Assignment 2: Advanced Methods

10134621

School of Mathematics

The University of Manchester

Fourth Year
Computational Finance

May 2021

Abstract

In this assignment we look to use Finite Difference methods in order to price various complex options. We begin by attempting to value a convertible bond that pays coupons using a Crank-Nicolson Finite Difference scheme. How different model parameters effect the solution will be explored and we will ultimately look to obtain the most accurate value of the option using higher order interpolations and extrapolation. In the later section of the report we complicate the option further by embedding an American call option within. We again use a Crank-Nicolson scheme but now with the penalty method to value this option which offers accurate results with high efficiency. Again, how various parameters effect our results will be explored and at the end we will attempt to get the most accurate value of our option within a given time limit.

1. European style convertible Bonds

In this first section we look to price an option where the holder can choose between a principle F or an amount R of the underlying S at $t = T$. They also receive a continuous coupon throughout holding.

1.1. Boundary Conditions

The value of our option, $V(S, t)$, satisfies the following PDE

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 V}{\partial S^2} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0, \tag{1.1.1}$$

where

$$\theta(t) = (1 + \mu)Xe^{\mu t}. \tag{1.1.2}$$

In the limit $S \to \infty$, we purpose the ansatz

$$V(S, t) = SA(t) + B(t). \tag{1.1.3}$$

Substituting equation (1.1.3) into (1.1.1) gives two further ODEs that can be solved to ultimately give

$$V(S \to \infty, t) = R(S - X)e^{-(\kappa+r)(T-t)} + \left(XR - \frac{C}{\alpha + r}e^{-\alpha t}\right)e^{-r(T-t)} + \frac{C}{\alpha + r}e^{-\alpha t}. \tag{1.1.4}$$

For $t = 0$, equation (1.1.4) is evaluated to be

$$V(S \to \infty, 0) = R(S - X)e^{-(\kappa+r)T} + \frac{C}{\alpha + r} + (XR - \frac{C}{\alpha + r})e^{-rT}. \tag{1.1.5}$$

In the limit $S \to 0$, $V(S, t)$ satisfies a simplified PDE of the form

$$\frac{\partial V}{\partial t} + \kappa\theta(t)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t} = 0. \tag{1.1.6}$$

1.2. Crank-Nicolson finite difference

Using the Crank-Nicolson finite difference method we obtain the expressions for the matrix coefficients. At $j = 0$,

$$\begin{aligned}
a_0 &= 0,\\
b_0 &= \frac{-1}{\Delta t} - \frac{\kappa\theta(i\Delta t)}{\Delta S} - \frac{r}{2},\\
c_0 &= \frac{\kappa\theta(i\Delta t)}{\Delta S},\\
d_0 &= \left(\frac{-1}{\Delta t} + \frac{r}{2}\right)V_0^{i+1} - Ce^{-i\Delta t}.
\end{aligned}$$

For general j,

$$\begin{aligned}
a_j &= \frac{-1}{4}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} + \frac{\kappa}{4\Delta S}(\theta(i\Delta t) - j\Delta S), \\
b_j &= \frac{1}{\Delta t} + \frac{1}{2}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} + \frac{r}{2}, \\
c_j &= \frac{-1}{4}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} - \frac{\kappa}{4\Delta S}(\theta(i\Delta t) - j\Delta S), \\
d_j &= \left[\frac{1}{4}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} / \frac{\kappa}{4\Delta S}(\theta(i\Delta t) - j\Delta S)\right] V_{j-1}^{i+1} \\
&\quad + \left[\frac{1}{\Delta t} - \frac{1}{2}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} - \frac{r}{2}\right] V_j^{i+1} \\
&\quad + \left[\frac{1}{4}\sigma^2 j^{2\beta}(\Delta S)^{2(\beta-1)} + \frac{\kappa}{4\Delta S}(\theta(i\Delta t) - \Delta S)\right] V_{j+1}^{i+1} + Ce^{-\alpha i\Delta t}.
\end{aligned}$$

Finally for $j = j_{max}$,

$$\begin{aligned}
a_{j_{max}} &= 0, \\
b_{j_{max}} &= 1, \\
c_{j_{max}} &= 0, \\
d_{j_{max}} &= R(S - X)e^{-(\kappa+r)(T-i\Delta t)} + \frac{C}{\alpha + r}e^{-\alpha i\Delta t} + \left(XR - \frac{C}{\alpha + r}e^{-\alpha T}\right) e^{-r(T-i\Delta t)}.
\end{aligned}$$

The implementation of these equations in the code is shown in Listing (1).

1.3. Exploring the model parameters

We choose the following model paramters as standard: $T = 2$, $F = 50$, $R = 1$, $r = 0.0114$, $\kappa = 0.125$, $\mu = 0.0174$, $X = 50.5$, $C = 0.285$ and $\alpha = 0.01$.

We consider two separate cases. The first of which we take $\beta = 1$ and $\sigma = 0.4$, and a plot of the option value $V(S, t = 0)$ is shown as a function of the underlying in the left hand plot of Figure (1). For the second case we take $\beta = 0.869$ and $\sigma = 0.668$; its corresponding plot is shown on the right of Figure (1). In both of these cases we used a Thomas solver as direct solvers are generally more efficient for these calculations and don't require balancing of several parameters, the code for which is shown in Listing (2). An SOR solver, which provides identical results in this case, was also implemented and is shown in Listing (3).

These plots appear almost identical, so to highlight the difference better we can plot the absolute difference between the values over this range, which is shown in Figure (2). To explain this shape consider that in equation (1.1.1), σ and β appear as part of the combination $\sigma^2 S^{2\beta}$ which is the coefficient of the $\frac{\partial^2 V}{\partial S^2}$ term. This second order derivative term has the largest effect when the changes in the gradient of $V(S, t)$ are the largest. From figure (1) we see that the changes in gradient occur at approximately $S \approx 25$ and $S \approx 100$. So we expect that the option value around these points will experience the greatest change when we alter the values of σ and β. Additionally, for the value $S \approx 50$, these choices of β and σ produce the same value of $\sigma^2 S^{2\beta}$, so we expect no difference at this value. We see exactly these properties in Figure (2).

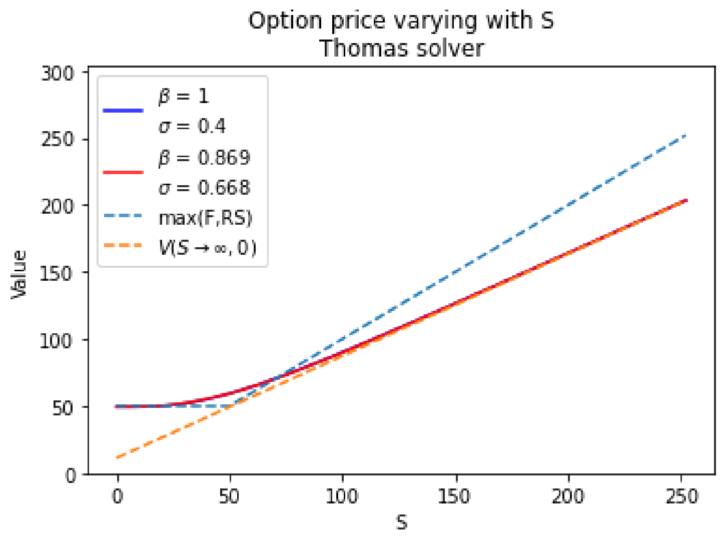

Figure 1: An image showing the value of the option varying with the underlying price S. The two solid lines show the value of the option at two different pairwise values of β and σ. The dashed blue line shows the terminal condition of the option and the dashed orange line shows the option value at $t = 0$ in the limit $S \to \infty$.

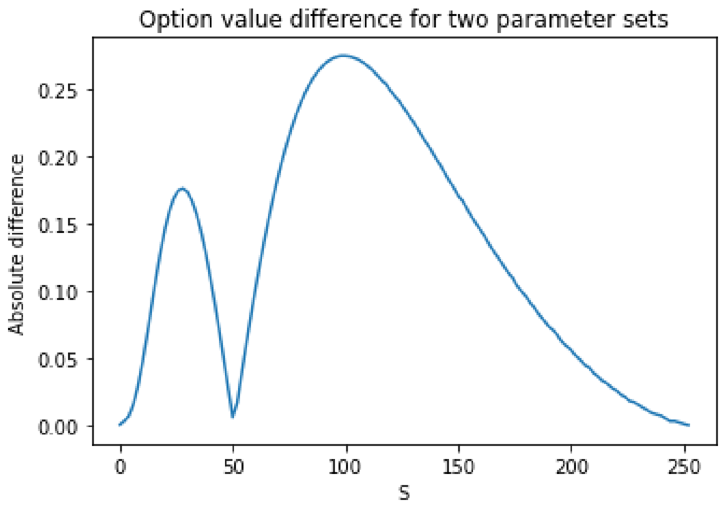

Figure 2: An image showing the value of the absolute difference in the value of an option evaluated at two different pairwise parameter sets.

1.4. Optimal option value

Now we take $S_0 = 50.5$, $\sigma = 0.668$ and $\beta = 0.869$ and attempt to get an accurate and efficient estimate of the option value.

We will use a generic Lagrange interpolation of order 8, the code for which is shown in Listing (4). Beyond this order offers no increase in accuracy to 10 d.p as highlighted by Table (1), which shows how the option value changes every time we double the order of our Lagrange interpolation. For completeness we show the required computation time, which doesn't experience much change.

We can also look at the optimal value of i_{max} and j_{max} using Richardson extrapolation. We need to take care when varying these parameters to make sure that we always have a grid point at the discontinuity, which in this case occurs at $S = F = 50$. Thus we choose a base $S_{max} = 5F$ with an initial $i_{max} = j_{max} = 5$ and double this each time to ensure we always hit the point $S = F$. Second order Richardson extrapolation gives a better estimate of the option value, which we will denote by E_x, using [1]

$$E_x = \frac{4V_{2i_{max},2j_{max}} - V_{i_{max},j_{max}}}{3}. \tag{1.4.1}$$

We can also look at the convergence rate, c, which is

$$c = \frac{\log(R)}{\log(k)}, \tag{1.4.2}$$

where k is the rate at which we increase i_{max} and j_{max}, $k = 2$ in this case, and R is the ratio of difference between estimates, i.e.

$$R = \frac{V_{k*i_{\max},k*j_{\max}} - V_{i\max,j\max}}{V_{k*k*i\max,k*k*j\max} - V_{k*i\max,k*j\max}}. \tag{1.4.3}$$

Table (2) shows these various measurements along with the amount of computation time needed for $S_{max} = 5F$. We see that we can actually achieve 2 decimal place accuracy using $i_{max} = j_{max} = 40$ at the lowest. Given how small the computation time is, it is potentially worth taking a larger value as to be more confident in our results, $i_{max} = j_{max} = 320$ say, which ensures accuracy to an additional decimal place. Also note that the convergence rate c hovers around 2, which is what we expect, and will likely show a stronger relationship if we alter the value of S_{max}.

Lagrange interpolation order	Option value	Computation time
2	56.007918003	2×10^{-7}
4	55.9097747084	3×10^{-7}
8	55.9385200979	4×10^{-7}
16	55.9385200979	7×10^{-7}
32	55.9385200979	4×10^{-7}
64	55.9385200979	3×10^{-7}
128	55.9385200979	5×10^{-7}

Table 1: A table showing how the option value changes as we increase the order of Lagrange interpolation for $i_{\max} = i_{\max} = 5$ and $S_{\max} = 5F$. Beyond order 8 the value does not change to 10 d.p.

$i_{\max}/j_{\max}$	R	c	E_x	Computation time
5	0.0182022	-5.77975	74.25136013	3.23e-05
10	21.2965	4.41254	59.37811902	3.82e-05
20	3.11945	1.64129	59.62084877	9.82e-05
40	4.80413	2.26427	59.57470863	0.0003402
80	4.04638	2.01663	59.57405099	0.0009976
160	3.91202	1.96792	59.57436989	0.00396
320	3.79513	1.92415	59.57456557	0.0160735
640	3.6205	1.85619	59.57466568	0.0583985
1280	3.36057	1.74871	59.57471588	0.256964
2560	3.03072	1.59966	59.57474099	1.13136
5120	2.69416	1.42984	59.57475354	4.3302

Table 2: A table showing how the Richardson extrapolated option value changes as we increase i_{max} and j_{max} along with the computation time for each calculation, convergence c and parameter R. These calculations were performed for $S_{max} = 5F$.

Changing S_{max} can potentially give a better result but we have to be careful to scale the values of $i_{max} = j_{max} = 320$ in order to ensure we have a grid point at the discontinuity. Figure (3) shows two plots, both varying with S_{max} in steps of F with a linear scaling on i_{max} and j_{max}. The left hand plot shows how the option value $V(S, t = 0)$ changes and the right hand plot shows the computation time. We see that increasing S_{max} has minimal effect on the option value as it is

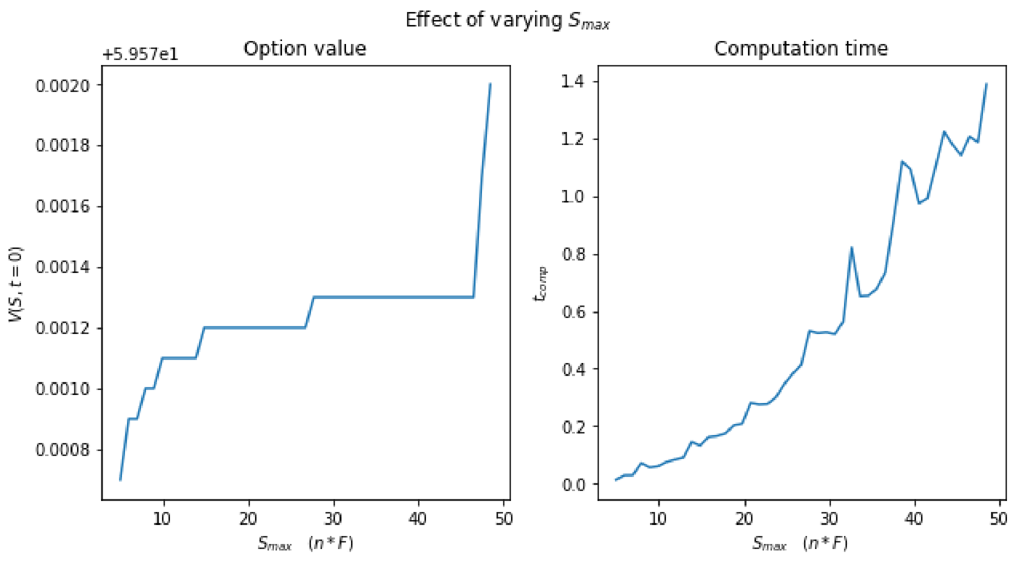

Figure 3: An image showing two plots, both varying with S_{max} in steps of F. Note i_{max} and j_{max} were scaled linearly with S_{max}. The left hand plot shows the option value $V(S, t = 0)$ and the right hand plot shows the computation time.

only changing at the fourth decimal place while the computation time is continuously increasing. Thus it is not worth using a large value of S_{max} for such a small accuracy increase when the computational cost if factored in. So $S_{max} = 20$ for example should be sufficient. We can repeat the extrapolation calculations of Table (2) for this value of $S_{\max}$, this is shown for some higher values of $i_{\max}$ and $j_{\max}$ in Table (3), where we see our scheme is indeed 2nd order.

$i_{\max}/j_{\max}$	R	c	E_x	Computation time
320	4.09292	2.03313	59.57468517	0.0183232
640	3.99549	1.99837	59.57470090	0.0710719
1280	3.94919	1.98156	59.57474580	0.298137

Table 3: A table showing how the Richardson extrapolated option value changes as we increase i_{max} and j_{max} along with the computation time for each calculation, convergence c and parameter R. These calculations were performed for $S_{max} = 20F$.

So ultimately we arrive at using $i_{max} = j_{max} = 1280$, $S_{max} = 20F$ and a Lagrange interpolation order of 8. This gives the option value as $V(S = 50.2, t = 0) = 59.57386206$ with a computation time of 0.332102s. As a quick confidence check we double all these parameters and find the value of the option does not change to 4 d.p and the computation time increases slightly. Thus, we are fairly confident that we have achieved an accurate option value in a time efficient way.

2. American style embedded options

We are now tasked with pricing an American style convertible bond, which obeys the inequality

$$V \geq RS, \tag{2.0.1}$$

for $t < T$. Thus the boundary condition at large S is now

$$V(S \to \infty, t) \to RS. \tag{2.0.2}$$

We embed a call option into this option so we have the extra condition

$$V(S,t) \leq \max(C_P, RS), \tag{2.0.3}$$

for $t < t_0 = 0.6208$, where $C_P = 60$ is the price the issuer can buy back the bond. Note that the combined conditions of equation (2.0.1) and equation (2.0.3) mean that for $S > C_p$ the value of the option must be $V(S, t = 0) = RS$ as this is the only way both conditions can be satisfied, given $R = 1$. We now have extra discontinuities in our problem. We will have a discontinuity in the S domain at the point $C_p = RS$ and a discontinuity in the time domain when $t = t_0$. To handle the S domain discontinuity we will force S_{max} to always be a multiple of C_p and then choose i_{max} and j_{max} as to always place a grid point at the discontinuity. For the time domain, we just have to ensure that for extrapolation we double the number of grid points in time so that the new points are placed either side of the old ones.

To implement the extra conditions in the code we utilise the penalty method where we re-write our PDE in equation (1.1.1) as

$$\begin{aligned}\frac{\partial P}{\partial t} + \frac{1}{2}\sigma^2 S^{2\beta}\frac{\partial^2 P}{\partial S^2} + \kappa(\theta(t) - S)\frac{\partial V}{\partial S} - rV + Ce^{-\alpha t}&\\ + \rho\max(RS - V, 0) + \rho\max(t_{0-} - t, 0)\max(V - \max(C_p, RS), 0) &= 0,\end{aligned} \tag{2.0.4}$$

where ρ is a large number. The first 'max' expression is present if $V < RS$, in which case it dominates the equation and forces V back up to this boundary. The other term is a product of two 'max' expressions which is only relevant if two conditions are satisfied. The first being that $t < t_0$ and the second being if $V > \max(C_p, RS)$, in which case this term dominates and forces V back down to the boundary.

The implementation of the penalty method with this altered scheme is shown in Listing (5).

2.1. Value as a function of S

Figure (4) shows the option value at the current time, $V(S, t = 0)$, varying with the price of the underlying, S. The green points are the optimal decision point, i.e where our decision on whether it is beneficial to exercise the option early changes.

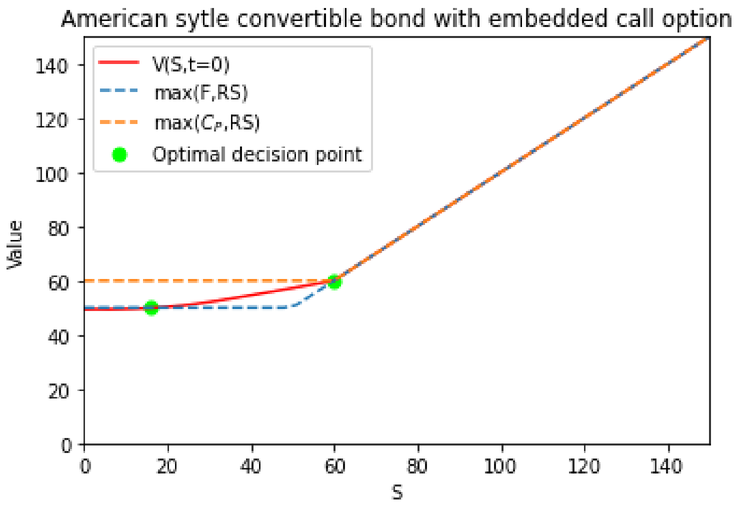

Figure 4: An image showing how the value of an American style convertible bond with an embedded call option changes as a function of the underlying S. The terminal condition for the bond (dashed blue line) and the boundary condition for the embedded call (dashed orange line) are shown, along with the optimal exercise points (green dots).

2.2. Value with varying κ

The risk-neutral process followed by the underlying S is given by

$$dS = \kappa \left[\theta(t) - S\right] dt + \sigma S^{\beta} dW, \tag{2.2.1}$$

thus if $S \ll \theta(t)$ we expect that the value of the option will increase, and if $S \gg \theta(t)$ we expect the option value to decrease, with this effect being magnified by the size of κ.

Figure (5) shows the value of the option as a function of the underlying price S for various $\kappa \in \{0.0625, 0.125, 0.1875\}$. The value of $\theta(0)$ is plotted as a vertical line. We see that below this theta value the option value is larger for larger β and as we approach $\theta(0)$ the values converge. For values of $S \gg \theta(0)$ we expect the order of these lines to reverse, however this effect is potentially obscured by the fact that all the lines must equal RS as soon as $S > C_p = 60$ which occurs shortly beyond $\theta(t) = 51.38$.

We can illustrate this point further by altering the value function theta to

$$\theta(t) = \frac{1}{2}(1 + \mu)X e^{\mu t}, \tag{2.2.2}$$

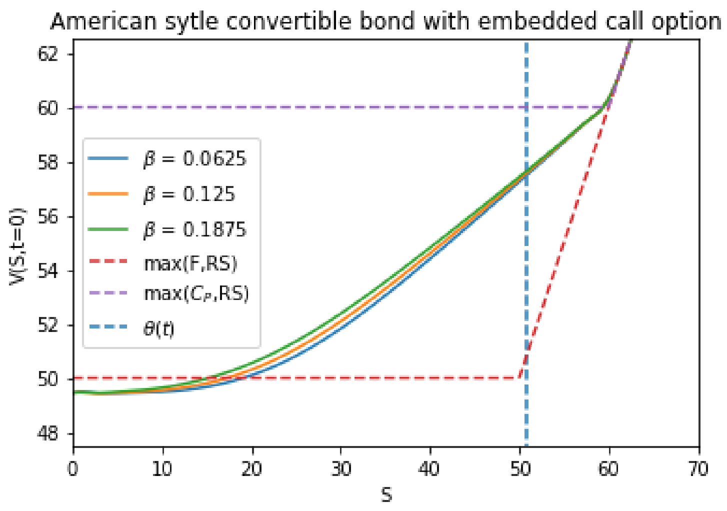

Figure 5: An image showing how the value of an American style convertible bond with an embedded call option changes as a function of the underlying S for different values of κ. The terminal condition for the bond (dashed red line) and the boundary condition for the embedded call (dashed purple line) are shown, along with the value of $\theta(t)$ (dashed blue line).

so now $\theta(0) = 25.69$ and thus our option values should have enough time to reverse their position, so the one corresponding to the smallest β is the highest, before they are forced to the boundary RS. This is indeed exactly what we see in Figure (6).

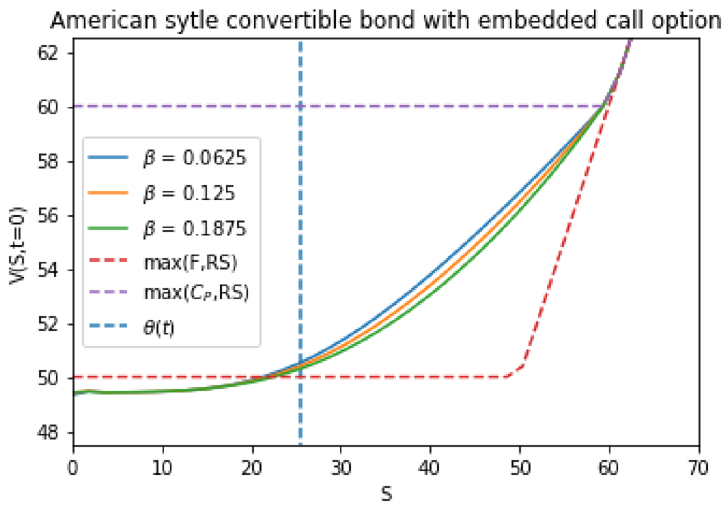

Figure 6: An image showing how the value of an American style convertible bond with an embedded call option changes as a function of the underlying S for different values of κ. The terminal condition for the bond (dashed red line) and the boundary condition for the embedded call (dashed purple line) are shown, along with the value of our shifted $\theta(t)$ (dashed blue line).

2.3. Time limited value

We are given 1 second of computation time to determine the most accurate option value with $S_0 = 50.5$. At the start of Section (2) we discussed the discontinuities in the domain and how to overcome them. The only parameters we really have to vary are $i_{\max}$, $j_{\max}$, $S_{\max}$ and the penalty method parameters such as the tolerance and maximum iterations. We saw in Sub-section (1.4) that anything above $S_{\max} = 20$ is likely excessive so this is a good starting point. We always want to build the grid to have a grid point on the discontinuity at C_p so we start from $i_{\max} = j_{\max} = 20$ and increment in steps of 20. Figure (7) shows how both the option value $V(S, t = 0)$ and computation time change as we do this. We see that 2 d.p precision can be achieved with

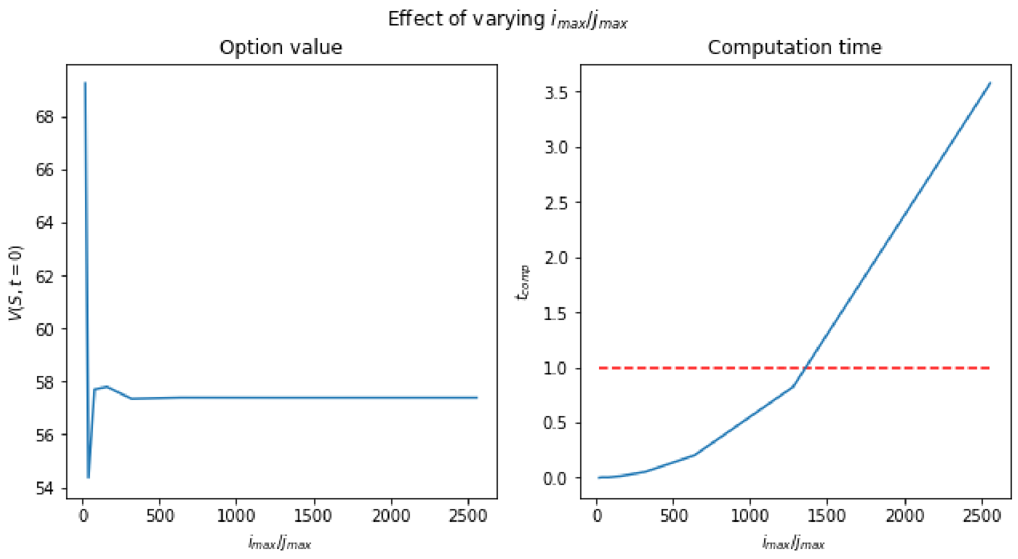

Figure 7: An image showing two plots, both as functions of $i_{\max}/j_{\max}$. The left hand plot shows the extrapolated option value $V(S, t = 0)$ from equation (1.4.1)and the right hand plot shows the computation time. The red dashed line is our time limit t=1s.

$i_{\max} = j_{\max} = 1280$ in under 1s. The convergence is not monotonic despite being confident in our scheme due to obtaining sensible results in previous parts, interpolating to a high order and aligning the grid points correctly. There are potential errors in the matrix solver, but generally we find the system is solved exactly, to almost zero error, in no more than 2 iterations, pushing ρ as high as 1×10^{11}. There are potentially other errors in the code that gives rise to this non-monotonic behavior. Either way, we do see converge and ultimately the most accurate value of the option we think we can achieve under 1s is $V(S, t = 0) = 57.40$.

3. Length and date

The number of words in this document is 1777
This document was submitted on 08/05/21 at 18:00

References

[1] L. F. Richardson, “Ix. the approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam,” *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, vol. 210, no. 459-470, pp. 307–357, 1911.

A. Convertible Bonds

Listing 1: C++ code showing the implementation of the matrix coefficients as part of the Crank-Nicolson method.

```cpp
// loop over the time levels
    for (int i{ i_max - 1 }; i >= 0; i--) {

        // declare matrix parameter vectors
        std::vector<double> a(j_max + 1);
        std::vector<double> b(j_max + 1);
        std::vector<double> c(j_max + 1);
        std::vector<double> d(j_max + 1);

        // initial values at j = 0
        a[0] = 0;
        b[0] = -(1 / dt) - (kappa * theta(mu, X, dt, i) / dS) - (r / 2);
        c[0] = kappa * theta(mu, X, dt, i);
        d[0] = (-(1 / dt) + (r / 2)) * v_old[0] - C * exp(-i * dt);

        // loop through middling values of j
        for (int j{ 1 }; j <= j_max - 1; j++) {

            a[j] = -0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) + (kappa / (4 * dS)) * (theta(mu, X, dt, i) - j * dS);
            b[j] = (1 / dt) + 0.5 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) + (r / 2);
            c[j] = -0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) - (kappa / (4 * dS)) * (theta(mu, X, dt, i) - j * dS);
            d[j] = (0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) - (kappa / (4 * dS)) * (theta(mu, X, dt, i) - j * dS)) * v_old[j - 1]
                + ((1 / dt) - 0.5 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) - (r / 2)) * v_old[j]
                + (0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) + (kappa / (4 * dS)) * (theta(mu, X, dt, i) - j * dS)) * v_old[j + 1]
                + C * exp(-alpha * i * dt);
        }

        // initialise values at j = j_max
        a[j_max] = 0;
        b[j_max] = 1;
        c[j_max] = 0;
        d[j_max] = R * (S[j_max] - X) * exp(-(kappa + r) * (T - i * dt)) + (X * R + (C / (alpha + r)) * exp(-alpha * i * dt)) * exp(-r * (T - i * dt)) + (C / (alpha + r)) * exp(-alpha * i * dt);
```

Listing 2: C++ code showing the implementation of a Thomas solver.

```cpp
// Thomas solver
std::vector<double> thomas_solve(const std::vector<double>& a, const std::vector<double>& b_, const std::vector<double>& c, std::vector<double>& d)
{
    // get size of vector
    int n = a.size();

    // local parameters
    std::vector<double> b(n), temp(n);

    // initial first value of b
    b[0] = b_[0];

    // get other values
    for (int j = 1; j < n; j++)
    {
        b[j] = b_[j] - c[j - 1] * a[j] / b[j - 1];
        d[j] = d[j] - d[j - 1] * a[j] / b[j - 1];
    }

    // calculate solution
    temp[n - 1] = d[n - 1] / b[n - 1];
    for (int j = n - 2; j >= 0; j--) temp[j] = (d[j] - c[j] * temp[j + 1]) / b[j];

    return temp;
}
```

Listing 3: C++ code showing the implementation of an SOR solver.

```cpp
// SOR solver
void SOR_solve(const std::vector<double>& a, const std::vector<double>& b, const std::vector<double>& c, const std::vector<double>& d,
    std::vector<double>& solution, const int& max_iter, const double& tolerence, const double& omega, int& iterations)
{
    // get size of vector
    int n = solution.size() - 1;

    // decalre y
    double y;

    // SOR loop
    for (iterations; iterations < max_iter; iterations++) {

        // reset error to 0
        double error = 0;

        // initial value
        y = (d[0] - c[0] * solution[1]) / b[0];
        solution[0] = solution[0] + omega * (y - solution[0]);

        // middelling values
        for (int j = 1; j < n; j++)
        {
            double y = (d[j] - a[j] * solution[j - 1] - c[j] * solution[j + 1]) / b[j];
            solution[j] = solution[j] + omega * (y - solution[j]);
        }

        // final value
        y = (d[n] - a[n] * solution[n - 1]) / b[n];
        solution[n] = solution[n] + omega * (y - solution[n]);

        // calculate residual norm ||r|| as sum of absolute values
        error += std::fabs(d[0] - b[0] * solution[0] - c[0] * solution[1]);
        for (int j = 1; j < n; j++) {
            error += std::fabs(d[j] - a[j] * solution[j - 1] - b[j] * solution[j] - c[j] * solution[j + 1]);
        }
        error += std::fabs(d[n] - a[n] * solution[n - 1] - b[n] * solution[n]);

        // make an exit condition when solution found
        if (error < tolerence) return;
    }

    if (iterations = max_iter) std::cout << "No convergence!" << std::endl;
}
```

Listing 4: C++ code showing a generic Lagrange interpolation function.

```
// generic lagrange interpolation
double lagrange_interpolation(const std::vector<double>& y, const std::vector<double>& x, double x0, unsigned int n)
{
    if (x.size() < n) return lagrange_interpolation(y, x, x0, x.size());
    if (n == 0)throw;

    // local parameters
    int nHalf = n / 2;
    double dx = x[1] - x[0];

    // calculate j star
    int jStar;
    if (n % 2 == 0) jStar = int((x0 - x[0]) / dx) - (nHalf - 1);  // even degree
    else jStar = int((x0 - x[0]) / dx + 0.5) - (nHalf);  // odd degree

    jStar = std::max(0, jStar);
    jStar = std::min(int(x.size() - n), jStar);

    if (n == 1)return y[jStar];

    double temp = 0.;
    for (unsigned int i = jStar; i < jStar + n; i++) {

        double  int_temp;
        int_temp = y[i];

        for (unsigned int j = jStar; j < jStar + n; j++) {

            if (j == i) { continue; }
            int_temp *= (x0 - x[j]) / (x[i] - x[j]);
        }
        temp += int_temp;
    }  // end of interpolate

    return temp;
}
```

B. Embedded options

Listing 5: C++ code showing the implementation of the Crank-Nicolson Finite Difference scheme with the penalty method for equation (2.0.4).

```
// Crank Nicolson Finite Difference
double crank_nicolson(const double& T, const double& F, const double& R, const double& r, const double& kappa, const double&
        mu,
    const double& S0, const double& X, const double& C, const double& alpha, const double& beta, const double& sigma, const
        int& i_max, const int& j_max, double& S_max,
    const double& rho, const double& tol, const int& iter_max, const double& Cp, const double& t0)
{
    // declare and initialise local parameters (dS, dt)
    double dS = S_max / j_max;
    double dt = T / i_max;
```

```cpp

    // create storage for stock price and old and new option price
    std::vector<double> S(j_max + 1);
    std::vector<double> v_old(j_max + 1);
    std::vector<double> v_new(j_max + 1);

    // initialise our stock prices
    for (int j{ 0 }; j <= j_max; j++) S[j] = j * dS;

    // initialise final conditions on the option price
    for (int j{ 0 }; j <= j_max; j++) {
        v_old[j] = std::max(F, R * S[j]);
        v_new[j] = std::max(F, R * S[j]);
    }

    // loop over the time levels
    for (int i{ i_max - 1 }; i >= 0; i--) {

        // declare matrix parameter vectors
        std::vector<double> a(j_max + 1);
        std::vector<double> b(j_max + 1);
        std::vector<double> c(j_max + 1);
        std::vector<double> d(j_max + 1);

        // initial values at j = 0
        a[0] = 0;
        b[0] = -(1 / dt) - (kappa * theta(mu, X, dt, i) / dS) - (r / 2);
        c[0] = kappa * theta(mu, X, dt, i) / dS;
        d[0] = (-(1 / dt) + (r / 2)) * v_old[0] - C * exp(-i * dt);

        // loop through middling values of j
        for (int j{ 1 }; j <= j_max - 1; j++) {

            a[j] = -0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) + (kappa / (4 * dS)) * (theta(mu, X, dt,
                   i) - j * dS);
            b[j] = (1 / dt) + 0.5 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) + (r / 2);
            c[j] = -0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) - (kappa / (4 * dS)) * (theta(mu, X, dt,
                   i) - j * dS);
            d[j] = (0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) - (kappa / (4 * dS)) * (theta(mu, X, dt,
                   i) - j * dS)) * v_old[j - 1]
                + ((1 / dt) - 0.5 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) - (r / 2)) * v_old[j]
                + (0.25 * pow(sigma, 2) * pow(j, 2 * beta) * pow(dS, 2 * (beta - 1)) + (kappa / (4 * dS)) * (theta(mu, X, dt,
                      i) - j * dS)) * v_old[j + 1]
                + C * exp(-alpha * i * dt);
        }

        // initialise values at j = j_max
        a[j_max] = 0;
        b[j_max] = 1;
        c[j_max] = 0;
        d[j_max] = R * S[j_max];
        // d[j_max] = R * (S[j_max] - X) * exp(-(kappa + r) * (T - i * dt)) + (C / (alpha + r)) * exp(-alpha * i * dt) + (X *
             R - (C / (alpha + r)) * exp(-alpha * T)) * exp(-r * (T - i * dt));

        // penalty method
        int penalty_itr;
        for (penalty_itr = 0; penalty_itr < iter_max; penalty_itr++) {

            // create new vectors containing a copy of the FD approximations
            std::vector<double> a_hat(a), b_hat(b), c_hat(c), d_hat(d);

            // apply penalty to finite difference scheme
            for (int j{ 1 }; j < j_max; j++) {

                // apply american penalty if needed
                if (v_new[j] < R * S[j]) {
                    b_hat[j] = b[j] + rho;
                    d_hat[j] = d[j] + rho * (R * S[j]);
                }

                // if in embedded call region
                if (i * dt <= t0) {

                    // apply call penalty if needed
                    if (v_new[j] > std::max(Cp, R * S[j])) {
                        b_hat[j] = b[j] + rho;
                        d_hat[j] = d[j] + rho * std::max(Cp, R * S[j]);
                    }
                }
            }

            // solve with Thomas method
            std::vector<double> y = thomas_solve(a_hat, b_hat, c_hat, d_hat);

            // check for differenc between y and v_new
            double error = 0;
            for (int j{ 0 }; j < j_max; j++) {
                error += pow(v_new[j] - y[j], 2);
            }

            // update v_new
            v_new = y;

            // exit if solution converged
            if (error < pow(tol, 2)) {
                //std::cout << "Solved after " << penalty_itr << " iterations" << std::endl;
```

```
                    break;
                }

            }

            // if no solution found
            if (penalty_itr >= iter_max) {
                std::cout << "Error: No convergence" << std::endl;
                throw;
            }

            // set old to new
            v_old = v_new;
        }

        // use lagrange interpolation to get estimated option value
        double option_value = lagrange_interpolation(v_new, S, S0, 8);

        return option_value;
}
```