Student Id: 9910821

Introduction

This report is a two part study of:

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

II) An extension to the above contract where the holder is able to exercise the decision to convert the bond in stock at **any time before** the maturity of the contract. Moreover a put option will be embedded in this contract

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

1 European Type Option Convertible Bond

The PDE describing such a convertible bond contract is given by

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

with the ansatz

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

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

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

and

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

Finally we have the last part of the numerical scheme as

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

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

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

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

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

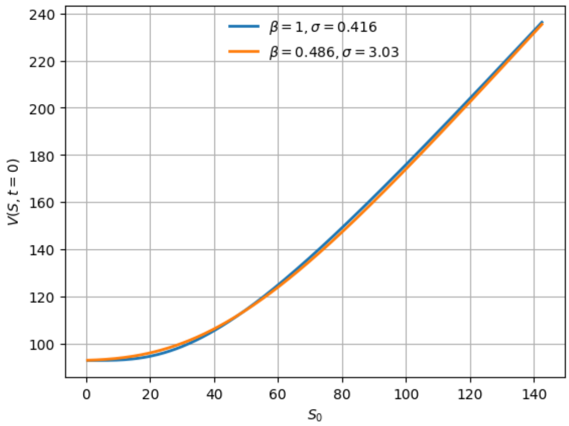


Figure 1: Value of the convertible bond $V(S, t = 0)$ against inital underlying asset price at time S_0 for two combinations of β and σ.

1.1 Investigating β and σ

For the rest of this section assume these values were used unless otherwise specified: $T = 2$, $F = 95$, $R = 2$, $r = 0.0229$, $\kappa = 0.125$, $\mu = 0.0113$, $X = 47.66$, $C = 1.09$, $\alpha = 0.02$, $\beta = 0.486$ and $\sigma = 3.03$. For the SOR method a value of $\omega = 1.4$ was used with a maximum cap of iterations of 100,000. The value of the option $V(S, t)$ was investigated as a function of the initial underlying asset price S_0 for two cases:

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

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

The Crank-Nicolson method with the numerical scheme as calculated previously, combined with a SOR iterative method of solving the matrix equation, was implemented in code. This produced the plots seen in Figure 1. The two configurations were therefore found to have the same effect and produce plots for the price of the bond which were very close. This prompted further analysis on the linked relationship between β, σ and $V(S, t)$. A 3D graph of the value of the portfolio for a particular S_0, here chosen to be equal to X, and the two other parameters was plotted.

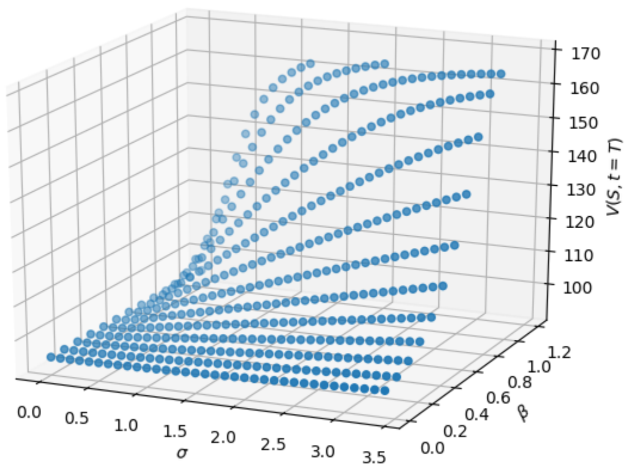


Figure 2: Value of the convertible bond $V(S = X, t = T)$ against parameters β and σ.

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

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

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

1.2 Varying step sizes

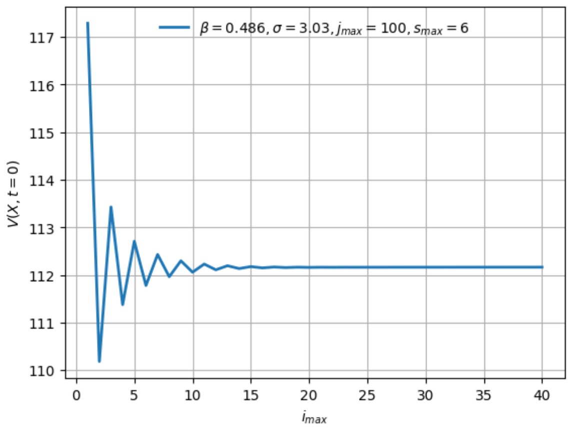


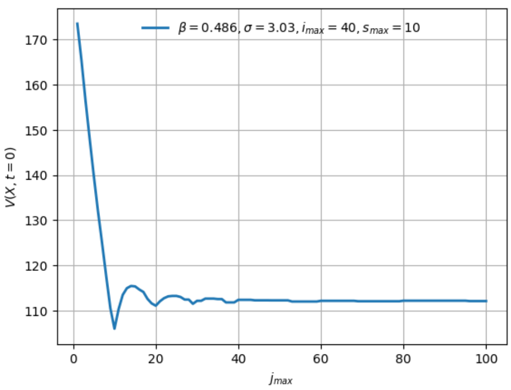
Figure 3: Trend of the convertible bond $V(S = X, t = T)$ as parameter i_{max} is varied.

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

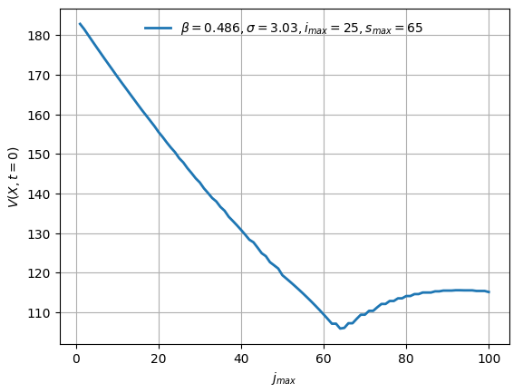
When it came to varying j_{max} which is the number of steps in S per timestep, it was noticed that since the stepsize in S is calculated by dividing S_{max} by the number of steps then these had to go hand in hand when varying one of them. Figure 4 illustrates this very clearly. Keeping the range of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

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

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



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



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

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

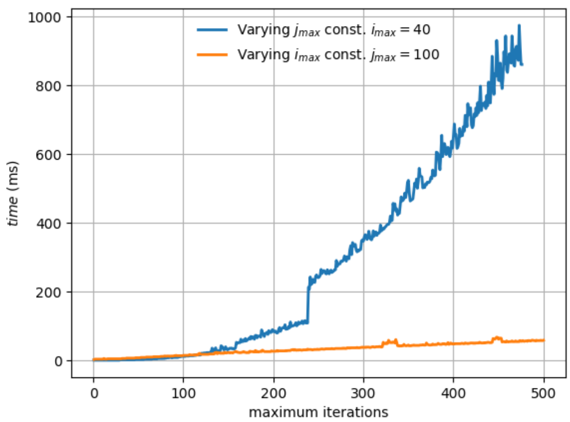


Figure 5: Variation of the time required to calculate the convertible bond price as j_{max} and i_{max} are varied.

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

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

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

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

2 American Type Option Convertible Bond with Embedded Option

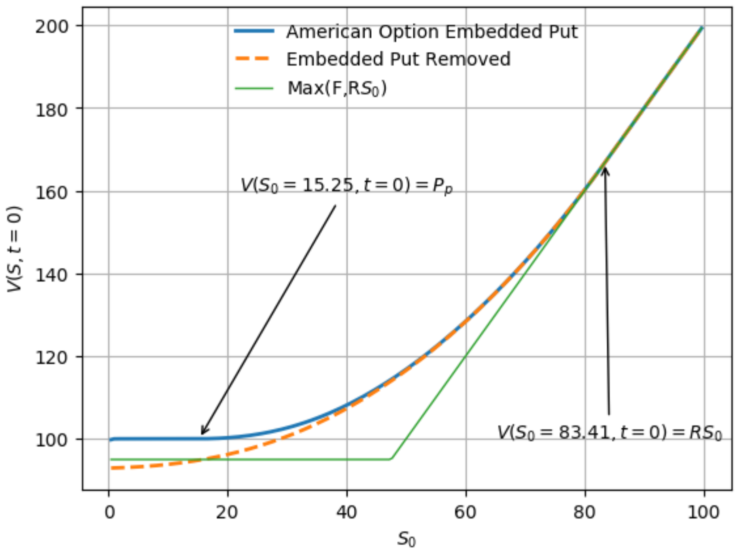


Figure 6: Value of the american type option convertible bond $V(S, t = 0)$ against inital underlying asset price S_0 with and without the embedded put option.

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

$$V \geq RS \tag{25}$$

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

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

must hold.

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

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

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

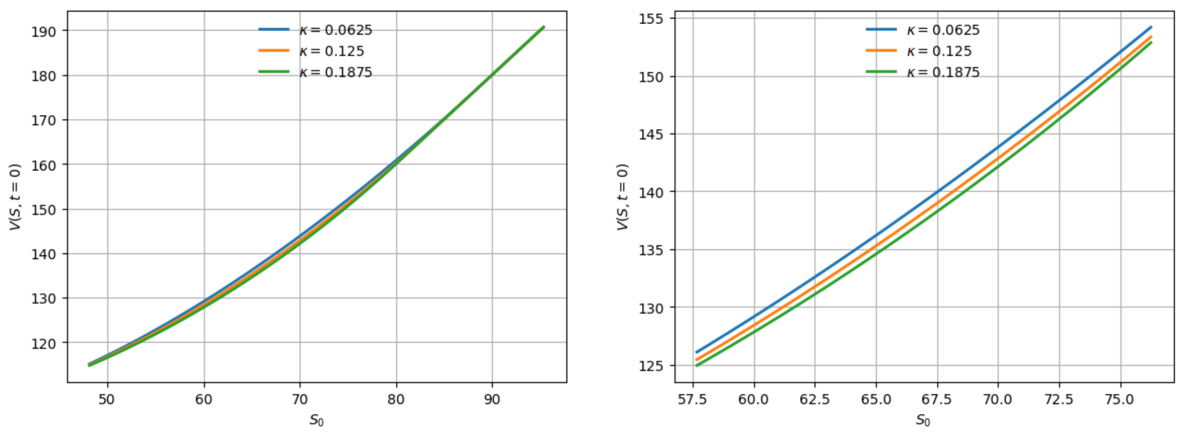


Figure 7: Value of the american type option convertible bond $V(S, t = 0)$ with embedded put option for different values of parameter κ. Right side plot is a zoomed version of the left side plot.

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

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

$$\Delta t = \frac{T - t_0}{i_{max} f} \qquad \text{for} \qquad t_0 < t \leq T \tag{27}$$

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

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

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

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

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

References

[1] C. Thierfelder, "The trending ornstein-uhlenbeck process and its applications in mathematical finance," *Mathematical Finance*, 2015.

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

[3] N. H. Chan and C. T. Ng, *Fractional constant elasticity of variance model*, vol. Volume 52 of *Lecture Notes–Monograph Series*, pp. 149–164. Beachwood, Ohio, USA: Institute of Mathematical Statistics, 2006.

[4] M. Choudhry, "51 - interest-rate models i," in *The Bond and Money Markets*, Securities Institution Professional Reference Series, pp. 873 – 887, Oxford: Butterworth-Heinemann, 2001.

[5] P. A. Forsyth and K. R. Vetzal, "Quadratic convergence for valuing american options using a penalty method," *SIAM Journal on Scientific Computing*, vol. 23, no. 6, pp. 2095–2122, 2002.

[6] L. X. Li, *Pricing Convertible Bonds using Partial Differential Equations.* University of Toronto, 2005.

Appendix 1: European Type Option Code

Portfolio Pricing Program Listing

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

/* Code for the Crank Nicolson Finite Difference
 */
double crank_nicolson(double S0, double X, double F, double T, double r, double sigma,
                        double R, double kappa, double mu, double C, double alpha, double beta, int iMax, int jMax, int S_max, double tol, double omega, int iterMax, int &sorCount)
{
  // declare and initialise local variables (ds,dt)
  double dS = S_max / jMax;
  double dt = T / iMax;
  // create storage for the stock price and option price (old and new)
  vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
  // setup and initialise the stock price
  for (int j = 0; j <= jMax; j++)
  {
    S[j] = j * dS;
  }
  // setup and initialise the final conditions on the option price
  for (int j = 0; j <= jMax; j++)
  {
    vOld[j] = max(F, R * S[j]);
    vNew[j] = max(F, R * S[j]);
  }
  // start looping through time levels
  for (int i = iMax - 1; i >= 0; i--)
  {
    // declare vectors for matrix equations
    vector<double> a(jMax + 1), b(jMax + 1), c(jMax + 1), d(jMax + 1);
    // set up matrix equations a[j]=
    double theta = (1 + mu) * X * exp(mu * i * dt);
    a[0] = 0;
    b[0] = (-1 / dt) - (r / 2) - (kappa * theta / dS);
    c[0] = (kappa * theta / dS);
    d[0] = (-C * exp(-alpha * i * dt)) + (vOld[0] * (-(1 / dt) + (r / 2)));
    for (int j = 1; j <= jMax - 1; j++)
    {
      //
      a[j] = (pow(sigma, 2) * pow(j * dS, 2 * beta) / (4 * pow(dS, 2))) - (kappa * (theta - j * dS) / (4 * dS));
      b[j] = (-1 / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (2. * pow(dS, 2))) - (r / 2.);
      c[j] = ((pow(sigma, 2.) * pow(j * dS, 2. * beta)) / (4. * pow(dS, 2.))) + ((kappa * (theta - j * dS)) / (4. * dS));
      d[j] = (-vOld[j] / dt) - ((pow(sigma, 2.) * pow(j * dS, 2. * beta) /
```

```
    (4. * pow(dS, 2.))) * (vOld[j + 1] - 2. * vOld[j] + vOld[j - 1])) - (((
    kappa * (theta - j * dS)) / (4. * dS)) * (vOld[j + 1] - vOld[j - 1])) +
    ((r / 2.) * vOld[j]) - (C * exp(-alpha * dt * i));
    }
    double A = R * exp((kappa + r) * (i * dt - T));
    double B = -X * A + C * exp(-alpha * i * dt) / (alpha + r) + X * R *
    exp(r * (i * dt - T)) - C * exp(-(alpha + r) * T + r * i * dt) / (alpha
    + r);
    a[jMax] = 0;
    b[jMax] = 1;
    c[jMax] = 0;
    d[jMax] = jMax * dS * A + B;

    // solve matrix equations with SOR
    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]);
        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]);
        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]);
        error += (y - vNew[jMax]) * (y - vNew[jMax]);
        vNew[jMax] = y;
      }
      // make an exit condition when solution found
      if (error < tol * tol)
      {
        sorCount += sor;
        break;
      }
    }
    if (sor >= iterMax)
    {
      std::cout << " Error NOT converging within required iterations\n";
      std::cout.flush();
      throw;
    }
    vOld = vNew;
  }
  // finish looping through time levels

  // output the estimated option price
  double optionValue;
  //linear interp
```

```cpp
  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;
  // alternatively
  //optionValue = lagrangeInterpolation(vNew, S, S0, 4);
  return optionValue;
}

int main()
{
  // Initial condition
  double T = 2., F = 95., R = 2., r = 0.0229, kappa = 0.125, altSigma =
    0.416,
         mu = 0.0213, X = 47.66, C = 1.09, alpha = 0.02, beta = 0.486,
    sigma = 3.03, tol = 1.e-8, omega = 1.4;

  int iterMax = 100000, iMax = 200, jMax = 200, S_max = 6 * X;
  int length = 300;
  double S_range = 3 * X;
  int sor;

  // Run to obtain 3d graph
  std::ofstream outFile9("./data/varying_s_sigma_beta.csv");
  for (double altSigma = 0; altSigma < 3.5; altSigma += 0.1)
  {
    for (double beta = 0; beta < 1.3; beta += 0.1)
    {
      double S0 = X;
      outFile9 << beta << " , " << altSigma << " , " << S0 << " , " <<
    crank_nicolson(S0, X, F, T, r, altSigma, R, kappa, mu, C, alpha, beta,
    iMax, jMax, S_max, tol, omega, iterMax, sor) << "\n";
    }
  }
  outFile9.close();

  // Run to obtain varying configurations of beta, sigma graph
  std::ofstream outFile1("./data/varying_s_beta_1.csv");
  std::ofstream outFile2("./data/varying_s_beta_0_4.csv");
  for (int j = 1; j <= length - 1; j++)
  {
    vector<double> gamma(jMax + 1);
    outFile1 << j * S_range / length << " , " << crank_nicolson(j * S_range
     / length, X, F, T, r, altSigma, R, kappa, mu, C, alpha, 1, iMax, jMax,
    S_max, tol, omega, iterMax, sor) << "\n";
    outFile2 << j * S_range / length << " , " << crank_nicolson(j * S_range
     / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax,
    S_max, tol, omega, iterMax, sor) << "\n";
  }
  outFile1.close();
  outFile2.close();

  // Run to obtain varying imax graph
  std::ofstream outFile3("./data/varying_imax.csv");
  jMax = 100;
  for (iMax = 1; iMax <= 500; iMax += 1)
  {
```

```
    double S = X;
    auto t1 = std::chrono::high_resolution_clock::now();
    double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
    auto t2 = std::chrono::high_resolution_clock::now();
    auto time_taken =
        std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
            .count();
    outFile3 << S_max << "," << iMax << "," << jMax << "," << S << " , " << std::fixed << result << "," << time_taken << "\n";
  }
  outFile3.close();

  // Run to obtain varying smax per varying jmax graph
  for (int s_Mult = 10; s_Mult <= 10; s_Mult += 1)
  {
    double S = X;
    S_max = s_Mult * X;
    string title = "./data/smax_jmax/" + to_string(s_Mult) + "_varying_jmax.csv";
    std::ofstream outFile4(title);
    iMax = 40;
    for (jMax = 1; jMax <= 500; jMax += 1)
    {
      auto t1 = std::chrono::high_resolution_clock::now();
      double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
      auto t2 = std::chrono::high_resolution_clock::now();
      auto time_taken =
          std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
              .count();
      outFile4 << S_max << "," << iMax << "," << jMax << "," << S << " , " << std::fixed << result << "," << time_taken << "\n";
    }
    outFile4.close();
  }

  //Run to obtain graph of varying smax
  std::ofstream outFile5("./data/varying_smax.csv");
  for (int s_Mult = 10; s_Mult <= 50; s_Mult += 1)
  {
    jMax = s_Mult * 10;
    double S = X;
    S_max = s_Mult * X;
    int sorCount;
    auto t1 = std::chrono::high_resolution_clock::now();
    double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount);
    auto t2 = std::chrono::high_resolution_clock::now();
    auto time_taken =
        std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
            .count();
    outFile5 << S_max << "," << iMax << "," << jMax << "," << S << " , " << std::fixed << result << "," << time_taken << "\n";
  }
  outFile5.close();

  double oldResult = 0;
```

```
  double oldDiff = 0;
  for (int N = 100; N < 3200; N *= 2)
  {
    jMax = N;
    iMax = N;
    double S = X;
    double S_max = int(N / 30) * X;
    int sorCount{0};
    auto t1 = std::chrono::high_resolution_clock::now();
    double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
   alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount);
    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();
    cout << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
   setprecision(10) << result << "," << time_taken << "," << setprecision
   (3) << oldDiff / diff << "," << sorCount << "\n";
    oldDiff = diff;
    oldResult = result;
  }

  // Run to obtain graph to check with analytic value
  std::ofstream outFile6("./data/analytic.csv");
  S_max = 6 * X;
  iMax = 200, jMax = 200;
  for (int j = 1; j <= length - 1; j++)
  {
    outFile6 << j * S_range / length << " , " << crank_nicolson(j * S_range
    / length, X, F, T, r, altSigma, R, 0, mu, C, alpha, 1, iMax, jMax,
   S_max, tol, omega, iterMax, sor) << "\n";
  }
  outFile6.close();

  // Run to obtain final accurate value
  S_max = 58 * X;
  iMax = 800, jMax = 800;
  double S0 = X;
  auto t1 = std::chrono::high_resolution_clock::now();
  double result = crank_nicolson(S0, X, F, T, r, sigma, R, kappa, mu, C,
   alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
  auto t2 = std::chrono::high_resolution_clock::now();
  auto time_taken =
      std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
          .count();
  std::cout << setprecision(10) << result << "," << time_taken << endl;
}
```

Graphing Program Listing

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

X=47.66
R=2
F=95
```

```python
T=2.0
C = 1.09
alpha = 0.02
r = 0.0229
T = 2.
sigma = 0.416

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

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

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

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

    return call

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

plt.figure()
plt.grid()
plt.plot(variationData[0]['x'][:40],variationData[0]['y'][:40],label=r'$\beta=0.486,\sigma=3.03,j_{max}=%i,s_{max}=%i$'%(variationData[0]['jMax'],variationData[0]['sMax']),linewidth=2)
plt.xlabel(r'$i_{max}$')
plt.ylabel(r'$V(X,t=0)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/european_varying_imax.png',bbox_inches='tight',pad_inches=0.2)

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

```

fig, ax1 = plt.subplots()
ax1.set_xlabel(r's_{max} (multiples of X)')
ax1.set_ylabel(r'$V(X,t=0)$')
ax1.grid()
ax1.scatter(np.asarray(variationData[1]['x'][:20]),variationData[1]['y'][:20],label=r'$V(X,t=0)$ for $\beta=0.486,\sigma=3.03,i_{max}=%i$'%(variationData[1]['iMax']))
ax2 = ax1.twinx()
ax2.set_ylabel(r'$j_{max}$')
fig.tight_layout()
ax2.plot(np.asarray(variationData[1]['x'][:20]),variationData[1]['jMax'][:20],label=r'$j_{max}$',color="orange")
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2,loc='lower right',fancybox=False, framealpha=0.0)
plt.savefig('images/european_varying_smax_zoomed.png',bbox_inches='tight',pad_inches=0.2)

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

for smax in (10,65):
    currentData={'x':[], 'y':[], 'iMax':0, 'sMax':0}
    with open('data/smax_jmax/'+str(smax)+'_varying_jmax.csv', newline='\n') as csvfile:
        reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V'],quoting=csv.QUOTE_NONNUMERIC)
        for row in reader:
            currentData['x'].append(row['jMax'])
            currentData['y'].append(row['V'])
            currentData['iMax']=row['iMax']
            currentData['sMax']=int(row['sMax']/X)

    plt.figure()
    plt.plot(currentData['x'][:100],currentData['y'][:100],label=r'$\beta=0.486,\sigma=3.03,i_{max}=%i, s_{max}=%i$'%(currentData['iMax'],currentData['sMax']),linewidth=2)
    plt.xlabel(r'$j_{max}$')
    plt.ylabel(r'$V(X,t=0)$')
    plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
```

```python
    plt.grid()
    plt.savefig('images/smax_jmax/'+str(smax)+'_european_varying_jmax.png',
    bbox_inches='tight', pad_inches=0.2)
    plt.close()

allData=[]
with open('data/varying_s_beta_1.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.
    QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)

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

plt.figure()
plt.grid()
plt.plot(allData[0]['S'],allData[0]['V'],label=r'$\beta=1,\sigma=0.416$',
    linewidth=2)
plt.plot(allData[1]['S'],allData[1]['V'],label=r'$\beta=0.486,\sigma=3.03$'
    ,linewidth=2)
plt.xlabel(r'$S_0$')
plt.ylabel(r'$V(S,t=0)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/european_varying_s.png',bbox_inches='tight', pad_inches
    =0.2)

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


currentData={'S':[], 'V':[], 'beta':[], 'sigma':[]}
with open('data/varying_s_sigma_beta.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['beta','sigma','S','V'],
    quoting=csv.QUOTE_NONNUMERIC)
    for row in reader:
        if(row['V']==-1):
            continue
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
        currentData['beta'].append(row['beta'])
        currentData['sigma'].append(row['sigma'])

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(currentData['sigma'],currentData['beta'],currentData['V'])
ax.set_xlabel(r'$\sigma$')
ax.set_ylabel(r'$\beta$')
ax.set_zlabel(r'$V(X,t=0)$')
```

```
plt.show()
plt.savefig('images/european_varying_s_varying_sigma_varying_beta.png', bbox_inches='tight', pad_inches=0.2)

plt.figure()
plt.grid()
plt.xlabel(r'maximum iterations')
plt.ylabel(r'$time$ (ms)')

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


plt.plot(currentData['x'],currentData['time'],label=r'Varying $j_{max}$ const. $i_{max}=40$',linewidth=2)



variationData=[]
currentData={'x':[],'y':[],'jMax':0,'sMax':0,'time':[]}

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

plt.plot(currentData['x'],currentData['time'],label=r'Varying $i_{max}$ const. $j_{max}=100$',linewidth=2)

plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/european_time.png',bbox_inches='tight', pad_inches=0.2)
```

Appendix 2: American Type Option Code

Portfolio Pricing Program Listing

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
#include <chrono>
#include <iomanip>

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;
}

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;
}

/* Code for the Crank Nicolson Finite Difference with Penalty
 */
double crank_nicolson1(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 P = 100.;
  double dS = S_max / jMax;
  double f = (T - t0) / T;
  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 (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], P))
          {
            bHat[j] = b[j] - penalty;
            dHat[j] = d[j] - penalty * max(R * S[j], P);
          }
        }
        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)
      {
        sorCount += q;
        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
  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, 4);

  return optionValue;
}
/* Code for the Crank Nicolson Finite Difference with PSOR
 */
double crank_nicolson2(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 P = 100.;
  double dS = S_max / jMax;
  double f = (T - t0) / T;
  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 (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;
  // 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 = std::max(std::max(y, R * S[0]), P);
      }
      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 = std::max(std::max(y, R * j * dS), P);
      }
```

```
        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 = std::max(std::max(y, R * jMax * dS), P);
        }
        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 * tol)
      {
        sorCount += sor;
        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, 4);

  return optionValue;
}

int main()
```

```
{
  double T = 2., F = 95., R = 2., r = 0.0229, kappa = 0.125, altSigma = 0.416,
         mu = 0.0213, X = 47.66, C = 1.09, alpha = 0.02, beta = 0.486, sigma = 3.03, tol = 1.e-8, omega = 1., S_max = 13 * X;
  int iMax = 600;
  int jMax = 600;
  double t0 = 0.57245;

  int iterMax = 100000;
  int length = 300;
  double S_range = 3 * X;
  int sor;

  // Produces graph comparing embedded put option and without vs changing S0
  std::ofstream outFile1("./data/no_put_american_varying_s_beta_0_4.csv");
  std::ofstream outFile2("./data/american_varying_s_beta_0_4.csv");
  for (int j = 1; j <= length - 1; j++)
  {
    std::cout << j << std::endl;
    outFile1 << j * S_range / length << " , " << crank_nicolson1(j * S_range / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, 0.) << "\n";
    outFile2 << j * S_range / length << " , " << crank_nicolson1(j * S_range / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0) << "\n";
    outFile1.flush();
    outFile2.flush();
  }
  outFile1.close();
  outFile2.close();

  // Produces graph for different values of kappa vs changing S0
  std::ofstream outFile4("./data/american_varying_s_kappa_625.csv");
  std::ofstream outFile5("./data/american_varying_s_kappa_125.csv");
  std::ofstream outFile6("./data/american_varying_s_kappa_187.csv");

  for (int j = 1; j <= length - 1; j++)
  {
    std::cout << j << std::endl;
    double result1 = crank_nicolson1(j * S_range / length, X, F, T, r, sigma, R, 0.0625, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0);
    double result2 = crank_nicolson1(j * S_range / length, X, F, T, r, sigma, R, 0.125, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0);
    double result3 = crank_nicolson1(j * S_range / length, X, F, T, r, sigma, R, 0.1875, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0);
    outFile4 << j * S_range / length << " , " << result1 << "\n";
    outFile5 << j * S_range / length << " , " << result2 << "\n";
    outFile6 << j * S_range / length << " , " << result3 << "\n";
    outFile4.flush();
    outFile5.flush();
    outFile6.flush();
  }
  outFile4.close();
```

```cpp
  outFile5.close();
  outFile6.flush();

  // Produces tables of price,value,iterations,time and convergence rate for
  // comparing penalty and psor methods
  std::ofstream outFile7("./data/american_varying_smax_penalty.csv");
  double oldResult = 0, oldDiff = 0;
  double S = X;
  iMax = 100;
  jMax = 100;
  for (int n = 100; n <= 10000; n *= 2)
  {
    iMax = n;
    jMax = n;
    S_max = int(n / 30) * X;
    int sorCount{0};
    auto t1 = std::chrono::high_resolution_clock::now();
    double result = crank_nicolson1(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount, t0);
    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();
    double extrap = (4 * result - oldResult) / 3.;
    outFile7 << S_max << "," << iMax << "," << jMax << "," << S << "," << setprecision(10) << result << "," << time_taken << "," << extrap << "," << setprecision(3) << oldDiff / diff << "," << sorCount << "\n";
    oldDiff = diff;
    oldResult = result;
  }
  outFile7.close();

  std::ofstream outFile8("./data/american_varying_smax_sor.csv");
  oldResult = 0;
  oldDiff = 0;
  S = X;
  iMax = 100;
  jMax = 100;
  for (int n = 100; n <= 10000; n *= 2)
  {
    iMax = n;
    jMax = n;
    S_max = int(n / 30) * X;
    int sorCount{0};
    auto t1 = std::chrono::high_resolution_clock::now();
    double result = crank_nicolson2(S, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount, t0);
    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();
    double extrap = (4 * result - oldResult) / 3.;
    outFile8 << S_max << "," << iMax << "," << jMax << "," << S << "," << setprecision(10) << result << "," << time_taken << "," << extrap << "," << setprecision(3) << oldDiff / diff << "," << sorCount << "\n";
```

```
    oldDiff = diff;
    oldResult = result;
  }
  outFile8.close();

  // Produces final, most accurate value in <1s
  double S0 = X;
  iMax = 1300;
  jMax = 1300;
  S_max = 30 * X;
  auto t1 = std::chrono::high_resolution_clock::now();
  double result = crank_nicolson1(S0, X, F, T, r, sigma, R, kappa, mu, C,
    alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor, t0);
  auto t2 = std::chrono::high_resolution_clock::now();
  auto time_taken =
      std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
          .count();
  cout << fixed << result << "," << time_taken << endl;
}
```

Graphing Program Listing

```
import matplotlib.pyplot as plt
import numpy as np
import csv
from bisect import bisect_left,bisect_right
F=95.
C = 1.09
R=2.

allData=[]
with open('data/american_varying_s_beta_0_4.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.
  QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)

with open('data/no_put_american_varying_s_beta_0_4.csv', newline='\n') as
   csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.
  QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)

plt.figure()
plt.grid()
parity = [(R*s*C)/F * 100 for s in allData[1]['S'] if (R*s*C)/F <2 ]
x=[ s for s in allData[0]['S'] if s <100]
parity=x
plt.plot(parity,allData[0]['V'][:len(parity)],label=r'American Option
   Embedded Put',linewidth=2)
plt.plot(parity,allData[1]['V'][:len(parity)],label=r'Embedded Put Removed'
   ,linewidth=2,linestyle='--')
```

```
equity = [ max(95.,2.*s) for s in allData[1]['S']]
lessC=bisect_right(allData[0]['V'], 100)
plt.annotate(r'$V(S_0=%.2f,t=0)=P_p$'%(parity[lessC]), xy=(parity[lessC], allData[0]['V'][lessC]), xytext=(22, 160),arrowprops=dict(arrowstyle="->"))
lessRS=0
for i in allData[0]['V']:
  if(allData[1]['S'][lessRS]*R>=i):
    break
  lessRS+=1
plt.annotate(r'$V(S_0=%.2f,t=0)=RS_0$'%(parity[lessRS]), xy=(parity[lessRS],allData[0]['V'][lessRS]), xytext=(65, 100),arrowprops=dict(arrowstyle="->"))
plt.plot(allData[1]['S'][:len(parity)],equity[:len(parity)],label=r'Max(F,R$S_0$)',linewidth=1)
plt.xlabel(r'$S_0$')
plt.ylabel(r'$V(S,t=0)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/american_varying_s.png',bbox_inches='tight', pad_inches=0.2)

allData=[]
with open('data/american_varying_s_kappa_625.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)
with open('data/american_varying_s_kappa_125.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)
with open('data/american_varying_s_kappa_187.csv', newline='\n') as csvfile:
    reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.QUOTE_NONNUMERIC)
    currentData={'S':[], 'V':[]}
    for row in reader:
        currentData['S'].append(row['S'])
        currentData['V'].append(row['V'])
    allData.append(currentData)

plt.figure()
plt.grid()
start=120
end=160
plt.plot(allData[0]['S'][start:end],allData[0]['V'][start:end],label=r'$ \kappa = 0.0625$',linewidth=2)
plt.plot(allData[1]['S'][start:end],allData[1]['V'][start:end],label=r'$ \kappa = 0.125$',linewidth=2)
```

```
plt.plot(allData[2]['S'][start:end],allData[2]['V'][start:end],label=r'$ \kappa = 0.1875$',linewidth=2)
plt.xlabel(r'$S_0$')
plt.ylabel(r'$V(S,t=0)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/american_varying_s_varying_k.png',bbox_inches='tight',pad_inches=0.2)

plt.figure()
plt.grid()
start=100
end=200
plt.plot(allData[0]['S'][start:end],allData[0]['V'][start:end],label=r'$ \kappa = 0.0625$',linewidth=2)
plt.plot(allData[1]['S'][start:end],allData[1]['V'][start:end],label=r'$ \kappa = 0.125$',linewidth=2)
plt.plot(allData[2]['S'][start:end],allData[2]['V'][start:end],label=r'$ \kappa = 0.1875$',linewidth=2)
plt.xlabel(r'$S_0$')
plt.ylabel(r'$V(S,t=0)$')
plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
plt.savefig('images/complete_american_varying_s_varying_k.png',bbox_inches='tight', pad_inches=0.2)
```