

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. Moreover a put option will be embedded in this contract

through the use of advanced numerical methods such as Crank-Nicolson with PSOR and Penalty method. Particularly, convergence and convergence rates will be studied together with susceptibility to certain algorithmic parameters.

1 European Type Option Convertible Bond

The PDE describing such a convertible bond contract is given by

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$a_0 = 0 \quad (12)$$

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

$$c_0 = \frac{\kappa\theta}{\Delta S} \quad (14)$$

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

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

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

with the ansatz

$$V(S, t) = SA(t) + B(t). \quad (17)$$

It can be shown by partial differentiation and integrating using an integrating factor method that

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

and

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

Finally we have the last part of the numerical scheme as

$$a_0 = 0 \quad (20)$$

$$b_0 = 1 \quad (21)$$

$$c_0 = 0 \quad (22)$$

$$d_0 = SA(t) + B(t). \quad (23)$$

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 $j = 0 \rightarrow jmax$.

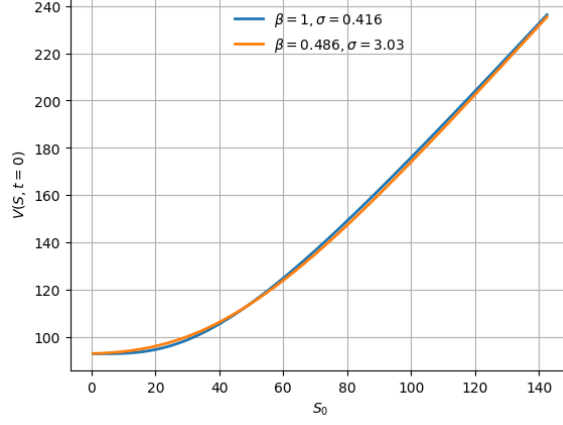


Figure 1: Value of the convertible bond $V(S, t = 0)$ against initial underlying asset price at time S_0 for two combinations of β and σ .

1.1 Investigating β and σ

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 $\sigma = 3.03$. For the SOR method a value of $\omega = 1.4$ was used with a maximum cap of iterations of 100,000. 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 parameters as previously defined
- 2) ($\beta = 0.486$, $\sigma = 3.03$) with all other parameters 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. 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 $V(S, t)$. A 3D graph of the value of the portfolio for a particular S_0 , here chosen to be equal to X , and the two other parameters was plotted.

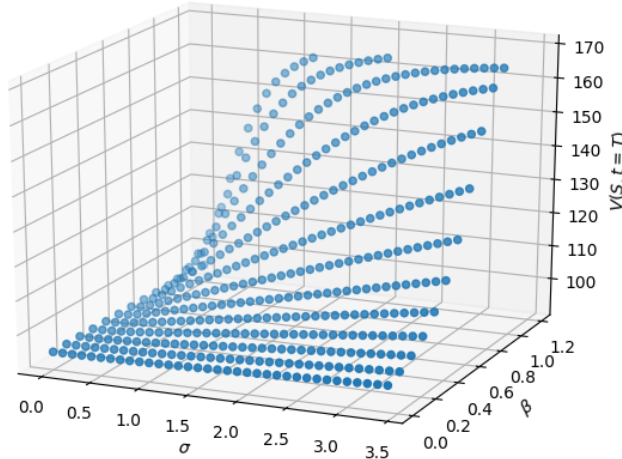


Figure 2: Value of the convertible bond $V(S = X, t = T)$ against parameters β and σ .

Figure 2 illustrates such a relationship which is interesting both in shape and in what it can be modelled by. Going back a few steps, the risk-neutral process followed by the underlying stock price is given by

$$dS = \kappa(\theta(t) - S)dt + \sigma S^\beta dW \quad (24)$$

which is an Ornstein-Uhlenbeck (OU) process [1] with a drift term of function $\theta(t)$, together with a Constant Elasticity of Variance [2] model where the local variance is a powerlaw of elasticity. Using this model, σ is defined to be the actual Black-Scholes volatility or standard deviation of the underlying asset, while β is the elasticity parameter of the local volatility. Moreover, using this model the values of β which should be used are for ≤ 1 . Above this, there are implications on the inaccessibility of the origin which for a stock price means it cannot go bankrupt which is not true for stocks or convertible bonds but true for certain commodities such as gold. Thus, we shall stick for values of $\beta \leq 1$ in this report. In this regime, the model captures the so-called 'leverage effect' where the stock price and volatility are inversely proportional [3]. The parameter β in Equation 24 controls the steepness of the implied volatility skew which is something seen in Figure 2. The parameter σ is now part of a scale parameter which fixes the 'at-the-money' (S close to X regime) volatility level. This happens since the variance of the underlying is given by $\text{Var}(S) \propto S^{2\beta} \sigma^2$. So, there are $\sigma - \beta$ planes on Figure 2 which have close values of the convertible bond for multiple combinations of (σ, β) .

1.2 Varying step sizes

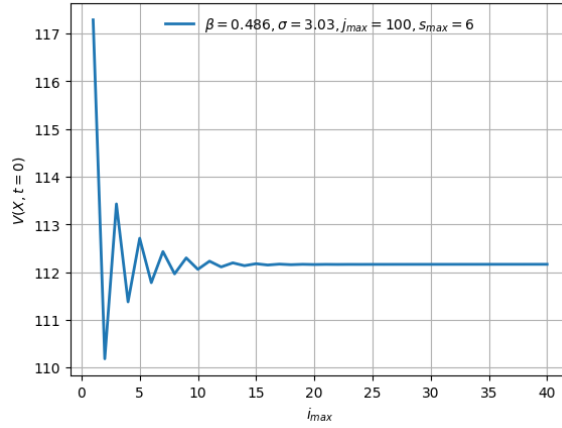


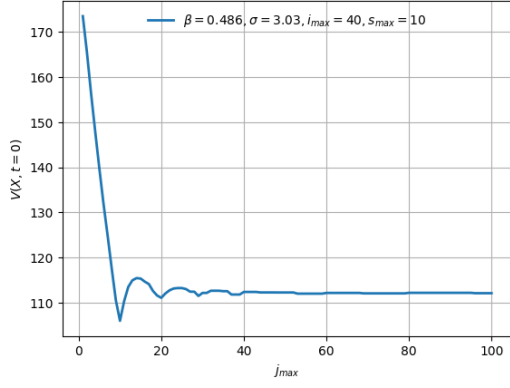
Figure 3: Trend of the convertible bond $V(S = X, t = T)$ as parameter i_{max} is varied.

Finally, for this section the parameters i_{max} , j_{max} and S_{max} were investigated to study how a variation in their value affected the result. The region selected was the at-the-money $S = X$ region of stock price to have comparable results across all three parameters. Starting with the variation in the time steps, Figure 3 illustrates such relationship. Here, it is clear that increasing i_{max} rapidly converges towards a single value of $V(X, T)$ and after $i_{max} = 25$ there is really no point in increasing this parameter too much.

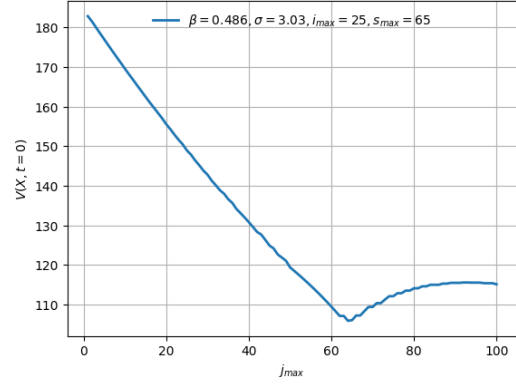
When it came to varying j_{max} which is the number of steps in S per timestep, it was noticed that since the stepsize in S is calculated by dividing S_{max} by the number of steps then these had to go hand in hand when varying one of them. Figure 4 illustrates this very clearly. Keeping the range of j_{max} the same and increasing S_{max} shows the same plot but being stretched out in the x-axis.

This happens since increasing the maximum cutoff S from which to start at each timestep but keeping the number of steps constant would mean larger jumps thus a less accurate result everytime. Instead, ensuring that the overall stepsize in S is constant or small enough is paramount in keeping the result accurate. Recall that the error in the Crank-Nicolson method is $\mathcal{O}(\Delta S^2, \Delta t^2)$. Moreover, from these plots we can infer that increasing S_{max} is pointless (performance-wise) beyond a certain point since we will get the same result by taking more time.

The last issue left to investigate was the time requirements and processing complexity of varying these parameters. As can be inferred from Figure 5 i_{max} follows a linear time increase while j_{max} is much higher order. This is because of the fact that a single loop from time $t = T$ to $t = 0$ is done but a further loop of j_{max} length is done per time step. Since the error of Crank-Nicolson is given by $\mathcal{O}(\Delta S^2, \Delta t^2)$ both quantities are important and are dependent directly on i_{max} and j_{max} .



(a) Stability and convergence can be observed after $j_{max} = 40$



(b) The plot from 4a is stretched and since S_{max} is x6.5 as much, the first minimum is also stretched by that much.

Figure 4: Plots of the price of the convertible bond $V(X, T)$ against changing j_{max} for different values of S_{max} .

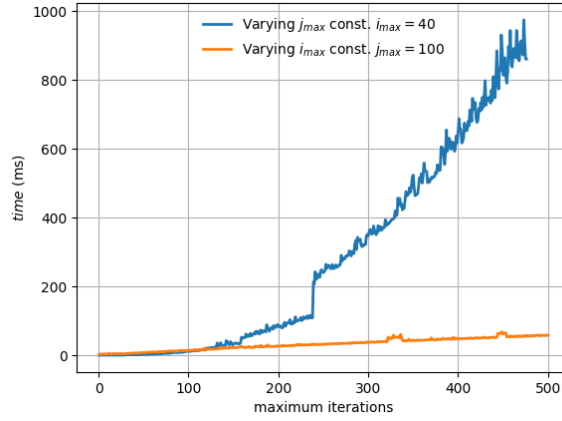


Figure 5: Variation of the time required to calculate the convertible bond price as j_{max} and i_{max} are varied.

Table 1 shows the converging trend and rates of convergence for the algorithm used. The convergence rate is linear with timescale (unlike quadratic as expected from Crank-Nicolson) and reflects upon the fact that this contract is more complex than, for example, a simple American option. Errors such as interpolation (here linear was used since Lagrange with $n = 4$ was found to be problematic at higher values of N) could be the source of such a worse convergence rate than expected.

N	$V(X, 0)$	Iters	Diff.Ratio	Time(ms)
100	112.1147738	3791		17
200	112.1652622	4320		63
400	112.1657249	5410	109	239
800	112.1659555	7440	2.01	1023
1600	112.1660718	11430	1.98	3896
3200	112.1661302	26950	1.99	15129

Table 1: Table showing convergence results and rates of PSOR method with a Crank-Nicolson numerical scheme. Here, $j_{max} = i_{max} = N$ and $S_{max} = NX/30$

Thus, the final value for $\sigma = 3.03$ and $\beta = 0.486$ was calculated to be $V(S = X, t = 0) = 112.166$ with $S_{max} = 58X$, $j_{max} = 800$, $i_{max} = 800$.

2 American Type Option Convertible Bond with Embedded Option

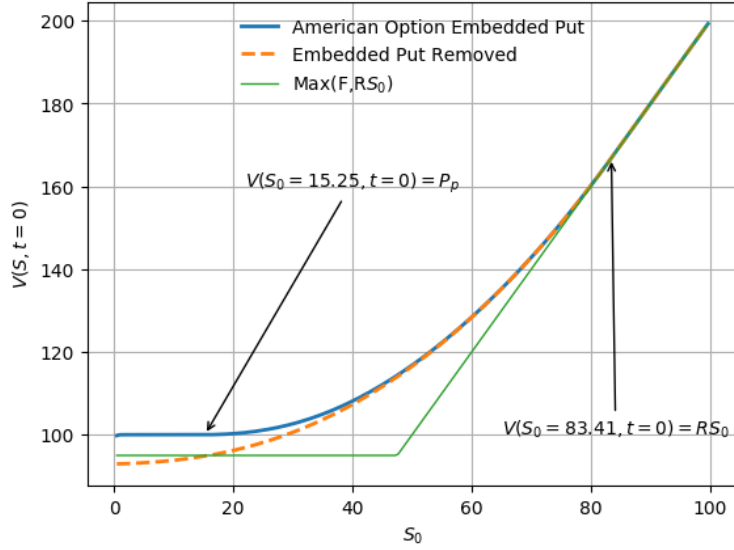


Figure 6: Value of the american type option convertible bond $V(S, t = 0)$ against initial underlying asset price S_0 with and without the embedded put option.

For this section the numerical method used was Crank-Nicolson Method with Penalty method and Thomas method for the matrix equation solver. The extension to the european option type convertible bond detailed in Section 1 is to change it to an American type option. By this we mean that the holder has the option to convert the bond in stock at any time before the maturity of the contract. To ensure this, the inequality

$$V \geq RS \quad (25)$$

must hold for all $t < T$. This means the $V_{american} > V_{european}$. A last addition is to embed a put option in this contract which means the holder has the option to sell the bond back to the issuer over some time period such that

$$V(S, t) \geq P_p \quad \text{for} \quad t \leq t_0 \quad (26)$$

must hold.

Figure 6 shows the results of adding these conditions in the code. The limit for large S is observed as expected to tend to RS and compared to Figure 1 the value of the option is higher. This is due to the fact that the effective increase in power given to the holder increases the price. Furthermore, adding the put option increases further the price since again this gives more power to the holder. This put option might be a sort of safety net in case the value of the stock decreases too much and as with most financial contracts, a decrease in risk must increase the price. The bond floor is thus observed to be raised when compared to the no-option case.

Finally, the arrows are pointing to two decision points at which the price of the contract becomes more than P_p thereafter and becomes more than RS_0 thereafter, respectively. These are important points since the holder would only ever buy the contract for values of S_0 between those two points otherwise they would just buy the contract to sell it again or would buy the underlying equity.

The sensitivity to the mean reversion rate [4] κ was studied. Referring back to Equation 24, this is the rate at which the stock will revert back to the long term mean price described by $\theta(t)$. As can be seen from Figure 7 an increase in κ decreases the value of the bond in the at-the-money region of the underlying stock. This is expected since less fluctuations in the stock price movements make it less attractive to buy this contract due to the probabilities of the stock price changing drastically in the

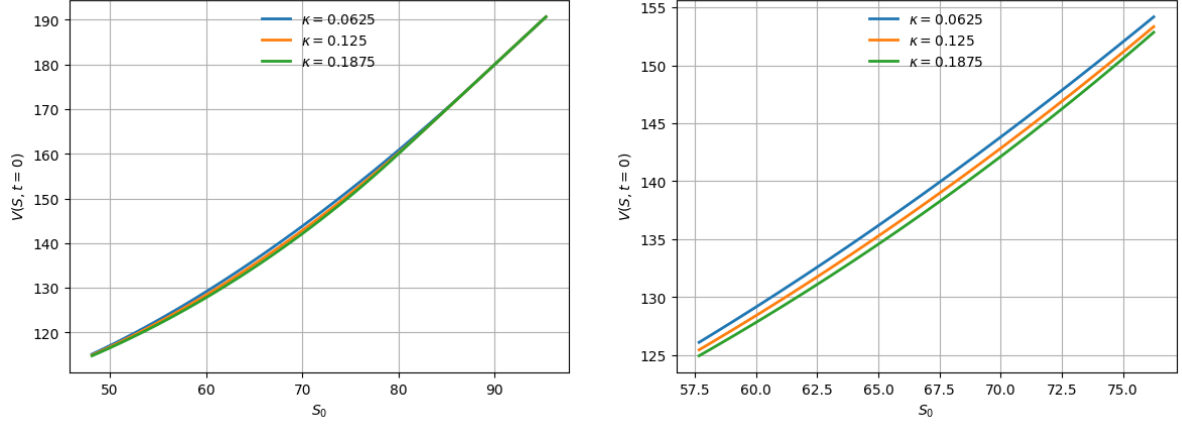


Figure 7: Value of the american type option convertible bond $V(S, t = 0)$ with embedded put option for different values of parameter κ . Right side plot is a zoomed version of the left side plot.

future being lower thus the convertibility of the bond being unused.

Lastly, it was requested to obtain the most accurate value possible in one second of processing. Referring to accuracy first, the important thing to eliminate is any source of error. There are errors on the boundary (finite-element) and errors in discontinuities of the domain. Interpolation errors were minimised by using Lagrange interpolation of order 4. Since there is a discontinuity at time t_0 one must change the timestep used before and now split the domain effectively in two. The timestep was chosen such that

$$\Delta t = \frac{T - t_0}{i_{max} f} \quad \text{for} \quad t_0 < t \leq T \quad (27)$$

$$\Delta t = \frac{t_0}{i_{max}(1 - f)} \quad \text{for} \quad 0 < t \leq t_0 \quad (28)$$

where $f = \frac{T - t_0}{T}$. A better timestepping system would have been Rannacher smoothing [5] however this was beyond the scope of the analysis and as such literature was used to check that the expected convergence ratios were in the right ranges. The results, comparing the method PSOR to the penalty method are detailed in Table 2.

N	Penalty				PSOR			
	$V(X, 0)$	Iters	Diff.Ratio	Time(ms)	$V(X, 0)$	Iters	Diff.Ratio	Time(ms)
100	114.5317065	126		11	114.5283612	4606		22
200	114.5067677	225		34	114.5034113	5280		56
400	114.4891972	427	1.42	130	114.4858357	6389	1.42	163
800	114.4804949	827	2.02	554	114.4771308	8780	2.02	516
1600	114.4776084	1627	3.01	1594	114.474243	12808	3.01	1967
3200	114.476139	3227	1.96	6210	114.4727731	19206	1.96	6599
6400	114.4753978	8838.6	1.98	24269	114.4720314	28648	1.98	24556

Table 2: Table comparing convergence results and efficiencies of PSOR and Penalty methods with a Crank-Nicolson numerical scheme. Here, $j_{max} = i_{max} = N$ and $S_{max} = NX/30$. Important to note here is the amount of time and iterations saved via the Penalty method.

The results show the convergence rate is nowhere near square with timestep as it should be for a crank nicolson scheme but rather linear. This is may be due to the increased complexity the convertibility in the whole life of the bond brings with it. Because of the smoothing method taken combined with other errors such as interpolation errors the rate is less than optimal however it is within expected ranges [6]. Finally, using the information from Table 2 the most accurate value in the given time was found to be: $V(X, t_0 = 0) = 114.479937$ in 965ms with $j_{max} = i_{max} = 1300$ and $s_{max} = 30X$.

References

- [1] C. Thierfelder, “The trending ornstein-uhlenbeck process and its applications in mathematical finance,” *Mathematical Finance*, 2015.
- [2] V. Linetsky and R. Mendoza, *Constant Elasticity of Variance (CEV) Diffusion Model*. American Cancer Society, 2010.
- [3] N. H. Chan and C. T. Ng, *Fractional constant elasticity of variance model*, vol. Volume 52 of *Lecture Notes–Monograph Series*, pp. 149–164. Beachwood, Ohio, USA: Institute of Mathematical Statistics, 2006.
- [4] M. Choudhry, “51 - interest-rate models i,” in *The Bond and Money Markets*, Securities Institution Professional Reference Series, pp. 873 – 887, Oxford: Butterworth-Heinemann, 2001.
- [5] P. A. Forsyth and K. R. Vetzal, “Quadratic convergence for valuing american options using a penalty method,” *SIAM Journal on Scientific Computing*, vol. 23, no. 6, pp. 2095–2122, 2002.
- [6] L. X. Li, *Pricing Convertible Bonds using Partial Differential Equations*. University of Toronto, 2005.

Appendix 1: European Type Option Code

Portfolio Pricing Program Listing

```
1 #include <iostream>
2 #include <fstream>
3 #include <cmath>
4 #include <vector>
5 #include <algorithm>
6 #include <chrono>
7 #include <iomanip>
8 using namespace std;
9
10 /* Code for the Crank Nicolson Finite Difference
11 */
12 double crank_nicolson(double S0, double X, double F, double T, double r,
13     double sigma,
14     double R, double kappa, double mu, double C, double
15     alpha, double beta, int iMax, int jMax, int S_max, double tol, double
16     omega, int iterMax, int &sorCount)
17 {
18     // declare and initialise local variables (ds,dt)
19     double dS = S_max / jMax;
20     double dt = T / iMax;
21     // create storage for the stock price and option price (old and new)
22     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
23     // setup and initialise the stock price
24     for (int j = 0; j <= jMax; j++)
25     {
26         S[j] = j * dS;
27     }
28     // setup and initialise the final conditions on the option price
29     for (int j = 0; j <= jMax; j++)
30     {
31         vOld[j] = max(F, R * S[j]);
32         vNew[j] = max(F, R * S[j]);
33     }
34     // start looping through time levels
35     for (int i = iMax - 1; i >= 0; i--)
36     {
37         // declare vectors for matrix equations
38         vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
39         // set up matrix equations a[j]=
40         double theta = (1 + mu) * X * exp(mu * i * dt);
41         a[0] = 0;
42         b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
43         c[0] = (kappa * theta / dS);
44         d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-1 / dt) + (r / 2));
45         for (int j = 1; j <= jMax - 1; j++)
46         {
47             //
48             a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (
49                 kappa * (theta - j * dS) / (4 * dS));
50             b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. *
51                 pow(dS, 2))) - (r / 2.);
52             c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.)
53                 )) + ((kappa * (theta - j * dS)) / (4. * dS));
54             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));
}
49
50 double A = R * exp((kappa + r) * (i * dt - T));
51 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);
52 a[jMax] = 0;
53 b[jMax] = 1;
54 c[jMax] = 0;
55 d[jMax] = jMax * dS * A + B;
56
57 // solve matrix equations with SOR
58 int sor;
59 for (sor = 0; sor < iterMax; sor++)
60 {
61     double error = 0.;
62     // implement sor in here
63     {
64         double y = (d[0] - c[0] * vNew[1]) / b[0];
65         y = vNew[0] + omega * (y - vNew[0]);
66         error += (y - vNew[0]) * (y - vNew[0]);
67         vNew[0] = y;
68     }
69     for (int j = 1; j < jMax; j++)
70     {
71         double y = (d[j] - a[j] * vNew[j - 1] - c[j] * vNew[j + 1]) / b[j];
72         y = vNew[j] + omega * (y - vNew[j]);
73         error += (y - vNew[j]) * (y - vNew[j]);
74         vNew[j] = y;
75     }
76     {
77         double y = (d[jMax] - a[jMax] * vNew[jMax - 1]) / b[jMax];
78         y = vNew[jMax] + omega * (y - vNew[jMax]);
79         error += (y - vNew[jMax]) * (y - vNew[jMax]);
80         vNew[jMax] = y;
81     }
82     // make an exit condition when solution found
83     if (error < tol * tol)
84     {
85         sorCount += sor;
86         break;
87     }
88 }
89 if (sor >= iterMax)
90 {
91     std::cout << " Error NOT converging within required iterations\n";
92     std::cout.flush();
93     throw;
94 }
95 vOld = vNew;
96 }
97 // finish looping through time levels
98
99 // output the estimated option price
100 double optionValue;
101 //linear interp

```

```

102     int jStar = S0 / dS;
103     double sum = 0.;
104     sum += (S0 - S[jStar]) / (dS)*vNew[jStar + 1];
105     sum += (S[jStar + 1] - S0) / (dS)*vNew[jStar];
106     optionValue = sum;
107     // alternatively
108     //optionValue = lagrangeInterpolation(vNew, S, S0, 4);
109     return optionValue;
110 }
111
112 int main()
113 {
114     // Initial condition
115     double T = 2., F = 95., R = 2., r = 0.0229, kappa = 0.125, altSigma =
        0.416,
116         mu = 0.0213, X = 47.66, C = 1.09, alpha = 0.02, beta = 0.486,
        sigma = 3.03, tol = 1.e-8, omega = 1.4;
117
118     int iterMax = 100000, iMax = 200, jMax = 200, S_max = 6 * X;
119     int length = 300;
120     double S_range = 3 * X;
121     int sor;
122
123     // Run to obtain 3d graph
124     std::ofstream outFile9("./data/varying_s_sigma_beta.csv");
125     for (double altSigma = 0; altSigma < 3.5; altSigma += 0.1)
126     {
127         for (double beta = 0; beta < 1.3; beta += 0.1)
128         {
129             double S0 = X;
130             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";
131         }
132     }
133     outFile9.close();
134
135     // Run to obtain varying configurations of beta, sigma graph
136     std::ofstream outFile1("./data/varying_s_beta_1.csv");
137     std::ofstream outFile2("./data/varying_s_beta_0.4.csv");
138     for (int j = 1; j <= length - 1; j++)
139     {
140         vector<double> gamma(jMax + 1);
141         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) << "\n";
142         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) << "\n";
143     }
144     outFile1.close();
145     outFile2.close();
146
147     // Run to obtain varying imax graph
148     std::ofstream outFile3("./data/varying_imax.csv");
149     jMax = 100;
150     for (iMax = 1; iMax <= 500; iMax += 1)
151     {

```

```

152     double S = X;
153     auto t1 = std::chrono::high_resolution_clock::now();
154     double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
155     alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
156     auto t2 = std::chrono::high_resolution_clock::now();
157     auto time_taken =
158         std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
159         .count();
160     outFile3 << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
161     std::fixed << result << "," << time_taken << "\n";
162 }
163 outFile3.close();
164
165 // Run to obtain varying smax per varying jmax graph
166 for (int s_Mult = 10; s_Mult <= 10; s_Mult += 1)
167 {
168     double S = X;
169     S_max = s_Mult * X;
170     string title = "./data/smax_jmax/" + to_string(s_Mult) + "_varying_jmax
171     .csv";
172     std::ofstream outFile4(title);
173     iMax = 40;
174     for (jMax = 1; jMax <= 500; jMax += 1)
175     {
176         auto t1 = std::chrono::high_resolution_clock::now();
177         double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
178         alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
179         auto t2 = std::chrono::high_resolution_clock::now();
180         auto time_taken =
181             std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
182             .count();
183         outFile4 << S_max << "," << iMax << "," << jMax << "," << S << " , "
184         << std::fixed << result << "," << time_taken << "\n";
185     }
186     outFile4.close();
187 }
188
189 //Run to obtain graph of varying smax
190 std::ofstream outFile5("./data/varying_smax.csv");
191 for (int s_Mult = 10; s_Mult <= 50; s_Mult += 1)
192 {
193     jMax = s_Mult * 10;
194     double S = X;
195     S_max = s_Mult * X;
196     int sorCount;
197     auto t1 = std::chrono::high_resolution_clock::now();
198     double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
199     alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount);
200     auto t2 = std::chrono::high_resolution_clock::now();
201     auto time_taken =
202         std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
203         .count();
204     outFile5 << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
205     std::fixed << result << "," << time_taken << "\n";
206 }
207 outFile5.close();
208
209 double oldResult = 0;

```

```

203 double oldDiff = 0;
204 for (int N = 100; N < 3200; N *= 2)
205 {
206     jMax = N;
207     iMax = N;
208     double S = X;
209     double S_max = int(N / 30) * X;
210     int sorCount{0};
211     auto t1 = std::chrono::high_resolution_clock::now();
212     double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
213     alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount);
214     double diff = result - oldResult;
215     auto t2 = std::chrono::high_resolution_clock::now();
216     auto time_taken =
217         std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
218         .count();
219     cout << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
220     setprecision(10) << result << "," << time_taken << "," << setprecision
221     (3) << oldDiff / diff << "," << sorCount << "\n";
222     oldDiff = diff;
223     oldResult = result;
224 }
225
226 // Run to obtain graph to check with analytic value
227 std::ofstream outFile6("./data/analytic.csv");
228 S_max = 6 * X;
229 iMax = 200, jMax = 200;
230 for (int j = 1; j <= length - 1; j++)
231 {
232     outFile6 << j * S_range / length << " , " << crank_nicolson(j * S_range
233     / length, X, F, T, r, altSigma, R, 0, mu, C, alpha, 1, iMax, jMax,
234     S_max, tol, omega, iterMax, sor) << "\n";
235 }
236 outFile6.close();
237
238 // Run to obtain final accurate value
239 S_max = 58 * X;
240 iMax = 800, jMax = 800;
241 double S0 = X;
242 auto t1 = std::chrono::high_resolution_clock::now();
243 double result = crank_nicolson(S0, X, F, T, r, sigma, R, kappa, mu, C,
244     alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
245 auto t2 = std::chrono::high_resolution_clock::now();
246 auto time_taken =
247     std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
248     .count();
249 std::cout << setprecision(10) << result << " , " << time_taken << endl;
250 }

```

Graphing Program Listing

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import csv
4 import scipy.stats as si
5
6 X=47.66
7 R=2
8 F=95

```

```

9 T=2.0
10 C = 1.09
11 alpha = 0.02
12 r = 0.0229
13 T = 2.
14 sigma = 0.416
15
16 def euro_vanilla_call(S, K, T, r, sigma):
17
18     #S: spot price
19     #K: strike price
20     #T: time to maturity
21     #r: interest rate
22     #sigma: volatility of underlying asset
23
24     d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
25 )
26     d2 = (np.log(S / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
27 )
28
29     call = (R*S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si.norm.
30 cdf(d2, 0.0, 1.0))
31
32     return call
33
34 variationData=[]
35 with open('data/varying_imax.csv', newline='\n') as csvfile:
36     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V
37 '], quoting=csv.QUOTE_NONNUMERIC)
38     currentData={'x':[], 'y':[], 'jMax':0, 'sMax':0}
39     for row in reader:
40         currentData['x'].append(row['iMax'])
41         currentData['y'].append(row['V'])
42         currentData['jMax']=row['jMax']
43         currentData['sMax']=int(row['sMax']/X)
44     variationData.append(currentData)
45
46 plt.figure()
47 plt.grid()
48 plt.plot(variationData[0]['x'][:40], variationData[0]['y'][:40], label=r'$\
49 \beta=0.486, \sigma=3.03, j_{\max}=%i, s_{\max}=%i$'%(variationData[0]['jMax',
50 ], variationData[0]['sMax']), linewidth=2)
51 plt.xlabel(r'$i_{\max}$')
52 plt.ylabel(r'$V(X, t=0)$')
53 plt.legend(loc='upper center', fancybox=False, framealpha=0.0)
54 plt.savefig('images/european_varying_imax.png', bbox_inches='tight',
55 pad_inches=0.2)
56
57 with open('data/varying_smax.csv', newline='\n') as csvfile:
58     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V
59 '], quoting=csv.QUOTE_NONNUMERIC)
60     currentData={'x':[], 'y':[], 'jMax':[], 'iMax':0}
61     for row in reader:
62         currentData['x'].append(int(row['sMax']/X))
63         currentData['y'].append(row['V'])
64         currentData['jMax'].append(row['jMax'])
65         currentData['iMax']=row['iMax']
66     variationData.append(currentData)

```

```

59 fig, ax1 = plt.subplots()
60 ax1.set_xlabel(r's_{max} (multiples of X)')
61 ax1.set_ylabel(r'$V(X, t=0)$')
62 ax1.grid()
63 ax1.scatter(np.asarray(variationData[1]['x'][:20]), variationData[1]['y']
64             [:20], label=r'$V(X, t=0)$ for $\beta=0.486, \sigma=3.03, i_{\max}=%i$'%(
65             variationData[1]['iMax']))
66 ax2 = ax1.twinx()
67 ax2.set_ylabel(r'$j_{\max}$')
68 fig.tight_layout()
69 ax2.plot(np.asarray(variationData[1]['x'][:20]), variationData[1]['jMax']
70         [:20], label=r'$j_{\max}$', color="orange")
71 lines, labels = ax1.get_legend_handles_labels()
72 lines2, labels2 = ax2.get_legend_handles_labels()
73 ax2.legend(lines + lines2, labels + labels2, loc='lower right', fancybox=
74           False, framealpha=0.0)
75 plt.savefig('images/european_varying_smax_zoomed.png', bbox_inches='tight',
76           pad_inches=0.2)
77
78 fig, ax1 = plt.subplots()
79 ax1.set_xlabel(r's_{max} (multiples of X)')
80 ax1.set_ylabel(r'$V(X, t=0)$')
81 ax1.grid()
82 ax1.scatter(np.asarray(variationData[1]['x']), variationData[1]['y'], label=r
83             '$V(X, t=0)$ for $\beta=0.486, \sigma=3.03, i_{\max}=%i$'%(variationData[1][
84             'iMax']))
85 ax2 = ax1.twinx()
86 ax2.set_ylabel(r'$j_{\max}$')
87 fig.tight_layout()
88 ax2.plot(np.asarray(variationData[1]['x']), variationData[1]['jMax'], label=r
89         '$j_{\max}$', color="orange")
90 lines, labels = ax1.get_legend_handles_labels()
91 lines2, labels2 = ax2.get_legend_handles_labels()
92 ax2.legend(lines + lines2, labels + labels2, loc='lower right', fancybox=
93           False, framealpha=0.0)
94 plt.savefig('images/european_varying_smax.png', bbox_inches='tight',
95           pad_inches=0.2)
96
97 for smax in (10, 65):
98     currentData = {'x': [], 'y': [], 'iMax': 0, 'sMax': 0}
99     with open('data/smax_jmax/' + str(smax) + '_varying_jmax.csv', newline='\n'
100             ) as csvfile:
101         reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S
102             ', 'V'], quoting=csv.QUOTE_NONNUMERIC)
103         for row in reader:
104             currentData['x'].append(row['jMax'])
105             currentData['y'].append(row['V'])
106             currentData['iMax'] = row['iMax']
107             currentData['sMax'] = int(row['sMax']/X)
108
109     plt.figure()
110     plt.plot(currentData['x'][:100], currentData['y'][:100], label=r'$\beta
111             =0.486, \sigma=3.03, i_{\max}=%i, s_{\max}=%i$'%(currentData['iMax'],
112             currentData['sMax']), linewidth=2)
113     plt.xlabel(r'$j_{\max}$')
114     plt.ylabel(r'$V(X, t=0)$')
115     plt.legend(loc='upper center', fancybox=False, framealpha=0.0)

```

```

103     plt.grid()
104     plt.savefig('images/smax_jmax/' + str(smax) + '_european_varying_jmax.png',
105               bbox_inches='tight', pad_inches=0.2)
106     plt.close()
107
108 allData=[]
109 with open('data/varying_s_beta_1.csv', newline='\n') as csvfile:
110     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
111     QUOTENONNUMERIC)
112     currentData={'S':[], 'V':[]}
113     for row in reader:
114         currentData['S'].append(row['S'])
115         currentData['V'].append(row['V'])
116     allData.append(currentData)
117
118 with open('data/varying_s_beta_0_4.csv', newline='\n') as csvfile:
119     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
120     QUOTENONNUMERIC)
121     currentData={'S':[], 'V':[]}
122     for row in reader:
123         currentData['S'].append(row['S'])
124         currentData['V'].append(row['V'])
125     allData.append(currentData)
126
127 plt.figure()
128 plt.grid()
129 plt.plot(allData[0]['S'], allData[0]['V'], label=r'$\beta=1, \sigma=0.416$',
130         linewidth=2)
131 plt.plot(allData[1]['S'], allData[1]['V'], label=r'$\beta=0.486, \sigma=3.03$',
132         linewidth=2)
133 plt.xlabel(r'$S_0$')
134 plt.ylabel(r'$V(S, t=0)$')
135 plt.legend(loc='upper center', fancybox=False, framealpha=0.0)
136 plt.savefig('images/european_varying_s.png', bbox_inches='tight', pad_inches
137           =0.2)
138
139 import matplotlib.pyplot as plt
140 from mpl_toolkits.mplot3d import Axes3D
141
142 currentData={'S':[], 'V':[], 'beta':[], 'sigma':[]}
143 with open('data/varying_s_sigma_beta.csv', newline='\n') as csvfile:
144     reader = csv.DictReader(csvfile, fieldnames=['beta', 'sigma', 'S', 'V'],
145     quoting=csv.QUOTENONNUMERIC)
146     for row in reader:
147         if (row['V']==-1):
148             continue
149         currentData['S'].append(row['S'])
150         currentData['V'].append(row['V'])
151         currentData['beta'].append(row['beta'])
152         currentData['sigma'].append(row['sigma'])
153
154 fig = plt.figure()
155 ax = fig.add_subplot(111, projection='3d')
156 ax.scatter(currentData['sigma'], currentData['beta'], currentData['V'])
157 ax.set_xlabel(r'$\sigma$')
158 ax.set_ylabel(r'$\beta$')
159 ax.set_zlabel(r'$V(X, t=0)$')

```



```

154 plt.show()
155 plt.savefig('images/european_varying_s_varying_sigma_varying_beta.png',
            bbox_inches='tight', pad_inches=0.2)
156
157 plt.figure()
158 plt.grid()
159 plt.xlabel(r'maximum iterations')
160 plt.ylabel(r'$time$ (ms)')
161
162 currentData={'x':[], 'y':[], 'iMax':0, 'sMax':0, 'time':[]}
163 with open('data/smax_jmax/10_varying_jmax.csv', newline='\n') as csvfile:
164     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V',
165     'time'], quoting=csv.QUOTE_NONNUMERIC)
166     for row in reader:
167         currentData['x'].append(row['jMax'])
168         currentData['y'].append(row['V'])
169         currentData['iMax']=row['iMax']
170         currentData['sMax']=int(row['sMax']/X)
171         currentData['time'].append(row['time'])
172
173 plt.plot(currentData['x'], currentData['time'], label=r'Varying $j_{\max}$
174         const. $i_{\max}=40$', linewidth=2)
175
176
177 variationData=[]
178 currentData={'x':[], 'y':[], 'jMax':0, 'sMax':0, 'time':[]}
179
180 with open('data/varying_imax.csv', newline='\n') as csvfile:
181     reader = csv.DictReader(csvfile, fieldnames=['sMax', 'iMax', 'jMax', 'S', 'V',
182     'time'], quoting=csv.QUOTE_NONNUMERIC)
183     for row in reader:
184         currentData['x'].append(row['iMax'])
185         currentData['y'].append(row['V'])
186         currentData['jMax']=row['jMax']
187         currentData['sMax']=int(row['sMax']/X)
188         currentData['time'].append(row['time'])
189
190 plt.plot(currentData['x'], currentData['time'], label=r'Varying $i_{\max}$
191         const. $j_{\max}=100$', linewidth=2)
192
193 plt.legend(loc='upper center', fancybox=False, framealpha=0.0)
194 plt.savefig('images/european_time.png', bbox_inches='tight', pad_inches=0.2)

```

Appendix 2: American Type Option Code

Portfolio Pricing Program Listing

```
1 #include <iostream>
2 #include <fstream>
3 #include <cmath>
4 #include <vector>
5 #include <algorithm>
6 #include <chrono>
7 #include <iomanip>
8
9 using namespace std;
10
11 double lagrangeInterpolation(const vector<double> &y, const vector<double>
    &x, double x0, unsigned int n)
12 {
13     if (x.size() < n)
14         return lagrangeInterpolation(y, x, x0, x.size());
15     if (n == 0)
16         throw;
17     int nHalf = n / 2;
18     int jStar;
19     double dx = x[1] - x[0];
20     if (n % 2 == 0)
21         jStar = int((x0 - x[0]) / dx) - (nHalf - 1);
22     else
23         jStar = int((x0 - x[0]) / dx + 0.5) - (nHalf);
24     jStar = std::max(0, jStar);
25     jStar = std::min(int(x.size() - n), jStar);
26     if (n == 1)
27         return y[jStar];
28     double temp = 0.;
29     for (unsigned int i = jStar; i < jStar + n; i++)
30     {
31         double int_temp;
32         int_temp = y[i];
33         for (unsigned int j = jStar; j < jStar + n; j++)
34         {
35             if (j == i)
36             {
37                 continue;
38             }
39             int_temp *= (x0 - x[j]) / (x[i] - x[j]);
40         }
41         temp += int_temp;
42     }
43     // end of interpolate
44     return temp;
45 }
46
47 std::vector<double> thomasSolve(const std::vector<double> &a, const std:::
    vector<double> &b_, const std::vector<double> &c, std::vector<double> &d
    )
48 {
49     int n = a.size();
50     std::vector<double> b(n), temp(n);
51     // initial first value of b
```

```

52 b[0] = b_[0];
53 for (int j = 1; j < n; j++)
54 {
55     b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
56     d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
57 }
58 // calculate solution
59 temp[n - 1] = d[n - 1] / b[n - 1];
60 for (int j = n - 2; j >= 0; j--)
61     temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];
62 return temp;
63 }
64
65 /* Code for the Crank Nicolson Finite Difference with Penalty
66 */
67 double crank_nicolson1(double S0, double X, double F, double T, double r,
68     double sigma,
69     double R, double kappa, double mu, double C, double
70     alpha, double beta, int iMax, int jMax, int S_max, double tol, double
71     omega, int iterMax, int &sorCount, double t0)
69 {
70     // declare and initialise local variables (ds,dt)
71     double P = 100.;
72     double dS = S_max / jMax;
73     double f = (T - t0) / T;
74     double dt = (T - t0) / (iMax * f);
75     // create storage for the stock price and option price (old and new)
76     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
77     // setup and initialise the stock price
78     for (int j = 0; j <= jMax; j++)
79     {
80         S[j] = j * dS;
81     }
82     // setup and initialise the final conditions on the option price
83     for (int j = 0; j <= jMax; j++)
84     {
85         vOld[j] = max(F, R * S[j]);
86         vNew[j] = max(F, R * S[j]);
87     }
88     // start looping through time levels
89     for (int i = iMax; i >= 0; i--)
90     {
91
92         if (i * dt < t0)
93         {
94             dt = t0 / (iMax * (1 - f));
95         }
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))) - (
109 kappa * (theta - j * dS) / (4 * dS));
110 b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. *
111 pow(dS, 2))) - (r / 2.);
112 c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))
113 ) + ((kappa * (theta - j * dS)) / (4. * dS));
114 d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) /
115 (4. * pow(dS, 2.))) * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((
116 kappa * (theta - j * dS)) / (4. * dS)) * (vOld[j + 1] - vOld[j - 1])) +
117 ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
118 }
119 double A = R * exp((kappa + r) * (i * dt - T));
120 double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R *
121 exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt) / (alpha
122 + r);
123 a[jMax] = 0;
124 b[jMax] = 1;
125 c[jMax] = 0;
126 d[jMax] = jMax * dS * A + B;
127 double penalty = 1.e8;
128 int q;
129 for (q = 0; q < 100000; q++)
130 {
131     vector<double> bHat(b), dHat(d);
132     for (int j = 1; j < jMax; j++)
133     {
134         if (i * dt < t0)
135         {
136             if (vNew[j] < max(R * S[j], P))
137             {
138                 bHat[j] = b[j] - penalty;
139                 dHat[j] = d[j] - penalty * max(R * S[j], P);
140             }
141         }
142         else
143         {
144             // turn on penalty if V < RS
145             if (vNew[j] < R * S[j])
146             {
147                 bHat[j] = b[j] - penalty;
148                 dHat[j] = d[j] - penalty * R * S[j];
149             }
150         }
151     }
152     // solve matrix equations with SOR
153     vector<double> y = thomasSolve(a, bHat, c, dHat);
154     // calculate difference from last time
155     double error = 0.;
156     for (int j = 0; j <= jMax; j++)
157         error += fabs(vNew[j] - y[j]);
158     vNew = y;
159     if (error < 1.e-8)
160     {
161         sorCount += q;
162         break;
163     }
164 }

```

```

157     if (q == 100000)
158     {
159         std::cout << " Error NOT converging within required iterations\n";
160         std::cout.flush();
161         throw;
162     }
163
164     // set old=new
165     vOld = vNew;
166 }
167 // finish looping through time levels
168
169 // output the estimated option price
170 double optionValue;
171
172 int jStar = S0 / dS;
173 double sum = 0.;
174 sum += (S0 - S[jStar]) / (dS)*vNew[jStar + 1];
175 sum += (S[jStar + 1] - S0) / dS * vNew[jStar];
176 optionValue = sum;
177
178 //optionValue = lagrangeInterpolation(vNew, S, S0, 4);
179
180 return optionValue;
181 }
182 /* Code for the Crank Nicolson Finite Difference with PSOR
183 */
184 double crank_nicolson2(double S0, double X, double F, double T, double r,
185     double sigma,
186     double R, double kappa, double mu, double C, double
187     alpha, double beta, int iMax, int jMax, int S_max, double tol, double
188     omega, int iterMax, int &sorCount, double t0)
189 {
190     // declare and initialise local variables (ds,dt)
191     double P = 100.;
192     double dS = S_max / jMax;
193     double f = (T - t0) / T;
194     double dt = (T - t0) / (iMax * f);
195     // create storage for the stock price and option price (old and new)
196     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
197     // setup and initialise the stock price
198     for (int j = 0; j <= jMax; j++)
199     {
200         S[j] = j * dS;
201     }
202     // setup and initialise the final conditions on the option price
203     for (int j = 0; j <= jMax; j++)
204     {
205         vOld[j] = max(F, R * S[j]);
206         vNew[j] = max(F, R * S[j]);
207     }
208     // start looping through time levels
209     for (int i = iMax; i >= 0; i--)
210     {
211         if (i * dt < t0)
212         {
213             dt = t0 / (iMax * (1 - f));
214         }
215     }

```

```

212 // declare vectors for matrix equations
213 vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
214 // set up matrix equations a[j]=
215 double theta = (1 + mu) * X * exp(mu * i * dt);
216 a[0] = 0;
217 b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
218 c[0] = (kappa * theta / dS);
219 d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
220 for (int j = 1; j <= jMax - 1; j++)
221 {
222     //
223     a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (
kappa * (theta - j * dS) / (4 * dS));
224     b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. *
pow(dS, 2))) - (r / 2.);
225     c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.)
)) + ((kappa * (theta - j * dS)) / (4. * dS));
226     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));
227 }
228 double A = R * exp((kappa + r) * (i * dt - T));
229 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);
230 a[jMax] = 0;
231 b[jMax] = 1;
232 c[jMax] = 0;
233 d[jMax] = jMax * dS * A + B;
234 // solve matrix equations with SOR
235 int sor;
236 for (sor = 0; sor < iterMax; sor++)
237 {
238     double error = 0.;
239     // implement sor in here
240     {
241         double y = (d[0] - c[0] * vNew[1]) / b[0];
242         y = vNew[0] + omega * (y - vNew[0]);
243         if (i * dt < t0)
244         {
245             y = std::max(std::max(y, R * S[0]), P);
246         }
247         else
248         {
249             y = std::max(y, R * S[0]);
250         }
251         error += (y - vNew[0]) * (y - vNew[0]);
252         vNew[0] = y;
253     }
254     for (int j = 1; j < jMax; j++)
255     {
256         double y = (d[j] - a[j] * vNew[j - 1] - c[j] * vNew[j + 1]) / b[j];
257         y = vNew[j] + omega * (y - vNew[j]);
258         if (i * dt < t0)
259         {
260             y = std::max(std::max(y, R * j * dS), P);
261         }

```

```

262     else
263     {
264         y = std::max(y, R * j * dS);
265     }
266     error += (y - vNew[j]) * (y - vNew[j]);
267     vNew[j] = y;
268 }
269 {
270     double y = (d[jMax] - a[jMax] * vNew[jMax - 1]) / b[jMax];
271     y = vNew[jMax] + omega * (y - vNew[jMax]);
272     if (i * dt < t0)
273     {
274         y = std::max(std::max(y, R * jMax * dS), P);
275     }
276     else
277     {
278         y = std::max(y, R * jMax * dS);
279     }
280     error += (y - vNew[jMax]) * (y - vNew[jMax]);
281     vNew[jMax] = y;
282 }
283 // make an exit condition when solution found
284 if (error < tol * tol)
285 {
286     sorCount += sor;
287     break;
288 }
289 }
290 if (sor >= iterMax)
291 {
292     std::cout << " Error NOT converging within required iterations\n";
293     std::cout.flush();
294     throw;
295 }
296
297 if (sorCount == iterMax)
298     return -1;
299
300 // set old=new
301 vOld = vNew;
302 }
303 // finish looping through time levels
304
305 // output the estimated option price
306 double optionValue;
307 /*
308 int jStar = S0 / dS;
309 double sum = 0.;
310 sum += (S0 - S[jStar]) / (dS)*vNew[jStar + 1];
311 sum += (S[jStar + 1] - S0) / dS * vNew[jStar];
312 optionValue = sum;
313 */
314 optionValue = lagrangeInterpolation(vNew, S, S0, 4);
315
316 return optionValue;
317 }
318
319 int main()

```

```

320 {
321     double T = 2., F = 95., R = 2., r = 0.0229, kappa = 0.125, altSigma =
        0.416,
322     mu = 0.0213, X = 47.66, C = 1.09, alpha = 0.02, beta = 0.486,
        sigma = 3.03, tol = 1.e-8, omega = 1., S_max = 13 * X;
323     int iMax = 600;
324     int jMax = 600;
325     double t0 = 0.57245;
326
327     int iterMax = 100000;
328     int length = 300;
329     double S_range = 3 * X;
330     int sor;
331
332     // Produces graph comparing embedded put option and without vs changing
        S0
333     std::ofstream outFile1("./data/no-put-american-varying-s-beta-0-4.csv");
334     std::ofstream outFile2("./data/american-varying-s-beta-0-4.csv");
335     for (int j = 1; j <= length - 1; j++)
336     {
337         std::cout << j << std::endl;
338         outFile1 << j * S_range / length << " , " << crank_nicolson1(j *
            S_range / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax,
            jMax, S_max, tol, omega, iterMax, sor, 0.) << "\n";
339         outFile2 << j * S_range / length << " , " << crank_nicolson1(j *
            S_range / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax,
            jMax, S_max, tol, omega, iterMax, sor, t0) << "\n";
340         outFile1.flush();
341         outFile2.flush();
342     }
343     outFile1.close();
344     outFile2.close();
345
346     // Produces graph for different values of kappa vs changing S0
347     std::ofstream outFile4("./data/american-varying-s-kappa-625.csv");
348     std::ofstream outFile5("./data/american-varying-s-kappa-125.csv");
349     std::ofstream outFile6("./data/american-varying-s-kappa-187.csv");
350
351     for (int j = 1; j <= length - 1; j++)
352     {
353         std::cout << j << std::endl;
354         double result1 = crank_nicolson1(j * S_range / length, X, F, T, r,
            sigma, R, 0.0625, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega,
            iterMax, sor, t0);
355         double result2 = crank_nicolson1(j * S_range / length, X, F, T, r,
            sigma, R, 0.125, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega,
            iterMax, sor, t0);
356         double result3 = crank_nicolson1(j * S_range / length, X, F, T, r,
            sigma, R, 0.1875, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega,
            iterMax, sor, t0);
357         outFile4 << j * S_range / length << " , " << result1 << "\n";
358         outFile5 << j * S_range / length << " , " << result2 << "\n";
359         outFile6 << j * S_range / length << " , " << result3 << "\n";
360         outFile4.flush();
361         outFile5.flush();
362         outFile6.flush();
363     }
364     outFile4.close();

```



```

365 outFile5.close();
366 outFile6.flush();
367
368 // Produces tables of price,value,iterations,time and convergence rate
    for
369 // comparing penalty and psor methods
370 std::ofstream outFile7("./data/american-varying-smax-penalty.csv");
371 double oldResult = 0, oldDiff = 0;
372 double S = X;
373 iMax = 100;
374 jMax = 100;
375 for (int n = 100; n <= 10000; n *= 2)
376 {
377     iMax = n;
378     jMax = n;
379     S_max = int(n / 30) * X;
380     int sorCount{0};
381     auto t1 = std::chrono::high_resolution_clock::now();
382     double result = crank_nicolson1(S, X, F, T, r, sigma, R, kappa, mu, C,
    alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount, t0);
383     double diff = result - oldResult;
384     auto t2 = std::chrono::high_resolution_clock::now();
385     auto time_taken =
386         std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
387         .count();
388     double extrap = (4 * result - oldResult) / 3.;
389     outFile7 << S_max << "," << iMax << "," << jMax << "," << S << "," <<
    setprecision(10) << result << "," << time_taken << "," << extrap << "," <<
    << setprecision(3) << oldDiff / diff << "," << sorCount << "\n";
390     oldDiff = diff;
391     oldResult = result;
392 }
393 outFile7.close();
394
395 std::ofstream outFile8("./data/american-varying-smax-sor.csv");
396 oldResult = 0;
397 oldDiff = 0;
398 S = X;
399 iMax = 100;
400 jMax = 100;
401 for (int n = 100; n <= 10000; n *= 2)
402 {
403     iMax = n;
404     jMax = n;
405     S_max = int(n / 30) * X;
406     int sorCount{0};
407     auto t1 = std::chrono::high_resolution_clock::now();
408     double result = crank_nicolson2(S, X, F, T, r, sigma, R, kappa, mu, C,
    alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount, t0);
409     double diff = result - oldResult;
410     auto t2 = std::chrono::high_resolution_clock::now();
411     auto time_taken =
412         std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
413         .count();
414     double extrap = (4 * result - oldResult) / 3.;
415     outFile8 << S_max << "," << iMax << "," << jMax << "," << S << "," <<
    setprecision(10) << result << "," << time_taken << "," << extrap << "," <<
    << setprecision(3) << oldDiff / diff << "," << sorCount << "\n";

```

```

416     oldDiff = diff;
417     oldResult = result;
418 }
419 outFile8.close();
420
421 // Produces final, most accurate value in <1s
422 double S0 = X;
423 iMax = 1300;
424 jMax = 1300;
425 S_max = 30 * X;
426 auto t1 = std::chrono::high_resolution_clock::now();
427 double result = crank_nicolson1(S0, X, F, T, r, sigma, R, kappa, mu, C,
    alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0);
428 auto t2 = std::chrono::high_resolution_clock::now();
429 auto time_taken =
430     std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
431     .count();
432 cout << fixed << result << ", " << time_taken << endl;
433 }

```

Graphing Program Listing

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import csv
4 from bisect import bisect_left, bisect_right
5 F=95.
6 C = 1.09
7 R=2.
8
9 allData=[]
10 with open('data/american_varying_s_beta_0_4.csv', newline='\n') as csvfile:
11     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
    QUOTE_NONNUMERIC)
12     currentData={'S':[], 'V':[]}
13     for row in reader:
14         currentData['S'].append(row['S'])
15         currentData['V'].append(row['V'])
16     allData.append(currentData)
17
18 with open('data/no_put_american_varying_s_beta_0_4.csv', newline='\n') as
    csvfile:
19     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
    QUOTE_NONNUMERIC)
20     currentData={'S':[], 'V':[]}
21     for row in reader:
22         currentData['S'].append(row['S'])
23         currentData['V'].append(row['V'])
24     allData.append(currentData)
25
26 plt.figure()
27 plt.grid()
28 parity = [(R*s*C)/F * 100 for s in allData[1]['S'] if (R*s*C)/F < 2 ]
29 x=[ s for s in allData[0]['S'] if s < 100]
30 parity=x
31 plt.plot(parity, allData[0]['V'][:len(parity)], label=r'American Option
    Embedded Put', linewidth=2)
32 plt.plot(parity, allData[1]['V'][:len(parity)], label=r'Embedded Put Removed',
    linewidth=2, linestyle='—')

```

```

33 equity = [ max(95., 2.*s) for s in allData[1]['S']]
34 lessC=bisect_right(allData[0]['V'], 100)
35 plt.annotate(r'$V(S_0=%.2f, t=0)=P_p$'%(parity[lessC]), xy=(parity[lessC],
    allData[0]['V'][lessC]), xytext=(22, 160), arrowprops=dict(arrowstyle="→"
    ))
36 lessRS=0
37 for i in allData[0]['V']:
38     if (allData[1]['S'][lessRS]*R>=i):
39         break
40     lessRS+=1
41 plt.annotate(r'$V(S_0=%.2f, t=0)=RS_0$'%(parity[lessRS]), xy=(parity[lessRS]
    , allData[0]['V'][lessRS]), xytext=(65, 100), arrowprops=dict(arrowstyle=
    "→"))
42 plt.plot(allData[1]['S'][:len(parity)], equity[:len(parity)], label=r'Max(F,
    R$S_0$)', linewidth=1)
43 plt.xlabel(r'$S_0$')
44 plt.ylabel(r'$V(S, t=0)$')
45 plt.legend(loc='upper center', fancybox=False, framealpha=0.0)
46 plt.savefig('images/american_varying_s.png', bbox_inches='tight', pad_inches
    =0.2)
47
48 allData=[]
49 with open('data/american_varying_s_kappa_625.csv', newline='\n') as csvfile
    :
50     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
    QUOTENONNUMERIC)
51     currentData={'S':[], 'V':[]}
52     for row in reader:
53         currentData['S'].append(row['S'])
54         currentData['V'].append(row['V'])
55     allData.append(currentData)
56 with open('data/american_varying_s_kappa_125.csv', newline='\n') as csvfile
    :
57     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
    QUOTENONNUMERIC)
58     currentData={'S':[], 'V':[]}
59     for row in reader:
60         currentData['S'].append(row['S'])
61         currentData['V'].append(row['V'])
62     allData.append(currentData)
63 with open('data/american_varying_s_kappa_187.csv', newline='\n') as csvfile
    :
64     reader = csv.DictReader(csvfile, fieldnames=['S', 'V'], quoting=csv.
    QUOTENONNUMERIC)
65     currentData={'S':[], 'V':[]}
66     for row in reader:
67         currentData['S'].append(row['S'])
68         currentData['V'].append(row['V'])
69     allData.append(currentData)
70
71 plt.figure()
72 plt.grid()
73 start=120
74 end=160
75 plt.plot(allData[0]['S'][start:end], allData[0]['V'][start:end], label=r'$ \
    kappa = 0.0625$', linewidth=2)
76 plt.plot(allData[1]['S'][start:end], allData[1]['V'][start:end], label=r'$ \
    kappa = 0.125$', linewidth=2)

```

```

77 plt.plot(allData[2]['S'][start:end],allData[2]['V'][start:end],label=r'$ \
    \kappa = 0.1875$',linewidth=2)
78 plt.xlabel(r'$S_0$')
79 plt.ylabel(r'$V(S,t=0)$')
80 plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
81 plt.savefig('images/american_varying_s_varying_k.png',bbox_inches='tight',
    pad_inches=0.2)
82
83 plt.figure()
84 plt.grid()
85 start=100
86 end=200
87 plt.plot(allData[0]['S'][start:end],allData[0]['V'][start:end],label=r'$ \
    \kappa = 0.0625$',linewidth=2)
88 plt.plot(allData[1]['S'][start:end],allData[1]['V'][start:end],label=r'$ \
    \kappa = 0.125$',linewidth=2)
89 plt.plot(allData[2]['S'][start:end],allData[2]['V'][start:end],label=r'$ \
    \kappa = 0.1875$',linewidth=2)
90 plt.xlabel(r'$S_0$')
91 plt.ylabel(r'$V(S,t=0)$')
92 plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
93 plt.savefig('images/complete_american_varying_s_varying_k.png',bbox_inches=
    'tight', pad_inches=0.2)

```