Student Id: 9914471

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 $t = T$.

W calculate the value of this option using the finite-difference method with a Crank-Nicolson Scheme. We then explore the effects of varying β and σ. 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:

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

where $\theta(t)$ is

$$\theta(t) = (1+\mu)Xe^{\mu t}. \tag{2}$$

2.1 Numerical Scheme

2.1.1 Low S numerical scheme

The boundary condition at $S = 0$ is:

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

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

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

Therefore, the numerical scheme is:

$$a_0 = 0 \tag{5}$$

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

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

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

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

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

We can again extract the numerical scheme from this

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

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

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

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

2.1.3 Large S limit

At very large S, the PDE simplifies slightly to

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

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

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

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

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

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

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

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

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

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

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

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 β and σ.

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

$$V(S,T) = max(F,RS). \tag{22}$$

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

$$V(S,T) = N + Rmax(0, S - C_p). \tag{23}$$

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].

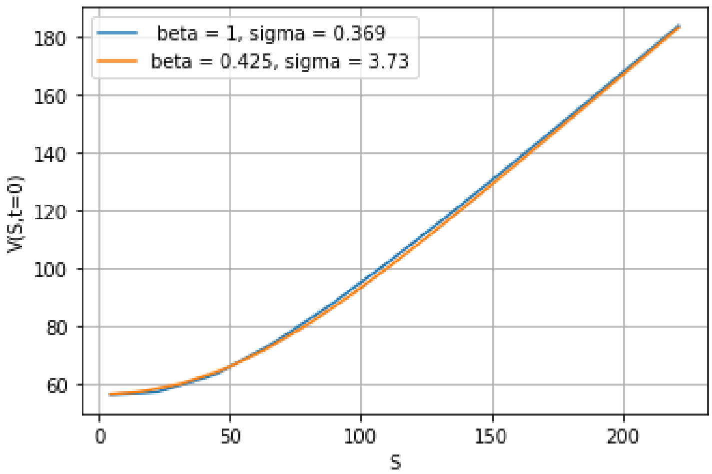

Figure 1: A graph showing relationship between the convertible bond value against stock price for two different σ and β .

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.

σ 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 converible bond, the underlying stock follows a more complex process, as it is an OU process, as well as being a cev model.

β 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 σ but include a $0 < \beta < 1$, you create a constant elasticity of variance model, inducing asset price dependent volatility.

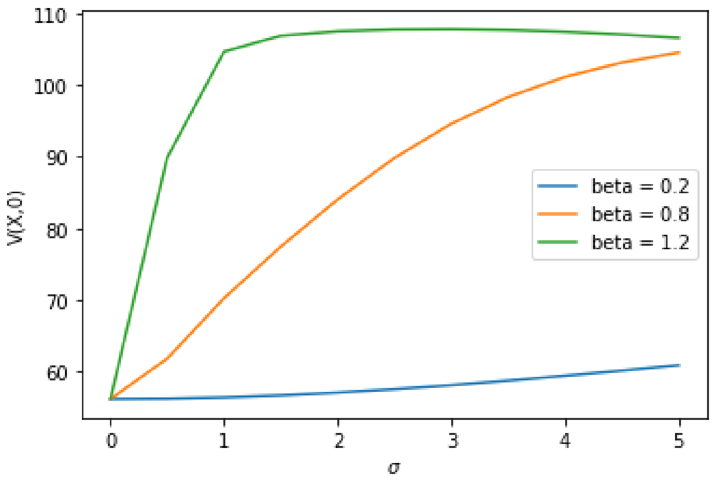

Figure 2: A graph showing relationship between the option value against σ for three different values of β .

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 σ, but it can clearly be seen that different values of σ and β 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}

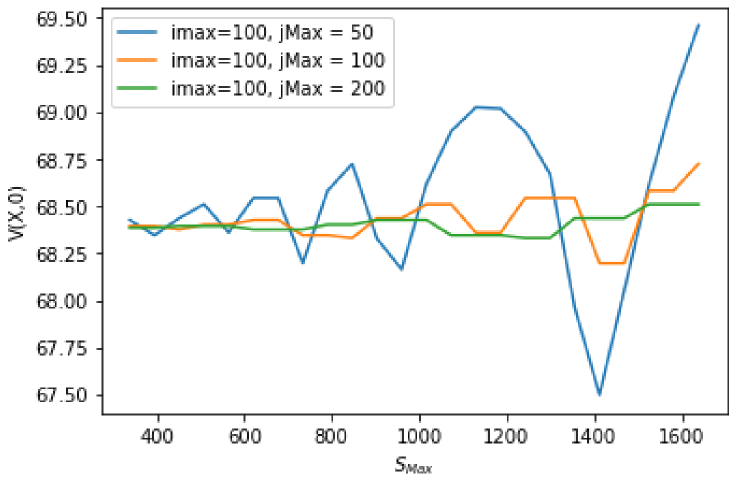

Figure 3: A graph showing relationship between S_{max} and the value of the option.

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 ΔS. 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.

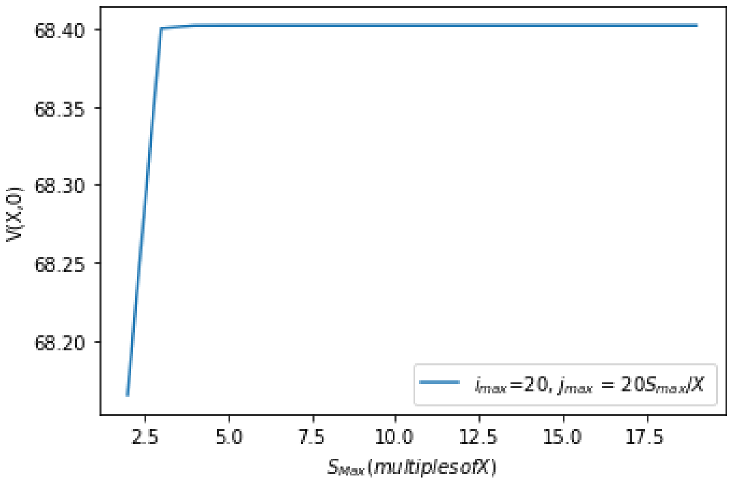

Figure 4: A graph showing relationship between S_{max} and the value of the option for fixed Δj.

2.3.2 Effects of changing i_{max}

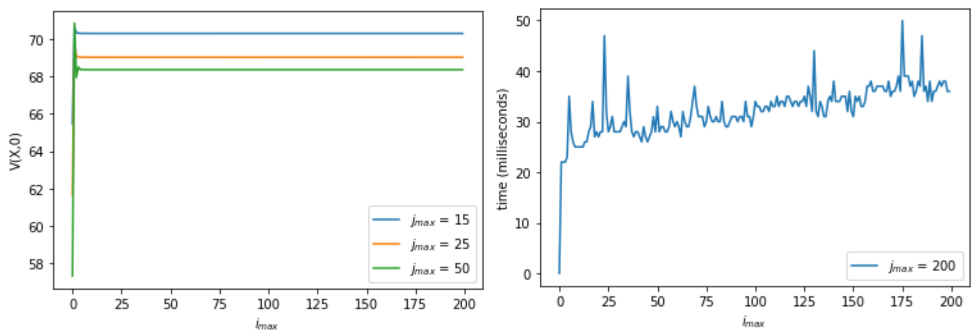

Figure 5: A graph showing relationship between i_{max} and the value of the option for different j_{max} (left) and its effect on computation time (right).

Figure 5 shows that the value of the option in 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}

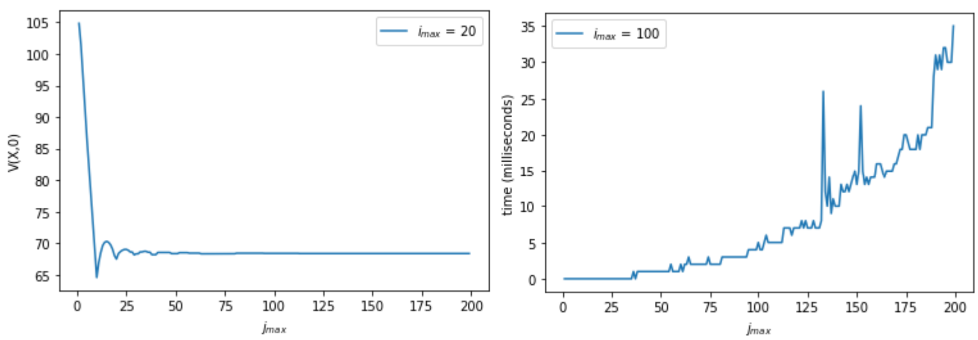

Figure 6: A graph showing relationship between j_{max} and the value of the option (left) and its effect on computation time (right).

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. Δj 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 Δj.

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

$$V(S = X, T = 0) = 68.3501 \tag{24}$$

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

$$V(S = X, T = 0) = 68.3513 \tag{25}$$

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

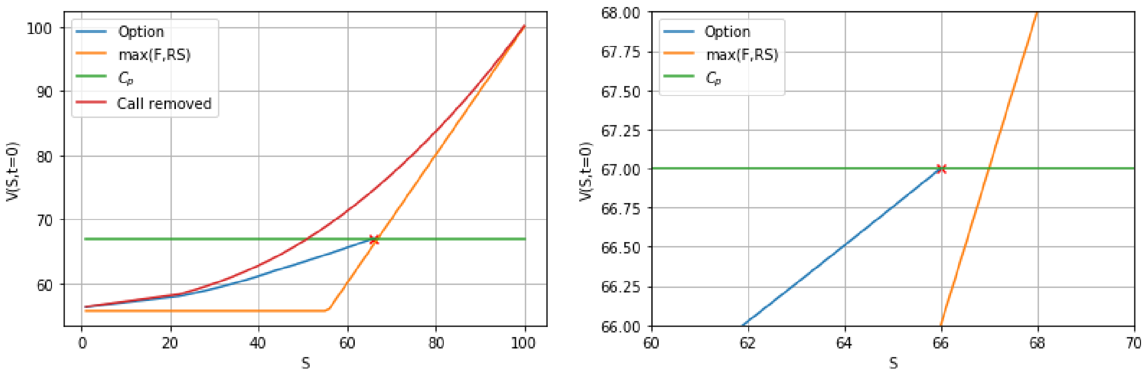

Figure 7: A graph showing the value of the American style bond contract as a function of asset price S using PSOR, at t = 0 (left), and a magnified part of the graph, showing the point where the option becomes equal to C_p . The decision point has been marked on the graph.

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.

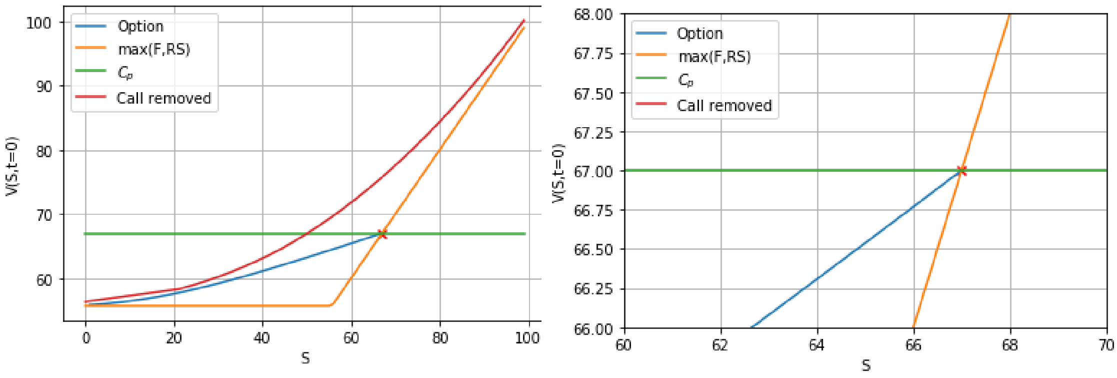

Figure 8: A graph showing the value of the American style bond contract as a function of asset price S using penalty method, at t = 0 (left), and a magnified part of the graph, showing the point where the option becomes equal to C_p . The decision point has been marked on the graph.

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

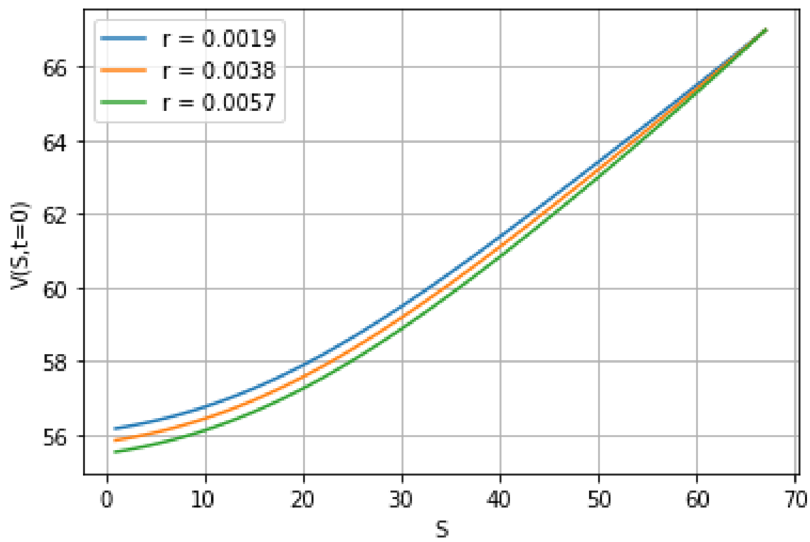

Figure 9: A graph showing the value of the American style bond contract as for three different interest rates.

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:

$$f = \frac{T - t_0}{T} \tag{26}$$

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

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

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

$$V(S = X, t = 0) = 64.92024109 \tag{29}$$

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

```
#pragma once
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
#include <chrono>
using namespace std::chrono;
using namespace std;

double lagrangeInterpolation(const vector<double>& y, const vector<double>& x, double x0,
    unsigned int n)
{
  if (x.size() < n)return lagrangeInterpolation(y, x, x0, x.size());
  if (n == 0)throw;
  int nHalf = n / 2;
  int jStar;
  double dx = x[1] - x[0];
  if (n % 2 == 0)
    jStar = int((x0 - x[0]) / dx) - (nHalf - 1);
  else
    jStar = int((x0 - x[0]) / dx + 0.5) - (nHalf);
  jStar = std::max(0, jStar);
```

```
  jStar = std::min(int(x.size() - n), jStar);
  if (n == 1)return y[jStar];
  double temp = 0.;
  for (unsigned int i = jStar; i < jStar + n; i++) {
    double  int_temp;
    int_temp = y[i];
    for (unsigned int j = jStar; j < jStar + n; j++) {
      if (j == i) { continue; }
      int_temp *= (x0 - x[j]) / (x[i] - x[j]);
    }
    temp += int_temp;
  }
  // end of interpolate
  return temp;
}
void sorSolve_EURO(const std::vector<double>& a, const std::vector<double>& b, const std::
    vector<double>& c, const std::vector<double>& rhs,
  std::vector<double>& x, int iterMax, double tol, double omega, int& sor)
{
  // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
  int n = a.size() - 1;
  // sor loop
  for (sor = 0; sor < iterMax; sor++)
  {
    double error = 0.;
    // implement sor in here
    {
      double y = (rhs[0] - c[0] * x[1]) / b[0];
      x[0] = x[0] + omega * (y - x[0]);
    }
    for (int j = 1; j < n; j++)
    {
      double y = (rhs[j] - a[j] * x[j - 1] - c[j] * x[j + 1]) / b[j];
      x[j] = x[j] + omega * (y - x[j]);
    }
    {
      double y = (rhs[n] - a[n] * x[n - 1]) / b[n];
      x[n] = x[n] + omega * (y - x[n]);
    }
    // calculate residual norm ||r|| as sum of absolute values
    error += std::fabs(rhs[0] - b[0] * x[0] - c[0] * x[1]);
    for (int j = 1; j < n; j++)
      error += std::fabs(rhs[j] - a[j] * x[j - 1] - b[j] * x[j] - c[j] * x[j + 1]);
    error += std::fabs(rhs[n] - a[n] * x[n - 1] - b[n] * x[n]);
    // make an exit condition when solution found
    if (error < tol)
      break;
  }
}

/* Solution code for the Crank Nicolson Finite Difference
search for COURSEWORK EDIT for parts that needed to be altered for the coursework
 */

//This contains linear interpolation at the end
double crank_nicolson_E_LINEAR(double S0, double X, double F, double T, double r, double sigma
    ,
  double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
    int S_max, double tol, double omega, int iterMax)
{
  // declare and initialise local variables (ds,dt)
  double dS = S_max / jMax;
  double dt = T / iMax;
  // create storage for the stock price and option price (old and new)
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
    vNew[j] = max(F, R * S[j]);
  }
```

```
  // start looping through time levels
  for (int i = iMax - 1; i >= 0; i--)
  {
    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));

    for (int j = 1; j <= jMax - 1; j++)
    {
      a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
    * dS) / (4 * dS));
      b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
    / 2.);
      c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
    theta - j * dS)) / (4. * dS));
      d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.)))
     * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS))
     * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
    }
    double A = R * exp((kappa + r) * (i * dt - T));
    double B = X * R * (1 - exp((kappa + r) * (i * dt - T))) + (C / (alpha + r)) * (exp(-alpha
     * i * dt) - exp(-alpha * T));
    B = X * R * exp((kappa + r) * (dt * i - T)) + C * exp(-alpha * i * dt) / (alpha + r) + R *
     exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt);
    B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T)) - C *
    exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
    a[jMax] = 0;
    b[jMax] = 1;
    c[jMax] = 0;
    d[jMax] = dS * jMax * A + B;
    int sor;
    // solve matrix equations with SOR
    sorSolve_EURO(a, b, c, d, vNew, iterMax, tol, omega, sor);
    if (sor == iterMax)
      cout << "\n NOT CONVERGED \n";

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

  // output the estimated option price
  double sum;
  {
    int jStar = S0 / dS;
    sum = 0.;
    if (jStar > 0 && jStar < jMax) {
      sum += ((S0 - S[jStar]) * (S0 - S[jStar + 1]) / (2 * dS * dS)) * vNew[jStar - 1];
      sum -= ((S0 - S[jStar - 1]) * (S0 - S[jStar + 1]) / (dS * dS)) * vNew[jStar];
      sum += ((S0 - S[jStar - 1]) * (S0 - S[jStar]) / (2 * dS * dS)) * vNew[jStar + 1];
    }
    else {
      sum += (S0 - S[jStar]) / dS * vNew[jStar + 1];
      sum += (S[jStar + 1] - S0) / dS * vNew[jStar];
    }
  }
  return sum;
}


/* Template code for the Crank Nicolson Finite Difference
 */
//This contains lagrangian interpolation at the end
double crank_nicolson_E_LAG(double S0, double X, double F, double T, double r, double sigma,
  double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
    int S_max, double tol, double omega, int iterMax)
{
  // declare and initialise local variables (ds,dt)
  double dS = S_max / jMax;
  double dt = T / iMax;
  // create storage for the stock price and option price (old and new)
```

```
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
    vNew[j] = max(F, R * S[j]);
  }
  // start looping through time levels
  for (int i = iMax - 1; i >= 0; i--)
  {
    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));

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

    if (sor == iterMax)
      return -1;

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

  // output the estimated option price
  double optionValue = lagrangeInterpolation(vNew, S, S0, vNew.size());
  return optionValue;
}

//This function creates a txt file (sigmabeta.txt) Which explores the effect of changinging
    sigma and beta on option value
void Getsigmabeta() {
  //Creates three csv files with increasing sigma at a given beta for a fixed s0
  //  double S0 = X;
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.083333333333,
    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;
  //
  int iterMax = 10000, iMax = 100, jMax = 200;
  double S0 = X;
  jMax = 40;
  iMax = 26;
```

```
  double sMax = 20 * X;
  beta = 0.425;
  sigma = 3.73;

  //Explore effect of Smax
  //Look at given imax and jmax, then increase Smax
  std::ofstream sigmabeta("./sigmabeta.txt");
  for (int i = 0; i < 11; i++) {
    sigmabeta << i * 0.5 << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma = i*0.5, R, kappa, mu, C, alpha, 0.2, iMax, jMax, sMax, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma = i * 0.5, R, kappa, mu, C, alpha, 0.8, iMax, jMax, sMax, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma = i * 0.5, R, kappa, mu, C, alpha, 1.2, iMax, jMax, sMax, tol, omega, iterMax) <<
      "\n";
  }
  cout << "DONE V FUNC S_MAX" << endl;
}
void GetSmax() {
  //Explore effect of Smax
  //Look at given imax and jmax, then increase Smax
  std::ofstream V_function_sMax("./V_function_sMax.txt");
  for (int i = 2; i < 20; i++) {
    // declare and initialise Black Scholes parameters - Currently looking at a solution we can get a definite answer for
    double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.083333333333,
      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;
    //
    int iterMax = 10000, iMax = 100, jMax = 200;
    double S0 = X;
    beta = 0.425;
    sigma = 3.73;
    V_function_sMax << i << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 20, 10*i, i * X, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 10*i, i * X, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 10*i, i * X, tol, omega, iterMax) <<
      "\n";
  }
  cout << "DONE V FUNC S_MAX" << endl;
}
//This attempts to help find the most efficient value for imax, jmax and Smax
void GetEfficientResult() {
  // declare and initialise Black Scholes parameters - Currently looking at a solution we can get a definite answer for
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.083333333333,
    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;
  //
  int iterMax = 10000, iMax = 100, jMax = 200;
  double S0 = X;
  beta = 0.425;
  sigma = 3.73;
  iMax = 26;
  jMax = 40;
  double sMax = 10 * X;
  cout <<
    "imax  = " << iMax << "   " <<
    "jmax  = " << jMax << "   " <<
    "Smax  = " << sMax << "   ";
  auto start = high_resolution_clock::now();
  double V1 = crank_nicolson_E_LAG(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, sMax, tol, omega, iterMax);
  auto stop = high_resolution_clock::now();
  auto duration1 = duration_cast<microseconds>(stop - start);
  cout << "OPTION VALUE = " << V1 << "  ";
  cout << "DURATION (microseconds): " << duration1.count() << endl;

  cout <<
    "imax  = " << iMax << "   " <<
    "jmax  = " << jMax << "   " <<
    "Smax  = " << sMax << "   ";
```

```
  start = high_resolution_clock::now();
  V1 = crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, 6*jMax, sMax, tol, omega, iterMax);
  stop = high_resolution_clock::now();
  duration1 = duration_cast<microseconds>(stop - start);
  cout << "OPTION VALUE = " << V1 << "  ";
  cout << "DURATION (microseconds): " << duration1.count() << endl;
}

//This code creates csv files to allow us to explore the effect of imax, jmax and smax on time
void GetTimeData() {
  // declare and initialise Black Scholes parameters - Currently looking at a solution we can get a definite answer for
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.08333333,
    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;
  //
  int iterMax = 10000, iMax = 100, jMax = 200;
  double S0 = X;
  beta = 0.425;
  sigma = 3.73;
  //Get data for increasing smax with time
  //Look at given imax and jmax, then increase Smax
  std::ofstream time_sMax("./time_sMax.txt");
  for (int i = 4; i < 20; i++) {
    time_sMax << i * X << " , ";
      auto start = high_resolution_clock::now();
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, i * X, tol, omega, iterMax);
      auto stop = high_resolution_clock::now();
      auto duration = duration_cast<milliseconds>(stop - start);
      time_sMax << duration.count() << "\n";
  }

  //Get data for increasing imax with time
    //Look at given imax and jmax, then increase Smax
  std::ofstream time_iMax("./time_iMax.txt");
  for (int i = 0; i < 200; i++) {
    time_iMax << i << " , ";
    auto start = high_resolution_clock::now();
    crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, jMax, S_max, tol, omega, iterMax);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
    time_iMax << duration.count() << "\n";
  }
  cout << "DONE iMax as function of time" << endl;
  //Get data for increasing jmax with timw
  std::ofstream time_jMax("./time_jMax.txt");
  for (int i = 1; i < 200; i++) {
    time_jMax << i << " , ";
    auto start = high_resolution_clock::now();
    crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, i, S_max, tol, omega, iterMax);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
    time_jMax << duration.count() << "\n";

  }

}


//This populates the rest of the csv files on european bond data
void getEurobondData() {
  // declare and initialise Black Scholes parameters - Currently looking at a solution we can get a definite answer for
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.08333333,
    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;
  //
  int iterMax = 10000, iMax = 100, jMax = 25;
  beta = 0.425;
  sigma = 3.73;
  double S0 = X;
  double V1 = 0, V2 = 0;
```

```
  //Accuracy code

  for (int i = 1; i < 400; i++) {

    auto start = high_resolution_clock::now();
    V1 = crank_nicolson_E_LAG(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 26, 40, 5 * X, tol, omega, iterMax);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    if (abs(V2 - V1) < 1e-5) {
      cout << "S_max = " << i << endl;
      cout << "OPTION VALUE = " << V1 << endl;
      cout << "DURATION (microseconds):" << duration.count() << endl;
      break;
    }
    else {
      V2 = V1;
    }
  }
  //



  // declare and initialise grid paramaters
  //int iMax = 100, jMax = 25;
  // declare and initialise local variables (ds,dt)
  //double S_max = 10 * X;
  //int iterMax = 5000000;
  //double tol = 1.e-7, omega = 1.;

  //Checking value against theory
  std::ofstream analytical("./analytical.txt");
  for (int s = 1; s <= 300; s++) {
    analytical << s << " , " << crank_nicolson_E_LINEAR(s, X, F, T, r, 3.73, R, 0, mu, C, alpha, 1., iMax, jMax, S_max, tol, omega, iterMax) << "\n";
  }

  //Create graph of varying  S and optionvalue
  int length = 50;
  double S_maxi = 4 * X;
  std::ofstream V_function_S("./V_function_S.txt");
  std::ofstream V_function_beta("./V_function_beta.txt");

  for (int j = 1; j <= length - 1; j++) {
    //Plottting V(S,t) as a function of S (t=0) for two cases
    beta = 1.;
    sigma = 0.369;
    //Puts value of S, and value of stock for different parameters into csv file
    V_function_S << j * S_maxi / length << " , " << crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 1., iMax, jMax,
      S_max, tol, omega, iterMax) << " , " << crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 3.73, R, kappa, mu, C, alpha, 0.425, iMax, 35,
        S_max, tol, omega, iterMax) << "\n";


    //"You may wish to explore different values of beta and sigma?" Do research before implementing this. But preliminary (BETA =1)
    V_function_beta << j * S_maxi / length << " , " <<
      crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 3., iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 2, iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 1, iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 0.5, iMax, jMax, S_max, tol, omega, iterMax) << " , " <<
      //crank_nicolson2(j * S_maxi / length, X, F, T, r, 0.369, R, kappa, mu, C, alpha, 0, iMax, jMax, S_max, tol, omega, iterMax) <<
      "\n";

  }
  cout << "DONE V FUNC S" << endl;

  //Assume now,
//  double S0 = X;
```

```
  beta = 0.425;
  sigma = 3.73;

  //Explore effect of Smax
  //Look at given imax and jmax, then increase Smax
  std::ofstream V_function_sMax("./V_function_sMax.txt");
  for (int i = 6; i < 20; i++) {
    V_function_sMax << i << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 50 * i,
     i * X, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 2 * i,
    i * X, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(X, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 100, 2 * i,
    i * X, tol, omega, iterMax) <<
      "\n";
  }
  cout << "DONE V FUNC S_MAX" << endl;
  //Explore effect of imax
  //Look at given imax and jmax, then increase Smax
  std::ofstream V_function_iMax("./V_function_iMax.txt");
  for (int i = 0; i < 200; i++) {
    V_function_iMax << i << " , " <<
      crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, 15,
    S_max, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, 25,
    S_max, tol, omega, iterMax) << " , " <<
      crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, i, 50,
    S_max, tol, omega, iterMax) << " , " <<
      "\n";
  }
  cout << "DONE V FUNC iMax" << endl;
  //Explore effect of jmax
  //Look at given imax and jmax, then increase Smax
  std::ofstream V_function_jMax("./V_function_jMax.txt");
  for (int i = 1; i < 200; i++) {
    V_function_jMax << i << " , " <<
      crank_nicolson_E_LINEAR(S0, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, 20, i,
    S_max, tol, omega, iterMax) <<
      "\n";
  }
  cout << "DONE V FUNC jMax" << endl;

}
```

4.2 American Style Convertible Bond code

```
#pragma once
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
#include <chrono>
#include <iomanip>
using namespace std::chrono;
using namespace std;

std::vector<double> thomasSolve(const std::vector<double>& a, const std::vector<double>& b_,
    const std::vector<double>& c, std::vector<double>& d)
{
  int n = a.size();
  std::vector<double> b(n), temp(n);
  // initial first value of b
  b[0] = b_[0];
  for (int j = 1; j < n; j++)
  {
    b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
    d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
  }
  // calculate solution
  temp[n - 1] = d[n - 1] / b[n - 1];
  for (int j = n - 2; j >= 0; j--)
    temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];
  return temp;
}

```

```
void sorSolve_AM(const std::vector<double>& a, const std::vector<double>& b, const std::vector
    <double>& c, const std::vector<double>& rhs,
  std::vector<double>& x, int iterMax, double tol, double omega, int& sor, double dS, double
    cp, double t0, int i, double dt)
{
  // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
  int n = a.size() - 1;
  // sor loop
  for (sor = 0; sor < iterMax; sor++)
  {
    double error = 0.;
    // implement sor in here
    {
      double y = (rhs[0] - c[0] * x[1]) / b[0];
      if (i * dt < t0) {
        x[0] = max(0., min(cp, x[0] + omega * (y - x[0])));
      }
      else {
        x[0] = max(x[0] + omega * (y - x[0]), 0.);
      }
      error += (y - x[0]) * (y - x[0]);
    }
    for (int j = 1; j < n; j++)
    {
      double y = (rhs[j] - a[j] * x[j - 1] - c[j] * x[j + 1]) / b[j];
      if (i * dt < t0) {
        x[j] = max(min(x[j] + omega * (y - x[j]),cp), j * dS);
      }
      else {
        x[j] = max(x[j] + omega * (y - x[j]), j * dS);
      }
      error += (y - x[j]) * (y - x[j]);
    }
    {
      double y = (rhs[n] - a[n] * x[n - 1]) / b[n];
      if (i * dt < t0) {
        x[n] = max(min(x[n] + omega * (y - x[n]),cp), n * dS);
      }
      else {
        x[n] = max(x[n] + omega * (y - x[n]), n * dS);
      }
      error += (y - x[n]) * (y - x[n]);
    }
    // make an exit condition when solution found
    if (error < tol)
      break;
  }
  if (sor >= iterMax)
  {
    std::cout << " Error NOT converging within required iterations\n";
  }
}

/* Solution code for the Crank Nicolson Finite Difference
search for COURSEWORK EDIT for parts that needed to be altered for the coursework
 */
double crank_nicolson_AM_LINEAR(double S0, double X, double F, double T, double r, double
    sigma,
  double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
    int S_max, double tol, double omega, int iterMax, double cp, double t0)
{
  // declare and initialise local variables (ds,dt)
  cp = 67;
  double dS = S_max / jMax;
  double dt = T / iMax;
  // create storage for the stock price and option price (old and new)
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
```

```
    vNew[j] = max(F, R * S[j]);
  }
  // start looping through time levels
  for (int i = iMax - 1; i >= 0; i--)
  {
    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
    for (int j = 1; j <= jMax - 1; j++)
    {
      //
      a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j * dS) / (4 * dS));
      b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r / 2.);
      c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (theta - j * dS)) / (4. * dS));
      d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.))) * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS)) * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
    }
    double A = R * exp((kappa + r) * (i * dt - T));
    double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
    a[jMax] = 0;
    b[jMax] = 1;
    c[jMax] = 0;
    d[jMax] = jMax * dS * A + B;
    // solve matrix equations with SOR
    int sor;
    for (sor = 0; sor < iterMax; sor++)
    {
      double error = 0.;
      // implement sor in here
      {
        double y = (d[0] - c[0] * vNew[1]) / b[0];
        y = vNew[0] + omega * (y - vNew[0]);
        if (i * dt < t0)
        {
          y = max(0., min(cp, y));
        }
        else
        {
          y = std::max(y, R * S[0]);
        }
        error += (y - vNew[0]) * (y - vNew[0]);
        vNew[0] = y;
      }
      for (int j = 1; j < jMax; j++)
      {
        double y = (d[j] - a[j] * vNew[j - 1] - c[j] * vNew[j + 1]) / b[j];
        y = vNew[j] + omega * (y - vNew[j]);
        if (i * dt < t0)
        {
          y = max(min(y, cp), j * dS);
        }
        else
        {
          y = std::max(y, R * j * dS);
        }
        error += (y - vNew[j]) * (y - vNew[j]);
        vNew[j] = y;
      }
      {
        double y = (d[jMax] - a[jMax] * vNew[jMax - 1]) / b[jMax];
        y = vNew[jMax] + omega * (y - vNew[jMax]);
        if (i * dt < t0)
        {
          y = max(min(y, cp), jMax * dS);
        }
        else
```

```
        {
          y = std::max(y, R * jMax * dS);
        }
        error += (y - vNew[jMax]) * (y - vNew[jMax]);
        vNew[jMax] = y;
      }
      // make an exit condition when solution found
      if (error < tol)
        break;
    }
    if (sor >= iterMax)
    {
      std::cout << " Error NOT converging within required iterations\n";
      std::cout.flush();
      throw;
    }

    if (sor == iterMax)
      return -1;

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

  // output the estimated option price
  double sum;
  {
    int jStar = S0 / dS;
    sum = 0.;
    if (jStar > 0 && jStar < jMax) {
      sum += ((S0 - S[jStar]) * (S0 - S[jStar + 1]) / (2 * dS * dS)) * vNew[jStar - 1];
      sum -= ((S0 - S[jStar - 1]) * (S0 - S[jStar + 1]) / (dS * dS)) * vNew[jStar];
      sum += ((S0 - S[jStar - 1]) * (S0 - S[jStar]) / (2 * dS * dS)) * vNew[jStar + 1];
    }
    else {
      sum += (S0 - S[jStar]) / dS * vNew[jStar + 1];
      sum += (S[jStar + 1] - S0) / dS * vNew[jStar];
    }
  }
  return sum;
}

/* This code seems to run much faster when in a separate solution without header files, maybe just copy and paste this
 */
double crank_nicolson_AM_FAST(double S0, double X, double F, double T, double r, double sigma,
  double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax, int S_max, double tol, double omega, int iterMax, int& sorCount, double t0)
{
  // declare and initialise local variables (ds,dt)
  double cp = 67.;

  double dS = S_max / jMax;
  double dt = T / iMax;
  // create storage for the stock price and option price (old and new)
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
    vNew[j] = max(F, R * S[j]);
  }
  // start looping through time levels
  for (int i = iMax - 1; i >= 0; i--)
  {
    //if (i * dt < t0) { dt = t0 / iMax; }
    //if (i * dt >= t0) { dt = (T - t0) / iMax; }
    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
```

```
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
    for (int j = 1; j <= jMax - 1; j++)
    {
      //
      a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j
    * dS) / (4 * dS));
      b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r
    / 2.);
      c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (
    theta - j * dS)) / (4. * dS));
      d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.)))
     * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS))
     * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
    }
    double A = R * exp((kappa + r) * (i * dt - T));
    double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T))
     - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
    a[jMax] = 0;
    b[jMax] = 1;
    c[jMax] = 0;
    d[jMax] = jMax * dS * A + B;
    // solve matrix equations with SOR
    int sor;
    for (sor = 0; sor < iterMax; sor++)
    {
      double error = 0.;
      // implement sor in here
      {
        double y = (d[0] - c[0] * vNew[1]) / b[0];
        y = vNew[0] + omega * (y - vNew[0]);
        if (i * dt < t0)
        {
          y = max(0., min(cp, y));
        }
        else
        {
          y = std::max(y, R * S[0]);
        }
        error += (y - vNew[0]) * (y - vNew[0]);
        vNew[0] = y;
      }
      for (int j = 1; j < jMax; j++)
      {
        double y = (d[j] - a[j] * vNew[j - 1] - c[j] * vNew[j + 1]) / b[j];
        y = vNew[j] + omega * (y - vNew[j]);
        if (i * dt < t0)
        {
          y = max(min(y, cp), j * dS);
        }
        else
        {
          y = std::max(y, R * j * dS);
        }
        error += (y - vNew[j]) * (y - vNew[j]);
        vNew[j] = y;
      }
      {
        double y = (d[jMax] - a[jMax] * vNew[jMax - 1]) / b[jMax];
        y = vNew[jMax] + omega * (y - vNew[jMax]);
        if (i * dt < t0)
        {
          y = max(min(y, cp), jMax * dS);
        }
        else
        {
          y = std::max(y, R * jMax * dS);
        }
        error += (y - vNew[jMax]) * (y - vNew[jMax]);
        vNew[jMax] = y;
      }
      // make an exit condition when solution found
      if (error < tol)
```

```
        break;
    }
    if (sor >= iterMax)
    {
      std::cout << " Error NOT converging within required iterations\n";
      std::cout.flush();
      throw;
    }

    if (sorCount == iterMax)
      return -1;

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

  // output the estimated option price
  double optionValue;

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

  return optionValue;
}

/* Template code for the Crank Nicolson Finite Difference
 */
double crank_nicolson_penalty(double S0, double X, double F, double T, double r, double sigma,
  double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax,
    int S_max, double tol, double omega, int iterMax, int& sorCount, double t0)
{
  // declare and initialise local variables (ds,dt)
  double cp = 67.;
  //dS calculated as before
  double dS = S_max / jMax;
  //What is f used for?
  double f = (T - t0) / T;
  //STILL T/iMax
  double dt = (T - t0) / (iMax * f);
  // create storage for the stock price and option price (old and new)
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
    vNew[j] = max(F, R * S[j]);
  }
  // start looping through time levels
  for (int i = iMax; i >= 0; i--)
  {
    //If you are before t-, change increments to
    if (i * dt < t0)
    {
      dt = t0 / (iMax * (1 - f));
    }

    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
    for (int j = 1; j <= jMax - 1; j++)
    {
```

```
      //
      a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j * dS) / (4 * dS));
      b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r / 2.);
      c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (theta - j * dS)) / (4. * dS));
      d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) / (4. * pow(dS, 2.))) * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((kappa * (theta - j * dS)) / (4. * dS)) * (vOld[j + 1] - vOld[j - 1])) + ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
    }
    double A = R * exp((kappa + r) * (i * dt - T));
    double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R * exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt) / (alpha + r);
    a[jMax] = 0;
    b[jMax] = 1;
    c[jMax] = 0;
    d[jMax] = jMax * dS * A + B;
    double penalty = 1.e8;
    int q;
    for (q = 0; q < 100000; q++)
    {
      vector<double> bHat(b), dHat(d);
      for (int j = 1; j < jMax; j++)
      {
        if (i * dt < t0)
        {
          if (vNew[j] > max(R * S[j], cp))
          {
            bHat[j] = b[j] - penalty;
            dHat[j] = d[j] - penalty * max(R * S[j], cp);

          }
        }
        else
        {
          // turn on penalty if V < RS
          if (vNew[j] < R * S[j])
          {
            bHat[j] = b[j] - penalty;
            dHat[j] = d[j] - penalty * R * S[j];
          }
        }
      }
      // solve matrix equations with SOR
      vector<double> y = thomasSolve(a, bHat, c, dHat);
      // calculate difference from last time
      double error = 0.;
      for (int j = 0; j <= jMax; j++)
        error += fabs(vNew[j] - y[j]);
      vNew = y;
      if (error < 1.e-8)
        break;
    }
    if (q == 100000)
    {
      std::cout << " Error NOT converging within required iterations\n";
      std::cout.flush();
      throw;
    }

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

  // output the estimated option price
  //Why 4?
  return lagrangeInterpolation(vNew, S, S0, 4);
}


//ALlows you to extract information in csv file for efficiency of penalty method
void GetPenaltyEfficiency()
{
  // declare and initialise Black Scholes parameters - Currently looking at a solution we can
```

```
    get a definite answer for
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333,
    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;
  double t0 = 1.2448;
  //

  int iterMax = 100000;
  //Create graph of varying S0 and beta and bond
  int length = 300;
  int sorCount;
  double S0 = X;
  std::ofstream outFile("ameribond_eff.txt");
  double oldResult = 0, oldDiff = 0;
  double S = X;
  int iMax = 100;
  int jMax = 100;
  S_max = 200 * X;
  //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;

  for (int n = 100; n <= 10000; n *= 2)
  {
    //Set aprameters for iteration
    iMax = n;
    jMax = n;
    S_max = n / 20 * X;

    auto t1 = std::chrono::high_resolution_clock::now();
    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);
    double diff = result - oldResult;
    auto t2 = std::chrono::high_resolution_clock::now();
    auto time_taken = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();

    outFile << n << "," << setprecision(10) << result << "," << time_taken << "," <<
    setprecision(3) << oldDiff / diff << "\n";
    cout << "RESULT : " << result << "  TIME: " << time_taken << "," << setprecision(3) <<
    oldDiff / diff << "\n";

    oldDiff = diff;
    oldResult = result;
  }

}

void getAmeribondEfficiency() {
  // declare and initialise Black Scholes parameters - Currently looking at a solution we can
    get a definite answer for
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333,
    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;
  //
  int iterMax = 100000, iMax = 40, jMax = 25;
  beta = 0.425;
  sigma = 3.73;
  double S0 = X;
  double t0 = 1.2448, cp = 67;

  std::ofstream outFile5("american_varying_smax.txt");
  tol = 1.e-7;
  for (int i = 1; i <= 1; i++)
  {
    double jMax = 300;
    double S = X;
    int sorCount;
    auto t1 = std::chrono::high_resolution_clock::now();
    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);
    auto t2 = std::chrono::high_resolution_clock::now();
    auto time_taken =
      std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
      .count();
    cout << result << "," << time_taken << "\n";
    t1 = std::chrono::high_resolution_clock::now();
```

```
    result = crank_nicolson_AM_LINEAR(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);
    t2 = std::chrono::high_resolution_clock::now();
    time_taken =
      std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
      .count();
    cout << result << "," << time_taken << "\n";
  }
  outFile5.close();

}


//Get data for ameribond. Gets information for first part of American Bond graphs
void getAmeribondData() {
  // declare and initialise Black Scholes parameters - Currently looking at a solution we can
    get a definite answer for
  double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.0833333333,
    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 = 15 * X;
  //
  int iterMax = 10000, iMax = 700, jMax = 700;
  beta = 0.425;
  sigma = 3.73;
  double S0 = X;
  double t0 = 1.2448, cp = 67;
  int sor;
  //Checking value against theory

  std::ofstream analytical("./Ameribond1.txt");
  for (int s = 1; s <= 100; s++) {
    analytical << s << " , " << crank_nicolson_penalty(s, X, F, T, r, sigma, R, kappa, mu, C,
    alpha, beta, iMax, jMax, S_max, tol, omega, iterMax,sor, t0) << "\n";
  }
  cout << "AMERICAN BOND PART 1 DONE" << endl;

  //Checking how the value changes with different values of r
  std::ofstream r_file("./Ameribond_r.txt");
  for (int s = 1; s <= 67; s++) {
    r_file << s << " , " <<
      crank_nicolson_penalty(s, X, F, T, 0.0019, sigma, R, kappa, mu, C, alpha, beta, iMax,
    jMax, S_max, tol, omega, iterMax,sor, t0) << " , " <<
      crank_nicolson_penalty(s, X, F, T, 0.0038, sigma, R, kappa, mu, C, alpha, beta, iMax,
    jMax, S_max, tol, omega, iterMax,sor, t0) << " , " <<
      crank_nicolson_penalty(s, X, F, T, 0.0057, sigma, R, kappa, mu, C, alpha, beta, iMax,
    jMax, S_max, tol, omega, iterMax,sor, t0) <<
      "\n";

  }

  cout << "AMERICAN BOND PART 2 DONE" << endl;

}
```