MATH40082
Finite Difference Methods Assignment

10945438

1 Coupon Bonds with Stochastic Interest Rate

1.1 Introduction

In the context of this work, the value of coupon bonds are examined, assuming stochastic interest rate term structures under the risk-neutral measure given by

$$dr = \kappa\big(\theta e^{\mu t} - r\big)\,dt \;+\; \sigma\, r^{\beta}\,dW, \tag{1}$$

where κ, θ, μ, σ, and β are calibrated/estimated market behaviour parameters, and $B(r,t;T)$ is the price of a coupon bond that matures at time T, evaluated at time t, when the interest rate is r. The bond price satisfies the partial differential equation (PDE) (2) given a continuous coupon rate $Ce^{-\alpha t}$, where C and α are constants defined in the context of the bond contract.

$$\frac{\partial B}{\partial t} \;+\; \frac{1}{2}\,\sigma^2\, r^{2\beta}\,\frac{\partial^2 B}{\partial r^2} \;+\; k\big(\theta\, e^{\mu t} - r\big)\,\frac{\partial B}{\partial r} \;-\; r\,B \;+\; C\,e^{-\alpha t} \;=\; 0. \tag{2}$$

Along with the above PDE, the pricing problem is further specified by a domain of $r \in [0,\infty)$ and $t < T$ and a set of boundary conditions given by:

- A terminal condition: $B(r, t = T; T) = F$,
- At $r = 0$: $\frac{\partial B}{\partial t} + \kappa\theta e^{\mu t}\frac{\partial B}{\partial r} + Ce^{-\alpha t} = 0$,
- As $r \to \infty$: $B(r,t;T) \to 0$.

1.2 Finite Difference Scheme | Task 2.1

The PDE described by equation (2) being parabolic in nature implies that a numerical scheme can, employing finite difference methods appropriately, be constructed to find a solution. This involves both temporal discretisation (t) as well as discretisation with respect to the state variable (interest rate r). In this work, the Crank-Nicolson method was chosen due to its stability and convergence characteristics, as well as its second-order accuracy in both time and the state variable.

Equation (2) can be effectively discretised by utilising second-order finite difference approximations $O((\Delta r)^2, (\Delta t)^2)$ to the bond-pricing PDE's derivatives. These approximations applied to equation (2) are seen in the following derivation:

Beginning with a Taylor-series expansion of B in t about $t + \frac{1}{2}\Delta t$, results in

$$B(r,t) = B\big(r, t+\tfrac{1}{2}\Delta t\big) \;-\; \tfrac{1}{2}\,\Delta t\,\frac{\partial B}{\partial t}\big(r, t+\tfrac{1}{2}\Delta t\big) \;+\; \tfrac{1}{8}\,(\Delta t)^2\,\frac{\partial^2 B}{\partial t^2}\big(r, t+\tfrac{1}{2}\Delta t\big) \;+\; O\big((\Delta t)^3\big), \text{ and} \tag{3}$$

$$B(r,t+\Delta t) = B\big(r, t+\tfrac{1}{2}\Delta t\big) \;+\; \tfrac{1}{2}\,\Delta t\,\frac{\partial B}{\partial t}\big(r, t+\tfrac{1}{2}\Delta t\big) \;+\; \tfrac{1}{8}\,(\Delta t)^2\,\frac{\partial^2 B}{\partial t^2}\big(r, t+\tfrac{1}{2}\Delta t\big) \;+\; O\big((\Delta t)^3\big). \tag{4}$$

Subtracting (3) from (4) gives

$$B(r,t+\Delta t) - B(r,t) = \Delta t\,\frac{\partial B}{\partial t}\big(r, t+\tfrac{1}{2}\Delta t\big) + O\big((\Delta t)^3\big), \tag{5}$$

so

$$\frac{B(r,t+\Delta t) - B(r,t)}{\Delta t} = \frac{\partial B}{\partial t}\big(r, t+\tfrac{1}{2}\Delta t\big) + O\big((\Delta t)^2\big). \tag{6}$$

In order to relevantly estimate $\frac{\partial B}{\partial r}$ at the half-step $t + \frac{1}{2}\Delta t$, the derivative approximation must be found by averaging in time the central difference approximations for $\frac{\partial B}{\partial r}$ at t and $t + \frac{1}{2}\Delta t$, as their is effectively no grid point at the half-step:

$$\begin{aligned}\frac{\partial B}{\partial r}\big(r, t + \tfrac{1}{2}\Delta t\big) &\approx \tfrac{1}{2}\Big[\frac{\partial B}{\partial r}(r,t) + \frac{\partial B}{\partial r}(r, t+\Delta t)\Big] \\ &= \frac{B(r+\Delta r,t) - B(r-\Delta r,t) + B(r+\Delta r,t+\Delta t) - B(r-\Delta r,t+\Delta t)}{4\,\Delta r} + O\big((\Delta r)^2, (\Delta t)^2\big)\,.\end{aligned} \tag{7}$$

where the error term has come to include the time-averaging error of order $((\Delta t)^2)$. Similarly, for the $-rB$ term, a half-time-step averaging is performed:

$$-r\,B\big(r, t + \tfrac{1}{2}\Delta t\big) \approx -r\;\tfrac{1}{2}\big[B(r,t) + B(r, t+\Delta t)\big] + O\big((\Delta t)^2\big). \tag{8}$$

And for the second derivative,

$$\frac{\partial^2 B}{\partial r^2}\Big(r, t + \tfrac{1}{2}\Delta t\Big) = \tfrac{1}{2}\Big[\frac{\partial^2 B}{\partial r^2}(r,t) + \frac{\partial^2 B}{\partial r^2}(r, t+\Delta t)\Big] + O\big((\Delta r)^2, (\Delta t)^2\big), \tag{9}$$

where

$$\frac{\partial^2 B}{\partial r^2}(r,t) = \frac{B(r+\Delta r,t) - 2\,B(r,t) + B(r-\Delta r,t)}{(\Delta r)^2} + O\big((\Delta r)^2\big), \text{ and} \tag{10}$$

$$\frac{\partial^2 B}{\partial r^2}(r,t+\Delta t) = \frac{B(r+\Delta r,t+\Delta t) - 2\,B(r,t+\Delta t) + B(r-\Delta r,t+\Delta t)}{(\Delta r)^2} + O\big((\Delta r)^2\big), \tag{11}$$

where the $O\big((\Delta r)^2\big)$ terms in (10) and (11) contribute to the overall error seen in (9).

The above approximations (6)-(11) can be incorporated into a numerical scheme, first writing them in grid-point index notation, where $t_i = i\,\Delta t$, with $i = 0, 1, \ldots, i_{\text{Max}}$, $r_j = j\,\Delta r$, with $j = 0, 1, \ldots, j_{\text{Max}}$, and $B^i_j \equiv B(r_j, t_i)$, such that at the half-time-step, the finite-difference approximations give

$$\frac{\partial B}{\partial t} \approx \frac{B^{i+1}_j - B^i_j}{\Delta t}, \tag{12}$$

$$\frac{\partial B}{\partial r} \approx \frac{B^i_{j+1} - B^i_{j-1} + B^{i+1}_{j+1} - B^{i+1}_{j-1}}{4\Delta r}, \tag{13}$$

$$B \approx \tfrac{1}{2}\big(B^i_j + B^{i+1}_j\big), \tag{14}$$

$$\frac{\partial^2 B}{\partial r^2} \approx \frac{B^i_{j+1} - 2B^i_j + B^i_{j-1} + B^{i+1}_{j+1} - 2B^{i+1}_j + B^{i+1}_{j-1}}{2(\Delta r)^2}, \tag{15}$$

Substituting these approximations into the initial equation (2) gives, after collecting unknown values on the left and known values on the right, the following:

$$a_j B^i_{j-1} + b_j B^i_j + c_j B^i_{j+1} = d_j, \text{ where} \tag{16}$$

$$a_j \;=\; \frac{1}{4}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} \;-\; \frac{1}{4\,\Delta r}\;k\,\Big(\theta\big(e^{\mu i \Delta t} + e^{\mu (i+1)\Delta t}\big) - j\,\Delta r\Big), \tag{17}$$

$$b_j \;=\; -\frac{1}{\Delta t} \;-\; \frac{1}{2}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} \;-\; \frac{j\,\Delta r}{2}, \tag{18}$$

$$c_j \;=\; \frac{1}{4}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} \;+\; \frac{1}{4\,\Delta r}\,k\left(\theta\big(e^{\mu i\Delta t}+e^{\mu(i+1)\Delta t}\big)-j\,\Delta r\right),\ \text{and} \tag{19}$$

$$d_j \;=\; -\,a_j\,B_{j-1}^{i+1} \;+\; \Big(-\frac{1}{\Delta t} \;+\; \frac{1}{2}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} \;+\; \frac{j\,\Delta r}{2}\Big)\,B_j^{i+1} \;-\; c_j\,B_{j+1}^{i+1} \;-\; \tfrac{1}{2}\,C\,e^{-\alpha i\Delta t} \;-\; \tfrac{1}{2}\,C\,e^{-\alpha(i+1)\Delta t}, \tag{20}$$

for $1 \le j < j\text{Max}$, and such that d represents the unknown values and non-coefficient terms at the relevant time-point.

The above discretisation requires the consideration of boundary conditions in order to evaluate the system at $j = 0$ and $j = j_{\text{Max}}$. The terminal condition of the given system, that is the value of B at $t = T$, allows for the i_{Max} indexed B value to be written as

$$B_j^{\,i_{\text{Max}}} = F \tag{21}$$

The second boundary condition being a simplified PDE governing the bond value B at $r = 0$ (and $j = 0$), can be used to give the following discretised equation in the relevant notation:

$$b_0\,B_0^i \;+\; c_0\,B_1^i \;=\; d_0,\ \text{where} \tag{22}$$

$$b_0 \;=\; -\frac{1}{\Delta t} \;-\; \frac{k\,\theta\Big(e^{\mu i\,\Delta t}+e^{\mu(i+1)\,\Delta t}\Big)}{2\,\Delta r},\ c_0 \;=\; \frac{k\,\theta\Big(e^{\mu i\,\Delta t}+e^{\mu(i+1)\,\Delta t}\Big)}{2\,\Delta r},\ \text{and} \tag{23}$$

$$d_0 \;=\Big(-\frac{1}{\Delta t} \;+\; \frac{k\,\theta\Big(e^{\mu i\,\Delta t}+e^{\mu(i+1)\,\Delta t}\Big)}{2\,\Delta r}\Big)\,B_0^{i+1} \;-\; c_0\,B_1^{i+1} - \;\tfrac{1}{2}\,C\,e^{-\alpha\,i\,\Delta t} \;-\; \tfrac{1}{2}\,C\,e^{-\alpha\,(i+1)\,\Delta t}. \tag{24}$$

where once again d_0 contains the values of later time steps. The third boundary condition, being $B(r,t;T) \to 0$, as $r \to \infty$, can be used to impose a Dirichlet boundary condition in this work's numerical scheme that truncates the asymptotic behaviour to some maximal value of r, r_{Max} (at j_{Max}), considered in the domain. This is simply denoted as

$$B_{j\text{Max}}^i = 0,\ \text{for every } i. \tag{25}$$

The above condition allows for the discretisation by conveniently setting the last row of the tridiagonal matrix as $a_{j\text{Max}} = c_{j\text{Max}} = d_{j\text{Max}} = 0$, and $b_{j\text{Max}} = 1$.

1.3 Application and Basic Result | Task 2.1

The discretised system given in (16), the results derived from the boundary condition in (22) and the Dirichlet condition, constitute a numerical scheme that allows for the iterative solution of the resulting matrix system. The solution of the Crank-Nicolson scheme is computed backward in time-steps on a 101×101 grid (a result of the given discretisation parameters: $i_{\max} = 100$, $j_{\max} = 100$, $r_{\max} = 1$). At each time-step a matrix of the form

$$\mathbf{A}_{a_j,\,b_j,\,c_j}\,\mathbf{B}^{\,i} \;=\; \mathbf{d}^{\,i} \tag{26}$$

is computed, where $\mathbf{A}$ is a tridiagonal matrix with diagonals a_j, b_j, c_j (lower, main, and upper, respectively). The system matrix is set down in SciPy's banded format, then solved using SciPy `solve_banded`. The dependence on time results in having to re-compute the matrix at each time-step. In summary, the terminal price $B_j^{i_{\max}} = F$ is set, then for each time iteration ($i = i_{\max} - 1, \ldots, 0$), a_j, b_j, c_j and $\mathbf{d}^{\,i}$ are computed, the boundary condition rows $j = 0$ and $j = j_{\max}$ are imposed, and the system is solved to get $\mathbf{B}^{\,i}$, the result is copied forward to become the "old" vector. Given this work's relevant parameters, (T=3, $F = 81$, $\theta =$

0.0262, $r_0 = 0.0381$, $\kappa = 0.08169$, $\mu = 0.015$, $C = 2.58$, $\alpha = 0.02$, $\beta = 0.413$ and $\sigma = 0.111$), the scheme yields

$$\boxed{B(r_0 = 0.0381,\ t = 0;\ T = 3)\ \approx\ 75.65012897225886}$$

as the price of the bond.

1.4 Neumann Boundary Condition: Properties and Application | Task 2.1

An alternative to the strict Dirichlet boundary condition in $B \to 0$ as $r \to \infty$, is the more elegant

$$\frac{\partial B}{\partial r} \to 0 \text{ as } r \to \infty \tag{27}$$

which provides a more flexible approach, and permits a smooth transition of bond price as related variables (r) change indefinitely. The above Neumann boundary condition captures/necessitates that at high values of $r \to \infty$, changes and/or fluctuations in bond price B become negligible, retaining the asymptotic behaviour in the pricing model, without truncations and introduction of abrupt constraints. This boundary condition can be used to set down a discretised equation for the bond at index j_{Max}, given by

$$-\,B^i_{j_{\text{Max}}-1}\ +\ B^i_{j_{\text{Max}}}\ =\ 0. \tag{28}$$

In order to compare the two models in practice, the interest rate was varied and the bond value yielded by both models were found, the resulting values were plotted in plot 1.

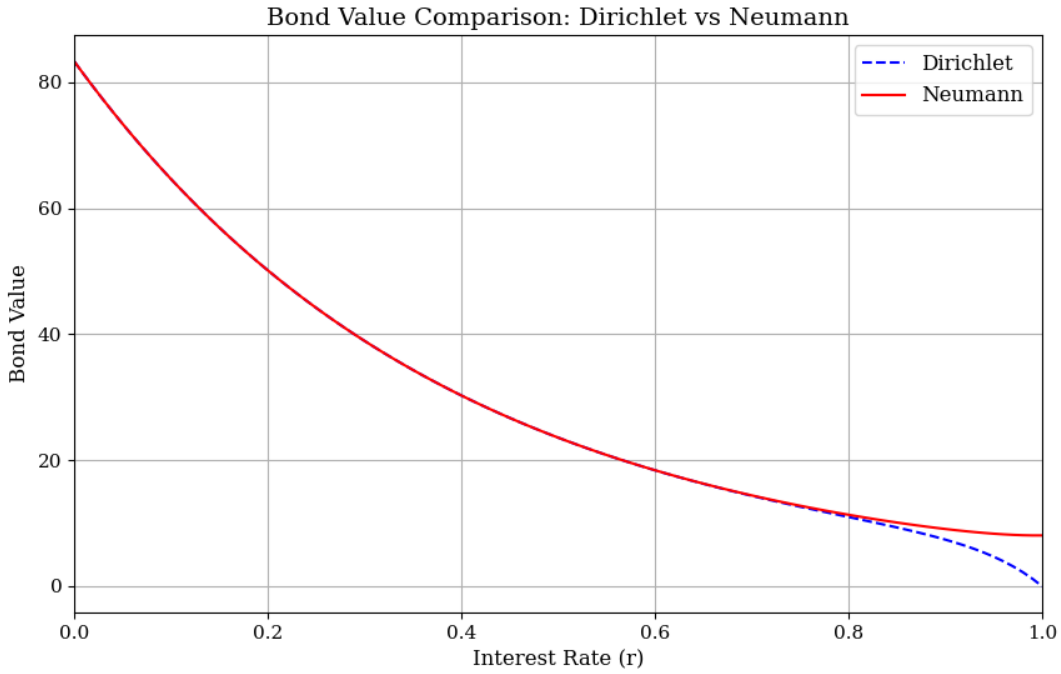

Figure 1: Bond price $B(r, 0; T = 3)$ as a function of the interest rate r, with the provided/given parameters. The dashed blue curve represents the numerical results yielded by the Dirichlet boundary condition $B(r_{\text{max}}) = 0$, while the red curve is from the Neumann boundary condition $\partial B/\partial r(r_{\text{max}}) = 0$.

Upon observation, both solutions agree remarkably well within the rather realistic range of $0 < r < 0.6$, indicating that the choice of r_{Max} boundary condition has negligible impact on the bond's valuation in that

range. It is clear however that a divergence is exhibited strongly at interest values above $r > 0.8$, culminating at a hard disagreement caused by the Dirichlet condition necessitating a bond value of zero at $r_{\max}$. This divergence suggests a more in-depth examination of the Dirichlet and Neumann boundary conditions would be a fruitful investigative endeavor.

1.4.1 Further Analysis | Task 2.1

In order to explore this divergence, the upper boundary for the interest rate, $r_{\max}$, was varied, and the resulting changes in valuation with the Dirichlet condition was noted and plotted in Figure 2.

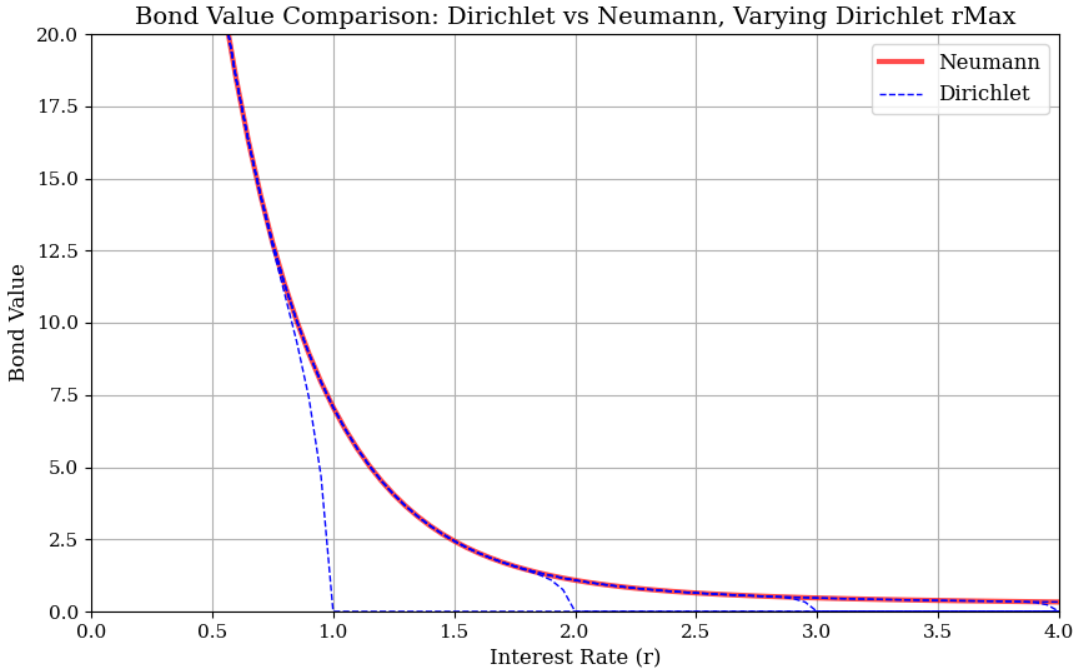

Figure 2: Bond price $B(r, 0; T = 3)$ as a function of the interest rate r, with the provided/given parameters. The dashed blue curves are numerical results yielded by the Dirichlet boundary conditions $B(r_{\max}) = 0$ given different r boundary endpoints, while the red curve is from the Neumann boundary condition with a boundary at 5 $\partial B/\partial r(r_{\max}) = 0$.

The results shown in the above figure clearly display the non-negligible impact that the chosen maximal upper limit boundary location has on the bond valuation with the Dirichlet condition, while the Neumann curve has remained far more stable when compared with the curve limited by r_{Max}=1 in Figure 1. This implies an inherent instability in the Dirichlet boundary condition under different r boundary settings, suggesting it functions relatively poorly at values nearing r_{Max} as it is highly dependent on the configuration of the numerical scheme $\sim$ an indicator of concern in the standard practice of numerical schemes and finite difference methodology, this analysis implies the Neumann boundary condition provides a more robust framework.

2 Options on a Bond

2.1 Introduction

The examination and valuation of an *option* V on the previously discussed bond is a separate undertaking, and in the context of this work in particular, the American call option will be studied, where early exercise is allowed before T_1 (expiry time). V is known to satisfy a PDE given by

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 r^{2\beta}\frac{\partial^2 V}{\partial r^2} + \kappa(\theta e^{\mu t} - r)\frac{\partial V}{\partial r} - rV = 0. \tag{29}$$

for $r \in [0, \infty)$, $t < T_1$, and where $V(r, t; T_1, T)$ is the relevant call option, derived on the right to buy the coupon bond $B(r, t; T)$, at time T_1. At expiry, the option value $V(r, t = T_1; T)$ is known to satisfy the following equation:

$$V(r, t = T_1; T) = \max(B(r, T_1; T) - X, 0), \tag{30}$$

where $B(r, T_1; T)$ denotes the bond price at time T_1 for maturity T and X is a given value. If the option is exercised before maturity T_1, the value conforms to a payoff of

$$V(r, t; T_1, T) = B(r, t; T) - X. \tag{31}$$

Resultant of the no-arbitrage principle, and accounting for the case of early exercise, the option is known to satisfy

$$V(r, t; T_1, T) \geq \max(B(r, t; T) - X, 0), \quad \text{for } t \leq T_1. \tag{32}$$

The option-valuation portion of this work is further prescribed by the following boundary conditions:

$$V(0, t; T_1, T) = B(0, t; T) - X, \quad \text{at } r = 0, \text{ and} \tag{33}$$

$$V(r, t; T_1, T) \to 0, \quad \text{as } r \to \infty. \tag{34}$$

2.2 Finite Difference Scheme | Task 2.2

2.2.1 Non-Early Exercise Numerical Scheme | Task 2.2

Once again, the problem-at-hand can be made to utilise finite difference methods and be incorporated into a numerical scheme, so the use of grid-point index notation will be extended, where $r_j = j\Delta r,\ j = 0, \ldots, j_{\max}$, $t_i = i\Delta t,\ i = 0, \ldots, i_{\max}$, and $V_j^i = V(r_j, t_i; T_1, T)$. Equivalent approximations to those presented for B in (6)-(11) can be used in reference to the PDE given by equation (29) to give finite-difference approximations at half-time-step:

$$\frac{\partial V}{\partial t} \approx \frac{V_j^{i+1} - V_j^i}{\Delta t}, \tag{35}$$

$$\frac{\partial V}{\partial r} \approx \frac{V_{j+1}^i - V_{j-1}^i + V_{j+1}^{i+1} - V_{j-1}^{i+1}}{4\Delta r}, \tag{36}$$

$$V \approx \tfrac{1}{2}\big(V_j^i + V_j^{i+1}\big), \tag{37}$$

$$\frac{\partial^2 V}{\partial r^2} \approx \frac{V_{j+1}^i - 2V_j^i + V_{j-1}^i + V_{j+1}^{i+1} - 2V_j^{i+1} + V_{j-1}^{i+1}}{2(\Delta r)^2}, \tag{38}$$

where the above approximations are then to be substituted into the option pricing PDE (equation 29), and the time-unknown manipulated to the LHS and vice versa for the known values, to give the below discretised equation (for $1 \leq j \leq j_{\max} - 1$),

$$a_j V^i_{j-1} + b_j V^i_j + c_j V^i_{j+1} = d_j, \text{ where} \tag{39}$$

$$a_j = \tfrac{1}{4}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} \;-\; \frac{\kappa\big(\theta(e^{\mu t_i} + e^{\mu t_{i+1}}) - j\,\Delta r\big)}{4\,\Delta r},$$

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

$$c_j = \tfrac{1}{4}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} \;+\; \frac{\kappa\big(\theta(e^{\mu t_i} + e^{\mu t_{i+1}}) - j\,\Delta r\big)}{4\,\Delta r},$$

$$d_j = -\,a_j\,V^{i+1}_{j-1} \;+\; \Big(-\frac{1}{\Delta t} + \tfrac{1}{2}\,\sigma^2\, j^{2\beta}\,(\Delta r)^{2\beta-2} + \frac{j\,\Delta r}{2}\Big)\,V^{i+1}_j \;-\; c_j\,V^{i+1}_{j+1}.$$

The numerical factors to be used at the boundary rows ($r = 0,\ r = r_{\text{Max}}$, or equivalently $j = 0$ and $j = j_{\text{Max}}$) are once again separate from those characterising the interior rows and are dependant on the behavior exhibited by the solution at the boundaries. The boundary condition given in equation (33) implies that for $j = 0$, $a_0 = c_0 = 0$, $b_0 = 1$, and $d_0 = B^i_0 - X$.

Alternately, at $j = j_{\text{Max}}$ (i.e. $r = r_{\text{Max}}$), the second boundary condition in equation (34) implies $a_{j_{\text{Max}}} = c_{j_{\text{Max}}} = d_{j_{\text{Max}}} = 0$ and $b_{j_{\text{Max}}} = 1$.

2.2.2 Early Exercise Numerical Scheme | Task 2.2

The no-arbitrage condition necessitates that at any time iteration, if the value of the given option $V(r_j, t_i; T_1, T)$ is greater than that of the immediate exercise payoff $B(r_j, t_i; T) - X$, the option should be held, whereas if $V(r_j, t_i; T_1, T) = B(r_j, t_i; T) - X$, the option is exercised. This can be incorporated into the Crank-Nicolson scheme using the Projected Successive Over Relaxation (PSOR) method.

The PSOR method builds off of the tridiagonal system of equations arising from the Crank-Nicolson scheme as in the preceding section, specifically:

$$\mathbf{A}_{a_j,\,b_j,\,c_j}\ \mathbf{V}^{\,i} \;=\; \mathbf{d}^{\,i}, \tag{40}$$

where j indices have been omitted for notational clarity. In PSOR, a guess for the value of a given V^i is made, $V^{i,0}$, which is iteratively refined in a series of *relaxation sweeps* (whence its name derives) by looping through the j index, and updating values with the most recent and proximate values. The PSOR method relies on a simple consideration that a given iteration value is simply an additive correction to its iterative predecessor, that is $V^{i,k+1}_j = V^{i,k}_j + (V^{i,k+1}_j - V^{i,k}_j)$, where k is the iteration index for the PSOR sweep/loop. The PSOR method advances further and allows for early excercise to be incorporated by considering the optimality of either having held or exercised the option at every iteration. Practically, this method obeys

$$V^{i,k+1}_j = \max\Big(V^{i,k+\frac{1}{2}}_j,\, B^i_j - X\Big), \tag{41}$$

where k is the iteration index for the PSOR algorithm, and $1 < \omega < 2$ is the over-relaxation parameter (assists in accelerating convergence), and where

$$V^{i,k+\frac{1}{2}}_j = V^{i,k}_j + \omega\big(y^{i,k+1}_j - V^{i,k}_j\big), \text{ then where} \tag{42}$$

$$y^{i,k+1}_j = \frac{1}{b_j}\Big(d^i_j - a_j\,V^{i,k}_{j-1} - c_j\,V^{i,k}_{j+1}\Big). \tag{43}$$

The algorithm's stopping condition is reliant on a chosen error tolerance value, a result of the convergence of the additive correction via successive iterations, defined by

$$\sum_j \epsilon_j^2 < \text{tolerance}^2, \quad \text{where} \quad \epsilon_j = V^{i,k+1}_j - V^{i,k}_j. \tag{44}$$

2.3 Application and Basic Results and Plots | Task 2.2

The discretised system and the associated found parameters, as well as the practical forms of boundary conditions discussed in subsection 2.2.1, constitute a numerical scheme that allows for the iterative solution of the resulting matrix system, similar to the solution to the lone bond, but now with including the loop for convergence of successive relaxations over the rate index j, in line with the PSOR modification. The solution of the Crank-Nicolson scheme is again computed backward in time-steps on a 101×101 grid (a result of the given discretisation parameters: $i_{\max} = 100$, $j_{\max} = 10000$, $r_{\max} = 5$).

At each time-step the bond matrix is computed as previously discussed, but in accommodation of the PSOR method, in iterations *temporally* up to the option expiry T_1, the factors in equation (40) are computed, and used to form the exercise payoff E_j^i to allow for the optimal exercise decision at each node, the application of (41) - (43) is carried out until convergence is reached in accordance with the tolerance criteria, by monitoring $\sum_j (V_j^{i,k+1} - V_j^{i,k})^2$ and exiting when it falls below tolerance^2, or when k=1000.

In summary *alongside* the earlier bond procedure, the terminal option payoff $V_j^{i_1} = \max(B_j^{i_1} - X, 0)$ is set at expiry ($i_1 = \lfloor T_1/\Delta t \rfloor$), then for each relevant time iteration ($i = i_1 - 1, \ldots, 0$), the factors (of the previous section) a_j, b_j, c_j and $\mathbf{d}^i$ are computed, the boundary condition rows are imposed, and the PSOR process sweeps through the j index, and the system is solved to get $\mathbf{V}^i$, the result is copied forward to become the "old" vector.

Using this work's relevant parameters, (T=3, T_1=1.0729, $X = 82$, $F = 81$, $\theta = 0.0262$, $r_0 = 0.0381$, $\kappa = 0.08169$, $\mu = 0.015$, $C = 2.58$, $\alpha = 0.02$, $\beta = 0.413$ and $\sigma = 0.111$), a plot of the the value of discussed American call option V against the interest rate r at time t = T1 and time t = 0, was made and is displayed in Figure 3.

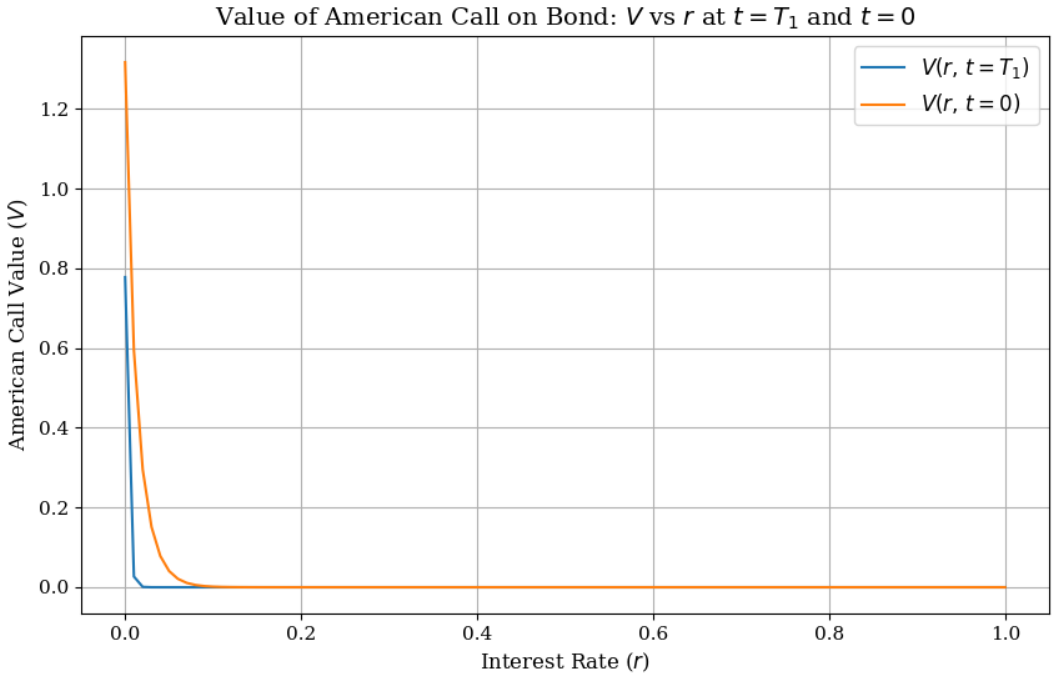

Figure 3: Value of given American call option $V(r, t; T_1, T)$ against the rate r at time $t = T_1$ and time $t = 0$, for the given market-fitted parameters.

The maximum value of r at time $t = T_1$ at which the option is exercised is effectively the rate boundary above which the optimal choice is holding (not exercising), in accordance with the no arbitrage result in (32). Effectively $V(r, T_1) - \max(B(r, T_1) - X, 0)$ is the difference between the value of the option and the value of its immediate exercise, which at the desired boundary r, is effectively zero. The grid can be trivially searched and the maximal value of the interest rate can be identified. This was carried out in the context of the previously discussed parameters and procedures and the resulting rate was found to be

$$\boxed{r_{\text{MaxExercise}} \approx 0.004}$$

as the maximal value of the rate r for which the option is exercised.

The discussed implementation was also used (with this work's previously stated parameters) to find the price of the option at the exercise point, where the scheme was found to yield

$$\boxed{V(r_0,\ 0;\ T_1,\ T) \approx 0.06200113893777939}$$

as the price of the American call option.

2.3.1 Further Analysis | Task 2.2

In order to explore the importance of the extent of numerical discretisations and specifically the j index as it plays a key role in the PSOR modification and the baseline Crank-Nicolson scheme, the maximum index, j_{Max}, was varied, and the resulting changes in valuation of the previously discussed American call option was plotted in Figure 4.

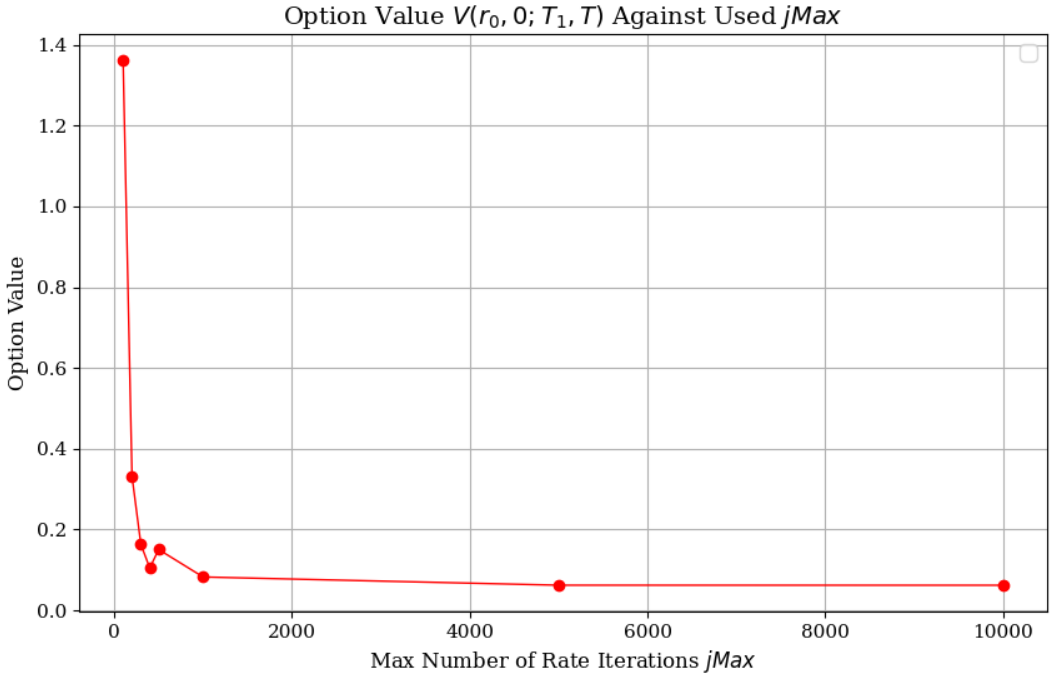

Figure 4: Values of the given American call option $V(r_0, t; T_1, T)$ against the rate discretisation j_{Max}.

The trend shown in Figure 4 reveals a significant relationship between the extent of numerical discretisation (in the interest rate), j_{Max}, and the calculated option values. The increases in j_{Max} lead to significant

changes in the option valuation, however this trend quickly saturates, such that after a certain degree of finer discretisation, increases in j_{Max} yield *diminishing returns* in valuation accuracy, and the option value plateaus near the value given in section 2.3. It is therefore paramount in applications of computational finance that a balance is maintained between computational efficiency and the desired accuracy.

3 Appendix | Assignment Code

3.1 Code for Tasks 2.1 and 2.2

```
# Importing libraries
import math
import numpy as np
import scipy
from scipy.stats import norm
import matplotlib.pyplot as plt
from scipy.linalg import solve_banded
from math import exp

# Latex style for plots
plt.rcParams.update({
    "text.usetex": False,
    "font.family": "serif",
    "axes.labelsize": 12,
    "axes.titlesize": 14,
    "legend.fontsize": 12,
    "xtick.labelsize": 11,
    "ytick.labelsize": 11
})

rng = np.random.default_rng(seed=123)

### Finite Difference Methods Assignment
## Task 2.1 Bonds
# Part b

# General information and market-fitted parameters
T = 3  # Maturity time
F = 81
theta = 0.0262
r0 = 0.0381
kappa = 0.08169
mu = 0.015
C = 2.58
alpha = 0.02
beta = 0.413
sigma = 0.111

# Crank-Nicolson method for the bond pricing PDE
# Crank-Nicolson Scheme Parameters

iMax = 100 # max number of time steps
```

```
jMax = 100 # max number of space steps

rMax = 1.0 # max interest rate
dr = rMax / jMax  # r step size
dt = T / iMax # time step size

# Numpy arrays for storing values
r = np.zeros(jMax+1)
t = np.zeros(iMax+1)
B_new = np.zeros(jMax+1)
B_old = np.zeros(jMax+1)

for i in range(iMax+1):
    t[i] = i*dt

for j in range(jMax+1):
    r[j] = j*dr

# Record the value of the bond at maturity
B_old[:] = F

# Matrix solution for the Crank-Nicolson scheme
# Storage for A (tridiagonal matrix)
A_bands = np.zeros(shape=(3,jMax+1))

band_structure = (1, 1) # (lower_bands, upper_bands)

# Allocate storage for the RHS term (d)
d = np.zeros(jMax+1)

# Loop over time steps
for i in range(iMax-1, -1, -1):

    # Fill in the tridiagonal matrix A
    # Clear the A_bands array for each time step
    A_bands.fill(0.0)

    # Special case for j = 0 (PDE boundary condition at r = 0)
    common_part_bc = (kappa * theta * ( np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)))/(2.0*dr)
    A_bands[1,0] = -1.0/dt - common_part_bc # b_0
    A_bands[0,1] =  common_part_bc # c_0

    # a[j], b[j], c[j] for matrix middle rows
    for j in range(1, jMax):

        # Terms for convenience
        j_term = (j**(2*beta)) * (dr**(2*beta - 2))
        k_part = kappa * (theta * (np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr)

        # a_j B_j-1 + b_j B_j + c_i B_j+1 = d_i
        A_bands[2, j-1] = 0.25 * sigma**2 * j_term  -  (k_part / (4*dr)) # a_j
```

```
        A_bands[1, j] = -1.0/dt  - 0.5 * sigma**2 * j_term  - 0.5 * j * dr # b_j
        A_bands[0, j+1] = 0.25 * sigma**2 * j_term  +  (k_part / (4*dr)) # c_j


    # Boundary condition at for j = 0 jMax
    # a_jMax B_jMax-1 + b_jMax B_jMax = d_jMax
    A_bands[2,jMax-1] = 0.0 # a_jMax
    A_bands[1,jMax] = 1.0 # b_jMax

    # Fill in the RHS term (d) for the Crank-Nicolson scheme
    # Clear the d array for each time step
    d.fill(0.0)

    # Common term
    #exp_term = 0.5*C*np.exp(-alpha*t[i]) + 0.5*C*np.exp(-alpha*t[i+1])
    exp_term = 0.5*C*np.exp(-alpha*(t[i]+0.5*dt))

    # Special case for j = 0 (PDE boundary condition at r = 0)
    c0 = (kappa*theta*(np.exp(mu*t[i]) + np.exp(mu*t[i+1])))/(2.0*dr)
    d[0] = (-1.0/dt + c0) * B_old[0] - c0 * B_old[1] - exp_term

    # Case for 1 <= j < jMax
    for j in range(1, jMax):
        aa = A_bands[2, j-1]
        bb = A_bands[1, j] + 2.0/dt
        cc = A_bands[0, j+1]

        d[j] = -aa * B_old[j-1] - bb * B_old[j] - cc * B_old[j+1] - exp_term

    # Case for j = jMax
    d[jMax] = 0.0 # d_jMax

    # Solve the equation
    B_new = solve_banded(band_structure, A_bands, d)
    B_old = np.copy(B_new)

print(B_new)

B_r0 = np.interp(r0, r, B_new)
print("Bond value at r0, t=0  :", B_r0)


# Part c
# Function to compute the bond value using the Crank-Nicolson method with Dirichlet
boundary conditions
def crank_dirichlet(A_bands, d, band_structure, B_old):
    for i in range(iMax-1, -1, -1):

        # Fill in the tridiagonal matrix A
        A_bands.fill(0.0)

```

```
        # Special case for j = 0 (PDE boundary condition at r = 0)
        common_part_bc = (kappa * theta *
                          (np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt))) \
                         / (2.0*dr)
        A_bands[1,0] = -1.0/dt - common_part_bc    # b_0
        A_bands[0,1] =  common_part_bc             # c_0

        # a[j], b[j], c[j] for matrix middle rows
        for j in range(1, jMax):
            j_term = (j**(2*beta)) * (dr**(2*beta - 2))
            k_part = kappa * (
                theta*(np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr
            )
            A_bands[2,j-1] = 0.25*sigma**2*j_term - k_part/(4*dr)  # a_j
            A_bands[1,j  ] = -1.0/dt - 0.5*sigma**2*j_term - 0.5*j*dr  # b_j
            A_bands[0,j+1] = 0.25*sigma**2*j_term + k_part/(4*dr)  # c_j

        # Boundary condition at for j = 0 jMax
        # a_jMax B_jMax-1 + b_jMax B_jMax = d_jMax
        A_bands[2,jMax-1] = 0.0   # a_jMax
        A_bands[1,jMax  ] = 1.0   # b_jMax

        # Fill in the RHS term (d) for the Crank-Nicolson scheme
        d.fill(0.0)
        #exp_term = 0.5*C*np.exp(-alpha*t[i]) + 0.5*C*np.exp(-alpha*t[i+1])
        exp_term = 0.5*C*np.exp(-alpha*(t[i]+0.5*dt))

        c0 = common_part_bc
        d[0] = (-1.0/dt + c0)*B_old[0] - c0*B_old[1] - exp_term

        for j in range(1, jMax):
            aa = A_bands[2,j-1]
            bb = A_bands[1,j] + 2.0/dt
            cc = A_bands[0,j+1]
            d[j] = -aa*B_old[j-1] - bb*B_old[j] - cc*B_old[j+1] - exp_term

        d[jMax] = 0.0   # d_jMax

        # Solve the equation
        B_old = solve_banded(band_structure, A_bands, d)

    return B_old

# Function to compute the bond value using the Crank-Nicolson method with Neumann
boundary conditions
def crank_neumann(A_bands, d, band_structure, B_old):
    for i in range(iMax-1, -1, -1):

        # Fill in the tridiagonal matrix A
        A_bands.fill(0.0)

```

```
        # Special case for j = 0 (PDE boundary condition at r = 0)
        common_part_bc = (kappa * theta *
                          (np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt))) \
                         / (2.0*dr)
        A_bands[1,0] = -1.0/dt - common_part_bc
        A_bands[0,1] =  common_part_bc

        # a[j], b[j], c[j] for matrix middle rows
        for j in range(1, jMax):
            j_term = (j**(2*beta)) * (dr**(2*beta - 2))
            k_part = kappa * (
                theta*(np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr
            )
            A_bands[2,j-1] = 0.25*sigma**2*j_term - k_part/(4*dr)
            A_bands[1,j  ] = -1.0/dt - 0.5*sigma**2*j_term - 0.5*j*dr
            A_bands[0,j+1] = 0.25*sigma**2*j_term + k_part/(4*dr)

        # Neumann row: -B_{jMax-1} + B_{jMax} = 0
        A_bands[2,jMax-1] = -1.0   # a_jMax
        A_bands[1,jMax  ] =  1.0   # b_jMax

        # Fill in the RHS term (d) for the Crank-Nicolson scheme
        d.fill(0.0)
        #exp_term = 0.5*C*np.exp(-alpha*t[i]) + 0.5*C*np.exp(-alpha*t[i+1])
        exp_term = 0.5*C*np.exp(-alpha*(t[i]+0.5*dt))

        c0 = common_part_bc
        d[0] = (-1.0/dt + c0)*B_old[0] - c0*B_old[1] - exp_term

        for j in range(1, jMax):
            aa = A_bands[2,j-1]
            bb = A_bands[1,j] + 2.0/dt
            cc = A_bands[0,j+1]
            d[j] = -aa*B_old[j-1] - bb*B_old[j] - cc*B_old[j+1] - exp_term

        d[jMax] = 0.0   # d_jMax

        # Solve the equation
        B_old = solve_banded(band_structure, A_bands, d)

    return B_old

# Runs for Comparison Plots

iMax = 100 # max number of time steps
jMax = 100 # max number of space steps

rMax = 1.0 # max interest rate
dr = rMax / jMax  # r step size
dt = T / iMax # time step size

```

```
# Numpy arrays for storing values
r = np.zeros(jMax+1)
t = np.zeros(iMax+1)
B_new = np.zeros(jMax+1)
B_old = np.zeros(jMax+1)

for i in range(iMax+1):
    t[i] = i*dt

for j in range(jMax+1):
    r[j] = j*dr

A_bands = np.zeros(shape=(3,jMax+1))

band_structure = (1, 1) # (lower_bands, upper_bands)

# Allocate storage for the RHS term (d)
d = np.zeros(jMax+1)

# Dirichlet run:
B_old[:] = F
B_dirich = crank_dirichlet(A_bands, d, band_structure, B_old.copy())

B_r0_Dirichlet = np.interp(r0, r, B_dirich)
print("Bond value at r0, t=0  Dirichlet:", B_r0_Dirichlet)

# Neumann run (start again from maturity):
B_old[:] = F
B_neum  = crank_neumann(A_bands, d, band_structure, B_old.copy())

B_r0_Neumann = np.interp(r0, r, B_neum)
print("Bond value at r0, t=0  Neumann:", B_r0_Neumann)

# Plotting the results, comparing Dirichlet and Neumann (0 to rMax)

plt.figure(figsize=(10, 6))
plt.plot(r, B_dirich, label='Dirichlet', color='blue', linestyle='--')
plt.plot(r, B_neum, label='Neumann', color='red')
plt.title('Bond Value Comparison: Dirichlet vs Neumann')
plt.xlabel('Interest Rate (r)')
plt.ylabel('Bond Value')
plt.legend()
plt.grid()
plt.xlim(0, rMax)
#plt.ylim(0, 100)
plt.show()

# Code for Extra
# Neumann
rMax = 5.0
dr = rMax/jMax
```

```
dt = T/iMax
r_neu = np.linspace(0, rMax, jMax+1)
B_old = np.full(jMax+1, F)
A_bands = np.zeros((3, jMax+1))
d = np.zeros(jMax+1)
B_neu = crank_neumann(A_bands, d, (1,1), B_old)

# Dirichlet rMax=1
rMax = 1.0
dr = rMax/jMax
r1 = np.linspace(0, rMax, jMax+1)
B_old = np.full(jMax+1, F)
B_dir1 = crank_dirichlet(A_bands, d, (1,1), B_old)
B_dir1_i = np.interp(r_neu, r1, B_dir1)

# Dirichlet rMax=2
rMax = 2.0
dr = rMax/jMax
r2 = np.linspace(0, rMax, jMax+1)
B_old = np.full(jMax+1, F)
B_dir2 = crank_dirichlet(A_bands, d, (1,1), B_old)
B_dir2_i = np.interp(r_neu, r2, B_dir2)

# Dirichlet rMax=3
rMax = 3.0
dr = rMax/jMax
r3 = np.linspace(0, rMax, jMax+1)
B_old = np.full(jMax+1, F)
B_dir3 = crank_dirichlet(A_bands, d, (1,1), B_old)
B_dir3_i = np.interp(r_neu, r3, B_dir3)

# Dirichlet rMax=4
rMax = 4.0
dr = rMax/jMax
r4 = np.linspace(0, rMax, jMax+1)
B_old = np.full(jMax+1, F)
B_dir4 = crank_dirichlet(A_bands, d, (1,1), B_old)
B_dir4_i = np.interp(r_neu, r4, B_dir4)

# Dirichlet rMax=5
rMax = 5.0
dr = rMax/jMax
r5 = np.linspace(0, rMax, jMax+1)
B_old = np.full(jMax+1, F)
B_dir5 = crank_dirichlet(A_bands, d, (1,1), B_old)
B_dir5_i = np.interp(r_neu, r5, B_dir5)

plt.figure(figsize=(10, 6))
plt.plot(r_neu, B_neu, label='Neumann', color='red', linewidth=3, alpha=0.7)
plt.plot(r_neu, B_dir1_i, label='Dirichlet', color='blue', linestyle='--', linewidth=1)
plt.plot(r_neu, B_dir2_i, color='blue', linestyle='--', linewidth=1)
```

```
plt.plot(r_neu, B_dir3_i, color='blue', linestyle='--', linewidth=1)
plt.plot(r_neu, B_dir4_i, color='blue', linestyle='--', linewidth=1)
plt.plot(r_neu, B_dir5_i, color='blue', linestyle='--', linewidth=1)
plt.title('Bond Value Comparison: Dirichlet vs Neumann, Varying Dirichlet rMax')
plt.xlabel('Interest Rate (r)')
plt.ylabel('Bond Value')
plt.legend()
plt.grid()
plt.xlim(0, 4)
plt.ylim(0, 20)
plt.show()


############################################################

## Task 2.2 Options on Bonds (American Call Option)
# Part b

# General information and market-fitted parameters
T = 3  # Maturity time
T1 = 1.0729 # Option expiration time
F = 81
X = 82
theta = 0.0262
r0 = 0.0381
kappa = 0.08169
mu = 0.015
C = 2.58
alpha = 0.02
beta = 0.413
sigma = 0.111

# Crank-Nicolson method for the bond pricing PDE
# Crank-Nicolson Scheme Parameters

iMax = 1000 # max number of time steps
jMax = 10000 # max number of space steps
kMax = 1000   # max number of PSOR relaxations
omega = 1.2  # over-relaxation parameter
tol = 1e-6 # tolerance for convergence

rMax = 5.0 # max interest rate
dr = rMax / jMax  # r step size
dt = T / iMax # time step size

# Numpy arrays for storing values
r = np.zeros(jMax+1)
t = np.zeros(iMax+1)
B_new = np.zeros(jMax+1)
B_old = np.zeros(jMax+1)

```

```
V_new = np.zeros(jMax+1)
V_old = np.zeros(jMax+1)

#V_old[j] = max(B_old[j] - X, 0.0)

for i in range(iMax+1):
    t[i] = i*dt

for j in range(jMax+1):
    r[j] = j*dr

# Record the value of the bond at maturity
B_old[:] = F

V_old[:] = np.maximum(B_old - X, 0.0)

# Matrix solution for the Crank-Nicolson scheme
# Storage for A (tridiagonal matrix)
A_bands = np.zeros(shape=(3,jMax+1))

band_structure = (1, 1) # (lower_bands, upper_bands)

# Allocate storage for the (bond) RHS term (d)
d = np.zeros(jMax+1)

###

for i in range(iMax-1, -1, -1):

    # Fill in the tridiagonal matrix A
    A_bands.fill(0.0)

    # Special case for j = 0 (PDE boundary condition at r = 0)
    common_part_bc = (kappa * theta *
                      (np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt))) \
                      / (2.0*dr)
    A_bands[1,0] = -1.0/dt - common_part_bc    # b_0
    A_bands[0,1] =  common_part_bc             # c_0

    # a[j], b[j], c[j] for matrix middle rows
    for j in range(1, jMax):
        j_term = (j**(2*beta)) * (dr**(2*beta - 2))
        k_part = kappa * (theta*(np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr)
        A_bands[2,j-1] = 0.25*sigma**2*j_term - k_part/(4*dr)  # a_j
        A_bands[1,j  ] = -1.0/dt - 0.5*sigma**2*j_term - 0.5*j*dr  # b_j
        A_bands[0,j+1] = 0.25*sigma**2*j_term + k_part/(4*dr)  # c_j

    # Boundary condition at for j = 0 jMax
    # a_jMax B_jMax-1 + b_jMax B_jMax = d_jMax
    A_bands[2,jMax-1] = 0.0   # a_jMax
    A_bands[1,jMax  ] = 1.0   # b_jMax
```

```
449
450         # Fill in the RHS term (d) for the Crank-Nicolson scheme
451         d.fill(0.0)
452         #exp_term = 0.5*C*np.exp(-alpha*t[i]) + 0.5*C*np.exp(-alpha*t[i+1])
453         exp_term = 0.5*C*np.exp(-alpha*(t[i]+0.5*dt))
454
455         c0 = common_part_bc
456         d[0] = (-1.0/dt + c0)*B_old[0] - c0*B_old[1] - exp_term
457
458         for j in range(1, jMax):
459             aa = A_bands[2,j-1]
460             bb = A_bands[1,j] + 2.0/dt
461             cc = A_bands[0,j+1]
462             d[j] = -aa*B_old[j-1] - bb*B_old[j] - cc*B_old[j+1] - exp_term
463
464         d[jMax] = 0.0   # d_jMax
465
466         # Solve the equation
467         B_new = solve_banded(band_structure, A_bands, d)
468         B_old = np.copy(B_new)
469
470         # Update the option value using the bond value
471
472         if i == int(T1/dt):
473             V_old = np.maximum(B_old - X, 0.0)   # Update V_old with the exercise values
474
475         # excercise values
476         E = np.maximum(B_old - X, 0.0)
477
478         V_new = V_old.copy()
479
480         a_opt = np.zeros(jMax+1)
481         b_opt = np.zeros(jMax+1)
482         c_opt = np.zeros(jMax+1)
483         d_opt = np.zeros(jMax+1)
484
485         # Special case for j = 0
486         a_opt[0] = 0.0
487         b_opt[0] = 1.0
488         c_opt[0] = 0.0
489         d_opt[0] = B_old[0] - X
490
491         # a[j], b[j], c[j] for matrix middle rows
492         for j in range(1, jMax):
493             j_term = (j**(2*beta)) * (dr**(2*beta - 2))
494             k_part = kappa * (theta*(np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr)
495
496             a_opt[j] = 0.25*sigma**2*j_term - k_part/(4*dr)
497             b_opt[j] = -1.0/dt - 0.5*sigma**2*j_term - 0.5*j*dr
498             c_opt[j] = 0.25*sigma**2*j_term + k_part/(4*dr)
```

```
499             d_opt[j] = (-a_opt[j]*V_old[j-1]+(-1.0/dt + 0.5*sigma**2*j_term +
                0.5*j*dr)*V_old[j]-c_opt[j]*V_old[j+1])
500
501         # Boundary condition at for j = jMax
502         a_opt[jMax] = 0.0
503         b_opt[jMax] = 1.0
504         c_opt[jMax] = 0.0
505         d_opt[jMax] = 0.0
506
507
508         if i <= int(T1/dt):
509             # Loop for PSOR method
510             for k in range(kMax):
511                 epsilon = 0.0 # convergence parameter
512
513                 y = (1/b_opt[0]) * (d_opt[0] - c_opt[0]*V_new[1])
514                 y = V_new[0] + omega * (y - V_new[0])
515                 y = np.maximum(y, B_old[0] - X)
516
517                 epsilon += np.square(y - V_new[0])
518                 V_new[0] = y
519
520                 # Matrix middle rows
521                 for j in range(1, jMax):
522                     y = (1/b_opt[j]) * (d_opt[j] - a_opt[j]*V_new[j-1] - c_opt[j]*V_new[j+1])
523                     y = V_new[j] + omega * (y - V_new[j])
524                     y = np.maximum(y, B_old[j] - X)
525
526                     epsilon += np.square(y - V_new[j])
527                     V_new[j] = y
528
529                 # For j = jMax
530                 y = (1/b_opt[jMax]) * (d_opt[jMax] - a_opt[jMax]*V_new[jMax-1])
531                 y = V_new[jMax] + omega * (y - V_new[jMax])
532                 y = np.maximum(y, 0.0)
533
534                 epsilon += np.square(y - V_new[jMax])
535                 V_new[jMax] = y
536
537                 # Check for convergence
538                 if (epsilon < tol**2):
539                     break
540
541                 # Save values for plot
542             if i == int(T1/dt):
543                 V_at_T1 = V_new.copy()
544                 B_at_T1 = B_new.copy()
545
546             V_old = V_new.copy()
547
548 V_at_0 = V_old.copy()
```

```
549
550 tol = 1e-10
551 # Find maximum r for which the option is exercised
552 max_ex_index = np.maximum(B_at_T1 - X, 0.0)
553 proximity_to_exercise = V_at_T1 - max_ex_index
554 j_indices_max_optimal = np.where((max_ex_index > 0) & (proximity_to_exercise <
    tol))[0].max() # to basically ignore irrelevant indices of i
555 r_indices_max_optimal = r[j_indices_max_optimal]
556
557 ###
558 # Plot for V_at_T1 and V_at_0 against r
559 plt.figure(figsize=(10, 6))
560 plt.plot(r, V_at_T1, label=f'$V(r,\,t=T_1)$')
561 plt.plot(r, V_at_0,  label='$V(r,\,t=0)$')
562 plt.title('Value of American Call on Bond: $V$ vs $r$ at $t=T_1$ and $t=0$')
563 plt.xlabel('Interest Rate ($r$)')
564 plt.ylabel('American Call Value ($V$)')
565 plt.legend()
566 plt.grid()
567 #plt.xlim(0, rMax)
568 #plt.xlim(0, 0.1)
569 #plt.ylim(0, 100)
570 plt.show()
571
572
573 # Print the maximum r for which the option is exercised
574 print("Max r for which option is exercised:", r_indices_max_optimal)
575
576 # Printing the option values at r0
577 print("Option value at r0, t=0  :", V_at_0[int(r0/dr)])
578 print("Option value at r0, t=T1 :", V_at_T1[int(r0/dr)])
579
580 # Extra in 2.2
581 # Function for Task 2.2 Extra
582 def Crank_PSOR_American(iMax, jMax, T, T1, F, X, theta, r0, kappa, mu, C, alpha, beta,
    sigma):
583     # Numpy arrays for storing values
584     r = np.zeros(jMax+1)
585     t = np.zeros(iMax+1)
586     B_new = np.zeros(jMax+1)
587     B_old = np.zeros(jMax+1)
588
589     V_new = np.zeros(jMax+1)
590     V_old = np.zeros(jMax+1)
591
592     #V_old[j] = max(B_old[j] - X, 0.0)
593
594     for i in range(iMax+1):
595         t[i] = i*dt
596
597     for j in range(jMax+1):
```

```
        r[j] = j*dr

    # Record the value of the bond at maturity
    B_old[:] = F

    V_old[:] = np.maximum(B_old - X, 0.0)

    # Matrix solution for the Crank-Nicolson scheme
    # Storage for A (tridiagonal matrix)
    A_bands = np.zeros(shape=(3,jMax+1))

    band_structure = (1, 1) # (lower_bands, upper_bands)

    # Allocate storage for the (bond) RHS term (d)
    d = np.zeros(jMax+1)

    ###

    for i in range(iMax-1, -1, -1):

        # Fill in the tridiagonal matrix A
        A_bands.fill(0.0)

        # Special case for j = 0 (PDE boundary condition at r = 0)
        common_part_bc = (kappa * theta *
                          (np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt))) \
                          / (2.0*dr)
        A_bands[1,0] = -1.0/dt - common_part_bc    # b_0
        A_bands[0,1] =  common_part_bc             # c_0

        # a[j], b[j], c[j] for matrix middle rows
        for j in range(1, jMax):
            j_term = (j**(2*beta)) * (dr**(2*beta - 2))
            k_part = kappa * (theta*(np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr)
            A_bands[2,j-1] = 0.25*sigma**2*j_term - k_part/(4*dr)  # a_j
            A_bands[1,j  ] = -1.0/dt - 0.5*sigma**2*j_term - 0.5*j*dr  # b_j
            A_bands[0,j+1] = 0.25*sigma**2*j_term + k_part/(4*dr)  # c_j

        # Boundary condition at for j = 0 jMax
        # a_jMax B_jMax-1 + b_jMax B_jMax = d_jMax
        A_bands[2,jMax-1] = 0.0   # a_jMax
        A_bands[1,jMax  ] = 1.0   # b_jMax

        # Fill in the RHS term (d) for the Crank-Nicolson scheme
        d.fill(0.0)
        #exp_term = 0.5*C*np.exp(-alpha*t[i]) + 0.5*C*np.exp(-alpha*t[i+1])
        exp_term = 0.5*C*np.exp(-alpha*(t[i]+0.5*dt))

        c0 = common_part_bc
        d[0] = (-1.0/dt + c0)*B_old[0] - c0*B_old[1] - exp_term
```

```
        for j in range(1, jMax):
            aa = A_bands[2,j-1]
            bb = A_bands[1,j] + 2.0/dt
            cc = A_bands[0,j+1]
            d[j] = -aa*B_old[j-1] - bb*B_old[j] - cc*B_old[j+1] - exp_term

        d[jMax] = 0.0   # d_jMax

        # Solve the equation
        B_new = solve_banded(band_structure, A_bands, d)
        B_old = np.copy(B_new)

        # Update the option value using the bond value

        if i == int(T1/dt):
            V_old = np.maximum(B_old - X, 0.0)   # Update V_old with the exercise values

        # excercise values
        E = np.maximum(B_old - X, 0.0)

        V_new = V_old.copy()

        a_opt = np.zeros(jMax+1)
        b_opt = np.zeros(jMax+1)
        c_opt = np.zeros(jMax+1)
        d_opt = np.zeros(jMax+1)

        # Special case for j = 0
        a_opt[0] = 0.0
        b_opt[0] = 1.0
        c_opt[0] = 0.0
        d_opt[0] = B_old[0] - X

        # a[j], b[j], c[j] for matrix middle rows
        for j in range(1, jMax):
            j_term = (j**(2*beta)) * (dr**(2*beta - 2))
            k_part = kappa * (theta*(np.exp(mu*i*dt) + np.exp(mu*(i+1)*dt)) - j*dr)

            a_opt[j] = 0.25*sigma**2*j_term - k_part/(4*dr)
            b_opt[j] = -1.0/dt - 0.5*sigma**2*j_term - 0.5*j*dr
            c_opt[j] = 0.25*sigma**2*j_term + k_part/(4*dr)
            d_opt[j] = (-a_opt[j]*V_old[j-1]+(-1.0/dt + 0.5*sigma**2*j_term +
            0.5*j*dr)*V_old[j]-c_opt[j]*V_old[j+1])

        # Boundary condition at for j = jMax
        a_opt[jMax] = 0.0
        b_opt[jMax] = 1.0
        c_opt[jMax] = 0.0
        d_opt[jMax] = 0.0
```

```
        if i <= int(T1/dt):
            # Loop for PSOR method
            for k in range(kMax):
                epsilon = 0.0 # convergence parameter

                y = (1/b_opt[0]) * (d_opt[0] - c_opt[0]*V_new[1])
                y = V_new[0] + omega * (y - V_new[0])
                y = np.maximum(y, B_old[0] - X)

                epsilon += np.square(y - V_new[0])
                V_new[0] = y

                # Matrix middle rows
                for j in range(1, jMax):
                    y = (1/b_opt[j]) * (d_opt[j] - a_opt[j]*V_new[j-1] -
                    c_opt[j]*V_new[j+1])
                    y = V_new[j] + omega * (y - V_new[j])
                    y = np.maximum(y, B_old[j] - X)

                    epsilon += np.square(y - V_new[j])
                    V_new[j] = y

                # For j = jMax
                y = (1/b_opt[jMax]) * (d_opt[jMax] - a_opt[jMax]*V_new[jMax-1])
                y = V_new[jMax] + omega * (y - V_new[jMax])
                y = np.maximum(y, 0.0)

                epsilon += np.square(y - V_new[jMax])
                V_new[jMax] = y

                # Check for convergence
                if (epsilon < tol**2):
                    break

                # Save values for plot
            if i == int(T1/dt):
                V_at_T1 = V_new.copy()
                B_at_T1 = B_new.copy()

            V_old = V_new.copy()

    V_at_0 = V_old.copy()

    return V_at_0

# Plotting the option value for varying jMax

rMax = 5.0
j_vals = [100, 200, 300, 400, 500, 1000, 5000, 10000]
V_list = []
plt.figure(figsize=(10, 6))
```

```
749
750 for jMax in j_vals:
751     dr = rMax/jMax
752     dt = T/iMax
753
754     V_at_0_ = Crank_PSOR_American(iMax, jMax, T, T1, F, X, theta, r0, kappa, mu, C,
        alpha, beta, sigma)
755     V_list.append(V_at_0_[int(r0/dr)])
756
757 plt.plot(j_vals, V_list, marker='o', linewidth=1, color='red')
758
759 plt.title('Option Value $V(r_0,0;T_1,T)$ Against Used $jMax$')
760 plt.xlabel('Max Number of Rate Iterations $jMax$')
761 plt.ylabel('Option Value')
762 #plt.legend()
763 plt.grid()
764 #plt.xlim(0, 5)
765 #plt.ylim(0, 20)
766 plt.show()
```