Final Year Project Report: Solving the Black-Scholes Equation Numerically

Name: Abiola Victor Akinsowon

Student Number: 20424972

Email Address: victor.akinsowon2@mail.dcu.ie

Programme: AP4

Supervisor: Assoc. Prof. Lampros Nikolopoulos

10 April 2024

Declaration

Name: Abiola Victor Akinsowon

Student ID Number: 20424972

Programme: AP4

Module Code: PS451

Assignment Title: Solving the Black-Scholes Equation Numerically

Submission Date: 10 April 2024

I understand that the University regards breaches of academic integrity and plagiarism as grave and serious. I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references. I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. I have used the DCU library referencing guidelines (available at: http://www.library.dcu.ie/LibraryGuides/Citing& ReferencingGuide/player.html) and/or the appropriate referencing system recommended in the assignment guidelines and/or programme documentation. By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy (available at: http://www.dcu.ie/registry/examinations/index.shtml).

Name: Abiola Victor Akinsowon

Date: 10/04/2024

Student Name:	Abiola Victor Akinsowon
Student Number:	20424972
Project Title:	Solving the Black-Scholes Equation Numerically
Main Project Working Location:	Dublin City University, Collins Ave Ext, Whitehall, Dublin 9
Supervisor Name:	Lampros Nikolopoulos
Brief Listing of All Risks Associated with Project:	None

Signature page
Student: Abiola Victor Akinsowon

I have attempted, to the best of my ability, to assess the risks associated with my project and to suggest appropriate controls and other measures to reduce the risk level. I have also consulted with my supervisor in performing and completing this risk assessment. I will implement these controls and other measures to reduce the risk level consistently throughout my project and make my supervisor aware if at any stage I feel the current risk assessment is insufficient as the project evolves*.

Student Name: Abiola Victor Akinsowon

Student number: 20424972

Signature: Submission of this document is equivalent to a signature.

Date: 10/04/2024

Supervisor: Lampros Nikolopoulos

I confirm that the student has undertaken a risk assessment of their project work and that they have consulted with me on this matter. Furthermore, the assessment above appears thorough and I will inform the student if at any stage I feel the current risk assessment is insufficient as the project evolves**.

Supervisor Name: Lampros Nikolopoulos

Signature: Submission of this document is contingent upon supervisor's approval and implies it has been sort and approved.

Date: 10/04/2024

Abstract

This study investigates the application of numerical methods to solve the Time-Dependent Schrödinger Equation and the Black-Scholes Equation. The Crank-Nicolson Method, chosen for its accuracy and computational efficiency, is implemented using C++ programming language, with a focus on matrix-based approaches and the elimination of matrix-vector operations. Comparative analysis reveals that as discretization refines, numerical solutions converge towards analytical solutions, with both CN-I and CN-II variants yielding identical results. Transitioning to the Black-Scholes Equation, the numerical model accurately prices options, validated through comparison with external solutions. Looking forward, recommendations include further exploration of advanced numerical techniques, integration of real-time stock market data for dynamic option pricing, and collaboration with experts in finance and computational science for innovative solutions with broader applications.

Acknowledgments

I extend my sincere gratitude to Assoc. Prof. Lampros Nikolopoulos for their invaluable supervision, mentorship, and guidance throughout this project. His expertise and support have been instrumental in shaping and facilitating the progress of this research. I am deeply grateful for the wealth of knowledge and insights shared by Lampros Nikolopoulos, which have greatly contributed to the completion of this report.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The convergence of financial mathematics and quantum physics presents numerous avenues for exploration and understanding. At the heart of this convergence lies the Black-Scholes equation (BSE), celebrated in finance for its effectiveness in pricing options. However, this endeavor pushes the boundaries by revealing profound connections between the Black-Scholes equation and the Time-Dependent Schrödinger Equation (TDSE). As a partial differential equation governing the diffusion process of option prices[1], the Black-Scholes equation exhibits striking similarities with the stochastic nature of the TDSE. The necessity to delve into this realm stems from the recognition that traditional approaches may only scratch the surface of its mathematical essence. This similarity underscores the intricate relationship between financial dynamics and quantum principles, prompting a comprehensive exploration of their shared mathematical foundations.

The general form of the Black-Scholes equation is[2]

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{1.1}$$

where V represents the option price, t denotes time, S denotes the underlying asset price, σ represents the volatility of the asset returns and r denotes the risk-free interest rate.

The Time-Dependent Schrödinger Equation is a fundamental equation in quantum mechanics that describes the time evolution of a quantum state. It is a partial differential equation, describing how the wave function of a quantum system changes with time and space coordinates. The Schrödinger Equation takes the general form [3]:

$$i\hbar\frac{\partial \psi(x,t)}{\partial t} = H\psi(x,t) \tag{1.2}$$

where $\psi(x,t)$ represents the wavefunction and H denotes the Hamiltonian operator. This operator is pivotal in describing the measurable properties of a physical system and encompasses operators associated with both kinetic (T) and potential (V) energies, expressed as

$$H = T + V = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x,t) \tag{1.3}$$

These equations, both partial differential in nature, can be adapted to illustrate diffusion phenomena[1], highlighting the interconnection between financial mathematics and quantum physics, fostering new methods for tackling the Black-Scholes equation and enhancing interdisciplinary comprehension. Now that we've introduced the concept of the Black-Scholes equation and highlighted its similarities with the Time-Dependent Schrödinger Equation, it's essential to delve deeper into the theoretical underpinnings.

In Section 2, we will establish a foundational understanding of the modeling and pricing of options, grasp the intricacies of the Black-Scholes Equation and explore several numerical methods to solve the Time-Dependent Schrödinger Equation. These numerical methods will serve as the basis for our approach in Section 3, where we will apply similar techniques to solve the Black-Scholes Equation.

1.1 Aims and Objectives

The primary objectives of the project were as follows:

- Conduct a comprehensive study of the Black-Scholes Equation to gain a thorough understanding of its variables and underlying principles.
- Employ Crank-Nicolson methods to numerically solve the Time-Dependent Schrödinger Equation, a pivotal step in bridging the gap between financial mathematics and quantum physics.
- Develop and implement numerical algorithms tailored for solving the Black-Scholes Equation, aiming to provide accurate and efficient solutions for various financial scenarios.

2 Theory

2.1 Derivative Instruments in Finance: Modeling and Pricing

2.1.1 The Role of Derivatives in Finance

Expanding on our exploration of financial mathematics and its parallels with quantum physics, it's evident that derivatives are instrumental components within the realm of finance. Derivatives serve dual purposes in finance: risk management and speculation. These financial instruments derive their value from an underlying asset[4], enabling investors to either hedge against market volatility or capitalize on price movements. In financial markets, the spot price ($S(t)$) reflects the immediate value of traded assets, influenced by the interplay of supply and demand. Navigating this uncertainty, traders aim for returns beyond traditional banking options while effectively managing associated risks. Asset price evolution is often modeled using Geometric Brownian motion, which encompasses both deterministic growth (drift) and stochastic fluctuations (volatility). This framework provides a nuanced comprehension of price changes, emphasizing proportional shifts rather than absolute values—an essential aspect for option pricing. Options, a type of derivative contract, grant the buyer the right, but not the obligation, to buy or sell an underlying asset at a predetermined price within a specified time frame. In exchange for this right, the buyer pays a premium to the seller, also known as the writer or issuer. Options serve as powerful tools for both mitigating risk and engaging in speculation, offering avenues to address uncertainties in the market. Option pricing, exemplified by American call options, depends on various factors such as the asset's current value (S), the strike price (K), and the time until expiration (T). This intricate relationship forms the foundation for determining option prices in financial markets. Specifically, call options provide the holder the right to purchase an asset at a predetermined price (the strike price) on or before the expiration date, while put options offer the right to sell the asset at the strike price within the same timeframe.

2.1.2 Black-Scholes Equation Assumptions

The Black-Scholes Equation, a fundamental formula for pricing options contracts, establishes equitable option prices based on these parameters, enabling fair transactions in financial markets. The derivation of the Black-Scholes equation relies on simplifying assumptions to navigate the complexities of real-world financial markets. These assumptions include[2]:

1. **No Credit Risk, Only Market Risk**: The option price is solely influenced by market fluctuations of the underlying asset, without considering the reliability of the counterparty.

2. **Maximally Efficient Market**: The market is infinitely liquid and frictionless, with all relevant information instantaneously and comprehensively available. Transactions can be executed at any time without additional costs.

3. **Continuous Trading**: Trading occurs continuously, with the time interval between successive quotations of the underlying asset's price tending to zero.

4. **Stochastic Asset Price Evolution**: Asset prices follow geometric Brownian motion, characterized by deterministic growth (drift) and stochastic fluctuations (volatility).

   The geometric Brownian motion equation is given by:

   $$dS = \mu S(t)dt + \sigma S(t)dW(t) \tag{2.1}$$

   where $S(t)$ is the spot price of the asset at time t, μ is the drift coefficient representing the average growth rate of the asset price, σ is the volatility coefficient, which measures the magnitude of statistical price fluctuations and $dW(t)$ is the Wiener process representing random fluctuations.

5. **Constant Interest Rates and Volatility**: The risk-free interest rate (r) and volatility (σ) remain constant over the option's lifespan, although they can be relaxed if known functions of time.

6. **Absence of Dividends**: The underlying asset does not pay dividends during the option's lifespan, although this assumption can be relaxed if dividends are a known function of time.

7. **Arbitrarily Divisible Underlying Asset**: The quantity of the underlying asset in the portfolio can be any real number, not limited to integers.

8. **Arbitrage-Free Market**: The market is free from arbitrage opportunities, ensuring that no risk-less profit can be obtained from price disparities.

Using these assumptions, we adopt the perspective of the option writer to hedge against risk. By adjusting the amount of underlying asset owned and maintaining a cash amount that can be invested, the writer aims to ensure their portfolio's balance and eliminate risk.

2.1.3 Derivation of Black-Scholes Equation

Building upon the assumptions detailed in the preceding section, we proceed to derive the Black-Scholes Equation utilizing Itô's lemma, while acknowledging the stochastic process of $S(t)$ as described in equation (2.1). Itô's Lemma, a cornerstone of stochastic calculus, facilitates the calculation of differentials for functions of stochastic processes, encompassing both deterministic and stochastic aspects. We express the change in V accordingly[5],

$$dV = \left(\mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2\frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t}\right)dt + \sigma S\frac{\partial V}{\partial S}dW(t) \tag{2.2}$$

We create a portfolio by buying one option and selling a certain amount (Δ) of the underlying asset. If Δ is negative, it means we're buying $|\Delta|$ units of the asset. For example, if we buy a put option, we might also buy some stock to lower risk, so Δ is negative. The key idea of the Black-Scholes model is finding the right Δ to make the portfolio deterministic. The value of this portfolio is calculated as follows,

$$\Pi(t) = V - \Delta S \tag{2.3}$$

The change in the value of this portfolio over a time-step dt is given by:

$$d\Pi(t) = dV - \Delta dS, \tag{2.4}$$

assuming Δ remains constant during the time-step. Substituting equations (2.1) and (2.2) into equation (2.4), we derive:

$$d\Pi(t) = \left(\mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t} - \mu\Delta S\right)dt + \sigma S\left(\frac{\partial V}{\partial S} - \Delta\right)dW(t) \tag{2.5}$$

Note that there are two terms on the right-hand side of equation (2.5). The first term is deterministic, whereas the second term is stochastic as it involves the standard Wiener process $dW(t)$. However, if we choose

$$\Delta = \frac{\partial V}{\partial S}$$

then the stochastic term becomes zero, and equation (2.5) simplifies to:

$$d\Pi = \left(\frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t}\right)dt. \tag{2.6}$$

Therefore, by selecting $\Delta = \frac{\partial V}{\partial S}$, we transform the stochastic expression into a deterministic one.

When investing in riskless assets, the return on investment Π grows at a rate of $r\Pi dt$. If the change in investment value ($d\Pi$) exceeds this rate, there's an opportunity for riskless profit. Investors can borrow to invest in the portfolio if $d\Pi > r\Pi dt$, or short the portfolio if $d\Pi < r\Pi dt$, ensuring riskless profit. The assumption of no transaction costs enables investors to execute these strategies without added expenses, leading to the fundamental concept in finance: $d\Pi = r\Pi dt$. Therefore, we should have $d\Pi = r\Pi dt$, and hence, by equation (2.6):

$$r\Pi dt = \left(\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t}\right) dt. \tag{2.7}$$

Now, replace Π in this equation by $V - \Delta S$ as given in equation (2.3), and replace Δ by $\frac{\partial V}{\partial S}$ as given in equation (2.7). Then, divide both sides by dt. This manipulation leads to the Black-Scholes equation:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0. \tag{2.8}$$

where V is the option price, S is the spot price of the underlying asset, r is the risk-free interest rate and σ is the volatility of the asset price. It's crucial to note that this equation is independent of the drift parameter μ and is applicable to any derivative that satisfies the previously outlined assumptions, particularly in the context of call and put options. The difference between a call and a put option becomes evident when examining their respective boundary conditions. For a call option, we encounter [2]:

$$\begin{aligned} t = T: &\quad C(S,T) = \max(S(T) - K, 0), \\ S = 0: &\quad C(0,t) = 0, \\ S \to \infty: &\quad C(S,t) \sim S. \end{aligned} \tag{2.9}$$

and for a put we have

$$\begin{aligned} t = T: &\quad P(S,T) = \max(K - S(T), 0), \\ S = 0: &\quad P(0,t) = K\exp(-r(T-t)), \\ S \to \infty: &\quad P(S,t) \to 0. \end{aligned} \tag{2.10}$$

For better understanding, when the underlying asset price, represented by S, approaches zero, the value of a call option (C) diminishes to zero, while a put option is certain to be exercised. In such cases, the put option's value is determined by the strike price discounted by the risk-free interest rate. Conversely, as the asset price escalates towards infinity, exceeding the strike price ($S > K$), the value of a put option becomes

negligible, whereas the value of a call option is solely dependent on the underlying asset's price. These scenarios illustrate how call and put options are evaluated under different conditions, laying down the fundamental principles of pricing in an arbitrage-free market. Having established a foundational understanding of option modeling and pricing, and having comprehended the intricacies of the Black-Scholes Equation, we can now delve into its diffusion properties and draw parallels with the Time-Dependent Schrödinger Equation.

2.1.4 Diffusion-Like Behavior in the BSE and the TDSE

In analyzing the Black-Scholes equation, it's noteworthy how it shares fundamental similarities with the Time-Dependent Schrödinger Equation in their mathematical structures. Both equations exhibit characteristics akin to diffusion equations. Specifically, the one-dimensional diffusion heat equation, denoted as[6],

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial x^2} \tag{2.11}$$

where C represents the diffusing substance and D is the diffusion coefficient, is of relevance. Utilizing Equation (2.11), the Black-Scholes equation can be reframed as a diffusion equation[1],

$$\frac{\partial V}{\partial t} = -\frac{1}{2}\sigma^2\frac{\partial^2 V}{\partial S^2} \tag{2.12}$$

In this expression, the first term captures the rate of change of the option price V with respect to time t. The second term,

$$\frac{1}{2}\sigma^2\frac{\partial^2 V}{\partial S^2}$$

characterizes diffusion arising from the stochastic movement of the underlying asset price S. This representation highlights the analogous behavior of option pricing to diffusion processes, where the option price diffuses over time due to both random fluctuations and deterministic trends in the underlying asset price.

The Time-Dependent Schrödinger Equation is a fundamental equation in quantum mechanics that describes how the wave function of a quantum system evolves over time. In its diffusion form, the TDSE is written as:

$$i\hbar\frac{\partial \psi}{\partial t} = -D\frac{\partial^2 \psi}{\partial x^2} \tag{2.13}$$

Here, ψ represents the wave function of the system, t denotes time, and x represents spatial position. The term $\hbar$ denotes the reduced Planck constant, which plays a crucial role in quantum mechanics. The coefficient D represents the diffusion coefficient, which captures the dynamics of quantum systems, particularly in scenarios where diffusion-like behavior is observed, such as the spreading of wave packets or particles,

$$D = \frac{\hbar^2}{2m}$$

Although originating from different domains—finance and quantum mechanics—both equations share this common characteristic, emphasizing the profound connections between seemingly unrelated fields. This underscores the potential for interdisciplinary insights and cross-domain applicability. Next, we'll delve into numerical methods to solve the Time-Dependent Schrödinger Equation then the Black-Scholes equation, leveraging these connections to deepen our understanding of their respective domains.

2.2 Numerical Methods

2.2.1 Finite Differences

In all but the simplest of cases analytical techniques often fail to obtain solutions to complex differential equations[7]. Therefore one is left with the use of numerical techniques in order to find approximate solutions. The numerical treatment of Equation (1.2) requires the discretization of our domain in both space and time. This section will serve to illustrate the construction of said domain and the notation.

Let x_0 and x_{N_x} denote the boundary points of our domain along the x-axis. The interval $[x_0, x_{N_x}]$ is divided into N_x subintervals, each containing N_x points separated by step sizes Δx, where [7]

$$\Delta x = x_{j+1} - x_j \quad \text{and} \quad j \in \{1, 2, 3, \ldots, N_x - 1\} \tag{2.14}$$

Let Δt be the time step, and $t = n\Delta t$, where $n \in \{0, 1, \ldots, N\}$. The wave function at any given point is denoted by $\psi(x, t)$ so that he difference equations take the following forms[7]:

$$\begin{aligned} \frac{\partial \psi}{\partial t} &\approx \frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} + O(\Delta t) \\ \frac{\partial^2 \psi}{\partial x^2} &\approx \frac{\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n}{\Delta x^2} + O(\Delta x^2) \end{aligned}$$

Difference equations (2.15)

We then introduce several commonly used numerically methods for stepping the TDSE through time and solving for future time steps from a given initial wave function $\psi(x, 0)$.

2.2.2 Explicit Methods

Beginning with explicit methods, we'll introduce a numerical technique to solve the Time-Dependent Schrödinger Equation. This method directly computes ψ_i^{n+1} from ψ_i, providing a straightforward approach for stepping through time and solving for future time steps from an initial wave function $\psi(x, 0)$ denoted by,

$$\psi_j^0 = \frac{1}{\sqrt[4]{2\pi\sigma^2}} exp\left[\frac{-(x-x_0)^2}{4\sigma^2} + ikx\right] \tag{2.16}$$

To transform Equation (1.2) into a finite difference equation at the current time step n, we initially substitute the derivative in the Hamiltonian operator with the appropriate difference equations (Eq. 2.15). This will subsequently result in:

$$H\psi = \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V\right)\psi = -\frac{\hbar^2}{2m}\frac{\left(\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n\right)}{\Delta x^2} + V_j\psi_j^n \tag{2.17}$$

Here, $V(x) \equiv V_j$ denotes a time-invariant potential. To progress numerically and compute future time steps ψ, we replace the left-hand side (LHS) of Equation (1.2) with our first-order accurate difference equation:

$$i\hbar\frac{\partial\psi}{\partial t} = i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} \tag{2.18}$$

By equating the right-hand side (RHS) of Equation (1.2) with that of Equation (2.17), we derive:

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} = -\frac{\hbar^2}{2m}\frac{\left(\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n\right)}{\Delta x^2} + V_j\psi_j^n \tag{2.19}$$

Given that ψ is known at any given instant—either from a prior time step or from initial conditions—the only unknown in Equation (2.19) is ψ_j^{n+1}. With algebraic manipulation, we can solve for this unknown at each time step:

$$\psi_j^{n+1} = \psi_j^n - \frac{i\Delta t}{\hbar}\left(-\frac{\hbar^2}{2m}\frac{\psi_j^{n+1} - 2\psi_j^n + \psi_{j-1}^n}{\Delta x^2} + V_j\psi_j^n\right) \tag{2.20}$$

In a more condensed format, it is simply

$$\psi^{n+1} = (1 - \frac{i\Delta t}{\hbar} H^n)\psi^n \tag{2.21}$$

The straightforward nature of this equation provides easy access to solutions at future time steps, denoted as ψ^{n+1}. However, despite its simplicity, the explicit method suffers from low accuracy, instability, and lack of unitarity.

2.2.3 Implicit Method

Transitioning from explicit methods, implicit methods provide an alternative solution to address stability challenges when solving the Time-Dependent Schrödinger Equation. Implicit methods integrate future time steps into the solution process, effectively mitigating stability issues common in explicit methods. Unlike explicit methods, which rely solely on current or past nodal points for computing future steps, implicit methods require solving a system of interconnected linear equations with multiple unknowns [8]. Despite being computationally more demanding due to solving a substantial system of simultaneous equations, implicit methods often exhibit unconditional stability or mandate weak stability conditions to ensure numerical boundedness. This flexibility allows for the use of larger step sizes in solution propagation. However, it's important to note that stability alone doesn't guarantee accuracy, emphasizing the need for a balanced selection of step sizes to achieve the desired level of precision.

Implicitly solving the Schrödinger equation involves assessing the spatial derivatives of the right-hand side (RHS) of Equation (1.2) at forthcoming time steps, diverging from the present time points as in explicit schemes. This implicit solution is represented as,

$$i\hbar \frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} = \frac{-\hbar^2}{2m} \frac{\left(\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}\right)}{\Delta x^2} + V_j \psi_j^{n+1} \tag{2.22}$$

Solving this equation requires handling a considerable system of linear equations, which, when represented in matrix form, appears as a tridiagonal matrix. Tridiagonal matrix solvers provide effective solutions to this system, a topic we will explore in more detail later on. Equation (2.22) can be succinctly expressed as,

$$\psi^n = \left(1 + \frac{i\Delta t}{\hbar} H^{n+1}\right) \psi^{n+1} \tag{2.23}$$

Through slight manipulation, we can determine the solution for ψ^{n+1}.

$$\psi^{n+1} = \left(1 + \frac{i\Delta t}{\hbar} H^{n+1}\right)^{-1} \psi^n \tag{2.24}$$

Due to limitations discussed earlier, both of these schemes do not meet all the requirements of the resulting propagation algorithm, which only conditionally converges to a physically acceptable solution. However, it's possible to derive a numerical scheme that is second-order accurate in time, stable, and unitary. This scheme, known as the Crank-Nicolson (CN) method, is highly favored for its simplicity and has been widely used since its introduction by Crank and Nicolson in 1947. This method will serve as the primary approach for solving both the TDSE and the BSE.

2.3 Crank-Nicolson

This numerical integration scheme can be formulated using different approaches. In fact, we can explore two distinct versions, denoted as I and II, of the Crank-Nicolson (CN) scheme, as we illustrate below.

2.3.1 Solving TDSE using Crank-Nicolson I

To devise a numerical scheme that ensures both unitarity and stability, we can merge the implicit and explicit methods outlined in Equation (2.20) and Equation (2.22). This leads to a second-order accurate numerical scheme in time, characterized by stability and unitarity and can be derived as follows[7]:

To initiate the process, we compute the average of the spatial discretizations derived in Equation (2.20) and Equation (2.22). Subsequently, we substitute this average into the right-hand side of Equation (1.2), resulting in our Schrödinger equation:

$$i\hbar\frac{\partial\psi}{\partial t} = \frac{1}{2}[H^n\psi^n + H^{n+1}\psi^{n+1}] \tag{2.25}$$

Next, we substitute the derivative of the left-hand side of Equation (2.25) with the first-order difference equation from Equation (2.15), and after some straightforward algebraic manipulation, we attain the following expression:

$$(1 + \frac{i\Delta t}{2\hbar}H^{n+1})\psi^{n+1} = (1 - \frac{i\Delta t}{2\hbar}H^n)\psi^n \tag{2.26}$$

Through rearrangement and solving for the future unknowns, we arrive at the definitive expression for our numerical solution:

$$\psi^{n+1} = \frac{(1 - \frac{i\Delta t}{2\hbar}H^n)}{(1 + \frac{i\Delta t}{2\hbar}H^{n+1})}\psi^n \tag{2.27}$$

This equation can be represented using tridiagonal matrices because they offer computational efficiency and simplicity. However, computational costs may arise from matrix

vector operations such as matrix inversions. For this reason we will devise an alternative approach to the CN that eliminates the matrix vector operations.

2.3.2 Solving TDSE using Crank-Nicolson II

An alternative method to derive the Crank-Nicolson scheme involves transforming the differential equation into an integration problem, thereby bypassing the use of the finite difference operator. We consider the standard Time-Dependent Schrödinger Equation in the position picture[8]:

$$\psi\left(x, t+\Delta t\right) = \psi\left(x, t\right) - i\Delta t \int_{t}^{t+\Delta t} \Delta t' \hat{H}(x, t')\psi(t') \tag{2.28}$$

Instead of approximating the time derivative of the wavefunction, we aim to approximate the integral of $\hat{H}(x,t)\psi(x,t)$ using a weighted quadrature rule:

$$\int_{t}^{t+\Delta t} \Delta t' \hat{H}\left(x, t'\right) \psi\left(x, t'\right) = \sum_{n} w_q \hat{H}\left(x_j, t_n\right) \psi\left(x_j, t_n\right) = \sum_{n} w_n H_j^n \psi_j^n \tag{2.29}$$

This is know as the Newton–Cotes formula where $\hat{H}_j^n$ acts on ψ_j^n as

$$\hat{H}_j^n \psi_j^n = \frac{1}{h^2}\left(\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n\right) + V_j^n \psi_j^n \tag{2.30}$$

The weights w_n regulate the significance of each $\hat{H}_j^n \psi_j^n$, computed at the base points (x_j, t_n). The determination of t_n and w_n defines the quadrature rule, typically with fixed base points and only the weights adjusted based on specific approximation criteria. When the time points t_n are evenly spaced, they adhere to the so-called Newton-Cotes integration rules. This framework allows for the derivation of an alternative expression for the Crank-Nicolson (CN) scheme. This variant emerges when we approximate the integral (2.28) using the single-point open Newton–Cotes formula (2.29), where $w_1 = \Delta t$ and $t_1 = (t_n + t_{n+1})/2 = t_{n+1/2}$ [8].

$$\psi^{n+1} = \psi^n - i\Delta t H(t_{n+1/2})\psi^{n+1/2} + \frac{\Delta t^3}{24}\psi(\ddot{t_e}) \tag{2.31}$$

where $t_q < t_e < t_{n+1}$. Now, utilizing Taylor expansions around the midpoint $t_{n+1/2} = t_n + \frac{\Delta t}{2} = t_{n+1} - \frac{\Delta t}{2}$, we obtain:

$$\psi^{n+1/2} = \psi\left(t_n + \frac{\Delta t}{2}\right) = \psi^n + \frac{\Delta t}{2}\dot{\psi}^{n+1/2} + \frac{1}{2}\left(\frac{\Delta t}{2}\right)^2 \ddot{\psi}_{n+1/2} + \ldots \tag{2.32}$$

$$\psi^{n+1/2} = \psi\left(t_{n+1} - \frac{1}{2}\right) = \psi^{n+1} - \frac{\Delta t}{2}\dot{\psi}^{n+1/2} + \frac{1}{2}\left(\frac{\Delta t}{2}\right)^2 \ddot{\psi}^{n+1/2} + \dots \tag{2.33}$$

Here, $\dot{\psi} = \frac{d\psi}{dt}$ and $\ddot{\psi} = \frac{d^2\psi}{dt^2}$ represent the first and second time derivatives of ψ, respectively. Adding Equations (2.32) and (2.33) yields:

$$\psi^{n+1/2} = \psi^n + \psi^{n+1/2} + \left(\frac{\Delta t}{2}\right)^2 \ddot{\psi}^{n+1/2} + \dots \tag{2.34}$$

Substituting this into Equation (2.31) leads to:

$$\psi^{n+1} = \psi^n - i\frac{\Delta t}{2}H(t_{n+1/2})(\psi_n + \psi^{n+1}) + \mathcal{O}(\Delta t^2) \tag{2.35}$$

By rearranging terms, we arrive at an alternative CN-II algorithm:

$$(\mathbf{I} + \frac{i\Delta t}{2}H^{n+1/2})\psi^{n+1} = (\mathbf{I} - \frac{i\Delta t}{2}H^{n+1/2})\psi^n \tag{2.36}$$

Upon scrutinizing the two alternative formulations of the CN algorithm, it becomes evident that they coincide for time-independent Hamiltonians due to $\mathbf{H}_n = \mathbf{H}_{n+1/2} = \mathbf{H}_{n+1}$. However, when dealing with time-dependent Hamiltonians, the latter expression proves to be more efficient as it necessitates the evaluation of the Hamiltonian at only one point, namely, the midpoint $t_{n+1/2}$. Nevertheless, in both scenarios, the computational workload can be further reduced by eliminating the matrix-vector component of the first equation. This can be achieved by taking Equation (2.26), computing the Hamiltonians at $t_{n+1/2} = t_n + \Delta t/2$ and then introducing and subtracting ψ_j from the RHS to derive the following:

$$\left(\mathbf{I} + \bar{\mathbf{A}}\right)\psi^{n+1} = -\left(\mathbf{I} + \bar{\mathbf{A}}\right)\psi^n + 2\psi^n \tag{2.37}$$

which can be rearrange to give

$$\left(\mathbf{I} + \bar{\mathbf{A}}\right)\frac{\psi^{n+1} + \psi^n}{2} = \psi^n \tag{2.38}$$

where $\bar{A} \equiv A_{t_q+1/2}$. Finally, we can solve for ψ_{j+1} by setting

$$\bar{\psi} = \frac{\psi^{n+1} + \psi^n}{2} \tag{2.39}$$

then setting Equation (2.38) to

$$\left(\mathbf{I} + \bar{\mathbf{A}}\right) \bar{\psi} = \psi^n \tag{2.40}$$

which gives

$$\psi^{n+1} = 2\bar{\psi} - \psi^n \tag{2.41}$$

We are now able to solve the linear system of equations for $\bar{\psi}$ because both $\bar{A}$ and ψ_j are known. We can then subtract $2\bar{\psi}$ by ψ^n to obtain ψ^{n+1}. Consequently, the matrix inversion part has been completely eliminated from the solution process, forwarding the solution.

Having explored numerical methods such as the Crank-Nicolson scheme for solving the TDSE, it's crucial to shift our focus towards understanding boundary conditions. While the CN method offers efficient solutions for the TDSE within defined domains, understanding how boundary conditions are incorporated is essential for ensuring the validity of the solutions. Therefore, we will now examine how boundary conditions for the TDSE are incorporated and their significance in accurately modeling quantum systems within the framework of the TDSE.

2.4 Boundary Conditions for TDSE

When solving differential equations through discretization techniques, the process often culminates in representing the problem in matrix form, typically involving a square matrix A of size $N \times N$. For boundary-value problems, the resulting matrix equation usually takes the form of an eigenvalue equation,

$$Ay = \lambda y \tag{2.42}$$

Here, A encapsulates crucial information about the original differential problem, the discretization scheme employed, and any imposed boundary conditions. The discretization process entails dividing the configuration space into a grid. Each grid point x_i is associated with unknown values of the function $\phi(x_i)$, its first derivative $\phi'(x_i)$, and its second derivative $\phi''(x_i)$. Importantly, this process inherently incorporates boundary conditions, ensuring precise satisfaction of the differential equations at every grid point, including the boundaries $x_0 = a$ and $x_{N+1} = b$. By adjusting the order of the Sturm-Liouville equations and rearranging them accordingly, the resulting algebraic formulation naturally accounts for these boundary conditions. Consequently, there's no need for separate imposition of boundary conditions, as they are inherently embedded within the discretization process. This holistic approach not only simplifies the computational procedure but also guarantees the accuracy and integrity of the solution throughout the entire domain.[8]

3 Using Crank-Nicolson to solve Black-Scholes Equation

We can now apply a similar approach to the Black-Scholes Equation to calculate the one-time payoff upon exercising the put option, denoted by $[K - S(t)]^+$ for any t up to and including the maturity, where K represents the predetermined strike price. Following a methodology akin to that used for solving the Time-Dependent Schrödinger Equation, we employ difference methods to discretize both the stock and time dimensions. Subsequently, we employ the Crank-Nicolson Method to determine the value of an American put option V at any of the resulting grid points. Specifically, if the option exists between $t = 0$ and T, we partition this interval into N equally spaced sub-intervals of length $\Delta t = T/N$, resulting in a total of $N + 1$ time steps. Similarly, let $S_{\max}$ denote the maximum stock price at which the put value is practically zero. By dividing the range of possible stock values into M subintervals, each with a width of $\Delta S = S_{\max}/M$, we obtain $M + 1$ discrete stock prices. We can develop the Crank-Nicolson scheme for the Black-Scholes equation by exploring both implicit and explicit finite difference methods[9].

3.1 Black-Scholes Equation: Explicit Method

Let's define $V_i^n \equiv V(S_j, t_n)$ and utilize Equation (1.1) to derive the Explicit finite difference scheme. Applying the difference equation from Equation (2.15), the temporal derivative $\partial V/\partial t$ is reasonably approximated by the forward difference:

$$\frac{\partial V(S_j, t_n)}{\partial t} \approx \frac{V_j^{n+1} - V_j^n}{\Delta t} \tag{3.1}$$

The first stock derivative $\partial V/\partial S$ is approximated by a central difference quotient across:

$$\frac{\partial V(S_j, t_n)}{\partial S} \approx \frac{V_{j+1}^n - V_{j-1}^n}{2\Delta S} \tag{3.2}$$

Using the difference equation from Equation (2.15), the second-order stock derivative is taken to be:

$$\frac{\partial^2 V(S_j, t_n)}{\partial S^2} = \frac{V_{j+1}^n - 2V_j^n + V_{j-1}^n}{\Delta S^2} \tag{3.3}$$

Substituting Equations 3.1, 3.2, and 3.3 into the Black-Scholes Equation, the discretized Black-Scholes equation becomes (using $S = i\Delta S$):

$$\frac{V_j^{n+1} - V_j^n}{\Delta t} + rj\Delta S\frac{V_{j+1}^n - V_{j-1}^n}{2\Delta S} + \frac{1}{2}\sigma^2\Delta S^2\frac{V_{j+1}^n - 2V_j^n + V_{j-1}^n}{\Delta S^2} = rV_j^n \tag{3.4}$$

Here, $j \in [1, M-1]$ and $n \in [0, N-1]$. Equation (3.4) is rearranged to relate $V(S_j)$ at time t_{n+1} to three different put values $V(S_j)|^{i+1}j = i-1$ at time t_n. In practice, this implies that working our way backwards from the terminal condition $V(S_T, T) = [K - S(T)]^+$, each time step t_i requires solving $M-1$ simultaneous equations to compute the various put values $V(St_j)$. This arises by imposing suitable boundary conditions at the edges $S = 0$ and $S = S_{max}$.

3.2 Black-Scholes Equation: Implicit Method

We can apply a similar implicit method used for solving the TDSE to solve the BSE implicitly by evaluating the spatial derivatives at future time steps instead of the present ones. This approach is implemented as follows,

$$\frac{\partial V(S_j, t_n)}{\partial S} \approx \frac{V_{j+1}^{n+1} - V_{j-1}^{n+1}}{2\Delta S} \tag{3.5}$$

$$\frac{\partial^2 V(S_j, t_n)}{\partial S^2} = \frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{\Delta S^2} \tag{3.6}$$

The Black-Scholes Equation becomes

$$\frac{V_j^{n+1} - V_j^n}{\Delta t} + rj\Delta S \frac{V_{j+1}^{n+1} - V_{j-1}^{n+1}}{2\Delta S} + \frac{1}{2}\sigma^2 \Delta S^2 \frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{\Delta S^2} = rV_j^n \tag{3.7}$$

which evidently relates three different put values $V(S_i)|^{j+1}i = j-1$ at time $n+1$ to one put value $V(S_j)$at time n. We are now equipped to employ both implicit and explicit methods in deriving the Crank-Nicolson scheme for solving the BSE.

3.3 Black-Scholes Equation: Crank-Nicolson

Taking the equally weighted average of the implicit scheme from Equation (3.4) and the explicit scheme from Equation (3.7) we obtain,

$$\begin{aligned} &\frac{V_j^{n+1} - V_j^n}{\Delta t} + rj\Delta S + \frac{1}{2} rj\Delta S \left(\frac{V_{j+1}^n - V_{j-1}^n}{2\Delta S} + \frac{V_{j+1}^{n+1} - V_{j-1}^{n+1}}{2\Delta S} \right) \\ &+ \frac{1}{2^2}\sigma^2 j^2 \Delta S^2 \left(\frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{\Delta S^2} \right) = rV_j^n \end{aligned} \tag{3.8}$$

Separating the tn terms from the $t_n + 1$ terms, and canceling out the ΔS factors:

$$\begin{aligned}\frac{V_j^{n+1}}{\Delta t} + \frac{1}{2^2} rj(V_{j+1}^{n+1} + V_{j-1}^{n+1}) + \frac{1}{2^2}\sigma^2 j^2 (V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}) = \\ \frac{V_j^n}{\Delta t} - \frac{1}{2^2} rj(V_{j+1}^n + V_{j-1}^n) - \frac{1}{2^2}\sigma^2 j^2 (V_{j+1}^n - 2V_j^n + V_{j-1}^n) + rV_j^n\end{aligned} \tag{3.9}$$

Factorizing the appropriate j-terms on each side and multiplying through by Δt:

$$\begin{aligned}&\frac{1}{2^2} j\Delta t(\sigma^2 j - r) + \left(1 - \frac{1}{2}\sigma^2 j^2 \Delta t\right) V_j^{n+1} + \frac{1}{2^2} j\Delta t(\sigma^2 j + r) V_{j+1}^{n+1} \\ &= -\frac{1}{2^2} j\Delta t(\sigma^2 j - r) V_{j-1}^n + \left(1 + \left(r + \frac{1}{2}\sigma^2 j^2\right)\Delta t\right) - \frac{1}{2^2} j\Delta t(\sigma^2 j + r) V_{j+1}^n\end{aligned} \tag{3.10}$$

or simply

$$a_j V_{j-1}^{n+1} + b_j V_j^{n+1} + c_j V_{j+1}^{n+1} = -a_j V_{j-1}^n + d_j V_j^n - c_j V_{j+1}^n \tag{3.11}$$

where the coefficients are defined as

$$a_j = \frac{1}{4} j\Delta t(\sigma^2 j - r) \tag{3.12}$$

$$b_j = (1 - \frac{1}{2}\sigma^2 j^2 \Delta t) \tag{3.13}$$

$$c_j = \frac{1}{4} j\Delta t(\sigma^2 j + r) \tag{3.14}$$

$$d_j = (1 + (r + \frac{1}{2}\sigma^2 j^2)\Delta t) \tag{3.15}$$

The derivation of the Crank-Nicolson scheme for the BSE is now complete. Using Equation 3.11, we can express it in a matrix equation format, particularly as a tridiagonal matrix, which will be examined more closely in Section 4. In Section 4, we will provide a detailed analysis of the code utilized to solve the TDSE using both the CN-I and CN-II approaches. Subsequently, a comparable methodology will be applied to tackle the BSE.

4 Methods and Implementation

In this section, we outline the step-by-step implementation of the Crank-Nicolson methods discussed earlier, including the associated algorithms. These methods will be applied to solve the Time-Dependent Schrödinger Equation over multiple time steps, comparing their accuracy and efficiency. Subsequently, we will employ the same methodology to solve the Black-Scholes Equation, followed by an analysis of the results.

4.1 Coding Crank-Nicolson I for the TDSE in C++

We'll begin by defining the discretization of our computational domain mentioned in Section 2.2.1. For spatial discretization, this entails determining boundary points and the uniform step size between adjacent grid points. For temporal discretization, we specify the duration of simulated time and the discrete temporal step size. We implement a function to generate equally spaced intervals. The following code snippet in C++ demonstrates this process, including the necessary include statements and namespaces being utilized. The "Eigen" library is utilized for streamlined vector and matrix operations:

```
#include <iostream>
#include "Eigen/Dense"
#include <vector>
#include <complex>
#include <iomanip>
#include <fstream>
#include <stdexcept>

using namespace std;
using namespace Eigen;

VectorXd generateEvenlySpacedIntervals(double start, double end,
    double interval) {
    int size = static_cast<int>((end - start) / interval) + 1;
    VectorXd intervals(size);

    for (int i = 0; i < size; i++) {
        intervals(i) = start + i * interval;
    }

    return intervals;
}
```

Listing 1: Function to Generate Equally Spaced Intervals

Having developed a function that generates a vector with evenly spaced intervals, we can now specify parameters for both the size and discretization level of our spatial and temporal domains. The code below initializes parameters for a 1D grid in space (x) and time (t). It defines the starting and ending points for both space and time intervals, as well as the step sizes (dx and dt). Additionally, it sets up constants for the Planck constant (hb) and particle mass (m). Using the function "generateEvenlySpacedIntervals," it creates vectors x_i and t_n containing equally spaced intervals for space and time, respectively. Finally, it determines the sizes of the vectors x_i and t_n and assigns them to variables Nx and Nt, representing the number of grid points in space and time, respectively.

```
1     double start_x, end_x, dx;
2     start_x   = 0;
3     end_x     = 20;
4     dx       = 0.05;
5
6     double start_t, end_t, dt;
7     start_t = 0;
8     end_t  = 1;
9     dt = 0.05/20;
10
11    double hb, m;
12    hb = 1;
13    m = 1;
14
15
16    VectorXd x_i = generateEvenlySpacedIntervals(start_x, end_x,
      dx);
17    VectorXd t_n = generateEvenlySpacedIntervals(start_t, end_t,
      dt);
18
19    int Nx, Nt;
20    Nx = x_i.size();
21    Nt = t_n.size();
```

Listing 2: Initialization of TDSE Parameters and Generation of Equally Spaced Grid Points

In the above code snippet, both spatial and temporal domains have been discretized into 401 points, with the resulting grid sizes stored as variables N_x and N_t, respectively.

Following that, we establish a function tasked with initializing the initial wave function referenced in Equation (2.16). The initial wave function takes the shape of a Gaussian, with parameters such as its initial spatial width σ, momentum k, and its center position x_0. It is more suitable to delegate the construction of our Gaussian shape to a function. This is accomplished as follows:

```
VectorXcd init_psi_func(const VectorXd& x, double x_0, double k,
    double sigma) {
    VectorXcd result(x.size());
    double norm, realPart, imagPart;

    norm = std::pow((2 * M_PI) * (std::pow(sigma, 2)), -0.25);

    for (int i = 0; i < x.size(); i++) {
        realPart = -(x(i) - x_0) * (x(i) - x_0) / (4 * sigma *
   sigma);

        imagPart = k * x(i);

        result(i) = norm * std::exp(std::complex<double>(
   realPart, 0)) * std::exp(std::complex<double>(0, imagPart));
    }

    return result;
}
```

Listing 3: Initial Wave Function

To solve Equation (2.27) in its discrete form, we utilize a matrix equation approach as follows,

$$\psi^{n+1} = A^{-1}B\psi^n \tag{4.1}$$

By expanding Equation (2.26) using the approximate finite difference approximations from Equation (2.15), we arrive at the following:

$$\begin{aligned} &\psi_j^{n+1} + \frac{i\Delta t}{2\hbar}\left[-\frac{\hbar^2}{2m}\frac{\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}}{\Delta x^2} + V_i\psi_i^{n+1}\right] \\ &= \psi_i^n - \frac{i\Delta t}{2\hbar}\left[-\frac{\hbar^2}{2m}\frac{\psi_{j+1}^{n} - 2\psi_j^{n} + \psi_{j-1}^{n}}{\Delta x^2} + V_i\psi_i^{n+1}\right] \end{aligned} \tag{4.2}$$

In this formulation, we employ second-order central differences to discretize our spatial variables. Represented in tridiagonal matrix form, it appears as follows:

$$\underbrace{\begin{bmatrix} \alpha_1 & -\mu & & & \\ -\mu & \alpha_2 & -\mu & & \\ & \ddots & \ddots & \ddots & \\ & & -\mu & \alpha_{N-1} & -\mu \\ & & & -\mu & \alpha_N \end{bmatrix}}_{\text{Matrix A}} \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \\ \vdots \\ \vdots \\ \psi_{N_x}^{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} \beta_1 & \mu & & & \\ \mu & \beta_2 & \mu & & \\ & \ddots & \ddots & \ddots & \\ & & \mu & \beta_{N-1} & \mu \\ & & & \mu & \beta_N \end{bmatrix}}_{\text{Matrix B}} \begin{bmatrix} \psi_1^{n} \\ \psi_2^{n} \\ \vdots \\ \vdots \\ \psi_{N_x}^{n} \end{bmatrix} \tag{4.3}$$

where

$$\alpha_i = 1 - \frac{i\Delta t\hbar}{2m\Delta x^2} + \frac{i\Delta t}{2\hbar}V_i \tag{4.4}$$

$$\beta_i = 1 + \frac{i\Delta t\hbar}{2m\Delta x^2} - \frac{i\Delta t}{2\hbar}V_i \tag{4.5}$$

$$\mu = \frac{i\Delta t\hbar}{4m\Delta x^2} \tag{4.6}$$

In the code segment below, we initialize matrices to represent the system's Hamiltonian and wavefunction. The kinetic energy matrix, T, and the potential energy matrix, V, are constructed, with the former representing second-order central differences for discretizing spatial variables. Considering a free electron, we remove the potential term. Next, we form the Hamiltonian matrix, H, by summing T and V. Matrices FTCS (which represents Matrix A) and BTCS (which represents Matrix B), both of size $N_x \times N_x$, are formed by populating the relevant diagonals as described in Equation (4.3). To compute the Crank-Nicolson matrix, we employ LU decomposition[7] to find the inverse of BTCS. Finally, we initialize the wavefunction vector ψ and iterate through time steps to update it using the Crank-Nicolson method, resulting in the solution of the quantum mechanical problem in its discrete form. This is accomplished as follows:

```
1     MatrixXcd iden = MatrixXcd::Identity(Nx, Nx);
2
3     Eigen::MatrixXcd T = Eigen::MatrixXcd::Zero(Nx, Nx);
4     Eigen::MatrixXcd V = Eigen::MatrixXcd::Zero(Nx, Nx);
5
6     T.diagonal().array() = -2.0;
7     T.diagonal(1).array() = T.diagonal(-1).array() = 1.0;
8     V.diagonal().array() = 0;
9
10    T *= ((-hb * hb) / (2.0 * m * dx * dx));
11
12    Eigen::MatrixXcd H = T + V;
```

```
13
14      Eigen::MatrixXcd FTCS = iden - ((std::complex<double>(0.0,
    1.0) * dt / (2.0 * hb)) * H);
15      Eigen::MatrixXcd BTCS = iden + ((std::complex<double>(0.0,
    1.0) * dt / (2.0 * hb)) * H);
16
17      MatrixXcd CN2 = FTCS * BTCS.inverse();
18
19      VectorXcd psi = init_psi_func(x_i, x_0, k, sigma);
20      VectorXcd psi_t;
21
22      for (int it = 0; it < Nt; it++) {
23          psi_t = CN2 * psi ;
24          psi = psi_t;
25
26      }
```

Listing 4: Computing the Crank-Nicolson matrix and solving the TDSE using LU decompostion

To verify the code, the probability density is computed and compared against an analytical solution. The equation for the probability density is as follows:

$$P(x,t) = |\psi(x,t)|^2 \tag{4.7}$$

This procedure is executed in the code and saved into a CSV file for each spatial discretization. Subsequently, it is visualized using Python for simplicity. The C++ code is outlined below:

```
1  std::ofstream outputFile("psi-t-norm-cn.csv");
2
3      if (!outputFile.is_open()) {
4          std::cerr << "Failed to open the file for writing." <<
    std::endl;
5          return 1;
6      }
7
8      int precision = 3;
9      outputFile << std::scientific;
10
11     double nrm_t;
12
13
14     for (int i = 0; i < Nx; i++) {
15
16       nrm_t = abs(psi_t[i]) * abs(psi_t[i]);
17       outputFile << x_i[i] << ',' << nrm_t << std::endl;
```

```
18
19      }
20      std::cout<< "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$" << endl
        ;
21      outputFile.close();
```

Listing 5: Probability Density of TDSE Numerical Scheme

Finally, the numerical results are compared to the analytical solution, which is implemented using the following function:

```
1  VectorXcd exact_psi(const VectorXd& x, double x_0, double k,
       double sigma, double dt, int Nx, int Nt) {
2      VectorXcd result(Nx);
3      complex<double> norm, gaussian_term, phase_term;
4      double time;
5      complex<double> zi(0.0, 1.0);
6
7      for (int i = 0; i < Nx; i++) {
8          time = Nt * dt;
9          norm = pow(pow(2.0 * M_PI, 0.25) * sqrt(sigma + (time *
       zi) / (2.0 * sigma)), -1);
10         gaussian_term = -(pow((x(i) - x_0 - k * time),2))/ (4 *
       pow(sigma, 2) + 2.0 * zi * time) + (zi*k*(x[i] - k*time/2.0))
       ;
11         result[i] = norm * exp(gaussian_term);
12
13     }
14     return result;
15 }
```

Listing 6: TDSE Analytical Solution

The probability density is recalculated in a manner similar to the numerical solution and exported into a CSV file for each spatial discretization. This allows for comparison against the numerical solution in Python. The code snippet is presented below:

```
1  VectorXcd exact_psi = exact_psi(x_i, x_0, k, sigma, dt, Nx, Nt);
2      std::ofstream outputFile3("ex-psi-initial-norm.csv");
3      if (!outputFile3.is_open()) {
4          std::cerr << "Failed to open the file for writing." <<
       std::endl;
5          return 1;
6      }
7      outputFile3.precision(precision);
8      outputFile3 << std::scientific;
9
10     double exact_psi_abs, realPart_tc, imagPart_tc;
```

```
11      for (int i = 0; i < Nx; i++){
12
13        exact_psi_abs = abs(exact_psi[i]) * abs(exact_psi[i]);
14
15        outputFile3 << x_i[i] << ',' << exact_psi_abs << endl;
16      }
```

Listing 7: Probability Density of TDSE Analytical Solution

The TDSE has been solved numerically employing the Crank-Nicolson I scheme in C++. Our next step involves refining our computational approach by transitioning to Crank-Nicolson II, with the objective of enhancing computational efficiency through the elimination of matrix-vector operations.

4.2 Coding Crank-Nicolson II for the TDSE in C++

The numerical application of the Crank-Nicolson II scheme closely resembles the techniques employed in implementing the Crank-Nicolson I scheme. However, in this implementation, we omit the matrix inversion operation, streamlining the solution process. Once more, to address the problem in its discretized form, we employ Equation (2.40).

$$\left(\mathbf{I} + \bar{\mathbf{A}}\right) \bar{\psi} = \psi_j$$

where $\bar{\psi}$ is

$$\bar{\psi} = \frac{\psi_{j+1} + \psi_j}{2}$$

and $\left(\mathbf{I} + \bar{\mathbf{A}}\right)$ represents our tridiagonal matrix,

$$\begin{bmatrix} \beta_1 & -\mu & & & \\ -\mu & \beta_2 & -\mu & & \\ & \ddots & \ddots & \ddots & \\ & & -\mu & \beta_{N-1} & -\mu \\ & & & -\mu & \beta_N \end{bmatrix} \tag{4.8}$$

We'll compute $\bar{\psi}$ using the Thomas algorithm, a highly efficient numerical technique for solving tridiagonal systems of linear equations. This method, involving a series of forward and backward substitutions, proves particularly advantageous for efficiently solving large systems of linear equations. The following function incorporates it into the code:

```
    void thomas_tridiag(const VectorXcd& a, const VectorXcd& b,
   const VectorXcd& c, const VectorXcd& psi, VectorXcd& u) {
    int j, n;
    complex<double> bet;

    n = a.size();

    VectorXcd gam(n);is needed.

    if (b[0] == 0.0) {
        throw("Error 1 in tridag");
    }

    u[0] = psi[0] / (bet = b[0]);

    for (j = 1; j < n; j++) {

        gam[j] = c[j - 1] / bet;
        bet = b[j] - a[j] * gam[j];

        if (bet == 0.0)
            throw("Error in tridag");

        u[j] = (psi[j] - a[j] * u[j - 1]) / bet;
    }

    for (j = (n - 2); j >= 0; j--)
        u[j] -= gam[j + 1] * u[j + 1];
}
```

Listing 8: Thomas Algorithm

We can employ the same parameters as before, but this time we forgo the LU decomposition method utilized in CN-I and opt to implement the Thomas Algorithm instead. In the code below, vectors are utilized to represent the coefficients of a tridiagonal matrix based on the equations (4.5) and (4.6):

```
    VectorXcd a(Nx - 1);
    a.setConstant(complex<double>(0, -dt * h / (4 * m * pow(dx,
   2))));

    VectorXcd b(Nx);
    b.setConstant(complex<double>(1, dt * h / (2 * m * pow(dx,
   2))));

    VectorXcd c(Nx - 1);
```

```
    c.setConstant(complex<double>(0, -dt * h / (4 * m * pow(dx,
2))));
```

Listing 9: Initialization of Tridiagonal Matrix Coefficients for TDSE

Next, we initialize the initial wave function as previously described in Section 4.1. Additionally, empty vectors are initialized to store our output values. The following code snippet incorporates these initializations:

```
    VectorXcd f(Nx);
    f.setZero();
    VectorXcd psi = init_psi(Nx, x_0, k, sigma, dx);
    VectorXcd final(Nx);
```

Listing 10: Initialization of Initial Wave Function and Output Vectors

The Thomas Algorithm is employed here to solve for $\bar{\psi}$ by substituting the coefficients of a tridiagonal matrix. After some straightforward manipulation, we derive ψ_{j+1}.

$$\psi_{j+1} = 2\bar{\psi} - \psi_j$$

This process is iterated through each time step, progressively updating our initial wave function. The code is as follows:

```
    for (int i = 0; i < Nt; ++i) {

        thomas_tridiag(a, b, c, psi, f);

        VectorXcd final = 2*f - psi;

        psi = final;
    }
```

Listing 11: Implementation of the Thomas Algorithm for Time Evolution of Initial Wave Function

The matrix inversion operation has been eliminated, simplifying the solution process. Subsequently, the probability density is calculated and outputted in a CSV file for each spatial discretization. Finally, the solution is validated by comparing it to the analytical solution, following the approach used in the previous section.

4.3 Solving Black-Scholes Equation

Now, employing the Crank-Nicolson code previously discussed, we can tackle the Black-Scholes Equation. We will utilize CN-I alongside the Thomas Algorithm because CN-II cannot be applied unless both matrices are identical. Leveraging the Thomas algorithm still avoids the need for matrix inversion, maintaining computational efficiency compared to scenarios where it is not employed. Using the coefficients specified in Equation (3.11), this set of equations can be expressed in matrix form as follows:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{M-2} & b_{M-2} & c_{M-2} \\ & & & a_{M-1} & b_{M-1} \end{bmatrix} \begin{bmatrix} V_1^{n+1} \\ V_2^{n+1} \\ \vdots \\ \vdots \\ V_{M-1}^{n+1} \end{bmatrix} = \begin{bmatrix} d_1 & -c_1 & & & \\ -a_2 & d_2 & -c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & -a_{M-2} & d_{M-2} & -c_{M-2} \\ & & & -a_{M-1} & d_{M-1} \end{bmatrix} \begin{bmatrix} V_1^n \\ V_2^n \\ \vdots \\ \vdots \\ V_{M-1}^n \end{bmatrix} \tag{4.9}$$

More concisely

$$T_1 V^{n+1} = T_2 V^n \tag{4.10}$$

where T_1 and T_2 are the two tridiagonal $(M-1) \times (M-1)$ matrices, V^{n+1}, V^n are the put values vectors. Since we know the terminal condition $[K - S(T)]^+$ (i.e V^N) and we are working our way backwards in time, we must compute,

$$V^n = T_2^{-1} \left(T_1 V^{n+1} \right) \tag{4.11}$$

for $n = N-1, N-2, ..., 1, 0$. We can now advance the initial put option V values over time using the Thomas Algorithm from the preceding section. Initially, we substitute the initial wave function with the one-time payoff received upon exercising the put option $[K - S(T)]^+$ for any t up to and including the maturity (expiration date of the option contract), where K represents the predetermined strike. This procedure is executed via the following function:

```
    VectorXd init_V(double K, double dS, int Nt, int Nx) {

    VectorXd V(Nt - 1);

    for(int j = 0; j < (Nt - 1); j++) {
        V[j] = std::max(K - dS * (j+1), 0.0);
    }
    return V;
}
```

Listing 12: Initial Put Option Values

The parameters for the stock price S_{max}, expiry date T, risk free rate r, volatility of the stock sd, predetermined strike K, number of stock price intervals M, number of time intervals N, time interval Δt and stock price interval ΔS are initialized as follows:

```
    int N  = 10;
    int M  = 20;
    double T = 0.4167;
    double Smax = 100.0;
    double r = 0.10;
    double sd = 0.40;
    double K = 50.0;
    double dS = Smax/M;
    double dt = T/N;
```

Listing 13: Initial Parameters for BSE

We utilize vectors to denote the coefficients of a tridiagonal matrix, derived from Equations (3.12), (3.13), (3.14), and (3.15). These coefficients vary with the stock price and are calculated within a loop for each price interval. The implementation of coefficients for matrix T_1 is depicted below:

```
    VectorXd a(M - 1);
    VectorXd b(M);
    VectorXd c(M - 1);
    VectorXd d(M);

    for(int j = 0; j < (M-1); j++) {
        a[j] = 0.25 * (j+1) * dt * (std::pow(sd,2) * (j+1) - r);
        c[j] = 0.25 * (j+1) * dt * (std::pow(sd,2) * (j+1) + r);
        b[j] = (1 - 0.5 * std::pow(sd * (j+1), 2) * dt);
    }
```

Listing 14: Implementation of coefficients for matrix T1

Following that, we proceed with the creation of the matrix T_1. Additionally, we set up the initial values for our put option (V^{n+1}) employing our initial put option function. We then perform matrix multiplication between T_1 and V^{n+1} as illustrated in Equation (4.10), subsequently applying the Thomas Algorithm to compute V^n. We also initialize empty matrices for output values as depicted below:

```
    VectorXd f(M-1);
    f.setZero();
    VectorXd final(M-1);
    VectorXd start_psi(M);

    VectorXd psi = init_V(K, dS, M, N);
    MatrixXd T1 = MatrixXd::Zero(M-1, M-1);

    for(int j = 0; j < (M-1); j++) {
        T1(j, j) = b[j];
        if (j < M-2) {
            T1(j, j+1) = c[j];
            T1(j+1, j) = a[j+1];
        }
    }
    VectorXd init_psi = T1 * psi;
```

Listing 15: Initialization of matrices and calculation of put option

The coefficients for matrix T_2 are implemented as follows:

```
    VectorXd a2(M - 1);
    VectorXd d2(M);
    VectorXd c2(M - 1);

    for (int i = 0; i < (M-1); i++) {
        a2[i] = -(0.25 * (i+1) * dt * (std::pow(sd,2) * (i+1) -
  r));
        d2[i] = (1 + (r + 0.5 * std::pow(sd * (i+1), 2)) * dt);
        c2[i] = -(0.25 * (i+1) * dt * (std::pow(sd,2) * (i+1) +
  r));
     }
```

Listing 16: Initialization of T_2

Ultimately, a time loop is employed to apply the Thomas Algorithm in solving for V_n. In options trading, the payoff of a put option signifies the potential profit or value the option holder would gain upon exercising the option at a specific time. For a put option, this payoff is determined by subtracting the current stock price (S) from the strike price (K). By evaluating the current value of V^n against the payoff value $[K - S(T)]^+$, the

code guarantees that the put option's worth at every grid point is not less than what the option holder would earn if they chose to instantly exercise the option. After each iteration of the loop, V^n then transitions to become the subsequent value of V^{n+1}, which is then multiplied by T_1. This process iterates through the loop to calculate the succeeding value of V_n.

```
    for(int j = N; j > 0; j--){

        thomas_tridiag(a2, d2, c2, init_psi, f);

        init_psi = f;

        for(int i = 0; i < (M - 1); i++) {
                init_psi[i] = max(init_psi[i], K - dS * (i+1));
            }

        init_psi = T1 * init_psi;

        }
```

Listing 17: Time loop solving for option prices at various stock prices

Now that we have successfully solved the Black-Scholes Equation numerically using our Crank-Nicolson scheme, it is essential to incorporate boundary conditions into our code. In the following section, we will discuss the implementation of boundary conditions in our code to ensure the fidelity of our numerical solutions.

4.4 Boundary Conditions for BSE

When implementing equation (4.10) in C++, several aspects require clarification. Primarily, the consideration of boundary conditions so we include the $k(n)$ vector. In this regard, we note that the boundary at $t = T$ is determined by the well-defined put option pay-off: whichever value is greater between $K - S(T)$ and 0. Consequently, we specify that the put option's value is zero at all times when $S = S_{\max}$ (this is particularly sensible when $S_{\max} \gg K$), and similarly, that the put option's value is K at all times when $S = 0$. Formally,

$$Boundaries = \begin{cases} V_j^N = [K - j \cdot \Delta S], & \forall j \in [0, M] \\ V_M^n = 0, & \forall n \in [0, N] \\ V_0^n = K, & \forall n \in [0, N] \end{cases} \tag{4.12}$$

Then equation (3.9) becomes:

$$T_1 V^{n+1} + k^{n+1} = T_2 V^n \tag{4.13}$$

which can be then represented in matrix form:

$$\begin{aligned}
&\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{M-1} & b_{M-1} & c_{M-1} \\ & & & a_M & b_M \end{bmatrix} \begin{bmatrix} V_1^{n+1} \\ V_2^{n+1} \\ \vdots \\ \vdots \\ V_{M-1}^{n+1} \end{bmatrix} + \begin{bmatrix} a_1(V_0^n + V_0^{n+1}) \\ 0 \\ \vdots \\ \vdots \\ c_{M-1}(V_M^n + V_M^{n+1}) \end{bmatrix} \\
&= \begin{bmatrix} d_1 & -c_1 & & & \\ -a_2 & d_2 & -c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & -a_{M-1} & d_{M-1} & -c_{M-1} \\ & & & -a_M & d_M \end{bmatrix} \begin{bmatrix} V_1^n \\ V_2^n \\ \vdots \\ \vdots \\ V_{M-1}^n \end{bmatrix}
\end{aligned} \tag{4.14}$$

This is implemented as follows as we solve for V^n:

```
    VectorXd k(M-1);
    k.setZero();

    k[0] = a[0] * (2.0 * K);

    init_psi = init_psi + k;

    for(int j = N; j > 0; j--){

        thomas_tridiag(a2, d2, c2, init_psi, f);

        init_psi = f;

        init_psi = T2 * init_psi;
        init_psi = init_psi + k; //Added into time loop

        }
```

Listing 18: Implementation of BSE Boundary Conditions

With the completion of the code for solving both the TDSE and the BSE using different variants of the Crank-Nicolson schemes, our focus shifts to the analysis of our results. In Section 5 and 6, we will delve into a thorough examination of our findings, evaluating

both the accuracy and computational efficiency of our numerical solutions. This analysis serves to validate the reliability of our numerical methods.

5 Results for TDSE

In this section, we present the results of our numerical simulations conducted to solve the TDSE using various Crank-Nicolson schemes. Through meticulous analysis, we evaluate the accuracy and computational efficiency of our solutions, shedding light on the behavior of quantum systems under different conditions. Additionally, we employ exact solutions, where available, to verify the accuracy of our numerical results and provide further validation of our computational approach. These results provide valuable insights into the effectiveness of our numerical methods.

5.1 Initial wave function

To verify the accuracy of our numerical approach, we examine the Gaussian wave packet mentioned in Equation (2.16):

$$\psi_j^0 = \frac{1}{\sqrt[4]{2\pi\sigma^2}} exp\left[\frac{-(x-x_0)^2}{4\sigma^2} + ikx\right]$$

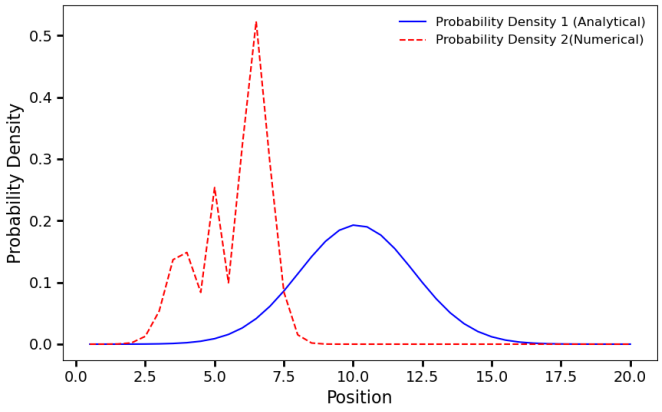
which is centered at x_0, with wave number k and spatial width σ. The exact solution to our initial wave function is given by[7],

$$\psi(x,t) = \sqrt{\frac{2\sigma}{\sqrt{2\pi}(\sigma+it)}} exp\left[\frac{-(x-x_0-kt)^2}{4\sigma^2+2it} + (ik*x-kt/2)\right] \tag{5.1}$$

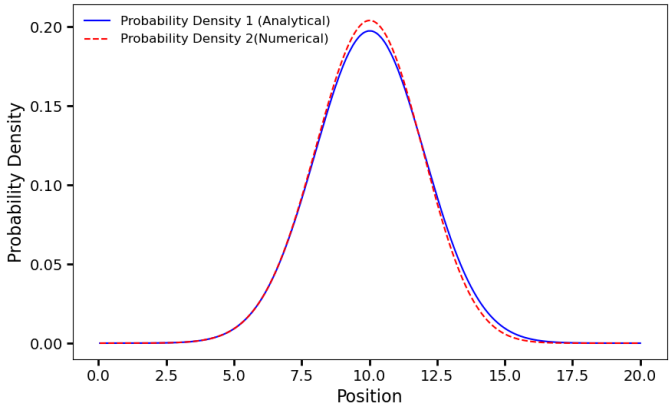
This equation has been incorporated into the code in Section 3.1 and serves as a benchmark to evaluate the precision of our methods.

5.2 Testing of Numerical Schemes

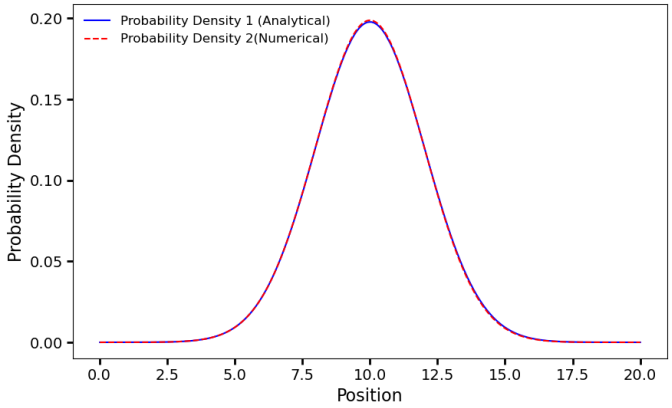
This section aims to clarify the differences between both numerical schemes by examining the accuracy of the Crank-Nicolson methods. The code used for these methods is further validated by ensuring that the numerical solutions converge towards the exact solution as the spatial and temporal discretization approaches zero. The spatial domain size was held constant at 20, while the spatial discretization was varied. Similarly, the temporal domain size remained constant at 1, with variations in temporal discretization. The initial parameters, including $k = 5$, $\sigma = 0.25$, and the initial starting point x_0, were kept unchanged.



(a) $N_x = 21$

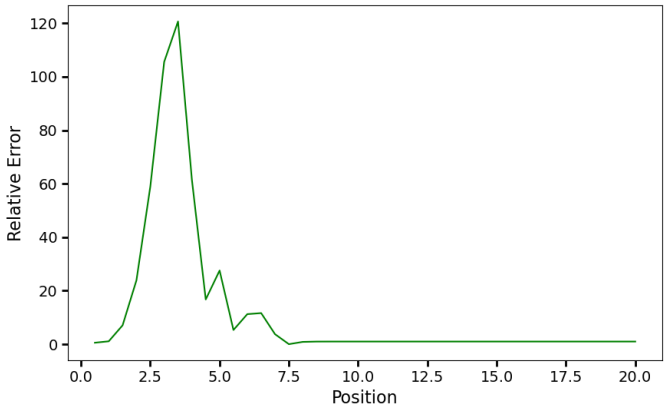


(b) $N_x = 401$

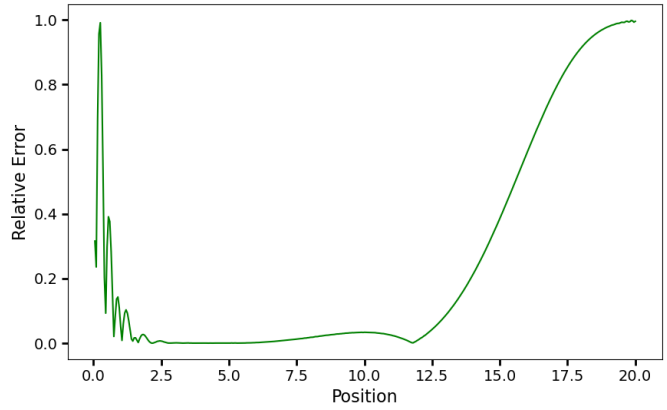


(c) $N_x = 1001$

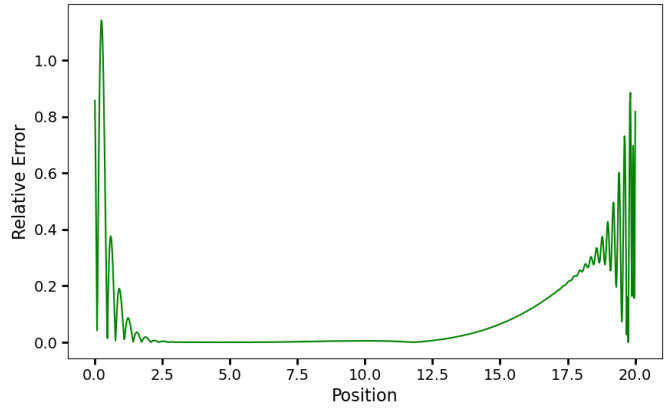
Figure 1: Probability distributions obtained via CN-I under varying levels of discretization.



(a) $N_x = 21$



(b) $N_x = 401$

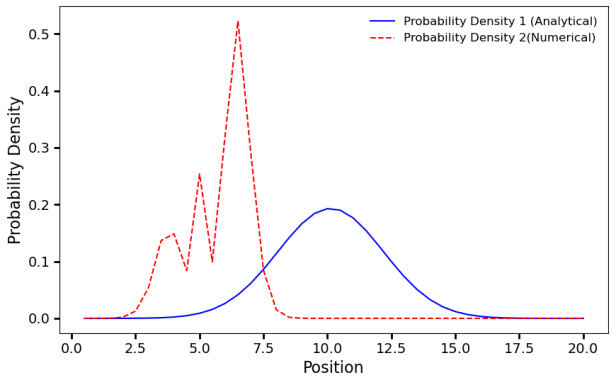


(c) $N_x = 1001$

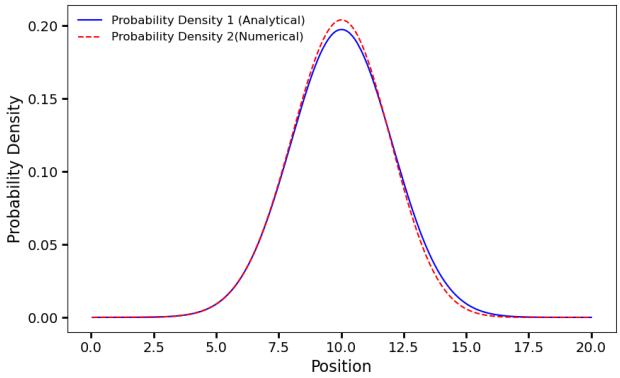
Figure 2: Relative Error between Numerical and Analytical solutions via CN-I under varying levels of discretization

Figure 1a illustrates the Probability distributions obtained via CN-I with limited discretizations, revealing a significant relative error magnitude of approximately 1×10^2 attributed to the small discretization levels in Figure 2a. Figure 1b demonstrates that increasing the number of discretizations results in the solution converging towards the analytical solution, accompanied by a notable decrease in relative error as depicted in Figure 2b. Finally, Figure 1c showcases the near-complete convergence of the numerical solution towards the analytical solution with the highest number of discretizations, with the relative error approaching zero towards the middle of the solution. This underscores the accuracy achieved by the numerical solution. The observed differences between the numerical and analytical solutions, particularly at the edges, may be ascribed to boundary effects.

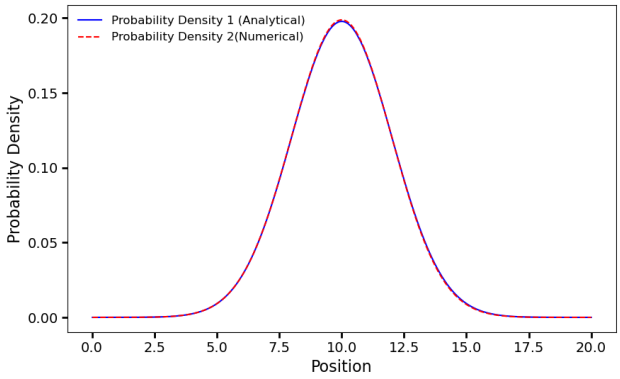
The same process is applied to CN-II below:



(a) $N_x = 21$

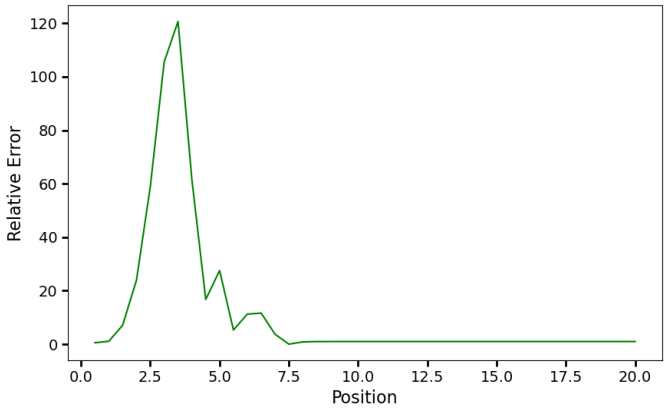


(b) $N_x = 401$

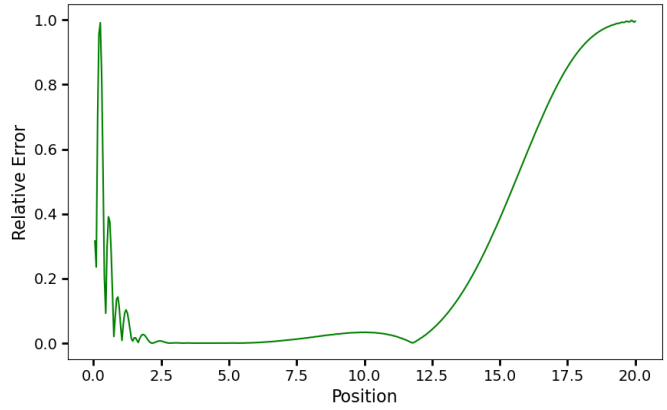


(c) $N_x = 1001$

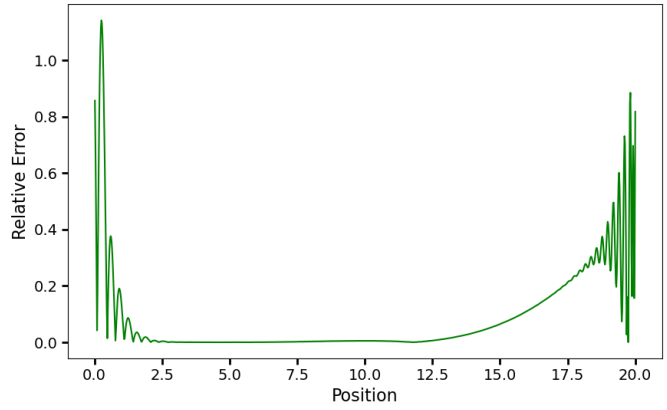
Figure 3: Probability distributions obtained via CN-II under varying levels of discretization.



(a) $N_x = 21$



(b) $N_x = 401$



(c) $N_x = 1001$

Figure 4: Relative Error between Numerical and Analytical solutions via CN-I under varying levels of discretization

In Figure 3, Probability distributions obtained via CN-II across different discretization levels are depicted. CN-II yielded identical results to CN-I, indicating their numerical accuracy. Furthermore, the relative error between CN-II and CN-I solutions was also found to be identical.

When comparing an exact solution with a numerical approximation, the least squares method is employed to quantify the discrepancy between the two. The goal is to minimize the sum of the squared differences between the exact solution y_i and the numerical solution $f(x_i)$ at each data point i, resulting in a single figure that approaches zero when the two solutions are in close agreement. Mathematically, the least squares method is represented as:

$$\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{5.2}$$

where y_i represents the value of the exact solution at data point i, $f(x_i)$ represents the value of the numerical solution at data point i and n denotes the total number of data points. By minimizing this sum of squared residuals, the least squares method provides a quantitative measure of how closely the numerical solution aligns with the exact solution. A resulting value close to zero indicates a high level of agreement between the two solutions.

Δx	Δt	N_x	L
1	0.05	20	1.097
0.67	0.0335	30	0.469
0.44	0.022	45	0.896
0.296	0.0148	68	1.584
0.198	0.0099	101	0.761
0.131	0.00655	153	0.111
0.088	0.0044	228	0.023
0.059	0.0025	339	0.006
0.039	0.002	513	0.002

(a) CN-I

Δx	Δt	N_x	L
1	0.05	20	1.097
0.67	0.0335	30	0.481
0.44	0.022	45	0.896
0.296	0.0148	68	1.584
0.198	0.0099	101	0.761
0.131	0.00655	153	0.111
0.088	0.0044	228	0.023
0.059	0.0025	339	0.006
0.039	0.002	513	0.002

(b) CN-II

Table 1: Comparison of least square errors for second-order spatial accuracy in CN-I and CN-II where L is the sum of squared errors.

The behavior observed in Table 1 can be attributed to the effects of discretization and numerical precision. With a small number of data points, the discretization may not accurately capture the solution's nuances, leading to larger discrepancies between the numerical and analytical solutions.

However, as the number of data points increases, the discretization becomes finer, allowing for a more accurate representation of the solution. This finer discretization helps reduce truncation errors associated with numerical methods, leading to a more consistent convergence of the least square error values toward 0. Furthermore, with more data points, the influence of numerical precision errors tends to diminish, further contributing to the convergence of the least square error values.

5.3 Computational Efficiency

In theory, explicit methods are renowned for their speed as they avoid the need to invert large matrices for computing the next step, while implicit methods often entail such inversions to calculate subsequent steps. Both numerical approaches demonstrated remarkable swiftness. However, the straightforward solution of the eigenvalue equation of A or calculation of its inverse A^{-1} is generally a formidable task given how the calculation time and storage costs scale with the size N of the problem, roughly as N^3 and N^2, respectively. One has to rely on specialized sophisticated techniques to proceed with the solution. Additionally, CN-I method requires utilizing the solution of the eigenvalue equation of A or calculation of its inverse A^{-1}, which further contributes to its computational complexity. The CN-II (explicit) method surpasses CN-I (implicit) in speed by a significant margin, achieved by eliminating the matrix-vector operation and thus enhancing the solution process efficiency. The table below shows multiple simulations that were conducted using both CN methods, while varying the number of spatial discretizations, and computing the processing time necessary to advance the wave function for a single step.

Δt	Δx	N_x	CN-I	CN-II
0.0025	0.05	401	$1.33 \times 10^1 s$	$2.00 \times 10^{-3} s$
0.0025	0.02	1001	$1.63 \times 10^2 s$	$7.30 \times 10^{-1} s$
0.0025	0.01	2001	$1.40 \times 10^3 s$	$1.25 \times 10^0 s$

Table 2: Calculation speed per time step

As the CN method demonstrates unconditional stability, there exist no limitations on the relationships between the spatial and temporal step sizes. The CN-II method could achieve a notably faster solution compared to the CN-I method, while still maintaining a reasonably accurate result within the same simulated time frame.

The computational efficiency of the numerical solution was influenced by the specifications of the hardware used for simulations. The computational tasks were executed on a 2015 MacBook Air, equipped with a 1.6 GHz Dual-Core Intel Core i5 processor and 8GB of RAM. Additionally, the Mac features a Intel HD Graphics 6000 1536 MB graphics processing unit (GPU) and utilizes a solid-state drive (SSD) for storage. These hardware components collectively contributed to the overall performance and speed of the numerical computations.

6 Results for BSE

Partial differential equations (PDEs), including the Black-Scholes Equation, can sometimes be solved exactly using analytical techniques, but only under specific conditions or simplifications. In more general cases, analytical solutions may not be feasible due to the complexity of the equations or the absence of closed-form solutions. As a result, numerical methods are often employed to approximate solutions, providing a practical approach to addressing a wide range of PDEs. This highlights the importance of both analytical and numerical methods in tackling PDEs in finance and other fields, where exact solutions may be elusive but approximations can still offer valuable insights. We will expand upon our findings by conducting a comparative analysis with other numerical schemes to validate the robustness of our solution.

6.1 Evaluation of the Crank-Nicolson Method for Pricing American Put Options

The code structure ensures modularity, facilitating easy implementation of additional functionalities like an iterative linear equation solver. Input parameters such as maturity, stock value, interest rate, volatility, strike price, and intervals can be set in the main function. The code outputs a table displaying put option values over discrete stock prices and time steps. Notably, put values evolve backward in time, reflecting changes in stock prices. The following is the output generated by the code:

Stock	0.0	5.0	10.0	15.0	20.0	25.0	30.0	35.0	40.0	45.0	50.0	55.0	60.0	65.0	70.0	75.0	80.0	85.0	90.0	95.0	100.0
Time																					
0.42	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	5.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.38	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	5.00	1.18	0.16	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.33	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	5.17	1.87	0.50	0.12	0.03	0.01	0.00	0.00	0.00	0.00	0.00	0.00
0.29	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	5.39	2.34	0.84	0.26	0.08	0.02	0.01	0.00	0.00	0.00	0.00	0.00
0.25	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	5.63	2.72	1.15	0.44	0.16	0.05	0.02	0.01	0.00	0.00	0.00	0.00
0.21	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	5.84	3.03	1.42	0.62	0.25	0.10	0.04	0.02	0.01	0.00	0.00	0.00
0.17	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.00	6.04	3.31	1.67	0.80	0.36	0.16	0.07	0.03	0.01	0.01	0.00	0.00
0.13	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.03	6.21	3.55	1.90	0.97	0.47	0.23	0.11	0.05	0.02	0.01	0.00	0.00
0.08	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.07	6.37	3.77	2.12	1.14	0.59	0.30	0.15	0.07	0.04	0.02	0.01	0.00
0.04	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.14	6.52	3.97	2.31	1.30	0.71	0.38	0.20	0.11	0.05	0.03	0.01	0.00
0.00	50.00	45.00	40.00	35.00	30.00	25.00	20.00	15.00	10.20	6.66	4.16	2.50	1.45	0.83	0.46	0.26	0.14	0.08	0.04	0.02	0.00

Figure 5: Option prices based on varying stock prices over time

In this particular scenario, we apply the following parameters: $T = 0.4167$ years (equivalent to 5 months), $S_{\max} = 100$, $r = 0.10$, $\sigma = 0.40$, $K = 50$, $N = 10$, and $M = 20$. Consequently, put option prices are computed twice a month, starting from maturity T and ending at the present time $t = 0$. The table presented above illustrates all discretized stock prices in the initial row and all discretized time steps in the initial column. The option's value can be assessed at any point leading up to its maturity and across various stock prices. Observing where the stock price equals zero (column 0.00) and simultaneously the option was purchased (row 0.47), the put option is valued

at \$50 because the stock holds no worth and can be sold for the same price it was listed. When the maximum possible stock value, denoted as $S_{\max}$, is attained, the put option's value becomes negligible or essentially zero. This indicates that the stock price has potentially risen to a level where the put option loses its value due to its exercise price being significantly below the current market price. As a result, investors would likely not exercise the put option at $S_{\max}$ since they could sell the stock at a higher price in the market. To determine the value of an American put option today, given a stock price of $S = 55$, one would refer to row 0.00 and column 55.0, resulting in a value of $V(S = 55) = \$2.50$. We can illustrate the movement of option prices over time corresponding to different stock prices.

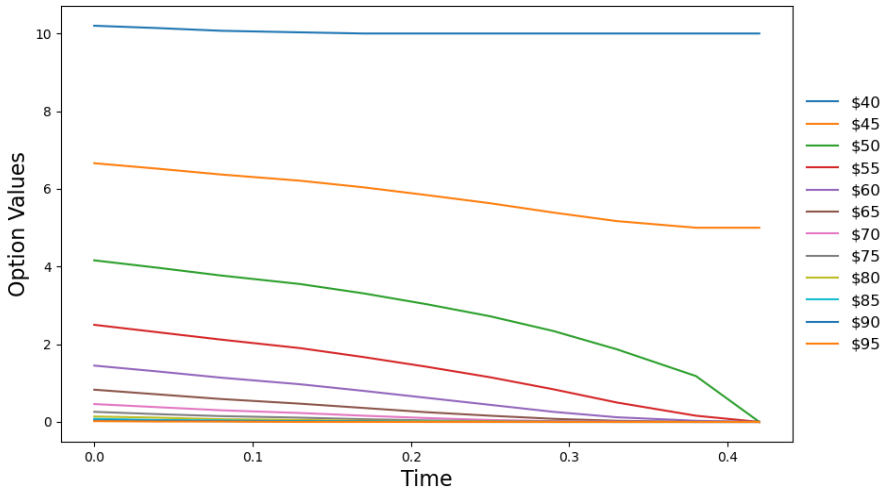


Figure 6: Variation in Option Prices Over Time Relative to Stock Prices

6.2 Testing Black-Scholes Equation Numerical Schemes

By contrasting this particular Crank-Nicolson scheme utilized for solving the Black-Scholes equation with alternative implementations of the Crank-Nicolson method, we can evaluate its precision. The alternative Crank-Nicolson algorithm extracted from a reference paper will serve as our analytical benchmark. The relative error between the numerical solution and the analytical solution is computed as follows:

$$\epsilon = \left| \frac{\text{numerical solution - comparative solution}}{\text{comparative solution}} \right| \tag{6.1}$$

This equation measures the difference between the numerical and analytical solutions relative to the analytical solution itself. It provides insight into the accuracy of the numerical approximation compared to the exact solution. The comparison is carried out on the current day when the put option is written.

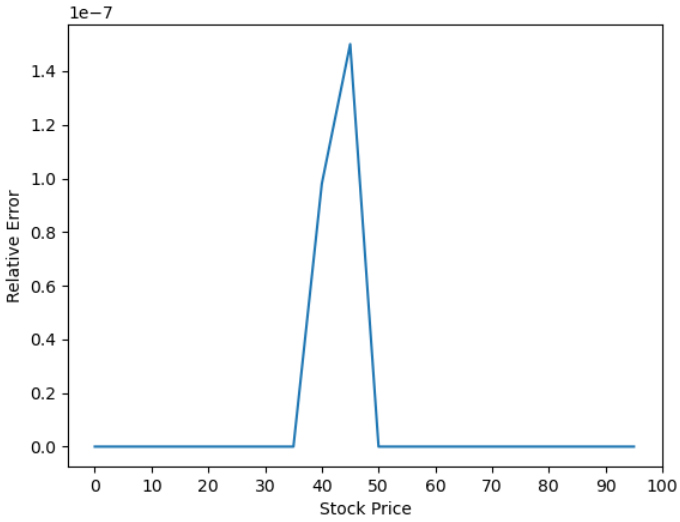


Figure 7: Relative Error between Numerical and Analytical Solution

The comparison between our numerical solution of the Black-Scholes Equation, taken as if the option was written for current day, and the comparative solution reveals an outstanding level of agreement. Utilizing the least squares metric to quantify the disparity between our numerical findings and the comparative solution, we obtained a value of 2.0×10^{-12}. This demonstrates the precision of our numerical solution, as it approaches zero.

7 Summary and Discussion

In this study, we investigated various numerical methods to solve the Time-Dependent Schrödinger Equation and the Black-Scholes Equation. After thorough analysis, we chose the Crank-Nicolson Method for its accuracy and computational efficiency. Our numerical solutions closely matched analytical solutions, with refinement in discretization leading to convergence. Interestingly, both variants of the Crank-Nicolson Method produced identical results, showcasing consistency. Transitioning to the Black-Scholes Equation, our numerical model accurately priced options, validated through comparison with external solutions.

Our findings underscore the effectiveness of the Crank-Nicolson Method in accurately solving complex equations in quantum mechanics and finance. The observed convergence with analytical solutions highlights the reliability of our numerical approach. The consistency between CN-I and CN-II variants suggests robustness across implementations. Moreover, the successful application of our numerical model to price options in the financial realm demonstrates its practical utility and adaptability.

The accuracy and efficiency of our numerical methods have significant implications for both scientific research and financial practice. In scientific research, our approach facilitates the exploration of quantum mechanical phenomena with high precision, enabling deeper insights into the behavior of quantum systems. In finance, our model offers a reliable tool for option pricing, aiding investors and financial analysts in making informed decisions and managing risks effectively.

Despite our successes, there are limitations to consider. Our numerical solutions rely on assumptions and simplifications inherent in numerical modeling, which may not fully capture the complexities of real-world systems. Additionally, computational efficiency may vary depending on hardware capabilities and algorithm optimizations. Incorporating real stock market data introduces challenges related to data accuracy, latency, and model calibration, which require careful consideration.

Looking ahead, further exploration of advanced numerical techniques and model refinements is recommended to address limitations and enhance accuracy. Additionally, integrating real-time stock market data into the numerical framework offers exciting opportunities for dynamic option pricing and predictive modeling. Collaborations with experts in finance and computational science have the potential to yield innovative solutions with broader applications in quantum mechanics, finance, and beyond.

References

[1] Charles D. Joyner. Black-scholes equation and heat equation. *Georgia Southern University*, 2016.

[2] Wolfgang Paul and Jörg Baschnagel. Stochastic processes. *From Physics to finance*, 1999.

[3] Wikipedia. Schrödinger equation. Accessed: April 9, 2024.

[4] M.S. Joshi. *The Concepts and Practice of Mathematical Finance*. Mathematics, Finance and Risk. Cambridge University Press, 2003.

[5] Raymond H. Chan, Yves ZY. Guo, Spike T. Lee, and Xun Li. *Financial Mathematics, Derivatives and Structured Products*. Springer Link, 2019.

[6] J. Crank. *The Mathematics of Diffusion*. Oxford science publications. Clarendon Press, 1979.

[7] Daniel Hockey. Numerical methods for solving the time- dependent schrödinger equation. *Dublin City University*, 2020.

[8] Lampros AA Nikolopoulos. *Computing Atomic Quantum Dynamics in Laser Fields*. AIP Publishing, 2022.

[9] Simon Ellersgaard Nielsen. Computational finance with c++ (course project). *Imperial College London*, 2012.

8 Appendices

```cpp
#include <iostream>
#include "Eigen/Dense"
#include <complex>
#include <iomanip>
#include <fstream>
#include <stdexcept>

using namespace std;
using namespace Eigen;

VectorXcd init_psi(const VectorXd& x, double x_0, double k, double sigma) {
    VectorXcd result(x.size());
    double norm, realPart, imagPart;

    norm = std::pow((2 * M_PI) * (std::pow(sigma, 2)), -0.25);

    for (int i = 0; i < x.size(); i++) {
        realPart = -(x(i) - x_0) * (x(i) - x_0) / (4 * sigma * sigma);

        imagPart = k * x(i);

        result(i) = norm * std::exp(std::complex<double>(realPart, 0)) * std::exp(std::complex<double>(0, imagPart));
    }

    return result;
}

VectorXd generateEvenlySpacedIntervals(double start, double end, double interval) {
    int size = static_cast<int>((end - start) / interval) + 1;
    VectorXd intervals(size);

    for (int i = 0; i < size; i++) {
        intervals(i) = start + i * interval;
    }

    return intervals;
}

VectorXcd exact_psi(const VectorXd& x, double x_0, double k,
```

```
    double sigma, double dt, int Nx, int Nt) {
40      VectorXcd result(Nx);
41      complex<double> norm, gaussian_term, phase_term;
42      double time;
43      complex<double> zi(0.0, 1.0);
44
45      for (int i = 0; i < Nx; i++) {
46          time = Nt * dt;
47          norm = pow(pow(2.0 * M_PI, 0.25) * sqrt(sigma + (time *
    zi) / (2.0 * sigma)), -1);
48
49          gaussian_term = -(pow((x(i) - x_0 - k * time),2))/ (4 *
    pow(sigma, 2) + 2.0 * zi * time) + (zi*k*(x[i] - k*time/2.0))
    ;
50          result[i] = norm * exp(gaussian_term);
51
52      }
53      return result;
54  }
55
56  int main() {
57      std::complex<double> zi(0.0, 1.0);
58
59      double x_0, sigma, k;
60      x_0   = 5.0;
61      k     = 5.0;
62      sigma = 0.25;
63
64      double start_x, end_x, dx;
65      start_x   = 0;
66      end_x     = 20;
67      dx       = 0.05;
68
69      double start_t, end_t, dt;
70      start_t = 0;
71      end_t   = 1;
72      dt = dx/20;
73
74      double hb, m;
75      hb = 1;
76      m = 1;
77
78      VectorXd x_i = generateEvenlySpacedIntervals(start_x, end_x,
    dx);
79      VectorXd t_n = generateEvenlySpacedIntervals(start_t, end_t,
    dt);
```

```
80
81      int Nx, Nt;
82      Nx = x_i.size();
83      Nt = t_n.size();
84
85      MatrixXcd iden = MatrixXcd::Identity(Nx, Nx);
86
87      Eigen::MatrixXcd T = Eigen::MatrixXcd::Zero(Nx, Nx);
88      Eigen::MatrixXcd V = Eigen::MatrixXcd::Zero(Nx, Nx);
89
90      T.diagonal().array() = -2.0;
91      T.diagonal(1).array() = T.diagonal(-1).array() = 1.0;
92      V.diagonal().array() = 0;
93
94      T *= ((-hb * hb) / (2.0 * m * dx * dx));
95
96      Eigen::MatrixXcd H = T + V;
97
98      Eigen::MatrixXcd FTCS = iden - ((std::complex<double>(0.0,
    1.0) * dt / (2.0 * hb)) * H);
99      Eigen::MatrixXcd BTCS = iden + ((std::complex<double>(0.0,
    1.0) * dt / (2.0 * hb)) * H);
100
101     MatrixXcd CN2 = FTCS * BTCS.inverse();
102
103     VectorXcd psi = init_psi(x_i, x_0, k, sigma);
104     VectorXcd psi_t;
105
106     for (int it = 0; it < Nt; it++) {
107         psi_t = CN2 * psi ;
108         psi = psi_t;
109
110     }
111
112     std::ofstream outputFile("psi-t-norm-cn.csv");
113
114     if (!outputFile.is_open()) {
115         std::cerr << "Failed to open the file for writing." <<
    std::endl;
116         return 1;
117     }
118
119     int precision = 3;
120     outputFile << std::scientific;
121
122     double nrm_t;
```

```
123     VectorXd num_sol(Nx);
124 
125 
126     for (int i = 0; i < Nx; i++) {
127 
128       nrm_t = abs(psi_t[i]) * abs(psi_t[i]);
129       outputFile << x_i[i] << ',' << nrm_t << std::endl;
130       num_sol[i] = nrm_t;
131 
132     }
133     outputFile.close();
134 
135 
136     VectorXcd exact_psi = exact_psi(x_i, x_0, k, sigma, dt, Nx,
    Nt);
137 
138     std::ofstream outputFile1("ex-psi-initial-norm.csv");
139     if (!outputFile1.is_open()) {
140         std::cerr << "Failed to open the file for writing." <<
    std::endl;
141         return 1;
142     }
143 
144     outputFile1.precision(precision);
145     outputFile1 << std::scientific;
146 
147 
148     double exact_psi_abs, realPart_tc, imagPart_tc;
149     VectorXd exact_sol(Nx);
150     for (int i = 0; i < Nx; i++){
151 
152       exact_psi_abs = abs(exact_psi[i]) * abs(exact_psi[i]);
153       exact_sol[i] = exact_psi_abs;
154 
155       outputFile1 << x_i[i] << ',' << exact_psi_abs << endl;
156     }
157 
158     outputFile1.close();
159 
160     cout << "end" << endl;
161 }
```

Listing 19: Code in C++ utilizing LU Decomposition for CN-I.

```cpp
#include <iostream>
#include "Eigen/Dense"
#include <vector>
#include <complex>
#include <iomanip>
#include <fstream>
#include <stdexcept>

using namespace std;
using namespace Eigen;

VectorXcd init_psi(int N, double x_0, double k, double sigma, double dx) {

    VectorXcd result(N);
    double norm, realPart, imagPart;

    norm = std::pow((2 * M_PI) * (std::pow(sigma, 2)), -0.25);

    for (int i = 0; i < N; i++) {
        realPart = -(i * dx - x_0) * (i * dx - x_0) / (4 * sigma * sigma);
        imagPart = k * i * dx;

        result(i) = norm * std::exp(std::complex<double>(realPart, 0)) * std::exp(std::complex<double>(0, imagPart));
    }

    return result;
}

VectorXcd exact_psi(double x_0, double k, double sigma, double dt, int Nx, int Nt, double dx) {
    VectorXcd result(Nx);
    complex<double> norm, gaussian_term, phase_term;
    double time;
    complex<double> zi(0.0, 1.0);

    for (int i = 0; i < Nx; i++) {
        time = Nt * dt;
        norm = pow(pow(2.0 * M_PI, 0.25) * sqrt(sigma + (time * zi) / (2.0 * sigma)), -1);
        gaussian_term = -(pow((dx * i - x_0 - k * time),2))/ (4 * pow(sigma, 2) + 2.0 * zi * time) + (zi*k*(dx * i - k*time/2.0));
        result[i] = norm * exp(gaussian_term);
```

```
40
41      }
42      return result;
43  }
44
45  VectorXd generateEvenlySpacedIntervals(double start, double end,
        double interval) {
46      int size = static_cast<int>((end - start) / interval) + 1;
47      VectorXd intervals(size);
48
49      for (int i = 0; i < size; i++) {
50          intervals(i) = start + i * interval;
51      }
52
53      return intervals;
54  }
55
56  void thomas_tridiag(const VectorXcd& a, const VectorXcd& b,
       const VectorXcd& c, const VectorXcd& psi, VectorXcd& u) {
57      int j, n;
58      complex<double> bet;
59
60      n = a.size();
61
62      VectorXcd gam(n);
63
64      if (b[0] == 0.0) {
65          throw("Error 1 in tridag");
66      }
67
68      u[0] = psi[0] / (bet = b[0]);
69
70      for (j = 1; j < n; j++) {
71
72          gam[j] = c[j - 1] / bet;
73          bet = b[j] - a[j] * gam[j];
74
75          if (bet == 0.0)
76              throw("Error in tridag");
77
78          u[j] = (psi[j] - a[j] * u[j - 1]) / bet;
79      }
80
81      for (j = (n - 2); j >= 0; j--)
82          u[j] -= gam[j + 1] * u[j + 1];
83  }
```

```
84
85 int main() {
86
87     std::complex<double> zi(0.0, 1.0);
88
89     double x_0, sigma, k;
90
91     x_0   = 5.0;
92     k     = 5.0;
93     sigma = 0.25;
94     double h = 1;
95     double m = 1;
96
97     double start_x, end_x, dx;
98     start_x   = 0;
99     end_x     = 20;
100    dx        = 1;
101
102    double start_t, end_t, dt;
103    start_t = 0;
104    end_t   = 1;
105    dt = dx/20;
106
107    VectorXd x_i = generateEvenlySpacedIntervals(start_x, end_x,
       dx);
108    VectorXd t_n = generateEvenlySpacedIntervals(start_t, end_t,
       dt);
109
110    int Nx, Nt;
111    Nx = x_i.size();
112    Nt = t_n.size();
113
114    VectorXcd a(Nx - 1);
115    a.setConstant(complex<double>(0, -dt * h / (4 * m * pow(dx,
     2))));
116
117    VectorXcd b(Nx);
118    //b.setConstant(1.0 + r);
119    b.setConstant(complex<double>(1, dt * h / (2 * m * pow(dx,
     2))));
120
121    VectorXcd c(Nx - 1);
122    c.setConstant(complex<double>(0, -dt * h / (4 * m * pow(dx,
     2))));
123
124    VectorXcd f(Nx);
```

```
125     f.setZero();
126     VectorXcd psi = init_psi(Nx, x_0, k, sigma, dx);
127 
128     VectorXcd final(Nx);
129 
130 
131      for (int i = 0; i < Nt; ++i) {
132 
133         thomas_tridiag(a, b, c, psi, f);
134 
135         VectorXcd final = 2*f - psi;
136 
137         psi = final;
138     }
139 
140 
141     ofstream outfile;
142     outfile.open("cn.csv");
143 
144     double nrm_t;
145     VectorXd num_sol(Nx);
146 
147     for (int i=0; i < Nx; i++){
148         nrm_t = abs(psi[i]) * abs(psi[i]);
149         num_sol[i] = nrm_t;
150         outfile << dx * i << "," << nrm_t << endl;
151     }
152 
153     outfile.close();
154 
155     VectorXcd exact_psi = exact_psi(x_0, k, sigma, dt, Nx, Nt,
    dx);
156 
157     std::ofstream outputFile3("ex-psi-initial-norm2.csv");
158     if (!outputFile3.is_open()) {
159         std::cerr << "Failed to open the file for writing." <<
    std::endl;
160         return 1;
161     }
162 
163     outputFile3 << std::scientific;
164 
165     double exact_psi_abs, realPart_tc, imagPart_tc;
166     VectorXd exact_sol(Nx);
167     for (int i = 0; i < Nx; i++){
168 
```

```
169         exact_psi_abs = abs(exact_psi[i]) * abs(exact_psi[i]);
170         exact_sol[i] = exact_psi_abs;
171 
172         outputFile3 << dx * i << ',' << exact_psi_abs << endl;
173       }
174 
175       cout << "end" << endl;
176 
177       return 0;
178 }
```

Listing 20: Code in C++ utilizing Thomas Algorithm for CN-II.

```cpp
#include <iostream>
#include "Eigen/Dense"
#include <vector>
#include <complex>
#include <iomanip>
#include <fstream>
#include <stdexcept>

using namespace std;
using namespace Eigen;

//Initial Values of stocks
VectorXd init_V(double K, double dS, int Nt, int Nx) {

    VectorXd V(Nt - 1);

    for(int j = 0; j < (Nt - 1); j++) {
        V[j] = std::max(K - dS * (j+1), 0.0);
    }

    return V;
}

//Thomas Algorithim
void thomas_tridiag(const VectorXd& a, const VectorXd& b, const
   VectorXd& c, const VectorXd& psi, VectorXd& u) {
    int j, n;
    double bet;

    n = a.size();

    VectorXd gam(n);

    if (b[0] == 0.0) {
        throw("Error 1 in tridag");

    }

    u[0] = psi[0] / (bet = b[0]);

    for (j = 1; j < n; j++) {

        gam[j] = c[j - 1] / bet;
        bet = b[j] - a[j] * gam[j];

        if (bet == 0.0)
```

```
            throw("Error in tridag");

        u[j] = (psi[j] - a[j] * u[j - 1]) / bet;
    }

    for (j = (n - 2); j >= 0; j--)
        u[j] -= gam[j + 1] * u[j + 1];
}


int main() {
//Variables
    int N  = 10;
    int M  = 20;
    double T = 0.4167;
    double Smax = 100;
    double r = 0.1;
    double sd = 0.4;
    double K = 50.0;
    double dS = Smax/M;
    double dt = T/N;



//Initialize Matrices
    VectorXd a(M - 1);
    VectorXd b(M);
    VectorXd c(M - 1);

    double sd2 = std::pow(sd,2);
    for(int j = 0; j < (M-1); j++) {
        a[j] = 0.25 * (j+1) * dt * (sd2 * (j+1) - r);
        c[j] = 0.25 * (j+1) * dt * (sd2 * (j+1) + r);
        b[j] = (1 - 0.5 * std::pow(sd * (j+1), 2) * dt);
    }



//Empty Vector / Becomes Output Vector
    VectorXd f(M-1);
    f.setZero();
    VectorXd final(M-1);
    VectorXd start_psi(M);
    VectorXd k(M-1);
    k.setZero();

```

```
92 //Initial Values Set
93     VectorXd psi = init_V(K, dS, M, N);
94     MatrixXd T2 = MatrixXd::Zero(M-1, M-1);
95 
96     for(int j = 0; j < (M-1); j++) {
97         T2(j, j) = b[j];
98         if (j < M-2) {
99             T2(j, j+1) = c[j];
100            T2(j+1, j) = a[j+1];
101        }
102    }
103 
104    k[0] = a[0] * (2.0 * K);
105 
106    VectorXd init_psi = T2 * psi;
107    init_psi = init_psi + k;
108 
109 //Matrix Coefficients
110    VectorXd a2(M - 1);
111    VectorXd d2(M);
112    VectorXd c2(M - 1);
113 
114    for (int i = 0; i < (M-1); i++) {
115        a2[i] = -(0.25 * (i+1) * dt * (sd2 * (i+1) - r));
116        d2[i] = (1 + (r + 0.5 * std::pow(sd * (i+1), 2)) * dt);
117        c2[i] = -(0.25 * (i+1) * dt * (sd2 * (i+1) + r));
118    }
119 
120    std::cout << std::setw(7) << "Stock:";
121        for(int i = 0; i < (M+1); i++) {
122        std::cout << std::setw(6) <<  std::fixed << std::
   setprecision(2) << std::setprecision(1) << i * dS;
123        }
124 
125        std::cout << std::endl << std::setw(7) << "Time:";
126        for(int i = 0; i < (M+1); i++) {
127            std::cout << std::setw(6) << "------";
128        }
129 
130        std::cout << std::endl;
131 
132        std::cout << std::setw(6) << std::setprecision(2) << N *
    dt << "| ";
133        std::cout << std::setw(6) << K;
134 
135        for(int j = 0; j < (M-1); j++) {
```

```
136             std::cout << std::setw(6) << std::fixed << std::
    setprecision(2) << psi[j];
137         }
138 
139         std::cout << std::setw(6) << "0.00" << std::endl;
140 
141 
142      for(int j = N; j > 0; j--){
143 
144         thomas_tridiag(a2, d2, c2, init_psi, f);
145 
146         init_psi = f;
147 
148         std::cout << std::setw(6) << std::setprecision(2) << (j
    -1) * dt << "| ";
149         std::cout << std::setw(6) << K;
150 
151 
152         for(int i = 0; i < (M - 1); i++) {
153                 init_psi[i] = max(init_psi[i], K - dS * (i+1));
154                 std::cout << std::setw(6) << std::fixed << std::
    setprecision(2) << init_psi[i];
155             }
156         std::cout << std::setw(6) << "0.00" << std::endl;
157 
158         init_psi = T2 * init_psi;
159         init_psi = init_psi + k;
160 
161         }
162 
163      return 0;
164 }
```

Listing 21: Code in C++ utilizing Thomas Algorithm for BSE.