Introduction

This report is a two part study of:

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

II) An extension to the above contract where the holder is able to exercise the decision to convert the bond in stock at **any time before** the maturity of the contract. This is known as an American embedded option

through the use of advanced numerical methods such as Crank-Nicolson with PSOR.

1   European 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 (See Appendix 1) 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 $j = 0 \to jmax$.

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 $\sigma = 3.03$. 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 paramaters as previously defined

2) ($\beta = 0.486$, $\sigma = 3.03$) with all other paramaters 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.
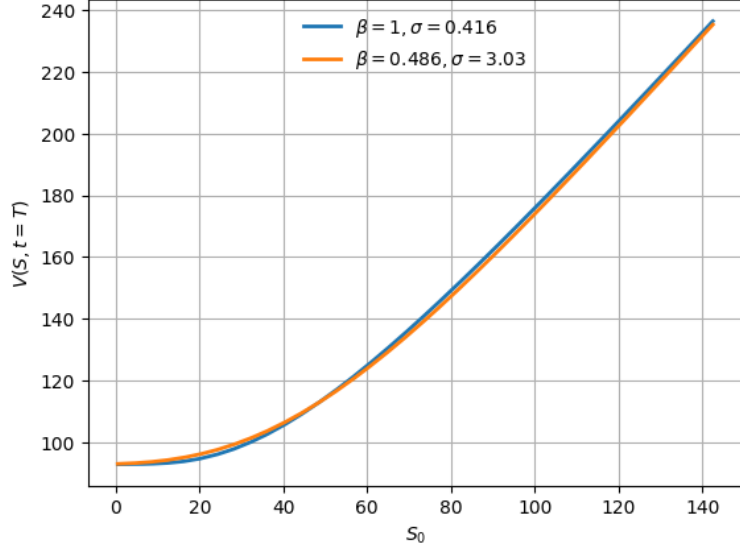
Figure 1: Value of the convertible bond $V(S, t = T)$ against inital underlying asset price at time S_0 for two combinations of β and σ.

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 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
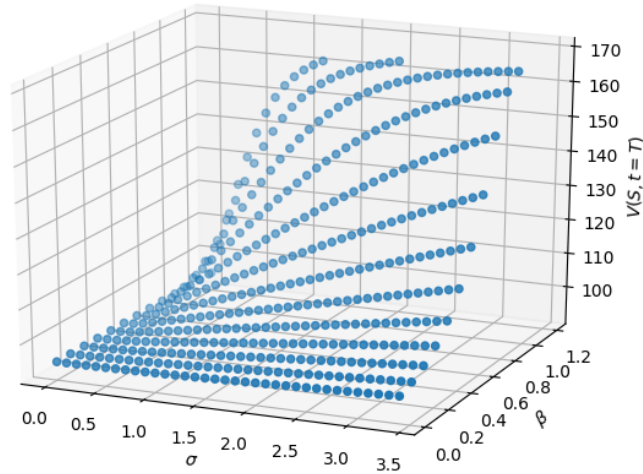

Figure 2: Value of the convertible bond $V(S = X, t = T)$ against parameters β and σ.

β which should be used are for ≤ 1. Above this, there are implications on the inaccessibility of the origin

3

which for a stock price means it cannot go bankrupt which is not true. Thus, we shall stick for values of $\beta \leq 1$ here. In this regime, the model captures the so-called 'leverage effect' which practically means that stock price and volatily 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. So, there are $\sigma - \beta$ planes on Figure 2 which have close values of the convertible bond for multiple combinations of (σ , β). This happens since having a steep implied variance skew but a lower actual variation, which is an average of a time period, and the other way round counteract each other.

1.2 Varying step sizes

As will be described later though, an increase in i_{max} is preferrable rather than j_{max} for convergence due to computational time requirements. Finally, for this section the parameters i_{max}, j_{max} and S_{max}
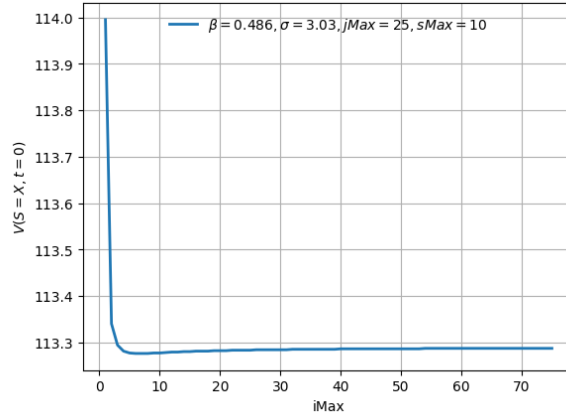

Figure 3: Trend of the convertible bond $V(S = X, t = T)$ as parameter i_{max} is varied.

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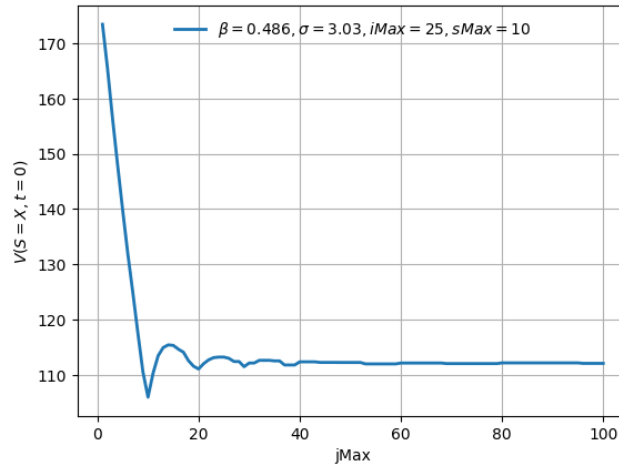
(a) Stability and convergence can be obverved after $j_{max} = 40$

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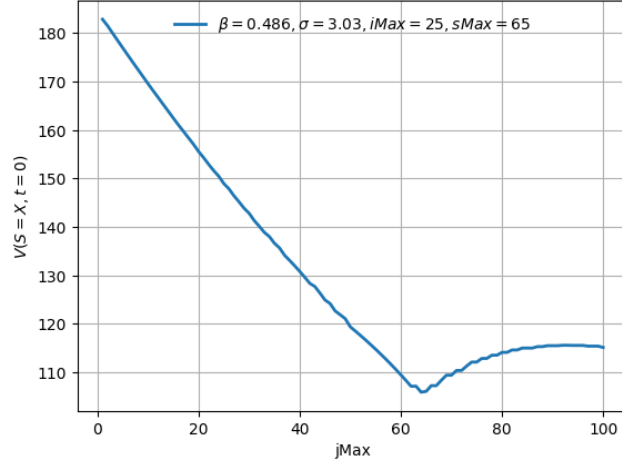
(b) The plot from **??** is stretched and since S_{max} is x6.5 as much, the first minimum is also stretched by that much.

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

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 $\mathcal{O}(\Delta S^2, \Delta t^2)$.

Following this, the natural progression is to vary S_{max} for a given value of i_{max}. However, due to the results in Figure 4 we have to make sure we increase j_{max} as we go along. Figure 6 clearly shows
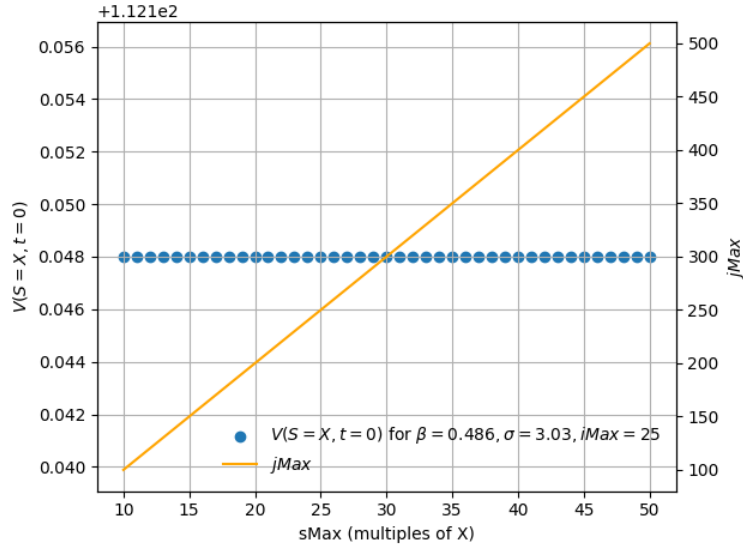

Figure 5: Trend of the convertible bond $V(S = X, t = 0)$ as parameter S_{max} is varied and j_{max} is varied to be kept at a comparable value.

that already at $S_{max} = 10X$ the value of the convertible bond converges. In fact, for higher S_{max} the result is identical to 7 significant figures since this is the residual error limit that was set on the SOR method, with a maximum cap of 10,000 iterations which all iterations of S_{max} shown stay under. The computational requirement of increasing S_{max} is linked to that of increasing j_{max} and since the re-

5

turn is an unreasonable amount of significant figures precision then it is not worth using an $S_{max} > 10X$.

The last issue left to investigate is the time requirements and processing complexity of varying these parameters. As can be infered from Figure 6 i_max follows a linear time increase while j_max is exponential. 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} however j_{max} has the highest influence on the value converging to the analytic value so a compromise must be made.
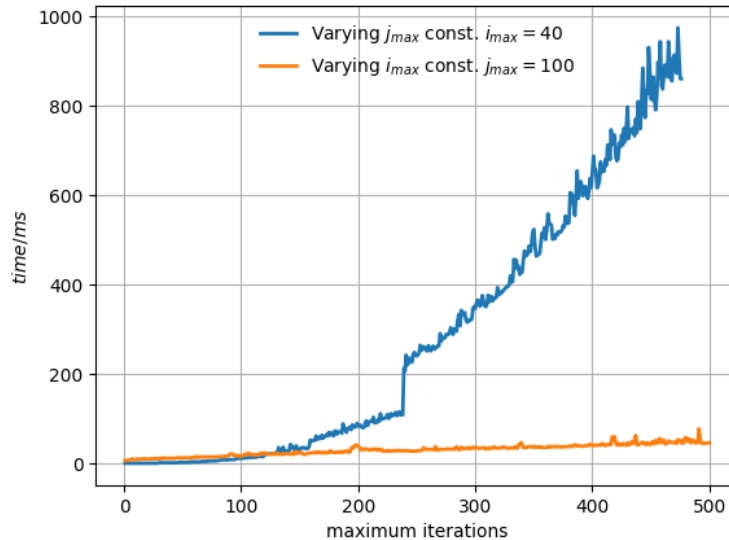

Figure 6: Variation of the time required to calculate the convertible bond price as j_{max} and i_{max} are varied.

The final value for $\sigma = 3.03$ and $\beta = 0.486$ was calculated to be $V(S = X, t = 0) = 112.163$ with $S_{max} = 10X$, $j_{max} = 400$, $i_{max} = 50$.

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.

Appendix 2

Portfolio Pricing Program Listing

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

using namespace std;

/*
 * ON INPUT:
 * a, b and c -- are the diagonals of the matrix
 * rhs        -- is the right hand side
 * x          -- is the initial guess
 * iterMax    -- is maximum iterations
 * tol        -- is the tolerance level
 * omega      -- is the relaxation parameter
 * sor        -- not used
 * ON OUTPUT:
 * a, b, c, rhs        -- unchanged
 * x                   -- solution to Ax=b
 * iterMax, tol, omega -- unchanged
 * sor                 -- number of iterations to converge
 */
void sorSolve(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 &sorCount)
{
  // assumes vectors a,b,c,d,rhs and x are same size (doesn't check)
  int n = a.size() - 1;
  // sor loop
  for (sorCount = 0; sorCount < iterMax; sorCount++)
  {
    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]);
```

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

```cpp
      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
      sorSolve(a, b, c, d, vNew, iterMax, tol, omega, sorCount);
      //vNew = thomasSolve(a, b, c, d);

      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;

    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-7, omega = 1., S_max = 10 * X;
    //
    /*
    double T = 3., F = 56., R = 1., r = 0.0038, kappa = 0.083333333, altSigma
```

```
              = 0.369 ,
153                   mu = 0.0073 , X = 56.47 , C = 0.106 , alpha = 0.01 , beta = 1. , sigma
              = 3.73 , S_max = 10 * X, tol = 1.e−7, omega = 1.;
154     */
155     int iterMax = 10000 , iMax = 100 , jMax = 100;
156     // Create graph of varying S0 and beta and bond
157     int length = 300;
158     double S_range = 3 * X;
159     int sor ;
160     /*
161     std :: ofstream outFile7 (". / data / varying_s_beta . csv" ) ;
162
163     for (double beta = 0; beta < 1.3; beta += 0.1)
164     {
165       for (int j = 1; j <= length − 1; j++)
166       {
167         outFile7 << beta << " , " << altSigma << " , " << j * S_range /
              length << " , " << crank_nicolson ( j * S_range / length , X, F, T, r,
              altSigma , R, kappa , mu, C, alpha , beta , iMax , jMax , S_max , tol , omega ,
              iterMax , sor ) << "\n";
168       }
169     }
170     outFile7 . close () ;
171     */
172     /*
173     std :: ofstream outFile8 (". / data / varying_s_sigma . csv" ) ;
174     beta = 1;
175     for (double altSigma = 0; altSigma < 3.5; altSigma += 0.1)
176     {
177       for (int j = 1; j <= length − 1; j++)
178       {
179         outFile8 << beta << " , " << altSigma << " , " << j * S_range /
              length << " , " << crank_nicolson ( j * S_range / length , X, F, T, r,
              altSigma , R, kappa , mu, C, alpha , beta , iMax , jMax , S_max , tol , omega ,
              iterMax , sor ) << "\n";
180       }
181     }
182     outFile8 . close () ;
183     */
184     /*
185     std :: ofstream outFile9 (". / data / varying_s_sigma_beta . csv" ) ;
186     for (double altSigma = 0; altSigma < 3.5; altSigma += 0.1)
187     {
188       for (double beta = 0; beta < 1.3; beta += 0.1)
189       {
190         double S0 = X;
191         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";
192       }
193     }
194     outFile9 . close () ;
195     */
196     /*
197
198     std :: ofstream outFile1 (". / data / varying_s_beta_1 . csv" ) ;
199     std :: ofstream outFile2 (". / data / varying_s_beta_0_4 . csv" ) ;
200     for (int j = 1; j <= length − 1; j++)
```

```cpp
201     {
202       vector<double> gamma(jMax + 1);
203       outFile1 << j * S_range / length << " , " << crank_nicolson(j * S_range
              / length, X, F, T, r, altSigma, R, kappa, mu, C, alpha, 1, iMax, jMax,
          S_max, tol, omega, iterMax, sor, gamma) << "\n";
204       outFile2 << j * S_range / length << " , " << crank_nicolson(j * S_range
              / length, X, F, T, r, sigma, R, kappa, mu, C, alpha, beta, iMax, jMax,
          S_max, tol, omega, iterMax, sor, gamma) << "\n";
205     }
206     outFile1.close();
207     outFile2.close();
208     */
209     /*
210     std::ofstream outFile3("./data/varying_imax.csv");
211     jMax = 100;
212     for (iMax = 1; iMax <= 500; iMax += 1)
213     {
214       double S = X;
215       auto t1 = std::chrono::high_resolution_clock::now();
216       double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
          alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
217       auto t2 = std::chrono::high_resolution_clock::now();
218       auto time_taken =
219           std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
220               .count();
221       outFile3 << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
          std::fixed << result << "," << time_taken << "\n";
222     }
223     outFile3.close();
224     */
225     /*
226     for (int s_Mult = 10; s_Mult <= 10; s_Mult += 1)
227     {
228       double S = X;
229       S_max = s_Mult * X;
230       string title = "./data/smax_jmax/" + to_string(s_Mult) + "_varying_jmax
          .csv";
231       std::ofstream outFile4(title);
232       iMax = 40;
233       for (jMax = 1; jMax <= 500; jMax += 1)
234       {
235         auto t1 = std::chrono::high_resolution_clock::now();
236         double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
          alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
237         auto t2 = std::chrono::high_resolution_clock::now();
238         auto time_taken =
239             std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
240                 .count();
241         outFile4 << S_max << "," << iMax << "," << jMax << "," << S << " , "
          << std::fixed << result << "," << time_taken << "\n";
242       }
243       outFile4.close();
244     }
245     */
246     /*
247     std::ofstream outFile5("./data/varying_smax.csv");
248     iMax = 25;
249     tol = 1.e-7;
```

```
250    for (int s_Mult = 10; s_Mult <= 50; s_Mult += 1)
251    {
252      jMax = s_Mult * 10;
253      double S = X;
254      S_max = s_Mult * X;
255      int sorCount;
256      auto t1 = std::chrono::high_resolution_clock::now();
257      double result = crank_nicolson(S, X, F, T, r, sigma, R, kappa, mu, C,
         alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sorCount);
258      auto t2 = std::chrono::high_resolution_clock::now();
259      auto time_taken =
260          std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1)
261             .count();
262      outFile5 << S_max << "," << iMax << "," << jMax << "," << S << " , " <<
         std::fixed << result << "," << time_taken << "\n";
263    }
264    outFile5.close();
265    */
266    /*
267    S_max = 10 * X;
268    std::ofstream outFile6("./data/analytic.csv");
269    iMax = 25, jMax = 100;
270
271    for (int j = 1; j <= length - 1; j++)
272    {
273      vector<double> gamma(jMax + 1);
274
275      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, gamma) << "\n";
276    }
277    outFile6.close();
278    */
279    S_max = 10 * X;
280    iMax = 50, jMax = 400;
281    double S0 = X;
282    std::cout << std::fixed << crank_nicolson(S0, X, F, T, r, sigma, R, kappa
         , mu, C, alpha, beta, iMax, jMax, S_max, tol, omega, iterMax, sor);
283 }
```

Graphing Program Listing

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

```python
r=0.0038
R=1
F=56
sigma = 0.369
K=56.47
'''
#Calulate value of coupon
#Through integrating Cexp(-(alpha+r)t)dt from 0 to T
COUPON = C/(alpha+r) * (1- np.exp((-(alpha+r)*T)))

BOND = F*np.exp(-r*T)

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

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

    return call

ANALYTIC_PRICE = []
STOCK_PRICE = []
BOND_FLOOR = []
CONV_BOND = []
for s in range(1,140):
    STOCK_PRICE.append(s)
    ANALYTIC_PRICE.append(R*euro_vanilla_call(s, K, T, r, sigma) + BOND +
    COUPON)

#plt.plot(S,V1, label = " beta = 1")
#plt.plot(STOCK_PRICE,BOND_FLOOR, label = "Bond")
plt.plot(STOCK_PRICE,ANALYTIC_PRICE, label = "Analytic")
plt.plot(variationData[0]['x'],variationData[0]['y'], label = "Crank")
plt.xlabel('Stock price')
plt.ylabel('Eurocall Option')
plt.legend()
plt.savefig('images/analytic.png',bbox_inches='tight', pad_inches=0.2)
```

```python
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import csv
4  import scipy.stats as si
5
6  X=47.66
7  R=2
8  F=95
9  T=2.0
10 C = 1.09
11 alpha = 0.02
12 r = 0.0229
13 T = 2.
14 sigma = 0.416
15 '''
16 def euro_vanilla_call(S, K, T, r, sigma):
17
18     #S: spot price
19     #K: strike price
20     #T: time to maturity
21     #r: interest rate
22     #sigma: volatility of underlying asset
23
24     d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
       )
25     d2 = (np.log(S / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T)
       )
26
27     call = (R*S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si.norm.
       cdf(d2, 0.0, 1.0))
28
29     return call
30
31 variationData=[]
32 with open('data/varying_imax.csv', newline='\n') as csvfile:
33     reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V
       '],quoting=csv.QUOTE_NONNUMERIC)
34     currentData={'x':[],'y':[],'jMax':0,'sMax':0}
35     for row in reader:
36         currentData['x'].append(row['iMax'])
37         currentData['y'].append(row['V'])
38         currentData['jMax']=row['jMax']
39         currentData['sMax']=int(row['sMax']/X)
40     variationData.append(currentData)
41
42 plt.figure()
43 plt.grid()
44 plt.plot(variationData[0]['x'],variationData[0]['y'],label=r'$\beta=0.486,\
       sigma=3.03,jMax=%i,sMax=%i$'%(variationData[0]['jMax'],variationData
       [0]['sMax']),linewidth=2)
45 plt.xlabel('iMax')
46 plt.ylabel(r'$V(S=X,t=0)$')
47 plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
48 plt.savefig('images/european_varying_imax.png',bbox_inches='tight',
       pad_inches=0.2)
49
50 with open('data/varying_smax.csv', newline='\n') as csvfile:
51     reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V
```

```python
    '] , 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 'sMax ( multiples of X) ')
ax1 . set_ylabel ( r '$V(S=X, t=0)$ ')
ax1 . grid ()
ax1 . scatter ( np . asarray ( variationData [ 1 ] [ 'x ' ] [ : 2 0 ] ) , variationData [ 1 ] [ 'y
    '] [ : 2 0 ] , label=r '$V(S=X, t=0)$ for $\beta =0.486 ,\ sigma =3.03 ,iMax=%i$ '%(
    variationData [ 1 ] [ 'iMax '] ) )
ax2 = ax1 . twinx ()
ax2 . set_ylabel ( r '$jMax$ ')
fig . tight_layout ()
ax2 . plot ( np . asarray ( variationData [ 1 ] [ 'x ' ] [ : 2 0 ] ) , variationData [ 1 ] [ 'jMax
    '] [ : 2 0 ] , label=r '$jMax$ ' , 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 'sMax ( multiples of X) ')
ax1 . set_ylabel ( r '$V(S=X, t=0)$ ')
ax1 . grid ()
ax1 . scatter ( np . asarray ( variationData [ 1 ] [ 'x ' ] ) , variationData [ 1 ] [ 'y ' ] , label=r
    '$V(S=X, t=0)$ for $\beta =0.486 ,\ sigma =3.03 ,iMax=%i$ '%( variationData [ 1 ] [ '
    iMax '] ) )
ax2 = ax1 . twinx ()
ax2 . set_ylabel ( r '$jMax$ ')
fig . tight_layout ()
ax2 . plot ( np . asarray ( variationData [ 1 ] [ 'x ' ] ) , variationData [ 1 ] [ 'jMax ' ] , label=r
    '$jMax$ ' , 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 range (10 ,101) :
    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 '] )
```

```python
97                  currentData['iMax']=row['iMax']
98                  currentData['sMax']=int(row['sMax']/X)
99
100     plt.figure()
101     plt.plot(currentData['x'],currentData['y'],label=r'$\beta=0.486,\sigma
        =3.03,iMax=%i,sMax=%i$'%(currentData['iMax'],currentData['sMax']),
        linewidth=2)
102     plt.xlabel('jMax')
103     plt.ylabel(r'$V(S=X,t=0)$')
104     plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
105     plt.grid()
106     plt.savefig('images/smax_jmax/'+str(smax)+'_european_varying_jmax.png',
        bbox_inches='tight', pad_inches=0.2)
107     plt.close()
108 '''
109 '''
110 allData=[]
111 with open('data/varying_s_beta_1.csv', newline='\n') as csvfile:
112     reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.
        QUOTE_NONNUMERIC)
113     currentData={'S':[],'V':[]}
114     for row in reader:
115         currentData['S'].append(row['S'])
116         currentData['V'].append(row['V'])
117     allData.append(currentData)
118
119 with open('data/varying_s_beta_0_4.csv', newline='\n') as csvfile:
120     reader = csv.DictReader(csvfile,fieldnames=['S','V'],quoting=csv.
        QUOTE_NONNUMERIC)
121     currentData={'S':[],'V':[]}
122     for row in reader:
123         currentData['S'].append(row['S'])
124         currentData['V'].append(row['V'])
125     allData.append(currentData)
126
127 plt.figure()
128 plt.grid()
129 plt.plot(allData[0]['S'],allData[0]['V'],label=r'$\beta=1,\sigma=0.416$',
        linewidth=2)
130 plt.plot(allData[1]['S'],allData[1]['V'],label=r'$\beta=0.486,\sigma=3.03$
        ',linewidth=2)
131 #plt.plot(allData[0]['S'],np.ones(len(allData[0]['S'])) * F,label=r'
        Principal Only',linewidth=2)
132 equityOnly_1 = [ R*euro_vanilla_call(s, X, T, r, sigma) for s in allData
        [0]['S'] ]
133 equityOnly_2 = [ R*euro_vanilla_call(s, X, T, r, 3.03) for s in allData
        [0]['S'] ]
134 #plt.plot(allData[0]['S'],equityOnly_1,label=r'Equity Only $\sigma=0.416$',
        linewidth=2)
135 #plt.plot(allData[0]['S'],equityOnly_2,label=r'Equity Only $\sigma=3.03$',
        linewidth=2)
136
137 plt.xlabel(r'$S_0$')
138 plt.ylabel(r'$V(S,t=T)$')
139 plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
140 plt.savefig('images/european_varying_s.png',bbox_inches='tight', pad_inches
        =0.2)
141
```

```python
142  import matplotlib.pyplot as plt
143  from mpl_toolkits.mplot3d import Axes3D
144
145  currentData={'S':[],'V':[],'beta':[]}
146  with open('data/varying_s_beta.csv', newline='\n') as csvfile:
147      reader = csv.DictReader(csvfile,fieldnames=['beta','altsigma','S','V'],
         quoting=csv.QUOTE_NONNUMERIC)
148      currentData={'S':[],'V':[],'beta':[]}
149      for row in reader:
150          currentData['S'].append(row['S'])
151          currentData['V'].append(row['V'])
152          currentData['beta'].append(row['beta'])
153
154  fig = plt.figure()
155  ax = fig.add_subplot(111, projection='3d')
156  ax.scatter(currentData['S'],currentData['V'],currentData['beta'])
157  plt.savefig('images/european_varying_s_varying_beta.png',bbox_inches='tight
         ', pad_inches=0.2)
158  '''
159  '''
160  currentData={'S':[],'V':[],'beta':[],'sigma':[]}
161  with open('data/varying_s_sigma_beta.csv', newline='\n') as csvfile:
162      reader = csv.DictReader(csvfile,fieldnames=['beta','sigma','S','V'],
         quoting=csv.QUOTE_NONNUMERIC)
163      for row in reader:
164          if(row['V']==-1):
165              continue
166          currentData['S'].append(row['S'])
167          currentData['V'].append(row['V'])
168          currentData['beta'].append(row['beta'])
169          currentData['sigma'].append(row['sigma'])
170  '''
171  '''
172  fig = plt.figure()
173  ax = fig.add_subplot(111, projection='3d')
174  ax.scatter(currentData['sigma'],currentData['beta'],currentData['V'])
175  ax.set_xlabel(r'$\sigma$')
176  ax.set_ylabel(r'$\beta$')
177  ax.set_zlabel(r'$V(S,t=T)$')
178  plt.show()
179  plt.savefig('images/european_varying_s_varying_sigma_varying_beta.png',
         bbox_inches='tight', pad_inches=0.2)
180  '''
181  '''
182  fig = plt.figure()
183  ax2 = fig.add_subplot(111)
184  ax2.hist2d(currentData['sigma'],currentData['beta'],bins=100,weights=
         currentData['V'])
185  ax2.set_xlabel(r'$\sigma$')
186  ax2.set_ylabel(r'$\beta$')
187  #ax2.set_zlabel(r'$V(S,t=T)$')
188  plt.show()
189  plt.savefig('images/hist2d_european_varying_s_varying_sigma_varying_beta.
         png',bbox_inches='tight', pad_inches=0.2)
190  '''
191  plt.figure()
192  plt.grid()
193  plt.xlabel(r'maximum iterations')
```

```python
194 plt.ylabel(r'$time/ms$')
195
196 currentData={'x':[],'y':[],'iMax':0,'sMax':0,'time':[]}
197 with open('data/smax_jmax/10_varying_jmax.csv', newline='\n') as csvfile:
198     reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V
        ','time'],quoting=csv.QUOTE_NONNUMERIC)
199     for row in reader:
200         currentData['x'].append(row['jMax'])
201         currentData['y'].append(row['V'])
202         currentData['iMax']=row['iMax']
203         currentData['sMax']=int(row['sMax']/X)
204         currentData['time'].append(row['time'])
205
206
207 plt.plot(currentData['x'],currentData['time'],label=r'Varying $j_{max}$
        const. $i_{max}=40$',linewidth=2)
208
209
210
211 variationData=[]
212 currentData={'x':[],'y':[],'jMax':0,'sMax':0,'time':[]}
213
214 with open('data/varying_imax.csv', newline='\n') as csvfile:
215     reader = csv.DictReader(csvfile,fieldnames=['sMax','iMax','jMax','S','V
        ','time'],quoting=csv.QUOTE_NONNUMERIC)
216     for row in reader:
217         currentData['x'].append(row['iMax'])
218         currentData['y'].append(row['V'])
219         currentData['jMax']=row['jMax']
220         currentData['sMax']=int(row['sMax']/X)
221         currentData['time'].append(row['time'])
222
223 plt.plot(currentData['x'],currentData['time'],label=r'Varying $i_{max}$
        const. $j_{max}=100$',linewidth=2)
224
225 plt.legend(loc='upper center',fancybox=False, framealpha=0.0)
226 plt.savefig('images/european_time.png',bbox_inches='tight', pad_inches=0.2)
```