School of Computing

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Solving a Nonlinear Partial Differential Equation as a Nonlinear Matrix Equation

Kanishk Jha

Submitted in accordance with the requirements for the degree of MSc in Advanced Computer Science

Session 2022/2023

The candidate confirms that the following have been submitted*:*

<As an example>

Items	**Format**	**Recipient(s) and Date**
Deliverables 1, 2, 3	*Report*	*SSO (xx/xx/xx)*
Participant consent forms	*Signed forms in envelop*	*SSO (xx/xx/xx)*
Deliverable 4	*Software codes or URL*	*Supervisor, assessor (xx/xx/xx)*
Deliverable 5	*User manuals*	*Client, supervisor (xx/xx/xx)*

Type of Project: Exploratory Software

Summary

A standard approach to numerically approximating the solution of a time and space-dependent deterministic Partial Differential Equation (PDE) is discretising the unknown function variable across the space dimensions, and obtaining a system of ordinary differential equations in time. This method is called the Method of Lines. It is particularly used in well-posed mathematical problems with properly defined boundary conditions in a certain region. For a 2-dimensional problem, the discretisation leads to studying a 2-dimensional mesh consisting of discrete nodes. The matrix containing the grid-node values is vectorised to obtain a solution at each node, and a linear system of equations of the form **Ax=b** is solved. An important issue with this approach is that as the size of the problem increases for finer grids the memory requirements and computational complexity of certain well-established iterative methods become increasingly unfeasible. This problem becomes even more complex for nonlinear partial differential equations as a system of nonlinear equations has to be solved. Another issue is that this approach ignores the complete structure of the grid that defines the unknown function, instead opting for methods that focus more on the equation at each node. While this approach is mathematically correct, it is not very intuitive and does not deal with the unique aspects arising from the discretisation of the various terms present in the PDE that define the equation. This project aims to solve such PDEs without vectorising the grid by formulating a matrix equation to represent the discretisation. The focus of this project is purely on nonlinear matrix equations obtained from nonlinear deterministic PDEs. While there is ample literature on the different methods used in this project, a sparse amount of research exists in forming an algorithm that solves a problem of this complexity by bringing together these methods. Thus, this project also aims to bridge the research gap in the form of a solver that achieves this via some modification to the aforementioned methods. To this effect, it aims to present a solution that is tractable and comparable to the standard approach and provides an accurate approximation of the solution.

Acknowledgements

<This page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by others to the project.

Note that it is not acceptable to solicit assistance on 'proof reading' which is defined as "the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the text"; see [*http://www.leeds.ac.uk/qat/documents/policy/Proof-reading-policy.pdf*](http://www.leeds.ac.uk/qat/documents/policy/Proof-reading-policy.pdf)*. >*

Table of Contents

Chapter 1
Introduction

The project involves a theoretical study of different methods that exist in the domain of the research question posed, and how they are utilised in bridging the research gap between known methods of solving linear matrix equations and unknown approaches to solving nonlinear matrix equations, in the context of nonlinear partial differential equations. This is then followed by scripts in the MATLAB programming language to present software capable of doing so practically. Thus, the project is both an endeavour in theoretical study and exploratory software.

1.1 Context

Differential equations model the behaviour of physical entities that are characterised by change. They are used to define phenomena across a vast array of disciplines, including physics, biology, engineering and economics (insert reference here). They are considered to be the most intuitive approach to representing systems that change continuously with respect to some parameter which is usually represented as one of the many independent variables on which the equation is based. One of the most important uses of differential equations in recent years has been in the numerical simulation of weather patterns for weather forecasting (insert reference here). Due to their importance, it is essential to categorise them properly to obtain an accurate solution to a particular family of equations. A widely recognised scheme involves categorising differential equations as either Ordinary Differential Equations or Partial Differential Equations. Ordinary differential equations refer to differential equations where the unknown function depends on only one independent parameter or variable.

Partial differential equations, or PDEs, are equations in which the unknown function, usually denoted by **u**, that models the physical entity under study depends on more than one independent variable. Despite their importance in modelling many real-life phenomena, most PDEs do not have closed-form solutions, i.e., solutions that can be computed as the result of an explicit formula. Usually, certain regions and conditions can be analytically solved, but computing even these solutions become increasingly complex as the dimensionality of the problem grows. Therefore, the preferred approach is numerically approximating the solution up to a certain acceptable tolerance of error (or accuracy).

Many methods and approaches have been studied and used for this numerical approximation process. A standard approach is called the Method of Lines (or Numerical Method of Lines) (Scheisser Book) which discretises all but one of the continuous variables on which the unknown function depends, and as a result obtains a system of Ordinary

Differential Equations (ODEs), which are then solved by relevant solvers. For unknown functions that depend on both space and time, the discretisation is usually done across the different space axes, and the resulting system of ODEs in time is solved via different time-stepping schemes. These time-stepping schemes further convert the system of ODEs into a system of linear or nonlinear equations, based on the linearity or nonlinearity of the original PDE. Linear systems of the form **Ax=b**, where **A** is a matrix and **b** and **x** are vectors of conforming dimensions, can be solved directly via well-established linear solvers like Gaussian elimination (insert ref here). For a system of nonlinear equations of this scale, the general approach is to use iterative methods like the Newton-Raphson (insert ref here) or Gradient Descent (insert ref here) to converge to a solution by linearising the system into a linear form similar to **Ax=b,** and solving this at every step. Irrespective of the approach taken, the variables representing the unknown function at every discrete point on the mesh line or grid are stored as an array or vector to facilitate the aforementioned formulations.

1.2 Project Aim

This project aims to numerically approximate the solution of a time-dependent non-linear 2-dimensional deterministic PDE via the method of lines approach while ensuring that the grid structure of this discretisation is preserved as a matrix. The project's primary goal is to achieve a tractable approach to solving such a nonlinear PDE through a nonlinear matrix equation solver, instead of standard linear system solvers. To this end, the time-dependent 2-dimensional Burger's equation is studied and a matrix-based approach is applied to solving it.

As previously mentioned, the standard approach involves storing the variables as a vector to better formulate the problem as a system of linear equations of the form **Ax=b**. This formulation represents the discretised form of the PDE correctly for each node in the grid and is a mathematically correct way of representing the form. However, there are a few issues with this approach.

Firstly, we face certain computational challenges when trying to improve the accuracy of this solution. A known approach to improving accuracy for a discretised PDE, for any discretisation scheme in space, is to make the discretised grid finer by introducing even more nodal values. The intuition behind this is that as the spacing between nodal values approaches 0, we approach the true value of the unknown function. However, as the number of unknown variables increases, the size of the vectorised form increases drastically. For a problem with an **m-by-n** grid structure, the vectors **x** and **b** would store $\boldsymbol{m} \times \boldsymbol{n}$ elements, and the **A** matrix would store $\boldsymbol{m} \times \boldsymbol{n}$**-by-**$\boldsymbol{m} \times \boldsymbol{n}$ elements. This eventually leads to practically unfeasible requirements for storage and computation on a standard machine and high-performance computing or heavy parallelisation is required to solve problems of this scale.

These issues are noticeable even for moderately sized grids of around 100 elements per spatial axis.

Secondly, this representation is not intuitive of the original form of the equation itself. As we will see later in the paper when the Burgers' equation is presented, **Ax=b** does not represent that equation in its proper form. The issue with this is that direct or indirect solvers that attempt to solve this problem do so by considering the patterns and structures built into the matrix **A** or even the linear system as a whole, instead of dealing with the underlying structure of the different terms of the equation itself and trying to solve an equation that presents each such term separately. Moreover, the vectorisation approach requires devising a scheme of indexing the unknown variables to convert them from their natural 2-dimensional state to a 1-dimensional vector and maintaining this scheme throughout the solution.

This project aims to present an easily generalisable approach to solving PDEs without performing the vectorisation of unknown variables. This is done to present an alternative to the vectorisation scheme that could lead to further improvements in the process with future research, and also to deal with the aforementioned issues to some extent. As will be shown in Chapter 2, there are methods of doing this that directly convert the problem into a linear matrix equation of a known form, which can then be solved via well-established methods. However, to present a truly general method, the problem chosen is a nonlinear PDE of sufficient complexity to ensure that the algorithm developed for it can easily be used for other simpler PDEs as well. Moreover, each term of the original equation is given proper representation in the final discretised matrix equation. This is also done to address a research gap that exists in using matrix equation formulations for nonlinear PDEs.

1.3 Objectives

To accomplish the goals of this project, the following objectives must be completed:

1) Understand the formulation of Burger's equation and its utility in real-life flow models, and choose a 2d variant to solve.
2) Discuss different known approaches to discretising space variables, and choose a suitable approach. Apply this suitable discretisation approach to the space variables of the equation.
3) Discuss different strategies to discretise the system of ODEs with respect to time, utilising well-established methods of solving initial value problems. Identify and implement a suitable strategy to discretise the discretised space system with respect to time.
4) Formulate a suitable nonlinear matrix equation to represent the discretised system.

5) Observe the specific type of matrix equation that is representative of the system and the background literature related to standard numerical solvers that exist for matrix equations. Explain why these do not work for the unique form of the existing nonlinear equation.
6) Identify an approach to linearising the nonlinear matrix equation and present a modification that will work for the current problem.
7) Identify the sparsity patterns of the matrices involved in the equation to better optimise the solution algorithm.
8) Apply the chosen solver to solve the equation and obtain the solution for our grid matrix approximation.
9) Create and execute an algorithm that encompasses all of the aforementioned objectives, in MATLAB.
10) Present data visualisation of the computational aspects of the entire process, and provide evidence that this approach is tractable.

1.4 Deliverables

As this project aims to present exploratory software, there are primarily three deliverables:

1) A GitHub repository containing MATLAB code files that represent the system of equations, the methods chosen and applied for solving the system, and helper functions that both aid these methods and also visualise key aspects of the report.
2) Graphs, tables, and images to visualise and compare the performance of the chosen approaches with more traditional methods and provide proper test results, as well as demonstrate overall approaches and algorithms used in the project.
3) A detailed report to describe the context of the aforementioned data visualisation components, and clearly illustrate the result and conclusion of this study.

1.5 Notation and preliminary definitions

As per standard convention, the variable representing the unknown function in the Partial Differential Equation is denoted by $u(x, y, t)$. Throughout the paper, u is used to denote the unknown function $u(x, y, t)$ unless stated otherwise. Upon discretising the particular 2-dimensional region under study, at any grid point (x_j, y_i) the unknown variable is represented as $u_{i,j}$. Here, for a discrete step in x-the axis Δx, $x_j = x_0 + j\Delta x$ where x_0 is the first or leftmost value both on the x-axis and in the matrix storing the discrete x values. Similarly, for a discrete step in y-axis $\Delta y, y_i = y_0 + i\Delta y$ where y_0 is the first or bottommost value on the y-axis, but conversely topmost value in the matrix storing the discrete y values.

This is simply to account for the fact that in standard matrix notation, we count from the topmost row and go down while increasing index values. Upon further discretisation with respect to time, at a given time step n, the unknown variable is represented by $u_{i,j}^{n}$.

Throughout the paper, a matrix will be represented in bold lettering and uppercase. For example, a matrix denoted by the letter "A" will be written as **A**. A vector will be represented in bold lettering and lowercase, like **b**. Unless specifically mentioned, a matrix or a vector will be assumed to be of conforming dimensions to participate in the equation it exists in. It will also be assumed that the matrix or vector is real, i.e., consisting of elements that belong to the set of real number $\mathbb{R}$. Whenever a vector $\boldsymbol{e_i}$ is mentioned and no specific context is given, it represents the standard basis vector, i.e., the i^{th} column vector of the identity matrix **I** under standard addition and multiplication operations of two matrices. A matrix denoted by $\mathbf{A}_{\mathrm{m}\times\mathrm{n}}$ refers to a matrix with m rows and n columns for any field over which it exists, and the subscript will be removed when the context is clear. Similarly, a vector denoted by $\boldsymbol{v}_{n\times 1}$ and $\boldsymbol{v}_{1\times n}$ represents a column vector containing n rows and a row vector containing n columns respectively. Moreover, the greater value of the number of rows and columns of a matrix will be referred to as the dimension of the matrix. A matrix denoted by $\mathbf{A}_{\mathrm{n}}$ refers to a square matrix of n rows and n columns. An exception to this format of displaying a matrix and information about its number of rows and columns is made for the case of matrix equations. For all such equations, and unless otherwise specified, it will be assumed that all the matrices involved in the equation are of conforming dimensions for the operation they are a part of. For any matrix **A**, the notations $\mathbf{A}^{\mathbf{T}}, \mathbf{A}^{*}$ denote the transpose and conjugate transpose of the matrix respectively.

1.5.1 Vector and Matrix Norms

For a vector $\boldsymbol{v} \in \mathbb{C}^{n}$, $\parallel \boldsymbol{v} \parallel_{\mathbf{2}}$ represents the Euclidean or ℓ_2 norm. If each element of the vector is represented by v_k, $\parallel \boldsymbol{v} \parallel_{\mathbf{2}} = \sqrt{\Sigma |v_k|^2}$, where $|v_k|$ is the absolute magnitude of the element. Its induced matrix norm, represented for a matrix **A** $\in \mathbb{C}^{m\times n}$ by $\parallel \boldsymbol{A}_{m\times n} \parallel_{\mathbf{2}}$, is the spectral norm. For a symmetric matrix, the spectral norm is defined as the spectral radius or the eigenvalue of the matrix with the largest absolute magnitude. For asymmetric matrices, $\parallel \mathbf{A} \parallel_2 = \sqrt{\parallel \mathbf{A}^{\mathbf{T}}\mathbf{A} \parallel_2}$, where $\mathbf{A}^{\mathbf{T}}\mathbf{A}$ is symmetric for any matrix **A**.

The Frobenius norm of a matrix $\mathbf{A}_{\mathbf{m}\times\mathbf{n}}$, $\parallel \mathbf{A} \parallel_F = \sqrt{\Sigma_{i=1,j=1}^{m,n} |a_{i,j}|^2}$, where $a_{i,j}$ represents the element at the i^{th} row and j^{th} column of the matrix.

1.5.2 Operators

Throughout the paper, the symbol '×' will be used to denote standard matrix-matrix product, matrix-vector product, scalar-matrix product, scalar-vector product and scalar-scalar product. The context will make clear which two structures are involved in the multiplication.

The symbol '$\odot$' will be used to refer to the Hadamard product between two matrices. The Hadamard product is simply the element-wise multiplication between two matrices of equal dimensions. The Hadamard product is commutative, associative and distributive concerning the standard addition of two matrices. It is commutative, but not associative or distributive with respect to standard multiplication of two matrices. An important observation that is noted for the Hadamard product is that if we multiply the Hadamard product of two matrices **A** and **B** by a standard basis vector $\boldsymbol{e_i}$, it distributes over the product like follows:

$$(\mathbf{A} \odot \mathbf{B}) \times \boldsymbol{e_i} = (\mathbf{A} \times \boldsymbol{e_i}) \odot (\mathbf{B} \times \boldsymbol{e_i})$$

This is a property that applies only to the standard basis vectors and the zero vectors. This occurs because multiplying a matrix by a standard basis vector is equivalent to choosing a particular column of the matrix based on the position of 1 in the vector. As Hadamard products are simply element-wise products between two matrices, choosing a column vector of the resulting matrix is equivalent to choosing a column vector at the same position for the individual matrices and computing their Hadamard product. If this is done for all the standard basis vectors and we place the resultant vectors side-by-side, we will get the original Hadamard product matrix. Additionally, the Hadamard product of two vectors is the standard matrix-vector multiplication of the diagonal matrix of one vector with the other vector.

The symbol '$\otimes$' will be used to refer to the Kronecker product of two matrices. For two matrices **A** $\in \mathbb{C}^{m_A \times n_A}$ and **B** $\in \mathbb{C}^{m_B \times n_B}$, the Kronecker product can be represented as:

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{1,1} \times \boldsymbol{B} & a_{1,2} \times \boldsymbol{B} & \cdots & a_{1,n_A} \times \boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m_A,1} \times \boldsymbol{B} & a_{m_A,2} \times \boldsymbol{B} & \cdots & a_{m_A,n_A} \times \boldsymbol{B} \end{bmatrix}$$

Where $a_{i,j}$ represents the element at the ith row and jth column of the matrix **A**. This results in a larger matrix of dimensions $m_A \times m_B\ by\ n_A \times n_B$.

The operator vec($\boldsymbol{A}_{m\times n}$) refers to the operation of converting a matrix into a column vector by simply stacking the columns one below the other, to obtain a vector **a** of size $(m \times n)$ by 1. The operator diag($\boldsymbol{A}_{m\times n}$) refers to creating a strictly diagonal matrix by taking the diagonal elements of the matrix **A**. The operator diag($\boldsymbol{a}_{m\times 1}$) or diag($\boldsymbol{a}_{1\times m}$) refers to creating a strictly diagonal matrix by taking all the elements of the vector **a** and placing them along the diagonal of the new matrix.

1.6 Ethical, legal, and social issues

There are no known ethical, legal or social issues associated with this project as it aims to simply obtain a different approach to solving a mathematical equation, and is thus primarily of academic interest.

Chapter 2
Background Research

2.1 Literature Review

2.1.1 Burgers' equation

First formulated by Harry Bateman as a study in fluid dynamics, the equation resembled an approach to prove a solution existed for equations that sought to tackle the D'Alembert Paradox (Bateman, 1915). It was later extensively derived, formulated and studied by J.M. Burgers in his seminal paper (Burgers, 1948). The original equation depended on the x-axis in space and time, and is currently known as the 1-dimensional Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

Where u is the unknown function defining the behaviour of the entity under study and it depends on the independent variables x and t, and ν is the viscosity coefficient. It is used primarily as a mathematical model to represent more complex partial differential equations that exist in the theory of turbulence. It was also extended to model heat flow via the Cole-Hopf transformation (Cole, 1951), which also provided an approach to linearising the equation. In recent years, especially with increasingly more powerful computing hardware, Burgers' equation has been utilised greatly in Computational Fluid Dynamics. It is also of great interest mathematically as it resembles one of the few hyperbolic-parabolic PDEs (Schiesser, 1991) with a well-known solution. This allows it to be used as an equation for simulation in multiple applied mathematics fields (Bonkile et al, 2018). A great amount of research has been done in solving the Burgers' Equation. As the equation itself exists in different forms based on the axes of spatial freedom provided to the physical entity being simulated, varying approaches have tackled different forms of it. In the 2-dimensional form, it exists as a pair of equations for two unknowns, u and v(Jain and Holla,1978):

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

Where u and v are usually implied to represent the velocity of flow of an entity in the x and y-direction respectively. As required of this paper, the focus is given towards a variant of the coupled (2+1) - dimensional Burgers' equations that results from equal velocity in both x and y-directions. While this equation is not used widely and is purely of mathematical interest it removes the complexity gained from solving a system of PDEs, by converting the coupled equations into a single equation. As the focus of this project is on dealing simply with the complexity of nonlinear PDEs, and any solution can be extended to the coupled variant, this equation is chosen. The equation is discussed to some extent in (Liu et al, 1994) and (Mohamed, 2019), and exists in the form:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + u\frac{\partial u}{\partial y} = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Most of the known literature deals with tailored approaches to solving the Burgers' equation and the standard Method of Lines is usually not adopted in the wake of these more well-suited algorithms. However, as the aim of this project is not to specifically solve the Burgers' equation but to present a fairly simple and general approach to solving any PDE of a similar level of complexity, the (Fletcher, 1982) paper is chosen as a reference for the schemes used throughout this paper. While it deals with the more complex coupled 2-dimensional equations, it still provides a detailed comparative analysis of the pros and cons of using different discretisation schemes as part of the Method of Lines approach.

2.1.2 Linear Matrix Equations

As mentioned in Chapter 1, the approach to solving discretised linear PDEs using matrix formulations exists in the form of solutions to linear matrix equations. The (sorForSylvester paper) paper describes in detail how these equations can be used for discretised linear PDEs while also demonstrating how algorithms typically used for linear system of equations can be adopted for linear matrix equations as well. This is an important observation as it provides an idea that methods that are well established in one domain can be extended with some modification to a more generalised domain. As will be demonstrated in chapters 3 and 4, this is of immense help in overcoming the challenges encountered by the nonlinearity of the chosen equation in this paper.

There is, in general, an abundance of research in the field of efficiently obtaining exact solutions as well as numerical approximations of linear matrix equations. The research is equally focused on well-formed solutions to these equations and studying properties of stability in the matrices. The most common form of the linear matrix equation is the generalised equation:

$$\mathbf{AXE} + \mathbf{DXB} = \mathbf{C}$$

whereby standard notations **A**, **E**, **D**, **B**, and **C** are coefficient matrices existing in the complex space and **X** is a matrix of the unknowns to be obtained. Under certain conditions, this equation can be simplified further into one of two forms. Much of the research, solutions, and approach strategies regarding these two forms are described in (Simoncini, 2016).

Based on the condition that **E** = **B** = **I** the equation takes the form of the Sylvester equation, based on (Sylvester, 1884). If in addition to this condition, **B** = **A*** then the equation becomes the Lyapunov equation, based on (Lyapunov, 1992). It should be noted that while linear matrix equations are themselves a subject of study, both Sylvester and Lyapunov's equations arise from dealing with a differential equation problem. The Lyapunov equation itself has been of immense importance due to its original equation representing properties of motion in the form of stability and instability. It has thus been used extensively in control theory. However, in terms of PDEs Lyapunov equations are relegated to fairly simple linear equations because slight changes in different terms in a PDE will result in very different coefficient matrices **A** and **B** in the original linear matrix equation. This is also witnessed for a fairly simple PDE in (sorForSylvester paper) where even a separate constant value results in **B** ≠ **A***. Thus for this project, the focus will be given to continuous time Sylvester linear matrix equations.

2.1.2.1 Sylvester Equation

The Sylvester equation is represented by $\mathbf{AX} + \mathbf{XB} = \mathbf{C}$, and its Kronecker form is subsequently $\boldsymbol{Ax} = \boldsymbol{c}$, where $\boldsymbol{A} = \mathbf{I} \otimes \mathbf{A} + \mathbf{B}^{\mathbf{T}} \otimes \mathbf{I}$, $\boldsymbol{x} = vec(\mathbf{X})$, $\boldsymbol{c} = vec(\mathbf{C})$. The issues and observations regarding the stability of the systems the Sylvester equation represents are dependent on the stability of the solution matrix, which in turn is observed by (Simoncini,2016) to revolve around the distribution of singular values. This is also important in the development of iterative solutions that improve on an initial guess, and their convergence analysis. It should be noted that bounds on norms of the solution matrix and the backward error, as well as further stability analysis, are limited to the coefficient matrices **A** and **B** being symmetric. Not much concrete literature is observed on nonsymmetric **A** and **B**, and according to (Simoncini,2016) is still an open area of research. Certain closed-form solutions have been derived for the Sylvester equation, but are highly inefficient and unstable for solving the equations. As such, they have acted as stepping stones for other more powerful algorithms to be built upon. The Sylvester equation exists in three distinguishable forms, from the perspective of computational and storage restraints on machines.

Small-scale Sylvester equations, which typically have coefficient matrices with dimensions existing below 1000, have different algorithms to approximate solutions. The most common algorithm of these is the Bartels-Stewart Algorithm (Bartels and Stewart, 1972) which performs a Schur Decomposition followed by a forward-backwards substitution step which solves a triangular variable matrix obtained from the decomposition form. This algorithm is the basis of the *sylvester* script utilised by Matlab but is computationally expensive in some scenarios. Multiple improvements have been devised, for example replacing the Schur decomposition with a Hessenberg decomposition which has a significantly lower cost (Golub et al., 1979). Another approach involves ignoring the requirement of a Kronecker formulation for the calculation of the elements of **X** by utilising one of three column-wise block solvers for the equation (Sorensen and Zhou, 2003). This second approach is especially vital when the coefficient matrices have complex conjugate eigenvalues and the default Bartels-Stewart algorithm relies on the equivalence between the 2x2 Sylvester equations with the corresponding 4x1 linear system obtained from the Kronecker form. Iterative algorithms have also been formulated that compute iteratively better **X** matrices from an initial guess and are developed analogously to how Newton's iteration method works in solving a linear system of equations. Numerical methods based on closed forms also exist but are mostly experimental in nature as a proper analysis of the stability of such measures is not provided (Hu and Cheng, 2006).

For Sylvester equations where one of the coefficient matrices(either **A** or **B**) is very large, adopting the same solvers from the previous step while ensuring there is not a storage overflow is the main challenge. One approach to solving this is to perform a spectral decomposition on the smaller coefficient matrix and compute each column of the solution matrix via a linear shift to the large matrix. This is well described in an algorithmic form in (Simoncini,2016). The eigenvalues of the smaller matrix act as the scalar shifts in this scenario. Solving this linear system is computationally less costly and highly parallelisable. An alternative in case spectral decomposition is not possible or computationally very expensive, is performing a Schur decomposition (El Guennouni et al., 2002) and forming a sequential equation to solve instead where the shifts are the invariant factors. This is somewhat analogous to the approach utilised in (Bartels and Stewart,1972), in that we utilise the solution obtained from a previous step to calculate the solution of the next step. Another highly utilised approach is via Projection methods which give rise to low-rank approximations that can be relatively easy to solve and analyse and also project the solution matrix onto a well-defined space of our choice. The entire projection method results in a smaller-sized Sylvester equation needing to be solved, which can be done by methods available for small-scale Sylvester equations. An interesting property of this approach is that the Sylvester operator performs the same shift operation as part of the residual calculation (Simoncini, 2016), as was occurring for the spectral decomposition solver. However, now the shifts are not scalars(eigenvalues) or invariant factors, but a matrix. For projection methods and the following types of Sylvester equations, the base assumption is that the coefficient matrix **C** is a large sparse matrix of relatively low rank. Despite this usually being the case in any solution that directly formulates such an equation it is not possible to do so in the approaches chosen in this paper. While this is explained in much greater detail in the following chapters, simply put it is because we cannot directly formulate a linear matrix equation(Sylvester or otherwise) from an inherently nonlinear structure.

For Sylvester equations where both the coefficient matrices **A** and **B** are large, three types of approaches are followed, namely projection methods, matrix update schemes like the Alternating Direction Implicit iterations(Simoncini,2016), and sparse data format recurrences. An interesting and important observation is that normal preconditioning strategies do not work to make the Sylvester equation better formed, as they inadvertently destroy the symmetry properties of the problem itself. These can however be employed if we instead try to solve the Kronecker form of the equation, which requires heavy computation when **A** and **B** are both large. They also destroy the Kronecker structure of the matrix equation which is the main reason why computational strategies can exist that have costs in the order of the dimensionality of either **A** or **B** but not the product of their dimensions. For projection methods, Krylov subspace is usually used as the projection space for its ability to be utilised in exponential integral solutions (Saad, 1989). The issue faced with projection methods, in this case, is that the transformation matrices for the chosen subspaces for **A** and **B** are dense and need to be stored. Thus the subspaces cannot be very large. This has been somewhat resolved in recent years with research into enriched spaces. With regards to the second approach, the use of ADI iterations to solve Sylvester equations was represented first in (Ellner and Wachspress, 1986). Current versions of the ADI iterations have been one of the leading approaches to solving both Sylvester and Lyapunov equations, with the major focus being the computation of optimal or near-optimal shift parameters in order to solve accurate iterations of shifted linear systems for the solution matrix. Regarding the final approach, various data sparse techniques have also been researched for solving Sylvester and Lyapunov equations, with one dealing with an expanded multigrid algorithm for coefficient matrices derived from the discretisation of partial differential equations (Grasedyck and Hackbusch, 2007). Iterate data sparse methods are discussed in (Baur, 2008) and utilise the H-matrix for low-rank approximations effectively.

2.1.2.2 Prerequisite to applying Linear Matrix Equations

These methods and the literature associated with them demonstrate that if a dynamical system can be represented in the form of a well-defined linear matrix equation like the Sylvester matrix equation, there are multiple thoroughly researched methods to solving these equations efficiently. Moreover, software libraries like SILICOT (insert ref here) and Lyapack (for Lyapunov equations) (insert ref here) utilise these algorithms to provide computational efficiency to users. However, to apply these methods to a nonlinear matrix equation as the one obtained in this paper, a crucial step is to first linearise the equation.

Linearisation of a nonlinear matrix equation requires manipulation and extension of a few different ideas and approaches.

2.2 Methods and Techniques

2.2.1 Linearisation of Nonlinear Equations

There are certain well-known iterative approaches to numerically solving nonlinear equations and, by extension, systems of nonlinear equations. One of the most common methods is the Newton-Raphson method (Applied Numerical Analysis book, Gerald Wheatley), which is both simple and highly convergent given the right conditions, but is not a very robust algorithm. At a given iteration n for the known approximation x_n to the unknown variable x and a function $F(x) = 0$, it computes $x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)}$. Here, $x_{n+1} - x_n = \delta$ and as $F(x_n) \to 0$, $x_{n+1} \to x_n$,$\delta \to 0$ and we attain convergence. Thus, in essence it is a root finding algorithm. The Newton-Raphson approach requires a good initial guess to converge to a solution, and works very well with Initial Boundary Value Problems (IBVPs) as such problems utilise domain knowledge to provide a good initial guess at each iteration. Another requirement is obtaining the derivative of the equation at each iteration. For systems of nonlinear equations, this requires computing the derivative of each equation with respect to all the unknown variables involved in the system and is a computationally expensive process. The matrix obtained from this computation is called the Jacobian matrix ($\mathbf{J}$) (Numerical Optimisation, Jorge, Stephen, Chapter 10) and the resulting system to be solved is a system of linear equations. For a system consisting of n equations and n unknowns represented by $x^i, i \in 1,2, \dots, n$, this is denoted by $\mathbf{J_{n\times n}\delta_{n\times 1}} = -\mathbf{F}(\mathbf{X})_{\boldsymbol{n}\times\mathbf{1}}$ where $\boldsymbol{\delta}$ is a vector of all δ^i values across the system and $\mathbf{F}(\mathbf{X})$ is a vector of the function values for each equation with respect to the current iterate values of the unknowns. As is observed, the system is linear with respect to $\boldsymbol{\delta}$ and focus is given to efficiently solving this linear system by studying and exploiting underlying structures in the Jacobian matrix. However, as will be shown in detail in Chapter 3, for matrix systems we require a more efficient approach to computing the derivative.

2.2.1.1 Fréchet derivative

The Fréchet derivative exists as a generalisation of the standard derivative to vector-valued functions over most standard vector spaces as long as they have a defined norm attached to them. This is done by representing the derivative as a linear mapping operation from one vector space to another (Doodoon reference-Obsidian). This allows the Fréchet derivative to be generalisable to any function as long as the space of the function is a normed vector space, which is true for the space over which coefficients, elements, vector and matrix norms are defined for the PDE chosen in this paper. For a function $f(x)$ on a variable x

which maps it from a normed vector space U to a normed vector space W, there exists a Fréchet derivative A if $\lim_{\|h\|\to 0} \frac{\|f(x+h)-f(x)-Ah\|_w}{\|h\|_v} = 0$, where $\|.\|_W$ and $\|.\|_V$ denote the norms defined over vector spaces W and V respectively and V ⊆ U (Coleman R paper). The h represents an infinitesimally small perturbation in the value of variable x, similar to the h defined in standard derivative. However, in this case all the terms can be generalised to vector-valued functions and matrix-valued functions, and in particular functions that map a matrix to another matrix.

2.2.1.2 Extension to Nonlinear Matrix Equations

With some modification, the Fréchet derivative can be extended to obtain a "derivative" for a nonlinear matrix equation. While the actual derivation is described in detail in chapters 3 and 4, the idea was thoroughly explored while computing the square root of a matrix in (Higham's paper) via the Newton's method. There, the nonlinear equation was $\mathbf{X}^2 - \mathbf{A} = \mathbf{0}$, where **A** represented the matrix whose square root had to be computed. A simple yet effective approach was implemented wherein the expansion $\mathrm{F}(\mathbf{X}+\mathbf{H}) = \mathbf{X}^2 - \mathbf{A} + \mathbf{XH} + \mathbf{HX} + \mathbf{H}^2$ was equated to the Taylor series [[Taylor]] expansion $\mathrm{F}(\mathbf{X}+\mathbf{H}) = F(\mathbf{X}) + F'(\mathbf{X}).\mathbf{H} + F''(\mathbf{X}).\mathbf{H}^2$. This trivialises the process of computing the derivative by representing it as a linear matrix equation and simply solving this linear matrix equation. Then the $F'(\mathbf{X}).\mathbf{H}$ can considered to be a generalisation of $\mathbf{J\delta}$ and simply equating it to -F(**X**) gives us the equation to solve at each iteration of the Newton method. For the particular problem of computing square root other approaches were deemed much more efficient and thus modifications were made to the Newton iteration to have comparable efficiency in computation. These modifications led the paper into a much more detailed analysis on which approaches were numerically stable and guaranteed convergence for each iteration. For the purpose of this paper, it is evident that the derivative itself represents a linear matrix equation form that can be solved with aforementioned known methods. As will be discussed in Chapter 3, the derivative of the equation studied for this paper cannot simply be represented as a linear matrix equation of known form. Therefore, we require adoption and extension of a well-known approach to solving linear system of equations iteratively.

2.2.1.3 Iterative methods for solving Linear systems

Similar to iterative approaches for solving nonlinear system of equations, iterative methods exist for solving large linear system of equations where direct solvers become computationally expensive to use. These methods are well-known and utilise properties of the matrix **A** in the linear system **Ax** = **b** to solve the system iteratively to obtain an accurate approximation of the solution **x**. They are broadly split into Stationary Iterative methods and methods based on approximations in the Krylov subspace (Krylov subspace reference). For the purpose of this paper, focus is given towards the simpler Stationary Iterative methods, but it should be noted that the algorithm created in this paper can adopt Krylov subspace methods with reasonable adjustments. Stationary approaches like the Jacobi, Gauss-Seidel, Richardson and Relaxation-like methods (insert ref here) all depend on splitting the matrix **A** into constituent components which then allow us to make a convenient split in the system itself. Thus, if **A** = **D** + **E** is the split, we can simply take part of the split to the right-hand side which contains the known vector **b**. This modified equation then represents the **x** on the right-hand side as a known vector and is used to obtain the unknown value of **x** on the left during the current iteration. The benefit of doing this split is that, by convention the matrix **D**, is easier to invert and thus the new modified linear system to be solved becomes a much more computationally viable process than inverting the matrix **A** in its entirety. The different stationary methods then simply replace the matrix **D** with certain values or simple matrices that adhere to certain properties on **A**. However, it is the general idea of splitting that we will extend to our matrix equation, as will be shown in detail in chapters 3 and 4.

2.2.2 Methods for discretisation in space

To obtain a nonlinear matrix equation we require knowledge of methods to discretise the PDE from its original form. For any PDE, the standard numerical approach involves discretising the equation with respect to different independent, continuous variables and obtaining a system of non-linear equations that are then solved by standard non-linear solvers. This approach is also known as the Method of Lines, or more specifically for numerical approaches, the Numerical Method of Lines [[Schiesser Book]]. For discretisation, three different methods are available based on the scenarios in which the continuous variables can exist and the computing power available for solving the problem. For all three approaches, there is a requirement for boundary conditions, which is why they work best with Boundary Value Problems (BVPs). These methods are namely, the Finite Difference Methods (FDMs), the Finite Element Methods (FEMs) and Finite Volume Methods (FVMs). As shown in (Fletcher,1982) for 2-dimensional PDEs the FDMs are much more efficient than FEMs while also being similar or even better in terms of accuracy for the Burgers' equations. Thus, for this paper focus is given to the Finite Difference Methods. It is however important to note FEMs [[Applied Numerical Analysis]], and the more computationally heavy FVMs are almost necessary methods when the bounded region being studied is irregular and has mixed boundary conditions. Also, FDM approaches can easily be replaced with FEM or FVM approaches and thus the algorithm being presented in this paper is not limited to a certain discretisation approach.

A finite difference scheme was introduced in the Courant-Friedrichs-Lewy paper [[CFL Paper]] and was the first proper approach to discretising a Partial Differential Equation with respect to its continuous variables into discrete forms that existed within a boundary. It was a viable method for both time-independent as well as time-dependent PDEs, and as such has seen extensive use in computation problems, especially Computational Fluid Dynamics. It also provided an important result of convergence that is now assumed as a standard result for all discretisation schemes. This result states that the approximate solution converges to the analytical solution of the equation being discretised as the step size, or the distance between two adjacent nodes, converges to 0. It is also computationally the least expensive method, as it simply breaks a bounded region into a grid or mesh of different points. As presented in [[S. Chakravarty]] there are three different applicable schemes, namely the forward, backward and central difference schemes. The forward scheme deals with the difference between the currently observed node and the node at the next (or forward) step. The backward scheme deals with the difference between the currently observed node and the node at the previous (or backward) step. These approaches are also called downwind and upwind schemes respectively as they signify the direction of flow of an entity in how they calculate the difference. They thus provide a very effective representation of convective flow terms in a PDE (Leveque,Numerical conservation). The central difference deals with the difference between nodes at backward and forward steps. The error analysis for this method was first researched in [[Gershgorin]] and has been thoroughly analysed since. The difference schemes are obtained from expanding Taylor's series [[Taylor]] and truncating terms beyond the required derivative term. This introduces a truncation error, that has been proven to have an upper bound on the chosen step size raised to the power of the order of the difference scheme. Here, the order of a difference scheme refers to the size of the stencil used for the approximation, or the number of points involved in calculating the approximation of a nth order derivative at a particular grid point. As such, much more accurate results are obtained by utilising higher-order stencils. As provided briefly in a table in [[Fletcher Coupled]], the schemes are based and named on the number of points used to approximate a derivative. However, it only represents these schemes for a central difference method, and it should be noted that one-sided point-based higher-order schemes also exist. However, for higher-order stencils, the computation becomes more complex as we have to account for missing values for points that exist outside the bounded region, as well as solve a larger system of equations at each time step. There have been approximations for points existing outside the bounded region [[Hicks]] that work for homogeneous boundary conditions.

2.2.3 Methods for discretisation in time

After performing the Method of Lines discretisation on any time-dependent PDE, we obtain a system of Ordinary Differential Equations (ODEs) that depend only on the derivatives of the independent time variable, t. There exist broadly three separate approaches to discretising the continuous time ODE. All of these methods have a step difference approach that is somewhat similar to Finite Difference Schemes. However, these families of methods follow either a multi-step or multi-stage scheme to obtain a discretisation for the Initial Value Problem of continuous time-dependent ODEs. Each family in one form or the other generalises methods that existed previously for initial value problems, namely the implicit and explicit forms of the Euler formulas, called Backward and Forward Euler respectively, and the overarching Θ - method that performs linear combinations of the implicit and explicit Euler methods. This Θ approach gives rise to an equation, at a particular time step k and time increment Δt, of the form $\frac{U^{k+1}-U^{k}}{\Delta t} = \theta F\left(U^{k+1}\right) + (1-\theta)F\left(U^{k}\right)$, where U^{k+1} is the value of unknowns to be determined at the next time step, U^{k} is the known value of unknowns at the current time step, and $F\left(U^{k+1}\right)$ and $F\left(U^{k}\right)$ are unknown and known values of the system of equations of the unknowns at the next and current time step respectively. A particular variant of the Θ-method that is utilised as a scheme in this paper is the Crank-Nicholson method (insert ref) when $\Theta = 0.5$. Out of the aforementioned three methods, the Adams family of methods is ignored as the paper focuses on approaches that lie in the other two family of methods.

The first family of methods is called the Runge-Kutta (RK) family of methods. It comprises both explicit and implicit methods. The classical methods provided by C. Runge are of explicit nature, and sought to solve an ODE of the form $\mathrm{y}' = \mathrm{f}\left(\mathrm{t}, \mathrm{y}(\mathrm{t})\right)$ where y is the unknown variable to be solved that depends only on the independent time variable t. However, further improvements to his original paper [[Runge]] by Wilhelm Kutta [[Kutta]] and later Evert Johannes Nyström laid the groundwork for implicit methods to be utilised as well. The RK family of methods are single-step, multi-stage methods that calculate the value of the unknown variable at a time-step $n+1$ based on the knowledge of the value of the unknown variable at time-step n only. However, they do this by calculating the value of the $\mathrm{f}\left(\mathrm{t}, \mathrm{y}(\mathrm{t})\right)$ multiple times for incremental values of y and t. This repeated calculation is said to occur in stages per time step, which is why they are called multi-stage methods. The difference between the implicit and explicit suite of methods in this family comes from the dependence of a particular stage's unknown variable increment on the values of the variable calculated in the next stages. If there is no dependence and every stage depends only on stages that occur prior to it, the methods are explicit in nature and can be solved by forward substitution methods. However, if every stage depends on values in all the other stages, then the methods are implicit in nature and require solving a system of non-linear equations of size s, where s is the number of stages, which was discovered by Jean Kuntzman in [[Kuntzman]]. This quickly becomes computationally expensive, especially when combined with the method of lines approach to solving an entire PDE system. However, the classic fully implicit RK schemes, that are modelled on the Gaussian quadrature formulae, all possess absolute stability, or are A-stable. This is not the case for explicit RK schemes however, as it is observed that no explicit RK scheme is A-stable, and thus implicit schemes are preferred for problems where artificial oscillations could occur due to instability. For either suite of methods, the accuracy depends on both the local truncation error that occurs from obtaining a finite approximation of the infinite expansion of the Taylor's series of the unknown variable, and the overall global error that occurs from performing such an approximation process for s stages. As detailed in [[Scientific Computing]], the local truncation error always has to be of an order higher than the required order of global error by a value of 1. There is however, no known correlation between the required order of global error of the equation and the number of stages involved in the process, which makes finding higher order RK methods a challenging task. As described in [[History Of RK]], the minimum number of stages required for methods up to the 4th order are equivalent to the required order itself. For the 5th and 6th order, the minimum number of stages is greater than the required order by 1. This relation between the stages varies as the required order increases, and beyond 8th order RK methods, not much is known about the minimum number of stages required.

The other family of methods is called the Backward Differentiation Formula (BDF). They are used especially for solving stiff differential equations, i.e., equations or systems that require extremely small step sizes in regions where the solution is relatively straight and flat. These types of equations require a special type of approach as standard solvers will apply large step sizes in such regions to avoid stagnating the solution. This family of methods comprises of only implicit methods, and are multistep in nature. They were first introduced in [[Curtiss]], and later generalised into a family of methods via the formula provided in [[Gear]]. The approach of these methods is to use m previous solutions of the unknown variable $y(t)$ to construct a Lagrange interpolating polynomial $p_m(t)$ of degree m, such that

$p_m(t_{n-i+1}) = y(t_{n-i+1}), i = 0, \dots, m$, where n represents the current time-step value, and the approximation $p'_m(t_n + h) = f\left(t_{n+1}, y(t_{n+1})\right)$ is utilised to obtain a solution at the unknown point $y(t_{n+1})$. This is provided in the general formula $\Sigma_{k=0}^{m} a_k y_{n-k+1} = h\beta f\left(t_{n+1}, y(t_{n+1})\right)$ where a_k and β are coefficients chosen such that the method obtains the required order of accuracy. Since for each time-step we need to calculate the functional value at the unknown $y(t_{n+1})$ the methods are implicit in nature. It is important to note that while the BDF methods require less computation than the fully implicit RK schemes, they require certain extra memory considerations to store the results of the unknown variables in the previous m time steps. The stability of the methods is dominated by the absolute stability, or A-stability. It has been observed that BDF - 1 and BDF - 2 methods are completely A-stable, and BDF- 3 and higher are almost A-stable except certain small regions of instability. However, all BDF methods of order 6 and above are unstable and are not recommended for use except for specially handcrafted equations. For BDF methods the order of the global error for a given m-step BDF method is also m with the local truncation error being of an order greater than 1.

2.3 Choice of Methods

While the reasoning behind the choices will be elaborated upon when discussing the methodology in Chapter 3, the methods chosen to solve the PDE are as follows:

1) The 3-point and alternatively, combined 3-point and 5-point central finite difference schemes are adopted for discretisation in space.

2) The implicit Euler (BDF-1) approach and alternatively Crank-Nicolson method are adopted for the discretisation in time.

3) Newton’s approach is utilised for solving the nonlinear matrix equation.

4) The Fréchet derivative is used to compute the derivative of the equation at each iteration of the Newton’s method.

5) The basic idea behind stationary iterative methods used to solve linear system of equations is utilised to solve the linear matrix equation obtained from computing the Fréchet derivative. This will lead to different solvers which will address the research gap stated in Chapter 1.

Chapter 3
Software Requirements and Research Methodology

This chapter presents the research question, the approach to answering that question and the design and requirements of the software that represents the algorithm answering the question in a practical codebase. This acts as an understanding of what the gap in research as mentioned in Chapter 1 is, and how this paper plans to address it while also paving the way for future research or improvements. As such, the first two sections will be dedicated purely to the research question and the methodology behind answering it, and the third section will discuss in brief the approach taken to programming the theoretical algorithm.

3.1 Research Question

The main question posed by this paper is "Can nonlinear Partial Differential Equations of sufficient complexity be solved by converting them into, and solving the resulting nonlinear matrix equations?" As shown in Chapter 2, a lot of research exists around linear PDEs and linear matrix equations and how to solve them, but besides the well-known Riccati equation (riccati ref here) and certain simplifications of it, not much research exists for nonlinear matrix equations. Additionally, an interesting issue that arises as the project progresses is the presence of nonlinear matrix equations involving separate product operators. To the extent of the researcher's knowledge, attempts made to solve such equations involve either complete vectorisation of the equation leading to the Kronecker form or approaches that utilise a particular way the equation is formed. Moreover, there is little documentation cementing these approaches as standard and as a whole scant discussion on the topic of mixed operator equations. Thus, major focus in the paper is given to efficiently solving this form of the matrix equation.

3.2 Research Methodology

The 2d Burgers' equation exists over a spatial domain $(x, y) \in \Omega$ with a well-defined boundary $\partial\Omega$. The domain studied in this paper is regular and square in shape. It can be represented as $\Omega = \{(x, y): 0 \leq x \leq 1, 0 \leq y \leq 1\}$. The equation is thus:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + u\frac{\partial u}{\partial y} = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right); (x, y) \in \Omega, \mathrm{t} > 0 \qquad -(i)$$

$$u = f(x, y, t); (x, y) \in \partial\Omega, \mathrm{t} > 0 \qquad -(i - a)$$

Here $f(x, y, t)$ denotes an arbitrary function that depends solely on the independent variables $x, y, \text{and } t$. Equation $(i - a)$ represents conditions that the system follows at the boundary of the region. While there are different types of boundary conditions that are used, for this equation we deal with Dirichlet boundary conditions (Dirichlet ref here), which directly define the value of u at the boundary. It should be noted that while we implement functions that look different for different parts of the boundary, they are essentially the same function but at different extremes of the x and y values. This will become clear in Chapter 4 when we introduce the actual boundary conditions. The equation is also subject to the initial conditions of:

$$u(x,y,0) = g(x,y); (x,y) \in \Omega, \mathrm{t} = 0 \qquad - (i-b)$$

Where $g(x,y)$ is an arbitrary function that depends only on the space variables x and y. The region can then be represented as below:

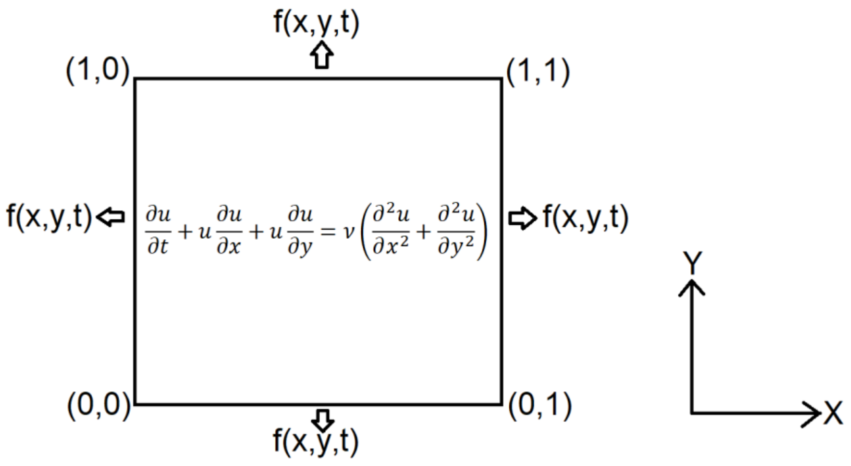
f(x,y,t)
(1,0)
(1,1)
f(x,y,t)
$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + u\frac{\partial u}{\partial y} = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$
f(x,y,t)
Y
(0,0)
(0,1)
X
f(x,y,t)

Figure 3.1: Domain studied for 2D Burgers' equation, at t>0

The first step is to choose and utilise any standard discretisation scheme to convert (i) into a matrix equation that properly reflects the different parts of the equation. This formulation should adhere to conditions present in $(i-a)$ and $(i-b)$ as well. The relatively simple Finite Difference Schemes are chosen as they are highly efficient and accurate in regular regions.

An important consideration that has to be taken into account is whether to solve the equation (i) in its natural form as presented above, or as the conservative form:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial (u^2)}{\partial x} + \frac{1}{2}\frac{\partial (u^2)}{\partial y} = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - (ii)$$

Solving (ii) is crucial in regions that contain shock waves, i.e., sudden change in flow velocities or other fluid properties. These lead to discontinuous weak solutions to nonlinear PDEs which finite difference schemes fail to converge to (Leveque, Numerical Conservation). While a lot of research exists that explains and proves this phenomenon, involving theories from both Physics and Computational Fluid Dynamics, a simple example can be shown in terms of the 1D Burgers' equation which depends on the x-axis in space. At the point where $x = 0$, a shock front exists where, for any value of x below 0 the velocity u is

1 and for any value of x greater than 0 it is 0. This indicates that u has two clearly separate solutions to the left and right of $x = 0$ that are invariant with the grid position (or value of x) in these regions. Thus, varying the grid size will not make the numerical method converge to a weak solution. This has the added issue of providing misleading results because applying boundary conditions will make the final result look accurate while not being so. The conservative form, (ii) of the equation (i) solves these issues and allows standard difference schemes to be applied to obtain a solution that is accurate and convergent to the analytical solution. As the region studied in this paper is to the right of the shock front it does not require the Burgers' equation to be represented in its conservative form. However, to address this issue and to present the simplicity of adopting methods used for the standard form, the conservative equation is also solved in this paper. Another issue in the discretisation of space variables of the Burgers' equation arises when dealing with convective terms. Here a decision has to be made about using the more accurate central difference schemes as they fail to capture the convective flow being represented by such terms and this may cause artificial oscillations to occur. In the case of the viscous Burgers' equation denoted by (i) the two opposing terms are the diffusion terms, $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$, and the convective terms $u\frac{\partial u}{\partial x}$ and $u\frac{\partial u}{\partial y}$. The coefficient of viscosity, ν, which is the inverse of the Reynold's number Re, decides which term dominates in the equation. For large values of Reynold's number, or as the system transitions from laminar flow to turbulent flow, the convective terms dominate as ν becomes increasingly smaller. In such cases, it is much more necessary to use the less accurate upwind schemes as they properly model convection. This penalty obtained in terms of accuracy can be mitigated to some extent by choosing a finer grid. For the fluid under study in this paper, the Reynold's number is chosen to be a relatively moderate value and central finite difference schemes are used for all the terms. However, it is trivial to convert to fully upwind or even mixed schemes.

Choosing a proper scheme for discretisation in time is a similarly complicated issue as the order of accuracy should be comparable to the order of accuracy of space discretisation schemes. However, typical schemes of higher order require either extra computation or extra storage for previous time steps. Furthermore, explicit schemes fail for certain types of PDEs, and these have to be taken into account when choosing a scheme. For Hyperbolic, and by extension Hyperbolic-Parabolic, PDEs like the 2D viscous Burgers' equation, explicit time schemes always lead to an unstable solution irrespective of the size of the time step. This forces us to choose implicit schemes which in turn leads to the entire process of having to solve a system of equations at each time step. For explicit time schemes, this issue does not exist as these schemes use known values at the current time step to extrapolate to a solution at the next time step. A simple implicit Euler, which is alternatively an approach of the BDF family of methods, is used as the first scheme and also for the equations shown in this chapter. This is followed by the more accurate Crank-Nicolson method, which also paves the way for higher-order BDF schemes to be implemented if required. As the time increment Δt is chosen to be sufficiently small, both schemes provide similarly accurate results as the grid size of the space discretisation dominates the order of accuracy.

The matrix equation that is obtained upon discretisation of (i) is of the form:

$$\mathbf{A_1} \times \mathbf{U} - \mathbf{A_2} + \mathbf{U} \odot (\mathbf{A_3} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{U} \times \mathbf{A_4}) - \mathbf{A_5} \times \mathbf{U} - \mathbf{U} \times \mathbf{A_6} = \mathbf{0} - (\boldsymbol{iii})$$

Here $\mathbf{A_1}, \mathbf{A_2}, \mathbf{A_3}, \mathbf{A_4}, \mathbf{A_5}, \mathbf{A_6}$ are the coefficient matrices with known values, **0** is a matrix of zeros and **U** is the unknown matrix corresponding to the discretised nodal values of the 2-dimensional space region at the next time step. The left-hand side of (iii) is also referred to as $F(\mathbf{U})$. It is evident that the mixture of Hadamard products and matrix products in the equation above makes it a non-trivial process to solve the equation while taking advantage of its inherent matrix structure. Therefore, it is necessary to either linearise the equation using domain knowledge or use standardised methods of solving nonlinear equations that define a linearisation step applicable to all equations. For the Burgers' equation the linearisation utilising domain knowledge exists in the form of the Cole-Hopf transformation (Cole, 1951) and its extension to 2-dimensional problems, as explained in Chapter 2. However, as the aim of this paper is to provide a generalised approach to solving nonlinear PDEs with mixed operators, the standard and simple Newton method is chosen. The choice of extending Newton's method from vector-valued functions to matrix-valued functions is validated by (Higham's paper) as detailed in Chapter 2. The Newton's method requires the first-order derivative of the equation(s) it is attempting to solve. In the case of vector valued functions this is represented by the Jacobian matrix. However, if the system of equations is represented as an equation of matrices, computing the first-order derivatives without vectorising the unknowns results in a 4-dimensional tensor. Computing such a tensor is an expensive task irrespective of choosing analytical or Quasi-Newton strategies to obtaining the first-order derivatives. Any operations performed on this tensor are also prohibitively costly, as is storage of the tensor for even moderately sized grids. An alternative is thus sought to efficiently compute and store the first-order derivative of equation (iii) itself, and this exists in the form of the Fréchet derivative. It is important to note, and this is also shown to some extent in Chapter 2, that we do not seek the Fréchet derivative explicitly but instead the term $\mathrm{F}'^{(\mathbf{X})}.\mathbf{H}$, where $F'(\mathbf{X})$ is the Fréchet derivative. For equation (iii), this term is equal to:

$$\mathbf{A_1} \times \mathbf{H} + \mathbf{H} \odot (\mathbf{A_3} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{A_3} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{U} \times \mathbf{A_4}) + \mathbf{U} \odot (\mathbf{H} \times \mathbf{A_4}) - \mathbf{A_5} \times \mathbf{H} - \mathbf{H} \times \mathbf{A_6}$$

While the other matrices are the same as in (iii), **H** represents a matrix consisting of the, potentially infinitesimal, change in **U** analogous to the h used in standard and Fréchet derivative definitions. As is evident, this matrix form upon being equated to a known matrix will present a matrix equation that is linear with respect to the unknown **H**. In the Newton's method, we equate this to $-F(\mathbf{U})$, analogous to $\mathbf{J\delta} = -\mathbf{f}(\mathbf{u})$, thus obtaining:

$$\mathbf{A_1} \times \mathbf{H} + \mathbf{H} \odot (\mathbf{A_3} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{A_3} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{U} \times \mathbf{A_4}) + \mathbf{U} \odot (\mathbf{H} \times \mathbf{A_4}) - \mathbf{A_5} \times \mathbf{H} - \mathbf{H} \times \mathbf{A_6} = -\boldsymbol{F}(\mathbf{U}) \qquad - (\boldsymbol{iv})$$

It is evident that (iv), despite being a linear matrix equation, cannot be manipulated to represent a form similar to the Sylvester equation or even the more generalised linear matrix equation due to the presence of the Hadamard product. A simple and direct solution to this equation involves vectorising both sides. This will lead to an equation of the form:

$$\begin{aligned}\{(\mathbf{I} \otimes \mathbf{A_1}) + (\mathbf{I} \otimes \mathbf{A_5}) + \left(\mathbf{A_6^T} \otimes \mathbf{I}\right) + \operatorname{diag}\left(\operatorname{vec}(\mathbf{A_3} \times \mathbf{U})\right) + \operatorname{diag}\left(\operatorname{vec}(\mathbf{U} \times \mathbf{A_4})\right) \\ + \operatorname{diag}\left(\operatorname{vec}(\mathbf{U})\right) \times (\mathbf{I} \otimes \mathbf{A_3}) + \operatorname{diag}(\operatorname{vec}(\mathbf{U})) \times (\mathbf{A_4^T} \otimes \mathbf{I})\}\operatorname{vec}(\mathbf{H}) \\ = \operatorname{vec}(-\boldsymbol{F}(\mathbf{U})) \qquad - (\boldsymbol{v})\end{aligned}$$

Equation (v) is similar in form to $\mathbf{J\delta} = -\mathbf{f}(\mathbf{u})$, where $\mathbf{J} = \{(\mathbf{I} \otimes \mathbf{A_1}) + (\mathbf{I} \otimes \mathbf{A_5}) + \left(\mathbf{A_6^T} \otimes \mathbf{I}\right) + \operatorname{diag}\left(\operatorname{vec}(\mathbf{A_3} \times \mathbf{U})\right) + \operatorname{diag}\left(\operatorname{vec}(\mathbf{U} \times \mathbf{A_4})\right) + \operatorname{diag}\left(\operatorname{vec}(\mathbf{U})\right) \times (\mathbf{I} \otimes \mathbf{A_3}) + \operatorname{diag}(\operatorname{vec}(\mathbf{U})) \times (\mathbf{A_4^T} \otimes \mathbf{I})\}$, $\delta = vec(\mathbf{H})$ and $-\mathbf{f}(\mathbf{u}) = \boldsymbol{vec}\left(-\boldsymbol{F}(\mathbf{U})\right)$**.** This equation requires computation and storage of multiple Kronecker products and matrix products to solve the equation. This approach becomes increasingly expensive in terms of both time and memory as the matrices grow in size. Thus, there is a requirement for experimental solvers that can utilise the structure of the matrices in the equation. These solvers utilise the formulation in (v) to extend methods that exist for solving vector-valued functions. Two separate approaches are chosen to solve equation (iv). Both are iterative approaches and differ in which terms are shifted to the right-hand side of the equation as part of the iterative step.

The first approach borrows the general idea of splitting from stationary iterative methods, as discussed in Chapter 2, by splitting the large matrix $\mathbf{J}$ in equation (v) and shifting all the terms corresponding to Hadamard products in equation (iv) to the right-hand side of the equation. These terms, in the context of equation (v), will be shown clearly in Chapter 4. It is then assumed that the entire right-hand side of the equation consists of known values at any iteration, similar to the approach for any iterative solver. This new form of equation (v) is reverted back to a matrix equation by inverting the vectorisation applied to (iv). This essentially converts equation (iv) into a Sylvester equation, that is then solved at each iteration. The main challenge in applying this method is obtaining a proper convergence criterion to make informed decisions, and to program an efficient solver for the Sylvester equation that is obtained at each iteration. For this purpose, the standard $sylvester$ function provided by MATLAB is used as a comparison.

The second approach is slightly more involved and utilises the property of multiplying Hadamard product terms with standard basis vectors, as mentioned in Chapter 1. This approach then provides a way of splitting (iv) into a number of equations that can each be manipulated into a linear system of equations, with the vector of unknowns being a particular column of the matrix **H**. The total number of such separate linear system of equations depends on the number of columns present in the matrix **H**. This split is possible by multiplication of **H** with a standard basis vector which provides a particular column vector of **H** itself. There are however certain terms in the equation (iv) in which the matrix $\mathbf{H}$ cannot be multiplied with the basis vectors and thus remains in matrix form. These terms are shifted to the right-hand side and assumed to be of known value at each iteration of the solver. This approach is much more experimental in nature since it does not borrow ideas from any known iterative solver and therefore requires a proper convergence criterion to signify its importance. This approach, as will be shown in Chapter 4, is also highly parallelisable due to the fact that it solves each column of **H** separately. Another challenge, though not observed in this instance, is utilising an efficient linear solve algorithm for the linear system of equations being solved for each column of **H**.

The resulting algorithm can be summarised in the following sequence chart:

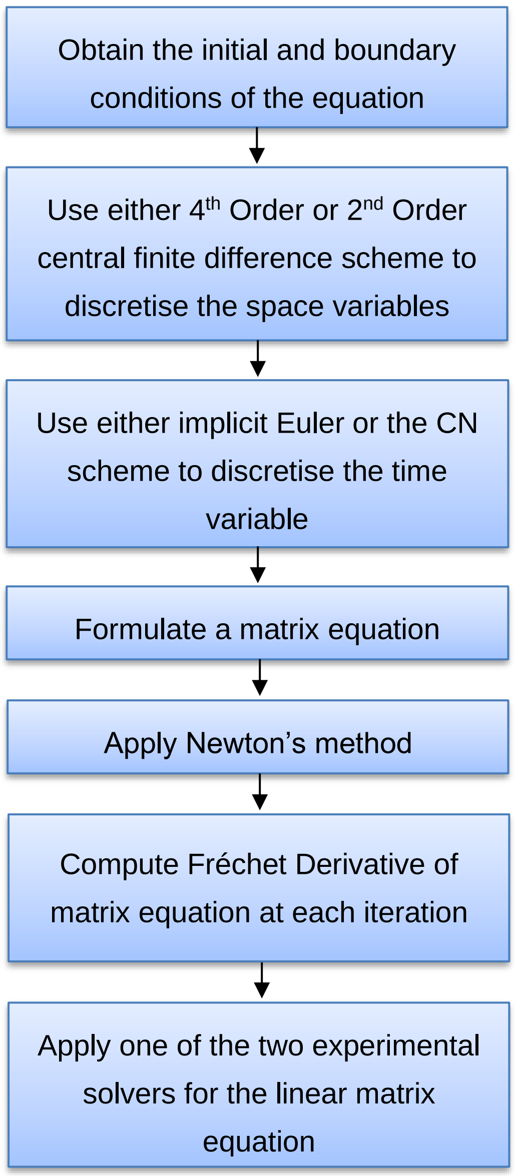
Obtain the initial and boundary conditions of the equation
Use either 4th Order or 2nd Order central finite difference scheme to discretise the space variables
Use either implicit Euler or the CN scheme to discretise the time variable
Formulate a matrix equation
Apply Newton's method
Compute Fréchet Derivative of matrix equation at each iteration
Apply one of the two experimental solvers for the linear matrix equation

Figure 3.2: Algorithm obtained from research methodology to solve the equation

3.3 Software Requirements

There is only one major functional requirement that the software has to deliver. This is the solution of the 2D Burgers' equation, adhering to the algorithm presented in the previous section. This requirement is in turn comprised of a few requirements that represent the different steps of the algorithm itself. Fulfilling them also ensure that this approach can be used as a stepping stone to build solvers for other equations and methods. These requirements are as follows:

Table 3.1: Functional requirements of the exploratory software

Requirement Number	Requirement	Description
1	Coefficient Matrices	Clearly portray the design and structure of the different important coefficient matrices in equation (iv), to allow any modification in the future, or implementation of a different discretisation scheme to the different terms in equation (i).
2	Discretised Equation	Compute the nonlinear matrix equation (iv).
3	Newton's method	Implement the standard Newton's method algorithm.
4	Linear Solvers	Implement each experimental linear solver separately to allow for future improvement and modification.

There are 2 optional requirements that are included in the software if the direct approach to solving equation (iv) is implemented, i.e., to solve equation (vi) in its natural form. These are:

1. Compute the large matrix $\mathbf{J}$ as shown in the previous section, and store it efficiently.
2. Implement a standard solver for a linear system of equations that utilises the structure of $\mathbf{J}$ to efficiently solve the system.

3.4 System Design

The programs to fulfil the requirements will be written in the form of scripts in MATLAB. The choice to program in MATLAB is made as it provides a wide variety of tools to implement problems related to scientific computation efficiently. Moreover, it follows a column ordered architecture for its data structures, which makes it a bit easier to implement matrices intuitively. While the particular software does not require a well-defined architecture to be implemented, it is designed in a step-by-step manner starting from the first functional requirement and moving down the table 3.1. This is somewhat in accordance with the sequential flow of the theoretical algorithm as latter steps of it cannot be implemented without properly implementing the former. Thus, the overall solver is programmed by building a new step on top of a previous step, utilising it in the process.

Chapter 4
Software Implementation

This chapter will delve into the exact matrix equation formed from the discretisation of the equation, the constituent matrices of this equation and standard approaches to computing them, and how they are implemented in the Newton's method. It will then present the two experimental iterative solvers discussed in the previous chapter, the derivations involved in obtaining the linear matrix equation or linear system of equations to solve at each iteration, and the algorithm followed by the solvers. The chosen schemes of discretisation will be the 2nd order Central Finite Difference in space and the 2nd Order Crank-Nicolson scheme in time. Some information about a solver for the conservative form of the equation will be presented at the end of the chapter.

4.1 Discretisation of the 2D Burgers' Equation

As shown in Chapter 3, this paper is attempting to solve the following 2D PDE:

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+u\frac{\partial u}{\partial y}=\mathrm{v}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right);(x,y)\in\Omega\,,\mathrm{t}\ >0\quad-(\mathrm{i})$$

Subject to the following boundary conditions:

$$u(0,y,t)=\frac{1}{1+e^{\frac{Re\times(y-t)}{2}}};x=0,y\in[0,1],\mathrm{t}>0\quad-(\mathrm{ii})$$

$$u(1,y,t)=\frac{1}{1+e^{\frac{Re\times(1+y-t)}{2}}};x=1,y\in[0,1],\mathrm{t}>0-(\mathrm{iii})$$

$$u(x,0,t)=\frac{1}{1+e^{\frac{Re\times(x-t)}{2}}};x\in[0,1],y=0,\mathrm{t}>0\quad-(\mathrm{iv})$$

$$u(x,1,t)=\frac{1}{1+e^{\frac{Re\times(x+1-t)}{2}}};x\in[0,1],y=1,\mathrm{t}>0\quad-(\mathrm{v})$$

And the following initial conditions:

$$u(x,y,0)=\frac{1}{1+e^{\frac{Re\times(x+y)}{2}}};(x,y)\in\Omega,\mathrm{t}=0\qquad-(vi)$$

Here, Re is the Reynold's number which is the inverse of the viscosity coefficient v. Furthermore, as shown in Chapter 3, Ω is the spatial domain within which the equation is being studied, defined as $\Omega=\{(x,y):0\leq x\leq 1,0\leq y\leq 1\}$. The standard form of the 2nd order Central finite difference in space for first-order partial derivative is $\frac{\partial u}{\partial x}\approx\frac{u_{i,j+1}-u_{i,j-1}}{2\times\Delta x}$, and for second-order partial derivative is $\frac{\partial^2 u}{\partial x^2}\approx\frac{u_{i,j+1}-2\times u_{i,j}+u_{i,j-1}}{\Delta x^2}$. Same formulas apply for derivation with respect to y, except we vary the index i instead of j. The 2nd Order Crank-Nicolson scheme is the same as mentioned in Chapter 2.

Applying the 2nd Order accurate Central finite difference scheme in space, and the 2nd Order accurate Crank-Nicolson scheme in time we get, at a time step k, the following discretised nonlinear equation for a particular nodal value $u_{i,j}^{k}$:

$$\begin{aligned}\frac{2\times u_{i,j}^{k+1}}{\Delta t}-\frac{2\times u_{i,j}^{k}}{\Delta t}+\frac{1}{2\times h}\times u_{i,j}^{k+1}\times\left(u_{i+1,j}^{k+1}-u_{i-1,j}^{k+1}\right)+\frac{1}{2\times h}\times u_{i,j}^{k+1}\times\left(u_{i,j+1}^{k+1}-u_{i,j-1}^{k+1}\right)\\-\frac{1}{h^2\times Re}\times\left(u_{i+1,j}^{k+1}-2\times u_{i,j}^{k+1}+u_{i-1,j}^{k+1}\right)\\-\frac{1}{h^2\times Re}\times\left(u_{i,j+1}^{k+1}-2\times u_{i,j}^{k+1}+u_{i,j-1}^{k+1}\right)+\frac{1}{2\times h}\times u_{i,j}^{k}\times\left(u_{i+1,j}^{k}-u_{i-1,j}^{k}\right)\\+\frac{1}{2\times h}\times u_{i,j}^{k}\times\left(u_{i,j+1}^{k}-u_{i,j-1}^{k}\right)-\frac{1}{h^2\times Re}\times\left(u_{i+1,j}^{k}-2\times u_{i,j}^{k}+u_{i-1,j}^{k}\right)\\-\frac{1}{h^2\times Re}\times\left(u_{i,j+1}^{k}-2\times u_{i,j}^{k}+u_{i,j-1}^{k}\right)\\=0\qquad\qquad -(vii)\end{aligned}$$

Here, the unknowns are all the nodal points at the next time step $k+1$. Thus, the nonlinear equation at each node $u_{i,j}^{k}$ is to obtain a value for the unknown $u_{i,j}^{k+1}$, which depends on the 4 unknown neighbouring nodes $u_{i+1,j}^{k+1}, u_{i-1,j}^{k+1}, u_{i,j+1}^{k+1}$ and $u_{i,j-1}^{k+1}$. Additionally, a square grid is chosen such that $\Delta x = \Delta y = h$, and Δt is the time-step value. A sample discretised square grid can thus be represented as:

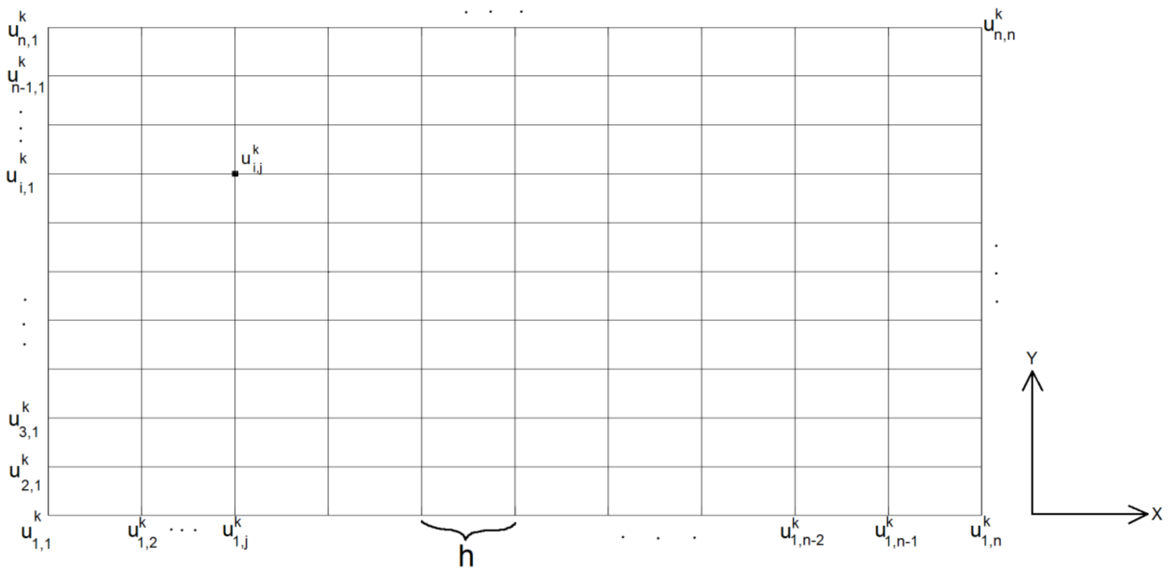

Figure 4.2: Sample discretised square grid of the region under study

The main requirement here is to represent the coefficients that multiply with the unknowns to give the resulting equation at each nodal point, in the form of matrices. In equation (vii) the 1st,3rd,4th,5th and 6th terms have unique coefficients that require separate representation. The 2nd and last 4 terms can be represented by the same coefficient matrices as will be used for the 1st and the 3rd,4th,5th and 6th terms respectively. The only change will be that a matrix of known u values will be used for these terms, instead of the unknown matrix **U**. Representation of coefficients for the 1st term is of trivial nature and thus is computed as part of the solver instead of having a separate script. It is a simple diagonal matrix of the form:

$$T = \begin{bmatrix} \frac{2}{\Delta t} & & & & \\ & \frac{2}{\Delta t} & & & \\ & & \frac{2}{\Delta t} & & \\ & & & \ddots & \\ & & & & \frac{2}{\Delta t} \end{bmatrix}$$

Figure 4.3: Coefficient matrix for the first and second terms of the discretised equation

In the matrix above and for all the following coefficient matrices, empty spaces indicate 0. This is to signify that the matrix is sparse in nature. For the remaining terms, careful consideration is required in terms of the nodes that are utilised. For example, the 3rd term depends on nodes at a distance of $\pm h$ on the y-axis. However, this cannot be possible for boundary nodes and thus it is important to design the matrix accordingly. Even though the nodes at the boundary will have the boundary conditions enforced on them and will not resemble the equation (vii) , the design is done to ensure non-existent nodes are not mentioned in the resultant equation. Furthermore, for the 3rd and 4th terms the coefficient matrix is designed ignoring the term $u_{i,j}^{k+1}$ as this will be multiplied to the term as a standalone matrix using Hadamard product. Thus, the coefficient matrix for the 3rd term is:

$$CH = \begin{bmatrix} 0 & 0 & 0 & & \cdots & & 0 \\ -\frac{1}{2\times h} & 0 & \frac{1}{2\times h} & & & & \vdots \\ & -\frac{1}{2\times h} & 0 & \frac{1}{2\times h} & & & \\ & & \ddots & & \ddots & & \\ & & & & -\frac{1}{2\times h} & 0 & \frac{1}{2\times h} \\ 0 & 0 & 0 & & \cdots & & 0 \end{bmatrix}$$

Figure 4.4: The coefficient matrix of the 3rd term, called Convective Horizontal or CH

In the CH matrix above, the first and last rows are kept 0 to account for nodes at the boundary. When the dependent nodes vary along the x-axis the first and last columns are 0 to account for boundary nodes. This can be set up very efficiently in MATLAB using very few lines of code and MATLAB’s inbuilt support for sparse matrices:

```
i = [2:n-1 2:n-1];
j = [1:n-2 3:n];
v = [repelem(-(1/(2*dy)),n-2) repelem((1/(2*dy)),n-2)];
CH = sparse(i,j,v,n,n);
```

Figure 4.5: Code snippet for script in MATLAB to create the matrix CH

Even though a square grid is used for the problem, the scripts in MATLAB are setup to implement rectangular grids as well with slight modification of the code. Similar but separate scripts exist for the remaining coefficient matrices and they vary based on the number of elements and their positions in the matrix.

For a square grid, the coefficient matrix of the 4th term is simply the transpose of CH. However, for rectangular grids this will not be the case as $\Delta x \neq \Delta y$. The matrix, called Convective Vertical or CV is thus:

$$CV = \begin{bmatrix} 0 & -\frac{1}{2\times h} & & & & 0 \\ 0 & 0 & -\frac{1}{2\times h} & & & 0 \\ 0 & \frac{1}{2\times h} & 0 & \ddots & & 0 \\ \vdots & & \frac{1}{2\times h} & & & \vdots \\ & & & \ddots & -\frac{1}{2\times h} & \\ & & & & 0 & \\ 0 & & & & \frac{1}{2\times h} & 0 \end{bmatrix}$$

Figure 4.6: The coefficient matrix for the 4th term, called CV

The 5th and 6th terms of the equation (vii) represent the diffusive terms of the original equation (i). There are no multiplicative terms to ignore here like in the case of the convective terms, and thus the coefficient matrices represent the entire term. Similar to the convective terms, horizontal and vertical matrices are created to deal with diffusion along x and y-axis separately.

The coefficient matrix of the 5th term, called Diffusion Horizontal, is thus:

$$
\mathrm{DH} = \begin{bmatrix}
0 & 0 & 0 & & \cdots & & 0 \\
-\frac{1}{h^2 \times Re} & \frac{2}{h^2 \times Re} & -\frac{1}{h^2 \times Re} & & & & \vdots \\
 & -\frac{1}{h^2 \times Re} & \frac{2}{h^2 \times Re} & -\frac{1}{h^2 \times Re} & & & \\
 & & \ddots & \ddots & \ddots & & \\
 & & & \ddots & \ddots & \ddots & \\
 & & & & -\frac{1}{h^2 \times Re} & \frac{2}{h^2 \times Re} & -\frac{1}{h^2 \times Re} \\
0 & 0 & 0 & & \cdots & & 0
\end{bmatrix}
$$

Figure 4.7: The coefficient matrix of the 5th term, called DH

Similarly, the Diffusion Vertical matrix to represent the 6th term is:

$$
\mathrm{DV} = \begin{bmatrix}
0 & -\frac{1}{h^2 \times Re} & & & & 0 \\
0 & \frac{2}{h^2 \times Re} & -\frac{1}{h^2 \times Re} & & & 0 \\
0 & -\frac{1}{h^2 \times Re} & \frac{2}{h^2 \times Re} & \ddots & & 0 \\
\vdots & & -\frac{1}{h^2 \times Re} & \ddots & \ddots & \vdots \\
 & & & \ddots & -\frac{1}{h^2 \times Re} & \\
 & & & & \frac{2}{h^2 \times Re} & \\
0 & & & & -\frac{1}{h^2 \times Re} & 0
\end{bmatrix}
$$

Figure 4.8: The coefficient matrix of the 6th term, called DV

Utilising the coefficient matrices, and considering the grid of unknowns at time step $k+1$ is represented by matrix **U** and the grid of known values of u at time step k is represented by matrix $\mathbf{U_{old}}$, the nonlinear matrix equation is:

$$
\mathbf{T} \times \mathbf{U} - \mathbf{T} \times \mathbf{U_{old}} + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{U} + \mathbf{U} \times \mathbf{DV} + \mathbf{U_{old}} \odot (\mathbf{CH} \times \mathbf{U_{old}}) + \mathbf{U_{old}} \odot (\mathbf{U_{old}} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{U_{old}} + \mathbf{U_{old}} \times \mathbf{DV} = \mathbf{0} - (viii)
$$

Here **0** is the matrix of all zeros and it, along with the first 6 terms of the equation correspond to the equation (iii) in Chapter 3, with $\mathbf{A_1} = \mathbf{T}, \mathbf{A_2} = \mathbf{T} \times \mathbf{U_{old}}, \mathbf{A_3} = \mathbf{CH}, \mathbf{A_4} = \mathbf{CV}, -\mathbf{A_5} = \mathbf{DH}, -\mathbf{A_6} = \mathbf{DV}$. The additional terms in the equation can be combined into a single resultant matrix **R** as they do not contain unknown terms. Within the context of this chapter, the matrix **R** will be used to represent this resultant matrix unless stated otherwise. The entire left-hand side of the equation is denoted by $F(\mathbf{U})$.There is a slight but important caveat that has to be taken into account when programming this equation. Since the coefficient matrices are set up such that they consider the entire grid including the boundary nodes, enforcement of the boundary conditions is necessary to ensure accurate results. As the entire $F(\mathbf{U})$ term has to equate to 0, the conditions for the boundary nodes are simply:

$$u_{i,j}^{k+1} - f(x_j, y_i, t_{k+1}) = 0, (x_j, y_i) \in \partial\Omega$$

Where $f(x_j, y_i, t_{k+1})$ is replaced by one of the four equations $(ii), (iii), (iv)$ or (v) based on the values of x and y at the boundary. This ensures that at the boundary nodes $F(\mathbf{U})$ = 0. This is implemented in MATLAB as follows:

```
n = size(U,1);
FU = (2/dt) * U - (2/dt) * Uold + U .* (U * CV) + U .* (CH * U) + U * DV + ...
    DH * U + Uold .* (Uold * CV) + Uold .* (CH * Uold) + Uold * DV + DH * Uold;

%Set boundary values of FU equal to 0(can
%parallelise)
FU(1,1:n) = U(1,1:n)-Unew(1,1:n);
FU(2:n-1,1:n-1:n) = U(2:n-1,1:n-1:n)-Unew(2:n-1,1:n-1:n);
FU(n,1:n) = U(n,1:n)-Unew(n,1:n);
```

Figure 4.9: MATLAB code snippet implementing the nonlinear matrix equation

There is a matrix in the code snippet above that has not been discussed. This is the matrix $\mathbf{U_{new}}$. Also, in MATLAB the '*' operator stands for standard multiplication, and '.*' operator stands for element-wise multiplication, or Hadamard product. This matrix is part of the Newton step during which the term $F(\mathbf{U})$ is calculated and will be discussed in the following section. As mentioned, $F(\mathbf{U})$ has to be equal to 0. However, as a direct approach in general does not exist for solving the nonlinear matrix equation, this condition can only be established at each time step for the boundary nodes. The internal nodes are usually non-zero values that tend towards 0 using the Newton method if the Newton method converges. This occurs usually if the initial guess for the system is sufficiently close to the required root of the system and linear solvers used internally also converge to a solution.

4.2 Application of Newton's Method and Fréchet Derivative

Once the equation $(viii)$ exists, the Newton's method is applied to solve it at each time step. As mentioned in Chapter 2, the Newton's method is an iterative method and runs for a fixed number of iterations, or once convergence is achieved. For a system of nonlinear equations, it is accepted that convergence is guaranteed if for an initial guess $\mathbf{U_0}$ and known solution $\mathbf{U}$ the absolute difference $\parallel \mathbf{U} - \mathbf{U_0} \parallel$ is sufficiently small and the Jacobian matrix $\mathbf{J}$ is invertible (Higham's paper). This is a general observation and actual criteria for convergence are more dependent on the problem itself. However, for the purpose of programming an early-stopping criterion is required at every iteration of the process which indirectly obeys the standard convergence result. Since in general the solution $\mathbf{U}$ is not known, the intuitive approach is checking if the absolute difference ‖ $\mathbf{U_{n+1}} - \mathbf{U_n}$ ‖ is less than some tolerable value. Here $\mathbf{U_{n+1}}$ is the solution computed at iteration n of the Newton's algorithm using an initial guess $\mathbf{U_n}$. This initial guess is denoted by $\mathbf{U_{new}}$, as shown in the previous section. This check can run into stalling issues due to regions where the algorithm is unable to obtain a large enough step towards the root and the check assuming that a solution has been obtained. Thus, the far more robust check of ‖ $\mathbf{F}(\mathbf{U})$ ‖ being less than some tolerable value is accepted. While this may also lead to stalling issues, it would still resemble a solution close to 0. Since a necessary requirement is $F(\mathbf{U}) = 0$, even a stalled iteration satisfying this check will have reached an acceptable solution. To ensure that, for non-convergent problems, the program is not stuck in an infinite loop another necessary condition is to give a maximum number of iterations for the method and check if this number has been exceeded at each iteration. Combining these checks, the general Newton's algorithm for a system of equations is as follows:

```
While (‖ $F(U)$ ‖> $tol_F$ and Iteration ≤ Maximum Iterations) do:
    Solve: $F'(\mathbf{U}) \times \mathbf{H} = -\mathbf{F}(\mathbf{U})$
    Apply: $\mathbf{U_{n+1}} = \mathbf{U_n} + \mathbf{H}$
    Apply: Iteration = Iteration + 1
```

As shown in Chapter 2, for a system of equations with vectorised grid the term $\mathrm{F}'(\mathbf{U}) \times \mathbf{H}$ is generally denoted by $\mathbf{J} \times \boldsymbol{\delta}$ where $\mathbf{J}$ is the Jacobian matrix. However, due to reasons stated in Chapter 3, the Jacobian tensor is not computed and instead the Fréchet Derivative is used. For equation $(viii)$, the Fréchet Derivative is computed as follows:

$$\begin{aligned} F(\mathbf{U}+\mathbf{H}) = {} & \mathbf{T} \times (\mathbf{U}+\mathbf{H}) - \mathbf{T} \times \mathbf{U_{old}} + (\mathbf{U}+\mathbf{H}) \odot \big(\mathbf{CH} \times (\mathbf{U}+\mathbf{H})\big) + (\mathbf{U}+\mathbf{H}) \\ & \odot \big((\mathbf{U}+\mathbf{H}) \times \mathbf{CV}\big) + \mathbf{DH} \times (\mathbf{U}+\mathbf{H}) + (\mathbf{U}+\mathbf{H}) \times \mathbf{DV} + \mathbf{R} - (\boldsymbol{ix}) \end{aligned}$$

As stated previously, $\mathbf{R} = \mathbf{U_{old}} \odot (\mathbf{CH} \times \mathbf{U_{old}}) + \mathbf{U_{old}} \odot (\mathbf{U_{old}} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{U_{old}} + \mathbf{U_{old}} \times \mathbf{DV}$. Upon expanding the terms in equation (ix) and implementing the distributive properties of both standard matrix product and Hadamard product over standard addition, the above equation becomes:

$$\begin{aligned} \Rightarrow F(\mathbf{U}+\mathbf{H}) = {} & \mathbf{T} \times \mathbf{U} - \mathbf{T} \times \mathbf{U_{old}} + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{U} + \mathbf{U} \times \mathbf{DV} + \mathbf{R} \\ & + \mathbf{T} \times \mathbf{H} + \mathbf{H} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{U} \odot (\mathbf{H} \times \mathbf{CV}) \\ & + \mathbf{DH} \times \mathbf{H} + \mathbf{H} \times \mathbf{DV} + \mathbf{H} \odot (\mathbf{CH} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{H} \times \mathbf{CV}) - (\boldsymbol{x}) \end{aligned}$$

The Taylor's series expansion of a general matrix-valued nonlinear function $F(\mathbf{U}+\mathbf{H})$ that is quadratic with respect to the parameter $\mathbf{U}$ and centred around the point $\mathbf{U}$ is:

$$F(\mathbf{U}+\mathbf{H}) = F(\mathbf{U}) + F'(\mathbf{U}) \times \mathbf{H} + F''(\mathbf{U}) \times \mathbf{H}^2$$

Equating the above expansion to equation (ix), we get:

$$F(\mathbf{U}) = \mathbf{T} \times \mathbf{U} - \mathbf{T} \times \mathbf{U_{old}} + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{U} + \mathbf{U} \times \mathbf{DV} + \mathbf{R}$$

$$F'(\mathbf{U}) \times \mathbf{H} = \mathbf{T} \times \mathbf{H} + \mathbf{H} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{U} \odot (\mathbf{H} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{H} + \mathbf{H} \times \mathbf{DV}$$

$$F''(\mathbf{U}) \times \mathbf{H}^2 = \mathbf{H} \odot (\mathbf{CH} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{H} \times \mathbf{CV})$$

As (ix) is equated to the expansion of a quadratic matrix-valued function, this operation is valid as the maximum degree of $\mathbf{U}$ decreases as the order of derivative increases, with the second-order derivative having only $\mathbf{H}$ terms, and $\mathrm{F}'(\mathbf{U}) \times \mathbf{H}$ is obtained as a result. The required linear system of equations to be solved at each Newton iteration is then:

$$\mathbf{T} \times \mathbf{H} + \mathbf{H} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{U} \odot (\mathbf{H} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{H} + \mathbf{H} \times \mathbf{DV} = -\boldsymbol{F}(\mathbf{U}) - (\boldsymbol{x})$$

As mentioned in Chapter 3, two separate but comparable approaches are utilised to solve equation (x).

4.3 Iterative Linear Solve Involving Sylvester equation formulation

The first approach to solving the linear system of equations denoted by $F'(\mathbf{U}) \times \mathbf{H} = -F(\mathbf{U})$ is based on the general splitting approach utilised by stationary iterative methods, as mentioned in Chapters 2 and 3. The derivation of this process is as follows:

1. Represent equation (x) in the form **Ax** = **b**. This is done by vectorising it, which results in the equation:
$\{(\mathbf{I} \otimes \mathbf{T}) + (\mathbf{I} \otimes \mathbf{DH}) + (\mathbf{DV^T} \otimes \mathbf{I}) + diag(vec(\mathbf{CH} \times \mathbf{U})) + diag(vec(\mathbf{U} \times \mathbf{CV})) + diag(vec(\mathbf{U})) \times (\mathbf{I} \otimes \mathbf{CH}) + diag(vec(\mathbf{U})) \times (\mathbf{CV^T} \otimes \mathbf{I})\} vec(\mathbf{H}) = vec(-F(\mathbf{U}))$
2. Split the large matrix, say **A**, into two matrices **D** and **E** such that $\mathbf{D} = (\mathbf{I} \otimes \mathbf{T}) + (\mathbf{I} \otimes \mathbf{DH}) + (\mathbf{DV^T} \otimes \mathbf{I})$ and $\mathbf{E} = diag(vec(\mathbf{CH} \times \mathbf{U})) + diag(vec(\mathbf{U} \times \mathbf{CV})) + diag(vec(\mathbf{U})) \times (\mathbf{I} \otimes \mathbf{CH}) + diag(vec(\mathbf{U})) \times (\mathbf{CV^T} \otimes \mathbf{I})$.
3. Shift the term $\mathbf{E} \times vec(\mathbf{H})$ to the right-hand side.
4. For an iteration k of this iterative linear solver, the left-hand side will consist of the unknown $\mathbf{H_{k+1}}$ and the right-hand side will comprise of the known value of **H** at the current iteration, $\mathbf{H_k}$. Now solve for $\mathbf{H}_{\mathrm{k+1}}$.

It is not required to actually compute the matrix **A**, but its importance is in analysing the method's validity, as shown here, and convergence, as will be shown in Chapter 5. The result of the above split is equivalent to splitting equation (x), at an iteration k, into the following equation:

$$\mathbf{T} \times \mathbf{H_{k+1}} + \mathbf{DH} \times \mathbf{H_{k+1}} + \mathbf{H_{k+1}} \times \mathbf{DV} = -F(\mathbf{U}) - \mathbf{H_k} \odot (\mathbf{CH} \times \mathbf{U}) - \mathbf{U} \odot (\mathbf{CH} \times \mathbf{H_k}) - \mathbf{H_k} \odot (\mathbf{U} \times \mathbf{CV}) - \mathbf{U} \odot (\mathbf{H_k} \times \mathbf{CV})$$

$$\Rightarrow (\mathbf{T} + \mathbf{DH}) \times \mathbf{H_{k+1}} + \mathbf{H_{k+1}} \times \mathbf{DV} = -\boldsymbol{F}(\mathbf{U}) - \mathbf{H_k} \odot (\mathbf{CH} \times \mathbf{U}) - \mathbf{U} \odot (\mathbf{CH} \times \mathbf{H_k}) - \mathbf{H_k} \odot (\mathbf{U} \times \mathbf{CV}) - \mathbf{U} \odot (\mathbf{H_k} \times \mathbf{CV}) \qquad - (\boldsymbol{xi})$$

It is evident that equation (xi) is simply a Sylvester equation with respect to the unknown matrix $\mathbf{H}_{k+1}$. There are two comparative approaches chosen here to solve this equation. Irrespective of the approach chosen, an early-stopping criterion is required to ensure that the algorithm is stopped upon reaching some form of convergence. If all the terms in equation (xi) are taken to one side, then the resulting equation should equate to 0. Thus, at every iteration k, this modified equation is computed with respect to known matrix $\mathbf{H_k}$ and if it is within some tolerable threshold, the solver stops. The other check is same as in Newton's method wherein a maximum number of iterations is used to prevent an infinite loop from occurring.

4.3.1 Inbuilt MATLAB Sylvester function

The inbuilt $sylvester$ function provided by MATLAB is a very powerful and efficient function for small scale matrices. This is mainly due to the effectiveness of the Bartel-Stewart algorithm, and as will be shown in Chapter 5, is faster than parallelisation approaches for small-sized grids. However, as the size of the matrices increases it becomes increasingly infeasible both in terms of computational time and in terms of memory. A prerequisite for using this function is that all the coefficient matrices be provided in their natural form. This means that even if a matrix is sparse in nature, it will have to be passed in as a dense matrix to the $sylvester$ function. This approach loses the benefits of the $sparse$ function inbuilt in MATLAB which provides memory-efficient storage for large, sparse matrices. Thus, the memory requirements increase immensely as the size of the grid increases. Furthermore, the very nature of the Bartel-Stewart algorithm does not allow any scope for parallelisation. This leads to an increase in the computational time.

4.3.2 Shifted linear system solver for Sylvester Equation

The other approach chosen to solve the Sylvester equation involves computing the spectral decomposition of the coefficient matrix **DV** followed by solving a shifted linear system of equations for each column of the projected solution matrix $\widehat{\mathbf{H}_{k+1}}$. The final solution matrix is then obtained by computing $\mathbf{H_{k+1}} = \widehat{\mathbf{H_{k+1}}} \times \mathbf{V^{-1}}$, where the matrix **V** is the matrix of eigenvectors obtained from spectral decomposition of **DV**. This method is chosen due to its ability to be parallelised effortlessly. The solution of the system of linear equations at each column of the projected solution matrix can be computed separately, thus reducing computation time immensely. Moreover, as opposed to the requirement for Schur decomposition of both coefficient matrices in the Bartel-Stewart algorithm, only a single spectral decomposition, of comparable time, is performed. The only step that is potentially expensive and unstable is the computation of $\mathbf{V^{-1}}$. However, MATLAB's inbuilt $mrdivide$ function, represented by the '\' operator, provides a stable way to perform this operation. For small-scale matrices, the overhead from running parallel processes and the computation of the inverse result in this approach being slower than the inbuilt $sylvester$ function. However, for moderate to large scale matrices this is a clear improvement over the standard approach.

4.4 Iterative Linear Solve involving column-wise systems

The other iterative linear solve approach employs the property observed on multiplying Hadamard product of two matrices with $\boldsymbol{e_i}$ and the associative property of both standard and Hadamard matrix multiplication, as mentioned in Chapter 1. The algorithm for this approach is derived as follows:

1. Multiply both sides of equation (x) with $\boldsymbol{e_i}$ from the right. As previously stated, a standard multiplication between any matrix and $\boldsymbol{e_i}$ from the right is equivalent to choosing the i^{th} column of that matrix.
2. The resulting equation will be as follows:

$$(\mathbf{T} \times \mathbf{H} + \mathbf{H} \odot (\mathbf{CH} \times \mathbf{U}) + \mathbf{U} \odot (\mathbf{CH} \times \mathbf{H}) + \mathbf{H} \odot (\mathbf{U} \times \mathbf{CV}) + \mathbf{U} \odot (\mathbf{H} \times \mathbf{CV}) + \mathbf{DH} \times \mathbf{H} + \mathbf{H} \times \mathbf{DV}) \times \boldsymbol{e_i} = -\boldsymbol{F}(\mathbf{U}) \times \boldsymbol{e_i}$$

$$\Rightarrow \mathbf{T} \times \mathbf{H} \times \boldsymbol{e_i} + \mathbf{H} \times \boldsymbol{e_i} \odot (\mathbf{CH} \times \mathbf{U}) \times \boldsymbol{e_i} + \mathbf{U} \times \boldsymbol{e_i} \odot (\mathbf{CH} \times \mathbf{H}) \times \boldsymbol{e_i} + \mathbf{H} \times \boldsymbol{e_i} \odot (\mathbf{U} \times \mathbf{CV}) \times \boldsymbol{e_i} + \mathbf{U} \times \boldsymbol{e_i} \odot (\mathbf{H} \times \mathbf{CV}) \times \boldsymbol{e_i} + \mathbf{DH} \times \mathbf{H} \times \boldsymbol{e_i} + \mathbf{H} \times \mathbf{DV} \times \boldsymbol{e_i} = -\boldsymbol{F}(\mathbf{U}) \times \boldsymbol{e_i}$$

$$\Rightarrow \mathbf{T} \times (\mathbf{H})_\mathbf{i} + (\mathbf{H})_\mathbf{i} \odot (\mathbf{CH} \times \mathbf{U})_\mathbf{i} + (\mathbf{U})_\mathbf{i} \odot (\mathbf{CH} \times \mathbf{H} \times \boldsymbol{e_i}) + (\mathbf{H})_\mathbf{i} \odot (\mathbf{U} \times \mathbf{CV})_\mathbf{i} + (\mathbf{U})_\mathbf{i} \odot (\mathbf{H} \times \mathbf{CV} \times \boldsymbol{e_i}) + \mathbf{DH} \times (\mathbf{H})_\mathbf{i} + \mathbf{H} \times (\mathbf{DV})_\mathbf{i} = \left(-F(\mathbf{U})\right)_\mathbf{i}$$

   Here, for a matrix **A**, $(\mathbf{A})_i$ is used to denote the i^{th} column of that matrix.

$$\Rightarrow \mathbf{T} \times (\mathbf{H})_\mathbf{i} + (\mathbf{H})_\mathbf{i} \odot (\mathbf{CH} \times \mathbf{U})_\mathbf{i} + (\mathbf{U})_\mathbf{i} \odot (\mathbf{CH} \times (\mathbf{H})_\mathbf{i}) + (\mathbf{H})_\mathbf{i} \odot (\mathbf{U} \times \mathbf{CV})_\mathbf{i} + (\mathbf{U})_\mathbf{i} \odot (\mathbf{H} \times (\mathbf{CV})_\mathbf{i}) + \mathbf{DH} \times (\mathbf{H})_\mathbf{i} + \mathbf{H} \times (\mathbf{DV})_\mathbf{i} = \left(-F(\mathbf{U})\right)_\mathbf{i} \qquad - (\boldsymbol{xii})$$

3. As is evident from equation (xii) for a few terms, the i^{th} column of **H** is not obtained. These terms are moved to the right-hand side similar to the approach employed in the previous iterative solver. Then, for an iteration k of the solver, the right-hand side of the equation is a vector obtained by performing the necessary matrix-vector products, and the $\mathbf{H}$ matrix used in these calculations will be a matrix of known values, $\mathbf{H_k}$. The left-hand side will be a matrix-vector product, where the vector of unknowns is the i^{th} column of the matrix of unknowns $\mathbf{H_{k+1}}$, and the matrix will

consist of all the terms that multiply with $(\mathbf{H})_i$ in equation (xii). Thus, the final equation, of the form **Ax=b**, is as follows:

$$\begin{aligned}(\mathbf{T} + diag((\mathbf{CH} \times \mathbf{U})_\mathbf{i}) + diag((\mathbf{U})_\mathbf{i}) \times \mathbf{CH} + diag((\mathbf{U} \times \mathbf{CV})_\mathbf{i}) + \mathbf{DH}) \times (\mathbf{H_{k+1}})_i \\ = \left(-F(\mathbf{U})\right)_\mathbf{i} - (\mathbf{U})_\mathbf{i} \odot (\mathbf{H}_k \times (\mathbf{CV})_\mathbf{i}) - \mathbf{H}_k \times (\mathbf{DV})_\mathbf{i} \qquad -(xiii)\end{aligned}$$

The solver therefore, for a particular outer iteration k, loops through the number of columns of the unknown matrix $\mathbf{H_{k+1}}$ and solves a system of linear equations at each step. This process is easily parallelisable as the different columns can be solved separately, and is thus of comparable cost to the shifted linear system approach to solving the Sylvester equation that was discussed in the last section. Similar to the previous linear solver, early stopping criteria are required for this solver as well. This solver utilises the same criterion for early-stopping as the previous solver, i.e., checking whether for an iteration k, the equation (xi) with all terms stacked on one side and computed using known matrix $\mathbf{H_k}$ lies within some accepted tolerance value. The check to limit the number of iterations to a maximum accepted value is also implemented.

4.5 The Conservative form

Certain considerations are made to accommodate a solver for the conservative form of equation (i), which is represented by the equation (ii) of Chapter 3. These considerations are made in regards to the discretised form of the equation, at a nodal value $u_{i,j}^{k}$:

$$\begin{aligned}\frac{2 \times u_{i,j}^{k+1}}{\Delta t} - \frac{2 \times u_{i,j}^{k}}{\Delta t} + \frac{1}{4 \times h} \times (u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1}) \times \left(u_{i+1,j}^{k+1} - u_{i-1,j}^{k+1}\right) + \frac{1}{4 \times h} \times (u_{i,j+1}^{k+1} \\ + u_{i,j-1}^{k+1}) \times \left(u_{i,j+1}^{k+1} - u_{i,j-1}^{k+1}\right) - \frac{1}{h^2 \times Re} \times \left(u_{i+1,j}^{k+1} - 2 \times u_{i,j}^{k+1} + u_{i-1,j}^{k+1}\right) \\ - \frac{1}{h^2 \times Re} \times \left(u_{i,j+1}^{k+1} - 2 \times u_{i,j}^{k+1} + u_{i,j-1}^{k+1}\right) + \frac{1}{4 \times h} \times (u_{i+1,j}^{k+1} \\ + u_{i-1,j}^{k+1}) \times \left(u_{i+1,j}^{k+1} - u_{i-1,j}^{k+1}\right) + \frac{1}{4 \times h} \times (u_{i,j+1}^{k+1} + u_{i,j-1}^{k+1}) \times \left(u_{i,j+1}^{k+1} - u_{i,j-1}^{k+1}\right) \\ - \frac{1}{h^2 \times Re} \times \left(u_{i+1,j}^{k} - 2 \times u_{i,j}^{k} + u_{i-1,j}^{k}\right) - \frac{1}{h^2 \times Re} \times \left(u_{i,j+1}^{k} - 2 \times u_{i,j}^{k} + u_{i,j-1}^{k}\right) \\ = 0\end{aligned}$$

The changes are with regards to the conservative terms and as is observed, two new coefficient matrices are required to accommodate the new terms. Furthermore, the scalar values stored in the original convective matrices **CH** and **CV** are updated to reflect the new values. The new coefficient matrices are in essence similar to their standard **CH** and **CV** counterparts but, as is evident from the above equation, do not store any negative coefficients. In the MATLAB scripts, these two coefficient matrices are called Conservative Convective Horizontal (**CCH**) and Conservative Convective Vertical (**CCV**).

The resulting nonlinear matrix equation $(viii)$ is also slightly altered:

$$\begin{aligned}\mathbf{T} \times \mathbf{U} - \mathbf{T} \times \mathbf{U_{old}} + (\mathbf{CCH} \times \mathbf{U}) \odot (\mathbf{CH'} \times \mathbf{U}) + (\mathbf{U} \times \mathbf{CCV}) \odot (\mathbf{U} \times \mathbf{CV'}) + \mathbf{DH} \times \mathbf{U} + \mathbf{U} \times \mathbf{DV} \\ + (\mathbf{CCH} \times \mathbf{U_{old}}) \odot (\mathbf{CH'} \times \mathbf{U_{old}}) + (\mathbf{U_{old}} \times \mathbf{CCV}) \odot (\mathbf{U_{old}} \times \mathbf{CV'}) + \mathbf{DH} \times \mathbf{U_{old}} \\ + \mathbf{U_{old}} \times \mathbf{DV} = \mathbf{0}\end{aligned}$$

Where $\mathbf{CH}'$ and $\mathbf{CV}'$ represent the slight modification in the original matrices **CH** and **CV** to account for the modified scalar value. The entire approach to solving this equation does not change, and the same linear solvers can be used to solve the new linear equations that are obtained from computing the Fréchet Derivative for this equation.

2.1 Tables using the ‘table caption’ and ‘table description’ Styles

Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table. Text before table.

Table 2.1 Caption of Table — automatically appears in the List of Tables when that is updated The 'table caption' style has been applied to this paragraph by pressing Ctrl Shift T.

This is the table description in the 'table description' style. It is optional text to give more information about the table and does not appear in the List of Tables.

Heading One	Heading Two	Heading Three
1.1	1.2	1.3
1.21	1.22	12.3
12.31	12.32	12.33

2.2 Figures using the 'figure caption' and 'figure description' Styles

Figures can be added using the Illustrations section of the Insert tab.

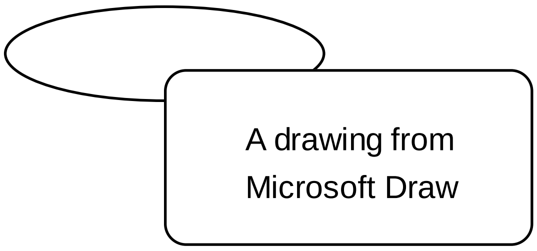

Figure 2.1 Caption of Figure — automatically appears in the List of Figures when that is updated. The 'figure caption' style has been applied to this paragraph by pressing Ctrl Shift F.

This is the figure description in the 'figure description' style. It is optional text to give more information about the figure and does not appear in the List of Figures.

List of References

<It is expected that the list would reflect the breadth and depth of scholarly research undertaken by the student during the course of the project.>

Appendix A
External Materials

<Level 1 Heading with ‘heading 1’ Style Applied by Pressing Ctrl Shift 1> Text under appendix heading. Text under appendix heading. Text under appendix heading. Text under appendix heading. Text under appendix heading. Text under appendix heading.

A.1 Level 2 Heading with ‘heading 2’ Style Applied by Pressing Ctrl Shift 2

Text under level 2 heading. Text under level 2 heading. Text under level 2 heading. Text under level 2 heading.

A.1.1 Level 3 Heading with ‘heading 3’ Style Applied by Pressing Ctrl Shift 3

Text under level 3 heading. Text under level 3 heading. Text under level 3 heading. Text under level 3 heading.

A.1.1.1 Level 4 Heading with ‘heading 4’ Style Applied by Pressing Ctrl Shift 4

Text under level 4 heading. Text under level 4 heading. Text under level 4 heading. Text under level 4 heading.

Appendix B
Ethical Issues Addressed

Text under appendix heading. Text under appendix heading. Text under appendix heading. Text under appendix heading. Text under appendix heading. Text under appendix heading.

B.1 Level 2 Heading

Text under level 2 heading. Text under level 2 heading. Text under level 2 heading. Text under level 2 heading.