also makes more sense to rename it "ComputeStatisticSoFar". We will then just have to change the implementation of this method in the "Mean" class so as to output a vector of size 1.

Remark 5.4. *I encountered the following issue when designing this class; I got a compile time error when implemented the clone method as standard. This is because the standard clone() method implicitly calls the copy constructor of the class, which by default tries to copy everything, but this is not allowed for unique pointers (its copy constructor is deleted). To get around this, I implemented a copy constructor for the ConvergenceTable class which implemented a deep copy. More specifically, given an original ConvergenceTable object, the copy constructor constructs a new such object by initialising its unique pointer data member with a clone of the object held by the unique pointer data member of the original.*

Remark 5.5. *The above design principle is an example of the decorator pattern. That is, we have added extra functionality to a class without changing its interface. It was achieved in this case by introducing a new inherited class of the base class which takes a pointer to a base class object as a data member.*

5.4.4 Additional functionality II: Multiple statistics

It will be further useful if we can compute multiple statistics at once (for instance the sample mean and variance). We can add this functionality by again employing the decorator pattern.

Indeed we design a class "MultipleStatistics" which inherits from the StatisticsGatherer class and stores a vector of (unique pointers to) base class objects. We then just call each of the individual methods each time a method is called.

All that we need to change in the interface is to make "ComputeStatisticsSoFar" return a vector of vector doubles. When called by a single statistic, we will just make it of size (1)(1). When called by multiple statistic, we will make it of size (1)(n). When called by a convergence table, we will make it of size (m)(1). Finally, when multiple statistics are called by a convergence tables, it will be of size (m)(n).

Actually implementing this in code is a little involved due to all the subscript notation so well spell it out here. We start with the MultipleStatistics class. When calling the compute statistic method, this should output a vector of vectors which just concatenates the vector of vectors constructed by the inner objects (if one thinks of each as a nxn matrix one is simply concantenating the matrixes column by column). Therefore if we are given m statistics, so that the inner object is vector of vector of vector doubles of size m, and for $i = 0, ..., M-1$ we define N_i to be the size of InnerObjects[i].ComputeStatisticSoFar(), for each

$$k_i \in N_0 + ... + N_{i-1}, ..., N_0 + ... + N_i - 1 \qquad N_{-1} = 0$$

we want to define a vector of vector doubles with

$$v[k_i] = InnerObjects[i][k_i].$$

We then return v in the ComputeStatisticsSoFar for the MultipleStatistics object. Note in particular that if $N_i = 1$ for each i (as we would expect) then $k_i \equiv i$ and so we just output a vector of doubles.

Note it would have taken only a little bit of forward thinking to design the class in this way from the start.

5.4.5 Aside: range for loops etc.

We now make a brief aside to discuss some language constructs and issues I came across when coding the above.

Range-for loops

One thing I found extremely helpful were range-for loops (new for C++11). This provide an easy way to loop over the elements of a container v:

for(auto i : v){ i represents element in container which is looped over }

Note in particular one doesn't need to use subscript notation, in particular i now represents an element in the container. Of course, one can also pass by const, reference etc.

Checking for nullptr

It is important to always check whether a ptr is null before doing something to it:

typename* p;
if (p) // do something.

This is particularly important when dereferncing, because dereferencing a nullptr is undefined behaviour and so can lead to errors which aren't picked up till run time. In particular, every clone method should have a check to test whether this is null.

Use std::make-unique

In the memory package we have (from C++14 onwards) std::make-unique for making unique pointers e.g.

unique-ptr<int> p = make-unique<int>(5);

With the advent of make-unique, one never needs to explicitly use new or delete again and this is now the recommended practice. Indeed even with unique pointers the use of new can still cause memory leaks. For instance consider

```
    void f(unique-ptr p, unique-ptr q)
// code
f(unique-ptr (new ...), unique-ptr (new...))
```

One possible implementation of this by the compiler is:

1. allocate memory for the first new
2. allocate memory for the second new
3. construct the first object
4. construct the second object
5. construct the first unique pointer
6. construct the second unique pointer
7. call f

In particular, if an exception is thrown in either 3. or 4. then we will get a memory leak! This cannot happen if we used make-unique however as the allocation and construction is handled by the same object so a thrown exception will involve a call to delete (the exact reason why is more high level). We say that make-unique is *exception safe*.

In general, with the advent of C++14 one's code should never contain the use of new, delete or raw pointers. In particular our clone() methods should be outputting unique pointers (although I probably won't bother going back to change the clones already implemented) which are constructed via make-unique.

5.5 The RandomNumberGenerator class

We now move on to representing the notion of a random number generator in code.

5.5.1 The design

Recall that a random number generator is just a deterministic sequence of numbers, "evolved" according to some rule, which in the large looks uniformly distributed. In particular given a "seed" or current number in the sequence, all the random number generator needs to do is know how to get to the next number in the sequence. We shall therefore represent random number generators by introducing an abstract base class "RandomNumberGenerator" having a pure virtual method "GetNextNumberInSequence" which determines how to get the next number in the sequence according to whatever generator one is describing in a derived class.

Note for our purposes (we explain why a few sections down) it will actually be more convenient to have a method which returns a "draw" from a Uniform variable on $[0, 1]$. So instead of "GetNextNumberInSequence" we will have a pure virtual method "GetUniformDraw" which will implicitly turn an integer in the random sequence to a uniform draw. For this reason we shall leave how the "seed" variable is stored to derived classes, because there will be some interplay between the seed and the uniform draw that is generator dependent. We will however give a pure virtual method for setting the seed, because this determines where in the sequence we start and sometimes we might want to reset the sequence so that we can generate the same sequence again (e.g. for testing purposes, moment matching or finite differencing).

What shall be the return type of the method? At first glance one would think a double, but it will be very useful for later applications to be able to make many independent uniform draws at once. This suggests a return type of a vector of doubles. However, thinking of the Monte Carlo simulation, this uniform draw method will get called every singe loop resulting in potentially millions of vectors being created in one simulation, and this is obviously very costly. To get around this, we use the useful technique of passing in a vector of doubles by non-const ref and have the method update the vector in its scope (in the context of the Monte Carlo simulation this vector will be provided us outside of the loop). We therefore make the return type void.

There is one issue with making many draws at once. Indeed, not all random number generators are suitable in higher dimensions. This is because there can be funny correlation between the draws which can lead to a k dimensional draws clustering around a $k-1$ dimensional plane, and since these are measure 0 our Monte Carlo simulation will never converge. To check for this, we will therefore store as an unsigned long long "Dimensionality" and provided methods for getting and resetting it. We can then throw an error in a derived class if its being used for a dimension it is note suited to. Note the resetting method will need to be virtual because the resetting procedure can be generator dependent.

We shall provide one more method to the class which will be very useful for us, namely a "Get-GaussianDraw" method which turns the uniform draw into a draw from a standard normal variable. As before, for reasons of optimisation we make its return type void and pass in as a parameter a non-const ref to a vector. The method will then fill this vector with a call to "GetUniforms" which it then uses to refill it with Gaussian draws.

We can actually make this method concrete by implementing *inverse transform sampling* and an implementation of the inverse normal cdf due to Moro (I borrowed the code from Joshi – I'm pretty sure its basically some Taylor expansion approximation). The advantage of this method is that it works for any random number generator, including low-discrepancy numbers. In contrast other sampling methods based on rejection sampling will destroy the special properties of such sequences. Nevertheless such sampling methods, such as Box–Muller, can be faster computationally. For this reason we will make this method virtual to allow it to be overriden if indeed another method of generating Gaussians is quicker.

Let us in fact now make a quick aside to go over these sampling methods.

5.5.2 Aside: inverse transform sampling and Box–Muller

Let X be a random variable into R^n with distribution function F_X. Our aim is to obtain a draw from the range of X. One method is via inverse transform sampling.

Suppose that U is a uniform random variable on $[0,1]^n$. Now F_X is monotonically increasing hence has an inverse $F_X^{-1} : [0,1] \to R^n$. Define now $Y : R^n \to R^n$ by

$$Y(x) = F_X^{-1}(U(x)).$$

Then

$$F_Y(x) = P(Y < x) = P(U < F_X(x)) = F_X(x).$$

Therefore Y has the same distribution as X. It follows that if u is a random draw from U then

$$y = F_X^{-1}(u)$$

is a random draw from X. This is the content of the inverse transform method: it allows to get random draws from X provided one has a procedure for obtaining random draws from a uniform random variable.

The downside with this method is that computing the inverse cdf can be costly computationally speaking. In the case of a Gaussian random variable, the Box–Muller transform provides an alternative method.

Indeed let U_1, U_2 be Uniform random variables on $[0,1]$. Then the claim is that

$$\begin{aligned} Z_1 &:= \sqrt{-2\log(U_1)}\cos(2\pi U_2) \\ Z_2 &:= \sqrt{-2\log(U_1)}\sin(2\pi U_2) \end{aligned}$$

are independent $N(0,1)$ variables. Indeed, first recall more generally that if X, Y are random variables and we define the random variables

$$(S,T) = h(X,Y)$$

then the joint density function of S, T is given by[17]

$$f_{S,T}(s,t) = f_{X,Y}(h^{-1}(s,t))|J_{h^{-1}}(s,t)|$$

with J the Jacobian of the transformation. Since in our case

$$h^{-1}(S,T) = (e^{-\frac{S^2+T^2}{2}}, \frac{1}{2\pi}\tan^{-1}(T/S))$$

the result easily follows. In particular, given two independent Uniforms we can sample from the Gaussian distribution by using the above transformations.

It turns out that the trigonometric functions implicit in the Box–Muller transform are actually quite costly computationally. We can remove them at the cost of rejecting certain draws as follows.

Indeed, let U, V be Uniform, independent on $[0,1]$ with (U,V) constrained to lie on the unit circle:

$$U^2 + V^2 \leq 1.$$

Then we claim that

$$S = U^2 + V^2, W = \frac{2}{\pi}\tan^{-1}(V/U)$$

are independent uniform random variables on $(0,1)$ [Note the restriction on (U, v) ensures the images of S, W indeed lie in $(0,1)$]. Indeed since W is simply proportional to the angle formed between the origin and (U,V) the Independence is manifest. On the other hand

$$P(S \leq s) = P((U,V) \text{ lies inside the circle of radius } \sqrt{s}) = \frac{\pi/4s}{\pi/4} = s$$

[17]We briefly recall the proof. We have

$$F_{S,T}(s,t) = P(h(X,Y) \leq (s,t)) = F_{X,Y}(h^{-1}(s,t)) = \int_{-\infty}^{(h^{-1})^1(s,t)} \int_{-\infty}^{(h^{-1})^2(s,t)} f_{X,Y}(x_1,x_2)dx_1dx_2.$$

Now make the change of variables $x_i = (h^{-1})^i(z_1,z_2)$ and differentiate.

and similarly

$$P(W \leq w) = P(\text{ angle between } (U,V) \text{ and origin is less then}^{18} \ \frac{\pi}{2}w) = \frac{\pi/4w}{\pi/4} = s.$$

We can therefore substitute $U_1 = S, U_2 = W$ in our Box–Muller transform to find that for independent uniforms U, V constrained to satisfy $U^2 + V^2 \leq 1$ we have

$$Z_1 := \sqrt{\frac{-2\log(S(U,V))}{S(U,V)}}U$$
$$Z_2 := \sqrt{\frac{-2\log(S(U,V))}{S(U,V)}}V$$

are independent $N(0,1)$ variables.

We can therefore generate Gaussian draws from uniforms, without using the costly trig functions, provided we *reject* those draws which do not form points which lie in the unit circle. This method therefore utilises *rejection sampling*. The fact we have to discard certain draws is precisely why such a sampling method is inadequate for low discrepancy sequences.

It is natural to ask what is the expected number of draws until success. Let us denote by a trial a pair of draws, and a successful trial is one that isn't rejected. Let also X be the number of trials until success and A_k the event that the k'th trial is the first success. Then

$$E[X] = \sum_{k=1}^{\infty} E[X|A_k]P(A_k)$$
$$= \sum_{k=1}^{\infty} k\frac{\pi}{4}\left(\frac{3\pi}{4}\right)^k$$

as

$$P(\textit{rejected}) = P(S > 1) = 1 - \frac{\pi}{4}.$$

Now recalling that for $|x| < 1$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \to \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

we infer

$$E[X] = \frac{4}{\pi} \approx 1.2732.$$

So we need on average approximately 1.3 trials, and hence 2.6 draws until we make a successful draw.

5.5.3 A concrete class: Park–Miller

We now implement a concrete derived class which implements the Park–Miller random number generator. Let us begin by recalling how this works.

Park–Miller sequence

The Park–Miller random number generator is an example of a *multiplicative congruential generator*:

$$X_{n+1} = aX_n \bmod m.$$

Specifically we have

$$a = 7^5 = 16,807, \quad m = 2^{31} - 1 = 2,147,483,647.$$

Here the a is chosen for number theoretic reasons whilst m, a Mersenne prime, is the largest number that can be put into an int which makes the sequence very portable. These choices of a, m make the generator reasonably good in that it passes many statistical tests checking for uniformity[19]. It also has the optimal[20] period of $2^{31} - 2$ that is, for any given any initial seed it takes that long until the sequence begins to repeat. It is therefore a very good sequence for making random draws if the number of draws one wants to make is not exceedingly large (in particular, it can fit into an int).

Turning now to implementation of this sequence in a portable manner one arrives at the following issue: although each element in the sequence can fit into an int, the number $a * (m - 1)$ is larger than 32 bits so when implementing the algorithm in the obvious fashion one will see overflow before the modular arithmetic can be done. To ameliorate this, Schrage introduced an algorithm for implementing the Park–Miller sequence using only 32 bit numbers.

First let q, r with $r < q$ be such that

$$m = aq + r.$$

Then for any $z \in \{1, ..., m-1\}$ we have

$$a * (z \mod q), \quad r * \lfloor z/q \rfloor \in \{1, ..., m-1\}$$

and

$$(a * z) \mod m = \begin{cases} a * (z \mod q) - r * \lfloor z/q \rfloor, & \text{if expression} \geq 0, \\ a * (z \mod q) - r * \lfloor z/q \rfloor + m, & \text{otherwise.} \end{cases}$$

In other words, given an element in the PM sequence we can find the next element by using only 32 bit numbers. The Schrage algorithm specifically takes

$$r = 2836, \quad q = 127,773.$$

A ParkMiller class

Forgetting for the moment our RandomNumberGeneratorClass we design a class to represent a Park–Miller generator.

[19]It is not exactly uniform because e.g. very small numbers in the sequence are always followed by numbers smaller than the average, hence not random. Indeed if X_n is very small that X_{n+1} is only a factor of 1.6×10^4 larger than it, and this will be much smaller than the average given the largest is or order 2×10^9. This can be improved upon by choosing other generators such as e.g. shuffled Park–Miller.

[20]Provided of course one does not start from 0.

As a data member this class we have an int "Seed" which represents the current member of the sequence and will be initialised upon construction (we throw an error if user tries to use 0). It will then have a method "GetNextNumberInSequence" which implements the Schrage algorithm for generating the next number. We will also provide a method for setting the seed in case we want to reset the generator. Note both these methods will be non-const since they change the seed. We will also make the constants appearing in the algorithm static const ints defined the GetNextNumber method[21]. This means those constants will not go out of scope between successive calls on the method, thus saving time in the construction[22] Note we could also have declared them static data members of the class which means all objects in that class will share them. However it is more logical that they are stored in the method because they are more naturally linked to the method than the class (in particular if the method is never called then they need not be constructed).

We finally provided methods for returning the min and max of the sequence – we will need this when it comes to convert the sequence to uniform draws on $[0, 1]$.

Remark 5.6. *When designing classes, the best practice is to first introduce public members, then protected and then private. This is because users will read the file top to bottom and users will only be interested in the members they have access to (i.e. the public members) and will not care about the implementation details encapsulated in the private members. I will try to stick to this in the future. Note also that class members declared not under any of the headings are implicitly private.*

Remark 5.7. *Remember to find files saved in other directories one simply uses "../..." next to the include directive.*

5.5.4 Aside: how compilation works

I now want to make a brief digression to discuss how compilation works in C++ since it lead to some confusion on my part.

The compilation of a C++ program involves three steps:

Preprocessing: the preprocessor takes a C++ source code file (a .cpp file) and deals with the includes, defines and other preprocessor directives. The output of this step is a "pure" C++ file without pre-processor directives. There is such a file constructed for each .cpp file in the project.

Compilation: the compiler takes the pre-processor's output and produces an object file (a .obj file) from it.

Linking: the linker takes the object files produced by the compiler and produces either a library or an executable file.

Preprocessing: The preprocessor handles the preprocessor directives, like include and define. It is agnostic of the syntax of C++, which is why it must be used with care.

It works on one C++ source file at a time by replacing include directives with the content of the respective files (which is usually just declarations), doing replacement of macros (define), and selecting different portions of text depending of if, ifdef and ifndef directives. Thus at this stage header files are merged with the source files for which they have been included.

The preprocessor works on a stream of preprocessing tokens. Macro substitution is defined as replacing tokens with other tokens.

[21]Note the general practice of avoiding the use of global variables. This is because it is not clear which functions use them, and this could ultimately lead to undesirable coupling between different areas of code. Note however that global variables are only global relative to the source file in which they are defined, in according with the above compilation procedure (this had confused me earlier because I thought every source file in the project would have access to them).

[22]It is of course minimal in this case but it is best practice.

After all this, the preprocessor produces a single output that is a stream of tokens resulting from the transformations described above. It also adds some special markers that tell the compiler where each line came from so that it can use those to produce sensible error messages.

Some errors can be produced at this stage with clever use of the if and error directives.

Compilation: The compilation step is performed on each output of the preprocessor i.e. each object file produced. The compiler parses the pure C++ source code (now without any preprocessor directives) and converts it into assembly code. This is then turned into a binary file of some format (i.e. a files of 0's and 1's). This object file contains the compiled code (in binary form) of the symbols defined in the input. Symbols in object files are referred to by name.

Object files can refer to symbols that are not defined. This is the case when you use a declaration, and don't provide a definition for it. The compiler doesn't mind this (will assume its defined in another file), and will happily produce the object file as long as the source code is well-formed. However if a variable is declared in one source and one tries to use it in another file for which the former has not been included then one will get a compile time error.

Compilers usually let you stop compilation at this point. This is very useful because with it you can compile each source code file separately. The advantage this provides is that you don't need to recompile everything if you only change a single file.

The produced object files can be put in special archives called static libraries, for easier reusing later on.

It's at this stage that "regular" compiler errors, like syntax errors or failed overload resolution errors, are reported.

Note these seperate files are called *translation units.*

Linking: The linker is what produces the final compilation output from the object files the compiler produced. This output can be either a shared (or dynamic) library (and while the name is similar, they haven't got much in common with static libraries mentioned earlier) or an executable (a .exe file).

It links all the object files by replacing the references to undefined symbols (e.g. variables, functions etc.) with the correct addresses. Each of these symbols can be defined in other object files or in libraries. If they are defined in libraries other than the standard library, you need to tell the linker about them (this is what the include directives actually accomplish).

At this stage the most common errors are missing definitions or duplicate definitions. The former means that either the definitions don't exist (i.e. they are not written), or that the object files or libraries where they reside were not given to the linker. The latter is obvious: the same symbol was defined in two different object files or libraries (in particular declaring the same variable in two different source files which do not include each other will lead to a link-time error).

Note if a source files has previously been compiled and has not been changed since the compiler will remember this and not recompile it. This can be very important if one is including large libraries, and is part of the reason why having different source files for different actions is encouraged.

We see now that header files don't actually do much, they just provide a convenient way to decouple different parts of a code in a much which allows errors to be picked up at compile time (this is why we only need to provide declarations). However they do serve a design purpose; a header file should contain all what the user needs to know to use the code (e.g. what parameters functions take) with all the implementation details, which the user doesn't need to know, hidden in the source files.

Note also it is bad practice to use using declarations (e.g. using std::string) or directives (e.g. using namespace std) in header files. This is because header files are generally designed to be used in other files and so this could ultimately lead to name clashes (for instance, a class defined in std might have a common name that is likely to be used in another file). It is fine however to use them in source files, provided they are introduced after any include directives (for the same reason), because then the declarations/directives are constrained to that particular unit.

5.5.5 The Park–Miller random number generator

We now want to design a derived class of our RandomNumberGenerator class which implements the Park–Miller generator. Now, even though we have already implemented this algorithm as a class, it does not quite fit into how we have designed the RandomNumberGenerator class. But this is quite easily remedied by having our class take a ParkMiller object as a data member and then using its methods in each of the generators methods. This design pattern of "adapting" one class to fit into the interface of another design class is called *the adapter pattern*. It is most typically used when working with old code.

The methods are fairly simple to implement. The only thing to be aware of is in the set dimensionality method once has to be sure to call the set dimensionasility method of the base class using the familiar namespace syntax (i.e. ::) as the derived class to do not have direct access to the private members of the base class. There is also a subtlelty with the GetUniformDraw method in that we need to ensure our draws like in $(0, 1)$, in particular they do not obtain the end point values. This is because we are using the inverse sampling method, and so 0, 1 would get mapped to $-\infty, \infty$ respectively, and is achieved by simply dividing the random number genrated by the inner ParkMiller object by the max + 1 of the ParkMiller sequence.

We also need to change our clone method to adhere to our new rule of not using raw pointers. Thus we change the return type of the base clone method to a unique pointer to a base class object. The only issue is then that smart pointers don't respect covariance, so we cannot make the return type of the clone method in the derived class return a unique pointer to a derived class object. Nevertheless, if we keep the return type fixed but implement it using make-unique¡Derived¿(*this) then the returned unique pointer will respect polymorphism. That is to say, we can manipulate a derived class object via a unique pointer to its base. Thus using the syntax

$$unique - ptr < Base > p = make - unique < Derived > // = DerivedObject.clone()$$

we get the familiar polymorphic semantics of raw pointers[23].

Note that this lack of covariance in the return type of the smart pointers DOES lead to some issues if a Derived class inherits from more than one class since then it is not clear which base return type to use (this is NOT an issue with raw pointers). One way around this would be simply to have two clone methods, one for each base class, which explicitly returns a smart pointer to the specific base class.

5.5.6 Added functionality: Anti-thetic sampling

We now add extra functionality to our class by allowing for anti-thetic sampling, whose theory we now recall.

Anti-thetic sampling: the theory

Anti-thetic sampling is an attempt to reduce variance in a Monte Carlo simulation by introducing negative correlation between pairs of draws. There are various ways of achieving this but for our purposes we assume we are working with uniform variables. It then follows that if U is uniform on

[23]In particular we still need virtual destructors!

$[0,1]$ then so is $1-U$ so that from independent draws $U_1, ..., U_n$ we can generate additional draws $1-U_1, ..., 1-U_n$. The corresponding pairs are called anti-thetic pairs because if U is large then $1-U$ is small. More precisely they have *negative correlation*[24]; therefore when making draws any very large draws which are far from the mean will be balanced out by a very small draw.

We can extend this analysis to any distribution by means of the inverse transform method. Indeed, given a random variable X then we know that $F_X^{-1}(U)$ and $F_X^{-1}(1-U)$ are of the same distribution. They are also anti-thetic because F_X^{-1} is monotonic [see proof later]. Moreover, in the special case that F_X is "symmetric" about the origin we have

$$F_X^{-1}(1-U) = -F_X^{-1}(U).$$

In particular for an $N(0,1)$ variable X and $-X$ are anti-thetic. In fact $\rho_{X,-X} = -1$.

Let us now quantify the effect of anti-thetic sampling on the variance in a Monte Carlo simulation. Suppose we are trying to estimate $E[f(X)]$ and we have a means of generating anti-thetic draws from X, where X is either uniform or normal. Consider

$$\hat{\mu}_{AT} = \frac{1}{2N}\sum_{i=1}^{N}(f(X_i)+f(X_i')) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(X_i)+f(X_i')}{2}\right)$$

where X_i' is an anti-thetic of X_i (i.e. $X_i' = 1-X_i$ if X is uniform or $X' = -X_i$ is X is normal) and each pairwise draws are independent. Then clearly $\hat{\mu}_{AT}$ is the sample mean of

$$\frac{f(X)+f(X')}{2}$$

so that by the central limit theorem

$$E[f(X)] = E[\frac{f(X)+f(X')}{2}] = \hat{\mu}_{AT} + \frac{\sigma_{AT}}{\sqrt{2N}}N(0,1) + ...$$

where

$$\sigma_{AT}^2 = 2Var[\frac{f(X)+f(X')}{2}] = (Var[f(X)] + Cov(f(X), f(X'))).$$

Here $E[f(X') = E[f(X)]$ etc. follows from the fact that X' is an anti-thetic of X.

Comparing this estimator with one obtained from a standard Monte Carlo estimate of the same number of draws we see that we have reduced the variance iff

$$Cov(f(X), f(X')) < 0$$

with greater reduction achieved the more negative this is. What conditions on f guarantee this? We have the following: Let (Ω, P) be a probability space. Let also f be monotonic, and g also monotonic but of the opposite nature. Then

$$\int_\Omega fg dP < \int_\Omega f dP \int_\Omega g dP.$$

Indeed, we may without loss of generality assume that f is increasing. Then for any $t \geq s$

$$0 > (f(t)-f(s))(g(t)-g(s))$$

[24]I think this might be the correct definition of anti-thetic.

which yields the result after integrating first in t, then in s, and using that P is a probability measure (so that the space has measure 1). If we assume now that the f we are estimating is monotonic, we have that the function $g(X) := f(X'(X))$ is also monotonic but of the opposite nature. Therefore by the above inequality

$$E[f(X)f(X')] = E[f(X)g(X)] < E[f(X)]E[f(X')]$$

so that $Cov(f(X), f(X')) < 0$. Note in particular that for a vanilla option the payoff function has the form

$$f(g(X))$$

where X is $N(0,1)$ and g is a monotonic exponential function. In particular, if the payoff function is monotonic then anti-thetic sampling will reduce the variance.

Implementing anti-thetic sampling
We now add anti-thetic sampling to our random number generator class. We do this using a decorator pattern. In particular, we design a dervied class which takes a RandomNumberGenerator object as a data member.

The class will also store a bool DrawAntiThetic to indicate whether the draw we are currently making is a new draw or the anti-thetic of the previous one. Subsequently, we will also strore a vector¡double¿ to hold the previous draws. Given these, the methods are straightforward to implement.

Remark 5.8. *I encountered the following issue when implementing the class. Recall previously I encountered an issue with the clone() method (when using raw pointers) for class which stored unique pointers as a data member. This is because the clone method implicitly tried to shallow copy the object which resulted in a compile time error. Now this time around I implemented the proper clone() method using smart pointers (in particular make-unique) but, without having yet defined a deep copy constructor, I did not recieve any compile time errors. At first I though this mean make-unique was implicitly using move or something but this isn't the case. Indeed, the same issue with shallow copying remains, its just that this issue now becomes a link-time error, because its calling a function whose definition is in another file.*

Remark 5.9. *Don't forget to check for null pointers!*

5.6 Evolving the spot

We now finally move onto evolving the spot.

5.6.1 The design

Now we could design a "Simulations" class hierarchy for representing spot dynamics but we should instead leave it to a method in the vanilla engine itself (this is what Joshi does). Therefore we will have a "VanillaEngine" class which acts as an interface having a pure virtual method "GetFinalSpot" whose actual implementation will be left to concrete derived classes.

What should this function look like? In order to work, it will need as input a model for involving the spot, market parameters for that model, a method of random number generation and finally the exercise time of the product. Now the first of these will be encoded in the implementation itself whilst the second two are model dependent (in particular, some models require might require more than one random number generator). We shall therefore only take as a parameter a double representing the final time, and we make the return type double also; the market parameters and the random number generator will then have to be taken as data members in derived classes. Note this means that the method will be non-const since it will change the generator when its called.

We might aswell at this stage allow for Euler stepping too. We therefore take in another parameter representing the number of Euler steps and track the spot between these times (the step times are given by $t_i = T/N * i, i = 1, ..., N$, the initial time being given). Note that for pricing we still only care about the final time, so we keep the return type double. However the dimension of the random number generator will need to know about N. We will assume this is done outside the function in the engine itself.

There remains the question as to how the function will obtain the random draws. Indeed, recall we set up the generator to act on a vector of draws by reference. We will therefore need to introduce this container in the function somehow. We have a few options; pass in a vector as a parameter by reference, create a container in the method, store the container as a data member in derived classes. However, the first is undesirable because it will be model dependent based on how many random numbers one needs per draw and the second is very costly in the context of an MC loop[25]. We therefore prefer the third. Note in particular this choice allows the flexibility of representing different models which taken in different types of random numbers[26].

5.6.2 A BlackScholes model

We now implement this method for the BlackScholes model.

We first assume that the relevant market parameters are stored as data members. We also store a RandomNumberGenerator object aswell as a container for holding the Gaussian draws. The implementation is then quite simple. The only thing to note is that it is better to work with the log of the spot and then take exponentiations at the end. This is cheaper computationally than taking the exponential over each step size.

5.7 Completing the class

We now finish the construction of VanillaEngine class.

We supplement the GetSpot method with a method for doing the simulation. The RunSimulation method will take in a Product, a stats gatherer and the number of Euler steps, aswell as the number of paths. It will then compute the discount factor for the given product and set up the MonteCarlo loop in which we call the GetSpotMethod, compute the PayOff on the path, discount and then feed into the stats gatherer object. The stats gatherer object will then store the statistic at the end of the loop and we can do with it what we wish.

Note the advantage of taking in all these objects as parameters instead of storing as members is that we can then easily price many different products without having to keep updating the engine. A similar statement holds if want to compare results with varying numbers of Euler steps. Since however in both these cases the market enviroment would remain fixed we shall store the short rate for discounting as a member, initialised in the constructor. Note we make it protected because derived classes will also want access to it.

There is one final thing we need to do, and that is ensure that the dimension of the generator we are going to use matches the Number of Euler steps (which will in turn need to resize whatever

[25] We could create a static local variable to ameliorate this although this seems overkill.

[26] E.g. for a BlackScholes class we would have a container for the Gaussian draws, whilst for a JumpDiffusion class we might have a container for the Gaussian draws and the jump draws.

stored containers we have). Since the generator and the containers are tied to derived classes, we therefore allow for another pure virtual method CheckDimension which we implement appropriately in derived classes.

5.8 Checking the engine

We can now run tests to see if our engine works.

5.8.1 Issues encountered

Lets begin by discussing some errors I find in the code whilst testing:

- I made a mistake in the convergence table class. Basically, I never gave the ResultsSoFar member a size so was getting a range error. To deal with I had to change the format of ResultsSoFar so that I could just push-back into it (thus never needing to give it a size). Under this new format, for m statistics and N paths:

  1. Results has size $m * (N^* + 1)$ where N^* is largest integer such that $2^{N*} < N$.
  2. Results[i] has size m for each i.
  3. Results[i][j] is the result after 2^{i+1} paths for the j'th statistic.

  This also meant I had to change the implementation of the ComputeStatistics method for the MultipleStatistics class.

- I made an additional mistake in the implementation of the MultipleStatistics class. Recall this class takes a vector of unique-ptrs to StatsGatherer objects as a data member. Originally, I was taking in a vector of Stats Gatherer objects and using these to fill up the vector using make-unique. But this obviously resulted in an error because a) one cannot have a vector of abstract classes b) the make-unique I was using was trying to return an object of an abstract class. This was easily dealt with my instead taking in a vector of unique-ptrs to Stats Gatherer objects and then initialising the data member by calling each of their clone() methods (properly implemented to return unique-pointers). Note pointer polymorphism means that one can indeed have a vector of pointers to different objects in a class hierarchy, and also the clone() method respects this.

- I also made a further mistake in the MultipleStats class, although this one was rather subtle. When implementing the compute statistics method I had the following code:

  ```
  for(const auto v : InnerObjectsPtr[k]->ComputeStatistic()[i])
  ```

  The intent was to loop over the elements of the result vector defined by the k'th statistic. However when running, the code only looped over the first element. The issue was to do with *temporary* objects.

  To understand, we need to explain what happens when a function returns a value. Consider the following:

  ```
  A foo() { A tmp; return tmp;}
  A a = foo();
  ```

This will result in the following:

A' constructor is called when tmp is declared
A's copy constructor is called by the return of foo() to generate a copy of tmp which is stored in a
A's destructor is called for tmp as goes out of scope
A's destructor is called by a when program ends.

Consider now the following:

```
A foo() { A tmp; return tmp;}
foo();
```

This will result in the following:

A' constructor is called when tmp is declared
A's copy constructor is called by the return of foo() to generate a copy of tmp and a temporary object of type A is created to hold the copy
A's destructor is called for tmp as goes out of scope
A's destructor is called by the temporary object (occurs when see ;).

In particular since we did not give a name to foo() it is just an r value, and the temporary object held by it gets destroyed pretty much straight after the function call.

We now see what the issue was with our above code; in the for object the call of the method results only in a temporary object which is then destroyed once the semi-colon is seen in the body of the for loop. So the result is as if the vector returned is of size one. This is easily remedied by creating a vector tmp to hold the object.

Note the compiler actually gave a warning about this (not an error) by referring to a dangling pointer. Here a dangling pointer is one which points to invalid data, or data that is not valid anymore (so its invalid in this case because it has been destroyed).

- I also made a mistake in the implementation of the GetFinalSpot in my use of a lambda expression. Basically, I had something like

  ```
  auto foo = [CurrentLogSpot](...)return CurrentLogSpot+...
  // do something
  CurrentLogSpot = f
  ```

  and I had written my code under the assumption that f would update the current log spot which would then update the CurrentLogSpot in the lambda body. This didn't happen however because I didn't capture by reference; doing so, I get the intended behaviour. Why does this work? Basically, a lambda expression creates a function object with data members whatever is in the capture class, and these members are stored by reference if indicated so in the clause. This is why capturing by reference works in this case, because the function object created by the lambda is then tied to the member variable (not a copy of it).

The following remarks are also in order:

Remark 5.10. *I kept getting errors about trying to reference a deleted function when trying to initialise a vector of unique pointers using an initialiser list (curly braces). The issue it seems is that, at least for vector, initialising in this way calls the copy constructors of the objects found in the initialiser list. To get around this, I just pushed-back.*

Remark 5.11. *One can actually initialise the members of a class within the class definition, but only by using = or curly braces.*

5.8.2 Testing the engine

We now begin testing our engine proper. We try pricing a call using market parameters

$$d = 0, T = 1, r = 0.05, S = 100, K = 100, vol = 0.2.$$

We run the tests with 1 million paths and output our data to file to be read in excel. Here is the resulting data:

Analytic value: 10.4506

Number of Euler steps: 12 Random number generator: Park--Miller Variance reduction: Anti-thetic

Sample Mean	Sample Variance	Number Paths (Powers of 2)
17.9385	643.579	1
16.4016	364.98	2
14.5943	363.836	3
12.1719	312.01	4
11.3545	233.748	5
9.95375	218.09	6
10.3524	221.999	7
9.92064	195.241	8
10.1665	218.879	9
10.0437	209.541	10
10.2993	215.775	11
10.3133	213.772	12
10.4121	216.432	13
10.424	216.513	14
10.3724	215.786	15
10.3818	215.069	16
10.4219	215.208	17
10.4368	216.121	18
10.4375	216.216	19
10.454	216.92	20

We see that the price agrees to 2 decimal places only after around a million paths (note: $2^{10} \approx 1000$). We also see that the sample variance seems to be settling down to around 217, so an estimate for the standard deviation is approx. 14.73. We can use this to compute the standard error of the Monte Carlo.

Recall the standard error is given by the expression

$$\frac{\hat{\sigma}_N}{\sqrt{N}} N(0,1)$$

and with it we can compute confidence intervals:

$$\begin{aligned} P(|E[f(X)] - \hat{\mu}| \leq \alpha) = P(|N(0,1)| \leq \alpha\sqrt{N}/\sigma) &= N(\alpha\sqrt{N}/\sigma) - N(-\alpha\sqrt{N}/\sigma) \\ &= 2N(\alpha\sqrt{N}/\hat{\sigma}_N) - 1. \end{aligned}$$

Therefore in our case if we want to be 99 percent certain that our estimate is within 2 decimal places of the correct value then we need to choose N so that

$$\sqrt{N} \geq 10^3 * 14.73 * N^{-1}(0.5 * 1.99) \approx \sqrt{1.44 * 10^9}.$$

This is of the order of 1 billion, and is consistent with the data.

Note I also experimented with different number of Euler steps and anti-thetic sampling:

Analytic value: 10.4506

Sample mean (1 Euler Step)	Sample mean (12 Euler Steps)	Sample mean (52 Euler Steps)	Sample mean (365 Euler Steps)	Number of paths (powers of 2)
68.7047	17.9385	5.22072	3.97743	1
41.2239	16.4016	6.95899	11.9155	2
23.5229	14.5943	8.25763	7.73031	3
16.7295	12.1719	7.69888	9.80634	4
13.9934	11.3545	8.10197	10.7641	5
12.1947	9.95375	8.06384	12.8455	6
11.2019	10.3524	9.01197	11.0411	7
11.276	9.92064	9.72857	10.8218	8
10.7652	10.1665	10.1537	10.7485	9
10.3886	10.0437	10.3176	10.2994	10
10.1245	10.2993	10.2952	10.2646	11
10.315	10.3133	10.3891	10.3442	12
10.4006	10.4121	10.4311	10.4209	13
10.5154	10.424	10.3531	10.4348	14
10.5147	10.3724	10.441	10.4203	15
10.42	10.3818	10.4044	10.438	16
10.4355	10.4219	10.4262	10.4572	17
10.4348	10.4368	10.442	10.4418	18
10.4386	10.4375	10.4525	10.4645	19
10.4456	10.454	10.4512	10.4628	20

As one can see, this lead to some gains but not much[27]. Note also that increasing the number of Euler steps increases the computational cost.

The code for running the tests can be found in the "RunTheSimulation" function defined in the main.cpp file.

I now want to make a brief aside to discuss outputting to a file.

5.8.3 Aside: inputting and outputting to files

In C++ inputting from files and outputting to files is handled by the standard library packages fstream (for file stream) and iostream (for input/output streams). Defined in this package are classes "ifstream" and "ofstream" which specifically handle said operations.

For instance, suppose we want to output data to a text file. This is done using the following code:

ofstream ost "myfile.txt";
ost << "Hello World" << endl; ost.close();

Note in particular that is important that one makes sure to close the file once finished (one also needs to open the file, but this is done automatically upon initialisation if providing an initialiser list). This is done automatically once the ofstream object goes out out scope, so the ideal is to always open and close files within a function body.

Now this basic outputting of data leads to a file that is quite primitive in nature. In particular it is quite difficult to format it correctly. To get around this, I decided to export the data from the MC simulation to a .csv file (comma seperated value) which can be read by excel, and thus the formatting of the file can be handled by excel (for which it was desinged to do). Indeed when reading .csv files, excel will interpret new lines as new rows and commas as the start of a new column. Thus to produce output that can easily be read into a spreadsheet all one has to do is to produce "," in the appropriate places in the source file.

Note it probably makes more sense to actually leave most of the formatting duties to the excel so file. In particular its probably easiest just to keep outputting the data.

Remark 5.12. *For future reference; ofstream is a class which inherits from the class ostream. In particular the latter reperesents an output stream, whilst the former represents a* particular *type of output stream, namely to a file.*

6 Investigations

We finish this section with some investigations[28]. We begin by studying the terminal distribution of the spot, namely the log-normal distribution.

6.1 The log-normal distribution

Recall that a random variable X is said to be log-normal, $X \sim lognormal(\mu, \sigma^2)$ if $\log(X) \sim N(\mu, \sigma^2)$. In particular X has density

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma x} \exp\left[-\frac{1}{2}\left(\frac{\log x - \mu}{\sigma}\right)^2 \right].$$

[27] This is perhaps not so surprising for the anti thetic case as the payoff is not monotonic.

[28] Joshi says we use what we have done to perform these although it doesn't really seem like we need to...

From this, or using that $\log(X)$ is normal, we compute

$$E[X^p] = \exp\left(p(\mu + \frac{p\sigma^2}{2})\right)$$

from which we infer

$$\begin{aligned}
E[X] &= exp(\mu + \sigma^2/2), \\
Var[X] &= exp(2\mu + \sigma^2)(exp(\sigma^2) - 1), \\
Skew[X] &= (\exp(\sigma^2 + 2)\sqrt{\exp(\sigma^2) - 1}, \\
Kurtosis[X] &= \exp(4\sigma^2) + 2\exp(3\sigma^2) + 3\exp(2\sigma^2) - 3
\end{aligned}$$

where recall

$$\begin{aligned}
Skew[X] &= \frac{E[(X-\mu)^3]}{\sigma^3} = \frac{E[X^3] - 3\mu\sigma^2 - \mu^3}{\sigma^3}, \\
Kurtosis[X] &= \frac{E[(X-\mu)^4]}{\sigma^4}.
\end{aligned}$$

The fact that the skew is positive then indicates that more of the probability mass lies to the left of the mean than to the right whereas the fact that the Kurtosis is greater than 3 (the Kurtosis of a normal variable) indicates that X has "fat tails". Indeed[29]

$$\begin{aligned}
X_{median} &: = F_X(1/2) = \exp(\mu) < E[X], \\
X_{mode} &: = x \text{ such that } f_X(x) \text{ is the maximum } = \exp(\mu - \sigma^2)
\end{aligned}$$

which supports the former claim, whilst a simple analysis shows that for fixed μ, σ there exists an $\alpha^* = \alpha^*(\mu, \sigma)$ such that for all $\alpha \geq \alpha^*$

$$P(X \geq \alpha) > P(N(\mu, \sigma^2) \geq \alpha)$$

hence showing that tail or extreme events or more likely. Note both phenomena essentially arise because the logarithm is "asymetric" in that it blows up in a small interval $(0, 1)$ compared to a blow up at infinity. It is also the case that both phenomena become more pronounced for larger values of σ.

This positive skewness and fat-tailedness is also evident from the following plots of the density function.

If we specialise now to the special case of the terminal distribution of spot in a Black–Scholes world then we have

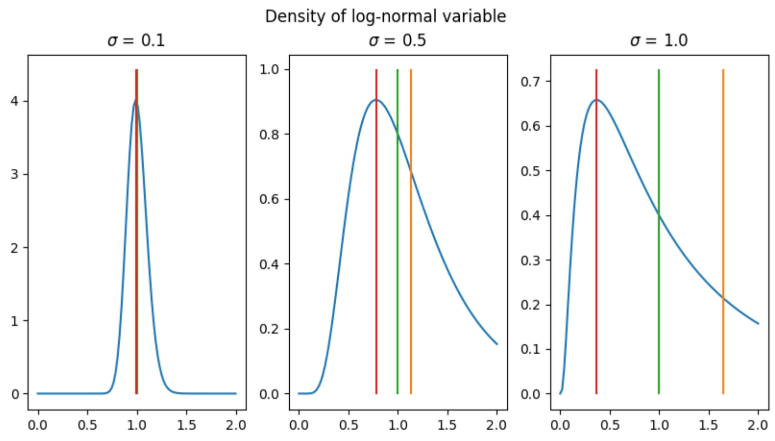
$$\mu = \log S + (r - \frac{1}{2}\sigma^2)T, \quad \sigma \to \sigma\sqrt{T}$$

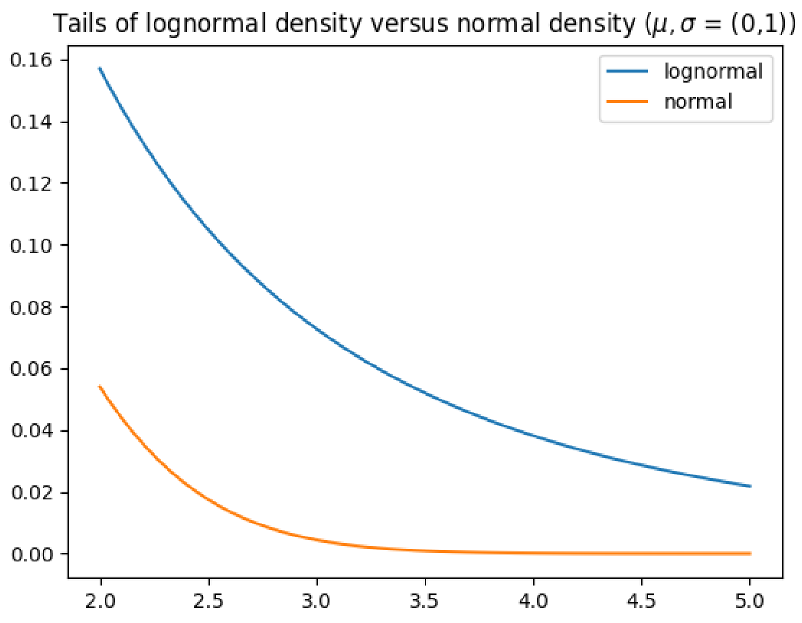
so that in particular[30] $E[X] = Se^{rT}$. In terms of the terminal distrubution of spot then, the positive skew and fat-tailedness of the lognormal distribution has the following interpretation:

1. It is more likely that the terminal value of spot is less than if it grew at the risk free rate.

[29]Recall that $P(X \approx x) \approx f_X(x)$ so that the mode as defined is indeed the most frequently attained value probabilistic-ally.

[30]Note one does not even have to do any computations to verify this since we price in the risk-neutral measure.


Density of log-normal variable
σ = 0.1
σ = 0.5
σ = 1.0


Tails of lognormal density versus normal density (μ, σ = (0,1))
lognormal
normal

2. The likelihood of the terminal value being very large is greater than that of a normal variable being very large.

Moreover, both phenomena are enhanced as we increase the volatility.

These facts also explain the following a priori strange conclusion of put-call parity. Indeed if the forward price of spot is currently 0 (i.e. $S = e^{rT}K$) then, by put-call parity, a call with exercise at time T has the same value as a put of same exercise. Now this seems counter intuitive because the former has unlimited upside and the latter doesn't. This is explained however by the above however; in particular it is much more likely that the spot has terminal value $< e^{rT}K$ than it is for it to take a large value therefore making it much more likely that the put pays off than the call. On the other hand, there is still a non-trivial probability that the spot is very large thus giving a non-small probability to the call paying out a large amount.

6.2 Vanilla Options as a function of volatility

We shall now use the above results to study the effect of volatility on vanilla options.

Let us begin with calls and puts. As we have just seen, increasing the vol increases the right skewness of the terminal distribution. This makes the put more likely to pay out, hence the price of a put should be increasing with vol. On the other hand, it also increases the likelihood of the terminal value being very large, in which case the call pays out a large amount. Therefore we would also expect the call price to increase with vol and this can indeed be confirmed from the solution formula (it is also consistent with put-call parity as the forward price is independent of vol).

If we were to make several plots of the call price as a function of spot, for different values of vol, we would find that for small values the price is quite convex and almost tangent to the graph of $(S - e^{-rT}K)_+$ (its value for 0 volatility) whilst for large values of vol the price curve is almost straight and tending to the graph of S_+ (its value for infinite volatility). We confirm this with the plots below.

6.2.1 Aside I: plotting

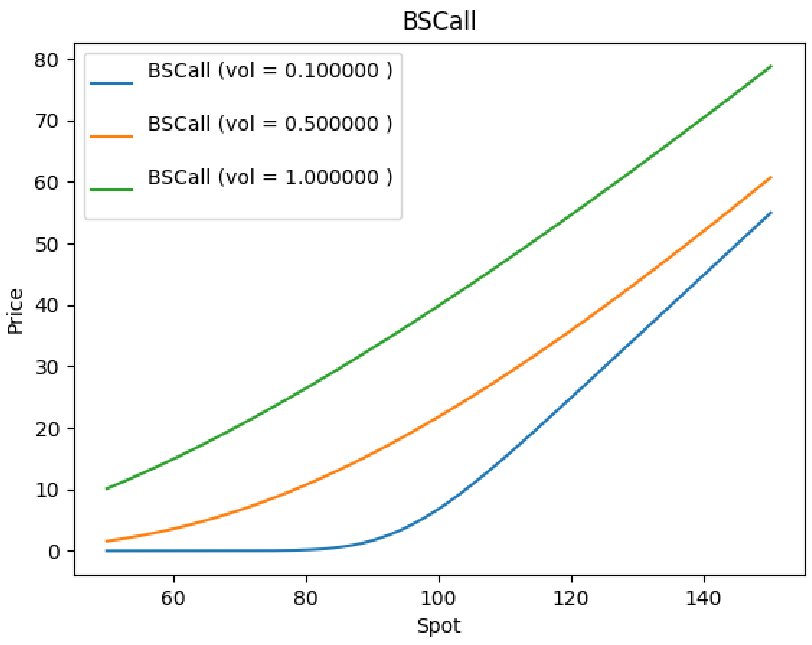
In order to produce the plots it made much more sense to output the plotting data to a text file and then read that into python and use matplot lib.

The act of outputting the graph done is done by a template function "OutputtingPlottingData". This has as template parameter a class f with function-like syntax (a functor) along with strings for the title, xlabel, and ylabel of the graph and also doubles for the upper and lower limits of the graphing domain. It then opens a file named "title.txt" and outputs data in the following format:

title
xlabel
ylabel
x data f(x) data

The reason for this format is that it can be easily read in python using readline etc. For the graph data, we use the loadtxt fct in the numpy module. Once the data has been read it is then easy to construct the desired plot. We wrap all this into a function, "Make2DPlot" which takes in a string representing the filename, which can be obtained by rightclicking on the text file and choosing the option to copy the file path[31] (which gives the full path name, including the directory).

[31] Another option would be to send the outputted C++ file straight to the directory where the python plotting script is held.



Note there is a subtlety in the formatting here. Indeed, most files will be saved in the directory "...\User...If one copies this into a string in python and runs it, one will encounter an error because python thinks that \U is starting an eight-character Unicode escape[32], such as \U00014321, and an escape followed by an 's' is invalid. Instead one either has to replace all backslashes by double blackslashes or prefix the string with r to produce a raw string[33]. Obviously the latter is more appealing as it is more templatised (in particular we don't know what string we'll be putting in so don't know what characters to replace). To handle this conversion we shall wrap up the plotting function inside another function which does the conversion[34].

We can also add some added functionality to allow for producing multiple graphs on the same plot. To handle this, we just add an extra parameter to the plotting function that takes in a bool representing if we are doing multiple plots or not. If false then we proceed as before whereas if true then we no longer use a ylabel and instead take in the title string as a label for the graph. In the case of multiple plotting, we then have a little routine which stores the file paths in a numpy array of strings and then loops over them whilst calling the plotting function. We also at this stage allow for the input of a title and ylabel for the entire plot. This wrapping routine also handles the conversion to raw literals.

Note from the view of software design the above plotting routine is far from ideal. It should however be sufficient for our purposes as we will not be plotting a great deal.

Remark 6.1. *When coding up everything I noticed the following annoying thing; I had a while loop for the plotting which read "while($x \leq$ Number)..." When I actually run it, I always missed the case*

[32]An escape character is a character representing whitespace, e.g. \n starts a new line.

[33]This just means all characters in the string are interpreted *literally*.

[34]Note the easiest way to get the path of a file on a windows computer is to hold down shift and then right click on the file. One of the options that comes up will then be to copy the file path.

when x ==Number. This confused me at first but ultimately I realised it was due to a rounding error. Indeed x was a double and each loop it was incremented by 0.1 which ultimately introduced some rounding errors so that at the last stage of the loop $x+0.1$ >Number. This business of rounding errors is something to keep in mind for the future!

Remark 6.2. *When writing the python script I realised something useful about the fact that the language is interpreted instead of compiled. Indeed instead of designing a proper wrapper function for the conversion of the strings I just did it using an if-else statement (to distinguish between multiple and single plots), using r then blank space wherever the strings should be, upon which I just copy-pasted the file path. Obviously, if I tried to this in a compiled language it wouldn't compile, regardless if it was only, say, the if statement that was to be used, because with any use I only properly filled out one of the conditional paths.*

6.2.2 Aside II: bind

When plotting the calls, I made good use of the bind function from the standard library functional package (introduced in C++11). This essentially allows to define new functors my freezing the parameters of another functor. For instance, suppose g is a functor which takes in 3 doubles for the operator overload of (). The aim is to define a new functor f whose operator overload of () takes in 2 doubles:

$$f(a,b) = g(a,b,c_0).$$

We do this using bind as follows:

auto f = std::bind(g, std::placeholder::_1, std::placeholder::_2, c_0)

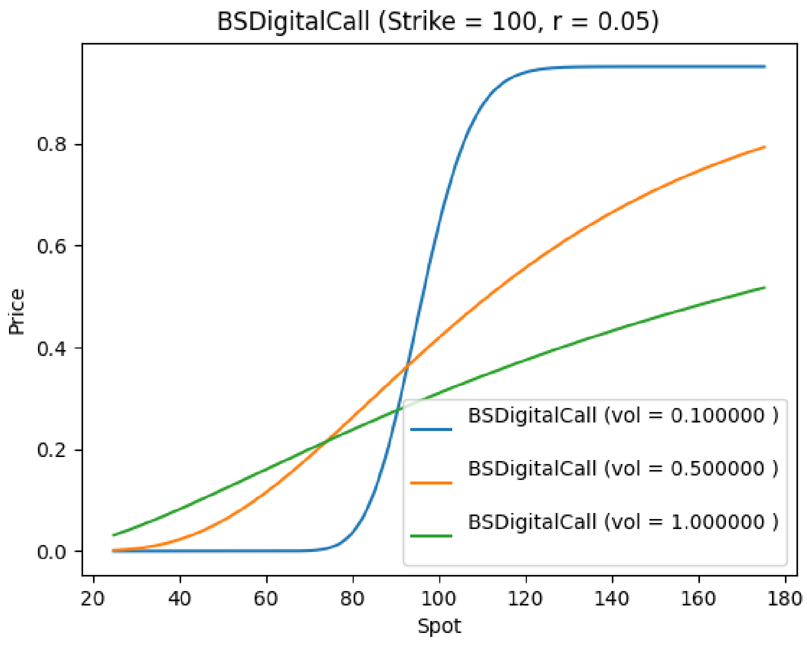
Obviously, this generalises to any number of frozen parameters and in any order. Essentially what happens is the placeholder objects are stored in the function object generated by bind, and when that object calls the operator overload of () with specific arguments the placeholders take on the argument values.

What's really cool is that bind() can be used on functions, not just functors. This is actually how I used it on the BSCall functions implemented earlier on (recall my plotting template took a functor as an argument). The great advantage of this is that its easy to generate a 1D functor but with variable other parameters (useful in this case because I want to view the BSCall as a function of spot, but also want to vary the vol in a discrete fashion). When I did this previously (i.e. prior to C++11) I had to first define a BSCall function class obtained by freezing the other parameters and then construct different objects of that class by varying the vol parameter. This is much more work! In fact, with bind() I think it is not necessary to ever have to define function classes like that.

Note one can do something similar with lambda expressions.

6.2.3 Back to plotting: digital calls

We turn now to digital calls and puts. This case is more interesting, and depends upon whether the options is in or out of the money. Indeed, if the option is in the money then increase the vol makes it more likely to end up out of the money. On the other hand, the upside of potentially larger values of spot has no effect on the option price since it is capped at 1. Therefore, in the money digital calls are decreasing in volatility. On the other hand, if the option is deep out of the money then the increased likelihood of the spot making a large move from increasing the vol means there



is a greater chance the option might end up in the money. Therefore for deep out the money digital calls, the price will be an increasing value of vol (at least for reasonable values – for very large values of vol, essentially all the mass will get pushed to the left). All this is borne out by an analysis of the solution formula (in fact, it is proportional to $P(S_T \geq K)$). The same analysis holds for the digital put but in reverse. We also have the following plots:

Note that our analysis of the call and digital call values as vol changes essentially illustrates the following fact about the terminal value of spot in the BS model:

1. As volatility increases the likelihood of the spot ending in the money decreases (because the distribution becomes more positively skewed).
2. On the other hand, as volatility increases the likelihood of the spot taking very large values increases (because the distribution has greater kurtosis).

6.3 When are American puts worth more than European puts?

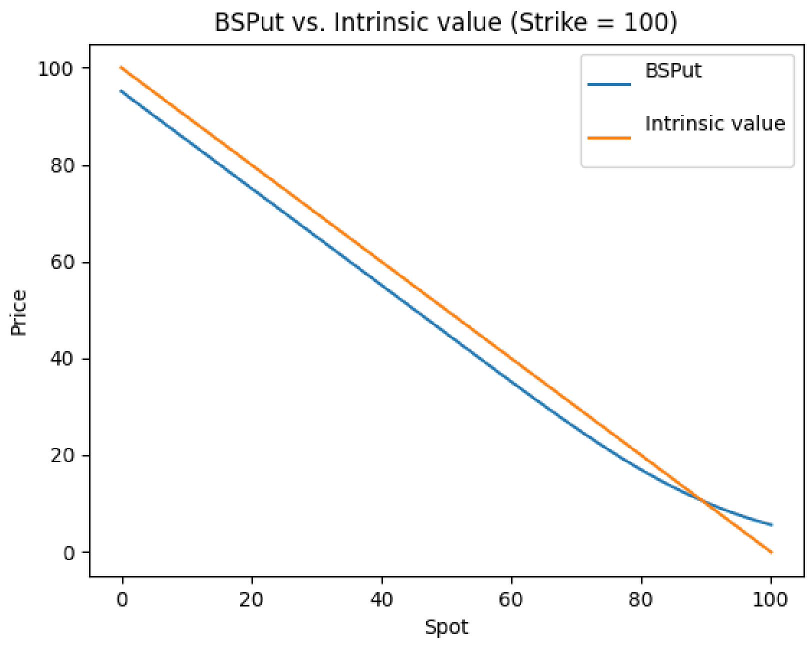
Recall the price of a European call option satisfies the following model independent lower bound:

$$C > S - KP_0$$

where P_0 is the price process of a zero coupon bond expiring at the exercise time of the option. Therefore assuming non-negative interest rates

$$C > C_{intrinsic} := S - K.$$

Since an American option of the same parameters is also worth more than its European counterpart, we conclude that under non-negative interest rates it is never optimal to early exercise an American option.



The same analysis however does NOT carry over to the case of a put option, because in that case the optimal lower bound is 0. It is therefore natural to ask under what conditions, if any, is it optimal to early exercise an American put. If we suppose we are in a BS world this is equivalent to asking if there exist parameters such that

$$BSPut(S, K, T) < K - S.$$

Indeed if it were never optimal to early exercise an American put then we would have $APut = EPut = BSPut$ in a BS world. On the other hand if the above inequality holds then there exists the following arbitrage opportunity:

1. Buy the American put for price $BSPut(S, K, T)$.
2. Immediately exercise and recieve $K - S$, thus obtaining a risk-free profit.

We now look to find such a bound. We fix reasonable parameters $(d, T, r, vol) = (0, 1, 0.05, 0.2)$ with a strike of 100 and plot the two graphs of the corresponding BSPut price and intrinsic value. We easily see that the former is less than the latter for all values of spot below approximately 90.

Note however that the corresponding American put with these parameters need not have current price equal to the intrinsic value. This is because the intrinsic value could be greater at a later time prior to expiry.

A Solving the Black–Scholes equation

The procedure to solve the BS equation proceeds by first transforming it to a linear heat equation, and then solving that by Fourier analysis. Now this only works in the constant coefficient case (i.e.

constant drift, vol and short rate) but it turns out that the solution is the same upon substituting in the effective vols and short rates (this is easily justified using risk-neutral pricing). So in what follows we assume constant coefficients, but will substitute the appropriate formula at the end.

A.1 Transformation to the linear heat equation

We suppose that $C(S,t)$ is a solution to the BS equation with terminal value $f(S)$. We claim that we can transform C to a solution of the linear heat equation.

We first remove the S coefficients. Define

$$S' = \log S$$

so that

$$\begin{aligned} S\partial_S &= \partial_{S'}, \\ S^2\partial_S^2 &= \partial_{S'}^2 - \partial_{S'}. \end{aligned}$$

Hence $A(S,t) = C(e^S,t)$ satisfies

$$\begin{aligned} rA(S,t) &= \partial_t A(S,t) + (r - \frac{1}{2}\sigma^2)\partial_S A(S,t) + \frac{1}{2}\sigma^2\partial_S^2 A(S,t), \quad t \in [0,T], \\ A(S,T) &= f(e^S). \end{aligned}$$

Next we write the first order terms as a single derivative. Define

$$\tau = t, \quad x = S - (r - \frac{1}{2}\sigma^2)t$$

so that

$$\partial_\tau = (r - \frac{1}{2}\sigma^2)\partial_S + \partial_t, \quad \partial_x = \partial_S.$$

Hence $A'(x,\tau) = A(x + (r - \frac{1}{2}\sigma^2)\tau, \tau)$ satisfies

$$\begin{aligned} rA'(x,\tau) &= \partial_\tau A'(x,\tau) + \frac{1}{2}\sigma^2\partial_x^2 A'(x,\tau), \quad \tau \in [0,T], \\ A'(x,T) &= f(e^{x+(r-\frac{1}{2}\sigma^2)T}). \end{aligned}$$

Finally, we remove the potential term and get the right sign on the time derivative by defining

$$\phi(x,s) = e^{-r(T-s)}A'(x,T-s)$$

which yields

$$\begin{aligned} 0 &= \partial_s\phi(x,s) - \frac{1}{2}\sigma^2\partial_x^2\phi(x,s), \quad s \in [0,T], \\ \phi(x,0) &= e^{-rT}f(e^{x+(r-\frac{1}{2}\sigma^2)T}). \end{aligned}$$

A.2 Solving the heat equation

We now solve the initial value problem

$$\begin{aligned} 0 &= \partial_s \phi(x,s) - \frac{1}{2}\sigma^2 \partial_x^2 \phi(x,s), \quad s \in [0,T], \\ \phi(x,0) &= g(x) \end{aligned}$$

using the Fourier transform.

Recall that if f is a L^1 function on the reals then its Fourier transform is defined according to

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} e^{2\pi i x \xi} f(x) dx.$$

With this definition we have the Fourier inversion formula

$$f(x) = \int_{-\infty}^{\infty} e^{-2\pi i x \xi} f(\xi) d\xi.$$

We recall also the following facts which are simple to prove:

- If f is suitably regular then
$$\mathcal{F}(f^{(n)})(\xi) = (2\pi i \xi)^n \hat{f}(\xi).$$
- If f and g are suitably regular then
$$\mathcal{F}(f * g)(\xi) = \hat{f}(\xi)\hat{g}(\xi)$$
which yields
$$(f * g)(x) = \mathcal{F}^{-1}(\hat{f}\hat{g})(x).$$
- For $f_\alpha = \exp(-\alpha x^2)$ with $\alpha > 0$ we have
$$\hat{f_\alpha}(\xi) = \sqrt{\frac{\pi}{\alpha}} \exp\left(- \frac{\pi^2 \xi^2}{\alpha} \right)$$

Using these facts, we may solve the heat equation as follows.

Assuming that ϕ is sufficiently regular at infinity we can take the Fourier transform of ϕ in the spatial variable to find

$$\begin{aligned} 0 &= \partial_s \hat{\phi}(\xi,s) + \frac{1}{2} 4\pi^2 \sigma^2 \xi^2 \hat{\phi}(\xi,s), \quad s \in [0,T], \\ \phi(\xi,0) &= \hat{g}(\xi). \end{aligned}$$

Therefore

$$\hat{\phi}(\xi,s) = \hat{g}(\xi) e^{-\frac{1}{2}4\pi^2\sigma^2\xi^2 s} = \sqrt{\frac{\pi}{\alpha_s}}^{-1} \hat{g}(\xi)\hat{f_{\alpha_s}}(\xi), \qquad \alpha_s := \frac{1}{2\sigma^2 s}$$

where f_α is defined as above. Thus

$$\begin{aligned} \phi(x,s) &= \sqrt{\frac{\pi}{\alpha_s}}^{-1} \int_{-\infty}^{\infty} g(x-y) e^{-\frac{y^2}{2\sigma^2 s}} \\ &= E[g(x + \sigma\sqrt{s}N(0,1)] \end{aligned}$$

where in the last line we use that the integral is invariant under $y \to -y$.

A.3 Transforming back to the Black–Scholes equation

We now invert the above series of transformations to get the desired solution of the Black–Scholes equation.

Indeed tracing back we find

$$\begin{aligned} C(S,t) &= e^{rt}\phi(\log S - (r - \frac{1}{2}\sigma^2)t, T-t) \\ &= e^{-r(T-t)}E\Big[f\Big(S\exp\Big((r-\frac{1}{2}\sigma^2)(T-t) + \sigma\sqrt{T-t}N(0,1)\Big)\Big)\Big] \end{aligned}$$

Remark A.1. *Solving the SDE for S we find*

$$S_{t_2} = S_{t_1}\exp\Big((r-\frac{1}{2}\sigma^2)(t_2-t_1) + \sigma\sqrt{t_2-t_1}N(0,1)\Big).$$

Thus we can write $C(S,t) = e^{-r(T-t)}E\Big[f(S_T|S_t)\Big]$.

Remark A.2. *We will see when we come to risk-neutral pricing that the same formula holds in the non-constant coefficient case provided we replace r and σ by the average short rate and effective volatility over the interval $[t,T]$.*