CQF Project

Christian de Chenu

July 2013

QuickStart Guide

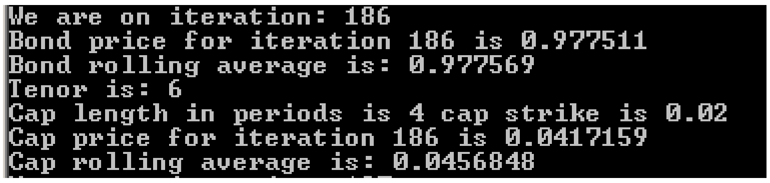
HJM Model

1. The folder HJM Model contains two folders. One contains the C++ code and the other the executable. **Start by moving everything to a writable disk like a Hard Drive. This will allow the exe file to output data.** The folder containing the executable is called HJM Model exe. Begin by opening this folder.

2. In this folder is five files:

- *csvdata.csv*: This file contains instantaneous forward rates from bank of England.
- *Curve Fitting.xlsm*: This file displays graphs output by the HJM Model program. It reads this data from the file fitteddata.csv which is output during runtime.
- *HJM Model.exe*: This is the actual executable.
- *Test Comparison.xlsm* This is a modified version of the model given in class that allows us to compare results
- *HJM.cpp:* Source code. This file is also contained in the project folder.

3. Click on HJM Model.exe. This will start the program running which will then automatically read in the *csv data.csv* file. This will then output 4 files, *differences.csv, covariance matrix.csv, vols.csv* and finally *fitteddata.csv*. Each of those files should be self-explanatory, however it is worth noting that the best way to view the *fitteddata* file is to first open it in excel and then open the *Curve Fitting* file which loads its data from *fitteddata*.
4. As the program runs you will see the following ouput which updates in real time:



```
We are on iteration: 186
Bond price for iteration 186 is 0.977511
Bond rolling average is: 0.977569
Tenor is: 6
Cap length in periods is 4 cap strike is 0.02
Cap price for iteration 186 is 0.0417159
Cap rolling average is: 0.0456848
```

Line 1: The iteration that is currently being processed
Line 2: The Bond price calculated for the given iteration
Line 3: As the program completes an iteration it calculates a rolling average of the average bond price so far. This will gradually converge until the program terminates after the given number of iterations (default 1000)
Line 4: Gives the Tenor in months of the bond. Default is 6 months. Note that this also serves to be the period length for the cap.
Line 5: Gives the cap length in period as well as the strike. In the image above the cap is 4 periods long where each period is of 6 month (the tenor of the bond). Thus this cap is 2 years long. The strike is 0.02 or 2%.
Line 6: The cap price calculated for the given iteration
Line 7: As the program completes an iteration it calculates a rolling average of the cap price so far. This will gradually converge until the program terminates after the given number of iterations (default 1000)

QuickStart Guide

Uncertain Volatility and Static Hedge

1. The folder Uncertain Volatility contains two files.
   Finite Difference Project: This is the actual project containing the relevant VBA code
   Graph Version: This is a modified version which I have used to output a graph of results. This is included simply for completeness.

2. Cells B4 and C4 contain the Black and Scholes market prices of the calls.

3. Cells C14 to C27 contain the inputs and the layout is designed similar to the ones given in class. Note that Vol- represents the lower bound on the volatility and Vol+ represents the upper bound. There is also inputs for expirations of all three options as well as quantities on the 2 calls.

4. Pricing is automatic once new inputs are given. However if you want to solve to maximise (or minimise) the binary use the excel solver which will already be set up with correct parameters.

Note when reading this report:

I have included much of my code. The easiest way to read the code it to scan over the comments in green which should clarify what each of the routines are doing.

Topic 1: HJM Model

Background:

This project implements the Heath, Jarrow & Morton approach to the modelling the curve. The model is innovative as it considers the whole of the forward rate curve in the pricing of fixed-income products.

The basic idea is simple: Instead of modelling a short-term interest rate and deriving the forward rates from there, they model the whole forward rate curve. The core insight with it is the recognition that there is an explicit relationship between the drift and volatility parameters of the forward rates in a no-arbitrage world.

Forward Rates:

We write F(t; T) for the forward rate curve at time t. Thus the price of a zero-coupon bond at time t and maturing at time T, when it pays $1, is

$$Z(t;T) = e^{-\int_t^T F(t;s)ds}$$

Equation 1

We now assume that all zero-coupon bonds evolve according to

$$dZ(t;T) = \mu(t,T)Z(t;T)\,dt + \sigma(t,T)Z(t;T)\,dX$$

Equation 2

ie it is a one factor model as there is one source of randomness driving the dynamics of the underlying process. From equation 1 we have

$$F(t;T) = -\frac{\partial}{\partial T}\log Z(t;T)$$

Equation 3

We then differentiate this with respect to t and substituting from Eqn. 2 results in an equation for the evolution of the forward curve:

$$dF(t;T) = \frac{\partial}{\partial T}\left(\tfrac{1}{2}\sigma^2(t,T) - \mu(t,T)\right)dt - \frac{\partial}{\partial T}\sigma(t,T)\,dX$$

Equation 4

Our ZCB in Eqn 1 represents the curve in its initial state, which will then evolve as per equation 2. We now have the drift of the forward rates

The Spot Rate:

First we can define the spot rate as the forward rate for a maturity equal to the current date i.e.

$$r(t)=F(t;t)$$

Suppose we are currently at t* and we know the whole forward rate curve today F(t*;T), then we can write the spot rate for any time *t* in the future as

$$r(t) = F(t;t) = F(t^*;t) + \int_{t^*}^{t} dF(s;t).$$

From equation 4 we have

$$r(t) = F(t^*;t) + \int_{t^*}^{t} \left(\sigma(s,t)\frac{\partial \sigma(s,t)}{\partial t} - \frac{\partial \mu(s,t)}{\partial t} \right) ds - \int_{t^*}^{t} \frac{\partial \sigma(s,t)}{\partial t} dX(s).$$

Differentiating w.r.t. time will provide us at the stochastic differential equation for *r*.

$$dr = \left(\frac{\partial F(t^*;t)}{\partial t} - \frac{\partial \mu(t,s)}{\partial s} |_{s=t} \right.$$

$$+ \int_{t^*}^{t} \left(\sigma(s,t)\frac{\partial^2 \sigma(s,t)}{\partial t^2} + \left(\frac{\partial \sigma(s,t)}{\partial t} \right)^2 - \frac{\partial^2 \mu(s,t)}{\partial t^2} \right) ds$$

$$\left. - \underline{\int_{t^*}^{t} \frac{\partial^2 \sigma(s,t)}{\partial t^2} dX(s)} \right) dt - \frac{\partial \sigma(t,s)}{\partial s} |_{s=t} dX.$$

Relationship between the risk-neutral forward rate drift and volatility

If we write the stochastic differential equation for the risk-neutral forward curve as

$$dF(t;T) = m(t,T)dt + \nu(t,T)dX.$$

This implies from equation 4

$$\nu(t,T) = -\frac{\partial}{\partial T}\sigma(t,T)$$

Is the forward rate volatility.

From the same equation, the drift of the forward rate is given by

$$\frac{\partial}{\partial T}\left(\tfrac{1}{2}\sigma^2(t,T) - \mu(t,T)\right) = \nu(t,T)\int_t^T \nu(t,s)ds - \frac{\partial}{\partial T}\mu(t,T),$$

In the risk neutral world we have μ(t,T)=r(t) and so the drift of the risk-neutral forward rate curve is related to its volatility by

$$m(t,T) = \nu(t,T)\int_t^T \nu(t,s)ds.$$

Pricing Derivatives.

The details of this will be explained in the implementation section, however If we want to use a Monte Carlo method, then we must simulate the evolution of the whole forward rate curve, calculate the value of all cashflows under each evolution and then calculate the present value of these cashflows by discounting at the realized spot rate r(t).

Outline of model:

We apply principal component analysis to identify the factors that are driving changes in the forward curve. Rather than decomposing the rates themselves we instead calculate the covariance of changes because what we are actually modelling is changes in the rates.

We model a stochastic process

$$\mathbf{F}(\mathbf{t}) = (F_1(t), \cdots, F_n(t))$$

With a system of SDEs

$$\begin{aligned} dF_1(t) &= m_1(t)dt + \sigma_1(t)dW_1 \\ \vdots \quad & \quad \vdots \\ dF_n(t) &= m_n(t)dt + \sigma_n(t)dW_n \end{aligned}$$

m and σ represent drift and vol. The Ws are an n dimensional Brownian Motion with correlation structure represented by a covariance matrix.

We then decompose this into eigenvalues and eigenvectors. Because the covariance matrix is symmetric we can decompose it as

$$\boldsymbol{\Sigma} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}'$$

The reason we do this is that Eigenvectors are orthogonal to the covariance matrix and thus represent independent factors. We then take the largest of these as these represent the main driving factors.

The first entry in a particular eigenvector represents the movement of the first maturity point that we have (such as 1 month), the second would be the second maturity point (such as 6month) and so on. Thus each eigenvector thus represents the change in the whole curve. The Eigenvalue represents the variance.
We then choose the k largest factors based on their eigenvalues and transform our equations into

$$\begin{aligned} dF_1(t) &= m_1(t)dt + \sum_{i=1}^{k} \sqrt{\lambda_i} v_{i,1} dX_i \\ \vdots \quad & \quad \vdots \qquad \vdots \\ dF_n(t) &= m_n(t)dt + \sum_{i=1}^{k} \sqrt{\lambda_i} v_{i,n} dX_i \end{aligned}$$

After we have reduced the dimensions, the dX become Principal Components. We can calculate the amount of variance covered by each component by calculating the value of each lambda as a proportion of the sum of all the lambdas. We can then add up the individual contributions the lambdas make to something akin to an R-squared statistic. In some cases your 1st component could make up more than 90% of the variance.

We will also need the risk neutral drift which requires integration. We only know values for discrete points on each of the principal components so we will have to engage in curve-fitting. I have done this using a polynomial regression which I have detailed in the implementation section.

We can then move onto performing the actual Monte-Carlo simulation. We use the last row of the known observations as the first row to initialise the Monte-Carlo simulation. In order to calculate the rate for each cell we use the multi-factor Musiela Paramaterization SDE discretised by the Euler method as

$$\nu_i dX_i = \nu_i \phi_i \sqrt{(\delta t)}.$$

Musiela Parameterization also requires numerical integration over fitted vol functions in order to calculate the drift.

Once we estimated all parameters for the risk-neutral SDE, we can start pricing interest rate derivatives by averaging cashflows across multiple simulations.

Implementation:

I have chosen to implement the model using the language C++. There are a number of reasons for this:

1. I have not had the opportunity to gain experience in this language in my job and thought this was a perfect opportunity for a learning experience.
2. C++ is still widely used in quant finance.
3. C++ is thought to be highly flexible so once I got past some initial teething problems it should be possible for me to have the program interact with a range of libraries and external programs if needed.
4. C++ is faster than VBA.

Step one: Reading in the time series data

The first thing I need to do is feed the instantaneous forward rates we acquire from the bank of England website. I chose to use the ifstream class to read in data. Similar to the isstream class and having many similar methods it allows reading in of files (hence the "f"). I also make sure it is in a simple readable format for C++ such as CSV.

```
ifstream infile("csvdata.csv");
```

Step two: Put the data into vectors.

For this part I use the C++ class "vector" to store the individual forward rates. Each vector represents a single row of data i.e. data from one day in different tenors. We also have a Vector of Vectors containing each of these rows which thus functions as a matrix.

I have detailed the process of putting the data into vectors below. The getline function in the while loop continues to read in lines as long as there are more in the file. **This method of doing it means I am not restricting myself to a specific size for the matrix**.

It is also worth noting that the Vector class was specifically chosen as you do not need to specify its size on its construction. You can simply push new data into it and grow it organically.

The nested while loop breaks each line up into its constituent "words" which are forward rates. These are then converted into doubles. Again we are not restricted on the number of tenors contained in each row.

```
while (getline(infile, line)){ //while there are lines in the file, get the line
 stringstream strstr(line); //put the line in a stringstream
 string word = ""; //make a blank word
 double number;//make a new number
 vector<double> row;//make a vector row to put the line into

 while (getline(strstr,word, ',')){
 //put the word separated by ',' into string "word"
       number = string_to_double(word);//convert the "word" to a double
       row.push_back(number);//put the new number from the line into the row
 }
              data.push_back(row);//put the row into the whole matrix
}
```

Step three: Convert the interest rates into differences

For this we create a new Vector structure again consisting of a Vector of Vectors. We assess the size of the matrix from step two. We then loop over this matrix calculating differences in the individual forward rates.

```
vector< vector<double> > differences;

for(int i=0; i<num_rows-1; i++){
//Loop over only go to rows-1 since because its differences we have 1 less

  vector<double> row;//make a row to put the row into

  for(int j=0; j<num_columns; j++){
    if (i==0||j==0){
    //if we are on the 0'th row or column,
    then just put in the data directly as it is header info
    row.push_back(data[i][j]);//put the data in without change
    }else{
    row.push_back(data[i+1][j]-data[i][j]);//put the differences of the data in
    }
  }
    differences.push_back(row);//put the row into the whole matrix
}
```

```
vector< vector<double> > covariancematrix;
vector<double> x;
vector<double> y;
covariancematrix.resize(num_columns);
for(int m=0; m<num_columns; m++){
  covariancematrix[m].resize(num_columns);
}
x.resize(num_rows-2);
y.resize(num_rows-2);

for(int i=0; i<num_columns; i++){ //for each column of the differences matrix

  for(int j=1; j<num_rows-1; j++){//only from 1 onwards as we dont want to
header info
    x[j-1]=differences[j][i];//first construct the x column to compare to y
  }

  for(int h=0; h<num_columns; h++){
    for(int j=1; j<num_rows-1; j++){//only from 1 onwards as we dont want to
header info
    y[j-1]=differences[j][h];//then construct the y column we want to compare x
to.
  }
  if(h==0||i==0){ //so if we are in the edge LHS of the matrix for either
comparison vector then its just header information.
    if(i==0){
      covariancematrix[i][h]=differences[i][h];//then just make it equal the
data
    }else{
      covariancematrix[i][h]=differences[h][i];//except in the case of the
covariance matrix we want the side info the same as the top header info
hence[h][i] rather than [i][h]
    }
  }else if(h<i){
    covariancematrix[i][h]=covariancematrix[h][i];
  }else{
    //calculate covariance
    double xmean;
    double sumx = 0;
    double ymean;
    double sumy = 0;
    for(unsigned p = 0; p < x.size(); p++){
    sumx += x[p];
    sumy += y[p];
  }
  xmean=sumx/x.size();
  ymean=sumy/y.size();
  double total = 0;

  for(int k = 0; k < x.size(); k++){
    total += (x[k] - xmean) * (y[k] - ymean);
  }
    covariancematrix[i][h]=((total/x.size())* 252 / 10000);
  }
 }
}
```

Step Five: Output covariance to a file

Finally we output the covariance matrix to an excel readable file

```
myfile.open ("covariance matrix.csv");
       for(int i=0; i<num_columns; i++){
         //only go to rows-1 since its differences we have 1 less
         for(int j=0; j<num_columns; j++){
           myfile <<covariancematrix[i][j] <<",";
         }
         myfile << "\n";
       }
       myfile.close();
```

Step Six: Eigenvalues and Eigenvectors

After we have produced our covariance matrix we decompose it into eigenvalues and eigenvectors (see background for more details on this). For this I have used a simple and useful technique for PCA: The Power Method.

The Power Method:

1. We assume all the eigenvalues are distinct which is a reasonable assumption given the nature of the data. The eigenvector associated with the largest eigenvalue is given by following this procedure

2. First we make an initial guess for the eigenvector (in my code it's the unit vector).

3. Now we iterate using

$$y^{k+1} = Mx^k \text{ and}$$
$$\beta^{k+1} = \text{The element of } y^{k+1} \text{having the largest modulus}$$

4. Follow this with

$$\mathbf{x}^{i+1} = \frac{1}{\beta^{k+1}}\mathbf{y}^{k+1}.$$

5. Then as k increases, x^k tends to the eigenvector and β^k to the eigenvalue. We stop iterating once we have reached a tolerance, in my code this is 10^{-9}. We have thus then found the first principal component and this is our vector v1.

6. We continue on to find the next principal component by defining a new matrix.

$$\mathbf{N} = \mathbf{M} - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T.$$

And then we repeat the method on N to find the next principal component.

```
double beta,betaprevious,error,normalise;
       MatrixXd Vector(num_columns-1,1);
       MatrixXd VolMatrix(num_columns-1,3);
       MatrixXd temp=covmat;//we will work on this temp matrix


       for(int k=0; k<3; k++){

              for(int i=0; i<num_columns-1; i++){
                     Vector(i,0)=1;
//first reset the Vector. This is our initial guess for the eigenvector
              }

              betaprevious=0;
              error=1;//just to initialise it

              while(error>0.000000001){

                     Vector=temp*Vector;//we iterate over Vector
                     beta=Vector.maxCoeff();
//this is the element of the vector with the largest modulus
                     for(int j=0; j<num_columns-1; j++){
                            Vector(j,0)=Vector(j,0)/beta;
                     }

                     error=abs(beta-betaprevious);//recalc the error
                     betaprevious=beta;//reset beta
              }

              normalise=0;//now we create the constant for normalising the vector
              for(int i=0; i<num_columns-1; i++){
                     normalise=normalise+Vector(i,0)*Vector(i,0);
              }
              normalise=sqrt(normalise);

              for(int i=0; i<num_columns-1; i++){
                     Vector(i,0)=sqrt(beta)*Vector(i,0)/normalise;//create the vol
                     VolMatrix(i,k)=Vector(i,0);
              }

              //now find the new matrix M-lamda*v1*v1 transpose
              MatrixXd newmatrix(num_columns-1,num_columns-1);

              for(int i=0; i<num_columns-1; i++){
                     for(int j=0; j<num_columns-1; j++){
                            newmatrix(i,j)= temp(i,j)-Vector(i,0)*Vector(j,0);
                     }
              }
              temp=newmatrix;
       }

       //Out put the vol matrix
       myfile.open ("vols.csv");
       for(int i=0; i<num_columns-1; i++){
              for(int j=0; j<3; j++){
                     myfile << VolMatrix(i,j) << ",";
              }
              myfile << "\n";
       }
```

Step Seven: Fitting the Principal Components.

In order to the fit each of the Principal Components to a curve I used Polynomial Regression. This process works as follows:
We wish to fit a polynomial of degree M

$$f(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_M x^M = \sum_{m=0}^{M} \alpha_m x^m$$

Where we have N readings. Thus we choose a set of polynomial coefficients

$$\alpha_0, \alpha_1, \alpha_2, \cdots \alpha_M$$

So as to minimise the errors

$$e_n = y_n - f(x_n)$$

Then taking our X matrix of x variables and our Y matrix of y variables we want to find the polynomials coefficient vector of α. We do this using a Vandermonde Matrix:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^M \\ 1 & x_2 & x_2^3 & \cdots & x_2^M \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_N & x_N^2 & \cdots & x_N^M \end{bmatrix}$$

With the components of V being $(x_n)^m$ We thus want to minimise the error vector

$$e = y - V\alpha$$

We do this by minimising the 2-norm of the error vector

$$\|e\|_2^2 = || \mathbf{y} - \mathbf{V\alpha}|^2 = \sum_{n=1}^{N} \left(\left(\sum_{m=0}^{M} V_{n,m} \alpha_m \right) - y_n \right)^2$$

To find the minimum we differentiate this with respect to each of the α and equate to zero.

$$0 = \frac{\partial \|e\|^2}{\partial \alpha_k} = 2 \sum_{n=1}^{N} \left(\left(\sum_{m=0}^{M} V_{n,m} \alpha_m \right) - y_n \right) V_{n,k}$$

This gives:

$$\sum_{n=1}^{N} \sum_{m=0}^{M} V_{n,k} V_{n,m} \alpha_m = \sum_{n=1}^{N} V_{n,k} y_n$$

Which is the same as:

$$V^T V \alpha = V^T y$$

Solving this gives

$$\boldsymbol{\alpha} = ((\boldsymbol{V^T V})^{-1} \boldsymbol{V^T}) \boldsymbol{y}$$

I have outlined the code involved in this below. Note that for the 1st principal component I have used the ln x when fitting i.e. using the equation **a0+a1lnx**

Each of the matrices described above has been coded using the MatrixXd class. Once we reach have fit each of the three principal components we output this data to the file fitteddata.csv. The data can then be viewed using the Curve Fitting.xlsm file.

```
int n=num_columns-1;  //n is the number of observations
int eigenpos=0;//the position we are currently fitting. 0 is position PC1

//now we have the matrices of Y, alpha and Vandermonde for each component.
MatrixXd Y1(n,1);
MatrixXd Y2(n,1);
MatrixXd Y3(n,1);
MatrixXd Vandermonde1(n,m1);
MatrixXd Vandermonde2(n,m);
MatrixXd Vandermonde3(n,m);



//First fit number 1, ie position 0
//Each of the Y components are the eigenvector times
//the square root of the eigen value.
for(int i=0; i<n; i++){
       Y1(i,0)=VolMatrix(i,eigenpos);
       //number 1 so position 0
}
//then we populate the vandermonde matrix
for(int i=0; i<n; i++){
              for(int j=0; j<m1; j++){
                     Vandermonde1(i,j)=pow(log(covariancematrix[i+1][0]),j);
                     //special case, we are using natural log of the maturity
              }
}
//now populate the matrix of polynomial coefficients
//using the formula described in the report
polyCoeffic1 = ((Vandermonde1.transpose() * Vandermonde1).inverse()
* Vandermonde1.transpose()) * Y1;

//Now fit number 2, ie position 1
eigenpos=1;

for(int i=0; i<n; i++){
       Y2(i,0)=VolMatrix(i,eigenpos);
       //number 2 so position 1
}
for(int i=0; i<n; i++){
       for(int j=0; j<m; j++){
              Vandermonde2(i,j)=pow(covariancematrix[i+1][0],j);
       }
}

polyCoeffic2 = ((Vandermonde2.transpose() * Vandermonde2).inverse()
* Vandermonde2.transpose()) * Y2;
```

```
//Now fit number 3, ie position 2
eigenpos=2;

for(int i=0; i<n; i++){
       Y3(i,0)=VolMatrix(i,eigenpos);
       //number 3 so position 2
}

for(int i=0; i<n; i++){
       for(int j=0; j<m; j++){
              Vandermonde3(i,j)=pow(covariancematrix[i+1][0],j);

       }
}
polyCoeffic3 = ((Vandermonde3.transpose() * Vandermonde3).inverse() *
Vandermonde3.transpose()) * Y3;




//now output it all to a file.
myfile.open ("fitteddata.csv");
myfile <<"maturity,vol1,vol1fitted,vol2,vol2fitted,vol3,vol3fitted\n";
for(int i=0; i<n; i++){
       myfile << covariancematrix[i+1][0]<< ","
       << Y1(i,0)<< "," <<
polyCoeffic1(0,0)+polyCoeffic1(1,0)*log(covariancematrix[i+1][0]) << ","
       << Y2(i,0)<< "," <<
polyCoeffic2(0,0)+polyCoeffic2(1,0)*covariancematrix[i+1][0]+polyCoeffic2(2,0)*pow
(covariancematrix[i+1][0],2)+polyCoeffic2(3,0)*pow(covariancematrix[i+1][0],3) <<
","
       << Y3(i,0)<< "," <<
polyCoeffic3(0,0)+polyCoeffic3(1,0)*covariancematrix[i+1][0]+polyCoeffic3(2,0)*pow
(covariancematrix[i+1][0],2)+polyCoeffic3(3,0)*pow(covariancematrix[i+1][0],3) <<
"\n";
       }
myfile << "\n";
myfile.close();
```

Step Nine: Make the final Monte Carlo Matrix.

Once we estimated all parameters for the risk-neutral SDE, we can start pricing interest rate derivatives, such as bonds and caps, using monte-carlo simulation. We do this as follows:
As described in the first section, we use the last row of the known observations as the first row to initialise the Monte-Carlo. Musiela Parameterization also requires numerical integration over fitted vol functions in order to calculate the drift.
I have used the trapezium rule to perform the integration.

```
double M(double Tau){

       int N;
       double M1, M2, M3;
       double dTau;

       if(Tau==0){
              return 0;
       }else{

              dTau= 0.01;
              N=Tau/dTau;
              dTau=Tau/N; //recalc dTau

              //using trapezium rule to compute M1
              M1 = 0.5 * Vol_1(0);
              for(int i=1; i<N ; i++){
                     M1 = M1 + Vol_1(i * dTau);
              }

        M1 = M1 + 0.5 * Vol_1(Tau);
        M1 = M1 * dTau;
        M1 = Vol_1(Tau) * M1;
       //Vol_1 represents v_i(t,T)
       //and M1 represents integral part for one factor (Slide 15)

       //using trapezium rule to compute M2
        M2 = 0.5 * Vol_2(0);
        for(int i=1; i<N ; i++){
            M2 = M2 + Vol_2(i * dTau);
              }
        M2 = M2 + 0.5 * Vol_2(Tau);
        M2 = M2 * dTau;
        M2 = Vol_2(Tau) * M2;

       //using trapezium rule to compute M3
        M3 = 0.5 * Vol_3(0);
        for(int i=1; i<N ; i++){
            M3 = M3 + Vol_3(i * dTau);
              }
        M3 = M3 + 0.5 * Vol_3(Tau);
        M3 = M3 * dTau;
        M3 = Vol_3(Tau) * M3;

              return M1 + M2 + M3; //sum for multi-factor
       }
```

In order to price derivatives we must perform multiple simulations. For each round of simulation we must generate three independent Normal random vectors to represent the dXi. I have done this using the default random number generator.

Initialisation:

```
        std::default_random_engine generator;
        std::normal_distribution<double> distribution(0.0,1.0);
```

Population of the dXi

```
for(int i=0; i<row_steps-2; i++){
//the -2 is there cause the header AND the first line dont need an RNG
//for each of the dX_i we fill them up with 1000 increments of
uncorrelated Brownian Motions
       dX_1.push_back(distribution(generator));
       dX_2.push_back(distribution(generator));
       dX_3.push_back(distribution(generator));
}
```

In order to store each of the iterations of the forward curve matrix I create a three dimensional matrix called MC

```
vector < vector < vector<double> > > MC;
```

MC is a matrix where

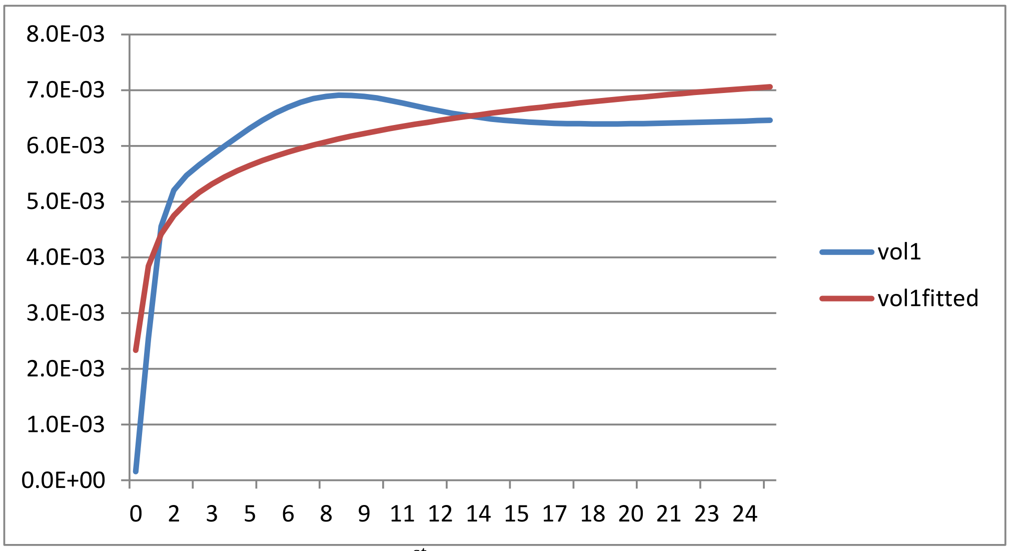
1. The Matrix MC is a vector of iterations, where
2. each iteration is a vector of rows where
3. each row is a vector of forward rates.

We then populate each row with the forward rates which we then push into the given iteration matrix. For the purpose of this project we keep the iterations between 100 and 1000.

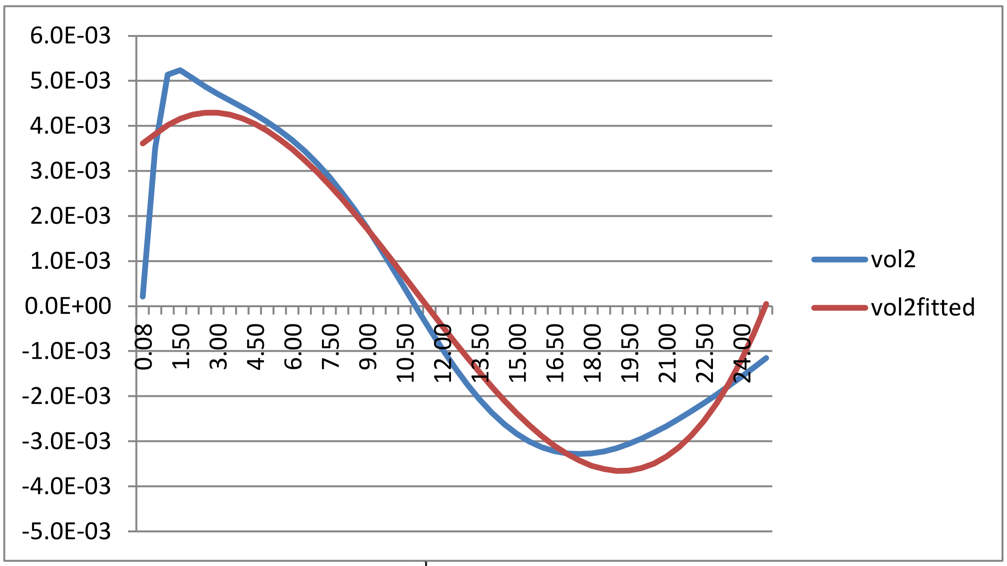
Once the evolution of the entire curve to maturity T* is calculated we can then price derivatives, including a ZCB.

Results

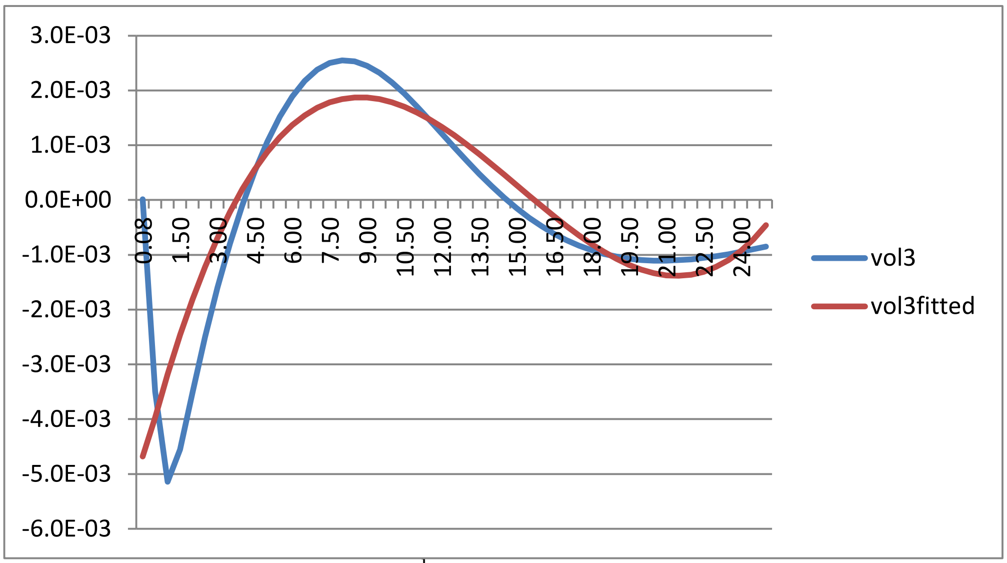
Polynomial Curve Fitting:
Below are the results of the fitted curves. The data in these curves is output by the program to the file fitteddata.csv. This can then be viewed using the file Curve Fitting.xlsm.



Fitting 1st Principal Component



Fitting 2nd Principal Component



Fitting 3rd Principal Component

Zero Coupon Bond.

Column 1 of a single iteration in the MC matrix provides us with a spot rate, ie the shortest rate we have. In the state we present it, it is discretised and mean-reverting. We can thus price the bond numerically by summation over the evolution of r_t.

$$Z(t,T) = exp\left(-\sum r_t dt\right)$$

However we needn't use the HJM model to price zero coupon bonds as their prices can be viewed in the market and can in fact be priced from the initial forward curve. There are thus two ways to price these bonds, in a single iteration of our MC matrix we can go from top to bottom (using the method above) or from left to right (using the current forward curve).

The great thing about this is it allows us to test our model. I have outlined the results below for different tenors. In its default setting the program will finish with the result for maturity 6M.
The simulated results used 1000 iterations.

Maturity	ZCB Simulated	ZCB Analytic
6M	0.977516	0.97762845
1Y	0.956092	0.95799204
2Y	0.915073	0.91668128
3Y	0.875693	0.87518625
5Y	0.800483	0.79778003
7Y	0.730869	0.72719936
10Y	0.637480	0.63493719

Cap Pricing

A cap is a type of interest rate option that pays a cashflow based on whether libor is above a rate agreed at the inception of the derivative. The payoff is defined as

$$\max(L(t,T) - K, 0) \times \tau \times N$$

Where L is forward libor calculated as an average of the instantaneous forward rates produced by the HJM model. We calculate this using the formula

$$\overline{L}(t,\tau) = \frac{1}{n}\sum_{i=1}^{n}\overline{f}_i$$

Tau is the day count fraction. For the purpose of the testing the model I have used 6 month libor over different maturities and thus I have kept this at 0.5

N is the contracts notional which I have scaled to N=1.

The cap can then be represented as the sum of each of the individual caplets. In my testing 6 month libor was used and thus a 1 year cap will have 2 caplets, a 2 year cap will have 4 and a 3 year cap will have 6.

As a way of testing my model I have priced each of these alongside identical settings using the HJM model provided in class. I have modified the model from class to run 1000 iterations and also to calculate the cashflows for caps of different maturities using 6 month libor.

While I have also modified the model from class to use the log of x when fitting the 1st principal component, I have not altered either polynomial fitting methods provided or the methods for calculating the eigenvalues and eigenvectors. While the latter should have little effect on the overall results, I suspect that the different methods of polynomial fitting resulted in the slightly different results below. The values below are the results of 1000 iterations in each of the models and each represents the average of the summed cashflows of all the caplets for each iteration. Since the notional was 1 we can think of each of these as a percentage of a contracted notional.

2%

Maturity	Cap C++	Cap VBA
1Y	2.31840	2.32342
2Y	4.51866	4.56190
3Y	6.66880	6.54430

4%

Maturity	Cap C++	Cap VBA
1Y	0.430185	0.45744
2Y	0.926893	0.98280
3Y	1.50223	1.59715

Note: if you run the model in its default state, as compiled, it will price a 2 year cap (ie 4 periods) with a strike of 2%.

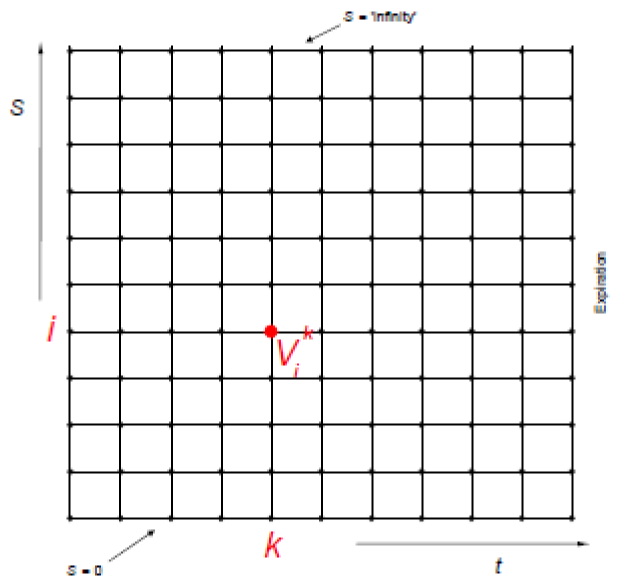
Topic 2: Uncertain Volatility and Static Hedge

Background:

This project implements the finite difference model to price a hedged exotic; in this case a European binary call option. We then find the optimised hedging ratios for short and long positions that reduce the bid/ask spread.

Finite Difference Model:

The first stage of this project is the finite difference model. We start off with a grid usually with equal time steps delta t and equal asset steps delta s. We denote these steps k and i with time in reverse direction. i.e. as k increases time goes backwards. t=T-k.deltat



Thus instead of V(S,T), we use

$$V_i^k = V(i\,\delta S, T - k\,\delta t)$$

Thus our goal is find V as a function of i and k.
We are going to find the option value at *every* point in the grid above.

Differentiation using the grid:

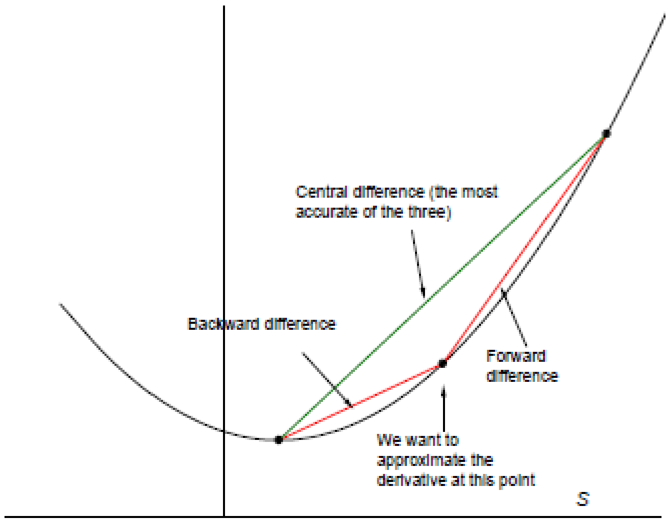
Using the definition of the first time derivative of V we can approximate the time derivative from our grid using the individual V values at each of the dots.:

$$\frac{\partial V}{\partial t}(S,t) \approx \frac{V_i^k - V_i^{k+1}}{\delta t}$$

Thus it is basically a discrete form of differentiation.
Similarly doing a taylor series on the original V function giving:

$$\frac{\partial V}{\partial t}(S,t) = \frac{V_i^k - V_i^{k+1}}{\delta t} + O(\delta t)$$

Then if we cross examine the grid at one of the time steps:


Central difference (the most accurate of the three)
Backward difference
Forward difference
We want to approximate the derivative at this point
S

We then get the forward difference, backward difference and central difference approximations:

$$\frac{V_{i+1}^k - V_i^k}{\delta S},$$

$$\frac{V_i^k - V_{i-1}^k}{\delta S}$$

$$\frac{V_{i+1}^k - V_{i-1}^k}{2\,\delta S}.$$

We find from this that the central difference (ie the line that straddles the point in the middle) is the most accurate of these formulae.

Following on from this we can approximate the Gamma using the results above

$$\frac{\partial^2 V}{\partial S^2}(S,t) \approx \frac{\frac{V_{i+1}^k - V_i^k}{\delta S} - \frac{V_i^k - V_{i-1}^k}{\delta S}}{\delta S}$$

$$= \frac{V_{i+1}^k - 2V_i^k + V_{i-1}^k}{\delta S^2}.$$

Finally we can approximate theta in a similar way arriving at

$$\frac{\partial V}{\partial t}(S,t) \approx \frac{V_i^k - V_i^{k+1}}{\delta t}.$$

Thus if we had red dots covering everywhere on the grid, we would then have everything needed for the Black-Scholes equation. In order to fill in the blanks we use boundary conditions. These can be summarised as:

1. At S=0 we know that the value of the underlying remains zero. For a cap this means V is also zero and for a put it is just the discounted value of the previous V working backwards.
2. When S is large the gamma goes to zero the value of the option becomes linear in the stock and hence we can interpolate value of V.

Uncertain Volatility:

In this project we suppose that volatility lies within a particular band

$$\sigma^- < \sigma < \sigma^+$$

Also suppose that we want to find the "worst case" price for the option. Let's call this V- (assume we are on the sell side and we want the maximum price that fits with our volatility assumption). We can then follow the Black-Scholes and no-arbitrage argument as far as we can take it.

$$\frac{\partial V^-}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V^-}{\partial S^2} + rS\frac{\partial V^-}{\partial S} - rV^- = 0$$

We can observe that for a long position in a call option for example we assume that volatility is the lower bound σ-. For a short call we assume that we have the upper bound σ+. However we can do it in full generality. The return on the max price (worst case) portfolio is then set equal to the risk free rate:

$$\min_{\sigma^- < \sigma < \sigma^+} (d\Pi) = r\Pi\, dt.$$

Thus we set

$$\min_{\sigma^- < \sigma < \sigma^+} \left(\frac{\partial V}{\partial t} + \tfrac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} \right) dt = r\left(V - S\frac{\partial V}{\partial S} \right) dt.$$

We can see from this that the volatility term is multiplied by the option's gamma. Therefore the value of σ that will give this a max or min value will depend on the sign of gamma. When gamma is positive we choose σ to be the minimum volatility σ- and when σ is negative we choose σ to be the highest value σ+.

$$\Gamma = \frac{\partial^2 V^-}{\partial S^2}$$

$$\sigma(\Gamma) = \begin{cases} \sigma^-, & \text{if} \quad \Gamma > 0 \\ \sigma^+, & \text{if} \quad \Gamma < 0 \end{cases}$$

Implementation:

This project builds upon the finite difference implementation given in class. A brief rundown of the basic implementation is as follows:

Finite Difference:
There are two parts to the basic finite difference implementation before we add in additional concepts such as Uncertain Volatility.

1. We start by calculating the payoffs for each value of Si. For a cap this will be Max[(Si-K),0] where Si is the value of the underlying at expiry and K is the strike of the option. In this part we are looping over Number of Asset Steps (ie bottom to top of graph).

2. We now start working backwards though time, while again looping over the asset steps, as we fill in the dots by calculating the delta, gamma and theta in addition to using boundary conditions.

Uncertain Volatility:
Following on from the theory section we work under the assumption that we want to price the derivatives under a range of volatility taking uncertain volatility i.e. we have a volatility range from σ- to σ+ which believe the true volatility lies. To implement this we simply want the model to choose σ to be the minimum volatility σ- when gamma is positive and choose σ to be the highest value σ+ when gamma is negative.

The implementation of this is very simple. When the program is initiated rather than give it a single volatility we provide a volminus and a volplus range. Then at each step of the finite difference walk through we check the value of gamma.

```
If gamma > 0 Then
          Vol = Volminus
          Else
          Vol = Volplus
End If
```

And simply set the volatility equal to the appropriate value.

Pricing with arbitrary vanillas:
In order to price a statically hedged option we need to modify the original code for the finite difference model by pricing a *portfolio* of options (ie a basket of options) and price it as a whole. We do this by combining the binary with call option payoffs to the FD grid at the final column of the grid (ie at expiration).

Note: for the time being I am assuming homogeneous maturity. Inhomogeneous maturity will be dealt with later.

We do this as follows:

1. Create a payoff function for binary (i.e. a Heaviside payoff function)

```
Function Heaviside(value)
If value > 0 Then
Heaviside = 1
ElseIf value = 0 Then
Heaviside = 0.5
Else
Heaviside = 0
End If
End Function
```

2. Code the payoff for the portfolio. This will consist of a binary option and an arbitrary number of call options at two separate strikes.

```
FinalPayoff =
a * Heaviside(q * (S(i) - Strike)) +
b * q1 * Application.Max((S(i) - CallStrike1), 0) +
c * q2 * Application.Max((S(i) - CallStrike2), 0)
```

   In this case the final payoff is set equal to the payoff of the 3 options. The first is the Binary option which uses the Heaviside function to determine its payoff. "q" is simply a switch to allow the user to switch between a call option and a put option (q is 1 for a call option and -1 for a put). S(i) is the price of the underlying for the given asset step. a,b and c are switches which are explained in the next section.

Pricing with arbitrary expiry of instruments:
In order to price a portfolio of options using a single pass through we implement a jump condition across the time step at which each option expires. We are working backwards in time so we start with the expiration of the last contract. This is done as follows:

1. We first determine the longest expiry "maxExp" by looking at the individual expiries of the three options.

```
maxExp = Application.Max(BinExpn, Call1Expn, Call2Expn)
```

2. Within the payoff function given under point 2 above, we add switches a,b,c which allows us to turn off and postpones the initialisation of the final payoff of each of the options. Each of these is initially set to 1.
   We also have Bint, Call1t and Call2t which are the times t (working backwards) where we initialise each of the instruments Binary, Call1 and Call2. To begin with we set them all to zero and assume they all have the same expiry and need to be initialised immediately.

```
a = 1
b = 1
c = 1
'these are used to turn each option on and off
Bint = Call1t = Call2t = 0
'this is the point k  (working backwards) where we initialise the given
option
```

3. Then for each of these options we check if its expiry is less than the max expiry.

```
If Call1Expn < maxExp Then
b = 0
End If
```

4. Then we calculate the value of Call1t so we know at which value of k to add the payoff for the specified instrument. Note the rounddown function is to make sure we are adding the payoff at the point k which is the first time step after the expiration of the option. This is accurate to O(dt). The final nested if statement is to re-initialise the instrument should it in fact be included in the intial payoff at max expiration on the right of the grid.

```
If BinExpn < maxExp Then
a = 0
BinTimeSteps = BinExpn / dt
rounddown1 = NTS - BinTimeSteps
Bint = Application.WorksheetFunction.RoundDown(rounddown1, 0)
If Bint = 0 Then
a = 1
End If
End If
```

5. Finally when we are looping over each time step we check if Call1t is the same value of k and if so we add on its payoff.

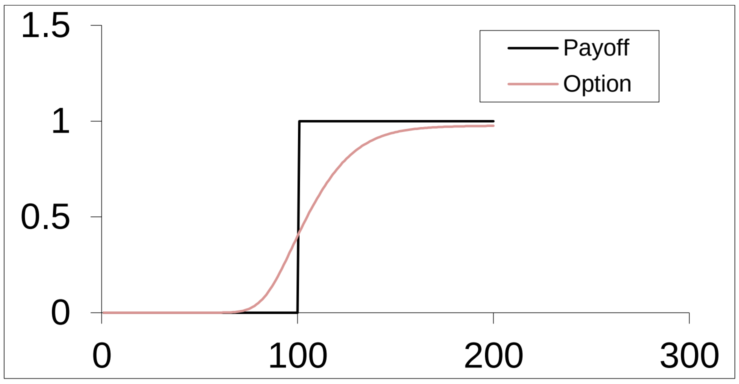
Hedge Optimisation:

Using our finished model we can now calculate the price of the portfolio. In order to get the price of the hedged binary we simply subtract the market price of the call options which we can calculate with black-scholes.

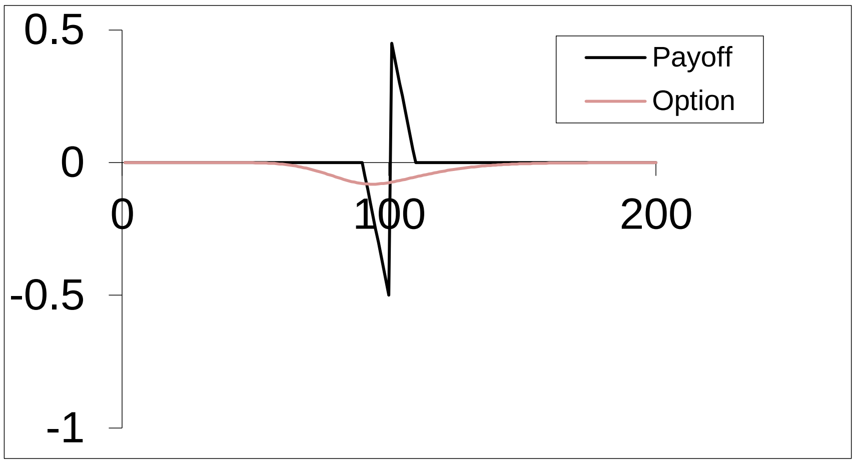
Results

1. Binary option price without hedging

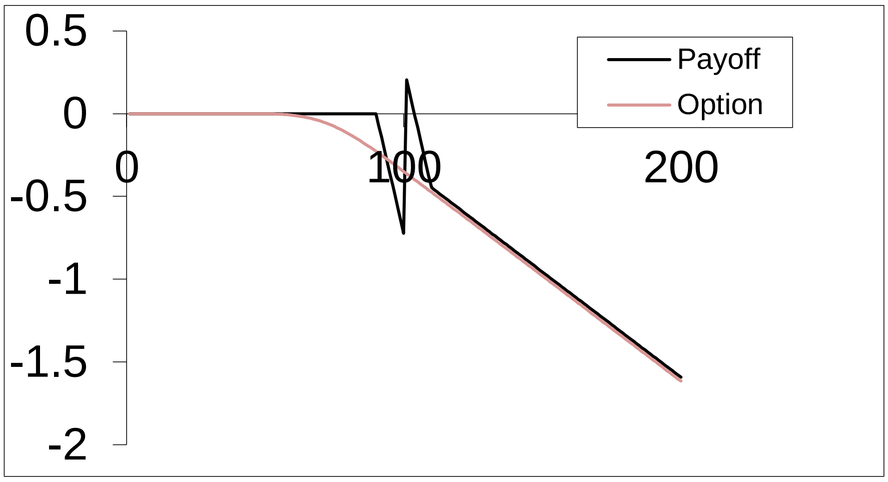
For this we simply set the quantities of each of the calls to zero.
Value of binary Bid: 0.41 Ask: 0.61



2. Statically hedged binary option prices with the quantities of each of the call options being 0.05 (short the lower strike and long the other).
   Value of binary Bid 0.45 Ask 0.563

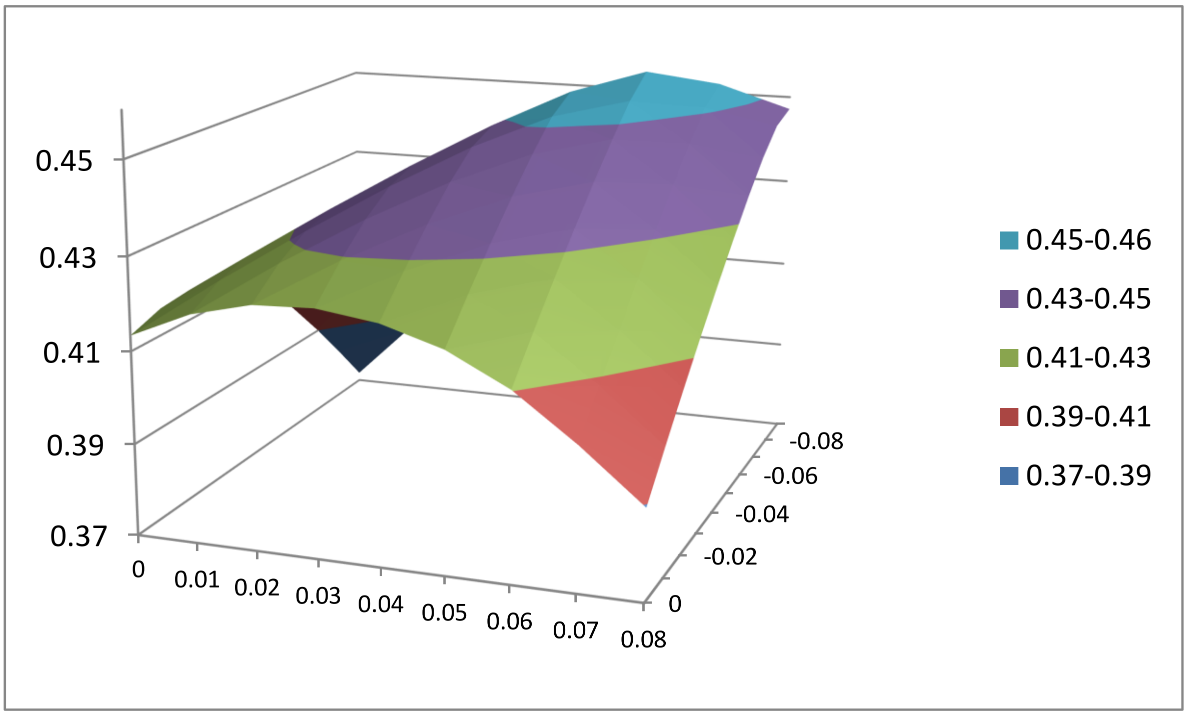


3. Optimally statically hedged binary option prices
   Value of binary Bid 0.46 Ask 0.557



Thus through static hedging we have improved on our spread of the binary from 0.20 to 0.113. Then we have improved on that again by optimally statically hedging to 0.097.

Finally I have included a plot of binary prices for different quantities for the long and short caps. Cap quantities are on the base axes with the binary price on the vertical axis.


0.45
0.43
0.41
0.39
0.37
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
-0.08
-0.06
-0.04
-0.02
0
0.45-0.46
0.43-0.45
0.41-0.43
0.39-0.41
0.37-0.39