FRIDAY, MARCH 13, 2015

C++ Coding - Black Scholes Option Pricing - Binomial Trees

The example question for these solutions can be found on my website (click here).

5.1 Binomial Tree For Option Pricing

The two most popular models for using binomial trees to price options are

- Cox et al. (1979) (CRR for short) whose extra degree of freedom is to set

$$ud = 1$$

thus

$$u = e^{\sigma\sqrt{\Delta t}}, \quad d = e^{-\sigma\sqrt{\Delta t}}, \quad q = \frac{e^{r\Delta t} - d}{u - d}$$

- Rendleman and Bartter (1979) who choose:

$$q = \frac{1}{2}$$

and so

$$u = e^{(r-\frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}}, \quad d = e^{(r-\frac{1}{2}\sigma^2)\Delta t - \sigma\sqrt{\Delta t}}.$$

We wish to generate a stock price tree, so denote the value of the underlying asset after timestep i and upstate j by S_{ij} and we have that:

$$S_{ij} = S_0 u^j d^{i-j}$$

First download and start with this template code:

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

/* Template code for the binomial tree
 * Instructions to build the code are here
 * http://www.maths.manchester.ac.uk/~pjohnson/Tutorials/node1.html
 */
int main()
{
  // declare and initialise Black Scholes parameters

  // declare and initialise tree paramaters (steps in tree)

  // declare and initialise local variables (u,d,p)

  // create storage for the stock price tree and option price tree

  // setup and initialise the stock price tree

  // setup and initialise the final conditions on the option price tree

  // loop through time levels, setting the option price at each node in
 the tree

  // output the estimated option price

}
```

download

First declare and initialise the Black Scholes parameters for your chosen problem. Here we are going to value a Black Scholes vanilla European call option with, S_0 = 100, X =

ABOUT ME
Dr Johnson
View my complete profile

100, T = 1, r = 0.06 and σ = 0.2, so declare variables for each of these. Next add in an integer to store the number of steps in the tree and call it n. Finally add in some local variables to describe the tree, so we have the time step length dt, u, d and q. Your code should look like

```
// declare and initialise Black Scholes parameters
double S0=100.,X=100.,T=1.,r=0.06,sigma=0.2;
// declare and initialise tree paramaters (steps in tree)
int n=3;
// declare and initialise local variables (u,d,q)
double dt,u,d,q;
```

download

Then calculate the values for dt, u, d and q, using the appropriate formula, you should check the value of these against those found in my lecture notes (click here, slide 19).

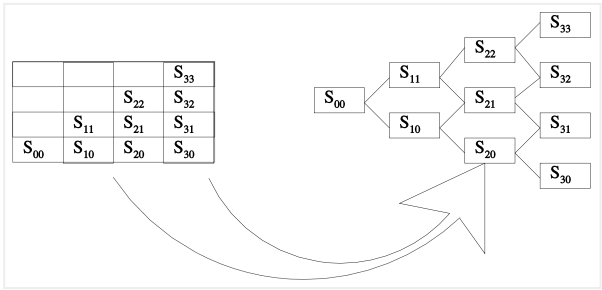
Next we need to create some storage for the values of the stock at each node in the tree. Declare a vector of vectors stockTree, into which we will place our stock price nodes.

```
// create storage for the stock price tree and option price tree
vector<vector<double>> stockTree(n+1,vector<double>(n+1));
```

download

where this has been intialised to an n + 1 × n + 1 2D array, and n is the number of nodes in the tree.

Now use a for loop and the function pow to input the value of the stock at each node in the tree, where $S_{i,j}$ →stockTree[i][j].



Now print out the vector stockTree[i][j] to the screen. Your code might look something like this

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;
/* Template code for the binomial tree
 * Instructions to build the code are here
 * http://www.maths.manchester.ac.uk/~pjohnson/Tutorials/node1.html
 */
int main()
{
  // declare and initialise Black Scholes parameters
   // declare and initialise Black Scholes parameters
  double S0=100.,X=100.,T=1.,r=0.06,sigma=0.2;
  // declare and initialise tree paramaters (steps in tree)
  int n=3;
  // declare and initialise local variables (u,d,q)
  double dt,u,d,q;
  dt = T/n;
  u = exp(sigma*sqrt(dt));
  d = exp(-sigma*sqrt(dt));
  q = (exp(r*dt)-d)/(u-d);
  // create storage for the stock price tree and option price tree
  vector<vector<double>> stockTree(n+1,vector<double>(n+1));
  // setup and initialise the stock price tree
  for(int i=0;i<=n;i++)
  {
    for(int j=0;j<=i;j++)
    {
      stockTree[i][j]=S0*pow(u,j)*pow(d,i-j);
      cout << i << " " << j << " " << stockTree[i][j] << endl;
    }
    cout << endl;
  }
/** OUTPUT
```

```
0 0 100
1 0 89.0947
1 1 112.24

2 0 79.3787
2 1 100
2 2 125.978

3 0 70.7222
3 1 89.0947
3 2 112.24
3 3 141.398
*/
}
```

download

Compare the values with those from the example in the lectures.

5.2 The Option Value Tree

Let us generate a simple example so that we can compare results at every stage to something that we can work out on paper. This is an important idea in debugging, to solve the problem and do all of your bug checking on a small scale before attempting the full problem.

First declare a vector of vectors which shall hold the values of the option

```
// create storage for the stock price tree and option price tree
vector<vector<double>> valueTree(n+1,vector<double>(n+1));
```

download

and set it to the same size as stockTree. Here we use the same relation $V_{i,j} \rightarrow$valueTree[i][j].

Now fill in the final values of the tree, given that we have already first generated the stock tree:

$$V_{n,j} = payoff(S_{n,j}),$$

where payoff is the appropriate function for the type of option we are solving for. For a European call option you should get

```
for(int j=0;j<=n;j++)
{
  valueTree[n][j]=max(stockTree[n][j]-X,0.);
  cout << n << " " << j << " " << valueTree[n][j] << endl;
}
```

download

Now we need to loop backwards through the tree to generate the value at each node using the equation:

$$V_{ij} = e^{-r\Delta t}(qV_{i+1,j+1} + (1-q)V_{i+1,j}).$$

You will need to do the following.

- Write for loops to move backwards through the vector array caculating the value of the option at each node.
- Print out the tree to screen (with n=3) and compare to the simple example (click here, slide 19) to check your code is working.
- If your values don't match - try to work out why!!
- Now print out the value at (0, 0) increasing the number of steps in the tree. Do the results look feasible? Compare them against the exact values from the formula.

Finally your code should look like this

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

/* Template code for the binomial tree
 * Instructions to build the code are here
 * http://www.maths.manchester.ac.uk/~pjohnson/Tutorials/node1.html
 */
int main()
```

```
{
  // declare and initialise S0,X,T,sigma,r,n
  double S0=100,X=100,T=1,sigma=0.2,r=0.05;
  int n=3;
  // declare and initialise local variables (u,d,q)
  double dt,u,d,q;
  dt = T/n;
  u = exp(sigma*sqrt(dt));
  d = exp(-sigma*sqrt(dt));
  q = (exp(r*dt)-d)/(u-d);
  // create storage for the stock price tree and option price tree
  vector<vector<double>> stockTree(n+1,vector<double>(n+1));
  // setup and initialise the stock price tree
  for(int i=0;i<=n;i++)
  {
    for(int j=0;j<=i;j++)
    {
      stockTree[i][j]=S0*pow(u,j)*pow(d,i-j);
    }
  }
  vector<vector<double>> valueTree(n+1,vector<double>(n+1));
  for(int j=0;j<=n;j++)
  {
    valueTree[n][j]=max(stockTree[n][j]-X,0.);
  }
  for(int i=n-1;i>=0;i--)
  {
    for(int j=0;j<=i;j++)
    {
      valueTree[i][j] = exp(-r*dt)*( q*valueTree[i+1][j+1] + (1-q)*valu
eTree[i+1][j]);
    }
  }
  cout << " V(S="<<S0<<",t=0) = " << valueTree[0][0] << endl;
/** OUTPUT
 V(S=100,t=0) = 11.552
 */
}
```

download

Finally, you could try to improve your code with the following

- Create a function returning the value of the binomial tree for a set of given parameters.
- Write a code storing two time-levels, and compare (at every stage if needed) with the previous code.
- Is it possible to store just one time-level? Try to write a code for this.
- Do you notice any difference (time taken for computation) between the codes with different storage requirements?

5.3 Evaluating Convergence Properties

Taking the code from last time, we can finish this off by taking the main algorithm out into a function and allowing the payoff function to be set independently. A representative code might look like the following:

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

/* Template code for the binomial tree
 * Instructions to build the code are here
 * http://www.maths.manchester.ac.uk/~pjohnson/Tutorials/node1.html
 */

// return the payoff of the function you want to evaluate
double payoff(double S,double X)
{
  return max(S-X,0.);
}

// return the value of the binomial tree
double binomialTree(double S0,double X,double T,double r,double sigma,i
nt n)
{
  // declare and initialise local variables (u,d,q)
  double dt,u,d,q;
```

```
  dt = T/n;
  u = exp(sigma*sqrt(dt));
  d = exp(-sigma*sqrt(dt));
  q = (exp(r*dt)-d)/(u-d);
  //create storage for the stock price tree and option price tree
  vector<vector<double>> stockTree(n+1,vector<double>(n+1));
  // setup and initialise the stock price tree
  for(int i=0;i<=n;i++)
  {
    for(int j=0;j<=i;j++)
    {
      stockTree[i][j]=S0*pow(u,j)*pow(d,i-j);
    }
  }
  vector<vector<double>> valueTree(n+1,vector<double>(n+1));
  for(int j=0;j<=n;j++)
  {
    valueTree[n][j]=payoff(stockTree[n][j],X);
  }
  for(int i=n-1;i>=0;i--)
  {
    for(int j=0;j<=i;j++)
    {
      valueTree[i][j] = exp(-r*dt)*( q*valueTree[i+1][j+1] + (1-q)*valu
eTree[i+1][j]);
    }
  }
  return valueTree[0][0];
}

int main()
{
  // declare and initialise Black Scholes parameters
   // declare and initialise Black Scholes parameters
  double S0=100.,X=100.,T=1.,r=0.06,sigma=0.2;
  // declare and initialise tree paramaters (steps in tree)
  int n=3;
  cout << " V(S="<<S0<<",t=0) = " << binomialTree(S0,X,T,r,sigma,n) <<
endl;
/** OUTPUT
 V(S=100,t=0) = 11.552
 */
}
```

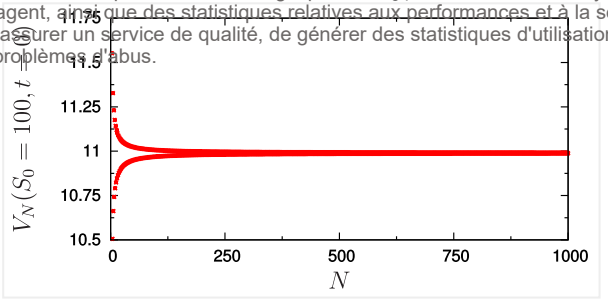
download

Now we wish to investigate what happens when N is increasing. Write the following loop in your code to output some results to file, you will need to adjust the output filename depending on you preference for where the file should be saved

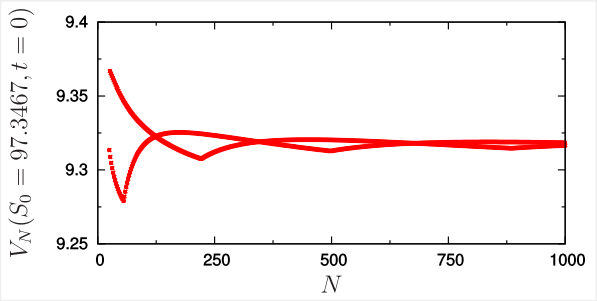
```
  // open output stream
  ofstream output("binomial-convergence.csv");
  // check it is open
  if(output.is_open())
  {
    cout << "File opened successfully" << endl;
    // output various n and V_n to file
    for(int n=3;n<=1000;n++)
    {
      output << n<<" , " << binomialTree(S0,X,T,r,sigma,n) << endl;
    }
    cout << "File write complete" << endl;
  }
  else
  {
    cout << "File could not be opened for some reason.\n";
  }
/** OUTPUT
 File opened successfully
 File write complete
 */
```

download

Run the code for S_0 = 100 and X = 100 and then import the data into a ploting package and visualise the results. You should see the following pattern



The pattern can be explained by realising that the nodes at terminal time will either be placed exactly on the strike price, or above/below depending on whether the number of steps is odd or even. We will see quite a different pattern if you choose $S_0 \neq X$, so for example set S_0 = 97.3467 in your code and rerun the results. After putting this in a plotting package you should see something like this



To produce this graph I have set n = 25 as the minimum grid size to show up the pattern better. This time we see humps as well as an odd-even effect. The humps are again caused by the positioning of the grid nodes relative to the strike price. Although the convergence of the tree is $O(1/n)$, the humps make it very difficult to achieve high accuracy. There are several papers that try to address this, including Leisen and Reimer (1996) and Heston and Zhou (2000), whilst Joshi (2007) gives a good comparison of different methods when American options are being priced.

References

Cox, John C, Stephen A Ross, and Mark Rubinstein. Option pricing: A simplified approach. Journal of financial Economics, 7(3):229–263, 1979.

Heston, Steve and Guofu Zhou. On the rate of convergence of discrete-time contingent claims. Mathematical Finance, 10(1):53–75, 2000.

Joshi, Mark S. The convergence of binomial trees for pricing the american put. Available at SSRN 1030143, 2007. URL http://fbe.unimelb.edu.au/__data/assets/pdf_file/0006/806280/170.pdf.

Leisen, Dietmar PJ and Matthias Reimer. Binomial models for option valuation-examining and improving convergence. Applied Mathematical Finance, 3(4): 319–346, 1996.

Rendleman, Richard J and Brit J Bartter. Two-state option pricing. The Journal of Finance, 34(5):1093–1110, 1979.

Posted by Dr Johnson at 1:20 PM

No comments:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Asian option pricing with C++ via Monte Carlo Methods

Asian option pricing with C++ via Monte Carlo Methods

By **QuantStart Team**

In this article I'm going to discuss how to price a certain type of *Exotic* option known as a **Path-Dependent Asian** in C++ using Monte Carlo Methods. It is considered "exotic" in the sense that the pay-off is a function of the underlying asset at multiple points throughout its lifetime, rather than just the value at expiry. An Asian option actually utilises the mean of the underlying asset price sampled at appropriate intervals as the basis for its pay-off, which is where the "path-dependency" of the asset comes from. The name actually arises because they were first devised in 1987 in Tokyo as options on crude oil futures.

There are two types of Asian option that we will be pricing. They are the *arithmetic Asian* and the *geometric Asian*. They differ only in how the mean of the underlying values is calculated. We will be studying discrete Asian options in this article, as opposed to the theoretical continuous Asian options. The first step will be to break down the components of the program and construct a set of classes that represent the various aspects of the pricer. Before that however, I will brief explain how Asian options work (and are priced by Monte Carlo).

Overview of Asian Options

An Asian option is a type of *exotic* option. Unlike a vanilla European option where the price of the option is dependent upon the price of the underlying asset at expiry, an Asian option pay-off is a function of multiple points up to and including the price at expiry. Thus it is *path-dependent* as the price relies on knowing how the underlying behaved at certain points *before* expiry. Asian options in particular base their price off the *mean average* price of these sampled points. To simplify this article we will consider N equally distributed sample points beginning at time $t = 0t = 0$ and ending at maturity, TT.

Unlike in the vanilla European option Monte Carlo case, where we only needed to generate multiple spot values at expiry, we now need to generate multiple spot *paths*, each sampled at the correct points. Thus instead of providing a `double` value representing spot to our option, we now need to provide a `std::vector<double>` (i.e. a vector of double values), each element of which represents a sample of the spot price on a particular path. We will still be modelling our asset price path via a Geometric Brownian Motion (GBM), and we will create each path by adding the correct drift and variance at each step in order to maintain the properties of GBM.

In order to calculate the arithmetic mean AA of the spot prices we use the following formula:

$$A(0,T) = \frac{1}{N}\sum_{i=1}^{N} S(t_i)$$

Similarly for the geometric mean:

$$A(0,T) = \exp\left(\frac{1}{N}\sum_{i=1}^{N} \log(S(t_i))\right)$$

$$A(0,T) = \exp\left(\frac{1}{N}\sum_{i=1}^{N} \log(S(t_i))\right)$$

These two formulae will then form the basis of our `pay_off_price` method, which is attached to our `AsianOption` class.

Asian Options - An Object Oriented Approach

In order to increase maintainability we will separate the components of the Asian options pricer. As mentioned in the class on option pay-off hierarchies we are able to create an *abstract base class* called `PayOff`, which defines an interface that all subsequent inherited pay-off classes will implement. The major benefit of this approach is that we can encapsulate multiple various types of pay-off functionality without the need to modify the remaining classes, such as our AsianOption class (to be discussed below). We make use of the `operator()` to turn our pay-off classes into a *functor* (i.e. a function object). This means that we can "call" the object just as we would a function. "Calling" a `PayOff` object has the effect of calculating the pay-off and returning it.

Here is the declaration for the `PayOff` base class:

```
class PayOff {
 public:
  PayOff(); // Default (no parameter) constructor
  virtual ~PayOff() {}; // Virtual destructor

  // Overloaded () operator, turns the PayOff into an abstract function object
  virtual double operator() (const double& S) const = 0;
};
```

The second class will represent many aspects of the exotic path-dependent Asian option. This class will be called `AsianOption`. It will once again be an abstract base class (which means that it contains at least one *pure virtual function*). Since there are many types of Asian option - arithmetic, geometric, continuous, discrete - we will once again make use of the inheritance hierarchy. In particular, we will override the `pay_off_price` function, which determines how the mean pay-off is to be calculated, once the appropriate `PayOff` object has been provided.

Here is the declaration for the `AsianOption` base class:

```
class AsianOption {
 protected:
  PayOff* pay_off;  // Pay-off class (in this instance call or put)

 public:
  AsianOption(PayOff* _pay_off);
  virtual ~AsianOption() {};

  // Pure virtual pay-off operator (this will determine arithmetic or geometric)
  virtual double pay_off_price(const std::vector<double>& spot_prices) const = 0;
};
```

In C++ there is always a trade-off between simplicity and extensibility. More often than not a program must become more complicated if it is to be extendable elsewhere. Thus we have the choice of creating a separate object to handle the path generation used by the Asian option or write it procedurally. In this instance I have elected to use a procedural approach because I don't feel that delving into the complexities of random number generation classes has a beneficial effect on learning how Asian options are priced *at this stage*. In later articles we will encapsulate the random number and path generation, but right now we will **keep it simple**.

Pay-Off Classes

The first class we will consider is the `PayOff`. As stated above this is an abstract base class and so can never be instantiated directly. Instead it provides an *interface* through which all inherited classes will be bound to. The destructor is `virtual` to avoid memory leaks when the inherited and base classes are destroyed. We'll take a look at the full listing for the `payoff.h` header file and then step through the important sections:

```
#ifndef __PAY_OFF_HPP
#define __PAY_OFF_HPP

#include <algorithm> // This is needed for the std::max comparison function, used in the pay-off calculations

class PayOff {
 public:
  PayOff(); // Default (no parameter) constructor
  virtual ~PayOff() {}; // Virtual destructor

  // Overloaded () operator, turns the PayOff into an abstract function object
  virtual double operator() (const double& S) const = 0;
};

class PayOffCall : public PayOff {
 private:
  double K; // Strike price

 public:
  PayOffCall(const double& K_);
  virtual ~PayOffCall() {};

  // Virtual function is now over-ridden (not pure-virtual anymore)
  virtual double operator() (const double& S) const;
};

class PayOffPut : public PayOff {
 private:
  double K; // Strike

 public:
  PayOffPut(const double& K_);
  virtual ~PayOffPut() {};
  virtual double operator() (const double& S) const;
};

#endif
```

The declaration for the `PayOff` class is straightforward except for the overload of `operator()`. The syntax says that this method cannot be implemented within this class (`= 0`) and that it should be overridden by an inherited class (`virtual`). It is also a `const` method as it does not modify anything. It simply takes a spot price S and returns the option pay-off, for a given strike, K:

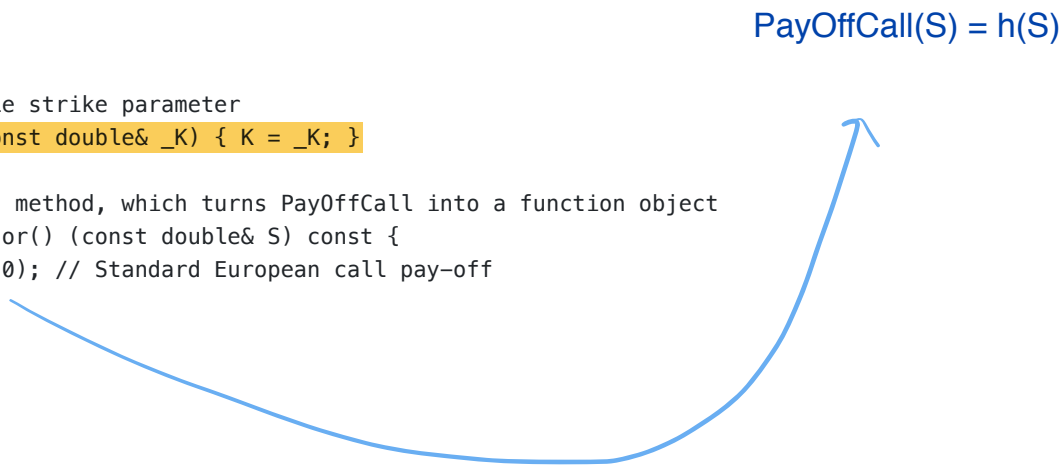
```
// Overloaded () operator, turns the PayOff into an abstract function object
virtual double operator() (const double& S) const = 0;
```

Beyond the `PayOff` class we implement the `PayOffCall` and `PayOffPut` classes which implement call and put functionality. We could also introduce digital and double digital pay-off classes here, but for the purposes of demonstrating Asian options, I will leave that as an exercise! Notice now that in each of the declarations of the `operator()` the `= 0` pure virtual declaration has been removed. This states that the methods should be implemented by these classes:

```
// Virtual function is now over-ridden (not pure-virtual anymore)
virtual double operator() (const double& S) const;
```

That sums up the header file for the `PayOff` class hierarchy. We will now look at the source file (below) and then step through the important sections:

```
#ifndef __PAY_OFF_CPP
#define __PAY_OFF_CPP

#include "payoff.h"

PayOff::PayOff() {}

// ==========
// PayOffCall
// ==========

// Constructor with single strike parameter
PayOffCall::PayOffCall(const double& _K) { K = _K; }

// Over-ridden operator() method, which turns PayOffCall into a function object
double PayOffCall::operator() (const double& S) const {
  return std::max(S-K, 0.0); // Standard European call pay-off
}

// =========
// PayOffPut
// =========

// Constructor with single strike parameter
PayOffPut::PayOffPut(const double& _K) {
  K = _K;
}

// Over-ridden operator() method, which turns PayOffPut into a function object
double PayOffPut::operator() (const double& S) const {
  return std::max(K-S, 0.0); // Standard European put pay-off
}

#endif
```



The only major point of note within the source file is the implementation of the call and put `operator()` methods. They make use of the `std::max` function found within the `<algorithm>` standard library:

```
// Over-ridden operator() method, which turns PayOffCall into a function object
double PayOffCall::operator() (const double& S) const {
  return std::max(S-K, 0.0); // Standard European call pay-off
}

..

// Over-ridden operator() method, which turns PayOffPut into a function object
double PayOffPut::operator() (const double& S) const {
  return std::max(K-S, 0.0); // Standard European put pay-off
}
```

The concludes the `PayOff` class hierarchy. It is quite straightforward and we're not using any real advanced features beyond abstract base classes and pure virtual functions. However, we have written quite a lot of code to encapsulate something seemingly as simple as a simple pay-off function. The benefits of such an approach will become clear later on, when or if we decide we wish to create more complex pay-offs as we will not need to modify our `AsianOption` class to do so.

Path Generation Header

To generate the paths necessary for an Asian option we will use a procedural approach. Instead of encapsulating the random number generator and path generator in a set of objects, we will make use of two functions, one of which generates the random Gaussian numbers (via the Box-Muller method) and the other which takes these numbers and generates sampled Geometric Brownian Motion asset paths for use in the `AsianOption` object. The article on European option pricing via Monte Carlo explains the concept of risk-neutral pricing, Monte Carlo techniques and the Box-Muller method. I have included the full function in the listing below for completeness.

The second function, `calc_path_spot_prices` is more relevant. It takes a reference to a vector of doubles (`std::vector<double>&`) and updates it to reflect a random spot path based on a set of random Gaussian increments of the asset price. I will show the full listing for the header file which contains these functions, then I will step through `calc_path_spot_prices` in detail:

```
#ifndef __PATH_GENERATION_H
#define __PATH_GENERATION_H

#include <vector>
#include <cmath>

// For random Gaussian generation
// Note that there are many ways of doing this, but we will
// be using the Box-Muller method for demonstration purposes
double gaussian_box_muller() {
  double x = 0.0;
  double y = 0.0;
  double euclid_sq = 0.0;

  do {
    x = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    y = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    euclid_sq = x*x + y*y;
  } while (euclid_sq >= 1.0);

  return x*sqrt(-2*log(euclid_sq)/euclid_sq);
}

// This provides a vector containing sampled points of a
// Geometric Brownian Motion stock price path
void calc_path_spot_prices(std::vector<double>& spot_prices,  // Vector of spot prices to be filled in
                           const double& r,   // Risk free interest rate (constant)
                           const double& v,   // Volatility of underlying (constant)
                           const double& T) { // Expiry
  // Since the drift and volatility of the asset are constant
  // we will precalculate as much as possible for maximum efficiency
  double dt = T/static_cast<double>(spot_prices.size());
  double drift = exp(dt*(r-0.5*v*v));
  double vol = sqrt(v*v*dt);

  for (int i=1; i<spot_prices.size(); i++) {
    double gauss_bm = gaussian_box_muller();
    spot_prices[i] = spot_prices[i-1] * drift * exp(vol*gauss_bm);
  }
}

#endif
```

Turning our attention to `calc_path_spot_prices` we can see that the function requires a vector of doubles, the risk-free rate rr, the volatility of the underlying $\sigma\sigma$ and the time at expiry, TT:

```
void calc_path_spot_prices(std::vector<double>& spot_prices,  // Vector of spot prices to be filled in
                           const double& r,   // Risk free interest rate (constant)
                           const double& v,   // Volatility of underlying (constant)
                           const double& T) { // Expiry
```

Since we are dealing with constant increments of time for our path sampling frequency we need to calculate this increment. These increments are always identical so in actual fact it can be pre-calculated outside of the loop for the spot price path generation. Similarly, via the properties of Geometric Brownian Motion, we know that we can increment the drift and variance of the asset in a manner which can be pre-computed. The only difference between this increment and the European case is that we are replacing TT with $dtdt$ for each subsequent increment of the path. See the European option pricing article for a comparison:

```
double dt = T/static_cast<double>(spot_prices.size());
double drift = exp(dt*(r-0.5*v*v));
double vol = sqrt(v*v*dt);
```

The final part of the function calculates the new spot prices by iterating over the `spot_price` vector and adding the drift and variance to each piece. We are using the arithmetic of logarithms here, and thus can *multiply* by our drift and variance terms, since it is the *log of the asset price* that is subject to normally distributed increments in Geometric Brownian Motion. Notice that the loop runs from $i = 1i = 1$, not $i = 0i = 0$. This is because the `spot_price` vector is already pre-filled with SS, the initial spot, elsewhere in the program:

```
for (int i=1; i<spot_prices.size(); i++) {
  double gauss_bm = gaussian_box_muller();
  spot_prices[i] = spot_prices[i-1] * drift * exp(vol*gauss_bm);
}
```

That concludes the path-generation header file. Now we'll take a look at the `AsianOption` classes themselves.

Asian Option Classes

The final component of our program (besides the `main` file of course!) is the Asian option inheritance hierarchy. We wish to price multiple types of Asian option, including geometric Asian options and arithmetic Asian options. One way to achieve this is to have separate methods on an `AsianOption` class. However this comes at the price of having to continually add more methods if we wish to make more granular changes to our `AsianOption` class. In a production environment this would become unwieldy. Instead we can use the abstract base class approach and generate an abstract `AsianOption` class with a pure virtual method for `pay_off_price`. This method is implemented in subclasses and determines how the averaging procedure for the asset prices over the asset path lifetime will occur. Two publicly-inherited subclasses `AsianOptionArithmetic` and `AsianOptionGeometric` implement this method.

Let's take a look at the full listing for the header declaration file and then we'll step through the interesting sections:

```
#ifndef __ASIAN_H
#define __ASIAN_H

#include <vector>
#include "payoff.h"

class AsianOption {
 protected:
  PayOff* pay_off;  // Pay-off class (in this instance call or put)

 public:
  AsianOption(PayOff* _pay_off);
  virtual ~AsianOption() {};

  // Pure virtual pay-off operator (this will determine arithmetic or geometric)
  virtual double pay_off_price(const std::vector<double>& spot_prices) const = 0;
};

class AsianOptionArithmetic : public AsianOption {
 public:
  AsianOptionArithmetic(PayOff* _pay_off);
  virtual ~AsianOptionArithmetic() {};

  // Override the pure virtual function to produce arithmetic Asian Options
  virtual double pay_off_price(const std::vector<double>& spot_prices) const;
};

class AsianOptionGeometric : public AsianOption {
 public:
  AsianOptionGeometric(PayOff* _pay_off);
  virtual ~AsianOptionGeometric() {};

  // Overide the pure virtual function to produce geometric Asian Options
  virtual double pay_off_price(const std::vector<double>& spot_prices) const;
};

#endif
```

The first thing to notice about the abstract `AsianOption` class is that it has a pointer to a `PayOff` class as a protected member:

```
protected:
 PayOff* pay_off;  // Pay-off class (in this instance call or put)
```

This is an example of *polymorphism*. The object will not know what type of `PayOff` class will be passed in. It could be a `PayOffCall` or a `PayOffPut`. Thus we can use a pointer to the `PayOff` abstract class to represent "storage" of this as-yet-unknown `PayOff` class. Note also that the constructor takes this as its only parameter:

```
public:
 AsianOption(PayOff* _pay_off);
```

Finally we have the declaration for the pure virtual `pay_off_price` method. This takes a vector of spot prices, but notice that it is passed as a reference to `const`, so this vector will not be modified within the method:

```
// Pure virtual pay-off operator (this will determine arithmetic or geometric)
virtual double pay_off_price(const std::vector<double>& spot_prices) const = 0;
```

The listings for the `AsianOptionArithmetic` and `AsianOptionGeometric` classes are analogous to those in the `PayOff` hierarchy, with the exception that their constructors take a pointer to a `PayOff` object. That concludes the header file.

The source file essentially implements the two `pay_off_price` methods for the inherited subclasses of `AsianOption`:

```
#ifndef __ASIAN_CPP
#define __ASIAN_CPP

#include <numeric>  // Necessary for std::accumulate
#include <cmath>  // For log/exp functions
#include "asian.h"

// ======================
// AsianOptionArithmetic
// ======================

AsianOption::AsianOption(PayOff* _pay_off) : pay_off(_pay_off) {}

// ======================
// AsianOptionArithmetic
// ======================

AsianOptionArithmetic::AsianOptionArithmetic(PayOff* _pay_off) : AsianOption(_pay_off) {}

// Arithmetic mean pay-off price
double AsianOptionArithmetic::pay_off_price(const std::vector& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double sum = std::accumulate(spot_prices.begin(), spot_prices.end(), 0);
  double arith_mean = sum / static_cast<double>(num_times);
  return (*pay_off)(arith_mean);
}

// ====================
// AsianOptionGeometric
// ====================

AsianOptionGeometric::AsianOptionGeometric(PayOff* _pay_off) : AsianOption(_pay_off) {}

// Geometric mean pay-off price
double AsianOptionGeometric::pay_off_price(const std::vector& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double log_sum = 0.0;
  for (int i=0; i<spot_prices.size(); i++) {
    log_sum += log(spot_prices[i]);
  }
  double geom_mean = exp(log_sum / static_cast<double>(num_times) );
  return (*pay_off)(geom_mean);
}

#endif
```

Let's take a look at the arithmetic option version. First of all we determine the number of sample points via the size of the `spot_price` vector. Then we use the `std::accumulate` algorithm and iterator syntax to sum the spot values in the vector. Finally we take the arithmetic mean of those values and use pointer dereferencing to call the `operator()` for the `PayOff` object. For this program it will provide a call or a put pay-off function for the average of the spot prices:

```
double AsianOptionArithmetic::pay_off_price(const std::vector& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double sum = std::accumulate(spot_prices.begin(), spot_prices.end(), 0);
  double arith_mean = sum / static_cast<double>(num_times);
  return (*pay_off)(arith_mean);
}
```

The geometric Asian is similar. We once again determine the number of spot prices. Then we loop over the spot prices, summing the logarithm of each of them and adding it to the grand total. Then we take the geometric mean of these values and finally use pointer dereferencing once again to determine the correct call/put pay-off value:

```
double AsianOptionGeometric::pay_off_price(const std::vector& spot_prices) const {
  unsigned num_times = spot_prices.size();
  double log_sum = 0.0;
  for (int i=0; i<spot_prices.size(); i++) {
    log_sum += log(spot_prices[i]);
  }
  double geom_mean = exp(log_sum / static_cast<double>(num_times) );
  return (*pay_off)(geom_mean);
}
```

Note here what the `AsianOption` classes *do not* require. Firstly, they don't require information about the underlying (i.e. vol). They also don't require time to expiry or the interest rate. Thus we are really trying to encapsulate the *term sheet* of the option in this object, i.e. all of the parameters that would appear on the contract when the option is made. However, for simplicity we have neglected to include the actual *sample times*, which would also be written on the contract. This instead is moved to the path-generator. However, later code will amend this, particularly as we can re-use the `AsianOption` objects in more sophisticated programs, where interest rates and volatility are subject to stochastic models.

The Main Program

The final part of the program is the `main.cpp` file. It brings all of the previous components together to produce an output for the option price based on some default parameters. The full listing is below:

```
#include <iostream>

#include "payoff.h"
#include "asian.h"
#include "path_generate.h"

int main(int argc, char **argv) {
  // First we create the parameter list
  // Note that you could easily modify this code to input the parameters
  // either from the command line or via a file
  unsigned num_sims = 100000;   // Number of simulated asset paths
  unsigned num_intervals = 250;  // Number of intervals for the asset path to be sampled
  double S = 30.0;  // Option price
  double K = 29.0;  // Strike price
  double r = 0.08;   // Risk-free rate (8%)
  double v = 0.3;    // Volatility of the underlying (30%)
  double T = 1.00;    // One year until expiry
  std::vector<double> spot_prices(num_intervals, S);  // The vector of spot prices

  // Create the PayOff objects
  PayOff* pay_off_call = new PayOffCall(K);

  // Create the AsianOption objects
  AsianOptionArithmetic asian(pay_off_call);

  // Update the spot price vector with correct
  // spot price paths at constant intervals
  double payoff_sum = 0.0;
  for (int i=0; i<num_sims; i++) {
    calc_path_spot_prices(spot_prices, r, v, T);
    payoff_sum += asian.pay_off_price(spot_prices);
  }
  double discount_payoff_avg = (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);

  delete pay_off_call;

  // Finally we output the parameters and prices
  std::cout << "Number of Paths: " << num_sims << std::endl;
  std::cout << "Number of Ints:  " << num_intervals << std::endl;
  std::cout << "Underlying:      " << S << std::endl;
  std::cout << "Strike:          " << K << std::endl;
  std::cout << "Risk-Free Rate:  " << r << std::endl;
  std::cout << "Volatility:      " << v << std::endl;
  std::cout << "Maturity:        " << T << std::endl;

  std::cout << "Asian Price:     " << discount_payoff_avg << std::endl;

  return 0;
}
```

The first interesting aspect of the `main.cpp` program is that we have now added a `unsigned num_intervals = 250;` line which determines how frequently the spot price will be sampled in the Asian option. As stated above, this would usually be incorporated into the option *term sheet*, but I have included it here instead to help make the pricer easier to understand without too much object communication overhead. We have also created the vector of spot prices, which are pre-filled with the default spot price, SS:

```
unsigned num_intervals = 250;  // Number of intervals for the asset path to be sampled
..
std::vector<double> spot_prices(num_intervals, S);  // The vector of spot prices
```

Then we create a `PayOffCall` object and assign it to a `PayOff` pointer. This allows us to leverage *polymorphism* and pass that object through to the `AsianOption`, without the option needing to know the actual type of `PayOff`. *Note that whenever we use the `new` operator, we must make use of the corresponding `delete` operator.*

```
PayOff* pay_off_call = new PayOffCall(K);
..
delete pay_off_call;
```

The next step is to create the `AsianOptionArithmetic` object. We could have as easily chosen the `AsianOptionGeometric` and the program would be trivial to modify to do so. It takes in the pointer to the `PayOff` as its lone constructor argument:

```
AsianOptionArithmetic asian(pay_off_call);
```

Then we create a loop for the total number of path simulations. In the loop we recalculate a new spot price path and then add that pay-off to a running sum of all pay-offs. The final step is to discount the average of this pay-off via the risk-free rate across the lifetime of the option ($T - 0 = TT - 0 = T$). This discounted price is then the final price of the option, with the above parameters:

```
double payoff_sum = 0.0;
for (int i=0; i<num_sims; i++) {
  calc_path_spot_prices(spot_prices, r, v, T);
  payoff_sum += asian.pay_off_price(spot_prices);
}
double discount_payoff_avg = (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
```

The final stage of the main program is to output the parameters and the options price:

```
Number of Paths: 100000
Number of Ints:  250
Underlying:      30
Strike:          29
Risk-Free Rate:  0.08
Volatility:      0.3
Maturity:        1
Asian Price:     2.85425
```

The benefits of such an object-oriented approach are now clear. We can easily add more `PayOff` or `AsianOption` classes without needing to extensively modify any of the remaining code. These objects are said to have a *separation of concerns*, which is exactly what is needed for large-scale software projects.

That is not to say that the program could not be improved! One obvious task is to determine how to incorporate the number of stock samples - and interval spacing - within the `AsianOption` hierarchy. Another is to encapsulate the random number and path generation into its own "engine" that can be used in other option pricers.

As always, if you have any difficulty following the above, please feel free to email me at **mike@quantstart.com**.

Successful Algorithmic Trading

How to find new trading strategy ideas and objectively assess them for your portfolio using a custom-built backtesting engine in Python.

Find Out More

Advanced Algorithmic Trading

How to implement advanced trading strategies using time series analysis, machine learning and Bayesian statistics with R and Python.

Find Out More