

01/04/2020

## 1 Introduction

In this report, we are asked to price a convertible bond contract in which the holder has the option to choose between receiving the principle  $F$  or alternatively  $R$  underlying stocks with price  $S$  at time  $t = T$ .

We calculate the value of this option using the finite-difference method with a Crank-Nicolson Scheme. We then explore the effects of varying  $\beta$  and  $\sigma$ . Finally we explore the effect of  $i_{max}, j_{max}$  and  $S_{max}$  on the solution and attempt to get an accurate value for the convertible bond.

we then move on to valuing American style bond contract, where the holder is able to covert the bond to stock at any time before maturity. This contract also has an embedded call option that will allow it to be called back by the issuer at a certain time, if certain conditions are met.

## 2 European Option

The market value of the European option with a continuous coupon is given by:

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

where  $\theta(t)$  is

$$\theta(t) = (1 + \mu)X e^{\mu t}. \quad (2)$$

### 2.1 Numerical Scheme

#### 2.1.1 Low S numerical scheme

The boundary condition at  $S = 0$  is:

$$\frac{\partial V}{\partial t} + \kappa\theta(t) \frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0. \quad (3)$$

We can then use the finite difference method to estimate the partial derivatives, and allow us to use a matrix method to calculate them.

When these are substituted into the PDE at  $S=0$ , we get

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

Therefore, the numerical scheme is:

$$a_0 = 0 \quad (5)$$

$$b_0 = -\frac{1}{\Delta t} - \frac{\kappa\theta(t)}{\Delta S} - \frac{r}{2} \quad (6)$$

$$c_0 = \frac{\kappa\theta(t)}{\Delta S} \quad (7)$$

$$d_0 = -\left(\frac{1}{\Delta t} - \frac{r}{2}\right)V_0^{i+1} - Ce^{-\alpha t}. \quad (8)$$

### 2.1.2 Intermediate points numerical scheme

For the intermediate points, we can use a different set of approximations for the partial derivatives. Using these estimates and putting them into the PDE, and then rearranging, we get

$$\begin{aligned}
& \left( \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \right) V_{j-1}^i + \\
& \quad \left( \frac{-1}{\Delta t} - \frac{1}{\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{r}{2} \right) V_j^i + \\
& \quad \left( \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} + \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \right) V_{j+1}^i = \\
& - \left( \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \right) V_{j-1}^{i+1} - \\
& \quad \left( \frac{-1}{2\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{r}{2} + \frac{1}{\Delta t} \right) V_j^{i+1} - \\
& \quad \left( \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} + \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \right) V_{j+1}^{i+1} \\
& - Ce^{-\alpha t}. \quad (9)
\end{aligned}$$

We can again extract the numerical scheme from this

$$a_j = \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \quad (10)$$

$$b_j = \frac{-1}{\Delta t} - \frac{1}{2\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{r}{2} \quad (11)$$

$$c_j = \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} + \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \quad (12)$$

$$\begin{aligned}
d_j = & - \left( \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \right) V_{j-1}^{i+1} - \\
& \left( \frac{-1}{2\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} - \frac{r}{2} + \frac{1}{\Delta t} \right) V_j^{i+1} - \\
& \left( \frac{1}{4\Delta S^2} \sigma^2 (j\Delta S)^{2\beta} + \frac{1}{4\Delta S} \kappa(\theta(t) - j\Delta S) \right) V_{j+1}^{i+1} \\
& - Ce^{-\alpha t}. \quad (13)
\end{aligned}$$

### 2.1.3 Large S limit

At very large  $S$ , the PDE simplifies slightly to

$$\frac{\partial V}{\partial t} + \kappa(X - S) \frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0. \quad (14)$$

In the case of large  $S$ , we can assume the solution to the equation is of the form

$$V(S, t) = SA(t) + B(t). \quad (15)$$

By substituting this in to equation 4, we can extract the form of  $A$  and  $B$ . Through this, we calculated

$$A(t) = Re^{(\kappa+r)(t-T)} \quad (16)$$

$$B(t) = -XA(t) + \frac{C}{\alpha + r} e^{-\alpha t} - \frac{C}{\alpha + r} e^{-(\alpha+r)T+rt} + XRe^{r(t-T)} \quad (17)$$

For the upper bound  $S = S_{max}$  we have found the analytical solution and from this we can calculate the upper bound.

$$a_{j_{max}} = 0 \quad (18)$$

$$b_{j_{max}} = 1 \quad (19)$$

$$c_{j_{max}} = 0 \quad (20)$$

$$d_j = A(t)S + B(t). \quad (21)$$

## 2.2 Convertible bond value as a function of underlying asset price for two cases

We explore the effect of varying the underlying asset price for different  $\beta$  and  $\sigma$ .

We can resolve the convertible bond into three parts. The coupon payment, modelled in continuous time, the bond part and the stock part. Ignoring the coupon payment temporarily, the final payment can be written as

$$V(S, T) = \max(F, RS). \quad (22)$$

This can be rewritten by splitting it into a bond and call option.

$$V(S, T) = N + R\max(0, S - C_p). \quad (23)$$

where the strike  $C_p$  of each call option is  $F/R$ .

A graph for convertible bond price against share price would show a linear behaviour for high share price levels. This is because it becomes much more likely the holder will want the shares, and therefore behaves more like a call option.

At low share prices, the holder of the bond will likely not convert it to stock, and therefore it acts like a simple bond. At low share prices, the value of the convertible bond approaches the bond floor, which is the sum of the discounted cash flows distributed by the bond [2].

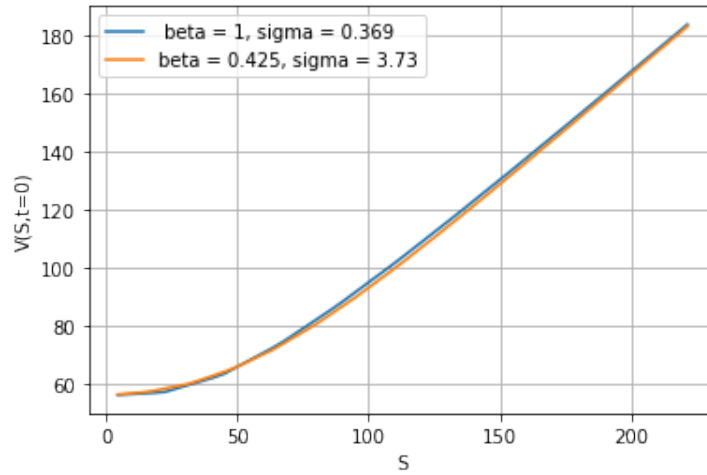


Figure 1: A graph showing relationship between the convertible bond value against stock price for two different  $\sigma$  and  $\beta$ .

Figure 1 shows the value of the convertible bond as a function of the underlying stock price for two different parameter values. This shows the bond behaviour discussed earlier. It also shows the different parameter configurations produce the same underlying behaviour. First we will explore the meaning behind these two parameters.

$\sigma$  is known as the implied volatility the Black-Scholes model. It is a measure of the future variability of the stock the option is modelled off. However in the case of this convertible bond, the underlying stock follows a more complex process, as it is an OU process, as well as being a cev model.

$\beta$  is known as the elasticity of variance in the a constant elasticity of variance model. A  $\beta < 1$  means the asset price has a variance which increases as  $S$  decreases. This allows us to model assets where the volatility of the price increases as the price moves down. This is known as the leverage effect. Alternatively if  $\beta > 1$  we experience the inverse leverage effect, where volatility increases as price increases.

A  $\beta = 1$  reverts the theory to standard geometric motion, whilst a  $\beta = 0.5$  is included in the Cox-Ingersoll-Ross model. This allows us to include a mean -reversion property into the stock price [1].

The reason for the two different sets of values in 1 producing similar plots for the convertible bond value is because they represent the same underlying stock behaviour. A  $\beta = 1$  but  $\sigma > 1$  creates a Black-Scholes environment, however it is one where the stock is intrinsically already highly volatile. If you have a smaller  $\sigma$  but include a  $0 < \beta < 1$ , you create a constant elasticity of variance model, inducing asset price dependent volatility.

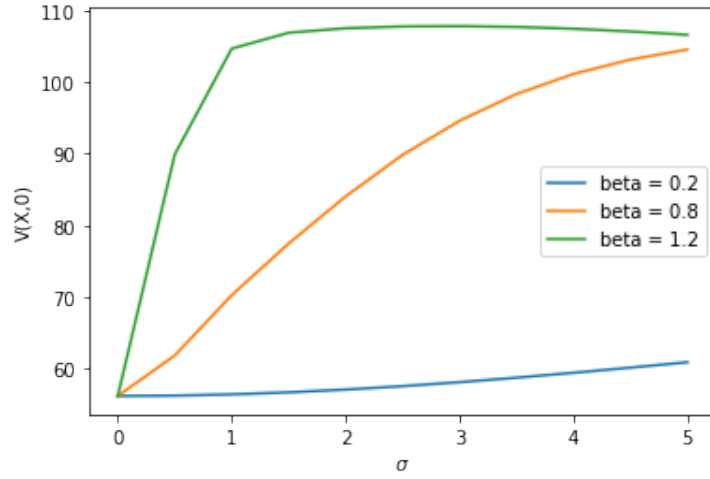


Figure 2: A graph showing relationship between the option value against  $\sigma$  for three different values of  $\beta$ .

Figure 2 shows the effect of increasing sigma for three different values of beta. This shows that increasing beta increases the rate at which the value increases as a function of  $\sigma$ , but it can clearly be seen that different values of  $\sigma$  and  $\beta$  could still lead to the same option price. For example for  $\beta = 0.8$  and  $\beta = 1.2$ , at  $\sigma = 5$ , similar values for the bond are seen.

This implies that having a high elasticity of variance and low implied volatility can be equal to having a lower elasticity of variance but higher implied volatility.

## 2.3 Accurate estimates for Option price

To explore the effect of getting an accurate asset price, we must first explore the effects of  $i_{max}$ ,  $j_{max}$  and  $S_{max}$  on the estimated value of the stock.

### 2.3.1 Effects of changing $S_{max}$

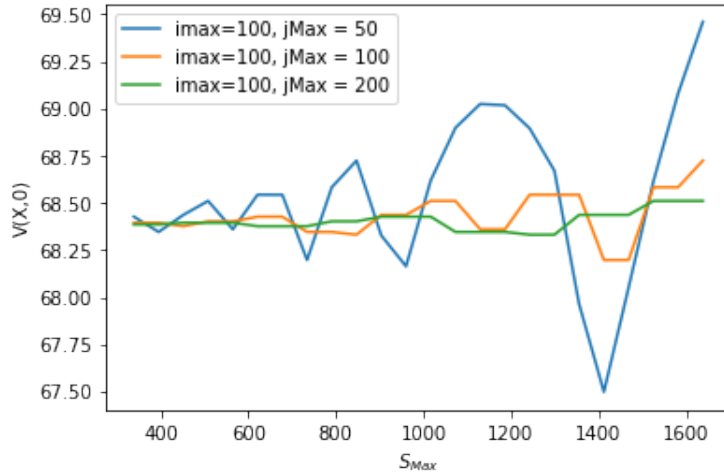


Figure 3: A graph showing relationship between  $S_{max}$  and the value of the option.

In figure 3, we show the effect of increasing  $S_{max}$  for different  $j_{max}$ . This is because  $j_{max}$  is a particularly important variable in this case. If  $S_{max}$  is increased significantly without increasing  $j_{max}$ , the calculated value oscillates more, because the size of the increments in  $S$  become significant, affecting the accuracy. However as long as  $j_{max}$  is large enough, this should not affect the answer significantly. Therefore ideally we want to increase  $S_{max}$  without increasing the size of  $\Delta S$ . This should lead to convergence, and is shown in figure 4. I also found that increasing  $S_{max}$  too much can cause instability. This graph below also indicates that increasing  $S_{max}$  beyond a certain point does not have a large effect on the accuracy, but can effect computation time.

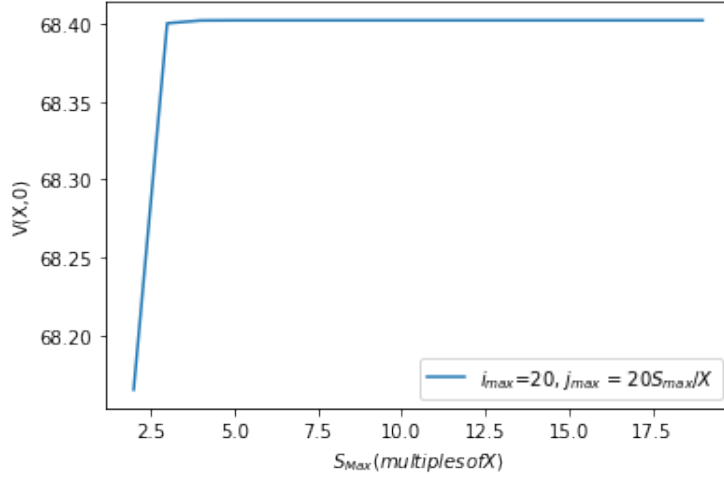


Figure 4: A graph showing relationship between  $S_{max}$  and the value of the option for fixed  $\Delta j$ .

### 2.3.2 Effects of changing $i_{max}$

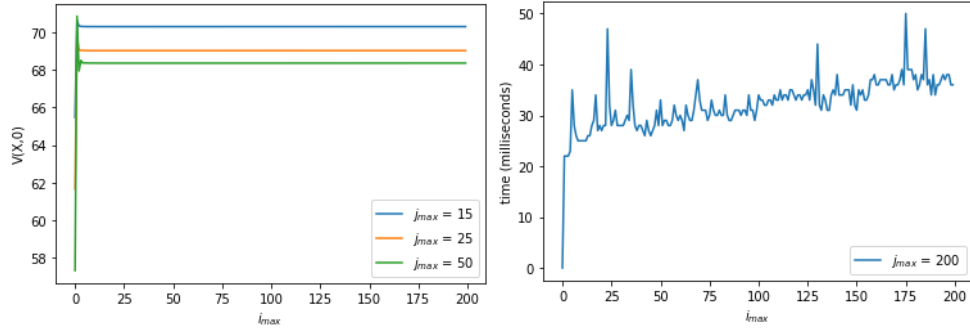


Figure 5: A graph showing relationship between  $i_{max}$  and the value of the option for different  $j_{max}$  (left) and its effect on computation time (right).

Figure 5 shows that the value of the option is not very sensitive to  $i_{max}$ . This is likely because it is not a time-dependent option, and therefore even small values of  $i_{max}$  accurately model the option. Increasing  $i_{max}$  also has a linear effect on computation time, however increasing it is not that useful as small values of  $i_{max}$  correctly model the option regardless.

What can already be seen is a dependence on  $j_{max}$ , which will be explored next.

### 2.3.3 Effects of changing $j_{max}$

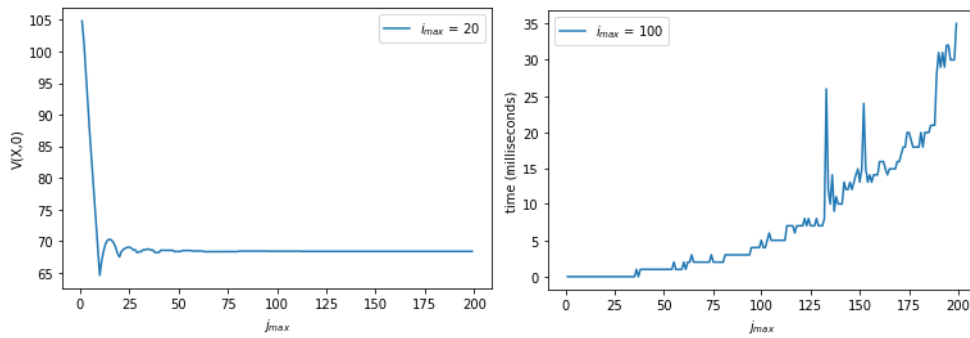


Figure 6: A graph showing relationship between  $j_{max}$  and the value of the option (left) and its effect on computation time (right).

As shown in figure 6, a  $j_{max}$  above 50 results in a stable solution. This is because a  $j_{max}$  this high accurately models the stock price continuously. What can also be seen is a roughly quadratic increase in the time take to compute the option value. If  $j_{max}$  is doubled, it takes four times as long.

### 2.3.4 Efficient calculation

It is clear that ,  $i_{max}$  has the smallest effect on the accuracy, and is more so for the stability of the Crank-Nicolson process.  $\Delta j$  has the greatest effect on correct option modelling. Therefore to look at efficiency we look at the most accurate answer we can get in a fixed time. The time chosen is approximately 500 milliseconds. We would want also want to find the smallest possible  $S_{max}$  which gives us a solution which converges for constant  $\Delta j$ .

In this time, the maximum parameters are  $i_{max} = 26$  ,  $j_{max} = 40$  ,  $S_{max} = 5X$  in 673 milliseconds with Lagrange interpolation. This leads to a value of

$$V(S = X, T = 0) = 68.3501 \quad (24)$$

where a Lagrange interpolation has also been used. However it is difficult to check the accuracy of the result , as an analytic solution is not possible. For this reason we instead compare to a case where we have increased the significant parameter ( $j_{max}$ ) significantly to see if massively increasing this parameter show a convergence to a different solution. This can be compared to a case where  $i_{max} = 200$  ,  $j_{max} = 400$  ,  $S_{max} = 5X$  which takes 146,401 milliseconds and returns a value of

$$V(S = X, T = 0) = 68.3513 \quad (25)$$

meaning the large increase in parameters and computation time leads only to a small change. This makes me believe the previous calculation is accurate to within roughly 1%.

## 3 American Option

### 3.1 Value as a function of stock price

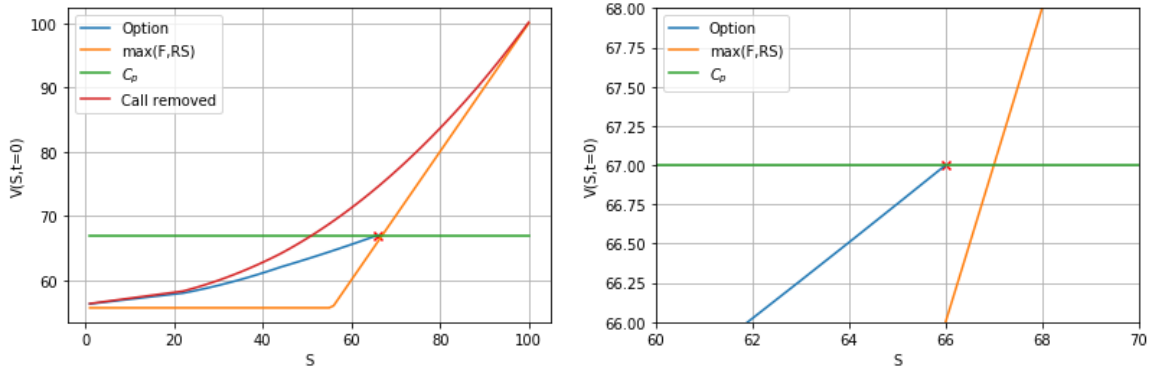


Figure 7: A graph showing the value of the American style bond contract as a function of asset price  $S$  using PSOR, at  $t = 0$  (left), and a magnified part of the graph, showing the point where the option becomes equal to  $C_p$  . The decision point has been marked on the graph.

Figure 7 shows the value of an American style bond contract as a function of  $S$ , with and without the embedded call option, using PSOR. Embedding the call reduces the value of the option at a given  $S$ , as there is a probability the issuer can call it back if the stock price increases too early, leading to unrealised gains when compared to the option without the embedded call. At low  $S$ , the value of the option with and without the call option converge, as it becomes less likely that the call option will be executed.

The curve meets the value  $C_p$  when  $S=66$ , rather than when  $S=67$  (when the parity meets  $C_p$ ). This is because we have not accounted for the discontinuity in time at this point. When we account for the discontinuity in time using the Penalty method, we get the graphs shown in figure 8.

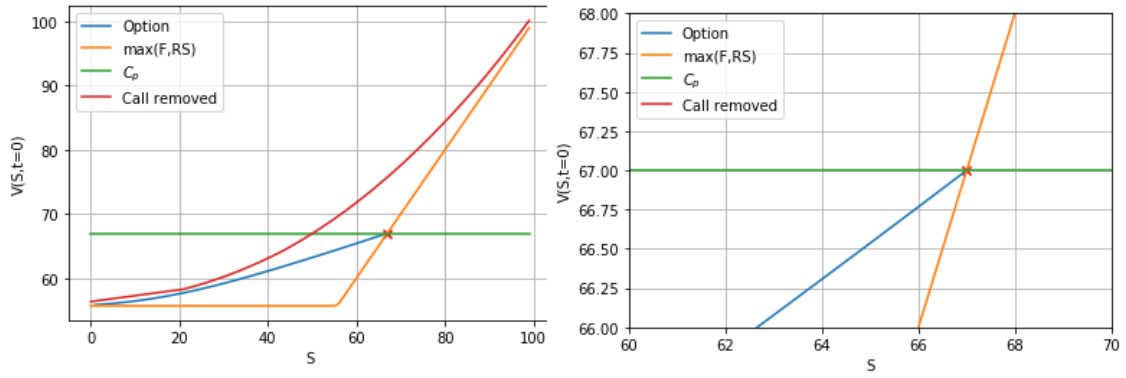


Figure 8: A graph showing the value of the American style bond contract as a function of asset price  $S$  using penalty method, at  $t = 0$  (left), and a magnified part of the graph, showing the point where the option becomes equal to  $C_p$ . The decision point has been marked on the graph.

The marked point is the point where beyond which the convertible bond would not exist. This is because if the stock price was higher than this value, the issuer would immediately be able to call it back. The value of the bond would be higher than  $C_p$ , and therefore the issuer would be able to make a risk-free profit (arbitrage).

### 3.2 Value for different interest rate environments

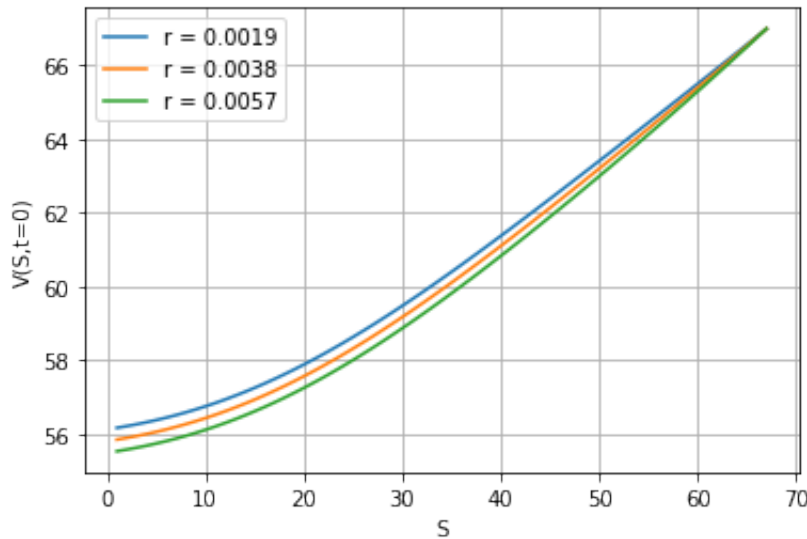


Figure 9: A graph showing the value of the American style bond contract as for three different interest rates.

In figure 9, we show the value as a function of  $S$  for three different interest rates, calculated using the penalty method. It shows a lower interest rates results in a consistently higher value for the option at  $t=0$  for all  $S$ . The size of the difference gets larger as  $S$  decreases. This is because at low  $S$ , the behaviour becomes more similar to a bond. This means it is unlikely we will choose to convert to shares, and therefore are receiving coupon payments as well as the principle. However these future cash flows are discounted, and a higher interest rate results in greater discounting. This can explain the reduced value for the convertible bond in a higher interest rate environment.

### 3.3 Getting an accurate value for American style option

We are looking to get the most accurate value of an American style option in one second of computation time at  $S_0 = 56.47$ . However we must first work around any discontinuities in the domain.

There is a discontinuity in the time domain at  $t = t_0 = 1.2448$ . Therefore our  $i_{max}$  must make the spacing's in time fall exactly on this boundary. Therefore the following modification was made for the spacing in time:

$$f = \frac{T - t_0}{T} \quad (26)$$

$$\Delta t = \frac{t_0}{i_{\max}(1-f)} \quad \text{for } 0 < t \leq t_0 \quad (27)$$

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

N	V(S=X,t=0)	time (milliseconds)	Difference Ratio
<b>100</b>	64.77814266	1	
<b>200</b>	64.7711116	6	
<b>400</b>	64.76345296	25	0.918
<b>800</b>	64.7596352	96	2.01
<b>1600</b>	64.75773137	446	2.01
<b>3200</b>	64.75678185	1728	2.01
<b>6400</b>	64.75630771	5739	2

Table 1: A Table showing the convergence of the value of the convertible bond for the penalty method.  $i_{\max} = j_{\max} = N$  and  $S_{\max} = NX/20$

I decided to use the penalty method as it is much more accurate and efficient than PSOR, meaning I will be able to get a more accurate answer in less time. The convergence of the penalty method is shown in table 1. Another thing worthy of noting is the difference ratio is 2 rather than 4. This is because of the increased complexity of convertible bonds, even compared to normal American options. They are known to be difficult to model and this is evident in the table.

Using this information, we attempted to calculate the most accurate value for the convertible bond in one second. For  $i_{\max} = 2000$ ,  $j_{\max} = 2500$  and  $S_{\max} = 250X$  we get a value of:

$$V(S = X, t = 0) = 64.92024109 \quad (29)$$

in 924 milliseconds.

## References

- [1] Vadim Linetsky and Rafael Mendoza. “The Constant Elasticity of Variance Model”. In: 2009.
- [2] Jan de. Spiegeleer, Wim Schoutens, and Cynthia Vanhulle. *The handbook of hybrid securities: convertible bonds, CoCo bonds, and bail-in*. Wiley, 2014.

## 4 Appendix

### 4.1 European Style Convertible Bond code

```

1 #pragma once
2 #include <iostream>
3 #include <fstream>
4 #include <cmath>
5 #include <vector>
6 #include <algorithm>
7 #include <chrono>
8 using namespace std::chrono;
9 using namespace std;
10
11 double lagrangeInterpolation(const vector<double>& y, const vector<double>& x, double x0,
12     unsigned int n)
13 {
14     if (x.size() < n) return lagrangeInterpolation(y, x, x0, x.size());
15     if (n == 0) throw;
16     int nHalf = n / 2;
17     int jStar;
18     double dx = x[1] - x[0];
19     if (n % 2 == 0)
20         jStar = int((x0 - x[0]) / dx) - (nHalf - 1);
21     else
22         jStar = int((x0 - x[0]) / dx + 0.5) - (nHalf);
23     jStar = std::max(0, jStar);

```



```

23 jStar = std::min(int(x.size() - n), jStar);
24 if (n == 1) return y[jStar];
25 double temp = 0.;
26 for (unsigned int i = jStar; i < jStar + n; i++) {
27     double int_temp;
28     int_temp = y[i];
29     for (unsigned int j = jStar; j < jStar + n; j++) {
30         if (j == i) { continue; }
31         int_temp *= (x0 - x[j]) / (x[i] - x[j]);
32     }
33     temp += int_temp;
34 }
35 // end of interpolate
36 return temp;
37 }
38 void sorSolve_EURO(const std::vector<double>& a, const std::vector<double>& b, const std::
    vector<double>& c, const std::vector<double>& rhs,
39     std::vector<double>& x, int iterMax, double tol, double omega, int& sor)
40 {
41     // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
42     int n = a.size() - 1;
43     // sor loop
44     for (sor = 0; sor < iterMax; sor++)
45     {
46         double error = 0.;
47         // implement sor in here
48         {
49             double y = (rhs[0] - c[0] * x[1]) / b[0];
50             x[0] = x[0] + omega * (y - x[0]);
51         }
52         for (int j = 1; j < n; j++)
53         {
54             double y = (rhs[j] - a[j] * x[j - 1] - c[j] * x[j + 1]) / b[j];
55             x[j] = x[j] + omega * (y - x[j]);
56         }
57         {
58             double y = (rhs[n] - a[n] * x[n - 1]) / b[n];
59             x[n] = x[n] + omega * (y - x[n]);
60         }
61         // calculate residual norm ||r|| as sum of absolute values
62         error += std::fabs(rhs[0] - b[0] * x[0] - c[0] * x[1]);
63         for (int j = 1; j < n; j++)
64             error += std::fabs(rhs[j] - a[j] * x[j - 1] - b[j] * x[j] - c[j] * x[j + 1]);
65         error += std::fabs(rhs[n] - a[n] * x[n - 1] - b[n] * x[n]);
66         // make an exit condition when solution found
67         if (error < tol)
68             break;
69     }
70 }
71
72 /* Solution code for the Crank Nicolson Finite Difference
73 search for COURSEWORK EDIT for parts that needed to be altered for the coursework
74 */
75
76 //This contains linear interpolation at the end
77 double crank_nicolson_E_LINEAR(double S0, double X, double F, double T, double r, double sigma
    ,
78     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)
79 {
80     // declare and initialise local variables (ds,dt)
81     double dS = S_max / jMax;
82     double dt = T / iMax;
83     // create storage for the stock price and option price (old and new)
84     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
85     // setup and initialise the stock price
86     for (int j = 0; j <= jMax; j++)
87     {
88         S[j] = j * dS;
89     }
90     // setup and initialise the final conditions on the option price
91     for (int j = 0; j <= jMax; j++)
92     {
93         vOld[j] = max(F, R * S[j]);
94         vNew[j] = max(F, R * S[j]);
95     }

```

```

96 // start looping through time levels
97 for (int i = iMax - 1; i >= 0; i--)
98 {
99     // declare vectors for matrix equations
100     vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
101     // set up matrix equations a[j]=
102     double theta = (1 + mu) * X * exp(mu * i * dt);
103     a[0] = 0;
104     b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
105     c[0] = (kappa * theta / dS);
106     d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
107
108     for (int j = 1; j <= jMax - 1; j++)
109     {
110         a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
111         * dS) / (4 * dS));
112         b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
113         / 2.);
114         c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
115         theta - j * dS)) / (4. * dS));
116         d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.)))
117         * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS))
118         * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
119     }
120     double A = R * exp((kappa + r) * (i * dt - T));
121     double B = X * R * (1 - exp((kappa + r) * (i * dt - T))) + (C / (alpha + r)) * (exp(-alpha
122     * i * dt) - exp(-alpha * T));
123     B = X * R * exp((kappa + r) * (dt * i - T)) + C * exp(-alpha * i * dt) / (alpha + r) + R *
124     exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt);
125     B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T)) - C *
126     exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
127     a[jMax] = 0;
128     b[jMax] = 1;
129     c[jMax] = 0;
130     d[jMax] = dS * jMax * A + B;
131     int sor;
132     // solve matrix equations with SOR
133     sorSolve_EURO(a, b, c, d, vNew, iterMax, tol, omega, sor);
134     if (sor == iterMax)
135         cout << "\n NOT CONVERGED \n";
136
137     // set old=new
138     vOld = vNew;
139 }
140 // finish looping through time levels
141
142 // output the estimated option price
143 double sum;
144 {
145     int jStar = S0 / dS;
146     sum = 0.;
147     if (jStar > 0 && jStar < jMax) {
148         sum += ((S0 - S[jStar]) * (S0 - S[jStar + 1]) / (2 * dS * dS)) * vNew[jStar - 1];
149         sum -= ((S0 - S[jStar - 1]) * (S0 - S[jStar + 1]) / (dS * dS)) * vNew[jStar];
150         sum += ((S0 - S[jStar - 1]) * (S0 - S[jStar]) / (2 * dS * dS)) * vNew[jStar + 1];
151     }
152     else {
153         sum += (S0 - S[jStar]) / dS * vNew[jStar + 1];
154         sum += (S[jStar + 1] - S0) / dS * vNew[jStar];
155     }
156 }
157 return sum;
158 }
159
160 /* Template code for the Crank Nicolson Finite Difference
161 */
162 //This contains lagrangian interpolation at the end
163 double crank_nicolson_E_LAG(double S0, double X, double F, double T, double r, double sigma,
164 double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
165 int S_max, double tol, double omega, int iterMax)
166 {
167     // declare and initialise local variables (ds,dt)
168     double dS = S_max / jMax;
169     double dt = T / iMax;
170     // create storage for the stock price and option price (old and new)

```

```

163 vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
164 // setup and initialise the stock price
165 for (int j = 0; j <= jMax; j++)
166 {
167     S[j] = j * dS;
168 }
169 // setup and initialise the final conditions on the option price
170 for (int j = 0; j <= jMax; j++)
171 {
172     vOld[j] = max(F, R * S[j]);
173     vNew[j] = max(F, R * S[j]);
174 }
175 // start looping through time levels
176 for (int i = iMax - 1; i >= 0; i--)
177 {
178     // declare vectors for matrix equations
179     vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
180     // set up matrix equations a[j]=
181     double theta = (1 + mu) * X * exp(mu * i * dt);
182     a[0] = 0;
183     b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
184     c[0] = (kappa * theta / dS);
185     d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
186
187     for (int j = 1; j <= jMax - 1; j++)
188     {
189         //
190         a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
191 * dS) / (4 * dS));
192         b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
193 / 2.);
194         c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
195 theta - j * dS)) / (4. * dS));
196         d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.)))
197 * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS))
198 * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
199         //
200     }
201     double A = R * exp((kappa + r) * (i * dt - T));
202     double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T))
203 - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
204     a[jMax] = 0;
205     b[jMax] = 1;
206     c[jMax] = 0;
207     d[jMax] = jMax * dS * A + B;
208     int sor;
209     // solve matrix equations with SOR
210     sorSolve_EURO(a, b, c, d, vNew, iterMax, tol, omega, sor);
211     //vNew = thomasSolve(a, b, c, d);
212
213     if (sor == iterMax)
214         return -1;
215
216     // set old=new
217     vOld = vNew;
218 }
219 // finish looping through time levels
220
221 // output the estimated option price
222 double optionValue = lagrangeInterpolation(vNew, S, S0, vNew.size());
223 return optionValue;
224 }
225
226 //This function creates a txt file (sigmabeta.txt) Which explores the effect of changing
227 sigma and beta on option value
228 void Getsigmabeta() {
229     //Creates three csv files with increasing sigma at a given beta for a fixed s0
230     // double S0 = X;
231     double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333333333,
232     mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 1., sigma = 3.73, tol = 1.e-7,
233     omega = 1., S_max = 10 * X;
234     //
235     int iterMax = 10000, iMax = 100, jMax = 200;
236     double S0 = X;
237     jMax = 40;
238     iMax = 26;

```

```

231 double sMax = 20 * X;
232 beta = 0.425;
233 sigma = 3.73;
234
235 //Explore effect of Smax
236 //Look at given imax and jmax, then increase Smax
237 std::ofstream sigmabeta("./sigmabeta.txt");
238 for (int i = 0; i < 11; i++) {
239     sigmabeta << i * 0.5 << " , " <<
240     crank_nicolson_E_LINEAR(X, X, F, T, r, sigma = i*0.5, R, kappa, mu, C, alpha, 0.2, iMax,
241     jMax, sMax, tol, omega, iterMax) << " , " <<
242     crank_nicolson_E_LINEAR(X, X, F, T, r, sigma = i * 0.5, R, kappa, mu, C, alpha, 0.8,
243     iMax, jMax, sMax, tol, omega, iterMax) << " , " <<
244     crank_nicolson_E_LINEAR(X, X, F, T, r, sigma = i * 0.5, R, kappa, mu, C, alpha, 1.2,
245     iMax, jMax, sMax, tol, omega, iterMax) <<
246     "\n";
247 }
248 cout << "DONE V FUNC S_MAX" << endl;
249 }
250 void GetSmax() {
251     //Explore effect of Smax
252     //Look at given imax and jmax, then increase Smax
253     std::ofstream V_function_sMax("./V_function_sMax.txt");
254     for (int i = 2; i < 20; i++) {
255         // declare and initialise Black Scholes parameters - Currently looking at a solution we
256         // can get a definite answer for
257         double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333333,
258         mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 1., sigma = 3.73, tol = 1.e-7,
259         omega = 1., S_max = 10 * X;
260         //
261         int iterMax = 10000, iMax = 100, jMax = 200;
262         double S0 = X;
263         beta = 0.425;
264         sigma = 3.73;
265         V_function_sMax << i << " , " <<
266         crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 20, 10*i, i
267         * X, tol, omega, iterMax) << " , " <<
268         crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 10*i, i
269         * X, tol, omega, iterMax) << " , " <<
270         crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 10*i, i
271         * X, tol, omega, iterMax) <<
272         "\n";
273     }
274     cout << "DONE V FUNC S_MAX" << endl;
275 }
276 //This attempts to help find the most efficient value for imax, jmax and Smax
277 void GetEfficientResult() {
278     // declare and initialise Black Scholes parameters - Currently looking at a solution we can
279     // get a definite answer for
280     double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333333,
281     mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 1., sigma = 3.73, tol = 1.e-7,
282     omega = 1., S_max = 10 * X;
283     //
284     int iterMax = 10000, iMax = 100, jMax = 200;
285     double S0 = X;
286     beta = 0.425;
287     sigma = 3.73;
288     iMax = 26;
289     jMax = 40;
290     double sMax = 10 * X;
291     cout <<
292     "imax = " << iMax << " " <<
293     "jmax = " << jMax << " " <<
294     "Smax = " << sMax << " ";
295     auto start = high_resolution_clock::now();
296     double V1 = crank_nicolson_E_LAG(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax,
297     jMax, sMax, tol, omega, iterMax);
298     auto stop = high_resolution_clock::now();
299     auto duration1 = duration_cast<microseconds>(stop - start);
300     cout << "OPTION VALUE = " << V1 << " ";
301     cout << "DURATION (microseconds): " << duration1.count() << endl;
302
303     cout <<
304     "imax = " << iMax << " " <<
305     "jmax = " << jMax << " " <<
306     "Smax = " << sMax << " ";

```

```

296 start = high_resolution_clock::now();
297 V1 = crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, 6*
    jMax, sMax, tol, omega, iterMax);
298 stop = high_resolution_clock::now();
299 duration1 = duration_cast<microseconds>(stop - start);
300 cout << "OPTION VALUE = " << V1 << " ";
301 cout << "DURATION (microseconds): " << duration1.count() << endl;
302 }
303
304 //This code creates csv files to allow us to explore the effect of imax, jmax and smax on time
305 void GetTimeData() {
306     // declare and initialise Black Scholes parameters - Currently looking at a solution we can
    get a definite answer for
307     double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.08333333,
308         mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 1., sigma = 3.73, tol = 1.e-7,
        omega = 1., S_max = 10 * X;
309     //
310     int iterMax = 10000, iMax = 100, jMax = 200;
311     double S0 = X;
312     beta = 0.425;
313     sigma = 3.73;
314     //Get data for increasing smax with time
315     //Look at given imax and jmax, then increase Smax
316     std::ofstream time_sMax("./time_sMax.txt");
317     for (int i = 4; i < 20; i++) {
318         time_sMax << i * X << " , ";
319         auto start = high_resolution_clock::now();
320         crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax,
            i * X, tol, omega, iterMax);
321         auto stop = high_resolution_clock::now();
322         auto duration = duration_cast<milliseconds>(stop - start);
323         time_sMax << duration.count() << "\n";
324     }
325
326     //Get data for increasing imax with time
327     //Look at given imax and jmax, then increase Smax
328     std::ofstream time_iMax("./time_iMax.txt");
329     for (int i = 0; i < 200; i++) {
330         time_iMax << i << " , ";
331         auto start = high_resolution_clock::now();
332         crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, jMax,
            S_max, tol, omega, iterMax);
333         auto stop = high_resolution_clock::now();
334         auto duration = duration_cast<milliseconds>(stop - start);
335         time_iMax << duration.count() << "\n";
336     }
337     cout << "DONE iMax as function of time" << endl;
338     //Get data for increasing jmax with timw
339     std::ofstream time_jMax("./time_jMax.txt");
340     for (int i = 1; i < 200; i++) {
341         time_jMax << i << " , ";
342         auto start = high_resolution_clock::now();
343         crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, i,
            S_max, tol, omega, iterMax);
344         auto stop = high_resolution_clock::now();
345         auto duration = duration_cast<milliseconds>(stop - start);
346         time_jMax << duration.count() << "\n";
347     }
348 }
349
350 }
351
352
353 //This populates the rest of the csv files on european bond data
354 void getEurobondData() {
355     // declare and initialise Black Scholes parameters - Currently looking at a solution we can
    get a definite answer for
356     double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.08333333,
357         mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 1., sigma = 3.73, tol = 1.e-7,
        omega = 1., S_max = 10 * X;
358     //
359     int iterMax = 10000, iMax = 100, jMax = 25;
360     beta = 0.425;
361     sigma = 3.73;
362     double S0 = X;
363     double V1 = 0, V2 = 0;

```

```

364 //Accuracy code
365
366
367 for (int i = 1; i < 400; i++) {
368
369     auto start = high_resolution_clock::now();
370     V1 = crank_nicolson_E_LAG(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 26, 40, 5 *
371     X, tol, omega, iterMax);
372     auto stop = high_resolution_clock::now();
373     auto duration = duration_cast<microseconds>(stop - start);
374     if (abs(V2 - V1) < 1e-5) {
375         cout << "S_max = " << i << endl;
376         cout << "OPTION VALUE = " << V1 << endl;
377         cout << "DURATION (microseconds):" << duration.count() << endl;
378         break;
379     }
380     else {
381         V2 = V1;
382     }
383 }
384 //
385
386
387 // declare and initialise grid paramaters
388 //int iMax = 100, jMax = 25;
389 // declare and initialise local variables (ds,dt)
390 //double S_max = 10 * X;
391 //int iterMax = 5000000;
392 //double tol = 1.e-7, omega = 1.;
393
394 //Checking value against theory
395 std::ofstream analytical("./analytical.txt");
396 for (int s = 1; s <= 300; s++) {
397     analytical << s << " , " << crank_nicolson_E_LINEAR(s, X, F, T, r, 3.73, R, 0, mu, C,
398     alpha, 1., iMax, jMax, S_max, tol, omega, iterMax) << "\n";
399 }
400
401 //Create graph of varying S and optionvalue
402 int length = 50;
403 double S_maxi = 4 * X;
404 std::ofstream V_function_S("./V_function_S.txt");
405 std::ofstream V_function_beta("./V_function_beta.txt");
406
407 for (int j = 1; j <= length - 1; j++) {
408     //Plottting V(S,t) as a function of S (t=0) for two cases
409     beta = 1.;
410     sigma = 0.369;
411     //Puts value of S, and value of stock for different parameters into csv file
412     V_function_S << j * S_maxi / length << " , " << crank_nicolson_E_LINEAR(j * S_maxi /
413     length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 1., iMax, jMax,
414     S_max, tol, omega, iterMax) << " , " << crank_nicolson_E_LINEAR(j * S_maxi / length, X,
415     F, T, r, 3.73, R, kappa, mu, C, alpha, 0.425, iMax, 35,
416     S_max, tol, omega, iterMax) << "\n";
417
418     //You may wish to explore different values of beta and sigma? Do research before
419     implementing this. But preliminary (BETA =1)
420     V_function_beta << j * S_maxi / length << " , " <<
421     crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha,
422     3., iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
423     crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha,
424     2, iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
425     crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha,
426     1, iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
427     crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha,
428     0.5, iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
429     //crank_nicolson2(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 0,
430     iMax, jMax, S_max, tol, omega, iterMax) <<
431     "\n";
432 }
433
434 cout << "DONE V FUNC S" << endl;
435
436 //Assume now,
437 // double S0 = X;

```

```

430 beta = 0.425;
431 sigma = 3.73;
432
433 //Explore effect of Smax
434 //Look at given imax and jmax, then increase Smax
435 std::ofstream V_function_sMax("./V_function_sMax.txt");
436 for (int i = 6; i < 20; i++) {
437     V_function_sMax << i << " , " <<
438     crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 50 * i,
439     i * X, tol, omega, iterMax) << " , " <<
440     crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 2 * i,
441     i * X, tol, omega, iterMax) << " , " <<
442     crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 2 * i,
443     i * X, tol, omega, iterMax) <<
444     "\n";
445 }
446 cout << "DONE V FUNC S_MAX" << endl;
447 //Explore effect of imax
448 //Look at given imax and jmax, then increase Smax
449 std::ofstream V_function_iMax("./V_function_iMax.txt");
450 for (int i = 0; i < 200; i++) {
451     V_function_iMax << i << " , " <<
452     crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, 15,
453     S_max, tol, omega, iterMax) << " , " <<
454     crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, 25,
455     S_max, tol, omega, iterMax) << " , " <<
456     crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, 50,
457     S_max, tol, omega, iterMax) << " , " <<
458     "\n";
459 }
460 cout << "DONE V FUNC iMax" << endl;
461 //Explore effect of jmax
462 //Look at given imax and jmax, then increase Smax
463 std::ofstream V_function_jMax("./V_function_jMax.txt");
464 for (int i = 1; i < 200; i++) {
465     V_function_jMax << i << " , " <<
466     crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 20, i,
467     S_max, tol, omega, iterMax) <<
468     "\n";
469 }
470 cout << "DONE V FUNC jMax" << endl;
471 }

```

## 4.2 American Style Convertible Bond code

```

1 #pragma once
2 #include <iostream>
3 #include <fstream>
4 #include <cmath>
5 #include <vector>
6 #include <algorithm>
7 #include <chrono>
8 #include <iomanip>
9 using namespace std::chrono;
10 using namespace std;
11
12 std::vector<double> thomasSolve(const std::vector<double>& a, const std::vector<double>& b_,
13     const std::vector<double>& c, std::vector<double>& d)
14 {
15     int n = a.size();
16     std::vector<double> b(n), temp(n);
17     // initial first value of b
18     b[0] = b_[0];
19     for (int j = 1; j < n; j++)
20     {
21         b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
22         d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
23     }
24     // calculate solution
25     temp[n - 1] = d[n - 1] / b[n - 1];
26     for (int j = n - 2; j >= 0; j--)
27         temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];
28     return temp;
29 }

```

```

30 void sorSolve_AM(const std::vector<double>& a, const std::vector<double>& b, const std::vector
    <double>& c, const std::vector<double>& rhs,
31     std::vector<double>& x, int iterMax, double tol, double omega, int& sor, double dS, double
    cp, double t0, int i, double dt)
32 {
33     // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
34     int n = a.size() - 1;
35     // sor loop
36     for (sor = 0; sor < iterMax; sor++)
37     {
38         double error = 0.;
39         // implement sor in here
40         {
41             double y = (rhs[0] - c[0] * x[1]) / b[0];
42             if (i * dt < t0) {
43                 x[0] = max(0., min(cp, x[0] + omega * (y - x[0])));
44             }
45             else {
46                 x[0] = max(x[0] + omega * (y - x[0]), 0.);
47             }
48             error += (y - x[0]) * (y - x[0]);
49         }
50         for (int j = 1; j < n; j++)
51         {
52             double y = (rhs[j] - a[j] * x[j - 1] - c[j] * x[j + 1]) / b[j];
53             if (i * dt < t0) {
54                 x[j] = max(min(x[j] + omega * (y - x[j]), cp), j * dS);
55             }
56             else {
57                 x[j] = max(x[j] + omega * (y - x[j]), j * dS);
58             }
59             error += (y - x[j]) * (y - x[j]);
60         }
61         {
62             double y = (rhs[n] - a[n] * x[n - 1]) / b[n];
63             if (i * dt < t0) {
64                 x[n] = max(min(x[n] + omega * (y - x[n]), cp), n * dS);
65             }
66             else {
67                 x[n] = max(x[n] + omega * (y - x[n]), n * dS);
68             }
69             error += (y - x[n]) * (y - x[n]);
70         }
71         // make an exit condition when solution found
72         if (error < tol)
73             break;
74     }
75     if (sor >= iterMax)
76     {
77         std::cout << " Error NOT converging within required iterations\n";
78     }
79 }
80
81 /* Solution code for the Crank Nicolson Finite Difference
82 search for COURSEWORK EDIT for parts that needed to be altered for the coursework
83 */
84 double crank_nicolson_AM_LINEAR(double S0, double X, double F, double T, double r, double
    sigma,
85     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, double cp, double t0)
86 {
87     // declare and initialise local variables (ds,dt)
88     cp = 67;
89     double dS = S_max / jMax;
90     double dt = T / iMax;
91     // create storage for the stock price and option price (old and new)
92     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
93     // setup and initialise the stock price
94     for (int j = 0; j <= jMax; j++)
95     {
96         S[j] = j * dS;
97     }
98     // setup and initialise the final conditions on the option price
99     for (int j = 0; j <= jMax; j++)
100     {
101         vOld[j] = max(F, R * S[j]);

```



```

102     vNew[j] = max(F, R * S[j]);
103 }
104 // start looping through time levels
105 for (int i = iMax - 1; i >= 0; i--)
106 {
107     // declare vectors for matrix equations
108     vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
109     // set up matrix equations a[j]=
110     double theta = (1 + mu) * X * exp(mu * i * dt);
111     a[0] = 0;
112     b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
113     c[0] = (kappa * theta / dS);
114     d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
115     for (int j = 1; j <= jMax - 1; j++)
116     {
117         //
118         a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
119 * dS) / (4 * dS));
120         b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
121 / 2.);
122         c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
123 theta - j * dS)) / (4. * dS));
124         d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.)))
125 * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS))
126 * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
127     }
128     double A = R * exp((kappa + r) * (i * dt - T));
129     double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T))
130 - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
131     a[jMax] = 0;
132     b[jMax] = 1;
133     c[jMax] = 0;
134     d[jMax] = jMax * dS * A + B;
135     // solve matrix equations with SOR
136     int sor;
137     for (sor = 0; sor < iterMax; sor++)
138     {
139         double error = 0.;
140         // implement sor in here
141         {
142             double y = (d[0] - c[0] * vNew[1]) / b[0];
143             y = vNew[0] + omega * (y - vNew[0]);
144             if (i * dt < t0)
145             {
146                 y = max(0., min(cp, y));
147             }
148             else
149             {
150                 y = std::max(y, R * S[0]);
151             }
152             error += (y - vNew[0]) * (y - vNew[0]);
153             vNew[0] = y;
154         }
155         for (int j = 1; j < jMax; j++)
156         {
157             double y = (d[j] - a[j] * vNew[j - 1] - c[j] * vNew[j + 1]) / b[j];
158             y = vNew[j] + omega * (y - vNew[j]);
159             if (i * dt < t0)
160             {
161                 y = max(min(y, cp), j * dS);
162             }
163             else
164             {
165                 y = std::max(y, R * j * dS);
166             }
167             error += (y - vNew[j]) * (y - vNew[j]);
168             vNew[j] = y;
169         }
170         {
171             double y = (d[jMax] - a[jMax] * vNew[jMax - 1]) / b[jMax];
172             y = vNew[jMax] + omega * (y - vNew[jMax]);
173             if (i * dt < t0)
174             {
175                 y = max(min(y, cp), jMax * dS);
176             }
177             else

```

```

172     {
173         y = std::max(y, R * jMax * dS);
174     }
175     error += (y - vNew[jMax]) * (y - vNew[jMax]);
176     vNew[jMax] = y;
177 }
178 // make an exit condition when solution found
179 if (error < tol)
180     break;
181 }
182 if (sor >= iterMax)
183 {
184     std::cout << " Error NOT converging within required iterations\n";
185     std::cout.flush();
186     throw;
187 }
188
189 if (sor == iterMax)
190     return -1;
191
192 // set old=new
193 vOld = vNew;
194 }
195 // finish looping through time levels
196
197 // output the estimated option price
198 double sum;
199 {
200     int jStar = S0 / dS;
201     sum = 0.;
202     if (jStar > 0 && jStar < jMax) {
203         sum += ((S0 - S[jStar]) * (S0 - S[jStar + 1]) / (2 * dS * dS)) * vNew[jStar - 1];
204         sum -= ((S0 - S[jStar - 1]) * (S0 - S[jStar + 1]) / (dS * dS)) * vNew[jStar];
205         sum += ((S0 - S[jStar - 1]) * (S0 - S[jStar]) / (2 * dS * dS)) * vNew[jStar + 1];
206     }
207     else {
208         sum += (S0 - S[jStar]) / dS * vNew[jStar + 1];
209         sum += (S[jStar + 1] - S0) / dS * vNew[jStar];
210     }
211 }
212 return sum;
213 }
214
215 /* This code seems to run much faster when in a separate solution without header files, maybe
216    just copy and paste this
217 */
218 double crank_nicolson_AM_FAST(double S0, double X, double F, double T, double r, double sigma,
219     double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
220     int S_max, double tol, double omega, int iterMax, int& sorCount, double t0)
221 {
222     // declare and initialise local variables (ds,dt)
223     double cp = 67.;
224
225     double dS = S_max / jMax;
226     double dt = T / iMax;
227     // create storage for the stock price and option price (old and new)
228     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
229     // setup and initialise the stock price
230     for (int j = 0; j <= jMax; j++)
231     {
232         S[j] = j * dS;
233     }
234     // setup and initialise the final conditions on the option price
235     for (int j = 0; j <= jMax; j++)
236     {
237         vOld[j] = max(F, R * S[j]);
238         vNew[j] = max(F, R * S[j]);
239     }
240     // start looping through time levels
241     for (int i = iMax - 1; i >= 0; i--)
242     {
243         //if (i * dt < t0) { dt = t0 / iMax; }
244         //if (i * dt >= t0) { dt = (T - t0) / iMax; }
245         // declare vectors for matrix equations
246         vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
247         // set up matrix equations a[j]=

```

```

246 double theta = (1 + mu) * X * exp(mu * i * dt);
247 a[0] = 0;
248 b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
249 c[0] = (kappa * theta / dS);
250 d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * ((-1 / dt) + (r / 2)));
251 for (int j = 1; j <= jMax - 1; j++)
252 {
253     //
254     a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
255 * dS) / (4 * dS));
256     b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
257 / 2.);
258     c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
259 theta - j * dS)) / (4. * dS));
260     d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.)))
261 * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS))
262 * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
263 }
264 double A = R * exp((kappa + r) * (i * dt - T));
265 double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T))
266 - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
267 a[jMax] = 0;
268 b[jMax] = 1;
269 c[jMax] = 0;
270 d[jMax] = jMax * dS * A + B;
271 // solve matrix equations with SOR
272 int sor;
273 for (sor = 0; sor < iterMax; sor++)
274 {
275     double error = 0.;
276     // implement sor in here
277     {
278         double y = (d[0] - c[0] * vNew[1]) / b[0];
279         y = vNew[0] + omega * (y - vNew[0]);
280         if (i * dt < t0)
281         {
282             y = max(0., min(cp, y));
283         }
284         else
285         {
286             y = std::max(y, R * S[0]);
287         }
288         error += (y - vNew[0]) * (y - vNew[0]);
289         vNew[0] = y;
290     }
291     for (int j = 1; j < jMax; j++)
292     {
293         double y = (d[j] - a[j] * vNew[j - 1] - c[j] * vNew[j + 1]) / b[j];
294         y = vNew[j] + omega * (y - vNew[j]);
295         if (i * dt < t0)
296         {
297             y = max(min(y, cp), j * dS);
298         }
299         else
300         {
301             y = std::max(y, R * j * dS);
302         }
303         error += (y - vNew[j]) * (y - vNew[j]);
304         vNew[j] = y;
305     }
306     {
307         double y = (d[jMax] - a[jMax] * vNew[jMax - 1]) / b[jMax];
308         y = vNew[jMax] + omega * (y - vNew[jMax]);
309         if (i * dt < t0)
310         {
311             y = max(min(y, cp), jMax * dS);
312         }
313         else
314         {
315             y = std::max(y, R * jMax * dS);
316         }
317         error += (y - vNew[jMax]) * (y - vNew[jMax]);
318         vNew[jMax] = y;
319     }
320     // make an exit condition when solution found
321     if (error < tol)

```

```

316         break;
317     }
318     if (sor >= iterMax)
319     {
320         std::cout << " Error NOT converging within required iterations\n";
321         std::cout.flush();
322         throw;
323     }
324
325     if (sorCount == iterMax)
326         return -1;
327
328     // set old=new
329     vOld = vNew;
330 }
331 // finish looping through time levels
332
333 // output the estimated option price
334 double optionValue;
335
336 int jStar = S0 / dS;
337 double sum = 0.;
338 sum += (S0 - S[jStar]) / (dS)* vNew[jStar + 1];
339 sum += (S[jStar + 1] - S0) / (dS)* vNew[jStar];
340 optionValue = sum;
341 //optionValue = lagrangeInterpolation(vNew, S, S0, vNew.size());
342
343 return optionValue;
344 }
345
346 /* Template code for the Crank Nicolson Finite Difference
347 */
348 double crank_nicolson_penalty(double S0, double X, double F, double T, double r, double sigma,
349 double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
350 int S_max, double tol, double omega, int iterMax, int& sorCount, double t0)
351 {
352     // declare and initialise local variables (ds,dt)
353     double cp = 67.;
354     //dS calculated as before
355     double dS = S_max / jMax;
356     //What is f used for?
357     double f = (T - t0) / T;
358     //STILL T/iMax
359     double dt = (T - t0) / (iMax * f);
360     // create storage for the stock price and option price (old and new)
361     vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
362     // setup and initialise the stock price
363     for (int j = 0; j <= jMax; j++)
364     {
365         S[j] = j * dS;
366     }
367     // setup and initialise the final conditions on the option price
368     for (int j = 0; j <= jMax; j++)
369     {
370         vOld[j] = max(F, R * S[j]);
371         vNew[j] = max(F, R * S[j]);
372     }
373     // start looping through time levels
374     for (int i = iMax; i >= 0; i--)
375     {
376         //If you are before t-, change increments to
377         if (i * dt < t0)
378         {
379             dt = t0 / (iMax * (1 - f));
380         }
381
382         // declare vectors for matrix equations
383         vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
384         // set up matrix equations a[j]=
385         double theta = (1 + mu) * X * exp(mu * i * dt);
386         a[0] = 0;
387         b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
388         c[0] = (kappa * theta / dS);
389         d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
390         for (int j = 1; j <= jMax - 1; j++)
391         {

```

```

391 //
392 a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
* dS) / (4 * dS));
393 b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
/ 2.);
394 c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
theta - j * dS)) / (4. * dS));
395 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));
396 }
397 double A = R * exp((kappa + r) * (i * dt - T));
398 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);
399 a[jMax] = 0;
400 b[jMax] = 1;
401 c[jMax] = 0;
402 d[jMax] = jMax * dS * A + B;
403 double penalty = 1.e8;
404 int q;
405 for (q = 0; q < 100000; q++)
406 {
407     vector<double> bHat(b), dHat(d);
408     for (int j = 1; j < jMax; j++)
409     {
410         if (i * dt < t0)
411         {
412             if (vNew[j] > max(R * S[j], cp))
413             {
414                 bHat[j] = b[j] - penalty;
415                 dHat[j] = d[j] - penalty * max(R * S[j], cp);
416             }
417         }
418         else
419         {
420             // turn on penalty if V < RS
421             if (vNew[j] < R * S[j])
422             {
423                 bHat[j] = b[j] - penalty;
424                 dHat[j] = d[j] - penalty * R * S[j];
425             }
426         }
427     }
428 }
429 // solve matrix equations with SOR
430 vector<double> y = thomasSolve(a, bHat, c, dHat);
431 // calculate difference from last time
432 double error = 0.;
433 for (int j = 0; j <= jMax; j++)
434     error += fabs(vNew[j] - y[j]);
435 vNew = y;
436 if (error < 1.e-8)
437     break;
438 }
439 if (q == 100000)
440 {
441     std::cout << " Error NOT converging within required iterations\n";
442     std::cout.flush();
443     throw;
444 }
445 // set old=new
446 vOld = vNew;
447 }
448 // finish looping through time levels
449 // output the estimated option price
450 //Why 4?
451 return lagrangeInterpolation(vNew, S, S0, 4);
452 }
453
454 //Allows you to extract information in csv file for efficiency of penalty method
455 void GetPenaltyEfficiency()
456 {
457     // declare and initialise Black Scholes parameters - Currently looking at a solution we can

```

```

    get a definite answer for
461 double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333,
462 mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 0.425, sigma = 3.73, tol = 1.e-7,
    omega = 1., S_max = 10 * X;
463 double t0 = 1.2448;
464 //
465
466 int iterMax = 100000;
467 //Create graph of varying S0 and beta and bond
468 int length = 300;
469 int sorCount;
470 double S0 = X;
471 std::ofstream outFile("ameribond_eff.txt");
472 double oldResult = 0, oldDiff = 0;
473 double S = X;
474 int iMax = 100;
475 int jMax = 100;
476 S_max = 200 * X;
477 //cout << crank_nicolson1(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax,
    S_max, tol, omega, iterMax, sorCount, t0) << endl;
478
479 for (int n = 100; n <= 10000; n *= 2)
480 {
481     //Set aparameters for iteration
482     iMax = n;
483     jMax = n;
484     S_max = n / 20 * X;
485
486     auto t1 = std::chrono::high_resolution_clock::now();
487     double result = crank_nicolson_penalty(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta,
        iMax, jMax, S_max, tol, omega, iterMax, sorCount, t0);
488     double diff = result - oldResult;
489     auto t2 = std::chrono::high_resolution_clock::now();
490     auto time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
491
492     outFile << n << "," << setprecision(10) << result << "," << time_taken << "," <<
        setprecision(3) << oldDiff / diff << "\n";
493     cout << "RESULT : " << result << "    TIME: " << time_taken << "," << setprecision(3) <<
        oldDiff / diff << "\n";
494
495     oldDiff = diff;
496     oldResult = result;
497 }
498
499 }
500
501 void getAmeribondEfficiency() {
502     // declare and initialise Black Scholes parameters - Currently looking at a solution we can
    get a definite answer for
503 double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333,
504 mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 0.425, sigma = 3.73, tol = 1.e-7,
    omega = 1., S_max = 20 * X;
505 //
506 int iterMax = 100000, iMax = 40, jMax = 25;
507 beta = 0.425;
508 sigma = 3.73;
509 double S0 = X;
510 double t0 = 1.2448, cp = 67;
511
512 std::ofstream outFile5("american_varying_smax.txt");
513 tol = 1.e-7;
514 for (int i = 1; i <= 1; i++)
515 {
516     double jMax = 300;
517     double S = X;
518     int sorCount;
519     auto t1 = std::chrono::high_resolution_clock::now();
520     double result = crank_nicolson_AM_FAST(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta,
        iMax = 1000, jMax = 300 * i, S_max = S * cp, tol = 1e-8, omega, iterMax, sorCount, t0 =
        0);
521     auto t2 = std::chrono::high_resolution_clock::now();
522     auto time_taken =
523         std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
524         .count();
525     cout << result << "," << time_taken << "\n";
526     t1 = std::chrono::high_resolution_clock::now();

```

```

527     result = crank_nicolson_AM_LINEAR(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax
528     = 1000, jMax = 300 * i, S_max = S * cp, tol = 1e-8, omega, iterMax, sorCount, t0 = 0);
529     t2 = std::chrono::high_resolution_clock::now();
530     time_taken =
531     std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
532     .count();
533     cout << result << ", " << time_taken << "\n";
534 }
535 outFile5.close();
536 }
537
538
539 //Get data for ameribond. Gets information for first part of American Bond graphs
540 void getAmeribondData() {
541     // declare and initialise Black Scholes parameters - Currently looking at a solution we can
542     // get a definite answer for
543     double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.08333333333,
544     mu = 0.0073, X = 56.47, C = 0.106, alpha = 0.01, beta = 0.425, sigma = 3.73, tol = 1.e-7,
545     omega = 1., S_max = 15 * X;
546     //
547     int iterMax = 10000, iMax = 700, jMax = 700;
548     beta = 0.425;
549     sigma = 3.73;
550     double S0 = X;
551     double t0 = 1.2448, cp = 67;
552     int sor;
553     //Checking value against theory
554     std::ofstream analytical("./Ameribond1.txt");
555     for (int s = 1; s <= 100; s++) {
556         analytical << s << " , " << crank_nicolson_penalty(s, X, F, T, r, sigma, R, kappa, mu, C,
557         alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0) << "\n";
558     }
559     cout << "AMERICAN BOND PART 1 DONE" << endl;
560
561     //Checking how the value changes with different values of r
562     std::ofstream r_file("./Ameribond_r.txt");
563     for (int s = 1; s <= 67; s++) {
564         r_file << s << " , " <<
565         crank_nicolson_penalty(s, X, F, T, 0.0019, sigma, R, kappa, mu, C, alpha, beta, iMax,
566         jMax, S_max, tol, omega, iterMax, sor, t0) << " , " <<
567         crank_nicolson_penalty(s, X, F, T, 0.0038, sigma, R, kappa, mu, C, alpha, beta, iMax,
568         jMax, S_max, tol, omega, iterMax, sor, t0) << " , " <<
569         crank_nicolson_penalty(s, X, F, T, 0.0057, sigma, R, kappa, mu, C, alpha, beta, iMax,
570         jMax, S_max, tol, omega, iterMax, sor, t0) <<
571         "\n";
572     }
573     cout << "AMERICAN BOND PART 2 DONE" << endl;
574 }

```