Student Id: 9910821

Introduction

This report is a two part study of:

I) Pricing a convertible bond contract in which, **at expiry** T the holder has the option to choose between receiving the principle F or alternatively receiving R underlying stocks with price S

II) An extension to the above contract where the holder is able to exercise the decision to convert the bond in stock at **any time before** the maturity of the contract. This is known as an American embedded option

through the use of advanced numerical methods such as Crank-Nicolson with PSOR.

Task 2.1

The PDE describing such a convertible bond contract is given by

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 V}{\partial S^2} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0 \tag{1}$$

To use advanced numerical methods of (approximately) solving such PDEs we need a numerical scheme. This is a method of rewriting Equation 1 as a matrix equation as in Equation 2.

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & . & . & . & . & 0 \\ a_1 & b_1 & c_1 & 0 & . & . & . & . & . \\ 0 & a_2 & b_2 & c_2 & 0 & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & a_j & b_j & c_j & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & . & . & . & . & . & . & a_{jmax} & b_{jmax} \end{pmatrix} \begin{pmatrix} V_j^0 \\ V_j^1 \\ V_j^2 \\ . \\ V_j^i \\ . \\ V_{jmax}^i \end{pmatrix} = \begin{pmatrix} d_j^0 \\ d_j^1 \\ d_j^2 \\ . \\ d_j^i \\ . \\ d_{jmax}^i \end{pmatrix} \tag{2}$$

where j represents the steps in S and i the steps in t. The Crank-Nicolson method takes approximations of derivatives by Taylor expanding at the half time steps thus yielding

$$\frac{\partial V}{\partial t} \approx \frac{V_j^{i+1} - V_j^i}{\Delta t} \tag{3}$$

$$\frac{\partial V}{\partial S} \approx \frac{1}{4\Delta S}(V_{j+1}^i - V_{j-1}^i + V_{j+1}^{i+1} - V_{j-1}^{i+1}) \tag{4}$$

$$\frac{\partial^2 V}{\partial S^2} \approx \frac{1}{2\Delta S^2}(V_{j+1}^i - 2V_j^i + V_{j+1}^{i+1} - 2V_j^{i+1} + V_{j-1}^{i+1}) \tag{5}$$

$$V \approx \frac{1}{2}(V_j^i + V_j^{i+1}). \tag{6}$$

So substituting Equations 3 - 6 into Equation 1 gives the numerical scheme for the non-boundary regime $1 \leq j < jmax$.

$$a_j = \frac{\sigma^2 S^{2\beta}}{4\Delta S^2} - \frac{\kappa(\theta - S)}{4\Delta S} \tag{7}$$

$$b_j = \frac{1}{\Delta t} - \frac{\sigma^2 S^{2\beta}}{2\Delta S^2} - \frac{r}{2} \tag{8}$$

$$c_j = \frac{\sigma^2 S^{2\beta}}{4\Delta S^2} + \frac{\kappa(\theta - S)}{4\Delta S} \tag{9}$$

$$d_j = -\frac{V_j^{i+1}}{\Delta t} - \frac{\sigma^2 S^{2\beta}}{4\Delta S^2}(V_{j+1}^{i+1} - 2V_j^{i+1} + V_{j-1}^{i+1}) - \frac{\kappa(\theta - S)}{4\Delta S}(V_{j+1}^{i+1} - V_{j-1}^{i+1}) + \frac{r}{2}V_j^{i+1} - Ce^{-\alpha t} \tag{10}$$

The boundary conditions are problem dependent so for this particular we have two boundaries at $S = 0$ and $\lim_{S\to+\infty}$. Consider the first boundary, when $S = 0$ i.e $j = 0$. Using Equations 3 and 6 and a modified Equation 4 which becomes

$$\frac{\partial V}{\partial S} \approx \frac{1}{\Delta S}(V_{j+1}^i - V_j^i). \tag{11}$$

The numerical scheme after substituing the approximated derivates is now given by

$$a_0 = 0 \tag{12}$$

$$b_0 = -\frac{1}{\Delta t} - \frac{\kappa\theta}{\Delta S} - \frac{r}{2} \tag{13}$$

$$c_0 = \frac{\kappa\theta}{\Delta S} \tag{14}$$

$$d_0 = (-\frac{1}{\Delta t} + \frac{r}{2})V_j^{i+1} - Ce^{-\alpha t} \tag{15}$$

For the $\lim_{S\to+\infty}$ we have the condition that

$$\frac{\partial V}{\partial t} + \kappa(X - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0 \tag{16}$$

with the ansatz

$$V(S, t) = SA(t) + B(t). \tag{17}$$

It can be shown (See Appendix 1) by partial differentiation and integrating using an integrating factor method that

$$A(t) = Re^{(\kappa+r)(t-T)} \tag{18}$$

and

$$B(t) = -XRe^{(\kappa+r)(t-T)} + \frac{C}{\alpha + r}e^{-\alpha t} - \frac{C}{\alpha + r}e^{-(\alpha+r)T+rt} + XRe^{r(t-T)}. \tag{19}$$

Finally we have the last part of the numerical scheme as

$$a_0 = 0 \tag{20}$$

$$b_0 = 1 \tag{21}$$

$$c_0 = 0 \tag{22}$$

$$d_0 = SA(t) + B(t). \tag{23}$$

Using this complete numerical scheme, the method is to solve backwards in time from $i = imax$ to $i = 0$ where at each time step the Equation 2 is solved using a method such as Successive Over Relaxation (SOR) for $j = 0 \to jmax$.

For the rest of this section assume these values were used unless otherwise specified: $T = 2$, $F = 95$, $R = 2$, $r = 0.0229$, $\kappa = 0.125$, $\mu = 0.0113$, $X = 47.66$, $C = 1.09$, $\alpha = 0.02$, $\beta = 0.486$ and $\sigma = 3.03$. The value of the option $V(S, t)$ was investigated as a function of the initial underlying asset price S_0 for two cases:

1) ($\beta = 1$, $\sigma = 0.416$) with all other paramaters as previously defined

2) ($\beta = 0.486$, $\sigma = 3.03$) with all other paramaters as previously defined

The Crank-Nicolson method with the numerical scheme as calculated previously, combined with a SOR iterative method of solving the matrix equation, was implemented in code. This produced the plots seen in Figure 1.

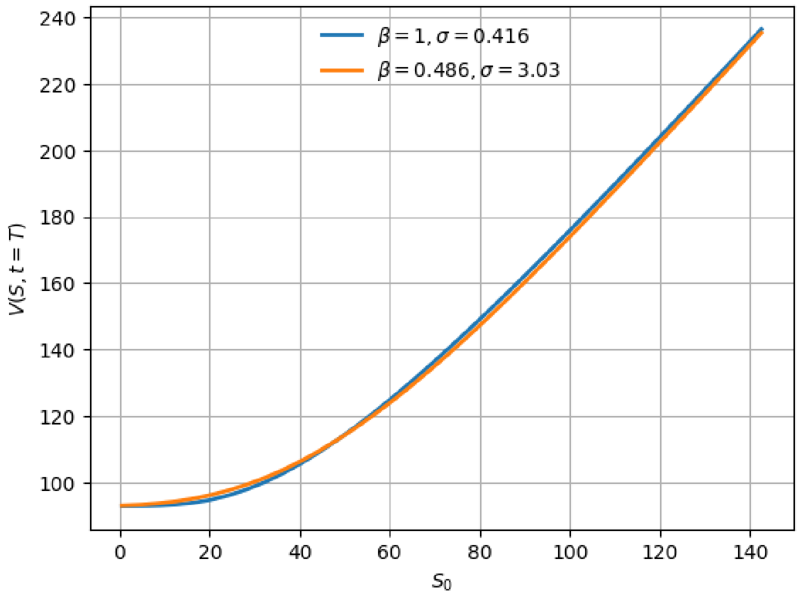

Figure 1: Value of the convertible bond $V(S, t = T)$ against inital underlying asset price at time S_0 for two combinations of β and σ.

The two configurations were therefore found to have the same effect and produce plots for the price of the bond which were very close. This prompted further analysis on the linked relationship between β, σ and $V(S,t)$. Plotting a 3D graph of the value of the portfolio for a particular S_0, here chosen to be equal to X, and vary the two other parameters. Figure 2 illustrates such a relationship which is interesting both in shape and in what it means.
Consulting literature [1], σ is the volatility or standard deviation of the underlying asset, while β is the elasticity parameter of the local volatility.

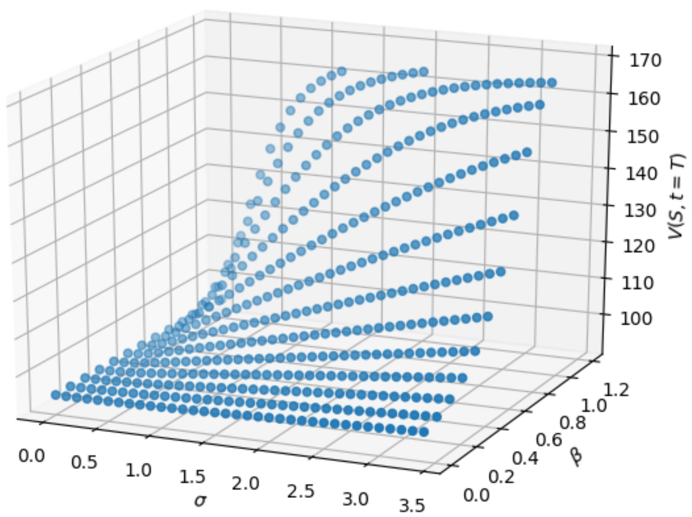

Figure 2: Value of the convertible bond $V(S = X, t = T)$ against parameters β and σ.

References

[1] V. Linetsky and R. Mendoza, *Constant Elasticity of Variance (CEV) Diffusion Model.* American Cancer Society, 2010.

Appendix 2

Portfolio Pricing Program Listing

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

/*
 * ON INPUT:
 * a, b and c -- are the diagonals of the matrix
 * rhs        -- is the right hand side
 * x          -- is the initial guess
 * iterMax    -- is maximum iterations
 * tol        -- is the tolerance level
 * omega      -- is the relaxation parameter
 * sor        -- not used
 * ON OUTPUT:
 * a, b, c, rhs          -- unchanged
 * x                     -- solution to Ax=b
 * iterMax, tol, omega   -- unchanged
 * sor                   -- number of iterations to converge
 */
void sorSolve(const std::vector<double> &a, const std::vector<double> &b, const std::vector<double> &c, const std::vector<double> &rhs,
              std::vector<double> &x, int iterMax, double tol, double omega, int &sorCount)
{
  // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
  int n = a.size() - 1;
  // sor loop
  for (sorCount = 0; sorCount < iterMax; sorCount++)
  {
    double error = 0.;
    // implement sor in here
    {
      double y = (rhs[0] - c[0] * x[1]) / b[0];
      x[0] = x[0] + omega * (y - x[0]);
    }
    for (int j = 1; j < n; j++)
    {
      double y = (rhs[j] - a[j] * x[j - 1] - c[j] * x[j + 1]) / b[j];
      x[j] = x[j] + omega * (y - x[j]);
    }
    {
      double y = (rhs[n] - a[n] * x[n - 1]) / b[n];
      x[n] = x[n] + omega * (y - x[n]);
    }
    // calculate residual norm ||r|| as sum of absolute values
    error += std::fabs(rhs[0] - b[0] * x[0] - c[0] * x[1]);
    for (int j = 1; j < n; j++)
      error += std::fabs(rhs[j] - a[j] * x[j - 1] - b[j] * x[j] - c[j] * x[j + 1]);
    error += std::fabs(rhs[n] - a[n] * x[n - 1] - b[n] * x[n]);
    // make an exit condition when solution found
```

```
    if (error < tol)
      break;
  }
}
std::vector<double> thomasSolve(const std::vector<double> &a, const std::vector<double> &b_, const std::vector<double> &c, std::vector<double> &d)
{
  int n = a.size();
  std::vector<double> b(n), temp(n);
  // initial first value of b
  b[0] = b_[0];
  for (int j = 1; j < n; j++)
  {
    b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
    d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
  }
  // calculate solution
  temp[n - 1] = d[n - 1] / b[n - 1];
  for (int j = n - 2; j >= 0; j--)
    temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];
  return temp;
}
/* Template code for the Crank Nicolson Finite Difference
 */
double crank_nicolson(double S0, double X, double F, double T, double r, double sigma,
                      double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax, int S_max, double tol, double omega, int iterMax, int &sorCount)
{
  // declare and initialise local variables (ds,dt)
  double dS = S_max / jMax;
  double dt = T / iMax;
  // create storage for the stock price and option price (old and new)
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
    vNew[j] = max(F, R * S[j]);
  }
  // start looping through time levels
  for (int i = iMax - 1; i >= 0; i--)
  {
    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
```

```
    for (int j = 1; j <= jMax - 1; j++)
    {
      //
      a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j * dS) / (4 * dS));
      b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r / 2.);
      c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (theta - j * dS)) / (4. * dS));
      d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.))) * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS)) * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
    }
    double A = R * exp((kappa + r) * (i * dt - T));
    double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
    a[jMax] = 0;
    b[jMax] = 1;
    c[jMax] = 0;
    d[jMax] = jMax * dS * A + B;
    // solve matrix equations with SOR
    sorSolve(a, b, c, d, vNew, iterMax, tol, omega, sorCount);
    //vNew = thomasSolve(a, b, c, d);

    if (sorCount == iterMax)
      return -1;

    // set old=new
    vOld = vNew;
  }
  // finish looping through time levels

  // output the estimated option price
  double optionValue;

  int jStar = S0 / dS;
  double sum = 0.;
  sum += (S0 - S[jStar]) / (dS)*vNew[jStar + 1];
  sum += (S[jStar + 1] - S0) / (dS)*vNew[jStar];
  optionValue = sum;

  return optionValue;
}

int main()
{
  //
  double T = 2., F = 95., R = 2., r = 0.0229, kappa = 0.125, altSigma = 0.416,
         mu = 0.0213, X = 47.66, C = 1.09, alpha = 0.02, beta = 0.486, sigma = 3.03, tol = 1.e-7, omega = 1., S_max = 10 * X;
  //
  /*
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.083333333, altSigma = 0.369,
         mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 1., sigma
```

```
 = 3.73, S_max = 10 * X, tol = 1.e-7, omega = 1.;
*/
int iterMax = 10000, iMax = 100, jMax = 100;
//Create graph of varying S0 and beta and bond
int length = 300;
double S_range = 3 * X;
int sor;
/*
std::ofstream outFile7("./data/varying_s_beta.csv");

for (double beta = 0; beta < 1.3; beta += 0.1)
{
  for (int j = 1; j <= length - 1; j++)
  {
    outFile7 << beta << " , " << altSigma << " , " << j * S_range /
 length << " , " << crank_nicolson(j * S_range / length, X, F, T, r,
 altSigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega,
 iterMax, sor) << "\n";
  }
}
outFile7.close();
*/
/*
std::ofstream outFile8("./data/varying_s_sigma.csv");
beta = 1;
for (double altSigma = 0; altSigma < 3.5; altSigma += 0.1)
{
  for (int j = 1; j <= length - 1; j++)
  {
    outFile8 << beta << " , " << altSigma << " , " << j * S_range /
 length << " , " << crank_nicolson(j * S_range / length, X, F, T, r,
 altSigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega,
 iterMax, sor) << "\n";
  }
}
outFile8.close();
*/
std::ofstream outFile9("./data/varying_s_sigma_beta.csv");
for (double altSigma = 0; altSigma < 3.5; altSigma += 0.1)
{
  for (double beta = 0; beta < 1.3; beta += 0.1)
  {
    double S0 = X;
    outFile9 << beta << " , " << altSigma << " , " << S0 << " , " <<
 crank_nicolson(S0, X, F, T, r, altSigma, R, kappa, mu, C, alpha, beta,
 iMax, jMax, S_max, tol, omega, iterMax, sor) << "\n";
  }
}
outFile9.close();
/*

std::ofstream outFile1("./data/varying_s_beta_1.csv");
std::ofstream outFile2("./data/varying_s_beta_0_4.csv");
for (int j = 1; j <= length - 1; j++)
{
  vector<double> gamma(jMax + 1);
  outFile1 << j * S_range / length << " , " << crank_nicolson(j * S_range
  / length, X, F, T, r, altSigma, R, kappa, mu, C, alpha, 1, iMax, jMax,
```

```
  S_max, tol, omega, iterMax, sor, gamma) << "\n";
    outFile2 << j * S_range / length << " , " << crank_nicolson(j * S_range
    / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax,
  S_max, tol, omega, iterMax, sor, gamma) << "\n";
  }
  outFile1.close();
  outFile2.close();

  std::ofstream outFile3("./data/varying_imax.csv");
  auto incFn = [](int val) { return val + 1; };
  jMax = 25;
  for (iMax = 1; iMax <= 75; iMax = incFn(iMax))
  {
    vector<double> gamma(jMax + 1);
    double S = X;
    outFile3 << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
    crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax
  , jMax, S_max, tol, omega, iterMax, sor, gamma) << "\n";
  }
  outFile3.close();
  */
  /*
  for (int s_Mult = 10; s_Mult <= 50; s_Mult+=1)
  {
    double S = X;
    S_max = s_Mult * X;
    string title = "./data/smax_jmax/"+to_string(s_Mult)+"_varying_jmax.csv
  ";
    std::ofstream outFile4(title);
    iMax = 25;
    for (jMax = 1; jMax <= 10*s_Mult; jMax +=1)
    {
      vector<double> gamma(jMax + 1);
      outFile4 << S_max << "," << iMax << "," << jMax << "," << S << " , "
  << crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta,
  iMax, jMax, S_max, tol, omega, iterMax, sor, gamma) << "\n";
    }
    outFile4.close();
  }
  */
  /*
  std::ofstream outFile5("./data/varying_smax.csv");
  iMax = 25;
  tol = 1.e-7;
  for (int s_Mult = 10; s_Mult <= 50; s_Mult+=1)
  {
    jMax = s_Mult * 10;
    double S = X;
    S_max = s_Mult * X;
    int sorCount;
    vector<double> gamma(jMax + 1);
    double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
  alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount, gamma);
    outFile5 << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
    result << "\n";
  }
  outFile5.close();

```

```
  S_max = 10 * X;
  std::ofstream outFile6("./data/analytic.csv");
  iMax = 25, jMax = 100;

  for (int j = 1; j <= length - 1; j++)
  {
    vector<double> gamma(jMax + 1);

    outFile6 << j * S_range / length << " , " << crank_nicolson(j * S_range
    / length, X, F, T, r, altSigma, R, 0, mu, C, alpha, 1, iMax, jMax,
   S_max, tol, omega, iterMax, sor, gamma) << "\n";
  }
  outFile6.close();
  */
}
```

Graphing Program Listing

```
import scipy.stats as si
import numpy as np
import csv
import matplotlib.pyplot as plt

C = 1.09
alpha = 0.02
r = 0.0229
T = 2.
F = 95.
R = 2.
sigma = 0.416
K = 47.66
'''
T=3
C=0.106
alpha=0.01
r=0.0038
R=1
F=56
sigma = 0.369
K=56.47
'''
#Calulate value of coupon
#Through integrating Cexp(-(alpha+r)t)dt from 0 to T
COUPON = C/(alpha+r) * (1- np.exp((-(alpha+r)*T)))

BOND = F*np.exp(-r*T)

variationData=[]
with open('data/analytic.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.
   QUOTE_NONNUMERIC)
    currentData={'x':[],'y':[]}
    for row in reader:
        currentData['x'].append(row['S'])
        currentData['y'].append(row['V'])
    variationData.append(currentData)

def euro_vanilla_call(S, K, T, r, sigma):

```

```python
    #S: spot price
    #K: strike price
    #T: time to maturity
    #r: interest rate
    #sigma: volatility of underlying asset

    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = (np.log(S / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))

    call = (S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si.norm.cdf(d2, 0.0, 1.0))

    return call

ANALYTIC_PRICE = []
STOCK_PRICE = []
BOND_FLOOR = []
CONV_BOND = []
for s in range(1,140):
    STOCK_PRICE.append(s)
    ANALYTIC_PRICE.append(R*euro_vanilla_call(s, K, T, r, sigma) + BOND + COUPON)

#plt.plot(S,V1, label = " beta = 1")
#plt.plot(STOCK_PRICE,BOND_FLOOR, label = "Bond")
plt.plot(STOCK_PRICE,ANALYTIC_PRICE, label = "Analytic")
plt.plot(variationData[0]['x'],variationData[0]['y'], label = "Crank")
plt.xlabel('Stock price')
plt.ylabel('Eurocall Option')
plt.legend()
plt.savefig('images/analytic.png',bbox_inches='tight', pad_inches=0.2)
```

```python
import matplotlib.pyplot as plt
import numpy as np
import csv
import scipy.stats as si

X=47.66
R=2
F=95
T=2.0
C = 1.09
alpha = 0.02
r = 0.0229
T = 2.
sigma = 0.416
'''
def euro_vanilla_call(S, K, T, r, sigma):

    #S: spot price
    #K: strike price
    #T: time to maturity
    #r: interest rate
    #sigma: volatility of underlying asset

    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
```

```
    )
    d2 = (np.log(S / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))

    call = (R*S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si.norm.cdf(d2, 0.0, 1.0))

    return call

variationData=[]
with open('data/varying_imax.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'x':[],'y':[],'jMax':0,'sMax':0}
    for row in reader:
        currentData['x'].append(row['iMax'])
        currentData['y'].append(row['V'])
        currentData['jMax']=row['jMax']
        currentData['sMax']=int(row['sMax']/X)
    variationData.append(currentData)

plt.figure()
plt.grid()
plt.plot(variationData[0]['x'],variationData[0]['y'],label=r'$\beta=0.486,\sigma=3.03,jMax=%i,sMax=%i$'%(variationData[0]['jMax'],variationData[0]['sMax']),linewidth=2)
plt.xlabel('iMax')
plt.ylabel(r'$V(S=X,t=0)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/european_varying_imax.png',bbox_inches='tight',pad_inches=0.2)

with open('data/varying_smax.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'x':[],'y':[],'jMax':[],'iMax':0}
    for row in reader:
        currentData['x'].append(int(row['sMax']/X))
        currentData['y'].append(row['V'])
        currentData['jMax'].append(row['jMax'])
        currentData['iMax']=row['iMax']
    variationData.append(currentData)

fig, ax1 = plt.subplots()
ax1.set_xlabel(r'sMax (multiples of X)')
ax1.set_ylabel(r'$V(S=X,t=0)$')
ax1.grid()
ax1.scatter(np.asarray(variationData[1]['x'][:20]),variationData[1]['y'][:20],label=r'$V(S=X,t=0)$ for $\beta=0.486,\sigma=3.03,iMax=%i$'%(variationData[1]['iMax']))
ax2 = ax1.twinx()
ax2.set_ylabel(r'$jMax$')
fig.tight_layout()
ax2.plot(np.asarray(variationData[1]['x'][:20]),variationData[1]['jMax'][:20],label=r'$jMax$',color="orange")
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2,loc='lower right',fancybox=
```

```
    False, framealpha=0.0)
plt.savefig('images/european_varying_smax_zoomed.png',bbox_inches='tight', pad_inches=0.2)

fig, ax1 = plt.subplots()
ax1.set_xlabel(r'sMax (multiples of X)')
ax1.set_ylabel(r'$V(S=X,t=0)$')
ax1.grid()
ax1.scatter(np.asarray(variationData[1]['x']),variationData[1]['y'],label=r'$V(S=X,t=0)$ for $\beta=0.486,\sigma=3.03,iMax=%i$'%(variationData[1]['iMax']))
ax2 = ax1.twinx()
ax2.set_ylabel(r'$jMax$')
fig.tight_layout()
ax2.plot(np.asarray(variationData[1]['x']),variationData[1]['jMax'],label=r'$jMax$',color="orange")
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2,loc='lower right',fancybox=False, framealpha=0.0)
plt.savefig('images/european_varying_smax.png',bbox_inches='tight', pad_inches=0.2)

'''
'''
for smax in range(10,101):
    currentData={'x':[], 'y':[], 'iMax':0, 'sMax':0}
    with open('data/smax_jmax/'+str(smax)+'_varying_jmax.csv', newline='\n') as csvfile:
        reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V'],quoting=csv.QUOTE_NONNUMERIC)
        for row in reader:
            currentData['x'].append(row['jMax'])
            currentData['y'].append(row['V'])
            currentData['iMax']=row['iMax']
            currentData['sMax']=int(row['sMax']/X)

    plt.figure()
    plt.plot(currentData['x'],currentData['y'],label=r'$\beta=0.486,\sigma=3.03,iMax=%i,sMax=%i$'%(currentData['iMax'],currentData['sMax']), linewidth=2)
    plt.xlabel('jMax')
    plt.ylabel(r'$V(S=X,t=0)$')
    plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
    plt.grid()
    plt.savefig('images/smax_jmax/'+str(smax)+'_european_varying_jmax.png', bbox_inches='tight', pad_inches=0.2)
    plt.close()
'''
'''
allData=[]
with open('data/varying_s_beta_1.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
```

```
    allData.append(currentData)

with open('data/varying_s_beta_0_4.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'S':[],'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)

plt.figure()
plt.grid()
plt.plot(allData[0]['S'],allData[0]['V'],label=r'$\beta=1,\sigma=0.416$',linewidth=2)
plt.plot(allData[1]['S'],allData[1]['V'],label=r'$\beta=0.486,\sigma=3.03$',linewidth=2)
#plt.plot(allData[0]['S'],np.ones(len(allData[0]['S'])) * F,label=r'Principal Only',linewidth=2)
equityOnly_1 = [ R*euro_vanilla_call(s, X, T, r, sigma) for s in allData[0]['S'] ]
equityOnly_2 = [ R*euro_vanilla_call(s, X, T, r, 3.03) for s in allData[0]['S'] ]
#plt.plot(allData[0]['S'],equityOnly_1,label=r'Equity Only $\sigma=0.416$',linewidth=2)
#plt.plot(allData[0]['S'],equityOnly_2,label=r'Equity Only $\sigma=3.03$',linewidth=2)

plt.xlabel(r'$S_0$')
plt.ylabel(r'$V(S,t=T)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/european_varying_s.png',bbox_inches='tight', pad_inches=0.2)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

currentData={'S':[],'V':[],'beta':[]}
with open('data/varying_s_beta.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['beta','altsigma','S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'S':[],'V':[],'beta':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
        currentData['beta'].append(row['beta'])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(currentData['S'],currentData['V'],currentData['beta'])
plt.savefig('images/european_varying_s_varying_beta.png',bbox_inches='tight', pad_inches=0.2)
'''
currentData={'S':[],'V':[],'beta':[],'sigma':[]}
with open('data/varying_s_sigma_beta.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['beta','sigma','S','V'],quoting=csv.QUOTE_NONNUMERIC)
    for row in reader:
```

```
        if(row['V']==-1):
            continue
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
        currentData['beta'].append(row['beta'])
        currentData['sigma'].append(row['sigma'])
'''
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(currentData['sigma'],currentData['beta'],currentData['V'])
ax.set_xlabel(r'$\sigma$')
ax.set_ylabel(r'$\beta$')
ax.set_zlabel(r'$V(S,t=T)$')
plt.show()
plt.savefig('images/european_varying_s_varying_sigma_varying_beta.png', bbox_inches='tight', pad_inches=0.2)
'''

fig = plt.figure()
ax2 = fig.add_subplot(111)
ax2.hist2d(currentData['sigma'],currentData['beta'],bins=100,weights=currentData['V'])
ax2.set_xlabel(r'$\sigma$')
ax2.set_ylabel(r'$\beta$')
#ax2.set_zlabel(r'$V(S,t=T)$')
plt.show()
plt.savefig('images/hist2d_european_varying_s_varying_sigma_varying_beta.png',bbox_inches='tight', pad_inches=0.2)
```