Assignment 2

ID:10423513

1 Introduction

In this project, we would like to explore using numerical methods to solve Black-Scholes PDE for some convertible bond contracts. We will mainly apply finite difference with Crank-Nicholson scheme, as well as some improvements such as policy iteration and penalty method. First we start from European style contract and proceed to American style one with put option embedded. Since there is impossible to derive analytic solutions for both situations, we are then tasked to predict contract values and analyse efficiency. Ideally we can produce an efficient value at the end.

2 European Style Options

2.1 Boundary Conditions

According to the background theory of convertible bonds, for large S we can derive the boundary condition by solving the problem PDE:

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

We can assume that the solution is of the linear form of two functions $A(t)$ and $B(t)$:

$$V(S,t) = SA(t) + B(t) \tag{2}$$

Now substitute (2) into (1) we obtain:

$$SA(t)' + B(t)' + \kappa(X - S)A(t) - r(SA(t) + B(t)) - Ce^{-\alpha t} = 0 \tag{3}$$

where derivative of $A(t)$ and $B(t)$ with respect to t. We can then rearrange (3) to the following:

$$S\big[A(t)' - \kappa A(t) - rA(t)\big] + B(t)' - rB(t) + \kappa X A(t) + Ce^{-\alpha t} = 0$$

Since this equation holds for any large enough S, we can have that those terms with coefficient S should equal to zero, which follows as:

$$A(t)' - \kappa A(t) - rA(t) = 0 \tag{4}$$

$$B(t)' - rB(t) = -(\kappa X A(t) + Ce^{\alpha t}) \tag{5}$$

Like above we can now solve two ODEs to obtain $A(t)$ and $B(t)$, with final condition that

$$V(S,T) = RS \quad as\ S \to \infty$$

From this final condition we have:

$$A(T) = R; \qquad B(T) = 0$$

Thus we can easily solve (4) and (5) by using methods for ODE, such as integrating factor for (5).
The boundary condition for large S is:

$$A(t) = Re^{-(\kappa+r)(T-t)} \tag{6}$$

$$B(t) = XRe^{-r(T-t)}(1 - e^{-\kappa(T-t)}) + \frac{C}{\alpha + r}(e^{-\alpha t} - e^{-(\alpha+r)T}e^{rt}) \tag{7}$$

$$V(S,t) = SA(t) + B(t) \quad as\ S \to \infty \tag{8}$$

$$V(S,T) = RS \tag{9}$$

2.2 Crank-Nicolson Method

In the previous section we have obtained the boundary condition at large S, where we say the grid with the largest value $j = jmax$. Now we also need to find out the condition at $j = 0$ where $S = 0$. We know the Black-Scholes PDE that $V(S,t)$ satisfies:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 V}{\partial S^2} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0. \tag{10}$$

Simply we let $S = 0$ then

$$\frac{\partial V}{\partial t} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0. \quad when\ S = 0 \tag{11}$$

Since it is impossible to get V_{-1}, here we can only evaluate the derivative around the point $V(S,t)$ and we cannot use central differencing. In this case the error is $O((\Delta S)^2, \Delta t)$, the same as explicit finite difference. Then we have

$$\frac{\partial V}{\partial t} \approx \frac{V_0^{i+1} - V_0^i}{\Delta t}$$

$$\frac{\partial V}{\partial S} \approx \frac{V_1^i - V_0^i}{\Delta S}$$

Apply these two approximations to (11) and arrange the known on one side and unknown on another side, we obtain corresponding $a_0^i, b_0^i, c_0^i, d_0^i$(I will write them explicitly together with those at other positions)
Now the only left thing is to approximate interior points, where we can use central differencing and evaluate at $V(S, t + \Delta t/2)$ to remove the stability constraints and improve convergence to $(\Delta t)^2$ w.r.t time. Similar to what we put into use on lecture notes, after evaluating derivatives at $V(S, t + \Delta t/2)$, we have following approximations

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

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

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

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

Now put all above approximations onto PDE (10), rearranging we can obtain our a_j, b_j, c_j, d_j which satisfy the matrix:

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & \dots & \dots & \dots & \dots & 0 \\ a_1 & b_1 & c_1 & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & a_2 & b_2 & c_2 & 0 & \dots & \dots & \dots & \dots \\ \dots & \dots & a_3 & b_3 & c_3 & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & a_j & b_j & c_j & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & a_{jmax} & b_{jmax} \end{pmatrix} \begin{pmatrix} V_0^i \\ V_1^i \\ V_2^i \\ V_3^i \\ \vdots \\ \vdots \\ V_{jmax-1}^i \\ V_{jmax}^i \end{pmatrix} = \begin{pmatrix} d_0^i \\ d_1^i \\ d_2^i \\ d_3^i \\ \vdots \\ \vdots \\ d_{jmax-1}^i \\ d_{jmax}^i \end{pmatrix}$$

where for $1 \leq j \leq jmax - 1$

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

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

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

$$d_j = -a_j V_{j-1}^{i+1} - (-\tfrac{1}{2}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} + \tfrac{1}{\Delta t} - \tfrac{r}{2})V_j^{i+1} - c_j V_{j+1}^{i+1} - Ce^{-\alpha t}$$

For interior points we are evaluating at $t = (i + \frac{1}{2})\Delta t$. Above I do not write a_j^i for simplicity but it involves time variable.

For $j = 0$, by using boundary conditions we derived in the beginning of this section we have

$$a_0^i = 0$$

$$b_0^i = -\tfrac{1}{\Delta t} - \kappa\theta(t)\tfrac{1}{\Delta S} - r$$

$$c_0^i = \tfrac{\kappa\theta(t)}{\Delta S}$$

$$d_0^i = -\tfrac{1}{t}V_0^{i+1} - Ce^{-\alpha t}$$

Here time $t = i\Delta t$ since we take Taylor expansions at point $V(S, t)$.

For $j = jmax$, by the result of the last section, we have

$$a_{jmax}^i = 0$$

$$b_{jmax}^i = 1$$

$$c_{jmax}^i = 0$$

$$d_{jmax}^i = S_{max}A(t) + B(t)$$

where $A(t)$ and $B(t)$ are (6) and (7) respectively. Also note that here time $t = i\Delta t$ since we are deriving an analytic solution in this case.

2.3 How β and σ affect

Based on calculation of option prices, we are always interesting on how variant is it. Since this BS PDE is not original as normal European options, where the variance elasticity is involved. Now we proceed to plot some figures to investigate how this term (β, σ) influence. To better catch difference between figures, in all cases I choose $imax == jmax = 200$ and $S_{max} = 140$. As we can observe at

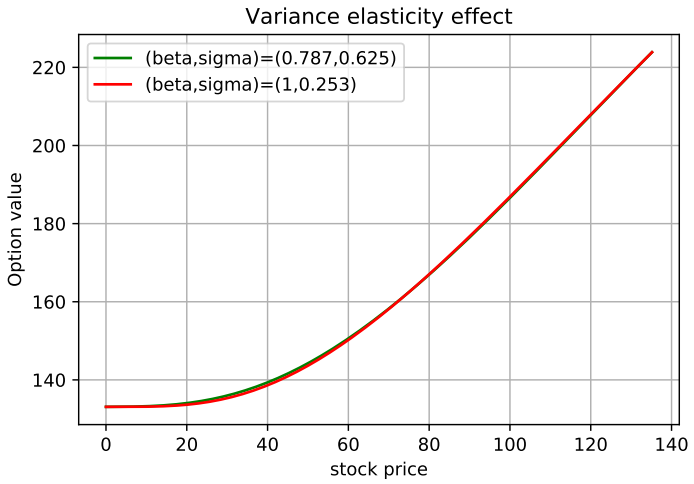


Figure 1: Graph of 2 cases

the beginning two figures can be distinguished slightly, but as S enlarges, two lines almost coincide and overlap each other. Now let us look at another graph and then explain them altogether.

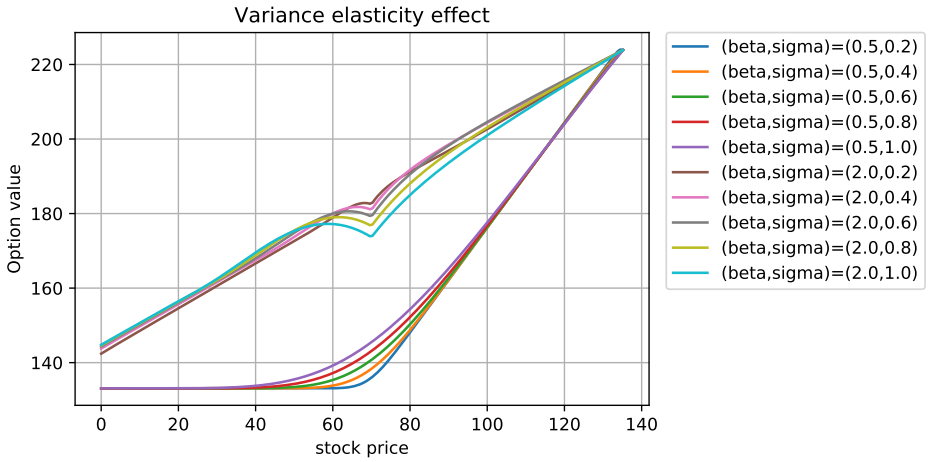


Figure 2: Graph of 10 cases

According to Figure(2), if at a fixed σ, option value of bigger elasticity β is always bigger, as we can see all 5 lines with $\beta = 2.0$ are above the others. And the most variant position is at around $S = 67.58$, which is related to X in the formula.

Reason:The model with variance elasticity aims to capture leverage effect. When $0 \leq \beta < 1$, the leverage effect is invoked, this is usually observed where volatility declines while stock price increases.

So we can explain why two plots overlap in Figure(1), since green line has bigger σ but later the volatility falls, to the same level with $(\beta = 1, \sigma = 0.253)$. In Figure(2), some plots have $\beta = 2.0$, this case is commonly called inverse-leverage effect. In this situation volatility tends to go up as stock price jumps. That's why as S increase those 5 lines start to vary.

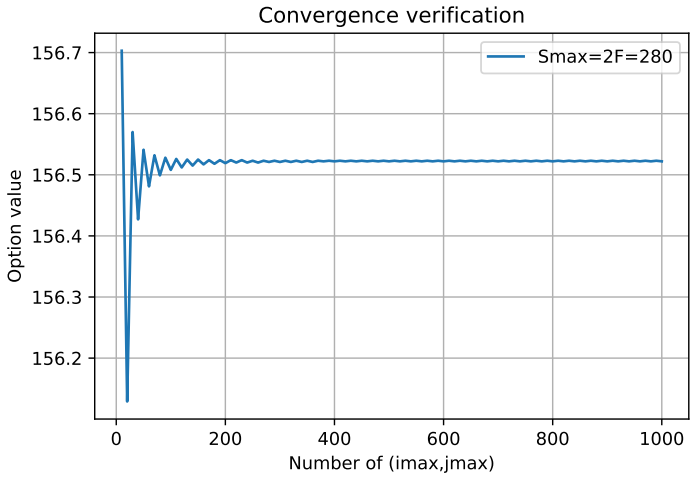
2.4 Efficiency Analysis

With a fixed set of other parameters, we now proceed to investigate the convergence and efficiency. Here I will apply Quadratic(higher order) interpolation to achieve better accuracy. And I will modify $(imax, jmax, S_{max})$ combination to seek their impact. In order to compare the accuracy, ideally we need an analytic solution but in this case it is not realistic to get one. Since more computation effort usually means higher accuracy, here I will use a combination $(imax, jmax, S_{max})$ with large values, and then compute by **Higher order interpolation** method, it can take even minutes to get an answer and **I will treat it as an approximated analytic solution**

The process of producing this result is very straightforward, I set $imax = jmax = 20000$ such that them can be almost continuous, and $S_{max} = 280$. After obtaining the V_j^0 vector, then use quadratic interpolation (described as follow) to evaluate the option value at $S_0 = 67.58$. Quadratic interpolation:

$$V(S_0) = \frac{(S_0 - S_i)(S_0 - S_{i+1})}{(S_{i-1} - S_i)(S_{i-1} - S_{i+1})} V_{i-1} + \frac{(S_0 - S_{i-1})(S_0 - S_{i+1})}{(S_i - S_{i-1})(S_i - S_{i+1})} V_i + \frac{(S_0 - S_{i-1})(S_0 - S_i)}{(S_{i+1} - S_{i-1})(S_{i+1} - S_i)} V_{i+1}$$

The graph shows clear convergence of option value. Thus it is proper to set up an approximated



analytic value.

Also we would like to know does it converge at a rate as expected, say error is $O((\Delta S)^2, (\Delta t)^2)$, it is very difficult to spot this since different values of $imax, jmax, S_{max}$ would reach the converged value at different maximum steps. For example we decrease Δt by 10 times but it already converges to 156.523 at around $\Delta t/5$ or less. So we can only roughly infer that the convergence rate is within the feasible range. Here I show how error changes with increasing $imax$ with fixed $(jmax = 1000, S_{max} = 280)$.

imax	$\varepsilon = \lvert V - 156.523 \rvert$
10	$\varepsilon = 0.172$
20	$\varepsilon = 0.043$
40	$\varepsilon = 0.006$

Base on above, we can assume our setup $(20000, 20000, 280)$ an approximated solution, with **The benchmark value**: $V(S_0 = 67.58) = 156.523$

Now we use this benchmark to compare with other combinations of $(imax, jmax, S_{max})$ to find out the most efficient group. In principle, $imax$ and $jmax$ concerns property of continuity in time and $jmax$ are both continuous. And S_{max} decides how close we can evaluate analytically when $S \to \infty$. $jmax$ itself is not meaningful, instead $dS = \frac{S_{max}}{jmax}$ can determine the magnitude of error. Since it is related to S_{max}, if we set it very high then we also need very large $jmax$ to match. Based on principle analysis, I plot out three graphs to show how each of the combination affect.

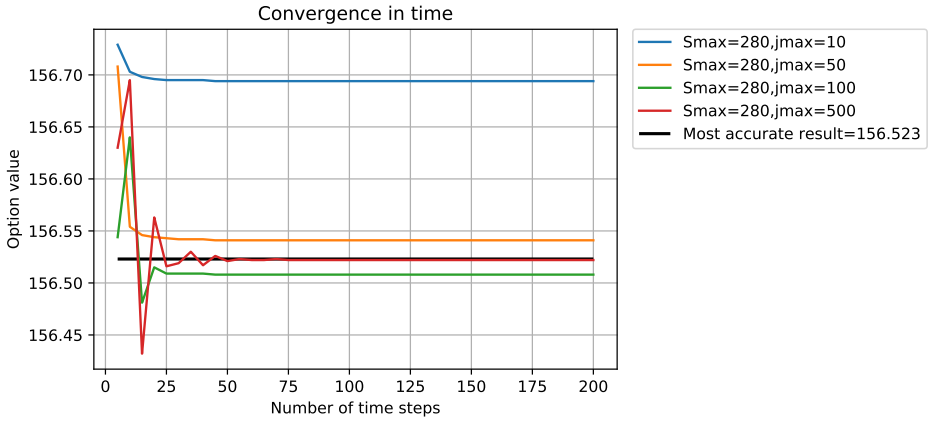


Figure 3: How $imax$ affects

Figure(3) is based on fixed $S_{max} = 280$. We can see 4 lines converge differently. For some small $jmax$ option values will converge to a wrong result. Notice that all 4 plots are converged after 50, **we can therefore treat** $imax = 100$ **a guaranteed level.**

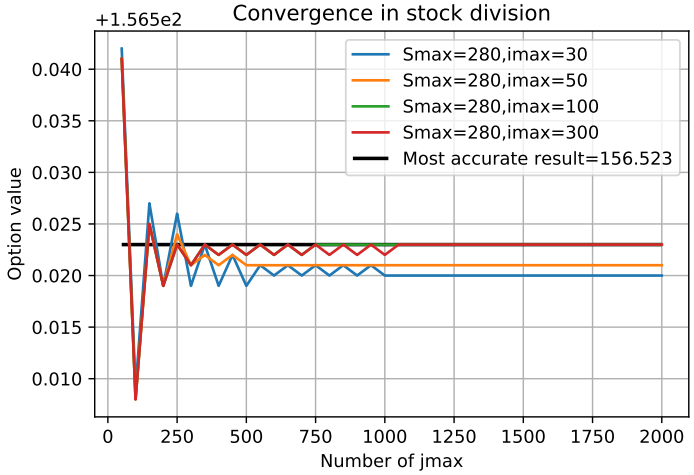
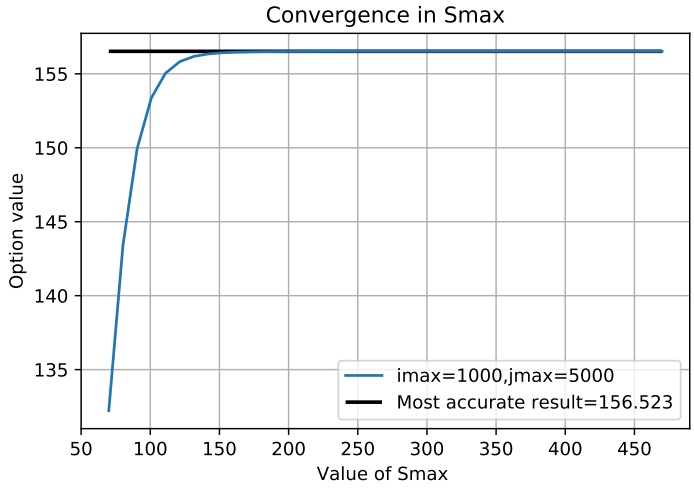


Figure 4: How $jmax$ affects

Similarly, we keep $S_{max} = 280$ for Figure(4) and see how $(imax, jmax)$ combination impacts. The behaviour of 4 cases are expected, all of them need $jmax > 1000$ to converge, even with a rather big value of $imax$. This is because dS influences more, it is involved with discontinuity error. **Accordingly, we can then treat** $dS = 0.10 < \frac{280}{2000} = 0.14$ **an acceptable level.**
Now proceed to Figure(5), where large enough $(imax, jmax)$ are settled. Since S_{max} directly matters



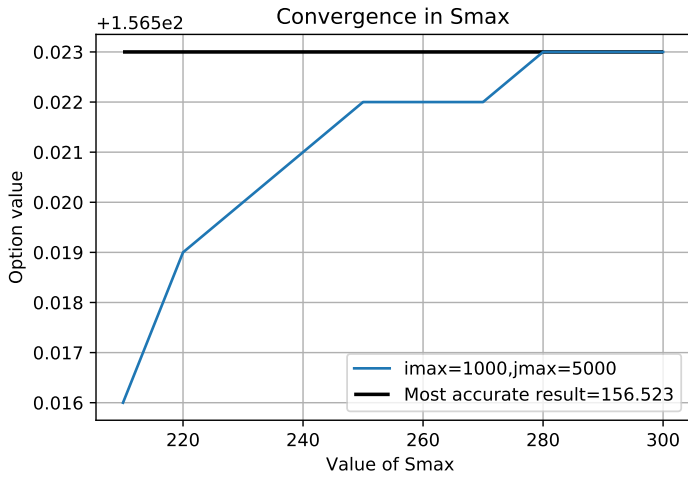


Figure 5: How $S_{max} affects$

our evaluation at boundary condition $S \to \infty$ during the pricing process. Also at expiry, the only non-linearity error happens at $2S = F = 140$, so we should choose $S_{max} = 70N$ to remove it. We can observe option value varies noticeably with different S_{max}. **At around $S_{max} = 150$ it start to converge to the correct value, the second graph of Figure(5) shows more clear that 280 is the key point. Thus we can treat $S_{max} = 2F = 280$ a safe level, this also underpins our assumption at the beginning where we fix $S_{max} = 280$.**

•**Conclusion** From above analysis, to secure the most efficient combination $(imax, jmax, S_{max})$, say the highest accuracy with least computation, we can choose $(100, 2800, 280)$. We may also decrease their values to achieve the same accuracy, but securely it is a proper and acceptable combination designed for efficiency. With this setup, the option value $V(67.58) = 156.523$

3 Embedded Options

Methodology

In this part I apply penalty method, then the PDE looks like:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 V}{\partial S^2} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} + \rho \max(RS - V, 0) = 0. \tag{12}$$

And accordingly numerical matrix will be added with penalty terms if $V_j^i < RS_j$, otherwise stays. Note that the time needs to be divided into two regions where we check embedded put is available or not. For $t < t_0$ where the holder can sell bond back, we need to compare V_j^i with both two conditions: Convert and Sell the bond back.

3.1 Prediction of Option Values

Based on above method, we apply parameters provided in the question to give a prediction of option values. We observe Figure(6) the figure of $V(S, t)$ is always above two horizontal lines. It starts from $P_p = 150$ and then goes up as S increases, finally it joins $f(S) = 2S$. This fits our assumption that

$$V(S, t) \geq P_p \quad if \quad t \leq t_0$$

There are two optimal decision points on the graph

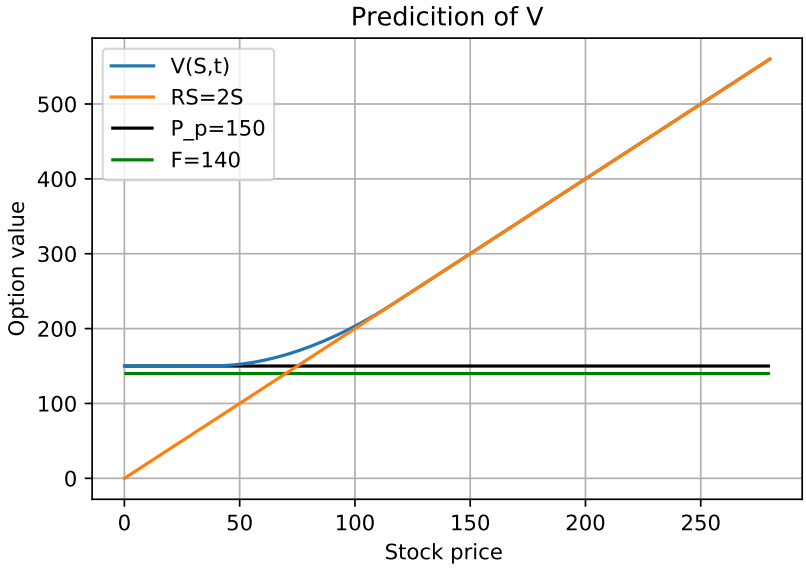


Figure 6: How $V(S,t)$ behaves

- $V(S,t) = P_p$. In this case, sell back the bond is the optimal decision. When S_0 starts at a low level, the possibility of big increase is small, forcing the option value to equal $P_p = 150$.
- $V(S,t) = RS$. In this situation, S_0 is rather big and holder will not doubt convert the bond.

3.2 Effect of parameter

Next we investigate how option value changes with various parameters C, which is related to the issuer's coupon policy. For $C \in \{0.705, 1.41, 2.115\}$, observe the following graph From the first graph

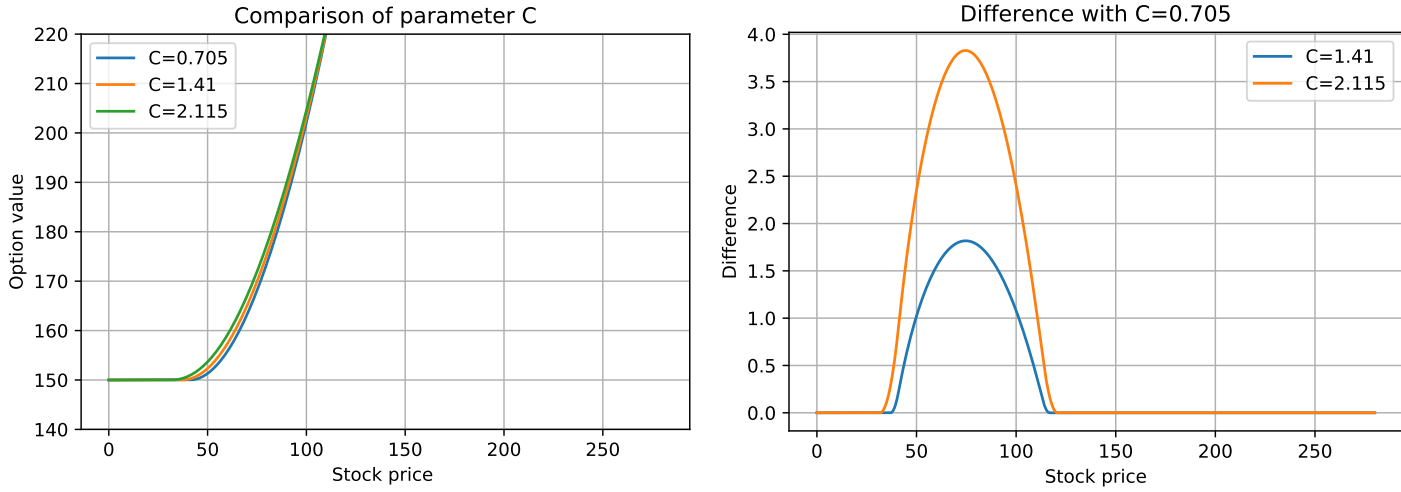


Figure 7: How parameter C affects

of Figure(7), there is only slight difference between three lines because change in C is small. We

can find bigger C means bigger option values. However when S increases, three lines gradually meet together and even overlap completely for large S. According to the second graph, I plot out the difference between other two lines with bigger C and the one with $C = 0.705$. Difference starts to emerge at around $S = 30$ and reaches the peak at around $S = 75$, then goes down. This is expected, because they will behave identically at two optimal decision points that discussed in (3.1).
• **Reason**: Larger C $\rightarrow$ larger V, it makes sense from the definition of C. The holder will receive bond coupon at each instant by an amount of $Ce^{-\alpha t}$. So if C is larger the contract will be more valuable.

3.3 Efficiency&Accuracy

In numerical scheme, the result we obtain is an approximated result. Commonly this will be different from real answer due to existence of error. Usually there are a few types of error:

$$V_{numerical} = V_{analytic} + Error_{scheme} + Error_{machine} + Error_{boundary}$$

The main error in Crank-Nicholson for American style options will be non-linearity error, which happens at early exercise positions. It is hard to capture this free boundary position. It also exists in time gird, because early exercise boundary does not move exactly as Δt. Basically, we can apply Body-fitted method to mitigate discontinuity error, but it is complicated to code up. Instead I will compare three matrix solvers and the method to interpolate or extrapolate the option value.
Matrix Solver

- Crank-Nicholson with PSOR;
- Crank-Nicholson with policy iteration;
- Crank-Nicholson with penalty method.

All three methods do not really remove non-linearity error produced at early exercises. PSOR method applies iterations to solve linear matrix problems while other two use direct methods. So Policy iteration and penalty are faster such that we can put in more girds to force $\Delta t, \Delta S \rightarrow 0$.

Interpolation&Extrapolation

- Linear interpolation;
- Quadratic interpolation;
- Quadratic interpolation+Richardson extrapolation.

This part involve how can we obtain an option value with given initial stock price S_0. Quadratic interpolation is always more accurate than linear interpolation in theory, but it takes more time. Notice that extrapolation is a fantastic way to verify convergence and give accurate results.
Analytic solution
In comparison of accuracy, usually we need a benchmark. Here I will do similar to that in (2.4). I will apply penalty method with quadratic interpolation, then extrapolate[1] the final answer with at most ($imax = jmax = 4^7 = 16384$). In this case the process takes seconds to produce $\mathbf{V(67.58)_{true} = 162.944801}$.
Source of discontinuity
Unlike European style option where discontinuity in S dominates. Our embedded put contract involves early conversion as well as the option to sell back the bond. So there may also exist significant discontinuity in time t. Now we empirically investigate which source of discontinuity stands out.
•Observe Figure(8),clearly imax dominates the magnitude of error. Orange line tells if we set imax large, then the error is small even with a small jmax. Blue line tells how error decreases with increasing imax. **Thus with fixed computation, increasing more on imax is a better strategy!**

[1]Extrapolation starts at n=4 and doubles each time

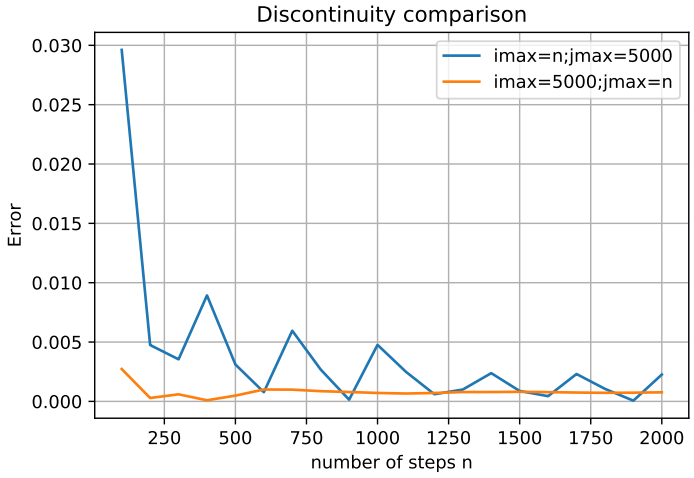


Figure 8: imax and jmax comparison

One second challenge

Given one second of computation limit, I draw a few tables to compare accuracy of different setups in terms of error, in 6 decimal points. Since above $V(67.58)_{true}$ is obtained by using Richardson extrapolation, itself is verified as $(imax, jmax)$ increases. Thus any results yielded by other methods close to V_{true} are correct. After some trails, below I give 6 well-performed candidates:

- (1) Penalty+Linear interpolation;
- (2) PSOR+Quadratic interpolation;
- (3) Policy iteration+Quadratic interpolation;
- (4) Penalty+Quadratic interpolation;
- (5) Policy iteration+Quadratic interpolation+Extrapolation;
- (6) Penalty+Quadratic interpolation+Extrapolation;

(1)	(2)	(3)	(4)	(5)	(6)
$\varepsilon = 1.6 * 10^{-4}$	$\varepsilon = 1.46 * 10^{-2}$	$\varepsilon = 2.14 * 10^{-3}$	$\varepsilon = 2.64 * 10^{-4}$	$\varepsilon = 8.32 * 10^{-5}$	$\varepsilon = 9.28 * 10^{-5}$
$t = 0.886s$	$t = 0.989$	$t = 1.054s$	$t = 1.08s$	$t = 1.187s$	$t = 0.736$
$imax = 8000$	$imax = 1500$	$imax = 4000$	$imax = 2200$	same with (6)	$imax = 4; \leq 5000; * = 2$
$jmax = 500$	$jmax = 500$	$jmax = 1000$	$jmax = 2000$	$jmax = 500 + \frac{imax}{20}$	$jmax = 200 + \frac{imax}{10}$
$S_{max} = 300$	$S_{max} = 300$	$S_{max} = 300$	$S_{max} = 300$	$S_{max} = 300$	$S_{max} = 300$

Table 1: Comparison among above methods

Stick to 1 second, method (6) is the most preferable one. Error yielded by this is decreasing ideally to a very low level. In dealing with discontinuities, the only idea here is to shorten gird length both in S and t. Here we emphasize on t according to previous analysis. However, our V_{true} is not a real solution so there must exists bias, as well as other source of error. Generally thinking, the value produced by

(6) is a proper result for one second.
The most accurate solution within 1 second: $\mathbf{V(67.58)_6 = 162.944708}$

.1 Source.cpp

```
#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
#include<fstream>
#include <cassert>
#include <chrono>
using namespace std;
using namespace chrono;
class CB
{
public:
    CB(double T_, double F_, double R_, double r_, double kappa_, double mu_, double X_,
        double C_, double alpha_, double beta_, double sigma_, double Smax_, int imax_, int jmax_) :
        T(T_), F(F_), R(R_), r(r_), kappa(kappa_),
        mu(mu_), X(X_), C(C_), alpha(alpha_), beta(beta_), sigma(sigma_), Smax(Smax_), imax(imax_), jmax(jmax_){};
    //constructor.
    vector<double> EU_CN()
    {
        vector<double>vold(jmax + 1), vnew(jmax + 1);
        for (int j = 0; j <= jmax; j++)
        {
            vold[j] = max(F, R * j * dS);
        }
        for (int i = imax-1; i >= 0; i--)
        {
            double t = (i + 0.5) * dt; double t1 = i * dt;
            vector<vector<double>>M = matrix(vold, t, t1);
            vnew = thomasSolve(M[0], M[1], M[2], M[3]);
            vold = vnew;
        }
        return vnew;
    }

    vector<double> AMP(double P,double t0,double rho,double tol,int iterMax)
    {
        vector<double>vold(jmax + 1), vnew(jmax + 1);
        for (int j = 0; j <= jmax; j++)
        {
            vold[j] = max(F, R * j * dS);
        }
        for (int i = imax - 1; i >= 0; i--)
        {
            double t = (i + 0.5) * dt; double t1 = i * dt;
            if (t1 >= t0)
            {
                int penaltyIt;
                vector<vector<double>>M = matrix1(vold, t, t1);
                for (penaltyIt = 0; penaltyIt < iterMax; penaltyIt++)
                {
                    vector<double> aHat(M[0]), bHat(M[1]), cHat(M[2]), dHat(M
```

```
[3]);
                    for (int j = 1; j < jmax; j++)
                    {
                        if (vnew[j] < R*dS*j)
                        {
                            bHat[j] = M[1][j] - rho; dHat[j] = M[3][j] - rho * (R * dS * j);
                        }
                    }
                    vector<double> y = thomasSolve(aHat, bHat, cHat, dHat);
                    double error = 0.;
                    for (int j = 0; j <= jmax; j++)
                        error += (vnew[j] - y[j]) * (vnew[j] - y[j]);
                    vnew = y;
                    if (error < tol * tol)
                    {
                        break;
                    }
                }
                if (penaltyIt >= iterMax)
                {
                    cout << " Error NOT converging within required iterations" << endl;
                    throw;
                }
                vold = vnew;
            }
            if (t1 < t0)
            {
                {
                    int penaltyIt;
                    vector<vector<double>>M = matrix1(vold, t, t1);
                    for (penaltyIt = 0; penaltyIt < iterMax; penaltyIt++)
                    {
                        vector<double> aHat(M[0]), bHat(M[1]), cHat(M[2]), dHat(M[3]);
                        for (int j = 0; j < jmax; j++)
                        {
                            double D = max(R * dS * j, P);
                            if (vnew[j] < D)
                            {
                                bHat[j] = M[1][j] - rho; dHat[j] = M[3][j] - rho * D;
                            }

                        }
                        vector<double> y = thomasSolve(aHat, bHat, cHat, dHat);
                        double error = 0.;
                        for (int j = 0; j <= jmax; j++)
                            error += (vnew[j] - y[j]) * (vnew[j] - y[j]);
                        vnew = y;
                        if (error < tol * tol)
                        {
                            break;
                        }
```

```
102                     }
103                     if (penaltyIt >= iterMax)
104                     {
105                         cout << " Error NOT converging within required
    iterations" << endl;
106                         throw;
107                     }
108                     vold = vnew;
109                 }
110             }
111         }
112         return vnew;
113     }
114     vector<double> AMPolicy(double P, double t0, double rho, double tol, int
    iterMax)
115     {
116         vector<double>vold(jmax + 1), vnew(jmax + 1);
117         for (int j = 0; j <= jmax; j++)
118         {
119             vold[j] = max(F, R * j * dS);
120         }
121         for (int i = imax - 1; i >= 0; i--)
122         {
123             double t = (i + 0.5) * dt; double t1 = i * dt;
124             if (t1 >= t0)
125             {
126                 int penaltyIt;
127                 vector<vector<double>>M = matrix1(vold, t, t1);
128                 for (penaltyIt = 0; penaltyIt < iterMax; penaltyIt++)
129                 {
130                     vector<double> aHat(M[0]), bHat(M[1]), cHat(M[2]), dHat(M
    [3]);
131                     for (int j = 1; j < jmax; j++)
132                     {
133                         if (vnew[j] < R * dS * j)
134                         {
135                             aHat[j] = 0.; cHat[j] = 0.;
136                             bHat[j] = 1.;dHat[j] = R * dS * j;
137                         }
138                     }
139                     vector<double> y = thomasSolve(aHat, bHat, cHat, dHat);
140                     double error = 0.;
141                     for (int j = 0; j <= jmax; j++)
142                         error += (vnew[j] - y[j]) * (vnew[j] - y[j]);
143                     vnew = y;
144                     if (error < tol * tol)
145                     {
146                         break;
147                     }
148                 }
149                 if (penaltyIt >= iterMax)
150                 {
151                     cout << " Error NOT converging within required iterations"
     << endl;
152                     throw;
153                 }
```

```
154                 vold = vnew;
155             }
156             if (t1 < t0)
157             {
158                 {
159                     int penaltyIt;
160                     vector<vector<double>>M = matrix1(vold, t, t1);
161                     for (penaltyIt = 0; penaltyIt < iterMax; penaltyIt++)
162                     {
163                         vector<double> aHat(M[0]), bHat(M[1]), cHat(M[2]),
    dHat(M[3]);
164                         for (int j = 0; j < jmax; j++)
165                         {
166                             double D = max(R * dS * j, P);
167                             if (vnew[j] < D)
168                             {
169                                 aHat[j] = 0.; cHat[j] = 0.;
170                                 bHat[j] = 1.; dHat[j] = D;
171                             }
172 
173                         }
174                         vector<double> y = thomasSolve(aHat, bHat, cHat, dHat)
    ;
175                         double error = 0.;
176                         for (int j = 0; j <= jmax; j++)
177                             error += (vnew[j] - y[j]) * (vnew[j] - y[j]);
178                         vnew = y;
179                         if (error < tol * tol)
180                         {
181                             break;
182                         }
183                     }
184                     if (penaltyIt >= iterMax)
185                     {
186                         cout << " Error NOT converging within required
    iterations" << endl;
187                         throw;
188                     }
189                     vold = vnew;
190                 }
191             }
192         }
193         return vnew;
194     }
195     double AMexplicit(double S0, double X, double T, double r, double sigma,
    int iMax, int jMax, double S_max)
196     {
197         double dS = S_max / jMax;
198         double dt = T / iMax;
199         vector<double> S(jMax + 1), vOld(jMax + 1), vNew(jMax + 1);
200         for (int j = 0; j <= jMax; j++)
201         {
202             S[j] = j * dS;
203         }
204         for (int j = 0; j <= jMax; j++)
205         {
```

```
206             vOld[j] = max(X - S[j], 0.);
207             vNew[j] = max(X - S[j], 0.);
208         }
209         for (int i = iMax - 1; i >= 0; i--)
210         {
211             vNew[0] = X * exp(-r * (T - i * dt));
212             for (int j = 1; j <= jMax - 1; j++)
213             {
214                 double A, B, C;
215                 A = 0.5 * sigma * sigma * j * j * dt + 0.5 * r * j * dt;
216                 B = 1. - sigma * sigma * j * j * dt;
217                 C = 0.5 * sigma * sigma * j * j * dt - 0.5 * r * j * dt;
218                 vNew[j] = 1. / (1. + r * dt) * (A * vOld[j + 1] + B * vOld[j]
    + C * vOld[j - 1]);
219             }
220             vNew[jMax] = 0.;
221             vOld = vNew;
222         }
223         int jstar;
224         jstar = S0 / dS;
225         double sum = 0.;
226         sum = sum + (S0 - S[jstar + 1]) / (S[jstar] - S[jstar + 1]) * vNew[
    jstar];
227         sum = sum + (S0 - S[jstar]) / (S[jstar + 1] - S[jstar]) * vNew[jstar +
     1];
228         return sum;
229     }
230     double inter(double S0)
231     {
232         int js;
233         js = S0 / dS;
234         vector<double> v = EU_CN();
235         vector<double> S1 = S();
236         return  ( S0 - S1[js + 1] ) / ( S1[js] - S1[js + 1] ) * v[js] + ( S0 -
     S1[js] ) / ( S1[js + 1] - S1[js] ) * v[js + 1];
237     }
238     double Hinter(double S0)
239     {
240         int js;
241         js = S0 / dS;
242         vector<double> v = EU_CN();
243         vector<double> S1 = S();
244         double A = ((S0 - S1[js]) * (S0 - S1[js + 1])) / ((S1[js - 1] - S1[js
    ]) * (S1[js - 1] - S1[js + 1])) * v[js - 1];
245         double B = ((S0 - S1[js-1]) * (S0 - S1[js + 1])) / ((S1[js] - S1[js
    -1]) * (S1[js] - S1[js + 1])) * v[js];
246         double C = ((S0 - S1[js-1]) * (S0 - S1[js])) / ((S1[js+1] - S1[js-1])
    * (S1[js+1] - S1[js])) * v[js + 1];
247         return A + B + C;
248     }
249     double inter1(double S0, double P, double t0, double rho, double tol, int
    iterMax)
250     {
251         int js;
252         js = S0 / dS;
253         vector<double> v = AMPolicy(P, t0, rho, tol, iterMax);
```

```
254         vector<double> S1 = S();
255         return  (S0 - S1[js + 1]) / (S1[js] - S1[js + 1]) * v[js] + (S0 - S1[
    js]) / (S1[js + 1] - S1[js]) * v[js + 1];
256     }
257     double Hinter1(double S0, double P, double t0, double rho, double tol, int
     iterMax,int c)
258     {
259         int js;
260         js = S0 / dS;
261         vector<double> v;
262         if (c == 1)
263         {
264             v = AMP(P, t0, rho, tol, iterMax);
265         }
266         if (c == 2)
267         {
268             v = AMPolicy(P, t0, rho, tol, iterMax);
269         }
270         vector<double> S1 = S();
271         double A = ((S0 - S1[js]) * (S0 - S1[js + 1])) / ((S1[js - 1] - S1[js
    ]) * (S1[js - 1] - S1[js + 1])) * v[js - 1];
272         double B = ((S0 - S1[js - 1]) * (S0 - S1[js + 1])) / ((S1[js] - S1[js
    - 1]) * (S1[js] - S1[js + 1])) * v[js];
273         double C = ((S0 - S1[js - 1]) * (S0 - S1[js])) / ((S1[js + 1] - S1[js
    - 1]) * (S1[js + 1] - S1[js])) * v[js + 1];
274         return A + B + C;
275     }
276     double check(double S1, double t1)
277     {
278         return R * S1 * exp(-(kappa + r) * (T - t1)) + X * R * exp(-r * (T -
    t1)) * (1. - exp(-kappa * (T - t1))) +
279             C / (alpha + r) * (exp(-alpha * t1) - exp(-(alpha + r) * T) * exp(
    r * t1));
280     }
281 private:
282     double theta(double t)
283     {
284         return (1. + mu) * X * exp(mu * t);
285     }
286     double K(double t)
287     {
288         return C * exp(-alpha * t);
289     }
290     vector<double> S()
291     {
292         vector<double>v(jmax + 1);
293         for (int j = 0; j <= jmax; j++)
294         {
295             v[j] = j * dS;
296         }
297         return v;
298     }
299     vector<vector<double>> matrix(const vector<double> &vold,double t,double
    t1)
300     {
301         vector<double> a(jmax + 1), b(jmax + 1), c(jmax + 1), d(jmax + 1);
```

```
302         a[0] = 0.;
303         b[0] = -1. / dt - kappa * theta(t1) / dS - r;
304         c[0] = kappa * theta(t1) / dS;
305         d[0] = -(1. / dt) * vold[0] - K(t1);
306         for (int j = 1; j <= jmax - 1; j++)
307         {
308             a[j] = 0.25*(sigma * sigma * pow(j, 2. * beta) * pow(dS, 2. * (
    beta - 1.)) - kappa * theta(t) / dS + kappa * j);
309             b[j] = -0.5 * sigma * sigma * pow(j, 2. * beta) * pow(dS, 2. * (
    beta - 1.)) - 1. / dt - r / 2.;
310             c[j]= 0.25*(sigma * sigma * pow(j, 2. * beta) * pow(dS, 2. * (beta
     - 1.)) + kappa * theta(t) / dS - kappa * j);
311             d[j] = -a[j] * vold[j - 1] - c[j] * vold[j + 1] - (b[j] + 2. / dt)
     * vold[j] - K(t);
312         }
313         a[jmax] = 0.;
314         b[jmax] = 1.;
315         c[jmax] = 0.;
316         d[jmax] = R * Smax * exp(-(kappa + r)*(T - t1)) + X * R * exp(-r * (T
    - t1)) * ( 1. - exp(-kappa * (T - t1)) ) +
317             C / (alpha + r) * ( exp(-alpha * t1) - exp(-(alpha + r) * T) * exp
    (r * t1) );
318         return { a,b,c,d };
319     }
320     vector<vector<double>> matrix1(const vector<double>& vold, double t,
    double t1)
321     {
322         vector<double> a(jmax + 1), b(jmax + 1), c(jmax + 1), d(jmax + 1);
323             a[0] = 0.;
324             b[0] = -1. / dt - kappa * theta(t1) / dS - r;
325             c[0] = kappa * theta(t1) / dS;
326             d[0] = -(1. / dt) * vold[0] - K(t1);
327             for (int j = 1; j <= jmax - 1; j++)
328             {
329                 a[j] = 0.25 * (sigma * sigma * pow(j, 2. * beta) * pow(dS, 2.
    * (beta - 1.)) - kappa * theta(t) / dS + kappa * j);
330                 b[j] = -0.5 * sigma * sigma * pow(j, 2. * beta) * pow(dS, 2. *
     (beta - 1.)) - 1. / dt - r / 2.;
331                 c[j] = 0.25 * (sigma * sigma * pow(j, 2. * beta) * pow(dS, 2.
    * (beta - 1.)) + kappa * theta(t) / dS - kappa * j);
332                 d[j] = -a[j] * vold[j - 1] - c[j] * vold[j + 1] - (b[j] + 2. /
     dt) * vold[j] - K(t);
333             }
334             a[jmax] = 0.;
335             b[jmax] = 1.;
336             c[jmax] = 0.;
337             d[jmax] = R * Smax;
338             return { a,b,c,d };
339     }
340     vector<double> thomasSolve(const std::vector<double>& a, const std::vector
    <double>& b_, const std::vector<double>& c, std::vector<double>& d)
341     {
342         int n = a.size();
343         std::vector<double> b(n), temp(n);
344         // initial first value of b
345         b[0] = b_[0];
```

```
346         for (int j = 1; j < n; j++)
347         {
348             b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
349             d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
350         }
351         // calculate solution
352         temp[n - 1] = d[n - 1] / b[n - 1];
353         for (int j = n - 2; j >= 0; j--)
354             temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];
355         return temp;
356     }
357 
358     double T, F, R, r, kappa, mu, X, C, alpha, beta, sigma, Smax;
359     int imax, jmax;
360     double dt = T / imax; double dS = Smax / jmax; double dt1 = T / (imax *
        imax);
361 };
362 
363 int main()
364 {
365     /*
366     CB example1(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 0.705, 0.01, 0.787,
         0.625, 280., 500, 1000);
367     CB example2(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 1.41, 0.01, 0.787,
        0.625, 280., 500, 1000);
368     CB example3(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 2.115, 0.01, 0.787,
         0.625, 280., 500, 1000);
369     vector<double> a1 = example1.AMP(150., 1.8849, 1e6, 1e-6, 1000);
370     vector<double> a2 = example2.AMP(150., 1.8849, 1e6, 1e-6, 1000);
371     vector<double> a3 = example3.AMP(150., 1.8849, 1e6, 1e-6, 1000);
372     ofstream w("vs.csv");
373     double ds = 280. / 1000.;
374     for (int j = 0; j <= 1000; j++)
375     {
376         w << ds * j << ',' << a2[j]-a1[j] << ','<< a3[j]-a1[j] << endl;
377         //w << ds * j << ',' << a1[j] << ','<<a2[j]<<',' << a3[j]<< endl;
378     }
379     */
380     /*
381     CB example2(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 1.41, 0.01, 0.787,
        0.625,280., 4000, 1000);
382 
383     auto start = high_resolution_clock::now();
384     double a=example2.Hinter1(67.58, 150., 1.8849, 1e6, 1e3, 1000,2);
385     auto stop = high_resolution_clock::now();
386     auto duration = duration_cast<milliseconds>(stop - start);
387     cout.precision(12); cout << "value :" << fabs(162.944801-a) << endl;
388     cout << "Time taken:" << duration.count() << endl;
389     */
390     /*
391     ofstream w("out.csv");
392     for (int n = 100; n <= 2000; n +=100)
393     {
394         CB example1(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 1.41, 0.01,
        0.787, 0.625, 280., n, 5000);
395         CB example2(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 1.41, 0.01,
```

```
      0.787, 0.625, 280., 5000,n);
396         double a1 = example1.inter1(67.58, 150., 1.8849, 1e6, 1e-6, 1000);
397         double a2 = example2.inter1(67.58, 150., 1.8849, 1e6, 1e-6, 1000);
398         w << n << ','<<fabs(162.944801 - a1) << ',' << fabs(162.944801 - a2)
      << endl;
399     }
400     */
401
402     double vo = 0.;
403     double extravalue = 0.;
404     auto start = high_resolution_clock::now();
405     for (int n = 4; n <= 5000; n *= 2)
406     {
407         CB example2(5., 140., 2., 0.0202, 0.05, 0.0186, 67.58, 1.41, 0.01,
      0.787, 0.625, 300.,n,500+n/20.);
408         double vn = example2.inter1(67.58, 150., 1.8849, 1e6, 1e3, 1000);
409         if (n >= 8)
410         {
411             extravalue = (4. * vn - vo) / 3.;
412         }
413         double dif = fabs(162.944801 - extravalue);
414         if (dif < 1e-3)
415         {
416             cout.precision(12); cout << dif<< endl;
417         }
418         vo = vn;
419     }
420     auto stop = high_resolution_clock::now();
421     auto duration = duration_cast<milliseconds>(stop - start);
422     cout << "Time taken:" << duration.count() << endl;
423
424     return 0;
425 }
```