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

It prices options!

Luke Armitage

introduce software.
list and details of headers, classes, functions and variables.
detail how to add new payoff functions.
detail nonlinear solver method used in 3.
report on test runs of code.

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

clever things i've done:
overloading of 'get' functions, so you don't have to enter the same values again and again to test.

1 Analysis of `PriceAmerican` function

Part of the project brief is to analyse the function `PriceAmerican`. Here's the code from `Project.h`, interspersed with my comments.

```
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;
    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]),
           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++)
    {
        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--)
    {
        pv=v;
        for(int j0=0;j0<=n;j0++)
            for(int j1=0; j1 <= n; j1++)
            {
                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];
                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.