

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

This task consisted in valuing a portfolio comprising of shorting a call option with strike price X_1 , longing a call option with strike price X_2 , longing $2X_2$ binary cash or nothing call options with strike price X_2 and unit payoff and longing a call option with strike price equal to zero with parameters:

Appendix

Portfolio Pricing Program Listing

```
1 #include <iostream>
2 #include <fstream>
3 #include <cmath>
4 #include <vector>
5 #include <algorithm>
6 using namespace std;
7
8 /*
9  * ON INPUT:
10  * a, b and c — are the diagonals of the matrix
11  * rhs — is the right hand side
12  * x — is the initial guess
13  * iterMax — is maximum iterations
14  * tol — is the tolerance level
15  * omega — is the relaxation parameter
16  * sor — not used
17  * ON OUTPUT:
18  * a, b, c, rhs — unchanged
19  * x — solution to Ax=b
20  * iterMax, tol, omega — unchanged
21  * sor — number of iterations to converge
22  */
23 void sorSolve(const std::vector<double> &a, const std::vector<double> &b,
24              const std::vector<double> &c, const std::vector<double> &rhs,
25              std::vector<double> &x, int iterMax, double tol, double omega
26              , int &sorCount)
27 {
28     // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
29     int n = a.size() - 1;
30     // sor loop
31     for (sorCount = 0; sorCount < iterMax; sorCount++)
32     {
33         double error = 0.;
34         // implement sor in here
35         {
36             double y = (rhs[0] - c[0] * x[1]) / b[0];
37             x[0] = x[0] + omega * (y - x[0]);
38         }
39         for (int j = 1; j < n; j++)
40         {
41             double y = (rhs[j] - a[j] * x[j - 1] - c[j] * x[j + 1]) / b[j];
42             x[j] = x[j] + omega * (y - x[j]);
43         }
44         {
45             double y = (rhs[n] - a[n] * x[n - 1]) / b[n];
46             x[n] = x[n] + omega * (y - x[n]);
47         }
48         // calculate residual norm ||r|| as sum of absolute values
49         error += std::fabs(rhs[0] - b[0] * x[0] - c[0] * x[1]);
50         for (int j = 1; j < n; j++)
51             error += std::fabs(rhs[j] - a[j] * x[j - 1] - b[j] * x[j] - c[j] * x[
52 j + 1]);
53         error += std::fabs(rhs[n] - a[n] * x[n - 1] - b[n] * x[n]);
54         // make an exit condition when solution found
```

```

52     if (error < tol)
53         break;
54     }
55 }
56 std::vector<double> thomasSolve(const std::vector<double> &a, const std::
    vector<double> &b_, const std::vector<double> &c, std::vector<double> &d
    )
57 {
58     int n = a.size();
59     std::vector<double> b(n), temp(n);
60     // initial first value of b
61     b[0] = b_[0];
62     for (int j = 1; j < n; j++)
63     {
64         b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
65         d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
66     }
67     // calculate solution
68     temp[n - 1] = d[n - 1] / b[n - 1];
69     for (int j = n - 2; j >= 0; j--)
70         temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];
71     return temp;
72 }
73 /* Template code for the Crank Nicolson Finite Difference
74 */
75 double crank_nicolson(double S0, double X, double F, double T, double r,
    double sigma,
76                     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, std::vector<double> &gamma)
77 {
78     // declare and initialise local variables (ds,dt)
79     double dS = S_max / jMax;
80     double dt = T / iMax;
81     // create storage for the stock price and option price (old and new)
82     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
83     // setup and initialise the stock price
84     for (int j = 0; j <= jMax; j++)
85     {
86         S[j] = j * dS;
87     }
88     // setup and initialise the final conditions on the option price
89     for (int j = 0; j <= jMax; j++)
90     {
91         vOld[j] = max(F, R * S[j]);
92         vNew[j] = max(F, R * S[j]);
93     }
94     // start looping through time levels
95     for (int i = iMax - 1; i >= 0; i--)
96     {
97         // declare vectors for matrix equations
98         vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
99         // set up matrix equations a[j]=
100         double theta = (1 + mu) * X * exp(mu * i * dt);
101         a[0] = 0;
102         b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
103         c[0] = (kappa * theta / dS);
104         d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));

```

```

105     for (int j = 1; j <= jMax - 1; j++)
106     {
107         //
108         a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (
kappa * (theta - j * dS) / (4 * dS));
109         b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. *
pow(dS, 2))) - (r / 2.);
110         c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.)
)) + ((kappa * (theta - j * dS)) / (4. * dS));
111         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));
112     }
113     double A = R * exp((kappa + r) * (i * dt - T));
114     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);
115     a[jMax] = 0;
116     b[jMax] = 1;
117     c[jMax] = 0;
118     d[jMax] = jMax * dS * A + B;
119     // solve matrix equations with SOR
120     sorSolve(a, b, c, d, vNew, iterMax, tol, omega, sorCount);
121     //vNew = thomasSolve(a, b, c, d);
122     gamma[0] = 0;
123     for (size_t j = 1; j < jMax; j++)
124     {
125         gamma[j] = (1 / (2 * pow(dS, 2))) * (vNew[j + 1] - 2 * vNew[j] + vNew
[j - 1] + vOld[j + 1] - 2 * vOld[j] + vOld[j - 1]);
126     }
127     gamma[jMax] = 0;
128     if (sorCount == iterMax)
129         return -1;
130
131     // set old=new
132     vOld = vNew;
133 }
134 // finish looping through time levels
135
136 // output the estimated option price
137 double optionValue;
138
139 int jStar = S0 / dS;
140 double sum = 0.;
141 sum += (S0 - S[jStar]) / (dS) * vNew[jStar + 1];
142 sum += (S[jStar + 1] - S0) / (dS) * vNew[jStar];
143 optionValue = sum;
144
145 return optionValue;
146 }
147
148 int main()
149 {
150     //
151     double T = 2., F = 95., R = 2., r = 0.0229, kappa = 0.125, altSigma =
0.416,

```

```

152     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;
153 //
154 /*
155 double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.083333333, altSigma
    = 0.369,
156     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.;
157 */
158 int iterMax = 10000, iMax = 100, jMax = 100;
159 //Create graph of varying S and optionvalue
160 int length = 300;
161 double S_range = 3 * X;
162 int sor;
163 std::ofstream outFile1("./data/varying-s-beta-1.csv");
164 std::ofstream outFile2("./data/varying-s-beta-0-4.csv");
165 for (int j = 1; j <= length - 1; j++)
166 {
167     vector<double> gamma(jMax + 1);
168     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";
169     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";
170 }
171 outFile1.close();
172 outFile2.close();
173
174 std::ofstream outFile3("./data/varying-imax.csv");
175 auto incFn = [](int val) { return val + 1; };
176 jMax = 25;
177 for (iMax = 1; iMax <= 75; iMax = incFn(iMax))
178 {
179     vector<double> gamma(jMax + 1);
180     double S = X;
181     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";
182 }
183 outFile3.close();
184
185 std::ofstream outFile4("./data/varying-jmax.csv");
186 iMax = 25;
187 for (jMax = 1; jMax <= 100; jMax = incFn(jMax))
188 {
189     vector<double> gamma(jMax + 1);
190     double S = X;
191     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";
192 }
193 outFile4.close();
194
195 std::ofstream outFile5("./data/varying-smax.csv");
196 iMax = 10;
197 tol = 1.e-7;
198 for (int s_Mult = 10; s_Mult <= 100; s_Mult = incFn(s_Mult))

```

```

199 {
200     jMax = s_Mult * 5;
201     double S = X;
202     S_max = s_Mult * X;
203     double result = 0, prevResult = 0;
204     std::cout << s_Mult << std::endl;
205     do
206     {
207         int sorCount;
208         prevResult = result;
209         jMax += 5;
210         vector<double> gamma(jMax + 1);
211         result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C, alpha,
212             beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount, gamma);
213     } while (trunc(1.e5 * prevResult) != trunc(1.e5 * result) && result !=
214         -1);
215     outFile5 << S_max << ", " << iMax << ", " << jMax << ", " << S << " , " <<
216         prevResult << "\n";
217 }
218 outFile5.close();
219
220 S_max = 30 * X;
221 std::ofstream outFile6("./data/analytic.csv");
222 iMax = 10, jMax = 150;
223
224 for (int j = 1; j <= length - 1; j++)
225 {
226     vector<double> gamma(jMax + 1);
227
228     outFile6 << j * S_range / length << " , " << crank_nicolson(j * S_range
229         / length, X, F, T, r, altSigma, R, 0, mu, C, alpha, 1, iMax, jMax,
230         S_max, tol, omega, iterMax, sor, gamma) << "\n";
231 }
232 outFile6.close();
233 }

```

Graphing Program Listing

```

1 import scipy.stats as si
2 import numpy as np
3 import csv
4 import matplotlib.pyplot as plt
5
6 C = 1.09
7 alpha = 0.02
8 r = 0.0229
9 T = 2.
10 F = 95.
11 R = 2.
12 sigma = 0.416
13 K = 47.66
14 ', '
15 T=3
16 C=0.106
17 alpha=0.01
18 r=0.0038
19 R=1
20 F=56
21 sigma = 0.369

```

```

22 K=56.47
23 '''
24 #Calculate value of coupon
25 #Through integrating Cexp(-(alpha+r)t)dt from 0 to T
26 COUPON = C/(alpha+r) * (1- np.exp(-(alpha+r)*T))
27
28 BOND = F*np.exp(-r*T)
29
30 variationData=[]
31 with open('data/analytic.csv', newline='\n') as csvfile:
32     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
33     QUOTE_NONNUMERIC)
34     currentData={'x':[], 'y':[]}
35     for row in reader:
36         currentData['x'].append(row['S'])
37         currentData['y'].append(row['V'])
38         variationData.append(currentData)
39
40 def euro_vanilla_call(S, K, T, r, sigma):
41     #S: spot price
42     #K: strike price
43     #T: time to maturity
44     #r: interest rate
45     #sigma: volatility of underlying asset
46
47     d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
48     )
49     d2 = (np.log(S / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
50     )
51
52     call = (S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si.norm.
53     cdf(d2, 0.0, 1.0))
54
55     return call
56
57 ANALYTIC_PRICE = []
58 STOCK_PRICE = []
59 BOND_FLOOR = []
60 CONV_BOND = []
61 for s in range(1,140):
62     STOCK_PRICE.append(s)
63     ANALYTIC_PRICE.append(R*euro_vanilla_call(s, K, T, r, sigma) + BOND +
64     COUPON)
65
66 #plt.plot(S,V1, label = " beta = 1")
67 #plt.plot(STOCK_PRICE,BOND_FLOOR, label = "Bond")
68 plt.plot(STOCK_PRICE,ANALYTIC_PRICE, label = "Analytic")
69 plt.plot( variationData[0]['x'],variationData[0]['y'], label = "Crank")
70 plt.xlabel('Stock price')
71 plt.ylabel('Eurocall Option')
72 plt.legend()
73 plt.savefig('images/analytic.png',bbox_inches='tight', pad_inches=0.2)
74
75 import matplotlib.pyplot as plt
76 import numpy as np
77 import csv
78

```

```

5
6 X=47.66
7 R=2
8 F=95
9
10 variationData=[]
11 with open('data/varying_imax.csv', newline='\n') as csvfile:
12     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V'], quoting=csv.QUOTE_NONNUMERIC)
13     currentData={'x':[], 'y':[], 'jMax':0, 'sMax':0}
14     for row in reader:
15         currentData['x'].append(row['iMax'])
16         currentData['y'].append(row['V'])
17         currentData['jMax']=row['jMax']
18         currentData['sMax']=int(row['sMax']/X)
19     variationData.append(currentData)
20
21 plt.figure()
22 plt.grid()
23 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)
24 plt.xlabel('iMax')
25 plt.ylabel(r'$V(S=X, t=0)$')
26 plt.legend(loc='upper center', fancybox=False, framealpha=0.0)
27 plt.savefig('images/european_varying_imax.png', bbox_inches='tight', pad_inches=0.2)
28
29 with open('data/varying_jmax.csv', newline='\n') as csvfile:
30     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V'], quoting=csv.QUOTE_NONNUMERIC)
31     currentData={'x':[], 'y':[], 'iMax':0, 'sMax':0}
32     for row in reader:
33         currentData['x'].append(row['jMax'])
34         currentData['y'].append(row['V'])
35         currentData['iMax']=row['iMax']
36         currentData['sMax']=int(row['sMax']/X)
37     variationData.append(currentData)
38
39 plt.figure()
40 plt.plot(variationData[1]['x'], variationData[1]['y'], label=r'$\beta=0.486,\sigma=3.03,iMax=\%i,sMax=\%i$'%(variationData[1]['iMax'], variationData[1]['sMax']), linewidth=2)
41 plt.xlabel('jMax')
42 plt.ylabel(r'$V(S=X, t=0)$')
43 plt.legend(loc='upper center', fancybox=False, framealpha=0.0)
44 plt.grid()
45 plt.savefig('images/european_varying_jmax.png', bbox_inches='tight', pad_inches=0.2)
46
47 with open('data/varying_smax.csv', newline='\n') as csvfile:
48     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V'], quoting=csv.QUOTE_NONNUMERIC)
49     currentData={'x':[], 'y':[], 'jMax':[], 'iMax':0}
50     for row in reader:
51         currentData['x'].append(int(row['sMax']/X))
52         currentData['y'].append(row['V'])
53         currentData['jMax'].append(row['jMax'])

```



```

54     currentData[ 'iMax' ]=row[ 'iMax' ]
55     variationData.append(currentData)
56
57 fig, ax1 = plt.subplots()
58 ax1.set_xlabel(r'sMax (multiples of X)')
59 ax1.set_ylabel(r'$V(S=X, t=0)$')
60 ax1.grid()
61 ax1.scatter(np.asarray( variationData[2][ 'x' ][:20]), variationData[2][ 'y'
    ][:20], label=r'$V(S=X, t=0)$ for $\beta=0.486, \sigma=3.03, iMax=\%i$'%(
    variationData[2][ 'iMax' ]))
62 ax2 = ax1.twinx()
63 ax2.set_ylabel(r'$jMax$')
64 fig.tight_layout()
65 ax2.plot(np.asarray( variationData[2][ 'x' ][:20]), variationData[2][ 'jMax'
    ][:20], label=r'$jMax$', color="orange")
66 lines, labels = ax1.get_legend_handles_labels()
67 lines2, labels2 = ax2.get_legend_handles_labels()
68 ax2.legend(lines + lines2, labels + labels2, loc='lower right', fancybox=
    False, framealpha=0.0)
69 plt.savefig('images/european-varying-smax-zoomed.png', bbox_inches='tight',
    pad_inches=0.2)
70
71 fig, ax1 = plt.subplots()
72 ax1.set_xlabel(r'sMax (multiples of X)')
73 ax1.set_ylabel(r'$V(S=X, t=0)$')
74 ax1.grid()
75 ax1.scatter(np.asarray( variationData[2][ 'x' ]), variationData[2][ 'y' ], label=r
    '$V(S=X, t=0)$ for $\beta=0.486, \sigma=3.03, iMax=\%i$'%(variationData[2][ '
    iMax' ]))
76 ax2 = ax1.twinx()
77 ax2.set_ylabel(r'$jMax$')
78 fig.tight_layout()
79 ax2.plot(np.asarray( variationData[2][ 'x' ]), variationData[2][ 'jMax' ], label=r
    '$jMax$', color="orange")
80 lines, labels = ax1.get_legend_handles_labels()
81 lines2, labels2 = ax2.get_legend_handles_labels()
82 ax2.legend(lines + lines2, labels + labels2, loc='lower right', fancybox=
    False, framealpha=0.0)
83 plt.savefig('images/european-varying-smax.png', bbox_inches='tight',
    pad_inches=0.2)
84
85 allData=[]
86 with open('data/varying-s-beta_1.csv', newline='\n') as csvfile:
87     reader = csv.DictReader( csvfile, fieldnames=[ 'S', 'V' ], quoting=csv.
    QUOTE_NONNUMERIC)
88     currentData={ 'S':[], 'V':[] }
89     for row in reader:
90         currentData[ 'S' ].append(row[ 'S' ])
91         currentData[ 'V' ].append(row[ 'V' ])
92     allData.append(currentData)
93
94 with open('data/varying-s-beta_0_4.csv', newline='\n') as csvfile:
95     reader = csv.DictReader( csvfile, fieldnames=[ 'S', 'V' ], quoting=csv.
    QUOTE_NONNUMERIC)
96     currentData={ 'S':[], 'V':[] }
97     for row in reader:
98         currentData[ 'S' ].append(row[ 'S' ])
99         currentData[ 'V' ].append(row[ 'V' ])

```

```

100     allData.append(currentData)
101
102 plt.figure()
103 plt.grid()
104 plt.plot(allData[0]['S'],allData[0]['V'],label=r'$\beta=1,\sigma=0.416$',
           linewidth=2)
105 plt.plot(allData[1]['S'],allData[1]['V'],label=r'$\beta=0.486,\sigma=3.03$',
           linewidth=2)
106 plt.plot(allData[1]['S'],np.ones(len(allData[1]['S'])) * F,label=r'Bond
           Only',linewidth=2)
107 plt.plot(allData[1]['S'],np.asarray(allData[1]['S'])*R,label=r'Options Only
           ',linewidth=2)
108
109 plt.xlabel('S')
110 plt.ylabel(r'$V(S,t=0)$')
111 plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
112 plt.savefig('images/european_varying_s.png',bbox_inches='tight', pad_inches
           =0.2)

```