Project 2: Finite Differences

MATH40082 Computational Finance

Student No: 9917395

1 Introduction

Finite difference methods (FDMs) are a tool based on discretisation, which is widely used in mathematics and the natural sciences to compute numerical solutions to ordinary or partial differential equations. By breaking down the domain into a typically uniform, finite grid, the solution to such equations can be expressed as a linear algebra problem, solved by matrix analysis [1].

This approximation is widely used and, while it may be very powerful, the different sources of error must be acknowledged. One possible source is due to the loss of precision and roundoff errors as a consequence of the computer's finite memory. However, the main source typically comes from discretisation errors, which will depend on the chosen scheme. For parabolic PDEs (such as the heat equation), one of three schemes is typically used: the explicit, implicit or the Crank Nicolson method [2]. The latter uses central differences and is hence a second order method, which is also stable everywhere in the domain.

A large application of FDMs is in the financial industry, where they can be used to value exotic options of unknown analytic solution. One such example are convertible bonds, where at maturity $t = T$, the holder can choose between receiving a fixed principle F, or receiving R underlying stocks. The payoff of one such bond contract is hence given by:

$$V(S, t = T) = \max\{F, RS\} \tag{1}$$

In this project, a bond contract paying continuous coupons will be valued using a Crank-Nicolson scheme. In the risk neutral measure, the bond is governed by the stochastic differential equation:

$$dS = \kappa(\theta(t) - S)dt + \sigma S^{\beta} dW \tag{2}$$

Where:

- κ is the mean reversion rate, which quantifies the extent to which the underlying asset is likely to revert to it's long term mean over time.
- β denotes the elasticity of the variance.
- $\theta(t) = (1 + \mu)Xe^{\mu t}$ for parameters X and μ captures how the firm pays out dividends.

It may be shown that the value of this contract is governed by the following PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 V}{\partial S^2} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0 \tag{3}$$

The following standard values for the model parameter will be assumed: $T = 3$, $F = 190$, $R = 2$, $r = 0.0043$, $\kappa = 0.83333$, $\mu = 0.0076$, $X = 95.24$, $C = 0.408$, $\alpha = 0.03$, $\beta = 0.142$, $\sigma = 17.1$.

Furthermore, it will be studied how American style call option can be embedded into the contract. In this case, the holder will be able to convert into stock at any time $t < T$ but the issuer may buy back the bond at a price C_p for a period $t < t_0$.

2 Numerical Scheme

2.1 Boundary Condition at $S \to \infty$:

The boundary conditions for the problem must be derived for $S = 0$ and $S \to \infty$. For $S = 0$ and by substitution into (3), it follows that the boundary is governed by the PDE:

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

The boundary condition at $S \to \infty$ can be derived by solving the approximate problem:

$$\frac{\partial V}{\partial t} + \kappa(X - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0 \tag{4}$$

Which has a solution of the form $V(S,t) = SA(t) + B(t)$ for functions $A(t), B(t)$ to be determined. If we substitute this Ansatz into (4), we arrive at:

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

Since the equation must hold $\forall S$, gathering powers of S leads to two ODEs for $A(t)$ and $B(t)$:

$$\underline{S^1 \text{ terms:}} \qquad A'(t) - (\kappa + r)A(t) = 0 \tag{5}$$

$$\underline{S^0 \text{ terms:}} \qquad B'(t) + \kappa X A(t) - rB(t) + Ce^{-\alpha t} = 0 \tag{6}$$

Along with the boundary conditions $A(t = T) = R$, $B(t = T) = 0$.

ODE (5) is separable and can be solved relatively straightforward as:

$$A(t) = \lambda e^{(\kappa + r)t} \qquad \text{for some } \lambda \in \mathbb{R}.$$

Applying the boundary condition $A(t = T) = R$, we find that $\lambda = Re^{-(\kappa+r)T}$ and hence:

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

To solve ODE (6), we can use the integrating factor:

$$B(t) = -e^{rt} \int e^{-rt} \left[\kappa X R e^{(\kappa+r)(t-T)} + Ce^{-\alpha t} \right] dt$$

Which yields:

$$B(t) = \frac{C}{\alpha + r} e^{-\alpha t} - XRe^{(\kappa+r)(t-T)} + \mu e^{rt} \qquad \text{for some } \mu \in \mathbb{R}.$$

Applying the boundary condition $B(t = T) = 0$ gives $\mu e^{rt} = XRe^{r(t-T)} - \frac{C}{\alpha+r} e^{-\alpha T + r(t-T)}$. Therefore, alltogether:

$$\boxed{B(t) = \frac{C}{\alpha + r} \left[e^{-\alpha t} - e^{-\alpha T + r(t-T)} \right] + XR \left[e^{r(t-T)} - e^{(\kappa+r)(t-T)} \right].}$$

And hence:

$$V(S \to \infty, t) \approx SRe^{(\kappa+r)(t-T)} + \frac{C}{\alpha + r} \left[e^{-\alpha t} - e^{-\alpha T + r(t-T)} \right] + XR \left[e^{r(t-T)} - e^{(\kappa+r)(t-T)} \right] \tag{7}$$

2.2 Numerical scheme derivation:

A numerical scheme for a Crank-Nicolson method with $S = j\Delta S$ and $t = i\Delta t$ can be derived using central differences [3]:

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

$$\frac{\partial V}{\partial S} \approx \frac{1}{4\Delta S} \left[V_{j+1}^i - V_{j-1}^i + V_{j+1}^{i+1} - V_{j-1}^{i+1} \right]$$

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

$$V \approx \frac{1}{2} \left[V_j^i + V_j^{i+1} \right]$$

to approximate PDE (3) governing the contract. Inserting the approximations for the derivatives into the PDE leads to:

$$\frac{V_j^{i+1}-V_j^i}{\Delta t}+\frac{1}{4}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)}[V_{j+1}^i-2V_j^i+V_{j-1}^i+V_{j+1}^{i+1}-2V_J^{i+1}+V_{j-1}^{i+1}\Big]$$
$$+\frac{1}{4}\kappa\Big(\frac{\theta(i\Delta t)}{\Delta S}-j\Big)\Big[V_{j+1}^i-V_{j-1}^i+V_{j+1}^{i+1}-V_{j-1}^{i+1}\Big]-\frac{r}{2}\Big[V_j^i+V_j^{i+1}\Big]+Ce^{-\alpha i\Delta t}=0.$$

Where $\theta(i\Delta t)=(1+\mu)Xe^{\mu i\Delta t}$. Grouping together all terms which depend on the current time i on one side, and those that depend on time $i+1$ on the other, leads to a matrix problem of the form: $a_jV_{j-1}^i+b_jV_j^i+c_jV_{j+1}^i=d_j$, where:

Coefficients in numerical scheme:

$$a_j=\frac{1}{4}\Big[\sigma^2j^{2\beta}(\Delta S)^{2(\beta-1)}-\kappa\Big(\frac{\theta(i\Delta t)}{\Delta S}-j\Big)\Big]$$
$$b_j=\Big[-\frac{\sigma^2j^{2\beta}(\Delta S)^{2(\beta-1)}}{2}-\frac{r}{2}-\frac{1}{\Delta t}\Big]$$
$$c_j=\frac{1}{4}\Big[\sigma^2j^{2\beta}(\Delta S)^{2(\beta-1)}+\kappa\Big(\frac{\theta(i\Delta t)}{\Delta S}-j\Big)\Big]$$

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

2.3 Linear algebra boundary conditions:

At $t=T$, $V(S,T)=\max(RS,F)$ and hence:

$$\boxed{V(j\Delta S,i_{max}\Delta t)=\max(Rj\Delta S,F)}$$

At $S\to\infty$:

$$V(S\to\infty,t)=SA(t)+B(t)\iff V(j_{max}\Delta S,\ i\Delta t)=j_{max}\Delta S\cdot A(i\Delta t)+B(i\Delta t),$$

and hence we require:

$$\boxed{a_{j_{max}}=0\qquad b_{j_{max}}=1\qquad d_{j_{max}}=(j_{max}\Delta S)\cdot A(i\Delta t)+B(i\Delta t).}$$

The boundary at $S=0$ is governed by the PDE:

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

Which can be approximated to:

$$\frac{V_0^{i+1}-V_0^i}{\Delta t}+\kappa\theta(i\Delta t)\Big(\frac{V_1^i-V_0^i}{\Delta S}\Big)-\frac{r}{2}\Big(V_0^i+V_0^{i+1}\Big)+Ce^{-\alpha i\Delta t}=0.$$

Note that forward differencing was used since V_{-1}^i is undefined, and hence we cannot use central differences. This will give rise to first order discretisation errors, which will dominate at the boundary. The above equation can be rewritten as:

$$\Big(-\frac{1}{\Delta t}-\frac{\kappa\theta(i\Delta t)}{\Delta S}-\frac{r}{2}\Big)V_0^i+\frac{\kappa\theta(i\Delta t)}{\Delta S}V_1^i=\Big(-\frac{1}{\Delta t}+\frac{r}{2}\Big)V_0^{i+1}-Ce^{-\alpha i\Delta t}.$$

And hence:

$$\boxed{b_0=-\Big(\frac{1}{\Delta t}+\frac{\kappa\theta(i\Delta t)}{\Delta S}+\frac{r}{2}\Big)\qquad c_0=\frac{\kappa\theta(i\Delta t)}{\Delta S}\qquad d_0=\Big(-\frac{1}{\Delta t}+\frac{r}{2}\Big)V_0^{i+1}-Ce^{-\alpha i\Delta t}.}$$

3 Analytic Solution for $\kappa = 0$, $\beta = 1$

The above scheme was solved using lower-upper (LU) decomposition, which solves tridiagonal matrix equations directly. This solver was preferred over iterative methods such as successive over-relaxation (SOR) due to it's better accuracy and ease to implement. To confirm the scheme is indeed correct, an analytic solution can be derived for the special case $\kappa = 0$, $\beta = 1$.

Under this circumstances, it may be known that:

$$V(S,t) = R \times C(S,t \mid F/R,\ D = r) + Fe^{r(t-T)} + \frac{C}{(\alpha + r)}\left[e^{-\alpha t} - e^{-\alpha T + r(t-T)}\right] \tag{8}$$

Where $C(S,t \mid F/R, D = r)$ denotes the value of a European Call option with strike price $\frac{F}{R}$, which pays a continuous dividend yield D equal to the interest rate r. A plot comparing the numerical and analytic solutions at $t = 0$ is provided in figure 1.

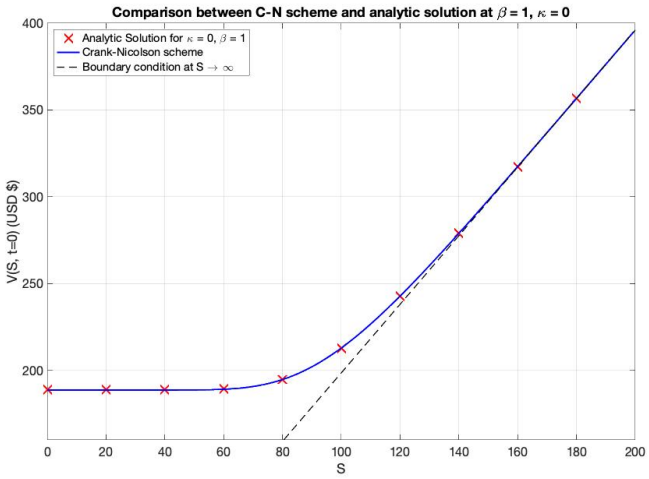


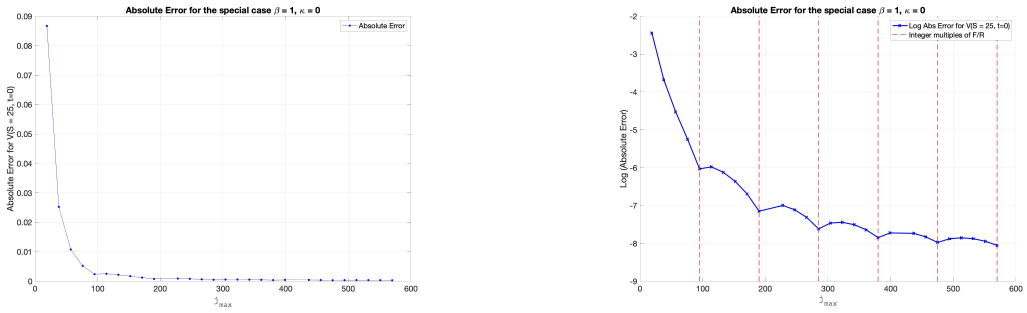
Figure 1: Plot of the analytical vs numerical solutions in the special case $\beta = 1$, $\kappa = 0$. The numerical solution is shown by the solid blue line and the analytic is depicted by red crosses at a few values of S. If a line had also been used for the latter, it would have been impossible to resolve the two curves. The dotted black line shows the boundary condition at $S \to \infty$ given by equation (7). The numerical solution indeed appears to converge smoothly towards this boundary.

For this special case, we can examine how the grid size influences the absolute error of the numerical solution. For a fixed $S = 25$, the absolute error on the contract was plotted against j_{max}, the total number of grid points in the S domain. These were chosen as $j_{max} = \frac{n \times S^{max}}{25}$, $n \in \mathbb{N}$ to ensure that the point $S = 25$ was always included in the grid. As more points are included in the grid, the smaller the spacing and hence, the better the approximation.

By equation (1), we known that the derivative of the payoff for the contract is discontinuous at $S = \frac{F}{R} = 95$. This may translate into non-linearity errors whenever attempting to use finite differences around this point. For best precision, dS should be chosen such that this point is always included in the grid. This explains the trend observed in plot 2(b), where the absolute error is plotted on a logarithmic scale. Whenever j_{max} approaches an integer multiple of $F/R = 95$ (i.e. whenever $j_{max} = 95m$, for some $m \in \mathbb{N}$) the absolute error is minimal. These multiples are depicted by the dashed red lines in the figure. The reason for this is that, since S^{max} was chosen at $5 \times \frac{F}{R}$:

$$n = 5m \qquad \text{and} \qquad dS = \frac{S^{max}}{j_{max}} = \frac{5}{m}, \qquad m \in \mathbb{N}$$

and hence dS will always be a divisor of $F/R = 95$. This implies that the point of discontinuity F/R is always included as a grid point, so linearity errors are minimised.



(a) Linear plot

(b) Semilog plot

Figure 2: Plot of the absolute error of the contract at $S = 25$, $t = 0$ for the special case $\beta = 1$, $\kappa = 0$. The standard deviation was chosen arbitrarily at $\sigma = 0.334$. Plot (a) is a linear plot. We can observe how the absolute error falls rapidly with j_{max} and seems to hit a local minima at $j_{max} = F/R$ and fluctuate thereafter. This trend is confirmed in plot (b) where the log absolute error is plotted instead.

4 Studying the effect of σ and β

Figure 3 shows a plot of $V(S, t = 0)$ vs S at different values of β and σ. The solid red line illustrates the case $\beta = 0.142,\ \sigma = 17.1$, while the blue line shows $\beta = 1,\ \sigma = 0.344$.

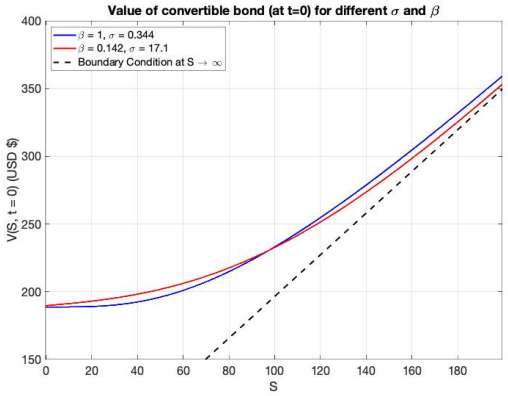


Figure 3: Plot of $V(S, t = 0)$ at $\beta = 1$, $\sigma = 0.344$ (in blue) and $\beta = 1, \sigma = 17.1$ (in red).

We can see from figure 3 that the two curves in fact look quite similar. The reason for this is that, we know that the variance of the increment in S, $\text{var}(S_{t+dt} - S_t)$, drives the contract price. For the SDE:

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

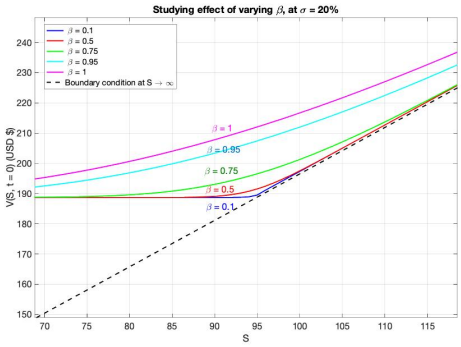
We can approximate the variance as [4]:

$$\text{var}(S_{t+dt} - S_t) \sim \sigma^2 S^{2\beta} dt$$

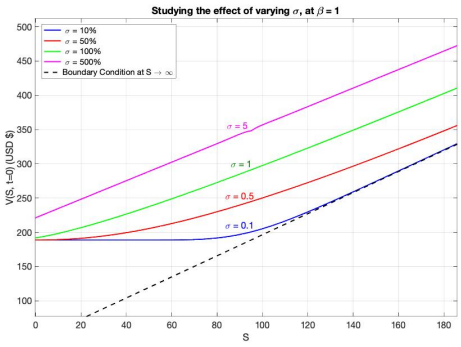
Furthermore, we know that variance of the stock is most influential on the contract value close to discontinuities. This explains why the two curves look alike. At the point $S = 95$ where variances is most significant towards the contract value, the ratio of the two variances:

$$\frac{17.1^2 \times 95^{(2\times 0.142)}}{0.344^2 \times 95^2} = 0.998 \approx 1$$

So at this point, the curves meet. For $S > 95$, the variance of the curve with $\beta = 1, \sigma = 0.344$ (the blue curve in figure 3) exceeds that of the red curve (with $\beta = 0.142$, $\sigma = 17.1$). The opposite is true for $S < 95$. This suggests that β controls the shape of the curve (shape parameter) while σ controls the spread (scale parameter) [5]. Hence, increasing σ should spread out the curve, while a smaller one should shrink it. This was explored further by fixing β and plotting curves at different σ, and vice versa. The results are given in figure 4.



(a) Plot of $V(S, t = 0)$ at fixed $\sigma = 0.2$ and varying β.



(b) Plot of $V(S, t = 0)$ at fixed $\beta = 1$ and varying σ.

Figure 4: Studying the effects of varying σ and β. All other parameters are kept as standard.

Figure 4(a) shows the effect on the value curve of varying β at fixed σ. As explained previously, the variance is most influential on the contract price around the discontinuity at $S = 95$. As expected, β controls the shape of the curve in this region. For β close to 0, the value curve stays flat until it hits the boundary condition at $S \to \infty$. For $\beta = 0.5$, the variance and hence the contract value grow linearly in S, until the boundary is reached. Similarly, for $\beta = 1$, the growth is quadratic.

In contrast, figure 4 (b) shows the effect on the value curve of varying σ at fixed β. We note that increasing σ, stretches out the curve towards infinity. A good analogy is to think of the $V(S, t = 0)$ curve as a string with it's end fixed at $S = 0$. σ then controls the strength with which the string is pulled from infinity. If this is large, the string becomes taut and the resultant curve appears to be linear. Consequently, the solutions converge towards the boundary at $S \to \infty$ at a slower rate.

5 Studying the effect of `iMax`, `jMax` and `S_max`

The effect of varying `iMax` and `jMax` was studied at $\beta = 0.142$, $\sigma = 17.1$ (all other parameters as standard). In order to observe the expected second order convergence, cubic interpolation will be used to infer the value at a fixed $S = 45$, $t = 0$. i.e. At each value of S, up to four terms will be considered in the Lagrange polynomial expansion of $V(S)$.

i_{max}	j_{max}	$V(S = 45)$	**Difference** ($\times 10^{-3}$)	**R**	**c**	**Elapsed Time (s)**
20	20	199.528	-	-	-	< 0.001
40	40	199.79	262	-	-	0.002
80	80	199.862	71.7	3.65968	1.87172	0.011
160	160	199.88	18.7	3.84119	1.94155	0.062
320	320	199.885	4.89	3.81758	1.93266	0.39
640	640	199.886	1.21	4.05664	2.02029	2.73
1280	1280	199.887	0.29	4.08816	2.03145	21.2
2560	2560	199.887	0.07	4.15065	2.05334	164

Table 1: Study of convergence rate of the estimates for $V(S = 45,\ t = 0)$. The difference is taken with the preceding value. R denotes the ratio of each difference with it's preceding value. $c = \frac{\log(R)}{\log(2)}$ is the convergence rate.

We can see from table 1 that as `jMax` and `iMax` double, R quadruples, which leads to a second order convergence as expected for a Crank-Nicolson scheme. On the other hand, duplicating `iMax` and `jMax` increases the computational time by a factor of ~ 7 on average. Since the convergence has been shown to be smooth, extrapolation methods can be used to improve the accuracy of the method. Richardson

extrapolation can be used such that, for duplicating `iMax` and `jMax`:

$$V(S,t) = \frac{2^2 V(S,t;i_{max},j_{max}) - V(S,t;\frac{i_{max}}{2},\frac{j_{max}}{2})}{2^2-1}$$

$i_{max} = j_{max}$	$V(S=45)$	**Difference** ($\times 10^{-3}$)	**Extrapolated Value**	**Difference in Extrap. Value** ($\times 10^{-5}$)
20	199.528	-	-	-
40	199.79	262	199.877	-
80	199.862	71.7	199.886	813
160	199.88	18.7	199.887	98.8
320	199.885	4.89	199.887	29.7
640	199.886	1.21	199.887	2.28
1280	199.887	0.29	199.887	0.87
2560	199.887	0.07	199.887	0.36

Table 2: Studying the effect of using Richardson extrapolation on the value of the bond contract at $S = 45$, $t = 0$.

The above table illustrates the effect of using Richardson extrapolation. We can note how, at $i_{max} = j_{max} = 160$, we obtain the same value (to 6 s.f) for the extrapolated value as for $i_{max} = j_{max} = 1280$ without extrapolation. This means that we can obtain precision to at least 6 significant figures in only 0.06 second of computational time, instead of 21.2 seconds. Furthermore, comparing the 3rd and 5th column we can confirm that the differences between the extrapolated values are converging at a much faster rate than those with no extrapolation.

The final consideration will be to study the effect of the upper boundary in the stock domain S^{max}. Since the discontinuity occurs at $S = \frac{F}{R} = 95$, this defines a natural scale for the problem. Setting S^{max} to an integer multiple of this values should ensure a smooth convergence. Up to now, this was kept fixed at $5 \cdot \frac{F}{R}$, but different integer multiples n of this scale are considered in table 3. The value of j_{max} was adjusted such that dS was held constant. By observing the convergence of the contract prices with increasing S^{max}, we should be able to extract an optimal range for this parameter.

n	S^{max}	j_{max}	$V(S=45)$	**Difference**	**Time Elapsed (s)**
2	190	380	199.884	-	0.162
3	285	570	199.887	0.003	0.336
4	380	760	199.887	2.90×10^{-6}	0.575
5	475	950	199.887	5.07×10^{-10}	0.899
6	570	1140	199.887	4.55×10^{-13}	1.245
7	665	1330	199.887	5.40×10^{-13}	1.687
8	760	1520	199.887	8.50×10^{-14}	2.18
9	855	1710	199.887	0	2.748
10	950	1900	199.887	0	3.499

Table 3: Studying the effect of S^{max} at $\kappa = 0.08333$, $\beta = 0.142$, $\sigma = 17.1$, $i_{max} = 200$.

We can see from table 3 that using $S^{max} = 5 \times \frac{F}{R} = 475$ is accurate to the order 10^{-13}, since increasing to $S^{max} = 6 \times \frac{F}{R}$ results in a difference of that magnitude. For $n > 8$, the difference converges to zero, so there is no point in going beyond this value. In order to hold dS constant, the values of j_{max} had to be scaled accordingly with increasing n, which had a direct impact on the computational time (factor of $1.5\times$ increase on average). This may become relevant when attempting very precise calculations at high i_{max} and j_{max}. We can therefore conclude that $n \in [5, 7]$ should give good enough precision to 10-14 decimal places, while having a minor impact on computational cost.

Finally, the contract was valued at the requested value $S_0 = 95.24$, $t = 0$, and with $\beta = 0.142$, $\sigma = 17.1$ (all other parameters as standard). This value is very close to the discontinuity, and hence, different values of $i_{max},\ j_{max}$ and S^{max} were tested. A relatively smooth convergence was observed for $S^{max} = 5 \times \frac{F}{R}$, and with $i_{max},\ j_{max}$ duplicating from 20. The results are shown in table 4.

$i_{max} = j_{max}$	$V(S = 95.24)$	**Extrapolated Value**	**R**	**Time Elapsed (s)**
20	227.8094	-	-	< 0.001
40	228.4983	228.72788	-	0.002
80	228.6662	228.72221	4.10	0.011
160	228.7085	228.72262	3.97	0.063
320	228.7194	228.72299	3.90	0.399
640	228.7222	228.72319	3.79	2.898
1280	228.7230	228.72328	3.62	20.92
2560	228.7232	228.72333	3.37	162.7

Table 4: Values with and without extrapolation for $V(S = 95.24,\ t = 0)$. The ratio of differences and time elapsed are also given.

Using extrapolation methods, we can obtain a value to 7 significant figures in 20 seconds of computational time. The final value is:

$$V(S = 95.24, t = 0) = 228.7233$$

The next iteration would require ~ 20 minutes of run-time. Furthermore, we can notice that as $i_{max} = j_{max}$ increases, the errors at the boundary and machine errors start to affect the convergence rate, ($c = 1.75$ at $j_{max} = 2560$). So going past this point would also reduce the efficiency of the extrapolation.

6 American style convertible bond

Here we examine the case where the issuer of the bond may want to embed American style options into the contract. To do so, the firm modifies the contract to allow the holder to convert into stock at any time prior to expiry ($t < T$). This leads to the inequality:

$$V \geq RS \qquad \forall\ t < T \tag{9}$$

We can further assume that in the limit $S \to \infty$, the holder will always choose to covert such that: $V(S, t) \to RS$ in this limit. This leads to the new linear algebra conditions at $S = j_{max}dS$:

$$a[j_{max}] = 0 \quad b[j_{max}] = 1 \quad c[j_{max}] = 0 \quad d[j_{max}] = (j_{max}dS)R$$

Secondly, the issuer reserves the option to buy back the bond at some strike price C_p but only during a limited period $t < t_0$. This leads to the inequality:

$$V(S, t) \leq \max(C_p, RS) \qquad \text{if } t < t_0 \tag{10}$$

This will assume that the holder can choose to convert before being bought out by the issuer. This means that whenever $RS > C_p$, the contract will cease to exist, since it will have either been called or converted. This gives a free choice for the option value for $RS > C_p$. By using inequality (10), we are choosing $V = RS$ in this region, to capture better what occurs in reality where time flows forwardly.

In order to value this contract, the penalty method with LU decomposition will be used. This is very efficient algorithm widely used in constrained optimisation problems. Whenever the value function fails to meet inequalities (9) and (10), a "penalty" is applied.

Figure 5 shows a plot of $V(S, t = 0)$ for a contract where the issuer may buy back the bond at a price $C_p = 230$ if $t < t_0 = 1.2654$. All other parameters were kept at standard. The red cross at $RS = C_p$ denotes a decision point. At this value, the holder may convert into stock to avoid being bought out. However, another valid assumption is that it is being called. Whichever occurs first does not impact the

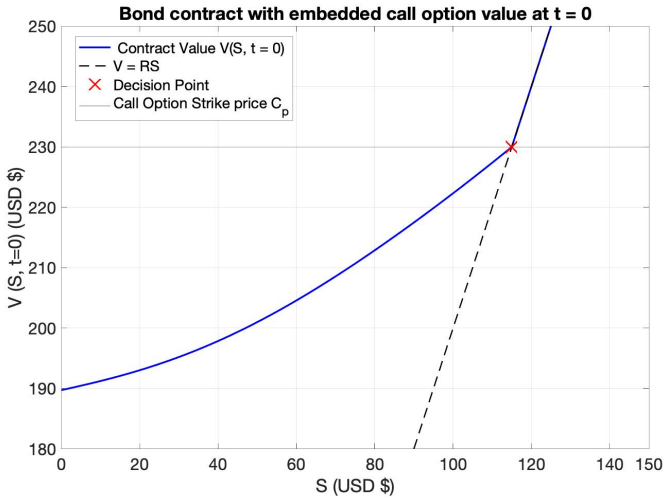
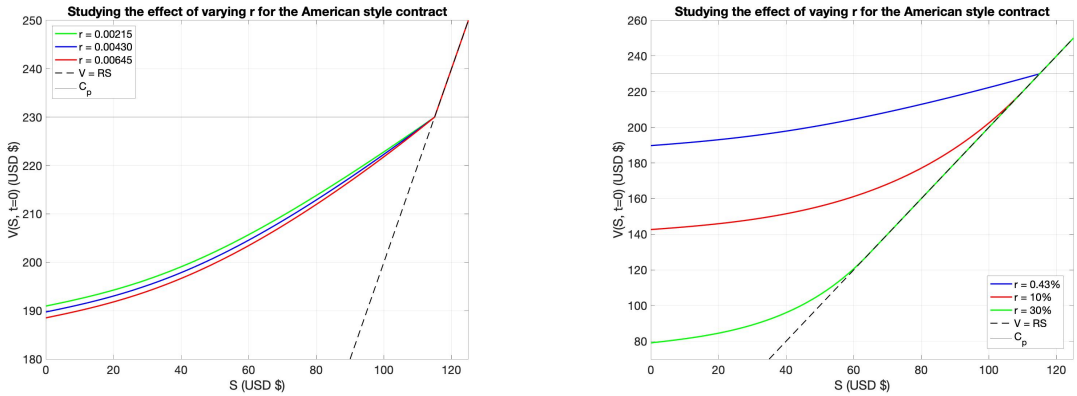


Figure 5: Value at $t = 0$ of the American style contract with an embedded call option with $C_p = 230,\ t_0 = 1.2654$.

contract pricing, since $V = RS = C_p$ in either case.

Furthermore, we can study how the risk-free interest rate r affects the contract value. Figure 6 (a) shows plots for the contract value at $t = 0$ at three different interest rates $r = \{0.00215,\ 0.0043,\ 0.00645\}$. We can observe that, as the interest rate increases, the value of the contract falls at low S, until eventually all three curves meet at $RS = C_p$ where the contract ceases to exist.



(a) Plot of $V(S, t = 0)$ at $r = \{0.00215,\ 0.0043,\ 0.00645\}$. (b) Plot of $V(S, t = 0)$ at $r = \{0.0045,\ 0.1,\ 0.3\}$.

Figure 6: Studying the effects of varying r on the contract value.

Let us focus on the value at $S = 0$ where the discrepancy between the three curves is largest. In this case, the contract will never be converted nor called, so the only things contributing its value are the principle F and the coupon $K(t = 0) = C$. Since $F >> C$, in the risk neutral measure we can estimate the contract value to be the discounted expected value of F, plus a small (but positive) contribution due to the coupon payment. Since the coupon payment is independent of r, we can conclude that contract value at $S = 0$ gets larger whenever the discount factor (and consequently, the interest rate) gets smaller. Therefore, all three curves start at different positions, at $S = 0$, but they must all end up at $RS = C_p$ where the contract can no longer exist. This means that the contract with the highest interest rate must grow slightly faster with S than that with the lowest, to compensate for the slower start.

If instead we consider the more radical cases $r = 0.1$ and $r = 0.3$, shown on figure 6(b) by the red and green lines respectively, the same observation can be made. At $S = 0$, the contracts' values are approximately $Fe^{-0.1\times3} \approx 141$ and $Fe^{-0.3\times3} \approx 77$ plus small corrections due to coupon. In order to ar-

rive at $RS = C_p$ the contract with the higher interest rates must converge a lot faster to the limit $V = RS$.

Finally, we wish to estimate the most precise value at $S_0 = 95.24$ obtainable within 1 second of computational time. Due to the free boundary nature of the problem and the increased amount of non linearity errors, attempting to use extrapolation will be ineffective, since achieving a steady convergence is very complicated. Discontinuities occur at $S = F/R = 95$ and $S = C_p/R = 115$ at $t = T$, and along the lines of constant $t = t_0 = 1.2654$ and boundary errors at $V = RS$. Therefore, an efficient way to align the grid points with the discontinuities must be found.

First, one can note that $95 = 19 \times 5$ and $115 = 23 \times 5$, so both share a common factor of 5. Therefore, choosing $dS = 5/n,\ \forall n \in \mathbb{N}$ will guarantee that the two points are included in the grid (occurring at $j = 19n$ and $j = 23n$ respectively). If we choose for example $S^{max} = 4 \times F/R$, then $j_{max} = 76n,\ n \in \mathbb{N}$ gives the desired dS. Table 5 shows the results obtained for $i_{max} = j_{max} = 76n,\ n = \{2, 4, 6, ..., 12\}$. The differences with the preceding value between values appear to converge towards zero at a steady rate. For a computational time of 1 second, we can estimate the value to 5 significant figures at $V(S = 95.24, t = 0) = 219.98$.

$i_{max} = j_{max}$	$V(95.24)$	| Diff | $(\times 10^{-4})$	Time Elapsed (s)
76	219.952	-	0.012
228	219.979	263	0.176
380	219.982	35.8	0.66
532	219.984	15.7	1.72
684	219.985	8.50	3.51
836	219.985	5.24	6.13

Table 5: Unbunched time gridpoints.

$i_{max} = j_{max}$	$V(95.24)$	| Diff | $(\times 10^{-5})$	Time Elapsed (s)
76	219.968	-	0.011
304	219.9822	146	0.098
532	219.9825	26.7	0.26
760	219.9827	19.2	0.475
988	219.9828	7.16	0.846
1216	219.9828	1.94	1.193
1444	219.9828	4.77	1.674
1672	219.9828	0.74	2.248
1900	219.9828	0.61	2.849
2128	219.9828	0.31	3.548

Table 6: Bunched time gridpoints.

One possible method to minimise the non-linearity errors at $t = t_0$ is by using a non-constant grid spacing. In particular, by bunching more grid points near this discontinuity. To do so, whenever t_0 lies between two grid points, i.e. whenever: $(i - \frac{1}{2})dt \leq t_0 \leq (i + \frac{1}{2})dt$ the grid spacing dt was reduced by a factor of 100. For example, if we consider $dt = 0.05$ for $t_0 = 1.2654$, 100 additional gridpoints were placed between 1.225 and 1.275. This will ensure that the values at $t = t_0$ and its vicinity are included in the grid. Figure 7 shows how the temporal domain of the grid is discretised for the bunched and unbunched cases.

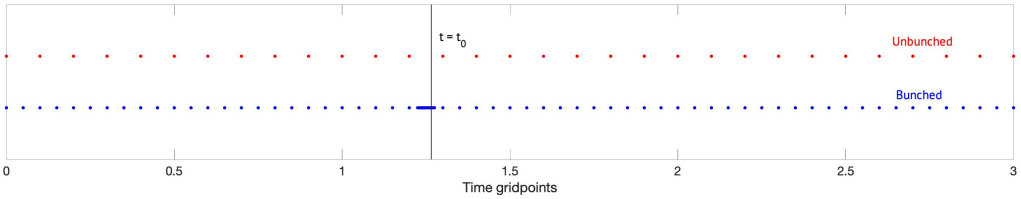


Figure 7: Discretisation of the time domain for $dt = 0.05$, and bunching applied close to $t = t_0 = 1.2654$.

The results of using bunched coordinates is given in table 6. Not only does the penalty method converge significantly faster for larger i_{max}, j_{max} pairs, but also the differences between the values fall at a faster rate. Except for an outlier at $i_{max} = 1216$, the differences in values converge smoothly towards zero. This time, in the computational limit of 1 second, we can obtain a 7 significant figure estimate: $V(S = 95.24, t = 0) = 219.9828$.

7 References:

[1] Jordan, C., Calculus Of Finite Differences, By Charles Jordan. Introd. By Harry C. Carver. 3rd ed. New York: Chelsea Pub. Co., (C1965).

[2] Johnson, P., Lecture 6: *Finite-difference methods*, lecture notes, MATH40082 Computational Finance, University of Manchester, delivered March 2020.

[3] Johnson, P., Lecture 7: *Extensions to finite-difference methods*, lecture notes, MATH40082 Computational Finance, University of Manchester, delivered March 2020.

[4] Yuen, K., Yang, H. and Chu, K., 2001. Estimation in the Constant Elasticity of Variance Model. British Actuarial Journal, [online] 7(2), pp.119-120. Available at: https://www.actuaires.org/AFIR/Colloquia/Tokyo/Chu_Yang_Yuen.pdf [Accessed 18 April 2020].

[5] Linetsky, V. and Mendoza, R., 2009. The Constant Elasticity of Variance Model. [online] Available at: https://pdfs.semanticscholar.org/6017/9494f6c8288e004308b15aad83fd9cc72ea7.pdf [Accessed 20 April 2020].

8 Appendices:

8.1 European style convertible bond:

```
#include <vector>
#include <cassert>
#include <fstream>

template <class T>
T max(T n1, T n2)
{
    return (n1 > n2) ? n1 : n2;
}

std::vector<double> LU_decompose (std::vector<double> a, std::vector<double> b,
                                  std::vector<double> c, std::vector<double> d_i
    )
{
    assert(d_i.size() == c.size() && c.size() == b.size() && b.size() == a.size
    () );
    size_t size = d_i.size();

    double b_0 = b[0];

    std::vector<double> beta(size);
    beta[0] = b_0;
    for (int j{1}; j< size; j++) beta[j] = b[j] - (a[j]*c[j-1])/beta[j-1];
    double beta_max = beta[size-1];

    std::vector<double> D(size);
    D[0] = d_i[0];
    for (int j{1}; j< size; j++) D[j] = d_i[j] - a[j]/beta[j-1] * D[j-1];
    double D_max = D[size-1];

    std::vector<double> V_i(size);
    V_i[size-1] = D_max/beta_max;
    for (int j=size-2; j>=0; j--) V_i[j] = 1/beta[j]*(D[j]-c[j]*V_i[j+1]);

    return V_i;
}


```

```
37 // A generic lagrange interpolation function
38 double lagrangeInterpolation(const std::vector<double>& y,const std::vector<
       double>& x,double x0,unsigned int n)
39 {
40     if(x.size()<n)return lagrangeInterpolation(y,x,x0,x.size());
41     if(n==0)throw;
42     int nHalf = n/2;
43     int jStar;
44     double dx=x[1]-x[0];
45     if(n%2==0)
46         jStar = int((x0 - x[0])/dx) -(nHalf-1);
47     else
48         jStar = int((x0 - x[0])/dx+0.5)-(nHalf);
49     jStar=std::max(0,jStar);
50     jStar=std::min(int(x.size()-n),jStar);
51     if(n==1)return y[jStar];
52     double temp = 0.;
53     for(unsigned int i=jStar;i<jStar+n;i++){
54         double  int_temp;
55         int_temp = y[i];
56         for(unsigned int j=jStar;j<jStar+n;j++){
57             if(j==i){continue;}
58             int_temp *= ( x0 - x[j] )/( x[i] - x[j] );
59         }
60         temp += int_temp;
61     }
62     // end of interpolate
63     return temp;
64 }
65
66 // CONVERTIBLE BOND:
67 // Define parameters globally:
68 double T = 3., F = 190., R = 2., r = 0.0043, K = 0.08333333, mu = 0.0076, X =
       95.24, C = 0.408, alpha = 0.03, beta = 0.142, sigma = 17.1;
69
70 // Functions for convertible bond:
71 double theta(double t) {return (1+mu)*X*exp(mu*t);}
72
73 // Functions for boundary at S -> inf:
74 double A(double t)
75 {
76     return R*exp((K+r)*(t-T));
77 }
78 double B(double t)
79 {
80     return C/(alpha+r)*(exp(-alpha*t)-exp(-alpha*T+r*(t-T))) + X*R*(exp(r*(t-T))
       - exp((K+r)*(t-T)));
81 }
82
83 // Numerical scheme coefficients:
84 double get_a(int j, int i, double dS, double dt)
85 {
86     return 0.25*(sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1))-K*(theta(i*dt)/
       dS-j));
87 }
88
89 double get_b(int j, int i, double dS, double dt)
90 {
91     return -0.5*sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1)) - 0.5*r - 1./dt;
92 }
93
94 double get_c(int j, int i, double dS, double dt)
95 {
```

```
96      return 0.25*(sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1))+K*(theta(i*dt)/
        dS-j));
97  }
98
99  double get_d(int j, int i, double dS, double dt, std::vector<double> vOld)
100 {
101     return -get_a(j, i, dS, dt)*vOld[j-1] - (-0.5*sigma*sigma*pow(j, 2*beta)*pow
        (dS, 2*(beta-1)) - 0.5*r + 1./dt)*vOld[j] - get_c(j,i,dS, dt)*vOld[j+1] - C*
        exp(-alpha*i*dt);
102 }
103
104 // Implement Crank-Nicolson method for the convertible bond:
105 double Crank_Nicolson_Bond(double S_want, int iMax, int jMax, double S_max)
106 {
107     // Declare and Initialise local variables: dS,dt, Smax:
108     // double S_max = 5*F/R;
109     double dS = S_max/jMax;
110     double dt = T/iMax ;
111     // std::cout << "dS: " << dS << " dt: " << dt << std::endl;
112
113     // First appply terminal condition V(S,t = T):
114     std::vector<double> vNew(jMax+1), S(jMax+1);
115     for (int j{}; j<=jMax; j++) S[j] = j*dS;
116     for (int j{}; j<=jMax; j++) vNew[j] = max(F, R*S[j]);
117     std::vector<double> vOld = vNew;
118
119     for (int i = iMax-1; i >= 0; i--)
120     {
121         std::vector<double> a(jMax+1), b(jMax+1), c(jMax+1), d(jMax+1);
122
123         // Set up boundary condition at S = 0:
124         a[0] = 0;
125         b[0] = -(1./dt+K*theta(i*dt)/dS+r/2);
126         c[0] = K*theta(i*dt)/dS;
127         d[0] = (-1./dt+r/2)*vOld[0]-C*exp(-alpha*i*dt);
128
129         for (int j{1}; j<jMax; j++)
130         {
131             a[j] = get_a(j, i, dS, dt);
132             b[j] = get_b(j, i, dS, dt);
133             c[j] = get_c(j, i, dS, dt);
134             d[j] = get_d(j, i, dS, dt, vOld);
135         }
136
137         // Set up boundary condition at S -> inf.
138         a[jMax] = 0;
139         b[jMax] = 1;
140         c[jMax] = 0;
141         d[jMax] = (jMax*dS)*A(i*dt)+B(i*dt);
142
143         // Call solver:
144         vNew = LU_decompose(a, b, c, d);
145         vOld = vNew;
146     }
147     return lagrangeInterpolation(vNew,S,S_want,4);;
148 }
149
150 double N(double x){return 0.5*erfc(-x/sqrt(2));}
151
152 double CallOption(double St, double X, double T, double r, double D, double
    sigma, double t)
153 {
154     double d1 = (log(St/X)+(r-D+0.5*sigma*sigma)*(T-t) )/ (sigma*sqrt(T-t));
```

```
155     double d2 = d1-sigma*sqrt(T-t);
156     return St*exp(-D*(T-t)) * N(d1) - X * exp(-r*(T-t))*N(d2);
157 }
158
159 void analytic_estimate()
160 {
161     double est{};
162     std::ofstream DataFile;
163     DataFile.open("Values_an.csv");
164     DataFile.width(20);  DataFile << "S";
165     DataFile.width(20); DataFile << "Value" << std::endl;
166
167     for (int S = 0; S<200; S+=20)
168     {
169         double call =  CallOption(S, F/R, T, r, r, sigma, 0);
170         est = R*call+F*exp(-r*T)+C/(alpha+r)*(1-exp(-(alpha+r)*T));
171         DataFile.width(20);  DataFile << S;
172         DataFile.width(20);  DataFile << est << std::endl;
173     }
174     std::cout << "Succesfully saved to file." << std::endl;
175 }
176
177 void get_abs_err()
178 {
179     double call =  CallOption(25, F/R, T, r, r, sigma, 0);
180     double est = R*call+F*exp(-r*T)+C/(alpha+r)*(1-exp(-(alpha+r)*T));
181
182     std::ofstream DataFile;
183     DataFile.open("Abs_Err_table.csv");
184     DataFile.width(20);  DataFile << "iMax";
185     DataFile.width(20);  DataFile << "jMax";
186     DataFile.width(20);  DataFile << "dt";
187     DataFile.width(20);  DataFile << "dS";
188     DataFile.width(20);  DataFile << "Value_50";
189     DataFile.width(20); DataFile << "Abs_Err" << std::endl;
190
191     for (int n{1}; n <= 30; n++)
192     {
193         double Smax = 5*F/R;
194         int jmax = Smax/25*n;
195         double vNumerical = Crank_Nicolson_Bond(25, 200, jmax, Smax);
196         double dS =  Smax/jmax;
197         int j_want = 25/dS;
198
199         if (j_want*dS == 25)
200         {
201             DataFile.width(20);  DataFile << jmax;
202             DataFile.width(20);  DataFile << jmax;
203             DataFile.width(20);  DataFile << 200;
204             DataFile.width(20);  DataFile << dS;
205
206             double num = vNumerical;
207             DataFile.width(20);  DataFile << num;
208             double abs_err = fabs(est-num);
209             DataFile.width(20);  DataFile << abs_err << std::endl;
210         }
211     }
212     std::cout << "Succefully saved to file." << std::endl;
213 }
214
215 void get_convergange_rate()
216 {
217     int k = 2;
```

```
218
219     double vOld{1}, diffOld{1};
220     std::cout << "iMax=jMax" << "     " << "vNew" << "      " << "      Extrap    "
        << "   diffNew    "  << "     " << "    R" << "         " << "      c" << "    "
         << "    Duration(s)     " << std::endl;
221     for (int jmax{5}; jmax<3000; jmax*= k)
222     {
223         auto start = std::chrono::high_resolution_clock::now();
224         double vNew = Crank_Nicolson_Bond(95.24, jmax, jmax, 4*F/R);
225         auto stop = std::chrono::high_resolution_clock::now();
226         auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        stop - start);
227
228         double diffNew = vNew - vOld;
229         double R = diffOld/diffNew;
230         double c = log(R)/log(k);
231         double extrapValue = (4*vNew - vOld)/3.;
232         std::cout << jmax << "         " << vNew << "    " << extrapValue << "   "
        << diffNew  << "    " << R << "    " << c << "    " << duration.count()/1000.
        << std::endl;
233
234         vOld = vNew;
235         diffOld = diffNew;
236     }
237 }
238
239 void get_convergange_rate_extrap()
240 {
241     int k = 2;
242
243     double vOld{1}, diffOld{1}, extrap_old{1};
244
245     std::cout << "iMax=jMax    vNew     Diff    Extrap    Diff_Extrap    Rat    "
         << std::endl;
246
247     for (int jmax{20}; jmax<3000; jmax*= k)
248     {
249         auto start = std::chrono::high_resolution_clock::now();
250         double vNew = Crank_Nicolson_Bond(45 , jmax, jmax, 5*F/R);
251         auto stop = std::chrono::high_resolution_clock::now();
252         auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        stop - start);
253
254         double diffNew = vNew - vOld;
255         double R = diffOld/diffNew;
256         double extrapValue = (4*vNew - vOld)/3.;
257         double diffExtrap = extrapValue-extrap_old;
258
259         std::cout << jmax << "         " << vNew << "     " << diffNew << "    " <<
          extrapValue << "   " << diffExtrap  << "      " << duration.count()/1000 <<
        std::endl;
260
261         vOld = vNew;
262         diffOld = diffNew;
263         extrap_old = extrapValue;
264     }
265 }
266
267
268 void testing_Smax()
269 {
270     double diff{1}; double vOld{1};
271     std::cout << "n    Smax    iMax    jMax    V(45)    Diff    Duration(s)" << std
```

```
    ::endl;
272     for (int n{2}; n<=10; n++)
273     {
274         double S_max = n*F/R;
275         int jMax = S_max/0.5;
276 
277         auto start = std::chrono::high_resolution_clock::now();
278         double vNew = Crank_Nicolson_Bond(45, 100, jMax, S_max);
279         auto stop = std::chrono::high_resolution_clock::now();
280         auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
    stop - start);
281 
282         diff = vNew - vOld;
283 
284         std::cout << n << "     " << S_max << "     " << 100 << "     " << jMax <<
    "     " << vNew << "   " << diff << "    " << duration.count()/1000. << std::
    endl;
285 
286         vOld = vNew;
287     }
288 }
```

8.2 American style convertible bond:

```
1  #include <iostream>
2  #include <iostream>
3  #include <iomanip>
4  #include <cmath>
5  #include <chrono>
6  #include <vector>
7  #include <cassert>
8  #include <fstream>
9 
10 // Define parameters globally:
11 double T = 3., F = 190., R = 2., r = 0.0043, K = 0.08333333333, mu = 0.0076, X =
       95.24, C = 0.408, alpha = 0.03, beta = 0.142, sigma = 17.1, t_0 = 1.2654,
       Cp = 230;
12 
13 template <class T>
14 T max(T n1, T n2)
15 {
16     return (n1 > n2) ? n1 : n2;
17 }
18 
19 std::vector<double> bunch_time_points(double t_0, double iMax)
20 {
21     std::vector<double> time_grid_points;
22     double dt = T/iMax;
23     for (int i{}; i<=iMax; i++)
24     {
25         if ( t_0 >= (i-0.5)*dt  && t_0 <=  (i+0.5)*dt)
26         {
27             for (int j=0; j<=100; j++)
28             {
29                 time_grid_points.push_back((i-0.5+j/100.)*dt);
30             }
31 
32         }
33         else time_grid_points.push_back(i*dt);
34     }
35     return time_grid_points;
36 }
37 
38 std::vector<double> LU_decompose (std::vector<double> a, std::vector<double> b,
```

```
39                                      std::vector<double> c, std::vector<double> d_i
    )
40 {
41     assert(d_i.size() == c.size() && c.size() == b.size() && b.size() == a.size
    () );
42     size_t size = d_i.size();
43 
44     double b_0 = b[0];
45 
46     std::vector<double> beta(size);
47     beta[0] = b_0;
48     for (int j{1}; j< size; j++) beta[j] = b[j] - (a[j]*c[j-1])/beta[j-1];
49     double beta_max = beta[size-1];
50 
51     std::vector<double> D(size);
52     D[0] = d_i[0];
53     for (int j{1}; j< size; j++) D[j] = d_i[j] - a[j]/beta[j-1] * D[j-1];
54     double D_max = D[size-1];
55 
56     std::vector<double> V_i(size);
57     V_i[size-1] = D_max/beta_max;
58     for (int j=size-2; j>=0; j--) V_i[j] = 1/beta[j]*(D[j]-c[j]*V_i[j+1]);
59 
60     return V_i;
61 }
62 
63 // A generic lagrange interpolation function
64 double lagrangeInterpolation(const std::vector<double>& y,const std::vector<
    double>& x,double x0,unsigned int n)
65 {
66     if(x.size()<n) return lagrangeInterpolation(y,x,x0,x.size());
67     if(n==0) throw;
68     int nHalf = n/2;
69     int jStar;
70     double dx=x[1]-x[0];
71     if(n%2==0)
72         jStar = int((x0 - x[0])/dx) -(nHalf-1);
73     else
74         jStar = int((x0 - x[0])/dx+0.5)-(nHalf);
75     jStar=std::max(0,jStar);
76     jStar=std::min(int(x.size()-n),jStar);
77     if(n==1)return y[jStar];
78     double temp = 0.;
79     for(unsigned int i=jStar;i<jStar+n;i++){
80         double  int_temp;
81         int_temp = y[i];
82         for(unsigned int j=jStar;j<jStar+n;j++){
83             if(j==i){continue;}
84             int_temp *= ( x0 - x[j] )/( x[i] - x[j] );
85         }
86         temp += int_temp;
87     }
88     // end of interpolate
89     return temp;
90 }
91 
92 // CONVERTIBLE BOND:
93 
94 // Functions for convertible bond:
95 double theta(double t) {return (1+mu)*X*exp(mu*t);}
96 
97 // Functions for boundary at S -> inf:
98 double A(double t)
```

```
99  {
100     return R*exp((K+r)*(t-T));
101 }
102 double B(double t)
103 {
104     return C/(alpha+r)*(exp(-alpha*t)-exp(-alpha*T+r*(t-T))) + X*R*(exp(r*(t-T))
        - exp((K+r)*(t-T)));
105 }
106
107 // Numerical scheme coefficients:
108 double get_a(int j, int i, double dS, double dt)
109 {
110         return 0.25*(sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1))-K*(theta(i*
        dt)/dS-j));
111 }
112
113 double get_b(int j, int i, double dS, double dt)
114 {
115     return -0.5*sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1)) - 0.5*r - 1./dt;
116 }
117
118 double get_c(int j, int i, double dS, double dt)
119 {
120     return 0.25*(sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1))+K*(theta(i*dt)/
        dS-j));
121 }
122
123 double get_d(int j, int i, double dS, double dt, std::vector<double> vOld)
124 {
125     return -get_a(j, i, dS, dt)*vOld[j-1] - (-0.5*sigma*sigma*pow(j, 2*beta)*pow
        (dS, 2*(beta-1)) - 0.5*r + 1./dt)*vOld[j] - get_c(j,i,dS, dt)*vOld[j+1] - C*
        exp(-alpha*i*dt);
126 }
127
128 // Implement Crank-Nicolson method for the convertible bond:
129 double Crank_Nicolson_American_Bond(double S_want, int iMax, int jMax, double
        S_max)
130 {
131     // Declare and Initialise local variables: dS,dt, Smax:
132     double dS = S_max/jMax;
133     double dt = T/iMax ;
134     // std::cout << "dS: " << dS << " dt: " << dt << std::endl;
135
136     // First appply terminal condition V(S,t = T):
137     std::vector<double> vNew(jMax+1), S(jMax+1);
138     for (int j{}; j<=jMax; j++) S[j] = j*dS;
139     for (int j{}; j<=jMax; j++) vNew[j] = max(F, R*S[j]);
140     std::vector<double> vOld = vNew;
141
142     for (int i = iMax-1; i >= 0; i--)
143     {
144         std::vector<double> a(jMax+1), b(jMax+1), c(jMax+1), d(jMax+1);
145
146         // Set up boundary condition at S = 0:
147         a[0] = 0;
148         b[0] = -(1./dt+K*theta(i*dt)/dS+r/2);
149         c[0] = K*theta(i*dt)/dS;
150         d[0] = (-1./dt+r/2)*vOld[0]-C*exp(-alpha*i*dt);
151
152         for (int j{1}; j<jMax; j++)
153         {
154             a[j] = get_a(j, i, dS, dt);
155             b[j] = get_b(j, i, dS, dt);
```

```
156             c[j] = get_c(j, i, dS, dt);
157             d[j] = get_d(j, i, dS, dt, vOld);
158         }
159 
160         // Set up boundary condition at S -> inf.
161         a[jMax] = 0;
162         b[jMax] = 1;
163         c[jMax] = 0;
164         d[jMax] = R*jMax*dS;
165 
166         // Apply penalty method:
167         double penalty=1.e8, tolerance = 1.e-8; int iterMax = 100, penaltyIt =
    0;
168         for(int penaltyIt=0; penaltyIt< iterMax; penaltyIt++)
169         {
170             std::vector<double> bHat(b),dHat(d);
171             for(int j=1;j<jMax;j++)
172             {
173                 // turn on penalty if V < RS for conversion:
174                 if(vNew[j] < S[j]*R)
175                 {
176                     bHat[j] = b[j] - penalty;
177                     dHat[j] = d[j] - penalty*S[j]*R;
178                 }
179 
180                 // If t < t_0, turn on penalty if V > max(Cp, RS) for call
    Option:
181                 if (i*dt < t_0 && vNew[j] > max(Cp, R*S[j]) )
182                 {
183                   bHat[j] = b[j] - penalty;
184                   dHat[j] = d[j] - penalty*max(Cp, R*S[j]);
185                 }
186 
187             }
188             // solve matrix equations with LU decompomposition:
189             std::vector<double> y = LU_decompose(a,bHat,c,dHat);
190 
191             // calculate difference from last time
192             double error=0.;
193             for(int j=0;j<=jMax;j++) error += fabs(vNew[j] - y[j]);
194             vNew = y;
195             if(error < tolerance) break;
196         }
197         if (penaltyIt >= iterMax)
198             {std::cout << "Failed to converge after " << iterMax << " iterations
    . " << '\n';}
199         vOld = vNew;
200     }
201     double value = lagrangeInterpolation(vNew,S,S_want,4);
202     // std::cout << "V(S=" << S_want << ", t=0) = " << value << std::endl;
203     return value;
204 }
205 
206 
207 // Implement Crank-Nicolson method for the convertible bond:
208 double Crank_Nicolson_American_Bond_bunched(double S_want, int iMax, int jMax,
    double S_max)
209 {
210     // Declare and Initialise local variables: dS,dt, Smax:
211     double dS = S_max/jMax;
212     std::vector<double> time_grid_points = bunch_time_points(t_0, iMax);
213     double iMax_star = time_grid_points.size()-1;
214     // std::cout << "dS: " << dS << " dt: " << dt << std::endl;
```

```
215
216     // First appply terminal condition V(S,t = T):
217     std::vector<double> vNew(jMax+1), S(jMax+1);
218     for (int j{}; j<=jMax; j++) S[j] = j*dS;
219     for (int j{}; j<=jMax; j++) vNew[j] = max(F, R*S[j]);
220     std::vector<double> vOld = vNew;
221
222
223     for (int k = iMax_star-1; k >= 0; k--)
224     {
225         double t = time_grid_points[k];
226         double dt = time_grid_points[k+1] - time_grid_points[k];
227         // std::cout << "t: " << t << " dt: " << dt << std::endl;
228
229         std::vector<double> a(jMax+1), b(jMax+1), c(jMax+1), d(jMax+1);
230
231         // Set up boundary condition at S = 0:
232         a[0] = 0;
233         b[0] = -(1./dt+K*theta(t)/dS+r/2);
234         c[0] = K*theta(t)/dS;
235         d[0] = (-1./dt+r/2)*vOld[0]-C*exp(-alpha*t);
236
237         for (int j{1}; j<jMax; j++)
238         {
239             a[j] = 0.25*(sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1))-K*(theta
    (t)/dS-j));
240             b[j] = -0.5*sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1)) - 0.5*r -
     1./dt;
241             c[j] = 0.25*(sigma*sigma*pow(j, 2*beta)*pow(dS, 2*(beta-1))+K*(theta
    (t)/dS-j));
242             d[j] = -a[j]*vOld[j-1] - (-0.5*sigma*sigma*pow(j, 2*beta)*pow(dS,
    2*(beta-1)) - 0.5*r +                          1./dt)*vOld[j] - c[j]*vOld[j
    +1] - C*exp(-alpha*t);
243         }
244
245         // Set up boundary condition at S -> inf.
246         a[jMax] = 0;
247         b[jMax] = 1;
248         c[jMax] = 0;
249         d[jMax] = R*jMax*dS;
250
251         // Apply penalty method:
252         double penalty=1.e8, tolerance = 1.e-8; int iterMax = 100, penaltyIt =
    0;
253         for(int penaltyIt=0; penaltyIt< iterMax; penaltyIt++)
254         {
255             std::vector<double> bHat(b),dHat(d);
256             for(int j=1;j<jMax;j++)
257             {
258                 // turn on penalty if V < RS for conversion:
259                 if(vNew[j] < S[j]*R)
260                 {
261                     bHat[j] = b[j] - penalty;
262                     dHat[j] = d[j] - penalty*S[j]*R;
263                 }
264
265                 // If t < t_0, turn on penalty if V > max(Cp, RS) for call
    Option:
266                 if (t < t_0 && vNew[j] > max(Cp, R*S[j]) )
267                 {
268                   bHat[j] = b[j] - penalty;
269                   dHat[j] = d[j] - penalty*max(Cp, R*S[j]);
270                 }
```

```
271
272             }
273             // solve matrix equations with LU decompomposition:
274             std::vector<double> y = LU_decompose(a,bHat,c,dHat);
275
276             // calculate difference from last time
277             double error=0.;
278             for(int j=0;j<=jMax;j++) error += fabs(vNew[j] - y[j]);
279             vNew = y;
280             if(error < tolerance) break;
281         }
282         if (penaltyIt >= iterMax)
283             {std::cout << "Failed to converge after " << iterMax << " iterations
    . " << '\n';}
284         vOld = vNew;
285     }
286     double value = lagrangeInterpolation(vNew,S,S_want,4);
287     // std::cout << "V(S=" << S_want << ", t=0) = " << value << std::endl;
288     return value;
289 }
290
291 void results_table()
292 {
293     std::cout << "iMax jMax        V           Diff          Time   " << std::endl;
294
295     double vOld{1};
296     for (int n{1}; n<=30; n+=3)
297     {
298         int iMax = n*76, jMax = n*76;
299         auto start = std::chrono::high_resolution_clock::now();
300         double vNew = Crank_Nicolson_American_Bond_bunched(95.24, iMax, jMax, 4*
    F/R);
301         auto stop = std::chrono::high_resolution_clock::now();
302         auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
    stop - start);
303
304         double diff = vNew - vOld;
305
306         std::cout << iMax << "    " << jMax << "     " << vNew << "    " << fabs(
    diff) << "     " << duration.count()/1000. << std::endl;
307         vOld = vNew;
308     }
309 }
```