Developer Documentation for Two-Dimensional Black-Scholes Model Option Pricer

It prices options!

Luke Armitage

1 Introduction

This program is designed to the specification provided as part of the project brief. It is designed to compute prices of American and European style stock options with a variety of payoff functions. It uses a two-dimensional Black-Scholes model, calibrated with historical stock data, and a discrete-time approximation of a continuous-time, two dimensional binary tree. It can output, to a spreadsheet, stock prices with strike prices in a given range, for a range of given maturity dates.

DISCLAIMER: All code was written by me except for `Project.h`, and the code in `solver.h` (see section 2.7) is based on the bisection method nonlinear solver found in `Solver03.h` provided as part of the course material.

2 What's in the box?

The file `main.cpp` is comprised mostly of the function `main()`. This function starts out by greeting the user, and jumps in to producing calibration figures from historical stock data. It will attempt to open the file `data.csv`, and quit if that fails. If the file is successfully opened, the function will take the data from the `ifstream data` and store it in the `vector<vector<double> > S`, pushing each column of data into a column of `S`. `data.csv` is then closed, and the user is asked for the time interval of the historical data. Then the function calculates the calibration data from the historical data and displays it in the following format:

```
Calibration values:
   Historical Volatility of S0:  <value>
                            S1:  <value>
   Historical Correlation:       <value>
```

The function then gets from the user an interest rate and the current stock prices, and creates a `BSModel` object with the user's data and the calibration data. If the data is invalid for the model, the program will quit. If the model is valid, the function moves on to pricing options.

The user is asked to specify an expiry date and choose a payoff function, and then specify a lower and upper strike price. The function then follows the following procedure, first for a European style option:

1. Attempt to open the file `Eprices.csv` for writing. If that fails, the program exits.
2. Output the column headers to the file.
3. For each strike price in the given range, set the strike price for the payoff function, and output the strike price as a row title to the file.
4. For each expiry date in the given range, set up a `CorrBinModel` with that expiry date and then output to the file the option price at the given strike price.
5. Close the file and tell the user where they can find the results.

This is repeated for American options, writing to `Aprices.csv`. This takes a lot (around 10 times) longer due to the greater complexity of the pricing formula for American options. The program then prompts for a `char` to close it.

Now we run through all the header files associated with the project:

2.1 `binomial.h`

`binomial()` – calculates the binomial coefficent $\binom{n}{k}$. Takes two parameters: `int n` and `int k`. Returns a `double`.

2.2 `calibration.h`

This file contains statistical functions, used in calibrating the Black-Scholes model.

`logReturns()` – calculates the logarithmic difference between consecutive terms in a column of a dataset. Takes one parameter: `vector<double> _vector`. Returns a `vector<double>`.

`vecMean()` – calculates the mean of each column of a dataset. Takes one parameter: `vector<double> _vector`. Returns a `vector<double>`.

`sampleVar()` – calculates the sample variance each column of a dataset. Takes one parameter: `vector<double> _vector`. Returns a `vector<double>`.

`sampleCovar()` – calculates the sample covariance of a two-column dataset. Takes one parameter: `vector<double> _vector`. Returns a `double`.

2.3 payoff.h

This header file defines `SpreadCall`, `MinCall`, and `MaxCall` – three subclasses of `Payoff`, as defined in `Project.h`. The only difference between the three classes here is the definition of the member function `Value`. Each also has a data member `K`, which is the strike price of the option, and a corresponding accessor function `get_K()`.

To add new payoff functions, just copy the structure of the class definitions in `payoff.h`, with the name of the new payoff function as the class name, and have the function `Value()` return the value of the payoff. Then in `main.cpp`, add a declaration of the new class and expand the choices of both the `while` loop and the `if-else` sequence that follows, to include the new payoff.

2.4 pricer.h

`PriceEuropean()` – calculates a fair price for a European style option. Takes three parameters:

`CorrBinModel& model` – a reference to a `CorrBinModel` object.

`Payoff& payoff` – a reference to a `Payoff` object, with a member function `Value()` which returns the value of the payoff function of the stock price S at a given node (j_0, j_1) in the binary tree model.

`N` – the expiry date of the American option, the latest time at which it can be exercised.

Returns a `double`.

2.5 Project.h and Project.cpp

`Project.h` is the header file provided for the project, `Project.cpp` defines the member functions of classes declared in `project.h` which are not defined there. Also defines the constant `EPSILON`.

`BSModel2` – class modelling a two-dimensional Black-Scholes model. Members:

Constructor – assigns values to internal parameters.

`IsWellDefined()` – check function for validity of member variables – returns a `bool`.

`s0` – vector of initial stock prices.

`r` – interest rate.

`sigma` – vector with volatilities of each stock.

`rho` – correlation between the logarithmic returns of each stock.

Each member variable has a corresponding accessor function.

`CorrBinModel` – class modelling a two-dimensional correlated binary model. Members:

Constructor – calculates `q`, `alpha` and `beta`. Assigns values to internal parameters.

`S()` – calculates pair of stock prices at time step n, node (j_0, j_1) of two-dimensional binomial tree. Takes three parameters: `int n`, `int j0`, `int j1`. Returns `vector <double>`.

`Prob()` – calculates the risk-neutral probability at time step n, node (j_0, j_1) of two-dimensional binomial tree. Takes three parameters: `int n`, `int j0`, `int j1`. Returns `double`.

`IsArbitrageFree()` – check function for validity of member variable `q` – returns a `bool`.

`s0` – vector of initial stock prices.

`r` – interest rate.

`h` – time step size of binary tree model.

`alpha` – vector of parameters used in approximating prices.

`beta` – vector of parameters used in approximating prices.

`q` – vector with probabilities used for risk-neutral probability.

Each of `q` `r` and `h` have a corresponding accessor function.

`Payoff` – class modelling a payoff function for an option. Has one member function, `Value()`, which is a pure virtual function.

Also defined in this header is the function `PriceAmerican()` which is broken down in section 3, and a function which allows vectors to be displayed easily using familiar `std::cout` methods.

2.6 `projectIO.h`

This header defines the following functions for getting user input, with overloaded versions of each (taking one parameter) to let a value to be set inside code rather than through the console. To use these, instead of calling `getTime()` which asks for user input, call `getTime(`<*value*>`)` to return <*value*> and print it to console.

- `getTime()`
- `getRate()`
- `getStock()`
- `getStrike()`

This header also defines a function `stod()` which takes a parameter of type `string` and returns a value of type `double` (this is necessary because for some reason the `std::stod` from the standard library didn't work for me). I also tried to make a function `ifstreamToVector` to convert a data stream to a matrix, but it wouldn't quite work. Left that in there for posterity.

2.7 `solver.h`

This file contains a nonlinear solver by the bisection method, and a class with very detail specific data members required to solve the equation for `q[1]` in `Project.cpp`.

SolveByBisect() – finds a solution x to the equation $f(x) = c$. Takes five parameters: `F* Function` (of templated type, corresponds to f), `double target` (corresponds to c, `leftEnd` and `rightEnd`, the endpoints of the interval that x is in, and `Acc`, how close we need to get to x. Returns `double`.

3 Analysis of `PriceAmerican` function

Part of the project brief is to analyse the function `PriceAmerican`. It calculates a fair price for an American style option with a given expiry time, based on a given correlated binary model and payoff function.

```
5 inline double PriceAmerican (const CorrBinModel& model,
                               const Payoff& payoff,
                               int N)
```

This is the function declaration. The function takes three arguments:

`model` – a reference to a `CorrBinModel` object.

`payoff` – a reference to a `Payoff` object, with a member function `Value()` which returns the value of the payoff function of the stock price S at a given node (j_0, j_1) in the binary tree model.

`N` – the expiry date of the American option, the latest time at which it can be exercised.

A function declared with the keyword `inline` suggests to the compiler that it may, if the function is "small" enough, insert the definition of the function 'in-line', in place of the function call. This can speed up the program by reducing the number of times it needs to jump to the function definition and back, but at the cost of compile time and an increase in the size of the executable file. The keyword `const` means that the function should only access data members of each object without changing them – for example to call the member function `Get_q()` of the object `model`.

Now for the function definition:

```
{
    vector<vector<double> > v,pv;
10  vector<double> q = model.Get_q();
    double d = exp(-model.Get_r()*model.Get_h()),
           q00d = d*(1-q[0])*(1-q[1]),
           q01d = d*(1-q[0])*q[1],
           q10d = d*q[0]*(1-q[1]),
15         q11d = d*q[0]*q[1];
    double ev,cv;
    v.resize(N+1);
```

The variable `d` corresponds to the discount term in the American option model. The variable `q11d` corresponds to the discount term multiplied by the probability of traversing from node (j_0, j_1) at time step n to node (j_0+1, j_1+1) at time step $n+1$

– similarly for `q00d`, `q01d` and `q10d`. Lines 5 and 12 define variables used later and line 13 gives `v` room to store all $N+1$ vectors of prices we'll be using.

```
    for(int j0=0; j0<=N; j0++)
    {
20      v[j0].resize(N+1);
        for(int j1=0; j1<=N; j1++)
            v[j0][j1] = payoff.Value(model.S(N,j0,j1));
    }
```

This nested `for` loop works its way through each vector in `v`, resizes it to $N+1$ elements and then sets each element `v[j0][j1]` to the payoff at the node (j_0, j_1) at time step N. Then `v` becomes an $N+1$-by-$N+1$ 'matrix', holding all the payoff values at the expiry time.

```
    for(int n=N-1; n>=0; n--)
25  {
        pv=v;
        for(int j0=0;j0<=n;j0++)
            for(int j1=0; j1 <= n; j1++)
            {
30              ev = payoff.Value(model.S(n,j0,j1));
                cv = q00d*pv[j0][j1]
                   + q01d*pv[j0][j1+1]
                   + q10d*pv[j0+1][j1]
                   + q11d*pv[j0+1][j1+1];
35              v[j0][j1] = (ev>cv)?ev:cv;
            }
    }
    return v[0][0];
};
```

The `for` loop on line 20 means we're working our way backwards from the expiry date, so at each time step we are considering the first $n+1$ rows and $n+1$ columns of `v` – and at the $n=0$ step, we are considering the single value `v[0][0]`, which is then returned by the function after the loop ends.

The assignment `pv=v` stores in `pv` the state of `v` at the beginning of each run through the loop, which is necessary because we will be changing the values during each run depending on the state of `v` after the previous cycle.

The nested `for` loop runs through the $n+1$-by-$n+1$ submatrix of `v`, following the same procedure at each node (j_0, j_1) (for $j_0, j_1 = 1, \ldots, n$). First, `ev` stores the payoff for exercising at the node (j_0, j_1) and time n, and `cv` stores the potential future return. This is calculated by summing, over the four successor nodes, the probability of moving to that node multiplied by the payoff corresponding to that node – which is stored in `pv`. The statement `v[j0][j1] = (ev>cv)?ev:cv` means that the program will compare the values of `ev` and `cv`, and store the greater of the two in `v[j0][j1]`.

In this way, the function calculates the payoffs for all nodes at each time step n and then returns the value it calculates for the first node $(0, 0)$ at time 0 – the fair

price of the American option.